



Transwarp Data Hub Version 4.8

Inceptor使用手册

星环信息科技（上海）有限公司

版本号 T00148x-03-020, 2017-04-17

目录

1. Inceptor介绍	2
2. Inceptor基础	5
2.1. Inceptor中的对象	5
2.1.1. 数据库	5
2.1.2. 表	5
2.1.2.1. 表的分类	5
2.1.2.1.1. 按Inceptor的所有权分类	6
2.1.2.1.2. 按表的存储格式分类	6
2.1.2.1.3. 按表是否分区分类	7
2.1.2.1.4. 按表是否分桶分类	7
2.1.3. 视图	8
2.1.4. 函数	9
2.2. Inceptor中的数据类型	9
2.2.1. 简单数据类别	9
2.2.2. 复杂数据类别	15
2.2.2.1. ARRAY 示例	15
2.2.2.2. MAP 示例	16
2.2.2.3. STRUCT 示例	17
2.3. Inceptor中的关键字	18
2.3.1. 基本概念	18
2.3.2. 非保留关键词的受限范围	18
2.3.3. 关键词列表	19
2.3.3.1. 保留关键词	19
2.3.3.2. 非保留关键词	26
2.3.4. 数据类型对应表	34
2.3.4.1. Inceptor与Oracle的数据类型对应表	34
2.3.4.2. Inceptor与DB2的数据类型对应表	35
2.3.4.3. Inceptor与JDBC的数据类型对应表	36
2.4. Inceptor交互方法	37
2.4.1. 连接Inceptor	37
2.4.2. 交互语言	38
2.5. 命令行交互	38
2.5.1. 进入命令行	38
2.5.1.1. InceptorServer 1	38
2.5.1.2. InceptorServer 2	39
2.5.2. Transwarp命令行	40
2.5.2.1. Transwarp指令选项	40
2.5.2.2. Transwarp命令行中的非SQL命令	41
2.5.3. Beeline命令行	41
2.5.3.1. beeline指令选项	41
2.5.3.2. beeline命令行中的非SQL指令	42
2.5.4. Inceptor命令行快速入门	42

2.6. Inceptor Library	43
3. Inceptor SQL手册	45
3.1. Inceptor SQL手册一览	45
3.1.1. Inceptor SQL中的语句类型	45
3.1.2. 手册的格式规范	46
3.1.3. Inceptor SQL手册中使用的表	47
3.2. SQL语句的编程规范	48
3.2.1. 注释	48
3.2.2. 大小写规则	48
3.2.3. 缩进与换行	48
3.2.4. 子查询嵌套	49
3.2.5. 表别名	49
3.2.6. 运算符前后间隔要求	50
3.2.7. 临时表	50
3.2.8. GROUP BY / ORDER BY	50
3.2.9. 字段类型	51
3.3. 数据定义语言 (DDL)	51
3.3.1. CREATE/DROP/ALTER DATABASE	51
3.3.1.1. CREATE DATABASE	51
3.3.1.2. DROP DATABASE	51
3.3.1.3. ALTER DATABASE	52
3.3.1.4. USE DATABASE	52
3.3.2. CREATE/DROP/ALTER/TRUNCATE TABLE	53
3.3.2.1. CREATE TABLE	53
3.3.2.1.1. 直接建表	53
3.3.2.1.2. CREATE TABLE LIKE	54
3.3.2.1.3. CREATE TABLE AS SELECT (CTAS)	55
3.3.2.2. DROP TABLE	55
3.3.2.3. ALTER TABLE	55
3.3.2.3.1. 重命名: ALTER TABLE RENAME TO	55
3.3.2.3.2. 修改或添加TBLPROPERTIES	56
3.3.2.3.3. 修改或添加SERDEPROPERTIES	56
3.3.2.3.4. 修改外表目录	56
3.3.2.4. TRUNCATE TABLE	56
3.3.3. CHANGE/ADD/REPLACE COLUMNS	56
3.3.3.1. CHANGE COLUMNS	56
3.3.3.2. ADD REPLACE COLUMNS	57
3.3.4. CREATE/DROP VIEW	58
3.3.4.1. CREATE VIEW	58
3.3.4.2. DROP VIEW	58
3.3.5. CREATE/DROP FUNCTION	58
3.3.5.1. CREATE/DROP TEMPORARY FUNCTION	58
3.3.5.2. CREATE/DROP PERMANENT FUNCTION	59
3.3.6. SHOW	60
3.3.6.1. SHOW DATABASES	60

3.3.6.2. SHOW TABLES	60
3.3.6.3. SHOW TBLPROPERTIES	60
3.3.6.4. SHOW CREATE TABLE	61
3.3.6.5. SHOW PARTITIONS	61
3.3.6.6. SHOW COLUMNS	61
3.3.6.7. SHOW FUNCTIONS	61
3.3.7. DESCRIBE	61
3.3.7.1. DESCRIBE DATABASE	61
3.3.7.2. DESCRIBE TABLE	62
3.3.7.3. DESCRIBE COLUMN	62
3.3.7.4. DESCRIBE PARTITION	62
3.3.7.5. DESCRIBE FUNCTION	62
3.4. 数据操作语言(DML)	62
3.4.1. 导入数据: LOAD	63
3.4.2. 向表插入数据	64
3.4.2.1. 单次插入	64
3.4.2.2. 多次插入	64
3.4.3. 向文件系统中插入数据	65
3.4.3.1. 写入本地文件系统	65
3.4.3.2. 写入HDFS	66
3.4.3.3. Multi-Insert	67
3.4.4. 查询语句	68
3.4.4.1. 过滤: WHERE和HAVING	69
3.4.4.1.1. WHERE子句	69
3.4.4.1.2. HAVING子句	71
3.4.4.2. ORDER BY, SORT BY, DISTRIBUTE BY和CLUSTER BY	71
3.4.4.2.1. ORDER BY	71
3.4.4.2.2. SORT BY	72
3.4.4.2.3. DISTRIBUTE BY与SORT BY合用	72
3.4.4.2.4. CLUSTER BY	72
3.4.4.3. GROUP BY	72
3.4.4.3.1. 单列GROUP BY	73
3.4.4.3.2. GROUP BY (number)	74
3.4.4.3.3. 多列GROUP BY	75
3.4.4.3.4. 用表达式GROUP BY	76
3.4.4.3.5. 在GROUP BY子句中过滤: HAVING 子句	76
3.4.4.4. GROUP BY 扩展: ROLLUP/CUBE/GROUPING SETS	77
3.4.4.4.1. ROLLUP	77
3.4.4.4.2. CUBE	78
3.4.4.4.3. GROUPING SETS	79
3.4.4.5. 多表查询: JOIN	80
3.4.4.5.1. 笛卡尔连接: Cartesian Join	80
3.4.4.5.2. 内连接: INNER JOIN	81
3.4.4.5.3. 外连接: OUTER JOIN	82
3.4.4.5.4. 隐式连接: Implicit JOIN	83

3.4.4.5.5. 自然连接: NATURAL JOIN	83
3.4.4.5.6. 多表连接	84
3.4.4.5.7. 重复连接	84
3.4.4.5.8. 表的自连接	85
3.4.4.5.9. 左半连接和左半反连接	85
3.4.4.5.10. 不等价连接	87
3.4.4.5.11. MAP JOIN	88
3.4.4.6. 子查询	88
3.4.4.6.1. 非关联子查询: Non-Correlated Subqueries	88
3.4.4.6.2. 关联子查询(Correlated Subquery)	90
3.4.4.6.3. 子查询的多层嵌套	91
3.4.4.7. WITH...AS	92
3.4.4.8. 化名	94
3.4.4.8.1. 表化名(Table Alias)	95
3.4.4.8.2. 列化名(Column Alias)	95
3.4.4.9. 集合运算: UNION/INTERCEPT/EXCEPT	96
3.4.4.9.1. UNION 和 UNION ALL	97
3.4.4.9.2. INTERSECT 和 INTERSECT ALL	98
3.4.4.9.3. EXCEPT 和 EXCEPT ALL	99
3.4.4.10. 分页	100
3.4.4.10.1. 本节使用表	100
3.4.4.10.2. 语法	101
3.4.4.11. SQL层次化查询	102
3.4.4.11.1. 简单介绍	102
3.4.4.11.2. 本节使用的表	103
3.4.4.11.3. 用例	103
3.4.4.12. Sequence语法	104
3.4.4.12.1. Sequence的创建、修改、删除、查询	104
3.4.4.12.2. 对Sequence的访问	107
3.4.4.12.3. Sequence的使用实例	108
3.5. 事务控制语言 (TCL)	109
3.5.1. 提交和回滚: COMMIT和ROLLBACK	109
3.5.2. 嵌套事务	110
3.5.3. 自治事务	111
3.6. 数据控制语言 (DCL)	112
3.6.1. 角色管理语法	112
3.6.2. Inceptor对象权限管理语法	113
3.7. Database Links	113
3.7.1. 使用前的准备	114
3.7.2. Database Link语法一览	114
3.7.3. 查看: SHOW DATABASE LINKS	115
3.7.4. 描述: DESCRIBE DATABASE LINK	115
3.7.5. 创建: CREATE DATABASE LINK	115
3.7.5.1. 连接到InceptorServer 1	116
3.7.5.2. 连接到InceptorServer 2	116

3.7.5.3. 连接到Oracle数据库	116
3.7.5.4. 连接到Mysql数据库	117
3.7.5.5. 连接到DB2数据库	117
3.7.5.6. 连接到PostgreSQL数据库	118
3.7.6. 删除: DROP DATABASE LINK	118
3.7.7. 使用Database Link	118
3.8. 数据稽查	119
3.8.1. 本节使用数据	119
3.8.2. 基本介绍	120
3.8.3. 相关语法	120
3.8.4. Error Table的信息	125
3.8.5. Error Table的权限控制	126
3.9. TEXT表	126
3.9.1. 建TEXT表	127
3.9.2. 分隔符	127
3.9.2.1. 单字符分隔符	128
3.9.2.2. 多字符分隔符	130
3.10. CSV表	131
3.10.1. 建CSV外表	131
3.11. ORC表	134
3.11.1. 将要使用的表	135
3.11.2. 建ORC事务表	135
3.11.3. ORC事务表的CRUD	136
3.11.3.1. INSERT	137
3.11.3.2. UPDATE	138
3.11.3.3. DELETE	139
3.11.3.4. MERGE INTO	140
3.11.3.5. CRUD以子查询为目标	141
3.12. Holodesk表	142
3.12.1. 简介	142
3.12.2. Holodesk建表语法	143
3.13. 基于定宽文本文件建外表	146
3.13.1. 简介	146
3.13.2. 建表语法	147
3.13.3. 建表示例	147
3.14. 分区表	149
3.14.1. 单值分区表	149
3.14.1.1. 建表	149
3.14.1.2. 导入数据	151
3.14.1.2.1. 单次插入	151
3.14.1.2.2. 多次插入	152
3.14.1.2.3. 动态分区插入	152
3.14.1.3. 其他单值分区表DDL和DML	153
3.14.1.3.1. 清空分区中的数据	153
3.14.1.3.2. 删除分区	153

3.14.1.3.3. 添加分区	154
3.14.1.3.4. 重命名分区	154
3.14.2. 范围分区表	154
3.14.2.1. 建表	154
3.14.2.2. 导入数据	155
3.14.2.3. ALTER 范围分区表	156
3.14.2.3.1. 添加范围分区	156
3.14.2.3.2. 删除范围分区	157
3.15. 分桶表	157
3.15.1. 建表	157
3.15.2. 向分桶表内导数据	158
4. Inceptor PL/SQL手册 (Oracle 方言)	163
4.1. Inceptor PL/SQL一览	163
4.2. Inceptor PL/SQL手册中的表	165
4.3. 基础知识	166
4.3.1. 声明	166
4.3.1.1. 变量	166
4.3.1.2. 常量	167
4.3.1.3. DEFAULT	168
4.3.2. 命名规则	169
4.3.2.1. 重复命名	169
4.3.2.2. 大小写敏感性	170
4.3.2.3. 命名解析	170
4.3.2.4. 嵌套命名	171
4.3.3. 赋值	172
4.3.3.1. 赋予变量逻辑值	173
4.3.3.2. 赋予变量查询结果	173
4.4. 数据类型	174
4.4.1. 标量类型	174
4.4.1.1. STRINGS	174
4.4.1.2. INT	175
4.4.2. %TYPE属性	176
4.4.3. %ROWTYPE属性	177
4.4.4. 复合类型	178
4.5. 创建PL/SQL语句块	178
4.5.1. PL/SQL语句块组成部分	178
4.5.2. 最简单的匿名块	178
4.5.3. 稍复杂的匿名块	179
4.5.4. 完整的匿名块	179
4.5.5. 匿名块里的嵌套	180
4.6. 流程控制语句	181
4.6.1. IF	182
4.6.1.1. IF-THEN	182
4.6.1.2. IF-THEN-ELSE	182
4.6.1.3. IF-THEN-ELSIF	183

4.6.2. LOOP	185
4.6.2.1. the simple loop	185
4.6.3. WHILE	186
4.6.4. FOR	187
4.6.5. FORALL	189
4.6.5.1. 例一 使用FORALL往表中插入数据	189
4.6.5.2. 例二 使用FORALL查询表中数据	190
4.6.6. EXIT WHEN	191
4.6.7. CONTINUE(WHEN)	192
4.6.8. GOTO	193
4.6.8.1. GOTO语句的限制	195
4.6.9. CASE	195
4.7. PL/SQL存储过程	196
4.7.1. 不带参数的过程	197
4.7.2. 带参数的过程	197
4.7.2.1. IN 类型	198
4.7.2.2. 默认模式	198
4.7.2.2.1. 示例一	198
4.7.2.2.2. 示例二	199
4.7.2.3. OUT类型	200
4.7.2.3.1. 示例一	200
4.7.2.3.2. 示例二	201
4.7.2.4. INOUT类型	202
4.7.2.4.1. 例一 IN与INOUT的对比	202
4.7.2.4.2. 例二 OUT与INOUT的对比	203
4.7.2.4.3. 案例三	204
4.7.2.5. 实参与形参的关联	205
4.7.2.5.1. 位置表示法	205
4.7.2.5.2. 命名表示法	207
4.7.3. 过程重载	209
4.7.4. 系统预定义过程	211
4.8. PL/SQL函数	211
4.8.1. 创建函数	211
4.8.2. 调用函数	212
4.8.2.1. 单独调用函数	212
4.8.2.2. 在过程中调用函数	212
4.8.2.3. SQL语句中调用函数	213
4.8.2.4. 函数的嵌套调用	214
4.8.3. 函数重载	214
4.8.4. 系统预定义函数	216
4.9. Records	216
4.9.1. Records语法	217
4.9.2. Records创建及使用	217
4.9.2.1. 方法一：自定义每一个分量的名字和类型	217
4.9.2.2. 方法二：基于表，定义每一个分量的名字和类型	218

4.9.2.3. 方法三：利用%ROWTYPE属性快速声明一个记录型变量	219
4.10. Collections	220
4.10.1. VARRAY	220
4.10.1.1. VARRAY语法	221
4.10.1.2. VARRAY声明与赋值	221
4.10.1.3. VARRAY类型的使用范围	222
4.10.2. NESTED TABLE	223
4.10.2.1. NESTED TABLE语法	223
4.10.2.2. 表类型变量的声明与赋值	223
4.10.3. Associative arrays	224
4.10.3.1. Associative arrays语法	224
4.10.3.2. Associative arrays变量的声明和赋值	225
4.10.4. Collections 方法	225
4.10.4.1. COUNT()	226
4.10.4.1.1. VARRAY	226
4.10.4.1.2. NESTED TABLE	226
4.10.4.1.3. Associative arrays	227
4.10.4.2. FIRST() 和 LAST()	228
4.10.4.2.1. VARRAY	228
4.10.4.2.2. Associative arrays	228
4.10.4.3. PRIOR(n) 和 NEXT(n)	229
4.10.4.4. EXTEND(k)	231
4.10.4.4.1. VARRAY	231
4.10.4.4.2. NESTED TABLE	233
4.10.4.5. TRIM(k)	234
4.10.4.5.1. VARRAY	235
4.10.4.5.2. NESTED TABLE	235
4.10.4.5.3. Associative arrays	236
4.10.4.6. DELETE(k)	237
4.10.4.6.1. VARRAY	237
4.10.4.6.2. NESTED TABLE	238
4.10.4.6.3. Associative arrays	239
4.10.4.7. EXISTS()	239
4.10.4.7.1. Varray	239
4.10.4.7.2. Associative Arrays	240
4.10.4.8. LIMIT()	241
4.11. 游标 (Cursors)	241
4.11.1. 显式游标	242
4.11.1.1. 不带参数的显式游标使用示例	242
4.11.1.2. 带参数的显式游标使用示例	245
4.11.2. 隐式游标	248
4.11.2.1. 隐式游标使用示例	249
4.11.3. 游标变量	249
4.11.3.1. 游标变量使用示例	250
4.11.3.2. 强类型游标变量	250

4.11.3.2.1. 弱类型游标变量	251
4.11.3.2.2. 游标变量作为参数传递	252
4.12. 与SQL的交互	253
4.12.1. PL/SQL过程与SQL的交互	253
4.12.1.1. INSERT	253
4.12.1.2. DELETE	254
4.12.1.3. UPDATE	255
4.12.1.4. MERGE	256
4.12.1.4.1. 仅满足条件下，更新	256
4.12.1.4.2. 满足条件更新，不满足条件插入	257
4.12.2. PL/SQL函数与SQL的交互	258
4.12.3. 隐式游标	259
4.12.4. BULK COLLECT	260
4.12.4.1. 在fetch into中使用bulk collect	260
4.12.4.2. 在select into中使用bulk collect	261
4.12.5. 动态SQL	262
4.12.5.1. EXECUTE IMMEDIATE	262
4.12.5.1.1. 处理DDL语句	262
4.12.5.1.2. 处理DCL语句	264
4.12.5.1.3. 处理单行查询	265
4.12.5.1.4. 处理不含占位符的DML语句	265
4.12.5.1.5. 处理占位符的DML语句	267
4.12.5.2. 使用OPEN-FOR, FETCH和CLOSE语句	270
4.12.5.2.1. 语法	270
4.12.5.2.2. 示例	271
4.13. Packages	271
4.13.1. Packages的创建	272
4.13.1.1. 语法	272
4.13.1.1.1. 创建包头	272
4.13.1.1.2. 创建包体	272
4.13.1.1.3. Packages的调用	272
4.13.1.2. 实例	273
4.13.1.2.1. 创建包头	273
4.13.1.2.2. 创建包体	273
4.13.1.2.3. 包的调用	273
4.13.2. Packages的使用案例	274
4.13.2.1. 仅创建Packages包头	274
4.13.2.2. 创建Packages包头和包体	275
4.13.3. 系统预定义包	277
4.14. 预定义函数/过程/包	277
4.14.1. 语法	277
4.14.2. 案例	279
4.14.2.1. set_env与get_env	279
4.14.2.2. sqlcode与sqlerrm	280
4.14.2.3. get_columns	281

4.14.2.4. raise_application_error	281
4.14.2.5. PUT_LINE	282
4.14.2.6. dbms_output	284
4.14.2.7. owa_util	284
4.15. 异常	285
4.15.1. 支持的系统预定义异常	285
4.15.1.1. NO_DATA_FOUND	285
4.15.1.2. TOO_MANY_ROWS	287
4.15.1.3. CURSOR_ALREADY_OPEN	288
4.15.1.4. ROWTYPE_MISMATCH	290
4.15.1.5. SUBSCRIPT_BEYOND_COUNT	292
4.15.1.6. SUBSCRIPT_OUTSIDE_LIMIT	294
4.15.1.7. COLLECTION_IS_NULL	295
4.15.1.8. INVALID_CURSOR	297
4.15.2. 暂不支持的系统预定义异常	298
4.15.2.1. INVALID_NUMBER	299
4.15.2.2. VALUE_ERROR	299
4.15.2.3. ZERO_DIVIDE	300
4.15.2.4. DUP_VAL_ON_INDEX	301
4.15.2.5. CASE_NOT_FOUND	302
4.15.2.6. ACCESS_INTO_NULL	303
4.15.2.7. SELF_IS_NULL	304
4.15.2.8. SYS_INVALID_ROWID	304
4.15.2.9. NOT_LOGGED_ON	305
4.15.2.10. LOGIN_DENIED	305
4.15.2.11. 其它不支持的异常	306
4.15.3. 用户自定义异常	306
4.15.4. 嵌套异常	307
4.15.4.1. 异常处理	307
4.15.4.2. 异常分别出现在inner block和outer block里	308
4.15.4.2.1. 内部和外部的异常处理块，分别处理inner block和outer block里的异常	308
4.15.4.2.2. 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	308
4.15.4.3. 异常发生在inner block中	309
4.15.4.3.1. 内部块中的异常处理块处理inner block里的异常	309
4.15.4.3.2. 外部块中的异常处理块处理inner block里的异常	310
4.15.4.3.3. 没有异常处理块处理inner block里的异常	311
4.15.4.4. 异常发生在outer block中	311
4.15.4.4.1. 外部块中的异常处理块处理outer block里的异常	311
4.15.4.4.2. 内部块中的异常处理块处理outer block里的异常	312
4.16. 注意事项	313
4.16.1. 函数/存储过程的版本兼容	313
4.16.2. Inceptor对PL/SQL中分号的支持	313
4.16.2.1. Beeline+InceptorServer 2	313
4.16.2.2. CLI+InceptorServer 1	314

4.16.3. PL/SQL中结果的打印	315
4.16.4. 标识符	316
4.16.4.1. SELECT/SELECT INTO	316
4.16.4.1.1. 案例一:SELECT/SELECT INTO语句后紧跟函数调用	316
4.16.4.1.2. 案例二:SELECT/SELECT INTO语句后紧跟赋值语句	317
4.16.4.2. FOR LOOP ... END LOOP	318
4.16.4.2.1. 案例一:END LOOP后紧跟函数调用	318
4.16.4.2.2. 案例二:END LOOP后紧跟赋值语句	319
4.16.5. 标准SQL调用PL/SQL函数必须满足的条件	320
4.16.5.1. 必须是函数,不能是过程	320
4.16.5.1.1. 不支持SQL语句中调用PL/SQL过程	320
4.16.5.2. PL/SQL函数返回值必须是基本类型	322
4.16.5.2.1. PL/SQL函数返回值必须是基本类型	322
4.16.5.2.2. 不支持PL/SQL函数返回值是非基本类型	322
4.16.5.3. PL/SQL函数中不能有标准SQL语句	322
4.16.6. 不支持RETURN INTO语句	323
4.16.7. 查看预定义函数/过程/包	323
4.16.7.1. 相关命令合集	323
4.16.7.2. 案例合集	325
4.16.8. PL/SQL中的PUT_LINE打印	327
4.16.9. Hive异常的处理	330
4.16.10. Debug	332
5. Inceptor SQL PL手册 (DB2 方言)	335
5.1. Inceptor SQL PL一览	335
5.1.1. 输出	335
5.1.2. 声明	335
5.1.3. 赋值	335
5.1.4. 过程	336
5.1.5. 函数	336
5.1.6. 游标	336
5.1.7. 异常	337
5.1.8. 动态SQL	337
5.2. Inceptor SQL PL手册中的表	338
5.3. 基础知识	339
5.3.1. 快速入门	339
5.3.2. 声明	340
5.3.2.1. 基本变量的声明	340
5.3.2.1.1. STRING	340
5.3.2.1.2. INT	340
5.3.2.1.3. Boolean	341
5.3.2.1.4. DATE	341
5.3.2.1.5. TIMESTAMP	341
5.3.2.1.6. STATEMENT	341
5.3.2.2. 常量的声明	342
5.3.2.3. 重复命名	343

5.3.2.4. 大小写敏感性	343
5.3.2.5. 嵌套命名	344
5.3.3. 赋值	344
5.3.3.1. SET	344
5.3.3.2. VALUES INTO	345
5.3.3.3. SELECT ... INTO	346
5.4. 数据类型	346
5.4.1. 内置类型	346
5.4.1.1. STRING	346
5.4.1.2. INT	347
5.4.1.3. Boolean	348
5.4.1.4. DATE	349
5.4.1.5. TIMESTAMP	350
5.4.1.6. STATEMENT	350
5.4.2. 复合类型	351
5.4.3. 锚定类型	351
5.4.4. 特殊类型	351
5.5. 创建SQL PL语句块	351
5.5.1. 创建示例	351
5.6. 流程控制语句	352
5.6.1. IF	352
5.6.1.1. IF-THEN	352
5.6.1.2. IF-THEN-ELSE	353
5.6.1.3. IF-THEN-ELSIF	354
5.6.2. FOR	355
5.6.3. LOOP	356
5.6.3.1. the simple loop	356
5.6.4. WHILE	357
5.6.5. REPEAT	357
5.6.6. GOTO	358
5.6.7. LEAVE	359
5.6.8. ITERATE	360
5.6.9. CASE	361
5.7. 存储过程	362
5.7.1. 语法	362
5.7.2. 参数的选择	362
5.7.2.1. 不带参数	362
5.7.2.1.1. 例1:不带参数	362
5.7.2.2. IN 参数	363
5.7.2.2.1. 例2:IN参数	363
5.7.2.3. OUT 参数	363
5.7.2.3.1. 例3:OUT参数	363
5.7.2.3.2. 例4:OUT参数	364
5.7.2.3.3. 例5: IN, OUT参数	365
5.7.2.4. INOUT 参数	365

5.7.2.4.1. 例6:INOUT参数与OUT参数	365
5.7.2.4.2. 例7:INOUT参数与IN参数	366
5.7.2.4.3. 例8:INOUT参数综合运用	367
5.7.2.5. 不指定参数类型	368
5.7.2.5.1. 例9:不指定参数类型	368
5.7.3. 属性	368
5.7.4. 过程重载	369
5.8. SQLPL函数	370
5.8.1. 创建函数	370
5.8.2. 参数选择	370
5.8.2.1. IN	370
5.8.2.2. OUT	371
5.8.2.3. INOUT	371
5.8.3. 函数属性	373
5.8.4. 调用函数	373
5.8.4.1. 单独调用函数	373
5.8.4.2. 在存储过程内调用函数	374
5.8.4.3. SQL语句中调用函数	375
5.8.4.4. SQLPL函数中对其他函数的调用	375
5.8.5. 标准SQL调用SQLPL函数必须满足的条件	376
5.8.5.1. 必须是函数, 不能是过程	376
5.8.5.2. SQLPL函数返回值必须是基本类型	377
5.8.5.2.1. SQLPL函数返回值必须是基本类型	377
5.8.5.2.3. SQLPL函数中不能有标准SQL语句	377
5.8.6. 函数重载	378
5.9. VARRAY	379
5.9.1. 定义数组类型及变量	379
5.9.2. 数组变量的赋值	379
5.9.2.1. SET	379
5.9.2.2. VALUES ... INTO	380
5.9.3. 数组变量的方法调用	380
5.9.3.1. ARRAY_FIRST	381
5.9.3.2. ARRAY_LAST	381
5.9.3.3. ARRAY_NEXT	381
5.9.3.4. ARRAY_PRIOR	382
5.9.3.5. ARRAY_DELETE	382
5.9.3.6. CARDINALITY	383
5.10. 行数据类型	383
5.10.1. 语法	384
5.10.2. 声明与赋值	384
5.10.2.1. 例一	384
5.10.2.2. 例二	385
5.10.2.3. 例三	386
5.10.2.4. 例四	386
5.10.2.5. 例五	387

5.10.3. 基于表的行数据类型的声明	388
5.11. 游标	388
5.11.1. 组成部分	388
5.11.2. 基本用法	389
5.11.3. ALLOCATE CURSOR	389
5.11.3.1. 语法	389
5.11.3.2. 用例说明	390
5.12. 与SQL的交互	391
5.12.1. 将以SELECT返回值赋值给变量	391
5.12.2. SELECT INTO	391
5.12.3. 存储过程与SQL的交互	392
5.12.4. 函数与SQL的交互	392
5.12.5. 游标	393
5.12.6. 动态SQL	394
5.12.6.1. EXECUTE IMMEDIATE	394
5.12.6.1.1. 处理DDL语句	394
5.12.6.1.2. 处理DCL语句	395
5.12.6.1.3. 处理单行查询	396
5.12.6.1.4. 处理不含占位符的DML语句	397
5.12.6.1.5. 处理含有占位符的DML语句	399
5.12.6.2. PREPARE … EXECUTE	400
5.12.6.2.1. SELECT	400
5.12.6.2.2. INSERT	401
5.12.6.2.3. UPDATE	402
5.12.6.2.4. DELETE	403
5.12.7. GET DIAGNOSTICS	403
5.12.7.1. ROW_COUNT	403
5.12.7.2. EXCEPTION 1	404
5.13. 系统预定义函数/过程	405
5.13.1. 查看预定义函数/过程	405
5.13.1.1. 相关命令合集	405
5.13.1.2. 案例合集	406
5.13.2. 预定义函数/过程/包的介绍	408
5.13.3. 预定义函数/过程的使用	409
5.13.3.1. set_env与get_env	409
5.13.3.2. sqlcode与sqlerrm	409
5.13.3.3. sqlerrm(int)	410
5.13.3.4. PUT_LINE	410
5.13.3.5. raise_application_error	411
5.13.3.6. get_columns	412
5.14. 异常	413
5.14.1. SQLCODE与SQLSTATE	413
5.14.2. SIGNAL	415
5.14.3. 异常分类	416
5.14.3.1. 预定义异常	416

5.14.3.2. 自定义异常	416
5.14.3.2.1. 语法	416
5.14.3.2.2. 实例	416
5.14.3.3. 异常处理	417
5.14.3.4. 实例	418
5.14.3.4.1. 自定义异常的处理	418
5.14.3.4.2. 预定义异常的处理	420
5.14.4. RESIGNAL	421
5.14.4.1. 语法	421
5.14.4.2. 用例	422
5.14.5. DEBUG	423
5.15. 注意事项	424
5.15.1. 函数/存储过程的版本兼容	424
5.15.2. 方言的选择	424
5.15.2.1. Beeline+InceptorServer 2	425
5.15.2.2. CLI+InceptorServer 1	425
5.15.3. 分号的支持	425
5.15.4. PL/SQL中结果的打印	425
6. Inceptor函数和运算符手册	427
6.1. Inceptor函数和运算符手册中的表	427
6.2. 关系运算符	428
6.3. 算术运算符	431
6.4. 逻辑运算符	434
6.5. 数学函数	435
6.6. 类型转换函数	444
6.7. 日期函数	445
6.8. 条件函数	457
6.9. 字符串函数	458
6.10. XML函数	474
6.11. 复杂类型函数	478
6.12. 表生成函数	482
6.12.1. LATERRAL VIEW	483
6.13. 聚合函数	485
6.14. 窗口函数	490
6.14.1. OVER子句	490
6.14.2. 窗口函数详解	492
6.15. Context函数	496
6.15.1. current_user	496
6.15.2. has_role	497
6.15.3. current_database	498
6.15.4. current_time	499
6.15.5. current_date	499
6.15.6. current_timestamp	499
6.16. 脱敏相关函数	499
6.17. 其他函数和运算符	500

7. Inceptor多租户手册	506
7.1. Inceptor权限和角色管理	506
7.1.1. Inceptor对用户身份的判断	506
7.1.2. Inceptor管理员身份的获取	506
7.1.3. Inceptor角色管理	507
7.1.3.1. CREATE ROLE: 创建角色	507
7.1.3.2. DROP ROLE: 删除角色	507
7.1.3.3. SHOW CURRENT ROLES: 查看用户当前所有的角色	508
7.1.3.4. SET ROLE: 设置角色	508
7.1.3.5. SHOW ROLES: 查看所有Inceptor中的角色	509
7.1.3.6. GRANT ROLE: 授予角色	509
7.1.3.7. REVOKE ROLE: 收回角色	510
7.1.3.8. SHOW ROLE GRANT: 查看用户/角色拥有的角色	511
7.1.3.9. SHOW PRINCIPALS: 查看拥有某个角色的用户和角色	512
7.1.4. Inceptor权限管理	512
7.1.4.1. GRANT: 权限的授予	514
7.1.4.1.1. 授予全局权限	514
7.1.4.1.2. 授予数据库级别的权限	515
7.1.4.1.3. 授予表和视图级别的权限	515
7.1.4.2. REVOKE: 权限的收回	516
7.1.4.2.1. 收回全局权限	516
7.1.4.2.2. 收回数据库级别的权限	517
7.1.4.2.3. 收回表和视图级别的权限	517
7.1.4.3. SHOW: 权限的查看	518
7.1.5. Inceptor权限传播控制	519
7.1.5.1. 设置	519
7.1.6. Inceptor操作所需权限总结	521
7.2. Inceptor中的行级和列级权限	523
7.2.1. 列级和行级权限设置操作一览	523
7.2.2. 行级和列级权限 (PERMISSION) 对比表级权限 (PRIVILEGE)	524
7.2.3. 行级权限的设置	524
7.2.3.1. 用法1: 将权限直接设置给用户	524
7.2.3.2. 用法2: 使用一张辅助表设置行级权限	527
7.2.4. 行级权限的取消	529
7.2.5. 列级权限的设置	529
7.2.5.1. 背景	529
7.2.5.2. 授权	530
7.2.5.3. 不同用户查看表	531
7.2.6. 列级权限的取消	531
7.2.7. 行级和列级权限的查看	532
7.3. Inceptor中设置HDFS文件的ACL	532
7.3.1. 设置用户/组对某张表的FACL	532
7.3.2. 取消用户/组对某张表的FACL	533
7.3.3. 取消某张表上的全部FACL	533
7.3.4. 查看用户/组对某张表的FACL	534

7.3.5. 查看某张表上的所有FACL	534
7.4. 空间配额管理操作	535
7.4.1. 语法总结	535
7.4.1.1. 设置配额	535
7.4.1.1.1. 查看配额	535
7.4.1.1.2. 取消配额	536
7.4.2. 示例操作	536
7.4.2.1. 数据库数据空间配额管理	536
7.4.2.2. 用户空间配额管理	537
7.4.2.3. 临时空间配额管理	538
7.4.2.3.1. 用户临时空间配额管理	538
7.4.2.3.2. 所有临时空间配额管理	538
8. Inceptor JDBC手册	540
8.1. Inceptor JDBC手册一览	540
8.1.1. 获取Inceptor JDBC驱动	540
8.2. 应用程序中的JDBC交互	541
9. Inceptor ODBC手册	547
9.1. Inceptor ODBC手册一览	547
9.2. Windows操作系统中的ODBC交互准备	547
9.2.1. 安装ODBC驱动	547
9.2.2. 修改配置文件和系统变量	547
9.2.3. 创建合适的DSN	551
9.2.3.1. 连接无需认证的InceptorServer 1的DSN	551
9.2.3.2. 连接无需认证的InceptorServer 2的DSN	555
9.2.3.3. 连接LDAP认证的InceptorServer 2的DSN	558
9.2.3.4. 连接Kerberos认证的InceptorServer 2的DSN	561
9.2.3.5. DSN高级选项	565
9.3. 在Linux系统中配置ODBC环境	568
9.3.1. 配置前的准备	568
9.3.2. 安装ODBC Driver Manager	568
9.3.3. 安装ODBC驱动	569
9.3.3.1. 在Ubuntu系统中安装	570
9.3.3.2. 在CentOS系统中安装	570
9.3.3.3. 在SUSE系统中安装	570
9.3.4. 配置ODBC驱动	571
9.3.5. 配置DSN	572
9.3.6. SASL插件的安装	574
9.3.6.1. Ubuntu系统	574
9.3.6.2. CentOS系统	574
9.3.6.3. SUSE系统	574
9.3.7. 测试连接	574
7.4. 在Linux系统中安装和配置Kerberos客户端	576
9.4.1. 在Ubuntu系统中安装和配置Kerberos客户端	576
9.4.2. 在CentOS或者SUSE系统中安装和配置Kerberos客户端	578
9.5. 应用程序中的ODBC交互	579

10. Inceptor外部工具连接手册	582
10.1. Inceptor外部工具连接手册一览	582
10.2. 使用JDBC的外部工具连接	582
10.2.1. 连接前的准备	582
10.2.2. DbVisualiser连接Inceptor	582
10.2.2.1. 添加驱动	582
10.2.2.1.1. 连接InceptorServer 1的驱动	582
10.2.2.1.2. 连接InceptorServer 2的驱动	585
10.2.2.2. 创建连接	587
10.2.2.2.1. 无需认证的InceptorServer 1	587
10.2.2.2.2. 需要认证的InceptorServer 2	589
10.2.2.3. 通过DbVisualiser创建并查看存储过程	592
10.2.4. Squirrel SQL连接Inceptor	594
10.2.4.1. 添加驱动	595
10.2.4.1.1. 连接InceptorServer 1的驱动	595
10.2.4.1.2. 连接InceptorServer 2的驱动	597
10.2.4.2. 添加alias	600
10.2.4.2.1. 无需认证的InceptorServer 1的alias	601
10.2.4.2.2. 无需认证的InceptorServer 2的alias	605
10.2.4.2.3. LDAP认证的InceptorServer 2的alias	608
10.2.4.2.4. Kerberos认证的InceptorServer 2的alias	612
10.3. 使用ODBC的外部工具连接	613
10.3.1. Tableau连接Inceptor	613
11. Inceptor运维手册	618
11.1. Inceptor的配置	618
11.1.1. InceptorServer 1	618
11.1.2. 配置InceptorServer 2	620
11.1.2.1. LDAP认证模式	620
11.1.2.2. Kerberos认证模式	621
11.2. 多Inceptor的安装配置	622
11.3. Inceptor CPU和内存资源分配	628
附录 A: Inceptor字段规范	633
12. 客户服务	634

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)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)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: Hadoop®和SPARK®是ApacheTM 软件基金会在美国和其他国家的商标或注册的商标。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 © 2013年-2017年星环信息科技(上海)有限公司。保留所有权利。

©星环信息科技(上海)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)有限公司所有。未得到星环信息科技(上海)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。

手册版本信息

版本号: T00148x-03-020

发布日期: 2017-04-17

1. Inceptor介绍

TranswarpInceptor是星环科技推出的用于数据仓库和交互式分析的大数据平台软件，它基于Hadoop和Spark技术平台打造，加上自主开发的创新功能组件，有效的解决了企业级大数据数据处理和分析的各种技术难题，帮助企业快速的构建和推广数据业务。

TranswarpInceptor可提供完整的SQL支持，支持主流的SQL模块化扩展，兼容通用开发框架和工具，支持事务特性保证数据的准确性，允许多租户的隔离与管理，且能够利用内存或者SSD来加速数据的读取，支持与关系型数据库实时对接并做统计分析，辅以高性能的SQL执行引擎，为企业提供高性价比和高度可扩展的解决方案。



图 1. Inceptor 架构

Inceptor中星环科技自主开发的创新组件包括：

- **SQL编译器 SQL 2003 Compiler**

企业级数据仓库、数据集市等应用大多基于SQL开发，而Hadoop业界的产品大部分对SQL的兼容程序比较差，或者不支持SQL的模块化扩展，因而应用迁移的成本非常高，甚至是不具备可行性。

为了降低应用迁移成本，Transwarp Inceptor开发了完整的SQL编译器，支持ANSI SQL 92和SQL 99标准，支持ANSI SQL 2003 OLAP核心扩展，可以满足绝大部分现有的数据仓库业务对SQL的要求，方便应用平滑迁移。

除了更好的SQL语义分析层，Inceptor还包含强大的优化器保证SQL在引擎上有最佳的性能。Inceptor包含3级优化器：首先是基于规则的优化器，应用静态优化规则生成一个优化的逻辑执行计划；其次是基于成本的优化器，通过衡量多个不同执行计划的CPU、IO和网络成本，选择一个相对更合理的计划并生成物理执行计划；最后是代码生成器，对一些比较核心的执行逻辑生成更高效的执行代码或者Java Byte Code，从而保证SQL业务在分布式平台上具有最佳性能。

- **存储过程编译器 PL/SQL Compiler**

国内现有的数据仓库应用大都基于SQL 2003，且大量使用存储过程来构建复杂应用。因此为满足需求，除了SQL编译器以外，Transwarp Inceptor还包含存储过程编译器用于对存储过程的编译和执行。

Inceptor支持SQL2003标准，存储过程，兼容Oracle、DB2、Teradata方言，包括完整的数据类型、流程控制、Package、游标、异常处理以及动态SQL执行，并且支持在存储过程中做高速统计，增删改查与分布式事务操作。因此，有了存储过程编译器的补充，Inceptor可以满足绝大部分数据应用的从关系型数据库到Inceptor平台的迁移。

除了SQL语法层面的支持，存储过程编译器包含一个完整的优化器，其中包括CFG Optimizer，Parallel Optimizer，和DAG Optimizer。CFG Optimizer主要用于对存储过程中的代码进行优化，完成循环展开，冗余代码消除，函数内联等主要优化。Parallel Optimizer用于将一些原本串行的逻辑做并行化处理，利用集群的计算能力来提高整体执行速度，对一些关键的功能如游标的性能提升非常明显。DAG Optimizer会根据生成的DAG进行二次优化，生成更合理的物理执行计划，重点降低Shuffle等任务开销。

为了有效兼容其他数据库，Inceptor支持通过不同的方言设置来隔离不同的SQL标准之间的差异，从而避免数据计算和处理标准的二义性，保证数据处理的正确性。

• 事务管理单元 Transaction Manager

为了更好的满足数据仓库业务场景的需求，Inceptor提供完整的增删改SQL支持，允许从多数据源中加工数据。同时为了有效的保证数据处理的准确性，Inceptor提供分布式事务的支持，保证了处理过程中数据的ACID（原子性、一致性、隔离性和持久性）。

Inceptor支持以Begin Transaction启动事务，以commit或者rollback来结束事务。事务管理单元通过两阶段封锁协议和MVCC来实现一致性和隔离性的控制，支持Serializable Snapshot Isolation隔离级别，因而可以保证并发情况下的事务一致性。

Inceptor支持SQL 2003中关于增删改查部分的语义规范，支持Insert，Update，Delete，Truncate以及Merge Into原语，支持单条或者从其他数据表以及嵌套查询中更新数据表，并且内置一致性检查功能以防止非法改动。

通过SQL编译器的优化，增删改SQL执行计划通过分布式引擎在集群中并发执行，系统整体的吞吐率可达关系数据库的数倍，能够满足批处理业务的高吞吐率要求。另外，通过合理的资源规划，Inceptor在做数据的增删改的同时，允许租户对数据做高速的统计分析。

• 分布式内存列式存储 Holodesk

为了加速交互式分析的速度，Inceptor推出了基于内存或者SSD的列式存储引擎Holodesk。Holodesk把数据在内存或者SSD中做列式存储，辅以基于内存的执行引擎，可以完全避免IO带来的延时，极大的提高数据扫描速度。

除了列式存储加快统计分析速度，Holodesk支持为数据字段构建分布式索引。通过智能索引技术为查询构建最佳查询方案，Inceptor可以将SQL查询延时降低到毫秒级。

Holodesk允许用户对多字段组合构建OLAP-Cube，并将cube直接存储于内存或者SSD，无需额外的BI工具构建Cube，因此对于一些复杂的统计分析和报表交互查询，Holodesk可实现秒级的反应。

除了性能优势，Holodesk在可用性方面也表现出色。Holodesk的元数据和存储都原生支持高可用性，通过一致性协议和多版本来支持异常处理和灾难恢复。在异常情况下，Holodesk能够自动恢复重建所有的表信息和数据，无需手工恢复，从而减少开发与运维的成本，保证系统的稳定性。

Inceptor重点优化了基于SSD的Holodesk性能，使得基于PCIE SSD的性能达到全内存方案的80%以上。因此结合使用低成本的内存、闪存混合存储方案，可接近全内存存储的分析性能，保证解决方案的高性价比。

- 分布式执行引擎 **Distributed Execution Engine**

Inceptor基于Apache Spark深度开发了专用分布式计算引擎，不仅大幅提高了计算性能，而且有效的解决了Spark在稳定性方面的很多问题，确保计算引擎能够7x24小时无间断运行。此外，Inceptor引擎独立构建了分布式数据层，将计算数据从计算引擎JVM内存空间中独立出来，可以有效减少JVM GC对系统性能和稳定性的影响。

在SQL执行计划优化方面，Inceptor实现了基于代价的优化器和基于规则的优化器，辅以100多种优化规则，以保证SQL应用在无需手工改动的情况下能够发挥最大的性能。对于数据倾斜等常见的数据处理难题，执行引擎也能够自动识别并加以优化，可解决绝大部分存在数据倾斜的计算场景，杜绝数据倾斜对系统稳定性的影响。

为了更好的适应各种数据场景，Inceptor的执行引擎包含两种执行模式：低延时模式和高吞吐模式。低延时模式主要应用在数据量比较小的场景，执行引擎会生成执行延时低的物理执行计划，通过减少或避免一些高延时的任务（如IO，网络等）来保证SQL较短的执行时间，达到或者逼近关系型数据库在这些场景下的性能。高吞吐模式主要应用在大数据的场景，通过合理的分布式执行来提高超大数据量上的复杂统计分析的性能。因此，Inceptor的执行引擎可以满足从GB到PB的各种数据量上的数据业务需求。

- 数据源连接器 **Stargate**

企业数据可能会分散在多个系统中，彼此无法共享数据或者进行相关的分析，从而造成数据孤岛的现象。构建统一的大数据平台可以有效的解决大部分场景下的数据孤岛问题。采用统一的大数据平台之后，会存在一些数据因为各种关系无法迁移统一平台上的现象。为了解决此类问题，Inceptor推出了数据源连接器Stargate。

Stargate是连接执行引擎和各种数据源的连接器，可将多种不同数据源的数据接入引擎做实时的统计分析，无需事先将数据导入HDFS，从而方便用户的业务构建多样化需求。

在语法层面，Inceptor兼容Oracle DB-Link规范，通过创建database link来预先建立和其他数据源的连接池，接着就可以在SQL中通过 table_name@database link的方式在Inceptor中实时访问该数据源的数据，无需其他操作。在执行计划开始后，Stargate通过预先建立的连接从其他数据源中抽取需要的数据，输入进入执行引擎层参与SQL计算。在计算完成后，释放相关的数据库连接以及对应的资源。

目前Stargate支持关系数据库包括Oracle，DB2，Mysql，Teradata以及PostgreSQL。此外，Stargate目前可以接入Holodesk，HDFS，Hyperbase等平台内数据源，未来将支持Redis等为数据源。

- 中间件管理单元 **Connector**

Inceptor完整的支持JDBC4.0和ODBC3.5标准，因此能够支持Hibernate/Spring等中间件，完全兼容Tableau/QlikView/Cognos等报表工具，可以和企业当前的数据应用层完整对接。

此外，Inceptor也支持与其他数据同步工具的对接，已经完成了和IBM CDC的相互认证与整合，并且能够支持Oracle Golden Gate、SAP Data Service等工具。因此，企业用户可以实时的将交易数据同步到Inceptor内做交互式统计分析业务。

2. Inceptor基础

2.1. Inceptor中的对象

在Inceptor中，您可以使用常见的数据库对象，包括数据库（database），表（table），视图（view）和函数（function）。您可以使用Inceptor SQL、Inceptor PL/SQL以及Inceptor SQL PL来操作这些数据库对象。Inceptor中数据库对象的元数据保存在Inceptor Metastore中，而数据库对象内的数据可以存放在：

- 内存或者SSD中（Holodesk表）
- HDFS中（TEXT表/ORC表/CSV表）

另外，Inceptor的SQL引擎还可以对接TDH中的其他分布式存储，如Transwarp HBase和Transwarp ES。这些不在本手册的讨论范围内，如果您有兴趣可以参考《Transwarp Data Hub Hyperbase使用手册》。

本章我们将对Inceptor数据库对象进行总览，具体细节在本手册的其他部分讨论。

2.1.1. 数据库

在Inceptor中，数据库是存放一组表的目录。Inceptor有一个默认的数据库default，用户也可以在Inceptor中创建其他数据库。

如果您使用InceptorServer 2，通过beeline连接Inceptor时可以指定连接的数据库：

```
beeline -u "jdbc:hive2://<server_ip/hostname>:10000/<database>"
```



安全模式下，连接InceptorServer 2的连接串需要提供认证信息，具体请参考[Inceptor多租户手册](#)。

在所在数据库内可以直接使用对象名指代对象，如果要使用其他数据库中的对象，需要使用<database_name>. <object_name> 来指代，例如 my_db.my_table。您也可以用 USE 指令切换使用的数据库。

每个Inceptor中的数据库都是HDFS上的目录，路径为：

```
hdfs://<nameservice>/<id>/user/hive/warehouse/<database_name>.db
```

<nameservice> 是HDFS的nameservice名称；<id> 是Inceptor的服务名；<database_name> 是数据库名。数据库中的表都存在数据库对应的HDFS目录下。HDFS上属于Inceptor对象的目录和文件管理由Inceptor执行。

2.1.2. 表

Inceptor和其他数据库类似，将表（table）作为最主要的存放数据的对象。逻辑上，Inceptor的表用行（row）和列（column）来组织存储在其中的数据。

2.1.2.1. 表的分类

Inceptor中的表可以按以下不同维度划分：

1. 按 Inceptor的所有权 分类可分为：外部表（或简称为外表）和托管表。
2. 按 表的存储格式 分类可分为：TEXT表、ORC表、CSV表和Holodesk表。
3. 按表 是否分区 可分为：分区表和非分区表。
4. 按表 是否分桶 可分为：分桶表和非分桶表。

同一维度下一张表只能有一个属性，而不同维度不互相干涉（有一些特例，我们会另外说明），例如一张表可以同时有下面属性：

（外部表， TEXT表， 分区表， 分桶表）

本节对这些维度下不同类型的表进行简要介绍。

2.1.2.1.1. 按Inceptor的所有权分类

表按Inceptor对它的所有权可以分为 外表(external table) 和 托管表（managed table）。托管表常常又称 内表。

- 托管表

CREATE TABLE 默认创建托管表。Inceptor对托管表有所有权——用 **DROP** 删除托管表时，Inceptor会将表中数据全部删除。

- 外表

外表用 **CREATE EXTERNAL TABLE** 创建，外表中的数据可以保存在HDFS的一个指定路径上（和 **LOCATION <hdfs_path>** 合用）。Inceptor对外表没有所有权。用 **DROP** 删除外部表时，Inceptor删除表在metastore中的元数据而不删除表中数据，也就是说 **DROP** 仅仅解除Inceptor对外表操作的权利。

2.1.2.1.2. 按表的存储格式分类

按存储格式可以将表分为TEXT表、ORC表、CSV表和Holodesk表。

- TEXT表

TEXT表即文本格式的表，是Inceptor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。针对不同的使用场景，用户可以将其中的数据放入ORC表或Holodesk表中。

Inceptor提供两种方式将文本文件中的数据导入TEXT表中：

- a. 建外部TEXT表，让该表指向HDFS上的一个目录，Inceptor会将目录下文件中的数据都导入该表。我们推荐使用这种方式导数据。
- b. 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，我们 不推荐 使用这种方式导数据。

关于TEXT表的更多内容和具体操作，请参考[TEXT表](#)。

- CSV表

CSV表的数据来源为CSV格式（Comma-Separated Values）的文件。文件以纯文本形式存储表格数据（数字和文本），CSV文件由任意数目的记录组成，记录间以某种换行符分隔；每条记录由字段组成，字段间

的分隔符是其它字符或字符串，最常见的是逗号或制表符。通常，所有记录都有完全相同的字段序列。和TEXT表相似，CSV表常用于向Inceptor中导入原始数据，然后针对不同场景，用户可以将其中的数据放入ORC表或Holodesk表中。

关于CSV表的更多内容和具体操作，请参考[CSV表](#)。

- **ORC表**

ORC表即ORC格式的表。在Inceptor中，ORC表还分为ORC事务表和非事务表。

- a. ORC事务表支持事务处理和更多增删改语法（**INSERT VALUES/UPDATE/DELETE/MERGE**），所以如果您需要对表进行事务处理，应该选择使用ORC事务表。
- b. ORC非事务表则主要用来做统计分析。

ORC表以及事务处理相关细节请参考[ORC表和事务控制语言（TCL）](#)

- **Holodesk表**

Holodesk表存储在内存或者SSD中（可以根据您的需要设置），同时，星环科技为其提供了一系列优化工具，使得在Holodesk表上进行大批量复杂查询能达到极高的性能。所以，如果您的数据量特别大，查询非常复杂，您应该选择使用Holodesk表。

关于Holodesk表的更多内容和具体操作，请参考“[Holodesk表](#)”章节。

2.1.2.1.3. 按表是否分区分类

按表是否分区可以将表为两类：分区表和非分区表。

- **分区表**

如果在建表时使用了 **PARTITIONED BY**，表即为分区表。分区表下的数据按分区键的值（或值的范围）放在HDFS下的不同目录中，可以有效减少查询时扫描的数据量，提升查询效率。

关于分区表的更多内容和具体操作，请参考“[分区表](#)”章节。

- **非分区表**

非分区表即除分区表之外的表。



注意，Holodesk表不能分区。

2.1.2.1.4. 按表是否分桶分类

按表是否分桶可以将表分为两类：分桶表和非分桶表。

- **分桶表**

如果在建表时使用了 **CLUSTERED BY … INTO … BUCKETS**，表即为分桶表。分桶表下的数据按分桶键的哈希值放在HDFS下的不同目录中，可以有效减少查询时扫描的数据量，提升查询效率。

关于分桶表的更多内容和具体操作，请参考“[分桶表](#)”章节。

- **非分桶表**

非分桶表即除分桶表之外的表。

2.1.3. 视图

视图由 `CREATE VIEW ... AS SELECT...` 创建，将一个查询建成一张逻辑上的表——当我们创建一个视图以后，它就可以和 `CREATE TABLE ... AS SELECT` 所建的表一样用来查询。但是和表不同的是，Inceptor中的视图是 非实物化(unmaterialized) 的，也就是说视图中没有实际的数据，每次对视图进行查询时，建视图所用的查询语句会会被再次执行一次。所以，我们也可以将视图理解为 查询的封装。视图不能和表重名。

视图的作用很多，比如简化查询、对表中内容的权限控制等。

例 1. 简化查询

下面的语句查询所有在trans_info中有交易记录的账户持有人名字：

```
SELECT DISTINCT name FROM (SELECT name FROM user_info JOIN trans_info ON user_info.acc_num =
trans_info.acc_num);
+-----+
| name |
+-----+
| 管** |
| 李** |
| 邱* |
| 潘** |
| 华* |
| 魏** |
| 祝** |
| 马** |
+-----+
```

我们可以把这个查询中的子查询封装进一个视图中，从而简化这个查询。

```
CREATE VIEW user_join_trans AS SELECT name FROM user_info JOIN trans_info ON user_info.acc_num =
= trans_info.acc_num;
SELECT DISTINCT name FROM user_join_trans;
+-----+
| name |
+-----+
| 管** |
| 李** |
| 邱* |
| 潘** |
| 华* |
| 魏** |
| 祝** |
| 马** |
+-----+
```

例 2. 权限控制

user_info表中包含涉及账户安全的信息，如身份证号码、密码等。现在我们想要做到让一部分Inceptor用户（例如Inceptor用户Alice和Bob）只能读到user_info表中不涉及账户安全的信息，我们可以做下面的操作：

1. 确保Alice和Bob对user_info表没有读权限。
2. 建视图user_info_secure，包含user_info中不涉及账户安全的列：

```
CREATE VIEW user_info_secure AS SELECT name, acc_num, reg_date, acc_level FROM user_info;
```

3. 赋予用户Alice和Bob对视图user_info_secure的读权限：

```
GRANT SELECT ON user_info_secure TO USER alice, USER bob;
```

2.1.4. 函数

您可以用函数对Inceptor中的数据进行多种计算。Inceptor拥有大量的内置函数，这些函数以及它们的用法在[Inceptor函数和运算符手册](#)中列出。

另外，您可以在Inceptor中创建自定义函数。自定义函数分为 **临时函数** 和 **永久函数**，临时函数在重启Inceptor前在各个session间都是有效的，重启Inceptor后该函数将不再存在，如需使用需要重新创建。永久函数则在Inceptor重启后依然可以使用。临时函数和永久函数的创建和删除可以通过Inceptor SQL进行，细节请参考[CREATE/DROP FUNCTION](#)部分介绍。



版本信息

Inceptor从TDH4.5.2开始支持永久函数。

2.2. Inceptor中的数据类型

Inceptor对简单数据类别(primitive data type)和复杂数据类别(complex data type)都支持。

2.2.1. 简单数据类别

数据类型	描述	实例
TINYINT	1字节(8位)有符号整数，从-128到127	1
SMALLINT	2字节(16位)有符号整数，从-32768到32767	1
INT	4字节(32位)有符号整数，从-2147483648到2147483647	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	1.0
DOUBLE	8字节双精度浮点数	1.0

数据类型	描述	实例
DECIMAL(m,n) , DECIMAL (默认为 DECIMAL(10,2))	不可变的，任意精度的，有符号的十进制数	1.012, 1e+44
BOOLEAN	true/false	TRUE/FALSE
STRING	字符串	'a', "a"
VARCHAR	可变长度的字符	'a', 'a'
DATE	日期。格式为 'yyyy-MM-dd' 或者 yyyy-MM-dd HH:mm:ss。带有 HH:mm:ss 的DATE类型不属于标准DATE类型，我们不推荐使用 yyyy-MM- dd HH:mm:ss 来表示DATE。同时，ORC表 不支持 yyyy-MM-dd HH:mm:ss 格式的DATE，会将 HH:mm:ss 部分去掉。对于 yyyy-MM-dd HH:mm:ss 形式的时间我们建议使用TIMESTAMP	'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'
INTERVAL SECOND/HOUR/ MINUTE/DAY/MO NTH/YEAR	用于存储一段以年，月或日为单位的时间	INTERVAL '10' day
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'。TIME类型只能在DB2方言模式下使用。	'10:07:05'

说明

- **DECIMAL(m,n)**

- **DECIMAL(m,n)** 类型让用户可以自定义数值的精度。**DECIMAL(m,n)** 有两个参数： **m** 为类型的精度 (precision)：它规定了该 **DECIMAL** 类型总共可以有多少位（包括小数点前和小数点后的位）。Inceptor中精度最大不能超过38。 **n** 为类型的标度 (scale)：它规定了该 **DECIMAL** 类型小数点后面的位数。**n** 不能大于 **m**。如果数据无法按照指定的精度和标度表示，将被认为是 **NULL** 值。**DECIMAL** 可以用于所有常见运算（如 +, -, *, /）和相应的函数（如 ***FLOOR**, **CEIL**, **ROUND**, 等等）。用户也可以用数值型的常见方法来在 **DECIMAL** 和别的数值型之间转换。下面举几个例子：

```
--不指定精度和标度默认为DECIMAL(10,2)
SELECT CAST(12345678.3456 AS DECIMAL) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 12345678.35 |
+-----+
--由于123456789.3456无法被表示为DECIMAL(10,2)，被判定为NULL
SELECT CAST(123456789.3456 AS DECIMAL) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| NULL |
+-----+
--增加精度
SELECT CAST(123456789.3456 AS DECIMAL(11,2)) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 123456789.35 |
+-----+
```

- 数值字面值（numeric literal）向 **DECIMAL** 的转换

从TDH4.6开始，带小数点的数值字面值会自动被识别为 **DECIMAL** 类型（在TDH4.6之前，带小数点的数值字面值会自动被识别为 **DOUBLE** 类型）。例如：

```
CREATE TABLE test AS SELECT 1.1 FROM system.dual;

DESC test;
+-----+-----+-----+-----+-----+-----+
| col_name | data_type | default_value | not_null | unique | comment |
+-----+-----+-----+-----+-----+-----+
| _c0      | decimal(2,1) | NULL          | No       | No     |           |
+-----+-----+-----+-----+-----+-----+
```

如果将数值常量向类型为 **DOUBLE** 的列插入，该数值会被隐式转换为 **DOUBLE** 类型：

```
DESC double_conversion;
+-----+-----+-----+-----+-----+-----+
| col_name | data_type | default_value | not_null | unique | comment |
+-----+-----+-----+-----+-----+-----+
| id       | int        | NULL          | No       | No     |           |
| double_column | double    | NULL          | No       | No     |           |
+-----+-----+-----+-----+-----+-----+

INSERT INTO double_conversion VALUES(1, 1.0);

SELECT * FROM double_conversion;
+-----+
| id   | double_column |
+-----+
| 1    | 1.0           |
+-----+
```

然而，这个隐式转换在使用 **NAMED_STRUCT** 时 不会发生：

```

DESC demo;
+-----+-----+-----+-----+-----+
| col_name | data_type | default_value | not_null | unique | comment
+-----+-----+-----+-----+-----+
| id | int | NULL | No | No |
| info | struct<name:string,height:double> | NULL | No | No |
+-----+-----+-----+-----+-----+
INSERT INTO TABLE demo VALUES(1, NAMED_STRUCT('name', 'zhang san', 'height', 1.78));
Error: COMPILE FAILED: Semantic error: [Error 10044] Line 1:18 Cannot insert into target
table because column number/types are different 'demo': Cannot convert column 1 from:
struct<name:string,height:decimal(3,2)>
to
struct<name:string,height:double> (state=42000,code=10044) ①

```

① Inceptor报错：无法将 **DECIMAL** 类型转换为 **DOUBLE** 类型。

所以，我们建议在向 ***DOUBLE** 类型的目标（包括表的列、 **STRUCT** 内的字段等）插入数值字面量时，使用 **CAST** 将数值字面量显式转换为 **DOUBLE**：

```

INSERT INTO demo VALUES(1, NAMED_STRUCT('name', 'zhang san', 'height', CAST(1.78 AS
DOUBLE)));
SELECT * FROM demo;
+-----+-----+
| id | info |
+-----+-----+
| 1 | {"name":"zhang san","height":1.78} |
+-----+-----+

```

- **VARCHAR**

VARCHAR 生成时会带有一个 长度指定数(length specifier) (1和65355之间)，用来定义字符串中的最大字符数。如果一个向 **VARCHAR** 转换的 **STRING** 型中的字符个数超过了长度指定数，那么这个 **STRING** 会被自动缩短。和 **STRING** 类型一样， **VARCHAR** 末尾的空格数是有意义的，会影响比较结果。

- **DATE**

- 可被识别为 **DATE** 的字符串

三种形式的字符串可以在 **INSERT** 进类型为 **DATE** 的目标字段时被正确识别：**yyyy-MM-dd**, **yyyy-MM-dd HH:mm:ss**, **yyyyMMdd**

```

--列a为DATE类型
DESC ckdttest;
a          date

--插入三种形式的字符串
INSERT INTO ckdttest VALUES ('2015-10-31');
INSERT INTO ckdttest VALUES ('2015-01-01 00:00:00');
INSERT INTO ckdttest VALUES ('20160702');

--三个字符串都被识别:
SELECT * FROM ckdttest;
2015-01-01 ①
2016-07-02
2015-10-31

```

① 注意 '2015-01-01 00:00:00' 被 **INSERT** 后从表中读取时时间部分没有了，这是因为 **ckdttest** 为一张ORC表，ORC表中 **yyyy-MM-dd HH:mm:ss** 的 **HH:mm:ss** 部分会丢失。

这种形式的识别不是隐式转换，而是在目标字段为 **DATE** 类型时发生的一次*CAST*。事实上，没有任何类型（除 **DATE** 本身）可以隐式转换为 **DATE**。下面这个例子可以说明：

```
--进行INTERVAL运算时，没有隐式转换发生，INTERVAL无法和STRING类型的数据直接相加
SELECT '2015-01-01' + INTERVAL '1' YEAR FROM ckttest;
[Hive Error]: COMPILE FAILED: Semantic error: [Error 10014] Line 1:7 Wrong arguments ''1'':
No matching method for {0} with {1}No matching method for class
org.apache.hadoop.hive.udf.generic.GenericUDFOPDTIPlus with (string, interval_year_month)

--所以要先将字符串CAST为DATE
SELECT CAST('2015-01-01' AS DATE) + INTERVAL '1' YEAR FROM ckttest LIMIT 1;
2016-01-01
```

- **DATE** 类型的显式转换（CAST）

DATE 类型只能和 **DATE**, **TIMESTAMP** 和 **STRING** 相互 **CAST**。具体如下表：

有效的 CAST	转换结果
CAST(<date> AS DATE)	相同 DATE 值
CAST(<timestamp> AS DATE)	根据本地时区从 <timestamp> 得出年/月/日，将其作为 DATE 值返回
CAST(<string> AS DATE)	如果字符串的形式是 'yyyy-MM-dd'，将对应年/月/日作为 DATE 值返回。如果字符串不具有这种形式，返回 NULL 。
CAST(<date> AS TIMESTAMP)	根据本地时区生成并返回对应 DATE 值/月/日零点的 TIMESTAMP 值。
CAST(<date> AS STRING)	根据 DATE 的年/月/日值生成并返回 'yyyy-MM-dd' 格式的字符串。

- 将不同形式的字符串转化为 **DATE**

在实际的数据中，**DATE** 字符串的类型会有非常多不同的形式，例如 'yyyyMMdd', 'yyyy-MM-dd', 'yyyy/MM/dd' 等等。要将这些 **STRING** 转换为 **DATE** 需要经过两步：

- 从Inceptor提供的众多日期函数中选择一个合适的函数（例如 **TDH_TODATE** 函数），将原始数据转换为 **yyyy-MM-dd** 格式的字符串。
- 将上一步中的字符串 **CAST** 为 **DATE** 类型。

- **TIMESTAMP**

支持传统的 **UNIX TIMESTAMP**，提供达到纳秒级别精度的选择。**TIMESTAMP** 是以和UNIX epoch（协调世界时1970年1月1日00:00:00）之间的秒数差定义的。可以向 **TIMESTAMP** 隐性转换的数类型有：

- 整数数值型：当作 **UNIX TIMESTAMP** 秒数处理。
- 浮点数数值型：当作有小数点后精度的 **UNIX TIMESTAMP** 处理。
- 字符串：必须具有 "yyyy-MM-dd HH:mm:ss[.ffffffff]" 格式。小数点后秒数精度可选。

Inceptor提供了在时区间转换的函数：**TO_UTC_TIMESTAMP**, **FROM_UTC_TIMESTAMP**。所有现成的日期时间函数(month, day, year, hour等)都可以用于 **TIMESTAMP**。在文本文件

中, **TIMESTAMP** 的格式必须是 `yyyy-MM-dd HH:mm:ss[.fffffffff]` 格式。如果是以其他格式表示的, 先用合适的数据类型 (`INT`, `FLOAT`, `STRING` 等) 声明然后再用UDF转换成 **TIMESTAMP**。

- **INTERVAL**

在Inceptor中, **INTERVAL** 类型可以分别表示以年、月、日为单位的一段时间。**INTERVAL** 类型的数据只能和 **DATE** 类型或者 **TIMESTAMP** 相加, 得到一个新的 **DATE** 或者 **TIMESTAMP**。

```
INTERVAL 'n' SECOND -- 长度为n秒的一段时间
INTERVAL 'n' MINUTE -- 长度为n分钟的一段时间
INTERVAL 'n' HOUR -- 长度为n小时的一段时间
INTERVAL 'n' DAY -- 长度为n天的一段时间
INTERVAL 'n' MONTH -- 长度为n月的一段时间
INTERVAL 'n' YEAR -- 长度为n年的一段时间
```

例如:

```
--下面三条计算都得到1小时以后的TIMESTAMP
CAST('1998-08-04' AS TIMESTAMP) + INTERVAL '3600' SECOND
CAST('1998-08-04' AS TIMESTAMP) + INTERVAL '60' MINUTE
CAST('1998-08-04' AS TIMESTAMP) + INTERVAL '1' HOUR

--下面三条计算都得到1年以后的DATE
CAST('1998-08-04' AS DATE) + INTERVAL '365' DAY
CAST('1998-08-04' AS DATE) + INTERVAL '1' YEAR
CAST('1998-08-04' AS DATE) + INTERVAL '12' MONTH
```

- **TIME**

TIME 类型必须在DB2方言模式下使用。在使用不同的命令行时打开DB2方言开关的方法请参考[Inceptor交互方法](#)。下面我们以beeline命令行为例。

```
!set plsqlClientDialect db2
set plsql.server.dialect=db2;

CREATE TABLE time_basic_orc(t TIME, d DATE, ts TIMESTAMP) CLUSTERED BY (d) INTO 2 BUCKETS STORED AS ORC TBLPROPERTIES("transactional"="true");

INSERT INTO TABLE time_basic_orc VALUES ('2015-01-11 10:07:05', '2015-01-11 10:07:05', '2015-01-11 10:07:05');

SELECT * FROM time_basic_orc;
+-----+-----+-----+
| t    | d    | ts   |
+-----+-----+-----+
| 10:07:05 | 2015-01-11 | 2015-01-11 10:07:05.0 |
+-----+-----+-----+

SELECT TIME '2015-12-12 11:22:33' FROM time_basic_orc FETCH FIRST 1 ROWS ONLY;
+-----+
| _c0  |
+-----+
| 11:22:33 |
+-----+
```

简单数据类别之间的隐性转换(conversion)

Inceptor原生数据之间的隐性转换遵从下面这张表。左侧一栏表示源类型, 顶部一栏表示目标类型, TRUE和FALSE表示这种隐式转换是否允许。如果需要强制实现该表中禁止的转换必须使用显性转换(`cast`)。这里提供了Inceptor原生简单数据类型之间的隐性转化关系, 没有包含方言支持的类型。

	void	boolean	tinyint	smallint	int	bigint	float	double	decimal	string	varchar	timestamp	date	binary
void to	true	true	true	true	true	true	true	true	true	true	true	true	true	true
tinyint to	false	false	true	true	true	true	true	true	true	true	false	false	false	false
smallint to	false	false	false	true	true	true	true	true	true	true	false	false	false	false
int to	false	false	false	false	true	true	true	true	true	true	false	false	false	false
bigint to	false	false	false	false	false	true	true	true	true	true	false	false	false	false
float to	false	false	false	false	false	false	true	true	true	true	false	false	false	false
double to	false	false	false	false	false	false	false	true	true	true	false	false	false	false
decimal to	false	false	false	false	false	false	false	false	true	true	false	false	false	false
string to	false	false	false	false	false	false	false	true	true	true	false	false	false	false
varchar to	false	false	false	false	false	false	false	true	true	true	false	false	false	false
boolean to	false	true	false	false	false	false	false	false	false	false	false	false	false	false
timestamp to	false	false	false	false	false	false	false	false	false	true	true	false	false	false
date to	false	false	false	false	false	false	false	false	false	true	false	true	false	false
binary to	false	false	false	false	false	false	false	false	false	false	false	false	false	true

2.2.2. 复杂数据类别

数据类型	描述	示例
ARRAY	一组有序字段，所有字段的数据类型必须相同	ARRAY(1,2)
MAP	一组无序的键/值对。键的类型必须是 原生数据类型，值的类型可以是 原生或复杂 数据类型。同一个MAP的键的类型必须相同，值的类型也必须相同。	MAP('a', 1, 'b', 2)
STRUCT	一组命名的字段，字段的数据类型可以不同	STRUCT('a', 1, 1.0)

2.2.2.1. ARRAY 示例

创建数据表array_test，将person_id参数定义为 **ARRAY<INT>** 数据类型（即一组由 INT 数据组成的数据列），然后将已存在的文本text.txt导入至array_test中。操作如下：

```

33 create table array_test(name string, person_id array<INT>)
34 ROW FORMAT DELIMITED
35 FIELDS TERMINATED BY ','
36 COLLECTION ITEMS TERMINATED BY ':'
37 ;
38
39 LOAD DATA LOCAL INPATH '/tmp/test.txt' INTO TABLE array_test ;
40
41 $ cat /tmp/test.txt
42 034,1:2:3:4
43 035,5:6
44 036,7:8:9:10

```

查看array_test全表，并检索其中一列的信息。结果如下：

```

0: jdbc:hive2://172.16.1.144:10000/default> select * from array_test;
+-----+-----+
| name | person_id |
+-----+-----+
| 034  | [1,2,3,4]  |
| 035  | [5,6]       |
| 036  | [7,8,9,10] |
+-----+-----+
3 rows selected (2.906 seconds)
0: jdbc:hive2://172.16.1.144:10000/default> select person_id[1] from array_test;

+-----+
| _c0 |
+-----+
| 2   |
| 6   |
| 8   |
+-----+

```

2.2.2.2. MAP 示例

创建数据表map_test，将perf参数定义为 `MAP<string,int>` 数据类型(键为 STRING 类型，值为 INT 类型)，然后将已存在的文本test_map.txt导入至map_test中。操作如下：

```

48 create table map_test(id string, perf map<string, int>)
49 ROW FORMAT DELIMITED
50 FIELDS TERMINATED BY '|'
51 COLLECTION ITEMS TERMINATED BY ','
52 MAP KEYS TERMINATED BY ':';
53
54 LOAD DATA LOCAL INPATH '/tmp/test_map.txt' INTO TABLE map_test ;
55
56 $ cat /tmp/test_map.txt
57 1|job:80,team:60,person:70
58 2|job:60,team:80
59 3|job:90,team:70,person:100

```

查看map_test全表，并检索perf中“job”和“Person”的信息。结果如下：

```

0: jdbc:hive2://172.16.1.144:10000/default> select * from map_test;
+-----+
| id |          perf           |
+-----+
| 1  | {"job":80,"team":60,"person":70} |
| 2  | {"job":60,"team":80}             |
| 3  | {"job":90,"team":70,"person":100}|
+-----+
3 rows selected (2.855 seconds)
0: jdbc:hive2://172.16.1.144:10000/default> select perf['job'] from map_test;
+-----+
| _c0 |
+-----+
| 80   |
| 60   |
| 90   |
+-----+
3 rows selected (2.533 seconds)
0: jdbc:hive2://172.16.1.144:10000/default> select perf['person'] from map_test;
+-----+
| _c0 |
+-----+
| 70   |
| NULL |
| 100  |
+-----+
3 rows selected (1.729 seconds)

```

2.2.2.3. STRUCT 示例

创建数据表person, 将info定义为 `STRUCT<name:STRING,age:INT>` 数据类型（由name和age构成的字段，其中name为 `STRING` 类型， age为 `INT` 类型），然后将已存在的文本person.txt导入至person表中。操作如下：

```

1 -----
2 create table person(id INT, info struct<name:STRING, age:INT>)
3 ROW FORMAT DELIMITED
4 FIELDS TERMINATED BY ','
5 COLLECTION ITEMS TERMINATED BY ':'
6 ;
7 -----
8 LOAD DATA LOCAL INPATH "/tmp/person.txt" INTO TABLE person;
9
10 -----
11 cat /tmp/person.txt
12
13 1,zhou:30
14 2,yan:30
15 3,chen:20
16 4,li:80
17 -----
18 -----

```

查看person全表，并检索info中“name”和“age”的信息。

```

0: jdbc:hive2://172.16.1.144:10000/default> select * from person;
+-----+-----+
| id   |      info      |
+-----+-----+
| 1    | {"name":"zhou","age":30} |
| 2    | {"name":"yan","age":30}  |
| 3    | {"name":"chen","age":20} |
| 4    | {"name":"li","age":80}   |
+-----+
4 rows selected (13.675 seconds)
0: jdbc:hive2://172.16.1.144:10000/default> select info.name from person;
+-----+
| name |
+-----+
| zhou |
| yan  |
| chen |
| li   |
+-----+
4 rows selected (4.336 seconds)
0: jdbc:hive2://172.16.1.144:10000/default> select info.age from person;
+-----+
| age  |
+-----+
| 30   |
| 30   |
| 20   |
| 80   |
+-----+

```

2.3. Inceptor中的关键字

2.3.1. 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：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，索引，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：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，索引，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](#)。

2.3.2. 非保留关键词的受限范围

非保留关键词主要不能用于（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;`

2.3.3. 关键词列表

下面表格中展现了Inceptor在三大方言下的所有保留关键词和非保留关键词列表。

2.3.3.1. 保留关键词

Oracle	DB2	Teradata
AND	ACTION	ACTION
ANTISEMI	ANCHOR	ANCHOR
APPLICATION	AND	AND
APPLICATIONS	ANTISEMI	ANTISEMI
APP.getProperties	APPLICATION	APPLICATION
AS	APPLICATIONS	APPLICATIONS
ASC	APP.getProperties	APP.getProperties
AUTONOMOUS_TRANSACTION	AS	AS
BATCHINSERT	ASC	ASC

Oracle	DB2	Teradata
BATCHUPDATE	ASCII	ASCII
BATCHVALUES	ASSOCIATE	ASSOCIATE
BEFORE	AUTONOMOUS	AUTONOMOUS
BEGIN	BATCHINSERT	BATCHINSERT
BFILE	BATCHUPDATE	BATCHUPDATE
BLACKLIST	BATCHVALUES	BATCHVALUES
BLOB	BEFORE	BEFORE
BREAK	BEGIN	BEGIN
BULK	BFILE	BFILE
BULK_EXCEPTIONS	BLACKLIST	BLACKLIST
BULK_ROWCOUNT	BLOB	BLOB
CALL	BREAK	BREAK
CASCADE	BULK	BULK
CASE	CALL	CALL
CAST	CALLED	CALLED
CHAR	CALLER	CALLER
CLOB	CASCADE	CASCADE
CLOSE	CASE	CASE
COLLECT	CAST	CASESPECIFIC
COLLECTION	CHAR	CAST
COLUMN	CLOB	CHAR
COLUMNS	CLOSE	CHARACTER
COMMIT	COLLECT	CLOB
COMMITTED	COLLECTION	CLOSE
COMPACTIONS	COLUMN	COLLECT
CONCATENATE	COLUMNS	COLLECTION
CONF	COMMIT	COLUMN
CONNECT	COMMITTED	COLUMNS
CONSTANT	COMPACTIONS	COMMIT
CONTINUE	CONCATENATE	COMMITTED
CROSS	CONDITION	COMPACTIONS
CUBE	CONF	COMPRESS
CURRENT	CONNECT	CONCATENATE
CURSOR	CONSTANT	CONDITION

Oracle	DB2	Teradata
DATABASE	CONTAINS	CONF
DATABASES	CONTINUE	CONNECT
DBPROPERTIES	CROSS	CONSTANT
DEC	CUBE	CONTAINS
DECIMAL	CURRENT	CONTINUE
DECLARE	CURSOR	CROSS
DEPENDENCY	DATABASE	CUBE
DESC	DATABASES	CURRENT
DESCRIBE	DB2_RETURN_STATUS	CURSOR
DIRECTORIES	DB2_SQL_NESTING_LEVEL	DATABASE
DIRECTORY	DBPROPERTIES	DATABASES
DISTINCT	DEC	DBPROPERTIES
DOCVALUES	DECIMAL	DEC
ELSE	DECLARE	DECIMAL
ELSEIF	DEFINITION	DECLARE
ELSIF	DEPENDENCY	DEFINITION
END	DESC	DEPENDENCY
ERRORS	DESCRIBE	DESC
ES	DIRECTORIES	DESCRIBE
ESCAPED	DIRECTORY	DIRECTORIES
EXCEPT	DISTINCT	DIRECTORY
EXCEPTION	DOCVALUES	DISTINCT
EXCEPTION_INIT	ELSE	DOCVALUES
EXCEPTIONS	ELSEIF	ELSE
EXCHANGE	ELSIF	ELSEIF
EXECUTE	END	ELSIF
EXIT	ERRORS	END
EXPORT	ES	ERRORS
EXTENDED	ESCAPED	ES
EXTRACT	EXCEPT	ESCAPED
FILEFORMAT	EXCEPTION	EXCEPT
FOLLOWING	EXCHANGE	EXCEPTION
FOR	EXECUTE	EXCHANGE
FORALL	EXPORT	EXECUTE

Oracle	DB2	Teradata
FORMAT	EXTENDED	EXPORT
FORMATTED	EXTRACT	EXTENDED
FROM	FILEFORMAT	EXTRACT
FUNCTION	FOLLOWING	FILEFORMAT
FUNCTIONS	FOR	FOLLOWING
GOTO	FORMAT	FOR
GRAPH_PATH	FORMATTED	FORMAT
HAVING	FOUND	FORMATTED
HOLODESK	FROM	FOUND
IDENTIFIED	FUNCTION	FROM
IDXPROPERTIES	FUNCTIONS	FUNCTION
IF	GOTO	FUNCTIONS
IGNORE	GRAPH_PATH	GOTO
IMPORT	HANDLER	GRAPH_PATH
INDEXES	HAVING	HANDLER
INDICES	HOLODESK	HAVING
INPUTFORMAT	IDENTIFIED	HOLODESK
ISOLATION	IDXPROPERTIES	IDENTIFIED
ISOPEN	IF	IDXPROPERTIES
JOBPROPERTIES	IGNORE	IF
JOIN	IMPORT	IGNORE
LAST	INDEXES	IMPORT
LESS	INPUTFORMAT	INDEXES
LINES	ISOLATION	INPUTFORMAT
LINK	JOBPROPERTIES	ISOLATION
LINKS	JOIN	JOBPROPERTIES
LOCATION	LAST	JOIN
LONG	LESS	LAST
LOOP	LINES	LESS
MACRO	LINK	LINES
MAP	LINKS	LINK
MAPJOIN	LOCATION	LINKS
MATCHED	LOCATOR	LOCATION
MAXEXTENTS	LOCATORS	LOCATOR

Oracle	DB2	Teradata
MAXTRANS	LONG	LOCATORS
MAXVALUE	LOOP	LONG
MINEXTENTS	MACRO	LOOP
MONTH	MAP	MACRO
MORE	MAPJOIN	MAP
NATURAL	MATCHED	MAPJOIN
NOMAXVALUE	MAXEXTENTS	MATCHED
NOORDER	MAXTRANS	MAXEXTENTS
NULL	MAXVALUE	MAXTRANS
NULLS	MESSAGE_TEXT	MAXVALUE
NVARCHAR	MINEXTENTS	MESSAGE_TEXT
ON	MINUTES	MINEXTENTS
ONLY	MODIFIES	MINUTES
OP_CONCAT	MONTH	MODIFIES
OPEN	MONTHS	MONTH
OPTION	MORE	MONTHS
OR	NATURAL	MORE
ORC	NOMAXVALUE	NATURAL
ORC_TRANSACTION	NOORDER	NOMAXVALUE
ORDER	NULL	NOORDER
OUTPUTFORMAT	NULLS	NULL
OVER	NVARCHAR	NULLS
OVERWRITE	OFFSET	NVARCHAR
PACKAGES	ON	ON
PARTIALSCAN	ONLY	ONLY
PARTITION	OP_CONCAT	OP_CONCAT
PARTITIONED	OPEN	OPEN
PARTITIONS	OPTION	OPTION
PCTFREE	OR	OR
PCTUSED	ORC	ORC
PERMISSION	ORC_TRANSACTION	ORC_TRANSACTION
PLSQL	ORDER	ORDER
PRECEDING	OUTPUTFORMAT	OUTPUTFORMAT
PRESERVE	OVER	OVER

Oracle	DB2	Teradata
PRIOR	OVERWRITE	OVERWRITE
PROTECTION	PARTIALSCAN	PARTIALSCAN
READONLY	PARTITION	PARTITION
RECORD	PARTITIONED	PARTITIONED
RECORDREADER	PARTITIONS	PARTITIONS
RECORDWRITER	PCTFREE	PCTFREE
REDUCE	PCTUSED	PCTUSED
REJECT	PERMISSION	PERMISSION
REPLICATION	PLSQL	PLSQL
RESTRICT	PRECEDING	PRECEDING
RETURN	PRESERVE	PRESERVE
ROLES	PRIOR	PRIOR
ROLLBACK	PROTECTION	PROTECTION
ROLLUP	readonly	QUALIFY
SCHEMAS	RECORDREADER	readonly
SECOND	RECORDWRITER	RECORDREADER
SELECT	REDUCE	RECORDWRITER
SEQUENCES	REJECT	REDUCE
SERDEPROPERTIES	REPLICATION	REJECT
SERIALIZABLE	RESIGNAL	REPLICATION
SHOW_DATABASE	RESTRICT	RESIGNAL
SORT	RESULT	RESTRICT
SORTED	RESULT_SET_LOCATOR	RESULT
START	RETURN	RESULT_SET_LOCATOR
STORAGE	RETURNS	RETURN
STORED	RETURN_STATUS	RETURNS
STREAMWINDOW	ROLES	ROLES
SUBSTRING	ROLLBACK	ROLLBACK
SYSTIME	ROLLUP	ROLLUP
SYSTIMESTAMP	SCHEMAS	SCHEMAS
TABLES	SECOND	SECOND
TABLESAMPLE	SECONDS	SECONDS
TABLESPACE	SELECT	SELECT
TBLPROPERTIES	SEQUENCES	SEQUENCES

Oracle	DB2	Teradata
TEMPORARY	SERDEPROPERTIES	SERDEPROPERTIES
THAN	SERIALIZABLE	SERIALIZABLE
THEN	SET	SHOW_DATABASE
TIMESTAMP	SETS	SORT
TRANSACTION	SHOW_DATABASE	SORTED
TRANSACTIONS	SORT	SPECIFIC
TRANSFORM	SORTED	SQLEXCEPTION
UNBOUNDED	SPECIFIC	START
UNION	SQLEXCEPTION	STORAGE
UNIONTYPE	START	STORED
UNIQUE	STORAGE	STREAMWINDOW
UNIQUEJOIN	STORED	SUBSTRING
UNLIMITED	STREAMWINDOW	SYSDATE
USE_BULKLOAD	SUBSTRING	SYSTIME
UTC_TMESSTAMP	SYSDATE	SYSTIMESTAMP
VALUES	SYSTIME	TABLES
VARCHAR	SYSTIMESTAMP	TABLESAMPLE
VARCHAR2	TABLES	TABLESPACE
VARYING	TABLESAMPLE	TBLPROPERTIES
WHEN	TABLESPACE	TEMPORARY
WHERE	TBLPROPERTIES	THAN
WIDCARD	TEMPORARY	THEN
WINDOW	THAN	TIMESTAMP
WORK	THEN	TRANSACTION
	TIMESTAMP	TRANSACTIONS
	TRANSACTION	TRANSFORM
	TRANSACTIONS	UNBOUNDED
	TRANSFORM	UNION
	UNBOUNDED	UNIONTYPE
	UNION	UNIQUE
	UNIONTYPE	UNIQUEJOIN
	UNIQUE	UNLIMITED
	UNIQUEJOIN	UPPERCASE
	UNLIMITED	USE_BULKLOAD

Oracle	DB2	Teradata
	UNSET	UTC_TMESSSTAMP
	USE_BULKLOAD	VALUES
	UTC_TMESSSTAMP	VARCHAR
	VALUES	VARCHAR2
	VARCHAR	VARYING
	VARCHAR2	WHEN
	VARYING	WHERE
	WHEN	WIDCARD
	WHERE	WINDOW
	WIDCARD	WORK
	WINDOW	YES
	WORK	
	YES	

2.3.3.2. 非保留关键词

Oracle	DB2	Teradata
ADD	ADD	ADD
ADHOC	ADHOC	ADHOC
ADMIN	ADMIN	ADMIN
AFTER	AFTER	AFTER
ALL	ALL	ALL
ALTER	ALLOCATE	ALLOCATE
ANALYZE	ALTER	ALTER
APP	ANALYZE	ANALYZE
APPS	APP	APP
ARCHIVE	APPS	APPS
ARRAY	ARCHIVE	ARCHIVE
AT	ARRAY	ARRAY
ATTACH	AT	AT
BETWEEN	ATOMIC	ATOMIC
BIGINT	ATTACH	ATTACH
BINARY	BETWEEN	BETWEEN
BODY	BIGINT	BIGINT

Oracle	DB2	Teradata
BOOLEAN	BINARY	BINARY
BOTH	BOOLEAN	BOOLEAN
BUCKET	BOTH	BOTH
BUCKETS	BUCKET	BUCKET
BY	BUCKETS	BUCKETS
BYTE	BY	BY
CACHE	BYTE	BYTE
CHANGE	CACHE	BYTEINT
CLUSTER	CCSID	CACHE
CLUSTERED	CHANGE	CCSID
CLUSTERSTATUS	CLIENT	CHANGE
COMBINE	CLUSTER	CLIENT
COMMENT	CLUSTERED	CLUSTER
COMPACT	CLUSTERSTATUS	CLUSTERED
COMPUTE	COMBINE	CLUSTERSTATUS
CREATE	COMMENT	COMBINE
CSVFILE	COMPACT	COMMENT
CURRVAL	COMPUTE	COMPACT
CYCLE	CREATE	COMPUTE
DATA	CSVFILE	CREATE
DATE	CURRVAL	CS
DATETIME	CYCLE	CSVFILE
DAY	DATA	CURRVAL
DEFAULT	DATE	CYCLE
DEFERRED	DATETIME	DATA
DEFINED	DAY	DATE
DELETE	DAYS	DATETIME
DELIMITED	DB2_TOKEN_STRING	DAY
DETACH	DEFAULT	DAYS
DISABLE	DEFERRED	DEFAULT
DISTRIBUTE	DEFINED	DEFERRED
DOUBLE	DELETE	DEFINED
DROP	DELIMITED	DELETE
ENABLE	DETERMINISTIC	DELIMITED

Oracle	DB2	Teradata
EXCLUSIVE	DETACH	DETERMINISTIC
EXISTS	DIAGNOSTICS	DETACH
EXPLAIN	DISABLE	DIAGNOSTICS
EXTERNAL	DISTRIBUTE	DISABLE
FACL	DO	DISTRIBUTE
FALSE	DOUBLE	DO
FETCH	DROP	DOUBLE
FIELDS	DYNAMIC	DROP
FILE	ENABLE	DYNAMIC
FIRST	EXCLUSIVE	ENABLE
FLOAT	EXISTS	EXCLUSIVE
FOUND	EXIT	EXISTS
FULL	EXPLAIN	EXIT
FULLTEXT	EXTERNAL	EXPLAIN
FWCFILE	FACL	EXTERNAL
GLOBAL	FALSE	FACL
GRANT	FETCH	FALSE
GROUP	FIELDS	FETCH
GROUPING	FILE	FIELDS
HOLD_DDLTIME	FIRST	FILE
HOUR	FLOAT	FIRST
HYPERDRIVE	FULL	FLOAT
IMMEDIATE	FULLTEXT	FULL
IN	FWCFILE	FULLTEXT
INCREMENT	GET	FWCFILE
INDEX	GLOBAL	GET
INFINITE	GRANT	GLOBAL
INITIAL	GROUP	GRANT
INITRANS	GROUPING	GROUP
INNER	HOLD	GROUPING
INOUT	HOLD_DDLTIME	HOLD
INPATH	HOUR	HOLD_DDLTIME
INPUTDRIVER	HOURS	HOUR
INSERT	HYPERDRIVE	HOURS

Oracle	DB2	Teradata
INT	IMMEDIATE	HYPERDRIVE
INTEGER	IN	IMMEDIATE
INTERSECT	INCREMENT	IN
INTERVAL	INDEX	INCREMENT
INTO	INFINITE	INDEX
IS	INHERIT	INFINITE
ITEMS	INITIAL	INHERIT
JAR	INITRANS	INITIAL
KEYS	INNER	INITRANS
LATERAL	INOUT	INNER
LEFT	INPATH	INOUT
LENGTH	INPUT	INPATH
LEVEL	INPUTDRIVER	INPUT
LIKE	INSERT	INPUTDRIVER
LIMIT	INT	INSERT
LIST	INTEGER	INT
LOAD	INTERSECT	INTEGER
LOCAL	INTERVAL	INTERSECT
LOCALPARALLELISM	INTO	INTERVAL
LOCK	IS	INTO
LOCKS	ITEMS	IS
LOG	ITERATE	ITEMS
LOGICAL	JAR	ITERATE
MATERIALIZED	KEYS	JAR
MERGE	LANGUAGE	KEYS
MINUS	LATERAL	LATERAL
MINUTE	LEAVE	LEAVE
MINVALUE	LEFT	LEFT
MSCK	LENGTH	LENGTH
NEXT	LEVEL	LEVEL
NEXTVAL	LIKE	LIKE
NO	LIMIT	LIMIT
NOCACHE	LIST	LIST
NOCOPY	LOAD	LOAD

Oracle	DB2	Teradata
NOCYCLE	LOCAL	LOCAL
NO_DROP	LOCALPARALLELISM	LOCALPARALLELISM
NOMINVALUE	LOCK	LOCK
NOSCAN	LOCKS	LOCKS
NOT	LOG	LOG
NOTFOUND	LOGICAL	LOGICAL
NUM	MATERIALIZED	MATERIALIZED
NUMBER	MERGE	MERGE
NUMERIC	MINUS	MINUS
OF	MINUTE	MINUTE
OFF	MINVALUE	MINVALUE
OFFLINE	MSCK	MSCK
OFFSET	NEW	MULTISET
OUT	NEXT	NAMED
OUTER	NEXTVAL	NEW
OUTPUTDRIVER	NO	NEXT
OWNER	NOCACHE	NEXTVAL
PACKAGE	NOCYCLE	NO
PARQUET	NO_DROP	NOCACHE
PERCENT	NOMINVALUE	NOCYCLE
PERMANENT	NOSCAN	NO_DROP
PLANT	NOT	NOMINVALUE
PLUS	NUM	NOSCAN
PRAGMA	NUMBER	NOT
PRECOMPILE	NUMERIC	NUM
PRETTY	OF	NUMBER
PRINCIPALS	OFF	NUMERIC
PROCEDURE	OFFLINE	OF
PROMPT	OLD	OFF
PUBLIC	OUT	OFFLINE
PURGE	OUTER	OFFSET
QUOTA	OUTPUTDRIVER	OLD
RAISE	OWNER	OUT
RANGE	PARAMETER	OUTER

Oracle	DB2	Teradata
RCFILE	PARQUET	OUTPUTDRIVER
READ	PERCENT	OWNER
READS	PERMANENT	PARAMETER
REBUILD	PLANT	PARQUET
REF	PLUS	PERCENT
REGEXP	PRECOMPILE	PERMANENT
RENAME	PREPARE	PLANT
REPAIR	PRETTY	PLUS
REPLACE	PRINCIPALS	PRECOMPILE
REVERSE	PROCEDURE	PREPARE
REVOKE	PROMPT	PRETTY
RIGHT	PUBLIC	PRIMARY
RLIKE	PURGE	PRINCIPALS
ROLE	QUOTA	PROCEDURE
ROW	RANGE	PROMPT
ROWCOUNT	RCFILE	PUBLIC
ROWS	READ	PURGE
ROWTYPE	READS	QUIT
SAVE	REBUILD	QUOTA
SCHEMA	REGEXP	RANGE
SEGMENT	REGISTERS	RCFILE
SEMI	RENAME	READ
SEPARATED	REPAIR	READS
SEQUENCE	REPEAT	REBUILD
SEQUENCEFILE	REPLACE	REGEXP
SERDE	REVOKE	REGISTERS
SERIALLY_REUSEABLE	RIGHT	RENAME
SERVER	RLIKE	REPAIR
SET	ROLE	REPEAT
SETS	ROW	REPLACE
SHARD	ROW_COUNT	REVOKE
SHARED	ROWS	RIGHT
SHOW	ROWTYPE	RLIKE
SKEwed	SAVEPOINT	ROLE

Oracle	DB2	Teradata
SLIDE	SCHEMA	ROW
SMALLINT	SEGMENT	ROW_COUNT
SPACE	SEMI	ROWS
SPOOL	SEPARATED	ROWTYPE
SSL	SEQUENCE	SAVEPOINT
STATISTICS	SEQUENCEFILE	SCHEMA
STOP	SERDE	SEGMENT
STREAM	SERVER	SEMI
STREAMJOB	SHARD	SEPARATED
STREAMJOBS	SHARED	SEQUENCE
STREAMS	SHOW	SEQUENCEFILE
STREAMTABLE	SIGNAL	SERDE
STRING	SKEWED	SERVER
STRUCT	SLIDE	SET
STRUCT_INDEX	SMALLINT	SETS
SYSDATE	SPACE	SHARD
TABLE	SPECIAL	SHARED
TERMINATED	SPOOL	SHOW
TEXTFILE	SQL	SIGNAL
TIME	SQLSTATE	SKEWED
TINYINT	SQLWARNING	SLIDE
TO	SSL	SMALLINT
TOUCH	STATEMENT	SPACE
TRIGGER	STATISTICS	SPECIAL
TRUE	STOP	SPOOL
TRUNCATE	STREAM	SQL
TYPE	STREAMJOB	SQLSTATE
UNARCHIVE	STREAMJOBS	SQLWARNING
UNDO	STREAMS	SSL
UNLOCK	STREAMTABLE	STATEMENT
UNSET	STRING	STATISTICS
UNSIGNED	STRUCT	STOP
UPDATE	STRUCT_INDEX	STREAM
URI	TABLE	STREAMJOB

Oracle	DB2	Teradata
USE	TERMINATED	STREAMJOBS
USE_INDEX	TEXTFILE	STREAMS
USER	TIME	STREAMTABLE
USING	TINYINT	STRING
UTC	TO	STRUCT
VARRAY	TOUCH	STRUCT_INDEX
VIEW	TRIGGER	TABLE
WHILE	TRUE	TERMINATED
WITH	TRUNCATE	TEXTFILE
WRITE	TYPE	TIME
YEAR	UNARCHIVE	TINYINT
	UNDO	TITLE
	UNICODE	TO
	UNLOCK	TOUCH
	UNSIGNED	TRIGGER
	UNTIL	TRUE
	UPDATE	TRUNCATE
	URI	TYPE
	USE	UC
	USE_INDEX	UNARCHIVE
	USER	UNDO
	USING	UNICODE
	UTC	UNLOCK
	VALUE	UNSET
	VIEW	UNSIGNED
	WHILE	UNTIL
	WITH	UPDATE
	WITHOUT	URI
	WRITE	USE
	YEAR	USE_INDEX
	YEARS	USER
		USING
		UTC
		VALUE

Oracle	DB2	Teradata
		VIEW
		VOLATILE
		WHILE
		WITH
		WITHOUT
		WRITE
		YEAR
		YEARS

2.3.4. 数据类型对应表

本节我们列出Inceptor中的数据类型和Oracle、DB2以及JDBC中的数据类型的对应。

2.3.4.1. Inceptor与Oracle的数据类型对应表

Oracle	Inceptor
CHAR	Char
VARCHAR	Varchar
NCHAR	Char
Varchar2	Varchar2
NVarchar2	Varchar2
Number (p, s)	Number (p, s)
Number	Number
Number (p)	Number (p)
Decimal	Decimal
Bit	Boolean
Boolean	Boolean
SmallInt	Decimal (38, 0)
Integer	Decimal (38, 0)
Long	Binary
Long Raw	Binary
Raw	Binary
Float	N/A, 可用Decimal (p, s)代替
BinaryFloat	Float
Double	N/A, 可用Decimal (p, s)代替

Oracle	Inceptor
BinaryDouble	Double
CLOB	CLOB
NCLOB	CLOB
BLOB	BLOB
BFile	N/A. 可以用Binary代替使用
Date	Date
Timestamp	Timestamp
Timestamp With Timezone	N/A
Timestamp with Local Timezone	N/A
Interval Year To Month	Interval Year To Month
Interval Day To Second	Interval Day To Second
Struct	Struct
Array	Array
RowId	N/A
URowId	N/A

2.3.4.2. Inceptor与DB2的数据类型对应表

DB2	Inceptor
CHAR	Char
NCHAR	Char
VARCHAR	Varchar
NVARCHAR	Varchar
Boolean	Boolean
SmallInt	SmallInt
Integer	Int/Integer
BigInt	BigInt
Numeric (p, s)	Numeric (p, s)
Numeric (p)	Numeric (p)
Numeric	Numeric
Decimal (p, s)	Decimal (p, s)
Decimal (p)	Decimal (p)
Decimal	Decimal
DecFloat	N/A, 可用Decimal (p, s)代替

DB2	Inceptor
Real	Float
Float	Double
Double	Double
CLOB	CLOB
BLOB	BLOB
NCLOB	CLOB
DBCLOB	BLOB
Graphic	String
VarGraphic	String
Date	Date
Timestamp	Timestamp
Time	Time
XML	N/A

注意 DB2的Decimal当出现精度丢失时，会使用truncate方式，比如decimal(5, 2) a; set a = 3.159, a的实际值会是3.15；而Inceptor的Decimal在处理时会使用四舍五入的方式，上面例子中，a会等于3.16。

2.3.4.3. Inceptor与JDBC的数据类型对应表

JDBC	Inceptor
Null	Void
CHAR	Char
VARCHAR	Varchar
NVARCHAR	Varchar
LongVarchar	String
LongNvarchar	String
Numeric	Numeric
Decimal	Decimal
Bit	Boolean
Boolean	Boolean
TinyInt	TinyInt
SmallInt	SmallInt
Integer	Int/Integer
BigInt	BigInt
Real	Float

JDBC	Inceptor
Float	Float
Double	Double
Binary	Binary
VarBinary	Binary
LongVarBinary	Binary
Date	Date
Time	Time
TimeStamp	TimeStamp
IntervalYM	IntervalYM
IntervalDS	IntervalDS
Struct	Struct
Array	Array

2.4. Inceptor交互方法

2.4.1. 连接Inceptor

Inceptor提供两个服务——InceptorServer 1和InceptorServer 2——让远程客户端向Inceptor提交请求并获取结果，但是一个Inceptor服务不能同时使用InceptorServer 1和InceptorServer 2。我们建议用户使用InceptorServer 2，它支持InceptorServer 1所没有的授权机制，以保证您集群中数据的安全性。在安全模式下，InceptorServer 2可以用Kerberos或者LDAP进行认证，认证方式不同的情况下，和Inceptor的连接方式也会有细微不同，但是通过认证并建立连接后的使用是相同的。



- InceptorServer 1和InceptorServer 2的选择在安装Inceptor时就要决定，具体请参考《Transwarp Data Hub 安装手册》或者本手册的[Inceptor的配置](#)。
- 关于Kerberos，LDAP以及更多安全方面的内容，请参考《Transwarp Data Hub安全手册》。

InceptorServer 1和InceptorServer 2各支持多种连接方式：

- **命令行连接：** 用户直接登陆集群中的服务器，如果使用InceptorServer 1，进入Transwarp命令行进行操作；如果使用InceptorServer 2，进入Beeline命令行进行操作。具体操作请参考[命令行交互](#)。
- **应用程序连接：** Inceptor支持标准的JDBC和ODBC接口，用户可以直接编写JDBC和ODBC程序来使用Inceptor。具体操作请参考[Inceptor JDBC手册](#)和[Inceptor ODBC手册](#)。
- **外部工具连接：** Inceptor支持使用标准JDBC和ODBC接口的工具链接，例如Tableau®，SQuirreL SQL等。具体操作请请参考[Inceptor外部工具连接手册](#)。

2.4.2. 交互语言

Inceptor是一个SQL引擎，它支持99%以上的SQL 99标准和SQL 2003扩展。同时，Inceptor还支持Oracle PL/SQL和DB2 SQL PL过程语言。Inceptor中默认的过程语言是Oracle PL/SQL，要切换过程语言方言，需要执行下面指令：

- Oracle方言开关（默认打开）

命令行

```
set plsql.client.dialect=oracle;
set plsql.server.dialect=oracle;
```

Beeline命令行

```
!set plsqlClientDialect oracle
set plsql.server.dialect=oracle;
```

- DB2方言开关

命令行

```
set plsql.client.dialect=db2;
set plsql.server.dialect=db2;
```

Beeline命令行

```
!set plsqlClientDialect db2
set plsql.server.dialect=db2;
```

2.5. 命令行交互

我们将用server_ip/hostname来指代Inceptor Server所在的节点名称或ip。Inceptor Server所在节点可以通过管理界面的Inceptor角色页面查看。

2.5.1. 进入命令行



进入命令行后，您即可开始SQL操作。如果您想要了解更多命令行选项以及命令行中可以执行的非SQL指令，请参考[Transwarp命令行](#)和[Beeline命令行](#)。

2.5.1.1. InceptorServer 1

登陆集群中的任意一台服务器，执行下面指令，进入Transwarp命令行：

```
$ transwarp -t -h <server_ip/hostname>
```

2.5.1.2. InceptorServer 2

InceptorServer 2使用Beeline作为命令行工具。

- 没有认证

登陆集群中的任意一台服务器，执行下面指令：

```
$ beeline -u "jdbc:hive2://<server_ip/hostname>:10000/<database_name>"
```

这里，<database_name>处提供您想要连接的Inceptor中的数据库的名字，比如default。连接完成后您还可以在beeline命令行中使用USE来切换使用的数据库。

- LDAP认证

登陆集群中的任意一台服务器，执行下面指令：

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

这里，<database_name>处提供您想要连接的Inceptor中的数据库的名字，比如default。连接完成后您还可以在beeline命令行中使用USE来切换使用的数据库。

在<username>和<password>处需要分别提供登陆的用户在LDAP中的用户名和密码，比如下面命令以hive用户身份连接本地的Inceptor中的default数据库：

```
$ beeline -u "jdbc:hive2://localhost:10000/default" -n hive -p 123
```

- Kerberos认证



注意，执行下面指令之前，当前用户必须有一张有效的Kerberos TGT (Ticket Granting Ticket)。查看当前有效的TGT的指令为klist；获取一张有效TGT的指令为kinit。具体操作请参考附录：Kerberos基本操作。Inceptor会根据当前持有TGT的principal判断登陆用户的身份。

登陆集群中的任意一台服务器，执行下面指令：

```
$ beeline -u "jdbc:hive2://<server_ip/hostname>:10000/<database_name>;principal=<principal_name>"
```

这里，<database_name>处提供您想要连接的Inceptor中的数据库的名字，比如default。连接完成后您还可以在beeline命令行中使用USE来切换使用的数据库。

<principal_name>处填写您想要连接的Inceptor server的principal。比如下面命令用hive@tw-node119@TDH这个principal连接本地的Inceptor中的default数据库：

```
$ beeline -u "jdbc:hive2://localhost:10000/default;principal=hive@tw-node119@TDH"
```

InceptorServer HA

如果您设置了InceptorServer HA，并希望使用完整的HA功能，必须采用如下格式进行登陆：

- 没有认证

```
$ beeline -u jdbc:hive2://<server1_ip/hostname>:10000,
<server2_ip/hostname>:10000/<database_name>
```

示例如下：

```
$ beeline -u jdbc:hive2://hadoop1:10000,hadoop2:10000/default
```



- LDAP认证

```
$ beeline -u jdbc:hive2://<server1_ip/hostname>:10000,
<server2_ip/hostname>:10000/<database_name> -n <username> -p <password>
```

示例如下：

```
$ beeline -u jdbc:hive2://hadoop1:10000,hadoop2:10000/default -n hive -p
123456
```

- Kerberos认证

暂时未支持Kerberos证书。

2.5.2. Transwarp命令行

2.5.2.1. Transwarp指令选项

下面为Transwarp命令可以接受的选项：

- **-h <host>**: 指定 host 上的Inceptor Server
- **-e '<query-string>'**: 执行一条SQL查询，例如：

```
$ transwarp -e 'SELECT a.col FROM table1 a' -h localhost
```

- **-f <filename>**

执行文件中的SQL，例如：

```
$ transwarp -f /tmp/test.sql -h node1
```

- **-H, --help**: 打印帮助信息。

2.5.2.2. Transwarp命令行中的非SQL命令

下面指令在Transwarp命令行中执行，指令需要以“;”结尾。

- **quit, exit:** 退出命令行
- **set <key>=<value>:** 设置配置项的值。
- **set:** 打印所有用户或者Inceptor重载的配置项。
- **set -v:** 打印所有配置项。
- **add FILE[S] <filepath> <filepath>***
- **!<os_shell_command>:** 在Transwarp命令行中执行操作系统命令行指令。
- **dfs <dfs_command>:** 在Transwarp命令行中执行HDFS指令。
- **source <filepath>:** 在Transwarp命令行中执行一个脚本。
- **add jar[s] <filepath> [<filepath> ...]:** 将 <filepath> 路径中的jar包添加到Inceptor的classpath中。
- **list jars:** 列出添加过的jar。
- **deleteres jar[s] <filepath> [<filepath> ...]:** 删除添加过的jar。

2.5.3. Beeline命令行

Beeline是InceptorServer 2的命令行工具。

2.5.3.1. beeline指令选项

指令 **beeline** 可接受的选项如下所列：

- **-u <database URL>:** 指定连接的JDBC URL，例如：

```
$ beeline -u "jdbc:hive://localhost:10000/default"
```

- **-n <username>:** 指定连接使用的用户名，例如：

```
$ beeline -u "jdbc:hive://localhost:10000/default" -n alice
```

- **-p <password>:** 指定连接使用的密码，例如：

```
$ beeline -u "jdbc:hive://localhost:10000/default" -n alice -p 123
```

- **-e <query>:** 执行一条查询。查询语句必须放在单引号或者双引号中间，例如：

```
$ beeline -u "jdbc:hive://localhost:10000/default" -e "SELECT * FROM table1"
```

- **-f <file>:** 执行一个脚本。

- **--color=[true|false]:** 是否彩色显示。默认为 **false**。

- **--showHeader=[true|false]:** 打印查询结果时是否打印表头， 默认为 **true**。

- **--headerInterval=<n>**: 指定每多少行查询结果重新打印一次表头， 默认为50。
 - **--force=[true|false]**: 是否在出错的情况下强制执行脚本， 默认为false。
 - **--help**: 打印帮助信息。

2.5.3.2. beeline命令行中的非SQL指令

下面指令在beeline命令行中执行，指令需要以“;”结尾。

- **!quit**: 退出命令行（该指令不能以“;”结尾，为特例）。
 - **set <key>=<value>**: 设置配置项的值。
 - **set**: 打印所有用户或者Inceptor重载的配置项。
 - **set -v**: 打印所有用户或Inceptor重载的配置项以及Hadoop中所有的配置项。
 - **add FILE[S] <filepath> <filepath>***
 - **dfs <dfs_command>**: 在Transwarp命令行中执行HDFS指令。
 - **add jar[s] <filepath> [<filepath> ...]**: 将 <filepath> 路径中的jar包添加到Inceptor的classpath中。
 - **list jars**: 列出添加过的jar。
 - **deleteres jar[s] <filepath> [<filepath> ...]**: 删除添加过的jar。

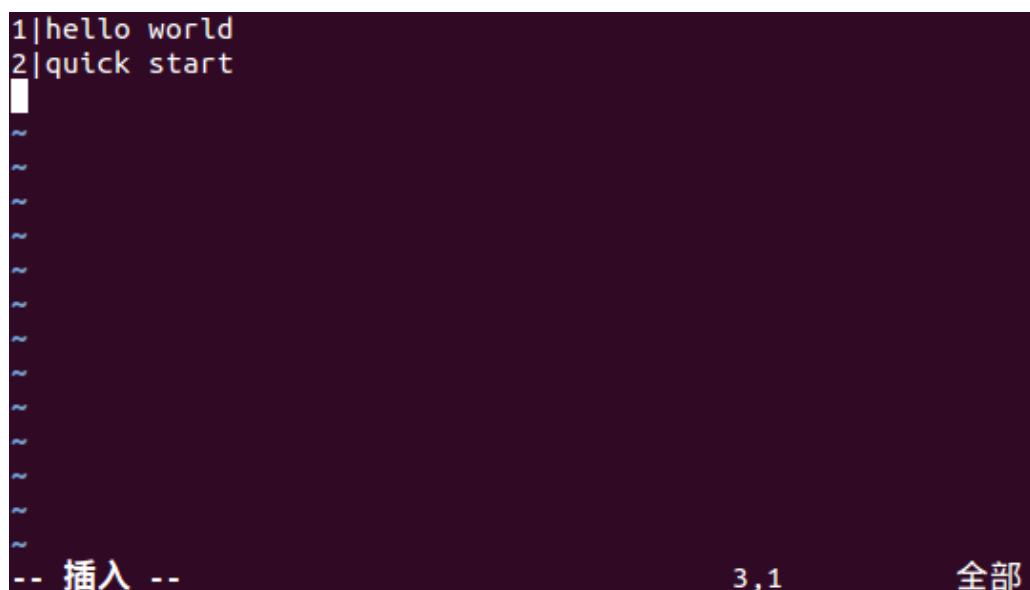
2.5.4. Inceptor命令行快速入门

任务：在本地创建文本，导入到Inceptor新建的表格中。

1. 登录安装有Inceptor Server的节点，新建一个文本文件quickstart.txt，并将其放在tmp文件夹中。

```
vim /tmp/quickstart.txt
```

2. 向quickstart.txt中写入两行文字，如下图所示。



输入后按Esc键， 输入:wq保存退出。

3. 命令行连接Inceptor

- InceptorServer 1

```
transwarp -t -h localhost
```

- InceptorServer 2 简单认证

```
beeline -u "jdbc:hive2://localhost:10000/default"
```

- InceptorServer 2 LDAP认证

```
beeline -u "jdbc:hive2://localhost:10000/default" -n <username> -p <password>
```

- InceptorServer 2 Kerberos认证

```
kinit <your_principal>
beeline -u "jdbc:hive2://localhost:10000/default;principal=hive/node1@TDH"
```

4. 新建外表quickstart

```
CREATE EXTERNAL TABLE quickstart (a INT, b STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY '|';
```

5. 将位于本地的quickstart.txt导入至表格quickstart中

```
LOAD DATA LOCAL INPATH '/tmp/quickstart.txt' INTO TABLE quickstart;
```

注意如果Inceptor使用LDAP或者Kerberos认证，您需要是这个文件的owner。

6. 查看表格内容，结果为：

```
[localhost:10000] transwarp> select * from quickstart;
1      hello world
2      quick start
Time taken: 3.74 seconds
```

2.6. Inceptor Library

Inceptor Library 是Inceptor项目的子模块，旨在为Inceptor的上层应用提供对强大易用的各种Library的支持和服务。Library又可被称为库，包含并管理着各种数据库实体，这些数据库实体是指所有可通过语句创建的数据库相关对象，例如：UDF、PL/SQL函数、存储过程、包、数据库、表、视图、DBlink等。这些对象被预置在系统中，用户的使用与否不决定它们的存在性。通常，一个初始用户在初次使用系统时，即使没有创建任何对象实体，也可以对它们发起访问。

目前，有三个库默认受Inceptor Library管理：Data Dictionary、Discover Library、Midas Library，分别服务于Inceptor、Discover和Midas。以Data Dictionary为例，该库装载着已经被预定义的各个视图，这些视图被称为字典表，字典表负责记录系统中所有实体对象的元信息，以方便用户对元信息的访问查询。

Inceptor

Library在安装Inceptor时自动创建，并且当集群在进行自动化升级时随之升级。除了自动化的管理控制，Inceptor Library还为用户提供了简单的命令来创建、删除、升级已定义的Library。Inceptor Library允许Inceptor的使用者以自定义的方式，定制特定的Library，并允许用户对Library做自动化、持久化的设置。库中内容的创建、删除、升级完全由Inceptor Library控制，用户需要做的，只是按照一定的规范提供自定义库的创建、删除、升级语句，Inceptor Library将负责版本管理。

在已成功部署的集群中，Inceptor Library的数据和脚本位于“/usr/lib/ngmr-shell/library/”目录下。该目录的结构如下图所示：

```
tree /usr/lib/ngmr-shell/library/
└── data_dictionary
    ├── inception
    │   ├── install
    │   │   └── install-1.1.sql
    │   ├── uninstall
    │   │   └── uninstall-1.1.sql
    │   └── upgrade
    └── mysql
        ├── install
        │   └── install-1.1.sql
        └── uninstall
            └── uninstall-1.1.sql
    └── discover
        └── inception
            ├── data
            │   ├── classification_predict.txt
            │   └── regression_train.txt
            ├── install
            │   └── install-1.1.sql
            ├── uninstall
            │   └── uninstall-1.1.sql
            └── upgrade
    └── midas
        └── inception
            ├── data
            │   ├── Deals.csv
            │   ├── Transactions.csv
            │   └── Weighting.csv
            ├── install
            │   └── install-1.1.sql
            ├── uninstall
            │   └── uninstall-1.1.sql
            └── upgrade
```

从该结构可以看出，Inceptor Library维护着各Library的安装、卸载、升级脚本，对于Discover和Midas还通过data目录管理着相关所需数据。

Inceptor Library帮助实现了对于Inceptor之上工作的应用的结构性管理，并且操作方便有效。对于不同库，用Inceptor Library进行定义和管理的思想整体是一致的，但是操作语法、步骤略有不同，具体执行时请用户参考各个库的相关使用手册。

3. Inceptor SQL手册

3.1. Inceptor SQL手册一览

企业级数据仓库、数据集市等应用大多基于SQL来开发，而Hadoop业界的产品大部分对SQL的兼容程序比较差，或者不支持SQL的模块化扩展，因而应用迁移的成本非常高，甚至是不具备可行性。为了降低应用迁移成本，Transwarp Inceptor开发了完整的SQL编译器，支持ANSI SQL 92和SQL 99标准，并且支持ANSI SQL 2003 OLAP核心扩展，可以满足绝大部分现有的数据仓库业务对SQL的要求，方便应用平滑迁移。本手册将介绍Inceptor SQL的使用。

3.1.1. Inceptor SQL中的语句类型

Inceptor SQL中的语句可以分为下面几类：

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

- a. **CREATE**: 创建Inceptor对象。
- b. **DROP**: 删除Inceptor对象。
- c. **ALTER**: 编辑/修改Inceptor对象。
- d. **TRUNCATE**: 清空表中数据。
- e. **SHOW**: 列出Inceptor对象。
- f. **DESCRIBE**: 描述Inceptor对象。

[数据定义语言 \(DDL\)](#) 中将会有介绍。

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

- a. **LOAD**: 将文件中的数据导入表。
- b. **INSERT**: 将查询结果插入表。
- c. **SELECT**: 查询表中数据。**SELECT** 支持下面子句：
 - **WHERE**: 查询结果过滤。
 - **JOIN**: 表连接。
 - **GROUP BY**: 聚合操作。
 - **DISTRIBUTE/SORT/CLUSTER BY**: 查询结果的分桶和桶内排序。
 - **UNION/INTERCEPT/EXCEPT**: 查询结果的集合运算。
 - **SELECT** (子查询)。
 - **START WITH/CONNECT BY/AND PRIOR** (层次化查询)。

[数据操作语言 \(DML\)](#) 中将会有介绍。

- TCL (Transaction Control Language, 事务控制语言)

- a. **BEGIN TRANSACTION:** 开始事务。
- b. **COMMIT:** 提交事务。
- c. **ROLLBACK:** 回滚事务。

[事务控制语言 \(TCL\)](#) 中将会详细介绍。

- **DCL (Data Control Language, 数据控制语言)**

- a. **GRANT:** 授予权限。
- b. **REVOKE:** 收回权限。

[数据控制语言 \(DCL\)](#) 中将会介绍。Inceptor中的安全管理，包括认证、授权、资源隔离等将在[Inceptor多租户手册](#)中详细介绍。

- **Database Links (数据库连接)**

数据库连接让用户可以在Inceptor中使用其他数据库（Oracle, MySQL, DB2, PostgreSQL以及其他Inceptor中的数据）。

[Database Links](#)中将会介绍。

- **数据稽查**

将原始数据中的脏数据放入用户指定的error table，在脏数据存在的情况下尽可能的保护系统或保证业务的顺畅执行。

[数据稽查](#)中将会介绍。

- 另外，Inceptor中[不同类型的表](#)在DDL和导数据方面有较大差异，本手册将在下面几章专门介绍[不同类型表的DDL和导数据方法](#)。我们建议在浏览过[数据定义语言 \(DDL\)](#) 和[数据操作语言\(DML\)](#)对Inceptor中的DDL和数据导入有大致概念后，阅读下面几章，根据您的需求来建表和导入数据。其中，ORC表部分还会介绍ORC事务表特有的CRUD。

