QianBase SP/SQL 语言参考 1.6.6

2020 年 12 月

目录

前言

修订历史

版本	日期	描述
1.1.0	2019 年 7 月	本手册介绍了 QianBase 产品支持的 SQL 语句，函数和其他 SQL 语言元素的语法的参考信息。
1.2.0	2019 年 9 月	新增存储例程介绍
1.3.0	2020 年 1 月	新增触发器使用内容
1.6.6	2020 年 12 月	版本升级

本文简介

本指南介绍了 QianBase 产品支持的 SQL 语句，函数和其他 SQL 语言元素的语法的参考信息。

目标读者

本指南的目标读者为 QianBase 系统管理员和用户

相关文档

本指南为 QianBase 文档库的一部分，QianBase 文档库**包括但不限于**以下文档:

文档名称	说明

QianBase 安装部署指南	本文介绍安装 QianBase，包括安装前准备、安装 Hadoop 发行版、故障排除、配置、启用安全功能、提高安全性和卸载 QianBase 等。
易鲸捷 Designer 用户指南	本文介绍易鲸捷图形化数据库管理工具
易鲸捷迁移工具用户指南	本文介绍如何安装和使用易鲸捷迁移工具。
QianBase 技术白皮书	本文介绍 QianBase 技术架构，组件介绍，技术特点等。
QianBase 数据库规划文档	本文介绍节点数量规划、数据目录和安装部署目录规划、集群角色分配规划等。
QianBase 管理员手册	本文介绍 QianBase 的日常运维常用系统命令、常用检查 SQL，用户权限配置，连接设置等内容。
QianBase 常见问题提排查与解决	本文介绍如何排查和解决 QianBase 的常见问题。
QianBase 灾难恢复手册	本文介绍 QianBase 灾难恢复设计原理，方案建议以及使用手册。
QianBase 备份恢复手册	本文介绍 QianBase 备份恢复设计原理，方案建议以及使用手册。
QianBase 数据库扩容指南	本文介绍 QianBase 如何更换节点，增加节点，删除节点等操作。
QianBase 数据库参数调优建议	本文介绍如何进行数据模型优化，CQD 参数优化等 。
QianBase 客户端安装手册	本文介绍 QianBase JDBC， ODBC 以及 Trafci 驱动安装。

QianBase JDBC 程序员参考指南	本文介绍 QianBase JDBC 驱动连接设置，开发人员指南。
QianBase ODBC 程序员参考指南	本文介绍 QianBase ODBC 驱动连接设置，开发人员指南。
QianBase SPSQL 存储过程用户手册	本文介绍 QianBase SPSQL 存储过程的使用。
Esgyn DBManager 用户手册	本文介绍图形化数据库监控运维工具 DB Manager 的使用。
QianBase 数据库迁移指南	本文介绍如何将常见关系型数据库（Oracle、MySQL、SQL server 等）迁移至 QianBase。
QianBase SQL 用户手册	本文是 QianBase 的 SQL 使用手册。

批评与建议

我们支持您对本指南做出的任何批评与建议，并尽力提供符合您需求的文档。

若您发现任何错误、或有任何改进建议，请发邮件至 support@esgyn.cn。

1. 简介

1.1 存储例程

存储例程是使用 SQL 代码定义的，可执行的 schema 对象。

SPSQL 当前支持三种存储例程:

- 存储过程: 存储过程由 CREATE PROCEDURE 创建并由 CALL 语句调用。过程可以有用于输入和输出的参数，但没有返回值。过程还可以生成结果集作为输出返回给客户端。
- 函数: 函数由 CREATE FUNCTION 创建，可以在表达式中调用。函数可以有输入参数和返回值。函数不能有输出参数或结果集输出。
- 包（package）: 包由 CREATE PACKAGE 创建。包是一组相关变量，过程和函数的容器。包本身不能调用，只能调用包中定义的过程和功能。当前只能在包过程和函数中访问包中定义的变量。

1.2 使用存储的例程

1.2.1 使用存储过程

存储过程由 CREATE PROCEDURE 语句定义。 您可以为要定义的过程指定名称，参数，SQL 语句主体和可选属性。

```
CREATE PROCEDURE greeting(IN name VARCHAR(100), OUT v_result
VARCHAR(200)) AS
    BEGIN
    SET result = 'Hello, ' || name || '!';
    END;
/
```

这将定义一个名为“ greeting”的存储过程，该存储过程接受两个参数，每个参数都有一个可选的方向类型，名称和数据类型。 第一个参数的方向类型为 IN，这意味着它是一个输入参数，参数的名称为“ name”，数据类型为 VARCHAR（100）。 第二个参数用于输出（方向类型 OUT）。 SPSQL 还支持另一个参数方向类型 INOUT，该类型可以用于输入和输出。 省略方向类型时，默认为 IN。

成功定义该过程后，可以使用 CALL 语句将其调用。

```
CALL greeting('SPSQL', ?);

RESULT
------------------------------------------------------------
------------------------------------------------------------
--
Hello, SPSQL!

--- SQL operation complete.
```

1.2.2 使用函数

函数由 CREATE FUNCTION 语句定义。

```
CREATE FUNCTION hello(name VARCHAR(100)) RETURN VARCHAR(200)
AS
BEGIN
  return 'Hello, ' || name || '!';
END;
/
```

函数只能有输入参数，并且必须有返回值。 定义之后，可以在可以使用表达式的任何位置调用函数。

```
SELECT hello('SPSQL') FROM DUAL;

(EXPR)
------------------------------------------------------------
------------------------------------------------------------
--
Hello, SPSQL!

--- 1 row(s) selected.
```

1.2.3 使用包

包可以通过 CREATE PACKAGE 语句定义。 包由规格和主体组成。 规范定义接口，主体定义实现。

包的规格:

```
CREATE OR REPLACE PACKAGE users AS
  session_count INT := 0;
  FUNCTION get_count() RETURN INT;
  PROCEDURE f_add(name VARCHAR(100));
END;
/
```

包主体:

```
CREATE OR REPLACE PACKAGE BODY users AS
  FUNCTION get_count() RETURN INT
  IS
  BEGIN
    RETURN session_count;
  END;
  PROCEDURE add(name VARCHAR(100))
  IS
  BEGIN
    session_count = session_count + 1;
  END;
END;
/
```

包中的调用过程和功能:

```
CALL users.add('John');
CALL users.add('Sarah');
CALL users.add('Paul');
SELECT users.get_count() FROM DUAL;

(EXPR)
------------------

3

--- 1 row(s) selected.
```

1.3 日志记录

1. 简介

SPSQL 使用 log4j 进行日志记录，每个服务器节点上日志文件的位置为 $ TRAF_HOME / logs / trafodion.sql.java.log。

注意：此日志文件还用于其他基于 JAVA 的 trafodion 组件。

日志记录配置文件位于$ TRAF_CONF / log4j.sql.config 中：

log4j.logger.org.trafodion.sql.udr.spsql = INFO, sqlAppender

默认情况下，日志记录级别为 INFO，您可以将日志记录级别更改为 TRACE 或 ALL，以获取更多的执行信息，以进行调试或分析。

log4j.logger.org.trafodion.sql.udr.spsql = ALL，sqlAppender

SPSQL 启动时，它将以下消息写入日志文件：

2019-08-20 10:34:22.867 ,INFO ,org.trafodion.sql.udr.spsql.SPSQL ,======== Starting SPSQL ========

在日志消息中，有四个部分用逗号（，）分隔：

- Timestamp: 2019-08-20 10:34:22.867
- Log Level: INFO
- Class: org.trafodion.sql.udr.spsql.SPSQL
- Message: ======== Starting SPSQL ========

启用 TRACE 级别后，在 SPSQL 中执行的所有 SQL 语句也将记录在日志文件中：

查询语句：

2019-08-20 10:34:27.807 ,TRACE ,org.trafodion.sql.udr.spsql.Conn ,Execute Query: SELECT * FROM t1

非查询语句：

2019-08-20 10:34:29.793 ,TRACE ,org.trafodion.sql.udr.spsql.Conn ,Execute SQL: INSERT INTO t1 VALUES (1)

如果执行成功，还可以找到每个 SQL 语句的执行时间：

Query executed successfully (12 ms)

当执行 SPSQL 时出现错误时，将记录带有回溯的错误消息：

2019-09-18 03:39:08.977 ,ERROR ,org.trafodion.sql.udr.spsql.Exec ,Unhandled exception in SP/SQL
2019-09-18 03:39:08.977 ,ERROR ,org.trafodion.sql.udr.spsql.SPSQL ,SQL exception:
java.sql.SQLException: *** ERROR[8102] The operation is prevented by a unique constraint.
    at org.apache.trafodion.jdbc.t2.SQLMXStatement.executeDirect(Native Method)
    at org.apache.trafodion.jdbc.t2.SQLMXStatement.execute(SQLMXStatement.java:139)
    at org.trafodion.sql.udr.spsql.Conn.executeSql(Conn.java:163)
    at org.trafodion.sql.udr.spsql.Exec.executeSql(Exec.java:1370)
    at org.trafodion.sql.udr.spsql.Stmt.insertValues(Stmt.java:905)
    at org.trafodion.sql.udr.spsql.Stmt.insert(Stmt.java:788)
.....

