# 优炫数据库服务器编程手册 2.1

UXSINO
优 炫 软 件

# 优炫数据库服务器编程手册 2.1



## 法律声明

优炫数据库管理系统(简称：UXDB) 是由北京优炫软件股份有限公司开发并发布的一款商业性数据库管理系统。



### 注意

由于产品版本安装或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

北京优炫软件股份有限公司（总部）

- 地址：北京市海淀区学院南路62号中关村资本大厦11层（邮编：100081）
- 网址：http://www.uxsino.com
- 邮箱：<uxdb_support@uxsino.com>
- 电话：010-82886998
- 传真：010-82886338
- 服务热线：400-650-7837

# 目录

# 表格清单

# 范例清单

# 前言

## 1. 文档目的

本文档介绍了关于使用用户定义的函数、数据类型、触发器等扩展服务器功能。为相关技术人员和用户提供了必要的参考。

## 2. 文档对象

- 技术支持工程师
- 维护工程师
- 优炫数据库用户

## 3. 修改记录

修改记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

表 1. 文档更新记录

| 工具版本 | 发布日期 | 修改说明 |
| --- | --- | --- |
| 2.1.1.5C | 2022-12-09 | 第一次正式发布。 |

# 第 1 章 扩展 SQL

在下面的小节中，将讨论如何通过增加各种元素来扩展UXsinoDBSQL 查询语言。

- 函数（从第 1.3 节 “用户定义的函数”开始）
- 聚集（从第 1.12 节 “用户定义的聚集”开始）
- 数据类型（从第 1.13 节 “用户定义的类型”开始）
- 操作符（从第 1.14 节 “用户定义的操作符”开始）
- 用于索引的操作符类（从第 1.16 节 “索引的接口扩展”开始）
- 相关对象的包（从第 1.17 节 “打包相关对象到一个扩展中”开始）

## 1.1. 扩展性如何工作

UXsinoDB是可扩展的，因为它的操作是目录驱动的。如果熟悉标准的关系型数据库系统，会知道它们把有关数据库、表、列等的信息存储在总所周知的系统目录中（某些系统称之为数据目录）。目录对于用户来说好像其他的表一样，但是DBMS把自己的内部信息记录在其中。UXsinoDB和标准关系型数据库系统的一个关键不同是UXsinoDB在其目录中存储更多信息：不只是有关表和列的信息，还有关于数据类型、函数、访问方法等等的信息。这些表可以被用户修改，并且因为UXsinoDB的操作是基于这些表的，所以UXsinoDB可以被用户扩展。通过比较，传统数据库系统只能通过在源代码中改变硬编码的过程或者载入由DBMS提供者特殊编写的模块进行扩展。

此外，UXsinoDB服务器能够通过动态载入把用户编写的代码结合到它自身中。也就是，用户能够指定一个实现了一个新类型或函数的对象代码文件（例如一个共享库），并且UXsinoDB将按照要求载入它。把用SQL编写的代码加入到服务器会更繁琐。这种“即时”修改其操作的能力让UXsinoDB独特地适合新应用和存储结构的快速原型设计。

## 1.2. UXsinoDB类型系统

UXsinoDB数据类型被划分为基础类型、容器类型、域和伪类型。

### 1.2.1. 基础类型

基础类型是那些被实现在SQL语言层面之下的类型（通常用一种底层语言，如 C），例如integer。它们通常对应于常说的抽象数据类型。UXsinoDB只能通过由用户提供的函数在这类类型上操作，并且只能理解到用户描述这种类型行为的程度。

枚举（enum）类型可以被认为是基础类型的一个子类。主要区别是它们可以使用SQL命令创建，不需要用到底层的编程。

### 1.2.2. 容器类型

UXsinoDB有三种“容器”类型，它们是包含多个其他类型值的类型。它们是数组、组合以及范围。

数组可以保存全部是同种类型的多个值。为每一种基本类型、组合类型、范围类型以及域类型都会自动创建一个数组类型。但是没有数组的数组。就类型系统的认知而言，多维数组就和一维数组一样。

只要用户创建一个表，就会创建组合类型或者行类型。也可以使用CREATE TYPE来定义一个没有关联表的“stand-alone”组合类型。一个组合类型只是一个具有相关域名称的类型列表。一个组合类型的值是一个行或者域值记录。用户可以访问来自SQL查询的组成域。

范围类型可以保存同种类型的两个值，它们是该范围的上下界。范围类型是用户创建的，不过也存在一些内建的范围类型。

### 1.2.3. 域

一个域是基于一种特定底层类型的，并且出于很多目的它可以与其底层类型互换。不过，一个域能够具有约束来限制它的合法值于其底层基础类型允许值的一个子集。可以使用SQL命令CREATE DOMAIN创建域。

### 1.2.4. 伪类型

有一些用于特殊目的“伪类型”。伪类型不能作为表列或者容器类型的组件出现，但是它们能被用于声明函数的参数和结果类型。这在类型系统中提供了一种机制来标识函数的特殊分类。

### 1.2.5. 多态类型

特别让人感兴趣的五种伪类型是anyelement、anyarray、anynonarray、anyenum以及anyrange，它们被统称为多态类型。任何使用这些类型声明的函数被称作是一个多态函数。通过使用根据一次特定调用实际传递的数据类型所决定的相关数据类型，一个多态函数能够在多种不同数据类型上操作。

多态参数和结果是相互关联的，并且它们在解析调用多态函数的查询时被决定到一种特定的数据类型。每一个被声明为anyelement的位置（参数或返回值）被允许具有任意特定的实际数据类型，但是在任何给定的查询中它们必须全部是相同的实际类型。每一个被声明为anyarray的位置可以有任意数组数据类型，但是相似地，它们必须全部具有相同类型。并且类似地，被声明为anyrange的位置必须是全部是相同的范围类型。此外，如果有位置被声明为anyarray并且其他位置被声明为anyelement，anyarray位置中的实际数组类型必须是一个数组，该数组的元素都是出现在anyelement位置的同一种类型。相似地，如果有位置被声明为anyrange并且其他位置被声明为anyelement，anyrange位置的实际范围类型必须是一个范围，该范围的子类型是出现在anyelement位置的同一种类型。anynonarray被当做和anyelement相同，但是增加了额外的约束要求实际类型不能是一种数组类型。anyenum被当做和anyelement相同，但是增加了额外的约束要求实际类型不能是一种枚举类型。

因此，当使用一种多态类型声明了多于一个参数位置，有效效果是只有实际参数类型的某些组合才被允许。例如，一个被声明为equal(anyelement, anyelement)的函数将要求任意两个输入值，只要它们是同一种数据类型。

当一个函数的返回值被声明为多态类型时，必须至少有一个参数位置也是多态的，并且作为该参数提供的实际数据类型决定了该调用的实际结果类型。例如，如果还没有一种数组下标机制，可以定义一个函数来实现下标：subscript(anyarray, integer) returns anyelement。这个声明约束了实际的第一个参数是一种数组类型，并且允许解析器从实际的第一个参数类型推断正确的结果类型。另一个示例是一个被声明为f(anyarray) returns anyenum的函数将只接受枚举类型的数组。

注意anynonarray和anyenum并不表示独立的类型变量，它们是和anyelement相同的类型，只是有一个额外的约束。例如，将一个函数声明为f(anyelement, anyenum)等效于把它声明为f(anyenum, anyenum)：两种实际参数必须是相同的枚举类型。

一个可变函数（可以有可变数量的参数，如第 1.5.5 节 “带有可变数量参数的SQL函数”中所述）能够是多态的：这可以通过声明其最后一个参数为VARIADIC anyarray来实现。为了匹配和决定实际结果类型的参数，这样一种函数的行为和写了合适数量的anynonarray参数是一样的。

## 1.3. 用户定义的函数

UXsinoDB提供四种函数，如下所示。

- 查询语言函数（用SQL编写的函数）（第 1.5 节 “查询语言（SQL）函数”）
- 过程语言函数（例如，用PL/uxSQL或PL/Tcl编写的函数）（第 1.8 节 “过程语言函数”）
- 内部函数（第 1.9 节 “内部函数”）
- C 语言函数（第 1.10 节 “C 语言函数”）

每一类函数可以采用基本类型、组合类型或者它们的组合作为参数。此外，每一类函数可以返回一个基本类型或一个组合类型。函数也能被定义成返回基本类型或组合类型值的集合。

很多类函数可以接受或者返回特定的伪类型（例如，多态类型），但是可用的功能会变化。详情可以参考每一种函数的描述。

定义SQL函数最容易，大部分SQL函数的概念也能用到其他类型的函数上。

# 1.4. 用户定义的过程

过程是一种类似于函数的数据库对象。两者的区别在于过程不返回值，因此没有返回类型声明。而函数可以作为一个查询或者DML命令的一部分被调用，过程则需要明确地用CALL语句调用。

本章剩余部分中对如何定义用户定义的函数的解释同样适用于过程，不同的地方有：需要使用CREATE PROCEDURE命令定义、没有返回类型、一些如严格性这样的其他特性不适用。

函数和过程一起构成了例程。有ALTER ROUTINE以及DROP ROUTINE这样的命令可以操作函数和过程而不需要知道它们是哪一种。不过，要注意没有CREATE ROUTINE命令。

# 1.5. 查询语言（SQL）函数

SQL 函数执行一个由任意 SQL 语句构成的列表，返回列表中最后一个查询的结果。在简单（非集合）的情况中，最后一个查询的结果的第一行将被返回（记住一个多行结果的“第一行”不是良定义的，除非使用ORDER BY）。如果最后一个查询正好根本不返回行，将会返回空值。

或者，一个 SQL 函数可以通过指定函数的返回类型为SETOF *sometype*被声明为返回一个集合（也就是多个行），或者等效地声明它为RETURNS TABLE(*columns*)。在这种情况下，最后一个查询的结果的所有行会被返回。下文将给出进一步的细节。

一个 SQL 函数的主体必须是一个由分号分隔的 SQL 语句的列表。最后一个语句之后的分号是可选的。除非函数被声明为返回void，最后一个语句必须是一个SELECT或者一个带有RETURNING子句的INSERT、UPDATE或者DELETE。

SQL语言中的任何命令集合都能被打包在一起并且被定义成一个函数。除了SELECT查询，命令可以包括数据修改查询（INSERT、UPDATE以及DELETE）和其他 SQL 命令（不能在SQL函数中使用事务控制命令，例如COMMIT、SAVEPOINT，以及一些工具命令，例如VACUUM）。不过，最后一个命令必须是一个SELECT或者带有一个RETURNING子句，该命令必须返回符合函数返回类型的数据。或者如果要定义一个执行动作但是不返回有用的值的函数，可以把它定义为返回void。例如，这个函数从emp表中移除具有负值薪水的行，如下所示。

```
CREATE FUNCTION clean_emp() RETURNS void AS '
  DELETE FROM emp
```

```
    WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();

 clean_emp
-----------

(1 row)
```

> **注意**
>
> 在被执行前，SQL 函数的整个主体都要被解析。虽然 SQL 函数可以包含修改系统目录的命令（如CREATE TABLE），但这类命令的效果对于该函数中后续命令的解析分析不可见。例如，如果把CREATE TABLE foo (...); INSERT INTO foo VALUES(...);打包到一个 SQL 函数中是得不到预期效果的，因为在解析INSERT命令时foo还不存在。在这类情况下，推荐使用PL/uxSQL而不是 SQL 函数。

CREATE FUNCTION命令的语法要求函数体被写作一个字符串常量。使用用于字符串常量的美元引用通常最方便。选择使用常规的单引号引用的字符串常量语法，必须在函数体中双写单引号（'）和反斜线（\）（假定转义字符串语法）。

## 1.5.1. SQL函数的参数

一个 SQL 函数的参数可以在函数体中用名称或编号引用。下面会有两种方法的示例。

要使用一个名称，将函数参数声明为带有一个名称，然后在函数体中只写该名称。如果参数名称与函数内当前 SQL 命令中的任意列名相同，列名将优先。如果不想这样，可以用函数本身的名称来限定参数名，也就是*function_name.argument_name*（如果这会与一个被限定的列名冲突，照例还是列名赢得优先。可以通过为 SQL 命令中的表选择一个不同的别名来避免这种混淆）。

在更旧的数字方法中，参数可以用语法$*n*引用：$1指的是第一个输入参数，$2指的是第二个，以此类推。不管特定的参数是否使用名称声明，这种方法都有效。

如果一个参数是一种组合类型，那么点号记法（如*argname.fieldname* 或$1.*fieldname*）也可以被用来访问该参数的属性。同样，可能需要用函数的名称来限定参数的名称以避免歧义。

SQL 函数参数只能被用做数据值而不能作为标识符，如下所示。

```
INSERT INTO mytable VALUES ($1);
```

如下所示是不可以的。

```
INSERT INTO $1 VALUES (42);
```

> **注意**
>
> UXsinoDB可以使用名称来引用 SQL 函数参数。要在老的服务器中使用的函数必须使用$*n*记法。

## 1.5.2. 基本类型上的SQL

最简单的SQL函数没有参数并且简单地返回一个基本类型，例如integer。

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;

-- Alternative syntax for string literal:
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();

 one
-----
   1
```

注意为该函数的结果在函数体内定义了一个列别名（名为result），但是这个列别名在函数以外是不可见的。因此，结果被标记为one而不是result。

定义用基本类型作为参数的SQL函数，如下所示。

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$
    SELECT x + y;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

省掉参数的名称而使用数字，如下所示。

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

这里是一个更有用的函数，它可以被用来借记一个银行账号，如下所示。

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
```

```
    WHERE accountno = tf1.accountno;
  SELECT 1;
$$ LANGUAGE SQL;
```

一个用户可以这样执行这个函数来从账户 17 中借记 $100.00，如下所示。

```
SELECT tf1(17, 100.0);
```

在这个示例中，为第一个参数选择了名称accountno，但是这和表bank中的一个列名相同。在UPDATE命令中， accountno引用列bank.accountno，因此tf1.accountno必须被用来引用该参数。当然可以通过为该参数使用一个不同的名称来避免这样的问题。

实际上从该函数得到一个更有用的结果而不是一个常数1，因此一个更可能的定义如下所示。

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
  UPDATE bank
    SET balance = balance - debit
    WHERE accountno = tf1.accountno;
  SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

它会调整余额并且返回新的余额。 同样的事情也可以用一个使用RETURNING的命令实现，如下所示。

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
  UPDATE bank
    SET balance = balance - debit
    WHERE accountno = tf1.accountno
  RETURNING balance;
$$ LANGUAGE SQL;
```

SQL函数必须恰好返回其声明的结果类型。这可能会要求插入一个显式的造型。例如，假设想要add_em函数返回类型float8。如下做法不可行。

```
CREATE FUNCTION add_em(integer, integer) RETURNS float8 AS $$
  SELECT $1 + $2;
$$ LANGUAGE SQL;
```

即便在其他的环境中UXsinoDB也会想要插入一个隐式造型把integer转换成float8。如下所示。

```
CREATE FUNCTION add_em(integer, integer) RETURNS float8 AS $$
  SELECT ($1 + $2)::float8;
$$ LANGUAGE SQL;
```

## 1.5.3. 组合类型上的SQL函数

在编写使用组合类型参数的函数时，必须指定想要的参数，还要指定参数的期望属性（域）。例如，假定emp是一个包含雇员数据的表，并且因此它也是该表每一行的组合类型的名称。这里是一个函数double_salary，它计算某个人的双倍薪水，如下所示。

```
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
    cubicle     point
);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';

 name | dream
------+-------
 Bill |  8400
```

注意语法$1.salary的使用是要选择参数行值的一个域。还要注意调用的SELECT命令是如何使用*table_name*.*来选择一个表的整个当前行作为一个组合值的。该表行也可以只用表名来引用，如下所示。

```
SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

但这种用法已被废弃因为它很容易让人搞混。

有时候实时构建一个组合参数很方便。这可以用ROW结构完成。例如，可以调整被传递给函数的数据，如下所示。

```
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
    FROM emp;
```

也可以构建一个返回组合类型的函数。返回单一emp行的函数，如下所示。

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,
        1000.0 AS salary,
        25 AS age,
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

在这个示例中，为每一个属性指定了一个常量值，但是可以用任何计算来替换这些常量。

有关定义函数有两件重要的事情，如下所示。

- 查询中的选择列表顺序必须和列在与组合类型相关的表中出现的顺序完全相同（如上所示，列的命名与系统无关）。

- 必须确保每个表达式的类型匹配该组合类型相应的列，必要时插入一个造型。否则有如下所示的错误。

```
ERROR:  function declared to return emp returns varchar instead of text at column 1
```

与基础类型的情况一样，该函数将不会自动插入任何造型。

定义同样的函数的一种不同的方法，如下所示。

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

这里写了一个只返回正确组合类型的单一列的SELECT。 在这种情况下这种写法实际并非更好，但是它在一些情况下比较方便 — 例如，需要通过调用另一个返回所期望的组合值的函数来计算结果。另一个示例是，如果尝试写一个函数返回组合上的一个域，而不是返回纯粹的组合类型，那么总是有必要把它写成返回单一列，因为没有其他办法能够产生一个正好是那种域类型的值。

可以直接调用这个函数或者在一个值表达式中使用它，如下所示。

```
SELECT new_emp();

        new_emp
------------------------
 (None,1000.0,25,"(2,2)")
```

或者把它当做一个表函数调用，如下所示。

```
SELECT * FROM new_emp();

 name | salary | age | cubicle
------+--------+-----+---------
 None | 1000.0 |  25 | (2,2)
```

第二种方式在第 1.5.7 节 “SQL 函数作为表来源”中有更完全的描述。

当使用一个返回组合类型的函数时，可能只想要其结果中的一个域（属性），如下所示。

```
SELECT (new_emp()).name;

 name
------
 None
```

额外的圆括号是必须的，它用于避免解析器被搞混。如果不写这些括号，如下所示。

```
SELECT new_emp().name;
ERROR:  syntax error at or near "."
```

```
LINE 1: SELECT new_emp().name;
                         ^
```

另一个选项是使用函数记号来抽取一个属性，如下所示。

```
SELECT name(new_emp());

 name
------
 None
```

字段记法和函数记法是等效的。

另一种使用返回组合类型的函数的方法是把结果传递给另一个接收正确行类型作为输入的函数，如下所示。

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
   SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

## 1.5.4. 带有输出参数的SQL函数

一种描述一个函数的结果的替代方法是定义它的输出参数，如下所示。

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
LANGUAGE SQL;

SELECT add_em(3,7);
 add_em
--------
     10
(1 row)
```

这和第 1.5.2 节 “基本类型上的SQL”中展示的add_em版本没有本质上的不同。输出参数的真正价值是它们提供了一种方便的方法来定义返回多个列的函数，如下所示。

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT x + y, x * y'
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
```

```
(1 row)
```

这里实际发生的是为该函数的结果创建了一个匿名的组合类型。上述示例具有与下面相同的最终结果，如下所示。

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

但是不必单独定义组合类型常常很方便。注意输出参数的名称并非只是装饰，而且决定了匿名组合类型的列名（如果为一个输出参数忽略了名称，系统将自行选择一个名称）。

在从 SQL 调用这样一个函数时，输出参数不会被包括在调用参数列表中。这是因为UXsinoDB只考虑输入参数来定义函数的调用签名。这也意味着在为诸如删除函数等目的引用该函数时只有输入参数有关系。可以用下面的命令之一删除上述函数。

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

参数可以被标记为IN（默认）、OUT、INOUT或者VARIADIC。一个INOUT参数既作为一个输入参数（调用参数列表的一部分）又作为一个输出参数（结果记录类型的一部分）。VARIADIC参数是输入参数，但被按照后文所述特殊对待。

## 1.5.5. 带有可变数量参数的SQL函数

只要“可选的”参数都是相同的数据类型，SQL函数可以被声明为接受可变数量的参数。可选的参数将被作为一个数组传递给该函数。声明该函数时要把最后一个参数标记为VARIADIC，这个参数必须被声明为一个数组类型，如下所示。

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
   SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
 mleast
--------
    -1
(1 row)
```

实际上，所有位于或者超过VARIADIC位置的实参会被收集成一个一位数组，如下所示。

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);   -- 不起作用
```

但是实际无法这样写 — 或者说至少它将无法匹配这个函数定义。一个被标记为VARIADIC的参数匹配其元素类型的一次或者多次出现，而不是它自身类型的出现。

有时候能够传递一个已经构造好的数组给 variadic 函数是有用的，特别是当一个 variadic 函数想要把它的数组参数传递给另一个函数时这会特别方便。此外，这是在一个允许不可信用户创建对象的方案中调用一个variadic函数的唯一安全的方式。可以通过在调用中指定VARIADIC来做到这一点，如下所示。

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

这会阻止该函数的 variadic 参数扩展成它的元素结构，从而允许数组参数值正常匹配。VARIADIC只能被附着在函数调用的最后一个实参上。

在调用中指定VARIADIC也是将空数组传递给 variadic 函数的唯一方式，如下所示。

```
SELECT mleast(VARIADIC ARRAY[]::numeric[]);
```

简单地写成SELECT mleast()是没有作用的，因为一个 variadic参数必须匹配至少一个实参（如果想允许这类调用，可以定义第二个没有 参数且也叫mleast的函数）。

从一个 variadic 参数产生的数组元素参数会被当做自己不具有名称。这意味着不能使用命名参数调用 variadic 函数，除非指定了VARIADIC。如下所示的调用是可以使用的。

```
SELECT mleast(VARIADIC arr => ARRAY[10, -1, 5, 4.4]);
```

如下所示的调用是不可以使用的。

```
SELECT mleast(arr => 10);
SELECT mleast(arr => ARRAY[10, -1, 5, 4.4]);
```

## 1.5.6. 带有参数默认值的SQL函数

函数可以被声明为对一些或者所有输入参数具有默认值。只要调用函数时没有给出足够多的实参，就会插入默认值来弥补缺失的实参。由于参数只能从实参列表的尾部开始被省略，在一个有默认值的参数之后的所有参数都不得不也具有默认值（尽管使用命名参数记法可以允许放松这种限制，这种限制仍然会被强制以便位置参数记法能工作）。不管是否使用它，这种能力都要求在某些用户不信任其他用户的数据中调用函数时做一些预防措施。

如下所示。

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
   SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);
 foo
-----
  60
(1 row)

SELECT foo(10, 20);
 foo
-----
  33
```

```
(1 row)

SELECT foo(10);
 foo
-----
  15
(1 row)

SELECT foo();  -- 因为第一个参数没有默认值，所以会失败
ERROR:  function foo() does not exist
```

=符号也可以用来替代关键词 DEFAULT。

## 1.5.7. SQL 函数作为表来源

所有的 SQL 函数都可以被用在查询的FROM子句中，但是对于返回组合类型的函数特别有用。如果函数被定义为返回一种基本类型，该表函数会产生一个单列表。如果该函数被定义为返回一种组合类型，该表函数会为该组合类型的每一个属性产生一列。

提供一个示例，可以把函数结果的列当作常规表的列来使用，如下所示。

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
   SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(1 row)
```

注意只从函数得到了一行。这是因为没有使用SETOF。这会在下一节中介绍。

## 1.5.8. 返回集合的SQL函数

当一个 SQL 函数被声明为返回SETOF*sometype*时，该函数的最后一个查询会被执行完，并且它输出的每一行都会被作为结果集的一个元素返回。

在FROM子句中调用函数时通常会使用这种特性。在这种情况下，该函数返回的每一行都变成查询所见的表的一行。例如，假设表foo具有和上文一样的内容，如下所示。

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
   SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

结果如下所示。

```
 fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

也可以返回多个带有由输出参数定义的列的行，如下所示。

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

SELECT * FROM sum_n_product_with_tab(10);
 sum | product
-----+---------
  11 |      10
  13 |      30
  15 |      50
  17 |      70
(4 rows)
```

这里的关键点是必须写上RETURNS SETOF record来指示该函数返回多行而不是一行。如果只有一个输出参数，则写上该参数的类型而不是record。

通过多次调用集合返回函数来构建查询的结果非常有用，每次调用的参数来自于一个表或者子查询的连续行。做这种事情最好的方法是使用LATERAL关键词。如下是使用集合返回函数枚举树结构中元素的示例。

```
SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
 SubChild2 | Child1
(6 rows)

CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL STABLE;

SELECT * FROM listchildren('Top');
 listchildren
```

```
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, child FROM nodes, LATERAL listchildren(name) AS child;
  name |  child
--------+-----------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

这个示例和使用的简单连接的效果没什么不同，但是在更复杂的计算中，把一些工作放在函数中会是一种很方便的选项。

返回集合的函数也能在查询的选择列表中调用。对于该查询本身产生的每一行都会调用集合返回函数，并且会从该函数的结果集中的每一个元素生成一个输出行。之前的示例也可以用这样的查询实现，如下所示。

```
SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
  name  | listchildren
--------+--------------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

在最后一个SELECT中，注意对于Child2、Child3等没有出现输出行。这是因为listchildren对这些参数返回空集，因此没有产生结果行。这和使用LATERAL语法时，与该函数结果的内连接得到的行为是一样的。

UXsinoDB中，写在查询的选择列表中的集合返回函数的行为几乎和写在LATERAL FROM子句项中的集合返回函数完全一样。如下所示。

```
SELECT x, generate_series(1,5) AS g FROM tab;
```

几乎等效于

SELECT x, g FROM tab, LATERAL generate_series(1,5) AS g;

这会是完全一样的，除了在这个特别的示例中，规划器会选择把g放在嵌套循环连接的外侧，因为g对tab没有实际的横向依赖。那会导致一种不同的输出行顺序。选择列表中的集合返回函数总是会被计算，就好像它们在FROM子句剩余部分的嵌套循环连接的内侧一样，因此在考虑来自FROM子句的下一行之前，这些函数会运行到完成。

如果在查询的选择列表中有不止一个集合返回函数，则行为类似于把那些函数放到一个单一的LATERAL ROWS FROM( ... ) FROM子句项中的行为。对于来自底层查询的每一行，都有一个用到每个函数首个结果的输出行，然后是一个使用每个函数第二个结果的输出行，以此类推。如果某些集合返回函数产生的输出比其他函数少，会用空值代替缺失的数据，因此为一个底层行形成的总行数等于产生最多输出的集合返回函数的输出行数。因此集合返回函数会“步调一致”地运行直到它们的输出被耗尽，然后用下一个底层行继续执行。

集合返回函数可以被嵌套在一个选择列表中，不过在FROM子句项中不允许这样做。在这种情况下，嵌套的每一层会被单独对待，就像它是一个单独的LATERAL ROWS FROM( ... )项一样。例如，在下面语句中，集合返回函数srf2、srf3和srf5将为tab的每一行步调一致地运行，然后会对较低层的函数产生的每一行以步调一致的形式应用srf1和srf4。

SELECT srf1(srf2(x), srf3(y)), srf4(srf5(z)) FROM tab;

在CASE或COALESCE这样的条件计算结构中，不能使用集合返回函数。例如，在如下语句中，看起来这个语句应该产生满足x > 0的输入行的五次重复，以及不满足的行的一次重复。但实际上，由于在CASE表达时被计算前，generate_series(1, 5)会被运行在一个隐式的LATERAL FROM项中，它会为每个输入行产生五次重复。为了减少混乱，这类情况会产生一个解析时错误。

SELECT x, CASE WHEN x > 0 THEN generate_series(1, 5) ELSE 0 END FROM tab;

### 注意

如果函数的最后一个命令是带有RETURNING的INSERT、UPDATE或者DELETE，该命令将总是会被执行完，即使函数没有用SETOF定义或者调用查询不要求取出所有结果行也是如此。RETURNING子句产生的多余的行会被悄无声息地丢掉，但是在命令的目标表上的修改仍然会发生（而且在从该函数返回前就会全部完成）。

### 注意

在UXsinoDB之前版本中，把多个集合返回函数放在同一个选择列表中的行为并不容易察觉，除非它们总是产生同等的行数。否则，得到的输出行数将会是各集合返回函数产生的行数的最小公倍数。此外，嵌套的集合返回函数不会按照上述的方式工作。相反，一个集合返回函数只能有最多一个集合返回参数，集合返回函数的每一次嵌套会被独立运行。此外，条件执行（CASE等中的集合返回函数）以前是被允许的，但是会让事情更加复杂。在编写需要在较老的UXsinoDB版本中工作的查询时，推荐使用LATERAL语法，因为这种语法能够在不同的版本间提供一致的结果。如果有一个依赖于集合返回函数的条件执行，那么可能可以通过把条件测试移到一个自定义集合返回函数中来修正该问题。例如，

```
SELECT x, CASE WHEN y > 0 THEN generate_series(1, z) ELSE 5 END
 FROM tab;
```

可以变成

```
CREATE FUNCTION case_generate_series(cond bool, start int, fin int, els int)
  RETURNS SETOF int AS $$
BEGIN
  IF cond THEN
    RETURN QUERY SELECT generate_series(start, fin);
  ELSE
    RETURN QUERY SELECT els;
  END IF;
END$$ LANGUAGE pluxsql;

SELECT x, case_generate_series(y > 0, 1, z, 5) FROM tab;
```

这种表达形式将在所有版本的UXsinoDB中以相同的方式工作。

## 1.5.9. 返回TABLE的SQL函数

还有另一种方法可以把函数声明为返回一个集合，即使用 RETURNS TABLE(*columns*)语法。 这等效于使用一个或者多个OUT参数外加把函数标记为返回 SETOF record（或者是SETOF单个输出参数的 类型）。这种写法是在最近的 SQL 标准中指定的，因此可能比使用 SETOF的移植性更好。

求和并且相乘也可以使用如下语句。

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

不允许把显式的OUT或者INOUT参数用于RETURNS TABLE记法 — 必须把所有输出列放在TABLE列表中。

## 1.5.10. 多态SQL函数

SQL函数可以被声明为接受并且返回多态类型anyelement、anyarray、anynonarray、anyenum以及 anyrange。更多关于多态函数的解释请见第 1.2.5 节 “多态类型”。这是一个从两种任意数据类型的元素构建一个数组的多态函数make_array，如下所示。

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}
(1 row)
```

注意类型造型'a'::text的使用是为了指定该参数的类型是text。如果该参数只是一个字符串这就是必须的，因为否则它会被当作unknown类型，并且unknown的数组也不是一种合法的类型。如果没有改类型造型，会出现如下错误。

```
ERROR:  could not determine polymorphic type because input has type "unknown"
```

允许具有多态参数和固定的返回类型，但是反过来不行。如下所示。

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A function returning a polymorphic type must have at least one polymorphic argument.
```

多态化可以用在具有输出参数的函数上，如下所示。

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |   f3
----+---------
 22 | {22,22}
(1 row)
```

多态化也可以用在 variadic 函数上，如下所示。

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);
 anyleast
----------
       -1
(1 row)

SELECT anyleast('abc'::text, 'def');
 anyleast
```

```
----------
 abc
(1 row)

CREATE FUNCTION concat_values(text, VARIADIC anyarray) RETURNS text AS $$
   SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat_values('|', 1, 4, 2);
 concat_values
---------------
 1|4|2
(1 row)
```

### 1.5.11. 带有排序规则的SQL函数

当一个SQL函数具有一个或者更多可排序数据类型的参数时，对每一次函数调用都会根据分配给实参的排序规则为其确定一个排序规则。如果成功地确定（即在参数之间没有隐式排序规则的冲突），那么所有的可排序参数都被认为隐式地具有该排序规则。这将会影响函数中对排序敏感的操作的行为。例如，使用anyleast函数时，结果将依赖于数据库的默认排序规则，如下所示。

```
SELECT anyleast('abc'::text, 'ABC');
```

在C区域中，结果将是ABC，但是在很多其他区域中它将是abc。可以在任意参数上增加一个COLLATE子句来强制要使用的排序规则，如下所示。

```
SELECT anyleast('abc'::text, 'ABC' COLLATE "C");
```

此外，如果希望一个函数用一个特定的排序规则工作而不管用什么排序规则调用它，可根据需要在函数定义中插入COLLATE子句。这种版本的anyleast将总是使用en_US区域来比较字符串，如下所示。

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
   SELECT min($1[i] COLLATE "en_US") FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

但是注意如果应用到不可排序数据类型上，这将会抛出一个错误。

如果在实参之间无法确定共同的排序规则，那么 SQL 函数会把它的参数当作拥有其数据类型的默认排序规则（通常是数据库的默认排序规则，但是域类型的参数可能会不同）。

可排序参数的行为可以被想成是多态的一种受限形式，只对于文本数据类型有效。

## 1.6. 函数重载

可以用同样的 SQL 名称定义多于一个函数，只要它们的参数不同即可。换句话说，函数名可以被重载。不管是否使用它，这种能力都要求在某些用户不信任其他用户的数据中调用函数时做一些预防措施。当一个查询被执行时，服务器将从数据类型和所提供的参数个数来决定要调用哪个函数。重载也可用来模拟具有可变参数个数（最大个数有限）的函数。

在创建一个重载函数家族时，应该小心不要创建歧义。例如，给定如下函数。

CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...