[TEXT表](#)

[ORC表](#)

[CSV表](#)

[Holodesk表](#)

[分桶表](#)

[分区表](#)

3.1.2. 手册的格式规范

本手册将包含大量Inceptor SQL语法和样例的介绍，为了增强可读性，我们在语法中遵循下面的格式：

- Inceptor SQL指令的关键字将大写，在实际操作中，Inceptor SQL是 大小写不敏感 的。
- 变量用小写，放在“< >”中，例如：

```
SELECT COUNT(*) FROM <table>;
```

- 可选内容放在“[]”中。例如

```
CREATE TABLE [IF NOT EXISTS] <table> (<column> <data_type>, [<column_name> <data_type>, ...]);
```

- “|”表示“或者”，例如：

```
DESC|DESCRIBE <table>;
```

同时表示下面两个语句：

```
DESC <table>;
DESCRIBE <table>;
```

3.1.3. Inceptor SQL手册中使用的表

在Inceptor SQL使用指南中，将会用几张虚构的表作为例子。以下是这些表的内容：

- 用户信息表 user_info：列从左到右依次为name, acc_num, password, citizen_id, bank_acc, reg_date, acc_level：

马**	6513065	115591	14*****	960*****	41	20110101	A
祝**	6670192	205239	23*****	737*****	71	20100101	C
华*	5224133	1531547	42*****	326*****	07	20080214	B
魏**	13912384	841242	52*****	685*****	48	20091202	A
宁**	14580952	986634	42*****	977*****	76	20081031	D
邱*	10700735	737297	34*****	143*****	18	20121024	A
李*	18725869	600709	46*****	430*****	84	20130702	E
潘**	6600641	990590	51*****	484*****	08	20110430	C
李**	12755506	015859	31*****	424*****	37	20110916	D
管**	12394923	783438	33*****	999*****	74	20141003	C

- 交易信息表transactions：列从左到右依次为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
289818112	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	KC9102928	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，列名从左到右为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, 列名从左到右为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
MAT100	微积分

3.2. SQL语句的编程规范

在写SQL语句时，希望用户能遵循如下的编写规范。规范化的SQL编程可以帮助用户或者我们的支持人员日后对语句含义与实现的功能进行理解。尤其是当用户在写复杂度较高、长度较长的语句时应格外注意。

3.2.1. 注释

不换行的简单注释，在注释文字前用“--”标识。简单注释特别适用于SQL语句中对字段的注解，如临时表建立时对字段的描述。

如果注释内容特别多，多于一行，应把多行注释写在“/* … */”内。多行描述区主要用于描述该脚本的功能，作者、目标表和源表、修改记录等。

例 3. 注释示例

```
-- 创建一张新表
CREATE TABLE table_A( id INT, name STRING);

/* 第一行注释：创建一张表
第二行注释：再导入数据 */
CREATE TABLE table_B( id INT, name STRING);
INSERT INTO table_B SELECT * FROM table_c;
```

3.2.2. 大小写规则

SQL语句中的所有保留字均需大写，且不要使用缩写，如ALL, AS, CASE, CREATE, DATABASE, DELETE, FROM, IN, INSERT, JOIN, LEFT, NO, NOT, NULL, OUT, SELECT, TABLE, TITLE, UPDATE, VIEW, WHERE等。

表的别名也大写。

3.2.3. 缩进与换行

建议将SQL语句按照子句分行编写，以SELECT、FROM、WHERE、UPDATE、INSERT为起始另起一行，尽量对齐各子句的起始位置。

1. 逗号放在每行字段的开头。

2. 分号放在SQL语句的末尾。
3. 每行宽度不超过120字符（每个字符为8个点阵宽），超过行宽的代码应换行并与上行对齐编排。
4. 把AS部分和相应字段放在同一行。
5. 当多个字段包含“AS”时，建议尽量将各“AS”对齐在同一列上。
6. 同一子句中，应排齐“WHERE”、“AND”、“OR”这三个单词的末尾。

注：缩进时不建议使用Tab，建议使用4个空格。

例 4. 缩进与换行嵌套示例

```

SELECT First_Name
      ,Last_Name
      ,Employee_Number    AS    Employee_ID
      ,Salary_Amount      AS    Employee_Salary
  FROM Employee
 WHERE Salary_Amount > 35000.00
   AND Salary_Amount < 85000.00
   OR Department_Number = 403
 ORDER BY First_Name;

```

3.2.4. 子查询嵌套

对于复杂语句，代码的分层编排非常重要，一般对嵌套语句的格式有以下两点要求：

1. 对应的括号应尽量排在同一列。
2. 同一级别的子句内部要对齐。

例 5. 子查询嵌套示例

```

SELECT ...
  FROM vt_Txn_Trade_His_Date_B    a1
 LEFT JOIN (
    SELECT b1.Trade_Seat_Code
          ,b2.Acct_ID
      FROM vt_Txn_Trade_His_Date_B    b1,
           (SELECT Acct_ID
              ,MIN(Trade_No)    AS Trade_No)
      FROM vt_Txn_Trade_His_Date_B
     WHERE TRIM(Acct_ID) <> ''
     GROUP BY Acct_ID
     HAVING COUNT(Acct_ID) >= 2
    ) b2
 WHERE b1.Trade_No = b2.Trade_No
  ) a2
ON    a1.Acct_ID = a2.ACCT_ID;

```

3.2.5. 表别名

1. 表别名建议以简单字符命名。
2. 多层次的嵌套子查询别名需体现层次关系。
3. 对于同一层的多个子句，建议别名采用相同前缀附加不同数字后缀（1、2、3…）的形式，以便识别与区分子句关系。
4. 在需要的情况下对表别名添加注释。示例参照[子查询嵌套示例](#)。

3.2.6. 运算符前后间隔要求

算术运算符、逻辑运算符的前后至少保留一个空格。

例 6. 运算符间隔示例

```
SELECT First_Name
      ,Last_Name
      ,Employee_Number    AS      Employee_ID
      ,Salary_Amount      AS      Employee_Salary
      ,(Salary_Amount + Salary_Amount * 2) / 1.5
                                AS      Employee_Salary_Double
  FROM Employee
 WHERE Salary_Amount > 35000.00
   AND Salary_Amount < 85000.00
   OR Department_Number = 403
 ORDER BY First_Name;
```

3.2.7. 临时表

建议在语句中尽量使用可变临时表，通过临时表的使用体现逻辑和加工的流程。

3.2.8. GROUP BY / ORDER BY

在查询语句的ORDER BY和GROUP BY操作中，允许用户用字段序号代表字段，例如“group by[第一个字段]”可以写为“group by 1”。但是不允许字段名和序号同时出现，应保证采用一种模式时，不出现另一种模式。

[group by和order by的示例](#)中的两种写法是等价的，但不能混用。

例 7. group by和order by的示例

```
SELECT First_Name ①
      ,Last_Name
      ,Employee_Number    AS      Employee_ID
      ,Salary_Amount      AS      Employee_Salary
  FROM Employee
 WHERE Salary_Amount > 35000.00
   AND Salary_Amount < 85000.00
   OR Department_Number = 403
 ORDER BY First_Name
         ,Last_Name
         ,Employee_Salary;

SELECT First_Name ②
      ,Last_Name
      ,Employee_Number    AS      Employee_ID
      ,Salary_Amount      AS      Employee_Salary
  FROM Employee
 WHERE Salary_Amount > 35000.00
   AND Salary_Amount < 85000.00
   OR Department_Number = 403
 ORDER BY 1, 2, 4;
```

① 写法一，仅用字段名称表达。

② 写法二，仅用数字表达。

3.2.9. 字段类型

字段在做比较或转换的时候应该使用显式的写法对数据做处理，以防止因字段格式的差异导致执行失败，如：

- 文本：在访问文本字段时建议做trim()处理，去除多余空格。例如，table_A.name的某值为' Lucy'，table_B.name的某值为' Lucy '，若不进行处理，在做name等值匹配时这两条的比较结果将是false。
- 日期：执行日期相关运算时，必要时应用形如“CAST(‘20121231’ AS DATE FORMAT ‘YYYYMMDD’)”的语句做格式转化，防止录入数据时以STRING类型保存的日期字段，在计算时被视为表达时间的特有类型（如date、timestamp）处理，导致结果异常。

3.3. 数据定义语言 (DDL)

数据定义语言 (Data Definition Language) 一般由 **CREATE**, **DROP** 和 **ALTER** 开头，作用于DATABASE, TABLE, VIEW, FUNCTION等图表对象，对它们进行添加，删除和修改等操作。

3.3.1. CREATE/DROP/ALTER DATABASE

3.3.1.1. CREATE DATABASE

CREATE DATABASE 创建一个数据库。

语法

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

- ① 如果 <database_name> 指定的数据库已经存在，Inceptor会报错，加上 **IF NOT EXISTS** 选项则可以让Inceptor不报错。
- ② 可选项，用 **COMMENT** 为数据库加注释，注意注释要放在单引号中。
- ③ 可选项，用于添加一些键值对形式的数据库属性。

例 8. 创建数据库

建一个数据库exchange_platform。

```
CREATE DATABASE IF NOT EXISTS exchange_platform;
```

3.3.1.2. DROP DATABASE

DROP DATABASE 删除一个给定名字的数据库。

语法

```
DROP DATABASE [IF EXISTS] <database_name> ①
[RESTRICT|CASCADE]; ②
```

- ① 如果 `<database_name>` 指定的数据库不存在，执行删除操作时会导致Inceptor报错，加上 `IF EXISTS` 选项则可以让Inceptor不报错。
- ② 可选项，默认值为 `RESTRICT`。`RESTRICT` 使 `DROP DATABASE` 语句不能删除非空数据库；`CASCADE` 则使 `DROP DATABASE` 将数据库以及里面的表一并删除。

例 9. 删除数据库

```
DROP DATABASE IF EXISTS exchange_platform;
```

3.3.1.3. ALTER DATABASE

`ALTER DATABASE` 可以用于修改数据库的 `DBPROPERTIES` 和owner。

语法：修改 `DBPROPERTIES`

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

例 10. 修改 DBPROPERTIES

```
ALTER DATABASE exchange_platform SET DBPROPERTIES ('date'='2015-12');
```

语法：修改数据库owner

```
ALTER DATABASE <database_name> SET OWNER [USER|ROLE] <user_or_role>;
```

数据库owner可以是Inceptor的用户也可以是角色。

例 11. 修改数据库owner

```
ALTER DATABASE exchange_platform SET OWNER USER alice;
```

3.3.1.4. USE DATABASE

`USE DATABASE` 指定当前使用的数据库，使得指定数据库中的表、视图和流可以直接使用。

语法

```
USE <database_name>;
```

3.3.2. CREATE/DROP/ALTER/TRUNCATE TABLE

3.3.2.1. CREATE TABLE

`CREATE TABLE` 创建一张表。建表方式可以分为三大类：直接建表（建表时定义列）、`CREATE TABLE LIKE` 和 `CREATE TABLE AS SELECT`。

3.3.2.1.1. 直接建表

语法：

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] ①
[<database_name>.]<table_name> ②
[(<column_name> <data_type> [COMMENT '<column_comment>'] ③
[, <column_name> <data_type> ...])
[COMMENT '<table_comment>'] ③
[PARTITIONED BY (<part_key> <data_type> [COMMENT '<partition_comment>'] ④
[, <part_key> <data_type>...])]
[CLUSTERED BY (<col_name> [, <col_name>...]) ⑤
[SORTED BY (<col_name> [ASC|DESC] [, <col_name> [ASC|DESC]...])] ⑥
INTO <num_buckets> BUCKETS]
[
[ROW FORMAT <row_format>] ⑦
[STORED AS (TEXTFILE|ORC|CSVFILE)] ⑧
| STORED BY '<storage.handler.class.name>' [WITH SERDEPROPERTIES (<...>)] ⑨
]
[LOCATION '<hdfs_path>'] ⑩
[TBLPROPERTIES ('<property_name>='<property_value>', ...)]
```

- ① `TEMPORARY` 为临时表选项，`EXTERNAL` 为外表选项。
- ② 加上 `<database_name>`，则将表建在指定的数据库中。
- ③ 可以为列、表和分区用 `COMMENT` 加注释，注意注释要放在单引号中。
- ④ 指定分区键，具体使用方式请参考“分区表”。
- ⑤ 分桶子句，具体使用方式请参考“分桶表”。
- ⑥ 桶内排序选项，`ASC` 为升序，`DESC` 为降序。
- ⑦ 行格式，在建TEXT表时使用，具体细节请参考“TEXT表”部分。
- ⑧ 指定文件格式，该选项在建TEXT表，ORC表或CSV表时使用。
- ⑨ 指定使用的storage handler。
- ⑩ 指向HDFS上的一个目录。这个选项我们推荐只在建外表时使用，也就是和 `EXTERNAL` 选项合用（虽然也可以在建内表时使用，但是我们 不建议 这样做）。该路径必须是绝对路径，并且执行操作的用户必须是这个路径指向的目录或文件的owner。如果 `<hdfs_path>` 指向的目录不存在，Inceptor会尝试新建这个目录，但是安全模式下Inceptor可能没有在指定路径新建目录的权限，所以星环科技建议尽量避免让 `<hdfs_path>` 指向不存在的目录。

表属性，由键值对表示，在建Holodesk表，ORC事务表和CSV表时使用，具体细节请参考“ORC表”，“Holodesk表”和“CSV表”。

临时表



临时表仅在当前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。

例 12. 通过定义列建表

将表建在当前数据库中：

```
CREATE TABLE user_info (
    name          STRING,
    acc_num       STRING,
    password      STRING,
    citizen_id   STRING,
    bank_acc     STRING,
    reg_date     DATE,
    acc_level    STRING
);
```

将表建在指定数据库中：

```
CREATE TABLE exchange_platform.user_info (
    name          STRING,
    acc_num       STRING,
    password      STRING,
    citizen_id   STRING,
    bank_acc     STRING,
    reg_date     DATE,
    acc_level    STRING
);
```

建HDFS外表：

```
CREATE EXTERNAL TABLE user_info (
    name STRING,
    acc_num STRING,
    password STRING,
    citizen_id STRING,
    bank_acc STRING,
    reg_date DATE,
    acc_level STRING
)
LOCATION '/user/exchange/user_info/' ;
```

直接建表可用的选项有很多，这些选项可以在同一条建表语句中使用，它们出现的顺序须遵循上面的语法。在本手册中，我们还会介绍很多其他建表的语法，包括：

- 建分区表
- 建分桶表
- 建TEXT表
- 建ORC表
- 建Holodesk表

3.3.2.1.2. CREATE TABLE LIKE

CREATE TABLE LIKE 通过拷贝一张已存在表或视图的定义建表，但不拷贝已存在表的数据。

语法

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] ①
[<database>.]<table_name> ②
LIKE <existing_table_or_view_name>
[LOCATION '<hdfs_path>'] ③
```

① **TEMPORARY** 为临时表选项，**EXTERNAL** 为外表选项。

- ② 加上 <database_name>, 则将表建在指定的数据库中。
- ③ 指定表在HDFS上的目录。

例 13. 通过拷贝表定义建表

```
CREATE TABLE IF NOT EXISTS exchange_platform.user_info LIKE user_info;
```

3.3.2.1.3. CREATE TABLE AS SELECT (CTAS)

CTAS 把一次查询结果建成表，表的schema和查询结果的schema一致。

语法

```
CREATE TABLE <table_name>
[
  [ROW FORMAT <row_format>]
  [STORED AS <file_format>]
  | STORED BY '<storage.handler.class.name>' [WITH SERDEPROPERTIES (<...>)]
]
[LOCATION <hdfs_path>]
AS SELECT <select_statement>;
```

CTAS 分为两个部分，CREATE 部分和 SELECT 部分。SELECT 部分可以是任何查询语句；CREATE 部分将查询结果写入表中。CTAS 不能用于建分区表、分桶表和外表，使用 CTAS 建表时也不可以定义列。

例 14. CTAS

用user_info表做为例子，user_info中含有帐户密码，身份证号码和银行帐户等会影响帐户安全的信息，现在用 CTAS 建一张不包含这些信息的表。

```
CREATE TABLE nonsecure_user_info AS SELECT name, acc_num, reg_date, acc_level FROM user_info;
```

3.3.2.2. DROP TABLE

语法

```
DROP TABLE [IF EXISTS] <table_name>;
```

当被删除的表是托管表时，表的元数据和表中数据都会被删除。如果被删除的表是外部表，则只有它的元数据会被删除。

3.3.2.3. ALTER TABLE

3.3.2.3.1. 重命名：ALTER TABLE RENAME TO

语法

```
ALTER TABLE <table_name> RENAME TO <new_table_name>;
```

3.3.2.3.2. 修改或添加TBLPROPERTIES

我们可以用这个语句给表添加和修改自定义的表属性。

语法

```
ALTER TABLE <table_name> SET TBLPROPERTIES ('<property_name>' = '<property_value>' ... );
```

3.3.2.3.3. 修改或添加SERDEPROPERTIES

语法

```
ALTER TABLE <table_name> SET SERDEPROPERTIES ('<property_name>' = '<property_value>' ... );
```

3.3.2.3.4. 修改外表目录

语法

```
ALTER TABLE <table_name> SET LOCATION '<new_location>';
```

将外表指向的HDFS目录改为 `<new_location>`。注意，执行该操作的用户必须是 `<new_location>` 的owner。

3.3.2.4. TRUNCATE TABLE

TRUNCATE TABLE 清空表或者分区中的数据，但不删除表或分区的元数据。这个操作只能用于托管表，不能用于外表。

语法

```
TRUNCATE TABLE table_name [PARTITION (<partition_key> = <partition_value>[, ...] )];
```

加上 `[PARTITION (<partition_key> = <partition_value>[, ...])]` 选项可以只清空指定分区中的数据。

例 15. 清空表中数据

```
TRUNCATE TABLE user_info;
```

例 16. 清空指定表中指定分区的数据

```
TRUNCATE TABLE user_info_part PARTITION (acc_level='A');
```

3.3.3. CHANGE/ADD/REPLACE COLUMNS

3.3.3.1. CHANGE COLUMNS

修改列名、列的数值类型、列在表中的位置和列的注解

语法

```
ALTER TABLE table_name CHANGE [COLUMN] col_old_name col_new_name column_type
[COMMENT col_comment] [FIRST|AFTER column_name] [CASCADE|RESTRICT]; ①
```

- ① 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。

上面指令可以对一列的列名、列的数值类型、列在表中的位置、列的注解或者这些的任意组合做出修改。这个指令仅仅修改表的元数据。

举例

```
CREATE TABLE test_change (a INT, b INT, c INT);

//将列a重命名为列a1
ALTER TABLE test_change CHANGE a a1 INT;

//将列a1重命名为列a2，它的数据类型改为STRING，并将它放在列b之后
ALTER TABLE test_change CHANGE a1 a2 STRING AFTER b;
//表test_change的新结构为：(b INT, a2 STRING, c INT)

//将列c重命名为c1，并将它定位第一列
ALTER TABLE test_change CHANGE c c1 INT FIRST;
//表test_change的新结构为：(c1 INT, b INT, a2 STRING)
```

3.3.3.2. ADD|REPLACE COLUMNS

- ADD COLUMNS：在表中加入新的列
- REPLACE COLUMNS：将原来的表中列结构替换为新的列结构

语法

```
ALTER TABLE table_name ADD|REPLACE COLUMNS (col_name data_type [COMMENT col_comment], ...)
[CASCADE|RESTRICT] ①
```

- ① 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。

ADD COLUMNS

可以将新的列加入表中，位置在所有列之后，分区之前。

举例

```
CREATE TABLE test_change (a INT, b INT, c INT);

// 在表test_change中添加一个新列d，类型为INT
ALTER TABLE test_change ADD COLUMNS
```

REPLACE COLUMNS

REPLACE COLUMNS将所有已有的列删除然后加入新指定的列。只能对使用Inceptor自带SerDe（DynamicSerDe, MetadataTypedColumnsetSerDe, LazySimpleSerDe and ColumnarSerDe）的表使用REPLACE COLUMNS。REPLACE COLUMNS也可以被用于删除列：

举例

```
CREATE TABLE test_change (a INT, b INT, c INT);
//将test_change中的列由(a INT, b INT, c INT)替换为(a int, b int)起到将列c删除的效果
ALTER TABLE test_change REPLACE COLUMNS (a int, b int);
```



只有TEXT、CSV、基于定宽文本文件外表这三种表支持对 字段类型 的修改。

3.3.4. CREATE/DROP VIEW

3.3.4.1. CREATE VIEW

语法

```
CREATE VIEW [IF NOT EXISTS] <view_name> [(<column_name>, <column_name>, ...) ] ①
AS SELECT <select_statement>; ②
```

- ① 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。
- ② 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。

例 17. 创建视图

```
CREATE VIEW non_secure_info AS SELECT name, reg_date, acc_level FROM user_info;
```

注意:不支持CREATE VIEW AS SELECT ... UNION SELECT ...

3.3.4.2. DROP VIEW

语法

```
DROP VIEW [IF EXISTS] <view_name>;
```

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，Inceptor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，但是用户需要自己处理这些变成无效的视图。

3.3.5. CREATE/DROP FUNCTION

本节介绍如何使用InceptorSQL创建自定义函数。自定义函数分为 临时函数和永久函数，临时函数在重启Inceptor前在各个session间都是有效的，重启Inceptor后该函数将不再存在，如需使用需要重新创建。永久函数则在Inceptor重启后依然可以使用。

3.3.5.1. CREATE/DROP TEMPORARY FUNCTION

语法：创建临时函数

```
CREATE TEMPORARY FUNCTION <function_name> AS <class_name>;
```

以上语句创建一个由class_name实现的临时函数。这个新创建的函数只能在当前session使用。用户可以使用任意一个在class path中的class。用户也可以通过 **ADD JAR[S]** 向classpath加jar包：

语法：添加jar包

```
ADD JAR[S] <local_or_hdfs_path>; ①
```

① **<local_or_hdfs_path>** 指定添加的jar包的路径。添加的jar可以是本地的（Inceptor Server所在节点）或者HDFS上的。请确保hive用户对jar所在目录有读权限。



注意，通过 **ADD JAR[S]** 添加的jar包不能有重名的类。

语法：删除临时函数

```
DROP TEMPORARY FUNCTION [IF EXISTS] <function_name>
```

更新临时函数的jar

如果您需要更新临时函数的jar，请重启Inceptor服务，重新添加jar包并创建该临时函数。

例 18. 临时函数的创建

创建临时函数

```
ADD JAR /tmp/udf_first.jar; --添加jar包
CREATE TEMPORARY FUNCTION tempf AS 'org.apache.hadoop.hive.ql.udf.tempf.UDFPrime1';
DESC FUNCTION tempf;
+-----+
| tab_name      |
+-----+
| function from udf_first |
+-----+

LIST JAR;
+-----+
| file path      |
+-----+
| /tmp/udf_first.jar |
+-----+
```

3.3.5.2. CREATE/DROP PERMANENT FUNCTION



版本信息

永久函数的创建和删除是TDH4.5.2开始支持的新增功能。

语法：创建永久函数

```
CREATE PERMANENT FUNCTION [<db_name>.]<function_name> AS <class_name>
[USING JAR|FILE|ARCHIVE '<file_uri>' [, JAR|FILE|ARCHIVE '<file_uri>'] ];
```

说明

- 以上语句创建一个由class_name实现的永久函数。这个函数将在metastore登记，不需要在每个session重新建临时函数。需要加入环境的jar包，文件或者档案可以通过 **USING** 指定。第一次使用该函数时，

这些资源会像被 `ADD JAR/FILE` 一样加到环境中。如果 Inceptor 不在 local mode，那么资源的地址也必须是非本地 URI，比如 HDFS 地址。

- 该函数会被加进当前使用的数据库，或者是由 `db_name` 指定的数据库。该函数可以通过 `db_name.function_name` 调用，如果函数属于当前数据库，那么也可以直接通过 `function_name` 调用。

语法：删除永久函数

```
DROP PERMANENT FUNCTION [IF EXISTS] <function_name>;
```

3.3.6. SHOW

`SHOW` 用来列出数据库、表、视图、列、函数等图表对象。

3.3.6.1. SHOW DATABASES

列出数据库。

语法

```
SHOW DATABASES [LIKE '<identifier_with_wildcards>'];
```

使用 `[LIKE '<identifier_with_wildcards>']` 选项可以用通配符 “*”（表示任意个字母）和 “|”（表示选择）进行模糊搜索。比如，“employees”，“emp*”，“emp*|ees”都可以和“employees”匹配，`<identifier_with_wildcards>*` 需要放在引号中。

例 19. 模糊搜索数据库

```
SHOW DATABASES LIKE 'test*';
SHOW DATABASES LIKE 'test*|*db';
```

3.3.6.2. SHOW TABLES

列出表和视图。

语法

```
SHOW TABLES [IN <database_name>] [LIKE '<identifier_with_wildcards>'];
```

如果想要查看非当前数据库中的表和视图，可以使用 `[IN <database_name>]` 来指定数据库。使用 `[LIKE '<identifier_with_wildcards>']` 选项可以用通配符进行模糊搜索，`<identifier_with_wildcards>` 需要放在引号中。

3.3.6.3. SHOW TBLPROPERTIES

语法

```
SHOW TBLPROPERTIES <table_name>;
SHOW TBLPROPERTIES <table_name>("<property_name>");
```

SHOW TBLPROPERTIES

查看指定表的TBLPROPERTIES。如果只要查看某个属性，可以在表名后面的括号中指明。

3.3.6.4. SHOW CREATE TABLE

查看指定表或视图的建表语句。

语法

```
SHOW CREATE TABLE ([<db_name>.]<table_name>|<view_name>);
```

3.3.6.5. SHOW PARTITIONS

列出指定表中的所有分区。

语法

```
SHOW PARTITIONS <table_name> [PARTITION (<partition_key> = <partition_value>, ...)];
```

使用 [PARTITION (<partition_key> = <partition_value>, ...)] 选项可以过滤查看的结果。

3.3.6.6. SHOW COLUMNS

列出所有指定表中的列

语法

```
SHOW COLUMNS FROM|IN <table_name> [FROM|IN <database_name>];
```

FROM 和 **IN** 没有区别，可替换使用。

3.3.6.7. SHOW FUNCTIONS

列出所有函数。

语法

```
SHOW FUNCTIONS ['<regex>'];
```

加上 [<regex>] 则显示所有匹配指定正则表达式的函数。正则表达式 <regex> 必须放在引号中。

3.3.7. DESCRIBE

DESCRIBE（可简写为 **DESC**）用于查看数据库、表、视图、列、分区和函数的属性。**DESCRIBE** 的两个变化：**DESCRIBE EXTENDED** 和 **DESCRIBE FORMATTED** 会输出一些额外信息。

3.3.7.1. DESCRIBE DATABASE

语法

```
DESCRIBE|DESC DATABASE [EXTENDED] <db_name>;
```

3.3.7.2. DESCRIBE TABLE

语法

```
DESCRIBE|DESC [EXTENDED|FORMATTED] [<db_name>.]<table_name>;
```

3.3.7.3. DESCRIBE COLUMN

语法

```
DESCRIBE|DESC [EXTENDED|FORMATTED] [<db_name>.]<table_name>.<column_name>;
```

3.3.7.4. DESCRIBE PARTITION

语法

```
DESCRIBE|DESC [EXTENDED|FORMATTED] [<db_name>.]<table_name> PARTITION (<partition_key> = <partition_value>);
```

例 20. 描述分区

```
DESC user_info_part PARTITION (acc_level='A');
```

3.3.7.5. DESCRIBE FUNCTION

语法

```
DESCRIBE|DESC FUNCTION [EXTENDED] <function_name>;
```

DESCRIBE FUNCTION 给出函数的用法和示例。

例 21. 描述函数

```
DESC FUNCTION EXTENDED tdh_todate;
```

3.4. 数据操作语言(DML)

Inceptor SQL中的数据操作语言由 **LOAD**, **INSERT** 和 **SELECT** 开头。其中 **LOAD** 和 **INSERT** 都可以用来填充表。填充表是指将指定的文件 (**LOAD** 语句) 或者查询的结果 (**INSERT** 语句) 放入表对应的目录中，这个目录不能有子目录，也就是说如果被填充的表是分区表，则必须指定将数据文件放入的分区中。一些常见的DML如 **UPDATE**, **DELETE** 和 **INSERT ... VALUES** 只能对**ORC事务表**、Hyperbase内存表、ES

表使用。而INSERT ... SELECT、SELECT则可以作用于任意类型的表。以下是Inceptor中的支持的数据操作语言。

3.4.1. 导入数据：LOAD

LOAD语句将文件中的数据导入已创建的表。导入操作数据文移动到表或分区所对应的目录中。



虽然可以使用 **LOAD** 导入数据，但是安全模式下的权限设置步骤较多，我们不推荐使用 **LOAD** 导入数据，而是推荐建外表并将外表 **LOCATION** 设置为数据所在目录的方法，具体方法请参考[CREATE/DROP/ALTER/TRUNCATE TABLE](#)。

语法

```
LOAD DATA [LOCAL] INPATH '<path>' ①
[OVERWRITE] INTO TABLE <tablename> ②
[PARTITION (<partition_key>=<partition_value>, ...)]; ③
```

- ① **<path>** 可以指向一个文件也可以指向一个目录。**[LOCAL]** 是本地路径选项，加上该选项后 **<path>** 是Inceptor Server所在节点的本地目录。如果用HDFS上的路径，当 **<path>** 指向一个文件时，Inceptor会将文件移入表在HDFS上的目录中；指向一个目录时，Inceptor会将目录下所有的文件移入表在HDFS上的目录中。如果用本地路径，Inceptor将文件或者目录中的数据拷贝到表在HDFS上的目录中。使用本地数据时，**<path>** 必须是绝对路径；使用HDFS上的数据时，**<path>** 可以是绝对路径或者或者相对路径（相对 `/user/<user_name>`）。**<path>** 不能有子目录。
- ② **[OVERWRITE]** 选项会将目标表或者分区已有的内容会被导入的文件覆盖。不加 **[OVERWRITE]** 选项则导入的文件不覆盖已有文件，但是如果目标表或者分区中存在文件和被导入的文件重名，那么原先的文件会被新文件覆盖。
- ③ 可以向表中指定的分区导入数据。



当 **<path>** 是HDFS路径时，执行该操作的用户必须是 **<path>** 的owner，同时hive用户必须要对 **<path>** 有读写权限。

例 22. 将本地数据导入一张表

```
LOAD DATA LOCAL INPATH '/tmp/user_info_table.txt' INTO TABLE user_info;
```

例 23. 将本地数据导入一张表下的分区

```
LOAD DATA LOCAL INPATH '/tmp/user_info_table_A.txt' INTO TABLE partition_user_info PARTITION
(acc_level = 'A');
```

例 24. 将HDFS上的数据导入一张表

```
LOAD DATA INPATH '/user/root/test' INTO TABLE user_info2;
```

3.4.2. 向表插入数据

INSERT ... SELECT 用于将查询结果插入表中。**INSERT ... VALUES** 用于将值直接插入表中，但是只有Hyperbase、ES表、ORC事务表支持这种用法。

关于INSERT语法本节只介绍**INSERT ... SELECT**，对于**INSERT ... VALUES**我们只在介绍支持该语法的表类型时进行介绍。



注意插入表中的数据和表的列数相同，并且的对应列数据类型一致。

3.4.2.1. 单次插入

语法

```
INSERT (OVERWRITE|INTO) TABLE <table_name> ①
[PARTITION (<partition_key>=<partition_value> ...) [IF NOT EXISTS]] ②
SELECT <select_statement> FROM <source>; ③
```

- ① **INSERT OVERWRITE** 覆盖表中原数据； **INSERT INTO** 不覆盖表中原数据。
- ② 可以将数据插入分区表的指定分区中，要了解更多关于分区表数据插入的内容，请参考“分区表”一章。
- ③ **<select_statement>** 中的列需要和 **<table_name>** 的列数量一致且数据类型一一对应。



不支持对ORC事务表执行**INSERT OVERWRITE**。

例 25. 向表中插入数据并覆盖原数据

```
INSERT OVERWRITE TABLE user_info2 SELECT * FROM user_info;
```

3.4.2.2. 多次插入

使用一个数据源可以将多个查询结果插入不同表中。

语法

```
FROM <source>
  INSERT (OVERWRITE|INTO) TABLE <table_name1> [PARTITION (<partkey>=<val>...) [IF NOT EXISTS]]
  SELECT <select_statement1>
  [INSERT (OVERWRITE|INTO) TABLE <tablename2> [PARTITION (<partkey>=<val>...) [IF NOT EXISTS]]
  SELECT <select_statement2>
  [INSERT (OVERWRITE|INTO) TABLE <tablename3> [PARTITION (<partkey>=<val>...) [IF NOT EXISTS]]
  SELECT <select_statement3>
  ;
```



除了最后一个 **<select_statement>** 之外，如果 **<select_statement>** 以一个列名结尾，那么该列必须有化名，否则下一个 **INSERT** 关键字将被当做这个 **<select_statement>** 最后一列的列化名处理，导致语义错误。

例 26. 多次插入

```
FROM user_info
INSERT INTO TABLE user_name SELECT name n
INSERT OVERWRITE TABLE user_name_num SELECT name, acc_num ac
INSERT INTO TABLE user_name_level SELECT name, acc_level;
```

3.4.3. 向文件系统中插入数据

Inceptor支持将查询结果写入文件系统中指定的目录下。



写入文件系统使用INSERT语句，但是不同于将数据INSERT进表中有INTO和OVERWRITE两种选项，将数据写入文件系统必须INSERT OVERWRITE。

3.4.3.1. 写入本地文件系统

语法

```
INSERT OVERWRITE LOCAL DIRECTORY <directory> [ROW FORMAT <row_format>] [STORED AS <file_format>]
  SELECT ... FROM ...
row_format
  : DELIMITED [FIELDS TERMINATED BY <char> [ESCAPED BY <char>]] [COLLECTION ITEMS TERMINATED BY
<char>] [MAP KEYS TERMINATED BY <char>] [LINES TERMINATED BY <char>]
```

说明

- 注意，这个语句将查询结果写入一个 目录，而不是文件，写入的结果可能是多个文件。
- 写入本地文件系统要加上 **LOCAL** 关键字；
- ROW FORMAT**指定文件的行格式，不指定使用默认值；
- STORED AS**指定文件格式，不指定则使用默认值；

例1:

将user_info中的内容写入本地/tmp/user_info/目录下。

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/user_info/' SELECT * FROM user_info;
```

在本地文件系统中查看写入结果，如下（您如果自己执行这个SQL，/tmp/user_info/目录下的文件数量可能和这里不同）：

```
# ls /tmp/user_info/
000000_0  000001_0
```

打开其中一个文件：

```
马** ^A6513065^A115591^A14*****7^A960*****41^A20110101^AA
祝** ^A6670192^A205239^A23*****8^A737*****71^A20100101^AC
华* ^A5224133^A531547^A42*****1^A326*****07^A20080214^AB
魏** ^A3912384^A841242^A52*****4^A685*****48^A20091202^AA
宁** ^A4580952^A986634^A42*****7^A977*****76^A20081031^AD
```

可以看出不指定ROW FORMAT和STORED AS，文件格式和行格式都使用了默认值。

例2：

将user_info中的内容写入本地/tmp/user_info2/目录下，同时指定ROW FORMAT。

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/user_info2/' ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'
SELECT * FROM user_info;
```

在本地文件系统中查看写入结果，如下（您如果自己执行这个SQL，/tmp/user_info2/目录下的文件数量可能和这里不同）：

```
# ls /tmp/user_info2/
000000_0 000001_0
```

打开其中一个文件：

```
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
```

可以看出，文件中使用了我们指定的行格式。

3.4.3.2. 写入HDFS

语法

```
INSERT OVERWRITE DIRECTORY <directory> [ROW FORMAT <row_format>]
  SELECT ... FROM ...
row_format
  : DELIMITED [FIELDS TERMINATED BY <char> [ESCAPED BY <char>]] [COLLECTION ITEMS TERMINATED BY
<char>]
  [MAP KEYS TERMINATED BY <char>] [LINES TERMINATED BY <char>]
```

说明

- 注意，这个语句将查询结果写入一个 目录，而不是文件，写入的结果可能是多个文件。
- ROW FORMAT指定文件的行格式，不指定使用默认值；

例1：

将user_info中的内容写入HDFS上的/tmp/user_info/目录下。

```
INSERT OVERWRITE DIRECTORY '/tmp/user_info/' SELECT * FROM user_info;
```

现在看一下HDFS上的对应目录（您如果自己执行这个SQL，/tmp/user_info/目录下的文件数量可能和这里不同）：

```
# hdfs dfs -ls /tmp/user_info/
Found 2 items
-rwxrwxrwx 3 yarn hadoop      354 2015-12-15 19:58 /tmp/user_info/000000_0
-rwxrwxrwx 3 yarn hadoop      353 2015-12-15 19:58 /tmp/user_info/000001_0
```

将其中一个文件get到本地

```
# hdfs dfs -get /tmp/user_info/000000_0
```

打开查看，我们可以看到

```
马** ^A6513065^A115591^A14*****7^A960*****41^A20110101^AA
祝** ^A6670192^A205239^A23*****8^A737*****71^A20100101^AC
华* ^A5224133^A531547^A42*****1^A326*****07^A20080214^AB
魏** ^A3912384^A841242^A52*****4^A685*****48^A20091202^AA
宁** ^A4580952^A986634^A42*****7^A977*****76^A20081031^AD
```

可以看出不指定ROW FORMAT时行格式使用了默认值。

例2：

将user_info中的内容写入HDFS上的/tmp/user_info2/目录下，同时指定ROW FORMAT。

```
INSERT OVERWRITE DIRECTORY '/tmp/user_info2/' ROW FORMAT DELIMITED FIELDS TERMINATED BY '|' SELECT *
FROM user_info;
```

现在看一下HDFS上的对应目录（您如果自己执行这个SQL，/tmp/user_info2/目录下的文件数量可能和这里不同）：

```
# hdfs dfs -ls /tmp/user_info2/
Found 2 items
-rwxrwxrwx 3 yarn hadoop 354 2015-12-15 20:14 /tmp/user_info2/000000_0
-rwxrwxrwx 3 yarn hadoop 353 2015-12-15 20:14 /tmp/user_info2/000001_0
```

将其中一个文件get到本地

```
# hdfs dfs -get /tmp/user_info/000000_0
```

打开查看，我们可以看到

```
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
```

可以看出，文件中使用了我们指定的行格式。

3.4.3.3. Multi-Insert

在Inceptor中还可以用一个SQL语句将从一个数据源中检索出来的多个结果插入不同文件和表中。

语法

```
FROM <source>
INSERT [INTO|OVERWRITE] TABLE <table_name> SELECT ...
INSERT OVERWRITE [LOCAL] DIRECTORY <directory_name> SELECT ...
...
```

说明

- 如果加上LOCAL选项，Inceptor会将数据写入本地。
- 可以将用INSERT OVERWRITE写入HDFS上目录和本地目录在同一次查询中混用。
- 用INSERT OVERWRITE将数据写入HDFS目录是从Inceptor提取大量数据的最好方法。Inceptor可以在一个map-reduce作业中将数据并行写入HDFS。

例

```
FROM user_info
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/user_info4/' SELECT name a
INSERT OVERWRITE DIRECTORY '/tmp/user_info5/' SELECT password b
INSERT INTO TABLE new_user SELECT name c
INSERT OVERWRITE TABLE user_name SELECT name d;
```

3.4.4. 查询语句

查询语句 **SELECT** 开头，可以通过添加多种从句从Inceptor中的表中获得信息。

最常使用的数据查询语句的语法如下：

语法

```
SELECT [ALL | DISTINCT] select_expression, select_expression, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[CLUSTER BY col_list
 | [DISTRIBUTE BY col_list] [SORT BY col_list]
]
[LIMIT (M,)N
 | [OFFSET M ROWS FETCH NEXT | FIRST] N ROWS ONLY];
```

说明

- SELECT 子句指定查询结果中出现的列。SELECT子句中可以包含的select_expression有：
- 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。
- WHERE关键词提供对查询结果的过滤条件。
- 表名和列名不区分大小写。
- 结构中的末尾部分提供了分页功能，允许跳过起始的M行，从第M+1行开始返回N行查询结果。Inceptor支持MySQL和DB2中的分页语法。

举例

下面查询获取表t1中所有行和所有列中的信息。

```
[${host}] SELECT * FROM t1;
1
1
2
3
3
3
```

也可以查询表中指定的列。下面查询获取表t1中的列a中的信息：

```
[${host}] SELECT a FROM t1;
1
1
2
3
3
3
```

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。要指定数据库，可以用“.”来表示表和数据库的从属关系，比如

```
SELECT * FROM database_name.table_name;
```

或者还可以在查询前使用USE语句来指定当前数据库：

```
USE database_name;
SELECT query_specifications;
USE default;
```

ALL和DISTINCT关键字

ALL和DISTINCT告诉Inceptor是否要返回查询中出现的重复行。选择DISTINCT选项Inceptor会不返回重复行；选择ALL则返回所有行，无论重复与否。不选默认为选择ALL。

举例

```
[${host}] SELECT a FROM t1;
1
1
2
3
3
3

[${host}] SELECT DISTINCT a FROM t1;
2
1
3
```

3.4.4.1. 过滤：WHERE和HAVING

查询中的一个常见操作是使用过滤来获得对用户有用的信息。过滤的条件可以通过WHERE子句或者HAVING子句指明。WHERE子句和HAVING子句的区别在于，一次查询中如果有WHERE子句，Inceptor会先执行WHERE子句的过滤条件再执行SELECT语句，而查询中如果用到了HAVING子句，Inceptor会先执行SELECT语句，再执行HAVING子句。换句话说，WHERE在SELECT之前过滤，而HAVING在SELECT之后过滤。

3.4.4.1.1. WHERE子句

举例

- 单个过滤条件：下例从用户信息表中挑出所有在2010年之前注册的用户：

```
[$host] SELECT * FROM partition_user_info WHERE reg_date < 2010000;
魏** 3912384 841242 52*****4 685*****48 20091202 A
宁** 4580952 986634 42*****7 977*****76 20081031 D
华* 5224133 531547 42*****1 326*****07 20080214 B
```

- 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。下例从用户信息表中挑出在2012年之前注册的A级或者B级用户：

```
[$host] SELECT * FROM partition_user_info WHERE reg_date < 20120000 AND acc_level = 'A' OR
acc_level = 'B';
马** 6513065 115591 14*****7 960*****41 20110101 A
魏** 3912384 841242 52*****4 685*****48 20091202 A
华* 5224133 531547 42*****1 326*****07 20080214 B
```

- 用BETWEEN来表示带有范围的过滤条件：现在我们想要挑出在2010年和2012年之间注册的用户。我们可以使用AND语句来连接两个不等式：

```
SELECT * FROM partition_user_info WHERE reg_date > 20100000 AND reg_date < 20120000;
```

也可以用BETWEEN来表示范围的区间：

```
[$host] SELECT * FROM partition_user_info WHERE reg_date BETWEEN 20100000 AND 20120000;
李** 2755506 015859 31*****9 424*****37 20110916 D
马** 6513065 115591 14*****7 960*****41 20110101 A
祝** 6670192 205239 23*****8 737*****71 20100101 C
潘** 6600641 990590 51*****5 484*****08 20110430 C
```

- 用IN来表示带有从属关系的过滤条件：

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：

举例

下例两个语句都可用来查询账户级别为A, B, C的账户持有人，两个语句的效果相同：

```
[$host] SELECT name FROM user_info WHERE acc_level = 'A' OR acc_level = 'B' OR acc_level = 'C';
马**
祝**
华*
魏**
邱*
潘**
管**

[$host] SELECT name FROM user_info WHERE acc_level IN ('A', 'B', 'C');
马**
祝**
华*
魏**
邱*
潘**
管**
```

使用NOT IN可以表示过滤条件为不属于某个集合。下例查询账户级别不是A, B或C的账户持有人姓名：

```
[${host}] SELECT name FROM user_info WHERE acc_level NOT IN ('A', 'B', 'C');
宁**
李*
李**
```

- 在WHERE子句中嵌套子查询:

举例

下面语句通过对user_info和transactions两张表的查询查看在2014年第二季度有过交易的用户:

```
[${host}] SELECT name FROM user_info WHERE acc_num IN (SELECT acc_num FROM transactions WHERE
trans_time BETWEEN 20140401000000 AND 20140630235959);
马**
魏**
邱*
```

3.4.4.1.2. HAVING子句

举例

如果过滤条件受带GROUP BY的查询结果影响，那么就不能用WHERE子句来过滤，而要用HAVING子句。

```
[${host}] SELECT acc_num, max(price*amount)
> FROM transactions
> WHERE trans_time<'20140630235959'
> GROUP BY acc_num
> HAVING max(price*amount)>5000;
6513065 12866.0
5224133 8023.999999999999
3912384 6843.000000000001
0700735 5250.0
```

3.4.4.2. ORDER BY, SORT BY, DISTRIBUTE BY和CLUSTER BY

3.4.4.2.1. ORDER BY

语法

```
SELECT select_statement ORDER BY col_name [ASC|DESC] [,col_name_2 [ASC|DESC],...]
```

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。

举例

sequence是一张只有一列（列名column1），列中有1-10十个整数，按column1分3个桶的表。

```
[${host}] SELECT * FROM sequence ORDER BY column1;
1
2
3
4
5
6
7
8
9
10
```

3.4.4.2.2. SORT BY

语法

```
SELECT select_statement SORT BY col_name [ASC|DESC] [,col_name_2 [ASC|DESC],...]
```

说明

- SORT BY在每个reducer之内排序，来达到 局部有序(*local ordering*)。
- 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。因为在数据非常多的情况下，ORDER BY可能导致超时，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。

举例

```
[${host}] SELECT * FROM sequence SORT BY column1;
3
6
9
1
4
7
10
2
5
8
```

SORT BY让sequence的各个reducer内都有序。

3.4.4.2.3. DISTRIBUTE BY与SORT BY合用

SORT BY让Inceptor在各个reducer中排序。我们可以先用DISTRIBUTE BY为输出结果人工分桶，之后使用SORT BY可以保证数据在这些桶中有序。

语法

```
SELECT select_statement
DISTRIBUTE BY col_name_1
SORT BY col_name_2 [ASC|DESC] [,col_name_3 [ASC|DESC],...]
```

所有DISTRIBUTE BY列的列值相同的记录会被放进同一个reducer中。

3.4.4.2.4. CLUSTER BY

如果DISTRIBUTE BY和SORT BY子句中的列是同一个而且SORT BY顺序选择是升序，那么DISTRIBUTE BY col SORT BY col可以用CLUSTER BY col来代替，效果完全相同。比如以下两段代码的结果完全相同。

```
[${host}] SELECT * FROM user_info DISTRIBUTE BY acc_level SORT BY acc_level;
[${host}] SELECT * FROM user_info CLUSTER BY acc_level;
```

3.4.4.3. GROUP BY

GROUP BY子句将查询结果按列值并组，也就是指定列列值相同的将被并入同组。GROUP BY常常与聚合函数合

用——

将查询结果按列值并组，然后再对每组分别使用聚合函数。更多关于聚合函数的内容请参考“函数和运算符”下的“聚合函数”章节。

注意： 使用GROUP BY时，SELECT语句和GROUP BY子句所包含的列必须相同。

语法

```
SELECT select_expression, select_expression, ...
GROUP BY groupby_expression [, groupby_expression, ...]
```

说明

- select_expression可以是列，表达式，也可以是聚合函数。
- groupby_expression可以是列，也可以是表达式

3.4.4.3.1. 单列GROUP BY

单列GROUP BY就是GROUP BY 子句中只有一列。

举例 我们可以用对transactions表用GROUP BY查看各个账户进行交易的次数：

```
SELECT acc_num, count(trans_id) FROM transactions GROUP BY acc_num;
2394923 1
2755506 1
6513065 6
3912384 4
0700735 3
6600641 1
5224133 2
6670192 2
```

但是，如果我们想要查看各个账户所有交易的流水号(trans_id)，我们不能使用GROUP BY：

无效代码

```
SELECT acc_num, trans_id FROM transactions GROUP BY acc_num;
[Hive Error]: Query returned non-zero code: 10, cause: FAILED: Error in semantic analysis: Line 1:16
Expression not in GROUP BY key 'trans_id'
```

上面这段代码是无效的，因为SELECT语句中包含了列trans_id，但是trans_id不包含在GROUP BY子句中。我们看到6513065这个账号进行了6次交易，上面查询要Inceptor为6513065返回一个trans_id值，但是Inceptor不知道应该返回哪一个trans_id，故Inceptor会报错。

举例

下面代码查看进行了买(b)和卖(s)操作的账户个数。

```
SELECT trans_type, count(DISTINCT acc_num) FROM transactions GROUP BY trans_type;
b      4
s      8
```

有4个账户进行了买操作，有8个账户进行了卖操作。注意，count(DISTINCT acc_num) 中的DISTINCT确保在统计账户个数时，一个acc_num仅统计一次。

注意： 一次查询可以使用多个聚合函数，但是聚合函数的参数中的DISTINCT列必须相同。

3.4.4.3.2. GROUP BY (number)

- 语法:

```
SELECT select_expression1, select_expression2, ... GROUP BY groupby_expression [, groupby_expression, ...]
```

Inceptor中支持group by 1 (第一列), group by 2 (第二列) 这类用法, 其中数字1是指select_expression的位置。

- 利用 group by 语句查看test1中Num列也就是第一列每一个数字对应的字母的个数。

```
select * from test1;
+-----+-----+
| num | letter |
+-----+-----+
| 1   | a    |
| 1   | b    |
| 1   | c    |
| 1   | d    |
| 2   | a    |
| 2   | b    |
| 2   | c    |
| 3   | e    |
| 3   | f    |
| 4   | g    |
| 5   | h    |
| 6   | i    |
+-----+
12 rows selected (2.188 seconds)
```

- 利用 group by 语句查看test1中Num列也就是第一列每一个数字对应的字母的个数

```
select num, count(letter) from test1 group by num;
+-----+-----+
| num | _c1 |
+-----+-----+
| 1   | 4   |
| 2   | 3   |
| 3   | 2   |
| 4   | 1   |
| 5   | 1   |
| 6   | 1   |
+-----+
6 rows selected (3.726 seconds)
```

- 我们可以把上例中的 'num' 换成' 1', 来查看第一列每一个数字对应的字母的个数。

```

select * from test1;
+---+-----+
| num | letter |
+---+-----+
| 1   | a    |
| 1   | b    |
| 1   | c    |
| 1   | d    |
| 2   | a    |
| 2   | b    |
| 2   | c    |
| 3   | e    |
| 3   | f    |
| 4   | g    |
| 5   | h    |
| 6   | i    |
+---+-----+
12 rows selected (2.137 seconds)
select num,count(letter) from test1 group by 1;
+---+-----+
| num | _c1  |
+---+-----+
| 1   | 4    |
| 2   | 3    |
| 3   | 2    |
| 4   | 1    |
| 5   | 1    |
| 6   | 1    |
+---+-----+
6 rows selected (3.254 seconds)

```

- 利用 group by 语句查看test1中letter列也就是第二列每一个字母对应的数字的个数。

```

select letter,count(num) from test1 group by letter;
+---+-----+
| letter | _c1  |
+---+-----+
| i     | 1    |
| f     | 1    |
| e     | 1    |
| h     | 1    |
| g     | 1    |
| b     | 2    |
| a     | 2    |
| d     | 1    |
| c     | 2    |
+---+-----+
9 rows selected (3.756 seconds)

```

- 我们可以把上例中的 'letter' 换成' 2'，来查看第二列每一个字母对应的数字的个数。

```

select count(num), letter from test1 group by 2;
+---+-----+
| _c0 | letter |
+---+-----+
| 2   | b    |
| 2   | a    |
| 1   | d    |
| 2   | c    |
| 1   | i    |
| 1   | f    |
| 1   | e    |
| 1   | h    |
| 1   | g    |
+---+-----+
9 rows selected (2.188 seconds)

```

3.4.4.3.3. 多列GROUP BY

多列GROUP BY就是GROUP BY子句中有不止一列。

举例

下例查询各账户进行的买和卖交易各有多少笔：

```
SELECT acc_num, trans_type, count(trans_id) FROM transactions GROUP BY acc_num, trans_type;
2755506 s      1
6513065 s      2
6513065 b      4
0700735 s      2
2394923 s      1
3912384 b      2
3912384 s      2
5224133 s      1
5224133 b      1
0700735 b      1
6670192 s      2
6600641 s      1
```

3.4.4.3.4. 用表达式GROUP BY

举例

下例查询2014年各个月份中发生的交易数量。tdh_todate是Inceptor自带的函数，可以用来提取trans_time中的月份：

```
SELECT tdh_todate(trans_time, 'yyyyMMddHHmmss', 'MM'), count(trans_id) FROM transactions GROUP BY
tdh_todate(trans_time, 'yyyyMMddHHmmss', 'MM');
09      1
07      1
06      2
05      2
04      2
03      4
02      3
01      1
10      1
08      1
11      1
12      1
```

3.4.4.3.5. 在GROUP BY子句中过滤：HAVING子句

如果过滤条件受带GROUP BY的查询结果影响，那么就不能用WHERE子句来过滤，而要用HAVING子句

语法

```
SELECT select_expression, select_expression, ...
FROM table_name
GROUP BY groupby_expression, group_expression
HAVING having_expression
```

举例

```
[$host]  SELECT acc_num, max(price*amount)
        > FROM transactions
        > WHERE trans_time<'20140630235959'
        > GROUP BY acc_num
        > HAVING max(price*amount)>5000;
6513065 12866.0
5224133 8023.999999999999
3912384 6843.0000000001
0700735 5250.0
```

上例返回所有2014年上半年各账户超过5000元的最大单笔交易额。有两个过滤条件：WHERE子句中的过滤条件和查询结果无关，而HAVING子句中的过滤要在查询结束后才执行。

注意：在WHERE子句中不能有聚合函数，因为Inceptor在执行GROUP BY子句之前就会执行WHERE子句。

HAVING子句中可以含有聚合函数:

```
[$host]  SELECT acc_num, max(price*amount)
          > FROM transactions
          > WHERE trans_time<'20140630235959'
          > GROUP BY acc_num
          > HAVING min(price*amount)>2000;
3912384 6843.0000000001
5224133 8023.99999999999
6670192 4124.0
```

上面例子查询2014年上半年单笔交易额至少有2000元的各个账户的最大单笔交易额。

3.4.4.4. GROUP BY 扩展: ROLLUP/CUBE/GROUPING SETS

Inceptor提供三种GROUP BY扩展: ROLLUP, CUBE和GROUPING SETS。这些扩展在进行聚合计算时提供不同方式的小结。本章节我们用一张库存表作为例子。库存表名为inventory, 记录了一家水果店的库存情况:

```
SELECT * FROM inventory;
apple  NJ      A      500.0
apple  NJ      B      800.0
orange NJ      A      300.0
orange NJ      C      100.0
pear   NJ      B      700.0
apple  SH      A      1000.0
apple  SH      B      500.0
pear   SH      A      400.0
pear   SH      C      500.0
apple  HZ      B      200.0
```

该表的列名分别为type, store, grade和amount。

3.4.4.4.1. ROLLUP

语法

```
SELECT col1, col2, ..., aggregate_func(expression) FROM table_name GROUP BY ROLLUP(col1, col2, ...)
```

说明

ROLLUP生成聚合行、超聚合行和总计行。举例来说，下面的代码

```
SELECT a, b, c, SUM(expression) FROM table GROUP BY ROLLUP(a, b, c);
```

会为 (a, b, c)、(a, b) 和 (a) 值的每个唯一组合生成一个带有小计的行。还将计算一个总计行。详细地说，以上代码会计算以下四条查询并将结果一并输出：

```
SELECT a, b, c, sum(expression) FROM table GROUP BY a, b, c;    // (a, b, c) 组合小计
SELECT a, b, NULL, sum(expression) FROM table GROUP BY a, b;    // (a, b) 组合小计
SELECT a, NULL, NULL, sum(expression) FROM table GROUP BY a;    // (a) 组合小计
SELECT NULL, NULL, NULL, sum(expression) FROM table;           // 总计
```

举例

```

SELECT type, store, grade, sum(amount) FROM inventory GROUP BY ROLLUP(type, store, grade) ORDER BY
type, store, grade;
NULL    NULL    NULL    5000
apple   NULL    NULL    3000
apple   HZ     NULL    200
apple   HZ     B      200
apple   NJ     NULL    1300
apple   NJ     A      500
apple   NJ     B      800
apple   SH     NULL    1500
apple   SH     A      1000
apple   SH     B      500
orange  NULL    NULL    400
orange  NJ     NULL    400
orange  NJ     A      300
orange  NJ     C      100
pear    NULL    NULL    1600
pear    NJ     NULL    700
pear    NJ     B      700
pear    SH     NULL    900
pear    SH     A      400
pear    SH     C      500

```

3.4.4.4.2. CUBE

语法

```
SELECT col1, col2, ..., aggregate_func(expression) FROM table_name GROUP BY CUBE(col1, col2, ...)
```

说明

Cube生成聚合行、超聚合行、交叉表格行和总计行。举例来说，

```
SELECT a, b, c, SUM (expression) FROM table GROUP BY CUBE (a,b,c);
```

会为 (a, b, c)、(a, b)、(a, c)、(b, c)、(a)、(b) 和 (c) 值的每个唯一组合生成一个带有小计的行，还会生成一个总计行。详细来说，以上代码会进行以下八条计算，并将结果一并输出：

```

SELECT a, b, c, sum(expression) FROM table GROUP BY a, b, c; // (a, b, c) 组合小计
SELECT a, b, NULL, sum(expression) FROM table GROUP BY a, b; // (a, b) 组合小计
SELECT a, NULL, c, sum(expression) FROM table GROUP BY a, c; // (a, c) 组合小计
SELECT NULL, b, c, sum(expression) FROM table GROUP BY b, c; // (b, c) 组合小计
SELECT a, NULL, NULL, sum(expression) FROM table GROUP BY a; // (a) 组合小计
SELECT NULL, b, NULL, sum(expression) FROM table GROUP BY b; // (b) 组合小计
SELECT NULL, NULL, c, sum(expression) FROM table GROUP BY c; // (c) 组合小计
SELECT NULL, NULL, NULL, sum(expression) FROM table;           // 总计

```

举例

```

SELECT type, store, grade, sum(amount) FROM inventory GROUP BY CUBE(type, store, grade) ORDER BY
type, store, grade;
NULL    NULL    NULL      5000
NULL    NULL    A        2200
NULL    NULL    B        2200
NULL    NULL    C        600
NULL    HZ     NULL      200
NULL    HZ     B        200
NULL    NJ     NULL      2400
NULL    NJ     A        800
NULL    NJ     B        1500
NULL    NJ     C        100
NULL    SH     NULL      2400
NULL    SH     A        1400
NULL    SH     B        500
NULL    SH     C        500
apple   NULL    NULL      3000
apple   NULL    A        1500
apple   NULL    B        1500
apple   HZ     NULL      200
apple   HZ     B        200
apple   NJ     NULL      1300
apple   NJ     A        500
apple   NJ     B        800
apple   SH     NULL      1500
apple   SH     A        1000
apple   SH     B        500
orange  NULL    NULL      400
orange  NULL    A        300
orange  NULL    C        100
orange  NJ     NULL      400
orange  NJ     A        300
orange  NJ     C        100
pear    NULL    NULL      1600
pear    NULL    A        400
pear    NULL    B        700
pear    NULL    C        500
pear    NJ     NULL      700
pear    NJ     B        700
pear    SH     NULL      900
pear    SH     A        400
pear    SH     C        500

```

3.4.4.4.3. GROUPING SETS

语法

```
SELECT col1, col2, ..., aggregate_func(expression) FROM table_name GROUP BY GROUPING SETS(col1,
col2, ...)
```

说明

GROUPING SETS生成交叉表格行。举例来说，

```
SELECT a, b, c, sum(expression) FROM table GROUP BY GROUPING SETS(a,b,c);
```

会为 (a)、(b) 和 (c) 值的每个唯一组合生成一个带有小计的行。详细地说，以上代码会进行以下三条计算，并将结果一并输出：

```

SELECT a, NULL, NULL, sum(expression) FROM table GROUP BY a; // (a) 组合小计
SELECT NULL, b, NULL, sum(expression) FROM table GROUP BY b; // (b) 组合小计
SELECT NULL, NULL, c, sum(expression) FROM table GROUP BY c; // (c) 组合小计

```

举例

```

SELECT type, store, grade, sum(amount) FROM inventory GROUP BY GROUPING SETS(type, store, grade)
ORDER BY type, store, grade;
NULL    NULL      A    2200
NULL    NULL      B    2200
NULL    NULL      C    600
NULL    HZ        NULL   200
NULL    NJ        NULL   2400
NULL    SH        NULL   2400
apple   NULL      NULL   3000
orange  NULL      NULL   400
pear    NULL      NULL   1600

```

3.4.4.5. 多表查询: JOIN

语法

```

SELECT select_expression, select_expression, ...
FROM table_reference JOIN table_reference JOIN table_reference, ... [ON join_condition]

```

说明

join_condition 连接条件可以是等值条件（这种连接叫做等价连接），也可以是不等值条件。

注意:不支持JOIN条件中带有OR条件,也不支持JOIN条件中使用BETWEEN/IN。

3.4.4.5.1. 笛卡尔连接: Cartesian Join

语法

```

SELECT select_expression, select_expression, ...
FROM table_name JOIN table_name ON join_condition
JOIN table_name ON join_condition
...

```

join_condition为连接条件，如果该条件恒成立（比如 $1=1$ ），该连接就是笛卡尔连接。连接的结果是由被连接表中所有记录组成的有序数组(ordered tuples)。所以，笛卡尔连接输出的记录条数等于被连接表的记录条数的乘积。

举例

```

[$host]  SELECT * FROM join_demo1;
1
2
3
[$host]  SELECT * FROM join_demo2;
a
b
c
d
[$host]  SELECT * FROM join_demo1 JOIN join_demo2 ON 1=1;
1      a
1      b
1      c
1      d
2      a
2      b
2      c
2      d
3      a
3      b
3      c
3      d
[$host]  SELECT count(*) FROM join_demo1 JOIN join_demo2 ON 1=1;
12

```

这样的表连接会产生大量的数据，而且并不经常具有意义，所以很少出现在实际应用中。大多数这样的表连接产生于没有加JOIN条件的错误。所以为了代码的清晰，如果真的需要进行笛卡尔积连接，最好使用专门的关键词CROSS JOIN：

语法

```
SELECT table1.col1, table2.col2, ...
FROM table1 CROSS JOIN table2
```

3.4.4.5.2. 内连接：INNER JOIN

语法

```
SELECT select_expression, select_expression, ...
FROM table_reference1 (JOIN|INNER JOIN) table_reference2
ON (table_reference1.column_1 = table_reference2.column_a)
```

内连接只显示参与连接的表中有匹配的记录。JOIN和INNER JOIN在这里用法一样。在一次查询中可以连接两个以上的表：

```
SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key2)
```

举例

假设我们有两张表，student_info和course_info。student_info包含了五个学生的名字和他们分别选修的课程编号：

```
SELECT * FROM student_info;
赵一 SCU100
钱二 MUS101
孙三 NULL
李四 COM101
周五 BIO101
```

course_info包含了五门选修课的课程编号和课程名称：

```
SELECT * FROM course_info;
BIO101 生物基础
COM101 面向对象编程
MUS101 西方古典音乐赏析
SCU100 雕塑
MAT100 微积分
```

现在我们想要通过将两张表在课程编号匹配值处连接来查看学生选择的选修课课名：

```
[$host] SELECT student_info.stu_name, course_info.course_name FROM student_info JOIN course_info ON
(student_info.course_id = course_info.course_id);
赵一 雕塑
钱二 西方古典音乐赏析
李四 面向对象编程
周五 生物基础
```

举例

使用JOIN可以从用户信息表和交易信息表中查看哪些用户进行了哪些交易：

```
[$host]  SELECT user_info.name, transactions.trans_id FROM user_info JOIN transactions ON
(user_info.acc_num = transactions.acc_num);
马** 943197522
马** 499506900
马** 289018112
马** 895916502
马** 404905188
马** 213859826
祝** 900192386
祝** 952639648
华* 594819547
华* 817414815
魏** 929634984
魏** 209441379
魏** 719753265
魏** 597565609
邱* 648230055
邱* 975639131
邱* 459590958
潘** 952110653
李** 991691937
管** 162742112
```

3.4.4.5.3. 外连接：OUTER JOIN

内连接会将被连接的两张表中互相没有匹配值的纪录忽略。如果想要在连接结果中看到没有匹配值的记录，则应该使用外连接。外连接又分为左外连接(left outer join)、右外连接(right outer join)和全外连接(full outer join)。

语法

```
SELECT select_expression, select_expression, ...
FROM table_reference (LEFT|RIGHT|FULL) OUTER JOIN table reference
```

左外连接：LEFT OUTER JOIN

在之前的学生选修课程的例子中，孙三的记录因为他暂时没有选修课而被忽略了。如果我们想要在查询结果中能够看到暂时没有选修课的学生，我们可以用左外连接 命令Inceptor将左边表中（这里的student_info）的所有记录返回：

```
[$host]  SELECT student_info.stu_name, course_info.course_name FROM student_info LEFT OUTER JOIN
course_info ON (student_info.course_id = course_info.course_id);
赵一    雕塑
钱二    西方古典音乐赏析
孙三    NULL
李四    面向对象编程
周五    生物基础
```

右外连接：RIGHT OUTER JOIN

右外连接和左外连接相似，但是会将右边表（这里的course_info）中的所有记录返回：

```
[$host]  SELECT student_info.stu_name, course_info.course_name FROM student_info RIGHT OUTER JOIN
course_info ON (student_info.course_id = course_info.course_id);
周五    生物基础
李四    面向对象编程
钱二    西方古典音乐赏析
赵一    雕塑
NULL    微积分
```

全外连接：FULL OUTER JOIN

如果想要将两张表中的所有记录返回，可以用全外连接：

```
[${host}] SELECT student_info.stu_name, course_info.course_name FROM student_info FULL OUTER JOIN
course_info ON (student_info.course_id = course_info.course_id);
周五    生物基础
赵一    雕塑
NULL    微积分
李四    面向对象编程
钱二    西方古典音乐赏析
孙三    NULL
```

3.4.4.5.4. 隐式连接：Implicit JOIN

语法

```
SELECT table1.col1, table2.col2, ...
FROM table1, table2
WHERE table1.col3 = table2.col1;
```

隐式JOIN的命令中不含有JOIN…ON…关键词，而是通过WHERE子句作为连接条件将两张表连接。

例子

```
[${host}] SELECT student_info.stu_name, course_info.course_name FROM student_info, course_info WHERE
student_info.course_id = course_info.course_id;
赵一    雕塑
钱二    西方古典音乐赏析
李四    面向对象编程
周五    生物基础
```

在连接和过滤都有的语句中，使用隐式JOIN会让人容易混淆哪些子句是用来连接的而哪些子句是用来过滤的。比如：

```
[${host}] SELECT name, trans_id FROM user_info, transactions WHERE user_info.acc_num =
transactions.acc_num AND trans_time < 20140630235959;
马** 943197522
马** 499506900
马** 404905188
马** 213859826
祝** 952639648
华* 817414815
魏** 929634984
魏** 209441379
魏** 719753265
魏** 597565609
邱* 648230055
邱* 975639131
邱* 459598958
潘** 952110653
```

3.4.4.5.5. 自然连接：NATURAL JOIN

语法

```
SELECT table1.col1, table1.col2, ... table2.col1, table2.col3, ...
FROM table1 NATURAL JOIN table2;
```

使用NATURAL JOIN，用户可以不需要明确写出JOIN条件。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。

举例

下例查询所有用户和他们进行过交易的流水号。

```
[$host] SELECT name, trans_id from user_info NATURAL JOIN transactions;
邱* 648230055
邱* 975639131
邱* 459590958
李** 991691937
管** 162742112
马** 943197522
马** 499506900
马** 289018112
马** 895916502
马** 404905188
马** 213859826
华* 594819547
华* 817414815
魏** 929634984
魏** 209441379
魏** 719753265
魏** 597565609
潘** 952110653
祝** 900192386
祝** 952639648
```

3.4.4.5.6. 多表连接

可以在一次查询中用多个JOIN子句连接多张表。

语法

```
SELECT select_expression, select_expression, ...
FROM table_reference [(RIGHT|LEFT|FULL) OUTER] JOIN table_reference ON (join_condition)
[(RIGHT|LEFT|FULL) OUTER] JOIN table_reference ON (join_condition) ...
```

说明

每一个JOIN子句都可以是不同的连接（内连接，左/右/全外连接，左半连接等等）。

举例

这里有一张记录了教师和他们所教课程编码的表：

```
[$host] SELECT * FROM lecturer_info;
石** MAT101
李* COM101
樊* MUS101
曹** SCU100
钱* BIO101
```

我们通过连接学生信息表，课程信息表和教师信息表来查看所有学生正在上的课和课程的教师

```
[$host] SELECT stu_name, course_name, lecturer_name FROM student_info LEFT OUTER JOIN course_info
ON (student_info.course_id = course_info.course_id)LEFT OUTER JOIN lecturer_info ON
(course_info.course_id = lecturer_info.course_id);
赵一 雕塑 曹**
钱二 西方古典音乐赏析 樊*
孙三 NULL NULL
李四 面向对象编程 李*
周五 生物基础 钱*
```

3.4.4.5.7. 重复连接

有时候同一张表在一次查询中需要和多张其他表连接。假设我们有三张表table1, table2和table3。其

中table1要和table2和table3各连接一次。这种情况下，用户需要给table1在两次连接中起两个不同的化名来让Inceptor能够分辨在各子句中table1的角色：

```
SELECT t1.col1, tb1.col2, t2.col1, t3.col1, ...
FROM table1 t1 JOIN table2 t2
  ON t1.col1 = t2.col1
  INNER JOIN table3 t3
  ON t2.col2 = t3.col1
  INNER JOIN table1 tb1
  ON t3.col1 = tb1.col2;
```

3.4.4.5.8. 表的自连接

一张表也可以和自己连接，此时需要给表取两个不同的化名来让Inceptor能够分辨在各子句中表的角色。

语法

```
SELECT select_expression, select_expression, ...
FROM table_reference alias1 JOIN table_reference alias2 ON (join_condition)
```

举例

我们用一个包含了员工信息的表来作为例子。表中含有员工工号，员工姓名，员工的上级工号和入职时间：

```
[$host]  SELECT * FROM employee_info;
10817  张**  94832  20140509
13049  王**  94832  20140616
58290  许**  12083  20120901
94832  戴**  12083  20111212
12083  郑*   56412  20110916
56412  吴**  NULL    20110916
```

我们用这张表和自己连接来查询员工和他们上级的姓名：

```
[$host]  SELECT e.employee_name, sup.employee_name FROM employee_info e LEFT OUTER JOIN
employee_info sup ON (e.sup_id = sup.employee_id);
张** 戴**
王** 戴**
许** 郑*
戴** 郑*
郑* 吴**
吴** NULL
```

3.4.4.5.9. 左半连接和左半反连接

左半连接用来查看左表中符合JOIN条件的记录。左半反连接用来查看左表不符合JOIN条件的记录。左半连接和左半反连接都*只显示左表中的记录*。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。而左半反连接可以通过在LEFT ...NOT IN/EXISTS中嵌套子查询来实现。关于WHERE... IN/EXISTS和WHERE NOT IN /EXISTS的更多说明和例子可以在子查询章节中找到。

左半连接

语法：LEFT SEMI JOIN

```
SELECT select_expression, select_expression, ...
FROM table_reference_1
LEFT SEMI JOIN table_reference_2 ON (join_condition)
```

举例

首先，我们有两张表test1和test2。test1和test2的列名都为num和letter:

```
[$host]  SELECT * FROM test1;
1      a
1      b
1      c
2      a
3      e
[$host]  SELECT * FROM test2;
1      f
1      g
2      i
```

下例我们分别用LEFT SEMI JOIN和在WHERE IN/EXISTS中嵌套子查询执行相同任务：查看test1中的num列和test2.num匹配的所有记录。

使用LEFT SEMI JOIN

```
[$host]  SELECT t1.num, t1.letter FROM test1 t1 LEFT SEMI JOIN test2 t2 ON t1.num = t2.num;
1      a
1      b
1      c
2      a
```

使用在WHERE...IN中嵌套子查询

```
[$host]  SELECT t1.num, t1.letter FROM test1 t1 WHERE t1.num IN (SELECT t2.num FROM test2 t2);
1      a
1      b
1      c
2      a
```

使用在WHERE EXISTS中嵌套子查询

```
[$host]  SELECT t1.num, t1.letter FROM test1 t1 WHERE EXISTS (SELECT 1 FROM test2 t2 WHERE t2.num =
t1.num);
1      a
1      b
1      c
2      a
```

举例

Inceptor不支持LEFT ANTI SEMI JOIN这样的语法，但是左半连接的效果可以又WHERE...NOT IN和WHERE NOT EXISTS来达到。下例我们分别用WHERE...NOT IN, WHERE NOT EXISTS 来执行左半反连接：查看test1中num列值在test2.num中*没有*匹配值的记录：

在WHERE...NOT IN中嵌套子查询

```
[$host]  SELECT t1.num, t1.letter FROM test1 t1 WHERE t1.num NOT IN (SELECT t2.num FROM test2 t2);
3      e
```

在WHERE NOT EXISTS中嵌套子查询

```
[$host]  SELECT t1.num, t1.letter FROM test1 t1 WHERE NOT EXISTS (SELECT 1 FROM test2 t2 WHERE
t1.num = t2.num);
3      e
```

3.4.4.5.10. 不等价连接

到现在为止，我们只用到了等价连接(equi-join)，也就是JOIN条件是一个等式。我们也可以将表不等价连接(non-equi join)。

语法

```
SELECT select_expression, select_expression, ...
FROM table_reference1 JOIN table reference_2
ON equi_join_condition
WHERE non_equi_join_condition
```

从语法中我们看到，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。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。

举例

我们用一个自不等价连接作为例子。假设公司想要在员工中间组织一次下棋比赛，每对员工之间都要进行一场比赛，我们想要生成包含所有员工对的数据：

```
[${host}]  SELECT e1.employee_name, e2.employee_name FROM employee_info e1 JOIN employee_info e2 ON 1
= 1 WHERE e1.employee_name <> e2.employee_name;
张** 王**
张** 许**
张** 戴**
张** 郑*
张** 吴**
王** 张**
王** 许**
王** 戴**
王** 郑*
王** 吴**
许** 张**
许** 王**
许** 戴**
许** 郑*
许** 吴**
戴** 张**
戴** 王**
戴** 许**
戴** 郑*
戴** 吴**
郑* 张**
郑* 王**
郑* 许**
郑* 戴**
郑* 吴**
吴** 张**
吴** 王**
吴** 许**
吴** 戴**
吴** 郑*
```

这里join条件是`e1.employee_name <> e2.employee_name`，因为一个人不能和自己比赛。这条指令会导致每个员工对都会重复一次，所以我们可以修改连接条件为`e1.employee_name > e2.employee_name`:

```
[${host}] SELECT e1.employee_name, e2.employee_name FROM employee_info e1 JOIN employee_info e2 ON 1
= 1 WHERE e1.employee_name > e2.employee_name;
张** 吴**
王** 张**
王** 戴**
王** 吴**
许** 张**
许** 王**
许** 戴**
许** 吴**
戴** 张**
戴** 吴**
郑* 张**
郑* 王**
郑* 许**
郑* 戴**
郑* 吴**
```

3.4.4.5.11. MAP JOIN

如果两张被连接的表中有一张比较小（100MB以下），那么可以通过MAP JOIN来提高执行速度。MAP JOIN会将小表放入内存中，在map阶段直接拿另一张表的数据和内存中表数据做匹配，由于省去了shuffle，速度会比较快。

语法

```
SELECT /*+ MAPJOIN(b) */ select_expression, select_expression, ...
FROM table_reference JOIN table_reference ON join_condition
```

Inceptor已经有了自动MAP JOIN的功能，就是在有一张表在100MB一下时，Inceptor会自动执行MAP JOIN。所以用户可以无需特别指明使用MAP JOIN。如果在参与JOIN的表都较大时却指明使用MAP JOIN，可能会导致内存溢出。

3.4.4.6. 子查询

子查询是嵌套在查询语句中的查询语句。子查询根据是否和包含它的父查询的结果相关分为非同步子查询和同步子查询。Inceptor高度支持子查询的各种嵌套：非同步子查询可以在FROM, WHERE, SELECT和HAVING子句中嵌套。同步子查询可以在WHERE和SELECT中嵌套，而不能在HAVING和FROM子句中嵌套。

3.4.4.6.1. 非关联子查询：Non-Correlated Subqueries

非同步子查询内容和包含它的父查询结果不相关。当子查询和父查询不相关，Inceptor会在执行父查询之前先执行完成子查询。

在WHERE子句中嵌套

举例：单行单列的子查询结果

下例查询股票交易平台用户信息表中第一个注册的账户的账户持有人，账户号码，持有人身份证件，账户级别和注册时间。

```
[${host}] SELECT name, acc_num, citizen_id, acc_level, reg_date FROM user_info WHERE reg_date =
(SELECT min(reg_date) FROM user_info);
华* 5224133 42***** B 20080214
```

注意，这里子查询SELECT min(reg_date) FROM user_info的结果是单列单行的，所以WHERE子句中的过滤条件可以是一个等式。如果子查询的结果有多列或者多行，过滤条件需要有变化。

举例：单行多列的子查询结果

IN运算符 当子查询结果有不止一条记录，要用IN来表示查询结果须是子查询结果集合中的元素：

下例查询员工信息表中所有有下级员工的员工名字：

```
[$host] SELECT employee_name FROM employee_info WHERE employee_id IN (SELECT sup_id FROM
employee_info);
戴**
郑*
吴**
```

NOT IN运算符

我们也可以查询没有下级员工的员工名字：

```
[$host] SELECT employee_name FROM employee_info WHERE employee_id NOT IN (SELECT sup_id FROM
employee_info);
张**
王**
许**
```

在FROM子句中嵌套

举例

下例查询所有进行过交易的账户持有人名字：

```
[$host] SELECT DISTINCT name FROM (SELECT name FROM user_info JOIN transactions ON
user_info.acc_num = transactions.acc_num);
邱*
管**
李**
潘**
祝**
魏**
马**
华*
```

下例查询所有个人平均交易额大于所有平均交易额的用户名字

```
[$host] SELECT name FROM user_info JOIN (SELECT transactions.acc_num, avg(price*amount) avg_trans
FROM transactions GROUP BY transactions.acc_num) temp ON user_info.acc_num = temp.acc_num WHERE
avg_trans > (SELECT avg(price*amount) FROM transactions);
祝**
华*
李**
管**
```

在SELECT子句中嵌套

举例

下例查看各用户的个人平均交易额和所有交易的平均交易额的差：

```
[${host}] SELECT acc_num, avg(price*amount) - (SELECT avg(price*amount) FROM transactions) FROM
transactions GROUP BY acc_num;
2394923 1126.3500000000004
2755506 4797.35
6513065 -20.31666666666606
3912384 -685.649999999996
0700735 -1364.649999999996
6600641 -4062.649999999996
5224133 1577.350000000004
6670192 971.350000000004
```

在HAVING子句中嵌套

举例

下例查询最大一笔交易的执行账户和交易额。

```
[${host}] SELECT acc_num, max(price*amount) FROM transactions GROUP BY acc_num HAVING
max(price*amount) = (SELECT max(price*amount) FROM transactions);
6513065 12866.0
```

3.4.4.6.2. 关联子查询(Correlated Subquery)

同步子查询的内容和父查询相关。Inceptor会对每条在父查询中出现的记录执行一次子查询。

注意:关联子查询中的关联条件不支持OR,也不支持仅包含非等值比较。

在WHERE子句中嵌套

Inceptor支持的WHERE子句嵌套需要满足以下要求:

1. WHERE子句中必须包含至少一条等值关系,如果执行业务没有客观要求等值关联,请用户手动添加条件“1=1”。这是为了避免资源被贪婪占用导致枯竭,以保证系统的稳定性。
2. 主查询和子查询之间必须用标量比较运算符连接(包括‘>’、‘<’、‘=’、‘<>’、‘>=’、‘<=’)。
3. 要求子查询的结果必须是一行一列的返回,即标量。
4. 子查询中允许有等值与非等值条件。

举例

下例查询了总共进行过3笔交易的账户持有人姓名和账户号码。

```
[${host}] SELECT user_info.name, user_info.acc_num FROM user_info WHERE 3=(SELECT COUNT(*) FROM
transactions WHERE user_info.acc_num = transactions.acc_num);
邱* 0700735
```

举例

```
[${host}] SELECT COUNT(*) FROM TABLEA A, TABLEB B
WHERE ((1=1)) AND ((A.salary > (SELECT (SUM(csA.age) - 114)
FROM TABLEA csA, TABLEB csB
WHERE (A.age = csA.age) AND
((csB.salary / -9) >= (A.salary * -79)))));
```

EXISTS和NOT EXISTS

语法

```
SELECT select_expression, select_expression, ...
FROM table_reference
WHERE (EXISTS|NOT EXISTS) (subquery)
```

在WHERE中嵌套子查询时经常会用到EXISTS和NOT EXISTS。当我们只关心子查询有记录返回，而不关心子查询返回的记录内容和记录条数时，我们就可以用WHERE EXISTS。WHERE EXISTS用来查看子查询中的关系是否成立并且返回使得子查询中关系成立的记录（也就是过滤掉使得子查询中的关系不成立的记录）。比如，假设查询买过单股价格在100元以内的股票的用户，语句应为：

举例

```
[$host]  SELECT user_info.name, user_info.acc_num FROM user_info WHERE EXISTS (SELECT 1 FROM
transactions WHERE user_info.acc_num = transactions.acc_num AND price < 100);
邱*    0700735
```

事实上，我们建议如果WHERE子句需要满足某种关系（大于、等于、小于、不等于，等等），尽量使用WHERE EXISTS并在子查询中表达关系，而不是通过比较子查询的结果和别的量来表达关系。WHERE NOT EXISTS则用来查看子查询中的关系是否成立并且返回使得子查询中关系不成立的记录（也就是过滤掉使得子查询中的关系成立的记录）。

举例

下例查询了所有没有进行交易的账户持有人姓名和账户号码

```
[$host]  SELECT user_info.name, user_info.acc_num FROM user_info WHERE NOT EXISTS (SELECT 1 FROM
transactions WHERE user_info.acc_num = transactions.acc_num);
宁**  4580952
李*    8725869
```

这里，WHERE NOT EXISTS子句中嵌套的子查询返回的是一个常数，这充分体现了EXISTS和NOT EXISTS仅关心子查询 是否 返回结果，而不关心返回的结果 是什么。

在SELECT子句中嵌套

举例

下例返回所有账户的持有人姓名，账户号码和账户平均交易额：

```
[$host]  SELECT user_info.name, user_info.acc_num, (SELECT avg(price*amount) FROM transactions WHERE
user_info.acc_num = transactions.acc_num) FROM user_info;
马**  6513065 4958.333333333333
祝**  6670192 5950.0
华*    5224133 6556.0
魏**  3912384 4293.0
宁**  4580952 NULL
邱*    0700735 3614.0
李*    8725869 NULL
潘**  6600641 916.0
李**  2755506 9776.0
管**  2394923 6105.0
```

3.4.4.6.3. 子查询的多层嵌套

Inceptor支持多层嵌套，如：

```

SELECT select_expression FROM (
    SELECT select_expression FROM (
        SELECT select_expression FROM...
        ...
    )
)

```

举例

```

[$host]  SELECT name FROM (
            SELECT name, acc_num FROM (
                SELECT name, acc_num, password FROM (
                    SELECT name, acc_num, password, bank_acc FROM user_info
                )
            );
马**
祝**
华*
魏**
宁**
邱*
李*
潘**
李**
管**

```

3.4.4.7. WITH…AS

当子查询嵌套层数较多，语句会难以阅读和维护。我们可以通过用WITH…AS定义公共表表达式（CTE）来简化查询，提高可阅读性和易维护性。

语法

```
WITH cte_name AS (select_statement) sql_containing_cte_name
```

说明

- cte_name是公共表表达式的名字
- select_statement是一个完整的SELECT语句
- sql_containing_cte_name是包含了刚刚定义的公共表表达式的SQL语句，注意，定义了一个CTE以后必须马上使用，否则这个CTE定义将失效。

举例

```

[$host]  WITH nv AS (SELECT name FROM user_info JOIN transactions ON user_info.acc_num =
transactions.acc_num)
SELECT DISTINCT name FROM nv;
邱*
管**
李**
潘**
祝**
魏**
马**
华*

```

在WITH…AS连续定义多个CTE

用户可以通过一次WITH定义多个CTE，中间用逗号连接：

```
WITH cte_1 AS (select_statement_1),
  cte_2 AS (select_statement_2),
  cte_3 AS (select_statement_3),
  ...
sql_containing_all_defined_ctes
```

说明

- 所有定义的cte必须都马上使用。
- 后定义的cte可以引用已经定义的cte。

举例

下例查询所有个人平均交易额大于所有平均交易额的用户名字

```
[$host]  SELECT name FROM user_info JOIN (SELECT transactions.acc_num, avg(price*amount) avg_trans
FROM transactions GROUP BY transactions.acc_num) personal_avg ON user_info.acc_num =
personal_avg.acc_num WHERE avg_trans > (SELECT avg(price*amount) FROM transactions);
祝**
华*
李**
管**
```

用WITH...AS改写:

```
[$host]  WITH
          personal_avg AS (SELECT transactions.acc_num, avg(price*amount)avg_trans FROM
transactions GROUP BY transactions.acc_num),
          namelist AS (SELECT name FROM user_info JOIN personal_avg ON user_info.acc_num =
personal_avg.acc_num WHERE avg_trans > (SELECT avg(price*amount)FROM transactions))
          SELECT * FROM namelist;
祝**
华*
李**
管**
```

目前Inceptor只支持公共表达式在 SELECT、INSERT SELECT、CREATE TABLE AS SELECT 这三种语句中出现。

注意，根据公共表达式的使用语法，WITH-AS短语必须紧接SELECT语句，不允许在其中穿插其他关键字。

当INSERT、CREATE TABLE AS访问WITH-AS时，必须由SELECT进行传递。

- 公共表达式只能作为数据源出现在INSERT语句中，下面是INSERT+公共表达式的语法和示例：

INSERT+公共表达式语法

```
INSERT INTO <dst>
WITH cte_name AS (select_statement) sql_containing_cte_name
```

公共表达式作为INSERT的数据源示例

```
INSERT INTO transtbl
WITH cte AS (
    SELECT class, MAX(date_of_birth)
    FROM student_info
    GROUP BY class
)
SELECT * from cte;
```



注意，不允许把WITH-AS同访问它的SELECT语句拆分开，即如下是错误写法：

```
WITH cte AS (
    SELECT class, MAX(date_of_birth)
    FROM student_info
    GROUP BY class
)
INSERT INTO transtbl
SELECT * from cte;
```

- 下面是CREATE TABLE AS+公共表达式的语法和示例：

CREATE TABLE AS+公共表达式语法

```
CREATE TABLE <table> AS
WITH cte_name AS (select_statement) sql_containing_cte_name
```

CREATE TABLE AS+公共表达式的应用示例

```
CREATE TABLE ctetbl AS
WITH cte AS (
    SELECT name, class
    FROM student_info
    WHERE date_of_birth > '19940809'
)
SELECT * FROM cte;
```

3.4.4.8. 化名

在和Inceptor交互时，我们可以为表和列取化名（alias）。表和列的化名在一下几种情况会用到：

- 表名和列名有时会很长，查询时反复使用这些名字会很麻烦。
- 当一张表在查询中重复使用并且扮演不同角色时，Inceptor也需要帮助辨别这些角色。

- 查询需要利用子查询的结果。

这时候，用户可以给表或列起临时的化名，这并不是给表或者列重命名，化名只能在当前查询中使用。

3.4.4.8.1. 表化名(Table Alias)

语法

```
SELECT select_expression, select_expression, ...
FROM table_reference [AS] table_alias
```

说明

- table_reference 可以是表，视图或者子查询。
- 可以选择使用AS连接table_reference和table_alias，来清楚地标注出使用了化名

举例

employee_info表在查询中重复使用并且扮演不同角色：

```
[$host]  SELECT e1.employee_name, e2.employee_name FROM employee_info e1 JOIN employee_info AS e2 ON
1 = 1 WHERE e1.employee_name > e2.employee_name;
张** 吴**
王** 张**
王** 戴**
王** 吴**
许** 张**
许** 王**
许** 戴**
许** 吴**
戴** 张**
戴** 吴**
郑* 张**
郑* 王**
郑* 许**
郑* 戴**
郑* 吴**
```

举例

给予查询起化名：

```
[$host]  SELECT temp.name FROM (SELECT name, acc_level FROM user_info) temp;
马**
祝**
华*
魏**
宁**
邱*
李*
潘**
李**
管**
```

3.4.4.8.2. 列化名(Column Alias)

语法

```
SELECT select_expression [AS] col_alias, select_expression [AS] col_alias, ...
FROM table_reference
```

说明

- select_expression可以是列名，或者子查询。
- 可以选择使用AS连接select_expression和col_alias，来清楚地标注出使用了化名

举例：给列起化名

```
[$host]  SELECT n FROM (SELECT name as n FROM user_info);
马**
祝**
华*
魏**
宁**
邱*
李*
潘**
李**
管**
```

举例：给子查询起列化名：

```
[$host]  SELECT acc_num, (avg(price*amount) - (SELECT avg(price*amount) FROM transactions)) AS diff
FROM transactions GROUP BY acc_num;
6600641 -4062.649999999996
6513065 -20.3166666666606
5224133 1577.350000000004
6670192 971.350000000004
3912384 -685.649999999996
0700735 -1364.649999999996
2394923 1126.350000000004
2755506 4797.35
```

3.4.4.9. 集合运算：UNION/INTERCEPT/EXCEPT

Inceptor提供三种方法来对SELECT语句结果进行集合运算：并集(UNION)、交集(INTERSECT)和减去(EXCEPT)。而每个集合运算都有两种选择，带有ALL和不带有ALL。所以我们一共有以下六个集合运算：

- UNION/UNION ALL
- INTERSECT/INTERSECT ALL
- EXCEPT/EXCEPT ALL

集合运算的语法相似：

语法

```
select_expression_1 (UNION|INTERSECT|EXCEPT) select_expression_2
select_expression_1 (UNION ALL|INTERSECT ALL|EXCEPT ALL) select_expression_2
```

可以将多个SELECT语句的结果合成单个结果。如果说JOIN是将表左右连接，那么集合运算是以一定条件将表首尾相接，所以其中每一个SELECT语句返回的列数必须相同，列值类型必须对应，对应列列名也必须相同。否则Inceptor会抛出一个schema error。当SELECT语句结果列名不同时，要使用列化名。一次查询中我们也可以进行多次集合运算，比如：

```
select_expression_1 UNION select_expression_2 UNION select_expression_3 INTERSECT
select_expression_4 ...
```

下面，我们将用test_a和test_b两张表来演示如何进行集合运算：

```

[$host] DESCRIBE test_a;
a_int          int      None
a_double       double   None
a_string        string   None

[$host] SELECT * FROM test_a;
1    1.01    a
1    1.02    b
1    1.02    b
2    2.02    c
2    2.04    c
3    3.03    d

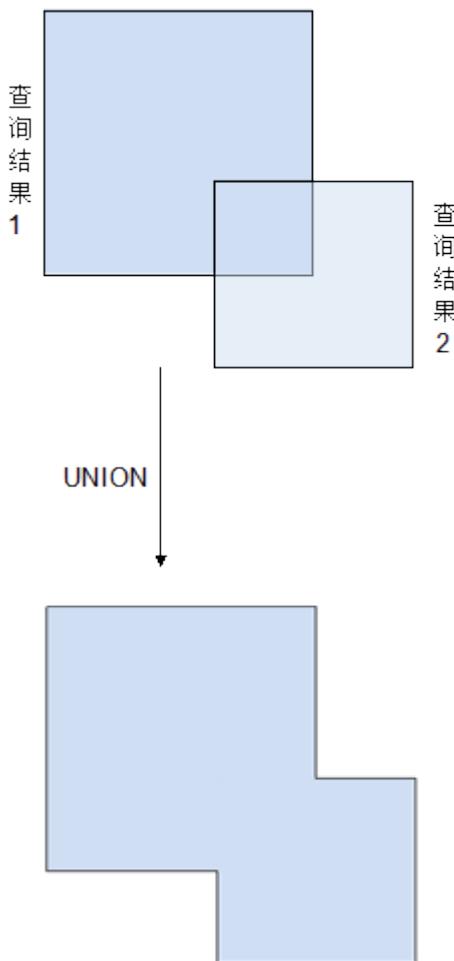
[$host] DESCRIBE test_b;
b_int          int      None
b_string1      string   None
b_double       double   None
b_string2      string   None

[$host] SELECT * FROM test_b;
1    b      1.02    f
1    b      1.02    h
1    c      1.04    g

```

3.4.4.9.1. UNION 和 UNION ALL

UNION和UNION ALL形成查询结果的并集。



注意:

UNION对结果去重，而UNION ALL不去重。

语法

```
select_statement_1 UNION select_statement_2;
select_statement_1 UNION ALL select_statement_2;
```

如果我们要对UNION ALL的结果进行进一步的处理，我们可以像下面这样将整个UNION子句内嵌进一个FROM子句：

```
SELECT *
FROM (
  select_statement
  UNION ALL
  select_statement
) unionResult
```

举例

取test_a和test_b两表的并集：

- 采用去重UNION:

```
[$host] SELECT * FROM( SELECT a_double AS value FROM test_a UNION SELECT b_double AS value FROM test_b )ORDER BY value;
```

返回结果如下：

```
1.01
1.02
1.04
2.02
2.04
3.03
```

- 采用不去重的UNION ALL:

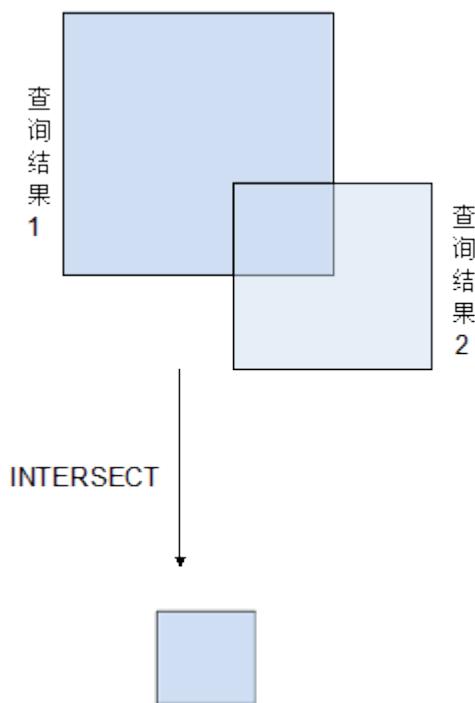
```
[$host] SELECT * FROM( SELECT a_double AS value FROM test_a UNION ALL SELECT b_double AS value FROM test_b )ORDER BY value;
```

返回结果如下：

```
1.01
1.02
1.02
1.02
1.02
1.04
2.02
2.04
3.03
```

3.4.4.9.2. INTERSECT 和 INTERSECT ALL

INTERSECT和INTERSECT ALL返回两个查询结果的交集。



INTERSECT对返回的结果去重，而INTERSECT ALL不去重。它们的语法相同：

语法

```
select_expression_1 INTERSECT select_expression_2
select_expression_1 INTERSECT ALL select_expression_2
```

举例

INTERSECT 去重：

```
[$host]  SELECT a_int ii, a_double id, a_string is FROM test_a INTERSECT SELECT b_int ii, b_double
id, b_string1 is FROM test_b;
1      1.02      b
```

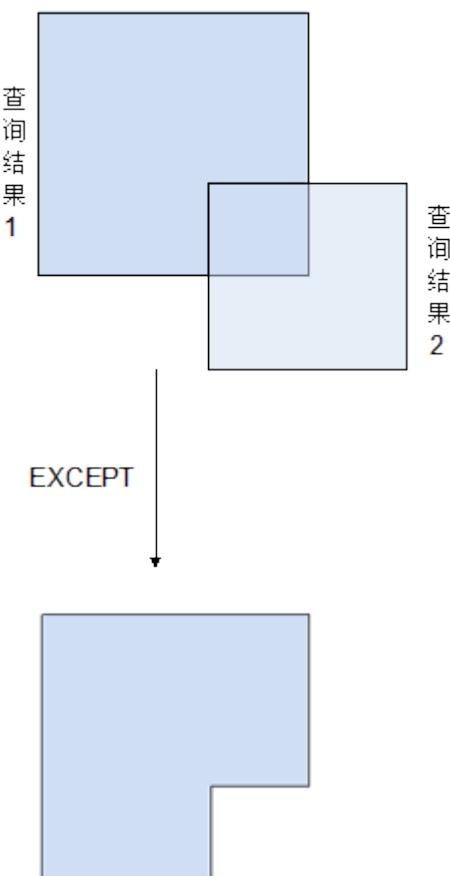
INTERSECT ALL不去重：

```
[$host]  SELECT a_int ii, a_double id, a_string is FROM test_a INTERSECT ALL SELECT b_int ii,
b_double id, b_string1 is FROM test_b;
1      1.02      b
1      1.02      b
```

3.4.4.9.3. EXCEPT 和 EXCEPT ALL

EXCEPT和EXCEPT ALL做集合减法。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，然后不去重的A中剩下的记录。 EXCEPT和EXCEPT ALL的语法相同：

```
select_statement_1 EXCEPT select_statement_2
select_statement_1 EXCEPT ALL select_statement_2
```



举例

EXCEPT

```
[${host}]  SELECT a_int ei FROM test_a EXCEPT SELECT b_int ei FROM test_b;
2
3
```

EXCEPT ALL

```
[${host}]  SELECT a_int ei FROM test_a EXCEPT ALL SELECT b_int ei FROM test_b;
2
2
3
```

3.4.4.10. 分页

Inceptor通过分页功能支持用户指定从结果集的哪一行开始返回结果，以及需要返回的行数。在结果集记录数很大的情况下，分页能帮助提高工作效率。

3.4.4.10.1. 本节使用表

```
item_info:
+-----+
| id      | date        | owner      |
+-----+
| 625392835 | 2015-03-04 | 'zhang'   |
| 614381023 | 2015-01-08 | 'yang'     |
| 650915021 | 2015-06-15 | 'li'       |
| 613280522 | 2015-07-08 | 'sun'      |
| 682305152 | 2015-08-09 | 'fang'     |
| 612438091 | 2015-09-23 | 'yu'       |
| 608274565 | 2015-02-28 | 'cai'      |
| 601749371 | 2015-10-11 | 'deng'     |
| 692341124 | 2015-05-30 | 'chang'   |
| 610843513 | 2015-06-09 | 'yang'     |
+-----+
```

3.4.4.10.2. 语法

Inceptor允许两种分页语法，是为了提供对MySQL、DB2以及Oracle方言的支持。

1. MySQL方言

```
SELECT column1, ..., expression1, ... FROM table_name
ORDER BY ...
LIMIT OFFSET_LINES, ROWCOUNT
```

- OFFSET_LINES表示从结果集中的第OFFSET_LINES+1行开始返回结果。
- ROWCOUNT表示需要返回的结果行数。
- 如果从第一行开始返回，应缺省OFFSET_LINES；或者将OFFSET_LINES设为零。

举例

```
SELECT * FROM item_info WHERE owner <> 'li' ORDER BY id LIMIT 5;
601749371  '2015-10-11'      'deng'
608274565  '2015-02-28'      'cai'
610843513  '2015-06-09'      'yang'
612438091  '2015-09-23'      'yu'
613280522  '2015-07-08'      'sun'

SELECT * FROM item_info WHERE owner <> 'li' ORDER BY id LIMIT 0,5;
601749371  '2015-10-11'      'deng'
608274565  '2015-02-28'      'cai'
610843513  '2015-06-09'      'yang'
612438091  '2015-09-23'      'yu'
613280522  '2015-07-08'      'sun'

SELECT * FROM item_info WHERE owner <> 'li' ORDER BY id LIMIT 5,10; ①
614381023  '2015-01-08'      'yang'
625392835  '2015-03-04'      'zhang'
682305152  '2015-08-09'      'fang'
692341124  '2015-05-30'      'chang'

SELECT * FROM item_info WHERE owner <> 'li' ORDER BY id LIMIT 10,5; ②
```

- ① 可返回的记录行数小于等于ROWCOUNT时，将返回至结果集末尾。
 ② 若指定的起始位置超出结果集结尾，将没有任何返回。

2. DB2方言

```
SELECT column1, ..., expression1, ... FROM table_name
OFFSET <OFFSET_LINES> [ROW|ROWS]
FETCH first/next <ROWCOUNT> [ROW|ROWS] only
```

- OFFSET_LINES表示从结果集中的第OFFSET_LINES+1行开始返回结果，当不缺省OFFSET子句时FETCH部分应该写“FETCH next <ROWCOUNT> ROWS only”。

- ROWCOUNT表示需要返回的结果行数。
- 如果从第一行开始返回，应缺省OFFSET部分，FETCH部分写“first ... ROWS only”；或者也可以在OFFSET部分将OFFSET_LINES设为零。

举例

```

SELECT * FROM item_info WHERE owner <> 'li' ORDER BY id FETCH first 5 ROWS only;
601749371  '2015-10-11'      'deng'
608274565  '2015-02-28'      'cai'
610843513  '2015-06-09'      'yang'
612438091  '2015-09-23'      'yu'
613280522  '2015-07-08'      'sun'

SELECT * FROM item_info WHERE owner <> 'li' ORDER BY id OFFSET 0 ROWS FETCH next 5 ROWS only;
601749371  '2015-10-11'      'deng'
608274565  '2015-02-28'      'cai'
610843513  '2015-06-09'      'yang'
612438091  '2015-09-23'      'yu'
613280522  '2015-07-08'      'sun'

SELECT * FROM item_info WHERE owner <> 'li' ORDER BY id OFFSET 5 ROWS FETCH next 10 ROWS
only; ①
614381023  '2015-01-08'      'yang'
625392835  '2015-03-04'      'zhang'
682305152  '2015-08-09'      'fang'
692341124  '2015-05-30'      'chang'

SELECT * FROM item_info WHERE owner <> 'li' ORDER BY id OFFSET 10 ROWS FETCH next 5 ROWS
only; ②

```

① 可返回的记录行数小于等于ROWCOUNT时，将返回至结果集末尾。

② 若指定的起始位置超出结果集结尾，将没有任何返回。

3. Oracle方言

处于Oracle方言模式时，以上两种语法都支持。

3.4.4.11. SQL层次化查询

3.4.4.11.1. 简单介绍

除了常用数据查询语言形式，Inceptor还支持层次化查询。层次化查询是树型结构的查询，是SQL中经常用到的功能之一，通常由根节点、父节点、子节点、以及叶节点组成。Inceptor支持的层次化查询的语法如下：

```

SELECT [hq_level], column1, column2, ... expression1, expression2, ...
FROM table_name
[WHERE where_clause]
[[START WITH start_condition] [CONNECT BY PRIOR prior_condition]];

```

层次化查询提供了以下四个特殊操作：

1. hq_level：用于表示节点的层级，为伪列。注意：hq_level 中间是两个下横线，而且不要将hq误写为hp。
2. START WITH：指定根记录条件，满足START WITH给定条件的记录为根记录。
3. CONNECT BY：用于指定父记录和子记录之间的关联关系。
4. 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。

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，父节点在上子节点

在下。读者可以通过层次化查询的用法示例理解层次化查询的作用。

3.4.4.11.2. 本节使用的表

no_loop_employee表:

employee_id	name	manager_id	salary	department_id
1	kochhr		5000	1
2	greenberg	1	4500	2
3	faviet	1	4400	3
4	chen	2	4000	2
5	sciarra	2	4000	2
6	urman	3	3800	3
7	popp	2	4000	2
8	whlen	6	3700	3

loop_employee表:

employee_id	name	manager_id
1	kochhr	
2	greenberg	1
3	faviet	8
4	chen	2
5	sciarra	2
6	urman	3
7	popp	2
8	whlen	6

3.4.4.11.3. 用例

- 执行层次化查询，根据CONNECT BY条件按级查找员工的employee_id、name及其manager_id:

```
[${host}] SELECT employee_id, name, manager_id FROM no_loop_employee START WITH name = 'kochhr'
CONNECT BY name != 'faviet' AND PRIOR employee_id = manager_id ORDER BY employee_id;
```

返回结果如下:

1	kochhr	NULL
2	greenberg	1
4	chen	2
5	sciarra	2
7	popp	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:

```
[${host}] 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;
```

返回结果如下:

1	kochhr	NULL	1
2	greenberg	1	1
2	greenberg	1	2
3	faviet	1	1
3	faviet	1	2
4	chen	2	1
4	chen	2	2
4	chen	2	3
5	sciarra	2	1
5	sciarra	2	2
5	sciarra	2	3
6	urman	3	1
6	urman	3	2
6	urman	3	3
7	popp	2	1
7	popp	2	2
7	popp	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。

```
[$host] SELECT a.employee_id, a.name 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;
```

返回结果如下：

```
6    urman
8    whlen
```

- 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。

```
[$host] SELECT employee_id, name, manager_id FROM loop_employee START WITH employee_id = 3 CONNECT BY PRIOR employee_id = manager_id;
```

该语句在执行时会报错。

Inceptor目前暂时不支持如下场景的层次化查询：



1. 不支持这些操作：SYS_CONNECT_BY_PATH、CONNECT_BY_ISLEAF、CONNECT_BY_ROOT、NOCYCLE、ORDER SIBLINGS BY。
2. 不允许把子查询作为START WITH的条件。
3. 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；只能为非关联子查询。
4. 不允许在WHERE IN非关联子查询中访问hq_level。

3.4.4.12. Sequence语法

一个Sequence（序列）可服务于多个用户、多个表、多个字段，用于为有唯一性要求的字段（如学号、身份证号、电话等）生成连续不等的值。不同用户使用相同序列向同一字段创建值，可保证值的唯一性，帮助提高应用程序的一致性。

Sequence通过指定序列起始数值、终止数值以及数值改变方式定义。应用时，通过以下两个伪列实现对Sequence的访问：

1. CURRVAL：返回序列当前值
2. NEXTVAL：返回序列由数值改变方式决定的下一个值，即自增值

3.4.4.12.1. Sequence的创建、修改、删除、查询

- Create Sequence

```
CREATE SEQUENCE [db_name.]sequence_name
  [INCREMENT BY <integer>]          ①
  [START WITH <integer>]            ②
  [MAXVALUE <integer> | NOMAXVALUE] ③
  [MINVALUE <integer> | NOMINVALUE] ④
  [CYCLE | NOCYCLE]                ⑤
  [CACHE <integer> | NOCACHE]       ⑥
  [ORDER | NOORDER]                ⑦
```

① db_name可选，是Sequence从属的数据库名称；sequence_name为序列名称。

② INCREMENT BY 用于设置产生数值的间隔。<integer>可以是正或负64位整型，但不能为0，正表

示递增，负代表递减。其绝对值必须小于等于MAXVALUE-MINVALUE。默认为1。

③ **START WITH** 用于设置序列起始值。对于递增序列，默认值为MINVALUE；对于递减序列，默认为MAXVALUE。

④ **MAXVALUE** 用于设定序列最大值，MAXVALUE必须大于等于起始值且大于MINVALUE；若设置为NOMAXVALUE，则表示当序列递增时，最大值为 $2^{63}-1$ ，当序列递减时，最大值为-1。默认为NOMAXVALUE。

⑤ **MINVALUE** 用于设定序列最小值，MINVALUE必须小于等于起始值且小于MAXVALUE；若设置为NOMINVALUE，则表示当序列递增时，最小值为1，当序列递减时，最小值为 -2^{63} 。默认为NOMINVALUE。

⑥ 若选为 **CYCLE**，当序列递增（或递减）到最大值（最小值）时，重置为最小值（最大值）；若选为 **NOCYCLE**，当序列递增（或递减）到最大值（最小值）时抛出异常。默认为NOCYCLE。

⑦ **CACHE** 用于加快分布式序列的获取速度，每个节点每次获取该值大小个数目的序列值作为自己的缓存，系统直接从缓存获取序列值。假设设置为CACHE 10，每当缓存中的序列值被提取完，系统将自动缓存序列中接下来的10个值。CACHE的最小值为2，最大值为 **CEIL(MAXVALUE-MINVALUE)/ABS(INCREMENT)**。 **NOCACHE** 表示不采用缓存，在NOCACHE状态下每次读取序列都会产生网络通信，因而影响执行速率，所以建议采用缓存。默认缓存20个序列值。

⑧ 目前只是添加了语法支持，并无实际意义。

1. 创建序列时，如果缺省所有可选项，即语句如下：

```
CREATE SEQUENCE sequence_name;
```

则默认创建一个自增值为1，其起始值为1，无边界限制的NOCYCLE序列，缓存值20。相当于构造了这样的序列创建语句：

```
CREATE SEQUENCE sequence_name
INCREMENT BY 1
START WITH 1
NOMAXVALUE
NOMINVALUE
NOCYCLE
CACHE 20;
```

2. 创建序列时，如果设置自增值为负数（假设为-1），并缺省其余可选项，即语句如下：

```
CREATE SEQUENCE sequence_name INCREMENT BY -1;
```

则默认创建一个自增值为-1，起始值为-1，无边界限制的NOCYCLE序列，缓存值为20。相当于构造了这样的序列创建语句：

```
CREATE SEQUENCE sequence_name
INCREMENT BY -1
START WITH -1
NOMAXVALUE
NOMINVALUE
NOCYCLE
CACHE 20;
```

3. 由于同一个Sequence允许被多用户访问，因此对于单个用户而言无法保证序列的连续性。

例 27. 在当前数据库创建一个从1000起始，自增值为-1的递减序列seq_num，最小值为0，NOCYCLE，缓存数目为100。

```
CREATE SEQUENCE seq_num
  INCREMENT BY -1
  START WITH 1000
  MINVALUE 0
  NOCYCLE
  CACHE 100;
```

例 28. 在当前数据库创建一个从10起始，自增值为1的递增序列seq_employee，其余选项缺省

```
CREATE SEQUENCE seq_employee
  INCREMENT BY 1
  START WITH 10;
```

- Alter Sequence

修改指定序列的配置项。

```
ALTER SEQUENCE [db_name.]sequence_name
  [INCREMENT BY <integer>]
  [MAXVALUE <integer> | NOMAXVALUE]
  [MINVALUE <integer> | NOMINVALUE]
  [CYCLE | NOCYCLE]
  [CACHE <integer> | NOCACHE]
  [ORDER | NOORDER];
```

该语法的选项与CREATE SEQUENCE语法的含义相同，只是不支持START WITH选项，若需修改序列起始值必须删除序列并重建。没有被修改的选项保持原来的设置。

例 29. 将seq_num的自增值修改为-2，最小值为1，并启动CYCLE

```
ALTER SEQUENCE seq_employee
  INCREMENT BY -2
  MINVALUE 1
  CYCLE;
```



修改后的序列仅影响未来产生的序列值。

- Drop Sequence

删除指定序列。

```
DROP SEQUENCE [IF EXISTS] [db_name.]sequence_name;
```

例 30. 删除seq_num序列

```
DROP SEQUENCE IF EXISTS seq_num;
```

- Show

查看当前数据库中有哪些序列。

```
SHOW SEQUENCES;
```

3.4.4.12.2. 对Sequence的访问

对Sequence的访问就是对CURRVAL和NEXTVAL的访问，调用CURRVAL和NEXTVAL的语句形式分别如下：

```
-- 访问CURRVAL
[ db_name . ]<sequence_name>.CURRVAL
```

```
-- 访问NEXTVAL
[ db_name . ]<sequence_name>.NEXTVAL
```

- 访问CURRVAL和NEXTVAL的场景限制

Inceptor对允许访问CURRVAL和NEXTVAL的场景有限制，它们仅可以出现在如下子句中：

1. 作为SELECT对象，而且该SELECT语句不包含于子查询中或以视图为查询对象。
2. INSERT语句中的VALUES子句；INSERT语句中的SELECT子句。
3. CREATE TABLE ... AS SELECT中的SELECT从句。
4. UPDATE语句中的SET子句。

不允许访问或者出现CURVAL和NEXTVAL的语句有：

1. 子查询或视图查询。
2. SELECT DISTINCT语句。
3. 带有GROUP BY 或 ORDER BY的SELECT语句。
4. 由UNION、INTERSECT或MINUS组合的语句。
5. SELECT语句的WHERE子句中。
6. CREATE TABLE或ALTER TABLE语句中列的默认值。
7. CREATE VIEW ... AS SELECT中的SELECT从句。

例 31. 创建表employee_tbl

```
CREATE TABLE employee_tbl
AS SELECT seq_employee.NEXTVAL, name, age, degree, onboard
      FROM employee;
```

例 32. select语句查询NEXTVAL和CURRVAL

```
SELECT seq_employee.NEXTVAL, seq_employee.CURRVAL
      FROM system.dual;
```

例 33. 向表employee_tbl插入一条记录，id值由序列seq_employee生成

```
INSERT INTO employee_tbl
VALUES(seq_employee.NEXTVAL, 'JK4', 43, 'BC', 2015);
```

例 34. 向表employee_tbl插入从表employee选出的年龄大于30的记录，id值由序列seq_employee生成

```
INSERT INTO employee_tbl
SELECT seq_employee.NEXTVAL, name, age, degree, onboard
FROM employee
WHERE age > 30;
```

例 35. NEXTVAL在PLSQL中的使用

```
CREATE OR REPLACE PROCEDURE func (b int)
IS
DECLARE
  v_id int;
BEGIN
  v_id := seq1.NEXTVAL;
  DBMS_OUTPUT.PUT_LINE(v_id);
END
```

- 使用CURRVAL和NEXTVAL的注意事项

- Session中第一个被调用的NEXTVAL将返回序列的起始值，之后的NEXTVAL返回按照INCREMENT BY改变的序列值。CURRVAL的结果是上一个NEXTVAL的返回。因此在同一Session中查询同一序列的CURRVAL和NEXTVAL时，首次访问CURRVAL之前必须先通过NEXTVAL对序列值初始化。
- NEXTVAL出现在select对象的多个列时，每次出现都会自增一次，例如：

```
SELECT seq_employee.NEXTVAL,
       seq_employee.NEXTVAL,
       seq_employee.CURRVAL
  FROM system.dual;
```

结果为：
10 11 11

NEXTVAL对于每条被查询（select）、被修改（update）、被插入（insert）的记录都自增一次，例如：

```
SELECT seq_employee.NEXTVAL,
       seq_employee.NEXTVAL,
       seq_employee.CURRVAL
  FROM employee
 WHERE age = 29;
```

结果为：
10 11 11
12 13 13

3.4.4.12.3. Sequence的使用实例

这里以一则创建订单的用例帮助用户理解序列的使用方法。

首先创建一个序列orderSeq用于产生有唯一性要求的订单号：

```
CREATE SEQUENCE orderSeq NOCACHE;
```

接着创建一个过程（procedure）用于创建新的订单，新建订单的流程是：通过orderSeq.nextval获取一个新的订单id，然后将其插入订单表orders中，并更新表itemLatestOrder中所购商品的最近购买订单号。语句如下：

```
CREATE OR REPLACE PROCEDURE createOrder ( itemId INT )
IS
DECLARE
    orderId INT
BEGIN
    orderId := orderSeq.nextval
    INSERT INTO orders VALUES (orderId, itemId)
    UPDATE itemLatestOrder SET order_id = orderId WHERE item_id = itemId
END;
```

3.5. 事务控制语言（TCL）

本章介绍事务处理语言。Inceptor支持的事务处理语法包括：

- **BEGIN TRANSACTION:** 开始事务
- **COMMIT:** 提交事务
- **ROLLBACK:** 回滚事务。



目前，Inceptor仅支持对ORC事务表的事务处理。关于ORC事务表的细节请参考[ORC表](#)。

模式选择

默认情况下Inceptor关闭Transaction Mode，要对ORC表进行事务处理，需要通过下面的开关打开ORC表对应的Transaction Mode：

```
SET transaction.type = inceptor;
```

3.5.1. 提交和回滚：COMMIT和ROLLBACK

事务处理指令为BEGIN TRANSACTION（开始事务），COMMIT（提交事务）和ROLLBACK（回滚/撤回事务）。一次事务以BEGIN TRANSACTION开始，以COMMIT或ROLLBACK结束。对ORC表的事务提交和回滚有两种方式：

- 手动方式：使用BEGIN TRANSACTION开始一次事务，使用COMMIT提交事务，使用ROLLBACK回滚事务。

语法

```
BEGIN TRANSACTION
<sql_statements>
[<sql_statements>]
...
COMMIT|ROLLBACK
```

- 自动提交（默认方式）：如果增删改指令（UPDATE, DELETE, MERGE, INSERT）没有以BEGIN TRANSACTION开始，那么Inceptor默认对所有的增删改指令都自动提交，不能回滚。

语法

```
[BEGIN TRANSACTION]
  <sql_statement>
[COMMIT]
[BEGIN TRANSACTION]
  <sql_statement>
[BEGIN TRANSACTION]
[COMMIT]
...
```

这里，用户并不需要输入括号中的事务处理指令，Inceptor自动执行。

Inceptor支持在事务中执行一个或多个子事务。Inceptor提供两种执行子事务的方式：嵌套事务(Nested Transaction)和自治事务(Autonomous Transaction)。

3.5.2. 嵌套事务

嵌套事务处理就在一个事务里进行另一个事务。嵌套事务需要写进一个procedure中：

语法

```
CREATE OR REPLACE PROCEDURE pro_test() IS
BEGIN
  BEGIN TRANSACTION
    <sql_statement>
  ...
  BEGIN TRANSACTION
    <sql_statement>
  ...
  (COMMIT|ROLLBACK)
  (COMMIT|ROLLBACK)
END;
```

在嵌套事务中，子事务与主事务互相影响。

举例

```
SET transaction.type=inceptor;
TRUNCATE TABLE t1;
TRUNCATE TABLE t2;
CREATE OR REPLACE PROCEDURE pro_test() IS
BEGIN
  -- 主事务
  BEGIN TRANSACTION
    INSERT INTO t1 VALUES ('a', 1)
    -- 子事务
    BEGIN TRANSACTION
      INSERT INTO t2 VALUES ('b', 2)
      COMMIT
    ROLLBACK
  END;
BEGIN
  pro_test()
END;
```

上面的例子中，

```
BEGIN TRANSACTION
  INSERT INTO t2 VALUES ('b', 2)
COMMIT
```

为

```

BEGIN TRANSACTION
  INSERT INTO t1 VALUES ('a', 1)
  BEGIN TRANSACTION
    INSERT INTO t2 VALUES ('b', 2)
  COMMIT
ROLLBACK

```

的子事务。当Inceptor读到子事务中的COMMIT时，Inceptor会将该COMMIT之前所有的任务提交。也就是说，“向t1插入('a', 1)”和“向t2插入('b', 2)”这两条任务都会被提交。主事务中的ROLLBACK和子事务中的COMMIT之间没有任何SQL语句，所以主事务中的ROLLBACK并不回滚任何任务。上面例子执行后，我们查看t1和t2中的记录将会得到下面的结果：

```

SELECT * FROM t1;
'a', 1
SELECT * FROM t2;
'b', 2