2. 用户定义的 SQL 过程

2.1 CREATE PROCEDURE

CREATE PROCEDURE 允许您创建用户定义的 SQL 过程（存储过程）。

2.1.1 语法

```
CREATE [OR REPLACE] PROCEDURE procedure_name[parameters]
[AS | IS]
body
```

参数:

```
([IN | OUT | INOUT | IN OUT] name data_type, ...)
|
(name [IN | OUT | INOUT | IN OUT] data_type, ...)
```

主体:

```
[declare_block] BEGIN statements [exception_block] END
```

2.1.2 示例

```
CREATE OR REPLACE PROCEDURE proc1() AS
DECLARE
```

```
  v VARCHAR(200);
BEGIN
  OPEN cur FOR 'SELECT c1 FROM t1';
  FETCH cur INTO v;
  CLOSE cur;
EXCEPTION
  WHEN OTHERS THEN
    PRINT 'Error';
END
```

2.2 DROP PROCEDURE

DROP PROCEDURE 语句删除用户定义的 SQL 过程。

2.2.1 语法

```
DROP PROCEDURE [IF EXISTS] procedure_name;
```

2.2.2 示例

```
DROP PROCEDURE proc1;
DROP PROCEDURE IF EXISTS proc2;
```

3. 数据类型

与 SP/SQL 相关联的数据类型有两组，一组是用于定义例程参数的数据类型，另一组是在例程主体中使用的数据类型。

3.1 布尔数据类型

布尔数据类型仅在例程主体中可用。

3.1.1 语法

在例程主体中：

```
BOOL | BOOLEAN
```

3.2 字符串数据类型

固定长度和可变长度字符数据类型。

3.2.1 语法

在例程参数中：

```
CHAR[ACTER] [(length [unit])] [char-set]
CHAR[ACTER] VARYING (length [unit]) [char-set]
VARCHAR (length [unit]) [char-set]
VARCHAR2 (length [unit]) [char-set]
NCHAR [(length [unit])]
NCHAR VARYING (length [unit])
NATIONAL CHAR[ACTER] [(length [unit])]
NATIONAL CHAR[ACTER] VARYING (length [unit])
```

在例程主体中：

```
CHAR [(length [unit])] [char-set]
VARCHAR [(length [unit])] [char-set]
VARCHAR2 [(length [unit])] [char-set]
NCHAR [(length [unit])]
```

注：

- length 是一个正整数，指定列中允许的字符数（或字节数，见下文）。您必须指定例程参数的长度值。例程主体中的长度是可选的。
- unit 是 CHAR[ACTER[S]] 或 BYTE[S] 的可选单位。默认值为 CHAR[ACTER[S]]。此单位仅对 UTF8 字符有意义。
- char-set 是字符集名称，可以是 ISO88591、 UTF8 或 UCS2。

3.3 游标数据类型

游标数据类型仅在例程主体中可用。

3.3.1 语法

在例程主体中：

```
SYS_REFCURSOR
```

3.4 时间日期数据类型

日期时间类型可以表示日期、时间或日期和时间

3.4.1 语法

在例程参数中：

```
DATE
TIME [(precision)]
TIMESTAMP [(precision)]
```

在例程主体中：

```
DATE
TIME [(precision)]
DATETIME [(precision)]
TIMESTAMP [(precision)]
```

注：

- DATE 的格式是 yyyy-mm-dd。
- TIME 的格式是 hh:mm:ss.ffffff。
- TIMESTAMP 的格式是 yyyy-mm-dd hh:mm:ss.ffffff。
- precision 是一个无符号整数，指定小数秒中的位数，并存储在四个字节中。时间精度的默认值为 0，最大值为 6。
- 在例程主体中，DATETIME 是 TIMESTAMP 的同义词。

3.5 数字数据类型

数字数据类型可以是精确值，也可以是近似值。数字数据类型与任何其他数字数据类型兼容。

3.5.1 语法

例程参数中的精确数字类型：

```
NUMERIC [(precision [,scale])]
SMALLINT
INT[EGER]
LARGEINT
DEC[IMAL] [(precision [,scale])]
```

例程参数中的近似数字类型：

```
FLOAT
REAL
DOUBLE PRECISION
```

例程主体中的精确数字类型：

```
TINYINT
SMALLINT | INT2
INT[EGER] | INT4 | PLS_INTEGER| BINARY_INTEGER
LARGEINT | BIGINT | INT8
DEC[IMAL] [(precision [,scale])]
NUMERIC [(precision [,scale])]
NUMBER [(precision [,scale])]
```

例程主体中的近似数字类型：

```
FLOAT | BINARY_FLOAT
REAL
DOUBLE [PRECISION] | BINARY_DOUBLE
```

4. 声明

4.1 DECLARE 语句块

DECLARE 语句块是 BEGIN..END 语句块之前的变量声明块。

4.1.1 语法

```
DECLARE
  var datatype [NOT NULL] [:= | = | DEFAULT expression];
  ...
BEGIN
  ...
END;
```

4.1.2 实例

```
DECLARE
  code CHAR(10);
  status INT := 1;
  count SMALLINT = 0;
  limit INT DEFAULT 100;
  max_limit CONSTANT INT := 1000;
BEGIN
  ...
END;
```

4.2 DECLARE 语句

DECLARE 语句可用于声明同一类型的一个或多个变量。

4.2.1 语法

```
 DECLARE var [, var2, ...] [AS] datatype [:= | = | DEFAULT
expression] [, ...];
```

注:

声明语句只能在 BEGIN…END 语句块中使用，而声明块只能在 BEGIN…END 语句块之前使用。

4.2.2 实例

```
BEGIN
  ...
  DECLARE code CHAR(10);
  DECLARE status, status2 INT DEFAULT 1;
  DECLARE count SMALLINT, limit INT DEFAULT 100;
  ...
END;
```

5. 赋值

5.1 赋值运算符

赋值运算符（:= 或 =）可以用来设置值。

5.1.1 语法

```
 var [:= | = ] expression;
```

5.1.2 实例

```
code := 'A';
status := 1;
count = 0;
```

5.2 赋值语句

也可使用 SET 语句来赋值。

5.2.1 语法

```
 SET var = expression [, ...];
SET (var [, var2, ...]) = (expression [, expression2, ...]);
```

5.2.2 实例

```
SET code = 'A';
SET status = 1, count = 0;
SET (count, limit) = (0, 100);
```

5.3 SELECT 语句的赋值

还可使用 SET 语句从查询结果的第一行赋值:

5.3.1 语法

```
SET var = (SELECT col FROM ...);
SET (var [, var2, ...]) = (SELECT col [, col2, ...] FROM ...);
SELECT var = col [, var2 = col2, ...] FROM ...;
```

5.3.2 实例

```
SET code = (SELECT code FROM conf WHERE name = 'A');
SET (count, limit) = (SELECT count, limit FROM conf WHERE
name = 'A');
SELECT @count = count, @limit = limit FROM conf WHERE name =
'A';
```

6. 控制流

6.1 BREAK 语句

BREAK 语句退出最内层循环。

6.1.1 语法

```
BREAK;
```

6.1.2 实例

```
DECLARE count INT DEFAULT 3;
WHILE 1=1 BEGIN
  SET count = count - 1;
  IF count = 0
    BREAK;
END
```

6.2 CASE 表达式

CASE 表达式执行 IF-THEN-ELSE 逻辑。

6.2.1 语法

简单 CASE 表达式:

```
CASE expr
  WHEN expr THEN expr
  ...
  [ELSE expr]
END
```

搜索 CASE 表达式:

```
CASE
  WHEN boolean_expr THEN expr
  ...
  [ELSE expr]
END
```

6.2.2 实例

简单 CASE 表达式:

```
CASE state
   WHEN 'AZ' THEN 'Arizona'
   WHEN 'CA' THEN 'California'
   ELSE 'N/A'
END
```

