Transwarp Data Hub Version 9.3
Inceptor 使用手册

2023-02-28

目录

1. 前言 2
1.1. 读者对象 2
1.2. 产品版本 2
1.3. 修订记录 2
1.4. 文档阅读约定 2
1.4.1. 符号约定 2
1.4.2. 命令行格式约定 3
1.5. 获取和更新文档 3
1.6. 意见反馈 3
2. 产品简介 4
2.1. 产品定位 4
2.2. 产品优势 4
2.3. 软件架构 5
2.3.1. 产品定位 5
2.3.2. 核心组件 6
2.4. 管理组件 7
3. 连接数据库 8
3.1. 连接方式 8
3.1.1. 准备工作 8
3.1.1.1. 确认安全认证方式 8
3.1.1.2. 获取连接地址 9
3.2. 通过 Beeline 命令行连接 10
3.2.1. 操作步骤 10
3.2.2. 更多认证方式 12
3.2.3. HA（高可用）连接方式 12
3.3. 通过 Waterdrop 连接 12
3.3.1. 环境要求 12
3.3.2. 操作步骤 12
4. 开发规范 16
4.1. 概述 16
4.1.1. 数据库设计规范 16
4.1.1.1. 对象命名 16
4.1.1.2. 表结构设计 17
4.1.1.3. 字段设计 18
4.1.2. SQL 设计规范 18
4.1.2.1. DDL 18
4.1.2.2. DML 19
4.1.2.3. DQL 19
4.1.3. 应用程序开发规范 19
4.1.4. 推荐规范 20
4.1.4.1. 连接管理 20
4.1.4.2. 类/方法使用 20

5. 数据写入 21
5.1. 实时写入 22
5.1.1. 实时写入示例 22
5.1.1.1. 非安全模式下实时写入示例 23
5.1.1.2. 安全模式下实时写入案例 24
5.1.1.3. Kafka\Event Store、Slipstream 功能介绍 28
5.2. 批量写入 31
6. 应用程序开发 32
6.1. 概述 32
6.2. 基于 JDBC 开发 32
6.2.1. 下载 JDBC 驱动 32
6.2.1.1. 操作步骤 32
6.2.2. 常用 API 介绍. 33
6.2.3. 开发流程 33
6.2.4. 异常处理 35
6.2.5. 示例代码 35
6.2.5.1. 与免认证的 Inceptor 交互 36
6.2.5.2. 与启用 Kerberos 认证的 Inceptor 交互 36
6.2.5.3. 与启用 LDAP 认证的 Inceptor 交互 37
6.2.6. JDBC 接口参考. 38
6.2.6.1. java.sql.Connection 38
6.2.6.2. java.sql.CallableStatement 39
6.2.6.3. java.sql.DatabaseMetaData 40
6.2.6.4. java.sql.Driver 41
6.2.6.5. java.sql.PreparedStatement 42
6.2.6.6. java.sql.ResultSet 42
6.2.6.7. java.sql.ResultSetMetaData 43
6.2.6.8. java.sql.Statement 44
6.2.6.9. javax.sql.ConnectionPoolDataSource 44
6.2.6.10. javax.sql.DataSource 44
6.2.6.11. javax.sql.PooledConnection 45
6.3. 基于 ODBC 开发 45
6.3.1. 环境准备 46
6.3.1.1. 下载驱动 46
6.3.1.2. Windows 平台配置数据源 46
6.3.1.3. Linux 平台配置数据源 49
6.3.2. 开发流程 51
6.3.3. 示例代码 52
6.3.3.1. C 示例代码 52
6.3.3.2. C# 示例代码 55
6.3.3.3. Python 示例代码 56
6.3.3.4. Perl 示例代码 57
6.3.3.5. Ruby 示例代码 58
6.3.3.6. PHP 示例代码 59
6.3.4. ODBC API 支持. 59

6.3.4.1. 连接数据源 60
6.3.4.2. 获取驱动/数据源信息 60
6.3.4.3. 设置/获取驱动属性 60
6.3.4.4. 设置/获取描述符字段 61
6.3.4.5. 准备 SQL 请求 61
6.3.4.6. 提交请求 61
6.3.4.7. 获取结果及其相关信息 62
6.3.4.8. 获取数据源的系统表信息 62
6.3.4.9. 终止语句 63
6.3.4.10. 终止连接 63
6.3.5. API 映射关系说明 63
6.3.6. Catelog API 支持说明. 64
7. 资源与权限管理 66
7.1. 概述 66
7.1.1. 多租户平台架构 66
7.1.2. 使用流程 66
7.2. 启用租户管理功能 67
7.2.1. 操作步骤 67
7.2.2. 扩展阅读 67
7.3. 规划并创建租户 67
7.3.1. 概念介绍 68
7.3.2. 规划租户 68
7.3.3. 创建租户 69
7.3.3.1. 操作步骤 69
7.4. 通过页面管理资源/权限 71
7.4.1. 创建角色 72
7.4.2. 授予库/表/列权限 72
7.4.2.1. 准备工作 72
7.4.2.2. 操作步骤 73
7.4.2.3. 扩展阅读 74
7.4.3. 授予存储过程权限 74
7.4.3.1. 准备工作 74
7.4.3.2. 操作步骤 75
7.4.4. 分配队列计算资源 77
7.4.4.1. 准备工作 77
7.4.4.2. 操作步骤 78
7.4.5. 授予队列权限 80
7.4.5.1. 操作步骤 80
7.4.6. 分配存储配额 82
7.4.6.1. 准备工作 82
7.4.6.2. 操作步骤 82
7.4.6.3. 扩展阅读 84
7.4.7. 授予 HDFS 权限 84
7.4.7.1. 准备工作 84
7.4.7.2. 身份判断说明 84

7.4.7.3. 权限判断说明 84
7.4.7.4. 操作步骤 85
7.4.7.5. 扩展阅读 86
7.5. 通过 SQL 管理权限 86
7.5.1. 管理角色与权限 86
7.5.1.1. 判断用户身份 86
7.5.1.2. 获取管理员身份 86
7.5.1.3. 管理角色 87
7.5.1.4. 管理权限 92
7.5.1.5. 操作所需的权限总结 99
7.5.2. 管理行/列级权限 100
7.5.2.1. 行/列级与表级权限说明 101
7.5.2.2. SQL 语法及使用示例 101
7.5.2.3. 设置行级权限 101
7.5.2.4. 撤销行级权限设置 104
7.5.2.5. 设置列级权限 104
7.5.2.6. 撤销列级权限设置 106
7.5.2.7. 查看行/列级权限 106
7.5.3. 空间配额管理操作. 107
7.5.3.1. SQL 语法介绍 107
7.5.3.2. 示例操作 108
7.5.4. 管理 HDFS 读写权限. 110
7.5.4.1. 设置用户/组对某张表的FACL 111
7.5.4.2. 取消用户/组对某张表的FACL 111
7.5.4.3. 取消用户/组对某张表的FACL所需权限 111
7.5.4.4. 取消用户组bob对表alice_t1的FACL 111
7.5.4.5. 取消某张表上的全部FACL 111
7.5.4.6. 查看用户/组对某张表的FACL 112
7.5.4.7. 查看某张表上的所有FACL 112
8. SQL 参考 114
8.1. 数据定义语言（DDL） 115
8.1.1. 定义数据库. 116
8.1.1.1. CREATE DATABASE 116
8.1.1.2. ALTER DATABASE 117
8.1.1.3. DROP DATABASE 118
8.1.1.4. USE DATABASE 119
8.1.1.5. SHOW DATABASES 119
8.1.1.6. DESCRIBE DATABASE 120
8.1.2. 定义表. 121
8.1.2.1. CREATE TABLE 121
8.1.2.2. ALTER TABLE 130
8.1.2.3. DROP TABLE 138
8.1.2.4. TRUNCATE TABLE 138
8.1.2.5. SHOW TABLES 139
8.1.2.6. DESCRIBE TBALE 140

8.1.3. 定义分区. 141
8.1.3.1. ADD PARTIOTION 141
8.1.3.2. TRUNCATE TABLE PARTIOTION 142
8.1.3.3. DROP PARTIOTION 143
8.1.3.4. SHOW PARTITIONS 144
8.1.4. 定义视图. 145
8.1.4.1. CREATE VIEW 145
8.1.4.2. ATLER VIEW 146
8.1.4.3. DROP VIEW 147
8.1.4.4. 物化视图 148
8.1.4.5. SHOW VIEWS 151
8.1.4.6. DESC VIEW 151
8.2. 数据操作语言(DML) 152
8.2.1. INSERT. 152
8.2.1.1. INSERT INTO|OVERWRITE VALUES 152
8.2.1.2. 批量插入多条数据 153
8.2.1.3. 子查询插入数据 154
8.2.1.4. 插入数据至分区表 155
8.2.2. UPDATE. 157
8.2.3. MERGE INTO. 158
8.2.4. DELETE. 160
8.3. 数据查询语句（DQL） 161
8.3.1. 表查询 SELECT FROM. 161
8.3.2. 表排序. 165
8.3.3. 表连接 JOIN ON. 166
8.3.3.1. 自然连接 NATURAL JOIN 166
8.3.3.2. 内连接 [INNER|CROSS] JOIN 167
8.3.3.3. 外连接 LEFT|RIGHT|FULL OUTER JOIN 169
8.3.3.4. 左半连接 LEFT SEMI JOIN 170
8.3.3.5. 不等价连接 171
8.3.4. 集合计算. 171
8.3.4.1. 差集连接 EXCEPT 172
8.3.4.2. 交集连接 INTERSECT 173
8.3.4.3. 并集连接 UNION 174
8.3.5. 层次化查询 CONNECT BY 175
8.4. 冷热数据分离. 180
8.4.1. 指定表的存储介质. 180
8.4.2. 设置存储介质的属性. 180
8.5. Compact/小文件合并. 181
8.5.1. 手动合并小文件. 181
8.5.2. 自动合并小文件. 182
8.5.3. 管理Compact任务 183
8.6. 事务控制. 184
8.6.1. 手动提交事务. 184
8.6.2. 设置事务级别. 185

8.6.3. 管理事务. 185
8.7. ROWKEY. 187
8.7.1. ROWKEY 介绍 187
8.7.2. 使用 Rowkey 188
8.7.3. ROWKEY 的 UPSERT 功能 189
8.8. 动态分区. 191
8.8.1. 动态创建分区. 191
8.8.2. 查看分区情况. 192
8.8.3. 动态插入数据. 192
8.8.4. 使用案例. 193
8.9. 函数与运算符. 193
8.9.1. Inceptor函数和运算符入门. 193
8.9.2. Inceptor函数和运算符指南中的表. 195
8.9.3. 关系运算符. 197
8.9.4. 算术运算符. 201
8.9.5. 逻辑运算符. 205
8.9.6. 数学函数. 207
8.9.6.1. 简单求值 207
8.9.6.2. 幂/方根/对数 214
8.9.6.3. 进制/度量转换 216
8.9.6.4. 三角函数 218
8.9.6.5. 线性回归函数 224
8.9.7. 类型转换函数. 228
8.9.7.1. BINARY(<string>\|<binary>) 228
8.9.7.2. CAST 228
8.9.7.3. DATE 229
8.9.7.4. decimal(expression) 230
8.9.8. 日期函数. 231
8.9.8.1. ADD_MONTHS(<date>,<num_months>) 231
8.9.8.2. EXTRACT(DAY\|MONTH\|YEAR FROM <date>) 232
8.9.8.3. TO_UNIX_TIMESTAMP(<date>[, <pattern>]) 232
8.9.8.4. UNIX_TIMESTAMP([<date>[, <pattern>]]) 或 TO_TIMESTAMP([<date>[, <pattern>]])233
8.9.8.5. FROM_UNIXTIME(<unix_time>[, <format>]) 233
8.9.8.6. TO_CHAR(<date>[, <pattern>]) 234
8.9.8.7. TO_DATE(<date>[, <pattern>]) 234
8.9.8.8. TDH_TODATE(<date>[, <originalformat>,<targetformat>]) 235
8.9.8.9. DAY(<date>) 或 DAYOFMONTH(<date>) 235
8.9.8.10. DAYOFYEAR(<date>[,<format>]) 236
8.9.8.11. year(<date>) 236
8.9.8.12. QUARTER(<date>[,<format>]) 237
8.9.8.13. month(<date>) 237
8.9.8.14. HOUR(<date>) 238
8.9.8.15. MINUTE(<date>) 238
8.9.8.16. WEEKOFYEAR 239
8.9.8.17. DATEDIFF 239

8.9.8.18. DATE_ADD(<start_date>, <num_days>) 240
8.9.8.19. DATE_SUB(<start_date>, <num_days>) 241
8.9.8.20. DATE_FORMAT(<date>, <pattern>) 241
8.9.8.21. STR_TO_DATE(<date>, <format>) 242
8.9.8.22. SYSDATE 242
8.9.8.23. DAYOFWEEK(<date>) 242
8.9.8.24. DAYOFWEEK_ISO(<date>) 243
8.9.8.25. MILLISECOND(<date>) 243
8.9.8.26. MONTHS_BETWEEN 244
8.9.8.27. LAST_DAY(<date>) 244
8.9.8.28. DAYS(<date>) 244
8.9.8.29. TO_DAY_INTERVAL(<number>) 245
8.9.8.30. TO_MONTH_INTERVAL(<number>) 245
8.9.8.31. TO_YEAR_INTERVAL(<number>) 245
8.9.8.32. TO_HOUR_INTERVAL(<number>) 245
8.9.8.33. TO_MINUTE_INTERVAL(<number>) 246
8.9.8.34. TO_SECOND_INTERVAL(<number>) 246
8.9.8.35. TD_INT_TO_TIME(<HHmmss>) 246
8.9.8.36. systime() 246
8.9.8.37. SYSTIMESTAMP 247
8.9.8.38. TRUNC 247
8.9.8.39. NEXT_DAY(<date>, '<dayofweek>') 248
8.9.8.40. FROM_UTC_TIMESTAMP 248
8.9.8.41. TO_UTC_TIMESTAMP 248
8.9.8.42. numtoyminterval(interval_number, year\|month) 249
8.9.8.43. numtodsinterval(interval_number, day\|hour\|minute\|second) 249
8.9.8.44. timestampdiff(unit, start_timestamp, end_timestamp 249
8.9.9. 条件函数. 250
8.9.10. 字符串函数 251
8.9.10.1. 数学求值 251
8.9.10.2. 内容选取 254
8.9.10.3. 类型/形式转换 264
8.9.10.4. 加密解密 270
8.9.10.5. 正则表达式相关 272
8.9.10.6. 位置相关 274
8.9.10.7. 条件判断 277
8.9.11. XML函数. 278
8.9.12. 复杂类型函数 283
8.9.13. 表生成函数 288
8.9.13.1. LATERRAL VIEW函数 291
8.9.14. 聚合/分析函数. 293
8.9.14.1. avg 293
8.9.14.2. collect_set 294
8.9.14.3. compute_stats 294
8.9.14.4. count 295

8.9.14.5. covar_pop 295
8.9.14.6. covar_samp 295
8.9.14.7. corr 296
8.9.14.8. ewah_bitmap 296
8.9.14.9. histogram_numeric 296
8.9.14.10. kurtosis 297
8.9.14.11. max 297
8.9.14.12. min 298
8.9.14.13. percentile 298
8.9.14.14. percentile 298
8.9.14.15. percentile_approx 299
8.9.14.16. percentile_cont 299
8.9.14.17. percentile_disc 300
8.9.14.18. skew 301
8.9.14.19. std 302
8.9.14.20. stddev/stddev_pop 302
8.9.14.21. stddev_samp 302
8.9.14.22. sum 303
8.9.14.23. variance/var_pop 303
8.9.14.24. var_samp 304
8.9.14.25. wd_concat 304
8.9.14.26. xmlagg 304
8.9.15. 窗口函数 305
8.9.15.1. OVER子句 305
8.9.15.2. DISTINCT与窗口函数 308
8.9.15.3. 窗口函数详解 312
8.9.16. Context函数. 322
8.9.16.1. current_user 323
8.9.16.2. logged_in_user 324
8.9.16.3. has_role 324
8.9.16.4. current_database 325
8.9.16.5. current_time 325
8.9.16.6. current_date 326
8.9.16.7. current_timestamp 326
8.9.17. 脱敏相关函数 326
8.9.18. 其他函数和运算符 329
8.9.18.1. oracle_coalesce 329
8.9.18.2. oracle_decode 330
8.9.18.3. to_card_15_to_18 (<15_digit_int>) 330
8.9.18.4. zeroifnull (<expr>) 330
8.9.18.5. nullif(expr1, expr2) 331
8.9.18.6. nullifzero (<expr>) 331
8.9.18.7. reflect 332
8.9.18.8. java_method 332
8.9.18.9. reflect2 332

8.9.18.10. hash . . . 333
8.9.18.11. assert_true-def . . . 333
8.9.18.12. uniq() . . . 333
8.9.18.13. grouping (<column>) . . . 333
8.9.18.14. grouping_id . . . 334
8.9.18.15. range_decode . . . 335
8.9.18.16. uuid() . . . 336
8.9.18.17. replicate_rows . . . 336
8.10. 空串和 NULL . . . 337
8.10.1. 字符串类型区别 . . . 337
8.10.2. Inceptor 版本兼容 . . . 337
8.10.3. Oracle 兼容模式 . . . 338
8.10.3.1. 自动补充空格 . . . 338
8.10.3.2. 字符串类型转化 . . . 339
8.10.4. DB2 兼容模式 . . . 339
8.10.5. TD 兼容模式 . . . 340
8.11. PL/SQL 指南（Oracle 方言） . . . 340
8.11.1. Inceptor PL/SQL入门 . . . 340
8.11.2. Inceptor PL/SQL指南中的表 . . . 342
8.11.3. 基础知识 . . . 345
8.11.3.1. 声明 . . . 345
8.11.3.2. 命名规则 . . . 348
8.11.3.3. 赋值 . . . 351
8.11.4. 数据类型 . . . 353
8.11.4.1. 标量类型 . . . 353
8.11.4.2. %TYPE 属性 . . . 355
8.11.4.3. %ROWTYPE 属性 . . . 356
8.11.4.4. 复合类型 . . . 357
8.11.5. 创建 PL/SQL 语句块 . . . 357
8.11.5.1. PL/SQL 语句块组成部分 . . . 357
8.11.5.2. 最简单的匿名块 . . . 358
8.11.5.3. 稍复杂的匿名块 . . . 358
8.11.5.4. 完整的匿名块 . . . 358
8.11.5.5. 匿名块里的嵌套 . . . 359
8.11.6. 流程控制语句 . . . 360
8.11.6.1. IF . . . 361
8.11.6.2. LOOP . . . 364
8.11.6.3. WHILE . . . 365
8.11.6.4. FOR . . . 366
8.11.6.5. FORALL . . . 368
8.11.6.6. EXIT WHEN . . . 370
8.11.6.7. CONTINUE（WHEN） . . . 371
8.11.6.8. GOTO . . . 372
8.11.6.9. CASE . . . 374
8.11.7. PL/SQL 存储过程 . . . 375

8.11.7.1. 不带参数的过程 376
8.11.7.2. 带参数的过程 376
8.11.7.3. 过程重载 387
8.11.7.4. 系统预定义过程 389
8.11.8. PL/SQL 函数. 389
8.11.8.1. 创建函数 389
8.11.8.2. 查看函数 390
8.11.8.3. 调用函数 390
8.11.8.4. 函数重载 392
8.11.8.5. 系统预定义函数 394
8.11.9. Records. 394
8.11.9.1. Records 语法 394
8.11.9.2. Records 创建及使用 395
8.11.10. Collections 397
8.11.10.1. VARRAY 397
8.11.10.2. NESTED TABLE 399
8.11.10.3. Associative arrays 401
8.11.10.4. Collections 方法 402
8.11.11. 游标（Cursors） 418
8.11.11.1. 显式游标 418
8.11.11.2. 隐式游标 425
8.11.11.3. 游标变量 426
8.11.12. 与 SQL 的交互 429
8.11.12.1. PL/SQL 过程与 SQL 的交互 429
8.11.12.2. PL/SQL 函数与 SQL 的交互 433
8.11.12.3. 隐式游标 434
8.11.12.4. BULK COLLECT 434
8.11.12.5. 动态 SQL 436
8.11.13. Packages. 449
8.11.13.1. 创建 Packages 449
8.11.13.2. Packages 使用案例 451
8.11.13.3. 系统预定义包 453
8.11.13.4. 预定义函数/过程/包 453
8.11.14. 异常. 462
8.11.14.1. 支持的系统预定义异常 462
8.11.14.2. 暂不支持的系统预定义异常 474
8.11.14.3. 用户自定义异常 482
8.11.14.4. 嵌套异常 483
8.11.15. 注意事项. 489
8.11.15.1. 函数/存储过程的版本兼容 489
8.11.15.2. Inceptor对PL/SQL中分号的支持 489
8.11.15.3. PL/SQL 中结果的打印 490
8.11.15.4. 标识符 491
8.11.15.5. 标准 SQL 调用 PL/SQL 函数必须满足的条件 495
8.11.15.6. 不支持 RETURN INTO 语句 498

8.11.15.7. 查看预定义函数/过程/包 498
8.11.15.8. PL/SQL 中的 PUT_LINE 打印 505
8.11.15.9. Hive 异常的处理 508
8.11.15.10. Debug 510
8.12. SQL PL 指南（DB2 方言） 513
8.12.1. Inceptor SQL PL入门. 513
8.12.1.1. 输出 513
8.12.1.2. 声明 513
8.12.1.3. 赋值 513
8.12.1.4. 过程 514
8.12.1.5. 函数 514
8.12.1.6. 游标 514
8.12.1.7. 异常 515
8.12.1.8. 动态 SQL 515
8.12.2. Inceptor SQL PL 指南中的表 515
8.12.3. 基础知识 518
8.12.3.1. 快速入门 518
8.12.3.2. 声明 519
8.12.3.3. 赋值 523
8.12.4. 数据类型 525
8.12.4.1. 内置类型 525
8.12.4.2. 复合类型 529
8.12.4.3. 锚定类型 529
8.12.4.4. 特殊类型 529
8.12.5. 创建 SQL PL 语句块 530
8.12.5.1. 创建示例 530
8.12.6. 流程控制语句 530
8.12.6.1. IF 530
8.12.6.2. FOR 534
8.12.6.3. LOOP 534
8.12.6.4. WHILE 535
8.12.6.5. REPEAT 536
8.12.6.6. GOTO 536
8.12.6.7. LEAVE 537
8.12.6.8. ITERATE 538
8.12.6.9. CASE 539
8.12.7. 存储过程 539
8.12.7.1. 语法 539
8.12.7.2. 参数的选择 540
8.12.7.3. 属性 545
8.12.7.4. 过程重载 546
8.12.8. SQLPL 函数 547
8.12.8.1. 创建函数 547
8.12.8.2. 查看函数 547
8.12.8.3. 参数选择 548

8.12.8.4. 函数属性 550
8.12.8.5. 调用函数 550
8.12.8.6. 标准 SQL 调用 SQLPL 函数必须满足的条件 553
8.12.8.7. 函数重载 555
8.12.9. VARRAY 556
8.12.9.1. 定义数组类型及变量 556
8.12.9.2. 数组变量的赋值 557
8.12.9.3. 数组变量的方法调用 557
8.12.10. 行数据类型. 560
8.12.10.1. 语法 560
8.12.10.2. 声明与赋值 561
8.12.10.3. 基于表的行数据类型的声明 564
8.12.11. 游标. 564
8.12.11.1. 组成部分 564
8.12.11.2. 基本用法 565
8.12.11.3. ALLOCATE CURSOR 565
8.12.12. 与 SQL 的交互 567
8.12.12.1. 将 SELECT 返回值赋值给变量 567
8.12.12.2. SELECT INTO 567
8.12.12.3. 存储过程与 SQL 的交互 568
8.12.12.4. 函数与 SQL 的交互 569
8.12.12.5. 游标 569
8.12.12.6. 动态 SQL 570
8.12.12.7. GET DIAGNOSTICS 582
8.12.13. 系统预定义函数/过程 584
8.12.13.1. 查看预定义函数/过程 584
8.12.13.2. 预定义函数/过程/包的介绍 590
8.12.13.3. 预定义函数/过程的使用 591
8.12.14. 异常. 595
8.12.14.1. SQLCODE与SQLSTATE 595
8.12.14.2. SIGNAL 597
8.12.14.3. 异常分类 598
8.12.14.4. RESIGNAL 604
8.12.14.5. DEBUG 605
8.12.15. 注意事项. 607
8.12.15.1. 函数/存储过程的版本兼容 607
8.12.15.2. 方言的选择 607
8.12.15.3. 分号的支持 608
8.12.15.4. PL/SQL 中结果的打印 608
9. 附录 609
9.1. 字段规范 609
9.2. 关键字 610
9.2.1. 基本概念. 610
9.2.2. 非保留关键词的受限范围. 610
9.2.3. 关键词列表. 611

9.2.3.1. 保留关键词 611
9.2.3.2. 非保留关键词 617
9.3. 表可存储格式 625
9.3.1. TEXTFILE. 626
9.3.2. ORCFILE/ORCTRANSACTIONFILE. 627
9.3.3. CSVFILE 628
9.3.4. EXCEL/JSON. 629
9.3.5. FWCFILE 630
9.3.6. SEQUENCEFILE. 631
9.3.7. RCFILE. 632
9.3.8. HOLODESK. 633
9.3.9. PARQUET 633
9.3.10. HYPERDRIVE 634
9.3.10.1. Transwarp Hyperbase的典型使用场景 634
9.3.10.2. 创建HYPERDRIVE表 634
9.3.11. ESDRIVE. 636
9.3.11.1. Esdrive典型使用场景 636
9.3.11.2. 创建ES表 636
9.3.12. SEARCH 636
9.4. 数据类型 637
9.4.1. Inceptor 支持的数据类型 637
9.4.2. 数据类型映射. 638
9.5. 系统表 639
9.6. Inceptor 与 Oracle 语法兼容 647
9.6.1. 基本元素. 647
9.6.2. 运算符. 648
9.6.3. SQL 语法. 648
9.6.4. 函数和表达式. 649
客户服务 651

免责声明

本说明书依据现有信息制作，其内容如有更改，恕不另行通知。星环信息科技（上海）股份有限公司在编写该说明书的时候已尽最大努力保证期内容准确可靠，但星环信息科技（上海）股份有限公司不对本说明书中的遗漏、不准确或印刷错误导致的损失和损害承担责任。具体产品使用请以实际使用为准。

注释：Java® 是 Oracle 公司在美国和其他国家的商标或注册的商标。Intel® 和 Xeon® 是英特尔公司在美国、中国和其他国家的商标或注册的商标。

版权所有 ©2016 年 - 2023 年星环信息科技（上海）股份有限公司。保留所有权利。

©星环信息科技（上海）股份有限公司版权所有，并保留对本说明书及本声明的最终解释权和修改权。本说明书的版权归星环信息科技（上海）股份有限公司所有。未得到星环信息科技（上海）股份有限公司的书面许可，任何人不得以任何方式或形式对本说明书内的任何部分进行复制、摘录、备份、修改、传播、翻译成其他语言、或将其全部或部分用于商业用途。

发布日期：2023-02-28

1. 前言

1.1. 读者对象

Inceptor 开发者指南是面向所有使用 Inceptor 数据库的用户，包括 Inceptor 数据库管理员和应用程序开发人员。

本开发者指南主要适用于执行以下任务：

- 创建和配置 Inceptor 数据库
- 监控和优化 Inceptor 数据库
- 管理和授权 Inceptor 的用户和角色
- 创建和管理模式对象，例如表、函数、存储过程等

在开始阅读本文档之前，您需要了解关系型数据库的概念，同时您也应该熟悉运行 Inceptor 数据库的操作系统。

1.2. 产品版本

本文档对应的产品及版本信息如 表 1 所示

表 1. 产品版本

产品	版本
Inceptor	9.3

1.3. 修订记录

本文档的修订记录如表 表 2 所示

表 2. 文档版本

版本	版本说明
v1.0	Inceptor 开发者指南

1.4. 文档阅读约定

1.4.1. 符号约定

在本文中可能出现下列标志，它们所代表的含义如 表 3 所示

表 3. 符号说明

符号	说明
⚠	用于警示潜在的危险情形，若不避免，可能会导致中度或轻微的人身伤害。

符号	说明
	用于突出重要及关键信息。“重要”不是安全警示信息，不涉及人身、设备及环境伤害。
	用于传递设备或环境安全警示信息，若不避免，可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “注意”不涉及人身伤害。
	用于添加备注信息以及其他相关说明。“备注”不是安全警示信息，不涉及人身、设备及环境伤害。

1.4.2. 命令行格式约定

格式	说明
<>	命令行参数（命令中必须由实际值进行替代的部分）采用“< >”括起来。
[]	表示用“[ ]”括起来的部分在命令配置时是可选的。
x|y…	表示从两个或多个选项中选取一个。
[x|y…]	表示从两个或多个选项中选取一个或者不选。
- -	表示由“- -”开始的行为注释行。

1.5. 获取和更新文档

您可以通过以下方式获取文档：

- 通过登录星环信息科技（上海）股份有限公司官方网站进行阅读文档， 文档支持中心
- 通过星环公司当地办事处相关联系人申请文档包

1.6. 意见反馈

星环信息科技（上海)股份有限公司为客户提供全方位的技术支持，用户如果在产品使用或维护过程 中，遇到难以确定或难以解决的问题，可以通过以下渠道自助获取信息服务：

- 论坛支持： http://support.transwarp.cn/
- 文档支持中心反馈： https://www.warpcloud.cn/#/documents-support/
  1. 选中需要反馈的段落，点击“纠错”；
  2. 填写问题描述以及您的邮箱；
  3. 点击**提交**，以上您已经成功向我们发送了意见反馈相关信息，我们会根据您的反馈进行处理并将后续结果发送至您预留的邮箱。

若通过以上方式仍不能解决问题，用户可以与就近的星环办事处联系，也可直接与公司技术支持部门人员联系。

- 技术支持热线电话：4007-676-098
- email： support@transwarp.io
- 官方网址： http://www.transwarp.cn/

2. 产品简介

本部分主要介绍 Inceptor 数据库的概况，以及作为一款国产化的自研数据库，使用 Inceptor 数据库执行用户数据库业务时所具备的技术优势。

本部分包含以下章节：

- 产品定位
- 产品优势
- 软件架构

2.1. 产品定位

随着大数据技术在企业数字化平台中深度应用，用户业务的分析需求变得越来越复杂。数据分析相关的业务逐步呈现为：离线业务与在线业务并存，分析型业务与检索型业务并存，结构化数据与非结构化数据并存。传统的、单一的数据平台架构已经很难满足用户业务对数据处理的低时延、高性能的使用诉求。

Inceptor 是星环自主研发的分布式分析型数据库，可以替代 Hadoop + MPP 的混合架构。我们使用标准的 SQL 语法支持用户业务的建设，并且能够给用户提供多模型数据分析、实时数据处理、存储与计算模块解耦、异构服务器混合部署等先进技术能力。用户可以通过一个 Inceptor 数据库，实现其数据仓库业务、实时数据仓库业务、数据集市业务、OLAP 数据分析业务、事务型分析型业务混合负载业务、异构数据联邦分析计算业务等各种场景的建设需求。

2.2. 产品优势

使用星环分布式分析型数据库 Inceptor 构建用户的数据业务，具备以下优势：

- **高扩展**

  Inceptor 数据可以按需进行水平扩展，且作业任务能够均匀调度，不存在木桶效应；

  存储和计算模块相互解耦，用户可以按需分配或者扩展集群资源；

  自主研发的作业调度技术，可以支持企业级混合负载业务的运行；

  底层利用容器技术实现资源调度和资源隔离，支持资源的弹性扩/缩容，可以为多租户弹性提供稳定可靠的数据库服务。

- **高性能**

  特有的向量化计算引擎，加速用户的查询分析性能；

  高性能的 OLAP 分析引擎，底层计算算子全向量化实现，高并发复杂分析在百毫秒内完成；

  匹配实时流，强化批流融合

  与星环自研的实时流引擎 Slipstream 无缝衔接，数据落地即可分析，提供了高吞吐（百万记录/秒）数据入库，实时业务秒级响应。

  深度适配新兴硬件，单服务器资源最大化利用。

- **高可靠**

  基于 Raft 一致性协议的存储引擎，具备自动故障迁移、自动数据修复的能力；

- **易开发**

标准 SQL 支持

支持 SQL 99/2003 标准语法，兼容 Teradata，Oracle，DB2 等方言；

多模态数据支持

支持 11 种数据模型的存储与计算，覆盖业务全场景；

联邦能力

支持 Inceptor 数据库与自身或者其他第三方数据库进行跨库的 SQL 联邦计算。

- **自主可靠**

  纯国产化数据库，帮助用户实现自主可控

  数据库的核心代码 100% 自研，且拥有第三方机构权威认证。

  对国产生态的支持

  全面适配国产 CPU、操作系统、ETL 工具、BI 分析报表等底层生态。

  成熟可靠

  数据库团队多年的技术积累，且 Inceptor 数据库在全行业都有落地实施案例。

2.3. 软件架构

2.3.1. 产品定位

Transwarp Inceptor 是一款创新型分布式架构，可提供超大规模、高效性能、稳定运行、极简使用的能力供用户数据库业务使用。其产品架构请参见下图：

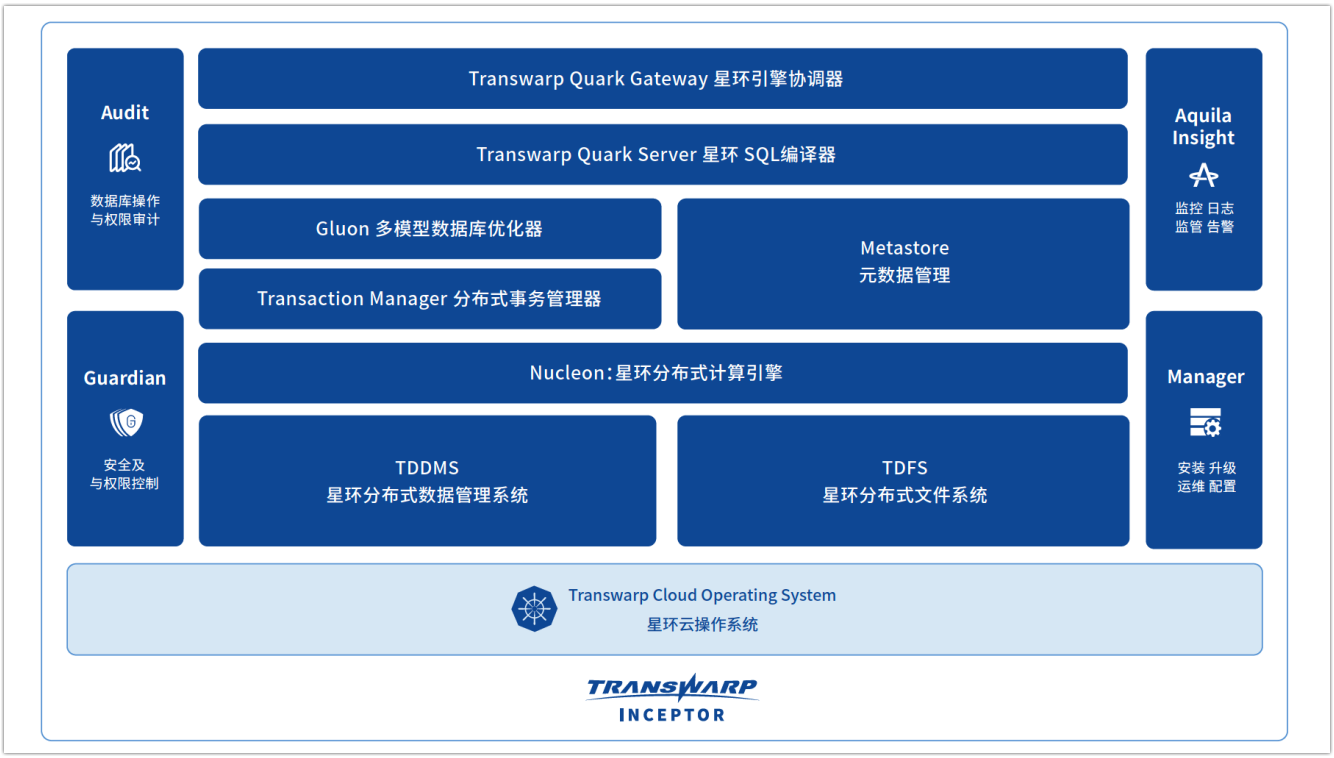


图 1. Inceptor 产品架构图

2.3.2. 核心组件

Inceptor 的产品功能模块清单及功能介绍如下：

模块名称	模块功能	模块介绍
Quark Gateway	引擎协调器	Quark Gateway 是星环自主研发的弱状态分布式计算引擎调配模块，可以将客户的负载均匀按需调度到不同的 Quark Server 服务上，同时还会负责监测 Quark Server 服务的健康状态，任务发送自动避开有故障的 Quark Server 服务达到系统整体高可用的目的。
Quark Server	SQL 引擎	Quark Server 是星环自主研发的分布式 SQL 查询引擎模块，兼容多种方言的 SQL 编译器，包括 HiveQL，Oracle，DB2，Teradata 等方言，也包括了算子和类型系统。
Gluon	多模型数据库优化器	Gluon 是星环自主研发的多模型数据库优化器，对 SQL 编译结果进行优化，通过基于规则优化、基于成本优化、物化视图等技术手段，自动判断选择合理高效的模型，拆分、分发计算任务。
Transaction Manager	事务管理器	Transaction Manager 是星环自主研发的分布式事务管理系统，是 Inceptor 中用于实现分布式事务的核心组件。
Nucleon	分布式执行引擎	Nucleon 是星环自主研发的分布式计算引擎。拥有高性能，高扩展性，高稳定性的特点。
Metastore	分布式元数据管理系统	Metastore 是星环自主研发的分布式元数据管理系统，可以支持上百万张表的元信息管理。

模块名称	模块功能	模块介绍
TDDMS	分布式数据管理系统	TDDMS（Transwarp Distributed Data Management System ）是星环自主研发的分布式数据管理系统，管理数据多副本间的强一致或最终一致；管理数据在存储介质上的合理分片分布，扩缩存储容量时，自动管理数据重分布，充分利用存储资源；保障数据高可用，在存储硬件故障时，保持数据存储服务不中断。
TDFS	星环分布式文件系统	TDFS（Transwarp Distributed File System）是星环研发的分布式文件系统，提供文件目录结构及有关服务；主要用于数据批量导入和导出的时候以文件形式进行数据交换的功能。

2.4. 管理组件

- **Transwarp Audit，大数据平台安全审计软件**

Transwarp Audit 面向 Inceptor 中对数据的操作和权限进行合理布控和监测，整合大数据平台各节点中的监控信息，实现对数据访问和操作的集中监控、查看和管理的智能化、可视化审计。 Audit 支持对各类审计事件做出快速、准确的定位，并进行过滤和归并，实现集中、综合的展现。

- **Transwarp Guardian，大数据安全管理软件**

星环科技提供的安全管理组件 Transwarp Guardian ，从用户认证、服务资源授权以及配额管理三方面为客户的业务逻辑、数据资产提供了全面的安全保障。 用户认证使用 LDAP，Kerberos 等协议保证只有经过身份甄别的用户才能访问系统，授权保证只有被赋予权限的用户才可以访问服务资源，而配额管理控制用户允许使用的资源大小，三个部分一同保证了平台大数据的安全。

- **Transwarp Aquila，智能运维分析平台软件**

Transwarp Aquila 作为一站式综合运维平台，提供监控仪表盘、告警通知、日志生命周期管理、日志检索、审计日志等功能。 同时，Aquila 还预置了基础监控、大数据监控、PaaS 层监控等多维度的监控资源，能做到整个 Inceptor 产品各个维度的开箱即用的一键运维。

- **Transwarp Manager，大数据管理软件**

Transwarp Manager 是负责配置、管理和运维 Inceptor 集群的图形化工具。用户只需通过几个手动步骤，就可以在 x86、ARM、MIPS 等各架构服务器或基于 Docker 的云端平台上完成集群部署，并且提供告警、 健康检测、 监控和度量等运维服务。 用户可以实时的浏览各服务的状态，并且在告警出现时采取恰当的措施以处理应对。 此外， Manager 还提供了一些便捷的运维功能，例如磁盘管理、软件升级和服务迁移等。

- **Transwarp Cloud Operating System，云原生操作系统**

TCOS 是星环自主研发的云原生操作系统，提供统一的资源调度框架，通过容器化编排，能够统一调度计算、存储、网络等各基础资源，TCOS 支持主流国产操作系统、国产服务器，并且支持混合部署，满足国产化部署需求。

3. 连接数据库

Inceptor 支持通过 Beeline、应用程序、外部工具等多种方式连接，本小节介绍在不同认证模式下，如何连接 Inceptor 数据库。

3.1. 连接方式

连接方式	说明
通过 Beeline 命令行连接	通过命令行方式执行 SQL 命令，如创建数据库。
通过 Waterdrop 客户端连接	通过客户端界面化方式管理数据库，支持跨平台管理多种数据库（如 Apache Hive）。
通过 JDBC 连接	支持标准的 JDBC/ODBC 接口，安装驱动后即可编写应用程序，读写 Inceptor 的数据。
通过 ODBC 连接	

除此以外，Inceptor 还支持使用标准 JDBC/ODBC 接口的工具进行连接，例如 Tableau 等。

3.1.1. 准备工作

3.1.1.1. 确认安全认证方式

Inceptor 支持 LDAP（轻型目录访问协议）和 Kerberos 认证方式，可帮助您提升数据安全性。在连接数据库前，您可以跟随本文确认 Inceptor 的认证方式，您也可以直接联系运维管理人员确认。

3.1.1.1.1. 操作步骤

1. 登录 Transwarp Manager 平台。
2. 在页面左侧，选择**仪表盘** > **集群**。
3. 在 **Inceptor** 区域，找到并单击对应的 Qurak 服务。
4. 单击**配置**标签，搜索参数 **hive.server2.authentication**，确认是否开启了 LDAP 认证。

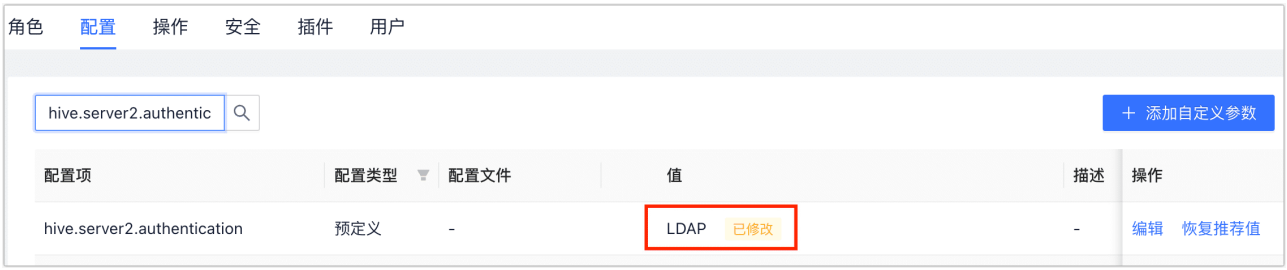


图 2. 已开启 LDAP 认证

值为 **NONE**：未开启 LDAP 认证。

值为 **LDAP**：已开启 LDAP 认证。

5. 单击**安全**标签，查看是否开启了 Kerberos 认证。

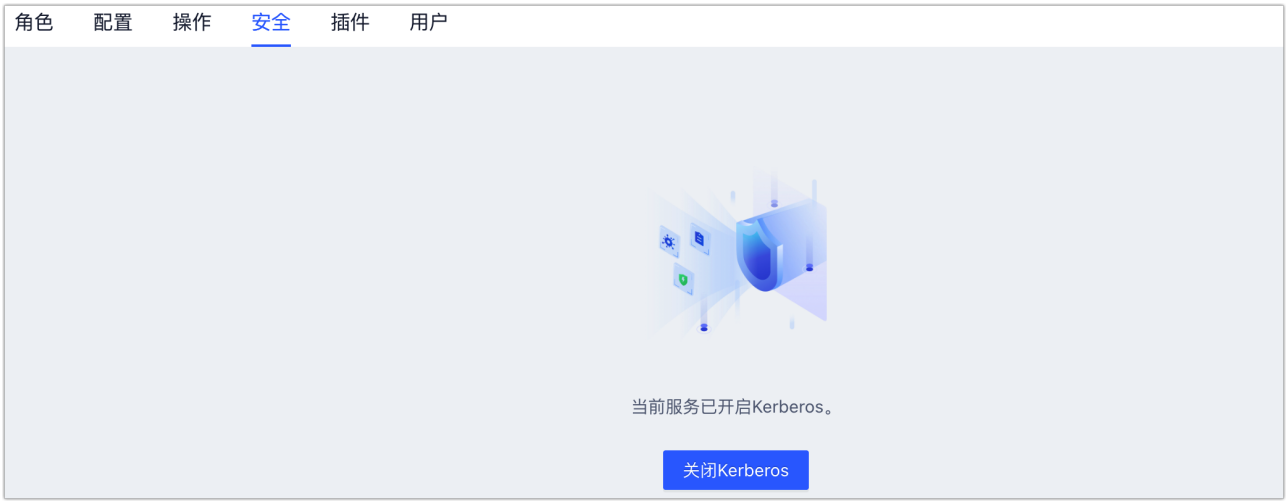


图 3. 已开启 Kerberos 认证

6. 如果上述认证均未开启时，则为免认证状态。

3.1.1.1.2. 扩展阅读

如果开启了 LDAP/Kerberos 认证，作为运维管理人员，您可以通过 Guardian 创建用户并授权，具体操作，见多租户指南。

3.1.1.2. 获取连接地址

Inceptor 通过 Quark Gateway 服务提供统一的 SQL 执行入口，连接数据库时，您需要提供 Quark Gateway 服务的连接地址，如 IP 地址或主机名。

3.1.1.2.1. 操作步骤

1. 登录 Transwarp Manager 平台。
2. 在左侧导航栏，单击**仪表盘** > **集群**。
3. 找到并单击对应的 Quark 服务。
4. 在**角色**标签页，获取 Quark 服务对应的 IP 地址和主机名。

图 4. 获取 IP 和主机名

3.2. 通过 Beeline 命令行连接

Beeline 是基于 SQLLine 的命令行客户端，它通过 JDBC 方式以 Thrift 协议连接至 Inceptor 数据库，连接后即可执行 SQL 命令（如创建数据库）。

> 更多关于 Beeline 的使用介绍，见 Beeline 官方文档。

3.2.1. 操作步骤

1. 下载客户端。
   a. 登录 Transwarp Manager 平台。
   b. 在页面右上角，选择 > **下载客户端**。

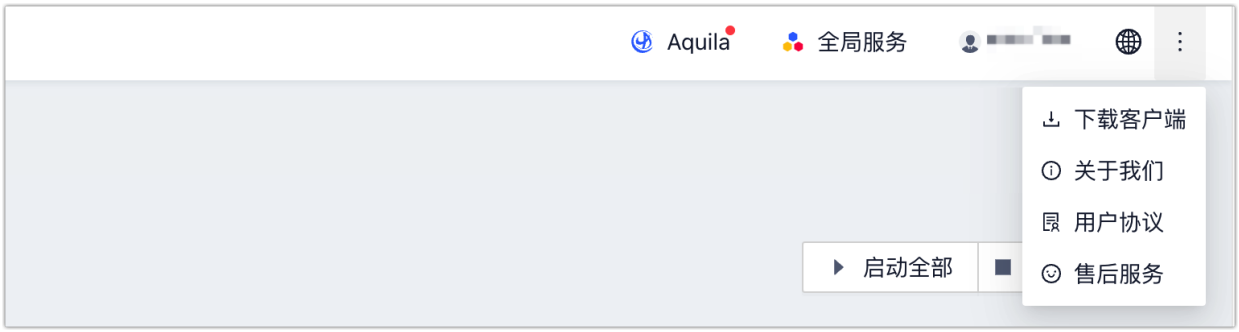


图 5. 下载客户端

   c. 选中要访问的服务（至少包含 Quark 服务），然后单击向右箭头。
   d. 单击**确定**，客户端将开始自动下载。

> 如果修改了集群中服务的配置（例如 Kerberos 认证从关闭设置为开启），则需要重新下载客户端。

2. 将下载好的客户端上传至集群中的任一机器中。

   您也可以将客户端，放置在 Inceptor 所属服务器同一内网的机器中，该机器需安装 Java （1.8 及以上版本）与 Hadoop 相关组件依赖。

3. 登录至该机器，进入客户端上传到的目录，执行下述命令解压文件。

```
tar -xvf tdh-client.tar
```

4. 执行下述命令（需 root 权限），完成客户端初始化。

```
source ./TDH-Client/init.sh
```

5. 确认 Inceptor 的认证方式，然后选择执行下述格式的命令，连接数据库。

   免认证：`beeline -u "jdbc:hive2://<server_ip/hostname>:<port>/<database_name>"`

   <server_ip/hostname>：Quark 服务的 IP 地址或主机名。

   <port>：服务端口，默认为 10000。

<database_name>：要连接到的数据库。

连接示例如下：

```
beeline -u "jdbc:hive2://172.16.20.1:10000/demodata"
```

LDAP 认证： `beeline -u "jdbc:hive2://<server_ip/hostname>:<port>/<database_name>" -n <username> -p <password>`

<server_ip/hostname>：Quark 服务的 IP 地址或主机名。

<port>：服务端口，默认为 10000。

<database_name>：要连接到的数据库。

<username>：用于身份认证的用户名。

<password>：用户名对应的密码。

连接示例如下：

```
beeline -u "jdbc:hive2://172.16.20.1:10000/demodata" -n core_user_01 -p Inceptor@123
```

Kerberos 认证： 您需要先执行 `kinit 用户名`，输入密码以获取 Kerberos 票据授权票据（TGT），然后再执行下述连接命令。

`beeline -u "jdbc:hive2://<server_ip/hostname>:<port>/<database_name>;principal=<principal_name>"`

<server_ip/hostname>：Quark 服务的 IP 地址或主机名。

<port>：服务端口，默认为 10000。

<database_name>：要连接到的数据库。

<principal_name>：Server Principal 中的 User Name 固定为 **hive**，即本参数的格式为 `hive/<Quark 服务所属设备的主机名>@TDH`，例如 `hive/idc3@TDH`。

连接示例如下：

```
# 获取 core_user_01 用户对应的票据，随后输入该用户的密码。
kinit core_user_01

# 执行连接命令
beeline -u "jdbc:hive2://172.16.20.3:10000/demodata;principal=hive/idc3@TDH"
```

6. 完成连接后，您可以执行所需的 SQL 命令，更多介绍，见 SQL 参考。

```
-- 查看数据库列表
SHOW DATABASES;
```

3.2.2. 更多认证方式

- Guardian Token 认证：当 Kerberos/LDAP 认证开启时，您也可以使用本方式，连接命令格式如下：

  ```
  $ beeline -u
  "jdbc:hive2://<server_ip/hostname>:<port>/<database_name>;guardianToken=<access_token>"
  ```

  关于 access_token 值的获取，请参见 Guardian 使用手册中的**使用 Guardian Access Token 访问服务**。

- CAS 认证：支持使用 CAS 进行单点登录/登出，此功能默认为开启，连接命令格式如下：

  ```
  beeline -u "jdbc:hive2://<server_ip/hostname>:10000/<database_name>;casTicket=<cas_ticket>"
  ```

  关于 cas_ticket 值的获取，请参见 Guardian 使用手册中的**使用 CAS Ticket 访问服务**。

3.2.3. HA（高可用）连接方式

如果您采用了高可用架构，您可以采用下述格式的命令进行连接（仅支持免认证和 LDAP 认证），从而使用完整的 HA 功能。

- 免认证

  ```
  beeline -u jdbc:hive2://<server2_ip/hostname>:<port>,
  <server2_ip/hostname>:<port>/<database_name>
  ```

- LDAP 认证

  ```
  beeline -u jdbc:hive2://<server2_ip/hostname>:<port>,
  <server2_ip/hostname>:<port>/<database_name> -n <username> -p <password>
  ```

3.3. 通过 Waterdrop 连接

Transwarp Waterdrop 是面向开发和数据库管理人员的管理工具，支持跨平台管理多种数据库（如 Apache Hive），帮助您实现界面化管理数据库、编辑/执行 SQL 语句等操作。

3.3.1. 环境要求

- Windows（x86_32/64）
- Linux（x86_32/64, ppc_64, s390）
- Mac OS X（x86_64）
- Solaris（x86_64）
- AIX（ppc_32/64）
- HP-UX（ia_64）

3.3.2. 操作步骤

下述步骤以 Windows 64 位操作系统为例，演示连接步骤。

1. 通过官网申请 Waterdrop 许可证（试用/商业版）。
2. 联系我们，获取驱动程序和安装包，然后完成激活操作 。
3. 在客户端电脑上，解压安装包并进入解压后的文件夹。
4. 双击 Waterdrop 应用程序，在弹出的对话框中，选择 **inceptor** > **Inceptor Server 2**。

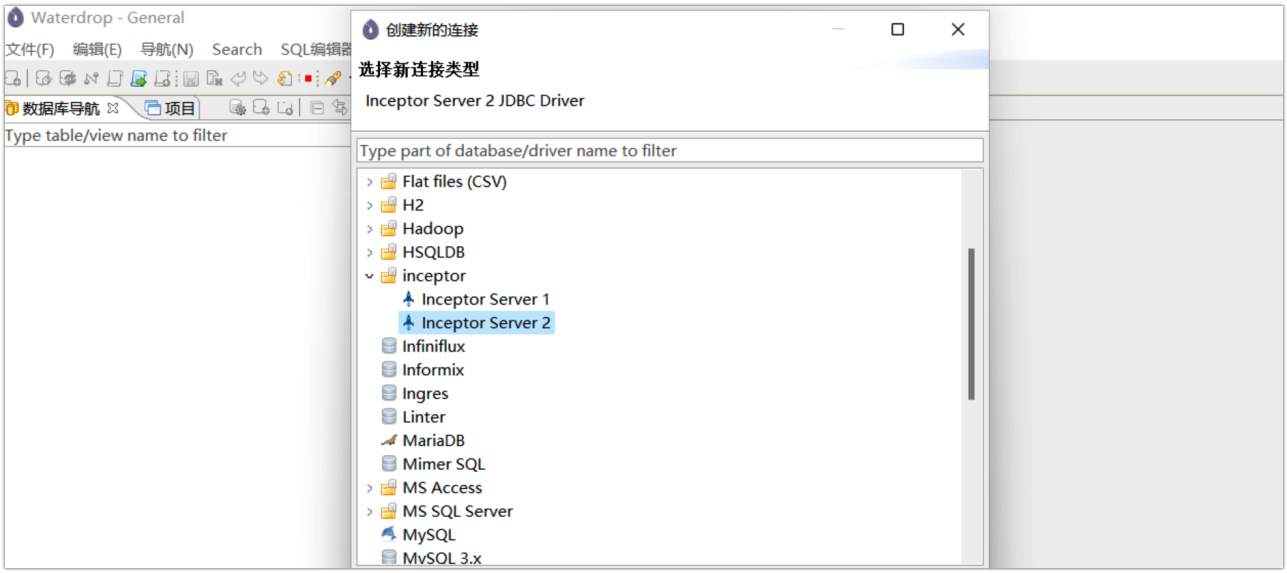


图 6. 选择连接类型

5. 单击 **Next**，在 **General** 选项卡中完成下述配置。

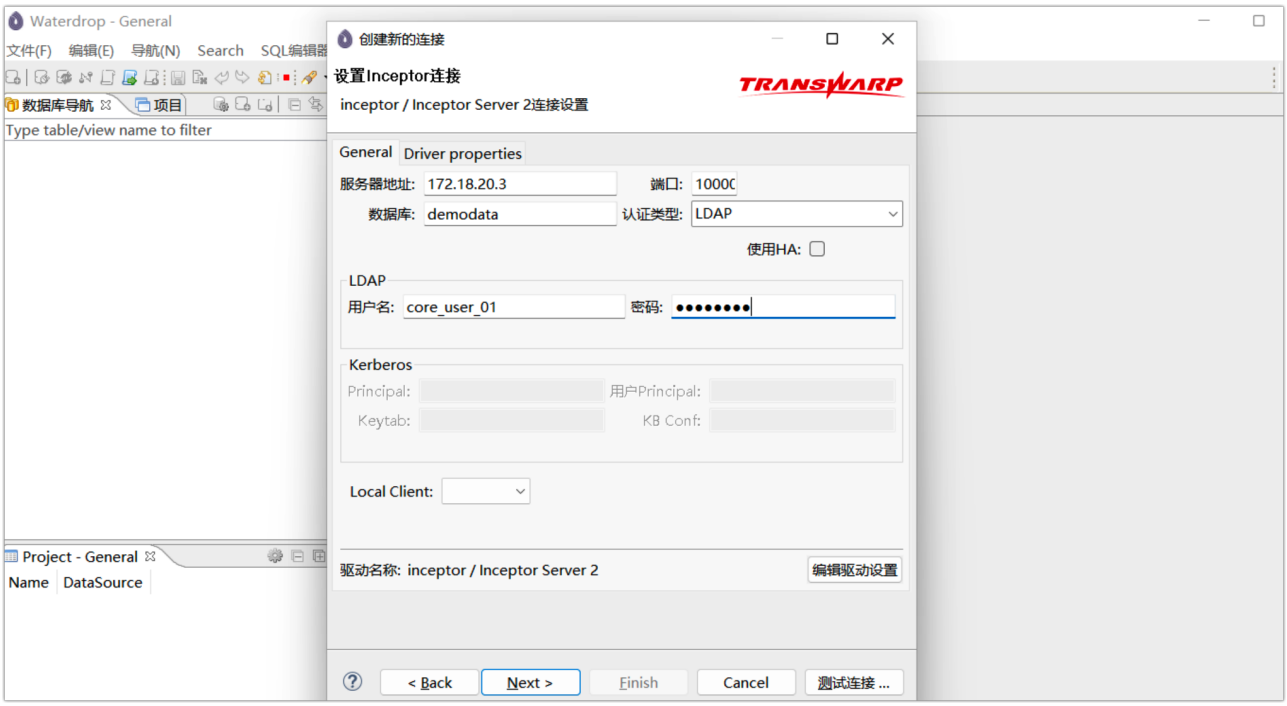


图 7. 配置连接信息

服务器地址：填写 Quark 服务的 IP 地址或主机名。

端口：服务端口，默认为 10000。

数据库：要连接的数据库。

认证类型：确认 Inceptor 的认证方式，然后选择配置下述信息。

免认证：保持为空，无需配置。

LDAP 认证：选择为 **LDAP**，然后填写用户名和密码。

Kerberos 认证：选择为 **Kerberos**，然后填写 Principle、用户 Principal，最后依次双击 **Keytab** 和 **KBConf** 对应的文本框，选择对应的配置文件。

> 相关配置文件可联系管理员获取。
>
> 为保障正常连接，您需要修改本地 hosts 文件，将 Quark 服务的 IP 地址与主机名进行关联。例如 Windows 平台下，该文件的路径为 `C:\Windows\System32\drivers\etc\hosts`。

使用 HA：如果您采用了高可用架构，选中该复选框并填写 HA 地址和端口。

6. 导入驱动文件。
   a. 单击**编辑驱动设置**，随后在弹出的对话框中单击 **Add File**。

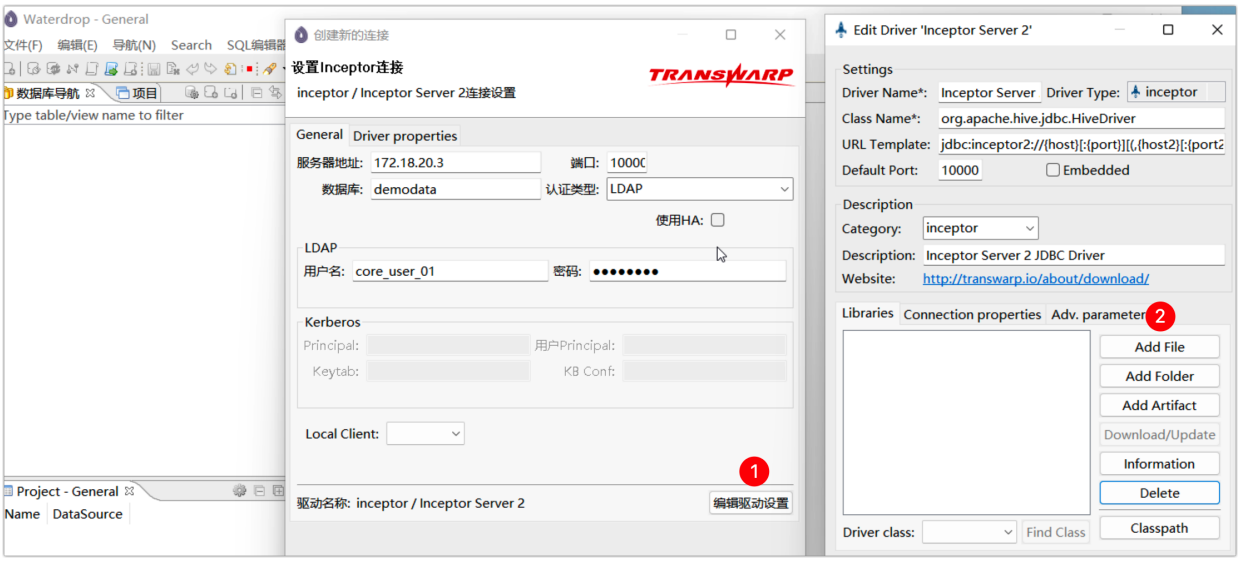


图 8. 添加驱动

   b. 选择并导入驱动文件，随后单击 **OK**。

> 如需验证配置正确性，可单击**测试连接**。

7. 单击 **Next**，无特殊要求可再次单击 **Next**。
8. 单击 **Finish** 完成配置。
9. 完成连接后，您可以双击对应的数据库获取表信息，也可以执行相关 SQL 命令。更多介绍，见 Waterdrop 用户手册和 Inceptor SQL 参考。

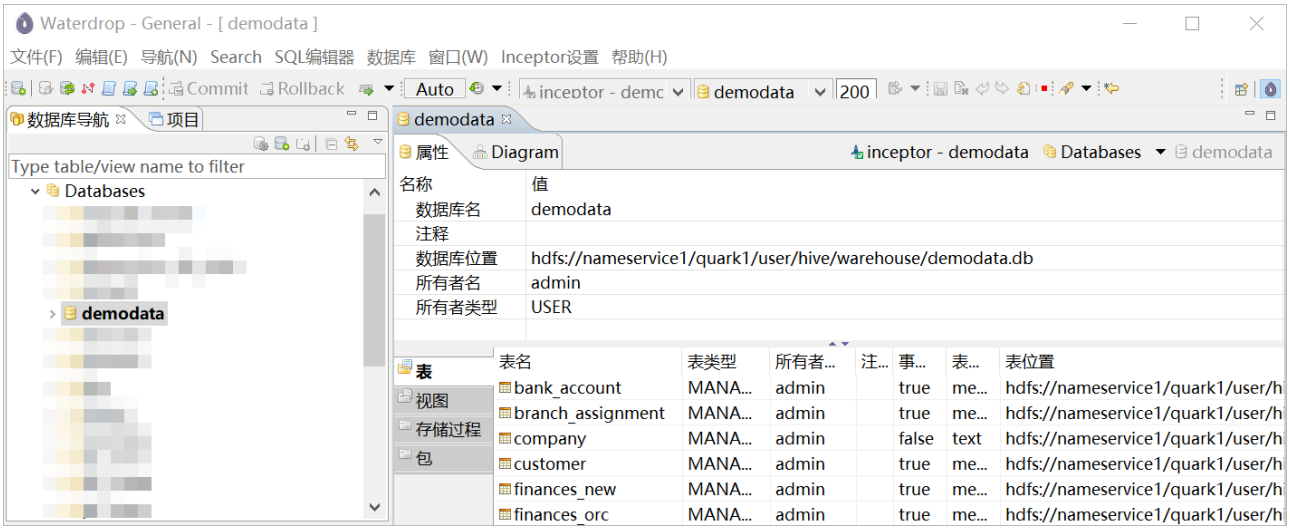

Waterdrop - General - [demodata]
文件(F) 编辑(E) 导航(N) Search SQL编辑器 数据库 窗口(W) Inceptor设置 帮助(H)
Commit Rollback Auto inceptor - demo demodata 200
数据库导航 项目
Type table/view name to filter
Databases
demodata
demodata
属性 Diagram
inceptor - demodata Databases demodata
名称 值
数据库名 demodata
注释
数据库位置 hdfs://nameservice1/quark1/user/hive/warehouse/demodata.db
所有者名 admin
所有者类型 USER
表
视图
存储过程
包
表名 表类型 所有者... 注... 事... 表... 表位置
bank_account MANA... admin true me... hdfs://nameservice1/quark1/user/h
branch_assignment MANA... admin true me... hdfs://nameservice1/quark1/user/h
company MANA... admin false text hdfs://nameservice1/quark1/user/h
customer MANA... admin true me... hdfs://nameservice1/quark1/user/h
finances_new MANA... admin true me... hdfs://nameservice1/quark1/user/h
finances_orc MANA... admin true me... hdfs://nameservice1/quark1/user/h

图 9. 连接成功示例

4. 开发规范

4.1. 概述

本章节从数据库对象、SQL 语句和应用程序开发角度介绍数据库开发的通用规范，为您提供规范化流程参考，帮助您更好地使用 Inceptor，提升工作效率，专注于业务创新。

通常情况下，数据库应用开发工作包含下述流程：

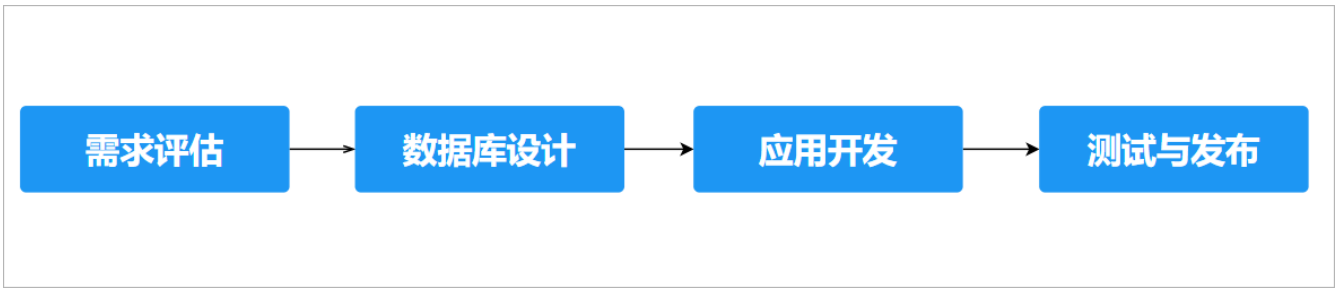


图 10. 数据库应用开发流程

- **需求评估**：由产品经理、需求方、技术人员等协同工作，围绕需求方案的合理性、可行性等内容深入调研评估。
- **数据库设计**：需要综合考虑性能、成本、效率和质量等因素设计数据库对象，例如 E-R 图（实体-联系图）、表结构、数据类型等。
- **应用开发**：围绕需求开发相关应用程序，完成数据库的连接、读写交互等程序代码的实施。
- **测试与发布**：通过测试及时暴露代码问题并修正，完成调整后平稳发布到线上。

为帮助您更好地运用 Inceptor 数据库的特性能力，发挥性能优势，本章节主要围绕数据库设计阶段和应用开发阶段介绍相关规范：

- 数据库对象设计规范
- SQL 设计规范
- 应用程序开发规范

4.1.1. 数据库设计规范

本章节从下述角度介绍数据库设计规范，为您设计高效的业务系统提供参考，帮助您充分发挥数据库性能。

- 对象命名
- 表结构设计（含分区/分桶）
- 字段设计

4.1.1.1. 对象命名

在使用数据库的过程中，您将根据业务需求创建各种对象（如数据库、表等）并读写数据，您可以通过阅读本文，了解数据库对象的命名规范。

适用对象	规范详情
所有对象	以小写英文字母开头，避免大小写混用。
	使用小写的英文名词全称、通用缩写和阿拉伯数字，单词间通过下划线（_）分隔。
	由于保留关键字具有特殊的意义，在命名对象时，需避免使用 Inceptor 保留关键字。
数据库	名称长度不超过 12 个字符，有助于辨识和降低数据传输量。
	按业务、产品线等角度划分数据库，通过名称即可获知数据库的用途，例如测试数据库库命名为 test_iot。
表	名称长度不超过 30 个字符，有助于辨识和降低数据传输量。
	同一业务的表使用相同前缀，且表名具有业务意义，例如 iot_device
	特殊意义的表，通过统一的表名说明其用途和日期，例如备份表命名为 bak_iot_device_20230101，临时表命名为 temp_iot_device_20230101。
	表名需要增加数字时，推荐数字放在末尾并有序递增，例如 iot_device_00、iot_device_01。
字段	名称长度不超过 30 个字符，有助于辨识和降低数据传输量。
	不同表之间的字段，如要表示的含义相同，应采用相同的字段名。
其他	其他数据库对象，如存储过程、视图等，命名规范类似。以存储过程为例，推荐以 proc 为前缀，即命名格式为 proc_{表全名或简写}。

4.1.1.2. 表结构设计

表是一种结构化的文件，用来存储特定类型的数据。您可以通过阅读本文，了解表结构设计的通用规范。

4.1.1.2.1. 通用原则

- 基于业务和数据源特点，选择合适的表存储格式。
- 表结构设计应以业务、性能需求出发，可适当冗余存储数据（如低频修改的字段）。
- 表、字段应具备 COMMENT 属性并配置注释信息。
- 需进行 JOIN 操作的字段应使用相同的数据类型，避免隐式转换。
- 不同表中含义相同的字段，应定义相同的名称和数据类型。

4.1.1.2.2. 分区/分桶设计

面对大规模数据时，通过良好的分区/分桶设计，可有效减少查询时要扫描的数据范围，提升查询效率。更多操作介绍，见创建分区表、创建分桶表。

- **分区**：通过特定条件（例如值相等）将表的数据分发到分区目录中。
- **分桶**：通过分桶键哈希取模（key hashcode %N）的方式，将表或分区中的数据随机分发到桶中。

分区和分桶的数量并非越多越好，建议如下：

类别	设计建议
分区	避免使用多级分区，以免产生过多分区数影响计算和存储性能。
	基于时间分区时，推荐按照年或月进行分区，颗粒度太细可能产生过多的分区影响性能。
	可选择 GROUP BY 或 DISTINCT 包含的列作为分区列。
	避免选择存在数据倾斜的列作为分区列。
分桶	选择离散度高的列作为分桶字段，避免桶的数据量悬殊过大，让数据均匀分布。 推荐在分桶操作前，查看每列的字段值数量分布情况。
	选择长度较短的列作为分桶字段，避免影响分桶判断的性能。
	不对分桶表执行 automerge，避免削弱或干扰分桶的优化作用。
	手动分桶时，选择的分桶数量不要产生过多的小文件，每个分桶建议不小于 200 MB。
	分桶梳理建议为质数，且小于 CPU 总核数。

4.1.1.2.3. 冷热分离设计

Inceptor 支持表级别的数据存储的冷热分离，例如将查询频度高的数据（称为热数据）存储在 SSD 介质；将查询频度低的数据（称为冷数据）存储在 HDD 介质，从而更好地平衡存储成本和查询性能。相关使用方法和注意事项，见数据存储冷热分离。

4.1.1.3. 字段设计

字段可以看做表格中的列，它定了事物的属性（例如姓名），而表通常会包含一个或多个字段。您可以通过阅读本文，了解字段设计的通用规范。

- 不同表中含义相同的字段，应定义相同的名称和数据类型。
- 具备 COMMENT 属性并配置注释信息。
- 定义为 NOT NULL 属性并设置默认值。
- 推荐使用变长字符串数据类型（VARCHAR）并指定最大长度，如值的长度固定，可选用 CHAR 来提升效率。
- 推荐选用高效合适的数据类型，例如在精度满足业务需求的情况下，数据类型的优先级从高到低为整数、浮点数、NUMERIC。
- 推荐时间类型选用 TIMESTAMP，可降低存储消耗，但需注意时间范围是否满足业务需求。
- 避免使用 BLOB、TEXT 类型，减少内存和存储资源的消耗。

4.1.2. SQL 设计规范

作为数据库的核心语言，SQL 语言通常包含 DDL（数据定义语言）、DML（数据操作语言）、DQL（数据查询语言）和 DCL（数据控制语言），作为开发者主要关系前三种语言的设计规范，DCL 通常由数据库管理员执行和管理。

4.1.2.1. DDL

DDL 语言主要用来定义/改变表结构、数据类型等操作，您可以通过阅读本文，了解 DDL 设计的通用规范。

本文侧重介绍 DDL 变更的相关规范，关于创建对象时需遵循的规范，见数据库设计规范。

- 集中统一执行 DDL，避免在业务高峰期执行以免影响业务访问。
- 大数据量场景下，需评估 DDL 变更引发的锁表风险。
- 降低存储过程、触发器和函数的使用，避免将业务逻辑与数据库深度耦合。
- 对表的多次 ALTER 操作合并为一次执行。
- 高危操作的 DDL（如 DROP TABLE），在操作前需谨慎评估，避免对象间或业务存在依赖关系而引发故障。
- 变更字段定义时附上原有的注释，避免注释丢失。
- 推荐使用 SQL 规范检测工具。

4.1.2.2. DML

- 避免一次更新或删除大批量数据（例如大于 50,000 条），可分批执行。
- 单条语句仅更新一个表，避免更新多个表。
- INSERT 语句需指定具体的字段。
- INSERT/DELETE 语句的优化规则为操作粒度小（不超过 100 条），具有明确的 WHERE 条件。
- INSERT 语句使用 BATCH 提交时（例如 INSERT INTO table_name VALUES(),(),()… ），应控制 VALUE 数量。
- SQL 语句在程序中传入的参数值类型和数据库中该字段的类型相同，避免发生隐式类型转换。

4.1.2.3. DQL

- 在使用 SELECT 语句时，指定要查询的字段，避免使用 SELECT *，降低网络传输和存储 I/O 消耗的时间。
- 通过客户端（如 JDBC）操作数据库时，使用 prepared statement 避免 SQL 注入并提升性能。
- 避免或减少在核心业务的 SQL 语句中使用复杂查询，便于业务理解，同时充分利用多核 CPU 提升性能。
- 无去重需求时，优先使用 UNION ALL 来代替 UNION，提升查询效率。
- 使用 COUNT(*) 来代替 COUNT(列名)，避免漏统计列为 NULL 值的行。
- 包含 DISTINCT、ORDER BY、GROUP BY 子句的查询，将中间结果集控制在 10000 行以内，
- 避免使用读锁，以免事务过大引发锁等待，从而影响业务。
- 避免跨数据库的 JOIN 操作，从而降低数据库间的耦合。
- 减少或避免排序，提升 SQL 执行效率。
- 减少或避免使用 IN 操作，可改写为 JOIN 操作。

4.1.3. 应用程序开发规范

作为应用开发人员，下载驱动程序后，即可基于 JDBC / ODBC 开发相关应用程序，连接并操作 Inceptor 中的数据。遵守应用程序的开发规范，可以帮助您设计简洁高效的交互，更好地使用 Inceptor，充分发挥其性能。

- 基于 JDBC 开发
- 基于 ODBC 开发

4.1.4. 推荐规范

4.1.4.1. 连接管理

- 使用连接池：由于建立连接属于相对昂贵的操作，推荐使用连接池来复用连接并设置连接池上限（maximumPoolSize 参数），避免消耗大量系统资源，提升执行效率。
- 及时释放连接：应用程序完成作业后，应及时断开连接以释放资源，例如通过超时时间参数进行控制。

4.1.4.2. 类/方法使用

- 禁用自动提交：通过应用程序来保证事务的提交，即在指定的业务SQL执行完之后显式提交事务，在退出客户端之前请保证所有事务已提交。
- 批量更新数据：对于批量插入更新，推荐通过 addBatch 将多条 SQL 的插入更新记录先缓存在客户端，然后在 executeBatch 时一起发送到数据库服务器，减少数据传输的往返次数，提升性能。
- 正确使用 get()：尽管 JDBC 允许程序通过 getString() 或 getObject() 来获取任何数据类型，而使用正确的 get 方法可避免数据类型转换。
- 使用预编译：通过 prepareStatement 预编译 SQL，减少语法校验次数，提升处理效率，同时避免拼接 SQL 时造成 SQL 注入。

5. 数据写入

本节主要介绍如何将数据写入 Inceptor，主要包括

- 实时写入 介绍如何将数据实时写入 Holodesk，适用于实时大屏等实时场景
- 批量写入 介绍如何将数据批量写入 Holodesk，适用于数据仓库等批处理场景

5.1. 实时写入

Inceptor 可以借助 Slipstream 流计算引擎，提供高吞吐、低延迟的数据实时写入能力

```
Transwarp Slipstream
是星环科技自主研发的企业级、高性能实时流计算引擎，支撑百万级高吞吐、毫秒级低延时业务需求。Slipstream
支持事件驱动和微批处理两种模式，支持exactly-once语义、复杂事件处理(CEP)、规则引擎等功能，支持SQL
编程与开发。Slipstream帮助用户快速开发实时数据仓库、实时报表分析、实时智能推荐、实时欺诈检测与风险控制等应用。
```

本节主要包含以下内容：

- 实时写入示例

  非安全模式下实时写入示例 非安全模式下，流式实时写入示例

  安全模式下实时写入示例 安全模式下，流式实时写入示例

- Kafka\Event Store、Slipstream 功能介绍

  创建 Topic 如何在数据源Kafka/Event Store中创建topic

  创建 Stream 如何在Slipstream中创建stream

  创建 Streamjob 如何在Slipstream中创建、查看、启动以及停止sreamjob

  创建Producer并发送数据 如何在数据源Kafka/Event Store中创建producer并发送数据

5.1.1. 实时写入示例

例 1. 如何查看并更改服务的安全模式

第一步：通过 Manager确定服务的安全模式信息

进入Manager中,选择服务kafka，进入安全界面，显示”当前服务已开启Kerberos“，即代表该服务为安全模式。

查询该服务的安全认证模式，当前为LDAP认证模式。

hive.metastore.warehouse.dir	预定义	hive-site.xml	hdfs://nameservice1/quark1/user/hive/warehouse
hive.server2.authentication	预定义	-	LDAP 已修改
hive.server2.authentication.oauth2.enabled	预定义	-	true
hive.server2.enabled	预定义	hive-site.xml	true

5.1.1.1. 非安全模式下实时写入示例

第一步：进入 TDH-Client\kafka\bin文件夹

```
cd TDH-Client\kafka\bin
```

第二步：使用Kafka/Event Store 脚本创建一个 Topic

```
./kafka-topics.sh --zookeeper localhost:2181 --create --topic topic_orders --partitions 3
--replication-factor 1 ①
```

① kafka-topics.sh 用于创建 Topic

第三步：连接Slipstream并创建Stream

```
CREATE stream stream_slipstream_to_inceptor_orders(
trans_id int,
acc_num int,
trans_time date,
trans_type string,
stock_id string,
price decimal,
amount int)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
TBLPROPERTIES(
"topic"="topic_orders",
"kafka.zookeeper"="localhost:2181",
"kafka.broker.list"="localhost:9092"
);
```

① 连接 Slipstream 方式，请参考 Slipstream 使用手册

第四步：通过 Slipstream 创建 Stream Job

```
CREATE STREAMJOB streamjob_slipstream_to_inceptor_orders AS (
"insert into database_inceptor.orders select trans_id , acc_num ,trans_time ,trans_type ,stock_id ,
price ,amount  from stream_slipstream_to_inceptor_orders")
JOBPROPERTIES("morphling.hdfs.flush.interval.ms"="1000"); ①
```

① morphling.hdfs.flush.interval.ms入库间隔参数，单位 ms。间隔越小延迟越低。

第五步：通过 Slipstream 启动 Stream job

```
START streamjob streamjob_slipstream_to_inceptor_orders;
```

第六步：使用 Kafka/Event Store 脚本，启动 Kafka/Event Store console producer 发送数据

```
./kafka-console-producer.sh --broker-list localhost:9092 --topic topic_orders
>943197522,6513065,2014-01-05,b,AA7105670,12.13,200
>929634984,3912384,2014-02-05,b,UA1467891,11.11,300
```

第七步：通过 Inceptor Quark查看数据入库结果

```
select trans_id , acc_num ,trans_time ,trans_type ,stock_id , price ,amount from orders;
```

5.1.1.2. 安全模式下实时写入案例

第一步：进入 TDH-Client\kafka\bin文件夹

```
cd TDH-Client\kafka\bin
```

第二步：使用 Kafka/Event Store 脚本创建一个 Topic

```
./kafka-topics.sh  --zookeeper hostname:port --create --topic topic_name --partitions 3
--replication-factor 1
-- hostname的值为ZooKeeper Server节点的主机名或IP地址
-- port为服务zookeeper的端口
```

第三步：在 Inceptor Quark 中创建表

```
Create table table_name(column1 datatype1, column2 datatype2, coulumn3 datatype3, ......) STORED AS
HOLODESK;
```

第四步：连接 Slipstream

```
beeline -u "jdbc:hive2://hostname:port/database_name" -n admin -p admin
```

① port为其参数hive.server2.thrift.port 的值，默认为10010，hostname为Slipstream节点的主机名或IP地址；

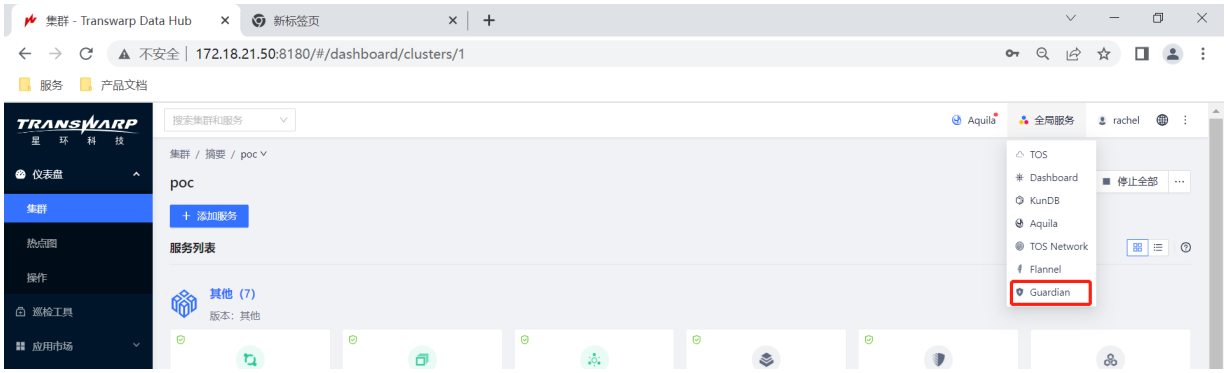
第五步：进行 Kafka 安全配置

为了在安全模式下成功创建 Stream 并连接上 kafka 中的 topic，需要上传kafka的keytab文件至Slipstream的每一个节点上。

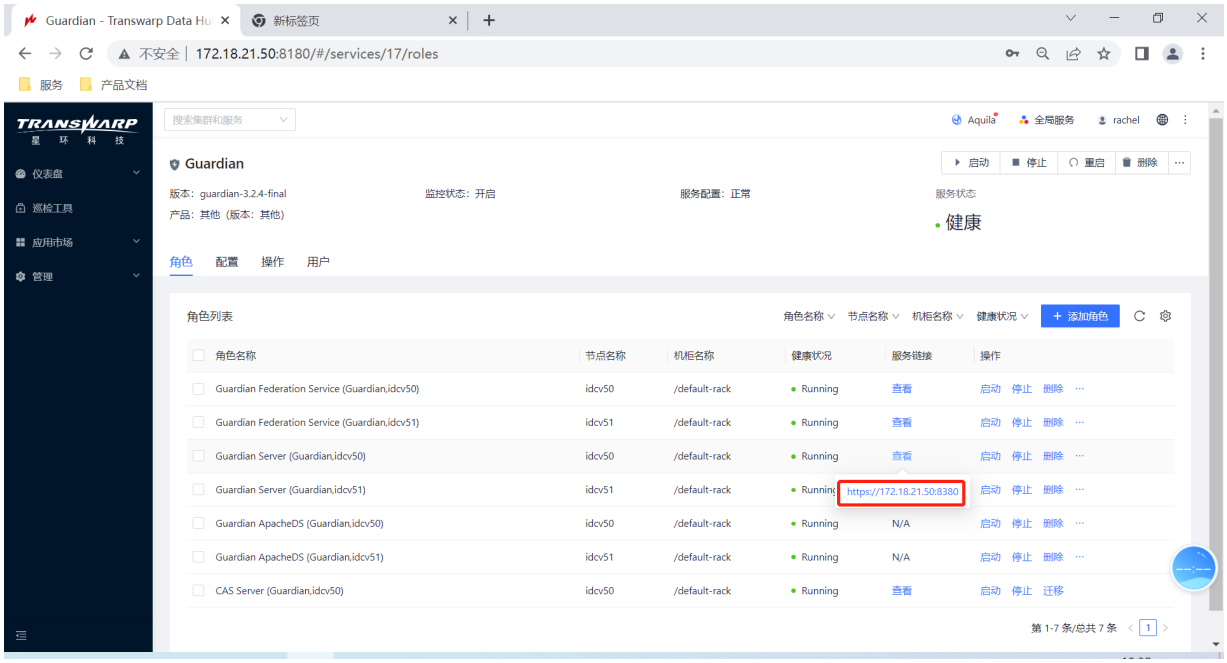
此操作需要进入 Guardian Server 中进行下载，并将下载的kafka.keytab文件放在Slipstream的每一个节点的相同路径下。为支持Slipstream访问该文件，推荐的文件路径为：/etc/transwarp/conf/kafka.keytab。

具体操作步骤如下：

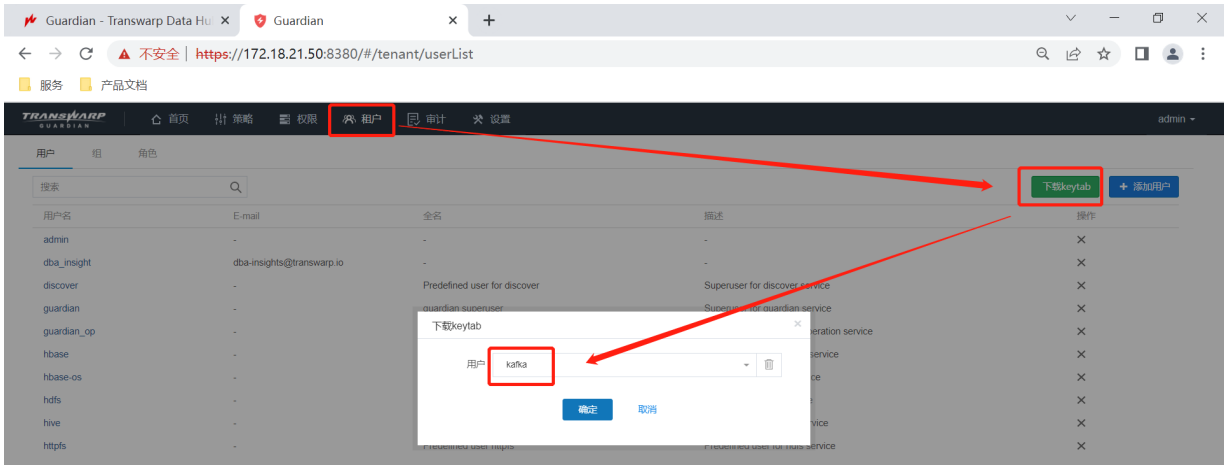
1. 进入集群的 Transwarp Manager 界面，点击右上角的全局服务，打开 Guardian 服务。



2. 在角色中任意选择 Guardian 节点并点击进入，打开 Guardian 网站。由于 TDH 集群的 Guardian 默认采用 Guardian Federation 提供的 OAuth 2.0 认证机制，首次登录时会自动跳转到 Guardian Federation 登录页面（需要在浏览器所在机器中配置好节点 IP 与主机名的映射，也可以手动将主机名更改成 IP 地址）



3. 登录之后，点击上侧菜单栏中的【租户】，然后点击右上角的【下载keytab】，在弹出框中输入用户[kafka]，点击确定进行下载。



4. 下载之后需要将该keytab文件上传至集群的每一个节点中，为保证Slipstream能够成功读取该文件，推荐将其路径为：

```
/etc/transwarp/conf/kafka.keytab
```

5. 之后在文件路径下运行命令，查看principal参数的具体值：

```
klist -k kafka.keytab
```

第六步：通过 Slipstream 建立 stream

```
CREATE stream stream_name(column1 datatype1, column2 datatype2, coulumn3 datatype3, ......)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' //数据输入时用”,“作为分隔符
TBLPROPERTIES("topic"="topic_name", "kafka.zookeeper"="hostname1:port1",
"kafka.broker.list"="hostname2:9092",
"transwarp.consumer.security.protocol"="SASL_PLAINTEXT",
"transwarp.consumer.sasl.kerberos.service.name"="kafka", //kafka为超级用户
"transwarp.consumer.sasl.jaas.config"="com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true storeKey=true keyTab=\"/etc/slipstream1/conf/kafka.keytab\" principal=\"kafka@TDH\""
//Principle参数的值
);
```

第七步：通过 Slipstream 创建并启动Stream Job

创建Stream Job将stream中的全部数据实时写入表格中：

```
CREATE STREAMJOB streamjob_name AS
("insert into table_name
Select * from stream_name")
JOBPROPERTIES("morphling.hdfs.flush.interval.ms"="1000");
```

① morphling.hdfs.flush.interval.ms入库间隔参数，单位 ms。间隔越小延迟越低。

启动Stream Job，数据开始实时入库：

```
START streamjob streamjob_name;
```

列出正在运行的StreamJob，检查job是否启动成功：

```
LIST STREAMJOBS;
```

需要关闭job时，运行Stop streamjob停止

```
STOP streamjob streamjob_name;
```

第八步：使用Kafka/Event Store 脚本，启动 Kafka/Event Store console producer 发送数据

当TDH-client开启安全时使用Kafka，需要进入kafka客户端对configure文件进行手动配置，或者新建配置文件security-client.properties，并在其中添加下列语句：

```
security.protocol=SASL_PLAINTEXT
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=kafka
```

然后按下列规则运行bin中的操作文件，其中../config表示文件security-client.properties被放置的路径：

```
kafka-console-producer.sh 使用--producer.config ../config/security-client.properties 进行指定；

kafka-console-consuemr.sh 使用 --consumer.config ../config/security-client.properties 进行指定；

其他脚本如kafka-topic.sh、kafka-consumer-groups.sh 一般都使用 --command-config ../config/security-client.properties 进行指定。
```

例如：此时security-client.properties放在bin中，hostname是kafka节点的主机名或IP地址：

```
./kafka-console-producer.sh --broker-list hostname:9092 --topic topic_name --producer.config security-client.properties
手动输入数据，回车提交：

>200001,beige azure white purple navajo,Manufacturer#5,Brand#51,ECONOMY BURNISHED COPPER,31,JUMBO BAG,1101,uickly regular accoun

>200002,beige azure white purple navajo,Manufacturer#5,Brand#51,ECONOMY BURNISHED COPPER,33,JUMBO BAG,1101,wwooowww
```

5.1.1.3. Kafka\Event Store、Slipstream 功能介绍

例 2. 管理Topic

Kafka/Event Store的相关脚本文件存放在在TDH-Client文件夹下。运行创建或查看topic脚本之前，需要执行source ./init.sh 初始化环境变量

```
创建topic
./kafka-topics.sh  --zookeeper <hostname:port> --create --topic <topic_name> --partitions 3
--replication-factor 1

查看Topic
./kafka-topics.sh  --zookeeper <hostname:port> --describe --topic <topic_name>
```

例 3. 管理STREAM

需要在slipstream服务中操作

非安全模式创建Stream

```
CREATE stream stream_name(column1 datatype1, column2 datatype2, coulumn3 datatype3, ......)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ①
TBLPROPERTIES("topic"="topic_name",
"kafka.zookeeper"="hostname1:port1", ②
"kafka.broker.list"="hostname2:port2"); ③
```

① 指定分隔符

② hostname1为zookeeper节点的主机名或IP地址，port1为Zookeeper的参数zookeeper.client.port的值

③ hostname2为brocker指定节点的主机名或IP地址，port2为broker的端口号

安全模式创建Stream:

```
CREATE stream stream_name(column1 datatype1, column2 datatype2, coulumn3 datatype3, ......)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ①
TBLPROPERTIES("topic"="topic_name",
"kafka.zookeeper"="hostname1:port1", ②
"kafka.broker.list"="hostname2:port2", ③
"transwarp.consumer.security.protocol"="SASL_PLAINTEXT",
"transwarp.consumer.sasl.kerberos.service.name"="kafka",④
"transwarp.consumer.sasl.jaas.config"="com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true storeKey=true keyTab=\"/etc/slipstream1/conf/kafka.keytab\"
principal=\"kafka@TDH\""
);
```

① 指定分隔符

② hostname1为zookeeper节点的主机名或IP地址，port1为Zookeeper的参数zookeeper.client.port的值

③ hostname2为brocker指定节点的主机名或IP地址，port2为broker的端口号

④ kafka超级用户

⑤ 可以通过Klist命令查询

⑥ 安全模式下的配置与操作方法请查看下文的"流式实时写入示例"

查看当前数据库的stream

```
SHOW streams;
```

查看其他数据库中的所有stream

```
SHOW streams from <database_name>;
```

删除Stream

```
DROP stream <stream_name>;
```

例 4. 管理STREAM JOB

需要在slipstream服务中操作

创建Streamjob

```
CREATE STREAMJOB <streamjob_name> AS("insert into table_name Select * from stream_name")
JOBPROPERTIES("morphling.hdfs.flush.interval.ms"="1000"); ①
```

① 用于设置入库间隔，单位 ms。间隔越小延迟越短，开销越大

启停Streamjob

```
启动Streamjob
START STREAMJOB <streamjob_name>;

暂停Streamjob
STOP STREAMJOB <streamjob_name>;
```

查询Streamjob

```
--查询当前数据库中的所有Streamjob
SHOW STREAMJOBS;

--查询当前数据库中所有处于运行中的Streamjob
LIST STREAMJOBS;
```

删除Streamjob

```
DROP STREAMJOB <streamjob_name>;
--若此StreamJob处于正在运行的状态时，则需要先停止任务才能对其进行删除。
```

例 5. 创建Producer发送数据

非安全模式下，创建producer

```
./kafka-console-producer.sh --broker-list hostname:port --topic topic_name
--hostname和port是producer节点指定的主机名或IP地址。
```

安全模式下创建Producer

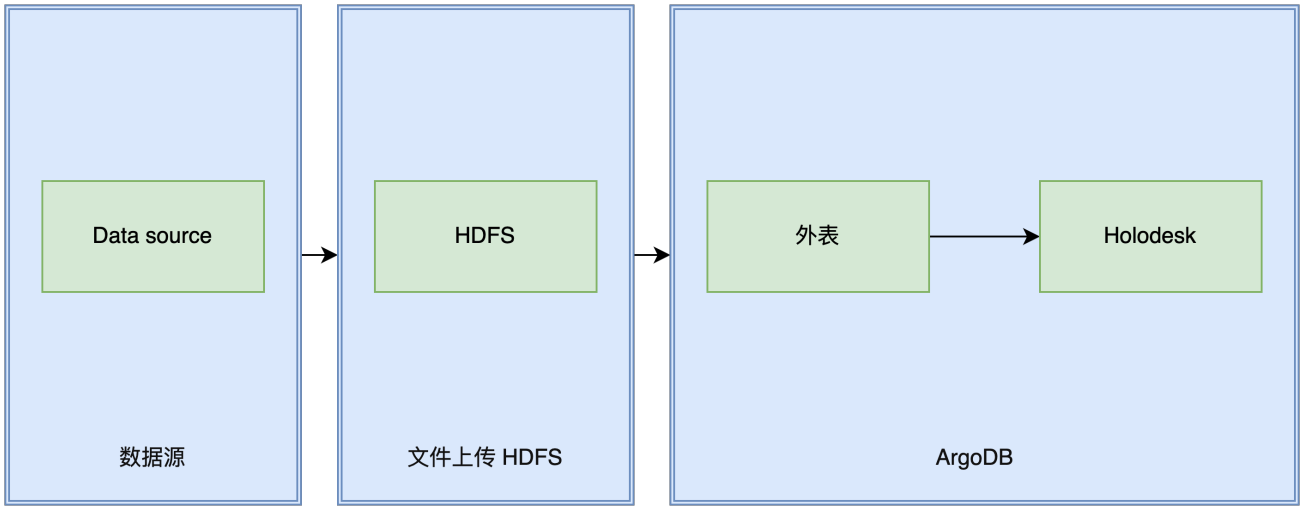
```
当TDH-client开启安全时使用Kafka，需要进入kafka文件夹对configure文件进行手动配置，或者新建配置文件security-client.properties并放置于kafka的bin路径下
在文件中添加下列语句：
security.protocol=SASL_PLAINTEXT
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=kafka

./kafka-console-producer.sh --broker-list hostname:9092 --topic topic_name --producer.config security-client.properties
然后发送数据，数据通过对应的topic以及运行的Streamjob，实现将producer发送的数据实时的写入目标表中。

--安全模式下，运行脚本文件时，需指定配置项
1）kafka-console-producer.sh 指定配置项。例如--producer.config../config/security-client.properties
2）kafka-console-consuemr.sh 指定配置项。例如 --consumer.config ../config/security-client.properties
3）其他脚本如kafka-topic.sh、kafka-consumer-groups.sh 指定配置项。例如 --command-config ../config/security-client.properties
```

5.2. 批量写入

接下来，我们介绍如何通过 SQL 将本地文件批量写入 Holodesk 表。



将数据从本地磁盘批量写入 Holodesk

```
--以 a.txt 文件为例
第一步：将数据文件上传到 HDFS
hfds dfs -put /tmp/a/a.txt

第二步：创建一张 text 外表
CREATE EXTERNAL TABLE text_a (id INT, name STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/tmp/a';

第三步：创建一张 holodesk 表
create table holo_a(id int,name string) stored as holodesk;

第四步：text 外表数据导入 holodesk
insert into holo_a select * from text_a;
```

此外，我们支持第三方数据源通过API 写入方式Holodesk

6. 应用程序开发

6.1. 概述

Transwarp Inceptor（简称 Inceptor）是星环科技自主研发的分布式分析型数据库，可以替代 Hadoop+MPP 混合架构。支持标准 SQL 语法，提供多模分析、实时数据处理、存算解耦、混合负载、数据联邦、异构服务器混合部署等先进技术能力。

作为应用开发人员，您只需完成驱动安装，即可基于 JDBC / ODBC 开发相关应用程序，连接并操作 Inceptor 中的数据，为帮助您更快熟悉开发流程，我们为您准备了文档和示例代码供您参考：

- 基于 JDBC 开发
- 基于 ODBC 开发

为设计更高效的业务系统，充分发挥数据库性能，在进行应用程序开发前，推荐阅读开发规范，了解数据库和 SQL 语句的设计规范。

此外，如需快速了解 Inceptor 的 SQL 语法及方言支持情况，见 SQL 参考。如您使用过程中遇到任何技术问题，可联系技术支持获取帮助。

6.2. 基于 JDBC 开发

JDBC（ Java 数据库连接）是一种用于执行 SQL 语句的 Java API，提供查询、更新数据库中数据的方法，工作流程如下：

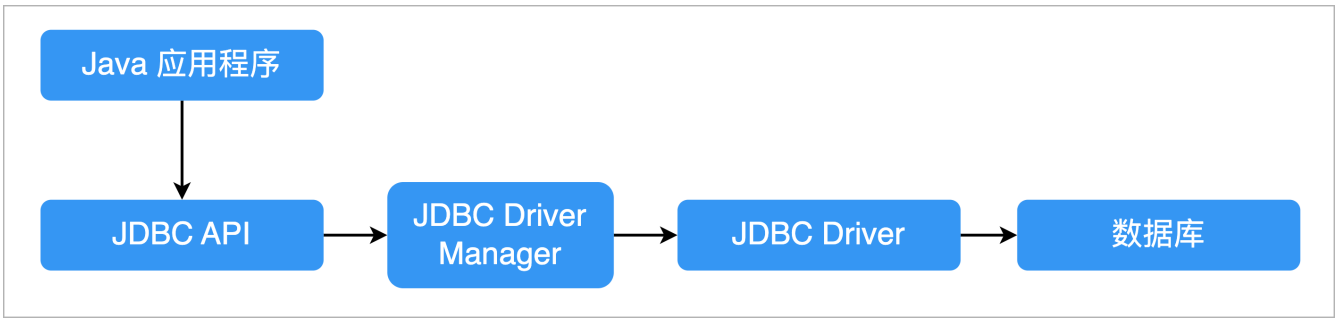


图 11. JDBC 工作流程

我们为您准备了丰富的教程，帮助您快速开发 Java 程序来连接和读写 Inceptor 数据库，您可以按序阅读，也可以选择阅读所需文档：

- 首次尝试开发：您可以阅读开发流程了解基本开发步骤，然后基于示例代码简单修改即可。
- 熟悉开发流程：您可以阅读 JDBC 参考，快速查找相关接口的使用说明。

6.2.1. 下载 JDBC 驱动

JDBC 或 ODBC 驱动程序用于连接 Transwarp Inceptor 数据库，在开发应用程序前，您需要登录 Transwarp Manager 平台下载相关驱动程序。

6.2.1.1. 操作步骤

1. 登录 Transwarp Manager 平台。

2. 在集群页面的 **Transwarp Inceptor** 区域框中，找到并单击运行中的 Quark 组件。
3. 单击页面右上角的 ··· 图标，然后选择下载 JDBC 或 ODBC 驱动。

图 12. 下载 JDBC 驱动

4. 在弹出的面板中，单击**下载**（推荐下载 All-in-one 包）。

6.2.2. 常用 API 介绍

JDBC API 提供来自 Java 编程语言的通用数据访问，主要位于 java.sql 包中，定义了访问数据库的接口和类，常用的 API 如下：

- DriverManager：负责加载各种不同驱动程序（Driver），并根据不同的请求，向调用者返回相应的数据库连接（Connection）。
- Driver：驱动程序，负责将自身加载到 DriverManager 中去，并处理相应的请求并返回相应的数据库连接（Connection）。
- Connection：数据库连接，负责进行与数据库间的通讯，SQL 执行以及事务处理都是在某个特定 Connection 环境中进行的。可以产生用以执行 SQL 的 Statement。
- Statement：用以执行 SQL 查询和更新（针对静态 SQL 语句和单次执行）。
- PreparedStatement：用以执行包含动态参数的 SQL 查询和更新（在服务器端编译，允许重复执行以提高效率）。
- CallableStatement：用以调用数据库中的存储过程。
- SQLException：代表在数据库连接的创建和关闭和 SQL 语句的执行过程中发生了例外情况（即错误）。

6.2.3. 开发流程

Inceptor 支持完整的 SQL 标准语法，可兼容 Oracle、IBM DB2、Teradata 方言，本章节中，我们将介绍如何开发 Java 应用程序对 Inceptor 数据进行操作，通用开发流程如下：

在开发 Java 应用程序前，请确保 JDK 版本为 1.8 及以上。

图 13. JDBC 开发流程

接下来，我们将介绍各步骤的详细操作及示例代码供您参考：

1. 在项目中加载您下载的驱动包，以 IntelliJ IDEA 开发工具为例，流程如下：
   a. 在 IntelliJ IDEA 顶部，选择 **File** > **Project Structure**。
   b. 在弹出的对话框单击 **Modules** 页签，然后单击右侧的 **Dependencies** 页签。
   c. 单击 + > **JARs or Directories…**，然后选择您下载的驱动以完成加载。

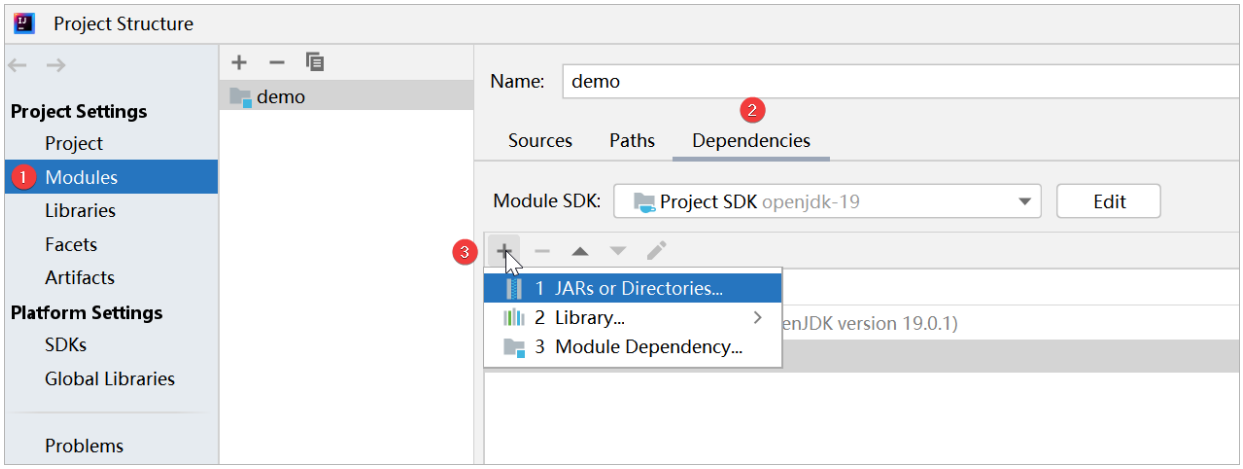


图 14. 加载驱动

2. 开始编写项目中的 Java 代码，首先我们在代码中加载 jar 包 和 JDBC 驱动。

```
// 导入 jar 包
import java.sql.*;
// 加载 JDBC 驱动
Class.forName("org.apache.hive.jdbc.HiveDriver");
```

3. 在 DriverManager 中，使用 `DriverManager.getConnection()` 方法获取数据库连接（Connection）。

> 下述示例代码以连接免认证的 Inceptor 数据库为例，如数据库开启了认证，请参考示例代码章节。

```
String jdbcURL = "
jdbc:transwarp2://<host_ip>:<port>/<database_name>;<JDBC_Conf>?<Inceptor_Conf>";
Connection conn = DriverManager.getConnection(jdbcURL);
```

<host_ip>：数据库的连接地址，即 Quark 组件的服务地址，您可以登录 Transwarp Manager 平台，找到 Quark 组件并获取其服务地址。

<port>：数据库服务端口，默认为 10000。

<database_name>：要连接的数据库名。

<JDBC_Conf>：JDBC 连接参数，多个参数间通过英文分号（;）分隔，常见参数如下：

limitSelect：限制 select 的结果集的行数，例如：jdbc:transwarp2://localhost:10000/default;limitSelect=0

fetchSize：修改数据库交互时，每次提取出来的记录行总数，例如：jdbc:transwarp2://localhost:10000/default;fetchSize=2000

autoReconnect：连接出错后自动重连，例如：jdbc:transwarp2://localhost:10000/default;autoReconnect=true

<Inceptor_Conf>：Inceptor 连接参数，多个参数间通过英文分号（;）分隔，常用参数为 **ngmr.execute.mode**，即设置 Inceptor 的执行模式。

例如设置为 local 模式，适用于低延时、高并发、参与计算数据量少的场景：jdbc:transwarp2://localhost:10000/default;limitSelect=1?ngmr.execute.mode=local

4. 提交 SQL 语句至数据库执行，可使用 `Statement` 或 `PreparedStatement` 构建。

   Statement 构建方式

```
Statement stmt = conn.createStatement();
stmt.executeUpdate( "insert into table_test values(a,a)" );
stmt.executeUpdate( "delete from table_test where col1=a" );
stmt.executeQuery( "select * from table_test" );
stmt.execute( "begin transaction" );
```

   PreparedStatement 构建方式

```
ps = conn.prepareStatement( "SELECT i.*, j.* FROM Omega i, Zappa j WHERE i = ? AND j = ?" );
ps.setString(1, "Poor Yorick");
ps.setInt(2, 8008);
rs = ps.executeQuery();
```

5. （可选）通过 JDBC 执行事务。

> 默认情况下，处于自动提交模式，即每执行一条 SQL 都是作为事务自动执行的，您也可以使用 begin transaction，可以在一个事务中执行多条语句，事务以 commit 结束。

```
stmt.execute("begin transaction");
stmt.executeUpdate(SQL 1);
stmt.executeUpdate(SQL 2);
stmt.executeUpdate(SQL 3);
stmt.execute("commit");
```

6. 使用 `ResultSet.getXXX()` 方法获取数据结果。

```
int numColumns = rs.getMetaData().getColumnCount();
while ( rs.next() ) {
for ( int i = 1 ; i <= numColumns ; i++ ) {
System.out.println( "COLUMN " + i + " = " + rs.getObject(i) );
}
}
```

7. 关闭所有的数据库资源。

```
rs.close();
stmt.close();
conn.close();
```

6.2.4. 异常处理

异常处理允许我们以受控的方式处理异常情况，而不是直接退出程序。例如数据库操作失败，JDBC 将抛出一个 SQLException，此类异常很少能够恢复，此时需要尽可能详细地打印异常日志，将 SQLException 翻译成应用程序领域相关的异常（非强制处理异常），然后回滚数据库并通知用户。

6.2.5. 示例代码

为帮助您更好地上手 Inceptor 相关应用程序的开发，下文提供了不同验证模式下，与 Inceptor 进行交互的相关代码功能参考。

- 免认证模式
- Kerberos 认证模式
- LDAP 认证模式

6.2.5.1. 与免认证的 Inceptor 交互

当 Inceptor 未开启认证时，您可以参考下述代码连接并操作 Inceptor，相关说明请参考代码中的注释。

```
package demo;
import java.sql.*;
public class JDBCExample {

  // 驱动程序
  private static String driverName = "org.apache.hive.jdbc.HiveDriver";

  public static void main(String[] args) throws SQLException {
    try {
      Class.forName(driverName);
      // 加载驱动
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
      System.exit(1);
    }
    // 本示例连接至 default 数据库，您也可以指定要连接的数据库
    String jdbcURL = "jdbc:transwarp2://172.18.130.13:10000/default";
    Connection conn = DriverManager.getConnection(jdbcURL);
        // 执行查询并获取结果
    Statement  stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select * from table_jdbc");
    ResultSetMetaData rsmd = rs.getMetaData();
    int size = rsmd.getColumnCount();
    while(rs.next()) {
      StringBuffer value = new StringBuffer();
      for(int i = 0; i < size; i++) {
        value.append(rs.getString(i+1)).append("\t");
      }
    // 获取输出结果并关闭连接
    System.out.println(value.toString());
    }
    rs.close();
    stmt.close();
    conn.close();
  }
}
```

6.2.5.2. 与启用 Kerberos 认证的 Inceptor 交互

当 Inceptor 启用了 Kerberos 认证时，您可以参考下述代码连接并操作 Inceptor。

```java
import java.sql.*;

public class JDBCExample {
  // 驱动程序
  private static String driverName = "org.apache.hive.jdbc.HiveDriver";

  public static void main(String[] args) throws SQLException {
    try {
      Class.forName(driverName);
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
      System.exit(1);
    }

    // 建立连接
    String jdbcURL = "jdbc:transwarp2://localhost:10000/default;principal=hive/node@TDH;"+ ①
        "authentication=kerberos;"+
        "kuser=user_principal;"+ ②
        "keytab=keytab_path;"+ ③
        "krb5conf=krb5.conf_path"; ④
    Connection conn = DriverManager.getConnection(jdbcURL);
    // 执行查询并获取结果
    Statement  stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select c1, c2 from table1");
    ResultSetMetaData rsmd = rs.getMetaData();
    int size = rsmd.getColumnCount();
    while(rs.next()) {
      StringBuffer value = new StringBuffer();
      for(int i = 0; i < size; i++) {
        value.append(rs.getString(i+1)).append("\t");
      }
    // 获取输出结果并关闭连接
    System.out.println(value.toString());
    }
    rs.close();
    stmt.close();
    conn.close();
  }
}
```

① 连接至开启 Kerberos 认证的 Inceptor 的 JDBC 连接串，格式为 `jdbc:transwarp2://<host_ip>:10000/<database_name>;principal=<username>/node@TDH;`

- <host_ip>：数据库的连接地址。
- <database_name>：连接的数据库名。
- <username>：Kerberos 中 Principal 的用户名，默认为 hive。

② kuser：将值设置为 Inceptor server的用户的 principal。

③ keytab_path：将值设置为 user_principal 对应 keytab 文件的绝对路径 。

目前尚不支持在应用程序中使用 principal + 密码的认证方式。

④ krb5.conf_path：将值设置为 Kerberos 配置文件 krb5.conf 的绝对路径 。

6.2.5.3. 与启用 LDAP 认证的 Inceptor 交互

当 Inceptor 启用了 LDAP 认证时，您可以参考下述代码连接并操作 Inceptor。

```java
import java.sql.*;
public class JDBCExample {

  // 驱动程序
  private static String driverName = "org.apache.hive.jdbc.HiveDriver";

  public static void main(String[] args) throws SQLException {
    try {
      Class.forName(driverName);
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
      System.exit(1);
    }

    // 连接数据库，本示例连接至 default 数据库，您也可以指定要连接的数据库
    String jdbcURL = "jdbc:transwarp2://localhost:10000/default";
    // 访问 Inceptor 的用户名。
    String user = "user";
    // 用户名对应的密码。
    String password = "password";
    Connection conn = DriverManager.getConnection(jdbcURL, user, password);
    // 执行查询并获取结果
    Statement  stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select c1, c2 from table1");
    ResultSetMetaData rsmd = rs.getMetaData();
    int size = rsmd.getColumnCount();
    while(rs.next()) {
      StringBuffer value = new StringBuffer();
      for(int i = 0; i < size; i++) {
        value.append(rs.getString(i+1)).append("\t");
      }
    // 获取输出结果并关闭连接
    System.out.println(value.toString());
    }
    rs.close();
    stmt.close();
    conn.close();
  }
}
```

6.2.6. JDBC 接口参考

本章节介绍 JDBC 4 支持的接口及返回类型功能参考，包含：

- java.sql.Connection
- java.sql.CallableStatement
- java.sql.DatabaseMetaData
- java.sql.Driver
- java.sql.PreparedStatement
- java.sql.ResultSet
- java.sql.ResultSetMetaData
- java.sql.Statement
- javax.sql.ConnectionPoolDataSource
- javax.sql.DataSource
- javax.sql.PooledConnection

6.2.6.1. java.sql.Connection

数据库连接接口，相关方法名和返回值类型如下。

方法名	返回值类型
close()	void

方法名	返回值类型
commit()	void
createStatement()	Statement
getAutoCommit()	boolean
getClientInfo()	Properties
getClientInfo(String name)	String
getTransactionIsolation()	int
isClosed()	boolean
isReadOnly()	boolean
prepareStatement(String sql)	PreparedStatement
rollback()	void
setAutoCommit(boolean autoCommit)	void
setClientInfo(Properties properties)	void
setClientInfo(String name,String value)	void

6.2.6.2. java.sql.CallableStatement

存储过程执行接口，相关方法名和返回值类型如下。

方法名	返回值类型
registerOutParameter(int parameterIndex, int type)	void
wasNull()	boolean
getString(int parameterIndex)	String
getBoolean(int parameterIndex)	boolean
getByte(int parameterIndex)	byte
getShort(int parameterIndex)	short
getInt(int parameterIndex)	int
getLong(int parameterIndex)	long
getFloat(int parameterIndex)	float
getDouble(int parameterIndex)	double
getBigDecimal(int parameterIndex)	BigDecimal
getBytes(int parameterIndex)	byte[]
getDate(int parameterIndex)	Date
getTime(int parameterIndex)	Time
getTimestamp(int parameterIndex)	Timestamp
getObject(int parameterIndex)	Object

6.2.6.3. java.sql.DatabaseMetaData

数据库对象定义接口，相关方法名和返回值类型如下。

方法名	返回值类型
getTables(String catalog,String schemaPattern, String tableNamePattern, String[] types)	ResultSet
getColumns(String catalog,String schemaPattern, String tableNamePattern, String columnNamePattern)	ResultSet
getTableTypes()	ResultSet
getUserName()	String
isReadOnly()	boolean
nullsAreSortedHigh()	boolean
nullsAreSortedLow()	boolean
nullsAreSortedAtStart()	boolean
nullsAreSortedAtEnd()	boolean
getDatabaseProductName()	String
getDatabaseProductVersion()	String
getDriverName()	String
getDriverVersion()	String
getDriverMajorVersion()	int
getDriverMinorVersion()	int
usesLocalFiles()	boolean
usesLocalFilePerTable()	boolean
supportsMixedCaseIdentifiers()	boolean
storesUpperCaseIdentifiers()	boolean
storesLowerCaseIdentifiers()	boolean
supportsMixedCaseQuotedIdentifiers()	boolean
storesUpperCaseQuotedIdentifiers()	boolean
storesLowerCaseQuotedIdentifiers()	boolean
storesMixedCaseQuotedIdentifiers()	boolean
supportsAlterTableWithAddColumn()	boolean
supportsAlterTableWithDropColumn()	boolean
supportsColumnAliasing()	boolean
nullPlusNonNullIsNull()	boolean
supportsConvert()	boolean

方法名	返回值类型
supportsConvert(int fromType, int toType)	boolean
supportsTableCorrelationNames()	boolean
supportsDifferentTableCorrelationNames()	boolean
supportsExpressionsInOrderBy()	boolean
supportsOrderByUnrelated()	boolean
supportsGroupBy()	boolean
supportsGroupByUnrelated()	boolean
supportsGroupByBeyondSelect()	boolean
supportsLikeEscapeClause()	boolean
supportsMultipleResultSets()	boolean
supportsMultipleTransactions()	boolean
supportsNonNullableColumns()	boolean
supportsMinimumSQLGrammar()	boolean
supportsCoreSQLGrammar()	boolean
supportsExtendedSQLGrammar()	boolean
supportsANSI92EntryLevelSQL()	boolean
supportsANSI92IntermediateSQL()	boolean
supportsANSI92FullSQL()	boolean
supportsIntegrityEnhancementFacility()	boolean
supportsOuterJoins()	boolean
supportsFullOuterJoins()	boolean
supportsLimitedOuterJoins()	boolean
isCatalogAtStart()	boolean
supportsSchemasInDataManipulation()	boolean
supportsSavepoints()	boolean
supportsResultSetHoldability(int holdability)	boolean
getResultSetHoldability()	int
getDatabaseMajorVersion()	int
getDatabaseMinorVersion()	int
getJDBCMajorVersion()	int
getJDBCMinorVersion()	int

6.2.6.4. java.sql.Driver

数据库驱动接口，相关方法名和返回值类型如下。

方法名	返回值类型
acceptsURL(String url)	boolean
connect(String url, Properties info)	Connection
jdbcCompliant()	boolean
getMajorVersion()	int
getMinorVersion()	int

6.2.6.5. java.sql.PreparedStatement

预处理语句接口，相关方法名和返回值类型如下。

方法名	返回值类型
clearParameters()	void
execute()	boolean
executeQuery()	ResultSet
excuteUpdate()	int
getMetaData()	ResultSetMetaData
setBoolean(int parameterIndex, boolean x)	void
setBigDecimal(int parameterIndex,BigDecimal x)	void
setByte(int parameterIndex, byte x)	void
setBytes(int parameterIndex, byte[]x)	void
setDate(int parameterIndex, Date x)	void
setDouble(int parameterIndex, doublex)	void
setFloat(int parameterIndex, float x)	void
setInt(int parameterIndex, int x)	void
setLong(int parameterIndex, long x)	void
setShort(int parameterIndex, short x)	void
setString(int parameterIndex, String x)	void
addBatch()	void
executeBatch()	int[]
clearBatch()	void

6.2.6.6. java.sql.ResultSet

执行结果集接口，相关方法名和返回值类型如下。

方法名	返回值类型
findColumn(String columnLabel)	int

方法名	返回值类型
getBigDecimal(int columnIndex)	BigDecimal
getBigDecimal(String columnLabel)	BigDecimal
getBoolean(int columnIndex)	boolean
getBoolean(String columnLabel)	boolean
getByte(int columnIndex)	byte
getBytes(int columnIndex)	byte[]
getByte(String columnLabel)	byte
getBytes(String columnLabel)	byte[]
getDate(int columnIndex)	Date
getDate(String columnLabel)	Date
getDouble(int columnIndex)	double
getDouble(String columnLabel)	double
getFloat(int columnIndex)	float
getFloat(String columnLabel)	float
getInt(int columnIndex)	int
getInt(String columnLabel)	int
getLong(int columnIndex)	long
getLong(String columnLabel)	long
getShort(int columnIndex)	short
getShort(String columnLabel)	short
getString(int columnIndex)	String
getString(String columnLabel)	String
getTime(int columnIndex)	Time
getTime(String columnLabel)	Time
getTimestamp(int columnIndex)	Timestamp
getTimestamp(String columnLabel)	Timestamp
isAfterLast()	boolean
isBeforeFirst()	boolean
isFirst()	boolean
next()	boolean

6.2.6.7. java.sql.ResultSetMetaData

描述 ResultSet 对象的相关信息，相关方法名和返回值类型如下。

方法名	返回值类型
getColumnCount()	int
getColumnName(int column)	String
getColumnType(int column)	int
getColumnTypeName(int column)	String

6.2.6.8. java.sql.Statement

SQL 语句接口，相关方法名和返回值类型如下。

方法名	返回值类型
close()	void
execute(String sql)	boolean
executeQuery(String sql)	ResultSet
executeUpdate(String sql)	int
getConnection()	Connection
getResultSet()	ResultSet
getQueryTimeout()	int
getUpdateCount()	int
isClosed()	boolean
setQueryTimeout(int seconds)	void
setFetchSize(int rows)	void
cancel()	void

6.2.6.9. javax.sql.ConnectionPoolDataSource

数据源连接池接口，相关方法名和返回值类型如下。

方法名	返回值类型
getLoginTimeout()	int
getLogWriter()	PrintWriter
getPooledConnection()	PooledConnection
getPooledConnection(String user,String password)	PooledConnection
setLoginTimeout(int seconds)	void
setLogWriter(PrintWriter out)	void

6.2.6.10. javax.sql.DataSource

数据源接口，相关方法名和返回值类型如下。

方法名	返回值类型
getConneciton()	Connection
getConnection(String username,String password)	Connection
getLoginTimeout()	int
getLogWriter()	PrintWriter
setLoginTimeout(int seconds)	void
setLogWriter(PrintWriter out)	void

6.2.6.11. javax.sql.PooledConnection

由连接池创建的连接接口，相关方法名和返回值类型如下。

方法名	返回值类型
addConnectionEventListener(ConnectionEventListener listener)	void
close()	void
getConnection()	Connection
removeConnectionEventListener(ConnectionEventListener listener)	void
addStatementEventListener(StatementEventListener listener)	void
removeStatementEventListener(StatementEventListener listener)	void

6.3. 基于 ODBC 开发

ODBC（开放数据库连接）是一种广泛接受的应用程序编程接口，使用结构化查询语言（SQL）作为数据库访问语言，不仅可以避免应用程序直接操作数据库，同时增强了应用的移植性和维护性，提升开发效率，典型组件及架构如下图所示：

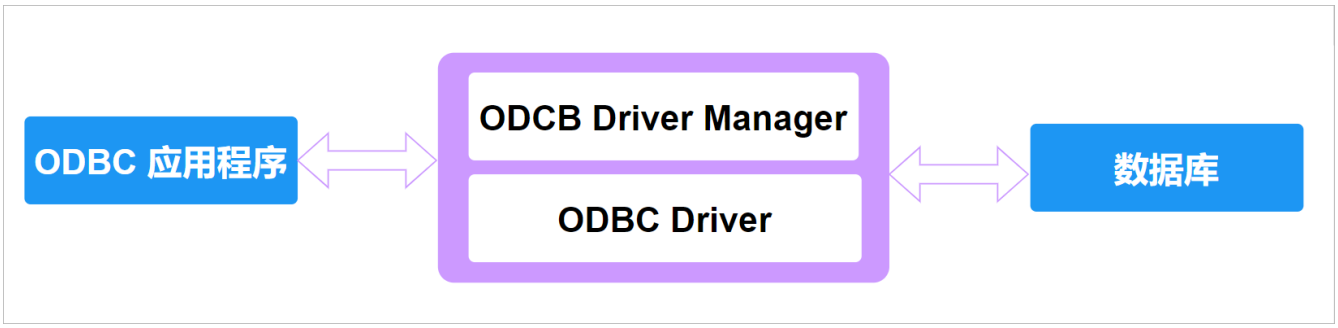


图 15. ODBC 程序典型架构

- ODBC 客户端：使用 ODBC API 与数据库服务器交互的应用程序。
- ODBC Driver：一个或一组包含针对特定数据库服务器的 ODBC API 实现的库。
- ODBC Driver Manager：一个或一组可供 ODBC 客户端用来与 ODBC Driver 交互的库。

我们为您准备了丰富的教程，帮助您快速开发应用程序来连接和读写 Inceptor 数据库，您可以按序阅读，

也可以选择阅读所需文档：

- 首次尝试开发：您可以阅读开发流程了解基本开发步骤，然后基于示例代码简单修改即可。
- 熟悉开发流程：您可以阅读 ODBC API 支持，快速查找相关接口的使用说明。

6.3.1. 环境准备

在正式开发应用程序前，您需要跟随下述步骤完成环境准备工作：

- 下载 ODBC 驱动
- 配置数据源

  Windows 平台

  Linux 平台

6.3.1.1. 下载驱动

ODBC 驱动程序用于连接 Transwarp Inceptor 数据库，在开发应用程序前，您需要登录 Transwarp Manager 平台下载相关驱动程序。

6.3.1.1.1. 操作步骤

1. 登录 Transwarp Manager 平台。
2. 在集群页面的 **Transwarp Inceptor 区域框**中，找到并单击运行中的 Quark 组件。
3. 单击页面右上角的 ··· 图标，然后选择下载 ODBC 驱动。
4. 在弹出的面板中，选择您的开发平台（如 Windows 64 位），然后单击**下载**。

图 16. 下载 ODBC 驱动

ODBC 驱动支持的平台如下：

Windows

CentOS 6/7

Ubuntu 20.04（X64 架构）

kylin v10（ARM X64 架构）

6.3.1.2. Windows 平台配置数据源

下载 ODBC 驱动后，您还需要参考本文流程，安装驱动并添加数据源配置，完成操作后即可开始程序的开发工作。

6.3.1.2.1. 操作步骤

1. 下载 ODBC 驱动。
2. 双击下载的 ODBC 驱动程序，跟随提示完成安装。
3. 按 **Win** 键，在弹出的页面中输入 **odbc**，单击结果中的 ODBC 数据源（64 位）。
4. 在弹出的界面中，单击用户 DSN 页签中的**添加**。

图 17. 添加 DSN

5. 在弹出的对话框中，双击 **Transwarp Inceptor ODBC Driver** 进行编辑。
6. 根据下述说明，完成数据源配置。

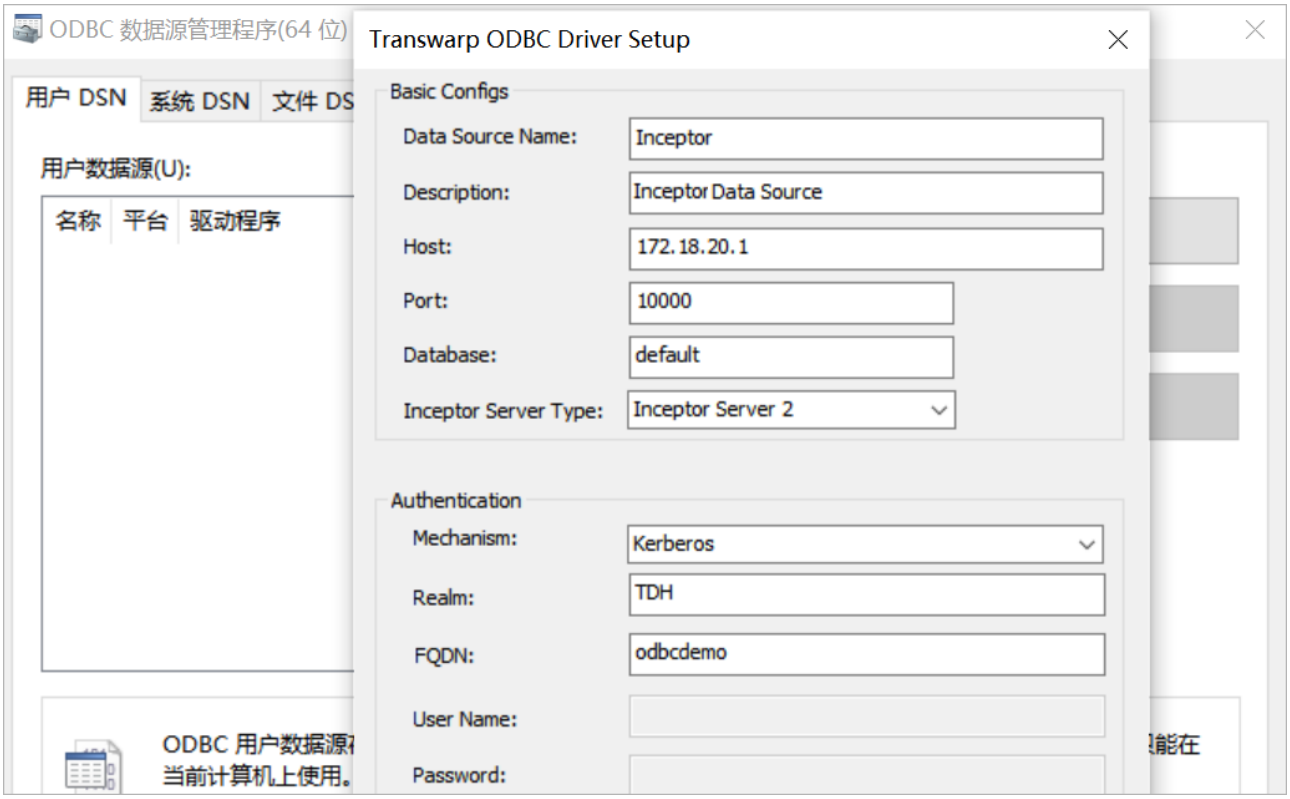


图 18. 配置数据源

Data Source Name：填写有业务意义的数据源名称。

Description：填写描述信息。

Host：数据库的连接地址，即 Quark 组件的服务地址，您可以登录 Transwarp Manager 平台，找到

Quark 组件并获取其服务地址。

Port：数据库服务端口，默认为 10000。

Database：要链接的数据库，默认为 default。

Inceptor Server Type：根据 Hive Server 版本选择，选择为 Inceptor Server 1 时，Mechanism 仅可设置为 NONE。

Mechanism：数据库认证方式，本案例选择为 Kerberos，此时需要填写 Realm 和 FQDN 信息，如选择为 **LDAP**，则需要填写 User Name 和 Password 信息。 关于 Inceptor 的认证方式配置，见 Guardian 手册。

7. 单击 **Test**，通过连接测试后单击 **OK**，然后单击**确定**。
8. （可选）当使用 ODBC 出错时，您可以在**跟踪**页签跟踪相关日志。

图 19. 跟踪日志

a. 跟踪完成后将 ODBC 的相关日志发送给 Transwarp 技术支持人员。

该日志为 ODBC Driver Manager 日志，由 Windows 系统本身提供，描述了应用层调用驱动的 API 记录及其返回值。

b. ODBC Driver 日志：由 ODBC 研发人员自己定义。

由于该操作将会较大程度影响 ODBC 性能，推荐仅在排查故障时使用。

6.3.1.2.2. 常见问题

- 问：如何获取 ODBC Driver 日志？

答：日志文件存放路径为 /tmp/odbc.log，如需调整路径，您可以打开 ODBC 驱动的安装目录中的 **log4cplus.properties** 文件，修改 log4cplus.appender.R.File 参数的值来调整日志存储路径。

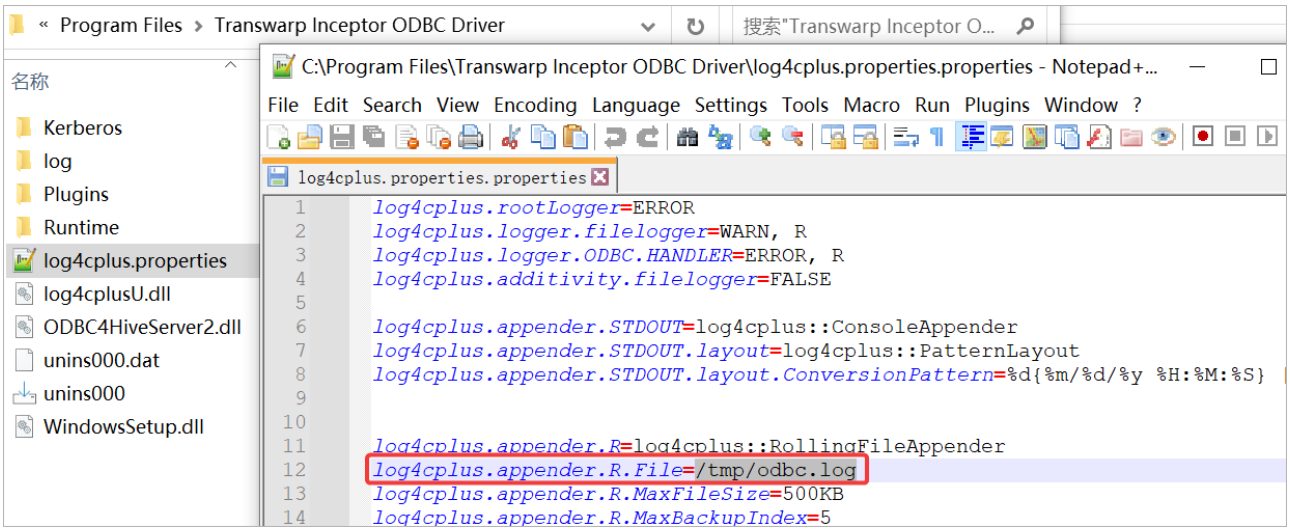


图 20. 调整日志路径

- 问：如何调整 ODBC Driver 的日志等级？

  答：默认级别为 **ERROR**，您可以打开 ODBC 驱动的安装目录中的 **log4cplus.properties** 文件，修改 **log4cplus.properties** 文件中 **log4cplus.rootLogger** 和 **log4cplus.logger.ODBC.HANDLER** 参数的值，例如改成 **INFO** 或 **DEBUG**。

6.3.1.3. Linux 平台配置数据源

下载 ODBC 驱动后，您还需要参考本文流程，安装驱动并添加数据源配置，完成操作后即可开始程序的开发工作。

6.3.1.3.1. 操作步骤

下述步骤以 CentOS 7（64位）操作系统为例演示操作流程，其他 Linux 平台操作流程类似。

1. 登录 Linux 设备，执行下述命令安装 unixODBC，用于管理 ODBC 驱动。

```
yum install unixODBC unixODBC-devel
```

2. 配置环境变量信息。

   a. 执行 vim ~/.bashrc 命令，然后按 a 进入编辑模式，在文件末尾追加下述配置信息。

```
export PATH=/usr/local/bin:${PATH}
export LD_LIBRARY_PATH=/usr/local/lib/:${LD_LIBRARY_PATH}
export ODBCSYSINI=/usr/local/inceptor
export ODBCINI=/usr/local/inceptor/odbc.ini
```

   b. 执行下述命令使配置生效。

```
source ~/.bashrc
```

3. 执行 odbcinst -j 命令，确认 unixODBC 是否安装成功，正常输出示例如下。

```
[root@local]# odbcinst -j
unixODBC 2.3.4
DRIVERS............: /etc/odbcinst.ini
SYSTEM DATA SOURCES: /etc/odbc.ini
FILE DATA SOURCES..: /etc/ODBCDataSources
USER DATA SOURCES..: /root/.odbc.ini
SQLULEN Size.......: 8
SQLLEN Size........: 8
SQLSETPOSIROW Size.: 8
```