```

3.5.3. 自治事务

自治事务提供一个在主事务中嵌套独立子事务的机制。自治事务从当前事务开始，在其自身的语境中执行。它们能独立地被提交或重新运行，而不影响正在运行的事务，也不被正在运行的事务影响。因为自治事务是与主事务相分离的，所以它不能检测到被修改过的行的当前状态。但是主事务能够检测到已经执行过的自治事务的结果。

要创建一个自治事务，您必须在匿名块的最高层或者存储过程、函数、数据包或触发的定义部分中，使用PL/SQL中的PRAGMA AUTONOMOUS_TRANSACTION语句。在这样的模块或过程中执行的SQL语句都是自治的。

语法

```

-- 定义一个自治事务
CREATE OR REPLACE PROCEDURE transaction_name() IS
PRAGMA AUTONOMOUS_TRANSACTION
BEGIN
  <sql_statements>
  .
  .
  .
  COMMIT|ROLLBACK
END;

-- 使用该自治事务
BEGIN TRANSACTION;
<sql_statements>;
BEGIN
  transaction_name()
END;
COMMIT|ROLLBACK;

```

我们使用上一节的两个事务举例，将上一节中的事务2定义为自治事务，放在事务1中：

举例

```

CREATE OR REPLACE PROCEDURE autonomous_insert() IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    --子事务
    BEGIN TRANSACTION;
    INSERT INTO t2 VALUES ('b', 2);
    COMMIT;
END;

--主事务
BEGIN TRANSACTION;
INSERT INTO t1 VALUES ('a', 1);
BEGIN
    autonomous_insert();
END;
ROLLBACK;

```

在上例中，子事务中的COMMIT将只提交子事务中的任务而不对主事务有任何影响，也就是说该COMMIT只提交“向t2插入('b', 2)”这一条任务。而由于主事务以ROLLBACK结束，“向t1插入('a', 1)”这一任务将被回滚。上面例子执行后，我们查看t1和t2中的记录将会得到下面的结果：

```

SELECT * from t1;
SELECT * from t2;
'b', 2

```

3.6. 数据控制语言（DCL）

数据控制语言（DCL）用于管理Inceptor中的角色以及Inceptor用户的角色和权限。本章将大致介绍Inceptor中的DCL，Inceptor安全方面的内容，包括角色管理、权限管理、行级权限、计算资源隔离等，将在[Inceptor多租户手册](#)中详细介绍。

3.6.1. 角色管理语法

创建角色

```
CREATE ROLE <role_name>;
```

删除角色

```
DROP ROLE <role_name>;
```

列出用户当前角色

```
SHOW CURRENT ROLES;
```

切换角色

```
SET ROLE (<role_name>|ALL);
```

列出Inceptor中所有的角色

```
SHOW ROLES;
```

将角色赋予用户或其他角色

```
GRANT <role_name> [, <role_name>] ...
TO <principal_specification> [, <principal_specification>] ...
[ WITH ADMIN OPTION ];

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

将角色从用户或其他角色处收回

```
REVOKE [ADMIN OPTION FOR] role_name [, role_name] ...
FROM principal_specification [, principal_specification] ... ;

principal_specification
: USER user
| ROLE role
```

显示所有用户或角色被赋予的角色

```
SHOW ROLE GRANT (USER|ROLE) <principal_name>;
```

显示所有被赋予指定角色的用户和角色

```
SHOW PRINCIPALS <role_name>;
```

3.6.2. Inceptor对象权限管理语法

赋予权限

```
GRANT
<priv_type> [, <priv_type> ] ...
ON <table_or_view_name>
TO <principal_specification> [, <principal_specification>] ...
[WITH GRANT OPTION];

principal_specification
: USER <user>
| ROLE <role>

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

收回权限

```
REVOKE [GRANT OPTION FOR]
<priv_type> [, <priv_type> ] ...
ON <table_or_view_name>
FROM <principal_specification> [, <principal_specification>] ... ;

principal_specification
: USER <user>
| ROLE <role>

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

3.7. Database Links

一个database link定义了从一个物理数据库A到另一个物理数据库B的 **单向** 的连接，使得用户可以在A中使用B中的数据，让两个物理数据库可以像同一个逻辑数据库一样使用。这里，单向的意思是如果A中有一个连接到B的database link，那么A数据库中的用户可以用它连接到B，但是B数据库中的用户不能使用它连接

到A。



本章中我们将以“本地数据库”来指代database link所在的数据库，以“远程数据库”指代database link要从本地数据库连接的远程数据库。

Inceptor支持建立database link连接到远程数据库并进行 **查询操作（SELECT）**。Inceptor database link支持的远程数据库有：

- Oracle
- DB2
- PostgreSQL
- MySQL
- Inceptor
- Inceptor2

3.7.1. 使用前的准备

1. Database link是TDH4.3中的新功能，所以如果您的TDH是4.3之前的版本，您将不能使用database link。您需要先将您的集群升级到TDH4.3。
2. 请检查您的metastore版本，确保它在0.12.7或以上。如果您的metastore版本低于0.12.7，请先进行升级。
3. 使用database link连接到远程数据库之前，请先确保对应您要连接的远程数据库的驱动已经放在Inceptor server的classpath中，在您的TDH集群上，这个classpath是/usr/lib/hive/lib。

3.7.2. Database Link语法一览

Inceptor中database link的语法包括：

- 查看数据库中的database link

`SHOW DATABASE LINKS`

- 描述database link

`DESCRIBE DATABASE LINK <db_link_name>`

- 创建database link

`CREATE DATABASE LINK <db_link_name> CONNECT TO <user_name> IDENTIFIED BY <password> USING <jdbc_connection_string>`

- 删除database link

`DROP DATABASE LINK <db_link_name>`

- 调用远程数据库中的表

```
SELECT ... FROM <table_name>@<db_link_name>
```

下面我们具体介绍这些指令的用法。

3.7.3. 查看：SHOW DATABASE LINKS

语法

```
SHOW DATABASE LINKS
```

举例

```
[$host] show database links;
dblink2
oracle_dblink_qls
oracle_dblink_yy
```

3.7.4. 描述：DESCRIBE DATABASE LINK

语法

```
DESCRIBE DATABASE LINK <db_link_name>
```

举例

```
describe database link 119link;
DbLink(dbLinkName:119link, userName:hive, pwd:123, srvc:jdbc:hive2://172.16.1.119:10000/default)
```

3.7.5. 创建：CREATE DATABASE LINK

语法

```
CREATE DATABASE LINK <db_link_name> CONNECT TO <user_name> IDENTIFIED BY <password> USING
<jdbc_connection_string>;
```

- <db_link_name>处为database link的名字。
- <user_name>为远程数据库中用户的用户名
- <password>为远程数据库中用户的密码
- <jdbc_connection_string>为连接到远程数据库的JDBC连接串

创建database link的步骤如下：

1. 将您要连接的远程数据库的驱动放在集群中 所有 节点的/usr/lib/hive/lib目录下，然后重启Inceptor server。
2. 获取您要连接的远程数据库的JDBC连接串。
3. 获取和您在远程数据库中使用的用户名及密码。

- 在Inceptor中创建database link。



请确保您要使用的远程数据库的驱动程序在/usr/lib/hive/lib/中others有rx权限，否则Inceptor将不能使用驱动，导致建好的database link使用出错。

下面，我们将接着这四个步骤演示连接到Inceptor支持的各种远程数据库。

3.7.5.1. 连接到InceptorServer 1

- 在Inceptor内用database link连接一个远程Inceptor您不需要准备驱动。
- 获取连接到Inceptor1的JDBC连接串，形式如下：

```
jdbc:hive://<server_ip/hostname>:10000/<database_name>
```

这里<server_ip/hostname>是Inceptor server所在节点的hostname或者ip；<database_name>处提供您在Inceptor中要使用的database。

- 由于Inceptor1中没有用户概念，我们不需要提供用户名和密码，也就是说创建database link语句中的“CONNECT TO <user_name> IDENTIFIED BY <password>”部分需要省掉。
- 在Inceptor中里创建database link。

举例：连接到在172.16.1.56的Inceptor1

```
CREATE DATABASE LINK demo02hive USING 'jdbc:hive://172.16.1.56:10000/default';
```

3.7.5.2. 连接到InceptorServer 2

- 在Inceptor内用database link连接一个远程Inceptor您不需要准备驱动。
- 获取连接到Inceptor2的JDBC连接串，形式如下：

```
jdbc:hive2://<server_ip/hostname>:10000/<database_name>
```

这里<server_ip/hostname>是Inceptor server所在节点的hostname或者ip；<database_name>处提供您在Inceptor中要使用的database。

- 获取远程Inceptor2中用户的用户名和密码。
- 在Inceptor中里创建database link。

举例：连接到在172.16.1.119的Inceptor2

```
CREATE DATABASE LINK hive119 CONNECT TO hive IDENTIFIED BY '123' USING 'jdbc:hive2://172.16.1.119:10000/default';
```

3.7.5.3. 连接到Oracle数据库

- 将Oracle数据库的驱动放在集群中所有节点的/usr/lib/hive/lib目录下。这个驱动您可以从Oracle官网下载到，然后重启Inceptor server。

2. 获取连接到Oracle数据库的JDBC连接串，形式如下：

```
jdbc:oracle:thin:@<oracle_hostname/ip>:1521/<database_name>
```

这里<oracle_hostname/ip>是Oracle数据库所在节点的hostname或者ip；<database_name>处提供您在Oracle中要使用的database。

3. 获取远程Oracle数据库中用户的用户名和密码。
4. 在Inceptor中创建database link。

举例：连接到172.16.1.55上的Oracle数据库

```
CREATE DATABASE LINK oracle_dblink_alice CONNECT TO alice IDENTIFIED BY '123' USING
'jdbc:oracle:thin:@172.16.1.55:1521/XE';
```



- 您需要确保您连接Oracle数据库所用的用户有create session权限。
- 使用Oracle中的表进行查询时必须将Oracle中的表的表名大写。

3.7.5.4. 连接到Mysql数据库

1. 将Mysql数据库的驱动放在集群中 所有 节点的/usr/lib/hive/lib目录下。这个驱动您可以从Mysql官网下载到，然后重启Inceptor server。
2. 获取连接到Mysql数据库的JDBC连接串，形式如下：

```
jdbc:mysql://<mysql_hostname/ip>:3306/<database_name>
```

这里<mysql_hostname/ip>是Mysql数据库所在节点的hostname或者ip；<database_name>处提供您在Mysql中要使用的database。

3. 获取远程Mysql数据库中用户的用户名和密码。
4. 在Inceptor中创建database link。

举例：连接到172.16.1.118上的Mysql数据库

```
CREATE DATABASE LINK mysql_bob_118 CONNECT TO bob IDENTIFIED BY '123456' USING
'jdbc:mysql://172.16.1.118:3306/test';
```



- 远程Mysql数据库中的用户不能够绑定hostname。

3.7.5.5. 连接到DB2数据库

1. 将DB2数据库的驱动放在集群中 所有 节点的/usr/lib/hive/lib目录下。这个驱动您可以从IBM官网下载到，然后重启Inceptor server。
2. 获取连接到DB2数据库的JDBC连接串，形式如下：

```
jdbc:db2://<db2_hostname/ip>:<port>/<database_name>
```

这里<db2_hosename/ip>是Mysql数据库所在节点的hostname或者ip; <port>处提供DB2数据库服务的端口，一般为50000; <database_name>处提供您在DB2数据库中要使用的database。

3. 获取远程DB2数据库中用户的用户名和密码。
4. 在Inceptor中创建database link。

举例：连接到172.16.1.94上的DB2数据库

```
CREATE DATABASE LINK db2_db2inst1_94 CONNECT TO db2inst1 IDENTIFIED BY 'd' USING
'jdbc:db2://172.16.1.94:50000/dbtest01';
```

3.7.5.6. 连接到PostgreSQL数据库

1. 将PostgreSQL数据库的驱动放在集群中所有节点的/usr/lib/hive/lib目录下。这个驱动您可以从IBM官网下载到，然后重启Inceptor server。
2. 获取连接到PostgreSQL数据库的JDBC连接串，形式如下：

```
jdbc:postgresql://<postgresql_hostname/ip>:<port>/<database_name>
```

这里<db2_hosename/ip>是Mysql数据库所在节点的hostname或者ip; <port>处提供DB2数据库服务的端口，默认值是5432; <database_name>处提供您在PostgreSQL数据库中要使用的database。

3. 获取远程PostgreSQL数据库中用户的用户名和密码。
4. 在Inceptor中创建database link。

举例：连接到172.16.2.104上的PostgreSQL数据库

```
CREATE DATABASE LINK postgresql_2104_root CONNECT TO root IDENTIFIED BY '123456' USING
'jdbc:postgresql://172.16.2.104:5432/transwarp_manager';
```



您需要确保您有访问PostgreSQL的权限（在pg_hba.conf文件中设置）。

3.7.6. 删除：DROP DATABASE LINK

语法

```
DROP DATABASE LINK <db_link_name>
```

在<db_link_name>处提供要删除的database link名称。

举例

```
[$host] DROP DATABASE LINK mysql_bob_118;
```

3.7.7. 使用Database Link

语法：调用远程数据库中的表

```
SELECT...FROM <table_name>@<db_link_name>
```

- <table_name>处提供远程数据库中的表
- <db_link_name>处提供database link名

举例：查看Oracle中所有alice有权限看到的表

```
[$host] SELECT table_name FROM USER_TABLES@oracle_dblink_alice;
```

举例：查看Oracle数据库中表BANK_INFO中的信息

```
[$host] SELECT * FROM BANK_INFO@oracle_dblink_alice;
```

注意，在Inceptor中使用database link使用远程Oracle数据库时，远程Oracle数据库中的表的表名必须大写。

举例：查看一张Inceptor中的表中的数据

```
[$host] SELECT * FROM BANK_INFO@hive119;
```

举例：查看一张Mysql中的表中的数据

```
[$host] SELECT * FROM table_bob@mysql_bob_118;
```

举例：查看一张DB2中的表中的数据

```
[$host] SELECT * FROM TIMETEST@db2_db2inst1_94;
```

举例：查看一张PostgreSQL中的表中的数据

```
[$host] SELECT * FROM service@postgresql_2104_root;
```

3.8. 数据稽查

3.8.1. 本节使用数据

/DataAudit/employee/employee.txt中的数据：

```
1,"YH1",'77a',PG,2013
2,"LC1",29,BC,2013
3,"KY1",22,PG,2013
4,"MH1",23,PG,2013
5,"LX1",32,PG,2014
1,"YH2",36,PG,2013
2,"LC3",29,BC,2013
3,"KY3",22,PG,2013
4,"MH3",23,PG,2013
5,"LX3",'abc',PG,2014
,"AM5",36,FG,2014
```

3.8.2. 基本介绍

用户利用INSERT或者LOAD导入数据时有时会无意将脏数据录入业务表，查询语句在访问这些脏表时，可能会运行失败或者返回错误结果。为避免此情况，Inceptor提供数据稽查功能。具体实施时，数据稽核会根据规则将脏数据写入指定的脏数据表（Error Table），并标明每一条脏数据为何非法，在数据导入完成后，将返回总共记录数、导入记录数的接口、或者数据质量报告，以方便监控程序判断以及处理。也可以仅打印出显示报错信息。实现上述特性都是为了使数据稽查能够在脏数据存在的情况下尽可能的保护系统或保证业务的顺畅执行。

进行数据稽查时，系统将对如下问题数据报错并记录至Error Table：

1. 字段值中含有定界标识符，导致读取数据时一行数据被误读为两行。因为错误的切分方式会使读入记录的字段数与定义不符，系统将通过检测列数总个数来识别这样的脏数据。
2. 以目标结果表的类型判断为标准，进行类型匹配与类型转换，如果类型不匹配，则输出脏数据。
3. 在通过UDF结合过滤条件，实现其他的数据转换以及过滤时，对不匹配的数据记录打印报错或记录于Error Table。
4. 对不符合NOT NULL限制的记录报错。

完整的数据稽查功能是按照如下的处理流程实现的，建议用户在使用数据稽查功能以及设置相关配置时，结合该流程决定配置参数：

1. 用户可以在创建一个外表的同时指定Log Error Table。
2. 当从外表读取数据的时候，每解析一行记录，若访问到上述四种无效数据，就将该数据写入Error Table。
3. 允许指定REJECT策略，即当错误率达到一定的行数或者比例时，就停止读取。



- 目前数据稽查功能只对外表开放。
- 对于SELECT或过滤操作，可将脏数据写入Error Table；但是无法录入GROUP BY或者复杂SQL中的问题数据，目前仅仅将报错忽略掉。
- 聚合函数遇到脏数据时不报错而是直接跳过脏数据去计算合法数据。如：对于“SELECT count(column_name) FROM table_name”，在数据稽查的保护下，如果table_name有非法数据，返回结果会少于实际行数，但是不会出现相关报错信息。
- 不论是否进行稽查，count(*)和count(1)在任何情况下都会输出满足条件的包括脏数据的总行数。

3.8.3. 相关语法

指定Error Table

创建外表时指定Log Error Table。

```
CREATE EXTERNAL TABLE table_name (column1 datatype1, column2 datatype2, ...)
LOG ERRORS INTO error_table_name [OVERWRITE]
[SEGMENT REJECT LIMIT n [ ROWS | Percent ] ]
```

- error_table_name是存放当前表脏数据信息的Error Table的名称，如果不存在系统会自动创建一个。只能在创建表时指定。注意关键字LOG ERRORS INTO。
- 若允许Overwrite Error Table，需在相应位置写“OVERWRITE”，否则忽略。禁止Overwrite的坏处是每次都会在原有记录的基础上写入脏数据信息，使该Error Table不断扩增；好处是能追溯至更久远的信

息。用户应根据需求设置。

- 若需执行REJECT策略，应补充“SEGMENT REJECT LIMIT n...”部分，如果没有需求就忽略。n是REJECT阈值，表示REJECT之前允许的非法数据的行数或比例。**应注意**，分布式结构下，由于语句的执行被切分交给不同task实现，所以n是相对于一个task中的数据行数而言的，而并非总数据行数。
- 没有指定Error Table的表也可以受到数据稽查的保护，只是错误信息不会收录于Error Table。
- 指定了Error Table并不代表启动了数据稽查，需要通过后面将介绍的开关控制。

例 36. 创建外表指定关联的Error Table

- 创建employee_err表，导入employee.txt中的记录（包含脏数据）。并指定Error Table为employee_error_table，允许Overwrite，采用LIMIT=2的Reject策略，即访问脏数据大于两行时就停止执行。

```
CREATE EXTERNAL TABLE employee_err (
    id int NOT NULL,
    name string,
    age tinyint,
    degree string,
    onboard int
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LOCATION '/DataAudit/employee'
LOG ERRORS INTO employee_error_table
SEGMENT REJECT LIMIT 2 ROWS;
SELECT * FROM employee_err;
+-----+-----+-----+-----+-----+
| id | name | age | degree | onboard |
+-----+-----+-----+-----+-----+
| 1  | "YH1" | NULL | PG    | 2013   |
| 2  | "LC1" | 29   | BC    | 2013   |
| 3  | "KY1" | 22   | PG    | 2013   |
| 4  | "MH1" | 23   | PG    | 2013   |
| 5  | "LX1" | 32   | PG    | 2014   |
| 1  | "YH2" | 36   | PG    | 2013   |
| 2  | "LC3" | 29   | BC    | 2013   |
| 3  | "KY3" | 22   | PG    | 2013   |
| 4  | "MH3" | 23   | PG    | 2013   |
| 5  | "LX3" | NULL  | PG    | 2014   |
| NULL | "AM5" | 36   | FG    | 2014   |
+-----+-----+-----+-----+-----+
```

控制开关

下面是和数据稽查相关的三个开关，用于控制其工作特性：

```
SET inception.data.audit = true/false; ①
SET inception.strict.evaluate = true/false; ②
SET inception.notnull.audit = true/false; ③
```

① 数据稽查总开关，开启时会做脏数据和NOT NULL检查，默认关闭。

② 是否在遇到脏数据时报Exception，默认关闭。

③ 是否对NOT NULL Constraint进行检查，默认关闭。

- inception.data.audit是Top Level开关，设为true后，后面两个开关强制为true，启动inception.data.audit之后再对两个子开关进行设置将不起作用。
- 后两个开关可以在inception.data.audit关闭时做设置。区别在于，如果开启data.audit，关于脏数据和NOT NULL限制的报错都会写入Error Table；若在关闭data.audit后启动strict.evaluate或notnull.audit，执行对应检测时报错将被打印于界面。



例 37. 开inceptor.data.audit的范例

```

SET inceptor.data.audit = true;
TRUNCATE TABLE employee_error_table;
SELECT * FROM employee_err;
+-----+-----+-----+-----+
| id | name | age | degree | onboard |
+-----+-----+-----+-----+
| 2  | "LC1" | 29 | BC    | 2013   |
| 3  | "KY1" | 22 | PG    | 2013   |
| 4  | "MH1" | 23 | PG    | 2013   |
| 5  | "LX1" | 32 | PG    | 2014   |
| 1  | "YH2" | 36 | PG    | 2013   |
| 2  | "LC3" | 29 | BC    | 2013   |
| 3  | "KY3" | 22 | PG    | 2013   |
| 4  | "MH3" | 23 | PG    | 2013   |
+-----+-----+-----+-----+
SELECT count(age) FROM employee_err; ①
+-----+
| _c0 |
+-----+
| 9   |
+-----+ ②
SELECT count(*) FROM employee_error_table;
+-----+
| _c0 |
+-----+
| 3   |
+-----+ ③

```

① 总开关打开后，结果仅返回合法数据。

② age字段中的合法值有9个。

③ 有三条脏数据录入Error Table。

例 38. 关inceptor.data.audit, 开inceptor.strict.evaluate的范例

```

SET inceptor.data.audit = false;
SET inceptor.strict.evaluate = true;
SELECT * FROM employee_err;

Error: Error while processing statement: FAILED: Execution Error, return code 1 FROM
io.transwarp.inceptor.execution.SparkTask. Job aborted due to stage failure: Task 0 in stage
77.0 failed 4 times, most recent failure: Lost task 0.3 in stage 77.0 (TID 248, tw-node129):
java.lang.RuntimeException: org.apache.hadoop.hive.ql.metadata.HiveException:
java.lang.NumberFormatException: '77a' (state=08S01,code=1)

```

Error Table的属性控制

- 通过如下语句停止对table_name数据稽查，不再对其记录脏数据信息。只是对table_name的变化，不影响其他表。

```
ALTER TABLE table_name SET ERRORS LOG OFF;
```

- 使数据稽查重新作用于table_name，此时必须设置OVERWRITE和REJECT的开关状态，n必须为整数。也可以用于修改OVERWRITE和REJECT的设置。

```
ALTER TABLE table_name SET ERRORS LOG ON OVERWRITE [on|off] REJECT [on|off] LIMIT n ROWS;
```

例 39. 通过SET OFF LOG, 停止对employee_err做数据稽查

```

ALTER TABLE employee_err SET ERRORS LOG OFF;
TRUNCATE TABLE employee_error_table;
SELECT * FROM employee_err;
+-----+-----+-----+-----+-----+
| id  | name | age  | degree | onboard |
+-----+-----+-----+-----+-----+
| 1   | "YH1" | NULL | PG    | 2013  |
| 2   | "LC1" | 29   | BC    | 2013  |
| 3   | "KY1" | 22   | PG    | 2013  |
| 4   | "MH1" | 23   | PG    | 2013  |
| 5   | "LX1" | 32   | PG    | 2014  |
| 1   | "YH2" | 36   | PG    | 2013  |
| 2   | "LC3" | 29   | BC    | 2013  |
| 3   | "KY3" | 22   | PG    | 2013  |
| 4   | "MH3" | 23   | PG    | 2013  |
| 5   | "LX3" | NULL | PG    | 2014  |
| NULL| "AM5" | 36   | FG    | 2014  |
+-----+-----+-----+-----+-----+
①
SELECT count(age) FROM employee_err;
+-----+
| _c0 |
+-----+
| 11  |
+-----+
②
SELECT count(*) FROM employee_error_table;
+-----+
| _c0 |
+-----+
| 0   |
+-----+
③

```

① SET OFF LOG之后，即使结果为脏依然返回。

② 对employee_err关闭数据稽查后，count(age)返回总字段数。

③ 没有脏数据信息录入Error Table。

例 40. 对employee_err重新打开数据稽查，OVERWRITE on, REJECT on LIMIT 2 ROWS

```

ALTER TABLE employee_err SET ERRORS LOG ON OVERWRITE ON REJECT ON LIMIT 2 ROWS;
SELECT * FROM employee_err;
+-----+-----+-----+-----+-----+
| id  | name | age  | degree | onboard |
+-----+-----+-----+-----+-----+
| 2   | "LC1" | 29   | BC    | 2013  |
| 3   | "KY1" | 22   | PG    | 2013  |
| 4   | "MH1" | 23   | PG    | 2013  |
| 5   | "LX1" | 32   | PG    | 2014  |
| 1   | "YH2" | 36   | PG    | 2013  |
| 2   | "LC3" | 29   | BC    | 2013  |
| 3   | "KY3" | 22   | PG    | 2013  |
| 4   | "MH3" | 23   | PG    | 2013  |
+-----+-----+-----+-----+-----+
SELECT count(age) FROM employee_err;
+-----+
| _c0 |
+-----+
| 9   |
+-----+
①
SELECT count(*) FROM employee_error_table;
+-----+
| _c0 |
+-----+
| 3   |
+-----+

```

① Error Table中写入了三条报错信息。

例 41. 在上例基础上，把对employee_err的数据稽查设置为，OVERWRITE off, REJECT on LIMIT 2 ROWS。

```
ALTER TABLE employee_err SET ERRORS LOG ON OVERWRITE OFF REJECT ON LIMIT 2 ROWS;
SELECT * FROM employee_err;
+-----+-----+-----+-----+-----+
| id | name | age | degree | onboard |
+-----+-----+-----+-----+-----+
| 2  | "LC1" | 29 | BC    | 2013  |
| 3  | "KY1" | 22 | PG    | 2013  |
| 4  | "MH1" | 23 | PG    | 2013  |
| 5  | "LX1" | 32 | PG    | 2014  |
| 1  | "YH2" | 36 | PG    | 2013  |
| 2  | "LC3" | 29 | BC    | 2013  |
| 3  | "KY3" | 22 | PG    | 2013  |
| 4  | "MH3" | 23 | PG    | 2013  |
+-----+-----+-----+-----+-----+
SELECT count(age) FROM employee_err;
+-----+
| _c0 |
+-----+
| 9   |
+-----+
SELECT count(*) FROM employee_error_table;
+-----+
| _c0 |
+-----+
| 6   | ①
+-----+
```

① Error Table在原有基础上新增了三条报错信息。

例 42. 在上例基础上，把对employee_err的数据稽查设置为，OVERWRITE off, REJECT on LIMIT 0 ROWS。

```
ALTER TABLE employee_err SET ERRORS LOG ON OVERWRITE OFF REJECT ON LIMIT 0 ROWS;
SELECT count(*) FROM employee_error_table;
+-----+
| _c0 |
+-----+
| 6   |
+-----+
SELECT * FROM employee_err;
Error: Error while processing statement: FAILED: Execution Error, return code 1 FROM
io.transwarp.inceptor.execution.SparkTask. Job aborted due to stage failure: Task 1 in stage
148.0 failed 4 times, most recent failure: Lost task 1.3 in stage 148.0 (TID 374, tw-node127):
org.apache.hadoop.hive.ql.metadata.HiveException:
org.apache.hadoop.hive.ql.metadata.HiveException:
org.apache.hadoop.hive.ql.metadata.HiveException: Stop reading table since error rate exceeds
the reject criteria. The error rows number 1 exceeds the reject rows number in DDL 0 (state=
08S01,code=1)
SELECT count(*) FROM employee_error_table;
+-----+
| _c0 |
+-----+
| 6   | ①
+-----+
```

① 没有向Error Table写入新信息。

例 43. 在上例基础上，把对employee_err的数据稽查设置为，OVERWRITE on, REJECT off

```
ALTER TABLE employee_err SET ERRORS LOG ON OVERWRITE ON reject off;
SELECT count(*) FROM employee_error_table;
+-----+
| _c0 |
+-----+
| 6   |
+-----+ ①
SELECT * FROM employee_err;
+-----+-----+-----+-----+-----+
| id | name | age | degree | onboard |
+-----+-----+-----+-----+-----+
| 2  | "LC1" | 29  | BC    | 2013  |
| 3  | "KY1" | 22  | PG    | 2013  |
| 4  | "MH1" | 23  | PG    | 2013  |
| 5  | "LX1" | 32  | PG    | 2014  |
| 1  | "YH2" | 36  | PG    | 2013  |
| 2  | "LC3" | 29  | BC    | 2013  |
| 3  | "KY3" | 22  | PG    | 2013  |
| 4  | "MH3" | 23  | PG    | 2013  |
+-----+-----+-----+-----+-----+
SELECT count(*) FROM employee_error_table;
+-----+
| _c0 |
+-----+
| 3   |
+-----+ ②
```

① 查询语句执行之前Error Table中有六条报错记录。

② 查询语句执行之后Error Table被Overwrite。

3.8.4. Error Table的信息

检查是否成功指定Error Table

有时我们需要查看表table_name的Error Table是否已被指定为error_table_name，可以通过“DESC FORMATTED table_name”实现。如果在返回结果中存在如下信息，则表示该表脏数据信息可被记录于指定的Error Table。

```
ErrorTableSetting{errorTableName='employee_error_table', rejectEnable=true, overwriteOn=false, rowCount=2}
```

Error Table的结构

Error Table的创建语句如下，由系统自动构建不用手动操作：

```
CREATE TABLE error_table_name (
  sql_time string,
  issue_sql string,
  real_table_name string,
  error_file_name string,
  error_block_offset bigint,
  error_msg string,
  raw_data string
)
```

各个字段分别有其专门的代表含义：

- sql_time：报错语句在何时被执行。
- issue_sql：报错语句的具体内容。
- real_table_name：在访问哪张表时读到脏数据。

- error_file_name: 出现脏数据的具体文件。
- error_block_offset: 脏数据所在文件的块偏移。
- error_msg: 具体报错信息。
- raw_data: 出错记录的原始数据。

可以通过“DESC error_table_name;”查看Error Table的结构:

```
desc employee_error_table;
+-----+-----+-----+-----+-----+
| col_name | data_type | comment | notnull_constraint | unique_const |
+-----+-----+-----+-----+-----+
| sql_time | string    |          |          |          |
| issue_sql | string    |          |          |          |
| real_table_name | string |          |          |          |
| error_file_name | string |          |          |          |
| error_block_offset | bigint |          |          |          |
| error_msg | string    |          |          |          |
| raw_data | string    |          |          |          |
+-----+-----+-----+-----+-----+
```

查看Error Table内容

对Error Table可以像普通表一样操作，所以用“SELECT * FROM error_table_name”就能够查看其中完整内容。在之前的例子中也可以看到，我们通过“SELECT count(*) FROM error_table_name”获得脏数据记录行数。

例 44. 查看Error Table的内容

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/error/' SELECT * FROM employee_error_table; ①

cat /tmp/error/00000_0
cat /tmp/error/00001_0
cat /tmp/error/00002_0 ②
```

① 查看Error Table内容，因为显示信息过长，可以把返回信息保存在文件中以便显示。

② 到本地文件中查看返回结果信息。

3.8.5. Error Table的权限控制

- 如果外表的SELECT权限被GRANT给用户B，用户B就拥有外表的读权限以及Error Table的写权限，但是没有Error Table读权限。也就是说，如果需要读Error Table，必须额外将SELECT权限GRANT给用户B。
- 如果外表的权限没有被GRANT给用户B，用户B将不具有对ERROR TABLE的任何权限。

3.9. TEXT表

TEXT表是文本格式的表，是Inceptor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。针对不同的使用场景，用户可以将其中的数据放入ORC表或Holodesk表中。

Inceptor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，Inceptor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** 使用这个方式导数据。

本章将讨论TEXT表的一些细节。

3.9.1. 建TEXT表

由于TEXT表通常用于将文本文件中的原始数据导入Inceptor中，最常见的建表方式是通过直接定义列建表。**CREATE TABLE LIKE** 和 **CREATE TABLE AS SELECT** 也可以用于建TEXT表，用法简单明了，请参考“**CREATE/DROP/ALTER TABLE**”章节，这里不赘述。

语法：通过定义列建表

```
CREATE [TEMPORARY] [EXTERNAL] TABLE <table_name> ①
(<column_name> <data_type>, <column_name> <data_type>, ...)
[PARTITIONED BY ...] ②
[CLUSTERED BY ...] ③
[ROW FORMAT ...] ④
[STORED AS TEXTFILE] ⑤
[LOCATION '<hdfs_path>'] ⑥
[TBLPROPERTIES ('<property_name>'='<property_value>', ...)]; ⑦
```

- ① **[TEMPORARY]** 为临时表选项，**[EXTERNAL]** 为外表选项。
- ② 分区选项，TEXT表可以分区，本章不详细讨论如何分区，请参考“分区表”章节。
- ③ 分桶选项，TEXT表可以分桶，本章不详细讨论如何分桶，请参考“分桶表”章节。
- ④ 指定字段分隔符，如果不指定将使用Inceptor的默认分隔符，具体细节请参考**分隔符**部分。
- ⑤ 指定表存储为TEXTFILE（文本文件）。因为TEXTFILE是Inceptor的默认存储格式，该选项可省略。
- ⑥ 指向HDFS上的一个目录。这个选项我们推荐只在建外表时使用，也就是和 **EXTERNAL** 选项合用（虽然也可以在建内表时使用，但是我们 **不建议** 这样做）。该路径可以是一个绝对路径，比如 `/user/alice/employee` 或者一个完整的URL：`hdfs://nameservice1/user/alice/employee`。执行该建表操作的用户必须是这个路径指向的目录或文件的owner。如果 `<hdfs_path>` 指向的目录不存在，Inceptor会尝试新建这个目录，但是安全模式下Inceptor可能没有在指定路径新建目录的权限，所以星环科技建议尽量避免让 `<hdfs_path>` 指向不存在的目录。
- ⑦ 表属性，由键值对表示。

例 45. 建外表并指定其在HDFS上的路径

```
CREATE EXTERNAL TABLE employee_id (id INT) LOCATION '/user/alice/employee_id';
```

注意事项

- 执行该操作的用户必须是指定的HDFS目录的owner。
- 表的行格式（由 **ROW FORMAT** 指定）和文件格式（由 **STORED AS** 指定）须和HDFS目录下的数据一致，保证Inceptor能够正常解析。

3.9.2. 分隔符

当一张TEXT表直接使用文本文件作为数据源时，用户需要在建表时用 **ROW FORMAT DELIMITED ...** （指

定单字符分隔符) 或者 **ROW FORMAT SERDE** 'org.apache.hadoop.hive.contrib.serde2.MultiDelimitSerDe' ... (指定多字符分隔符) 指定文件中文本的分隔符, 以便让Inceptor识别文本中的字段。如果建表时指定的分隔符和文本文件中实际使用的分隔符不匹配, Inceptor读取数据时会发生错误。如果不显式指定分隔符, Inceptor会使用默认值。

3.9.2.1. 单字符分隔符

单字符分隔符使用 **ROW FORMAT DELIMITED** 指定, 语法如下。分隔符的指定可以使用字符本身或者字符的八进制ASCII编码, 例如 “\001” 等。

语法

```
ROW FORMAT DELIMITED
[FIELDS TERMINATED BY '<column_delimiter>' [ESCAPED BY '<char>']] ①
[COLLECTION ITEMS TERMINATED BY '<complex_type_delimiter>'] ②
[MAP KEYS TERMINATED BY '<kv_delimiter>'] ③
[LINES TERMINATED BY '<newline_char>'] ④
```

- ① **<column_delimiter>** 为列分隔符, 默认值为 “\001”。如果文本中的实际数据包含指定的列分隔符, 可以用 **ESCAPED BY** 指定转义符, 将列分隔符和实际数据进行区别, 转义符无默认值。
- ② **<complex_type_delimiter>** 为复杂数据类型 (ARRAY/MAP/STRUCT) 中的字段分隔符, 默认值为’\002’。
- ③ **<kv_delimiter>** 为*MAP* 类型中将键和值分隔的分隔符, 默认值为’\003’。
- ④ **<newline_char>** 为换行符, 默认值为’\n’。

例 46. 自定义列分隔符

已知有一份原始数据在HDFS上的 /user/alice/employee 目录下, 各列由 “,” 分开, 如下:

```
1, Alice
2, Bob
```

以这份数据为源建表, 我们需要将 “,” 指定为列分隔符:

```
CREATE EXTERNAL TABLE employee (id INT, name STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY
', ' LOCATION '/user/alice/employee';
```

建表完成后查看表中内容可得:

```
SELECT * FROM employee;
+----+-----+
| id | name |
+----+-----+
| 1  | Alice |
| 2  | Bob  |
+----+-----+
```

例 47. 自定义ARRAY中字段的分隔符

已知有一份原始数据在HDFS上的 /user/alice/employee_absence 目录下，内容如下：

```
1, Alice, 2014-01-04|2014-10-24
2, Bob, 2014-02-14|2014-07-02|2014-10-31
```

数据记录了员工的ID、名字和请假情况，列分隔符为“，”。请假情况为一系列的日期，我们要把请假日期放在一个ARRAY中，那么该ARRAY中字段的分隔符是“|”。以这份数据为源建表，ARRAY中字段的分隔符由 **COLLECTION ITEMS TERMINATED BY** 指定：

```
CREATE TABLE employee_absence (id INT, name STRING, absent_date ARRAY<DATE>) ROW FORMAT
DELIMITED FIELDS TERMINATED BY ',' COLLECTION ITEMS TERMINATED BY '|' LOCATION
'/user/alice/employee_absence';
```

建表完成后查看表中内容可得：

```
SELECT * FROM employee_absence;
+---+---+-----+
| id | name | absent_date |
+---+---+-----+
| 1 | Alice | ["2014-01-04", "2014-10-24"] |
| 2 | Bob | ["2014-02-14", "2014-07-02", "2014-10-31"] |
+---+---+-----+
```

例 48. 自定义STRUCT中字段的分隔符

已知有一份原始数据在HDFS上的 /user/alice/employee_info 目录下，内容如下：

```
1, Alice, 26|Shanghai
2, Bob, 28|Beijing
```

数据记录了员工的ID、名字和个人信息，列分隔符为“，”。个人信息部分包括年龄（INT类型）和家乡（STRING类型），我们需要把个人信息放在一个STRUCT中，那么该STRUCT中字段的分隔符是“|”。以这份数据为源建表，STRUCT中字段的分隔符由 **COLLECTION ITEMS TERMINATED BY** 指定：

```
CREATE TABLE employee_info (id INT, name STRING, info STRUCT<age:INT, hometown:STRING>) ROW FORMAT
DELIMITED FIELDS TERMINATED BY ',' COLLECTION ITEMS TERMINATED BY '|' LOCATION
'/user/alice/employee_info';
```

建表完成后查看表中内容可得：

```
SELECT * FROM employee_info;
+---+---+-----+
| id | name | info |
+---+---+-----+
| 1 | Alice | {"age":26, "hometown":"Shanghai"} |
| 2 | Bob | {"age":28, "hometown":"Beijing"} |
+---+---+-----+
```

例 49. 自定义MAP中键值对之间的分隔符以及键、值之间的分隔符

已知有一份原始数据在HDFS上的 /user/alice/employee_salary 目录下，内容如下：

```
1, Alice, 2013:120000|2014:125000|2015:130000
2, Bob, 2014:150000|2015:160000
```

数据记录了员工的ID、名字和历年的薪酬状况，列分隔符为“，”。我们需要把历年的薪酬状况放在一个MAP中，以年份为键、薪酬为值。键值对之间的分隔符为“|”；每个键值对之内，键和值之间的分隔符为“：“。以这份数据为源建表，MAP中键值对之间的分隔符由 **COLLECTION ITEMS TERMINATED BY** 指定，键值对内键和值之间的分隔符由 **MAP KEYS TERMINATED BY** 指定：

```
CREATE TABLE employee_salary (id INT, name STRING, salary MAP<STRING, DOUBLE>) ROW FORMAT  
DELIMITED FIELDS TERMINATED BY ',' COLLECTION ITEMS TERMINATED BY '|' MAP KEYS TERMINATED BY  
':' LOCATION '/user/alice/employee_salary';
```

建表完成后查看表中内容可得：

```
SELECT * FROM employee_salary;
+----+----+-----+
| id | name | salary           |
+----+----+-----+
| 1  | Alice | {"2013":120000.0, "2014":125000.0, "2015":130000.0} |
| 2  | Bob   | {"2014":150000.0, "2015":160000.0}    |
+----+----+-----+
```

3.9.2.2. 多字符分隔符

使用 **ROW FORMAT DELIMITED ...** 指定的分隔符只能是一个 **char**，实际生产中有很多文本中的数据由多个字符组成的字符串作为分隔符，为了该需求，星环科技开发了多字符分隔符的功能。使用多个字符作为分隔符的建表语法为：

语法

```
CREATE [TEMPORARY] [EXTERNAL] TABLE <table_name> (<column_name> <data_type>, ...)  
[PARTITIONED BY ...]  
[CLUSTERED BY ...]  
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.MultiDelimitSerDe' ①  
WITH SERDEPROPERTIES ('input.delimited'='<delimiting_chars>') ②  
[STORED AS TEXTFILE]  
[LOCATION '<hdfs_path>']  
[TBLPROPERTIES ('<property_name>'='<property_value>', ...)];
```

① `org.apache.hadoop.hive.contrib.serde2.MultiDelimitSerDe` 中字母的大小写必须和我们在这里提供的一致。

② `input.delimited` 中字母的必须全部小写；`<delimiting_chars>` 处提供多字符分隔符。



Inceptor支持使用多字符分隔符做列分隔符或行换行符，但不支持使用多字符分隔符做ARRAY, STRUCT, MAP字段分隔符。

例 50. 使用多字符分隔符

已知有一份原始数据在HDFS上的 /user/alice/employee_multi 目录下，内容如下：

```
1@_@Alice
2@_@Bob
```

我们需要在建表时指定' @_@' 为分隔符：

```
CREATE TABLE employee_multi (id INT, name STRING) ROW FORMAT SERDE
'org.apache.hadoop.hive.contrib.serde2.MultiDelimitSerDe' WITH SERDEPROPERTIES
('input.delimited'='@_@') LOCATION '/user/alice/employee_multi';
```

建表完成后查看表中内容可得：

```
SELECT * FROM employee_multi;
+---+---+
| id | name |
+---+---+
| 1  | Alice |
| 2  | Bob  |
+---+---+
```

3.10. CSV表

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。建CSV表时，在[建表语句](#)中用 `STORED AS CSVFILE` 指定存储格式为CSV文件。另外，还可以在 `TBLPROPERTIES` 中自定义分隔符和NULL字符。下面我们进行详细的介绍。

3.10.1. 建CSV外表

语法

```
CREATE EXTERNAL TABLE <table_name> ①
(<column_name> <data_type>, <column_name> <data_type>, ...)
STORED AS CSVFILE ②
[LOCATION '<hdfs_path>'] ③
[TBLPROPERTIES ( ④
  ['field.delim'='<field_delimiter>'], ⑤
  ['line.delim'='<newline_char>'], ⑥
  ['serialization.null.format'=''], ⑦
  ['quote.delim'='<quote_delimiter>'], ⑧
  ['<property_name>'='<property_value>'], ...)]; ⑨
```

- ① 用 `EXTERNAL` 表示建外表。
- ② 指定表存储为CSVFILE (CSV文件)。
- ③ 指向HDFS上的一个目录。该路径可以是一个绝对路径，比如 `/user/alice/employee` 或者一个完整的URL: `hdfs://nameservice1/user/alice/employee`。安全模式下，执行该建表操作的用户必须是这个路径指向的目录或文件的owner。如果 `<hdfs_path>` 指向的目录不存在，Inceptor会尝试新建这个目录，但是安全模式下Inceptor可能没有在指定路径新建目录的权限，所以星环科技建议尽量避免让 `<hdfs_path>` 指

向不存在的目录。

- ④ 在 **TBLPROPERTIES** 中，可以指定CSV文件中的分隔符，见下。
- ⑤ '**field.delim**' 属性的值指定字段分隔符，默认值为 “,”。由 “<field_delimiter>” 分隔的字段会被解析为不同列中的字段。
- ⑥ **line.delim** 属性的值指定换行符，默认值为 “\n”。
- ⑦ **serialization.null.format** 属性的值指定NULL值字符，默认为空字段，也就是说Inceptor会认为文件中两个连续的两个字段分隔符中有NULL值。
- ⑧ '**quote.delim**' 属性的值指定用什么字符作为 **quote_delimiter**，默认值为 “””。这个字符的作用为：如果字段本身包含了字段分隔符、换行符或者NULL值字符作为数据的一部分，将该字段放在两个 **quote_delimiter** 字符之间可以让Inceptor将字段内部出现的字段分隔符和换行符作为数据处理。同时，在一对 **quote_delimiter** 包裹的字段内部，**quote_delimiter** 是自身的转义符：如果出现连续的两个 **quote_delimiter**，那么Inceptor会将第二个 **quote_delimiter** 作为数据的一部分处理，将第一个 **quote_delimiter** 作为第二个的转义符。
- ⑨ **TBLPROPERTIES** 中可以包含用户自定义属性。



- 目前，**field.delim**, **line.delim**, **quote.delim** 和 **serialization.null.format** 的值都只能是单个字符（一个CHAR），不支持多字符。
- 暂时不支持复杂数据类型MAP, STRUCT和ARRAY。

例 51. 建CSV外表

假设我们有这样一个CSV文件，在HDFS上的/user/alice/csv1目录下。文件中数据如下：

```
field1,"field2-part1
""field2-part2,
field3-part3", field3
2field1,"2field2-part1
2field2-part2,
2field3-part3", 2field3
```

对这个文件这样建一张外表：

```
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');
```

根据这里 **TBLPROPERTIES** 中的设置，Inceptor会这样处理原文件中的数据：

- 文件中有两条记录，第一条记录的结尾为 **field3**。
- 每条记录有三个字段：field1, field2和field3，字段间用“,”隔开。
- 第二个字段field2中包含了换行符“\n”和字段分隔符“,”分开的几个part，但是因为放在一对“”中间用于，它们同属一个字段。
- 第一条记录中的 **"field2-part2** 中包含两个连续的 "，那么第二个 " 是数据本身的一部分。

我们可以查看表中记录来验证：

```
SELECT col1 FROM csv_table;
2field1
field1

SELECT col2 FROM csv_table;
2field2-part1
2field2-part2,
2field3-part3
field2-part1
"field2-part2,
field3-part3

SELECT col3 FROM csv_table;
2field3
field3
```

因为上面的建表语句中我们使用的分隔符都是Inceptor的默认值，建表时直接省去 **TBLPROPERTIES ('field.delim'=',', 'quote.delim'='"', 'line.delim'='\n')** 可以达到同样的效果：

```
CREATE EXTERNAL TABLE csv_table
(
    col1 STRING,
    col2 STRING,
    col3 STRING
)
STORED AS CSVFILE
LOCATION '/user/alice/csv1';
```

例 52. CSV外表中的NULL值

假设我们有这样一份CSV文件，在HDFS上的/user/alice/csv2目录下，文件内容如下：

```
aaa,,ccc,"",ddd
fff,,hhh,"",jjj
```

对这个文件这样建外表：

```
CREATE EXTERNAL TABLE csv_table_2
(
    col1 STRING,
    col2 STRING,
    col3 STRING,
    col4 STRING,
    col5 STRING
)
STORED AS CSVFILE
LOCATION '/user/alice/csv2'
TBLPROPERTIES('serialization.null.format'='');
```

根据这里 **TBLPROPERTIES** 中的设置，Inceptor会这样处理原文件中的数据：

- 文件中有两条记录，第一条记录的结尾为 **ddd**。
- 每条记录有五个字段：字段间用“,”隔开。
- 第一个字段后出现了两个连续的“,”，所以第二个字段为NULL。
- 第四个字段中虽然出现了NULL字符，但是NULL字符包裹在一对“”中，所以第四个字段时空字段（长度为零的字段），但是不是NULL值。

我们进行一下验证：

```
SELECT col2 FROM csv_table_2;
NULL
NULL

SELECT LENGTH(col4) FROM csv_table_2;
0
0
```

使用上述方法建表后，CSV文件中的数据也随之导入Inceptor。星环科技不建议使用任何其他方式（如 **INSERT** 或 **LOAD**）向CSV表中导入数据。

3.11. ORC表

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。
- 在 **TBLPROPERTIES** 里需要加上 "**transactional**"="**true**"，以标识这是一个要用作事务操作的表。
- 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。

ORC事务表相对与Inceptor中的其他表支持更多CRUD（增删改）语法，包括：

- INSERT INTO … VALUES
- UPDATE
- DELETE
- MERGE INTO

本章将重点介绍 ORC事务表 的建表、导数据以及CRUD。

3.11.1. 将要使用的表

为了举例，我们将会用到下面四张表：

ta表，包含了人名和他们的年龄：

name	age
Zhang San	18
Li Lei	20
Han Meimei	20
Wang Wu	22

tg表，包含了人名和他们的gpa：

name	gpa
Xiao Ming	3.3
Li Lei	3.5
Lv Si	3.9
Han Meimei	4.0
Zhang San	3.2

th表，包含了人名和他们的gpa：

name	gpa
Xiao Hong	3.8
Lv Si	3.8
Liu Tao	3.0
Zhang San	2.8

t1表，两列字段字段分别为id和value。

id	value
254	6
230	8
237	82
283	89
211	179
205	91
291	14
222	86

3.11.2. 建ORC事务表

语法

```

//非分区表
CREATE TABLE <table_name> (<column> <data_type>, <column> <data_type>, ...)
CLUSTERED BY (<bucket_key>) INTO <n> BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");

//单值分区表 (Unique Value Partition)
CREATE TABLE <table_name> (<column> <data_type>, <column> <data_type>, ...)
PARTITIONED BY (<partition_key> <data_type>)
CLUSTERED BY (<bucket_key>) INTO <n> BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");

//范围分区表 (Range Partition)
CREATE TABLE <table_name> (<column> <data_type>, <column> <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>, ...),
    PARTITION [<partition_name_2>] VALUE LESS THAN(<key1_bound_value2>, <key2_bound_value2>, ...),
    ...
)
CLUSTERED BY (<bucket_key>) INTO <n> BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");

```

1. 分区表中的分区键partition_key不能和表中的列重复。



2. 在表结构中，分区键被系统排在非分区键之后。

关于更多单值分区和范围分区的内容，请参考[分区表](#)。

举例

```

[$host] SET transaction.type=inceptor;

[$host] CREATE TABLE ta (name STRING, age INT) CLUSTERED BY (age) INTO 2 BUCKETS STORED AS ORC
TBLPROPERTIES ("transactional"="true"); ①

[$host] CREATE TABLE tg (name STRING, gpa DOUBLE) CLUSTERED BY (gpa) INTO 4 BUCKETS STORED AS ORC
TBLPROPERTIES ("transactional"="true"); ②

[$host] CREATE TABLE test (a INT, b STRING, c DOUBLE) PARTITIONED BY (date STRING) CLUSTERED BY (a)
INTO 8 BUCKETS STORED AS ORC TBLPROPERTIES ("transactional"="true"); ③

[$host] create table t1(value int) partitioned by range(id int)
(
partition less1 values less than (1),
partition less10 values less than (10),
partition less100 values less than (100)
)
clustered by (value) into 5 buckets stored as orc TBLPROPERTIES ("transactional"="true"); ④

```

① 创建非分区ORC表。

② 创建非分区ORC表。

③ 创建单值分区ORC表。

④ 创建范围分区ORC表。



桶的个数对事务处理的性能有关键性的影响，我们建议您设置合理的个数，一般是CPU个数的倍数，并且每个桶平均的大小控制不要超过200MB或者一百万行记录。

3.11.3. ORC事务表的CRUD

ORC事务表比其他表支持更多CRUD（增删改）语法，包括：

- INSERT INTO ... VALUES

- UPDATE
- DELETE
- MERGE INTO

事务处理

- 下面将要介绍的ORC事务表CRUD操作必须在 **事务处理模式** (Transaction Mode) 下进行。默认情况下Inceptor关闭Transaction Mode，要对ORC表进行事务处理，需要通过下面的开关打开ORC表对应的Transaction Mode：

```
SET transaction.type = inceptor;
```

- Inceptor支持常见的 **BEGIN TRANSACTION/COMMIT/ROLLBACK** 事务处理语言 (TCL)。在不使用这些关键词的情况下默认为自动提交——每个CRUD操作都立即生效。下面在介绍CRUD时我们也默认使用自动提交。关于如何使用TCL来控制事务处理逻辑，请参考[事务控制语言 \(TCL\)](#)。

3.11.3.1. INSERT

Inceptor支持单条或者批量的往事务ORC表中插入数据，SQL语法完全兼容SQL99标准。

单条插入数据

语法

```
//非分区表
INSERT INTO table_name VALUES (value, value, ...)

//单值分区表
INSERT INTO table_name PARTITION (partition_key = value) VALUES (value, value, ...)

//范围分区表
INSERT INTO table_name VALUES (value, value, ...) ①
INSERT INTO table_name PARTITION partition_name VALUES (value, value, ...)②
```

① 对范围分区表进行全表范围的INSERT INTO时可以不指明分区键，此时INSERT面向全表范围。

② 对范围分区表进行INSERT INTO时指定插入的分区。



不支持对ORC事务表执行INSERT OVERWRITE。

举例

```
[$host] SET transaction.type=inceptor;

//向非分区表单条插入
[$host] INSERT INTO ta VALUES ('Zhang San', 18);

//向单值分区表单条插入
[$host] INSERT INTO test PARTITION (date = '20150612') VALUES (1, 'a', 1.0);

//向范围分区表单条插入
[$host] INSERT INTO t1 VALUES(0,5);
[$host] INSERT INTO t1 PARTITION less10 VALUES(0,5);
```

批量插入查询结果

语法

```
//非分区表
INSERT INTO TABLE table_name SELECT select_statement;

//单值分区表
INSERT INTO TABLE table_name PARTITION (partition_key = value) select_statement;

//范围分区表
INSERT INTO TABLE table_name SELECT select_statement; //全表范围
INSERT INTO table_name PARTITION partition_name SELECT select_statement; //指定分区
```

举例：

```
[$host] SET transaction.type=inceptor;

//向非分区表批量插入。建一张新表tag，这张表中包含ta和tg中同名的人的名字、年龄和gpa:
[$host] CREATE TABLE tag (name STRING, age INT, gpa DOUBLE) CLUSTERED BY (gpa) INTO 8 BUCKETS
STORED AS ORC TBLPROPERTIES ("transactional"="true");
[$host] INSERT INTO TABLE tag SELECT ta.name, age, tg.gpa FROM ta JOIN tg ON (ta.name = tg.name);

//向单值分区表批量插入
[$host] INSERT INTO TABLE test PARTITION (date = '20151031')SELECT * FROM sample LIMIT 3;

//向范围分区表批量插入
[$host] INSERT INTO t1(id, value) SELECT id,value FROM another_table;
[$host] INSERT INTO t1 PARTITION less10 SELECT id,value FROM another_table WHERE id < 10;
```

3.11.3.2. UPDATE

Inceptor支持单条或者批量的往ORC事务表中更新记录，并且可以更新任意列以及组合。

单条更新

语法

```
//非分区表以及范围分区表
UPDATE table_name SET column_name = value WHERE filter_statement

//单值分区表
UPDATE table_name PARTITION (partition_key = value) SET (column_name = value) WHERE filter_statement

//范围分区表
UPDATE table_name PARTITION partition_name SET (column_name = value) WHERE filter_statement
```

举例

```
[$host] SET transaction.type=inceptor;

//为非分区表单条更新。将tg中Zhang San的gpa改成3.1:
[$host] UPDATE tg SET gpa = 3.1 WHERE name = 'Zhang San';

//为单值分区表单条更新
[$host] UPDATE test PARTITION (date = '20150612') SET a = 2 WHERE b = 'a';

//为范围分区表单条更新
[$host] UPDATE t1 PARTITION less10 SET value = 100 WHERE id = 1;
```

用查询结果批量更新

语法

```
//非分区表
UPDATE table_name SET (column, column, ...) = (SELECT select_statement WHERE filter_statement)

//分区表
UPDATE table_name PARTITION (partition_key = value) SET (column, column, ...) = (SELECT
select_statement WHERE filter_statement)

//范围分区表
UPDATE table_name PARTITION partition_name SET (column, column, ...) = (SELECT select_statement
WHERE filter_statement)
```

注意

- 请确保等号左右的列数相同并且对应列的数据类型也相同。
- 在用查询往事务表中更新数据时，必须保证查询的结果的行数和表中被更新的行数一致，否则整个更新的逻辑就会有错误。因此，Inceptor要求 (SELECT select_statement WHERE filter_statement) 中必须要有过滤条件filter_statement，并且这个filter_statement必须建立和被更新表的关联条件。

举例

```
[$host] SET transaction.type=inceptor;

//向非分区表批量UPDATE。从tg中选出与tag表中name相同的记录，将tag的对应记录的age设定为23，gpa值设定为tg中对应记录的
gpa值。
[$host] UPDATE tag SET (age, gpa) = (SELECT 23, gpa FROM tg WHERE (tg.name = tag.name));

//向单值分区表批量UPDATE
[$host] UPDATE test PARTITION (date = '20151031') g SET (a, b) = (SELECT e, f FROM sample s WHERE
g.a = s.e);

//向范围分区表批量UPDATE
[$host] UPDATE t1 PARTITION less10 SET (value) = (SELECT value FROM another_table WHERE t1.id =
another_table.id);
```

举例：嵌套子查询的UPDATE

在子查询中找到tg和th中名字相同的人，如果他也在tag表中，那么将他在tag中的年龄设置为18，gpa设置为tg中的gpa。

```
[$host] SET transaction.type=inceptor;
[$host] UPDATE tag SET (age, gpa) = (SELECT 18, x.gpa FROM (SELECT g.name, g.gpa gpa FROM th h JOIN
tg g ON g.name=h.name) x WHERE tag.name = x.name);
```

反例：错误的UPDATE使用方法

```
[$host] UPDATE tag SET (age, gpa) = (SELECT 20, gpa FROM tg);
```

说明

这种写法有两个问题：

- 无法确定两个表记录数是否一样，只要不一样就会出错。
- 无法确定th中哪条记录会更新tag中的哪条记录。在分布式系统中，这个对应关系完全是无序的。

Inceptor会对这种SQL直接编译报错。

3.11.3.3. DELETE

Inceptor支持对ORC事务表的部分或者全部删除。

语法：删除部分记录

```
//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
```

语法：删除全部记录

```
//删除表中全部记录（对分区表和非分区表都适用）
DELETE FROM table_name

//从单值分区表中删除一个分区的全部记录
DELETE FROM table_name PARTITION (partition_key = value)

//从范围分区表中删除一个分区的全部记录
DELETE FROM table_name PARTITION partition_name
```

举例：删除部分记录

```
[$host] SET transaction.type=inceptor;

//从非分区表中删除部分记录
[$host] DELETE FROM ta WHERE name = 'Zhang San';

//从单值分区表中删除分区中部分记录
[$host] DELETE FROM test PARTITION (date = '20150612') WHERE a = 1;

//从范围分区表中删除分区中部分记录
[$host] DELETE FROM t1 partition less10 WHERE id = 254;
```

举例：删除全部记录

```
[$host] SET transaction.type=inceptor;

//从表中删除全部记录（对分区表和非分区表都适用）
[$host] DELETE FROM ta;

//从单值分区表中删除一个分区的全部记录
[$host] DELETE FROM test PARTITION (date = '20151031');

//从范围分区表中删除一个分区的全部记录
[$host] DELETE FROM t1 PARTITION less100;
```

3.11.3.4. MERGE INTO

MERGE语句用来合并UPDATE和INSERT语句。通过MERGE语句，根据一张表或子查询的连接条件对另外一张表进行查询，连接条件匹配上的进行UPDATE，无法匹配的执行INSERT。这个语法仅需要一次全表扫描就完成了全部工作，执行效率要高于INSERT+UPDATE。其中，对单一值分区表必须指定目标分区；而对于范围分区表可以指定也可以不指定。

语法

```
//非分区表
MERGE INTO table
USING { table | view | subquery } alias
ON ( condition )
WHEN MATCHED THEN merge_update_clause
WHEN NOT MATCHED THEN merge_insert_clause

//单值分区表
MERGE INTO table
PARTITION (partition_key = value)
USING { table | view | subquery } alias
ON ( condition )
WHEN MATCHED THEN merge_update_clause
WHEN NOT MATCHED THEN merge_insert_clause

//范围分区表（指定目标分区时需要添加方括号中的内容）
MERGE INTO table
[PARTITION partition_name]
USING { table | view | subquery } alias
ON ( condition )
WHEN MATCHED THEN merge_update_clause
WHEN NOT MATCHED THEN merge_insert_clause
```

注意，MERGE的源表必须有化名。

举例（非分区表）：仅更新，不插入新纪录

找出ta和tag中年龄相同的人，如果他的在tag中的年龄小于他在ta中的年龄加二，那么将他在tag中的年龄改为他在ta中的年龄加二，gpa改为4.0。

```
[$host] SET transaction.type=inceptor;
[$host] MERGE INTO tag USING ta a ON (a.name = tag.name) WHEN MATCHED THEN UPDATE SET age =
a.age+2, gpa=4.0 WHERE tag.age < (a.age+2);
```

举例（非分区表）：满足条件的情况更新，不满足条件下插入新纪录

找出ta和tg中同名的人，如果他们在tag中有同名记录而且在tag中的年龄大于在ta中的年龄加一，则将他们在tag中的年龄更新为ta中的年龄加一，gpa更新为tg中的gpa+0.1。如果ta和tg中同名的人在tag中没有同名记录，则将他们名字、ta中的年龄和tg中的gpa作为新的记录插入tag。

```
[$host] SET transaction.type=inceptor;
[$host] MERGE INTO tag d USING (SELECT a.name name, a.age age, g.gpa gpa FROM ta a JOIN tg g ON
a.name = g.name) s ON (s.name = d.name) WHEN MATCHED THEN UPDATE SET age = s.age+1, gpa = 0.1+s.gpa
WHERE d.age > s.age+1 WHEN NOT MATCHED THEN INSERT (name, age, gpa) VALUES (s.name, s.age, s.gpa);
```

举例（单值分区表）

```
[$host] SET transaction.type=inceptor;
[$host] MERGE INTO sample PARTITION (date = '20150702') s USING test t ON (s.a = t.a) WHEN MATCHED
THEN UPDATE SET c = t.c+10.0 WHEN NOT MATCHED THEN INSERT (a, c) VALUES (t.a, t.c);
```

举例（范围分区表）

```
[$host] MERGE INTO t1 d USING another_table s ON (d.id = s.id) WHEN MATCHED THEN UPDATE SET value =
s.value WHEN NOT MATCHED THEN INSERT(id, value) VALUES(s.id, s.value);
[$host] MERGE INTO t1 d PARTITION less10 USING another_table s ON (d.id = s.id) WHEN MATCHED THEN
UPDATE SET value = s.value WHEN NOT MATCHED THEN INSERT(id, value) VALUES(s.id, s.value);
```

3.11.3.5. CRUD以子查询为目标

Inceptor允许以子查询为增删改（CRUD）的目标。有了这样的支持，用户可以通过过滤目标范围，减少对目

标的扫描，从而优化语句的执行。具体优化方式请参见《Inceptor优化手册 3.7节》。注意，CRUD目标为子查询时，不允许指定分区。

注意，当子查询作为CRUD的目标时，请务必向该子查询赋予别名。

举例

```
--UPDATE
--非分区表
[$host] UPDATE (SELECT name, gpa FROM tg WHERE name = 'Zhang San') a SET name = 'Li Si' WHERE 1=1;

--范围分区表
[$host] UPDATE (SELECT id, value FROM t1 WHERE value > 100) a SET value = 100 WHERE id > 250;

--DELETE
--非分区表
[$host] DELETE FROM (SELECT name, age FROM ta WHERE age > 20) a WHERE name <> 'Han Meimei';

--范围分区表
[$host] DELETE FROM (SELECT value FROM t1 WHERE id in (230, 222)) a WHERE value > 100;

--MERGE INTO
--非分区表
[$host] MERGE INTO (SELECT * FROM tg WHERE gpa >= 3.0) g USING th h ON (g.name = h.name) WHEN MATCHED THEN UPDATE SET g.gpa = g.gpa + 0.1;

--范围分区表
[$host] MERGE INTO (SELECT * FROM t1 WHERE value = 6) d USING (SELECT * FROM another_table WHERE value = 6) s ON (d.id = s.id) WHEN MATCHED THEN UPDATE SET d.value = d.value + 1;
```

- 1. CRUD中的子查询只允许涉及一张表，且该表一定为CRUD操作的目标表。暂不支持目标子查询涉及单值分区表。
- 2. 子查询中不能出现GROUP BY、ORDER BY、CLUSTERED BY 和SORT BY、CLUSTER BY操作。
- 3. UPDATE/INSERT的列必须能够映射到目标表中的列，对目标表的列进行算数变换后不能用于 UPDATE/INSERT，但可以作为过滤条件。例如，“ss_sold_date_sk”是属于store_sales的一个字段，语句“MERGE INTO (SELECT ss_sold_date_sk, ... UPDATE ...)”是允许的，但是不允许“MERGE INTO (SELECT ss_sold_date_sk + 1, ... UPDATE ...)”。



3.12. Holodesk表

3.12.1. 简介

Holodesk是用于应对海量数据OLAP高性能分析查询难题的一款产品，它着力于交互式分析中即时查询效率的提高且能够保证扩展性与稳定性。Transwarp Holodesk 通过 Zookeeper 来管理元数据，从而避免因为单点故障而导致的数据丢失，数据checkpoint 在 HDFS 中。服务在故障恢复之后，Holodesk 能够通过 Zookeeper 中的信息自动重建数据与索引，因此有很高的可靠性。

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。

Holodesk适用于两种存储介质：内存和SSD。在机器同时配置了内存和SSD的情况下，请优先使用SSD。在使用Holodesk之前，必须在Tranwarp Manager的Inceptor设置页面中合理配置以下三个相关资源：ngmr.fastdisk.dir、ngmr.fastdisk.size以及ngmr.localedir，分别代表Holodesk的实际所位置、Holodesk占用资源的比例、Shuffle数据的本地存放位置。

3.12.2. Holodesk建表语法

普通建表

创建普通Holodesk空表的语法为：

```
CREATE TABLE <holodesk_table_name> (
  <column_name1> <DATATYPE1>,
  <column_name2> <DATATYPE2>,
  <column_name3> <DATATYPE3>,
  ...
) STORED AS HOLODESK;
```

例 53. Holodesk普通建表

```
CREATE TABLE holodeskEmployee(
  ID INT,
  Region STRING,
  Sex VARCHAR(4),
  Department STRING,
  Salary DECIMAL
) STORED AS HOLODESK;
```

创建普通Holodesk，并导入数据的语法为：

```
CREATE TABLE <holodesk_table_name> STORED AS HOLODESK
AS SELECT ... FROM ...;
```

例 54. Holodesk普通建表，并从employee表导入数据

```
CREATE TABLE holodeskEmployee STORED AS HOLODESK
AS SELECT * FROM employee;
```

Index和Cube

对于过滤率高或聚合率高的场景，我们提供Cube、Index两种手段来帮助优化业务执行。在运用Index以及Cube加速的过程中，Index与Cube的定义是关键。Index字段与Cube组合字段的选取，主要来自于对业务逻辑中维度的取舍，以及对度量条件的选择等。为达到较好的优化效果，Index字段通常为过滤字段，Cube字段通常为聚合字段。Index和Cube都是在建表时创建。

1. Index

创建Index的语法。

```

CREATE TABLE <holodesk_table_name>(
    <column_name1> <DATATYPE1>,
    <column_name2> <DATATYPE2>,
    ...
) STORED AS HOLODESK
TBLPROPERTIES (
    'holodesk.index' = '<column_name_index1>, <column_name_index2>, ...'
);

```

2. Cube

创建Cube的语法。

```

CREATE TABLE <holodesk_table_name>(
    <column_name1> <DATATYPE1>,
    <column_name2> <DATATYPE2>,
    ...
) STORED AS HOLODESK
TBLPROPERTIES (
    'holodesk.dimension' = '<column_name_a_dim1>, <column_name_a_dim2>, ... | 
    <column_name_b_dim1>, <column_name_b_dim2>, ... | ...
);

```

例 55. 针对某应用场景设计Holodesk表的创建

假设对于holodeskEmployee表将有如下的查询语句：

```

SELECT Sex, Region, COUNT(ID), AVG (Salary)
FROM holodeskEmployee
WHERE Department = 'IT'
GROUP BY Sex, Region
ORDER BY Sex, Region;

```

如果该语句对holodeskEmployee的聚合力与过滤力较高，建议在建表时为“Department”字段建立Index，并根据（“Sex”，“Region”）建立Cube，可提升查询效率。建表语句如下：

```

CREATE TABLE holodeskEmployee
STORED AS HOLODESK
TBLPROPERTIES (
    'holodesk.index' = 'Department',
    'holodesk.dimension' = 'Sex, Region'
) AS SELECT * FROM Employee;

```

Global Index

Global Index对于过滤率高的情况有很好的优化效应，因此我们通常对过滤率高的待过滤的字段建立Global Index。Global Index和普通Holodesk Index的功能并不冲突，可以两个共存。Global Index从创建到删除的步骤如下：

1. 创建Holodesk空表

```

CREATE TABLE <holodesk_table_name>(
    <column_name1> <DATATYPE1>,
    <column_name2> <DATATYPE2>,
    ...
) STORED AS HOLODESK
TBLPROPERTIES (
    'holodesk.index' = '<column_name_index1>, <column_name_index2>, ...',
    'holodesk.dimension' = '<column_name_a_dim1>, <column_name_a_dim2>, ... | 
    <column_name_b_dim1>, <column_name_b_dim2>, ... | ...'
);

```

2. 设置Reducer数目

Holodesk Global Index分不同文件存放，设置Reducer数量就是用于设定被分割的文件数目。假设欲将Global Index的数量设置为m，正确的语法为：

```
SET mapred.reduce.tasks = m;
```

3. 创建Global Index

```
CREATE GLOBAL INDEX <holodesk_index_name>
ON <holodesk_table_name>( <global_index_column1>, <global_index_column2>, ... );
```

Global Index目前只支持如下字段类型：

- intType
- doubleType
- floatType
- stringType(n)
- varchar32Type
- longType
- byteType
- shortype
- booleanType

4. 导入数据

```
INSERT INTO <holodesk_table_name>
SELECT * FROM table_B;
```

5. 删除Global Index

```
DROP INDEX <holodesk_index_name> ON <holodesk_table_name>;
```

例 56. 创建表holodeskEmployeeGI，并为其创建Global Index

假设employee表在Department维度有较高的过滤率，而且某语句将在Department字段上做过滤限制。需要对employee创建一个holodesk表，并在Department字段上创建Global Index。过程如下：

1. 建表

建表时对Department字段创建普通Index

```
CREATE TABLE holodeskEmployeeGI(
    ID INT,
    Region STRING,
    Sex VARCHAR(4),
    Department STRING,
    Salary DECIMAL
) STORED AS HOLODESK
TBLPROPERTIES ('holodesk.index' = 'Department');
```

2. 设置Reducer数量为5

```
SET mapred.reduce.tasks = 5;
```

3. 对Department字段创建Global Index

```
CREATE GLOBAL INDEX holoEmployeeGlobalIndex
ON holodeskEmployeeGI (Department);
```

4. 从employee导入数据

```
INSERT INTO holoEmployeeGI
SELECT * FROM employee;
```

即创建成功，需删除时请执行下面的语句。

5. 删除Global Index

```
DROP INDEX holoEmployeeGlobalIndex ON holodeskEmployeeGI;
```



更多关于Holodesk的介绍请用户参考《Holodesk使用手册》。该手册详细介绍了Holodesk参数的配置原则、Holodesk的常见优化方法并提供了若干优化案例以方便用户理解与应用。

3.13. 基于定宽文本文件建外表

3.13.1. 简介

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式的分隔符，每条数据之间有行分隔符。例如，某个给定文件data.txt，其中的部分内容如下：

5243508000RMC
582075938PRAN

每行数据其实包含了三个字段：前四个字符代表id，中间五个字符代表num，末尾四个字符代表namecode。这类数据不依靠显式标识区分字段，而是靠宽度默认切分。

对于这种情况，我们提供基于定宽文本文件建外表的支持，允许用户定义每个定长字段的长度，系统自动从每行记录截取设定长度，将之对应至相应字段。

3.13.2. 建表语法

```

CREATE EXTERNAL TABLE test(
    <colname1> <DataType1>,          ①
    <colname2> <DataType2>,
    ...
    <colnameN> <DataTypeN>,
)
STORED AS FWCFFILE
LOCATION "<hdfs_address>"        ②
TBLPROPERTIES(
    'fields.width' = '<len1>, <len2>, ... , <lenN>' ③
    [, 'padding.type'='right|left'] ④
    [, 'padding.char'=' ']          ⑤
    [, 'serialization.encoding'='<encoding_type>'] ⑥
    [, 'serialization.null.format'='\N']); ⑦

```

① 表的各个字段和类型。

② 数据源在HDFS上的存放位置。

③ 每个定宽字段的长度，例如<len1>对应第一个字段的长度，<len2>对应第二个，依此类推。

④ 字段的实际数据可能短于为其对应字段划分的长度，这时必须要在源数据中通过填充字符使其二者等长。
用户可以在数据左侧或者右侧填充字符，并通过此属性向系统告知填充位置。该属性有两个可选值：right和left。

⑤ 使用的填充字符。

⑥ 文本文件编码方式。

⑦ 源数据中代表null值的字符。

3.13.3. 建表示例

某一用gb18030编码的文本文件company.txt，内容如下：

CCC100004886773	上海石油*****有限公司	0113226944-X	20151231000000
CCC100004886779	上海吱吱****有限公司	0131213516-6	20151231000000
CCC100004886777	巨庸*****有限公司	0171785900-7	20151231000000
CCC100004886775	承席****有限公司	0174618322-9	20151231000000
CCC100004886781	上海市*****服装店	01L4404776-X	20151231000000

该文件存放在HDFS的 “/temp/fwcf/data” 目录下。它包含五条记录，实际提供5个字段数据：源数据中第三列含有两个字段，例如，“0113226944-X”中，“01”是一个字段，“13226944-X”属于后一个字段；其余每个列对应一个字段。该文件采用的填充字符为‘ ’（空格），类型为右侧插入，每个字段的字节数分别为：20、33、2、19、14。NULL的表示方式为’\N’。

根据以上对应规则，我们使用如下建表语句对该数据建外表：

```

CREATE EXTERNAL TABLE test(
    col1 STRING,
    col2 STRING,
    col3 STRING,
    col4 STRING,
    col5 STRING,
    col6 STRING)
STORED AS FWCFILE
LOCATION "/temp/fwcc/data"
TBLPROPERTIES(
    'fields.width'='20,33,2,19,14',
    'padding.type'='right',
    'padding.char'=' ',
    'serialization.encoding'='gb18030',
    'serialization.null.format'='\N'
);

```

对该表进行查询的返回结果如下：

```

select col1, col2, col3, col4, col5 from test;

CCC100004886773 上海石油*****有限公司 01 13226944-X 20151231000000
CCC100004886779 上海吱吱*****有限公司 01 31213516-6 20151231000000
CCC100004886777 巨庸*****有限公司 01 71785900-7 20151231000000
CCC100004886775 承席*****有限公司 01 74618322-9 20151231000000
CCC100004886781 上海市*****服装店 01 L4404776-X 20151231000000

```

- 注意，用户应保证源数据中对应同一字段（或者被填充之后）的数据的宽度一致。而且每个字段对应数据的长度（加上填充字符）必须同为该字段划分的宽度相等，否则将报错。

例如，给定如下数据：

7924570RMC***
*
659381PRAA***
*

其中每行数据前6个字符组成第一个字段；剩余部分组成第二个字段，第二个字段右侧被字符“*”填充，宽度为8。因此正确的建表语句中‘fields.width’的取值只允许为：



'fields.width' = '6,8'

总长度不应超过14，例如若做如下设置则出错：

'fields.width' = '6,10'

- 目前尚不支持如下情况：

- 换行符都必须是单字符，不支持多字符。
- 暂不支持复合数据类型，如STRUCT、ARRAY、MAP。
- 暂不支持插入操作，只能单纯作为数据源。

- 建议文件大小在256M以内，能保证有较好性能。

3.14. 分区表



Inceptor中只支持对TEXT表、ORC表和CSV表分区，不支持对Holodesk表分区。

在逻辑上，分区表和未分区表没有区别；在物理上，分区表中的数据按分区键的值放在HDFS上表目录下的对应子目录中，一个分区对应一个子目录。例如一张表user_acc_level按acc_level分区：

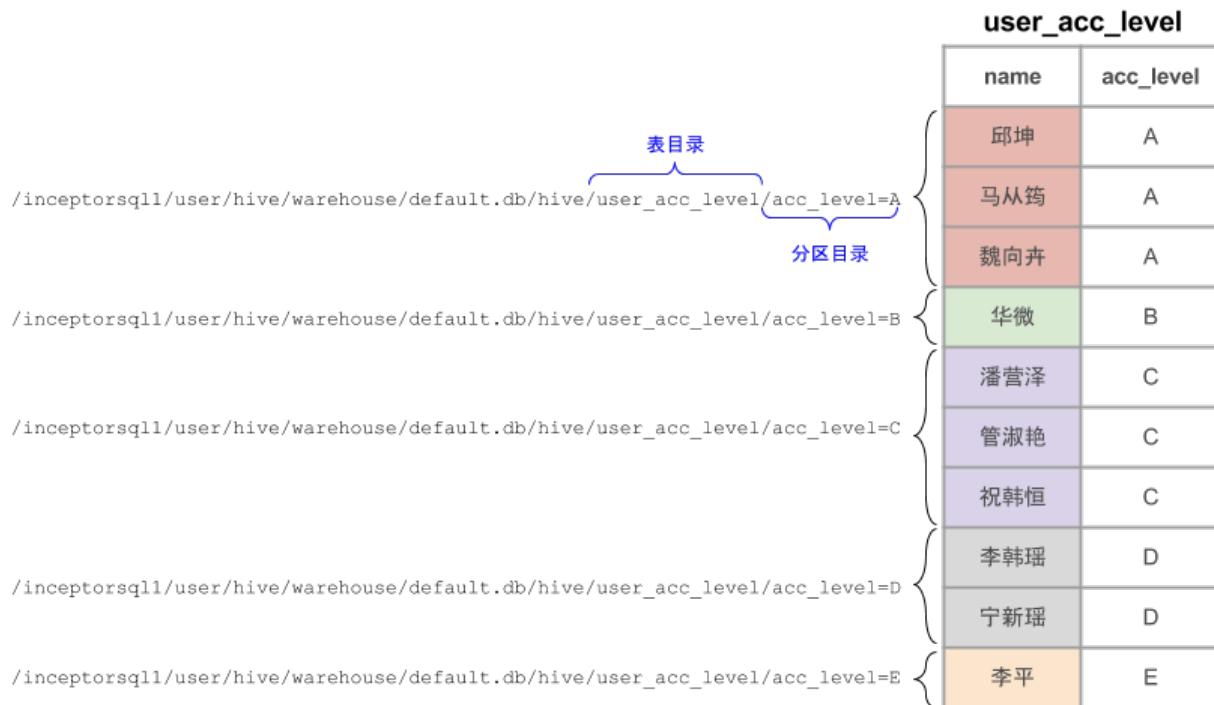


图 2. 分区表示例

假设我们要查询所有acc_level=A的用户，Inceptor会只扫描acc_level=A目录下的数据，在数据量大的情况下可以显著提升查询效率。Inceptor支持多层次分区——使用多个分区键分区。多层次分区的表目录下会有各级分区子目录。

一张表中的数据还可以通过 **分桶** 放在HDFS上不同的子目录中，一张表可以同时分区和分桶。您可以参考“**分桶表**”了解更多相关信息。

Inceptor支持对表的单值分区和范围分区：

- **单值分区**：一个分区对应分区键的一个值。
- **范围分区**：一个分区对应分区键值的一个范围（区间）。

分区在创建表时完成，也可以通过 **ALTER TABLE** 来添加或者删除分区。本章我们分别介绍单值分区表和范围分区表的创建、修改以及分区表数据的导入。

3.14.1. 单值分区表

3.14.1.1. 建表

单值分区表的建表方式有两种：直接定义列和 **CREATE TABLE LIKE**。注意，单值分区表不能用 **CREATE**

TABLE AS SELECT 建表。

语法：直接定义列

```
CREATE [TEMPORARY] [EXTERNAL] TABLE <table_name>
(<column_name> <data_type>, <column_name> <data_type>, ...)
PARTITIONED BY (<partition_key> <data_type>, ...) ①
[CLUSTERED BY ...] ②
[ROW FORMAT ...] ③
[STORED AS TEXTFILE|ORC|CSVFILE] ④
[TBLPROPERTIES ('<property_name>'='<property_value>', ...)];
```

- ① 在这里指定分区键名称和分区键的数据类型，一张表可以有多个分区键，分区键不能和表中的列重复。
- ② 分桶子句，这里不详细描述，请参考“分桶表”章节。
- ③ 当表为TEXT表时，指定行格式的关键字，这里不详细描述，请参考“TEXT表”章节。
- ④ 在这里指定存储格式，TEXT表、ORC表和CSV表可以分区，Holodesk表不能分区。

例 57. 创建单值分区表

建一张名为user_acc_level的表，分区键为acc_level：

```
CREATE TABLE user_acc_level (name STRING) PARTITIONED BY (acc_level STRING);
```

分区表建成后，分区键就可以作为一个“伪”列在查询中使用。也正因为如此，同一个表的分区键和列 **不能重名**。

例 58. 创建单值分区表的错误方式

如果建分区表时Inceptor报这样的错：`Error: Error while processing statement: FAILED: Error in semantic analysis: Column repeated in partitioning columns (state=,code=10)`，那么您在建表时，一定是发生了分区键和列重名的情况，例如：

```
CREATE TABLE demo (a INT, b STRING) PARTITIONED BY (a INT);
Error: Error while processing statement: FAILED: Error in semantic analysis: Column repeated in
partitioning columns (state=,code=10)
```

例 59. 创建多层分区表

建一张user_loc表，有两个分区键：acc_level和loc：

```
CREATE TABLE user_loc (name STRING) PARTITIONED BY (acc_level STRING, loc STRING);
```

语法：CREATE TABLE LIKE

```
CREATE [TEMPORARY] [EXTERNAL] TABLE <table_name> LIKE <partitioned_table>;
```

这样创建的 `<table_name>` 会和 `<partitioned_table>` 一样的列和分区键。

3.14.1.2. 导入数据

和非分区表一样，向分区表导入数据可以采用 `INSERT ... SELECT`（插入查询结果）或者 `LOAD DATA INPATH`（将文件中的数据加载进表中）。其中，星环科技 建议 采用 `INSERT ... SELECT` 方法，不建议 采用 `LOAD DATA INPATH` 方法。

语法：向分区表载入文件中的数据（不推荐使用）

```
LOAD DATA [LOCAL] INPATH '<path>' ①
[OVERWRITE] INTO TABLE <tablename> ②
PARTITION (<partition_key>=<partition_value>, ...); ③
```

- ① `<path>` 可以指向一个文件也可以指向一个目录。`[LOCAL]` 是本地路径选项，加上该选项后 `<path>` 是 Inceptor Server 所在节点的本地目录。如果用 HDFS 上的路径，当 `<path>` 指向一个文件时，Inceptor 会将文件移入表在 Inceptor 上的目录中；指向一个目录时，Inceptor 会将目录下所有的文件移入表在 HDFS 上的目录中。如果用本地路径，Inceptor 将文件或者目录中的数据拷贝到表在 HDFS 上的目录中。使用本地数据时，`<path>` 必须是绝对路径；使用 HDFS 上的数据时，`<path>` 可以是绝对路径或者相对路径（相对 `/user/<user_name>`）。`<path>` 不能有子目录。在安全模式下，用户需要设置好合适的 `<path>` 的权限，否则将载入失败，所以星环科技 不建议 这种导入方式。
- ② `[OVERWRITE]` 选项会将目标分区已有的内容会被导入的文件覆盖。不加 `[OVERWRITE]` 选项则导入的文件不覆盖已有文件，但是如果目标表或者分区中存在文件和被导入的文件重名，那么原先的文件会被新文件覆盖。
- ③ 指定目标分区。

下面我们详细介绍如何使用 `INSERT ... SELECT` 向分区表内导数据。

3.14.1.2.1. 单次插入

语法：向分区表插入查询结果

```
INSERT OVERWRITE TABLE <table_name> ①
PARTITION (<partition_key1>=<partition_value>[, <partition_key2>=<partition_value>, ...]) ②
[IF NOT EXISTS] ③
SELECT <select_statement> FROM <source>; ④

INSERT INTO TABLE <table_name> ⑤
PARTITION (<partition_key>=<partition_value>[, <partition_key>=<partition_value>, ...])
SELECT <select_statement> FROM <source>;
```

- ① `INSERT OVERWRITE` 插入的数据会覆盖分区中原有数据。
- ② 指定目标分区。
- ③ 加上 `[IF NOT EXISTS]` 选项，那么在目标分区中已经有数据的情况下，插入不会发生。
- ④ `<select_statement>` 中的列必须和 `<table_name>` 中 非分区键 的列数量相同，且数据类型一一对应。
- ⑤ `INSERT INTO` 不覆盖分区中原有数据，目标分区如果有数据，那么原数据和新数据会同时存在。



向单值分区表导入数据和向非分区表导入数据有一个很重要的区别：除了[动态分区插](#)入，向单值分区表导入数据时 必须 指定目标表中的分区。同时，用户必须自己确保数据插入的正确性，也就是合适的数据插入合适的分区中。

例 60. 向分区表插入查询结果

以user_info表为源，向[创建单值分区表](#)中的user_acc_level的 acc_level='A' 的分区插入数据：

```
INSERT OVERWRITE TABLE user_acc_level PARTITION (acc_level='A') SELECT name FROM user_info
WHERE user_info.acc_level = 'A';
```

3.14.1.2.2. 多次插入

使用一个数据源可以将多个查询结果插入不同表以及不同表的不同分区中，目标中表和分区可以同时使用。

语法

```
FROM <source>
  INSERT (OVERWRITE|INTO) TABLE <table_name1> [PARTITION (<partkey>=<val>...)] [IF NOT EXISTS]
  SELECT <select_statement1>
  [INSERT (OVERWRITE|INTO) TABLE <tablename2> [PARTITION (<partkey>=<val>...)] [IF NOT EXISTS]]
  SELECT <select_statement2>
  [INSERT (OVERWRITE|INTO) TABLE <tablename3> [PARTITION (<partkey>=<val>...)] [IF NOT EXISTS]]
  SELECT <select_statement3>
  ...
  ;
```



除了最后一个 `<select_statement>` 之外，如果 `<select_statement>` 以一个列名结尾，那么该列必须有化名，否则下一个 `INSERT` 关键字将被当做这个 `<select_statement>` 最后一列的列化名处理，导致语义错误。

例 61. 用一个源插入多个分区或表

```
FROM user_info
INSERT OVERWRITE TABLE user_acc_level PARTITION (acc_level='B') SELECT name WHERE
user_info.acc_level='B'
INSERT INTO TABLE user_cid PARTITION (cid = '14*****7') SELECT name WHERE
user_info.citizen_id = '14*****7'
INSERT INTO TABLE user_bank_acc SELECT name, bank_acc;
```

3.14.1.2.3. 动态分区插入

前两种插入分区表的方法要求在插入数据的时候指定目标分区，那么如果数据中分区键的取值很多，就需要很多 `INSERT ... PARTITION(<partition_key>=<value>)` 语句来完成数据导入。动态分区插入 功能让Inceptor在插入数据时动态地判断数据的目标分区，减少了用户的工作量。介绍动态分区插入的用法之前，我们先区分两个概念：

- 静态分区键（Static Partition Key, SPK）：由用户在插入数据指令中手动指定分区的分区键。
- 动态分区键（Dynamic Partition Key, DPK）：Inceptor在执行插入指令时动态判断分区的分区键。

一张表可以同时被静态分区键和动态分区键分区。但是，建表时，动态分区键必须出现在所有静态分区键之后，这是因为HDFS上动态分区下目录的不能有静态分区的子目录。

语法：建动态分区表

```
CREATE TABLE <table_name> PARTITIONED BY ([<spk> <data_type>, ... ,] <dpk> <data_type>, [<dpk>
<data_type>, ...]);
```

语法：动态分区插入

```
SET hive.exec.dynamic.partition=true; ①
INSERT OVERWRITE TABLE <table_name>
PARTITION ([<spk>=<value>, ..., ] <dpk>, [..., <dpk>]) ②
SELECT <select_statement> FROM <from_statement>; ③
```

- ① 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用该行指令手动打开。
- ② PARTITION(...) 子句用于指定分区。静态分区键要用 <spk>=<value> 指定分区值；动态分区只需要给出分出分区键名称 <dpk>。动态分区键的给出必须出现在 PARTITION(...) 子句中的最后。
- ③ <select_statement> 中要插入的动态分区键的列必须出现在 <select_statement> 中的最后，并且出现顺序和对应的动态分区键在 PARTITION(...) 子句中出现的顺序一致。

例 62. 动态分区插入

user_al_cid是一个由al和cid两个分区键分区的表。其中al是静态分区键，cid是动态分区键。user_al_cid的建表语句如下：

```
CREATE TABLE user_al_cid(name STRING) PARTITIONED BY (al STRING, cid STRING);
```

现在我们采用动态分区插入向user_al_cid插入user_info中的数据。

```
SET hive.exec.dynamic.partition=true;
INSERT OVERWRITE TABLE user_al_cid PARTITION (al='A', cid) SELECT name, citizen_id FROM
user_info WHERE user_info.acc_level = 'A';
```

例 63. 多次插入中包含有动态分区插入

```
SET hive.exec.dynamic.partition=true;
FROM user_info
INSERT OVERWRITE TABLE user_al_cid PARTITION (al='B', cid) SELECT name, citizen_id WHERE
user_info.acc_level = 'B';
INSERT OVERWRITE TABLE user_acc_level PARTITION (acc_level = 'C') SELECT name WHERE
User_info.acc_level = 'C';
```

3.14.1.3. 其他单值分区表DDL和DML

3.14.1.3.1. 清空分区中的数据

语法

```
TRUNCATE TABLE <table_name> PARTITION (<partition_key> = <value>);
```

该操作会将分区目录下的数据清空，但是会保留分区目录。

3.14.1.3.2. 删除分区

语法

```
ALTER TABLE <table_name> DROP PARTITION (<partition_key>=<value>);
```

该操作会将整个分区目录删除。

3.14.1.3.3. 添加分区

语法

```
ALTER TABLE <table_name> ADD PARTITION (<partition_key>=<value>);
```

该操作会在表目录下新建这个分区的目录。

3.14.1.3.4. 重命名分区

语法

```
ALTER TABLE <table_name> PARTITION (<partition_key>=<value>) RENAME TO PARTITION (<partition_key>=<new_value>);
```

注意，新的分区名不能和已有分区重名。

重命名分区

```
ALTER TABLE user_acc_level PARTITION (acc_level = 'F') RENAME TO PARTITION (acc_level = 'H');
```

3.14.2. 范围分区表

本节介绍范围分区表的建表、导数据以及其他相关操作。

3.14.2.1. 建表

范围分区表只能通过直接定义列来建表。

语法

```
CREATE [TEMPORARY] [EXTERNAL] TABLE <table_name>
(<column_name> <data_type>, <column_name> <data_type>, ...)
PARTITIONED BY RANGE (<partition_key> <data_type>, ...) ①
  (PARTITION [<partition_name>] VALUES LESS THAN (<cutoff>), ②
  [PARTITION [<partition_name>] VALUES LESS THAN (<cutoff>),
  ...
  ]
  PARTITION [<partition_name>] VALUES LESS THAN (<cutoff>|MAXVALUE) ③
  )
[CLUSTERED BY ...]
[ROW FORMAT ...]
[STORED AS TEXTFILE|ORC|CSVFILE]
[TBLPROPERTIES ('<property_name>'='<property_value>', ...)];
```

① 范围分区的关键字为 **PARTITIONED BY RANGE**。

② 使用范围分区时，可以建表时使用 **PARTITION VALUES LESS THAN <cutoff>** 指定表中的分区（**<cutoff>** 为该分区的区间上限），也可以在建表后使用 **ALTER TABLE ADD PARTITION** 添加分区，请参考本节的“修改表”部分内容。**PARTITION VALUES LESS THAN <cutoff>** 的定义方式规定了范围分区表的范围区间总是头尾相连的。每个分区都可有一个唯一的、用户自定义的 **<partition_name>** 作为分区名，这个分区名将是HDFS上分区子目录的目录名，如果不定义分区名，Inceptor会自动将分区命名为 **<partition_key>_less_than_<cutoff>**。星环科技 建议自定义分区名，不要使用Inceptor的自动命名，命名时注意分区名中不能有“-”字符。Inceptor暂 不支持 重命名范围分区。

- ③ 最后出现的分区可以使用 **MAXVALUE** 作为上限, **MAXVALUE** 代表该分区键的数据类型所允许的最大值。

例 64. 创建范围分区表

建一张名为user_reg_date的范围分区表, 分区键为reg_date:

```
CREATE TABLE user_reg_date (name STRING)
PARTITIONED BY RANGE (reg_date DATE) (
    PARTITION before2010 VALUES LESS THAN ('2010-12-31'),
    PARTITION before2012 VALUES LESS THAN ('2012-12-31'),
    PARTITION beforemax VALUES LESS THAN (MAXVALUE)
);
```

例 65. 创建多层范围分区表

建一张名为rp_demo的表, 有三个范围分区键:

```
CREATE TABLE rp_demo (value INT)
PARTITIONED BY RANGE (id1 INT, id2 INT, id3 INT)
(
    PARTITION p5_105_205 VALUES LESS THAN (5, 105, 205),
    PARTITION p5_105_215 VALUES LESS THAN (5, 105, 215),
    PARTITION p5_115_205 VALUES LESS THAN (5, 115, 205),
    PARTITION p5_115_215 VALUES LESS THAN (5, 115, 215),
    PARTITION p5_115_max VALUES LESS THAN (5, 115, MAXVALUE),
    PARTITION p10_105_205 VALUES LESS THAN (10, 105, 205),
    PARTITION p10_105_215 VALUES LESS THAN (10, 105, 215),
    PARTITION p10_115_205 VALUES LESS THAN (10, 115, 205),
    PARTITION p10_115_215 VALUES LESS THAN (10, 115, 215),
    PARTITION pall_max values less than (MAXVALUE, MAXVALUE, MAXVALUE)
);
```

语法: **CREATE TABLE LIKE**

```
CREATE [TEMPORARY] [EXTERNAL] TABLE <table_name> LIKE <partitioned_table>;
```

这样创建的 **<table_name>** 会和 **<partitioned_table>** 一样的列和分区键。

注意, Inceptor不支持范围分区和单值分区混合对表进行多层分区。

3.14.2.2. 导入数据

向范围分区表导入数据的方法是通过 **INSERT ... SELECT** 将查询结果插入范围分区表。和单值分区表不同的是, 插入时不需要指定分区字段的值, Inceptor会自动判断记录所属的分区, 形式上比较像动态分区插入。

语法

```
INSERT (OVERWRITE|INTO) TABLE <table_name> PARTITION (<dynamic_partition_key>) SELECT
<select_statement> FROM <source>;
```

例 66. 向范围分区表插入数据

```
INSERT OVERWRITE TABLE user_reg_date PARTITION (reg_date) SELECT name, reg_date FROM user_info;
```

3.14.2.3. ALTER 范围分区表

Inceptor支持使用 **ALTER** 语句添加、删除和重命名范围分区。

3.14.2.3.1. 添加范围分区

语法

```
ALTER TABLE <table_name> ADD [IF NOT EXISTS] PARTITION [<partition_name>] VALUES LESS THAN (<values>);
```



添加范围分区只能增加新的分区，而不能 **分裂** 原有的分区。所以，只能向表中末尾的分区之后添加分区，并且新添加的分区的范围和表中已有的分区的范围 **不能有交集**。比如，一张分区表的范围为：[MINVALUE, 10],[10,20],[20,30)。向这张表新添加范围为[a, b)的分区时，a必须大于30，否则会引起错误。因为这个限制，如果一张表的最后一个分区是以 **MAXVALUE** 结尾的，它将不能再增加范围分区。

例 67. 添加范围分区

[创建范围分区表](#)中建的user_reg_date表的最后一个分区是以 **MAXVALUE** 结尾的，所以不能向它添加新的范围分区。建一张新表user_reg_date_rp:

```
CREATE TABLE user_reg_date_rp(name STRING) PARTITIONED BY RANGE (reg_date DATE) (PARTITION before2010 VALUES LESS THAN ('2010-12-31'), PARTITION before2012 VALUES LESS THAN ('2012-12-31'), PARTITION before2014 VALUES LESS THAN ('2014-12-31'));
```

用 **SHOW PARTITIONS** 可以查看这张表中的分区：

```
SHOW PARTITIONS user_reg_date_rp;
+-----+
| partition |
+-----+
| before2010 |
| before2012 |
| before2014 |
+-----+
```

用 **ALTER TABLE** 向这张表添加范围分区：

```
ALTER TABLE user_reg_date_rp ADD PARTITION before2016 VALUES LESS THAN ('2016-12-31');
```

现在用***SHOW PARTITIONS*** 可以看见新增分区：

```
SHOW PARTITIONS user_reg_date_rp;
+-----+
| partition |
+-----+
| before2010 |
| before2012 |
| before2014 |
| before2016 |
+-----+
```

3.14.2.3.2. 删除范围分区

语法

```
ALTER TABLE <table_name> DROP PARTITION <partition_name>;
```

删除某个分区后，余下的分区范围会进行调整——被删除分区的下一个分区的范围会自动扩展，保证余下分区的范围首尾相连。例如，一张表原有分区[10, 20), [20, 30), [30, 40)，现在删除[20, 30)分区后，[30, 40)分区会自动扩展成[20, 40)分区。

例 68. 删除范围分区

我们使用[添加范围分区](#)例子中的user_reg_date_rp表。当前表中的分区为：

```
SHOW PARTITIONS user_reg_date_rp;
+-----+
| partition |
+-----+
| before2010 |
| before2014 |
| before2016 |
+-----+
```

现在删除 before2012 分区：

```
ALTER TABLE user_reg_date_rp DROP PARTITION before2012;
```

再次查看表中分区：

```
SHOW PARTITIONS user_reg_date_rp;
+-----+
| partition |
+-----+
| before2010 |
| before2014 |
| before2016 |
+-----+
```

3.15. 分桶表

对表分桶可以将表中记录按分桶键的哈希值分散进多个文件中，这些小文件称为“桶”。

3.15.1. 建表

分桶表的建表有三种方式：直接建表，`CREATE TABLE LIKE` 和 `CREATE TABLE AS SELECT`。Holodesk 表分桶使用不同的语法，请参考“[Holodesk表](#)”。

语法：直接建分桶表

```

CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS]
[<database_name>.]<table_name>
[(<column_name> <data_type> [, <column_name> <data_type> ...])]
[PARTITIONED BY ...] ①
CLUSTERED BY (<col_name>) ②
[SORTED BY (<col_name> [ASC|DESC] [, <col_name> [ASC|DESC]...])] ③
INTO <num_buckets> BUCKETS ④
[ROW FORMAT <row_format>]
[STORED AS <file_format>] ⑤
[LOCATION '<hdfs_path>']
[TBLPROPERTIES ('<property_name>'='<property_value>', ...)]

```

- ① 分区子句，具体细节请参考“分区表”。表可以同时分区和分桶，当表分区时，每个分区下都会有 `<num_buckets>` 个桶。
- ② `<col_name>` 为分桶键。分桶键只能有一个。和分区键不同的是，分桶键必须是表中已有的列，所以在指定分桶键时，**不需要** 指定分桶键的数据类型。
- ③ 可以选择使用 `SORTED BY ...` 在桶内排序，排序键和分桶键 **无需** 相同。`ASC` 为升序选项，`DESC` 为降序选项，默认排序方式是升序。
- ④ `<num_buckets>` 指定分桶个数，也就是表目录下小文件的个数。
- ⑤ 指定存储格式，可以分桶的格式有 `TEXTFILE`, `CSVFILE` 和 `ORC`。

! 在建表时使用 `CLUSTERED BY` 和 `SORTED BY` 只决定了 Inceptor 读取 这张表的方式，但不决定 Inceptor 将数据写入表的方式。也就是说，建表时的 `CLUSTERED BY` 和 `SORTED BY` 仅仅是利用表的元数据给 Inceptor 提示，让 Inceptor 可以根据提示的数据分布选择最优的数据读取逻辑。但是表中数据的 **实际** 分布是在数据写入时决定的。如果写入数据的方式不当，那么表中数据将呈现出和表的元数据不同的分布（例如没有正确分桶或没有正确排序）。所以执行导入数据的用户需要确保数据正确地被写入表中，具体操作请参考[向分桶表内导数据](#)。

例 69. 建分桶表

下面语句建一个分桶表 `user_cid_regdate`，分桶键为 `cid`，分三成个桶，桶内靠 `regdate` 排序。

```
CREATE TABLE user_cid_regdate (name STRING, cid STRING, regdate DATE) CLUSTERED BY (cid) SORTED BY (regdate) INTO 3 BUCKETS;
```

3.15.2. 向分桶表内导数据

向分桶表内导数据的方法为使用 `INSERT…SELECT` 向表中插入查询结果。需要注意的是，Inceptor 向分桶表内写数据时，并不会自动将数据分桶或排序，所以，在导数据的用户要手动分桶和排序，确保目标表中的数据和它定义的分布一致。手动分桶的方法有两种，效果相同。

方法一： 打开 `enforce bucketing` 开关。

方法二： 将 `reducer` 个数设置为目标表的桶数，并在 `SELECT` 语句中用 `DISTRIBUTE BY <bucket_key>` 对查询结果按目标表的分桶键分进 `reducer` 中。

如果目标表在建表时规定了桶内有序（也就是用了 `SORTED BY` 关键字），那么导入数据时要在 `SELECT` 语句中加上 `SORT BY` 来保证插入的数据是桶内有序的。

语法1：向分桶表导数据，用方法一手动分桶

```
SET hive.enforce.bucketing=true; ①
INSERT (INTO|OVERWRITE) TABLE <bucketed_table> SELECT <select_statement>
[SORT BY <sort_key> [ASC|DESC], [<sort_key> [ASC|DESC], ...]]; ②
```

- ① 打开enforce bucketing开关，强制Inceptor分桶。这个开关默认情况下是打开的，但是可能在Inceptor使用过程中被关掉，所以向分桶表导数据前，请确保开关是打开的。
- ② 如果目标表是桶内有序的，那么要对查询结果要按对应目标表排序键的列排序，且升降序 [ASC|DESC] 也和目标表定义一致。

语法2：向分桶表导数据，用方法二手动分桶

```
SET mapred.reduce.tasks = <num_buckets>; ①
INSERT (INTO|OVERWRITE) TABLE <bucketed_table>
SELECT <select_statement>
DISTRIBUTE BY <bucket_key>, [<bucket_key>, ...] ②
[SORT BY <sort_key> [ASC|DESC], [<sort_key> [ASC|DESC], ...]]; ③
```

- ① 将reducer个数设置为目标表的桶数（<num_buckets>）。向分桶表插入数据时，SELECT 的每个reducer产生的结果会被插入一个桶中，所以执行插入的用户需要保证reducer个数和目标表中分桶数一致。
- ② 将查询结果按对应目标表分桶键的列分进各个reducer，做到查询结果的数据分布和表定义的分桶一致。
- ③ 如果目标表是桶内有序的，需要将查询结果按对应目标表排序键的列在每个reducer中排序，且升降序 [ASC|DESC] 也和目标表定义一致。当排序键和分桶键相同且排序规则为升序，Inceptor提供下面的快捷语法：

语法2'

```
SET mapred.reduce.tasks = <num_buckets>;
INSERT (INTO|OVERWRITE) TABLE <bucketed_table>
SELECT <select_statement>
CLUSTER BY <bucket_sort_key>, [<bucket_sort_key>, ...]; ①
```

- ① 和 DISTRIBUTE BY <bucket_sort_key>, [<bucket_sort_key>, …] SORT BY <bucket_sort_key> ASC, [<bucket_sort_key> ASC, …] 效果相同。

例 70. 向分桶表导数据

假设我们现在要向[建分桶表](#)中建的user_cid_regdate表导入数据。这张表按cid分了三个桶。所以正确地导入数据后表目录下应该有三个小文件。

我们先演示 不当 的导数据方法——在导数据时 不手动分桶：

```
SET hive.enforce.bucketing=false;
INSERT OVERWRITE TABLE user_cid_regdate SELECT name, citizen_id, reg_date FROM user_info;
```

查看HDFS上的表目录：

```
user_cid_regdate/000000_0
user_cid_regdate/000001_0
```

我们看到表目录下只有两个小文件，而不是三个。这是因为Inceptor采用了默认的方法写数据，导致表中数据分布和表的元数据不一致。您的Inceptor服务和我们当前演示使用的Inceptor服务可能设置不同，在您自己执行这些操作时可能会看到不一样的小文件数量。

要保证表中数据的确分三个桶，我们需要执行下面两个操作之一（两个操作作用相同）：

使用[方法一](#)手动分桶：

```
SET hive.enforce.bucketing=true;
INSERT OVERWRITE TABLE user_cid_regdate SELECT name, citizen_id, reg_date FROM user_info;
```

使用[方法二](#)手动分桶：

```
SET mapred.reduce.tasks = 3;
INSERT OVERWRITE TABLE user_cid_regdate SELECT name, citizen_id, reg_date FROM user_info
DISTRIBUTE BY citizen_id;
```

现在查看HDFS上的表目录：

```
user_cid_regdate/000000_0
user_cid_regdate/000001_0
user_cid_regdate/000002_0
```

我们看到表目录下的小文件数量和桶数一致了。

例 71. 向分桶且桶内有序表导数据

在[向分桶表导数据](#)中，我们确保了导入数据正确地分桶，但是没有在导入数据时手动排序，所以打开表目录下的小文件之一000001_0可以看到桶内数据没有排序，和user_cid_regdate的元数据不一致（user_cid_regdate建表时指定桶内按regdate排序，见[建分桶表](#)）：

```
马**^A14*****7^A2011-01-01
华* ^A42*****1^A2008-02-14
魏**^A52*****4^A2009-12-02
宁**^A42*****7^A2008-10-31
管**^A33*****4^A2014-10-03
```

要保证桶内数据有序，需要执行下面两个操作之一（两个操作作用相同）：

```
SET hive.enforce.bucketing=true;
INSERT OVERWRITE TABLE user_cid_regdate SELECT name, citizen_id, reg_date FROM user_info SORT BY reg_date;
```

或者

```
SET mapred.reduce.tasks = 3;
INSERT OVERWRITE TABLE user_cid_regdate SELECT name, citizen_id, reg_date FROM user_info
DISTRIBUTE BY citizen_id SORT BY reg_date;
```

现在查看表目录下的小文件，可以发现所有小文件中数据按regdate按升序排列：

小文件000000_0：

```
李**^A31*****9^A2011-09-16
李* ^A46*****3^A2013-07-02
```

小文件000001_0：

```
华* ^A42*****1^A2008-02-14
宁**^A42*****7^A2008-10-31
魏**^A52*****4^A2009-12-02
马**^A14*****7^A2011-01-01
管**^A33*****4^A2014-10-03
```

小文件000002_0：

```
祝**^A23*****8^A2010-01-01
潘**^A51*****5^A2011-04-30
邱* ^A34*****8^A2012-10-24
```

例 72. 分区分桶表

下面语句建一个分区分桶表user_cid_accllevel，分桶键为cid（分三个桶），分区键为acc_level。

```
CREATE TABLE user_cid_accllevel (name STRING, cid STRING) PARTITIONED BY (acc_level STRING)
CLUSTERED BY (cid) INTO 3 BUCKETS;
```

向分区分桶表中导数据时需要保证分桶、排序和分区都要正确：

```
SET hive.enforce.bucketing=true;
INSERT OVERWRITE TABLE user_cid_accllevel PARTITION (acc_level = 'A') SELECT name, citizen_id
FROM user_info WHERE acc_level = 'A';
```

4. Inceptor PL/SQL手册（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](#)。

4.1. Inceptor PL/SQL一览

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

- [預定义函数/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

4.2. Inceptor PL/SQL手册中的表

在Inceptor_plsql手册中，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: 列从左到右依次为trans_id, acc_num, trans_time, trans_type, stock_id, price, amount;其中trans_id|acc_num|trans_time|trans_type|stock_id|均为string类型, |price|amount均为double类型。

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
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:列从左到右依次为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

- ORC非分区表zara:列从左到右依次为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

- ORC非分区表za:列从左到右依次为name, age

john	20
fd	24
zz	10
alice	25

- ORC非分区表ra:列从左到右依次为name, gpa

smith	3.4
lisi	3.3
ll	3.3
lily	3.6

- ORC非分区表orctest2:列从左到右依次为age, gpa

22	4.3
20	4.0
21	3.9

- ORC分区表zza:列从左到右依次为name, age, level(分区键为level)

zhaoliu	23	A
john	30	A
john	20	B
wangwu	22	C
lily	30	C
alice	25	C
john	20	C
sara	21	C

4.3. 基础知识

4.3.1. 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，常量的值不能改变。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。

4.3.1.1. 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 73. 变量的声明

```

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

4.3.1.2. 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 74. 常量的声明

```

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

4.3.1.3. 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 75. 用关键字“DEFAULT”给变量赋默认值

```

DECLARE
    V1 STRING DEFAULT 'HELLO'; ①
    V2 INT DEFAULT 100;
    V3 BOOLEAN DEFAULT FALSE;
BEGIN
    V1 := 'hello world'; ②
    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

4.3.2. 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。

```

CREATE OR REPLACE PACKAGE test
IS
    PROCEDURE proc2( test_name OUT DOUBLE);
END;
/

```

4.3.2.1. 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 76. 同时声明两个变量，变量名相同，数据类型不相同

```
DECLARE
    V1 BOOLEAN;
    V1 INT;
BEGIN
    V1:= FALSE;
END;
/
```

输出结果为：

```
Error: Error while processing statement: FAILED: Error in semantic analysis:
org.apache.hadoop.hive.ql.parse.SemanticException:
ANONYMOUS BLOCK (LINE 3, COLUMN 3, TEXT "V1 INT"): Existing variable v1 (state=,code=10)
```

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。

4.3.2.2. 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 77. 同时声明两个变量，变量名字母的大小写形式不同

```
DECLARE
    zip_code INT;
    ZIP_CODE INT;
BEGIN
    zip_code:= 90120;
END;
/
```

输出结果为：

```
Error: Error while processing statement: FAILED: Error in semantic analysis:
org.apache.hadoop.hive.ql.parse.SemanticException:
ANONYMOUS BLOCK (LINE 3, COLUMN 3, TEXT "ZIP_CODE INT"): Existing variable zip_code
(state=,code=10)
```

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。

4.3.2.3. 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 78. 删除zara表中某个存在的gpa

```

DECLARE
gpa DOUBLE default 4.5; ①
BEGIN
    DELETE FROM zara WHERE gpa=gpa; ②
    DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows. '); ③
END;
/

```

- ① 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。
- ② 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 $4.5=4.5$ 。
- ③ 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 3 rows. |
+-----+
```

例 79. 删除zara表中某个不存在的gpa

```

DECLARE
gpa DOUBLE default 2.0; ①
BEGIN
    DELETE FROM zara WHERE gpa=gpa; ②
    DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows. '); ③
END;
/

```

- ① 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。
- ② 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 $2.0=2.0$ 。
- ③ 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 3 rows. |
+-----+
```

4.3.2.4. 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 80. 嵌套命名

```

DECLARE
    V1 STRING;  ①
    V2 INT;
BEGIN
    V1 := 'HELLO';  ②
    V2 := 123;
    DBMS_OUTPUT.PUT_LINE(V1);
    DBMS_OUTPUT.PUT_LINE(V2);

DECLARE
    V1 BOOLEAN;  ③
    V3 DOUBLE;
BEGIN
    V1 := FALSE;  ④
    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;
/

```

- ① 声明一个字符串类型的变量，名为V1。
- ② 给变量V1赋值。
- ③ 声明一个布尔值类型的变量，名为V2。
- ④ 给变量V2赋值。

输出结果为：

-----	-----
	output
+	-----
	HELLO
	123
	FALSE
	12.13
	World
+	-----

4.3.3. 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 81. 声明三个不同类型的变量

```

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.