搜索 CASE 表达式:

```
 CASE
   WHEN state = 'AZ' THEN 'Arizona'
   WHEN state = 'CA' THEN 'California'
   ELSE 'N/A'
 END
```

6.3 EXIT WHEN 语句

EXIT WHEN 语句退出由给定标签标记的循环或块。如果未指定标签，则 EXIT 将退出最内层循环。

如果指定的是布尔表达式，并且计算结果为 true，则执行 EXIT 语句，否则将忽略该表达式，并继续执行 EXIT 后的语句。

6.3.1 语法

```
EXIT [label] [WHEN boolean_expression];
```

6.3.2 实例

```
WHILE count > 0 LOOP
  count := count - 1;
  EXIT WHEN count = 0;
END LOOP;
```

```
<<lbl>>
WHILE 1=1 LOOP
  <<lbl1>>
  WHILE 1=1 LOOP
    EXIT lbl;
  END LOOP;
END LOOP;
```

6.4 FOR 语句（游标循环）

FOR 语句打开一个游标，对每一行重复执行一个或多个语句，然后关闭游标。

6.4.1 语法

```
FOR cur_name IN [() select_stmt ()] LOOP
  statements
END LOOP;
```

注：可使用 cur_name.col_name 语句参考游标列

6.4.2 实例

```
FOR item IN (
    SELECT dname, loc as location
    FROM dept
    WHERE dname LIKE '%A%'
    AND deptno > 10
    ORDER BY location)
LOOP
  DBMS_OUTPUT.PUT_LINE('Name = ' || item.dname || ', Location
= ' || item.location);
END LOOP;
```

6.5 FOR 语句（整数范围）

对于指定的整数值范围，FOR 语句用来重复执行一个或多个语句。

6.5.1 语法

```
FOR index IN [REVERSE] lower_bound..upper_bound [BY | STEP
increment] LOOP
 statements
END LOOP;
```

注：

- index -隐式声明的整数变量。
- 如果指定 REVERSE，则索引将减少。
- 如果指定 BY (或 STEP)定义迭代步骤，默认值为 1。

6.5.2 实例

```
FOR i IN 1..10 LOOP
  -- i will have values: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
END LOOP;
```

```
FOR i IN REVERSE 1..10 LOOP
  -- i will have values: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
END LOOP;
```

```
FOR i IN 1..10 BY 2 LOOP
  -- i will have values: 1, 3, 5, 7, 9
END LOOP;
```

6.6 IF 语句

IF 语句根据布尔表达式的值执行一组语句。

QianBase 支持 IF 语句的多个语法。

6.6.1 语法

6.6.1.1 IF - THEN - ELSIF/ELSEIF - ELSE - END IF

```
IF boolean_expression THEN
  statements
[ELSIF | ELSEIF THEN
  statements
...]
[ELSE
  statements]
END IF;
```

6.6.1.2 IF - BEGIN - END - ELSE - BEGIN – END

```
IF boolean_expression
  single_statement | block
[ELSE
  single_statement | block];
```

6.6.1.3 .IF THEN

```
.IF boolean_expression THEN statement
```

6.6.2 实例

6.6.2.1 IF - THEN - ELSIF/ELSEIF - ELSE - END IF

```
IF state = 'CA' THEN
  code := 1;
ELSIF state = 'NY' THEN
  code := 2;
ELSIF state = 'MA' THEN
  code := 3;
ELSE
```

```
  code := 5;
END IF;
```

6.6.2.2 IF - BEGIN - END - ELSE - BEGIN – END

```
IF state = 'CA'
  SET code = 1;
ELSE
  SET code = 5;
```

```
IF state = 'CA'
BEGIN
  SET code = 1;
  SET type = 'A';
END
ELSE
BEGIN
  SET code = 5;
  SET type = 'B';
END
```

6.7 LEAVE 语句

LEAVE语句退出由给定标签标记的循环或块。如果未指定标签，则LEAVE退出最内层循环。

6.7.1 语法

```
LEAVE [label];
```

6.7.2 实例

```
lbl:
WHILE count > 0 DO
  SET count = count - 1;
  IF count = 0 THEN
    LEAVE lbl;
  END IF;
END WHILE;
```

```
lbl:
WHILE 1=1 DO
  lbl1:
  WHILE 1=1 DO
    LEAVE lbl;
  END WHILE;
END WHILE;
```

6.8 LOOP 语句

LOOP 语句执行一个或多个语句，直到使用 EXIT、LEAVE 或 BREAK 语句退出循环或引发异常为止。

6.8.1 语法

```
[<<label>> | label:]
LOOP
  statements
END LOOP;
```

6.8.2 实例

```
LOOP
  count := count - 1;
  EXIT WHEN count = 0;
END LOOP;
```

```
lbl:
LOOP
  SET count = count - 1;
  IF count = 0 THEN
    LEAVE lbl;
  END IF;
END LOOP;
```

6.9 WHILE 语句

WHILE 语句在条件为 true 时执行一个或多个语句。

6.9.1 语法

```
[<<label>> | label:]
WHILE boolean_expression LOOP | DO | BEGIN
  statements
END [LOOP | WHILE;]
```

6.9.2 实例

```
-- Oracle, PostgreSQL, Netezza
WHILE count > 0 LOOP
  count := count - 1;
END LOOP;
```

```
-- DB2, Teradata, MySQL
WHILE count > 0 DO
  SET count = count - 1;
END WHILE;
```

```
-- SQL Server
WHILE count > 0 BEGIN
  SET count = count - 1;
END
```

7. 游标

7.1 CLOSE 语句

CLOSE 语句关闭游标。

7.1.1 语法

```
CLOSE cursor_name;
```

参数	类型	值	描述
cursor_name		标识符	以前打开的游标名称。

7.1.2 实例

```
DECLARE id INT;
DECLARE cur CURSOR FOR 'SELECT id FROM db.orders';
OPEN cur;
FETCH cur INTO id;
CLOSE cur;
```

7.2 Cursor 属性

Cursor 属性获取有关当前游标状态的信息。

7.2.1 语法

```
cursor_name%ISOPEN
cursor_name%FOUND
cursor_name%NOTFOUND
```

- cursor_name 是已声明的游标或游标变量的名称。
- %ISOPEN 在游标打开时返回 true，否则返回 false。
- %FOUND 在没有从游标提取数据时则值为 NULL，如果从游标提取数据返回一行则值为 TRUE，否则为 FALSE。
- %NOTFOUND 在没有从游标提取数据时值为 NULL，如果从游标提取数据返回移行则值为 FALSE，否则为 TRUE。

7.2.2 实例

```
DECLARE
   v1 VARCHAR(30);
   CURSOR c1 IS SELECT name FROM users LIMIT 1;
   BEGIN
   OPEN c1;
   IF c1%ISOPEN THEN
     DBMS_OUTPUT.PUT_LINE('Cursor open');
   END IF;
   FETCH c1 INTO v1;
   IF c1%FOUND THEN
     DBMS_OUTPUT.PUT_LINE('Row found');
   END IF;
   FETCH c1 INTO v1;
   IF c1%NOTFOUND THEN
     DBMS_OUTPUT.PUT_LINE('Row not found');
    END IF;
    CLOSE c1;
END;
```

7.3 DECLARE CURSOR 语句

您可以用 DECLARE CURSOR 语句声明使用动态 SQL 的游标。

7.3.1 语法

```
DECLARE name [(arg1,arg2,…)] CURSOR FOR | AS | IS
dynamic_sql_string | select_statement;
```

参数	类型	值	描述
dynamic_sql_string	VARCHAR	变量或表达式	用于定义游标的动态 SQL。
select_statement			用于定义游标的 SQL SELECT 语句。

注：

dynamic_sql_string 表达式是在游标打开时计算，而非声明时。

7.3.2 实例

- 使用动态 SQL 字符串。

```
DECLARE tabname VARCHAR DEFAULT 'db.orders';
DECLARE id INT;
DECLARE cur CURSOR FOR 'SELECT id FROM ' || tabname;
OPEN cur;
FETCH cur INTO id;
WHILE SQLCODE=0 THEN
  PRINT id;
  FETCH cur INTO id;
END WHILE;
CLOSE cur;
```

- 使用 SQL SELECT 语句。

```
DECLARE id INT;
DECLARE cur CURSOR FOR SELECT id FROM db.orders;
OPEN cur;
FETCH cur INTO id;
```

7. 游标

```
WHILE SQLCODE=0 THEN
  PRINT id;
  FETCH cur INTO id;
END WHILE;
CLOSE cur;
```