对于test(1, 1.5)这样的输入就无法立刻清楚地知道应该调用哪个函数。当前实现的解决规则请参见《优炫数据库参考手册 V2.1》“SQL语言”章节中的“类型转换”，但是设计一个依赖于这种行为的系统是不明智的。

一个具有单个组合类型参数的函数通常不应与该类型的任何属性（域）重名。回想一下，*attribute*(*table*)被认为等效于*table.attribute*。在出现“一个组合类型上的函数”与“组合类型的一个属性”的情况下，将总是使用属性。可以通过用模式限定该函数名（即*schema.func*(*table*)）来覆盖这种选择，但是最好不要选择有冲突的名称以避免此类问题。

另一种可能的冲突在于 variadic 和非 variadic 函数之间。例如，可以创建foo(numeric)和foo(VARIADIC numeric[])。 在这种情况下，对于提供了一个数字参数的调用（例如foo(10.1)）就不清楚应该匹配哪一个函数。规则是使用在搜索路径中出现得较早的函数，或者当两者都在同一个模式中时优先使用非 variadic 的那一个函数。

在重载 C 语言函数时有一个额外的约束：重载函数家族中的每一个函数的 C 名称必须与其他所有函数的 C 名称不同，不管是内部的还是动态载入的。如果这条规则被违背，该行为将不可移植。可能会得到一个运行时链接器错误，或者这些函数之一将被调用（通常 是内部的那一个）。SQL **CREATE FUNCTION**命令的AS子句的另一种形式可以把 SQL 函数名和 C 源代码中的函数名分离。如下所示。

```
CREATE FUNCTION test(int) RETURNS int
    AS 'filename', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'filename', 'test_2arg'
    LANGUAGE C;
```

这里的 C 函数名称反映了很多种可能的习惯之一。

# 1.7. 函数易变性分类

每一个函数都有一个易变性分类，可能是VOLATILE、STABLE或者IMMUTABLE。 如果CREATE FUNCTION命令没有指定一个分类，则默认是VOLATILE。易变性分类是给优化器的关于该函数行为的一种承诺，如下所示。

- 一个VOLATILE函数可以做任何事情，包括修改数据库。在使用相同的参数连续调用时，它能返回不同的结果。优化器不会对这类函数的行为做任何假定。在每一行需要 volatile 函数值时，一个使用 volatile 函数的查询都会重新计算该函数。

- 一个STABLE函数不能修改数据库并且被确保对一个语句中的所有行用给定的相同参数返回相同的结果。这种分类允许优化器把该函数的多个调用优化成一个调用。特别是，在一个索引扫描条件中使用包含这样一个函数的表达式是安全的（因为一次索引扫描只会计算一次比较值，而不是为每一行都计算一次，在一个索引扫描条件中不能使用 VOLATILE函数）。

- 一个IMMUTABLE函数不能修改数据库并且被确保用相同的参数永远返回相同的结果。这种分类允许优化器在一个查询用常量参数调用该函数时提前计算该函数。例如，一个SELECT ... WHERE x = 2 + 2这样的查询可以被简化为SELECT ... WHERE x = 4，因为整数加法操作符底层的函数被标记为IMMUTABLE。

为了最好的优化结果，应该把函数标记为对它们合法的易变性分类中最严格的那种。

任何带有副作用的函数必须被标记为VOLATILE，这样对它的调用就不能被优化掉。甚至如果一个函数的值在一个查询中会 变化，即使它没有副作用也需要被标记为VOLATILE。这样的示例有random()、currval()、timeofday()等。

另一种重要的示例是current_timestamp家族的函数有资格被标记为STABLE，因为它们的值在一个事务中不会改变。

在考虑先规划然后立即执行的简单交互式查询时，在STABLE和IMMUTABLE分类间的区别相对较小：一个函数是在规划时只执行一次还是在查询执行开始期间只执行一次没有太大关系。但是如果计划被保存下来然后在后面被重用，区别就大了。如果在不允许过早把一个函数变成规划期间的一个常数时把它标记为IMMUTABLE，会导致在后续重用该计划时用到一个陈旧的值。当使用预备语句或者使用会缓存计划的函数语言（PL/uxSQL）时，将会产生严重的错误。

对于用 SQL 或者其他任何标准过程语言编写的函数，还有第二种由易变性分类决定的特性，即由调用该函数的 SQL 命令所作的数据修改的可见性。VOLATILE函数将看到这些更改，STABLE或者IMMUTABLE函数则看不到。这种行为使用 MVCC 的快照行为实现：STABLE和IMMUTABLE函数使用一个在调用查询开始时建立的快照，而VOLATILE函数在它们执行的每一个查询的开始都获得一个新鲜的快照。

> **注意**
>
> 用 C 编写的函数按照它们自己需要的方式管理快照，但是通常最好让 C 函数也按照上面的方式来。

由于这种快照行为，一个只包含SELECT命令的函数可以被安全地标记为STABLE，即便它选择的表可能正在被并发查询所修改。UXsinoDB将使用为调用查询所建立的快照来执行STABLE函数中的所有命令，因此它将在整个查询期间看到一种数据库的固定视图。

对IMMUTABLE函数中的SELECT使用了相同的快照行为。通常在一个IMMUTABLE函数中从数据库表选择是不明智的，因为如果表内容变化就会破坏不变性。不过，UXsinoDB不会强制不让这样做。

一种常见的错误是当一个函数的结果依赖于一个配置参数时把它标记为IMMUTABLE。例如，一个操纵时间戳的函数有可能结果依赖于TimeZone设置。为了安全起见，这类函数应该被标记为STABLE。

> **注意**
>
> UXsinoDB要求STABLE和IMMUTABLE函数中不包含非SELECT的 SQL 命令以阻止数据修改（这也不是完全万无一失，因为这类函数还可以调用修改数据库的VOLATILE函数。如果那样做，将发现该STABLE或IMMUTABLE函数不会发现由被调用函数所作的数据库改变，因为它们对它的快照不可见）。

# 1.8. 过程语言函数

UXsinoDB允许用除 SQL 和 C 之外的语言编写用户定义的函数。这些语言通常被称为过程语言（PL）。 过程语言并不内建在UXsinoDB服务器中，它们通过可装载模块提供。更多信息参见第 5 章 过程语言以及接下来的章节。

# 1.9. 内部函数

内部函数由 C 编写并且已经被静态链接到UXsinoDB服务器中。该函数定义的“主体”指定该函数的 C 语言名称， 它必须和声明 SQL 函数所用的名称一样（为了向后兼容性的原因，也接受空主体，那时会认为 C 语言函数名与 SQL 函数名相同）。

通常，所有存在于服务器中的内部函数都在数据库集簇的初始化期间被声明，但是用户可以使用CREATE FUNCTION为一个内部函数创建额外的别名。在CREATE FUNCTION中用语言名internal来声明内部函数。例如，要为sqrt函数创建一个别名，如下所示。

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
   AS 'dsqrt'
   LANGUAGE internal
   STRICT;
```

（大部分内部函数应该被声明为“严格”）。

> **注意**
>
> 上述场景中并非所有“预定义”的函数都是 “内部”函数。有些预定义的函数由 SQL 编写。

# 1.10. C 语言函数

用户定义的函数可以用 C 编写（或者可以与 C 兼容的语言，例如 C++）。这类函数被编译成动态载入对象（也被称为共享库）并且由服务器在需要时载入。动态载入是把“C语言”函数和“内部”函数区分开的特性 — 两者真正的编码习惯实际上是一样的（因此，标准的内部函数库是用户定义的 C 函数很好的源代码实例）。

当前仅有一种调用约定被用于C函数（“版本1”）。如下文所示，为函数编写一个UX_FUNCTION_INFO_V1()宏就能指示对该调用约定的支持。

## 1.10.1. 动态载入

在一个会话中第一次调用一个特定可载入对象文件中的用户定义函数时，动态载入器会把那个对象文件载入到内存以便该函数被调用。因此用户定义的 C 函数的CREATE FUNCTION必须为该函数指定两块信息：可载入对象文件的名称，以及要在该对象文件中调用的特定函数的 C 名称（链接符号）。如果没有显式指定 C 名称，则它被假定为和 SQL 函数名相同。

下面的算法被用来基于CREATE FUNCTION命令中给定的名称来定位共享对象文件。

1. 如果名称是一个绝对路径，则载入给定的文件。
2. 如果该名称以字符串$libdir开始，那么这一部分会被UXsinoDB包的库目录名（在编译时确定）替换。
3. 如果该名称不包含目录部分，会在配置变量dynamic_library_path指定的路径中搜索该文件。
4. 否则（在该路径中没找到该文件，或者它包含一个非绝对目录），动态载入器将尝试接受给定的名称，这大部分会导致失败（依赖当前工作目录是不可靠的）。

如果这个序列不起作用，会把平台相关的共享库文件名扩展（通常是.so）追加到给定的名称并且再次尝试上述的过程。如果还是失败，则载入失败。

推荐相对于$libdir或者通过动态库路径来定位共享库。如果升级版本时新的安装在一个不同的位置，则可以简化升级过程。$libdir实际表示的目录可以用命令ux_config --pkglibdir来找到。

用于运行UXsinoDB服务器的用户 ID 必须能够通过要载入文件的路径。常见的错误是把文件或更高层的目录变得对uxdb用户不可读或者不可执行。

在任何情况下，CREATE FUNCTION命令中给定的文件名会被原封不动地记录在系统目录中，这样如果需要再次载入该文件则会应用同样的过程。

> **注意**
>
> UXsinoDB不会自动编译 C 函数。在从CREATE FUNCTION命令中引用对象文件之前，它必须先被编译好。更多信息参见第 1.10.5 节 “编译和链接动态载入的函数”。

为了确保动态载入对象文件不会被载入到一个不兼容的服务器，UXsinoDB会检查该文件是否包含一个带有合适内容的“magic block”。这允许服务器检测到明显的不兼容，例如为不同UXsinoDB主版本编译的代码。要包括一个 magic block，在写上包括头文件fmgr.h的语句之后，在该模块的源文件之一（并且只能在其中一个）中添加如下内容。

```
UX_MODULE_MAGIC;
```

在被第一次使用后，动态载入对象文件会留在内存中。在同一个会话中对该函数未来的调用将只会消耗很小的负荷进行符号表查找。如果需要重新载入一个对象文件（例如重新编译以后），需要开始一个新的会话。

可以选择让一个动态载入文件包含初始化和终止化函数。如果文件包含一个名为_UX_init的函数，则文件被载入后会立刻调用该函数。该函数不接受参数并且应该返回 void。如果文件包括一个名为_UX_fini的函数，则在卸载该文件之前会立即调用该函数。同样地，该函数不接受参数并且应该返回 void。注意将只在卸载文件的过程中会调用_UX_fini，进程结束时不会调用它（当前，卸载被禁用并且从不发生，但是未来可能会改变）。

## 1.10.2. C 语言函数中的基本类型

要了解如何编写 C 语言函数，需要了解UXsinoDB如何在内部表达基本数据类型以及如何与函数传递它们。在内部，UXsinoDB把一个基本类型认为是“一团内存”。在类型上定义的用户定义函数说明了UXsinoDB在该类型上操作的方式。也就是说，UXsinoDB将只负责把数据存在磁盘以及从磁盘检索数据，而使用用户定义函数来输入、处理和输出该数据。

基本类型有三种内部格式之一，如下所示。

- 传值，定长
- 传引用，定长
- 串引用，变长

传值类型在长度上只能是 1、2 或 4 字节（如果机器上sizeof(Datum)是 8，则还有 8 字节）。应当小心地定义类型以便它们在所有的架构上都是相同的尺寸（字节）。例如，long类型很危险，因为它在某些机器上是 4 字节但在另外一些机器上是 8 字节，而int类型在大部分 Unix 机器上都是 4 字节。在 Unix 机器上int4类型一种合理的实现可能如下所示。

```
/* 4 字节整数，传值 */
typedef int int4;
```

（实际的 UXsinoDB C 代码会把这种类型称为int32，因为C中的习惯是int*XX*表示*XX*位。注意因此还有尺寸为 1 字节的 C 类型int8。SQL 类型int8在 C 中被称为int64。参见表 1.1 “内建 SQL 类型等效的 C 类型”）。

在另一方面，任何尺寸的定长类型可以用传引用的方法传递。例如，这里有一种UXsinoDB类型的实现示例，如下所示。

```
/* 16 字节结构，传引用 */
typedef struct
{
    double  x, y;
} Point;
```

在UXsinoDB函数中传进或传出这种类型时，只能使用指向这种类型的指针。要返回这样一种类型的值，用palloc分配正确的内存量，然后填充分配好的内存，并且返回一个指向该内存的指针（还有，如果只想返回与具有相同数据类型的一个输入参数相同的值，可以跳过额外的palloc并且返回指向该输入值的指针）。

最后，所有变长类型必须也以引用的方式传递。所有变长类型必须用一个正好 4 字节的不透明长度域开始，该域会由SET_VARSIZE设置，绝不要直接设置该域！所有要被存储在该类型中的数据必须在内存中接着该长度域的后面存储。长度域包含该结构的总长度，也就是包括长度域本身的尺寸。

另一个重点是要避免在数据类型值中留下未被初始化的位。例如，要注意把可能存在于结构中的任何对齐填充字节置零。如果不这样做，数据类型的逻辑等价常量可能会被规划器认为是不等的，进而导致低效的（不过还是正确的）计划。

> **警告**
>
> 绝不要修改通过引用传递的输入值的内容。如果这样做很可能会破坏磁盘上的数据，因为给出的指针可能直接指向一个磁盘缓冲区。这条规则唯一的例外在第 1.12 节 “用户定义的聚集”中有解释。

例如，可以这样定义类型text，如下所示。

```
typedef struct {
    int32 length;
    char data[FLEXIBLE_ARRAY_MEMBER];
} text;
```

[FLEXIBLE_ARRAY_MEMBER]记号表示数据部分的实际长度不由该声明指定。

在操纵变长字节时，必须小心地分配正确数量的内存并且正确地设置长度域。如果想在一个text结构中存储 40 字节，可以使用如下所示的代码片段。

```
#include "uxdb.h"
...
char buffer[40]; /* our source data */
```

```
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

VARHDRSZ和sizeof(int32)一样，但是用宏VARHDRSZ来引用变长类型的载荷的尺寸被认为是比较好的风格。还有，必须使用SET_VARSIZE宏来设置长度域，而不是用简单的赋值来设置。

表 1.1 “内建 SQL 类型等效的 C 类型”指定在编写使用一种UXsinoDB内建类型的 C 语言函数时，哪一种 C 类型对应于哪一种 SQL 类型。“定义文件”列给出了要得到该类型定义需要包括的头文件（实际的定义可能在一个由列举文件包括的不同文件中。推荐用户坚持使用已定义的接口）。注意在任何源文件中应该总是首先包括uxdb.h，因为它声明了很多需要的东西。

表 1.1. 内建 SQL 类型等效的 C 类型

| SQL 类型 | C 类型 | 定义文件 |
|---|---|---|
| boolean | bool | uxdb.h（可能是编译器内建） |
| box | BOX* | utils/geo_decls.h |
| bytea | bytea* | uxdb.h |
| "char" | char | （编译器内建） |
| character | BpChar* | uxdb.h |
| cid | CommandId | uxdb.h |
| date | DateADT | utils/date.h |
| smallint (int2) | int16 | uxdb.h |
| int2vector | int2vector* | uxdb.h |
| integer (int4) | int32 | uxdb.h |
| real (float4) | float4* | uxdb.h |
| double precision (float8) | float8* | uxdb.h |
| interval | Interval* | datatype/timestamp.h |
| lseg | LSEG* | utils/geo_decls.h |
| name | Name | uxdb.h |
| oid | oid | uxdb.h |
| oidvector | oidvector* | uxdb.h |
| path | PATH* | utils/geo_decls.h |
| point | POINT* | utils/geo_decls.h |
| regproc | regproc | uxdb.h |
| text | text* | uxdb.h |
| tid | ItemPointer | storage/itemptr.h |
| time | TimeADT | utils/date.h |
| time with time zone | TimeTzADT | utils/date.h |
| timestamp | Timestamp* | datatype/timestamp.h |

| SQL 类型 | C 类型 | 定义文件 |
| --- | --- | --- |
| varchar | VarChar* | uxdb.h |
| xid | TransactionId | uxdb.h |

## 1.10.3. 版本 1 的调用约定

版本-1 的调用规范依赖于宏来降低传参数和结果的复杂度。版本-1函数的 C 声明总是如下所示。

Datum funcname(UX_FUNCTION_ARGS)

此外，宏调用如下所示。

UX_FUNCTION_INFO_V1(funcname);

必须出现在同一个源文件中（按惯例会正好写在该函数本身之前）。这种宏调用不是internal语言函数所需要的，因为UXsinoDB会假定所有内部函数都使用版本-1规范。不过，对于动态载入函数是必需的。

在版本-1 函数中，每一个实参都使用对应于该参数数据类型的UX_GETARG_*xxx*()宏取得。在非严格的函数中，需要使用UX_ARGNULL_*xxx*()对参数是否为空提前做检查。结果要用对应于返回类型的UX_RETURN_*xxx*()宏返回。UX_GETARG_*xxx*()的参数是要取得的函数参数的编号，从零开始计。UX_RETURN_*xxx*()的参数是实际要返回的值。

这里是一些使用版本-1调用约定的示例，如下所示。

```
#include "uxdb.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

UX_MODULE_MAGIC;

/* 传值 */

UX_FUNCTION_INFO_V1(add_one);

Datum
add_one(UX_FUNCTION_ARGS)
{
   int32   arg = UX_GETARG_INT32(0);

   UX_RETURN_INT32(arg + 1);
}

/* 传引用，定长 */

UX_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(UX_FUNCTION_ARGS)
```

```
{
  /* FLOAT8 的宏隐藏了它的传引用本质。 */
  float8   arg = UX_GETARG_FLOAT8(0);

  UX_RETURN_FLOAT8(arg + 1.0);
}

UX_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(UX_FUNCTION_ARGS)
{
  /* 这里，Point 的传引用本质没有被掩盖。 */
  Point    *pointx = UX_GETARG_POINT_P(0);
  Point    *pointy = UX_GETARG_POINT_P(1);
  Point    *new_point = (Point *) palloc(sizeof(Point));

  new_point->x = pointx->x;
  new_point->y = pointy->y;

  UX_RETURN_POINT_P(new_point);
}

/* 传引用，变长 */

UX_FUNCTION_INFO_V1(copytext);

Datum
copytext(UX_FUNCTION_ARGS)
{
  text    *t = UX_GETARG_TEXT_PP(0);

  /*
   * VARSIZE_ANY_EXHDR是该结构的尺寸（以字节为单位）减去其头部的
   * VARHDRSZ或VARHDRSZ_SHORT。用一个完整长度的头部构建该拷贝。
   */
  text    *new_t = (text *) palloc(VARSIZE_ANY_EXHDR(t) + VARHDRSZ);
  SET_VARSIZE(new_t, VARSIZE_ANY_EXHDR(t) + VARHDRSZ);

  /*
   * VARDATA是指向新结构的数据区域的指针。来源可以是一个短数据，
   * 所以要通过VARDATA_ANY检索它的数据。
   */
  memcpy((void *) VARDATA(new_t), /* 目标 */
       (void *) VARDATA_ANY(t), /* 源头 */
       VARSIZE_ANY_EXHDR(t));  /* 多少字节 */
  UX_RETURN_TEXT_P(new_t);
}

UX_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(UX_FUNCTION_ARGS)
{
```

```
    text  *arg1 = UX_GETARG_TEXT_PP(0);
    text  *arg2 = UX_GETARG_TEXT_PP(1);
    int32 arg1_size = VARSIZE_ANY_EXHDR(arg1);
    int32 arg2_size = VARSIZE_ANY_EXHDR(arg2);
    int32 new_text_size = arg1_size + arg2_size + VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA_ANY(arg1), arg1_size);
    memcpy(VARDATA(new_text) + arg1_size, VARDATA_ANY(arg2), arg2_size);
    UX_RETURN_TEXT_P(new_text);
}
```

假定上述代码已经准备在文件funcs.c中并且被编译成一个共享对象，可以用这样的命令在UXsinoDB中定义函数，如下所示。

```
CREATE FUNCTION add_one(integer) RETURNS integer
    AS 'DIRECTORY/funcs', 'add_one'
    LANGUAGE C STRICT;

-- 注意SQL函数名“add_one”的重载
CREATE FUNCTION add_one(double precision) RETURNS double precision
    AS 'DIRECTORY/funcs', 'add_one_float8'
    LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
    AS 'DIRECTORY/funcs', 'makepoint'
    LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
    AS 'DIRECTORY/funcs', 'copytext'
    LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
    AS 'DIRECTORY/funcs', 'concat_text'
    LANGUAGE C STRICT;
```

这里，*DIRECTORY*表示共享库文件的目录（例如UXsinoDB的教程目录，它包含这一节中用到的示例的代码）。（更好的做法是先把*DIRECTORY*放入搜索路径，在AS子句中只使用'funcs'。在任何情况下，可以为一个共享库省略系统相关的扩展名，通常是.so）。

注意已经把函数指定为“strict”，这意味着如果有任何输入值为空，系统应该自动假定得到空结果。通过这种做法，避免在函数代码中检查空值输入。如果不这样做，必须使用UX_ARGISNULL()明确地检查空值输入。

乍一看，版本-1编码习惯好像是在使用普通C调用约定之上的无意义的愚民政策。不过，它们确实允许处理可为NULL的参数/返回值以及被“TOAST”过（压缩或者线外）的值。

宏UX_ARGISNULL(*n*)允许一个函数测试是否每一个输入为空（当然，只需要在没有声明为“strict”的函数中这样做）。和UX_GETARG_*xxx*()宏一样，输入参数也是从零开始计数。注意应该在验证了一个参数不是空之后才执行UX_GETARG_*xxx*()。要返回一个空结果，应执行UX_RETURN_NULL()，它对严格的以及非严格的函数都有用。

在版本-1接口中提供的其他选项是UX_GETARG_*xxx*()宏的两个变种。其中的第一种是UX_GETARG_*xxx*_COPY()，它确保返回的指定参数的拷贝可以被安全地写入（通常的宏有时会返回一个指向表中物理存储的值，它不能被写入。使用UX_GETARG_*xxx*_COPY()宏可以保证得到一个可写的结果）。第二种变种UX_GETARG_*xxx*_SLICE()宏有三个参数。第一个是函数参数的编号（如上文）。第二个和第三个是要被返回的段的偏移量和长度。偏移量从零开始计算，而负值的长度则表示要求返回该值的剩余部分。当大型值的存储类型为“external”时，这些宏提供了访问这些大型值的更有效的方法（列的存储类型可以使用ALTER TABLE *tablename* ALTER COLUMN *colname* SET STORAGE *storagetype*来指定。*storagetype*取plain、external、extended或者main）。

最后，版本-1 的函数调用规范可以返回集合结果（第 1.10.8 节 “返回集合”）、实现触发器函数（第 2 章 触发器）和过程语言调用处理器。更多细节可见源代码发布中的src/backend/utils/fmgr/README。

## 1.10.4. 编写代码

在开始更高级的话题之前，应该讨论一下用于UXsinoDB C 语言函数的编码规则。虽然可以把不是 C 编写的函数载入到UXsinoDB中，这通常是很困难的，因为其他语言（例如 C++、FORTRAN 或者 Pascal）通常不会遵循和 C 相同的调用规范。也就是说，其他语言不会以同样的方式在函数之间传递参数以及返回值。由于这个原因，会假定 C 语言函数确实是用 C 编写的。

编写和编译 C 函数的基本规则如下所示。

- 使用ux_config --includedir-server 找出UXsinoDB服务器头文件安装在 系统的哪个位置。
- 编译并且链接代码（这样它就能被动态载入到UXsinoDB中）总是要求特殊的标志。对特定的操作系统的做法详见第 1.10.5 节 “编译和链接动态载入的函数”
- 记住为共享库按第 1.10.1 节 “动态载入”中所述定义一个“magic block”。
- 在分配内存时，使用UXsinoDB函数palloc和 pfree，而不是使用对应的 C 库函数malloc和free。在每个事务结束时会自动释放通过palloc分配的内存，以免内存泄露。
- 总是要使用memset把结构中的字节置零（或者最开始就用palloc0分配它们）。即使对结构中的每个域都赋值，也可能有对齐填充（结构中的空洞）包含着垃圾值。如果不这样做，很难支持哈希索引或哈希连接，因为必须选出数据 结构中有意义的位进行哈希计算。规划器有时也依赖于用按位相等来比较常量，因此如果逻辑等价的值不是按位相等的会导致出现不想要的规划结果。
- 大部分的内部UXsinoDB类型都声明在uxdb.h中，不过函数管理器接口（UX_FUNCTION_ARGS等）在 fmgr.h中，因此将需要包括至少这两个文件。为了移植性，最好在包括任何其他系统或者用户头文件之前， 先包括uxdb.h。包括uxdb.h也将会包括elog.h和palloc.h。
- 对象文件中定义的符号名不能相互冲突或者与UXsinoDB服务器可执行程序中定义的符号冲突。如果出现有关于此的错误消息，必须重命名函数或者变量。

## 1.10.5. 编译和链接动态载入的函数

在使用 C 编写的UXsinoDB扩展函数之前，必须以一种特殊的方式编译并且链接它们，以便产生一个能被服务器动态载入的文件。简而言之，需要创建一个*共享库*。

超出本节所含内容之外的信息请参见操作系统文档，特别是 C 编译器（cc）和链接编辑器（ld）的手册页。另外，UXsinoDB源代码的contrib目录中包含了一些可以工作的示例。但是，如果依靠这些示例，也会让模块依赖于UXsinoDB源码的可用性。

创建共享库通常与链接可执行文件相似：首先源文件被编译成对象文件，然后对象文件被链接起来。对象文件需要被创建为*独立位置代码*（PIC），这在概念上意味着当它们被可执行文件载入时，它们可以被放置在内存中的任意位置（用于可执行文件的对象文件通常不会以那种方式编译）。链接一个共享库的命令会包含特殊标志来把它与链接一个可执行文件区分开（至少在理论上 — 在某些系统上实际上很丑陋）。

在下列示例中，假定源代码在一个文件foo.c中，并且创建一个共享库foo.so。除非特别注明，中间的对象文件将被称为foo.o。一个共享库能包含多于一个对象文件，但是在这里只使用一个。

FreeBSD

用来创建PIC的编译器标志是-fPIC。要创建共享库，编译器标志是-shared。

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

这适用于FreeBSD从 3.0 开始的版本。

HP-UX

该系统编译器创建PIC的编译器标志是+z。当使用GCC自己的-fPIC时。用于共享库的链接器标志是-b。如下所示。

```
cc +z -c foo.c
```

或者可以写成如下方式。

```
gcc -fPIC -c foo.c
```

然后如下所示。

```
ld -b -o foo.sl foo.o
```

和大部分其他系统不同，HP-UX为共享库使用扩展.sl。

Linux

创建PIC的编译器标志是-fpic。创建一个共享库的编译器标志是-shared。完整示例如下所示。

```
cc -fPIC -c foo.c
cc -shared -o foo.so foo.o
```

macOS

这里是一个示例。它假定安装了开发者工具。

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

创建PIC的编译器标志是-fPIC。对于ELF系统，带着标志-shared的编译器被用来链接共享库。在旧的非 ELF 系统上，ld -Bshareable会被使用。

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

创建PIC的编译器标志是-fPIC。ld -Bshareable被用来链接共享库。

```
gcc -fPIC -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

创建PIC的编译器标志是-KPIC（用于 Sun 编译器）以及-fPIC（用于GCC）。要链接共享库，编译器选项对几种编译器都是-G或者是对GCC使用-shared。

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

or

```
gcc -fPIC -c foo.c
gcc -G -o foo.so foo.o
```

> **提示**
>
> 如果过于复杂，可以考虑使用GNU Libtool，它会用一个统一的接口隐藏平台差异。

结果的共享库文件接着就可以被载入到UXsinoDB。当把文件名指定给**CREATE FUNCTION**命令时，必须把共享库文件的名字给它，而不是中间的对象文件。注意系统的标准共享库扩展（通常是.so或者.sl）在**CREATE FUNCTION**命令中可以被忽略，并且通常为了最好的可移植性应该被忽略。

服务器会期望在哪里寻找共享库文件请参见第 1.10.1 节 “动态载入”。

## 1.10.6. 组合类型参数

组合类型没有像 C 结构那样的固定布局。组合类型的实例可能包含空值域。此外，继承层次中的组合类型可能具有和同一继承层次中其他成员不同的域。因此，UXsinoDB提供了函数接口来访问 C 的组合类型的域。

假设想要写一个函数来回答查询，如下所示。

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
  FROM emp
```

```
    WHERE name = 'Bill' OR name = 'Sam';
```

如果使用版本-1的调用规范，可以定义c_overpaid，如下所示。

```
#include "uxdb.h"
#include "executor/executor.h"  /* 用于 GetAttributeByName() */

UX_MODULE_MAGIC;

UX_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(UX_FUNCTION_ARGS)
{
    HeapTupleHeader  t = UX_GETARG_HEAPTUPLEHEADER(0);
    int32            limit = UX_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        UX_RETURN_BOOL(false);
    /* 另外，可能更想对空 salary 用 UX_RETURN_NULL() 。*/

    UX_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

GetAttributeByName是返回指定行的属性的UXsinoDB系统函数。它有三个参数：类型为HeapTupleHeader的传入参数、想要访问的函数名以及一个说明该属性是否为空的返回参数。GetAttributeByName返回一个Datum值，可以把它用合适的DatumGet*XXX*()宏转换成正确的数据类型。注意如果空值标志被设置，那么返回值是没有意义的，所以在对结果做任何事情之前应该先检查空值标志。

也有GetAttributeByNum函数，它可以用目标属性的属性号而不是属性名来选择目标属性。

下面的命令声明 SQL 中的c_overpaid：

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
    AS 'DIRECTORY/funcs', 'c_overpaid'
    LANGUAGE C STRICT;
```

注意用了STRICT，这样不需要检查输入参数是否为 NULL。

## 1.10.7. 返回行（组合类型）

要从一个 C 语言函数中返回一个行或者组合类型值，可以使用一种特殊的 API，它提供的宏和函数隐藏了大部分的构建组合数据类型的复杂性。要使用这种 API，源文件中必须包括**#include "funcapi.h"**。

有两种方式可以构建一个组合数据值（以后就叫一个“元组”）：可以从一个 Datum 值的数组构造，或者从一个 C 字符串（可被传递给该元组各列的数据类型的输入转换函数）的数组构

造。在两种情况下，都首先需要为该元组的结构获得或者构造一个TupleDesc描述符。在处理 Datum 时，需要把该TupleDesc传递给BlessTupleDesc，接着为每一行调用heap_form_tuple。在处理 C 字符串时，需要把该TupleDesc传递给TupleDescGetAttInMetadata，接着为每一行调用BuildTupleFromCStrings。对于返回一个元组集合的函数，这些设置步骤可以在第一次调用该函数时一次性完成。

有一些助手函数可以用来设置所需的TupleDesc。在大部分返回组合值的函数中推荐的方式是调用。

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

传递传给调用函数本身的同一个fcinfo结构（这当然要求使用的是版本-1 的调用规范）。resultTypeId可以被指定为NULL或者一个本地变量的地址以接收该函数的结果类型OID。resultTupleDesc应该是一个本地TupleDesc变量的地址。检查结果是不是TYPEFUNC_COMPOSITE，如果是则resultTupleDesc已经被用所需的TupleDesc填充（如果不是，可以报告一个错误，并且 返回“function returning record called in context that cannot accept type record”字样的消息）。