4.3.3.1. 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 82. 将逻辑值赋值给变量

```

DECLARE
    done BOOLEAN;
    counter INT:= 0;
BEGIN
    done := FALSE;
    WHILE done != TRUE
    LOOP
        counter := counter + 1;
        done := (counter > 500);
    END LOOP;
    PUT_LINE('the counter is:' || counter);
END;
/

```

输出结果为：

output
the counter is:501

4.3.3.2. 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 83. 将查询结果赋值给变量

```

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

4.4. 数据类型

4.4.1. 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/DDOUBLE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》。

4.4.1.1. STRINGS

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；对于字符串变量，要在PL/SQL语句块中的DECLARE部分声明。

例 84. 对于字符串常量，可以直接引用

```

BEGIN
    DBMS_OUTPUT.put_line ('Hello World');
END;
/

```

输出结果为：

output
Hello World

例 85. 对于字符串变量，要事先声明

```

DECLARE
    l_message STRING; ①
BEGIN
    l_message:= 'Hello World'; ②
    DBMS_OUTPUT.put_line (l_message); ③
END;
/

```

① 定义一个STRING类型的变量为l_message。

② 给变量l_message赋值。

③ 输出变量l_message的值。

输出结果为：

```
+-----+
|   output   |
+-----+
| Hello World |
+-----+
```

4.4.1.2. 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 86. 直接使用整数

```

BEGIN
    DBMS_OUTPUT.put_line (100);
END;
/

```

输出结果为：

```
+-----+
|   output   |
+-----+
| 100        |
+-----+
```

例 87. 声明一个INT类型的变量

```

DECLARE
    l_int  int; ①
BEGIN
    l_int:= 1000; ②
    DBMS_OUTPUT.put_line (l_int); ③
END;
/

```

① 定义一个INT类型的变量为l_int

② 给变量l_int赋值

③ 输出变量l_int的值

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。

4.4.2. %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 88. 使用%TYPE属性声明变量，其数据类型为另一个变量的类型

```

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 89. 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

```

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

4.4.3. %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 90. 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

```

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

4.4.4. 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。

4.5. 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。

4.5.1. 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



1. 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。
2. 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。

4.5.2. 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 91. 创建最简单的匿名块

```
BEGIN
    DBMS_OUTPUT.put_line ('Hello World!') ①
END;
/
```

① 输出信息Hello World

输出结果:

```
+-----+
|   output   |
+-----+
| Hello World! |
+-----+
```

4.5.3. 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 92. 创建稍复杂的匿名块

```
DECLARE
    l_message STRING := 'Hello World!' ①
BEGIN
    DBMS_OUTPUT.put_line (l_message) ②
END;
/
```

① 定义一个string类型的变量l_message，并赋值。

② 输出l_message的值。

输出结果:

```
+-----+
|   output   |
+-----+
| Hello World! |
+-----+
```

4.5.4. 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 93. 创建完整的匿名块

```

DECLARE
    transid STRING; ①
BEGIN
    SELECT trans_id
    into transid
    from transactions t1 ②
    PUT_LINE(transid) ③
EXCEPTION
    WHEN too_many_rows ④
    THEN
        DBMS_OUTPUT.put_line ('too many rows') ⑤
END;
/

```

① 定义一个string类型的变量transid。

② 查询transactions表中账号为6513065的交易号，并放进变量transid中，此处transactions后需加上表的别名。

③ 输出变量transid的值。

④ 引用一个情况名为too_many_rows的异常。

⑤ 当异常发生时的所输出的信息。

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](#)。

4.5.5. 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 94. 匿名块里的嵌套

```

DECLARE
    test1 STRING; ①
BEGIN
    DECLARE
        test2 STRING; ②
    BEGIN
        SELECT trans_time
        INTO test2
        FROM transactions
        WHERE acc_num=6513064; ③
        dbms_output.put_line(test2); ④
    EXCEPTION
        WHEN NO_DATA_FOUND THEN ⑤
            dbms_output.put_line('no values');
    END
    SELECT trans_id
    INTO test1
    FROM transactions
    WHERE price=12.13; ⑥
    dbms_output.put_line(test1); ⑦
EXCEPTION
    WHEN TOO_MANY_ROWS THEN ⑧
        dbms_output.put_line('too many rows');
END;
/

```

- ① 声明外部语句块中的变量test1。
- ② 声明内部语句块中的变量test2。
- ③ 查询transactions表中账号为6513064的交易号，并放进变量test2中。
- ④ 输出变量test2的值。
- ⑤ 引用一个NO_DATA_FOUND的异常情况名。
- ⑥ 查询transactions表中价格为12.13的交易号，并放进变量test1中。
- ⑦ 输出变量test1的值。
- ⑧ 引用一个TOO_MANY_ROWS的异常情况名。

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。
 - CLI+InceptorServer 1中通过以下命令：set plsql.use.slash=true;
 - 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](#)。



4.6. 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

4.6.1. IF

4.6.1.1. IF-THEN

- 语法：

```
IF condition THEN
  {...statements...}
END IF;
```

例 95. 查询transactions表中交易号为648230055的价格

```
DECLARE
  v_price DOUBLE;
  v_comment string; ①
BEGIN
  SELECT price
    INTO v_price
   FROM transactions
  WHERE trans_id=648230055; ②
  dbms_output.put_line('the price is:'||v_price); ③
  IF v_price > 20 THEN ④
    v_comment := 'the price is too high' ⑤
  END IF;
  dbms_output.put_line(v_comment); ⑥
END;
/
```

- ① 分别声明两个DOUBLE，STRING类型的变量v_price，v_comment。
- ② 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。
- ③ 输出v_price的值。
- ④ 确定IF的条件语句为v_price的值大于20。
- ⑤ 在满足IF的条件语句的情况下，给变量v_comment赋值。
- ⑥ 输出变量v_comment的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| the price is:22.66 |
| the price is too high |
+-----+
```

4.6.1.2. IF-THEN-ELSE

- 语法：

```
IF condition THEN
  {...statements...}
ELSE
  {...statements...}
END IF;
```

例 96. 查询transactions表中交易号为943197522的价格

```
DECLARE
  v_price DOUBLE;
  v_comment string; ①
BEGIN
  SELECT price
  INTO v_price
  FROM transactions
  WHERE trans_id=943197522; ②
  dbms_output.put_line('the price is:'||v_price); ③
  IF v_price > 20 THEN ④
    v_comment := 'the price is too high'; ⑤
  ELSE
    v_comment := 'the price is not too high'; ⑥
  END IF;
  dbms_output.put_line(v_comment); ⑦
END;
/
```

- ① 分别声明两个DOUBLE, STRING类型的变量v_price, v_comment。
- ② 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。
- ③ 输出v_price的值。
- ④ 确定IF的条件语句为v_price的值大于20。
- ⑤ 在满足IF的条件语句的情况下，给变量v_comment赋值。
- ⑥ 在不满足IF的条件语句的情况下，给变量v_comment赋值。
- ⑦ 输出变量v_comment的值。

输出结果为：

```
+-----+
|       output
+-----+
| the price is:12.13
| the price is not too high
+-----+
```

4.6.1.3. IF-THEN-ELSIF

- 语法：

```
IF condition THEN
  {...statements...}
ELSIF condition THEN
  {...statements...}
ELSE
  {...statements...}
END IF;
```

例 97. 查询transactions表中交易号为943197522的价格

```

DECLARE
    v_price DOUBLE;
    v_comment string; ①
BEGIN
    SELECT price
        INTO v_price
        FROM transactions
        WHERE trans_id=943197522; ②
    dbms_output.put_line('the price is:'||v_price); ③
    IF v_price > 15 THEN ④
        v_comment := 'the price 大于 15'; ⑤
    ELSIF v_price > 10 then ⑥
        v_comment := 'the price 大于 10'; ⑦
    ELSIF v_price > 5 then ⑧
        v_comment := 'the price 大于 5'; ⑨
    ELSE
        v_comment := 'the price is too low'; ⑩
    END IF;
    dbms_output.put_line(v_comment);
END;
/

```

- ① 声明两个类型分别为DOUBLE、STRING的变量v_price、v_comment。
 - ② 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。
 - ③ 输出v_price的值。
 - ④ 确定IF的条件语句为v_price的值大于15。
 - ⑤ 在满足IF的条件语句的情况下，给变量v_comment赋值。
 - ⑥ 确定第一个ELSIF的条件语句为v_price的值大于10。
 - ⑦ 在满足第一个ELSIF的条件语句的情况下，给变量v_comment赋值。
 - ⑧ 确定第二个ELSIF的条件语句为v_price的值大于5。
 - ⑨ 在满足第二个ELSIF的条件语句的情况下，给变量v_comment赋值。
 - ⑩ 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。
- 输出变量v_comment的值。

输出结果为：

output
the price is:12.13
the price 大于 10

例 98. 查询transactions表中交易号为209441379的价格

```

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：

4.6.2. LOOP

4.6.2.1. the simple loop

- 语法：

```

LOOP
    {...statements...}
    EXIT WHEN conditions --条件满足，退出循环语句
END LOOP;

```

例 99. the simple loop

```

DECLARE
    test int := 0; ①
BEGIN
    LOOP
        test := test + 1; ②
        dbms_output.put_line('test当前值为:' || test); ③
        EXIT WHEN test=10; ④
    END LOOP;
END;
/

```

- ① 定义一个整数类型的变量test，并附初值为0。
- ② 在0的基础上，给test的值不断加1。
- ③ 输出当前test的值。
- ④ 确定退出循环语句的条件语句为，当test的值为10时。

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](#)。

4.6.3. WHILE

- 语法：

```

WHILE conditions
LOOP
    {...statements...}
END LOOP;

```

例 100. the while

```

DECLARE
    test int := 0; ①
BEGIN
    WHILE test<10 ②
    LOOP
        dbms_output.put_line('test当前值:' || test); ③
        test := test + 1 ④
    END LOOP;
END;
/

```

- ① 定义一个整数类型的变量test，并附初值为0。
- ② 确定循环条件的while条件语句。
- ③ 输出当前test的值。
- ④ 在符合WHILE条件语句的情况下，当前test值的基础上，不断加1。

输出结果为：

```

+-----+
|   output   |
+-----+
|test当前值:0|
|test当前值:1|
|test当前值:2|
|test当前值:3|
|test当前值:4|
|test当前值:5|
|test当前值:6|
|test当前值:7|
|test当前值:8|
|test当前值:9|
+-----+

```

4.6.4. 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 101. FOR IN

```
BEGIN
  FOR test IN 1..5 LOOP ①
    dbms_output.put_line('test当前值为:' || test); ②
  END LOOP;
END;
/
```

- ① 对于1到5之间的整数。
 ② 从1开始，以每次自动加1的方式，依次输出test的值。

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 102. FOR IN REVERSE

```
BEGIN
  FOR test IN REVERSE 1..5 LOOP ①
    dbms_output.put_line('test当前值为:' || test); ②
  END LOOP;
END;
/
```

- ① 对于1到5之间的整数。
 ② 从5开始，以每次自动减1的方式，依次输出当前值。

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 103. 有循环退出的条件

```

BEGIN
  FOR test IN REVERSE 1..5 LOOP ①
    dbms_output.put_line('test当前值为:' || test); ②
    EXIT WHEN test=3; ③
  END LOOP;
END;
/

```

- ① 对于1到5之间的整数。
- ② 从5开始，以每次自动减1的方式，依次输出当前值。
- ③ 循环退出的条件为 当test的值为3时。

输出结果为：

```

+-----+
|   output   |
+-----+
|test当前值为:5 |
|test当前值为:4 |
|test当前值为:3 |
+-----+

```

例 104. IN后面的数字是从大到小的顺序

```

BEGIN
  FOR test IN 5..1 LOOP
    dbms_output.put_line('test当前值为:' || test);
  END LOOP;
END;
/

```

输出结果为：

```

BEGIN
  FOR test IN 5..1 LOOP
    dbms_output.put_line('test当前值为:' || test);
  END LOOP;
END;
/

```

可以看到，跟在IN 后面的数字如果是从大到小的顺序，Inceptor不会返回任何结果。

4.6.5. 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。

4.6.5.1. 例一 使用FORALL往表中插入数据

例 105. 利用forall语句，往表za中批量插入数据

```

DECLARE
    TYPE NumList IS VARRAY(20) OF int; ①
    depts NumList := NumList(20,22,23); ②
BEGIN
    FORALL i IN depts.FIRST()..depts.LAST() ③
        INSERT INTO za VALUES('lily',depts(i)); ④
    IF SQL%FOUND THEN ⑤
        dbms_output.put_line('successfully insert new data'); ⑥
    ELSE
        dbms_output.put_line('nothing'); ⑦
    END IF;
END;
/

```

- ① 声明一个名为NumList的VARRAY类型，可容纳20个整数。
- ② 定义一个NumList类型的变量depts，并赋初值。
- ③ 使用forall语句，对于变量depts中的全部的值所对应的下标。
- ④ 往表za中插入数据，姓名为lily，年龄分别为变量depts的值。
- ⑤ 条件判断语句为如果sql成功执行了。
- ⑥ sql语句成功执行，所输出的信息。
- ⑦ sql语句没有成功执行，所输出的信息。

输出结果为：

```

+-----+
|       output      |
+-----+
| successfully insert new data |
+-----+

```

4.6.5.2. 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 106. 利用forall语句，查询表transactions中的交易时间

```

DECLARE
    TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF double; ①
    V_ORG_VARRAY ORG_VARRAY_TYPE; ②
BEGIN
    V_ORG_VARRAY := ORG_VARRAY_TYPE('12.13','11.11','6.12','4.5','22.66'); ③
    FORALL I IN V_ORG_VARRAY.FIRST()..V_ORG_VARRAY.LAST() ④
        SELECT trans_time
        FROM transactions
        WHERE price=V_ORG_VARRAY(i); ⑤
END;
/

```

- ① 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。
- ② 定义一个ORG_VARRAY_TYPE类型的变量V_ORG_VARRAY。
- ③ 给变量V_ORG_VARRAY赋值。
- ④ 使用forall语句，对于变量V_ORG_VARRAY中的全部的值。
- ⑤ 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。

输出结果为：

output
20140105100520
20140205140521
20140506133109
201404301111523
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](#)。

4.6.6. 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 107. EXIT WHEN

```
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

4.6.7. CONTINUE(WHEN)

CONTINUE为PL/SQL中退出当前循环语句的条件语句，通常与LOOP语句一起使用。

例 108. CONTINUE(WHEN)

```

CREATE OR REPLACE PROCEDURE continue_proc() ①
IS
DECLARE
    var1 INT:=1;
    var2 INT; ②
BEGIN
<<outer_loop>>
LOOP
    var2 := 1; ③
    dbms_output.put_line("Outer loop body " || var1); ④
    var1 := var1 + 1; ⑤
<<inner_loop>>
LOOP
    CONTINUE outer_loop when var1 = 2; ⑥
    dbms_output.put_line("Inner loop body " || var2); ⑦
    var2 := var2 + 1; ⑧
    EXIT inner_loop when var2 = 3; ⑨
END LOOP;
    EXIT WHEN var1 = 3; ⑩
END LOOP outer_loop;
END;
/
BEGIN
    continue_proc();
END;
/

```

- ① 创建一个不带参数的过程continue_proc()。
- ② 分别定义两个int类型的变量，并给变量var1赋初值为1。
- ③ 在outer_loop中，给变量var2赋初值1。
- ④ 在outer_loop中，输出变量var1当前的值，也就是初值为1。
- ⑤ 在outer_loop中，给变量var1在当前值的基础上加上1。
- ⑥ 在inner_loop中，定义continue，如果var1的值为2，就继续outer_loop。
- ⑦ 在inner_loop中，输出此时var2的值，也就是var2的初值为1。
- ⑧ 在inner_loop中，给变量var2在当前值的基础上加上1。
- ⑨ 当var2的值为3时，就退出inner_loop。
- ⑩ 当var1的值为3时，就退出outer_loop。

输出结果为：

```

+-----+
|      output      |
+-----+
| Outer loop body 1 |
| Outer loop body 2 |
| Inner loop body 1 |
| Inner loop body 2 |
+-----+

```

4.6.8. 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 109. 打印变量的值

```
DECLARE
    V_int INT:= 1; ①
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE('V_int当前值为:' || V_int); ②
        V_int := V_int + 1; ③
        IF V_int > 5 THEN
            GOTO label0ffLOOP ④
        END IF;
    END LOOP;
    <<label0ffLOOP>>
    DBMS_OUTPUT.PUT_LINE('V_int当前值为:' || V_int);
END;
/
```

- ① 定义一个int类型的变量V_int，赋初值为1。
- ② 打印出变量V_int的当前值。
- ③ 在变量V_int初值的基础上不断加1。
- ④ 当变量V_int大于5时，跳转到标号label0ffLOOP。

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 110. 求和

```

DECLARE
    v_i INT := 0;
    v_s INT:= 0; ①
BEGIN
    <<label_1>>
    v_i := v_i + 1; ②
    IF v_i <= 1000 THEN
        v_s := v_s + v_i ; ③
        GOTO label_1 ④
    END IF;
    DBMS_OUTPUT.PUT_LINE(v_s); ⑤
END;
/

```

- ① 分别定义两个int类型的变量，并赋初值为0。
- ② 变量v_i从零开始，不断加1。
- ③ 当v_i小于等于1000时，将两个变量的和赋值给变量v_s。
- ④ 当v_i 大于1000时，跳转到标号label_1。
- ⑤ 输出变量v_s的值。

输出结果为：

-----+
output
-----+
500500
-----+

4.6.8.1. 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。

4.6.9. CASE

例 111. 查询表transactions中账号为“6513065”的价格，并依次输出价格的状态

```

DECLARE
    message string; ①
    p_price transactions.price%type; ②
    CURSOR cur IS SELECT price FROM transactions where acc_num='6513065'; ③
BEGIN
    OPEN cur ;
    LOOP
        FETCH cur INTO p_price; ④
        EXIT WHEN cur%NOTFOUND; ⑤
        DBMS_OUTPUT.PUT_LINE(p_price); ⑥
        message:= case ⑦
            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;
        DBMS_OUTPUT.PUT_LINE(message); ⑧
    END LOOP;
    CLOSE cur; ⑨
END;
/

```

- ① 声明一个字符串类型的变量message。
- ② 声明一个变量p_price，数据类型为表transactions中价格字段的类型。
- ③ 声明游标cur用来查询表transactions中账号为6513065的价格。
- ④ 将游标查询到的结果放进变量p_price里。
- ⑤ 如果没有查询到结果就退出。
- ⑥ 依次输出所查询到的价格信息。
- ⑦ 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’。
- ⑧ 依次输出message的信息。

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

4.7. 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，具体细节与方法请参考[通过外部工具查看存储过程](#)。

4.7.1. 不带参数的过程

例 112. 创建一个名为hello_world的过程

```
CREATE OR REPLACE PROCEDURE
hello_world() ①
IS
    l_message STRING := 'Hello World!' ②
BEGIN
    DBMS_OUTPUT.put_line (l_message) ③
END;
/
BEGIN
    hello_world();
END;
/
```

① 创建一个不带参数的过程hello_world。

② 定义一个string类型的变量l_message，并赋值。

③ 输出变量l_message的值。

输出结果：

output
Hello World!

例 113. 以同样的方法创建hello_universe过程

```
CREATE OR REPLACE PROCEDURE
hello_universe()
IS
    l_message STRING := 'Hello Universe!'
BEGIN
    DBMS_OUTPUT.put_line (l_message)
END;
/
```

输出结果为：

output
Hello Universe!

4.7.2. 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。

4.7.2.1. 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 114. 创建IN模式的参数过程param

```
CREATE OR REPLACE PROCEDURE
param( ①
      p_in IN INT) ②
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('in param:'||p_in) ③
END;
/
BEGIN
    param(222) ④
END;
/
```

- ① 创建一个名为param的过程。
- ② 过程param的形参为p_in，参数类型为IN，参数的数据类型为INT。
- ③ 输出参数p_in的值。
- ④ 调用过程param，其中'222'为实参。

输出结果为：

```
+-----+
|   output   |
+-----+
| in param:222 |
+-----+
```

4.7.2.2. 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。

4.7.2.2.1. 示例一

例 115. 创建带有参数的过程hello_place

```

CREATE OR REPLACE PROCEDURE
hello_place (①
    place_in STRING) ②
IS
    l_message STRING ③
BEGIN
    l_message := 'Hello ' || place_in ④
    DBMS_OUTPUT.put_line (l_message) ⑤
END;
/
BEGIN
    hello_place ('World')
    hello_place ('Universe')
END;
/

```

- ① 创建一个带有参数的过程hello_place。
- ② 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。
- ③ 定义一个string类型的变量。
- ④ 给变量l_message赋值。
- ⑤ 输出变量l_message的值。

hello_place过程输出结果为：

output
Hello World
Hello Universe

4.7.2.2. 示例二

例 116. 创建带有参数的过程ptransaction

```

CREATE OR REPLACE PROCEDURE
ptransaction( ①
    p_trans_id transactions.trans_id%TYPE) ②
IS
    v_price transactions.price%TYPE ③
BEGIN
    SELECT price
    INTO v_price
    FROM transactions
    WHERE trans_id=p_trans_id ④
    DBMS_OUTPUT.PUT_LINE('the price is:'||v_price) ⑤
END;
/
BEGIN
    ptransaction('943197522') ⑥
END;
/

```

- ① 创建一个名为ptransaction的过程。
- ② 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。
- ③ 定义一个变量v_price，数据类型为表transactions中price字段的类型。
- ④ 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。
- ⑤ 输出变量v_price的值。
- ⑥ 调用过程ptransaction，'943197522'为该过程的实参。

ptransaction过程输出结果为：

-----	-----
	output
-----	-----
	the price is:12.13
-----	-----

4.7.2.3. 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。

4.7.2.3.1. 示例一

例 117. 创建一个名为param1的过程

```

CREATE OR REPLACE PROCEDURE
param1( ①
    p_out  OUT string ) ②
IS
BEGIN
    p_out:='20140105100520' ③

END;
/
DECLARE
    v_out STRING ④
BEGIN
    param1(v_out) ⑤
    DBMS_OUTPUT.PUT_LINE('out param:' ||v_out) ⑥
END;
/

```

- ① 创建一个名为param1的过程。
- ② 参数为p_out，参数类型为OUT，参数的数据类型为STRING。
- ③ 给参数p_out赋值。
- ④ 声明一个数据类型为STRING的变量v_out。
- ⑤ 调用过程param1，其中v_out用来存放过程param1所输出的值。
- ⑥ 打印出变量v_out所获得的值。

输出结果为：

```

+-----+
|          output      |
+-----+
| out param:20140105100520 |
+-----+

```

4.7.2.3.2. 示例二

例 118. 创建一个名为param2的过程

```

CREATE OR REPLACE PROCEDURE
param2( ①
    p_in IN DOUBLE,
    p_out OUT STRING ) ②
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('in param:'||p_in) ④
    select acc_num
    into p_out
    from transactions
    where price=p_in ⑤
END;
/
DECLARE
    v_out STRING ⑥
BEGIN
    param2(12.13,v_out) ⑦
    DBMS_OUTPUT.PUT_LINE('out param:'||v_out) ⑧
END;
/

```

- ① 创建一个名为param2的过程。
- ② 第一个参数为p_in，类型为IN，数据类型为DOUBLE。
- ③ 第二个参数为p_out，类型为OUT，数据类型为STRING。
- ④ 输出第一个参数p_in的值。
- ⑤ 将表transactions中price的值和p_in相等所对应的账号放进p_out里。
- ⑥ 声明一个字符串类型的变量v_out。
- ⑦ 调用过程param2，其中v_out用来存放param所输出的数据。
- ⑧ 打印出变量v_out所获得的值。

输出结果为：

```

+-----+
|      output      |
+-----+
| in param:12.13 |
| out param:6513065 |
+-----+

```

4.7.2.4. 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。

4.7.2.4.1. 例一 IN与INOUT的对比

例 119. 创建一个名为param的过程

```

CREATE OR REPLACE PROCEDURE
param( ①
    p_in IN DOUBLE,      ②
    p_inout  INOUT STRING ) ③
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('in param:'||p_in) ④
    DBMS_OUTPUT.PUT_LINE('inout param:'||p_inout) ⑤
END;
/
DECLARE
    v_inout string:='6513065' ⑥
BEGIN
    param(11.12,v_inout) ⑦
END;
/

```

- ① 创建一个名为param的过程。
- ② 第一个参数为p_in，类型为IN，数据类型为DOUBLE。
- ③ 第二个参数为p_inout，类型为INOUT，数据类型为STRING。
- ④ 输出第一个参数p_in的值。
- ⑤ 输出p_inout的值。
- ⑥ 声明一个变量v_inout，数据类型为字符串，并附初值。
- ⑦ 调用过程param，其中v_inout用来存放param所输出的值。

输出结果为：

```

+-----+
|       output      |
+-----+
| in param:11.12   |
| inout param:6513065 |
+-----+

```

4.7.2.4.2. 例二 OUT与INOUT的对比

例 120. 创建一个名为param的过程

```

CREATE OR REPLACE PROCEDURE
param( ①
    p_out OUT double,   ②
    p_inout INOUT string ) ③
IS
BEGIN
    p_out:='11.11' ④
    p_inout:='20140105100520' ⑤
END;
/
DECLARE
    v_out double v_inout STRING ⑥
BEGIN
    param(v_out,v_inout) ⑦
    DBMS_OUTPUT.PUT_LINE('out param:'||v_out) ⑧
    DBMS_OUTPUT.PUT_LINE('inout param:'||v_inout) ⑨
END;
/

```

- ① 创建一个名为param的过程。
- ② 第一个参数为p_out，参数类型为OUT。
- ③ 第二个参数为p_inout，参数类型为INOUT。
- ④ 给p_out赋值。
- ⑤ 给p_inout赋值。
- ⑥ 声明DOUBLE类型的变量v_out，STRING类型的变量v_inout。
- ⑦ 调用过程param，其中v_out, v_inout分别用来存放param所输出的数据。
- ⑧ 打印出变量v_out所获得的值。
- ⑨ 打印出变量v_inout所获得的值。

输出结果为：

```

+-----+
|       output      |
+-----+
| out param:11.11  |
| inout param:20140105100520 |
+-----+

```

4.7.2.4.3. 案例三

例 121. 创建一个名为param的过程

```

CREATE OR REPLACE PROCEDURE
param( ①
    p_in IN double,   ②
    p_inout  INOUT string ) ③
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('in param:'||p_in) ④
    select acc_num
        into p_inout
        from transactions
    where price=p_in ⑤
    DBMS_OUTPUT.PUT_LINE('inout param:'||p_inout) ⑥
END;
/
DECLARE
    v_inout STRING ⑦
BEGIN
    param(11.11,v_inout) ⑧
END;
/

```

- ① 创建过程param。
- ② 第一个参数为p_in，类型为IN，数据类型为DOUBLE。
- ③ 第二个参数为p_inout，类型为INOUT，数据类型为STRING。
- ④ 输出p_in的值。
- ⑤ 将表transactions中price的值和p_in相等所对应的账号放进p_inout里。
- ⑥ 输出p_inout的值。
- ⑦ 声明一个字符串类型的变量v_inout。
- ⑧ 调用过程param，其中v_inout用来存放param所输出的数据。

输出结果为：

```

+-----+
|      output      |
+-----+
| in param:11.11  |
| out param:3912384 |
+-----+

```

4.7.2.5. 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。

4.7.2.5.1. 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 122. 创建一个含有五个参数的过程param

```

CREATE OR REPLACE PROCEDURE
param( ①
    ptrans_id IN STRING, ②
    ptrans_time IN STRING, ③
    ptrans_type OUT STRING, ④
    pacc_num INOUT STRING, ⑤
    pprice OUT DOUBLE) ⑥
IS
BEGIN
    ptrans_type:='b' ⑦
    pprice:=12.13 ⑧
    DBMS_OUTPUT.PUT_LINE('the trans_id is:'||ptrans_id)
    SELECT acc_num INTO pacc_num FROM transactions WHERE trans_id=ptrans_id ⑨
    SELECT trans_time INTO ptrans_time FROM transactions WHERE trans_id=ptrans_id ⑩
    DBMS_OUTPUT.PUT_LINE('the trans_time is:'||ptrans_time)
END;
/

```

- ① 创建过程param。
- ② 第一个参数为ptrans_id，类型为IN，数据类型为STRING。
- ③ 第二个参数为ptrans_time，类型为IN，数据类型为STRING。
- ④ 第三个参数为ptrans_type，类型为OUT，数据类型为STRING。
- ⑤ 第四个参数为pacc_num，类型为INOUT，数据类型为STRING。
- ⑥ 第五个参数为pprice，类型为OUT，数据类型为DOUBLE。
- ⑦ 给ptrans_type赋值。
- ⑧ 给pprice赋值。
- ⑨ 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。
- ⑩ 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。

例 123. 调用上节所创建的过程param

```

DECLARE
    v_trans_type string ①
    v_price double   ②
    v_acc_num string ③
BEGIN
    param('943197522','20140105100520',v_trans_type,v_acc_num,v_price) ④
    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) ⑤
END;
/

```

- ① 声明一个字符串类型的变量v_trans_type。
- ② 声明一个double类型的变量v_price。
- ③ 声明一个字符串类型的变量v_acc_num。
- ④ 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trans_id，'20140105100520' 对应trans_time，v_trans_type对应ptrans_type，v_acc_num对应acc_num，v_price对应pprice。
- ⑤ 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。

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
+-----+

```

4.7.2.5.2. 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 124. 创建一个含有五个参数的过程param

```

CREATE OR REPLACE PROCEDURE
param( ①
    ptrans_id IN STRING, ②
    ptrans_time IN STRING, ③
    ptrans_type OUT STRING, ④
    pacc_num INOUT STRING, ⑤
    pprice OUT DOUBLE) ⑥
IS
BEGIN
    ptrans_type:='b' ⑦
    pprice:=12.13 ⑧
    DBMS_OUTPUT.PUT_LINE('the trans_id is:'||ptrans_id)
    SELECT acc_num INTO pacc_num FROM transactions WHERE trans_id=ptrans_id ⑨
    SELECT trans_time INTO ptrans_time FROM transactions WHERE trans_id=ptrans_id ⑩
    DBMS_OUTPUT.PUT_LINE('the trans_time is:'||ptrans_time)
END;
/

```

- ① 创建过程param。
- ② 第一个参数为ptrans_id，类型为IN，数据类型为STRING。
- ③ 第二个参数为ptrans_time，类型为IN，数据类型为STRING。
- ④ 第三个参数为ptrans_type，类型为OUT，数据类型为STRING。
- ⑤ 第四个参数为pacc_num，类型为INOUT，数据类型为STRING。
- ⑥ 第五个参数为pprice，类型为OUT，数据类型为DOUBLE。
- ⑦ 给ptrans_type赋值。
- ⑧ 给pprice赋值。
- ⑨ 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。
- ⑩ 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。

例 125. 调用过程param

```

DECLARE
    v_trans_type string ①
    v_price double    ②
    v_acc_num string ③
BEGIN
    param(ptrans_id => '943197522',
          ptrans_time => '20140105100520',
          ptrans_type => v_trans_type,
          pacc_num => v_acc_num,
          pprice => v_price) ④
    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) ⑤
END;
/

```

① 声明一个字符串类型的变量v_trans_type。

② 声明一个double类型的变量v_price。

③ 声明一个字符串类型的变量v_acc_num。

④ 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。

⑤ 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。

输出结果为：

```

+-----+
|       output      |
+-----+
| the type is:b   |
| the price is:12.13 |
| the trans_id is:943197522 |
| the acc_num is:6513065 |
| the trans_time is:20140105100520 |
+-----+

```

4.7.3. 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 126. 重载过程overload_test

```

CREATE OR REPLACE PROCEDURE overload_test (place_in STRING) ①
IS
    l_message STRING;
BEGIN
    l_message := 'Hello ' || place_in;
    DBMS_OUTPUT.put_line (l_message);
END;
/
CREATE OR REPLACE PROCEDURE overload_test(v_name INT) ②
IS
    l_message STRING;
BEGIN
    l_message := v_name||'is a good number';
    DBMS_OUTPUT.put_line (l_message);
END;
/
BEGIN
    overload_test ('world'); ③
    overload_test(888); ④
    overload_test('999'); ⑤
END;
/

```

- ① 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。
- ② 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。
- ③ 过程内的实参为world，类型为字符串，则会执行第一个过程。
- ④ 过程内的实参为888，类型为整数类型，则会执行第二个过程。
- ⑤ 过程内的实参为999，类型为字符串，则会执行第一个过程。

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

```

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;
/
BEGIN
  overload_test('world');
  overload_test(888);
END;
/

```

4.7.4. 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](#)。

4.8. 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。

4.8.1. 创建函数

- 语法：

```

CREATE [OR REPLACE] FUNCTION function_name ([parameter1[mode1] datatype1, parameter2[mode2]
datatype2 ...]) ①
RETURN datatype
IS
  PL/SQL Block;

```

- ① 此处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。

例 127. 创建名为hello_message的函数

```
CREATE OR REPLACE FUNCTION
hello_message
(place_in IN STRING)
RETURN STRING
IS
BEGIN
    RETURN 'Hello ' || place_in;
END;
/
```



1. 子程序的名字描述被返回的数据。而不是被执行的过程。
2. 子程序中多了一个RETURN从句。
3. 该函数是设置函数返回值的数据类型。

4.8.2. 调用函数

4.8.2.1. 单独调用函数

例 128. 单独调用函数hello_message的方法

```
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

4.8.2.2. 在过程中调用函数

例 129. 创建过程hello_place中调用hello_message函数

```
CREATE OR REPLACE PROCEDURE
hello_place
(place_in string) ①
IS
BEGIN
    DBMS_OUTPUT.put_line (hello_message (place_in)); ②
END;
/
BEGIN
    hello_place('world');
END;
/
```

① 创建带有参数的过程hello_place。

② 调用函数hello_message。

输出结果为：

```
+-----+
| output |
+-----+
| Hello world |
+-----+
```

4.8.2.3. 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ORC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 130. 利用函数hello_message往表ra中插入一条数据

```
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 |
+-----+-----+
| smith | 3.4 |
| lisi | 3.3 |
| ll | 3.3 |
| Hello tom | 4.8 |
| lily | 3.6 |
+-----+-----+
```

可以看到，ra表中成功插入了一条数据('hello tom', 4.8)。

4.8.2.4. 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 131. 创建函数hello_message_exclamation，并在其内部调用hello_message函数

```
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 ! |
+-----+
```

4.8.3. 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，来决定应该执行的函数。

例 132. 重载函数overload_test_func

```

CREATE OR REPLACE FUNCTION overload_test_func
  (place_in string) RETURN string ①
IS
BEGIN
  RETURN 'Hello ' || place_in;
END;
/
CREATE OR REPLACE FUNCTION overload_test_func
  (place_in INT) RETURN string ②
IS
BEGIN
  RETURN place_in * place_in;
END;
/
DECLARE
  V1 STRING;
  V2 STRING;
  V3 STRING;
BEGIN
  V1 :=overload_test_func('world'); ③
  V2 :=overload_test_func(11); ④
  V3 :=overload_test_func('111'); ⑤
  DBMS_OUTPUT.PUT_LINE(V1);
  DBMS_OUTPUT.PUT_LINE(V2);
  DBMS_OUTPUT.PUT_LINE(V3);
END;
/

```

- ① 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。
- ② 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。
- ③ 函数内的实参为world，类型为字符串，则会执行第一个函数。
- ④ 函数内的实参为11，类型为整数类型，则会执行第二个函数。
- ⑤ 函数内的实参为111，类型为字符串，则会执行第一个函数。

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：

```

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;
/

```

4.8.4. 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](#)。

4.9. Records



1. 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)。
2. 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。
3. 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。

4.9.1. 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
```

4.9.2. Records创建及使用

4.9.2.1. 方法一：自定义每一个分量的名字和类型

例 133. 定义一个记录变量，查询表transactions中金额为1100的交易号

```

DECLARE
  TYPE test_record_type IS RECORD ①
    (test_trans_id string, test_trans_time string ,test_price double); ②
    testrecord1 test_record_type; ③
BEGIN
  SELECT trans_id,trans_time,price
  INTO testrecord1
  FROM transactions
  WHERE amount=1100; ④
  dbms_output.put_line(testrecord1.test_trans_id); ⑤
END;
/

```

- ① 声明一个名为test_record_type的记录类型。
- ② 分别定义test_record_type记录类型里的各个分量的数据类型。
- ③ 定义一个名为testrecord1的test_record_type变量。
- ④ 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。
- ⑤ 输出变量testrecord1中的分量test_trans_id的值。

输出结果：

```

+-----+
|   output   |
+-----+
| 895916502 |
+-----+

```

4.9.2.2. 方法二：基于表，定义每一个分量的名字和类型

例 134. 基于表中的字段类型定义每一个分量的数据类型

```

DECLARE
  TYPE test_record_type IS RECORD ①
    (test_trans_id transactions.trans_id%type,
     test_trans_time transactions.trans_time%type ,
     test_price transactions.price%type) ; ②
    testrecord2 test_record_type; ③
BEGIN
  SELECT trans_id,trans_time,price
  INTO testrecord2
  FROM transactions
  WHERE amount=1300; ④
  dbms_output.put_line(testrecord2.test_trans_time||' ' ||testrecord2.test_trans_id); ⑤
END;
/

```

- ① 声明一个名为test_record_type的记录类型。
- ② 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。
- ③ 定义一个名为testrecord2的test_record_type变量。
- ④ 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。
- ⑤ 输出变量testrecord2中的分量test_trans_time和test_trans_id的值。

输出结果为：

output
20140702113025 991691937

4.9.2.3. 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 135. 基于表，定义一个记录型变量

```

DECLARE
    test_record transactions%ROWTYPE; ①
BEGIN
    SELECT *
    INTO test_record
    FROM transactions
    WHERE stock_id='AA7105670'; ②
    dbms_output.put_line(test_record.trans_id||' '||test_record.trans_time||
    ||test_record.price|| ' '||test_record.amount); ③
END;
/

```

- ① 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。
- ② 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。
- ③ 输出记录型变量test_record中的分量trans_id, trans_time, price, amount的值。

输出结果为：

output	
943197522	20140105100520 12.13 200.0

4.10. 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，可以看作是一个一维数组，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，可以看作是一个数据字典，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。

4.10.1. 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）。

4.10.1.1. 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。

```
CREATE 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。

4.10.1.2. VARRAY声明与赋值

例 136. 变量的声明与赋值

```

DECLARE
    TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING; ①
    V_ORG_VARRAY ORG_VARRAY_TYPE; ②
BEGIN
    V_ORG_VARRAY := ORG_VARRAY_TYPE('1', '2', '3', '4', '5'); ③
    DBMS_OUTPUT.PUT_LINE('输出1: ' || V_ORG_VARRAY(1) || ',' || V_ORG_VARRAY(2) || ',' || V_ORG_VARRAY(3) || ',' || V_ORG_VARRAY(4)); ④
    DBMS_OUTPUT.PUT_LINE('输出2: ' || V_ORG_VARRAY(5)); ⑤
    V_ORG_VARRAY(5) := '5001'; ⑥
    DBMS_OUTPUT.PUT_LINE('输出3: ' || V_ORG_VARRAY(5)); ⑦
END;
/

```

- ① 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。
- ② 定义一个ORG_VARRAY_TYPE类型的VARRAY变量V_ORG_VARRAY。
- ③ 初始化变量V_ORG_VARRAY的值。
- ④ 输出变量V_ORG_VARRAY前4个成员的值。
- ⑤ 输出变量V_ORG_VARRAY第5个成员的值。
- ⑥ 给变量V_ORG_VARRAY第5个成员赋值为5001。
- ⑦ 再次输出变量V_ORG_VARRAY第5个成员的值。

输出结果为：

```

+-----+
|   output   |
+-----+
|输出1: 1、2、3、4|
|输出2: 5      |
|输出3: 5001   |
+-----+

```

4.10.1.3. VARRAY类型的使用范围

例 137. 声明的VARRAY类型和VARRAY变量，仅在当前的PL/SQL语句块中生效

```

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。

```

Error: Error while processing statement: FAILED: Error in semantic analysis:
org.apache.hadoop.hive.ql.parse.SemanticException:
ANONYMOUS BLOCK (LINE 2, COLUMN 16, TEXT "ORG_VARRAY_TYPE('1', '2', '3', '4', '5')"): BUG!
ORG_VARRAY_TYPE is not any of UDF/PL function/collection name. (state=, code=10)

```

4.10.2. 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）。



1. 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。
2. 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。

4.10.2.1. 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。

4.10.2.2. 表类型变量的声明与赋值

例 138. NESTED TABLE变量的声明与赋值

```

DECLARE
    TYPE list_of_names_t IS TABLE OF string; ①
    happyfamily    list_of_names_t:=list_of_names_t (); ②
BEGIN
    happyfamily.EXTEND (4); ③
    happyfamily (1) := 'Veva';
    happyfamily (2) := 'Chris';
    happyfamily (3) := 'Eli';
    happyfamily (4) := 'Steven'; ④
END;
/

```

- ① 定义一个名为list_of_names_t的table类型，数据类型为字符串。
- ② 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。
- ③ 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。
- ④ 依次给变量happyfamily赋值为vera, chris, eli, steven。

输出结果为：

```
+-----+
| output |
+-----+
|
```

4.10.3. 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）。

4.10.3.1. 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
```

4.10.3.2. Associative arrays 变量的声明和赋值

例 139. Associative arrays 变量的声明与赋值

```

DECLARE
  TYPE numbers_aat IS TABLE OF STRING INDEX BY INT; ①
  l_row INT; ②
  l_numbers numbers_aat; ③
BEGIN
  l_numbers (100) := 'LILY'; ④
  l_numbers (80) := 'ALICE'; ⑤
  l_row:= l_numbers.first(); ⑥
  l_row:= l_numbers.next(l_row); ⑦
  DBMS_OUTPUT.put_line (l_row||' '||l_numbers(l_row)); ⑧
END;
/

```

- ① 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。
- ② 定义一个整数类型的变量。
- ③ 定义一个名为l_numbers的numbers_aat变量。
- ④ 给变量l_numbers中下标为80的元素赋值为LILY。
- ⑤ 给变量l_numbers中下标为100的元素赋值为LILY。
- ⑥ 把变量l_numbers中第一个元素的下标赋值给l_row。
- ⑦ 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。
- ⑧ 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。

输出结果为：

```
+-----+
| output |
+-----+
| 100 LILY |
+-----+
```

4.10.4. Collections 方法

1. 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。
2. 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，有可能是字符串。
3. 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。
4. 对于类型Associative arrays，将不支持任何集合元素方法。

4.10.4.1. 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。

4.10.4.1.1. VARRAY

例 140. 利用COUNT()方法

```

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');
  FOR I IN 1..V_ORG_VARRAY.COUNT() ①
  LOOP
    DBMS_OUTPUT.PUT_LINE('V'||I||'=' || V_ORG_VARRAY(I));
  END LOOP;
END;
/

```

① 利用COUNT()方法计算出变量V_ORG_VARRAY条目的个数。

输出结果为：

output
V(1)=1
V(2)=2
V(3)=3
V(4)=4
V(5)=5

4.10.4.1.2. NESTED TABLE

例 141. 使用 COUNT() 方法遍历 NESTED TABLE 变量中的值

```

DECLARE
    TYPE list_of_names_t IS TABLE OF string; ①
    happyfamily    list_of_names_t:=list_of_names_t (); ②
BEGIN
    happyfamily.EXTEND (4); ③
    happyfamily (1) := 'Veva';
    happyfamily (2) := 'Chris';
    happyfamily (3) := 'Eli';
    happyfamily (4) := 'Steven'; ④
    FOR l_row IN 1 .. happyfamily.COUNT()
    LOOP
        DBMS_OUTPUT.put_line (happyfamily (l_row)); ⑤
    END LOOP;
END;
/

```

- ① 定义一个名为list_of_names_t的table类型，数据类型为字符串。
- ② 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。
- ③ 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。
- ④ 依次给变量happyfamily赋值为vera, chris, eli, steven。
- ⑤ 使用count方法，依次打印出变量happyfamily的值。

输出结果为：

output
Veva
Chris
Eli
Steven

4.10.4.1.3. Associative arrays

例 142. 不能用 COUNT() 方法遍历 Associative arrays 变量中的值

```

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：

```

Error: Error while processing statement: FAILED: Execution Error, return code 100 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = 100, error message = NO_DATA_FOUND
*****
ANONYMOUS BLOCK (LINE 12, COLUMN 22, TEXT "l_numbers(l_row)")
***** (state=08S01,code=100)

```

4.10.4.2. 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。

4.10.4.2.1. VARRAY

例 143. 使用FIRST()和LAST()方法

```

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');
  FOR I IN V_ORG_VARRAY.FIRST()..V_ORG_VARRAY.LAST() ①
  LOOP
    DBMS_OUTPUT.PUT_LINE('V'||I||'= ' || V_ORG_VARRAY(I));
  END LOOP;
END;
/

```

① 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。

输出结果为：

output
V(1)=1
V(2)=2
V(3)=3
V(4)=4
V(5)=5

4.10.4.2.2. Associative arrays

例 144. 使用first()和last()方法

```

DECLARE
  TYPE numbers_aat IS TABLE OF STRING INDEX BY INT; ①
  l_numbers numbers_aat; ②
  l_row INT;
  first_numbers INT;
  last_numbers INT; ③
BEGIN
  l_numbers(100) := 'LILY';
  l_numbers(90) := 'GRACE';
  l_numbers(80) := 'ALICE';
  l_numbers(95) := 'LISI'; ④
  first_numbers :=l_numbers.first()
  last_numbers:=l_numbers.last() ⑤
  DBMS_OUTPUT.put_line ('the first one:'||l_numbers(first_numbers)); ⑥
  DBMS_OUTPUT.put_line ('the last one:'||l_numbers(last_numbers));
END;
/

```

- ① 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。
- ② 定义一个名为l_numbers的numbers_aat变量。
- ③ 分别定义三个整数类型的变量。
- ④ 给变量l_numbers中的一些元素赋值，可以跳跃，也可以不按顺序。
- ⑤ 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。
- ⑥ 依次打印出变量l_numbers中的第一个和最后一个元素。

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。

4.10.4.3. 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 145. 使用PRIOR(n)方法

```

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.COUNT(); ①
  WHILE I IS NOT NULL LOOP
    DBMS_OUTPUT.PUT_LINE('V'||I||'=' || V_ORG_VARRAY(I));
    I := V_ORG_VARRAY.PRIOR(I); ②
  END LOOP;
END;
/

```

- ① 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。
- ② 返回I条目的前一个条目的下标，并赋值给I。

输出结果为：

output
V(5)=5
V(4)=4
V(3)=3
V(2)=2
V(1)=1

例 146. 使用NEXT(n)方法

```

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.FIRST(); ①
  WHILE I IS NOT NULL LOOP
    DBMS_OUTPUT.PUT_LINE('V'||I||'=' || V_ORG_VARRAY(I));
    I := V_ORG_VARRAY.NEXT(I); ②
  END LOOP;
END;
/

```

- ① 使用FIRST()返回变量V_ORG_VARRAY第一个条目的下标，并赋值给I。
- ② 返回I条目的后一个条目的下标，并赋值给I。

输出结果为：

output
V(1)=1
V(2)=2
V(3)=3
V(4)=4
V(5)=5

例 147. 同时使用PRIOR(n)方法和NEXT(n)方法

```

DECLARE
  I INT default 3; ①
  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.PRIOR(I); ②
  WHILE I IS NOT NULL LOOP
    DBMS_OUTPUT.PUT_LINE('V'||I||'= ' || V_ORG_VARRAY(I));
    I := V_ORG_VARRAY.NEXT(I); ③
  END LOOP;
END;
/

```

① 声明一个整数类型的变量，并赋初值为3。

② 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。

③ 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。

输出结果为：

output

V(2)=2
V(3)=3
V(4)=4
V(5)=5

4.10.4.4. 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VARRAY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。



4.10.4.4.1. VARRAY

例 148. 使用EXTEND(k)方法

```

DECLARE
    I INT; ①
    TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING; ②
    V_ORG_VARRAY ORG_VARRAY_TYPE; ③
BEGIN
    V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5'); ④
    I:=V_ORG_VARRAY.LAST(); ⑤
    V_ORG_VARRAY.EXTEND(2); ⑥
    V_ORG_VARRAY(I +1) := '6'; ⑦
    V_ORG_VARRAY(I +2) := '7'; ⑧
END;
/

```

- ① 声明一个INT类型的变量I。
- ② 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。
- ③ 定义一个ORG_VARRAY_TYPE类型的VARRAY变量V_ORG_VARRAY。
- ④ 初始化变量V_ORG_VARRAY的值。
- ⑤ 将变量V_ORG_VARRAY最后一个条目的下标赋值给I。
- ⑥ 给变量V_ORG_VARRAY的最后追加两个新的条目。
- ⑦ 给变量V_ORG_VARRAY新追加的第一个条目赋值为6。
- ⑧ 给变量V_ORG_VARRAY新追加的第一个条目赋值为7。

输出结果为：

```

+-----+
| output |
+-----+

```

例 149. 使用COUNT查询新的变量

```

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

4.10.4.4.2. NESTED TABLE

例 150. 使用 EXTEND() 对 NESTED TABLE 变量进行扩展

```

DECLARE
    TYPE list_of_names_t IS TABLE OF string; ①
    I INT;
    happyfamily  list_of_names_t:=list_of_names_t (); ②
BEGIN
    happyfamily.EXTEND (4); ③
    happyfamily (1) := 'Veva';
    happyfamily (2) := 'Chris';
    happyfamily (3) := 'Eli';
    happyfamily (4) := 'Steven'; ④
    I:= happyfamily.LAST(); ⑤
    happyfamily.EXTEND (2); ⑥
    happyfamily (I+1) := 'lily';
    happyfamily (I+2) := 'alice'; ⑦
    FOR l_row IN 1 .. happyfamily.COUNT
    LOOP
        DBMS_OUTPUT.put_line (happyfamily (l_row)); ⑧
    END LOOP;
END;
/

```

- ① 定义一个名为list_of_names_t的table类型，数据类型为字符串。
- ② 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。
- ③ 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。
- ④ 依次给变量happyfamily赋值
- ⑤ 将变量happyfamily最后一个元素的下标，赋值给I。
- ⑥ 使用extend方法，给变量happyfamily增加两个元素的个数。
- ⑦ 依次给新增加的元素赋值为lily，alice。
- ⑧ 使用count方法，依次打印出变量happyfamily的值。

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。

4.10.4.5. 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。

4.10.4.5.1. VARRAY

例 151. 使用TRIM(k)方法删除部分条目

```

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 152. 使用TRIM(k)方法删除全部条目

```

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

4.10.4.5.2. NESTED TABLE

例 153. 使用 TRIM() 对 NESTED TABLE 变量进行缩减

```

DECLARE
    TYPE list_of_names_t IS TABLE OF string; ①
    happyfamily    list_of_names_t:=list_of_names_t ();
BEGIN
    happyfamily.EXTEND (4); ③
    happyfamily (1) := 'Veva';
    happyfamily (2) := 'Chris';
    happyfamily (3) := 'Eli';
    happyfamily (4) := 'Steven'; ④
    happyfamily.TRIM(2); ⑤
    FOR l_row IN 1 .. happyfamily.COUNT
    LOOP
        DBMS_OUTPUT.put_line (happyfamily (l_row)); ⑥
    END LOOP;
END;
/

```

- ① 定义一个名为list_of_names_t的table类型，数据类型为字符串。
- ② 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。
- ③ 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。
- ④ 依次给变量happyfamily赋值为vera, chris, eli, steven。
- ⑤ 使用TRIM(2)方法，删除变量happyfamily最后两个元素。
- ⑥ 使用count方法，依次打印出变量happyfamily的值。

输出结果为：

-----+
output
-----+
Veva
Chris
-----+

4.10.4.5.3. Associative arrays

例 154. 不能使用 trim() 方法删除Associative arrays变量中的元素

```

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：

```

Error: Error while processing statement: FAILED: Execution Error, return code 1 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit internal exception in executing PL Task.
org.apache.hadoop.hive.ql.metadata.HiveException: Error! You can not use TRIM method with an
associate array!
*****
ANONYMOUS BLOCK (LINE 12, COLUMN 9, TEXT "trim(2)")
*****
(state=08S01, code=1)

```

4.10.4.6. 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。

4.10.4.6.1. VARRAY

例 155. 使用DELETE(K)删除第k行条目

```

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：

output
V(1)=1
V(2)=2
V(3)=3
V(4)=5

例 156. 使用DELETE()方法删除全部条目

```

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 |
+-----+
|
```

4.10.4.6.2. NESTED TABLE

例 157. 使用 DELETE() 对NESTED TABLE变量进行缩减

```

DECLARE
    TYPE list_of_names_t IS TABLE OF string; ①
    happyfamily    list_of_names_t:=list_of_names_t (); ②
BEGIN
    happyfamily.EXTEND (4); ③
    happyfamily (1) := 'Veva';
    happyfamily (2) := 'Chris';
    happyfamily (3) := 'Eli';
    happyfamily (4) := 'Steven'; ④
    happyfamily.DELETE(2); ⑤
    FOR l_row IN 1 .. happyfamily.COUNT
    LOOP
        DBMS_OUTPUT.put_line(l_row||','||happyfamily(l_row)); ⑥
    END LOOP;
END;
/

```

- ① 定义一个名为list_of_names_t的table类型，数据类型为字符串。
- ② 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。
- ③ 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。
- ④ 依次给变量happyfamily赋值为vera, chris, eli, steven。
- ⑤ 使用DELETE(2)方法，删除变量happyfamily第2个元素。
- ⑥ 使用count方法，依次打印出变量happyfamily的值。

输出结果为：

```
+-----+
| output |
+-----+
| 1,Veva |
| 2,Eli   |
| 3,Steven|
+-----+
```

4.10.4.6.3. Associative arrays

例 158. 使用DELETE方法删除Associative arrays变量中的元素

```

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';
  l_numbers.delete(80); ①
  l_row:= l_numbers.first(); ②
  l_row:= l_numbers.NEXT(l_row); ③
  DBMS_OUTPUT.put_line (l_row||' '||l_numbers(l_row)); ④
END;
/

```

- ① 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。
- ② 将变量l_numbers中第一个元素的下标赋值给l_row。
- ③ 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。
- ④ 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。

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 |
+-----+
```

4.10.4.7. 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。

4.10.4.7.1. Varray

例 159. 使用EXISTS()方法，查询VARRAY变量中第2行的数据是否存在

```

DECLARE
    l boolean; ①
    TYPE ORG_VARRAY_TYPE IS VARRAY(10) OF STRING; ②
    V_ORG_VARRAY ORG_VARRAY_TYPE; ③
BEGIN
    V_ORG_VARRAY := ORG_VARRAY_TYPE('1','2','3','4','5'); ④
    l:=V_ORG_VARRAY.exists(2); ⑤
    DBMS_OUTPUT.PUT_LINE(l); ⑥
END;
/

```

- ① 定义一个布尔类型的变量为l。
- ② 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。
- ③ 定义一个ORG_VARRAY_TYPE类型的VARRAY变量V_ORG_VARRAY。
- ④ 依次给变量V_ORG_VARRAY赋值。
- 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。
- ⑥ 输出变量l的值。

输出结果为：

```
+-----+
| output |
+-----+
| TRUE   |
+-----+
```

4.10.4.7.2. Associative Arrays

例 160. 使用EXISTS()方法，查询Associative Arrays变量中key为95的元素是否存在

```

DECLARE
    TYPE numbers_aat IS TABLE OF STRING INDEX BY INT;
    l_row INT;
    l boolean; ①
    l_numbers numbers_aat;
BEGIN
    l_numbers (100) := 'LILY';
    l_numbers (90) := 'GRACE';
    l_numbers (80) := 'ALICE';
    l_numbers (95) := 'LISI';
    l := l_numbers.exist(95); ②
    DBMS_OUTPUT.put_line (l); ③
END;
/

```

- ① 定义一个布尔类型的变量为l。
- ② 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。
- ③ 输出变量l的值。

输出结果为：

```
+-----+
| output |
+-----+
| TRUE   |
+-----+
```

4.10.4.8. 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 161. Inceptor中对变量VARRAY使用LIMIT()方法，会报出null值

```
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 162. Inceptor中对变量NESTED TABLE使用LIMIT()方法，会报出null值

```
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   |
+-----+
```

4.11. 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。

4.11.1. 显式游标

显示游标可以在SELECT语句上创建，它使用的步骤为：

- 在声明部分声明游标：

语法：

```
DECLARE CURSOR cursor_name IS select_statement;
```

- 在执行部分或异常处理部分打开游标：

语法：

```
OPEN cursor_name;
```

- 取数据：

语法：

```
FETCH cursor_name;
```

- 关闭游标：

语法：

```
CLOSE cursor_name;
```

表 1. 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。

4.11.1.1. 不带参数的显式游标使用示例

例 163. 在LOOP中声明一个显式游标，查询表transactions中的价格

```

DECLARE
    transactions_type transactions%ROWTYPE; ①
    CURSOR cur IS SELECT * FROM transactions; ②
BEGIN
    OPEN cur; ③
    LOOP
        FETCH cur INTO transactions_type; ④
        EXIT WHEN cur%NOTFOUND; ⑤
        DBMS_OUTPUT.PUT_LINE(transactions_type.price); ⑥
    END LOOP;
    CLOSE cur;
    EXCEPTION
        WHEN OTHERS THEN ⑦
            DBMS_OUTPUT.PUT_LINE('Something unexpected happened!!'); ⑧
            CLOSE cur;
    END;
/

```

- ① 定义一个字段名和类型与表transactions均相同的记录变量transactions_type。
- ② 定义一个显式游标cur，用来查询表transactions中的全部信息。
- ③ 打开游标cur。
- ④ 把游标cur查询到的信息放进变量transactions_type中。
- ⑤ 如果游标cur没有查询到信息，就退出。
- ⑥ 输出变量transactions_type中的字段price的值。
- ⑦ 声明一个异常情况名OTHERS。
- ⑧ 如果异常OTHERS发生，则返回的值。

输出结果为：

output
12.13
11.11
6.12
4.5
22.66
68.43
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 164. 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

```

DECLARE
    transactions_type transactions%ROWTYPE; ①
    CURSOR cur IS SELECT * FROM transactions; ②
BEGIN
    FOR transactions_type IN cur
    LOOP ③
        DBMS_OUTPUT.PUT_LINE( transactions_type.acc_num || ' ' || transactions_type.trans_time || '
        || transactions_type.price); ④
    END LOOP;
END;
/

```

① 定义一个字段名和类型与表transactions均相同的记录变量transactions_type。

② 定义一个显式游标cur，用来查询表transactions中的全部信息。

③ for..loop 循环语句，对于每一个在游标里的变量值。

④ 输出变量transactions_type的字段名，账号，交易时间和价格的值。

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
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 165. 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

```

DECLARE
    CURSOR transactions_cursor IS SELECT * FROM transactions; ①
    transactions_record transactions_cursor%rowtype; ②
BEGIN
    OPEN transactions_cursor;
    LOOP
        FETCH transactions_cursor INTO transactions_record;
        exit WHEN transactions_record%notfound;
        dbms_output.put_line(transactions_record.acc_num ||
        ||transactions_record.trans_time || ' ' ||transactions_record.price);
    END LOOP;
    CLOSE transactions_cursor;
END;
/

```

- ① 声明一个名为transactions_cursor的游标
- ② 声明一个transactions_cursor%rowtype类型的记录变量

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	

4.11.1.2. 带参数的显式游标使用示例

例 166. 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。

```

DECLARE
    CURSOR cur(tacc_num string) IS SELECT * FROM transactions WHERE acc_num=tacc_num; ①
    transactions_type transactions%rowtype; ②
BEGIN
    OPEN cur('6513065'); ③
    LOOP
        FETCH cur INTO transactions_type; ④
        EXIT WHEN cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(transactions_type.trans_time); ⑤
    END LOOP;
    CLOSE cur;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Something unexpected happened!!'); ⑥
        CLOSE cur;
END;
/

```

- ① 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。
- ② 定义一个字段名和类型与表transactions均相同的记录变量transactions_type。
- ③ 打开游标cur，并赋参数值为6513065。
- ④ 把游标cur查询到的信息放进变量transactions_type中。
- ⑤ 输出变量transactions_type中的交易时间。
- ⑥ 如果有情况名为OTHERS的异常发生，则返回Something unexpected happened!!的信息。

输出结果为：

output
20140105100520
20140506133109
20140916105811
20141225133500
20140628133001
20140508094805

例 167. 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

```

DECLARE
    transactions_type transactions%ROWTYPE; ①
    CURSOR cur(tacc_num string) IS SELECT * FROM transactions WHERE acc_num=tacc_num; ②
BEGIN
    FOR transactions_type IN cur('6513065')
    LOOP ③
        DBMS_OUTPUT.PUT_LINE( transactions_type.acc_num || ' ' || ' '
    transactions_type.trans_time || transactions_type.price); ④
    END LOOP;
END;
/

```

- ① 定义一个字段名和类型与表transactions均相同的记录变量transactions_type。
- ② 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。
- ③ for..loop循环，对于每一个在游标cur (' 6513065 ') 中的变量transactions_type的值。
- ④ 输出表transactions中账号为6513065的账号，交易时间和价格。

返回结果如下：

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

例 168. 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

```

DECLARE
    CURSOR cur(tacc_num string) IS SELECT trans_time, price FROM transactions WHERE acc_num =
tacc_num; ①
    transactions_type cur%ROWTYPE; ②
BEGIN
    OPEN cur('6513065'); ③
    LOOP
        FETCH cur INTO transactions_type; ④
        EXIT WHEN cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(transactions_type.acc_num || ' ' || transactions_type.trans_time
|| ' ' || transactions_type.price); ⑤
    END LOOP;
    CLOSE cur;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Something unexpected happened!!'); ⑥
        CLOSE cur;
END;
/

```

① 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。

② 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。

③ Open Cursor，并传入参数值6513065。

④ 将cur的结果放入transactions_type。

⑤ 输出变量transactions_type的字段名：账户、交易时间和价格。

⑥ 出现异常时的显示信息。

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

4.11.2. 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 2. 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
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。

4.11.2.1. 隐式游标使用示例

例 169. 指向DML语句的隐式游标，使用update语句对表zara进行更新

```
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 updated |
+-----+
```

4.11.3. 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**。

4.11.3.1. 游标变量使用示例

4.11.3.2. 强类型游标变量

例 170. 查询表transactions中价格为12.13的交易号和交易时间

```

DECLARE
    TYPE cur_transaction IS REF CURSOR RETURN transactions%ROWTYPE; ①
    sqlcur cur_transaction; ②
    v_trans_id STRING;
    v_trans_time STRING; ③
BEGIN
    OPEN sqlcur FOR SELECT trans_id,trans_time FROM transactions WHERE price=12.13; ④
    LOOP
        FETCH sqlcur INTO v_trans_id,v_trans_time; ⑤
        EXIT WHEN sqlcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_trans_id||' ' ||v_trans_time); ⑥
    END LOOP;
    CLOSE sqlcur;
END;
/

```

- ① 声明一个名为cur_transaction的游标类型，指定返回值的类型为transactions%rowtype。
- ② 定义一个名为sqlcur的游标变量。
- ③ 依次定义两个类型为字符串的变量v_trans_id, v_trans_time。
- ④ 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。
- ⑤ 将游标查询的结果放进变量v_trans_id, v_trans_time。
- ⑥ 输出变量v_trans_id, v_trans_time的值。

输出结果为：

output
943197522 20140105100520

4.11.3.2.1. 弱类型游标变量

例 171. 查询表transactions中价格为12.13的交易号和交易时间

```

DECLARE
    TYPE cur_transaction IS REF CURSOR; ①
    sqlcur cur_transaction; ②
    v_trans_id STRING;
    v_trans_time STRING; ③
BEGIN
    OPEN sqlcur FOR SELECT trans_id,trans_time FROM transactions WHERE price=12.13; ④
    LOOP
        FETCH sqlcur INTO v_trans_id,v_trans_time; ⑤
        EXIT WHEN sqlcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_trans_id||' '||v_trans_time); ⑥
    END LOOP;
    CLOSE sqlcur;
END;
/

```

- ① 声明一个名为cur_transaction的游标类型，没有定义返回值的类型。
- ② 定义一个名为sqlcur的cur_transaction游标变量。
- ③ 依次定义两个类型为字符串的变量v_trans_id, v_trans_time。
- ④ 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。
- ⑤ 将游标查询的结果放进变量v_trans_id, v_trans_time。
- ⑥ 输出变量v_trans_id, v_trans_time的值。

输出结果为：

output
943197522 20140105100520

4.11.3.2.2. 游标变量作为参数传递

例 172. 将游标变量作为参数传递

```

CREATE OR REPLACE PACKAGE pkg_a IS
    TYPE empcurtyp IS REF CURSOR RETURN user_info%ROWTYPE; ①
END pkg_a;

CREATE OR REPLACE PROCEDURE process_emp_cv(emp_cv IN pkg_a.empcurtyp)
IS ②
    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; ③
BEGIN
    OPEN emp FOR SELECT * FROM user_info;
    process_emp_cv(emp); ④
    CLOSE emp;
END;
/

```

- ① 在包pkg_a中定义一个游标类型，和表user_info的行类型相同。
- ② 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。
- ③ 声明游标变量emp，类型为在包pkg_a中定义的游标类型。
- ④ 将游标变量emp座位参数传给过程process_emp_cv。

输出结果为：

```

+-----+
|   output   |
+-----+
| Name = 祝** |
| Name = 华*  |
| Name = 魏** |
| Name = 宁** |
| Name = 邱*  |
| Name = 李*  |
| Name = 潘** |
| Name = 李** |
| Name = 管** |
+-----+

```

4.12. 与SQL的交互

4.12.1. PL/SQL过程与SQL的交互

Inceptor中可以利用PL/SQL过程对Inceptor中的ORC表进行增删改查等操作。

4.12.1.1. INSERT

例 173. 创建过程test(), 往非分区表za中插入一条记录

```

CREATE OR REPLACE PROCEDURE
test() IS ①
BEGIN
    INSERT INTO za values ('lily',20); ②
    IF SQL%FOUND THEN ③
        dbms_output.put_line('successfully insert new data'); ④
    ELSE
        dbms_output.put_line('nothing'); ⑤
    END IF;
END;
/
BEGIN
test()
END;
/

```

- ① 创建一个名为test()的过程。
- ② 往表za中插入一条数据('lily', 20)。
- ③ IF条件语句为如果SQL语句执行了。
- ④ 成功执行sql语句所输出的内容。
- ⑤ 没有成功执行sql语句所输出的内容。

输出结果为：

```
+-----+-----+
|       output      |
+-----+-----+
| successfully insert new data |
+-----+-----+
```

4.12.1.2. DELETE

例 174. 创建过程test(),删除非分区表za中年龄为40的记录

```

CREATE OR REPLACE PROCEDURE
test() IS ①
BEGIN
    DELETE FROM za WHERE age=40; ②
    IF SQL%FOUND THEN ③
        dbms_output.put_line('successfully delete the data'); ④
    ELSE
        dbms_output.put_line('nothing'); ⑤
    END IF;
END;
/
begin
test();
end;
/

```

- ① 创建一个名为test()的过程。
- ② 删除表za中年龄为40的所有记录。
- ③ IF条件语句为如果SQL语句执行了。
- ④ 成功执行sql语句所输出的内容。
- ⑤ 没有成功执行sql语句所输出的内容。

输出结果为：

```

+-----+
|       output      |
+-----+
| successfully delete the data |
+-----+

```

4.12.1.3. UPDATE

例 175. 创建过程test(),更新表ra中姓名为smith的gpa为3.4

```

CREATE OR REPLACE PROCEDURE
test() IS ①
BEGIN
    UPDATE ra SET gpa=3.4
    WHERE name='smith'; ②
    IF SQL%FOUND THEN ③
        dbms_output.put_line('successfully update the data'); ④
    ELSE
        dbms_output.put_line('nothing'); ⑤
    END IF;
END;
/
begin
test();
end;
/

```

- ① 创建一个名为test()的过程。
- ② 更新表ra中姓名为smith的绩点为3.4。
- ③ IF条件语句为如果SQL语句执行了。
- ④ 成功执行sql语句所输出的内容。
- ⑤ 没有成功执行sql语句所输出的内容。

输出结果为：

```
+-----+-----+
|       output      |
+-----+-----+
| successfully update the data |
+-----+-----+
```

4.12.1.4. MERGE

4.12.1.4.1. 仅满足条件下，更新

例 176. 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.

```

CREATE OR REPLACE PROCEDURE
test() IS ①
BEGIN
    MERGE INTO zara z USING za a ON (z.name=a.name) ②
    WHEN MATCHED
        THEN UPDATE SET z.gpa=4; ③
    IF SQL%FOUND THEN ④
        dbms_output.put_line('successfully update the data '); ⑤
    ELSE
        dbms_output.put_line('nothing'); ⑥
    END IF;
END;
/
BEGIN
test()
END;
/

```

- ① 创建一个名为test()的过程。
- ② 对表zara进行merge操作，条件为表zara和表za姓名相同。
- ③ 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。
- ④ IF条件语句为如果SQL语句执行了。
- ⑤ 成功执行sql语句所输出的内容。
- ⑥ 没有成功执行sql语句所输出的内容。

输出结果为：

```
+-----+
|       output      |
+-----+
| successfully update the data |
+-----+
```

4.12.1.4.2. 满足条件更新，不满足条件插入

例 177. 创建过程test(),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

```

CREATE OR REPLACE PROCEDURE
test() IS ①
BEGIN
    MERGE INTO za z USING zara r ON (z.name=r.name) ②
    WHEN MATCHED THEN UPDATE SET z.age=40 ③
    WHEN NOT MATCHED THEN INSERT (name,age) VALUES (r.name,r.age); ④
    IF SQL%FOUND THEN ⑤
        dbms_output.put_line('successfully update data and insert data'); ⑥
    ELSE
        dbms_output.put_line('nothing'); ⑦
    END IF
END;
/
BEGIN
test()
END;
/

```

- ① 创建一个名为test()的过程。
- ② 对表za进行merge操作，条件为表zara和表za姓名相同。
- ③ 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。
- ④ 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。
- ⑤ IF条件语句为如果SQL语句执行了。
- ⑥ 成功执行sql语句所输出的内容。
- ⑦ 没有成功执行sql语句所输出的内容。

输出结果为：

```

+-----+
|          output      |
+-----+
| successfully update data and insert data |
+-----+

```

4.12.2. 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 178. 创建函数hello_message

```
CREATE OR REPLACE FUNCTION ①
    hello_message (place_in string) ②
    RETURN string ③
IS
BEGIN
    RETURN 'Hello ' || place_in; ④
END;
/
```

- ① 创建函数的语句。
- ② 函数名为hello_message，形参为place_in，数据类型为字符串。
- ③ 函数hello_message的返回值的类型为字符串。
- ④ 函数hello_message的具体内容。

例 179. 调用函数hello_message

```
BEGIN
    INSERT INTO ra VALUES (hello_message('tom'),4.8); ①
    IF SQL%FOUND THEN ②
        dbms_output.put_line('successfully insert new data'); ③
    ELSE
        dbms_output.put_line('nothing'); ④
    END IF;
END;
/
```

- ① 往表ra中插入一条数据，其中姓名为函数来代替，即hello_message(' tom')。
- ② IF条件语句为如果SQL语句执行了。
- ③ 成功执行sql语句所输出的内容。
- ④ 没有成功执行sql语句所输出的内容。

输出结果为：

output
successfully insert new data

4.12.3. 隐式游标

例 180. 使用update语句对表zara进行更新

```

BEGIN
  UPDATE zara SET name='grace'  where age=20; ①
  IF SQL%FOUND THEN    ②
    dbms_output.put_line('name is updated'); ③
  ELSE
    dbms_output.put_line('name is not updated'); ④
  END IF;
END;
/

```

- ① 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。
- ② IF条件语句为如果SQL语句执行了。
- ③ 成功执行sql语句所输出的内容。
- ④ 没有成功执行sql语句所输出的内容。

输出结果为：

output
name is updated

4.12.4. 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，VARRAY等集合类型中。

4.12.4.1. 在fetch into中使用bulk collect

例 181. 声明一个游标，查询表transactions中账号为6513065的交易号和交易时间

```

DECLARE
    TYPE trans_type IS RECORD(trans_id string,trans_time string); ①
    CURSOR cur IS
        SELECT trans_id,trans_time
        FROM transactions WHERE acc_num=6513065; ②
    TYPE transactions_type IS TABLE OF trans_type; ③
    v_transactions_type transactions_type; ④
BEGIN
    OPEN cur; ⑤
    LOOP
        FETCH cur BULK COLLECT INTO v_transactions_type; ⑥
        EXIT WHEN cur%notfound; ⑦
    END LOOP;
    CLOSE cur; ⑧
    dbms_output.put_line(v_transactions_type.count()); ⑨
END;
/

```

- ① 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。
- ② 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。
- ③ 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。
- ④ 定义一个类型为transactions_type的变量v_transactions_type。
- ⑤ 打开游标。
- ⑥ 将游标查询到的结果放进变量v_transactions_type。
- ⑦ 如果游标没有查询到结果就退出。
- ⑧ 关闭游标。
- ⑨ 输出变量v_transactions_type的行数。

输出结果为：

```

+-----+
| output |
+-----+
| 6      |
+-----+

```

4.12.4.2. 在select into中使用bulk collect

例 182. 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

```

DECLARE
    TYPE trans_type IS RECORD(trans_id string,trans_time string,price double); ①
    TYPE transactions_type IS TABLE OF trans_type; ②
    v_transactions_type transactions_type; ③
BEGIN
    SELECT trans_id,trans_time,price
    BULK COLLECT INTO v_transactions_type
    FROM transactions; ④
    IF SQL%FOUND THEN ⑤
        dbms_output.put_line('successfully select the data into v_transactions_type'); ⑥
    ELSE
        dbms_output.put_line('nothing'); ⑦
    END IF;
END;
/

```

- ① 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。
- ② 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。
- ③ 定义一个类型为transactions_type的变量v_transactions_type。
- ④ 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。
- ⑤ IF条件语句为如果sql语句执行了。
- ⑥ 成功执行sql语句所输出的内容。
- ⑦ 没有成功执行sql语句所输出的内容。

输出结果为：

```

+-----+
|          output
+-----+
| successfully select the data into v_transactions_type |
+-----+

```

4.12.5. 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。

4.12.5.1. 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。

4.12.5.1.1. 处理DDL语句

CREATE语句

- 查看数据库中的表

```
show tables;
+-----+
| tab_name |
+-----+
| finances
| ra
| transactions
| user_info
| za
| zara
| zza
+-----+
```

例 183. 使用EXECUTE IMMEDIATE创建表test_create

```
DECLARE
  create_table  STRING;
BEGIN
  create_table:='create table ' ||'test_create' ||'(sno string , sname string )'; ①
  EXECUTE IMMEDIATE create_table;
END;
/
```

① 注意字符串与连接符之间的空格，否则Inceptor可能会报错。

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

col_name	data_type	comment	notnull_constraint	unique_constraint
sno	string			
sname	string			

例 184. 使用EXECUTE IMMEDIATE修改表test_create中列sno的列名和数据类型

```
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

```
describe test_create;
+-----+-----+-----+-----+-----+
| col_name | data_type | comment | notnull_constraint | unique_constraint |
+-----+-----+-----+-----+-----+
| sid      | int       |          |                  |                  |
| sname    | string    |          |                  |                  |
+-----+-----+-----+-----+-----+
```

DROP语句

- 以同样的方法创建表test_create1, 现在我们来用EXECUTE IMMEDIATE删除表test_create1

例 185. 使用EXECUTE IMMEDIATE删除表test_create1

```
DECLARE
  drop_table  STRING;
BEGIN
  drop_table:='drop table |||'test_create1';
  EXECUTE IMMEDIATE drop_table;
END;
/
```

再次查看数据库中的表，可以发现表test_create1已经不存在了。

4.12.5.1.2. 处理DCL语句

GRANT

例 186. 创建过程grant_priv

```
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

```
0: jdbc:hive2://localhost:10000/default> show grant user user1 on all;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| database | table | partition | column | principal_name | principal_type | privilege | grant_option | grant_time | grantor |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|          |        |          |        | user1           | USER          | CREATE     | false      | 1452264935000 | hive      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row selected (0.14 seconds)
```

REVOKE

例 187. 创建过程revoke_priv

```
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

```
0: jdbc:hive2://localhost:10000/default> show grant user user1 on all;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| database | table | partition | column | principal_name | principal_type | privilege | grant_option | grantor |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

4.12.5.1.3. 处理单行查询

例 188. 使用EXECUTE IMMEDIATE查询表中数据

```
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](#)。

4.12.5.1.4. 处理不含占位符的DML语句

INSERT+UPDATE+DELETE

例 189. 使用EXECUTE IMMEDIATE语句，分别往表orctest2中插入，更新和删除数据

```

DECLARE
    insert_table STRING;
    update_table STRING;
    delete_table STRING; ①
BEGIN
    insert_table:= 'INSERT INTO orctest2 values(20,4.0)'; ②
        EXECUTE IMMEDIATE insert_table; ③
    update_table:=' UPDATE orctest2 SET gpa=4.4 WHERE age=20'; ④
        EXECUTE IMMEDIATE update_table;
    delete_table:='DELETE FROM orctest2 WHERE age=20'; ⑤
        EXECUTE IMMEDIATE delete_table;
END;
/

```

- ① 分别声明三个字符串类型的变量insert_table, update_table, delete_table。
- ② 将SQL语句“往表orctest2中插入一条数据”赋值给变量insert_table。
- ③ 使用关键字EXECUTE IMMEDIATE执行变量insert_table里的SQL语句。
- ④ 将SQL语句“更新表orctest2中年龄为20的绩点为4.4”赋值给变量update_table。
- ⑤ 将SQL语句“删除表orctest2中年龄为20的记录”赋值给变量delete_table。

CREATE+INSERT

例 190. 使用EXECUTE IMMEDIATE创建表格并插入数据

```

DECLARE
    create_table string;
    insert_table string;
BEGIN
    create_table:='create table orctest2 (age int, gpa double)
    CLUSTERED BY (age) INTO 2 BUCKETS
    STORED AS ORC
    TBLPROPERTIES ("transactional"="true");';
    EXECUTE IMMEDIATE create_table;
    insert_table:='insert into orctest2 values (22,4.3)';
    EXECUTE IMMEDIATE insert_table;
END;
/

```

- 查看表orctest2及表中的数据

```

describe orctest2;
+-----+-----+-----+-----+-----+
| col_name | data_type | comment | notnull_constraint | unique_c |
+-----+-----+-----+-----+-----+
| age      | int       | from deserializer |          |
| gpa      | double    | from deserializer |          |
+-----+-----+-----+-----+

```

可以看到在成功创建orctest2的同时，也成功插入了一条数据

```

select * from orctest2;
+---+---+
| age | gpa |
+---+---+
| 22  | 4.3 |
+---+---+

```

CREATE+INSERT+UPDATE+DELETE+INSERT

例 191. 使用EXECUTE IMMEDIATE处理一系列DML语句

```

DECLARE
    create_table STRING;
    insert_table STRING;
    update_table STRING;
    delete_table STRING;
BEGIN
    create_table:='create table orctest3 (pid int, price double)
                  CLUSTERED BY (pid) INTO 2 BUCKETS
                  STORED AS ORC
                  TBLPROPERTIES ("transactional"="true");' ①
    EXECUTE IMMEDIATE create_table;
    insert_table:='insert into orctest3 values (111,12.13)'; ②
    EXECUTE IMMEDIATE insert_table;
    update_table:='update orctest3 set price=14.15 where pid=111'; ③
    EXECUTE IMMEDIATE update_table;
    delete_table:='delete from orctest3 where pid=111'; ④
    EXECUTE IMMEDIATE delete_table;
    insert_table:='insert into orctest3 values (112,11.11)'; ⑤
    EXECUTE IMMEDIATE insert_table;
END;
/

```

- ① 创建ORC表orctest3，字段名分为pid, price，数据类型分为INT, DOUBLE
- ② 往表orctest3中插入数据 (111, 12. 13)
- ③ 将pid为111的价格更新为14. 15
- ④ 删除表orctest3中pid为111的记录
- ⑤ 往表orctest3中插入数据 (112, 11. 11)

输出结果：

```

+-----+
| output |
+-----+

```

查看表orctest3的数据

```

select * from orctest3;
+-----+-----+
| pid | price |
+-----+-----+
| 112 | 11.11 |
+-----+-----+

```

4.12.5.1.5. 处理占位符的DML语句

PROCEDURE内的动态SQL

例 192. 在存储过程内的动态SQL

```

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 193. 使用EXECUTE IMMEDIATE语句分别处理占位符的DML语句

```

DECLARE
  insert_table STRING;
  update_table STRING;
  delete_table STRING; ①
BEGIN
  insert_table:='insert into orctest2 values (:age, :gpa)'; ②
    EXECUTE IMMEDIATE insert_table USING 24,3.3; ③
  update_table:='UPDATE orctest2 SET gpa= :newgpa WHERE age=24'; ④
    EXECUTE IMMEDIATE update_table USING 4.8; ⑤
  delete_table:='delete from orctest2 where gpa= :oldgpa'; ⑥
    EXECUTE IMMEDIATE delete_table USING 4.8; ⑦
END;
/

```

- ① 分别声明三个字符串类型的变量insert_table, update_table, delete_table。
- ② 将SQL语句“往表orctest2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。
- ③ 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。
- ④ 将SQL语句“更新表orctest2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。
- ⑤ 执行变量update_table的SQL语句，将表orctest2中年龄为24的绩点更新为4.8。
- ⑥ 将SQL语句“删除表orctest2中绩点为某个值的记录”赋值给变量delete_table。
- ⑦ 执行变量delete_table的SQL语句，删除表orctest2中绩点为4.8的记录。

CREATE+INSERT

例 194. 创建表的同时插入数据

```

DECLARE
  create_table STRING;
  insert_table STRING;
BEGIN
  create_table:='CREATE table orctest4(sid int, sno int)
    CLUSTERED BY (sid) INTO 2 BUCKETS
    STORED AS ORC
    TBLPROPERTIES ("transactional"="true");'
  EXECUTE IMMEDIATE create_table;
  insert_table:='insert into orctest4 values (:sid, :sno)';
  EXECUTE IMMEDIATE insert_table using 1,101;
  EXECUTE IMMEDIATE insert_table using 2,202;
END;
/

```

查看新建的表orctest4中的数据

```

select * from orctest4;
+-----+-----+
| sid | sno |
+-----+-----+
| 2   | 202 |
| 1   | 101 |
+-----+-----+

```

CREATE+INSERT+UPDATE

例 195. 创建表的同时插入数据并更新

```

DECLARE
  create_table STRING;
  insert_table STRING;
  update_table STRING;
BEGIN
  create_table:='CREATE table orctest5(wid int, wno int)
    CLUSTERED BY (wid) INTO 2 BUCKETS
    STORED AS ORC
    TBLPROPERTIES ("transactional"="true");'
  EXECUTE IMMEDIATE create_table;
  insert_table:='insert into orctest5 values (:wid, :wno)';
  EXECUTE IMMEDIATE insert_table using 3,303;
  EXECUTE IMMEDIATE insert_table using 4,404;
  update_table:='update orctest5 set wno= :new_wno where wid=4';
  EXECUTE IMMEDIATE update_table using 405;
END;
/

```

查看表orctest5中的数据

```

select * from orctest5;
+-----+-----+
| wid | wno |
+-----+-----+
| 4   | 405 |
| 3   | 303 |
+-----+-----+

```

CREATE+UPDATE+INSERT+DELETE

例 196. 创建表的同时插入数据更新及删除数据

```

DECLARE
    create_table STRING;
    insert_table STRING;
    update_table STRING;
    delete_table STRING;
BEGIN
    create_table:='CREATE table orctest6(pid int, pno int)
                  CLUSTERED BY (pid) INTO 2 BUCKETS
                  STORED AS ORC
                  TBLPROPERTIES ("transactional"="true");' ①
    EXECUTE IMMEDIATE create_table;
    insert_table:='insert into orctest6 values (:pid, :pno)'; ②
    EXECUTE IMMEDIATE insert_table using 11,123;
    EXECUTE IMMEDIATE insert_table using 22,246;
    EXECUTE IMMEDIATE insert_table using 33,369; ③
    update_table:='update orctest6 set pno= :new_pno where pid=11'; ④
    EXECUTE IMMEDIATE update_table using 404; ⑤
    delete_table:='delete from orctest6 where pid= :oldpid'; ⑥
    EXECUTE IMMEDIATE delete_table using 22; ⑦
END;
/

```

① 创建ORC表orctest6，字段名分别为pid, pno，数据类型均为INT。

② 使用占位符，往表中插入数据。

③ 占位符的值依次为（11, 123），（22, 246），（33, 369）。

④ 使用占位符，更新表orctest6中pid为11的pno。

⑤ 占位符的值为404。

⑥ 使用占位符，删除表中的数据。

⑦ 占位符的值为22。

查看表orctest6中的数据

可以看到首先三条数据已被成功插入到表orctest6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 orctest6;
+-----+-----+
| 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](#)。

4.12.5.2. 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。

4.12.5.2.1. 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;
```

4.12.5.2.2. 示例

例 197. 使用游标变量执行动态SQL

```

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		
289018112	6513065	20140916105811	b	UT7592845	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。

4.13. 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。

4.13.1. Packages的创建

4.13.1.1. 语法

4.13.1.1.1. 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。

4.13.1.1.2. 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。

4.13.1.1.3. Packages的调用

- 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。

4.13.1.2. 实例

4.13.1.2.1. 创建包头

例 198. 创建一个名为test的包头

```
CREATE OR REPLACE PACKAGE test ①
IS
    transid STRING; ②
    l_row int := 1; ③
    PROCEDURE test_procedure(test_name string); ④
    TYPE aa_type IS TABLE OF INTEGER; ⑤
END;
/
```

- ① 创建一个名为test的包头。
- ② 定义一个名为transid的全局变量，类型为string。
- ③ 定义一个名为l_row的全局变量，类型为int。
- ④ 声明一个名为test_procedure的过程，形参名为test_name，类型为字符串。
- ⑤ 声明一个名为aa_type的NESTED TABLE类型，用来存放整数型的数据。

4.13.1.2.2. 创建包体

例 199. 创建一个名为test的包体

```
CREATE OR REPLACE PACKAGE body test ①
IS
    PROCEDURE test_procedure(test_name string) is
    BEGIN
        dbms_output.put_line(test_name || '是个好同学! ');
    END;
END;
/
```

- ① 声明一个名为test的包体，此处必须与包头名一致。
- ② 具体定义过程test_procedure的内容。

4.13.1.2.3. 包的调用

例 200. 调用包内类型aa_type

```

DECLARE
    v_aa_type  test_aa_type:=test_aa_type(); ①
BEGIN
    v_aa_type.extend(3); ②
    v_aa_type(1) := '1001';
    v_aa_type(2) := '2001';
    v_aa_type(3):= '3001'; ③
    FOR i IN 1..v_aa_type.count()
    LOOP
        dbms_output.put_line(v_aa_type(i)); ④
    END LOOP;
    test.test_procedure('张三')
END;
/

```

- ① 声明一个包内 NESTED TABLE 类型 aa_type 的变量 v_aa_type。
- ② 给变量 v_aa_type 分配空间。
- ③ 依次给变量 v_aa_type 赋值。
- ④ 依次打印出变量 v_aa_type 里的值。

输出结果为：

-----	-----
	output
+-----+	
1001	
2001	
3001	
张三是个好同学!	
+-----+	

4.13.2. Packages的使用案例

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。所以本章中，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。

4.13.2.1. 仅创建 Packages 包头

例 201. 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

```

CREATE OR REPLACE PACKAGE aa_pkg IS ①
    TYPE aa_type IS TABLE OF INTEGER; ②
END;
/

```

- ① 创建一个名为 aa_pkg 的包头。
- ② 创建包内类型 NESTED TABLE 类型，名为 aa_type。

例 202. 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

```

CREATE OR REPLACE PROCEDURE
print_aa (aa aa_pkg.aa_type) IS ①
  i int; ②
BEGIN
  i := aa.FIRST();
  WHILE i IS NOT NULL LOOP
    DBMS_OUTPUT.PUT_LINE (i || ' ' ||aa(i));
    i := aa.NEXT(i);
  END LOOP; ③
END;
/

```

① 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。

② 定义一个整数类型的变量i。

③ 使用集合元素方法FIRST()和NEXT(i)，依次打印出aa内的值。

例 203. 创建一个aa_pkg.aa_type类型的变量为aa_var

```

DECLARE
  aa_var aa_pkg.aa_type:=aa_pkg.aa_type(); ①
BEGIN
  aa_var.extend(3); ②
  aa_var(1) := '1001';
  aa_var(2) := '2001';
  aa_var(3):= '3001'; ③
  print_aa(aa_var); ④
END;
/

```

① 创建一个类型为aa_pkg包内类型aa_type的变量aa_var，并初始化。

② 给变量aa_var分配大小为3的空间。

③ 依次给变量aa_var赋值。

④ 利用过程print_aa打印出变量的值。

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

4.13.2.2. 创建Packages包头和包体

例 204. 创建一个名为testpackage的包头

```
CREATE OR REPLACE PACKAGE testpackage ①
IS
    TYPE record_type IS RECORD(trans_time string,price double); ②
    TYPE cur IS REF CURSOR; ③
    PROCEDURE testprocedure(); ④
END;
/
```

- ① 创建一个名为testpackage的包头。
- ② 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。
- ③ 声明一个名为cur的动态游标。
- ④ 声明一个名为testprocedure()的过程，且不带参数。

例 205. 创建一个名为testpackage的包体

```
CREATE OR REPLACE PACKAGE BODY testpackage ①
IS
    v_record_type record_type; ②
    sqlcur cur; ③
    PROCEDURE testprocedure() IS ④
BEGIN
    OPEN sqlcur FOR
        SELECT trans_time,price
        FROM transactions
        WHERE trans_id=943197522; ⑤
    LOOP
        FETCH sqlcur INTO v_record_type; ⑥
        EXIT WHEN sqlcur%notfound ; ⑦
        dbms_output.put_line(v_record_type.trans_time||' '||v_record_type.price); ⑧
    END LOOP;
    CLOSE sqlcur; ⑨
END;
/
```

- ① 创建一个名为testpackage的包体，此处和包头名相同。
- ② 定义一个类型为record_type的变量v_record_type。
- ③ 定义一个游标类型为cur的游标变量sqlcur。
- ④ 具体定义过程testprocedure()的内容。
- ⑤ 打开游标sqlcur，去查询表transactions中交易号为943197522的交易时间和价格。
- ⑥ 将游标查询到的结果放进变量v_record_type里。
- ⑦ 如果游标没有查询到结果，就退出。
- ⑧ 输出变量v_record_type的分量trans_time和分量price的值。
- ⑨ 关闭游标。

例 206. 调用包内函数

```
BEGIN
    testpackage.testprocedure() ①
END;
/
```

① 调用包内过程testpackage. testprocedure()，此处不可以直接调用testprocedure()。

输出结果为：

output
20140105100520 12.13

4.13.3. 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](#)。

4.14. 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`

4.14.1. 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。

```
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)
```

4.14.2. 案例

4.14.2.1. set_env与get_env

例 207. set_env与get_env的使用

```

DECLARE
    a STRING
    b STRING
BEGIN
    set_env('aa','hello') ①
    a := get_env('aa') ②
    DBMS_OUTPUT.PUT_LINE('the value of a is: '||a) ③
    set_env('bb','world')
    b:= get_env('bb')
    DBMS_OUTPUT.PUT_LINE('the value of a is: '||b)
END;
/

```

- ① 调用过程set_env， 定义一个名为aa的环境变量， 值为hello。
- ② 调用函数get_env， 获取环境变量aa的值，并赋值给a。
- ③ 输出变量a的值，可以发现变量a与环境变量aa的值相同。

输出结果为：

```
+-----+
|       output      |
+-----+
| the value of a is: hello |
| the value of a is: world |
+-----+
```

4.14.2.2. sqlcode与sqlerrm

例 208. sqlcode与sqlerrm的使用

```

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
        v_code := sqlcode() ①
        v_errm := sqlerrm() ②
        dbms_output.put_line('error code is: '||v_code);
        dbms_output.put_line('error message is: '||v_errm);
END;
/

```

- ① 当异常发生时，获取当前异常的Error code，并赋值给变量v_code。
- ② 当异常发生时，获取当前异常的Error message，并赋值给变量v_errm。

输出结果为：

```
+-----+
|       output      |
+-----+
| error code is: -1422 |
| error message is: TOO_MANY_ROWS |
+-----+
```

4.14.2.3. get_columns

例 209. get_columns 的使用

```

CREATE OR REPLACE PROCEDURE get_columns_test()
IS
    TYPE nest_table IS TABLE OF STRING;
    TYPE nest_table_1 IS TABLE OF STRING;
    a_table nest_table; ①
    b_table nest_table_1 default null; ②
    s string;
BEGIN
    a_table := nest_table('this','is','a','table','ID'); ③
    b_table := get_columns("transactions",a_table); ④
    DBMS_OUTPUT.PUT_LINE('a_table:');
    FOR i IN a_table.first() .. a_table.last() LOOP
        DBMS_OUTPUT.PUT_LINE(a_table(i)); ⑤
    END LOOP;
    IF b_table is not null THEN
        FOR i IN b_table.first() .. b_table.last() LOOP
            s := s || b_table(i) || ", "; ⑥
        END LOOP;
    END;
    DBMS_OUTPUT.PUT_LINE('b_table:');
    s := substr(s,1,oracle_instr(s,',',-1)-1); ⑦
    DBMS_OUTPUT.PUT_LINE(s); ⑧
END;
/
BEGIN
    get_columns_test();
END;
/

```

- ① 声明一个已定义好的nest_table类型的变量a_table。
- ② 声明一个已定义好的nest_table1类型的变量b_table， 默认为null值。
- ③ 给变量a_table赋值。
- ④ 使用get_columns函数， 获取表transactions的列名，并赋值给变量b_table。
- ⑤ 依次打印出变量a_table的值。
- ⑥ 将变量b_table的值用逗号连接起来，并赋值给变量s。
- ⑦ 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。
- ⑧ 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。

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
+-----+

```

4.14.2.4. raise_application_error

例 210. raise_application_error的使用

```

DECLARE
    num_tables  INT;
BEGIN
    SELECT COUNT(*) INTO num_tables FROM transactions;  ①
    IF num_tables < 1000 THEN
        RAISE_APPLICATION_ERROR
            (-20101, 'Expecting at least 1000 tables');  ②
    ELSE
        dbms_output.put_line(num_tables);  ③
    END IF;
END;
/

```

- ① 查询表transactions的行数，并赋值给变量num_tables。
- ② 如果表transactions的行数小于1000，则抛出异常。
- ③ 否则，就输出表transactions的行数。

输出结果为：

```

Error: Error while processing statement: FAILED: Execution Error, return code -20101 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -20101, error message = Expecting at least 1000 tables
*****
System function raise_application_error (LINE -1, COLUMN -1, TEXT "null")
ANONYMOUS BLOCK (LINE 6, COLUMN 0, TEXT "RAISE_APPLICATION_ERROR
(-20101, 'Expecting at least 1000 tables');");
*****
(state=08S01,code=-20101)

```

例 211. 上述例子不抛出异常的情况

```

DECLARE
    num_tables  INT;
BEGIN
    SELECT COUNT(*) INTO num_tables FROM transactions;  ①
    IF num_tables < 10  THEN
        RAISE_APPLICATION_ERROR
            (-20101, 'Expecting at least 1000 tables');  ②
    ELSE
        dbms_output.put_line(num_tables);  ③
    END IF;
END;
/

```

- ① 查询表transactions的行数，并赋值给变量num_tables。
- ② 如果表transactions的行数小于10，则抛出异常。
- ③ 否则，就输出表transactions的行数。

输出结果为：

-----	-----
	output
-----+-----	-----+-----
20	-----+-----
-----+-----	

4.14.2.5. 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 212. 程序正常运行时，PUT_LINE的打印

```

CREATE OR REPLACE FUNCTION  put_line_test(v1 int)
  RETURN DOUBLE          ①
IS
  v2 DOUBLE
  v3 STRING
BEGIN
  put_line(null)      ②
  DBMS_OUTPUT.PUT_LINE(v1)
  v2 := 2
  v3 := "I'm a string."
  put_line(v2)        ③
  DBMS_OUTPUT.PUT_LINE('v2: ' || v2)
  put_line('v2 + v1') ④
  DBMS_OUTPUT.PUT_LINE(v2 + v1)
  put_line(v3)        ⑤
  v3 := null
  DBMS_OUTPUT.PUT_LINE(v3)
  return v2 * v1    ⑥
END;
/
BEGIN
  dbms_output.put_line("Executing put_line_test(1)")
  put_line(put_line_test(1)) ⑦
END;
/

```

- ① 创建名为put_line_test的函数，形参v1的数据类型为整型，返回值为双精度类型。
- ② 使用put_line函数，打印出null值。
- ③ 使用put_line函数，打印出变量v2的值。
- ④ 使用put_line函数，打印出字符串v2+v1。
- ⑤ 使用put_line函数，打印出v3的值。
- ⑥ 函数返回值为v1和v2的乘积。
- ⑦ 使用put_line函数，打印出put_line_test(1)的全部值。

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

4.14.2.6. 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 213. 使用包内过程，打印结果

```
BEGIN
FOR test IN 1..5 LOOP
dbms_output.put_line('test:'||test); ①
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 |
+-----+
```

4.14.2.7. owa_util

例 214. 创建过程outer_proc()

```

CREATE OR REPLACE PROCEDURE outer_proc() ①
IS
v_owner  STRING
v_name   STRING
v_lineno INT
v_type   STRING ②
BEGIN
  DBMS_OUTPUT.PUT_LINE('Outer outer proc')
  owa_util.who_called_me(v_owner, v_name, v_lineno, v_type) ③
  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) ④
END;
/
BEGIN
  outer_proc();
END;
/

```

① 创建一个名为outer_proc()的过程。

② 分别声明四个变量。

③ 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。

④ 依次输出四个变量的值。

输出结果为：

output	
Outer outer proc	
Owner:	
Name:	
Lineno: 2	
Type: ANONYMOUS_BLOCK	

4.15. 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](#)。

4.15.1. 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。

4.15.1.1. 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 215. 查询transactions表中账号为6513064的交易时间

```

DECLARE
    test string; ①
BEGIN
    SELECT trans_time
    INTO test
    FROM transactions
    WHERE acc_num=6513064; ②
    dbms_output.put_line(test); ③
END;
/

```

- ① 声明一个string类型的test。
- ② 将账号为6513065的交易时间的信息放进test。
- ③ 输出test的值。

可以看到Inceptor会抛出一个NO_DATA_FOUND的异常。

```

Error: Error while processing statement: FAILED: Execution Error, return code 100 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = 100, error message = NO_DATA_FOUND
*****
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT "")
***** (state=08S01, code=100)

```

例 216. 在查询的过程中直接引用一个名为NO_DATA_FOUND的EXCEPTION

```

DECLARE
    test1 string; ①
BEGIN
    SELECT trans_time
    INTO test1
    FROM transactions
    WHERE acc_num=6513064; ②
    dbms_output.put_line(test1); ③
EXCEPTION
    WHEN NO_DATA_FOUND THEN ④
        dbms_output.put_line('no values'); ⑤
END;
/

```

- ① 声明一个string类型的test。
- ② 将账号为6513065的交易时间的信息放进test。
- ③ 输出test的值。
- ④ 引用一个NO_DATA_FOUND的EXCEPTION。
- ⑤ 当异常发生时，所输出的信息。

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 |
+-----+

```

4.15.1.2. TOO_MANY_ROWS

执行 select into 时，结果集超过一行。

例 217. 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

```
DECLARE
    test_record transactions%rowtype; ①
BEGIN
    SELECT *
    INTO test_record
    FROM transactions
    WHERE acc_num=6513065; ②
    dbms_output.put_line(test_record.trans_time); ③
END;
/
```

- ① 基于transactions表定义一个名为test_record的记录变量。
- ② 将账号为6513065的全部信息放进test_record记录变量。
- ③ 输出test_record的交易时间。

可以看到Inceptor会抛出一个名为TOO_MANY_ROWS的异常。

```
Error: Error while processing statement: FAILED: Execution Error, return code -1422 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -1422, error message = TOO_MANY_ROWS
*****
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT "")
*****
(state=08S01, code=-1422)
```

例 218. 在上面的查询中增加一个名为TOO_MANY_ROWS的EXCEPTION

```
DECLARE
    test_record transactions%rowtype; ①
BEGIN
    SELECT *
    INTO test_record
    FROM transactions
    WHERE acc_num=6513065; ②
    dbms_output.put_line(test_record.trans_time); ③
EXCEPTION
    WHEN TOO_MANY_ROWS THEN ④
        dbms_output.put_line('too many rows'); ⑤
END;
/
```

- ① 基于transactions表定义一个名为test_record的记录变量。
- ② 将账号为6513065的全部信息放进test_record记录变量。
- ③ 输出test_record的交易时间。
- ④ 引用一个名为TOO_MANY_ROWS的EXCEPTION。
- ⑤ 当异常发生时，所输出的信息。

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   |
+-----+
```

4.15.1.3. 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 219. 游标的正确使用方法

```

DECLARE
  CURSOR cur_tran IS
    SELECT trans_id,trans_time,price,amount
      from transactions WHERE acc_num=6513065; ①
  v_trans_id transactions.trans_id%TYPE;
  v_trans_time transactions.trans_time%TYPE;
  v_price transactions.price%TYPE;
  v_amount transactions.amount%TYPE; ②
BEGIN
  OPEN cur_tran; ③
  LOOP
    FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount; ④
    EXIT WHEN cur_tran%notfound; ⑤
    dbms_output.put_line(v_trans_id); ⑥
  END LOOP;
  CLOSE cur_tran; ⑦
END;
/

```

- ① 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。
- ② 分别定义四个%type类型的变量。
- ③ 打开游标。
- ④ 将游标查询的结果依次放入变量中。
- ⑤ 如果没有找到游标就退出。
- ⑥ 输出查询结果中的交易号。
- ⑦ 关闭游标。

正确的输出结果为：

output
943197522
499506900
289018112
895916502
404905188
213859826

例 220. 打开一个已经打开了的游标

```

DECLARE
    CURSOR cur_tran IS
        SELECT trans_id,trans_time,price,amount
        FROM transactions WHERE acc_num=6513065; ①
        v_trans_id transactions.trans_id%TYPE;
        v_trans_time transactions.trans_time%TYPE;
        v_price transactions.price%TYPE;
        v_amount transactions.amount%TYPE; ②
BEGIN
    OPEN cur_tran; ③
    LOOP
        FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount; ④
        EXIT WHEN cur_tran%notfound; ⑤
    END LOOP;
    OPEN cur_tran; ⑥
END;
/

```

- ① 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。
- ② 分别定义四个%type类型的变量。
- ③ 第一次打开游标。
- ④ 将游标查询的结果依次放入变量中。
- ⑤ 如果没有找到游标就退出。
- ⑥ 第二次打开游标。

可以看出，Inceptor会抛出一个CURSOR_ALREADY_OPEN的异常

```

Error: Error while processing statement: FAILED: Execution Error, return code -6511 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -6511, error message = CURSOR_ALREADY_OPEN
*****
ANONYMOUS BLOCK (LINE 12, COLUMN 5, TEXT "open cur_tran")
***** (state=08S01,code=-6511)

```



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 221. 打开一个已经打开了的游标，并增加一个CURSOR_ALREADY_OPEN的EXCEPTION

```

DECLARE
    CURSOR cur_tran IS
        SELECT trans_id,trans_time,price,amount
        FROM transactions WHERE acc_num=6513065; ①
        v_trans_id transactions.trans_id%TYPE;
        v_trans_time transactions.trans_time%TYPE;
        v_price transactions.price%TYPE;
        v_amount transactions.amount%TYPE ; ②
BEGIN
    OPEN cur_tran; ③
    LOOP
        FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount; ④
        EXIT WHEN cur_tran%notfound; ⑤
    END LOOP;
    OPEN cur_tran; ⑥
EXCEPTION
    WHEN CURSOR_ALREADY_OPEN THEN ⑦
        dbms_output.put_line('the cursor is already open'); ⑧
END;
/

```

- ① 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。
- ② 分别定义四个%type类型的变量。
- ③ 第一次打开游标。
- ④ 将游标查询的结果依次放入变量中。
- ⑤ 如果没有找到游标就退出。
- ⑥ 第二次打开游标。
- ⑦ 引用一个CURSOR_ALREADY_OPEN的EXCEPTION。
- ⑧ 当异常发生时，所输出的信息。

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 |
+-----+
```

4.15.1.4. 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 222. 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

```

DECLARE
    CURSOR cur_tran IS SELECT * from transactions WHERE acc_num=6513065; ①
    v_trans_id transactions.trans_id%TYPE;
    v_trans_time transactions.trans_time%TYPE;
    v_price transactions.price%TYPE;
    v_amount transactions.amount%TYPE; ②
BEGIN
    OPEN cur_tran; ③
    LOOP
        FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount; ④
        EXIT WHEN cur_tran%notfound; ⑤
        dbms_output.put_line(v_trans_id); ⑥
    END LOOP;
    CLOSE cur_tran; ⑦
END;
/

```

- ① 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。
- ② 分别定义四个%type类型的变量。
- ③ 第一次打开游标。
- ④ 将游标查询的结果依次放入4个变量中。
- ⑤ 如果没有找到游标就退出。
- ⑥ 输出查询结果中的交易号信息。
- ⑦ 关闭游标。

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。

```

Error: Error while processing statement: FAILED: Execution Error, return code -6504 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -6504, error message = ROWTYPE_MISMATCH
*****
ANONYMOUS BLOCK (LINE 9, COLUMN 6, TEXT "FETCH cur_tran INTO
v_trans_id,v_trans_time,v_price,v_amount")
***** (state=08S01,code=-6504)

```

例 223. 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

```

DECLARE
    CURSOR cur_tran IS SELECT * from transactions WHERE acc_num=6513065; ①
    v_trans_id transactions.trans_id%TYPE;
    v_trans_time transactions.trans_time%TYPE;
    v_price transactions.price%TYPE;
    v_amount transactions.amount%TYPE ; ②
BEGIN
    OPEN cur_tran; ③
    LOOP
        FETCH cur_tran INTO v_trans_id,v_trans_time,v_price,v_amount; ④
        EXIT WHEN cur_tran%notfound; ⑤
        dbms_output.put_line(v_trans_id); ⑥
    END LOOP;
    CLOSE cur_tran; ⑦
EXCEPTION
    WHEN ROWTYPE_MISMATCH THEN ⑧
        dbms_output.put_line('Row type mismatch, fetching transactions data ...'); ⑨
END;
/

```

- ① 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。
- ② 分别定义四个%type类型的变量。
- ③ 第一次打开游标。
- ④ 将游标查询的结果依次放入4个变量中。
- ⑤ 如果没有找到游标就退出。
- ⑥ 输出查询结果中的交易号信息。
- ⑦ 关闭游标。
- ⑧ 引用一个ROWTYPE_MISMATCH的EXCEPTION。
- ⑨ 当异常发生时输出的信息。

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 ... |
+-----+-----+
```

4.15.1.5. 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 224. 声明一个VARRAY类型，最大容量为5的变量

```

DECLARE
  TYPE test_array  IS VARRAY(20)  OF DOUBLE ; ①
  v_test test_array ; ②
BEGIN
  v_test:=test_array(123456); ③
  dbms_output.put_line('v_test(1):'||v_test(2)); ④
END;
/

```

- ① 声明一个最大容量为20，元素数据类型为DOUBLE的VARRAY类型test_array。
- ② 声明一个名为v_test，类型为test_array的VARRAY变量。
- ③ v_test变量中仅有一个值为123456的元素。
- ④ 输出v_test下标为6的值。

可以看到，Inceptor会抛出Subscript beyond count的异常。

```

Error: Error while processing statement: FAILED: Execution Error, return code -6533 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -6533, error message = Subscript beyond count
*****
ANONYMOUS BLOCK (LINE 5, COLUMN 35, TEXT "v_test(2)")
***** (state=08S01, code=-6533)

```

例 225. 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

```

DECLARE
  TYPE test_array  IS VARRAY(5)  OF DOUBLE ; ①
  v_test test_array; ②
BEGIN
  v_test:=test_array(123456); ③
  dbms_output.put_line('v_test(1):'||v_test(6)); ④
EXCEPTION
  WHEN SUBSCRIPT_BEYOND_COUNT THEN ⑤
  dbms_output.put_line('subscript out of range'); ⑥
END;
/

```

- ① 声明一个最大容量为20，元素数据类型为DOUBLE的VARRAY类型test_array。
- ② 声明一个名为v_test，类型为test_array的VARRAY变量。
- ③ v_test变量中仅有一个值为123456的元素。
- ④ 输出v_test下标为6的值。
- ⑤ 引用一个SUBSCRIPT_BEYOND_COUNT的异常。
- ⑥ 异常发生时，所输出的值。

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 |
+-----+

```

4.15.1.6. 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 226. 声明一个VARRAY类型，最大值为20，元素下标为-1

```

DECLARE
    TYPE test_array IS VARRAY(20) OF DOUBLE; ①
    v_test test_array; ②
BEGIN
    v_test:=test_array(123456); ③
    dbms_output.put_line('v_test(-1):'||v_test(-1)); ④
END;
/

```

- ① 声明一个最大容量为20，元素数据类型为DOUBLE的VARRAY类型test_array。
- ② 声明一个名为v_test，类型为test_array的VARRAY变量。
- ③ v_test变量中仅有一个值为123456的元素。
- ④ 输出v_test下标为-1的值。

可以看到，Inceptor会抛出一个Subscript outside of limit的异常。

```

Error: Error while processing statement: FAILED: Execution Error, return code -6532 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -6532, error message = Subscript outside of limit
*****
ANONYMOUS BLOCK (LINE 5, COLUMN 36, TEXT "v_test(-1)")
***** (state=08S01, code=-6532)

```

例 227. 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

```

DECLARE
    TYPE test_array IS VARRAY(20) OF DOUBLE; ①
    v_test test_array; ②
BEGIN
    v_test:=test_array(123456); ③
    dbms_output.put_line('v_test(-1):'||v_test(-1)); ④
EXCEPTION
    WHEN SUBSCRIPT_OUTSIDE_LIMIT THEN ⑤
        dbms_output.put_line('please check the subscript'); ⑥
END;
/

```

① 声明一个最大容量为20，元素数据类型为DOUBLE的VARRAY类型test_array。

② 声明一个名为v_test，类型为test_array的VARRAY变量。

③ v_test变量中仅有一个值为123456的元素。

④ 输出v_test下标为-1的值。

⑤ 引用一个SUBSCRIPT_OUTSIDE_LIMIT的异常。

⑥ 异常发生时，所输出的值。

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 |
+-----+
```

4.15.1.7. 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 228. 声明一个集合类型，并没有赋予其初始的值

```

DECLARE
    TYPE emp_ssn_array IS TABLE OF DOUBLE; ①
    best_acc_num emp_ssn_array; ②
BEGIN
    best_acc_num(0):='6513065'; ③
    dbms_output.put_line('best_acc_num(0):'||best_acc_num(0)); ④
END;
/

```

- ① 声明一个table类型。
- ② 声明一个名为best_acc_num的table变量。
- ③ 直接给best_acc_num(0)赋值。
- ④ 输出best_acc_num(0)的值。

可以看到Inceptor会抛出没有初始化集合元素的异常。

```

Error: Error while processing statement: FAILED: Execution Error, return code -6531 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -6531, error message = Reference to uninitialized collection
*****
ANONYMOUS BLOCK (LINE 4, COLUMN 0, TEXT "best_acc_num(0)")
***** (state=08S01, code=-6531)

```

例 229. 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

```

DECLARE
    TYPE emp_ssn_array IS TABLE OF DOUBLE; ①
    best_acc_num emp_ssn_array; ②
BEGIN
    best_acc_num(0):='6513065'; ③
    dbms_output.put_line('best_acc_num(0):'||best_acc_num(0)); ④
EXCEPTION
    WHEN collection_is_null THEN ⑤
        dbms_output.put_line('please initialize the collection element'); ⑥
END;
/

```

- ① 声明一个table类型。
- ② 声明一个名为best_acc_num的table变量。
- ③ 直接给best_acc_num(0)赋值。
- ④ 输出best_acc_num(0)的值。
- ⑤ 引用一个collection_is_null的异常。
- ⑥ 异常发生时，所输出的值。

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 |
+-----+

```

4.15.1.8. 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 230. 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

```

DECLARE
  CURSOR cur_tran IS SELECT trans_id,trans_time,price,amount from transactions WHERE
acc_num=6513065; ①
  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
  OPEN cur_tran; ②
  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); ③
  END LOOP;
  CLOSE cur_tran; ④
  CLOSE cur_tran; ⑤
END;

```

- ① 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。
- ② 打开游标。
- ③ 输出游标查询结果中的交易时间。
- ④ 关闭游标。
- ⑤ 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。

输出结果为：

```

Error: Error while processing statement: FAILED: Execution Error, return code -1001 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -1001, error message = INVALID_CURSOR
*****
ANONYMOUS BLOCK (LINE 15, COLUMN 9, TEXT "CLOSE cur_tran;")
*****
(state=08S01,code=-1001)

```

例 231. 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

```

DECLARE
    CURSOR cur_tran IS SELECT trans_id,trans_time,price,amount from transactions WHERE
acc_num=6513065; ①
    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
    OPEN cur_tran; ②
    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); ③
    END LOOP;
    CLOSE cur_tran; ④
    CLOSE cur_tran; ⑤
EXCEPTION
    WHEN INVALID_CURSOR THEN ⑥
        dbms_output.put_line('THE CURSOR IS INVALID');
END;

```

- ① 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。
- ② 打开游标。
- ③ 输出游标查询结果中的交易时间。
- ④ 关闭游标。
- ⑤ 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。
- ⑥ 增加一个EXCEPTION的部分，如果程序运行过程中出现了INVALID_CURSOR的异常，则会打印出THE CURSOR IS INVALID的信息。

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。

4.15.2. 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，并对其进行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。

4.15.2.1. 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

```
DESCRIBE transactions;
+-----+-----+-----+-----+-----+
| col_name | data_type | comment | notnull_constraint | unique_constraint |
+-----+-----+-----+-----+-----+
| trans_id | string    |          | NULL             | NULL            |
| acc_num  | string    |          | NULL             | NULL            |
| trans_time | string   |          | NULL             | NULL            |
| trans_type | string   |          | NULL             | NULL            |
| stock_id | string   |          | NULL             | NULL            |
| price    | double   |          | NULL             | NULL            |
| amount   | double   |          | NULL             | NULL            |
+-----+-----+-----+-----+-----+
7 rows selected (0.329 seconds)
```

- 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 |
+-----+-----+-----+-----+-----+-----+
| 943197522 | 6513065 | 20140105100520 | b           | AA7105670 | 12.13  |
| 499506900 | 6513065 | 20140506133109 | s           | CA2789982 | 6.12   |
| 289018112 | 6513065 | 20140916105811 | b           | UT7592045 | 9.81   |
| 895916502 | 6513065 | 20141225133500 | s           | KC9102028 | 7.49   |
| 404905188 | 6513065 | 20140628133001 | b           | SH6277444 | 7.02   |
| 213859826 | 6513065 | 20140508094805 | b           | CL2121979 | 18.38  |
+-----+-----+-----+-----+-----+-----+
```

也就是说，Inceptor并不支持这种INVALID_NUMBER异常。

4.15.2.2. 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 232. 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

```
DECLARE
  l_name varchar2(2)
BEGIN
  l_name := 'steven';
END;
/
```

输出结果为：

```
+-----+
| output |
+-----+
```

例 233. 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

```
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 |
+-----+
+-----+
```

4.15.2.3. 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 234. 定义一个int变量，值为100除以0，可以看到并不会报错

```
DECLARE
    v_num int;
BEGIN
    v_num :=(100/0);
END;
/
```

输出结果为：

```
+-----+
| output |
+-----+
+-----+
```

例 235. 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

```
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 |
+-----+
+-----+
```

4.15.2.4. 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 236. 创建一个过程，向表中插入两条数据，姓名均为' lily'

```
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 |
+-----+-----+
| john | 20  |
| sara | 21  |
| fd   | 24  |
| zz   | 10  |
| lily | 22  |
| alice | 25 |
| zhangsan | 23 |
| smith | 25 |
| lily | 23 |
+-----+-----+
```

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。

4.15.2.5. 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 237. Inceptor并不会报出CASE_NOT_FOUND的异常

```

DECLARE
    message string; ①
    p_price transactions.price%type; ②
    CURSOR cur IS SELECT price FROM transactions where acc_num='6513065'; ③
BEGIN
    OPEN cur ;
    LOOP
        FETCH cur INTO p_price; ④
        EXIT WHEN cur%NOTFOUND; ⑤
        DBMS_OUTPUT.PUT_LINE(p_price); ⑥
        message:= case ⑦
            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;
        DBMS_OUTPUT.PUT_LINE(message); ⑧
    END LOOP;
    CLOSE cur; ⑨
END;
/

```

- ① 声明一个字符串类型的变量message。
- ② 声明一个变量p_price，数据类型为表transactions中价格字段的类型。
- ③ 声明游标cur用来查询表transactions中账号为6513065的价格。
- ④ 将游标查询到的结果放进变量p_price里。
- ⑤ 如果没有查询到结果就退出。
- ⑥ 依次输出所查询到的价格信息。
- ⑦ 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'。
- ⑧ 依次输出message的信息。

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。

4.15.2.6. 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 238. 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

```

DECLARE
    v_test test_type;
BEGIN
    v_test.v_name := 'test';
EXCEPTION
    when access_into_null then
        dbms_output.put_line('first initialized object v_test');
END;
/

```

输出结果为：

```

Error: Error while processing statement: FAILED: Error in semantic analysis:
org.apache.hadoop.hive.ql.parse.SemanticException:
ANONYMOUS BLOCK (LINE 1, COLUMN 15, TEXT "test_type"): Unknown identifier test_type
ANONYMOUS BLOCK (LINE 1, COLUMN 8, TEXT "v_test test_type") (state=,code=10)

```

可以看到，Inceptor并不会返回' first initialized 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。

4.15.2.7. 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 239. 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：

```

Error: Error while processing statement: FAILED: Parse Error: line 1:0 cannot recognize input
near 'create' 'type' 'ty_name' in ddl statement (state=,code=11)

```

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。

4.15.2.8. 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 240. 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：

```
Error: Error while processing statement: FAILED: Error in semantic analysis: Line 1:42 Invalid table alias or column reference 'rownum': (possible column names are: name, acc_num, password, citizen_id, bank_acc, reg_date, acc_level) (state=,code=10)
```

例 241. 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：

```
Error: Error while processing statement: FAILED: Error in semantic analysis: Line 1:34 Invalid table alias or column reference 'rownum': (possible column names are: name, acc_num, password, citizen_id, bank_acc, reg_date, acc_level) (state=,code=10)
```

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。

4.15.2.9. NOT_LOGGED_ON

PL/SQL 应用程序在没有连接 oracle 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。

4.15.2.10. 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)
Beeline version 0.12.0-transwarp-TDH40 by Apache Hive
```

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。

4.15.2.11. 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。

4.15.3. 用户自定义异常

例 242. 定义一个异常，查询transactions表中的信息

```

DECLARE
    n_count INT; ①
    too_many EXCEPTION; ②
BEGIN
    SELECT COUNT(*) INTO n_count FROM transactions; ③
    IF (n_count>5) THEN
        RAISE too_many;
    END IF; ④
    dbms_output.put_line(n_count);
EXCEPTION
    WHEN no_data_found THEN ⑤
        dbms_output.put_line('no data found'); ⑥
    WHEN too_many THEN ⑦
        dbms_output.put_line('user_defined exception found:too many'); ⑧
END;
/

```

- ① 声明一个整数类型的变量。
- ② 声明一个名为too_many的异常。
- ③ 查询transactions表中的行数信息，放进n_count变量里。
- ④ 定义异常too_many发生的条件。
- ⑤ 引用no_data_found系统预定义异常。
- ⑥ 异常发生时所输出的信息。
- ⑦ 引用too_many用户自定义异常。
- ⑧ 异常发生时所输出的信息。

输出结果

```

+-----+
|          output          |
+-----+
| user_defined exception found:too many |
+-----+

```

例 243. 当上述自定义的异常没有发生时

```

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     |
+-----+
```

4.15.4. 嵌套异常

4.15.4.1. 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。

4.15.4.2. 异常分别出现在inner block和outer block里

4.15.4.2.1. 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 244. 内部和外部的异常处理块，分别处理inner block和outer block里的异常

```

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	
-----	-----

4.15.4.2.2. 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 245. 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

```

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 |
+-----+
```

4.15.4.3. 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。

4.15.4.3.1. 内部块中的异常处理块处理inner block里的异常

例 246. 内部块中的异常处理块处理inner block里的异常

```

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 |
+-----+
```

4.15.4.3.2. 外部块中的异常处理块处理inner block里的异常

例 247. 外部块中的异常处理块处理inner block里的异常

```

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 |
+-----+
```

4.15.4.3.3. 没有异常处理块处理inner block里的异常

例 248. 没有异常处理块处理inner block里的异常

```

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。

```

Error: Error while processing statement: FAILED: Execution Error, return code 100 from
org.apache.hadoop.hive.ql.exec.PLTTask. Hit PL exception in executing PL Task.
Error code = 100, error message = NO_DATA_FOUND
*****
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT "")
*****
(state=08S01, code=100)

```

4.15.4.4. 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。

4.15.4.4.1. 外部块中的异常处理块处理outer block里的异常

例 249. 外部块中的异常处理块处理outer block里的异常

```

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
-----+	

4.15.4.4.2. 内部块中的异常处理块处理outer block里的异常

例 250. 内部块中的异常处理块处理outer block里的异常

```

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 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：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。

```

Error: Error while processing statement: FAILED: Execution Error, return code -1422 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -1422, error message = TOO_MANY_ROWS
*****
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT "")
*****
(state=08S01,code=-1422)

```

4.16. 注意事项

4.16.1. 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。

4.16.2. 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。



4.16.2.1. Beeline+InceptorServer 2

- 在server2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 251. 在server2中打开分号支持

```

!set plsqlUseSlash true ①
create or replace procedure test() is ②
declare
    var int;
begin
    var:=1;
    while var !=5
    loop
        dbms_output.put_line("while loop body"||var||"."); ③
        var := var+1;
    end loop;
end;
/
begin
    test(); ④
end;
/
!set plsqlUseSlash false ⑤
/

```

- ① 设置plsqlUseSlash属性为true。
- ② 创建一个名为test()的过程。
- ③ 在变量var不等于5的情况下，依次输出var的值。
- ④ 调用过程test()。
- ⑤ 设置plsqlUseSlash属性为false。

输出结果为：

```

+-----+
|      output      |
+-----+
| while loop body1. |
| while loop body2. |
| while loop body3. |
| while loop body4. |
+-----+

```

4.16.2.2. CLI+InceptorServer 1

- 在server1中我们可以通过以下命令

```

set plsql.use.slash=true;
set plsql.use.slash=false;

```

例 252. 在server1中打开分号支持

```
set plsql.use.slash=true;
create or replace procedure test() is
declare
  var int;
begin
  var:=1;
  while var !=5
  loop
    dbms_output.put_line("while loop body"||var||".");
    var := var+1;
  end loop;
end;
/
begin
  test();
end;
/
set plsql.use.slash=false;
/
```

输出结果为：

```
while loop body1.
while loop body2.
while loop body3.
while loop body4.
```

4.16.3. 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 253. 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

```
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;
No rows affected (0.003 seconds)
```

例 254. 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 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

4.16.4. 标识符

4.16.4.1. 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。

4.16.4.1.1. 案例一：SELECT/SELECT INTO语句后紧跟函数调用

例 255. 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。

```
Error: Error while processing statement: FAILED: Parse Error: line 6:13 cannot recognize input
near '.' 'put_line' '(' in from source (state=,code=11)
```

例 256. 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 257. 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;
/

```

输出结果为:

output
1000

4.16.4.1.2. 案例二:SELECT/SELECT INTO语句后紧跟赋值语句

例 258. 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。

```
Error: Error while processing statement: FAILED: Parse Error: line 6:5 cannot recognize input
near ':=' ''hello world'' 'dbms_output' in from source (state=,code=11)
```

例 259. 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 260. 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 |
+-----+
```

4.16.4.2. 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。

4.16.4.2.1. 案例一:END LOOP后紧跟函数调用

例 261. 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。

```
Error: Error while processing statement: FAILED: Parse Error: line 7:11 cannot recognize input
near '.' 'PUT_LINE' '(' in labeled statement (state=,code=11)
```

例 262. 解决方案:在END LOOP的结尾处加上分号

```

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

4.16.4.2.2. 案例二:END LOOP后紧跟赋值语句

例 263. 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。

```
Error: Error while processing statement: FAILED: Parse Error: line 8:3 cannot recognize input
near ':=' ''hello world again'' 'DBMS_OUTPUT' in labeled statement (state=,code=11)
```

例 264. 解决方案:在END LOOP的结尾处加上分号

```

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

4.16.5. 标准SQL调用PL/SQL函数必须满足的条件

4.16.5.1. 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。

4.16.5.1.1. 不支持SQL语句中调用PL/SQL过程

- 创建一个名为test()的过程

```
CREATE OR REPLACE PROCEDURE
test(acc int) IS
BEGIN
    put_line('651306'||acc);
END;
/
```

- 调用test()过程

```
BEGIN
    test(5);
END;
/
```

- 可以看到Inceptor中单独调用test()过程是可以的。

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

```
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) FROM user_info;
```

输出结果为：

_c0
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。

4.16.5.2. 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。

4.16.5.2.1. 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。

```
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) FROM user_info;
```

输出结果为：

_c0
115591
205239
531547
841242
986634
737297
600709
990590
015859
783438

4.16.5.2.2. 不支持PL/SQL函数返回值是非基本类型

- Inceptor中不支持PL/SQL函数返回值是非基本类型，也就是以下函数的创建方法是不支持的