- 使用 CURSOR 参数。

```
DECLARE id INT;
DECLARE cur(cid) CURSOR FOR SELECT id FROM db.orders WHERE
customer_id=cid;
OPEN cur(100);
FETCH cur INTO id;
WHILE SQLCODE=0 THEN
  PRINT id;
  FETCH cur INTO id;
END WHILE;
CLOSE cur;
```

7.4 FETCH 语句

FETCH 语句从游标中检索下一行，并将列值赋给局部变量。

7.4.1 语法

```
FETCH [FROM] cursor_name INTO var1 [, var2, ...];
```

参数	类型	值	描述
cursor_name		标识符	之前打开的游标的名称。
varN		变量	局部变量。

7.4.2 实例

```
DECLARE tabname VARCHAR DEFAULT 'db.orders';
DECLARE id INT;
DECLARE cur CURSOR FOR 'SELECT id FROM ' || tabname;
OPEN cur;
FETCH cur INTO id;
WHILE SQLCODE=0 THEN
  PRINT id;
  FETCH cur INTO id;
END WHILE;
CLOSE cur;
```

7.5 OPEN 语句

OPEN 语句用来打开游标。

7.5.1 语法

```
OPEN cursor_name [FOR expression | select_statement];
OPEN cursor_name (var1, var2, …);
```

注:

- cursor_name: 如果未指定 FOR 子句，则是之前声明的游标的名称。
- FOR 表达式: 包含动态 SQL 的变量或表达式。
- FOR select_statement: SELECT 语句。

7.5.2 实例

- 打开之前声明的游标。

```
DECLARE tabname VARCHAR(20) DEFAULT 'db.orders';
DECLARE id INT;
DECLARE cur CURSOR FOR 'SELECT id FROM ' || tabname;
OPEN cur;
FETCH cur INTO id;
WHILE SQLCODE=0 THEN
  PRINT id;
  FETCH cur INTO id;
END WHILE;
CLOSE cur;
```

- 用动态 SQL 打开游标。

```
DECLARE tabname VARCHAR(20) DEFAULT 'db.orders';
DECLARE id INT;
OPEN cur FOR 'SELECT id FROM ' || tabname;
FETCH cur INTO id;
WHILE SQLCODE=0 THEN
  PRINT id;
  FETCH cur INTO id;
END WHILE;
CLOSE cur;
```

7. 游标

- 用参数打开游标。

```
DECLARE id INT;
DECLARE cur(cid) CURSOR FOR SELECT id FROM db.orders WHERE
customer_id=cid;
OPEN cur(100);
FETCH cur INTO id;
WHILE SQLCODE=0 THEN
  PRINT id;
  FETCH cur INTO id;
END WHILE;
CLOSE cur;
```

8. 动态 SQL

8.1 EXEC/ EXECUTE/ EXECUTE IMMEDIATE

EXECUTE （EXEC 或 EXECUTE IMMEDIATE）语句执行动态 SQL 语句，并将标量结果返回到局部变量。

还可使用此语句调用存储过程。

8.1.1 语法

```
EXEC | EXECUTE | EXECUTE IMMEDIATE dynamic_sql_string [INTO
var1, var2, ...];
|
EXEC | EXECUTE proc_name [parm1 = val1, ... ]
```

参数	类型	值	描述
dynamic_sql_string	VARCHAR	变量或表达式	要执行的动态 SQL。
INTO var1, var2, …	任何类型	变量	要赋值的变量，可选。

注：

- 如果查询返回超过 1 行数据，且指定了 INTO 子句，则在赋值时只使用第一行中的列。
- 如果语句返回 1 行或多行数据，且未指定 INTO 子句，则将结果集数据发送到标准输出。
- EXEC 和 EXECUTE 是 EXECUTE IMMEDIATE 的同义词。

8.1.2 实例

- 将结果返回到变量。

```
DECLARE cnt INT;
EXECUTE 'SELECT COUNT(*) FROM db.orders' INTO cnt;
```

- 执行 DML 语句。

8. 动态 SQL

```
DECLARE tabname VARCHAR(100) DEFAULT 'tab1';
EXECUTE IMMEDIATE 'CREATE TABLE ' || tabname || ' (c1 INT)';
```

- 将结果打印到标准输出。

```
EXEC 'SELECT ''A'', ''B'' FROM dual';
```

9. 预编译语句

9.1 PREPARE

PREPARE 语句创建可多次执行的预编译 SQL 语句。

您还可在执行预编译语句时提供参数。

9.1.1 语法

```
PREPARE statement_name FROM dynamic_sql_string;
```

参数	类型	值	描述
dynamic_sql_string	VARCHAR	变量或表达式	要执行的动态 SQL。

注：

- 如果 PREPARE 语句失败，则对指定语句运行 EXECUTE 的任何后续尝试都将失败。
- 如果 SQL 字符串中有“?”符号，当用 EXECUTE 语句执行预编译语句时，需要提供相同数量的参数。

9.1.2 实例

- 将结果返回到变量。

```
PREPARE stmt1 FROM 'SELECT SALARY FROM EMPLOYEE WHERE jobcode=?';
EXECUTE stmt1 USING 100;
```

9.2 EXECUTE

EXECUTE 语句执行预编译的 SQL 语句。

您还可以在执行预编译语句时提供参数。

9.2.1 语法

```
EXECUTE statement_name [INTO var1,var2,…] [USING v1, v2,…];
```

注:

- EXECUTE 语句的语法与 EXECUT 动态 SQL 语句的语法非常相似，区别在于<statement_name>是一个 ID，不是一个变量，并且 EXECUTE 预编译语句支持 USING 子句。
- 参数数量需要与预编译语句中使用的参数数量相同。

9.2.2 实例

- 将结果返回到变量。

```
PREPARE stmt1 FROM 'SELECT SALARY FROM EMPLOYEE WHERE
jobcode=?';
EXECUTE stmt1 USING 100;
```

9.3 CLOSE PREPARE

CLOSE PREPARE 语句关闭预编译的 SQL 语句。

预编译语句关闭后就无法执行了。

9.3.1 语法

```
CLOSE PREPARE statement_name;
```

9.3.2 实例

- 将结果返回到变量。

```
PREPARE stmt1 FROM 'SELECT SALARY FROM EMPLOYEE WHERE
jobcode=?';
EXECUTE stmt1 USING 100;
CLOSE PREPARE stmt1;
```

10. 异常块

10.1 异常处理

QianBase PL/SQL 可处理程序中的异常。

10.1.1 语法

```
BEGIN
  -- Statements that can raise an exception
EXCEPTION
  WHEN condition THEN
    -- Statements
  WHEN condition2 THEN
    -- Statements2
    ...
END
```

注：目前只支持 OTHERS 条件。

10.1.2 实例

```
DECLARE
  v VARCHAR(200);
BEGIN
  OPEN cur FOR 'SELECT c1 FROM t1';
  FETCH cur INTO v;
  CLOSE cur;
EXCEPTION WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE('Error');
END
```

11. 条件处理

11.1 DECLARE CONDITION 语句

您可使用 DECLARE CONDITION 语句来声明用户定义的条件。

然后用 DECLARE HANDLER 为此条件定义一个处理程序，并使用 SIGNAL 语句引发条件。

11.1.1 语法

```
DECLARE condition_name CONDITION;
```

11.1.2 实例

如果行数与指定的行数不等，则引发条件：

```
DECLARE cnt INT DEFAULT 0;
DECLARE wrong_cnt_condition CONDITION;
DECLARE EXIT HANDLER FOR wrong_cnt_condition
  PRINT 'Wrong number of rows';
SELECT COUNT(*) INTO cnt FROM TABLE (VALUES (1,2));
IF cnt <> 1 THEN
  SIGNAL wrong_cnt_condition;
END IF;
```

11.2 DECLARE HANDLER 语句

使用 DECLARE HANDLER 语句定义一个或多个 QianBase PL/SQL 语句，以便在条件出现时执行。

11.2.1 语法

```
DECLARE [CONTINUE | EXIT] HANDLER FOR
  [SQLEXCEPTION | NOT FOUND | user_condition] code_block;
```

注:

- CONTINUE: 当处理程序完成时，将返回到紧跟在引发异常条件的 QianBase PL/SQL 语句之后。
- EXIT: 处理程序完成后，控件将返回到声明处理程序块的末尾。
- code_block: 在指定条件出现时执行的 QianBase PL/SQL 语句。