> **提示**
>
> get_call_result_type能够解析一个多态函数结果的实际类型，因此不仅在返回组合类型的函数中，在返回标量多态结果的函数中它也是非常有用的。resultTypeId输出主要用于返回多态标量的函数。

> **注意**
>
> get_call_result_type有一个兄弟get_expr_result_type，它被用来解析被表示为一棵表达式树的函数调用的输出类型。在尝试确定来自函数外部的结果类型时可以用它。也有一个get_func_result_type，当只有函数的 OID 可用时可以用它。不过这些函数无法处理被声明为返回record的函数，并且get_func_result_type无法解析多态类型，因此应该优先使用get_call_result_type。

比较老的，现在已经被废弃的获得TupleDesc的函数如下所示。

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

它可以为一个提到的关系的行类型得到TupleDesc，如下所示。

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

可以基于一个类型 OID 得到TupleDesc。这可以被用来为一种基础或者组合类型获得TupleDesc。不过，对于返回record的函数它不起作用，并且它无法解析多态类型。

一旦有了一个TupleDesc，如果计划处理 Datum 可以进行如下调用。

TupleDesc BlessTupleDesc(TupleDesc tupdesc)

如果计划处理 C 字符串，可进行如下调用。

AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)

如果正在编写一个返回集合的函数，可以把这些函数的结果保存在FuncCallContext结构中 —分别使用*tuple_desc*或者*attinmeta*域。

在处理 Datum 时，用 Datum 形式的用户数据构建一个HeapTuple，如下所示。

HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)

在处理 C 字符串时，用 C 字符串形式的用户数据构建一个HeapTuple。*values*是一个 C 字符串数组，每一个元素是返回行的一个属性。每一个C 字符串应该是该属性数据类型的输入函数所期望的格式。为了对一个属性返回空值，*values*数组中对应的指针应该被设置为NULL。对于返回的每一行都将再次调用这个函数，如下所示。

HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)

一旦已经构建了一个要从函数中返回的元组，它必须被转换成一个Datum。把一个HeapTuple转换成合法的 Datum。如果只想返回一行，那么这个Datum可以被直接返回，在一个集合返回函数中它也可以被当做当前的返回值，如下所示。

HeapTupleGetDatum(HeapTuple tuple)

## 1.10.8. 返回集合

也提供了一种特殊的 API 来支持从 C 语言函数中返回集合（多个行）。集合返回函数必须遵循版本-1 的调用规范。如上文所述，源文件中还必须包括funcapi.h。

对返回的每一个项，一个集合返回函数（SRF）都会被调用一次。因此，这个SRF必须保存足够的状态来记住它正在做什么并且在每次调用时返回下一个项。结构FuncCallContext被提供来帮助控制这个过程。在一个函数中，fcinfo->flinfo->fn_extra被用来在多次调用中保持指向FuncCallContext的指针。

```
typedef struct FuncCallContext
{
  /*
   * 本次调用以前已经被调用过多少次
   *
   * SRF_FIRSTCALL_INIT() 会把 call_cntr 初始化为 0，
   * 并且在每次调用 SRF_RETURN_NEXT() 时增加。
   */
  uint64 call_cntr;

  /*
   * 可选：最大调用次数
   *
   * 这里的 max_calls 只是为了方便，设置它是可选的。
```

```
     * 如果没有设置，必须提供替代的方法来了解函数什么时候做完。
     */
    uint64 max_calls;

    /*
     * 可选：指向用户提供的上下文信息的指针
     *
     * user_fctx 是一个指向自己的数据的指针，它可用来在函数的多次
     * 调用之间保存任意的上下文信息。
     */
    void *user_fctx;

    /*
     * 可选：指向包含属性类型输入元数据的结构的指针
     *
     * attinmeta 被用在返回元组（即组合数据类型）时，在返回基本数据类型
     * 时不会使用。只有想用BuildTupleFromCStrings()创建返回元组时才需要它。
     */
    AttInMetadata *attinmeta;

    /*
     * 用于保存必须在多次调用间都存在的结构的内存上下文
     *
     * SRF_FIRSTCALL_INIT() 会设置 multi_call_memory_ctx，并且由
     * SRF_RETURN_DONE() 来清理。对于任何需要在 SRF 的多次调用间都
     * 存在的内存来说，它是最合适的内存上下文。
     */
    MemoryContext multi_call_memory_ctx;

    /*
     * 可选：指向包含元组描述的结构的指针
     *
     * tuple_desc 被用在返回元组（即组合数据类型）时，并且只有在用
     * heap_form_tuple() 而不是 BuildTupleFromCStrings() 构建元组时才需要它。
     * 注意这里存储的 TupleDesc 指针通常已经被先运行过 BlessTupleDesc()。
     */
    TupleDesc tuple_desc;

} FuncCallContext;
```

SRF使用一些自动操纵FuncCallContext （并且期望通过fn_extra找到它）的函数和宏。如下所示。

SRF_IS_FIRSTCALL()

判断函数是否是第一次被调用。在第一次调用时（只能在第一次调用时），如下所示。

SRF_FIRSTCALL_INIT()

可初始化FuncCallContext。在每一次函数调用时（包括第一次），如下所示。

SRF_PERCALL_SETUP()

为使用FuncCallContext做适当的设置并且清除上一次留下来的任何已返回的数据。

如果函数有数据要返回，如下所示。

SRF_RETURN_NEXT(funcctx, result)

把它返回给调用者（result必须是类型Datum，可以是一个单一值或者按上文所述准备好的元组）。最后，当函数完成了数据返回后，可使用如下示例 来清理并且结束SRF。

SRF_RETURN_DONE(funcctx)

SRF被调用时的当前内存上下文被称作一个瞬时上下文，在两次调用之间会清除它。这意味着不必对用palloc分配的所有东西调用pfree，它们将自动被释放。不过，如果想要分配任何需要在多次调用间都存在的数据结构，需要把它们放在其他地方。对于任何需要在SRF结束运行之前都存在的数据来说，*multi_call_memory_ctx*引用的内存上下文是一个合适的位置。在大部分情况中，这意味着应该在做第一次调用设置时就切换到*multi_call_memory_ctx*中。

> **警告**
>
> 虽然函数的实参在多次调用之间保持不变，但如果在瞬时上下文中反TOAST 了参数（通常由UX_GETARG_*xxx*宏完成），那么被反 TOAST 的拷贝将在每次循环中被释放。相应地，如果把这些值的引用保存在*user_fctx*中，也必须在反 TOAST 之后把它们拷贝到*multi_call_memory_ctx*中，或者确保只在那个上下文中反 TOAST 这些值。

完整的伪代码示例如下所示。

```
Datum
my_set_returning_function(UX_FUNCTION_ARGS)
{
  FuncCallContext  *funcctx;
  Datum            result;
  further declarations as needed

  if (SRF_IS_FIRSTCALL())
  {
    MemoryContext oldcontext;

    funcctx = SRF_FIRSTCALL_INIT();
    oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
    /* 这里是一次性设置代码： */
    user code
    if returning composite
      build TupleDesc, and perhaps AttInMetadata
    endif returning composite
    user code
    MemoryContextSwitchTo(oldcontext);
  }

  /* 这里是每一次都要做的设置代码： */
  user code
  funcctx = SRF_PERCALL_SETUP();
```

```
    user code

    /* 这里只是一种测试是否执行完的方法： */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* 这里返回另一个项： */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* 这里已经完成了项的返回并且需要进行清理： */
        user code
        SRF_RETURN_DONE(funcctx);
    }
}
```

一个返回组合类型的简单SRF的完整示例。如下所示。

```
UX_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(UX_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                 call_cntr;
    int                 max_calls;
    TupleDesc           tupdesc;
    AttInMetadata       *attinmeta;

    /* 只在第一次函数调用时做的事情 */
    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext   oldcontext;

        /* 创建一个函数上下文，让它在多次调用间都保持存在 */
        funcctx = SRF_FIRSTCALL_INIT();

        /* 切换到适合多次函数调用的内存上下文 */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

        /* 要返回的元组总数 */
        funcctx->max_calls = UX_GETARG_UINT32(0);

        /* 为结果类型构造一个元组描述符 */
        if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
            ereport(ERROR,
                    (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                     errmsg("function returning record called in context "
                            "that cannot accept type record")));

        /*
```

```
         * 生成后面需要用来从原始 C 字符串产生元组的属性元数据
         */
        attinmeta = TupleDescGetAttInMetadata(tupdesc);
        funcctx->attinmeta = attinmeta;

        MemoryContextSwitchTo(oldcontext);
    }

    /* 在每一次函数调用都要完成的事情 */
    funcctx = SRF_PERCALL_SETUP();

    call_cntr = funcctx->call_cntr;
    max_calls = funcctx->max_calls;
    attinmeta = funcctx->attinmeta;

    if (call_cntr < max_calls)    /* 如果还有要发送的 */
    {
        char      **values;
        HeapTuple    tuple;
        Datum        result;

        /*
         * 为构建返回元组准备一个值数组。这应该是一个 C
         * 字符串数组，之后类型输入函数会处理它。
         */
        values = (char **) palloc(3 * sizeof(char *));
        values[0] = (char *) palloc(16 * sizeof(char));
        values[1] = (char *) palloc(16 * sizeof(char));
        values[2] = (char *) palloc(16 * sizeof(char));

        snprintf(values[0], 16, "%d", 1 * UX_GETARG_INT32(1));
        snprintf(values[1], 16, "%d", 2 * UX_GETARG_INT32(1));
        snprintf(values[2], 16, "%d", 3 * UX_GETARG_INT32(1));

        /* 构建一个元组 */
        tuple = BuildTupleFromCStrings(attinmeta, values);

        /* 把元组变成 datum */
        result = HeapTupleGetDatum(tuple);

        /* 清理（实际并不必要） */
        pfree(values[0]);
        pfree(values[1]);
        pfree(values[2]);
        pfree(values);

        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* 如果没有要发送的 */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

在 SQL 中声明这个函数有两种方法，如下所示。

- 第一种方法

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

- 第二种方法，使用 OUT 参数

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

注意在这种方法中，函数的输出类型在形式上是一种匿名的record类型。

源码中的目录contrib/tablefunc下的模块包含集合返回函数更加复杂的示例。

## 1.10.9. 多态参数和返回类型

C语言函数可以被声明为接受和返回多态类型anyelement、anyarray、anynonarray、anyenum以及anyrange。关于多态函数的详细介绍请见第 1.2.5 节 “多态类型”。

当函数参数或者返回类型被定义为多态类型时，函数的编写者无法提前知道会用什么数据类型调用该函数或者该函数需要返回什么数据类型。在fmgr.h中提供了两种例程来允许版本-1 的C函数发现其参数的实际数据类型以及它要返回的类型。这些例程被称为get_fn_expr_rettype(FmgrInfo *flinfo)和get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)。它们返回结果或者参数的类型的 OID，或者当该信息不可用时返回InvalidOid。结构flinfo通常被当做fcinfo->flinfo访问。参数argnum则是从零开始计。get_call_result_type也可被用作get_fn_expr_rettype的一种替代品。还有get_fn_expr_variadic，它可以被用来找出 variadic 参数是否已经被合并到了一个数组中。这主要用于VARIADIC "any"函数，因为对于接收普通数组类型的variadic 函数来说总是会发生这类合并。

例如，假设要写一个接收一个任意类型元素并且返回一个该类型的一维数组的函数，如下所示。

```
UX_FUNCTION_INFO_V1(make_array);
Datum
make_array(UX_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid        element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum      element;
    bool       isnull;
    int16      typlen;
    bool       typbyval;
    char       typalign;
```

```
    int     ndims;
    int     dims[MAXDIM];
    int     lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* 得到提供的元素，小心它为 NULL 的情况 */
    isnull = UX_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = UX_GETARG_DATUM(0);

    /* 只有一个维度 */
    ndims = 1;
    /* 和一个元素 */
    dims[0] = 1;
    /* 且下界是 1 */
    lbs[0] = 1;

    /* 得到该元素类型所需的信息 */
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

    /* 现在构建数组 */
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    UX_RETURN_ARRAYTYPE_P(result);
}
```

下面的命令在 SQL 中声明函数make_array：

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
    AS 'DIRECTORY/funcs', 'make_array'
    LANGUAGE C IMMUTABLE;
```

有一种只对 C 语言函数可用的多态变体：它们可以被声明为接受类型为"any"的参数（注意这种类型名必须用双引号引用，因为它也是一个 SQL 保留字）。这和anyelement相似，不过它不约束不同的"any"参数为同一种类型，它们也不会帮助确定函数的结果类型。C 语言函数也能声明它的第一个参数为VARIADIC "any"。这可以匹配一个或者多个任意类型的实参（不需要是同一种类型）。这些参数不会像普通 variadic 函数那样被收集到一个数组中，它们将被单独传递给该函数。使用这种特性时，必须用UX_NARGS()宏以及上述方法来判断实参的个数和类型。还有，这种函数的用户可能希望在他们的函数调用中使用VARIADIC关键词，以期让该函数将数组元素作为单独的参数对待。如果想要这样，在使用get_fn_expr_variadic检测被标记为VARIADIC的实参之后，函数本身必须实现这种行为。

## 1.10.10. 共享内存和 LWLock

外接程序可以在服务器启动时保留LWLock和共享内存。必须通过在shared_preload_libraries中指定外接程序的共享库来预先载入它。从_UX_init函数中调用如下函数，可以保留共享内存。

void RequestAddinShmemSpace(int size)

通过从_UX_init中调用如下函数，可以保留LWLock。这将确保一个名为tranche_name的LWLock数组可用，该数组的长度为num_lwlocks。使用GetNamedLWLockTranche可得到该数组的指针。

void RequestNamedLWLockTranche(const char *tranche_name, int num_lwlocks)

为了避免可能的竞争情况，在连接并且初始化共享内存时，每一个后端应该使用LWLock AddinShmemInitLock，如下所示。

```
static mystruct *ptr = NULL;

if (!ptr)
{
    bool    found;

    LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
    ptr = ShmemInitStruct("my struct name", size, &found);
    if (!found)
    {
        initialize contents of shmem area;
        acquire any requested LWLocks using:
        ptr->locks = GetNamedLWLockTranche("my tranche name");
    }
    LWLockRelease(AddinShmemInitLock);
}
```

## 1.10.11. 把 C++ 用于可扩展性

尽管UXsinoDB后端是用 C 编写的，只要遵循下面的指导方针也可以用 C++ 编写扩展。

- 所有被后端访问的函数必须对后端呈现一种 C 接口，然后这些 C 函数调用 C++ 函数。例如，对后端访问的函数要求extern C链接。对需要在后端和 C++ 代码之间作为指针传递的任何函数也要这样做。
- 使用合适的释放方法释放内存。例如，大部分后端内存是通过palloc()分配的，所以应使用pfree()来释放。在这种情况中使用 C++ 的delete会失败。
- 防止异常传播到 C 代码中（在所有extern C函数的顶层使用一个捕捉全部异常的块）。即使 C++ 代码不会显式地抛出任何异常也需要这样做，因为类似内存不足等事件仍会抛出异常。任何异常都必须被捕捉并且用适当的错误传回给 C 接口。如果可能，用 -fno-exceptions 来编译 C++ 以完全消灭异常。在这种情况下，必须在 C++ 代码中检查失败，例如检查new()返回的 NULL。
- 如果从 C++ 代码调用后端函数，确定 C++ 调用栈值包含传统 C 风格的数据结构（POD）。这是必要的，因为后端错误会产生远距离的longjmp()，它无法正确的退回具有非 POD 对象的 C++ 调用栈。

总之，最好把 C++ 代码放在与后端交互的extern C函数之后，并且避免异常、内存和调用栈泄露。

# 1.11. 函数优化信息

默认情况下，函数只是一个“black box”，数据库系统对它的行为了解得很少。 不过，这意味着使用函数的查询执行效率可能会低于它们的能力。可以提供额外的知识帮助计划器优化函数调用。

一些基本事实可以通过CREATE FUNCTION命令中提供的声明性注释来提供。这里面最重要的是函数的volatility category (IMMUTABLE、STABLE或VOLATILE)；在定义函数时，要始终小心地正确指定这个。 并行安全属性(PARALLEL UNSAFE、PARALLEL RESTRICTED或PARALLEL SAFE)也必须被指定，如果希望在并行查询中使用该函数。 指定函数的估算执行开销也会有作用，和/或集返回函数估计返回的行数。不过，指定这两个事实的声明方式只允许指定常数值，而这通常是不够的。

也可以将一个planner support function 附加到SQL-可调用函数（称为target function）， 从而提供关于目标函数的知识，该函数过于复杂而无法以声明方式表示。 计划器支持函数必须写在 C 中（尽管它们的目标函数可以不是），所以这是一个高级功能，相对很少有人会使用。

计划器支持函数必须具有SQL签名，当建立目标函数时，它通过指定SUPPORT子句附加到它的目标函数。

supportfn(internal) returns internal

计划器支持函数的 API 的详细信息可以在 UXsinoDB源代码中的src/include/nodes/supportnodes.h文件中找到。 这提供了计划器支持函数的概述。支持函数的可能请求集合是可扩展的，所以在将来的版本中可能会有更多(功能)。

在规划期间，根据指定函数的特性，一些函数调用可以进行简化。例如，int4mul(n, 1)可以被简化为n。 这种类型的转换可以通过计划器支持函数执行，通过它实现 SupportRequestSimplify请求类型。对于在查询解析树中找到其目标函数的每个实例，将调用支持函数。如果它发现特定的调用可以简化成某种其他窗体，它可以构建并返回表示该表达式的解析树。 这将为基于函数的操作符自动工作，非常一在刚才的示例中，n * 1也将简化为n。（但注意这只是一个示例；这个特殊的优化实际上不是标准的UXsinoDB执行）。不保证UXsinoDB在支持函数能够简化的情况下，永远不会调用目标函数。 确保简化表达式与目标函数的实际执行之间严格等效。

对于返回 boolean 的目标函数，估计使用该函数的 WHERE 子句将选择的行的比重通常会有用。 这可以通过实现SupportRequestSelectivity请求类型的支持函数来完成。

如果目标函数的运行时间高度依赖于其输入，提供非固定开销估算可能很有用。这可以通过实现SupportRequestCost请求类型的支持函数来完成。

对于返回集的目标函数，为提供要返回的行数的非常量估计通常很有用。这可以通过实现SupportRequestRows请求类型的支持函数来完成。

对于返回 boolean的目标函数，可以将WHERE中出现的函数调用转换为一个可索引操作符子句或多个子句。转换的子句可能与函数的条件完全相同，或者它们可能比较弱态一些(也就是说，它们可能接受函数条件所不接受的一些值)。在后一种情况下，索引条件被称作lossy；它仍然可用于扫描索引，但必须为索引返回的每一行执行函数调用，以看它是否真的通过 WHERE条件或没有。要建立这样的条件，支持函数必须实现SupportRequestIndexCondition需求类型。

# 1.12. 用户定义的聚集

UXsinoDB中的聚集函数用状态值和状态转换函数定义。也就是，一个聚集操作使用一个状态值，它在每一个后续输入行被处理时被更新。要定义一个新的聚集函数，要为状态值选择一种数据类型、一个状态的初始值和一个状态转换函数。状态转换函数接收前一个状态值和该聚集当前行的

输入值，并且返回一个新的状态值。万一该聚集的预期结果与需要保存在运行状态之中的数据不同，还能指定一个*最终函数*。最终函数接受结束状态值并且返回作为聚集结果的任何东西。原则上，转换函数和最终函数只是也可以在聚集环境之外使用的普通函数（实际上，通常出于性能的原因，会创建特殊的只能作为聚集的一部分工作的转换函数）。

因此，除了该聚集的用户所见的参数和结果数据类型之外，还有一种可能不同于参数和结果状态的内部状态值数据类型。

如果定义一个聚集但不使用一个最终函数，就得到了一个从每一行的列值计算一个运行函数的聚集。sum是这类聚集的一个示例。sum从零开始，并且总是把当前行的值加到它的运行总和上。例如，如果希望让一个sum聚集能工作在复数数据类型上，只需要该数据类型的加法函数。聚集定义如下所示。

```
CREATE AGGREGATE sum (complex)
(
   sfunc = complex_add,
   stype = complex,
   initcond = '(0,0)'
);
```

可以进行如下操作。

```
SELECT sum(a) FROM test_complex;

  sum
-----------
 (34,53.9)
```

（注意依赖于函数重载：有多于一个名为sum的聚集，但是UXsinoDB能够找出哪种 sum 适用于一个类型为complex的列）。

如果没有非空输入值，上述的sum定义将返回零（初始状态值）。也许在这种情况下返回空—SQL 标准期望sum以这种方式行事。可以通过忽略initcond阶段简单地做到这一点，这样初始状态值就为空。通常这表示sfunc将需要检查一个空状态值输入。但是对于sum和一些其他简单聚集（如max和min），把第一个非空输入值插入到状态变量中并且接着在第二个非空输入值上开始应用转换函数就足够了。如果初始状态值为空并且转换函数被标记为“strict”（即不为空输入调用），UXsinoDB会自动这样做。

“strict”转换函数的另一点默认行为是只要碰到了一个空输入值，之前的状态值就保持不变。因此，空值会被忽略。如果需要某些其他用于空输入的行为，不要把转换函数声明为 strict，而是把它编码为测试空输入并且做所需要的事情。

avg（均值）是一种更复杂的聚集示例。它要求两份运行状态：输入的总和以及输入的计数。最终结果通过将这些量相除而得到。均值是使用一个数组作为状态值的典型实现。例如，内建的avg(float8)实现如下所示。

```
CREATE AGGREGATE avg (float8)
(
   sfunc = float8_accum,
   stype = float8[],
   finalfunc = float8_avg,
   initcond = '{0,0,0}'
```

);

> **注意**
>
> （float8_accum要求一个三元素的数组，而不只是两个元素，因为它累积平方和以及输入的总和以及计数。因此它也可以被用于其他聚集函数以及avg）。

SQL 中的聚集函数调用允许用DISTINCT和ORDER BY选项控制以什么顺序把行传递给该聚集的转换函数。这些选项的实现不需要该聚集的支持函数关心。

进一步的细节可见CREATE AGGREGATE命令。

## 1.12.1. 移动聚集模式

聚集函数可以选择性地支持*移动聚集模式*，这种模式允许很大程度上提高在具有移动帧起点的窗口中执行的聚集函数的速度。基本思想是在通常的“前向”转换函数之外，聚集提供一个*逆向转换函数*，该函数允许当行退出窗口帧时从聚集的运行状态值中移除它们的值。例如一个sum聚集使用加法作为前向转换函数，它可以使用减法作为逆向转换函数。如果没有一个逆向转换函数，每一次帧起点移动时，窗口函数机制必须重新从头计算该聚集，这会导致运行时间与输入行的数量乘以平均帧长度成比例。如果有一个逆向转换函数，运行时间只与输入行的数量成比例。

当前状态值和包含在当前状态中最早的行的聚集输入值被传递给逆向转换函数。它必须重新构建出如果给定的输入行不再被聚集（只聚集其后的行）时状态值会是什么样。这有时要求前向转换函数保存比普通聚集模式下更多的状态。因此，移动聚集模式使用一种完全不同于普通模式的实现：它有自己的状态数据类型、自己的前向转换函数以及自己的状态函数（如果需要）。如果不需要额外的状态，这些可以和普通模式的数据类型和函数相同。

作为一个示例，可以把上面给定的sum聚集扩展成支持移动聚集模式，如下所示。

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)',
    msfunc = complex_add,
    minvfunc = complex_sub,
    mstype = complex,
    minitcond = '(0,0)'
);
```

名称以m开始的参数定义移动聚集实现。除了逆向转换函数minvfunc，它们都对应于没有m的普通聚集参数。

用于移动聚集模式的前向转换函数不允许返回空值作为新状态值。如果逆向转换函数返回空值，这被当作一种指示，它表明该逆向函数无法为这个特定输入逆转状态计算，因此该聚集计算将根据当前的帧开始位置重新从头计算。这种习惯允许移动聚集模式被用在一些不适合逆转运行状态值的少数情况下。逆向转换函数在这些情况下可以“撒手不管”，然后在它能够工作的大部分情况中再出来干活。例如，一个浮点数的聚集可能会在必须从运行状态值中移除一个NaN（不是一个数字）输入时撒手不管。

在编写移动聚集支持函数时，很重要的是确保逆向转换函数能够准确地重构正确的状态值。否则会导致用不用移动聚集模式时结果中产生用户可见的差别。为一个聚集增加一个逆向转换函数的

示例最初看起来很简单，但是却无法满足float4或者float8输入上的sum的要求。sum(float8)的一种未经考虑的定义如下所示。

```
CREATE AGGREGATE unsafe_sum (float8)
(
   stype = float8,
   sfunc = float8pl,
   mstype = float8,
   msfunc = float8pl,
   minvfunc = float8mi
);
```

但是，这个聚集可能给出与没有逆向转换函数时很不同的结果。例如，如下查询返回0作为它的第二个结果，而不是期待的1。其原因是浮点值的有限精度：把1加到1e20还是会得到1e20，因此从中减去1e20会得到0而不是1。这是对于浮点计算的一种一般性限制，而不是UXsinoDB的限制。

```
SELECT
  unsafe_sum(x) OVER (ORDER BY n ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)
FROM (VALUES (1, 1.0e20::float8),
             (2, 1.0::float8)) AS v (n,x);
```

## 1.12.2. 多态和可变聚集

聚集函数可以使用多态状态转换函数或最终函数，这样同样的函数能被用来实现多个聚集。关于多态函数的解释可参见第 1.2.5 节 “多态类型”。更进一步，聚集函数本身可以被指定为具有多态输入类型和状态类型，允许一个聚集函数服务于多种输入数据类型。这里是一个多态聚集的示例，如下所示。

```
CREATE AGGREGATE array_accum (anyelement)
(
   sfunc = array_append,
   stype = anyarray,
   initcond = '{}'
);
```

这里，每一次给定聚集调用的实际状态类型是把实际输入类型作为元素的数组类型。该聚集的行为是串接所有输入成一个该类型的数组（注意：内建的聚集array_agg提供了相似的功能，但是具有比上述定义更好的性能）。

这里是使用两种不同实际数据类型作为参数的输出，如下所示。

```
•
   SELECT attrelid::regclass, array_accum(attname)
      FROM ux_attribute
      WHERE attnum > 0 AND attrelid = 'ux_tablespace'::regclass
      GROUP BY attrelid;

     attrelid   |            array_accum
  --------------+---------------------------------------
   ux_tablespace | {spcname,spcowner,spcacl,spcoptions}
  (1 row)
```

•
```
SELECT attrelid::regclass, array_accum(atttypid::regtype)
    FROM ux_attribute
    WHERE attnum > 0 AND attrelid = 'ux_tablespace'::regclass
    GROUP BY attrelid;

  attrelid   |       array_accum
-------------+--------------------------
 ux_tablespace | {name,oid,aclitem[],text[]}
(1 row)
```

参见上面示例，通常一个具有多态结果类型的聚集函数有一个多态状态类型。这是必须的，因为否则就无法有意义地声明最终函数：它会需要有一个多态结果类型但是不能有多态参数类型，CREATE FUNCTION将当场拒绝那些无法从调用中推断结果类型的函数。但是使用一个多态状态类型有时并不方便。最常见的情况是，聚集支持函数使用 C 编写并且状态类型应该被声明为internal，因为在 SQL 层面上没有与它等效的类型。为了表述这种情况，可以声明最终函数为接受额外的匹配该聚集输入参数的“dummy”参数。这种假参数总是被传递为空值，因为当最终函数被调用时没有特定的值可用。它们的唯一用途是允许一个多态最终函数的结果类型被连接到该聚集的输入类型。例如，内建聚集array_agg的定义等效于如下内容。

```
CREATE FUNCTION array_agg_transfn(internal, anynonarray)
  RETURNS internal ...;
CREATE FUNCTION array_agg_finalfn(internal, anynonarray)
  RETURNS anyarray ...;

CREATE AGGREGATE array_agg (anynonarray)
(
    sfunc = array_agg_transfn,
    stype = internal,
    finalfunc = array_agg_finalfn,
    finalfunc_extra
);
```

这里，finalfunc_extra选项指定该最终函数接收除了状态值之外，还接收对应于该聚集输入参数的额外假参数。额外的anynonarray参数允许array_agg_finalfn的声明成为合法。

与常规函数的习惯大致相同，可以通过把一个聚集函数的最后一个参数声明为一个VARIADIC数组，这样可以让该函数接受可变数量的参数（见第 1.5.5 节 “带有可变数量参数的SQL函数”）。该聚集的转换函数也必须有相同的数组类型作为它们的最后一个参数。通常这类转换函数也会被标上VARIADIC，但这不被严格要求。

> **注意**
>
> 可变聚集最容易被误用的情况是与ORDER BY选项，因为解析器无法在这样一种组合中是否给出了错误的实际参数数量。要记住在ORDER BY右侧的任何东西都是一个排序键，而不是一个聚集的参数。例如，在如下语句中，解析器将认为看到的是一个聚集函数参数和三个排序键。
>
> ```
> SELECT myaggregate(a ORDER BY a, b, c) FROM ...
> ```
>
> 但是，用户可能想要的是如下查询语句。

SELECT myaggregate(a, b, c ORDER BY a) FROM ...

如果myaggregate是可变的，两种调用都是合法的。

出于相同的原因，在创建具有相同名称以及不同数量的常规参数的聚集函数时一定要慎重考虑。

## 1.12.3. 有序集聚集

目前为止，描述的聚集都是“普通”聚集。UXsinoDB还支持有序集聚集，它和普通聚集在两个关键点上相区别。首先，除了对每个输入行都要计算一次的普通聚集参数之外，一个有序集聚集可以有“直接”参数，这类参数针对每次聚集操作只计算一次。其次，用于普通聚集参数的语法需要显式地为它们指定一个排序顺序。一个有续集聚集通常被用来实现一种依赖于特定行序的计算（例如排名或者百分位数），因此排序是任何调用都要求的。例如，percentile_disc的内建定义等效于如下示例。

```
CREATE FUNCTION ordered_set_transition(internal, anyelement)
  RETURNS internal ...;
CREATE FUNCTION percentile_disc_final(internal, float8, anyelement)
  RETURNS anyelement ...;

CREATE AGGREGATE percentile_disc (float8 ORDER BY anyelement)
(
   sfunc = ordered_set_transition,
   stype = internal,
   finalfunc = percentile_disc_final,
   finalfunc_extra
);
```

这个聚集接受一个float8直接参数（百分位数分数）以及一个可以是任意可排序数据类型的聚集输入。它可以用来得到一个家庭收入的中位数，如下所示。