4. 安装 ODBC 驱动。
   a. 下载 ODBC 驱动，然后将下载好的驱动文件放至 Linux 设备中。
   b. 以 CentOS 为例，执行下述命令安装驱动。

```
# 需根据下载的文件名称，替换下述命令中要安装的 rpm 包名称
rpm -ivh inceptor-connector-odbc-8.31.0-1.el7.x86_64.rpm
```

5. 执行 `vim /usr/local/inceptor/odbc.ini` 命令，然后按 **a** 进入编辑模式，根据下述说明配置 ODBC 的 DSN（数据源名称）信息。

   关于 Inceptor 的认证方式配置，见 Guardian 手册。

   配置信息已基于认证方式分类并提供了配置示例，以 LDAP 认证连接至 Inceptor 数据库为例，我们仅需配置 [transwarp-hs2-ldap] 中包含的参数，具体如下：

```
# 数据源名称
[transwarp-hs2-ldap]
# 连接的描述信息
Description = LDAP test
# 驱动名称，无需修改
Driver = Transwarp Inceptor ODBC Driver
# 数据库所属设备的 host 名称
Server = idc1
# Hive 版本，无需修改
Hive = Hive Server 2
# 数据库的连接地址，即 Quark 组件的服务地址，您可以登录 Transwarp Manager 平台，找到 Quark
组件并获取其服务地址。
Host = 172.16.1.212
# 数据库服务端口，默认为 10000
Port = 10000
# 要连接的数据库
Database = default
# 访问 Inceptor 数据库的用户名和密码
User = hive
Password = Inceptor@123
# 认证方式为 LDAP，无需修改
Mech = LDAP
```

6. 按下 **Esc** 键退出编辑模式，输入 **:wq** 并按回车键，保存配置并退出编辑界面。
7. 执行 `isql -v {数据源名称} {用户名} {密码}` 格式的命令测试 ODBC 连通性，正常连接的输出示例如下。

```
root@local]# isql -v transwarp-hs2-ldap hive Inceptor@123
+---------------------------------------+
| Connected!                            |
|                                       |
| sql-statement                         |
| help [tablename]                      |
| quit                                  |
|                                       |
+---------------------------------------+
```

6.3.1.3.2. 常见问题

- 问：通过 isql -v 命令测试 ODBC 连接时，遇到了一些报错信息，如何解决？

  答：可能遇到的错误信息及相关解决方案如下：

错误提示	解决方案
“Data source name not found, and no default driver specified”	环境变量设置不正确或未设置，请参考本文的步骤 2 设置环境变量。
“Connection refused”	连接请求被拒绝，请检查服务端的防火墙设置，确保端口可正常通信。
“No route to host”	服务器连接地址配置错误或防火墙引起的网络通信，需检查地址配置正确性和网络连通性。
“Can’t open lib ’/usr/local/lib64/libODBC4HiveServer2.so’ : file not found]”	执行 `ldd /usr/local/lib64/libODBC4HiveServer2.so`，查看输出结果中提示 `libsasl2.so.2= not found`，此时执行 `ln -sv /usr/lib64/libsasl2.so.3.0.0 /usr/lib64/libsasl2.so.2` 添加软链接即可解决
“SASL Other:No worthy mechd found , Segmentation faild”	执行命令 `yum install cyrus-sasl-gssapi cyrus-sasl-plain -y` 安装相关工具

- 问：如何获取 ODBC Driver 日志？

  答：日志文件存放路径为 /tmp/odbc.log，如需调整路径，您可以修改 /usr/local/inceptor/log4cplus.properties 文件中 log4cplus.appender.R.File 参数的值。

```
log4cplus.rootLogger=ERROR
log4cplus.logger.filelogger=WARN, R
log4cplus.logger.ODBC.HANDLER=ERROR, R
log4cplus.additivity.filelogger=FALSE

log4cplus.appender.STDOUT=log4cplus::ConsoleAppender
log4cplus.appender.STDOUT.layout=log4cplus::PatternLayout
log4cplus.appender.STDOUT.layout.ConversionPattern=%d{%m/%d/%y %H:%M:%S} [%t] %-5p %c{2} %%%x%% - %m [%l]%n


log4cplus.appender.R=log4cplus::RollingFileAppender
log4cplus.appender.R.File=/tmp/odbc.log
log4cplus.appender.R.MaxFileSize=500KB
log4cplus.appender.R.MaxBackupIndex=5
log4cplus.appender.R.layout=log4cplus::PatternLayout
log4cplus.appender.R.layout.ConversionPattern=%p %D{%Y-%m-%d %H:%M:%S} %F:%L "%m"%n
```

图 21. 调整日志路径

- 问：如何调整 ODBC Driver？

  答：默认级别为 **ERROR**，修改 **/usr/local/inceptor/log4cplus.properties** 文件，调整 **log4cplus.rootLogger** 和 **log4cplus.logger.ODBC.HANDLER** 参数的值，例如改成 **INFO** 或 **DEBUG**。

6.3.2. 开发流程

Inceptor 支持完整的 SQL 标准语法，可兼容Oracle、IBM DB2、Teradata 方言，本章节中，我们将介绍应用程序与 Inceptor 数据交互的通用流程：

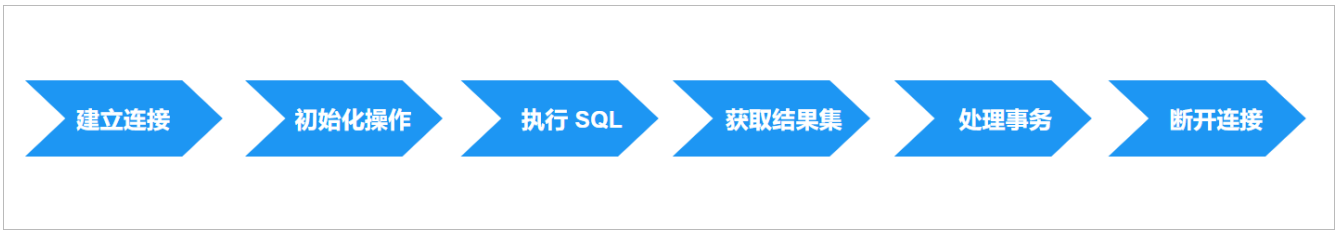


图 22. ODBC 开发流程

您可以基于业务需求灵活调整上述流程，各步骤说明如下：

> 在开发应用程序前，您需要手动完成 Windows/Linux 平台下 ODBC 驱动安装和数据配置。

1. 建立连接

   分配 Environment Handle 和 Connection Handle

   设置 ODBC 版本并连接到 Inceptor

   设置可选的 Connection Attribute
2. 初始化操作

   分配 Statement Handle

   设置可选的 Statement Attribute
3. 执行 SQL：准备 SQL 语句并执行。
4. 获取结果集：根据执行的 SQL 类型获取结果集：

   SELECT/SHOW/Catalog API：获取列信息、行信息等，将数据加入缓存

   DELETE/UPDATE/INSERT：获取受影响的行数
5. 处理事务：提交或回滚事务，如无事务操作则无需此步骤。
6. 断开连接：断开连接，释放 Connection Handle 和 Environment Handle。

6.3.3. 示例代码

为帮助您更好地上手 Inceptor 相关应用程序的开发，本章节提供了多种语言的示例代码，演示如何操作 Inceptor 数据库。

- C 示例代码
- C# 示例代码
- Python 示例代码
- Perl 示例代码
- Ruby 示例代码
- PHP 示例代码

6.3.3.1. C 示例代码

本文介绍 C 语言开发环境下，如何完成连接数据库、创建表、插入数据、查询数据和删除表的相关操作。

1. 本段代码为头文件 common.h，包括必要的头文件、变量声明、函数声明。

```
// 本示例展示如何连接数据库、创建表、插入数据、查询并打印结果、删除表
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#ifdef WIN32
#include <windows.h>
#endif
#include <sql.h>
#include <sqlext.h>
#ifdef WIN32
#define snprintf _snprintf
#endif
SQLHENV env;
SQLHDBC conn;
SQLCHAR str[1024];
HSTMT hstmt = SQL_NULL_HSTMT; //handle statement
char table_name[]="zzz";//your table_name. This program use $table_name as a sample
int rc;
//check execution statement
#define CHECK_STMT_RESULT(rc, msg, hstmt)   \
    if (!SQL_SUCCEEDED(rc)) \
    { \
        print_diag(msg, SQL_HANDLE_STMT, hstmt);    \
        exit(1);\
    }
//output info
void print_diag(char *msg, SQLSMALLINT htype, SQLHANDLE handle)
{
    char sqlstate[32];
    char message[1000];
    SQLINTEGER nativeerror;
    SQLSMALLINT textlen;
    SQLRETURN ret;
    if (msg)
        printf("%s\n", msg);
    ret = SQLGetDiagRec(htype, handle, 1, (SQLCHAR*)sqlstate, &nativeerror,
                        (SQLCHAR*)message, 256, &textlen);
    if (ret == SQL_INVALID_HANDLE)
        printf("Invalid handle\n");
    else if (ret != SQL_ERROR)
        printf("%s=%s\n", (CHAR *)sqlstate, (CHAR *)message);
}
void connect_database(); //connect to database.
void create_table();//create a table
void insert_into_table();//insert into table
void select_all();//select * from table and print result
void drop_table();//drop table
void disconnect();
```

2. 与 Inceptor 建立连接。

```
//connect to database.
void connect_database()
{
    SQLCHAR dsn[1024]; //dsn name
    SQLSMALLINT strl;
    snprintf((char*)dsn,sizeof(dsn), "DSN=your_inceptor_odbc_data_source_name");
// Allocate an environment
    rc =SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
    CHECK_STMT_RESULT(rc, "SQLExecDirect Allocate an environment  failed", hstmt);
// Register this as an application that expects 3.x behavior
    rc =SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void *) SQL_OV_ODBC3, 0);
    CHECK_STMT_RESULT(rc, "SQLExecDirectRegister this as an application failed", hstmt);
// Allocate a connection
    rc =SQLAllocHandle(SQL_HANDLE_DBC, env, &conn);
    CHECK_STMT_RESULT(rc, "SQLExecDirect Allocate a connection failed", hstmt);
// Connect to the driver
    rc =SQLDriverConnect(conn, NULL, dsn, SQL_NTS,
                         str, sizeof(str), &strl,
                         SQL_DRIVER_COMPLETE);
    CHECK_STMT_RESULT(rc, "SQLExecDirect Connect to the driver failed", hstmt);
    printf("Connected.\n");
}
```

3. 创建表 table_name。

```
//create a table
void create_table()
{
    if(NULL == table_name) {
        printf("WRONG! table_name can't be NULL\n");
        exit(1);
    }
    // Allocate a statement
    rc =SQLAllocHandle(SQL_HANDLE_STMT, conn, &hstmt);
    CHECK_STMT_RESULT(rc, "SQLExecDirect  Allocate a statement failed", hstmt);
//use odbc;
    rc =SQLExecDirect(hstmt, (SQLCHAR *) "use odbc", SQL_NTS);
    CHECK_STMT_RESULT(rc, "SQLExecDirect Execute a sql failed", hstmt);
// Execute a sql
    char create_table_sql[200];
    snprintf(create_table_sql,sizeof(create_table_sql),"create table if not exists %s(i int, s
string) stored as holodesk",table_name); //DSN=[your dsn description]
    rc =SQLExecDirect(hstmt, (SQLCHAR *)create_table_sql, SQL_NTS);
    CHECK_STMT_RESULT(rc, "SQLExecDirect Execute a sql failed", hstmt);
//  Free ODBC handles
    rc =SQLFreeStmt(hstmt, SQL_CLOSE);
    CHECK_STMT_RESULT(rc, "SQLExecDirect  Free ODBC handles failed", hstmt);
    printf("Table %s created.\n",table_name);
}
```

4. 向表中插入数据。

```
//insert into table
void insert_into_table()
{
    // Allocate a statement
    rc =SQLAllocHandle(SQL_HANDLE_STMT, conn, &hstmt);
    CHECK_STMT_RESULT(rc, "SQLExecDirect  Allocate a statement failed", hstmt);
    char insert_sql1[100];
    snprintf(insert_sql1,sizeof(insert_sql1),"insert into %s(i,s)  values(1,'foo')",table_name);
    char insert_sql2[100];
    snprintf(insert_sql2,sizeof(insert_sql2),"insert into %s(i,s)  values(2,'bar')",table_name);
    /* insert values*/
    rc = SQLExecDirect(hstmt, (SQLCHAR *) insert_sql1, SQL_NTS);
    CHECK_STMT_RESULT(rc, "SQLExecDirect insert into table failed", hstmt);
    rc = SQLExecDirect(hstmt, (SQLCHAR *) insert_sql2, SQL_NTS);
    CHECK_STMT_RESULT(rc, "SQLExecDirect insert into table failed", hstmt);
    printf("Insert values into table %s done.\n",table_name);
}
```

5. 查询并打印结果集。

```
//select * from table and print result
void select_all()
{
    SQLINTEGER longvalue;
    SQLLEN      indLongvalue;
    char    charvalue[100];
    SQLLEN  indCharvalue;
    // Allocate a statement
    rc =SQLAllocHandle(SQL_HANDLE_STMT, conn, &hstmt);
    CHECK_STMT_RESULT(rc, "SQLExecDirect  Allocate a statement failed", hstmt);
    /* bind column1 */
    rc = SQLBindCol(hstmt, 1, SQL_C_LONG, &longvalue, 0, &indLongvalue);
    CHECK_STMT_RESULT(rc, "SQLBindCol failed", hstmt);
    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, &charvalue, sizeof(charvalue), &indCharvalue);
    CHECK_STMT_RESULT(rc, "SQLBindCol failed", hstmt);
    char select_sql[100];
    snprintf(select_sql,sizeof(select_sql),"select * from %s",table_name);
    rc = SQLExecDirect(hstmt, (SQLCHAR *) select_sql, SQL_NTS);
    CHECK_STMT_RESULT(rc, "SQLExecDirect select failed", hstmt);
    printf("Result set:\n");
    while(1)
    {
        rc = SQLFetch(hstmt);
        if (rc == SQL_NO_DATA)
            break;
        if (rc == SQL_SUCCESS)
        {
            printf("%ld %s\n", (long) longvalue, charvalue);
        }
        else
        {
            print_diag("SQLFetch failed", SQL_HANDLE_STMT, hstmt);
            //exit(1);//commented by yangyao
        }
    }
    rc = SQLFreeStmt(hstmt, SQL_CLOSE);
    CHECK_STMT_RESULT(rc, "SQLFreeStmt failed", hstmt);
    printf("\nSelect fropm table %s done.\n",table_name);
}
```

6. 删除表 table_name。

```
//drop table
void drop_table()
{
    // Allocate a statement
    rc =SQLAllocHandle(SQL_HANDLE_STMT, conn, &hstmt);
    CHECK_STMT_RESULT(rc, "SQLExecDirect  Allocate a statement failed", hstmt);
    //drop sql statement
    char drop_sql[100];
    snprintf(drop_sql,sizeof(drop_sql),"drop table %s",table_name);
    //execute drop_sql
    rc = SQLExecDirect(hstmt, (SQLCHAR *) drop_sql, SQL_NTS);
    CHECK_STMT_RESULT(rc, "SQLExecDirect drop failed", hstmt);
    printf("Drop table %s done.\n",table_name);
}
```

程序所使用的 main 函数如下：

```
#include "common.h"
void main()
{
    connect(); //connect to database.
    create_table();//create a table
    insert_into_table();//insert into table
    select_all();//select * from table and print result
    drop_table();//drop table
}
```

6.3.3.2. C# 示例代码

本文介绍 C# 语言开发环境下，如何完成连接数据库、创建表、插入数据、查询数据和删除表的相关操作。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Odbc;
namespace catt1{
    class Program{
        static void Main(string[] args){
            try{
                //connect odbc by specifying DSN name
                OdbcConnection connection = new
OdbcConnection("DSN=your_inceptor_odbc_data_source_name");
                connection.Open();
                //create table
                string createTable = "create table tt(c1 int , c2 int) stored as holodesk";
                OdbcCommand command = new OdbcCommand(createTable, connection);
                command = new OdbcCommand(createTable, connection);
                command.ExecuteNonQuery();
                //insert data
                string insertTable = "insert into tt(c1,c2) values(?,?)";
                command = new OdbcCommand(insertTable, connection);
                command.CommandType = System.Data.CommandType.Text;
                command.Parameters.AddWithValue("@c1", 1);
                command.Parameters.AddWithValue("@c2", 2);
                command.ExecuteNonQuery();
                //select data with DataReader
                string selectTalbe = " select * from tt";
                command = new OdbcCommand(selectTalbe, connection);
                OdbcDataReader reader = command.ExecuteReader();
                while (reader.Read()){
                    Console.Out.Write(reader[0].ToString() + "\t");
                    Console.Out.WriteLine(reader[1].ToString());
                }
                //select data with OdbcDataAdapter
                OdbcDataAdapter da = new OdbcDataAdapter(selectTalbe, connection);
                System.Data.DataSet ds = new System.Data.DataSet();
                da.Fill(ds, "tt");
                //drop table
                string dropTable = "drop table if exists tt";
                command = new OdbcCommand(dropTable, connection);
                command.ExecuteNonQuery();
                //close connection
                connection.Close();
            }catch (Exception ex){
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

6.3.3.3. Python 示例代码

本文介绍 Python 语言开发环境下，如何完成连接数据库、创建表、插入数据、查询数据和删除表的相关操作。

> 使用该代码前，请先安装 pyodbc 模块。

```
import pyodbc
#connect odbc
cnxn = pyodbc.connect("DSN=your_inceptor_odbc_data_source_name")

# Python 2.7,  避免中文乱码
cnxn.setdecoding(pyodbc.SQL_CHAR, encoding='utf-8')
cnxn.setdecoding(pyodbc.SQL_WCHAR, encoding='utf-8')
cnxn.setencoding(str, encoding='utf-8')
cnxn.setencoding(unicode, encoding='utf-8')

# Python 3.x，避免中文乱码
cnxn.setdecoding(pyodbc.SQL_CHAR, encoding='utf-8')
cnxn.setdecoding(pyodbc.SQL_WCHAR, encoding='utf-8')
cnxn.setencoding(encoding='utf-8')

cursor = cnxn.cursor()
#create table
cursor.execute("create table tt(c1 int , c2 int) stored as holodesk")
#insert data
cursor.execute("insert into tt values (?,?)", 1,1)
cursor.execute("insert into tt values (?,?)", 2,2)
#select data
row = cursor.execute("select * from tt").fetchone()
print row
#drop table
cursor.execute("drop table tt")
cnxn.close()
```

6.3.3.4. Perl 示例代码

本文介绍 Perl 语言开发环境下，如何完成连接数据库、创建表、插入数据、查询数据和删除表的相关操作。

6.3.3.4.1. 环境准备

由于 Per 需要通过调用 DBI（数据库接口） 与 DBD::ODBC（ODBC 数据库驱动） 来访问 Inceptor，在执行示例代码前，您还需要编译安装这两个模块，以 CentOS 65 为例：

1. 安装 DBI。

```
# 安装 DBI 所需的依赖包
yum install perl perl-ExtUtils-CBuilder perl-ExtUtils-MakeMaker
# 下载并编译安装 DBI
wget ftp://ftp.funet.fi/pub/languages/perl/CPAN/modules/by-module/DBI/DBI-1.630.tar.gz
tar -xvzf DBI-1.630.tar.gz
cd DBI-1.630
perl Makefile.PL
make
make install
```

2. 安装 DBD::ODBC。

```
wget ftp://ftp.funet.fi/pub/languages/perl/CPAN/modules/by-module/DBD/DBD-ODBC-1.52.tar.gz
tar -xvzf DBD-ODBC-1.52.tar.gz
cd DBD-ODBC-1.52
perl Makefile.PL
make
make install
```

6.3.3.4.2. 示例代码

完成上述操作后，即可尝试运行下述 Perl 示例代码：

```
#!/usr/bin/perl
use strict;
use DBI;
my $data_source="DBI:ODBC:your_odbc_data_source_name";
my $dbh=DBI->connect($data_source,"","", {AutoCommit => 1,PrintError=>1,RaiseError=>1});
sub select_data(){
    print "select data ...\n";
    my $sth=$dbh->prepare("select * from tt");
    $sth->execute();
    my @row;
    while (@row = $sth->fetchrow_array) {  # retrieve one row
        print join(", ", @row), "\n";
    }
    $sth->finish();
}
sub insert_data(){
    print "insert data ...\n";
    my $sth=$dbh->prepare("insert into tt values (?,1)");
    my $p = (1,2,3);
    $sth->execute($p);
    $sth->finish();
}
sub create(){
    $dbh->do("create table tt(c1 int , c2 int) stored as holodesk");
}
sub drop(){
    $dbh->do("drop table tt");
}
sub main(){
    drop();
    create();
    insert_data();
    select_data();
    drop();
    $dbh->disconnect();
}
my $ret =main();
exit($ret);
```

6.3.3.5. Ruby 示例代码

本文介绍 Ruby 语言开发环境下，如何完成连接数据库、创建表、插入数据、查询数据和删除表的相关操作。

6.3.3.5.1. 环境准备

由于 Ruby 需要通过调用 DBI（数据库接口） 与 DBD::ODBC（ODBC 数据库驱动） 来访问 Inceptor，在执行示例代码前，您还需要编译安装这两个模块，以 CentOS 65 为例：

1. 安装 Ruby。

```
yum install ruby ruby-dev -y
```

2. 前往 Ruby-ODBC 下载站点，下载 0.9994 版本并将其放至 CentOS 65 设备 中。
3. 解压并安装 Ruby-ODBC。

```
# 解压文件并进入解压后目录
tar -zxvf ruby-odbc-0.994.tar.gz && cd ruby-odbc-0.994
# 编译安装，--with-odbc-dir 必须指向odbc driver manager的位置
# 例如 driver manager 的库为 /usr/lib/x86_64-linux-gnu/libodbc.so.1，则该选项如下
ruby ext/extconf.rb --with-odbc-dir=/usr/lib/x86_64-linux-gnu
```

4. 安装 DBI 和 DBD::ODBC。

```
sudo gem install dbi
sudo gem install dbd-odbc
```

6.3.3.5.2. 示例代码

完成上述操作后，即可尝试运行下述 Ruby 示例代码：

```
# inceptor_odbc.rb - simple Incepotr script using Ruby DBI module
 require "dbi"
 begin
     # connect to the MySQL server
     dbh = DBI.connect("dbi:ODBC:your_inceptor_data_source_name", "", "")
     # create transaction table
     dbh.do("create table rubytest(c1 int , c2 int) stored as holodesk")
     #insert table
     sth = dbh.prepare("INSERT INTO rubytest (c1,c2) VALUES(?,1)")
     sth.execute(2);
     sth.finish
     #select data
     sth = dbh.execute("SELECT * FROM rubytest")
     sth.each do |row|
        printf "c1: %d, c2: %d/n", row[0], row[1]
     end
     sth.finish
     #drop table
     dbh.do("drop table rubytest")
 rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code: #{e.err}"
     puts "Error message: #{e.errstr}"
 ensure
     # disconnect from server
     dbh.disconnect if dbh
 end
```

6.3.3.6. PHP 示例代码

本文介绍 PHP 语言开发环境下，如何完成连接数据库、创建表、插入数据、查询数据和删除表的相关操作。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <title>PHP and ODBC: XHTML Example 1</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    </head>
    <body>
    <?php
        $conn = odbc_connect("transwarp-hs2-lda", "", ""); // dsn name, user, password
        if (!($conn)) {
          echo "<p>Connection to DB via ODBC failed: ";
          echo odbc_errormsg ($conn );
          echo "</p>\n";
        }
        $create_table="create table phptest(c1 int , c2 int) stored as holodesk";
        $success = odbc_do($conn,"drop table if exists phptest");
    $success = odbc_do($conn,$create_table);
        $success = odbc_exec($conn,"insert into phptest values(1, 2)");
        $rs = odbc_exec($conn,"select * from phptest");
        echo "<table><tr>";
        echo "<th>Test</th></tr>";
        while (odbc_fetch_row($rs))
         {
         $result = odbc_result($rs,"c1");
         echo "<tr><td>c1: $result</td></tr>";
        }
        odbc_do($conn,"drop table if exists phptest");
        odbc_close($conn);
        echo "</table>";
    ?>
    </body>
</html>
```

6.3.4. ODBC API 支持

Inceptor 支持 3.x 版本的 ODBC API，本文介绍支持的 API 列表供您参考，详细的用法介绍和一致性级别说明，见 Microsoft ODBC API 参考。

- 连接数据源
- 获取驱动/数据源信息
- 设置/获取驱动属性
- 设置/获取描述符字段
- 准备 SQL 请求
- 提交请求
- 获取结果及其相关信息
- 获取数据源的系统表信息
- 终止语句
- 终止连接

6.3.4.1. 连接数据源

API	一致性级别	说明
SQLAllocHandle	ISO 92	分配环境、连接、语句或描述符句柄。
SQLConnect	ISO 92	在驱动程序和数据源间建立连接，连接句柄引用与数据源的连接的所有信息的存储，包括状态、事务状态和错误信息。
SQLDriverConnect	ODBC	可替代 SQLConnect，使用连接字符串或对话框连接到数据源。
SQLBrowseConnect	ODBC	发现和枚举连接到数据源所需的特性和属性值。
SQLAllocEnv	已弃用	由 SQLAllocHandle 取代。
SQLAllocConnect	已弃用	由 SQLAllocHandle 取代。
SQLAllocStmt	已弃用	由 SQLAllocHandle 取代。

6.3.4.2. 获取驱动/数据源信息

API	一致性级别	说明
SQLDataSources	ISO 92	返回数据源的相关信息，由驱动程序管理器实现。
SQLDrivers	ODBC	返回驱动程序说明和属性信息，由驱动程序管理器实现。
SQLGetInfo	ISO 92	返回特定驱动程序和数据源的信息。
SQLGetFunctions	ISO 92	返回相关驱动程序是否支持函数。
SQLGetTypeInfo	ISO 92	返回数据源支持的数据类型信息。

6.3.4.3. 设置/获取驱动属性

API	一致性级别	说明
SQLSetConnectAttr	ISO 92	设置连接属性。
SQLGetConnectAttr	ISO 92	返回连接属性的设置。
SQLSetEnvAttr	ISO 92	设置环境属性。
SQLGetEnvAttr	ISO 92	返回环境属性的设置。

API	一致性级别	说明
SQLSetStmtAttr	ISO 92	设置语句属性。
SQLGetStmtAttr	ISO 92	返回语句属性的设置。
SQLSetConnectOption	已弃用	由 SQLSetConnectAttr 取代。
SQLGetConnectOption	已弃用	由 SQLGetConnectAttr 取代。
SQLSetStmtOption	已弃用	由 SQLSetStmtAttr 取代。
SQLGetStmtOption	已弃用	由 SQLGetStmtAttr 取代。
SQLParamOptions	已弃用	由 SQLSetStmtAttr 取代。

6.3.4.4. 设置/获取描述符字段

API	一致性级别	说明
SQLGetDescField	ISO 92	获取单个描述符字段的值。
SQLGetDescRec	ISO 92	获取多个描述符字段的值。
SQLSetDescField	ISO 92	设置单个描述符字段。
SQLSetDescRec	ISO 92	设置多个描述符字段。
SQLCopyDesc	ISO 92	将描述符信息从一个描述符句柄复制到另一个描述符句柄。

6.3.4.5. 准备 SQL 请求

API	一致性级别	说明
SQLPrepare	ISO 92	准备稍后执行的 SQL 语句。
SQLBindParameter	ODBC	将缓冲区绑定到 SQL 语句中的参数标记。
SQLGetCursorName	ISO 92	返回与指定语句句柄关联的游标名。
SQLSetCursorName	ISO 92	设置游标名。
SQLSetScrollOptions	已弃用	由 SQLGetInfo 和 SQLSetStmtAttr 取代。
SQLSetParam	已弃用	由 SQLBindParameter 取代。

6.3.4.6. 提交请求

API	一致性级别	说明
SQLExecute	ISO 92	执行已准备的 SQL 语句。
SQLExecDirect	ISO 92	执行 SQL 语句。
SQLNativeSql	ODBC	返回由驱动程序翻译的 SQL 语句文本。
SQLDescribeParam	ODBC	返回语句中的参数信息，Connector/ODBC不支持，返回结果仅为参考。
SQLNumParams	ISO 92	返回 SQL 语句中参数的数量。
SQLParamData	ISO 92	与 SQLPutData 结合使用，在执行时提供参数数据。

API	一致性级别	说明
SQLPutData	ISO 92	发送参数的部分或全部数据值。
SQLTransact	已弃用	由 SQLEndTran 取代。

6.3.4.7. 获取结果及其相关信息

API	一致性级别	说明
SQLRowCount	ISO 92	返回受 INSERT、UPDATE、DELETE 请求语句所影响的行数。
SQLNumResultCols	ISO 92	返回结果集中列的数量。
SQLDescribeCol	ISO 92	返回结果集中列的信息。
SQLColAttribute	ISO 92	返回结果集中列的描述符信息。
SQLFetch	ISO 92	返回多行结果数据。
SQLFetchScroll	ISO 92	返回滚动的数据行。
SQLSetPos	ODBC	设置行集中的游标位置，并允许应用程序刷新行集中的数据或更新/删除结果集中的数据。
SQLBulkOperations	ODBC	执行批量插入和批量书签操作，包括通过书签进行更新、删除和提取。
SQLMoreResults	ODBC	确定是否有更多结果集可用，如有则执行初始化以对结果集进行处理。
SQLGetDiagField	ISO 92	返回诊断记录的单个字段的当前值，包含错误、警告和状态信息。
SQLGetDiagRec	ISO 92	返回诊断记录的多个字段的当前值，包含错误、警告和状态信息。
SQLBindCol	ISO 92	将应用程序数据缓冲区绑定到结果集中的列。
SQLGetData	ISO 92	返回结果集一行的部分/全部数据列数据。
SQLError	已弃用	由 SQLGetDiagRec 取代。
SQLColAttributes	已弃用	由 SQLColAttribute 取代。
SQLExtendedFetch	已弃用	由 SQLFetchScroll 取代。

6.3.4.8. 获取数据源的系统表信息

API	一致性级别	说明
SQLColumnPrivileges	ODBC	返回指定表的列和关联权限的列表。
SQLColumns	Open Group	返回指定表的列名。
SQLForeignKeys	ODBC	返回外键的列名。
SQLPrimaryKeys	ODBC	返回主键的列名。
SQLSpecialColumns	Open Group	返回指定表中列的信息，包含唯一标识表中行的最佳列集、事务更新行中的任何值时自动更新的列。
SQLStatistics	ISO 92	返回有关单个表及其关联的索引列表的统计信息。

API	一致性级别	说明
SQLTablePrivileges	ODBC	返回表列表与每个表关联的特权信息。
SQLTables	Open Group	返回指定数据源中的表名。
SQLProcedures	ODBC	返回指定数据源中的过程名。
SQLProcedureColumns	ODBC	返回过程的参数和结果集列。

6.3.4.9. 终止语句

API	一致性级别	说明
SQLFreeStmt	ISO 92	结束语句处理，放弃挂起的结果，释放与语句句柄关联的所有资源。
SQLCloseCursor	ISO 92	关闭已在语句句柄上打开的游标，同时丢弃挂起的结果。
SQLCancel	ISO 92	取消语句的处理。
SQLCancelHandle	ODBC	取消对连接或语句的处理。
SQLEndTran	ISO 92	提交或回滚事务。

6.3.4.10. 终止连接

API	一致性级别	说明
SQLDisconnect	ISO 92	关闭连接。
SQLFreeHandle	ISO 92	释放与特定环境、连接、语句或描述符句柄关联的资源。
SQLFreeConnect	已废弃	由 SQLFreeHandle 取代。
SQLFreeEnv	已废弃	由 SQLFreeHandle 取代。

6.3.5. API 映射关系说明

Inceptor 目前采用的 ODBC 驱动为 3.x 版本，当用其调用 2.x 版本的应用程序时，会根据下表自动映射并转换。

ODBC 2.x	ODBC 3.x
SQLAllocConnect	SQLAllocHandle
SQLAllocEnv ①	
SQLAllocStmt	
SQLBindParam	SQLBindParameter
SQLColAttributes	SQLColAttribute
SQLError	SQLGetDiagRec
SQLFreeConnect	SQLFreeHandle
SQLFreeEnv	
SQLFreeStmt	

ODBC 2.x	ODBC 3.x
SQLGetConnectOption	SQLGetConnectAttr
SQLGetStmtOption	SQLGetStmtAttr
SQLParamOptions	SQLSetStmtAttr
SQLSetConnectOption	SQLSetConnectAttr
SQLSetParam	SQLBindParameter
SQLSetScrollOption	SQLSetStmtAttr
SQLSetStmtOption	
SQLTransact	SQLEndTran

①：应用程序通过 ODBC 3.x 驱动调用该接口时，会自动调用 SQLSetEnvAttr 并将 SQL_ATTR_ODBC_VERSION 环境属性设置为 SQL_OV_ODBC2，再映射到 SQLAllocHandle。

6.3.6. Catelog API 支持说明

Catelog API 字符串参数分为四种不同类型的：普通参数（OA）、模式值参数（PV）、标识符参数（ID）和值列表参数（VL），更多介绍，见Catelog API。下表列出了当前对 Catelog API 的支持情况：

函数	参数	SQL_时键入 ATTR_METADATA_ID = SQL_FALSE	是否支持	SQL_时键入 ATTR_METADATA_ID = SQL_TRUE	是否支持
SQLColumnPrivileges	CatalogName SchemaName TableName ColumnName	OA OA OA PV		ID ID ID ID	
SQLColumns	CatalogName SchemaName TableName ColumnName	OA PV PV PV		ID ID ID ID	
SQLForeignKeys	PKCatalogName PKSchemaName PKTableName FKCatalogName FKSchemaName FKTableName	OA OA OA OA OA OA		ID ID ID ID ID ID	
SQLPrimaryKeys	CatalogName SchemaName TableName	OA OA OA		ID ID ID	
SQLProcedureColumns	CatalogName SchemaName ProcName ColumnName	OA PV PV PV		ID ID ID ID	
SQLProcedures	CatalogName SchemaName ProcName	OA PV PV		ID ID ID	

函数	参数	SQL_时键入 ATTR_METADATA_ID = SQL_FALSE	是否支持	SQL_时键入 ATTR_METADATA_ID = SQL_TRUE	是否支持
SQLSpecialColumns	CatalogName SchemaName TableName	OA OA OA		ID ID ID	
SQLStatistics	CatalogName SchemaName TableName	OA OA OA		ID ID ID	
SQLTablePrivileges	CatalogName SchemaName TableName	OA PV PV		ID ID ID	
SQLTables	CatalogName SchemaName TableName TableType	PV PV PV VL		ID ID ID VL	

7. 资源与权限管理

7.1. 概述

Inceptor 通过多租户机制管理资源与权限，其中，租户是一个用户或一组用户的集合，而多租户技术致力于为用户提供共用的系统或程序组件，并确保用户间资源/数据的隔离性。本文将介绍 Transwarp Inceptor 多租户架构及其使用流程。

7.1.1. 多租户平台架构

Transwarp Inceptor 的多租户管理系统集账号管理、身份认证、授权与审计于一体，可对平台内的账号、数据和资源提供统一管理服务，帮助您提升数据安全性和资源隔离性、降低维护成本。

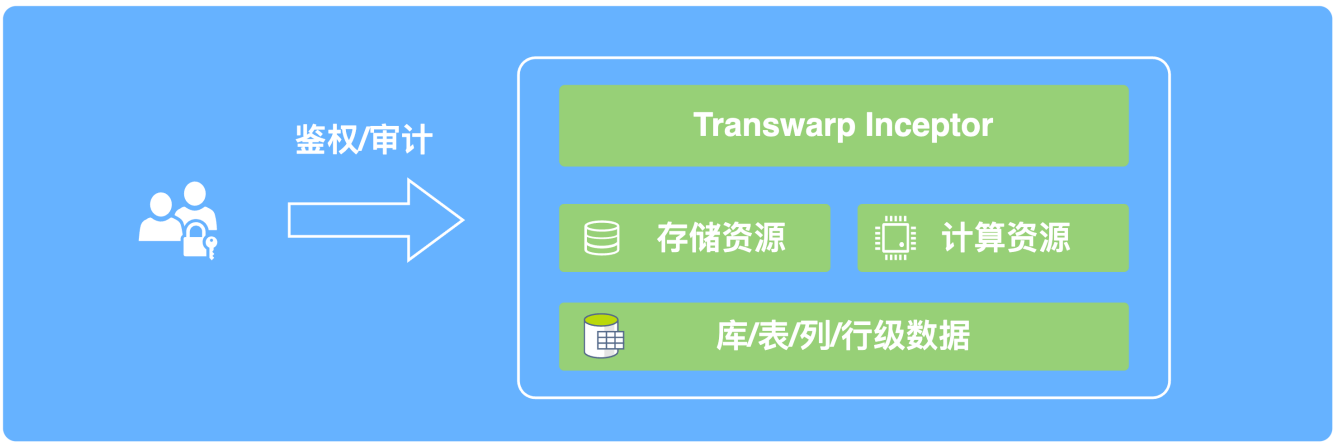


图 23. 多租户平台架构

- **鉴权**：支持 LDAP（轻型目录访问协议）和 Kerberos 认证方式，用户只需使用一套账号密码，即可完成多种服务的身份认证，降低管理成本，提升安全性和运维效率。
- **审计**：支持对鉴权登录、用户操作、数据读写等记录信息进行审计，帮助时刻掌握数据安全，回溯安全事件根因。相关介绍，见 Transwarp Aquila 手册。
- **数据权限**：提供基于角色的权限控制，支持库/表/列/行级数据的权限控制，最大限度满足业务的多样化需求。
- **存储资源**：可根据业务特点合理分配数据库的存储空间，提升存储资源利用率，避免存储容量被意外占满的风险。
- **计算资源**：支持根据业务对性能的要求设置调度策略，例如为核心业务预留部分算力保障其平稳运行。此外，Inceptor 新增了 FURION 调度策略，结合权重、任务级调度等特性，可满足复杂场景下的资源精细化分配的需求。

7.1.2. 使用流程

- 启用租户管理功能
- 规划并创建租户
- 创建角色
- 为角色分配资源/权限

7.2. 启用租户管理功能

Inceptor 的多租户管理体系基于 Transwarp Guardian 服务实现，在创建/管理租户前，您需要跟随本文步骤，安装该服务并启用插件。

7.2.1. 操作步骤

1. 登录 Transwarp Manager 平台。
2. 安装 Guardian 组件，如已安装可跳过本步骤。
   a. 在页面右上角，选择**全局服务** > **Guardian**。
   b. 根据页面提示，完成安装 Guardian 安装流程。

   > 如需了解详细的安装步骤介绍，见 Guardian 使用手册。

3. 为 Quark 服务开启 Guardian 插件。
   a. 在左侧导航栏，单击**仪表盘** > **集群**。
   b. 在 **Transwarp Inceptor** 区域内，找到并单击对应的 Qurak 服务。
   c. 单击**插件**页签，然后单击 Guardian 服务对应的的**开启**。

图 24. 开启 Guardian 插件

   d. 在弹出的对话框中，单击**确定**。
4. 选择服务的安全模式，具体操作及详细介绍，见 Guardian 使用手册。

   简单模式（默认）：无需认证即可访问相关服务。

   Kerberos 模式：访问的服务或者用户需要具备有效的 Kerberos TGT。

7.2.2. 扩展阅读

为保障各服务的认证统一和正常运行，推荐为 HDFS、ZooKeeper 等服务也开启 Guardian 插件，具体操作，见 Guardian 使用手册。

7.3. 规划并创建租户

Inceptor 通过 Guardian 组件来创建和管理租户，实现多种大数据组件（如 Hive、YARN、HDFS 等）的权限统一管理，本小节将介绍：

- 租户常见概念

- 规划租户
- 创建租户

7.3.1. 概念介绍

Inceptor 基于 ARBAC 模型（Administrative Role Based Access Control）授权机制，您可以通过为用户/组授予角色的方式实现权限的精细化管控。

图 25. 概念介绍

- **用户**：即终端用户，可使用该信息登录各服务。
- **用户组**：是一组用户的集合，支持子组，当您对用户组授予角色时，用户组中的用户会自动继承该角色相应的权限，从而简化权限管理。
- **角色**：是一个或多个权限合集，您可以授权一个角色多个操作权限，然后将角色授予给用户或用户组，该用户或用户组将继承该角色中的所有权限。
- **权限**：对指定资源执行的具体操作，例如对某个表执行 SELECT 语句。

7.3.2. 规划租户

在创建租户前，我们需要基于业务应用的需求规划租户所涉及的用户、用户组和角色，然后再授予相应权限，实现权限和资源的高效管理。

假设某企业使用 Inceptor 作为统一的数据服务平台，目前有下述两个业务：

- 核心业务：对性能要求高，需要更多资源保障业务平稳运行，拥有较高的库/表权限。
- 普通业务：对性能要求不高，权限控制上更为严格，需要隐藏某些包含敏感数据的列，保障数据安全。

基于上述需求，我们可以为这两个业务创建不同的用户组，然后再分别为其添加具体的用户，具体如下：

用户组	包含用户
core-bussiness-group	core_user_01 core_user_02 ……
normal-bussiness-group	normal_user_01 normal_user_02 ……

有了上述用户/组后，在后续的授权操作时，我们只需要创建角色并为其配置一个/多个权限，然后将角色授予给相关的用户/组即可，无需重复配置权限。

7.3.3. 创建租户

基于上述需求，我们可以为这两个业务创建不同的用户组，然后再分别为其添加具体的用户，操作流程如下。

7.3.3.1. 操作步骤

1. 登录 Guardian 平台。
   a. 登录 Transwarp Manager。
   b. 在页面右上角，选择**全局服务** > **Guardian**。
   c. 找到 Guardian Server 对应的服务链接并单击访问。

图 26. 获取登录地址

   d. 填写管理员账号和密码完成登录。
2. 在顶部菜单栏，单击**租户**。
3. 添加用户组，重复执行下述步骤分别为核心业务和普通业务创建用户组。
   a. 单击**组**页签，然后单击页面右上角的**添加用户组**。
   b. 在弹出的对话框中，填写用户组信息并单击**确定**。

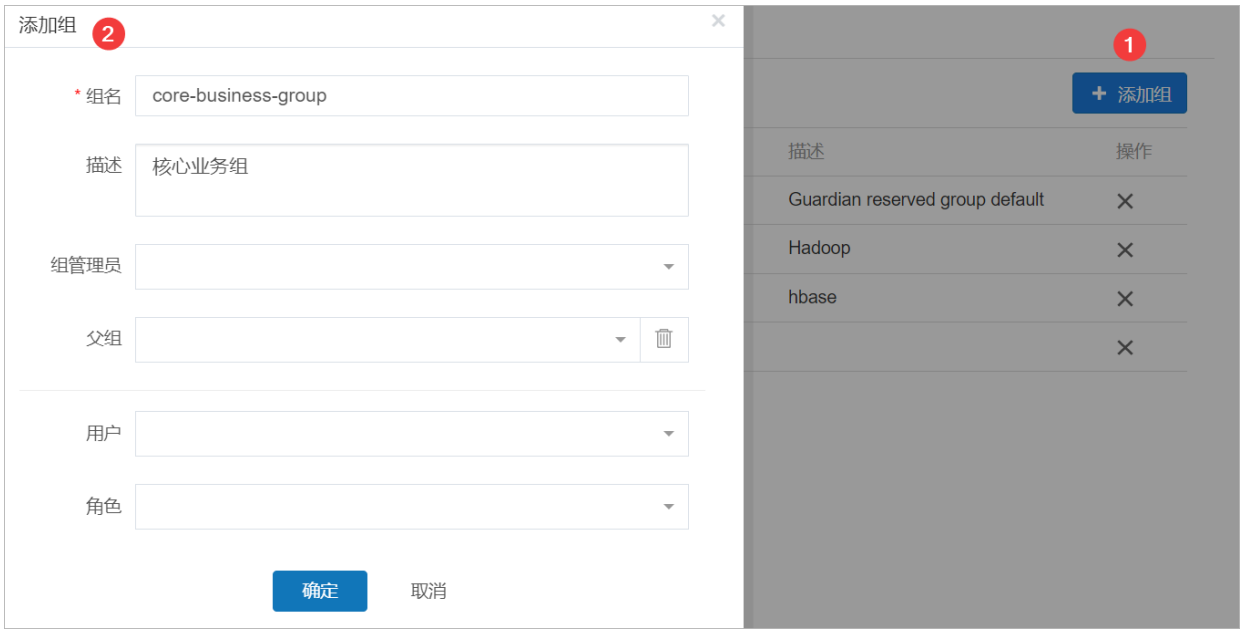


图 27. 创建用户组

组名：填写用户组名称，支持英文字母、数字、短横线（-）和下划线（_）。

邮箱：填写用户邮箱。

描述：填写具有业务意义的描述，便于后续识别。

组管理员：指定一个组管理员，也可以后续设置。

父组：如需将此用户组作为子组，可选择一个父组。

用户：选择加入至该组的用户。

角色：选择角色，本案例中保持为空，您也可以稍后创建角色并为其分配权限。

4. 添加用户，重复执行下述步骤为核心业务组和普通业务组创建所属用户。
   a. 单击**用户**页签，然后单击页面右上角的**添加用户**。
   b. 在弹出的对话框中，填写用户信息并单击**确定**。

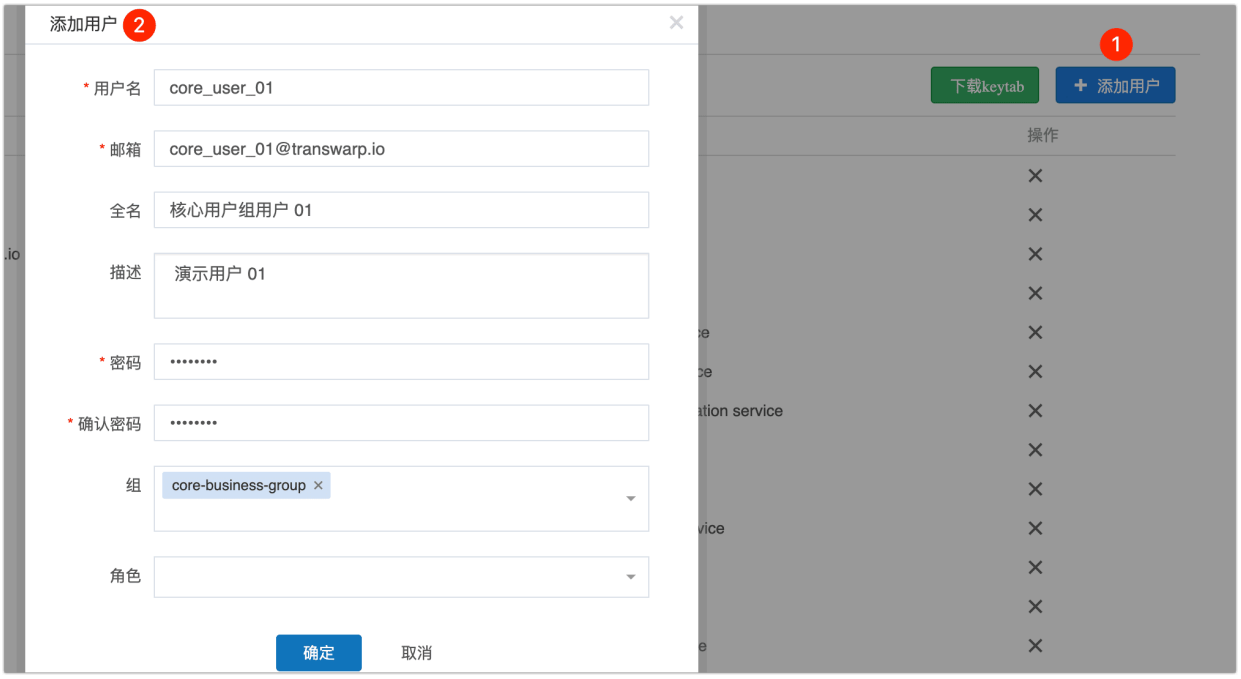


图 28. 创建用户

用户名：填写用户名称，支持英文字母、数字、短横线（-）和下划线（_）。

邮箱：填写用户邮箱。

描述：填写具有业务意义的描述，便于后续识别。

密码：设置用户的密码。

组：设置用户所属的组为刚创建的用户组。

角色：本案例中保持为空（自动获取 **public** 角色），您也可以稍后创建角色并为其分配权限。

> 您可以单击**角色**页签，查看 **public** 角色所具备的详细权限。

7.4. 通过页面管理资源/权限

完成租户中用户和组的创建后，我们可以通过角色来实现下述资源或权限的灵活管控：

类别	说明
授予库/表/列权限	支持授予指定数据库、表或列的粒度权限，保障敏感数据不被展示。
授予存储过程权限	存储过程是一组为了完成特定功能的SQL 语句集，支持授予创建、更新、执行存储过程的权限。
分配队列计算资源	Inceptor 通过队列组织和管理集群资源，支持创建队列/子队列，设置权重、预留 CPU 核数及调度策略。
授予队列权限	支持授予队列的的访问、提交作业、管理的权限。
分配存储配额	支持为数据库空间、用户、临时空间分配最大可用的存储容量。

> 本章节中，我们以为 Quark1 服务为例演示授权流程，如您部署了多个 Quark Gateway，配置流程类似。

7.4.1. 创建角色

角色是一个或多个权限的合集，我们可以基于业务需求创建角色并为其关联所需的权限，后续只需要将该角色授予给租户（用户/组）即可实现复杂权限的便捷管理。本文将介绍如何创建角色。

1. 登录 Guardian 平台。
2. 在顶部菜单栏，单击**租户**。
3. 单击**角色**页签，然后单击**添加角色**。
4. 在弹出的对话框中，填写角色名称并单击**保存**。

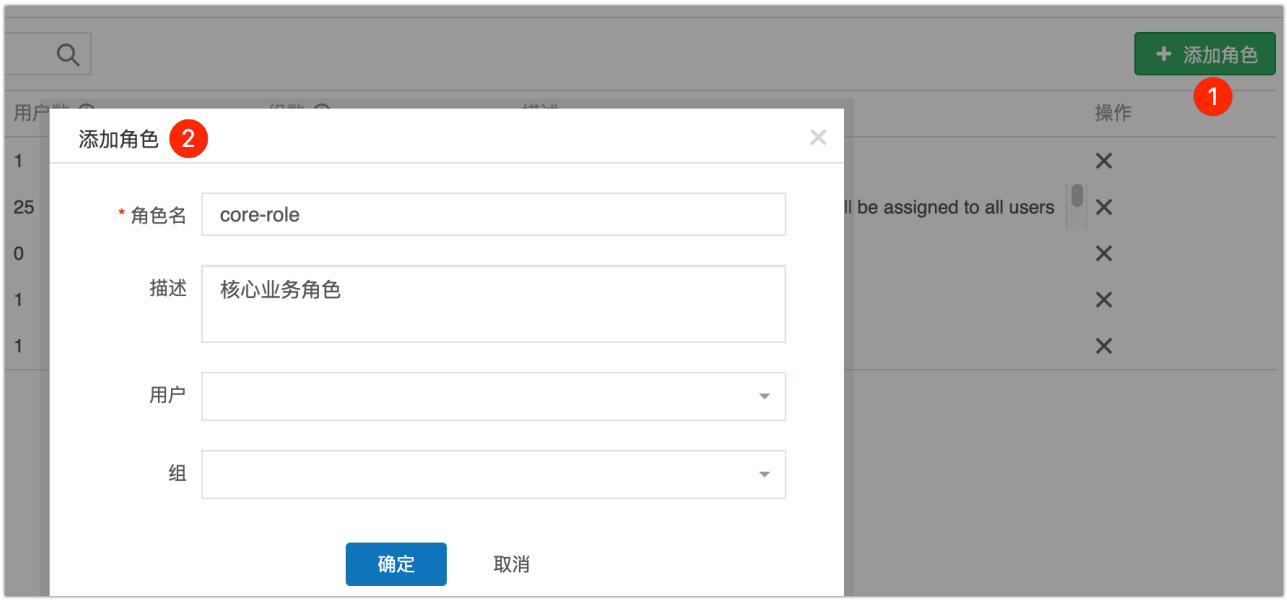


图 29. 创建角色

7.4.2. 授予库/表/列权限

Inceptor 支持库、表、列、行级别的细粒度权限控制，最大限度满足企业多样的数据管控需求，保护数据访问安全性。

7.4.2.1. 准备工作

在管理列级权限前，您需要跟随下述步骤开启该功能。

1. 登录至 Quark 节点所属设备，将下述配置添加至 **/etc/quark{替换为服务ID}/conf/guardian-site.xml** 文件中。

```
<property>
    <name>inceptor.security.column.authorization.enabled</name>
    <value>true</value>
</property>
```

2. 登录 Transwarp Manager 平台。
3. 在页面左侧，单击**仪表盘** > **集群**，找到并单击 Quark 服务。
4. 重启 Quark 服务。

> 重启该服务可能导致业务短暂不可用，请在业务低峰期或运维窗口期操作。

7.4.2.2. 操作步骤

1. 登录 Guardian 平台。
2. 在顶部菜单栏，单击**租户**。
3. 单击**角色**页签，找到并单击目标角色。
4. 在**我的权限**右侧的下拉框中，选择 **INCEPTOR** > **Quark1**，然后在其右侧选择**数据库/表/列权限**。

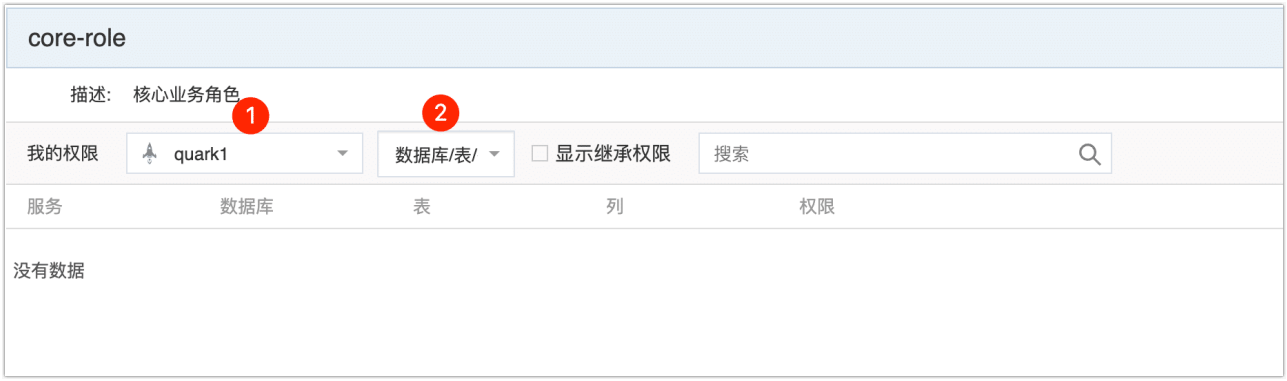


图 30. 选择授权类别

5. 单击页面右侧的**添加权限**，在弹出的对话框中选择数据库、表或列，并选择要授予的权限。

> 如需设全局权限，可选择为 **Global**。

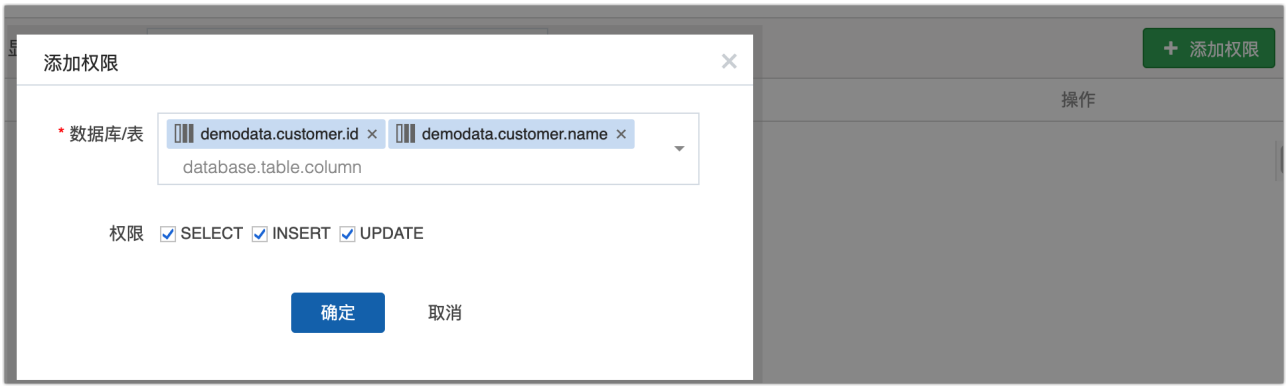


图 31. 选择库/表/列并授权

数据库/表：选择要授权的数据库、表或列，本案例选择的授权对象为 **demodata** 库中，**customer** 表的 **id** 和 **name** 列。

权限：选择要授权的权限，当授予 **ADMIN** 权限时（授权对象为列时无此选项），表示该用户可管理相关库/表的权限分配。

6. 单击**确定**。
7. 将角色授权给用户或组。
   a. 单击**添加用户**或**添加组**。
   b. 在弹出的对话框中，选中用户或组，本案例中将角色授予给之前新建的核心业务组。

图 32. 授权角色给组

c. 单击**确定**。

8. （可选）通过命令行方式登录 Inceptor，验证权限是否生效。

下述案例中，我们以具备该角色的用户登录数据库，首先查询不具备权限的表（**company**）的数据，提示无权限；然后查询具备权限的表（**customer**），查询结果只会展示已授权的列（**id** 和 **name**），而未授权的列（**phone**）不会显示。

例 6. 库/表/列权限验证示例

```
> USE demodata;
No rows affected (0.047 seconds)

> SELECT * FROM company;
Error: COMPILE FAILED: Semantic error: [Error 10002] Line 0:0 Invalid column reference.
Error encountered near token 'TOK_ALLCOLREF' (state=42000,code=10002)

> SELECT * FROM customer;
+-----+-------+
| id  | name  |
+-----+-------+
| 1   | Jack  |
+-----+-------+
1 row selected (0.532 seconds)
```

7.4.2.3. 扩展阅读

除库/表/列的权限控制外，Inceptor 还支持行级别的权限控制，例如我们希望各银行的用户只能查询到所属分行的数据，即可通过行级权限管控来实现。

具体操作，见通过 SQL 命令管理列/行权限。

7.4.3. 授予存储过程权限

存储过程是一组为了完成特定功能的SQL 语句集，为提升其安全性，您可以跟随本文教程管理存储过程权限。

7.4.3.1. 准备工作

管理存储过程权限前，您需要跟随下述步骤开启该功能。

1. 登录至 Quark 节点所属设备，将下述配置添加至 **/etc/quark{替换为服务ID}/conf/guardian-site.xml** 文件中。

```
<property>
    <name>guardian.client.inceptor.enable.plsql.check</name>
    <value>true</value>
</property>
```

2. 登录 Transwarp Manager 平台。
3. 在页面左侧，单击**仪表盘** > **集群**，找到并单击 Quark 服务。
4. 重启 Quark 服务。

> 重启该服务可能导致业务短暂不可用，请在业务低峰期或运维窗口期操作。

7.4.3.2. 操作步骤

1. 登录 Guardian 平台。
2. 在顶部菜单栏，单击**租户**。
3. 单击**角色**页签，找到并单击目标角色。
4. 在**我的权限**右侧的下拉框中，选择 **INCEPTOR** > **Quark1**，然后在其右侧选择**存储过程**。

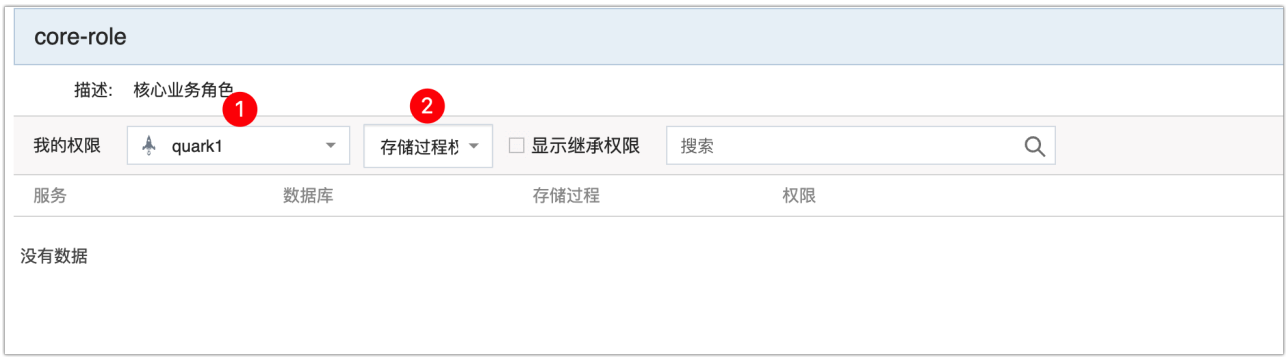


图 33. 选择授权类别

5. 单击页面右侧的**添加权限**，在弹出的对话框中选择数据库或数据库中的存储过程，本案例中选择的对象为 **demodata** 库的所有存储过程，然后选择要授予的权限。

> 如需设全局权限，可选择为 **Global**。

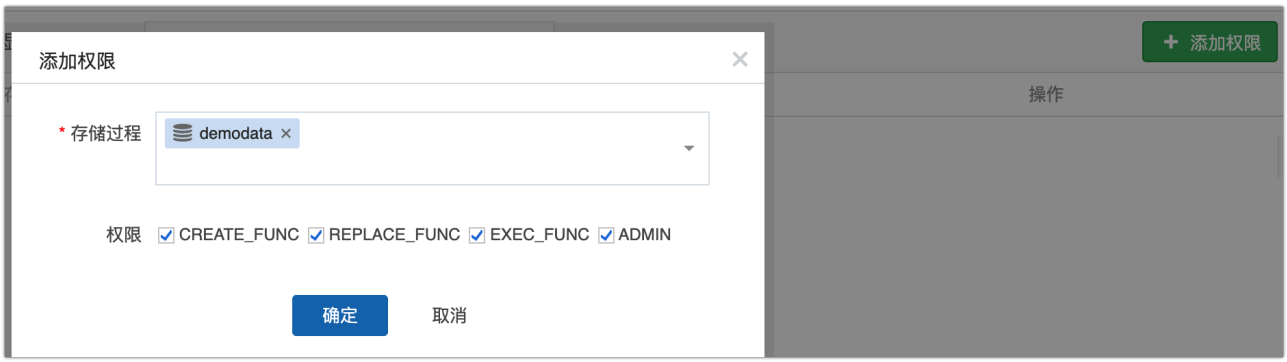


图 34. 授予存储过程权限

CREATE_FUNC：创建存储过程。

REPLACE_FUNC：可使用 `CREATE OR REPLACE` 创建存储过程。

EXEC_FUNC：执行存储过程，当选择了具体的存储过程时，仅支持授予该权限。

ADMIN：执行存储过程的权限分配。

6. 单击**确定**。
7. 将角色授权给用户或组。
   a. 单击**添加用户**或**添加组**。
   b. 在弹出的对话框中，选中用户或组，本案例中将角色授予给之前新建的核心业务组。

图 35. 授权角色给组

   c. 单击**确定**。
8. （可选）通过命令行方式登录 Inceptor，验证权限是否生效。

下述案例中，我们以具备该角色的用户来登录数据库，在已授权的 **demodata** 数据库中，首先创建存储过程 **hello_world**，提示无权限，然后调用已有的存储过程可正常执行并返回结果。

例 7. 存储过程权限验证示例

```
> !set plsqlUseSlash true
> CREATE OR REPLACE FUNCTION
> hello_world
>  (place_in IN STRING)
> RETURN STRING
> IS
> BEGIN
>   RETURN 'Hello ' || place_in
> END;
> /
Error: FAILED: Hive Internal Error: org.apache.hadoop.hive.ql.security.authorization.
plugin.HiveAccessControlException(Exception while processing [PL function creation denied
by Guardian service, user core-user-01 has no permission CREATE_FUNC on database [demodata
]]) (state=08S01,code=12)

> !set plsqlUseSlash true
> DECLARE
>    l_message string
> BEGIN
>    l_message := hello_message ('Universe')
>    DBMS_OUTPUT.put_line (l_message)
> END;
> /
+-----------------+
|     output      |
+-----------------+
| Hello Universe  |
+-----------------+
1 row selected (0.063 seconds)
```

7.4.4. 分配队列计算资源

在多个租户共用资源的场景下，可能因资源被批处理任务抢占或被多个小任务长时间占用，进而导致部分任务无法得到处理。

为精细化管控资源和提升 CPU 利用率，Inceptor 基于 FAIR（公平调度），新增了 FURION 调度策略，它采用加权的调度排序算法，同时将调度粒度缩小至任务级，此外还通过树形结构构建队列关系，可根据节点在调度指标上的关系决定调度顺序。

本文介绍如何开启 FURION 调度策略并分配计算资源。

7.4.4.1. 准备工作

默认情况，Inceptor 采用 FIFO（先进先出）调度策略，在授予计算配额前，您需要跟随下述步骤，开启 Furion 调度策略。

1. 登录 Transwarp Manager 平台。
2. 在页面左侧，单击**仪表盘** > **集群**，找到并单击 Quark 服务。
3. 单击**配置**页签，然后单击页面右侧的**添加自定义参数**。
4. 在弹出的对话框中，根据下述说明完成参数配置，然后单击**确定**。

图 36. 添加自定义参数

配置项：填写为 **spark.guardian.enable**。

值：填写为 **true**。

配置文件：选择为 **hive-site.xml**。

5. 搜索 **inceptor.scheduler.enabled** 参数，将其值修改为 **true**。

图 37. 修改预定义参数

6. 在页面右上角，选择 ··· > **配置服务**，在弹出的对话框中单击**确定**。
7. 等待配置应用完成后，单击**重启**。

> 重启该服务可能导致业务短暂不可用，请在业务低峰期或运维窗口期操作。

7.4.4.2. 操作步骤

假设某企业基于 Inceptor 建立了数据服务平台，目前主要有查询、跑批和探索分析的业务场景，队列以查询业务优先级最高，跑批业务次之，分析业务最低。我们希望借助队列调度策略能力，实现下述需求，从而实现基于业务特性分配计算资源，提升利用率和效率。

- 查询业务：保留 10% 的资源独占使用。
- 跑批业务：保留 20% 的资源独占使用。
- 分析业务：由于该业务对于性能要求的随机性较大，希望它和跑批业务一起使用未被预留的 70% 的资源。

接下来，我们介绍如何配置队列来实现上述需求。

1. 登录 Guardian 平台。
2. 在顶部菜单栏，单击**权限**，然后选择 **INCEPTOR** > **Quark1**。
3. 单击**计算配额**页签。
4. 在 **Furion 调度配置**区域框右侧，单击**编辑**。
5. 单击 **root** 队列右侧的 **+** 图标。
6. 在弹出的对话框中，根据下述说明完成配置。

   下图以配置查询业务的队列为例，其他队列的配置流程类似。

图 38. 创建队列

Queue 名称：填写队列名称，本案例填写为 **query**。

权重：权重数字越大，优先级越高。

预留百分比：要预留独享的 CPU 百分比（相对于集群的总 CPU 核数），该资源不与同级别的队列共享。

预留 CPU 核数：要预留独享的 CPU 核数，该配置与预留百分比配置一项即可。

最大 CPU 百分比：队列可使用的 CPU 资源上限百分比，超过该上限的队列中的任务将不再被调

度。

最大 CPU 核数：队列可使用的最大 CPU 核数，该配置与最大 CPU 百分比配置一项即可。

调度策略：根据业务需求选择队列内任务的调度策略。

FIFO：先进先出调度策略，根据任务提交顺序分配资源并执行，可能因某个负载较重的任务而阻塞后续任务。

FAIR：公平调度策略，第一个任务到达时会将队列所有资源给任务，当第二个任务到达时，系统会分一半的资源给该任务，以此类推。

7. 单击**确定**。
8. 参照步骤 5，为跑批、分析业务创建相应的队列，配置完成后单击**保存**，如下图所示。

图 39. 队列创建结果

9. 在页面下方，设置单个队列/连接/用户可执行的最大 SQL 数量。

图 40. 设置最大 SQL 数

10. （可选）验证配置并指定队列来执行 SQL 语句。

> 执行 SQL 的用户需确保拥有队列的 SUBMIT 权限，更多介绍，见授予队列权限。

a. 通过命令行方式登录 Inceptor，指定使用刚刚创建的 **query** 队列，然后执行一段查询 SQL 语句。

```
> set ngmr.furion.pool=query;
No rows affected (0.007 seconds)

> select * from customer limit 1;
+-----+-----------------------+-------------+
| id  |          name         |    phone    |
+-----+-----------------------+-------------+
| 1   | Jack                  | 133*******  |
+-----+-----------------------+-------------+
1 row selected (1.272 seconds)
```

b. 登录 Transwarp DBAService，通过**查询**页面，可查看到刚刚运行的查询所用的队列（调度池）为 **query**。

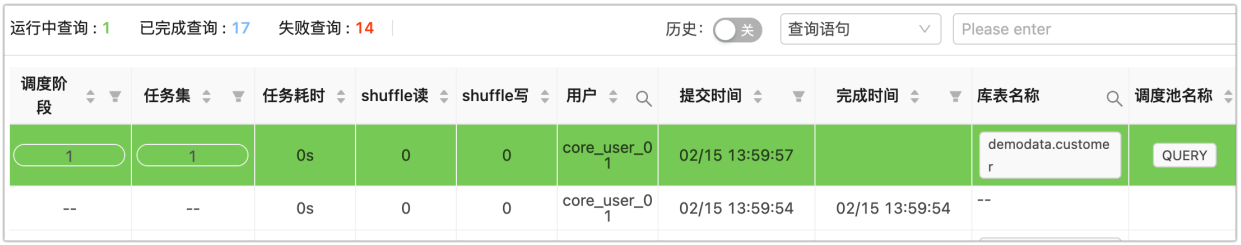


图 41. 查看查询所用队列

关于 Transwarp DBAService 的相关介绍和使用方法，见使用手册。

7.4.5. 授予队列权限

完成了队列资源的分配后，我们还需要为不同的业务用户授予对应的队列执行权限，确保 SQL 语句的执行遵循预期的资源分配规则和策略。

7.4.5.1. 操作步骤

本案例中，我们将为 core-role 角色授予 batch 队列的提交权限，具体步骤如下。

1. 登录 Guardian 平台。
2. 在顶部菜单栏，单击**租户**。
3. 单击**角色**页签，找到并单击目标角色。
4. 在**我的权限**右侧的下拉框中，选择 INCEPTOR > Quark1，然后在其右侧选择**队列权限**。

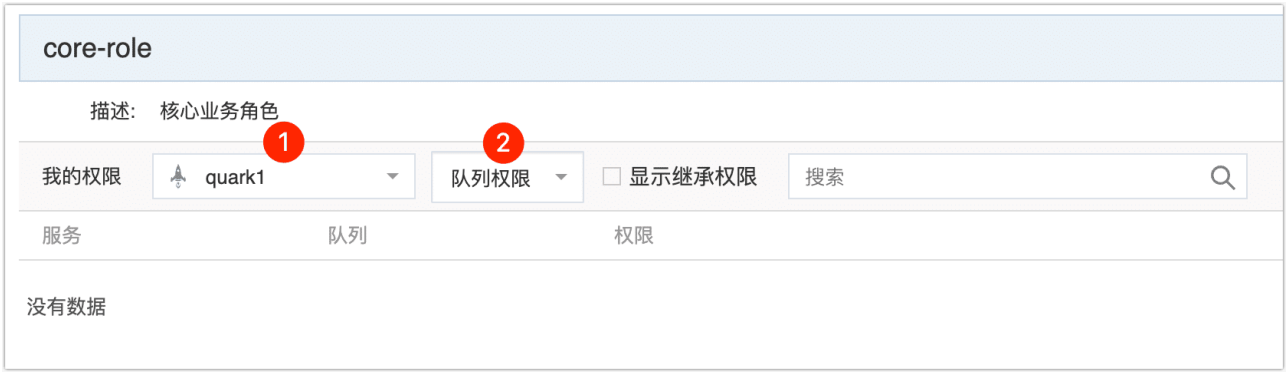


图 42. 选择队列权限

5. 单击页面右侧的**添加权限**，首先设置全局队列服务的访问权限，即选择队列为 Global，权限为 ACCESS。

图 43. 授予 ACCESS 权限

6. 单击**确定**。
7. 再次添加权限，本次选择队列为 **batch**，权限为 **SUBMIT**，允许提交 SQL 到该队列及其子队列。

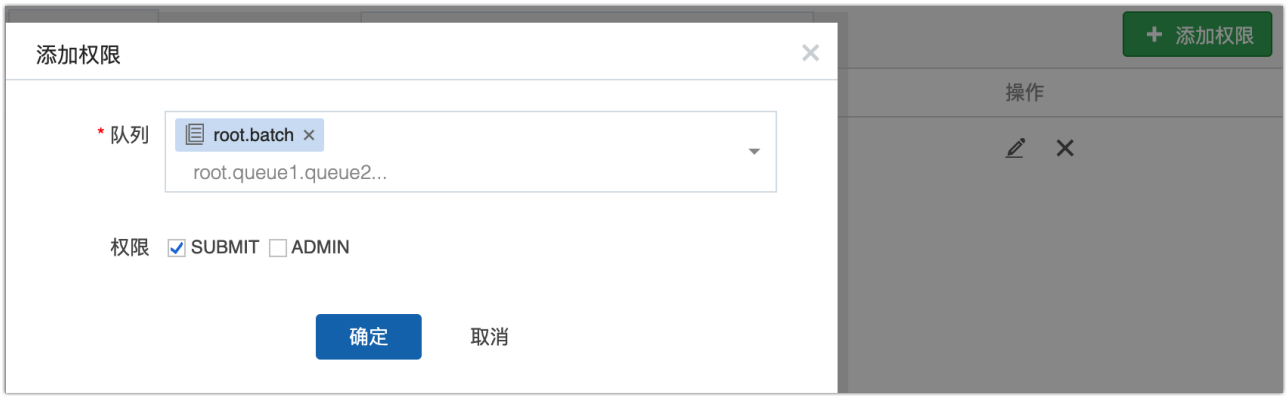


图 44. 授予 SUBMIT 权限

8. 单击**确定**。
9. 将角色授权给用户或组。
   a. 单击**添加用户**或**添加组**。
   b. 在弹出的对话框中，选中用户或组，本案例中将角色授予给之前新建的核心业务组。

图 45. 授权角色给组

   c. 单击**确定**。

> 拥有该角色的用户/组连接至 Inceptor 后，默认使用 batch 队列的资源来执行 SQL 语句。如您为该角色授予了多个队列的权限，在执行 SQL 语句前，请先执行 `set ngmr.furion.pool=队列名;` 命令指定要使用的队列。

7.4.6. 分配存储配额

合理分配数据库的存储空间有助于提升资源利用率，规避存储容量被意外占满的风险，本文介绍如何设置存储配额。

7.4.6.1. 准备工作

默认情况下，HDFS 服务读取其原生的配额设置信息，因此，在通过 Guardian 管理存储配额前，您还需要执行下述操作。

1. 登录 Transwarp Manager 平台。
2. 在页面左侧，单击**仪表盘** > **集群**，找到并单击 HDFS 服务。
3. 单击**插件**页签，开启 Guardian 插件服务。
4. 等待插件开启成功后，登录至 HDFS NameNode 节点所属设备，将下述配置添加至 **/etc/hdfs{替换为服务ID}/conf/guardian-site.xml** 文件中，

```
<property>
    <name>guardian.hdfs.quota.check.enabled</name>
    <value>true</value>
</property>
```

5. 重复执行步骤 4，完成集群下所有节点的配置修改。
6. 返回至 Transwarp Manager，重启 HDFS 服务。

> 重启该服务可能导致业务短暂不可用，请在业务低峰期或运维窗口期操作。

7.4.6.2. 操作步骤

1. 登录 Guardian 平台。
2. 在顶部菜单栏，单击**权限**，然后选择 **INCEPTOR** > **Quark1**。
3. 单击**存储配额**页签，然后根据要配置的对象单击**数据库**或**临时空间**。

> 临时空间的存储配额的生效范围为所有 Inceptor 数据库。

4. 以配置数据库配额为例，单击**添加配额**。
5. 在弹出的对话框中，选择目标数据库（可多选）、具体配额和单位。

图 46. 配置存储配合

本案例中，我们为 **demodata** 数据库配置了 10 TB 的存储配额，此外，我们还为临时空间配置了 5 GB 的存储配额。

6. 单击**确定**。
7. 单击**操作**列中的 + 图标，然后选择要授予存储配额的用户、容量和单位，单击**确定**。

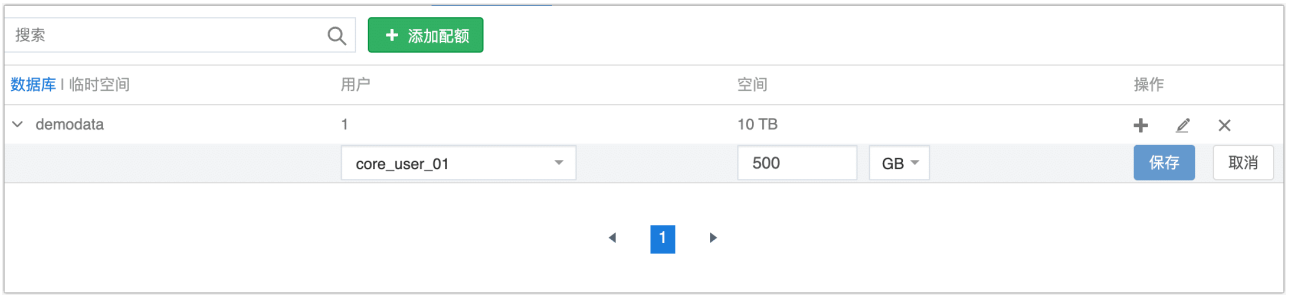


图 47. 授予用户存储配额

8. （可选）以数据库的 **Owner** 或 **ADMIN** 身份连接至 Inceptor 数据库，通过 SQL 命令验证存储配额已生效，具体如下。

```
> SHOW QUOTA ON DATABASE demodata;
+----------+
|  quota   |
+----------+
| 10.0 TB  |
+----------+
1 row selected (0.046 seconds)

> SHOW QUOTA ON TEMPORARY SPACE;
+---------+
|  quota  |
+---------+
| 5.0 GB  |
+---------+
1 row selected (0.024 seconds)

> SHOW QUOTA USER  core_user_01 ON DATABASE demodata;
+-----------+
|   quota   |
+-----------+
| 500.0 GB  |
+-----------+
1 row selected (0.037 seconds)
```

7.4.6.3. 扩展阅读

通过 SQL 命令管理存储配额

7.4.7. 授予 HDFS 权限

Inceptor 默认以 hive 用户的身份对 HDFS 目录进行操作（如创建表），而其他用户对其中的文件没有读写权限。在某些场景下（如数据导入），希望能够从 HDFS 目录中直接读取到对应表的数据，此时可以为该用户授予相关权限。本文介绍如何通过 Guardian 为用户授予 HDFS 权限。

7.4.7.1. 准备工作

在授予 HDFS 权限前，您需要跟随下述步骤，开启本功能。

1. 登录 Transwarp Manager 平台。
2. 在页面左侧，单击**仪表盘** > **集群**，找到并单击 HDFS 服务。
3. 单击**配置**页签，搜索 **inceptor.scheduler.enabled** 参数，将其值修改为 **true**。

图 48. 修改参数

4. 在页面右上角，选择 ··· > **配置服务**，在弹出的对话框中单击**确定**。
5. 等待配置应用完成后，单击**重启**。

> 重启该服务可能导致业务短暂不可用，请在业务低峰期或运维窗口期操作。

7.4.7.2. 身份判断说明

HDFS 通过下述方式判断使用者的身份和组信息：

- HDFS 未启用 Kerberos认证：通过使用者在服务器操作系统上的用户名和组判断其身份。

> 此场景下，只能通过 HDFS 命令方式进行权限/资源的配置，具体操作，见 Guardian 用户手册。

- HDFS 启用了 Kerberos认证：通过使用者的 principal 判断其身份，同时 HDFS 会将用户的 principal 映射到 NameNode 所在服务器上的用户，并通过其组信息判断用户所在的组。

7.4.7.3. 权限判断说明

HDFS在对每个操作进行权限判断时，会根据以下顺序进行：

1. 如果当前操作的执行者是 HDFS 的“超级用户”（hdfs用户）或者属于“超级组”，那么有权执行该操作

2. 如果操作的目录/文件在HDFS中的权限满足条件，那么有权执行该操作。
3. 通过 Guardian 判定是否有权限。

7.4.7.4. 操作步骤

1. 登录 Guardian 平台。
2. 在顶部菜单栏，单击**租户**。
3. 单击**角色**页签，找到并单击目标角色。
4. 在**我的权限**右侧的下拉框中，选择 **HDFS** > **hdsf1**。
5. 单击页面右侧的**添加权限**，首先设置全局 HDFS 服务的访问权限，即选择队列为 **Global**，权限为 **ACCESS**。

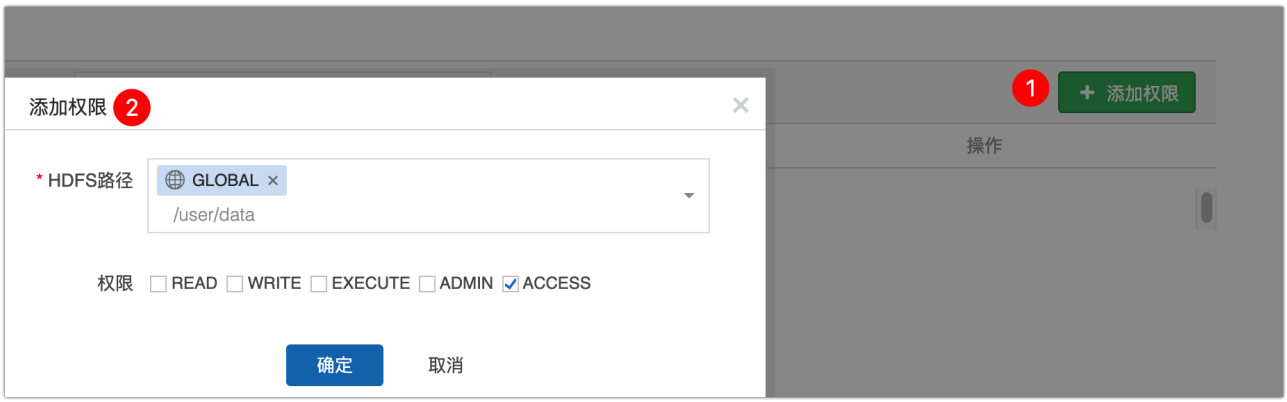


图 49. 授予 ACCESS 权限

6. 单击**确定**。
7. 再次添加权限，本次选择 **demodata** 数据库对应的文件路径，权限为 **READ**、**WRITE**、**EXECUTE**。

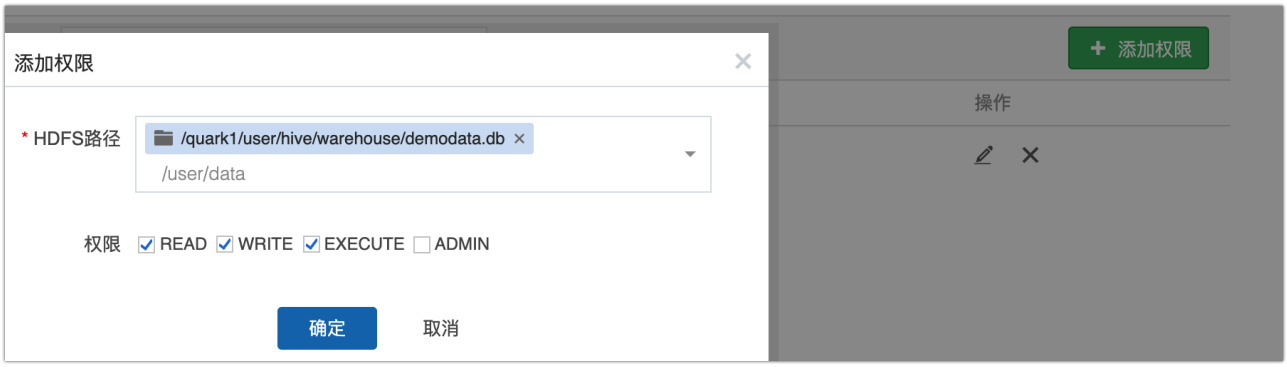


图 50. 授予读写执行权限

HDFS 路径：以层级形式展示文件路径，支持选中多个路径。

READ：读权限。

WRITE：写权限。

EXECUTE：可执行权限。

ADMIN：可以赋予和修改其他用户对于该目录的权限。

8. 单击**确定**。
9. 将角色授权给用户或组。

a. 单击**添加用户**或**添加组**。

b. 在弹出的对话框中，选中用户或组，本案例中将角色授予给之前新建的核心业务组。

图 51. 授权角色给组

c. 单击**确定**。

7.4.7.5. 扩展阅读

通过 SQL 命令管理 HDFS 权限

7.5. 通过 SQL 管理权限

7.5.1. 管理角色与权限

本文介绍如何通过 SQL 命令的方式管理数据库中的角色与权限，此外，您还可以通过页面直观地配置，提升运维效率。

7.5.1.1. 判断用户身份

如果 Inceptor 以 Kerberos认证，Inceptor 通过登陆者的 TGT 判断登录者的 principal，进而得出登录者的身份。如果是 LDAP 认证，Inceptor 通过登陆者的 LDAP 用户名来判断登录者的身份。

7.5.1.2. 获取管理员身份

本章很多操作都要求您是 Inceptor 中的管理员。您需要有Inceptor的超级用户（hive）的身份信息，只有凭 hive 的身份信息您通过 Kerberos 或者 LDAP 的认证连接上 Inceptor 才可以获取管理员角色。

- 如果您的Inceptor服务选用了Kerberos认证，您需要通过Kerberos的认证连接到Inceptor：

  您需要以您所在节点的hive principal获得一张有效的TGT，以便让Inceptor辨识身份：

  ```
  kinit -kt <hive_keytab_path> hive/<hostname>@TDH
  ```

  这里 <hostname>处提供您所登陆的节点的hostname。<hive_keytab_path>处您提供hive/<hostname>@TDH的keytab文件所在位置。比如：

```
kinit -kt /etc/sql2/hive.keytab hive/baogang2@TDH
klist
Ticket cache: FILE:/tmp/krb5cc_0
Default principal: hive/baogang2@TDH

Valid starting     Expires            Service principal
11/21/15 15:27:03  11/22/15 01:27:03  krbtgt/TDH@TDH
    renew until 11/22/15 15:27:03
```

这时，您连接Inceptor时的身份就是hive。连接Inceptor的指令是：

```
beeline -u "jdbc:hive2://<server_ip/hostname>:10000/default;principal=<hive_principal>"
```

这里 **<server_ip/hostname>** 处提供Inceptor服务所在节点的hostname或者ip；**<hive_principal>** 处提供您想要连接的Inceptor服务的principal。比如：

```
beeline -u "jdbc:hive2://baogang2:10000/default;principal=hive/baogang2@TDH"
```

- 如果您的Inceptor服务启用了LDAP认证，您需要通过LDAP的认证连接到Inceptor：

```
beeline -u "jdbc:hive2://<server_ip/hostname>:10000/default" -n hive -p <password>
```

这里<password>是hive用户在LDAP中的密码。如果LDAP中还没有hive用户，您需要先向LDAP添加hive用户并设置好密码。

- 成功连接上Inceptor后您的角色是PUBLIC，在行使admin角色独有的权限前，您应当先将自己当前的角色设为ADMIN。

```
SET ROLE ADMIN;
SHOW CURRENT ROLES;
```

role
ADMIN

Inceptor开启安全后，需要注意以下两点：

1. 采用Kerberos进行认证时，jdbc连接串中Server Principal的User Name固定为hive，即字符串形如：

```
beeline -u
"jdbc:hive2://<server_ip/hostname>:10000/default;principal=hive/<hostname>@<rea
lm>"
```

2. 开启安全后，当非hive用户操作外表时，要求hive用户对于该外表对应目录也有读写权限。

下面我们介绍Inceptor中的权限管理操作。

7.5.1.3. 管理角色

Inceptor管理角色（“role”）信息，通过向角色授权来批量管理用户权限。Table, view和database的权限既可以被授予给用户也可以被授予给角色。Inceptor中有两个特殊角色：**PUBLIC** 和 **ADMIN**。所有用户都

有 PUBLIC 角色；未经授权，只有hive用户有 ADMIN 角色。一个用户可以同时拥有多个角色，除 ADMIN 之外的角色都在用户的当前角色（CURRENT ROLES）中。但是用户也可以使用 SET ROLE 选择某个特定角色作为当前角色。

创建新角色、给用户和角色授权等权限是 ADMIN 角色独有的，拥有 ADMIN 角色的用户必须先将她当前的角色设为 ADMIN 才能行使 ADMIN 独有的权利。

> 注意，如果需要通过SQL进行实现Inceptor的创建和删除，必须提前在Guardian的界面上关闭Inceptor服务的Guardian插件。

7.5.1.3.1. CREATE ROLE：创建角色

语法

```
CREATE ROLE role_name ;
```

ALL, DEFAULT和NONE三个词是Inceptor预留的关键词，不可以做角色名。

举例： 创建角色r1

```
CREATE ROLE r1;
SHOW ROLES;
```

role
admin
public
qml
r1
r2
r3

7.5.1.3.2. DROP ROLE：删除角色

语法

```
DROP ROLE role_name;
```

举例

```
DROP ROLE r1;
SHOW ROLES;
```

role
admin
public
qml

role
r2
r3

7.5.1.3.3. SHOW CURRENT ROLES：查看用户当前所有的角色

语法

```
SHOW CURRENT ROLES;
```

默认情况下，用户所有的角色除admin外都将在current roles中显示出来。该语句无须用户当前角色为admin。

举例：Alice用户查看当前自己当前所拥有的角色

```
SHOW CURRENT ROLES;
```

role
PUBLIC

7.5.1.3.4. SET ROLE：设置角色

启用 Guardian 插件之后，SET ROLE将不再生效，需通过 Guardian 设置角色。

语法

```
SET ROLE role_name;
```

将用户当前持有的角色设置为role_name，如果用户不拥有role_name指定的角色，Inceptor会报错。该语句无须用户当前角色为admin。

例1

```
SET ROLE ADMIN;
SHOW CURRENT ROLES;
```

role
ADMIN

例2

```
SET ROLE r1;
SHOW CURRENT ROLES;
```

role
r1

语法

```
SET ROLE ALL;
```

使用SET ROLE <role_name>后，用户将仅有<role_name>指定的一个角色。使用SET ROLE ALL则可以启用用户在当前session所有 **除ADMIN角色以外** 所有其它的角色。

举例

```
SHOW CURRENT ROLES;
```

role
r1

```
SET ROLE ALL;
SHOW CURRENT ROLES;
```

role
PUBLIC
r1

7.5.1.3.5. SHOW ROLES：查看所有Inceptor中的角色

语法

```
SHOW ROLES;
```

举例

```
SHOW ROLES;
```

role
admin
public
qml
r1
r2
r3

7.5.1.3.6. GRANT ROLE：授予角色

语法

```
GRANT role_name [, role_name] ...
TO principal_specification [, principal_specification] ...
;

principal_specification
  : USER user
  | ROLE role
```

将角色授予给用户或角色。

例1

```
GRANT r1 TO USER user1;
```

7.5.1.3.7. REVOKE ROLE：收回角色

语法

```
REVOKE role_name [, role_name] ...
FROM principal_specification [, principal_specification] ... ;

principal_specification
  : USER user
  | ROLE role
```

将角色从用户或者别的角色处收回。

从user1收回r1角色。

```
REVOKE r1 FROM USER user1;
```

user1不再有r1角色。

```
SHOW ROLE GRANT USER user1;
```

role	grant_option	grant_time	grantor
PUBLIC	false	0	

7.5.1.3.8. SHOW ROLE GRANT：查看用户/角色拥有的角色

语法： 查看user_name指定的用户拥有的角色

```
SHOW ROLE GRANT USER user_name;
```

举例

```
GRANT r1, r2, r3 TO USER user1;
SHOW ROLE GRANT USER user1;
```

role	grant_option	grant_time	grantor
PUBLIC	false	0	
r1	false	1511925171000	hive
r2	false	1511925172000	hive
r3	false	1511925172000	hive

语法： 查看role_name指定的角色拥有的角色

```
SHOW ROLE GRANT ROLE role_name;
```

举例

```
GRANT r2, r3 TO ROLE r1;
SHOW ROLE GRANT ROLE r1;
```

role	grant_option	grant_time	grantor
r2	false	1511925172000	hive
r3	false	1511925172000	hive

7.5.1.3.9. SHOW PRINCIPALS：查看拥有某个角色的用户和角色

语法

```
SHOW PRINCIPALS role_name;
```

举例

```
SHOW PRINCIPALS r1;
```

principal_name	principal_type	grant_option	grantor	grantor_type	grant_time
user1	USER	false	N/A		0

7.5.1.4. 管理权限

Inceptor中的权限有以下几种：

- **CREATE**：创建DATABASE、TABLE或者VIEW的权限；
- **SELECT**：查看DATABASE中的TABLE或VIW，或者对TABLE或者VIEW执行 **SELECT** 的权限。

- **INSERT**：对TABLE或VIEW执行 **INSERT** 的权限。
- **UPDATE**：对TABLE或VIEW执行 **UPDATE** 的权限。
- **DELETE**：对TABLE或VIEW执行 **DELETE** 的权限。
- **ALL**：对TABLE或VIEW执行 **SELECT+INSERT+UPDATE+DELETE** 的权限。

权限的级别可以是全局、DATABASE、TABLE和VIEW。

- 全局权限包括：

  创建DATABASE的权限（全局的 **CREATE** 权限）；

  对全局所有表的 **SELECT**，**INSERT**，**UPDATE**，**DELETE** 或 **ALL** 权限（SELECT|INSERT|UPDATE|DELETE|ALL ON *.*）。
- DATABASE级别权限包括：

  在某个DATABASE中建表的权限（ CREATE ON DATABASE <database>.* ）；

  查看某个DATABASE中所有表的权限 （ **SELECT ON DATABASE** ）；

  对某个DATABASE中所有表的 **SELECT**，**INSERT**，**UPDATE**，**DELETE** 或 **ALL** 权限（SELECT|INSERT|UPDATE|DELETE|ALL ON <database>.*）。
- TABLE和VIEW级别的权限包括：

  对某个TABLE或VIEW的 **SELECT**，**INSERT**，**UPDATE**，**DELETE** 或 **ALL** 权限（SELECT|INSERT|UPDATE|DELETE|ALL ON <table_or_view_name>）。

未经任何授权，一个普通用户（不拥有 **ADMIN** 角色的用户）在Inceptor中的权限只有：

- 查看所有DATABASE的权限；
- 查看default数据库下所有表的权限；
- 在default数据库中建表和视图的权限；
- 自己是owner的（自己创建的）DATABASE的 **CREATE** 和 **SELECT** 权限，并可将这些权限赋予其他用户或角色。
- 自己是owner的（自己创建的）TABLE和VIEW的 **SELECT**，**INSERT**，**UPDATE** 和 **DELETE** 权限，并且可以将这些权限授予其他用户或角色。

> Owner将自己的表的权限授予他人的能力会引起权限的扩散，请参考Inceptor权限传播控制查看如何避免这个问题。

其他权限，包括创建数据库，在其他数据库中建表，对非自己的表/视图进行操作等都需要Inceptor管理员的授权。

权限管理的语法可以概括如下：

```
--授予权限
GRANT
    <priv_type> [, <priv_type> ] ...
    [ON <object_specification>]
    TO <principal_specification> [, <principal_specification>] ...
    ;

--收回权限
REVOKE
    <priv_type> [, <priv_type> ] ...
    [ON <object_specification>]
    FROM <principal_specification> [, <principal_specification>] ... ;

--查看权限
SHOW GRANT
    [<principal_specification>]
    [ON (ALL| (<object_specification>)];

priv_type
  : CREATE | INSERT | SELECT | UPDATE | DELETE | ALL

object_specification
  : DATABASE <database_name>
  | [TABLE] <table_or_view_name>
  | [TABLE] *.*
  | [TABLE] <database_name>.*
  | [TABLE] *

principal_specification
  : USER <user>
  | ROLE <role>
```

下面详细介绍权限管理操作。

7.5.1.4.1. GRANT：权限的授予

授予全局权限

语法：授予全局 CREATE 权限

```
GRANT
  CREATE
  TO <principal_specification> [,<principal_specification> ...]
  ;
```

说明： 该语句授予用户/角色创建DATABASE的权限

例 8. 授予用户alice和角色r1全局 CREATE 权限

```
GRANT CREATE TO USER alice, ROLE r1;
```

语法：授予对全局所有表的权限

```
GRANT
  SELECT | INSERT | UPDATE | DELETE | ALL
  ON [TABLE] *.*  ①
  TO <principal_specification>[, <principal_specification>, ...]
  ;
```

① 注意，`.`表示全局所有表。

例 9. 授予用户alice和角色r1全局所有表的 SELECT 权限

```
GRANT SELECT ON *.* TO USER alice, ROLE r1;
```

授予数据库级别权限

语法：授予数据库的 CREATE 和 SELECT 权限

```
GRANT
  CREATE | SELECT
  ON DATABASE <database_name>
  TO <principal_specification> [,<principal_specification> ...]
  ;
```

说明：对数据库的 **CREATE** 权限是在数据库中建表的权限。对数据库的 **SELECT** 权限就是该数据库中的表进行SELECT的权限。

例 10. 授予用户alice在对db数据库的 SELECT 和 CREATE 权限

```
GRANT SELECT, CREATE ON DATABASE db TO USER alice;
```

语法：授予数据库中所有表的权限

```
GRANT
  SELECT | INSERT | UPDATE | DELETE | ALL
  ON [TABLE] <database_name>.* ①
  TO <principal_specification> [, <principal_specification> ...]
  ;

GRANT
  SELECT | INSERT | UPDATE | DELETE | ALL
  ON [TABLE] * ②
  TO <principal_specification> [, <principal_specification> ...]
  ;
```

① 授予 **<database_name>** 指定的数据库中所有表的权限。

② 授予当前数据库中所有表的权限。

例 11. 授予用户alice对test数据库中所有表的 SELECT 权限

```
GRANT SELECT ON test.* TO USER alice;
```

也可以先切换到test数据库再授权：

```
USE test;
GRANT SELECT ON * TO USER alice;
```

授予表和视图级别的权限

语法：授予表和视图级别的权限

```
GRANT
  SELECT | INSERT | UPDATE | DELETE | ALL
  ON [TABLE] <table_or_view_name>
  TO <principal_specification> [, <principal_specification> ...]
  ;
```

例 12. 授予用户alice对表ta进行 MERGE 的权限

Inceptor 中没有 MERGE 这个权限，对表执行 MERGE 需要表的 SELECT 和 UPDATE 权限：

```
GRANT SELECT, UPDATE ON ta TO USER alice;
```

7.5.1.4.2. REVOKE：权限的收回

收回全局权限

语法：收回全局 CREATE 权限

```
REVOKE [GRANT OPTION FOR]
  CREATE
  FROM <principal_specification> [, <principal_specification>, ...];
```

说明：该语句收回用户/角色创建DATABASE的权限

例 13. 收回用户alice和角色r1全局 CREATE 权限

```
REVOKE CREATE FROM USER alice, ROLE r1;
```

语法：收回对全局所有表的权限

```
REVOKE [GRANT OPTION FOR]
  SELECT | INSERT | UPDATE | DELETE | ALL
  ON [TABLE] *.*  ①
  FROM <principal_specification>[, <principal_specification>, ...]
  ;
```

① 注意，`.`表示全局所有表。

例 14. 收回用户alice和角色r1对全局所有表的 SELECT 权限

```
REVOKE SELECT ON *.* FROM USER alice, ROLE r1;
```

收回数据库级别的权限

语法：收回指定数据库的 CREATE 和 SELECT 权限

```
REVOKE [GRANT OPTION FOR]
  CREATE | SELECT
  ON DATABASE <database_name>
  FROM <principal_specification> [,<principal_specification> ...]
  ;
```

例 15. 收回用户alice对db数据库的 SELECT 和 CREATE 权限

```
REVOKE SELECT, CREATE ON DATABASE db FROM USER alice;
```

语法：收回数据库中所有表的权限

```
REVOKE [GRANT OPTION FOR]
  SELECT | INSERT | UPDATE | DELETE | ALL
  ON [TABLE] <database_name>.* ①
  FROM <principal_specification> [, <principal_specification> ...]
  ;

REVOKE [GRANT OPTION FOR]
  SELECT | INSERT | UPDATE | DELETE | ALL
  ON [TABLE] * ②
  FROM <principal_specification> [, <principal_specification> ...]
  ;
```

① 收回 **<database_name>** 指定的数据库中所有表的权限。

② 收回当前数据库中所有表的权限。

例 16. 收回用户alice对test数据库中所有表的 SELECT 权限

```
REVOKE SELECT ON test.* FROM USER alice;
```

也可以先切换到test数据库再收回权限：

```
USE test;
REVOKE SELECT ON * FROM USER alice;
```

收回表和视图级别的权限

语法：收回表和视图级别的权限

```
REVOKE [GRANT OPTION FOR]
  SELECT | INSERT | UPDATE | DELETE | ALL
  ON [TABLE] <table_or_view_name>
  FROM <principal_specification> [, <principal_specification> ...]
  ;
```

例 17. 收回用户alice对表ta的 SELECT 授权权限

```
REVOKE GRANT OPTION FOR SELECT ON TABLE ta FROM USER alice;
```

例 18. 收回用户alice对表ta进行 MERGE 的权限

Inceptor 中没有 **MERGE** 这个权限，对表执行 **MERGE** 需要表的 **SELECT** 和 **UPDATE** 权限，收回这两个权限用户alice即无法对表ta进行 **MERGE** 操作。

```
REVOKE SELECT, UPDATE ON ta FROM USER alice;
```

7.5.1.4.3. SHOW：权限的查看

语法：查看权限

```
SHOW GRANT
    [<principal_name>] ①
    [ON (ALL| (<object_specification>)]; ②

object_specification
  : DATABASE <database_name>
  | [TABLE] <table_or_view_name>
  | [TABLE] *.*
  | [TABLE] <database_name>.*
  | [TABLE] *
```

① 指定查看某个用户或角色拥有的权限；

② 指定查看某个或多个对象上的权限。

例 19. 授予用户bob数据库db中所有表的 SELECT 后查看权限

```
GRANT SELECT ON db.* TO USER bob;
```

先查看用户bob拥有的权限：

```
SHOW GRANT USER bob;
```

查看结果为：

database	table	partition	column	principal_name	principal_type	privilege	grant_option	grant_time	grantor

然后我们查看db中所有表的权限：

```
SHOW GRANT ON db.*;
```

我们能看到除bob外，用户alice对db中所有的表也都有 **SELECT** 权限

database	table	partition	column	principal_name	principal_type	privilege	grant_option	grant_time	grantor
db				alice	USER	SELECT	false	0	N/A
db				bob	USER	SELECT	false	0	N/A
db				hive	USER	ADMIN	false	0	N/A
db				hive	USER	CREATE	false	0	N/A
db				hive	USER	SELECT	false	0	N/A

例 20. 查看全局所有表的权限

```
SHOW GRANT ON *.*;
```

输出类似如下：

database	table	partition	column	principal_name	principal_type	privilege	grant_option	grant_time	grantor
				admin	USER	ADMIN	false	0	N/A
				hive	USER	ADMIN	false	0	N/A
				hive	USER	CREATE	false	0	N/A
				hive	USER	SELECT	false	0	N/A
				public	ROLE	ACCESS	false	0	N/A

结果只会返回对全局对象有权限的用户或角色。

7.5.1.5. 操作所需的权限总结

下面我们总结在Inceptor中进行各项操作的 **必要** 权限。 表格中，

- “Y”表示“需要有此权限”
- “YD”表示“需要对操作所在数据库有此权限”
- “YG”表示“需要此权限，且需要将权限授予别的用户或角色的权限（GRANT OPTION）”。

操作	CREATE	SELECT	INSERT	UPDATE	DELETE	Ownership	ADMIN
CREATE TABLE	YD						
DROP TABLE						Y	
DESCRIBE TABLE		Y					
SHOW PARTITIONS		Y					
ALTER TABLE LOCATION						Y	
ALTER PARTITION LOCATION						Y	
ALTER TABLE ADD PARTITION			Y				
ALTER TABLE DROP PARTITION					Y		
其他 ALTER TABLE 语句						Y	
TRUNCATE TABLE						Y	
CREATE VIEW	YD	YG(源)					

操作	CREATE	SELECT	INSERT	UPDATE	DELETE	Ownership	ADMIN
ALTER VIEW PROPERTIES						Y	
ALTER VIEW RENAME						Y	
DROP VIEW PROPERTIES						Y	
DROP VIEW						Y	
SHOW COLUMNS		Y					
CREATE TABLE AS SELECT	YD	Y（源）					
SHOW TABLES		YD					
SELECT		Y					
INSERT OVERWRITE			Y		Y		
INSERT INTO			Y				
UPDATE				Y			
DELETE					Y		
LOAD			Y		Y		
SHOW CREATE TABLE		YG					
EXPLAIN		Y					
CREATE DATABASE	Y (global)						
DROP DATABASE						Y	
CREATE DATABASE	Y (global)						
CREATE FUNCTION							Y
DROP FUNCTION							Y
ALTER DATABASE							Y

默认分配select权限后，用户可以执行select和show create操作。但是在某些场景下需要对show create权限进行控制，即允许用户执行select，而不允许show create权限。

7.5.2. 管理行/列级权限

针对数据敏感的场景，例如希望各银行的用户只能查询到各自分行的数据，不希望展示某些列（如电话号）的数据，此时可为指定的表授予行/列级权限，最大限度满足企业多样的数据管控需求，保护数据访问安全性。

本文介绍如何通过 SQL 命令的方式管理行/列级权限，此外，您还可以通过页面直观地配置库、表、列级权限。

7.5.2.1. 行/列级与表级权限说明

类别	说明
表权限（PRIVILEGE）	限制执行 **SELECT**、**INSERT**、**UPDATE**、**DELETE** 操作。
行权限（PERMISSION）	限制表级的 **SELECT**、**UPDATE**、**DELETE** 操作对指定行生效。
列权限（PERMISSION）	限制表级的 **SELECT** 操作对指定列生效。

您需要为用户授予表级权限，行/列级权限设置才有意义，例如某用户没有表 A 的 SELECT 权限，那么为其授予行级 SELECT 权限也依旧无法读取表 A 的数据。

> INSERT 操作不受行/列级权限限制，此外，由于行级权限没有“行 Owner” 的概念，不会自动继承行级权限，例如用户插入了一条数据，在对其执行 **SELECT**、**UPDATE**、**DELETE** 操作时依然受控于行权限约束。

7.5.2.2. SQL 语法及使用示例

- 设置行级权限
- 撤销行级权限设置
- 设置列级权限
- 撤销列级权限设置
- 查看行/列级权限设置

> 执行本文介绍的 SQL 命令前，您需要以 **ADMIN** 角色或表 Owner 的身份连接数据库。

7.5.2.3. 设置行级权限

语法

```
GRANT PERMISSION ON TABLE <table_name> FOR ROWS <where_clause>;
```

说明：〈where_clause〉为一个where字句。

7.5.2.3.1. 示例一：授予行权限给用户

在本示例中，我们在表中新建一个用户列（**username**），将每行记录的 username 值设为对该行记录有权限的用户，然后结合 context 函数，即可实现只有当前用户（**current_user**）和 **username** 的值相匹配时才能对记录进行操作。

本示例中，我们使用的表为 **finances_holo**，其完整数据如下：

```
+------+---------+-----------+
| eid  |  funds  | username  |
+------+---------+-----------+
| 001  | 500.0   | alice     |
| 002  | 500.0   | bob       |
| 003  | 1000.0  | alice     |
| 004  | 2000.0  | carol     |
+------+---------+-----------+
```

1. 以 ADMIN 角色或表 Owner 的身份连接数据库，执行下述操作即可在实现上述需求，同时我们授予 ADMIN 角色对所有记录拥有操作权限。

```
GRANT PERMISSION ON TABLE finances_holo FOR ROWS WHERE username = CURRENT_USER() OR HAS_ROLE(
'admin');
```

2. 查看行级权限已开启，如下所示。

```
> SHOW PERMISSION ON TABLE finances_holo;
+-------------+-------------------------------------------------------+
| row/column  |                       definition                      |
+-------------+-------------------------------------------------------+
| ROWS        | WHERE username = CURRENT_USER() OR HAS_ROLE('admin')  |
+-------------+-------------------------------------------------------+
```

3. （可选）验证用户权限。

上述授权操作并不会为用户赋予表级权限，例如某用户本身不具备对 **finances_holo** 表的 SELECT 权限，那么他也无法读取到数据。接下来，我们以 alice 用户为例，演示其在不同表权限下的数据读写情况。

场景一：alice 拥有表的 **SELECT/INSERT/UPDATE/DELETE** 权限时。

执行 SELECT 操作查看所有记录时，只会展示 **username** 值为 **alice** 的记录。

```
> SELECT * FROM finances_holo;
+------+---------+-----------+
| eid  |  funds  | username  |
+------+---------+-----------+
| 003  | 1000.0  | alice     |
| 001  | 500.0   | alice     |
+------+---------+-----------+
```

执行 **UPDATE/DELETE** 操作，只会对上述两条记录生效，以 UPDATE 操作为例，如下所示。

INSERT 权限不受行级权限影响。

```
> UPDATE finances_holo SET funds = 800 WHERE eid = "001";
1 row affected (0.711 seconds)

> SELECT * FROM finances_holo;
+------+---------+-----------+
| eid  |  funds  | username  |
+------+---------+-----------+
| 003  | 1000.0  | alice     |
| 001  | 800.0   | alice     |
+------+---------+-----------+
2 rows selected (1.68 seconds)
```

场景二：alice 不具备 finances_holo 表的任何权限。

以执行 SELECT 操作查看数据为例，Inceptor 将返回无权限提示信息。

```
> SELECT * FROM finances_holo;
Error: COMPILE FAILED: Internal error HiveAccessControlException: [Error 20388] Permission
denied: Principal [name=alice, type=USER] does not have following privileges for operation
QUERY [[SELECT] on Object [type=TABLE_OR_VIEW, name=demodata.finances_holo]] (state=42000
,code=20388)
```

7.5.2.3.2. 示例二：使用辅助表设置行级权限

本示例中，我们希望安装用户所在的分支银行对 finances_new 表授予行级权限，即用户只能看到其所属支行的数据。

首先，我们为表新增一个支行列（**branch**），将每行记录的 username 值设置为对该行记录有权限的支行名称，然后通过一个辅助表来记录用户与支行的对应关系，在行级权限设置时，通过该表过滤出有权限的用户。

本示例中，我们使用了两个表，分别是：

- **finances_new**：完整数据如下。

```
+------+---------+---------+
| eid  |  funds  | branch  |
+------+---------+---------+
| 001  | 500.0   | A       |
| 002  | 500.0   | A       |
| 003  | 1000.0  | A       |
| 004  | 2000.0  | B       |
+------+---------+---------+
```

- **branch_assignment**（辅助表），记录了用户与支行的关系，完整数据如下。

```
> SELECT * FROM branch_assignment;
+-----------+---------+
| username  | branch  |
+-----------+---------+
| bob       | A       |
| alice     | A       |
| carol     | B       |
+-----------+---------+
```

1. 以 ADMIN 角色或表 Owner 的身份连接数据库，执行下述操作即可在实现上述需求，同时我们授予 ADMIN 角色对所有记录拥有操作权限。

```
GRANT PERMISSION ON TABLE finances_new FOR ROWS WHERE branch = (SELECT branch FROM
branch_assignment WHERE username = CURRENT_USER()) OR HAS_ROLE('ADMIN');
```

> 为保障权限判断运行正常，我们为用户授予了 branch_assignment 表的 SELECT 权限。

2. 执行 `SHOW PERMISSION ON finances_new;` 命令可查看行级权限已开启。
3. （可选）验证用户权限。

   假设我们已经为不同支行的用户授予了 **finances_new** 表的 **SELECT** 权限，接下来，我们以 alice 和 carol 用户为例，演示其数据读取情况。

   以 alice 身份连接 Inceptor 数据库，只能看到 A 支行的数据。

```
> SELECT * FROM finances_new;
+------+---------+---------+
| eid  |  funds  | branch  |
+------+---------+---------+
| 001  | 500.0   | A       |
| 002  | 500.0   | A       |
| 003  | 1000.0  | A       |
+------+---------+---------+
```

   以 carol 身份连接 Inceptor 数据库，只能看到 B 支行的数据。

```
> SELECT * FROM finances_new;
+------+---------+---------+
| eid  |  funds  | branch  |
+------+---------+---------+
| 004  | 2000.0  | B       |
+------+---------+---------+
```

7.5.2.4. 撤销行级权限设置

语法

```
REVOKE PERMISSION ON TABLE <table_name> FOR ROWS;
```

<table_name>更换为目标表名，执行该语句将撤销对指定表的行级权限设置。

示例

1. 以 ADMIN 角色或表 Owner 的身份连接数据库，查看 **finances_holo** 表的行级权限设置。

```
SHOW PERMISSION ON TABLE finances_holo;
+-------------+-------------------------------------------------------+
| row/column  |                      definition                       |
+-------------+-------------------------------------------------------+
| ROWS        | WHERE username = CURRENT_USER() OR HAS_ROLE('admin')  |
+-------------+-------------------------------------------------------+
```

2. 撤销行级权限设置，随后查看权限设置已清空。

```
> REVOKE PERMISSION ON TABLE finances_holo FOR ROWS;
No rows affected (0.115 seconds)

> SHOW PERMISSION ON TABLE finances_holo;
+-------------+-------------+
| row/column  | definition  |
+-------------+-------------+
+-------------+-------------+
```

7.5.2.5. 设置列级权限

语法

```
GRANT PERMISSION ON [TABLE] <table_name> FOR COLUMN <case_when>;
```

这里的 **<case_when_clause>** 为一个 **CASE WHEN** 条件函数。

7.5.2.5.1. 使用示例

假设我们有一个银行用户信息表（**bank_account**），其完整数据如下。

```
+--------+---------------------+------------------+
|  name  |         id          |      email       |
+--------+---------------------+------------------+
| jack   | 511702198907108849  | jack@126.com     |
| alice  | 530724198511061294  | alice@gmail.com  |
| tom    | 320702199203128764  | tom@outlook.com  |
| amy    | 440183198811183889  | amy@qq.com       |
+--------+---------------------+------------------+
```

为保障信息安全，我们希望只有 ADMIN 用和表 Owner 可查看完整数据，对普通的业务用户只展示脱敏后的

数据，具体如下：

- **name** 列：只显示第一个字符，其他隐藏为星号。
- **id** 列：只显示前三位，后四位，其他隐藏。
- **email** 列：邮箱的用户仅显示第一个字母，其余隐藏为星号，同时可正常展示邮箱域名。

1. 以 ADMIN 角色或表 Owner 的身份连接数据库，执行下述命令为 normal_user 和 hive 用户授予 **bank_account** 表的 **SELECT** 权限。

```
GRANT SELECT ON bank_account TO USER hive, USER normal_user;
```

2. 依次执行下述命令，完成列级权限设置。

```
-- 当用户角色不为 ADMIN 时，只显示 name 列的第一个字符，其他隐藏为星号；
GRANT PERMISSION ON bank_account
FOR COLUMN name CASE WHEN HAS_ROLE('ADMIN')
THEN name ELSE MASK(name, 1) END;


-- 当用户角色不为 ADMIN 时，只显示 id 列的前三位，后四位，其他隐藏为星号；
GRANT PERMISSION ON bank_account
FOR COLUMN id CASE WHEN HAS_ROLE('ADMIN')
THEN id ELSE MASK(id, 3, 4) END;


-- 当用户角色不为 ADMIN 时，只显示 email 列邮箱前缀的第一个字母，前缀的其他字母隐藏为星号，邮箱域名显示。
GRANT PERMISSION ON bank_account
FOR COLUMN email CASE WHEN HAS_ROLE('ADMIN')
THEN email ELSE MASK_EMAIL(email) END;
```

关于脱敏用法的详细介绍，见脱敏函数。

3. 查看授权设置，示例如下。

```
> SHOW PERMISSION ON bank_account;
+---------------+-------------------------------------------------------------------+
|  row/column   |                              definition                           |
+---------------+-------------------------------------------------------------------+
| COLUMN name   | CASE WHEN HAS_ROLE('ADMIN') THEN name ELSE MASK(name, 1) END      |
| COLUMN id     | CASE WHEN HAS_ROLE('ADMIN') THEN id ELSE MASK(id, 3, 4) END       |
| COLUMN email  | CASE WHEN HAS_ROLE('ADMIN') THEN email ELSE MASK_EMAIL(email) END |
+---------------+-------------------------------------------------------------------+
```

4. （可选）验证用户权限。

- 以管理员 hive 身份连接数据库，可查看到完整的数据。

```
> SET ROLE ADMIN;
No rows affected (0.031 seconds)
> SELECT * FROM bank_account;
+--------+---------------------+------------------+
|  name  |         id          |      email       |
+--------+---------------------+------------------+
| jack   | 511702198907108849  | jack@126.com     |
| alice  | 530724198511061294  | alice@gmail.com  |
| tom    | 320702199203128764  | tom@outlook.com  |
| amy    | 440183198811183889  | amy@qq.com       |
+--------+---------------------+------------------+
```

- 以普通用户 normal_user 身份连接数据库，只能查看到脱敏后的数据。

```
> SELECT * FROM bank_account;
+--------+---------------------+------------------+
|  name  |         id          |      email       |
+--------+---------------------+------------------+
| j***   | 511***********8849  | j***@126.com     |
| a****  | 530***********1294  | a****@gmail.com  |
| t**    | 320***********8764  | t**@outlook.com  |
| a**    | 440***********3889  | a**@qq.com       |
+--------+---------------------+------------------+
```

7.5.2.6. 撤销列级权限设置

7.5.2.6.1. 撤销指定列的权限设置

语法

```
REVOKE PERMISSION ON [TABLE] <table_name> FOR COLUMN <column_name>;
```

示例

撤销 bank_account 表中，对 email 列的权限设置。

```
REVOKE PERMISSION ON bank_account FOR COLUMN email;
```

7.5.2.6.2. 撤销表中所有列的权限设置

语法

```
REVOKE PERMISSION ON [TABLE] <table_name> FOR COLUMNS;
```

示例

撤销 bank_account 表中，对所有列的权限设置。

```
REVOKE PERMISSION ON bank_account FOR COLUMNS;
```

7.5.2.7. 查看行/列级权限

语法

```
SHOW PERMISSION ON TABLE <table_name>;
```

> 如需查看表权限，可通过 **SHOW GRANT** 命令。

示例

```
> SHOW PERMISSION ON TABLE bank_account;
+---------------+-------------------------------------------------------------------+
|  row/column   |                              definition                           |
+---------------+-------------------------------------------------------------------+
| COLUMN name   | CASE WHEN HAS_ROLE('ADMIN') THEN name ELSE MASK(name, 1) END      |
| COLUMN id     | CASE WHEN HAS_ROLE('ADMIN') THEN id ELSE MASK(id, 3, 4) END       |
| COLUMN email  | CASE WHEN HAS_ROLE('ADMIN') THEN email ELSE MASK_EMAIL(email) END  |
+---------------+-------------------------------------------------------------------+
```

7.5.3. 空间配额管理操作

空间配额管理操作只能在InceptorServer 2中进行。

7.5.3.1. SQL 语法介绍

7.5.3.1.1. 设置配额

- 对某个database设置数据空间配额(执行者须为database的owner或具有admin角色)

```
GRANT QUOTA double_value(K|M|G|T) ON DATABASE db_name;
```

- 为某个用户设置使用某个Database数据空间的配额（执行者须为database的owner或具有admin角色）

```
GRANT QUOTA double_value(K|M|G|T) ON DATABASE db_name TO USER user_name;
```

- 为某个用户设置临时空间配额（执行者须有admin角色）

```
GRANT QUOTA double_value(K|M|G|T) ON TEMPORARY SPACE TO USER user_name;
```

- 设置所有临时空间总的配额（执行者须有admin角色）

```
GRANT QUOTA double_value(K|M|G|T) ON TEMPORARY SPACE;
```

7.5.3.1.2. 查看配额

- 查看某个database数据空间配额(执行者须为database的owner或具有admin角色)

```
SHOW QUOTA ON DATABASE db_name;
```

- 查看某个用户使用某个database数据空间的配额（执行者须为database的owner或具有admin角色）

```
SHOW QUOTA USER user_name ON DATABASE db_name;
```

- 查看某个用户具有的临时空间配额（执行者须是目标用户或者有admin角色）

```
SHOW QUOTA USER user_name ON TEMPORARY SPACE;
```

- 查看所有临时空间总的配额（执行者须有admin角色）

```
SHOW QUOTA ON TEMPORARY SPACE;
```

7.5.3.1.3. 取消配额

- 取消某个database的数据空间配额(执行者须为database的owner或具有admin角色)

```
GRANT QUOTA unlimited ON DATABASE db_name;
```

- 取消某个用户使用某个Database数据空间的配额(执行者须为database的owner或具有admin角色)

```
GRANT QUOTA unlimited  ON DATABASE db_name TO USER user_name;
```

- 取消某个用户临时空间配额（执行者须有admin角色）

```
GRANT QUOTA unlimited  ON TEMPORARY SPACE TO USER user_name;
```

- 取消所有临时空间总的配额（执行者须有admin角色）

```
GRANT QUOTA double_value(K|M|G|T) ON TEMPORARY SPACE;
```

7.5.3.2. 示例操作

注意，在GRANT QUOTA之前必须确保执行该操作的用户对于目标数据库具有ADMIN权限，否则报错。

7.5.3.2.1. 数据库数据空间配额管理

- 设置某个Database的数据空间配额

  GRANT QUOTA double_value(K|M|G|T) ON DATABASE db_name;

```
SET ROLE ADMIN;
GRANT QUOTA 2T ON DATABASE db1;
```

- 查看某个Database的数据空间配额

  SHOW QUOTA ON DATABASE db_name;

```
SHOW QUOTA ON DATABASE db1;
```

quota
2.000 TB

- 取消某个Database的空间配额设置，只需将上述语法中的QUOTA的值设置为 unlimited即可。

  GRANT QUOTA unlimited ON DATABASE db_name;

```
SET ROLE ADMIN;
GRANT QUOTA unlimited ON DATABASE db1;
```

```
SHOW QUOTA ON DATABASE db1;
```

quota
unlimited

7.5.3.2.2. 用户空间配额管理

- 设置某个用户使用某个database的数据空间的配额

  GRANT QUOTA double_value(K|M|G|T) ON DATABASE db_name TO USER user_name;

```
SET ROLE ADMIN;
GRANT QUOTA 1T ON DATABASE db1 TO USER user1;
```

- 查看某个用户使用某个database的数据空间的配额

  SHOW QUOTA USER user_name ON DATABASE db_name;

```
SHOW QUOTA USER user1 ON DATABASE db1;
```

quota
1.000 TB

- 取消某个用户使用某个database的数据空间的配额

  GRANT QUOTA unlimited ON DATABASE db_name TO USER user_name;

```
SET ROLE ADMIN;
GRANT QUOTA unlimited ON DATABASE db1 TO USER user1;
```

```
SHOW QUOTA USER user1 ON DATABASE db1;
```

quota
unlimited

7.5.3.2.3. 临时空间配额管理

用户临时空间配额管理

- 设置某个用户的临时空间配额

  GRANT QUOTA double_value(K|M|G|T) ON TEMPORARY SPACE TO USER user_name;

```
SET ROLE ADMIN;
GRANT QUOTA 500G ON TEMPORARY SPACE TO USER user1;
```

- 查看某个用户的临时空间配额

  SHOW QUOTA USER user_name ON TEMPORARY SPACE;

```
SHOW QUOTA USER user1 ON TEMPORARY SPACE;
```

quota
500.000 GB

- 取消某个用户的临时空间配额

  GRANT QUOTA unlimited ON TEMPORARY SPACE TO USER user_name;

```
SET ROLE ADMIN;
GRANT QUOTA unlimited ON TEMPORARY SPACE TO USER user1;
```

```
SHOW QUOTA USER user1 ON TEMPORARY SPACE;
```

quota
unlimited

所有临时空间配额管理

- 设置所有临时空间总的配额

  GRANT QUOTA double_value(K|M|G|T) ON TEMPORARY SPACE;

```
SET ROLE ADMIN;
GRANT QUOTA 2T ON TEMPORARY SPACE;
```

- 查看所有临时空间总的配额

  SHOW QUOTA ON TEMPORARY SPACE;

```
SHOW QUOTA ON TEMPORARY SPACE;
```

- 取消所有临时空间总的配额

  GRANT QUOTA double_value(K|M|G|T) ON TEMPORARY SPACE;

```
SET ROLE ADMIN;
GRANT QUOTA unlimited ON TEMPORARY SPACE;
```

```
SHOW QUOTA ON TEMPORARY SPACE;
```

quota
unlimited

7.5.4. 管理 HDFS 读写权限

在Inceptor中对HDFS目录的操作都是以hive用户来进行的，例如创建表在HDFS的warehouse目录下的目录、向该目录中写入数据等。其他用户对这些文件并没有读写权限。

在某些情况下，用户有对某张表的 **SELECT** 权限，并期望能够直接从该表在HDFS的目录中读取到对应表的数据，因此用户需要这些HDFS文件的权限（**r, w, x**），使得该用户能够直接获取HDFS上相应的数据。

Inceptor中提供了通过SQL设置表对应的HDFS目录的FACL()。从{initials}4.6开始支持对用户组设置ACL。

HDFS需要开启ACL功能——参数 **dfs.namenode.acls.enabled** 的值应该为 **true**。您可以在管理界面的HDFS配置页面查看和修改这个参数。

7.5.4.1. 设置用户/组对某张表的FACL

语法：设置用户/组对某张表的FACL

```
GRANT
  FACL '<permissions>' ①
  ON TABLE <table>
  TO USER|GROUP <user_or_group_name>; ②
```

① **<permissions>** 需要写作 **rwx**，**rw-**，**r-x**，**-wx**，**r--**，**-w-**，**--x** 或 **---**，并且放在单引号或者双引号之间。

② 和授予TABLE/VIEW/DATABASE权限不同，HDFS目录的权限一次只能授予一个用户/组。

设置用户/组对某张表的FACL所需权限

- **ADMIN** 和表owner可以将任意权限赋予任何人或者组。
- 对表有 **SELECT** 权限的用户可以赋予 **r-x** 给自己。
- 对表有 **INSERT** 权限的用户可以赋予 **-w-** 给自己。
- 既具有 **SELECT** 又具有 **INSERT** 权限的用户可以赋予 **rwx** 给自己。

例 21. 设置用户bob对表alice_t1的FACL

```
GRANT FACL 'rwx' ON TABLE alice_t1 TO USER bob;
```

7.5.4.2. 取消用户/组对某张表的FACL

语法：取消用户/组对某张表的FACL

```
REVOKE
  FACL
  ON TABLE <table>
  FROM USER|GROUP <user_or_group_name>;
```

7.5.4.3. 取消用户/组对某张表的FACL所需权限

- **ADMIN** 和表的Owner可以将任意用户/组对该表的FACL取消。
- 普通用户只能取消自己对该表的FACL。

7.5.4.4. 取消用户组bob对表alice_t1的FACL

```
REVOKE FACL ON TABLE alice_t1 FROM USER bob;
```

7.5.4.5. 取消某张表上的全部FACL

语法：取消某张表上的全部FACL

```
REVOKE
  FACL
  ON TABLE <table_name>;
```

取消某张表上的全部FACL所需权限

只有 **ADMIN** 或者表的owner可以执行该命令

7.5.4.6. 查看用户/组对某张表的FACL

语法：查看用户/组对某张表的FACL

```
SHOW
  FACL
  USER|GROUP <user_or_group_name>
  ON TABLE <table_name>;
```

查看用户/组对某张表的FACL所需权限

ADMIN、表的owner或者用户自己有权限执行该命令。

例 22. 查看用户bob对表alice_t1的FACL

重新赋予bob对alice_t1的 **rwx** 权限，并查看。

```
GRANT FACL 'rwx' ON TABLE alice_t1 TO USER bob;
SHOW FACL USER bob ON TABLE alice_t1;
```

facl
user:bob:rwx
default:user:bob:rwx

该结果反映出以下两点信息：

用户bob对表alice_t1在HDFS上的目录有 **rwx** 权限。

用户bob对alice_t1的目录的权限可以向下传递到alice_t1目录的子目录。

7.5.4.7. 查看某张表上的所有FACL

语法：查看某张表上的所有FACL

```
SHOW
  FACL
  ON TABLE <table_name>;
```

查看某张表上的所有FACL所需权限

ADMIN 表owner有权限执行该命令。

例 23. 查看表alice_t1上的所有FACL

```
GRANT FACL 'rwx' ON TABLE alice_t1 TO GROUP sales;
SHOW FACL ON TABLE alice_t1;
```

facl
user:bob:rwx
group::r-x
group:sales:rwx
default:user::rwx
default:user:bob:rwx
default:group::r-x
default:group:sales:rwx
default:mask::rwx
default:other::r-x

该结果反映出以下两点信息：

用户组sales对表alice_t1在HDFS上的目录有 **rwx** 权限。

用户组sales对alice_t1的目录的权限可以向下传递到alice_t1目录的子目录。

8. SQL 参考

本章节展示了本产品中的 SQL 语法的基础功能，并提供每种功能的通用语法，以及具体案例供读者进行操作测试。

主要包含以下部分：

- DDL 语法（Data Definition Language）：数据定义语言

  用来定义数据库对象(数据库，表，字段）。

- DML 语法（Data Manipulation Language）：数据操作语言

  用来对数据库表中的数据进行增删改。

- DQL 语法（Data Query Language）：数据查询语言

  用来查询数据库中表的记录。

- DCL 语法（Data Control Language）：数据库控制语言。

  是用来设置或更改数据库用户或角色权限。本产品支持使用 DCL 语法，但我们推荐使用 Guardian 实现控制数据访问权限，它提供了完整的安全解决方案，全面覆盖了用户认证、服务资源授权以及配额管理。

我们向您提供 Inceptor 中重点功能的使用方法及相关案例，主要包含：冷热数据分离、Compact/小文件合并、事务控制、ROWKEY、动态分区、函数与运算符。

我们还提供 Inceptor 中兼容 Oracle 的 PL/SQL 指南以及兼容 DB2 的 SQL PL 指南。

本产品支持完整的 SQL 标准语法，兼容 Oracle、IBM DB2、Teradata 方言，以及 Oracle 和 DB2 存储过程。平滑迁移数据默认的数据库方言是 Oracle，用户通过修改数据库参数的方式切换数据库方言。参数配置的具体方法如下：

不同方言的产品表现有所不同，因此推荐使用同一种方言。不推荐在一个 SQL 中混合使用多种方言。

当使用不同方言操作不同对象（例如table、view 等）时，需要更改服务配置并重启生效。

参数配置的具体方法如下：

参数/值/方言	Oracle	IBM DB2	Teradata
server.dialect	Oracle	DB2	Teradata
client.dialect	Oracle	DB2	Teradata

8.1. 数据定义语言（DDL）

DDL 用于创建、删除、修改和清空数据库中的对象，包括数据库、表、分区、分桶、视图等等。 DDL 语句包含： ALTER … 、CREATE … 、DROP … 、COMMENT、RENAME。

按照操作对象，我们主要展示本产品中定义数据库、定义表、定义分区、定义视图。

- 本产品只支持在创建表时定义分桶，且不支持修改分桶信息。
- 本产品不支持 CREATE USER 语句，请在 Guardian 中统一进行**用户及角色的管理**。具体操作，见多租户

指南。

8.1.1. 定义数据库

数据库是组织、存储和管理数据的仓库，而定义数据库主要包括创建数据库、修改数据库属性以及删除数据库。

另外我们还向你提供使用、显示和描述数据库的基本操作方式。

8.1.1.1. CREATE DATABASE

功能描述

- 使用 CREATE DATABASE 创建一个新的数据库。

注意事项

- 建议根据业务系统和项目创建 DATABASE，并通过 Guardian 分别设置不同用户的权限。
- DATABASE 的创建者默认为 DATABASE 的 OWNER。
- 如果 <database_name> 指定的数据库已经存在，会报错， 加上 IF NOT EXISTS 选项在建库前检测是否已存在同名库。
- 在安全模式下，在指定的 HDFS 路径创建数据库的用户必须是这个 HDFS 路径的 OWNER。

语法格式

```
CREATE DATABASE [IF NOT EXISTS] <database_name>--[1]
[COMMENT '<text>'] --[2]
[WITH DBPROPERTIES ('<property_name>'='<property_value>', ...)]--[3]
```

参数说明

- <database_name>：数据库名称，支持中文、英文、数字、下划线，但是不能只使用数字； 最大长度 128 字节。
- <text>：注释信息。
- <property_name>：数据库属性名称。
- <property_value>：数据库属性值。
- 可选项[1]，如果 <database_name> 指定的数据库已经存在，会报错，加上 IF NOT EXISTS 选项在建库前检测是否已存在同名库。
- 可选项[2]，用 [COMMENT] 为数据库加注释，注意注释要放在引号中。
- 可选项[3]，用于添加一些键值来定义数据库属性。

示例

例 24. 创建数据库 db_test

```
CREATE DATABASE IF NOT EXISTS db_test;
```

推荐阅读

- 修改数据库属性
- 删除数据库
- 使用数据库
- 展示数据库
- 描述数据库

8.1.1.2. ALTER DATABASE

功能描述

- 使用 ALTER DATABASE 修改数据库的所有者、数据库属性。
- 不建议您重命名数据库，如果一定需要进行此操作，须更改参数 inceptor.rename.database.enabled = true，再对数据库的名字进行修改。

注意事项

- 只有拥有数据库所有者权限的用户才能执行 ALTER DATABASE 命令，系统管理员默认拥有此权限。
- 如果需要查询指定数据库的属性，可以使用 DESC DATABASE 命令查看。

语法格式

- 修改数据库 DBPROPERTIES

```
ALTER DATABASE <database_name> SET DBPROPERTIES ('<property_name>'='<property_value>', ...);
```

- 修改数据库 OWNER

```
--修改数据库所属者
ALTER DATABASE <database_name> SET OWNER [USER|ROLE] <user_or_role>;--[1]
```

参数说明

- <database_name>：已存在的数据库的名称。
- <property_name>：数据库的属性名称。
- <property_value>：数据库属性值。
- <user_or_role>：更改数据库所属者,如果可选项 [1] 中为 USER 则此处取用户名；若为 ROLE 则此处取角色名。
- 可选项[1]，数据库 OWNER 可以是的用户也可以是角色。

示例

例 25. 修改数据库 DBPROPERTIES、所属者

```
--修改数据库日期 date 为'2015-12'
ALTER DATABASE db_test SET DBPROPERTIES ('date'= '2015-12');


--修改数据库所属者为 rachel
ALTER DATABASE db_test SET OWNER USER rachel;
```

推荐阅读

- 创建数据库
- 删除数据库
- 使用数据库
- 展示数据库
- 描述数据库

8.1.1.3. DROP DATABASE

功能描述

- 使用 DROP DATABASE 删除已存在的指定数据库。

注意事项

- 只有数据库所有者有权限执行 DROP DATABASE 命令，系统管理员默认拥有此权限。
- 不能对系统默的数据库执行删除操作。如果想查看当前服务中有哪几个数据库，可以用 SHOW DATABASES 命令查看。
- 执行删除数据库命令，需要该数据库为空。否则上述语句执行失败，需要添加 CASCADE 来实现。

语法格式

- 数据库为空时