11.2.2 实例

```
DECLARE name VARCHAR(100);
DECLARE no_rows INT DEFAULT 0;
DECLARE CONTINUE HANDLER FOR NOT FOUND
  SET no_rows = 1;
OPEN cur FOR 'SELECT name FROM db.orders';
FETCH cur INTO name;
WHILE no_rows = 0 THEN
  PRINT id;
  FETCH cur INTO name;
END WHILE;
CLOSE cur;
```

11.3 GET DIAGNOSTICS 语句

GET DIAGNOSTICS 语句可以用来检索错误消息和上一个 SQL 语句的行数。

11.3.1 语法

获取错误文本:

```
GET DIAGNOSTICS EXCEPTION 1 var_name = MESSAGE_TEXT;
```

获取与上一个 SQL 语句相关联的行的行数:

```
GET DIAGNOSTICS var_name = ROW_COUNT;
```

11.3.2 实例

如果行数与指定的行数不等，则引发条件:

```
BEGIN;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    GET DIAGNOSTICS EXCEPTION 1 text = MESSAGE_TEXT;
    PRINT 'Text: ' || text;
  END;
  SELECT * FROM table_not_exists;
  END;
END;
```

11.4 RESIGNAL 语句

RESIGNAL 语句用于在条件或异常处理程序中重新触发异常，以便在更高的级别上进行处理。

11.4.1 语法

```
RESIGNAL;
RESIGNAL SQLSTATE [VALUE] sqlstate [SET MESSAGE_TEXT =
message_text];
```

11.4.2 实例

11.4.2.1 实例 1

重新触发相同异常:

```
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  BEGIN
    PRINT 'Error raised';
    RESIGNAL;
  END;
  PRINT 'Before executing SQL';
  SELECT * FROM table_not_exists;
  PRINT 'After executing SQL';
END;
```

结果:

```
Before executing SQL
Error raised
```

11.4.2.2 实例 2

在外部条件处理程序中获取重新触发的条件:

```
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
    PRINT 'Error raised, outer handler';
  BEGIN
```

```
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN
      PRINT 'Error raised, resignal';
      RESIGNAL;
    END;
    PRINT 'Before executing SQL';
    SELECT * FROM table_not_exists;
    PRINT 'After executing SQL - must not be printed';
  END;
  PRINT 'Continue outer block after exiting inner';
END;
```

结果:

```
Before executing SQL
Error raised, resignal
Error raised, outer handler
Continue outer block after exiting inner
```

11.4.2.3 实例 3

使用指定的 SQLSTATE 和消息文本重新触发条件:

```
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    GET DIAGNOSTICS EXCEPTION 1 text = MESSAGE_TEXT;
    PRINT 'SQLSTATE: ' || SQLSTATE;
    PRINT 'Text: ' || text;
  END;
  BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
      RESIGNAL SQLSTATE '02031' SET MESSAGE_TEXT = 'Some
error';
    SELECT * FROM table_not_exists;
  END;
END;
```

结果:

```
SQLSTATE: 02031
Text: Some error
```

11.5 SIGNAL 语句

SIGNAL 语句引发用户定义的条件（异常）。

11.5.1 语法

```
SIGNAL condition_name;
```

11.5.2 实例

如果行数与指定的行数不等，则引发条件：

```
DECLARE cnt INT DEFAULT 0;
DECLARE wrong_cnt_condition CONDITION;
DECLARE EXIT HANDLER FOR wrong_cnt_condition
  PRINT 'Wrong number of rows';
SELECT COUNT(*) INTO cnt FROM TABLE (VALUES (1,2));
IF cnt <> 1 THEN
  SIGNAL wrong_cnt_condition;
END IF;
```

12. 锚定类型

12.1 %ROWTYPE 属性

%ROWTYPE 属性声明一个记录变量，该变量与指定的数据库表有相同的列和数据类型。

12.1.1 语法

```
var_name [schema.]table_name%ROWTYPE
```

12.1.2 实例

```
DECLARE
  v orders%ROWTYPE;
BEGIN
  SELECT * INTO v FROM orders LIMIT 1;
  DBMS_OUTPUT.PUT_LINE('Item:  '  ||  v.name  ||  '  -  '  ||
v.description);
END;
```

```
DECLARE
  v orders%ROWTYPE;
  CURSOR c IS SELECT * FROM orders;
BEGIN
  OPEN c1;
  FETCH c1 INTO v1;
  DBMS_OUTPUT.PUT_LINE('Item:  '  ||  v.name  ||  '  -  '  ||
v.description);
  CLOSE c1;
END;
```

```
BEGIN
  FOR v IN (SELECT * FROM orders)
  LOOP
    DBMS_OUTPUT.PUT_LINE('Item:  '  ||  v.name  ||  '  -  '  ||
v.description);
```

12. 锚定类型

```
  END LOOP;
END;
```

```
DECLARE
  v orders%ROWTYPE;
BEGIN
  EXECUTE IMMEDIATE 'SELECT * FROM orders LIMIT 1' INTO v;
  DBMS_OUTPUT.PUT_LINE('Item: ' || v.name || ' - ' ||
v.description);
END;
```

12.2 %TYPE 属性

%TYPE 属性用来声明一个变量，该变量与指定的引用列有相同的数据类型。

12.2.1 语法

```
var_name [schema.]table.column_name%TYPE
```

注：如果找不到 table.column_name，则数据类型从第一个赋值表达式获取。

12.2.2 实例

```
DECLARE
  i orders.item%TYPE;
BEGIN
  SELECT item INTO i FROM orders LIMIT 1;
  DBMS_OUTPUT.PUT_LINE('Item: ' || i);
END;
```

13. 运算符

13.1 加法

您可以使用（+）算术运算符将两数相加，连接两个字符串，或者在 Date 类型的值上增加一个区间值。

13.1.1 语法

```
expr + expr
```

13.1.2 实例

两个整数相加:

```
3 + 1 -- result 4
DECLARE a INT := 3;
DECLARE b INT := 1;
DECLARE c INT;
SET c = a + b; -- variable c will be 4
```

连接两个字符串:

```
'ABC' + 'abc'  -- result 'ABCabc'
DECLARE a VARCHAR(10) := 'ABC';
DECLARE b VARCHAR(10) := 'abc';
DECLARE c VARCHAR(10);
SET c = a + b; -- result of c is 'ABCabc'
```

在 Date 类型上增加添加一个整数:

```
DATE '2014-12-31' + 1 -- result DATE '2015-01-01'
DECLARE a DATE := DATE '2014-12-31';
DECLARE b DATE;
SET b = a + 1; -- result of b is DATE '2015-01-01'
```

13.2 减法

您可以使用（-）算术运算符将两数相减，或者从 Date 值中减去天数。

13.2.1 语法

```
expr - expr
```

13.2.2 实例

两个整数相减:

```
3 - 1 -- result 2
DECLARE a INT := 3;
DECLARE b INT := 1;
DECLARE c INT;
SET c = a - b; -- variable c will be 2
```

从 Date 类型中减去一个整数:

```
DATE '2014-12-31' - 1 -- result DATE '2014-12-30'
DECLARE a DATE := DATE '2014-12-31';
DECLARE b DATE;
SET b = a - 1; -- result of b is DATE '2014-12-30'
```

13.3 乘法

您可以使用（*）算术运算符将两数相乘。

13.3.1 语法

```
expr * expr
```

13.3.2 实例

两数相乘:

```
3 * 3 -- result 9
DECLARE a INT := 3;
DECLARE b INT := 3;
DECLARE c INT;
SET c = a * b; -- variable c will be 9
```

13.4 除法

您可以使用（/）算术运算符将两数相除。

13.4.1 语法

```
expr / expr
```

13.4.2 实例

两数相除:

```
9 / 3 -- result 3
DECLARE a INT := 9;
DECLARE b INT := 3;
DECLARE c INT;
SET c = a / b; -- variable c will be 3
```

13.5 模

您可以使用（%）算术运算符计算两数的模。

13.5.1 语法

```
expr % expr
```

13.5.2 实例

求两数的模:

```
10 % 2 -- result 1
11 % 2 -- result 0
```

13.6 比较

您可以使用等号和比较运算符（=、==、<>、!=、<、>、>=、<=）来确定一个运算对象是否等于、不等于、小于或大于另一个运算对象。

13.6.1 语法

```
expr == expr | expr = expr
expr != expr | expr <> expr
expr > expr
expr < expr
expr >= expr
expr <= expr
```

13.6.2 实例

```
3 == 1 -- false
3 <> 1 -- true
3  > 1 -- true
3  < 1 -- false
'abcde' != 'abc' -- true
'CA' = false -- NULL
DATE '2014-12-31' > DATE '2014-12-30' -- true
```