```
SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_disc
-----------------
           50489
```

这里0.5是一个直接参数，它对于要作为一个在行之间变化的百分位数分数没有意义。

和普通聚集的情况不同，用于有序集聚集的输入行排序不是在幕后完成的，而是由该聚集的支持函数负责完成。典型的实现方法是在该聚集的状态值中保持对于一个“tuplesort”对象的引用，把到来的行输入给该对象，然后完成排序并且在最终函数中读出该数据。这种设计允许最终函数能够执行特殊操作，例如把附加的“假想”行注入到被排序的数据中。虽然用由PL/uxSQL或另一种 PL 语言编写的支持函数通常能够实现普通聚集，但是有序集聚集通常必须用 C 编写，因为它们的状态值无法用任何 SQL 数据类型来定义（在上面的示例中，注意状态值被声明为类型internal — 这很典型）。此外，由于最终函数会执行排序，后面就不能继续通过再次执行转移函数增加输入行。这意味着最终函数不是READ_ONLY的，它必须在CREATE AGGREGATE中被声明为READ_WRITE，或者在可以有额外的最终函数调用利用已经排序好的状态时声明为SHAREABLE。

用于一个有序集聚集的状态转移函数接收当前状态值外加对于每一行的聚集输入值，并且返回更新后的状态值。这和普通聚集的定义相同，但是注意没有提供直接参数（如果有）。最终函数接

收最后的状态值、直接参数（如果有）的值以及对应于聚集输入的空值（如果指定了finalfunc_extra）。正如普通聚集，只有聚集是多态时finalfunc_extra才真正有用，那时就需要额外的假参数把最终函数的结果类型连接到该聚集的输入类型。

当前，有序集聚集不能被用做窗口函数，并且因此没有必要让它们支持移动聚集模式。

## 1.12.4. 部分聚集

可选地，一个聚集函数可以支持*部分聚集*。部分聚集的思想是在输入数据的不同子集上独立的运行该聚集的状态转移函数，然后把从这些子集得到的状态值组合起来产生最终的状态值，这样得到的状态值与在单次聚集操作中扫描所有输入得到的状态值相同。这种模式可以被用来进行并行聚集，用不同的工作者进程扫描表的不同部分。每一个工作者产生一个部分状态值，最后把这些部分状态值组合产生最终状态值（在未来，这种模式可能也会被用于组合在本地表和远程表上的聚集，但目前还未实现）。

为了支持部分聚集，聚集定义必须提供一个*组合函数*，这个函数接收两个该聚集的状态类型（表示在输入行的两个不同子集上得到的聚集结果）并且产生一个该状态类型的新值，该结果表示组合哪些聚集结果后的状态。至于来自两个集合的输入行的相对顺序则并没有指定。这意味着通常不可能为对输入行顺序敏感的聚集定义出可用的组合函数。

作为简单的示例，通过指定组合函数为与其转移函数中相同的“两者中较大者”和“两者中较小者”比较函数，MAX和MIN聚集可以支持部分聚集。SUM聚集则只需要一个额外的函数作为组合函数（同样，组合函数与其转移函数相同，除非状态值的宽度比输入数据类型更宽）。

组合函数很像一个把状态类型值而不是底层输入类型值作为其第二个参数的转移函数。尤其是处理空值和严格函数的规则是相似的。此外，如果聚集定义指定了非空的initcond，记住那不仅会被作为每一次部分聚集运行的初始状态，还会被作为组合函数的初始状态，对每一个部分结果都会调用组合函数将部分结果组合到该初始状态中。

如果聚集的状态类型被声明为internal，则组合函数应负责在用于聚集状态值的内存上下文中分配其结果。这意味着当第一个输入为NULL时，不能简单地返回第二个输入，因为那个值将会在错误的上下文中并且将不具有足够的寿命。

当聚集的状态类型被声明为internal时，通常聚集定义提供*序列化函数*和*反序列化函数*也是合适的，这两个函数允许这样一种状态值被从一个进程复制到另一个进程。如果没有这些函数就无法执行并行聚集，并且未来的本地/远程聚集之类的应用也可能无法工作。

一个序列化函数必须接收一个单一的internal类型参数并且返回一个bytea类型的结果，它表示把状态值打包成一个平面化的字节串。反过来，反序列化函数是上述转换的逆变换。反序列化函数必须接收两个类型为bytea和internal的参数，并且返回类型为internal的结果（第二个参数没有被使用并且总是为零，它的存在是由于类型安全性的原因）。反序列化函数的结果应该直接在当前内存上下文中分配，这与组合函数的结果不同，因为它不需要长期存在。

还有一点值得提示的是关于要被并行执行的聚集，聚集本身必须被标记上PARALLEL SAFE。其支持函数上的并行安全性标记不会被参考。

## 1.12.5. 聚集的支持函数

用 C 编写的函数能够通过调用AggCheckCallContext检测它是作为聚集支持函数调用的，如下所示。

```
if (AggCheckCallContext(fcinfo, NULL))
```

检查这个区别的原因是当它为真时，第一个输入必须是一个临时状态值并且可以因此安全地被就地修改而不是分配一个新的副本。示例可见int8inc()（虽然聚集的转移函数总是被允许就地修改转移值，但不鼓励聚集的最终函数这样做。如果最终函数要这样做，必须在创建聚集时声明这种行为。更多细节请参见CREATE AGGREGATE）。

AggCheckCallContext的第二个参数可以被用来检索保存有聚集状态值的内存上下文。这对希望把“扩展”对象（见第 1.13.1 节 “TOAST 考量”）用作状态值的转移函数有用。在第一次调用时，转移函数应该返回一个扩展对象，其内存上下文是聚集状态上下文的一个子节点，然后在后续的调用中都保持返回同一个扩展对象。示例请见array_append()（array_append()不是任意内建聚集的转移函数，但其编写的目的就是在被用作一种自定义聚集的转移函数时表现得有效）。

另一种可用于由 C 编写的聚集函数的支持例程是AggGetAggref，它返回定义该聚集调用的Aggref解析节点。这主要对有序集聚集有用，它能检查Aggref的子结构来找出它们本应实现的排序顺序。在UXsinoDB源代码的orderedsetaggs.c中可以找到示例。

# 1.13. 用户定义的类型

如第 1.2 节 “UXsinoDB类型系统”中所述，UXsinoDB能够被扩展成支持新的数据类型。这一节描述了如何定义新的基本类型，它们是被定义在SQL语言层面之下的数据类型。创建一种新的基本类型要求使用低层语言（通常是 C）实现在该类型上操作的函数。

这一节中的示例可以在源代码src/tutorial目录下的complex.sql和complex.c中找到。运行这些示例的指令可以在该目录的README文件中找到。

一种用户定义的类型必须总是具有输入和输出函数。这些函数决定该类型如何出现在字符串中（用于用户输入或者对用户的输出）以及如何在内存中组织该类型。输入函数采用一个空终止的字符串作为它的参数并且返回该类型的内部（内存）表达。输出函数采用该类型的内部表达作为参数并且返回一个空终止的字符串。如果想对该类型做更多事情而不是只存储它，必须为任何操作提供额外的实现函数。

假设要定义一种类型complex，它表示复数。一种在内存中表达复数的自然的方法是下面的 C 结构，如下所示。

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

将需要让它成为一种传引用类型，因为它没办法放到一个单一的Datum值中。

至于该类型的外部字符串表达，选择了一种字符串形式的(x,y)。

输入和输出函数通常并不难编写，特别是输出函数。但是在定义类型的外部字符串表达时，必须最终为该表达编写一个完整并且鲁棒的解析器作为输入函数。如下所示。

```
UX_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(UX_FUNCTION_ARGS)
{
    char       *str = UX_GETARG_CSTRING(0);
    double      x,
```

```
            y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for type %s: \"%s\"",
                        "complex", str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    UX_RETURN_POINTER(result);
}
```

输出函数可以简单地写成如下内容。

```
UX_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(UX_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) UX_GETARG_POINTER(0);
    char       *result;

    result = psprintf("(%g,%g)", complex->x, complex->y);
    UX_RETURN_CSTRING(result);
}
```

应当让输入和输出函数互为彼此的逆函数。如果不这样做，当需要把数据转储到一个文件并且以后将它重新读入时会遇到很严重的问题。在涉及到浮点数时这是一个特别常见的问题。

可选地，一种用户定义的类型可以提供二进制输入和输出例程。二进制 I/O 通常比文本 I/O 更快但是可移植性更差。与文本 I/O 一样，需要定义准确的外部二进制表达。大部分的内建数据类型都尝试提供一种不依赖机器的二进制表达。对于 complex，将建立在为类型float8提供的二进制 I/O 转换器上，如下所示。

```
UX_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(UX_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) UX_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    UX_RETURN_POINTER(result);
}
```

```
UX_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(UX_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) UX_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    UX_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

一旦编写了 I/O 函数并且把它们编译到了一个共享库中，就可以在 SQL 中定义complex类型。首先把它声明为一种 shell 类型，如下所示。

```
CREATE TYPE complex;
```

这个语句的作用是为要定义的类型创建了一个占位符，这样允许定义其 I/O 函数时引用该类型。现在可以定义 I/O 函数，如下所示。

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;
```

最后，可以提供该数据类型的完整定义。

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
```

);

在定义了一种新的基本类型后， UXsinoDB会自动提供对这种类型的数组支持。数组类型通常具有和基本类型相同的名称以及一个前置的下划线字符（_）。

一旦数据类型存在，就能够声明额外的函数来提供在该数据类型上有用的操作。然后可以在函数之上定义操作符，并且如果需要，可以创建操作符类来支持对该数据类型进行索引。这些额外的内容会在下面的小节中讨论。

如果数据类型的内部表达是可变长的，则内部表达必须遵循可变长数据的标准布局：头四个字节必须是一个char[4]域，它从来不会被直接访问（通常被称为*vl_len_*）。必须使用SET_VARSIZE()宏在这个域中存储整个数据的尺寸（包括长度域本身），并且使用VARSIZE()来检索它（这些宏之所以存在，是因为长度域可能会根据平台来进行解码）。

更多细节请见CREATE TYPE命令的描述。

## 1.13.1. TOAST 考量

如果数据类型值的尺寸（内部形式）是可变的，更适合让该数据类型变成可TOAST的。即便值总是很小不会被压缩或者线外存储也应该这样做，因为TOAST也能通过减少头部负荷来为小数据减少空间。

为了支持TOAST存储，在该数据类型上操作的 C 函数必须总是要使用 UX_DETOAST_DATUM解包任何交给它们的被 TOAST 过的值（习惯上这些细节都通过定义类型相关的GETARG_DATATYPE_P宏隐藏起来）。然后，在运行CREATE TYPE命令时，指定内部长度为variable并且选择某个不是plain的适当的存储选项。

如果数据对齐无关紧要（不管是为一个特定函数或者因为数据类型指定了字节对齐），那么有可能避免UX_DETOAST_DATUM的一些开销。可以转而使用UX_DETOAST_DATUM_PACKED（习惯上通过定义一个GETARG_DATATYPE_PP宏隐藏）并且使用宏VARSIZE_ANY_EXHDR以及VARDATA_ANY来访问一个可能包装过的数据。此外，即使数据类型定义指定了一种对齐方式，这些宏返回的数据也不是对齐过的。如果对齐很重要，必须使用常规的UX_DETOAST_DATUM接口。

> **注意**
>
> 老的代码经常声明*vl_len_*为一个int32域而不是char[4]。只要结构定义含有其他具有至少int32对齐的域，这就是 OK 的。但是在使用可能未对齐的数据时，使用这样一种结构定义就是危险的，编译器可能会把它当作一个授权来假定数据实际上已经被对齐，在对于对齐很严格的架构上会导致核心转储。

TOAST支持带来的另一个特性是能够拥有一种扩展内存中数据表达，它比存储在磁盘上的格式使用起来更方便。常规的或者“扁平的” varlena 存储格式最终只是一堆字节，它不能包含指针，因为它可能会被复制到内存中的其他位置。对于复杂数据类型，扁平格式使用起来可能代价更高，因此UXsinoDB提供了一种方式把扁平格式“扩展”成更适合计算的一种表达，然后在该数据类型的函数之间传递这种在内存中的格式。

要使用扩展存储，数据类型必须遵循src/include/utils/expandeddatum.h中给定的规则定义一种扩展的格式，并且提供函数把扁平的 varlena 值“扩展”到该格式以及从该格式“扁平化”回常规的 varlena 表达。然后确保所有该数据类型的 C 函数都能接受这两种表达（可能通过一接收到其中一种就立刻转换成另一种来做到）。这不要求一次性修改所有该数据类型的现有函数，因为标准的UX_DETOAST_DATUM宏可以把扩展输入转换成常规扁平格式。因此，现有的用于扁平

varlena 格式的函数仍然能够用于扩展输入（虽然效率略低）。它不需要被转换，直到需要提高性能。

直到如何对付扩展表达的 C 函数通常分为两类：只能处理扩展格式的，以及能同时处理扩展或扁平 varlena 输入的。前者更容易编写，但是可能总体效率较低，因为由单个函数将一种扁平输入转换为扩展的形式的开销可能会超过在扩展格式上操作所节省的开销。在只需要处理扩展格式时，可以把扁平输入到扩展形式的转换隐藏在一个参数获取宏中，这样该函数就显得不比处理传统 varlena 输入的函数更复杂了。要处理两种类型的输入，需要编写一个参数获取函数来反 TOAST 外部、短头部以及压缩的 varlena 输入，但不需要处理扩展输入。这样一个函数可以被定义为返回一个指向由扁平 varlena 格式和扩展格式组成的联合的指针。调用者可以使用VARATT_IS_EXPANDED_HEADER()宏来判断它们接收到的是哪种格式。

TOAST机制不仅允许把常规 varlena 值同扩展值区分开来，还能区分指向扩展值的“read-write”和“read-only”指针。只需要检查扩展值或者只会以安全的并且非语义可见的方式更改扩展值的 C 函数不需要关心它们收到的是哪种类型的指针。如果收到一个读写指针，要为输入值产生一个修改版本的 C 函数将被允许就地修改该扩展输入值，但是如果它们收到的是一个只读指针则不能修改，在这种情况下它们不得不先复制该值产生一个用于修改的新值。构建了新扩展值的 C 函数应该总是返回一个指向该值的读写指针。还有，如果一个就地修改读写扩展值的 C 函数中途失败，它应该负责让该值处于一种正常的状态。

有关使用扩展值的示例，请见标准数组这种基础结构，特别是src/backend/utils/adt/array_expanded.c。

# 1.14. 用户定义的操作符

对于一个完成实际工作的底层函数的调用来说，每一个操作符都是“语法糖”，因此在创建操作符之前必须先创建底层函数。不过，一个操作符不只是语法糖，因为它携带了额外的信息来帮助查询规划器优化使用该操作符的查询。下一节将致力于解释这些额外信息。

UXsinoDB支持左一元、右一元和二元操作符。操作符可以被重载，也就是说相同的操作符名称可以被用于具有不同操作数数量和类型的操作符。在执行一个查询时，系统会根据提供的操作数的数量和类型决定要调用的操作符。

这里有一个创建用于对两个复数做加法的操作符的示例。假设已经创建了类型 complex（见第 1.13 节 “用户定义的类型”的定义。首先需要一个函数做这个加法，然后可以定义该操作符，如下所示。

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    function = complex_add,
    commutator = +
);
```

现在可以执行一个查询，如下所示。

```
SELECT (a + b) AS c FROM test_complex;
```

```
        c
----------------
 (5.2,6.05)
 (133.42,144.95)
```

这里已经展示了如何创建一个二元操作符。要创建一元操作符，只要忽略leftarg（左一元）和rightarg（右一元）之一即可。在CREATE OPERATOR中只要求procedure子句和参数子句。示例中展示的commutator子句是一个可选的子句，它被用作一个查询优化器使用的提示。有关commutator以及其他优化器提示的细节出现在下一小节中。

# 1.15. 操作符优化信息

一个UXsinoDB的操作符定义能够包括几种可选的子句，它们可以把有关操作符行为的有用的事情告诉系统。只要合适就应该提供这些子句，因为它们能够为使用该操作符的查询带来可观的速度提升。但是如果提供了它们，必须确保它们是正确的！不正确地使用一个优化子句可能导致很慢的查询、错误的输出或者其他不好的事情。如果没有把握可以省去优化子句，这样做的唯一后果是查询会比正常的速度慢。

在UXsinoDB的未来版本中可能会增加更多的优化子句。这里描述的优化子句都是版本 2.1 能理解的。

还可以将计划器支持的函数附加到作为操作符基础的函数中，从而提供另一种向系统讲述操作符行为的方法。更多信息参见第 1.11 节 “函数优化信息”。

## 1.15.1. COMMUTATOR

如果提供了COMMUTATOR子句，它指定一个操作符作为被定义的操作符的交换子。如果对于所有可能输入的 x、y 值， (x A y) 等于 (y B x)，可以说操作符 A 是操作符 B 的交换子。注意，B 也是 A 的交换子。例如，用于一种特定数据类型的操作符 < 和 > 通常互为交换子，并且操作符 + 通常和它本身是交换的。但是操作符 - 通常不能与任何东西交换。

一个可交换操作符的左操作数类型与其交换子的右操作数类型相同，反之亦然。因此要查找交换子，只需要给UXsinoDB该交换子操作符的名称即可，并且在COMMUTATOR子句中也只需要提供它的名称。

为将要在索引和连接子句中使用的操作符提供交换子信息是很关键的，因为这允许查询优化器把这样一个子句“翻转”成不同计划类型所需的形式。例如，考虑一个这样的 WHERE 子句tab1.x = tab2.y，其中tab1.x和tab2.y是一种用户定义的类型，并且假设tab2.y被索引。除非优化器能决定如何把该子句翻转成tab2.y = tab1.x，否则它无法产生一个索引扫描，因为索引扫描机制期望看到被索引列出现在被给出的操作符的左边。UXsinoDB将无法简单地假定有一个可用的变换 — =操作符的创建者必须指定它是合法的（通过为该操作符标记交换子信息）。

在定义一个子交换的操作符时，这样做就行了。自拟定义一堆交换的操作符时，事情有一点棘手：如何在没有定义第二个操作符时完成第一个操作符的定义？因为第一个操作符需要第二个操作符作为其交换子。对这个问题有两种解决方案，如下所示。

- 一种方法是忽略定义的第一个操作符的COMMUTATOR子句，并且然后在第二个操作符的定义中提供第一个操作符作为交换子。由于UXsinoDB知道交换的操作符是成对出现的，当它看到第二个定义时它将自动回去并且填上第一个定义中缺失的COMMUTATOR子句。

- 另一种更直接的方法是就在两个定义中包括COMMUTATOR子句。当UXsinoDB处理第一个定义并且意识到COMMUTATOR引用了一个不存在的操作符时，系统将为那个操作符在系统目录中创造一个虚拟项。这个虚拟项只有操作符名称、左右操作数类型和结果类型的数据，因为

这些是UXsinoDB在此时能够推断出来的所有东西。第一个操作符的目录项将会链接到这个虚拟项。稍后，当定义第二个操作符时，系统用来自第二个定义的额外信息更新那个虚拟项。如果尝试在虚拟操作符还未被填充之前使用它，将只会得到一个错误消息。

## 1.15.2. NEGATOR

如果提供了NEGATOR子句，它指定一个操作符是正在被定义的操作符的求反器。如果操作符 A 和 B 都返回布尔结果并且对于所有可能的 x、y 输入都有 (x A y) 等于 NOT (x B y)，那么可以说 A 是 B 的求反器。注意 B 也是 A 的求反器。例如，< 和 >= 就是大部分数据类型的一对求反器。一个操作符不可能是它自身的求反器。

与交换子不同，一对一元操作符可以合法地被标记为对方的求反器。这意味着对于所有 x 有 (A x) 等于 NOT (B x)，或者对右一元操作符也相似。

一个操作符的求反器必须具有和被定义的操作符相同的左或右操作数类型，因此正如COMMUTATOR一样，NEGATOR子句中只需要给出操作符的名称即可。

提供一个求反器对查询优化器非常有帮助，因为它允许NOT (x = y)这样的表达式被简化为x <> y。这可能比想象的更多地发生，因为NOT操作可能会被作为其他调整的结果被插入。

求反器对的定义可以使用与定义交换子对相同的方法来完成。

## 1.15.3. RESTRICT

如果提供了RESTRICT子句，它为该操作符指定一个限制选择度估计函数（注意这是一个函数名而不是一个操作符名）。RESTRICT子句只对返回boolean的二元操作符有意义。一个限制选择度估计器背后的思想是猜测一个表中有多大比例的行对于当前的操作符和一个特定的常数值将会满足一个column OP constant形式的WHERE子句条件。这能通过告知优化器具有这种形式的WHERE子句将会消除掉多少行来协助它的工作（如果常数位于左部会发生什么？可以使用COMMUTATOR）。

编写一个新的限制选择度估算函数已经超出了本章的范围，但是通常可以将系统的一个标准估算器用于很多自己的操作符。标准的限制估算器如下所示。

eqsel用于=
neqsel用于<>
scalarltsel用于<
scalarlesel用于<=
scalargtsel用于>
scalargesel用于>=

能经常成功地为具有非常高或者非常低选择度的操作符使用eqsel或neqsel，即使它们实际上并非相等或不相等。例如，近似相等几何操作符使用eqsel的前提是假定它们通常只匹配表中的一小部分项。

可以使用scalarltsel、scalarlesel、scalargtsel以及scalargesel来比较被转换为数字标量进行范围比较具有意义的数据类型。如果可能，增加一种能被src/backend/utils/adt/selfuncs.c中的函数convert_to_scalar()所理解的数据类型（最后，这个函数应该被通过ux_type系统目录的一列所标识的针对每个数据类型的函数所替换，但是那还没有发生）。如果没有这样做，还是能工作，但是优化器的估计将不会达到最好的效果。

有一些额外的选择度估算函数是为src/backend/utils/adt/geo_selfuncs.c中的几何操作符设计的：areasel、positionsel和contsel。在写这份材料时，这些还只是存根，但是可能想要使用它们（或者甚至改进它们）。

## 1.15.4. JOIN

如果提供了JOIN子句，表示用于该操作符的一个连接选择度估计函数（注意这是一个函数名而不是一个操作符名）。JOIN子句只对返回boolean的二元操作符有意义。一个连接选择度估算器背后的思想是猜测一对表中有多大比例的行对于当前的操作符将会满足一个如下形式的WHERE子句条件。和RESTRICT子句一样，这通过让优化器知道哪种连接序列需要做的工作最少来极大地帮助优化器。

table1.column1 OP table2.column2

这一章将不解释如何编写一个连接选择度估算函数，而只是建议在适当的时候使用一种标准估算器，如下所示。

eqjoinsel用于=
neqjoinsel用于<>
scalarltjoinsel用于<
scalarlejoinsel用于<=
scalargtjoinsel用于>
scalargejoinsel用于>=
areajoinsel用于基于 2D 区域比较
positionjoinsel用于基于 2D 位置比较
contjoinsel用于基于 2D 包含比较

## 1.15.5. HASHES

如果存在HASHES子句，它告诉系统它被许可为基于这个操作符的一个连接使用哈希连接方法。HASHES只对返回boolean的二元操作符有意义，并且实际上该操作符必须必须表达某种数据类型或数据类型对的相等。

哈希连接之下的假设是连接操作符只能对哈希到相同哈希码的左右值返回真。如果两个值被放到不同的哈希桶中，连接将根本不会比较它们，这隐式地假定该连接操作符的结果必须是假。因此，为不表示某种形式相等的操作符指定HASHES是没有意义的。在大部分情况下，只有为在两端都是相同数据类型的操作符支持哈希才有意义。不过，有时可以为两种或更多数据类型设计兼容的哈希函数，也就是说，对于“相等”的值（即使具有不同的表达）会产生相同哈希码的函数。例如，在哈希不同宽度的证书时，安排这个属性相当简单。

要被标记为HASHES，连接操作符必须出现在一个哈希索引操作符族中。当创建该操作符时这不会被强制，因为要引用的操作符族当然不可能已经存在。但是如果这样的操作符族不存在，尝试在哈希连接中使用该操作符将在运行时失败。系统需要用该操作符族来为操作符的输入数据类型寻找数据类型相关的哈希函数。当然，在创建操作符族之前，还必须创建合适的哈希函数。

在准备一个哈希函数时应当慎重，因为有一些方法是依赖于机器的，这样它可能无法做正确的事情。例如，如果数据类型是一个结构，其中可能有无用的填充位，不能简单地把整个结构传递给hash_any（除非编写自己的操作符和函数按照推荐的策略来保证未被使用的位总是为零）。另一个示例是在符合IEEE浮点标准的机器上，负数零和正数零是不同的值（不同的位模式），但是它们被定义为相等。如果一个浮点值可能包含负数零，那么需要额外的步骤来保证它产生的哈希值与正数零产生的相同。

一个可哈希连接的操作符必须拥有一个出现在同一操作符族中的交换子（如果两个操作数数据类型相同，那么就是它自身，否则是一个相关的相等操作符）。如果情况不是这样，在使用该操作符时，可能会发生规划器错误。此外，一个支持多种数据类型的哈希操作符族为数据类型的每一种组合都提供相等操作符是一个好主意（但是并不被严格要求），这会带来更好的优化。

> **注意**
>
> 一个可哈希连接的操作符底层的函数必须被标记为可交换或者稳定。如果它是不稳定的，系统将永远不会为一个哈希连接尝试使用该操作符。

> **注意**
>
> 如果一个可哈希连接的操作符有一个被标记为 strict 的底层函数，该函数必须也是 complete：也就是对于任意两个非空输入它应当返回真或假，从不会返回空。如果没有遵守这个规则，IN操作的哈希优化可能产生错误的结果（特别是，当依据标准的正确答案可能是空时，IN可能会返回假，或者它会产生一个错误来抱怨它没有准备会收到一个空结果）。

### 1.15.6. MERGES

如果存在MERGES子句，它告诉系统它被许可为基于这个操作符的一个连接使用归并连接方法。MERGES只对返回boolean的二元操作符有意义，并且实际上该操作符必须必须表达某种数据类型或数据类型对的相等。

归并连接的思想是排序左右手表并且接着并行扫描它们。这样，两种数据类型必须能够被完全排序，并且该连接操作符必须只为落在排序顺序上“同一位置”的值对返回成功。实际上这意味着该连接操作符必须和相等的行为一样。但是只要两种不同的数据类型在逻辑上是兼容的，就能对它们使用归并连接。例如，smallint-versus-integer相等操作符就是可归并连接的。只需要将两种数据类型变成逻辑上兼容的序列的排序操作符。

要被标记为MERGES，该连接操作符必须作为一个btree索引操作符的相等成员出现。当创建该操作符时，这不是强制的，因为要引用的操作符族当然可能还不存在。但是除非能找到一个匹配的操作符族，否则该操作符将不会被实际用于归并连接。MERGES标志因此扮演着一种对于规划器的提示，表示值得去寻找一个匹配的操作符族。

一个可归并连接的操作符必须拥有一个出现在同一操作符族中的交换子（如果两个操作数数据类型相同，那么就是它自身，否则是一个相关的相等操作符）。如果情况不是这样，在使用该操作符时，可能会发生规划器错误。此外，一个支持多种数据类型的btree操作符族为数据类型的每一种组合都提供相等操作符是一个好主意（但是并不被严格要求），这会带来更好的优化。

> **注意**
>
> 一个可归并连接的操作符底层的函数必须被标记为可交换或者稳定。如果它是不稳定的，系统将永远不会为一个归并连接尝试使用该操作符。

## 1.16. 索引的接口扩展

迄今为止已经描述的过程能够定义新的类型、新的函数以及新的操作符。但是，不能在一种新数据类型的列上定义索引。要做这件事情，必须为新数据类型定义一个操作符类。在这一小节稍后的部分，将用一个示例阐述这部份内容：一个用于 B-树索引方法的操作符类，它以绝对值的升序存储和排序复数。

操作符类可以被分组成操作符族来体现语义兼容的类之间的联系。当只涉及到一种单一数据类型时，一个操作符类就足矣。因此先把重点放在这种情况上，然后再回到操作符族。

## 1.16.1. 索引方法和操作符类

ux_am表为每一种索引方法都包含一行（内部被称为访问方法）。UXsinoDB中内建了对表常规访问的支持，但是所有的索引方法则是在ux_am中描述。可以通过编写必要的代码并且在ux_am中创建一项来增加一种新的索引访问方法 — 但这超出了本章的范围。

一个索引方法的例程并不直接了解它将要操作的数据类型。而是由一个操作符类标识索引方法用来操作一种特定数据类型的一组操作。之所以被称为操作符类是因为它们指定的一件事情就是可以被用于一个索引的WHERE子句操作符集合（即，能被转换成一个索引扫描条件）。一个操作符类也能指定一些索引方法内部操作所需的支持函数，这些过程不能直接对应于能用于索引的任何WHERE子句操作符。

可以为相同的数据类型和索引方法定义多个操作符类。通过这种方式，可以为一种数据类型定义多个索引语义集合。例如，一个B-树索引要求在它要操作的每一种数据类型上都定义一个排序顺序。对一种复数数据类型来说，拥有一个可以根据复数绝对值排序的 B-树操作符类和另一个可以根据实数部分排序的操作符类可能会有用。典型地，其中一个操作符类将被认为是最常用的并且将被标记为那种数据类型和索引方法的默认操作符类。

相同的操作符类名称可以被用于多个不同的索引方法（例如，B-树和哈希索引方法都有名为int4_ops的操作符类）。但是每一个这样的类都是一个独立实体并且必须被单独定义。

## 1.16.2. 索引方法策略

与一个操作符类关联的操作符通过“策略号”标识，它被用来标识每个操作符在其操作符类中的语义。例如，B-树在键上施行了一种严格的顺序（较小到较大），因此“小于”和“大于等于”这样的操作符就是 B-树所感兴趣的。因为UXsinoDB允许用户定义操作符，UXsinoDB不能看着一个操作符（如<和>=）的名字并且说出它是哪一种比较。取而代之的是，索引方法定义了一个“策略”集合，它们可以被看成是广义的操作符。每一个操作符类会说明对于一种特定的数据类型究竟是哪个实际的操作符对应于每一种策略以及该索引语义的解释。

B-树索引方法定义了五种策略，如表 1.2 “B-树策略”所示。

表 1.2. B-树策略

| 操作 | 策略号 |
| --- | --- |
| 小于 | 1 |
| 小于等于 | 2 |
| 等于 | 3 |
| 大于等于 | 4 |
| 大于 | 5 |

哈希索引只支持等值比较，因此它们只使用一种策略，如表 1.3 “哈希策略”所示。

表 1.3. 哈希策略

| 操作 | 策略号 |
| --- | --- |
| 等于 | 1 |

GiST 索引更加灵活：它们根本没有一个固定的策略集合。取而代之的是，每一个特定 GiST 操作符类的“consistency”支持例程会负责解释策略号。例如，一些内建的 GiST 索引操作符类

索引二维几何对象，它们提供表 1.4 “GiST 二维“R-树”策略”中所示的“R-树”策略。其中四个是真正的二维测试（重叠、相同、包含、被包含），其中四个只考虑 X 方向，其他四个提供 Y 方向上的相同测试。

表 1.4. GiST 二维“R-树”策略

| 操作 | 策略号 |
| --- | --- |
| 左参数严格地位于右参数的左边 | 1 |
| 左参数不会延伸到右参数的右边 | 2 |
| 重叠 | 3 |
| 左参数不会延伸到右参数的左边 | 4 |
| 左参数严格地位于右参数的右边 | 5 |
| 相同 | 6 |
| 包含 | 7 |
| 被包含 | 8 |
| 不会延伸到高于 | 9 |
| 严格低于 | 10 |
| 严格高于 | 11 |
| 不会延伸到低于 | 12 |

SP-GiST 索引在灵活性上与索引相似：它们没有一个固定的策略集合。取而代之的是，每一个操作符类的支持例程负责根据该操作符类的定义解释策略号。例如，被内建操作符类用于点的策略号如表 1.5 “SP-GiST 点策略”中所示。

表 1.5. SP-GiST 点策略

| 操作 | 策略号 |
| --- | --- |
| 左参数严格地位于右参数的左边 | 1 |
| 左参数严格地位于右参数的右边 | 5 |
| 相同 | 6 |
| 被包含 | 8 |
| 严格地低于 | 10 |
| 严格地高于 | 11 |

GIN 索引与 GiST 和 SP-GiST 索引类似，它们也没有一个固定的策略集合。取而代之的是，每一个操作符类的支持例程负责根据该操作符类的定义解释策略号。例如，被内建操作符类用于数组的策略号如表 1.6 “GIN 数组策略”所示。

表 1.6. GIN 数组策略

| 操作 | 策略号 |
| --- | --- |
| 重叠 | 1 |
| 包含 | 2 |
| 被包含 | 3 |
| 等于 | 4 |

在没有固定的策略集合这一点上，BRIN 索引和 GiST、SP-GiST 和 GIN 索引是类似的。每一个操作符类的支持函数会根据操作符类的定义解释策略编号。例如，表 1.7 “BRIN 最小最大策略”中展示了内建的Minmax操作符类所使用的策略编号。

表 1.7. BRIN 最小最大策略

| 操作 | 策略号 |
| --- | --- |
| 小于 | 1 |
| 小于等于 | 2 |
| 等于 | 3 |
| 大于等于 | 4 |
| 大于 | 5 |

注意上文列出的所有操作符都返回布尔值。实际上，所有作为索引方法搜索操作符定义的操作符必须返回类型boolean，因为它们必须出现在一个WHERE子句的顶层来与一个索引一起使用（某些索引访问方法还支持排序操作符，它们通常不返回布尔值，这种特性在第 1.16.7 节“排序操作符”中讨论）。

## 1.16.3. 索引方法支持例程

对于系统来说只有策略信息通常不足以断定如何使用一种索引。实际上，为了能工作，索引方法还要求额外的支持例程。例如，B-树索引方法必须能比较两个键并且决定其中一个是否大于、等于或小于另一个。类似地，哈希索引方法必须能够为键值计算哈希码。这些操作并不对应在 SQL 命令的条件中使用的操作符。它们是索引方法在内部使用的管理例程。

与策略一样，操作符类会标识哪些函数应该为一种给定的数据类型扮演这些角色以及相应的语义解释。索引方法定义它需要的函数集合，而操作符类则会通过为函数分配由索引方法说明的“支持函数号”来标识正确的函数。

如表 1.8 “B-树支持函数”所示， B-树要求一个比较支持函数，并且允许在操作符类作者的选项中提供两个额外的支持函数。

表 1.8. B-树支持函数

| 函数 | 支持号 |
| --- | --- |
| 比较两个键并且返回一个小于零、等于零或大于零的整数，它表示第一个键小于、等于或者大于第二个键。 | 1 |
| 返回C可调用的排序支持函数的地址（可选）。 | 2 |
| 将一个测试值与一个基础值加上/减去一个偏移量的结果进行比较，根据比较的结果返回真或假（可选） | 3 |

如表 1.9 “哈希支持函数”所示，哈希索引要求一个支持函数，并且允许在操作符类作者的选项中提供第二个支持函数。

表 1.9. 哈希支持函数

| 函数 | 支持号 |
| --- | --- |
| 为一个键计算32位哈希值 | 1 |

| 函数 | 支持号 |
| --- | --- |
| 给定一个64-位salt，计算一个键的64位哈希值。如果salt为0，结果的低32位必须匹配会由函数1计算出来的值（可选） | 2 |

如表 1.10 “GiST 支持函数”所示，GiST 索引有九个支持函数，其中两个是可选的。

表 1.10. GiST 支持函数

| 函数 | 描述 | 支持号 |
| --- | --- | --- |
| consistent | 判断键是否满足查询修饰语 | 1 |
| union | 计算一个键集合的联合 | 2 |
| compress | 计算一个要被索引的键或值的压缩表达 | 3 |
| decompress | 计算一个压缩键的解压表达 | 4 |
| penalty | 计算把新键插入到带有给定子树键的子树中带来的罚值 | 5 |
| picksplit | 判断一个页面中的哪些项要被移动到新页面中并且计算结果页面的联合键 | 6 |
| equal | 比较两个键并且在它们相等时返回真 | 7 |
| distance | 判断键到查询值的距离（可选） | 8 |
| fetch | 为只用索引扫描计算一个压缩键的原始表达（可选） | 9 |

如表 1.11 “SP-GiST 支持函数”所示，SP-GiST 索引要求五个支持函数。

表 1.11. SP-GiST 支持函数

| 函数 | 描述 | 支持号 |
| --- | --- | --- |
| config | 提供有关该操作符类的基本信息 | 1 |
| choose | 判断如何把一个新值插入到一个内元组中 | 2 |
| picksplit | 判断如何划分一组值 | 3 |
| inner_consistent | 判断对于一个查询需要搜索哪一个子划分 | 4 |
| leaf_consistent | 判断键是否满足查询修饰语 | 5 |

如表 1.12 “GIN 支持函数”所示，GIN 索引有六个支持函数，其中三个是可选的。

表 1.12. GIN 支持函数

| 函数 | 描述 | 支持号 |
| --- | --- | --- |
| compare | 比较两个键并且返回一个小于零、等于零或大于零的整数， | 1 |

| 函数 | 描述 | 支持号 |
| --- | --- | --- |
| | 它表示第一个键小于、等于或者大于第二个键 | |
| extractValue | 从一个要被索引的值中抽取键 | 2 |
| extractQuery | 从一个查询条件中抽取键 | 3 |
| consistent | 判断值是否匹配查询条件（布尔变体）（如果支持函数 6 存在则是可选的） | 4 |
| comparePartial | 比较来自查询的部分键和来自索引的键，并且返回一个小于零、等于零或大于零的整数，表示 GIN 是否应该忽略该索引项、把该项当做一个匹配或者停止索引扫描（可选） | 5 |
| triConsistent | 判断值是否匹配查询条件（三元变体）（如果支持函数 4 存在则是可选的） | 6 |

如表 1.13 “BRIN 支持函数”中所示，BRIN 索引具有四个基本的支持函数。这些基本函数可能会要求提供额外的支持函数。

表 1.13. BRIN 支持函数

| 函数 | 描述 | 支持编号 |
| --- | --- | --- |
| opcInfo | 返回描述被索引列的摘要数据的内部信息 | 1 |
| add_value | 向一个现有的摘要索引元组增加一个新值 | 2 |
| consistent | 判断值是否匹配查询条件 | 3 |
| union | 计算两个摘要元组的联合 | 4 |

和搜索操作符不同，支持函数返回特定索引方法所期望的数据类型，例如在 B 树的比较函数中是一个有符号整数。每个支持函数的参数数量和类型也取决于索引方法。对于 B 树和哈希，比较和哈希支持函数和包括在操作符类中的操作符接收一样的输入数据类型，但是大部分 GiST、SP-GiST、GIN 和 BRIN 支持函数则不是这样。

## 1.16.4. 示例

现在已经看过了基本思想，这里是创建一个新操作符类的示例（可以在源代码的src/tutorial/complex.c和src/tutorial/complex.sql中找到这个示例）。该操作符类封装了以绝对值顺序排序复数的操作符，因此为它取名为complex_abs_ops。首先，需要一个操作符集合。定义操作符的过程已经在第 1.14 节 “用户定义的操作符”中讨论过。对于一个 B-树上的操作符类，需要的操作符如下所示。

- 绝对值小于（策略 1）
- 绝对值小于等于（策略 2）
- 绝对值等于（策略 3）
- 绝对值大于等于（策略 4）
- 绝对值大于（策略 5）

定义一个比较操作符的相关集合最不容易出错的方式是，先编写 B-树比较支持函数，然后编写该支持函数的包装器函数。这降低了极端情况下得到不一致结果的几率。遵照这种方法，进行如下编写。