```
DROP DATABASE [IF EXISTS] <database_name>;--[1]
```

- 数据库非空

```
DROP DATABASE [IF EXISTS] <database_name> CASCADE;
```

参数说明

- <database_name>：已存在且指定需要删除的数据库的名称。
- 可选项[1]，如果 <database_name> 指定的数据库不存在，会报错，加上 IF EXISTS 选项避免报错。

示例

例 26. 删除数据库

```
DROP DATABASE IF EXISTS db_test_rename;
```

推荐阅读

- 创建数据库
- 修改数据库属性
- 使用数据库
- 展示数据库
- 描述数据库

8.1.1.4. USE DATABASE

功能描述

- 使用 USE 转换当前数据库，运行之后的操作将默认在指定的目标数据库中执行。

注意事项

- 用户需要具备该数据库的相关权限。

语法格式

```
USE <database_name>;
```

参数说明

- <database_name>：已存在且需要使用的数据库的名称。

示例

```
USE argo_sql_test;
```

推荐阅读

- 创建数据库
- 修改数据库属性
- 删除数据库
- 展示数据库
- 描述数据库

8.1.1.5. SHOW DATABASES

功能描述

- 使用 SHOW DATABASES 展示当前集群中的所有数据库。

语法格式

```
SHOW DATABASES;
```

推荐阅读

- 创建数据库
- 修改数据库属性
- 删除数据库
- 使用数据库
- 描述数据库

8.1.1.6. DESCRIBE DATABASE

功能描述

- 使用 DESCRIBE DATABASE 查看已存在的数据库的名称、注释信息、位置、所属者名称及类型、参数等信息。

语法格式

```
DESCRIBE|DESC DATABASE --[1]
[EXTENDED] db_name; --[2]
```

参数说明

- <database_name>：已存在的数据库的名称。
- 可选项 [1]，描述数据库的关键字支持使用 DESC 缩写。
- 可选项 [3]，添加关键字 EXTENDED 展示完整信息。

示例

```
DESCRIBE DATABASE argo_sql_test;
```

输出示例：

```
+-------------+-------+-------------------------------------------------------------+
----------+----------+----------+
|   db_name   |comment|                             location
|owner_name|owner_type|parameters|
+-------------+-------+-------------------------------------------------------------+
----------+----------+----------+
|argo_sql_test|SQL测试 |hdfs://nameservice1/quark3/user/hive/warehouse/argo_sql_test.db| admin
| USER     |          |
+-------------+-------+-------------------------------------------------------------+
----------+----------+----------+
```

推荐阅读

- 创建数据库
- 修改数据库属性
- 删除数据库
- 展示数据库
- 描述数据库

8.1.2. 定义表

表是数据库中的一种特殊数据结构，用于存储数据对象以及对象之间的关系。 所涉及的 SQL 语句，主要包含创建表、修改表、删除表以及清空表数据。

另外我们还向你提供显示和描述表的基本操作方式。

8.1.2.1. CREATE TABLE

- 使用 CREATE TABLE 基础建表语句直接建表（建表时定义列），也可以选择添加 LIKE 或 AS SELECT 关键字实现复制源表或子查询的方式快速建表。
- 使用 PARTITION BY 创建分区表，通过 PARTITION BY RANGE 创建范围分区表。
- 使用 CLUSTERED BY 创建分桶表。
- 使用 CREATE TEMPORARY TABLE 创建临时表。
- 使用 CREATE EXTERNAL TABLE 创建外表
- 3.2版本中，当表小于1G时，可以在 TABLE 前增加 DIMENSION 参数，这样小表占用的资源更少。

```
CREATE DIMENSION TABLE <small_table_name>...;
```

8.1.2.1.1. 基本语法创建表

功能描述

- 使用 CREATE TABLE columns 直接定义列的方式进行建表。
- 使用 CREATE TABLE LIKE 通过拷贝一张已存在的表的方式实现快速建表，但不拷贝已存在表的数据。也就是说快速建立一张与源表定义相同的空表。
- 使用子查询 CREATE TABLE AS SELECT（CTAS）的方式将查询的结果重新定义成一张新表。也就是说通过 CTAS 方式建立的表含有子查询的结果数据。

注意事项

- CTAS 不能用于创建分区表、分桶表，使用 CTAS 建表时也不可以定义列。

语法格式

- 定义列建表

```
CREATE TABLE [IF NOT EXISTS] --[1]
[<database_name>.]<table_name> --[2]
(<column_name1> <DATATYPE1> [NOT NULL] [COMMENT "<text>"], --[3]
<column_name2> <DATATYPE2> [NOT NULL] [COMMENT "<text>"], --[4]
<column_name3> <DATATYPE3> [NOT NULL] [COMMENT "<text>"],
...)
STORED AS HOLODESK
[WITH TABLET NUMBER <tablet_number> CAPACITY NUMBER <capacity_number> REPLICATION
<replication_number>]--[5]
[WITH TABLESIZE <table_size>KB|MB|GB|TB|PB|EB|ZB|BB]--[6]
[TBLPROPERTIES|SERDEPROPERTIES ("<property_name>"="<property_value>", ...)];--[7]
```

- 使用子查询建表（CTAS）

```
CREATE TABLE [IF NOT EXISTS] --[1]
[<database_name>.]<table_name> --[2]
STORED AS HOLODESK
AS SELECT <select_statement>;
```

- 通过拷贝源表快速建表

```
CREATE TABLE [IF NOT EXISTS] --[1]
[<database_name>.]<table_name> --[2]
LIKE <souse_table_name>;
```

参数说明

- <database_name>：将表建在指定的数据库中。
- <table_name>：表名称。
- <column_name>：列名称。
- <DATATYPE>：数据类型。
- <text>：注释信息。
- <table_size>：表大小。
- <tablet_number>：磁盘数。
- <capacity_number>：存储空间数量。
- <replication_number>：表的副本数，推荐为奇数。
- <property_name>：表属性名称，包括。
- <property_value>：表属性值。
- <select_statement>：子查询语句结果。其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。

  FROM 子句指定的表中的列

  Literals，比如数字或者字符串

  表达式，如 employee.salary+1000

  函数（包括 Inceptor 自带函数和用户自定义函数）的调用，比如 count(*)
- <souse_table_name>：源表名。
- 可选项[1]，如果 <table_name> 指定的表已经存在，会报错，加上 IF NOT EXISTS 选项在建表前检测是否已存在同名表。
- 可选项[2]，支持指明表所在的数据库名称。
- 可选项[3]，支持非空 [NOT NULL] 约束。

- 可选项[4]，用 [COMMENT] 为表加注释，注意注释要放在引号中。
- 可选项[5]，支持对表的磁盘数 tablet_number、存储空间数量 capacity_number 和副本数 REPLICATION 进行设定。需要注意这三者需要同时都进行设定。

> 磁盘数 tablet_number、存储空间数量 capacity_number 和副本数 REPLICATION 三者必须同时进行设定。

- 可选项[6]，支持对表的大小 table_size 进行设定，若您需要同时设定表大小以及磁盘数量，必须先设定磁盘数，再设定表大小。
- 可选项[7]，支持使用 TBLPROPERTIES 或 SERDEPROPERTIES 对表的属性进行设定。

示例

例 27. 创建一张普通表

```
CREATE TABLE t1(trans_id INT, acc_num INT, trans_time DATE, trans_type STRING, stock_id STRING,
price DECIMAL, amount INT)
STORED AS HOLODESK;
```

例 28. 创建一张大小为 10kb 的表

```
CREATE TABLE t_10KB(trans_id INT, acc_num INT, trans_time DATE, trans_type STRING, stock_id
STRING, price DECIMAL,amount INT)
STORED AS HOLODESK
with tablesize 10kb;
```

例 29. 创建一张磁盘数为 10 的表

```
CREATE TABLE t_10tablet(trans_id INT, acc_num INT, trans_time DATE, trans_type STRING, stock_id
STRING, price DECIMAL,amount INT)
STORED AS HOLODESK
WITH TABLET NUMBER 10 CAPACITY NUMBER 2 REPLICATION 3;
```

例 30. 创建一张磁盘数为 3，且大小为 1 GB 的表

```
CREATE TABLE t_3tablet_1GB(trans_id INT, acc_num INT, trans_time DATE, trans_type STRING,
stock_id STRING, price DECIMAL,amount INT)
STORED AS HOLODESK
WITH TABLET NUMBER 10 CAPACITY NUMBER 2 REPLICATION 3
with tablesize 1GB;
```

例 31. 快速建表

```
CREATE TABLE t7_copy LIKE t1;

-- 查询 t7_copy 发现没有数据。
+-----------+----------+-------------+-------------+-----------+--------+--------+
| trans_id  | acc_num  | trans_time  | trans_type  | stock_id  | price  | amount |
+-----------+----------+-------------+-------------+-----------+--------+--------+
+-----------+----------+-------------+-------------+-----------+--------+--------+
```

例 32. 子查询建表

```
CREATE TABLE t_ctas STORED AS HOLODESK as SELECT * from  t1;

-- 查询 t_ctas 发现存在查询结果数据（源表 t1 中的数据）
+-----------+----------+-------------+-------------+-------------+--------+--------+
| trans_id  | acc_num  | trans_time  | trans_type  |  stock_id   | price  | amount |
+-----------+----------+-------------+-------------+-------------+--------+--------+
| 1         | 1        | 2014-02-01  | discount    | NO20140201  | 10     | 10     |
| 1         | 2        | 2023-02-28  | online      | NO20230228  | 12.11  | 10     |
| 1         | 3        | 2019-10-01  | wholesale   | NO20191001  | 8.99   | 10     |
+-----------+----------+-------------+-------------+-------------+--------+--------+
```

8.1.2.1.2. 创建分区表

功能描述

- 使用 CREATE TABLE PARTITION BY 创建单值分区表。
- 使用 CREATE TABLE PARTITION BY RANGE 创建范围分区表。

注意事项

- 范围分区表只能使用直接定义列的方式建表。
- 分区以文件目录的形式存在，且分区列是一个虚拟列。因此分区键不能存在于表的字段中，但可以作为一个“伪”列在查询中使用。

语法格式

- 创建单值分区表

```
CREATE [IF NOT EXISTS] --[1]
[<database_name>.]<table_name>( --[2]
<column_name1> <DATATYPE1> [NOT NULL] [COMMENT "<text>"], --[3]
<column_name2> <DATATYPE2> [NOT NULL] [COMMENT "<text>"], --[4]
<column_name3> <DATATYPE3> [NOT NULL] [COMMENT "<text>"],
...)
partitionED BY(<partition_key1> <DATATYPE>, <partition_key2> <DATATYPE>, ...)
STORED AS HOLODESK
[TBLPROPERTIES|SERDEPROPERTIES ("<property_name>"="<property_value>", ...)];--[5]
```

- 创建范围分区表

```
CREATE TABLE [IF NOT EXISTS]  --[1]
[<database_name>.]<table_name>( --[2]
<column_name1> <DATATYPE1> [NOT NULL] [COMMENT "<text>"], --[3]
<column_name2> <DATATYPE2> [NOT NULL] [COMMENT "<text>"], --[4]
<column_name3> <DATATYPE3> [NOT NULL] [COMMENT "<text>"],
...)
partitionED BY RANGE (<partition_key1> <DATATYPE>, <partition_key2> <DATATYPE>, ...)
(partition <partition_name1> VALUES LESS THAN (<key1_value1>, <key2_value2>,...),
partition <partition_name2> VALUES LESS THAN (<key1_value1>, <key2_value2>,...),
...
[partition <partition_name> VALUES LESS THAN (MAXVALUE, MAXVALUE,...)] --[5]
)
STORED AS HOLODESK
[TBLPROPERTIES|SERDEPROPERTIES ("<property_name>"="<property_value>", ...)]; --[6]
```

参数说明

- 〈database_name〉：将表建在指定的数据库中。
- 〈table_name〉：表名称。

- <column_name>：列名称。
- <DATATYPE>：数据类型。
- <text>：注释信息。
- <partition_key>：分区键。
- <partition_name>：范围分区中的指定某个分区名称。
- <value>：范围分区中一个分区的上限值。
- MAXVALUE：当最后一个分区的值是 MAXVALUE 时，它将不能再增加分区。
- <table_size>：表大小。
- <tablet_number>：磁盘数。
- <capacity_number>：存储空间数量。
- <replication_number>：表的副本数，推荐为奇数。
- <property_name>：表属性名称。
- <property_value>：表属性值。
- 可选项[1]，如果 <table_name> 指定的表已经存在，会报错，加上 IF NOT EXISTS 选项在建表前检测是否已存在同名表。
- 可选项[2]，指明表所在的数据库名称。
- 可选项[3]，支持非空 [NOT NULL] 约束。
- 可选项[4]，用 [COMMENT] 为表加注释，注意注释要放在引号中。。
- 可选项[5]，可以选择使用 MAXVALUE 设置范围分区表的最后一个分区，之后该分区表将不能再添加分区。
- 可选项[6]，支持使用 TBLPROPERTIES 或 SERDEPROPERTIES 对表的属性进行设定。

示例

例 33. 创建一张单值分区表

```
CREATE TABLE t3_partition(trans_id INT, acc_num INT, trans_type STRING, stock_id STRING, price
DECIMAL, amount INT)
PARTITIONED BY(trans_time DATE)
STORED AS HOLODESK;
```

例 34. 创建一张范围分区表

```
CREATE TABLE t4_range_partition(trans_id INT, acc_num INT, trans_type STRING, stock_id STRING,
price DECIMAL, amount INT)
PARTITIONED BY RANGE (trans_time DATE) (
PARTITION before2014_02 VALUES LESS THAN ('2014-02-28'),
PARTITION before2014_03 VALUES LESS THAN ('2014-03-31')
)
STORED AS HOLODESK;
```

例 35. 创建一张单值多重分区表

```
CREATE TABLE multi_partition(acc_num INT, trans_type STRING, stock_id STRING, price DECIMAL,
amount INT)
PARTITIONED BY(trans_time DATE, trans_id INT)
STORED AS HOLODESK;
```

例 36. 创建一张范围多重分区表

```
CREATE TABLE multi_range_partition(acc_num INT, trans_type STRING, stock_id STRING, price
DECIMAL, amount INT)
PARTITIONED BY RANGE (trans_time DATE, trans_id INT) (
PARTITION a VALUES LESS THAN ('2014-02-28', 3),
PARTITION b VALUES LESS THAN ('2014-03-31', 5)
)
STORED AS HOLODESK;
```

8.1.2.1.3. 创建分桶表

功能描述

- 使用 CREATE TABLE CLUSTERED BY 创建分桶表。

注意事项

- 分桶只能在创建分桶表中直接定义，不支持对已有分桶表进行修改。
- 创建分桶表时可以不使用 INTO <num_buckets> BUCKETS 指定分桶数，分桶数默认为比磁盘数 *5 大的质数。

语法格式

```
CREATE TABLE [IF NOT EXISTS]  --[1]
[<database_name>.]<table_name>( --[2]
<column_name1> <DATATYPE1> [NOT NULL] [COMMENT "<text>"], --[3]
<column_name2> <DATATYPE2> [NOT NULL] [COMMENT "<text>"], --[4]
<column_name3> <DATATYPE3> [NOT NULL] [COMMENT "<text>"],
...)
CLUSTERED BY (<bucket_column1>, <bucket_column1>, ...) [INTO <num_buckets> BUCKETS]--[5]
STORED AS HOLODESK
[WITH TABLET NUMBER <tablet_number> CAPACITY NUMBER <capacity_number> REPLICATION
<replication_number>]--[6]
[WITH TABLESIZE <table_size>KB|MB|GB|TB|PB|EB|ZB|BB]--[7]
[TBLPROPERTIES|SERDEPROPERTIES ("<property_name>"="<property_value>", ...)];--[8]
```

参数说明

- <database_name>：将表建在指定的数据库中。
- <table_name>：表名称。
- <column_name>：列名称。
- <DATATYPE>：数据类型。
- <text>：注释信息。
- <num_buckets>：分桶数。
- <bucket_column>：分桶列。
- <table_size>：表大小。

- <tablet_number>：大表 |FACT TABLE 的 TABLET 数量为磁盘数；小表 |DIMENSION TABLE 的 TABLET 数量为节点数；创建表时，默认为大表。
- <capacity_number>：存储空间数量。
- <replication_number>：表的副本数，推荐为奇数。
- <property_name>：表属性名称。
- <property_value>：表属性值。
- 可选项[1]，如果 <table_name> 指定的表已经存在，会报错，加上 IF NOT EXISTS 选项在建表前检测是否已存在同名表。
- 可选项[2]，指明表所在的数据库名称。
- 可选项[3]，支持非空 [NOT NULL] 约束。
- 可选项[4]，用 [COMMENT] 为表加注释，注意注释要放在引号中。
- 可选项[5]，可以使用 INTO <num_buckets> BUCKETS 指定分桶数.也可以选择不指定，则默认分桶数为比磁盘数 *5 大的质数。
- 可选项[6]，支持对表的磁盘数 tablet_number、存储空间数量 capacity_number 和副本数 REPLICATION 进行设定。

> 磁盘数 tablet_number、存储空间数量 capacity_number 和副本数 REPLICATION 三者必须同时进行设定。

- 可选项[6]，支持对表的大小 table_size 进行设定，若您需要同时设定表大小以及磁盘数量，必须先设定磁盘数，再设定表大小。
- 可选项[7]，支持使用 TBLPROPERTIES 或 SERDEPROPERTIES 对表的属性进行设定。

示例

例 37. 创建一张分桶表

```
CREATE TABLE t5_bucket(trans_id INT, acc_num INT, trans_time DATE, trans_type STRING, stock_id
STRING, price DECIMAL, amount INT)
CLUSTERED BY (trans_id)
STORED AS HOLODESK;
```

8.1.2.1.4. 创建临时表

临时表可分为局部临时表和全局临时表：

- **局部临时表 TEMPORARY TABLE** 仅在当前 session 可见，当前 session 结束后会被删除。如果一张临时表和一张永久表重名，在临时表所在的 session 中该表名将指代临时表，同名永久表将无法在 session 中访问。临时表不支持分区。
- **全局临时表 GLOBAL TEMPORARY TABLE** 在当前 session1 创建后，其他 session 可见，并可以使用，当 session1 结束后，会等待其他使用到该表的 session结 束使用后删除。全局临时表无法与永久表重名，且其他 session 无法创建同名。全局临时表不支持分区。

功能描述

- 使用 CREATE TEMPORARY TABLE 命令创建局部临时表。
- 使用 CREATE GLOBAL TEMPORARY TABLE 命令创建全局临时表。

语法格式

```
CREATE [GLOBAL] TEMPORARY TABLE  --[1]
[IF NOT EXISTS] -- [2]
[<database_name>.]<table_name>( --[3]
<column_name1> <DATATYPE1> [NOT NULL] [COMMENT "<text>"], --[4]
<column_name2> <DATATYPE2> [NOT NULL] [COMMENT "<text>"], --[5]
<column_name3> <DATATYPE3> [NOT NULL] [COMMENT "<text>"],
...)
STORED AS HOLODESK
[TBLPROPERTIES|SERDEPROPERTIES ("<property_name>"="<property_value>", ...)]; --[6]
```

参数说明

- <database_name>：将表建在指定的数据库中。
- <table_name>：表名称。
- <column_name>：列名称。
- <DATATYPE>：数据类型。
- <text>：注释信息。
- 可选项[1]，选择创建局部临时表 TEMPORARY TABLE，或是全局临时表 GLOBAL TEMPORARY TABLE。
- 可选项[2]，如果 <table_name> 指定的表已经存在，会报错，加上 IF NOT EXISTS 选项在建表前检测是否已存在同名表。
- 可选项[3]，指明表所在的数据库名称。
- 可选项[4]，支持非空 [NOT NULL] 约束。
- 可选项[5]，用 [COMMENT] 为表加注释，注意注释要放在引号中。
- 可选项[6]，支持使用 TBLPROPERTIES 或 SERDEPROPERTIES 对临时表的属性进行设定。

示例

例 38. 创建临时表

```
-- 创建一张局部临时表
CREATE TEMPORARY TABLE t2_tmp(trans_id INT, acc_num INT, trans_time DATE, trans_type STRING,
stock_id STRING, price DECIMAL, amount INT)
STORED AS holodesk;

-- 创建一张全局临时表
CREATE global TEMPORARY TABLE global_tmp(trans_id INT, acc_num INT, trans_time DATE, trans_type
STRING, stock_id STRING, price DECIMAL, amount INT)
STORED AS holodesk;
```

8.1.2.1.5. 创建外表

外表（External table）是一种特殊类型的数据库对象，它定义了一个可读的数据源，但它不是数据库中的表，而是指向一个外部数据源中数据的元数据。外表提供了一种将外部数据文件与数据库相关联的方法，可以在 SQL 查询中像操作普通表一样使用它们。

外表常用于以下场景中：

- 数据的导入与导出：可以使用外表将外部数据源中的数据导入到数据库中或者将数据库中的数据导出到外部数据文件中，比如可以将 Excel 或 CSV 格式的数据文件导入到 Inceptor 数据库中。
- 数据库数据的备份和恢复：可以将数据库中的数据备份到外部数据文件中，然后再将其恢复到数据库中。

- 数据的转换和集成：可以使用外表将多个数据源中的数据进行集成，并通过 SQL 查询来获取需要的数据。

需要注意的是，使用外表需要确保外部数据源的数据格式与外表的定义相符，否则会出现查询失败等问题。

语法格式

```
CREATE [TEMPORARY] EXTERNAL TABLE  --[1]
[IF NOT EXISTS] -- [2]
[<database_name>.]<table_name>( --[3]
<column_name1> <DATATYPE1> [NOT NULL] [COMMENT "<text>"], --[4]
<column_name2> <DATATYPE2> [NOT NULL] [COMMENT "<text>"], --[5]
<column_name3> <DATATYPE3> [NOT NULL] [COMMENT "<text>"],
...)
STORED AS HOLODESK
[TBLPROPERTIES|SERDEPROPERTIES ("<property_name>"="<property_value>", ...)]; --[6]
```

参数说明

- <database_name>：将表建在指定的数据库中。
- <table_name>：表名称。
- <column_name>：列名称。
- <DATATYPE>：数据类型。
- <text>：注释信息。
- 可选项[1]，添加 TEMPORARY 关键字创建临时表
- 可选项[2]，如果 <table_name> 指定的表已经存在，会报错，加上 IF NOT EXISTS 选项在建表前检测是否已存在同名表。
- 可选项[3]，指明表所在的数据库名称。
- 可选项[4]，支持非空 [NOT NULL] 约束。
- 可选项[5]，用 [COMMENT] 为表加注释，注意注释要放在引号中。
- 可选项[6]，支持使用 TBLPROPERTIES 或 SERDEPROPERTIES 对临时表的属性进行设定。

示例

例 39. 创建外表

```
CREATE EXTERNAL TABLE external_t(trans_id INT, acc_num INT, trans_time DATE, trans_type STRING,
stock_id STRING, price DECIMAL, amount INT)
STORED AS holodesk;
```

推荐阅读

- 修改表
- 删除表
- 清空表数据
- 展示表
- 描述表

8.1.2.2. ALTER TABLE

本节主要向您介绍：

- 修改表名称。
- 设置表的属性参数。表属性分别由两个参数 TBLPROPERTIES 和 SERDEPROPERTIES 定义。
- 更改表的列结构，包含增加列、删除列，指定列的位置顺序。
- 对指定表中改变列的相关信息，如列的名称，数据类型等。
- 修改表和列的注释信息。
- 设置保护属性。
- （仅 orc、text、torc、rcfile、parquet、csv 格式支持）修改表或分区的位置路径。
- 手动合并表碎片。

推荐阅读

- 创建表
- 删除表
- 清空表数据
- 展示表
- 描述表

8.1.2.2.1. 重命名表

功能描述

- 使用 RENAME 关键字修改表名称。

语法格式

```
ALTER TABLE <table_name_old> RENAME TO <table_name_new>;
```

参数说明

- <table_name_old>：表原有名称。
- <column_name>：列名称。
- <DATATYPE>：数据类型。
- <table_name_new>：表重命名之后的新名字

示例

```
ALTER TABLE t7_copy RENAME TO t7_rename;
```

8.1.2.2.2. 设置表属性

功能描述

- 使用 TBLPROPERTIES 设置整个表的一些属性信息，如表的压缩格式、表的存储位置、表的格式、表的注释及 设置 ROWKEY 等。
- 使用 SERDEPROPERTIES 用于描述表中数据的序列化和反序列化格式，包括分隔符、数据类型等等

语法格式

```
ALTER TABLE <table_name> SET TBLPROPERTIES|SERDEPROPERTIES --[1]
('<property_name>' = '<property_value>' ... );
```

参数说明

- <table_name>：表名称。
- <column_name>：列名称。
- <DATATYPE>：数据类型。
- <property_name>：表属性名称。
- <property_value>：表属性值。
- 可选项[1]，使用 TBLPROPERTIES 或 SERDEPROPERTIES 对表的属性进行设定。

示例

```
ALTER TABLE t2_tmp SET TBLPROPERTIES ('date'= '2015-12-34')
```

8.1.2.2.3. 增减列

功能描述

- 使用 ALTER TABLE ADD 增加指定表中的列。
- 使用 ALTER TABLE DROP 删除增加指定表中的列。

注意事项

- ADD|DROP COLUMNS 每次都只能增加或删除一列，且不能删除分区列、分桶列或含有 Global Index 的列。
- ADD COLUMNS 可以将新的列加入到表中。列的位置不能定义更改，默认有如下三种情况：

  对于非分区表，新增加列的默认位置位置在所有列之后；

  对于范围分区表，新增加的列位置在所有列之后；

  对于单值分区表，新增加的列位置在所有列之后，分区列之前。

语法格式

增加列：

```
ALTER TABLE <table_name> ADD COLUMNS
(<column_name> <DATATYPE> [NOT NULL]--[1]
[COMMENT '<text>'] --[2]
[CASCADE|RESTRICT]);--[3]
```

删除列

```
ALTER TABLE <table_name> DELETE COLUMNS (<column_name>);
```

参数说明

- <table_name>：表名称。
- <column_name>：列名称。
- <column_name_table>：目标表中字段名称。
- <text>：注释信息。
- <DATATYPE>：数据类型。
- 可选项[1]，支持非空 [NOT NULL] 约束。
- 可选项[2]，用 [COMMENT] 为数据库加注释，注意注释要放在引号中。
- 可选项[3]，CASCADE 表示将修改字段元信息的影响扩展至与其相关的分区；而 RESTRICT 表示仅影响该字段从属的表的元信息。

示例

例 40. 增加列

```
--增加列 add_col, 数据类型为 INT
ALTER TABLE t1 ADD COLUMNS (add_col INT comment '增加列');
```

例 41. 删除列

```
--删除 t1 中的列 add_col
ALTER TABLE t1 DELETE COLUMNS (add_col);
```

8.1.2.2.4. 修改列 CHANGE COLUMN

功能描述

- 使用 ALTER CHANGE COLUMN 修改表中的列名称，数据类型，注释信息。

注意事项

- 该指令仅仅修改表的元数据。
- 重名名列只允许修改非分区键、非分桶键之外的其他普通列的名称。
- 修改字段数据类型，为强制类型转换，有数据丢失的风险，需谨慎使用。

  只允许同一种数据类型，增加字段精度或规模的修改。如 char(1) 更改为 char(10)、decimal(10,2) 更改为 decimal(15,2)。

  降低字段精度或规模可能导致数据精度的损失，是不被允许的。如 char(2) 更改为 char(1)，或 decimal(15,2) 更改为 decimal(10,2)。

  不允许一种类型更改为另一种类型的转换，如 boolean 更改为 char(10)。
- 不支持一切更改已有数据格式的操作，如修改表中列的排序。

语法格式

```
ALTER TABLE <table_name> CHANGE [COLUMN] <col_old_name> <col_new_name> <column_type>--[1]
[COMMENT '<text>']--[2]
[CASCADE|RESTRICT]; --[3]
```

参数说明

- <table_name>：表名称。
- <col_old_name>：列的原名称
- <col_new_name>：列的新名词，可以和 <col_old_name> 相同。
- <DATATYPE>：数据类型，可以保持不变填写原有数据类型。
- <text>：注释信息。
- 可选项[1]，缩写时 COLUMN 可省略。
- 可选项[2]，用 [COMMENT] 为数据库加注释，注意注释要放在引号中。
- 可选项[3]，CASCADE 表示将修改字段元信息的影响扩展至与其相关的分区；而 RESTRICT 表示仅影响该字段从属的表的元信息。

示例

例 42. 重命名列

```
ALTER TABLE t7_rename CHANGE COLUMN trans_id trans_id_changed int COMMENT '重命名列';
```

例 43. 修改列的数据类型

```
ALTER TABLE t7_rename CHANGE COLUMN price price decimal(15,4) COMMENT '修改数据类型';
```

8.1.2.2.5. 改变注释

功能描述

- 使用 TBLPROPERTIES 的 'comment' 参数改变表注释信息。
- 使用 CHANGE [COLUMN] 改变指定表中某一个字段的注释信息。

注意事项

- 使用 CHANGE [COLUMN] 语法时，可以同时修改列名称或添加字段长度，具体操作请参考改变列的相关信息。

语法格式

定义表注释

```
ALTER TABLE [<database_name>.]<table_name> SET TBLPROPERTIES ('COMMENT' = '<text>');--[1]
```

删除表注释

```
ALTER TABLE [<database_name>.]<table_name> UNSET TBLPROPERTIES ('comment');
```

定义列注释

```
ALTER TABLE [<database_name>.]<table_name> CHANGE [COLUMN] <column_name> <column_name> <DATATYPE>
COMMENT '<text>';
```

删除列注释

```
ALTER TABLE [<database_name>.]<table_name> CHANGE [COLUMN] <column_name> <column_name> <DATATYPE>
COMMENT '';
```

参数说明

- <database_name>：将表建在指定的数据库中。
- <table_name>：表名称。
- <column_name>：列名称。
- <DATATYPE>：数据类型。
- <text>：注释信息。
- 可选项 [1]，可以指定目标表所处数据库名称。

实例

例 44. 添加表注释

```
ALTER TABLE t1 SET TBLPROPERTIES ('comment' = 'holo 表 t1 的注释');
desc create table t1; --查看建表语句命令，查看添加的注释信息。
```

例 45. 删除表注释

```
ALTER TABLE t1 UNSET TBLPROPERTIES ('comment');
desc formatted t1; --描述表详细信息命令，查看删除的注释信息。
```

例 46. 添加列注释

```
ALTER TABLE t1 change column trans_id trans_id int COMMENT 't1 表 trans_id 列的注释';
desc table t1; --查看列信息
```

例 47. 删除列注释

```
ALTER TABLE t1 change column trans_id trans_id int COMMENT '';
desc table t1; --查看列信息
```

8.1.2.2.6. 设置保护

功能描述

- 添加关键字 ENABLE NO_DROP，被指定的对象将无法被删除；

  而 DISABLE NO_DROP 则表示设置表或分区可以被删除。

- 添加关键字 ENABLE OFFLINE，指定表或分区无法被查询；

  而 DISABLE OFFLINE 则设置对象支持被查询。

注意事项

- 设置了保护之后的表，可以通过 DESC FORMATTED 中的字段 Protect Mode 查看。

语法格式

设置删除保护

```
--非分区表
ALTER TABLE [<database_name>.]<table_name> --[1]
ENABLE|DISABLE NO_DROP; --[2]

--单值分区
ALTER TABLE [<database_name>.]<table_name> PARTITION (<partition_key> = <value>) ENABLE|DISABLE
NO_DROP;

--范围分区
ALTER TABLE [<database_name>.]<table_name> PARTITION <partition_name> ENABLE|DISABLE NO_DROP;
```

设置查询保护

```
--非分区表
ALTER TABLE [<database_name>.]<table_name>  ENABLE|DISABLE OFFLINE;

--单值分区
ALTER TABLE [<database_name>.]<table_name> PARTITION (<partition_key> = <value>) ENABLE|DISABLE
OFFLINE;

--范围分区
ALTER TABLE [<database_name>.]<table_name> PARTITION <partition_name> ENABLE|DISABLE OFFLINE;
```

参数说明

- <database_name>：将表建在指定的数据库中。
- <table_name>：表名称。
- <partition_key>：分区键。
- <partition_name>：范围分区中的指定某个分区名称。
- <value>：分区对应取值。
- 可选项 [1]，可以指定目标表所处数据库名称。
- 可选项 [2]，ENABLE 表示开启保护，作用对象将无法被删除或查询；DISABLE 表示关闭保护，作用对象将被允许删除或查询。

示例

例 48. 表删除保护

```
ALTER TABLE t8_rename ENABLE NO_DROP;
DROP TABLE t8_rename; --此时会出现报错信息：
ALTER TABLE t8_rename DISABLE NO_DROP; --运行之后就能够正常删除该表
```

例 49. 单值分区删除保护

```
ALTER TABLE t3_partition PARTITION (trans_time = '2023-2-11') ENABLE NO_DROP;
ALTER TABLE t3_partition DROP PARTITION (trans_time = '2023-2-11'); --此时会出现报错信息：
ALTER TABLE t3_partition PARTITION (trans_time = '2023-2-11') DISABLE NO_DROP; --
运行之后就能够正常删除该分区
```

例 50. 范围分区删除保护

```
ALTER TABLE t4_range_partition PARTITION before2014_03 ENABLE NO_DROP;
ALTER TABLE t4_range_partition DROP PARTITION before2014_03; --此时会出现报错信息：
ALTER TABLE t4_range_partition PARTITION before2014_03 DISABLE NO_DROP; --
运行之后就能够正常删除该分区
```

例 51. 表查询保护

```
ALTER TABLE t8_rename ENABLE OFFLINE;
SELECT * FROM t8_rename;--查询报错
ALTER TABLE t8_rename DISABLE OFFLINE; --运行之后就能够正常查询
```

例 52. 单值分区查询保护

```
ALTER TABLE t3_partition ADD PARTITION (trans_time = '2023-2-11');

ALTER TABLE t3_partition PARTITION (trans_time = '2023-2-11') ENABLE OFFLINE;
SELECT * FROM t3_partition WHERE trans_time = '2023-2-11';--查询报错
ALTER TABLE t3_partition PARTITION (trans_time = '2023-2-11') DISABLE OFFLINE; --
运行之后就能够正常查询
```

例 53. 范围分区查询保护

```
ALTER TABLE t4_range_partition ADD PARTITION before2014_03 VALUES LESS THAN ('2014-03-31');

ALTER TABLE t4_range_partition PARTITION before2014_03 ENABLE OFFLINE;
SELECT * FROM t3_partition WHERE trans_time >= '2014-02-28' and trans_time < '2014-03-31'; --
查询报错
ALTER TABLE t4_range_partition PARTITION before2014_03 DISABLE OFFLINE; --运行之后就能够正常查询
```

8.1.2.2.7. 更改路径

功能描述

- 使用 ALTER TABLE SET LOCATION 设置表的 HDFS 路径。
- 使用 ALTER TABLE PARTITION SET LOCATION 设置表中某个分区的 HDFS 路径；

分区路径可以与其所在表的 HDFS 路径不同，如果不手动更改则默认存在表的 HDFS 路径下。

注意事项

- Holodesk 表暂时不支持修改路径功能。
- 需要具有 对应的 HDFS 路径的写入权限。具体查看方式如下：

  集群节点上运行 `hdfs dfs -ls <hdfs_path>` 查询路径的详细信息：

  输出结果的前三列分别是权限信息，路径拥有者，用户组。

  权限信息中第一位 d 表示这是一个目录，后面三组 rwx 分别表示文件拥有者、用户组和其他用户的读、写和执行权限。其中 r 表示读权限，w 表示写权限，x 表示执行权限，而 - 表示没有对应的权限。

语法格式

修改表路径

```
ALTER TABLE [<database_name>.]<table_name> SET LOCATION '<hdfs_path>';--[1]
```

修改单值分区路径

```
ALTER TABLE [<database_name>.]<table_name> PARTITION (<partition_key> = <value>) SET LOCATION
"<hdfs_path>";
```

修改范围分区路径

```
ALTER TABLE [<database_name>.]<table_name> PARTITION <partition_name> SET LOCATION "<hdfs_path>";
```

参数说明

- <database_name>：将表建在指定的数据库中。
- <table_name>：表名称。
- <hdfs_path>：HDFS 目录，可以是相对路径也可以是绝对路径。
- <partition_key>：分区键。
- <partition_name>：范围分区中的指定某个分区名称。
- <value>：分区对应取值。
- 可选项 [1]，可以指定目标表所处数据库名称。

示例

```
--创建 torc 分区分桶表
CREATE TABLE torc_location (a int, b double) partitioned by (p int) CLUSTERED BY (a) INTO 3
BUCKETS STORED  AS  ORC_TRANSACTION;
alter table torc_location add partition (p=1) partition (p=2);

-- 修改表路径
ALTER TABLE torc_location SET LOCATION '';

-- 修改分区路径
ALTER TABLE torc_location partition (p=1) SET LOCATION '';
```

8.1.2.3. DROP TABLE

功能描述

- 使用 DROP TABLE 删除指定的表。

注意事项

- 默认情况下，删除表会将表移入回收站，7天后才能彻底删除。

语法格式

```
DROP TABLE [IF EXISTS] <table_name> --[1]
[IGNORE PROTECTION];--[2]
```

参数说明

- <table_name>：表名称。
- 可选项[1]，如果 <table_name> 指定的数据库不存在，会报错，加上 IF EXISTS 选项避免报错。
- 可选项[2]，如果需要强制执行，忽略保护措施删除表格，可以添加 IGNORE PROTECTION 关键字。

示例

例 54. 删除表

```
DROP TABLE t7_rename;
```

推荐阅读

- 创建表
- 修改表属性
- 清空表数据
- 展示表
- 描述表

8.1.2.4. TRUNCATE TABLE

功能描述

- 使用 TRUNCATE TABLE 清空指定表中的所有数据，但不删除表的元数据。

语法格式

```
TRUNCATE TABLE <table_name>;
```

参数说明

- <table_name>：表名称。

示例

例 55. 清空表数据

```
--插入批量数据:
BATCHINSERT INTO t1 BATCHVALUES (
VALUES (1, 1, '2014-2-1', 'discount', 'NO20140201', 10.00,10),
VALUES(1, 2, '2023-2-28', 'online', 'NO20230228', 12.11,10),
VALUES (1, 3, '2019-10-1', 'wholesale', 'NO20191001', 8.99,1000));

--插入单条数据:
INSERT INTO t1 VALUES(0, 1, '2014-2-1', 'discount', 'NO20140201', 10.00,10);

--清空表数据:
TRUNCATE TABLE t1;

-- 查询表数据
SELECT * from t1;
```

推荐阅读

- 创建表
- 修改表属性
- 删除表
- 展示表
- 描述表

8.1.2.5. SHOW TABLES

功能描述

- 使用 SHOW TABLES 展示当前数据库中的所有表。
- SHOW TABLES 还会同时展示当前数据库中的所有视图。
- 使用 SHOW CREATE TABLE 展示指定表的建表信息及部分属性信息。

注意事项

- 展示建表信息时，可以设置参数使输出结果中不包含表的位置信息：

```
set inceptor.show.create.table.without.location=true;
```

语法格式

展示数据库中的全部表及视图

```
SHOW TABLES;
```

展示某一个表的建表信息及部分属性

```
SHOW CREATE TABLE <table_name>;
```

参数说明

- <table_name>：表名称。

示例

```
SHOW CREATE TABLE t1;
```

推荐阅读

- 创建表
- 修改表
- 删除表
- 清空表数据
- 描述表

8.1.2.6. DESCRIBE TBALE

功能描述

- 使用 DESCRIBE TABLE 命令查看表的列信息，包括列名、数据类型、分区分桶列、列注释等信息。
- 使用 DESCRIBE FORMATTED 命令查看指定表的完整信息，包括表所在数据库、用户、创建时间、表注释、分桶数 Num Buckets、分桶列 Bucket Columns、主键 Rowkey 等属性信息。
- 使用 DESCRIBE 还可以查看指定表中某一列的信息。

语法格式

查看表中所有列信息

```
DESCRIBE|DESC --[1]
[TABLE] <table_name>;--[2]
```

查看表的完整信息

```
DESCRIBE|DESC FORMATTED <table_name>;
```

查看目标表中指定某一列的信息

```
DESCRIBE|DESC <table_name>.<column_name>;
```

参数说明

- <table_name>：表名称。
- <column_name>：列名称。
- 可选项 [1]，描述关键字支持使用 DESC 缩写。
- 可选项 [2]，关键字 TABLE 可省略。

示例

```
DESC TABLE t1;
DESC FORMATTED t1;
DESCRIBE t1.acc_num;
```

推荐阅读

- 创建表
- 修改表
- 删除表
- 清空表数据
- 展示表

8.1.3. 定义分区

数据库分区是将数据量大的表的数据均摊到不同的硬盘、系统或不同服务器存储介质中（将数据从物理上分成若干个小表存储），实际上仍是一张表。 为了在特定的 SQL 操作中减少数据读写的总量以缩减响应时间，分区功能可以实现将表的数据均衡到不同的地方，提升查询效率。

本文所涉及的 SQL 语句，主要包含不同类型分区的添加、删除和清空表分区三种功能。另外还向您介绍了展示分区的操作方法。

- 单值分区和范围分区都支持**多重分区**字段，更多介绍，见多重分区表的创建。

分区分类

- 按照每个分区对应于分区键的取值，可分为：

  单值分区：使用 partition 语句实现创建单值分区表。分区键为一个固定的值 。

  范围分区：使用 partition by range 语句实现创建 Range 分区表，并通过 values less than 划分分区。分区键为一个区间，且范围分区只能添加新的分区区间不能分裂原有分区区间。

- 按照分区的创建方式，可分为：

  静态分区：是由用户手动创建并指定的。

  动态分区：是由写入数据中分区列的值自动判断目标分区，不需要用户手动指定分区。

  > 为避免误操作创建大量分区，一般情况下建议用户预先手动创建分区，再使用动态分区插入数据功能。

8.1.3.1. ADD PARTIOTION

功能描述

- 使用 ALTER TABLE ADD PARTIOTION 在指定表中添加分区。

注意事项

- 本产品不支持重命名分区。
- 范围分区只能添加新的分区，不能分裂原有分区。
- 只能向表中末尾的分区之后添加分区，并且不能与原有分区存在交集。

语法格式

- 添加单值分区

```
ALTER TABLE <table_name> ADD partition (<partition_key1> = <value>,<partition_key2> = <value>,...)
[partition (<partition_key1> = <value>,<partition_key2> = <value>,...)];--[1]
```

- 添加范围分区

```
ALTER TABLE <table_name> ADD partition <partition_name> VALUES LESS THAN (<value>|MAXVALUE) --[2]
[partition <partition_name> VALUES LESS THAN (<value>|MAXVALUE)];
```

参数说明

- <table_name>：表名称。
- <partition_key>：分区键。
- <partition_name>：范围分区中的指定某个分区名称。
- <value>：分区键对应的值。
- 可选项[2]，支持在一个 SQL 语句中，同时添加多个分区。
- 可选项[2]，范围分区表中，当最后一个分区的值是 MAXVALUE 时，将不能再增加分区。

示例

例 56. 添加单值分区

```sql
ALTER TABLE t3_partition ADD PARTITION (trans_time = '2023-2-11');

-- 添加多重单值分区
ALTER TABLE multi_partition ADD PARTITION (trans_time = '2023-2-11', trans_id = 1);

-- 添加多个分区
ALTER TABLE t3_partition ADD PARTITION (trans_time = '2023-2-12') PARTITION (trans_time =
'2023-2-13');
```

例 57. 添加范围分区

```sql
ALTER TABLE t4_range_partition ADD PARTITION before2014_04 VALUES LESS THAN ('2014-04-30');

-- 添加多重范围分区
ALTER TABLE multi_range_partition ADD PARTITION c VALUES LESS THAN ('2014-04-30', 10);

-- 添加最后一个范围分区
ALTER TABLE t4_range_partition ADD PARTITION last_date_partition VALUES LESS THAN (maxvalue);

-- 查看表中分区情况
show partitions t4_range_partition
```

推荐阅读

- 删除分区
- 清空分区

8.1.3.2. TRUNCATE TABLE PARTIOTION

功能描述

- 使用 TRUNCATE TABLE PARTIOTION 清空指定表中的指定分区目录下的数据，但保留分区。

语法格式

- 清空单值分区

```
TRUNCATE TABLE <table_name> partition (<partition_key> = <value>, <partition_key> = <value>,...
);
```

- 清空范围分区

```
TRUNCATE TABLE <table_name> partition <partition_name>;
```

参数说明

- 〈table_name〉：表名称。
- 〈partition_key〉：分区键。
- 〈partition_name〉：范围分区中的指定某个分区名称。
- 〈value〉：分区键对应的值。

示例

例 58. 清空分区表

- 清空单值分区表

```
--数据写入单值分区表
INSERT INTO TABLE t3_partition
PARTITION (trans_time = '2023-2-11')
SELECT t1.trans_id, t1.acc_num, t1.trans_type, t1.stock_id, t1.price, t1.amount FROM t1;

--清空单值分区
TRUNCATE TABLE t3_partition PARTITION (trans_time = '2023-2-11');
```

- 清空范围分区

```
--数据写入范围分区表
INSERT INTO TABLE t4_range_partition
PARTITION before2014_04
SELECT t1.trans_id, t1.acc_num, t1.trans_type, t1.stock_id, t1.price, t1.amount, t1
.trans_time FROM t1
WHERE t1.trans_time >= '2014-03-31' and t1.trans_time < '2014-04-30';


--清空范围分区表
TRUNCATE TABLE t4_range_partition PARTITION before2014_04;
```

推荐阅读

- 添加分区
- 删除分区

8.1.3.3. DROP PARTIOTION

功能描述

- 使用 ALTER TABLE DROP PARTIOTION 删除指定表中的指定分区及其数据。
- 使用 IGNORE PROTECTION 关键字忽略保护强制执行删除分区。

语法格式

- 删除单值分区

```
ALTER TABLE <table_name> DROP partition (<partition_key>=<value>, <partition_key> = <value>,...)
[IGNORE PROTECTION]; --[1]
```

- 删除范围分区

```
ALTER TABLE <table_name> DROP partition <partition_name> [IGNORE PROTECTION];
```

参数说明

- <table_name>：表名称。
- <partition_key>：分区键。
- <partition_name>：范围分区中的指定某个分区名称。
- <value>：分区键对应的值。
- 可选项 [1]，如果需要忽略分区保护，强制删除分区，可以通过添加 IGNORE PROTECTION 关键字实现。

示例

例 59. 删除分区

- 删除单值分区

```
ALTER TABLE t3_partition DROP PARTITION (trans_time = '2023-2-11');
```

- 删除范围分区

```
ALTER TABLE t4_range_partition DROP PARTITION before2014_04;
```

推荐阅读

- 添加分区
- 清空分区

8.1.3.4. SHOW PARTITIONS

功能描述

- 使用 SHOW PARTITIONS 展示指定目标表中的分区信息，包括分区键，分区名称以及分区对应取值。

语法格式

```
SHOW PARTITIONS <table_name>;
```

参数说明

- <table_name>：表名称。

示例

```
SHOW PARTITIONS t3_partition;
```

8.1.4. 定义视图

Inceptor 支持普通/物化视图，普通视图只是一层简单的查询代理，所涉及的 SQL 语句， 主要包含创建视图、改变视图定义或属性和删除视图。另外我们还向你提供显示、描述视图的基本操作方式。

物化视图的更多介绍，见定义物化视图。

- 视图由一个预定义的查询 SELECT 语句组成。
  1. 对视图可以像表一样进行 INSERT，UPDATE，DELETE 操作。
  2. 但普通视图是一个虚拟的表，是非实物化 (unmaterialized)，是一个表中的数据经过某种筛选后的显示方式。 也就是说视图中没有实际的数据，每次对视图进行查询时，创建视图所用的查询语句会被再次执行一次。
- 视图的数量没有限制，但是命名不能和视图以及表重复，具有唯一性。
- 视图可以被嵌套，一个视图中可以嵌套另一个视图。
- 视图不能有相关联的触发器和默认值，不能在视图后使用 ORDER BY 排序。
- 当普通视图对应的基本表修改后（如数据写入，增删列等），视图可以进行自动更新。

8.1.4.1. CREATE VIEW

功能描述

- 使用 CREATE VIEW AS SELECT 创建视图。

注意事项

- 在创建视图时可以选择定义列名，但是不能定义列类型，列类型由 AS SELECT <select_statement> 的查询结果决定。
- CREATE VIEW 的语法和 CTAS 非常相像。区别在于VIEW是非实体化的，CREATE VIEW 给查询创建一个快捷方式，而 CTAS 将查询结果写入磁盘中。
- 不支持 CREATE VIEW AS SELECT … ，UNION SELECT … 语句。

语法格式

```
CREATE VIEW [IF NOT EXISTS] <view_name> --[1]
[(<column_name>, <column_name>,...)]--[2]
AS SELECT <select_statement>;
```

参数说明

- <view_name>：视图名称。

- <column_name>：列名称。
- <select_statement>：子查询语句结果。
- 可选项[1]，如果 <view_name> 指定的视图已经存在，会报错，加上 IF NOT EXISTS 选项避免报错。
- 可选项[2]，支持定义视图的列名。

示例

例 60. 创建视图

```
CREATE VIEW IF NOT EXISTS v1 AS SELECT * FROM t1;
```

推荐阅读

- 改变视图定义或属性
- 删除视图
- 物化视图

8.1.4.2. ATLER VIEW

功能描述

- 使用 ATLER VIEW AS SELECT 改变指定视图的定义。
- 使用 ALTER VIEW SET TBLPROPERTIES 设定指定视图的自定义属性。

注意事项

- 在改变一个视图的定义前，该视图的定义必须已经被创建过。
- 如果一个视图有分区，那么它不能通过 ALTER VIEW AS SELECT替换。

语法格式

- 改变视图定义

```
ALTER VIEW <view_name> AS SELECT <select_statement>;
```

- 设置视图属性TBLPROPERTIES

```
ALTER VIEW <view_name> SET TBLPROPERTIES ('<property_name>'='<property_value>');
```

参数说明

- <view_name>：视图名称。
- <select_statement>：子查询语句结果。
- <property_name>：视图属性名称。
- <property_value>：视图属性值。

示例

例 61. 改变视图

- 改变视图定义

```
ALTER VIEW v1 AS SELECT * FROM t8_sub_t1;
```

- 改变视图 TBLPROPERTIES

```
ALTER VIEW v1 SET TBLPROPERTIES ('date'='2023-2-8');
```

推荐阅读

- 创建视图
- 删除视图
- 物化视图

8.1.4.3. DROP VIEW

功能描述

- 使用 DROP VIEW 删除视图。

注意事项

- DROP VIEW 将指定视图的元数据删除。虽然视图和表有很多共同之处，但是 DROP TABLE 不能用来删除 VIEW。
- 如果删除的视图被其他视图所使用，不会给出任何警告。
- 依靠于被删除视图的视图会变成无效视图，但是用户需要自行处理这些变成无效视图的视图。

语法格式

```
DROP VIEW [IF EXISTS] <view_name>; --[1]
```

参数说明

- <view_name>：视图名称。
- 可选项[1]，如果 <view_name> 指定的视图已经不存在，会报错，加上 IF EXISTS 选项避免报错。

示例

例 62. 删除视图

```
DROP VIEW IF EXISTS v1;
```

推荐阅读

- 创建视图

- 改变视图定义或属性
- 物化视图

8.1.4.4. 物化视图

物化视图是一种一般用于加速特定查询速度的缓存。可以对其进行 创建、查看以及删除。

- 物化视图在某种意义上说就是一个物理表(且不仅仅是一个物理表)，这通过其可以被 SHOW TABLES 查询出来，而得到佐证
- 物化视图也是一种段（segment），所以具有自己的物理存储属性
- 物化视图会占用数据库磁盘空间，这点从 user_segment 的查询结果，可以得到佐证
- 物化视图中不支持 order by，limit 子句
- 尚不支持使用 ALTER MATERIALIZED VIEW 修改物化视图
- 尚不支持物化视图基于对应的源表的更新（如数据写入，增删列等）进行自动更新功能

物化视图与普通视图的区别与优势：

1. 普通视图：是一个通过定义封装 SQL 查询语句的虚拟表，没有实际的物理存储。 每次对视图进行查询时，创建视图所用的查询语句会被再次执行一次。这实际上是把视图查询转化成对基本表的查询，如果这个视图是由一个复杂的多表查询所定义，那么，即使是视图的一个简单查询，也要把它变成一个复杂的结合体，需要花费一定的时间。
2. 物化视图：是一个比较特殊的物理表，是有实际的物理存储的，是将实际需要关联很复杂的数据转化成一张表，这样数据查询就只需要查这个物化视图，不需要执行创建视图时的复制语句，这将会降低资源的消耗。

推荐阅读

- 创建普通视图
- 修改普通视图
- 删除普通视图

8.1.4.4.1. 创建物化视图

功能描述

- 使用 CREATE MATERIALIZED VIEW 创建物化视图

语法格式

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] <materialized_view_name>  --[1]
[ENABLE/DISABLE REWRITE]--[2]
[COMMENT "<text>"] --[3]
[VIEWPARTITION (<partition_key1> = <value>,<partition_key2> , ...)]--[4]
[TABLEPROPERTIES ('<property_name>'='<property_value>', ...)] --[5]
AS SELECT <select_statement>;
```

参数说明

- <materialized_view_name>：视图名称。
- <text>：表注释。
- <partition_key>：分区键。

- <value>：分区对应取值。
- <property_name>：视图属性名称。
- <property_value>：视图属性值。
- <select_statement>：子查询语句结果。
- 可选项[1]，如果 <materialized_view_name> 指定的物化视图已经存在，会报错，加上 IF NOT EXISTS 选项避免报错。
- 可选项[2]，其中 ENABLE REWRITE 指能参与 MBO 的 SQL REWRITE，DISABLE REWRITE 指不参与 MBO 的 SQL REWRITE，不指定则默认为 DISABLE REWRITE。
- 可选项[3]，用 [COMMENT] 为数据库加注释，注意注释要放在引号中。
- 可选项[4]，支持对物化视图添加分区。
- 可选项[5]，支持设置物化视图的属性。

示例

例 63. 创建物化视图

```
CREATE MATERIALIZED VIEW IF NOT EXISTS materialized_v1 AS SELECT * from t1;
```

8.1.4.4.2. 查看物化视图

功能描述

- 使用 SHOW 可以查询当前数据库中的所以物化视图。
- 使用 DESC FORMATTED 查看物化视图的属性信息，查询结果中的部分属性信息及含义如下所示：

属性名称	含义
View Original Text	创建 materialized view 时指定的 query 字符串
View Expanded Text	是 View Original Text 的扩展形式，主要是为所有列加上限定名等。后续使用 materialized view 时主要基于该字符串,而不是View Original Text.
View Rewrite Enabled	是否允许用改写该 materialized view
View Source Tables	该 materialized view 中涉及到的源表名称
View Out Of Date	该 materialized view 是否过期，如果源表的数据在建 materialized view 之后有更新，那表示该 materialized view 已过期。

语法格式

- 查看指定物化视图的属性信息

```
DESC FORMATTED <materialized_view_name>;
```

- 查看当前数据库中的所有物化视图

```
-- 查看当前数据库所有表，包括 materialized view
SHOW TABLES;

-- 查看当前数据库所有 materialized view
SHOW MATERIALIZED VIEWS;
```

参数说明

- <materialized_view_name>：物化视图名称。

示例

例 64. desc FORMATTED

```
DESC FORMATTED materialized_v1;
```

例 65. show

```
show tables;
-- 查询结构如下
+------------------------+
|        tab_name        |
+------------------------+
| btable                 |
| datatable              |
| loop_employee          |
| materialized_v1        |
| multi_partition        |
| multi_range_partition  |
| no_loop_employee       |
| t1                     |
+------------------------+

 SHOW MATERIALIZED VIEWS;
-- 查询结构如下
+------------------+
|     tab_name     |
+------------------+
| materialized_v1  |
+------------------+
```

8.1.4.4.3. 删除物化视图

功能描述

- 使用 DROP MATERIALIZED VIEW 删除指定物化视图。

语法格式

```
DROP MATERIALIZED VIEW [IF EXISTS] <materialized_view_name>;
```

参数说明

- <materialized_view_name>：物化视图名称。
- 可选项[1]，如果 <materialized_view_name> 指定的物化视图不存在，会报错，加上 IF EXISTS 选项避免报错。

示例

例 66. 删除物化视图

```
DROP MATERIALIZED VIEW materialized_v1;
```

8.1.4.5. SHOW VIEWS

功能描述

- 使用 SHOW VIEWS 查看当前数据库中的所有普通视图。
- 使用 SHOW MATERIALIZED VIEWS 查看当前数据库中的所有物化视图。

注意事项

- 使用 SHOW TABLES 可以查看当前数据库中的所有表、视图以及物化视图。

语法格式

```
SHOW VIEWS;
SHOW MATERIALIZED VIEWS;
```

8.1.4.6. DESC VIEW

功能描述

- 使用 DESCRIBE 查看目标视图查询输出列的信息。
- 使用 DESCRIBE FORMATTED 查看目标视图的完整信息，包括列信息，所处数据库，所属者信息，表类型，创建视图的原始语句等。

注意事项

- 使用 SHOW TABLES 可以查看当前数据库中的所有表、视图以及物化视图。

语法格式

```
DESCRIBE|DESC [TABLE] <view_name>;--[1]


DESCRIBE|DESC FORMATTED <view_name>;
```

参数说明

- <view_name>：视图（包括物化视图）的名称。
- 可选项 [1]：可以选择省略关键字 TABLE。

示例

```
desc v1;
desc formatted materialized_v1;
```

8.2. 数据操作语言(DML)

DML 用于操作现有数据库表的数据。主要涉及的 SQL 语句有： INSERT、UPDATE、DELETE、MERGE INTO

8.2.1. INSERT

使用 INSERT 命令执行数据插入操作，按照插入数据的场景，所涉及的 SQL 语句， 主要包含插入单条数据、批量插入多条数据、子查询插入数据、插入数据至分区表。

推荐阅读:

- 修改数据
- 插入并修改数据
- 删除数据

8.2.1.1. INSERT INTO|OVERWRITE VALUES

功能描述

- 使用 INSERT INTO|OVERWRITE VALUES 实现单表单条数据的写入，一般适用于少量数据的手动写入场景。

语法格式

```
INSERT INTO|OVERWRITE <table_name>--[1]
[PARTITION (<partition_key>=<value>...) --[2]
[IF NOT EXISTS]] --[3]
VALUES(<value1>, <value2>,...);
```

参数说明

- <table_name>：表名称。
- <partition_key>：分区键。
- <value>：分区对应取值。
- 可选项[1]，INSERT OVERWRITE 覆盖表中原数据； INSERT INTO 不覆盖表中原数据。
- 可选项[2]，可以将数据插入分区表的指定分区中。
- 可选项[3]，如果 <partition_key> 指定的分区键已经存在，Inceptor 会报错，加上 IF NOT EXISTS 选项避免报错。

示例

例 67. 插入单条数据

```
INSERT INTO t1 VALUES(2, 1, '2014-2-1', 'discount', 'NO20140201', 10.00,10);
```

推荐阅读

- 批量插入多条数据
- 子查询插入数据

- 插入数据至分区表

8.2.1.2. 批量插入多条数据

功能描述

- 使用 BATCHINSERT 或 INSERT 语法手动实现批量插入功能

注意事项

- 目前支持百列千行级别的 BATCHINSERT

语法格式

- INSERT语法

```
INSERT INTO|OVERWRITE <table_name>  --[1]
[PARTITION (<partition_key>=<value>...) --[2]
[IF NOT EXISTS]] --[3]
VALUES (<value1>, <value2>, <value3>,...), (<value1>, <value2>, <value3>,...), (<value1>,
<value2>, <value3>,...),...;
```

- BATCHINSERT语法

```
BATCHINSERT INTO|OVERWRITE <table_name>
[PARTITION (<partition_key>=<value>...)
[IF NOT EXISTS]]
BATCHVALUES (VALUES (<value1>, <value2>, <value3>,...),
VALUES(<value1>, <value2>, <value3>,...),
VALUES (<value1>, <value2>, <value3>,...),
...);
```

参数说明

- <table_name>：表名称。
- <column_name>：列名。
- <partition_key>：分区键。
- <value>：分区对应取值。
- 可选项[1]，INSERT OVERWRITE 覆盖表中原数据； INSERT INTO 不覆盖表中原数据。
- 可选项[2]，可以将数据插入分区表的指定分区中。
- 可选项[3]，如果 <partition_key> 指定的分区键已经存在，Inceptor 会报错，加上 IF NOT EXISTS 选项则可以让 Inceptor 不报错。

示例

例 68. 批量写入

```
-- BATCHINSERT 语法
BATCHINSERT INTO t1 BATCHVALUES (
VALUES (3, 1, '2014-2-1', 'discount', 'NO20140201', 10.00,10),
VALUES(3, 2, '2023-2-28', 'online', 'NO20230228', 12.11,10),
VALUES (3, 3, '2019-10-1', 'wholesale', 'NO20191001', 8.99,1000));


-- INSERT 语法
INSERT INTO t1 VALUES(4, 1, '2022-2-1', 'online', 'NO20220201', 10.24,11),(4, 2, '2023-2-11',
'shop', 'NO20230211', 15.50,22),(4, 3, '2022-2-22', 'wholesale', 'NO20220222', 9.19,33);
```

推荐阅读

- 插入单条数据
- 子查询插入数据
- 插入数据至分区表

8.2.1.3. 子查询插入数据

功能描述

- 使用 INSERT SELECT 将子查询结果数据从源表写入目标表中。
- 支持使用一个数据源将多个查询结果分别插入不同表中。

语法格式

```
INSERT INTO|OVERWRITE <table_name1> --[1]
[PARTITION (<partition_key>=<value>...) --[2]
[IF NOT EXISTS]] --[3]
(<column_name1>, <column_name1>,...)
SELECT <select_statement> FROM <source>;
```

参数说明

- <table_name>：表名称。
- <column_name>：列名。
- <select_statement>：子查询语句结果。其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。
- <source>：数据源。
- <partition_key>：分区键。
- <value>：分区对应取值。
- 可选项[1]，INSERT OVERWRITE 覆盖表中原数据； INSERT INTO 不覆盖表中原数据。
- 可选项[2]，可以将数据插入分区表的指定分区中。
- 可选项[3]，如果 <partition_key> 指定的不存在存在，Inceptor 会报错，加上 IF NOT EXISTS 选项避免报错。

示例

例 69. 子查询插入数据

```
INSERT INTO t8_sub_t1 select * from t1;
```

推荐阅读

- 插入单条数据
- 批量插入多条数据
- 插入数据至分区表

8.2.1.4. 插入数据至分区表

- 按照分区表中的分区键值为固定值或是一个区间，本节分为数据写入单值分区表和数据写入范围分区表。
- 按照分区表中的分区键的值是否确定，分区表还可以分为静态分区表、动态分区表。此处仅展示静态分区插入功能，动态分区插入功能请查看动态分区章节。
- 分区表中插入数据适用场景为：

  表的大小超过 2GB。

  表中包含历史数据，新的数据被增加到新的分区中。

功能描述

- 使用 INSERT INTO TABLE PARTITION 的方式将数据插入指定的分区中
- 支持手动插入的方式将数据写入进静态分区表中，只需要注意写入的数据符合与分区表的分区列相对应。

注意事项

- 在插入数据时，需手动指定分区。
- 在向单值静态分区表中插入数据时，由于已经指定了分区键的具体值，所以不需要查询分区列插入。

  但在向范围静态分区表中插入数据时，由于并没有指定分区列的具体值，所以需要查询分区列插入，并且分区列应放置在最后且该值需要处于范围分区以内。

语法格式

- 数据写入**单值**静态分区表

```
INSERT INTO TABLE <table_name>
PARTITION (<partition_key1> = <value>[, <partition_key2> = <value>, ...])
SELECT <select_statement> FROM <source>;
[source,sql]
```

- 数据写入**范围**静态分区表

```
INSERT INTO TABLE <table_name>
PARTITION <partition_name>
SELECT <select_statement> FROM <source> WHERE <filter_statement>
```

参数说明

- <table_name>：表名称。
- <select_statement>：子查询语句结果，其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。
- <partition_key>：分区键。
- <partition_name>：范围分区中的指定某个分区名称。
- <value>：对应取值。
- <filter_statement>：过滤条件语句，即需要过滤符合分区范围的值。

示例

例 70. 数据写入单值静态分区

```
--添加单值分区
ALTER TABLE t3_partition ADD PARTITION (trans_time= '2014-2-11');
ALTER TABLE t3_partition ADD PARTITION (trans_time= '2014-04-30');


-- 子查询写入
INSERT INTO TABLE t3_partition
PARTITION (trans_time = '2014-2-11')
SELECT t1.trans_id, t1.acc_num, t1.trans_type, t1.stock_id, t1.price, t1.amount FROM t1;

-- 手动写入单条数据
INSERT INTO t3_partition
PARTITION (trans_time = '2014-04-30')
VALUES (5, 1, 'discount', 'NO20140201', 10.00,10);

--手动批量写入
INSERT INTO t3_partition
PARTITION (trans_time = '2014-04-30')
VALUES(6, 1, 'sales', 'NO20140201', 10.00,10), (6, 2, 'sales', 'NO20230228', 12.11,10),(6, 3,
'sales', 'NO20191001', 8.99,1000);
```

例 71. 数据写入范围静态分区

```
--添加范围分区
ALTER TABLE t4_range_partition ADD PARTITION before2014_05 VALUES LESS THAN ('2014-05-30');

--子查询写入
INSERT INTO TABLE t4_range_partition
PARTITION before2014_05
SELECT t1.trans_id, t1.acc_num, t1.trans_type, t1.stock_id, t1.price, t1.amount, t1.trans_time
FROM t1 WHERE t1.trans_time >= '2014-04-30' and t1.trans_time < '2014-05-30';

-- 手动写入单条数据
INSERT INTO t4_range_partition
PARTITION before2014_05
VALUES (3, 1, 'discount', 'NO20140201', 10.00,10, '2014-5-3');

--手动批量写入
INSERT INTO t3_partition
PARTITION before2014_05
VALUES(4, 1, 'sales', 'NO20140201', 10.00,10, '2014-5-4'), (4, 2, 'sales', 'NO20230228', 12.11
,10, '2014-5-5'),(6, 3, 'sales', 'NO20191001', 8.99,1000, '2014-5-6');
```

推荐阅读

- 插入单条数据
- 批量插入多条数据

- 子查询插入数据

8.2.2. UPDATE

功能描述

- 使用 UPDATE TABLE 修改指定表指定列中的数据。
- 本产品既支持对单列数据进行修改，也可以使用查询语句对多列数据批量修改。

注意事项

- 如果需要对分区表中的数据进行修改时，需要修改参数 hive.crud.dynamic.partition=true（默认 false）。

语法格式

- 修改单列的值

```
UPDATE <table_name>
[PARTITION <partition_name>] --[1]
[PARTITION (<partition_key> = <value>] --[2]
SET <column_name> = <value>
[PARTITION <partition_name>]
[PARTITION (<partition_key> = <value>]
[WHERE <filter_statement>];
```

- 修改多列的值

```
UPDATE  <table_name>
[PARTITION <partition_name>]
[PARTITION (<partition_key> = <value>]
SET (<column_name1>, <column_name2>, ...) = (SELECT <select_statement>)
[WHERE <filter_statement>];
```

参数说明

- <table_name>：表名称。
- <column_name>：列名。
- <select_statement>：子查询语句结果。其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。
- <filter_statement>：过滤条件语句。
- <partition_name>：范围分区表中的指定分区的名称。
- <partition_key>：分区键。
- <value>:分区键对应值。
- 可选项[1]，可以对范围分区表的指定分区中的数据进行修改。
- 可选项[2]，可以对单值分区表的指定分区中的数据进行修改。

示例

例 72. 修改单列数据

```
--普通表
UPDATE TABLE t1  SET trans_type = 'update';

--单值分区表
UPDATE t3_partition partition (trans_time='2014-2-11') SET trans_id=7 WHERE amount= 10;

--范围分区表
UPDATE t4_range_partition partition before2014_05 SET trans_id=7 WHERE amount= 10;
```

例 73. 修改多列数据

```
--普通表
UPDATE t1 SET trans_id=10, amount=1000 || 'changed' WHERE trans_date > '2014-2';

--范围分区表
UPDATE t4_range_partition partition before2014_05 SET trans_id=9, trans_type=  trans_type|| '
changed' WHERE amount= 10;

--单值分区表
UPDATE t3_partition partition (trans_time='2014-2-11') SET trans_id=8, trans_type=  trans_type
|| ' changed' WHERE amount= 10;
```

推荐阅读

- 插入数据
- 插入并修改数据
- 删除数据

8.2.3. MERGE INTO

功能描述

- 对一个表同时执行 INSERT 和 UPDATE 操作。MERGE 命令从一个或多个数据源中选择行来 UPDATE 或 INSERT 到一个或多个表。
- MERGE INTO 的作用就是解决用 B 表更新 A 表的数据。如果A表中没有，则把 B 表的数据插入 A 表或向一个表中插入数据，如果该表已有该数据则更新，反之新增数据。

注意事项

- MERGE 的源表必须有化名。
- 两表连接判定条件 <merge_condition> 的对应关系，必须为一一对应。
- UPDATE 和 INSERT 子句是可选的，也可以加 WHERE 子句。
- 在 ON 条件中使用常量过滤谓词来 INSERT 所有的行到目标表中,不需要连接源表和目标表。
- UPDATE 子句后面可以跟 DELETE 子句来去除一些不需要的行。
- MERGE INTO 中所有的 UPDATE、INSERT、DELETE 都是针对目标表来操作的。由于 MERGE INTO 已经指定了操作的表，所以 UPDATE、INSERT、DELETE 都不需要再显示指定表名。

语法格式

```
-- 非分区表
MERGE INTO <table_name1> [table_rename1] --[1]
USING [table|view|subquery] <table_name2> <table_reanme2> --[2]
ON (<merge_condition>)
WHEN MATCHED THEN <update_statement>
WHEN NOT MATCHED THEN <insert_statement>;

-- 单值分区表
MERGE INTO <table_name1>
PARTITION (<partition_key> = <value>)
USING [table|view|subquery] <table_name> <table_reanme>
ON (<merge_condition>)
WHEN MATCHED THEN <update_statement>
WHEN NOT MATCHED THEN <insert_statement>;

-- 范围分区表
MERGE INTO <table_name>
partition <patition_name>
USING [table|view|subquery] <table_name> <table_reanme>
ON (<merge_condition>)
WHEN MATCHED THEN <update_statement>
WHEN NOT MATCHED THEN <insert_statement>;
```

参数说明

- <table_name1>：目标表名称。
- <table_name2>：源表名称。
- <table_reanme>：必须需对源表指定别名。
- <patition_name>：范围分区分区名称。
- <partition_key>：分区键。
- <value>：分区键对应值。
- <merge_condition>：两表连接判定条件，需要注意此处必须为一一对应。
- <update_statement>：更新目标表数据语句。
- <insert_statement>：目标表插入数据语句。
- 可选项[1]，目标表也可以指定别名。
- 可选项[2]，选择源数据类型为表，视图或是子查询语句。

示例

例 74. 插入并更新非分区表

```
--创建源表并插入数据
CREATE TABLE t11_source(trans_id INT, acc_num INT,  trans_type STRING, stock_id STRING, price
DECIMAL, amount INT, trans_time DATE)
STORED AS HOLODESK;
BATCHINSERT INTO t11_source BATCHVALUES (VALUES (7, 1, 'discount', 'NO2019098', 10.00,10,'2014-
04-30'),VALUES(7, 2, 'online', 'NO20237889', 12.11,10,'2014-04-30'),VALUES (7, 3, 'wholesale',
'NO20140430', 8.99,1000,'2014-04-30'));

-- merge into 普通表
MERGE INTO t8_sub_t1 target
USING t1 source ON (target.stock_id=source.stock_id)
WHEN matched THEN UPDATE SET target.trans_type = target.trans_type || ' MERGE update'
WHEN NOT MATCHED THEN INSERT (trans_id, acc_num, trans_type, stock_id, price, amount) VALUES
(source.trans_id, source.acc_num, source.trans_type || ' MERGE update', source.stock_id,
source.price, source.amount);
```

例 75. 插入并更新单值分区表

```
MERGE INTO t3_partition PARTITION (trans_time='2014-2-11') target
USING t11_source source ON (target.stock_id=source.stock_id)
WHEN matched THEN UPDATE SET target.trans_type = target.trans_type || ' MERGE update'
WHEN NOT matched THEN INSERT (trans_id, acc_num, trans_type, stock_id, price, amount) VALUES
(source.trans_id, source.acc_num, source.trans_type || ' MERGE update', source.stock_id,
source.price, source.amount);
```

例 76. 插入并更新范围分区表

```
MERGE INTO t4_range_partition PARTITION before2014_05 target
USING t11_source source ON (target.stock_id=source.stock_id)
WHEN matched THEN UPDATE SET target.trans_type = target.trans_type || ' MERGE update'
WHEN NOT matched THEN INSERT (trans_id, acc_num, trans_type, stock_id, price, amount,
trans_time) VALUES (source.trans_id, source.acc_num, source.trans_type || ' MERGE update',
source.stock_id, source.price, source.amount, source.trans_time);
```

推荐阅读

- 插入数据
- 修改数据
- 删除数据

8.2.4. DELETE

功能描述

- 使用 DELETE FROM WHERE 删除指定表中的对应数据。

语法格式

```
-- 非分区表
DELETE FROM <table_name> WHERE <filter_statement>;

-- 删除单值分区表中一个分区的部分内容
DELETE FROM <table_name> partition (<partition_key> = <value>) WHERE <filter_statement>;

-- 删除范围分区表中一个分区的部分内容
DELETE FROM <table_name> partition <partition_name> WHERE <filter_statement>;
```

参数说明

- <table_name>：表名称。
- <column_name>：列名。
- <partition_key>：分区键。
- <partition_name>：范围分区中的指定某个分区名称。
- <filter_statement>：过滤条件语句。
- <value>：分区对应取值。

示例

例 77. 删除数据

```
-- 非分区表
DELETE FROM t1 WHERE acc_num >= 2;

-- 删除单值分区表中一个分区的部分内容
DELETE FROM t3_partition PARTITION (trans_time = '2014-04-30')
WHERE trans_id = 2;

-- 删除范围分区表中一个分区的部分内容
DELETE FROM t4_range_partition PARTITION before2014_04 WHERE trans_id != 2;
```

推荐阅读

- 插入数据
- 修改数据
- 插入并修改数据

8.3. 数据查询语句（DQL）

DQL 用于对数据库中的数据对象进行查询，其基本语法是由 SELECT FROM 实现简单查询， 同时根据查询条件选择关键字 WHERE（条件查询）、GROUP BY（分组查询）、HAVING（分组后条件查询）、LIMIT（限制查询数目）子句进行组合查询。

- 使用 ORDER BY|SORT BY|DISTRIBUTE BY|CLUSTER BY 对查询结果进行排序
- 使用 JOIN ON 进行多表关联查询。
- 使用 EXCEPT、INTERSECT、UNION 对多个查询结果进行集合计算。

当以上子句同时存在时，它们的执行顺序为： FROM > ON > JOIN > WHERE > GROUP BY > SELECT > HAVING > ORDER BY > LIMIT

除了常用数据查询语言形式，Inceptor 还支持层次化查询 CONNECT BY。

8.3.1. 表查询 SELECT FROM

SELECT 语句就像叠加在数据库表上的过滤器，利用 SQL 关键字从数据表中过滤出用户需要的数据。 根据需要添加 ALL|DISTINCT、WHERE、GROUP BY、HAVING、ORDER BY、LIMIT 进行条件查询。

功能描述

- SELECT 用于从表或视图中取出数据。
- WHERE 子句指定条件进行查询，必须紧跟在 FROM 子句之后。
- 当需要分组查询时需要使用 GROUP BY 子句，支持单列分组，多列分组，或子句中使用 Having 进行分组后条件查询。
- HAVING 一般与 GROUP BY 组合来使用，表示在得到分类汇总记录的基础之上，进一步筛选记录。
- LIMIT 用来限定查询结果的起始行，以及总行数。
- ALL 和 DISTINCT 告诉 Inceptor 是否要返回查询中出现的重复行。

注意事项

- HAVING 和 WHERE 都可以加条件，但 WHERE 在分组 GROUP BY 之前加条件，HAVING 在分组之后加条件.

且 WHERE 的效率要远远高于 HAVING

- 查询时如果不指定数据库，Inceptor 默认从 default 数据库中寻找用户指定的查询来源，如需指定数据库，方法如下：

  用 "." 来表示表和数据库的从属关系，比如 <database_name>.<table_name>

  在查询前使用 USE DATABASE 语句来指定当前数据库。

- 含逻辑运算符的过滤条件：我们可以使用 AND, OR 和 NOT 来指定过滤条件。
- 用 BETWEEN 来表示带有范围的过滤条件或范围区间。
- 有时候，过滤条件不是单个值，也不是一个区间中的值，而是一个有限集合。在这种情况下，我们可以用 OR 来连接这些过滤条件，也可以用 IN 来表示和过滤条件集合的从属关系。
- 在 WHERE 子句中不能有聚合函数，因为 Inceptor 在执行 GROUP BY 子句之前会执行 WHERE 子句。

语法格式

```
SELECT [ALL|DISTINCT] <select_statement> --[1]
FROM <TABLE_reference>
[WHERE <filter_statement>]
[GROUP BY <COLUMN[S]>]
[HAVING <filter_statement>]
[ORDER BY <COLUMN[S]> ASC|DESC] --[2]
[LIMIT [<m>, ]<n>];--[3]
```

参数说明

- <select_statement>：子查询语句结果。其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。

  FROM 子句指定的表中的列；

  Literals，比如数字或者字符串；

  表达式，如employee.salary+1000；

  函数（包括 Inceptor 自带函数和用户自定义函数）的调用。比如count(*)。

- <TABLE_reference>：指定查询的来源，它可以是表、视图、JOIN构造或者一个子查询。
- <filter_statement>：过滤条件。
- <COLUMN[S]>：指定列的名称，可以是多个列。
- <m>: 限定查询结果的起始行。
- <n>: 限定查询结果的总行数。
- 可选项[1]，DISTINCT 选项会不返回重复行，选择 ALL 则返回所有行（默认）。
- 可选项[2]，DESC 表示降序，ASC 表示升序（默认）。
- 可选项[3]，LIMIT 子句用于限制结果条数。Limit 接受一个或两个数字参数，如果只有一个参数，那么表示返回数据的最大行数；如果是两个参数，那么第一个参数表示返回数据的第一行的偏移量（第一行数据的偏移量是 0），第二个参数指定返回数据的最大条目数。

示例

例 78. 查询表信息。

- 创建表并插入数据

```sql
--创建员工表
CREATE TABLE emp(
id int comment '编号',
workno varchar(10) comment '工号',
name varchar(10) comment '姓名',
gender char(1) comment '性别',
age tinyint comment '年龄',
idcard char(18) comment '身份证号',
workaddress varchar(50) comment '工作地址',
entrydate date comment '入职时间'
)comment '员工表'
stored as holodesk;

-- 插入数据
INSERT INTO emp (id, workno, name, gender, age, idcard, workaddress, entrydate)
VALUES (1, '00001', 'lily', '女', 20, '123456789012345678', '北京', '2000-01-
01');
INSERT INTO emp (id, workno, name, gender, age, idcard, workaddress, entrydate)
VALUES (2, '00002', 'bob', '男', 18, '123456789012345670', '北京', '2005-09-
01');
INSERT INTO emp (id, workno, name, gender, age, idcard, workaddress, entrydate)
VALUES (3, '00003', 'rachard', '男', 38, '123456789712345670', '上海', '2005-08-
01');
INSERT INTO emp (id, workno, name, gender, age, idcard, workaddress, entrydate)
VALUES (4, '00004', 'rachel', '女', 18, '123456757123845670', '北京', '2009-12-01');
INSERT INTO emp (id, workno, name, gender, age, idcard, workaddress, entrydate)
VALUES (5, '00005', 'mary', '女', 32, null, '上海', '2012-06-01');
INSERT INTO emp (id, workno, name, gender, age, idcard, workaddress, entrydate)
VALUES (12, '00012', 'Angel', '男', 32, '123446757152345670', '北京', '2004-02-
01');
INSERT INTO emp (id, workno, name, gender, age, idcard, workaddress, entrydate)
VALUES (13, '00013', 'levy', '男', 88, '123656789012345678', '江苏', '2020-11-
01');
INSERT INTO emp (id, workno, name, gender, age, idcard, workaddress, entrydate)
VALUES (14, '00014', 'lisa', '女', 65, '123456719012345670', '西安', '2019-05-
01');
INSERT INTO emp (id, workno, name, gender, age, idcard, workaddress, entrydate)
VALUES (15, '00015', 'allen', '男', 70, '12345674971234567X', '西安', '2018-04-
01');
INSERT INTO emp (id, workno, name, gender, age, idcard, workaddress, entrydate)
VALUES (16, '00016', 'kate', '女', 18, null, '北京', '2012-06-01');
```

- 查询

```sql
-- 查询指定字段 name, workno, age并返回
select name,workno,age from emp;

-- 查询返回所有字段
select * from emp;

-- 查询所有员工的工作地址,并设置别名, as 可省略。
select workaddress as '工作地址' from emp;

-- 查看工作地址有哪些, 并去重
select distinct workaddress '工作地址' from emp;

-- 查询年龄小于 20 的员工信息
select * from emp where age < 20;

--查询年龄在15岁(包含) 到 50岁(包含)之间的员工信息
select * from emp where age between 15 and 50;

-- 查询没有身份证号的员工信息
select * from emp where idcard is null;

-- 查询性别为 女 且年龄小于 25岁的员工信息
select * from emp where gender = '女' and age < 25;

-- 查询年龄等于18 或 20 或 32 的员工信息
select * from emp where age = 18 or age = 20 or age =32;
select * from emp where age in(18,20,32);

-- 查询身份证号最后一位是X的员工信息
select * from emp where idcard like '%X';

-- 统计总条数
select count(*) from emp;

-- 统计该企业员工的平均年龄
select avg(age) from emp;

-- 根据性别分组 , 统计男性员工和女性员工的平均年龄
select gender, avg(age) from emp group by gender ;

--统计各个工作地址上班的男性及女性员工的数量
select workaddress, gender, count(*) '数量' from emp group by gender, workaddress;

--根据工作地址分组 , 获取员工数量大于等于2的工作地址
select workaddress, count(*) address_count from emp group by
workaddress having address_count >= 2;

--查询年龄小于45的员工 , 并根据工作地址分组 , 获取员工数量大于等于3的工作地址
select workaddress, count(*) address_count from emp where age < 45 group by
workaddress having address_count >= 3;

--根据入职时间, 对员工进行降序排序
select * from emp order by entrydate desc;

--根据年龄对公司的员工进行升序排序 , 年龄相同 , 再按照入职时间进行降序排序
select * from emp order by age asc , entrydate desc;

-- 查看第2条数据, 共展示5条数据
select * from emp limit 2,5;

--查询性别为男, 且年龄在20-80 岁(含)以内的前3个员工信息, 对查询的结果按年龄升序排序, 年龄相同按入职时间升序排序
select * from emp where gender = '男' and age between 20 and 80 order by age asc , entrydate
asc limit 3 ;
```

推荐阅读

- 表排序
- 表连接 JOIN ON
- 集合计算 EXCEPT|INTERSECT|UNION
- 层次化查询 CONNECT BY

8.3.2. 表排序

功能描述

指定数据的排序方式，以及升序和降序。

- **ORDER BY**：对查询结果进行 全排序（total ordering），所以所有数据都会经过一个单独的 reducer。如果数据很多， 只有一个 reducer 会导致计算花费大量时间。

  在数据非常多的情况下，且在“严格模式 （strict mode）”下 （hive.mapred.mode=strict，默认值是非严格)，ORDER BY 子句后必须跟着 LIMIT 子句。

- **SORT BY**：在每个reducer之内排序，来达到局部有序（local ordering）。 通过SORT BY 处理的数据在各自的 reducer 中有序，但是 reducer 中数据的范围可能互相会有重叠。
- **DISTRIBUTE BY**：可以为输出结果分桶,所有 DISTRIBUTE BY 列的列值相同的记录会被放进同一个 reducer 中。
- **CLUSTER BY** ：实现了在分桶之后进行升序排序。

注意事项

- ORDER BY 和 SORT BY 的语法几乎完全一样，但是当一个任务用到了不止一个 reducer，ORDER BY 和 SORT BY 的输出会不一样。 因为是局部有序，通过 SORT BY 处理的数据在各自的 reducer 中有序，但是 reducer 中数据的范围可能互相会有重叠。

语法格式

- ORDER BY

```
SELECT <select_statement> ORDER BY <col_name> [ASC|DESC] [,<col_name_2> [ASC|DESC],...];
```

- SORT BY

```
SELECT <select_statement> SORT BY <column_name1> [ASC|DESC] --[1]
[,<column_name2> [ASC|DESC],...];
```

- DISTRIBUTE BY

```
SELECT <select_statement> DISTRIBUTE BY <column_name1>;
```

- CLUSTER BY

```
SELECT <select_statement> CLUSTER BY <column_name1>;
```

参数说明

- 〈select_statement〉：子查询语句结果。其中的列需要和 〈table_name〉 的列数量一致且数据类型一一对应。
  a. FROM 子句指定的表中的列；
  b. Literals，比如数字或者字符串；
  c. 表达式，如 employee.salary+1000；

d. 函数（包括 Inceptor 自带函数和用户自定义函数）的调用。比如 count(*)。

- <column_name>：列名称
- 可选项[1]，DESC 表示降序，ASC 表示升序（默认）。
- 可选项[2]，可以使用多个字段实现组合排序。

示例

例 79. 表排序

```
select * from emp sort by workno desc;
select * from emp order by workno desc;
select * from emp DISTRIBUTE BY workno;
select * from emp CLUSTER BY workno;
```

创建示例表请查看：创建表 emp。

推荐阅读

- 表查询SELECT
- 表连接JOIN ON
- 集合计算EXCEPT|INTERSECT|UNION
- 层次化查询CONNECT BY

8.3.3. 表连接 JOIN ON

JOIN 用于把来自两个或多个表的行结合起来。所涉及的 SQL 语句， 根据 <join_condition> 连接条件，可分为不等价连接（non-equi join）、自然连接（使用 NATURAL JOIN，可以不需要明确写出 <join_condition>），以及等值连接如下：

- 内连接 JOIN|INNER|CROSS JOIN
- 外连接 LEFT|RIGHT|FULL OUTER JOIN
- 左半连接 LEFT SEMI JOIN

不支持 JOIN 条件中带有 OR 条件子句，也不支持 JOIN 条件中使用 BETWEEN/IN。

推荐阅读

- 表查询 SELECT
- 表连接 JOIN ON
- 集合计算 EXCEPT|INTERSECT|UNION
- 层次化查询 CONNECT BY

8.3.3.1. 自然连接 NATURAL JOIN

功能描述

- 使用 NATURAL JOIN，用户可以不需要明确写出 JOIN 条件子句。Inceptor 会自动在被连接的两张表中寻找名字相同的列。如果两张表 TABLE1 和 TABLE2 中存在同名列 col1，Inceptor 会自动生成 JOIN 条件子句 TABLE1.col1=TABLE2.col1。
- 如果 Inceptor 在被连接的两张表中找不到同名列，Inceptor 会将指令作为无条件的连接，也就是一个笛卡尔积连接。

语法格式

```
SELECT <table_name1>.<column_name1>, <table_name1>.<column_name2>, ...
<table_name2>.<column_name1>, <table_name2>.<column_name3>, ...
FROM <table_name1> NATURAL JOIN <table_name2>;
```

参数说明

- <table_name>：表名称。
- <column_name>：列名称。

示例

例 80. NATURAL JOIN

- 创建表并插入数据

```
-- 创建员工绩效表 perf
CREATE TABLE perf(
id int comment '编号',
workno varchar(10) comment '工号',
standard char(10) comment '标准',
achievement char(10) comment '是否达标',
grade char(10) comment '评级',
bonus decimal comment '奖金'
)comment '员工绩效表'
stored as holodesk;

-- 插入数据
INSERT INTO perf (id, workno, standard, achievement, grade, bonus)
VALUES (1, '00001', 'B', 'Y', 'B', 1000.00);
INSERT INTO perf (id, workno, standard, achievement, grade, bonus)
VALUES (2, '00002', 'B', 'N', 'C', 0.00);
INSERT INTO perf (id, workno, standard, achievement, grade, bonus)
VALUES (3, '00003', 'A', 'Y', 'A', 5000.39);
INSERT INTO perf (id, workno, standard, achievement, grade, bonus)
VALUES (4, '00004', 'B', 'Y', 'B', 1000.00);
INSERT INTO perf (id, workno, standard, achievement, grade, bonus)
VALUES (5, '00005', 'A', 'Y', 'AA', 10349.04);
INSERT INTO perf (id, workno, standard, achievement, grade, bonus)
VALUES (6, '00012', 'A', 'N', 'B', 0.00);
INSERT INTO perf (id, workno, standard, achievement, grade, bonus)
VALUES (7, '00013', 'AA', 'N', 'A', 0.00);
INSERT INTO perf (id, workno, standard, achievement, grade, bonus)
VALUES (8, '00014', 'AA', 'Y', 'AA', 21022.55);
INSERT INTO perf (id, workno, standard, achievement, grade, bonus)
VALUES (9, '00017', 'B', 'Y', 'A', 2034.55);
```

- 自然连接

```
select perf.id, perf.bonus, emp.name, emp.gender from perf natural join emp;
```

8.3.3.2. 内连接 [INNER | CROSS] JOIN

功能描述---

- 使用 JOIN|INNER JOIN 对两个连接表格进行查询，连接结果只返回匹配连接条件的行。
- 当连接条件恒成立时，两表之间的内连接称为笛卡尔积连接 CROSS JOIN（又被称为“交叉连接”）。笛卡尔积连接会把左表的每一条记录和右表的所有记录相连接，如果左表的记录数为 m，右表的记录数为 n，则结果集中会产生 m * n 条记录。

语法格式

- 内连接 JOIN|INNER JOIN

```
SELECT <select_statement1>, <select_statement2>, ... FROM <table_name1>
JOIN|INNER JOIN <table_name2>
[ON (<join_condition>)];
[JOIN|INNER JOIN <table_name3>
ON (<join_condition>)...];
```

- 不含 JOIN ON 的隐式用法

```
SELECT  <table_name1>.<column_name1>,  <table_name2>.<column_name3>, ...
FROM  <table_name1>,  <table_name2>
WHERE  <table_name1>.<column_name1> =  <table_name2>.<column_name3>;----
```

- 笛卡尔积连接 CROSS JOIN

```
SELECT <table_name1>.<column_name1>, <table_name2>.<column_name2>, ...
FROM <table_name1>
CROSS JOIN <table_name2> [where (<join_condition>)];--[1]
```

参数说明

- 〈table_name〉：表名称。
- 〈column_name〉：列名称。
- 〈join_condition〉：连接条件。
- 可选项[1]，笛卡尔积可以选择不添加 where 子句，此时返回的记录数为左表〈table_name1〉和右表〈table_name2〉记录数的乘积。

示例

例 81. 内连接

```
--[INNER] JOIN
select emp.id, emp.name, emp.workaddress, perf.achievement from emp INNER JOIN perf on emp
.workno = perf.workno;

-- 隐式用法
select emp.id, emp.name, emp.workaddress, perf.achievement from emp, perf where emp.workno =
perf.workno;

-- 笛卡尔积连接
select emp.id, emp.name, emp.workaddress, perf.achievement from emp cross JOIN perf where emp
.workno = perf.workno;
```

- 创建示例表请查看：创建表 perf、创建表 emp
- 若 cross join 不添加 where 子句，表 emp 拥有 10 条数据，perf 拥有 9 条数据，则经过笛卡尔积连接之后会返回 9 * 10 条数据。

8.3.3.3. 外连接 LEFT|RIGHT|FULL OUTER JOIN

功能描述

外连接分为 左外连接 LEFT OUTER JOIN、右外连接 RIGHT OUTER JOIN、全外连接 FULL OUTER JOIN

- 左外连接 LEFT OUTER JOIN：

  也可以缩写为 LEFT JOIN。左向外联接的结果集包括 LEFT OUTER 子句中指定的左表的所有行，而不仅仅是联接列所匹配的行。如果左表的某行在右表中没有匹配行，则在相关联的结果集行中右表的所有选择列表列均为空值。

- 右外连接 RIGHT OUTER JOIN：

  也可以缩写为 RIGHT JOIN。右向外联接的结果集包括 RIGHT OUTER 子句中指定的右表的所有行，而不仅仅是联接列所匹配的行。如果右表的某行在左表中没有匹配行，则在相关联的结果集行中左表的所有选择列表列均为空值。

- 全外连接 FULL OUTER JOIN：

  返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值。

语法格式

```
SELECT <select_statement>, <select_statement>, ... FROM <table_name1>
[LEFT|RIGHT|FULL] OUTER JOIN <table_name2>
[ON <join_condition>];
[[LEFT|RIGHT|FULL] OUTER JOIN <table_name3> ON <join_condition>...] --[1]
```

参数说明

- <table_name>：表名称。
- <join_condition>：连接条件。
- <select_statement>：子查询语句结果。其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。
  a. FROM 子句指定的表中的列；
  b. Literals，比如数字或者字符串；
  c. 表达式，如 employee.salary+1000；
  d. 函数（包括 Inceptor 自带函数和用户自定义函数）的调用。比如 count(*)。
- 可选项[1]，支持多表连接。

示例

例 82. 外连接

```
--左外连接
select perf.id, emp.name, emp.workaddress, perf.achievement, perf.bonus, perf.grade from emp
LEFT outer join perf on emp.workno = perf.workno;

--右外连接
select perf.id, emp.name, emp.workaddress, perf.achievement, perf.bonus, perf.grade from emp
right outer join perf on emp.workno = perf.workno;

--全外连接
select perf.id, emp.name, emp.workaddress, perf.achievement, perf.bonus, perf.grade from emp
full outer join perf on emp.workno = perf.workno;
```

创建示例表请查看：创建表 perf、创建表 emp

8.3.3.4. 左半连接 LEFT SEMI JOIN

功能描述

- 左半连接只返回左表中的记录。
- 左半连接用来查看左表中符合和 JOIN 条件的记录。
- 左半连接可以通过 LEFT SEMI JOIN, WHERE… IN 和 WHERE EXISTS 中嵌套子查询来实现。

语法格式

```
SELECT <select_statement>, <select_statement>, ... FROM <table_name1> LEFT SEMI JOIN <table_name2>
ON <join_condition>;
```

参数说明

- <table_name>：表名称。
- <join_condition>：连接条件。
- <select_statement>：子查询语句结果。其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。
  a. FROM 子句指定的表中的列；
  b. Literals，比如数字或者字符串；
  c. 表达式，如 employee.salary+1000；
  d. 函数（包括 Inceptor 自带函数和用户自定义函数）的调用。比如 count(*)。

示例

例 83. 左半连接

```
select perf.id, emp.name, emp.workaddress, perf.achievement, perf.bonus, perf.grade from emp
LEFT semi join perf on emp.workno = perf.workno;
```

创建示例表请查看：创建表 perf、创建表 emp。

8.3.3.5. 不等价连接

功能描述

- 要执行不等价连接，ON 子句中的连接条件必须是等价条件，不等价条件体现在 WHERE 子句中的过滤条件中。

注意事项

- 不等价连接和笛卡尔积相像，很容易返回大量结果（在两表行数乘积的级别），ON 子句中要求有等价条件可以限制结果的数量。如果确定不需要限制结果数量，可以在 ON 子句中的等价条件里放一个永远成立的等式，比如 1=1。执行这样的操作必须格外小心。

语法格式

```
SELECT <select_statement>, <select_statement>, ... FROM <table_name1> JOIN <table_name2> ON
<equi_join_condition> WHERE <non_equi_join_condition>
```

参数说明

- <table_name>：表名称。
- <equi_join_condition>：等值连接条件子句。
- <non_equi_join_condition>：不等值连接条件子句。
- <select_statement>：子查询语句结果。其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。
  a. FROM 子句指定的表中的列；
  b. Literals，比如数字或者字符串；
  c. 表达式，如 employee.salary+1000；
  d. 函数（包括 Inceptor 自带函数和用户自定义函数）的调用。比如 count(*)。

示例

例 84. 不等价连接

```
SELECT perf.id, emp.name, emp.workaddress, perf.achievement, perf.bonus, perf.grade FROM emp
JOIN perf on 1=1 WHERE perf.id != emp.id;
```

创建示例表请查看：创建表 perf、创建表 emp。

8.3.4. 集合计算

本节主要介绍，使用 EXCEPT、INTERSECT 和 UNION 分别对多个查询语句进行去重取差、取交集或取并集。

推荐阅读

- 表查询 SELECT
- 表排序
- 表连接 JOIN ON

- 层次化查询 CONNECT BY

8.3.4.1. 差集连接 EXCEPT

功能描述

返回查询结果的差集。

- A EXCEPT B 将 A 中所有和 B 重合的记录除去，然后返回去重后的 A 中剩下的记录。
- A EXCEPT ALL B 将 A 中所有和 B 重合的记录除去，然后返回不去重的 A 中剩下的记录。

语法格式

```
SELECT <COLUMN[S]> FROM
<select_statement>
EXCEPT [ALL] --[1]
SELECT <COLUMN[S]> FROM
<select_statement>;
```

参数说明

- <COLUMN[S]>：指定列的名称，可以是多个列。
- <select_statement>：子查询语句结果。其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。
  a. FROM 子句指定的表中的列；
  b. Literals，比如数字或者字符串；
  c. 表达式，如 employee.salary+1000；
  d. 函数（包括 Inceptor 自带函数和用户自定义函数）的调用。比如 count(*)。
- 可选项[1]，可以选择是否添加 [ALL] 关键字对查询结果去重。添加 ALL 则返回不去重的记录，不添加则返回去重的记录。

示例

例 85. 差集连接

- 建表并插入数据

```
--建表
CREATE TABLE perf_confirm(
id int comment '编号',
workno varchar(10) comment '工号',
standard char(10) comment '标准',
achievement char(10) comment '是否达标',
grade char(10) comment '评级',
bonus decimal comment '奖金'
)comment '员工绩效表确认'
stored as holodesk;

--插入数据
INSERT INTO perf_confirm (id, workno, standard, achievement, grade, bonus)
VALUES (1, '00001', 'B', 'Y', 'B', 1000.00);
INSERT INTO perf_confirm (id, workno, standard, achievement, grade, bonus)
VALUES (2, '00002', 'B', 'N', 'C', 0.00);
INSERT INTO perf_confirm (id, workno, standard, achievement, grade, bonus)
VALUES (3, '00003', 'A', 'Y', 'A', 5000.39);
INSERT INTO perf_confirm (id, workno, standard, achievement, grade, bonus)
VALUES (5, '00005', 'A', 'Y', 'AA', 15349.04);
INSERT INTO perf_confirm (id, workno, standard, achievement, grade, bonus)
VALUES (6, '00012', 'A', 'N', 'B', 0.00);
INSERT INTO perf_confirm (id, workno, standard, achievement, grade, bonus)
VALUES (7, '00013', 'AA', 'N', 'A', 0.00);
INSERT INTO perf_confirm (id, workno, standard, achievement, grade, bonus)
VALUES (8, '00014', 'AA', 'Y', 'AA', 21022.55);
INSERT INTO perf_confirm (id, workno, standard, achievement, grade, bonus)
VALUES (9, '00015', 'AA', 'Y', 'AA+', 30044.22);
INSERT INTO perf_confirm (id, workno, standard, achievement, grade, bonus)
VALUES (10, '00016', 'B', 'Y', 'AA', 11233.33);
INSERT INTO perf_confirm (id, workno, standard, achievement, grade, bonus)
VALUES (10, '00017', 'AAA', 'Y', 'AAA', 40387.53);
```

- 差集连接

```
-- 查询表 perf_confirm 中有，但 perf 中没有的评级。去重
select grade from perf_confirm EXCEPT select grade from perf;

--不去重
select grade from perf_confirm EXCEPT all select grade from perf;
```

8.3.4.2. 交集连接 INTERSECT

功能描述

- 返回查询结果的交集。
- A INTERSECT B 将 A 中所有和 B 重合的记录返回，然后返回去重后的 A 中剩下的记录。
- A INTERSECT ALL B 将 A 中所有和 B 重合的记录返回，然后返回不去重的 A 中剩下的记录。

语法格式

```
SELECT <COLUMN[S]> FROM
<select_statement>
INTERSECT [ALL] --[1]
SELECT <COLUMN[S]> FROM
<select_statement>;
```

参数说明

- <COLUMN[S]>：指定列的名称，可以是多个列。

- <select_statement>：子查询语句结果。其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。
  a. FROM 子句指定的表中的列；
  b. Literals，比如数字或者字符串；
  c. 表达式，如 employee.salary+1000；
  d. 函数（包括 Inceptor 自带函数和用户自定义函数）的调用。比如 count(*)。
- 可选项[1]，可以选择是否添加 [ALL] 关键字对查询结果去重。添加 ALL 则返回不去重的记录，不添加则返回去重的记录。

示例

例 86. INTERSECT

```
-- 查询表 perf_confirm 和表 perf 中都有的评级，去重
select grade from perf_confirm INTERSECT select grade from perf;

-- 不去重
select grade from perf_confirm INTERSECT all select grade from perf;
```

创建示例表请查看：创建表 perf、创建表 perf_confirm

8.3.4.3. 并集连接 UNION

功能描述

- 返回查询结果的并集。
- A UNION B 将 A 和 B 中的所有记录返回，然后返回去重后的记录。
- A UNION ALL B 将 A 和 B 中的所有记录返回，然后返回不去重的记录，返回条数为表 A 和表 B 条数之和。

语法格式

```
SELECT <COLUMN[S]> FROM
<select_statement>
UNION [ALL] --[1]
SELECT <COLUMN[S]> FROM
<select_statement>;
```

参数说明

- <COLUMN[S]>：指定列的名称，可以是多个列。
- <select_statement>：子查询语句结果。其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。
  a. FROM 子句指定的表中的列；
  b. Literals，比如数字或者字符串；
  c. 表达式，如 employee.salary+1000；
  d. 函数（包括 Inceptor 自带函数和用户自定义函数）的调用。比如 count(*)。
- 可选项[1]，可以选择是否添加 [ALL] 关键字对查询结果去重。添加 ALL 则返回不去重的记录，不添加则返回去重的记录。

示例

例 87. UNION

```
--查询表perf_confirm和表perf中涉及到的所有评级，去重
select grade from perf_confirm UNION select grade from perf;

-- 不去重，返回条数为两表条数之和
select grade from perf_confirm UNION all select grade from perf;
```

创建示例表请查看：创建表 perf、创建表 perf_confirm

8.3.5. 层次化查询 CONNECT BY

功能描述

- 层次化查询是树型结构的查询，是 SQL 语法中常用的功能之一，通常由根节点、父节点、子节点、以及叶节点组成。
- 层次化查询的结果为树状结构，根节点和层级关系分别由 START WITH 和 CONNECT BY 决定，且父节点在上子节点在下。
- 支持将层次化查询作为 WHERE IN 非关联条件子句。

注意事项

- Inceptor目前暂时不支持如下场景的层次化查询：

  尚不支持的操作有：
  SYS_CONNECT_BY_PATH、CONNECT_BY_ISLEAF、CONNECT_BY_ROOT、NOCYCLE、ORDER SIBLINGS BY。

  不允许把子查询作为 START WITH 的条件。

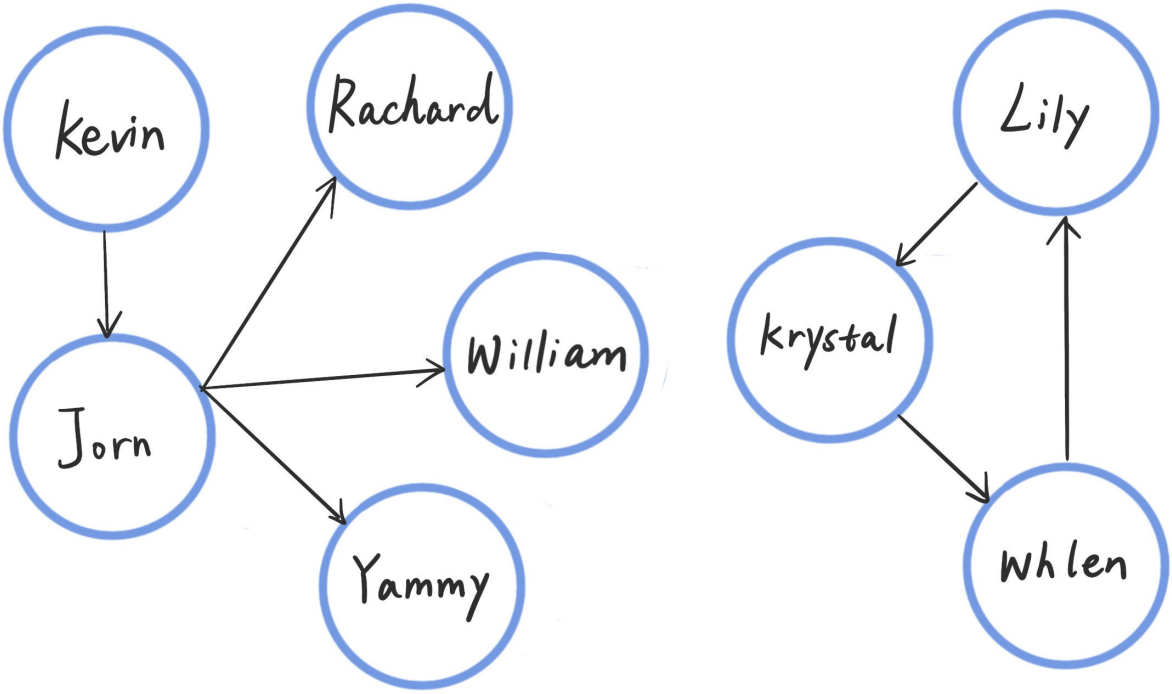
  层次化查询作为 WHERE 子查询时只能通过 WHERE IN 和父查询连接，不允许用比较运算符连接，且只能为非关联子查询。

  不允许在 WHERE IN 非关联子查询中访问 hq__level。
- 如果返回结果中一个子节点引用了父节点，则被称为出现了循环（loop）。Inceptor 会在层次化查询出现循环时抛出错误，示例如下。

建表并插入数据

```
CREATE TABLE loop_employee ( employee_id int, name string, manager_id int) stored as
holodesk;
insert INTO loop_employee values(1,'Kevin',null),(2,'Jorn',1),(3,'Lily',8),(4,'Rachard',2)
,(5,'William',2),(6,'Krystal',3),(7,'Yammy',2),(8,'Whlen',6);
+--------------+----------+--------------+
| employee_id  |   name   | manager_id   |
+--------------+----------+--------------+
| 1            | Kevin    | NULL         |
| 2            | Jorn     | 1            |
| 3            | Lily     | 8            |
| 4            | Rachard  | 2            |
| 5            | William  | 2            |
| 6            | Krystal  | 3            |
| 7            | Yammy    | 2            |
| 8            | Whlen    | 6            |
+--------------+----------+--------------+
```

对应的层级关系为：



查询 Lily 及其下属的相关信息，由于下级员工 Whlen 为 Lily 的上级，构成了循环出现报错

```
SELECT employee_id, name, manager_id
FROM loop_employee
START WITH employee_id = 3
CONNECT BY PRIOR employee_id = manager_id;

-- 返回报错结果
EXECUTION FAILED: Task MAPRED-SPARK error SparkException: [Error 1] Job aborted due to
stage failure: Task 0 in stage 897.0 failed 4 times, most recent failure: Lost task 0.3 in
stage 897.0 (TID 1880, 192.168.111.3, jobId 933, sqlId 1352, sessionId 1582): java.lang
.RuntimeException: Loop founded in hierarchical query (state=08S01,code=1)
```

语法格式

```
SELECT [<hq__level>], <column_name1>, <column_name2>, ...
<expression1>, <expression2>, ...
FROM <table_name>
[WHERE <where_clause>]
START WITH <start_condition>
CONNECT BY PRIOR <prior_condition>;
```

参数说明

- <hq__level>：用于表示节点的层级，为伪列。

  "hq__level" 中间的符号为两个下划线。

- <table_name>：表名称。
- <where_clause>：条件子句。
- <start_condition>：开始条件。
- <prior_condition>： PRIOR 条件子句。
- START WITH：指定根记录条件，满足 START WITH 给定条件的记录为根记录。
- CONNECT BY：用于指定父记录和子记录之间的关联关系。
- PRIOR：在 CONNECT BY 中指定父记录行。若 CONNECT BY 条件为组合条件，只有一个条件需要指明 PRIOR。允许存在多个 PRIOR 条件子句。
- 可选项[1]，支持层次查询时使用 WHERE 子句进行条件过滤。

示例

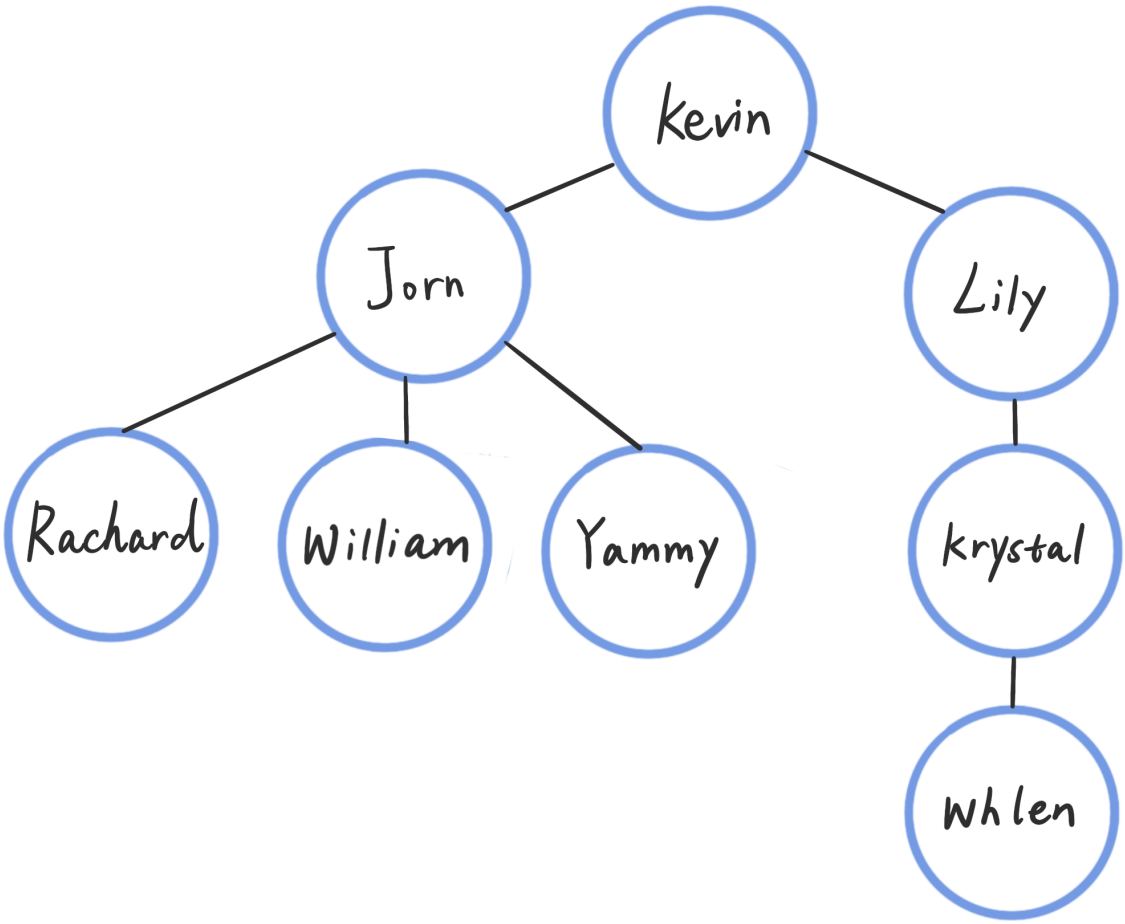
- 建表并插入数据

```
CREATE TABLE no_loop_employee ( employee_id int, name string, manager_id int, salary int,
department_id int) stored as holodesk;

BATCHINSERT INTO no_loop_employee batchvalues (values(1,'Kevin',null,5000,1),values(2,'Jorn',1
,4500,2),values(3,'Lily',1,4700,3),values(4,'Rachard',2,4000,2),values(5,'William',2,4000,2),va
lues(6,'Krystal',3,3800,3),values(7,'Yammy',2,4000,2),values(8,'Whlen',6,3700,3));
```

employee_id	name	manager_id	salary	department_id
1	Kevin	NULL	5000	1
2	Jorn	1	4500	2
3	Lily	1	4700	3
4	Rachard	2	4000	2
5	William	2	4000	2
6	Krystal	3	3800	3
7	Yammy	2	4000	2
8	Whlen	6	3700	3

- 对应的层级关系为：



- 执行层次化查询：

例 88. 层次化查询

```
-- 查询 Kevin 及其下级中员工的 employee_id、name 及其 manager_id，并去除 Lily 相关员工：
SELECT employee_id, name, manager_id FROM no_loop_employee
START WITH name = 'Kevin'
CONNECT BY name != 'Lily' AND PRIOR employee_id = manager_id
ORDER BY employee_id;
+--------------+----------+-------------+
| employee_id  |   name   | manager_id  |
+--------------+----------+-------------+
| 1            | Kevin    | NULL        |
| 2            | Jorn     | 1           |
| 4            | Rachard  | 2           |
| 5            | William  | 2           |
| 7            | Yammy    | 2           |
+--------------+----------+-------------+


--按照 “employee_id = manager_id” 做层次化查询，返回 employee_id、name、
manager_id，以及每条查找结果的层级。
--- 当语句未指定根记录时，会返回以所有记录为根节点的全部树：
SELECT employee_id, name, manager_id, hq__level
FROM no_loop_employee
CONNECT BY PRIOR employee_id = manager_id
ORDER BY employee_id, hq__level;
+--------------+----------+-------------+------------+
| employee_id  |   name   | manager_id  | hq__level  |
+--------------+----------+-------------+------------+
| 1            | Kevin    | NULL        | 1          |
| 2            | Jorn     | 1           | 1          |
| 2            | Jorn     | 1           | 2          |
| 3            | Lily     | 1           | 1          |
| 3            | Lily     | 1           | 2          |
| 4            | Rachard  | 2           | 1          |
| 4            | Rachard  | 2           | 2          |
| 4            | Rachard  | 2           | 3          |
| 5            | William  | 2           | 1          |
| 5            | William  | 2           | 2          |
| 5            | William  | 2           | 3          |
| 6            | Krystal  | 3           | 1          |
| 6            | Krystal  | 3           | 2          |
| 6            | Krystal  | 3           | 3          |
| 7            | Yammy    | 2           | 1          |
| 7            | Yammy    | 2           | 2          |
| 7            | Yammy    | 2           | 3          |
| 8            | Whlen    | 6           | 1          |
| 8            | Whlen    | 6           | 2          |
| 8            | Whlen    | 6           | 3          |
| 8            | Whlen    | 6           | 4          |
+--------------+----------+-------------+------------+


--支持将层次化查询作为 WHERE IN 非关联条件子句。
---查询 id=6 的员工及其下级员工的员工 ID 和名字
SELECT a.name, a.employee_id FROM no_loop_employee a
WHERE a.name IN (
SELECT b.name FROM no_loop_employee b
START WITH b.employee_id = 6
CONNECT BY PRIOR b.employee_id=b.manager_id
)ORDER BY a.employee_id;
+----------+--------------+
|   name   | employee_id  |
+----------+--------------+
| Krystal  | 6            |
| Whlen    | 8            |
+----------+--------------+
```

推荐阅读

- 表查询SELECT
- 表排序
- 表连接JOIN ON
- 集合计算EXCEPT|INTERSECT|UNION

8.4. 冷热数据分离

在海量大数据场景下，通过冷热数据分离能够有效平衡存储成本和查询性能需求。例如使用 HDD 介质，能够降低查询频度低的数据（称为冷数据）的存储成本；使用SSD介质，能够提高查询频度高的数据（称为热数据）的查询效率。

我们支持表级别的冷热数据分离。在创建表时，用户可以手动指定表的存储介质。表创建成功后，表的使用方式和普通表没有任何差异。

冷热数据分离功能包括 指定表的存储介质 和 设置存储介质的属性。

8.4.1. 指定表的存储介质

在创建表时，通过设置 storage_policy 指定表的存储介质。

- storage_policy='HOT'，表示将表数据存储在属性为 "kGold" 的存储介质
- storage_policy='COLD'，表示将表数据存储在属性为 "kBronze" 的存储介质

创建热表

```
Create table table_hot(
id int,
name string)
stored as holodesk
storage_policy='HOT';
```

创建冷表

```
Create table table_cold(
id int,
name string)
stored as holodesk
storage_policy='COLD';

--若不指定storage_policy，则默认为冷表，表示将表数据存储在属性为"kBronze"的存储介质
```

- 暂不支持冷热数据自动迁移

8.4.2. 设置存储介质的属性

第一步：通过 Manager 查看TDDMS 服务的配置项 tabletserver.store.datadirs，确认当前数据库系统的存储资源。

tabletserver.store.datadirs 用于配置不同节点的数据存储路径，通常是一块磁盘对应一个存储路径。

表 4. 例如某 3 节点集群的配置情况如下

主机	值	解释
host1	/mnt/disk1/shiva-tabletserver-tddms1/data,/mnt/disk2/shiva-tabletserver-tddms1/data,/mnt/disk3/shiva-tabletserver-tddms1/data,/mnt/disk4/shiva-tabletserver-tddms1/data	host1共有4块存储介质，分别为disk1、disk2、disk3、disk4
host2	/mnt/disk1/shiva-tabletserver-tddms1/data,/mnt/disk2/shiva-tabletserver-tddms1/data,/mnt/disk3/shiva-tabletserver-tddms1/data,/mnt/disk4/shiva-tabletserver-tddms1/data	host2共有4块存储介质，分别为disk1、disk2、disk3、disk4

主机	值	解释
host3	/mnt/disk1/shiva-tabletserver-tddms1/data,/mnt/disk2/shiva-tabletserver-tddms1/data,/mnt/disk3/shiva-tabletserver-tddms1/data,/mnt/disk4/shiva-tabletserver-tddms1/data	host3共有4块存储介质，分别为disk1、disk2、disk3、disk4

第二步：通过 Manager 配置 TDDMS 服务的配置项 tabletserver.store.quality_types 的值，指定存储介质的属性

- 存储介质属性为 kGold，则用于存储热表数据
- 存储介质属性为 kBronze，则用于存储冷表数据

表 5. 例如某 3 节点集群的配置情况如下（HDD、SSD分别使用 kBronze、kGold 属性区分）

主机	值	解释
host1	kGold,kGold,kBronze,kBronze	host1 中，disk1 用于存储热表数据，disk2 用于存储热表数据，disk3 用于存储冷表数据，disk4 用于存储冷表数据
host2	kBronze,kBronze,kGold,kGold	host2 中，disk1 用于存储冷表数据，disk2 用于存储冷表数据，disk3 用于存储热表数据，disk4用于存储热表数据
host3	kGold,kGold,kBronze,kBronze	host3 中，disk1 用于存储热表数据，disk2 用于存储热表数据，disk3 用于存储冷表数据，disk4用于存储冷表数据

- 参数值默认为空时，不区分存储介质的属性，此时数据会写入所有的存储介质。
- tabletserver.store.quality_types与tabletserver.store.datadirs的值需要一一对应。例如磁盘扩缩容时，需要更新配置
- 不同属性的存储介质需要满足三副本原则
- 参数值注意区分大小写

> 除了冷热数据分离存储（2级），我们还支持冷温热数据分离存储（3级）。storage_policy='WARM'，表示将表数据存储在属性为"kSilver"的存储介质。

8.5. Compact/小文件合并

小批量、频繁的数据写入和更新会产生小文件。如果数据库系统中存在大量的小文件，可能会影响查询性能。为了避免小文件问题导致查询性能下降，我们提供高效的小文件合并机制。

本节主要包括手动合并小文件 、 自动合并小文件 、管理Compact任务

8.5.1. 手动合并小文件

在 DBA Service 可以查看当前的小文件数量。如果发现小文件数量过多（DBA上称为"块/block"），可以通过手动触发命令，立即执行小文件合并任务。

```
ALTER TABLE <table_name> compact "full"|"major"|"minor";
```

- full compact，表示将 base 文件合并成一个 base 文件,并且删除无用的 delta 文件
- major compact，表示将 base 及其 delta 文件合并,生成一个新的 base 文件

- minor compact，表示将 base 的 delta 文件合并,生成一个新的 delta 文件

> Inceptor Task 数与数据文件数量（base文件）有关，因此 full compact 能够通过减少数据文件数量来减少 Task 数。

8.5.2. 自动合并小文件

用户可以根据业务特点，自定义小文件自动合并策略，平衡小文件数量和合并开销。

holodesk.compaction.trigger.enabled 用于控制是否通过Compact Service 服务执行 Compact 任务。若设置为 false，则由计算引擎服务执行 Compact 任务。

引入版本：2.1	作用域：session、global
默认值：true	范围：true、false

- 作用：Compact Service 是 Inceptor 上的小文件合并专用服务。设置该参数TRUE，能够通过 Compact Service 进行小文件合并任务，小文件合并效果更好。
- 影响：设置该参数False，则通过计算引擎服务执行 Compact 任务，可能占用计算资源。
- 限制：如果 Compact Service 服务服务/异常，那么该参数失效，此时通过计算引擎服务执行 Compact 任务。

holodesk.fullcompaction.filesize.threshold 用于控制一个 Base 文件是否属于小文件，大于阈值的 Base 文件不是小文件，不会进行 Full Compact

引入版本：2.1	作用域：global（通过 Manager 对 Compact Service 设置参数）
默认值：134217728（128MB）	范围：单位 Byte

- 影响：阈值越大，越容易触发 Full Compact

holodesk.fullcompaction.filenum.threshold 用于控制full compact触发阈值，当小文件数量（block）超过阈值时合并

- 如果是非分区分桶表，那么相同 tablet 上的小文件超过阈值时合并
- 如果是分区表，那么相同分区，相同 tablet 的小文件超过阈值时合并
- 如果是分桶表，那么相同bucket，相同 tablet 的小文件超过阈值时合并
- 如果是分区分桶表，那么相同分区、相同 bucket、相同 tablet 的小文件超过阈值时合并

引入版本：2.1	作用域：global（通过 Manager 对 Compact Service 设置参数）
默认值：3	范围：[1,+∞)

- 作用：full compaction 通过合并 base 文件来减少小文件数量
- 影响：阈值越小，越容易触发 Full Compact

holodesk.majorcompaction.delta.filesize.ratio 用于控制 Major Compact 触发阈值，当 delta size/ base size 超过阈值时合并

引入版本：2，1	作用域：global（通过 Manager对Compact Service 设置参数）
默认值：0.2	范围：(0,1)

- 作用：Major Compact 通过合并 Base 文件以及 Delta 文件来减少小文件数量
- 影响：阈值越小，越容易触发 Major Compact

holodesk.minorcompaction.delta.num.threshold 用于控制 Minor Compact 触发阈值，当一个 Base 文件的 Delta 文件数量超过阈值时合并。

引入版本：2.1	作用域：global（通过 Manager对Compact Service 设置参数）
默认值：3	范围：[1,+∞)

- 作用：Minor Compact 通过合并Base文件的多个 Delta 文件来减少小文件数量
- 影响：阈值越小，越容易触发 Minor Compact

8.5.3. 管理Compact任务

通过 DBA Service 查看 Compact Service 服务可以管理 Compact 任务

Compact触发检查

```
Checking for default.t1_ee1aabdc-f0e8-4536-b76d-8408a6163fee
--Compact 触发检查，定期检查表是否满足小文件触发条件。一旦满足，则会自动创建小文件合并任务。
```

Compact Task

```
isCompact: true
shivaTableName:default.t1_ee1aabdc-f0e8-4536-b76d-8408a6163fee
Fast compaction for default.t1_ee1aabdc-f0e8-4536-b76d-8408a6163fee
--Compact Task，用于合并小文件
```

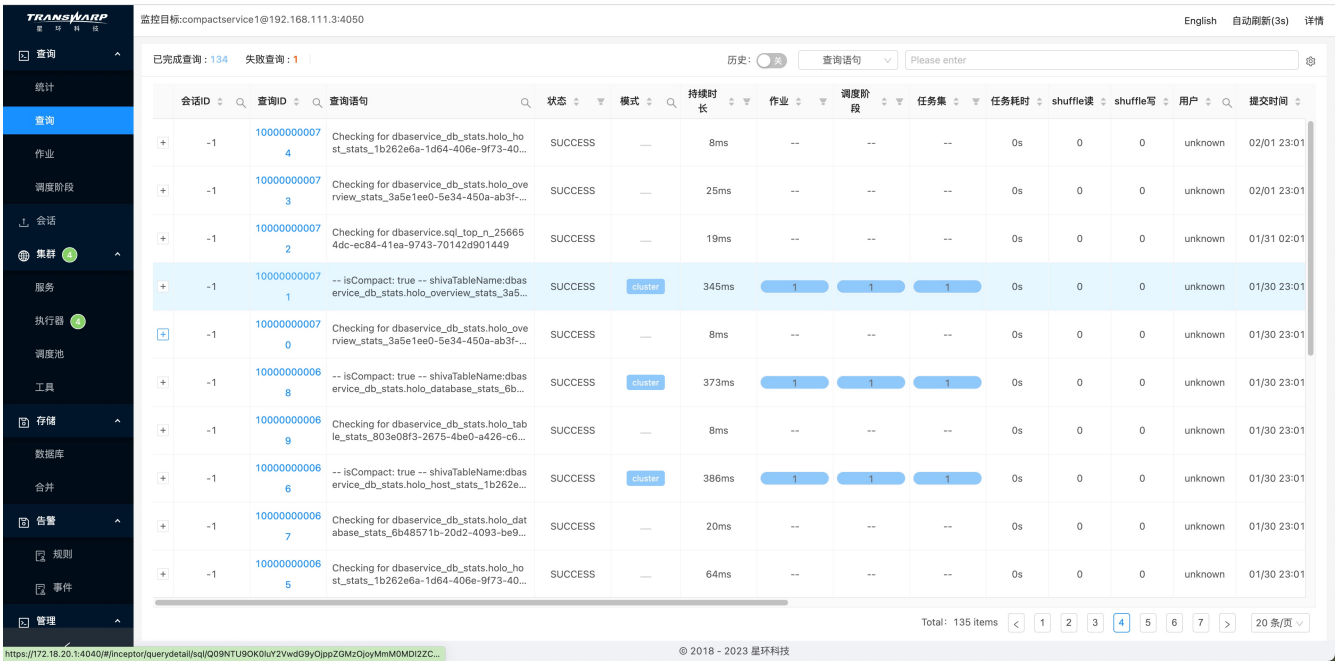


图 52. Compact 任务

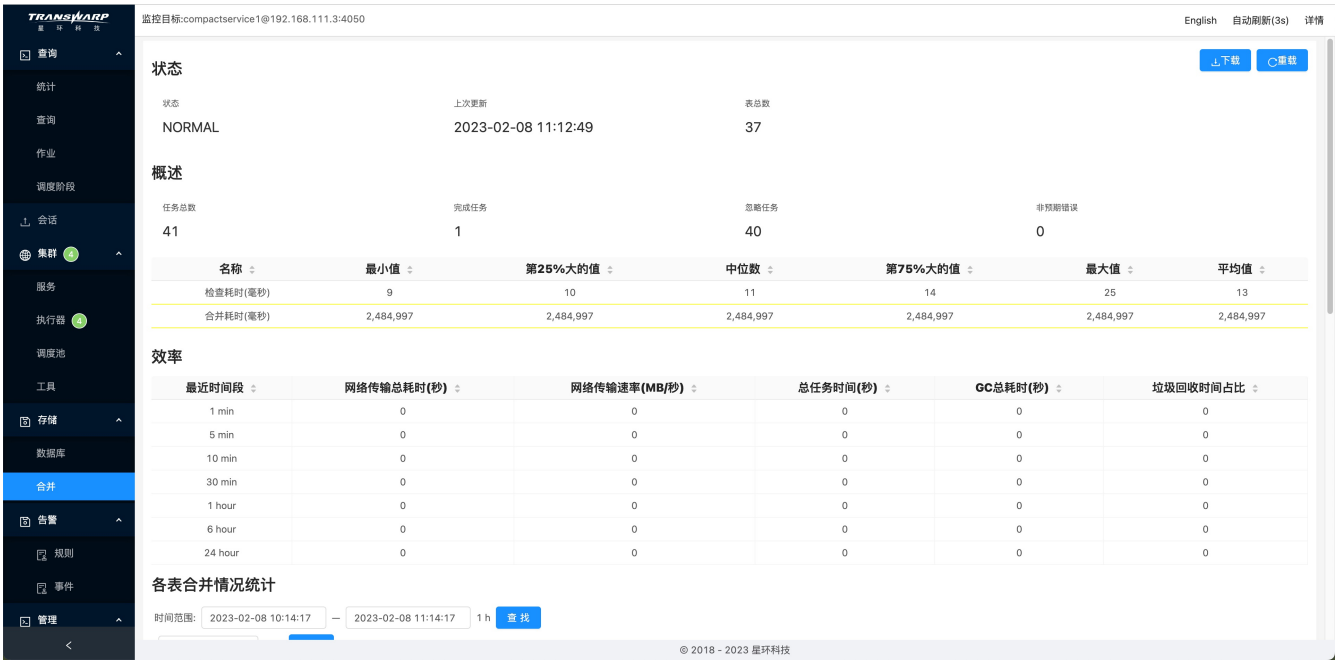


图 53. Compact 报表

8.6. 事务控制

我们提供自动提交、手动提交两种事务提交方式。如无指定，SQL 会以自动提交的方式提交事物。

本节主要包括 、设置事务级别 、。

8.6.1. 手动提交事务

第一步：设置参数 transaction.type=holodesk

transaction.type 用于手动指定事务表的事务管理器。

引入版本：1.3	作用域：session、global
默认值：none	范围：none、holodesk、inceptor

- 作用：设置参数为 none 时，Inceptor 会根据事务表的类型（HOLODESK）自动选择对应的事务管理器保证 ACID。
- 限制：begin transaction手动开启一个事务时，需要先设置transaction.type手动指定事务管理器。

第二步：begin transaction

提交事务

```
set transaction.type=holodesk;
begin transaction;
Batchinsert into acid_test batchvalues(
values(1),
values(1)
);    ----此时事务并未提交
commit;    ----此时事务提交，数据真正写入
```

回滚事务

```
set transaction.type=holodesk;
begin transaction;
Batchinsert into acid_test batchvalues(
values(2),
values(2)
);    ----此时事务并未提交
rollback;    ----此时事务回滚，数据并未写入
```

8.6.2. 设置事务级别

我们提供完整的事务支持能力，用户可以通过 restful 的方式设置数据库的事务级别。

- SerializableSnapshot，基于快照的可串行化，即 SSI
- Serializable，基于悲观锁的可串行化
- RepeatableRead，可以重复读
- ReadCommitted，读提交
- ReadUncommitted，读未提交

查看当前的事务级别

```
curl -X GET "<host>:<port>/bulk_transaction_isolation?[pretty]"
```

设置事务级别

```
curl -X PUT "<host>:<port>/bulk_transaction_isolation?level=<isolation_level>&[timeout_s=<timeout>]"
```

8.6.3. 管理事务

Inceptor 支持当前锁情况的查询，以及自动分析锁等待链，提供诊断视图的能力。

下面我们提供一个功能演示：

第一步：通过两个客户端分别登陆 user1 和 user2 两个用户

第二步：在 user1 会话界面，执行下述 SQL 语句：

```
create table acid_test(a int) stored as holodesk;

set transaction.type=holodesk;
begin transaction;
Batchinsert into acid_test batchvalues(
values(1),
values(2)
);  --不提交事务
```

第三步：检查加锁情况

登录DBA Service 界面，点击【存储-事务】，检查加锁情况

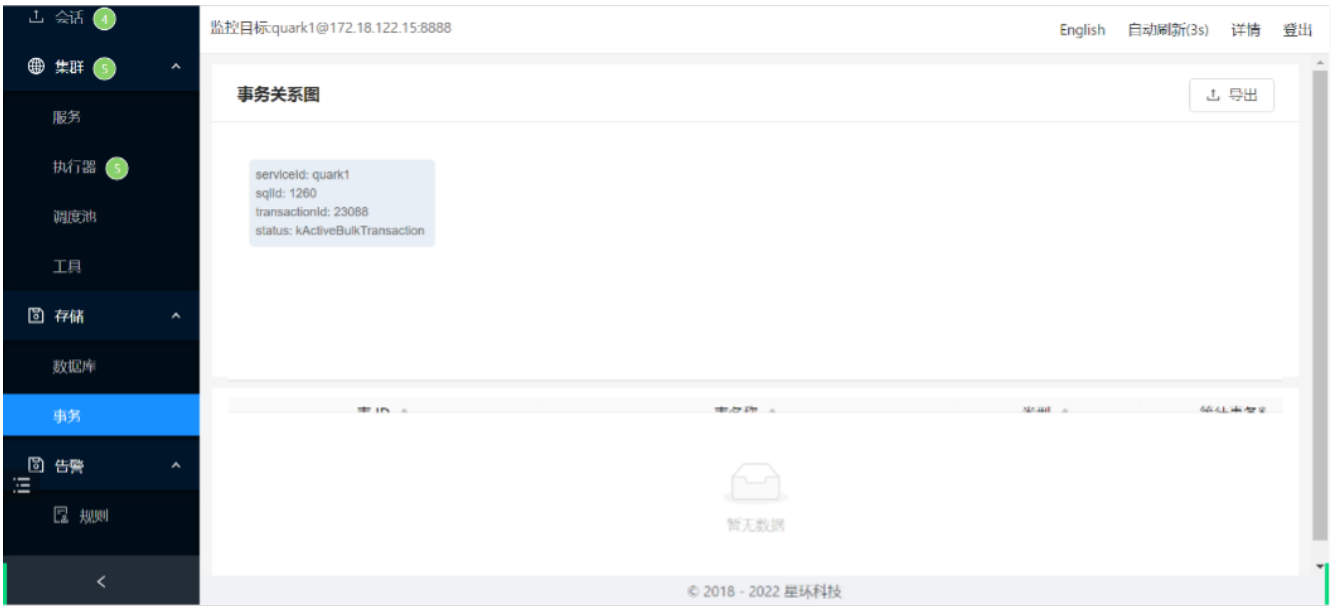


图 54. 可以看到 user1 加锁成功

第四步： 在user2 会话界面，执行下述 SQL 语句：

```
update acid_test set a = a+1 where a = 2;   --user2等锁中，等待user1事务提交才能执行成功
```