13.7 [NOT] IN

您可以使用 [NOT] IN 运算符来检查值列表中是否存在某个值。

13.7.1 语法

```
expr [NOT] IN (expr*)
```

注：

对于布尔值列表，只支持布尔文本（true、false），在值列表中使用布尔表达式会导致语法错误。

13.7.2 实例

```
3 IN (1,2,3,4,5) -- true
3 IN (1,2) -- false
3 IN (1,2,'3') - false
'3' NOT IN (1,2,3,4,5) -- true
'abc' NOT IN ('ABC', 'Abc', 'BC') -- true
```

14. 触发器

14.1 创建或修改触发器

触发器在数据库里以独立的对象存储，它与存储过程和函数不同的是，存储过程与函数需要用户显示调用才执行，而触发器是由一个事件来启动运行。即触发器是当某个事件发生时自动地隐式运行。并且，触发器不能接收参数。

14.1.1 语法

```
Create [or replace] trigger [[catalog.]schema.]trigger_name
before|after            {insert|update|delete            [or
{insert|update|delete} [or {insert|update|delete}]]}
on [[catalog.]schema.]table_name
for each row |statememt
as
begin
  …
end;
/
```

14.1.2 实例

```
create table test_pk (c1 int primary key,c2 int);
create table test_log(c1 timestamp,c2 varchar(30));
create table test_py_log (c1 int, c2 int);

create or replace trigger test_trig
before insert or update or delete
on test_pk
for each row
as
begin
if inserting then
  insert into test_log values (systimestamp,'insert');
elsif updating then
  insert into test_log values (systimestamp,'update');
  insert into test_py_log values (new.c1, new.c2);
```

```
  insert into test_py_log values (old.c1, old.c2);
elsif deleting then
  insert into test_log values (systimestamp,'delete');
else
  null;
end if;
end;
/
```

14.2 删除触发器

14.2.1 语法

```
drop trigger trigger_name;
```

14.2.2 实例

```
drop trigger test_trig;
```

14.3 显示当前触发器

14.3.1 语法

```
showddl trigger trigger_name;
```

14.3.2 实例

```
showddl trigger test_trig;
```

14.4 获取 schema 下所有触发器

14.4.1 语法

```
get triggers;
```

14.5 关键字解释

14.5.1 action_sql

action_sql 这个关键字是获取当前执行的 sql 语句
需要注意的是:

- 获取的最长的值是 8192，所以当 sql 语句超过该值，就会自动截取
- 在定义表的列的时候，不要使用该字段，否则在触发器中会被替换引发错误
- 在 prepare 的语句中，在查询的时候，目前是没有支持把参数换为真正的值

14.5.2 新旧值替换的关键字

NEW. 替换新值的标识
OLD. 替换旧值的标识
这两个字段在使用的时候，用法如下:
new.colname
old.colname
在执行相关的 sql 语句之前，我们会将这些值替换为 SQL 语句中的真正的值。
需要注意的是:

- New 值只有在 delete 和 update 的时候才存在
- Old 值只有在 delete 和 update 的时候才存在

14.5.3 条件关键字

INSERTING ： 作为判断条件，只有当前执行的 sql 语句是 insert 的时候，才会满足条件
DELETING ： 作为判断条件，只有当前执行的 sql 语句是 delete 的时候，才会满足条件
UPDATING： 作为判断条件，只有当前执行的 sql 语句是 update 的时候，才会满足条件

14.6 相关的 CQD

对应的 CQD 是：USE_TRIGGER
默认是打开的状态，如果不想使用触发器，可以在 defaults 表将该值设为 OFF 即可

14.7 注意事项

- 目前对数据类型的支持方面，我们暂时不支持 lob 类型和 nchar，nvarchar 类型。目前对 lob 类型，如果在触发器使用该类型的值，我们会报错，但是 nchar 和 nvarchar 目前还没有进行处理。
- 目前没有做循环检查，因此在使用的时候需要注意。如果出现下列情况，则会出现死循环，触发器调用一直不会结束。

例如：

```
Create table t1(c1 int);
Create table t2(c1 int);
Create or replace trigger t1_trig
after insert on t1
for each row
as
begin
insert into t2 values(1);
end;
/

create or replace trigger t2_trig
after insert on t2
for each row
as
begin
insert into t1 values(1);
end;
/
```

- 每列的长度不能超过 12M（由于我们使用了 base64 加密传输，因此数据可能会比实际数据长 1/3），否则在解析的时候会出错。
- 目前触发器的名字不能使用双引号
- 语句级触发器是不支持 new 和 old 值的
- 在触发器中，最好不要使用类似 count(*)的操作，主要是由于多个会话在对同一张表执行该操作的时候，可能会出错。
- 尽量不要再多个会话中，一个一直执行授权，另一个在执行触发器。

15. 内置函数

15.1 CAST 函数

CAST 函数将表达式转换为指定的数据类型。

15.1.1 语法

```
CAST(expression AS datatype[(length)]);
```

注:

如果 CAST 指定的 length 是用于转换 CHAR 或 VARCHAR，则结果字符串将被截取为这个长度。

15.1.2 实例

15.1.2.1 实例 1

转换为指定长度的字符串:

```
CAST('Abc' AS CHAR(1));
--
A
```

15.1.2.2 实例 2

截取时间戳字符串。

```
CAST(TIMESTAMP '2015-03-12 10:58:34.111' AS CHAR(10));
--
2015-03-12
```

15.2 CHAR 函数

CHAR 函数将数字转换为字符串。

15.2.1 语法

```
CHAR(num_expression);
```

返回类型: STRING

15.2.2 实例

```
CHAR(1000);
--
1000
```

15.3 COALESCE 函数

COALESCE 函数返回第一个非空表达式。.

15.3.1 语法

```
COALESCE(expr1, expr2 [, expr3, ...]);
```

参数	类型	值
exprN	任何类型	变量或表达式

注:

- 当发现第一个非空表达式时，不计算后续表达式。
- COALESCE 和 NVL 函数是同义词。

返回值:

- 第一个非空表达式。
- NULL，如果所有表达式的计算结果都为 NULL。

返回类型: 第一个非空表达式的数据类型。

15.3.2 实例

```
COALESCE(NULL, 1, 2, 3);
```

结果: 1

15.4 CONCAT 函数

CONCAT 函数连接两个或多个字符串。

15.4.1 语法

```
CONCAT(expr, expr2 [, expr3, ...]);
```

注:

- 如果某个表达式的计算结果为 NULL，则将其视为空字符串。
- 只有当所有表达式的计算结果都为 NULL 时，CONCAT 才返回 NULL。

返回类型: STRING

15.4.2 实例

```
CONCAT('a', 'b', NULL, 'c');
```

结果：abc

15.5 CURRENT_DATE

CURRENT_DATE 函数返回当前日期（年、月和日）。

15.5.1 语法

```
CURRENT_DATE | CURRENT DATE
```

返回类型: DATE

15.6 CURRENT_TIMESTAMP 函数

CURRENT_TIMESTAMP 函数返回当前的日期与时间（年、月、日、时、分、秒和小数秒）。

15.6.1 语法

```
CURRENT_TIMESTAMP | CURRENT TIMESTAMP [(precision)]
```

参数	值	说明
precision	变量或表达式	小数秒的精度范围从 0 到 3。默认值为 3

返回类型: TIMESTAMP

15.6.2 实例

获取当前日期和时间，不带小数秒。

```
CURRENT_TIMESTAMP(0)
--
2015-03-02 13:04:42
```

15.7 CURRENT_TIME 函数

CURRENT_TIME 函数返回当前的时间（时、分、秒和小数秒）。

15.7.1 语法

```
CURRENT_TIME | CURRENT TIME [(precision)]
```

参数	值	描述
precision	变量或表达式	小数秒的精度范围从 0 到 3。默认值为 3

返回类型： TIME

15.7.2 实例

获取当前的时间，不带小数秒。

```
CURRENT_TIME(0)
--
13:04:42
```

15.8 CURRENT_USER 函数

CURRENT_USER 函数返回执行当前 EsgnDB PL/SQL 脚本的用户名称。

15.8.1 语法

```
CURRENT_USER | CURRENT USER
```

返回类型： STRING

15.8.2 实例

获取当前用户名称:

```
CURRENT_USER
--
paul
```

15.9 DATE 函数

DATE 函数将表达式转换为 DATE 数据类型。

15.9.1 语法

```
DATE(expression);
```

返回数据类型：DATE

15.9.2 实例

将字符串和时间戳转换为 DATE：

```
DATE('2015-03-12');
DATE('2015' || '-03-' || '12');
DATE(TIMESTAMP '2015-03-12 10:58:34.111');
```