```
#define Mag(c)  ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
   double      amag = Mag(a),
               bmag = Mag(b);

   if (amag < bmag)
      return -1;
   if (amag > bmag)
      return 1;
   return 0;
}
```

现在小于函数，如下所示。

```
UX_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(UX_FUNCTION_ARGS)
{
   Complex    *a = (Complex *) UX_GETARG_POINTER(0);
   Complex    *b = (Complex *) UX_GETARG_POINTER(1);

   UX_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

其他四个函数的区别只在于它们如何比较内部函数的结果与 0。

接下来基于这些函数声明 SQL 的函数和操作符，如下所示。

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
   AS 'filename', 'complex_abs_lt'
   LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
  leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
  commutator = > , negator = >= ,
  restrict = scalarltsel, join = scalarltjoinsel
);
```

指定正确的交换子和求反器操作符很重要，合适的限制和连接选择度函数也是一样，否则优化器将无法有效地利用索引。

其他值得注意的事情，如下所示。

- 只能有一个操作符被命名为=且两个操作数都为类型complex。在这种要求下，对于complex没有任何其他操作符=。但是如果在构建一种实际的数据类型，可能想让=成为复数的普通等值操作（不是绝对值的相等）。这样，需要为complex_abs_eq使用某种其他的操作符名称。
- 尽管UXsinoDB能够处理具有相同 SQL 名称的函数（只要它们具有不同的参数数据类型），但 C 只能处理具有给定名称一个全局函数。因此，不能简单地把 C 函数命名为abs_eq之类的东西。通常，在 C 函数名中包括数据类型的名称是一种好习惯，这样就不会与其他数据类型的函数发生冲突。
- 让函数也具有abs_eq这样的 SQL 名称，而依靠UXsinoDB通过参数数据类型来区分它和其他同名 SQL 函数。为了保持示例的简洁，让 C 级别和 SQL 级别的函数具有相同的名称。

下一步是注册 B-树所要求的支持例程。实现支持例程的 C 代码示例在包含操作符函数的同一文件中，如下所示，声明该函数。

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

现在已经有了所需的操作符和支持例程，就可以最终创建操作符类，如下所示。

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

做好了！现在应该可以在complex列上创建并且使用 B-树索引了。

可以把操作符项写得更繁琐，如下所示。

```
        OPERATOR        1       < (complex, complex) ,
```

但是当操作符操作的数据类型和正在定义的操作符类所服务的数据类型相同时可以不用这么做。

上述示例假定这个新操作符类是complex数据类型的默认 B-树操作符类。如果不是这样，只需要省去关键词DEFAULT。

## 1.16.5. 操作符类和操作符族

到目前为止，暗地里假设一个操作符类只处理一种数据类型。虽然在一个特定的索引列中必定只有一种数据类型，但是把被索引列与一种不同数据类型的值比较的索引操作通常也很有用。还有，如果与一种操作符类相关的扩数据类型操作符有用，通常情况是其他数据类型也有其自身相关的操作符类。在相关的类之间建立起明确的联系会很有用，因为这可以帮助规划器进行 SQL 查询优化（尤其是对于 B-树操作符类，因为规划器包含了大量有关如何使用它们的知识）。

为了处理这些需求，UXsinoDB使用了操作符族的概念。一个操作符族包含一个或者多个操作符类，并且也能包含属于该族整体而不属于该族中任何单一类的可索引操作符和相应的支持函数。

这样的操作符和函数是“松散地”存在于该族中，而不是被绑定在一个特定的类中。通常每个操作符类包含单一数据类型的操作符，而跨数据类型操作符则松散地存在于操作符族中。

一个操作符族中的所有操作符和函数必须具有兼容的语义，其中的兼容性要求由索引方法设定。可能因此而奇怪为什么要这么麻烦地把族的特定子集单另出来成为操作符类，并且实际上（由于很多原因）这种划分与操作符之间没有什么直接的关联，只有操作符族才是实际的分组。定义操作符类的原因是，它们指定了特定索引对操作符族的依赖程度。如果一个索引使用着一个操作符类，那么不删除该索引是不能删除该操作符类的 — 但是操作符族的其他部分（即其他操作符类和松散的操作符）可以被删除。因此，一个操作符类应该包含一个索引在特定数据类型上正常工作所需要的最小操作符和函数集合，而相关但不关键的操作符可以作为操作符族的松散成员被加入。

例如，UXsinoDB有一个内建的 B-树操作符族integer_ops，它包括分别用于类型bigint(int8)、integer (int4)和smallint (int2)列上索引的操作符类int8_ops、int4_ops以及int2_ops。这个族也包含跨数据类型比较操作符，它们允许对这些类型中的任意两种进行比较，这样可以通过一种类型的比较值来搜索另一种类型之上的索引。这个族可以用这些定义来重现，如下所示。

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
  -- 标准 int8 比较
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ,
  FUNCTION 2 btint8sortsupport(internal) ,
  FUNCTION 3 in_range(int8, int8, int8, boolean, boolean) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
  -- 标准 int4 比较
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint4cmp(int4, int4) ,
  FUNCTION 2 btint4sortsupport(internal) ,
  FUNCTION 3 in_range(int4, int4, int4, boolean, boolean) ;

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
  -- 标准 int2 比较
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint2cmp(int2, int2) ,
```

```
 FUNCTION 2 btint2sortsupport(internal) ,
 FUNCTION 3 in_range(int2, int2, int2, boolean, boolean) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
 -- 跨类型比较 int8 vs int2
 OPERATOR 1 < (int8, int2) ,
 OPERATOR 2 <= (int8, int2) ,
 OPERATOR 3 = (int8, int2) ,
 OPERATOR 4 >= (int8, int2) ,
 OPERATOR 5 > (int8, int2) ,
 FUNCTION 1 btint82cmp(int8, int2) ,

 -- 跨类型比较 int8 vs int4
 OPERATOR 1 < (int8, int4) ,
 OPERATOR 2 <= (int8, int4) ,
 OPERATOR 3 = (int8, int4) ,
 OPERATOR 4 >= (int8, int4) ,
 OPERATOR 5 > (int8, int4) ,
 FUNCTION 1 btint84cmp(int8, int4) ,

 -- 跨类型比较 int4 vs int2
 OPERATOR 1 < (int4, int2) ,
 OPERATOR 2 <= (int4, int2) ,
 OPERATOR 3 = (int4, int2) ,
 OPERATOR 4 >= (int4, int2) ,
 OPERATOR 5 > (int4, int2) ,
 FUNCTION 1 btint42cmp(int4, int2) ,

 -- 跨类型比较 int4 vs int8
 OPERATOR 1 < (int4, int8) ,
 OPERATOR 2 <= (int4, int8) ,
 OPERATOR 3 = (int4, int8) ,
 OPERATOR 4 >= (int4, int8) ,
 OPERATOR 5 > (int4, int8) ,
 FUNCTION 1 btint48cmp(int4, int8) ,

 -- 跨类型比较 int2 vs int8
 OPERATOR 1 < (int2, int8) ,
 OPERATOR 2 <= (int2, int8) ,
 OPERATOR 3 = (int2, int8) ,
 OPERATOR 4 >= (int2, int8) ,
 OPERATOR 5 > (int2, int8) ,
 FUNCTION 1 btint28cmp(int2, int8) ,

 -- 跨类型比较 int2 vs int4
 OPERATOR 1 < (int2, int4) ,
 OPERATOR 2 <= (int2, int4) ,
 OPERATOR 3 = (int2, int4) ,
 OPERATOR 4 >= (int2, int4) ,
 OPERATOR 5 > (int2, int4) ,
 FUNCTION 1 btint24cmp(int2, int4) ,

 -- 跨类型的in_range函数
 FUNCTION 3 in_range(int4, int4, int8, boolean, boolean) ,
```

FUNCTION 3 in_range(int4, int4, int2, boolean, boolean) ,
FUNCTION 3 in_range(int2, int2, int8, boolean, boolean) ,
FUNCTION 3 in_range(int2, int2, int4, boolean, boolean) ;

注意这种定义“重载”了操作符策略和支持函数号：每一个编号在该族中出现多次。只要一个特定编号的每一个实例都有可区分的输入数据类型，就允许这样做。输入类型等于操作符类输入类型的实例是该操作符类的主要操作符和支持函数，并且在大部分情况下应该被声明为该操作符类的一部分而不是作为操作符族的松散成员存在。

在一个B-树操作符族中，所有该族中的操作符必须以兼容的方式排序。对该族中的每一个操作符都必须有一个与该操作符具有相同的两个输入数据类型的支持函数。推荐让操作符族保持完整，即对每一种数据类型的组合都应该包括所有的操作符。每个操作符类只应该包括非跨类型操作符和用于其数据类型的支持函数。

为了构建一个多数据类型的哈希操作符族，必须为该族支持的每一种数据类型创建相兼容的哈希支持函数。这里的兼容性是指这些函数对于任意两个被该族中等值操作符认为相等的值会保证返回相同的哈希码，即便这些值具有不同的类型时也是如此。当这些类型具有不同的物理表示时，这通常难以实现，但是在某些情况下是可以做到的。此外，将该操作符族中一种数据类型的值通过隐式或者二进制强制造型转换成该族中另一种数据类型时，不应该改变所计算出的哈希值。注意每种数据类型只有一个支持函数，而不是每个等值操作符一个。推荐让操作符族保持完整，即对每一种数据类型的组合提供一个等值操作符。每个操作符类只应该包括非跨类型等值操作符和用于其数据类型的支持函数。

GiST、SP-GiST 和 GIN 索引没有任何明显的跨数据类型操作的概念。它们所支持的操作符集合就是一个给定操作符类能够处理的主要支持函数。

在 BRIN 中，需求取决于提供操作符类的框架。对于基于minmax的操作符类，必要的行为和B-树操作符族相同：族中的所有操作符必须以兼容的方式排序，并且转换不能改变相关的排序顺序。

## 1.16.6. 操作符类上的系统依赖

UXsinoDB使用操作符类来以更多方式推断操作符的属性，而不仅仅是它们是否能被用于索引。因此，即便不准备对数据类型的列建立索引，也可能想要创建操作符类。

特别地，ORDER BY和DISTINCT等 SQL 特性要求对值的比较和排序。为了在用户定义的数据类型上实现这些特性，UXsinoDB会为数据类型查找默认 B-树操作符类。这个操作符类的“equals”成员定义了用于GROUP BY和DISTINCT的值的等值概念，而该操作符类施加的排序顺序定义了默认的ORDER BY顺序。

如果一种数据类型没有默认的 B-树操作符类，系统将查找默认的哈希操作符类。但由于这类操作符类只提供等值，所以它只能支持分组而不能支持排序。

在一种数据类型没有默认操作符类时，如果尝试对该数据类型使用这些 SQL 特性，将得到类似“could not identify an ordering operator”（无法标识排序操作符）的错误。

通过在一个USING选项中指定一个非默认B-树操作符类的小于操作符，可以使用该操作符进行排序，如下所示。

SELECT * FROM mytable ORDER BY somecol USING ~<~;

或者，在USING中指定该操作符类的大于操作符可以选择升序的排序。

用户定义类型的数组的比较还依赖于该类型的默认B-树操作符类所定义的语义。如果没有默认的B-树操作符类，但有一个默认的哈希操作符类，则支持数组的相等比较，但不支持顺序的比较。

另一种要求更多数据类型相关知识的SQL特性是窗口函数的RANGE *offset* PRECEDING/FOLLOWING帧选项。对于如下查询不足以了解如何用x进行排序，数据库还必须理解如何对当前行的x值“减5”或者“加10”以标识当前窗口帧的边界。

```
SELECT sum(x) OVER (ORDER BY x RANGE BETWEEN 5 PRECEDING AND 10 FOLLOWING)
  FROM mytable;
```

把得到的边界与其他行的x值用B-树操作符类提供的比较操作符（定义了ORDER BY顺序）进行比较是可能的 — 但是加和减操作符并不是该操作符类的一部分，因此应该用哪些操作符呢？硬编码的选择是不切实际的，因为不同的排序顺序（不同的B-树操作符）可能需要不同的行为。因此，一个B-树操作符类可以指定一个in_range支持函数，它封装有对排序顺序有意义的加和减行为。如果有多种数据类型可以用作RANGE子句中的偏移量，甚至可以提供多个in_range支持函数。如果与窗口的ORDER BY子句关联的B-树操作符类没有一个匹配的in_range支持函数，则不支持RANGE *offset* PRECEDING/FOLLOWING选项。

另一个要点是，出现在一个哈希操作符族中的操作符是哈希连接、哈希聚集和相关优化的候选。这些情况下哈希操作符族就是至关重要的，因为它标识了要使用的哈希函数。

## 1.16.7. 排序操作符

有些索引访问方法（当前只有 GiST和SP-GiST）支持排序操作符的概念。到目前为止所讨论的都是*搜索*操作符。搜索索引时，会用搜索操作符来寻找所有满足 WHERE *indexed_column operator constant* 的行。注意被返回的匹配行的顺序是没有任何保证的。相反，一个排序操作符并不限制能被返回的行集合，而是决定它们的顺序。扫描索引时，会使用排序操作符来以 ORDER BY *indexed_column operator constant* 所表示的顺序返回行。这样定义排序操作符的原因是，如果该操作符能度量距离，它就能支持最近邻搜索。例如，如下查询寻找离一个给定目标点最近的十个位置。位置列上的 GiST 索引可以有效地完成这个查询，因为<->是一个排序操作符。

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

搜索操作符必须返回布尔结果，排序操作符通常返回某种其他类型，例如浮点、数字或者距离。这种类型通常不同于被索引的数据类型。为了避免硬编码有关不同数据类型行为的假设，需要定义一个排序操作符来提名一个 B-树操作符族指定结果数据类型的排序顺序。正如在前一节介绍的，B-树操作符族定义了UXsinoDB的顺序概念，因此这是一种自然的表达。由于点<->操作符返回float8，可以在一个操作符类创建命令中这样指定它。

```
OPERATOR 15   <-> (point, point) FOR ORDER BY float_ops
```

其中float_ops是包括float8上操作的内建操作符族。这种声明说明该索引能够以<->操作符的递增值顺序返回行。

## 1.16.8. 操作符类的特性

有两个操作符类的特性还没有讨论，主要是因为它们对于最常用的索引方法不太有用。

通常，把一个操作符声明为一个操作符类（或操作符族）的成员意味着该索引方法能够使用该操作符准确地检索满足WHERE条件的行集。如下所示。

```
SELECT * FROM table WHERE integer_column < 4;
```

恰好可以被该整数列上一个 B-树索引满足。但是也有情况下索引只是作为匹配行的非精确向导。例如，如果一个 GiST 索引只存储几何对象的边界框，那么它无法精确地满足测试非矩形对象（如多边形）之间相交的WHERE条件。但是可以使用该索引来寻找边界框与目标对象的边界框相交的对象，并且只在通过该索引找到的对象上做精确的相交测试。如果适用于这种场景，该索引被称为对该操作符是“有损的”。有损索引搜索通过在一行可能满足或者不满足该查询条件时返回一个recheck标志来实现。核心系统将接着在检索到的行上测试原始查询条件来看它是否应该被作为一个合法匹配返回。如果索引被保证能返回所有所需的行外加一些额外的行，这种方法就能有效，因为那些额外的行可以通过执行原始的操作符调用来消除。支持有损搜索的索引方法（当前有 GiST、SP-GiST 和 GIN）允许个别操作符类的支持函数设置 recheck 标志，因此这也是一种操作符类的重要特性。

再次考虑在索引中只存储复杂对象（如多边形）的边界框的情况。在这种情况下，把整个多边形存储在索引项中没有很大价值 — 也可以只存储一个更简单的box类型对象。这种情况通过CREATE OPERATOR CLASS中的STORAGE选项表示，如下所示。

```
CREATE OPERATOR CLASS polygon_ops
  DEFAULT FOR TYPE polygon USING gist AS
    ...
    STORAGE box;
```

当前，只有 GiST、GIN 和 BRIN 索引方法支持不同于列数据类型的STORAGE类型。在使用STORAGE时，GiST 的支持例程compress和decompress必须处理数据类型转换。在 GIN 中，STORAGE类型标识“key”值的类型，它通常不同于被索引列的类型 — 例如，一个用于整数数组列的操作符类可能具有整数键值。GIN 的支持例程extractValue和extractQuery负责从被索引值中抽取键。BRIN 类似于 GIN：STORAGE类型标识被存储的摘要值的类型，而操作符类的支持过程负责正确解释摘要值。

# 1.17. 打包相关对象到一个扩展中

一个对UXsinoDB有用的扩展通常包括多个 SQL 对象，例如，一种新的数据类型将需要新函数、新操作符以及可能的新索引操作符类。将所有这些对象收集到一个单一包中有助于简化数据库管理。UXsinoDB称这样一个包为一个扩展。要定义一个扩展，至少需要一个包含创建该扩展的对象的SQL命令的脚本文件以及一个指定扩展本身的一些基本属性的控制文件。如果扩展包括 C 代码，通常还有一个 C 代码编译而成的共享库文件。一旦有了这些文件，一个简单的CREATE EXTENSION命令可以把这些对象载入到数据库。

使用一个扩展而不是只运行SQL脚本载入一堆“松散”对象到数据库的主要优点是，UXsinoDB将能理解该扩展的对象是一起的。可以用一个单一的DROP EXTENSION命令删除所有的对象（不用维护一个单独的“卸载”脚本）。 甚至更有用的一点是，ux_dump知道它不应该转储该扩展中的个体成员对象—它将只在转储中包括一个CREATE EXTENSION命令。这大大简化了迁移到一个包含不同于旧版扩展中对象的新版扩展的工作。不过，注意在把这样一个转储载入到一个新数据库时，该扩展的控制、脚本和其他文件必须可用。

UXsinoDB不会删除包含在一个扩展中的个体对象，除非删除整个扩展。还有，虽然能够改变一个扩展的成员对象的定义（例如，通过CREATE OR REPLACE FUNCTION改变一个函数），记住被修改后的定义将不会被ux_dump转储。这种改变通常只有并发地在扩展脚本文件中做出相同更改时才有意义（但是对于包含配置数据的表有特殊的规定，见第 1.17.4 节 “扩展配置表”。在生产环境中，通常更好的方式是创建一个扩展更新脚本来执行对扩展中成员对象的更改。

扩展脚本可能会通过GRANT和REVOKE语句设置扩展中所含对象的特权。每一个对象的最终特权集合（如果设置了任何特权）将被存储在ux_init_privs系统目录中。使用ux_dump时，CREATE EXTENSION命令将被包括在转储中，后面会跟着必要的GRANT和REVOKE语句集合来将对象的特权设置成取得该转储时的样子。

UXsinoDB当前不支持扩展脚本发出CREATE POLICY或者SECURITY LABEL语句。这些东西的设置应该在扩展被创建好之后来进行。所有在扩展对象上创建的 RLS 策略和安全标签都将被包括在ux_dump创建的转储中。

扩展机制也对打包调整一个扩展中所含 SQL 对象定义的修改脚本有规定。例如，如果一个扩展的 1.1 版本比 1.0 版本增加了一个函数并且更改了另一个函数的函数体，该扩展的作者可以提供一个*更新脚本*来做这两个更改。那么ALTER EXTENSION UPDATE命令可以被用来应用这些更改并且跟踪在给定数据库中实际安装的是该扩展的哪个版本。

能作为一个扩展的成员的 SQL 对象的种类如ALTER EXTENSION所示。尤其是数据库集簇范围的对象（例如数据库、角色和表空间）不能作为扩展成员，因为一个扩展只在一个数据库范围内可见（尽管一个扩展脚本并没有被禁止创建这些对象，但是这样做将无法把它们作为扩展的一部分来跟踪）。还要注意虽然一个表可以是一个扩展的成员，它的扶助对象（例如索引）不会被直接认为是该扩展的成员。另一个重点是模式可以属于扩展，但是反过来不行：一个扩展本身有一个不被限定的名称并且不存在于任何模式“中”。不过，扩展的成员对象只要对象类型合适就可以属于模式。一个扩展拥有包含其成员对象的模式可能合适也可能不合适。

如果一个扩展的脚本创建任何临时对象（例如临时表），在当前会话的剩余部分会把它们当作扩展的成员，但是在会话结束会自动删除它们，这和任何其他临时对象是一样的。对于不删除整个扩展就不能删除扩展的成员对象的规则来说，这是一种例外。

## 1.17.1. 定义扩展对象

广泛分发的扩展应该尽量少地假定它们所占据的数据库。特别是，除非发出了SET search_path = ux_temp，应该假定每一个未限定的名称都可能解析成恶意用户定义的对象。要小心隐式依赖于search_path的结构：IN以及CASE *expression* WHEN总是使用搜索路径选择操作符。对于它们，可使用OPERATOR(*schema*.=) ANY和CASE WHEN *expression*。

## 1.17.2. 扩展文件

CREATE EXTENSION命令依赖每一个扩展都有的控制文件，控制文件必须被命名为扩展的名称加上一个后缀.control，并且必须被放在安装的SHAREDIR/extension目录中。还必须至少有一个SQL脚本文件，它遵循命名模式*extension*--*version*.sql（例如，foo--1.0.sql表示扩展foo的1.0版本）。默认情况下，脚本文件也被放置在SHAREDIR/extension目录中，但是控制文件中可以为脚本文件指定一个不同的目录。

一个扩展控制文件的格式与uxsinodb.conf文件相同，即是一个*parameter_name* = *value*赋值的列表，每行一个。允许空行和#引入的注释。注意对任何不是单一词或数字的值加上引号。

一个控制文件可以设置下列参数，如下所示。

- directory (string)

  包含扩展的SQL脚本文件的目录。除非给出一个绝对路径，这个目录名是相对于安装的SHAREDIR目录。默认行为等效于指定directory = 'extension'。

- default_version (string)

  该扩展的默认版本（就是如果在CREATE EXTENSION中没有指定版本时将会被安装的那一个）。尽管可以忽略这个参数，但如果没有出现VERSION选项时那将会导致CREATE EXTENSION失败，因此通常不建议这么做。

- comment (string)

一个关于该扩展的注释（任意字符串）。该注释会在初始创建扩展时应用，但是扩展更新时不会引用该注释（因为可能会覆盖用户增加的注释）。扩展的注释也可以通过在脚本文件中写上COMMENT命令来设置。

- encoding（string）

  该脚本文件使用的字符集编码。当脚本文件包含任何非 ASCII 字符时，可以指定这个参数。否则文件都会被假定为数据库编码。

- module_pathname（string）

  这个参数的值将被用于替换脚本文件中每一次出现的MODULE_PATHNAME。如果设置，将不会进行替换。通常，这会被设置为$libdir/*shared_library_name*并且接着MODULE_PATHNAME被用在CREATE FUNCTION命令中进行 C-语言函数的创建，因此该脚本文件不必把共享库的名称硬编码在其中。

- requires（string）

  这个扩展依赖的其他扩展名的一个列表，例如requires = 'foo, bar'。被依赖的扩展必须先于这个扩展安装。

- superuser（boolean）

  如果这个参数为true（默认情况），只有超级用户能够创建该扩展或者将它更新到一个新版本。如果被设置为false，只需要用来执行安装中命令或者更新脚本的特权。

- relocatable（boolean）

  如果一个扩展可能在初始创建之后将其所含的对象移动到一个不同的模式中，它就是relocatable。默认值是false，即该扩展是不可重定位的。详见第 1.17.3 节 “扩展可再定位性”。

- schema（string）

  这个参数只能为非可重定位扩展设置。它强制扩展被载入到给定的模式中而非其他模式中。只有在初始创建一个扩展时才会参考schema参数，扩展更新时则不会参考这个参数。详见第 1.17.3 节 “扩展可再定位性”

除了主要控制文件*extension*.control，一个扩展还可以有二级控制文件，它们以*extension*--*version*.control的风格命名。如果提供了二级控制文件，它们必须被放置在脚本文件的目录中。二级控制文件遵循主要控制文件相同的格式。在安装或更新该扩展的版本时，一个二级控制文件中设置的任何参数将覆盖主要控制文件中的设置。不过，参数directory以及default_version不能在二级控制文件中设置。

一个扩展的SQL脚本文件能够包含任何 SQL 命令，除了事务控制命令 （BEGIN、COMMIT等）以及不能在一个事务块中执行的命令（如VACUUM）。这是因为脚本文件会被隐式地在一个事务块中被执行。

一个扩展的SQL脚本文件也能包含以\echo开始的行，它将被扩展机制忽略（当作注释）。如果脚本文件被送给uxsql而不是由CREATE EXTENSION载入（见第 1.17.7 节 “扩展实例”中的示例脚本），这种机制通常被用来抛出错误。如果没有这种功能，用户可能会意外地把该扩展的内容作为“松散的”对象而不是一整个扩展载入，这样的状态恢复起来比较麻烦。

尽管脚本文件可以包含指定编码允许的任何字符，但是控制文件应该只包含纯 ASCII 字符，因为UXsinoDB没有办法知道一个控制文件是什么编码。实际上，如果想在扩展的注释中使用

非 ASCII 字符只有一个问题。推荐的方法是不使用控制文件的comment参数，而是使用脚本文件中的COMMENT ON EXTENSION来设置注释。

## 1.17.3. 扩展可再定位性

用户常常希望把扩展中包含的对象载入到一个与扩展的作者所设想的不一样的模式中。对于这种可重定位性，有三种支持的级别，如下所示。

- 一个完全可重定位的扩展能在任何时候被移动到另一个模式中，即使在它被载入到一个数据库中之后。这种移动通过ALTER EXTENSION SET SCHEMA命令完成，该命令会自动地把所有成员对象重命名到新的模式中。通常，只有扩展不包含任何对其所在模式的内部假设时才可能这样做。还有，该扩展的对象必须全部在同一个模式中（忽略那些不属于任何模式的对象，例如过程语言）。要让一个扩展变成完全可定位，在它的控制文件中设置relocatable = true。

- 一个扩展可能在安装过程中是可重定位的，但是安装完后就不再可重定位。典型的情况是扩展的脚本文件需要显式地引用目标模式，例如为 SQL 函数设置search_path属性。对于这样一种扩展，在其控制文件中设置relocatable = false，并且使用@extschema@在脚本文件中引用目标模式。在脚本被执行前，所有这个字符串的出现都将被替换为实际的目标模式名。用户可以使用CREATE EXTENSION的SCHEMA选项设置目标模式名。