```
CREATE OR REPLACE PACKAGE aa_pkg IS
  TYPE aa_type IS TABLE OF INTEGER;
END;
/
CREATE OR REPLACE FUNCTION plsql_function(t_name string)
  return aa_pkg_aa_type as
BEGIN
  DECLARE v_type aa_pkg_aa_type :=aa_pkg_aa_type('1','2','3','4','5');
  return v_type ;
END;
/
```

4.16.5.3. 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

```

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。

```
Error: Error while processing statement: FAILED: Execution Error, return code 1 from
io.transwarp.inceptor.execution.SparkTask. Job aborted due to stage failure: Task 0 in stage
38.0 failed 4 times, most recent failure: Lost task 0.3 in stage 38.0 (TID 122, tw-node128):
org.apache.hadoop.hive.ql.metadata.HiveException: PLSQL function is running in a non-driver
environment (usually in SQL statement), which doesn't allow nested SQL statement.
(state=08S01,code=1)
```

4.16.6. 不支持RETURN INTO 语句

例 265. 利用returning into 语句来返回对表user_info进行改动的记录的账号。

```

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。

```
Error: Error while processing statement: FAILED: Parse Error: line 4:18 cannot recognize input
near 'into' 'l_acc_num' 'dbms_output' in non sql statement (state=,code=11)
```

4.16.7. 查看预定义函数/过程/包

4.16.7.1. 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
```



由于我们的PLSQL是运行在Inceptor Server端的SESSION之中，当一条PLSQL语句尚未结束并且会运行很长时间的时候，可能客户端已经用Ctrl+C杀掉自己。但Server端由于还在执行PLSQL，无法侦听到客户端死掉，这样这个SESSION中的PLSQL会变成类似僵尸进程。虽然客户端已经决定放弃这次执行，但该PLSQL依然会继续执行到结束，设想其中如果有大量的SQL语句执行或者干脆有个死循环，会耗费大量的Server端资源。所以我们需要手动执行一些命令来终止Server端PLSQL的执行。

- 查看正在运行的PLSQL程序的SESSION ID

仅在InceptorServer 2}中有效

```
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

仅在InceptorServer 2中有效

```
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。



自检是借鉴了操作系统中进程调度的思想。自检的粒度为一条PLSQL语句，也就是说每执行一条PLSQL语句会自检一次（也就是被OS调度一次）。所以如果有一条SQL语句执行时间特别长，那么KILL该进程后并不会马上生效，需要等到该SQL语句执行完毕才会发生自检并终止。这种情况如果确定要终止该进程，可到Spark的4040页面去KILL掉当前的SQL，这样外层PLSQL就会立即终止。

4.16.7.2. 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)
| 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)
| 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)
| 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 default.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;
No rows affected (0.471 seconds)
```

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)
| -----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)
| 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

仅在InceptorServer 2中有效

```
PS PLSQL;
+-----+
| session id: plsql statement |
+-----+
+-----+
```

- 案例7:终止正在运行的PLSQL程序

仅在InceptorServer 2中有效

```
KILL PLSQL <SESSION_ID>
```

4.16.8. 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 266. 程序正常运行

```

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 267. 程序运行过程中出现异常， 默认只打印出异常栈

```

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：

```

Error: Error while processing statement: FAILED: Execution Error, return code 1 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = 1, error message = USER_DEFINED
*****
PROCEDURE 'p1' (LINE 9, COLUMN 4, TEXT "raise ex;")
PROCEDURE 'p2' (LINE 6, COLUMN 2, TEXT "p1(value);")
PROCEDURE 'p3' (LINE 5, COLUMN 2, TEXT "p2(value);")
ANONYMOUS BLOCK (LINE 8, COLUMN 2, TEXT "p3(value);")
***** (state=08S01, code=1)

```

例 268. 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 plsql.cache.output=true; ①
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;
/

```

① 手动设置plsql.cache.output属性为true。

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。

```

Error: Error while processing statement: FAILED: Execution Error, return code 1 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = 1, error message = USER_DEFINED
Program Output:
init value
p3: init value
p2: p3: init value
p1: p2: p3: init value
*****
PROCEDURE 'p1' (LINE 9, COLUMN 4, TEXT "raise ex;")
PROCEDURE 'p2' (LINE 6, COLUMN 2, TEXT "p1(value);")
PROCEDURE 'p3' (LINE 5, COLUMN 2, TEXT "p2(value);")
ANONYMOUS BLOCK (LINE 8, COLUMN 2, TEXT "p3(value);")
***** (state=08S01, code=1)
*****
```

4.16.9. 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 269. 默认情况下，Hive异常不被PLSQL处理

```
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：

```
Error: Error while processing statement: FAILED: Execution Error, return code 1 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit internal exception in executing PL Task.
org.apache.hadoop.hive.ql.parse.SemanticException: Update/Delete/Merge operations cannot apply
on views or non-transactional Inceptor tables
*****
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT "UPDATE transactions SET price = 12.12 WHERE trans_time
= 20140105100520;")
*****
(state=08S01,code=1)
```

例 270. 默认情况下，WHEN OTHERS语句也不会处理Hive异常

```
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：

```
Error: Error while processing statement: FAILED: Execution Error, return code 1 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit internal exception in executing PL Task.
org.apache.hadoop.hive.ql.parse.SemanticException: Update/Delete/Merge operations cannot apply
on views or non-transactional Inceptor tables
*****
ANONYMOUS BLOCK (LINE 0, COLUMN 0, TEXT "UPDATE transactions SET price = 12.12 WHERE trans_time
= 20140105100520;")
*****
(state=08S01,code=1)
```

- 手动设置变量plsql.catch.hive.exception为true

```
SET plsql.catch.hive.exception=true;
```

例 271. 手动设置命令，使用HIVE_EXCEPTION捕获Hive异常

```
SET plsql.catch.hive.exception=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：

output
Exception caught, error code = -99999, error message = Update/Delete/Merge operations cannot apply on views or non-transactional Inceptor tables

例 272. 手动设置命令，使用OTHERS捕获Hive异常

```
SET plsql.catch.hive.exception=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：

output
Exception caught, error code = -99999, error message = Update/Delete/Merge operations cannot apply on views or non-transactional Inceptor tables

4.16.10. 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 273. 打印异常栈

```

CREATE OR REPLACE PROCEDURE p1(value string)
IS
DECLARE
    ex EXCEPTION;
    pragma exception_init(ex, -1024); ①
BEGIN
    value := 'p1: ' || value;
    dbms_output.put_line(value);
    FOR i IN 1 .. 10 LOOP
        raise ex; ②
    END LOOP;
END;
/
CREATE OR REPLACE PROCEDURE p2(value string) ③
IS
BEGIN
    value := 'p2: ' || value;
    dbms_output.put_line(value);
    p1(value); ④
END;
/
CREATE OR REPLACE PROCEDURE p3(value string) ⑤
IS
BEGIN
    value := 'p3: ' || value;
    dbms_output.put_line(value);
    p2(value); ⑥
END;
/
DECLARE
    value string;
    id_v int;
    ret_v int;
BEGIN
    value := 'init value';
    dbms_output.put_line(value);
    p3(value); ⑦
END;
/

```

- ① 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。
- ② 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）。
- ③ 创建过程p2。
- ④ 在过程p2内调用过程p1。
- ⑤ 创建过程p3。
- ⑥ 在过程p3内调用过程p2。
- ⑦ 最后执行过程p3的内容。

输出结果为：

```

Error: Error while processing statement: FAILED: Execution Error, return code -1024 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -1024, error message = USER_DEFINED
*****
PROCEDURE 'p1' (LINE 9, COLUMN 4, TEXT "raise ex")
PROCEDURE 'p2' (LINE 5, COLUMN 2, TEXT "p1(value)")
PROCEDURE 'p3' (LINE 5, COLUMN 2, TEXT "p2(value)")
ANONYMOUS BLOCK (LINE 8, COLUMN 2, TEXT "p3(value)")
***** (state=08S01, code=-1024)

```

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 274. 给上述例子加上相应的异常处理，使得程序正常运行。

```

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
+-----+

```

5. Inceptor SQL PL手册（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](#)。

5.1. Inceptor SQL PL一览

5.1.1. 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>');
```

5.1.2. 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,...);
```

5.1.3. 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> ;
```

5.1.4. 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>, ... );
```

5.1.5. 函数

- **CREATE FUNCTIONS**

```
CREATE [OR REPLACE] FUNCTION <function_name> [ ( {optional parameters} ) ] [{optional function attributes}]
```

5.1.6. 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>;
```

5.1.7. 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>];
```

5.1.8. 动态SQL

- EXECUTE IMMEDIATE

```
EXECUTE IMMEDIATE <sql-statement>;
```

- PREPARE

```
PREPARE <sql-statement> FROM <variable>;
```

- EXECUTE

```
EXECUTE <statement-name> [USING <input-variable> [,<input-variable>, ...] ];
```

5.2. Inceptor SQL PL手册中的表

在Inceptor_sqlpl1手册中，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: 列从左到右依次为trans_id, acc_num, trans_time, trans_type, stock_id, price, amount;其中trans_id|acc_num|trans_time|trans_type|stock_id|均为string类型, |price|amount均为double类型。

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
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:列从左到右依次为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

- ORC非分区表zara:列从左到右依次为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

- ORC非分区表za:列从左到右依次为name, age

john	20
fd	24
zz	10
alice	25

- ORC非分区表ra:列从左到右依次为name, gpa

smith	3.4
lisi	3.3
ll	3.3
lily	3.6

- ORC非分区表orctest2:列从左到右依次为age, gpa

22	4.3
20	4.0
21	3.9

- ORC分区表zza:列从左到右依次为name, age, level(分区键为level)

zhaoliu	23	A
john	30	A
john	20	B
wangwu	22	C
lily	30	C
alice	25	C
john	20	C
sara	21	C

5.3. 基础知识

5.3.1. 快速入门

Inceptor中支持DB2 SQL PL 中的相关语法规规范，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

```
BEGIN ①
  PUT_LINE('hello world'); ②
END; ③
/
```

①语句块的开始，必要部分，不可缺省。

②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。

③语句块的结束，必要部分，不可缺省。

输出结果为：

```
+-----+
| output |
+-----+
| hello world |
+-----+
```

- 例2:声明变量，并赋值，打印变量

```

BEGIN      ①
  DECLARE v1  STRING;    ②
  SET   v1 = 'hello world';  ③
  PUT_LINE(v1);  ④
END;  ⑤
/

```

- ① 语句块的开始，必要部分，不可缺省。
- ② 声明一个名为v1的变量，数据类型为字符串。
- ③ 使用关键字SET，给变量v1赋值。
- ④ 使用PUT_LINE函数打印出变量v1的值。
- ⑤ 语句块的结束，必要部分，不可缺省。

输出结果为：

```

+-----+
|   output   |
+-----+
| hello world |
+-----+

```

5.3.2. 声明

5.3.2.1. 基本变量的声明

5.3.2.1.1. 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。

5.3.2.1.2. 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。

5.3.2.1.3. 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。

5.3.2.1.4. 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。

5.3.2.1.5. 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。

5.3.2.1.6. 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](#)。

5.3.2.2. 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

```
BEGIN
  DECLARE int_a INT CONSTANT 1; ①
  PUT_LINE(int_a); ②
END;
/
```

① 声明一个整数类型的常量int_a，值为1。

② 打印出常量int_a的值。

输出结果为：

```
+-----+
| output |
+-----+
| 1      |
+-----+
```

- 例2:字符串类型的常量

```
BEGIN
  DECLARE string_a STRING CONSTANT 'hello'; ①
  PUT_LINE(string_a); ②
END;
/
```

① 声明一个字符串类型的常量string_a，值为hello。

② 打印出常量string_a的值。

输出结果为：

```
+-----+
| output |
+-----+
| hello   |
+-----+
```

5.3.2.3. 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

```
BEGIN
DECLARE V1 STRING; ①
DECLARE V1 INT; ②
SET V1 = 'hello';
SET V1 = 100;
PUT_LINE(V1);
END;
/
```

① 声明一个字符串类型的变量V1。

② 声明一个整数类型的变量V1。

输出结果为:

```
Error: Error while processing statement: FAILED: Error in semantic analysis:
org.apache.hadoop.hive.ql.parse.SemanticException:
ANONYMOUS BLOCK (LINE 3, COLUMN 9, TEXT "DECLARE V1 INT;"): Existing variable v1
(state=,code=10)
```

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。

5.3.2.4. 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

```
BEGIN
DECLARE v_price INT; ①
DECLARE V_PRICE DOUBLE; ②
SET v_price = 100;
SET V_PRICE = 99.99;
PUT_LINE('price is: ||v_price);
PUT_LINE('PRICE IS:' ||V_PRICE);
END;
/
```

① 声明一个整数类型的变量v_price。

② 声明一个DOUBLE类型的变量V_PRICE。

输出结果为:

```
Error: Error while processing statement: FAILED: Error in semantic analysis:
org.apache.hadoop.hive.ql.parse.SemanticException:
ANONYMOUS BLOCK (LINE 3, COLUMN 9, TEXT "DECLARE V_PRICE DOUBLE;"): Existing variable v_price
(state=,code=10)
```

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。

5.3.2.5. 嵌套命名

Inceptor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

```
BEGIN
DECLARE V1 STRING;
DECLARE V2 DOUBLE; ①
SET V1 = 'hello';
SET V2 = 11.11; ②
PUT_LINE('the first v1:'||V1);
PUT_LINE('the first v2:'||V2); ③
BEGIN
DECLARE V1 STRING;
DECLARE V2 DOUBLE; ④
SET V1 = 'world';
SET V2 = 22.22; ⑤
PUT_LINE('the second v1:'||V1);
PUT_LINE('the second v2:'||V2); ⑥
END;
END;
/
```

- ① 分别声明两个STRING, DOUBLE类型的变量V1, V2。
- ② 分别给变量V1, V2赋值。
- ③ 打印出变量V1, V2的值。
- ④ 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。
- ⑤ 给变量V1, V2赋值。
- ⑥ 打印出变量V1, V2的值。

输出结果为：

output
the first v1:hello
the first v2:11.11
the second v1:world
the second v2:22.22

5.3.3. 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。

5.3.3.1. SET

- 例1：简单赋值

```
BEGIN
DECLARE V1 STRING;
DECLARE V2 INT; ①
SET V1 = 'hello world';
SET V2 = 1100; ②
PUT_LINE('V1:'||V1);
PUT_LINE('V2:'||V2); ③
END;
/
```

- ① 分别声明两个类型为整数型和字符串类型的变量V1， V2。
- ② 分别给变量V1， V2赋值。
- ③ 分别打印出变量V1， V2的值。

输出结果为：

```
+-----+
|      output      |
+-----+
| V1:hello world |
| V2:1100          |
+-----+
```

- 例2:将查询结果赋值给变量

```
BEGIN
DECLARE V1 STRING;
DECLARE V2 DOUBLE;
    SET V1 = (SELECT trans_time FROM transactions WHERE price = 11.11); ①
    SET V2 = (SELECT price FROM transactions WHERE trans_id=162742112); ②
PUT_LINE('V1:'||V1);
PUT_LINE('V2:'||V2);
END;
/
```

- ① 查询transactions表中价格为11.11的交易时间，并赋值给变量V1。
- ② 查询transactions表中交易号为162742112的价格，并赋值给变量V2。

输出结果为：

```
+-----+
|      output      |
+-----+
| V1:20140205140521 |
| V2:12.21          |
+-----+
```

5.3.3.2. VALUES INTO

- 例1:简单赋值

```
BEGIN
DECLARE V1 STRING;
DECLARE V2 INT; ①
VALUES ('hello world') INTO V1;
VALUES 1100 INTO V2; ②
PUT_LINE('V1:'||V1);
PUT_LINE('V2:'||V2); ③
END;
/
```

- ① 分别声明两个数据类型为字符串和整数类型的变量V1， V2。
- ② 分别给两个变量V1， V2赋值。
- ③ 分别打印出变量V1， V2的值。

输出结果为：

```
+-----+
|      output      |
+-----+
| V1:hello world  |
| V2:1100          |
+-----+
```

5.3.3.3. SELECT … INTO

- 例1:将查询结果赋值给变量

```
BEGIN
    DECLARE v_price DOUBLE; ①
    DECLARE v_stock STRING; ②
    SELECT price INTO v_price FROM transactions WHERE trans_time=20140328102400; ③
    SELECT stock_id INTO v_stock FROM transactions WHERE trans_time=20140628133001; ④
    PUT_LINE('price is:'||v_price); ⑤
    PUT_LINE('stockid is:'||v_stock); ⑥
END;
/
```

- ① 声明一个DOUBLE类型的变量v_price。
 ② 声明一个字符串类型的变量v_stock。
 ③ 查询表transactions中交易时间为20140328102400的价格，并赋值给变量v_price。
 ④ 查询表transactions中交易时间为20140628133001的股票交易流水号。
 ⑤ 打印出变量v_price的值。
 ⑥ 打印出变量v_stock的值。

输出结果为：

```
+-----+
|      output      |
+-----+
| price is:68.43   |
| stockid is:SH6277444 |
+-----+
```

5.4. 数据类型

5.4.1. 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。

5.4.1.1. STRING

- 语法:

```
DECLARE <variable_name> STRING [DEFAULT <value>];
```

- 例1: 声明-赋值-打印

```
BEGIN
  DECLARE V1 STRING; ①
  SET V1 = 'hello world'; ②
  PUT_LINE('V1:' || V1); ③
END;
/
```

① 声明一个字符串类型的变量V1。

② 给变量V1赋值为' hello world'。

③ 打印出变量V1的值。

输出结果为：

```
+-----+
|   output   |
+-----+
| V1:hello world |
+-----+
```

- 例2: 声明并赋值-打印

```
BEGIN
  DECLARE V4 STRING DEFAULT 'hello world'; ①
  PUT_LINE('V4:' || V4); ②
END;
/
```

① 声明一个字符串类型的变量V4，并赋初值为hello world。

② 打印出变量V4的值。

输出结果为：

```
+-----+
|   output   |
+-----+
| V4:hello world |
+-----+
```

5.4.1.2. INT

- 语法:

```
DECLARE <variable_name> INT [DEFAULT <value>];
```

- 例1: 声明-赋值-打印

```
BEGIN
  DECLARE V1 STRING; ①
  DECLARE V2 INT; ②
  SET V1 = 'hello world'; ③
  SET V2 = 100; ④
  PUT_LINE('V1:' || V1); ⑤
  PUT_LINE('V2:' || V2); ⑥
END;
/
```

- ① 声明一个字符串类型的变量V1。
- ② 声明一个整数类型的变量V2。
- ③ 给变量V1赋值为'hello world'。
- ④ 给变量V2赋值为100。
- ⑤ 打印出变量V1的值。
- ⑥ 打印出变量V2的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| V1:hello world  |
| V2:100           |
+-----+
```

- 例2: 声明并赋值-打印

```
BEGIN
  DECLARE  V3 INT  DEFAULT 10; ①
  DECLARE  V4 STRING  DEFAULT 'hello world'; ②
  PUT_LINE('V3:'||V3); ③
  PUT_LINE('V4:'||V4); ④
END;
/
```

- ① 声明一个整数类型的变量V3，并赋初值为10。
- ② 声明一个字符串类型的变量V4，并赋初值为hello world。
- ③ 打印出变量V3的值。
- ④ 打印出变量V4的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| V3:10            |
| V4:hello world   |
+-----+
```

5.4.1.3. Boolean

- 语法：

```
DECLARE <variable_name> Boolean [DEFAULT <value>];
```

- 例1: 声明-赋值-打印

```
BEGIN
  DECLARE  V1 Boolean; ①
  SET V1 = true; ②
  PUT_LINE('V1:'||V1); ③
END;
/
```

- ① 声明一个Boolean类型的变量V1。
- ② 给变量V1赋值为true。
- ③ 打印出变量V1的值。

输出结果为：

```
+-----+
| output |
+-----+
| V1:TRUE |
+-----+
```

- 例2: 声明并赋值-打印

```
BEGIN
DECLARE V4 Boolean DEFAULT false; ①
PUT_LINE('V4:'||V4); ②
END;
/
```

- ① 声明一个Boolean类型的变量V4，并赋初值为false。
- ② 打印出变量V4的值。

输出结果为：

```
+-----+
| output |
+-----+
| V4:FALSE |
+-----+
```

5.4.1.4. DATE

- 语法：

```
DECLARE <variable_name> TIMESTAMP [DEFAULT TIMESTAMP <value>];
```

- 案例1：

```
BEGIN
DECLARE date_a DATE; ①
DECLARE date_b DATE DEFAULT '2015-12-01'; ②
SET date_a = DATE '2015-10-10'; ③
PUT_LINE('date_a is:'||date_a); ④
PUT_LINE('date_b is:'||date_b); ⑤
END;
/
```

- ① 声明一个DATE类型的变量date_a。
- ② 声明一个DATE类型的变量date_b，并赋初值为2015-12-01。
- ③ 给变量date_a赋值为2015-10-10。
- ④ 打印出变量date_a的值。
- ⑤ 打印出变量date_b的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| date_a is:2015-10-10 |
| date_b is:2015-12-01 |
+-----+
```

5.4.1.5. TIMESTAMP

- 语法:

```
DECLARE <variable_name> TIMESTAMP [DEFAULT TIMESTAMP <value>];
```

- 例1: 声明-赋值-打印

```
BEGIN
    DECLARE ts_d TIMESTAMP; ①
    SET ts_d = TIMESTAMP '2011-05-05 08:09:23'; ②
    PUT_LINE(ts_d); ③
END;
/
```

① 声明一个TIMESTAMP类型的变量ts_d。

② 给变量ts_d赋值为2011-05-05 08:09:23。

③ 打印出变量ts_d的值。

输出结果为:

```
+-----+
|       output      |
+-----+
| 2011-05-05 08:09:23 |
+-----+
```

- 例2: 声明并赋值-打印

```
BEGIN
    DECLARE ts_d TIMESTAMP DEFAULT TIMESTAMP '2011-05-05 08:09:23'; ①
    PUT_LINE(ts_d); ②
END;
/
```

① 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。

② 打印出变量ts_d的值。

输出结果为:

```
+-----+
|       output      |
+-----+
| 2011-05-05 08:09:23 |
+-----+
```

5.4.1.6. 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

```
BEGIN
  DECLARE V1 STATEMENT; ①
  PREPARE V1 FROM 'SELECT price FROM transactions WHERE acc_num=6513065'; ②
END;
/
```

① 声明一个STATEMENT类型的变量V1。

② 使用关键字PREPARE将SQL语句放进变量V1里。



- 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](#)。

5.4.2. 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

5.4.3. 锚定类型

Inceptor中支持Anchored data type锚定类型

5.4.4. 特殊类型

Inceptor中支持Statement, Condition, RESULT_SET_LOCATOR三种特殊类型

5.5. 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。

5.5.1. 创建示例

- 例1: 简单匿名块的创建，直接打印常量

```
BEGIN ①
  PUT_LINE('hello world'); ②
END; ③
/
```

- ① 语句块的开始，必要部分，不可缺省。
- ② 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。
- ③ 语句块的结束，必要部分，不可缺省。

输出结果为：

```
+-----+
|   output   |
+-----+
| hello world |
+-----+
```

- 例2：稍复杂的匿名块，打印出变量

```
BEGIN      ①
DECLARE    ②
  v1  STRING;    ③
  SET  v1 = 'hello world';  ④
  PUT_LINE(v1);  ⑤
END; ⑥
/
```

- ① 语句块的开始，必要部分，不可缺省。
- ② 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。
- ③ 声明一个名为v1的变量，数据类型为字符串。
- ④ 使用关键字SET，给变量v1赋值。
- ⑤ 使用PUT_LINE函数打印出变量v1的值。
- ⑥ 语句块的结束，必要部分，不可缺省。

输出结果为：

```
+-----+
|   output   |
+-----+
| hello world |
+-----+
```

5.6. 流程控制语句

5.6.1. IF

5.6.1.1. 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

```

BEGIN
    DECLARE v_price DOUBLE;
    DECLARE v_comment STRING; ①
    SELECT price INTO v_price FROM transactions WHERE trans_id=648230055; ②
    PUT_LINE('the price is:'||v_price); ③
    IF v_price > 20 THEN ④
        SET v_comment = 'the price is too high'; ⑤
    END IF;
    PUT_LINE(v_comment); ⑥
END;
/

```

- ① 分别声明两个DOUBLE, STRING类型的变量v_price, v_comment。
- ② 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。
- ③ 输出v_price的值。
- ④ 确定IF的条件语句为v_price的值大于20。
- ⑤ 在满足IF的条件语句的情况下，给变量v_comment赋值。
- ⑥ 输出变量v_comment的值。

输出结果为：

output
the price is:22.66
the price is too high

5.6.1.2. 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

```

BEGIN
    DECLARE v_price DOUBLE;
    DECLARE v_comment STRING; ①
    SELECT price
        INTO v_price
        FROM transactions
        WHERE trans_id=943197522; ②
    PUT_LINE('the price is:'||v_price); ③
    IF v_price > 20 THEN ④
        SET v_comment = 'the price is too high'; ⑤
    ELSE
        SET v_comment = 'the price is not too high'; ⑥
    END IF;
    PUT_LINE(v_comment); ⑦
END;
/

```

- ① 分别声明两个DOUBLE, STRING类型的变量v_price, v_comment。
- ② 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。
- ③ 输出v_price的值。

- ④ 确定IF的条件语句为v_price的值大于20。
- ⑤ 在满足IF的条件语句的情况下，给变量v_comment赋值。
- ⑥ 在不满足IF的条件语句的情况下，给变量v_comment赋值。
- ⑦ 输出变量v_comment的值。

输出结果为：

```
+-----+  
|       output      |  
+-----+  
| the price is:12.13 |  
| the price is not too high |  
+-----+
```

5.6.1.3. 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

```
BEGIN
  DECLARE v_price DOUBLE;
  DECLARE v_comment STRING; ①
  SELECT price
    INTO v_price
   FROM transactions
  WHERE trans_id=943197522; ②
  put_line('the price is:'||v_price); ③
  IF v_price > 15 THEN ④
    SET v_comment = 'the price 大于 15'; ⑤
  ELSIF v_price > 10 then ⑥
    SET v_comment = 'the price 大于10'; ⑦
  ELSIF v_price > 5 then ⑧
    SET v_comment = 'the price 大于5'; ⑨
  ELSE
    SET v_comment = 'the price is too low'; ⑩
  END IF;
  put_line(v_comment);
END;
/
```

- ① 分别声明两个DOUBLE, STRING类型的变量v_price, v_comment。
- ② 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。
- ③ 输出v_price的值。
- ④ 确定IF的条件语句为v_price的值大于15。
- ⑤ 在满足IF的条件语句的情况下，给变量v_comment赋值。
- ⑥ 确定第一个ELSIF的条件语句为v_price的值大于10。
- ⑦ 在满足第一个ELSIF的条件语句的情况下，给变量v_comment赋值。
- ⑧ 确定第二个ELSIF的条件语句为v_price的值大于5。
- ⑨ 在满足第二个ELSIF的条件语句的情况下，给变量v_comment赋值。

⑩ 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。

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

```
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 |  
+-----+
```

5.6.2. FOR

- 语法：

```
FOR <loop-name> AS [<cursor-name> CURSOR [with hold] FOR ] <select-statement>  
DO  
    <inside-loop-logic>  
END FOR;
```

- 例1：

```
BEGIN  
    DECLARE sum_price double default 0; ①  
    FOR v1 AS  
        SELECT price FROM transactions WHERE trans_time=20141225133500; ②  
        DO  
            SET sum_price = sum_price + v1.price; ③  
        END FOR;  
    put_line('the price is:'||sum_price); ④  
END;  
/
```

- ① 声明一个DOUBLE类型的变量sum_price，默认值为0。
- ② 查询表transactions中交易时间为20141225133500的价格。
- ③ 将查询到的价格与变量sum_price相加的和赋值给变量sum_price。
- ④ 打印出变量sum_price的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| the price is:7.49 |
+-----+
```

5.6.3. LOOP

5.6.3.1. 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：

```
BEGIN
DECLARE v_counter INT default 0;
DECLARE v_sum INT default 0; ①
fetch_loop:
LOOP
  IF v_counter = 10 THEN
    leave fetch_loop; ②
  END IF;
  SET v_sum = v_sum + v_counter; ③
  SET v_counter = v_counter + 1; ④
END LOOP fetch_loop;
PUT_LINE('the sum is:' || v_sum); ⑤
END;
/
```

- ① 分别声明两个整数类型的变量，默认值为0。
- ② 当变量v_counter的值为10时，就退出循环。
- ③ 将变量v_sum与变量v_counter相加的和赋值给变量v_sum。
- ④ 将变量v_counter加1的值赋值给变量v_counter。
- ⑤ 打印出变量v_sum的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| the sum is:45   |
+-----+
```

5.6.4. WHILE

- 语法:

```
WHILE conditions
DO
{...statements...}
END WHILE;
```

- 例1:

```
BEGIN
DECLARE v_counter INT default 0;
DECLARE v_sum INT default 0; ①
WHILE v_counter < 10 DO ②
    SET v_sum = v_sum + v_counter; ③
    SET v_counter = v_counter + 1; ④
END WHILE;
PUT_LINE('the sum is:' || v_sum); ⑤
END;
/
```

- ① 分别声明两个整数类型的变量，默认值为0。
 ② 在变量v_counter小于10的情况下。
 ③ 将变量v_sum与变量v_counter相加的和赋值给变量v_sum。
 ④ 将变量v_counter加1的值赋值给变量v_counter。
 ⑤ 打印出变量v_sum的值。

输出结果为：

```
+-----+
|      output      |
+-----+
| the sum is:45  |
+-----+
```

5.6.5. 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:

```

BEGIN
  DECLARE v_counter INT default 0;
  DECLARE v_sum INT default 0; ①
  fetch_loop:
  REPEAT
    SET v_sum = v_sum + v_counter; ②
    SET v_counter = v_counter + 1; ③
    UNTIL v_counter > 15 ④
  END REPEAT fetch_loop;
  PUT_LINE('the sum is :'||v_sum); ⑤
END;
/

```

- ① 分别声明两个整数类型的变量，默认值为0。
- ② 将变量v_sum与变量v_counter相加的和赋值给变量v_sum。
- ③ 将变量v_counter加1的值赋值给变量v_counter。
- ④ 直到变量v_counter的值大于15。
- ⑤ 打印出变量v_sum的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| the sum is :120 |
+-----+
```

5.6.6. 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：

```

BEGIN
  DECLARE var INT default 1; ①
  print: ②
  IF var < 5 then
    SET var = var + 1; ③
    GOTO print; ④
  END IF;
  PUT_LINE('var:'||var); ⑤
END;
/

```

- ① 声明一个整数类型的变量var，并赋初值为1。
- ② 设置一个标签为print。
- ③ 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。
- ④ 跳转到标签print。
- ⑤ 打印出变量var的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| var:5          |
+-----+
```

- 例2:

```

CREATE OR REPLACE PROCEDURE num_emp(IN v_num STRING, OUT num_of_emp INT) ①
LANGUAGE SQL
BEGIN
  IF v_num='6513065'
  THEN
    SET num_of_emp = -1; ②
    GOTO putline_num; ③
  END IF;
  SELECT count(*) INTO num_of_emp FROM transactions WHERE acc_num = v_num; ④
  putline_num: ⑤
  PUT_LINE('COUNT:' || num_of_emp); ⑥
END;
/

```

① 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。

② 如果v_num的值为6513064，就给变量num_of_emp赋值为-1。

③ 跳转到标签putline_num。

④ 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。

⑤ 设置一个标签为putline_num。

⑥ 标签内的内容为打印出变量num_of_emp的值。

- 调用过程:

```

BEGIN
  DECLARE V1 INT; ①
  CALL num_emp('6513065',V1); ②
END;
/

```

① 声明一个整数类型的变量V1。

② 调用过程num_emp，参数v_num的实参为6513065。

输出结果为：

```

+-----+
| output |
+-----+
| COUNT:-1 |
+-----+

```

5.6.7. LEAVE

- 例1:

```

BEGIN
  DECLARE v_outer_index int;
  DECLARE v_sum int default 0; ①
  SET v_outer_index = 1;
  outer_loop:
    LOOP
      IF v_outer_index = 5 THEN
        leave outer_loop; ②
      END IF;
      SET v_sum = v_sum + v_outer_index;
      SET v_outer_index = v_outer_index + 1; ③
    END LOOP outer_loop;
    PUT_LINE('the outer sum:' || v_sum); ④
  END;
/

```

- ① 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。
- ② 如果v_outer_index的值为5，就离开outer_loop循环。
- ③ 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。
- ④ 打印出在outer_loop循环内，变量v_sum的值。

5.6.8. ITERATE

ITERATE语句可以在LOOP循环中，可以更早地跳出循环。

- 例1：

```

CREATE OR REPLACE PROCEDURE myproc(OUT var01 INT,OUT var02 INT) ①
BEGIN
  SET var01 = 0;
  SET var02 = 0; ②
  label2: LOOP ③
    SET var01 = var01 + 1; ④
    IF ( var01 > 100 or var02 > 100 ) THEN
      leave label2; ⑤
    END IF;
    IF var02 > 50 THEN
      iterate label2; ⑥
    END if;
    SET var02 = var02 +1; ⑦
  END LOOP;
  PUT_LINE('var01:' || var01);
  PUT_LINE('var02:' || var02); ⑧
END;
/

```

- ① 创建一个名为myproc的过程，该过程有两个形参，其参数类型均为OUT，数据类型均为整数类型。
- ② 分别给参数var01, var02赋初值为0。
- ③ 设置标签label2。
- ④ 标签内的内容为：给变量var01不断加1并赋值给var01。
- ⑤ 如果var01大于100，或者var02大于100，就离开标签label2，即此时var01的值不再发生变化。
- ⑥ 如果var02的值大于50，就强制离开标签label2；即使var01的值没有大于100，var02的值没有大于100，也会离开标签label2内的LOOP循环。
- ⑦ 给变量var02不断加1并赋值给var02。
- ⑧ 分别打印出变量var01和变量var02的值。

- 调用过程

```

BEGIN
  DECLARE v1 INT;
  DECLARE v2 INT; ①
  CALL myproc(v1,v2); ②
END;

```

① 分别声明两个整数类型的变量v1, v2。

② 调用过程myproc。

输出结果为：

```

+-----+
| output |
+-----+
| var01:101 |
| var02:51  |
+-----+

```

5.6.9. CASE

- 例1：

```

CREATE OR REPLACE PROCEDURE test() ①
BEGIN
  DECLARE a INT;
  DECLARE b INT default 9; ②
  SET a = case b when 7 then 0 ③
    when 8 then 1 ④
    else 2 ⑤
  end case;
  put_line('case1:'||a); ⑥
  SET a = case when b > 9 then 10
    when b < 9 then 11
    else 12
  end; ⑦
  put_line('case2:'||a); ⑧
END;
/
BEGIN
  test(); ⑨
END;
/

```

① 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。

② 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。

③ 当b的值为7时，将0赋值给变量a。

④ 当b的值为8时，将1赋值给变量a。

⑤ 其它情况下，将2赋值给变量a。

⑥ 打印出此时变量a的值。

⑦ 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。

⑧ 打印出第二种情况下，a的值。

⑨ 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。

输出结果为：

+-----+
output
+-----+
case1:2
case2:12
+-----+

5.7. 存储过程

5.7.1. 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;
```

5.7.2. 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，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。

5.7.2.1. 不带参数

5.7.2.1.1. 例1:不带参数

- 创建存储过程proc_1

```
CREATE OR REPLACE PROCEDURE proc_1() ①
BEGIN
    PUT_LINE('hello world'); ②
END;
/
```

① 创建不带参数的存储过程proc_1。

② 打印出字符串hello world。

- 调用存储过程proc_1

```
CALL proc_1();
```

输出结果为：

output
hello world

5.7.2.2. IN 参数

5.7.2.2.1. 例2:IN参数

- 创建存储过程proc_in

```
CREATE OR REPLACE PROCEDURE proc_in(IN var1 STRING) ①
BEGIN
    PUT_LINE('var1:' || var1); ②
END;
/
```

① 创建带有参数的存储过程proc_in，形参为var1，其参数类型为IN，其数据类型为STRING。

② 打印出变量var1的值。

- 调用存储过程proc_in

```
CALL proc_in('hello world');
```

输出结果为：

output
var1:hello world

5.7.2.3. OUT 参数

5.7.2.3.1. 例3:OUT参数

- 创建存储过程proc_out

```

CREATE OR REPLACE PROCEDURE proc_out(OUT var2 STRING) ①
BEGIN
    SET var2 = 'hello world'; ②
END;
/

```

- ① 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。
 ② 给变量var2赋值。

- 调用存储过程proc_out

```

BEGIN
DECLARE a STRING; ①
proc_out(a); ②
PUT_LINE('a:'||a); ③
END;
/

```

- ① 声明变量a，数据类型为STRING。
 ② 调用过程proc_out，实参为a。
 ③ 打印出变量a的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| a:hello world   |
+-----+
```

5.7.2.3.2. 例4:OUT参数

- 创建存储过程proc_out_2

```

CREATE OR REPLACE PROCEDURE proc_out_2(OUT var2 INT) ①
BEGIN
    DECLARE a INT default 2; ②
    SET var2 = a; ③
END;
/

```

- ① 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。
 ② 声明一个整数类型的变量a，默认值为2。
 ③ 将变量a的值赋给变量var2。

- 调用存储过程proc_out_2

```

BEGIN
DECLARE b INT; ①
CALL proc_out_2(b); ②
PUT_LINE('var2:'||var2); ③
END;
/

```

- ① 声明变量b，数据类型为整数型。
 ② 调用过程proc_out_2，实参为b。
 ③ 打印出变量b的值。

输出结果为：

```
+-----+
| output |
+-----+
| b:2    |
+-----+
```

5.7.2.3.3. 例5: IN,OUT参数

- 创建存储过程proc_out_3

```
CREATE OR REPLACE PROCEDURE proc_out_3(IN var1 INT, OUT var2 INT) ①
BEGIN
    DECLARE a INT default 2; ②
    SET var2 = var1 + a;      ③
    PUT_LINE('var2:'||var2); ④
END;
/
```

① 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。

- ② 声明一个整数类型的变量，默认值为2。
 ③ 将变量var1和a的和赋给变量var2。
 ④ 打印出变量var的值。

- 调用过程proc_out_3

```
BEGIN
    DECLARE c INT DEFAULT 10; ①
    DECLARE d INT; ②
    proc_out_3(c,d); ③
END;
/
```

- ① 声明一个整数类型的变量c，默认值为10。
 ② 声明一个整数类型的变量d。
 ③ 调用过程proc_out_3，实参分别为c和d。

输出结果为：

```
+-----+
| output |
+-----+
| var2:12 |
+-----+
```

5.7.2.4. INOUT 参数

5.7.2.4.1. 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

```

CREATE OR REPLACE PROCEDURE proc_inout(INOUT var1 INT, INOUT var2 STRING) ①
BEGIN
    DECLARE a INT default 2; ②
    SET var1 = a; ③
    set var2 = 'hello world'; ④
END;
/

```

- ① 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。
- ② 声明一个整数类型的变量a，默认值为2。
- ③ 将变量a的值赋给变量var1。
- ④ 给变量var2赋值。

- 调用过程proc_inout

```

BEGIN
    DECLARE c INT; ①
    DECLARE d string; ②
    CALL proc_inout(c,d); ③
    PUT_LINE('var1:'||c); ④
    PUT_LINE('var2:'||d); ⑤
END;
/

```

- ① 声明一个整数类型的变量c。
- ② 声明一个字符串类型的变量d。
- ③ 调用过程proc_inout，实参分别为c，d。
- ④ 打印出变量c的值。
- ⑤ 打印出变量d的值。

输出结果为：

```

+-----+
|       output      |
+-----+
| var1:2          |
| var2:hello world |
+-----+

```

5.7.2.4.2. 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

```

CREATE OR REPLACE PROCEDURE proc_inout2(INOUT var1 INT, INOUT var2 STRING) ①
BEGIN
    PUT_LINE('var1:'||var1); ②
    PUT_LINE('var2:'||var2); ③
END;
/

```

- ① 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。
- ② 打印出变量var1的值。
- ③ 打印出变量var2的值。

- 调用过程proc_inout2

```
BEGIN
    DECLARE c INT; ①
    DECLARE d string; ②
    SET c = 100; ③
    SET d = 'hello world'; ④
    CALL proc_inout2(c,d); ⑤
END;
/
```

- ① 声明一个整数类型的变量c。
 ② 声明一个字符串类型的变量d。
 ③ 给变量c赋值。
 ④ 给变量d赋值。
 ⑤ 调用过程proc_inout2，实参分别为c和d。

输出结果为：

output
var1:100
var2:hello world

5.7.2.4.3. 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_inout

```
CREATE OR REPLACE PROCEDURE proc_inout3(INOUT var1 INT, INOUT var2 STRING) ①
BEGIN
    SET var1=100; ②
    PUT_LINE('var2:'||var2); ③
END;
/
```

- ① 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。
 ② 给变量var1赋值。
 ③ 打印出变量var1的值。

- 调用存储过程proc_inout3

```
BEGIN
    DECLARE c INT; ①
    DECLARE d string; ②
    SET d = 'hello world'; ③
    CALL proc_inout3(c,d); ④
    PUT_LINE('var1:'||c); ⑤
END;
/
```

- ① 声明一个整数类型的变量c。
 ② 声明一个字符串类型的变量d。

- ③ 给变量d赋值。
- ④ 调用过程proc_inout3，实参分别为c和d。
- ⑤ 打印出变量c的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| var1:100          |
| var2:hello world |
+-----+
```

5.7.2.5. 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。

5.7.2.5.1. 例9:不指定参数类型

- 创建存储过程proc_var

```
CREATE OR REPLACE PROCEDURE proc_var(var1 STRING) ①
BEGIN
    PUT_LINE('var1:'||var1); ②
END;
/
```

- ① 创建带有参数的存储过程proc_in，形参为var1，不指定参数类型，数据类型为STRING。
- ② 打印出变量var1的值。

- 调用存储过程proc_var

```
CALL proc_var('hello world');
```

输出结果为：

```
+-----+
|       output      |
+-----+
| var1:hello world |
+-----+
```

5.7.3. 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。

5.7.4. 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

```

CREATE OR REPLACE PROCEDURE overload_test (IN place_in STRING) ①
BEGIN
  DECLARE l_message STRING;
  SET l_message = 'Hello ' || place_in;
  PUT_LINE(l_message);
END;
/
CREATE OR REPLACE PROCEDURE overload_test(IN v_name INT) ②
BEGIN
  DECLARE l_message STRING;
  SET l_message = v_name||' is a good number';
  PUT_LINE(l_message);
END;
/
BEGIN
overload_test ('world'); ③
overload_test(888); ④
overload_test('999'); ⑤
END;
/

```

① 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。

② 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。

③ 过程内的实参为world，类型为字符串，则会执行第一个过程。

④ 过程内的实参为888，类型为整数类型，则会执行第二个过程。

⑤ 过程内的实参为999，类型为字符串，则会执行第一个过程。

输出结果为：

```
+-----+
|      output      |
+-----+
| Hello world    |
| 888 is a good number |
| Hello 999       |
+-----+
```

5.8. SQLPL函数

5.8.1. 创建函数

- 语法:

```
CREATE [OR REPLACE] FUNCTION <function_name> ([ {optional parameters} ]) [{optional function attributes}] RETURNS datatype
```

- 例如:

创建一个名为sqlpl_function的函数

```
CREATE OR REPLACE FUNCTION sqlpl_function(IN v_time STRING)
RETURNS STRING
```

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。

5.8.2. 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。

5.8.2.1. IN

- 例1:参数类型为IN时，可以向过程内写入数据

```
CREATE OR REPLACE FUNCTION test_function(IN v_price DOUBLE)
RETURNS STRING ①
BEGIN
    DECLARE v_time STRING;
    SELECT trans_time INTO v_time FROM transactions WHERE price=v_price; ②
    RETURN v_time; ③
END;
/
BEGIN
    DECLARE V1 STRING;
    SET V1 = test_function(12.13); ④
    PUT_LINE('V1:'||V1);
END;
/
```

- ① 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。

- ② 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。
- ③ 函数test_function的返回值为v_time。
- ④ 将实参为12.13的函数返回值赋值给变量V1。

输出结果为：

```
+-----+
|       output      |
+-----+
| V1:20140105100520 |
+-----+
```

5.8.2.2. OUT

- 例2：参数类型为OUT类型时，可以从过程内读出数据

```
CREATE OR REPLACE FUNCTION test_function(OUT var1 DOUBLE)
RETURNS DOUBLE ①
BEGIN
    DECLARE a DOUBLE default 11.11; ②
    SET var1 = a; ③
    RETURN var1; ④
END;
/
BEGIN
    DECLARE var2 DOUBLE;
    DECLARE V1 DOUBLE;
    SET var2 = test_function(V1); ⑤
    PUT_LINE('var2:'||var2);
END;
/
```

- ① 创建函数test_function，函数返回值的数据类型为DOUBLE，参数var1的类型为OUT，其数据类型为DOUBLE。
- ② 声明一个DOUBLE类型的变量a，并赋初值为11.11。
- ③ 将变量a的值赋值给参数var1。
- ④ 函数test_function的返回值为var1。
- ⑤ 将实参为V1的函数test_function的返回值赋值给变量var2。

输出结果为：

```
+-----+
|       output      |
+-----+
| var2:11.11       |
+-----+
```

5.8.2.3. INOUT

- 例3：INOUT参数类型可以向过程中写入数据

```

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; ①
    SET V1 = test_function(v_price); ②
    PUT_LINE('V1:'||V1);
END;
/

```

① 声明一个DOUBLE类型的变量v_price，并赋初值为12.13。

② 将参数为v_price，参数的值为12.13的函数test_function的返回值赋值给变量V1。

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

```

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。

5.8.3. 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。

5.8.4. 调用函数

5.8.4.1. 单独调用函数

- 创建SQL PL函数sqlpl_function

```
CREATE OR REPLACE FUNCTION sqlpl_function(IN v_price DOUBLE) ①
RETURNS STRING ②
BEGIN
DECLARE v_time STRING; ③
SELECT trans_time INTO v_time FROM transactions WHERE price=v_price; ④
RETURN v_time; ⑤
END;
/
```

- ① 创建函数sqlpl_function，参数v_price的类型为IN，数据类型为DOUBLE。
- ② 函数返回值的数据类型为字符串。
- ③ 声明一个字符串类型的变量v_time。
- ④ 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。
- ⑤ 返回变量v_time的值。

- 单独调用函数sqlpl_function

```

BEGIN
  DECLARE v1 STRING;
  SET v1 = sqlpl_function(11.11); ①
  PUT_LINE('V1:'||v1);
END;
/

```

- ① 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。

输出函数为：

```

+-----+
|       output      |
+-----+
| V1:20140205140521 |
+-----+

```

5.8.4.2. 在存储过程内调用函数

- 创建SQL PL函数sqlpl_function

```

CREATE OR REPLACE FUNCTION sqlpl_function(IN v_price DOUBLE) ①
RETURNS STRING ②
BEGIN
  DECLARE v_time STRING; ③
  SELECT trans_time INTO v_time FROM transactions WHERE price=v_price; ④
  RETURN v_time; ⑤
END;
/

```

- ① 创建函数sqlpl_function，参数v_price的类型为IN，数据类型为DOUBLE。
 ② 函数返回值的数据类型为字符串。
 ③ 声明一个字符串类型的变量v_time。
 ④ 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。
 ⑤ 返回变量v_time的值。

- 在存储过程test_procedure内调用函数sqlpl_function

```

CREATE OR REPLACE PROCEDURE test_procedure(IN v_amount DOUBLE) ①
BEGIN
  DECLARE v_id STRING;
  DECLARE v_num STRING; ②
  SELECT trans_id INTO v_id FROM transactions WHERE amount=v_amount; ③
  SELECT acc_num INTO v_num FROM transactions WHERE trans_time=sqlpl_function(11.11); ④
  PUT_LINE('the id is:'||v_id);
  PUT_LINE('the accnum is:'||v_num);
END;
/
BEGIN
  test_procedure(900.0); ⑤
END;

```

- ① 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。
 ② 分别声明两个字符串类型的变量v_id, v_num。
 ③ 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。
 ④ 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。
 ⑤ 调用过程test_procedure，实参的值为900。

输出结果为：

```
+-----+
|       output      |
+-----+
| the id is:900192386 |
| the accnum is:3912384 |
+-----+
```

5.8.4.3. SQL语句中调用函数

- 创建SQL PL函数sqlpl_function

```
CREATE OR REPLACE FUNCTION sqlpl_function(IN v_price DOUBLE) ①
RETURNS DOUBLE ②
BEGIN
    RETURN v_price; ③
END;
/
```

① 创建函数sqlpl_function，参数v_price的类型为IN，数据类型为DOUBLE。

② 函数返回值的数据类型为字符串。

③ 返回变量v_time的值。

- 标准SQL语句中调用函数

```
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  |
+-----+-----+-----+-----+-----+-----+-----+
```

5.8.4.4. SQLPL函数中对其他函数的调用

- 创建函数test1

```
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

```
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

```
BEGIN
  DECLARE V1 DOUBLE;
  SET V1 = test2(1.11);
  PUT_LINE('V1:'||V1);
END;
/
```

输出结果为：

output
V1:12.21999999999999



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。

5.8.5. 标准SQL调用SQLPL函数必须满足的条件

5.8.5.1. 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

```
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

```
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。

5.8.5.2. 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。

5.8.5.2.1. 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。

```
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

```
SELECT sqlpl_function(password) FROM user_info;
```

输出结果为：

```
+-----+
| _c0 |
+-----+
| 115591 |
| 205239 |
| 531547 |
| 841242 |
| 986634 |
| 737297 |
| 600709 |
| 990590 |
| 015859 |
| 783438 |
+-----+
10 rows selected (3.188 seconds)
```

5.8.5.3. SQLPL函数中不能有标准SQL语句

- 创建SQL PL函数sqlpl_function

```
CREATE OR REPLACE FUNCTION sqlpl_function(IN v_price DOUBLE) ①
RETURNS STRING ②
BEGIN
  DECLARE v_time STRING; ③
  SELECT trans_time INTO v_time FROM transactions WHERE price=v_price; ④
  RETURN v_time; ⑤
END;
/
```

- ① 创建函数sqlpl_function，参数v_price的类型为IN，数据类型为DOUBLE。
- ② 函数返回值的数据类型为字符串。
- ③ 声明一个字符串类型的变量v_time。
- ④ 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。
- ⑤ 返回变量v_time的值。

- 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：

```
Error: Error while processing statement: FAILED: Error in semantic analysis: Line 1:44 Wrong arguments '12.13': org.apache.hadoop.hive.ql.metadata.HiveException: PLSQL function is running in a non-driver environment (usually in SQL statement), which doesn't allow nested SQL statement. (state=,code=10)
```

5.8.6. 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，来决定应该执行的函数。
- 例：重载函数overload_test_func

```
CREATE OR REPLACE FUNCTION overload_test_func( IN place_in string)
RETURNS STRING ①
BEGIN
    RETURN 'Hello ' || place_in; ②
END;
/
CREATE OR REPLACE FUNCTION overload_test_func( IN place_in INT)
RETURNS STRING ③
BEGIN
    RETURN place_in * place_in; ④
END;
/
BEGIN
    DECLARE V1 STRING;
    DECLARE V2 STRING;
    DECLARE V3 STRING;
    SET V1 =overload_test_func('world'); ⑤
    SET V2 =overload_test_func(11); ⑥
    SET V3 =overload_test_func('111'); ⑦
    PUT_LINE(V1);
    PUT_LINE(V2);
    PUT_LINE(V3);
END;
/
```

- ① 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。
- ② 第一个函数返回值为，在实参变量的前面加上字符'hello'。
- ③ 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型。
- ④ 第二个函数的返回值为，实参的平方。
- ⑤ 函数内的实参为world，类型为字符串，则会执行第一个函数。
- ⑥ 函数内的实参为11，类型为整数类型，则会执行第二个函数。
- ⑦ 函数内的实参为111，类型为字符串，则会执行第一个函数。

输出结果：

output
Hello world
121
Hello 111

5.9. VARRAY

5.9.1. 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。

5.9.2. 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。

5.9.2.1. SET

- 例1：

```
BEGIN
  DECLARE TYPE age_array AS INT ARRAY[5]; ①
  DECLARE v_age_array age_array; ②
  SET v_age_array = ARRAY[18,19,20,21,22,23]; ③
END;
/
```

① 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。

② 声明一个age_array类型的变量v_age_array。

③ 给变量v_age_array中5个整数类型的变量同时赋值。

- 例2:

```
BEGIN
DECLARE TYPE age_array AS INT ARRAY[5]; ①
DECLARE v_age_array age_array; ②
SET v_age_array(1) = 30; ③
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;
/
```

① 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。

② 声明一个age_array类型的变量v_age_array。

③ 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。

5.9.2.2. VALUES ... INTO

- 例2:

```
BEGIN
DECLARE TYPE age_array AS INT ARRAY[5]; ①
DECLARE v_age_array age_array; ②
VALUES 20 INTO v_age_array(1); ③
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;
/
```

① 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。

② 声明一个age_array类型的变量v_age_array。

③ 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。

5.9.3. 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

5.9.3.1. ARRAY_FIRST

- 例1:

```
BEGIN
DECLARE TYPE age_array AS INT ARRAY[5]; ①
DECLARE v_age_array age_array; ②
DECLARE v_age INT; ③
SET v_age_array = ARRAY[18,19,20,21,22]; ④
SET v_age = v_age_array.ARRAY_FIRST; ⑤
PUT_LINE('the first one:'||v_age_array(v_age)); ⑥
END;
/
```

- ① 定义一个ARRAY类型age_array，可存放5个整数类型的数据。
 ② 声明一个age_array类型的变量v_age_array。
 ③ 声明一个整数类型的变量v_age。
 ④ 给变量v_age_array赋值，数组中的顺序依次为18, 19, 20, 21, 22。
 ⑤ 将变量v_age_array中的第一个元素的下标赋值给变量v_age。
 ⑥ 打印出变量v_age_array中第一个元素的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| the first one:18 |
+-----+
```

5.9.3.2. ARRAY_LAST

- 例1:

```
BEGIN
DECLARE TYPE age_array AS INT ARRAY[5]; ①
DECLARE v_age_array age_array; ②
DECLARE v_age INT;
SET v_age_array = ARRAY[18,19,20,21,22]; ③
SET v_age = v_age_array.ARRAY_LAST; ④
PUT_LINE('the last one:'||v_age_array(v_age));
END;
/
```

- 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。

输出结果为：

```
+-----+
|       output      |
+-----+
| the last one:22 |
+-----+
```

5.9.3.3. ARRAY_NEXT

- 例1:

```

BEGIN
DECLARE TYPE age_array AS INT ARRAY[5]; ①
DECLARE v_age_array age_array; ②
DECLARE v_age INT;
SET v_age_array = ARRAY[18,19,20,21,22]; ③
SET v_age = ARRAY_NEXT(v_age_array,3); ④
PUT_LINE('the next one:'||v_age_array(v_age));
END;
/

```

- ① 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。

输出结果为：

```

+-----+
|       output      |
+-----+
| the next one:21 |
+-----+

```

5.9.3.4. ARRAY_PRIOR

- 例1：

```

BEGIN
DECLARE TYPE age_array AS INT ARRAY[5]; ①
DECLARE v_age_array age_array; ②
DECLARE v_age INT;
SET v_age_array = ARRAY[18,19,20,21,22]; ③
SET v_age = ARRAY_PRIOR(v_age_array,3); ④
PUT_LINE('the prior one:'||v_age_array(v_age));
END;
/

```

- ① 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。

输出结果为：

```

+-----+
|       output      |
+-----+
| the prior one:19 |
+-----+

```

5.9.3.5. ARRAY_DELETE

- 例1：

```

BEGIN
DECLARE TYPE age_array AS INT ARRAY[5];
DECLARE v_age_array age_array;
DECLARE v_age age_array; ①
DECLARE v_sum INT; ②
SET v_age_array = ARRAY[18,19,20,21,22];
SET v_age = ARRAY_DELETE(v_age_array); ③
SET v_sum = v_age.CARDINALITY; ④
PUT_LINE('变量v_age的个数:'||v_sum); ⑤
END;
/

```

- ① 声明一个age_array类型的变量v_age。

- ② 声明一个整数类型的变量v_sum。

- ③ 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。
 ④ 将变量v_age_array中的元素个数，赋值给变量v_sum。
 ⑤ 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。

输出结果为：

```
+-----+
|       output      |
+-----+
| 变量v_age的个数:0|
+-----+
```

- 例2:删除数组中的某一个元素

```
BEGIN
DECLARE TYPE age_array AS INT ARRAY[5];
DECLARE v_age_array age_array;
DECLARE v_age age_array;
DECLARE v_sum INT;
SET v_age_array = ARRAY[18,19,20,21,22];
SET v_age = ARRAY_DELETE(v_age_array,3);
SET v_sum = v_age.CARDINALITY;
PUT_LINE('变量v_age的个数:' || v_sum);
END;
/
```

5.9.3.6. CARDINALITY

- 例1：

```
BEGIN
DECLARE TYPE age_array AS INT ARRAY[5]; ①
DECLARE v_age_array age_array; ②
DECLARE v_age INT;
SET v_age_array = ARRAY[18,19,20,21,22,23]; ③
SET v_age = v_age_array.CARDINALITY; ④
PUT_LINE('数组个数: ' || v_age);
END;
/
```

- ① 将变量v_age_array中的元素个数赋值给变量v_age。

输出结果为：

```
+-----+
|       output      |
+-----+
| 数组个数:6|
+-----+
```

5.10. 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。

5.10.1. 语法

- 行数据类型的定义

```
CREATE 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。

```
CREATE 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
```

5.10.2. 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。

5.10.2.1. 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。

```
BEGIN
DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE); ①
DECLARE transactionrow transactiontype; ②
SET transactionrow = ('0700735','20140916105811',11.11); ③
PUT_LINE('acc_num:'||transactionrow.v_acc_num); ④
PUT_LINE('time:'||transactionrow.v_trans_time); ⑤
PUT_LINE('price:'||transactionrow.v_price); ⑥
END;
/
```

① 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。

② 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。

③ 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。

④ 打印出行变量transactionrow中的字段transactionrow.v_acc_num的值。

⑤ 打印出行变量transactionrow中的字段transactionrow.v_trans_time的值。

⑥ 打印出行变量transactionrow中的字段transactionrow.v_price的值。

输出结果为：

```
+-----+
|      output      |
+-----+
| acc_num:0700735 |
| time:20140916105811 |
| price:11.11    |
+-----+
```

5.10.2.2. 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。

```
BEGIN
DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE); ①
DECLARE transactionrow transactiontype; ②
VALUES ('0700735','20140916105811',11.11) INTO transactionrow ; ③
PUT_LINE('acc_num:' || transactionrow.v_acc_num); ④
PUT_LINE('time:' || transactionrow.v_trans_time); ⑤
PUT_LINE('price:' || transactionrow.v_price); ⑥
END;
/
```

① 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。

② 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。

③ 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。

④ 打印出行变量transactionrow中的字段transactionrow.v_acc_num的值。

⑤ 打印出行变量transactionrow中的字段transactionrow.v_trans_time的值。

⑥ 打印出行变量transactionrow中的字段transactionrow.v_price的值。

输出结果为：

```
+-----+
|      output      |
+-----+
| acc_num:0700735 |
| time:20140916105811 |
| price:11.11    |
+-----+
```

5.10.2.3. 例三

- 例3: 分别给每一个行变量的字段赋值

```

BEGIN
DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE); ①
DECLARE transactionrow transactiontype; ②
SET transactionrow.v_acc_num = '6513065'; ③
SET transactionrow.v_trans_time = '20140506133109'; ④
SET transactionrow.v_price = '12.13'; ⑤
PUT_LINE('acc_num'||transactionrow.v_acc_num); ⑥
PUT_LINE('time'||transactionrow.v_trans_time); ⑦
PUT_LINE('price'||transactionrow.v_price); ⑧
END;
/

```

① 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。

② 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。

③ 给行变量transactionrow的字段transactionrow.v_acc_num赋值为6513065。

④ 给行变量transactionrow的字段transactionrow.v_trans_time赋值为20140506133109。

⑤ 给行变量transactionrow的字段transactionrow.v_price赋值为12.13。

⑥ 打印出行变量transactionrow的字段transactionrow.v_acc_num的值。

⑦ 打印出行变量transactionrow的字段transactionrow.v_trans_time的值。

⑧ 打印出行变量transactionrow的字段transactionrow.v_price的值。

输出结果为：

```

+-----+
|       output      |
+-----+
| acc_num:6513065  |
| time:20140506133109 |
| price:12.13        |
+-----+

```

5.10.2.4. 例四

- 例4: 将一个查询结果集的单行值赋值给一个行变量

```

BEGIN
DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE); ①
DECLARE transactionrow transactiontype; ②
SET transactionrow = (SELECT acc_num,trans_time,price FROM transactions WHERE
trans_id=648230055); ③
PUT_LINE('acc_num'||transactionrow.v_acc_num); ④
PUT_LINE('time'||transactionrow.v_trans_time); ⑤
PUT_LINE('price'||transactionrow.v_price); ⑥
END;
/

```

① 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。

② 声明一个行变量transactionrow，其数据类型为transactiontype，该变量包含transactiontype类型的

的三个字段。

- ③ 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。
- ④ 打印出行变量transactionrow中的字段transactionrow.v_acc_num的值。
- ⑤ 打印出行变量transactionrow中的字段transactionrow.v_trans_time的值。
- ⑥ 打印出行变量transactionrow中的字段transactionrow.v_price的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| acc_num:0700735  |
| time:20140315111111 |
| price:22.66        |
+-----+
```

5.10.2.5. 例五

- 例5:通过SELECT INTO 语句将行值赋值给行变量

```
BEGIN
DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE); ①
DECLARE transactionrow transactiontype; ②
SELECT acc_num,trans_time,price INTO transactionrow FROM transactions WHERE
stock_id='GL2547626'; ③
    PUT_LINE('acc_num:'||transactionrow.v_acc_num); ④
    PUT_LINE('time:'||transactionrow.v_trans_time); ⑤
    PUT_LINE('price:'||transactionrow.v_price); ⑥
END;
/
```

- ① 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。
- ② 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。
- ③ 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。
- ④ 打印出行变量transactionrow中的字段transactionrow.v_acc_num的值。
- ⑤ 打印出行变量transactionrow中的字段transactionrow.v_trans_time的值。
- ⑥ 打印出行变量transactionrow中的字段transactionrow.v_price的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| acc_num:5224133  |
| time:20140801110003 |
| price:6.36        |
+-----+
```

5.10.3. 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

```
BEGIN
DECLARE TYPE transaction_row as row anchor to row of transactions; ①
DECLARE transactionrow transaction_row; ②
SELECT * INTO transactionrow FROM transactions WHERE stock_id='GL2547626'; ③
  PUT_LINE('acc_num:'||transactionrow.acc_num);
  PUT_LINE('time:'||transactionrow.trans_time);
  PUT_LINE('price:'||transactionrow.price); ④
END;
/
```

- ① 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。
- ② 声明一个transaction_row类型的变量transactionrow。
- ③ 查询表transactions中股票账号为GL2547626的全部信息，并放进变量transactionrow里。
- ④ 依次打印出变量transactionrow中的账号，交易时间和价格。

输出结果为：

output	
acc_num:	5224133
time:	20140801110003
price:	6.36

5.11. 游标

5.11.1. 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;
```

5.11.2. 基本用法

- 例1:

```
BEGIN
  DECLARE c1 CURSOR FOR SELECT trans_time FROM transactions WHERE price=12.13; ①
  DECLARE v_time STRING; ②
  OPEN c1; ③
  FETCH c1 INTO v_time; ④
  CLOSE c1; ⑤
  PUT_LINE('the time is:'||v_time); ⑥
END;
/
```

① 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。

② 声明一个名为v_time的变量，数据类型为字符串。

③ 打开游标。

④ 将游标查询到的结果放进变量v_time里。

⑤ 关闭游标。

⑥ 打印出变量v_time的值。

输出结果为:

```
+-----+
|       output      |
+-----+
| the time is:20140105100520 |
+-----+
```

5.11.3. 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。

5.11.3.1. 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>;
```

5.11.3.2. 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

```
CREATE OR REPLACE PROCEDURE pro_1() ①
LANGUAGE SQL
DYNAMIC RESULT SETS 2 ②
BEGIN
    DECLARE c1 CURSOR WITH RETURN FOR
        SELECT acc_num FROM transactions WHERE price=12.13; ③
    OPEN c1; ④
    BEGIN
        DECLARE c2 CURSOR WITH RETURN FOR
            SELECT trans_time FROM transactions WHERE price=12.13; ⑤
        OPEN c2;
    END;
END;
/
```

- ① 创建一个名为pro_1且不带参数的存储过程。
 ② 存储过程pro_1将返回两个结果集。
 ③ 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。
 ④ 打开游标c1。
 ⑤ 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。

- 调用pro_1()过程，在过程内分配两个带有返回值的游标

```
BEGIN
DECLARE rsl1, rsl2 result_set_locator varying; ①
DECLARE v1 STRING ;
CALL pro_1(); ②

ASSOCIATE RESULT SET LOCATOR (rsl1, rsl2) WITH PROCEDURE pro_1; ③
ALLOCATE c1 CURSOR FOR result set rsl1;
ALLOCATE c2 CURSOR FOR result set rsl2; ④

FETCH c1 INTO v1; ⑤
CLOSE c1; ⑥
PUT_LINE('the acc_num is:'||v1); ⑦

FETCH c2 INTO v1;
CLOSE c2;
PUT_LINE('the time is:'||v1);
END;
/
```

- ① 声明两个结果集rsl1, rsl2。
 ② 调用过程pro_1。
 ③ 结果集rsl1, rsl2分别用来存放过程pro_1返回的结果。
 ④ 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。
 ⑤ 将游标c1查询到的结果放进变量v1里。
 ⑥ 关闭游标c1。
 ⑦ 打印出变量v1的值，即所查询到的表transactions中的账号。

输出结果为：

```
+-----+
|          output           |
+-----+
| the acc_num is:6513065   |
| the time is:20140105100520|
+-----+
2 rows selected (6.618 seconds)
```

5.12. 与SQL的交互

5.12.1. 将以SELECT返回值赋值给变量

- 例1:将查询结果赋值给变量

```
BEGIN
DECLARE V1 STRING;
DECLARE V2 DOUBLE;
SET V1 = (SELECT trans_time FROM transactions WHERE price = 11.11); ①
SET V2 = (SELECT price FROM transactions WHERE trans_id=162742112); ②
PUT_LINE('V1:'||V1);
PUT_LINE('V2:'||V2);
END;
/
```

① 查询transactions表中价格为11.11的交易时间，并赋值给变量V1。

② 查询transactions表中交易号为162742112的价格，并赋值给变量V2。

输出结果为：

```
+-----+
|          output           |
+-----+
| V1:20140205140521       |
| V2:12.21                  |
+-----+
```

5.12.2. SELECT INTO

- 例1:通过SELECT INTO语句将行值赋值给行变量

```
BEGIN
DECLARE TYPE transactiontype AS ROW (v_acc_num STRING, v_trans_time STRING, v_price DOUBLE); ①
DECLARE transactionrow transactiontype; ②
SELECT acc_num,trans_time,price INTO transactionrow FROM transactions WHERE
stock_id='GL2547626'; ③
PUT_LINE('acc_num:'||transactionrow.v_acc_num); ④
PUT_LINE('time:'||transactionrow.v_trans_time); ⑤
PUT_LINE('price:'||transactionrow.v_price); ⑥
END;
/
```

① 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。

② 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。

- ③ 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。
- ④ 打印出行变量transactionrow中的字段transactionrow.v_acc_num的值。
- ⑤ 打印出行变量transactionrow中的字段transactionrow.v_trans_time的值。
- ⑥ 打印出行变量transactionrow中的字段transactionrow.v_price的值。

输出结果为：

```
+-----+
|      output      |
+-----+
| acc_num:5224133 |
| time:20140801110003 |
| price:6.36       |
+-----+
```

5.12.3. 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1：在存储过程内执行UPDATE，INSERT，DELETE，MERGE等DML语句

```
CREATE OR REPLACE PROCEDURE sql_procedure(IN v_name STRING, IN v_gpa DOUBLE) ①
BEGIN
    INSERT INTO zara VALUES('Amy', 23, 4.5); ②
    UPDATE zara SET age= 18 WHERE name = v_name; ③
    DELETE FROM zara WHERE gpa = v_gpa; ④
    MERGE INTO za z using zara r on (z.name=r.name) ⑤
    WHEN MATCHED THEN UPDATE SET z.age=40 ⑥
    WHEN NOT MATCHED THEN INSERT (name,age) VALUES(r.name,r.age); ⑦
END;
/
CALL sql_procedure('zhangsan', 4.5); ⑧
```

- ① 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。
- ② 往表zara中插入一条数据，姓名为Amy，年龄为23，绩点为4.5。
- ③ 将表zara中姓名和参数v_name相等的人的年龄更新为18。
- ④ 删除表zara中绩点与参数v_gpa的值相等的记录。
- ⑤ 对表za进行merge操作，条件为两个表中的姓名相同。
- ⑥ 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。
- ⑦ 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。
- ⑧ 调用过程sql_procedure，实参的值分别为zhangsan，4.5。

5.12.4. 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1:在函数内执行UPDATE, INSERT, DELETE, MERGE等DML语句

```

CREATE OR REPLACE FUNCTION sql_function(IN v_name STRING) ①
RETURNS DOUBLE ②
BEGIN
    DECLARE v_gpa DOUBLE;
    SELECT gpa INTO v_gpa FROM zara WHERE name=v_name; ③
    RETURN v_gpa; ④
END;
/
BEGIN
    DELETE FROM zara WHERE gpa= sql_function('lily'); ⑤
END;
/

```

① 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。

② sql_function函数的返回值的数据类型为DOUBLE。

③ 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。

④ 函数返回值为v_gpa的值。

⑤ 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。



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](#)。

5.12.5. 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1:游标的基本查询示例

```

BEGIN
    DECLARE c1 CURSOR FOR SELECT trans_time FROM transactions WHERE price=12.13; ①
    DECLARE v_time STRING; ②
    OPEN c1; ③
    FETCH c1 INTO v_time; ④
    CLOSE c1; ⑤
    PUT_LINE('the time is:'||v_time); ⑥
END;
/

```

① 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。

② 声明一个名为v_time的变量，数据类型为字符串。

③ 打开游标。

④ 将游标查询到的结果放进变量v_time里。

⑤ 关闭游标。

⑥ 打印出变量v_time的值。

输出结果为：

```
+-----+  
|       output      |  
+-----+  
| the time is:20140105100520 |  
+-----+
```

5.12.6. 动态SQL

5.12.6.1. 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。

5.12.6.1.1. 处理DDL语句

CREATE

- 例1：使用EXECUTE IMMEDIATE创建表sqlpl_create

```
BEGIN
DECLARE create_table STRING;
    SET create_table ='create table ' ||'sqlpl_create' ||'(sno string , sname string )'; ①
EXECUTE IMMEDIATE create_table;
END;
/
```

① 注意字符串与连接符之间的空格，否则Inceptor可能会报错。

ALTER

- 查看sqlpl_create表的结构

```
describe sqlpl_create;
+-----+-----+-----+-----+-----+
| col_name | data_type | comment | notnull_constraint | unique_constraint |
+-----+-----+-----+-----+-----+
| sno      | string    |         |                 |                 |
| sname    | string    |         |                 |                 |
+-----+-----+-----+-----+-----+
```

- 例2：使用EXECUTE IMMEDIATE修改表sqlpl_create中列sno的列名和数据类型

```
BEGIN
DECLARE alter_table STRING;
    SET alter_table ='alter table sqlpl_create CHANGE sno sid INT';
EXECUTE IMMEDIATE alter_table;
END;
/
```

- 再次查看sqlpl_create表的结构

```
describe sqlpl_create;
+-----+-----+-----+-----+-----+
| col_name | data_type | comment | notnull_constraint | unique_constraint |
+-----+-----+-----+-----+-----+
| sid      | int       |          |                  |                  |
| sname    | string    |          |                  |                  |
+-----+-----+-----+-----+-----+
```

DROP

- 以同样的方法创建表test_create, 现在我们来用EXECUTE IMMEDIATE删除表test_create
- 例3: 使用EXECUTE IMMEDIATE删除表test_create

```
BEGIN
DECLARE drop_table STRING;
SET 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。

5.12.6.1.2. 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

- 例1: 创建过程grant_priv

```
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;
/
```

- 调用例1的过程

```
BEGIN
grant_priv('create', 'user1');
END;
/
```

- 查看user1被赋予的权限

```
0: jdbc:hive2://localhost:10000/default> show grant user user1 on all;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| database | table | partition | column | principal_name | principal_type | privilege | grant_option | grant_time | grantor |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|          |        |          |        | user1           | USER          | CREATE     | false      | 1452264935000 | hive       |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row selected (0.14 seconds)
```

REVOKE

- 案例1: 创建过程revoke_priv

```

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;
/

```

- 调用例1的过程

```

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。

```

0: jdbc:hive2://localhost:10000/default> show grant user user1 on all;
+-----+-----+-----+-----+-----+-----+-----+-----+
| database | table | partition | column | principal_name | principal_type | privilege | grant_option | grant_time | grantor |
+-----+-----+-----+-----+-----+-----+-----+-----+
|         |       |          |        |             |             |           |           |           |           |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

5.12.6.1.3. 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1:EXECUTE IMMEDIATE后接字符串类型的变量

```

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2:EXECUTE IMMEDIATE后接字符串常量

```

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3:EXECUTE IMMEDIATE执行含有占位符的SELECT查询语句

```
BEGIN
  EXECUTE IMMEDIATE 'select price from transactions where acc_num=?' using 6513065; ①
END;
/
```

① 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

输出结果为：

```
+-----+
| output |
+-----+
| 12.13 |
| 6.12  |
| 9.81  |
| 7.49  |
| 7.02  |
| 18.38 |
+-----+
```

5.12.6.1.4. 处理不含占位符的DML语句

INSERT

- 以同样的方法创建表orctest1，字段名分别为age, gpa, 数据类型分别为int, double
- 例1:使用EXECUTE IMMEDIATE往表中插入数据

```
BEGIN
  DECLARE insert_table STRING;
  SET insert_table = 'INSERT INTO orctest1 values(20,4.0)';
  EXECUTE IMMEDIATE insert_table;
END;
/
```

- 案例1:查看表orctest1的数据，可以发现新增了一条数据

```
select * from orctest1;
+-----+-----+
| age | gpa |
+-----+-----+
| 22  | 4.3 |
| 20  | 4.0 |
| 21  | 3.9 |
+-----+-----+
```

UPDATE

- 例2:使用EXECUTE IMMEDIATE更新表中数据

```

BEGIN
  DECLARE update_table STRING;
    SET update_table =' UPDATE orctest2 SET gpa=4.4 WHERE age=20 ';
  EXECUTE IMMEDIATE update_table;
END;
/

```

- 案例2:查看表orctest1的数据,可以发现年龄为20的gpa已经发生了变化

```

select * from orctest1;
+-----+-----+
| age | gpa |
+-----+-----+
| 22  | 4.3 |
| 20  | 4.4 |
| 21  | 3.9 |
+-----+-----+

```

DELETE

- 例3:使用EXECUTE IMMEDIATE表中数据

```

BEGIN
  DECLARE delete_table STRING;
    SET delete_table ='DELETE FROM orctest1 WHERE age=20';
  EXECUTE IMMEDIATE delete_table;
END;
/

```

- 例3:查看表orctest1的数据,可以发现年龄为20的记录已经被成功删除

```

select * from orctest1;
+-----+-----+
| age | gpa |
+-----+-----+
| 22  | 4.3 |
| 21  | 3.9 |
+-----+-----+

```

CREATE+INSERT

- 例4:使用EXECUTE IMMEDIATE创建表格并插入数据

```

BEGIN
  DECLARE create_table string;
  DECLARE insert_table string;
    SET create_table ='create table orctest2 (age int, gpa double)
      CLUSTERED BY (age) INTO 2 BUCKETS
      STORED AS ORC
      TBLPROPERTIES ("transactional"="true");
  EXECUTE IMMEDIATE create_table;
    SET insert_table ='insert into orctest2 values (22,4.3)';
  EXECUTE IMMEDIATE insert_table;
END;
/

```

查看表orctest2及表中的数据

```

describe orctest2;
+-----+-----+-----+-----+-----+
| col_name | data_type | comment | notnull_constraint | unique_c |
+-----+-----+-----+-----+-----+
| age      | int       | from deserializer |           |           |
| gpa      | double    | from deserializer |           |           |
+-----+-----+-----+-----+-----+

```

可以看到在成功创建orctest2的同时，也成功插入了一条数据

```
select * from orctest2;
+-----+-----+
| age | gpa |
+-----+-----+
| 22  | 4.3 |
+-----+-----+
```

CREATE+INSERT+UPDATE+DELETE+INSERT

- 例5: 使用EXECUTE IMMEDIATE处理一系列DML语句

```
BEGIN
DECLARE create_table STRING;
DECLARE insert_table STRING;
DECLARE update_table STRING;
DECLARE delete_table STRING;
SET create_table ='create table orctest3 (pid int, price double)
CLUSTERED BY (pid) INTO 2 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");' ①
EXECUTE IMMEDIATE create_table;
SET insert_table ='insert into orctest3 values (111,12.13)'; ②
EXECUTE IMMEDIATE insert_table;
SET update_table ='update orctest3 set price=14.15 where pid=111'; ③
EXECUTE IMMEDIATE update_table;
SET delete_table ='delete from orctest3 where pid=111'; ④
EXECUTE IMMEDIATE delete_table;
SET insert_table ='insert into orctest3 values (112,11.11)'; ⑤
EXECUTE IMMEDIATE insert_table;
END;
/
```

① 创建ORC表orctest3，字段名分为pid, price，数据类型分为INT, DOUBLE

② 往表orctest3中插入数据 (111, 12. 13)

③ 将pid为111的价格更新为14. 15

④ 删除表orctest3中pid为111的记录

⑤ 往表orctest3中插入数据 (112, 11. 11)

输出结果：

```
+-----+
| output |
+-----+
+-----+
```

查看表orctest3的数据

```
select * from orctest3;
+-----+-----+
| pid | price |
+-----+-----+
| 112 | 11.11 |
+-----+-----+
```

5.12.6.1.5. 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

- 例1: 使用EXECUTE IMMEDIATE往表中插入数据

```
BEGIN
DECLARE insert_table STRING;
SET insert_table = 'INSERT INTO orctest2 values(?,?)';
EXECUTE IMMEDIATE insert_table USING 22,4.7;
EXECUTE IMMEDIATE insert_table USING 24,4.8;
END;
/
```

此处案例执行的结果是往表orctest2插入两条记录，分别为(22, 4.7), (24, 4.8)。

UPDATE

- 例2: 使用EXECUTE IMMEDIATE更新表中数据

```
BEGIN
DECLARE update_table STRING;
SET update_table =' UPDATE orctest2 SET gpa=4.4 WHERE age=? ';
EXECUTE IMMEDIATE update_table USING 22;
EXECUTE IMMEDIATE update_table USING 24;
END;
/
```

此处案例的执行结果是将表orctest2年龄为22和24的记录的gpa更新为4.4。

DELETE

- 例3: 使用EXECUTE IMMEDIATE删除表中数据

```
BEGIN
DECLARE delete_table STRING;
SET delete_table ='DELETE FROM orctest2 WHERE age=?';
EXECUTE IMMEDIATE delete_table using 20;
EXECUTE IMMEDIATE delete_table using 24;
END;
/
```

此处案例的执行结果是删除表orctest2中年龄为20和24的记录。

5.12.6.2. 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，必须是英文键盘下的问号。
 - PREPARE … EXECUTE只能用来处理SELECT查询及DML语句。

5.12.6.2.1. SELECT



要查看SELECT查询的结果需要设置plsql.show.sqlresults的属性为Ttrue，更多事项可参见[注意事项章节](#)。

- 例1:

```

BEGIN
  DECLARE stmt STATEMENT; ①
  PREPARE stmt FROM 'begin select count(*) from ' || 'transactions' || ' where acc_num = ? ;
end'; ②
  EXECUTE stmt using 6513065; ③
  EXECUTE stmt using 0700735; ④
END;
/

```

- ① 声明一个STATEMENT类型的变量stmt。
- ② 使用关键字PREPARE，将SQL语句'查询transactions表中账号名所对应的行数'放进变量stmt里。
- ③ 使用关键字执行变量stmt内的SQL语句，查询账号为6513065的行数。
- ④ 使用关键字执行变量stmt内的SQL语句，查询账号为0700735的行数。

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。

5.12.6.2.2. INSERT

- 例1：

```

BEGIN
  DECLARE insert_table STATEMENT; ①
  PREPARE insert_table FROM 'INSERT INTO orctest1 values(?,?)' ; ②
  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; ③
END;
/

```

- ① 声明一个STATEMENT类型的变量insert_table。
- ② 使用关键字PREPARE将SQL语句'往表orctest1中插入数据'放进变量insert_table里。
- ③ 使用关键字EXECUTE执行上述SQL语句，分别往表orctest1中插入4条不同的数据。

查看表orctest1中的数据

```

select * from orctest1;
+-----+-----+
| age | gpa |
+-----+-----+
| 22  | 4.3 |
| 22  | 4.7 |
| 24  | 4.8 |
| 18  | 4.4 |
| 19  | 4.5 |
| 21  | 4.6 |
| 23  | 4.7 |
+-----+-----+

```

可以看到，表orctest1中成功插入了4条新的数据。

5.12.6.2.3. UPDATE

- 例1:表中只有一个字段是未知的

```

BEGIN
DECLARE update_table STATEMENT; ①
  PREPARE update_table FROM 'UPDATE orctest1 SET gpa=5.0 WHERE age = ?' ; ②
  EXECUTE update_table using 18;
  EXECUTE update_table using 19; ③
END;
/

```

① 声明一个STATEMENT类型的变量update_table。

② 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。

③ 使用关键字EXECUTE执行变量update_table中的SQL语句，分别更新表orctest1中年龄为18, 19的gpa设为5.0。

再次查看表orctest1，可以看到年龄为18和19的gpa已经成功更新了。

```

select * from orctest1;
+---+---+
| age | gpa |
+---+---+
| 22  | 4.3  |
| 22  | 4.7  |
| 24  | 4.8  |
| 18  | 5.0  |
| 19  | 5.0  |
| 21  | 4.6  |
| 23  | 4.7  |
+---+---+

```

- 例2:表中两个字段都是未知的

```

BEGIN
DECLARE update_table STATEMENT; ①
  PREPARE update_table FROM 'UPDATE orctest1 SET gpa=? WHERE age = ?' ; ②
  EXECUTE update_table using 4.0,21;
  EXECUTE update_table using 4.4,23; ③
END;
/

```

① 声明一个STATEMENT类型的变量update_table。

② 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。

③ 使用关键字EXECUTE执行变量update_table中的SQL语句，分别更新表orctest1中年龄为21的gpa为4.0，年龄23的gpa为4.4。

查看表orctest1，可以看到年龄为21，和23的gpa已经成功更新了。

```

select * from orctest1;
+---+---+
| age | gpa |
+---+---+
| 22  | 4.3  |
| 22  | 4.7  |
| 24  | 4.8  |
| 18  | 5.0  |
| 19  | 5.0  |
| 21  | 4.0  |
| 23  | 4.4  |
+---+---+

```

5.12.6.2.4. DELETE

- 例1:

```

BEGIN
DECLARE delete_table STATEMENT; ①
  PREPARE delete_table FROM 'DELETE FROM orctest1 WHERE age= ?' ; ②
  EXECUTE delete_table using 22;
  EXECUTE delete_table using 24; ③
END;
/

```

① 声明一个STATEMENT类型的变量delete_table。

② 使用关键字PREPARE，将SQL语句’删除表orctest1的相应年龄的记录’放进变量delete_table里。

③ 使用关键字EXECUTE，执行变量delete_table里的SQL语句，分别删除表orctest1中年龄为22和24的记录。

再次查看表orctest1

```

select * from orctest1;
+-----+-----+
| age | gpa |
+-----+-----+
| 18  | 5.0 |
| 19  | 5.0 |
| 21  | 4.0 |
| 23  | 4.4 |
+-----+-----+

```

5.12.7. 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。

5.12.7.1. ROW_COUNT

- 例1:ROW_COUNT

```

CREATE OR REPLACE PROCEDURE upd_zara_gpa(in v_age INT, in v_gpa INT, out no_of_rows INT) ①
language sql ②
BEGIN
  UPDATE zara SET gpa = v_gpa WHERE age = v_age; ③
  GET DIAGNOSTICS no_of_rows = ROW\_COUNT; ④
  PUT_LINE('更新行:' || no_of_rows); ⑤
END;
/
BEGIN
  DECLARE row INT; ⑥
  upd_zara_gpa(22, 4.5, row); ⑦
END;

```

① 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，数据类型均为整型；参数no_of_rows的类型为OUT，数据类型为整型。

② 指定程序的主体语言为SQL。

③ 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。

④ 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。

⑤ 打印出SQL语句所处理的行数。

- ⑥ 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。
- ⑦ 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。

输出结果为：

```
+-----+
| output |
+-----+
| 更新行:2 |
+-----+
```

5.12.7.2. EXCEPTION 1

- 例1：创建存储过程proc_11

```
CREATE OR REPLACE PROCEDURE proc_11(out b int) ①
BEGIN
DECLARE sqlcode, sql_code int;
DECLARE sqlstate, sql_state string;
DECLARE errMsg string; ②
DECLARE continue handler for sqlexception ③
BEGIN
    GET DIAGNOSTICS exception 1 errMsg = DB2_TOKEN_STRING; ④
    VALUES(sqlcode, sqlstate) INTO sql_code, sql_state; ⑤
    IF sql_code != -438 or sql_state != '54321' or errMsg != 'exception on purpose' THEN
        signal sqlstate '30001'; ⑥
    END IF;
END;
signal sqlstate value '54321' set MESSAGE_TEXT = 'exception on purpose'; ⑦
SET b = 20; ⑧
PUT_LINE('sqlcode is:'||sql_code);
PUT_LINE('sqlstate is:'||sql_state); ⑨
PUT_LINE('value of b:'||b);
END;
/
```

- ① 创建存储过程proc_11，参数b的类型为OUT，数据类型为整型。
- ② 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。
- ③ 为异常sqlexception声明一个continue的异常处理器。
- ④ 使用关键字GET DIAGNOSTICS获取执行SQL返回的错误或警告信息。
- ⑤ 将sqlcode, sqlstate赋值给变量sql_code, sql_state。
- ⑥ 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。
- ⑦ 抛出一个异常，sqlstate的值为54321，MESSAGE_TEXT的值为exception on purpose。
- ⑧ 给参数b赋值为20。
- ⑨ 分别打印出变量sql_code, sql_state以及参数b的值。

- 例1：调用存储过程proc_11

```
BEGIN
DECLARE b INT default -10;
CALL proc_11(b);
IF b != 20 THEN
    signal sqlstate value '30002';
END IF;
END;
/
```

输出结果为：

```
+-----+
|      output      |
+-----+
| sqlcode is:-438 |
| sqlstate is:54321|
| value of b:20   |
+-----+
```

5.13. 系统预定义函数/过程

5.13.1. 查看预定义函数/过程

5.13.1.1. 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
```



由于我们的PLSQL是运行在Inceptor Server端的SESSION之中，当一条PLSQL语句尚未结束并且会运行很长时间的时候，可能客户端已经用Ctrl+C杀掉自己。但Server端由于还在执行PLSQL，无法侦听到客户端死掉，这样这个SESSION中的PLSQL会变成类似僵尸进程。虽然客户端已经决定放弃这次执行，但该PLSQL依然会继续执行到结束，设想其中如果有大量的SQL语句执行或者干脆有个死循环，会耗费大量的Server端资源。所以我们需要手动执行一些命令来终止Server端PLSQL的执行。

- 查看正在运行的PLSQL程序的SESSION ID

仅在InceptorServer 2中有效

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

仅在InceptorServer 2中有效

```
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。



自检是借鉴了操作系统中进程调度的思想。自检的粒度为一条PLSQL语句，也就是说每执行一条PLSQL语句会自检一次（也就是被OS调度一次）。所以如果有一条SQL语句执行时间特别长，那么KILL该进程后并不会马上生效，需要等到该SQL语句执行完毕才会发生自检并终止。这种情况如果确定要终止该进程，可到Spark的4040页面去KILL掉当前的SQL，这样外层PLSQL就会立即终止。

5.13.1.2. 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)
|-----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)
| 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)
+-----+
25 rows selected (0.15 seconds)
```

- 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 default.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:删除函数/过程

删除自定义过程overload_test_proc;

```
drop plsql procedure overload_test_proc;
No rows affected (0.471 seconds)
```

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)
| | -----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)
| | 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

仅在InceptorServer 2中有效

```
PS PLSQL;
+-----+
| session id: plsql statement |
+-----+
+-----+
No rows selected (0.093 seconds)
```

- 例5:终止正在运行的PLSQL程序

仅在InceptorServer 2中有效

```
KILL PLSQL <SESSION_ID>
```

5.13.2. 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()
```

5.13.3. 预定义函数/过程的使用

5.13.3.1. set_env与get_env

- 例1:`set_env`与`get_env`的使用

```
BEGIN
  DECLARE a STRING;
  DECLARE b STRING;
  set_env('aa','hello'); ①
  SET a = get_env('aa'); ②
  PUT_LINE('the value of a is: '||a); ③
  set_env('bb','world');
  SET b = get_env('bb');
  PUT_LINE('the value of b is: '||b);
END;
/
```

① 调用过程`set_env`，定义一个名为aa的环境变量，值为hello。

② 调用函数`get_env`，获取环境变量aa的值，并赋值给a。

③ 输出变量a的值，可以发现变量a与环境变量aa的值相同。

输出结果为：

```
+-----+
|       output      |
+-----+
| the value of a is: hello |
| the value of b is: world |
+-----+
2 rows selected (0.162 seconds)
```

5.13.3.2. sqlcode与sqlerrm

- 例1:`sqlcode`与`sqlerrm`的使用

```

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;
  dbms_output.put_line('error code is: ' ||sqlcode()); ①
  dbms_output.put_line('error message is: ' ||sqlerrm()); ②
END;
/

```

① 当异常发生时，获取当前异常的Error code，并赋值给变量v_code。

② 当异常发生时，获取当前异常的Error message，并赋值给变量v_errm。

输出结果为：

```

+-----+
|          output          |
+-----+
| error code is: -1422      |
| error message is: TOO_MANY_ROWS |
+-----+
2 rows selected (2.98 seconds)

```

5.13.3.3. sqlerrm(int)

- 例1：

```

CREATE OR REPLACE PROCEDURE test_errm(IN b INT)
BEGIN
  DECLARE errmessage STRING;
  SET errmessage = sqlerrm(b);
  PUT_LINE('error meaasge is: ' ||errmessage);
END;
/
BEGIN
  test_errm(438);
  test_errm(100);
  test_errm(110);
END;
/

```

输出结果为：

```

+-----+
|          output          |
+-----+
| error meaasge is:Application raised error or warning with diagnostic text: "".
| error meaasge is:No row was found for FETCH, UPDATE or DELETE; or the result of a query is an empty table
| error meaasge is:Unknown error code: 110
+-----+

```

5.13.3.4. 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

```

CREATE OR REPLACE FUNCTION put_line_test(v1 int)
RETURNS DOUBLE          ①
BEGIN
    DECLARE v2 DOUBLE;
    DECLARE v3 STRING;
    put_line(null); ②
    DBMS_OUTPUT.PUT_LINE(v1);
    SET v2 = 2;
    SET v3 = "I'm a string.";
    put_line(v2); ③
    DBMS_OUTPUT.PUT_LINE('v2: ' || v2);
    put_line('v2 + v1'); ④
    DBMS_OUTPUT.PUT_LINE(v2 + v1);
    put_line(v3); ⑤
    SET v3 = null;
    DBMS_OUTPUT.PUT_LINE(v3);
    return v2 * v1; ⑥
END;
/
BEGIN
    dbms_output.put_line("Executing put_line_test(1)");
    put_line(put_line_test(1)); ⑦
END;

```

- ① 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。
- ② 使用put_line函数，打印出null值。
- ③ 使用put_line函数，打印出变量v2的值。
- ④ 使用put_line函数，打印出字符串v2+v1。
- ⑤ 使用put_line函数，打印出v3的值。
- ⑥ 函数返回值为v1和v2的乘积。
- ⑦ 使用put_line函数，打印出put_line_test(1)的全部值。

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

5.13.3.5. raise_application_error

- 例1:raise_application_error的使用

```

BEGIN
    DECLARE num_tables INT;
    SELECT COUNT(*) INTO num_tables FROM transactions; ①
    IF num_tables < 1000 THEN
        RAISE_APPLICATION_ERROR
            (-20101, 'Expecting at least 1000 tables'); ②
    ELSE
        dbms_output.put_line(num_tables); ③
    END IF;
END;

```

- ① 查询表transactions的行数，并赋值给变量num_tables。

② 如果表transactions的行数小于1000，则抛出异常。

③ 否则，就输出表transactions的行数。

输出结果为：

```
Error: Error while processing statement: FAILED: Execution Error, return code -20101 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -20101, error message = Expecting at least 1000 tables
*****
System function raise_application_error (LINE -1, COLUMN -1, TEXT "null")
ANONYMOUS BLOCK (LINE 6, COLUMN 0, TEXT "RAISE_APPLICATION_ERROR
(-20101, 'Expecting at least 1000 tables');");
*****
(state=08S01,code=-20101)
```

- 例2:例1不抛出异常的情况

```
BEGIN
    DECLARE num_tables INT;
    SELECT COUNT(*) INTO num_tables FROM transactions; ①
    IF num_tables < 10 THEN
        RAISE_APPLICATION_ERROR
        (-20101, 'Expecting at least 1000 tables'); ②
    ELSE
        dbms_output.put_line(num_tables); ③
    END IF;
END;
```

① 查询表transactions的行数，并赋值给变量num_tables。

② 如果表transactions的行数小于10，则抛出异常。

③ 否则，就输出表transactions的行数。

输出结果为：

```
+-----+
| output |
+-----+
| 20     |
+-----+
1 row selected (6.441 seconds)
```

5.13.3.6. get_columns

- 例1:get_columns的使用

```

CREATE OR REPLACE PROCEDURE get_columns_test()
BEGIN
    DECLARE TYPE nest_table AS STRING ARRAY[5];
    DECLARE TYPE nest_table_1 AS STRING ARRAY[5];
    DECLARE a_table nest_table;
    DECLARE b_table nest_table_1 default null;
    DECLARE s string;
    DECLARE a_first STRING;
    DECLARE a_last STRING;
    DECLARE b_first STRING;
    DECLARE b_last STRING;
    SET a_table = ARRAY['this', 'is', 'a', 'table', 'ID'];
    SET b_table = get_columns("transactions", a_table);
    DBMS_OUTPUT.PUT_LINE('a_table:');
    SET a_first = a_table.ARRAY_FIRST;
    PUT_LINE('the first one:' || a_table(a_first));
    SET a_last = a_table.ARRAY_LAST;
    PUT_LINE('the last one:' || a_table(a_last));
    IF b_table is not null THEN
        SET b_first = b_table.ARRAY_FIRST;
        PUT_LINE('the first one:' || b_table(b_first));
        SET b_last = b_table.ARRAY_LAST;
        PUT_LINE('the last one:' || b_table(b_last));
    END IF;
END;
/
BEGIN
    get_columns_test();
END;

```

5.14. 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。

5.14.1. 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

```

BEGIN
DECLARE v_time STRING;
DECLARE sqlstate, sql_state string;
DECLARE sqlcode, sql_code int; ①
SELECT trans_time INTO v_time FROM transactions WHERE price=12.13; ②
values(sqlcode, sqlstate) into sql_code, sql_state; ③
PUT_LINE('the time is:'||v_time);
PUT_LINE('the sqlcode is:'||sql_code);
PUT_LINE('the sqlstate is:'||sql_state); ④
END;
/

```

- ① 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。
- ② 查询表transactions中价格为12.13的交易时间，并放进变量v_time里。
- ③ 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。
- ④ 分别打印出变量v_time, sql_code, sql_state的值。

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

```

BEGIN
DECLARE v_time STRING;
DECLARE sqlstate, sql_state string;
DECLARE sqlcode, sql_code int; ①
SELECT trans_time INTO v_time FROM transactions WHERE price=12.00; ②
values(sqlcode, sqlstate) into sql_code, sql_state; ③
PUT_LINE('the time is:'||v_time);
PUT_LINE('the sqlcode is:'||sql_code);
PUT_LINE('the sqlstate is:'||sql_state); ④
END;
/

```

- ① 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。
- ② 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。
- ③ 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。
- ④ 分别打印出变量v_time, sql_code, sql_state的值。

输出结果为：

```
+-----+
|       output      |
+-----+
| the time is:    |
| 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。

5.14.2. 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：

```
CREATE OR REPLACE PROCEDURE myproc(IN v_var01 INT) ①
BEGIN
    IF v_var01 > 100 THEN
        SIGNAL SQLSTATE '70889' SET message_text='The input parameter must be lower than 100!';
    ②
    END IF;
END;
/
CALL myproc(200);
```

① 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。

② 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!。

输出结果为：

```
Error: Error while processing statement: FAILED: Execution Error, return code -438 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -438, error message = The input parameter must be lower than 100!
*****
PROCEDURE 'myproc' (LINE 4, COLUMN 24, TEXT "SIGNAL SQLSTATE '70889' SET message_text='The
input parameter must be lower than 100!';")
ANONYMOUS BLOCK (LINE 1, COLUMN 5, TEXT "myproc(200)")
***** (state=08S01, code=-438)
```

5.14.3. 异常分类

5.14.3.1. 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，SQLEXCEPTION，SQLWARNING：

- **NOT FOUND**: SQLCODE的值为+100且SQLSTATE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。
- **SQLEXCEPTION**: SQLCODE值小于0且SQLSTATE的值为02000, 01XXX之外的所有值。

5.14.3.2. 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会忽视这个异常，并将继续执行异常所在语句的下一条语句，只有当发生了SQLEXCEPTION类型的异常时，Inceptor才会从异常发生处，跳出，并抛出一个异常。

5.14.3.2.1. 语法

- 声明自定义异常：

```
DECLARE condition_name CONDITION FOR SQLSTATE <value>;
```

- 抛出异常：

```
SIGNAL condition_name;
```

5.14.3.2.2. 实例

- 例1：自定义一个SQLWARNING类型的异常

```

CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
    DECLARE C1 CONDITION FOR SQLSTATE '01089'; ①
    BEGIN
        DECLARE v_price DOUBLE;
        SELECT price INTO v_price FROM transactions WHERE trans_id=v_id;
        PUT_LINE('the price is:'||v_price); ②
    END;
    SIGNAL C1; ③
END;
/
CALL condition_test('929634984');

```

- ① 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。
- ② 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。
- ③ 抛出异常C1。

输出结果为：

```

+-----+
|       output      |
+-----+
| the price is:11.11 |
+-----+

```

- 例2：自定义一个SQLEXCEPTION类型的异常

```

CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
    DECLARE C1 CONDITION FOR SQLSTATE '76089'; ①
    BEGIN
        DECLARE v_price DOUBLE;
        SELECT price INTO v_price FROM transactions WHERE trans_id=v_id;
        PUT_LINE('the price is:'||v_price); ②
    END;
    SIGNAL C1; ③
END;
/
CALL condition_test('929634984');

```

- ① 声明一个SQLSTATE值为76089的异常C1，即一个SQLEXCEPTION类型的异常。
- ② 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。
- ③ 抛出异常C1。

输出结果为：

```

Error: Error while processing statement: FAILED: Execution Error, return code -438 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -438, error message =
*****
PROCEDURE 'condition_test' (LINE 9, COLUMN 7, TEXT "SIGNAL C1;")
ANONYMOUS BLOCK (LINE 1, COLUMN 5, TEXT "condition_test('929634984')")*
***** (state=08S01,code=-438)

```

5.14.3.3. 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。

5.14.3.4. 实例

5.14.3.4.1. 自定义异常的处理

- 例1.1:自定义一个SQLEXCEPTION类型的异常

```
CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
    DECLARE C1 CONDITION FOR SQLSTATE '76089'; ①
    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;
    SIGNAL C1; ②
END;
/
CALL condition_test('929634984');
```

① 自定义一个SQLEXCEPTION类型的异常C1，SQLSTATE 值为76089。

② 抛出异常C1。

输出结果为：

```
Error: Error while processing statement: FAILED: Execution Error, return code -438 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -438, error message =
*****
PROCEDURE 'condition_test' (LINE 9, COLUMN 7, TEXT "SIGNAL C1;")
ANONYMOUS BLOCK (LINE 1, COLUMN 5, TEXT "condition_test('929634984')")*
***** (state=08S01,code=-438)
```

- 例1.2:为一个SQLEXCEPTION类型的异常声明一个CONTINUE类型的异常处理器

```
CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
    DECLARE C1 CONDITION FOR SQLSTATE '76089'; ①
    DECLARE CONTINUE HANDLER FOR C1 ②
    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;
    SIGNAL C1; ③
END;
/
CALL condition_test('929634984');
```

- ① 自定义一个SQLEXCEPTION类型的异常C1，SQLSTATE 值为76089。
- ② 为异常C1声明一个CONTINUE类型的异常处理器。
- ③ 抛出异常C1。

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

```
CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
    DECLARE C1 CONDITION FOR SQLSTATE '01089'; ①
    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;
    SIGNAL C1; ②
    PUT_LINE('there is a condition');
END;
/
CALL condition_test('594819547');
```

- ① 声明一个sqlstate以01开头的SQLWARNING类型的异常C1。
- ② 抛出异常C1。

输出结果为：

```
+-----+
|       output      |
+-----+
| the price is:6.36
| 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

```
CREATE OR REPLACE PROCEDURE condition_test(IN v_id STRING)
BEGIN
    DECLARE C1 CONDITION FOR SQLSTATE '01089'; ①
    DECLARE EXIT HANDLER FOR C1 ②
    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;
    SIGNAL C1; ③
    PUT_LINE('there is a condition');
END;
/
CALL condition_test('594819547');
```

- ① 声明一个sqlstate以01开头的SQLWARNING类型的异常C1。
- ② 为SQLWARNING类型的异常C1，声明一个EXIT类型的异常处理器。
- ③ 抛出异常C1。

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。

5.14.3.4.2. 预定义异常的处理

- 例3.1:NOT FOUND类型的预定义异常

```
BEGIN
  DECLARE v_amount DOUBLE;
  BEGIN
    DECLARE v_price DOUBLE;
    SELECT price INTO v_price FROM transactions WHERE trans_id=111111111;
    PUT_LINE('the price is:'||v_price);
  END;
  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      |
+-----+
| the price is:   |
| 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

```
BEGIN
  DECLARE v_amount DOUBLE;
  DECLARE EXIT HANDLER FOR NOT FOUND ①
  BEGIN
    DECLARE v_price DOUBLE;
    SELECT price INTO v_price FROM transactions WHERE trans_id=111111111;
    PUT_LINE('the price is:'||v_price);
  END;
  SELECT amount INTO v_amount FROM transactions WHERE price=6.36;
  PUT_LINE('the amount is:'||v_amount);
END;
/
```

① 为NOT FOUND类型的异常声明一个EXIT类型的异常处理器。

输出结果为：

```
+-----+
|       output      |
+-----+
| the amount is:800.0 |
+-----+
```

- 例4.1:SQLEXCEPTION类型的预定义异常

```

BEGIN
  DECLARE v_amount DOUBLE;
  BEGIN
    DECLARE v_price DOUBLE;
    SELECT price INTO v_price FROM t1 WHERE trans_id=111111111;
    PUT_LINE('the price is:'||v_price); ①
  END;
  SELECT amount INTO v_amount FROM transactions WHERE price=6.36;
  PUT_LINE('the amount is:'||v_amount);
END;
/

```

① 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SQLEXCEPTION类型的异常。

输出结果为：

```

Error: Error while processing statement: FAILED: Execution Error, return code -204 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -204, error message = Table or view doesn't exist
*****
ANONYMOUS BLOCK (LINE 5, COLUMN 37, TEXT "")
*****
(state=08S01,code=-204)

```

- 例4.2：为SQLEXCEPTION类型的预定义异常声明一个CONTINUE类型的异常处理器

```

BEGIN
  DECLARE v_amount DOUBLE;
  DECLARE CONTINUE HANDLER FOR sqlexception ①
  BEGIN
    DECLARE v_price DOUBLE;
    SELECT price INTO v_price FROM t1 WHERE trans_id=111111111;
    PUT_LINE('the price is:'||v_price);
  END;
  SELECT amount INTO v_amount FROM transactions WHERE price=6.36;
  PUT_LINE('the amount is:'||v_amount);
END;
/

```

① 为sqlexception类型的异常，声明一个CONTINUE类型的异常处理器。

输出结果为：

output
the amount is:800.0

本例中为SQLEXCEPTION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。

5.14.4. 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。

5.14.4.1. 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>];
```

5.14.4.2. 用例

- 例1:

```
CREATE OR REPLACE PROCEDURE exp_proc(out b int) ①
BEGIN
    DECLARE c CONDITION FOR sqlstate '54321'; ②
    DECLARE CONTINUE HANDLER FOR sqlstate '34567' ③
BEGIN
    DECLARE CONTINUE HANDLER for c ④
    BEGIN
        resignal sqlstate value '34567'; ⑤
    END;
    signal c; ⑥
END;
dbms_output.put_line('error code is: ' ||sqlcode()); ⑦
END;
/
BEGIN
    DECLARE b INT default 0;
    DECLARE CONTINUE HANDLER FOR sqlstate '34567'
    BEGIN
        SET b = -10;
    END;
    CALL exp_proc(b); ⑧
END;
/
```

- ① 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。
- ② 声明一个名为c的自定义异常，sqlstate的值为54321。
- ③ 对值为34567的sqlstate声明一个CONTINUE类型的异常处理器。
- ④ 为自定义异常c声明一个CONTINUE类型的异常处理器。
- ⑤ 使用关键字resignal抛出一个sqlstate的值为34567的异常。
- ⑥ 使用关键字signal抛出自定义异常c。
- ⑦ 输出该存储过程内的sqlcode的值。
- ⑧ 调用存储过程exp_proc。

输出结果为：

```
+-----+
|      output      |
+-----+
| error code is: 0 |
+-----+
1 row selected (0.083 seconds)
```

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。

5.14.5. 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：

```

CREATE OR REPLACE PROCEDURE p1(value string)
BEGIN
    DECLARE ex CONDITION FOR SQLSTATE '76089'; ①
    set value = 'p1: ' || value;
    dbms_output.put_line(value);
    SIGNAL ex; ②
END;
/
CREATE OR REPLACE PROCEDURE p2(value string) ③
BEGIN
    set value = 'p2: ' || value;
    dbms_output.put_line(value);
    p1(value); ④
END;
/
CREATE OR REPLACE PROCEDURE p3(value string) ⑤
BEGIN
    set value = 'p3: ' || value;
    dbms_output.put_line(value);
    p2(value); ⑥
END;
/
BEGIN
DECLARE value string;
DECLARE id_v int;
DECLARE ret_v int;
SET value = 'init value';
dbms_output.put_line(value);
p3(value); ⑦
END;
/

```

- ① 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。
- ② 抛出异常ex
- ③ 创建过程p2。
- ④ 在过程p2内调用过程p1。
- ⑤ 创建过程p3。
- ⑥ 在过程p3内调用过程p2。
- ⑦ 最后执行过程p3的内容。

输出结果为：

```

Error: Error while processing statement: FAILED: Execution Error, return code -438 from
org.apache.hadoop.hive.ql.exec.PLTask. Hit PL exception in executing PL Task.
Error code = -438, error message =
*****
PROCEDURE 'p1' (LINE 7, COLUMN 12, TEXT "SIGNAL ex;")
PROCEDURE 'p2' (LINE 5, COLUMN 2, TEXT "p1(value);")
PROCEDURE 'p3' (LINE 5, COLUMN 2, TEXT "p2(value);")
ANONYMOUS BLOCK (LINE 7, COLUMN 2, TEXT "p3(value);")
***** (state=08S01, code=-438)
*****
```

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

```

CREATE OR REPLACE PROCEDURE p1(value string)
BEGIN
    DECLARE ex CONDITION FOR SQLSTATE '76089'; ①
    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;
/

```

① 为自定义异常ex声明一个CONTINUE类型的异常处理器。

输出结果为：

output
init value
p3: init value
p2: p3: init value
p1: p2: p3: init value

4 rows selected (0.214 seconds)

5.15. 注意事项

5.15.1. 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。

5.15.2. 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。

5.15.2.1. Beeline+InceptorServer 2

- 以Beeline方式连接InceptorServer 2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;
```

5.15.2.2. CLI+InceptorServer 1

- 以CLI方式连接InceptorServer 1时，可通过以下命令，手动打开对DB2 dialect的支持。

客户端

```
set plsql.client.dialect=db2;
```

服务器端

```
set plsql.server.dialect=db2;
```

5.15.3. 分号的支持

- 以Beeline方式连接InceptorServer 2，或以CLI方式连接InceptorServer 1时的情况下，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。

5.15.4. 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

```
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;
No rows affected (0.003 seconds)
```

```
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 |
+-----+
```

6. Inceptor函数和运算符手册

6.1. Inceptor函数和运算符手册中的表

在《Inceptor函数和运算符手册》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: 列从左到右依次为name, acc_num, password, citizen_id, bank_acc, reg_date, acc_level:

马**	6513065	115591	14*****	960*****41	20110101	A
祝**	16670192	205239	23*****	737*****71	20100101	C
华*	15224133	1531547	42*****	326*****07	20080214	B
魏**	13912384	841242	52*****	685*****48	20091202	A
宁**	14580952	986634	42*****	977*****76	20081031	D
邱*	10700735	737297	34*****	143*****18	20121024	A
李*	18725869	600709	46*****	430*****84	20130702	E
潘**	16600641	990590	51*****	484*****08	20110430	C
李**	12755506	015859	31*****	424*****37	20110916	D
管**	12394923	783438	33*****	999*****74	20141003	C

- 交易信息表transactions: 列从左到右依次为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
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	KC9102928	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, 列名从左到右为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, 列名从左到右为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
MAT100	微积分

6.2. 关系运算符

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 from system.dual limit 1;
Result: true
```

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 from system.dual limit 1;
Result: false
```

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，如果A与B相等，则返回TRUE，否则返回FALSE。

举例

```
select 1 = 1 from system.dual limit 1;
Result: true
```

操作符	返回类型	描述
A == B	Boolean	判断A是否等于B

说明

和 = 用法一致

举例

```
select 1 == 1 from system.dual limit 1;
Result: true
```

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 from system.dual limit 1;
Result: false
```

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 from system.dual limit 1;
Result: false
```

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 from system.dual limit 1;
Result: true
```

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NULL。

举例

```
select 1 = 1 from system.dual limit 1;
Result: true
```

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 from system.dual limit 1;
Result: false
```

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，如果A与B相等，则返回TRUE，否则返回FALSE。

举例

```
select 1 = 1 from system.dual limit 1;
Result: true
```

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__' from system.dual limit 1;
Result: true
```

操作符	返回类型	描述
A rlike 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*' from system.dual limit 1;
Result: true
```

操作符	返回类型	描述
A regexp B	Boolean	判断A是否符合正则表达式B

说明

用法和“rlike”完全一致

举例

```
select 'foobar' regexp 'foo*' from system.dual limit 1;
Result: true
```

6.3. 算术运算符

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 from system.dual limit 1;
Result: 1.1
```

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。

举例

```
select 1 - 0.1 from system.dual limit 1;
Result: 0.9
```

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 from system.dual limit 1;
Result: 0.1
```

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 from system.dual limit 1;
Result: 1.0
```

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 from type limit 1;
Result: 1.0
```

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 from system.dual limit 1;
Result: 0
```

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 from system.dual limit 1;
Result: 3
```

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 from system.dual limit 1;
Result: 0
```

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 from system.dual limit 1;
Result: -2
```

6.4. 逻辑运算符

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 from system.dual limit 1;
Result: true
```

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 from system.dual limit 1;
Result: true
```

操作符	返回类型	描述
NOT 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 from system.dual limit 1;
Result: false
```

操作符	返回类型	描述
!A	Boolean	对A的逻辑布尔值求反

说明

用法和' NOT' 一致。

举例

```
select !true from system.dual limit 1;
Result: false
```

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) from system.dual limit 1;
Result: true
```

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) from system.dual limit 1;
Result: false
```

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 from system.dual s1 where exists (select 1 from system.dual s2 where s1.a=s2.a) limit 1;
Result: 1
```

6.5. 数学函数

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。

举例

```
select round(1.4523,3) from system.dual limit 1;
Result: 1.452
```

函数名	返回类型	描述
floor(double a)	BigInt	不大于a的最大整数

说明 返回不大于a的最大整数。

举例

```
select floor(1.314) from system.dual limit 1;
Result: 1
```

函数名	返回类型	描述
ceil(double a)	BigInt	不小于a的最小整数

说明

返回不小于a的最小整数。

举例

```
select ceil(2.01) from system.dual limit 1;
Result: 3
```

函数名	返回类型	描述
ceiling(double a)	BigInt	不小于a的最小整数

说明

用法和'ceil'一致。

举例

```
select ceiling(2.01) from system.dual limit 1;
Result: 3
```

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。

举例

```
select rand(100) from system.dual limit 1;
Result: 0.7220096548596434
```

函数名	返回类型	描述
exp(double n)	Double	返回e的n次方

说明

返回e的n次方。

举例

```
select exp(1) from system.dual limit 1;
Result: 2.7182818284590455
```

函数名	返回类型	描述
ln(double a)	Double	自然对数

说明

返回指定值的自然对数。

举例

```
select ln(exp(1.314)) from system.dual;
Result: 1.314
```

函数名	返回类型	描述
log10(double a)	Double	10为底的对数

说明

返回指定值以10为底的对数。

举例

```
select log10(100) from system.dual limit 1;
Result: 2.0
```

函数名	返回类型	描述
log2(double a)	Double	2为底的对数

说明

返回指定值以2为底的对数。

举例

```
select log2(4) from system.dual limit 1;
Result: 2.0
```

函数名	返回类型	描述
log(double base, double a)	Double	返回指定底数的对数

说明

返回a以base为底的对数。

举例

```
select log(10,100) from system.dual limit 1;
Result: 2.0
```

函数名	返回类型	描述
pow(double a, double p)	Double	a的p次幂

说明

返回a的p次幂。

举例

```
select pow(10,2) from system.dual limit 1;
Result: 100.0
```

函数名	返回类型	描述
power(double a, double p)	Double	a的p次幂

说明

返回a的p次幂。

举例

```
select power(10,2) from system.dual limit 1;
Result: 100.0
```

函数名	返回类型	描述
sqrt(double a)	BigInt	平方根

说明

返回a的平方根。

举例

```
select sqrt1(100) from system.dual;
Result: 10.0
```

函数名	返回类型	描述
bin(BIGINT a)	String	返回二进制格式

说明

返回二进制格式。

举例

```
select bin(10) from system.dual limit 1;
Result: 1010
```

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。

举例

```
select hex(16) from system.dual limit 1 ;
Result: 10
select hex('16') from system.dual limit 1 ;
Result: 3136
```

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numeric也转化为字符格式再转化为二进制。

举例

```
select unhex(3136) from system.dual limit 1 ;
Result: 16
```

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。

举例

```
SELECT conv(100, 2, 10) FROM system.dual LIMIT 1;
Result: 4
```

函数名	返回类型	描述
abs(double a)	Double	绝对值

说明

返回a的绝对值。

举例

```
select abs(-1314) from system.dual limit 1;
Result: 1314
```

函数名	返回类型	描述
pmod(int a, int b)	Numeric	余数的绝对值

说明

返回|a%b|。

举例

```
select pmod(-4,3) from system.dual limit 1;
Result: 2
```

函数名	返回类型	描述
sin(double a)	Double	正弦

说明

返回a的正弦值，其中a是弧度制。

举例

```
select sin(pi()/2) from system.dual;
Result: 1.0
```

函数名	返回类型	描述
asin(double a)	Double	反正弦

说明

返回a的反正弦值，a必须大于-1小于1，否则返回null。

举例

```
select asin(1) from system.dual;
Result: 1.5707963267948966
```

函数名	返回类型	描述
cos(double a)	Double	余弦

说明

返回a的余弦值，其中a是弧度制。

举例

```
select cos(pi()) from system.dual;
Result: -1.0
```

函数名	返回类型	描述
acos(double a)	Double	反余弦

说明

返回a的反余弦值，其中a是弧度制。

举例

```
select acos(1) from system.dual;
Result: 0.0
```

函数名	返回类型	描述
tan(double a)	Double	正切

说明

返回a的正切值，其中a是弧度制。

举例

```
select tan(pi()/4) from system.dual;
Result: 0.9999999999999999
```

函数名	返回类型	描述
atan(double a)	Double	余切

说明

返回a的余切值，其中a是弧度制。

举例

```
select atan(1) from system.dual;
Result: 0.7853981633974483
```

函数名	返回类型	描述
positive(int a)	Numeric	直接返回a

举例

```
select positive(-1) from system.dual;
Result: -1
```

函数名	返回类型	描述
negative(int a)	Numeric	返回-a

举例

```
select negative(1) from system.dual;
Result: -1
```

函数名	返回类型	描述
sign(x)	Double	判断正负

说明

x为正则返回1.0，为负则返回-1.0，为0则返回0.