15.10 DECODE 函数

DECODE 函数实现 IF-THEN-ELSE 逻辑。

15.10.1 语法

```
DECODE(expr, when_exp1, then_expr1 [, ...n] [, else_expr])
```

注：

- 如果 expr 是 NULL，它将与第一个为 NULL 的 when_exprN 匹配。如果 when_exprN 不匹配，则不计算 then_exprN。

15.10.2 实例

```
DECLARE var1 INT DEFAULT 3;
PRINT DECODE (var1, 1, 'A', 2, 'B', 3, 'C');    -- Result: C
PRINT DECODE (var1, 1, 'A', 2, 'B', 'C');       -- Result: C
SET var1 = NULL;
PRINT DECODE (var1, 1, 'A', 2, 'B', NULL, 'C'); -- Result: C
```

15.11 EXTRACT 函数

EXTRACT 函数能从 Date 类型中获取日期控件（月、日、小时等）。

15.11.1 语法

```
EXTRACT (date_field FROM expr)
date_field :
       YEAR
     | MONTH
     | DAY
     | HOUR
     | MINUTE
     | SECOND
```

注：

- expr 的类型必须是 DATE 或 TIMESTAMP 类型。

15.11.2 实例

```
EXTRACT (YEAR FROM DATE '2019-06-29') – Result: 2019
EXTRACT (HOUR FROM TIMESTAMP '2019-06-29 10:20:30.123456' –
Result: 10
```

15.12 FROM_UNIXTIME 函数

FROM_UNIXTIME 函数将指定的自 1970-01-01 00:00:00 以来的秒数转换为时间戳值。

15.12.1 语法

```
FROM_UNIXTIME(epoch, [format])
```

注：

- epoch 是自 1970-01-01 00:00:00 以来的秒数。
- format 是时间戳的格式，是可选择的。默认格式为 yyyy-MM-dd HH:mm:ss。

返回类型：STRING

15.12.2 实例

将秒数转换为时间戳值。

```
from_unixtime(1447141681);
---
2015-11-10 04:48:01

from_unixtime(1447141681, 'yyyy-MM-dd');
---
2015-11-10
```

15.13 INSTR 函数

INSTR 函数返回字符串中子字符串的起始位置。

15.13.1 语法

```
INSTR(string, substring [, position [, occurrence]])
```

注：

- position 指定搜索的起始位置，默认值为 1（字符串的开头）
- 如果 position 为负， INSTR 从字符串结尾开始反向搜索。
- occurrence 指定要搜索的子字符串的出现位置，默认值为 1（第一次出现位置）
- 如果 string 是 NULL ，则返回值为 NULL。
- 如果 string 不是 NULL 而且没有找到子字符串，则返回值是 0。

返回类型： INTEGER

15.13.2 实例

返回字符串中子字符串的起始位置。

```
INSTR('abc', 'b')                       -- Result 2
INSTR('abcabc', 'b', 3)                 -- Result 5
INSTR('abcabcabc', 'b', 3, 2)           -- Result 8
INSTR('abcabcabc', 'b', -3)             -- Result 5
INSTR('abcabcabc', 'b', -3, 2)          -- Result 2
```

15.14 LEN/LENGTH 函数

LEN/LENGTH 函数返回字符串的长度。

LEN 函数返回删除了有尾随空格的字符串的长度。

LENGTH 函数返回有尾随空格的字符串的长度。

15.14.1 语法

```
LEN(string_expression);
LENGTH(string_expression);
```

返回类型： INTEGER

15.14.2 实例

返回字符串的长度。

```
LEN('abc  ')                          -- Result 3
LENGTH('abc  ')                       -- Result 5
```

15.15 LOWER 函数

LOWER 函数将字符串表达式转换为小写。

15.15.1 语法

```
LOWER(expression);
```

返回数据类型： STRING

15.15.2 实例

将字符串转换为小写。

```
LOWER('ABC');
---
abc
```

15.16 MOD 函数

MOD 函数用于求模。

15.16.1 语法

```
MOD (expr, expr)
```

返回数据类型: INTEGER

15.16.2 实例

```
MOD (10, 2) – Result: 0
MOD (11, 2) – Result: 1
```

15.17 NOW 函数

NOW 函数返回当前的日期和时间（年、月、日、时、分、秒和小数秒）。

15.17.1 语法

```
NOW()
```

返回类型： TIMESTAMP

15.17.2 实例

获取当前的日期和时间。

```
NOW()
--
2015-11-02 07:59:25.833
```

15.18 NVL 函数

NVL 函数返回第一个非空表达式。

15.18.1 语法

```
NVL(expr1, expr2 [, expr3, ...]);
```

参数	类型	值
exprN	任何类型	变量或表达式

注：

- 当发现第一个非空表达式时，不计算后续表达式。
- NVL 和 COALESCE 函数是同义词。

返回值：

- 第一个非空表达式。
- NULL，如果所有表达式的计算结果都为 NULL。

返回类型：第一个非空表达式的数据类型。

15.18.2 实例

```
NVL(NULL, 1);
```

结果：1

15.19 NVL2 函数

如果第一个表达式是 NOT NULL，NVL2 函数返回第二个表达式的结果，否则，返回第三个表达式的结果。

15.19.1 语法

```
NVL2(expr1, expr2, expr3);
```

参数	类型	值
exprN	任何类型	变量或表达式

注：

如果expr1 不是 NULL, 则只计算expr2; 如果expr1 是NULL, 则只计算expr3。

返回类型：

expr2 或 expr3 返回表达式的数据类型取决于 expr1 是否为 NULL。

15.19.2 实例

```
NVL2(NULL, 1, 2);
```

结果：2

15.20 SUBSTR 函数

SUBSTR 函数从字符串中返回子字符串。

15.20.1 语法

```
SUBSTR(string, start_pos [, substring_len])
```

参数	类型	值	说明
string	字符串	变量或表达式	原始字符串
start_pos	整数	变量或表达式	子字符串的起始位置(从 1 开始)
substring_len	整数	变量或表达式	子字符串的长度

注:

- 如果 start_pos 是 0，则将其视为 1。
- 如果 start_pos 为负，则视为从末尾向前计数。
- SUBSTR 和 SUBSTRING 函数是同义词。

返回类型: String.

15.20.2 实例

```
SUBSTR('Remark', 3);
```

结果: 'mark'

```
SUBSTR('Remark', 3, 3);
```

结果: 'mar'

15.21 SYSDATE 函数

SYSDATE 函数返回当前的日期和时间（年、月、日、天、时、分和秒）。

15.21.1 语法

```
SYSDATE
```

返回类型： TIMESTAMP

15.21.2 实例

获取当前日期和时间。

```
SYSDATE
--
2015-03-03 11:06:31
```

15.22 TO_CHAR 函数

TO_CHAR 函数将表达式转换为字符串。

15.22.1 语法

```
TO_CHAR(expression);
```

返回类型： STRING

15.22.2 实例

```
TO_CHAR(CURRENT_DATE);
```

15.23 TO_TIMESTAMP 函数

TO_TIMESTAMP 函数使用指定的格式将字符串转换为 TIMESTAMP 数据类型。

15.23.1 语法

```
TO_TIMESTAMP(string_expression, format_expression);
```

返回数据类型： TIMESTAMP

格式元素：

YYYY	四位数年份
MM	月份 （1-12）
DD	日期（1-31）
HH24	一天中的某一小时（0-23）
MI	分 （0-59）
SS	秒 （0-59）

15.23.2 实例

```
TO_TIMESTAMP('2015-04-02', 'YYYY-MM-DD');
TO_TIMESTAMP('04/02/2015', 'mm/dd/yyyy');
TO_TIMESTAMP('2015-04-02       13:51:31',       'YYYY-MM-DD
HH24:MI:SS');
```

15.24 TRIM 函数

TRIM 函数从字符串中删除前导和尾随字符。

15.24.1 语法

```
TRIM(string_expression);
```

返回类型： STRING

15.24.2 实例

```
'#' || TRIM(' Hello ') || '#';
--
#Hello#
```

15.25 UPPER 函数

UPPER 函数将字符串表达式转换为大写。

15.25.1 语法

```
UPPER(expression);
```

返回数据类型： STRING

15.25.2 实例

将字符串转换为大写。

```
UPPER('abc');
---
ABC
```

16. 用户自定义 SQL 函数

16.1 CREATE FUNCTION

CREATE FUNCTION 语句创建用户自定义的 SQL 函数。

16.1.1 语法

```
CREATE [OR REPLACE] FUNCTION function_name ( [parameters] )
 RETURNS | RETURN data_type
 [AS | IS]
 body
parameters:
  [IN] name data_type, ...
  |
  name [IN] data_type, ...
body:
  statement | expression | BEGIN statements END);
```