- 如果扩展根本就不支持重定位，在它的控制文件中设置relocatable = false，并且还设置schema为想要的目标模式的名称。这将阻止使用CREATE EXTENSION的SCHEMA选项修改目标模式，除非它指定的是和控制文件中相同的模式。如果该扩展包括关于模式名的内部假设且模式名不能使用@extschema@的方法替换，这种选择通常是必须的。@extschema@替换机制在这种情况中也是可用的，不过由于模式名已经被控制文件所决定，它的使用受到了很大的限制。

在所有情况下，脚本文件将被用search_path执行，它最初会被设置为指向目标模式，也就是说CREATE EXTENSION做的也是等效的工作。

SET LOCAL search_path TO @extschema@;

这允许由这个脚本文件创建的对象进入到目标模式中。如果脚本文件希望，它可以改变search_path，但这种用法通常是不受欢迎的。在CREATE EXTENSION结束后，search_path会被恢复到之前的设置。

如果控制文件中给出了schema参数，目标模式就由该参数决定，否则目标模式由CREATE EXTENSION的SCHEMA选项给出，如果以上两者都没有给出则会用当前默认的对象创建模式（在调用者search_path中的第一个）。当使用扩展文件的schema参数时，如果目标模式还不存在将创建它，但是在另外两种情况下它必须已经存在。

如果在控制文件中的requires中列举了任何先导扩展，它们的目标模式会被追加到search_path的初始设置中。这允许新扩展的脚本文件能够看到它们的对象。

尽管一个不可重定位的扩展能够包含散布在多个模式中的对象，通常还是值得将意图用于外部使用的所有对象放置在一个模式中，这被认为是该扩展的目标模式。这样一种安排可以在依赖的扩展创建过程中方便地与search_path的默认设置一起工作。

## 1.17.4. 扩展配置表

某些扩展包括配置表，其中包含可以由用户在扩展安装后增加或修改的数据。通常，如果一个表是一个扩展的一部分，该表的定义和内容都不会被ux_dump转储。但是对于一个配置表来说并不

希望是这样的行为，任何用户做出的数据修改都需要被包括在转储中，否则该扩展在重载之后的行为将和转储之前不同。

要解决这个问题，一个扩展的脚本文件可以把一个它创建的表或者序列标记为配置关系，这将导致ux_dump把该表或者序列的内容（而不是它的定义）包括在转储中。要这样做，在创建表或序列之后调用函数ux_extension_config_dump(regclass, text)，如下所示。

```
CREATE TABLE my_config (key text, value text);
CREATE SEQUENCE my_config_seq;

SELECT ux_catalog.ux_extension_config_dump('my_config', '');
SELECT ux_catalog.ux_extension_config_dump('my_config_seq', '');
```

可以用这种方法标记任意数量的表或者序列。与serial或者bigserial列相关联的序列也可以被标记。

当ux_extension_config_dump的第二个参数是一个空字符串时，该表的全部内容都会被ux_dump转储。这通常只有在表被扩展脚本创建为初始为空时才正确。如果在表中混合有初始数据和用户提供的数据，ux_extension_config_dump的第二个参数提供了一种WHERE条件来选择要被转储的数据。例如，可能会有如下操作，并且确保只有扩展脚本创建的行中*standard_entry*才为真。

```
CREATE TABLE my_config (key text, value text, standard_entry boolean);

SELECT ux_catalog.ux_extension_config_dump('my_config', 'WHERE NOT standard_entry');
```

对于序列，ux_extension_config_dump的第二个参数没有影响。

更复杂的情况（例如用户可能会修改初始提供的数据）可以通过在配置表上创建触发器来处理，触发器将负责保证被修改的行会被正确地标记。

可通过再次调用ux_extension_config_dump来修改与一个配置表相关的过滤条件（这通常对于一个扩展更新脚本有用）。将一个表标记为不再是一个配置表的方法是用ALTER EXTENSION ... DROP TABLE将它与扩展脱离开。

注意这些表之间的外键关系将会指导这些表被 ux_dump 转储的顺序。特别地，ux_dump 将尝试先转储被引用的表再转储引用表。由于外键关系是在 CREATE EXTENSION 时间（先于数据被载入到表中）建立的，环状依赖还没有建立。当环状依赖存在时，数据将仍然被转储，但是该转储无法被 直接恢复并且必须要用户的介入。

与serial或者bigserial列相关联的序列需要被直接标记以转储它们的状态。只标记它们的父关系不足以转储它们的状态。

## 1.17.5. 扩展更新

扩展机制的一个优点是它提供了方便的方法来管理那些定义扩展中对象的 SQL 命令的更新。这是通过为扩展的安装脚本的每一个发行版本关联一个版本名称或者版本号实现的。 此外，如果希望用户能够动态地把他们的数据库从一个版本更新到下一个版本，应该提供更新脚本来做必要的更改。更新脚本的名称遵循*extension--old_version--target_version*.sql模式（例如，foo--1.0--1.1.sql包含着把扩展foo的版本1.0修改成版本1.1的命令）。

假定有一个合适的更新脚本可用，命令ALTER EXTENSION UPDATE将把一个已安装的扩展更新到指定的新版本。更新脚本运行在与CREATE EXTENSION提供给安装脚本相同的环境中：特别

是search_path会按照相同的方式设置，并且该脚本创建的任何新对象会被自动地加入到扩展中。此外，如果脚本选择删除扩展的成员对象，它们会自动与扩展解除关联。

如果一个扩展具有二级控制文件，用于更新脚本的控制参数是那些与新目标版本相关的参数。

更新机制可以被用来解决一种重要的特殊情况：将一个“松散的”对象集合转变成一个扩展。在扩展机制被加入到UXsinoDB之前，很多人编写的扩展模块简单地创建各种各样未打包的对象。给定一个包含这类对象的现有数据库，怎样才能将这些对象转变成一个正确打包的扩展？将它们全部删除然后做一次CREATE EXTENSION是一种方法，但是如果对象之间有依赖（例如，如果有一些表列使用了扩展创建的数据类型）这就行不通。修正这种情况的方法是创建一个空扩展，然后使用ALTER EXTENSION ADD把每一个以前就存在的对象附着到该扩展，最后创建在当前扩展版本中而不再未打包版本中的任何新对象。CREATE EXTENSION用它的FROM *old_version*选项支持这种情况，这会导致它不为目标版本运行正常的安装脚本，而是运行名为*extension--old_version--target_version*.sql的更新脚本。选择作为*old_version*使用的虚假版本名称是扩展作者的工作，不过unpackaged是一种习惯用法。如果有多个早期版本需要更新到扩展风格，使用多个虚假版本名称来标识它们。

ALTER EXTENSION能够执行更新脚本的序列来实现一个要求的更新。例如，如果只有foo--1.0--1.1.sql和foo--1.1--2.0.sql可用，当前安装了1.0版本并且要求更新到版本2.0，ALTER EXTENSION将依次应用它们。

UXsinoDB并不假定任何有关版本名称的性质：例如，它不知道1.1是否跟在1.0后面。它只是匹配可用的版本名称并且遵照要求应用最少更新脚本的路径进行（一个版本名称实际上可以是不含--或者前导或后缀-的字符串）。

有时提供“降级”脚本也有用，例如foo--1.1--1.0.sql允许把版本1.1相关的改变恢复原状。如果这样做，要当心降级脚本被意外应用的可能性，因为它会得到一个较短的路径。危险的情况是，有一个跳过几个版本的“快速路径”更新脚本还有一个降级到该快速路径开始点的降级脚本。先应用降级然后再应用快速路径可能比一次升级一个版本需要更少的步骤。如果降级版本删除了任何不可替代的对象，这将会得到意想不到的结果。

要检查意料之外的更新路径，可使用如下命令。

```
SELECT * FROM ux_extension_update_paths('extension_name');
```

这会为指定的扩展显示已知的每一个可区分的版本名对，每一个版本名对还带有一个从源版本到目标版本的更新路径序列，如果没有可用的更新路径则这部份信息为NULL。该路径显示为用--分隔符的文本形式。如果更喜欢数组格式，可以使用regexp_split_to_array(path,'--')。

## 1.17.6. 用更新脚本安装扩展

一个已经存在一段时间的扩展可能存在多个版本，作者将需要为它们编写更新脚本。例如，如果已经发布了扩展foo的版本1.0、1.1和1.2，就应该有更新脚本foo--1.0--1.1.sql和foo--1.1--1.2.sql。在之前版本中，还有必要创建新的脚本文件foo--1.1.sql和foo--1.2.sql，它们直接构建比较新的扩展版本，或者新的版本无法被直接安装，而是通过先安装1.0然后更新。那种方式是无聊的重复性工作，但是现在它是不必要的了，因为CREATE EXTENSION能够自动遵循更新链。例如，如果只有脚本文件foo--1.0.sql、foo--1.0--1.1.sql和foo--1.1--1.2.sql可用，那么安装版本1.2的请求会通过按顺序运行上述三个脚本来实现。这种处理和先安装1.0然后更新到1.2是一样的（和ALTER EXTENSION UPDATE一样，如果有多条路径可用则优先选择最短的）。按这种风格安排扩展 的脚本文件可以减少生产小更新所需的维护工作量。

如果以这种风格维护的扩展中使用了二级（版本相关的）控制文件，记住每个版本都需要一个控制文件，即使它没有单独的安装脚本，因为该控制文件将决定如何执行到这个版本的隐式更新。

例如，如果foo--1.0.control指定有requires = 'bar'，但foo的其他控制文件没有这样做，在从1.0更新到另一个版本时，该扩展对bar的依赖将被删除。

## 1.17.7. 扩展实例

这里是一个只用SQL的扩展的完整示例，一个两个元素的组合类型，它可以在它的槽（命名为“k”和“v”）中存储任何类型的值。非文本值会被自动强制为文本进行存储。

脚本文件pair--1.0.sql，如下所示。

```
-- 如果脚本是由 uxsql 而不是 CREATE EXTENSION 执行，则报错
\echo Use "CREATE EXTENSION pair" to load this file. \quit

CREATE TYPE pair AS ( k text, v text );

CREATE OR REPLACE FUNCTION pair(text, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::@extschema@.pair;';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, FUNCTION = pair);

-- "SET search_path"容易操作，但限定名称更好。
CREATE OR REPLACE FUNCTION lower(pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW(lower($1.k), lower($1.v))::@extschema@.pair;'
SET search_path = ux_temp;

CREATE OR REPLACE FUNCTION pair_concat(pair, pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW($1.k OPERATOR(ux_catalog.||) $2.k,
        $1.v OPERATOR(ux_catalog.||) $2.v)::@extschema@.pair;';
```

控制文件pair.control，如下所示。

```
# pair 扩展
comment = 'A key/value pair data type'
default_version = '1.0'
relocatable = false
```

虽然几乎不会需要一个 makefile 来安装这两个文件到正确的目录，还是可以使用一个Makefile，如下所示。

```
EXTENSION = pair
DATA = pair--1.0.sql

UX_CONFIG = ux_config
UXXS := $(shell $(UX_CONFIG) --uxxs)
include $(UXXS)
```

这个 makefile 依赖于UXXS，它在第 1.18 节 “扩展的构建基础设施”中描述。命令make install将把控制和脚本文件安装到ux_config报告的正确的目录中。

一旦文件被安装，使用CREATE EXTENSION命令就可以把对象载入到任何特定的数据库中。

# 1.18. 扩展的构建基础设施

如果正在考虑发布UXsinoDB扩展模块，为它们建立一个可移植的构建系统实在是相当困难。因此UXsinoDB安装为扩展提供了一种被称为UXXS构建基础设施，因此简单的扩展模块能够在一个已经安装的服务器上简单地编译。UXXS主要是为了包括 C 代码的扩展而设计，不过它也能用于纯SQL 的扩展。注意UXXS并不想成为一种用于构建任何与UXsinoDB交互的软件的通用构建系统框架。它只是简单地把简单服务器扩展模块的公共构建规则自动化。对于更复杂的包，可能需要编写自己的构建系统。

要把UXXS基础设施用于扩展，必须编写一个简单的 makefile。在这个 makefile 中，需要设置一些变量并且把它们包括在全局的UXXS makefile 中。这里有一个示例，它构建一个名为isbn_issn的扩展模块，其中包括一个含有 C 代码的共享库、一个扩展控制文件、一个SQL 脚本、一个包括文件（仅当其他模块可能需要通过调用而不是SQL访问这个扩展的函数时才需要）以及一个文档文件，如下所示。

```
MODULES = isbn_issn
EXTENSION = isbn_issn
DATA = isbn_issn--1.0.sql
DOCS = README.isbn_issn
HEADERS_isbn_issn = isbn_issn.h

UX_CONFIG = ux_config
UXXS := $(shell $(UX_CONFIG) --uxxs)
include $(UXXS)
```

最后三行应该总是相同的。在这个文件的前面部分，要对变量赋值或者增加自定义的make规则。

设置这三个变量之一来指定要构建什么，如下所示。

- MODULES

  要从源文件构建的具有相同词干的共享库对象的列表（不要在这个列表中包括库后缀）

- MODULE_big

  一个要从多个源文件中构建的共享库（在OBJS中列出对象文件）

- PROGRAM

  一个要构建的可执行程序（在OBJS中列出对象文件）

还可以设置下列变量。

- EXTENSION

  扩展名称；必须为每一个名称提供一个*extension*.control文件，它将被安装到*prefix*/share/extension中

- MODULEDIR

  subdirectory of *prefix*/share的子目录，DATA 和 DOCS 文件会被安装到其中（如果没有设置，设置了EXTENSION时默认为extension，没有设置EXTENSION时默认为contrib）

- DATA

  要安装到*prefix*/share/$MODULEDIR中的随机文件

- DATA_built

  要安装到*prefix*/share/$MODULEDIR中的随机文件，它们需要先被构建

- DATA_TSEARCH

  要安装到*prefix*/share/tsearch_data中的随机文件

- DOCS

  要安装到*prefix*/doc/$MODULEDIR中的随机文件

- HEADERS HEADERS_built

  要（构建并且）安装在*prefix*/include/server/$MODULEDIR/$MODULE_big下面的文件。

  和DATA_built不同，HEADERS_built中的文件不会被clean目标移除，如果想要移除它们，把它们也加入到EXTRA_CLEAN或者增加自己的规则来做这件事。

- HEADERS_$MODULE HEADERS_built_$MODULE

  要安装（如果指定了构建则在构建之后安装）在*prefix*/include/server/$MODULEDIR/$MODULE之下的文件，这里$MODULE必须是一个在MODULES or MODULE_big中用到的模块名。

  和DATA_built不同，HEADERS_built_$MODULE中的文件不会被clean目标移除，如果想要移除它们，把它们也加入到EXTRA_CLEAN或者增加自己的规则来做这件事。

  可以为同一个模块同时使用这两个变量或者两者的任意组合，除非在MODULES列表中有两个模块名称仅有前缀built_上的区别，因为那样会导致歧义。在那种情况下（还好不太可能），应该仅使用HEADERS_built_$MODULE变量。

- SCRIPTS

  要安装到*prefix*/bin中的脚本文件（非二进制）

- SCRIPTS_built

  要安装到*prefix*/bin中的脚本文件（非二进制），它们需要先被构建

- REGRESS

  回归测试案例（不带后缀）的列表，见下文

- REGRESS_OPTS

  要传递给ux_regress的附加开关

- ISOLATION

  隔离测试用例列表，请参见下文了解更多详细信息。

- ISOLATION_OPTS

要传递给ux_isolation_regress的附加开关

- TAP_TESTS

  是否需要运行 TAP 测试的开关定义，请参见下文

- NO_INSTALLCHECK

  不定义installcheck目标，如果测试要求特殊的配置就会很有用，或者不使用ux_regress

- EXTRA_CLEAN

  要在make clean中移除的额外文件

- UX_CPPFLAGS

  将被加到CPPFLAGS前面

- UX_CFLAGS

  将被加到CFLAGS后面

- UX_CXXFLAGS

  将被加到CXXFLAGS后面

- UX_LDFLAGS

  将被加到LDFLAGS前面

- UX_LIBS

  将被加到PROGRAM链接行

- SHLIB_LINK

  将被加到MODULE_big链接行

- UX_CONFIG

  要在其中构建的UXsinoDB安装的ux_config程序的路径（通常只用在PATH中的第一个ux_config）

把这个 makefile 作为Makefile放在保存扩展的目录中。然后可以执行make进行编译，并且接着make install来安装模块。默认情况下，该模块会为PATH中找到的第一个ux_config程序所对应的UXsinoDB安装编译和安装。可以通过在 makefile 中或者make命令行中设置UX_CONFIG指向另一个ux_config程序来使用一个不同的安装。

如果想保持编译目录独立，可以在扩展所属的源代码树之外的目录中运行make。 这个过程也被称为一个VPATH编译。做法如下所示。

```
mkdir build_dir
cd build_dir
make -f /path/to/extension/source/tree/Makefile
```

make -f /path/to/extension/source/tree/Makefile install

此外，对核心代码所作的方式一样为 VPATH 设置一个目录。一种方式是使用核心脚本config/prep_buildtree。一旦这样做，可以这样设置make变量VPATH，如下所示。

make VPATH=/path/to/extension/source/tree
make VPATH=/path/to/extension/source/tree install

这个过程可以在很多种目录布局下工作。

列举在REGRESS变量中的脚本会被用来对扩展进行回归测试，回归测试可以在做完make install之后用make installcheck调用。要让这能够工作，必须已经有一个运行着的UXsinoDB服务器。列举在REGRESS中的脚本文件必须扩展目录名为sql/的子目录中出现。这些文件必须带有扩展.sql，但扩展不能被包括在 makefile 的REGRESS列表中。对每一个测试还应该在名为expected/的子目录中有一个包含预期输出的文件，它具有和脚本文件相同的词干并带有扩展.out。make installcheck会用uxsql执行每一个测试脚本，并且将得到结果输出与相应的预期输出比较。任何区别都将以diff -c格式写入到文件regression.diffs中。注意尝试运行一个不带预期文件的测试将被报告为“故障”，因此确保拥有所有的预期文件。

ISOLATION变量中列出的脚本用于测试强调与模块并发会话的行为，可以在make install之后通过make installcheck调用。 要实现这个工作，必须有一个正在运行的UXsinoDB服务器。ISOLATION中列出的脚本文件必须显示在扩展名目录中名为 specs/的子目录中。这些文件必须具备扩展名.spec，并且不得包含在 makefile 中的ISOLATION列表中。 对于每个测试，在名为expected/的子目录中还应该有一个包含预期输出的文件，并且具有相同的词干和扩展名 .out。make installcheck执行每个测试脚本，并将结果输出与匹配的预期文件进行比较。任何差异都将以diff -c的格式写入到output_iso/regression.diffs文件中。请注意，尝试运行缺少其预期文件的测试将会报告“trouble”，因此请确保具有全部的预期文件。

TAP_TESTS 启用TAP测试，每个运行中的数据都存在于名为 tmp_check/的子目录中。

> **提示**
>
> 创建预期文件最简单的方法是创建空文件，然后做一次测试运行（这当然将报告区别）。 检查在results/目录中找到的实际结果文件 （对于REGRESS中的测试）， 或output_iso/results/ 目录(对于ISOLATION)中的测试，如果它们符合预期则把它们复制到expected/中。

# 第 2 章 触发器

本章提供了编写触发器函数的一般信息。触发器函数可以使用大部分可用过程性语言，包括PL/uxSQL（第 6 章 PL/uxSQL - SQL过程语言、 PL/Tcl（第 7 章 PL/Tcl - Tcl 过程语言、PL/Perl（第 8 章 PL/Perl - Perl 过程语言和 PL/Python（第 9 章 PL/Python - Python 过程语言)。阅读本章后，应该查阅过程性语言来找到用其编写触发器的相关细节。

也可以使用C编写一个触发器函数，尽管大部分人觉得使用一种过程性语言更简单。当前不能用纯SQL函数语言来编写触发器函数。

## 2.1. 触发器行为概述

一个触发器声明了当执行一种特定类型的操作时数据库应该自动执行一个特殊的函数。触发器可以被附加到表（分区的或者不分区的）、视图和外部表。

在表和外部表上，触发器可以被定义为在INSERT、UPDATE或DELETE操作之前或之后被执行，可以为每个SQL语句被执行一次或者为每个修改的行被执行一次。UPDATE触发器可以进一步地设置为只针对UPDATE语句的SET子句的特定列出发。触发器也可以被TRUNCATE语句触发。如果一个触发器事件发生，触发器函数会在适当的事件被调用来处理该事件。

在视图上，触发器可以被定义来取代INSERT、UPDATE或DELETE操作的执行。这种INSTEAD OF触发器对视图中需要被修改的每一行触发一次。触发器函数的职责是对视图的底层基本表执行必要的修改，并且在合适的时候返回被修改的行以便显示在视图中。视图上的触发器也可以被定义为对每个SQL语句执行一次，在INSERT\UPDATE或DELETE操作之前或之后。不过，只有在该视图上还有一个INSTEAD OF触发器时，上述那些触发器才会被触发。否则，以该视图为目标的任何语句都必须被重写成一个影响其底层基表的语句，然后附着在那些基表上的触发器将会被引发。

触发器函数必须在触发器本身被创建之前被定义好。触发器函数必须被定义成一个没有参数的函数，并且返回类型为trigger（触发器函数通过一个特殊传递的TriggerData结构作为其输入，而不是以普通函数参数的形式）。

一旦一个合适的触发器函数被创建，就可以使用CREATE TRIGGER建立触发器。同一个触发器函数可以被用于多个触发器。

UXsinoDB同时提供每行的触发器和每语句的触发器。对于一个每行的触发器，对于触发触发器的语句所修改的每一行都会调用一次触发器函数。相反，一个每语句的触发器对于其触发语句只被调用一次，而不管该语句影响了多少行。特别地，一个不影响任何行的语句仍然会导致任何可用每语句的触发器的执行。这两类触发器有时也分别被称作行级触发器和语句级触发器。TRUNCATE上的触发器只能被定义在语句级。

触发器也可以根据它们是否在操作之前、之后触发，或者被触发来取代操作来分类。它们分别指BEFORE触发器、AFTER 触发器以及INSTEAD OF触发器。语句级BEFORE触发器在语句开始做任何事情之前被触发，而语句级AFTER触发器则在语句做完所有事情之后被触发。这些触发器类型可以被定义在表、视图或外部表上。行级BEFORE触发器在每一个行被操作之前被触发，而行级AFTER触发器在语句结束之后被触发（但在任何语句级AFTER触发器之前）。这些触发器类型只能被定义在非分区表和外部表上，但不能定义在视图上。INSTEAD OF触发器只能被定义在视图上，并且只能定义在行级，当视图中的每一行被标识为需要被操作时，它们会立即触发。

一个以继承或者分区层次中父表为目标的语句不会导致受影响的子表的语句级触发器被引发，只有父表的语句级触发器会被引发。不过，受影响的子表的行级触发器将被引发。

如果一个INSERT包含ON CONFLICT DO UPDATE子句并且引用了EXCLUDED列，有可能所有行级BEFORE INSERT触发器和所有行级BEFORE UPDATE触发器的

效果可能会以一种对于被更新行最终状态透明的方式被应用。不过，对于要执行的两种集合的行级BEFORE触发器都不需要有EXCLUDED列引用。当同时有行级BEFORE INSERT和BEFORE UPDATE触发器影响被插入/更新的行时（如果在两者不幂等时修改或多或少地等价，这仍可能是有问题的），应该考虑可能出现的意料之外的结果。注意在指定了ON CONFLICT DO UPDATE时，不管有没有行被UPDATE影响（并且不管是否采用了其他UPDATE路径），语句级UPDATE都将被执行。一个带有ON CONFLICT DO UPDATE子句的INSERT将首先执行语句级BEFORE INSERT，然后执行语句级BEFOREUPDATE触发器，接着是语句级AFTER UPDATE触发器，最后是语句级AFTER INSERT触发器。

如果一个分区表上的UPDATE导致一行移动到另一个分区，它将被从原始分区DELETE掉然后再INSERT到新分区中。在这种情况下，原始分区上所有的行级BEFORE UPDATE触发器和所有行级BEFORE DELETE触发器会被引发。然后目标分区上所有的行级BEFORE INSERT触发器会被引发。当所有这些触发器都影响被移动的行时，应该对令人惊讶的结果有心理准备。至于AFTER ROW触发器，AFTER DELETE和AFTER INSERT触发器会被应用，但AFTER UPDATE触发器不会被应用，因为UPDATE已经被转换成了一个DELETE和一个INSERT。对于语句级触发器，即便发生行移动，DELETE和INSERT触发器也都不会被引发，只有UPDATE语句中用到的目标表上的UPDATE触发器将被引发。

被语句级触发器调用的触发器函数应该总是返回NULL。根据行级触发器的选择，被其调用的触发器函数可以返回一个表行（类型HeapTuple的一个值）给执行器。在一个操作前触发的行级触发器有下列选择。

- 它可以返回NULL来跳过对当前行的操作。这指示执行器不要执行调用触发器的行级操作（对一个特定表行的插入、修改或删除）。
- 仅对行级INSERT和UPDATE触发器来说，被返回的行称为将要被插入的行或者替代将被更新的行。这允许触发器函数修改将要被插入或更新的行。

一个无意导致任何这些行为的行级BEFORE触发器必须小心地它的结果，使之和被传入的行一样（即，INSERT和UPDATE触发器的NEW行，DELETE触发器的OLD行）。

一个行级INSTEAD OF触发器可以返回NULL来指示它没有修改任何来自于视图底层基表的数据，也可以返回被传入的视图行（INSERT和UPDATE操作的NEW行，或者DELETE操作的OLD行）。一个非空返回值被用于标志触发器在视图中执行了必须的数据修改。这将会导致被命令修改的行计数被增加。仅对于INSERT和UPDATE操作，触发器可能会在返回NEW行之前对其进行修改。这将会改变INSERT RETURNING或UPDATE RETURNING返回的数据，并在视图无法正确地显示提供给它的相同数据时有用。

对于在一个操作之后触发的行级触发器，返回值会被忽略，因此它们可以返回NULL。

一些情况适用于生成的列。存储生成的列在BEFORE触发器之后和AFTER触发器之前计算。因此，生成的值可以在AFTER触发器中检查。在BEFORE触发器中，OLD行包含旧的生成的值，正如人们所期待的，但 NEW 行尚未包含新的生成值并且不应访问。在C语言界面中，此时列的内容还没有被定义；在BEFORE触发器中，高级别编程语言应阻止访问NEW行中存储生成的列。在BEFORE触发器中更改到生成列的值将被忽略并覆盖。

如果为同一个关系上的同一事件定义了超过一个触发器，它们将按照其名称的字母表顺序被触发。在BEFORE和INSTEAD OF触发器的情况下，每一个触发器返回的可能被修改的行将成为下一个触发器的输入。如果任何一个BEFORE或INSTEAD OF触发器返回NULL，该操作将在该行上被禁用并且对于该行不会触发后续的触发器。

一个触发器定义也能指定一个布尔的WHEN条件，它将被测试来看该触发器是否应该被触发。在行级触发器中，WHEN条件可以检查该行的旧列值和/或新列值（语句级触发器也能有WHEN条件，但是该特性对它们不太有用）。在一个BEFORE触发器中，WHEN条件只是在该函数被或者将被执行前计算，因此使用WHEN条件与在该触发器函数的开始测试相同的条件没有本质区别。

不过，在一个AFTER触发器中，WHEN条件只是在行更新发生之后被计算，并且它决定在语句的末尾一个事件是否被排队来触发该触发器。因此当一个AFTER触发器的WHEN不返回真时，在语句的末尾没有必要将一个事件进行排队，也没有必要重新取出该行。如果触发器只对少数行触发，这可以使得修改很多行的语句明显加快。INSTEAD OF触发器不支持WHEN条件。

通常，行级BEFORE被用来检查或修改即将被插入或更新的数据。例如，一个BEFORE触发器可以被用来把当前时间插入到一个timestamp列中，或者检查该行的两个元素之间是否一致。行级AFTER触发器大多数被用来将更新传播到其他表，或者针对其他表进行一致性检查。进行这种工作分工的原因是，一个AFTER触发器可以肯定它看到的是该行的最终值，而一个BEFORE触发器则不能，因为还可能有其他BEFORE触发器在它之后触发。如果不知道让一个触发器是BEFORE或AFTER，则BEFORE形式更加有效，因为关于该操作的信息直到语句的末尾都不需要被保存。

如果一个触发器函数执行 SQL 命令，则这些命令可能会再次引发触发器。这就是所谓的级联触发器。对于级联的层数没有直接的限制。级联有可能会导致对同一个触发器的递归调用。例如，一个INSERT触发器可能执行一个向同一个表插入一个额外行的命令，这就导致该INSERT触发器被再次引发。所以在这种情形下，触发器程序员应该负责避免无限递归。

在定义一个触发器时，可以为它指定参数。在触发器定义中包括参数的目的是允许具有相似需求的不同触发器调用同一个函数。例如，可能有一个一般性的触发器函数，它需要两个列名作为参数，一个放当前用户而另一个放当前时间戳。在正确编写的情况下，这个触发器函数应该独立于它所触发的表。因此同一个函数可以被用于具有适当列的任意表上的INSERT事件，这样做的用途之一是可以自动追踪一个交易表中记录的创建。如果被定义成一个UPDATE触发器，它也可以被用来追踪最新的更新事件。

每一种支持触发器的编程语言都有自己的方法来让触发器输入数据对触发器函数可用。这种输入数据包括触发器事件的类型（如INSERT或UPDATE）以及被列在CREATE TRIGGER中的任何参数。对于一个行级触发器，输入数据还包括用于INSERT和UPDATE触发器的NEW行，和/或用于UPDATE和DELETE触发器的OLD行。语句级触发器当前没有任何方法检查被语句修改的单个行。

默认情况下，语句级触发器没有办法检查该语句修改的行。但是AFTER STATEMENT触发器可以请求创建*传递表*，这样可以让受影响的行集合对该触发器可用。AFTER ROW触发器也可以请求传递表，这样它们可以看到表中的整个变化，同时也能看到当前引发它们的个体行中的变化。检查传递表的方法仍是取决于使用的编程语言，但是通常的方法让传递表变得像触发器函数内部发出的SQL命令能够访问的只读临时表一样。

# 2.2. 数据改变的可见性

如果触发器函数中执行 SQL 命令，并且这些命令会访问触发器所在的表，那么需要注意数据可见性规则。因为这些规则决定了这些 SQL 命令是否将能看见引发触发器的数据改变。如下所示。

- 语句级触发器遵循简单的可见性规则：一个语句所作的改变对于语句级BEFORE触发器都不可见，而所有修改对于语句级AFTER触发器都是可见的。
- 导致触发器被引发的数据更改（插入、更新或删除）自然对于在一个行级BEFORE触发器中执行的 SQL 命令不可见，因为它还没有发生。
- 但是，在一个行级BEFORE触发器中执行的 SQL 命令将会看见之前在同一个外层命令中所作的数据更改的效果。这里需要小心，因为这些更改时间的顺序通常是不可预测的，一个影响多行的 SQL 命令可能以任何顺序访问这些行。
- 类似地，一个行级INSTEAD OF触发器将会看见之前在同一个外层命令中INSTEAD OF触发器引发所作的数据更改。

- 当一个行级AFTER触发器被引发时，所有由外层命令所作的数据更改已经完成，并且对于该被调用的触发器函数是可见的。

如果触发器函数使用任何一种标准过程语言编写的，那么只有在该函数被声明为VOLATILE时上述陈述才适用。被声明为STABLE或IMMUTABLE的函数在任何情况下将不能看到由调用命令所作出的更改。

有关数据可见性规则的更多信息可见第 10.5 节 “数据改变的可见性”。第 2.4 节 “一个完整的触发器实例”中的示例包含了对这些规则的示范。

# 2.3. 用 C 编写触发器函数

这一节描述了一个触发器函数的接口的低层细节。只有用 C 编写触发器函数时才需要这些信息。如果使用一种更高层的语言，那么这些细节就不需要处理。在大部分情况下，应该优先考虑使用一种过程语言。每一种过程语言的文档阐述了如何使用那种语言编写一个触发器。