第五步：检查锁等待链情况 登录 DBA Service 界面，点击存储-事务，查看锁等待情况

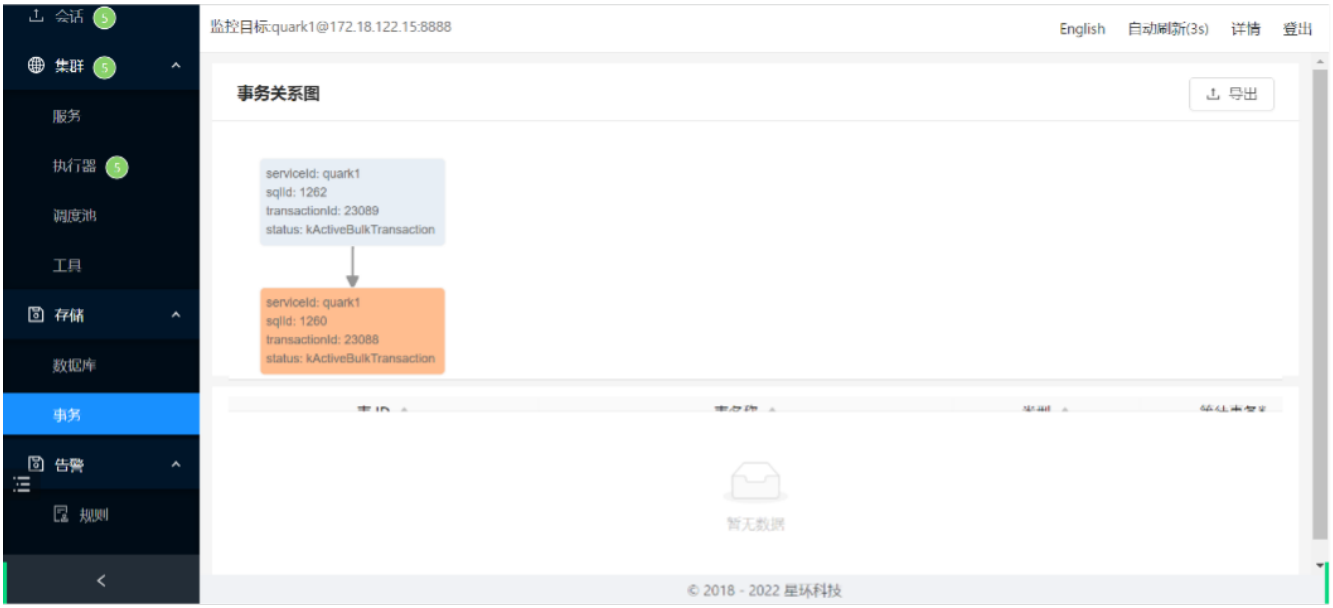


图 55. 可以看到 user2 等待 user1 释放锁资源

第六步：在 user1 会话界面，提交事务

```
commit;  --user1事务提交成功，释放锁资源。此时user2拿锁并执行成功
```

此时在 DBA Service 上可以看到，user1、user2 依次提交成功并释放锁资源。

8.7. ROWKEY

Inceptor 中 rowkey 是关系数据库系统中用于确保一条记录的唯一标识和正确引用的一系列列（或字段）。因此被设置为 rowkey 的字段（或一系列列）是唯一的、且不允许为空。 本节主要为您介绍 Inceptor 中 rowkey 的相关概念、使用及运用 rowkey实现UPSERT 功能等内容。

8.7.1. ROWKEY 介绍

在 Inceptor 中，用户在创建表时，可以选择手动定义 rowkey（user defined rowkey）来创建 rowkey 表。也可以不定义，此时系统会在内部自动生成 internal rowkey 作为每条数据的唯一标识。

- Inceptor 中的 rowkey 在功能上与传统主键类似，都是一个非空且唯一的字段。但在行为上与传统的 primary key 有所差别，这可以从当第二次插入相同的主键，Inceptor 不会出现报错而是覆盖原数据可以得证。
- 内部 internal rowkey 的生成是分布式的，能够保证所有数据的 internal rowkey 都不相同，所以即便在大批量写入的场景下也不会产生热点问题。

因此在 Inceptor 中设置了 rowkey，可以非常方便地支持实时数仓场景下的多种诉求：

- 支持高性能的 UPSERT 或 DELETE 功能。

  Inceptor 除了具有传统数据库高性能 Append Only 类型的写入特性外，还可以通过rowkey实现高性能的整行写入更新、部分列写入更新。在进行写入更新时只需要根据rowkey更新，不需要全表扫描，从而达到高性能的UPSERT，同时也能保证数据的唯一性。

- 支持基于 rowkey 的快速查询

  如果为表设置了 rowkey，在基于 rowkey 的查询场景中，就能快速根据rowkey定位到整行数据，提升查

询性能。 适用于实时风控、实时推荐等在线应用场景。

8.7.2. 使用 Rowkey

使用限制

- 被设置为 rowkey 的字段是唯一且非空的列或者列组合，同时只能在一个语句里设置多列为表的 rowkey。
- Inceptor 中定义rowkey 必须基于创建分桶表时设定 TBLPROPERTIES 实现。

  rowkey 表的 rowkey 列必须与分桶列完全一致。

- 不支持修改 rowkey，如需修改 rowkey 请重新建表。

语法格式

创建分桶表定义 rowkey

```
CREATE [[GLOBAL ]TEMPORARY] TABLE  --[1]
[IF NOT EXISTS] [<database_name>.]<table_name>( --[2]
<column_name1> <DATATYPE1> [NOT NULL] [COMMENT "<text>"], --[3]
<column_name2> <DATATYPE2> [NOT NULL] [COMMENT "<text>"], --[4]
<column_name3> <DATATYPE3> [NOT NULL] [COMMENT "<text>"],
...)
CLUSTERED BY (<bucket_column1>, <bucket_column1>, ...) [INTO <num_buckets> BUCKETS]--[5]
STORED AS HOLODESK
[WITH TABLET NUMBER <tablet_number> CAPACITY NUMBER <capacity_number> REPLICATION
<replication_number>]--[6]
[WITH TABLESIZE <table_size>KB|MB|GB|TB|PB|EB|ZB|BB]--[7]
TBLPROPERTIES("holodesk.rowkey" ="<rowkey_column1>, <rowkey_column1>, ..."[, "<property_name>"=
"<property_value>", ...])--[8]
```

参数说明

- <rowkey_column1>：holodesk.rowkey 对分桶表设定 rowkey，且 rowkey 列必须和分桶列 <bucket_column1> 完全一致。
- <database_name>：将表建在指定的数据库中。
- <table_name>：表名称。
- <column_name>：列名称。
- <DATATYPE>：数据类型。
- <text>：表注释。
- <num_buckets>：分桶数。
- <bucket_column>：分桶列。
- <table_size>：表大小。
- <tablet_number>：大表 |FACT TABLE 的 TABLET 数量为磁盘数；小表 |DIMENSION TABLE 的 TABLET 数量为节点数；创建表时，默认为大表。
- <capacity_number>：存储空间数量。
- <replication_number>：表的副本数，推荐为奇数。
- <property_name>：表属性名称。
- <property_value>：表属性值。
- 可选项[1]，如果 <table_name> 指定的表已经存在，会报错，加上 IF NOT EXISTS 选项在建表前检测是否已存在同名表。

- 可选项[2]，指明表所在的数据库名称。
- 可选项[3]，支持非空 [NOT NULL] 约束。
- 可选项[4]，用 [COMMENT] 为表加注释，注意注释要放在引号中。
- 可选项[5]，可以使用 INTO <num_buckets> BUCKETS 指定分桶数.也可以选择不指定，则默认分桶数为比磁盘数 *5 大的质数。
- 可选项[6]，支持对表的磁盘数 tablet_number、存储空间数量 capacity_number 和副本数 REPLICATION 进行设定。

> 磁盘数 tablet_number、存储空间数量 capacity_number 和副本数 REPLICATION 三者必须同时进行设定。

- 可选项[6]，支持对表的大小 table_size 进行设定，若您需要同时设定表大小以及磁盘数量，必须先设定磁盘数，再设定表大小。
- 可选项[7]，支持使用 TBLPROPERTIES 或 SERDEPROPERTIES 对表的属性进行设定。
- 可选项[8]，支持对 holodesk 表设定 rowkey 的同时设定其他参数属性，属性之间用英文逗号（,）分隔开。

示例

例 89. 创建一张单值分桶表并指定 rowkey

```
CREATE TABLE if not exists srk_bucket_product(
id  INT,
prod_NO INT,
prod_name string,
prod_time date,
price double)
CLUSTERED BY (id)
STORED AS HOLODESK
tblproperties("holodesk.rowkey"="id");
```

例 90. 创建一张多值分桶表并指定 rowkey

```
CREATE TABLE if not exists mrk_bucket_product(
id  INT,
prod_NO INT,
prod_name string,
prod_time date,
price double)
CLUSTERED BY (id,prod_NO,price)
STORED AS HOLODESK
tblproperties("holodesk.rowkey"="id,prod_NO,price");
```

8.7.3. ROWKEY 的 UPSERT 功能

功能描述

- Inceptor 使用 rowkey 实现 UPSERT 功能，且尚不支持 UPSERT 语法。
- UPSERT 是 UPDATE 和 INSERT 的组合。在关系数据库的上下文中，如果表中已经存在指定的rowkey值，则更新现有行，如果指定的 rowkey 值不存在，则插入新行。

操作示例

- Example3 展示了运用 rowkey 实现单值分桶表的 UPSERT 功能
- Example4 展示了运用 rowkey 实现多值分桶表的 UPSERT 功能

创建数据源表并插入数据

```
-- 其中rk_resource1为先插入的数据，rk_resource2为后插入的数据，根据 rowkey 值分别进行插入和更新操作

CREATE TABLE if not exists rk_resource1(
id  INT,
prod_NO INT,
prod_name string,
prod_time date,
price double)
STORED AS HOLODESK;
INSERT INTO rk_resource1 VALUES
(1, 11, '薯片', '2023-2-10', 10.00),
(2, 22, '火腿', '2023-2-11', 12.11),
(3, 33, '卫生纸', '2023-1-1', 15.00),
(4, 44, '牛肉', '2023-2-15', 49.98);

CREATE TABLE if not exists rk_resource2(
id  INT,
prod_NO INT,
prod_name string,
prod_time date,
price double)
STORED AS HOLODESK;
INSERT INTO rk_resource2 VALUES
(1, 12, '薯片|update', '2023-2-10', 10.00),
(2, 22, '火腿|update', '2023-2-11', 11.99),
(3, 33, '卫生纸|update', '2023-2-15', 15.00),
(5, 55, '青菜|insert', '2023-2-15', 3.55);
```

例 91. 数据写入单值 rowkey 分桶表

- 首先向表 srk_bucket_product 插入数据 rk_resource1

```
insert into srk_bucket_product select * from rk_resource1;

-- 查询表 srk_bucket_product 结果如下
+-----+----------+------------+-------------+--------+
| id  | prod_no  | prod_name  |  prod_time  | price  |
+-----+----------+------------+-------------+--------+
| 1   | 11       | 薯片        | 2023-02-10  | 10.0   |
| 2   | 22       | 火腿        | 2023-02-11  | 12.11  |
| 3   | 33       | 卫生纸       | 2023-01-01  | 15.0   |
| 4   | 44       | 牛肉        | 2023-02-15  | 49.98  |
+-----+----------+------------+-------------+--------+
```

- 再向表 srk_bucket_product 插入数据 rk_resource2

```
insert into srk_bucket_product select * from rk_resource2;

-- 再次查询表 srk_bucket_product 结果，可以看出
---- 1. 更新rowkey列id值为1，2，3的数据；
---- 2. 保留rowkey列id值为4的数据不变；
---- 3. 插入rowkey列id值为5的数据；
+-----+----------+-------------+-------------+--------+
| id  | prod_no  |  prod_name  |  prod_time  | price  |
+-----+----------+-------------+-------------+--------+
| 1   | 12       | 薯片|update   | 2023-02-10  | 10.0   |
| 2   | 22       | 火腿|update   | 2023-02-11  | 11.99  |
| 3   | 33       | 卫生纸|update  | 2023-02-15  | 15.0   |
| 4   | 44       | 牛肉          | 2023-02-15  | 49.98  |
| 5   | 55       | 青菜|insert   | 2023-02-15  | 3.55   |
+-----+----------+-------------+-------------+--------+
```

例 92. 数据写入复合 rowkey 分桶表

- 首先向表 mrk_bucket_product 插入数据 rk_resource1

```
insert into mrk_bucket_product select * from rk_resource1;

-- 查询表 mrk_bucket_product 结果如下
+-----+----------+------------+-------------+--------+
| id  | prod_no  | prod_name  |  prod_time  | price  |
+-----+----------+------------+-------------+--------+
| 4   | 44       | 牛肉        | 2023-02-15  | 49.98  |
| 1   | 11       | 薯片        | 2023-02-10  | 10.0   |
| 3   | 33       | 卫生纸       | 2023-01-01  | 15.0   |
| 2   | 22       | 火腿        | 2023-02-11  | 12.11  |
+-----+----------+------------+-------------+--------+
```

- 再向表 mrk_bucket_product 插入数据 rk_resource2

```
insert into mrk_bucket_product select * from rk_resource2;

-- 再次查询表 srk_bucket_product 结果，可以看出
---- 1. 更新rowkey列(id,prod_no,date)值为(3,33,15.00)的数据;
---- 2. 保留rowkey列(id,prod_no,date)值为(4,44,49.98)的数据不变;
---- 3. 插入rowkey列(id,prod_no,date)值不完全相同的两条数据，以及完全不同的一条新增数据。
+-----+----------+-------------+-------------+--------+
| id  | prod_no  |  prod_name  |  prod_time  | price  |
+-----+----------+-------------+-------------+--------+
| 1   | 12       | 薯片|update   | 2023-02-10  | 10.0   |
| 1   | 11       | 薯片          | 2023-02-10  | 10.0   |
| 2   | 22       | 火腿|update   | 2023-02-11  | 11.99  |
| 2   | 22       | 火腿          | 2023-02-11  | 12.11  |
| 3   | 33       | 卫生纸|update  | 2023-02-15  | 15.0   |
| 4   | 44       | 牛肉          | 2023-02-15  | 49.98  |
| 5   | 55       | 青菜|insert   | 2023-02-15  | 3.55   |
+-----+----------+-------------+-------------+--------+
```

8.8. 动态分区

动态分区功能开启后，您可以按需为新数据动态地创建分区。 同时在动态分区处理数据时，INSERT INTO 或 INSERT OVERWRITE 语句中不直接指定分区值，只指定分区列名（分区字段）。分区列的值在 SELECT 子句中提供，系统自动根据分区列的值将数据插入到相应分区。 本节主要为您介绍 Inceptor 中动态分区的创建、查看以及动态插入数据等内容。 以及动态分区功能的使用案例。

8.8.1. 动态创建分区

Inceptor 目前支持依据分区键以及对应插入数据，动态创建分区的功能，目前需要修改以下参数来实现：

- 插入数据时，分区列自动查找分区键并写入对应分区。

  set hive.exec.dynamic.partition=true;

- 插入数据时，分区列对应分区键不存在，自动创建分区。

  set argodb.dynamic.create.partition.enabled=true;

> 自动创建分区功能只能依据分区列的实际值添加对应的单值分区。
>
> 该参数只适用于 Holodesk 表。

- 另外可以设置单个 SQL 允许创建的最大分区数量。

set hive.exec.max.dynamic.partitions=1000;（默认为1000）

以及单个 task 允许创建的最大分区数量，set hive.exec.max.dynamic.partitions.pernode=1000;（默认为1000）

8.8.2. 查看分区情况

功能描述

- 使用 show partitions 语法可以查看指定分区表中的分区
- 通过分区过滤来查询对应分区数据

语法格式

```
-- 查看分区情况
show partitions <partition_table_name>;

-- 查看分区数据
select <column_name1>, <column_name2>,... from <table_name> where <partition_key> = <value>
```

8.8.3. 动态插入数据

功能描述

- 使用 INSERT INTO TABLE 直接将数据写入分区表，且不需要手动指定分区。而是通过数据中分区列的值自动判断写入至对应的分区。
- 在动态分区处理数据时，INSERT INTO 或 INSERT OVERWRITE 语句中不直接指定分区值，只指定分区列名（分区字段）。分区列的值在 SELECT 子句中提供，系统自动根据分区列的值将数据插入到相应分区。
- 默认状况下，动态分区插入功能是关闭的，要使用动态分区插入需要修改对应参数。
  a. 插入数据时，分区列自动查找分区键并写入对应分区：

     SET hive.exec.dynamic.partition=true;（默认false）

     set stargate.dynamic.partition.enabled=true;

  b. 非分区、非分桶列中如果有隐式转换（源数据列和目标表列不匹配）， 可以通过打开如下开关实现：

     SET ngmr.optimization.insert.single.file.skip.select=true; （默认true）

注意事项

- 范围分区动态插入仅支持单个分区键，插入新增列后列顺序为 普通列-范围分区列-新增列。
- 在向动态分区表中插入数据时，由于并没有指定分区列的具体值（通过SQL语句中分区列的数据自行判断）。 所以需要查询分区列插入，并且分区列应放置在最后且该值对应的分区键已存在。

语法格式

```
INSERT INTO TABLE <table_name>
SELECT <select_statement> FROM <source>;
```

参数说明

- <table_name>：目标表表名称。
- <select_statement>：子查询语句结果。其中的列需要和 <table_name> 的列数量一致且数据类型一一对应。
- <source>：数据源表名称。

8.8.4. 使用案例

```
--创建源表,并插入数据
CREATE TABLE source (trans_id INT,acc_num INT,trans_time DATE,trans_type STRING,stock_id STRING
,price decimal,amount INT)
STORED AS HOLODESK;
BATCHINSERT INTO source BATCHVALUES (VALUES (7, 1, '2014-04-30', 'discount', 'NO20140201', 10.00,10
),VALUES(7, 2, '2014-2-11', 'online', 'NO20230228', 12.11,10),VALUES (7, 3, '2014-04-30', '
wholesale', 'NO20191001', 8.99,1000),VALUES(7, 4, '2014-2-15', 'sales', 'NO20230228', 12.11,80));

-- 创建分区表
CREATE TABLE dynamic_partition_table(trans_id INT, acc_num INT, trans_type STRING, stock_id STRING,
price DECIMAL, amount INT)
PARTITIONED BY(trans_time DATE)
STORED AS HOLODESK;


-- 动态创建分区，修改参数如下：
set argodb.dynamic.create.partition.enabled=true;


-- 动态插入数据至分区表
--- 修改参数，开启数据动态插入功能
set hive.exec.dynamic.partition=true;
set stargate.dynamic.partition.enabled=true;
--- 插入数据
INSERT INTO TABLE dynamic_partition_table
SELECT source.trans_id, source.acc_num, source.trans_type, source.stock_id, source.price, source
.amount, source.trans_time FROM source;


--查看分区
show partitions dynamic_partition_table;
+------------------------+
|       partition        |
+------------------------+
| trans_time=2014-02-11  |
| trans_time=2014-02-15  |
| trans_time=2014-04-30  |
+------------------------+

--查看分区数据
select trans_id,stock_id,price,amount,trans_time from dynamic_partition_table;
+-----------+-------------+--------+---------+-------------+
| trans_id  |  stock_id   | price  | amount  | trans_time  |
+-----------+-------------+--------+---------+-------------+
| 7         | NO20230228  | 12.11  | 10      | 2014-02-11  |
| 7         | NO20230228  | 12.11  | 80      | 2014-02-15  |
| 7         | NO20140201  | 10.00  | 10      | 2014-04-30  |
| 7         | NO20191001  | 8.99   | 1000    | 2014-04-30  |
+-----------+-------------+--------+---------+-------------+
```

8.9. 函数与运算符

8.9.1. Inceptor函数和运算符入门

Inceptor 支持大量的函数和运算符。以下是所有 Inceptor 支持的函数和运算符。

- 关系运算符

```
<, <=, <=>, <>, =, ==, >, >=, isnull, isnotnull, like, rlike, regexp
```

- 算术运算符

+, -, *, /, %, &, |, ^, ~

- 逻辑运算符

AND, OR, |, NOT, !, IN, NOT IN, [NOT] EXISTS

- 数学函数

round, floor, ceil, rand, exp, ln, log10, log2, log, pow, power, sqrt, bin, hex, unhex,conv, num, intto_base, intfrom_base, abs, pmond, sin, asin, cos, acos, positive, negative, sign, pi, degrees, radians, tan, atan, e,
cbrt，factorial，shiftleft，shiftright，shiftrightunsigned，regr_slope，regr_intercept，regr_r2，regr_sxx，regr_syy，regr_sxy，regr_avgx，regr_avgy，regr_count

- 类型转换函数

binary、cast

- 日期函数

add_months, extract, from_unixtime, to_unix_timestamp, unix_timestamp, to_timestamp, to_date, tdh_todate, day, dayofmonth, dayofyear, quarter, hour, minute, second, weekofyear, datediff, date_add, date_sub, date_format, str_to_date, sysdate

- 条件函数

If, coalesce, CASE a WHEN b THEN c [WHEN d THEN e]* [ELSE f] END, CASE WHEN a THEN b [WHEN c THEN d]* [ELSE e] END

- 字符串函数

ascii, base64, concat, context_ngrams, concat_ws, concat_ws, encode, find_in_set, format_number, get_json_object, in_file, instr, length, locate, lower, lcase, lpad, ltrim, ngrams, parse_url, printf, regexp_extract, regexp_replace, repeat, reverse, rpad, rtrim, sentences, space, split, str_to_map, substr, substring, translate, trim, unbase64, upper, ucase, chr, oracle_instr, to_number, trunk, to_char, to_semiangle, nvl, decode, greatest, lnnvl, nvl2, range_decode, characters，substring_index，levenshtein，soundex，sha1，sha2, aes_encrypt，aes_decrypt，char_length，octet_length，crc32，uuid

- 复杂类型函数

map,struct,named_struct,array,sort_array, size, sort_array_by, extract_union

- 表生成函数

explode, inline, json_tuple, parse_url_tuple, stack, replicate_rows

- 窗口函数

sum, max, min, count, avg, first_value, last_value, lag, lead, row_number, rank, dense_rank, ntile, cume_dist, percent_rank

- **context函数**

current_user, has_role, current_database, current_time, current_date, current_timestamp, logged_in_user

8.9.2. Inceptor函数和运算符指南中的表

在《Inceptor 函数和运算符指南》中，将会用几张虚构的表作为数据用于演示。以下是这些表的内容：

- 用户信息表 user_info：

name	acc_num	password	citizen_id	bank_acc	reg_date	acc_level
马**	6513065	115591	14**7	960*41	20110101	A
祝**	6670192	205239	23**8	737*71	20100101	C
华*	5224133	531547	42**1	326*07	20080214	B
魏**	3912384	841242	52**4	685*48	20091202	A
宁**	4580952	986634	42**7	977*76	20081031	D
邱*	0700735	737297	34**8	143*18	20121024	A
李*	8725869	600709	46**3	430*84	20130702	E
潘**	6600641	990590	51**5	484*08	20110430	C
李**	2755506	015859	31**9	424*37	20110916	D
管**	2394923	783438	33**4	999*74	20141003	C

- 交易信息表 transactions：

trans_id	acc_num	trans_time	trans_type	stock_id	price	amount
943197522	6513065	20140105100520	b	AA7105670	12.13	200
929634984	3912384	20140205140521	b	UA1467891	11.11	300
499506900	6513065	20140506133109	s	CA2789982	6.12	100
209441379	3912384	20140430111523	s	CX5397790	4.50	1000
648230055	0700735	20140315111111	s	DT7966575	22.66	200
719753265	3912384	20140328102400	b	BY8490909	68.43	100
975639131	0700735	20140611102830	s	AT6934136	5.30	200
991691937	2755506	20140702113025	s	VR2575735	7.52	1300

trans_id	acc_num	trans_time	trans_type	stock_id	price	amount
289018112	6513065	20140916105811	b	UT7592045	9.81	500
162742112	2394923	20141031135018	s	UC1610649	12.21	500
597565609	3912384	20140214141519	s	IU1775004	4.16	600
459590958	0700735	20140430143020	b	XJ9717497	5.25	1000
594819547	5224133	20140801110003	b	GL2547626	6.36	800
895916502	6513065	20141225133500	s	KC9102028	7.49	1100
900192386	6670192	20141130113905	s	XC1915304	8.64	900
952639648	6670192	20140314145958	s	CP7629713	10.31	400
404905188	6513065	20140628133001	b	SH6277444	7.02	100
952110653	6600641	20140228140005	s	GH6828501	9.16	100
817414815	5224133	20140331115900	s	ZX5373511	10.03	800
213859826	6513065	20140508094805	b	CL2121979	18.38	700

- 学生信息表 student_info：

stu_name	course_id
赵一	SCU100
钱二	MUS101
孙三	NULL
李四	COM101
周五	BIO101

- 课程信息表 course_info：

course_id	course_name
BIO101	生物基础
COM101	面向对象编程
MUS101	西方古典音乐赏析
SCU100	雕塑

course_id	course_name
MAT100	微积分

- 存储 Json 数据的表 a:

20160612	{"eventid":"1","eventname"="rainstorm"}
20160712	{"eventid":"2","eventname"="sunny"}

8.9.3. 关系运算符

- A < B

操作符	返回类型	描述
A < B	Boolean	判断 A 是否小于 B

说明

A 和 B 可以为任何基本类型，如果 A 小于 B，则返回 TRUE，否则返回 FALSE。如果 A 或 B 值为“NULL”，结果返回“NULL”。

举例

```
SELECT 1 < 2 AS result FROM system.dual LIMIT 1;
```

result
true

- A > B

操作符	返回类型	描述
A > B	Boolean	判断 A 是否大于 B

说明

A 和 B 可以为任何基本类型，如果 A 大于B ，则返回 TRUE，否则返回 FALSE。如果 A 或 B 值为“NULL”，结果返回“NULL”。

举例

```
SELECT 1 > 2 AS result FROM system.dual LIMIT 1;
```

result
false

- A = B 或 A == B

操作符	返回类型	描述
A = B 或 A == B	Boolean	判断 A 是否等于 B

说明

A 和 B 可以为任何基本类型，如果 A 与 B 相等，则返回 TRUE，否则返回 FALSE。

举例

```
SELECT 1 = 1 AS result FROM system.dual LIMIT 1;
```

result
true

```
SELECT 1 == 1 AS result FROM system.dual LIMIT 1;
```

result
true

- A >= B

操作符	返回类型	描述
A >= B	Boolean	判断 A 是否大于或者等于 B

说明

A 和 B 可以为任何基本类型，如果 A 大于或者等于 B，则返回 TRUE，否则返回 FALSE。如果 A 或 B 值为“NULL”，结果返回“NULL”。

举例

```
SELECT 1 >= 2 AS result FROM system.dual LIMIT 1;
```

result
false

- A <= B

操作符	返回类型	描述
A <= B	Boolean	判断 A 是否小于或者等于 B

说明

A 和 B 可以为任何基本类型，如果 A 小于或者等于 B，则返回 TRUE，否则返回 FALSE。如果 A 或 B 值为“NULL”，结果返回“NULL”。

举例

```
SELECT 1 <= 2 AS result FROM system.dual LIMIT 1;
```

result
true

- A <> B

操作符	返回类型	描述
A <> B	Boolean	判断 A 是否不等于 B

说明

A 和 B 可以为任何基本类型，如果 A 不等于 B 返回 TRUE，否则返回 FALSE。如果A或B值为“NULL”，结果返回“NULL”。

举例

```
SELECT 1 <> 2 AS result FROM system.dual LIMIT 1;
```

result
true

- A <=> B

操作符	返回类型	描述
A <=> B	Boolean	判断 A 是否等于 B

说明

A 和 B 可以为任何基本类型，当 A 和 B 都不为 null 的时候，用法和 "=" 一致，但是当 A 和 B 都为 NULL 的时候返回 true，而 A 和 B 只有一个为 NULL 的时候返回 false。

举例

```
SELECT NULL <=> NULL AS result FROM system.dual LIMIT 1;
```

result
true

- A IS NULL

操作符	返回类型	描述
A IS NULL	Boolean	判断 A 是否为空

说明

A 可以为任何基本类型，如果 A 为空，则返回 TRUE，否则返回 FALSE。

举例

```
SELECT 1 IS NULL AS result FROM system.dual LIMIT 1;
```

result
false

- A like B

操作符	返回类型	描述
A like B	Boolean	判断 A 是否满足 sql 中的模式 B

说明

如果 A 或 B 值为“NULL”，结果返回“NULL”。字符串 A 与 B 通过 sql 进行匹配，如果相符返回 TRUE，不符返回 FALSE。B 字符串中的“_”代表任一字符，“%”则代表多个任意字符。例如：（‘foobar’ like ‘foo’）返回 FALSE，（‘foobar’ like ‘foo___’或者 ‘foobar’ like ‘foo%’）则返回 TURE

举例

```
SELECT 'foobar' like 'foo___' AS result FROM system.dual LIMIT 1;
```

result
true

- A rlike B 或 A regexp B

操作符	返回类型	描述
A rlike B 或 A regexp B	Boolean	判断 A 是否符合正则表达式 B

说明

如果 A 或 B 值为“NULL”，结果返回“NULL”。字符串 A 与 B 通过 java 进行匹配，如果相符返回 TRUE，不符返回 FALSE。例如：（‘foobar’ rlike ‘foo’）返回 FALSE，(’ foobar’rlike ‘^f.*r$’）返回 TRUE。

举例

```
SELECT 'foobar' rlike 'foo*' AS result FROM system.dual LIMIT 1;
```

result
true

```
SELECT 'foobar' regexp 'foo*' AS result FROM system.dual LIMIT 1;
```

result
true

8.9.4. 算术运算符

- A + B

操作符	返回类型	描述
A + B	A和B的公有类型	返回A和B的和

说明

A和B可以为除了Boolean以外，任何基本类型。如果A和B都是Interval，可隐式转化规则为：tinyint→Int→bigint →Float→Double，也就是说Int和Float的相加结果为Float。另外String和任何Interval相加结果，如果String可以转化为Double类型，则结果是double类型，否则结果为NULL，比如1+'1|1'的结果为NULL，1+'1'的结果则是2.0。

举例

```
SELECT 1 + 0.1 AS result FROM system.dual LIMIT 1;
```

result
1.1

- A - B

操作符	返回类型	描述
A - B	A和B的公有类型	返回A和B的差

说明

A和B可以为除了Boolean以外，任何基本类型。如果A和B都是Interval，可隐式转化规则为：tinyint→Int→bigint →Float→Double，也就是说Int和Float的相减结果为Float。另外String和任何Interval相减结果，如果String可以转化为Double类型，则结果是double类型，否则结果为NULL，比如1-'1|1'的结果为NULL，1-'1'的结果则是0.0。DATE或TIMESTAMP类型可以直接与INT类型相减，其中INT的单位是天数。

DATE类型数据作减法运算的结果受开关**inceptor.udf.v2.enabled**的影响：

	inceptor.udf.v2.enabled=true	**inceptor.udf.v2.enabled=false**
A，B均为DATE	返回DECIMAL(38,18)	返回INTERVAL_DAY_TIME
A为DATE，B为INTERVAL_DAY_TIME	返回DATE	返回TIMESTAMP

举例

```
SELECT 1 - 0.1 AS result FROM system.dual LIMIT 1;
```

result
0.9

```
SET inceptor.udf.v2.enabled=true;
SELECT CAST('20180731' AS DATE) - INTERVAL '2' DAY AS result FROM system.dual LIMIT 1;
```

result
2018-07-29

```
SET inceptor.udf.v2.enabled=false;
SELECT CAST('20180731' AS DATE) - INTERVAL '2' DAY AS result FROM system.dual LIMIT 1;
```

result
2018-07-29 00:00:00.0

- A * B

操作符	返回类型	描述
A * B	A和B的公有类型	返回A和B的积

说明

A和B可以为除了Boolean以外，任何基本类型。如果A和B都是Interval，可隐式转化规则为：tinyint→Int→bigint →Float→Double，也就是说Int和Float的相乘结果为Float。另外String和任何Interval相乘结，如果String可以转化为Double类型，则结果是double类型，否则结果为NULL，比如1 * ’1|1’ 的结果为NULL，1 * ’1’ 的结果则是1.0。需要说明的是，如果乘法造成溢出，将选择更高的类型。

举例

```
SELECT 1 * 0.1 AS result FROM system.dual LIMIT 1;
```

result
0.1

- A / B

操作符	返回类型	描述
A / B	A和B的公有类型	返回A和B相除的结果

说明

A和B可以为任何数字类型或者字符串类型。如果A和B都是数字类型，则返回double类型。另外字符类型和数字类型相除，字符类型如果可以转化为double类型则结果也是double类型，否则返回NULL。

举例

```
SELECT 1 / 1 AS result FROM system.dual LIMIT 1;
```

result
1.0

- A % B

操作符	返回类型	描述
A % B	A和B的公有类型	返回A被B除的余数

说明

A和B可以为任何数字类型或者字符串类型。如果A和B都是数字类型，则返回二者的共同类型。如果是字符类型和数字类型，字符类型如果可以转化为double类型则结果也是double类型，否则返回NULL。

举例

```
SELECT '10' % 3 AS result FROM system.dual LIMIT 1;
```

result
1.0

- A & B

操作符	返回类型	描述
A & B	A和B的公有类型	返回A和B的二进制值的按位与结果

说明

A和B可以为TinyInt,Int,BitInt类型。运算符查看两个参数的二进制表示法的值，并执行按位”与”操作。两个表达式的一位均为1时，则结果的该位为 1。否则，结果 的该位为 0。

举例

```
SELECT 1&2 AS result FROM system.dual LIMIT 1;
```

result
0

- A | B

操作符	返回类型	描述
A | B	A和B的公有类型	返回A和B的二进制值的按位或结果

说明

A和B可以为TinyInt,Int,BitInt类型。运算符查看两个参数的二进制表示法的值，并执行按位”或”操作。只要任一表达式的一位为 1，则结果的该位为 1。否则，结果的该位为 0。

举例

```
SELECT 1|2 AS result FROM system.dual LIMIT 1;
```

result
3

- A ^ B

操作符	返回类型	描述
A ^ B	A和B的公有类型	返回A和B的二进制值的按位异或结果

说明

A和B可以为TinyInt,Int,BitInt类型。运算符查看两个参数的二进制表示法的值，并执行按位”异或”操作。当且仅当只有一个表达式的某位上为 1 时，结果的该位才为 1。否则结果的该位为 0。

举例

```
SELECT 1^1 AS result FROM system.dual LIMIT 1;
```

result
0

- ~A

操作符	返回类型	描述
~A	和A类型相同	返回把A的二进制表达式按位取反的值

说明

A可以为TinyInt,Int,BitInt类型。对表达式 A 执行按位“非”（取反）。

举例

```
SELECT ~1 AS result FROM system.dual LIMIT 1;
```

result
-2

8.9.5. 逻辑运算符

- A AND B

操作符	返回类型	描述
A AND B	Boolean	返回A和B逻辑与的结果

说明

A和B必须是Boolean类型，返回二者的逻辑与的结果。即A和B同时正确时,返回TRUE,否则FALSE。如果A或B值为NULL，返回NULL。

举例

```
SELECT 1==1 and true AS result FROM system.dual LIMIT 1;
```

result
true

- A OR B

操作符	返回类型	描述
A OR B	Boolean	返回A和B逻辑或的结果

说明

A和B必须是Boolean类型，返回二者的逻辑或的结果。即A或B正确,或两者同时正确返返回TRUE,否则FALSE。如果A和B值同时为NULL，返回NULL。

举例

```
SELECT 1==1 or true AS result FROM system.dual LIMIT 1;
```

result
true

- NOT A 或 !A

操作符	返回类型	描述
NOT A 或 !A	Boolean	对A的逻辑布尔值求反

说明

A必须是Boolean类型，如果A为NULL或错误的时候返回TURE，否则返回FALSE。

举例

```
SELECT not true AS result FROM system.dual LIMIT 1;
```

result
false

```
SELECT !true AS result FROM system.dual LIMIT 1;
```

result
false

- A IN (val1, val2, …)

操作符	返回类型	描述
A IN (val1, val2, …)	Boolean	判断A是否是指定值中的一个

说明

A可以是除了Boolean之外的所有基本类型。如果A和指定值中的任何一个相等，即返回true，反之返回false。

举例

```
SELECT '1' in (1,2,3) AS result FROM system.dual LIMIT 1;
```

result
true

- A NOT IN (val1, val2, …)

操作符	返回类型	描述
A NOT IN (val1, val2, …)	Boolean	判断A是否不是指定值中的一个

说明

A可以是除了Boolean之外的所有基本类型。如果A和指定值中的任何值都不相等，返回true，反之返回false。

举例

```
SELECT '1' not in (1,2,3) AS result FROM system.dual LIMIT 1;
```

result
false

- [NOT] EXISTS (subquery)

操作符	返回类型	描述
[NOT] EXISTS (subquery)	Boolean	判断子查询中是否有结果

说明

只要子查询返回了至少一列的值，返回true，反之返回false。

举例

```
SELECT 1 AS result FROM system.dual s1
WHERE EXISTS (
    SELECT 1 AS result FROM system.dual s2 WHERE s1.dummy=s2.dummy
) LIMIT 1;
```

result
1

8.9.6. 数学函数

8.9.6.1. 简单求值

8.9.6.1.1. round(double a[, int d])

函数名	返回类型	描述
round(double a[, int d])	BigInt	四舍五入。

说明

未指定d时，四舍五入到小数点后一位；若指定了d，则四舍五入到小数点后第d位。

使用示例：

```
SELECT round(1.4523,3) AS result FROM system.dual LIMIT 1;
```

result
1.452

8.9.6.1.2. floor(double a)

函数名	返回类型	描述
floor(double a)	BigInt	返回不大于a的最大整数

使用示例：

```
SELECT floor(1.314) AS result FROM system.dual LIMIT 1;
```

result
1

8.9.6.1.3. ceil(double a) 或 ceiling(double a)

函数名	返回类型	描述
ceil(double a) 或 ceiling(double a)	BigInt	返回不小于a的最小整数

使用示例：

```
SELECT ceil(2.01) AS result FROM system.dual LIMIT 1;
```

result
3

```
SELECT ceiling(2.01) AS result FROM system.dual LIMIT 1;
```

result
3

8.9.6.1.4. rand([int seed])

函数名	返回类型	描述
rand([int seed])	Double	生成0-1的随机数

说明

未指定seed时随机返回0-1的随机数；指定seed时；相同的seed得到的随机数结果是一样的，例如每次rand(100)的值都是一样的。

使用示例：

```
SELECT rand(100) AS result FROM system.dual LIMIT 1;
```

result
0.7220096548596434

8.9.6.1.5. abs(double a)

函数名	返回类型	描述
abs(double a)	Double	返回a的绝对值

使用示例：

```
SELECT abs(-1314) AS result FROM system.dual LIMIT 1;
```

result
1314

8.9.6.1.6. pmod(int a, int b)

函数名	返回类型	描述
pmod(int a, int b)	Numeric	返回正的取模，公式：(a%b+b)%b

使用示例：

```
SELECT pmod(-4,3) AS result FROM system.dual LIMIT 1;
```

result
2

8.9.6.1.7. positive(int a)

函数名	返回类型	描述
positive(int a)	Numeric	直接返回a

使用示例：

```
SELECT positive(-1) AS result FROM system.dual;
```

result
-1

8.9.6.1.8. negative(int a)

函数名	返回类型	描述
negative(int a)	Numeric	返回-a

使用示例：

```
SELECT negative(1) AS result FROM system.dual;
```

result
-1

8.9.6.1.9. sign(x)

函数名	返回类型	描述
sign(x)	Double	判断正负

说明

x为正则返回1，为负则返回-1，为0则返回0.

使用示例：

```
SELECT sign(-10) AS result FROM system.dual LIMIT 1;
```

result
-1

8.9.6.1.10. pi()

函数名	返回类型	描述
pi()	Double	返回pi的值，保留到小数点后15位

使用示例：

```
SELECT pi() AS result FROM system.dual;
```

result
3.141592653589793

8.9.6.1.11. e()

函数名	返回类型	描述
e()	Double	返回自然常数e，保留到小数点后15位

使用示例：

```
SELECT e() AS result FROM system.dual;
```

result
2.718281828459045

8.9.6.1.12. A div B

函数名	返回类型	描述
A div B	Integer	返回整数除法运算的商，舍去小数部分，只保留整数部分。

参数说明：

- **A**：被除数。
- **B**：除数。

使用示例：

计算 10 除以 3 的商。

```
SELECT 10 div 3  AS result;
```

返回示例：

```
+---------+
| result  |
+---------+
| 3       |
+---------+
```

8.9.6.1.13. prime(<int>)

函数	返回类型	描述
prime(<int>)	INT	返回小于等于参数 **<int>** 的最大质数。

使用示例：

```
SELECT prime(4) AS result FROM system.dual LIMIT 1;
```

result
3

8.9.6.1.14. max_scalar(<scalar>, <scalar> [, <scalar>, …])

函数	返回类型	描述
max_scalar(<scalar>, <scalar> [, <scalar>, …])	数值类型	需要至少两个标量参数。返回所有参数中最大的。

使用示例：

```
SELECT max_scalar(1, 4, 8) AS result FROM system.dual LIMIT 1;
```

result
8

8.9.6.1.15. min_scalar(<scalar>, <scalar> [, <scalar>, …])

函数	返回类型	描述
min_scalar(<scalar>, <scalar> [, <scalar>, …])	数值类型	需要至少两个标量参数。返回所有参数中最小的。

使用示例：

```
SELECT min_scalar(1,4,8) AS result FROM system.dual LIMIT 1;
```

result
1

8.9.6.1.16. least (<arg>, <arg> [,<arg>, …])

函数	返回类型	描述
least (<arg>, <arg> [,<arg>, …])	见描述	至少需要两个参数。返回最小参数。

使用示例：

```
SELECT least(2, 3, 1) AS result FROM system.dual LIMIT 1;
```

result
1

8.9.6.1.17. trunc (<number>, <precision>)

函数	返回类型	描述
trunc (<number>, <precision>)	数值类型	需要两个参数：一个数字 <number>，和精度 <precision>（小数点后保留的位数）。该函数将 <number> 截到指定的 <precision>。

使用示例：

```
SELECT trunc(123.456, 1) AS result FROM system.dual LIMIT 1;
```

result
123.4

8.9.6.1.18. shiftleft(n, int k)

函数	返回类型	描述
shiftleft(n, int k)	数值类型	返回整数n左移k位后的数值。参数 **n** 可接受所有整数值，参数 **k** 必须是INT类型。

使用示例：

```
SELECT shiftleft(8,1) AS result FROM system.dual;
```

result
16

8.9.6.1.19. shiftright(n, int k)

函数	返回类型	描述
shiftright(n, int k)	数值类型	返回整数n右移k位后的数值。参数 **n** 可接受所有整数值，参数 **k** 必须是INT类型。

使用示例：

```
SELECT shiftright(8,1) AS result FROM system.dual;
```

result
4

8.9.6.1.20. shiftrightunsigned(n, int k)

函数	返回类型	描述
shiftrightunsigned(n, int k)	数值类型	返回整数n无符号右移k位后的数值。参数 **n** 可接受所有整数值，参数 **k** 必须是INT类型。

使用示例：

```
SELECT shiftrightunsigned(-8,1) AS result FROM system.dual;
```

result
2147483644

8.9.6.1.21. factorial(int n)

函数	返回类型	描述
factorial(int n)	INT	返回n的阶乘。如果n的值不是属于[0,20]的整数，返回null。

使用示例：

```
SELECT factorial(20) AS result FROM system.dual;
```

result
2432902008176640000

8.9.6.2. 幂/方根/对数

8.9.6.2.1. exp(double n)

函数名	返回类型	描述
exp(double n)	Double	返回e的n次方

使用示例：

```
SELECT exp(1) AS result FROM system.dual LIMIT 1;
```

result
2.718281828459045

8.9.6.2.2. ln(double a)

函数名	返回类型	描述
ln(double a)	Double	返回指定值的自然对数

使用示例：

```
SELECT ln(exp(1.314)) AS result FROM system.dual;
```

result
1.314

8.9.6.2.3. log10(double a)

函数名	返回类型	描述
log10(double a)	Double	返回a以10为底的对数

使用示例：

```
SELECT log10(100) AS result FROM system.dual LIMIT 1;
```

result
2.0

8.9.6.2.4. log2(double a)

函数名	返回类型	描述
log2(double a)	Double	返回a以2为底的对数

使用示例：

```
SELECT log2(4) AS result FROM system.dual LIMIT 1;
```

result
2.0

8.9.6.2.5. log(double base, double a)

函数名	返回类型	描述
log(double base, double a)	Double	返回a以base为底的对数

使用示例：

```
SELECT log(10,100) AS result FROM system.dual LIMIT 1;
```

result
2.0

8.9.6.2.6. pow(double a, double p)

函数名	返回类型	描述
pow(double a, double p)	Double	返回a的p次幂

使用示例：

```
SELECT pow(10,2) AS result FROM system.dual LIMIT 1;
```

result
100.0

8.9.6.2.7. power(double a, double p)

函数名	返回类型	描述
power(double a, double p)	Double	返回a的p次幂

使用示例：

```
SELECT power(10,2) AS result FROM system.dual LIMIT 1;
```

result
100.0

8.9.6.2.8. sqrt(double a)

函数名	返回类型	描述
sqrt(double a)	DOUBLE	返回a的平方根

使用示例：

```
SELECT sqrt(100) AS result FROM system.dual;
```

result
10.0

8.9.6.2.9. cbrt(a)

函数名	返回类型	描述
cbrt(a)	Double	计算参数a的实立方根。a可以是任意数值类型。

使用示例：

```
SELECT cbrt(27) AS result FROM system.dual;
```

result
3.0

8.9.6.3. 进制/度量转换

8.9.6.3.1. bin(BIGINT a)

函数名	返回类型	描述
bin(BIGINT a)	String	返回二进制格式

使用示例：

```
SELECT bin(10) AS result FROM system.dual LIMIT 1;
```

result
1010

8.9.6.3.2. hex(BIGINT a) / hex(String a)

函数名	返回类型	描述
hex(BIGINT a) / hex(String a)	String	返回十六进制格式

说明

如果参数为BIGINT或者BINARY，函数将参数的十六进制以字符串形式输出。如果参数为字符串，函数按顺序将各个字符的十六进制ascii码以字符串形式输出。

使用示例：

```
SELECT hex(16) AS result FROM system.dual LIMIT 1 ;
```

result
10

```
SELECT hex('16') AS result FROM system.dual LIMIT 1 ;
```

result
3136

8.9.6.3.3. unhex(String a)

函数名	返回类型	描述
unhex(String a)	String	返回十六进制格式

说明

十六进制转二进制格式，只接受字符格式，即使输入数值类型也转化为字符格式再转化为二进制。

使用示例：

```
SELECT unhex(3136) AS result FROM system.dual LIMIT 1 ;
```

result
16

8.9.6.3.4. conv(BIGINT num, int from_base, int to_base)

函数名	返回类型	描述
conv(BIGINT num, int from_base, int to_base)	String	度量体系转化。

说明

将指定数值num，由原来的度量体系(from_base)转换为指定的度量体系(to_base)。任一参数为NULL，则返回结果为NULL。num可以为int，也可以是数字字符串。度量体系的取值范围为2-36。num没有符号，默认为无符号数。如果num带有“-”号，则为负数。

使用示例：

```
SELECT conv(100, 2, 10) AS result FROM system.dual LIMIT 1;
```

result
4

8.9.6.3.5. degrees(double rad)

函数名	返回类型	描述
degrees(double rad)	Double	把弧度制的rad转化为角度。

使用示例：

```
SELECT degrees(pi()) AS result FROM system.dual;
```

result
180.0

8.9.6.3.6. radians(double ang)

函数名	返回类型	描述
radians(double ang)	Double	把角度制的ang转化为弧度。

使用示例：

```
SELECT radians(180) AS result FROM system.dual;
```

result
3.141592653589793

8.9.6.4. 三角函数

8.9.6.4.1. sin(double a)

函数名	返回类型	描述
sin(double a)	Double	返回a的正弦值，其中a是弧度制。

使用示例：

```
SELECT sin(pi()/2) AS result FROM system.dual;
```

result
1.0

8.9.6.4.2. sinh(double a)

函数名	返回类型	描述
sinh(double a)	Double	计算并返回指定数值的双曲正弦值，以弧度为单位。

参数说明：

- **double a**：要计算的值。

使用示例：

计算 `1.5` 的双曲正弦值。

```
SELECT sinh(1.5) AS result;
```

返回示例：

```
+---------------------+
|       result        |
+---------------------+
| 2.1292794550948173  |
+---------------------+
```

8.9.6.4.3. asin(double a)

函数名	返回类型	描述
asin(double a)	Double	返回a的反正弦值，a必须大于-1小于1，否则返回null。

使用示例：

```
SELECT asin(1) AS result FROM system.dual;
```

result
1.5707963267948966

8.9.6.4.4. asinh(double a)

函数名	返回类型	描述
asinh(double a)	Double	计算并返回指定数值的反双曲正弦值，以弧度为单位。

参数说明：

double a： 要计算反双曲正弦值的数。

使用示例：

计算数值 2 的反双曲正弦值。

```
SELECT asinh(2) AS result;
```

返回示例：

```
+---------------------+
|       result        |
+---------------------+
| 1.4436354751788103  |
+---------------------+
```

如果希望以度为单位表示反正切值，您可以将结果乘以 180/PI( )，或使用 degrees 函数。

8.9.6.4.5. cos(double a)

函数名	返回类型	描述
cos(double a)	Double	返回a的余弦值，其中a是弧度制。

使用示例：

```
SELECT cos(pi()) AS result FROM system.dual;
```

result
-1.0

8.9.6.4.6. cosh(double a)

函数名	返回类型	描述
cosh(double a)	Double	计算并返回指定数值的双曲余弦值，以弧度为单位。

参数说明：

- **double a：** 要计算的值。

使用示例：

计算输入值为 `1.5` 的双曲余弦值。

```
SELECT cosh(1.5) AS result;
```

返回示例：

```
+--------------------+
|       result       |
+--------------------+
| 2.352409615243247  |
+--------------------+
```

8.9.6.4.7. acosh(double a)

函数名	返回类型	描述
acosh(double a)	Double	计算并返回指定数值的反双曲余弦值，以弧度为单位。

参数说明：

- **double a**：要计算反双曲余弦的值，必须大于等于 1。

使用示例：

计算 3 的反双曲余弦值。

```
SELECT acosh(3) AS result;
```

返回示例：

```
+--------------------+
|       result       |
+--------------------+
| 1.762747174039086  |
+--------------------+
```

8.9.6.4.8. acos(double a)

函数名	返回类型	描述
acos(double a)	Double	返回a的反余弦值，其中a是弧度制。

使用示例：

```
SELECT acos(1) AS result FROM system.dual;
```

result
0.0

8.9.6.4.9. tan(double a)

函数名	返回类型	描述
tan(double a)	Double	返回a的正切值，其中a是弧度制。

使用示例：

```
SELECT tan(pi()/4) AS result FROM system.dual;
```

result
0.9999999999999999

8.9.6.4.10. tanh(double a)

函数名	返回类型	描述
tanh(double a)	Double	计算并返回指定数值的双曲正切值，以弧度为单位。

参数说明：

- **double a**：要计算双曲正切的值。

使用示例：

在 SQL 中计算 0.5 的双曲正切值。

```
SELECT tanh(0.5) AS result;
```

返回示例：

```
+----------------------+
|        result        |
+----------------------+
| 0.46211715726000974  |
+----------------------+
```

8.9.6.4.11. atan(double a)

函数名	返回类型	描述
atan(double a)	Double	返回a的余切值，其中a是弧度制。

使用示例：

```
SELECT atan(1) AS result FROM system.dual;
```

result
0.7853981633974483

8.9.6.4.12. atan2(x, y)

函数名	返回类型	描述
atan2(x, y)	Double	计算 X 轴及 Y 轴坐标值的反正切值，以弧度为单位。

参数说明：

- **x：** 点的横坐标。
- **y：** 点的纵坐标。

使用示例：

计算点（1, 1）相对于原点（0, 0）的反正切值。

```
SELECT atan2(-1, -1) AS result;
```

返回示例：

```
+---------------------+
|       result        |
+---------------------+
| -2.356194490192345  |
+---------------------+
```

注意事项：

- 如果希望以度为单位表示反正切值，您可以将结果乘以 180/PI( )，或使用 degrees 函数。
- 结果为正表示从 X 轴逆时针旋转的角度，结果为负表示从 X 轴顺时针旋转的角度。

8.9.6.4.13. atanh(double a)

函数名	返回类型	描述
atanh(double a)	Double	返回指定指的反双曲正切值，以弧度为单位。

参数说明：

- **double a：** 要计算反双曲正切的值，取值范围：-1 ~ 1。

使用示例：

计算 0.5 的反双曲正切值。

```
SELECT atanh(0.5) AS result;
```

返回示例：

```
+---------------------+
|       result        |
+---------------------+
| 0.5493061443340549  |
+---------------------+
```

8.9.6.5. 线性回归函数

Inceptor提供以下线性回归函数（Linear Regression Functions），适用于使用最小二乘法对两列（自变量与因变量）的依赖关系进行回归分析的场景。

本小节用例所使用的表regrTable，列名从左到右为regr_id, price1, price2数据如下:

```
1|10|5
2|20|10
3|30|15
4|40|10
5|50|20
6|60|30
7|70|15
8|80|20
9|90|60
10|100|40
11|110|50
12|120|100
```

8.9.6.5.1. regr_slope(col y,col x)

函数名	返回类型	描述
regr_slope(col y,col x)	Double	返回拟合所给两列数据的回归线斜率

说明

col y 和 **col x** 为列名，必须是数值类型。

计算公式为:

(N*SUM(x*y)−SUM(x)*SUM(y))/(N*SUM(x*x)−SUM(x)*SUM(x))

N为值都不为空的数据对的数量。当N*SUM(x*x)=SUM(x)*SUM(x)时，将会返回null。

使用示例:

```
SELECT regr_slope(price2,price1) AS result FROM regrtable;
```

result
1.0860134629768137

8.9.6.5.2. regr_intercept(col y,col x)

函数名	返回类型	描述
regr_intercept(col y,col x)	Double	返回拟合所给两列数据的回归线的y截距

说明

col y 和 **col x** 为列名，必须是数值类型。

计算公式为:

(SUM(y)*SUM(x*x)−SUM(x)*SUM(x*y))/(N*SUM(x*x)−SUM(x)*SUM(x))

N为值都不为空的数据对的数量。如果N*SUM(x*x)=SUM(x)*SUM(x)，将会返回null。

使用示例：

```
SELECT regr_intercept(price2, price1) AS result FROM regrtable;
```

result
31.062079281974576

8.9.6.5.3. regr_r2(col y,col x)

函数名	返回类型	描述
regr_r2(col y,col x)	Double	返回拟合所给两列数据的回归线的决定系数

说明

col y 和 **col x** 为列名，必须是数值类型。

计算公式为：

POWER(N*SUM(x*y)−SUM(x) * SUM(y), 2) / [(N*SUM(x^2)−SUM(x)^2)*(N*SUM(y^2)−SUM(y)^2)]

N为值都不为空的数据对的数量。如果N*SUM(x*x)=SUM(x)*SUM(x)，将会返回null；如果N*SUM(y*y)=SUM(y)*SUM(y)，将返回1。

使用示例：

```
SELECT regr_r2(price2, price1) AS result FROM regrtable;
```

result
0.6892008515045164

8.9.6.5.4. regr_sxx(col y,col x)

函数名	返回类型	描述
regr_sxx(col y,col x)	Double	返回自变量col x的平方和

说明

col y 和 **col x** 为列名，必须是数值类型。

计算公式为：

(SUM(X^2)−SUM(X)^2)/N

使用示例：

```
SELECT regr_sxx(price2, price1) AS result FROM regrtable;
```

result
14299.999999999998

8.9.6.5.5. regr_syy(col y,col x)

函数名	返回类型	描述
regr_syy(col y,col x)	Double	返回因变量col y的平方和

说明

col y 和 **col x** 为列名，必须是数值类型。

计算公式为：

$(SUM(y^2)-SUM(y)^2)/N$

使用示例：

```
SELECT regr_syy(price2, price1) AS result FROM regrtable;
```

result
8356.25

8.9.6.5.6. regr_sxy(col y,col x)

函数名	返回类型	描述
regr_sxy(col y,col x)	Double	返回自变量和因变量的乘方积

说明

col y 和 **col x** 为列名，必须是数值类型。

计算公式为：

(SUM(x*y)−SUM(x)*SUM(y))/N

如果N=0，则返回null。

使用示例：

```
SELECT regr_sxy(price2, price1) AS result FROM regrtable;
```

result
9075.0

8.9.6.5.7. regr_avgx(col y,col x)

函数名	返回类型	描述
regr_avgx(col y,col x)	Double	返回自变量col x的平均值

说明

col y 和 **col x** 为列名，必须是数值类型。

计算公式为：

SUM(x)/N

使用示例：

```
SELECT regr_avgx(price2, price1) as result FROM regrtable;
```

result
65.0

8.9.6.5.8. regr_avgy(col y,col x)

函数名	返回类型	描述
regr_avgy(col y,col x)	Double	返回因变量col y的平均值

说明

col y 和 **col x** 为列名，必须是数值类型。

计算公式为：

SUM(y)/N

使用示例：

```
SELECT regr_avgy(price2, price1) as result FROM regrtable;
```

result
31.25

8.9.6.5.9. regr_count(col y,col x)

函数名	返回类型	描述
regr_count(col y,col x)	Bigint	返回两列都不为NULL的输入行数

说明

col y 和 **col x** 为列名，必须是数值类型。

使用示例：

```
SELECT regr_count(price2, price1) as result FROM regrtable;
```

result
12

8.9.7. 类型转换函数

8.9.7.1. BINARY(<string>\|<binary>)

函数名	返回类型	描述
BINARY(<string>|<binary>)	STRING	将参数转换成BINARY类型。

参数说明：

参数仅允许为STRING或者BINARY类型。

使用示例：

```
SELECT BINARY('38') AS result FROM system.dual LIMIT 1;
```

返回示例：

result
38

8.9.7.2. CAST

函数名	返回类型	描述
CAST(<expr> AS <TYPE>)	STRING	将参数转换一个指定类型。

参数说明：

expr 代表被转换的对象，**TYPE** 代表指定目标类型。

expr 支持含有毫秒的日期STRING。**inceptor.udf.v2.enabled** 开关开启时，如果输入日期值含有毫秒，则该毫秒级数会保存在返回的DATE类型结果中，可以通过 **CAST <expr> AS TIMESTAMP** 提取出来。

当 **inceptor.udf.v2.enabled** 和 **inceptor.support.english.date.format** 中有一个为true时，支持英文日期格式作为合法输入格式，如 **CAST('20-JUL-18') AS DATE**。

使用示例 1：

```
SELECT CAST('38' AS INT) AS result FROM system.dual LIMIT 1;
```

返回示例 1：

result
38

使用示例 2：

```
SET inceptor.udf.v2.enabled=true;
SELECT CAST(CAST('31-JUL-18 12:12:12.111' AS TIMESTAMP) AS DATE) AS result FROM system.dual LIMIT 1;
```

返回示例 2：

result
2018-07-31 12:12:12

8.9.7.3. DATE

函数名	返回类型	描述
DATE(<date_string>)	DATE	将指定形式的STRING转换为DATE类型。

参数说明：

和 **CAST(<expr> AS DATE)** 形式的CAST 函数意义相同。**date_string** 必须是一个STRING，而且只能有下面几种形式：

- **'yyyyMMdd'**，如 **'20150101'**
- **'yyyy−MM−dd'**，如 **'2015−01−01'**
- **'yyyy−MM−dd HH:mm:ss'**，如 **'2015−01−01 00:00:00'**。支持含有毫秒的值，如 **'2018−07−31 12:12:12.111'**

在**inceptor.udf.v2.enabled**开关开启时，输入值中的毫秒级数会被保存在返回的DATE类型结果中，可以通过*CAST <expr> AS TIMESTAMP*提取出来。

此外，**inceptor.udf.v2.enabled**和*inceptor.support.english.date.format*中有一个为true时，支持英文日期格式作为合法输入格式。

使用示例 1：

```
SELECT DATE('2015-01-01 00:00:00') AS result FROM system.dual LIMIT 1;
```

返回示例 1：

result
2015-01-01

使用示例 2：

```
SET inceptor.udf.v2.enabled=true;
SELECT DATE('20-JUL-18 12:12:12.111') AS result FROM system.dual LIMIT 1;
```

返回示例 2：

result
2018-07-20 12:12:12

8.9.7.4. decimal(expression)

函数名	返回类型	描述
decimal(expression)	十进制数值	将一个表达式转换为十进制数值。

说明：

- **expression**：要转换为十进制数值的表达式，表达式可包含列名、数值位数和精度，其中精度最大为小数点后两位。

使用示例 1：

```
SELECT decimal('123.45') AS result;
```

返回示例 1：

```
+-------+
| result|
+-------+
| 123.45|
+-------+
```

使用示例 2：

```
--我们有这样一个表
SELECT * FROM test_decimal;;
+-----------------------------+
|              id             |
+-----------------------------+
| 1234567.81111111111         |
| 1234567.82222222222         |
| 1234567.83333333333         |
| 1234567.84444444444         |
+-----------------------------+

--使用 decimal 函数
SELECT decimal(id,9,2) AS result FROM test_decimal;
```

返回示例 2：

```
+-------------+
|   result    |
+-------------+
| 1234567.81  |
| 1234567.82  |
| 1234567.83  |
| 1234567.84  |
+-------------+
```

注意事项：

- 如果表达式无法转换为有效的十进制数值，返回空值（NULL）。

- 在进行数值计算时，注意精度丢失和舍入误差。

8.9.8. 日期函数

参数名	功能描述	默认值
inceptor.udf.v2.enabled	控制一些日期函数的新功能是否启用。为true时，表示启用新功能。具体的新功能请参考函数的详细信息。	false
inceptor.date.show.time	用于在*inceptor.udf.v2.enabled*开关开启时，控制日期类型的返回值是否显示时间部分。为false时，不显示日期类型的时间部分。**此参数只可在hive_site.xml文件中修改。**	true
inceptor.support.english.date.format	控制在日期函数中是否默认支持英文日期格式（即“dd-MMM-yy”、“dd-MMM-yy HH:mm:ss”、“dd-MMM-yyyy”、“dd-MMM-yyyy HH:mm:ss”、“yyyyMMdd”、“yyyyMMdd HH:mm:ss”）作为合法输入，为true时表示支持。**在inceptor.udf.v2.enabled开关开启时，系统支持英文日期格式作为合法输入，忽略此参数设置。**	true

8.9.8.1. ADD_MONTHS(<date>,<num_months>)

函数名	返回类型	描述
ADD_MONTHS(<date>,<num_months>)	默认为STRING; **inceptor.udf.v2.enabled** 开关开启时，返回DATE。	返回 **<date>** 加上 **<num_months>** 个月后的日期或时间。

说明

参数date应该是datetime value，或者可以隐性转换成DATE类型；支持含有毫秒的日期STRING作为输入，如'2018-07-31 12:12:12.111'。默认情况下，参数num_months必须是INT类型或者可以隐式转化为INT的类型。

在**inceptor.udf.v2.enabled**开关开启时：

返回结果默认包含不为零的时间部分。如要不显示时间部分，需要在hive_site.xml文件中设置**inceptor.date.show.time**为false;
输入值中的毫秒级数会被保存在返回的DATE类型结果中，可以通过CAST AS TIMESTAMP提取;
参数date值超出合法日期范围时会返回NULL;
参数num_months支持所有数字类型和可以转化为INT的String类型。

此外，**inceptor.udf.v2.enabled**和*inceptor.support.english.date.format*中有一个为true时，参数date支持英文日期格式作为合法输入格式。

举例

```
SELECT ADD_MONTHS('201501',1) AS result FROM system.dual LIMIT 1;
```

result
201502

```
SET inceptor.udf.v2.enabled=true;
SELECT ADD_MONTHS('31-JUL-18 12:12:12.111',1) AS result FROM system.dual LIMIT 1;
```

result
2018-08-31 12:12:12

8.9.8.2. EXTRACT(DAY\|MONTH\|YEAR FROM <date>)

函数名	返回类型	描述
EXTRACT(DAY|MONTH|YEAR FROM <date>)	STRING	抽取日期类型中的年/月/日

说明

参数date应该是datetime value，或者可以隐性转换成DATE类型。

举例

```
SELECT EXTRACT(YEAR FROM '2015-01-15') AS result FROM system.dual LIMIT 1;
```

result
2015

8.9.8.3. TO_UNIX_TIMESTAMP(<date>[, <pattern>])

函数名	返回类型	描述
TO_UNIX_TIMESTAMP(<date>[, <pattern>])	BIGINT	把日期格式转化为UNIX时间戳

说明

返回从 **'1970-01-01 8:00:00'** 开始，到指定时间为止的秒数。参数一最好是date类型，也支持其他字符类型；
参数二可选，作用是指定参数一的格式，如 **'yyyy-MM-dd HH:mm:ss'**，如果参数二指定的格式和参数一不匹配，则返回 **NULL**。

举例

```
SELECT TO_UNIX_TIMESTAMP("2015-01-15",'yyyy-MM-dd') AS result FROM system.dual LIMIT 1;
```

result
1421251200

8.9.8.4. UNIX_TIMESTAMP([<date>[, <pattern>]]) 或 TO_TIMESTAMP([<date>[, <pattern>]])

函数名	返回类型	描述
UNIX_TIMESTAMP([<date>[, <pattern>]])	BIGINT	把日期格式转化为UNIX时间戳

说明

返回从 **"1970-01-01 8:00:00"** 开始，到指定时间为止的秒数。
如果不加参数，则返回当前系统时间的UNIX时间戳。
参数一最好是DATE类型，也支持其他字符类型；
参数二可选，作用是指定参数一的格式，如 **'yyyy-MM-dd HH:mm:ss'**，如果参数二指定的格式和参数一不匹配，则返回NULL。

举例

```
SELECT UNIX_TIMESTAMP("2015-01-15",'yyyy-MM-dd') AS result FROM system.dual LIMIT 1;
```

result
1421251200

```
SELECT TO_TIMESTAMP("2015-01-15",'yyyy-MM-dd') AS result FROM system.dual LIMIT 1;
```

result
1421251200

8.9.8.5. FROM_UNIXTIME(<unix_time>[, <format>])

函数名	返回类型	描述
FROM_UNIXTIME(<unix_time>[, <format>])	STRING	把UNIX时间戳转化为时间格式

说明

参数一是LONG类型，或者可以隐性转换成LONG类型的UNIX时间戳；参数二可选，是字符类型，指定了返回的时间格式。当没有参数二时，返回标准的 **'yyyy-MM-dd HH:mm:ss'** 时间格式。

举例

```
SELECT FROM_UNIXTIME(0, 'yyyy-MM-dd HH:mm:ss') AS result FROM system.dual LIMIT 1;
```

result
1970-01-01 08:00:00

8.9.8.6. TO_CHAR(<date>[, <pattern>])

函数名	返回类型	描述
TO_CHAR(<date>[, <pattern>])	STRING	把日期<date>转化为指定格式<pattern>

说明

如果<date>是string类型，要求其格式必须为 **yyyy-MM-dd HH:mm:ss** 或者 **yyyy-MM-dd**。若<pattern>缺省，则以原形式返回。

举例

```
SELECT to_char('2011-05-11 10:00:12', 'yyyyMMdd') AS result FROM system.dual LIMIT 1;
```

result
20110511

8.9.8.7. TO_DATE(<date>[, <pattern>])

函数名	返回类型	描述
TO_DATE(<date>[, <pattern>])	默认为STRING; **inceptor.udf.v2.enabled** 开关开启时，返回DATE。	把字符串或者日期字符转化为 **yyyy-MM-dd** 的日期格式。

说明

参数date必须是字符类型或者是日期类型、TIMESTAMP类型，否则返回为NULL。支持含有毫秒的日期STRING作为输入，如'2018-07-31 12:12:12.111'。

在**inceptor.udf.v2.enabled**开关开启时：

返回结果默认包含不为零的时间部分。如要不显示时间部分，需要在hive_site.xml文件中设置**inceptor.date.show.time**为false;
输入值中的毫秒级数会被保存在返回的DATE类型结果中，可以通过CAST AS TIMESTAMP提取;
可以使用pattern参数指定输入的格式，如TO_DATE('23-JUL-2018','dd-MMM-yyyy');
参数date值超出合法日期范围时会返回NULL。

此外，**inceptor.udf.v2.enabled**或**inceptor.support.english.date.format**有一个为true时，参数date支持英文日期格式作为合法输入格式。

举例

```
SELECT TO_DATE('2015-01-15 04:17:52') AS result FROM system.dual LIMIT 1;
```

result
2015-01-15

```
SET inceptor.udf.v2.enabled=true;
SELECT TO_DATE('23-JULY-18','dd-MMM-yy') AS result FROM system.dual LIMIT 1;
```

result
2018-07-23

8.9.8.8. TDH_TODATE(<date>[, <originalformat>,<targetformat>])

函数名	返回类型	描述
TDH_TODATE(<date>[, <originalformat>,<targetformat>])	STRING	把字符按照给定的格式，转化为标准日期格式，或者指定格式

说明

参数date必须是字符类型或者是日期类型、TIMESTAMP类型，否则返回为 **NULL**。如果只提供日期一个参数，系统会自动识别 **yyyyMMdd**、**yyyy-MM-dd**、**yyyy/MM/dd** 格式的字符或日期，然后返回 **yyyy-MM-dd** 格式的字符串，其他返回 **NULL**。

当指定了 **originalformat**（参数一的原始日期格式）时，会根据指定的格式去解析日期，如果格式和日期参数不匹配也返回 **NULL**，建议手动指定 **originalformat**。

参数 **targetformat** 也是STRING类型，指定返回的日期格式，如 **yyyy*MM*dd**。

三个参数 **date**、**originalformat** 和 **targetformat** 都可以包含时分秒。但是 **originalformat** 和 **targetformat** 缺省的情况下，**tdh_todate** 只返回 **yyyy-MM-dd** 格式的年月日。所以要转换包含时分秒信息的日期必须指定 **original format** 和*targetformat*。时分秒的格式为 **HH:mm:ss.SS**（24小时制）或者 **hh:mm:ss.SS**（12小时制）。

inceptor.udf.compatible.with.oracle 参数来控制兼容性。当此开关为 **true** 时，可以识别非法日期和英文日期，例如 TDH_TODATE("2018-02-31") 会返回 NULL，TDH_TODATE("25-MAR-18") 则会返回 "2018-03-25"。

举例

```
SELECT TDH_TODATE('2015-01-15 00:00:00') AS result FROM system.dual LIMIT 1;
```

result
2015-01-15

```
SELECT TDH_TODATE('2015-11-24 18:00:00','yyyy-MM-dd HH:mm:ss','yyyy/MM/dd hh:mm:ss') AS result FROM
system.dual LIMIT 1;
```

result
2015/11/24 06:00:00

8.9.8.9. DAY(<date>) 或 DAYOFMONTH(<date>)

函数名	返回类型	描述
DAY(<date>) 或 DAYOFMONTH(<date>)	INT	返回指定时间是该月的第几天

说明

参数date必须是字符类型或者是日期类型、Timestamp类型，否则返回为 **NULL**。另外date必须是 **yyyy-MM-dd** 或者 **yyyy-MM-dd HH:MM:SS** 的格式，否则也返回NULL。

举例

```
SELECT DAY('2015-01-15') AS result FROM system.dual LIMIT 1;
```

result
15

```
SELECT DAYOFMONTH('2015-01-15') AS result FROM system.dual LIMIT 1;
```

result
15

8.9.8.10. DAYOFYEAR(<date>[,<format>])

函数名	返回类型	描述
DAYOFYEAR(<date>[,<format>])	INT	返回指定日期是该年的第几天

说明

参数date必须是字符类型或者是日期类型、Timestamp类型，否则返回为NULL。如果只提供日期一个参数，系统会自动识别 **yyyyMMdd**、**yyyy-MM-dd**、**yyyy/MM/dd** 三种格式的字符或日期，然后返回日期是该年的第几天，其他格式返回NULL。当指定了第二个可选参数(STRING类型的日期格式)时，会根据指定的格式去解析日期，如果格式和日期参数不匹配也返回NULL，建议手动指定第二个参数。

举例

```
SELECT DAYOFYEAR('2015|01|15','yyyy|MM|dd') AS result FROM system.dual LIMIT 1;
```

result
15

8.9.8.11. year(<date>)

函数名	返回类型	描述
year(<date>)	INT	返回给定日期或时间值的年份值。

参数说明：

- date：支持字符、日期和时间戳类型，且格式为 `yyyy-MM-dd HH:MM:SS`。

使用示例：

```
SELECT year('2023-12-31') AS result;
```

返回示例：

```
+---------+
| result  |
+---------+
| 2023    |
+---------+
```

8.9.8.12. QUARTER(<date>[,<format>])

函数名	返回类型	描述
QUARTER(<date>[,<format>])	INT	返回指定日期是该年的第几季度

说明

参数date必须是字符类型或者是日期类型、Timestamp类型，否则返回为NULL。如果只提供日期一个参数，系统会自动识别yyyyMMdd、yyyy-MM-dd、yyyy/MM/dd三种格式的字符或日期，然后返回日期是该年的第几季度，其他格式返回NULL。当指定了第二个可选参数(String类型的日期格式)时，会根据指定的格式去解析日期，如果格式和日期参数不匹配也返回NULL，建议手动指定第二个参数。

举例

```
SELECT QUARTER('2015|06|15','yyyy|MM|dd') AS result FROM system.dual LIMIT 1;
```

result
2

8.9.8.13. month(<date>)

函数名	返回类型	描述
month()	整数	返回给定日期或时间值的月份值

参数说明：

- date：支持字符、日期和时间戳类型，且格式为 `yyyy-MM-dd HH:MM:SS`。

使用示例：

获取当前日期的月份。

```
SELECT month('2023-12-31') AS result;
```

返回示例：

```
+--------+
| result |
+--------+
| 12     |
+--------+
```

8.9.8.14. HOUR(<date>)

函数名	返回类型	描述
HOUR(<date>)	INT	返回指定时间是该日的第几个小时

说明

参数date必须是字符类型或者是日期类型、Timestamp类型，同时必须yyyy-MM-dd HH:MM:SS的格式，否则返回NULL。

举例

```
SELECT HOUR('2015-01-30 22:58:59') AS result FROM system.dual LIMIT 1;
```

result
22

8.9.8.15. MINUTE(<date>)

函数名	返回类型	描述
MINUTE(<date>)	INT	返回指定时间是该小时的第多少分钟

说明

参数date必须是字符类型或者是日期类型、Timestamp类型，同时必须是yyyy-MM-dd HH:MM:SS的格式，否则返回NULL。

举例

```
SELECT MINUTE('2015-01-30 22:58:59') AS result FROM system.dual LIMIT 1;
```

result
58

```
*SECOND(<time>)
```

函数名	返回类型	描述
SECOND(<time>)	INT	返回指定时间是该分钟的第多少秒

说明

参数date必须必须是yyyy-MM-dd HH:MM:SS的格式，否则返回NULL。

举例

```
SELECT SECOND('2015-01-30 22:58:59') AS result FROM system.dual LIMIT 1;
```

result
59

8.9.8.16. WEEKOFYEAR

函数名	返回类型	描述
WEEKOFYEAR(<date>)	INT	返回指定日期是该年的第几周

说明

参数date必须是字符类型或者是日期类型、Timestamp类型，同时必须是是yyyy-MM-dd或者yyyy-MM-dd HH:MM:SS的格式，否则返回NULL。我们把周一作为一周的开始，同时第一个有着4天及以上的周，是第一周。

举例

```
SELECT WEEKOFYEAR('2015-01-15 22:58:59') AS result FROM system.dual LIMIT 1;
```

result
3

8.9.8.17. DATEDIFF

函数名	返回类型	描述
DATEDIFF(<date1>, <date2>)	INT	返回两个日期的相差天数

说明

参数 **date1/date2** 必须是字符类型或者是日期类型、Timestamp类型，同时必须是是yyyy-MM-dd或者yyyy-MM-dd HH:MM:SS的格式，否则返回NULL。支持含有毫秒的日期STRING作为输入，如'2018-07-31 12:12:12.111'。返回值为正说明date1较晚。

在**inceptor.udf.v2.enabled**开关开启时：

输入值中的毫秒级数会被保存在返回的DATE类型结果中，可以通过CAST AS TIMESTAMP提取；
输入值为非法日期时会返回NULL；
日期中的时间部分也会进行计算，得出的结果不足一天的结果计为0天。

此外，**inceptor.udf.v2.enabled**和*inceptor.support.english.date.format*中有一个为true时，参数支持英文日期格式作为合法输入格式。

举例

```
SELECT DATEDIFF('2015-01-15', '2015-02-08') AS result FROM system.dual LIMIT 1;
```

result
-24

```
SET inceptor.udf.v2.enabled=true;
SELECT DATEDIFF('30-JUL-18 12:12:12.111','20180731 12:12:10') AS result FROM system.dual LIMIT 1;
```

result
0

8.9.8.18. DATE_ADD(<start_date>, <num_days>)

函数名	返回类型	描述
DATE_ADD(<start_date>, <num_days>)	默认为STRING; **inceptor.udf.v2.enabled** 开关开启时，返回DATE。	返回指定时间之后的第num_days天的日期或时间

说明

参数start_date必须是字符类型或者是日期类型、Timestamp类型，同时必须是yyyy-MM-dd或者yyyy-MM-dd HH:MM:SS的格式，否则返回NULL。支持含有毫秒的日期STRING作为输入，如'2018-07-31 12:12:12.111'。默认情况下，参数num_days必须是Int类型或者可以隐式转化为Int的类型。

在**inceptor.udf.v2.enabled**开关开启时：

返回结果默认包含不为零的时间部分。如要不显示时间部分，需要在hive_site.xml文件中设置**inceptor.date.show.time**为false;
输入值中的毫秒级数会被保存在返回的DATE类型结果中，可以通过CAST AS TIMESTAMP提取;
参数start_date值超出合法日期范围时会返回NULL;
参数num_days支持所有数字类型和可以转化为INT的String类型。

此外，**inceptor.udf.v2.enabled**和*inceptor.support.english.date.format*中有一个为true时，参数start_date支持英文日期格式作为合法输入格式。

举例

```
SELECT DATE_ADD('2015-01-15', 1) AS result FROM system.dual LIMIT 1;
```

result
2015-01-16

```
SET inceptor.udf.v2.enabled=true;
SELECT DATE_ADD('2018-07-31 12:12:12.111',1) AS result FROM system.dual LIMIT 1;
```

result
2018-08-01 12:12:12

8.9.8.19. DATE_SUB(<start_date>, <num_days>)

函数名	返回类型	描述
DATE_SUB(<start_date>, <num_days>)	默认为STRING; **inceptor.udf.v2.enabled**开关开启时，返回DATE。	返回指定时间之前的第num_days天的时间

说明

参数start_date必须是字符类型或者是日期类型、Timestamp类型，同时必须是是yyyy-MM-dd或者yyyy-MM-dd HH:MM:SS的格式，否则返回NULL。支持含有毫秒的日期STRING作为输入，如'2018-07-31 12:12:12.111'。默认情况下，参数num_days必须是Int类型或者可以隐式转化为Int的类型。

在**inceptor.udf.v2.enabled**开关开启时：

返回结果默认包含不为零的时间部分。如要不显示时间部分，需要在hive_site.xml文件中设置**inceptor.date.show.time**为false;
输入值中的毫秒级数会被保存在返回的DATE类型结果中，可以通过CAST AS TIMESTAMP提取;
参数start_date值超出合法日期范围时会返回NULL;
参数num_days支持所有数字类型和可以转化为INT的String类型。

此外，**inceptor.udf.v2.enabled**和*inceptor.support.english.date.format*中有一个为true时，参数start_date支持英文日期格式作为合法输入格式。

举例

```
SELECT DATE_SUB('2015-01-15', 1) AS result FROM system.dual LIMIT 1;
```

result
2015-01-14

```
SET inceptor.udf.v2.enabled=true;
SELECT DATE_SUB('2018-07-31 12:12:12.111',1) AS result FROM system.dual LIMIT 1;
```

result
2018-07-30 12:12:12

8.9.8.20. DATE_FORMAT(<date>, <pattern>)

函数名	返回类型	描述
DATE_FORMAT(<date>, <pattern>)	STRING	把时间转化为指定的格式

说明

参数date必须是’yyyy-MM-dd HH:mm:ss’的字符或者Date、Timestamp格式，否则返回值为NULL；参数二必须是字符格式，指定了返回日期的格式。

举例

```
SELECT DATE_FORMAT('2015-01-15 00:00:00','yyyyMMdd') AS result FROM system.dual LIMIT 1;
```

result
20150115

8.9.8.21. STR_TO_DATE(<date>, <format>)

函数名	返回类型	描述
STR_TO_DATE(<date>, <format>)	DATE	根据指定的格式来解析日期并转化为标准格式

说明

参数date必须是字符或者Date、Timestamp格式，否则返回值为NULL；参数二必须是字符格式，指定了dateText的日期格式。最后返回的是标准的 **'yyyy-MM-dd'** 格式的日期。

举例

```
SELECT STR_TO_DATE('2015|01|15','yyyy|MM|dd') AS result FROM system.dual LIMIT 1;
```

result
2015-01-15

8.9.8.22. SYSDATE

函数名	返回类型	描述
SYSDATE	DATE	返回当前系统时间

说明

返回的时间是 **'yyyy-MM-dd HH:mm:ss'** 格式。注意使用时函数后没有括号。

举例

```
SELECT SYSDATE AS result FROM system.dual LIMIT 1;
```

result
2018-09-10 19:54:23

8.9.8.23. DAYOFWEEK(<date>)

函数	返回类型	描述
DAYOFWEEK(<date>)	INT	参数须是DATE类型，形式必须为：**'yyyy-MM-dd HH:mm:ss'** 或者 **'yyyy-MM-dd'**。返回日期在一周中是第几天（将周日算作第一天）。

```
SELECT DAYOFWEEK('2016-03-03') AS result FROM system.dual LIMIT 1;
```

result
5

8.9.8.24. DAYOFWEEK_ISO(<date>)

函数	返回类型	描述
DAYOFWEEK_ISO(<date>)	INT	参数须是DATE类型，形式必须为：**'yyyy-MM-dd HH:mm:ss'** 或者 **'yyyy-MM-dd'**。返回日期在一周中是第几天（将周一算作第一天）。

```
SELECT DAYOFWEEK_ISO('2016-03-03') AS result FROM system.dual LIMIT 1;
```

result
4

8.9.8.25. MILLISECOND(<date>)

函数	返回类型	描述
MILLISECOND(<date>)	INT	参数须是DATE类型，形式必须为 **'yyyy-MM-dd HH:mm:ss.xxx'** 或者 **'HH:mm:ss.xxx'**。返回参数中小数点后精确到毫秒的部分。小数点后最多能有三位数。

```
SELECT MILLISECOND('2009-07-30 12:58:59.847') AS result FROM system.dual LIMIT 1;
```

result
847

```
SELECT MILLISECOND('12:58:59.213') AS result FROM system.dual LIMIT 1;
```

result
213

8.9.8.26. MONTHS_BETWEEN

函数	返回类型	描述
MONTHS_BETWEEN (<end_date>, <start_date>)	DOUBLE	需要两个DATE类型的参数。返回以 **<end_date>** 结束，**<start_date>** 开始的一段时间中的月数。参数的形式必须为*'yyyyMM','yyyy-MM-dd HH:mm:ss'* 或者 **'yyyy-MM-dd'**。如果 **<end_date>** 在 **<start_date>** 之前，返回负值。

```
SELECT MONTHS_BETWEEN('2011-01-01','2010-10-15') AS result FROM system.dual LIMIT 1;
```

result
2.5483871

```
SELECT MONTHS_BETWEEN('2010-10-15','2011-01-01') AS result FROM system.dual LIMIT 1;
```

result
-2.5483871

8.9.8.27. LAST_DAY(<date>)

函数	返回类型	描述
LAST_DAY(<date>)	DATE	必须有一个DATE类型的参数，参数形式必须为：**yyyy-MM-dd HH:mm:ss**，**yyyy-MM-dd**，或者*yyyyMM*。返回参数所在月的最后一天的日期。

```
SELECT LAST_DAY('2003-03-15 01:22:33') AS result FROM system.dual LIMIT 1;
```

result
2003-03-31

8.9.8.28. DAYS(<date>)

函数	返回类型	描述
DAYS(<date>)	INT	需要一个DATE类型的参数 **<date>**，参数形式必须为 **yyyy-MM-dd**，**yyyy-MM-dd HH:mm:ss** 类型的日期。返回 **<date>** 和 **0001-01-02** 之间的天数。

```
SELECT DAYS('2016-03-05') AS result FROM system.dual LIMIT 1;
```

result
736028

8.9.8.29. TO_DAY_INTERVAL(<number>)

函数	返回类型	描述
TO_DAY_INTERVAL(<number>)	INTERVAL DAY	将参数转换成一个INTERVAL DAY类型返回。

```
SELECT DATE('2015-01-01') + TO_DAY_INTERVAL(2) AS result FROM system.dual LIMIT 1;
```

result
2015-01-03 00:00:00.0

8.9.8.30. TO_MONTH_INTERVAL(<number>)

函数	返回类型	描述
TO_MONTH_INTERVAL(<number>)	INTERVAL MONTH	将参数转换成一个INTERVAL MONTH类型返回。

```
SELECT DATE('2015-01-01') + TO_MONTH_INTERVAL(2) AS result FROM system.dual LIMIT 1;
```

result
2015-03-01

8.9.8.31. TO_YEAR_INTERVAL(<number>)

函数	返回类型	描述
TO_YEAR_INTERVAL(<number>)	INTERVAL YEAR	将参数转换成一个INTERVAL YEAR类型返回。

```
SELECT DATE('2015-01-01') + TO_YEAR_INTERVAL(2) AS result FROM system.dual LIMIT 1;
```

result
2017-01-01

8.9.8.32. TO_HOUR_INTERVAL(<number>)

函数	返回类型	描述
TO_HOUR_INTERVAL(<number>)	INTERVAL HOUR	将参数转换成一个INTERVAL HOUR类型返回。

```
SELECT DATE('2015-01-01') + TO_HOUR_INTERVAL(2) AS result FROM system.dual LIMIT 1;
```

result
2015-01-01 02:00:00.0

8.9.8.33. TO_MINUTE_INTERVAL(<number>)

函数	返回类型	描述
TO_MINUTE_INTERVAL(<number>)	INTERVAL MINUTE	将参数转换成一个INTERVAL MINUTE类型返回。

```
SELECT DATE('2015-01-01') + TO_MINUTE_INTERVAL(2) AS result FROM system.dual LIMIT 1;
```

result
2015-01-01 00:02:00.0

8.9.8.34. TO_SECOND_INTERVAL(<number>)

函数	返回类型	描述
TO_SECOND_INTERVAL(<number>)	INTERVAL SECOND	将参数转换成一个INTERVAL SECOND类型返回。

```
SELECT DATE('2015-01-01') + TO_SECOND_INTERVAL(2) AS result FROM system.dual LIMIT 1;
```

result
2015-01-01 00:00:02.0

8.9.8.35. TD_INT_TO_TIME(<HHmmss>)

函数	返回类型	描述
TD_INT_TO_TIME(<HHmmss>)	见描述	参数 **<HHmmss>** 必须是一个六位整数。将参数转换为时间返回。

```
SELECT TD_INT_TO_TIME(123456) AS result FROM system.dual LIMIT 1;
```

result
12:34:56

8.9.8.36. systime()

函数名	返回类型	描述
systime()	Date/Time	返回当前的系统时间，通常为所属服务器的系统时间。

使用示例:

```
SELECT systime() AS result;
```

返回示例:

```
+-----------+
|  result   |
+-----------+
| 16:27:46  |
+-----------+
```

8.9.8.37. SYSTIMESTAMP

函数	返回类型	描述
SYSTIMESTAMP	TIMESTAMP	返回当前系统TIMESTAMP。和 **current_timestamp** 同义。

```
SELECT SYSTIMESTAMP AS result FROM system.dual LIMIT 1;
```

result
2018-09-10 19:54:24.132

8.9.8.38. TRUNC

函数	返回类型	描述
TRUNC(<date>, <unit>)	DATE	返回给定日期 **date** 在指定单位 **unit** 内的开始值。例如，如果 **unit** 是 **MONTH**，则返回给定日期所在月的第一天。

说明

参数*unit*为日期单位，可以是：世纪（**CC**）、年（**YEAR/YYYY/YY**）、季度（**Q**）、月（**MONTH/MON/MM**）、日（**DAY/D/DY**）、时（**HH**）和分（**MI**）。参数*date*为日期，支持含有毫秒的值，如TRUNC('2018-07-31 12:12:12.111','DAY')。

在**inceptor.udf.v2.enabled**开关开启时：

返回结果默认包含不为零的时间部分。如要不显示时间部分，需要在hive_site.xml文件中设置**inceptor.date.show.time**为false；
输入值中的毫秒级数会被保存在返回的DATE类型结果中，可以通过CAST AS TIMESTAMP提取；
参数*date*值超出合法日期范围时会返回NULL。

此外，**inceptor.udf.v2.enabled**和*inceptor.support.english.date.format*中有一个为true时，参数*date*支持英文日期格式作为合法输入格式。

```
SELECT TRUNC('2016-10-22 11:22:33','YYYY') AS result FROM system.dual LIMIT 1;
```

result
2016-01-01

```
SELECT TRUNC('2016-10-22 11:22:33','MM') AS result FROM system.dual LIMIT 1;
```

result
2016-10-01

8.9.8.39. NEXT_DAY(<date>, '<dayofweek>')

函数	返回类型	描述
NEXT_DAY(<date>, '<dayofweek>')	DATE	需要两个参数：一个日期 **<date>** 和周中的一天 **<dayofweek>**（例如“星期天”、“Sunday”）。返回从 **<date>** 开始第一个 **<dayofweek>** 的日期。

下面例子返回 2015-12-25后的第一个星期天。

```
SELECT NEXT_DAY('2015-12-25','星期天') AS result FROM system.dual LIMIT 1
```

result
2015-12-27

8.9.8.40. FROM_UTC_TIMESTAMP

函数	返回类型	描述
FROM_UTC_TIMESTAMP(<date>, <timezone>)	DATE	需要两个参数：日期 **<date>** 和目标时区 **<timezone>**。该函数将 **<date>** 当做UTC时区的时间， 转化成 **<timezone>** 时区中的时间。

```
SELECT FROM_UTC_TIMESTAMP('1970-01-01 08:00:00','PST') AS result FROM system.dual LIMIT 1;
```

result
1970-01-01 00:00:00.0

8.9.8.41. TO_UTC_TIMESTAMP

函数	返回类型	描述
TO_UTC_TIMESTAMP(<date>, <timezone>)	DATE	需要两个参数：日期 **<date>** 和目标时区 **<timezone>**。该函数将 **<date>** 当做 **<timezone>** 时区的时间， 转化成UTC时区中的时间。

```
SELECT TO_UTC_TIMESTAMP('1970-01-01 00:00:00.0', 'PST') AS result FROM system.dual LIMIT 1;
```

result
1970-01-01 08:00:00.0

8.9.8.42. numtoyminterval(interval_number, year\|month)

函数	返回类型	描述
numtoyminterval(interval_number, year|month)	year-month	将<interval_number>转换为year-month interval类型。

```
SELECT numtoyminterval(12, 'month') AS RESULT FROM system.dual LIMIT 1;
```

result
1

8.9.8.43. numtodsinterval(interval_number, day\|hour\|minute\|second)

函数	返回类型	描述
numtodsinterval(interval_number, day|hour|minute|second)	day-time	将<interval_number>转换为day-time interval类型。

```
SELECT numtodsinterval(1, 'day') AS RESULT FROM system.dual;
```

result
1

8.9.8.44. timestampdiff(unit, start_timestamp, end_timestamp

函数名	返回类型	描述
timestampdiff(unit, start_timestamp, end_timestamp)	INT	计算两个日期/时间之间的时间差，并以指定的时间单位返回结果。

参数说明：

- **unit**：表示时间差的单位，取值：year、quarter、month、week、day、hour、minute、second。
- **start_timestamp**：表示开始时间的日期或时间戳。
- **end_timestamp**：表示结束时间的日期或时间戳。

使用示例：

计算两个时间的秒数差。

```
SELECT timestampdiff('second', '2023-01-01', '2023-01-01 19:37:40.419') AS result;
```

返回示例：

```
+---------+
| result  |
+---------+
| 70660   |
+---------+
```

8.9.9. 条件函数

表 6. IF

函数名	返回类型	描述
IF(booleantestCondition, T valueTrue, T valueFalseOrNull)	String	根据条件真假返回不同的值

说明

如果booleantestCondition为true，返回valueTrue，否则返回TvalueFalseOrNull。

举例

```
SELECT IF(true,'trans','age') AS result FROM system.dual LIMIT 1;
```

result
trans

表 7. COALESCE

函数名	返回类型	描述
COALESCE(a1, a2, …)	String	返回第一个非null的值

说明

返回第一个非null的值。

举例

```
SELECT COALESCE(null,'trans','age') AS result FROM system.dual LIMIT 1;
```

result
trans

表 8. CASE…WHEN

函数名	返回类型	描述
CASE a WHEN b THEN c [WHEN d THEN e] [ELSE f] END	相应参数类型	根据条件返回不同的值

说明

如果 a=b，返回 c，a=d 返回 e，以此类推，其他情况返回 f。CASE 到 END 中间可以有多个 WHEN x THEN y 的控制语句。

举例

```
SELECT CASE 't' WHEN 't' THEN 'trans' ELSE 'product' END AS result FROM system.dual;
```

result
trans

表 9. CASE WHEN

函数名	返回类型	描述
CASE WHEN a THEN b [WHEN c THEN d] [ELSE e] END	相应参数类型	根据参数真假返回不同值

说明

如果 a=true，返回 b，如果 c=true，返回 d，以此类推，其他情况返回 e。注意，*WHEN 后面必须跟布尔值*。CASE 到 END 中间可以有多个 WHEN x THEN y 的控制语句。

举例

```
SELECT CASE WHEN TRUE THEN 'trans' ELSE 'product' END AS result FROM system.dual;
```

result
trans

8.9.10. 字符串函数

8.9.10.1. 数学求值

8.9.10.1.1. ascii(A)

函数名	返回类型	描述
ascii(A)	Int	返回A第一个字符的ascii值。A可以是除了Binary之外的任何基本类型，返回A第一个字符的ascii值。如果输入为Binary类型则报错。

举例

```
SELECT ascii('20') AS result FROM system.dual;
```

result
50

8.9.10.1.2. format_number(number x, int d)

函数名	返回类型	描述
format_number(number x, int d)	Double	返回指定格式的number。A必须是数字类型，格式必须是Int、TinyInt、BigInt的一种。d表示number小数点后的位数。

举例

```
SELECT format_number(100,2) AS result FROM system.dual;
```

result
100.00

8.9.10.1.3. length(A) 或 characters(A)

函数名	返回类型	描述
length(A) 或 characters(A)	Int	返回A的长度，A可以是任何基本类型。

举例

```
SELECT length(1.1) AS result FROM system.dual;
```

result
3

```
SELECT characters(1.1) AS result FROM system.dual;
```

result
3

8.9.10.1.4. char_length(str)

函数名	返回类型	描述
char_length(str)	Int	返回输入字符串str或者对应字段长度，单位为字符。系统将一个多字节字符（例如，汉字）均当作一个字符处理。 此函数也可写作 **character_length()**，输入值为STRING或BINARY类型。

举例

```
SELECT char_length('我是汉字') AS result FROM system.dual;
```

result
4

8.9.10.1.5. octet_length(string\|binary)

函数名	返回类型	描述
octet_length(string|binary)	Int	返回字符串string以字节为单位的最大长度，或者对应字段byte数组的长度，对于一个汉字输入将返回3。输入值为STRING或BINARY类型。

举例

```
SELECT octet_length('我是汉字') AS result FROM system.dual;
```

result
12

8.9.10.1.6. greatest(value1, value2, value3, ….)

函数名	返回类型	描述
greatest(value1, value2, value3, ….)	value的共有类型	返回value中最大的值

说明

如果是数字类型比较，返回类型所有参数的公共类型，返回其中的最大值。如果是字符类型比较，则是逐位比较。

举例

```
SELECT greatest('10','9') AS result FROM system.dual LIMIT 1;
```

result
9

8.9.10.1.7. md5(<str>)

函数	返回类型	描述
md5(<str>)	STRING	需要一个STRING类型的参数 **<str>**。返回 **<str>** 的MD5哈希值。

举例

```
SELECT md5("a") AS result FROM system.dual LIMIT 1;
```

result
0cc175b9c0f1b6a831c399e269772661

8.9.10.1.8. crc32(str)

函数名	返回类型	描述
crc32(str)	String	生成str的32位循环冗余校验码多项式。通常用于可靠性较高的数据传输中，检查传输的数据是否完整。**str** 参数为STRING或BINARY类型。

举例

```
SELECT crc32("abc") AS result FROM system.dual;
```

result
891568578

8.9.10.2. 内容选取

8.9.10.2.1. concat(a,b,c……)

函数名	返回类型	描述
concat(a, b, c……)	String	合并输入的所有项。A可以是任何基本类型数据，连接多个字符串，合并为一个字符串，可以接受任意数量的输入字符串，返回字符串类型。

举例

```
SELECT concat(1,2,3) AS result FROM system.dual LIMIT 1;
```

result
123

8.9.10.2.2. context_ngrams (array <array<string>>,array<string>, int K, int pf)

函数名	返回类型	描述
context_ngrams (array <array<string>>, array<string>, int K, int pf)	array<struct<string, double>>	从给定一个字符串上下文中，返回字符串中出现频率在Top-K内，符合指定pattern的词汇。与sentences()函数一起使用时，返回分词结果中与ARRAY中指定的单词一起出现（包括顺序）频率最高的TOP-K结果。

举例

```
SET character.literal.as.string=true;
SELECT context_ngrams(sentences('hello A!hello B,hi A,hello B'),array('hello',null),3) FROM system
.dual;
```

_c0
[{"ngram":["B"],"estfrequency":2.0},{"ngram":["A"],"estfrequency":1.0}]

8.9.10.2.3. concat_ws(string SEP, string A, stringB…)

函数名	返回类型	描述
concat_ws(string SEP, string A, stringB…)	String	以SEP为分隔把A、B……合并起来。所有输入参数都必须是字符类型。

举例

```
SELECT concat_ws('|','1','2','3') AS result FROM system.dual;
```

result
1|2|3

8.9.10.2.4. lpad(string str, int len, string pad)

函数名	返回类型	描述
lpad(string str, int len, string pad)	String	用pad参数的值，在左方补全str为len长度。如果len的值小于str的长度，则返回从左截取len长度的字符。

举例

```
SELECT lpad('123456789',11,'a') AS result FROM system.dual;
```

result
aa123456789

8.9.10.2.5. ltrim(string A)

函数名	返回类型	描述
ltrim(string A)	String	去掉字符左侧的空格。

举例

```
SELECT ltrim(' bigdata ') AS result FROM system.dual LIMIT 1;
```

result
bigdata

8.9.10.2.6. ngrams(array<array<string>>, int N, int K, int pf)

函数名	返回类型	描述
ngrams(array<array<string>>, int N, int K, int pf)	Int	从给定一个字符串上下文中，返回从一组标记的句子的Top-K的N元语法词汇。

举例

```
SELECT ngrams(sentences(lower('tweet a')),2,100,1000) AS result FROM system.dual;
```

result
[{"ngram":["tweet","a"],"estfrequency":1.0}]

8.9.10.2.7. parse_url(string urlString, string part)

函数名	返回类型	描述
parse_url(string urlString, string part)	Int	返回URL中指定的部分。字符类型的part指定了返回URL的部分，可以是：'HOST', 'PATH', 'QUERY', 'REF', 'PROTOCOL', 'AUTHORITY', 'FILE', and 'USERINFO' 中的一个。

举例

```
SELECT parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'HOST') AS result FROM system
.dual LIMIT 1;
```

result
facebook.com

8.9.10.2.8. repeat(string str, int n)

函数名	返回类型	描述
repeat(string str, int n)	String	将str重复n次。str可以是除了Boolean之外的任何基本类型，返回n个str连在一起的字符串。

举例

```
SELECT repeat(1, 10) AS result FROM system.dual;
```

result
1111111111

8.9.10.2.9. replace(string input, string search, string replace)

函数名	返回类型	描述
replace(string input, string search, string replace)	字符串	将字符串中的所有指定子串替换进行替换。

参数说明：

- **string input：** 要替换的字符串。
- **string search：** 要被替换的子串。
- **string replace：** 替换成的新子串。

使用示例:

```
SELECT REPLACE('Hello, World!', 'World', 'Inceptor') AS result;
```

返回示例:

```
+-------------------+
|     result        |
+-------------------+
| Hello, Inceptor!  |
+-------------------+
```

8.9.10.2.10. rpad(string str, int len, string pad)

函数名	返回类型	描述
rpad(string str, int len, string pad)	String	用pad参数的值，在右方补全str为len长度。如果len的值小于str的长度，则返回从右截取len长度的字符。

举例

```
SELECT rpad('123456789',11,'a') AS result FROM system.dual;
```

result
123456789aa

8.9.10.2.11. rtrim(string A)

函数名	返回类型	描述
rtrim(string A)	String	去掉字符右侧的空格

举例

```
SELECT rtrim(' bigdata ') AS result FROM system.dual LIMIT 1;
```

result
bigdata

8.9.10.2.12. sentences(str, lang, country)

函数名	返回类型	描述
sentences(str, lang, country)	Array<Array<String>>	给一个包含自然语言句子的字符类型，在合适的位置分词，返回一些包含词组的数组。

举例

```
SELECT sentences('Hello there! How are you?') AS result FROM system.dual LIMIT 1;
```

result
[["Hello","there"],["How","are","you"]]

8.9.10.2.13. space(int n)

函数名	返回类型	描述
space(int n)	String	返回n个空格的字符

举例

```
SELECT space(5) AS result FROM system.dual LIMIT 1;
```

result

8.9.10.2.14. split(string str, string pat)

函数名	返回类型	描述
split(string str, string pat)	Array<string>	按指定分隔符分割字符。把字符串str按pat分隔，返回String数组。注意，在pat中，分隔符必须包含在中括号内。

举例

```
SELECT split('pro-duct','[-]') AS result FROM system.dual;
```

result
["pro","duct"]

8.9.10.2.15. substr(string A, int start [, length]) 或 substring(string A, int start [, length])

函数名	返回类型	描述
substr(string A, int start [, length]) 或 substring(string A, int start [, length])	String	返回A从start位置开始的子字符串，若指定length则返回从start开始，长度为length的子字符串。

举例

```
SELECT substr('bigdata',6,2) AS result FROM system.dual LIMIT 1;
```

result
ta

```
SELECT substring('bigdata',6,2) AS result FROM system.dual LIMIT 1;
```

result
ta

8.9.10.2.16. split_part(string, delimiter, field)

函数名	返回类型	描述
split_part(string, delimiter, field)	字符串	将字符串按照指定的分隔符进行分割，并返回指定位置的分割结果。

参数说明：

- **string：** 要分割的字符串。
- **delimiter：** 分隔符，用于将字符串进行分割。
- **field：** 指定要返回的分割结果的位置，从 1 开始计数。

使用示例：

使用逗号作为分隔符，将字符串 `apple,banana,cherry` 分割成三个部分，并返回第二个位置的分割结果，即 `banana`。

```
SELECT split_part('apple,banana,cherry', ',', 2) AS result;
```

返回示例：

```
+---------+
| result  |
+---------+
| banana  |
+---------+
```

8.9.10.2.17. substring_index(string, delimiter, N)

函数名	返回类型	描述
substring_index(string, delimiter, N)	String	按指定关键字获取子字符串

说明

返回string中第N次出现delimiter字符为止之前的子字符串。string和delimiter为STRING类型，N可以是NUMERIC类型。如果N为正，返回字符串是从左往右直到第N次出现delimiter为止之前的子字符串；反之，如果N为负，则返回字符串是从右到左直到第N次出现delimiter为止之前的子字符串。

举例

```
SELECT substring_index("http://www.example.com/",".",2) AS result FROM system.dual;
```

result
http://www.example

8.9.10.2.18. translate

函数名	返回类型	描述
translate(string|char|varchar input, string|char|varchar from, string|char|varchar to)	String	用指定字符替换目标

说明

translate方法作用于input string（第一个参数）的字符上，规则为出现在from_string中的字符用to_string中对应位置的字符来替换。如果to_string的长度比from_string短，那么from_string中多出来的位置的字符会在output中被移除。

举例

```
SELECT translate('abcdef', 'adc', '19') AS result FROM system.dual LIMIT 1;
```

result
1b9ef

8.9.10.2.19. trim(string s)

函数名	返回类型	描述
trim(string s)	String	去掉字符左右的空格

举例

```
SELECT trim(' bigdata ') AS result FROM system.dual LIMIT 1;
```

result
bigdata

8.9.10.2.20. oracle_instr(string1, string2 [, start_position [, nth_appearance]])

函数名	返回类型	描述
oracle_instr(string1, string2 [, start_position [, nth_appearance]])	String	返回子字符第一次出现的下标。

举例

```
SELECT oracle_instr('bigdata', 'data') AS result FROM system.dual LIMIT 1;
```

result
4

8.9.10.2.21. instrb(<str>, <substr>)

函数	返回类型	描述
instrb(<str>, <substr>)	INT	需要两个STRING类型的参数。该函数会返回子字符串 **<substr>** 第一次在字符串 **<str>** 中出现的位置。

举例

```
SELECT instrb('Facebook', 'boo') AS result FROM system.dual LIMIT 1;
```

result
5

8.9.10.2.22. elt(<n>, <str>[, <str>, …])

函数	返回类型	描述
elt(<n>, <str>[, <str>, …])	STRING	参数包括正整数 **<n>** 和一组（包含一个或多个）字符串 **<str>**。该函数返回参数中的第n个字符串。

举例

```
SELECT elt(1, 'face', 'book') AS result FROM system.dual LIMIT 1;
```

result
face

8.9.10.2.23. substrb(<str>, <n>[, <len>])

函数	返回类型	描述
substrb(<str>, <n>[, <len>])	STRING	两个必选参数：**<str>** 是一个STRING；**<n>** 是一个INT。该函数返回 **<str>** 中从第n个字符开始的子STRING，如果n是负数，则返回从倒数第n个字符开始的子STRING。可选参数 **<len>** 必须是一个正整数，用于指定返回的子STRING的长度。

举例

```
SELECT substrb('Facebook', 1) AS result FROM system.dual LIMIT 1;
```

result
Facebook

```
SELECT substrb('Facebook', -1) AS result FROM system.dual LIMIT 1;
```

result
k

```
SELECT substrb('Facebook', 1, 2) AS result FROM system.dual LIMIT 1;
```

result
Fa

8.9.10.2.24. substr_udf/substring_udf(string, int start, int len)

函数名	返回类型	描述
substr_udf(string, int start, int len)	字符串	从指定的起始位置开始，截取指定的长度的字符串。

参数说明：

- `string`：要截取子字符串的源字符串。
- `int start`：指定截取的起始位置，从 1 开始计数。
- `int len`：指定截取的长度。

使用示例：

从字符串 `Hello, World!` 中截取从第 8 个字符开始，长度为 5 的子字符串，即 `World`。

```
SELECT substr_udf('Hello, World!', 8, 5) AS result;
```

返回示例：

```
+---------+
| result  |
+---------+
| World   |
+---------+
```

8.9.10.2.25. substrb_udf(string, start, length)

函数名	返回类型	描述
substrb_udf(string, start, length)	字符串	从字符串中提取指定位置开始的指定长度的字节子串。

参数说明：

- **string**：要提取子串的字符串。
- **start**：起始位置，从 1 开始计数。
- **length**：要提取的字节长度。

使用示例：

从字符串 `Hello 世界` 中的第 7 个字节开始，提取 6 个字节长度的子串。

```
SELECT substrb_udf('Hello 世界', 7, 6) AS result;
```

返回示例:

```
+---------+
| result  |
+---------+
| 世界    |
+---------+
```

8.9.10.2.26. left(<str>, <n>)

函数	返回类型	描述
left(<str>, <n>)	STRING	需要两个参数：<str> 是字符串，<n> 是正整数。返回 <str> 中从左数长度为 <n> 的子字符串。

举例

```
SELECT left('bigdata', 5) AS result FROM system.dual LIMIT 1;
```

result
bigda

8.9.10.2.27. right(<str>, <n>)

函数	返回类型	描述
right(<str>, <n>)	STRING	需要两个参数：<str> 是字符串，<n> 是正整数。返回 <str> 中从右数长度为 <n> 的子字符串。

举例

```
SELECT right('bigdata', 4) AS result FROM system.dual LIMIT 1;
```

result
data

8.9.10.2.28. group_concat(<expr>,"<separator>")

函数	返回类型	描述
group_concat(<expr>,"<separator>")	string	结果是由<expr>指定的一组的非NULL字符串相拼接的STRING类型，分隔符通过<seperator>指定。

举例

查找user_info表中同一种acc_level类型的身份证号，并用逗号拼接返回结果。

```
SELECT acc_level, group_concat(citizen_id,",")
FROM user_info
GROUP BY acc_level;
```

acc_level	_c1
B	42**1
D	42*7,31*9
C	23*8,515,33***4
A	14*7,524,34***8
E	46**3

8.9.10.3. 类型/形式转换

8.9.10.3.1. base64(A)

函数名	返回类型	描述
base64(A)	Int	把Binary转化为Base64字符。A只可以是Binary类型的数据，返回A的Base64编码格式的值。

举例

```
SELECT base64(binary('1')) AS result FROM system.dual;
```

result
MQ==

8.9.10.3.2. lower(string A) 或 lcase(string A)

函数名	返回类型	描述
lower(string A) 或 lcase(string A)	String	把A中字母都转化为小写

举例

```
SELECT lower('A') AS result FROM system.dual;
```

result
a

```
SELECT lcase('A') AS result FROM system.dual;
```

result
a

8.9.10.3.3. printf(String format, Obj… args)

函数名	返回类型	描述
printf(String format, Obj… args)	String	返回指定输出格式的字符。

举例

```
SELECT printf("Hello World %d %s", 100, "days") AS result FROM system.dual LIMIT 1;
```

result
Hello World 100 days

8.9.10.3.4. reverse(A)

函数名	返回类型	描述
reverse(A)	String	把字符A转置

举例

```
SELECT reverse('bigdata') AS result FROM system.dual LIMIT 1;
```

result
atadgib

8.9.10.3.5. str_to_map(text[, delimiter1, delimiter2])

函数名	返回类型	描述
str_to_map(text[, delimiter1, delimiter2])	Map	把字符转化为map。第一个分隔符用于分隔不同组的K-V，第二个分隔符用于分隔每一组K-V。

举例

```
SELECT str_to_map('a-b=c-d','=','-') AS result FROM system.dual;
```

result
{"a":"b","c":"d"}

8.9.10.3.6. unbase64(string str)

函数名	返回类型	描述
unbase64(string str)	Binary	把base64格式转化为binary

举例

```
SELECT unbase64(base64(binary('bigdata'))) AS result FROM system.dual LIMIT 1;
```

result
bigdata

8.9.10.3.7. upper(string A) 或 ucase(string A)

函数名	返回类型	描述
upper(string A)	String	把字符中的字母都转化为大写，其他字符不变。

举例

```
SELECT upper('bigdata') AS result FROM system.dual LIMIT 1;
```

result
BIGDATA

```
SELECT ucase('bigdata') AS result FROM system.dual LIMIT 1;
```

result
BIGDATA

8.9.10.3.8. chr(A)

函数名	返回类型	描述
chr(A)	String	返回整型的char值。输入只能为INT，字符编码格式为内置的unicode。

举例

```
SELECT chr(123) AS result FROM system.dual;
```

result
{

8.9.10.3.9. to_number(value, format_mask)

函数名	返回类型	描述
to_number(value, format_mask)	Decimal	把string转化为number，按照指定的格式来格式化字符。参数 **format_mask** 用于指定输入值 **value** 的格式，便于系统识别。

举例

```
SELECT to_number('1239.1', '9999.0') AS result FROM system.dual LIMIT 1;
```

result
1239.1

8.9.10.3.10. trunc(date, format)

函数名	返回类型	描述
trunc(date, format)	String	返回日期的指定格式

说明

支持string类型的date字段按format返回特定的日期。format='yyyy'/'yy'，返回该年第一天的日期；format='mm'，返回该月第一天；format='dd'，得到日期；format='d'，得到该周第一天的日期；format='hh'，得到精确到小时的时间；format='mi'，得到精确到分钟的时间。

举例

```
SELECT trunc('2015-01-30 11:11:11','hh') AS result FROM system.dual LIMIT 1;
```

result
2015-01-30 11:00:00

8.9.10.3.11. to_char

函数名	返回类型	描述
to_char(date, pattern) / to_char(number [,format])	String	返回指定格式的日期(数字)字符

说明

输入的第一个参数为date的时候，把yyyy-MM-dd HH:mm:ss形式的string转化为输入pattern形式的string；第一个参数为数字时，则把数字转成string或按照一定format转string

举例

```
SELECT to_char('2015-01-30','yyyyMMdd') AS result FROM system.dual;
```

result
20150130

8.9.10.3.12. to_semiangle(string)

函数名	返回类型	描述
to_semiangle(string)	String	把全角字符转化为半角字符

举例

```
SELECT to_semiangle('bigdata') AS result FROM system.dual LIMIT 1;
```

result
bigdata

8.9.10.3.13. digits(<number>)

函数	返回类型	描述
digits(<number>)	STRING	将参数转换成STRING返回。参数可以是任何数值类型。如果参数是整数，会按类型将位数补足。

举例

```
SELECT digits(4) AS result FROM system.dual LIMIT 1;
```

result
0000000004

8.9.10.3.14. initcap(<str>)

函数	返回类型	描述
initcap(<str>)	STRING	将STRING类型参数 <str> 中所有词的首字母大写，其他字母小写。

举例

```
SELECT initcap('bIgData rockS!') AS result FROM system.dual LIMIT 1;
```

result
Bigdata Rocks!

8.9.10.3.15. decode_char (<binary>, <str>)

函数	返回类型	描述
decode_char (<binary>, <str>)	见描述	用 <str> 把 <binary> 转换为字符返回。需要两个函数：<binary> 是一串二进制字符，<str> 是编码方式。

举例

```
SELECT decode_char(binary('name'), 'UTF-8') AS result FROM system.dual LIMIT 1;
```

result
name

8.9.10.3.16. levenshtein(str1, str2)

函数	返回类型	描述
*levenshtein(str1, str2)	INT	返回字符串str1和str2之间的 Levenshtein 距离。 Levenshtein 距离又称编辑距离，指的是两个字符串之间，由一个转换成另一个所需的最少编辑操作次数，包括替换、插入、删除字符等操作。输入值为STRING类型。

举例

```
SELECT levenshtein("11564","1152") AS result FROM system.dual;
```

result
2

8.9.10.3.17. soundex(str)

函数	返回类型	描述
soundex(str)	STRING	将STRING类型文本串 **str** 转换为描述其语音的字母数字形式。soundex考虑了相似的发音字符和音节，比较的是字符串的发音而不是拼写。

举例

例如，数据库中存在一个名为“Lynn”的客户，而搜索时如果使用 **WHERE name='Lin'**，则不能返回想要的结果。但由于这两者的发音相似，将查询子句改写为 **WHERE soundex(name)=soundex('Lin')** 则能返回想要的结果：

```
SELECT soundex("Lin") AS result FROM system.dual;
```

result
L500

```
SELECT soundex("Lynn") AS result FROM system.dual;
```

result
L500

8.9.10.3.18. nlssort(string input, string collation)

函数名	返回类型	描述
nlssort(chars, nls_sort)	STRING	根据指定的字符集排序规则对输入字符串进行排序，并返回结果。

参数说明：

- **chars**：要排序的字符串，可以是表中的某一列。
- **nls_sort**：字符集排序规则，常用的规则例如：

  BINARY：二进制排序，按照字符串的二进制编码进行排序，区分大小写。

  BINARY_CI：二进制排序，不区分大小写。

使用示例：

```
SELECT * FROM course_info ORDER BY nlssort('course_id', 'NLS_SORT = BINARY');
```

返回示例：

```
+------------+-----------------------------------------+
| course_id  |                 course_name             |
+------------+-----------------------------------------+
| COM101     | Object-oriented Programming             |
| MUS101     | Appreciation of Western Classical Music |
| SCU100     | Sculpture                               |
| BIO101     | Fundamentals of Biology                 |
+------------+-----------------------------------------+
```

8.9.10.4. 加密解密

8.9.10.4.1. sha1(str)

函数	返回类型	描述
sha1(str)	STRING	返回字符串经过SHA-1算法计算的结果。参数 **str** 可以为为STRING或BINARY类型。

举例

```
SELECT sha("hello") AS result FROM system.dual;
```

result
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d

8.9.10.4.2. sha2(str,len)

函数	返回类型	描述
sha2(str, len)	STRING	返回字符串经过SHA-2算法计算的结果。 参数 **str** 可以为STRING或BINARY类型，参数 **len** 可以为224, 256(0), 384, 512，分别代表使用 SHA-224, SHA-256, SHA-384, 和 SHA-512 算法进行计算。

举例

```
SELECT sha2("hello", 0) AS result FROM system.dual;
```

result
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

8.9.10.4.3. aes_encrypt(input, key)

函数	返回类型	描述
aes_encrypt(input, key)	STRING	使用AES（Advanced Encryption Standard）算法加密字符串 **input**。**input** 可以是STRING和BINARY类型。 **key** 表示值的长度，有3个取值：128，192，256，单位为bit。默认只支持128bit的key值（即16个字符长度的字符串），对于192和256bit，需要有 Java Cryptography Extension (JCE) 无限强度权限策略文件的支持才可以使用。

举例

```
SELECT base64(aes_encrypt('ABC', '1234567890123456')) AS result FROM system.dual;
```

result
y6Ss+zCYObpCbgfWfyNWTw==

8.9.10.4.4. aes_decrypt(input, key)

函数	返回类型	描述
*aes_decrypt(input, key)	STRING	使用AES（Advanced Encryption Standard）算法对加密字符串 **input** 进行解密。**input** 可以是STRING和BINARY类型。 **key** 表示值的长度，有3个取值：128，192，256，单位为bit。默认只支持128bit的key值（即16个字符长度的字符串），对于192和256bit，需要有 Java Cryptography Extension (JCE) 无限强度权限策略文件的支持才可以使用。

举例

```
SELECT aes_decrypt(unbase64('y6Ss+zCYObpCbgfWfyNWTw==''), '1234567890123456') AS result FROM system
.dual;
```

result
ABC

8.9.10.5. 正则表达式相关

8.9.10.5.1. regexp_extract(string subject, string pattern, int index)

函数名	返回类型	描述
regexp_extract(string subject, string pattern, int index)	String	通过下标返回正则表达式指定的部分。

举例

```
SELECT regexp_extract('foothebar', 'foo(.*?)(bar)', 0) AS result FROM system.dual;
```

result
foothebar

8.9.10.5.2. regexp_replace

函数名	返回类型	描述
regexp_replace(string INITIAL_STRING, string PATTERN, string REPLACEMENT[, int POSITION[, int OCCURRENCE]])	Int	替换符合正则表达式的部分。

说明

把INITIAL_STRING中符合正则表达式PATTERN的部分替换为REPLACEMENT。可选项POSITION为INITIAL_STRING中开始搜索的位置，默认为1；OCCURRENCE为指定对第几次出现的匹配项进行替换，为0时表示替换所有出现的匹配项。POSITION和OCCURRENCE可以是所有能够转化成INT类型的数据。

举例

```
SELECT regexp_replace('foothebarfoobar', 'foo(.*?)(bar)', 'num', 1, 2) AS result FROM system.dual
LIMIT 1;
```

result
foothebarnum

8.9.10.5.3. regexp_like(string subject, string pattern)

函数名	返回类型	描述
regexp_like(string subject, string pattern)	布尔值	检查字符串 subject 是否匹配正则表达式 pattern。如果匹配，则返回 true，否则返回 false。

说明：

- string subject：要检查的字符串。
- string pattern：要匹配的正则表达式。

使用示例：

```
SELECT regexp_like('Hello, World!', 'H*') AS result;
```

返回示例：

```
+--------+
| result |
+--------+
| true   |
+--------+
```

8.9.10.5.4. regex_substr

函数	返回类型	描述
regex_substr (<source>, <regex> [, <pos> [, <occurrence> [, <match_parameter>]]])	见描述	该函数需要至少两个函数：一段文本 <source> 和一个正则表达式 <regex>。该函数会将 <source> 中和 <regex> 第一个发生匹配的子字符串返回。<pos> 是一个正整数，用于指定从 <source> 中的第几个字符开始匹配。 <occurrence> 是一个正整数，用于指定返回第几个发生匹配的子字符串。<match_parameter> 用于指定匹配方式，i：大小写不敏感；c：大小写敏感；n 将 .（通配符）和换行符匹配，默认为不匹配；m 将 <source> 作为多行来处理，将 ^ 作为行开头，$ 作为行结尾； x 忽略空格。

举例

```
SELECT regex_substr('Road Guiping, Xuhui District, SH',',[^,]+,') AS result FROM system.dual LIMIT
1;
```

result
, Xuhui District,

8.9.10.6. 位置相关

8.9.10.6.1. find_in_set(string str, string strList)

函数名	返回类型	描述
find_in_set(string str, string strList)	Int	返回字符串str第一次在strlist出现的位置。 参数strList以','分隔。函数返回字符串str第一次在strlist出现的位置。如果任一参数为NULL,返回NULL；如果第一个参数包含逗号，返回0。

举例

```
SELECT find_in_set('4','1,2,3,4') AS result FROM system.dual;
```

result
4

8.9.10.6.2. get_json_object

函数名	返回类型	描述
get_json_object(string json_string, string path)	Int	从指定地址获取json对象。

举例

```
SELECT a.timestamp, get_json_object(a.appevents, '$.eventid') AS id,
get_json_object(a.appevents, '$.eventname') AS name FROM log a;
```

timestamp	id	name
20160612	1	thunderstorm
20160712	2	sunny

8.9.10.6.3. in_file(str,filepath)

函数名	返回类型	描述
in_file(str,filepath)	Boolean	判断指定字符是否在文件中出现。str为String类型，filepath必须是存在的文件路径。该函数仅在Cluster Mode下生效，因此需要关闭开关ngmr.dd.local.mode.auto，即set ngmr.dd.local.mode.auto=false。

举例

```
set ngmr.dd.local.mode.auto=false;
SELECT in_file('123','/etc/hosts') AS result FROM system.dual;
```

result
false

8.9.10.6.4. instr(string str, string substr)

函数名	返回类型	描述
instr(string str, string substr)	Int	返回substr在str中出现的位置，不曾出现则返回0。

举例

```
SELECT instr('001122','1') AS result FROM system.dual;
```

result
3

8.9.10.6.5. locate(string substr, string str[, int pos])

函数名	返回类型	描述
locate(string substr, string str[, int pos])	Int	定位substr第一次在str中出现的位置。如果指定了pos值，则返回从str的第pos个位置之后，第一次出现substr的位置。如果不存在，则返回0。

举例

```
SELECT locate('123','456789123') AS result FROM system.dual;
```

result
7

8.9.10.6.6. oracle_instrb

函数	返回类型	描述
oracle_instrb(<str>, <substr>, [<start_position>, [<nth_appearance>]]))	INT	需要两个必选参数 **<str>** 和 **<substr>**。当不提供可选参数 **<start_position>** 和 **<nth_appearance>** 时，返回 **<substr>** 在 **<str>** 中第一次出现的位置（和函数 **instrb** 相同）。可选参数 **<start_position>** 指定从一个起始位置开始， **<substr>** 在 **<str>** 中第一次出现的位置。加上可选参数 **<nth_apperance>**，则返回 **<substr>** 在 **<str>** 中第n次出现的位置。

举例

```
SELECT oracle_instrb('ababab','ab') AS result FROM system.dual LIMIT 1;
```

result
1

```
SELECT oracle_instrb('ababab','ab',2) AS result FROM system.dual LIMIT 1;
```

result
3

```
SELECT oracle_instrb('ababab','ab',1,3) AS result FROM system.dual LIMIT 1;
```

result
5

8.9.10.6.7. position(string, substring)

函数名	返回类型	描述
position(string, substring)	整型	返回指定子字符串在目标字符串中首次出现的位置，如果未找到则返回 0。

参数说明：

- **string：** 目标字符串。
- **substring：** 要查找的子字符串。

使用示例：

在字符串 `Hello, world!` 中查找子字符串 `world` 的位置。

```
SELECT position('Hello, world!', 'world') AS result;
```

返回示例：

```
+---------+
| result  |
+---------+
| 8       |
+---------+
```

注意事项：

- `position` 函数对大小写敏感。如果需要进行大小写不敏感的子字符串查找，可以将目标字符串和子字符串都转换为小写（或大写）后再进行比较。
- 如果子字符串在目标字符串中多次出现，`position` 函数只会返回第一次出现的位置。

8.9.10.7. 条件判断

8.9.10.7.1. nvl(eExpr1, eExpr2)

函数名	返回类型	描述
nvl(eExpr1, eExpr2)	String	替换null为其他字符。eExpr1为空时返回eExpr2的值，仅支持基本类型（包括string）。

举例

```
SELECT nvl(null,'bigdata') AS result FROM system.dual LIMIT 1;
```

result
bigdata

8.9.10.7.2. nvl2(string1, value_if_not_null, value_if_null)

函数名	返回类型	描述
nvl2(string1, value_if_not_null, value_if_null)	String	根据字符是否为null返回不同的值。若string1为null，返回value_if_null，否则返回value_if_not_null。

举例

```
SELECT nvl2(null,'trans','product') AS result FROM system.dual LIMIT 1;
```

result
product

8.9.10.7.3. lnnvl(condition)

函数名	返回类型	描述
lnnvl(condition)	String	处理判断条件中有null值出现的情况

说明

condition是一个表达式，所得值是Boolean类型。condition为null时，返回true；condition为true时返回false；condition为false时返回true。

举例

```
SELECT lnnvl(true) AS result1, lnnvl(false) AS result2, lnnvl(null) AS result3
FROM system.dual LIMIT 1;
```

result1	result2	result3
false	true	true

8.9.10.7.4. decode

函数名	返回类型	描述
decode(expr, search1, result1[, search2, result2]…[, default])	String	根据字符是否相等选择不同返回值

说明

返回当expr=search1的时候返回result1，expr=search2的时候返回result2，否则返回default。**inceptor.udf.compatible.with.oracle** 参数来控制兼容性。当此开关为 **true** 时：

如果(search1, result1)是数值类型，则将expr和所有search值都转换为所有result值（包括default）中优先级（BYTE<SHORT<INT<LONG<DECIMAL<FLOAT<DOUBLE）最高的数值类型，并进行比较，返回值也转换为此类型；
否则，将expr和所有search值都转换为search1的类型进行比较，返回值也转换为result1的类型。如果search1或result1为CHAR类型，则使用VARCHAR2类型代替，且其长度为所有result值中最长的一个数据类型的长度。
当expr为NULL时，可以和值为NULL的search匹配成功。

举例

```
SELECT decode(1,1,'trans','product') AS result FROM system.dual;
```

result
trans

8.9.11. XML函数

- xmlforest

函数	返回类型	描述
xmlforest(<col>, [<col>, …])	XML文本	参数必须是列名。对于每行记录，将参数中各列的字段转换为一段XML文本，然后将该行记录产生的所有XML文本连在一起返回。

举例

```
SELECT xmlforest(name, age) a FROM student_age ORDER BY a LIMIT 1;
```

a
<name>Han Meimei</name><age>20</age>

- xmlcolattval(<col>, [<col>,…])

函数	返回类型	描述
xmlcolattval(<col>, [<col>,…])	XML文本	参数必须是列名。对于每行记录，将参数中的各列的字段转换为一段column attribute为列名的XML文本，然后将该行记录产生的所有XML文本连在一起返回。

举例

```
SELECT xmlcolattval(name, age) a FROM student_age ORDER BY a LIMIT 1;
```

a
<column name="Han Meimei"/><column age="20"/>

- xpath(<xml>,<xpath>)

函数	返回类型	描述
xpath(<xml>,<xpath>)	ARRAY<STRING>	需要两个参数：一段XML文本 **<xml>** 和一个 **<xpath>** 表达式。该函数会将所有 **<xml>** 中处在 **<xpath>** 节点下的值作为一个字符串ARRAY返回。

举例

```
SELECT xpath('<a><b>b1</b><b>b2</b><b>b3</b><c>c1</c><c>c2</c></a>', 'a/text()') AS result FROM
system.dual LIMIT 1;
```

result
[]

```
SELECT xpath('<a><b>b1</b><b>b2</b><b>b3</b><c>c1</c><c>c2</c></a>', 'a/b/text()') AS result FROM
system.dual LIMIT 1;
```

result
["b1","b2","b3"]

```
SELECT xpath('<a><b>b1</b><b>b2</b><b>b3</b><c>c1</c><c>c2</c></a>', 'a/c/text()') AS result FROM
system.dual LIMIT 1;
```

result
["c1","c2"]

```
SELECT xpath('<a><b>b1</b><b>b2</b><b>b3</b><c>c1</c><c>c2</c></a>',xpath_expr) values FROM
xpath_demo ORDER BY values;
```

values
[]
["b1","b2","b3"]
["c1","c2"]

- xmlelement(<str1>, <str2>)

函数	返回类型	描述
xmlelement(<str1>, <str2>)	XML文本	返回 **<str1>str2</str1>** 形式的XML文本。

```
SELECT xmlelement('a','b') AS result FROM system.dual LIMIT 1;
```

result
<a>b

- xmlsequence(<xml>)

函数	返回类型	描述
xmlsequence(<xml>)	XML文本	返回 **<xml>** 中最高一层的节点。

```
SELECT xmlsequence('<Employee><id>36</id><name>Haines</name></Employee>') AS result FROM system.dual
LIMIT 1;
```

result
<id>36</id><name>Haines</name>

- xpath_string (<xml>,<xpath>)

函数	返回类型	描述
xpath_string (<xml>,<xpath>)	STRING	需要两个参数：**<xml>** 为一段XML文本，**<xpath>** 为一个xpath。返回 **<xml>** 文本中第一个匹配 **<xpath>** 的节点中的内容。

```
SELECT xpath_string('<a><b>b</b><c>cc</c></a>','a/c') AS result FROM system.dual LIMIT 1;
```

result
cc

- xpath_boolean (<xml>,<xpath> [=<val>])

函数	返回类型	描述
xpath_boolean (<xml>,<xpath> [=<val>])	BOOLEAN	需要两个参数：**<xml>** 为一段XML文本，**<xpath>** 为一个xpath。判断 **<xml>** 中是否有匹配 **<xpath>** 的节点。如果加上 **=<val>** 选项，则判断 **<xml>** 中匹配 **<xpath>** 的节点中的内容是否有 **<val>**。

```
SELECT xpath_boolean('<a><b>0</b></a>','a/b') AS result FROM system.dual LIMIT 1;
```

result
true

```
SELECT xpath_boolean('<a><b>1</b></a>','a/b=0') AS result FROM system.dual LIMIT 1;
```

result
false

- xpath_number (<xml>,<expr(xpath)>) 或 xpath_double (<xml>,<expr(xpath)>)

函数	返回类型	描述
xpath_number (<xml>,<expr(xpath)>) 或 xpath_double (<xml>,<expr(xpath)>)	DOUBLE	需要两个参数：**<xml>** 为一段XML文本，**<expr(xpath)>** 为一个含有xpath的表达式。对 **<xml>** 求 **<expr(xpath)>** 的值，将值作为DOUBLE返回。与函数 **xpath_double** 为同义函数。

```
SELECT xpath_number('<a><b>1</b><b>2</b></a>','sum(a/b)') AS result FROM system.dual LIMIT 1;
```

result
3.0

- xpath_float (<xml>,<expr(xpath)>)

函数	返回类型	描述
xpath_float (<xml>,<expr(xpath)>)	FLOAT	需要两个参数：**<xml>** 为一段XML文本，**<expr(xpath)>** 为一个含有xpath的表达式。对 **<xml>** 求 **<expr(xpath)>** 的值，将值作为FLOAT返回。

```
SELECT xpath_float('<a><b>1</b><b>2</b></a>','sum(a/b)') AS result FROM system.dual LIMIT 1;
```

result
3.0

- xpath_long (<xml>,<expr(xpath)>)

函数	返回类型	描述
xpath_long (<xml>,<expr(xpath)>)	LONG	需要两个参数：**<xml>** 为一段XML文本，**<expr(xpath)>** 为一个含有xpath的表达式。对 **<xml>** 求 **<expr(xpath)>** 的值，将值作为LONG返回。

```
SELECT xpath_long('<a><b>1</b><b>2</b></a>','sum(a/b)') AS result FROM system.dual LIMIT 1;
```

result
3

- xpath_int (<xml>,<expr(xpath)>)

函数	返回类型	描述
xpath_int (<xml>,<expr(xpath)>)	INT	需要两个参数：**<xml>** 为一段XML文本，**<expr(xpath)>** 为一个含有xpath的表达式。对 **<xml>** 求 **<expr(xpath)>** 的值，将值作为INT返回。

```
SELECT xpath_int('<a><b>1</b><b>2</b></a>','sum(a/b)') AS result FROM system.dual LIMIT 1;
```

result
3

- xpath_short (<xml>,<expr(xpath)>)

函数	返回类型	描述
xpath_short (<xml>,<expr(xpath)>)	SHORT	需要两个参数：**<xml>** 为一段XML文本，**<expr(xpath)>** 为一个含有xpath的表达式。对 **<xml>** 求 **<expr(xpath)>** 的值，将值作为SHORT返回。

```
SELECT xpath_short('<a><b>1</b><b>2</b></a>','sum(a/b)') AS result FROM system.dual LIMIT 1;
```

result
3

8.9.12. 复杂类型函数

- map(key1, value1, key2, value2, …)

操作符	返回类型	描述
map(key1, value1, key2, value2, …)	map 类型	创建键值对

说明

通过指定的键/值对，创建一个 map 类型。指定的所有 key 必须有公共类型，所有的 value 也必须有公共类型，得到的 map 的 key 类型即所有 key 的公共类型。map(int,string,double,string) 得到的类型是 map<double,string>，而 map(int,string,boolean,string) 是不合法的,因为在这里 Int 和 Boolean 是没有公共类型的。

举例

```
SELECT map(1,2,'3',4) AS result FROM system.dual LIMIT 1;
```

result
{"1":2,"3":4}

- struct(val1, val2, val3, …)

操作符	返回类型	描述
struct(val1, val2, val3, …)	struct类型	创建 struct 结构体

说明

val[1-n] 是相同类型的变量，或者是有公共类型的。得到的是 val[1-n] 的公共类型的 struct 结构体。

举例

```
SELECT struct(1,2,3) AS result FROM system.dual LIMIT 1;
```

result
{"col1":1,"col2":2,"col3":3}

- named_struct(name1, val1, name2, val2, …)

操作符	返回类型	描述
named_struct(name1, val1, name2, val2, …)	struct类型	创建自定义字段名称的 struct 结构体

说明

val[1-n] 是相同类型的变量，或者是有公共类型的，name[1-n] 是字符类型。得到的是以 name[1-n]为名字，以 val[1-n] 为对应值的的 struct 结构体，每个名字有自己的类型。比如 named_struct ('name1',1,'name2','1') 得到的是

举例

```
SELECT named_struct('name1',100,'name2','200') AS mystruct
FROM system.dual LIMIT 1;
```

mystruct
{"name1":100,"name2":"200"}

- array(val1, val2, …)

操作符	返回类型	描述
array(val1, val2, …)	Array类型	通过指定的元素，创建一个数组。Array 的类型是所有元素的公共类型。

说明

由于向该函数传递的字符串要求必须是 STRING 类型，为防止系统将传递的字符串参数识别为 CHAR 型，必须要打开开关 character.literal.as.string，强制将字符串识别为 STRING 类型。

举例

```
set character.literal.as.string=true;
SELECT array(1,'1',1.1) AS result FROM system.dual LIMIT 1;
```

result
["1","1","1.1"]

- A[n]

操作符	返回类型	描述
A[n]	数组的类型	返回数组的第n个元素

说明

只要子查询返回了至少一列的值，返回 true，反之返回 false。

举例

```
SELECT array(1,2,3)[0] AS result FROM system.dual ;
```

result
1

- M[key]

操作符	返回类型	描述
M[key]	Map 元素中的类型	返回 map 中对应 key 的 value

说明

返回关键值对应的值，例如 mapM 为 {‘f’ → ‘foo’, ‘b’ → ‘bar’, ‘all’ → ‘foobar’}，则M[‘all’] 返回’foobar’。