16.1.2 实例

16.1.2.1 实例 1

创建一个没有参数的函数:

```
CREATE FUNCTION hello()
 varchar(20)
BEGIN
 RETURN 'Hello, world';
END;

-- Call the function
SELECT hello() FROM DUAL;
```

16.1.2.2 实例 2

创建一个带有参数的函数:

16. 用户自定义 SQL 函数

```
CREATE FUNCTION hello2(text STRING)
 varchar(20)
BEGIN
 RETURN 'Hello, ' || text || '!';
END;
-- Call the function
SELECT hello2('world') FROM DUAL;
```

16.2 DROP 函数

DROP FUNCTION 语句删除用户自定义的 SQL 函数。

16.2.1 语法

```
DROP FUNCTION [IF EXISTS] function_name;
```

16.2.2 实例

```
DROP FUNCTION func1;
DROP FUNCTION IF EXISTS func2;
```

17. 用户自定义 SQL 包

17.1 CREATE PACKAGE

CREATE PACKAGE 语句定义相关变量、过程和函数的集合。

包由包说明和包主体组成。包说明定义了包的变量、存储过程和函数接口。包主体定义了过程和功能的实现。

对于已存在的包使用 OR REPLACE 子句时，包首先会被删除，然后使用新的包说明或包主体重新创建它。当更换包说明时，包主体也将被删除。当只更换包主体时，包说明将被完整保存。

17.1.1 语法

包说明:

```
CREATE [OR REPLACE] PACKAGE package_name AS | IS package_spec
END
package_spec:
  variable declaration |
  function declaration |
  procedure declaration;
```

包主体:

```
CREATE [OR REPLACE] PACKAGE BODY package_name AS | IS
package_body END
package_body:
  private variable declaration |
  function definition |
  procedure definition;
```

注:

- 没有强制要求包说明和包主体的一致性。
- 忽略包说明中定义的变量。

17.1.2 实例

包说明:

```
CREATE OR REPLACE PACKAGE users AS
BEGIN
  session_count int := 0;
  function get_count() return int;
  procedure add(name varchar(100));
END;
```

包主体:

```
CREATE OR REPLACE PACKAGE BODY users AS
BEGIN
 function get_count() return int
  is
  begin
    return session_count;
  end;
  procedure f_add(name varchar(100))
  is
  begin
    -- ...
    session_count = session_count + 1;
  end;
END;
```

使用包:

```
CALL users.add('John');
CALL users.add('Sarah');
CALL users.add('Paul');
SELECT 'Number of users: ' || users.get_count() FROM DUAL;
```

17.2 DROP PACKAGE

DROP PACKAGE 语句删除用户自定义的 SQL 包。

DROP PACKAGE 语句删除包说明和包主体。

17.2.1 语法

包说明:

```
DROP PACKAGE [IF EXISTS] package_name;
```

18. 内置变量

18.1 ACTIVITY_COUNT

ACTIVITY_COUNT 内置变量包含受上一个 SQL 语句影响的行数。

18.1.1 语法

```
ACTIVITY_COUNT
```

18.1.2 实例

```
DECLARE var INT;
SELECT id INTO var FROM sometable;
IF ACTIVITY_COUNT = 1 THEN
  PRINT 'id = ' || var;
END IF;
```

18.2 ERRORCODE

ERRORCODE 内置变量包含上一个 SQL 语句的返回码。

返回码可以是零（0）、负数或正数。

返回码为 0 表示成功，正数表示警告，负数表示错误。

18.2.1 语法

```
ERRORCODE
```

18.2.2 实例

```
IF ERRORCODE <> 0 THEN
  DBMS_OUTOUT.PUT_LINE('SUCCESS');
ELSE
  DBMS_OUTOUT.PUT_LINE('FAILURE');
END IF;
```

18.3 SQLCODE

SQLCODE 内置变量包含上一个 SQL 语句的返回码。

返回码可以是零（0）、负数或正数。

返回码为 0 表示成功，正数表示警告，负数表示错误。

18.3.1 语法

```
SQLCODE
```

注：

- SQLCODE 100 表示未找到行或游标结束。

18.3.2 实例

```
DECLARE id INT;
DECLARE cur CURSOR FOR 'SELECT id FROM db.orders';
OPEN cur;
FETCH cur INTO id;
WHILE SQLCODE = 0 THEN
  FETCH cur INTO id;
END WHILE;
CLOSE cur;
```

18.4 SQLSTATE

SQLSTATE 内置变量包含最后一条 SQL 语句返回的 5 个字符的状态。

SQLSTATE 状态代码由两个字符的类代码和三个字符的子类代码组成。类代码 00（例如：SQLSTATE '00000'）表示成功完成。

18.4.1 语法

```
SQLSTATE
```

18.4.2 实例

```
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
     PRINT 'SQLSTATE: ' || SQLSTATE;
  END;
  SELECT * FROM table_not_exists;
END;
```

19. 配置

19.1 配置文件

SP/SQL 的配置文件位于$TRAF_CONF/hplsql-site.xml。

此文件中的所有配置选项均以“hplsq.”前缀开头。

19.1.1 语法

配置的文件是 XML 文件:

```
<configuration>
<property>
  <name>NAME</name>
  <value>VALUE</value>
  <description>An optional description</description>
</property>
…
</configuration>
```

您还可以使用 SET 语句动态设置选项:

```
SET option=value;
```

19.1.2 选项

19.1.2.1 hplsql.conn.<connection_profile>

hplsql.conn.<connection_profile>选项定义了连接配置文件<connection_profile>。
value 标记包含关于 JDBC 驱动程序、连接字符串、用户名和密码的信息，这些信息以分号（；）分隔。

```
<property>
  <property>
  <name>hplsql.conn.trafodion</name>
<value>org.trafodion.jdbc.t4.T4Driver;jdbc:t4jdbc://127.0.0
.1:23400/;catalog=TRAFODION;trafodion;traf123</value>
  <description>Apache        Trafodion        JDBC        T4
connection</description>
```

```
</property>
```

19.1.2.2 hplsql.conn.default

hplsql.conn.default 选项指定默认连接配置文件。默认值是“default”，它将使用 Trafodion JDBC T2 连接。

例如:

```
<property>
  <name>hplsql.conn.default</name>
  <value>trafodion</value>
  <description>The default connection profile</description>
</property>
```

注:

- 必须定义默认连接的连接配置文件。

19.1.2.3 hplsql.conn.init.default

hplsql.conn.init.default 选项定义与默认配置文件建立连接后要执行的 SQL 语句。

例如:

```
<property>
  <name>hplsql.conn.init.default</name>
  <value>
    CQD DEFAULT_CHARSET 'UTF8';
CQD TRAF_DEFAULT_COL_CHARSET 'UTF8';
SET SCHEMA MYSCHEMA;
  </value>
</property>
```

19.1.2.4 hplsql.use.only.one.connection

默认情况下，在 SP/SQL 中运行的 SQL 语句可能会使用不同的 JDBC 连接，此选项将确保所有 SQL 语句都使用相同的 JDBC 连接。

例如:

19. 配置

```
<property>
  <name>hplsql.use.only.one.connection</name>
  <value>true</value>
</property>
```

注：

- 使用此选项时，您可能会遇到一个 Trafodion JDBC 连接的最大语句限制。您可以通过关闭未使用的游标、删除不必要的独立 SELECT 语句或返回的 RESULT SETs 来缓解此问题。
- 这个选项对 hplsql.transaction.compatible 是必需的。

19.1.2.5 hplsql.transaction.compatible

当启用此选项，并使用 NO TRANSACTION REQUIRED 属性定义流程时，事务将以与其他一些数据库系统兼容的方式运行。

例如:

```
<property>
  <name>hplsql.transaction.compatible</name>
  <value>true</value>
</property>
```

注：

- 这个选项也需要启用 hplsql.use.only.one.connection。

19.1.2.6 hplsql.dbms.output.to.log

DBMS_OUTPUT.PUT_LINE 函数的内容通常只会显示在 UDR 服务器的终端上，启用此选项还会将内容写入日志文件$TRAF_HOME /log s/tracodion.sql.java.log 中。

例如:

```
<property>
  <name>hplsql.dbms.output.to.log</name>
  <value>true</value>
</property>
```