举例

```
SELECT sign(-10) FROM system.dual LIMIT 1;
Result: -1.0
```

函数名	返回类型	描述
pi()	Double	返回pi的值

举例

```
select pi() from system.dual;
Result: 3.141592653589793
```

函数名	返回类型	描述
degrees(double rad)	Double	弧度转角度

说明

把弧度制的rad转化为角度。

举例

```
select degrees(pi()) from system.dual;
Result: 180.0
```

函数名	返回类型	描述
radians(double ang)	Double	角度转弧度

说明

把角度制的ang转化为弧度。

举例

```
select radians(180) from system.dual;
Result: 3.141592653589793
```

函数名	返回类型	描述
e()	Double	返回e

说明

返回自然常数e

举例

```
select e() from system.dual;
Result: 2.718281828459045
```

运算符	返回类型	描述
<a> div 	INT	返回a除以b的商（整数部分）。

```
SELECT 5 div 2 FROM system.dual LIMIT 1;
+-----+
| _cθ |
+-----+
| 2    |
+-----+
```

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。

```
SELECT prime(4) FROM system.dual LIMIT 1;
+-----+
| _cθ |
+-----+
| 3    |
+-----+
```

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。

```
SELECT max_scalar(1, 4, 8) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 8   |
+-----+
```

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。

```
SELECT min_scalar(1,4,8) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 1   |
+-----+
```

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。

```
SELECT least(2, 3, 1) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 1   |
+-----+
```

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>。

```
SELECT trunc(123.456, 1) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 123.4 |
+-----+
```

6.6. 类型转换函数

表 3. BINARY

函数名	返回类型	描述
BINARY(<string> <binary>)	STRING	将参数转换成BINARY类型。

说明

参数仅允许为STRING或者BINARY类型。

举例

```
SELECT BINARY('38') FROM system.dual LIMIT 1;
```

表 4. 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。

说明

`expr` 代表被转换的对象，`TYPE` 代表指定目标类型。

举例

```
SELECT CAST('38' AS INT) FROM system.dual LIMIT 1;
```

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。

说明

和 `DATE(<expr> AS DATE)` 形式的`CAST` 函数意义相同。`<expr>` 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'

举例

```
SELECT DATE('2015-01-01 00:00:00') FROM system.dual LIMIT 1;
2015-01-01
```

6.7. 日期函数

表 5. ADD_MONTHS

函数名	返回类型	描述
ADD_MONTHS(<date>,<int>)	STRING	返回 <code><date></code> 加上 <code><int></code> 个月后的日期。

说明

参数 **date** 应该是datetime value, 或者可以隐性转换成DATE类型; 参数 **int** 应该是整数型, 或者可以隐性转换成整数型。

举例

```
SELECT ADD_MONTHS('201501',1) FROM system.dual LIMIT 1;
201502
```

表 6. EXTRACT

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') FROM system.dual LIMIT 1;
2015
```

表 7. TO_UNIX_TIMESTAMP

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') FROM system.dual LIMIT 1;
1421251200
```

表 8. UNIX_TIMESTAMP

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') FROM system.dual LIMIT 1;
1421251200
```

表 9. TO_TIMESTAMP

函数名	返回类型	描述
TO_TIMESTAMP([<date>[, <pattern>]])	BIGINT	把日期格式转化为UNIX时间戳

说明

和[UNIX_TIMESTAMP](#)用法完全一致

举例

```
SELECT TO_TIMESTAMP("2015-01-15", 'yyyy-MM-dd') FROM system.dual LIMIT 1;
1421251200
```

表 10. FROM_UNIXTIME

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') FROM system.dual LIMIT 1;
1970-01-01 08:00:00
```

表 11. TO_CHAR

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') FROM system.dual LIMIT 1;
20110511
```

表 12. TO_DATE

函数名	返回类型	描述
TO_DATE(<date>, <pattern>)	STRING	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。

举例

```
SELECT TO_DATE('2015-01-15 04:17:52') FROM system.dual LIMIT 1;
2015-01-15
```

表 13. TDH_TODATE

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 (24小时制) 或者 hh:mm:ss (12小时制)。

举例

```
SELECT TDH_TODATE('2015-01-15 00:00:00') FROM system.dual LIMIT 1;
2015-01-15

SELECT TDH_TODATE('2015-11-24 18:00:00','yyyy-MM-dd HH:mm:ss','yyyy/MM/dd hh:mm:ss') FROM
system.dual LIMIT 1;
2015/11/24 06:00:00
```

表 14. DAY

函数名	返回类型	描述
DAY(<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') FROM system.dual LIMIT 1;
15
```

表 15. DAYOFMONTH

函数名	返回类型	描述
DAYOFMONTH(<date>)	INT	返回指定时间是该月的第几天

说明

用法和[函数DAY](#)完全一致。

举例

```
SELECT DAYOFMONTH('2015-01-15') FROM system.dual LIMIT 1;
15
```

表 16. DAYOFYEAR

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') FROM system.dual LIMIT 1;
15
```

表 17. QUARTER

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') FROM system.dual LIMIT 1;
2
```

表 18. HOUR

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') FROM system.dual LIMIT 1;
22
```

表 19. MINUTE

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') FROM system.dual LIMIT 1;
58
```

表 20. SECOND

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') FROM system.dual LIMIT 1;
59
```

表 21. 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') FROM system.dual LIMIT 1;
3
```

表 22. 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。返回值为正说明date1较晚。

举例

```
SELECT DATEDIFF('2015-01-15', '2015-02-08') FROM system.dual LIMIT 1;
-24
```

表 23. DATE_ADD

函数名	返回类型	描述
DATE_ADD(<start_date>, <num_days>)	STRING	返回指定时间之后的第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。参数num_days必须是Int类型或者可以隐式转化为Int类型

举例

```
SELECT DATE_ADD('2015-01-15', 1) FROM system.dual LIMIT 1;
2015-01-16
```

表 24. DATE_SUB

函数名	返回类型	描述
DATE_SUB(<start_date>, <num_days>)	STRING	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。参数num_days必须是Int类型或者可以隐式转化为Int类型

举例

```
SELECT DATE_SUB('2015-01-15', 1) FROM system.dual LIMIT 1;
Result: 2015-01-14
```

表 25. DATE_FORMAT

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') FROM system.dual LIMIT 1;
20150115
```

表 26. STR_TO_DATE

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') FROM system.dual LIMIT 1;
2015-01-15
```

表 27. 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 FROM system.dual LIMIT 1;
2015-07-24 23:14:09
```

表 28. DAYOFWEEK

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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 5   |
+-----+
```

表 29. DAYOFWEEK_ISO

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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 4   |
+-----+
```

表 30. MILLISECOND

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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 847 |
+-----+
```



```
SELECT MILLISECOND('12:58:59.213') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 213 |
+-----+
```

表 31. 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'* yyyy MM', '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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 2.5483871 |
+-----+
```

```
SELECT MONTHS_BETWEEN('2010-10-15','2011-01-01') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| -2.5483871 |
+-----+
```

表 32. LAST_DAY

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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 2003-03-31 |
+-----+
```

表 33. DAYS

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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 736028 |
+-----+
```

表 34. TO_DAY_INTERVAL

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) FROM system.dual LIMIT 1;
2015-01-03 00:00:00
```

表 35. TO_MONTH_INTERVAL

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) FROM ckdttest LIMIT 1;
2015-03-01
```

表 36. TO_YEAR_INTERVAL

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) FROM ckdttest LIMIT 1;
2017-01-01
```

表 37. TO_HOUR_INTERVAL

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) FROM ckdttest LIMIT 1;
2015-01-01 02:00:00
```

表 38. TO_MINUTE_INTERVAL

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) FROM ckdttest LIMIT 1;
2015-01-01 00:02:00
```

表 39. TO_SECOND_INTERVAL

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) FROM ckdttest LIMIT 1;
2015-01-01 00:00:02
```

表 40. TD_INT_TO_TIME

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) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 12:34:56 |
+-----+
```

表 41. SYSTIMESTAMP

函数	返回类型	描述
SYSTIMESTAMP	TIMESTAMP	返回当前系统TIMESTAMP。和 <code>current_timestamp</code> 同义。注意， <code>*systimestamp</code> 使用时后面不加括号。*

```
SELECT SYSTIMESTAMP FROM system.dual LIMIT 1;
```

表 42. TRUNC

函数	返回类型	描述
TRUNC(<date>, <unit>)	DATE	需要两个参数：一个日期 <date>，和日期单位 <unit>。<unit> 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）。将 <date> 截到它所在的 <unit> 的开始。例如，如果 <unit> 是 MONTH，函数会将给定日期截到日期所在月的第一天。

```
SELECT TRUNC('2016-10-22 11:22:33', 'YYYY') FROM system.dual LIMIT 1;
```

```
+-----+
| _c0 |
+-----+
| 2016-01-01 |
+-----+
```

```
SELECT TRUNC('2016-10-22 11:22:33', 'MM') FROM system.dual LIMIT 1;
```

```
+-----+
| _c0 |
+-----+
| 2016-10-01 |
+-----+
```

表 43. NEXT_DAY

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') FROM system.dual LIMIT 1
```

```
+-----+
| _c0 |
+-----+
| 2015-12-27 |
+-----+
```

表 44. 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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 1970-01-01 00:00:00.0 |
+-----+
```

表 45. 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') FROM system.dual LIMIT 1;
SELECT TO_UTC_TIMESTAMP('1970-01-01 00:00:00.0', 'PST') FROM system.dual LIMIT 1;
```

6.8. 条件函数

表 46. IF

函数名	返回类型	描述
IF(booleanCondition, T valueTrue, T valueFalseOrNull)	String	根据条件真假返回不同的值

说明

如果booleanCondition为true，返回valueTrue，否则返回valueFalseOrNull。

举例

```
SELECT IF(true, 'trans', 'age') FROM system.dual LIMIT 1;
'trans'
```

表 47. 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') FROM system.dual LIMIT 1;
'trans'
```

表 48. 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 FROM system.dual;
'tran'
```

表 49. 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 FROM system.dual;
'trans'
```

6.9. 字符串函数

函数名	返回类型	描述
ascii(A)	Int	返回A第一个字符的ascii值

说明

A可以是除了Boolean之外的任何基本类型, 返回A第一个字符的ascii值。

举例

```
select ascii('20') from system.dual;
Result: 50
```

函数名	返回类型	描述
base64(A)	Int	把binary转化为Base64字符

说明

A可以是除了Boolean之外的任何基本类型，返回A的Base64编码格式的值。

举例

```
select base64(binary('1')) from system.dual;
Result: MQ==
```

函数名	返回类型	描述
concat(a, b, c.....)	String	合并输入的所有项

说明

A可以是除了Boolean之外的任何基本类型，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) from system.dual limit 1;
Result: 123
```

函数名	返回类型	描述
context_ngrams (array<array<string>>, array<string>, int K, intpf)	String	返回字符串中出现频率在Top-K内，符合指定pattern的词汇。

说明

从给定一个字符串上下文中，返回出现频率在Top-K内，符合指定pattern的词汇。

举例

```
SELECT context_ngrams(sentences(lower(tweet)), 2, 100 , 1000) FROM twitter;
```

函数名	返回类型	描述
concat_ws(string SEP, string A, string B...)	String	以SEP为分隔把A、B.....合并起来

说明

所有输入参数都必须是字符类型

举例

```
select concat_ws('|','1','2','3') from system.dual;
Result: 1|2|3
```

函数名	返回类型	描述
encode(A, B)	Int	将字符串A按B表示的编码格式编码，输出二进制字节码

说明

AB都是字符类型，用B指定的编码格式来对A编码。

举例

```
select encode('bigdata', 'UTF-16') from system.dual;
Result: 62696764617461
```

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

说明

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') from system.dual;
Result: 4
```

函数名	返回类型	描述
format_number(number x, int d)	Double	返回指定格式的number

说明

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) from system.dual;
Result: 100.00
```

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

说明

从指定地址获取json对象。

举例

```
select a.timestamp, get_json_object(a.appevents, '$.eventid'),
get_json_object(a.appevents, '$.eventname') from log a;
```

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

说明

str为String类型，filepath必须是存在的文件路径。

举例

```
select in_file('123', '/etc/hosts') from system.dual;
Result: false
```

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

说明

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') from system.dual;
Result: 3
```

函数名	返回类型	描述
length(A)	Int	返回A的长度

说明

A可以是除了Boolean之外的任何基本类型，返回A的长度。

举例

```
select length(1.1) from system.dual;
Result: 3
```

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

说明

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') from system.dual;
Result: 7
```

函数名	返回类型	描述
lower(string A)	String	把A中字母都转化为小写

说明

把A中字母都转化为小写。

举例

```
select lower('A') from system.dual;
Result: a
```

函数名	返回类型	描述
lcase(string A)	String	把A中字母都转化为小写

说明

用法和 'lower' 一致。

举例

```
select lower('A') from system.dual;
Result: a
```

函数名	返回类型	描述
lpad(string str, int len, string pad)	String	用pad补全字符到指定长度

说明

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') from system.dual;
Result: aa123456789
```

函数名	返回类型	描述
ltrim(string A)	String	去掉字符左侧的空格

说明

只去掉字符左侧的空格，返回字符类型。

举例

```
select ltrim(' bigdata ') from system.dual limit 1;
Result: 'bigdata'
```

函数名	返回类型	描述
ngrams(array<array<string>>, int N, int K, int pf)	Int	返回字符串中出现频率Top-K的N元语法词汇

说明

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)), 2, 100, 1000) FROM twitter;
```

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

说明

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') from system.dual limit 1;
Result: facebook.com
```

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串

说明

A可以是除了Boolean之外的任何基本类型，返回A的长度。

举例

```
SELECT printf("Hello World %d %s", 100, "days")FROM system.dual LIMIT 1;
Result: Hello World 100 days
```

函数名	返回类型	描述
regexp_extract(string subject, string pattern, int index)	String	返回指定部分的正则表达式

说明

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) from system.dual;
Result: foobar
```

函数名	返回类型	描述
regexp_replace(string INITIAL_STRING, string PATTERN, string REPLACEMENT)	Int	替换符合正则表达式的一部分

说明

把INITIAL_STRING中符合正则表达式PATTERN的部分替换为REPLACEMENT。

举例

```
SELECT regexp_replace('foothebarfoobar', 'foo(.?)(bar)', 'num') FROM system.dual LIMIT 1;
Result: numnum
```

函数名	返回类型	描述
repeat(string str, int n)	String	将str重复n次

说明

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) from system.dual;
Result: 1111111111
```

函数名	返回类型	描述
reverse(A)	String	把字符A转置

说明

返回转置的字符。

举例

```
select reverse('bigdata') from system.dual limit 1;
Result: atadgib
```

函数名	返回类型	描述
rpad(string str, int len, string pad)	String	用pad在右侧补全字符到指定长度

说明

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') from system.dual;
Result: 123456789aa
```

函数名	返回类型	描述
rtrim(string A)	String	去掉字符右侧的空格

说明

只去掉字符右侧的空格，返回字符类型。

举例

```
select rtrim(' bigdata ') from system.dual limit 1;
Result: ' bigdata'
```

函数名	返回类型	描述
sentences(str, lang, country)	Array<Array<String>>	把自然语言分成句子和词组

说明

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?') from system.dual limit 1;
Result: [[ "Hello", "there" ], [ "How", "are", "you" ]]
```

函数名	返回类型	描述
space(int n)	String	返回n个空格的字符

说明

返回n个空格的字符。

举例

```
select space(5) from system.dual limit 1;
Result:
```

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串

说明

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','[-]') from system.dual;
Result: ["pro","duct"]
```

函数名	返回类型	描述
str_to_map(text[, delimiter1, delimiter2])	Map	把字符转化为map

说明

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','=','-') from system.dual;
Result: {"a":"b","c":"d"}
```

函数名	返回类型	描述
substr(string A, int start [, length])	String	获取子字符串

说明

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) from system.dual limit 1;
Result: ta
```

函数名	返回类型	描述
substring(string A, int start [, length])	String	获取子字符串

说明

用法和 'substr' 一致

举例

```
select substring('bigdata',6,2) from system.dual limit 1;
Result: ta
```

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中多出来的字符会在output中被移除。

举例

```
select translate('abcdef', 'adc', '19') from system.dual limit 1;
Result: 1b9ef
```

函数名	返回类型	描述
trim(string s)	String	去掉字符左右的空格

说明

去掉字符左右的空格。

举例

```
select trim(' bigdata ') from system.dual limit 1;
Result: 'bigdata'
```

函数名	返回类型	描述
unbase64(string str)	Binary	把base64格式转化为binary

说明

把base64格式转化为binary。

举例

```
select unbase64(base64(binary('bigdata'))) from system.dual limit 1;
Result: 62696764617461
```

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

说明

把字符中的字母都转化为大写，其他字符不变。

举例

```
select upper('bigdata') from system.dual limit 1;
Result: BIGDATA
```

函数名	返回类型	描述
ucase(string A)	String	把字符中的字母都转化为大写

说明

用法和upper一致。

举例

```
select ucase('bigdata') from system.dual limit 1;
Result: BIGDATA
```

函数名	返回类型	描述
chr(A)	String	返回整型的char值

说明

chr(number_code): 只能传入int, 转为String.valueOf((char) number_code), 字符编码格式为内置的unicode。

举例

```
select chr(123) from system.dual;
Result: {
```

函数名	返回类型	描述
oracle_instr(string1, string2 [, start_position [, nth_appearance]])	String	返回子字符串第一次出现的下标

说明

新增四个参数，遵守Oracle的函数定义。

举例

```
SELECT oracle_instr('bigdata', 'data') FROM system.dual LIMIT 1;
Result: 4
```

函数名	返回类型	描述
to_number(value, format_mask)	Int	把string转化为number

说明

按照指定的格式来格式化字符。

举例

```
select to_number('1239.0') from system.dual limit 1;
Result: 1239
```

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') from system.dual limit 1;
Result: 2015-01-30 11:00:00
```

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') from system.dual;
Result: 20150130
```

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

说明

把全角字符转化为半角字符。

举例

```
select to_semiangle('bigdata') from system.dual limit 1;
Result: bigdata
```

函数名	返回类型	描述
nvl(eExpr1, eExpr2)	String	替换null为其他字符

说明

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') from system.dual limit 1;
Result: bigdata
```

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

说明

若string1为null， 返回value_if_not_null， 否则返回value_if_null。

举例

```
select nvl2(null,'trans','product') from system.dual limit 1;
Result: product
```

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), lnnvl(false), lnnvl(null) from system.dual limit 1;
Result: false true true
```

函数名	返回类型	描述
decode(value1, value2, value3, ..., defaultValue)	String	根据字符是否相等选择不同返回值

说明

返回当value1=value2的时候返回value3， 否则返回defaultValue。

举例

```
select decode(1,1,'trans','product') from system.dual;
Result: trans
```

函数名	返回类型	描述
range_decode(value, isLeftClose, isRightClose, left1, right1, value1, left2, right2, value2, defaultValue)	String	根据字符是否相等选择不同返回值

说明

返回当value1=value2的时候返回value3，否则返回defaultValue。

举例

```
SELECT range_decode('dep', false, true, "ACCOUNT", "HR", "NO-DEP") FROM system.dual LIMIT 1;
Result: NO-DEP
```

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') from system.dual limit 1;
Result: '9'
```

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。

```
SELECT instrb('Facebook', 'boo') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 5   |
+-----+
```

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。

```
SELECT elt(1, 'face', 'book') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| face |
+-----+
```

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_appearance>，则返回 <substr> 在 <str> 中第n次出现的位置。

```
SELECT oracle_instrb('ababab', 'ab') FROM system.dual LIMIT 1;
```

```
+-----+
| _c0 |
+-----+
| 1   |
+-----+
```

```
SELECT oracle_instrb('ababab', 'ab', 2) FROM system.dual LIMIT 1;
```

```
+-----+
| _c0 |
+-----+
| 3   |
+-----+
```

```
SELECT oracle_instrb('ababab', 'ab', 1, 3) FROM system.dual LIMIT 1;
```

```
+-----+
| _c0 |
+-----+
| 5   |
+-----+
```

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。

```
SELECT substrb('Facebook', 1) FROM system.dual LIMIT 1;
```

```
+-----+
| _c0 |
+-----+
| Facebook |
+-----+
```

```
SELECT substrb('Facebook', -1) FROM system.dual LIMIT 1;
```

```
+-----+
| _c0 |
+-----+
| k   |
+-----+
```

```
SELECT substrb('Facebook', 1, 2) FROM system.dual LIMIT 1;
```

```
+-----+
| _c0 |
+-----+
| Fa  |
+-----+
```

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。

```
SELECT md5("a") FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 0cc175b9c0f1b6a831c399e269772661 |
+-----+
```

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。

```
SELECT left('bigdata', 5) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| bigda |
+-----+
```

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。

```
SELECT right('bigdata', 4) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| data |
+-----+
```

函数	返回类型	描述
digits(<number>)	STRING	参数可以是任何数值类型。将参数转换成STRING返回。如果参数是整数, 会按类型将位数补足。

```
SELECT digits(4) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 0000000004 |
+-----+
```

函数	返回类型	描述
initcap(<str>)	STRING	需要一个STRING类型的参数。将参数中所有词的首字母大写, 其他字母小写。

```
SELECT initcap('bIgData rockS!') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| Bigdata Rocks! |
+-----+
```

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参数：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。

```
SELECT regex_substr('Road Guiping, Xuhui District, SH', ',[^,]+,') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| , Xuhui District, |
+-----+
```

函数	返回类型	描述
decode_char (<binary>, <str>)	见描述	需要两个参数：<binary> 是一串二进制字符，<str> 是编码方式。该函数用用 <str> 把 <binary> 转换为字符返回。

```
SELECT decode_char(binary('name'), 'UTF-8') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| name |
+-----+
```

6.10. XML函数

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;
+-----+
|      xmlforest      |
+-----+
| <name>Han Meimei</name><age>20</age> |
+-----+
```

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"/> |
+-----+
```

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()') FROM system.dual
LIMIT 1;
+-----+
| _c0 |
+-----+
| []   |
+-----+

SELECT xpath(' <a><b>b1</b><b>b2</b><b>b3</b><c>c1</c><c>c2</c></a>', 'a/b/text()') FROM system.dual
LIMIT 1;
+-----+
| _c0 |
+-----+
| ["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()') FROM system.dual
LIMIT 1;
+-----+
| _c0 |
+-----+
| ["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
system.dual ORDER BY values;
+-----+
| values |
+-----+
| []     |
| ["b1", "b2", "b3"] |
| ["c1", "c2"] |
+-----+
```

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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| <a>b</a> |
+-----+
```

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>') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| <id>36</id><name>Haines</name> |
+-----+
```

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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| cc |
+-----+
```

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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| true |
+-----+
```

```
SELECT xpath_boolean('<a><b>1</b></a>','a/b=0') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| false |
+-----+
```

函数	返回类型	描述
xpath_number (<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)') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 3.0 |
+-----+
```

函数	返回类型	描述
xpath_double (<xml>,<expr(xpath)>)	DOUBLE	和 xpath_number 同义。

用例见[\[xpath_number\]](#)。

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)') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 3.0 |
+-----+
```

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)') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 3   |
+-----+
```

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)') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 3   |
+-----+
```

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)') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 3   |
+-----+
```

6.11. 复杂类型函数

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。所指定的所有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) from system.dual limit 1;
Result: {"1":2,"3":4}
```

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) from system.dual limit 1;
Result: {"col1":1,"col2":2,"col3":3}
```

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') from system.dual limit 1;
Result: {"name1":100,"name2":"200"}
```

操作符	返回类型	描述
<code>array(val1, val2, ...)</code>	Array类型	创建数组

说明

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。

举例

```
select array(1,'1',1.1) from system.dual limit 1;
Result: ["1","1","1.1"]
```

操作符	返回类型	描述
<code>A[n]</code>	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] from system.dual ;
Result: 1
```

操作符	返回类型	描述
<code>M[key]</code>	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'] from system.dual ;
Result: 100
```

操作符	返回类型	描述
<code>S.x</code>	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。

举例

```
select named_struct('name1',100,'name2','200').name1 from system.dual limit 1;
Result: 100
```

操作符	返回类型	描述
size(Map<K, V>) ; size(Array<T>)	Int	返回元素数量

说明

返回数组或者map中的元素数量。

举例

```
select size(array('name','age','sex')) from system.dual;
Result: 3
```

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

说明

输入待排序的数组，输出排序之后的数组。默认顺序是按照从小到大排列。

举例

```
select sort_array(array(3,1,4,5,2)) from system.dual;
Result:[1,2,3,4,5]
```

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       |
+-----+-----+
```

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类型。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。

```
SELECT index(array(1,2,3),0) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 1   |
+-----+

SELECT index(absent_date,0) FROM employee_absence;
+-----+
| _c0 |
+-----+
| 2014-01-04 |
| 2014-02-14 |
+-----+
```

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类型。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。

```
SELECT array_contains(array(1, 2, 3), 2) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| true |
+-----+
```

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。

```
SELECT map_keys(map(1, 'a', 2, 'b')) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| [1,2] |
+-----+
```

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。

```
SELECT map_values(map(1, 'a', 2, 'b')) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| ["a","b"] |
+-----+
```

函数	返回类型	描述
create_union (<tag>, <obj> [, <obj>, …])	UNIONTYPE	需要至少两个参数。使用给定的 <tag> 和 <obj> 生成一个UNIONTYPE并返回。

```
SELECT create_union(0, 1) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| {0:1} |
+-----+
```

6.12. 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:

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;
Result:
name
age
sex
```

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

说明

生成表，每一个Struct作为一行。

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

```
select json_tuple('{"firstName":"Brett","lastName":"McLaughlin","email":"aaaa"}', 'firstName',
'email') from system.dual;
Result: Brett    aaaa
```

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

```
SELECT parse_url_tuple('http://www.transwarp.io/news/detail?id=38', 'HOST', 'PATH', 'QUERY',
'PROTOCOL', 'FILE') as (ip, ho, pa, pr, fi) from system.dual a;
Result:
www.transwarp.io /news/detail http /news/detail?id=38
```

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

```
select stack(2, array('name',1), array('age',2)) from system.dual;
Result:
["name","1"]
["age","2"]
```

6.12.1. LATERAL VIEW

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;
id char(2)
values array<int>

SELECT * FROM table_array;
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;
[Hive Error]: 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：

```
SELECT <selectexpr1>, <columnAlias1> [, <columnAlias2>, ...]
FROM baseTable ALTERAL VIEW udtf(expression) <tableAlias> ①
AS <columnAlias1> [, <columnAlias2>, ...] ②
```

① `tableAlias`指代存放UDTF结果的虚拟表的名称

② `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

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;
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;
ids array<char(2)>
values array<int>

SELECT * FROM table_array2;
[ "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_array
LATERAL VIEW explode(ids) a AS id
LATERAL VIEW explode(values) b AS 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。

6.13. 聚合函数

函数名	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) from somedata;
Result: 1314

```

函数名	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) from somedata;
Result: 1314

```

函数名	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) from somedata;
Result: 100000
```

函数名	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) from somedata;
Result: 100000
```

函数名	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) from somedata;
Result: 100000
```

函数名	返回类型	描述
variance(x)	Double	求方差

说明

返回某列所有值的方差。

举例

```
select variance(price) from somedata;
Result: 0.0
```

函数名	返回类型	描述
var_pop(x)	Double	求方差

说明

用法同 'variance' 一致。

举例

```
select var_pop(price) from somedata;
Result: 0.0
```

函数名	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) from somedata;
Result: 0.0
```

函数名	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) from somedata;
Result: 0.0
```

函数名	返回类型	描述
stddev(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) from somedata;
Result: 0.0
```

函数名	返回类型	描述
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_pop(price) from somedata;
Result: 0.0
```

函数名	返回类型	描述
stddev_samp(x)	Double	标准差

说明

返回指定列的样本标准差。

举例

```
select stddev_samp(price) from somedata;
Result: 7.45430868352036E-12
```

函数名	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， $(\text{SUM}(x*y) - \text{SUM}(x)*\text{SUM}(y) / \text{COUNT}(x, y)) / \text{COUNT}(x, y)$ ，否则返回null。

举例

```
select covar_pop(1,2) from somedata;
Result: 0.0
```

函数名	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) from somedata;
Result: 0.0
```

函数名	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) from somedata;
Result: NaN
```

函数名	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) from system.dual;
Result: 1.0
```

函数名	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')) from system.dual;
Result: [1.0]
```

函数名	返回类型	描述
percentile_approx(DOUBLE col, p [, B])	Double	数值区域的百分比数值点的近似值

说明

返回数值区域的百分比数值点的近似值。0 < P < 1, 否则返回NULL, 支持浮点型数值。参数B设置精确度, 越高的B值精确度越高, 缺省值为10,000。当列中distinct值的数目小于B时, 此时返回准确值。

举例

```
SELECT percentile_approx(val, 0.56, 100000) FROM somedata;
Result: 2.26
```

函数名	返回类型	描述
percentile_approx(col, array(p^1,, [, p,,2_]...)[, B])	Array	数值区域的百分比数值点的近似值

说明

与上个方法相似，除了接受与返回一组百分比数值点，而不是一个。

举例

```
SELECT percentile_approx(val, array(cast(0.5 as float), cast(0.95 as float), cast(0.98 as float)), 100000) FROM somedata;
Result: [0.05,1.64,2.26]
```

函数名	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) FROM somedata;
Result: [{"x":162.5,"y":8.0}, {"x":540.0,"y":5.0}, {"x":985.7142857142857,"y":7.0}]
```

函数名	返回类型	描述
collect_set(x)	Array	去重

说明

返回没有重复的记录。

举例

```
select collect_set(price) from somedata;
Result: 100000
```

6.14. 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。

6.14.1. 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;
A
A
B
C
D

SELECT col1, count(col1) OVER (ORDER BY col1 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM
row_range;
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) from
row_range;
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) FROM user_info;
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) FROM user_info;
宁**  1
李*   1
李**  1
潘**  1
管**  1
邱*   1
魏**  1
华*   1
祝**  1
马**  1

SELECT name, reg_date, acc_level, count(name) OVER (PARTITION BY acc_level ORDER BY reg_date) FROM user_info;
华* 20080214      B    1
宁** 20081031     D    1
李** 20110916     D    2
魏** 20091202     A    1
马** 20110101     A    2
邱*  20121024     A    3
祝** 20100101     C    1
潘** 20110430     C    2
管** 20141003     C    3
李*  20130702     E    1

SELECT name, acc_level, count(acc_level) OVER (PARTITION by acc_level) FROM user_info;
华*  B    1
李** D    2
宁** D    2
邱*  A    3
马** A    3
魏** A    3
潘** C    3
管** C    3
祝** C    3
李*  E    1

```

6.14.2. 窗口函数详解

下面我们列举所有Inceptor支持的窗口函数，包含：first_value, last_value, lead, lag, count, max, min, avg, sum, row_number, rank, dense_rank, percent_rank, cume_dist

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) FROM user_info;
华* 20080214 华*
宁** 20081031 华*
魏** 20091202 华*
祝** 20100101 华*
马** 20110101 华*
潘** 20110430 华*
李** 20110916 华*
邱* 20121024 华*
李* 20130702 华*
管** 20141003 华*
```

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) FROM user_info;
华* 20080214 管**
宁** 20081031 管**
魏** 20091202 管**
祝** 20100101 管**
马** 20110101 管**
潘** 20110430 管**
李** 20110916 管**
邱* 20121024 管**
李* 20130702 管**
管** 20141003 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) FROM user_info;
华* 20080214 华*
宁** 20081031 宁**
魏** 20091202 魏**
祝** 20100101 祝**
马** 20110101 马**
潘** 20110430 潘**
李** 20110916 李**
邱* 20121024 邱*
李* 20130702 李*
管** 20141003 管**
```

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) FROM user_info;
```

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) FROM user_info;
```

华*	魏**
宁**	祝**
魏**	马**
祝**	潘**
马**	李**
潘**	邱*
李**	李*
邱*	管**
李*	NULL
管**	NULL

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) FROM user_info;
```

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) FROM user_info;
```

华*	NULL
宁**	NULL
魏**	华*
祝**	宁**
马**	魏**
潘**	祝**
李**	马**
邱*	潘**
李*	李**
管**	邱*

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) FROM user_info;
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) FROM user_info;
华* B 1
宁** D 1
李** D 2
魏** A 1
马** A 2
邱* A 3
祝** C 1
潘** C 2
管** C 3
李* E 1
```

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) FROM user_info;
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
```

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) FROM user_info;
邱*   A      1
马**  A      1
魏**  A      1
华*   B      2
潘**  C      3
管**  C      3
祝**  C      3
李**  D      4
宁**  D      4
李*   E      5
```

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) FROM user_info;
邱*   0.3
马**  0.3
魏**  0.3
华*   0.4
潘**  0.7
管**  0.7
祝**  0.7
李**  0.9
宁**  0.9
李*   1.0
```

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”。

6.15. Context函数

从Inceptor4.3开始，您可以在Inceptor2中查看当前session的上下文（context）信息。目前Inceptor提供查看当前用户的身份（current_user）和当前用户是否拥有某个角色（has_role）。

下面我们详细介绍各个context函数的用法。

6.15.1. 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() FROM user_info LIMIT 1;
+-----+
| _c0 |
+-----+
| 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;
Error: Error while processing statement: FAILED: Hive Internal Error:
org.apache.hadoop.hive.ql.security.authorization.plugin.HiveAccessControlException(Permission
denied: Principal [name=alice, type=USER] does not have following privileges for operation QUERY
[[SELECT] on Object [type=TABLE_OR_VIEW, name=default.zzz]]) (state=, code=12)
```

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 |
+-----+-----+
```

6.15.2. 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('<role_name>') FROM <table_name> LIMIT 1;
```

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 user_account LIMIT 1;
+-----+
| _c0 |
+-----+
| false |
+-----+
```

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 system.dual LIMIT 1;
Error: Error while processing statement: FAILED: Hive Internal Error:
org.apache.hadoop.hive.ql.security.authorization.plugin.HiveAccessControlException(Permission
denied: Principal [name=alice, type=USER] does not have following privileges for operation QUERY
[[SELECT] on Object [type=TABLE_OR_VIEW, name=default.zzz]]) (state=,code=12)
```

例3

查看用户在当前session中是否有accounting角色：

```
SELECT has_role('accounting') FROM user_account LIMIT 1;
+-----+
| _c0 |
+-----+
| true |
+-----+
```



has_role在HiveServer1和简单模式的HiveServer2下返回一定为true。

6.15.3. current_database

返回用户当前所在数据库。

例

```
SELECT current_database() FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| default |
+-----+
```

6.15.4. current_time

返回当前系统时间。和 `systime` 同义。

例

```
SELECT current_time() FROM system.dual LIMIT 1;
```

6.15.5. current_date

返回当前系统日期和时间。

例

```
SELECT current_date() FROM system.dual LIMIT 1;
```

6.15.6. current_timestamp

返回当前系统TIMESTAMP。和 `systimestamp` 同义。

例

```
SELECT current_timestamp() FROM system.dual limit 1;
```

6.16. 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](#)。

表 50. MASK

函数名	返回类型	描述
<code>MASK(<data_string>[, <mask_string>], <pre_length>[, <post_length>])</code>	STRING	用于隐藏电话号码。使用 <code>mask_string</code> 替代 <code>data_string</code> 中的字符。保留前面 <code>pre_length</code> 长度的字符；可选择保留尾部 <code>post_length</code> 长度的字符。默认的 <code>mask_string</code> 为 <code>*</code> ，默认的 <code>post_length</code> 值为 0。

例 275. MASK

```
SELECT mask('1234567', '*', 2, 3) FROM system.dual LIMIT 1;
'12**567'
SELECT mask('1234567', 2, 3) FROM system.dual LIMIT 1;
'12**567'
SELECT mask('1234567', '*', 2) FROM system.dual LIMIT 1;
'12*****'
SELECT mask('1234567', 2) FROM system.dual LIMIT 1;
'12*****'
```

表 51. 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 <code>mask_string</code> 替代 <code>data_string</code> 中 <code>@</code> 符号之前的字符， <code>@</code> 后面的域名不会被隐藏。可选择保留前面 <code>pre_length</code> 长度的字符，可选择保留尾部（ <code>@</code> 之前） <code>post_length</code> 长度的字符。默认的 <code>mask_string</code> 为 <code>*</code> ，默认的 <code>pre_length</code> 值为 1，默认的 <code>post_length</code> 值为 0。

例 276. MASK_EMAIL

```
SELECT mask_email('test@sample.io', '*', 1, 0) FROM system.dual LIMIT 1;
't***@sample.io'
SELECT mask_email('test@sample.io') FROM system.dual LIMIT 1;
't***@sample.io'
```

表 52. 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 <code>mask_string</code> 替代 <code>data_string</code> 中的字符，只保留“有限公司”及其之后的所有字符。默认的 <code>mask_string</code> 为 <code>*</code> 。

例 277. MASK_COMPANY

```
SELECT mask_company('虚构的有限公司') FROM system.dual LIMIT 1;
'***有限公司'
SELECT mask_company('中科院', '#') FROM system.dual LIMIT 1;
'###'
```

表 53. CAESAR_CIPHER

函数名	返回类型	描述
CAESAR_CIPHER(<data_string>[, <shift_value>])	STRING	右移 <code>data_string</code> 中的数字和字母字符（只有数字和字母会被右移），可以使用 <code>shift_value</code> 设置偏移量。如“abc123”中的数字和字母右移1位得到“bcd234”。默认的 <code>shift_value</code> 为3。

例 278. CAESAR_CIPHER

```
SELECT caesar_cipher('012abc6789') FROM system.dual LIMIT 1;
'345def9012'
SELECT caesar_cipher('012ABC6789', 1) FROM system.dual LIMIT 1;
'123BCD7890'
```

6.17. 其他函数和运算符

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, '') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| NULL |
+-----+
SELECT oracle_coalesce(NULL, '', 1) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 1   |
+-----+
```

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 <default>。

```
SELECT oracle_decode(0, 0, 'zero', 1, 'one', 'others') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| zero |
+-----+
SELECT oracle_decode(1, 0, 'zero', 1, 'one', 'others') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| one  |
+-----+
SELECT oracle_decode(2, 0, 'zero', 1, 'one', 'others') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| others |
+-----+
```

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) FROM system.dual LIMIT 1;
+-----+
| _c0      |
+-----+
| 370802199402210029 |
+-----+
```

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) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 0   |
+-----+
```

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) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| null |
+-----+
```

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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 2   |
+-----+
```

```
SELECT field('d', 'b', 'a', 'c', 'h') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 0   |
+-----+
```

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 需要至少两个参数: 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) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 3   |
+-----+
```

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。

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) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| false |
+-----+
```

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') FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| 128 |
+-----+
```

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;
Error: Error while processing statement: FAILED: Execution Error, return code 1 from
io.transwarp.inceptor.execution.SparkTask. Job aborted due to stage failure: Task 0 in stage 11.0
failed 4 times, most recent failure: Lost task 0.3 in stage 11.0 (TID 29, tw-node2107):
java.lang.RuntimeException: org.apache.hadoop.hive.ql.metadata.HiveException:
org.apache.hadoop.hive.ql.metadata.HiveException: ASSERT_TRUE(): assertion failed. (state=08S01,code=1)
```

```
SELECT assert_true(0<=0) FROM system.dual LIMIT 1;
+-----+
| _c0 |
+-----+
| NULL |
+-----+
```

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() FROM system.dual;
```

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) FROM inventory GROUP BY ROLLUP(type) ORDER BY type;
```

type	_c1
All Fruits	5000.0
apple	2500.0
orange	400.0
pear	1600.0
pple	500.0

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) FROM inventory GROUP BY ROLLUP(type, store, grade) ORDER BY gt;
```

gt	gts	gtsg	_c3
0	0	0	300.0
0	0	1	500.0
0	1	3	2500.0
0	1	3	400.0
0	1	3	1600.0
0	0	1	700.0
0	0	1	800.0
0	0	1	400.0
0	1	3	500.0
0	0	0	500.0
0	0	0	700.0
0	0	0	100.0
0	0	0	800.0
0	0	0	500.0
0	0	1	200.0
0	0	0	500.0
0	0	0	200.0
0	0	0	400.0
0	0	1	900.0
0	0	0	1000.0
0	0	1	1500.0
1	3	7	5000.0

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_close>` 和 `<is_right_close>` 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>`。

```
SELECT range_decode(8.8,true,true,1,9,'product',10,20,'Rocks','productinitials') FROM system.dual
LIMIT 1;
+-----+
| _c0   |
+-----+
| product |
+-----+
```

7. Inceptor多租户手册

7.1. Inceptor权限和角色管理

本章我们介绍如何在Beeline命令行中使用基于SQL的授权（SQL-based Authorization）语法对Inceptor的用户和角色进行管理和授权。

7.1.1. Inceptor对用户身份的判断

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Inceptor以LDAP认证， Inceptor通过登陆者的LDAP用户名来判断登录者的身份。

7.1.2. Inceptor管理员身份的获取

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> 出提供您想要连接的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 |
+-----+
```

这样，您便获取了管理员角色。

下面我们介绍Inceptor中的权限管理操作。

7.1.3. Inceptor角色管理

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。

7.1.3.1. 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 |
| r1    |
+-----+
```

7.1.3.2. DROP ROLE： 删除角色

语法

```
DROP ROLE role_name;
```

举例

```
DROP ROLE r1;
SHOW ROLES;
+-----+
| role |
+-----+
| ADMIN |
| PUBLIC |
+-----+
```

7.1.3.3. 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：hive用户查看当前自己当前所有的角色，因为hive目前有的角色

```
SHOW CURRENT ROLES;
+-----+
| role |
+-----+
| PUBLIC |
+-----+
```

7.1.3.4. SET ROLE: 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    |
| r2    |
+-----+
```

7.1.3.5. SHOW ROLES: 查看所有Inceptor中的角色

语法

```
SHOW ROLES;
```

举例

```
SHOW ROLES;
+-----+
| role |
+-----+
| ADMIN |
| PUBLIC |
+-----+
```

7.1.3.6. GRANT ROLE: 授予角色

语法

```

GRANT role_name [, role_name] ...
TO principal_specification [, principal_specification] ...
[ WITH ADMIN OPTION ];

principal_specification
: USER user
| ROLE role

```

将角色授予给用户或角色。角色的接受者可以是用户，也可以是别的角色。如果加上了WITH ADMIN OPTION选项，则角色的接受者也有权将这个角色授予别的用户或者角色。

PUBLIC和ADMIN角色不能被GRANT给用户或者角色，也不能接受角色的授予。也就是说以下语法不成立：



```

GRANT (ADMIN|PUBLIC) TO USER (user_name|role_name);
GRANT role_name to ROLE (ADMIN|PUBLIC);

```

例1

```

GRANT r1 TO USER user1;
GRANT ADMIN TO USER user1;
Error: Error while processing statement: FAILED: Execution Error, return code 1 from
org.apache.hadoop.hive.ql.exec.DDLTask. Error granting roles for user1 to role admin: null (state=
08S01,code=1) ①

```

① GRANT ADMIN会报错。

例2

```

GRANT r1 TO USER user1 WITH ADMIN OPTION;
SHOW ROLE GRANT USER user1;
+-----+-----+-----+-----+
| role | grant_option | grant_time | grantor |
+-----+-----+-----+-----+
| PUBLIC | false | 0 | |
| r1 | true | 1448049579000 | hive |
+-----+-----+-----+-----+

```

7.1.3.7. REVOKE ROLE: 收回角色

语法

```

REVOKE [ADMIN OPTION FOR] 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。加上[ADMIN OPTION FOR]选项则只收回授予角色的权限，但是并不收回角色。

举例

```
SHOW ROLE GRANT USER user1;
+-----+-----+-----+-----+
| role | grant_option | grant_time | grantor |
+-----+-----+-----+-----+
| PUBLIC | false | 0 | hive |
| r1 | true | 1448049579000 | hive |
+-----+-----+-----+-----+
```

REVOKE ADMIN OPTION FOR r1 FROM USER user1; ①

```
SHOW ROLE GRANT USER user1;
+-----+-----+-----+-----+
| role | grant_option | grant_time | grantor |
+-----+-----+-----+-----+
| PUBLIC | false | 0 | hive |
| r1 | false | 1448049579000 | hive |
+-----+-----+-----+-----+
```

REVOKE r1 FROM USER user1; ③

```
SHOW ROLE GRANT USER user1;
+-----+-----+-----+-----+
| role | grant_option | grant_time | grantor |
+-----+-----+-----+-----+
| PUBLIC | false | 0 | ④ |
+-----+-----+-----+-----+
```

① 只收回user1授予别的用户或角色r1的权限，但是不从user1收回r1角色。

② 我们看到user1还有r1角色，但是没有grant_option。

③ 从user1收回r1角色。

④ user1不再有r1角色。

7.1.3.8. 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 | hive |
| r1 | false | 1448104461000 | hive |
| r2 | false | 1448104461000 | hive |
| r3 | false | 1448104461000 | 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       | 1448104356000 | hive    |
| r3   | false       | 1448104356000 | hive    |
+-----+-----+-----+-----+
```

举例：

因为我们不能将角色GRANT给ADMIN和PUBLIC，所以任何时候ADMIN和PUBLIC都不会拥有其他角色。

```
SHOW ROLE GRANT ROLE ADMIN;
+-----+-----+-----+-----+
| role | grant_option | grant_time | grantor |
+-----+-----+-----+-----+
SHOW ROLE GRANT ROLE PUBLIC;
+-----+-----+-----+-----+
| role | grant_option | grant_time | grantor |
+-----+-----+-----+-----+
```

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

语法

```
SHOW PRINCIPALS role_name;
```

举例

```
0: jdbc:hive2://localhost:10000/default> SHOW PRINCIPALS r1;
+-----+-----+-----+-----+-----+-----+
| principal_name | principal_type | grant_option | grantor | grantor_type | grant_time |
+-----+-----+-----+-----+-----+-----+
| user1          | USER           | false        | hive    | USER         | 1448104461000 |
| usera          | USER           | false        | hive    | USER         | 1448047478000 |
+-----+-----+-----+-----+-----+-----+
2 rows selected (0.052 seconds)
```



在Inceptor中，我们无法用SHOW PRINCIPALS查看拥有ADMIN和PUBLIC的用户和角色。

7.1.4. Inceptor权限管理

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VIEW，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>] ...
[WITH GRANT OPTION] ①
;

-- 收回权限
REVOKE [GRANT OPTION FOR] ②
<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>
```

- ① 如果在授权时加上 **WITH GRANT OPTION** 选项，那么被授权的用户和角色也可以将该权限授予别的用户或角色。
- ② 如果加上 **GRANT OPTION FOR** 选项，那么只收回用户或角色给别的用户或角色授权的权力，但是并不收回权限本身。

下面详细介绍权限管理操作。

7.1.4.1. GRANT：权限的授予

7.1.4.1.1. 授予权限

语法：授予全局 CREATE 权限

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

说明：该语句授予用户/角色创建DATABASE的权限

例 279. 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>, ...]
[WITH GRANT OPTION];
```

① 注意，` ` 表示全局所有表。

例 280. 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;
```

7.1.4.1.2. 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> ...]
[WITH GRANT OPTION];
```

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 权限就是查看该数据库中表的表名的权限（在该数据库中执行SHOW TABLES）。在未授权的情况下，任何用户或角色都只有查看default数据库中的表的权限。

例 281. 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> ...]
[WITH GRANT OPTION];

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

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

7.1.4.1.3. 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> ...]
  [WITH GRANT OPTION];
```

例 283. 向用户alice和角色r1授予表ta的SELECT权限并给予授权权限(WITH GRANT OPTION)

```
GRANT SELECT ON ta TO USER alice, ROLE r1 WITH GRANT OPTION;
```

例 284. 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;
```

7.1.4.2. REVOKE：权限的收回

7.1.4.2.1. 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 285. 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 286. 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;
```

7.1.4.2.2. 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 287. 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 288. 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 DATABASE test;
REVOKE SELECT ON * FROM USER alice;
```

7.1.4.2.3. 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 289. 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 290. 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;
```

7.1.4.3. 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 291. 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
db	*			bob	USER	SELECT	false	1470835273000	hive

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	1470834823000	hive
db	*			bob	USER	SELECT	false	1470835273000	hive

例 292. 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	ROLE	ALL	true	14706228837000	ADMIN
				alice	USER	CREATE	false	14708334829000	hive
				gerenxinxi	USER	CREATE	false	1470808031000	hive
				r1	USER	CREATE	false	1470832925000	hive
				r1	USER	SELECT	false	1470833053000	hive

7.1.5. Inceptor权限传播控制

Inceptor中，TABLE和VIEW的owner对TABLE或VIEW有所有权限，并自动拥有这些权限的 grant option（将权限赋予别人的能力）：

0: jdbc:hive2://localhost:10000/default> CREATE TABLE alice_tb (col1 string, col2 int); No rows affected (1.243 seconds)									
0: jdbc:hive2://localhost:10000/default> SHOW GRANT ON alice_tb;									
database	table	partition	column	principal_name	principal_type	privilege	grant_option	grant_time	grantor
default	alice_tb			alice	USER	DELETE	true	1470895057000	alice
default	alice_tb			alice	USER	INSERT	true	1470895057000	alice
default	alice_tb			alice	USER	SELECT	true	1470895057000	alice
default	alice_tb			alice	USER	UPDATE	true	1470895057000	alice

这样的能力会导致权限的不当传播。例如，管理员计划让用户alice可以读取有敏感信息的表bank_account，于是给用户alice授予了bank_account的 **SELECT** 权限。用户alice进行如下操作就可以间接地将bank_account中的数据暴露给用户bob：

```
CREATE TABLE bank_account_copy AS SELECT * FROM bank_account; ①  
GRANT SELECT ON bank_account_copy TO USER bob; ②
```

- ① alice将表bank_account中的数据插入自己建的表bank_account_copy中。
- ② 作为bank_account_copy的owner，alice可以将它的权限赋予给任何用户或角色。

这样，bank_account上的 **SELECT** 权限便不当地传播给了用户bob。

为了避免这样的权限传播，使管理员可以更加集中地管理权限，Inceptor从TDH4.6开始，提供TABLE/VIEW owner的默认grant option限制。进行适当的限制后，owner将不再自动拥有TABLE/VIEW的所有权限的grant option，也就无法将权限传播给别人。



权限传播控制 仅影响owner权限的grant option，不影响权限本身。Owner依旧自动拥有自己的TABLE/VIEW的 **SELECT, UPDATE, INSERT, DELETE** 权限。

7.1.5.1. 设置

进行权限传播控制需要管理员修改参数 `hive.security.authorization.createtable.owner.grants` 的值，它的值是如下选项的组合，用逗号隔开，不要有空格：

权限类型	含义
SELECT_NOGRANT	SELECT without grant option

权限类型	含义
SELECT_WGRANT	SELECT with grant option
INSERT_NOGRANT	INSERT without grant option
INSERT_WGRANT	INSERT with grant option
UPDATE_NOGRANT	UPDATE without grant option
UPDATE_WGRANT	UPDATE with grant option
DELETE_NOGRANT	DELETE without grant option
DELETE_WGRANT	DELETE with grant option

`hive.security.authorization.createable.owner.grants` 默认为空，表示*`SELECT_WGRANT, INSERT_WGRANT, UPDATE_WGRANT, DELETE_WGRANT*`，也就是owner拥有所有的权限的grant option。当您进行设置时，如果只写出部分权限的grant option，容易引起歧义，例如：`DELETE_WGRANT,UPDATE_WGRANT` —— owner的 `SELECT` 和 `INSERT` 权限是否有grant option不明确。所以我们建议将owner对四个权限是否有grant option全部写出。

例 293. 限制owner的 SELECT 权限传播

要限制owner的 `SELECT` grant option但不限制其他权限的grant option，我们在需要将 `hive.security.authorization.createtable.owner.grants` 的值设为 `SELECT_NOGRANT,INSERT_WGRANT,UPDATE_WGRANT,DELETE_WGRANT`。具体步骤如下：

1. 登陆集群管理界面，进入Inceptor的配置页面，找到该参数（可以通过搜索框搜索）：

The screenshot shows the Inceptor configuration interface. The top bar indicates 'Inceptor1 | HEALTHY'. The main area has a search bar containing 'owner.grants'. A table lists a single configuration item:

配置项	值	描述
<code>hive.security.authorization.createtable.owner.grants</code>	empty	the privileges automatically granted to the owner whenever a table gets created. An example like 'select,drop' will grant select and drop privilege to the owner of the table

2. 点击参数值，将其设置为

`SELECT_NOGRANT,INSERT_WGRANT,UPDATE_WGRANT,DELETE_WGRANT`:

The screenshot shows the Inceptor configuration interface with a red box highlighting the value 'SELECT_NOGRANT' in the 'Value' column of the configuration table. A callout arrow points from the text '输入参数值' (Input parameter value) to this highlighted field.

注意，这里的参数值一定要拼写正确，如果拼错，下一步Inceptor将无法重启。

3. 点击“保存更改”，然后点击右上角的“更多操作” > “配置服务”，将对参数的更改配置到集群的每个节点上。最后重启Inceptor服务，设置完成。

要查看设置效果，我们以用户alice的身份登陆，创建一张表，然后查看这张表的权限：

```
0: jdbc:hive2://localhost:10000/default> CREATE TABLE alice_t3 (col1 STRING);
No rows affected (8.177 seconds)
0: jdbc:hive2://localhost:10000/default> SHOW GRANT ON alice_t3;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| database | table | partition | column | principal_name | principal_type | privilege | grant_option | grant_time | grantor |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| default | alice_t3 |          |        | alice          | USER         | DELETE    | true       | 1470900451000 | alice      |
| default | alice_t3 |          |        | alice          | USER         | INSERT    | true       | 1470900451000 | alice      |
| default | alice_t3 |          |        | alice          | USER         | SELECT    | false      | 1470900451000 | alice      |
| default | alice_t3 |          |        | alice          | USER         | UPDATE    | true       | 1470900451000 | alice      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

可以看到，用户alice没有这张表的 `SELECT` 权限的grant option，但是有其他权限的grant option。

7.1.6. Inceptor操作所需权限总结

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)					
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					

操作	CREATE	SELECT	INSERT	UPDATE	DELETE	Ownership	ADMIN
CREATE DATABASE	Y (global)						
DROP DATABASE						Y	
CREATE DATABASE	Y (global)						
CREATE FUNCTION							Y
DROP FUNCTION							Y
ALTER DATABASE							Y

7.2. Inceptor中的行级和列级权限

在数据敏感行业，用户希望有更细粒度的权限控制。如银行数据仓库，操作人员可能属于不同的支行，他们只应该看到其所属支行的数据。比较传统的做法是，将一张大表按支行切分成多个小表或者创建多个VIEW，通过对表或者VIEW的表级权限（PRIVILEGE）管理来实现这个功能。但是这种做法在表或者VIEW的管理上相当繁琐。在Inceptor中，管理员或者表owner可以通过行级权限SQL来设置访问规则。同样在银行数据仓库，操作人员在查询带有敏感信息的表时，需要对敏感字段进行[脱敏显示](#)。例如管理员可以查看到表中电话号码所有的位数，而非管理员看到的电话号码会有多位进行了隐藏。在Inceptor中，管理员或者表的owner可以通过列级权限SQL来设置访问规则，使得同一列对不同权限的用户可以展示不同的信息。

本章将介绍如何使用Inceptor中的行级和列级权限设置。



行级和列级权限设置操作要求执行者有 **ADMIN** 角色或者为表的Owner。

7.2.1. 列级和行级权限设置操作一览

Inceptor中的行级和理解权限的相关操作包括：

- 为表设置行级权限：

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

这里 `<where_clause>` 为一个 **WHERE** 过滤子句。

- 取消对表的行级权限设置

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

- 为表设置列级权限：

```
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。

- 取消某张表某个列的列级权限设置

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

- 取消某张表所有列的列级权限设置

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

- 查看某张表的行级和列级权限设置

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

7.2.2. 行级和列级权限（PERMISSION）对比表级权限（PRIVILEGE）

表级别的权限称为PRIVILEGE，它们包括 **SELECT**, **INSERT**, **UPDATE** 和 **DELETE**。行级和列级权限则是对一些表级权限做出的 限制：

- 行级权限限制用户的表级 **SELECT**, **UPDATE** 和 **DELETE** 权限只能对部分行生效；
- 列级权限限制用户的表级 **SELECT** 权限只能查看列的部分信息。
- INSERT** 不受行级或列级权限限制，用户只需要对表有**INSERT**权限，便可以对表进行插入操作。另外，行级权限没有“行owner”的概念，和“表的创建用户自动对表有所有权限”不同，用户并不自动对自己插入的记录有**SELECT**, **UPDATE**和**DELETE**权限。

所以，用户要首先对表有表级权限，对用户的行级和列级权限设置才有意义。举个例子，如果用户alice对表a没有表级别的 **SELECT** 权限，那么无论给她设置什么样的行级权限，她都无法对表a进行 **SELECT** 操作。

7.2.3. 行级权限的设置

语法

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

说明

- 〈where_clause〉 为一个where字句。

下面我们介绍一些行级权限的常见用法。

7.2.3.1. 用法1：将权限直接设置给用户

设置行级权限最简单的方法是在表中建一个用户列（user），将每行记录的user值设为对该行记录有权限的用户。结合context函数（参考“Inceptor函数和运算符手册”中的“Context函数”章节）可以实现只有当前用户（current_user）和user值匹配时才能对记录进行操作。

在这个例子中我们用一张表finances，这张表中的记录如下：

```
SELECT * FROM finances_orc;
+-----+-----+-----+
| eid | funds | user |
+-----+-----+-----+
| 001 | 500.0 | alice |
| 002 | 500.0 | bob   |
| 003 | 1000.0 | alice |
| 004 | 2000.0 | carol |
+-----+-----+-----+
```

管理员或者表owner执行下面操作，将行级权限设置为“当前用户只对user值匹配其用户名的记录有权限，ADMIN角色对所有记录有权限”：

```
GRANT PERMISSION ON TABLE finances_orc FOR ROWS WHERE user = current_user() OR has_role('admin');
```

管理员或表owner现在可以查看finances的行级权限是否开启，以及开启的definition:

```
SHOW PERMISSION ON TABLE finances_orc;
+-----+-----+-----+
| type | enabled | definition |
+-----+-----+-----+
| Row Level Security | true | WHERE user = current_user() OR has_role('admin') |
+-----+-----+-----+
```

注意，经过行级权限设置后，如果一个用户已经对表有表级SELECT/UPDATE/DELETE权限，该用户的权限将被限制到表中user值和用户名匹配的记录上，但是用户如果对表没有表级权限SELECT/UPDATE/DELETE，GRANT PERMISSIONS并不赋予用户新的权限。我们以用户alice为例来演示这一点：

1. alice登陆Inceptor并查看当前用户

```
SELECT current_user() FROM dummy LIMIT 1;
+-----+
| _c0 |
+-----+
| alice |
+-----+
```

2. alice执行下面指令查看自己对表finances_orc的表级权限：

```
SHOW GRANT USER alice ON finances_orc;
```

我们分别看看几个不同场景下alice都能进行哪些操作。

- 示例场景1：如果她看到下面输出，说明她对finances_orc有 表级别的 SELECT, INSERT, UPDATE, DELETE权限：

database	table	partition	column	principal_name	principal_type	privilege	grant_option	grant_time	grantor
default	finances_orc			alice	USER	DELETE	false	1452530344000	hive
default	finances_orc			alice	USER	INSERT	false	1452530344000	hive
default	finances_orc			alice	USER	SELECT	false	1452530344000	hive
default	finances_orc			alice	USER	UPDATE	false	1452530344000	hive

那么当alice执行下面指令查看表中记录时，她能且只能看到user值为alice的两条记录：

```
SELECT * FROM finances_orc;
+-----+-----+-----+
| eid | funds | user |
+-----+-----+-----+
| 001 | 500.0 | alice |
| 003 | 1000.0 | alice |
+-----+-----+-----+
```

alice如果要执行UPDATE或者DELETE操作，她的操作也只会对这两行生效。比如：alice可以UPDATE她能看到的两条记录中的任意一条：

```
SELECT * FROM finances_orc;
+-----+-----+-----+
| eid | funds | user |
+-----+-----+-----+
| 001 | 500.0 | alice |
| 003 | 1000.0 | alice |
+-----+-----+
2 rows selected (3.197 seconds)
UPDATE finances_orc SET funds = 800 WHERE eid = "001";
+-----+
| _c0 |
+-----+
| 1 rows updated. |
+-----+
1 row selected (4.581 seconds)
SELECT * FROM finances_orc;
+-----+-----+-----+
| eid | funds | user |
+-----+-----+-----+
| 001 | 800 | alice |
| 003 | 1000.0 | alice |
+-----+-----+
2 rows selected (3.299 seconds)
```

alice也可以DELETE她能看到的两条记录中的任意一条：

```
SELECT * FROM finances_orc;
+-----+-----+-----+
| eid | funds | user |
+-----+-----+-----+
| 001 | 800 | alice |
| 003 | 1000.0 | alice |
+-----+-----+
2 rows selected (3.299 seconds)
DELETE FROM finances_orc WHERE eid = "001";
+-----+
| _c0 |
+-----+
| 1 rows deleted. |
+-----+
1 row selected (6.363 seconds)
SELECT * FROM finances_orc;
+-----+-----+-----+
| eid | funds | user |
+-----+-----+-----+
| 003 | 1000.0 | alice |
+-----+-----+
1 row selected (3.287 seconds)
```

当然，alice也可以向表中INSERT数据；INSERT权限不受行级权限影响。

```
INSERT INTO TABLE finances_orc VALUES ("001",800.0,"alice");
+-----+
| _c0 |
+-----+
| 1 rows affected. |
+-----+
1 row selected (1.568 seconds)
SELECT * FROM finances_orc;
+-----+-----+-----+
| eid | funds | user |
+-----+-----+-----+
| 003 | 1000.0 | alice |
| 001 | 800.0 | alice |
+-----+-----+
```

- 示例场景2：如果她看到下面输出，说明她对finances_orc没有任何 表级别的 权限：

```
0: jdbc:hive2://localhost:10000/default> SHOW GRANT USER alice ON finances_orc;
+-----+-----+-----+-----+-----+-----+-----+-----+
| database | table | partition | column | principal_name | principal_type | privilege | grant_option | grant_time | grantor |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
```

那么当alice执行下面指令查看表中记录时，Inceptor会报错：

```
SELECT * FROM finances_orc;
Error: Error while processing statement: FAILED: Hive Internal Error:
org.apache.hadoop.hive.ql.security.authorization.plugin.HiveAccessControlException(Permission
denied: Principal [name=alice, type=USER] does not have following privileges for operation
QUERY [[SELECT] on Object [type=TABLE_OR_VIEW, name=default.finances_orc]]) (state=,code=12)
```

alice尝试对表进行UPDATE, DELETE, INSERT也会导致Inceptor报类似的错。

- 示例场景3：如果她看到下面输出，说明她对finances_orc有 表级别的 SELECT和UPDATE权限，

```
0: jdbc:hive2://localhost:10000/default> SHOW GRANT USER alice ON finances_orc;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| database | table | partition | column | principal_name | principal_type | privilege | grant_option | grant_time | grantor |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| default | finances_orc |          |        | alice           | USER          | SELECT      | false       | 1452537639000 | hive        |
| default | finances_orc |          |        | alice           | USER          | UPDATE      | false       | 1452537639000 | hive        |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

那么alice可以查看表中的记录，也可以UPDATE她能看到的两条记录：

```
SELECT * FROM finances_orc;
+-----+-----+-----+
| eid | funds | user |
+-----+-----+-----+
| 003 | 1000.0 | alice |
| 001 | 800.0 | alice |
+-----+-----+-----+
2 rows selected (3.152 seconds)
UPDATE finances_orc SET funds = 5000.0 WHERE eid = "003";
+-----+
| _c0 |
+-----+
| 1 rows updated. |
+-----+
1 row selected (3.941 seconds)
SELECT * FROM finances_orc;
+-----+-----+-----+
| eid | funds | user |
+-----+-----+-----+
| 003 | 5000.0 | alice |
| 001 | 800.0 | alice |
+-----+-----+-----+
2 rows selected (3.398 seconds)
```

但是当她尝试删除表中记录时，Inceptor会报错：

```
DELETE FROM finances_orc WHERE eid = '003';
Error: Error while processing statement: FAILED: Hive Internal Error:
org.apache.hadoop.hive.ql.security.authorization.plugin.HiveAccessControlException(Permission
denied: Principal [name=alice, type=USER] does not have following privileges for operation
QUERY [[DELETE] on Object [type=TABLE_OR_VIEW, name=default.finances_orc]]) (state=,code=12)
```

7.2.3.2. 用法2：使用一张辅助表设置行级权限

如果我们想要按用户所在支行对finances进行行级授权（A支行的用户只能看见A支行的数据），一般的做法是在表中建一个支行列（branch），将每行记录的branch值设为对这行记录有权限的用户所在的支行；然后利用一张辅助表来记录用户和他所在的支行；在进行行级权限设置时，利用这张辅助表来过滤出有权限的用户。实例如下：

我们建一张新表finances_new，表中有一个名为branch的列：

```
SELECT * FROM finances_new;
+-----+-----+-----+
| eid | funds | branch |
+-----+-----+-----+
| 001 | 500.0 | A
| 002 | 500.0 | A
| 003 | 1000.0 | A
| 004 | 2000.0 | B
+-----+-----+-----+
```

以及一张辅助表branch_assignment，表中记录了用户所在的支行：

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

管理员或者finances_new表的owner可以用下面语句将finances_new的行级权限设置为“当前用户只对branch值对应其所在支行的记录有权限，管理员对所有记录有权限”：

```
GRANT PERMISSION ON TABLE finances_new FOR ROWS WHERE branch = (SELECT branch FROM branch_assignment WHERE user=current_user() or has_role('ADMIN'));
```



因为在行级权限中需要读branch_assignment表，管理员或者branch_assignment表的owner必须保证branch_assignment表中的用户都对branch_assignment表有SELECT权限，否则这些用户对表finances_new进行的操作将无法进行。

设置后用SHOW PERMISSION查看finances_new的行级权限能查看行级权限的设置定义：

```
0: jdbc:hive2://localhost:10000/default> SHOW PERMISSION ON finances_new;
+-----+-----+-----+-----+
| type | enabled | definition |
+-----+-----+-----+
| Row Level Security | true | WHERE branch = (SELECT branch FROM branch_assignment WHERE user=current_user() or has_role('ADMIN')) |
+-----+-----+-----+
```

现在假设alice, bob和carol三个用户都已经有 表级别 的SELECT权限。

```
0: jdbc:hive2://localhost:10000/default> SHOW GRANT ON finances_new;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| database | table | partition | column | principal_name | principal_type | privilege | grant_option | grant_time | grantor |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| default  | finances_new |          |          | alice        | USER       | SELECT    | false      | 1452781718000 | hive
| default  | finances_new |          |          | bob         | USER       | SELECT    | false      | 1452781067000 | hive
| default  | finances_new |          |          | carol       | USER       | SELECT    | false      | 1452781067000 | hive
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

我们分别以alice, bob和carol的身份连接Inceptor，查看finances_new中的记录。

- 以alice身份登陆：

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

- 以bob身份登陆：

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

- 以carol身份登陆:

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

7.2.4. 行级权限的取消

语法

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

在<table_name>处提供表名。该语句将取消对<table_name>指定的表的行级权限设置；执行后，仅有表级权限生效。

举例

```
SHOW PERMISSION ON TABLE finances_orc;
+-----+-----+-----+
| type | enabled | definition |
+-----+-----+-----+
| Row Level Security | true   | WHERE user = current_user() or has_role('admin') |
+-----+-----+-----+
REVOKE PERMISSION ON TABLE finances_orc FOR ROWS;
SHOW PERMISSION ON TABLE finances_orc;
+-----+-----+-----+
| type | enabled | definition |
+-----+-----+-----+
| Row Level Security | false  |           |
+-----+-----+-----+
```

7.2.5. 列级权限的设置

语法：设置列级权限

```
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。

列级权限设置通常结合[脱敏函数](#)使用。下面我们用一个实例来展示其用法。

7.2.5.1. 背景

我们有一张银行用户个人信息表，owner为gerenxinxi用户。

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

为了保证用户个人信息不泄露，银行规定非 **ADMIN** 用户和非owner用户查看bank_account表时需要对敏感列进行脱敏，具体脱敏规则如下：

- **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：邮箱前缀仅显示第一个字母，前缀其他隐藏，用星号代替，@及后面的地址显示

7.2.5.2. 授权

用户首先要对bank_account有表级 **SELECT** 权限：

授予表级权限（由gerenxinxi用户操作）

```
GRANT SELECT ON bank_account TO USER hive, USER zhangsan; ①
```

① 让zhangsan和hive用户都对bank_account有表级 **SELECT** 权限。

下面通过设置列级权限对用户的表级 **SELECT** 权限进行限制：

设置列级权限（由gerenxinxi用户操作）：

```
GRANT PERMISSION ON bank_account FOR COLUMN name CASE WHEN HAS_ROLE('ADMIN') THEN name ELSE
MASK(name, 1) END; ①
GRANT PERMISSION ON bank_account FOR COLUMN id CASE WHEN HAS_ROLE('ADMIN') THEN id ELSE MASK(id, 3,
4) END; ②
GRANT PERMISSION ON bank_account FOR COLUMN email CASE WHEN HAS_ROLE('ADMIN') THEN email ELSE
MASK_EMAIL(email) END; ③
```

- ① 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；
- ② 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；
- ③ 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。



这里的 **MASK** 和 **MASK_EMAIL** 为Inceptor SQL中的脱敏UDF，细节请参考[脱敏相关函数](#)。

查看授权（由gerenguanxi用户操作）：

```
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 |
+-----+-----+
```

7.2.5.3. 不同用户查看表

管理员hive用户登陆，分别以 PUBLIC 和 ADMIN 角色查看表：

hive用户以不同角色查看表bank_account

```
SET ROLE ALL; --使用 *PUBLIC* 角色
SELECT * FROM bank_account;
--只能看到脱敏信息
+-----+-----+-----+
| name | id   | email  |
+-----+-----+-----+
| j*** | 511*****8849 | j***@126.com
| t**  | 320*****8764 | t**@outlook.com
| a*** | 530*****1294 | a***@gmail.com
| a**  | 440*****3889 | a**@qq.com
+-----+-----+-----+
```

```
SET ROLE ADMIN; --切换到 *ADMIN* 角色
SELECT * FROM bank_account;
--可以看到全部信息
+-----+-----+-----+
| name | id   | email  |
+-----+-----+-----+
| jack  | 511702198907108849 | jack@126.com
| tom   | 320702199203128764 | tom@outlook.com
| alice | 530724198511061294 | alice@gmail.com
| amy   | 440183198811183889 | amy@qq.com
+-----+-----+-----+
```

普通用户zhangsan登陆，查看表中内容：

zhangsan用户查看表bank_account

```
SELECT * FROM bank_account;
--只能看到脱敏信息
+-----+-----+-----+
| name | id   | email  |
+-----+-----+-----+
| j*** | 511*****8849 | j***@126.com
| t**  | 320*****8764 | t**@outlook.com
| a*** | 530*****1294 | a***@gmail.com
| a**  | 440*****3889 | a**@qq.com
+-----+-----+-----+
```

7.2.6. 列级权限的取消

语法：取消某张表某个列的列级权限设置

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

例 294. 取消bank_account表email列的列级权限设置

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

语法：取消某张表所有列的列级权限设置

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

例 295. 取消bank_account表中所有列的列级权限设置

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

7.2.7. 行级和列级权限的查看



注意行级和列级权限的查看是 **SHOW PERMISSION**，而表权限的查看是 **SHOW GRANT**。

语法

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

查看表上的行级和列级权限设置。

举例

```
SHOW PERMISSION ON bank_account;
+-----+-----+-----+
| row/column | definition |
+-----+-----+
| ROWS      | WHERE user = current_user() or has_role('admin') |
| 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 |
+-----+-----+
```

7.3. Inceptor中设置HDFS文件的ACL

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。

7.3.1. 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 296. 设置用户bob对表alice_t1的FACL

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

7.3.2. 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>;
```

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。

例 297. 取消用户组bob对表alice_t1的FACL

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

7.3.3. 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

7.3.4. 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 298. 查看用户bob对表alice_t1的FACL

```
SHOW FACL USER bob ON TABLE alice_t1;
```

facl	
user:bob:rwx	①
default:user:bob:rwx	②

① 用户bob对表alice_t1在HDFS上的目录有 **rwx** 权限。

② 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。

7.3.5. 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 299. 查看表alice_t1上的所有FACL

```
SHOW FACL ON TABLE alice_t1;
```

fac1
user:bob:rwx
group::--x
group:sales:rwx
default:user::rwx
default:user:bob:rwx
default:group::--x
default:group:sales:rwx
default:mask::rwx
default:other::--x

① 用户组sales对表alice_t1在HDFS上的目录有 **rwx** 权限。

② 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。

7.4. 空间配额管理操作

空间配额管理操作只能在InceptorServer 2中进行。

7.4.1. 语法总结

7.4.1.1. 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;
```

7.4.1.1.1. 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)

语法: 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)

语法: 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)

语法: 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)

语法: SHOW QUOTA ON TEMPORARY SPACE;

7.4.1.1.2. 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)

语法: 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)

语法: 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)

语法: 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;

7.4.2. 示例操作

7.4.2.1. 数据库数据空间配额管理

- 设置某个Database的数据空间配额

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

```
set role admin;
No rows affected (0.043 seconds)
grant quota 2T on database db1;
No rows affected (0.282 seconds)
```

- 查看某个Database的数据空间配额

语法: SHOW QUOTA ON DATABASE db_name;

```
show quota on database db1;
+-----+
| quota |
+-----+
| 2.000 TB |
+-----+
1 row selected (0.169 seconds)
```

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

语法: GRANT QUOTA unlimited ON DATABASE db_name;

```
grant quota unlimited on database db1;
No rows affected (0.26 seconds)
show quota on database db1;
+-----+
| quota |
+-----+
| unlimited |
+-----+
1 row selected (0.193 seconds)
```

7.4.2.2. 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

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

```
grant quota 1T on database db1 to user user1;
No rows affected (0.178 seconds)
```

- 查看某个用户使用某个database的数据空间的配额

语法: SHOW QUOTA USER user_name ON DATABASE db_name;

```
show quota user user1 on database db1;
+-----+
| quota |
+-----+
| 1.000 TB |
+-----+
1 row selected (0.11 seconds)
```

- 取消某个用户使用某个database的数据空间的配额

语法: GRANT QUOTA unlimited ON DATABASE db_name TO USER user_name;

```
grant quota unlimited on database db1 to user user1;
No rows affected (0.222 seconds)
show quota user user1 on database db1;
+-----+
| quota |
+-----+
| unlimited |
+-----+
1 row selected (0.714 seconds)
```

7.4.2.3. 临时空间配额管理

7.4.2.3.1. 用户临时空间配额管理

- 设置某个用户的临时空间配额

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

```
set role admin;
No rows affected (0.051 seconds)
grant quota 500G on temporary space to user user1;
No rows affected (0.047 seconds)
```

- 查看某个用户的临时空间配额

语法: SHOW QUOTA USER user_name ON TEMPORARY SPACE;

```
show quota user user1 on temporary space;
+-----+
| quota |
+-----+
| 500.000 GB |
+-----+
1 row selected (0.092 seconds)
```

- 取消某个用户的临时空间配额

语法: GRANT QUOTA unlimited ON TEMPORARY SPACE TO USER user_name;

```
grant quota unlimited on temporary space to user user1;
No rows affected (0.073 seconds)
show quota user user1 on temporary space;
+-----+
| quota |
+-----+
| unlimited |
+-----+
1 row selected (0.034 seconds)
```

7.4.2.3.2. 所有临时空间配额管理

- 设置所有临时空间总的配额

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

```
grant quota 2T on temporary space;
No rows affected (0.225 seconds)
```

- 查看所有临时空间总的配额

语法: SHOW QUOTA ON TEMPORARY SPACE;

```
show quota on temporary space;
+-----+
| quota |
+-----+
| 2.000 TB |
+-----+
1 row selected (0.308 seconds)
```

- 取消所有临时空间总的配额

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

```
grant quota unlimited on temporary space;
No rows affected (0.093 seconds)
show quota on temporary space;
+-----+
| quota      |
+-----+
| unlimited  |
+-----+
1 row selected (0.274 seconds)
```

8. Inceptor JDBC手册

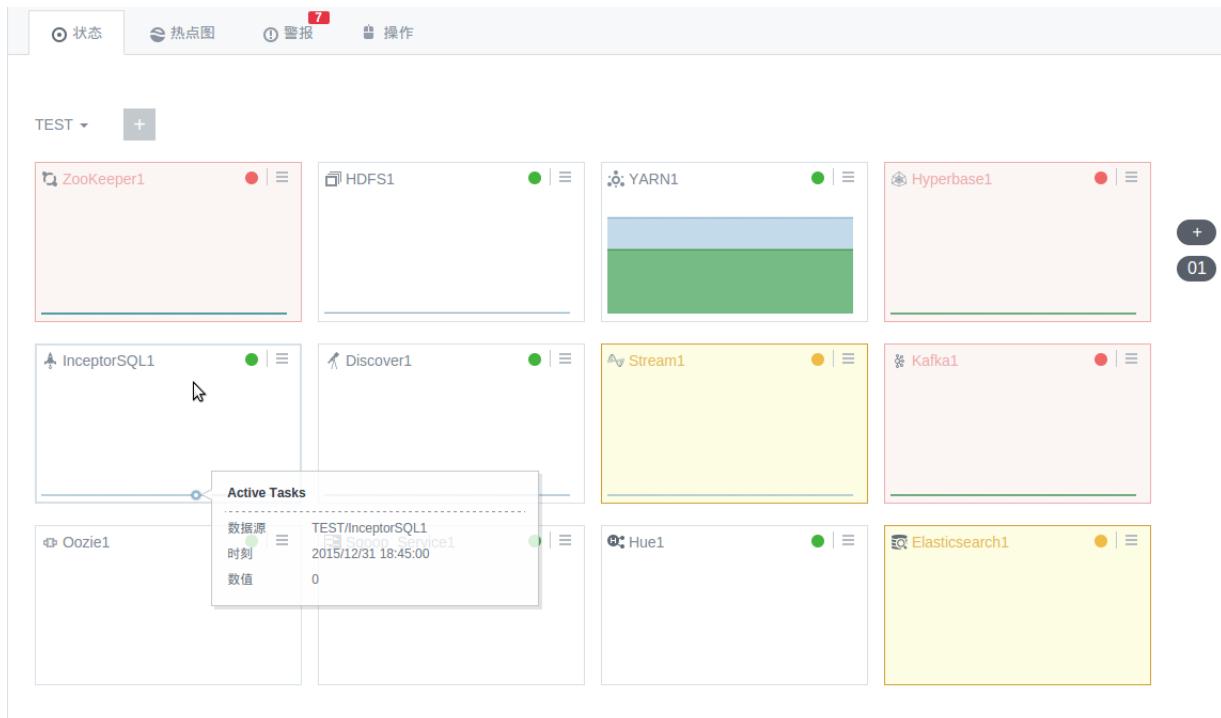
8.1. Inceptor JDBC手册一览

Inceptor提供标准的JDBC，使数据库开发人员可以方便地编写Inceptor应用程序。同时，Inceptor提供的JDBC让用户可以方便地使用BI工具连接到Inceptor进行操作。Inceptor JDBC手册将仅介绍如何使用Inceptor JDBC进行应用开发。使用第三方BI工具连接到Inceptor的使用方法请参见[Inceptor外部工具连接手册](#)。

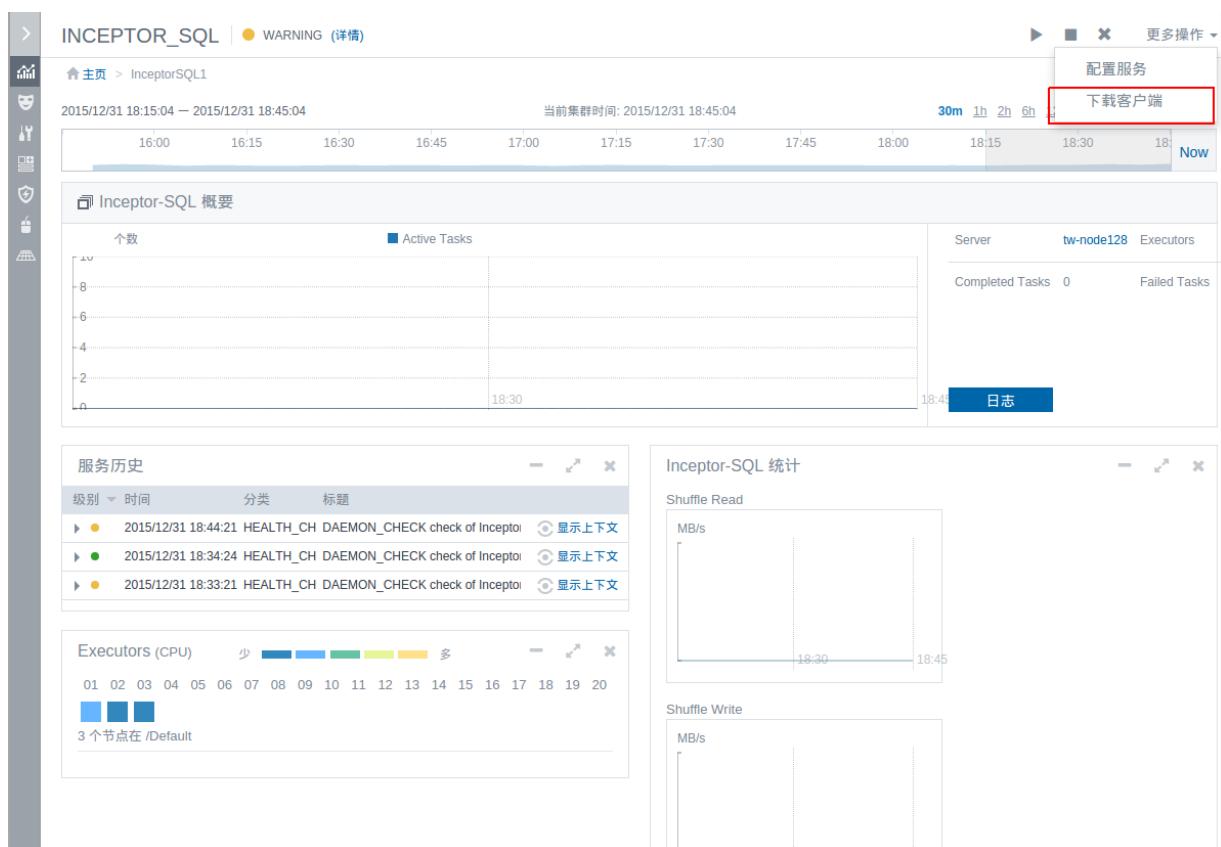
8.1.1. 获取Inceptor JDBC驱动

从TDH4.3开始，您可以从Transwarp Manager上直接下载Inceptor JDBC驱动。步骤如下：

1. 登陆Transwarp Manager，在Transwarp Manager的主界面上点击Inceptor服务



2. 进入Inceptor服务的界面，点击界面右上方的“更多操作”，会看到“配置服务”，“下载客户端”的选项



- 点击“下载客户端”，系统会自动下载Inceptor的JDBC驱动。

8.2. 应用程序中的JDBC交互

我们可以分别在应用程序中和工具中通过JDBC与Inceptor交互，本节将只介绍在应用程序中如何使用JDBC分别连接InceptorServer 1和InceptorServer 2，关于使用工具和Inceptor交互的内容请参考本手册中的“使用工具连接Inceptor”。



注意，使用JDBC交互前您需要将Inceptor JDBC的驱动已经放在您本地的CLASSPATH中。您可以从Transwarp Manager上下载Inceptor JDBC驱动，细节请参考《Transwarp Data Hub运维手册》中的“InceptorSQL服务的配置”一节。

下面，我们将给出在不同场景下通过JDBC访问Inceptor执行下面SQL语句的应用程序：

```
SELECT c1, c2 FROM table1;
```

通过JDBC访问Inceptor需要根据使用的Hive Sever选择驱动：



- 如果使用InceptorServer 1，驱动名为：org.apache.hadoop.hive.jdbc.HiveDriver
- 如果使用InceptorServer 2，驱动名为：org.apache.hive.jdbc.HiveDriver

这两个驱动名字很接近，请在选用的时候注意区分。

例 300. 和无需认证的InceptorServer 1交互

```
import java.sql.*;

public class JDBCExample {
    //Hive1 Driver class name
    private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver"; ①

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

        //Hive1 JDBC URL
        String jdbcURL = "jdbc:hive://localhost:10000/default"; ②

        Connection conn = DriverManager.getConnection(jdbcURL);
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("select c1, c2 from table1"); ③
        while(rs.next()) {
            System.out.println(rs.getString(1));
            System.out.println(rs.getString(2));
        }
        rs.close();
        stmt.close();
        conn.close();
    }
}
```

① org.apache.hadoop.hive.jdbc.HiveDriver为InceptorServer 1的JDBC驱动，在和InceptorServer 1交互时使用。

② 连接到InceptorServer 1时使用的JDBC连接串。这个连接串连接到Inceptor中的default数据库，如果您想要连接到其他数据库，“default”所在位置还可以填写其他数据库名。

③ 执行的SQL语句。

例 301. 和无需认证的InceptorServer 2交互

```

import java.sql.*;
public class JDBCExample {
    //Hive2 Driver class name
    private static String driverName = "org.apache.hive.jdbc.HiveDriver"; ①
    public static void main(String[] args) throws SQLException {
        try {
            Class.forName(driverName);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.exit(1);
        }
        //Hive2 JDBC URL with LDAP
        String jdbcURL = "jdbc:hive2://localhost:10000/default"; ②
        Connection conn = DriverManager.getConnection(jdbcURL);
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("select c1, c2 from table1"); ③
        while(rs.next()) {
            System.out.println(rs.getString(1));
            System.out.println(rs.getString(2));
        }
        rs.close();
        stmt.close();
        conn.close();
    }
}

```

① org.apache.hive.jdbc.HiveDriver为InceptorServer 2的JDBC驱动，在和InceptorServer 2交互时使用。

② 连接到使用LDAP认证的InceptorServer 2的JDBC连接串。这个连接串连接到Inceptor中的default数据库，如果您想要连接到其他数据库，“default”所在位置还可以填写其他数据库名。

③ 执行的SQL语句。

例 302. 和Kerberos认证的InceptorServer 2交互

```

import java.sql.*;

public class JDBCExample {
    //Hive2 Driver class name
    private static String driverName = "org.apache.hive.jdbc.HiveDriver"; ①

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

        //Hive2 JDBC URL with kerberos certification
        String jdbcURL = "jdbc:hive2://localhost:10000/default;principal=hive/node@TDH;" + ②
            "authentication=kerberos;" +
            "kuser=user_principal;" + ③
            "keytab=keytab_path;" + ④
            "krb5conf=krb5.conf_path"; ⑤

        Connection conn = DriverManager.getConnection(jdbcURL);
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("select c1, c2 from table1"); ⑥
        while(rs.next()) {
            System.out.println(rs.getString(1));
            System.out.println(rs.getString(2));
        }
        rs.close();
        stmt.close();
        conn.close();
    }
}

```

- ① org.apache.hive.jdbc.HiveDriver为InceptorServer 2的JDBC驱动，在和InceptorServer 2交互时使用。
- ② 连接到使用Kerberos认证的InceptorServer 2的JDBC连接串。“hive/node@TDH”处需要填您想要连接的Inceptor server的principal。这个连接串连接到Inceptor中的default数据库，如果您想要连接到其他数据库，“default”所在位置还可以填写其他数据库名。
- ③ “user_principal”处填写访问Inceptor server的用户的principal。
- ④ keytab_path处填写user_principal的keytab文件的 **绝对路径**。目前我们还不支持在应用程序中使用principal加密码认证，必须使用principal加keytab认证。
- ⑤ krb5.conf_path处填写Kerberos配置文件krb5.conf的 **绝对路径**。
- ⑥ 执行的SQL语句。

您也可以通过Kerberos客户端执行kinit user_principal来为user_principal获取TGT后再使用应用程序连接InceptorServer 2，这种情况下，您不需要上面示例中的user_principal和keytab_dir信息：



kuser=user_principal;
keytab=keytab_dir;

但是您必须保证每次运行应用程序前，user_principal都已经有有效的TGT。

例 303. 和LDAP认证的InceptorServer 2交互

```

import java.sql.*;
public class JDBCExample {
    //Hive2 Driver class name
    private static String driverName = "org.apache.hive.jdbc.HiveDriver"; ①
    public static void main(String[] args) throws SQLException {
        try {
            Class.forName(driverName);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.exit(1);
        }
        //Hive2 JDBC URL with LDAP
        String jdbcURL = "jdbc:hive2://localhost:10000/default"; ②
        String user = "user"; ③
        String password = "password"; ④
        Connection conn = DriverManager.getConnection(jdbcURL, user, password);
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("select c1, c2 from table1"); ⑤
        while(rs.next()) {
            System.out.println(rs.getString(1));
            System.out.println(rs.getString(2));
        }
        rs.close();
        stmt.close();
        conn.close();
    }
}

```

- ① org.apache.hive.jdbc.HiveDriver为InceptorServer 2的JDBC驱动，在和InceptorServer 2交互时使用。
- ② 连接到使用LDAP认证的InceptorServer 2的JDBC连接串。这个连接串连接到Inceptor中的default数据库，如果您想要连接到其他数据库，“default”所在位置还可以填写其他数据库名。
- ③ 访问InceptorServer 2的用户的用户名
- ④ 访问InceptorServer 2的用户的密码
- ⑤ 执行的SQL语句。

例 304. 取消一个SQL

```
import java.sql.*;

public class HiveCancelTest extends Thread{
    private static String driverName = "org.apache.hive.jdbc.HiveDriver";
    Statement stmt;

    HiveCancelTest(Statement stmt){
        this	stmt = stmt;
    }

    public void run(){
        try {
            //execute a complicated SQL query
            ResultSet rs = stmt.executeQuery("select count(*) from xxx");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

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

        Connection conn = DriverManager.getConnection("jdbc:hive2://172.16.1.87:10000/tpcds_orc_2",
        "", "");
        Statement stmt = conn.createStatement();
        //new a thread
        HiveCancelTest t1 = new HiveCancelTest(stmt);
        t1.start();

        //main thread do the check work, and cancel if the task exceed 10000 ms
        sleep(10000);
        if(t1.isAlive()){
            stmt.cancel();
        }
    }
}
```

9. Inceptor ODBC手册

9.1. Inceptor ODBC手册一览

Inceptor提供标准的ODBC，使数据库开发人员可以方便地编写Inceptor应用程序。同时，Inceptor提供的ODBC让用户可以方便地使用BI工具连接到Inceptor进行操作。Inceptor ODBC手册将仅介绍如何使用Inceptor ODBC进行应用开发。使用第三方BI工具连接到Inceptor的使用方法请参考[Inceptor外部工具连接手册](#)。

在用ODBC进行Inceptor应用程序开发前，您需要在您的计算机上部署ODBC连接环境。我们将分别介绍如何在Windows系统和Linux系统中进行相关部署。

9.2. Windows操作系统中的ODBC交互准备

使用工具通过ODBC连接Inceptor之前，您需要：

1. 安装ODBC驱动
2. 如果您的Inceptor server用Kerberos认证，修改相关配置文件以及系统变量
3. 创建合适的DSN

下面我们将具体介绍这些步骤。

9.2.1. 安装ODBC驱动

您应该已经有我们提供的InceptorServer 1和InceptorServer 2分别在x86/x64下的驱动：

```
TranswarpODBCDriver_Hive1_x64.exe
TranswarpODBCDriver_Hive1_x86.exe
TranswarpODBCDriver_Hive2_x64.exe
TranswarpODBCDriver_Hive2_x86.exe
```

如果您没有这些驱动，请和我们联系。

根据您的使用场景，选择您所需要使用的驱动（如果使用InceptorServer 1，选择TranswarpODBCDriver_Hive1_x64.exe或TranswarpODBCDriver_Hive1_x86.exe，如果使用InceptorServer 2，选择TranswarpODBCDriver_Hive2_x64.exe或TranswarpODBCDriver_Hive2_x86.exe），双击安装。安装过程简单明了，这里不赘述。

9.2.2. 修改配置文件和系统变量



如果您的Inceptor server使用了Kerberos认证，您需要完成本节的操作。如果您的Inceptor server没有使用Kerberos认证，您可以直接跳到下一节查看如何创建DSN。

1. 您需要修改您电脑上的hosts文件来提供Kerberos server所在节点的hostname解析。您可以在C盘中的Windows目录下直接搜索“hosts”。hosts文件路径一般为：

C:\Windows\System32\drivers\etc\hosts

打开这个文件进行编辑。在文件的末尾添加：

```
kerberos_server_ip  kerberos_server_hostname
```

这里kerberos_server_ip是您Kerberos server所在节点的ip; kerberos_server_name是这个节点的hostname。

比如：

```
# Copyright (c) 1993-2009 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for
Windows.
#
# This file contains the mappings of IP addresses to host
names. Each
# entry should be kept on an individual line. The IP address
should
# be placed in the first column followed by the corresponding
host name.
# The IP address and the host name should be separated by at
least one
# space.
#
# Additionally, comments (such as these) may be inserted on
individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#      102.54.94.97      rhino.acme.com          # source
server
#      38.25.63.10      x.acme.com              # x client
host

# localhost name resolution is handled within DNS itself.
#      127.0.0.1          localhost
#      ::1                localhost
172.16.1.86      baogang1
```

2. 您需要修改您电脑上的Kerberos配置文件krb5.ini来提供您集群的realm信息。这个文件的路径为：

C:\ProgramData\MIT\Kerberos5\krb.ini

打开这个文件进行编辑：

- [libdefaults]下的default_realm, 将其值设为您集群的realm名。
- [realms]下的kdc, 将其值设为：

```
kerberos_server_hostname:88
```

- c. [realm]下的admin_server，将其值设置为：

```
kerberos_server_hostname:749
```

比如：

```

#
# Unless required by applicable law or agreed to in writing,
software
# distributed under the License is distributed on an "AS IS"
BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
# See the License for the specific language governing permissions
and
# limitations under the License.

[libdefaults]
    default realm = TDH
    dns_lookup_realm = false
    dns_lookup_kdc = false
    ticket_lifetime = 24h
    forwardable = true
    udp_preference_limit = 1000000

[realms]
    TDH = {
        kdc = baogang1:88
        admin_server = baogang1:749
    }

[logging]
    kdc = C:\ProgramData\MIT\Kerberos5\krb5kdc.log
    admin_server = C:\ProgramData\MIT\Kerberos5\kadmin.log
    default = C:\ProgramData\MIT\Kerberos5\krb5lib.log

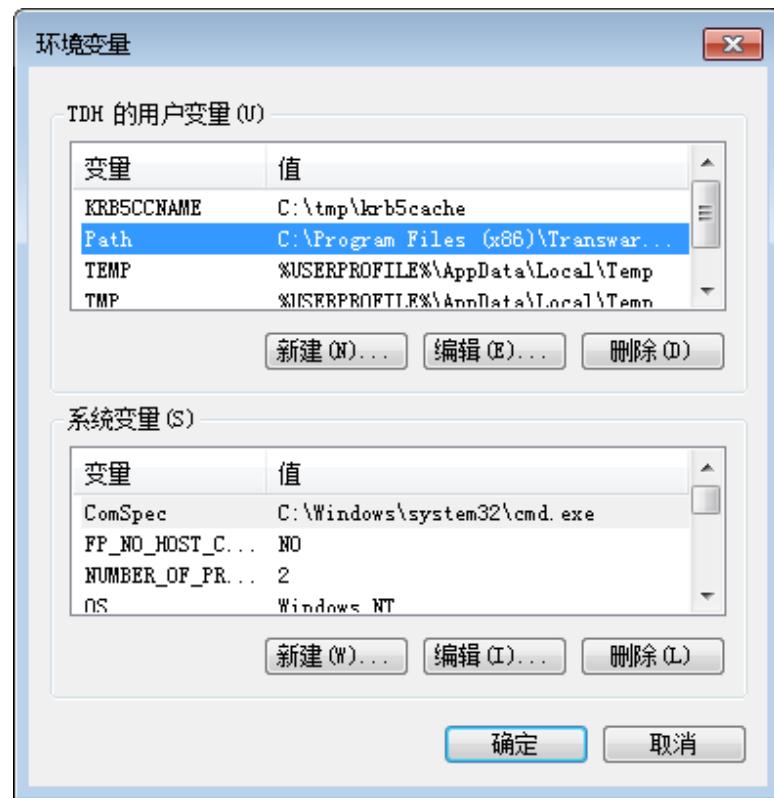
```

3. 最后您需要保证下面两个路径存在于您的 系统变量 的PATH中：

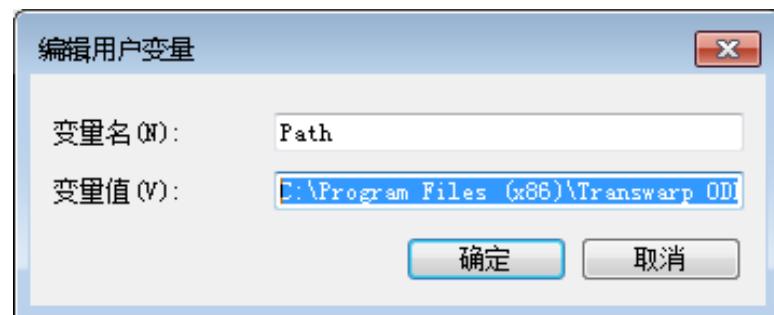
```
C:\Program Files (x86)\Transwarp ODBC Driver For HiveServer2\Kerberos;C:\Program Files (x86)\Transwarp ODBC Driver For HiveServer2\Runtime
```

打开您电脑的环境变量设置。

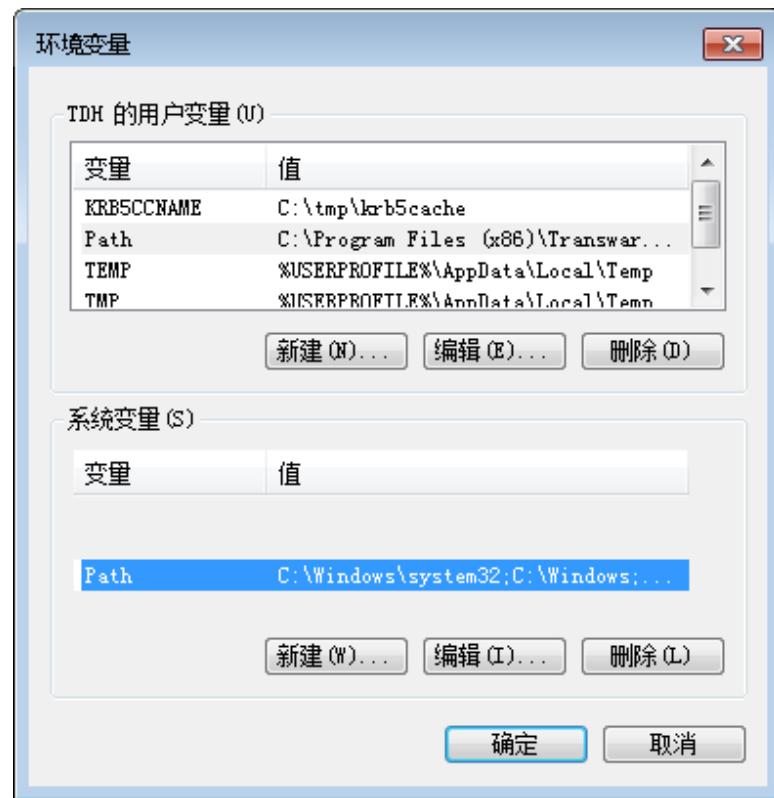
查看“系统变量”中的“PATH”是否包含上面两个路径，如果没有，您需要将它们添加进去。这两个路径已经在您用户变量的PATH中：



选中“用户变量”中的“PATH”，点击“用户变量”下的“编辑”：



拷贝这两个路径。然后点击“取消”。回到环境变量设置。选中“系统变量”中的“PATH”，点击“系统变量”下的“编辑”：



将刚才拷贝的两个路径粘贴到您系统变量的PATH中，注意路径与路径之间以分号隔开：



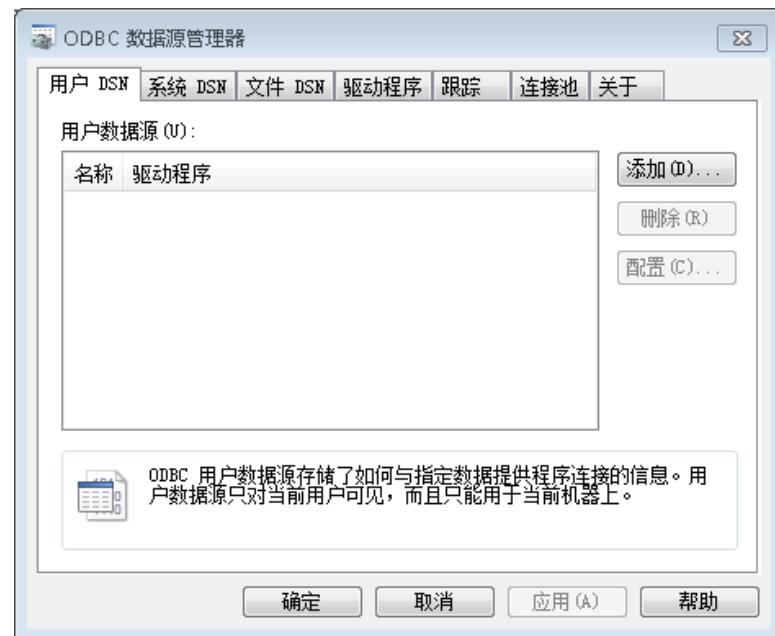
点击“确定”退出。

9.2.3. 创建合适的DSN

下面我们将介绍为使用不同认证方式的Inceptor server如何创建对应的DSN。

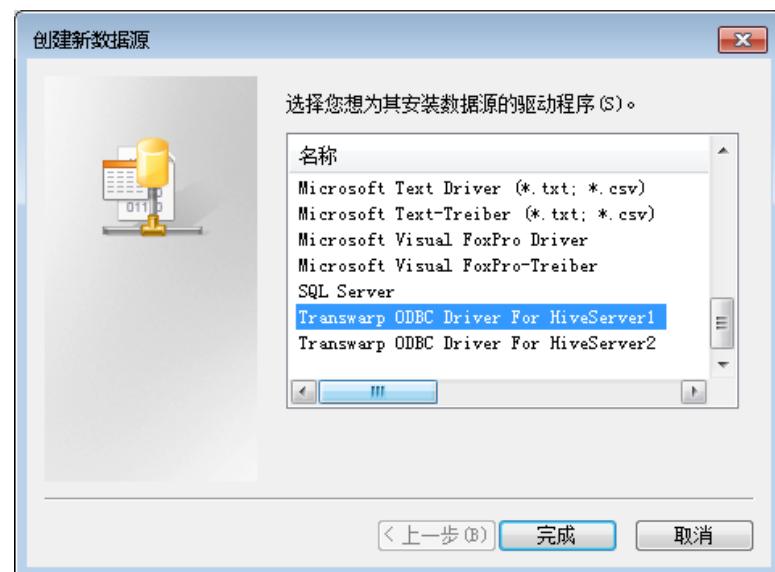
9.2.3.1. 连接无需认证的InceptorServer 1的DSN

1. 打开ODBC数据源管理器：

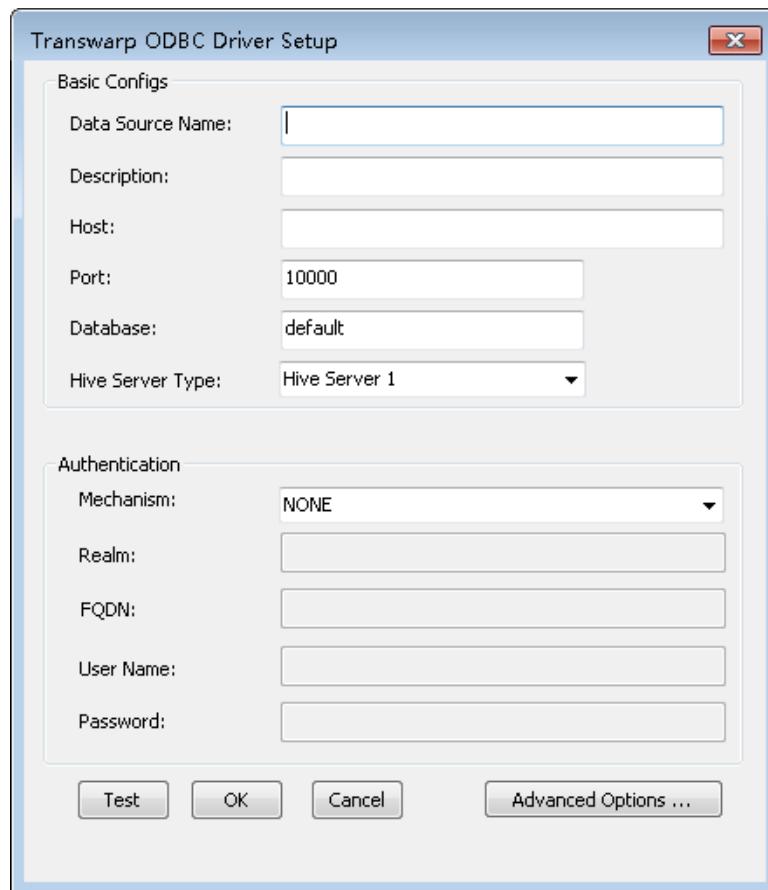


点击“Add”添加DSN。

2. 您应该能看到安装好的Transwarp ODBC Driver For HiveServer1和Transwarp ODBC Driver For HiveServer2这两个驱动。选择Transwarp ODBC Driver For HiveServer1:



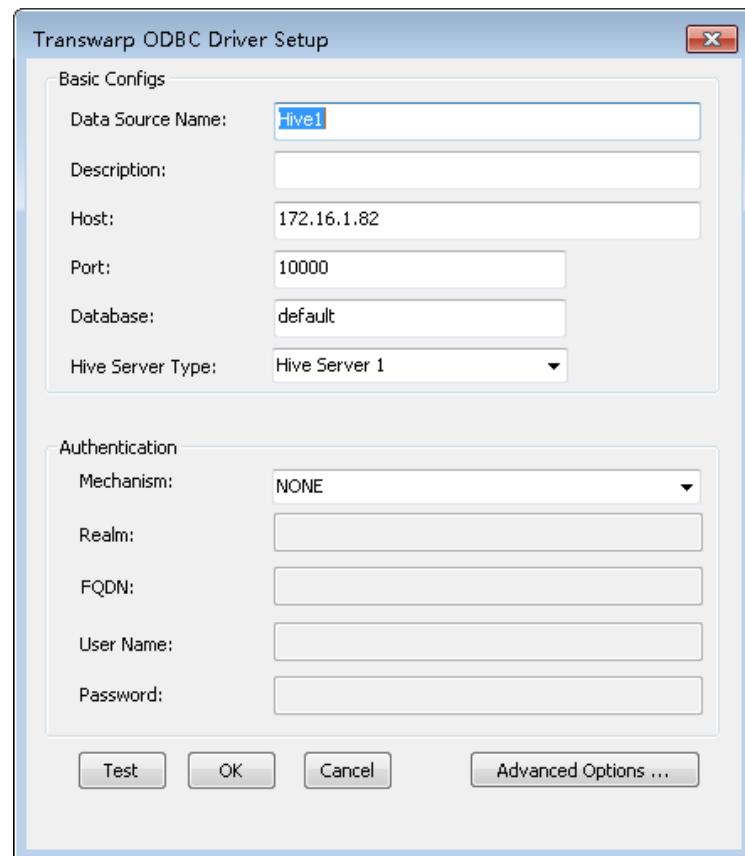
点击“Finish”继续。您将看到下面弹出的窗口：



3. 在该窗口中您需要：

- 在Data Source Name处为您的数据源命名，这里我们输入了Hive1。
- Description处您可以选择添加数据源的描述信息，这里我们空着未填。
- 在Host处填写Inceptor server所在节点的IP，这里我们填写了172.16.1.82。
- Port处为Inceptor server对应的端口号10000，您无需做更改。
- Database处你可以选择您想要使用的Inceptor中的数据库。这里我们使用default。
- Hive Server Type处选择Hive Server 1。
- Authentication部分的Mechanism选择NONE。

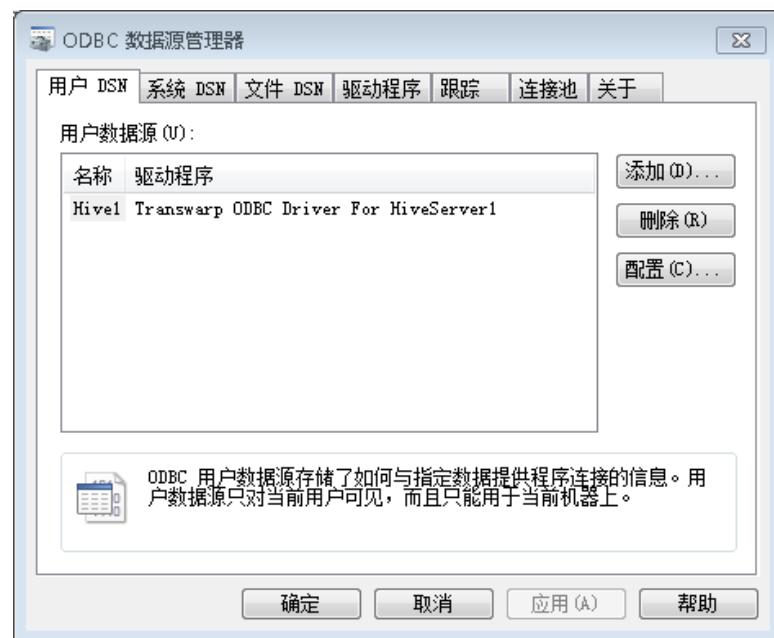
填写完毕后，窗口应该显示如下：



4. 点击上图中的“Test”，您可以测试这个DSN的连接。看到下图说明连接成功：

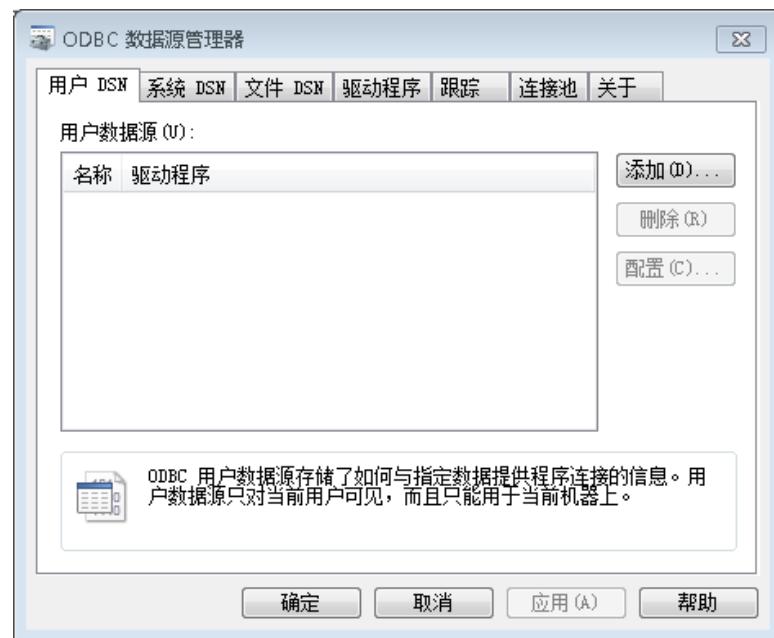


5. 现在ODBC Data Source Administrator中将显示我们刚刚添加的Hive1这个DSN。

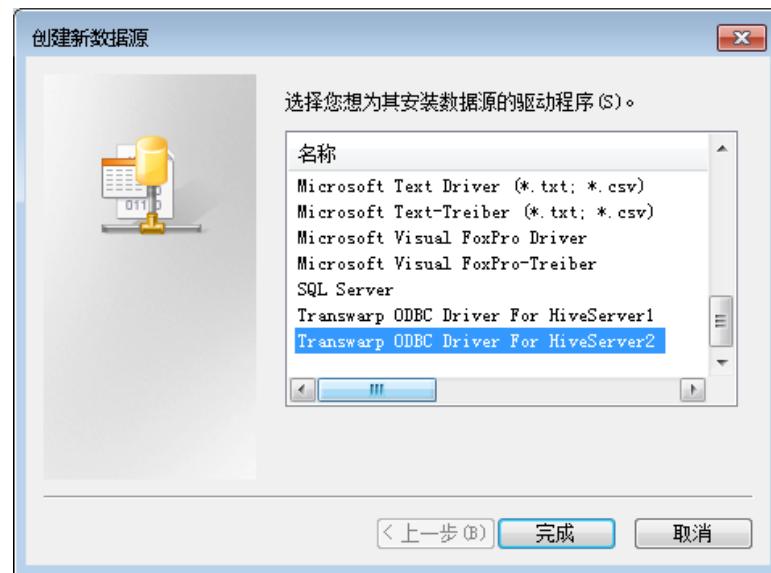


9.2.3.2. 连接无需认证的InceptorServer 2的DSN

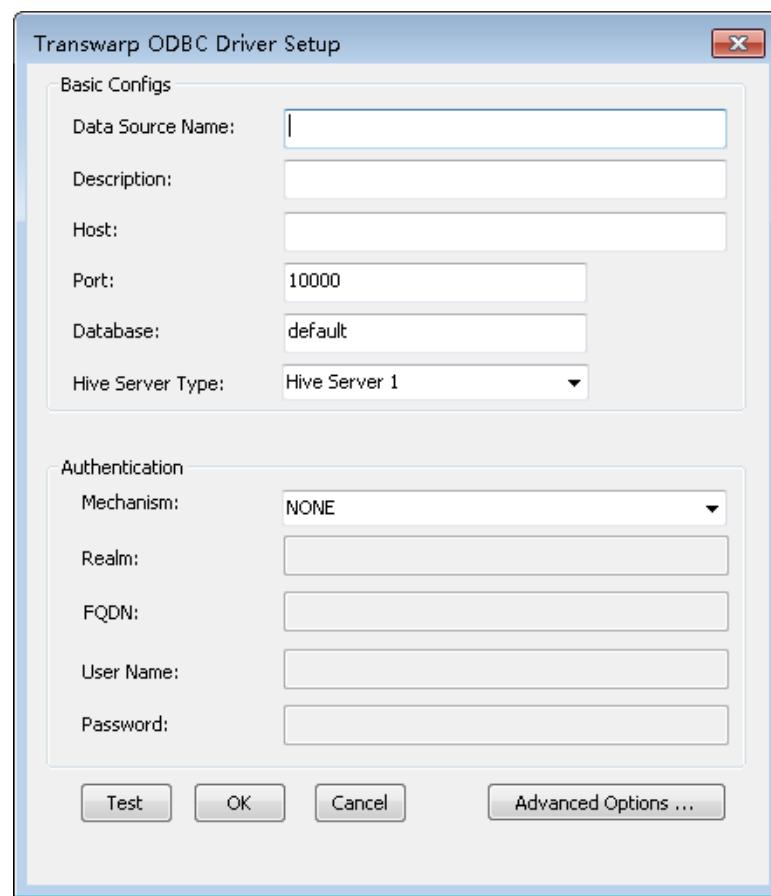
1. 打开您电脑中的ODBC Data Source Administrator，点击下图中的“Add”：



2. 选择Transwarp ODBC Driver For HiveServer2:



点击“Finish”继续。将弹出下面窗口：

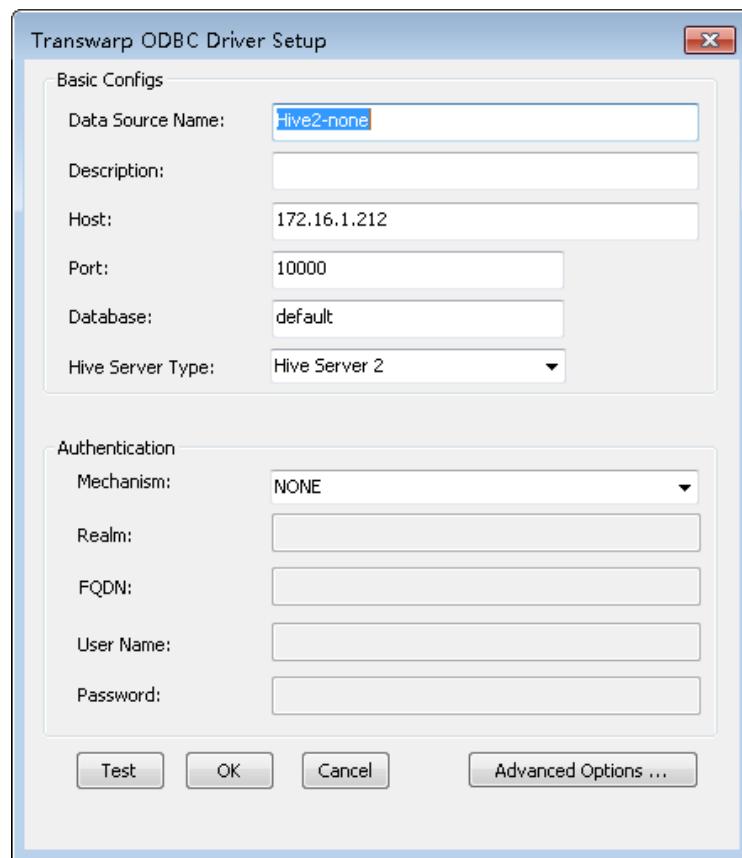


3. 在该窗口中您需要：

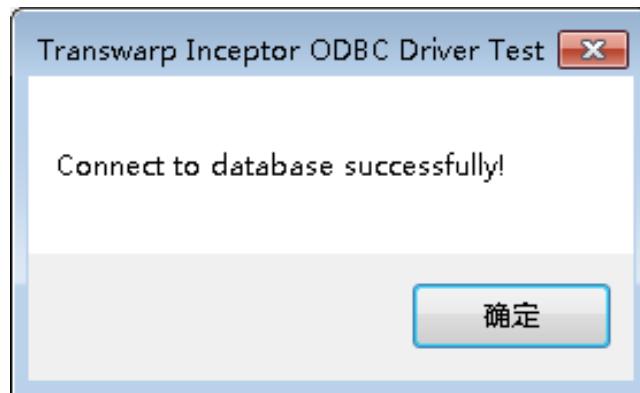
- 在Data Source Name处为您的数据源命名，这里我们输入了Hive2-none。
- Description处您可以选择添加数据源的描述信息，这里我们空着未填。
- 在Host处填写Inceptor server所在节点的IP，这里我们填写了172.16.1.212。
- Port处为Inceptor server对应的端口号10000，您无需做更改。
- Database处你可以选择您想要使用的Inceptor中的数据库。这里我们使用default。
- Hive Server Type处选择Hive Server 2。