举例

```
SELECT map('k1',100,'k2',200)['k1'] AS result FROM system.dual ;
```

result
100

- S.x

操作符	返回类型	描述
S.x	Struct 的类型	返回 Struct 的第 x 个元素

说明

返回结构 x 字符串在结构 S 中的存储位置。如 foobar \{int foo, int bar\} foobar.foo的领域中存储的整数。

由于向该函数传递的字符串要求必须是 STRING 类型，为防止系统将传递的字符串参数识别为 CHAR 型，必须要打开开关 character.literal.as.string，强制将字符串识别为 STRING 类型。

举例

```
SELECT named_struct('name1',100,'name2','200').name1 FROM system.dual LIMIT 1;
```

name1
100

- size(Map<K.V>) ; size(Array<T>)

操作符	返回类型	描述
size(Map<K.V>) ; size(Array<T>)	Int	返回数组或者 map 中的元素数量。

举例

```
SELECT size(array('name','age','sex')) AS result FROM system.dual;
```

result
3

- sort_array(array)

函数	返回类型	描述
sort_array(array)	array类型	对输入的 array 进行排序输出，默认按照从小到大排列。

举例

```
SELECT sort_array(array(3,1,4,5,2)) AS result FROM system.dual;
```

result
[1,2,3,4,5]

- sort_array_by

函数	返回类型	描述
sort_array_by(array(struct(<obj>), struct(<obj>),…),'<f1>','<f2>',… ,'ASC'|'DESC')	ARRAY	返回以指定排序方式排序后的Array

说明

第一个参数必须为 ARRAY 类型，且 array 中的元素必须为 STRUCT 类型。'<f1>','<f2>',…为struct(<obj>)中的元素，且数量不能超过struct中定义的数量。

'ASC'|'DESC' 指定排序方式，ASC 为升序排列，DESC 为降序排列。默认为升序。

举例

```
SELECT sort_array_by(array(struct('g',100),struct('b',200),struct('b',100)),'col1','col2','ASC') AS
result FROM system.dual;
```

result
[{"col1":"b","col2":100},{"col1":"b","col2":200},{"col1":"g","col2":100}]

- size(<map_or_array>)

函数	返回类型	描述
size(<map_or_array>)	int	只能有一个参数。参数必须是 MAP 或 ARRAY 类型。

举例

```
SELECT SIZE(MAP(1,2)) mapsize, SIZE(ARRAY(1,2)) arraysize FROM system.dual LIMIT 1;
```

mapsize	arraysize
1	2

- index(<array>, <n>)

函数	返回类型	描述
index(<array>, <n>)	—	需要两个参数，第一个参数 **<array>** 必须是 ARRAY。第二个参数 **<n>** 必须是非负整数。该函数返回 **<array>** 中的第 n 个元素。

举例

```
SELECT index(array(1,2,3),0) AS result FROM system.dual LIMIT 1;
```

result
1

```
SELECT index(absent_date,0) AS result FROM employee_absence;
```

result
2014-01-04
2014-02-14

- array_contains (<array>, <value>)

函数	返回类型	描述
array_contains (<array>, <value>)	BOOLEAN	需要两个参数，第一个参数 **<array>** 是 ARRAY。该函数检查第二个参数 **<value>** 是否是 **<array>** 中的一个元素。如果是，返回 TRUE。否则返回 FALSE。

举例

```
SELECT array_contains(array(1, 2, 3), 2) AS result FROM system.dual LIMIT 1;
```

result
true

- map_keys(<map>)

函数	返回类型	描述
map_keys(<map>)	ARRAY	需要一个 MAP 类型的参数。将参数中的所有键放在一个 ARRAY 中返回。

举例

```
SELECT map_keys(map(1,'a',2,'b')) AS result FROM system.dual LIMIT 1;
```

result
[1,2]

- map_values(<map>)

函数	返回类型	描述
map_values(<map>)	ARRAY	需要一个 MAP 类型的参数。将参数中的所有值放在一个 ARRAY 中返回。

举例

```
SELECT map_values(map(1,'a',2,'b')) AS result FROM system.dual LIMIT 1;
```

result
["a","b"]

8.9.13. 表生成函数

表生成函数（User-Defined Table-Generating Functions），简称为UDTF，是用于解决将一行输出为多行需求的一类函数。

下面是Inceptor支持的各种UDTF：

- explode

函数名	返回类型	描述
explode(ARRAY) ; explode(MAP)	多行数据	把数组或者map中的元素生成表

说明

输入Array时，array中每个元素为生成表的一条记录；输入Map时，每个K-V对为一条记录。

举例

```
SELECT explode(array('name','age','sex')) FROM system.dual;
```

col
name
age
sex

- inline

函数名	返回类型	描述
inline(ARRAY<STRUCT[,STRUCT]>)	多行数据	生成表，每行一个struct（结构体）

说明

生成表，每一个Struct（结构体）作为一行。

举例

```
SELECT inline(array(struct('name','age','sex')))
FROM system.dual;
```

col1	col2	col3
name	age	sex

- json_tuple

函数名	返回类型	描述
json_tuple(ARRAY)	多行数据	用json中的元组生成表

说明

返回值的tuple。这是 'get_json_object' 的高效版。

举例

支持向该函数传递string、char、varchar、varchar2类型

```
set character.literal.as.string=true;
SELECT json_tuple('{"firstName":"Brett","lastName":"McLaughlin","email":"aaaa"}', 'firstName',
'email') FROM system.dual;
```

c0	c1
Brett	aaaa

- parse_url_tuple

函数名	返回类型	描述
parse_url_tuple(url, partname1, partname2, …, partnameN)	多行数据	根据url生成多行数据

说明

输入Array时，array中每个元素为生成表的一条记录；输入Map时，每个K-V对为一条记录。

举例

由于向该函数传递的字符串要求必须是STRING类型，为防止系统将传递的字符串参数识别为CHAR型，必须要打开开关character.literal.as.string，强制将字符串识别为STRING类型。

```
set character.literal.as.string=true;
SELECT parse_url_tuple('http://www.example.com/news/detail?id=38', 'HOST', 'PATH', 'QUERY',
'PROTOCOL', 'FILE') as (ip, ho, pa, pr, fi) FROM system.dual a;
```

ip	ho	pa	pr	fi
www.example.com	/news/detail	id=38	http	/news/detail?id=38

- stack

函数名	返回类型	描述
stack(n, col1, col2…)	多行数据	把每个col分割为n份，生成为n行数据

说明

当n大于1的时候，col必须为集合类型，比如Array。

举例

由于向该函数传递的字符串要求必须是STRING类型，为防止系统将传递的字符串参数识别为CHAR型，必须要打开开关character.literal.as.string，强制将字符串识别为STRING类型。

```
set character.literal.as.string=true;
SELECT stack(2, array('name',1), array('age',2)) FROM system.dual;
```

col0
["name","1"]
["age","2"]

8.9.13.1. LATERRAL VIEW函数

介绍

如果通过SELECT语句查询一个UDTF的结果，那么该SELECT从句中就不能再出现其他对象。 例如，给定表table_array，它的schema和包含的数据如下：

```
DESC table_array;
```

col_name	data_type	default_value	not_null	unique	comment
id	char(2,ORACLE)	NULL	No	No	
values	array<int>	NULL	No	No	

```
SELECT * FROM table_array;
```

id	values
a	[2,2,3]
b	[2,3,4]
c	[3,4,5]

调用 **explode** 函数对values列进行展开，并返回对应id号，即执行下面这条语句，将会出现如下报错：

```
SELECT id, explode(values) FROM table_array;
```

```
COMPILE FAILED: Semantic error: [Error 10081] UDTF's are not supported outside the SELECT clause,
nor nested in expressions
```

为了实现在返回UDTF结果的同时查询其他对象，须引用关键字 **LATERAL VIEW** 。**LATERAL VIEW** 主要用于和UDTF联用，将UDTF生成的结果放到虚拟表中，然后使该虚拟表和输入行进行JOIN，以达到连接UDTF外的SELECT对象的目的。

语法和示例

LATERAL VIEW 的语法为：

```
-- tableAlias指代存放UDTF结果的虚拟表的名称
SELECT <selectexpr1>, <columnAlias1> [, <columnAlias2>, ...]
-- columnAlias指代虚拟表中列的名称，在SELECT中必须以该名称访问被展开结果
FROM baseTable ATERAL VIEW udtf(expression) <tableAlias>
AS <columnAlias1> [, <columnAlias2>, ...]
```

因此，为了展开查询table_array中的values并返回对应id，正确的语句应该为：

```
SELECT id, a.value
FROM table_array LATERAL VIEW explode(values) a AS value;
```

id	value
a	2
a	2
a	3
b	2
b	3
b	4
c	3
c	4
c	5

其中字段id来自于表table_array本身，value来自于explode(values)结果的虚拟表a。

下面再提供一个例子：

如果另外有一张表table_array2，它包含的两个字段都是ARRAY类型，其结构和数据如下所示：

```
DESC table_array2;
```

col_name	data_type	default_value	not_null	unique	comment
ids	array<char(2,ORACLE)>	NULL	No	No	
values	array<int>	NULL	No	No	

```
SELECT * FROM table_array2;
```

ids	values
["a ","b "]	[1,2,3]
["c ","d "]	[4,5,6]
["e "]	[5,6]

如果要求铺展显示table_array2的各个id和value值，正确的实现语句和结果应为：

```
SELECT id, value
FROM table_array2
LATERAL VIEW explode(ids) a AS id
LATERAL VIEW explode(values) b AS value;
```

id	value
a	1
a	2
a	3
b	1
b	2
b	3
c	4
c	5
c	6
d	4
d	5
d	6
e	5
e	6

其中字段id来自于explode(ids)对应的虚拟表a，value来自于explode(values)对应的虚拟表b。

8.9.14. 聚合/分析函数

8.9.14.1. avg

函数语法	返回类型	描述
avg(expr)	Double	平均值

说明

返回某常数或者某一列的平均值。

举例

```
SELECT avg(price) AS result FROM somedata;
```

result
100000.0

8.9.14.2. collect_set

函数语法	返回类型	描述
collect_set(expr)	Array	从输入值中创建一个集合，其中包含不同的值，从而实现去重的效果。

参数说明：

- **expr**：用于计算的表达式。

使用示例：

从一个表中的多列中创建一个包含不同值的集合。

```
SELECT
  collect_set(amount) AS result
FROM
  transactions;
```

返回示例：

```
+-------------------------------------------------------+
|                        result                         |
+-------------------------------------------------------+
| [800,400,500,100,1300,900,200,1000,600,1100,300,700]  |
+-------------------------------------------------------+
```

8.9.14.3. compute_stats

函数语法	返回类型	描述
compute_stats(expr)	Struct	返回一组基本类型值的统计摘要。

参数说明：

- expr：指定用于统计的表达式。

使用示例：

查看 **transactions** 表中，成交单价（**price**）的统计摘要信息，包含列类型、最小值、最大值、空值数量等信息。

```
SELECT
  compute_stats(price) AS result
FROM
  transactions;
```

返回示例：

```
+-------------------------------------------------------------------------------------+
|                                        result                                       |
+-------------------------------------------------------------------------------------+
| {"columntype":"Double","min":4.16,"max":68.43,"countnulls":0,"numdistinctvalues":20}  |
+-------------------------------------------------------------------------------------+
```

8.9.14.4. count

函数语法	返回类型	描述
count(expr)	Int	统计字段数量

说明

返回记录条数。

举例

```
SELECT count(1) AS result FROM somedata;
```

result
1314

8.9.14.5. covar_pop

函数语法	返回类型	描述
covar_pop(x,y)	Double	总体协方差

说明

返回两列数值的协方差，计算方式是：当x和y都不为null的时候，(SUM(x*y)-SUM(x)*SUM(y)/COUNT(x,y))/COUNT(x,y)，否则返回null。

举例

```
SELECT covar_pop(1,2) AS result FROM somedata;
```

result
0.0

8.9.14.6. covar_samp

函数语法	返回类型	描述
covar_samp(x,y)	Double	样本协方差

说明

返回两列的样本协方差。

举例

```
SELECT covar_samp(1,1) AS result FROM somedata;
```

result
0.0

8.9.14.7. corr

函数语法	返回类型	描述
corr(x, y)	String	统计字段数量

说明

返回两列数的皮尔森相关系数。

举例

```
SELECT corr(1,1) AS result FROM somedata;
```

result
NaN

8.9.14.8. ewah_bitmap

函数语法	返回类型	描述
ewah_bitmap(expr)	STRING	返回列的 EWAH 压缩位图表示形式。

参数说明：

- **expr：** 指定用于转换的表达式。

使用示例：

```
SELECT ewah_bitmap(trans_type) as result FROM transactions;
```

返回示例：

```
+--------------+
|    result    |
+--------------+
| [0,1,4,0,0]  |
+--------------+
```

8.9.14.9. histogram_numeric

函数语法	返回类型	描述
histogram_numeric(expr, nb)	array	统计字段数量

说明

用nb的值计算数字类型expr的直方图

举例

```
SELECT histogram_numeric(val, 3) AS result FROM somedata;
```

result
[{"x":296.5000000000003,"y":592.0},{"x":781.0,"y":377.0},{"x":1142.0000000000014,"y":345.0}]

8.9.14.10. kurtosis

函数语法	返回类型	描述
kurtosis(express)	Double	计算一组数值的峰度值，用于度量数据分布相对于正态分布的尖锐程度。峰度值大于 0 表示数据分布比正态分布更尖锐，峰度值小于 0 表示数据分布比正态分布更平坦，峰度值等于 0 表示数据分布与正态分布相似。

参数说明：

- **expr**：指定用于计算峰度值的表达式。

使用示例：

计算 **transactions** 表中，成交单价（**price**）的峰度值。

```
SELECT
  kurtosis(price) AS result
FROM
  transactions;
```

返回示例：

```
+---------------------+
|       result        |
+---------------------+
| 15.306714699109587  |
+---------------------+
```

8.9.14.11. max

函数语法	返回类型	描述
max(expr)	字段的公共类型	最小值

说明

返回某列的最小值。

举例

```
SELECT max(price) AS result FROM somedata;
```

result
100000

8.9.14.12. min

函数语法	返回类型	描述
min(expr)	字段的公共类型	最小值

说明

返回某列的最小值。

举例

```
SELECT min(price) AS result FROM somedata;
```

result
100000

8.9.14.13. percentile

函数语法	返回类型	描述
percentile(expr, p)	Double	数值区域的百分比数值点

说明

返回数值区域的百分比数值点。p的值必须取0到1之间,否则返回NULL，不支持浮点型数值。当expr输入为单值X时，无论p为何值，结果都是X。

举例

```
SELECT percentile(1,1) AS result FROM system.dual;
```

result
1.0

8.9.14.14. percentile

函数语法	返回类型	描述
percentile(BIGINT col, array(p1 [, p2]…))	Array<Double>	数值区域的百分比对应的数值点

说明

返回数值区域的一组百分比值分别对应的数值点。0 P 1,否则返回NULL,不支持浮点型数值。

举例

```
SELECT percentile(1,array('1')) AS result FROM system.dual;
```

result
[1.0]

8.9.14.15. percentile_approx

函数语法	返回类型	描述
percentile_approx(expr, array(p1 [, p2]…) [, accuracy])	Double/Array	返回组中数值列（包括浮点类型）的近似第 p 个百分位数，返回值为 DOUBLE 或数组。

参数说明：

- **expr**：指定用于计算的表达式。
- **array(p1 [, p2]…)**（可选）：传入一组数值可同时计算多个分位数的近似值，取值范围为：`(0,1)`，即不包含 0 和 1。
- **accuracy**（可选）：该参数以内存为代价控制近似精度，值越高，产生的近似值越好，默认值为 10,000，当 **expr** 中的非重复值的数量小于本参数的值时，会返回一个精确的百分位值。

使用示例：

计算 **transactions** 表中，成交单价（**price**）的第一四分位数和中位数。

```
SELECT
  percentile_approx(price, array(0.25, 0.5), 10000) AS result
FROM
  transactions;
```

返回示例：

```
+--------------+
|    result    |
+--------------+
| [6.12,8.64]  |
+--------------+
```

8.9.14.16. percentile_cont

函数语法	返回类型	描述
`percentile_cont(expr) WITHIN GROUP ( ORDER BY order_expr ) OVER([PARTITION BY partition_expr])`	Double	返回一个**连续**的百分位数，该函数将按照给定的排序顺序计算指定百分位数的值，返回类型与输入值类型一致。

参数说明：

- **expr**：百分位数的值。取值范围为 [0, 1]，表示所要计算的分位数，例如 0.5 表示中位数，0.25 表示第一四分位数。

- **order_expr**：可选，指定排序的表达式（如列名），支持的排序方式为 ASC（升序，默认）或 DESC（降序）。
- **partition_expr**：可选，指定参与分区的表达式。

使用示例：

在 **transactions** 表中，以连续的方式分别计算各交易类型（**trans_type**）的成交单价（**price**）的中位数。

```
SELECT
  trans_type,
  percentile_cont(0.5) WITHIN GROUP (
    ORDER BY
      price
  ) OVER(PARTITION BY trans_type) AS median_price
FROM
  transactions;
```

返回示例：

```
+-------------+---------------+
| trans_type  | median_price  |
+-------------+---------------+
| s           | 8.08          |
| s           | 8.08          |
| s           | 8.08          |
| s           | 8.08          |
| s           | 8.08          |
| s           | 8.08          |
| s           | 8.08          |
| s           | 8.08          |
| s           | 8.08          |
| s           | 8.08          |
| s           | 8.08          |
| s           | 8.08          |
| b           | 10.46         |
| b           | 10.46         |
| b           | 10.46         |
| b           | 10.46         |
| b           | 10.46         |
| b           | 10.46         |
| b           | 10.46         |
| b           | 10.46         |
+-------------+---------------+
```

8.9.14.17. percentile_disc

函数语法	返回类型	描述
`percentile_disc(expr) WITHIN GROUP ( ORDER BY order_expr ) OVER([PARTITION BY partition_expr])`	Double	返回一个**离散**的百分位数，该函数返回一个最接近指定百分位数的实际数据值，返回类型与输入值类型一致。如果没有正好对应的数据值，就取大于该分布值的下一个值。

参数说明：

- **expr**：百分位数的值。取值范围为 [0, 1]，表示所要计算的分位数，例如 0.5 表示中位数，0.25 表示第一四分位数。
- **order_expr**：可选，指定排序的表达式（如列名），支持的排序方式为 ASC（升序，默认）或 DESC（降序）。
- **partition_expr**：可选，指定参与分区的表达式。

使用示例：

在 transactions 表中，以离散的方式分别计算各交易类型（trans_type）的成交单价（price）的中位数。

```
SELECT
  trans_type,
  percentile_disc(0.5) WITHIN GROUP (
    ORDER BY
      price
  ) OVER(PARTITION BY trans_type) AS median_price
FROM
  transactions;
```

返回示例：

```
+-------------+---------------+
| trans_type  | median_price  |
+-------------+---------------+
| s           | 7.52          |
| s           | 7.52          |
| s           | 7.52          |
| s           | 7.52          |
| s           | 7.52          |
| s           | 7.52          |
| s           | 7.52          |
| s           | 7.52          |
| s           | 7.52          |
| s           | 7.52          |
| s           | 7.52          |
| s           | 7.52          |
| b           | 9.81          |
| b           | 9.81          |
| b           | 9.81          |
| b           | 9.81          |
| b           | 9.81          |
| b           | 9.81          |
| b           | 9.81          |
| b           | 9.81          |
+-------------+---------------+
```

8.9.14.18. skew

函数语法	返回类型	描述
skew(expr)	Double	计算一组数值的偏度，用于衡量数据分布的对称性，对于含有 NULL 值的数据，将被忽略。正偏度表示尾部向右延伸的分布；负偏度表示尾部向左延伸的分布。

参数说明：

- **expr**：列名或表达式。

使用示例：

在 transactions 表中，计算成交单价（price）的偏度值。

```
SELECT skew(price) AS result
FROM transactions;
```

返回示例：

```
+--------------------+
|       result       |
+--------------------+
| 3.758271312419232  |
+--------------------+
```

8.9.14.19. std

函数语法	返回类型	描述
std(x)	Double	标准差

说明

返回指定列的标准差。

举例

```
SELECT std(price) AS result FROM somedata;
```

result
0.0

8.9.14.20. stddev/stddev_pop

函数语法	返回类型	描述
stddev(x) 或 stddev_pop(x)	Double	标准差

说明

返回指定列的标准差,用法和 'std' 一致。

举例

```
SELECT stddev(price) AS result FROM somedata;
```

result
0.0

```
SELECT stddev_pop(price) AS result FROM somedata;
```

result
0.0

8.9.14.21. stddev_samp

函数语法	返回类型	描述
stddev_samp(x)	Double	样本标准差

说明

返回指定列的样本标准差。

举例

```
SELECT stddev_samp(price) AS result FROM somedata;
```

result
0.0

8.9.14.22. sum

函数语法	返回类型	描述
sum(expr)	Int	求和

说明

计算某一列所有值的和。

举例

```
SELECT sum(1) AS result FROM somedata;
```

result
1314

8.9.14.23. variance/var_pop

函数语法	返回类型	描述
variance(x) 或 var_pop(x)	Double	求方差

说明

返回某列所有值的方差。

举例

```
SELECT variance(price) AS result FROM somedata;
```

result
0.0

```
SELECT var_pop(price) AS result FROM somedata;
```

result
0.0

8.9.14.24. var_samp

函数语法	返回类型	描述
var_samp(x)	Double	求样本方差

说明

返回指定列的样本方差。

举例

```
SELECT var_samp(price) AS result FROM somedata;
```

result
0.0

8.9.14.25. wd_concat

函数语法	返回类型	描述
wd_concat(expr)	String	将列值中的字符串合并为一条记录。

参数说明：

- **expr：** 指定要合并字符串的表达式。

使用示例：

将用户信息表（**user_info**）中，账号等级（**acc_level**）列中的所有值合并为一条记录。

```
SELECT
  wd_concat(acc_level) AS result
FROM
  user_info;
```

返回示例：

```
+-------------+
|   result    |
+-------------+
| CCEDAACDAB  |
+-------------+
```

8.9.14.26. xmlagg

函数语法	返回类型	描述
xmlagg(expr)	String	获取一组 XML 片段并返回一个聚合的 XML 文档，输出结果时不会包含 XML 的元素标记。

参数说明：

- **expr**：指定要聚合的表达式。

使用示例：

将用户信息表（**user_info**）中，账号等级（**acc_level**）列中的所有值合并为一条记录。

> 如需在输出结果带上 XML 元素标记，可在 xmlagg 中嵌套使用 xmlforest 函数，例如：`SELECT xmlagg(xmlforest(acc_level)) AS result FROM user_info;`。

```
SELECT
  xmlagg(acc_level) AS result
FROM
  user_info;
```

返回示例：

```
+-------------+
|   result    |
+-------------+
| CDCCBADAAE  |
+-------------+
```

8.9.15. 窗口函数

窗口函数是SQL中一类特别的函数。和聚合函数相似，窗口函数的输入也是多行记录。不同的是，聚合函数的作用于由GROUP BY子句聚合的组，而窗口函数则作用于一个窗口，这里，窗口是由一个 **OVER子句** 定义的多行记录。聚合函数对其所作用的每一组记录输出一条结果，而窗口函数对其所作用的窗口中的每一行记录输出一条结果。一些聚合函数，如sum, max, min, avg,count等也可以当作窗口函数使用。

8.9.15.1. OVER子句

语法

```
SELECT window_function(args)
    OVER([PARTITION_BY_clause] [ORDER_BY_clause] [WINDOW_clause])

PARTITION_BY_clause: PARTITION BY column_name, column_name, ...
ORDER_BY_clause: ORDER BY column_name, column_name, ...
WINDOW_clause: ROWS|RANGE BETWEEN (CURRENT ROW |(UNBOUNDED |[num]) PRECEDING) AND (CURRENT ROW|(
UNBOUNDED |[num]) FOLLOWING)
```

说明

- OVER子句中的三个选项：PARTITION BY子句，ORDER BY子句和WINDOW子句都为可选项，也可以组合使用。OVER子句为空则表示窗口为整张表。
- PARTITION BY子句中可以用一个或多个键分区。和GROUP BY子句很像，PARTITION BY将表按分区键分区，每个分区是一个窗口，窗口函数作用于各个分区。
- ORDER BY子句决定窗口函数求值的顺序。ORDER BY子句也可以用一个或多个键排序。排序可以靠ASC或者DESC决定升序或者降序。当使用ORDER BY子句时，窗口可以由WINDOW子句指定。如果不指定，默认窗口等同于ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW，也就是窗口从表或者分区（如果OVER子句中用PARTITION BY分区）的开头开始到当前行结束。

- WINDOW子句让用户通过指定一个行区间来定义窗口。
- CURRENT ROW代表当前行。
- num PRECEDING定义窗口的下限：窗口从当前行向前数num行处开始；UNBOUNDED PRECEDING代表窗口没有下限。
- num FOLLOWING定义窗口的上限：窗口从当前行向后数num行处结束；UNBOUNDED FOLLOWING代表窗口没有上限。
- ROWS BETWEEN…和RANGE BETWEEN…的区别： ROWS定义的是物理行数的窗口，窗口宽度由行数计算。RANGE BETWEEN定义的是ORDER BY排序后的逻辑行数的窗口，窗口宽度由排名计算，只要值相同的不同列属于同一窗口。

举例：窗口子句:

```
ROWS BETWEEN CURRENT ROW AND CURRENT ROW  --窗口只包含当前行
ROWS BETWEEN 3 PRECEDING AND 5 FOLLOWING   --窗口从当前行向前数3行开始，到当前行向后数5行结束
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW --窗口从表或分区的开头开始，到当前行结束
ROWS BETWEEN CURRENT AND UNBOUNDED FOLLOWING --窗口从当前行开始，到表或分区的结尾结束
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
--窗口从表或分区的开头开始，到表或分区的结尾结束
```

举例：ROWS BETWEEN和RANGE BETWEEN的区别

```
SELECT * FROM row_range;
```

col1
A
A
B
C
D

```
SELECT col1, count(col1) OVER (ORDER BY col1 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
result FROM row_range;
```

col1	result
A	1
A	2
B	3
C	4
D	5

```
SELECT col1, count(col1) over (order by col1 range between unbounded preceding and current row) AS
result FROM row_range;
```

col1	result
A	2
A	2
B	3
C	4
D	5

举例：ORDER BY, PARTITION BY在OVER子句中的使用：

```
SELECT name, count(name) OVER (ORDER BY name) AS result FROM user_info;
```

name	result
华*	1
宁**	2
李*	3
李**	4
潘**	5
祝**	6
管**	7
邱*	8
马**	9
魏**	10

```
SELECT name, count(name) OVER (PARTITION BY name) AS result FROM user_info;
```

name	result
华*	1
宁**	1
马**	1
李*	1
李**	1
祝**	1
魏**	1
管**	1
邱*	1
潘**	1

```
SELECT name, reg_date, acc_level, count(name) OVER (PARTITION BY acc_level ORDER BY reg_date) AS
result FROM user_info;
```

name	reg_date	acc_level	result
宁**	2008-10-31	D	1
李**	2011-09-16	D	2
魏**	2009-12-02	A	1
马**	2011-01-01	A	2
邱*	2012-10-24	A	3
李*	2013-07-02	E	1
华*	2008-02-14	B	1
祝**	2010-01-01	C	1
潘**	2011-04-30	C	2
管**	2014-10-03	C	3

```
SELECT name, acc_level, count(acc_level) OVER (PARTITION by acc_level) AS result FROM user_info;
```

name	acc_level	result
宁**	D	2
李**	D	2
马**	A	3
魏**	A	3
邱*	A	3
李*	E	1
华*	B	1
祝**	C	3
潘**	C	3
管**	C	3

8.9.15.2. DISTINCT与窗口函数

目前一些窗口函数支持DISTINCT对象，其中包括COUNT、SUM、AVG。

语法

```
SELECT func(DISTINCT col1) OVER (...)
FROM <table> ...
```

示例

窗口函数 COUNT(DINSTINT) 示例

```
SELECT trans_type, acc_num, COUNT(DISTINCT acc_num) OVER (PARTITION BY trans_type)
FROM transactions;
```

trans_type	acc_num	_wcol0
b	3912384	4
b	6513065	4
b	700735	4
b	6513065	4
b	6513065	4
b	3912384	4
b	5224133	4
b	6513065	4
s	3912384	8
s	700735	8
s	700735	8
s	2394923	8
s	5224133	8
s	3912384	8
s	6670192	8
s	6600641	8

trans_type	acc_num	_wcol0
s	6513065	8
s	6513065	8
s	2755506	8
s	6670192	8

窗口函数 SUM(DINSTINT) 示例

```
SELECT trans_type, amount, SUM(DISTINCT amount) OVER (PARTITION BY trans_type)
FROM transactions;
```

trans_type	amount	_wcol0
b	100.0	3600.0
b	800.0	3600.0
b	100.0	3600.0
b	300.0	3600.0
b	700.0	3600.0
b	1000.0	3600.0
b	200.0	3600.0
b	500.0	3600.0
s	1100.0	6900.0
s	100.0	6900.0
s	1300.0	6900.0
s	900.0	6900.0
s	400.0	6900.0
s	100.0	6900.0

trans_type	amount	_wcol0
s	600.0	6900.0
s	200.0	6900.0
s	200.0	6900.0
s	500.0	6900.0
s	800.0	6900.0
s	1000.0	6900.0

窗口函数 AVG(DINSTINT) 示例

```
SELECT trans_type, price, AVG(DISTINCT price) OVER (PARTITION BY trans_type)
FROM transactions;
```

trans_type	price	_wcol0
b	68.43	17.31125
b	6.36	17.31125
b	7.02	17.31125
b	5.25	17.31125
b	12.13	17.31125
b	9.81	17.31125
b	11.11	17.31125
b	18.38	17.31125
s	10.31	9.00833333333333333
s	9.16	9.00833333333333333

trans_type	price	_wcol0
s	7.49	9.00833333333333
s	6.12	9.00833333333333
s	7.52	9.00833333333333
s	8.64	9.00833333333333
s	4.16	9.00833333333333
s	22.66	9.00833333333333
s	5.3	9.00833333333333
s	12.21	9.00833333333333
s	10.03	9.00833333333333
s	4.5	9.00833333333333

暂不支持子句中带有ORDER BY的窗口。

8.9.15.3. 窗口函数详解

下面我们列举所有Inceptor支持的窗口函数，包含：cume_dist、dense_rank、first_value、last_value、lag、lead、nth、ntile、percent_rank、rank、row_number。

8.9.15.3.1. cume_dist()

函数名	描述
cume_dist()	返回当前行在窗口中的累积分布函数

说明

返回值为当前行在所在窗口中的百分排名，和percent_rank很像。假设按升序排列，那么cume_dist计算的是“窗口中当前行之前，包括当前行和与当前行排名相同的行数”除以“窗口中的总行数”。

举例

```
SELECT name, cume_dist()
OVER(ORDER BY acc_level) AS cume_dist
FROM user_info;
```

name	cume_dist
马**	0.3
魏**	0.3
邱*	0.3
华*	0.4
祝**	0.7
潘**	0.7
管**	0.7
宁**	0.9
李**	0.9
李*	1.0

8.9.15.3.2. dense_rank()

函数名	描述
dense_rank()	返回窗口中行的排名

说明

dense_rank()对排名并列的行返回相同值，在并列排名之后的记录 **不会** 空出并列所占名次。

举例

```
SELECT name, acc_level, dense_rank()
OVER(ORDER BY acc_level) AS dense_rank
FROM user_info;
```

name	acc_level	dense_rank
马**	A	1

name	acc_level	dense_rank
魏**	A	1
邱*	A	1
华*	B	2
祝**	C	3
潘**	C	3
管**	C	3
宁**	D	4
李**	D	4
李*	E	5

8.9.15.3.3. first_value(expression)

函数名	描述
first_value(expression)	返回一个有序集合中的第一个值。

说明

first_value的参数可以是列，输出结果为一列的子查询或者其他结果为一列的表达式。

举例

下例查询user_info表中最早注册的用户：

```
SELECT name, reg_date, first_value(name)
OVER(ORDER BY reg_date) AS first_user
FROM user_info;
```

name	reg_date	first_user
华*	2008-02-14	华*
宁**	2008-10-31	华*
魏**	2009-12-02	华*
祝**	2010-01-01	华*
马**	2011-01-01	华*
潘**	2011-04-30	华*
李**	2011-09-16	华*
邱*	2012-10-24	华*
李*	2013-07-02	华*
管**	2014-10-03	华*

8.9.15.3.4. last_value(expression)

函数名	描述
last_value(expression)	返回一个有序集合中的最后一个值。

说明

last_value的参数可以是列，输出结果为一列的子查询或者其他结果为一列的表达式。

举例

下例查询user_info表中最晚注册的用户：

```
SELECT name, reg_date, last_value(name)
OVER(ORDER BY reg_date
     ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING)
AS last_user
FROM user_info;
```

name	reg_date	last_user
华*	2008-02-14	管**
宁**	2008-10-31	管**
魏**	2009-12-02	管**
祝**	2010-01-01	管**
马**	2011-01-01	管**
潘**	2011-04-30	管**
李**	2011-09-16	管**
邱*	2012-10-24	管**
李*	2013-07-02	管**
管**	2014-10-03	管**

注意，这里我们在ORDER子句中加了WINDOW子句，指定窗口为从当前行开始到表结尾结束。这是因为如果不指定窗口，默认窗口为从表开头开始，到当前行结束。ORDER BY的默认顺序为升序：reg_date以从小到大排列。所以在不指定窗口的情况下，第n行的last_value(name)值将是前n个注册的用户中最后一个注册的用户，也就是第n个注册的用户；last_value(name)的结果会和按reg_date升序排列一模一样：

```
SELECT name, reg_date, last_value(name)
OVER(ORDER BY reg_date) AS last_user
FROM user_info;
```

name	reg_date	last_user
华*	2008-02-14	华*
宁**	2008-10-31	宁**
魏**	2009-12-02	魏**
祝**	2010-01-01	祝**

name	reg_date	last_user
马**	2011-01-01	马**
潘**	2011-04-30	潘**
李**	2011-09-16	李**
邱*	2012-10-24	邱*
李*	2013-07-02	李*
管**	2014-10-03	管**

8.9.15.3.5. lag(expression, n)

函数名	描述
lag(expression, n)	返回从当前行开始向前的第n行记录

说明

n为可选项，不选默认值为1，就是返回当前行前一行的记录。当没有记录时，返回NULL。必须和OVER子句中的ORDER BY子句合用。

举例

```
SELECT name, lag(name)
OVER(ORDER BY reg_date) AS lag_name
FROM user_info;
```

name	lag_name
华*	NULL
宁**	华*
魏**	宁**
祝**	魏**
马**	祝**
潘**	马**
李**	潘**
邱*	李**
李*	邱*
管**	李*

```
SELECT name, lag(name,2)
OVER(ORDER BY reg_date) AS lag_name_2
FROM user_info;
```

name	lag_name_2
华*	NULL

name	lag_name_2
宁**	NULL
魏**	华*
祝**	宁**
马**	魏**
潘**	祝**
李**	马**
邱*	潘**
李*	李**
管**	邱*

8.9.15.3.6. lead(expression, n)

函数名	描述
lead(expression, n)	返回从当前行开始向后的第n行记录

说明

n为可选项，不选默认值为1，就是返回当前行后一行的记录。当没有记录时，返回NULL。

举例

```
SELECT name, lead(name)
OVER(ORDER BY reg_date) AS lead_name
FROM user_info;
```

name	lead_name
华*	宁**
宁**	魏**
魏**	祝**
祝**	马**
马**	潘**
潘**	李**
李**	邱*
邱*	李*
李*	管**
管**	NULL

```
SELECT name, lead(name,2)
OVER(ORDER BY reg_date) AS lead_name_2
FROM user_info;
```

name	lead_name_2
华*	魏**
宁**	祝**
魏**	马**
祝**	潘**
马**	李**
潘**	邱*
李**	李*
邱*	管**
李*	NULL
管**	NULL

8.9.15.3.7. nth_value()

函数名	描述
`nth_value(expression, int x) OVER ( [PARTITION BY partition_column] [ORDER BY sort_column] )`	返回位于窗口中，特定行数的值，如果指定的行不存在，则返回 NULL。

参数说明：

- **expression**：列名或表达式。
- **int x**：行的编号，正整数。
- **partition_expr**：指定参与分区的表达式。
- **sort_expr**：指定参与排序的表达式。

使用示例：

在 **transactions** 表中，查询每个交易类型（**trans_type**）下，成交单价（**price**）最高的账号 ID。

```
SELECT
  trans_type,
  price,
  nth_value(acc_num, 1) OVER (
    PARTITION BY trans_type
    ORDER BY
      price desc
  ) AS top_account_id
FROM
  transactions;
```

返回示例：

```
+-------------+--------+-----------------+
| trans_type  | price  | top_account_id  |
+-------------+--------+-----------------+
| s           | 22.66  | 0700735         |
| s           | 12.21  | 0700735         |
| s           | 10.31  | 0700735         |
| s           | 10.03  | 0700735         |
| s           | 9.16   | 0700735         |
| s           | 8.64   | 0700735         |
| s           | 7.52   | 0700735         |
| s           | 7.49   | 0700735         |
| s           | 6.12   | 0700735         |
| s           | 5.3    | 0700735         |
| s           | 4.5    | 0700735         |
| s           | 4.16   | 0700735         |
| b           | 68.43  | 3912384         |
| b           | 18.38  | 3912384         |
| b           | 12.13  | 3912384         |
| b           | 11.11  | 3912384         |
| b           | 9.81   | 3912384         |
| b           | 7.02   | 3912384         |
| b           | 6.36   | 3912384         |
| b           | 5.25   | 3912384         |
+-------------+--------+-----------------+
```

8.9.15.3.8. ntile()

函数名	描述
`ntile(int x) OVER ( [PARTITION BY partition_expr] [ORDER BY sort_expr] )`	将结果集分为 n 个大致相等的组，并为每个组分配一个编号。

参数说明：

- **int x**：将结果集分为多少个相等的组，必须是正整数。
- **partition_expr**：指定参与分区的表达式。
- **sort_expr**：指定参与排序的表达式。

注意事项：

如果行数不能被分组数整除，则将未被整除分配的记录，按照 ORDER BY 子句指定的顺序继续分配。

使用示例：

查询 **transactions** 表的数据，将每个交易类型（**trans_type**）的成交单价（**price**），按照降序的方式分成 2 个组。

```
SELECT
  trans_type,
  price,
  ntile(2) OVER (
    PARTITION BY trans_type
    ORDER BY
      price desc
  ) AS groups
FROM
  transactions;
```

返回示例：

```
+-------------+--------+---------+
| trans_type  | price  | groups  |
+-------------+--------+---------+
| s           | 22.66  | 1       |
| s           | 12.21  | 1       |
| s           | 10.31  | 1       |
| s           | 10.03  | 1       |
| s           | 9.16   | 1       |
| s           | 8.64   | 1       |
| s           | 7.52   | 2       |
| s           | 7.49   | 2       |
| s           | 6.12   | 2       |
| s           | 5.3    | 2       |
| s           | 4.5    | 2       |
| s           | 4.16   | 2       |
| b           | 68.43  | 1       |
| b           | 18.38  | 1       |
| b           | 12.13  | 1       |
| b           | 11.11  | 1       |
| b           | 9.81   | 2       |
| b           | 7.02   | 2       |
| b           | 6.36   | 2       |
| b           | 5.25   | 2       |
+-------------+--------+---------+
```

8.9.15.3.9. percent_rank()

函数名	描述
percent_rank()	返回当前行在窗口中的百分比排名

说明

假设按升序排列，那么percent_rank()计算的是“窗口中当前行之前，不包括当前行和与当前行排名相同的行数”除以“窗口中的总行数-1”。

举例

```
SELECT trans_type, COUNT(DISTINCT acc_num) OVER (PARTITION BY trans_type)
FROM transactions;
```

trans_type	_wcol0
b	4
b	4
b	4
b	4
b	4
b	4
b	4
b	4
s	8
s	8
s	8
s	8

trans_type	_wcol0
s	8
s	8
s	8
s	8
s	8
s	8
s	8
s	8

8.9.15.3.10. rank()

函数名	描述
rank()	返回窗口中行的排名

说明

rank()对排名并列的行返回相同值，在并列排名之后的记录会空出并列所占名次。

举例

```
SELECT name, acc_level, rank()
OVER(ORDER BY acc_level) AS rank
FROM user_info;
```

name	acc_level	rank
马**	A	1
魏**	A	1
邱*	A	1
华*	B	4
祝**	C	5
潘**	C	5
管**	C	5
宁**	D	8
李**	D	8
李*	E	10

8.9.15.3.11. row_number()

函数名	描述
row_number()	返回窗口中行的序号。

说明

表或分区中的第一行序号为1，之后序号逐行递增。在同一窗口中，行的序号是唯一的。

举例

```
SELECT name, row_number()
OVER(ORDER BY reg_date) AS row_num
FROM user_info;
```

name	row_num
华*	1
宁**	2
魏**	3
祝**	4
马**	5
潘**	6
李**	7
邱*	8
李*	9
管**	10

```
SELECT name, acc_level,row_number()
OVER(PARTITION BY acc_level ORDER BY reg_date) AS row_num
FROM user_info;
```

name	acc_level	row_num
宁**	D	1
李**	D	2
魏**	A	1
马**	A	2
邱*	A	3
李*	E	1
华*	B	1
祝**	C	1
潘**	C	2
管**	C	3

8.9.16. Context函数

您可以在Inceptor中查看当前session的上下文（context）信息。下面我们详细介绍各个context函数的用法。

8.9.16.1. current_user

函数current_user用于查看当前session中用户的用户名。

用法

```
SELECT current_user() FROM <table_name> LIMIT 1;
SELECT ... FROM <table_name> WHERE <column> = current_user();
```

说明

<table_name>处提供任意一张当前用户 **有SELECT权限的** 表的表名。

例1

查看当前用户名：

```
SELECT current_user() AS result FROM user_account LIMIT 1;
```

result
alice

例2

查看当前用户名，但是用户没有表的SELECT权限，Inceptor报错：

```
SELECT current_user() FROM zzz LIMIT 1;
```

```
COMPILE FAILED: Internal error HiveAccessControlException: [Error 20388] Permission denied:
Principal [name=alice, type=USER] does not have following privileges for operation QUERY [[SELECT]
on Object [type=TABLE_OR_VIEW, name=manual_func_op.zzz]]
```

例3

我们有一张用户表user_account：

```
SELECT * FROM user_account;
```

id	name
3	carol
1	alice
2	bob

从user_account表中查询出name为当前用户名的记录：

```
SELECT * FROM user_account
WHERE name = current_user();
```

id	name
1	alice

8.9.16.2. logged_in_user

logged_in_user 用于查看当前登录的用户名，用法与 current_user() 类似。

用法

```
SELECT logged_in_user() FROM <table_name> LIMIT 1;
SELECT ... FROM <table_name> WHERE <column> = logged_in_user();
```

说明

<table_name>处提供任意一张当前用户 **有SELECT权限的** 表的表名。如果<table_name>是当前登录用户没有SELECT权限的表，系统同样也会报错，与current_user()函数类似。

例1

查看当前登录的用户名：

```
SELECT logged_in_user() AS result FROM user_account LIMIT 1;
```

result
hive

8.9.16.3. has_role

函数has_role用于查看用户在当前session中是否拥有某个角色。

用法

```
SELECT has_role('ADMIN') AS result FROM user_account LIMIT 1;
```

result
false

说明

- <table_name>处提供任意一张当前用户 **有SELECT权限的** 表的表名。
- <role_name>处提供用户想要查看的角色。
- 如果用户在当前session有<role_name>指定的角色，那么查询结果为true；如果用户在当前session没有<role_name>指定的角色，那么查询结果为false。
- 在Inceptor中，角色名是大小写 **不敏感** 的，例如Accounting和accounting是同一个角色。

例1

查看用户在当前session中是否有admin角色：

```
SELECT has_role('ADMIN') FROM zzz LIMIT 1;
```

```
COMPILE FAILED: Internal error HiveAccessControlException: [Error 20388] Permission denied:
Principal [name=alice, type=USER] does not have following privileges for operation QUERY [[SELECT]
on Object [type=TABLE_OR_VIEW, name=manual_func_op.zzz]]
```

例2

查看用户在当前session中是否有admin角色，但是用户没有表的SELECT权限，Inceptor报错：

```
SELECT has_role('ADMIN') FROM zzz LIMIT 1;
```

```
COMPILE FAILED: Internal error HiveAccessControlException: [Error 20388] Permission denied:
Principal [name=alice, type=USER] does not have following privileges for operation QUERY [[SELECT]
on Object [type=TABLE_OR_VIEW, name=manual_func_op.zzz]]
```

例3

查看用户在当前session中是否有accounting角色：

```
SELECT has_role('accounting') AS result
FROM user_account LIMIT 1;
```

result
true

has_role在HiveServer1和简单模式的HiveServer2下返回一定为true。

8.9.16.4. current_database

返回用户当前所在数据库。

例

```
SELECT current_database() AS result FROM system.dual LIMIT 1;
```

result
manual_func_op

8.9.16.5. current_time

返回当前系统时间。和 **systime** 同义。

例

```
SELECT current_time() AS result FROM system.dual LIMIT 1;
```

result
19:54:24

8.9.16.6. current_date

返回当前系统日期和时间。

例

```
SELECT current_date() AS result FROM system.dual LIMIT 1;
```

result
2018-09-10 19:54:24

8.9.16.7. current_timestamp

返回当前系统TIMESTAMP。和 **systimestamp** 同义。

例

```
SELECT current_timestamp() AS result FROM system.dual LIMIT 1;
```

result
2018-09-10 19:54:24.931

8.9.17. 脱敏相关函数

Inceptor提供以下几个敏感词隐藏（脱敏）函数，常用于列级权限控制。

- MASK

函数名	返回类型	描述
MASK(<data_string>[, <mask_string>], <pre_length>[, <post_length>])	STRING	用于隐藏电话号码。使用 **mask_string** 替代 **data_string** 中的字符。保留前面 **pre_length** 长度的字符；可选择保留尾部 **post_length** 长度的字符。默认的 **mask_string** 为 ，**默认的 post_length*** 值为 0。

例 93. MASK

```
SELECT mask('1234567', '*', 2, 3) AS result FROM system.dual LIMIT 1;
```

result
12**567

```
SELECT mask('1234567', 2, 3) AS result FROM system.dual LIMIT 1;
```

result
12**567

```
SELECT mask('1234567', '*', 2) AS result FROM system.dual LIMIT 1;
```

result
12*

```
SELECT mask('1234567', 2) AS result FROM system.dual LIMIT 1;
```

result
12*

- MASK_EMAIL

函数名	返回类型	描述
MASK_EMAIL(<data_string>[, <mask_string>, <pre_length>, <post_length>])	STRING	用于隐藏email。使用 **mask_string** 替代 **data_string** 中 **@** 符号之前的字符，**@** 后面的域名不会被隐藏。可选择保留前面 **pre_length** 长度的字符，可选择保留尾部（**@** 之前） **post_length** 长度的字符。默认的 **mask_string** 为 ，默认的 **pre_length** 值为 1，默认的 **post_length*** 值为 0。

例 94. MASK_EMAIL

```
SELECT mask_email('test@sample.io', '*', 1, 0) AS result FROM system.dual LIMIT 1;
```

result
t***@sample.io

```
SELECT mask_email('test@sample.io') AS result FROM system.dual LIMIT 1;
```

result
t***@sample.io

- MASK_COMPANY

函数名	返回类型	描述
MASK_COMPANY(<data_string>[, <mask_string>])	STRING	用于隐藏公司名。使用 mask_string 替代 data_string 中的字符，只保留“有限公司”及其之后的所有字符。默认的 mask_string 为 *。

例 95. MASK_COMPANY

```
SELECT mask_company('虚构的有限公司') AS result FROM system.dual LIMIT 1;
```

result
***有限公司

```
SELECT mask_company('中科院', '#') AS result FROM system.dual LIMIT 1;
```

result
###

- CAESAR_CIPHER

函数名	返回类型	描述
CAESAR_CIPHER(<data_string>[, <shift_value>])	STRING	右移 **data_string** 中的数字和字母字符（只有数字和字母会被右移），可以使用 **shift_value** 设置偏移量。如“abc123”中的数字和字母右移1位得到“bcd234”。默认的 **shift_value** 为3。

例 96. CAESAR_CIPHER

```
SELECT caesar_cipher('012abc6789') AS result FROM system.dual LIMIT 1;
```

result
345def9012

```
SELECT caesar_cipher('012ABC6789', 1) AS result FROM system.dual LIMIT 1;
```

result
123BCD7890

8.9.18. 其他函数和运算符

8.9.18.1. oracle_coalesce

函数	返回类型	描述
oracle_coalesce (<arg1>, [<arg2>, …])	见描述	函数需要至少一个参数。返回第一个非NULL参数。空字符串*’’* 作为NULL STRING处理。如果所有参数都是NULL，返回NULL。

```
SELECT oracle_coalesce(NULL, '') AS result
FROM system.dual LIMIT 1;
```

result
NULL

```
SELECT oracle_coalesce(NULL, '', 1) AS result
FROM system.dual LIMIT 1;
```

result
1

8.9.18.2. oracle_decode

函数	返回类型	描述
oracle_decode(<expr>, <search1>, <result1>, [<search2>, <result2>, …] <default>)	见描述	至少需要四个参数：**<exp>**, **<search1>**, **<result1>** 和 **<default>**。可选参数必须成对出现。该函数将 **<expr>** 和参数中的所有 **<search>** 比较，如果和某个 **<search>** 相等，则返回对应 **<result>**。如果不和任何 **<search>** 相等，返回 **<deafult>**。

```
SELECT oracle_decode(0, 0, 'zero', 1, 'one', 'others') AS result
FROM system.dual LIMIT 1;
```

result
zero

```
SELECT oracle_decode(1, 0, 'zero', 1, 'one', 'others') AS result
FROM system.dual LIMIT 1;
```

result
one

```
SELECT oracle_decode(2, 0, 'zero', 1, 'one', 'others') AS result
FROM system.dual LIMIT 1;
```

result
others

8.9.18.3. to_card_15_to_18 (<15_digit_int>)

函数	返回类型	描述
to_card_15_to_18 (<15_digit_int>)	INT	用于将15位身份证号转为18位。参数必须是15位正整数。

```
SELECT to_card_15_to_18(370802940221002) AS result
FROM system.dual LIMIT 1;
```

result
370802199402210029

8.9.18.4. zeroifnull (<expr>)

函数	返回类型	描述
zeroifnull (<expr>)	见描述	如果 **<expr>** 为NULL，则返回0，否则返回 **<expr>**。

```
SELECT zeroifnull(NULL) AS result FROM system.dual LIMIT 1;
```

result
0

8.9.18.5. nullif(expr1, expr2)

函数名	返回类型	描述
nullif(expr1, expr2)	与 expr1 相同	比较两个表达式的值，如果相等则返回 NULL，否则返回第一个表达式的值。

参数说明：

- `expr1`：要比较的第一个表达式。
- `expr2`：要比较的第二个表达式。

使用示例：

比较两个字符串，如果不同则返回第一个字符串。

```
SELECT nullif('Inceptor','inceptor' ) AS result;
```

返回示例：

```
+-----------+
| result    |
+-----------+
| Inceptor  |
+-----------+
```

8.9.18.6. nullifzero (<expr>)

函数	返回类型	描述
nullifzero (<expr>)	见描述	如果 **<expr>** 为0，则返回NULL，否则返回 **<expr>**。

```
SELECT nullifzero(0) AS result FROM system.dual LIMIT 1;
```

result
NULL

field (<str>, <str1> [, <str2>, <str3>, …])

函数	返回类型	描述
field (<str>, <str1> [, <str2>, <str3>, …])	INT	需要至少两个原生类型的参数。返回 **<str>** 在其他参数中的index（从1开始数）。如果在其他参数中没有 **<str>**，返回0。

```
SELECT field('a', 'b', 'a', 'c', 'h') AS result
FROM system.dual LIMIT 1;
```

result
2

```
SELECT field('d', 'b', 'a', 'c', 'h') AS result
FROM system.dual LIMIT 1;
```

result
0

8.9.18.7. reflect

函数	返回类型	描述
reflect ("<java_class>", "<method>" [, <arg1>, <arg2>, …])	见描述。	使用该函数可以在Inceptor中直接调用Java方法。reflect 需要至少两个函数：Java方法 <method> 和它所在的类 <java_class>。如果 <method> 需要参数，在 <method> 后提供。和 java_method 同义。

```
SELECT reflect("java.lang.Math", "max", 2, 3) AS result
FROM system.dual LIMIT 1;
```

result
3

8.9.18.8. java_method

函数	返回类型	描述
java_method ("<java_class>", "<method>" [, <arg1>, <arg2>, …])	见描述	和 reflect 同义。

用例见 reflect。

8.9.18.9. reflect2

函数	返回类型	描述
reflect2 (<arg0>, "<method>" [<arg1>, <arg2>, …])	见描述	需要至少两个参数。Inceptor会根据 <arg0> 的类型选择使用的类（例如如果 <arg0> 是INT，Inceptor会使用java.lang.Integer类），然后在该类中找到 <method>，将 <arg0> 当做 <method> 的输入参数。如果 <method> 需要多个参数，在 "<method>" 之后提供。

```
SELECT reflect2(1, "equals", 2) AS result
FROM system.dual LIMIT 1;
```

result
false

8.9.18.10. hash

函数	返回类型	描述
hash (<arg1>, <arg2>, …)	INT	需要至少一个参数。返回Inceptor为所有参数计算的一个哈希值。

```
SELECT hash(1,'a') AS result FROM system.dual LIMIT 1;
```

result
159

8.9.18.11. assert_true-def

函数	返回类型	描述
assert_true-def (<condition>)	NULL或抛异常	需要一个条件 **<condition>** 作为参数。如果条件不成立，函数会抛异常；如果成立，返回 NULL。

```
SELECT assert_true(1<=0) FROM system.dual LIMIT 1;
```

```
-1
```

```
SELECT assert_true(0<=0) AS result FROM system.dual LIMIT 1;
```

result
NULL

8.9.18.12. uniq()

函数	返回类型	描述
uniq()	LONG	无须参数。为每行记录返回一个唯一的ID。

```
SELECT uniq() AS result FROM system.dual;
```

result
-281474976710656

8.9.18.13. grouping (<column>)

函数	返回类型	描述
grouping (<column>)	BOOLEAN	需要一个列名 **<column>** 作为参数。使用 **GROUP BY** 扩展（**ROLLUP/CUBE/GROUPING SETS**）会在超聚合行中产生NULL值。**grouping** 会判断 **<column>** 中的NULL值是实际的NULL值（返回0）还是超聚合产生的NULL。该函数只能在有 **GROUP BY** 的查询语句中使用，并且 **<column>** 必须是聚合列之一。

```
SELECT decode(grouping(type), 1, 'All Fruits', type) type, sum(amount) AS result FROM inventory
GROUP BY ROLLUP(type) ORDER BY type;
```

type	result
All Fruits	5000.0
apple	3000.0
orange	400.0
pear	1600.0

8.9.18.14. grouping_id

函数	返回类型	描述
grouping_id (<column> [<column>, …])	INT	**grouping** 可以用于判断一行记录是否是超聚合行；**grouping_id** 则可以判断一行记录是否是多次聚合产生的超聚合行，**grouping_id** 对参数中的每一列计算 **grouping**，返回所有 **grouping** 的和。

```
SELECT grouping(type) gt, grouping_id(type, store) gts, grouping_id(type, store, grade) gtsg, sum
(amount) AS sum_amt
FROM inventory
GROUP BY ROLLUP(type, store, grade)
ORDER BY gt;
```

gt	gts	gtsg	sum_amt
0	0	0	300.0
0	0	0	500.0
0	0	0	500.0
0	0	1	700.0
0	0	1	400.0
0	0	1	1300.0
0	0	0	400.0
0	0	0	1000.0
0	0	0	200.0

gt	gts	gtsg	sum_amt
0	0	0	100.0
0	0	1	900.0
0	0	1	1500.0
0	1	3	1600.0
0	0	0	700.0
0	1	3	400.0
0	1	3	3000.0
0	0	0	800.0
0	0	0	500.0
0	0	1	200.0
1	3	7	5000.0

8.9.18.15. range_decode

函数	返回类型	描述
range_decode (<expr>, <is_left_close>, <is_right_close>, <range1_start>, <range1_end>, <result> [<range2_start>, <range2_end>, <result>, …], <default_result>)	见描述	该函数在参数给定的不同range中寻找 **<expr>**，然后返回参数所在区间的result。

说明

该函数至少需要七个参数：**<expr>**, **<is_left_close>**, **<is_right_close>**, **<range1_start>**, **<range1_end>**, **<result>** * 和 ***<default_result>**。其中：

<expr> 是Inceptor要寻找的表达式。
<is_left_clode> 和 **<is_right_clode>** 是两个BOOLEAN，分别指定区间的是否左闭和右闭。TRUE为闭合，FALSE为打开。
<range_start> 和 **<range_end>** 分别是区间的开头和结尾。
<result> 为 **<expr>** 落在对应区间中的返回值。
如果 **<expr>** 不落在任何给定区间中，则返回 **<default_result>**。

举例

```
SELECT range_decode(8.8,true,true,1,9,'product',10,20,'Rocks','productinitials') AS result FROM
system.dual LIMIT 1;
```

result
product

8.9.18.16. uuid()

函数	返回类型	描述
uuid()	STRING	返回通用唯一识别码

说明

UUID是指在一台机器上生成的数字，它保证对在同一时空中的所有机器都是唯一的。

UUID由以下部分组成：

当前日期和时间。UUID的第一个部分与时间有关，如果你在生成一个UUID之后，过几秒又生成一个UUID，则第一个部分不同，其余相同。
时钟序列。
全局唯一的IEEE机器识别号。如果有网卡，从网卡MAC地址获得，没有网卡则以其他方式获得。

举例

```
SELECT uuid() AS result FROM system.dual;
```

result
cc72d83c-24a4-4ade-96d8-0b6a3bbda7ab

8.9.18.17. replicate_rows

函数	返回类型	描述
replicate_rows(n, col1, col2…)	STRING	对指定列复制n次

说明

参数 **n** 必须为BIGINT类型，col1, col2…为可选参数，指定要复制的列名。返回结果中，第一列为指定的复制次数，后面列为被复制的列。

举例

```
SELECT * FROM replicate_t;
```

x	y	z
3	2	0
2	3	1
0	2	2

```
SELECT replicate_rows(cast(2 AS BIGINT),x,y,z) FROM replicate_t;
```

col0	col1	col2	col3
2	3	2	0
2	3	2	0
2	2	3	1
2	2	3	1
2	0	2	2
2	0	2	2

8.10. 空串和 NULL

Inceptor 中，不同的字符串类型中对于空串 '' 和 NULL 的定义不同，本文主要向您介绍 Inceptor 中空串和 NULL 的含义，以及 Oracle、DB2、TD 方言中的相关兼容情况。

8.10.1. 字符串类型区别

对于 Inceptor 数据库中的字符串数据，空串和 NULL 有以下区别：

- STRING、VARCHAR(n) 类型的字段，赋值为空串，结果为空串。其中空串 '' 不等于 NULL。
- CHAR(n)、VARCHAR2(n) 类型的字段，赋值为空串，结果自动转为 NULL。其中空串 '' 就是 NULL。

注意事项

- Inceptor 中，字符串类型只支持 STRING 和定长数据类型 CHAR(n)、VARCHAR(n)、VARCHAR2(n)。
- Inceptor 中，定长字符串 CHAR(n)、VARCHAR(n)、VARCHAR2(n) 中的 n 指的是 unicode 编码的长度，而不是字节长度，即 n 表示可以存储 n 个字符。

8.10.2. Inceptor 版本兼容

- Inceptor 6.x 系列产品中，SQL 语句中的 '' 默认为空字符串，不等于 NULL，因为字符串常量默认为 STRING 类型，如：

```
select '' from system.dual;
```

- Inceptor 9.3 及之后的系列产品中，SQL 语句中的 '' 默认为 NULL，因为字符串常量的类型默认为 CHAR(n) ，n 是字符串常量的长度。
- Inceptor 9.3 之后的系列产品中，可以通过修改参数 **character.literal.as.string** 来设置字符串常量默认类型为 STRING，方便兼容老版行为。

```
set character.literal.as.string=true; --默认为 false
```

参数说明

修改参数 **character.literal.as.string = true** 的影响：

- 空字符串 '' 被识别为空串，而不是 NULL。

```
select '' ;
+-------+
|  _c0  |
+-------+
| NULL  |
+-------+

set character.literal.as.string = true ;
select '' ;
+------+
| _c0  |
+------+
|      |
+------+
```

- 字符串常量被识别为 STRING，而不是 CHAR(n)。

```
explain select '';
-- 结果如下
expressions:
            expr: ''
            type: string
```

当你遇到以下的情况之一，可以考虑设置这个开关，但是注意一旦设置之后这几个行为都会改变：

- 涉及到比较操作时，空串 '' 是可以比较的，NULL 则不行。例如 JOIN 时的两个字段都是空串 ''='' 的结果为 TRUE，但是 NULL 无法用 = 或 != 命中。

```
 select  ''='' ;
+-------+
|  _c0  |
+-------+
| true  |
+-------+
```

- UDF 中输入空串和输入 NULL，很可能会导致结果不同. 因为部分 UDF 会对 NULL 进行特殊处理，例如当某个参数为 NULL 时会返回 NULL。如果是空串可能可以有结果输出。
- 返回结果有多余空格：返回类型是字符类型的 UDF，当默认数据类型是 CHAR(n) 时，可能导致返回结果有多余空格. 也可以尝试设置这个开关。

8.10.3. Oracle 兼容模式

- 在 Oracle 兼容模式方言下，NULL 和空串 '' 有以下特点：

  对于数据类型 CHAR(n)、VARCHAR2(n)，NULL 和空串 '' 都可以命中 IS NULL，在底层存储中 NULL 和空串 '' 一致。也就是 `NULL is ''` 且 `'' is NULL`。

  SQL 语句中的 WHERE 和 JOIN 条件中，NULL 和空 '' 都无法用 = 或 != 来命中。
- Oracle 兼容模式下，字符串常量会自动转化为 CHAR(n) 类型。可以用命令 `explain select '';` 验证。

8.10.3.1. 自动补充空格

- Oracle 兼容模式下，CHAR(n) 类型是定长字符串，当字符长度小于预定于长度时，会自动在右侧补全空格。

```
select cast('a' as char(10));
+-------------+
|     _c0     |
+-------------+
| a           |
+-------------+
```

变长字符类型 VARCHAR(n) 和 VARCHAR2(n) 则不会补齐空格。

```
select cast('a    ' as varchar(10));
+------+
| _c0  |
+------+
| a    |
+------+

select cast('a    ' as varchar2(10));
+--------+
|  _c0   |
+--------+
| a      |
+--------+
```

- 对 CHAR(n) 类型作比较时，忽略且仅忽略字符右侧空格，所以 CHAR(n) 的长度 n （或右侧空格的数量）不会影响比较结果。

  验证右侧空格数量是否影响比较结果

```
select cast('a' as char(10)) = cast('a    ' as char(10));
+-------+
|  _c0  |
+-------+
| true  |
+-------+
```

8.10.3.2. 字符串类型转化

Oracle 兼容模式下，CHAR(n) 和 VARCHAR(n) 可以进行相关转换。

- CHAR(n) 转 VARCHAR(n) 时，会按照 CHAR(n) 补全之后的值，转到 VARCHAR(n)。
- VARCHAR(n) 转 CHAR(n) 时，若 VARCHAR(n) 的字段长度不足，CHAR(n) 会自动以字段右侧空格补足。

8.10.4. DB2 兼容模式

- DB2 兼容模式下，CHAR(n) 和 VARCHAR2(n) 忽视且仅忽视右侧空格。

```
set server.dialect=db2;
set client.dialect=db2;

select cast('a' as char(10)) = cast('a    ' as varchar2(10));
+-------+
|  _c0  |
+-------+
| true  |
+-------+
```

- DB2 兼容模式下，NULL 和空 ’’ 不同，也就是 `'' is not NULL'`。

```
select '' is not null;
+-------+
|  _c0  |
+-------+
| true  |
+-------+
```

- DB2 兼容模式下，空值 '' 可以用 = 或 != 命中，但 NULL 则不能。

```
select ''='';
+-------+
|  _c0  |
+-------+
| true  |
+-------+
```

- DB2 兼容模式下，JOIN 的字段如果为 NULL 则会报错。
- DB2 兼容模式下，字符串常量的默认类型是 VARCHAR2(n)。

```
explain select '';
--结果如下
expressions:
            expr: ''
            type: varchar2(65535,DB2)
```

8.10.5. TD 兼容模式

> TD 方言中，Inceptor 会自动将 VARCHAR(n) 数据类型转化成 VARCHAR2(n)。

- TD 兼容模式下，CHAR(n) 和 VARCHAR2(n) 忽视且仅忽视右侧空格。

```
select cast('a' as char(10)) = cast('a    ' as varchar2(10));
+-------+
|  _c0  |
+-------+
| true  |
+-------+
```

- TD 兼容模式下，NULL 和空 '' 不同，也就是 `'' is not NULL'`。
- TD 兼容模式下，空值 '' 可以用 = 或 != 命中，但 NULL 则不能。
- TD 兼容模式下，JOIN 的字段如果为 NULL 则会报错。
- TD 兼容模式下，字符串常量的默认类型是 VARCHAR2(n)，可以用命令 `explain select '';` 验证。

```
explain select '';
--结果如下:
expressions:
         expr: ''
         type: varchar2(65535,TD)
```

8.11. PL/SQL 指南（Oracle 方言）

Inceptor PL/SQL 是兼容 Oracle PL/SQL 的过程语言。在使用 Inceptor PL/SQL 和 Inceptor 进行交互前，请确保您打开 Oracle 方言开关。

8.11.1. Inceptor PL/SQL入门

Inceptor 高度支持 Oracle PL/SQL 语法，我们将在本手册中介绍 PL/SQL 在 Inceptor 中的使用。以下是本手册内容一览。

- **基础知识**

  常量，变量，%type 属性类型，%rowtype 属性类型等的声明，以及赋值方式及规范。

- **数据类型**

  数据类型可分为标量类型和复合类型。其中标量类型主要有 string，int，double 等 Inceptor 所支持的所有数据类型。复合类型主要有 Records，Collections，Cursors。

- **创建 PL/SQL 语句块**

  包括匿名块，过程和函数。

- **流程控制语句**

  IF，LOOP，WHILE，FOR，EXIT WHEN，CONTINUE，GOTO。

- **PL/SQL 存储过程**

  可分为带参数和不带参数的过程。

- **PL/SQL 函数**

  PL/SQL 函数可以单独调用，也可以在过程和 SQL 语句中调用。

- **Records**

  Records 中分量的名字和类型可以自定义，也可以基于表中字段的名字和类型来定义，此外还可以利用 %rowtype 属性快速定义一个 Records 变量。

- **Collections**

  collections，即集合元素，主要可分为 VARRAY，NESTED TABLE，Associative arrays 三种形式。

- **Cursors**

  Inceptor 中游标主要分为静态游标和动态游标，其中静态游标可分为显式游标和隐式游标；动态游标可分为强类型动态游标和弱类型动态游标。

- **与 SQL 的交互**

  本节中，我们分别介绍 PL/SQL 中的过程，函数，游标，以及 BULK COLLECT 方法与 SQL 的交互。

- **Packages**

  本节中主要介绍包头（Packages specification）和包体（Packages body）的创建方法，以及 Packages 的使用案例。

- **异常**

  支持的系统预定义异常：

  NO_DATA_FOUND, TOO_MANY_ROWS, CURSOR_ALREADY_OPEN, ROWTYPE_MISMATCH, SUBSCRIPT_BEYOND_COUNT, SUBSCRIPT_OUTSIDE_LIMIT, COLLECTION_IS_NULL, INVALID_CURSOR，此外 Inceptor 也支持用户自定义异常。

暂不支持的系统预定义异常：

INVALID_NUMBER, VALUE_ERROR, ZERO_DIVIDE, DUP_VAL_ON_INDEX, CASE_NOT_FOUND, ACCESS_INTO_NULL, SELF_IS_NULL, SYS_INVALID_ROWID, NOT_LOGGED_ON, LOGIN_DENIED, TIMEOUT_ON_RESOURCE, STORAGE_ERROR, PROGRAM_ERROR

- 预定义函数/过程/包

  sqlcode(void)

  sqlerrm(void)

  get_columns(string,nestedtable<string>)

  raise_application_error(int,string,bool)

  set_env(string,string)

  get_env(string)

  put_line(string)

  sqlerrm(int)

  dbms_output

  owa_util

- 注意事项

  Inceptor对 PL/SQL 中分号的支持

  PL/SQL 中结果的打印

  标识符

  标准 SQL 调用 PL/SQL 函数必须满足的条件

  不支持 RETURN INTO 语句

  查看预定义函数/过程/包

  PL/SQL 中的 PUT_LINE 打印

  Debug

  Hive 异常的处理

8.11.2. Inceptor PL/SQL指南中的表

在 Inceptor_plsql 指南中，创建了几张表用于演示操作，以下是表的具体内容

- 交易信息表 transactions：

  其中 trans_id，acc_num，trans_time，trans_type，stock_id 均为 string 类型，price 和 amount 均为 double 类型。

trans_id	acc_num	trans_time	trans_type	stock_id	price	amount
943197522	6513065	20140105100520	b	AA7105670	12.13	200
929634984	3912384	20140205140521	b	UA1467891	11.11	300

trans_id	acc_num	trans_time	trans_type	stock_id	price	amount
499506900	6513065	20140506133109	s	CA2789982	6.12	100
209441379	3912384	20140430111523	s	CX5397790	4.50	1000
648230055	0700735	20140315111111	s	DT7966575	22.66	200
719753265	3912384	20140328102400	b	BY8490909	68.43	100
975639131	0700735	20140611102830	s	AT6934136	5.30	200
991691937	2755506	20140702113025	s	VR2575735	7.52	1300
289018112	6513065	20140916105811	b	UT7592045	9.81	500
162742112	2394923	20141031135018	s	UC1610649	12.21	500
597565609	3912384	20140214141519	s	IU1775004	4.16	600
459590958	0700735	20140430143020	b	XJ9717497	5.25	1000
594819547	5224133	20140801110003	b	GL2547626	6.36	800
895916502	6513065	20141225133500	s	KC9102028	7.49	1100
900192386	6670192	20141130113905	s	XC1915304	8.64	900
952639648	6670192	20140314145958	s	CP7629713	10.31	400
404905188	6513065	20140628133001	b	SH6277444	7.02	100
952110653	6600641	20140228140005	s	GH6828501	9.16	100
817414815	5224133	20140331115900	s	ZX5373511	10.03	800
213859826	6513065	20140508094805	b	CL2121979	18.38	700

- 用户信息表 user_info：

name	acc_num	password	citizen_id	bank_acc	reg_date	acc_level
马**	6513065	115591	14**7	960*41	20110101	A
祝**	6670192	205239	23**8	737*71	20100101	C

name	acc_num	password	citizen_id	bank_acc	reg_date	acc_level
华*	5224133	531547	42**1	326*07	20080214	B
魏**	3912384	841242	52**4	685*48	20091202	A
宁**	4580952	986634	42**7	977*76	20081031	D
邱*	0700735	737297	34**8	143*18	20121024	A
李*	8725869	600709	46**3	430*84	20130702	E
潘**	6600641	990590	51**5	484*08	20110430	C
李**	2755506	015859	31**9	424*37	20110916	D
管**	2394923	783438	33**4	999*74	20141003	C

- HOLODESK 非分区表 zara：

name	age	gpa
lisi	20	2.0
wangwu	22	3.9
smith	22	3.1
zhaoliu	23	2.0
sara	21	3.8
zhangsan	23	2.0
lily	29	3.6

- HOLODESK 非分区表 za：

name	age
john	20
fd	24
zz	10
alice	25

- HOLODESK 非分区表 ra：

name	gpa
smith	3.4
lisi	3.3
ll	3.3
lily	3.6

- HOLODESK 非分区表 holotest2：

age	gpa
22	4.3
20	4.0
21	3.9

- HOLODESK 分区表 zza：

name	age	level（分区键）
zhaoliu	23	A
john	30	A
john	20	B
wangwu	22	C
lily	30	C
alice	25	C
john	20	C
sara	21	C

8.11.3. 基础知识

8.11.3.1. 声明

在 PL/SQL 中，可以用常量和变量存储值，在程序运行过程中，变量的值可以改变。
可以在任何 PL/SQL 语句块，子过程，包等的声明部分去声明一个常量或者变量。

8.11.3.1.1. 变量

PL/SQL 中，可以声明一个字符串，整数等类型的变量。PL/SQL 语句块或子程序运行时，变量都会被初始化，默认情况下，也就是变量没有被赋初值的时候，变量会被初始化成 NULL 值。

例 97. 变量的声明

```
!set plsqlUseSlash true
DECLARE
   V1 STRING
   V2 STRING:='字符串变量'
   V3 INT
   V4 INT:=100
   V5 BOOLEAN
   V6 BOOLEAN:='TRUE'
BEGIN
   DBMS_OUTPUT.PUT_LINE(V1)
   DBMS_OUTPUT.PUT_LINE(V2)
   DBMS_OUTPUT.PUT_LINE(V3)
   DBMS_OUTPUT.PUT_LINE(V4)
   DBMS_OUTPUT.PUT_LINE(V5)
   DBMS_OUTPUT.PUT_LINE(V6)
END;
/
```

输出结果为：

output
null
字符串变量
null
100
null
TRUE

8.11.3.1.2. 常量

声明一个常量，只需在特定的数据类型前加一个关键字“CONSTANT”。PL/SQL 语句块或子程序运行时，常量都会被初始化，默认情况下，也就是常量没有被赋初值的时候，此时常量会被初始化成 NULL 值。

例 98. 常量的声明

```
!set plsqlUseSlash true
DECLARE
   P1  CONSTANT STRING
   P2  CONSTANT STRING := '字符串常量'
   P3  CONSTANT INT
   P4  CONSTANT INT:=100
   P5  CONSTANT BOOLEAN
   P6  CONSTANT BOOLEAN:='TRUE'
BEGIN
   DBMS_OUTPUT.PUT_LINE(P1)
   DBMS_OUTPUT.PUT_LINE(P2)
   DBMS_OUTPUT.PUT_LINE(P3)
   DBMS_OUTPUT.PUT_LINE(P4)
   DBMS_OUTPUT.PUT_LINE(P5)
   DBMS_OUTPUT.PUT_LINE(P6)
END;
/
```

输出结果为：

output
null
字符串常量
null
100
null
TRUE

8.11.3.1.3. DEFAULT

初始化变量时，可以使用关键字“DEFAULT”来赋值，作用与赋值操作符“ := ”完全相同，关键字“DEFAULT”会使变量在声明的过程中，会使有一个初始值，但是在程序体中可以对该变量进行再一次的赋值。

例 99. 用关键字“DEFAULT”给变量赋默认值

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个字符串类型的变量 V1，并附初始值为 HELLO。
   V1 STRING DEFAULT 'HELLO'
   V2 INT DEFAULT 100
   V3 BOOLEAN DEFAULT FALSE
BEGIN
   -- 在程序体内对变量 V1，进行再一次的赋值，再次打印变量 V1 的值，会发现此时变量 V1 的值为 hello world。
   V1 :='hello world'
   DBMS_OUTPUT.PUT_LINE(V1)
   DBMS_OUTPUT.PUT_LINE(V2)
   DBMS_OUTPUT.PUT_LINE(V3)
END;
/
```

① 声明一个字符串类型的变量 V1，并附初始值为 HELLO。

② 在程序体内对变量 V1，进行再一次的赋值，再次打印变量 V1 的值，会发现此时变量 V1 的值为 hello world。

输出结果为：

output
hello world
100
FALSE

8.11.3.2. 命名规则

在 PL/SQL 中的常量，变量，游标，游标变量，异常，过程，函数，包均使用相同的命名规格，可以直接在语句块的部分创建或声明，也可以在包内直接声明。

- 直接创建一个名为 proc1 的过程，参数名为 V1，参数属性为 IN，参数的数据类型为 STRING。

```
CREATE OR REPLACE  proc1(V1  IN STRING)
IS
BEGIN
...
END;
```

- 在包内声明一个名为 proc2 的存储过程，参数名为 test_name，参数属性为 OUT，参数的数据类型为 DOUBLE。

```
!set plsqlUseSlash true
CREATE OR REPLACE PACKAGE test
IS
   PROCEDURE proc2( test_name OUT DOUBLE)
END;
/
```

8.11.3.2.1. 重复命名

在相同的作用域内，所有声明的标识符都必须是唯一的。即Inceptor 不允许在相同作用域内重复命名。即使数据类型不同，变量和参数的名字也不能相同。

例 100. 同时声明两个变量，变量名相同，数据类型不相同

```
!set plsqlUseSlash true
DECLARE
   V1 BOOLEAN
   V1 INT
BEGIN
   V1:= FALSE
END;
/
```

输出结果为：

```
COMPILE FAILED: Semantic error: [Error 11108]
ANONYMOUS BLOCK (LINE 3, COLUMN 0, TEXT \"V1 INT\"): Existing variable v1
```

可以看到，如果在相同作用域内重复命名两个标识符，Inceptor 会报错。

8.11.3.2.2. 大小写敏感性

PL/SQL中，对于所有的标识符，如常量，变量，参数都是不区分大小写的。所以在命名时，不可以通过改变标识符大小写的方式来命名多个不同的标识符。

例 101. 同时声明两个变量，变量名字母的大小写形式不同

```
!set plsqlUseSlash true
DECLARE
   zip_code INT
   ZIP_CODE INT
BEGIN
   zip_code:= 90120
END;
/
```

输出结果为:

```
COMPILE FAILED: Semantic error: [Error 11108]
ANONYMOUS BLOCK (LINE 3, COLUMN 0, TEXT \"ZIP_CODE INT\"): Existing variable zip_code
```

可以看到，尽管变量名的大小写形式不同，Inceptor 仍然识别为相同的变量名。

8.11.3.2.3. 命名解析

在 PL/SQL 中，变量名优先于数据库中表的列名，例如，在WHERE子句中如果变量名和表的列名相同，Inceptor 会认为两个都是变量名。

例 102. 删除 zara 表中某个存在的 gpa

```
!set plsqlUseSlash true
DECLARE
gpa DOUBLE default 4.5
BEGIN
   DELETE FROM zara WHERE gpa=gpa
   DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.')
END;
/
```

声明一个 DOUBLE 类型的变量 gpa，赋初值为 4.5，且表中存在一条记录的 gpa 为 4.5。

删除表 zara 中，列 gpa 的值为变量 gpa 的值，此处列名和变量名相同，Inceptor 会认为两个都是变量名，也就是说此时 where 条件为 4.5=4.5。

打印出执行上述 SQL 语句，一共删除了表 zara 中多少行的数据，由于 DELETE 语句中的 where 条件为 true，此处应该删除 zara 表中的全部记录。

输出结果为:

output
Deleted 7 rows.

例 103. 删除 zara 表中某个不存在的 gpa

```
!set plsqlUseSlash true
DECLARE
gpa DOUBLE default 2.0
BEGIN
   DELETE FROM zara WHERE gpa=gpa
   DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.')
END;
/
```

声明一个 DOUBLE 类型的变量 gpa，赋初值为 2.0，且表中不存在 gpa 为 4.5 的记录。

删除表 zara 中，列 gpa 的值为变量 gpa 的值，此处列名和变量名相同，Inceptor 会认为两个都是变量名，也就是说此时where条件为2.0=2.0。

打印出执行上述 SQL 语句，一共删除了表 zara 中多少行的数据，由于 DELETE 语句中的 where 条件为 true，此处应该删除 zara 表中的全部记录。

8.11.3.2.4. 嵌套命名

Inceptor 中如果在相同作用域内重复命名，会出错，但是可以在两个不同的作用域去声明两个相同的变量名，改变其中一个变量的值，不会影响另一个。即 Inceptor 中支持嵌套命名。

例 104. 嵌套命名

```
!set plsqlUseSlash true
DECLARE
   V1 STRING  -- 声明一个字符串类型的变量，名为 V1。
   V2 INT
BEGIN
   V1 := 'HELLO' -- 给变量 V1 赋值。
   V2 := 123
   DBMS_OUTPUT.PUT_LINE(V1)
   DBMS_OUTPUT.PUT_LINE(V2)

   DECLARE
     V1 BOOLEAN  -- 声明一个布尔值类型的变量，名为 V1。
     V3 DOUBLE
   BEGIN
     V1 := FALSE  -- 给变量 V1 赋值。
     V3 := 12.13
     DBMS_OUTPUT.PUT_LINE(V1)
     DBMS_OUTPUT.PUT_LINE(V3)
  END

DECLARE
   V4 STRING
BEGIN
   V4 := 'World'
   DBMS_OUTPUT.PUT_LINE(V4)
END
END;
/
```

输出结果为：

output
HELLO
123
FALSE
12.13
World

8.11.3.3. 赋值

在 PL/SQL 中，可以在声明变量的同时赋予初始值，也可以在声明变量之后，对变量进行赋值。在语句块或者子程序运行的时候，变量会被初始化，默认状况，即没有给变量赋值的情况下，变量会被初始化成 NULL 值。

例 105. 声明三个不同类型的变量

```
!set plsqlUseSlash true
DECLARE
   name STRING := 'ZHANGSAN'
   age INT
   grade DOUBLE
BEGIN
   age := 23
   DBMS_OUTPUT.PUT_LINE(name)
   DBMS_OUTPUT.PUT_LINE(age)
IF grade IS NULL THEN
   DBMS_OUTPUT.PUT_LINE('grade is NULL.')
END IF
END;
/
```

输出结果为:

output
ZHANGSAN
23
grade is NULL.

8.11.3.3.1. 赋予变量逻辑值

对于一个布尔值类型的变量，只能赋予其 FALSE，TRUE，或者 NULL 值。

例 106. 将逻辑值赋值给变量

```
!set plsqlUseSlash true
DECLARE
   done BOOLEAN
   counter INT:= 0
BEGIN
   done := FALSE
WHILE done != TRUE
LOOP
   counter := counter + 1
   done := (counter > 500)
END LOOP;
DBMS_OUTPUT.put_line ( 'the counter is:' || counter)
END;
/
```

输出结果为:

output
the counter is:501

8.11.3.3.2. 赋予变量查询结果

在 Inceptor 中可以使用 select into 语句将查询结果赋值给变量。

例 107. 将查询结果赋值给变量

```
!set plsqlUseSlash true
DECLARE
   id STRING
   time STRING
BEGIN
   SELECT trans_id,trans_time
      INTO id,time
      FROM transactions
      WHERE price=12.13;
   DBMS_OUTPUT.PUT_LINE('the id is:'||id)
   DBMS_OUTPUT.PUT_LINE('the time is:'||time)
END;
/
```

输出结果为:

output
the id is:943197522
the time is:20140105100520

8.11.4. 数据类型

8.11.4.1. 标量类型

标量类型的含义是存放单个值。Inceptor 中支持的标量类型为 INT/STRING/DOUBLE 等 Inceptor 中所有支持的数据类型。具体内容可参考《数据类型对应表》。

8.11.4.1.1. STRING

在 Inceptor 中 STRING 既可以是字符串常量，也可以是字符串变量。对于字符串常量，要用单引号引起来；对于字符串变量，要在 PL/SQL 语句块中的 DECLARE 部分声明。

例 108. 对于字符串常量，可以直接引用

```
!set plsqlUseSlash true
BEGIN
   DBMS_OUTPUT.put_line ('Hello World')
END;
/
```

输出结果为:

output
Hello World

例 109. 对于字符串变量，要事先声明

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个 STRING 类型的变量为 l_message。
   l_message STRING
BEGIN
   -- 给变量 l_message 赋值。
   l_message:= 'Hello World'
   -- 输出变量 l_message 的值。
   DBMS_OUTPUT.put_line (l_message)
END;
/
```

输出结果为：

output
Hello World

8.11.4.1.2. INT

在 Inceptor 中 INT 为整数类型，可以直接使用一些整数，也可以在 PL/SQL 语句块的 DECLARE 部分声明一个 INT 类型的变量，用于存放 INT 类型的数据。

例 110. 直接使用整数

```
!set plsqlUseSlash true
BEGIN
   DBMS_OUTPUT.put_line (100)
END;
/
```

输出结果为：

output
100

例 111. 声明一个 INT 类型的变量

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个 INT 类型的变量为 l_int
   l_int int
BEGIN
   -- 给变量 l_int 赋值
   l_int:= 1000
   -- 输出变量 l_int 的值
   DBMS_OUTPUT.put_line (l_int)
END;
/
```

输出结果为：

output
1000

此外，Inceptor中所有支持的数据类型都可以在 PL/SQL 语句块的 DECLARE 部分中声明，此处不再赘述。

8.11.4.2. %TYPE 属性

使用 %TYPE 属性，可以声明一个变量，其数据类型与已经存在的变量，记录类型，表变量类型或者数据表的字段的数据类型相同，但新声明的变量不会继承已经存在的变量的值。

例 112. 使用 %TYPE 属性声明变量，其数据类型为另一个变量的类型

```
!set plsqlUseSlash true
DECLARE
   name string  := 'JoHn SmItH'
   upper_name name%TYPE := UPPER(name)
   lower_name name%TYPE := LOWER(name)
BEGIN
   DBMS_OUTPUT.PUT_LINE ('name: ' || name)
   DBMS_OUTPUT.PUT_LINE ('upper_name: ' || upper_name)
   DBMS_OUTPUT.PUT_LINE ('lower_name: ' || lower_name)
END;
/
```

输出结果为:

output
name: JoHn SmItH
upper_name: JOHN SMITH
lower_name: john smith

例 113. 使用 %TYPE 属性声明变量 其数据类型为表中某字段的数据类型

```
!set plsqlUseSlash true
DECLARE
   v_acc_num transactions.acc_num%TYPE
   v_trans_id transactions.trans_id%TYPE
   v_price transactions.price%TYPE
BEGIN
   SELECT acc_num
     INTO v_acc_num
     FROM transactions
     WHERE stock_id='AA7105670';
   SELECT trans_id
     INTO v_trans_id
     FROM transactions
     WHERE trans_time='20140105100520';
   v_price:='11.11'
   DBMS_OUTPUT.PUT_LINE('acc_num:'|| v_acc_num)
   DBMS_OUTPUT.PUT_LINE('trans_id:'|| v_trans_id)
   DBMS_OUTPUT.PUT_LINE('price:'|| v_price)
END;
/
```

输出结果为:

output
acc_num:6513065
trans_id:943197522
price:11.11

8.11.4.3. %ROWTYPE 属性

使用 %ROWTYPE 属性，可以声明一个记录型的变量，其记录型变量的分量名和类型与表/视图/游标中字段名和类型均相同，同样，记录型变量不会继承表/视图/游标中各个字段的值。
如果相应的表/视图发生变动，如表中新增字段或有字段被删除，则所声明的变量的也会发生相应的变化。

例 114. 利用 %ROWTYPE 属性声明记录型变量 trans，trans 变量的分量名和类型与表 transactions 中的字段名和类型相同

```
!set plsqlUseSlash true
DECLARE
   trans transactions%ROWTYPE
BEGIN
   SELECT *
     INTO trans
     FROM transactions
     WHERE stock_id='AA7105670';
   DBMS_OUTPUT.PUT_LINE('price:'||trans.price)
   DBMS_OUTPUT.PUT_LINE('trans_time:'||trans.trans_time)
   DBMS_OUTPUT.PUT_LINE('trans_type:'||trans.trans_type)
END;
/
```

输出结果为：

output
price:12.13
trans_time:20140105100520
trans_type:b

8.11.4.4. 复合类型

复合类型，即内部存在分量的数据类型，主要有 Records 和 Collections，需要用户自己去定义该类型，具体内容可参考后文的 Records 和 Collections 章节。

8.11.5. 创建 PL/SQL 语句块

PL/SQL 语句块可以是一个没有名字的语句块，也可以是命名的语句块（即过程和函数）。本章节中我们主要讲述匿名块的创建过程。

8.11.5.1. PL/SQL 语句块组成部分

PL/SQL 块由四个基本部分组成：声明、执行体开始、异常处理、执行体结束。

- DECLARE —— 可选部分

  变量、常量、函数游标、用户定义异常的声明。

- BEGIN —— 必要部分

  SQL 语句和 PL/SQL 语句构成的执行程序。

- EXCEPTION —— 可选部分

  程序出现异常时，捕捉异常并处理异常。

- END —— 必须部分

- 在数据库执行 PL/SQL 程序时，PL/SQL 语句和 SQL 语句是分别进行解析和执行的。
- PL/SQL 块被数据库内部的 PL/SQL 引擎提取，将 SQL 语句取出送给 Inceptor 的 SQL 引擎处理，两种语句分别在两种引擎中分析处理，在数据库内部完成数据交互、处理过程。

8.11.5.2. 最简单的匿名块

最简单的匿名块只包含 PL/SQL 语句块中的两个必要部分，BEGIN 和 END。

例 115. 创建最简单的匿名块

```
!set plsqlUseSlash true
BEGIN
   DBMS_OUTPUT.put_line ('Hello World!') -- 输出信息 Hello World
END;
/
```

输出结果:

output
Hello World!

8.11.5.3. 稍复杂的匿名块

稍复杂的匿名块除了包含 PL/SQL 语句块中的两个必要部分，BEGIN 和 END 之外，还包含一个 DECLARE 的部分，即声明变量的部分。

例 116. 创建稍复杂的匿名块

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个 string 类型的变量 l_message，并赋值。
   l_message STRING := 'Hello World!'
BEGIN
   -- 输出 l_message 的值。
   DBMS_OUTPUT.put_line (l_message)
END;
/
```

输出结果:

output
Hello World!

8.11.5.4. 完整的匿名块

一个完整的匿名块包含了 PL/SQL 语句块的四个部分分别为 DECLARE，BEGIN，EXCEPTION，END。

例 117. 创建完整的匿名块

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个 string 类型的变量 transid。
   transid STRING;
BEGIN
   -- 查询 transactions 表中的交易号，并放进变量 transid 中，此处 transactions 后需加上表的别名。
   SELECT trans_id
   INTO transid
   FROM transactions t1;
   -- 输出变量 transid 的值。
   PUT_LINE(transid)
EXCEPTION
   -- 引用一个情况名为 too_many_rows 的异常。
   WHEN too_many_rows
   THEN
   -- 当异常发生时的所输出的信息。
   DBMS_OUTPUT.put_line ('too many rows')
END;
/
```

输出结果：

output
too many rows

SELECT 或 SELECT INTO 语句后如果跟函数调用或赋值语句，SELECT 或 SELECT INTO 语句不能以 FROM <TABLE_NAME> 结尾，否则函数名会被识别为 alias，Inceptor 会报语义错误。更多用法说明及解决方法可参考注意事项。

8.11.5.5. 匿名块里的嵌套

- 通过 Beeline 的方式连接 InceptorServer 2，并利用如下命令手动打开分号支持。

```
!set plsqlUseSlash true
```

例 118. 匿名块里的嵌套

```
!set plsqlUseSlash true
DECLARE
   -- 声明外部语句块中的变量 test1。
   test1 STRING
BEGIN
   DECLARE
     -- 声明内部语句块中的变量 test2。
     test2 STRING
   BEGIN
     -- 查询 transactions 表中账号为 6513064 的交易号，并放进变量 test2 中。
     SELECT trans_time
       INTO test2
       FROM transactions
       WHERE acc_num=6513064;
     -- 输出变量 test2 的值。
     dbms_output.put_line(test2)
   EXCEPTION
     -- 引用一个 NO_DATA_FOUND 的异常情况名。
     WHEN NO_DATA_FOUND THEN
     dbms_output.put_line('no values')
   END
-- 查询 transactions 表中价格为 12.13 的交易号，并放进变量 test1 中。
SELECT trans_id
   INTO test1
   FROM transactions
   WHERE price=12.13;
-- 输出变量 test1 的值。
dbms_output.put_line(test1)
EXCEPTION
   -- 引用一个 TOO_MANY_ROWS 的异常情况名。
   WHEN TOO_MANY_ROWS THEN
   dbms_output.put_line('too many rows')
END;
/
```

输出结果:

内部语句块中抛出异常发生时所返回的值，外部语句块中成功输出变量test1的值。

output
no values
943197522

- Inceptor中对PL/SQL语句块的分号支持默认是不开启的，我们可以通过命令语句手动打开。

  Beeline+InceptorServer 2中通过以下命令:!set plsqlUseSlash true

- 更多用法说明，可参见后文中的注意事项章节。

8.11.6. 流程控制语句

Inceptor 中的流程控制语句主要可分为以下三类，具体的使用方法可参考以下案例。

- 控制语句：IF 语句
- 循环语句：LOOP 语句，WHILE 语句，FOR 语句，FORALL 语句，EXIT 语句
- 顺序语句：GOTO 语句

8.11.6.1. IF

8.11.6.1.1. IF-THEN

- 语法：

```
IF condition THEN
  {...statements...}
END IF;
```

例 119. 查询 transactions 表中交易号为 648230055 的价格

```
!set plsqlUseSlash true
DECLARE
   -- 分别声明两个 DOUBLE，STRING 类型的变量 v_price，v_comment。
   v_price DOUBLE
   v_comment STRING
BEGIN
   -- 查询 transactions 表中交易号为 648230055 的价格，并将查询的结果放进变量 v_price 里。
   SELECT price
     INTO v_price
     FROM transactions
     WHERE trans_id=648230055;
   -- 输出 v_price 的值。
   dbms_output.put_line('the price is:'||v_price)
   -- 确定 IF 的条件语句为 v_price 的值大于 20。
   IF v_price > 20 THEN
      -- 在满足 IF 的条件语句的情况下，给变量 v_comment 赋值。
      v_comment := 'the price is too high'
   END IF
   -- 输出变量 v_comment 的值。
   dbms_output.put_line(v_comment)
END;
/
```

输出结果为：

output
the price is:22.66
the price is too high

8.11.6.1.2. IF-THEN-ELSE

- 语法：

```
IF condition THEN
  {...statements...}
ELSE
  {...statements...}
END IF;
```

例 120. 查询 transactions 表中交易号为 943197522 的价格

```
!set plsqlUseSlash true
DECLARE
   -- 分别声明两个 DOUBLE，STRING 类型的变量 v_price，v_comment。
   v_price DOUBLE
   v_comment string
BEGIN
   -- 查询 transactions 表中交易号为 943197522 的价格，并将查询的结果放进变量 v_price 里。
   SELECT price
     INTO v_price
     FROM transactions
     WHERE trans_id=943197522;
   -- 输出 v_price 的值。
   dbms_output.put_line('the price is:'||v_price)
   -- 确定 IF 的条件语句为 v_price 的值大于 20。
   IF v_price > 20 THEN
      -- 在满足 IF 的条件语句的情况下，给变量 v_comment 赋值。
      v_comment := 'the price is too high'
   ELSE
      -- 在不满足 IF 的条件语句的情况下，给变量 v_comment 赋值。
      v_comment := 'the price is not too high'
   END IF
   -- 输出变量 v_comment 的值。
   dbms_output.put_line(v_comment)
END;
/
```

输出结果为：

output
the price is:12.13
the price is not too high

8.11.6.1.3. IF-THEN-ELSIF

- 语法：

```
IF condition THEN
  {...statements...}
ELSIF condition THEN
  {...statements...}
ELSE
  {...statements...}
END IF;
```

例 121. 查询 transactions 表中交易号为 943197522 的价格

```
!set plsqlUseSlash true
DECLARE
   -- 声明两个类型分别为 DOUBLE、STRING 的变量 v_price、v_comment。
   v_price DOUBLE
   v_comment string
BEGIN
   -- 查询 transactions 表中交易号为 943197522 的价格，并将查询的结果放进变量 v_price 里。
   SELECT price
      INTO v_price
      FROM transactions
      WHERE trans_id=943197522;
    -- 输出 v_price 的值。
    dbms_output.put_line('the price is:'||v_price)
    -- 确定 IF 的条件语句为 v_price 的值大于 15。
    IF v_price > 15 THEN
       -- 在满足 IF 的条件语句的情况下，给变量 v_comment 赋值。
       v_comment := 'the price 大于 15'
    -- 确定 IF 的条件语句为 v_price 的值大于10。
    ELSIF v_price > 10 THEN
       -- 在满足第一个 ELSIF 的条件语句的情况下，给变量 v_comment 赋值。
       v_comment := 'the price 大于 10'
    -- 确定第二个 ELSIF 的条件语句为 v_price 的值大于 5。
    ELSIF v_price > 5 THEN
       -- 在满足第二个 ELSIF 的条件语句的情况下，给变量 v_comment 赋值。
       v_comment := 'the price 大于 5'
    ELSE
       -- 在以上 IF 和 ELSIF 的条件语句都不满足的情况下，给变量 v_comment 赋值。
       v_comment := 'the price is too low'
    END IF
    -- 输出变量 v_comment 的值。
    dbms_output.put_line(v_comment)
END;
/
```

输出结果为：

output
the price is:12.13
the price 大于 10

例 122. 查询 transactions 表中交易号为 209441379 的价格

```
!set plsqlUseSlash true
DECLARE
   v_price STRING
   v_comment string
BEGIN
   SELECT price
     INTO v_price
     FROM transactions
     WHERE trans_id=209441379;
   dbms_output.put_line('the price is:'||v_price)
   IF v_price > 15 THEN
      v_comment := 'the price 大于 15'
   ELSIF v_price > 10 then
      v_comment := 'the price 大于10'
   ELSIF v_price > 5 then
      v_comment := 'the price 大于5'
   ELSE
      v_comment := 'the price is too low'
   END IF
   dbms_output.put_line(v_comment)
END;
/
```

输出结果为：

output
the price is:4.5
the price is too low

8.11.6.2. LOOP

the simple loop

- 语法：

```
LOOP
   {...statements...}
   EXIT WHEN  conditions -- 条件满足，退出循环语句
END LOOP;
```

例 123. the simple loop

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个整数类型的变量 test，并附初值为 0。
   test int := 0
BEGIN
   LOOP
     -- 在 0 的基础上，给 test 的值不断加 1。
     test := test + 1
     -- 输出当前 test 的值。
     dbms_output.put_line('test当前值为:'||test)
     -- 确定退出循环语句的条件语句为，当 test 的值为 10 时。
     EXIT WHEN test=10
   END LOOP;
END;
/
```

输出结果为：

output
test当前值为:1
test当前值为:2
test当前值为:3
test当前值为:4
test当前值为:5
test当前值为:6
test当前值为:7
test当前值为:8
test当前值为:9
test当前值为:10

FOR LOOP 后面的 END LOOP 后可以接一个可选的标识符，如果 END LOOP 后面紧跟函数调用或赋值语句，则 Inceptor 也会报出语义错误，更多用法说明即解决方法可参见注意事项章节。

8.11.6.3. WHILE

- 语法：

```
WHILE conditions
LOOP
  {...statements...}
END LOOP;
```

例 124. the while

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个整数类型的变量 test，并附初值为 0。
   test int := 0
BEGIN
   -- 确定循环条件的 while 条件语句。
   WHILE test<10
   LOOP
     -- 输出当前 test 的值。
     dbms_output.put_line('test当前值为:'||test)
     -- 在符合 WHILE 条件语句的情况下，当前 test 值的基础上，不断加 1。
     test := test + 1
   END LOOP;
END;
/
```

输出结果为：

output
test当前值为:0
test当前值为:1
test当前值为:2
test当前值为:3
test当前值为:4
test当前值为:5
test当前值为:6
test当前值为:7
test当前值为:8
test当前值为:9

8.11.6.4. FOR

- 语法：

```
FOR 循环计数器 IN [ REVERSE ] 下限 .. 上限
LOOP
  {...statements...}
[EXIT WHEN conditions]
END LOOP
```

- 每循环一次，循环变量自动加 1。
- 使用关键字 REVERSE，循环变量自动减 1。
- 跟在 IN REVERSE 后面的数字必须是从小到大的顺序，而且必须是整数，不能是变量或表达式。
- 可以使用 EXIT 退出循环。

例 125. FOR IN

```
!set plsqlUseSlash true
BEGIN
   -- 对于 1 到 5 之间的整数。
   FOR test IN 1..5 LOOP
     -- 从 1 开始，以每次自动加 1 的方式，依次输出 test 的值。
     dbms_output.put_line('test当前值为:'||test)
   END LOOP;
END;
/
```

输出结果为：

output
test当前值为:1
test当前值为:2
test当前值为:3
test当前值为:4
test当前值为:5

例 126. FOR IN REVERSE

```
!set plsqlUseSlash true
BEGIN
   -- 对于 1 到 5 之间的整数。
   FOR test IN REVERSE 1..5  LOOP
      -- 从 5 开始，以每次自动减 1 的方式，依次输出当前值。
      dbms_output.put_line('test当前值为:'||test)
      END LOOP;
END;
/
```

输出结果为：

output
test当前值为:5
test当前值为:4
test当前值为:3
test当前值为:2
test当前值为:1

例 127. 有循环退出的条件

```
!set plsqlUseSlash true
BEGIN
   -- 对于 1 到 5 之间的整数。
   FOR test IN REVERSE 1..5   LOOP
      -- 从 5 开始，以每次自动减 1 的方式，依次输出当前值。
      dbms_output.put_line('test当前值为:'||test)
      -- 循环退出的条件为 当 test 的值为 3 时。
      EXIT WHEN test=3
   END LOOP;
END;
/
```

输出结果为:

output
test当前值为:5
test当前值为:4
test当前值为:3

8.11.6.5. FORALL

FORALL 循环语句的内部只能是一条 DML 语句，如 select、insert、update、delete 等，在其它条件下使

用，是不支持的。

8.11.6.5.1. 例一：使用 FORALL 往表中插入数据

例 128. 利用 forall 语句，往表 za 中批量插入数据

```
!set plsqlUseSlash true
DECLARE

    -- 声明一个名为 NumList 的 VARRAY 类型，可容纳 20 个整数。
    TYPE NumList IS VARRAY(20) OF INT
    -- 定义一个 NumList 类型的变量 depts，并赋初值。
    depts NumList := NumList(20,22,23)
BEGIN
    -- 使用 forall 语句，对于变量 depts 中的全部的值所对应的下标。
    FORALL i IN depts.FIRST()..depts.LAST()
       -- 往表 za 中插入数据，姓名为 lily,年龄分别为变量 depts 的值。
       INSERT INTO za VALUES('lily',depts(i))
    -- 条件判断语句为如果 sql 成功执行了。
    IF SQL%FOUND THEN
          -- sql 语句成功执行，所输出的信息。
          dbms_output.put_line('successfully insert new data')
    ELSE
          -- sql 语句没有成功执行，所输出的信息。
          dbms_output.put_line('nothing')
    END IF
END;
/
```

输出结果为：

output
successfully insert new data

8.11.6.5.2. 例二：使用 FORALL 查询表中数据

- 通过如下语句，手动设置，使打印查询结果。

```
set plsql.show.sqlresults=true;
```

例 129. 利用 forall 语句，查询表 transactions 中的交易时间

```
!set plsqlUseSlash true
set plsql.show.sqlresults=true;
DECLARE
   -- 声明一个名为 ORG_VARRAY_TYPE 的 VARRAY 类型，最多可容纳 10 个 DOUBLE 类型的数据。
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF double
   -- 定义一个 ORG_VARRAY_TYPE 类型的变量 V_ORG_VARRAY。
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   -- 给变量 V_ORG_VARRAY 赋值。
   V_ORG_VARRAY := ORG_VARRAY_TYPE('12.13','11.11','6.12','4.5','22.66')
   -- 使用 forall 语句，对于变量 V_ORG_VARRAY 中的全部的值。
   FORALL I IN V_ORG_VARRAY.FIRST()..V_ORG_VARRAY.LAST()
      -- 查询表 transactions 中价格和变量 V_ORG_VARRAY 中的值相等的全部交易时间。
      SELECT trans_time
         FROM transactions
         WHERE price=V_ORG_VARRAY(i);
END;
/
```

输出结果为：

output
20140105100520
20140205140521
20140506133109
20140430111523
20140315111111

- Inceptor 中默认对 SELECT 的查询结果不显示，要想显示 SELECT 的查询结果，可设置 set plsql.show.sqlresults=true;
- 更多用法说明，可参见后文的注意事项章节。

8.11.6.6. EXIT WHEN

EXIT 为 PL/SQL 中退出当前循环语句的条件语句，通常与 LOOP 语句在一起使用。

例 130. EXIT WHEN

```
!set plsqlUseSlash true
DECLARE
   test int := 0
BEGIN
   LOOP
      test := test + 1
      dbms_output.put_line('test当前值为: '||test)
   EXIT WHEN test=5
   END LOOP;
END;
/
```

输出结果:

output
test当前值为: 1
test当前值为: 2
test当前值为: 3
test当前值为: 4
test当前值为: 5

8.11.6.7. CONTINUE（WHEN）

CONTINUE 为 PL/SQL 中退出当前循环语句的条件语句，通常与 LOOP 语句在一起使用。

例 131. CONTINUE（WHEN）

```
!set plsqlUseSlash true
-- 创建一个不带参数的过程 continue_proc()。
CREATE OR REPLACE PROCEDURE continue_proc()
IS
DECLARE
   -- 分别定义两个 int 类型的变量，并给变量 var1 赋初值为 1。
   var1 INT:=1
   var2 INT
BEGIN
<<outer_loop>>
   LOOP
      -- 在 outer_loop 中，给变量 var2 赋初值 1。
      var2 := 1
      -- 在 outer_loop 中，输出变量 var1 当前的值，也就是初值为 1。
      dbms_output.put_line("Outer loop body " || var1)
      -- 在 outer_loop 中，给变量 var1 在当前值的基础上加上 1。
      var1 := var1 + 1
<<inner_loop>>
         LOOP
            -- 在 inner_loop 中，定义 continue，如果 var1 的值为 2，就继续 outer_loop。
            CONTINUE outer_loop WHEN var1 = 2
            -- 在 inner_loop 中，输出此时 var2 的值，也就是 var2 的初值为 1。
            dbms_output.put_line("Inner loop body " || var2)
            -- 在 inner_loop 中，给变量 var2 在当前值的基础上加上 1。
            var2 := var2 + 1
         -- 当 var2 的值为 3 时，就退出 inner_loop。
         EXIT inner_loop WHEN var2 = 3
         END LOOP;
         -- 当 var1 的值为 3 时，就退出 outer_loop。
         EXIT WHEN var1 = 3
   END LOOP outer_loop;
END;
/
BEGIN
   continue_proc()
END;
/
```

输出结果为：

output
Outer loop body 1
Outer loop body 2
Inner loop body 1
Inner loop body 2

8.11.6.8. GOTO

PL/SQL 中 GOTO 语句是无条件跳转到指定的标号去的意思，标签是用双箭头括号括起来的。当执行 GOTO 语句的时候，控制会立即转到由标签标识的语句。

- 语法：

```
GOTO label
... ...
<<label>>
```

例 132. 打印变量的值

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个 int 类型的变量 V_int，赋初值为 1。
   V_int INT:= 1
BEGIN
   LOOP
      -- 打印出变量 V_int 的当前值。
      DBMS_OUTPUT.PUT_LINE('V_int当前值为:'||V_int)
      -- 在变量 V_int 初值的基础上不断加1。
      V_int := v_int + 1
      -- 当变量 V_int 大于 5 时，跳转到标号 labelOffLOOP。
      IF v_int > 5 THEN
         GOTO labelOffLOOP
      END IF
   END LOOP;
   <<labelOffLOOP>>
   DBMS_OUTPUT.PUT_LINE('V_int当前值为:'||V_int)
END;
/
```

输出结果为：

output
V_int当前值为:1
V_int当前值为:2
V_int当前值为:3
V_int当前值为:4
V_int当前值为:5
V_int当前值为:6

例 133. 求和

```
!set plsqlUseSlash true
DECLARE
    -- 分别定义两个 int 类型的变量，并赋初值为 0。
    v_i INT := 0
    v_s INT:= 0
BEGIN
   <<label_1>>
    -- 变量 v_i 从零开始，不断加 1。
    v_i := v_i + 1
    -- 当 v_i 小于等于 1000 时，将两个变量的和赋值给变量 v_s。
    IF v_i <= 1000 THEN
       v_s := v_s + v_i
       -- 当 v_i 大于 1000 时，跳转到标号 label_1。
       GOTO label_1
    END IF
    -- 输出变量 v_s 的值。
    DBMS_OUTPUT.PUT_LINE(v_s)
END;
/
```

输出结果为：

output
500500

8.11.6.8.1. GOTO 语句的限制

- 一个标签后面至少跟着一个可执行语句。
- GOTO 语句的目标标签必须和 GOTO 语句在同一个作用域内。
- 对于块、循环或者 IF 语句而言，想要从外层跳转到内层是非法的，即不可以从外层跳转到内层。
- 从一个 IF 子句跳转到另一个子句中也是非法的，即不可以从一个子句跳转到另一个子句。
- 从一个异常处理块内部跳转到当前块是非法的，即不可以从异常处理块跳转到当前块。

8.11.6.9. CASE

例 134. 查询表 transactions 中账号为“6513065”的价格，并依次输出价格的状态。

```
!set plsqlUseSlash true
DECLARE
    -- 声明一个字符串类型的变量 message。
    message string
    -- 声明一个变量 p_price，数据类型为表 transactions中 价格字段的类型。
    p_price transactions.price%type
    -- 声明游标 cur 用来查询表 transactions 中账号为 6513065 的价格。
    CURSOR cur IS SELECT price FROM transactions WHERE acc_num='6513065';
BEGIN
    OPEN cur
    LOOP
        -- 将游标查询到的结果放进变量 p_price 里。
        FETCH cur INTO p_price
        -- 如果没有查询到结果就退出。
        EXIT WHEN cur%NOTFOUND
        -- 依次输出所查询到的价格信息。
        DBMS_OUTPUT.PUT_LINE(p_price)
        -- 使用 case 语句，对于所输出的每一个价格，判断价格的高低状态，并附相应的值给 message，例如价格大于
20，则 message 的信息为'the price is too high'。
        message:= CASE
        WHEN p_price >20 THEN 'price is too high'
        WHEN p_price>15 THEN 'price is high'
        WHEN p_price>10 THEN 'price is good'
        WHEN p_price >5 THEN 'price is ok'
        END
        -- 依次输出 message 的信息。
        DBMS_OUTPUT.PUT_LINE(message)
    END LOOP;
    CLOSE cur
END;
/
```

输出结果为：

output
12.13
price is good
6.12
price is ok
9.81
price is ok
7.49
price is ok
7.02
price is ok
18.38
price is high

8.11.7. PL/SQL 存储过程

Oracle
中的命名语句块分为存储过程（procedure）和函数（function），本章节中我们主要讲述过程的创建和调用。

> Inceptor 支持在外部链接工具中查看已定义的存储过程

8.11.7.1. 不带参数的过程

例 135. 创建一个名为 hello_world 的过程

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建一个不带参数的过程 hello_world。
hello_world()
IS
   -- 定义一个 string 类型的变量 l_message，并赋值。
   l_message STRING := 'Hello World!'
BEGIN
   -- 输出变量 l_message 的值。
   DBMS_OUTPUT.put_line (l_message)
END;
/

BEGIN
   hello_world()
END;
/
```

输出结果：

output
Hello World!

例 136. 以同样的方法创建 hello_universe 过程

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
hello_universe()
IS
  l_message STRING := 'Hello Universe!'
BEGIN
  DBMS_OUTPUT.put_line (l_message)
END;
/
BEGIN
   hello_universe()
END;
/
```

输出结果为：

output
Hello Universe!

8.11.7.2. 带参数的过程

Inceptor 中，支持创建带有参数的存储过程，且存储过程的参数可分为三类，包括 IN/OUT/INOUT。

- IN 类型：默认模式，在调用 procedure 的时候，procedure 的实参的值被传递到该 procedure，在 procedure 的内部，procedure 的形参是只读的。
- OUT 类型：在调用 procedure 的时候，不能传递给 procedure 实参的值，在 procedure 的内部，procedure 的形参是只可写的。
- INOUT 类型：在调用 procedure 的时候，实参的值可以被传递给该 procedure，在其内部，形参可以被读出也可以被写入，该 procedure 结束时，形参的内容将赋给调用时的实参。

8.11.7.2.1. IN 类型

在 IN 模式下，创建过程时所定义的形参的值不能更改，可以通过参数给过程传入数据，不能通过参数从过程传出数据给调用该过程的 PL/SQL 代码块。即 IN 模式的参数类似于一个常量，不可更改。

例 137. 创建 IN 模式的参数过程 param

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建一个名为 param 的过程。
param(
  -- 过程 param 的形参为 p_in，参数类型为 IN，参数的数据类型为 INT。
  p_in IN INT
)
IS
BEGIN
   -- 输出参数 p_in 的值。
   DBMS_OUTPUT.PUT_LINE('in param:'||p_in)
END;
/
BEGIN
   -- 调用过程 param，其中'222'为实参。
   param(222)
END;
/
```

输出结果为：

output
in param:222

8.11.7.2.2. 默认模式

在 PL/SQL 中创建带有参数的过程，如果没有指定参数模式，则视为默认模式（IN），其作用与带有 IN 的参数过程一致，但最好应该指定参数的模式。

示例一

例 138. 创建带有参数的过程 hello_place

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建一个带有参数的过程 hello_place。
hello_place (
  -- 参数为 place_in，不指定参数类型，即为默认模式，参数的数据类型为 STRING。
  place_in STRING
)
IS
  -- 定义一个 string 类型的变量。
  l_message STRING
BEGIN
  -- 给变量 l_message 赋值。
  l_message  := 'Hello ' || place_in
  -- 输出变量 l_message 的值。
  DBMS_OUTPUT.put_line (l_message)
END;
/
BEGIN
  hello_place ('World')
  hello_place ('Universe')
END;
/
```

hello_place 过程输出结果为：

output
Hello World
Hello Universe

示例二

例 139. 创建带有参数的过程 ptransaction

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建一个名为 ptransaction 的过程。
ptransaction(
-- 参数为 p_trans_id，不指定参数类型，即为默认模式，参数的数据类型为表 transactions 中 trans_id
字段的类型。
  p_trans_id transactions.trans_id%type
)
IS
   -- 定义一个变量 v_price，数据类型为表 transactions 中 price 字段的类型。
   v_price transactions.price%TYPE
BEGIN
   -- 将表 transactions 中 trans_id 的值为 p_trans_id 时所对应的 price 的值，放进变量 v_price 里。
   SELECT price
     INTO v_price
     FROM transactions
     WHERE trans_id=p_trans_id;
   -- 输出变量 v_price 的值。
   DBMS_OUTPUT.PUT_LINE('the price is:'||v_price)
END;
/

-- 调用过程 ptransaction，'943197522'为该过程的实参。
BEGIN
   ptransaction('943197522')
END;
/
```

ptransaction 过程输出结果为:

output
the price is:12.13

8.11.7.2.3. OUT类型

OUT 模式的参数的作用与 IN 模式的参数的作用正好相反。
在 OUT 模式下，可以把数据从一个过程传回给调用该程序的 PL/SQL 块，事实上，OUT 模式的参数在程序成功结束之前是没有任何值的（除非使用了 NOCOPY 提示）。

在程序的执行过程中，针对 OUT 模式的参数所做的任何赋值动作，实际上都是在对一个副本进行操作，这个副本是 PL/SQL 内部为这个 OUT 模式的参数创建的，当程序成功结束，并把控制权返回给调用他的外层块时，这个本地副本的值才会传递给真正的 OUT 模式的参数，此时 OUT 模式的参数的值才可以在调用他的 PL/SQL 语句块中使用。

- OUT 模式的参数类似于一个未初始化的变量，OUT 模式的形参所对应的实参必须是一个变量，不能是常量，字符或表达式。
- OUT 模式的参数不能存在缺省值，我们只能在所创建过程的内部给 OUT 模式的参数赋值。
- 由于 OUT 模式的参数的值一定要等到程序成功结束才会被真正的赋予，所以如果程序执行过程中发生了异常，任何对于 OUT 参数的赋值操作都会被回滚。

示例一

例 140. 创建一个名为 param1 的过程

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建一个名为 param1 的过程。
param1(
  -- 参数为 p_out，参数类型为 OUT，参数的数据类型为 STRING。
  p_out OUT string
)
IS
BEGIN
   -- 给参数 p_out 赋值。
   p_out:='20140105100520'

END;
/
DECLARE
   -- 声明一个数据类型为 STRING 的变量 v_out。
   v_out STRING
BEGIN
   -- 调用过程 param1，其中 v_out 用来存放过程 param1 所输出的值。
   param1(v_out)
   -- 打印出变量 v_out 所获得的值。
   DBMS_OUTPUT.PUT_LINE('out param:'||v_out)
END;
/
```

输出结果为：

output
out param:20140105100520

示例二

例 141. 创建一个名为 param2 的过程

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建一个名为 param2 的过程。
param2(
  -- 第一个参数为 p_in，类型为 IN，数据类型为 DOUBLE。
  p_in IN DOUBLE,
  -- 第二个参数为 p_out，类型为 OUT，数据类型为 STRING。
  p_out OUT STRING
)
IS
BEGIN
   -- 输出第一个参数 p_in 的值。
   DBMS_OUTPUT.PUT_LINE('in param:'||p_in)
   -- 将表 transactions 中 price 的值和 p_in 相等所对应的账号放进 p_out 里。
   SELECT acc_num
   INTO p_out
   FROM transactions
   WHERE price=p_in;
END;
/
DECLARE
   -- 声明一个字符串类型的变量 v_out。
   v_out STRING
BEGIN
   -- 调用过程 param2，其中 v_out 用来存放 param 所输出的数据。
   param2(12.13,v_out)
   -- 打印出变量 v_out 所获得的值。
   DBMS_OUTPUT.PUT_LINE('out param:'||v_out)
END;
/
```

输出结果为：

output
in param:12.13
out param:6513065

8.11.7.2.4. INOUT类型

- INOUT 模式的参数类似于一个未初始化的变量。
- INOUT 模式的形参所对应的实参必须是一个变量，不能是常量，字符或表达式。
- INOUT 模式的参数不能存在缺省值。

例一：IN 与 INOUT 的对比

例 142. 创建一个名为 param 的过程

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建一个名为 param 的过程。
param(
  -- 第一个参数为 p_in，类型为 IN，数据类型为 DOUBLE。
  p_in IN DOUBLE,
  -- 第二个参数为 p_inout，类型为 INOUT，数据类型为 STRING。
  p_inout INOUT STRING )
IS
BEGIN
   -- 输出第一个参数 p_in 的值。
   DBMS_OUTPUT.PUT_LINE('in param:'||p_in)
   -- 输出 p_inout 的值。
   DBMS_OUTPUT.PUT_LINE('inout param:'||p_inout)
END;
/
DECLARE
   -- 声明一个变量 v_inout，数据类型为字符串，并附初值。
   v_inout string:='6513065'
BEGIN
   -- 调用过程 param，其中 v_inout 用来存放 param 所输出的值。
   param(11.12,v_inout)
END;
/
```

输出结果为：

output
in param:11.12
inout param:6513065

例二：OUT 与 INOUT 的对比

例 143. 创建一个名为 param 的过程

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建一个名为 param 的过程。
param(
  -- 第一个参数为 p_out，参数类型为 OUT。
  p_out OUT double,
  -- 第二个参数为 p_inout，参数类型为 INOUT。
  p_inout INOUT string
)
IS
BEGIN
   -- 给 p_out 赋值。
   p_out:='11.11'
   -- 给 p_inout 赋值。
   p_inout:='20140105100520'
END;
/
DECLARE
   -- 声明 DOUBLE 类型的变量 v_out，STRING 类型的变量 v_inout。
   v_out DOUBLE v_inout STRING
BEGIN
    -- 调用过程 param，其中 v_out,v_inout 分别用来存放 param 所输出的数据。
    param(v_out,v_inout)
    -- 打印出变量 v_out 所获得的值。
    DBMS_OUTPUT.PUT_LINE('out param:'||v_out)
    -- 打印出变量 v_inout 所获得的值。
    DBMS_OUTPUT.PUT_LINE('inout param:'||v_inout)
END;
/
```

输出结果为：

output
out param:11.11
inout param:20140105100520

例三

例 144. 创建一个名为 param 的过程

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建过程 param。
param(
  -- 第一个参数为 p_in，类型为 IN，数据类型为 DOUBLE。
  p_in IN double,
  -- 第二个参数为 p_inout，类型为 INOUT，数据类型为 STRING。
  p_inout INOUT string
)
IS
BEGIN
  -- 输出 p_in 的值。
  DBMS_OUTPUT.PUT_LINE('in param:'||p_in)
  -- 将表 transactions 中 price 的值和 p_in 相等所对应的账号放进 p_inout 里。
  SELECT acc_num
    INTO p_inout
    FROM transactions
    WHERE price=p_in;
  -- 输出 p_inout 的值。
  DBMS_OUTPUT.PUT_LINE('inout param:'||p_inout)
END;
/
DECLARE
   -- 声明一个字符串类型的变量 v_inout。
   v_inout STRING
BEGIN
   -- 调用过程 param，其中 v_inout 用来存放 param 所输出的数据。
   param(11.11,v_inout)
END;
/
```

输出结果为：

output
in param:11.11
inout param:3912384

8.11.7.2.5. 实参与形参的关联

在 PL/SQL 过程中，使用两种方法把形参和所对应的实参关联在一起分别为位置表示法和命名表示法。

位置表示法

即隐式地（根据位置）把一个实参关联到一个形参。

例 145. 创建一个含有五个参数的过程 param

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建过程 param。
param(
  -- 第一个参数为 ptrans_id，类型为 IN，数据类型为 STRING。
  ptrans_id IN STRING,
  -- 第二个参数为 ptrans_time，类型为 IN，数据类型为 STRING。
  ptrans_time IN STRING,
  -- 第三个参数为 ptrans_type，类型为 OUT，数据类型为 STRING。
  ptrans_type OUT STRING,
  -- 第四个参数为 pacc_num，类型为 INOUT，数据类型为 STRING。
  pacc_num INOUT STRING,
  -- 第五个参数为 pprice，类型为 OUT，数据类型为 DOUBLE。
  pprice OUT DOUBLE
)
IS
BEGIN
   -- 给 ptrans_type 赋值。
   ptrans_type:='b'
   -- 给 pprice 赋值。
   pprice:=12.13
   DBMS_OUTPUT.PUT_LINE('the trans_id is:'||ptrans_id)
   -- 查询表 transactions 中的账号，条件为交易号等于 ptrans_id，将查询结果放进 pacc_num。
   SELECT acc_num INTO pacc_num FROM transactions WHERE trans_id=ptrans_id;
   -- 查询表 transactions 中的交易时间，条件为交易号等于 ptrans_id，将查询结果放进 ptrans_time。
   SELECT trans_time INTO ptrans_time FROM transactions WHERE trans_id=ptrans_id;
   DBMS_OUTPUT.PUT_LINE('the trans_time is:'||ptrans_time)
END;
/
```

例 146. 调用上节所创建的过程 param

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个字符串类型的变量 v_trans_type。
   v_trans_type STRING
   -- 声明一个 double 类型的变量 v_price。
   v_price double
   -- 声明一个字符串类型的变量 v_acc_num。
   v_acc_num STRING
BEGIN
   -- 使用位置表示法，将 param 中的实参和形参关联在一起，其中'943197522'对应形参 ptrans_id
, '20140105100520'对应 ptrans_time，v_trans_type 对应 ptrans_type，v_acc_num 对应 pacc_num
, v_price 对应 pprice。
   param('943197522','20140105100520',v_trans_type,v_acc_num,v_price)
   -- 分别打印出变量 v_trans_type，v_price，v_acc_num 在执行存储过程时，所获得的值。
   DBMS_OUTPUT.PUT_LINE('the type is:'||v_trans_type)
   DBMS_OUTPUT.PUT_LINE('the price is:'||v_price)
   DBMS_OUTPUT.PUT_LINE('the acc_num is:'||v_acc_num)
END;
/
```

输出结果为：

output
the trans_id is:943197522
the trans_time is:20140105100520
the type is:b
the price is:12.13
the acc_num is:6513065

命名表示法

即显式地把一个实参关联到一个形参，使用的形参的名字以及一个 连接符。

例 147. 创建一个含有五个参数的过程 param

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建过程param。
param(
  -- 第一个参数为 ptrans_id，类型为 IN，数据类型为 STRING。
  ptrans_id IN STRING,
  -- 第二个参数为 ptrans_time，类型为 IN，数据类型为 STRING。
  ptrans_time IN STRING,
  -- 第三个参数为 ptrans_type，类型为 OUT，数据类型为 STRING。
  ptrans_type OUT STRING,
  -- 第四个参数为 pacc_num，类型为 INOUT，数据类型为 STRING。
  pacc_num INOUT STRING,
  -- 第五个参数为 pprice，类型为 OUT，数据类型为 DOUBLE。
  pprice OUT DOUBLE)
IS
BEGIN
   -- 给 ptrans_type 赋值。
   ptrans_type:='b'
   -- 给 pprice 赋值。
   pprice:=12.13
   -- 查询表 transactions 中的账号，条件为交易号等于 ptrans_id，将查询结果放进 pacc_num。
   SELECT acc_num INTO pacc_num FROM transactions WHERE trans_id=ptrans_id;
   DBMS_OUTPUT.PUT_LINE('the trans_id is:'||ptrans_id)
   -- 查询表 transactions 中的交易时间，条件为交易号等于 ptrans_id，将查询结果放进 ptrans_time。
   SELECT trans_time INTO ptrans_time FROM transactions WHERE trans_id=ptrans_id;
   DBMS_OUTPUT.PUT_LINE('the trans_time is:'||ptrans_time)
END;
/
```

例 148. 调用过程 param

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个字符串类型的变量 v_trans_type。
   v_trans_type string
   -- 声明一个 double 类型的变量 v_price。
   v_price double
   -- 声明一个字符串类型的变量 v_acc_num。
   v_acc_num string
BEGIN
   -- 使用命名表示法，将 param 中的实参和形参关联在一起，其中'943197522'对应形参 ptrans_id
, '20140105100520'对应 ptrans_time，v_trans_type 对应 ptrans_type，v_acc_num 对应 pacc_num
, v_price 对应 pprice。
   param(ptrans_id  => '943197522',
         ptrans_time => '20140105100520',
         ptrans_type => v_trans_type,
         pacc_num => v_acc_num,
         pprice => v_price)
    -- 分别打印出变量 v_trans_type，v_price，v_acc_num 在执行存储过程时，所获得的值。
    DBMS_OUTPUT.PUT_LINE('the type is:'||v_trans_type)
    DBMS_OUTPUT.PUT_LINE('the price is:'||v_price)
    DBMS_OUTPUT.PUT_LINE('the acc_num is:'||v_acc_num)
END;
/
```

输出结果为：

output
the trans_id is:943197522
the trans_time is:20140105100520
the type is:b
the price is:12.13
the acc_num is:6513065

8.11.7.3. 过程重载

- Inceptor 中支持过程的重载，即子程序的名字相同，但参数的类型或数目不能相同（参数的名称可以相同），返回值也可能不同。
- 在实际的执行过程中，Inceptor 会根据实参的数据类型，来决定应该执行的过程。

例 149. 重载过程 overload_test

```
!set plsqlUseSlash true
-- 创建第一个名为 overload_test 的过程，形参名为 place_in，其数据类型为字符串。
CREATE OR REPLACE PROCEDURE overload_test (place_in STRING)
IS
  l_message STRING
BEGIN
  l_message  := 'Hello ' || place_in
  DBMS_OUTPUT.put_line (l_message)
END;
/
-- 创建第二个名为 overload_test 的过程，形参名为 v_name，其数据类型为整数类型。
CREATE OR REPLACE PROCEDURE overload_test(v_name INT)
IS
  l_message STRING
BEGIN
  l_message  := v_name||'is a good number'
  DBMS_OUTPUT.put_line (l_message)
END;
/
BEGIN
-- 过程内的实参为 world，类型为字符串，则会执行第一个过程。
overload_test ('world')
-- 过程内的实参为 888，类型为整数类型，则会执行第二个过程。
overload_test(888)
-- 过程内的实参为 999，类型为字符串，则会执行第一个过程。
overload_test('999')
END;
/
```

输出结果为：

output
Hello world
888is a good number
Hello 999

需要注意的是，在 Inceptor 中，函数和过程是不能在 DECLARE 声明块内直接声明的，所以以下用法用来创建过程的重载是错误的。

- 例：在声明块内声明过程 overload_test

```
!set plsqlUseSlash true
DECLARE
   PROCEDURE overload_test(place_in STRING)
   IS
      l_message STRING
   BEGIN
      l_message  := 'Hello ' || place_in
      DBMS_OUTPUT.put_line (l_message)
   END;

   PROCEDURE overload_test(v_name INT)
   IS
    l_message STRING
    BEGIN
    l_message  := v_name||'is a good number'
    DBMS_OUTPUT.put_line (l_message)
    END;
/
```

```
COMPILE FAILED: Parse error: [Error 1101] line 2:0 cannot recognize input near 'PROCEDURE
overload_test (' in declare spec item
```

8.11.7.4. 系统预定义过程

- Inceptor 中有两个系统预定义的过程，分别为 set_env(string,string)，put_line(string)。
- 我们可以使用相关命令来查看系统预定义函数/过程的有关信息，具体内容可参见注意事项章节。
- 系统预定义过程的具体使用方法和案例，可参见预定义函数/过程/包章节。

8.11.8. PL/SQL 函数

Inceptor 中支持创建 PL/SQL 存储过程，同时也支持创建 PL/SQL 函数，一般情况下，存储过程和函数的作用完全相同，主要区别点在于存储过程没有返回值，而函数有一个返回值，也可以通俗地理解为函数是一个有返回值的过程，可根据实际情况，选择存储过程和函数的使用。

8.11.8.1. 创建函数

语法

```
CREATE [OR REPLACE] FUNCTION <function_name> ([<parameter1> <mode1> <datatype1>, <parameter2> <
mode2> <datatype2> …])
RETURN <datatype>
IS
  PL/SQL Block;
```

参数说明

- <function_name>：函数名称。
- <mode>：表示参数的类型，可分为 IN，INOUT，OUT 三种类型，缺省状态下默认为 IN 类型，更多用法说明，可参见 PL/SQL 存储过程章节。
- <parameter>：函数中的参数名称。
- <datatype>：数据类型。

实例

例 150. 创建名为 hello_message 的函数

```
!set plsqlUseSlash true
CREATE OR REPLACE FUNCTION
hello_message
 (place_in IN STRING)
RETURN STRING
IS
BEGIN
  RETURN 'Hello ' || place_in
END;
/
```

- 子程序的名字描述被返回的数据。而不是被执行的过程。
- 子程序中多了一个 RETURN 从句。
- 该函数是设置函数返回值的数据类型。

8.11.8.2. 查看函数

语法

```
DESC PLSQL FUNCTION [EXTENDED] <function_name>; --[1]
```

参数说明

- <function_name>：函数名称。
- 可选项 [1]：可以选择添加关键字 EXTENDED 列出该 PL/SQL 函数的原文详细信息。

实例

例 151. 查看名为 hello_message 的函数

```
desc plsql function hello_message;
```

结果如下：

```
+-----------------------------------------------------------------------+
|                              description                              |
+-----------------------------------------------------------------------+
| Prototype:                                                            |
| FUNCTION db_rachel.hello_message(place_in IN  STRING) RETURN STRING  |
+-----------------------------------------------------------------------+
```

8.11.8.3. 调用函数

8.11.8.3.1. 单独调用函数

例 152. 单独调用函数 hello_message 的方法

```
!set plsqlUseSlash true
DECLARE
   l_message string
BEGIN
   l_message := hello_message ('Universe')
   DBMS_OUTPUT.put_line (l_message)
END;
/
```

输出结果为：

output
Hello Universe

8.11.8.3.2. 在过程中调用函数

例 153. 创建过程 hello_place 中调用 hello_message 函数

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
-- 创建带有参数的过程 hello_place。
hello_place
(place_in string)
IS
BEGIN
   DBMS_OUTPUT.put_line (hello_message (place_in))
END;
/
BEGIN
-- 调用函数 hello_message。
   hello_place('world')
END;
/
```

输出结果为：

output
Hello world

8.11.8.3.3. SQL 语句中调用函数

> 标准 SQL 调用自定义的 PLSQL 函数，必须满足以下四个条件
>
> - 必须是函数，不能是过程。
> - 返回值必须是基本类型。
> - 函数中不能有标准 SQL 语句。
> - 更多内容详解，可参见注意事项章节。

- 本节中我们所使用到的 HOLODESK 表 ra

  其中姓名 name 为 string 类型，成绩绩点 gpa 为 double类 型。

name	gpa
smith	3.4
lisi	3.3
ll	3.3
lily	3.6

例 154. 利用函数 hello_message 往表 ra 中插入一条数据

```
!set plsqlUseSlash true
BEGIN
  INSERT INTO ra VALUES (hello_message('tom'),4.8)
END;
/
```

- 再次查看表中的数据

```
SELECT * FROM ra;
```

name	gpa
ll	3.3
lily	3.6
Hello tom	4.8
smith	3.4
lisi	3.3

可以看到，ra 表中成功插入了一条数据（'hello tom',4.8)。

8.11.8.3.4. 函数的嵌套调用

Inceptor 可支持函数的嵌套调用，即在一个函数中调用另外一个函数。

例 155. 创建函数 hello_message_exclamation，并在其内部调用 hello_message 函数

```
!set plsqlUseSlash true
CREATE OR REPLACE FUNCTION
hello_message_exclamation
 (message_in IN STRING)
RETURN STRING
IS
BEGIN
  RETURN hello_message(message_in) || ' !'
END;
/
DECLARE
   l_message_a string
BEGIN
   l_message_a := hello_message_exclamation ('Universe')
   DBMS_OUTPUT.put_line (l_message_a)
END;
/
```

执行结果为：

output
Hello Universe !

8.11.8.4. 函数重载

- Inceptor 中支持函数的重载，即函数的名称相同，但是参数的类型或数目不能相同（参数的名称可以相同），返回值也可能不同。
- 在实际的执行过程中，Inceptor 会根据实参的数据类型，来决定应该执行的函数。

例 156. 重载函数 overload_test_func

```
!set plsqlUseSlash true
-- 创建第一个名为 overload_test_func 的过程，形参名为 place_in，其数据类型为字符串。
CREATE OR REPLACE FUNCTION overload_test_func
  (place_in string) RETURN string
  IS
  BEGIN
    RETURN 'Hello ' || place_in
  END;
/
-- 创建第二个名为 overload_test_func 的过程，形参名为 place_in，其数据类型为整数类型。
CREATE OR REPLACE FUNCTION overload_test_func
  (place_in INT) RETURN string
 IS
 BEGIN
   RETURN place_in * place_in
  END;
/
DECLARE
    V1 STRING
    V2 STRING
    V3 STRING
BEGIN
  V1 :=overload_test_func('world')
  V2 :=overload_test_func(11)
  V3 :=overload_test_func('111')
  -- 函数内的实参为 world，类型为字符串，则会执行第一个函数。
  DBMS_OUTPUT.PUT_LINE(V1)
  -- 函数内的实参为 11，类型为整数类型，则会执行第二个函数。
  DBMS_OUTPUT.PUT_LINE(V2)
  -- 函数内的实参为 111，类型为字符串，则会执行第一个函数。
  DBMS_OUTPUT.PUT_LINE(V3)
END;
/
```

输出结果：

output
Hello world
121
Hello 111

需要注意的是，在 Inceptor 中，函数和过程是不能在 DECLARE 声明块内直接声明的，所以以下用法用来创建过程的重载是错误的。

- 例：

```
!set plsqlUseSlash true
DECLARE
  FUNCTION overload_test_func (place_in string) RETURN string
  IS
  BEGIN
    RETURN 'Hello ' || place_in
  END;
/
 FUNCTION overload_test_func (place_in INT) RETURN string
 IS
  BEGIN
   RETURN place_in * place_in
  END;
/
DECLARE
    V1 STRING
    V2 STRING
    V3 STRING
BEGIN
  V1 :=overload_test_func('world')
  V2 :=overload_test_func(11)
  V3 :=overload_test_func('111')
  DBMS_OUTPUT.PUT_LINE(V1)
  DBMS_OUTPUT.PUT_LINE(V2)
  DBMS_OUTPUT.PUT_LINE(V3)
END;
/
```

8.11.8.5. 系统预定义函数

- Inceptor 中有八个系统预定义的函数，分别为

  sqlcode(void)

  sqlerrm(void)

  get_columns(string,nestedtable<string>)

  raise_application_error(int,string,bool)

  set_env(string,string)

  get_env(string)

  put_line(string)

  sqlerrm(int)
- 我们可以使用相关命令来查看系统预定义函数/过程的有关信息，具体内容可参见注意事项章节。
- 系统预定义函数的具体使用方法和案例，可参见预定义函数/过程/包章节。

8.11.9. Records

Records，又称记录型变量，内部有很多分量，每个分量都有自己的名字及数据类型。类似于 C 语言中的结构数据类型（STRUCTURE）。

- 在使用记录数据类型变量时，需要在声明部分先定义记录的组成、记录的变量，然后在执行部分引用该记录变量本身或其中的成员。
- 只能处理单行数据，数据的类型可以自己定义，也可以基于表来定义，只能在 PL/SQL 中运行。

8.11.9.1. Records 语法

- 记录型的创建：

  DECLARE TYPE 记录类型名 IS RECORED （分量列表）

例如：

```
DECLARE TYPE test_record_type IS RECORD (test_trans_id string, test_trans_time string
,test_price double)
```

- 记录型变量的创建

  变量名 记录类型名

  例如：

```
testrecord1 test_record_type
```

- 引用变量：

  变量名.分量名

  例如：

```
testrecord1.test_price
```

8.11.9.2. Records 创建及使用

8.11.9.2.1. 方法一：自定义每一个分量的名字和类型

例 157. 定义一个记录变量，查询表 transactions 中金额为 1100 的交易号

```
!set plsqlUseSlash true
DECLARE
-- 声明一个名为 test_record_type 的记录类型。
TYPE test_record_type IS RECORD
 -- 分别定义 test_record_type 记录类型里的各个分量的数据类型。
 (test_trans_id string, test_trans_time string ,test_price double)
 -- 定义一个名为 testrecord1 的 test_record_type 变量。
 testrecord1 test_record_type
BEGIN
   -- 查询 transactions 表中金额为 1100 的交易号，交易时间，以及价格，并放进变量 testrecord1 里。
   SELECT trans_id,trans_time,price
    INTO testrecord1
    FROM transactions
    WHERE amount=1100;
   -- 输出变量 testrecord1 中的分量 test_trans_id 的值。
   dbms_output.put_line(testrecord1.test_trans_id)
END;
/
```

输出结果：

output
895916502

8.11.9.2.2. 方法二：基于表，定义每一个分量的名字和类型

例 158. 基于表中的字段类型定义每一个分量的数据类型

```
!set plsqlUseSlash true
DECLARE
-- 声明一个名为 test_record_type 的记录类型。
TYPE test_record_type IS RECORD
 -- 定义 test_record_type 记录类型里的各个分量的数据类型分别为表 transactions 中 trans_id
, trans_time,price 字段的类型。
 (test_trans_id transactions.trans_id%type,
 test_trans_time transactions.trans_time%type ,
 test_price transactions.price%type)
 -- 定义一个名为 testrecord2 的 test_record_type 变量。
 testrecord2 test_record_type
BEGIN
   -- 查询 transactions 表中金额为 1300 的交易号，交易时间，以及价格，并放进变量 testrecord2 里。
   SELECT trans_id,trans_time,price
     INTO testrecord2
     FROM transactions
     WHERE amount=1300;
   -- 输出变量 testrecord2 中的分量 test_trans_time 和 test_trans_id 的值。
   dbms_output.put_line(testrecord2.test_trans_time||' ' ||testrecord2.test_trans_id)
END;
/
```

输出结果为：

output
20140702113025 991691937

8.11.9.2.3. 方法三：利用 %ROWTYPE 属性快速声明一个记录型变量

- %rowtype 属性：即基于表声明一个记录型变量，该记录型变量的字段名和字段类型分别为表中的字段名和字段类型。
- %前面可以写表名,但是 Inceptor 中不支持前面写游标名。
- 用法：表名%ROWTYPE。

例 159. 基于表，定义一个记录型变量

```
!set plsqlUseSlash true
DECLARE
   -- 基于 transactions 表定义一个名为 test_record 的记录型变量，该记录型变量的每一个分量的名字和类型均和表
transactions 中的字段名和字段类型相同。
   test_record transactions%ROWTYPE
BEGIN
   -- 查询 transactions 表中股票账号为 AA7105670 的全部信息，并将查询到的信息放入记录型变量 test_record
里。
   SELECT *
    INTO test_record
    FROM transactions
    WHERE stock_id='AA7105670';
  -- 输出记录型变量 test_record 中的分量 trans_id, trans_time, price, amount 的值。
  dbms_output.put_line(test_record.trans_id||' ' ||test_record.trans_time||' '||test_record
.price||' '||test_record.amount)
END;
/
```

输出结果为：

output
943197522 20140105100520 12.13 200.0

8.11.10. Collections

Inceptor中 PL/SQL 中的集合元素一共有三种形式，分别为Associative arrays 数组，Nested tables 嵌套表，和 VARRAYs 变量数组。

1. VARRAR，可以看作是一个预先已经定义好长度的一维数组，可使用数字编号可以依次操作每个数组元素。
2. Nested tables，可以看着是一个一维数组，可使用数字编号可以依次操作每个数组元素，要在给变量赋值前，给变量分配空间。
3. Associative arrays，可以看着是一个数据字典，有 key、value两列。key 值可以是任意数字和字符串，value 值可以是任意对象包括 collection 类型的对象。

其中，Associative arrays 可以定义索引的数据类型，而 Nested tables 和 VARRAR 默认都是以整数索引。 另外，Associative arrays 变量在赋值前不需要进行初始化，Nested tables 和 VARRAR 需要在赋值前进行初始化，否则会报 collection_is_null 异常。

8.11.10.1. VARRAY

- VARRAY 是 collections 类型的一维，有界，且不稀疏的集合。

  有界（即在声明该类型的时候，需要定义一个最大范围）。

  不稀疏（在删除掉数组中的某一个元素时，会重新排序）。

8.11.10.1.1. VARRAY 语法

- VARRAY 类型的创建

  DECLARE TYPE VARRAY 类型名 IS VARRAY(variable-sized arrays) OF 数据类型

例如：

声明一个名为 varray_type 的 VARRAY 类型，最多可以容纳两个数据，数据的类型为 STRING.