触发器函数必须使用“版本 1”函数管理器接口。

当一个函数被触发器管理器调用时，不会给它传递任何常规的参数，但是会有一个“context”指针传递给它，该指针指向一个TriggerData结构。C 函数可以通过执行一个宏来检查它们是否是从触发器管理器被调用。

CALLED_AS_TRIGGER(fcinfo)

展开成如下所示。

((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))

如果这返回真，那么将fcinfo->context造型成类型TriggerData *并且利用所指向的TriggerData结构就是安全的。该函数不能修改该TriggerData结构或者它指向的任何数据。

struct TriggerData被定义在commands/trigger.h中，如下所示。

```
typedef struct TriggerData
{
    NodeTag         type;
    TriggerEvent    tg_event;
    Relation        tg_relation;
    HeapTuple       tg_trigtuple;
    HeapTuple       tg_newtuple;
    Trigger         *tg_trigger;
    TupleTableSlot  *tg_trigslot;
    TupleTableSlot  *tg_newslot;
    Tuplestorestate *tg_oldtable;
    Tuplestorestate *tg_newtable;
} TriggerData;
```

其中的成员定义如下所示。

- *type*

  总是T_TriggerData。

- *tg_event*

  描述该函数是为什么事件被调用的。可以使用下列宏来检查tg_event。

  - TRIGGER_FIRED_BEFORE(tg_event)

    如果该触发器在操作前被引发则返回真。

  - TRIGGER_FIRED_AFTER(tg_event)

    如果该触发器在操作后被引发则返回真。

  - TRIGGER_FIRED_INSTEAD(tg_event)

    如果该触发器被引发替代操作则返回真。

  - TRIGGER_FIRED_FOR_ROW(tg_event)

    如果该触发器为一个行级事件而引发则返回真。

  - TRIGGER_FIRED_FOR_STATEMENT(tg_event)

    如果该触发器为一个语句级事件而引发则返回真。

  - TRIGGER_FIRED_BY_INSERT(tg_event)

    如果该触发器由一个INSERT命令引发则返回真。

  - TRIGGER_FIRED_BY_UPDATE(tg_event)

    如果该触发器由一个UPDATE命令引发则返回真。

  - TRIGGER_FIRED_BY_DELETE(tg_event)

    如果该触发器由一个DELETE命令引发则返回真。

  - TRIGGER_FIRED_BY_TRUNCATE(tg_event)

    如果该触发器由一个TRUNCATE命令引发则返回真。

- *tg_relation*

  一个结构指针，该结构描述该触发器为其引发的关系。关于这个结构的细节请参见utils/rel.h。最有趣的东西是tg_relation->rd_att（该关系元组的描述符）和tg_relation->rd_rel->relname（关系名称，该类型不是char*而是NameData。如果需要该名称的一个拷贝，可使用SPI_getrelname(tg_relation)来得到一个char*）。

- *tg_trigtuple*

  一个该触发器为其引发的行的指针。这是被插入、更新或删除的行。如果这个触发器是为一个INSERT或DELETE而引发，在不想把该行替换成另一行（在INSERT的情况中）或不想跳过该操作时应该从该函数中返回它。对于外部表上的触发器，此中的系统列值未被指定。

- *tg_newtuple*

  如果该触发器为一个UPDATE而引发，则是一个指向该行新版本的指针。如果是为一个INSERT或DELETE而引发，则是NULL。如果事件是一个UPDATE并且不想用一个不同的行替换这

个行或者不想跳过该操作时，必须从函数中返回它。对于外部表上的触发器，此中的系统列值未被指定。

- *tg_trigger*

  一个指向类型为Trigger的结构的指针，定义在utils/reltrigger.h中。

```
typedef struct Trigger
{
    Oid        tgoid;
    char      *tgname;
    Oid        tgfoid;
    int16      tgtype;
    char       tgenabled;
    bool       tgisinternal;
    Oid        tgconstrrelid;
    Oid        tgconstrindid;
    Oid        tgconstraint;
    bool       tgdeferrable;
    bool       tginitdeferred;
    int16      tgnargs;
    int16      tgnattr;
    int16     *tgattr;
    char     **tgargs;
    char      *tgqual;
    char      *tgoldtable;
    char      *tgnewtable;
} Trigger;
```

  其中*tgname*是该触发器的名称，*tgnargs*是*tgargs*中参数的数量，而*tgargs*是一个指向CREATE TRIGGER语句中指定的参数的指针数组。其他成员只用于内部用途。

- *tg_trigtuplebuf*

  包含*tg_trigtuple*的插槽。或者一个NULL指针，如果没有这样的元组的话。

- *tg_newtuplebuf*

  包含*tg_trigtuple*的插槽。或者一个NULL指针，如果没有这样的元组的话。

- *tg_oldtable*

  一个指向Tuplestorestate类型的结构的指针，该结构包含格式由*tg_relation*指定的零行或者多行。如果没有OLD TABLE传递关系，则为NULL指针。

- *tg_newtable*

  一个指向Tuplestorestate类型的结构的指针，该结构包含格式由*tg_relation*指定的零行或者多行。如果没有NEW TABLE传递关系，则为NULL指针。

为了允许通过SPI发出的查询引用传递表，请参见SPI_register_trigger_data(3)。

一个触发器函数必须返回一个HeapTuple指针或一个NULL指针（不是一个 SQL 空值，也就是不会设置*isNull*为真）。如果不希望修改正在被操作的行，要小心地根据情况返回*tg_trigtuple*或*tg_newtuple*。

# 2.4. 一个完整的触发器实例

这里有一个用 C 编写的触发器函数的非常简单的示例（用过程语言编写的触发器的示例可以在过程语言的文档中找到）。

如果该命令试图向列*x*中插入一个空值，函数trigf报告表ttest中的行数并且跳过实际的操作（这样该触发器会作为一个非空约束但不会中止事务）。

首先，表定义如下所示。

```
CREATE TABLE ttest (
    x integer
);
```

该触发器函数的源代码如下所示。

```
#include "uxdb.h"
#include "fmgr.h"
#include "executor/spi.h"       /* this is what you need to work with SPI */
#include "commands/trigger.h"   /* ... triggers ... */
#include "utils/rel.h"          /* ... and relations */

UX_MODULE_MAGIC;

UX_FUNCTION_INFO_V1(trigf);

Datum
trigf(UX_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* make sure it's called as a trigger at all */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* tuple to return to executor */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* check for null values */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;
```

```
    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connect to SPI manager */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* get number of rows in table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) returns int8, so be careful to convert */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

在编译了该源代码（见第 1.10.5 节 “编译和链接动态载入的函数”）之后，声明该函数和触发器，如下所示。

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'filename'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();
```

现在可以测试该触发器的操作，如下所示。

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                             ^^^^^^^^
                        remember what we said about visibility.
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                             ^^^^^^
                        remember what we said about visibility.
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
```

```
INFO:  trigf (fired after ): there are 0 rows in ttest
                             ^^^^^^
                      remember what we said about visibility.
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

在src/test/regress/regress.c和spi中有更多复杂的示例。

# 第 3 章 事件触发器

为了对第 2 章 触发器讨论的触发器机制加以补充，UXsinoDB也提供了事件触发器。和常规触发器（附着在一个表上并且只捕捉 DML 事件）不同，事件触发器对一个特定数据库来说是全局的，并且可以捕捉 DML 事件。

和常规触发器相似，可以用任何包括了事件触发器支持的过程语言或者 C 编写事件触发器，但是不能用纯 SQL 编写。

## 3.1. 事件触发器行为总览

只要与一个事件触发器相关的事件在事件触发器所在的数据库中发生，该事件触发器就会被引发。当前支持的事件是ddl_command_start、ddl_command_end、table_rewrite和sql_drop。未来的发行版中可能会增加对更多事件的支持。

ddl_command_start事件就在CREATE、ALTER、DROP、SECURITY LABEL、COMMENT、GRANT或者REVOKE 命令的执行之前发生。在事件触发器引发前不会做受影响对象是否存在的检查。 不过，一个例外是，这个事件不会为目标是共享对象 — 数据库、角色 以及表空间 — 的 DDL 命令发生，也不会为目标是事件触发器的 DDL 命令发生。事件触发器机制不支持这些对象类型。 ddl_command_start也会在SELECT INTO 命令的执行之前发生，因为这等价于 CREATE TABLE AS。

ddl_command_end事件就在同一组命令的执行之后发生。为了得到发生的DDL操作的更多细节，可以从ddl_command_end事件触发器代码中使用集合返回函数ux_event_trigger_ddl_commands()。注意该触发器是在那些动作已经发生之后（但是在事务提交前）引发，并且因此系统目录会被读作已更改。

sql_drop事件为任何删除数据库对象的操作在ddl_command_end事件触发器之前发生。要列出已经被删除的对象，可以从sql_drop事件触发器代码中使用集合返回函数ux_event_trigger_dropped_objects()。注意该触发器是在对象已经从系统目录删除以后执行，因此不能再查看它们。

table_rewrite事件在表被命令ALTER TABLE和 ALTER TYPE的某些动作重写之前发生。虽然其他控制语句（例如 CLUSTER和VACUUM）也可以用来重 写表，但是它们不会触发table_rewrite事件。

不能在一个中止的事务中执行事件触发器（其他函数也一样）。因此，如果一个DDL 命令出现错误失败，将不会执行任何相关的ddl_command_end触发器。反过来，如果一个ddl_command_start触发器出现错误失败，将不会引发进一步的事件触发器，并且不会尝试执行该命令本身。类似地，如果一个ddl_command_end触发器出现错误失败，DDL 命令的效果将被回滚，就像其他包含事务中止的情况中那样。

第 3.2 节 “事件触发器触发矩阵”中有事件触发器机制所支持的完整命令列表。

事件触发器通过命令CREATE EVENT TRIGGER创建。为了创建一个事件触发器，必须首先创建一个有特殊返回类型event_trigger的函数。这个函数不一定需要返回一个值，该返回类型仅仅是作为一种信号表示该函数要被作为一个事件触发器调用。

如果对于一个特定的事件定义了多于一个事件触发器，它们将按照触发器名称的字母表顺序被引发。

一个触发器定义也可以指定一个WHEN条件，这样事件触发器（例如ddl_command_start触发器）就可以只对用户希望介入的特定命令触发。这类触发器的通常用法是用于限制用户可能执行的 DDL 操作的范围。

# 3.2. 事件触发器触发矩阵

表 3.1 “支持事件触发器的命令标签”列出了所有命令的事件触发器支持情况。

表 3.1. 支持事件触发器的命令标签

| 命令标签 | ddl_command_start | ddl_command_end | sql_drop | table_rewrite | 注解 |
|---|---|---|---|---|---|
| ALTER AGGREGATE | X | X | - | - | |
| ALTER COLLATION | X | X | - | - | |
| ALTER CONVERSION | X | X | - | - | |
| ALTER DOMAIN | X | X | - | - | |
| ALTER DEFAULT PRIVILEGES | X | X | - | - | |
| ALTER EXTENSION | X | X | - | - | |
| ALTER FOREIGN DATA WRAPPER | X | X | - | - | |
| ALTER FOREIGN TABLE | X | X | X | - | |
| ALTER FUNCTION | X | X | - | - | |
| ALTER LANGUAGE | X | X | - | - | |
| ALTER LARGE OBJECT | X | X | - | - | |
| ALTER MATERIALIZED VIEW | X | X | - | - | |
| ALTER OPERATOR | X | X | - | - | |
| ALTER OPERATOR CLASS | X | X | - | - | |
| ALTER OPERATOR FAMILY | X | X | - | - | |
| ALTER POLICY | X | X | - | - | |
| ALTER PROCEDURE | X | X | - | - | |
| ALTER PUBLICATION | X | X | - | - | |
| ALTER SCHEMA | X | X | - | - | |
| ALTER SEQUENCE | X | X | - | - | |

| 命令标签 | ddl_command_start | ddl_command_end | sql_drop | table_rewrite | 注解 |
|---|---|---|---|---|---|
| ALTER SERVER | X | X | - | - | |
| ALTER STATISTICS | X | X | - | - | |
| ALTER SUBSCRIPTION | X | X | - | - | |
| ALTER TABLE | X | X | X | X | |
| ALTER TEXT SEARCH CONFIGURATION | X | X | - | - | |
| ALTER TEXT SEARCH DICTIONARY | X | X | - | - | |
| ALTER TEXT SEARCH PARSER | X | X | - | - | |
| ALTER TEXT SEARCH TEMPLATE | X | X | - | - | |
| ALTER TRIGGER | X | X | - | - | |
| ALTER TYPE | X | X | - | X | |
| ALTER USER MAPPING | X | X | - | - | |
| ALTER VIEW | X | X | - | - | |
| COMMENT | X | X | - | - | Only for local objects |
| CREATE ACCESS METHOD | X | X | - | - | |
| CREATE AGGREGATE | X | X | - | - | |
| CREATE CAST | X | X | - | - | |
| CREATE COLLATION | X | X | - | - | |
| CREATE CONVERSION | X | X | - | - | |
| CREATE DOMAIN | X | X | - | - | |
| CREATE EXTENSION | X | X | - | - | |
| CREATE FOREIGN DATA WRAPPER | X | X | - | - | |
| CREATE FOREIGN TABLE | X | X | - | - | |

| 命令标签 | ddl_command_start | ddl_command_end | sql_drop | table_rewrite | 注解 |
|---|---|---|---|---|---|
| CREATE FUNCTION | X | X | - | - | |
| CREATE INDEX | X | X | - | - | |
| CREATE LANGUAGE | X | X | - | - | |
| CREATE MATERIALIZED VIEW | X | X | - | - | |
| CREATE OPERATOR | X | X | - | - | |
| CREATE OPERATOR CLASS | X | X | - | - | |
| CREATE OPERATOR FAMILY | X | X | - | - | |
| CREATE POLICY | X | X | - | - | |
| CREATE PROCEDURE | X | X | - | - | |
| CREATE PUBLICATION | X | X | - | - | |
| CREATE RULE | X | X | - | - | |
| CREATE SCHEMA | X | X | - | - | |
| CREATE SEQUENCE | X | X | - | - | |
| CREATE SERVER | X | X | - | - | |
| CREATE STATISTICS | X | X | - | - | |
| CREATE SUBSCRIPTION | X | X | - | - | |
| CREATE TABLE | X | X | - | - | |
| CREATE TABLE AS | X | X | - | - | |
| CREATE TEXT SEARCH CONFIGURATION | X | X | - | - | |
| CREATE TEXT SEARCH DICTIONARY | X | X | - | - | |
| CREATE TEXT SEARCH PARSER | X | X | - | - | |

| 命令标签 | ddl_command_start | ddl_command_end | sql_drop | table_rewrite | 注解 |
|---|---|---|---|---|---|
| CREATE TEXT SEARCH TEMPLATE | X | X | - | - | |
| CREATE TRIGGER | X | X | - | - | |
| CREATE TYPE | X | X | - | - | |
| CREATE USER MAPPING | X | X | - | - | |
| CREATE VIEW | X | X | - | - | |
| DROP ACCESS METHOD | X | X | X | - | |
| DROP AGGREGATE | X | X | X | - | |
| DROP CAST | X | X | X | - | |
| DROP COLLATION | X | X | X | - | |
| DROP CONVERSION | X | X | X | - | |
| DROP DOMAIN | X | X | X | - | |
| DROP EXTENSION | X | X | X | - | |
| DROP FOREIGN DATA WRAPPER | X | X | X | - | |
| DROP FOREIGN TABLE | X | X | X | - | |
| DROP FUNCTION | X | X | X | - | |
| DROP INDEX | X | X | X | - | |
| DROP LANGUAGE | X | X | X | - | |
| DROP MATERIALIZED VIEW | X | X | X | - | |
| DROP OPERATOR | X | X | X | - | |
| DROP OPERATOR CLASS | X | X | X | - | |
| DROP OPERATOR FAMILY | X | X | X | - | |
| DROP OWNED | X | X | X | - | |
| DROP POLICY | X | X | X | - | |
| DROP PROCEDURE | X | X | X | - | |
| DROP PUBLICATION | X | X | X | - | |
| DROP RULE | X | X | X | - | |

| 命令标签 | ddl_command_start | ddl_command_end | sql_drop | table_rewrite | 注解 |
|---|---|---|---|---|---|
| DROP SCHEMA | X | X | X | - | |
| DROP SEQUENCE | X | X | X | - | |
| DROP SERVER | X | X | X | - | |
| DROP STATISTICS | X | X | X | - | |
| DROP SUBSCRIPTION | X | X | X | - | |
| DROP TABLE | X | X | X | - | |
| DROP TEXT SEARCH CONFIGURATION | X | X | X | - | |
| DROP TEXT SEARCH DICTIONARY | X | X | X | - | |
| DROP TEXT SEARCH PARSER | X | X | X | - | |
| DROP TEXT SEARCH TEMPLATE | X | X | X | - | |
| DROP TRIGGER | X | X | X | - | |
| DROP TYPE | X | X | X | - | |
| DROP USER MAPPING | X | X | X | - | |
| DROP VIEW | X | X | X | - | |
| GRANT | X | X | - | - | 只对本地对象 |
| IMPORT FOREIGN SCHEMA | X | X | - | - | |
| REFRESH MATERIALIZED VIEW | X | X | - | - | |
| REVOKE | X | X | - | - | 只对本地对象 |
| SECURITY LABEL | X | X | - | - | 只对本地对象 |
| SELECT INTO | X | X | - | - | |

# 3.3. 用 C 编写事件触发器函数

这一节描述了事件触发器函数接口的低层细节。只有在用 C 编写事件触发器函数时才需要用到这里的信息。如果使用更高层的语言，那么这些细节已经被处理好了。在大部分情况下都应该优

先考虑使用过程语言来编写事件触发器。每一种过程语言的文档都解释了如何用它编写事件触发器。

事件触发器函数必须使用“版本 1”的函数管理器接口。

当一个函数被事件触发器管理器调用时，向它传递的并不是普通参数，而是一个指向EventTriggerData结构的“context”指针。C 函数可以通过执行以下宏来检查它是否被事件触发器管理器调用。

CALLED_AS_EVENT_TRIGGER(fcinfo)

这个宏会被扩展，如下所示。

((fcinfo)->context != NULL && IsA((fcinfo)->context, EventTriggerData))

如果这个宏返回真，那么就可以安全地把fcinfo->context造型为类型EventTriggerData*并且使用所指向的EventTriggerData结构。函数不能修改EventTriggerData结构以及它指向的任何内容。

struct EventTriggerData在commands/event_trigger.h中被定义，如下所示。

```
typedef struct EventTriggerData
{
   NodeTag    type;
   const char *event;     /* 事件名称 */
   Node       *parsetree; /* 解析树 */
   const char *tag;       /* 命令标签 */
} EventTriggerData;
```

其中的成员定义如下所示。

*type*

总是T_EventTriggerData。

*event*

描述要为其调用这个函数的事件，可以是"ddl_command_start"、"ddl_command_end"、"sql_drop"、"table_rewrite"之一。这些事件的含义请见第 3.1 节 “事件触发器行为总览”。

*parsetree*

该命令的解析树的指针。其细节可以参考 UXsinoDB 的源代码。 解析树结构可能会在未经通知的情况下改变。

*tag*

与事件触发器的事件相关联的命令标签，例如 "CREATE FUNCTION"。

一个事件触发器函数必须返回一个NULL指针（不是一个 SQL 空值，也就是不要把*isNull*设置为真）。

# 3.4. 一个完整的事件触发器示例

这里是一个用 C 编写的事件触发器函数的简单示例（用过程语言编写的触发器示例可以在过程语言的文档中找到）。

函数noddl在每一次被调用时抛出一个异常。事件触发器定义把该函数和ddl_command_start事件关联在了一起。其效果就是所有 DDL命令（除第 3.1 节 “事件触发器行为总览”中提到的例外）都被阻止运行。

这是该触发器函数的源代码，如下所示。

```
#include "uxdb.h"
#include "commands/event_trigger.h"


UX_MODULE_MAGIC;

UX_FUNCTION_INFO_V1(noddl);

Datum
noddl(UX_FUNCTION_ARGS)
{
   EventTriggerData *trigdata;

   if (!CALLED_AS_EVENT_TRIGGER(fcinfo))  /* internal error */
      elog(ERROR, "not fired by event trigger manager");

   trigdata = (EventTriggerData *) fcinfo->context;

   ereport(ERROR,
      (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
            errmsg("command \"%s\" denied", trigdata->tag)));

   UX_RETURN_NULL();
}
```

在编译了源代码（见第 1.10.5 节 “编译和链接动态载入的函数”）后，声明函数和触发器，如下所示。

```
CREATE FUNCTION noddl() RETURNS event_trigger
   AS 'noddl' LANGUAGE C;

CREATE EVENT TRIGGER noddl ON ddl_command_start
   EXECUTE FUNCTION noddl();
```

现在可以测试该触发器的操作，如下所示。

```
=# \dy
                List of event triggers
 Name |       Event       | Owner | Enabled | Function | Tags
------+-------------------+-------+---------+----------+------
```

```
 noddl | ddl_command_start | dim   | enabled | noddl    |
(1 row)

=# CREATE TABLE foo(id serial);
ERROR:  command "CREATE TABLE" denied
```

在这种情况下，为了在需要时能运行某些 DDL 命令，必须删除该事件触发器或者禁用它。只在一个事务期间禁用该触发器会比较方便，如下所示。

```
BEGIN;
ALTER EVENT TRIGGER noddl DISABLE;
CREATE TABLE foo (id serial);
ALTER EVENT TRIGGER noddl ENABLE;
COMMIT;
```

事件触发器本身上的 DDL 命令不受事件触发器影响。

# 3.5. 一个表重写事件触发器示例

得益于table_rewrite事件的存在，可以实现一种只允许在维护窗口中重写的表重写策略。

这里是实现这种策略的示例。

```
CREATE OR REPLACE FUNCTION no_rewrite()
 RETURNS event_trigger
 LANGUAGE pluxsql AS
$$
---
--- 实现本地表重写策略：
---   public.foo 不允许重写，其他表只允许在 1am 和 6am 之间重写，
---   且前提是它们拥有不超过 100 块
---
DECLARE
  table_oid oid := ux_event_trigger_table_rewrite_oid();
  current_hour integer := extract('hour' from current_time);
  pages integer;
  max_pages integer := 100;
BEGIN
  IF ux_event_trigger_table_rewrite_oid() = 'public.foo'::regclass
  THEN
      RAISE EXCEPTION 'you''re not allowed to rewrite the table %',
                  table_oid::regclass;
  END IF;

  SELECT INTO pages relpages FROM ux_class WHERE oid = table_oid;
  IF pages > max_pages
  THEN
      RAISE EXCEPTION 'rewrites only allowed for table with less than % pages',
                  max_pages;
  END IF;

  IF current_hour NOT BETWEEN 1 AND 6
```

```
  THEN
      RAISE EXCEPTION 'rewrites only allowed between 1am and 6am';
  END IF;
END;
$$;

CREATE EVENT TRIGGER no_rewrite_allowed
                ON table_rewrite
   EXECUTE FUNCTION no_rewrite();
```

# 第 4 章 规则系统

本章讨论UXsinoDB中的规则系统。产生规则系统的概念很简单，但是在实际使用的时候会碰到很多细节问题。

某些其它数据库系统定义活动的数据库规则，通常是存储过程和触发器。在UXsinoDB中，这些东西可以通过函数和触发器来实现。

规则系统（更准确地说是查询重写规则系统）与存储过程和触发器完全不同。它把查询修改为需要考虑规则，并且然后把修改过的查询传递给查询规划器进行规划和执行。它非常强大，并且可以被用于许多东西如查询语言过程、视图和版本。

## 4.1. 查询树

要了解规则系统是如何工作的，必须要知道它什么时候被调用以及它的输入和结果是什么。

规则系统位于解析器和规划器之间。它采用解析器的输出（即一个查询树）和用户定义的重写规则（也是查询树，不过带有一些额外信息），并且常见零个或者更多个查询树作为结果。因此它的输入和输出总是那些规划器自身就能产生的东西，并且因此它看到的任何东西都可以被表示成一个SQL语句。

那么什么是一个查询树？它是一个SQL语句的一种内部表示，其中用于创建它的每一个单独的部分都被独立存储。如果设置了配置参数debug_print_parse、debug_print_rewritten或debug_print_plan，这些查询树可以被显示在服务器日志中。规则动作也被做为查询树存储在系统目录ux_rewrite中。它们没有被格式化为日志输出的形式，但是它们包含完全相同的信息。

阅读一棵未加工的查询树需要要一些经验。但是由于查询树的SQL表示形式足以用来理解规则系统，本章将不会教授如何阅读查询树。

在阅读本章中查询树的SQL表现形式时，读者需要能够知道语句被分解成了哪些部分并且能在查询树结构中标识它们。一棵查询树有如下几个部分。

- 命令类型

  这是一个简单的值来说明是哪一种命令（SELECT、INSERT、UPDATE、DELETE）产生了该查询树。

- 范围表

  范围表是被使用在该查询中的关系的列表。在一个SELECT语句中，范围表是在关键词FROM后面给出的关系。

  每一个范围表项标识一个表或视图，并且说明在该查询的其他部分要以哪个名称调用它。在查询树中，范围表项被使用编号而不是名称来引用，因此在一个SQL语句中出现重复的名字也没有关系。在规则的范围表被合并以后可能会发生这种情况。本章中的示例将不会有这种情况。

- 结果关系

  这是一个指向范围表的索引，它标识了该查询的结果应该去哪个关系。

  SELECT查询没有结果关系（特殊情况SELECT INTO几乎等于CREATE TABLE后面跟上INSERT … SELECT，并且不在这里单独讨论）。

  对于INSERT、UPDATE和DELETE命令，结果关系是修改要进行的表（或视图）。

- 目标列表

  目标列表是一个表达式的列表，它定义了查询的结果。在一个SELECT的情况下，这些表达式会构建出该查询最终的输出。它们对应于关键字SELECT和FROM之间的表达式（*是一个关系所有列名的缩写。解析器会把它扩展成独立的列，因此规则系统永远见不到它）。

  DELETE命令不需要一个目标列表，因为它们不产生任何结果。相反，规划器会向空的目标列表中加入一个特殊的CTID项来允许执行器找到要被删除的行（当结果关系是一个普通表时才加入CTID。如果结果关系是一个视图，则会被规则系统加入一个整行变量，如第 4.2.4 节 “更新一个视图”所述）。

  对于INSERT命令，目标列表描述了将要进入到结果关系中的新行。它由VALUES子句中的表达式或来自INSERT ... SELECT中SELECT子句的表达式构成。重写处理的第一步会为那些没有被原始命令赋值但有默认值的列增加目标列项。任何剩余的列（既没有给定值也没有默认值）将被规划器用一个常量空值表达式填充。

  对于UPDATE命令，目标列表描述要替换旧行的新行。在规则系统中，它只包含来自命令的SET column = expression部分的表达式。规划器将处理缺失的列，做法是为它们插入表达式，这种表达式会把旧行的值复制到新行。正如DELETE一样，会增加一个CTID或整行变量，这样执行器能够标识要被更新的旧行。

  目标列表中的每一个项所包含的表达式可以是一个常量值、一个指向范围表中关系的列的变量、一个参数或一个由函数调用、常量、变量、操作符等构成的表达式树。

- 条件

  查询的条件是一个表达式，它很像包含在目标列表项中的表达式。这个表达式的结果值是一个布尔值，它说明对最终结果行的操作（INSERT、UPDATE、DELETE或SELECT）是否应该被执行。它对应于一个SQL语句的WHERE子句。

- 连接树

  查询的连接树展示了FROM子句的结构。对于一个SELECT ... FROM a, b, c这样的简单查询，连接树就是FROM项的一个列表，因为允许以任何顺序连接它们。但是当JOIN表达式（特别是外连接）被使用时，必须按照连接显示的顺序来连接。在这种情况下，连接树展示了JOIN表达式的结构。与特定JOIN子句（来自ON或USING）相关的限制被存储为附加到那些连接树节点的条件表达式。发现把顶层WHERE表达式存储为附加到顶层连接树项的一个条件也很方便。这样实际上连接树表达了一个SELECT的FROM和WHERE子句。

- 其他

  查询树的其他部分（如ORDER BY子句）在这里并不受到关注。规则系统在应用规则时会替换这里的某些项，但是这些与规则系统的基础没有什么关系。

# 4.2. 视图和规则系统

UXsinoDB中的视图是通过规则系统来实现的。事实上，下面的命令

CREATE VIEW myview AS SELECT * FROM mytab;

与下面两个命令相比没有不同。

CREATE TABLE myview (*same column list as mytab*);

```
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
   SELECT * FROM mytab;
```

因为这就是CREATE VIEW命令在内部所作的。 这样做有一些副作用。其中之一就是在UXsinoDB系统目录中的视图信息与表的信息完全一样。所以对于解析器来说，表和视图之间完全没有区别。它们是同样的事物：关系。

## 4.2.1. SELECT规则如何工作

规则ON SELECT被应用于所有查询作为最后一步，即使给出的是一条INSERT、UPDATE或DELETE命令。而且它们与其他命令类型上的规则有着不同的语义，它们会就地修改查询树而不是创建一个新的查询树。因此首先描述SELECT规则。

目前，一个ON SELECT规则中只能有一个动作，而且它必须是一个无条件的INSTEAD的SELECT动作。 这个限制是为了令规则足够安全，以便普通用户也可以打开它们，并且它限制ON SELECT规则使之行为类似视图。

本章的示例是两个连接视图，它们做一些运算并且某些更多视图会轮流使用它们。最前面的两个视图之一后面将利用对INSERT、UPDATE和DELETE操作增加规则的方法被自定义，这样最终结果将是一个视图，它表现得像一个具有魔力的真正的表。这个示例不适合于作为简单易懂的示例，它可能会让本章更难懂。但是用一个覆盖所有关键点的示例来一步一步讨论要比举很多示例搞乱思维好。

在前两个规则系统描述中需要真实表如下所示，它们表示鞋店的数据。

```
CREATE TABLE shoe_data (
   shoename   text,        -- 主键
   sh_avail   integer,     -- 可用的双数
   slcolor    text,        -- 首选的鞋带颜色
   slminlen   real,        -- 最小鞋带长度
   slmaxlen   real,        -- 最大鞋带长度
   slunit     text         -- 长度单位
);

CREATE TABLE shoelace_data (
   sl_name    text,        -- 主键
   sl_avail   integer,     -- 可用的双数
   sl_color   text,        -- 鞋带颜色
   sl_len     real,        -- 鞋带长度
   sl_unit    text         -- 长度单位
);

CREATE TABLE unit (
   un_name    text,        -- 主键
   un_fact    real         -- 转换到厘米的参数
);
```

创建视图，如下所示。

```
CREATE VIEW shoe AS
   SELECT sh.shoename,
          sh.sh_avail,
          sh.slcolor,
```

```
        sh.slminlen,
        sh.slminlen * un.un_fact AS slminlen_cm,
        sh.slmaxlen,
        sh.slmaxlen * un.un_fact AS slmaxlen_cm,
        sh.slunit
    FROM shoe_data sh, unit un
   WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
   SELECT s.sl_name,
        s.sl_avail,
        s.sl_color,
        s.sl_len,
        s.sl_unit,
        s.sl_len * u.un_fact AS sl_len_cm
    FROM shoelace_data s, unit u
   WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
   SELECT rsh.shoename,
        rsh.sh_avail,
        rsl.sl_name,
        rsl.sl_avail,
        least(rsh.sh_avail, rsl.sl_avail) AS total_avail
    FROM shoe rsh, shoelace rsl
   WHERE rsl.sl_color = rsh.slcolor
     AND rsl.sl_len_cm >= rsh.slminlen_cm
     AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

创建shoelace视图的CREATE VIEW命令（也是最简单的一个）将创建一个shoelace关系和一个ux_rewrite项，这个ux_rewrite项说明有一个重写规则，只要一个查询的范围表中引用了关系shoelace，就必须应用它。该规则没有规则条件（稍后和非SELECT规则一起讨论，因为目前的SELECT规则不能有规则条件）并且它是INSTEAD规则。要注意规则条件与查询条件不一样。规则的动作有一个查询条件。该规则的动作是一个查询树，这个查询是视图创建命令中的SELECT语句的一个拷贝。

> **注意**
>
> 在ux_rewrite项中看到的两个额外的用于NEW和OLD的范围表项不是SELECT规则感兴趣的东西。

现在填充unit、shoe_data和shoelace_data，并且在视图上运行一个简单的查询，如下所示。

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');
```

```
INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
 sl8       |        1 | brown    |     40 | inch    |     101.6
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

这是可以视图上做的最简单的SELECT，所以用这次机会来解释视图规则的基本要素。SELECT * FROM shoelace会被解析器解释并生成查询树，如下所示。