- 下面的Authentication部分的Mechanism选择NONE。

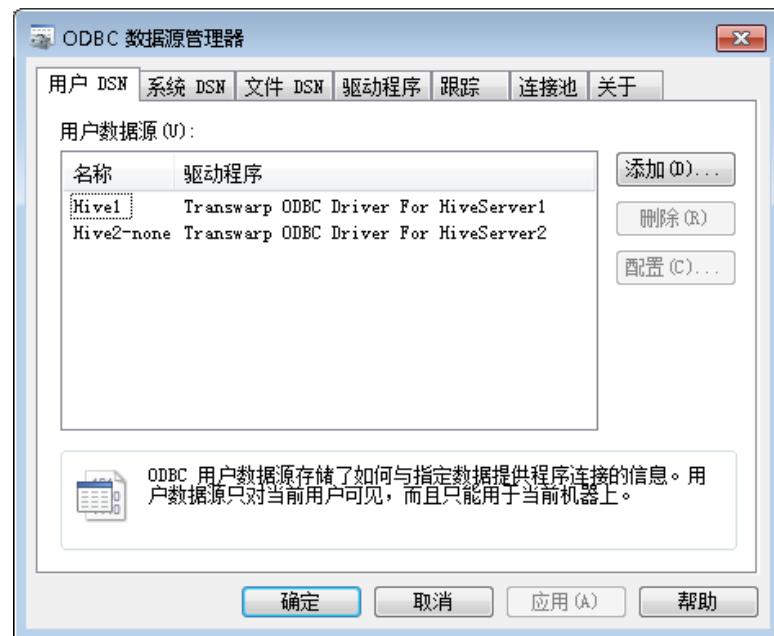
填写完毕后，窗口应该显示如下：



- 点击上图中的“Test”，您可以测试这个DSN的连接。看到下图说明连接成功：

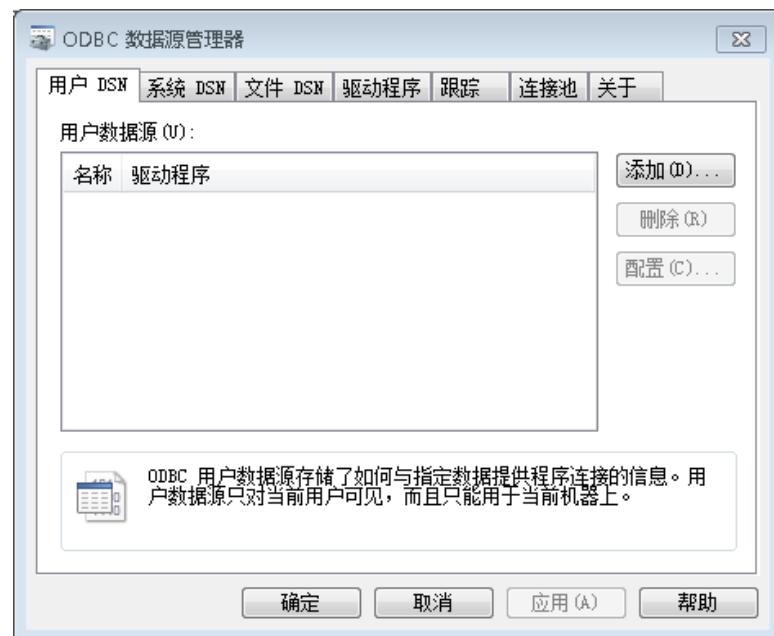


- 现在ODBC Data Source Administrator中将显示我们刚刚添加的Hive2-none这个DSN。

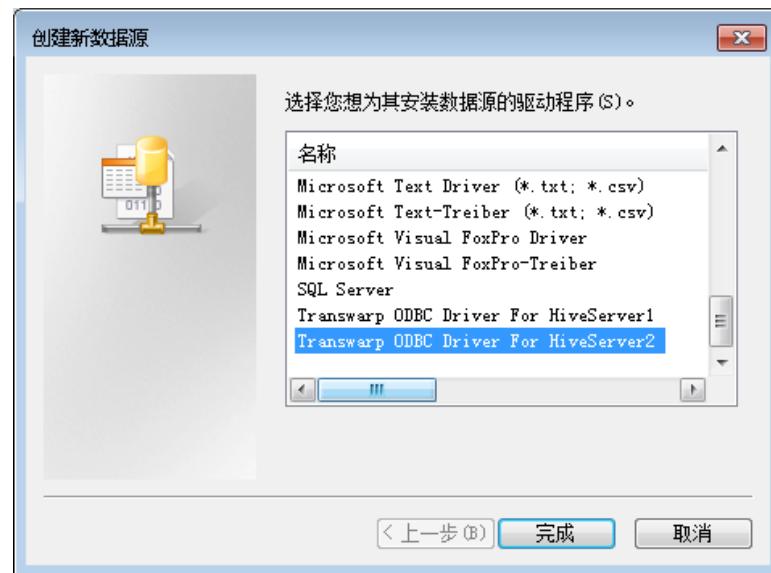


9.2.3.3. 连接LDAP认证的InceptorServer 2的DSN

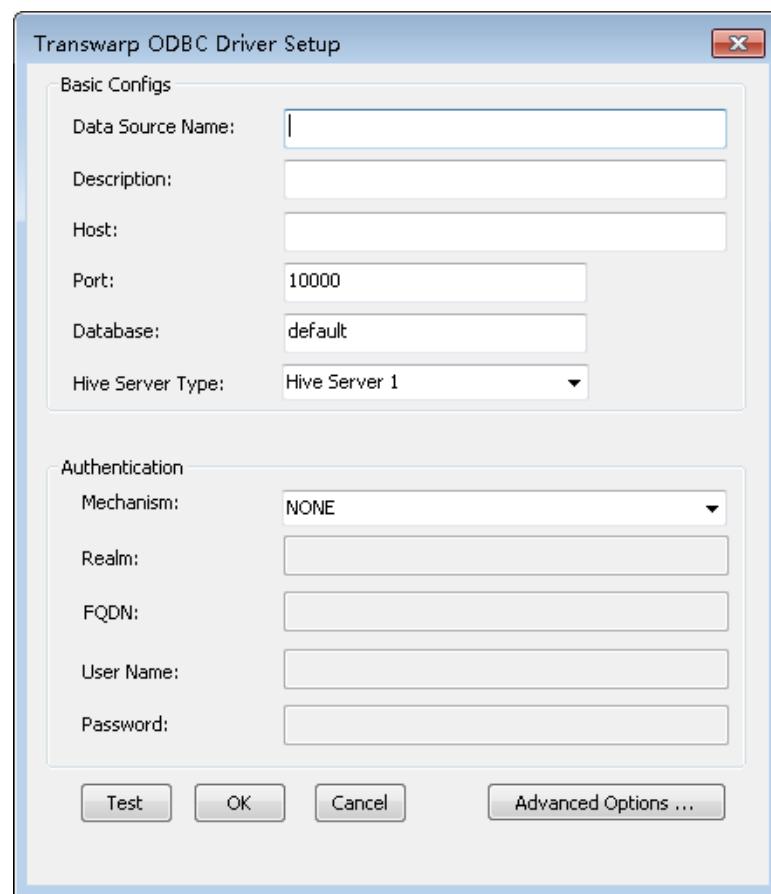
1. 打开您电脑中的ODBC Data Source Administrator，点击下图中的“Add”：



2. 选择Transwarp ODBC Driver For HiveServer2:



点击“Finish”继续。将弹出下面窗口：

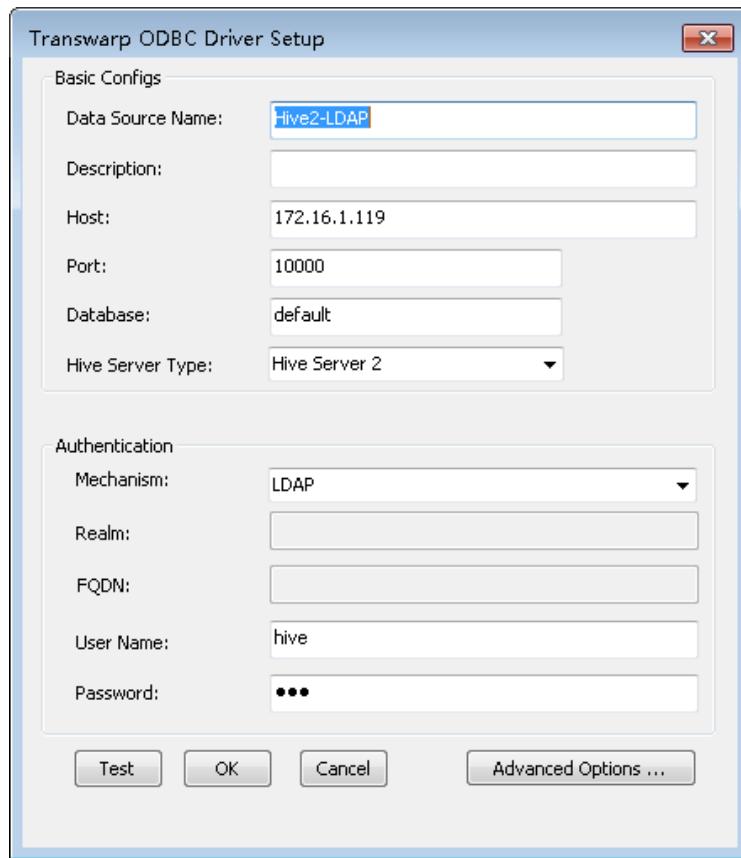


3. 在该窗口中您需要：

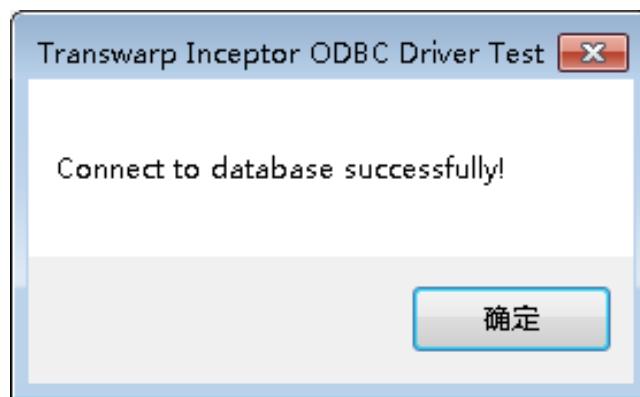
- 在Data Source Name处为您的数据源命名，这里我们输入了Hive2-LDAP。
- Description处您可以选择添加数据源的描述信息，这里我们空着未填。
- 在Host处填写Inceptor server所在节点的IP，这里我们填写了172.16.1.119。
- Port处为Inceptor server对应的端口号10000，您无需做更改。
- Database处你可以选择您想要使用的Inceptor中的数据库。这里我们使用default。
- Hive Server Type处选择Hive Server 2。

- 下面的Authentication部分的Mechanism选择LDAP。
- User Name填写您想要用来登陆HiveServer2的用户的用户名。这里我们填写了hive。
- Password填写您想要用来登陆HiveServer2的用户的密码。

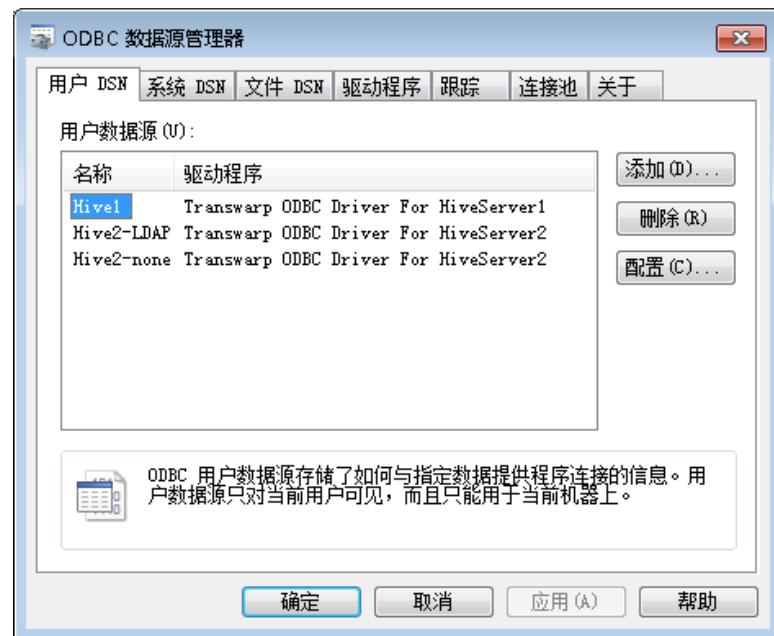
填写完毕后，窗口应该显示如下：



4. 点击上图中的“Test”，您可以测试这个DSN的连接。看到下图说明连接成功：

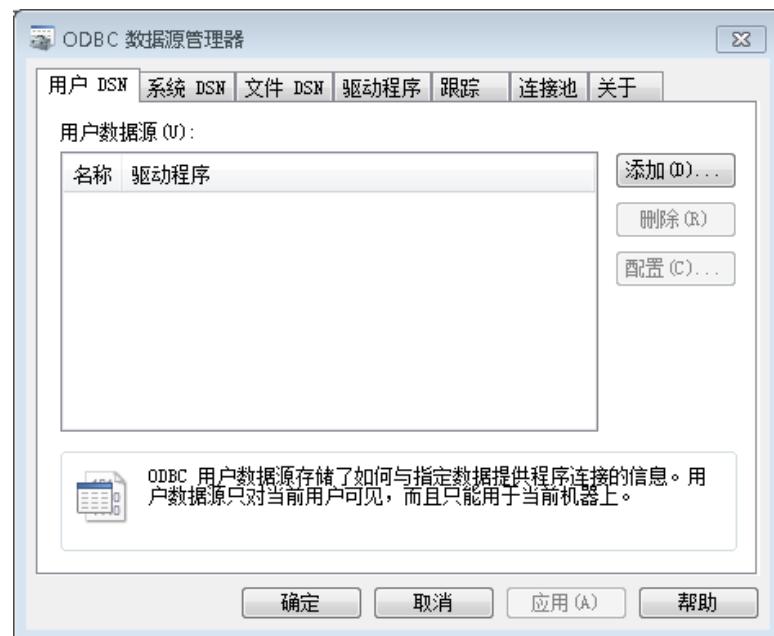


5. 现在ODBC Data Source Administrator中将显示我们刚刚添加的Hive2-LDAP这个DSN。

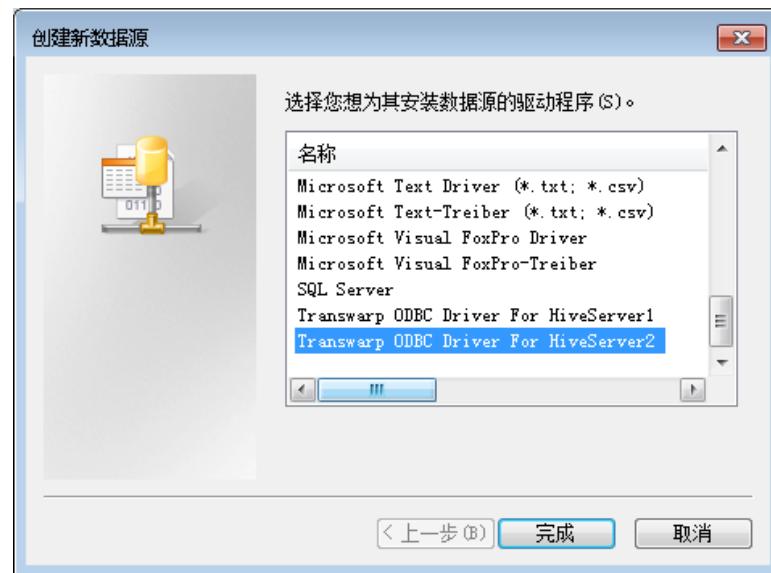


9.2.3.4. 连接Kerberos认证的InceptorServer 2的DSN

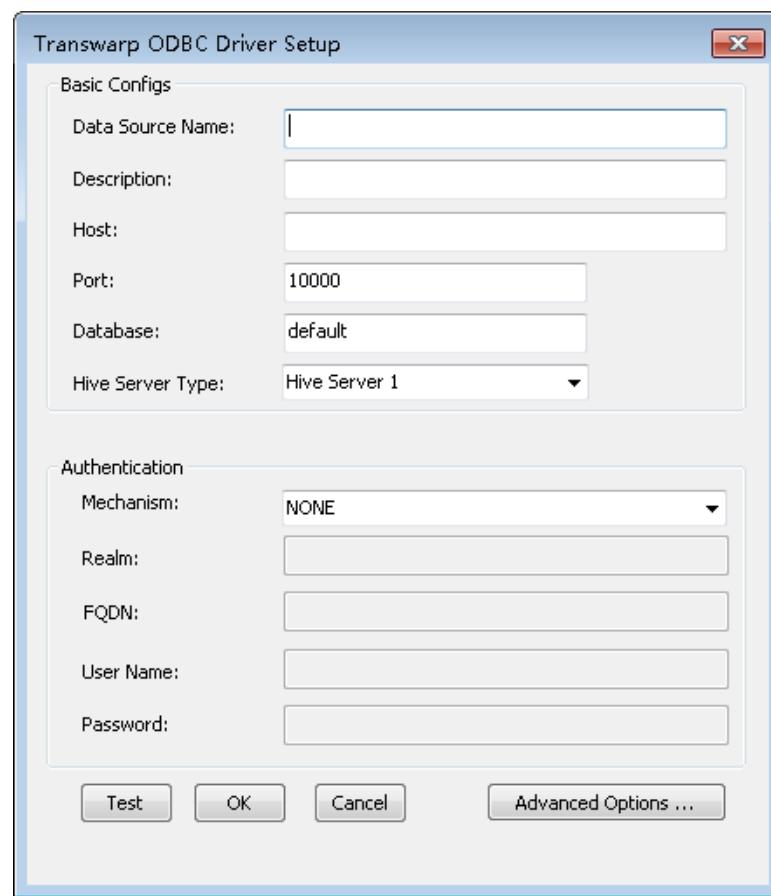
1. 打开您电脑中的ODBC Data Source Administrator，点击下图中的“Add”：



2. 选择Transwarp ODBC Driver For HiveServer2:



点击“Finish”继续。将弹出下面窗口：

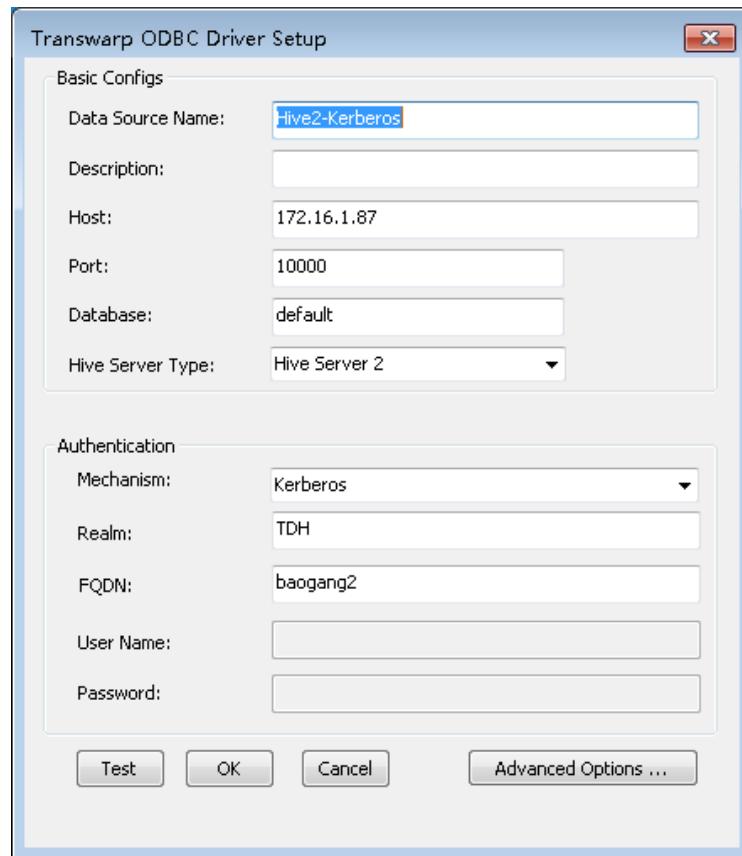


3. 在该窗口中您需要：

- 在Data Source Name处为您的数据源命名，这里我们输入了Hive2-Kerberos。
- Description处您可以选择添加数据源的描述信息，这里我们空着未填。
- 在Host处填写Inceptor server所在节点的IP，这里我们填写了172.16.1.87。
- Port处为Inceptor server对应的端口号10000，您无需做更改。
- Database处你可以选择您想要使用的Inceptor中的数据库。这里我们使用default。
- Hive Server Type处选择Hive Server 2。

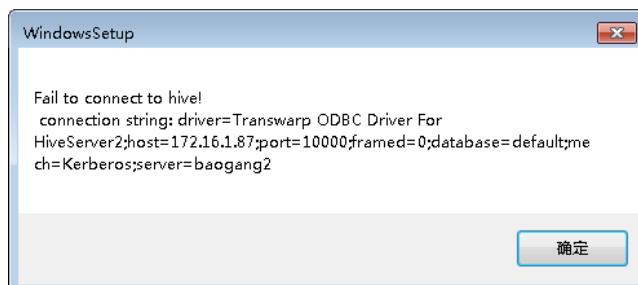
- 下面的Authentication部分的Mechanism选择Kerberos。
- Realm填写您集群的Kerberos域。这里我们填写了TDH。
- FQDN处填写您HiveServer2所在节点的hostname。这里我们填写了baogang2。

填写完毕后，窗口应该显示如下：



4. 点击上图中的“Test”，您可以测试这个DSN的连接。看到下图说明连接成功：

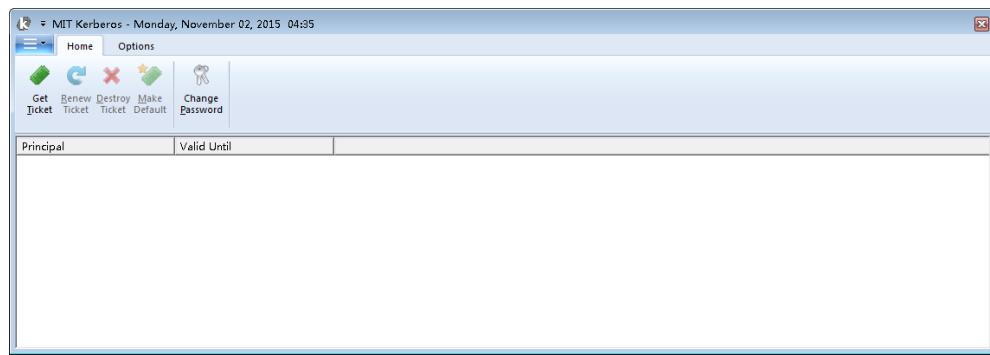
这时，您可能会看到下面的信息：



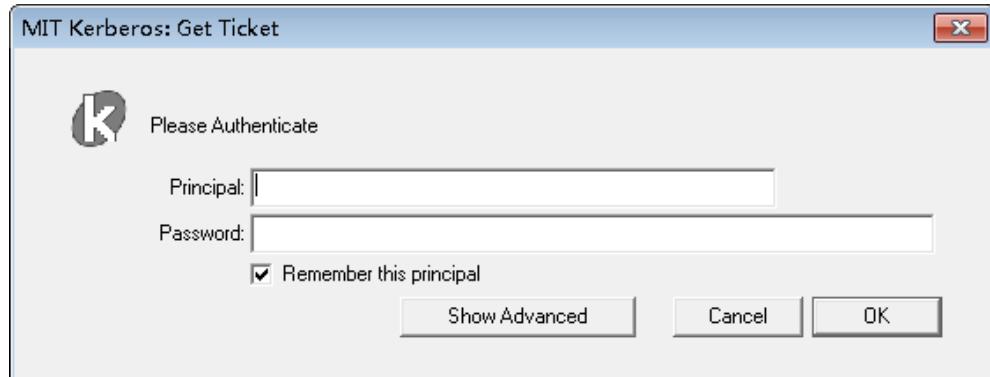
这时因为您还没有通过Kerberos的认证。为了获得认证，您需要在您的电脑上获取一张有效的TGT。获得TGT的方法为：

- a. 打开Kerberos客户端程序，这个程序包含在我们提供的Transwarp ODBC Driver For HiveServer2里，程序的路径为：

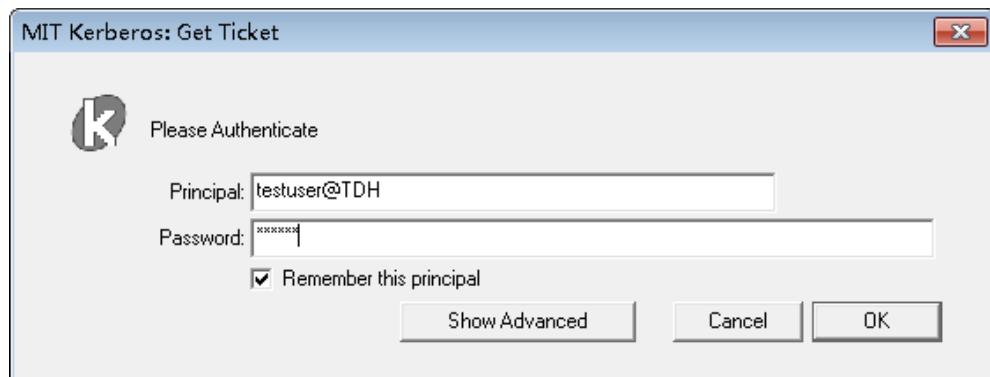
C:\Program Files (x86)\Transwarp ODBC Driver For HiveServer2\Kerberos



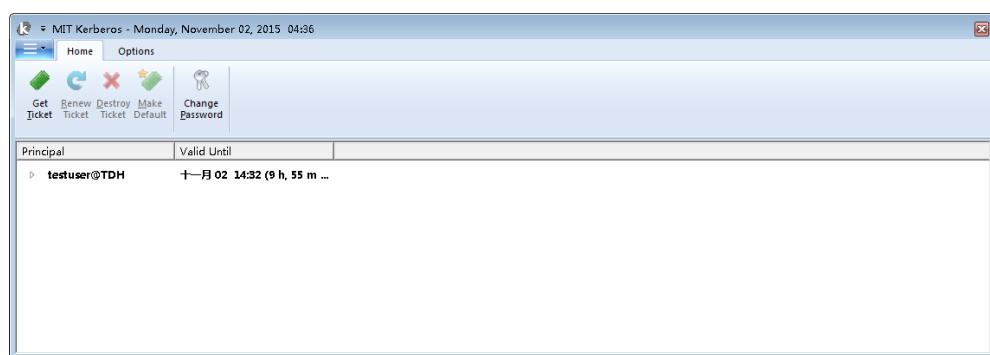
b. 点击“Get Ticket”，程序会弹出下面窗口：



c. 在“Principal”和“Password”中分别填入您用户的principal和对应密码，点击“OK”完成。



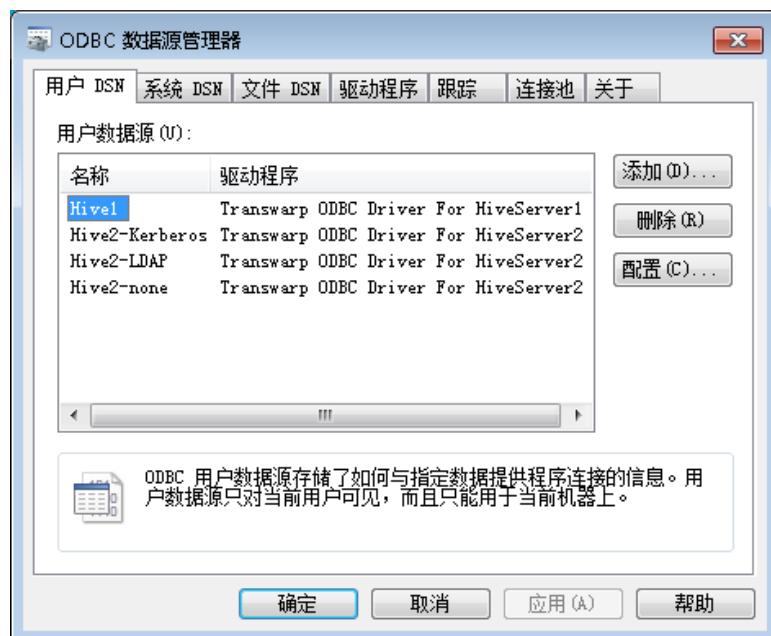
d. 认证成功后您会在程序中看到一张有效TGT：



现在再次测试连接，您会看到连接成功的信息：

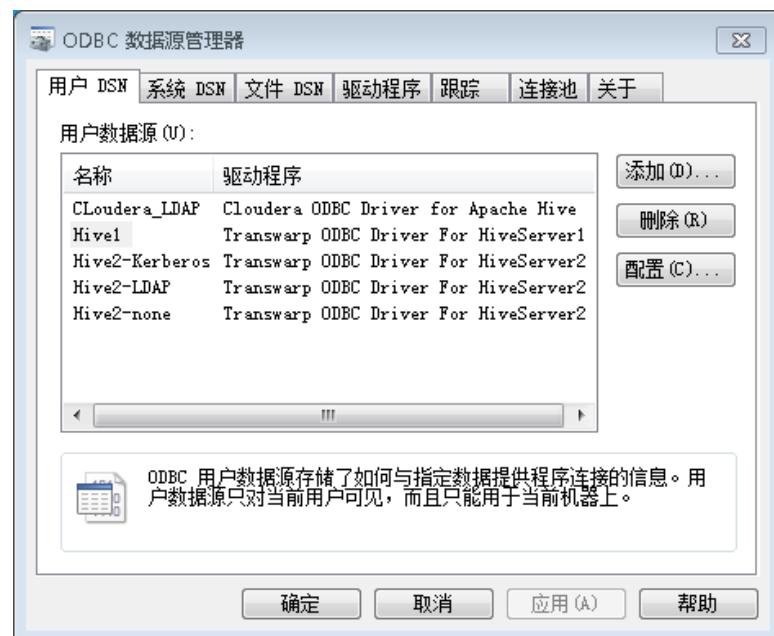


5. 现在ODBC Data Source Administrator中将显示我们刚刚添加的Hive2-kerberos这个DSN。

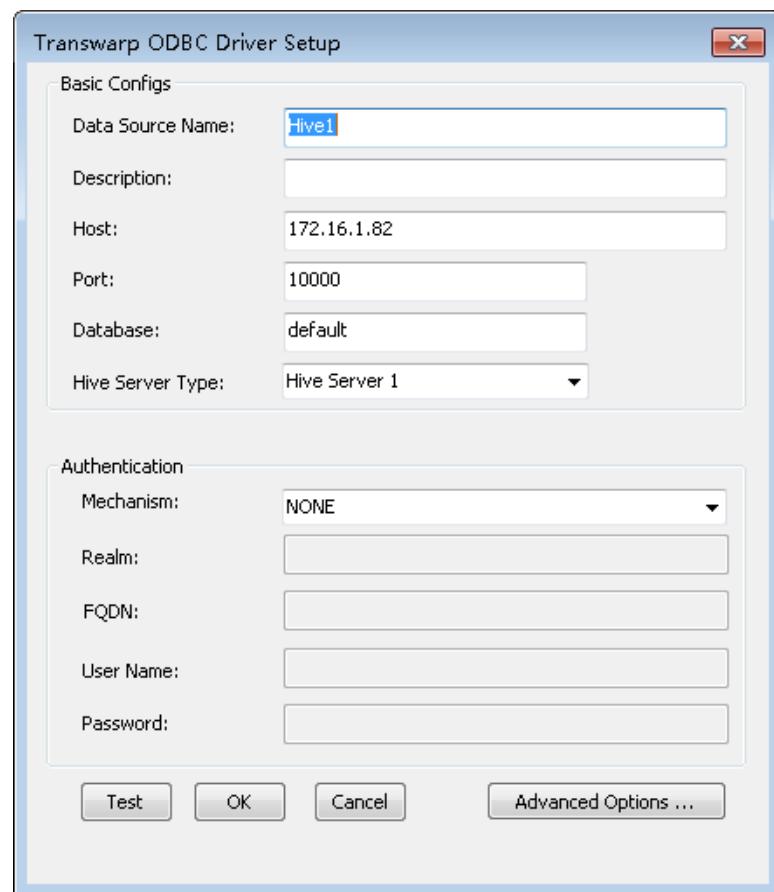


9.2.3.5. DSN高级选项

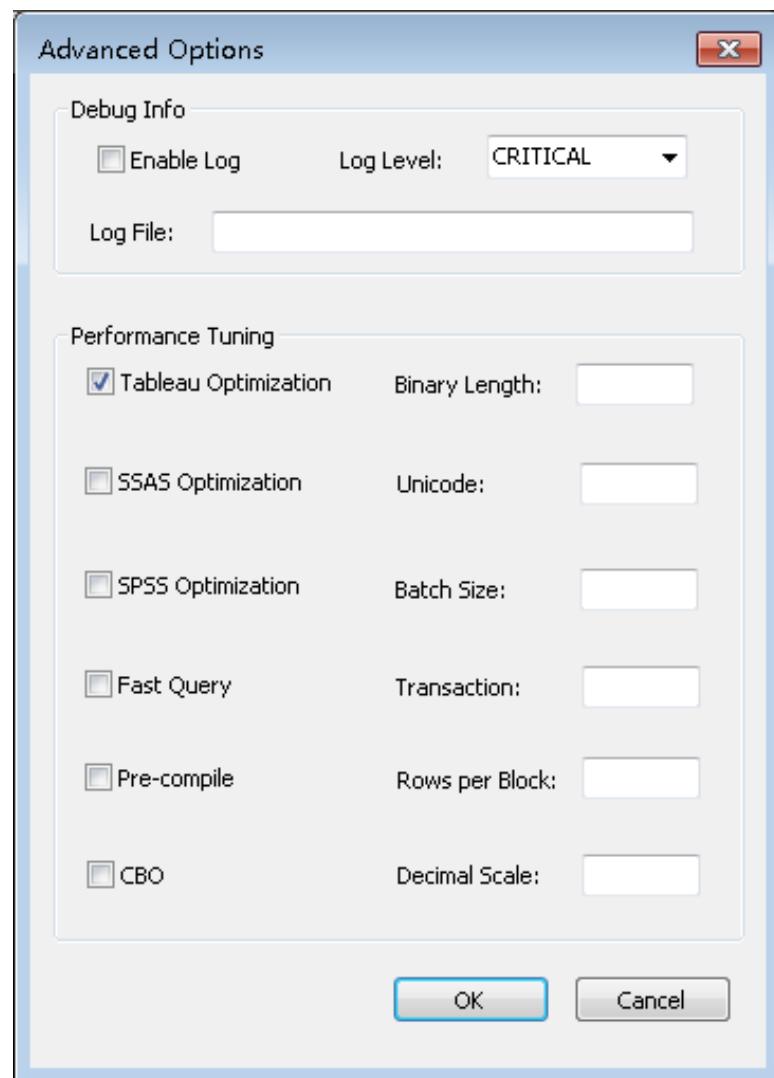
1. 在ODBC数据源管理器中任意选择一个驱动为Transwarp ODBC Driver (For HiveServer1和HiveServer2都可)的DSN。



2. 双击这个DSN，进入它的配置窗口：

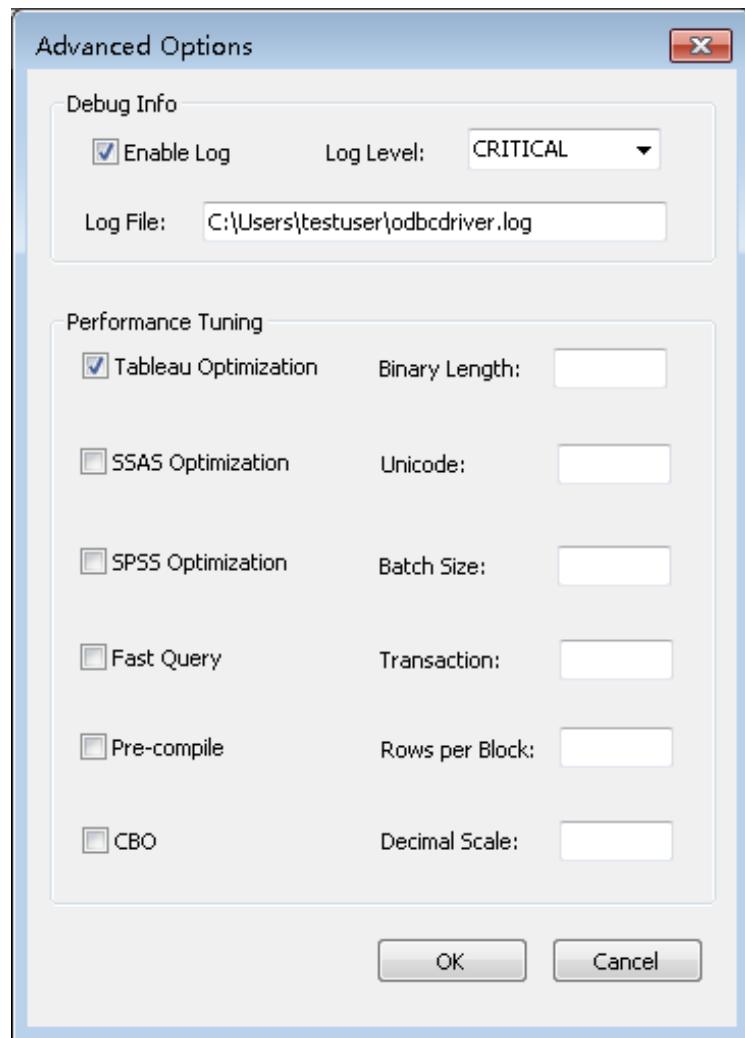


3. 点击右下角的“Advanced Options”对它的高级选项进行设置：



其中：

- a. 在“Debug Info”下您可以选择是否输出驱动日志。勾选“Enable Log”会开启日志的输出。在Log Level中您可以选择日志的级别。在“Log File”下您需要输入您日志文件的绝对路径。比如：



- b. 在“Performance Tuning”下您可以选择为不同目标设计的优化选项。比如勾选“Tableau Optimization”可以让使用Tableau连接Inceptor速度加快。

9.3. 在Linux系统中配置ODBC环境

9.3.1. 配置前的准备

1. 下载ODBC Driver Manager的安装包：

```
wget ftp://ftp.unixodbc.org/pub/unixODBC/unixODBC-2.3.4.tar.gz
```

您也可以用除了wget以外的方法将这个安装包下载到本地。

您还需要有我们提供的Linux版本的ODBC驱动Transwarp Linux ODBC。如果没有，请和我们联系。

9.3.2. 安装ODBC Driver Manager

1. 进入您下载了ODBC Driver Manager的目录，依次执行下面指令：

```

tar -xvzf unixODBC-2.3.4.tar.gz
cd unixODBC-2.3.4/
./configure --sysconfdir=/etc
sudo make
sudo make install

```

此时unixODBC将被安装在/usr/local/lib 和 /usr/local/bin 下面。

2. 执行下面指令查看LD_LIBRARY_PATH和PATH这两个环境变量:

```

# echo $LD_LIBRARY_PATH
# echo $PATH

```

请确保LD_LIBRARY_PATH 包含了/usr/local/lib 路径, PATH包含了/usr/local/bin 路径。如果没有, 请执行下面两个指令:

```

# export LD_LIBRARY_PATH=/usr/local/lib:${LD_LIBRARY_PATH}
# export PATH=/usr/local/bin:${PATH}

```

3. 执行下面指令测试ODBC Driver Manager, 确保 DRIVERS 和 SYSTEM DATA SOURCES 如下所示:

```

# odbcinst -j
unixODBC 2.3.4
DRIVERS.....: /etc/odbcinst.ini ①
SYSTEM DATA SOURCES: /etc/odbc.ini ②
FILE DATA SOURCES...: /etc/ODBCDataSources
USER DATA SOURCES...: /home/ke/.odbc.ini
SQLULEN Size....: 8
SQLLEN Size.....: 8
SQLSETPOSIROW Size.: 8

```

① DRIVERS需要显示 /etc/odbcinst.ini。

② SYSTEM DATA SOURCES 需要显示 /etc/odbc.ini。

9.3.3. 安装ODBC驱动

将ODBC驱动Transwarp Linux ODBC的压缩文件解压, 然后进入解压后的目录Inceptor Linux Odbc 4.3。目录下有这些文件:

```

# ls
# inceptor-odbc-centos64-4.3-1.el6.x86_64.rpm  inceptor-odbc-centos65-4.3-1.el6.x86_64.rpm  inceptor-
odbc-opensuse11-4.3-1.el6.x86_64.rpm

```

它们分别适用的系统如下:

文件名	适用系统
inceptor-odbc-centos64-4.3-1.el6.x86_64.rpm	CentOS 6.4
inceptor-odbc-centos65-4.3-1.el6.x86_64.rpm	CentOS 6.5
inceptor-odbc-opensuse11-4.3-1.el6.x86_64.rpm	OpenSUSE/Ubuntu(先用alien转成deb)

9.3.3.1. 在Ubuntu系统中安装

- 在Ubuntu系统中，我们可以将在SUSE系统安装的驱动inceptor-odbc-opensuse11-4.3-1.el6.x86_64.rpm先用alien转换成一个deb包，然后安装。如果您没有安装alien，您可以执行下面指令安装：

```
# sudo apt-get install alien
```

- 有了alien以后，将inceptor-odbc-opensuse11-4.3-1.el6.x86_64.rpm转换成deb包：

```
# sudo alien inceptor-odbc-opensuse11-4.3-1.el6.x86_64.rpm
inceptor-odbc-opensuse11_4.3-2_amd64.deb generated
```

- 安装inceptor-odbc-opensuse11_4.3-2_amd64.deb

```
# sudo dpkg -i inceptor-odbc-opensuse11_4.3-2_amd64.deb
```

9.3.3.2. 在CentOS系统中安装

执行下面指令：

```
rpm -ivh <package_name>
```

在<package_name>处根据您的CentOS系统的版本选择inceptor Linux Odbc 4.3目录下的inceptor-odbc-centos64-4.3-1.el6.x86_64.rpm或者inceptor-odbc-centos65-4.3-1.el6.x86_64.rpm。

您可能会遇到下面的状况：

```
# rpm -ivh inceptor-odbc-centos64-4.3-1.el6.x86_64.rpm
error: Failed dependencies:
libodbcinst.so.2()(64bit) is needed by inceptor-odbc-centos64-4.3-1.el6.x86_64
```

解决方法是加上--nodeps参数忽略依赖关系：

```
# rpm -ivh inceptor-odbc-centos64-4.3-1.el6.x86_64.rpm --nodeps
Preparing... ################################################ [100%, align=center]
1:inceptor-odbc-centos64 ################################################ [100%, align=center]
```

安装完成。

9.3.3.3. 在SUSE系统中安装

执行rpm -ivh安装inceptor Linux Odbc 4.3目录下的inceptor-odbc-opensuse11-4.3-1.el6.x86_64.rpm：

```
# rpm -ivh inceptor-odbc-opensuse11-4.3-1.el6.x86_64.rpm
```

您可能会遇到下面的状况：

```
# rpm -ivh inceptor-odbc-opensuse11-4.3-1.el6.x86_64.rpm
error: Failed dependencies:
libodbcinst.so.2()(64bit) is needed by inceptor-odbc-opensuse11-4.3-1.el6.x86_64
rpmlib(FileDigests) <= 4.6.0-1 is needed by inceptor-odbc-opensuse11-4.3-1.el6.x86_64
```

如果只是出现上述依赖问题，原因是系统rpm 太老。解决方法是加上--nodeps参数来直接忽略依赖关系：

```
# rpm -ivh inception-odbc-opensuse11-4.3-1.el6.x86_64.rpm --nodeps
Preparing... ################################################ [100%, align=center]
1:inception-odbc-opensuse1 ################################################ [100%, align=center]
```

安装完成。

9.3.4. 配置ODBC驱动

打开 /etc/odbcinst.ini文件：

```
# sudo vim /etc/odbcinst.ini
```

这个文件将需要包含ODBC Driver的名称、so文件路径、是否输出log以及log路径等一系列信息。一份odbcinst.ini样例文件如下：

```
[ODBC] ①
TraceFile = /tmp/odbc-driver-manager.log
Trace = Yes
[Transwarp ODBC Driver For HiveServer1] ②
Description=Transwarp ODBC Driver For HiveServer1
Driver=/lib/libODBC4HiveServer1.so
Trace=0
TraceLevel=1
TraceFile=/tmp/hs1-odbc-test.log
[Transwarp ODBC Driver For HiveServer2] ③
Description=Transwarp ODBC Driver For HiveServer2
Driver=/lib/libODBC4HiveServer2.so
Trace=1
TraceLevel=1
TraceFile=/tmp/hs2-odbc-test.log
```

① 从这里开始的两项配置ODBC Manager：

- **TraceFile** 指定ODBC Manager log的路径
- **Trace** 指定是否输出ODBC Manager log。Yes为输出，No为不输出。

② 指定一个Driver的名称，该行之后的6行信息将配置这个Driver。我们提供两个Driver，分别名为Transwarp ODBC Driver For HiveServer1和Transwarp ODBC Driver For HiveServer2。Transwarp ODBC Driver For HiveServer1用于ODBC连接InceptorServer 1；Transwarp ODBC Driver For HiveServer2用于ODBC连接InceptorServer 2。这里我们指定了Transwarp ODBC Driver For HiveServer1。所以，下面6行信息配置的是Transwarp ODBC Driver For HiveServer1：

- **Description** 是对Transwarp ODBC Driver For HiveServer1的描述，选填。
- **Driver** 指定Transwarp ODBC Driver For HiveServer1对应的so文件。Transwarp ODBC Driver For HiveServer1的默认安装路径为/lib/libODBC4HiveServer1.so
- **Trace** 指定是否输出Transwarp ODBC Driver For HiveServer1的log，1为输出，0为不输出。
- **TraceLevel** 指定输出Transwarp ODBC Driver For HiveServer1的log的等级：0为critical，1为error，2为debug，3为info。
- **TraceFile** 指定输出Transwarp ODBC Driver For HiveServer1的log的路径。

③ 这里我们指定了Transwarp ODBC Driver For HiveServer2。所以，下面6行信息配置的是Transwarp ODBC Driver For HiveServer2。

- **Description** 是对Transwarp ODBC Driver For HiveServer2的描述，选填。

- **Driver** 指定Transwarp ODBC Driver For HiveServer2对应的so文件，Transwarp ODBC Driver For HiveServer2的默认安装路径为/lib/libODBC4HiveServer2.so
- **Trace** 指定是否输出Transwarp ODBC Driver For HiveServer2的log，1为输出，0为不输出。
- **TraceLevel** 指定输出Transwarp ODBC Driver For HiveServer2的log的等级：0为critical，1为error，2为debug，3为info。
- **TraceFile** 指定输出Transwarp ODBC Driver For HiveServer2的log的路径。

9.3.5. 配置DSN

打开 /etc/odbc.ini文件：

```
# sudo vim /etc/odbc.ini
```

在文件中您可以添加DSN，一个DSN配置例子如下：

```
[Transwarp-hs2-none] // DSN名称
Description = No authorization // DSN描述，选填。
Driver = Transwarp ODBC Driver For HiveServer2 ①
Server = linux-1 // 服务所在节点的hostname，选填。
Hive = Hive Server 2 ②
Host = 10.0.0.211 ③
Port = 10000 ④
Database = default ⑤
Trace = 1 ⑥
TraceFile = stderr ⑦
Mech = NONE ⑧
User = none
Password = none
Realm = none
FQDN = none
```

- ① 该DSN所用的Driver的名称。如果该DSN连接到InceptorServer 1，您需要在这里填写Transwarp ODBC Driver For HiveServer1；如果该DSN连接到InceptorServer 2，您需要在这里填写Transwarp ODBC Driver For HiveServer2。
- ② 选择Inceptor Server类型。如果使用InceptorServer 1，填写Hive Server 1；如果选择InceptorServer 2，填写Hive Server 2。
- ③ DSN要连接的Inceptor所在节点的ip。
- ④ Inceptor所在的端口，默认设置在10000
- ⑤ 该DSN要连接Inceptor的数据库。
- ⑥ 是否输出该DSN的log。1为输出，0为不输出。
- ⑦ 输出DSN log的路径。
- ⑧ 从该行开始的五项进行DSN的认证配置：
 - **Mech** 指定DSN要连接的Inceptor所用的认证机制，一共有三种：NONE（没有认证），KERBEROS和LDAP。
 - **User** 指认认证所用的用户名。如果 **Mech** 选择了NONE，**User** 则不填。
 - **Password** 指认认证所用的密码。如果 **Mech** 选择了NONE，**Password** 则不填。
 - **Realm** 指定Kerberos的域，这个信息您只需要在 **Mech** 选择了KERBEROS的情况下提供。
 - **FQDN** 指定DSN要连接的Inceptor服务所在节点的hostname，这个信息您只需要在 **Mech** 选择了KERBEROS的情况下提供。

下面的例子配置了四个DSN:

```
[no-auth-tw-node1212] ①
Description = No authorization
Driver = Transwarp ODBC Driver For HiveServer2
Server = tw-node1212
Hive = Hive Server 2
Host = 172.16.1.212
Port = 10000
Database = default
Trace = 1
TraceFile = stderr
User = none
Password = none
Mech = NONE

[ldap-hive-tw-node119] ②
Description = LDAP
Driver = Transwarp ODBC Driver For HiveServer2
Server = tw-node119
Hive = Hive Server 2
Host = 172.16.1.119
Port = 10000
Database = default
Trace = 1
TraceFile = stderr
User = hive
Password = 123
Mech = LDAP

[kerberos-alice-baogang2] ③
Description = Kerberos
Driver = Transwarp ODBC Driver For HiveServer2
Server = baogang2
Hive = Hive Server 2
Host = 172.16.1.87
Port = 10000
Database = default
Trace = 1
TraceFile = stderr
User = alice
Password = 123456
Mech = Kerberos
Realm = tdh
FQDN = baogang2

[no-auth-test-02] ④
Description = test for hive1
Driver = Transwarp ODBC Driver For HiveServer1
Server = test-02
Hive = Hive Server 1
Host = 172.16.1.82
Port = 10000
Database = default
Trace = 1
TraceFile = stderr
User = NONE
Password = NONE
Mech = NONE
```

- ① 这个DSN连接到一个InceptorServer 2服务，选用Transwarp ODBC Driver For HiveServer2驱动。该服务没有认证机制（Mech = NONE），所以不需要用户名（User = none）和密码（Password = none）。
- ② 这个DSN连接到一个InceptorServer 2服务，选用Transwarp ODBC Driver For HiveServer2驱动。该服务使用LDAP认证，用户名为hive，密码为123。
- ③ 这个DSN连接到一个InceptorServer 2服务，选用Transwarp ODBC Driver For HiveServer2驱动。该服务使用Kerberos认证，用户名为alice，密码为123456。如果您想要在您的计算机上连接到一个使用Kerberos认证的Inceptor服务，您还需要在您的计算机上配置Kerberos。请参考本章末尾的“在Linux系统中配置Kerberos”内容。
- ④ 这个DSN连接到一个InceptorServer 1服务，选用Transwarp ODBC Driver For HiveServer1驱动。该服务没有认证机制，所以不需要用户名和密码。

9.3.6. SASL插件的安装

ODBC连接Inceptor的认证通过SASL来完成，您在使用ODBC连接时有可能会看到下面的报错：

```
SASL Other: No worthy mechs found
```

这是因为您的系统需要对应认证方式（简单（plain），LDAP或者Kerberos）的SASL插件，而这些插件目前在您的系统中还没有安装。这些插件根据您操作系统的不同名称也不同。

9.3.6.1. Ubuntu系统

在Ubuntu系统中，您需要的SASL插件如下：

- sasl2-bin
- libsasl2-2
- libsasl2-modules
- libsasl2-modules-gssapi-mit

执行下面指令安装：

```
# sudo apt-get install sasl2-bin libsasl2-2 libsasl2-modules libsasl2-modules-gssapi-mit
```

9.3.6.2. CentOS系统

在CentOS系统中，plain或者LDAP认证需要的SASL插件为cyrus-sasl-plain，执行下面指令安装：

```
# sudo yum install cyrus-sasl-plain
```

在CentOS系统中，Kerberos认证需要的SASL插件为cyrus-sasl-gssapi，执行下面指令安装：

```
# sudo yum install cyrus-sasl-gssapi
```

9.3.6.3. SUSE系统

在SUSE系统中，plain或者LDAP认证需要的SASL插件为cyrus-sasl-plain，执行下面指令安装：

```
# sudo zypper install cyrus-sasl-plain
```

在SUSE系统中，plain或者LDAP认证需要的对应Kerberos的SASL插件为cyrus-sasl-gssapi，执行下面指令安装：

```
# sudo zypper install cyrus-sasl-gssapi
```

9.3.7. 测试连接

执行下面指令查看一个DSN是否能够连接成功：

```
# isql -v <dsn_name>
```

这里dsn_name出填写您想要连接的DSN。下面我们分别连接上面配置的no-auth-tw-node1212、ldap-hive-tw-node119和no-auth-test-02。看到下面的输出说明连接成功。

```
# isql -v no-auth-tw-node1212
default,172.16.1.212,10000,0,NONE,
+-----+
| Connected!
| sql-statement
|   help [tablename]
|   quit
+-----+
SQL>

# isql -v ldap-hive-tw-node119
default,172.16.1.119,10000,0,LDAP,
+-----+
| Connected!
| sql-statement
|   help [tablename]
|   quit
+-----+
SQL>

# isql -v no-auth-test-02
+-----+
| Connected!
| sql-statement
|   help [tablename]
|   quit
+-----+
SQL>
```

要连接到Kerberos认证的DSN，您需要先在您的计算机上获取一张Kerberos TGT（在这之前，您需要现在您的计算机上配置Kerberos，这部分内容请参考下一章“在Linux系统中配置Kerberos”），否则将无法通过认证导致无法连接Inceptor：

```
# isql -v kerberos-alice-baogang2
default,172.16.1.87,10000,0,Kerberos,baogang2
SASL Other: GSSAPI Error: Unspecified GSS failure. Minor code may provide more information (No
Kerberos credentials available)
[ISQL]ERROR: Could not SQLConnect
```

要获取一张Kerberos TGT，您需要执行kinit指令，系统会提示您输入密码：

```
# kinit alice
Password for alice@TDH:
```

您可以用klist指令查看是否成功获取TGT，如果成功，您会看到类似下面的输出：

```
# klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: alice@TDH

Valid starting     Expires            Service principal
12/18/2015 17:00:19  12/19/2015 03:00:19  krbtgt/TDH@TDH
    renew until 12/19/2015 17:00:20
```

下面您可以测试您ODBC的连接（连接前请确保计算机上存有有效的TGT）：

```
# isql -v kerberos-alice-baogang2
default,172.16.1.87,10000,0,Kerberos,baogang2
+-----+
| Connected!
| sql-statement
| help [tablename]
| quit
+-----+
SQL>
```

9.4. 在Linux系统中安装和配置Kerberos客户端

如果您想要在您的计算机上用ODBC访问用Kerberos认证的Inceptor服务，您需要安装Kerberos客户端，通过Kerberos客户端获取Kerberos TGT从而通过TDH集群上的Kerberos认证。

9.4.1. 在Ubuntu系统中安装和配置Kerberos客户端

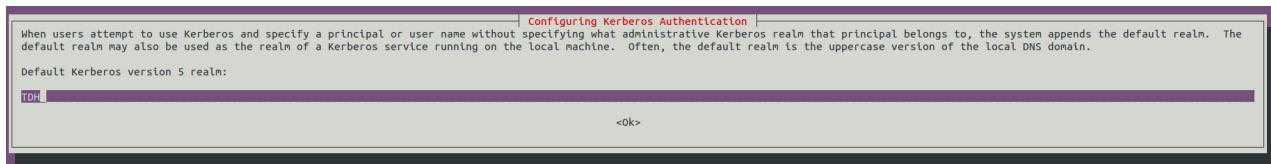
- 执行下面的指令安装Kerberos客户端：

```
# sudo apt-get install krb5-user libpam-krb5 libpam-ccreds auth-client-config
```

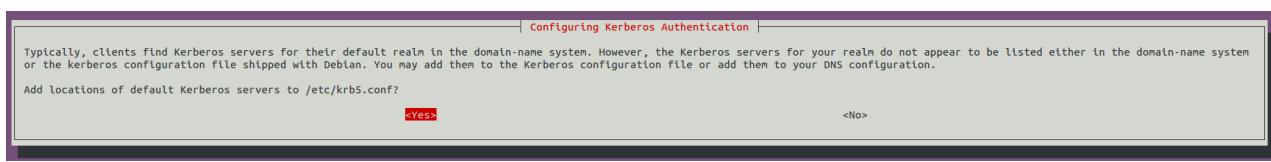
- 执行下面指令配置Kerberos客户端：

```
sudo dpkg-reconfigure krb5-config
```

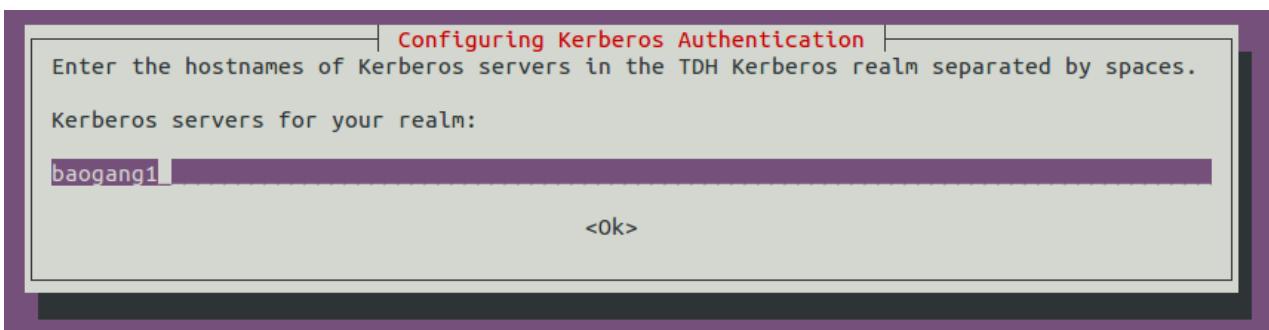
您会看到下面的窗口：



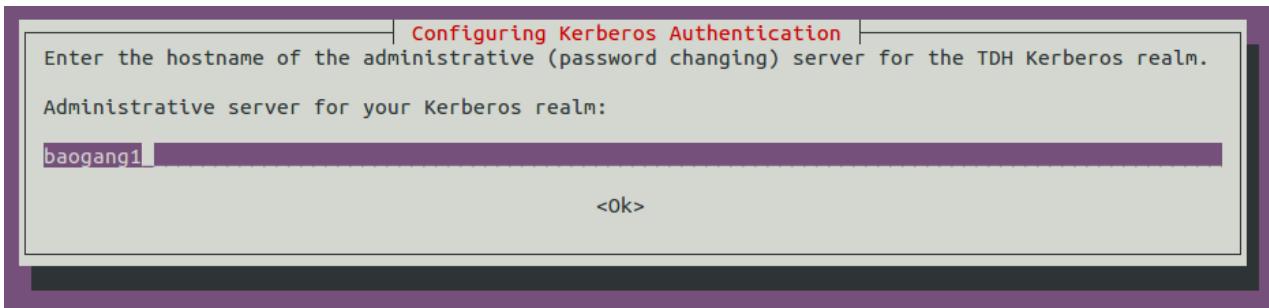
您需要在这里填写您的集群的Kerberos域。除非另外设置，集群的Kerberos域一般为TDH，您也可以向您系统的管理员咨询具体信息。按回车继续。



这里系统提示您提供集群中的Kerberos server和Kerberos admin server的信息。您需要向您系统的管理员咨询，获取该信息。按回车继续，在下面窗口您需要输入您集群中Kerberos server所在节点的hostname：



完成后按回车继续，在下面窗口您需要输入您集群中Kerberos admin server所在节点的hostname:



完成后按回车，完成设置。

这两步设置后，您的/etc krb5.conf文件中将加入下面的信息：

```
[libdefaults]
    default_realm = tdh
[realms]
    TDH = {
        kdc = baogang1
        admin_server = baogang1
    }
```

由于这里Kerberos server所在节点和Kerberos admin server所在节点都是由节点的hostname指定的，您需要在您计算机上的/etc/hosts中进行相应配置。

配置/etc/hosts：



vim /etc/hosts

在文件中添加ip/hostname对，中间用空格隔开。例如我们将在本地的/etc/hosts中添加：

172.16.1.86 baogang1

- 现在您的Kerberos客户端已经安装和配置完成，您可以执行kinit指令获取一张Kerberos TGT。要获取Kerberos TGT，您需要有Kerberos principal和对应的密码，如果没有该信息，您需要向您集群的管理员获取。

```
# kinit alice
Password for alice@TDH:
```

您可以用klist指令查看是否成功获取TGT，如果成功，您会看到类似下面的输出：

```
# klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: alice@TDH

Valid starting     Expires            Service principal
12/18/2015 17:00:19  12/19/2015 03:00:19  krbtgt/TDH@TDH
renew until 12/19/2015 17:00:20
```

- Kerberos客户端配置完成后，您还需要配置安装SASL GSSAPI插件才能在使用ODBC连接Inceptor时通过Kerberos认证。执行下面指令：

```
# sudo apt-get install sasl2-bin libsasl2-2 libsasl2-modules libsasl2-modules-gssapi-mit
```

- 下面您可以测试您ODBC的连接（连接前请确保计算机上存有有效的TGT）：

```
# isql -v kerberos-alice-baogang2
default,172.16.1.87,10000,0,Kerberos,baogang2
+-----+
| Connected!
| sql-statement
| help [tablename]
| quit
+-----+
SQL>
```

9.4.2. 在CentOS或者SUSE系统中安装和配置Kerberos客户端

如果您本地使用CentOS系统或者SUSE系统，那么您可以直接使用我们提供的TDH安装包中包含的Kerberos安装工具，您可能需要和您的集群管理员联系获取相关文件。

- TDH安装包一般在您集群的管理节点上的一个名为transwarp的目录下。transwarp目录所在位置和安装TDH时的选择有关，您可能需要和您的集群管理员联系获取相关信息。安装Kerberos客户端所使用的工具是transwarp/support/script下的kerberos-client-install.sh脚本。将这个脚本拷贝到您计算机上，然后运行：

```
# sh kerberos-client-install.sh -s <kdc_ip/hostname>
```

这里<kerberos_server_ip/hostname>处您需要提供您集群上kdc所在的节点的ip或者hostname。

如果您使用hostname指定kdc所在的节点，您需要在您计算机上的/etc/hosts中进行相应配置。

配置/etc/hosts：



```
# vim /etc/hosts
```

在文件中添加ip/hostname对，中间用空格隔开。例如我们将在本地的/etc/hosts中添加：

```
172.16.1.86 baogang1
```

- 您可以尝试用kinit获取一张TGT来查看配置是否成功：

```
# kinit alice
Password for alice@TDH:
```

您可以用klist指令查看是否成功获取TGT，如果成功，您会看到类似下面的输出：

```
# klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: alice@TDH

Valid starting     Expires            Service principal
12/18/2015 17:00:19  12/19/2015 03:00:19  krbtgt/TDH@TDH
    renew until 12/19/2015 17:00:20
```

- Kerberos客户端配置完成后，您还需要配置安装SASL GSSAPI插件才能在使用ODBC连接Inceptor时通过Kerberos认证。如果您的操作系统是CentOS，执行：

```
# sudo yum install cyrus-sasl-gssapi
```

如果您的操作系统是SUSE，执行：

```
# sudo zypper install cyrus-sasl-gssapi
```

- 下面您就可以测试您ODBC的连接（连接前请确保计算机上存有有效的TGT）：

```
# isql -v kerberos-alice-baogang2
default,172.16.1.87,10000,0,Kerberos,baogang2
+-----+
| Connected!
| sql-statement
| help [tablename]
| quit
+-----+
SQL>
```

9.5. 应用程序中的ODBC交互

下面是一份最简单的ODBC 代码。对于不同的认证模式。odbc的差别反应在DSN 建立上。

```
/*This program use $table_name as a sample, to show the usages of transwarp Hive ODBC Drivers, which
contains:
 *how to connect to database(with function connect);
 *how to create a table(with function create);
 *how to insert values into table(with function insert_into_table);
 *how to select from table and print result(with function select_all);
 *how to drop table(with function drop_table);
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#ifndef WIN32
#include <windows.h>
#endif
#include <sql.h>
#include <sqlext.h>
#ifndef WIN32
#define sprintf _snprintf
#endif
```

```

-----[SQLRENV env;
SQLHDBC conn;
SQLCHAR str[1024];
HSTMT hstmt = SQL_NULL_HSTMT; //handle statement
charable_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); \
    }

voidconnect(); //connect to database.
voidselect_all(); //select * from table and print result
voiddisconnect();

//connect to database.
voidconnect()
{
    SQLCHAR dsn[1024]; //dsn name
    SQLSMALLINT strl;
    sprintf((char*)dsn, sizeof(dsn), "DSN=your_dsn"); ①
    // 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");
}

//select * from table and print result
voidselect_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);
    chiselect_sql[100];
    sprintf(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 from table %s done.\n",table_name);
}

void disconnect(void)
{
    SQLRETURN rc;

    printf("disconnecting\n");
    rc =SQLDisconnect(conn);
}
-----
```

```

//printf("dis_connect_rc is :%d\n",rc);
if (!SQL_SUCCEEDED(rc))
{
print_diag("SQLDisconnect failed", SQL_HANDLE_DBC, conn);
result = FAILURE;
//exit(1);
return;
}

rc = SQLFreeHandle(SQL_HANDLE_DBC, conn);
if (!SQL_SUCCEEDED(rc))
{
print_diag("SQLFreeHandle(test_disconnect) failed", SQL_HANDLE_DBC, conn);
result = FAILURE;
//exit(1);
return;
}
conn = NULL;

rc = SQLFreeHandle(SQL_HANDLE_ENV, env);
if (!SQL_SUCCEEDED(rc))
{
print_diag("SQLFreeHandle(test_disconnect) failed", SQL_HANDLE_ENV, env);
result = FAILURE;
//exit(1);
return;
}
env = NULL;
}

voidmain()
{
connect(); //connect to database.
select_all(); //select * from table and print result
disconnect(); //drop table
}

```

- ① 这里将your_dsn改为您想要使用的DSN。DSN的创建和设置请参考本手册的“ODBC交互准备”一节。



注意，如果您的DSN连接到Kerberos认证的Inceptor server，您的客户端上应该有一张有效的TGT，否则将无法通过Kerberos认证，造成无法连接到Inceptor。关于如何获取TGT，请参考“ODBC交互准备”一节。

10. Inceptor外部工具连接手册

10.1. Inceptor外部工具连接手册一览

Inceptor提供了标准的JDBC和ODBC接口，让用户可以方便地使用BI工具连接到Inceptor。本手册中，我们将介绍如何通过第三方工具连接到Inceptor。

10.2. 使用JDBC的外部工具连接

10.2.1. 连接前的准备



进行本章操作前，您需要将Inceptor JDBC的驱动放在您本地的CLASSPATH中。您可以从Transwarp Manager上下载Inceptor JDBC驱动，细节请参考《Transwarp Data Hub运维手册》中的“InceptorSQL服务的配置”一节。

10.2.2. DbVisualiser连接Inceptor

本节我们将示范如何通过DbVisualiser分别连接：

1. 无需认证的InceptorServer 1
2. 需要认证的InceptorServer 2

使用DbVisualiser连接到Inceptor server，您需要：

1. 添加分别对应InceptorServer 1和InceptorServer 2的JDBC驱动；
2. 添加连接到您的Inceptor Server的alias；
3. 通过alias连接到您的Inceptor Server。

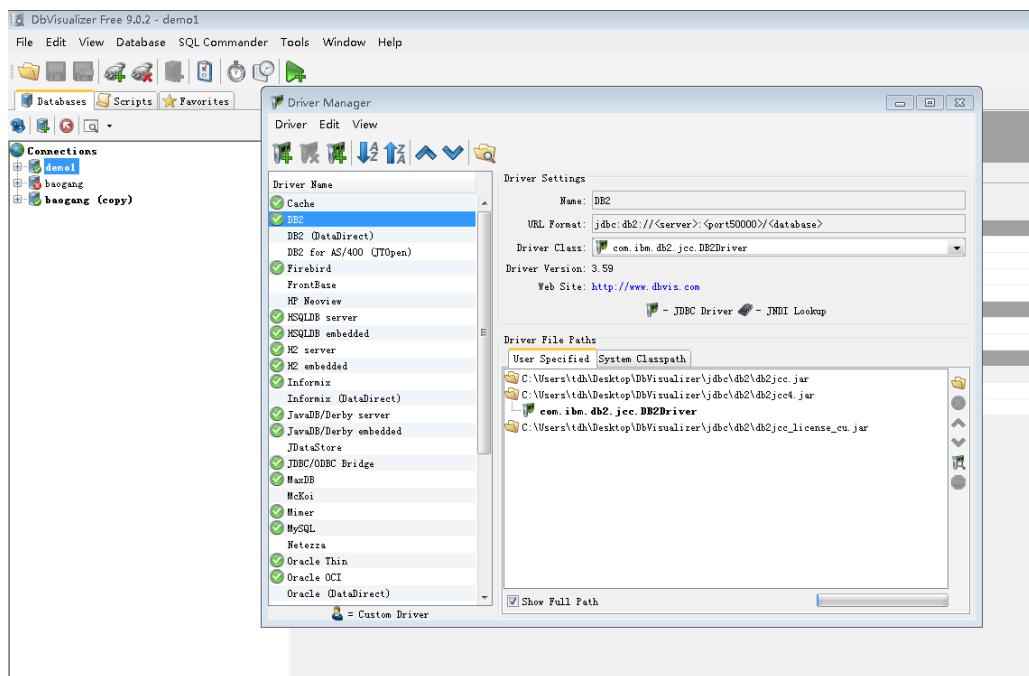
在开始前请确保您下载了驱动“inceptor-driver-4.5.jar”，并明确放置的位置。

10.2.2.1. 添加驱动

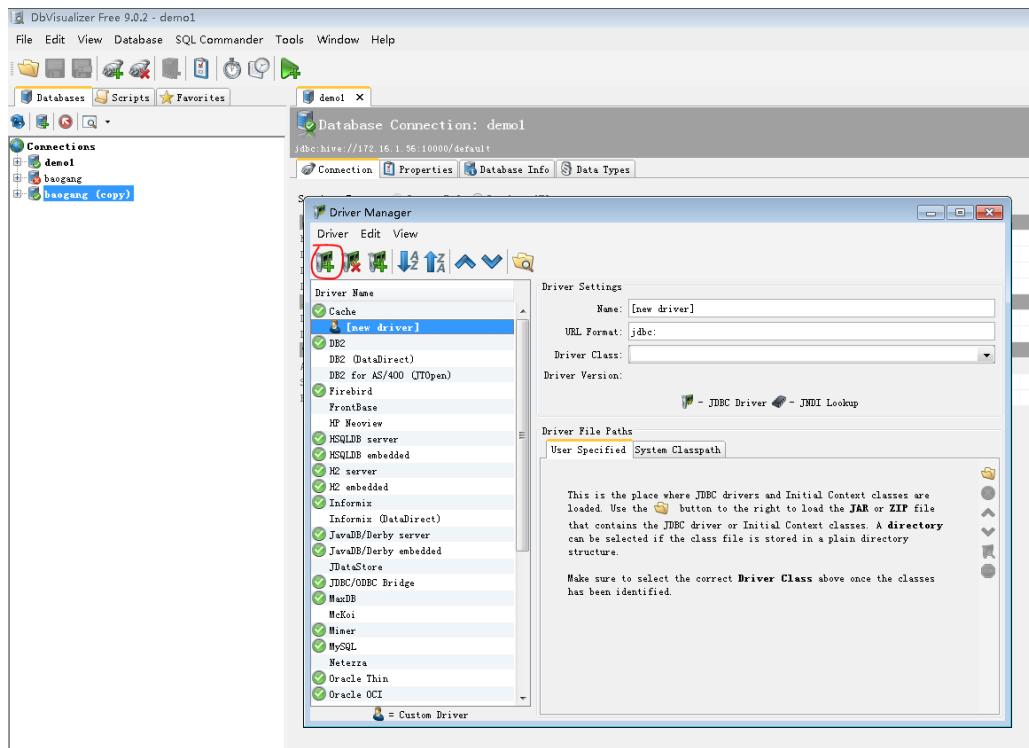
连接至InceptorServer 1和InceptorServer 2需要使用不同的驱动。

10.2.2.1.1. 连接InceptorServer 1的驱动

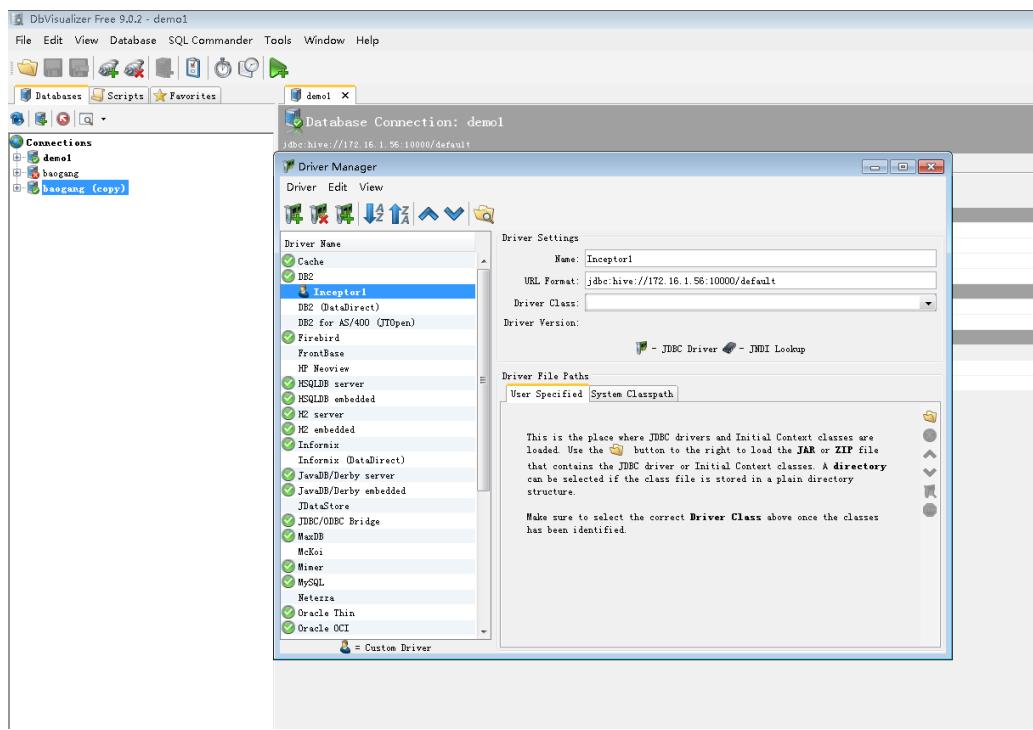
1. 打开DbVisualiser，选择“Tools-Driver Manager”，开始配置新驱动，您将看到如下窗口：



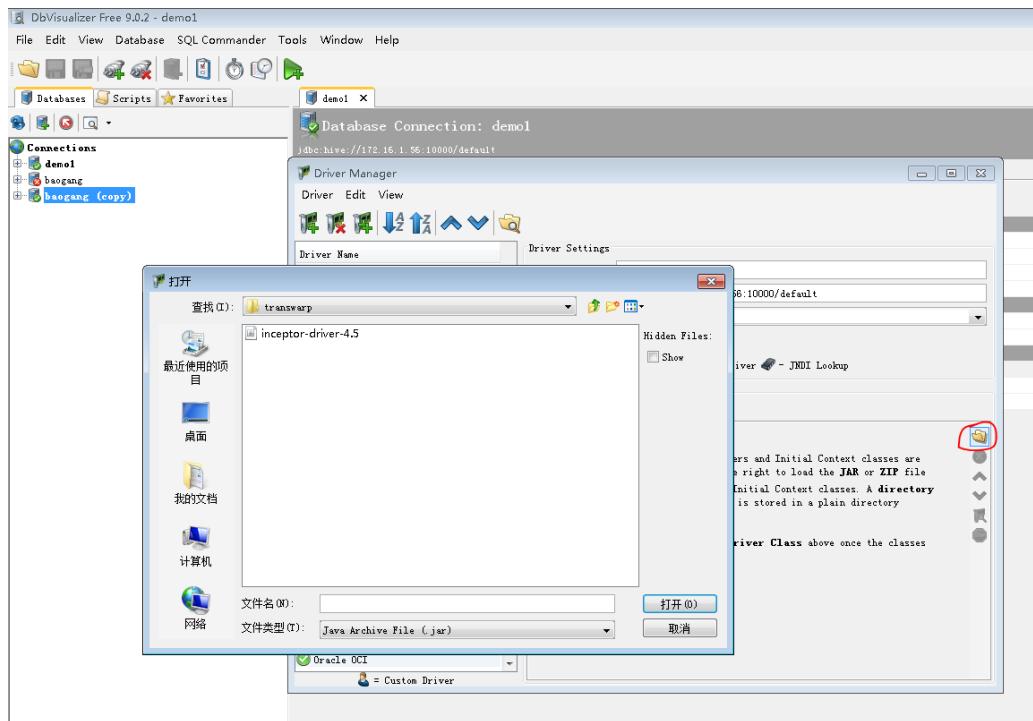
2. 点击Driver Manager左上角图标添加一个新驱动，如下所示：



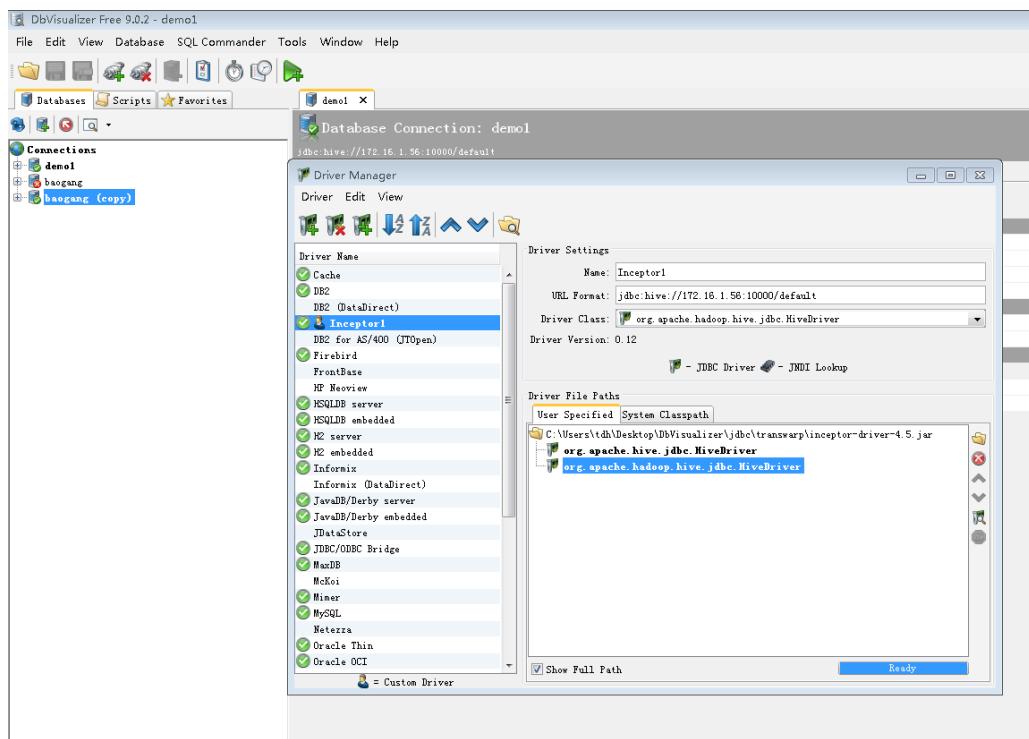
3. 请用户在显示窗口填写新驱动名称（自定义），以及URL连接字符串范例：



4. 点击窗口中代表文件夹的图标，在目录中找到inceptor-driver-4.5.jar的存放位置并选中：

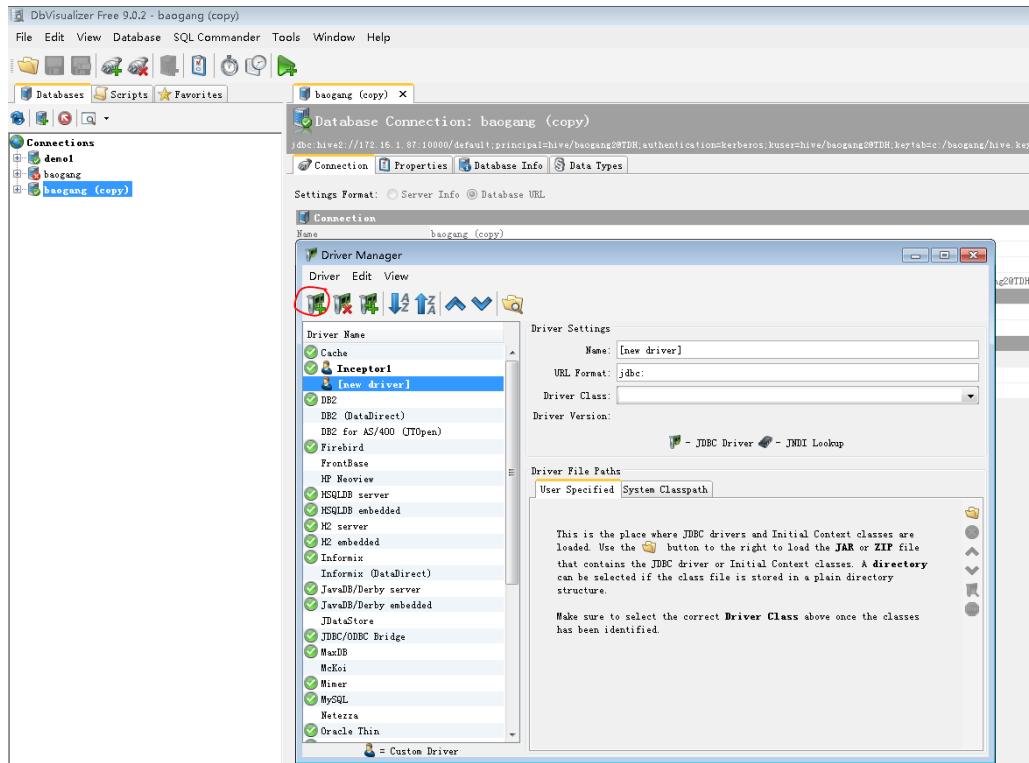


5. 等待inceptor-driver-4.5.jar加载完成后，在Driver Class中选择代表InceptorServer 1驱动的“org.apache.hadoop.jdbc.HiveDriver”。如果在驱动栏看见新驱动的名称前带有绿色对勾，即表示配置成功：

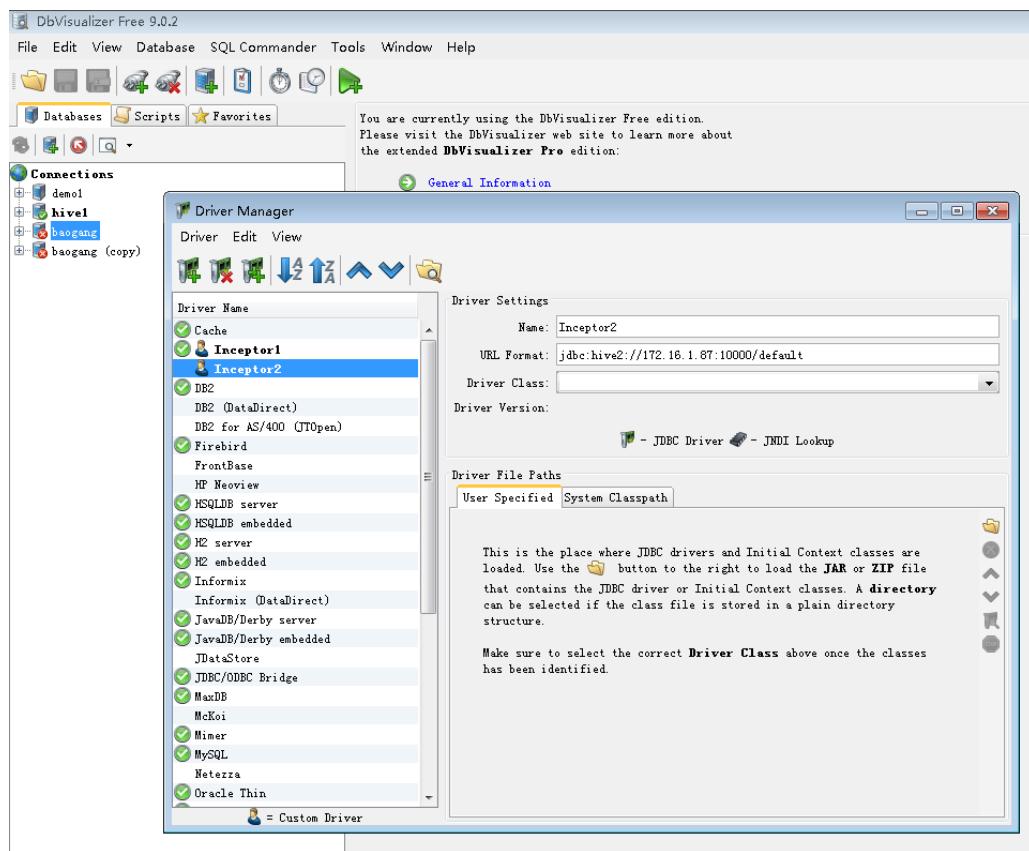


10.2.2.1.2. 连接InceptorServer 2的驱动

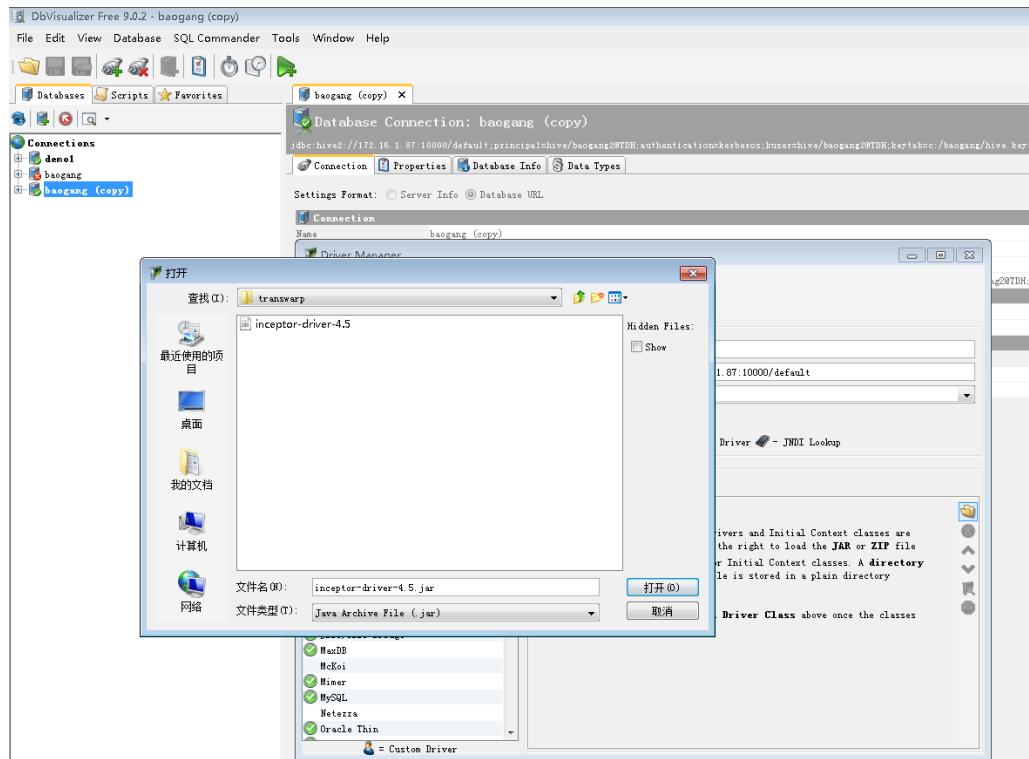
1. 打开DbVisualiser，选择“Tools–Driver Manager”，跳出Driver Manager窗口。点击Driver Manager左上角图标添加一个新驱动，出现如下所示界面：



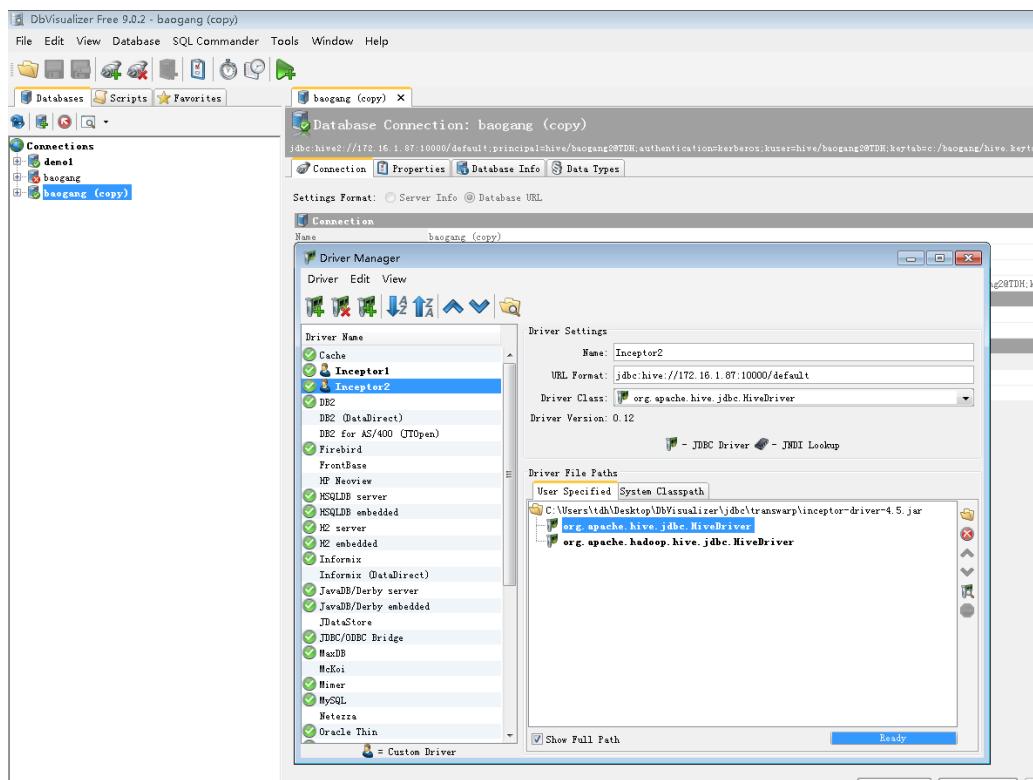
2. 请在显示窗口填写新驱动名称（自定义），以及URL连接字符串范例。点击窗口中代表文件夹的图标：



3. 在目录中找到inceptor-driver-4.5.jar的存放位置并选中：



4. 等待inceptor-driver-4.5.jar加载完成后，在Driver Class中选择代表InceptorServer 2驱动的“org.apache.hive.jdbc.HiveDriver”。如果在驱动栏看见新驱动的名称前带有绿色对勾，即表示配置成功：

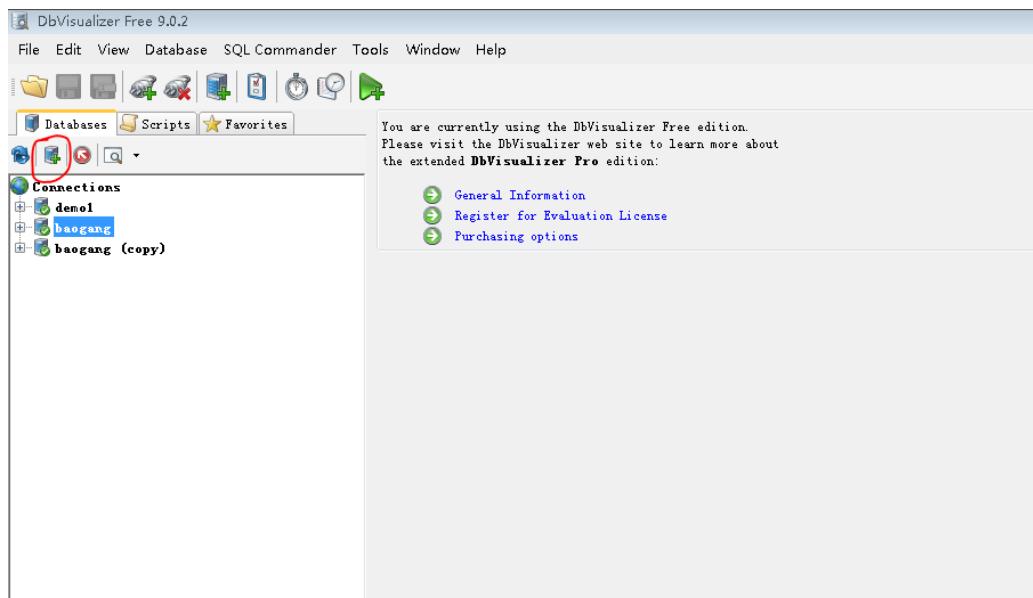


10.2.2.2. 创建连接

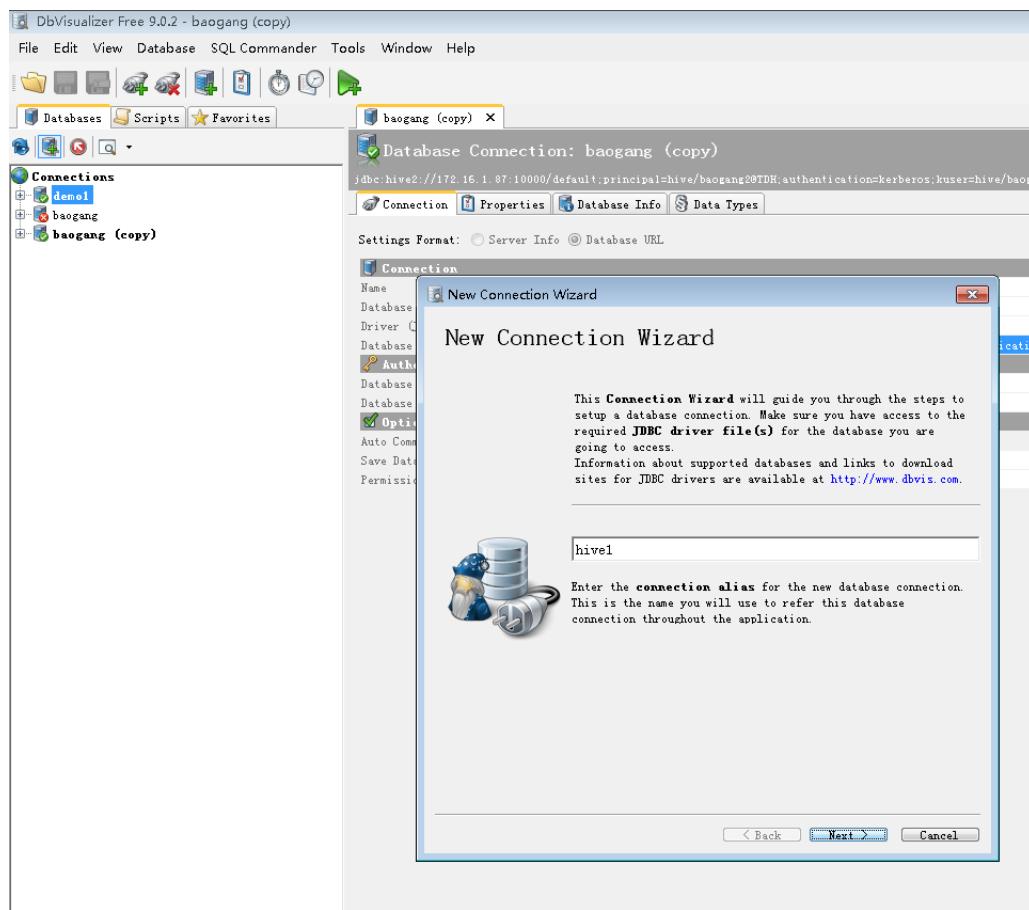
接下来将介绍如何用DbVisualiser连接至无需认证的InceptorServer 1以及需要认证的InceptorServer 2。

10.2.2.2.1. 无需认证的InceptorServer 1

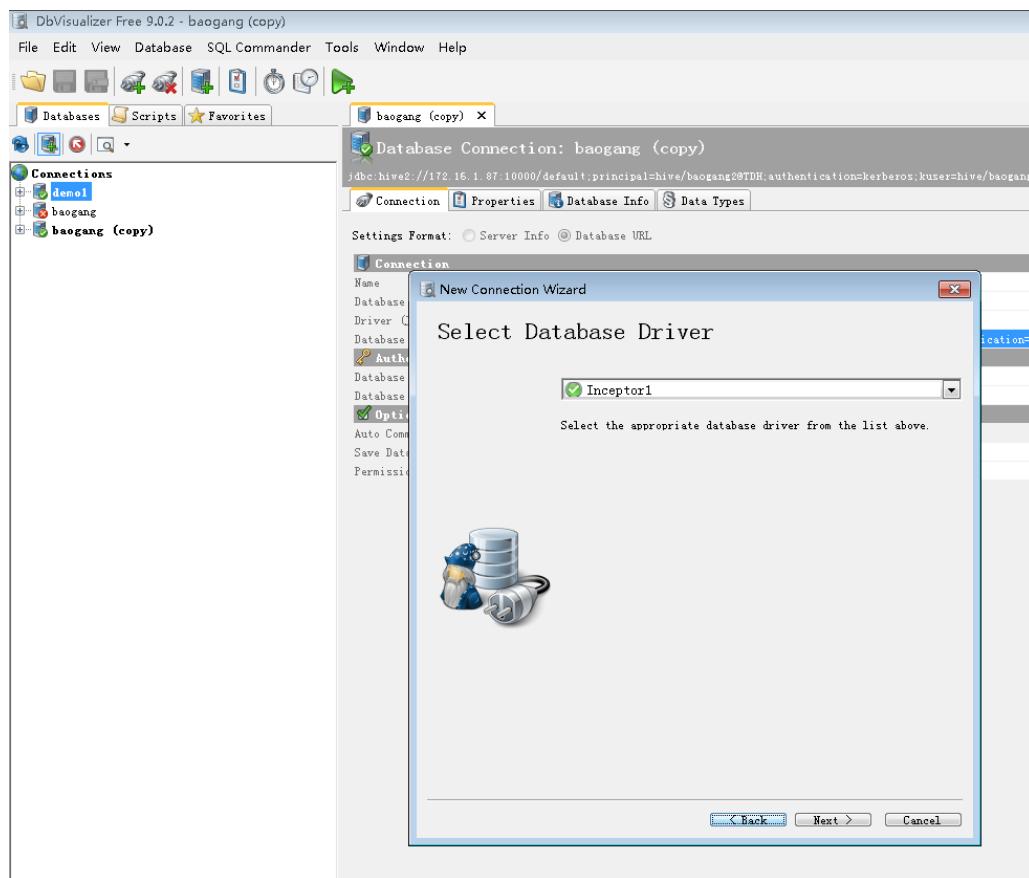
1. 如图点击DbVisualiser界面的左上角用红圈标注的图标，创建一个新的连接。



2. 出现如下界面，在该界面中为当前连接自定义一个名称，并点击“Next”，跳转至下一步。

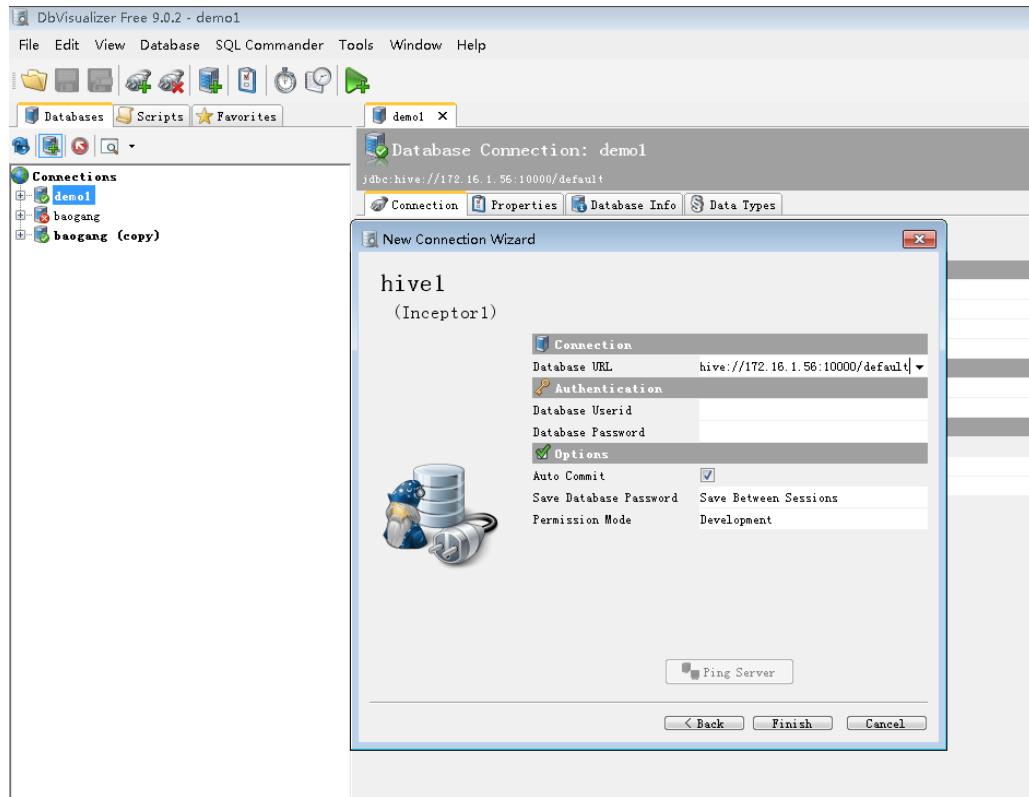


3. 选择驱动，连接InceptorServer 1时需要选择和InceptorServer 1对应的驱动。点击“Next”，跳转至下一步。

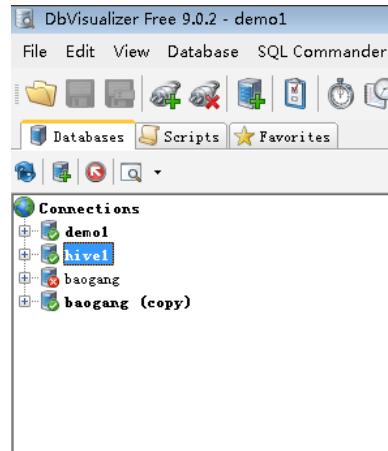


4. 在新窗口中键入连接字符串URL。此例中使用的连接字符串为：

```
jdbc:hive://172.16.1.56:10000/default
```

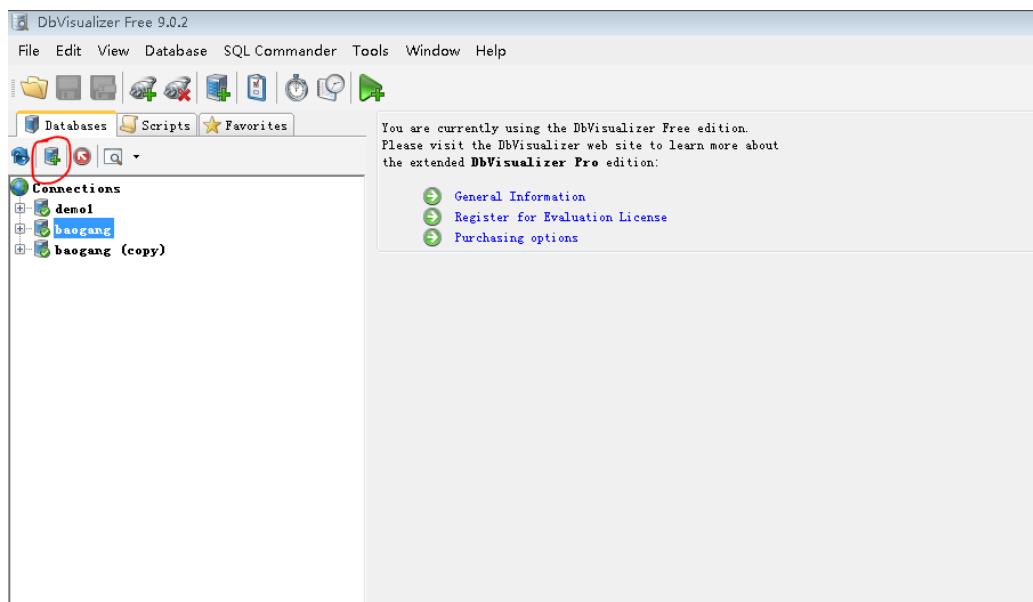


5. 配置成功时点击“finish”，如果新建连接中出现了绿色对勾表示连接成功。

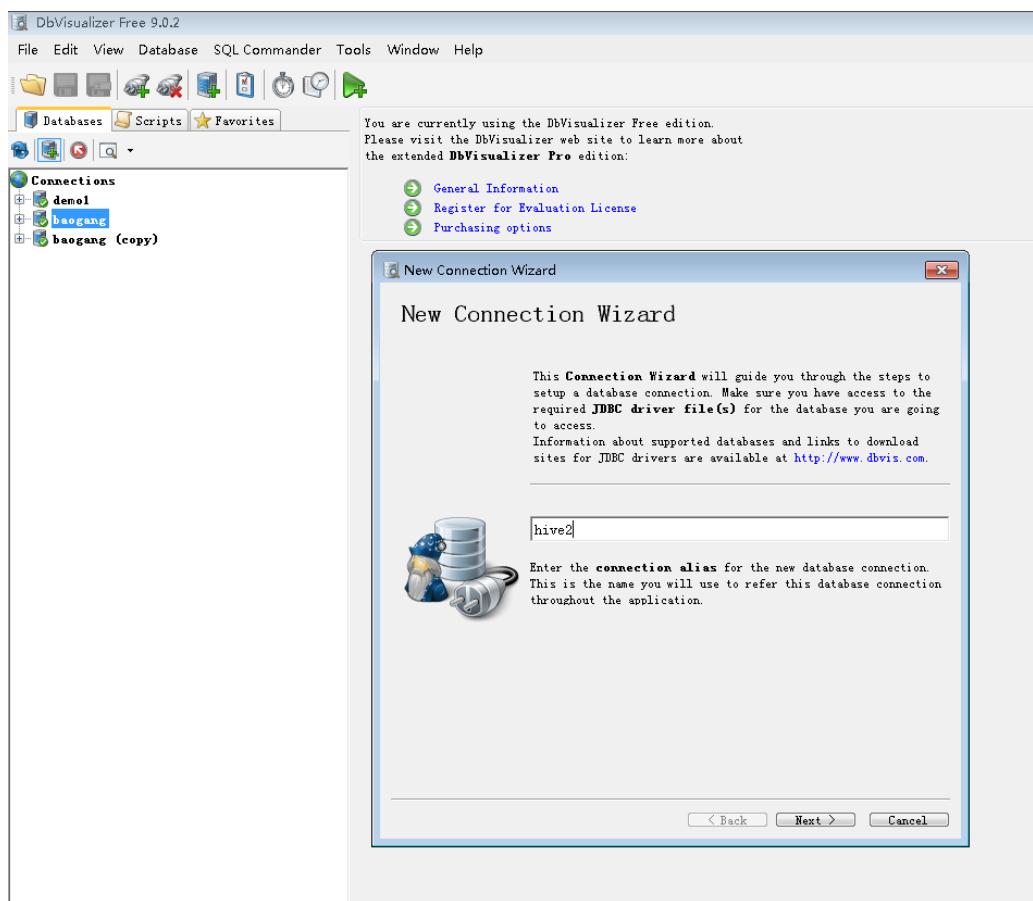


10.2.2.2. 需要认证的InceptorServer 2

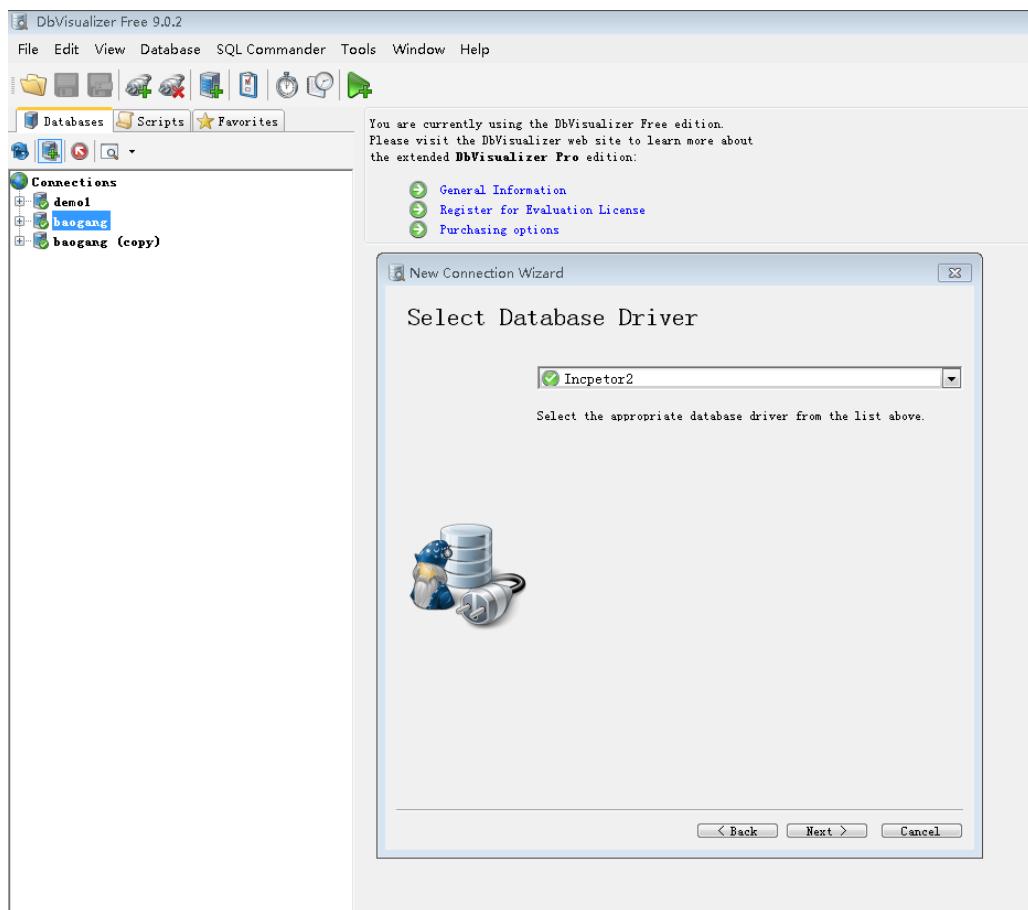
1. 如图点击DbVisualiser界面的左上角用红圈标注的图标，创建一个新的连接。



2. 出现如下界面，在该界面中为当前连接自定义一个名称，并点击“Next”，跳转至下一步。

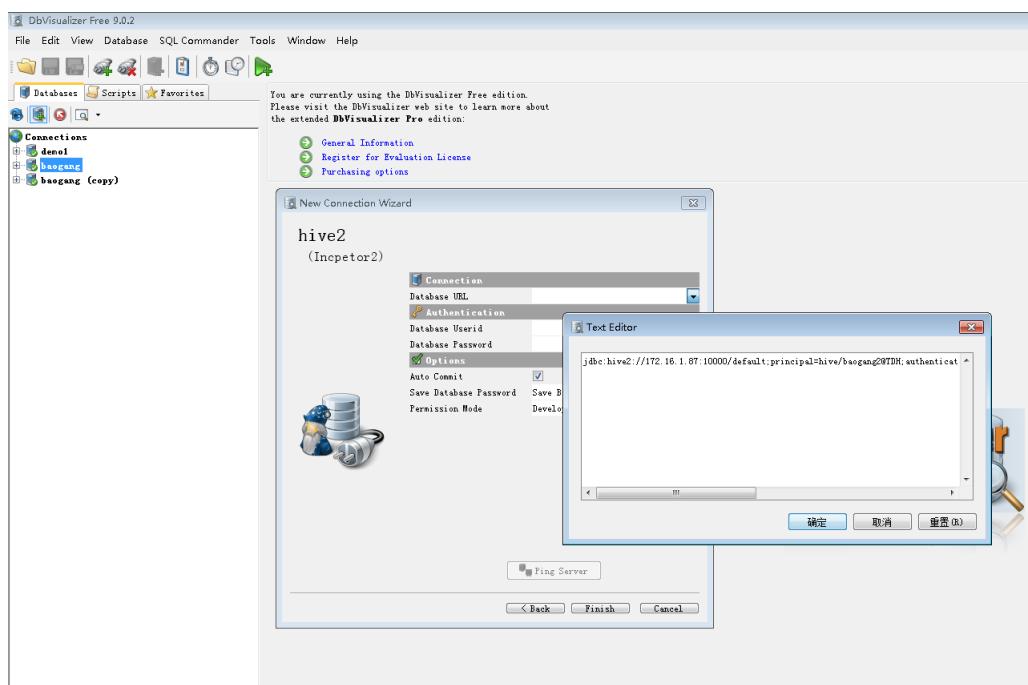


3. 选择驱动，连接InceptorServer 2时需要选择和InceptorServer 2对应的驱动。点击“Next”，跳转至下一步。

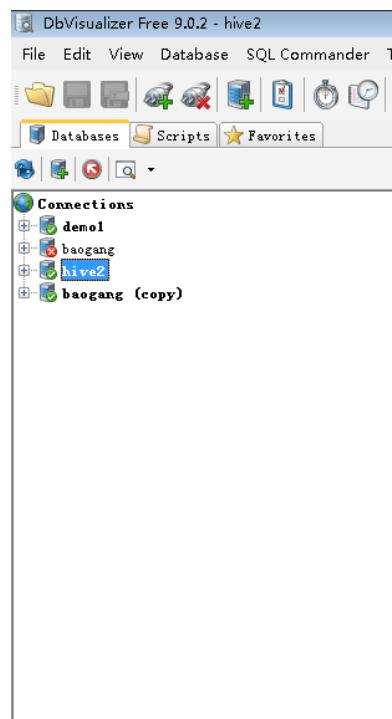


4. 在新窗口中键入连接字符串URL。注意，连接至需要认证的InceptorServer 2时，必须在URL中提供完整的认证信息，此例中使用的连接字符串为：

```
jdbc:hive2://172.16.1.87:10000/default;principal=hive/baogang2@TDH;authentication=kerberos;kuser=hive/baogang2@TDH;keytab=c:/kerberos/hive.keytab;krb5conf=c:/kerberos krb5.conf
```



5. 配置成功时点击“finish”，如果新建连接中出现了绿色对勾表示连接成功。

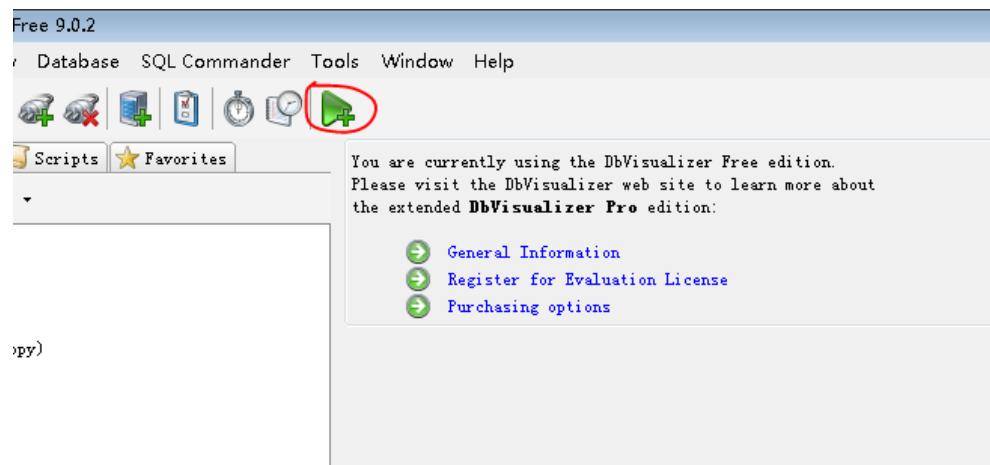


10.2.3. 通过DbVisualiser创建并查看存储过程

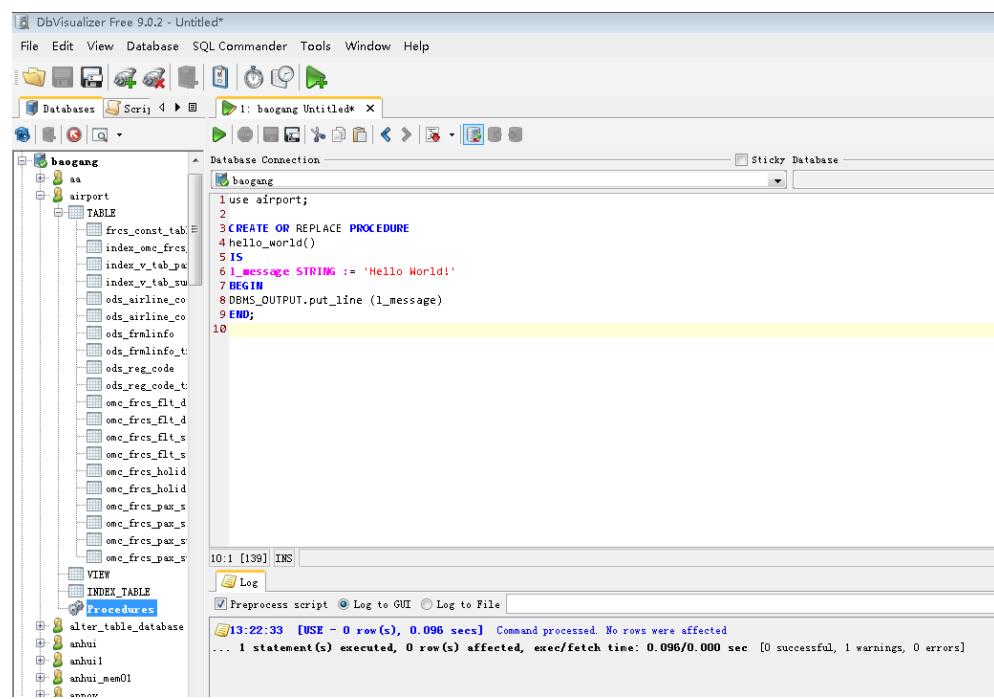
Inceptor支持在外部工具中查看用户创建的PL/SQL存储过程。但是目前仅允许在连接至InceptorServer 2时使用。下面将介绍如何操作。

1. 创建并查看不带参数的存储过程

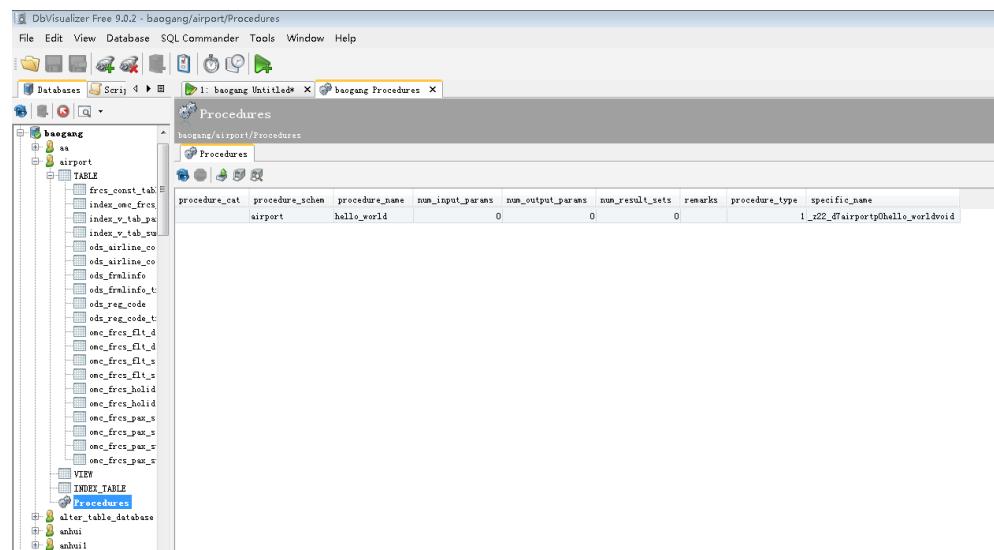
- 在窗口的菜单栏中找到下图中圈出的按钮，请点击以新建SQL语句编辑面板。



- 在数据库airport中创建图中所示的不带参数的存储过程。

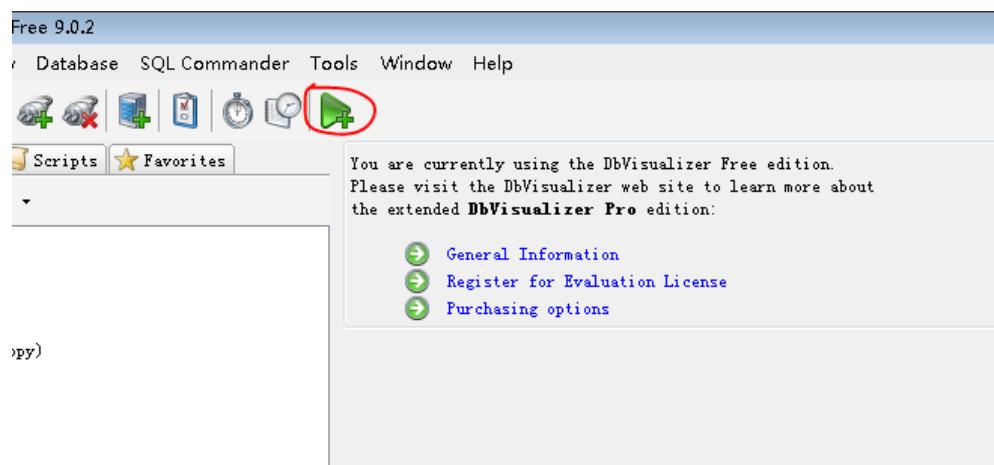


c. 展开airport，双击“Procedure”，点击ok，窗口右半部将出现存储过程hello_world()的信息。



2. 创建并查看带参数的存储过程

a. 在窗口的菜单栏中找到下图中圈出的按钮，请点击以新建SQL语句编辑面板。



b. 在数据库airport中创建图中所示的带参数的存储过程。

```

1 use airport;
2
3 CREATE OR REPLACE PROCEDURE
4   poperator IN STRING,
5   poperator_code OUT STRING,
6   papp_type OUT STRING
7   preg_code IN STRING
8 IS
9 BEGIN
10   poperator:= 'lix';
11   DBMS_OUTPUT.PUT_LINE('the operator is: '||poperator);
12   SELECT airline_code INTO poperator_code FROM ods_reg_code WHERE reg_code=preg_code;
13   DBMS_OUTPUT.PUT_LINE('the airline code is: '||poperator_code);
14   SELECT app_type INTO papp_type FROM ods_reg_code WHERE reg_code=preg_code;
15   DBMS_OUTPUT.PUT_LINE('the app type is: '||papp_type);
16 END;
17
18
19