```
DECLARE TYPE varray_type AS VARRAY(2) OF STRING;
```

- VARRAY 变量的创建

  VARRAY 变量名 VARRAY 类型名。

  例如：

  定义一个名为 v_varray_type 的 varray_type 类型的变量，语句如下：

```
v_varray_type   varray_type
```

- VARRAY变量的赋值

  VARRAY变量名(variable-sized arrays):= 变量的值

  例如：

  给变量 v_varray_type(1) 赋值

```
v_varray_type(1):='name'
```

- variable-sized arrays 为集合元素的最大个数，声明一个 VARRAY 类型时，必须定义可变数组的最大范围。
- VARRAY 类型的变量，可以不用赋初值。
- 一般 VARRAY 类型和 VARRAY 变量的声明，仅在当前 PL/SQL 语句块中有效，但如果在包内声明 VARRAY 类型和 VARRAY 变量，就可以跨语句块的使用。

8.11.10.1.2. VARRAY 声明与赋值

例 160. 变量的声明与赋值

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个能保存 10 个 STRING 数据类型的成员的 VARRAY 数据类型 ORG_VARRAY_TYPE。
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   -- 定义一个 ORG_VARRAY_TYPE 类型的 VARRAY 变量 V_ORG_VARRAY。
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   -- 初始化变量 V_ORG_VARRAY 的值。
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   -- 输出变量 V_ORG_VARRAY 前 4 个成员的值。
   DBMS_OUTPUT.PUT_LINE('输出1：' || V_ORG_VARRAY(1) || '、'|| V_ORG_VARRAY(2) || '、'||
V_ORG_VARRAY(3) || '、'|| V_ORG_VARRAY(4))
   -- 输出变量 V_ORG_VARRAY 第 5 个成员的值。
   DBMS_OUTPUT.PUT_LINE('输出2：' || V_ORG_VARRAY(5))
   -- 给变量 V_ORG_VARRAY 第 5 个成员赋值为 5001。
   V_ORG_VARRAY(5) := '5001'
   -- 再次输出变量 V_ORG_VARRAY 第 5 个成员的值。
   DBMS_OUTPUT.PUT_LINE('输出3：' || V_ORG_VARRAY(5))
END;
/
```

输出结果为：

output
输出1：1、2、3、4
输出2：5
输出3：5001

8.11.10.1.3. VARRAY 类型的使用范围

例 161. 声明的 VARRAY 类型和 VARRAY 变量，仅在当前的 PL/SQL 语句块中生效

```
!set plsqlUseSlash true
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   FOR I IN V_ORG_VARRAY.FIRST()..V_ORG_VARRAY.LAST()
   LOOP
      DBMS_OUTPUT.PUT_LINE('V('||I||')=' || V_ORG_VARRAY(I))
   END LOOP;
END;
/
```

输出结果为：

可以看到，例 1 中声明的 VARRAY 类型和 VARRAY 变量，仅在当前的 PL/SQL 语句块中生效，在例 2 创建的 PL/SQL 语句块中，直接调用 VARRAY 类型和 VARRAY变量，Inceptor 会报错。

```
COMPILE FAILED: Semantic error: [Error 11152]
ANONYMOUS BLOCK (LINE 2, COLUMN 0, TEXT \"V_ORG_VARRAY :=
ORG_VARRAY_TYPE('1','2','3','4','5')\"): Undefined identifier used in left or right side
ASSIGN: V_ORG_VARRAY
```

8.11.10.2. NESTED TABLE

- NESTED TABLE 是 collections 类型的一维，无边界，不稀疏的集合。

  无边界（在声明该类型的时候，不需要定义元素的最大个数，需要在给变量赋值之前，给变量分配空

间）。

不稀疏（在删除数组中的某个数据时，数组会重新排序）。

- NESTED TABLE 类型即 PL/SQL 表类型，与 records 类型相似，是对记录类型的扩展。
- 它可以处理多行记录，类似于C语言中的二维数组，使得可以在 PL/SQL 中模仿数据库中的表。

8.11.10.2.1. NESTED TABLE 语法

- 表类型的创建

```
DECLARE TYPE table_type IS TABLE OF TYPE
```

例如:

```
DECLARE TYPE list_of_names_t IS TABLE OF string
```

- 表类型变量的创建

```
v_table_type  table_type
```

```
children  list_of_names_t
```

- 表类型变量的初始化

v_table_type := 变量的值

```
children :=list_of_names_t ()
```

- TYPE 可以为全部的数据类型，包括标量类型和复合类型。
- 定义集合变量的时候，必须同时初始化集合元素，否则，Inceptor 会报错。

8.11.10.2.2. 表类型变量的声明与赋值

例 162. NESTED TABLE 变量的声明与赋值

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个名为 list_of_names_t 的 table 类型，数据类型为字符串。
   TYPE list_of_names_t IS TABLE OF string
   -- 定义一个名为 happyfamily 的 list_of_names_t 变量，并对集合元素进行初始化。
   happyfamily list_of_names_t:=list_of_names_t ()
BEGIN
   -- 给变量 happyfamily 赋值前，为其分配一个大小为4的空间，也就是最多可以容纳 4 个元素。
   happyfamily.EXTEND (4)
   -- 依次给变量 happyfamily 赋值为 vera，chris，eli，steven。
   happyfamily (1) := 'Veva'
   happyfamily (2) := 'Chris'
   happyfamily (3) := 'Eli'
   happyfamily (4) := 'Steven'
   FOR idx IN 1 .. 4
   LOOP
   DBMS_OUTPUT.PUT_LINE(happyfamily(idx))
   END LOOP
END;
/
```

输出结果为：

output
Veva
Chris
Eli
Steven

8.11.10.3. Associative arrays

- Associative arrays 是 collections 类型的一维，无边界的稀疏集合，只能用于PL/SQL。

  无边界（在声明 Associative arrays 类型时，无需定义元素的最大个数，给变量赋值前也不需要使用 extend 方法给变量分配空间）。

  稀疏的（对于变量 Associative arrays 中的下标可以随意取值，并对该下标所对应的元素赋值）。

8.11.10.3.1. Associative arrays 语法

- Associative arrays 类型的创建

```
DECLARE TYPE table_type IS TABLE OF TYPE INDEX BY index_type
```

例如：

```
DECLARE TYPE numbers_aat IS TABLE OF INT INDEX BY STRING
```

- Associative arrays 变量的创建

```
v_table_type  table_type
```

例如：

```
l_numbers numbers_aat
```

- Associative arrays 变量的赋值

  v_table_type := 变量的值

  例如：

```
l_numbers ('100') := 12345
```

8.11.10.3.2. Associative arrays 变量的声明和赋值

例 163. Associative arrays 变量的声明与赋值

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个名为 numbers_aat 的 Associative arrays类型，存储的值为字符串类型，索引为整数类型。
   TYPE numbers_aat IS TABLE OF STRING INDEX BY INT
   -- 定义一个整数类型的变量。
   l_row INT
   -- 定义一个名为 l_numbers 的 numbers_aat 变量。
   l_numbers numbers_aat
BEGIN
   -- 给变量 l_numbers 中下标为 100 的元素赋值为 LILY。
   l_numbers (100) := 'LILY'
   -- 给变量 l_numbers 中下标为 80 的元素赋值为 ALICE。
   l_numbers (80) := 'ALICE'
   -- 把变量 l_numbers 中第一个元素的下标赋值给 l_row。
   l_row:= l_numbers.first()
   -- 把变量 l_numbers 中第一个元素的下一个元素的下标赋值给 l_row。
   l_row:= l_numbers.next(l_row)
   -- 输出变量 l_numbers 中第一个元素的下一个元素的下标，及其所对应的值。
   DBMS_OUTPUT.put_line (l_row||' ' ||l_numbers(l_row))
END;
/
```

输出结果为：

output
100 LILY

8.11.10.4. Collections 方法

- 集合变量的方法，在存储过程和函数中用于操作 collection 对象，但是都不能在 SQL 语句中直接使用。
- 集合方法中调用的参数对应于 collection 的下标描述符，通常这些描述符都是数字，但是在 associative arrays 中，有可能是字符窜。
- 如果对未初始化的 NESTED TABLE 或者 VARRAY 使用集合元素方法将返回 COLLECTION_IS_NULL 异常。
- 对于类型 Associative arrays，将不支持任何集合元素方法。

8.11.10.4.1. COUNT()

用于计算 associative array、nested table，以及 VARRAY 中条目的个数，使用 DELETE 或者 TRIM 将减少 COUNT，使用 EXTEND 将增加 COUNT。

VARRAY

例 164. 利用 COUNT() 方法

```
!set plsqlUseSlash true
DECLARE
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   -- 利用 COUNT() 方法计算出变量 V_ORG_VARRAY 条目的个数。
   FOR I IN 1..V_ORG_VARRAY.COUNT()
   LOOP
      DBMS_OUTPUT.PUT_LINE('V('||I||')=' || V_ORG_VARRAY(I))
   END LOOP;
END;
/
```

输出结果为：

output
V(1)=1
V(2)=2
V(3)=3
V(4)=4
V(5)=5

NESTED TABLE

例 165. 使用 COUNT() 方法遍历 NESTED TABLE 变量中的值

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个名为 list_of_names_t 的 table 类型，数据类型为字符串。
   TYPE list_of_names_t IS TABLE OF string
   -- 定义一个名为 happyfamily 的 list_of_names_t 变量，并对集合元素进行初始化。
   happyfamily list_of_names_t:=list_of_names_t ()
BEGIN
   -- 给变量 happyfamily 赋值前，为其分配一个大小为 4 的空间，也就是最多可以容纳 4 个元素。
   happyfamily.EXTEND (4)
   -- 依次给变量 happyfamily 赋值为 vera，chris，eli，steven。
   happyfamily (1) := 'Veva'
   happyfamily (2) := 'Chris'
   happyfamily (3) := 'Eli'
   happyfamily (4) := 'Steven'
   -- 使用 count 方法，依次打印出变量 happyfamily 的值。
   FOR l_row IN 1 .. happyfamily.COUNT()
   LOOP
      DBMS_OUTPUT.put_line (happyfamily (l_row))
   END LOOP;
END;
/
```

输出结果为：

output
Veva
Chris
Eli
Steven

Associative arrays

例 166. 不能用 COUNT() 方法遍历 Associative arrays 变量中的值

```
!set plsqlUseSlash true
DECLARE
   TYPE numbers_aat IS TABLE OF STRING INDEX BY INT
   l_row INT
   l_numbers numbers_aat
BEGIN
   l_numbers (100) := 'LILY'
   l_numbers (90) := 'GRACE'
   l_numbers (80) := 'ALICE'
   l_numbers (95) := 'LISI'
   FOR l_row in 1..l_numbers.count()
   LOOP
      DBMS_OUTPUT.put_line (l_numbers(l_row))
   END LOOP;
END;
/
```

输出结果为：

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error 100] NO_DATA_FOUND
*********************************************
ANONYMOUS BLOCK (LINE 12, COLUMN 22, TEXT \"l_numbers(l_row)\")
*********************************************
```

8.11.10.4.2. FIRST()和LAST()

- 对于 VARRAY 和 NESTED TABLE 类型，FIRST() 返回数组的第一个条目的下标，LAST() 返回数组的最后一个条目的下标。
- 对于 Associative arrays 类型，FIRST() 返回数组中的最小下标，LAST() 返回数组的最大下标。

VARRAY

例 167. 使用 FIRST() 和 LAST() 方法

```
!set plsqlUseSlash true
DECLARE
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   -- 使用 FIRST() 和 LAST()方 法分别返回变量 V_ORG_VARRAY 第一个条目的下标和最后一个条目的下标。
   FOR I IN V_ORG_VARRAY.FIRST()..V_ORG_VARRAY.LAST()
   LOOP
      DBMS_OUTPUT.PUT_LINE('V('||I||')=' || V_ORG_VARRAY(I))
   END LOOP;
END;
/
```

输出结果为：

output
V(1)=1
V(2)=2
V(3)=3
V(4)=4
V(5)=5

Associative arrays

例 168. 使用 first() 和 last() 方法

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个名为 numbers_aat 的 Associative arrays 类型，存储的值为字符串类型，索引为整数类型。
   TYPE numbers_aat IS TABLE OF STRING INDEX BY INT
   -- 定义一个名为 l_numbers 的 numbers_aat 变量。
   l_numbers numbers_aat
   -- 分别定义三个整数类型的变量。
   l_row INT
   first_numbers INT
   last_numbers INT
BEGIN
   -- 给变量 l_numbers 中的一些元素赋值，可以跳跃，也可以不按顺序。
   l_numbers(100) := 'LILY'
   l_numbers(90) := 'GRACE'
   l_numbers(80) := 'ALICE'
   l_numbers(95) := 'LISI'
   -- 使用 first() 和 last() 方法，分别将 l_numbers 中的第一个和最后一个元素的下标赋值给相应的变量。
   first_numbers :=l_numbers.first()
   last_numbers:=l_numbers.last()
   -- 依次打印出变量 l_numbers 中的第一个和最后一个元素。
   DBMS_OUTPUT.put_line ('the first one:'||l_numbers(first_numbers))
   DBMS_OUTPUT.put_line ('the last one:'||l_numbers(last_numbers))
END;
/
```

输出结果为：

output
the first one:ALICE
the last one:LILY

由于 Associative arrays 类型的变量内的数据是稀疏的，所以使用 first() 和 last() 方法时，会分别返回变量中元素的所对应下标的最小值和最大值。

8.11.10.4.3. PRIOR(n) 和 NEXT(n)

- PRIOR(n)，返回给定条目的前一个条目的下标，即 PRIOR(n) 和 n-1 是一样的。
- NEXT(n)，返回给定条目的后一个条目的下标，即 NEXT(n) 和 n+1 是一样的。

例 169. 使用 PRIOR(n) 方法

```
!set plsqlUseSlash true
DECLARE
   I INT
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   -- 利用 COUNT() 方法计算出变量 V_ORG_VARRAY 条目的个数,也就是变量 V_ORG_VARRAY
最后一个条目的下标，并赋值给 I。
   I := V_ORG_VARRAY.COUNT()
   WHILE I IS NOT NULL LOOP
      DBMS_OUTPUT.PUT_LINE('V('||I||')=' || V_ORG_VARRAY(I))
      -- 返回 I 条目的前一个条目的下标，并赋值给 I。
      I := V_ORG_VARRAY.PRIOR(I)
   END LOOP;
END;
/
```

输出结果为：

output
V(5)=5
V(4)=4
V(3)=3
V(2)=2
V(1)=1

例 170. 使用 NEXT(n) 方法

```
!set plsqlUseSlash true
DECLARE
   I INT
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   -- 使用 FIRST() 返回变量 V_ORG_VARRAY 第一个条目的下标，并赋值给 I。
   I := V_ORG_VARRAY.FIRST()
   WHILE I IS NOT NULL LOOP
      DBMS_OUTPUT.PUT_LINE('V('||I||')=' || V_ORG_VARRAY(I))
      -- 返回 I 条目的后一个条目的下标，并赋值给 I。
      I := V_ORG_VARRAY.NEXT(I)
   END LOOP;
END;
/
```

输出结果为：

output
V(1)=1
V(2)=2
V(3)=3
V(4)=4
V(5)=5

例 171. 同时使用 PRIOR(n) 方法和 NEXT(n) 方法

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个整数类型的变量，并赋初值为 3。
   I INT default 3
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   -- 将变量 V_ORG_VARRAY 第三个元素的前一个元素的下标赋值给变量 I。
   I := V_ORG_VARRAY.PRIOR(I)
   -- 在 LOOP 循环内，将变量 V_ORG_VARRAY 第三个元素的后一个元素的下标赋值给变量 I，也就是说在 LOOP
循环内将打印出第三个元素之后的所有元素。
   WHILE I IS NOT NULL LOOP
       DBMS_OUTPUT.PUT_LINE('V('||I||')=' || V_ORG_VARRAY(I))
       I := V_ORG_VARRAY.NEXT(I)
   END LOOP;
END;
/
```

输出结果为：

output
V(2)=2
V(3)=3
V(4)=4
V(5)=5

8.11.10.4.4. EXTEND(k)

- 用于为 nested table 或者 VARRAY 分配空间，在不大于元素的最大个数的情况下，为已经赋值的集合变量增加条目个数；不加参数的情况下，默认只添加一个条目。
- 不能用于Associative arrays类型的变量。
- 新增的条目没有值（默认为 NULL），但是你可以对它们进行初始化。
- COUNT() 和 LAST() 方法可以反映 VARRAY 新的容量。
- 对 VARRA 的扩展不可以超过其最大容量。
- 对 VARRAY 扩展前必须要对其进行初始化。

VARRAY

例 172. 使用 EXTEND(k) 方法

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个 INT 类型的变量 I。
   I INT
   -- 声明一个能保存 10 个 STRING 数据类型的成员的 VARRAY 数据类型 ORG_VARRAY_TYPE。
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   -- 定义一个 ORG_VARRAY_TYPE 类型的 VARRAY 变量 V_ORG_VARRAY。
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   -- 初始化变量 V_ORG_VARRAY 的值。
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   -- 将变量 V_ORG_VARRAY 最后一个条目的下标赋值给 I。
   I:=V_ORG_VARRAY.LAST()
   -- 给变量 V_ORG_VARRAY 的最后追加两个新的条目。
   V_ORG_VARRAY.EXTEND(2)
   -- 给变量 V_ORG_VARRAY 新追加的第一个条目赋值为 6。
   V_ORG_VARRAY(I + 1) := '6'
   -- 给变量 V_ORG_VARRAY 新追加的第一个条目赋值为 7。
   V_ORG_VARRAY(I + 2) := '7'
   FOR idx IN 1 .. V_ORG_VARRAY.COUNT()
   LOOP
   DBMS_OUTPUT.PUT_LINE(V_ORG_VARRAY(idx))
   END LOOP
END;
/
```

输出结果为：

output
1
2
3
4
5
6
7

例 173. 使用 COUNT 查询新的变量

```
!set plsqlUseSlash true
DECLARE
   I INT
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   I:=V_ORG_VARRAY.LAST()
   V_ORG_VARRAY.EXTEND(2)
   V_ORG_VARRAY(I +1) := 6
   V_ORG_VARRAY(I +2) := 7
   FOR I IN 1..V_ORG_VARRAY.COUNT()
   LOOP
      DBMS_OUTPUT.PUT_LINE('V('||I||')=' || V_ORG_VARRAY(I))
   END LOOP;
END;
/
```

输出结果为：

output
V(1)=1
V(2)=2
V(3)=3
V(4)=4
V(5)=5
V(6)=6
V(7)=7

NESTED TABLE

例 174. 使用 EXTEND() 对 NESTED TABLE 变量进行扩展

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个名为 list_of_names_t 的 table 类型，数据类型为字符串。
   TYPE list_of_names_t IS TABLE OF string
   I INT
   -- 定义一个名为 happyfamily 的 list_of_names_t 变量，并对集合元素进行初始化。
   happyfamily list_of_names_t:=list_of_names_t ()
BEGIN
   -- 给变量 happyfamily 赋值前，为其分配一个大小为 4 的空间，也就是最多可以容纳 4 个元素。
   happyfamily.EXTEND (4)
   -- 依次给变量 happyfamily 赋值
   happyfamily (1) := 'Veva'
   happyfamily (2) := 'Chris'
   happyfamily (3) := 'Eli'
   happyfamily (4) := 'Steven'
   -- 将变量 happyfamily 最后一个元素的下标，赋值给 I。
   I:= happyfamily.LAST()
   -- 使用 extend 方法，给变量 happyfamily 增加两个元素的个数。
   happyfamily.EXTEND (2)
   -- 依次给新增加的元素赋值为 lily，alice。
   happyfamily (I+1) := 'lily'
   happyfamily (I+2) := 'alice'
   -- 使用 count 方法，依次打印出变量 happyfamily 的值。
   FOR l_row IN 1 .. happyfamily.COUNT
   LOOP
      DBMS_OUTPUT.put_line (happyfamily (l_row))
   END LOOP;
END;
/
```

输出结果为：

output
Veva
Chris
Eli
Steven
lily
alice

对于 table 变量，由于在赋值前需要用到 EXTEND 方法对变量分配空间，但是如果空间不够，后续可以再增加。

8.11.10.4.5. TRIM(k)

- 使用 TRIM(k) 方法在集合变量的尾部删除最后 k 个条目。
- 当 k 没有被指定时，即 TRIM()，则删除最后一个条目，已被删除的条目的值将丢失。
- 使用 TRIM(COUNT()) 方法，则删除集合变量中的全部元素。
- k 不能大于集合变量的元素的最大个数。

VARRAY

例 175. 使用 TRIM(k) 方法删除部分条目

```
!set plsqlUseSlash true
DECLARE
   I INT
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   V_ORG_VARRAY.TRIM(2)
   FOR I IN 1..V_ORG_VARRAY.COUNT()
   LOOP
      DBMS_OUTPUT.PUT_LINE('V('||I||')=' || V_ORG_VARRAY(I))
      END LOOP;
END;
/
```

输出结果为：

output
V(1)=1
V(2)=2
V(3)=3

例 176. 使用 TRIM(k) 方法删除全部条目

```
!set plsqlUseSlash true
DECLARE
   I INT
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   V_ORG_VARRAY.TRIM(V_ORG_VARRAY.COUNT())
   FOR I IN 1..V_ORG_VARRAY.COUNT()
   LOOP
      DBMS_OUTPUT.PUT_LINE('V('||I||')=' || V_ORG_VARRAY(I))
   END LOOP;
END;
/
```

输出结果为：

output

NESTED TABLE

例 177. 使用 TRIM() 对 NESTED TABLE 变量进行缩减

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个名为 list_of_names_t 的 table 类型，数据类型为字符串。
   TYPE list_of_names_t IS TABLE OF string
   -- 定义一个名为 happyfamily 的 list_of_names_t 变量，并对集合元素进行初始化。
   happyfamily list_of_names_t:=list_of_names_t ()
BEGIN
   -- 给变量 happyfamily 赋值前，为其分配一个大小为 4 的空间，也就是最多可以容纳4个元素。
   happyfamily.EXTEND (4)
   -- 依次给变量 happyfamily 赋值为 vera，chris，eli，steven。
   happyfamily (1) := 'Vera'
   happyfamily (2) := 'Chris'
   happyfamily (3) := 'Eli'
   happyfamily (4) := 'Steven'
   -- 使用 TRIM(2) 方法，删除变量 happyfamily 最后两个元素。
   happyfamily.TRIM(2)
   -- 使用 count 方法，依次打印出变量 happyfamily 的值。
   FOR l_row IN 1 .. happyfamily.COUNT
   LOOP
      DBMS_OUTPUT.put_line (happyfamily (l_row))
   END LOOP;
END;
/
```

输出结果为：

output
Vera
Chris

Associative arrays

例 178. 不能使用 trim() 方法删除 Associative arrays 变量中的元素

```
!set plsqlUseSlash true
DECLARE
   TYPE numbers_aat IS TABLE OF STRING INDEX BY INT
   l_row INT
   l_numbers numbers_aat
BEGIN
   l_numbers (100) := 'LILY'
   l_numbers (90) := 'GRACE'
   l_numbers (80) := 'ALICE'
   l_numbers (95) := 'LISI'
   l_row:= l_numbers.first()
   l_row:= l_numbers.next(l_row)
   l_numbers.trim(2)
   DBMS_OUTPUT.put_line (l_row||' ' ||l_numbers(l_row))
END;
/
```

输出结果为：

```
EXECUTION FAILED: Task EXEC PLSQL error HiveException: [Error 20357] Error! You can not use
TRIM method with an associate array!
**********************************************
ANONYMOUS BLOCK (LINE 12, COLUMN 9, TEXT \"l_numbers.trim(2)\")
**********************************************
```

8.11.10.4.6. DELETE([k])

- DELETE(k) 方法，删除集合变量第 k 个条目。
- 当 k 没有被指定时，即 DELETE()，则删除集合变量中的全部元素。

VARRAY

VARRAY 不支持使用 DELETE(K) 删除集合指定位置的元素。

例 179. 使用 DELETE(K) 方法删除指定条目，将报错

```
!set plsqlUseSlash true
DECLARE
   I INT;
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING;
   V_ORG_VARRAY ORG_VARRAY_TYPE;
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5');
   V_ORG_VARRAY.DELETE(4);
   FOR I IN 1..V_ORG_VARRAY.COUNT()
   LOOP
      DBMS_OUTPUT.PUT_LINE('V('||I||')=' || V_ORG_VARRAY(I));
   END LOOP;
END;
/
```

输出结果为：

```
COMPILE FAILED: Semantic error: [Error 11135]
ANONYMOUS BLOCK (LINE 7, COLUMN 12, TEXT \"V_ORG_VARRAY.DELETE(4);\"): Wrong number of
arguments in call to 'DELETE'
```

例 180. 使用 DELETE() 方法删除全部条目

```
!set plsqlUseSlash true
DECLARE
   I INT
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5')
   V_ORG_VARRAY.DELETE()
   FOR I IN 1..V_ORG_VARRAY.COUNT()
   LOOP
   DBMS_OUTPUT.PUT_LINE('V('||I||')=' || V_ORG_VARRAY(I))
   END LOOP;
END;
/
```

输出结果为：

output

NESTED TABLE

例 181. 使用 DELETE(K) 对 NESTED TABLE 变量进行缩减

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个名为 list_of_names_t 的 table 类型，数据类型为字符串。
   TYPE list_of_names_t IS TABLE OF string
   -- 定义一个名为 happyfamily 的 list_of_names_t 变量，并对集合元素进行初始化。
   happyfamily list_of_names_t:=list_of_names_t ()
BEGIN
   -- 给变量 happyfamily 赋值前，为其分配一个大小为 4 的空间，也就是最多可以容纳 4 个元素。
   happyfamily.EXTEND (4)
   -- 依次给变量 happyfamily 赋值为 vera，chris，eli，steven。
   happyfamily (1) := 'Veva'
   happyfamily (2) := 'Chris'
   happyfamily (3) := 'Eli'
   happyfamily (4) := 'Steven'
   -- 使用 DELETE(2) 方法，删除变量 happyfamily 第 2 个元素。
   happyfamily.DELETE(2)
   -- 使用 count 方法，依次打印出变量 happyfamily 的值。
   FOR l_row IN 1 .. happyfamily.COUNT
   LOOP
   DBMS_OUTPUT.put_line(l_row||','||happyfamily(l_row))
   END LOOP;
END;
/
```

输出结果为：

output
1,Veva
2,Eli
3,Steven

Associative arrays

例 182. 使用 DELETE 方法删除 Associative arrays 变量中的元素

```
!set plsqlUseSlash true
DECLARE
   TYPE numbers_aat IS TABLE OF STRING INDEX BY INT
   l_row INT
   l_numbers numbers_aat
BEGIN
   l_numbers (100) := 'LILY'
   l_numbers (90) := 'GRACE'
   l_numbers (95) := 'LISI'
   l_numbers (80) := 'ALICE'
   -- 使用 DELETE(80) 方法删除变量 l_numbers 中的第一个，即下标为 80 的元素。
   l_numbers.delete(80)
   -- 将变量 l_numbers 中第一个元素的下标赋值给 l_row。
   l_row:= l_numbers.first()
   -- 将变量 l_numbers 中第一个元素的后一个元素的下标赋值给 l_row。
   l_row:= l_numbers.NEXT(l_row)
   -- 输出变量 l_numbers 中第一个元素的后一个元素的下标，及其所对应的值。
   DBMS_OUTPUT.put_line (l_row||' ' ||l_numbers(l_row))
END;
/
```

输出结果为：

可以看到，最初变量 l_numbers 的第一个元素为 80，在删除掉第一个元素之后，变量 l_numbers 的第一个元素的后一个元素的下标，也相应发生了变化。

output
95 LISI

8.11.10.4.7. EXISTS()

判断第i位是否存在，如果存在则返回 true，不存在则返回 false。

Varray

例 183. 使用 EXISTS() 方法，查询 VARRAY 变量中第 2 行的数据是否存在

```
!set plsqlUseSlash true
DECLARE
   -- 定义一个布尔类型的变量为 l。
   l boolean
   -- 声明一个能保存 10 个 STRING 数据类型的成员的 VARRAY 数据类型 ORG_VARRAY_TYPE。
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   -- 定义一个 ORG_VARRAY_TYPE 类型的 VARRAY 变量 V_ORG_VARRAY。
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   -- 依次给变量 V_ORG_VARRAY 赋值。
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','1','3','4','5')
   -- 使用 EXISTS() 方法方法，判断变量 V_ORG_VARRAY 第二个元素是否存在，并赋值给 l。
   l:=V_ORG_VARRAY.exists(2)
   -- 输出变量 l 的值。
   DBMS_OUTPUT.PUT_LINE(l)
END;
/
```

输出结果为：

output
TRUE

Associative Arrays

例 184. 使用 EXISTS() 方法，查询 Associative Arrays 变量中 key 为 95 的元素是否存在

```
-- 输出变量 l 的值。
!set plsqlUseSlash true
DECLARE
   TYPE numbers_aat IS TABLE OF STRING INDEX BY INT
   l_row INT
   -- 定义一个布尔类型的变量为 l。
   l boolean
   l_numbers numbers_aat
BEGIN
   l_numbers (100) := 'LILY'
   l_numbers (90) := 'GRACE'
   l_numbers (80) := 'ALICE'
   l_numbers (95) := 'LISI'
   -- 使用 EXISTS() 方法方法，判断变量 l_numbers 中是否存在 key 等于 95 的元素，并赋值给 l。
   l := l_numbers.exists(95)
   DBMS_OUTPUT.put_line (l)
END;
/
```

输出结果为：

output
TRUE

8.11.10.4.8. LIMIT()

对于 VARRAY 类型，返回声明类型时，所定义的最大容量，在 Inceptor 中会返回 null 值。

例 185. Inceptor 中对变量 VARRAY 使用 LIMIT() 方法，会报出 null 值

```
!set plsqlUseSlash true
DECLARE
   l INT
   TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING
   V_ORG_VARRAY ORG_VARRAY_TYPE
BEGIN
   V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5','6','7','8','9','10')
   l:=V_ORG_VARRAY.limit()
   DBMS_OUTPUT.PUT_LINE(l)
END;
/
```

输出结果为：

output
null

例 186. Inceptor 中对变量 NESTED TABLE 使用 LIMIT() 方法，会报出 null 值

```
!set plsqlUseSlash true
DECLARE
   l INT
   TYPE list_of_names_t IS TABLE OF string
   happyfamily list_of_names_t:=list_of_names_t ()
BEGIN
   happyfamily.EXTEND(4)
   happyfamily (1) := 'Veva'
   happyfamily (2) := 'Chris'
   happyfamily (3) := 'Eli'
   happyfamily (4) := 'Steven'
   l:=happyfamily.limit()
   DBMS_OUTPUT.PUT_LINE(l)
END;
/
```

输出结果为:

output
null

8.11.11. 游标（Cursors）

在 PL/SQL 中，游标是指针，指向一段 DML 语句（INSERT/UPDATE/DELETE/MERGE）或者查询语句（SELECT）的上下文区域（context area）。PL/SQL 中，游标分为 **显式游标（explicit cursor）** 和 **隐式游标（implicit cursor）**。在声明游标时可以带有或不带有参数。

8.11.11.1. 显式游标

显示游标可以在 SELECT 语句上创建，它使用的步骤为：

1. 在声明部分声明游标：

   语法：

   ```
   DECLARE CURSOR cursor_name IS select_statement;
   ```

2. 在执行部分或异常处理部分打开游标：

   语法：

   ```
   OPEN cursor_name;
   ```

3. 取数据：

   语法：

   ```
   FETCH cursor_name;
   ```

4. 关闭游标：

语法：

```
CLOSE cursor_name;
```

表 10. 显式游标的属性

游标的属性	返回值类型	意义
%ROWCOUNT	整型	获得 FETCH 语句返回的数据行数
%ROWTYPE	记录类型	游标返回结果的记录类型
%FOUND	布尔型	最近的 FETCH 语句返回一行数据则为 TRUE，否则为 FALSE
%NOTFOUND	布尔型	与 %FOUND 属性返回值相反
%ISOPEN	布尔型	游标已经打开时值为 TRUE，否则为 FALSE

下面我们用一些例子来具体介绍显式游标的使用方法。

8.11.11.1.1. 不带参数的显式游标使用示例

例 187. 在 LOOP 中声明一个显式游标，查询表 transactions 中的价格

```
!set plsqlUseSlash true
DECLARE
    -- 定义一个字段名和类型与表 transactions 均相同的记录变量 transactions_type。
    transactions_type transactions%ROWTYPE
    -- 定义一个显式游标 cur，用来查询表 transactions 中的全部信息。
    CURSOR cur IS SELECT * FROM transactions;
BEGIN
    -- 打开游标 cur。
    OPEN cur
    LOOP
        -- 把游标 cur 查询到的信息放进变量 transactions_type 中。
        FETCH cur INTO transactions_type
        -- 如果游标 cur 没有查询到信息，就退出。
        EXIT WHEN cur%NOTFOUND
        -- 输出变量 transactions_type 中的字段 price 的值。
        DBMS_OUTPUT.PUT_LINE(transactions_type.price)
    END LOOP;
    CLOSE cur
    EXCEPTION
    -- 声明一个异常情况名 OTHERS。
    WHEN OTHERS THEN
        -- 如果异常 OTHERS 发生，则返回的值。
        DBMS_OUTPUT.PUT_LINE('Something unexpected happened!!')
        CLOSE cur
END;
/
```

输出结果为：

output
12.13
11.11
6.12
4.5
22.66
68.43

output
5.3
7.52
9.81
12.21
4.16
5.25
6.36
7.49
8.64
10.31
7.02
9.16
10.03
18.38

例 188. 在 FOR LOOP 中声明一个显式游标，查询表 transactions 中的账号，交易时间和价格

```
!set plsqlUseSlash true
DECLARE
    -- 定义一个字段名和类型与表 transactions 均相同的记录变量 transactions_type。
    transactions_type transactions%ROWTYPE
    -- 定义一个显式游标 cur，用来查询表 transactions 中的全部信息。
    CURSOR cur IS SELECT * FROM transactions;
BEGIN
    -- for..loop 循环语句,对于每一个在游标里的变量值。
    FOR transactions_type IN cur
    LOOP
    -- 输出变量 transactions_type 的字段名，账号，交易时间和价格的值。
    DBMS_OUTPUT.PUT_LINE( transactions_type.acc_num || ' ' ||transactions_type.trans_time || '
'  ||transactions_type.price)
    END LOOP;
END;
/
```

输出结果为：

output
6513065 20140105100520 12.13
3912384 20140205140521 11.11
6513065 20140506133109 6.12
3912384 20140430111523 4.5
0700735 20140315111111 22.66

output
3912384 20140328102400 68.43
0700735 20140611102830 5.3
2755506 20140702113025 7.52
6513065 20140916105811 9.81
2394923 20141031135018 12.21
3912384 20140214141519 4.16
0700735 20140430143020 5.25
5224133 20140801110003 6.36
6513065 20141225133500 7.49
6670192 20141130113905 8.64
6670192 20140314145958 10.31
6513065 20140628133001 7.02
6600641 20140228140005 9.16
5224133 20140331115900 10.03
6513065 20140508094805 18.38

例 189. 定义一个 cursor%rowtype 类型的变量来查询 transactions 表中的交易号、交易时间和所对应的股票价格

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个名为 transactions_cursor 的游标
   CURSOR transactions_cursor IS SELECT * FROM transactions;
   -- 声明一个 transactions_cursor%rowtype 类型的记录变量
   transactions_record transactions_cursor%ROWTYPE
BEGIN
   OPEN transactions_cursor
   LOOP
       FETCH transactions_cursor INTO transactions_record
       EXIT WHEN transactions_cursor%NOTFOUND;
       dbms_output.put_line(transactions_record.acc_num || ' ' ||transactions_record.trans_time
|| ' '  ||transactions_record.price);
      END LOOP;
   CLOSE transactions_cursor;
END;
/
```

查询结果如下，结果和上例一致：

output
6513065 20140105100520 12.13
3912384 20140205140521 11.11
6513065 20140506133109 6.12
3912384 20140430111523 4.5
0700735 20140315111111 22.66
3912384 20140328102400 68.43
0700735 20140611102830 5.3
2755506 20140702113025 7.52
6513065 20140916105811 9.81
2394923 20141031135018 12.21
3912384 20140214141519 4.16
0700735 20140430143020 5.25
5224133 20140801110003 6.36
6513065 20141225133500 7.49
6670192 20141130113905 8.64
6670192 20140314145958 10.31
6513065 20140628133001 7.02
6600641 20140228140005 9.16
5224133 20140331115900 10.03
6513065 20140508094805 18.38

8.11.11.1.2. 带参数的显式游标使用示例

例 190. 在 LOOP 中定义一个带有参数的游标，查询表 transactions 中账号为 6513065 的交易时间。

```
!set plsqlUseSlash true
DECLARE
    -- 定义一个带有参数的游标 cur 去查询表 transactions 中账号和参数的值相等的全部信息，其中参数名为
tacc_num，类型为 string。
    CURSOR cur(tacc_num string) IS SELECT * FROM transactions WHERE acc_num=tacc_num;
    -- 定义一个字段名和类型与表 transactions 均相同的记录变量 transactions_type。
    transactions_type transactions%rowtype
BEGIN
    -- 打开游标 cur，并赋参数值为 6513065。
    OPEN cur('6513065')
    LOOP
        -- 把游标 cur 查询到的信息放进变量 transactions_type 中。
        FETCH cur INTO transactions_type
        EXIT WHEN cur%NOTFOUND
        -- 输出变量 transactions_type 中的交易时间。
        DBMS_OUTPUT.PUT_LINE(transactions_type.trans_time)
    END LOOP;
    CLOSE cur
    -- 如果有情况名为 OTHERS 的异常发生，则返回 Something unexpected happened!! 的信息。
    EXCEPTION
        WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Something unexpected happened!!')
        CLOSE cur
END;
/
```

输出结果为：

output
20140105100520
20140506133109
20140916105811
20141225133500
20140628133001
20140508094805

例 191. 在 FOR LOOP 中定义一个游标，查询表 transactions 中账号为 6513065 的交易时间和价格

```
!set plsqlUseSlash true
DECLARE
    -- 定义一个字段名和类型与表 transactions 均相同的记录变量 transactions_type。
    transactions_type transactions%ROWTYPE
    -- 定义一个带有参数的游标 cur 去查询表 transactions 中账号和参数的值相等的全部信息，其中参数名为
tacc_num，类型为 string。
    CURSOR cur(tacc_num string) IS SELECT * FROM transactions WHERE acc_num=tacc_num;
BEGIN
    -- for..loop 循环，对于每一个在游标 cur('6513065') 中的变量 transactions_type 的值。
    FOR transactions_type IN cur('6513065')
    LOOP
        -- 输出表 transactions 中账号为 6513065 的账号，交易时间和价格。
        DBMS_OUTPUT.PUT_LINE( transactions_type.acc_num || ' ' || ' ' || transactions_type
.trans_time  ||transactions_type.price)
    END LOOP;
END;
/
```

返回结果如下：

output
6513065 2014010510052012.13
6513065 201405061331096.12
6513065 201409161058119.81
6513065 201412251335007.49
6513065 201406281330017.02
6513065 2014050809480518.38

例 192. 声明一个显式游标，通过 cursor%rowtype 定义变量，查询表 transactions 中账号为 6513065 的交易时间和价格

```
!set plsqlUseSlash true
DECLARE
    -- 定义一个显式游标 cur，用来查询表 transactions 中的账号、交易时间以及价格信息。
    CURSOR cur(tacc_num string) IS SELECT acc_num, trans_time, price FROM transactions WHERE
acc_num = tacc_num;
    -- 定义一个字段名和类型与表 transactions.acc_num、transactions.trans_time、transactions.price
均相同的记录变量 transactions_type。
    transactions_type cur%ROWTYPE;
BEGIN
    -- Open Cursor，并传入参数值 6513065。
    OPEN cur('6513065')
    LOOP
        -- 将 cur 的结果放入 transactions_type。
        FETCH cur INTO transactions_type
        EXIT WHEN cur%NOTFOUND
        -- 输出变量 transactions_type 的字段名：账户、交易时间和价格。
        DBMS_OUTPUT.PUT_LINE( transactions_type.acc_num || ' ' || transactions_type.trans_time
|| ' ' || transactions_type.price)
    END LOOP;
    CLOSE cur
    EXCEPTION
    WHEN OTHERS THEN
        -- 出现异常时的显示信息。
        DBMS_OUTPUT.PUT_LINE('Something unexpected happened!!')
        CLOSE cur
END;
/
```

输出结果上例一致：

output
6513065 20140105100520 12.13
6513065 20140506133109 6.12
6513065 20140916105811 9.81
6513065 20141225133500 7.49
6513065 20140628133001 7.02
6513065 20140508094805 18.38

8.11.11.2. 隐式游标

隐式游标是没有明确的声明语句的游标类型。所有的 DML 操作都被 Inceptor 内部解析为一个游标名为 SQL 的隐式游标。

表 11. 隐式游标的属性

游标的属性	返回值类型	意义
SQL%ROWCOUNT	整型	代表 DML 语句成功执行的数据行数
SQL%FOUND	布尔型	值为 TRUE 代表插入、删除、更新或单行查询操作成功

游标的属性	返回值类型	意义
SQL%NOTFOUND	布尔型	与 SQL%FOUND 属性返回值相反
SQL%ISOPEN	布尔型	永远为 FALSE

说明

- 对于 SELECT INTO 语句，如果执行成功，SQL%ROWCOUNT 的值为 1，如果没有成功，SQL%ROWCOUNT 的值为 0，同时产生一个异常 NO_DATA_FOUND。
- 在执行 DML 语句后，SQL%FOUND 的属性值将为 TRUE。

8.11.11.2.1. 隐式游标使用示例

例 193. 指向 DML 语句的隐式游标，使用 update 语句对表 zara 进行更新

```
!set plsqlUseSlash true
BEGIN
    UPDATE zara SET name='grace' WHERE age=20
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('name is updated')
    ELSE
        DBMS_OUTPUT.PUT_LINE('name is not updated')
    END IF
END;
/
```

输出结果为：

output
name is not updated

8.11.11.3. 游标变量

像游标一样，游标变量指向指定查询结果集当前行，但是相对游标，游标变量更加灵活因为其声明并不绑定指定查询。要定义一个游标变量，您需要先声明一个游标类型 **REF CURSOR**，然后定义这个类型的游标变量。

游标变量的使用步骤如下：

1. 声明游标类型：

   语法：

   ```
   DECLARE TYPE type_name IS REF CURSOR;
   ```

2. 声明该类型的游标变量：

   语法

   ```
   cursor_name type_name;
   ```

3. 在执行部分或异常处理部分打开：

语法：

```
OPEN cursor_name FOR sql_statement;
```

4. 取数据：

   语法：

```
FETCH cursor_name;
```

5. 关闭游标：

   语法：

```
CLOSE cursor_name;
```

游标变量可分为两类，强类型和弱类型的：

- 如果在声明游标类型 REF CURSOR 的时候指定了返回类型，那么 REF CURSOR 及其类型的游标变量被称为 **强类型**。
- 如果在声明游标类型 REF CURSOR 的时候不指定返回类型，那么 REF CURSOR 及其类型的游标变量被称为 **弱类型**。

8.11.11.3.1. 游标变量使用示例

强类型游标变量

例 194. 查询表 transactions 中价格为 12.13 的交易号和交易时间

```
!set plsqlUseSlash true
DECLARE
    -- 声明一个名为 cur_transaction 的游标类型，指定返回值的类型为 transactions%rowtype。
    TYPE cur_transaction IS REF CURSOR RETURN transactions%ROWTYPE
    -- 定义一个名为 sqlcur 的游标变量。
    sqlcur cur_transaction
    -- 依次定义两个类型为字符串的变量 v_trans_id,v_trans_time。
    v_trans_id STRING
    v_trans_time STRING
BEGIN
    -- 打开游标 sqlcur，去查询表 transactions中价格为 12.13 的交易号和交易时间。
    OPEN sqlcur FOR SELECT trans_id,trans_time FROM transactions WHERE price=12.13;
    LOOP
        -- 将游标查询的结果放进变量 v_trans_id,v_trans_time。
        FETCH sqlcur INTO v_trans_id,v_trans_time
        EXIT WHEN sqlcur%NOTFOUND
        -- 输出变量 v_trans_id,v_trans_time 的值。
        DBMS_OUTPUT.PUT_LINE(v_trans_id||' ' ||v_trans_time)
    END LOOP;
    CLOSE sqlcur
END;
/
```

输出结果为：

output
943197522 20140105100520

弱类型游标变量

例 195. 查询表 transactions 中价格为 12.13 的交易号和交易时间

```
!set plsqlUseSlash true
DECLARE
    -- 声明一个名为 cur_transaction 的游标类型，没有定义返回值的类型。
    TYPE cur_transaction IS REF CURSOR
    -- 定义一个名为 sqlcur 的 cur_transaction 游标变量。
    sqlcur cur_transaction
    -- 依次定义两个类型为字符串的变量 v_trans_id,v_trans_time。
    v_trans_id STRING
    v_trans_time STRING
BEGIN
    -- 打开游标 sqlcur，去查询表 transactions 中价格为 12.13 的交易号和交易时间。
    OPEN sqlcur FOR SELECT trans_id,trans_time FROM transactions WHERE price=12.13;
    LOOP
        -- 将游标查询的结果放进变量 v_trans_id,v_trans_time。
        FETCH sqlcur INTO v_trans_id,v_trans_time
        EXIT WHEN sqlcur%NOTFOUND
        -- 输出变量 v_trans_id,v_trans_time 的值。
        DBMS_OUTPUT.PUT_LINE(v_trans_id||' ' ||v_trans_time)
    END LOOP;
    CLOSE sqlcur
END;
/
```

输出结果为：

output
943197522 20140105100520

游标变量作为参数传递

例 196. 将游标变量作为参数传递

```
!set plsqlUseSlash true
CREATE OR REPLACE PACKAGE pkg_a IS
    -- 在包 pkg_a 中定义一个游标类型，和表 user_info 的行类型相同。
    TYPE empcurtyp IS REF CURSOR RETURN user_info%ROWTYPE
END pkg_a;
/

-- 定义一个过程 process_emp_cv，这个过程会一行行读取 user_info 中的记录的 name。
CREATE OR REPLACE PROCEDURE process_emp_cv(emp_cv in pkg_a.empcurtyp)
IS
    user user_info%ROWTYPE
BEGIN
    LOOP
        FETCH emp_cv INTO user
        EXIT WHEN emp_cv%NOTFOUND
        DBMS_OUTPUT.PUT_LINE('Name = ' || user.name)
    END LOOP;
END;
/
DECLARE
    emp pkg_a.empcurtyp
BEGIN
    -- 声明游标变量 emp，类型为在包 pkg_a 中定义的游标类型。
    OPEN emp FOR SELECT * FROM user_info;
    -- 将游标变量 emp 座位参数传给过程 process_emp_cv。
    process_emp_cv(emp)
    CLOSE emp
END;
/
```

输出结果为：

output
Name = 马**
Name = 祝**
Name = 华*
Name = 魏**
Name = 宁**
Name = 邱*
Name = 李*
Name = 潘**
Name = 李**
Name = 管**

8.11.12. 与 SQL 的交互

8.11.12.1. PL/SQL 过程与 SQL 的交互

Inceptor 中可以利用 PL/SQL 过程对 Inceptor 中的表进行增删改查等操作。

8.11.12.1.1. INSERT

例 197. 创建过程 test()，往非分区表 za 中插入一条记录

```
!set plsqlUseSlash true
-- 创建一个名为 test() 的过程。
CREATE OR REPLACE PROCEDURE
test() IS
BEGIN
   -- 往表 za 中插入一条数据 ('lily',20)。
   INSERT INTO za values ('lily',20)
   -- IF 条件语句为如果 SQL 语句执行了。
   IF SQL%FOUND THEN
      -- 成功执行 sql 语句所输出的内容。
      dbms_output.put_line('successfully insert new data')
   ELSE
      -- 没有成功执行 sql 语句所输出的内容。
      dbms_output.put_line('nothing')
   END IF
END;
/
BEGIN
test()
END;
/
```

输出结果为：

output
successfully insert new data

8.11.12.1.2. DELETE

例 198. 创建过程 test()，删除非分区表 za 中年龄为 40 的记录

```
!set plsqlUseSlash true
-- 创建一个名为 test() 的过程。
CREATE OR REPLACE PROCEDURE
test() IS
BEGIN
   -- 删除表 za 中年龄为 20 的所有记录。
   DELETE FROM za WHERE age=20
   -- IF 条件语句为如果 SQL 语句执行了。
   IF SQL%FOUND THEN
      -- 成功执行 sql 语句所输出的内容。
      dbms_output.put_line('successfully delete the data')
   ELSE
      -- 没有成功执行 sql 语句所输出的内容。
      dbms_output.put_line('nothing')
   END IF
END;
/

BEGIN
test()
END;
/
```

输出结果为：

output
successfully delete the data

8.11.12.1.3. UPDATE

例 199. 创建过程 test(),更新表 ra 中姓名为 smith 的 gpa 为 3.4

```
!set plsqlUseSlash true
-- 创建一个名为 test() 的过程。
CREATE OR REPLACE PROCEDURE
test() IS
BEGIN
   -- 更新表 ra 中姓名为 smith 的绩点为 3.4。
   UPDATE ra SET gpa=3.4
    WHERE name='smith'
   -- IF 条件语句为如果 SQL 语句执行了。
   IF SQL%FOUND THEN
      -- 成功执行 sql 语句所输出的内容。
      dbms_output.put_line('successfully update the data')
   ELSE
   -- 没有成功执行 sql 语句所输出的内容。
      dbms_output.put_line('nothing')
   END IF
END;
/

BEGIN
test()
END;
/
```

输出结果为:

output
successfully update the data

8.11.12.1.4. MERGE

仅满足条件的情况下，更新

例 200. 创建过程 test()，查找 za 表中与 zara 表中姓名相同的人的记录，如果相同就把 zara 表中对应记录的 gpa 改为 2。

```
!set plsqlUseSlash true
-- 创建一个名为 test() 的过程。
CREATE OR REPLACE PROCEDURE
test() IS
BEGIN
   -- 对表 zara 进行 merge 操作，条件为表 zara 和表 za 姓名相同。
   MERGE INTO zara z using za a on (z.name=a.name)
   WHEN MATCHED
    -- 如果表 za 和表 zara 的姓名相同，就把表 zara 中姓名和表 za 中相同的人的绩点更新为 4。
    THEN UPDATE SET z.gpa=4
   -- IF 条件语句为如果 SQL 语句执行了。
   IF SQL%FOUND THEN
      -- 成功执行 sql 语句所输出的内容。
      dbms_output.put_line('successfully update the data ')
   ELSE
   -- 没有成功执行 sql 语句所输出的内容。
      dbms_output.put_line('nothing')
   END IF
END;
/
BEGIN
test()
END;
/
```

输出结果为：

output
nothing

满足条件更新，不满足条件插入

例 201. 创建过程 test1()，查找 zara 表中姓名和 za 中姓名相同的记录，如果相同就把 za 表中的年龄改为 30，否则就把 zara 表中与 za 中姓名不相同的姓名和年龄插入到 za 表中

注意，被更新的字段不允许为表的分区字段或者分桶字段。

```
!set plsqlUseSlash true
-- 创建一个名为 test1() 的过程。
CREATE OR REPLACE PROCEDURE
test1() IS
BEGIN
   -- 对表 za 进行 merge 操作，条件为表 zara 和表 za 姓名相同。
   MERGE INTO za z using zara r on (z.name=r.name)
   -- 如果表 za 和表 zara 的姓名相同，就把表 za 中姓名和表 zara 中相同的人的年龄更新为 40。
   WHEN MATCHED THEN UPDATE SET z.age=40
   -- 如果表 za 和表 zara 的姓名不相同，就把表 zara 中和表 za 中姓名不相同的人的姓名和年龄，插入到表 za 中。
   WHEN NOT MATCHED THEN INSERT (name,age) VALUES  (r.name,r.age)
   -- IF 条件语句为如果 SQL 语句执行了。
   IF SQL%FOUND THEN
      -- 成功执行 sql 语句所输出的内容。
      dbms_output.put_line('successfully update data and insert data')
   ELSE
      -- 没有成功执行 sql 语句所输出的内容。
      dbms_output.put_line('nothing')
   END IF
END;
/
BEGIN
test1()
END;
/
```

输出结果为：

output
successfully update data and insert data

8.11.12.2. PL/SQL 函数与 SQL 的交互

Inceptor 中的 SQL 语句可以直接调用自定义的 PLSQL 函数，但对 PLSQL 函数有如下要求：

- 必须是函数，不能是过程。
- 返回值必须是基本类型。
- 函数中不能有标准 SQL 语句。

例 202. 创建函数 hello_message

```
!set plsqlUseSlash true
-- 创建函数的语句。
CREATE OR REPLACE FUNCTION
-- 函数名为 hello_message，形参为 place_in，数据类型为字符串。
hello_message (place_in string)
-- 函数 hello_message 的返回值的类型为字符串。
RETURN string
IS
-- 函数 hello_message 的具体内容。
BEGIN
   RETURN 'Hello ' || place_in
END;
/
```

例 203. 调用函数 hello_message

```
!set plsqlUseSlash true
BEGIN
   -- 往表 ra 中插入一条数据，其中姓名为函数来代替，即 hello_message('tom')。
   INSERT INTO ra VALUES (hello_message('tom'),4.8)
   -- IF 条件语句为如果 SQL 语句执行了。
   IF SQL%FOUND THEN
      -- 成功执行 sql 语句所输出的内容。
      dbms_output.put_line('successfully insert new data')
   ELSE
      -- 没有成功执行 sql 语句所输出的内容。
      dbms_output.put_line('nothing')
   END IF
END;
/
```

输出结果为：

output
successfully insert new data

8.11.12.3. 隐式游标

例 204. 使用 update 语句对表 zara 进行更新

```
!set plsqlUseSlash true
BEGIN
-- 把表 zara 中年龄为 20 的记录的姓名更新为 grace，此处是用 DML 语句声明的一个隐式游标。
   UPDATE zara SET name='grace' WHERE age=20
   -- IF 条件语句为如果 SQL 语句执行了。
   IF SQL%FOUND THEN
      -- 成功执行 sql 语句所输出的内容。
      dbms_output.put_line('name is updated')
   ELSE
      -- 没有成功执行 sql 语句所输出的内容。
      dbms_output.put_line('name is not updated')
   END IF
END;
/
```

输出结果为：

output
name is updated

8.11.12.4. BULK COLLECT

Inceptor 中支持 BULK COLLECT 的用法，即可以一次查询多行的结果，放进一个 NESTED TABLE 类型的变量里，但是，Inceptor 不支持将查询结果放进 Associative arrays 等集合类型中。

8.11.12.4.1. 在 fetch into 中使用 bulk collect

例 205. 声明一个游标，查询表 transactions 中账号为 6513065 的交易号和交易时间

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个名为 trans_type 的记录类型，分量名分别为 trans_id，trans_time，数据类型均为字符串。
   TYPE trans_type is RECORD(trans_id string,trans_time string)
   -- 声明一个名为 cur 的游标，去查询表 transactions 中账号为 6513065 的交易号和交易时间。
   CURSOR cur IS
     SELECT trans_id,trans_time
      FROM transactions WHERE acc_num=6513065;
   -- 声明一个名为 transactions_type 的 NESTED TABLE 类型，用来存放数据类型为 trans_type 的变量。
   TYPE transactions_type IS TABLE OF trans_type
   -- 定义一个类型为 transactions_type 的变量 v_transactions_type。
   v_transactions_type transactions_type
BEGIN
   -- 打开游标。
   OPEN cur
     LOOP
        -- 将游标查询到的结果放进变量 v_transactions_type。
        FETCH cur BULK COLLECT INTO v_transactions_type
      -- 如果游标没有查询到结果就退出。
      EXIT WHEN cur%notfound
      END LOOP;
      -- 关闭游标。
      CLOSE cur
   -- 输出变量 v_transactions_type 的行数。
   dbms_output.put_line(v_transactions_type.count())
END;
/
```

输出结果为：

output
6

8.11.12.4.2. 在 select into 中使用 bulk collect

例 206. 声明一个表类型的变量，查询表 transactions 中的交易号，交易时间，和价格

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个名为 trans_type 的记录类型，分量名分别为 trans_id，trans_time，price，数据类型分别为
string,string,double。
   TYPE trans_type is RECORD(trans_id string,trans_time string,price double)
   -- 声明一个名为 transactions_type 的 NESTED TABLE 类型，用来存放数据类型为 trans_type 的变量。
   TYPE transactions_type IS TABLE OF trans_type
   -- 定义一个类型为 transactions_type 的变量 v_transactions_type。
   v_transactions_type transactions_type
BEGIN
   -- 查询表 transactions 中的交易号，交易时间和价格信息，并放进变量 v_transactions_type 中。
   SELECT trans_id,trans_time,price
     BULK COLLECT INTO v_transactions_type
      FROM transactions;
   -- IF 条件语句为如果 sql 语句执行了。
   IF SQL%FOUND THEN
      -- 成功执行 sql 语句所输出的内容。
      dbms_output.put_line('successfully select the data into v_transactions_type')
   ELSE
      -- 没有成功执行 sql 语句所输出的内容。
      dbms_output.put_line('nothing')
   END IF
END;
/
```

输出结果为：

output
successfully select the data into v_transactions_type

8.11.12.5. 动态 SQL

- 动态 SQL 是每一次运行时都会生成和执行 SQL 语句的编程方法。动态 SQL 对编写具有通用目标且灵活可变的项目很有帮助，如必须执行 DDL 语句，在编译时间不知道 SQL 语句，数据类型的输入，输出变量的全部内容等情况下，动态 SQL 可以很好的解决问题。
- PL/SQL 中支持使用 Native Dynamic SQL 去编写动态 SQL 语句，即常使用 EXECUTE IMMEDIATE 语句，OPEN-FOR，FETCH 和 CLOSE 语句等来执行动态 SQL 语句，但 Inceptor 中不支持系统预定义包 DBMS_SQL，也就不支持使用包 DBMS_SQL，来创建，执行，描述动态 SQL 语句。

8.11.12.5.1. EXECUTE IMMEDIATE

Native Dynamic SQL 中最常使用 EXECUTE IMMEDIATE 语句来执行动态 SQL 语句，使用 EXECUTE IMMEDIATE 语句可以创建执行 DDL 语句，DCL 语句，DML 语句以及单行的 SELECT 语句，但该方法不能用于处理多行查询语句。

处理DDL语句

CREATE语句

- 查看数据库中的表

```
show tables;
```

tab_name
finances

tab_name
ra
transactions
user_info
za
zara
zza

例 207. 使用 EXECUTE IMMEDIATE 创建表 test_create

```
!set plsqlUseSlash true
DROP TABLE IF EXISTS test_create;
DECLARE
  create_table STRING
BEGIN
  -- 注意字符串与连接符之间的空格，否则 {inceptor} 可能会报错。
  create_table:='create table ' ||'test_create' ||'(sno string , sname string )'
EXECUTE IMMEDIATE create_table
END;
/
```

再次查看数据库中的表，可以发现多了一张名为"test_create"的表

```
show tables;
```

tab_name
finances
ra
test_create
transactions
user_info
za
zara
zza

ALTER语句

- 查看 test_create 表的结构

```
DESC test_create;
```

col_name	data_type	default_value	not_null	unique	comment
sno	string	NULL	No	No	
sname	string	NULL	No	No	

例 208. 使用 EXECUTE IMMEDIATE 修改表 test_create 中列 sno 的列名和数据类型

```
!set plsqlUseSlash true
DECLARE
   alter_table STRING
BEGIN
    alter_table:='alter table test_create CHANGE sno sid INT'
EXECUTE IMMEDIATE alter_table
END;
/
```

- 再次查看 test_create 表的结构

```
DESC test_create;
```

col_name	data_type	default_value	not_null	unique	comment
sid	int	NULL	No	No	
sname	string	NULL	No	No	

DROP语句

- 现在我们来用 EXECUTE IMMEDIATE 删除表 test_create

例 209. 使用 EXECUTE IMMEDIATE 删除表 test_create

```
!set plsqlUseSlash true
DECLARE
    drop_table STRING
BEGIN
    drop_table:='drop table '||'test_create';
EXECUTE IMMEDIATE drop_table;
END;
/
```

再次查看数据库中的表，可以发现表 test_create 已经不存在了。

```
show tables;
```

tab_name
finances
ra
transactions
user_info
za
zara
zza

处理 DCL 语句

GRANT

例 210. 创建过程 grant_priv

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE grant_priv(priv STRING, username STRING)
IS
priv_stat STRING
BEGIN
priv_stat:=' GRANT '|| priv || ' TO user ' || username
EXECUTE IMMEDIATE priv_stat
END;
/
BEGIN
 grant_priv('create', 'user1')
END;
/
```

查看 user1 被赋予的权限

```
SHOW GRANT USER user1 ON all;
```

database	table	partition	column	principal_name	principal_type	privilege	grant_option	grant_time	grantor
				user1	USER	CREATE	false	0	N/A

REVOKE

例 211. 创建过程 revoke_priv

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE revoke_priv(priv STRING, username STRING)
IS
priv_stat STRING
BEGIN
priv_stat:=' revoke '|| priv || ' from user  ' || username
EXECUTE IMMEDIATE priv_stat
END;
/
BEGIN
  revoke_priv('create', 'user1')
END;
/
```

再次查看 user1 的权限

可以看到，此时 user1 不再具有 CREATE 的权限

```
SHOW GRANT USER user1 ON all;
```

database	table	partition	column	principal_name	principal_type	privilege	grant_option	grant_time	grantor

处理单行查询

例 212. 使用 EXECUTE IMMEDIATE 查询表中数据

```
!set plsqlUseSlash true
set plsql.show.sqlresults=true;
DECLARE
 sql_stat STRING
BEGIN
 sql_stat:='select price from transactions where acc_num=6513065'
EXECUTE IMMEDIATE sql_stat
END;
/
```

输出结果

output
12.13
6.12
9.81
7.49
7.02
18.38

在 Inceptor 中，要想打印出 SELECT 查询的结果，需要手动设置命令，具体用法说明，可参见注意事项章节。

处理不含占位符的 DML 语句

INSERT+UPDATE+DELETE

例 213. 使用 EXECUTE IMMEDIATE 语句，分别往表 holotest2 中插入，更新和删除数据

```
!set plsqlUseSlash true
DECLARE
   -- 分别声明三个字符串类型的变量 insert_table，update_table，delete_table。
   insert_table STRING
   update_table STRING
   delete_table STRING
BEGIN
   -- 将SQL语句“往表 holotest2 中插入一条数据”赋值给变量 insert_table。
   insert_table:= 'INSERT INTO holotest2 values(20,4.0)'
   -- 使用关键字 EXECUTE IMMEDIATE 执行变量 insert_table 里的 SQL 语句。
     EXECUTE IMMEDIATE insert_table
   -- 将 SQL 语句“更新表 holotest2 中年龄为 20 的绩点为 4.4”赋值给变量 update_table。
   update_table:=' UPDATE holotest2 SET gpa=4.4 WHERE age=20'
     EXECUTE IMMEDIATE update_table
   -- 将 SQL 语句“删除表 holotest2 中年龄为 20 的记录”赋值给变量 delete_table。
   delete_table:='DELETE FROM holotest2 WHERE age=20'
     EXECUTE IMMEDIATE delete_table
END;
/
```

CREATE+INSERT

例 214. 使用 EXECUTE IMMEDIATE 创建表格并插入数据

```
!set plsqlUseSlash true
DROP TABLE IF EXISTS holotest2;
DECLARE
  create_table string
  insert_table string
BEGIN
  create_table:='create table holotest2 (age int, gpa double)
                 CLUSTERED BY (age) INTO 2 BUCKETS
                 STORED AS HOLODESK'
EXECUTE IMMEDIATE create_table
  insert_table:='insert into holotest2 values (22,4.3)'
EXECUTE IMMEDIATE insert_table
END;
/
```

- 查看表 holotest2 及表中的数据

```
DESC holotest2;
```

col_name	data_type	default_value	not_null	unique	comment
age	int	NULL	No	No	
gpa	double	NULL	No	No	

可以看到在成功创建 holotest2 的同时，也成功插入了一条数据

```
SELECT * FROM holotest2;
```

age	gpa
22	4.3

CREATE+INSERT+UPDATE+DELETE+INSERT

例 215. 使用 EXECUTE IMMEDIATE 处理一系列 DML 语句

```
!set plsqlUseSlash true
DROP TABLE IF EXISTS holotest3;
DECLARE
  create_table STRING
  insert_table STRING
  update_table STRING
  delete_table STRING
BEGIN
  -- 创建 HOLODESK 表 holotest3，字段名分为 pid，price，数据类型分为 INT，DOUBLE
  create_table:='CREATE TABLE holotest3 (pid int, price double)
                 CLUSTERED BY (pid) INTO 2 BUCKETS
                 STORED AS HOLODESK'
EXECUTE IMMEDIATE create_table
  -- 往表 holotest3 中插入数据 (111,12.13)
  insert_table:='INSERT INTO holotest3 VALUES (111,12.13)'
EXECUTE IMMEDIATE insert_table
  -- 将 pid 为 111 的价格更新为 14.15
  update_table:='UPDATE holotest3 SET price=14.15 WHERE pid=111'
EXECUTE IMMEDIATE update_table
  -- 删除表 holotest3 中 pid 为 111 的记录
  delete_table:='DELETE FROM holotest3 WHERE pid=111'
EXECUTE IMMEDIATE delete_table
  -- 往表 holotest3 中插入数据 (112,11.11)
  insert_table:='INSERT INTO holotest3 VALUES (112,11.11)'
EXECUTE IMMEDIATE insert_table
END;
/
```

查看表 holotest3 的数据

```
SELECT * FROM holotest3;
```

pid	price
112	11.11

处理占位符的 DML 语句

PROCEDURE 内的动态 SQL

例 216. 在存储过程内的动态 SQL

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE calc_stats (
w DOUBLE,
x DOUBLE,
y DOUBLE,
z DOUBLE)
IS
BEGIN
DBMS_OUTPUT.PUT_LINE(w + x + y + z)
END;
/

DECLARE
a DOUBLE := 11.11
b DOUBLE := 12.13
c DOUBLE:= 14.15
d DOUBLE:= 15.17
plsql_block STRING
BEGIN
plsql_block := 'BEGIN calc_stats(:w, :x, :y, :z) END'
EXECUTE IMMEDIATE plsql_block USING a, b,c,d
END;
/
```

输出结果为:

output
52.56

INSERT+UPDATE+DELETE

例 217. 使用 EXECUTE IMMEDIATE 语句分别处理占位符的 DML 语句

```
!set plsqlUseSlash true
DECLARE
  -- 分别声明三个字符串类型的变量 insert_table，update_table，delete_table。
  insert_table STRING
  update_table STRING
  delete_table STRING
BEGIN
   -- 将 SQL 语句“往表 holotest2 中插入一条数据，但是字段年龄，绩点的具体值没有给出，而是用占位符代替
”赋值给变量 insert_table。
   insert_table:='insert into holotest2 values (:age, :gpa)'
   -- 使用关键字 EXECUTE IMMEDIATE 执行变量 insert_table 里的 SQL 语句，往表中插入一条年龄为 24，绩点为
3.3 的记录。
    EXECUTE IMMEDIATE insert_table USING 24,3.3
   -- 将 SQL 语句“更新表 holotest2 中年龄为 24 的绩点，gpa 的没有给出具体的值，而是用占位符代替”赋值给变量
update_table。
   update_table:='UPDATE holotest2 SET gpa= :newgpa WHERE age=24'
   -- 执行变量 update_table 的 SQL 语句，将表 holotest2 中年龄为 24 的绩点更新为 4.8。
    EXECUTE IMMEDIATE update_table USING 4.8
   -- 将 SQL 语句“删除表 holotest2 中绩点为某个值的记录”赋值给变量 delete_table。
   delete_table:='delete from holotest2 where gpa= :oldgpa'
   -- 执行变量 delete_table 的 SQL 语句，删除表 holotest2 中绩点为 4.8 的记录。
    EXECUTE IMMEDIATE delete_table USING 4.8
END;
/
```

CREATE+INSERT

例 218. 创建表的同时插入数据

```
!set plsqlUseSlash true
DROP TABLE IF EXISTS holotest4;
DECLARE
 create_table STRING
 insert_table STRING
BEGIN
 create_table:='CREATE table holotest4(sid int, sno int)
                CLUSTERED BY (sid) INTO 2 BUCKETS
                STORED AS HOLODESK'
  EXECUTE IMMEDIATE create_table
    insert_table:='INSERT INTO holotest4 VALUES (:sid, :sno)'
  EXECUTE IMMEDIATE insert_table using 1,101
  EXECUTE IMMEDIATE insert_table using 2,202
END;
/
```

查看新建的表 holotest4 中的数据

```
select * from holotest4;
```

sid	sno
2	202
1	101

CREATE+INSERT+UPDATE

例 219. 创建表的同时插入数据并更新

```
!set plsqlUseSlash true
DROP TABLE IF EXISTS holotest5;
DECLARE
 create_table STRING
 insert_table STRING
 update_table STRING
BEGIN
 create_table:='CREATE table holotest5(wid int, wno int)
                CLUSTERED BY (wid) INTO 2 BUCKETS
                STORED AS HOLODESK'
  EXECUTE IMMEDIATE create_table
    insert_table:='insert into holotest5 values (:wid, :wno)'
  EXECUTE IMMEDIATE insert_table using 3,303
  EXECUTE IMMEDIATE insert_table using 4,404
    update_table:='update holotest5 set wno= :new_wno where wid=4'
  EXECUTE IMMEDIATE update_table using 405
END;
/
```

查看表 holotest5 中的数据

```
SELECT * FROM holotest5;
```

wid	wno
4	405
3	303

CREATE+UPDATE+INSERT+DELETE

例 220. 创建表的同时插入数据更新及删除数据

```
!set plsqlUseSlash true
DECLARE
 create_table STRING
 insert_table STRING
 update_table STRING
 delete_table STRING
BEGIN
 -- 创建 HOLODESK 表 holotest6，字段名分别为 pid，pno，数据类型均为 INT。
 create_table:='CREATE table if not exists holotest6
                (pid int, pno int)
                CLUSTERED BY (pid) INTO 2 BUCKETS
                STORED AS HOLODESK'
  EXECUTE IMMEDIATE create_table
    -- 使用占位符，往表中插入数据。
    insert_table:='insert into holotest6 values (:pid, :pno)'
  -- 占位符的值依次为 (11,123)，(22,246)，(33,369)。
  EXECUTE IMMEDIATE insert_table using 11,123
  EXECUTE IMMEDIATE insert_table using 22,246
  EXECUTE IMMEDIATE insert_table using 33,369
    -- 使用占位符，更新表 holotest6 中 pid 为 11 的 pno。
    update_table:='update holotest6 set pno= :new_pno where pid=11'
  -- 占位符的值为 404。
  EXECUTE IMMEDIATE update_table using 404
   -- 使用占位符，删除表中的数据。
   delete_table:='delete from holotest6 where pid= :oldpid'
  -- 占位符的值为 22。
  EXECUTE IMMEDIATE delete_table using 22
END;
/
```

查看表 holotest6 中的数据

可以看到首先三条数据已被成功插入到表 holotest6 中，其中 pid 为 11 的 pno 已被更新为 404，pid 为 22 的记录已被删除。

```
select * from holotest6;
```

pid	pno
11	404
33	369

需要注意的是，Inceptor 中，不能使用 EXECUTE IMMEDIATE 处理包含 returning 子句的 DML 语句，更多用法说明可参见注意事项章节。

8.11.12.5.2. 使用 OPEN-FOR，FETCH 和 CLOSE 语句

对于处理动态多行的查询操作，可以使用 OPEN-FOR 语句打开游标，使用 FETCH 语句循环提取数据，最终使用 CLOSE 语句关闭游标。

语法

- 定义游标变量

  TYPE cursortype IS REF CURSOR; cursor_variable cursortype;

- 打开游标变量

  OPEN cursor_variable FOR dynamic_string

- 循环提取数据

  FETCH cursor_variable INTO {var1[,var2]…| record_variable}; EXIT WHEN cursor_variable%NOTFOUND

- 关闭游标变量

  CLOSE cursor_variable;

示例

例 221. 使用游标变量执行动态 SQL

```
!set plsqlUseSlash true
set plsql.show.sqlresults=true;
DECLARE
TYPE EmpCurTyp IS REF CURSOR
v_emp_cursor EmpCurTyp
emp_record transactions%ROWTYPE
v_stmt_str string
v_acc_num string
BEGIN
v_stmt_str := 'SELECT * FROM transactions WHERE acc_num = :v_acc_num'
OPEN v_emp_cursor FOR v_stmt_str USING '6513065'
LOOP
FETCH v_emp_cursor INTO emp_record
EXIT WHEN v_emp_cursor%NOTFOUND
END LOOP;
CLOSE v_emp_cursor
END;
/
```

输出结果为:

output
943197522
6513065
20140105100520
b
AA7105670
12.13
200.0
499506900
6513065
20140506133109
s
CA2789982
6.12
100.0

output
289018112
6513065
20140916105811
b
UT7592045
9.81
500.0
895916502
6513065
20141225133500
s
KC9102028
7.49
1100.0
404905188
6513065
20140628133001
b
SH6277444
7.02
100.0
213859826
6513065
20140508094805
b
CL2121979
18.38
700.0

Inceptor 中目前只能通过 EXECUTE IMMEDIATE 和 OPEN-FOR，FETCH 和 CLOSE 语句来实现动态 SQL，不支持使用批量动态 SQL 以及用 DBMS_SQL 包来实现动态 SQL。

8.11.13. Packages

- 一个完整的 Packages 包括包头即声明（specification），和包体（body）两个部分。
- 声明（specification）可以独立存在，也就是说可以仅创建包的声明，不需要再去创建包体（body）。但是不可以在没有包的声明的情况下，直接去创建包体（body）。
- 包头名和包体名必须一致。

8.11.13.1. 创建 Packages

8.11.13.1.1. 语法

创建包头

> 包头中声明数据类型，游标，函数，过程，异常等。 包头中声明的变量可以在包之外引用，是全局变量。 可以在没有程序包主题的情况下存在。 可以重载 （程序包中的多个子程序可以具有相同的名称，形参不同）。 如果子程序的参数仅名称或模式不同，则不能重载。

- 语法

```
CREATE OR REPLACE PACKAGE package_name
IS | AS
  PUBLIC TYPE AND ITEM DECLARATIONS ①
  SUBPROGRAM  SPECIFICATIONS  ②
END;
```

① 包头中直接定义的变量，均为全局变量，可以在包外使用。

② 包头中仅声明过程，函数，RECORD，Collections 等的类型。

创建包体

> 包体中具体定义子程序，类型变量，游标，函数和过程等。 包体中声明的变量，只能在包内使用，是局部变量。 不能在没有包头（specification）声明的情况下独立存在。

- 语法

```
CREATE OR REPLACE  PACKAGE BODY package_name
IS | AS
  PRIVATE TYPE AND ITEM DECLARATIONS  ①
  SUBPROGRAM BODIES  ②
END;
```

① 包体中直接定义的变量均为局部变量，只能在包内使用。

② 包体中具体定义过程，函数，RECORD 类型的变量等。

调用 Packages

- 语法：

```
Package-name.type-name
Package-name.procedure-name
Package-name.function-name
...
```

Package-name 是程序包名称，type-name 是包内声明的类型名称，procedure-name 是包内声明的过程名称，function-name 是包内声明的函数名称。

8.11.13.1.2. 实例

创建包头

例 222. 创建一个名为 test 的包头

```
!set plsqlUseSlash true
-- 创建一个名为 test 的包头。
CREATE OR REPLACE PACKAGE test
IS
   -- 定义一个名为 transid 的全局变量，类型为 string。
   transid STRING
   -- 定义一个名为 l_row 的全局变量，类型为 int。
   l_row int := 1
   -- 声明一个名为 test_procedure 的过程，形参名为 test_name，类型为字符串。
   PROCEDURE test_procedure(test_name string)
   -- 声明一个名为 aa_type 的 NESTED TABLE 类型，用来存放整数型的数据。
   TYPE aa_type IS TABLE OF INTEGER
END;
/
```

创建包体

例 223. 创建一个名为 test 的包体

```
!set plsqlUseSlash true
-- 声明一个名为 test 的包体，此处必须与包头名一致。
CREATE OR REPLACE PACKAGE body test
IS
   -- 具体定义过程 test_procedure 的内容。
   PROCEDURE test_procedure(test_name string) IS
      BEGIN
        dbms_output.put_line(test_name ||'是个好同学！')
       END
END;
/
```

包的调用

例 224. 调用包内类型 aa_type

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个包内 NESTED TABLE 类型 aa_type 的变量 v_aa_type。
   v_aa_type test.aa_type:=test.aa_type()
BEGIN
   -- 给变量 v_aa_type 分配空间。
   v_aa_type.extend(3)
   -- 依次给变量 v_aa_type 赋值。
   v_aa_type(1) := '1001'
   v_aa_type(2) := '2001'
   v_aa_type(3):= '3001'
   -- 依次打印出变量 v_aa_type 里的值。
   FOR i IN 1..v_aa_type.count()
   LOOP
      dbms_output.put_line(v_aa_type(i))
   END LOOP;
   test.test_procedure('张三')
END;
/
```

输出结果为：

output
1001
2001
3001
张三是个好同学！

8.11.13.2. Packages 使用案例

Inceptor 中，在创建 Packages 的时候可以仅创建包头不用再创建包体；但是创建包体的时候，一定要有一个相同名字的包头的存在。

在本章节中，我们主要介绍在仅创建包头的情况下，如何使用 Packages，以及在创建一个完整的 Packages，即包括包头和包体的情况下，怎么使用包。

8.11.13.2.1. 仅创建Packages包头

例 225. 创建一个名为 aa_pkg 的包头，在包内定义一个名为 aa_type 的表类型

```
!set plsqlUseSlash true
-- 创建一个名为 aa_pkg 的包头。
CREATE OR REPLACE PACKAGE aa_pkg IS
   -- 创建包内类型 NESTED TABLE 类型，名为 aa_type。
   TYPE aa_type IS TABLE OF INTEGER
END;
/
```

例 226. 接着创建一个名为 print_aa 的过程，依次打印出表类型变量中的值

```
!set plsqlUseSlash true
-- 创建一个名为 print_aa 的过程，形参名为 aa，类型为 aa_pkg 包内类型 aa_type。
CREATE OR REPLACE PROCEDURE
 print_aa (aa aa_pkg.aa_type) IS
   -- 定义一个整数类型的变量i。
   i INT
BEGIN
   -- 使用集合元素方法 FIRST() 和 NEXT(i)，依次打印出 aa 内的值。
   i := aa.FIRST()
   WHILE i IS NOT NULL LOOP
      DBMS_OUTPUT.PUT_LINE (i ||' '||aa(i))
      i := aa.NEXT(i)
   END LOOP;
END;
/
```

例 227. 创建一个 aa_pkg.aa_type 类型的变量为 aa_var

```
-- 给变量 aa_var 分配大小为 3 的空间。
-- 依次给变量 aa_var 赋值。
-- 利用过程 print_aa 打印出变量的值。
!set plsqlUseSlash true
DECLARE
-- 创建一个类型为 aa_pkg 包内类型 aa_type 的变量 aa_var，并初始化。
   aa_var aa_pkg.aa_type:=aa_pkg.aa_type()
BEGIN
   aa_var.extend(3)
   aa_var(1) := '1001'
   aa_var(2) := '2001'
   aa_var(3):= '3001'
   print_aa(aa_var)
END;
/
```

输出结果为：

依次进行以上操作之后，返回的结果：

output
1 1001
2 2001
3 3001

8.11.13.2.2. 创建Packages包头和包体

例 228. 创建一个名为 testpackage 的包头

```
!set plsqlUseSlash true
-- 创建一个名为 testpackage 的包头。
CREATE OR REPLACE PACKAGE testpackage
IS
   -- 声明一个名为 record_type 的记录类型，分量名分别为 trans_time，price，分量类型分别为
string,double。
   TYPE record_type IS RECORD(trans_time STRING, price DOUBLE)
   -- 声明一个名为 cur 的动态游标。
   TYPE cur IS REF CURSOR
   -- 声明一个名为 testprocedure() 的过程，且不带参数。
   PROCEDURE testprocedure()
END;
/
```

例 229. 创建一个名为 testpackage 的包体

```
!set plsqlUseSlash true
-- 创建一个名为 testpackage 的包体，此处和包头名相同。
CREATE OR REPLACE PACKAGE BODY testpackage
IS
   -- 定义一个类型为 record_type 的变量 v_record_type。
   v_record_type record_type
   -- 定义一个游标类型为 cur 的游标变量 sqlcur。
   sqlcur cur
   -- 具体定义过程 testprocedure() 的内容。
   PROCEDURE testprocedure() IS
BEGIN
   -- 打开游标 sqlcur，去查询表 transactions 中交易号为 943197522 的交易时间和价格。
   OPEN sqlcur FOR
     SELECT trans_time,price
      FROM transactions
      WHERE trans_id=943197522;
   LOOP
      -- 将游标查询到的结果放进变量 v_record_type 里。
      FETCH sqlcur INTO v_record_type
        -- 如果游标没有查询到结果，就退出。
        EXIT WHEN sqlcur%notfound
        -- 输出变量 v_record_type 的分量 trans_time 和分量 price 的值。
        dbms_output.put_line(v_record_type.trans_time||' '||v_record_type.price)
   END LOOP;
   -- 关闭游标。
   CLOSE sqlcur
   END;
END;
/
```

例 230. 调用包内函数

```
!set plsqlUseSlash true
BEGIN
   -- 调用包内过程 testpackage.testprocedure()，此处不可以直接调用 testprocedure()。
   testpackage.testprocedure()
END;
/
```

输出结果为：

output
20140105100520 12.13

8.11.13.3. 系统预定义包

- Inceptor 中有两个系统预定义的包，分别为 dbms_output 和 owa_util。
- 我们可以使用相关命令来查看系统预定义包的有关信息，具体内容可参见注意事项章节。
- 系统预定义包的具体使用方法和案例，可参见预定义函数/过程/包章节。

8.11.13.4. 预定义函数/过程/包

在 Inceptor 中有一些系统预定义的函数/过程/包，可以用来直接调用，此外，我们可以通过相关命令随时查看系统内预定义的函数/过程/包的有关信息，具体查看方法可参见注意事项章节。
Inceptor 中一共有十个系统预定义的函数/过程，如下所示，我们将分章节介绍每一个函数/过程的具体内容及使用方法。

- set_env(string,string)
- get_env(string)
- sqlcode(void)
- sqlerrm(void)
- sqlerrm(int)
- get_columns(string,nestedtable<string>)
- put_line(string)
- raise_application_error(int,string,bool)
- dbms_output
- owa_util

8.11.13.4.1. 语法

- **set_env(string,string)**

  Inceptor 中，set_env 是一个过程，形参 enVar 的参数类型为 IN，数据类型为字符串，用来存放环境变量的名称；形参 value 的参数类型为 IN，数据类型为字符串，相应地用来存放环境变量的值。

  ```
  PROCEDURE set_env(enVar IN STRING, value IN STRING)
  ```

- **get_env(string)**

  Inceptor 中，get_env 是一个函数，形参 enVar 的参数类型为 IN，数据类型为字符串，该函数用来返回 set_env 中环境变量的所对应的值。

  ```
  FUNCTION get_env(enVar IN STRING) RETURN STRING
  ```

- **sqlcode(void)**

  Inceptor 中，sqlcode() 是一个不带参数的函数，用来返回当异常发生时，当前异常的 Error code。

  ```
  FUNCTION sqlcode() RETURN INT
  ```

- **sqlerrm(void)**

  Inceptor 中，不带参数的 sqlerrm()，用来返回当异常发生时，当前异常的 Error message。

  ```
  FUNCTION sqlerrm() RETURN STRING
  ```

- **sqlerrm(int)**

  Inceptor中，带参数的 sqlerrm()，用来返回既定 Error code 下的 Error message。

  ```
  FUNCTION sqlerrm(errCode IN INT) RETURN STRING
  ```

- **get_columns(string,nestedtable<string>)**

Inceptor 中，get_columns 可以作函数使用，返回数据表中的列名。

```
FUNCTION get_columns(table IN STRING, columns IN  ) RETURN
```

Inceptor 中，get_columns 也可以作过程使用，返回数据表的列名。其中columns参数表示排除的字段，即哪些字段不在结果中显示。

```
PROCEDURE default.get_columns_test()
```

- **put_line**

Inceptor,put_line 是一个过程，形参 msg 的参数类型为 IN，数据类型为字符串，该过程用来打印出字符串的值。

```
PROCEDURE dbms_output.put_line(msg IN STRING)
```

- **raise_application_error(int,string,bool)**

在Inceptor PL/SQL 中，可以使用预定义函数 raise_application_error，来抛出带有指定 error code, error message 的异常，目前第三个参数 keepExistError 是可选的，且没有任何作用。

```
FUNCTION raise_application_error(errorCode IN INT, msg IN STRING, keepExistError IN BOOL) RETURN
EXCEPTION
```

> 需要注意的是，Inceptor 中仅支持部分系统预定义的异常，也就是说对于 sqlcode，sqlerrm 函数，并非每一种异常发生，都会返回相对应的 Error code和Error message。 Inceptor 对于系统预定义异常的支持情况，可仔细阅读异常章节，获取更多信息。

- **dbms_output**

dbms_output 包内有一个过程，该过程可以实现将字符（包括变量和常量）的值打印出来，我们通常调用包内过程，来打印变量或者常量的值。

```
PACKAGE dbms_output IS
   PROCEDURE dbms_output.put_line(msg IN STRING)
PACKAGE BODY dbms_output IS
   PROCEDURE dbms_output.put_line(msg IN STRING)
```

- **owa_util**

在 Inceptor 中，系统预定义包 owa_util，仅支持包内过程 who_called_me，有四个形参，参数类型均为 OUT 类型，其中 owner 是存储过程的创建者信息，name 是存储过程调用者的名字信息，lineno 是调用语句在调用者中的行号，type 通常表明程序的类型，包括 ANONYMOUS_BLOCK, FUNCTION, PROCEDURE, PACKAGE BODY，每次调用包内过程 who_called_me 时，每一个形参都会获得一个具体的值，即实参。

```
PACKAGE owa_util IS
   PROCEDURE owa_util.who_called_me(owner OUT STRING, name OUT STRING, lineno OUT INT, type OUT
STRING)
PACKAGE BODY owa_util IS
   PROCEDURE owa_util.who_called_me(owner OUT STRING, name OUT STRING, lineno OUT INT, type OUT
STRING)
```

8.11.13.4.2. 案例

set_env 与 get_env

例 231. set_env 与 get_env 的使用

```
!set plsqlUseSlash true
DECLARE
  a STRING
  b STRING
BEGIN
  -- 调用过程 set_env，定义一个名为 aa 的环境变量，值为 hello。
  set_env('aa','hello')
  -- 调用函数 get_env，获取环境变量 aa 的值，并赋值给 a。
  a := get_env('aa')
  -- 输出变量 a 的值，可以发现变量 a 与环境变量 aa 的值相同。
  DBMS_OUTPUT.PUT_LINE('the value of a is: '||a)
  set_env('bb','world')
  b:= get_env('bb')
  DBMS_OUTPUT.PUT_LINE('the value of a is: '||b)
END;
/
```

输出结果为：

output
the value of a is: hello
the value of a is: world

sqlcode 与 sqlerrm

例 232. sqlcode 与 sqlerrm 的使用

```
!set plsqlUseSlash true
DECLARE
   test_record transactions%rowtype
   v_code INT
   v_errm STRING
BEGIN
    SELECT *
    INTO test_record
    FROM transactions
    WHERE acc_num=6513065;
    dbms_output.put_line(test_record.trans_time)
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
    -- 当异常发生时，获取当前异常的 Error code，并赋值给变量 v_code。
    v_code := sqlcode()
    -- 当异常发生时，获取当前异常的 Error message，并赋值给变量 v_errm。
    v_errm := sqlerrm()
    dbms_output.put_line('error code is: ' ||v_code)
    dbms_output.put_line('error message is: ' ||v_errm)
END;
/
```

输出结果为：

output
error code is: -1422
error message is: TOO_MANY_ROWS

get_columns

例 233. get_columns 的使用

```
-- 最终输出的字符串 s 的值，与最开始的值相比，少了最后一个列名之后的逗号。
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE get_columns_test()
IS
  TYPE nest_table is TABLE of STRING
  TYPE nest_table_1 is TABLE of STRING
  -- 声明一个已定义好的 nest_table 类型的变量 a_table。
  a_table nest_table
  -- 声明一个已定义好的 nest_table1 类型的变量 b_table，默认为 null 值。
  b_table nest_table_1 default null
  s string
BEGIN
  -- 给变量 a_table 赋值。
  a_table := nest_table('this','is','a','table','ID')
  -- 使用 get_columns 函数，获取表 transactions 的列名,并赋值给变量 b_table。
  b_table := get_columns("transactions")
  DBMS_OUTPUT.PUT_LINE('a_table:')
  -- 依次打印出变量 a_table 的值。
  FOR i in a_table.first() .. a_table.last() LOOP
     DBMS_OUTPUT.PUT_LINE(a_table(i))
  END LOOP;
  IF b_table is not null THEN
    -- 将变量 b_table 的值用逗号连接起来，并赋值给变量 s。
    FOR i in b_table.first() .. b_table.last() LOOP
     s := s || b_table(i) || ", "
     END LOOP;
  END;
  DBMS_OUTPUT.PUT_LINE('b_table:')
  -- instr 函数返回字符串 s 中从右往左数第一次出现逗号的位置，设为 m，substr 函数是从字符串 s
中第一个字符开始截取长度为 m-1 的字符串。
  -- 用于去除最后一个逗号“,”
  s := substr(s,1,oracle_instr(s,',',-1)-1)
  DBMS_OUTPUT.PUT_LINE(s)
END;
/

BEGIN
  get_columns_test()
END;
/
```

输出结果为:

output
a_table:
this
is
a
table
ID
b_table:
trans_id, acc_num, trans_time, trans_type, stock_id, price, amount

raise_application_error

例 234. raise_application_error 的使用

```
!set plsqlUseSlash true
DECLARE
  num_tables INT
BEGIN
  -- 查询表 transactions 的行数，并赋值给变量 num_tables。
  SELECT COUNT(*) INTO num_tables FROM transactions;
   IF num_tables < 1000 THEN
     -- 如果表 transactions 的行数小于 1000，则抛出异常。
     RAISE_APPLICATION_ERROR
      (-20101, 'Expecting at least 1000 tables')
    ELSE
      -- 否则，就输出表 transactions 的行数。
      dbms_output.put_line(num_tables)
    END IF
END;
/
```

输出结果为：

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -20101] Expecting at least 1000
tables
**********************************************
System function raise_application_error (LINE -1, COLUMN -1, TEXT \"null\")
ANONYMOUS BLOCK (LINE 8, COLUMN 0, TEXT \"RAISE_APPLICATION_ERROR
(-20101, 'Expecting at least 1000 tables')\")
**********************************************
```

例 235. 上述例子不抛出异常的情况

```
!set plsqlUseSlash true
DECLARE
  num_tables INT
BEGIN
  -- 查询表 transactions 的行数，并赋值给变量 num_tables。
  SELECT COUNT(*) INTO num_tables FROM transactions;
   IF num_tables < 10  THEN
     -- 如果表 transactions 的行数小于 10，则抛出异常。
     RAISE_APPLICATION_ERROR
      (-20101, 'Expecting at least 1000 tables')
    ELSE
      -- 否则，就输出表 transactions 的行数。
      dbms_output.put_line(num_tables)
    END IF
END;
/
```

输出结果为：

output
20

PUT_LINE

- 在 Inceptor 中使用 PUT_LINE 函数时，如果程序正常运行，则 PUT_LINE 会统一打印到终端。
- 如果程序运行过程中出现了未被处理的异常，可以设置相关命令，将 PUT_LINE 正常打印出的内容连同异常栈一起打印到终端。
- 更多用法说明，可参见注意事项章节。

例 236. 程序正常运行时，PUT_LINE 的打印

```
!set plsqlUseSlash true
-- 创建名为 put_line_test 的函数，形参 v1 的数据类型为整数型，返回值为双精度类型。
CREATE OR REPLACE FUNCTION put_line_test(v1 int)
   RETURN DOUBLE
IS
  v2 DOUBLE
  v3 STRING
BEGIN
   -- 使用 put_line 函数，打印出 null 值。
   put_line(null)
   DBMS_OUTPUT.PUT_LINE(v1)
   v2 := 2
   v3 := "I'm a string."
   put_line(v2)
   -- 使用 put_line 函数，打印出变量 v2 的值。
   DBMS_OUTPUT.PUT_LINE('v2: ' || v2)
   -- 使用 put_line 函数，打印出字符串 v2+v1。
   put_line('v2 + v1')
   DBMS_OUTPUT.PUT_LINE(v2 + v1)
   -- 使用 put_line 函数,打印出 v3 的值。
   put_line(v3)
   v3 := null
   DBMS_OUTPUT.PUT_LINE(v3)
   -- 函数返回值为 v1 和 v2 的乘积。
   return v2 * v1
END;
/

BEGIN
  dbms_output.put_line("Executing put_line_test(1)")
  -- 使用 put_line 函数，打印出 put_line_test(1) 的全部值。
  put_line(put_line_test(1))
END;
/
```

输出结果：

output
Executing put_line_test(1)
null
1
2.0
v2: 2.0
v2 + v1
3.0
I’ m a string.
null
2.0

dbms_output

- Inceptor 中，包 dbms_output 是系统预定义的，包内只有一个名为 put_line 的过程，且 put_line 过程也是系统预定义的。
- 在实际的使用中，可以直接调用过程 put_line 打印结果，也可以调用包内过程 put_line 来打印结果。

例 237. 使用包内过程，打印结果

```
!set plsqlUseSlash true
BEGIN
FOR test IN 1..5 LOOP
dbms_output.put_line('test:'||test)
END LOOP;
END;
/
```

① 调用包内过程 put_line。

输出结果为:

output
test:1
test:2
test:3
test:4
test:5

owa_util

例 238. 创建过程 outer_proc()

```
!set plsqlUseSlash true
-- 创建一个名为 outer_proc() 的过程。
CREATE OR REPLACE PROCEDURE outer_proc()
IS
-- 分别声明四个变量。
v_owner STRING
v_name STRING
v_lineno INT
v_type STRING
BEGIN
  DBMS_OUTPUT.PUT_LINE('Outer outer proc')
  -- 调用包内过程 owa_util.who_called_me，其中 v_owner, v_name, v_lineno, v_type
分别用来储存相对应形参所获得的值。
  owa_util.who_called_me(v_owner, v_name, v_lineno, v_type)
  -- 依次输出四个变量的值。
  DBMS_OUTPUT.PUT_LINE('Owner: ' || v_owner)
  DBMS_OUTPUT.PUT_LINE('Name: ' || v_name)
  DBMS_OUTPUT.PUT_LINE('Lineno: ' || v_lineno)
  DBMS_OUTPUT.PUT_LINE('Type: ' || v_type)
END;
/
BEGIN
  outer_proc()
END;
/
```

输出结果为：

output
Outer outer proc
Owner:
Name:
Lineno: 2
Type: ANONYMOUS_BLOCK

8.11.14. 异常

目前 Inceptor 中，PL/SQL 中的异常包括标准 PL/SQL 异常和 Hive 异常，其中 PL/SQL 异常包括系统预定义异常和用户自定义异常，Hive 异常即在不满足 Inceptor 数据库语法规范情况下发生的异常。

默认情况下，Hive 异常不被 PL/SQL 处理（即使有 WHEN OTHERS 语句也不会处理 Hive 异常），如果需要在 PL/SQL 中捕获 Hive 异常，需要设置变量 plsql.catch.hive.exception为true。

此时 Hive 异常能被 HIVE_EXCEPTION 与 OTHERS 捕获，更多用法说明，可参见注意事项章节。

8.11.14.1. 支持的系统预定义异常

ORCALE 一共有 21 种系统预定义的异常，Inceptor 支持其中的7种异常，也就是说在 PL/SQL 的语句中可以直接引用这7种异常的情况名，无需用户自定义异常，具体用法可参考以下章节的内容。

8.11.14.1.1. NO_DATA_FOUND

使用 select into 未返回行，或应用索引表未初始化的元素时。

例 239. 查询 transactions 表中账号为 6513064 的交易时间

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个 string 类型的 test。
   test string
BEGIN
   -- 将账号为 6513065 的交易时间的信息放进 test。
   SELECT trans_time
     INTO test
     FROM transactions
     WHERE acc_num=6513064;
   -- 输出 test 的值。
   dbms_output.put_line(test)
END;
/
```

可以看到 Inceptor 会抛出一个 NO_DATA_FOUND 的异常。

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error 100] NO_DATA_FOUND
**********************************************
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT \"SELECT trans_time
INTO test
FROM transactions
WHERE acc_num=6513064;\")
**********************************************
```

例 240. 在查询的过程中直接引用一个名为 NO_DATA_FOUND 的 EXCEPTION

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个 string 类型的 test1。
   test1 string
BEGIN
   -- 将账号为 6513065 的交易时间的信息放进 test1。
   SELECT trans_time
     INTO test1
     FROM transactions
     WHERE acc_num=6513064;
   -- 输出 test1 的值。
   dbms_output.put_line(test1)
EXCEPTION
   -- 引用一个 NO_DATA_FOUND 的 EXCEPTION。
   WHEN NO_DATA_FOUND THEN
      -- 当异常发生时，所输出的信息。
      dbms_output.put_line('no values')
END;
/
```

当异常发生时，返回的值，也就是说 Inceptor 支持 NO_DATA_FOUND 这一 ORACLE 系统预定义的异常。

output
no values

8.11.14.1.2. TOO_MANY_ROWS

执行 select into 时，结果集超过一行。

例 241. 定义一个名为 test_record 记录变量，来查询 transactions 表中账号为 6513065 的交易时间

```
!set plsqlUseSlash true
DECLARE
   -- 基于 transactions 表定义一个名为 test_record 的记录变量。
   test_record transactions%rowtype
BEGIN
   -- 将账号为 6513065 的全部信息放进 test_record 记录变量。
   SELECT *
    INTO test_record
    FROM transactions
    WHERE acc_num=6513065;
   -- 输出 test_record 的交易时间。
   dbms_output.put_line(test_record.trans_time)
END;
/
```

可以看到 Inceptor 会抛出一个名为 TOO_MANY_ROWS 的异常。

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -1422] TOO_MANY_ROWS
**********************************************
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT \"SELECT *
INTO test_record
FROM transactions
WHERE acc_num=6513065;\")
**********************************************
```

例 242. 在上面的查询中增加一个名为 TOO_MANY_ROWS的EXCEPTION

```
!set plsqlUseSlash true
DECLARE
   -- 基于 transactions 表定义一个名为 test_record 的记录变量。
   test_record transactions%rowtype
BEGIN
   -- 将账号为 6513065 的全部信息放进 test_record 记录变量。
   SELECT *
    INTO test_record
    FROM transactions
    WHERE acc_num=6513065;
   -- 输出 test_record 的交易时间。
   dbms_output.put_line(test_record.trans_time)
EXCEPTION
   -- 引用一个名为 TOO_MANY_ROWS 的 EXCEPTION。
   WHEN TOO_MANY_ROWS THEN
      -- 当异常发生时，所输出的信息。
      dbms_output.put_line('too many rows')
END;
/
```

当异常发生时返回的值，也就是说Inceptor支持 TOO_MANY_ROWS 这一 ORACLE 系统预定义的异常。

output
too many rows

8.11.14.1.3. CURSOR_ALREADY_OPEN

游标已经打开，即打开一个已经打开了的游标的时候会抛出的异常。

例 243. 游标的正确使用方法

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个游标，查询 transactions 表中账号为 6513065 的交易号，交易时间，价格和金额的信息。
   CURSOR cur_tran IS
      SELECT trans_id,trans_time,price,amount
      from transactions WHERE acc_num=6513065;
   -- 分别定义四个 %type 类型的变量。
   v_trans_id transactions.trans_id%TYPE
   v_trans_time transactions.trans_time%TYPE
   v_price transactions.price%TYPE
   v_amount transactions.amount%TYPE
BEGIN
   -- 打开游标。
   OPEN cur_tran
      -- 将游标查询的结果依次放入变量中。
      LOOP
         FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount
           -- 如果没有找到游标就退出。
           EXIT WHEN cur_tran%notfound
           -- 输出查询结果中的交易号。
           dbms_output.put_line(v_trans_id)
      END LOOP;
   -- 关闭游标。
   CLOSE cur_tran
END;
/
```

正确的输出结果为：

output
943197522
499506900
289018112
895916502
404905188
213859826

例 244. 打开一个已经打开了的游标

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个游标，查询 transactions 表中账号为 6513065 的交易号，交易时间，价格和金额的信息。
   CURSOR cur_tran IS
      SELECT trans_id,trans_time,price,amount
       FROM transactions WHERE acc_num=6513065;
   -- 分别定义四个 %type 类型的变量。
   v_trans_id transactions.trans_id%TYPE
   v_trans_time transactions.trans_time%TYPE
   v_price transactions.price%TYPE
   v_amount transactions.amount%TYPE
BEGIN
   -- 第一次打开游标。
   OPEN cur_tran
      LOOP
         -- 将游标查询的结果依次放入变量中。
         FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount
         -- 如果游标没有返回就退出。
         EXIT WHEN cur_tran%notfound
      END LOOP;
   -- 第二次打开游标。
   OPEN cur_tran
END;
/
```

可以看出，Inceptor 会抛出一个 CURSOR_ALREADY_OPEN 的异常

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -6511] CURSOR_ALREADY_OPEN
**********************************************
ANONYMOUS BLOCK (LINE 21, COLUMN 5, TEXT \"OPEN cur_tran\")
**********************************************
```

Inceptor 中，如果程序运行过程中发生了没有处理的异常，则 Inceptor 默认会打印出异常栈到终端，方便快速找出异常发生的原因和位置，欲了解更多关于 Debug info 的信息，可参见注意事项章节。

例 245. 打开一个已经打开了的游标，并增加一个 CURSOR_ALREADY_OPEN 的 EXCEPTION

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个游标，查询 transactions 表中账号为 6513065 的交易号，交易时间，价格和金额的信息。
   CURSOR cur_tran IS
     SELECT trans_id,trans_time,price,amount
      FROM transactions WHERE acc_num=6513065;
   -- 分别定义四个 %type 类型的变量。
   v_trans_id transactions.trans_id%TYPE
   v_trans_time transactions.trans_time%TYPE
   v_price transactions.price%TYPE
   v_amount transactions.amount%TYPE
BEGIN
   -- 第一次打开游标。
   OPEN cur_tran
      -- 将游标查询的结果依次放入变量中。
      LOOP
         FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount
          -- 如果游标没有返回就退出。
          EXIT WHEN cur_tran%notfound
      END LOOP;
   -- 第二次打开游标。
   OPEN cur_tran
EXCEPTION
   -- 引用一个 CURSOR_ALREADY_OPEN 的 EXCEPTION。
   WHEN CURSOR_ALREADY_OPEN THEN
   -- 当异常发生时，所输出的信息。
   dbms_output.put_line('the cursor is already open')
END;
/
```

当异常抛出时的输出结果，也就是说 Inceptor 支持 CURSOR_ALREADY_OPEN 异常。

output
the cursor is already open

8.11.14.1.4. ROWTYPE_MISMATCH

宿主游标变量与 PL/SQL 游标变量的返回类型不兼容，也就是说当一个打开的主游标变量传递到一个存储子程序时，实参的返回类型和形参的必须一致。
其中宿主游标变量，是我们所定义的游标类型；PL/SQL 游标变量是基于游标定义的记录变量，即 cursor%rowtype。

例 246. 声明一个游标，查询 transactions 表中账号为 6513065 的全部信息，并返回交易号的信息

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个名为 cur_tran 的游标，查询表中账号为 6513065 的全部信息，即一共 7 条信息。
   CURSOR cur_tran IS SELECT * from transactions WHERE acc_num=6513065;
   -- 分别定义四个 %type 类型的变量。
   v_trans_id transactions.trans_id%TYPE
   v_trans_time transactions.trans_time%TYPE
   v_price transactions.price%TYPE
   v_amount transactions.amount%TYPE
BEGIN
   -- 第一次打开游标。
   OPEN cur_tran
      -- 将游标查询的结果依次放入 4 个变量中。
      LOOP
         FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount
         -- 如果游标没有返回就退出。
         EXIT WHEN cur_tran%notfound
         -- 输出查询结果中的交易号信息。
         dbms_output.put_line(v_trans_id)
      END LOOP;
   -- 关闭游标。
   CLOSE cur_tran
END;
/
```

可以看到，由于定义的游标包含 7 个变量，将游标查询的结果放进 4 个变量内，显然会存在数据类型不匹配的现象，Inceptor 会抛出 ROWTYPE_MISMATCH 的异常。

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -6504] ROWTYPE_MISMATCH
**********************************************
ANONYMOUS BLOCK (LINE 14, COLUMN 6, TEXT \"FETCH cur_tran INTO
v_trans_id,v_trans_time,v_price,v_amount\")
**********************************************
```

例 247. 声明一个游标，查询 transactions 表中账号为 6513065 的全部信息，返回交易号的信息，并增加一个 ROWTYPE_MISMATCH 的异常

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个名为 cur_tran 的游标，查询表中账号为 6513065 的全部信息，即一共 7 条信息。
   CURSOR cur_tran IS SELECT * from transactions WHERE acc_num=6513065;
   -- 分别定义四个 %type 类型的变量。
   v_trans_id transactions.trans_id%TYPE
   v_trans_time transactions.trans_time%TYPE
   v_price transactions.price%TYPE
   v_amount transactions.amount%TYPE
BEGIN
   -- 第一次打开游标。
   OPEN cur_tran
      -- 将游标查询的结果依次放入 4 个变量中。
      LOOP
         FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount
         -- 如果没有找到游标就退出。
         EXIT WHEN cur_tran%notfound
         -- 输出查询结果中的交易号信息。
         dbms_output.put_line(v_trans_id)
      END LOOP;
   -- 关闭游标。
   CLOSE cur_tran
EXCEPTION
   -- 引用一个 ROWTYPE_MISMATCH 的 EXCEPTION。
   WHEN ROWTYPE_MISMATCH THEN
   -- 当异常发生时输出的信息。
   dbms_output.put_line('Row type mismatch, fetching transactions data ...')
END;
/
```

当异常发生时，所返回的值，即 Inceptor 支持 ROWTYPE_MISMATCH 的异常。

output
Row type mismatch, fetching transactions data …

8.11.14.1.5. SUBSCRIPT_BEYOND_COUNT

SUBSCRIPT_BEYOND_COUNT 是 NESTED TABLE 或者 VARRAY 的空间小于使用的下标的时候会报出的异常。

例 248. 声明一个 VARRAY 类型，最大容量为 5 的变量

```
!set plsqlUseSlash true
DECLARE
  -- 声明一个最大容量为 20，元素数据类型为 DOUBLE 的 VARRAY 类型 test_array。
  TYPE test_array IS VARRAY(20)  OF DOUBLE
   -- 声明一个名为 v_test，类型为 test_array 的 VARRAY 变量。
   v_test test_array
BEGIN
   -- v_test 变量中仅有一个值为 123456 的元素。
   v_test:=test_array(123456)
   -- 输出 v_test 下标为 2 的值。
   dbms_output.put_line('v_test(1):'||v_test(2))
END;
/
```

可以看到，Inceptor 会抛出 Subscript beyond count 的异常。

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -6533] Subscript beyond count
**********************************************
ANONYMOUS BLOCK (LINE 10, COLUMN 35, TEXT \"v_test(2)\")
**********************************************
```

例 249. 声明一个 VARRAY 类型，最大容量为 20 的变量，并增加一个 SUBSCRIPT_BEYOND_COUNT 的 EXCEPTION

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个最大容量为 5，元素数据类型为 DOUBLE 的 VARRAY 类型 test_array。
   TYPE test_array IS VARRAY(5) OF DOUBLE
   -- 声明一个名为 v_test，类型为 test_array 的 VARRAY 变量。
   v_test test_array
BEGIN
   -- v_test 变量中仅有一个值为 123456 的元素。
   v_test:=test_array(123456)
   -- 输出 v_test 下标为 6 的值。
   dbms_output.put_line('v_test(1):'||v_test(6))
EXCEPTION
   -- 引用一个 SUBSCRIPT_BEYOND_COUNT 的异常。
   WHEN SUBSCRIPT_BEYOND_COUNT THEN
   -- 异常发生时，所输出的值。
   dbms_output.put_line('subscript out of range')
END;
/
```

当异常发生时返回的值，可见 Inceptor 支持 SUBSCRIPT_BEYOND_COUNT 异常。

output
subscript out of range

8.11.14.1.6. SUBSCRIPT_OUTSIDE_LIMIT

使用嵌套表或 VARRAY 时，如果下标指定为负数，会报出该异常。

例 250. 声明一个 VARRAY 类型，最大值为 20，元素下标为 -1

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个最大容量为 20，元素数据类型为 DOUBLE 的 VARRAY 类型 test_array。
   TYPE test_array IS VARRAY(20)  OF DOUBLE
   -- 声明一个名为 v_test，类型为 test_array 的 VARRAY 变量。
   v_test test_array
BEGIN
   -- v_test 变量中仅有一个值为 123456 的元素。
   v_test:=test_array(123456)
   -- 输出 v_test 下标为 -1 的值。
   dbms_output.put_line('v_test(-1):'||v_test(-1))
END;
/
```

可以看到，Inceptor 会抛出一个 Subscript outside of limit 的异常。

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -6532] Subscript outside of limit
**********************************************
ANONYMOUS BLOCK (LINE 10, COLUMN 36, TEXT \"v_test(-1)\")
**********************************************
```

例 251. 声明一个 VARRAY 类型，最大值为 20，元素下标为 -1，并增加一个 SUBSCRIPT_OUTSIDE_LIMIT 的 EXCEPTION

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个最大容量为 20，元素数据类型为 DOUBLE 的 VARRAY 类型 test_array。
   TYPE test_array IS VARRAY(20) OF DOUBLE
   -- 声明一个名为 v_test，类型为 test_array 的 VARRAY 变量。
   v_test test_array
BEGIN
   -- v_test 变量中仅有一个值为 123456 的元素。
   v_test:=test_array(123456)
   -- 输出 v_test 下标为 -1 的值。
   dbms_output.put_line('v_test(-1):'||v_test(-1))
EXCEPTION
   -- 引用一个 SUBSCRIPT_OUTSIDE_LIMIT 的异常。
   WHEN SUBSCRIPT_OUTSIDE_LIMIT THEN
   -- 异常发生时，所输出的值。
   dbms_output.put_line('please check the subscript')
END;
/
```

当异常发生时，所返回的值，可见 Inceptor 支持 SUBSCRIPT_OUTSIDE_LIMIT 异常。

output
please check the subscript

8.11.14.1.7. COLLECTION_IS_NULL

集合元素未初始化，即当没有初始化集合元素的时候，会抛出的异常。

例 252. 声明一个集合类型，并没有赋予其初始的值

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个 table 类型。
   TYPE emp_ssn_array IS TABLE OF DOUBLE
   -- 声明一个名为 best_acc_num 的 table 变量。
   best_acc_num emp_ssn_array
BEGIN
   -- 直接给 best_acc_num(0) 赋值。
   best_acc_num(0):='6513065'
   -- 输出 best_acc_num(0) 的值。
   dbms_output.put_line('best_acc_num(0):'||best_acc_num(0))
END;
/
```

可以看到 Inceptor 会抛出没有初始化集合元素的异常。

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -6531] Reference to uninitialized
collection
**********************************************
ANONYMOUS BLOCK (LINE 8, COLUMN 0, TEXT \"best_acc_num(0)\")
**********************************************
```

例 253. 声明一个集合类型，没有赋予其初始的值，增加一个 collection_is_null 的异常情况名

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个 table 类型。
   TYPE emp_ssn_array IS TABLE OF DOUBLE
   -- 声明一个名为 best_acc_num 的 table 变量。
   best_acc_num emp_ssn_array
BEGIN
   -- 直接给 best_acc_num(0) 赋值。
   best_acc_num(0):='6513065'
   -- 输出 best_acc_num(0) 的值。
   dbms_output.put_line('best_acc_num(0):'||best_acc_num(0))
EXCEPTION
   -- 引用一个 collection_is_null 的异常。
   WHEN collection_is_null THEN
     -- 异常发生时，所输出的值。
     dbms_output.put_line('please initialize the collection element')
END;
/
```

当异常发生时返回的值，可以看到 Inceptor 支持 collection_is_null 异常。

output
please initialize the collection element

8.11.14.1.8. INVALID_CURSOR

在不合法的游标上进行操作，即关闭一个尚未打开的游标所抛出的异常。

例 254. 声明一个游标，打开游标，返回表中账号为 6513065 的全部交易时间，再关闭游标，即再次关闭已经关闭了的游标

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个名为 cur_tran 的游标，用来查询表 transactions 中账号为 6513065
的交易号，交易时间，价格和金额。
   CURSOR cur_tran IS SELECT trans_id,trans_time,price,amount from transactions WHERE acc_num
=6513065;
   v_trans_id transactions.trans_id%TYPE
   v_trans_time transactions.trans_time%TYPE
   v_price transactions.price%TYPE
   v_amount transactions.amount%TYPE
BEGIN
   -- 打开游标。
   OPEN cur_tran
      -- 输出游标查询结果中的交易时间。
      LOOP
         FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount
          EXIT WHEN cur_tran%notfound
         dbms_output.put_line(v_trans_time)
      END LOOP;
   -- 关闭游标。
   CLOSE cur_tran
   -- 再次关闭已经关闭了的游标，即关闭一个尚未打开的游标，此处会抛出一个 INVALID_CURSOR 的异常。
   CLOSE cur_tran
END;
/
```

输出结果为：

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -1001] INVALID_CURSOR
**********************************************
ANONYMOUS BLOCK (LINE 20, COLUMN 6, TEXT \"CLOSE cur_tran\")
**********************************************
```

例 255. 声明一个游标，不打开游标，直接返回表中账号为 6513065 的全部交易时间，再关闭游标，并且增加一个 INVALID_CURSOR 的异常

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个名为 cur_tran 的游标，用来查询表 transactions 中账号为 6513065
的交易号，交易时间，价格和金额。
   CURSOR cur_tran IS SELECT trans_id,trans_time,price,amount from transactions WHERE acc_num
=6513065;
   v_trans_id transactions.trans_id%TYPE
   v_trans_time transactions.trans_time%TYPE
   v_price transactions.price%TYPE
   v_amount transactions.amount%TYPE
BEGIN
   -- 打开游标。
   OPEN cur_tran
      -- 输出游标查询结果中的交易时间。
      LOOP
         FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount
          EXIT WHEN cur_tran%notfound
         dbms_output.put_line(v_trans_time)
      END LOOP;
   -- 关闭游标。
   CLOSE cur_tran
   -- 再次关闭已经关闭了的游标，即关闭一个尚未打开的游标，此处会抛出一个 INVALID_CURSOR 的异常。
   CLOSE cur_tran
EXCEPTION
   -- 增加一个 EXCEPTION 的部分，如果程序运行过程中出出现了 INVALID_CURSOR 的异常，则会打印出 THE CURSOR
IS INVALID 的信息。
   WHEN INVALID_CURSOR THEN
     dbms_output.put_line('THE CURSOR IS INVALID')
END;
/
```

output
20140105100520
20140506133109
20140916105811
20141225133500
20140628133001
20140508094805
THE CURSOR IS INVALID

此例中，由于在第二次关闭游标之前，程序都是正常运行的，也就是说，之前游标的查询，以及结果的返回都可以正常的打印出来，但是第二次关闭游标的时候，程序会出错，此时 INVALID_CURSOR 会处理这一异常，并返回当异常发生时，所返回的值。

8.11.14.2. 暂不支持的系统预定义异常

ORACLE 数据库中一共定义了 21 中常见的系统预定义异常，在创建 PL/SQL 时，如果不符合 ORACLE 的一些语法规范，系统就会报出相对应的异常。
此外在创建 PL/SQL 时，也可以直接引用系统预定义的异常情况名，而不需要用户自己去声明该异常情况名。处理这种异常情况时，只需在 PL/SQL 块的异常处理部分，直接引用相应的异常情况名，并对其完成相应的异常错误处理即可。
但在 ORACLE 预定义的 21 种异常中，Inceptor 不支持其中的 13 种系统预定义异常，也就是说当这些异常存在时，Inceptor 并不会报错，进一步地，在 Inceptor 中创建 PL/SQL 语句时，直接引用该异常情况名时，Inceptor 也不会返回任何结果。

8.11.14.2.1. INVALID_NUMBER

ORACLE 中，内嵌的 SQL 语句不能将字符转换为数字，否则会报出 INVALID_NUMBER 异常，但在 Inceptor 中，并不支持这一异常。

- 查看表 transactions 的结构，可以看到 acc_num 字段的类型为 string

```
DESC transactions;
```

col_name	data_type	default_value	not_null	unique	comment
trans_id	string	NULL	No	No	
acc_num	string	NULL	No	No	
trans_time	string	NULL	No	No	
trans_type	string	NULL	No	No	
stock_id	string	NULL	No	No	
price	double	NULL	No	No	
amount	double	NULL	No	No	

- 查看 transactions 表中 acc_num 为 ‘6513065’ 的全部信息，可以看到 Inceptor 并不会报错。

```
SELECT * FROM transactions WHERE acc_num=6513065;
```

trans_id	acc_num	trans_time	trans_type	stock_id	price	amount
943197522	6513065	20140105100520	b	AA7105670	12.13	200.0
499506900	6513065	20140506133109	s	CA2789982	6.12	100.0
289018112	6513065	20140916105811	b	UT7592045	9.81	500.0
895916502	6513065	20141225133500	s	KC9102028	7.49	1100.0
404905188	6513065	20140628133001	b	SH6277444	7.02	100.0
213859826	6513065	20140508094805	b	CL2121979	18.38	700.0

也就是说，Inceptor 并不支持这种 INVALID_NUMBER 异常。

8.11.14.2.2. VALUE_ERROR

ORACLE中，赋值时，如果变量长度不足以容纳实际数据，会报出 VALUE_ERROR 的异常，但在 Inceptor 中，并不支持这一异常。

例 256. 声明一个变量长度为 2，赋予其长度为 6 的值，可以看到 Inceptor 并不会报错

```
!set plsqlUseSlash true
DECLARE
  l_name varchar2(2)
BEGIN
   l_name :='steven'
END;
/
```

输出结果为：

output

例 257. 在上一步的基础之上，我们在 PL/SQL 中增加一个 oracle 系统预定义的异常 VALUE_ERROR，可以看到 Inceptor 也不会返回结果

```
!set plsqlUseSlash true
DECLARE
   s_name varchar2(2)
BEGIN
   s_name :='steven'
EXCEPTION
   WHEN value_error
   THEN
     dbms_output.put_line('value too large!')
END;
/
```

输出结果为：

output

8.11.14.2.3. ZERO_DIVIDE

ORACLE 中，如果除数为 0，会报出 ZERO_DIVIDE 的异常，但在 Inceptor 中，并不支持这一异常

例 258. 定义一个 int 变量，值为 100 除以 0，可以看到并不会报错

```
!set plsqlUseSlash true
DECLARE
   v_num int
BEGIN
   v_num :=(100/0)
END;
/
```

输出结果为：

output

例 259. 在上一步的基础之上，我们在 PL/SQL 中增加一个 oracle 系统预定义的异常 ZERO_DIVIDE，可以看到 Inceptor 也不会返回结果

```
!set plsqlUseSlash true
DECLARE
   v_num int
BEGIN
   v_num :=(100/0)
EXCEPTION
   WHEN zero_divide
   THEN
     dbms_output.put_line('how you can divide by zero???')
END;
/
```

输出结果为：

output

8.11.14.2.4. DUP_VAL_ON_INDEX

ORACLE 中，如果唯一索引对应的列上有重复的值，会报出 DUP_VAL_ON_INDEX 的异常，在 Inceptor 中并不支持主键的这个概念，所以关于 DUP_VAL_ON_INDEX 的异常也并不支持。

例 260. 创建一个过程，向表中插入两条数据，姓名均为’lily'

```
!set plsqlUseSlash true
CREATE or REPLACE PROCEDURE
test() is
BEGIN
   INSERT into za values('lily',23)
   INSERT into za values('lily',22)
   COMMIT
END;
/
BEGIN
test()
END;
/
```

输出结果为:

output

查看 za 表中的数据

```
SELECT * FROM za;
```

name	age
fd	24
zz	10
alice	25
zhaoliu	23
zhangsan	23
lily	29
lily	23
lily	22
john	20
sara	21
lisi	20
wangwu	22
smith	22

可以看到，在插入两条姓名均为 ’lily’ 的数据时候，Inceptor 并不会报错，而是把这两条数据全都成功地插入到了表中，也就是说 Inceptor 并不支持 DUP_VAL_ON_INDEX 异常。

8.11.14.2.5. CASE_NOT_FOUND

ORACLE 中，CASE 中若未包含相应的 WHEN ，并且没有设置 ELSE 时，会报出 CASE_NOT_FOUND 的异常，但在 Inceptor 中，并不支持这一异常。

例 261. Inceptor 并不会报出 CASE_NOT_FOUND 的异常

```
!set plsqlUseSlash true
DECLARE
    -- 声明一个字符串类型的变量 message。
    message string
    -- 声明一个变量 p_price，数据类型为表 transactions 中价格字段的类型。
    p_price transactions.price%type
    -- 声明游标 cur 用来查询表 transactions 中账号为 6513065 的价格。
    CURSOR cur IS SELECT price FROM transactions WHERE acc_num='6513065';
BEGIN
    OPEN cur
    LOOP
        -- 将游标查询到的结果放进变量 p_price 里。
        FETCH cur INTO p_price
        -- 如果没有查询到结果就退出。
        EXIT WHEN cur%NOTFOUND
        -- 依次输出所查询到的价格信息。
        DBMS_OUTPUT.PUT_LINE(p_price)
        -- 使用 case 语句，对于所输出的每一个价格，判断价格的高低状态，并附相应的值给 message，例如价格大于 5
小于 10，则 message 的信息为 'the price is good'。
        message:= CASE
        WHEN p_price<5 THEN 'price is ok'
        WHEN p_price<10 THEN 'price is good'
        WHEN p_price <15 THEN 'price is high'
        END
        -- 依次输出 message 的信息。
        DBMS_OUTPUT.PUT_LINE(message)
    END LOOP;
    CLOSE cur
END;
/
```

输出结果为：

output
12.13
price is high
6.12
price is good
9.81
price is good
7.49
price is good
7.02
price is good
18.38
null

可以看到，case 中并未包含价格大于 15 小于 20 的情况，即价格为 18.38 的情况，此时 Inceptor 对于价格为 18.38 的情况，返回 null 值，并不会报错。

8.11.14.2.6. ACCESS_INTO_NULL

ORACLE 中，企图将值写入未初始化对象的属性时，所报出的异常，但在 Inceptor 中，并不支持这一异常。

例 262. 创建一个变量为 test_type 类型，并且在 PL/SQL 中的 EXCEPTION 部分引用一个 access_into_null 异常情况名

```
!set plsqlUseSlash true
DECLARE
   v_test test_type
BEGIN
   v_test.v_name :='test'
EXCEPTION
   when access_into_null then
   dbms_output.put_line('first inittialized object v_test')
END;
/
```

输出结果为：

```
COMPILE FAILED: Semantic error: [Error 11037]
ANONYMOUS BLOCK (LINE 2, COLUMN 0, TEXT \"v_test test_type\"): Unknown identifier test_type
```

可以看到，Inceptor 并不会返回 'first inittialized object v_test'，也就是说 Inceptor 并不支持 ACCESS_INTO_NULL 这个 oracle 系统预定义的异常。

8.11.14.2.7. SELF_IS_NULL

ORACLE 中，调用一个对象类型非静态成员方法（其中没有初始化对象类型实例）的时候发生该异常，但在 Inceptor 中，并不支持这一异常。

例 263. 在 Inceptor 中创建一个对象类型

可以看到会报错，也就是说，Inceptor 中并不支持对象类型

```
CREATE type ty_name AS OBJECT(name STRING, age INT);
```

输出结果为：

```
COMPILE FAILED: Parse error: [Error 1101] line 1:7 cannot recognize input near 'CREATE type
ty_name' in ddl statement
```

因为 Inceptor 中并不支持创建对象类型，也更不能调用对象方法，所以 Inceptor 不支持 SELF_IS_NULL 这一 oracle 系统预定义的异常。

8.11.14.2.8. SYS_INVALID_ROWID

ORACLE 中，对于无效的 ROWID 字符串，会报出 SYS_INVALID_ROWID 异常，但在 Inceptor 中，并不支持这一异常。

1. oracle 中使用 SELECT 语句返回的结果集，若希望按特定条件查询前 N 条记录，可以使用伪列 ROWNUM。ROWNUM 是对结果集加的一个伪列，即先查到结果集之后再加上去的一个列（强调：先要有结果集）。简单的说 ROWNUM 是符合条件结果的序列号。它总是从 1 开始排起的。
2. rowid 确定了每条记录是在 Oracle 中的哪一个数据对象，数据文件、块、行上。能够显示表中的行是怎么存储的。

3. ROWID 的格式如下：

数据对象编号	文件编号	块编号	行编号
000000	FFF	BBBBBB	RRR

例 264. 我们利用 ROWNUM 来查询第一列（第一条插入表格的记录）的姓名和密码

可以看到 Inceptor 并不支持 ROWNUM 的用法。

```
SELECT name, password FROM user_info WHERE rownum=1;
```

输出结果为：

```
COMPILE FAILED: Semantic error: [Error 10004] Line 1:43 Invalid table alias or column
reference: (possible column names are: name, acc_num, password, citizen_id, bank_acc, reg_date,
acc_level). Error encountered near token 'rownum'
```

例 265. 查看表前两行的存储方式

可以发现 Inceptor 并不支持这种用法。

```
SELECT rowid FROM user_info WHERE rownum<2;
```

输出结果为：

```
COMPILE FAILED: Semantic error: [Error 10004] Line 1:34 Invalid table alias or column
reference: (possible column names are: name, acc_num, password, citizen_id, bank_acc, reg_date,
acc_level). Error encountered near token 'rownum'
```

因为 Inceptor 中暂不支持 ROWNUM 和 ROWID 的用法，所以，Inceptor 并不会支持 SYS_INVALID_ROWID oracle 系统预定义的异常。

8.11.14.2.9. NOT_LOGGED_ON

PL/SQL 应用程序在没有连接 oralce 数据库的情况下访问数据，会报出的异常。

Inceptor 中首先要以一个正确的用户名和密码登陆之后，才可以查看数据库中的表和数据，所以在没有登陆 Inceptor 之前，用户无法访问数据，Inceptor 也就不支持这个异常。

8.11.14.2.10. LOGIN_DENIED

ORACLE 中，PL/SQL 应用程序连接到 oracle 数据库时，提供了不正确的用户名或密码，会报出的异常。

- 在连接 Inceptor 数据库时，提供不正确的用户名或密码

```
beeline -u "jdbc:hive2://localhost:10000/default" -n hive -p 123456
scan complete in 8ms
Connecting to jdbc:hive2//localhost:10000/default
2015-11-19 21:51:11,554 INFO jdbc.Utils: Supplied authorities: localhost:10000
2015-11-19 21:51:11,556 INFO jdbc.Utils: Resolved authority: localhost:10000
Error: Could not open connection to jdbc:hive2//localhost:10000/default: Peer indicated failure:
PLAIN auth failed: Error validating LDAP user (state=08S01,code=0)
```

可以看到，以不正确的用户名或密码登陆 Inceptor，Inceptor 会报错，但不会抛出 LOGIN_DENIED 的异常。

8.11.14.2.11. 其它不支持的异常

- TIMEOUT_ON_RESOURCE ：Oracle 在等待资源时超时。
- STORAGE_ERROR：运行 PL/SQL 时，超出内存空间。
- PROGRAM_ERROR：PL/SQL 内部问题，可能需要重装数据字典 & pl./SQL 系统包。

8.11.14.3. 用户自定义异常

例 266. 定义一个异常，查询 transactions 表中的信息

```
!set plsqlUseSlash true
DECLARE
   -- 声明一个整数类型的变量。
   n_count INT
   -- 声明一个名为 too_many 的异常。
   too_many EXCEPTION
BEGIN
   -- 查询 transactions 表中的行数信息，放进 n_count 变量里。
   SELECT COUNT(*) INTO n_count FROM transactions;
   -- 定义异常 too_many 发生的条件。
   IF (n_count>5) THEN
      RAISE too_many
    END IF
   dbms_output.put_line(n_count)
EXCEPTION
   -- 引用 no_data_found 系统预定义异常。
   WHEN no_data_found THEN
     -- 异常发生时所输出的信息。
     dbms_output.put_line('no data found')
   -- 引用 too_many 用户自定义异常。
   WHEN too_many THEN
     -- 异常发生时所输出的信息。
     dbms_output.put_line('user_defined exception found:too many')
END;
/
```

输出结果

output
user_defined exception found:too many

例 267. 当上述自定义的异常没有发生时

```
!set plsqlUseSlash true
DECLARE
   n_count INT
   too_many EXCEPTION
BEGIN
   SELECT COUNT(*) INTO n_count FROM transactions;
   IF (n_count>100) THEN
      RAISE too_many
    END IF
   dbms_output.put_line(n_count)
EXCEPTION
  WHEN no_data_found THEN
     dbms_output.put_line('no data found')
  WHEN too_many THEN
     dbms_output.put_line('user_defined exception found:too many')
END;
/
```

输出结果：

output
20

8.11.14.4. 嵌套异常

8.11.14.4.1. 异常处理

Inceptor 中支持 PL/SQL 提供一个功能去处理异常，在 PL/SQL 块中叫做异常处理，使用异常处理我们能够测试代码和避免异常退出。

- PL/SQL 语句块的结构

```
DECLARE   ①
   Declaration section
 BEGIN      ②
   Execution  section
 EXCEPTION     ③
 WHEN ex_name1 THEN     ④
    Error handling statements  ⑤
 WHEN ex_name2 THEN
    Error handling statements
 WHEN Others THEN
   Error handling statements
END;
```

① 变量、常量、游标、用户定义异常的声明。

② SQL 语句和 PL/SQL 语句构成的执行程序，当执行语句违反数据库的规则时，会发生异常。

③ 程序出现异常时，捕捉异常并处理异常。

④ 引用一个异常情况名，来判断执行语句是否发生该异常。

⑤ 当执行语句发生的异常与异常情况名匹配时，所输出的信息。

- 异常处理块的结构

  程序出现异常时，捕捉异常并处理异常。

```
EXCEPTION
 WHEN ex_name1 THEN
    Error handling statements  ①
 WHEN ex_name2 THEN
    Error handling statements
 WHEN Others THEN
   Error handling statements
```

① 当异常发生时，所返回的值。

8.11.14.4.2. 异常分别出现在 inner block 和 outer block 里

内部和外部的异常处理块，分别处理 inner block 和 outer block 里的异常

例 268. 内部和外部的异常处理块，分别处理 inner block 和 outer block 里的异常

```
!set plsqlUseSlash true
DECLARE
   test1 STRING
BEGIN

   DECLARE
      test2 STRING
   BEGIN
      SELECT trans_time
       INTO test2
       FROM transactions
       WHERE acc_num=6513064;
      dbms_output.put_line(test2)
   EXCEPTION
     WHEN NO_DATA_FOUND THEN
      dbms_output.put_line('no values')
   END

   SELECT trans_id
    INTO test1
     FROM transactions
     WHERE acc_num=6513065;
   dbms_output.put_line(test1)
EXCEPTION
   WHEN TOO_MANY_ROWS THEN
      dbms_output.put_line('too many rows')
END;
/
```

输出结果：内部和外部的异常处理块，可分别成功处理 inner block 和 outer block 里的异常。

output
no values
too many rows

内部块中的异常处理块处理 outer block 里的异常，外部块中的异常处理块处理 inner block 里的异常

例 269. 内部块中的异常处理块处理 outer block 里的异常，外部块中的异常处理块处理 inner block 里的异常

```
!set plsqlUseSlash true
DECLARE
   test1 STRING
BEGIN

   DECLARE
      test2 STRING
   BEGIN
      SELECT trans_time
      INTO test2
      FROM transactions
      WHERE acc_num=6513064;
      dbms_output.put_line(test2)
   EXCEPTION
     WHEN TOO_MANY_ROWS THEN
      dbms_output.put_line('too many rows')
   END

   SELECT trans_id
   INTO test1
   FROM transactions
   WHERE acc_num=6513065;
   dbms_output.put_line(test1)
EXCEPTION
   WHEN NO_DATA_FOUND THEN
     dbms_output.put_line('no values')
END;
/
```

输出结果：只有外部块中的异常处理块可以成功处理 inner block 里的异常。

output
no values

8.11.14.4.3. 异常发生在 inner block 中

如果异常出现在内部的块中，内部异常处理块应该处理这个异常，如果内部处理块没有处理这个异常，控制会转移它的上一级的 PL/SQL 块中，由上一级对应的异常处理块处理这个异常，如果上一级也没有对应的异常处理块，程序将错误的结束。

内部块中的异常处理块处理 inner block 里的异常

例 270. 内部块中的异常处理块处理 inner block 里的异常

```
!set plsqlUseSlash true
DECLARE
   test1 STRING
BEGIN

   DECLARE
     test2 STRING
   BEGIN
      SELECT trans_time
      INTO test2
      FROM transactions
      WHERE acc_num=6513064;
     dbms_output.put_line(test2)
   EXCEPTION
    WHEN NO_DATA_FOUND THEN
     dbms_output.put_line('no values')
   END

EXCEPTION
   WHEN TOO_MANY_ROWS THEN
    dbms_output.put_line('too many rows')
END;
/
```

输出结果：内部块中的异常处理块成功处理了 inner block 里的异常。

output
no values

外部块中的异常处理块处理 inner block 里的异常

例 271. 外部块中的异常处理块处理 inner block 里的异常

```
!set plsqlUseSlash true
DECLARE
   test1 STRING
BEGIN

   DECLARE
      test2 STRING
   BEGIN
      SELECT trans_time
      INTO test2
      FROM transactions
      WHERE acc_num=6513064;
      dbms_output.put_line(test2)
   EXCEPTION
    WHEN TOO_MANY_ROWS THEN
     dbms_output.put_line('too many rows')
   END

EXCEPTION
   WHEN NO_DATA_FOUND THEN
    dbms_output.put_line('no values')
END;
/
```

输出结果：返回 inner block 里的 exception 的信息。

output
no values

没有异常处理块处理 inner block 里的异常

例 272. 没有异常处理块处理 inner block 里的异常

```
!set plsqlUseSlash true
DECLARE
   test1 STRING
BEGIN

   DECLARE
      test2 STRING
   BEGIN
      SELECT trans_time INTO test2 FROM transactions WHERE acc_num=6513064;
       dbms_output.put_line(test2)
    EXCEPTION
       WHEN TOO_MANY_ROWS THEN
        dbms_output.put_line('too many rows')
    END

EXCEPTION
   WHEN CURSOR_ALREADY_OPEN THEN
    dbms_output.put_line('cursor already open')
END;
/
```

输出结果为：没有异常处理块能够处理 inner block 里的异常，数据库抛出异常。

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error 100] NO_DATA_FOUND
**********************************************
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT \"SELECT trans_time INTO test2 FROM transactions WHERE
acc_num=6513064;\")
**********************************************
```

8.11.14.4.4. 异常发生在 outer block 中

如果异常出现在外部的块中，外部异常处理块应该处理这个异常，如果外部处理块没有处理这个异常，控制不会转移它的下一级的 PL/SQL 块中，程序将错误的结束。

外部块中的异常处理块处理 outer block 里的异常

例 273. 外部块中的异常处理块处理 outer block 里的异常

```
!set plsqlUseSlash true
DECLARE
   test1 STRING
BEGIN
   SELECT trans_id INTO test1 FROM transactions WHERE acc_num=6513065;
   dbms_output.put_line(test1)

   DECLARE
      test2 STRING
   BEGIN
      SELECT trans_time INTO test2 FROM transactions WHERE price=12.13;
      dbms_output.put_line(test2)
   EXCEPTION
     WHEN NO_DATA_FOUND THEN
      dbms_output.put_line('no values')
    END

EXCEPTION
   WHEN TOO_MANY_ROWS THEN
    dbms_output.put_line('too many rows')
END;
/
```

输出结果为：外部块中的异常处理块成功处理了 outer block 里的异常。

output
too many rows

内部块中的异常处理块处理 outer block 里的异常

例 274. 内部块中的异常处理块处理 outer block 里的异常

```
!set plsqlUseSlash true
DECLARE
   test1 STRING
BEGIN

   DECLARE
      test2 STRING
   BEGIN
      SELECT trans_time INTO test2 FROM transactions WHERE acc_num=6513064;
       dbms_output.put_line(test2)
    EXCEPTION
       WHEN TOO_MANY_ROWS THEN
        dbms_output.put_line('too many rows')
    END

EXCEPTION
   WHEN CURSOR_ALREADY_OPEN THEN
    dbms_output.put_line('cursor already open')
END;
/
```

输出结果：内部块中的异常处理块不能够处理 outer block 里的异常，数据库抛出异常。

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error 100] NO_DATA_FOUND
**********************************************
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT \"SELECT trans_time INTO test2 FROM transactions WHERE
acc_num=6513064;\")
**********************************************
```

8.11.15. 注意事项

8.11.15.1. 函数/存储过程的版本兼容

Inceptor PL/SQL 在特定的版本上会对后端引擎进行比较大的改进优化，因此在某些版本上的某些已创建的 PL/SQL 函数和过程需要升级，可由星环科技提供的自动升级工具完成，以完全保证前后版本的兼容性。

8.11.15.2. Inceptor对PL/SQL中分号的支持

Inceptor 中默认对 PL/SQL 语句块的分号是不支持的，也就是说常规下，Inceptor 中的 PL/SQL 只有在语句块结束的地方允许有分号。我们可以通过命令语句手动打开支持。使用命令的时候，我们需要在 pl/sql 文件的头部和尾部，各加一条命令，具体可参考以下案例。

- Inceptor 要求在一个完整的 plsql 语句块后面，加上一个新行，这行只包含‘/’字符。
- 此时 Inceptor 执行‘/’ 之前的 plsql 语句；如果不加‘/’ 的话，Inceptor 将不会执行之前的任何 plsql 语句。
- 一个完整的 plsql 语句块的最后如果不加‘/’字符，则 Inceptor 会认为语句块并没有结束，不会执行该语句块。

- 在 InceptorServer 2 中我们可以通过以下命令，来完成 Inceptor 对 PL/SQL 中分号的支持

```
!set plsqlUseSlash true
!set plsqlUseSlash false
```

需要注意的是该命令语句的后面，不可以加分号。

例 275. 在 server2 中打开分号支持

```
-- 设置 plsqlUseSlash 属性为 true。
!set plsqlUseSlash true
-- 创建一个名为 test() 的过程。
CREATE OR REPLACE PROCEDURE test() IS
DECLARE
   var INT;
BEGIN
   var:=1;
   WHILE var !=5
   -- 在变量 var 不等于 5 的情况下，依次输出 var 的值。
   LOOP
     dbms_output.put_line("while loop body"||var||".");
     var := var+1;
   END LOOP;
END;
/

-- 调用过程 test()。
BEGIN
  test();
END;
/

-- 设置 plsqlUseSlash 属性为 false。
!set plsqlUseSlash false
```

输出结果为：

output
while loop body1.
while loop body2.
while loop body3.
while loop body4.

8.11.15.3. PL/SQL 中结果的打印

Inceptor 中，PL/SQL 中直接执行 SELECT 语句是默认不打印结果的。要想查看结果，需要通过 set plsql.show.sqlresults=true 来打印结果。

例 276. 创建一个匿名块，查询表 transactions 中价格为 12.13 的交易时间和交易类型

```
!set plsqlUseSlash true
BEGIN
   SELECT trans_time,trans_type
     FROM transactions
      WHERE price=12.13;
END;
/
```

输出结果为：

output

- 设置 plsql.show.sqlresults 的值为 true

```
set plsql.show.sqlresults=true;
```

例 277. 创建一个匿名块，查询表 transactions 中价格为 12.13 的交易时间和交易类型

```
!set plsqlUseSlash true
set plsql.show.sqlresults=true;
BEGIN
   SELECT trans_time,trans_type
    from transactions
    WHERE price=12.13;
END;
/
```

输出结果为：

output
20140105100520
b

8.11.15.4. 标识符

8.11.15.4.1. SELECT/SELECT INTO

SELECT 或 SELECT INTO 语句后如果跟函数调用或赋值语句，则 SELECT 或 SELECT INTO 语句不能以 FROM <TABLE_NAME> 结尾，否则函数名会被识别为 alias，Inceptor 会报语义错误。解决方法是可以在结尾处 TABLE_NAME 后加上表的化名，或者在 SELECT 或 SELECT INTO 语句结尾处加上分号。

案例一：SELECT/SELECT INTO 语句后紧跟函数调用

例 278. SELECT/SELECT INTO 语句后紧跟函数调用

```
DECLARE
  v1 STRING
  v2  INT:=1000
BEGIN
  SELECT price FROM transactions
  DBMS_OUTPUT.PUT_LINE(v2)
END;
```

可以看到 Inceptor 会报出语法错误。

```
COMPILE FAILED: Parse error: [Error 1110] line 5:18 All alternatives of rule[query regular
body] can't match the input in query or non sql statement
```

例 279. 解决方案一：在结尾处 TABLE_NAME 后面加上表的化名

```
DECLARE
  v1 STRING
  v2  INT:=1000
BEGIN
  SELECT price FROM transactions t1
  DBMS_OUTPUT.PUT_LINE(v2)
END;
```

输出结果：

output
1000

例 280. 解决方案二：打开分号支持，在 SELECT/SELECT INTO 语句结尾处加上分号。

```
!set plsqlUseSlash true
DECLARE
  v1 STRING;
  v2  INT:=1000;
BEGIN
  SELECT price FROM transactions;
  DBMS_OUTPUT.PUT_LINE(v2);
END;
/
```

输出结果：

output
1000

案例二：SELECT/SELECT INTO 语句后紧跟赋值语句

例 281. SELECT/SELECT INTO 语句后紧跟赋值语句

```
DECLARE
  v1 STRING
  v2  INT:=1000
BEGIN
  SELECT price FROM transactions
  v1 := 'hello world'
  DBMS_OUTPUT.PUT_LINE(v1)
END;
```

可以看到 Inceptor 会报出语法错误。

```
COMPILE FAILED: Parse error: [Error 1110] line 5:18 All alternatives of rule[query regular
body] can't match the input in query or non sql statement
```

例 282. 解决方案一：在结尾处 TABLE_NAME 后面加上表的化名。

```
DECLARE
  v1 STRING
  v2  INT:=1000
BEGIN
  SELECT price FROM transactions t1
  v1 := 'hello world'
  DBMS_OUTPUT.PUT_LINE(v1)
END;
```

输出结果：

output
hello world

例 283. 解决方案二：打开分号支持，在 SELECT/SELECT INTO 语句结尾处加上分号

```
!set plsqlUseSlash true
DECLARE
  v1 STRING;
  v2  INT:=1000;
BEGIN
  SELECT price FROM transactions;
  v1 := 'hello world'
  DBMS_OUTPUT.PUT_LINE(v1)
END;
/
```

输出结果为：

output
hello world

8.11.15.4.2. FOR LOOP … END LOOP

FOR LOOP后面的END LOOP后可以接一个可选的标识符，如果END LOOP后面紧跟函数调用或赋值语句，则Inceptor也会报出语法错误，解决方法是可以在END LOOP的结尾处加上分号。

案例一：END LOOP 后紧跟函数调用

例 284. END LOOP 后紧跟函数调用

```
DECLARE
  v1 STRING :='hello world'
BEGIN
FOR test IN 1..5 LOOP
DBMS_OUTPUT.PUT_LINE('test为:'||test)
END LOOP
DBMS_OUTPUT.PUT_LINE(v1)
END;
```

可以看到，Inceptor 会报出语法错误。

```
COMPILE FAILED: Parse error: [Error 1101] line 7:11 cannot recognize input near '. PUT_LINE ('
in labeled statement
```

例 285. 解决方案：在 END LOOP 的结尾处加上分号

```
!set plsqlUseSlash true
DECLARE
  v1 STRING :='hello world'
BEGIN
FOR test IN 1..5 LOOP
DBMS_OUTPUT.PUT_LINE('test为:'||test)
END LOOP;
DBMS_OUTPUT.PUT_LINE(v1)
END;
/
```

输出结果为：

output
test为:1
test为:2
test为:3
test为:4
test为:5
hello world

案例二：END LOOP 后紧跟赋值语句

例 286. END LOOP 后紧跟赋值语句

```
DECLARE
  v1 STRING :='hello world'
  v2 STRING
BEGIN
FOR test IN 1..5 LOOP
DBMS_OUTPUT.PUT_LINE('test为:'||test)
END LOOP
v2 := 'hello world again'
DBMS_OUTPUT.PUT_LINE(v1)
DBMS_OUTPUT.PUT_LINE(v2)
END;
```

可以看到，Inceptor 会报出语法错误。

```
COMPILE FAILED: Parse error: [Error 1101] line 8:3 cannot recognize input near ':= 'hello world
again' DBMS_OUTPUT' in labeled statement
```

例 287. 解决方案:在 END LOOP 的结尾处加上分号

```
!set plsqlUseSlash true
DECLARE
  v1 STRING :='hello world'
  v2 STRING
BEGIN
FOR test IN 1..5 LOOP
DBMS_OUTPUT.PUT_LINE('test为:'||test)
END LOOP;
v2 := 'hello world again'
DBMS_OUTPUT.PUT_LINE(v1)
DBMS_OUTPUT.PUT_LINE(v2)
END;
/
```

输出结果为:

output
test为:1
test为:2
test为:3
test为:4
test为:5
hello world
hello world again

8.11.15.5. 标准 SQL 调用 PL/SQL 函数必须满足的条件

8.11.15.5.1. 必须是函数，不能是过程

Inceptor 中支持 SQL 语句中直接调用 PL/SQL 函数，但是不可以调用 PL/SQL 的过程。

不支持 SQL 语句中调用 PL/SQL 过程

- 创建一个名为 test() 的过程

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE
test (acc int) IS
BEGIN
   PUT_LINE('651306' || acc);
END;
/
```

- 调用 test() 过程

```
!set plsqlUseSlash true
BEGIN
   test(5);
END;
/
```

- 可以看到 Inceptor 中单独调用 test() 过程是可以的。

```
!set plsqlUseSlash true
BEGIN
   test(5);
END;
/

-- 输出结果如下：

+----------+
|  output  |
+----------+
| 6513065  |
+----------+
```

- 但是不可以在 SQL 语句中调用 test() 过程，以下 SQL 语句在 Inceptor 中是不支持的

```
SELECT test(5) FROM transactions;
```

> 相对应的，Inceptor 中不支持 SQL 语句中直接调用 PL/SQL 过程，但是可以直接调用 PL/SQL 函数，可参考以下用例。

- 创建一个名为 plsql_function 的 PL/SQL 函数

```
!set plsqlUseSlash true
DROP FUNCTION IF EXISTS plsql_function;
CREATE FUNCTION plsql_function(t_name string)
  return string as
BEGIN
  return t_name
END plsql_function;
/
```

- 在 SQL 语句中调用刚刚创建的 plsql_function 函数

```
SELECT plsql_function(password) AS psw FROM user_info;
```

输出结果为：

psw
115591
205239
531547
841242
986634
737297
600709
990590
015859
783438

可以看到 Inceptor 中可以返回正确的值，也就是说 SQL 语句中可以直接调用 PL/SQL 函数，但是不可以直接调用 PL/SQL 过程。

8.11.15.5.2. PL/SQL 函数返回值必须是基本类型

在创建 PL/SQL 函数的时候，会定义一个返回值的数据类型，在 Inceptor 中要求该数据类型必须为基本类型，也就是一些标量类型区别于复合类型，引用类型和 LOB 类型。标量类型仅存放单个的值，常见的基本类型为 int/double/string/char/varchar 等。对于返回值为非基本类型的数据类型，Inceptor 会报错。

PL/SQL 函数返回值必须是基本类型

- 创建一个名为 plsql_function 的 PL/SQL 函数，函数返回值是 string 类型。

```
!set plsqlUseSlash true
DROP FUNCTION IF EXISTS plsql_function;
CREATE FUNCTION plsql_function(t_name string)
 return string as
BEGIN
 return t_name
END plsql_function;
/
```

- 在 SQL 语句中调用刚刚创建的 plsql_function 函数

```
SELECT plsql_function(password) AS psw FROM user_info;
```

输出结果为：

psw
115591
205239
531547
841242
986634
737297
600709
990590
015859
783438

8.11.15.5.3. PL/SQL 函数中不能有标准 SQL 语句

- 创建一个名为 test 的 PL/SQL 函数,返回表 user_info 中的用户姓名

```
!set plsqlUseSlash true
CREATE or REPLACE FUNCTION test(t_name string) RETURN string
AS
BEGIN
  SELECT name from user_info;
    RETURN t_name
END;
/
```

- 在 SQL 语句中调用刚刚创建的 test 函数

```
SELECT test(name) FROM user_info;
```

可以看到虽然 PL/SQL 函数创建成功了，但是直接在 SQL 语句中调用该函数会报错，Inceptor 中并不支持这样的用法。

```
1
```

8.11.15.6. 不支持 RETURN INTO 语句

例 288. 利用 returning into 语句来返回对表 user_info 进行改动的记录的账号。

```
!set plsqlUseSlash true
DECLARE
   l_acc_num user_info.acc_num%type
BEGIN
   UPDATE user_info SET password='111111' WHERE acc_num='6513065'
    RETURNING acc_num INTO l_acc_num
   dbms_output.put_line('UPDATE acc_num='||l_acc_num)
   DELETE FROM user_info where password='783438'
    RETURNING acc_num INTO l_acc_num
    dbms_output.put_line('DELETE acc_num='||l_acc_num)
   COMMIT
END;
/
```

可以看到 Inceptor 会报错，不支持 turning into 语句。

```
COMPILE FAILED: Parse error: [Error 1101] line 5:18 cannot recognize input near 'INTO l_acc_num
dbms_output' in non sql statement
```

8.11.15.7. 查看预定义函数/过程/包

8.11.15.7.1. 相关命令合集

- 查看已有 PL/SQL 函数/过程

  （不指定 db_name 的话即对当前数据库操作）

```
SHOW PLSQL FUNCTIONS db_name;
```

- 查看已有 PL/SQL 包

  （不指定 db_name 的话即对当前数据库操作）

```
SHOW PLSQL PACKAGES db_name;
```

- 查看某一 PL/SQL 函数/过程的详细信息

  （EXTENDED 关键字会列出该 PL/SQL 函数/过程的原文）

```
DESC PLSQL FUNCTION EXTENDED  function_name
```

- 查看某一 PLSQL 包的详细信息

（EXTENDED 关键字会列出该 PLSQL 包的原文）

```
DESC PLSQL PACKAGE EXTENDED package_name
```

- 创建函数/过程/包/包体

```
CREATE (OR REPLACE) FUNCTION/PROCEDURE/PACKAGE/PACKAGE BODY
```

- 删除函数/过程/包/包体

```
DROP PLSQL FUNCTION/PROCEDURE/PACKAGE
```

> 由于 PLSQL 是运行在 Inceptor Server端的 SESSION 中，如果 PLSQL 语句正在运行时，客户端通过 Ctrl+C 结束执行，此时 Server 端由于还在执行 PLSQL，无法侦听到客户端的结束请求，该 PLSQL 会继续执行到结束。如果它包含了大量的 SQL 语句执行或者存在死循环，会耗费大量的服务端资源。所以我们需要手动执行一些命令来终止服务端对 PLSQL 的执行。

- 查看正在运行的 PLSQL 程序的 SESSION ID

```
PS PLSQL
```

这条命令会显示 SESSION ID 和 PLSQL 语句，使用者需要从中找到自己想要终止的 PLSQL 的 SESSION ID

- 终止正在运行的 PLSQL 程序

```
KILL PLSQL <SESSION_ID>
```

这条命令会发送一个终止信号给该 SESSION ID 中的 PLSQL 进程，待该进程下一次自检之时，发现有终止信号，则会结束自己。

8.11.15.7.2. 案例合集

- 例1：查看已有 PLSQL 函数（不指定 db_name）

```
SHOW PLSQL FUNCTIONS;
```

plsql functions
--------System functions--------
sqlcode(void)

plsql functions
sqlerrm(void)
get_columns(string,nestedtable<string>)
raise_application_error(int,string,bool)
set_env(string,string)
get_env(string)
put_line(string)
sqlerrm(int)
--------User defined functions----------
default.calc_stats(double,double,double,double)
default.grant_priv(string,string)
default.create_test(string,int,double)
default.insert_proc(string)

plsql functions
default.revoke_priv(string, string)
default.update_proc(string)
default.update_proc(string, string)
default.overload_test(int, int)
default.overload_test(int, string)
default.overload_test(string, string)
default.dynamic_sql_test(string, int, double)
default.overload_test_proc(int, int)
default.overload_test_proc(int, string)
default.overload_test_proc(string, string)

plsql functions
default.error_dynamic_sql_test(void)

- 例2：查看某一 PL/SQL 函数/过程的详细信息

  使用 EXTENDED 关键字查看自定义过程 create_test 的详细信息

```
DESC PLSQL FUNCTION EXTENDED create_test;
```

description
Prototype:
PROCEDURE manual_pl_ora.create_test(v_name IN STRING, v_age IN INT, v_grade IN DOUBLE)
Text:
create or replace procedure create_test(v_name in string, v_age in int, v_grade in double)
is
begin
insert into zara values (v_name,v_age,v_grade)
end

- 例3：查看已有 PLSQL 包（不指定 db_name）

```
SHOW PLSQL PACKAGES;
```

plsql packages
--------System packages---------
dbms_output
owa_util
--------User defined packages---------

- 例 4：查看某一 PLSQL 包的详细信息

  使用 EXTENDED 关键字查看预定义包 owa_util 的详细信息

```
DESC PLSQL PACKAGE EXTENDED owa_util;
```

description
Head:
PACKAGE owa_util IS
PROCEDURE owa_util.who_called_me(owner OUT STRING, name OUT STRING, lineno OUT INT, type OUT STRING)
Text:
N/A
Body:
PACKAGE BODY owa_util IS
PROCEDURE owa_util.who_called_me(owner OUT STRING, name OUT STRING, lineno OUT INT, type OUT STRING)
Text:
N/A

- 例 5：删除函数/过程/包/包体

删除自定义过程 overload_test_proc;

```
drop plsql procedure overload_test_proc;
```

再次查看已有的 PL/SQL 函数/过程，可以发现自定义过程 overload_test_proc 已被删除。

```
show plsql functions;
```

plsql functions
--------System functions--------
sqlcode(void)
sqlerrm(void)
get_columns(string,nestedtable<string>)

plsql functions
raise_application_error(int, string, bool)
set_env(string, string)
get_env(string)
put_line(string)
sqlerrm(int)
--------User defined functions--------
default.calc_stats(double, double, double, double)
default.grant_priv(string, string)
default.create_test(string, int, double)
default.insert_proc(string)
default.revoke_priv(string, string)
default.update_proc(string)

plsql functions
default.update_proc(string,string)
default.overload_test(int,int)
default.overload_test(int,string)
default.overload_test(string,string)
default.dynamic_sql_test(string,int,double)
default.error_dynamic_sql_test(void)

- 例 6：查看正在运行的 PLSQL 程序的 SESSION ID

```
PS  PLSQL;
```

- 例7：终止正在运行的 PLSQL 程序

```
KILL PLSQL <SESSION_ID>
```

8.11.15.8. PL/SQL 中的 PUT_LINE 打印

在 Inceptor 中，可以使用系统预定义函数 PUT_LINE 打印出常量或变量的值，需要注意以下几个方面:

- 如果程序正常结束，PUT_LINE 会统一打印到终端。
- 如果程序中出现了未被处理的异常，默认在终端只打印异常栈，PUT_LINE 内容需要去 hive.log 中查看。
- 可以设置开关 set plsql.cache.output=true，可以将 PUT_LINE 正常打印出的内容连同异常栈一起打印到终端。

- 因为开启该开关会占用系统资源，所以建议在需要调试时开启，在程序正常执行时关闭（默认关闭）。

例 289. 程序正常运行

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE p1(value string)
IS
DECLARE
  ex exception
BEGIN
  BEGIN
    value := 'p1: ' || value
       put_line(value)
    raise ex
  EXCEPTION
  when ex then
     put_line('caught ex exception')
  END
BEGIN
  value := value || ' more'
     put_line('after exception ' || value)
END
END;
/
CREATE OR REPLACE PROCEDURE p2(value string)
IS
BEGIN
  value := 'p2: ' || value
     put_line(value)
  p1(value)
END;
/
CREATE OR REPLACE PROCEDURE p3(value string) is
BEGIN
  value := 'p3: ' || value
     put_line(value)
  p2(value)
END;
/
DECLARE
  value string
  id_v int
  ret_v int
BEGIN
  value := 'init value'
     put_line(value)
  p3(value)
END;
/
```

输出结果为:

output
init value
p3: init value
p2: p3: init value
p1: p2: p3: init value
caught ex exception
after exception p1: p2: p3: init value more

例 290. 程序运行过程中出现异常，默认只打印出异常栈

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE p1(value string)
IS
DECLARE
  ex exception
BEGIN
  BEGIN
    value := 'p1: ' || value
       put_line(value)
    raise ex
   END
END;
/
CREATE OR REPLACE PROCEDURE p2(value string)
IS
BEGIN
  value := 'p2: ' || value
     put_line(value)
  p1(value)
END;
/
CREATE OR REPLACE PROCEDURE p3(value string) is
BEGIN
  value := 'p3: ' || value
     put_line(value)
  p2(value)
END;
/
DECLARE
  value string
  id_v int
  ret_v int
BEGIN
  value := 'init value'
     put_line(value)
  p3(value)
END;
/
```

输出结果为:

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error 1] USER_DEFINED
**********************************************
PROCEDURE 'p1' (LINE 9, COLUMN 0, TEXT \"raise ex\")
PROCEDURE 'p2' (LINE 6, COLUMN 0, TEXT \"p1(value)\")
PROCEDURE 'p3' (LINE 5, COLUMN 0, TEXT \"p2(value)\")
ANONYMOUS BLOCK (LINE 8, COLUMN 0, TEXT \"p3(value)\")
**********************************************
```

例 291. 手动打开开关，将正常打印出的内容连同异常栈一起打印到终端

```
!set plsqlUseSlash true
-- 手动设置 plsql.cache.output 属性为 true。
set plsql.cache.output=true;
CREATE OR REPLACE PROCEDURE p1(value string)
IS
DECLARE
  ex exception
BEGIN
  BEGIN
    value := 'p1: ' || value
    dbms_output.put_line(value)
    raise ex
   END
END;
/
CREATE OR REPLACE PROCEDURE p2(value string)
IS
BEGIN
  value := 'p2: ' || value
  dbms_output.put_line(value)
  p1(value)
END;
/
CREATE OR REPLACE PROCEDURE p3(value string) is
BEGIN
  value := 'p3: ' || value
  dbms_output.put_line(value)
  p2(value)
END;
/
DECLARE
  value string
  id_v int
  ret_v int
BEGIN
  value := 'init value'
  dbms_output.put_line(value)
  p3(value)
END;
/
set plsql.cache.output=false;
```

输出结果为:

可以看到在异常 ex 抛出之前的内容，连同异常栈的内容一块被打印到终端。

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error 1] USER_DEFINED
Program Output:
init value
p3: init value
p2: p3: init value
p1: p2: p3: init value
*********************************************
PROCEDURE 'p1' (LINE 9, COLUMN 0, TEXT \"raise ex\")
PROCEDURE 'p2' (LINE 6, COLUMN 0, TEXT \"p1(value)\")
PROCEDURE 'p3' (LINE 5, COLUMN 0, TEXT \"p2(value)\")
ANONYMOUS BLOCK (LINE 8, COLUMN 0, TEXT \"p3(value)\")
*********************************************
```

8.11.15.9. Hive 异常的处理

在异常章节中，我们可以了解到，Inceptor 中 PL/SQL 中的异常包括标准 PL/SQL 异常和 Hive 异常，其中 PL/SQL 异常包括系统预定义异常和用户自定义异常，Hive 异常即在不满足 Inceptor 数据库语法规范情况下，发生的异常 默认情况下，Hive 异常不被 PLSQL 处理（即使有 WHEN OTHERS 语句也不会处理 Hive 异常），如果需要在 PLSQL 中捕获 Hive 异常，需要设置变量 plsql.catch.hive.exception 为 true，此时 Hive 异常能被 HIVE_EXCEPTION 与 OTHERS 捕获。

例 292. 默认情况下，Hive 异常不被 PLSQL 处理

```
!set plsqlUseSlash true
BEGIN
   UPDATE transactions SET price = 12.12 WHERE trans_time = 20140105100520
EXCEPTION
WHEN HIVE_EXCEPTION THEN
   DBMS_OUTPUT.PUT_LINE('Exception caught, error code = ' || sqlcode() || ', error message = '
|| sqlerrm())
END;
/
```

输出结果为:

```
COMPILE FAILED: Semantic error: [Error 40005]
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT \"UPDATE transactions SET price = 12.12 WHERE
trans_time = 20140105100520\"): Update/Delete/Merge operations cannot apply on views or non-
transactional inceptor tables
```

例 293. 默认情况下，WHEN OTHERS 语句也不会处理 Hive 异常

```
!set plsqlUseSlash true
BEGIN
   UPDATE transactions SET price = 12.12 WHERE trans_time = 20140105100520
EXCEPTION
WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE('Exception caught, error code = ' || sqlcode() || ', error message = '
|| sqlerrm())
END;
/
```

输出结果为:

```
COMPILE FAILED: Semantic error: [Error 40005]
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT \"UPDATE transactions SET price = 12.12 WHERE
trans_time = 20140105100520\"): Update/Delete/Merge operations cannot apply on views or non-
transactional inceptor tables
```

- 手动设置变量 plsql.catch.hive.exception 为 true

```
SET plsql.catch.hive.exception=true;
```

手动设置命令，使用 HIVE_EXCEPTION 捕获 Hive 异常

为了让 HIVE_EXCEPTION 通过匿名块正常打印，而不是在编译时报错，必须关闭开关 plsql.compile.dml.check.semantic，禁止编译过程对 PL/SQL 内部语法进行检查。

```
!set plsqlUseSlash true
set plsql.catch.hive.exception=true;
set plsql.compile.dml.check.semantic=false;
BEGIN
   UPDATE transactions SET price = 12.12 WHERE trans_time = 20140105100520
EXCEPTION
WHEN HIVE_EXCEPTION THEN
   DBMS_OUTPUT.PUT_LINE('Exception caught, error code = ' || sqlcode() || ', error message = '
|| sqlerrm())
END;
/
```

输出结果为:

output
Exception caught, error code = -99999, error message = Update/Delete/Merge operations cannot apply on views or non-transactional inceptor tables

例 294. 手动设置命令，使用 OTHERS 捕获 Hive 异常

```
!set plsqlUseSlash true
SET plsql.catch.hive.exception=true;
set plsql.compile.dml.check.semantic=false;
BEGIN
   UPDATE transactions SET price = 12.12 WHERE trans_time = 20140105100520
EXCEPTION
WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE('Exception caught, error code = ' || sqlcode() || ', error message = '
|| sqlerrm())
END;
/
```

输出结果为:

output
Exception caught, error code = -99999, error message = Update/Delete/Merge operations cannot apply on views or non-transactional inceptor tables

8.11.15.10. Debug

Inceptor 中如果程序运行中出现了未被处理的异常，则可以在终端的报错中查看异常栈的信息，了解异常发生的位置所在。

例 295. 打印异常栈

```
!set plsqlUseSlash true
-- 创建过程 p1，在过程 p1 内定义一个名为 ex 的异常，该异常的 Error code 为 -1024。
CREATE OR REPLACE PROCEDURE p1(value string)
IS
DECLARE
    -- 抛出异常 ex（在这里直接抛出自定义异常，并没有给出相应的异常处理，这也是整个程序最后报错的根本原因）。
    ex EXCEPTION
    pragma exception_init(ex, -1024)
BEGIN
   value := 'p1: ' || value
   dbms_output.put_line(value)
   FOR i in 1 .. 10 LOOP
     raise ex
   END LOOP;
END;
/
-- 创建过程 p2。
CREATE OR REPLACE PROCEDURE p2(value string)
IS
BEGIN
  value := 'p2: ' || value
  dbms_output.put_line(value)
  -- 在过程 p2 内调用过程 p1。
  p1(value)
END;
/
-- 创建过程 p3。
CREATE OR REPLACE PROCEDURE p3(value string)
IS
BEGIN
  value := 'p3: ' || value
  dbms_output.put_line(value)
  -- 在过程 p3 内调用过程 p2。
  p2(value)
END;
/
-- 最后执行过程 p3的内容。
DECLARE
   value string
   id_v int
   ret_v int
BEGIN
  value := 'init value'
  dbms_output.put_line(value)
  p3(value)
END;
/
```

输出结果为：

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -1024] USER_DEFINED
*********************************************
PROCEDURE 'p1' (LINE 11, COLUMN 0, TEXT \"raise ex\")
PROCEDURE 'p2' (LINE 7, COLUMN 0, TEXT \"p1(value)\")
PROCEDURE 'p3' (LINE 7, COLUMN 0, TEXT \"p2(value)\")
ANONYMOUS BLOCK (LINE 8, COLUMN 0, TEXT \"p3(value)\")
*********************************************
```

从异常栈的信息中，我们可以看到，直接导致程序运行失败的位置是在最后一段 PL/SQL 语句块中，调用了 p3 这个过程，而根本的原因在过程 p1 内直接抛出异常，没有给出异常处理的内容。

例 296. 给上述例子加上相应的异常处理，使得程序正常运行。

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE p1(value string)
IS
DECLARE
  ex exception
BEGIN
  BEGIN
    value := 'p1: ' || value
    dbms_output.put_line(value)
    raise ex
  EXCEPTION
  when ex then
  dbms_output.put_line('caught ex exception')
  END
BEGIN
  value := value || ' more'
  dbms_output.put_line('after exception ' || value)
END
END;
/
CREATE OR REPLACE PROCEDURE p2(value string)
IS
BEGIN
  value := 'p2: ' || value
  dbms_output.put_line(value)
  p1(value)
END;
/
CREATE OR REPLACE PROCEDURE p3(value string) is
BEGIN
  value := 'p3: ' || value
  dbms_output.put_line(value)
  p2(value)
END;
/
DECLARE
  value string;
  id_v int;
  ret_v int;
BEGIN
  value := 'init value';
  dbms_output.put_line(value);
  p3(value);
END;
/
```

输出结果为:

output
init value
p3: init value
p2: p3: init value
p1: p2: p3: init value
caught ex exception
after exception p1: p2: p3: init value more

8.12. SQL PL 指南（DB2 方言）

Inceptor SQL PL是兼容 DB2 SQL PL 的过程语言。在使用 Inceptor SQL PL 和 Inceptor 进行交互前，请确保您打开 DB2 方言开关。

8.12.1. Inceptor SQL PL入门

8.12.1.1. 输出

Inceptor 中有两种输出方式，可以调用 DBMS_OUTPUT 包内函数 PUT_LINE，也可以直接调用系统预定义函数 PUT_LINE 来输出存储过程或者函数中的信息。

- **DBMS_OUTPUT.PUT_LINE**

```
DBMS_OUTPUT.PUT_LINE('<values>');
```

- **PUT_LINE**

```
PUT_LINE('<values>');
```

8.12.1.2. 声明

Inceptor 中声明变量，游标，自定义数据类型时，都需要用到关键字 DECLARE。

- **DECLARE Variables**

```
DECLARE <variable_name> <datatype> [DEFAULT <value>];
```

- **DECLARE CURSOR**

```
DECLARE <cursorname> CURSOR [WITH HOLD] [WITH RETURN TO CALLER|CLIENT] FOR <sql-statement>;
```

- **DECLARE Row data type**

```
DECLARE TYPE transactiontype AS ROW (variable_name data_type,variable_name
data_type,variable_name data_type,...);
```

8.12.1.3. 赋值

Inceptor 中有多种赋值语句，可根据具体应用场景，选择相应的赋值语句。

- **SET**

```
SET <variable_name> = <value> ;
```

- **VALUES INTO**

```
VALUES <value> INTO <variable_name> ;
```

- SELECT INTO

```
SELECT <sqlstatement> INTO <variable_name> ;
```

8.12.1.4. 过程

- CREATE PROCEDURE

```
CREATE [OR REPLACE] PROCEDURE <procedure_name> [ ( {optional parameters} ) ] [{optional
procedure attributes}]
```

- CALL

Inceptor 中可以使用关键字 CALL 来调用创建的存储过程。

```
CALL procedure_name ( <inputvalue1>, <inputvalue2>, ... );
```

8.12.1.5. 函数

- CREATE FUNCTIONS

```
CREATE [OR REPLACE] FUNCTION  <function_name> [ ( {optional parameters} ) ] [{optional function
attributes}]
```

8.12.1.6. 游标

- DECLARE CURSOR

```
DECLARE cursor_name CURSOR [WITH RETURN] FOR  sql_statement;
```

- OPEN CURSOR

```
OPEN cursor_name;
```

- FETCH CURSOR

```
FETCH cursor_name INTO Variable;
```

- CLOSE CURSOR

```
CLOSE cursor_name;
```

- ALLOCATE CURSOR

```
ALLOCATE <cursor-name> CURSOR FOR RESULT SET <locator-variable>;
```

8.12.1.7. 异常

- DECLARE Condition

```
DECLARE condition_name FOR SQLSTATE '<value>';
```

- DECLARE Condition handler

```
DECLARE handler-type HANDLER FOR condition_name;
```

- SIGNAL

```
SIGNAL SQLSTATE [value] <sqlstate> [SET MESSAGE_TEXT = <variable> or <diagnostic string
constant>];
```

- RESIGNAL

```
RESIGNAL SQLSTATE [VALUE] <sqlstate> [SET MESSAGE_TEXT = <variable> or <diagnostic string
constant>];
```

8.12.1.8. 动态 SQL

- EXECUTE IMMEDIATE

```
EXECUTE IMMEDIATE <sql-statement>;
```

- PREPARE

```
PREPARE <sql-statement>  FROM <variable>;
```

- EXECUTE

```
EXECUTE <statement-name>  [USING <input-variable> [,<input-variable>,...] ];
```

8.12.2. Inceptor SQL PL 指南中的表

在 Inceptor_sqlpl 指南中，创建了几张表用于演示操作，以下是表的具体内容

- 交易信息表 transactions：

  其中 trans_id，acc_num，trans_time，trans_type，stock_id 均为 string 类型，price 和 amount 均为 double 类型。

trans_id	acc_num	trans_time	trans_type	stock_id	price	amount
943197522	6513065	20140105100520	b	AA7105670	12.13	200

trans_id	acc_num	trans_time	trans_type	stock_id	price	amount
929634984	3912384	20140205140521	b	UA1467891	11.11	300
499506900	6513065	20140506133109	s	CA2789982	6.12	100
209441379	3912384	20140430111523	s	CX5397790	4.50	1000
648230055	0700735	20140315111111	s	DT7966575	22.66	200
719753265	3912384	20140328102400	b	BY8490909	68.43	100
975639131	0700735	20140611102830	s	AT6934136	5.30	200
991691937	2755506	20140702113025	s	VR2575735	7.52	1300
289018112	6513065	20140916105811	b	UT7592045	9.81	500
162742112	2394923	20141031135018	s	UC1610649	12.21	500
597565609	3912384	20140214141519	s	IU1775004	4.16	600
459590958	0700735	20140430143020	b	XJ9717497	5.25	1000
594819547	5224133	20140801110003	b	GL2547626	6.36	800
895916502	6513065	20141225133500	s	KC9102028	7.49	1100
900192386	6670192	20141130113905	s	XC1915304	8.64	900
952639648	6670192	20140314145958	s	CP7629713	10.31	400
404905188	6513065	20140628133001	b	SH6277444	7.02	100
952110653	6600641	20140228140005	s	GH6828501	9.16	100
817414815	5224133	20140331115900	s	ZX5373511	10.03	800
213859826	6513065	20140508094805	b	CL2121979	18.38	700

- 用户信息表 user_info：

name	acc_num	password	citizen_id	bank_acc	reg_date	acc_level
马**	6513065	115591	14**7	960*41	20110101	A

name	acc_num	password	citizen_id	bank_acc	reg_date	acc_level
祝**	6670192	205239	23**8	737*71	20100101	C
华*	5224133	531547	42**1	326*07	20080214	B
魏**	3912384	841242	52**4	685*48	20091202	A
宁**	4580952	986634	42**7	977*76	20081031	D
邱*	0700735	737297	34**8	143*18	20121024	A
李*	8725869	600709	46**3	430*84	20130702	E
潘**	6600641	990590	51**5	484*08	20110430	C
李**	2755506	015859	31**9	424*37	20110916	D
管**	2394923	783438	33**4	999*74	20141003	C

- 非分区表 zara：

name	age	gpa
lisi	20	2.0
wangwu	22	3.9
smith	22	3.1
zhaoliu	23	2.0
sara	21	3.8
zhangsan	23	2.0
lily	29	3.6

- 非分区表 za：

name	age
john	20
fd	24
zz	10
alice	25

- 非分区表 ra：

name	gpa
smith	3.4
lisi	3.3
ll	3.3
lily	3.6

- 非分区表 holotest2：

age	gpa
22	4.3
20	4.0
21	3.9

- 分区表 zza：

name	age	level（分区键）
zhaoliu	23	A
john	30	A
john	20	B
wangwu	22	C
lily	30	C
alice	25	C
john	20	C
sara	21	C

8.12.3. 基础知识

8.12.3.1. 快速入门

Inceptor 中支持 DB2 SQL PL 中的相关语法规范，用户在创建 SQL PL 语句块时，必须指定执行体的开始与结束部分，具体使用方法可参考以下示例。

- 例 1：直接打印常量

```
!set plsqlUseSlash true
-- 语句块的开始，必要部分，不可缺省。
BEGIN
  -- 执行体部分，此处为使用 PUT_LINE 函数直接打印出字符串的内容。
  PUT_LINE('hello world');
-- 语句块的结束，必要部分，不可缺省。
END;
/
```

输出结果为：

output
hello world

- 例 2：声明变量，并赋值，打印变量

```
!set plsqlUseSlash true
BEGIN
   DECLARE v1 STRING;
   SET v1 = 'hello world';
   PUT_LINE(v1);
END;
/
```

输出结果为:

output
hello world

8.12.3.2. 声明

8.12.3.2.1. 基本变量的声明

STRING

- 语法:

```
DECLARE <variable_name> STRING [DEFAULT <value>];
```

例如:

```
DECLARE V1 STRING;
DECLARE V2 STRING DEFAULT 'hello world';
```

此处声明两个字符串类型的变量，并给变量 V2 赋初值为 hello world。

INT

- 语法:

```
DECLARE <variable_name> INT [DEFAULT <value>];
```

例如:

```
DECLARE V1 INT DEFAULT '100';
DECLARE V2 INT;
```

此处声明两个整数类型的变量，并给变量 V1 赋初值为 100。

Boolean

- 语法:

```
DECLARE <variable_name> Boolean [DEFAULT <value>];
```

```
DECLARE V1 Boolean;
DECLARE V2 Boolean DEFAULT true;
```

此处声明两个布尔值类型的变量，并给变量 V2 赋初值为 true。

DATE

- 语法:

```
DECLARE <variable_name> TIMESTAMP [DEFAULT TIMESTAMP <value>];
```

例如:

```
DECLARE date_a DATE;
DECLARE date_b DATE DEFAULT '2015-12-01';
```

此处声明两个日期类型的变量，并且给变量 date_b 赋初值为 2015-12-01。

TIMESTAMP

- 语法:

```
DECLARE <variable_name> TIMESTAMP [DEFAULT TIMESTAMP <value>];
```

例如:

```
DECLARE ts_c TIMESTAMP;
DECLARE ts_d TIMESTAMP DEFAULT TIMESTAMP '2011-05-05 08:09:23';
```

此处声明两个时间标记类型的变量，并且给变量 ts_d 赋初值为 2011-05-05 08:09:23。

STATEMENT

Inceptor 中，STATEMENT 变量常被用来存储事先构建好的 SQL 语句，更多关于 STATEMENT 变量的用法说明，可参见动态 SQL 章节。

- 语法:

```
DECLARE <variable_name> STATEMENT;
```

例如:

```
DECLARE V1 STATEMENT;
```

此处声明一个 STATEMENT 类型的变量 V1 。

- 对于 STATEMENT 变量，不能直接使用 SET 等赋值语句，必须使用关键字 PREPARE。
- 对于 STATEMENT 变量，也不能通过 PUT_LINE 等输出函数打印变量的值，更多用法说明，可参见动态 SQL 章节

8.12.3.2.2. 常量的声明

Inceptor 中可以使用关键字 CONSTANT，来声明任意基本数据类型的常量，以下将以 INT，STRING 类型为例，来演示如何声明一个基本类型的常量。

- **语法:**

```
DECLARE <variable_name> <data_type> CONSTANT <value>;
```

- 例 1：整数类型的常量

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个整数类型的常量 int_a，值为 1。
  DECLARE int_a INT CONSTANT 1;
  -- 打印出常量 int_a 的值。
  PUT_LINE(int_a);
END;
/
```

输出结果为:

output
1

- 例2:字符串类型的常量

```
!set plsqlUseSlash true
BEGIN
  DECLARE string_a STRING CONSTANT 'hello';
  PUT_LINE(string_a);
END;
/
```

输出结果为:

output
hello

8.12.3.2.3. 重复命名

在相同的作用域内，所有声明的标识符都必须是唯一的。即 Inceptor 中不可以在相同作用域内重复命名。即使数据类型不同，变量和参数的名字也不能相同。

- 例 1：Inceptor 中对于变量的重复命名，会报错

```
!set plsqlUseSlash true
BEGIN
 -- 声明一个字符串类型的变量 V1。
 DECLARE V1 STRING;
 -- 声明一个整数类型的变量 V1。
 DECLARE V1 INT;
 SET V1 = 'hello';
 SET V1 = 100;
 PUT_LINE(V1);
END;
/
```

输出结果为:

```
COMPILE FAILED: Semantic error: [Error 11108]
ANONYMOUS BLOCK (LINE 5, COLUMN 8, TEXT \"DECLARE V1 INT;\"): Existing variable v1
```

可以看到，Inceptor 会报错，也就是说 Inceptor 中不支持变量在同一作用域内的重复命名。

8.12.3.2.4. 大小写敏感性

Inceptor 中对于所有的标识符,如常量、变量、参数都是不区分大小写的。所以在命名时，Inceptor不支持通过改变变量名大小写的方式来命名多个不同的变量。

- 案例 1：Inceptor 中不区分变量的大小写形式

```
!set plsqlUseSlash true
BEGIN
 -- 声明一个整数类型的变量 v_price。
 DECLARE v_price INT;
 -- 声明一个 DOUBLE 类型的变量 V_PRICE。
 DECLARE V_PRICE DOUBLE;
 SET v_price = 100;
 SET V_PRICE = 99.99;
 PUT_LINE('price is:'||v_price);
 PUT_LINE('PRICE IS:'||V_PRICE);
END;
/
```

输出结果为:

```
COMPILE FAILED: Semantic error: [Error 11108]
ANONYMOUS BLOCK (LINE 5, COLUMN 8, TEXT \"DECLARE V_PRICE DOUBLE;\"): Existing variable v_price
```

可以看到这段语句块在第三行，再次声明一个名称相同，仅大小写形式不同的变量时，就出错了，Inceptor 会认为这是重复命名，会报错。

8.12.3.2.5. 嵌套命名

Inceptor 中在同一作用域内重复命名，是不合法的，但是 Inceptor 中支持在嵌套的语句块内重复命名，具体使用方法可参考以下示例。

- 例 1：在嵌套的语句块内可以重复命名

```
!set plsqlUseSlash true
BEGIN
 -- 分别声明两个 STRING，DOUBLE 类型的变量 V1，V2。
 DECLARE V1 STRING;
 DECLARE V2 DOUBLE;
 -- 分别给变量 V1，V2 赋值。
 SET V1 = 'hello';
 SET V2 = 11.11;
 -- 打印出变量 V1，V2 的值。
 PUT_LINE('the first v1:'||V1);
 PUT_LINE('the first v2:'||V2);
   -- 在嵌套的语句块内，再次声明两个相同名称及类型的变量 V1，V2。
   BEGIN
     DECLARE V1 STRING;
     DECLARE V2 DOUBLE;
     -- 给变量 V1，V2 赋值。
     SET V1 = 'world';
     SET V2 = 22.22;
     -- 打印出变量 V1，V2 的值。
     PUT_LINE('the second v1:'||V1);
     PUT_LINE('the second v2:'||V2);
  END;
END;
/
```

输出结果为：

output
the first v1:hello
the first v2:11.11
the second v1:world
the second v2:22.22

8.12.3.3. 赋值

在 Inceptor 中，我们可以使用关键字 SET，VALUES INTO，或者 SELECT INTO 对所声明的变量进行赋值。

8.12.3.3.1. SET

- 例1：简单赋值

```
!set plsqlUseSlash true
BEGIN
  -- 分别声明两个类型为整数型和字符串类型的变量 V1，V2。
  DECLARE V1 STRING;
  DECLARE V2 INT;
   -- 分别给变量 V1，V2 赋值。
   SET V1 = 'hello world';
   SET V2 = 1100;
  -- 分别打印出变量 V1，V2 的值。
  PUT_LINE('V1:'||V1);
  PUT_LINE('V2:'||V2);
END;
/
```

输出结果为：

output
V1:hello world
V2:1100

- 例 2：将查询结果赋值给变量

```
-- 查询 transactions 表中交易号为 162742112 的价格，并赋值给变量 V2。
!set plsqlUseSlash true
BEGIN
  DECLARE V1 STRING;
  DECLARE V2 DOUBLE;
   -- 查询 transactions 表中价格为 11.11 的交易时间，并赋值给变量 V1。
   SET V1 = (SELECT trans_time FROM transactions WHERE price = 11.11);
   SET V2 = (SELECT price FROM transactions WHERE trans_id=162742112);
  PUT_LINE('V1:'||V1);
  PUT_LINE('V2:'||V2);
END;
/
```

输出结果为：

output
V1:20140205140521
V2:12.21

8.12.3.3.2. VALUES INTO

- 例1：简单赋值

```
!set plsqlUseSlash true
BEGIN
  -- 分别声明两个数据类型为字符串和整数类型的变量 V1，V2。
  DECLARE V1 STRING;
  DECLARE V2 INT;
   -- 分别给两个变量 V1，V2 赋值。
   VALUES ('hello world') INTO V1;
   VALUES 1100 INTO V2;
  -- 分别打印出变量 V1，V2 的值。
  PUT_LINE('V1:'||V1);
  PUT_LINE('V2:'||V2);
END;
/
```

输出结果为：

output
V1:hello world
V2:1100

8.12.3.3.3. SELECT … INTO

- 例1：将查询结果赋值给变量

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 DOUBLE 类型的变量 v_price。
  DECLARE v_price DOUBLE;
  -- 声明一个字符串类型的变量 v_stock。
  DECLARE v_stock STRING;
  -- 查询表 transactions 中交易时间为 20140328102400 的价格，并赋值给变量 v_price。
  SELECT price INTO v_price FROM transactions WHERE trans_time=20140328102400;
  -- 查询表 transactions 中交易时间为 20140628133001 的股票交易流水号。
  SELECT stock_id INTO v_stock FROM transactions WHERE trans_time=20140628133001;
  -- 打印出变量 v_price 的值。
  PUT_LINE('price is:'||v_price);
  -- 打印出变量 v_stock 的值。
  PUT_LINE('stockid is:'||v_stock);
END;
/
```

输出结果为:

output
price is:68.43
stockid is:SH6277444

8.12.4. 数据类型

8.12.4.1. 内置类型

内置类型即标量类型，仅可存放单个的值，即可用作变量也可用作常量，无须用户自定义，常见的有 STRING，BOOLEAN，INT，DOUBLE，DATE，STATEMENT，TIMESTAMP 等数据类型，更多 Inceptor 中支持的标量类型，可参见《数据类型对应表》章节。

8.12.4.1.1. STRING

- 语法:

```
DECLARE <variable_name> STRING [DEFAULT <value>];
```

- 例 1：声明-赋值-打印

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个字符串类型的变量 V1。
  DECLARE V1 STRING;
  -- 给变量 V1 赋值为 'hello world'。
  SET V1 = 'hello world';
  -- 打印出变量 V1 的值。
  PUT_LINE('V1:'||V1);
END;
/
```

输出结果为:

output
V1:hello world

- 例 2：声明并赋值-打印

```
!set plsqlUseSlash true
BEGIN
  DECLARE V4 STRING DEFAULT 'hello world';
  PUT_LINE('V4:'||V4);
END;
/
```

输出结果为：

output
V4:hello world

8.12.4.1.2. INT

- 语法：

```
DECLARE <variable_name> INT [DEFAULT <value>];
```

- 例 1：声明-赋值-打印

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个字符串类型的变量 V1。
  DECLARE V1 STRING;
  -- 声明一个整数类型的变量 V2。
  DECLARE V2 INT;
  -- 给变量 V1 赋值为 'hello world'。
  SET V1 = 'hello world';
  -- 给变量 V2 赋值为 100。
  SET V2 = 100;
  -- 打印出变量 V1 的值。
  PUT_LINE('V1:'||V1);
  -- 打印出变量 V2 的值。
  PUT_LINE('V2:'||V2);
END;
/
```

输出结果为：

output
V1:hello world
V2:100

- 例 2：声明并赋值-打印

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个整数类型的变量 V3，并赋初值为 10。
  DECLARE V3 INT DEFAULT 10;
  -- 声明一个字符串类型的变量 V4，并赋初值为 hello world。
  DECLARE V4 STRING DEFAULT 'hello world';
  -- 打印出变量 V3 的值。
  PUT_LINE('V3:'||V3);
  -- 打印出变量 V4 的值。
  PUT_LINE('V4:'||V4);
END;
/
```

输出结果为：

output
V3:10
V4:hello world

8.12.4.1.3. Boolean

- 语法：

```
DECLARE <variable_name> Boolean [DEFAULT <value>];
```

- 例 1：声明-赋值-打印

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 Boolean 类型的变量 V1。
  DECLARE V1 Boolean;
  -- 给变量 V1 赋值为 true。
  SET V1 = true;
  -- 打印出变量 V1 的值。
  PUT_LINE('V1:'||V1);
END;
/
```

输出结果为：

output
V1:true

- 例 2：声明并赋值-打印

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 Boolean 类型的变量 V4，并赋初值为 false。
  DECLARE V4 Boolean DEFAULT false;
  -- 打印出变量 V4 的值。
  PUT_LINE('V4:'||V4);
END;
/
```

输出结果为：

output
V4:false

8.12.4.1.4. DATE

- 语法:

```
DECLARE <variable_name> TIMESTAMP [DEFAULT TIMESTAMP <value>];
```

- 案例 1:

```
!set plsqlUseSlash true
BEGIN
   -- 声明一个 DATE 类型的变量 date_a。
   DECLARE date_a DATE;
   -- 声明一个 DATE 类型的变量 date_b，并赋初值为 2015-12-01。
   DECLARE date_b DATE DEFAULT '2015-12-01';
   -- 给变量 date_a 赋值为 2015-10-10。
   SET date_a = DATE '2015-10-10';
   -- 打印出变量 date_a 的值。
   PUT_LINE('date_a is:'||date_a);
   -- 打印出变量 date_b 的值。
   PUT_LINE('date_b is:'||date_b);
END;
/
```

输出结果为:

output
date_a is:2015-10-10
date_b is:2015-12-01

8.12.4.1.5. TIMESTAMP

- 语法:

```
DECLARE <variable_name> TIMESTAMP [DEFAULT TIMESTAMP <value>];
```

- 例 1：声明-赋值-打印

```
!set plsqlUseSlash true
BEGIN
    -- 声明一个 TIMESTAMP 类型的变量 ts_d。
    DECLARE ts_d TIMESTAMP;
    -- 给变量 ts_d 赋值为 2011-05-05 08:09:23。
    SET ts_d = TIMESTAMP '2011-05-05 08:09:23';
    -- 打印出变量 ts_d 的值。
    PUT_LINE(ts_d);
END;
/
```

输出结果为:

output
2011-05-05 08:09:23

- 例 2：声明并赋值-打印

```
!set plsqlUseSlash true
BEGIN
    -- 声明一个 TIMESTAMP 类型的变量 ts_d，给变量 ts_d 赋值为 2011-05-05 08:09:23。
    DECLARE ts_d TIMESTAMP DEFAULT TIMESTAMP '2011-05-05 08:09:23';
    -- 打印出变量 ts_d 的值。
    PUT_LINE(ts_d);
END;
/
```

输出结果为：

output
2011-05-05 08:09:23

8.12.4.1.6. STATEMENT

Inceptor 中，STATEMENT 变量常被用来存储事先构建好的 SQL 语句，更多关于 STATEMENT 变量的用法说明，可参见动态 SQL 章节。

- 语法：

```
DECLARE <variable_name> STATEMENT;
```

- 例 1：声明-赋值

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 STATEMENT 类型的变量 V1。
  DECLARE V1 STATEMENT;
  -- 使用关键字 PREPARE 将 SQL 语句放进变量 V1 里。
  PREPARE V1 FROM 'SELECT price FROM transactions WHERE acc_num=6513065';
END;
/
```

- 对于 STATEMENT 变量，不能直接使用 SET 等赋值语句，必须使用关键字 PREPARE。
- 对于 STATEMENT 变量，也不能通过 PUT_LINE 等输出函数打印变量的值，更多用法说明，可参见动态 SQL 章节。

8.12.4.2. 复合类型

Inceptor 中支持 Row 和 Array 两种复合数据类型，更多用法说明，可参见行数据类型和 ARRAY 章节

8.12.4.3. 锚定类型

Inceptor 中支持 Anchored data type 锚定类型

8.12.4.4. 特殊类型

Inceptor 中支持 Statement，Condition，RESULT_SET_LOCATOR 三种特殊类型

8.12.5. 创建 SQL PL 语句块

- SQL PL 语句块由关键字 BEGIN 来声明语句块的开始，关键字 END 来声明语句块的结束，在 SQL PL 语句块中，可以进行变量声明与赋值，异常的声明与处理等操作。
- 与 PL/SQL 语句块的构成不同的是，变量的声明部分，要在执行体开始之后，即在关键字 BEGIN 之后，才可以进行变量，常量等的声明。否则 Inceptor会 报语义错误。

8.12.5.1. 创建示例

- 例 1：简单匿名块的创建，直接打印常量

```
!set plsqlUseSlash true
-- 语句块的开始，必要部分，不可缺省。
BEGIN
  -- 执行体部分，此处为使用 PUT_LINE 函数直接打印出字符串的内容。
  PUT_LINE('hello world');
-- 语句块的结束，必要部分，不可缺省。
END;
/
```

输出结果为：

output
hello world

- 例 2：稍复杂的匿名块，打印出变量

```
!set plsqlUseSlash true
-- 语句块的开始，必要部分，不可缺省。
BEGIN
 -- 变量等的声明，可选部分，语句块中，不需要声明变量的情况下，可以缺省。
 DECLARE
    -- 声明一个名为 v1 的变量，数据类型为字符串。
    v1 STRING;
    -- 使用关键字 SET，给变量 v1 赋值。
    SET v1 = 'hello world';
    -- 使用 PUT_LINE 函数打印出变量 v1 的值。
    PUT_LINE(v1);
-- 语句块的结束，必要部分，不可缺省。
END;
/
```

输出结果为：

output
hello world

8.12.6. 流程控制语句

8.12.6.1. IF

8.12.6.1.1. IF-THEN

- 语法：

```
IF condition THEN
  {...statements...}
END IF;
```

- 例 1：查询表 transactions 中交易号为 648230055 的价格

```
!set plsqlUseSlash true
BEGIN
    -- 分别声明两个 DOUBLE，STRING 类型的变量 v_price，v_comment。
    DECLARE v_price DOUBLE;
    DECLARE v_comment STRING;
   -- 查询 transactions 表中交易号为 648230055 的价格，并将查询的结果放进变量 v_price 里。
   SELECT price INTO v_price FROM transactions WHERE trans_id=648230055;
   -- 输出 v_price 的值。
   PUT_LINE('the price is:'||v_price);
   -- 确定 IF 的条件语句为 v_price 的值大于 20。
   IF v_price > 20 THEN
      -- 在满足 IF 的条件语句的情况下，给变量 v_comment 赋值。
      SET v_comment = 'the price is too high';
   END IF;
   -- 输出变量 v_comment 的值。
   PUT_LINE(v_comment);
END;
/
```

输出结果为：

output
the price is:22.66
the price is too high

8.12.6.1.2. IF-THEN-ELSE

- 语法：

```
IF condition THEN
  {...statements...}
ELSE
  {...statements...}
END IF;
```

- 例：查询 transactions 表中交易号为 943197522 的价格

```
!set plsqlUseSlash true
BEGIN
   -- 分别声明两个 DOUBLE，STRING 类型的变量 v_price，v_comment。
   DECLARE v_price DOUBLE;
   DECLARE v_comment STRING;
   -- 查询 transactions 表中交易号为 943197522 的价格，并将查询的结果放进变量 v_price 里。
   SELECT price
     INTO v_price
     FROM transactions
     WHERE trans_id=943197522;
   -- 输出 v_price 的值。
   PUT_LINE('the price is:'||v_price);
   -- 确定 IF 的条件语句为 v_price 的值大于 20。
   IF v_price > 20 THEN
     -- 在满足 IF 的条件语句的情况下，给变量 v_comment 赋值。
     SET v_comment = 'the price is too high';
   ELSE
     -- 在不满足 IF 的条件语句的情况下，给变量 v_comment 赋值。
     SET v_comment = 'the price is not too high';
   END IF;
  -- 输出变量 v_comment 的值。
  PUT_LINE(v_comment);
END;
/
```

输出结果为：

output
the price is:12.13
the price is not too high

8.12.6.1.3. IF-THEN-ELSIF

- 语法：

```
IF condition THEN
  {...statements...}
ELSIF condition THEN
  {...statements...}
ELSE
  {...statements...}
END IF;
```

- 例 1：查询 transactions 表中交易号为 943197522 的价格

```
!set plsqlUseSlash true
BEGIN
  -- 分别声明两个 DOUBLE，STRING 类型的变量 v_price，v_comment。
  DECLARE v_price DOUBLE;
  DECLARE v_comment STRING;
   -- 查询 transactions 表中交易号为 943197522 的价格，并将查询的结果放进变量 v_price 里。
   SELECT price
      INTO v_price
      FROM transactions
      WHERE trans_id=943197522;
    -- 输出 v_price 的值。
    put_line('the price is:'||v_price);
    -- 确定 IF 的条件语句为 v_price 的值大于 15。
    IF v_price > 15 THEN
       -- 在满足 IF 的条件语句的情况下，给变量 v_comment 赋值。
       SET v_comment = 'the price 大于 15';
    -- 确定第一个 ELSIF 的条件语句为 v_price 的值大于 10。
    ELSIF v_price > 10 then
       -- 在满足第一个 ELSIF 的条件语句的情况下，给变量 v_comment 赋值。
       SET v_comment = 'the price 大于10';
    -- 确定第二个 ELSIF 的条件语句为 v_price 的值大于 5。
    ELSIF v_price > 5 then
       -- 在满足第二个 ELSIF 的条件语句的情况下，给变量 v_comment 赋值。
       SET v_comment = 'the price 大于5';
    ELSE
       -- 在以上 IF 和 ELSIF 的条件语句都不满足的情况下，给变量 v_comment 赋值。
       SET v_comment = 'the price is too low';
    END IF;
   put_line(v_comment);
END;
/
```

输出结果为：

output
the price is:12.13
the price 大于10

- 例 2：查询 transactions 表中交易号为 209441379 的价格

```
!set plsqlUseSlash true
BEGIN
   DECLARE v_price DOUBLE;
   DECLARE v_comment STRING;
   SELECT price
     INTO v_price
     FROM transactions
     WHERE trans_id=209441379;
   put_line('the price is:'||v_price);
   IF v_price > 15 THEN
      SET v_comment = 'the price 大于 15';
   ELSIF v_price > 10 then
      SET v_comment = 'the price 大于10';
   ELSIF v_price > 5 then
      SET v_comment = 'the price 大于5';
   ELSE
      SET v_comment = 'the price is too low';
   END IF;
  put_line(v_comment);
END;
/
```

输出结果为：

output
the price is:4.5
the price is too low

8.12.6.2. FOR

- 语法:

```
FOR <loop-name> AS [<cursor-name> CURSOR [with hold] FOR ] <select-statement>
 DO
   <inside-loop-logic>
END FOR;
```

- 例 1:

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 DOUBLE 类型的变量 sum_price，默认值为 0。
  DECLARE sum_price double default 0;
    FOR v1 AS
    -- 查询表t ransactions 中交易时间为 20141225133500 的价格。
    SELECT price FROM transactions WHERE trans_time=20141225133500
      DO
      -- 将查询到的价格与变量 sum_price 相加的和赋值给变量 sum_price。
      SET sum_price = sum_price + v1.price;
  END FOR;
  -- 打印出变量 sum_price 的值。
  put_line('the price is:'||sum_price);
END;
/
```

输出结果为:

output
the price is:7.49

8.12.6.3. LOOP

8.12.6.3.1. the simple loop

- 语法:

```
<loop-name:>
LOOP
{...statements...}
[EXIT WHEN conditions]-- 条件满足,退出循环语句
END LOOP <loop-name>;
```

- 例 1:

```
!set plsqlUseSlash true
BEGIN
  -- 分别声明两个整数类型的变量，默认值为 0。
  DECLARE v_counter INT default 0;
  DECLARE v_sum INT default 0;
  fetch_loop:
  LOOP
    -- 当变量 v_counter 的值为 10 时，就退出循环。
    IF v_counter = 10 THEN
      leave fetch_loop;
    END IF;
  -- 将变量 v_sum 与变量 v_counter 相加的和赋值给变量 v_sum。
  SET v_sum = v_sum + v_counter;
  -- 将变量 v_counter 加 1 的值赋值给变量 v_counter。
  SET v_counter = v_counter + 1;
  END LOOP fetch_loop;
-- 打印出变量 v_sum 的值。
PUT_LINE('the sum is:'||v_sum);
END;
/
```

输出结果为：

output
the sum is:45

8.12.6.4. WHILE

- 语法：

```
WHILE conditions
DO
{...statements...}
END WHILE;
```

- 例 1：

```
!set plsqlUseSlash true
BEGIN
  -- 分别声明两个整数类型的变量，默认值为 0。
  DECLARE v_counter INT default 0;
  DECLARE v_sum INT default 0;
  -- 在变量 v_counter 小于 10 的情况下。
  WHILE v_counter < 10 DO
    -- 将变量 v_sum 与变量 v_counter 相加的和赋值给变量 v_sum。
    SET v_sum = v_sum + v_counter;
    -- 将变量 v_counter 加 1 的值赋值给变量 v_counter。
    SET v_counter = v_counter + 1;
  END WHILE;
-- 打印出变量 v_sum 的值。
PUT_LINE('the sum is:'||v_sum);
END;
/
```

输出结果为：

output
the sum is:45

8.12.6.5. REPEAT

- 语法:

```
<loop-name:>
REPEAT
{...statements...}
[UNTIL conditions]-- 条件满足,退出循环语句
END REPEAT <loop-name>;
```

- 例1:

```
!set plsqlUseSlash true
BEGIN
  -- 分别声明两个整数类型的变量，默认值为 0。
  DECLARE v_counter INT default 0;
  DECLARE v_sum INT default 0;
  fetch_loop:
  REPEAT
    -- 将变量 v_sum 与变量 v_counter 相加的和赋值给变量 v_sum。
    SET v_sum = v_sum + v_counter;
    -- 将变量 v_counter 加 1 的值赋值给变量 v_counter。
    SET v_counter = v_counter + 1;
    -- 直到变量 v_counter 的值大于 15。
    UNTIL v_counter > 15
  END REPEAT fetch_loop;
-- 打印出变量 v_sum 的值。
PUT_LINE('the sum is :'||v_sum);
END;
/
```

输出结果为:

output
the sum is :120

8.12.6.6. GOTO

GOTO 语句是指无条件跳转到指定的标签去的意思，当执行 GOTO 语句的时候,控制会立即转到由标签标识的语句。在实际的应用中，频繁地使用 GOTO 语句，会使得代码可读性变差，应当尽可能地避免使用 GOTO 语句。

- 例 1:

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个整数类型的变量 var，并赋初值为 1。
  DECLARE var INT default 1;
  -- 设置一个标签为print。
  print:
    -- 标签的内容为:如果 var 小于 5，就给变量var不断加上 1，并赋值给 var。
    IF var < 5 then
      SET var = var + 1;
      -- 跳转到标签 print。
      GOTO print;
    END IF;
   -- 打印出变量 var 的值。
   PUT_LINE('var:'||var);
END;
/
```

输出结果为:

output
var:5

- 例2:

```
!set plsqlUseSlash true
-- 创建一个名为 num_emp 的过程，参数 v_num 的类型为 IN，数据类型为 STRING；参数 num_of_emp 的类型为
OUT，数据类型为整数类型。
CREATE OR REPLACE PROCEDURE num_emp(IN v_num STRING, OUT num_of_emp INT)
LANGUAGE SQL
BEGIN
  -- 如果 v_num 的值为 6513064，就给变量 num_of_emp 赋值为 -1。
  IF v_num='6513065'
   THEN
     SET num_of_emp = -1;
     -- 跳转到标签 putline_num。
     GOTO putline_num;
  END IF;
  -- 查找表 transactions 中账号为 v_num 的行数，并将查询结果赋值给变量 num_of_emp。
  SELECT COUNT(*) INTO num_of_emp
  FROM transactions WHERE acc_num = v_num;
  -- 设置一个标签为 putline_num。
  putline_num:
 -- 标签内的内容为打印出变量 num_of_emp 的值。
 PUT_LINE('COUNT:'||num_of_emp);
END;
/
```

- 调用过程:

```
!set plsqlUseSlash true
BEGIN
 -- 声明一个整数类型的变量 V1。
 DECLARE V1 INT;
 -- 调用过程 num_emp，参数 v_num 的实参为 6513065。
 CALL num_emp('6513065',V1);
END;
/
```

输出结果为:

output
COUNT:-1

8.12.6.7. LEAVE

- 例 1:

```
!set plsqlUseSlash true
BEGIN
  -- 分别声明两个个整数类型的变量，其中给变量 v_sum 赋初值为 0，给变量 v_outer_index 赋值为 1。
  DECLARE v_outer_index INT;
  DECLARE v_sum INT DEFAULT 0;
  SET v_outer_index = 1;
  outer_loop:
   LOOP
    -- 如果 v_outer_index 的值为 5，就离开 outer_loop 循环。
    IF v_outer_index = 5 THEN
      leave outer_loop;
    END IF;
  -- outer_loop 循环内的内容为：将 v_sum 与 v_inner_index 的和赋给 v_sum，并且 v_inner_index 的值不断加
1。
  SET v_sum = v_sum + v_outer_index;
  SET v_outer_index = v_outer_index + 1;
  END LOOP outer_loop;
  -- 打印出在 outer_loop 循环内，变量 v_sum 的值。
  PUT_LINE('the outer sum:'||v_sum);
END;
/
```

输出结果为：

output
the outer sum:10

8.12.6.8. ITERATE

ITERATE 语句可以在 LOOP 循环中构成一个新的迭代，从而起到更早跳出循环的效果。

- 例1：

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE myproc(OUT var01 INT,OUT var02 INT)
BEGIN
    SET var01 = 0;
    SET var02 = 0;
      label2: LOOP
         SET var01 = var01 + 1;
         IF ( var01 > 100 or var02 > 100 ) THEN
            leave label2;
         END IF;
         IF var02 > 50 THEN
           iterate label2;
          END if;
      SET var02 = var02 +1;
    END LOOP;
    PUT_LINE('var01:'||var01);
    PUT_LINE('var02:'||var02);
END;
/
```

- 调用过程

```
!set plsqlUseSlash true
BEGIN
   DECLARE v1 INT;
   DECLARE v2 INT;
   CALL myproc(v1,v2);
END;
/
```

输出结果为：

output
var01:101
var02:51

8.12.6.9. CASE

- 例 1:

```
!set plsqlUseSlash true
-- 创建一个名为 test 的过程，参数的数据类型均为 INT，其中参数c的类型为 OUT，参数 d 的类型为 IN。
CREATE OR REPLACE PROCEDURE test()
BEGIN
  -- 分别声明两个整数类型的变量 a, b，其中变量 b的默认值为 9。
  DECLARE a INT;
  DECLARE b INT default 9;
  -- 当 b 的值为 7 时，将 0 赋值给变量 a。
  SET a = CASE b WHEN 7 THEN 0
  -- 当 b 的值为 8 时，将 1 赋值给变量 a。
                 WHEN 8 THEN 1
  -- 其它情况下，将2赋值给变量a。
                 ELSE 2
                 END CASE;
  -- 打印出此时变量 a 的值。
  put_line('case1:'||a);
  -- 第二种情况下，当 b 的值大于 9，则将 10 赋值给 a；当 b 的值小于 9，将 11 赋值给 a，当 b 的值为 9，将 12
赋值给 a。
  SET a = CASE WHEN b > 9 THEN 10
  -- 打印出第二种情况下，a的值。
               WHEN b < 9 THEN 11
  -- 调用存储过程 test，第二个形参的实参为 10；此时在第一种情况下 a 的值为 2，第二种情况下 a 的值为 12。
               ELSE 12
               END;
  PUT_LINE('case2:'||a);
END;
/
BEGIN
 test();
END;
/
```

输出结果为:

output
case1:2
case2:12

8.12.7. 存储过程

8.12.7.1. 语法

- 创建过程

```
CREATE [OR REPLACE] PROCEDURE <procedure_name> ([ {optional parameters} ])
 [{optional procedure attributes}]
 BEGIN [atomic]
   ... statements ...
END;
```

- 调用过程

存储过程的调用可以使用关键字CALL。

```
CALL procedure_name();
```

也可以在语句块之间直接调用。

```
BEGIN
  procedure_name();
END;
```

8.12.7.2. 参数的选择

Inceptor 中，支持在创建 SQL PL 存储过程时，不带参数，或者引入 IN，OUT，INOUT 等参数类型。

- IN 类型：默认模式，在调用 procedure 的时候，procedure 的实参的值被传递到该 procedure；
- OUT 类型：在调用 procedure 的时候，不能传递给 procedure 实参的值，在 procedure 的内部，procedure 的形参是只可写的；
- INOUT 类型：在调用 procedure 的时候，实参的值可以被传递给该 procedure，在其内部，形参可以被读出也可以被写入，该 procedure 结束时，形参的内容将赋给调用时的实参。

8.12.7.2.1. 不带参数

例 1：不带参数

- 创建存储过程 proc_1

```
!set plsqlUseSlash true
-- 创建不带参数的存储过程 proc_1。
CREATE OR REPLACE PROCEDURE proc_1()
BEGIN
    -- 打印出字符串 hello world。
    PUT_LINE('hello world');
END;
/
```

- 调用存储过程 proc_1。

```
!set plsqlUseSlash true
CALL proc_1();
```

输出结果为：

output
hello world

8.12.7.2.2. IN 参数

例 2：IN 参数

- 创建存储过程 proc_in。

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE proc_in(IN var1 STRING)
BEGIN
    PUT_LINE('var1:'||var1);
END;
/
```

- 调用存储过程 proc_in。

```
!set plsqlUseSlash true
CALL proc_in('hello world');
```

输出结果为：

output
var1:hello world

8.12.7.2.3. OUT 参数

例 3：OUT 参数

- 创建存储过程 proc_out。

```
!set plsqlUseSlash true
-- 创建带有参数的存储过程 proc_out，形参为 var1，其参数类型为 OUT，其数据类型为 STRING。
CREATE OR REPLACE PROCEDURE proc_out(OUT var2 STRING)
BEGIN
    -- 给变量 var2 赋值。
    SET var2 = 'hello world';
END;
/
```

- 调用存储过程 proc_out。

```
!set plsqlUseSlash true
BEGIN
  -- 声明变量 a，数据类型为 STRING。
  DECLARE a STRING;
  -- 调用过程 proc_out，实参为 a。
  proc_out(a);
  -- 打印出变量 a 的值。
  PUT_LINE('a:'||a);
END;
/
```

输出结果为：

output
a:hello world

例 4：OUT 参数

- 创建存储过程 proc_out_2。

```
!set plsqlUseSlash true
-- 创建带有参数的存储过程 proc_out_2，形参为 var1，其参数类型为 OUT，其数据类型为 INT。
CREATE OR REPLACE PROCEDURE proc_out_2(OUT var2 INT)
BEGIN
    -- 声明一个整数类型的变量 a，默认值为 2。
    DECLARE a INT DEFAULT 2;
    -- 将变量 a 的值赋给变量 var2。
    SET var2 = a;
END;
/
```

- 调用存储过程 proc_out_2。

```
!set plsqlUseSlash true
BEGIN
  -- 声明变量 b，数据类型为整数型。
  DECLARE b INT;
  -- 调用过程 proc_out_2，实参为 b。
  CALL proc_out_2(b);
  -- 打印出变量 b 的值。
  PUT_LINE('b:'||b);
END;
/
```

输出结果为：

output
b:2

例 5: IN,OUT 参数

- 创建存储过程 proc_out_3。

```
!set plsqlUseSlash true
-- 创建名为 proc_out_3 的过程，形参 var1 的参数类型为 IN，数据类型为 INT，形参 var2 的参数类型为
OUT，数据类型为 INT。
CREATE OR REPLACE PROCEDURE proc_out_3(IN var1 INT,OUT var2 INT)
BEGIN
    -- 声明一个整数类型的变量，默认值为 2。
    DECLARE a INT DEFAULT 2;
    -- 将变量 var1 和 a 的和赋给变量 var2。
    SET var2 = var1 + a;
    -- 打印出变量 var 的值。
    PUT_LINE('var2:'||var2);
END;
/
```

- 调用过程 proc_out_3

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个整数类型的变量 c，默认值为 10。
  DECLARE c INT DEFAULT 10;
  -- 声明一个整数类型的变量 d。
  DECLARE d INT;
  -- 调用过程 proc_out_3，实参分别为 c 和 d。
  proc_out_3(c,d);
END;
/
```

输出结果为：

output
var2:12

8.12.7.2.4. INOUT 参数

例 6：INOUT 参数与 OUT 参数

INOUT 参数具有和 OUT 参数相同的作用，即可以在调用存储过程时，读出参数的值。

- 创建存储过程 proc_inout

```
!set plsqlUseSlash true
-- 创建名为 proc_inout 的过程，形参 var1 的参数类型为 INOUT，数据类型为 INT，形参 var2 的参数类型为
INOUT，数据类型为 STRING。
CREATE OR REPLACE PROCEDURE proc_inout(INOUT var1 INT, INOUT var2 STRING)
BEGIN
    -- 声明一个整数类型的变量 a，默认值为 2。
    DECLARE a INT default 2;
    -- 将变量 a 的值赋给变量 var1。
    SET var1 = a;
    -- 给变量 var2 赋值。
    set var2 = 'hello world';
END;
/
```

- 调用过程 proc_inout

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个整数类型的变量 c。
  DECLARE c INT;
  -- 声明一个字符串类型的变量 d。
  DECLARE d string;
  -- 调用过程 proc_inout，实参分别为 c，d。
  CALL proc_inout(c,d);
  -- 打印出变量 c 的值。
  PUT_LINE('var1:'||c);
  -- 打印出变量 d 的值。
  PUT_LINE('var2:'||d);
END;
/
```

输出结果为：

output
var1:2
var2:hello world

例 7：INOUT 参数与 IN 参数

INOUT 参数具有和 IN 参数相同的作用，即可以在调用存储过程时，写入参数的值。

- 创建存储过程 proc_inout2

```
!set plsqlUseSlash true
-- 创建名为 proc_inout2 的过程，形参 var1 的参数类型为 INOUT，数据类型为 INT，形参 var2的参数类型为
INOUT，数据类型为 STRING。
CREATE OR REPLACE PROCEDURE proc_inout2(INOUT var1 INT, INOUT var2 STRING)
BEGIN
    -- 打印出变量 var1 的值。
    PUT_LINE('var1:'||var1);
    -- 打印出变量 var2 的值。
    PUT_LINE('var2:'||var2);
END;
/
```

- 调用过程 proc_inout2

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个整数类型的变量 c。
  DECLARE c INT;
  -- 声明一个字符串类型的变量 d。
  DECLARE d string;
  -- 给变量 c 赋值。
  SET c = 100;
  -- 给变量 d 赋值。
  SET d = 'hello world';
  -- 调用过程 proc_inout2，实参分别为 c 和 d。
  CALL proc_inout2(c,d);
END;
/
```

输出结果为：

output
var1:100
var2:hello world

例 8：INOUT 参数综合运用

在实际的使用中，INOUT 参数既可以被用来读出参数的值，也可以被用来写入参数的值。

- 创建存储过程 proc_inout3

```
!set plsqlUseSlash true
-- 创建名为 proc_inout3 的过程，形参 var1 的参数类型为 INOUT，数据类型为 INT，形参 var2 的参数类型为
INOUT，数据类型为 STRING。
CREATE OR REPLACE PROCEDURE proc_inout3(INOUT var1 INT, INOUT var2 STRING)
BEGIN
    -- 给变量 var1 赋值。
    SET var1=100;
    -- 打印出变量 var2 的值。
    PUT_LINE('var2:'||var2);
END;
/
```

- 调用存储过程 proc_inout3

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个整数类型的变量 c。
  DECLARE c INT;
  -- 声明一个字符串类型的变量 d。
  DECLARE d string;
  -- 给变量 d 赋值。
  SET d = 'hello world';
  -- 调用过程 proc_inout3，实参分别为c和d。
  CALL proc_inout3(c,d);
   -- 打印出变量 c 的值。
   PUT_LINE('var1:'||c);
END;
/
```

输出结果为：

output
var2:hello world
var1:100

8.12.7.2.5. 不指定参数类型

创建存储过程时，在不指定形参的参数类型的情况下，该形参的参数类型默认为 IN 类型。

例 9：不指定参数类型

- 创建存储过程 proc_var

```
!set plsqlUseSlash true
-- 创建带有参数的存储过程 proc_var，形参为 var1，不指定参数类型，数据类型为 STRING。
CREATE OR REPLACE PROCEDURE proc_var(var1 STRING)
BEGIN
    -- 打印出变量 var1 的值。
    PUT_LINE('var1:'||var1);
END;
/
```

- 调用存储过程 proc_var

```
!set plsqlUseSlash true
CALL proc_var('hello world');
```

输出结果为：

output
var1:hello world

8.12.7.3. 属性

在 Inceptor 中创建 SQL PL 存储过程时，可以为存储过程设置一些属性，Inceptor 支持以下几种属性：

- language-sql,LANGUAGE SQL：指明存储过程的 SQL PL 环境。
- specific,SPECIFIC：为过程指定一个特定的名称。
- result-set,DYNAMIC RESULT SETS：返回结果集个数。

- allow-sql,allowed-SQL：指定存储过程中所使用 SQL 语句的类型，有 3 种参数的值。
- deterministic,DETERMINISTIC or NOT DETERMINISTIC： 指定存储过程是动态的还是非动态的。
- call-on-null,CALLED ON NULL INPUT：不管任何输入的参数是否为 NULL 值，都可以调用存储过程。
- commit-on-return,COMMIT ON RETURN：指定存储过程运行时，是否提交 COMMIT，有两个参数的值。
- external-action,EXTERNAL ACTION or NO EXTERNAL ACTION：指定存储过程，是否执行一些改变数据库状态的活动。
- inherit-special-registers,INHERIT SPECIAL REGISTERS：存储过程中，特殊寄存器的返回值状态。
- savepoint,SAVEPOINT LEVEL：指定存储过程中，是否建立一个新的保存点。
- parameter-ccsid,PARAMETER CCSID：指定存储过程中，写入和读出的字符串数据的编码模式。

8.12.7.4. 过程重载

- Inceptor 中支持过程的重载，即子程序的名字相同，但是参数的类型或数目不同（参数的名称可以相同），返回值也可能不同。如果完全相同，就不是重载了，而是重复定义，这在 Inceptor 中是不允许的。
- 在实际的执行过程中，Inceptor 会根据实参的数据类型，来决定应该执行的过程。

- 例 1：重载过程 overload_test

```
!set plsqlUseSlash true
-- 创建第一个名为 overload_test 的过程，形参名为 place_in，其数据类型为字符串。
CREATE OR REPLACE PROCEDURE overload_test (IN place_in STRING)
BEGIN
  DECLARE l_message STRING;
  SET l_message = 'Hello ' || place_in;
  PUT_LINE(l_message);
END;
/
-- 创建第二个名为 overload_test 的过程，形参名为 v_name，其数据类型为整数类型。
CREATE OR REPLACE PROCEDURE overload_test(IN v_name INT)
BEGIN
  DECLARE l_message STRING;
  SET l_message = v_name||' is a good number';
  PUT_LINE(l_message);
END;
/
BEGIN
-- 过程内的实参为 world，类型为字符串，则会执行第一个过程。
overload_test ('world');
-- 过程内的实参为 888，类型为整数类型，则会执行第二个过程。
overload_test(888);
-- 过程内的实参为 999，类型为字符串，则会执行第一个过程。
overload_test('999');
END;
/
```

输出结果为：

output
Hello world
888 is a good number
Hello 999

8.12.8. SQLPL 函数

8.12.8.1. 创建函数

- 语法:

```
CREATE [OR REPLACE] FUNCTION  <function_name> ([<parameters>]) [{optional function attributes}]
RETURNS <datatype>
```

- 参数说明

  <function_name>：函数名称。

  <datatype>：数据类型。

- 例如:

例 297. 创建一个名为 sqlpl_function 的函数

此处创建一个名为 sqlpl_function 的函数，参数 v_time 的类型为 IN，数据类型为 STRING，该函数的返回值的数据类型为 STRING。

```
CREATE OR REPLACE FUNCTION sqlpl_function(IN v_time STRING)
RETURNS STRING
BEGIN
  RETURN v_time;
END;
/
```

8.12.8.2. 查看函数

语法

```
DESC PLSQL FUNCTION [EXTENDED] <function_name>; --[1]
```

参数说明

- <function_name>：函数名称。
- 可选项 [1]：可以选择添加关键字 EXTENDED 列出该 PL/SQL 函数的原文详细信息。

实例

例 298. 查看名为 sqlpl_function 的函数

```
desc plsql function sqlpl_function;
```

结果如下：

```
+-----------------------------------------------------------------------+
|                              description                              |
+-----------------------------------------------------------------------+
| Prototype:                                                            |
| FUNCTION db_rachel.sqlpl_function(v_time IN  STRING) RETURN STRING  |
+-----------------------------------------------------------------------+
```

8.12.8.3. 参数选择

Inceptor 中，创建 SQL PL 函数同样支持 IN，OUT，INOUT 三种参数类型，其中IN为默认情况下的参数类型，可根据具体的创建实例，选择合适的参数类型。

- IN 类型：默认模式，在调用 procedure 的时候，procedure 的实参的值被传递到该 procedure，在 procedure 的内部，procedure 的形参是只读的；
- OUT 类型：在调用 procedure 的时候，不能传递给 procedure 实参的值，在 procedure 的内部，procedure 的形参是只可写的；
- INOUT 类型：在调用 procedure 的时候，实参的值可以被传递给该 procedure，在其内部，形参可以被读出也可以被写入，该 procedure 结束时，形参的内容将赋给调用时的实参。

8.12.8.3.1. IN

- 例 1：参数类型为 IN 时，可以向过程内写入数据

```
!set plsqlUseSlash true
-- 创建函数 test_function，函数的返回值的数据类型为 STRING，参数 v_price 的类型为 IN，数据类型为 DOUBLE。
CREATE OR REPLACE FUNCTION test_function(IN v_price DOUBLE)
RETURNS STRING
BEGIN
  DECLARE v_time STRING;
  -- 查询表 transactions 中与参数 v_price 值相等的价格所对应的交易时间，并将查询到的结果放进变量 v_time 里。
  SELECT trans_time INTO v_time FROM transactions WHERE price=v_price;
  -- 函数 test_function 的返回值为 v_time。
  RETURN v_time;
END;
/
BEGIN
  DECLARE V1 STRING;
  -- 将实参为 12.13 的函数返回值赋值给变量 V1。
  SET V1 = test_function(12.13);
  PUT_LINE('V1:'||V1);
END;
/
```

输出结果为：

output
V1:20140105100520

8.12.8.3.2. OUT

- 例 2：参数类型为 OUT 类型时，可以从过程内读出数据

```
!set plsqlUseSlash true
-- 创建函数 test_function，函数的返回值的数据类型为 STRING，参数 v_price 的类型为 IN，数据类型为 DOUBLE。
CREATE OR REPLACE FUNCTION test_function(IN v_price DOUBLE)
RETURNS STRING
BEGIN
  DECLARE v_time STRING;
  -- 查询表 transactions 中与参数 v_price 值相等的价格所对应的交易时间，并将查询到的结果放进变量 v_time 里。
  SELECT trans_time INTO v_time FROM transactions WHERE price=v_price;
  -- 函数 test_function 的返回值为 v_time。
  RETURN v_time;
END;
/
BEGIN
  DECLARE V1 STRING;
  -- 将实参为 12.13 的函数返回值赋值给变量 V1。
  SET V1 = test_function(12.13);
  PUT_LINE('V1:'||V1);
END;
/
```

输出结果为：

output
var2:11.11

8.12.8.3.3. INOUT

- 例 3：INOUT 参数类型可以向过程内写入数据

```
!set plsqlUseSlash true
CREATE OR REPLACE FUNCTION test_function(INOUT v_price DOUBLE)
RETURNS STRING
BEGIN
  DECLARE v_time STRING;
  SELECT trans_time INTO v_time FROM transactions WHERE price=v_price;
  RETURN v_time;
END;
/
BEGIN
  DECLARE V1 STRING;
  DECLARE v_price DOUBLE default 12.13;
  SET V1 = test_function(v_price);
  PUT_LINE('V1:'||V1);
END;
/
```

输出结果为：

output
V1:20140105100520

- INOUT 参数类型向过程内写入数据时，与参数类型 IN 的作用相同。
- 为更好的对比 INOUT 和 IN，此处案例和案例 1 中除了参数类型不同，其它大致类似，主要差别是在调用函数时，需要先声明一个变量，并赋初值，在去调用函数。
- 即案例 1 中 test_function(12.13) 和 案例 4 中 test_function(v_price)，（其中 v_price 的值为 12.13）的区别。

- 例 4：INOUT 参数类型可以从过程内读出数据

```
!set plsqlUseSlash true
CREATE OR REPLACE FUNCTION test_function(INOUT var1 DOUBLE)
RETURNS DOUBLE
BEGIN
    DECLARE a DOUBLE default 11.11;
    SET var1 = a;
    RETURN var1;
END;
/
BEGIN
  DECLARE var2 DOUBLE;
  DECLARE V1 DOUBLE;
  SET var2 = test_function(V1);
  PUT_LINE('var2:'||var2);
END;
/
```

输出结果为:

output
var2:11.11

- INOUT 参数类型从过程内读出数据时，与参数类型 OUT 的作用相同。
- 为更好的对比 INOUT 和 OUT 类型的区别，此处案例与案例 2 中除了参数类型不同之外，其它完全相同。

8.12.8.4. 函数属性

Inceptor 中，在创建存储过程时，可以为相应的存储过程设置 11 种不同的属性，在创建函数的过程中，这些属性的设置同样有效。
需要注意的是，函数类似于一个有返回值的存储过程，所以对于存储过程适用的 11 种属性，函数只支持其中的 10 种，在创建函数的过程中，不支持设置属性：DYNAMIC RESULT SETS（返回结果集的个数）。

- LANGUAGE SQL：指明存储过程的 SQL PL 环境。
- SPECIFIC：为过程指定一个特定的名称。
- DYNAMIC RESULT SETS：返回结果集个数。
- allowed-SQL：指定存储过程中所使用SQL语句的类型，有 3 种参数的值。
- DETERMINISTIC or NOT DETERMINISTIC： 指定存储过程是动态的还是非动态的。
- CALLED ON NULL INPUT：不管任何输入的参数是否为 NULL 值，都可以调用存储过程。
- COMMIT ON RETURN：指定存储过程运行时，是否提交 COMMIT，有两个参数的值。
- EXTERNAL ACTION or NO EXTERNAL ACTION：指定存储过程，是否执行一些改变数据库状态的活动。
- INHERIT SPECIAL REGISTERS：存储过程中，特殊寄存器的返回值状态。
- SAVEPOINT LEVEL：指定存储过程中，是否建立一个新的保存点。
- PARAMETER CCSID：指定存储过程中，写入和读出的字符串数据的编码模式。

关于创建函数的过程中，属性的设置问题，可参见存储过程章节中的属性部分，在此不再赘述。

8.12.8.5. 调用函数

8.12.8.5.1. 单独调用函数

- 创建 SQL PL 函数 sqlpl_function

```
!set plsqlUseSlash true
-- 创建函数sqlpl_function，参数v_price的类型为IN，数据类型为DOUBLE。
CREATE OR REPLACE FUNCTION sqlpl_function(IN v_price DOUBLE)
-- 函数返回值的数据类型为字符串。
RETURNS STRING
BEGIN
  -- 声明一个字符串类型的变量v_time。
  DECLARE v_time STRING;
  -- 查询表transactions中与参数v_price相等的价格所对应的交易时间，并放进变量v_time里。
  SELECT trans_time INTO v_time FROM transactions WHERE price=v_price;
  -- 返回变量v_time的值。
  RETURN v_time;
END;
/
```

- 单独调用函数 sqlpl_function

```
!set plsqlUseSlash true
BEGIN
   DECLARE v1 STRING;
   -- 将函数的返回值赋值给变量 v1，其中函数 sqlpl_function 的实参为 11.11。
   SET v1 = sqlpl_function(11.11);
   PUT_LINE('V1:'||v1);
END;
/
```

输出函数为：

output
V1:20140205140521

8.12.8.5.2. 在存储过程内调用函数

- 创建 SQL PL 函数 sqlpl_function

```
!set plsqlUseSlash true
-- 创建函数sqlpl_function，参数v_price的类型为IN，数据类型为DOUBLE。
CREATE OR REPLACE FUNCTION sqlpl_function(IN v_price DOUBLE)
-- 函数返回值的数据类型为字符串。
RETURNS STRING
BEGIN
  -- 声明一个字符串类型的变量v_time。
  DECLARE v_time STRING;
  -- 查询表transactions中与参数v_price相等的价格所对应的交易时间，并放进变量v_time里。
  SELECT trans_time INTO v_time FROM transactions WHERE price=v_price;
  -- 返回变量v_time的值。
  RETURN v_time;
END;
/
```

- 在存储过程 test_procedure 内调用函数 sqlpl_function

```
!set plsqlUseSlash true
-- 创建存储过程test_procedure，参数v_amount的参数类型为IN，数据类型为DOUBLE。
CREATE OR REPLACE PROCEDURE test_procedure(IN v_amount DOUBLE)
BEGIN
  -- 分别声明两个字符串类型的变量v_id,v_num。
  DECLARE v_id STRING;
  DECLARE v_num STRING;
  -- 查询表transactions中与参数v_amount值相等的amount所对应的交易号，并将查询结果放进变量v_id中。
  SELECT trans_id INTO v_id FROM transactions WHERE amount=v_amount;
  -- 查询表transactions中交易时间和函数sqlpl_function(11.11)
的返回值相等，所对应的账号，并将查询结果放进变量v_num中。
  SELECT acc_num INTO v_num FROM transactions WHERE trans_time=sqlpl_function(11.11);
  PUT_LINE('the id is:'||v_id);
  PUT_LINE('the accnum is:'||v_num);
END;
/
BEGIN
 -- 调用过程test_procedure，实参的值为900。
 test_procedure(900.0);
END;
/
```

输出结果为：

output
the id is:900192386
the accnum is:3912384

8.12.8.5.3. SQL 语句中调用函数

- 创建 SQL PL 函数 sqlpl_function

```
!set plsqlUseSlash true
-- 创建函数sqlpl_function，参数v_price的类型为IN，数据类型为DOUBLE。
CREATE OR REPLACE FUNCTION sqlpl_function(IN v_price DOUBLE)
-- 函数返回值的数据类型为字符串。
RETURNS DOUBLE
BEGIN
  -- 返回变量v_price的值。
  RETURN v_price;
END;
/
```

- 标准 SQL 语句中调用函数

```
!set plsqlUseSlash true
SELECT * FROM transactions
WHERE price=sqlpl_function(12.13);
```

输出结果为：

trans_id	acc_num	trans_time	trans_type	stock_id	price	amount
943197522	6513065	20140105100520	b	AA7105670	12.13	200.0

8.12.8.5.4. SQLPL 函数中对其他函数的调用

- 创建函数 test1

```
!set plsqlUseSlash true
CREATE OR REPLACE FUNCTION test1( IN t_id DOUBLE)
returns DOUBLE
BEGIN
   return t_id;
END;
/
```

- 创建函数 test2，并且在函数 test2 中调用函数 test1

```
!set plsqlUseSlash true
CREATE OR REPLACE FUNCTION test2(IN v_price DOUBLE)
RETURNS DOUBLE
BEGIN
   SET v_price = v_price + test1('11.11');
    RETURN v_price;
END;
/
```

- 调用函数

```
!set plsqlUseSlash true
BEGIN
  DECLARE V1 DOUBLE;
  SET V1 = test2(1.11);
  PUT_LINE('V1:'||V1);
END;
/
```

输出结果为:

output
V1:12.219999999999999

除此之外，标准 SQL 调用 SQL PL 函数还须满足三个条件，具体内容，可参考以下章节。

8.12.8.6. 标准 SQL 调用 SQLPL 函数必须满足的条件

8.12.8.6.1. 必须是函数，不能是过程

Inceptor 中支持 SQL 语句中直接调用 SQLPL 函数，但是不可以调用 SQLPL 的过程。

- 创建一个名为 sqlpl_function 的 SQLPL 函数

```
!set plsqlUseSlash true
CREATE OR REPLACE FUNCTION sqlpl_function(IN v_num STRING)
RETURNS STRING
BEGIN
  RETURN v_num;
END;
/
```

- 在 SQL 语句中调用刚刚创建的 sqlpl_function 函数

```
!set plsqlUseSlash true
SELECT price FROM transactions
WHERE acc_num=sqlpl_function('6513065');
```

输出结果为:

price
12.13
6.12
9.81
7.49
7.02
18.38

可以看到 Inceptor 中可以返回正确的值，也就是说 SQL 语句中可以直接调用 SQLPL 函数，但是不可以直接调用 SQLPL 过程。

> 要查看 SELECT 查询的结果，首先需要设置 plsql.show.sqlresults 的属性为 true。

8.12.8.6.2. SQLPL 函数返回值必须是基本类型

在创建 SQLPL 函数的时候，会定义一个返回值的数据类型，在 Inceptor 中要求该数据类型必须为基本类型，也就是一些标量类型区别于复合类型等。标量类型仅存放单个的值，常见的基本类型为 int/double/string 等。对于返回值为非基本类型的数据类型，Inceptor 会报错。

SQLPL 函数返回值必须是基本类型

- 创建一个名为 sqlpl_function 的 SQLPL 函数，函数返回值是 string 类型。

```
!set plsqlUseSlash true
CREATE OR REPLACE FUNCTION sqlpl_function(t_name string)
  returns string
BEGIN
  return t_name;
END;
/
```

- 在 SQL 语句中调用刚刚创建的 sqlpl_function 函数

```
!set plsqlUseSlash true
SELECT sqlpl_function(password) AS psw FROM user_info;
```

输出结果为：

psw
115591
205239
531547
841242
986634
737297
600709

psw
990590
015859
783438

8.12.8.6.3. SQLPL 函数中不能有标准 SQL 语句

- 创建 SQL PL 函数 sqlpl_function

```
!set plsqlUseSlash true
-- 创建函数sqlpl_function，参数v_price的类型为IN，数据类型为DOUBLE。
CREATE OR REPLACE FUNCTION sqlpl_function(IN v_price DOUBLE)
-- 函数返回值的数据类型为字符串。
RETURNS STRING
BEGIN
  -- 声明一个字符串类型的变量v_time。
  DECLARE v_time STRING;
  -- 查询表transactions中与参数v_price相等的价格所对应的交易时间，并放进变量v_time里。
  SELECT trans_time INTO v_time FROM transactions WHERE price=v_price;
  -- 返回变量v_time的值。
  RETURN v_time;
END;
/
```

- 直接执行含有调用函数的 SQL 语句，Inceptor 会报错

```
SELECT * FROM transactions
WHERE trans_time=sqlpl_function(12.13);
```

输出结果为：

```
COMPILE FAILED: Semantic error: [Error 30028] Line 2:32 PLSQL function is running in a non-
driver environment, usually in SQL statement which doesn't allow nested SQL statement. Error
encountered near token '12.13'
```

8.12.8.7. 函数重载

- Inceptor 中支持函数的重载，即函数的名称相同，但是参数的类型或数目不能相同不同（参数的名称可以相同），返回值也可能不同。如果完全相同，就不是重载了，而是重复定义。
- 在实际的执行过程中，Inceptor 会根据实参的数据类型，来决定应该执行的函数。

- 例：重载函数 overload_test_func

```
!set plsqlUseSlash true
-- 创建第一个名为 overload_test_func 的过程，形参名为
place_in，其数据类型为字符串，函数返回值的数据类型为字符串。
CREATE OR REPLACE FUNCTION overload_test_func( IN place_in string)
RETURNS STRING
  BEGIN
    -- 第一个函数返回值为，在实参变量的前面加上字符 'hello'。
    RETURN 'Hello ' || place_in;
  END;
/
-- 创建第二个名为 overload_test_func 的过程，形参名为
place_in，其数据类型为整数类型，函数返回值的数据类型为整数型。
CREATE OR REPLACE FUNCTION overload_test_func( IN place_in INT)
RETURNS STRING
 BEGIN
    -- 第二个函数的返回值为，实参的平方。
    RETURN place_in * place_in;
  END;
/
BEGIN
   DECLARE V1 STRING;
   DECLARE V2 STRING;
   DECLARE V3 STRING;
   -- 函数内的实参为 world，类型为字符串，则会执行第一个函数。
   SET V1 =overload_test_func('world');
   -- 函数内的实参为 11，类型为整数类型，则会执行第二个函数。
   SET V2 =overload_test_func(11);
   -- 函数内的实参为 111，类型为字符串，则会执行第一个函数。
   SET V3 =overload_test_func('111');
  PUT_LINE(V1);
  PUT_LINE(V2);
  PUT_LINE(V3);
END;
/
```

输出结果：

output
Hello world
121
Hello 111

8.12.9. VARRAY

8.12.9.1. 定义数组类型及变量

- 定义数组类型

```
DECLARE TYPE <array-type-name> AS <data-type> ARRAY[int-constant];
```

例如：

```
DECLARE TYPE v_array AS INT ARRAY(5);
```

此处声明一个数组类型的数据类型，名称为 v_array，该数组可以存放 5 个整数类型的数据。

- 声明该类型的数组变量

```
DECLARE <array-name> <array-type-name>
```

例如：

```
DECLARE age_array v_array
```

此处声明一个 v_array 类型的变量 age_array。

8.12.9.2. 数组变量的赋值

与基本变量的赋值类似，数组变量的赋值语句通常也包括 SET，SELECT …INTO，VALUES …INTO 等几种形式，接下来将以案例的形式，介绍几种赋值语句的具体使用。

8.12.9.2.1. SET

- 例 1：

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 ARRAY 类型的数据类型 age_array，可以存放 5 个整数类型的数据。
  DECLARE TYPE age_array AS INT ARRAY[5];
  -- 声明一个 age_array 类型的变量 v_age_array。
  DECLARE v_age_array age_array;
  -- 给变量 v_age_array 中 5 个整数类型的变量同时赋值。
  SET v_age_array = ARRAY[18,19,20,21,22,23];
END;
/
```

- 例 2:

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 ARRAY 类型的数据类型 age_array，可以存放 5 个整数类型的数据。
  DECLARE TYPE age_array AS INT ARRAY[5];
  -- 声明一个 age_array 类型的变量 v_age_array。
  DECLARE v_age_array age_array;
  -- 使用关键字 SET 分别给变量 v_age_array 中 5 个整数类型的变量赋值。
  SET v_age_array(1) = 30;
  SET v_age_array(2) = 31;
  SET v_age_array(3) = 32;
  SET v_age_array(4) = 33;
  SET v_age_array(5) = 34;
END;
/
```

8.12.9.2.2. VALUES … INTO

- 例 2:

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 ARRAY 类型的数据类型 age_array，可以存放 5 个整数类型的数据。
  DECLARE TYPE age_array AS INT ARRAY[5];
  -- 声明一个age_array类型的变量v_age_array。
  DECLARE v_age_array age_array;
  -- 使用关键字 VALUES ... INTO 分别给变量 v_age_array 中 5 个整数类型的变量赋值。
  VALUES 20 INTO v_age_array(1);
  VALUES 21 INTO v_age_array(2);
  VALUES 22 INTO v_age_array(3);
  VALUES 23 INTO v_age_array(4);
  VALUES 24 INTO v_age_array(5);
END;
/
```

8.12.9.3. 数组变量的方法调用

- ARRAY_FIRST：返回数组中第一个元素的下标

- ARRAY_LAST：返回数组中最后一个元素的下标
- ARRAY_NEXT：返回数组下一个元素的下标
- ARRAY_PRIOR：返回数组上一个元素的下标
- ARRAY_DELETE：删除数组元素
- CARDINALITY：返回数组中元素的个数
- TRIM_ARRAY：从右开始删除指定数目个元素
- ARRAY_VARIABLE：返回参数指定的元素
- ARRAY_EXISTS：判断数组是否有元素
- MAX_CARDINALITY：返回数组中元素的个数
- UNNEST：将数组转换为表

8.12.9.3.1. ARRAY_FIRST

- 例 1:

```
!set plsqlUseSlash true
BEGIN
  -- 定义一个 ARRAY 类型 age_array，可存放 5 个整数类型的数据。
  DECLARE TYPE age_array AS INT ARRAY[5];
  -- 声明一个 age_array 类型的变量 v_age_array。
  DECLARE v_age_array age_array;
  -- 声明一个整数类型的变量 v_age。
  DECLARE v_age INT;
  -- 给变量 v_age_array 赋值，数组中的顺序依次为 18,19,20,21,22。
  SET v_age_array = ARRAY[18,19,20,21,22];
  -- 将变量 v_age_array 中的第一个元素的下标赋值给变量 v_age。
  SET v_age = v_age_array.ARRAY_FIRST;
  -- 打印出变量 v_age_array 中第一个元素的值。
  PUT_LINE('the first one:'||v_age_array(v_age));
END;
/
```

输出结果为:

output
the first one:18

8.12.9.3.2. ARRAY_LAST

- 例 1:

```
-- 将变量 v_age_array 中的最后一个元素的下标赋值给变量 v_age。
!set plsqlUseSlash true
BEGIN
  DECLARE TYPE age_array AS INT ARRAY[5];
  DECLARE v_age_array age_array;
  DECLARE v_age INT;
  SET v_age_array = ARRAY[18,19,20,21,22];
  SET v_age = v_age_array.ARRAY_LAST;
  PUT_LINE('the last one:'||v_age_array(v_age));
END;
/
```

输出结果为:

output
the last one:22

8.12.9.3.3. ARRAY_NEXT

- 例 1:

```
!set plsqlUseSlash true
BEGIN
  DECLARE TYPE age_array AS INT ARRAY[5];
  DECLARE v_age_array age_array;
  DECLARE v_age INT;
  SET v_age_array = ARRAY[18,19,20,21,22];
  -- 将变量 v_age_array 中第三个元素的下一个元素的下标赋值给变量 v_age。
  SET v_age = ARRAY_NEXT(v_age_array,3);
  PUT_LINE('the next one:'||v_age_array(v_age));
END;
/
```

输出结果为:

output
the next one:21

8.12.9.3.4. ARRAY_PRIOR

- 例 1:

```
!set plsqlUseSlash true
BEGIN
  DECLARE TYPE age_array AS INT ARRAY[5];
  DECLARE v_age_array age_array;
  DECLARE v_age INT;
  SET v_age_array = ARRAY[18,19,20,21,22];
  -- 将变量 v_age_array 中第三个元素的上一个元素的下标赋值给变量 v_age。
  SET v_age = ARRAY_PRIOR(v_age_array,3);
  PUT_LINE('the prior one:'||v_age_array(v_age));
END;
/
```

输出结果为:

output
the prior one:19

8.12.9.3.5. ARRAY_DELETE

- 例 1:

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 age_array 类型的变量 v_age。
  DECLARE TYPE age_array AS INT ARRAY[5];
  DECLARE v_age_array age_array;
  DECLARE v_age age_array;
  -- 声明一个整数类型的变量 v_sum。
  DECLARE v_sum INT;
  -- 删除变量 v_age_array 中的数据，并将剩下的元素赋值给变量 v_age。
  SET v_age_array = ARRAY[18,19,20,21,22];
  SET v_age = ARRAY_DELETE(v_age_array);
  -- 将变量 v_age_array 中的元素个数，赋值给变量 v_sum。
  SET v_sum = v_age.CARDINALITY;
  -- 打印出变量 v_age 的元素个数，由于变量 v_age_array 的数据已经删  除，所以此处会返回 0。
  PUT_LINE('变量v_age的个数:'||v_sum);
END;
/
```

输出结果为:

output
变量v_age的个数:0

8.12.9.3.6. CARDINALITY

- 例 1:

```
!set plsqlUseSlash true
BEGIN
  DECLARE TYPE age_array AS INT ARRAY[5];
  DECLARE v_age_array age_array;
  DECLARE v_age INT;
  SET v_age_array = ARRAY[18,19,20,21,22,23];
  SET v_age = v_age_array.CARDINALITY;
  PUT_LINE('数组个数：'||v_age);
END;
/
```

输出结果为:

output
数组个数：6

8.12.10. 行数据类型

与 SQL/PL 中的 Records 类型相似，用户可以自定义 SQL PL 中的行数据类型，且每一个行数据类型包含多个分量的名称及数据类型。在定义一个行数据类型时，我们可以分别定义分量的名称和数据类型，也可以直接声明一个基于表的行数据类型，其分量的字段名和数据类型与表中字段名和数据类型完全相同。

8.12.10.1. 语法

- 行数据类型的定义

```
DECLARE TYPE type_name AS ROW (variable_name data_type,variable_name data_type,...);
```

例如:定义一个名为 transactiontype 的行数据类型。

```
DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE);
```

- 行变量的声明

  DECLARE 变量名 数据类型名

  例如：声明一个数据类型为 transactiontype 的变量 transactionrow。

```
DECLARE transactionrow transactiontype
```

- 行变量字段

  变量名.分量名

  例如：行变量 transactionrow 包含行数据类型 transactiontype 的所有字段，可以用以下方式去调用。

```
transactiontype.v_acc_num;transactiontype.trans_time;transactiontype.price
```

8.12.10.2. 声明与赋值

行变量的赋值支持多种方式，在实际的应用中可以选择相应的方式进行赋值，本节中我们将以分别定义每一个分量的名称和数据类型的方式来声明一个行数据类型，并列举出几种常见的赋值语句，可供参阅。

8.12.10.2.1. 例一

- 例 1：将一个元组赋值给一个行变量

  元组的元素和元素类型需要和行变量的行数据类型一致。

```
-- 打印出行变量 transactionrow 中的字段 transactionrow.v_trans_time 的值。
-- 打印出行变量 transactionrow 中的字段 transactionrow.v_price 的值。
!set plsqlUseSlash true
BEGIN
 -- 定义一个行数据类型 transactiontype，包含了三个字段 v_acc_num，v_trans_time，v_price，其中字段
v_acc_num 的数据类型为 STRING，v_trans_time 的数据类型为 STRING，v_price 的数据类型为 DOUBLE。
 DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE);
 -- 声明一个行变量transactionrow，其数据类型为transactiontype，该变量包含transactiontype类型的三个字段。
 DECLARE transactionrow transactiontype;
  -- 将一个元组赋值给行变量 transactionrow，至此行变量 transactionrow 中的字段
transactionrow.v_acc_num，transactionrow.v_trans_time 和 transactionrow.v_price 的值分别为
0700735,20140916105811，11.11。
  SET transactionrow = ('0700735','20140916105811',11.11);
  -- 打印出行变量 transactionrow 中的字段 transactionrow.v_acc_num 的值。
  PUT_LINE('acc_num:'||transactionrow.v_acc_num);
  PUT_LINE('time:'||transactionrow.v_trans_time);
  PUT_LINE('price:'||transactionrow.v_price);
END;
/
```

  输出结果为:

output
acc_num:0700735
time:20140916105811
price:11.11

8.12.10.2.2. 例二

- 例 2：将一个表达式赋值给行变量

  使用关键字 VALUES INTO 语句进行赋值。

```
!set plsqlUseSlash true
BEGIN
 -- 定义一个行数据类型 transactiontype，包含了三个字段 v_acc_num , v_trans_time , v_price，其中字段
v_acc_num 的数据类型为 STRING，v_trans_time 的数据类型为 STRING，v_price 的数据类型为 DOUBLE。
 DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE);
 -- 声明一个行变量 transactionrow，其数据类型为 transactiontype，该变量包含 transactiontype
类型的三个字段。
 DECLARE transactionrow transactiontype;
  -- 将一个表达式通过 VALUES INTO 语句赋值给行变量 transactionrow，至此行变量 transactionrow 中的字段
transactionrow.v_acc_num，transactionrow.v_trans_time 和 transactionrow.v_price 的值分别为
0700735，20140916105811,11.11。
  VALUES ('0700735','20140916105811',11.11) INTO transactionrow;
  -- 打印出行变量 transactionrow 中的字段 transactionrow.v_acc_num 的值。
  PUT_LINE('acc_num:'||transactionrow.v_acc_num);
  -- 打印出行变量 transactionrow 中的字段 transactionrow.v_trans_time 的值。
  PUT_LINE('time:'||transactionrow.v_trans_time);
  -- 打印出行变量 transactionrow 中的字段 transactionrow.v_price 的值。
  PUT_LINE('price:'||transactionrow.v_price);
END;
/
```

  输出结果为：

output
acc_num:0700735
time:20140916105811
price:11.11

8.12.10.2.3. 例三

- 例 3：分别给每一个行变量的字段赋值

```
!set plsqlUseSlash true
BEGIN
 -- 定义一个行数据类型 transactiontype，包含了三个字段 v_acc_num，v_trans_time，v_price，其中字段
v_acc_num 的数据类型为 STRING，v_trans_time 的数据类型为 STRING，v_price 的数据类型为 DOUBLE。
 DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE);
 -- 声明一个行变量 transactionrow，其数据类型为 transactiontype，该变量包含 transactiontype
类型的三个字段。
 DECLARE transactionrow transactiontype;
  -- 给行变量 transactionrow 的字段 transactionrow.v_acc_num 赋值为 6513065。
  SET transactionrow.v_acc_num = '6513065';
  -- 给行变量 transactionrow 的字段 transactionrow.v_trans_time 赋值为 20140506133109。
  SET transactionrow.v_trans_time = '20140506133109';
  -- 给行变量 transactionrow 的字段 transactionrow.v_price 赋值为 12.13。
  SET transactionrow.v_price = '12.13';
  -- 打印出行变量 transactionrow 的字段 transactionrow.v_acc_num 的值。
  PUT_LINE('acc_num:'||transactionrow.v_acc_num);
  -- 打印出行变量 transactionrow 的字段 transactionrow.v_trans_time 的值。
  PUT_LINE('time:'||transactionrow.v_trans_time);
  -- 打印出行变量 transactionrow 的字段 transactionrow.v_price 的值。
  PUT_LINE('price:'||transactionrow.v_price);
END;
/
```

  输出结果为：

output
acc_num:6513065
time:20140506133109
price:12.13

8.12.10.2.4. 例四

- 例 4：将一个查询结果集的单行值赋值给一个行变量

```
!set plsqlUseSlash true
BEGIN
 -- 定义一个行数据类型 transactiontype，包含了三个字段v_acc_num，v_trans_time，v_price，其中字段
v_acc_num 的数据类型为 STRING，v_trans_time 的数据类型为 STRING，v_price 的数据类型为 DOUBLE。
 DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE);
 -- 声明一个行变量 transactionrow，其数据类型为transactiontype，该变量包含 transactiontype
类型的三个字段。
 DECLARE transactionrow transactiontype;
  -- 查询表 transactions中交易号为648230055的账号，交易时间，价格，并赋值给行变量 transactionrow。
  SET transactionrow = (SELECT acc_num,trans_time,price FROM transactions WHERE trans_id
=648230055);
  -- 打印出行变量transactionrow中的字段transactionrow.v_acc_num 的值。
  PUT_LINE('acc_num:'||transactionrow.v_acc_num);
  -- 打印出行变量transactionrow中的字段transactionrow.v_trans_time 的值。
  PUT_LINE('time:'||transactionrow.v_trans_time);
  -- 打印出行变量transactionrow中的字段transactionrow.v_price 的值。
  PUT_LINE('price:'||transactionrow.v_price);
END;
/
```

输出结果为：

output
acc_num:0700735
time:20140315111111
price:22.66

8.12.10.2.5. 例五

- 例 5：通过 SELECT INTO 语句将行值赋值给行变量

```
!set plsqlUseSlash true
BEGIN
 -- 定义一个行数据类型transactiontype，包含了三个字段v_acc_num , v_trans_time , v_price
，其中字段v_acc_num的数据类型为STRING，v_trans_time的数据类型为STRING，v_price的数据类型为DOUBLE。
 DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE);
 -- 声明一个行变量transactionrow，其数据类型为transactiontype，该变量包含transactiontype类型的三个字段。
 DECLARE transactionrow transactiontype;
 -- 通过SELECT INTO语句将表transactions中股票账号为GL2547626的账号，交易时间和价格放进行变量
transactionrow中。
 SELECT acc_num,trans_time,price INTO transactionrow FROM transactions WHERE stock_id='
GL2547626';
  -- 打印出行变量transactionrow中的字段transactionrow.v_acc_num的值。
  PUT_LINE('acc_num:'||transactionrow.v_acc_num);
  -- 打印出行变量transactionrow中的字段transactionrow.v_trans_time的值。
  PUT_LINE('time:'||transactionrow.v_trans_time);
  -- 打印出行变量transactionrow中的字段transactionrow.v_price的值。
  PUT_LINE('price:'||transactionrow.v_price);
END;
/
```

输出结果为：

output
acc_num:5224133
time:20140801110003
price:6.36

8.12.10.3. 基于表的行数据类型的声明

本节中将基于表直接声明一个行数据类型，其分量的名称和数据类型与表中的字段名和数据类型完全相同。

- 例 1：声明一个名为 transaction_row 的行数据类型，其分量的名称和数据类型与表 transactions 中的字段名和数据类型完全相同

```
!set plsqlUseSlash true
BEGIN
 -- 声明一个与表transactions中的字段名和字段类型完全相同的行数据类型transaction_row。
 DECLARE TYPE transaction_row as row anchor to row of transactions;
 -- 声明一个transaction_row类型的变量transactionrow。
 DECLARE transactionrow transaction_row;
 -- 查询表transactions中股票账号为GL2547626的全部信息，并放进变量transactionrow里。
 SELECT  *  INTO transactionrow
 FROM transactions WHERE stock_id='GL2547626';
  -- 依次打印出变量transactionrow中的账号，交易时间和价格。
  PUT_LINE('acc_num:'||transactionrow.acc_num);
  PUT_LINE('time:'||transactionrow.trans_time);
  PUT_LINE('price:'||transactionrow.price);
END;
/
```

输出结果为：

output
acc_num:5224133
time:20140801110003
price:6.36

8.12.11. 游标

8.12.11.1. 组成部分

游标的组成通常可分为四个部分，分别为声明游标，打开游标，取数据，以及关闭游标。

- 在声明部分声明游标：

  语法：

```
DECLARE cursor_name CURSOR [WITH RETURN] FOR  sql_statement;
```

- 打开游标：

  语法：

```
OPEN cursor_name;
```

- 取数据：

  语法：

```
FETCH cursor_name INTO Variable;
```

- 关闭游标：

  语法：

```
CLOSE cursor_name;
```

8.12.11.2. 基本用法

- 例 1：

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个名为 c1 的游标，用来查找表 transactions 中价格为 12.13 的交易时间。
  DECLARE c1 CURSOR FOR SELECT trans_time FROM transactions WHERE price=12.13;
  -- 声明一个名为 v_time 的变量，数据类型为字符串。
  DECLARE v_time STRING;
  -- 打开游标。
  OPEN c1;
  -- 将游标查询到的结果放进变量 v_time 里。
  FETCH c1 INTO v_time;
  -- 关闭游标。
  CLOSE c1;
  -- 打印出变量 v_time 的值。
  PUT_LINE('the time is:'||v_time);
END;
/
```

  输出结果为：

output
the time is:20140105100520

8.12.11.3. ALLOCATE CURSOR

在调用存储过程时，可以使用 ALLOCATE CURSOR 语句来返回游标.具体的使用方法，将首先通过 ASSOCIATE LOCATOR 语句来声明存储过程的结果集，接着使用 ALLOCATE CURSOR 语句将不同的游标分配给相应的结果集。

8.12.11.3.1. 语法

- 声明结果集

```
ASSOCIATE RESULT SET LOCATOR  (result_name1, result_name2,...) WITH  PROCEDURE  procedure_name;
```

- 分配游标

```
ALLOCATE <cursor-name> CURSOR FOR RESULT SET <locator-variable>;
```

8.12.11.3.2. 用例说明

- 创建名为 pro_1 的存储过程，在过程内声明两个带有返回值的游标

```
!set plsqlUseSlash true
-- 创建一个名为pro_1且不带参数的存储过程。
CREATE OR REPLACE PROCEDURE pro_1()
LANGUAGE SQL
-- 存储过程pro_1将返回两个结果集。
DYNAMIC RESULT SETS 2
BEGIN
  -- 声明一个名为c1的游标，用来查询表transactions中价格为12.13的账号。
  DECLARE c1 CURSOR WITH RETURN FOR
      SELECT acc_num FROM transactions WHERE price=12.13;
  -- 打开游标c1。
  OPEN c1;
  BEGIN
     -- 声明一个名为c2的游标，用来查询表transactions中价格为12.13的交易时间。
     DECLARE c2 CURSOR WITH RETURN FOR
       SELECT trans_time FROM transactions WHERE price=12.13;
     OPEN c2;
  END;
END;
/
```

- 调用 pro_1() 过程，在过程内分配两个带有返回值的游标

```
!set plsqlUseSlash true
BEGIN
 -- 声明两个结果集rsl1, rsl2。
 DECLARE rsl1, rsl2 result_set_locator varying;
 DECLARE v1 STRING;
 -- 调用过程pro_1。
 CALL pro_1();

 -- 结果集rsl1, rsl2分别用来存放过程pro_1返回的结果。
 ASSOCIATE RESULT SET LOCATOR (rsl1, rsl2) WITH PROCEDURE pro_1;
 -- 游标c1查询到的结果作为结果集rsl1，游标c2查询到的结果作为结果集rsl2。
 ALLOCATE c1 CURSOR FOR result set rsl1;
 ALLOCATE c2 CURSOR FOR result set rsl2;

 -- 将游标c1查询到的结果放进变量v1里。
 FETCH c1 INTO v1;
 -- 关闭游标c1。
 CLOSE c1;
   -- 打印出变量v1的值，即所查询到的表transactions中的账号。
   PUT_LINE('the acc_num is:'||v1);
 -- 将游标c2查询到的结果放进变量v1里。
 FETCH c2 INTO v1;
 -- 关闭游标c2。
 CLOSE c2;
   -- 打印出变量v2的值，即所查询到的表transactions中的账号。
   PUT_LINE('the time is:'||v1);
END;
/
```

输出结果为:

output
the acc_num is:6513065
the time is:20140105100520

8.12.12. 与 SQL 的交互

8.12.12.1. 将 SELECT 返回值赋值给变量

例 299. 将查询结果赋值给变量

```
!set plsqlUseSlash true
BEGIN
  DECLARE V1 STRING;
  DECLARE V2 DOUBLE;
   -- 查询 transactions 表中价格为 11.11 的交易时间，并赋值给变量 V1。
   SET V1 = (SELECT trans_time FROM transactions WHERE price = 11.11);
   -- 查询 transactions 表中交易号为 162742112 的价格，并赋值给变量 V2。
   SET V2 = (SELECT price FROM transactions WHERE trans_id=162742112);
  PUT_LINE('V1:'||V1);
  PUT_LINE('V2:'||V2);
END;
/
```

输出结果为：

output
V1:20140205140521
V2:12.21

8.12.12.2. SELECT INTO

例 300. 通过 SELECT INTO 语句将行值赋值给行变量

```
!set plsqlUseSlash true
BEGIN
 -- 定义一个行数据类型 transactiontype，包含了三个字段 v_acc_num，v_trans_time，v_price，其中字段
v_acc_num 的数据类型为 STRING，v_trans_time 的数据类型为 STRING，v_price 的数据类型为 DOUBLE。
 DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE);
 -- 声明一个行变量 transactionrow，其数据类型为 transactiontype，该变量包含 transactiontype
类型的三个字段。
 DECLARE transactionrow transactiontype;
 -- 通过 SELECT INTO 语句将表 transactions 中股票账号为 GL2547626 的账号，交易时间和价格放进行变量
transactionrow 中。
 SELECT acc_num,trans_time,price INTO transactionrow FROM transactions WHERE stock_id=
'GL2547626';
  -- 打印出行变量 transactionrow 中的字段 transactionrow.v_acc_num 的值。
  PUT_LINE('acc_num:'||transactionrow.v_acc_num);
  -- 打印出行变量 transactionrow 中的字段 transactionrow.v_trans_time 的值。
  PUT_LINE('time:'||transactionrow.v_trans_time);
  -- 打印出行变量 transactionrow 中的字段 transactionrow.v_price 的值。
  PUT_LINE('price:'||transactionrow.v_price);
END;
/
```

输出结果为：

output
acc_num:5224133
time:20140801110003
price:6.36

8.12.12.3. 存储过程与 SQL 的交互

Inceptor 中可以利用创建存储过程，对表进行增删改查等操作，如下例所示，将 DML 语句如 UPDATE，INSERT，DELETE，MERGE 等语句，写进存储过程中，这样在调用存储过程的过程中，就可以批量执行 DML 语句。

例 301. 在存储过程内执行 UPDATE，INSERT，DELETE，MERGE 等 DML 语句

```
-- 调用过程 sql_procedure，实参的值分别为 zhangsan，4.5。
!set plsqlUseSlash true
-- 创建名为 sql_procedure 的存储过程，参数类型均为 IN 类型，其中形参 v_name 的数据类型为 STRING，形参
v_gpa 的数据类型为 DOUBLE。
CREATE OR REPLACE PROCEDURE sql_procedure(IN v_name STRING,IN v_gpa DOUBLE)
BEGIN
  -- 往表 zara 中插入一条数据，姓名为 Amy，年龄为 23，绩点为 4.5。
  INSERT INTO zara VALUES('Amy',23,4.5);
  -- 将表 zara 中姓名和参数 v_name 相等的人的年龄更新为 18。
  UPDATE zara SET age= 18 WHERE name = v_name;
  -- 删除表 zara 中绩点与参数 v_gpa 的值相等的记录。
  DELETE FROM zara WHERE gpa = v_gpa;
  -- 对表 za 进行 merge 操作，条件为两个表中的姓名相同。
  MERGE INTO za z using zara r on (z.name=r.name)
  -- 如果条件匹配，即两个表中存在姓名相同的记录，就将表 za 中姓名和表 zara 中姓名相同的人的年龄更新为 40。
  WHEN MATCHED THEN UPDATE SET z.age=40
  -- 如果条件不匹配，就将表 zara 中与表 za 中姓名不相同的姓名和对应的年龄插入到表 za 中。
  WHEN NOT MATCHED THEN INSERT (name,age) VALUES(r.name,r.age);
END;
/
CALL sql_procedure('zhangsan',4.5);
```

8.12.12.4. 函数与 SQL 的交互

与存储过程类似，在 Inceptor 中也可以将 UPDATE，INSERT，DELETE，MERGE 等 DML 语句，放进函数里，在调用函数时，执行函数内的 SQL 语句。

例 302. 在函数内执行 UPDATE，INSERT，DELETE，MERGE 等 DML 语句

```
!set plsqlUseSlash true
-- 创建一个名为 sql_function 的函数，参数的类型为 IN 类型，参数的数据类型为 STRING。
CREATE OR REPLACE FUNCTION sql_function(IN v_name STRING)
-- sql_function 函数的返回值的数据类型为 DOUBLE。
RETURNS DOUBLE
BEGIN
  DECLARE v_gpa DOUBLE;
  -- 查询表 zara 中姓名和参数相等的人的绩点，并放进一个 DOUBLE 类型的变量 v_gpa 里。
  SELECT gpa INTO v_gpa FROM zara WHERE name=v_name;
  -- 函数返回值为 v_gpa 的值。
  RETURN v_gpa;
END;
/
BEGIN
  -- 在一个语句块内执行 DML 语句 DELETE，删除表 zara 中绩点与 sql_function('lily') 返回值相等的记录。
  DELETE FROM zara WHERE gpa= sql_function('lily');
END;
/
```

本例中的 DELETE 语句，必须放在语句块中去执行，单独 SQL 语句调用函数会出错，原因是本例中的 sql_function 函数内含有标准 SQL 语句，更多关于 SQL 调用 SQL PL 函数的注意注意事项，可参见SQLPL 函数章节。

8.12.12.5. 游标

在 Inceptor 中可以通过游标查询数据，与 SQL 进行交互，本节中将以游标的基本查询为例，有关游标的更多使用用法，可参见游标章节。

例 303. 游标的基本查询示例

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个名为 c1 的游标，用来查找表 transactions 中价格为 12.13 的交易时间。
  DECLARE c1 CURSOR FOR SELECT trans_time FROM transactions WHERE price=12.13;
  -- 声明一个名为 v_time 的变量，数据类型为字符串。
  DECLARE v_time STRING;
  -- 打开游标。
  OPEN c1;
  -- 将游标查询到的结果放进变量 v_time 里。
  FETCH c1 INTO v_time;
  -- 关闭游标。
  CLOSE c1;
  -- 打印出变量 v_time 的值。
  PUT_LINE('the time is:'||v_time);
END;
/
```

输出结果为：

output
the time is:20140105100520

8.12.12.6. 动态 SQL

8.12.12.6.1. EXECUTE IMMEDIATE

- Dynamic SQL 中最常使用 EXECUTE IMMEDIATE 语句来执行动态 SQL 语句，使用 EXECUTE IMMEDIATE 语句可以创建执行 DDL 语句，DCL 语句，DML 语句以及单行的 SELECT 语句，但该方法不能用于处理多行查询语句。
- 在具体的使用过程中，是将相应的DDL 语句，DCL 语句，DML 语句以及单行的 SELECT 语句等 SQL 语句，放进一个已声明的字符串变量中，再用 EXECUTE IMMEDIATE 去执行该字符串变量，或者 EXECUTE IMMEDIATE 关键字后直接跟含有 SQL 语句的字符串常量。

处理 DDL 语句

CREATE

例 304. 使用 EXECUTE IMMEDIATE 创建表 sqlpl_create

```
!set plsqlUseSlash true
drop table if exists sqlpl_create;
BEGIN
  DECLARE create_table STRING;
  -- 注意字符串与连接符之间的空格，否则 {inceptor} 可能会报错。
  SET create_table = 'create table ' ||'sqlpl_create'
                     ||'(sno string , sname string)';
  EXECUTE IMMEDIATE create_table;
END;
/
```

ALTER

1. 查看 sqlpl_create 表的结构

```
desc sqlpl_create;
```

col_name	data_type	default_value	not_null	unique	comment
sno	string	NULL	No	No	
sname	string	NULL	No	No	

2. 使用 EXECUTE IMMEDIATE 修改表 sqlpl_create 中列 sno 的列名和数据类型

```
!set plsqlUseSlash true
BEGIN
  DECLARE alter_table STRING;
  SET alter_table ='alter table sqlpl_create CHANGE sno sid INT';
  EXECUTE IMMEDIATE alter_table;
END;
/
```

3. 再次查看 sqlpl_create 表的结构

```
desc sqlpl_create;
```

col_name	data_type	default_value	not_null	unique	comment
sid	int	NULL	No	No	
sname	string	NULL	No	No	

DROP

例 305. 以同样的方法创建表 test_create，现在我们来用 EXECUTE IMMEDIATE 删除表 test_create

1. 使用 EXECUTE IMMEDIATE 删除表 test_create

```
!set plsqlUseSlash true
BEGIN
  DECLARE drop_table  STRING;
    SET drop_table ='drop table '||'test_create';
  EXECUTE IMMEDIATE drop_table;
END;
/
```

2. 再次查看数据库中的表，可以发现表 test_create 已经不存在了。

处理DCL语句

在使用 EXECUTE IMMEDIATE 语句执行 DCL 语句时，需要确保当前角色为 admin，或者当前用户有权限对其它用户进行赋予/收回全局权限的操作。

GRANT

1. 创建过程 grant_priv