```

19:1 [472] INS

Log

Preprocess script Log to GUI Log to File

13:51:03 [USE - 0 rows, 0.234 secs] Command processed. No rows were affected
... 1 statement(s) executed, 0 row(s) affected, exec/fetch time: 0.234/0.000 sec [0 successful, 0 warnings, 0 errors]

c. 展开airport，双击“Procedure”，点击ok，窗口右半部将出现存储过程param的信息。

procedure_cat	procedure_schema	procedure_name	num_input_params	num_output_params	num_result_sets	remarks	procedure_type	specific_name
airport		param	0	0	0			!_r16_dairport!param!string!string
airport		hello_world	0	0	0			!_r22_dairport!hello_world!void

10.2.4. Squirrel SQL连接Inceptor

本节我们将示范如何通过Squirrel SQL分别连接：

1. 无需认证的InceptorServer 1
2. 无需认证的InceptorServer 2
3. LDAP认证的InceptorServer 2
4. Kerberos认证的InceptorServer 2

使用Squirrel SQL连接到Inceptor server，您需要：

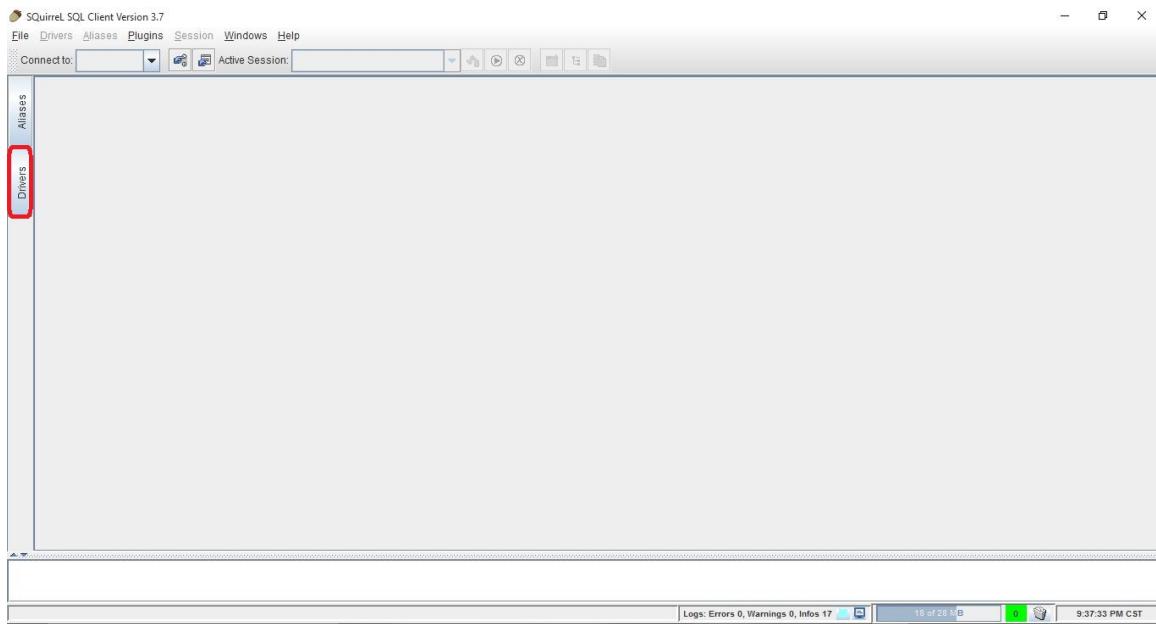
1. 添加分别对应InceptorServer 1和InceptorServer 2的JDBC驱动；
2. 添加连接到您的Inceptor server的alias；
3. 通过alias连接到您的Inceptor server。

10.2.4.1. 添加驱动

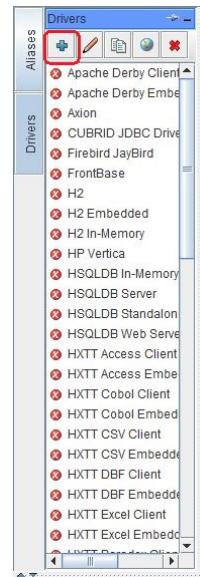
连接到InceptorServer 1和InceptorServer 2需要使用不同的驱动。

10.2.4.1.1. 连接InceptorServer 1的驱动

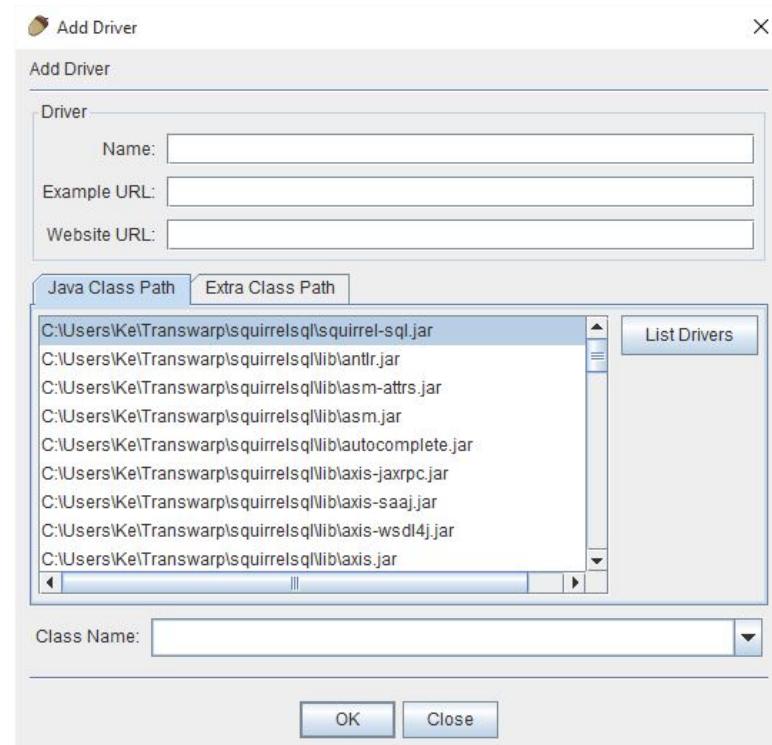
1. 打开Squirrel SQL:



2. 点击Squirrel SQL窗口左侧的“Drivers”，您可以看到一列驱动：



3. 点击上图中的“+”号来添加一个新的驱动，您将看到下面的窗口：

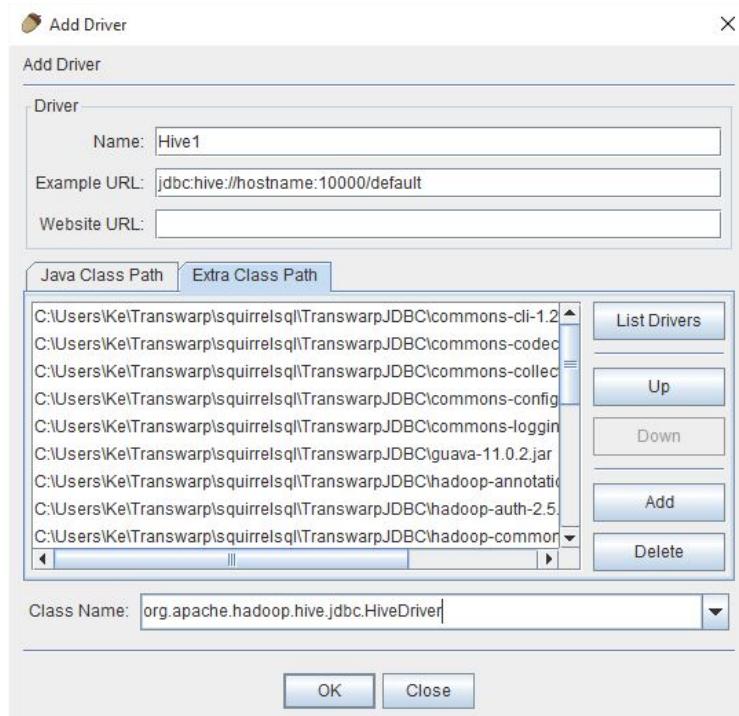


4. 在这个窗口中，您需要：

- 在Name处填写该驱动的名称。这里我们填写Hive1。
- 在Example URL处填写JDBC连接串的范例，它将是您之后添加的Alias的JDBC连接串的范例。这里我们输入下面的内容：

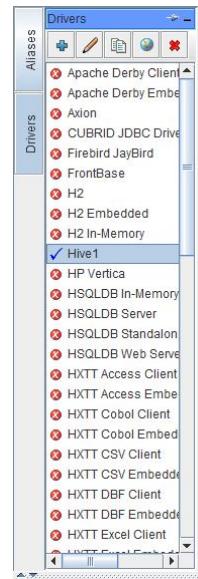
```
jdbc:hive://hostname:10000/default
```

- Website URL空着不填。
 - 点击Extra Class Path标签。点击"Add"按钮，将本节开头提到的JDBC所需要的jar包添加进Extra Class Path。
 - 在Class Name处填写连接InceptorServer 1驱动的类名：org.apache.hadoop.hive.jdbc.HiveDriver
- 填写完毕后，窗口应该显示如下信息：



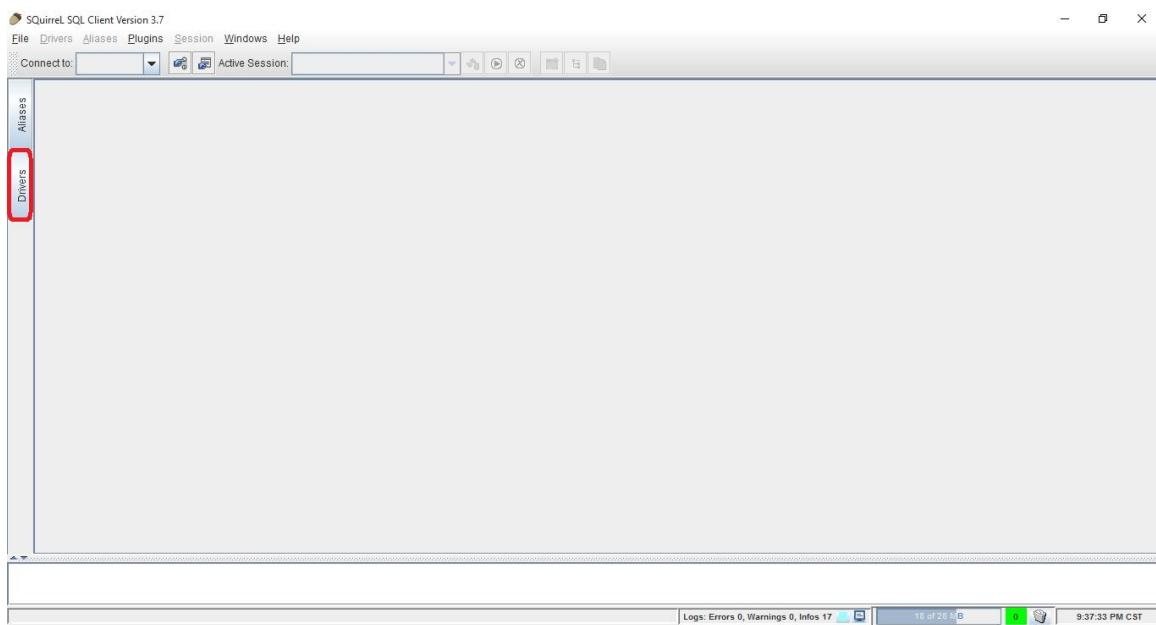
点击“OK”完成添加。

- 现在Squirrel SQL窗口左侧的驱动列表中可以看到我们刚刚添加的驱动Hive1:



10.2.4.1.2. 连接InceptorServer 2的驱动

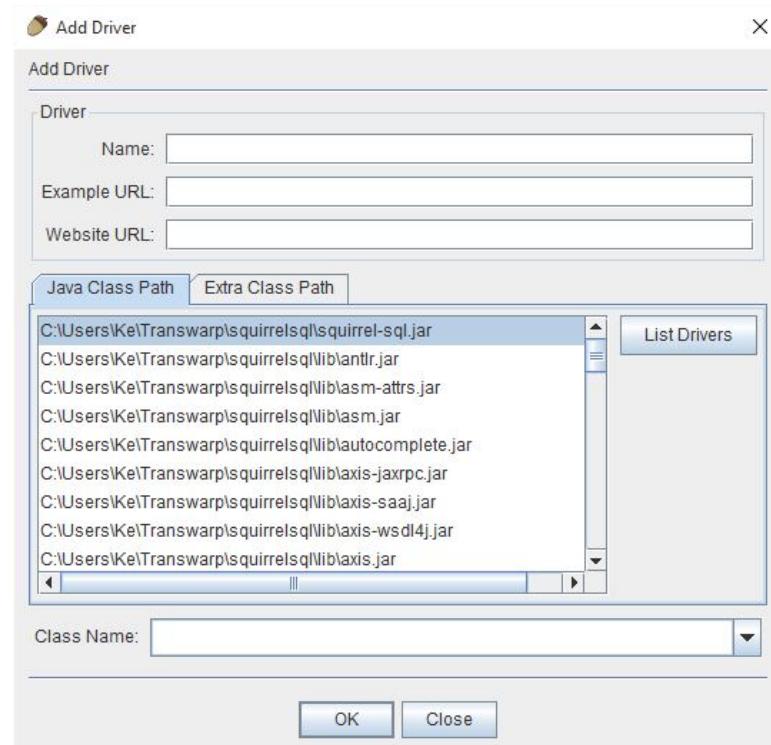
- 打开Squirrel SQL:



2. 点击Squirrel SQL窗口左侧的"Drivers"，您可以看到一列驱动：



3. 点击上图中的“+”号来添加一个新的驱动，您将看到下面的窗口：



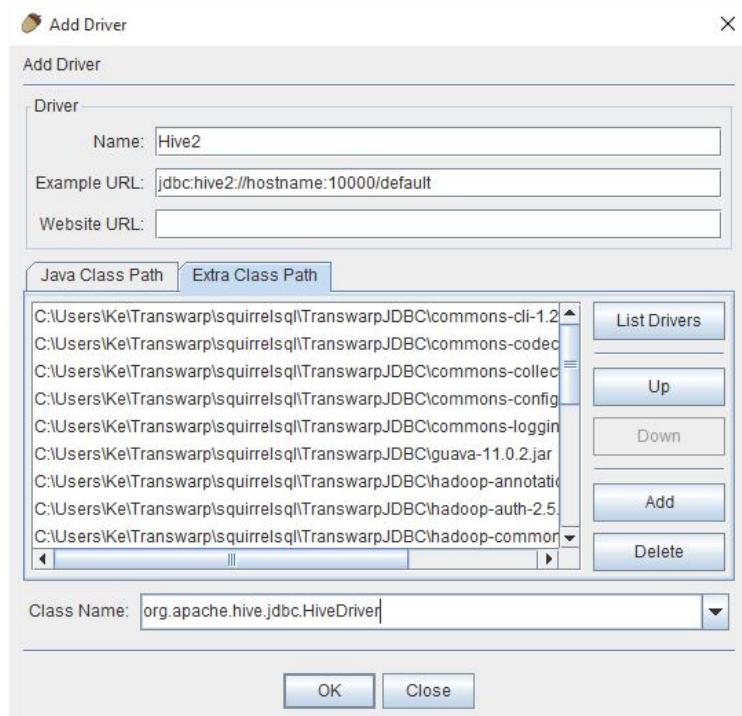
4. 在这个窗口中，您需要：

- 在Name处填写该驱动的名称。这里我们填写Hive2。
- 在Example URL处填写JDBC连接串的范例，它将是您之后添加的Alias的JDBC连接串的范例。这里我们输入下面的内容：

```
jdbc:hive2://hostname:10000/default
```

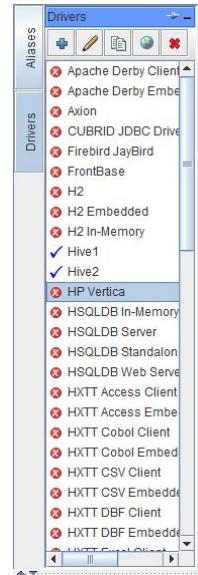
- Website URL空着不填。
- 点击Extra Class Path标签。点击“Add”按钮，将本节开头提到的JDBC所需要的jar包添加进Extra Class Path。
- 在Class Name处填写连接InceptorServer 2驱动的类名：org.apache.hive.jdbc.HiveDriver

填写完毕后，窗口应该显示如下信息：



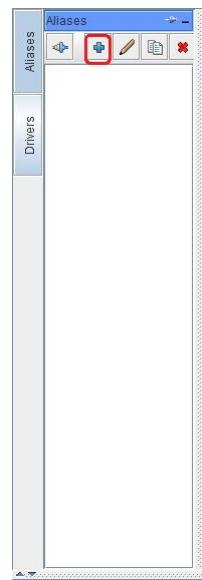
点击“OK”完成添加。

5. 现在Squirrel SQL窗口左侧的驱动列表中可以看到我们刚刚添加的驱动Hive2:



10.2.4.2. 添加alias

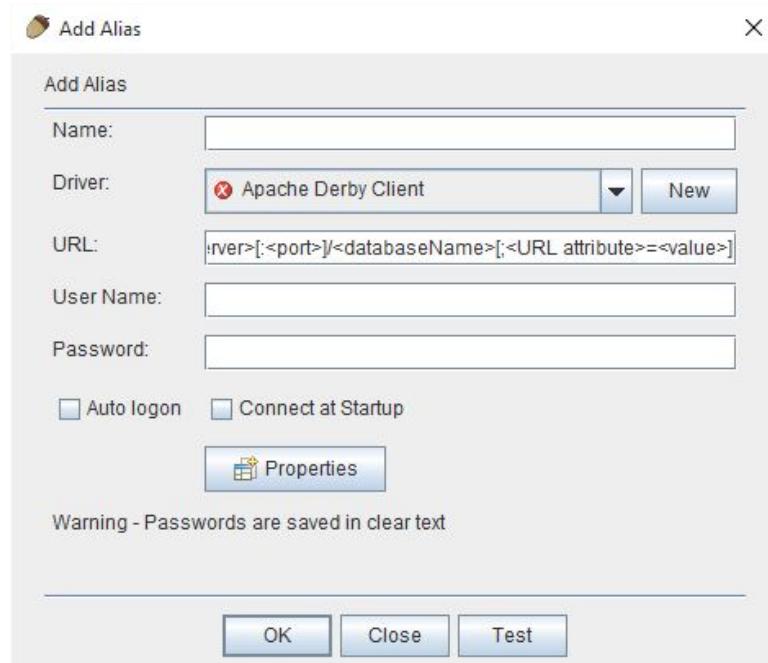
1. 点击“Drivers”标签上方的“Aliases”标签，您可以看到当前所有alias的列表。目前这个列表为空：



您需要点击图中的“+”号来添加新的alias。

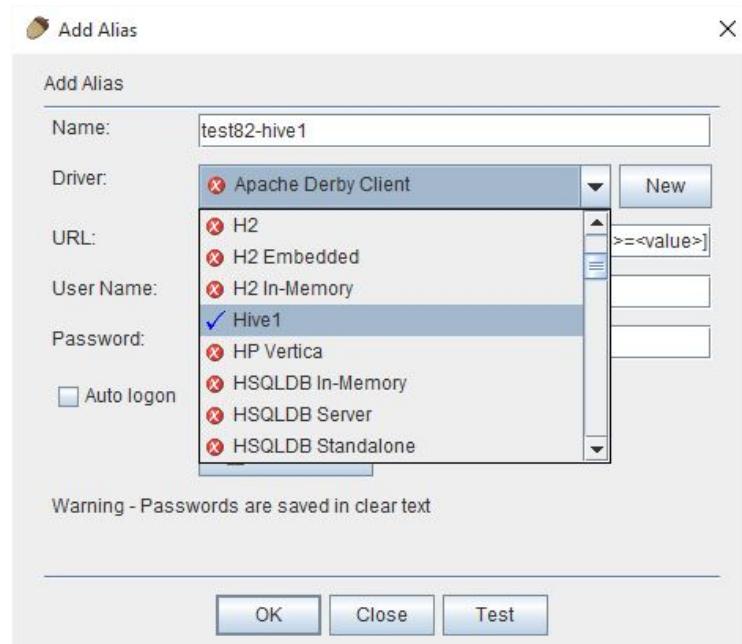
10.2.4.2.1. 无需认证的InceptorServer 1的alias

1. 点击alias列表上方的“+”号来添加一个新的alias，您将看到下面窗口：



2. 在这个窗口中您需要：

- 在Name处填写alias的名称。这里我们填写test82-hive1。
- 在Driver处点开下拉条，选中之前我们创建的驱动Hive1：

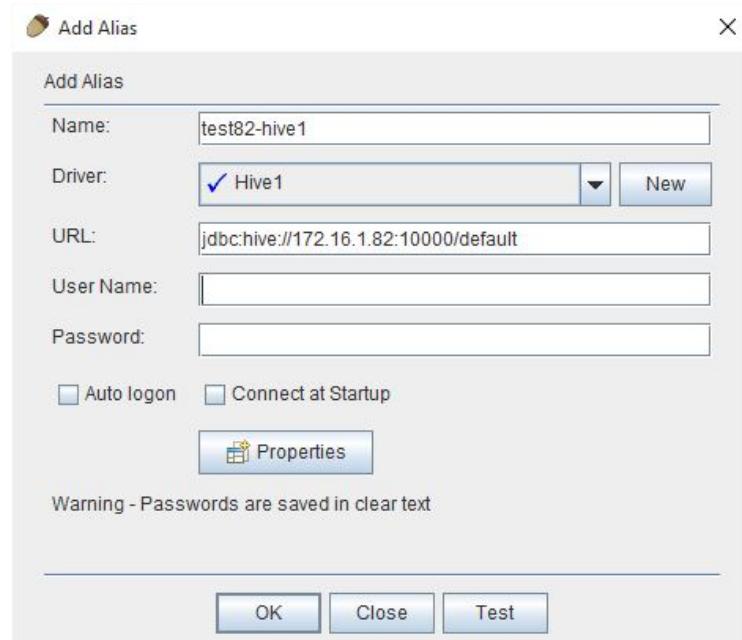


- URL处目前显示了我们之前提供的JDBC连接串范例，需要将其改写成实际的JDBC连接串，也就是说要将hostname改成您想要连接的Inceptor server所在节点的IP：

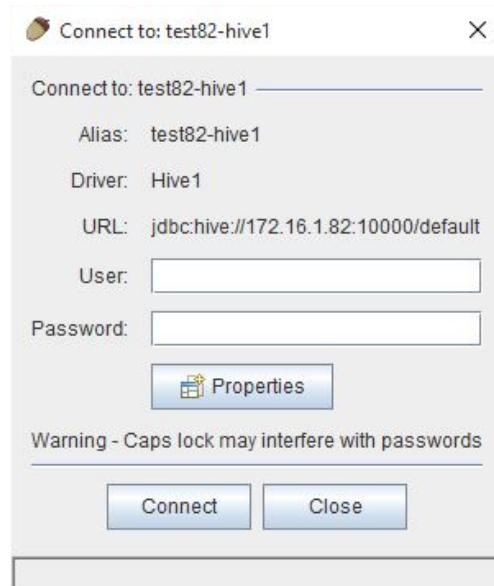
```
jdbc:hive://172.16.1.82:10000/default
```

- 因为我们要连接的是没有认证的InceptorServer 1，所以User Name和Password处空着不填。

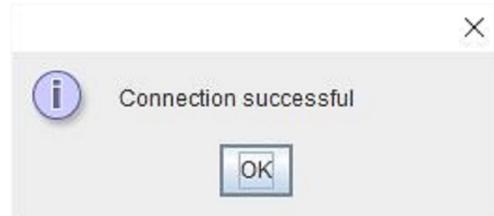
填写完毕后，窗口应该显示如下信息：



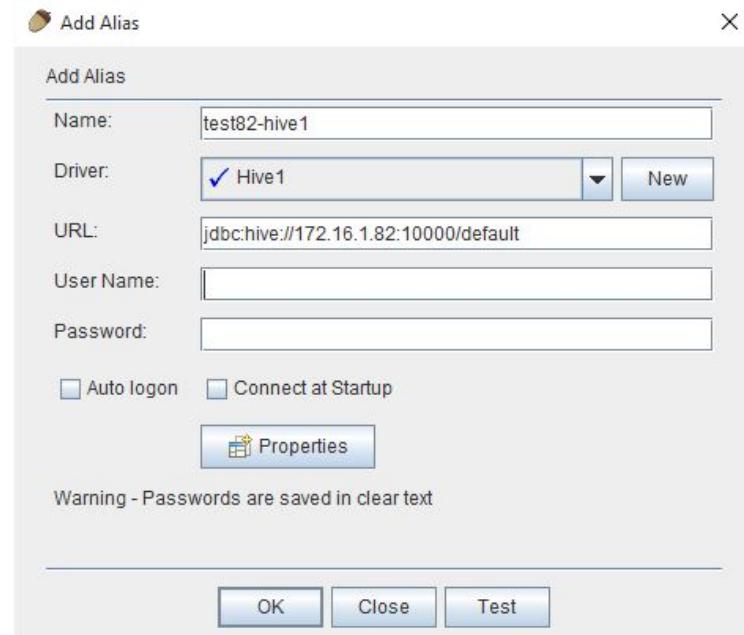
您可以点击Test来测试这个alias的连接，您将会看到下面弹出的窗口。同样，User和Password空着不填，直接点击Connect：



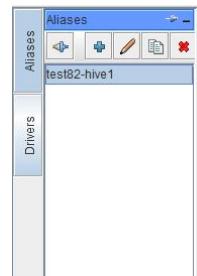
看到下面的弹出则说明测试连接成功：



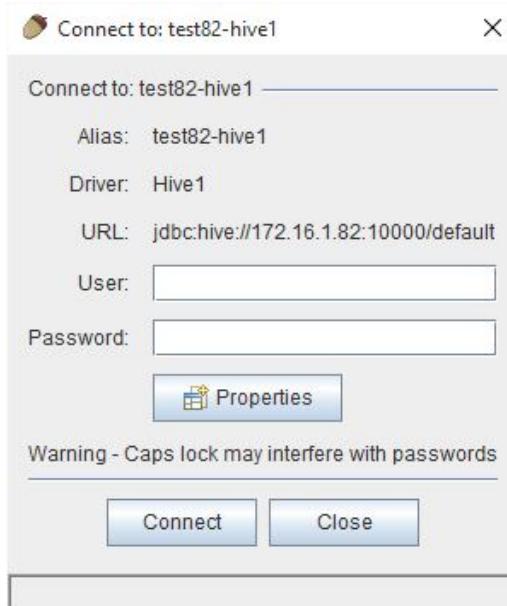
点击“OK”完成测试。回到添加alias的窗口点击“OK”完成添加：



- 现在点击Squirrel SQL窗口左侧的“Aliases”标签，我们可以看到新添加的test82-hive1：



4. 要建立和Inceptor server的连接，在上图的alias列表中双击test82-hive1，您会看到下面的窗口：



User和Password不填，直接点击Connect连接。

5. 连接成功后您将看到下面的显示：

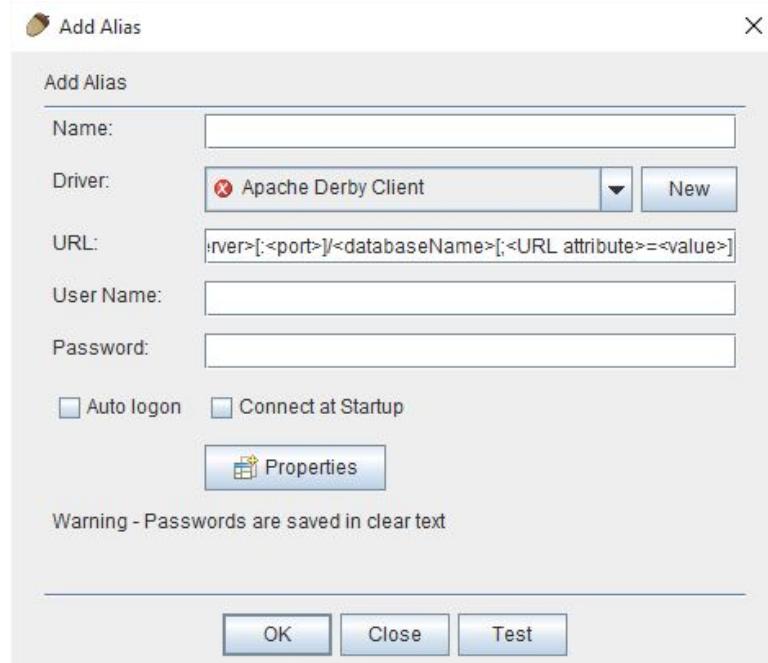
Property Name	Value
JDBC Driver CLASSNAME	org.apache.hadoop.hive.jdbc.HiveDriver
JDBC Driver CLASSPATH	C:\Users\Kei\Transwarp\squirrelsql\TranswarpJDBC\common\lib\hive-jdbc-1.2.1.jar;C:\Users\Kei\Transwarp\squirrelsql\TranswarpJDBC\mysql\mysql-connector-java-5.1.38-bin.jar;C:\Users\Kei\Transwarp\squirrelsql\TranswarpJDBC\oracle\ojdbc6.jar;C:\Users\Kei\Transwarp\squirrelsql\TranswarpJDBC\postgresql\postgresql-9.1-901.jdbc4.jar;C:\Users\Kei\Transwarp\squirrelsql\TranswarpJDBC\sqlserver\sqljdbc_4.0\enu\sqljdbc.jar;C:\Users\Kei\Transwarp\squirrelsql\TranswarpJDBC\mysql\mysql-connector-java-5.1.38-bin.jar;C:\Users\Kei\Transwarp\squirrelsql\TranswarpJDBC\oracle\ojdbc6.jar;C:\Users\Kei\Transwarp\squirrelsql\TranswarpJDBC\postgresql\postgresql-9.1-901.jdbc4.jar;C:\Users\Kei\Transwarp\squirrelsql\TranswarpJDBC\sqlserver\sqljdbc_4.0\enu\sqljdbc.jar
getResultSetHoldability	1
autoCommitIfAllRowsAffected	false
generatedKeyAlwaysReturned	<Unsupported>
getClientInfoForProperties	<Unsupported>
getMaxLogicalRowSize	0
getRowIdLifetime	<Unsupported>
supportsRefCursors	false
supportsStoreFunctionsUsingCallSyntax	true
getURL	<Unsupported>
isReadOnly	false
getUserName	<Unsupported>
storesUpperCaseIdentifiers	true
storesMixedCaseIdentifiers	true
getDriverName	Hive JDBC
supportsCatalogInTableDefinitions	false
supportsSchemasInTableDefinitions	false
getCatalogSeparator	:
getDatabaseProductVersion	1
supportsMultipleResultSets	false
supportsCatalogsInProcedureCalls	false

Driver class org.apache.hadoop.hive.jdbc.HiveDriver successfully registered for driver definition: Hive1
Please try out the Tools popup by hitting Ctrl-T in the SQL Editor. Do it three times to stop this message.
Please try out the Tools popup by hitting Ctrl-T in the SQL Editor. Do it three times to stop this message.

6. 现在，我们已经成功地使用Squirrel SQL和位于172.16.1.82上的Inceptor server建立连接，可以开始使用SQL Squirrel进行各种操作。

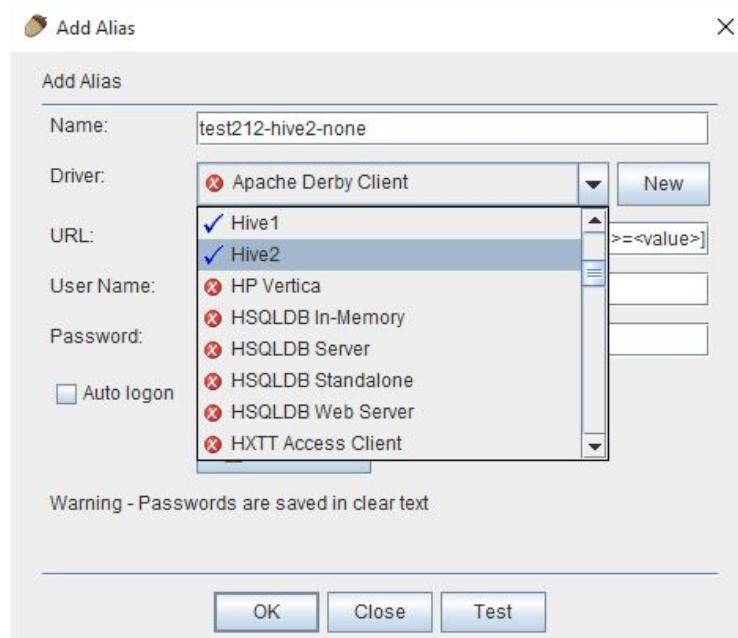
10.2.4.2.2. 无需认证的InceptorServer 2的alias

1. 点击alias列表上方的“+”号来添加一个新的alias，您将看到下面窗口：



2. 在这个窗口中您需要：

- 在Name处填写alias的名称。这里我们填写test212-hive-none。
- 在Driver处点开下拉条，选中之前我们创建的驱动Hive2：

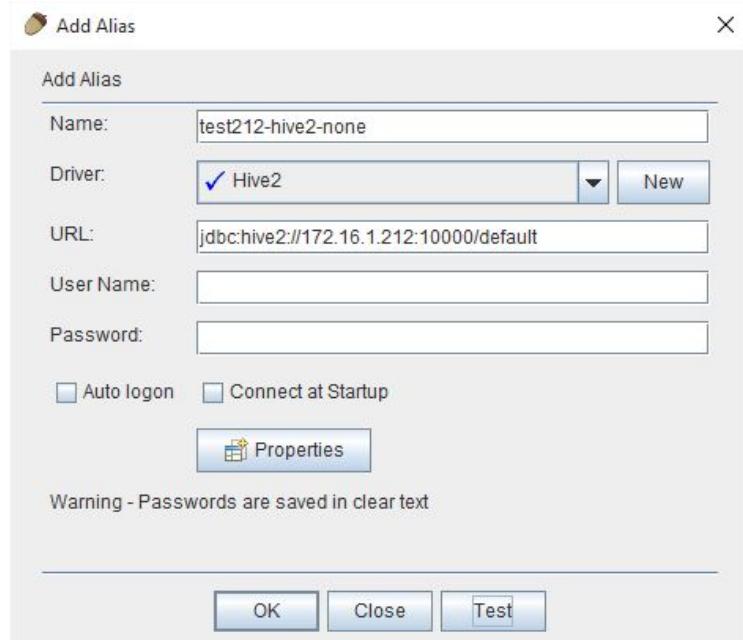


- URL处目前显示了我们之前提供的JDBC连接串范例，需要将其改写成实际的JDBC连接串，也就是说要将hostname改成您想要连接的Inceptor server所在节点的IP：

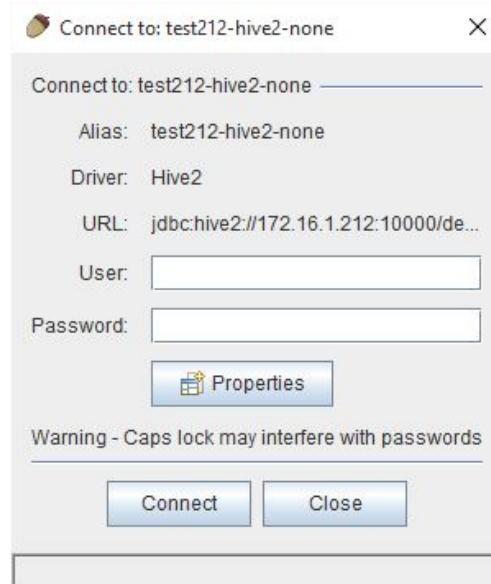
jdbc:hive://172.16.1.212:10000/default

- 因为我们要连接的Inceptor server是无需认证的InceptorServer 2，所以User Name和Password处空着不填。

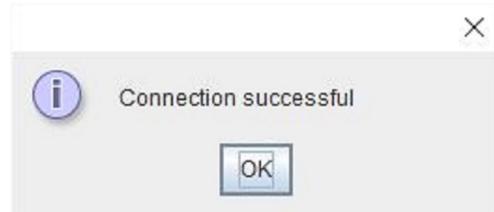
填写完毕后，窗口应该显示如下信息：



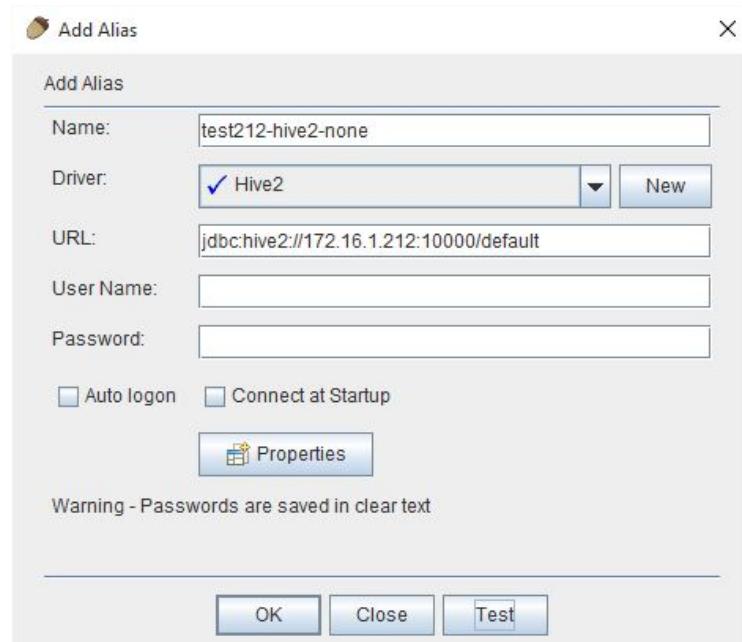
您可以点击Test来测试这个alias的连接，您将会看到下面弹出的窗口。同样，User和Password空着不填，直接点击Connect：



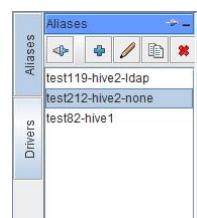
看到下面的弹出则说明测试连接成功：



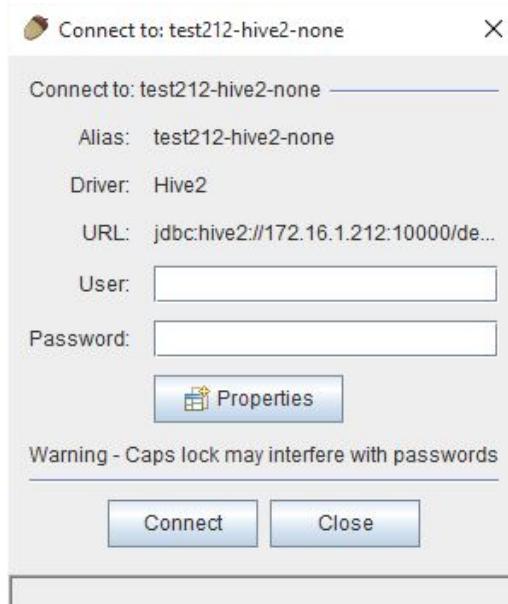
点击“OK”完成测试。回到添加alias的窗口点击“OK”完成添加：



3. 现在点击Squirrel SQL窗口左侧的“Aliases”标签，我们可以看到新添加的test212-hive2-none：

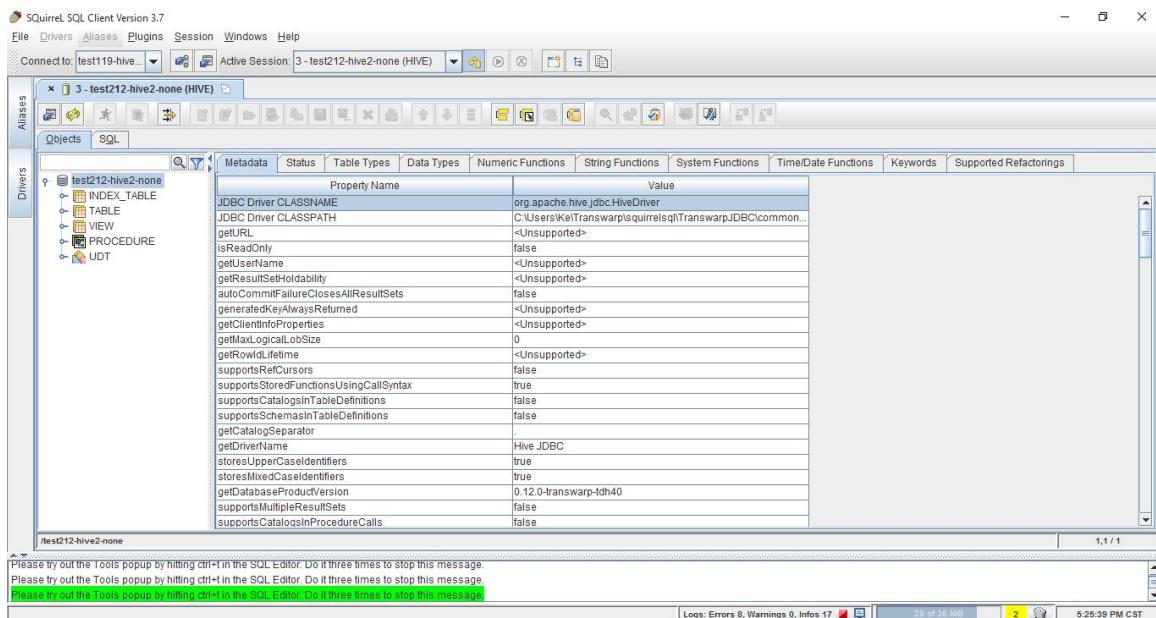


4. 要建立和Inceptor server的连接，在上图的alias列表中双击test82-hive2-none，您会看到下面的窗口：



User和Password不填，直接点击Connect连接。

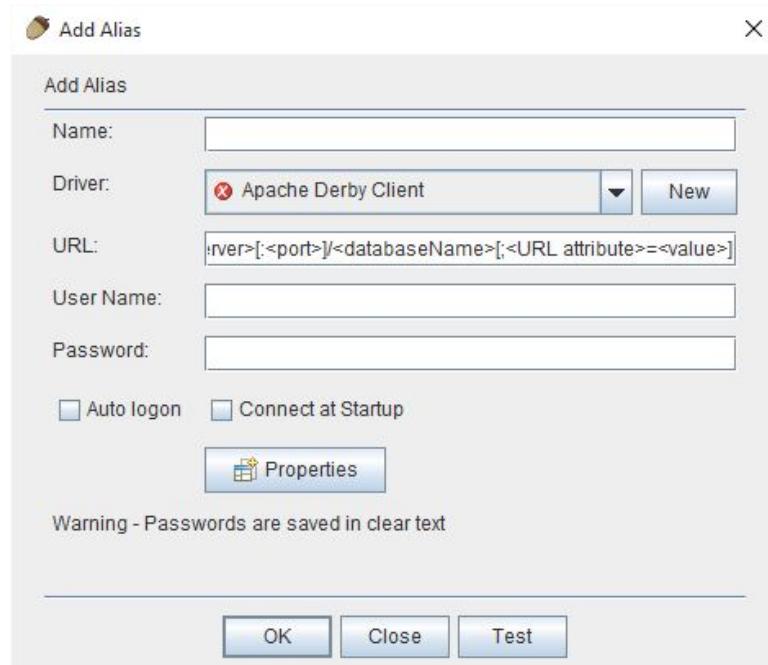
5. 连接成功后您将看到下面的显示：



6. 现在，我们已经成功地使用Squirrel SQL和位于172.16.1.212上的Inceptor server建立连接，可以开始使用SQL Squirrel进行各种操作。

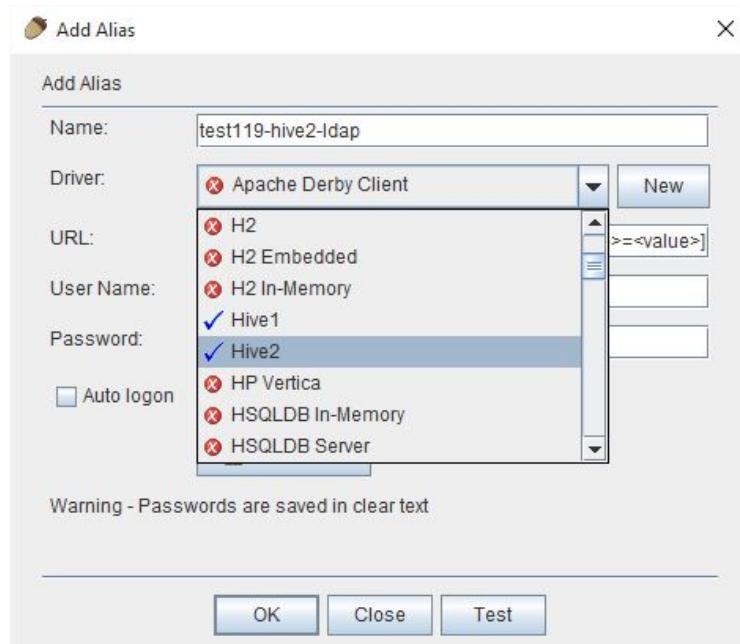
10.2.4.2.3. LDAP认证的InceptorServer 2的alias

1. 点击alias列表上方的“+”号来添加一个新的alias，您将看到下面窗口：



2. 在这个窗口中您需要：

- 在Name处填写该alias的名称。这里我们填写test119-hive2-ldap。
- 在Driver处点开下拉条，选中之前我们创建的驱动Hive2：

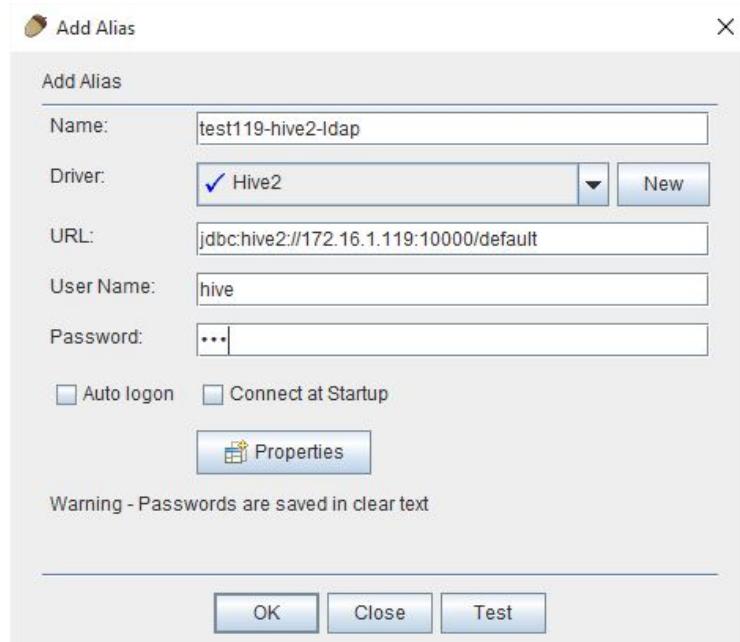


- URL处目前显示了我们之前提供的JDBC连接串范例，需要将其改写成实际的JDBC连接串，也就是说要将hostname改成您想要连接的Inceptor server所在节点的IP：

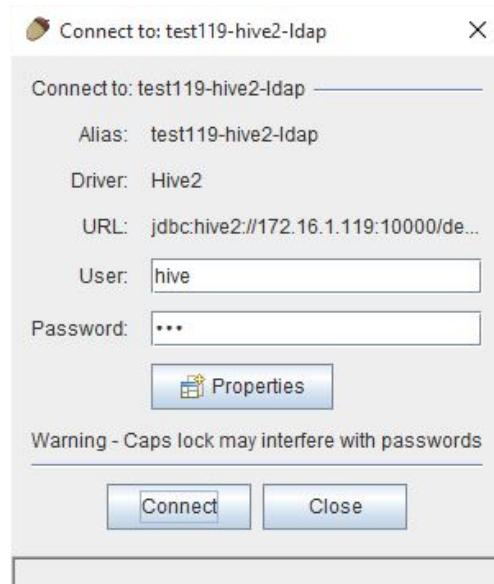
`jdbc:hive2://172.16.1.119:10000/default`

- User Name和Password分别填写连接到InceptorServer 2的用户在LDAP中的用户名和密码。这里我们以hive用户登陆。

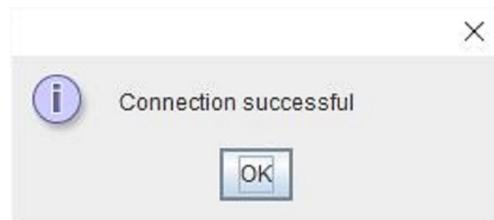
填写完毕后，窗口应该显示如下信息：



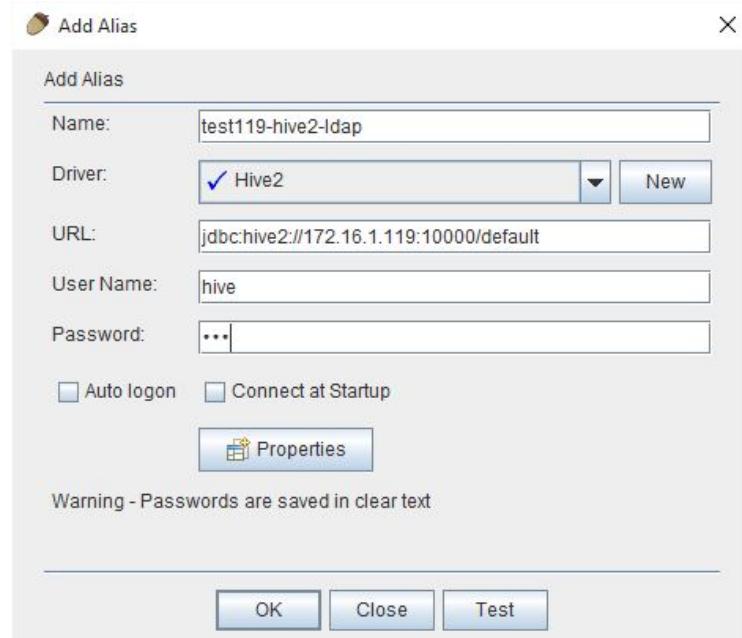
您可以点击Test来测试这个alias的连接，您将会看到下面弹出的窗口。点击Connect：



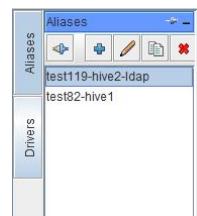
看到下面的弹出则说明测试连接成功：



点击“OK”完成测试。回到添加alias的窗口点击“OK”完成添加：



- 现在点击Squirrel SQL窗口左侧的“Aliases”标签，我们可以看到新添加的test119-hive2-ldap：



4. 要建立和Inceptor server的连接，在上图的alias列表中双击test119-hive2-ldap，您会看到下面的窗口：



点击Connect连接。

5. 连接成功后您将看到下面的显示：

Property Name	Value
JDBC Driver CLASSNAME	org.apache.hive.jdbc.HiveDriver
JDBC Driver CLASSPATH	C:\Users\Ke\Transwarp\squirrel\TranswarpJDBC\common\lib\hive-jdbc-0.12.0-transwarp-tdh4.jar
getURL	<Unsupported>
isReadOnly	false
getUserName	<Unsupported>
getResultSetHoldability	<Unsupported>
autoCommitIfFailureClosesAllResultSets	false
generateKeyAlwaysReturned	<Unsupported>
getClientInfoProperties	<Unsupported>
getMaxLogicalLocSize	0
getRowIdLifetime	<Unsupported>
supportsRefCursors	false
supportsStoredFunctionsUsingCallSyntax	true
supportsCatalogsInTableDefinitions	false
supportsSchemasInTableDefinitions	false
getCatalogSeparator	\
getDriverName	Hive JDBC
storesUpperCaseIdentifiers	true
storesMixedCaseIdentifiers	true
getDatabaseProductVersion	0.12.0-transwarp-tdh4
supportsMultipleResultSets	false
supportsCatalogInProcedureCalls	false

6. 现在，我们已经成功地使用Squirrel SQL和位于172.16.1.119上的Inceptor server建立连接，可以开始使用SQL Squirrel进行各种操作。

10.2.4.2.4. Kerberos认证的InceptorServer 2的alias

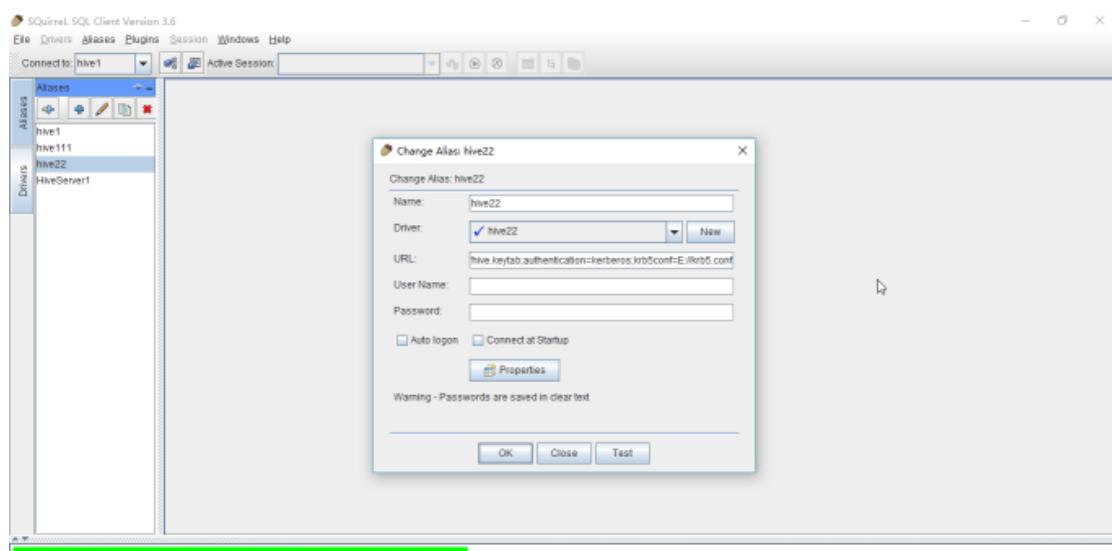
1. 点击alias列表上方的“+”号来添加一个新的alias。
2. 在这个窗口中您需要：
 - 在Name处填写该alias的名称。这里我们填写hive22。
 - 在Driver处点开下拉条，选中之前我们创建的驱动Hive2。
 - URL处目前显示了我们之前提供的JDBC连接串范例，需要将其改写成实际的JDBC连接串，也就是说要将hostname改成您想要连接的Inceptor server所在节点的IP，并且在URL中提供连接所用的principal、对应keytab和krb5.conf的所在地址，这里以principal为 `hive/CT-1@TDH`，keytab的所在地址为 `E:/hive.keytab;authentication=kerberos`，krb5.conf所在地址为 `E://krb5.conf` 为例，提供一个URL示例：

```
jdbc:hive2://172.16.2.65:10000/default;principal=hive/CT-1@TDH;kuser=hive;keytab=E://hive.keytab;authentication=kerberos;krb5conf=E://krb5.conf
```



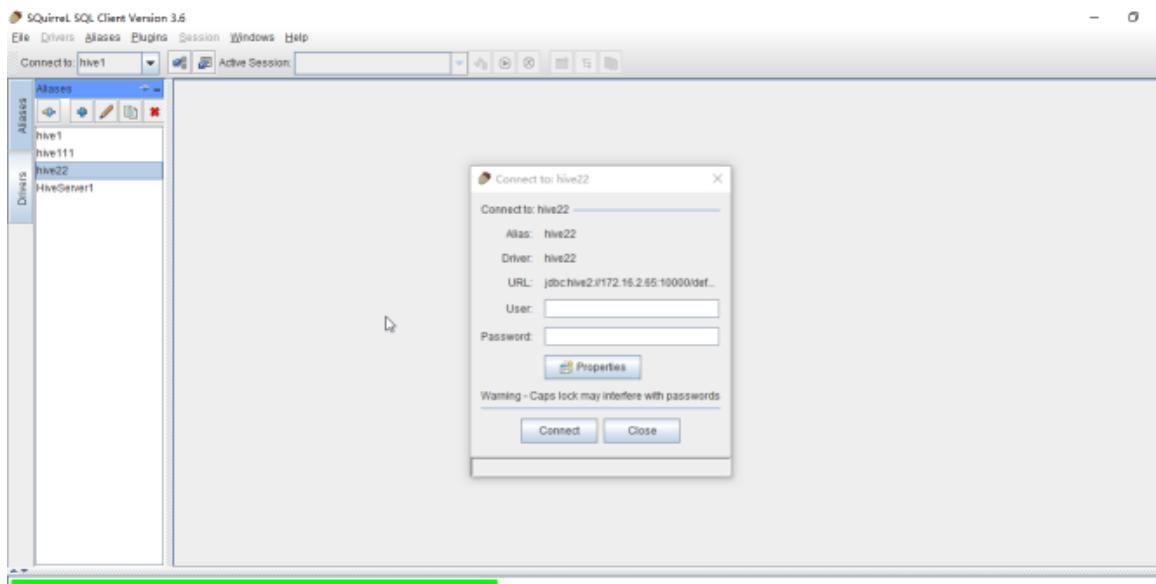
- hive.keytab从8180的对应用户上下载
- krb5.conf，从/etc下获取

信息提供完成后的页面如下：

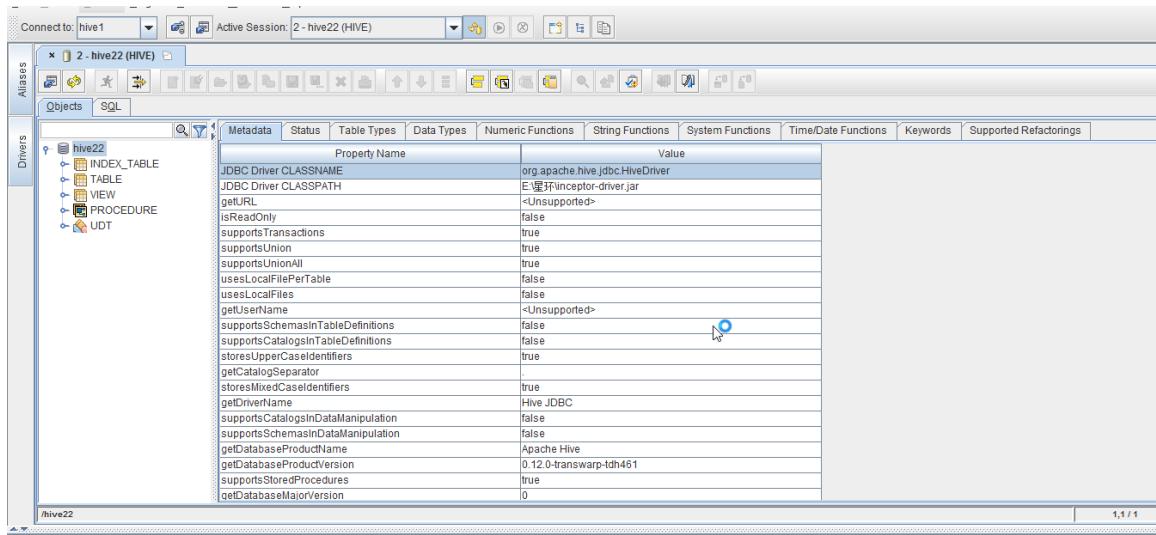


点击test，测试成功选择OK按钮，即成功创建了一个用Kerberos认证的Alias。

3. 连接此Alias时在Alias列表中双击待目标对象，出现如下窗口，选择 Connect。



4. 出现如下所示的窗机即表示连接成功:



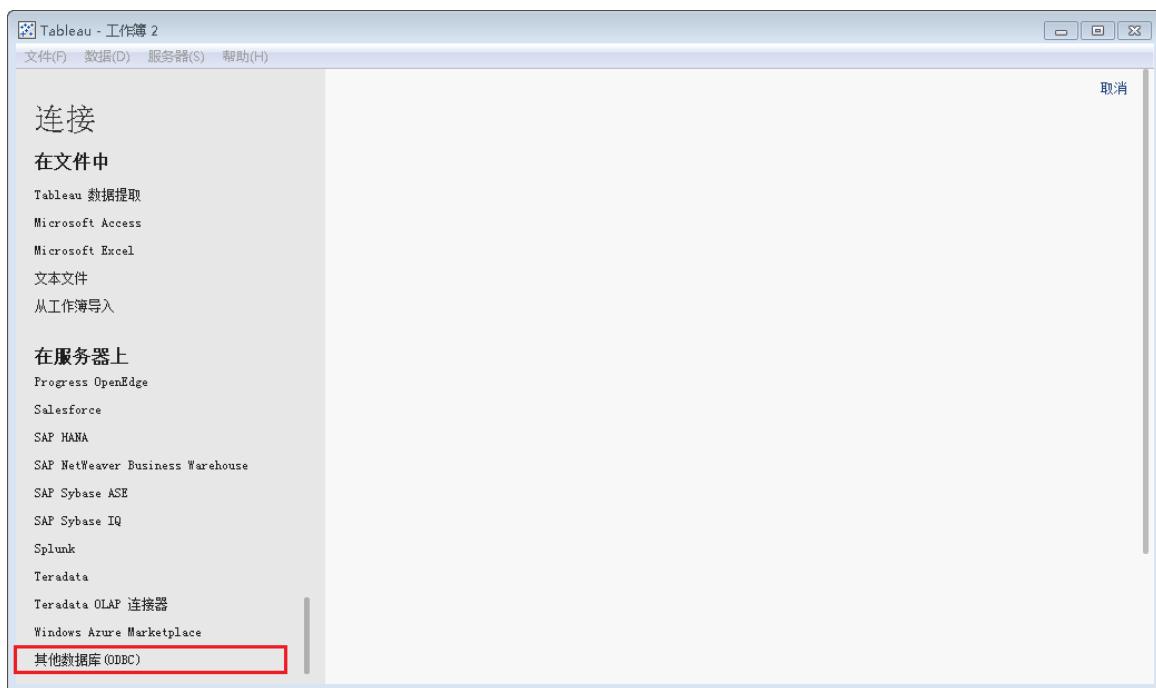
10.3. 使用ODBC的外部工具连接

在用外部工具通过ODBC连接到Inceptor前, 请确保您已经在您的计算机上部署了ODBC环境。部署方法请参考: [Window中的ODBC环境部署](#)和[Linux中的ODBC环境部署](#)。

10.3.1. Tableau连接Inceptor

本节我们将介绍如何使用Tableau连接Inceptor server。

1. 打开一个Tableau工作簿:

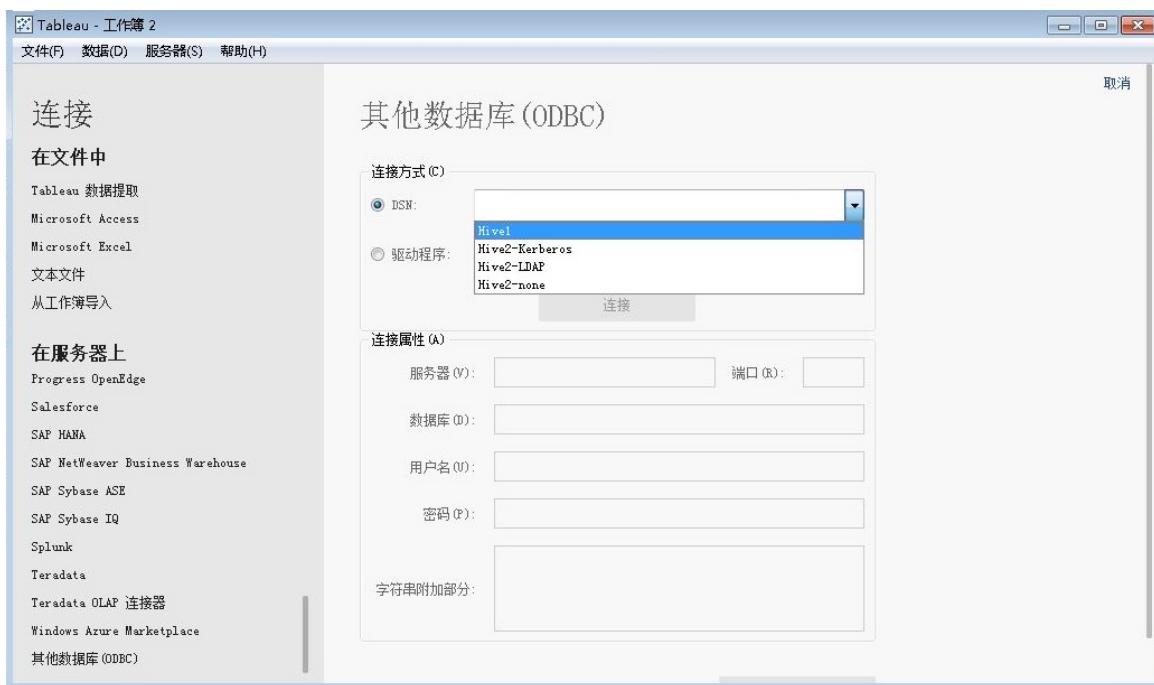


选择页面左侧“在服务器上”下的最后一个选项“其他数据库（ODBC）”：

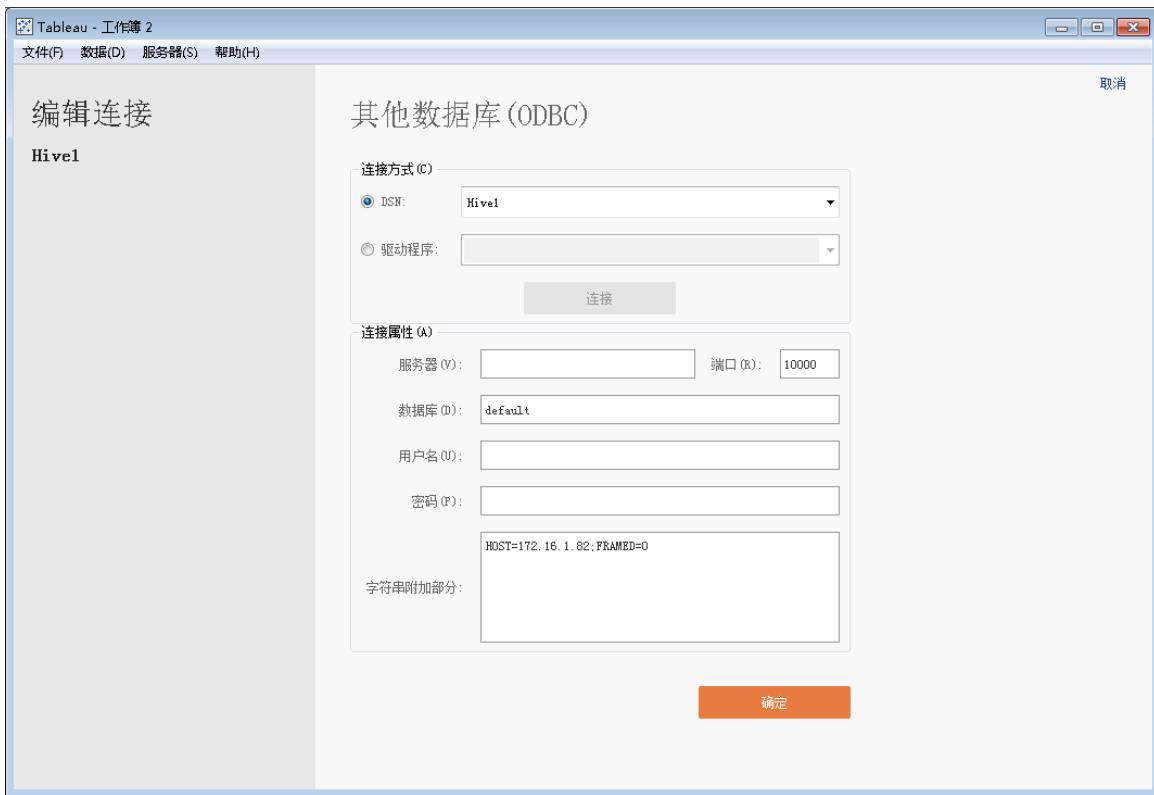
2. 您会看到如下页面：



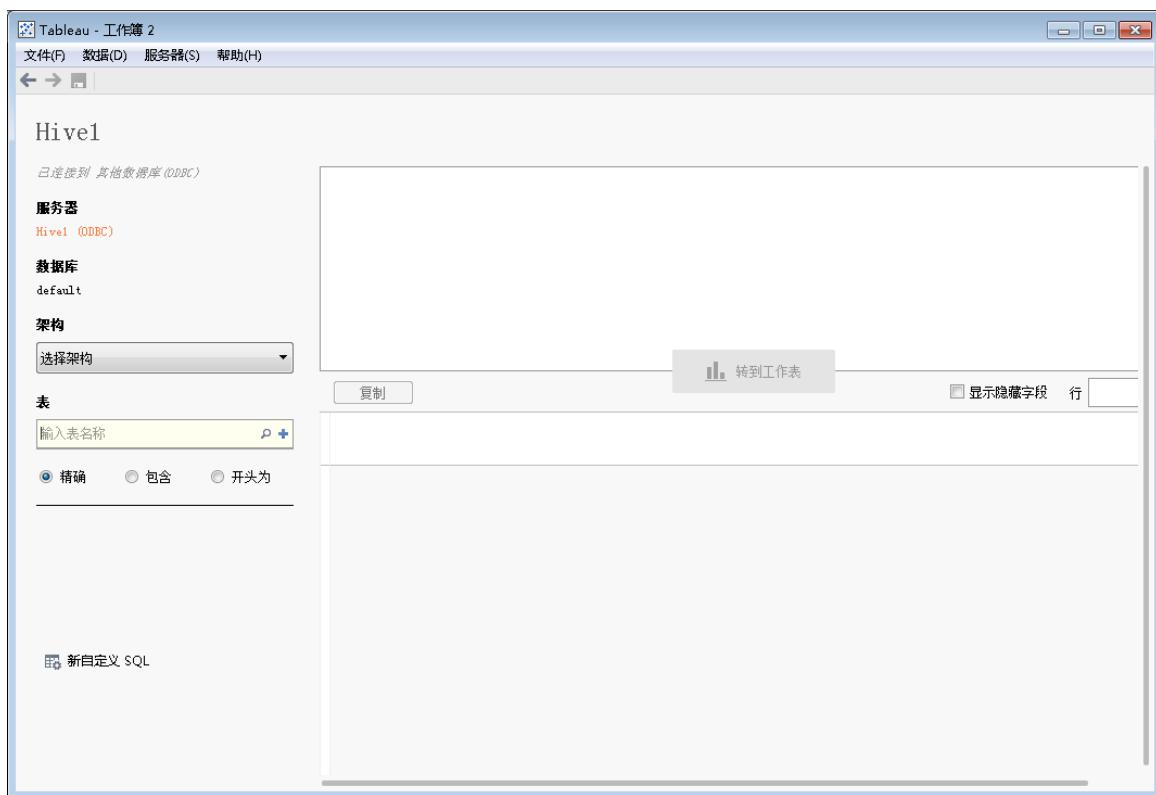
3. 在这个页面，根据您想要连接的Inceptor server选择对应的DSN。点开DSN的下拉条，您可以看到您已经添加好的DSN。



4. 选择对应您想要连接的Inceptor server的DSN，点击“连接”。
5. 点击“确定”：



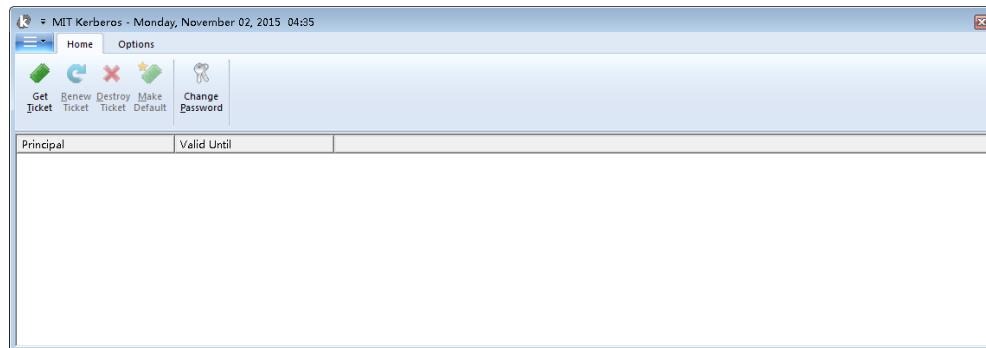
6. 看到下面的页面说明连接成功：



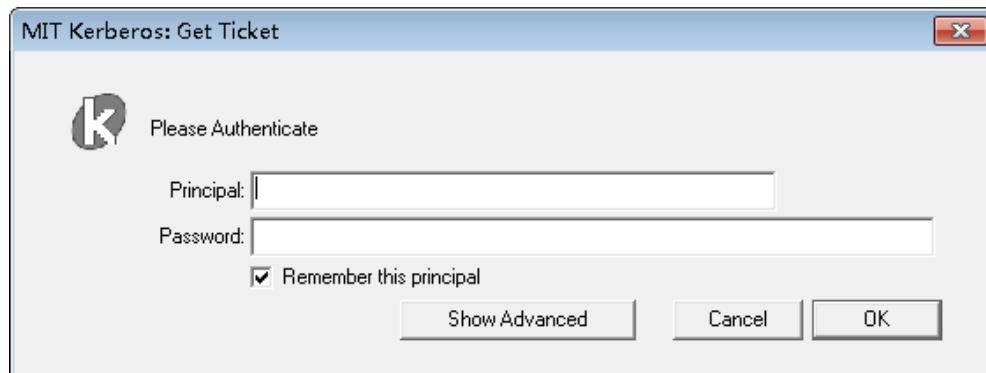
7. 如果您要连接到Kerberos认证的InceptorServer 2，您需要保证您的电脑上有一张有效的Kerberos TGT，否则将无法连接。获取Kerberos TGT的方法如下：

- 打开Kerberos客户端程序，这个程序包含在我们提供的Transwarp ODBC Driver For HiveServer2里，程序的路径为：

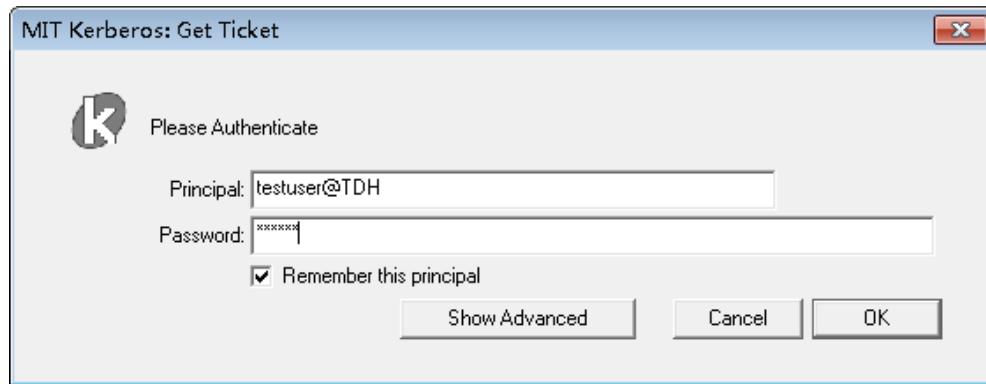
```
C:\Program Files (x86)\Transwarp ODBC Driver For HiveServer2\Kerberos
```



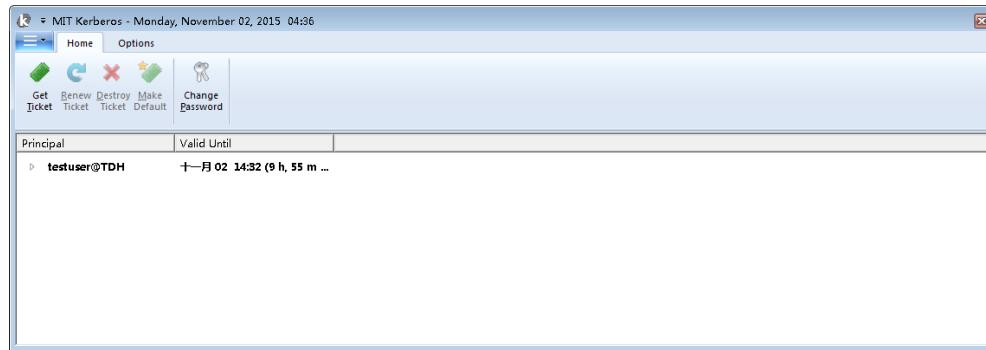
- 点击“Get Ticket”，程序会弹出下面窗口：



c. 在“Principal”和“Password”中分别填入您用户的principal和对应密码，点击“OK”完成。



d. 认证成功后您会在程序中看到一张有效TGT:



e. 获取有效的TGT后，您在Tableau中可以顺利连接上Kerberos认证的DSN。

11. Inceptor运维手册

11.1. Inceptor的配置

Inceptor可配置两种模式：InceptorServer 1（无安全认证模式）、InceptorServer 2（安全认证模式），两种模式均需要您在安装Inceptor服务时进行配置，您可根据自己的需求安装其中任意一种。

11.1.1. InceptorServer 1

- CLI登录方法

```
transwarp -t -h <server_ip/hostname>
```

- 配置方法

1. 进入添加服务页面，选择Inceptor服务，点击下一步。



2. 为Inceptor服务指定其依赖的服务，点击下一步。

1.选择服务 2.指定依赖 3.分配角色 4.安全模式 5.配置服务 6.安装

为选择的服务指定其依赖的服务

HDFS	YARN	Hyperbase	Inceptor-SQL
<input type="radio"/> HDFS1	YARN1		
<input type="radio"/> HDFS1	YARN1	Hyperbase1	
<input type="radio"/> HDFS1	YARN1		InceptorSQL1
<input checked="" type="radio"/> HDFS1	YARN1	Hyperbase1	InceptorSQL1

3. 为选中的服务分配角色， 默认已按推荐方法分配，您也可以修改分配，然后点击下一步。

4. 在安全模式配置页面中，选择“简单认证模式”，点击下一步。

1.选择服务 2.指定依赖 3.分配角色 4.安全模式 5.配置服务 6.安装

指定安全认证模式

简单认证模式 Kerberos认证模式

5. 在配置服务页面中，点击“高级参数”，选择hive.server2.enabled的值为“FALSE”，点击下一步

1.选择服务 2.指定依赖 3.分配角色 4.安全模式 5.配置服务 6.安装

配置选中的服务

属性	基础参数	高级参数	自定义参数	资源分配
INCEPTOR_SQL (InceptorSQL2)				
高级参数				
值				
		hive.server2.enabled	<input type="text" value="FALSE"/>	

上一步 **下一步**

6. 查看Inceptor的基础参数，确认完毕后点击下一步，在弹出的“安装确认”页面中点击“确认”，开始安装。

11.1.2. 配置InceptorServer 2

InceptorServer 2有两种认证模式：LDAP认证模式、Kerberos认证模式，您可以根据自己需求选择其中一种模式。

11.1.2.1. LDAP认证模式

- CLI登录方法

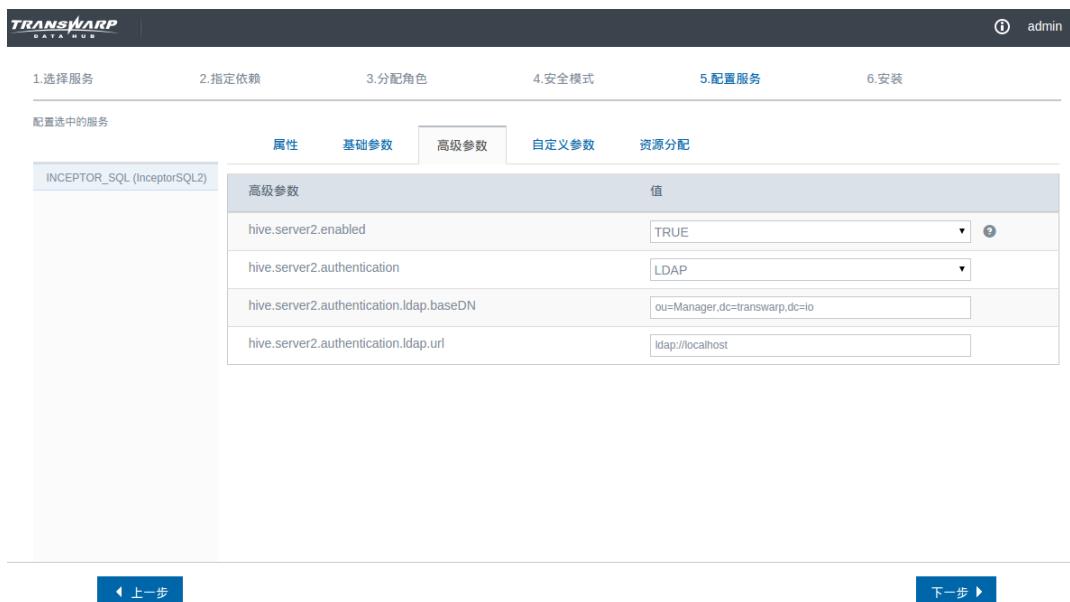
```
beeline -u 'jdbc:hive2://tw-node142:10000/default' -n hive -p hive
```

- 配置方法

1. 按照上述配置InceptorServer 1的步骤1-3操作。
2. 在安全模式配置页面中，选择“简单认证模式”，点击下一步。



3. 在配置服务页面中，点击“高级参数”，选择hive.server2.enabled的值为“TRUE”，选择hive.server2.authentication的值为“LDAP”，点击下一步。



4. 按照上述配置InceptorServer 1方法的步骤6操作。

在安装LDAP认证模式的Inceptor之前必须先配置LDAP，详细操作请见附件1《配置LDAP》。

11.1.2.2. Kerberos认证模式

- 用户认证

```
kinit <user_name>
```

- Beeline连接Inceptor Server

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

- 配置方法

1. 按照上述配置InceptorServer 1的步骤1-3操作。
2. 在安全模式配置页面中, 选择”Kerberos认证模式“, 输入密码, 点击下一步。

The screenshot shows a configuration wizard for Inceptor. The current step is '4. 安全模式' (Security Mode). A radio button for 'Kerberos认证模式' (Kerberos Authentication Mode) is selected. Below it, the 'KAdmin票据名:' (KAdmin ticket name) field contains 'kadmin/admin' and the 'Kadmin 密码:' (Kadmin password) field contains a masked password. At the bottom, there are '上一步' (Back) and '下一步' (Next) buttons.

3. 在配置服务页面中, 点击“高级参数”, 选择hive. server2. enabled的值为“TURE”, 选择hive. server2. authentication的值为“NONE”, 点击下一步。

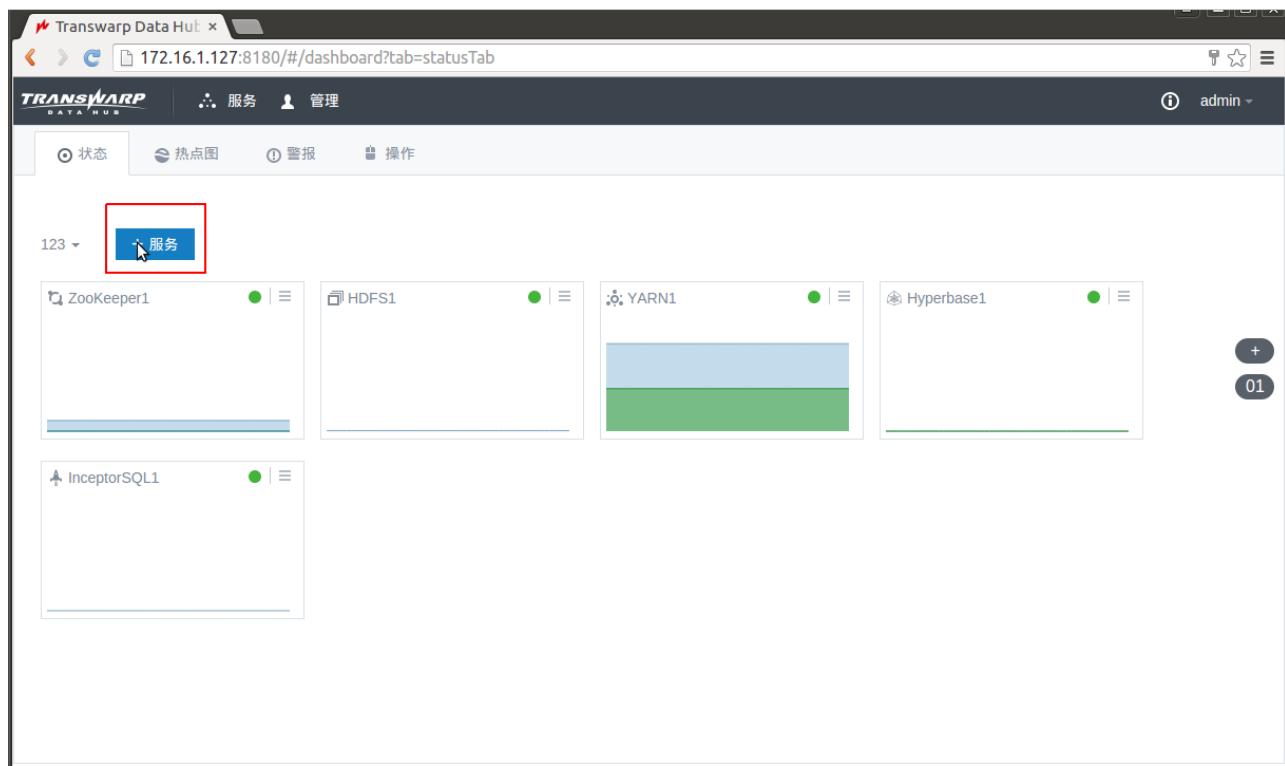
高级参数	值
hive.server2.enabled	TRUE
hive.server2.authentication	NONE

4. 按照上述配置InceptorServer 1方法的步骤6操作。

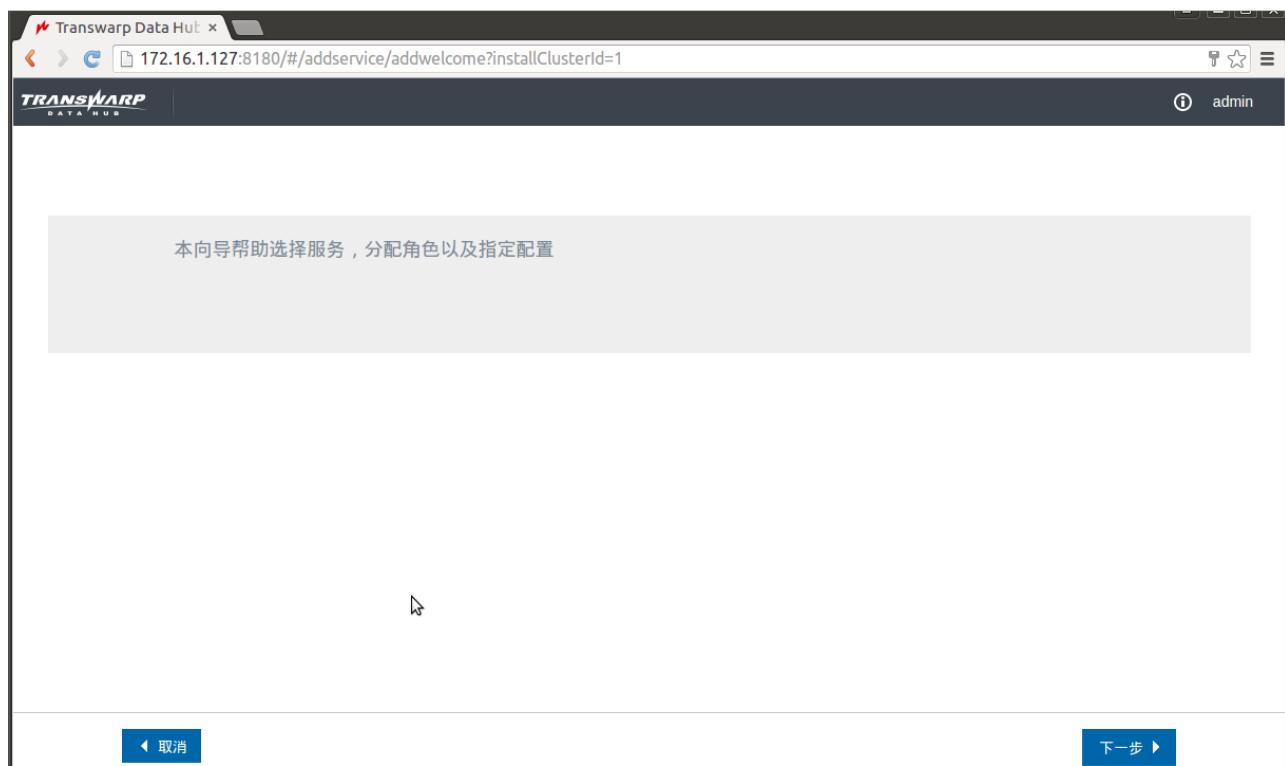
11.2. 多Inceptor的安装配置

一个集群中可以有多个Inceptor 服务。本章我们介绍如何向一个集群添加一个Inceptor服务。在我们的示例中，集群已经有一个名为InceptorSQL1的Inceptor服务。Transwarp Manager将自动将我们想要新增的Inceptor服务命名为InceptorSQL2。

1. 登陆Transwarp Manager，在首页点击“+”来添加一个服务：



2. Transwarp Manager会启动安装向导协助您安装：



点击“下一步”继续。

3. 在该页面选择“Inceptor”：

1.选择服务 2.指定依赖 3.分配角色 4.安全模式 5.配置服务 6.安装

选择想要添加的服务，可能存在某些由于依赖服务未添加而无法添加的服务，请先添加其依赖的服务。鼠标悬停在一个服务上可以看到它依赖的服务

<input checked="" type="radio"/> ZooKeeper ZooKeeper 用于协调同步其他服务	<input checked="" type="radio"/> HDFS HDFS是Hadoop应用的基本存储系统	<input checked="" type="radio"/> YARN YARN 是资源管理框架
<input checked="" type="radio"/> Hyperbase Hyperbase 是实时在线事务处理引擎	<input checked="" type="radio"/> Inceptor Inceptor是基于内存的交互式SQL分析引擎	<input checked="" type="radio"/> Discover Discover是基于内存的数据挖掘引擎
<input checked="" type="radio"/> Stream Transwarp Stream 是基于Spark Streaming的实时流处理引擎	<input checked="" type="radio"/> Kafka Kafka是一个分布式的消息发布订阅系统	<input checked="" type="radio"/> Oozie Oozie是Hadoop的一个工作流调度系统
<input checked="" type="radio"/> Sqoop Sqoop 用于在Hadoop和其他结构化数据存储中传递数据	<input checked="" type="radio"/> Hue Hue是Apache Hadoop中用来分析数据的用户界面	<input checked="" type="radio"/> Elastic Search Elasticsearch是一个分布式的搜索分析引擎

◀ 上一步 下一步 ▶

点击“下一步”继续。

4. 在该页面您需要指定将要安装的Inceptor服务的依赖。一个新增的Inceptor服务可以选择依赖或不依赖于集群中已有的Inceptor服务，也可以选择依赖或不依赖于集群中已有的Hyperbase服务：

1.选择服务 2.指定依赖 3.分配角色 4.安全模式 5.配置服务 6.安装

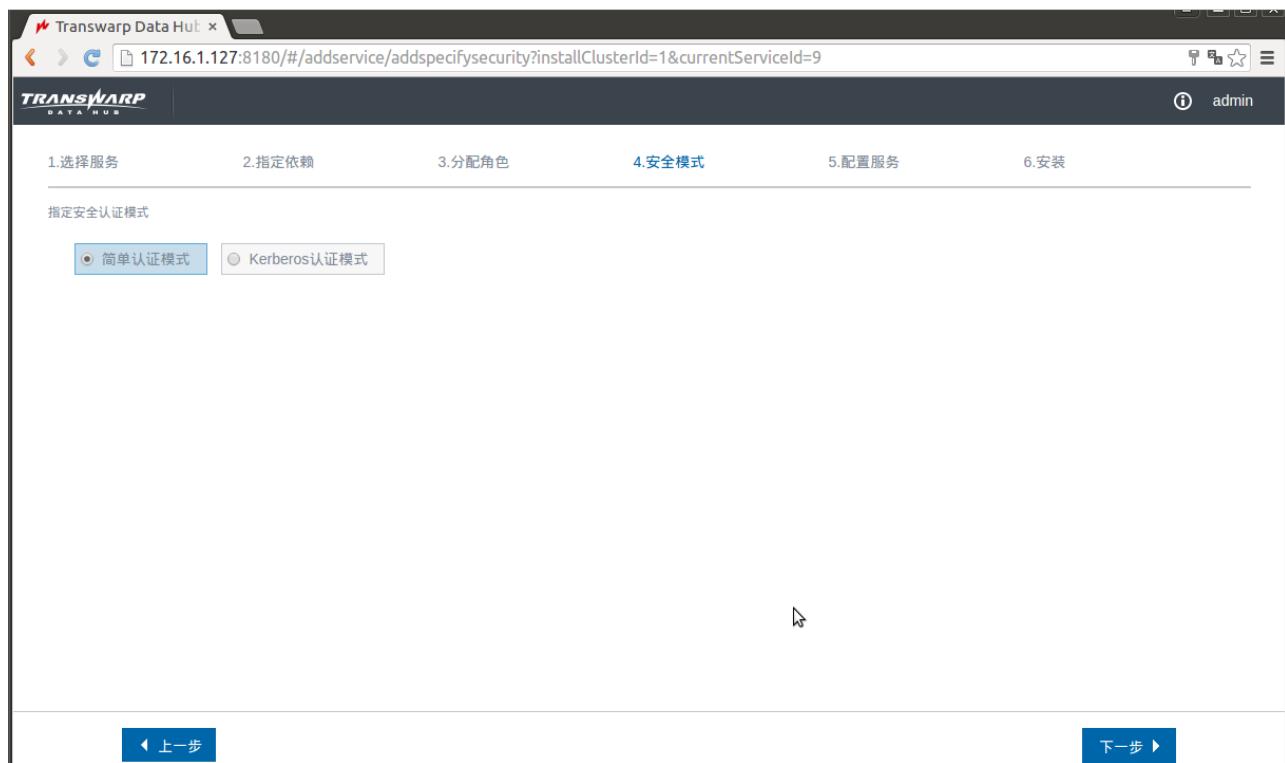
为选择的服务指定其依赖的服务

Zookeeper	HDFS	YARN	ELASTICSEARCH	Hyperbase	Inceptor
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1			
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1		Hyperbase1	
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1			InceptorSQL1
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1			StreamSQL1
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1		Hyperbase1	InceptorSQL1
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1		Hyperbase1	StreamSQL1
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1	Elasticsearch1		
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1	Elasticsearch1	Hyperbase1	
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1	Elasticsearch1		InceptorSQL1
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1	Elasticsearch1		StreamSQL1
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1	Elasticsearch1	Hyperbase1	InceptorSQL1
<input checked="" type="radio"/> ZooKeeper1	HDFS1	YARN1	Elasticsearch1	Hyperbase1	StreamSQL1

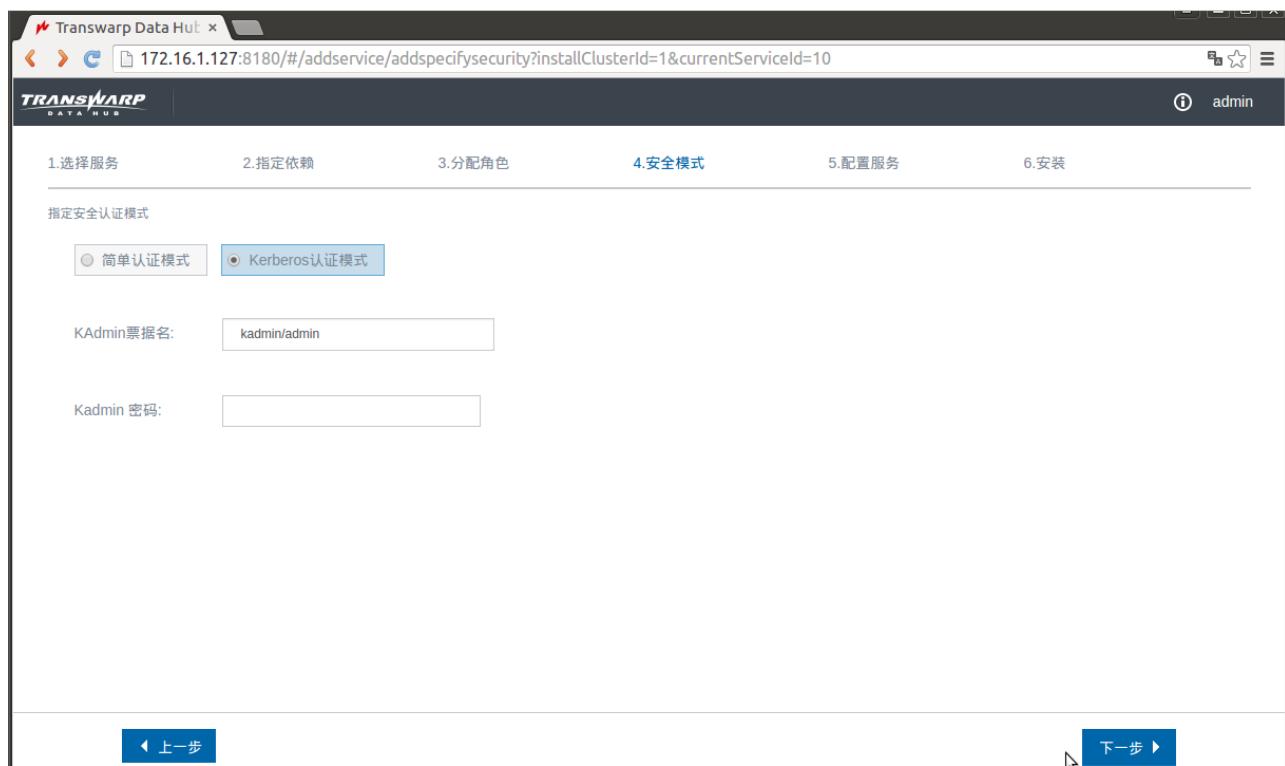
◀ 上一步 下一步 ▶

一个Inceptor服务依赖于另一个Inceptor服务意味着它们将共用一个metastore。而一个Inceptor服务依赖于一个Hyperbase服务意味着您可以通过该Inceptor服务操作它依赖的Hyperbase中的数据。这里您可以根据自己的需要选择。选择完毕后点解“下一步”继续。

5. 在该页面您需要指定集群中各节点在将要安装的Inceptor服务中的角色——指定哪个节点将是Inceptor server以及哪个节点将是metastore（注意，如果您在上一步中指定了将要安装的Inceptor服务依赖于已经安装的Inceptor服务，那么您在这一步将不能再自己选择metastore所在节点）。选择完毕后，点击“下一步”继续。
6. 在该页面您需要为新添加的Inceptor选择它的认证模式。如果您的集群已经在 安全模式 下（整个集群都开启了Kerberos），那么您需要为您的集群选择Kerberos认证模式。您也可以在这个Inceptor服务安装完毕后去服务页面启用Kerberos。



如果您选择了“Kerberos认证模式”，您需要输入Kadmin的principal名和kadmin的密码：



选择完毕后，点击“下一步”继续。

- 在这里您将需要设置一些Inceptor服务的参数。点击下图中的“高级参数”：

配置选中的服务

属性	基础参数	高级参数	自定义参数	资源分配
Inceptor1		高级参数	值	
		hive.server2.enabled	TRUE	
		hive.server2.authentication	NONE	

上一步 **下一步**

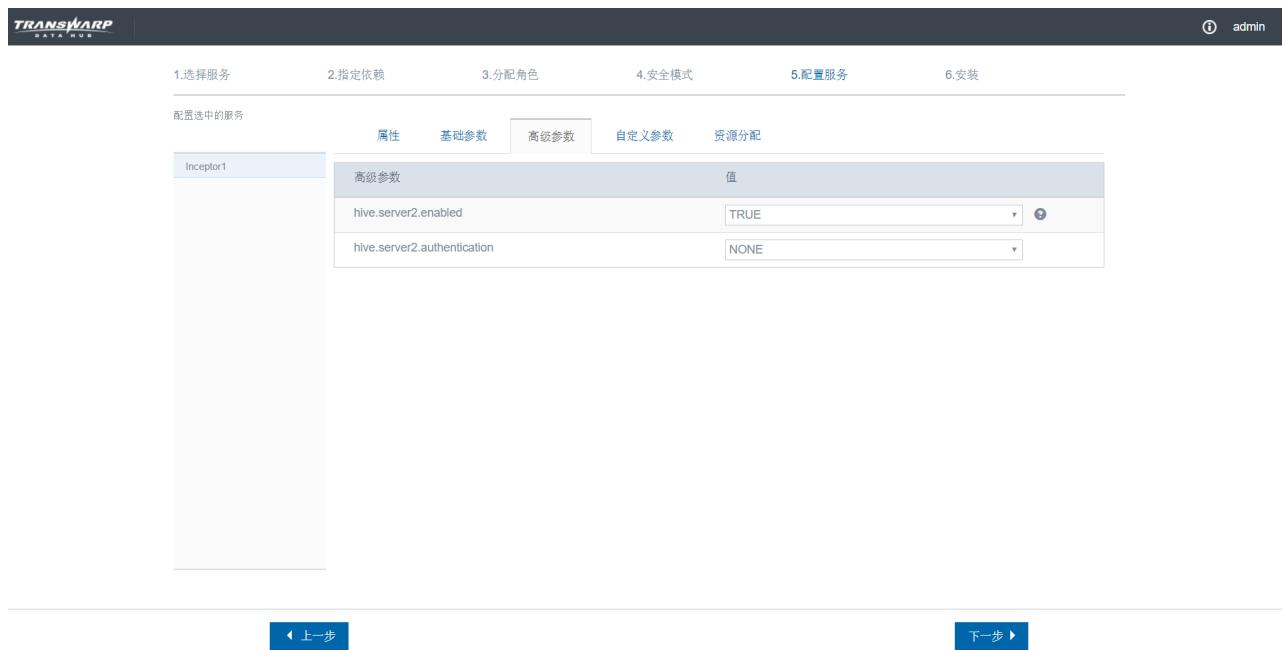
这里是您选择为Inceptor服务启用Hive Server 1还是Hive Server 2的地方，这里也是您选择Hive Server 2是否用LDAP认证的地方。注意，这两个选择在Inceptor安装完成后就不能再更改：

配置选中的服务

属性	基础参数	高级参数	自定义参数	资源分配
Inceptor1		高级参数	值	
		hive.server2.enabled	FALSE	

上一步 **下一步**

如果您想要使用Hive Server 2，您需要将hive.server2.enabled参数的值设置为TRUE，设置为FALSE为使用Hive Server 1。如果您将hive.server2.enabled 的值设为TRUE，您将会看到第二个参数：hive.server2.authentication

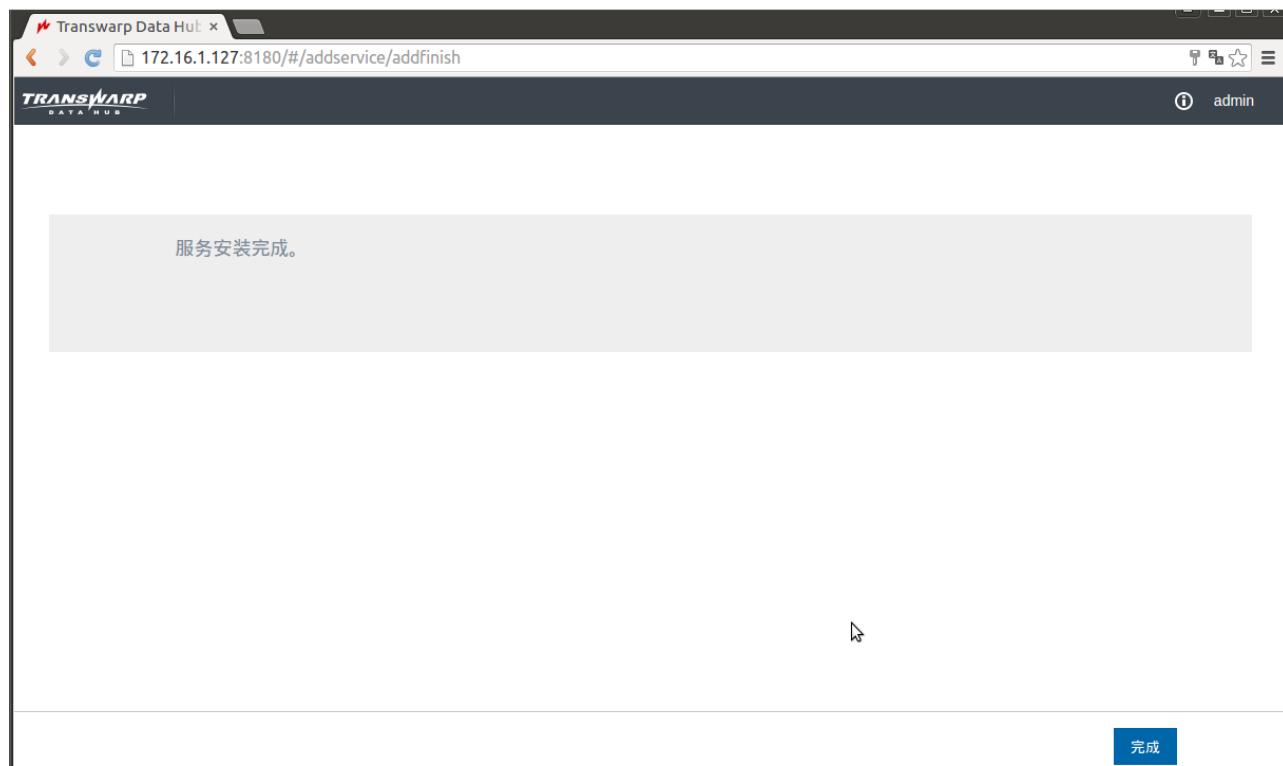


这里您可以选择是否使用LDAP对您将要安装的Inceptor服务进行验证。如果您想要用LDAP认证，那么您需要将hive.server.authentication参数的值设置为LDAP，那么您将会看到两个新的参数：

您需要将hive.server2.authentication.ldap.baseDN参数的值设置为ou=people, dc=TDH;
将hive.server2.authentication.ldap.url的值设置为您的LDAP服务所在的节点。这两个参数也可以在Inceptor服务安装完毕后通过服务页面设置。

设置完毕后点击“下一步”继续。

8. 在这里您可以检查您将要安装的Inceptor服务的各项参数，点击“下一步”继续。
9. 新的Inceptor服务马上就要进行安装，开始安装前，安装向导会向您进行确认。
10. 安装开始后会持续一段时间。
11. 安装完成后点击“继续”。
12. 在该页面点击“完成”退出安装向导，完成安装：

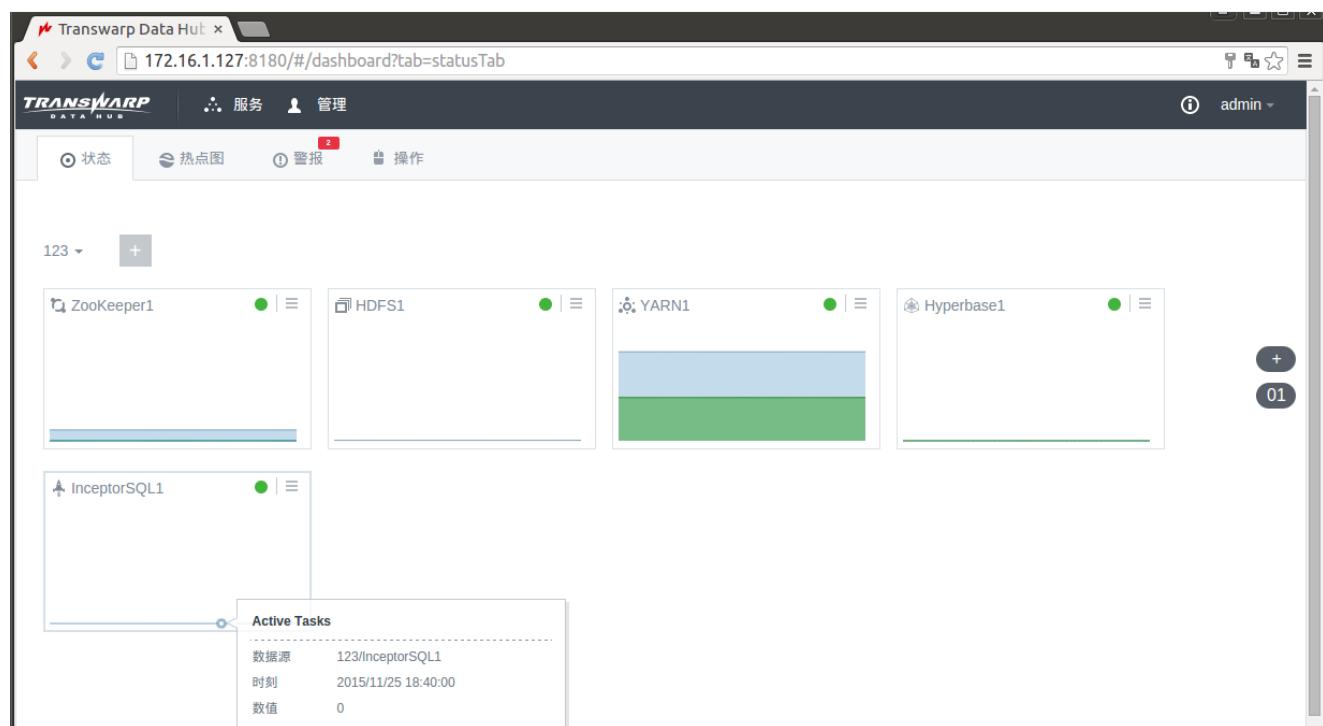


13. 回到Transwarp Manager的页面您将能看到新安装的Inceptor服务。

11.3. Inceptor CPU和内存资源分配

Inceptor的CPU和内存资源可以通过管理界面上的 **Inceptor资源分配** 页面 进行。

登陆Transwarp Manager后，点击首页上的Inceptor服务进入Inceptor服务页面：



Inceptor_SQL | HEALTHY

2015/11/25 18:19:47 — 2015/11/25 18:49:47

当前集群时间: 2015/11/25 18:49:47

30m 1h 2h 6h 12h 1d 1w 30d Now

16:00 16:15 16:30 16:45 17:00 17:15 17:30 17:45 18:00 18:15 18:30 18:45

个数 Active Tasks

Server tw-node128 Executors 3

Completed Tasks 1425 Failed Tasks 0

日志

服务历史

级别 时间 分类 标题

没有查询到历史.

Inceptor-SQL 统计

Shuffle Read MB/s

点击页面左上的箭头打开菜单栏:

概要

角色

配置

资源分配

安全

操作

Holodesk

Inceptor_SQL | HEALTHY

2015/11/25 18:20:17 — 2015/11/25 18:50:17

当前集群时间: 2015/11/25 18:50:17

30m 1h 2h 6h 12h 1d 1w 30d Now

16:00 16:15 16:30 16:45 17:00 17:15 17:30 17:45 18:00 18:15 18:30

个数 Active Tasks

Server tw-node128 Executors 3

Completed Tasks 1425 Failed Tasks 0

日志

服务历史

级别 时间 分类 标题

没有查询到历史.

Inceptor-SQL 统计

Shuffle Read MB/s

点击“资源分配”，进入 Inceptor 资源分配页面：

The screenshot shows the 'Resource Allocation' configuration for the 'Inceptor1' service in the Transwarp Data Hub. Key settings include:

- 运行Application的Queue:** default
- Application Master内存:** 512
- Executor资源:** Ratio (selected)
- Memory : core比例:** 0.34440103
- core百分比:** 0.501
- Executor 数量:** 每个worker上只运行一个executor (selected)
- Executor 分布:** 每个worker上都运行executor (selected)

我们先介绍页面上一些名词的概念。

- 运行Application的 **Queue**

YARN管理的任务队列。队列的创建和管理可以通过 **YARN资源管理** 页面进行。

- **Application Master**

Application Master负责为Inceptor向集群申请和获取资源。通常情况下Inceptor Application Master占用资源比较小，使用默认值512MB即可。

- **Worker**

集群中的服务器。

- **Executor**

在服务器上执行Inceptor任务的进程。您可以设置Executor资源、数量和分布。

下面介绍页面上的各项设置：

- 运行Application的**Queue**

您可以在那里选择在YARN中的哪个Queue中运行Inceptor。

- **Application Master内存**

这是 Inceptor Application Master占用内存的大小，通常情况下该进程占用资源比较小，使用默认值512MB即可。

- **Executor资源**

在这里您可以对executor的内存和CPU资源进行设置。Executor资源设置有两个选项：Fixed和Ratio。

- Fixed（适合同构系统）

“Fixed” 对应 固定 的executor资源。每个executor会分配到固定的CPU内核数（由“Executor 内核数”设置）和固定的内存（由“Executor 内存”设置）。例如下面的设置让每个executor都能分配到4个CPU内核和385MB的内存：

Executor资源:	<input checked="" type="radio"/> Fixed	<input type="radio"/> Ratio
Executor 内核数:	<input type="text" value="4"/>	
每个executor所使用的core的数量.		
Executor 内存:	<input type="text" value="385"/>	
每一个executor使用的内存(MB).		

- Ratio（适合异构系统）

“Ratio” 对应 按比例分配 executor资源。这里的涉及到两个比例：

- Memory: core比例:

Executor分配到的内存(GB)与CPU核数量的比例。例如，如果Memory: core比例 = 2，而executor分配到1个CPU核，那么executor将分配到2GB的内存。

- core百分比:

异构系统中每个executor使用的CPU核数量占所在节点CPU核总量的比例。例如，如果core百分比 = 0.1，而一个节点上有10个CPU核，那么该节点上的每executor将使用1个CPU。

结合这两个比例可以得到executor分配到的CPU核数和内存。例如，假设一个节点上有10个CPU核，并进行下面设置：

Executor资源:	<input type="radio"/> Fixed	<input checked="" type="radio"/> Ratio
Memory : core比例:	<input type="text" value="0.8"/>	
内存(GB)与CPU core数的比例。例如, 如果这个比例为2, 而CPU core数为1, 那么内存使用量就为2GB.		
core百分比:	<input type="text" value="0.5"/>	
异构系统中每个executor使用的core数量占所在节点core总量的比例。例如, 如果这个比例为0.1, 而一个节点上有10个core, 那么该节点上的executor将使用1个core.		

根据设置，我们可以算出：

- Executor分配到的CPU核数 = 节点CPU核总数 × core百分比 = $10 \times 0.5 = 5$
- Executor分配到的内存 = Executor分配到的CPU核数 × (Memory:core比例) = $5 \times 0.8 = 4$ (GB)

- Executor数量

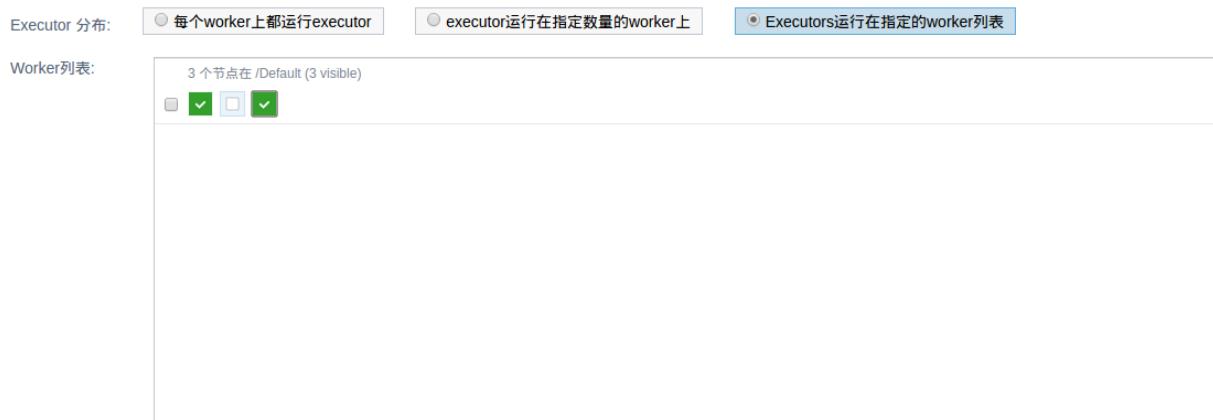
在这里指定集群中的Executor数量，您可以选择在“每个Worker上只运行一个Executor”还是在整个集群中“总共运行指定数目的Executor”。

- Executor分布

在这里您可以选择如何在集群中的各个节点上安排executor。选项有：

- “每个worker上都运行executor”。
- “Executor运行在指定数量的worker上”。选择这个选项可以指定集群中有多少台服务器运行executor，但是并不能指定是由哪几台服务器运行。
- “Executors”运行在指定的worker列表”。选择这个选项可以指定集群中具体哪几台服务器上会运行executor。

根据下面的设置，集群中的第一台和第三台服务器上会运行executor：



- Inceptor服务的启动和任务的调度由YARN管理。在每个worker上，executor靠这个worker上YARN NodeManager来启动和管理，所以每个worker上分配给executor的总资源（总CPU核数和总内存）必须小于这个worker上的NodeManager的资源，否则这台worker上的executor将无法启动。
- 如果您进行的资源分配超出集群中实际的资源。例如集群中每个节点的CPU核数量为8，但配置了每个节点上两个executor，每个executor分配5个核——这个设置所需的CPU数量超出了节点的CPU数量。在这种情况下，集群会无法启动executor，Inceptor服务也就无法正常运行。所以请管理员在进行CPU和内存资源分配时要符合集群实际的硬件状况。



完成所有的设置后，您需要点击Inceptor资源分配页面右上角的“Save”来保存，然后点击“更多操作”下的“配置服务”来将改动配置到后台。最后您需要重启Inceptor服务使改动生效。

附录 A: Inceptor字段规范



版本信息

从TDH4.6开始，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

12. 客户服务

技术支持

感谢你使用星环信息科技（上海）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

技术支持QQ专线: 3221723229, 3344341586

官方网址: www.transwarp.io

意见反馈

如果你在系统安装，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

感谢你的支持和反馈，我们一直在努力！