```
set plsql.use.slash=true;
CREATE OR REPLACE PROCEDURE grant_priv(priv STRING, username STRING)
BEGIN
  DECLARE priv_stat STRING;
    SET priv_stat =' GRANT '|| priv || ' TO user ' || username;
   EXECUTE IMMEDIATE priv_stat;
  END;
  /
```

2. 调用创建的过程

```
set plsql.use.slash=true;
BEGIN
 grant_priv('create', 'user1');
END;
/
```

3. 查看 user1 被赋予的权限

```
show grant user user1 on all;
```

database	table	partition	column	principal_name	principal_type	privilege	grant_option	grant_time	grantor
				user1	USER	CREATE	false	0	N/A

REVOKE

1. 创建过程 revoke_priv

```
set plsql.use.slash=true;
CREATE OR REPLACE PROCEDURE revoke_priv(priv STRING, username STRING)
BEGIN
  DECLARE priv_stat STRING;
   SET priv_stat =' revoke '|| priv || ' from user  ' || username;
   EXECUTE IMMEDIATE priv_stat;
END;
/
```

2. 调用过程

```
set plsql.use.slash=true;
BEGIN
  revoke_priv('create', 'user1');
END;
/
```

3. 再次查看 user1 的权限

   可以看到，此时 user1 不再具有CREATE的权限。

```
show grant user user1 on all;
```

database	table	partition	column	principal_name	principal_type	privilege	grant_option	grant_time	grantor

处理单行查询

要查看 SELECT 查询的结果需要设置 plsql.show.sqlresults 的属性为 true，更多事项可参见注意事项章节。

例 306. EXECUTE IMMEDIATE 后接字符串类型的变量

```
set plsql.show.sqlresults=true;
!set plsqlUseSlash true
BEGIN
   DECLARE sql_stat STRING;
     SET sql_stat ='select price from transactions where acc_num=6513065';
   EXECUTE IMMEDIATE sql_stat;
END;
/
```

输出结果为:

output
12.13
6.12
9.81
7.49
7.02
18.38

例 307. EXECUTE IMMEDIATE 后接字符串常量

```
set plsql.show.sqlresults=true;
!set plsqlUseSlash true
BEGIN
   EXECUTE IMMEDIATE 'select price from transactions where acc_num=6513065';
END;
/
```

输出结果为:

output
12.13
6.12
9.81
7.49
7.02
18.38

例 308. EXECUTE IMMEDIATE 执行含有占位符的 SELECT 查询语句

```
set plsql.show.sqlresults=true;
!set plsqlUseSlash true
BEGIN
   -- 此处用问号作占位符，且该问号必须为英文键盘情况下的问号
   EXECUTE IMMEDIATE 'select price from transactions where acc_num=?' using 6513065;
END;
/
```

输出结果为：

output
12.13
6.12
9.81
7.49
7.02
18.38

处理不含占位符的 DML 语句

INSERT

例 309. 以同样的方法创建表 holotest1，字段名分别为 age，gpa，数据类型分别为 int，double。

1. 使用 EXECUTE IMMEDIATE 往表中插入数据

```
set plsql.show.sqlresults=true;
!set plsqlUseSlash true
BEGIN
  DECLARE insert_table STRING;
   SET insert_table = 'INSERT INTO holotest1 values(20,4.0)';
  EXECUTE IMMEDIATE insert_table;
END;
/
```

2. 查看表 holotest1 的数据，可以发现新增了一条数据

```
select * from holotest1
```

age	gpa
20	4.0
21	3.9
22	4.3

UPDATE

1. 使用 EXECUTE IMMEDIATE 更新表中数据

```
!set plsqlUseSlash true
BEGIN
  DECLARE update_table STRING;
    SET update_table =' UPDATE holotest1 SET gpa=4.4 WHERE age=20 ';
  EXECUTE IMMEDIATE update_table;
END;
/
```

2. 查看表 holotest1 的数据，可以发现年龄为 20 的 gpa 已经发生了变化

```
select * from holotest1;
```

age	gpa
20	4.4
21	3.9
22	4.3

DELETE

1. 使用 EXECUTE IMMEDIATE 表中数据

```
!set plsqlUseSlash true
BEGIN
   DECLARE delete_table STRING;
     SET delete_table ='DELETE FROM holotest1 WHERE age=20';
   EXECUTE IMMEDIATE delete_table;
END;
/
```

2. 查看表 holotest1 的数据，可以发现年龄为 20 的记录已经被成功删除

```
select * from holotest1;
```

age	gpa
21	3.9
22	4.3

CREATE+INSERT

1. 使用 EXECUTE IMMEDIATE 创建表格并插入数据

```
!set plsqlUseSlash true
drop table if exists holotest2;
BEGIN
  DECLARE create_table string;
  DECLARE insert_table string;
    SET create_table ='create table holotest2 (age int, gpa double)
                 CLUSTERED BY (age) INTO 2 BUCKETS
                 STORED AS HOLODESK;';
   EXECUTE IMMEDIATE create_table;
    SET insert_table ='insert into holotest2 values (22,4.3)';
   EXECUTE IMMEDIATE insert_table;
END;
/
```

2. 查看表 holotest2 及表中的数据

```
desc holotest2;
```

col_name	data_type	default_value	not_null	unique	comment
age	int	NULL	No	No	
gpa	double	NULL	No	No	

3. 可以看到在成功创建 holotest2 的同时，也成功插入了一条数据

```
select * from holotest2;
```

age	gpa
22	4.3

CREATE+INSERT+UPDATE+DELETE+INSERT

1. 使用 EXECUTE IMMEDIATE 处理一系列 DML 语句

```
!set plsqlUseSlash true
drop table if exists holotest3;
BEGIN
  DECLARE create_table STRING;
  DECLARE insert_table STRING;
  DECLARE update_table STRING;
  DECLARE delete_table STRING;
   -- 创建 HOLODESK 表 holotest3，字段名分为 pid，price,数据类型分为 INT,DOUBLE
   SET create_table ='create table holotest3 (pid int, price double)
                CLUSTERED BY (pid) INTO 2 BUCKETS
                STORED AS HOLODESK';
EXECUTE IMMEDIATE create_table;
   -- 往表 holotest3 中插入数据 (111,12.13)
   SET insert_table ='insert into holotest3 values (111,12.13)';
EXECUTE IMMEDIATE insert_table;
   -- 将 pid 为 111 的价格更新为 14.15
   SET update_table ='update holotest3 set price=14.15 where pid=111';
EXECUTE IMMEDIATE update_table;
   -- 删除表 holotest3 中 pid 为 111 的记录
   SET delete_table ='delete from holotest3 where pid=111';
EXECUTE IMMEDIATE delete_table;
   -- 往表 holotest3 中插入数据 (112,11.11)
   SET insert_table ='insert into holotest3 values (112,11.11)';
EXECUTE IMMEDIATE insert_table;
END;
/
```

2. 查看表 holotest3 的数据

```
select * from holotest3;
```

pid	price
112	11.11

处理含有占位符的 DML 语句

> 在使用 EXECUTE IMMEDIATE 处理含有占位符的 SQL 语句时，问号必须是英文键盘下的问号。

INSERT

1. 使用 EXECUTE IMMEDIATE 往表中插入数据

```
!set plsqlUseSlash true
BEGIN
  DECLARE insert_table STRING;
   SET insert_table = 'INSERT INTO holotest2 values(?,?)';
  EXECUTE IMMEDIATE insert_table USING 22,4.7;
  EXECUTE IMMEDIATE insert_table USING 24,4.8;
END;
/
```

2. 此处案例执行的结果是往表 holotest2 插入两条记录，分别为 (22,4.7)，(24,4.8)。

UPDATE

1. 使用 EXECUTE IMMEDIATE 更新表中数据

```
!set plsqlUseSlash true
BEGIN
  DECLARE update_table STRING;
    SET update_table =' UPDATE holotest2 SET gpa=4.4 WHERE age=? ';
  EXECUTE IMMEDIATE update_table USING 22;
  EXECUTE IMMEDIATE update_table USING 24;
END;
/
```

2. 此处案例的执行结果是将表 holotest2 年龄为 22 和 24 的记录的 gpa 更新为 4.4。

DELETE

1. 使用 EXECUTE IMMEDIATE 删除表中数据

```
!set plsqlUseSlash true
BEGIN
   DECLARE delete_table STRING;
     SET delete_table ='DELETE FROM holotest2 WHERE age=?';
   EXECUTE IMMEDIATE delete_table using 20;
   EXECUTE IMMEDIATE delete_table using 24;
END;
/
```

2. 此处案例的执行结果是删除表 holotest2 中年龄为 20 和 24 的记录。

8.12.12.6.2. PREPARE … EXECUTE

同样，Inceptor 也支持使用 PREPARE 关键字将要执行的 SQL 语句放进变量，再用EXECUTE关键字来执行的方式，但需要注意：

- 用来存储 SQL 语句的变量，必须为 STATEMENT 类型的变量。
- 用来做占位符的问号为 ?。
- PREPARE … EXECUTE只能用来处理 SELECT 查询及 DML 语句。

SELECT

> 要查看 SELECT 查询的结果需要设置 plsql.show.sqlresults 的属性为 True，更多事项可参见注意事项章节。

例 310. SELECT

```
!set plsqlUseSlash true
set plsql.show.sqlresults=true;
BEGIN
  -- 声明一个 STATEMENT 类型的变量 stmt。
  DECLARE stmt STATEMENT;
   -- 使用关键字 PREPARE，将 SQL 语句 '查询t ransactions 表中账号名所对应的行数' 放进变量 stmt 里。
   PREPARE stmt FROM  ' begin select count(*) from ' || 'transactions' || ' where acc_num = ? ;
end';
  -- 使用关键字执行变量 stmt 内的 SQL 语句，查询账号为 6513065 的行数。
  EXECUTE stmt using 6513065;
  -- 使用关键字执行变量 stmt 内的 SQL 语句，查询账号为 0700735 的行数。
  EXECUTE stmt using 0700735;
END;
/
```

输出结果为：

output
6
3

执行结果为账号为 6513065 的有 6 行，账号为 0700735 的有 3 行。

INSERT

例 311. INSERT

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 STATEMENT 类型的变量 insert_table。
  DECLARE insert_table STATEMENT;
   -- 使用关键字 PREPARE 将 SQL 语句'往表 holotest1 中插入数据'放进变量 insert_table 里。
   PREPARE insert_table FROM 'INSERT INTO holotest1 values(?,?)';
  -- 使用关键字 EXECUTE 执行上述 SQL 语句，分别往表 holotest1 中插入 4 条不同的数据。
  EXECUTE insert_table using 18,4.4;
  EXECUTE insert_table using 19,4.5;
  EXECUTE insert_table using 21,4.6;
  EXECUTE insert_table using 23,4.7;
END;
/
```

查看表 holotest1 中的数据

```
select * from holotest1;
```

age	gpa
21	3.9
21	4.6
22	4.3
18	4.4
23	4.7
19	4.5

可以看到，表 holotest1 中成功插入了 4 条新的数据。

UPDATE

例 312. 表中只有一个字段是未知的

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 STATEMENT 类型的变量 update_table。
  DECLARE update_table STATEMENT;
   -- 使用关键字 PREPARE 将 SQL 语句 '更新表中的 gpa 的值为 5.0' 放进变量 update_table 里。
   PREPARE update_table FROM 'UPDATE holotest1 SET gpa=5.0 WHERE age = ?';
  -- 使用关键字 EXECUTE 执行变量 update_table 中的 SQL 语句，分别更新表 holotest1 中年龄为 18，19 的
gpa 设为 5.0。
  EXECUTE update_table using 18;
  EXECUTE update_table using 19;
END;
/
```

再次查看表 holotest1，可以看到年龄为 18 和 19 的 gpa 已经成功更新了。

```
select * from holotest1;
```

age	gpa
21	3.9
21	4.6
22	4.3
18	5.0
23	4.7
19	5.0

例 313. 表中两个字段都是未知的

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 STATEMENT 类型的变量 update_table。
  DECLARE update_table STATEMENT;
   -- 使用关键字 PREPARE 将 SQL 语句 '更新表中的 gpa 的值' 放进变量 update_table 里。
   PREPARE update_table FROM 'UPDATE holotest1 SET gpa=? WHERE age = ?';
  -- 使用关键字 EXECUTE 执行变量 update_table 中的 SQL 语句，分别更新表 holotest1 中年龄为 21 的 gpa 为
4.0，年龄 23 的 gpa 为 4.4。
  EXECUTE update_table using 4.0,21;
  EXECUTE update_table using 4.4,23;
END;
/
```

查看表 holotest1，可以看到年龄为 21，和 23 的 gpa 已经成功更新了。

```
select * from holotest1;
```

age	gpa
21	4.0
21	4.0
22	4.3
18	5.0
23	4.4
19	5.0

DELETE

例 314. DELETE

```
!set plsqlUseSlash true
BEGIN
  -- 声明一个 STATEMENT 类型的变量 delete_table。
  DECLARE delete_table STATEMENT;
   -- 使用关键字 PREPARE，将 SQL 语句 '删除表 holotest1 的相应年龄的记录' 放进变量 delete_table 里。
   PREPARE delete_table FROM 'DELETE FROM holotest1 WHERE age= ?';
  -- 使用关键字 EXECUTE，执行变量 delete_table 里的 SQL 语句，分别删除表 holotest1 中年龄为 22 和 24
的记录。
  EXECUTE delete_table using 22;
  EXECUTE delete_table using 24;
END;
/
```

再次查看表 holotest1

```
select * from holotest1;
```

age	gpa
21	4.0
21	4.0
18	5.0
23	4.4
19	5.0

8.12.12.7. GET DIAGNOSTICS

- ROW_COUNT：返回前面执行的 SQL 语句处理的行数。
- EXCEPTION 1：前面执行的 SQL 语句返回的 DB2 错误或警告消息文本。

8.12.12.7.1. ROW_COUNT

例 315. ROW_COUNT

```
!set plsqlUseSlash true
-- 创建一个名为 upd_zara_gpa 的过程，参数 v_age，参数 v_age 的类型均为 IN，数据类型均为整数型；参数
no_of_rows 的类型为 OUT，数据类型为整数型。
CREATE OR REPLACE PROCEDURE upd_zara_gpa(in v_age INT, in v_gpa INT, out no_of_rows INT)
 -- 指定程序的主体语言为 SQL。
 language sql
 BEGIN
   -- 查找表 zara 中年龄等于参数 v_age 的记录，并将参数 v_gpa 的值赋给对应的记录。
   UPDATE zara SET gpa = v_gpa WHERE age = v_age;
   -- 使用关键字 GET DIAGNOSTICS 获取所执行的 SQL 语句处理的行数，并赋给参数 no_of_rows。
   GET DIAGNOSTICS no_of_rows = ROW_COUNT;
   -- 打印出 SQL 语句所处理的行数。
   PUT_LINE('更新行:'||no_of_rows);
 END;
/
BEGIN
   -- 在调用过程 upd_zara_gpa 的语句块内，声明一个整数类型的变量 row。
   DECLARE row INT;
   -- 调用过程 upd_zara_gpa，参数分别为 22，4.5，row，此处执行的结果是，把表 zara 中年龄为 22 的人的 gpa
改为 4.5，并将更新的行数赋值给 row。
   upd_zara_gpa(22,4.5,row);
 END;
 /
```

输出结果为：

output
更新行:2

8.12.12.7.2. EXCEPTION 1

1. 创建存储过程 proc_11

```
!set plsqlUseSlash true
-- 创建存储过程 proc_11，参数 b 的类型为 OUT，数据类型为整数型。
CREATE OR REPLACE PROCEDURE proc_11(out b int)
BEGIN
 -- 分别声明两个整数类型的变量 sqlcode，sql_code；三个字符串类型的变量 sqlstate，sql_state，errMsg。
 DECLARE sqlcode, sql_code int;
 DECLARE sqlstate, sql_state string;
 DECLARE errMsg string;
 -- 为异常 sqlexception 声明一个 continue 的异常处理器。
 DECLARE continue handler for sqlexception
 BEGIN
  -- 使用关键字 GET DIAGNOSTICS 获取执行 SQL 返回的错误或警告信息。
  GET DIAGNOSTICS exception 1 errMsg = DB2_TOKEN_STRING;
  -- 将 sqlcode，sqlstate 赋值给变量 sql_code，sql_state。
  VALUES(sqlcode, sqlstate) INTO sql_code, sql_state;
  -- 如果 sql_code 不等于 -438，或 sql_state 不等于 54321，或 errMsg 不等于 exception on
purpose，就抛出异常，sqlstate 的值为 30001。
  IF sql_code != -438 or sql_state != '54321' or errMsg != 'exception on purpose' THEN
   signal sqlstate '30001';
  END IF;
 END;
 -- 抛出一个异常，sqlstate 的值为 54321，MESSAGE_TEXT 的值为 exception on purpose。
 signal sqlstate value '54321' set MESSAGE_TEXT = 'exception on purpose';
  -- 给参数 b 赋值为 20。
 SET b = 20;
 -- 分别打印出变量 sql_code，sql_state 以及参数 b 的值。
 PUT_LINE('sqlcode is:'||sql_code);
 PUT_LINE('sqlstate is:'||sql_state);
 PUT_LINE('value of b:'||b);
END;
/
```

2. 调用存储过程 proc_11

```
!set plsqlUseSlash true
BEGIN
 DECLARE b INT default -10;
 CALL proc_11(b);
 IF b != 20 THEN
  signal sqlstate value '30002';
 END IF;
END;
/
```

3. 输出结果为:

output
sqlcode is:-438
sqlstate is:54321
value of b:20

8.12.13. 系统预定义函数/过程

8.12.13.1. 查看预定义函数/过程

8.12.13.1.1. 相关命令合集

- 查看已有函数/过程

  （不指定 db_name 的话即对当前数据库操作）

  ```
  SHOW PLSQL FUNCTIONS db_name;
  ```

- 查看某一 PL/SQL 函数/过程的详细信息

  （EXTENDED 关键字会列出该 PL/SQL 函数/过程的原文）

  ```
  DESC PLSQL FUNCTION EXTENDED function_name
  ```

- 创建函数/过程

  ```
  CREATE (OR REPLACE) FUNCTION/PROCEDURE
  ```

- 删除函数/过程/包/包体

  ```
  DROP PLSQL FUNCTION/PROCEDURE
  ```

> 由于 PLSQL 是运行在 Inceptor Server 端的 SESSION 中，如果 PLSQL 语句正在运行时，客户端通过 Ctrl+C 结束执行，此时 Server 端由于还在执行 PLSQL，无法侦听到客户端的结束请求，该 PLSQL 会继续执行到结束。如果它包含了大量的 SQL 语句执行或者存在死循环，会耗费大量的服务端资源。所以我们需要手动执行一些命令来终止服务端对 PLSQL 的执行。

- 查看正在运行的PLSQL程序的SESSION ID

```
PS PLSQL
```

这条命令会显示 SESSION ID 和 PLSQL 语句，使用者需要从中找到自己想要终止的 PLSQL 的 SESSION ID。

- 终止正在运行的 PLSQL 程序

```
KILL PLSQL <SESSION_ID>
```

这条命令会发送一个终止信号给该 SESSION ID 中的 PLSQL 进程，待该进程下一次自检之时，发现有终止信号，则会结束自己。

8.12.13.1.2. 案例合集

- 例 1：查看已有 PLSQL 函数（不指定 db_name）

```
SHOW PLSQL FUNCTIONS;
```

plsql functions
--------System functions--------
sqlcode(void)
sqlerrm(void)
get_columns(string,nestedtable<string>)
raise_application_error(int,string,bool)
set_env(string,string)
get_env(string)
put_line(string)
sqlerrm(int)

plsql functions
--------User defined functions--------
default.calc_stats(double, double, double, double)
default.grant_priv(string, string)
default.create_test(string, int, double)
default.insert_proc(string)
default.revoke_priv(string, string)
default.update_proc(string)
default.update_proc(string, string)
default.overload_test(int, int)
default.overload_test(int, string)

plsql functions
default.overload_test(string,string)
default.dynamic_sql_test(string,int,double)
default.overload_test_proc(int,int)
default.overload_test_proc(int,string)
default.overload_test_proc(string,string)
default.error_dynamic_sql_test(void)

- 例 2：查看某一 PL/SQL 函数/过程的详细信息

  使用 EXTENDED 关键字查看自定义过程 create_test 的详细信息

```
DESC PLSQL FUNCTION EXTENDED create_test;
```

description
Prototype:
PROCEDURE manual_pl_db2.create_test(v_name IN STRING, v_age IN INT, v_grade IN DOUBLE)
Text:
create or replace procedure create_test(v_name in string, v_age in int, v_grade in double)
is

description
begin
insert into zara values(v_name,v_age,v_grade)
end

- 例 3：删除函数/过程

删除自定义过程 overload_test_proc;

```
drop plsql procedure overload_test_proc;
```

再次查看已有的 PL/SQL 函数/过程，可以发现自定义过程 overload_test_proc 已被删除;

```
SHOW plsql functions;
```

plsql functions
--------System functions--------
sqlcode(void)
sqlerrm(void)
get_columns(string,nestedtable<string>)
raise_application_error(int,string,bool)
set_env(string,string)
get_env(string)
put_line(string)
sqlerrm(int)

plsql functions
--------User defined functions--------
default.calc_stats(double, double, double, double)
default.grant_priv(string, string)
default.create_test(string, int, double)
default.insert_proc(string)
default.revoke_priv(string, string)
default.update_proc(string)
default.update_proc(string, string)
default.overload_test(int, int)
default.overload_test(int, string)

plsql functions
default.overload_test(string,string)
default.dynamic_sql_test(string,int,double)
default.error_dynamic_sql_test(void)

- 例 4：查看正在运行的 PLSQL 程序的 SESSION ID

```
PS PLSQL;
```

- 例 5：终止正在运行的 PLSQL 程序

```
KILL PLSQL <SESSION_ID>
```

8.12.13.2. 预定义函数/过程/包的介绍

Inceptor 中一共有十个系统预定义的函数/过程，如下所示，我们将分章节介绍每一个函数/过程的具体内容及使用方法。

> 需要注意的是，预定义函数 get_columns(string,nestedtable<string>) 在 DB2 SQLPL 的环境里不可以使用。

- **set_env(string,string)**

  Inceptor 中，set_env 是一个过程，形参 enVar 的参数类型为 IN，数据类型为字符串，用来存放环境变量的名称；形参 value 的参数类型为 IN，数据类型为字符串，相应地用来存放环境变量的值。

```
PROCEDURE set_env(enVar IN STRING, value IN STRING)
```

- **get_env(string)**

  Inceptor 中，get_env 是一个函数，形参 enVar 的参数类型为 IN，数据类型为字符串，该函数用来返回 set_env 中环境变量的所对应的值。

```
FUNCTION get_env(enVar IN STRING) RETURN STRING
```

- **sqlcode(void)**

Inceptor 中，sqlcode() 是一个不带参数的函数，用来返回当异常发生时，当前异常的 Error code。

```
FUNCTION sqlcode() RETURN INT
```

- **sqlerrm(void)**

  Inceptor 中，不带参数的 sqlerrm()，用来返回当异常发生时，当前异常的 Error message。

```
FUNCTION sqlerrm() RETURN STRING
```

- **sqlerrm(int)**

  Inceptor 中，带参数的 sqlerrm()，用来返回既定 Error code 下的 Error message。

```
FUNCTION sqlerrm(errCode IN INT) RETURN STRING
```

- **put_line**

  Inceptor，put_line 是一个过程，形参 msg 的参数类型为 IN，数据类型为字符串，该过程用来打印出变量或者常量的值。

```
PROCEDURE dbms_output.put_line(msg IN STRING)
```

- **raise_application_error(int,string,bool)**

  在 Inceptor PL/SQL 中，可以使用预定义函数 raise_application_error，来抛出带有指定 error code、error message 的异常，目前第三个参数 keepExistError 是可选的且没有任 何作用。

```
FUNCTION raise_application_error(errorCode IN INT, msg IN STRING, keepExistError IN BOOL) RETURN
EXCEPTION
```

- **get_columns(string,nestedtable<string>)**

  Inceptor 中，get_columns 可以作函数使用，返回数据表中的列名。

```
FUNCTION get_columns(table IN STRING, columns IN ) RETURN
```

  Inceptor 中，get_columns 也可以作过程使用，返回数据表的列名。

```
PROCEDURE default.get_columns_test()
```

8.12.13.3. 预定义函数/过程的使用

8.12.13.3.1. set_env 与 get_env

- 例 1：set_env 与 get_env 的使用

```
!set plsqlUseSlash true
BEGIN
   DECLARE a STRING;
   DECLARE b STRING;
   -- 调用过程 set_env，定义一个名为 aa 的环境变量，值为 hello。
   set_env('aa','hello');
   -- 调用函数 get_env，获取环境变量 aa 的值，并赋值给 a。
   SET a = get_env('aa');
   -- 输出变量 a 的值，可以发现变量 a 与环境变量 aa 的值相同。
   PUT_LINE('the value of a is: '||a);
   set_env('bb','world');
   SET b = get_env('bb');
   PUT_LINE('the value of a is: '||b);
END;
/
```

输出结果为：

output
the value of a is: hello
the value of a is: world

8.12.13.3.2. sqlcode 与 sqlerrm

- 例 1：sqlcode 与 sqlerrm 的使用

```
!set plsqlUseSlash true
BEGIN
  DECLARE TYPE testrecord AS ROW anchor to row of transactions;
  DECLARE test_record testrecord;
  DECLARE v_code INT;
  DECLARE v_errm STRING;
    SELECT *
    INTO test_record
    FROM transactions
    WHERE price=12.12;
    -- 当异常发生时，获取当前异常的 Error code，并赋值给变量 v_code。
    dbms_output.put_line('error code is: ' ||sqlcode());
    -- 当异常发生时，获取当前异常的 Error message，并赋值给变量 v_errm。
    dbms_output.put_line('error message is: ' ||sqlerrm());
END;
/
```

输出结果为：

output
NOT FOUND: Code: 100 Message: No row was found for FETCH, UPDATE or DELETE; or the result of a query is an empty tableANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT "SELECT *
INTO test_record
FROM transactions
WHERE price=12.12;")
error code is: 0
error message is:

8.12.13.3.3. sqlerrm(int)

- 例 1：

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE test_errm(IN b INT)
BEGIN
  DECLARE errmessage STRING;
  SET errmessage = sqlerrm(b);
  PUT_LINE('error message is:'||errmessage);
END;
/
BEGIN
  test_errm(438);
  test_errm(100);
  test_errm(110);
END;
/
```

输出结果为：

output
error message is:Application raised error or warning with diagnostic text: ″″.
error message is:No row was found for FETCH, UPDATE or DELETE; or the result of a query is an empty table
error message is:Unknown error code: 110

8.12.13.3.4. PUT_LINE

- 在 Inceptor 中使用 PUT_LINE 函数时，如果程序正常运行，则 PUT_LINE 会统一打印到终端。
- 如果程序运行过程中出现了未被处理的异常，可以设置相关命令，将 PUT_LINE 正常打印出的内容连同异常栈一起打印到终端。

- 例 1：程序正常运行时，PUT_LINE 的打印

```
!set plsqlUseSlash true
-- 使用 put_line 函数,打印出 v3 的值。
-- 创建名为 put_line_test 的函数，形参 v1 的数据类型为整数型，返回值为双精度类型。
CREATE OR REPLACE FUNCTION put_line_test(v1 int)
   RETURNS DOUBLE
BEGIN
   DECLARE v2 DOUBLE;
   DECLARE v3 STRING;
   -- 使用 put_line 函数，打印出 null 值。
   put_line(null);
   DBMS_OUTPUT.PUT_LINE(v1);
   SET v2 = 2;
   SET v3 = "I'm a string.";
   -- 使用 put_line 函数，打印出变量 v2 的值。
   put_line(v2);
   DBMS_OUTPUT.PUT_LINE('v2: ' || v2);
   -- 使用 put_line 函数，打印出字符串 v2+v1。
   put_line('v2 + v1');
   DBMS_OUTPUT.PUT_LINE(v2 + v1);
   put_line(v3);
   SET v3 = null;
   DBMS_OUTPUT.PUT_LINE(v3);
   -- 函数返回值为 v1 和 v2 的乘积。
   return v2 * v1;
END;
/
BEGIN
  dbms_output.put_line("Executing put_line_test(1)");
  -- 使用 put_line 函数，打印出 put_line_test(1) 的全部值。
  put_line(put_line_test(1));
END;
```

输出结果:

output
Executing put_line_test(1)
null
1
2.0
v2: 2.0
v2 + v1
3.0
I’ m a string.
null
2.0

8.12.13.3.5. raise_application_error

- 例 1：raise_application_error 的使用

```
!set plsqlUseSlash true
BEGIN
  DECLARE num_tables INT;
  -- 查询表 transactions 的行数，并赋值给变量 num_tables。
  SELECT COUNT(*) INTO num_tables FROM transactions;
   -- 如果表 transactions 的行数小于 1000，则抛出异常。
   IF num_tables < 1000 THEN
     RAISE_APPLICATION_ERROR
      (-20101, 'Expecting at least 1000 tables');
    ELSE
      -- 否则，就输出表 transactions 的行数。
      dbms_output.put_line(num_tables);
    END IF;
END;
```

输出结果为：

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -20101] Expecting at least 1000
tables
**********************************************
System function raise_application_error (LINE -1, COLUMN -1, TEXT \"null\")
ANONYMOUS BLOCK (LINE 7, COLUMN 0, TEXT \"RAISE_APPLICATION_ERROR
(-20101, 'Expecting at least 1000 tables');\")
**********************************************
```

- 例 2：例 1 不抛出异常的情况

```
!set plsqlUseSlash true
BEGIN
   DECLARE num_tables INT;
  -- 查询表 transactions 的行数，并赋值给变量 num_tables。
  SELECT COUNT(*) INTO num_tables FROM transactions;
   -- 如果表 transactions 的行数小于 10，则抛出异常。
   IF num_tables < 10 THEN
     RAISE_APPLICATION_ERROR
      (-20101, 'Expecting at least 1000 tables');
    ELSE
      -- 否则，就输出表 transactions 的行数。
      dbms_output.put_line(num_tables);
    END IF;
END;
```

输出结果为：

output
20

8.12.14. 异常

Inceptor 中，每一条 SQL 语句执行后，都会返回一个 SQLCODE 和 SQLSTATE 的值，相应的 SQLCODE 和 SQLSTATE 的值，都可以反映出 SQL 语句的执行结果。

8.12.14.1. SQLCODE与SQLSTATE

SQLCODE 是每一条 SQL 语句执行后都会返回的值，反映 SQL 语句执行的情况。

- SQLCODE=0 该 SQL 执行成功
- SQLCODE>0 该 SQL 执行成功，但返回了一个警告
- SQLCODE<0 该 SQL 没有执行成功，并且返回了一个错误

- SQLCODE=100 未找到指定值

SQLSTATE 是一个长为 5 个字符的字符串，一般定义的比较笼统，没有 SQLCODE 更加具体。通常，几个 SQLCODE 可能对应一个 SQLSTATE。

- SQLSTATE= “00000” 成功
- SQLSTATE= “01XXX” 警告
- SQLSTATE= “02000” 未找到
- 其他所有值为错误

> 值得注意的是，要在 SQL PL 中使用 SQLCODE 和 SQLSTATE，我们必要先声明它们，具体内容可参考以下示例。

本节中，我们将以两个例子说明，当 SQL 语句成功执行，和 SQL 语句没有返回行的情况下，SQLCODE 和 SQLSTATE 返回值的情况。

- 例 1：在 SQL 语句成功执行的情况下

```
!set plsqlUseSlash true
BEGIN
  DECLARE v_time STRING;
  -- 分别声明 sqlcode 和 sqlstate，以及用来存放相应 SQLCODE 和 SQLSTATE 值的变量 sql_state，sql_code。
  DECLARE sqlstate, sql_state string;
  DECLARE sqlcode, sql_code int;
    -- 查询表 transactions 中价格为 12.13 的交易时间，并放进变量 v_time 里。
    SELECT trans_time INTO v_time FROM transactions WHERE price=12.13;
    -- 将执行上述 SQL 查询语句所获得的 sqlcode, sqlstate 的值赋值给变量 sql_code, sql_state。
    values(sqlcode, sqlstate) into sql_code, sql_state;
    -- 分别打印出变量 v_time，sql_code, sql_state 的值。
    PUT_LINE('the time is:'||v_time);
    PUT_LINE('the sqlcode is:'||sql_code);
    PUT_LINE('the sqlstate is:'||sql_state);
END;
/
```

输出结果为：

output
the time is:20140105100520
the sqlcode is:0
the sqlstate is:00000

本例中的 SQL 查询语句执行成功，返回了正确的值，同时可以看到，sqlcode 返回值为 0，sqlstate 返回值为 00000。

- 例 2：SQL 语句执行过程中，没有返回值

```
!set plsqlUseSlash true
BEGIN
  DECLARE v_time STRING;
  -- 分别声明 sqlcode 和 sqlstate，以及用来存放相应 SQLCODE 和 SQLSTATE 值的变量 sql_state，sql_code。
  DECLARE sqlstate, sql_state string;
  DECLARE sqlcode, sql_code int;
    -- 查询表 transactions 中价格为 12.00 的交易时间，并放进变量 v_time 里，由于表 transactions 中没有
price 为 12.00 的值，所以这里会有一个 NOT FOUND 的异常。
    SELECT trans_time INTO v_time FROM transactions WHERE price=12.00;
    -- 将执行上述 SQL 查询语句所获得的 sqlcode, sqlstate 的值赋值给变量 sql_code, sql_state。
    values(sqlcode, sqlstate) into sql_code, sql_state;
    -- 分别打印出变量 v_time, sql_code, sql_state 的值。
    PUT_LINE('the time is:'||v_time);
    PUT_LINE('the sqlcode is:'||sql_code);
    PUT_LINE('the sqlstate is:'||sql_state);
END;
/
```

输出结果为:

output
NOT FOUND: Code: 100 Message: No row was found for FETCH, UPDATE or DELETE; or the result of a query is an empty tableANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT "SELECT trans_time INTO v_time FROM transactions WHERE price=12.00;")
null
the sqlcode is:100
the sqlstate is:02000

本例中的 SQL 查询语句中出现了一个 NOT FOUND 的异常，没有返回任何的值，同时也可以看到，sqlcode 的值为 100，sqlstate 的值为 02000。

8.12.14.2. SIGNAL

在 SQL PL 程序中，我们可以自定义 SQLSTATE 的值，它必须是 5 个字符串，可以用关键字 SIGNAL 直接抛出一个指定值的 SQLSTATE，即在程序中设置一个条件，如果在运行过程中没有满足这个条件，那么就会抛出一个指定的警告或异常。

- 语法:

```
SIGNAL SQLSTATE [value]  <sqlstate> [SET MESSAGE_TEXT = <variable> or <diagnostic string
constant>];
```

例如:

```
SIGNAL SQLSTATE '60088';
```

例如:

```
SIGNAL SQLSTATE VALUE '70088';
```

例如:

```
SIGNAL SQLSTATE '80809' SET message_text='The input parameter must be lower than 100!';
```

- 例 1:

```
!set plsqlUseSlash true
-- 创建一个名为 myproc 的过程，参数类型为 IN，参数的数据类型为 INT。
CREATE OR REPLACE PROCEDURE myproc1(IN v_var01 INT)
BEGIN
    -- 如果参数 v_var01 大于 100，就抛出一个 SQLSTATE 为 70889 的异常，error message 的信息为 The input
parameter must be lower than 100!。
    IF v_var01 > 100 THEN
       SIGNAL SQLSTATE '70889' SET message_text='The input parameter must be lower than 100!';
    END IF;
END;
/
CALL myproc1(200);
```

输出结果为:

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -438] The input parameter must be
lower than 100!
**********************************************
PROCEDURE 'myproc1' (LINE 5, COLUMN 16, TEXT \"SIGNAL SQLSTATE '70889' SET message_text='The
input parameter must be lower than 100!';\")
 (LINE 1, COLUMN 5, TEXT \"myproc1(200)\")
**********************************************
```

8.12.14.3. 异常分类

8.12.14.3.1. 预定义异常

当存储过程中的语句返回的 SQLSTATE 值超过 00000 的时候，就表明在存储过程中产生了一个异常（condition），它表示出现了数据没有找到，错误的状况或者出现了警告。这三类系统预定义的异常，分别为 NOT FOUND，SQLEXCEPTIOIN，SQLWARNING:

- NOT FOUND: SQLCODE 的值为 +100 且 SQLSATE 的值为 02000。这个异常通常在 SELECT 没有返回行的时候出现。
- SQLWARNING: SQLCODE 的值大于 0 且 SQLSTATE 的值以 01 开头。
- SQLEXCEPTIOIN: SQLCODE 值小于 0 且 SQLSTATE 的值为 02000，01XXX 之外的所有值。

8.12.14.3.2. 自定义异常

Inceptor 中，我们可以自定义一个 SQLSTATE 的值，并使用关键字 SIGNAL，抛出一个带有任意指定值的 SQLSTATE 的异常。其中对于 NOT FOUND 或者 SQLWARNING 异常，默认情况下，Inceptor 会报出关于异常的信息，并将继续执行异常所在语句的下一条语句，只有当发生了 SQLEXCEPTIOIN 类型的异常时，Inceptor 才会从异常发生处跳出，并抛出一个异常。

语法

- 声明自定义异常:

```
DECLARE condition_name CONDITION FOR SQLSTATE <value>;
```

- 抛出异常:

```
SIGNAL condition_name;
```

实例

- 例 1：自定义一个 SQLWARNING 类型的异常

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
  -- 声明一个 SQLSTATE 值为 01089 的异常 C1，由于 SQLSTATE 的值以 01 开头的异常为 SQLWARNING
类型的异常，所以异常 C1 是一个 SQLWARNING 类型的异常。
  DECLARE C1 CONDITION FOR SQLSTATE '01089';
  BEGIN
     DECLARE v_price DOUBLE;
     SELECT price INTO v_price
     FROM transactions WHERE trans_id=v_id;
     -- 查询并打印出表 transactions 中交易号等于参数 v_id 值所对应的价格。
     PUT_LINE('the price is:'||v_price);
  END;
  -- 抛出异常 C1。
  SIGNAL C1;
END;
/
CALL condition_test('929634984');
```

输出结果为：

output
the price is:11.11
WARNING: Code: 438 Message: PROCEDURE 'condition_test' (LINE 13, COLUMN 7, TEXT "SIGNAL C1;")

- 例 2：自定义一个 SQLEXCEPTIOIN 类型的异常

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
  -- 声明一个 SQLSTATE 值为 76089 的异常 C1，即一个 SQLEXCEPTIOIN 类型的异常。
  DECLARE C1 CONDITION FOR SQLSTATE '76089';
  BEGIN
     DECLARE v_price DOUBLE;
     SELECT price INTO v_price
     FROM transactions WHERE trans_id=v_id;
     -- 查询并打印出表 transactions 中交易号等于参数 v_id 值所对应的价格。
     PUT_LINE('the price is:'||v_price);
  END;
  -- 抛出异常 C1。
  SIGNAL C1;
END;
/
CALL condition_test('929634984');
```

输出结果为：

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -438]
**********************************************
PROCEDURE 'condition_test' (LINE 13, COLUMN 7, TEXT \"SIGNAL C1;\")
 (LINE 1, COLUMN 5, TEXT \"condition_test('929634984')\")
**********************************************
```

8.12.14.3.3. 异常处理

为了响应和处理存储过程中出现的异常，我们必须在存储过程体中声明异常处理器（condition handler），它可以决定存储过程怎样响应一个或者多个已定义的异常或者预定义异常。 如果产生了 NOT FOUND 或者 SQLWARNING 异常，并且没有为这个异常定义异常处理器，那么默认就会忽略这个异常，并且将控制流转向下一个语句。如果产生了 SQLEXCEPTION 异常，并且没有为这个异常定义异常处理器，那么存储过程就会失

败，并且会将控制流返回调用者。

- 语法：

```
DECLARE handler-type HANDLER FOR condition_name;
```

handler-type，即异常处理器类型，主要为以下几种类型：

- CONTINUE 在处理器操作完成之后，会继续执行产生这个异常语句之后的下一条语句。
- EXIT 在处理器操作完成之后，存储过程会终止，并将控制返回给调用者。
- UNDO 在处理器操作执行之前，DB2 会回滚存储过程中执行的 SQL 操作。在处理器操作完成之后，存储过程会终止，并将控制返回给调用者。

> Inceptor 中暂不支持 UNDO 异常处理器类型，即目前只支持使用 CONTINUE，EXIT 的异常处理行为去处理 Inceptor 中的自定义异常和预定义异常。

8.12.14.3.4. 实例

自定义异常的处理

- 例 1.1：自定义一个 SQLEXCEPTIOIN 类型的异常

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
  -- 自定义一个 SQLEXCEPTIOIN 类型的异常 C1， SQLSTATE 值为 76089。
  DECLARE C1 CONDITION FOR SQLSTATE '76089';
  BEGIN
     DECLARE v_price DOUBLE;
     SELECT price INTO v_price
     FROM transactions WHERE trans_id=v_id;
     PUT_LINE('the price is:'||v_price);
  END;
  -- 抛出异常 C1。
  SIGNAL C1;
END;
/
CALL condition_test('929634984');
```

输出结果为：

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -438]
**********************************************
PROCEDURE 'condition_test' (LINE 12, COLUMN 7, TEXT \"SIGNAL C1;\")
 (LINE 1, COLUMN 5, TEXT \"condition_test('929634984')\")
**********************************************
```

- 例 1.2：为一个 SQLEXCEPTIOIN 类型的异常声明一个 CONTINUE 类型的异常处理器

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
  -- 自定义一个 SQLEXCEPTIOIN 类型的异常 C1， SQLSTATE 值为 76089。
  DECLARE C1 CONDITION FOR SQLSTATE '76089';
  -- 为异常 C1 声明一个一个 CONTINUE 类型的异常处理器。
  DECLARE CONTINUE HANDLER FOR C1
  BEGIN
     DECLARE v_price DOUBLE;
     SELECT price INTO v_price
     FROM transactions WHERE trans_id=v_id;
     PUT_LINE('the price is:'||v_price);
  END;
  -- 抛出异常 C1。
  SIGNAL C1;
END;
/
CALL condition_test('929634984');
```

输出结果为:

output
the price is:11.11

- 例 2.1：自定义一个 SQLWARNING 类型的异常

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
  -- 声明一个 sqlstate 以 01 开头的 SQLWARNING 类型的异常 C1。
  DECLARE C1 CONDITION FOR SQLSTATE '01089';
  BEGIN
     DECLARE v_price DOUBLE;
     SELECT price INTO v_price
     FROM transactions WHERE trans_id=v_id;
     PUT_LINE('the price is:'||v_price);
  END;
  -- 抛出异常 C1。
  SIGNAL C1;
  PUT_LINE('there is a condition');
END;
/
CALL condition_test('594819547');
```

输出结果为:

output
the price is:6.36
WARNING: Code: 438 Message: PROCEDURE 'condition_test' (LINE 12, COLUMN 7, TEXT "SIGNAL C1;")
there is a condition

- 例 2.2：为一个 SQLWARNING 类型的异常，声明一个 EXIT 类型的异常处理器

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
  -- 声明一个 sqlstate 以 01 开头的 SQLWARNING 类型的异常 C1。
  DECLARE C1 CONDITION FOR SQLSTATE '01089';
  -- 为 SQLWARNING 类型的异常 C1，声明一个 EXIT 类型的异常处理器。
  DECLARE EXIT HANDLER FOR C1
  BEGIN
     DECLARE v_price DOUBLE;
     SELECT price INTO v_price
     FROM transactions WHERE trans_id=v_id;
     PUT_LINE('the price is:'||v_price);
  END;
  -- 抛出异常 C1。
  SIGNAL C1;
  PUT_LINE('there is a condition');
END;
/
CALL condition_test('594819547');
```

输出结果为：

output
the price is:6.36

对比实例 2.1，我们可以看到在实例 2.2 中，为 SQLWARNING 类型的异常 C1，声明了一个 EXIT 类型的异常处理器，所以本例中程序直接从发生异常所在处退出，而不会执行异常发生处的下一条语句。

预定义异常的处理

- 例 3.1：NOT FOUND 类型的预定义异常

```
!set plsqlUseSlash true
BEGIN
   DECLARE v_amount DOUBLE;
    BEGIN
      DECLARE v_price DOUBLE;
      -- 该语句的返回结果不存在，因此有一个 NOT FOUND 异常
      SELECT price INTO v_price
      FROM transactions WHERE trans_id=111111111;
      PUT_LINE('the price is:'||v_price);
      SELECT amount INTO v_amount
      FROM transactions WHERE price=6.36;
      PUT_LINE('the amount is:'||v_amount);
    END
END;
/
```

输出结果为：

output
NOT FOUND: Code: 100 Message: No row was found for FETCH, UPDATE or DELETE; or the result of a query is an empty tableANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT "SELECT price INTO v_price
FROM transactions WHERE trans_id=111111111;")
null
the amount is:800.0

本例中，分别在内外语句块中，实现查询并打印出相应的价格和金额，由于内语句块中，SELECT 语句的 WHERE 子条件的交易号，并不存在于表 transactions 中，所以此处 SELECT 语句没有返回行，即是一个 NOT FOUND 类型的异常。

- 例 3.2：为 NOT FOUND 类型的预定义异常声明一个 EXIT 类型的异常处理器

```
!set plsqlUseSlash true
BEGIN
  DECLARE v_price DOUBLE;
  DECLARE v_amount DOUBLE;
  -- 为 NOT FOUND 类型的异常声明一个 EXIT 类型的异常处理器。
  DECLARE EXIT HANDLER FOR NOT FOUND
  BEGIN
      PUT_LINE('NOT FOUND');
  END;
      SELECT price INTO v_price
      FROM transactions WHERE trans_id=111111111;
      PUT_LINE('the price is:'||v_price);
      SELECT amount INTO v_amount
      FROM transactions WHERE price=6.36;
      PUT_LINE('the amount is:'||v_amount);
END;
/
```

输出结果为：

output
NOT FOUND

- 例 4.1：SQLEXCEPTIOIN 类型的预定义异常

```
!set plsqlUseSlash true
set plsql.compile.dml.check.semantic=false;
BEGIN
   DECLARE v_amount DOUBLE;
    BEGIN
      DECLARE v_price DOUBLE;
      SELECT price INTO v_price
      FROM t1 WHERE trans_id=111111111;
      -- 查询表 t1 中的数据，此处数据库中表 t1 不存在，所以这里会有一个 SQLEXCEPTIOIN 类型的异常。
      PUT_LINE('the price is:'||v_price);
      SELECT amount INTO v_amount
      FROM transactions WHERE price=6.36;
      PUT_LINE('the amount is:'||v_amount);
    END;
END;
/
```

输出结果为：

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -204] Table or view does not exist:
t1
**********************************************
ANONYMOUS BLOCK (LINE 6, COLUMN 5, TEXT \"SELECT price INTO v_price
FROM t1 WHERE trans_id=111111111;\")
**********************************************
```

- 例 4.2：为 SQLEXCEPTIOIN 类型的预定义异常声明一个 CONTINUE 类型的异常处理器

```
!set plsqlUseSlash true
set plsql.compile.dml.check.semantic=false;
BEGIN
   DECLARE v_price DOUBLE;
   DECLARE v_amount DOUBLE;
   -- 为 sqlexception 类型的异常，声明一个 CONTINUE 类型的异常处理器。
   DECLARE CONTINUE HANDLER FOR sqlexception
    BEGIN
      PUT_LINE('sql exception');
    END;
      SELECT price INTO v_price FROM t1 WHERE trans_id=111111111;
      PUT_LINE('the price is:'||v_price);
      SELECT amount INTO v_amount
      FROM transactions WHERE price=6.36;
      PUT_LINE('the amount is:'||v_amount);
END;
/
```

输出结果为:

output
sql exception
null
the amount is:800.0

本例中为 SQLEXCEPTIOIN 类型的预定义异常声明了一个 CONTINUE 类型的异常处理器，所以程序在异常发生所在处，继续执行下一条语句。

8.12.14.4. RESIGNAL

与 SIGNAL 语句类似，RESIGNAL 语句可以用来直接抛出一个指定值的 SQLSTATE，但是 RESIGNAL 语句只能用在 CONDITION 类型的处理器之中。

8.12.14.4.1. 语法

- 直接抛出具有指定值的 SQLSTATE

```
RESIGNAL SQLSTATE [VALUE] <sqlstate> [SET MESSAGE_TEXT = <variable> or <diagnostic string
constant>];
```

- 直接声明一个自定义的异常

```
RESIGNAL <condition name> [SET MESSAGE_TEXT = <variable> or <diagnostic string constant>];
```

8.12.14.4.2. 用例

- 例 1:

```
!set plsqlUseSlash true
-- 创建一个名为 exp_proc 的存储过程，参数 b 的类型为 OUT，其数据类型为整数类型。
CREATE OR REPLACE PROCEDURE exp_proc(out b int)
  BEGIN
    -- 声明一个名为 c 的自定义异常，sqlstate 的值为 54321。
    DECLARE c CONDITION FOR sqlstate '54321';
    -- 对值为 34567 的 sqlstate 声明一个 CONTINUE 类型的异常处理器。
    DECLARE CONTINUE HANDLER FOR sqlstate '34567'
  BEGIN
   -- 为自定义异常 c 声明一个 CONTINUE 类型的异常处理器。
   DECLARE CONTINUE HANDLER for c
   BEGIN
     -- 使用关键字 resignal 抛出一个 sqlstate 的值为 34567 的异常。
     resignal sqlstate value '34567';
   END;
  -- 使用关键字 signal 抛出自定义异常 c。
  signal c;
  END;
  -- 输出该存储过程内的 sqlcode 的值。
  dbms_output.put_line('error code is: ' ||sqlcode());
END;
/
BEGIN
  DECLARE b INT default 0;
  DECLARE CONTINUE HANDLER FOR sqlstate '34567'
  BEGIN
    SET b = -10;
  END;
  -- 调用存储过程 exp_proc。
  CALL exp_proc(b);
END;
/
```

输出结果为：

output
error code is: 0

由于在存储过程 exp_proc 内声明了 CONTINUE 类型的异常处理器，在调用存储过程中，遇到异常，程序会继续向下执行，因此 sqlcode 的值为 0。

8.12.14.5. DEBUG

Inceptor 中如果程序运行中出现了未被处理的异常，则可以在终端的报错中查看异常栈的信息，了解异常发生的位置所在。

- 例 1：

```
!set plsqlUseSlash true
-- 创建存储过程 p1，在存储过程 p1 内声明一个名为 ex 的自定义异常，该异常的 sqlstate 为 76089，异常类型为
SQLEXCEPTION。
CREATE OR REPLACE PROCEDURE p1(value string)
BEGIN
   DECLARE ex CONDITION FOR SQLSTATE '76089';
   set value =  'p1: ' || value;
   dbms_output.put_line(value);
     -- 抛出异常 ex
     SIGNAL ex;
END;
/
-- 创建过程 p2。
CREATE OR REPLACE PROCEDURE p2(value string)
BEGIN
  set value = 'p2: ' || value;
  dbms_output.put_line(value);
  -- 在过程 p2 内调用过程 p1。
  p1(value);
END;
/
-- 创建过程 p3。
CREATE OR REPLACE PROCEDURE p3(value string)
BEGIN
  set value = 'p3: ' || value;
  dbms_output.put_line(value);
  -- 在过程 p3 内调用过程 p2。
  p2(value);
END;
/
BEGIN
DECLARE value string;
DECLARE id_v int;
DECLARE ret_v int;
  SET value = 'init value';
  dbms_output.put_line(value);
  -- 最后执行过程 p3 的内容。
  p3(value);
END;
/
```

输出结果为:

```
EXECUTION FAILED: Task EXEC PLSQL error PLException: [Error -438]
**********************************************
PROCEDURE 'p1' (LINE 7, COLUMN 7, TEXT \"SIGNAL ex;\")
PROCEDURE 'p2' (LINE 6, COLUMN 0, TEXT \"p1(value);\")
PROCEDURE 'p3' (LINE 6, COLUMN 0, TEXT \"p2(value);\")
ANONYMOUS BLOCK (LINE 8, COLUMN 0, TEXT \"p3(value);\")
**********************************************
```

从异常栈的信息中，我们可以看到，直接导致程序运行失败的位置是在最后一段 SQL PL 语句块中，调用了 p3 这个过程，而根本的原因在过程 p1 内直接抛出异常，没有处理该异常。

- 例 2：为例 1 加上一个 CONTINUE 类型的异常处理器，使得程序在运行过程中遇到异常，也可以继续执行

```
!set plsqlUseSlash true
CREATE OR REPLACE PROCEDURE p1(value string)
BEGIN
   -- 为自定义异常 ex 声明一个 CONTINUE 类型的异常处理器。
   DECLARE ex CONDITION FOR SQLSTATE '76089';
   DECLARE CONTINUE HANDLER FOR ex
   begin
   set value =  'p1: ' || value;
   dbms_output.put_line(value);
   end;
     SIGNAL ex;
END;
/
CREATE OR REPLACE PROCEDURE p2(value string)
BEGIN
  set value = 'p2: ' || value;
  dbms_output.put_line(value);
  p1(value);
END;
/
CREATE OR REPLACE PROCEDURE p3(value string)
BEGIN
  set value = 'p3: ' || value;
  dbms_output.put_line(value);
  p2(value);
END;
/
BEGIN
DECLARE value string;
DECLARE id_v int;
DECLARE ret_v int;
  SET value = 'init value';
  dbms_output.put_line(value);
  p3(value);
END;
/
```

输出结果为:

output
init value
p3: init value
p2: p3: init value
p1: p2: p3: init value

8.12.15. 注意事项

8.12.15.1. 函数/存储过程的版本兼容

Inceptor PL/SQL 在特定的版本上会对后端引擎进行比较大的改进优化，因此在某些版本上的某些已创建的 PL/SQL 函数和过程需要升级，可由星环科技提供的自动升级工具完成，以完全保证前后版本的兼容性。

8.12.15.2. 方言的选择

Inceptor 中默认情况下支持 ORACLE dialect，也就是在连接到 Inceptor 时，就可以直接创建 PL/SQL 语句块，但对于 DB2 dialect 的支持，需要手动打开。

- 以 Beeline 方式连接 Inceptor 时，可通过以下命令，手动打开对 DB2 dialect 的支持。

  客户端

```
!set plsqlClientDialect db2
```

服务器端

```
set plsql.server.dialect=db2;
```

8.12.15.3. 分号的支持

- 以 Beeline 方式连接 Inceptor 时，一旦打开手动 DB2 dialect 支持，则 Inceptor 对于分号的支持将自动打开，且不能更改。
- 也就是说，在 Inceptor 中创建 SQL PL 语句块时，必须在每一个完整的 SQL PL 语句块后面,加上一个新行,这行只包含 ‘/’ 字符,否则 Inceptor 不会执行当前的语句块，也不会执行该语句块之前的任何 SQL PL 语句。

8.12.15.4. PL/SQL 中结果的打印

Inceptor 中，PL/SQL 中直接执行 SELECT 语句是默认不打印结果的。要想查看结果,需要通过 set plsql.show.sqlresults=true; 来打印结果。

- 默认情况下，查询表 transactions 中价格为 12.13 的交易时间和交易类型

```
!set plsqlUseSlash true
BEGIN
   SELECT trans_time,trans_type
     from transactions
      WHERE price=12.13;
END;
/
```

输出结果为：

output

- 手动设置结果打印，查询表 transactions 中价格为 12.13 的交易时间和交易类型

设置 plsql.show.sqlresults 的值为 true，set plsql.show.sqlresults=true

```
!set plsqlUseSlash true
set plsql.show.sqlresults=true;
BEGIN
   SELECT trans_time,trans_type
    from transactions
    WHERE price=12.13;
END;
/
```

输出结果为：

output
20140105100520
b

9. 附录

9.1. 字段规范

Inceptor支持在多种字段中使用中文。

字段	是否支持中文	最大长度
database name	yes	128
database comment	yes	4000
database location	yes	4000
database properties	no	
table name	yes	128
table location	yes	4000
table properties key	no	
table properties value	yes	4000
table comment	yes	4000
table column name	yes	128
table column comment	yes	256
table column default value	yes	256
partition column key	yes	128
partition column value	yes	127
range partition column values	no	
alias	yes	无限制
dblink name	yes	255
dblink user	no	
dblink pwd	no	
dblink service	no	
plsql variable	yes	no limit
plsql function name	yes	128
plsql procedure name	yes	128
plsql function variable	yes	no limit
plsql package name	yes	128
plsql pakcage body name	yes	128

9.2. 关键字

9.2.1. 基本概念

在 Inceptor 的sql语法中，有特殊意义的词叫做关键词。关键词由保留关键词和非保留关键词组成。

- 保留关键词：属于关键词，而且不能用于命名任何实体或者对象，例如（但不仅限于）列，表，数据库，分区，函数，游标，数据库连接，（除了表以外的）别名。

  保留关键词使用方法（不推荐）：

  ```
  -- 其中begin，and，as 都是保留关键词。
  create table `begin`.`and`(`as` int);
  ```

- 非保留关键词：虽然是关键词，但是可以用于命名大部分实体或者对象，例如（但不仅限于）列，表，数据库，分区，函数，游标，数据库连接，（除了表以外的）别名。但是部分实体命名也不能使用非保留关键词，详见非保留关键词的受限范围。

9.2.2. 非保留关键词的受限范围

非保留关键词不能用于以下场景

> 非保留关键词不能用之处，保留关键词必定不能用

- 表和子查询的别名

  ```
  -- "log" 是非保留关键词，只有作为表的别名时候不能使用它。
  select log log from log log where log = 1;
  ```

- with as 的表名

  ```
  -- “int” 是非保留关键词。第一个“int” 是with as 的表名，即使是非保留关键词也不能使用，但是后面的“int”
  是表名，此处可以使用非保留关键词。
  with int as (select * from system.dual) select * from int;
  ```

- plsql 自定义类型

  ```
  -- “int” 处不能使用非保留关键词。
  type int is
  ```

- package 名

  ```
  -- 第一个“int” 是数据库名，可以用非保留关键词，第二个“int” 是package 名，不能使用非保留关键词。
  create or replace package int.int is
  pv int
  procedure pbf_increase(i out nocopy int, dummy nocopy date)
  end;
  ```

- plsql goto label

  ```
  goto int
  ```

- 窗口函数over后面的描述

```
SELECT *, Row_Number() over int FROM employee;
```

9.2.3. 关键词列表

Inceptor 支持Oracle、DB2、TD方言。在不同的方言下，保留关键词和非保留关键词有所不同。

9.2.3.1. 保留关键词

	oracle	db2	TD	引入版本
BOTH	—	—	保留关键词	5.0引入
COLUMN_ALIAS_LIST	—	—	保留关键词	5.0引入
COLUMN_LIST	—	—	保留关键词	5.0引入
DATADEBUG	保留关键词	保留关键词	保留关键词	5.0引入
DECIMAL64	保留关键词	保留关键词	保留关键词	5.0引入
INCLUDE_NULLS	—	—	保留关键词	5.0引入
LEADING	—	—	保留关键词	5.0引入
TAILING	—	—	保留关键词	5.0引入
UNPIVOT_COLUMN	—	—	保留关键词	5.0引入
VALUE_COLUMNS	—	—	保留关键词	5.0引入
__RUNTIME__	保留关键词	保留关键词	保留关键词	
ADD	保留关键词	保留关键词	保留关键词	
ALL	保留关键词	保留关键词	保留关键词	
ALLOCATE	—	保留关键词	保留关键词	
AND	保留关键词	保留关键词	保留关键词	
ANTISEMI	保留关键词	保留关键词	保留关键词	
ANY	保留关键词	保留关键词	保留关键词	
APPEND	保留关键词	保留关键词	保留关键词	
APPPROPERTIES	保留关键词	保留关键词	保留关键词	
AS	保留关键词	保留关键词	保留关键词	
ASSOCIATE	—	保留关键词	保留关键词	
ATOMIC	—	保留关键词	保留关键词	
AUTONOMOUS	—	保留关键词	保留关键词	
AUTONOMOUS_TRANSACTION	保留关键词	—	—	
BATCHINSERT	保留关键词	保留关键词	保留关键词	

BATCHUPDATE	保留关键词	保留关键词	保留关键词	
BATCHVALUES	保留关键词	保留关键词	保留关键词	
BEGIN	保留关键词	保留关键词	保留关键词	
BETWEEN	保留关键词	保留关键词	保留关键词	
BFILE	保留关键词	保留关键词	保留关键词	
BLACKLIST	保留关键词	保留关键词	保留关键词	
BLOB	保留关键词	保留关键词	保留关键词	
BREAK	保留关键词	保留关键词	保留关键词	
BULK	保留关键词	保留关键词	保留关键词	
BY	保留关键词	保留关键词	保留关键词	
CALL	保留关键词	保留关键词	保留关键词	
CALLED	—	保留关键词	保留关键词	
CASE	保留关键词	保留关键词	保留关键词	
CAST	保留关键词	保留关键词	保留关键词	
CHAR	保留关键词	保留关键词	保留关键词	
CLOB	保留关键词	保留关键词	保留关键词	
CLOSE	保留关键词	保留关键词	保留关键词	
CLUSTER	保留关键词	保留关键词	保留关键词	
COLLECT	保留关键词	保留关键词	保留关键词	
COLUMN	保留关键词	保留关键词	保留关键词	
COMMENT	保留关键词	保留关键词	保留关键词	
COMMIT	保留关键词	保留关键词	保留关键词	
COMMITTED	保留关键词	保留关键词	保留关键词	
COMPACTIONS	保留关键词	保留关键词	保留关键词	
CONNECT	保留关键词	保留关键词	保留关键词	
CONTAINS	—	保留关键词	保留关键词	
CONTINUE	保留关键词	保留关键词	保留关键词	
CREATE	保留关键词	保留关键词	保留关键词	
CREATE_PROC	保留关键词	保留关键词	保留关键词	
CROSS	保留关键词	保留关键词	保留关键词	
CUBE	保留关键词	保留关键词	保留关键词	
CURRENT	保留关键词	保留关键词	保留关键词	
CURRENT_TIMESTAMP	保留关键词	保留关键词	保留关键词	
CURSOR	保留关键词	保留关键词	保留关键词	
DATABASE	保留关键词	保留关键词	保留关键词	

DB2_RETURN_STATUS	—	保留关键词	—	
DB2_SQL_NESTING_LEVEL	—	保留关键词	—	
DB2_TOKEN_STRING	—	保留关键词	—	
DEC	保留关键词	保留关键词	保留关键词	
DECLARE	保留关键词	保留关键词	保留关键词	
DEFINITION	—	保留关键词	保留关键词	
DESC_PROC	保留关键词	保留关键词	保留关键词	
DETERMINISTIC	—	保留关键词	保留关键词	
DIAGNOSTICS	—	保留关键词	保留关键词	
DISTINCT	保留关键词	保留关键词	保留关键词	
DO	—	保留关键词	保留关键词	
DOCVALUES	保留关键词	保留关键词	保留关键词	
DROP	保留关键词	保留关键词	保留关键词	
DROP_PROC	保留关键词	保留关键词	保留关键词	
DYNAMIC	—	保留关键词	保留关键词	
ELSE	保留关键词	保留关键词	保留关键词	
ELSEIF	保留关键词	保留关键词	保留关键词	
ELSIF	保留关键词	保留关键词	保留关键词	
END	保留关键词	保留关键词	保留关键词	
ERRORS	保留关键词	保留关键词	保留关键词	
ESCAPE	保留关键词	保留关键词	保留关键词	
EXCEL	保留关键词	保留关键词	保留关键词	
EXCEPT	保留关键词	保留关键词	保留关键词	
EXCEPTION	保留关键词	保留关键词	保留关键词	
EXCEPTION_INIT	保留关键词	—	—	
EXCHANGE	保留关键词	保留关键词	保留关键词	
EXCLUDE	保留关键词	保留关键词	保留关键词	
EXEC_PROC	保留关键词	保留关键词	保留关键词	
EXECUTE	保留关键词	保留关键词	保留关键词	
EXIT	保留关键词	—	—	
EXTENDED	保留关键词	保留关键词	保留关键词	
EXTRACT	保留关键词	保留关键词	保留关键词	
FOLLOWING	保留关键词	保留关键词	保留关键词	
FOR	保留关键词	保留关键词	保留关键词	

FORALL	保留关键词	—	—	
FOUND	—	保留关键词	保留关键词	
FROM	保留关键词	保留关键词	保留关键词	
FUNCTION	保留关键词	保留关键词	保留关键词	
GET	—	保留关键词	保留关键词	
GOTO	保留关键词	保留关键词	保留关键词	
GRANT	保留关键词	保留关键词	保留关键词	
GROUP	保留关键词	保留关键词	保留关键词	
HANDLER	—	保留关键词	保留关键词	
HAVING	保留关键词	保留关键词	保留关键词	
IDENTIFIED	保留关键词	保留关键词	保留关键词	
IF	保留关键词	保留关键词	保留关键词	
IMMEDIATE	保留关键词	保留关键词	保留关键词	
IN	保留关键词	保留关键词	保留关键词	
INCLUDE	保留关键词	保留关键词	保留关键词	
INHERIT	—	保留关键词	保留关键词	
INITIAL	保留关键词	保留关键词	保留关键词	
INITRANS	保留关键词	保留关键词	保留关键词	
INOUT	保留关键词	保留关键词	保留关键词	
INSERT	保留关键词	保留关键词	保留关键词	
INTERSECT	保留关键词	保留关键词	保留关键词	
INTO	保留关键词	保留关键词	保留关键词	
IS	保留关键词	保留关键词	保留关键词	
ISOLATION	保留关键词	保留关键词	保留关键词	
ITERATE	—	保留关键词	保留关键词	
JOBPROPERTIES	保留关键词	保留关键词	保留关键词	
JOIN	保留关键词	保留关键词	保留关键词	
KEYEDBY	保留关键词	保留关键词	保留关键词	
LEAVE	—	保留关键词	保留关键词	
LESS	保留关键词	保留关键词	保留关键词	
LIKE	保留关键词	保留关键词	保留关键词	
LINK	保留关键词	保留关键词	保留关键词	
LOCATOR	—	保留关键词	保留关键词	
LOCATORS	—	保留关键词	保留关键词	
LOCK	保留关键词	保留关键词	保留关键词	

LONG	保留关键词	保留关键词	保留关键词	
LOOP	保留关键词	保留关键词	保留关键词	
MACRO	保留关键词	保留关键词	保留关键词	
MAP	保留关键词	保留关键词	保留关键词	
MATCHED	保留关键词	保留关键词	保留关键词	
MAXEXTENTS	保留关键词	保留关键词	保留关键词	
MAXTRANS	保留关键词	保留关键词	保留关键词	
MAXVALUE	保留关键词	保留关键词	保留关键词	
MESSAGE_TEXT	—	保留关键词	保留关键词	
MINEXTENTS	保留关键词	保留关键词	保留关键词	
MODIFIES	—	保留关键词	保留关键词	
MORE	保留关键词	保留关键词	保留关键词	
NATURAL	保留关键词	保留关键词	保留关键词	
NO_REWRITE	保留关键词	保留关键词	保留关键词	
NONE	保留关键词	保留关键词	保留关键词	
NOT	保留关键词	保留关键词	保留关键词	
NULL	保留关键词	保留关键词	保留关键词	
NUMBER	保留关键词	—	—	
NUMERIC	保留关键词	保留关键词	保留关键词	
NVARCHAR	保留关键词	保留关键词	保留关键词	
OF	保留关键词	保留关键词	保留关键词	
OFF	保留关键词	保留关键词	保留关键词	
OFFSET	保留关键词	保留关键词	保留关键词	
ON	保留关键词	保留关键词	保留关键词	
OPEN	保留关键词	保留关键词	保留关键词	
OR	保留关键词	保留关键词	保留关键词	
ORDER	保留关键词	保留关键词	保留关键词	
OVER	保留关键词	保留关键词	保留关键词	
PARTIALSCAN	保留关键词	保留关键词	保留关键词	
PCTFREE	保留关键词	保留关键词	保留关键词	
PCTUSED	保留关键词	保留关键词	保留关键词	
PLSQL	保留关键词	保留关键词	保留关键词	
PRAGMA	保留关键词	—	—	
PRECEDING	保留关键词	保留关键词	保留关键词	
PREPARE	—	保留关键词	保留关键词	

PRESERVE	保留关键词	保留关键词	保留关键词	
QUALIFY	保留关键词	保留关键词	—	
RAISE	保留关键词	—	—	
RECORD	保留关键词	—	—	
RECORDS	保留关键词	保留关键词	保留关键词	
REDUCE	保留关键词	保留关键词	保留关键词	
REJECT	保留关键词	保留关键词	保留关键词	
RENAME	保留关键词	保留关键词	保留关键词	
REPLICATION	保留关键词	保留关键词	保留关键词	
RESIGNAL	—	保留关键词	保留关键词	
RESULT_SET_LOCATOR	—	保留关键词	保留关键词	
RETURN	保留关键词	保留关键词	保留关键词	
RETURN_STATUS	—	保留关键词	—	
REVOKE	保留关键词	保留关键词	保留关键词	
REWRITE	保留关键词	保留关键词	保留关键词	
ROLLBACK	保留关键词	保留关键词	保留关键词	
ROLLUP	保留关键词	保留关键词	保留关键词	
ROW	保留关键词	保留关键词	保留关键词	
ROW_COUNT	—	保留关键词	保留关键词	
ROWTYPE	保留关键词	保留关键词	保留关键词	
SAVEPOINT	—	保留关键词	保留关键词	
SELECT	保留关键词	保留关键词	保留关键词	
SERIALIZABLE	保留关键词	保留关键词	保留关键词	
SERIALLY_REUSABLE	保留关键词	—	—	
SESSION	—	保留关键词	保留关键词	
SESSIONWINDOW	保留关键词	保留关键词	保留关键词	
SET	—	保留关键词	—	
SPECIAL	—	保留关键词	保留关键词	
SPECIFIC	—	保留关键词	保留关键词	
SQLEXCEPTION	—	保留关键词	保留关键词	
SQLWARNING	—	保留关键词	保留关键词	
START	保留关键词	保留关键词	保留关键词	
STORAGE	保留关键词	保留关键词	保留关键词	
STREAMS	保留关键词	保留关键词	保留关键词	
SUBSTRING	保留关键词	保留关键词	保留关键词	

SYSDATE	—	保留关键词	保留关键词	
SYSTIME	保留关键词	保留关键词	保留关键词	
SYSTIMESTAMP	保留关键词	保留关键词	保留关键词	
TABLE	保留关键词	保留关键词	保留关键词	
TABLESAMPLE	保留关键词	保留关键词	保留关键词	
TABLESPACE	保留关键词	保留关键词	保留关键词	
THAN	保留关键词	保留关键词	保留关键词	
THEN	保留关键词	保留关键词	保留关键词	
TO	保留关键词	保留关键词	保留关键词	
TRANSFORM	保留关键词	保留关键词	保留关键词	
TRIGGER	保留关键词	保留关键词	保留关键词	
UNBOUNDED	保留关键词	保留关键词	保留关键词	
UNION	保留关键词	保留关键词	保留关键词	
UNIQUE	保留关键词	保留关键词	保留关键词	
UNIQUEJOIN	保留关键词	保留关键词	保留关键词	
UNLIMITED	保留关键词	保留关键词	保留关键词	
UNTIL	—	保留关键词	保留关键词	
UPDATE	保留关键词	保留关键词	保留关键词	
USER	保留关键词	保留关键词	保留关键词	
VALUES	—	保留关键词	保留关键词	
VARCHAR	保留关键词	保留关键词	保留关键词	
VARCHAR2	保留关键词	保留关键词	保留关键词	
VARYING	保留关键词	保留关键词	保留关键词	
VIEW	保留关键词	保留关键词	保留关键词	
VIEWS	保留关键词	保留关键词	保留关键词	
WHEN	保留关键词	保留关键词	保留关键词	
WHERE	保留关键词	保留关键词	保留关键词	
WIDCARD	保留关键词	保留关键词	保留关键词	
WINDOW	保留关键词	保留关键词	保留关键词	
WITH	保留关键词	保留关键词	保留关键词	
WITHOUT	保留关键词	保留关键词	保留关键词	
YES	—	保留关键词	保留关键词	

9.2.3.2. 非保留关键词

	oracle	db2	TD	引入版本	移除版本
KUNDB	非保留关键词	非保留关键词	非保留关键词	5.0引入	
MOD	非保留关键词	非保留关键词	非保留关键词	5.0引入	
STARGATEDB	非保留关键词	非保留关键词	非保留关键词	5.0引入	
TIMELYRE	非保留关键词	非保留关键词	非保留关键词	5.0引入	
TD_UNPIVOT	—	—	非保留关键词	5.0引入	
TRIM	—	—	非保留关键词	5.0引入	
ADHOC	非保留关键词	非保留关键词	非保留关键词		
AFTER	非保留关键词	非保留关键词	非保留关键词		
ALTER	非保留关键词	非保留关键词	非保留关键词		
ANALYZE	非保留关键词	非保留关键词	非保留关键词		
ANALYZER	非保留关键词	非保留关键词	非保留关键词		
APP	非保留关键词	非保留关键词	非保留关键词		
APPS	非保留关键词	非保留关键词	非保留关键词		
ARCHIVE	非保留关键词	非保留关键词	非保留关键词		
ARRAY	非保留关键词	非保留关键词	非保留关键词		
AT	非保留关键词	非保留关键词	非保留关键词		
ATTACH	非保留关键词	非保留关键词	非保留关键词		
BACKUP	非保留关键词	非保留关键词	非保留关键词		
BODY	非保留关键词	—	—		
BOOLEAN	非保留关键词	非保留关键词	非保留关键词		
~~BOTH~~	~~非保留关键词~~	~~非保留关键词~~	~~非保留关键词~~		5.0移除
BUCKET	非保留关键词	非保留关键词	非保留关键词		
BUCKETS	非保留关键词	非保留关键词	非保留关键词		
CACHE	非保留关键词	非保留关键词	非保留关键词		
CACHEDMETRIC	非保留关键词	非保留关键词	非保留关键词		
CACHEDMETRICS	非保留关键词	非保留关键词	非保留关键词		
CAPACITY	非保留关键词	非保留关键词	非保留关键词		
CCSID	—	非保留关键词	非保留关键词		
CHANGE	非保留关键词	非保留关键词	非保留关键词		
CHECK	非保留关键词	非保留关键词	非保留关键词		
CLIENT	—	非保留关键词	非保留关键词		
COMPACT	非保留关键词	非保留关键词	非保留关键词		
COMPLETE	非保留关键词	非保留关键词	非保留关键词		
COMPRESS	—	—	非保留关键词		

	oracle	db2	TD	引入版本	移除版本
COMPUTE	非保留关键词	非保留关键词	非保留关键词		
COST	非保留关键词	非保留关键词	非保留关键词		
CS	—	—	非保留关键词		
CSVFILE	非保留关键词	非保留关键词	非保留关键词		
CURRVAL	非保留关键词	非保留关键词	非保留关键词		
CYCLE	非保留关键词	非保留关键词	非保留关键词		
DATA	非保留关键词	非保留关键词	非保留关键词		
DATE	非保留关键词	非保留关键词	非保留关键词		
DATETIME	非保留关键词	非保留关键词	非保留关键词		
DAY	非保留关键词	非保留关键词	非保留关键词		
DAYS	非保留关键词	非保留关键词	非保留关键词		
DBPROPERTIES	非保留关键词	非保留关键词	非保留关键词		
DEFAULT	非保留关键词	非保留关键词	非保留关键词		
DEFERRED	非保留关键词	非保留关键词	非保留关键词		
DELAY	非保留关键词	非保留关键词	非保留关键词		
DELETE	非保留关键词	非保留关键词	非保留关键词		
DELIMITED	非保留关键词	非保留关键词	非保留关键词		
DESC	非保留关键词	非保留关键词	非保留关键词		
DESCRIBE	非保留关键词	非保留关键词	非保留关键词		
DETTACH	非保留关键词	非保留关键词	非保留关键词		
DOUBLE	非保留关键词	—	—		
DOWNGRADE	非保留关键词	—	—		
EMIT	非保留关键词	非保留关键词	非保留关键词		
ENABLE	非保留关键词	非保留关键词	非保留关键词		
ES	非保留关键词	非保留关键词	非保留关键词		
EXCLUSIVE	非保留关键词	非保留关键词	非保留关键词		
EXIT	—	非保留关键词	非保留关键词		
EXPIRE	非保留关键词	非保留关键词	非保留关键词		
EXTERNAL	非保留关键词	非保留关键词	非保留关键词		
FACL	非保留关键词	非保留关键词	非保留关键词		
FACT	非保留关键词	非保留关键词	非保留关键词		
FETCH	非保留关键词	非保留关键词	非保留关键词		
FIELDS	非保留关键词	非保留关键词	非保留关键词		
FILE	非保留关键词	非保留关键词	非保留关键词		

	oracle	db2	TD	引入版本	移除版本
FIRST	非保留关键词	非保留关键词	非保留关键词		
FLOAT	非保留关键词	非保留关键词	非保留关键词		
FOUND	非保留关键词	—	—		
FULL	非保留关键词	非保留关键词	非保留关键词		
FULLTEXT	非保留关键词	非保留关键词	非保留关键词		
FWCFILE	非保留关键词	非保留关键词	非保留关键词		
GEO	非保留关键词	非保留关键词	非保留关键词		
GLOBAL	非保留关键词	非保留关键词	非保留关键词		
GRAPHIC	非保留关键词	非保留关键词	非保留关键词		
HELP	—	—	非保留关键词		
HOLD	—	非保留关键词	非保留关键词		
HOLD_DDLTIME	非保留关键词	非保留关键词	非保留关键词		
HOLODESK	非保留关键词	非保留关键词	非保留关键词		
HOUR	非保留关键词	非保留关键词	非保留关键词		
HOURS	非保留关键词	非保留关键词	非保留关键词		
HYPERDRIVE	非保留关键词	非保留关键词	非保留关键词		
IDXPROPERTIES	非保留关键词	非保留关键词	非保留关键词		
ITEMS	非保留关键词	非保留关键词	非保留关键词		
JAR	非保留关键词	非保留关键词	非保留关键词		
KEY	非保留关键词	非保留关键词	非保留关键词		
KEYS	非保留关键词	非保留关键词	非保留关键词		
LANGUAGE	—	非保留关键词	—		
LATERAL	非保留关键词	非保留关键词	非保留关键词		
LEFT	非保留关键词	非保留关键词	非保留关键词		
LENGTH	非保留关键词	非保留关键词	非保留关键词		
LEVEL	非保留关键词	非保留关键词	非保留关键词		
LIMIT	非保留关键词	非保留关键词	非保留关键词		
LOAD	非保留关键词	非保留关键词	非保留关键词		
LOCAL	非保留关键词	非保留关键词	非保留关键词		
LOG	非保留关键词	非保留关键词	非保留关键词		
LOGICAL	非保留关键词	非保留关键词	非保留关键词		
LOOKUP	非保留关键词	非保留关键词	非保留关键词		
MATERIALIZED	非保留关键词	非保留关键词	非保留关键词		
MEET	非保留关键词	非保留关键词	非保留关键词		

	oracle	db2	TD	引入版本	移除版本
MERGE	非保留关键词	非保留关键词	非保留关键词		
METRIC	非保留关键词	非保留关键词	非保留关键词		
METRICPROPERTIES	非保留关键词	非保留关键词	非保留关键词		
METRICS	非保留关键词	非保留关键词	非保留关键词		
METRICSOURCE	非保留关键词	非保留关键词	非保留关键词		
METRICSOURCES	非保留关键词	非保留关键词	非保留关键词		
MODE	非保留关键词	非保留关键词	非保留关键词		
MSCK	非保留关键词	非保留关键词	非保留关键词		
MUST	非保留关键词	非保留关键词	非保留关键词		
NAMED	—	—	非保留关键词		
NEW	—	非保留关键词	非保留关键词		
NEXT	非保留关键词	非保留关键词	非保留关键词		
NEXTVAL	非保留关键词	非保留关键词	非保留关键词		
NF	非保留关键词	非保留关键词	非保留关键词		
NO	非保留关键词	非保留关键词	非保留关键词		
NOCACHE	非保留关键词	非保留关键词	非保留关键词		
NOCOPY	非保留关键词	—	—		
NOCYCLE	非保留关键词	非保留关键词	非保留关键词		
NOSCAN	非保留关键词	非保留关键词	非保留关键词		
NOVALIDATE	非保留关键词	非保留关键词	非保留关键词		
NUM	非保留关键词	非保留关键词	非保留关键词		
NUMBER	—	非保留关键词	非保留关键词		
OLD	—	非保留关键词	非保留关键词		
OUT	非保留关键词	非保留关键词	非保留关键词		
OUTER	非保留关键词	非保留关键词	非保留关键词		
OUTPUTDRIVER	非保留关键词	非保留关键词	非保留关键词		
OWNER	非保留关键词	非保留关键词	非保留关键词		
PACKAGE	非保留关键词	—	—		
PACKAGES	非保留关键词	—	—		
PARAMETER	—	非保留关键词	非保留关键词		
PARQUET	非保留关键词	非保留关键词	非保留关键词		
PATTERN	非保留关键词	非保留关键词	非保留关键词		
PERCENT	非保留关键词	非保留关键词	非保留关键词		

	oracle	db2	TD	引入版本	移除版本
PERMANENT	非保留关键词	非保留关键词	非保留关键词		
PLANT	非保留关键词	非保留关键词	非保留关键词		
PLUS	非保留关键词	非保留关键词	非保留关键词		
POLICIES	非保留关键词	非保留关键词	非保留关键词		
POLICY	非保留关键词	非保留关键词	非保留关键词		
POLICYKEY	非保留关键词	非保留关键词	非保留关键词		
POLICYPROPERTIES	非保留关键词	非保留关键词	非保留关键词		
POLICYRULE	非保留关键词	非保留关键词	非保留关键词		
PRECOMPILE	非保留关键词	非保留关键词	非保留关键词		
PRETTY	非保留关键词	非保留关键词	非保留关键词		
PRIMARY	非保留关键词	非保留关键词	非保留关键词		
PROCEDURE	非保留关键词	非保留关键词	非保留关键词		
PROMPT	非保留关键词	非保留关键词	非保留关键词		
PUBLIC	非保留关键词	非保留关键词	非保留关键词		
PURGE	非保留关键词	非保留关键词	非保留关键词		
QUIT	—	—	非保留关键词		
QUOTA	非保留关键词	非保留关键词	非保留关键词		
RANGE	非保留关键词	非保留关键词	非保留关键词		
RCFILE	非保留关键词	非保留关键词	非保留关键词		
READ	非保留关键词	非保留关键词	非保留关键词		
READS	非保留关键词	非保留关键词	非保留关键词		
REAL	非保留关键词	非保留关键词	非保留关键词		
REBUILD	非保留关键词	非保留关键词	非保留关键词		
REF	非保留关键词	—	—		
REFERENCES	非保留关键词	非保留关键词	非保留关键词		
REGEXP	非保留关键词	非保留关键词	非保留关键词		
RELOAD	非保留关键词	非保留关键词	非保留关键词		
RELY	非保留关键词	非保留关键词	非保留关键词		
REMOVE	非保留关键词	非保留关键词	非保留关键词		
REPAIR	非保留关键词	非保留关键词	非保留关键词		
REPEAT	—	非保留关键词	非保留关键词		
REPLACE	非保留关键词	非保留关键词	非保留关键词		
RESET	非保留关键词	—	非保留关键词		

	oracle	db2	TD	引入版本	移除版本
RESOURCE	非保留关键词	非保留关键词	非保留关键词		
RESPECT	非保留关键词	非保留关键词	非保留关键词		
RESTRICT	非保留关键词	非保留关键词	非保留关键词		
RESULT	—	非保留关键词	非保留关键词		
REVERSE	非保留关键词	—	—		
RIGHT	非保留关键词	非保留关键词	非保留关键词		
ROLE	非保留关键词	非保留关键词	非保留关键词		
ROLES	非保留关键词	非保留关键词	非保留关键词		
ROUTED	非保留关键词	非保留关键词	非保留关键词		
RULE	非保留关键词	非保留关键词	非保留关键词		
RULEPROPERTIES	非保留关键词	非保留关键词	非保留关键词		
RULES	非保留关键词	非保留关键词	非保留关键词		
RULESET	非保留关键词	—	非保留关键词		
RULESETS	非保留关键词	—	非保留关键词		
SAVE	非保留关键词	—	—		
SCHEDULER	非保留关键词	非保留关键词	非保留关键词		
SCHEMA	非保留关键词	非保留关键词	非保留关键词		
SEARCH	非保留关键词	非保留关键词	非保留关键词		
SEGMENT	非保留关键词	非保留关键词	非保留关键词		
SEMI	非保留关键词	非保留关键词	非保留关键词		
SEPARATED	非保留关键词	非保留关键词	非保留关键词		
SEQUENCE	非保留关键词	非保留关键词	非保留关键词		
SEQUENCEFILE	非保留关键词	非保留关键词	非保留关键词		
SEQUENCES	非保留关键词	非保留关键词	非保留关键词		
SERDE	非保留关键词	非保留关键词	非保留关键词		
SERDEPROPERTIES	非保留关键词	非保留关键词	非保留关键词		
SERVER	非保留关键词	非保留关键词	非保留关键词		
SET	非保留关键词	—	非保留关键词		
SETPROPERTIES	非保留关键词	—	非保留关键词		
SETS	非保留关键词	—	非保留关键词		
SHARD	非保留关键词	非保留关键词	非保留关键词		
SHARED	非保留关键词	非保留关键词	非保留关键词		
SHOW	非保留关键词	非保留关键词	非保留关键词		

	oracle	db2	TD	引入版本	移除版本
SIGNAL	—	非保留关键词	非保留关键词		
SKEWED	非保留关键词	非保留关键词	非保留关键词		
SLIDE	非保留关键词	非保留关键词	非保留关键词		
SNAPSHOT	非保留关键词	非保留关键词	非保留关键词		
SPACE	非保留关键词	非保留关键词	非保留关键词		
SPOOL	非保留关键词	非保留关键词	非保留关键词		
SQL	—	非保留关键词	非保留关键词		
SQLSTATE	—	非保留关键词	非保留关键词		
SSL	非保留关键词	非保留关键词	非保留关键词		
STARGATE	非保留关键词	非保留关键词	非保留关键词		
STATEMENT	—	非保留关键词	非保留关键词		
STELLARDB	非保留关键词	非保留关键词	非保留关键词		
STREAM	非保留关键词	非保留关键词	非保留关键词		
STREAMJOB	非保留关键词	非保留关键词	非保留关键词		
STREAMJOBS	非保留关键词	非保留关键词	非保留关键词		
STRUCT	非保留关键词	非保留关键词	非保留关键词		
SYNC	非保留关键词	非保留关键词	非保留关键词		
SYSDATE	非保留关键词	—	—		
TBLPROPERTIES	非保留关键词	非保留关键词	非保留关键词		
TEXTFILE	非保留关键词	非保留关键词	非保留关键词		
TIES	—	—	非保留关键词		
TIME	非保留关键词	非保留关键词	非保留关键词		
TIMES	非保留关键词	非保留关键词	非保留关键词		
TIMESTAMP	非保留关键词	非保留关键词	非保留关键词		
TITLE	—	—	非保留关键词		
TRUNCATE	非保留关键词	非保留关键词	非保留关键词		
TTL	非保留关键词	非保留关键词	非保留关键词		
TYPE	非保留关键词	非保留关键词	非保留关键词		
UC	—	—	非保留关键词		
UNARCHIVE	非保留关键词	非保留关键词	非保留关键词		
UNDO	非保留关键词	—	—		
UNICODE	—	非保留关键词	非保留关键词		
UNSET	非保留关键词	—	非保留关键词		
UNSIGNED	非保留关键词	非保留关键词	非保留关键词		

	oracle	db2	TD	引入版本	移除版本
UNUSABLE	非保留关键词	非保留关键词	非保留关键词		
UPGRADE	非保留关键词	非保留关键词	非保留关键词		
URI	非保留关键词	非保留关键词	非保留关键词		
USE	非保留关键词	非保留关键词	非保留关键词		
USE_DELETE_MARK	非保留关键词	非保留关键词	非保留关键词		
UTC	非保留关键词	非保留关键词	非保留关键词		
UTC_TMESSTAMP	非保留关键词	非保留关键词	非保留关键词		
VALIDATE	非保留关键词	非保留关键词	非保留关键词		
VALUE	—	非保留关键词	非保留关键词		
VALUES	非保留关键词	—	—		
VARGRAPHIC	非保留关键词	非保留关键词	非保留关键词		
VARRAY	非保留关键词	—	—		
VOLATILE	—	—	非保留关键词		
WAIT	非保留关键词	非保留关键词	非保留关键词		
WHILE	非保留关键词	非保留关键词	非保留关键词		
WRITE	非保留关键词	非保留关键词	非保留关键词		
YEAR	非保留关键词	非保留关键词	非保留关键词		
YEARS	非保留关键词	非保留关键词	非保留关键词		
FALSE	非保留关键词	非保留关键词	非保留关键词		
TRUE	非保留关键词	非保留关键词	非保留关键词		

9.3. 表可存储格式

Inceptor 提供多模型支持能力，为多种结构化/半结构化等数据提供统一的管理解决方案，可以更好地处理多样化的数据类型，从而提高数据存储和查询的效率和灵活性，目前支持关系型、搜索、文本和对象数据模型。

您可以在建表时，基于业务特点，围绕数据规模、查询性能、存储需求等角度选择表的存储格式，本小节将介绍不同表格式的适用场景和相关语法。

- TEXTFILE
- ORCFILE/ORCTRANSACTIONFILE
- CSVFILE
- EXCEL/JSON
- FWCFILE
- SEQUENCEFILE
- RCFILE

- HOLODESK
- PARQUET
- HYPERDRIVE
- ESDRIVE
- SEARCH

9.3.1. TEXTFILE

TEXT 表是文本格式的表，是默认的表格式。在数据量大的情况下，TEXT 表的统计和查询性能都比较低；TEXT 表也不支持事务处理，所以通常用于将文本文件中的原始数据导入 Inceptor 中。

提供两种方式将文本文件中的数据导入 TEXT 表中：

- 建外部 TEXT 表，让该表指向 HDFS 上的一个目录，会将目录下文件中的数据都导入该表。星环科技推荐使用这个方式导入数据。
- 建 TEXT 表（外表内表皆可）后将本地或者 HDFS 上的一个文件或者一个目录下的数据 LOAD 进该表。这种方式在安全模式下需要多重认证设置，极易出错，星环科技不推荐使用这个方式导入数据。

创建TEXT表

语法：

```
CREATE [TEMPORARY] --[1]
[EXTERNAL]  TABLE  <table_name> --[2]
(<column_name> <data_type>, <column_name> <data_type>, ...)
[PARTITIONED  BY  ...]--[3]
[CLUSTERED  BY  ...]--[4]
[ROW  FORMAT ...] --[5]
[STORED  AS  TEXTFILE] --[6]
[LOCATION  '<hdfs_path>'] --[7]
[TBLPROPERTIES ('<property_name>'='<property_value>', ...)]; --[8]
```

参数说明：

- <table_name>：表名称。
- <column_name>：列名称。
- <data_type>：数据类型。
- <property_name>：表属性名称。
- <property_value>：表属性值。
- <hdfs_path>：表的存储路径，可以是相对路径也可以是绝对路径。
- 可选项[1]，添加 TEMPORARY 关键字创建临时表。
- 可选项[2]，添加 EXTERNAL 关键字创建外表。
- 可选项[3]，使用 PARTITIONED BY 创建分区表，具体请参考创建分区表。
- 可选项[4]，使用 CLUSTERED BY 创建分桶表，具体请参考创建分桶表。
- 可选项[5]，支持设定表的行格式。
- 可选项[6]，支持使用不同的表存储格式。
- 可选项[7]，这里用于手动指定表在 HDFS 中的路径。
- 可选项[8]，支持使用 TBLPROPERTIES 设定表的属性。

9.3.2. ORCFILE/ORCTRANSACTIONFILE

ORC 非事务表的建表只需在建表语句中用 **STORED AS ORC** 指定存储格式为 ORC 即可。

ORC 事务表的建表则需要几个额外的重点步骤：

- 为表分桶：为了保证增删改过程中的性能，我们要求 ORC 事务表必须是部分排序或者全局排序的，但是全局排序又过于耗费计算资源，因此我们要求 ORC 表必须是分桶表。
- 如果表的数据量特别大，建议在分桶的基础上再分区，ORC 事务表支持单值分区和范围分区。

创建 ORC 事务表

- 非分区分桶表

```
CREATE TABLE <table_name> (<column_name> <data_type>, <column_name> <data_type>, ...)
CLUSTERED BY (<bucket_key>) INTO <num_buckets> BUCKETS
STORED AS ORC_TRANSACTION;
```

- 单值分区分桶表

```
CREATE TABLE <table_name> (<column_name> <data_type>, <column_name> <data_type>, ...)
PARTITIONED BY (<partition_key> <data_type>)
CLUSTERED BY (<bucket_key>) INTO <num_buckets> BUCKETS
STORED AS ORC_TRANSACTION;
```

- 范围分区分桶表

```
CREATE TABLE <table_name> (<column_name> <data_type>, <column_name> <data_type>, ...)
PARTITIONED BY RANGE(<partition_key1> <data_type>, <partition_key2> <data_type>, ...) (
PARTITION [<partition_name_1>] VALUE LESS THAN(<key1_bound_value1>, <key2_bound_value1>, ...),
--[1]
PARTITION [partition_name_2] VALUE LESS THAN(key1_bound_value2, key2_bound_value2, ...),
...
)
CLUSTERED BY (<bucket_key>) INTO <num_buckets> BUCKETS
STORED AS ORC_TRANSACTION;
```

参数说明

- <table_name>：表名称。
- <column_name>：列名称。
- <data_type>：数据类型。
- <num_buckets>：分桶数。
- <bucket_key>：分桶列。
- <partition_key>：分区键。
- <partition_name>：范围分区中的指定某个分区名称。
- <key1_bound_value>：范围分区中一个分区的上限值。
- MAXVALUE：当最后一个分区的值是 MAXVALUE 时，它将不能再增加分区。
- 可选项[1]，选择是否对范围分区设置名称。

9.3.3. CSVFILE

CSV 表的数据来源是 CSV 文件。CSV 文件是纯文本文件，文件中包含数据以及分隔符。和 TEXT 表相似，CSV 表最常见的使用场景是用于建外表，将 CSV 文件中的数据导入 Inceptor，星环科技不建议在任何计算场景中使用 CSV 表。计算时，应该总是将 CSV 表中的数据用 **INSERT … SELECT** 语句插入 ORC 或者 Holodesk 表。

创建 CSV 外表

语法：

```
CREATE EXTERNAL TABLE <table_name>
(<column_name> <data_type>, <column_name> <data_type>, ...)
STORED AS CSVFILE
[LOCATION '<hdfs_path>']-- [1]
[TBLPROPERTIES ( --[2]
['field.delim'='<field_delimiter>',] --[3]
['line.delim'='<newline_char>',] --[4]
['serialization.null.format'='<null_pre>',] --[5]
['quote.delim'='<quote_delimiter>',] -- [6]
['<property_name>'='<property_value>',]...)]; --[7]
```

参数说明

- <table_name>：表名称。
- <column_name>：列名称。
- <data_type>：数据类型。
- <hdfs_path>：表的存储路径，可以是相对路径也可以是绝对路径。
- <field_delimiter>：字段分隔符，默认值为 “,” 。由 <field_delimiter> 分隔的字段会被解析为不同列中的字段。
- <newline_char>：行分隔符，默认为 “\n” 。
- <null_pre>：NULL 值字符，默认为空字段。只支持单个字符，即一个 CHAR。
- <quote_delimiter>：指定用什么字符作为单个字段的分隔符，默认值为“”” 。
- <property_name>：表属性名称。
- <property_value>：表属性值。
- 可选项[1]，这里用于手动指定表在 HDFS 中的路径。
- 可选项[2]，支持使用 TBLPROPERTIES 设定表的属性。
- 可选项[3]，支持使用 field.delim 指定字段（列）分隔符。
- 可选项[4]，支持使用 line.delim 指定行分隔符。
- 可选项[5]，支持使用 serialization.null.format 定义 NULL 值字段的存储字符，默认为空字段。
- 可选项[6]，支持使用 quote.delim 指定每列内部单个字段的分隔符。
- 可选项[7]，持上述 可选项 [3] - [6] 之外，还支持设定表的其他属性信息。

例子

csv文件如下： field1,"field2-part1" "field2-part2, field3-part3", field3 2field1,"2field2-part1 2field2-part2, 2field3-part3", 2field3

建表语句如下：

```
CREATE EXTERNAL TABLE csv_table
(
col1 STRING,
col2 STRING,
col3 STRING
)
STORED AS CSVFILE
LOCATION '/user/alice/csv1'
TBLPROPERTIES(
'field.delim'=',',
'quote.delim'='"',
'line.delim'='\n');
```

9.3.4. EXCEL/JSON

EXCEL 文件就是 excel 软件生成的文件，也叫工作簿，其后缀名一般都是 XLS（2003 版以后的是 XLSX）。JSON 是 JS 对象的字符串表示法，它使用文本表示一个 JS 对象的信息，本质是一个字符串。

创建 EXCEL/JSON 表

```
CREATE EXTERNAL TABLE <table_name>
(<column_name> <data_type>, <column_name> <data_type>, ...)
STORED AS EXCEL/JSON;
```

参数说明

- <table_name>：表名称。
- <column_name>：列名称。
- <data_type>：数据类型。

注意

- 支持读取多 sheet 的数据，但是写入到文件中，只会写入到一个 sheet 中，sheet 名字叫做“sheet1”。
- 支持读 xls/xlsx 或者两者混合，但是推荐使用后者。数据源文件，如果没有后缀则当成xlsx。写文件只支持写xlsx。
- 支持所有 primitive 类型
- 不支持 Excel 存储的文件 split。
- 不支持 json/excel 存储下的复杂类型。
- json 用的是默认的 inputformat，意味着 json 存储的数据文件，依然是按照默认换行符来换行，样例数据格式：

  {"name":"inceptor","id":"1"}

  {"name":"transwarp","id":"2"}

9.3.5. FWCFILE

FWCFILE 是定宽文本文件，有一些业务场景中，客户提供的数据文件为定宽文本文件，即每个字段的字节宽度是固定的，字段和字段之间没有显示的分隔符，每条数据之间有行分隔符。

创建FWC外表

语法：

```
CREATE EXTERNAL TABLE <table_name>(<column_name> <data_type>, <column_name> <data_type>, ...)
[STORED AS FWCFILE] -- [1]
[LOCATION "<hdfs_path>"] --[2]
TBLPROPERTIES(
['fields.width'='<fields_width>',] -- [3]
['padding.type'='LEFT|RIGHT', 'padding.char'='<pad_char>',] -- [4]
['serialization.encoding'='<code_type>',]--[5]
['serialization.null.format'='<null_pre>')] --[6]
['<property_name>'='<property_value>',]...)]; --[7]
```

参数说明

- <table_name>：表名称。
- <column_name>：列名称。
- <data_type>：数据类型。
- <hdfs_path>：表的存储路径，可以是相对路径也可以是绝对路径。
- <fields_wildth>：每个字段支持存入的字符数量。
- <pad_char>：padding 填充字符。
- <code_type>：编码格式，如 UTF-8。
- <null_pre>：NULL 值字符，默认为空字段。只支持单个字符，即一个 CHAR。
- <property_name>：表属性名称。
- <property_value>：表属性值。
- 可选项[1]，设定表存储格式，此处为 FWCFILE。
- 可选项[2]，这里用于手动指定表在 HDFS 中的路径。
- 可选项[3]，支持使用 fields_width 指定字段（列）可存入的最大字符数量。
- 可选项[4]，定义 padding 类型为左侧（LEFT）或右侧（RIGHT）插入填充字符，以及定义填充字符。
- 可选项[5]，支持定义文件的编码格式。
- 可选项[6]，支持使用 serialization.null.format 指定 NULL 值字段的存储字符。
- 可选项[7]，持上述 可选项 [3] - [6] 之外，还支持设定表的其他属性信息。

实例

```
DROP DATABASE IF EXISTS FWC;
CREATE DATABASE FWC;
USE FWC;
DROP TABLE IF EXISTS TEST;
CREATE EXTERNAL TABLE TEST(col1 STRING, col2 STRING, col3 STRING, col4 STRING, col5 STRING,
col6 STRING)
STORED AS FWCFILE ①
LOCATION "/home/yibin/temp/fwc/data"
TBLPROPERTIES('fields.width'='20,100,2,40,14,3', ②
'padding.type'='right','padding.char'=' ', ③
'serialization.encoding'='gb18030', 'serialization.null.format'='\N'); ④
```

① 文件类型为 **FWCFILE**

② 字段的字节数分别为：20,100,2,40,14,3

③ padding 类型为右侧插入填充字符，填充字符为空格

④ NULL 的表示方式为 **\N**

9.3.6. SEQUENCEFILE

SequeceFile是Hadoop API提供的一种二进制文件支持。这种二进制文件直接将<key, value>对序列化到文件中。一般对小文件可以使用这种文件合并，即将文件名作为key，文件内容作为value序列化到大文件中。

这种文件格式有以下好处：

1. 支持压缩，且可指定为基于Record或Block压缩（Block级压缩性能较优）

2. 本地化任务支持：因为文件可以被切分，因此MapReduce任务时数据的本地化情况是非常好的。

3. 难度低：因为是Hadoop框架提供的API，业务逻辑侧的修改比较简单。

坏处： 需要一个合并文件的过程，且合并后的文件将不方便查看。

创建SEQUENCEFILE表

语法：

```
CREATE [EXTERNAL] TABLE --[1]
[IF NOT EXISTS] <table_name> --[2]
[(<column_name> <data_type>  --[3]
[COMMENT '<column_comment>'], ...)] --[4]
[COMMENT '<table_comment>'] --[5]
[PARTITIONED BY (<partition_key> <data_type>  --[6]
[COMMENT '<column_comment>'], ...)] --[7]
[CLUSTERED BY (<bucket_column1>, <bucket_column2>, ...) --[8]
[SORTED BY (<column_name> [ASC|DESC], ...)]--[9]
INTO <num_buckets> BUCKETS]
[ROW FORMAT <row_format>] --[10]
[STORED AS SEQUENCEFILE] --[11]
[LOCATION <hdfs_path>] --[12]
```

参数说明

- <table_name>：表名称。
- <column_name>：列名称。
- <data_type>：数据类型。
- <column_comment>：列注释。

- <table_comment>：表注释。
- <partition_key>：分区键。
- <num_buckets>：分桶数。
- <bucket_column>：分桶列。
- <row_format>：表的行格式。
- <hdfs_path>：表的存储路径，可以是相对路径也可以是绝对路径。
- 可选项[1]，添加 EXTERNAL 关键字创建外表。
- 可选项[2]，如果 <table_name> 指定的表已经存在，会报错，加上 IF NOT EXISTS 选项在建表前检测是否已存在同名表。
- 可选项[3]，支持对表定义列。
- 可选项[4]，用 [COMMENT] 为列加注释，注意注释要放在引号中。
- 可选项[5]，用 [COMMENT] 为表加注释，注意注释要放在引号中。
- 可选项[6]，使用 PARTITIONED BY 创建分区表，具体请参考创建分区表。
- 可选项[7]，用 [COMMENT] 为分区加注释，注意注释要放在引号中。
- 可选项[8]，使用 CLUSTERED BY 创建分桶表，具体请参考创建分桶表。
- 可选项[9]，支持使用 SORTED BY 对分桶中的数据排序，DESC 表示降序，ASC 表示升序（默认）。
- 可选项[10]，支持设定表的行格式。
- 可选项[11]，支持使用不同的表存储格式。这里为 SEQUENCEFILE。
- 可选项[12]，这里用于手动指定表在 HDFS 中的路径。

_

9.3.7. RCFILE

结合列存储和行存储的优缺点，Facebook提出了基于行列混合存储的RCFile，它是基于SEQUENCEFILE实现的列存储格式，它即满足快速数据加载和动态负载高适应的需求外，也解决了SEQUENCEFILE的一些瓶颈。该存储结构遵循的是“先水平划分，再垂直划分”的设计理念。先将数据按行水平划分为行组，这样一行的数据就可以保证存储在同一个集群节点；然后在对行进行垂直划分。

创建RCFILE表

语法：

```
CREATE [EXTERNAL] TABLE --[1]
[IF NOT EXISTS] <table_name> --[2]
[(<column_name> <data_type> --[3]
[COMMENT '<column_comment>'], ...)] --[4]
[COMMENT '<table_comment>'] --[5]
[PARTITIONED BY (<partition_key> <data_type> --[6]
[COMMENT '<column_comment>'], ...)] --[7]
[CLUSTERED BY (<bucket_column>, <bucket_column>, ...) --[8]
[SORTED BY (<column_name> [ASC|DESC], ...)] --[9]
INTO <num_buckets> BUCKETS]
[ROW FORMAT <row_format>] --[10]
[STORED AS RCFILE] --[11]
[LOCATION <hdfs_path>] --[12]
```

参数说明

- <table_name>：表名称。

- <column_name>：列名称。
- <data_type>：数据类型。
- <column_comment>：列注释。
- <table_comment>：表注释。
- <partition_key>：分区键。
- <num_buckets>：分桶数。
- <bucket_column>：分桶列。
- <row_format>：表的行格式。
- <hdfs_path>：表的存储路径，可以是相对路径也可以是绝对路径。
- 可选项[1]，添加 EXTERNAL 关键字创建外表。
- 可选项[2]，如果 <table_name> 指定的表已经存在，会报错，加上 IF NOT EXISTS 选项在建表前检测是否已存在同名表。
- 可选项[3]，支持对表定义列。
- 可选项[4]，用 [COMMENT] 为列加注释，注意注释要放在引号中。
- 可选项[5]，用 [COMMENT] 为表加注释，注意注释要放在引号中。
- 可选项[6]，使用 PARTITIONED BY 创建分区表，具体请参考创建分区表。
- 可选项[7]，用 [COMMENT] 为分区加注释，注意注释要放在引号中。
- 可选项[8]，使用 CLUSTERED BY 创建分桶表，具体请参考创建分桶表。
- 可选项[9]，支持使用 SORTED BY 对分桶中的数据排序，DESC 表示降序，ASC 表示升序（默认）。
- 可选项[10]，支持设定表的行格式。
- 可选项[11]，支持使用不同的表存储格式。这里为 RCFILE。
- 可选项[12]，这里用于手动指定表在 HDFS 中的路径。

9.3.8. HOLODESK

Holodesk对满足以下特征的场景表现出了极强的处理能力，极力建议对这些场景创建Holodesk表：

1. 当机器拥有很大的内存或者部署了SSD时。
2. 过滤高的场景，包括单表扫描和多表MapJoin等。
3. 聚合率高的场景，例如GROUP BY之后，信息被大量聚合。

9.3.9. PARQUET

Parquet 是列式存储的一种文件类型，无论数据处理框架，数据模型或编程语言的选择如何，Parquet都是Hadoop生态系统中任何项目可用的列式存储格式。Parquet与TEXT、JSON、CSV等文件格式相比，它有三个核心特征：列式存储、自带Schema、具备Predicate Filter特性。

创建PARQUET表

语法：

```
CREATE [EXTERNAL] TABLE --[1]
[IF NOT EXISTS] <table_name> --[2]
[(<column_name> <data_type> --[3]
[COMMENT '<column_comment>'], ...)] --[4]
[COMMENT '<table_comment>'] --[5]
[PARTITIONED BY (<partition_key> <data_type> --[6]
[COMMENT '<column_comment>'], ...)] --[7]
[CLUSTERED BY (<bucket_column>, <bucket_column>, ...)]--[8]
[STORED AS PARQUET ];--[9]
```

参数说明

- <table_name>：表名称。
- <column_name>：列名称。
- <data_type>：数据类型。
- <column_comment>：列注释。
- <table_comment>：表注释。
- <partition_key>：分区键。
- <bucket_column>：分桶列。
- 可选项[1]，添加 EXTERNAL 关键字创建外表。
- 可选项[2]，如果 <table_name> 指定的表已经存在，会报错，加上 IF NOT EXISTS 选项在建表前检测是否已存在同名表。
- 可选项[3]，支持对表定义列。
- 可选项[4]，用 [COMMENT] 为列加注释，注意注释要放在引号中。
- 可选项[5]，用 [COMMENT] 为表加注释，注意注释要放在引号中。
- 可选项[6]，使用 PARTITIONED BY 创建分区表，具体请参考创建分区表。
- 可选项[7]，用 [COMMENT] 为分区加注释，注意注释要放在引号中。
- 可选项[8]，使用 CLUSTERED BY 创建分桶表，具体请参考创建分桶表。
- 可选项[9]，支持使用不同的表存储格式。这里为 PARQUET。

9.3.10. HYPERDRIVE

若需使用此存储格式需要提前安装星环的 Hyperbase 产品。

9.3.10.1. Transwarp Hyperbase的典型使用场景

- 对一张表中作为唯一标识的主键进行精确查询或后模糊查询 —— Hyperbase Rowkey 查询
- 对一张表中部分字段做精确查询或后模糊查询—— Hyperbase 全局索引查询
- 对一张表中部分字段进行全模糊查询或前模糊查询—— Hyperbase 全文索引查询
- 需要存储音频，视屏，图片等非格式化大段数据—— Hyperbase LOB 索引

9.3.10.2. 创建HYPERDRIVE表

建映射表语法

```
CREATE [EXTERNAL] TABLE <table_name> ( --[1]
<key> <key_data_type>,
<column_name1> <data_type>,
<column_name2> <data_type>,
...)
STORED BY
'org.apache.hadoop.hive.hbase.HBaseStorageHandler|io.transwarp.hyperdrive.HyperdriveStorageHandler'
--[2]
WITH SERDEPROPERTIES('hbase.columns.mapping':key,<column_name1>,<column_name2>,...') --[3]
[TBLPROPERTIES('hbase.table.name'='<table_in_hbase>')]; --[4]
```

参数说明

- <table_name>：表名称。
- <key>：key 名称。
- <column_name>：列名称。
- <data_type>：数据类型。
- <table_in_hbase>：映射表在 hbase 中的表名称。
- 可选项[1]，**[EXTERNAL]** 为外表选项，加上 **EXTERNAL** 即建外表。Inceptor 对外表没有所有权，删除表时，仅仅删除表中的元数据，不会删除原表数据。对于非外表，Inceptor 有所有权，所以在删除时会将元数据和数据都删除。
- [2]，指定表使用的Storage Handler：建HBase表，使用 **'org.apache.hadoop.hive.hbase.HBaseStorageHandler'** ；建 Hyperdrive 表使用 **'io.transwarp.hyperdrive.HyperdriveStorageHandler'** 。注意选择的Storage Handler要放在引号中。
- [3]，**hbase.columns.mapping** 属性定义映射表和HBase表之间的列对应关系。映射表的第一列必须对应 HBase 表中的 Row Key。映射表中剩余的列将是对应 HBase 表中的 **<cf>:<cq>** **(<column_family>:<column_qualifier>)** 组合。
- 可选项[4]，指定映射表对应的 Transwarp HBase 表的名称。

Hyperdrive映射表的简化建表

```
CREATE TABLE <table_name> (
<key> <key_data_type>,
<column_name1> <data_type>,
<column_name2> <data_type>,
...)
STORED AS HYPERDRIVE;
```

- 说明：

  我们鼓励用户尽量使用简化建表方式。但是简化建表方式会使用默认的方式建立映射表和Hyperdrive表之间的对应关系：映射表对应同名Hyperdrive表，映射表中的列对应列族为、column qualifier和映射表列名相同的列（例如，如果映射表中的列名为a，Hyperdrive表中的列名则为f:a）。所以，需要自定义对应关系的情况下还是需要选择普通建表方式。例如，当和映射表名的Hyperdrive表已经存在，那么就需要用 TBLPROPERTIES ('hbase.table.name'='<hbase_table>') 来指定另一个表名。

参数说明：

- <table_name>：表名称。
- <key>：key 名称。
- <column_name>：列名称。
- <data_type>：数据类型。

9.3.11. ESDRIVE

若使用此存储格式需要依赖Search

9.3.11.1. Esdrive典型使用场景

- 对表中字段需要进行分词查询
- 需要对word，excel，pdf，日志文件中进行分词信息查询
- 对表中大量字段会进行模糊查询

9.3.11.2. 创建ES表

- 创建一张Transwarp ES内表

```
CREATE TABLE es_start(
        key           STRING,
        content   STRING,
        tint         INT,
        tfloat     FLOAT,
        tbool       BOOLEAN
)
STORED AS ES;
```

- 创建一张Transwarp ES外表

```
CREATE EXTERNAL TABLE es_start_ex(
        key           STRING,
        content   STRING,
        tint         INT,
        tfloat     FLOAT,
        tbool       BOOLEAN
)
STORED AS ES
TBLPROPERTIES('elasticsearch.tablename'='default.es_start');
```

9.3.12. SEARCH

Search表与ES表使用基本相同且性能优于ES表。

创建SEARCH表

```
CREATE [EXTERNAL] TABLE --[1]
[IF NOT EXISTS] <table_name> --[2]
[(<column_name> <data_type> --[3]
[COMMENT '<column_comment>'], ...)] --[4]
[COMMENT '<table_comment>'] --[5]
[PARTITIONED BY (<partition_key> <data_type> --[6]
[COMMENT '<column_comment>'], ...)] --[7]
[CLUSTERED BY (<bucket_column>, <bucket_column>, ...) --[8]
[STORED AS SEARCH|STORED BY 'io.transwarp.searchdrive.SearchStorageHandler']; --[9]
```

参数说明

- <table_name>：表名称。
- <column_name>：列名称。
- <data_type>：数据类型。
- <column_comment>：列注释。
- <table_comment>：表注释。

- <partition_key>：分区键。
- <bucket_column>：分桶列。
- 可选项[1]，添加 EXTERNAL 关键字创建外表。
- 可选项[2]，如果 <table_name> 指定的表已经存在，会报错，加上 IF NOT EXISTS 选项在建表前检测是否已存在同名表。
- 可选项[3]，支持对表定义列。
- 可选项[4]，用 [COMMENT] 为列加注释，注意注释要放在引号中。
- 可选项[5]，用 [COMMENT] 为表加注释，注意注释要放在引号中。
- 可选项[6]，使用 PARTITIONED BY 创建分区表，具体请参考创建分区表。
- 可选项[7]，用 [COMMENT] 为分区加注释，注意注释要放在引号中。
- 可选项[8]，使用 CLUSTERED BY 创建分桶表，具体请参考创建分桶表。
- 可选项[9]，支持两种存储方式创建SEARCH表。

9.4. 数据类型

9.4.1. Inceptor 支持的数据类型

分类	数据类型	描述	示例	备注
数值型	INT	4字节（32位）有符号整数，从-2147483648到2147483647	1	
	TINYINT	1字节（8位）带符号整数，从-128到127	-1	
	SMALLINT	2字节（16位）有符号整数，从-32768到32767	-1	
	BIGINT	8字节（64位）有符号整数，从-9223372036854775808到9223372036854775807	1	
	float	4字节单精度浮点数	1	
	DOUBLE	8字节双精度浮点数	1	
	DECIMAL(m,n)	表示m位数中，有n位是小数，即m-n位整数	1.00E+44	当写入数据的小数位数大于n时，四舍五入存入；当整数位数大于(m-n)时，写入空值
	DECIMAL	默认为DECIMAL(10,2),不可变的，任意精度的，有符号的十进制数	1	
	NUMBER	包括整数和浮点数	1	
	NUMBER(m,n)	m是有效位,n表示有n个是小数	NUMBER(4,3) 0.0999	
	NUMERIC(m,n)	m是有效位,n表示有n个是小数	1	
布尔类型	BOOLEAN	true/false	true/false	

分类	数据类型	描述	示例	备注
字符串类型	STRING	字符串	'a'	字段长度最大允许 4K。
	CHAR(n)	定长字符	'a'	Inceptor 只支持带长度 n 的 CHAR(n)，表示可以存储 n 个字符。
	VARCHAR(n)	定长字符	'a'	Inceptor 只支持带长度 n 的 VARCHAR(n)，表示可以存储 n 个字符。
	VARCHAR2(n)	定长字符	'a'	Inceptor 只支持带长度 n 的 VARCHAR2(n)，表示可以存储 n 个字符。
日期/时间类型	DATE	日期。格式为 'yyyy-MM-dd' 或者 'yyyy-MM-dd HH:mm:ss'。	'2014-01-01'	
	TIMESTAMP	时间戳，表示日期和时间。格式：'yyyy-MM-dd HH:mm:ss.fffffffff'，可达到小数点后9位（纳秒级别）精度	'2014-01-01 00:00:00'	
	TIME	由三部分组成：小时、分钟和秒。小时部分的范围是从 0 到 24。分钟和秒部分的范围都是从 0 到 59。如果小时为 24，分钟和秒的值都是 0。格式为：'HH:mm:ss'。	'10:07:05'	
大对象类型	BLOB	大对象存储，字符集是Binary	可用于存储媒体、图片等二进制文件	
	CLOB	大对象存储，字符集是UTF8MB4	可用于存储 xml 等二进制文本数据文件	

9.4.2. 数据类型映射

Inceptor	Oracle	DB2	TD	PG	MySQL
int	Integer=Decimal(38,0)	int、integer	int	int	int
TINYINT					TINYINT
SMALLINT	SMALLINT=Decimal(38,0)	SMALLINT	SMALLINT	SMALLINT	SMALLINT
BIGINT		BIGINT		BIGINT	BIGINT
float	float、BINARYFLOAT	Real	float	float4	float
DOUBLE	BinaryDouble	Float、Double	DOUBLE	float8	DOUBLE
decimal(m,n)	decimal(m,n)	decimal(m,n)、DecFloat		decimal(m,n)	decimal(m,n)
decimal	decimal	decimal		decimal	decimal

Inceptor	Oracle	DB2	TD	PG	MySQL
boolean	Bit、Boolean	boolean		boolean	
string		Graphic、VarGraphic			
char(n)	NCHAR[(n)]、CHAR[(n [BYTE |CHAR])]		char(n)	char(n)	char(n)
varchar(n)			varchar(n)	varchar(n)	varchar(n)
VARCHAR2(n)	VARCHAR2(n)、NVARCHAR2(n)				
date	date	date	date	date	date
TIMESTAMP	TIMESTAMP	TIMESTAMP	TIMESTAMP		TIMESTAMP
TIME		TIME	TIME		TIME
BLOB	BLOB、RAW(n)、LONG RAW	BLOB			BIT、BINARY、VARBINARY、TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB
CLOB	CLOB、NCLOB	CLOB	CLOB		TINYTEXT、TEXT、MEDIUMMTEXT、LONGTEXT
NUMBER（Inceptor 存储时自动转换为 decimal(10,2)）	NUMBER				NUMBER
Number(m,n)	Number(m,n)				Number(m,n)
Numeric(m,n)		Numeric(m,n)	Numeric(m,n)	Numeric(m,n)	Numeric(m,n)

9.5. 系统表

- **buckets_v**

buckets_v 系统表存储了 Inceptor 中所有分桶的及分桶表的相关信息。

表 12. buckets_v 的字段

字段名称	类型	描述
bucket_column	varchar2(256,ORACLE)	分桶表中的分桶列列名
bucket_number	int	分桶表的分桶数量
table_name	varchar2(128,ORACLE)	分桶表名称
database_name	varchar2(128,ORACLE)	分桶表所在数据库名称
table_type	varchar2(128,ORACLE)	分桶表的表类型

- **columns_v**

columns_v 系统表存储了 Inceptor 中所有列信息。

表 13. columns_v 的字段

字段名称	类型	描述
column_id	int	列编号
column_name	varchar2(128,ORACLE)	列名称
column_type	string	列存储的数据类型
table_name	varchar2(128,ORACLE)	列所属表名称
database_name	varchar2(128,ORACLE)	列所属数据库名称
commentstring	varchar2(256,ORACLE)	列的注释信息
default_value	varchar2(256,ORACLE)	列默认值
nullable	boolean	列是否支持空值
unique_constraint	boolean	是否唯一约束
column_length	int	列支持存储数据的字段长度
column_scale	int	列的规模

- **databases_v**

databases_v 系统表存储了 Inceptor 中的所有数据库信息。

表 14. databases_v 的字段

字段名称	类型	描述
database_id	bigint	数据库编号
database_name	varchar2(128,ORACLE)	数据库名称
commentstring	varchar2(4000,ORACLE)	数据库的注释信息
database_location	varchar2(4000,ORACLE)	数据库的存储位置
owner_name	varchar2(128,ORACLE)	数据库所属者的名称（一般为创建者）
owner_type	varchar2(10,ORACLE)	数据库所属者的类型

- **dblinks_v**

dblinks_v 系统表存储了 Inceptor 中从一个数据库远程连接另一个数据库的相关信息。

表 15. dblinks_v 的字段

字段名称	类型	描述
dblink_id	bigint	数据库远程连接编号
dblink_name	varchar2(255,ORACLE)	远程连接的名称
service	varchar2(256,ORACLE)	远程连接服务 url
user_name	varchar2(128,ORACLE)	远程连接的用户名称

字段名称	类型	描述
password	char(8,ORACLE)	连接密码
vendor	varchar2(256,ORACLE)	远程连接数据库类型

- **functions_v**

functions_v 系统表存储了 Inceptor 中所有函数的相关信息。

表 16. functions_v 的字段

字段名称	类型	描述
function_name	string	函数名称
parameters	string	函数的参数类型
full_text	string	创建函数的完整 sql 语句
owner_name	varchar2(128,ORACLE)	函数所属者的名称（一般为创建者）
owner_type	varchar2(10,ORACLE)	函数所属者的类型
create_time	timestamp	函数创建时间
database_name	varchar2(128,ORACLE)	函数所属数据库名称

- **materialized_views_v**

materialized_views_v 系统表存储了 Inceptor 中所有物化视图的相关信息。

表 17. materialized_views_v 的字段

字段名称	类型	描述
table_id	bigint	表对应编号
table_name	varchar2(128,ORACLE)	表名称
database_name	varchar2(128,ORACLE)	所属数据库名称
origin_text	string	创建物化视图的原始 sql 语句
expanded_text	string	创建物化视图展开的 sql 语句（如：* 展开为列名等）
source_tables	string	物化视图对应源表名称
rewrite_enabled	boolean	是否可以修改
create_time	timestamp	物化视图创建时间
table_type	varchar2(128,ORACLE)	表类型（MATERIALIZED_VIEW）
owner_name	varchar2(767,ORACLE)	物化视图所属者的名称（一般为创建用户）
commentstring	string	物化视图的注释说明
transactional	string	是否为事务表
input_format	varchar2(4000,ORACLE)	表存储的 input 类型
table_format	string	表存储类型

字段名称	类型	描述
table_location	varchar2(4000,ORACLE)	表存储位置
row_permission	string	行级权限
column_permission	string	列级权限
hbase_name	string	物化视图对应的 hbase 表名
field_delim	string	物化视图中列的分隔符
line_delim	string	物化视图中行的分隔符
collection_delim	string	物化视图中集合列的分隔符

- **packages_v**

packages_v 系统表存储了 Inceptor 中所有 plsql package 的信息。

表 18. packages_v 的字段

字段名称	类型	描述
package_name	string	package 名称
full_text	string	创建 package 的完整 sql 语句
package_body	string	创建 package body 的完整 sql 语句
owner_name	varchar2(128,ORACLE)	package 所属者的名称（一般为创建者）
owner_type	varchar2(10,ORACLE)	package 所属者的类型 （role/user）
create_time	timestamp	package 的创建时间
database_name	varchar2(128,ORACLE)	package 所属数据库名称

- **partitions_v**

partitions_v 系统表存储了 Inceptor 中所有的分区信息。

表 19. partitions_v 的字段

字段名称	类型	描述
partition_id	bigint	分区编号
partition_name	varchar2(255,ORACLE)	分区名称
partition_value	string	分区对应取值
partition_key	string	分区键
table_name	varchar2(128,ORACLE)	分区所属表名称
database_name	varchar2(128,ORACLE)	分区所属数据库名称
location	varchar2(4000,ORACLE)	分区的存储位置

- **procedures_v**

procedures_v 系统表存储了 Inceptor 中所有存储过程的相关信息。

表 20. procedures_v 的字段

字段名称	类型	描述
procedure_name	string	存储过程名称
parameters	string	存储过程的参数信息
full_text	string	创建存储过程时的完整 sql 语句
owner_name	varchar2(128,ORACLE)	存储过程所属者的名称（一般为创建者）
owner_type	varchar2(10,ORACLE)	存储过程所属者的类型（如：role/user）
create_time	timestamp	存储过程的创建时间
database_name	varchar2(128,ORACLE)	存储过程所属数据库名称

- **processes_v**

processes_v 系统表存储了 Inceptor 中所有进程的执行情况。

表 21. processes_v 的字段

字段名称	类型	描述
operation_id	string	sql 语句对应的操作编号
session_id	string	sql 语句执行的会话编号
statement	string	具体执行的 sql 语句
last_access_time	string	上一次的访问时间
duration	string	访问持续时间(ms)
operation_status	string	运行状态
operation_type	string	操作类型
session_ip	string	提交 sql 语句的 IP 地址
user_name	string	提交 sql 语句的用户名称

- **range_partitions_v**

range_partitions_v 系统表存储了 Inceptor 中所有范围分区的相关信息。

表 22. range_partitions_v 的字段

字段名称	类型	描述
partition_id	bigint	范围分区编号
partition_name	varchar2(255,ORACLE)	范围分区名称
partition_range	string	范围分区的范围区间
partition_key	string	范围分区键
table_name	varchar2(128,ORACLE)	范围分区所属表名称
database_name	varchar2(128,ORACLE)	范围分区所属数据库名称
location	varchar2(4000,ORACLE)	范围分区的存储位置

- **serde_parameters_v**

serde_parameters_v 系统表存储了 Inceptor 中所有表中与存储和格式相关的参数信息。

表 23. serde_parameters_v 的字段

字段名称	类型	描述
parameter_key	varchar2(256,ORACLE)	参数 key（如：serialization.format）
parameter_value	string	参数值
database_name	varchar2(128,ORACLE)	参数所属数据库名称
table_name	varchar2(128,ORACLE)	参数所属表名称

- **table_parameters_v**

table_parameters_v 系统表存储了 Inceptor 中所有表的各种属性的参数信息。

表 24. table_parameters_v 的字段

字段名称	类型	描述
parameter_key	varchar2(256,ORACLE)	参数 key（如：transient_lastDdlTime）
parameter_value	string	参数值
database_name	varchar2(128,ORACLE)	参数所属数据库名称
table_name	varchar2(128,ORACLE)	参数所属表名称

- **table_stats_v**

table_stats_v 系统表存储了 Inceptor 中所有 hdfs 表的大小。如果不是hdfs表，则‘table_size’字段为空。

表 25. table_stats_v 的字段

字段名称	类型	描述
table_name	varchar2(128,ORACLE)	表名称
database_name	varchar2(128,ORACLE)	表所属数据库名称
table_format	string	表格式
table_location	varchar2(4000,ORACLE)	表的存储位置
table_size	string	hdfs表的大小

- **tables_v**

tables_v 系统表存储了 Inceptor 中所有表的相关信息。

表 26. tables_v 的字段

字段名称	类型	描述
table_id	bigint	表编号
table_name	varchar2(128,ORACLE)	表名称

字段名称	类型	描述
database_name	varchar2(128,ORACLE)	表所属数据库名称
create_time	timestamp	表的创建时间
table_type	varchar2(128,ORACLE)	表类型
owner_name	varchar2(767,ORACLE)	表所属者的名称（一般为创建者）
commentstring	string	表的注释信息
transactional	string	是否为事务表
last_load_time	timestamp	最后加载时间
input_format	varchar2(4000,ORACLE)	表的输入格式
table_format	string	表格式
table_location	varchar2(4000,ORACLE)	表的存储位置
row_permission	string	行级权限
column_permission	string	列级权限
hbase_name	string	hbase 名称
field_delim	string	表中列的分隔符
line_delim	string	表中行的分隔符
collection_delim	string	表中集合列的分隔符

- **temporary_columns_v**

temporary_columns_v 系统表存储了 Inceptor 中所有临时表的列信息。

表 27. temporary_columns_v 的字段

字段名称	类型	描述
column_name	string	列名
column_type	string	列类型
table_name	string	列所属临时表名称
database_name	string	列所属数据库名称
commentstring	string	列的注释信息
default_value	string	列默认值
table_permission	string	临时表权限
column_length	int	列支持存储数据的字段长度

- **temporary_tables_v**

temporary_tables_v 系统表存储了 Inceptor 中所有临时表的信息。

表 28. temporary_tables_v 的字段

字段名称	类型	描述
table_name	string	临时表名称
database_name	string	临时表所属数据库名称
create_time	string	临时表的创建时间
table_type	string	临时表类型
owner_name	string	临时表所属者的名称（一般为创建者）
table_format	string	临时表格式（如：text，holodesk，…）
table_location	string	临时表的存储位置
row_permission	string	行级权限
column_permission	string	列级权限
field_delim	string	临时表中列的分隔符
line_delim	string	临时表中行的分隔符
collection_delim	string	临时表中集合列的分隔符

- **udfs_v**

udfs_v 系统表存储了 Inceptor 中所有 udf 信息。

表 29. udfs_v 的字段

字段名称	类型	描述
udf_name	string	udf 名称
udf_class	string	udf 对应的 class 路径
udf_type	string	udf 类型（udf：输入单行数据输出单行数据；udaf：输入多行数据输出单行数据；udtf：输入多行数据输出多行数据）
is_native	boolean	是否为本地 udf

- **version_v**

version_v 系统表存储了 Inceptor 中所有的版本信息。

表 30. version_v 的字段

字段名称	类型	描述
schema_version	varchar2(127,ORACLE)	schema版本（如：0.12.53）
tdh_version	string	tdh_version（如：8.31.0）

- **views_v**

views_v 系统表存储了 Inceptor 中所有视图的相关信息。

表 31. views_v 的字段

字段名称	类型	描述
view_id	bigint	视图编号
view_name	varchar2(128,ORACLE)	视图名称
database_name	varchar2(128,ORACLE)	视图所在数据库名称
create_time	timestamp	视图的创建时间
origin_text	string	创建视图的原始 sql 语句
expanded_text	string	创建视图的展开 sql 语句（如：*转换为列名等）
owner_name	varchar2(767,ORACLE)	视图所属者的名称（一般为创建者）

9.6. Inceptor 与 Oracle 语法兼容

接下来将介绍 Oracle（版本：Oracle 21c ）与 Inceptor（版本：Inceptor 9.3）的 Oracle 方言在 SQL 和 PLSQL 语法上的兼容情况，作为 Oracle 迁移到 Inceptor 的指南.

9.6.1. 基本元素

分类	Oracle	Inceptor
对象 Objects	User	User
	role	N/A
	database	database
	schema	database
	tablespace	N/A
	Ordinary Table	Ordinary Table
	Temporary Table	Temporary Table
	External Table	External Table
	Dimension Table	Dimension Table
	Range Partition	Range Partition
	List Partition	Single Value Partition
	Hash Partition	bucket
	View	View
	Materialized View	Materialized View
	Sequence	Sequence
	Synonym	N/A
	DBlink	DBlink
	Index	Global Index

分类	Oracle	Inceptor
表属性 Table Attributes	NOT NULL	NOT NULL
	UNIQUE	N/A
	PRIMARY KEY	N/A
	FOREIGN KEY	N/A
	CHECK	N/A
	CHECK	N/A
	DEFAULT VALUE	DEFAULT VALUE
	Table Comment	Table Comment
	Column Comment	Column Comment

9.6.2. 运算符

分类	Oracle	Inceptor
算数运算符	+, - (as unary op)	+, - (as unary op)
	+, - (as binary op)	+, - (as binary op)
	*, /, %	*, /, %
连接运算符	||	||
分级查询运算符	PRIOR	PRIOR
	CONNECT_BY_ROOT	CONNECT_BY_ROOT
关系运算符	=, !=, <, >, , >=	=, !=, <, >, , >=
	IS [NOT] NULL, LIKE, [NOT] BETWEEN, [NOT] IN, [NOT] EXISTS	IS [NOT] NULL, LIKE, [NOT] BETWEEN, [NOT] IN, [NOT] EXISTS
	IS OF type	N/A
逻辑运算符	AND	AND
	NOT	NOT
	OR	OR

9.6.3. SQL 语法

分类	Oracle	Inceptor
DDL 语法	CREATE, ALTER, DROP	CREATE, ALTER, DROP
	TRUNCATE	TRUNCATE
	GRANT, REVOKE	GRANT, REVOKE
	CREATE TABLE AS	CREATE TABLE AS
	N/A	CREATE TABLE LIKE

分类	Oracle	Inceptor
DML 语法	SELECT	SELECT
	INSERT	INSERT
	UPDATE	UPDATE
	UPDATE SET	UPDATE SET
	MERGE INTO	MERGE INTO
	DELETE	DELETE
	EXPLAIN PLAN	EXPLAIN PLAN
	LOCK TABLE	N/A
DQL 语法	select, from, where, group by, order by, distinct, hint(mapjoin hint), with as	select, from, where, group by, order by, distinct, hint(mapjoin hint), with as
	CONNECT BY, START WITH	CONNECT BY, START WITH
	Order by	Order by
	INNERT JOIN	INNERT JOIN
	CROSS JOIN	CROSS JOIN
	OUTER JOIN	(LEFT|RIGHT|FULL) OUTER JOIN
	ANTI JOIN	(LEFT|RIGHT) ANTISEMI JOIN
	NATURAL JOIN	NATURAL JOIN
	LEFT JOIN	LEFT JOIN
	RIGHT JOIN	RIGHT JOIN
	SEMI JOIN	(LEFT|RIGHT) SEMI JOIN
	UNION	UNION
	UNION ALL	UNION ALL
	INTERSECT	INTERSECT
	INTERSECT ALL	INTERSECT ALL
	MINUS	MINUS
	MINUS ALL	MINUS ALL
	EXCEPT	EXCEPT
	EXCEPT ALL	EXCEPT ALL

9.6.4. 函数和表达式

分类	Oracle	Inceptor
函数 Functions	numeric_function	numeric_function
	character_function	character_function
	datetime_function	datetime_function
	comparison_function	comparison_function
	conversion_function	conversion_function
	collection_function	collection_function
	hierarchical_function	hierarchical_function
	XML_function	XML_function
	JSON_function	JSON_function
	encoding_decoding_function	encoding_decoding_function
	NULL_related_function	NULL_related_function
	Aggregation_function	Aggregation_function
	Window_function	Window_function
Null Expression	expr IS [NOT] NULL	expr IS [NOT] NULL
CASE Expression	CASE expr {WHEN comparison_expr THEN return_expr}+ [ELSE else_expr] END	CASE expr {WHEN comparison_expr THEN return_expr}+ [ELSE else_expr] END
	CASE {WHEN condition THEN return_expr}+ [ELSE else_expr] END	CASE {WHEN condition THEN return_expr}+ [ELSE else_expr] END
Interval Expression	DAY [(leading_field_precision)] TO SECOND [(fractional_second_precision)]	INTERVAL (m-n) DAY TO SECOND
	YEAR [(leading_field_precision)] TO MONTH	INTERVAL (m-n) YEAR TO MONTH

客户服务

技术支持

感谢您使用星环信息科技（上海）股份有限公司的产品和服务。如您在产品使用或服务中有任何技术问题，可以通过以下途径找到我们的技术人员给予解答。

email： support@transwarp.io

技术支持热线电话：4007-676-098

官方网址： http://www.transwarp.cn/

论坛支持： http://support.transwarp.cn/

意见反馈

如果您在系统安装，配置和使用中发现任何产品问题，可以通过以下方式反馈：

email： support@transwarp.io

感谢您的支持和反馈，我们一直在努力！