```
SELECT shoelace.sl_name, shoelace.sl_avail,
    shoelace.sl_color, shoelace.sl_len,
    shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

然后这将被交给规则系统。规则系统遍历范围表，检查有没有可用于任何关系的规则。在为shoelace（到目前为止的唯一一个）处理范围表时，它会发现查询树里有_RETURN规则，如下所示。

```
SELECT s.sl_name, s.sl_avail,
    s.sl_color, s.sl_len, s.sl_unit,
    s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace old, shoelace new,
    shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

要扩展该视图，重写器简单地创建一个子查询范围表项，它包含规则的动作的查询树，然后用这个范围表记录取代原来引用视图的那个。作为结果的重写后的查询树几乎与键入的一样，如下所示。

```
SELECT shoelace.sl_name, shoelace.sl_avail,
    shoelace.sl_color, shoelace.sl_len,
    shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
```

```
            s.sl_len,
            s.sl_unit,
            s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
          WHERE s.sl_unit = u.un_name) shoelace;
```

不过有一个区别：子查询的范围表有两个额外的项shoelace old和shoelace new。这些项并不直接参与到查询中，因为它们没有被子查询的连接树或者目标列表引用。重写器用它们存储最初出现在引用视图的范围表项中表达的访问权限检查信息。以这种方式，执行器仍然会检查该用户是否有访问视图的正确权限，尽管在重写后的查询中没有对视图的直接使用。

这是被应用的第一个规则。规则系统将继续检查顶层查询里剩下的范围表项（本例中没有了），并且它将递归的检查增加的子查询中的范围表项，看看其中有没有引用视图的（不过这样不会扩展old或new — 否则会得到无限递归！）。在这个示例中，没有用于shoelace_data或unit的重写规则，所以重写结束并且上面得到的就是给规划器的最终结果。

现在想写一个查询，它找出目前在店里哪些鞋子有匹配的（颜色和长度）鞋带并且完全匹配的鞋带双数大于等于二。

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

这词解析器的输出是查询树，如下所示。

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

第一个被应用的规则将是用于shoe_ready的规则并且它会导致查询树，如下所示。

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               least(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

相似地，用于shoe和shoelace的规则被替换到子查询的范围表中，得到一个三层的最终查询树，如下所示。

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
    shoe_ready.sl_name, shoe_ready.sl_avail,
    shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
          rsh.sh_avail,
          rsl.sl_name,
          rsl.sl_avail,
          least(rsh.sh_avail, rsl.sl_avail) AS total_avail
        FROM (SELECT sh.shoename,
                sh.sh_avail,
                sh.slcolor,
                sh.slminlen,
                sh.slminlen * un.un_fact AS slminlen_cm,
                sh.slmaxlen,
                sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                sh.slunit
              FROM shoe_data sh, unit un
              WHERE sh.slunit = un.un_name) rsh,
            (SELECT s.sl_name,
                s.sl_avail,
                s.sl_color,
                s.sl_len,
                s.sl_unit,
                s.sl_len * u.un_fact AS sl_len_cm
              FROM shoelace_data s, unit u
              WHERE s.sl_unit = u.un_name) rsl
        WHERE rsl.sl_color = rsh.slcolor
          AND rsl.sl_len_cm >= rsh.slminlen_cm
          AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail > 2;
```

这看起来是低效的，但是计划器会通过“pulling up”子查询将其折叠成一个单层查询树，然后它会计划联接， 就像手动写出来一样。因此，折叠查询树是重写系统本身不必关心的一种优化。

## 4.2.2. 非SELECT语句中的视图规则

有两个查询树的细节在上面的视图规则的描述中没有涉及。它们是命令类型和结果关系。实际上，视图规则不需要命令类型，但是结果关系可能会影响查询重写器工作的方式，因为如果结果关系是一个视图，需要采取特殊的措施。

一个SELECT的查询树和其它命令的查询树之间很少的几处不同。显然，它们有不同的命令类型并且对于SELECT之外的命令，结果关系指向结果将进入的范围表项。其它所有东西都完全相同。所以如果有两个表t1和t2分别有列a和b，如下所示，两个语句的查询树。

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

几乎是一样的。特别是：

- 范围表包含表t1和t2的项。

- 目标列表包含一个变量，该变量指向表t2的范围表项的列b。
- 条件表达式比较两个范围表项的列a以寻找相等。
- 连接树展示了t1和t2之间的一次简单连接。

结果是，两个查询树生成相似的执行计划：它们都是两个表的连接。 对于UPDATE语句，规划器把t1缺失的列加到目标列并且最终查询树如下所示。

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

因此在连接上运行的执行器将产生完全相同的结果集，如下所示。

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

但是在UPDATE中有个小问题：执行器计划中执行连接的部分不关心连接的结果的含义。它只是产生一个行的结果集。一个是SELECT命令而另一个是由执行器中的更高层处理的UPDATE命令，在那里执行器知道这是一个UPDATE，并且它知道这个结果应该进入表t1。但是这里的哪些行必须被新行替换呢？

要解决这个问题，在UPDATE和DELETE语句的目标列表里面增加了另外一个项：当前元组ID（CTID）。这是一个系统列，它包含行所在的文件块编号和在块中的位置。在已知表的情况下，CTID可以被用来检索要被更新的t1的原始行。在添加CTID到目标列之后，查询实际上是如下命令。

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

现在，另一个UXsinoDB的细节进入到这个阶段了。表中的旧行还没有被覆盖，这就是为什么ROLLBACK很快的原因。在一个UPDATE中，新的结果行被插入到表中（在剥除CTID之后），并且把CTID指向的旧行的行头部中的cmax和xmax项设置为当前命令计数器和当前事务 ID 。这样旧的行就被隐藏起来，并且在事务提交之后 vacuum 清理器就可以最终移除死亡的行。

知道了所有这些，就可以用完全相同的方式简单地把视图规则应用到任意命令中。没有任何区别。

## 4.2.3. UXsinoDB中视图的能力

上文演示了规则系统如何把视图定义整合到原始的查询树中。在第二个示例中，一个来自于一个视图的简单SELECT创建了一个四表连接（unit以不同的名字被用了两次）的最终查询树。

用规则系统实现视图的好处是，规划器拥有关于哪些表必须被扫描、这些表之间的联系、来自于视图的限制性条件、一个单一查询树中原始查询的条件等所有信息。当原始查询已经是一个视图上的连接时仍然是这样。规划器必须决定执行查询的最优路径，而且规划器拥有越多信息，该决定就越好。并且UXsinoDB中实现的规则系统保证这些信息是此时能获得的有关该查询的所有信息。

## 4.2.4. 更新一个视图

如果视图是INSERT、UPDATE或DELETE的目标关系会怎样？使用上文所述的替换将给出一个查询树，其中的结果关系指向一个子查询范围表项，这样无法工作。不过，UXsinoDB中有几种方法来支持更新视图。

如果子查询从一个单一基本关系选择并且该关系足够简单，重写器会自动地把该子查询替换成底层的基本关系，这样INSERT、UPDATE或DELETE会被以适当的方式应用到该基本关系。其中“足够简单”的视图被称为自动可更新。有关这种可以被自动更新的视图类别的详细信息，参见CREATE VIEW。

或者，该操作可以被定义在视图上的一个用户提供的INSTEAD　OF触发器处理。在这种情况下重写工作有一点点不同。对于INSERT，重写器对视图什么也不做，让它作为查询的结果关系。对于UPDATE和DELETE，仍有必要扩展该视图查询来产生命令将尝试更新或删除的“旧”行。因此该视图被按照通常的方式扩展，但是另一个未被扩展的范围表项会被增加到查询来表示该视图会尽其所能作为结果关系。

现在出现的问题是如何标识在视图中要被更新的行。回忆一下，当结果关系是一个表时，一个特殊的CTID项会被加入到目标列表来标识要被更新的行的物理位置。如果结果关系是一个视图这就行不通，因为一个视图根本就没有CTID，它的行没有实际的物理位置。对于一个UPDATE或DELETE操作，一个特殊的wholerow项会被增加到目标列表中，它会扩展来包括来自该视图的所有列。执行器使用这个值来提供“旧”行给INSTEAD OF触发器。现在就轮到触发器来基于新旧行值来找出要更新什么了。

另外一种可能性是让用户定义INSTEAD规则，这种规则指定对视图上的INSERT\UPDATE和DELETE命令的替代动作。这些规则将重写该命令，通常是重写成一个更新一个或多个表（而不是视图）的命令。这是第　4.4　节 INSERT、UPDATE和DELETE上的规则”的主题。

注意规则会首先被计算，然后在原始查询被规划和执行之前重写它。因此，如果一个视图上同时有INSTEAD OF触发器和INSERT、UPDATE或DELETE规则，那么首先会计算规则，然后根据其结果决定是否执行触发器，触发器可能完全都不会被使用。

Automatic rewriting of an 在一个简单视图上的INSERT、UPDATE或DELETE查询的自动重写总是在最后尝试。因此，如果一个视图有规则或触发器，它们将重载自动可更新视图的默认行为。

如果对该视图没有INSTEAD规则或INSTEAD OF触发器，并且重写器不能自动地把该查询重写成一个底层基本关系上的更新，将会抛出一个错误，因为执行器不能更新一个这样的视图。

# 4.3.　物化视图

UXsinoDB中的物化视图像视图一样使用了规则系统，但是以一种类表的形式保留了结果。在物化视图，如下所示。

CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;

和如下视图。

CREATE TABLE mymatview AS SELECT * FROM mytab;

之间的主要区别是物化视图不能直接被更新，并且用于创建物化视图的查询的存储方式和视图查询的存储方式完全相同，因此要为物化视图生成新鲜的数据，如下所示。

REFRESH MATERIALIZED VIEW mymatview;

The information about a materialized view in the 有关一个UXsinoDB系统目录中的物化视图的信息和一个表或视图的信息完全相同。因此对于解析器，一个物化视图就是一个关系，就像

一个表或一个视图。当一个物化视图被一个查询引用时，数据直接从物化视图中返回，如同表一样；规则只被用来填充物化视图。

虽然对物化视图中存储的数据的访问常常要快于直接访问底层表或通过一个视图访问，但是数据并不总是最新的；但是某些时候并不需要当前数据。考虑一个记录销售情况的表，如下所示。

```
CREATE TABLE invoice (
   invoice_no    integer       PRIMARY KEY,
   seller_no     integer,      -- 销售员的 ID
   invoice_date  date,         -- 销售日期
   invoice_amt   numeric(13,2)  -- 销售量
);
```

如果想快速绘制历史销售数据，可能希望汇总，并且可能并不关心当前日期的不完整数据，如下所示。

```
CREATE MATERIALIZED VIEW sales_summary AS
  SELECT
     seller_no,
     invoice_date,
     sum(invoice_amt)::numeric(13,2) as sales_amt
   FROM invoice
   WHERE invoice_date < CURRENT_DATE
   GROUP BY
     seller_no,
     invoice_date
   ORDER BY
     seller_no,
     invoice_date;

CREATE UNIQUE INDEX sales_summary_seller
  ON sales_summary (seller_no, invoice_date);
```

这个物化视图可能对在为销售员创建的控制面板上显示一个图表非常有用。可以用一个计划任务在每晚使用这个 SQL 语句更新该统计信息，如下所示。

```
REFRESH MATERIALIZED VIEW sales_summary;
```

物化视图的另一种使用是允许通过一个外部数据包装器对来自一个远程系统的数据进行更快的访问。下面有一个使用file_fdw的简单示例，但是由于本地系统上可以使用高速缓存，因此比起访问一个远程系统的性能差异可能会比这里所展示的更大。注意鉴于file_fdw不支持索引，也使用这种能力来在物化视图上放置索引。这种优势可能不适用于其他种类的外部数据访问。

```
CREATE EXTENSION file_fdw;
CREATE SERVER local_file FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE words (word text NOT NULL)
  SERVER local_file
  OPTIONS (filename '/usr/share/dict/words');
CREATE MATERIALIZED VIEW wrd AS SELECT * FROM words;
CREATE UNIQUE INDEX wrd_word ON wrd (word);
CREATE EXTENSION ux_trgm;
```

```
CREATE INDEX wrd_trgm ON wrd USING gist (word gist_trgm_ops);
VACUUM ANALYZE wrd;
```

现在对一个词进行拼写检查。直接使用file_fdw，如下所示。

```
SELECT count(*) FROM words WHERE word = 'caterpiler';

 count
-------
     0
(1 row)
```

通过EXPLAIN ANALYZE，如下所示。

```
 Aggregate  (cost=21763.99..21764.00 rows=1 width=0) (actual time=188.180..188.181 rows=1
 loops=1)
   ->  Foreign Scan on words  (cost=0.00..21761.41 rows=1032 width=0) (actual time=188.177..188.177
 rows=0 loops=1)
         Filter: (word = 'caterpiler'::text)
         Rows Removed by Filter: 479829
         Foreign File: /usr/share/dict/words
         Foreign File Size: 4953699
 Planning time: 0.118 ms
 Execution time: 188.273 ms
```

如果使用物化视图，该查询会快很多，如下所示。

```
 Aggregate  (cost=4.44..4.45 rows=1 width=0) (actual time=0.042..0.042 rows=1 loops=1)
   ->  Index Only Scan using wrd_word on wrd  (cost=0.42..4.44 rows=1 width=0) (actual
 time=0.039..0.039 rows=0 loops=1)
         Index Cond: (word = 'caterpiler'::text)
         Heap Fetches: 0
 Planning time: 0.164 ms
 Execution time: 0.117 ms
```

不管哪种方式，单词都是被拼错的，因此看看什么可能想要的。再次使用file_fdw，如下所示。

```
SELECT word FROM words ORDER BY word <-> 'caterpiler' LIMIT 10;

     word
---------------
 cater
 caterpillar
 Caterpillar
 caterpillars
 caterpillar's
 Caterpillar's
 caterer
 caterer's
 caters
 catered
```

(10 rows)

```
Limit  (cost=11583.61..11583.64 rows=10 width=32) (actual time=1431.591..1431.594 rows=10
loops=1)
  ->  Sort  (cost=11583.61..11804.76 rows=88459 width=32) (actual time=1431.589..1431.591 rows=10
loops=1)
        Sort Key: ((word <-> 'caterpiler'::text))
        Sort Method: top-N heapsort  Memory: 25kB
        ->  Foreign Scan on words  (cost=0.00..9672.05 rows=88459 width=32) (actual
time=0.057..1286.455 rows=479829 loops=1)
              Foreign File: /usr/share/dict/words
              Foreign File Size: 4953699
Planning time: 0.128 ms
Execution time: 1431.679 ms
```

使用物化视图，如下所示。

```
Limit  (cost=0.29..1.06 rows=10 width=10) (actual time=187.222..188.257 rows=10 loops=1)
  ->  Index Scan using wrd_trgm on wrd  (cost=0.29..37020.87 rows=479829 width=10) (actual
time=187.219..188.252 rows=10 loops=1)
        Order By: (word <-> 'caterpiler'::text)
Planning time: 0.196 ms
Execution time: 198.640 ms
```

如果能够忍受定期把远程数据更新到本地数据库，其性能收益可能是巨大的。

# 4.4. INSERT、UPDATE和DELETE上的规则

定义在INSERT、UPDATE和DELETE上的规则与前一节描述的视图规则有明显的不同。首先，它们的CREATE RULE命令允许更多操作，如下所示。

- 它们可以没有动作。
- 它们可以有多个动作。
- 它们可以是INSTEAD或ALSO（缺省）。
- 伪关系NEW和OLD变得有用了。
- 它们可以有规则条件。

第二，它们不是就地修改查询树，而是创建零个或多个新查询树并且可能把原始的那个查询树扔掉。

> **注意**
>
> 在很多情况下，由INSERT/UPDATE/DELETE上的规则执行的任务用触发器能做得更好。触发器在记法上要更复杂些，但是它们的语义理解起来更简单些。当原始查询包含不稳定函数时，规则容易产生令人惊讶的结果：在执行规则的过程中不稳定函数的执行次数可能比语气中的更多。
>
> 还有，有些情况根本无法用这些类型的规则支持，典型的是在原始查询中包括WITH子句以及在UPDATE查询的SET列表中包括多个赋值的

子SELECT。这是因为把这些结构复制到一个规则查询中可能导致子查询的多次计算，这与查询作者表达的意图相悖。

## 4.4.1. 更新规则如何工作

语法结构

CREATE [ OR REPLACE ] RULE *name* AS ON *event*
  TO *table* [ WHERE *condition* ]
  DO [ ALSO | INSTEAD ] { NOTHING | *command* | ( *command* ; *command* ... ) }

在随后的内容中，更新规则表示定义在INSERT、UPDATE或DELETE上的规则。

如果查询树的结果关系和命令类型等于CREATE RULE命令中给出的对象和事件，规则系统就会应用更新规则。对于更新规则，规则系统会创建一个查询树列表。一开始该查询树列表是空的。更新规则中可以有零个（NOTHING关键字）、一个或多个动作。为简单起见，先看一个只有一个动作的规则。这个规则可以有条件或者没有条件，并且它可以是INSTEAD或ALSO（缺省）。

什么是规则条件？它是一个限制，告诉规则动作什么时候做、什么时候不做。这个条件只能引用NEW和/或OLD伪关系，它们基本上代表作为对象给定的关系（但是有着特殊含义）。

所以，对这个单动作的规则生成下面的查询树，有三种情况。

没有条件，有ALSO或INSTEAD

来自规则动作的查询树，在其上增加原始查询树的条件

给出了条件，有ALSO

来自规则动作的查询树，在其上加入规则条件和原始查询树的条件

给出了条件，有INSTEAD

来自规则动作的查询树，在其上加入规则条件和原始查询树的条件；以及带有反规则条件的原始查询树

最后，如果规则是ALSO，那么未修改的原始查询树也被加入到列表。因为只有合格的INSTEAD规则已经被加入到原始查询树中，对于单动作的规则，将结束于一个或两个输出查询树。

对于ON INSERT规则，原始查询（如果没有被INSTEAD取代）是在任何规则增加的动作之前完成的。这样就允许动作看到被插入的行。但是对ON UPDATE 和ON DELETE规则，原始查询是在规则增加的动作之后完成的。这样就确保动作可以看到将要更新或者将要删除的行；否则，动作可能什么也不做，因为它们无法发现符合它们要求的行。

从规则动作生成的查询树会被再次丢给重写系统，并且可能有更多规则被应用而得到更多或更少的查询树。所以一个规则的动作必须有一种不同的命令类型或者和规则所在的关系不同的另一个结果关系。 否则这样的递归处理就会没完没了（规则的递规展开会被检测到，并当作一个错误报告）。

在ux_rewrite系统目录中的动作中的查询树只是模板。因为它们可以引用NEW和OLD的范围表项，在使用它们之前必须做一些替换。对于任何NEW的引用，都要先在原始查询的目标列表中搜索对应的项。如果找到，该项的表达式将会替换该引用。否则NEW和OLD的含义一样（对于UPDATE）或者被替换成一个空值（对于INSERT）。 任何对OLD的引用都用结果关系的范围表项的引用替换。

在系统完成应用更新规则后，它再应用视图规则到生成的查询树上。视图无法插入新的更新动作，所以没有必要向视图重写的输出应用更新规则。

#### 4.4.1.1. 第一个规则循序渐进

假设想要跟踪shoelace_data关系中的sl_avail列。所以建立一个日志表和一条规则，这条规则每次在shoelace_data上执行UPDATE时有条件地写入一个日志项。

```
CREATE TABLE shoelace_log (
   sl_name    text,         -- 改变的鞋带
   sl_avail   integer,      -- 新的可用值
   log_who    text,         -- 谁做的
   log_when   timestamp     -- 何时做的
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
   WHERE NEW.sl_avail <> OLD.sl_avail
   DO INSERT INTO shoelace_log VALUES (
                              NEW.sl_name,
                              NEW.sl_avail,
                              current_user,
                              current_timestamp
                            );
```

当前有人执行如下操作。

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

查看日志表，如下所示。

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

这就是所期望的。在后台发生的的事情如下。解析器创建查询树，如下所示。

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

这是一个带有规则条件表达式的ON UPDATE规则log_shoelace ，条件如下所示。

```
NEW.sl_avail <> OLD.sl_avail
```

动作如下所示。

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old;
```

（这看起来有点奇怪，因为通常不能写INSERT ... VALUES ... FROM。这里的FROM子句只是表示查询树里有用于new和old的范围表项。这些东西是必需的，这样它们就可以被INSERT命令的查询树中的变量引用）。

该规则是一个有条件的ALSO规则，所以规则系统必须返回两个查询树：更改过的规则动作和原始查询树。在第 1 步里，原始查询的范围表被集成到规则动作的查询树中。如下所示。

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data;
```

第 2 步把规则条件增加进去，所以结果集被限制为sl_avail改变了的行，如下所示。

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail;
```

（这看起来更奇怪，因为INSERT ... VALUES也没有WHERE子句，但是规划器和执行器处理它没有任何难度。不管怎样，它们需要为INSERT ... SELECT支持这种相同功能）。

第 3 步把原始查询树的条件加进去，把结果集进一步限制成只有被初始查询树改变的行，如下所示。

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

第 4 步把NEW引用替换为来自原始查询树的目标列表项或来自结果关系的相匹配的变量引用，如下所示。

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
 WHERE 6 <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

第 5 步，用结果关系引用把OLD引用替换掉，如下所示。

```
INSERT INTO shoelace_log VALUES (
      shoelace_data.sl_name, 6,
      current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
      shoelace_data shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

这就完成了。因为规则是ALSO，还要输出原始查询树。简而言之，从规则系统输出的是一个包含两个查询树的列表，它们与下面语句相对应。

```
INSERT INTO shoelace_log VALUES (
      shoelace_data.sl_name, 6,
      current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

这些会按照这个顺序被执行，并且这也正是规则要做的事情。

做的替换和追加的条件用于确保对于下面这样的原始查询不会有日志记录被写入，如下所示。

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

在这种情况下，原始查询树不包含sl_avail的目标列表项，因此NEW.sl_avail将被shoelace_data.sl_avail代替。所以，规则生成的额外命令，如下所示。

```
INSERT INTO shoelace_log VALUES (
      shoelace_data.sl_name, shoelace_data.sl_avail,
      current_user, current_timestamp )
  FROM shoelace_data
 WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

并且条件将永远不可能为真。

如果原始查询修改多个行，这也能争产工作。所以如果某人发出命令，如下所示。

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

实际上有四行（sl1、sl2、sl3和sl4）被更新。但sl3已经是sl_avail = 0。在这种情况下，原始查询树的条件不同并且导致规则产生额外的查询树，如下所示。

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
      current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

这个查询树将肯定插入三个新的日志项。这也是完全正确的。

到这里就能明白为什么原始查询树最后执行非常重要。如果UPDATE先被执行，则所有的行都已经被设为零，所以记日志的INSERT将无法找到任何符合0 <> shoelace_data.sl_avail的行。

## 4.4.2. 与视图合作

要保护一个视图关系不被INSERT、UPDATE或DELETE，一种简单的方法是让那些查询树被丢掉。创建规则，如下所示。

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
   DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
   DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
   DO INSTEAD NOTHING;
```

如果现在某人尝试对视图关系shoe做任何这些操作，规则系统将应用这些规则。因为这些规则没有动作而且是INSTEAD，作为的查询树列表将是空的并且整个查询将变得什么也不做，因为经过规则系统处理后没有什么东西剩下来被优化或执行了。

一个更好的使用规则系统的方法是创建一些规则，这些规则把查询树重写成一个在真实表上进行正确的操作的查询树。要在视图shoelace上做这件事，创建如下规则。

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
   DO INSTEAD
   INSERT INTO shoelace_data VALUES (
         NEW.sl_name,
         NEW.sl_avail,
         NEW.sl_color,
         NEW.sl_len,
         NEW.sl_unit
   );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
   DO INSTEAD
   UPDATE shoelace_data
     SET sl_name = NEW.sl_name,
         sl_avail = NEW.sl_avail,
         sl_color = NEW.sl_color,
         sl_len = NEW.sl_len,
         sl_unit = NEW.sl_unit
    WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
```

```
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

如果要在视图上支持RETURNING查询，需要让规则包含RETURNING子句来计算视图行。这对于基于单个表的视图来说通常非常简单，但是对于连接视图（如shoelace）就有点冗长了。对于插入的一个示例，如下所示。

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    )
    RETURNING
           shoelace_data.*,
           (SELECT shoelace_data.sl_len * u.un_fact
            FROM unit u WHERE shoelace_data.sl_unit = u.un_name);
```

注意，这个规则同时支持该视图上的INSERT和INSERT RETURNING查询 — 对于INSERT会简单地忽略RETURNING子句。

现在假设有时一包鞋带抵达了商店，并且随着它有一个大的清单。但是不想每次都手工更新shoelace视图。取而代之的是建立两个小表：一个用来从清单向其中插入东西，另一个则用了一个特殊的技巧。这些东西的创建命令如下所示。

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

现在可以用来自清单的数据填充表shoelace_arrive，如下所示。

```
SELECT * FROM shoelace_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
```

```
 sl6     |       20
 sl8     |       20
(3 rows)
```

快速查看当前的数据，如下所示。

```
SELECT * FROM shoelace;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

现在把到的货鞋带移动，并检查结果。

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;


SELECT * FROM shoelace ORDER BY sl_name;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who| log_when
---------+----------+--------+----------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

从一个INSERT ... SELECT到这些结果经过了很长的过程。并且该查询树转换的描述将出现在本章的最后。首先，这里是解析器的输出，如下所示。

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

现在应用第一条规则shoelace_ok_ins被应用并且把这个输出进行转换，如下所示。

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

并且丢掉shoelace_ok上的INSERT。这个被重写后的查询被再次传递给规则系统，并且第二个被应用的规则shoelace_upd会产生，如下所示。

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

同样这是一个INSTEAD规则并且前一个查询树会被丢弃掉。注意这个查询仍然使用视图shoelace。但是规则系统还没有完成这一步，所以它会继续并在其上应用_RETURN规则，并且会得到如下内容。

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

最后，规则log_shoelace被应用，生成额外的查询树，如下所示。

```
INSERT INTO shoelace_log
```

```
SELECT s.sl_name,
      s.sl_avail + shoelace_arrive.arr_quant,
      current_user,
      current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
      shoelace_ok old, shoelace_ok new,
      shoelace shoelace, shoelace old,
      shoelace new, shoelace_data shoelace_data,
      shoelace old, shoelace new,
      shoelace_data s, unit u,
      shoelace_data old, shoelace_data new
      shoelace_log shoelace_log
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

完成这些之后，规则系统用完了所有的规则并且返回生成的查询树。

所以结束于两个最终查询树，它们等效于SQL语句，如下所示。

```
INSERT INTO shoelace_log
SELECT s.sl_name,
      s.sl_avail + shoelace_arrive.arr_quant,
      current_user,
      current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
      shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
      shoelace_data shoelace_data,
      shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

结果是从一个关系来的数据插入了到另一个中，改变成第三个上的更新，改变成更新第四个外加做日志，在第五个中的最后更新缩减为两个查询。

有一个小细节有点丑陋。看看那两个查询，会发现shoelace_data关系在范围表中出现了两次而实际上绝对可以缩为出现一次。规划器不会处理它，因此INSERT的规则系统输出的执行规划如下所示。

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
```

```
            ->  Seq Scan on shoelace_arrive
 ->  Seq Scan on shoelace_data
```

在省略额外的范围表项后，如下所示。

```
Merge Join
 ->  Seq Scan
     ->  Sort
         ->  Seq Scan on s
 ->  Seq Scan
     ->  Sort
         ->  Seq Scan on shoelace_arrive
```

这在日志表中生成完全一样的项。因此，规则系统导致了shoelace_data表上的一次绝对不必要的扫描。并且同样的冗余扫描会在UPDATE中进行。但是要把这些全部实现实在是一项很困难的工作。

现在对UXsinoDB规则系统及其能力做最后一个演示。假设向数据库中添加一些有特别颜色的鞋带，如下所示。

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

想要建立一个视图来检查哪些shoelace项在颜色上不配任何鞋子。适用的视图如下所示。

```
CREATE VIEW shoelace_mismatch AS
   SELECT * FROM shoelace WHERE NOT EXISTS
      (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

运行结果如下所示。

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

现在建立它，这样没有库存的不匹配的鞋带都会被从数据库中删除。为了对UXsinoDB有点难度，不直接删除它们。而是再创建一个视图，如下所示。

```
CREATE VIEW shoelace_can_delete AS
   SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

然后用下面方法：

```
DELETE FROM shoelace WHERE EXISTS
   (SELECT * FROM shoelace_can_delete
          WHERE sl_name = shoelace.sl_name);
```

Voilà:

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |        80
 sl2     |        6 | black    |    100 | cm      |       100
 sl7     |        6 | brown    |     60 | cm      |        60
 sl4     |        8 | black    |     40 | inch    |     101.6
 sl3     |       10 | black    |     35 | inch    |      88.9
 sl8     |       21 | brown    |     40 | inch    |     101.6
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
 sl5     |        4 | brown    |      1 | m       |       100
 sl6     |       20 | brown    |    0.9 | m       |        90
(9 rows)
```

对一个视图上的DELETE，这个命令带有一个总共使用了四个嵌套/连接视图的子查询条件， 这四个视图之一本身有一个包含一个视图的子查询条件，该条件计算使用的视图列； 这个命令被重写成了一个查询树，该查询树从一个真正的表里面把需要删除的数据删除。

在现实世界里只有很少的情况需要上面的这样的构造。

# 4.5. 规则和权限

由于UXsinoDB规则系统对查询的重写，会访问没有在原始查询中指定的表/视图。使用更新规则时，这可能包括对表的写权限。

重写规则并不拥有一个独立的所有者。关系（表或视图）的所有者自动成为为其所定义的重写规则的所有者。UXsinoDB规则系统改变了默认的访问控制系统的行为。由于规则被使用的关系会按照规则所有者的权限来检查，而不是调用规则的用户。这表示用户只需要在其查询中显式指定的表/视图上的所需权限。

某用户有一个电话号码列表，其中一些是私人的，另外的一些是办公室助理需要的。该用户可以构建如下内容。

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
   SELECT person, CASE WHEN NOT private THEN phone END AS phone
   FROM phone_data;
GRANT SELECT ON phone_number TO assistant;
```

除了该用户以外（还有数据库超级用户）没有人可以访问phone_data表。但因为GRANT的原因，助理可以在phone_number视图上运行SELECT。规则系统将把phone_number上的SELECT重写为phone_data上的SELECT。因为该用户是phone_number的所有者，因此也是规则的所有者，对phone_data的读访问现在被根据该用户的权限检查，并且该查询被允许。同时也要检查访问phone_number的权限，但这是针对调用用户进行的，所以除了用户自己和助理外没有人可以使用它。

权限检查是按规则逐条进行的。所以此时助理是唯一的一个可以看到公共电话号码的人。 但助理可以建立另一个视图并且赋予该视图公共权限。这样，任何人都可以通过助理的视图看到phone_number数据。 助理不能做的事情是创建一个直接访问phone_data的视图（实际上助理是

可以的，但没有任何作用，因为每次访问都会因通不过权限检查而被否定）。而且该用户一旦注意到助理开放了他的phone_number视图，该用户还可以收回助理的访问权限。立刻，所有对助理视图的访问将会失败。

有人可能会认为这种逐条规则的检查是一个安全漏洞，但事实上不是。 如果这样做不能奏效，助理将必须建立一个与phone_number有相同列的表并且每天拷贝一次数据进去。 那么这是助理自己的数据因而助理可以为每一个想要访问的人授权。一个GRANT意味着“信任”。 如果某个信任的人做了上面的事情，那么是时候认为信任已经结束并且要使用 REVOKE。

需要注意的是，虽然视图可以用前文展示的技术来隐藏特定列的内容，它们不能可靠地在不可见行上隐藏数据，除非标志被设置。 例如，下面的视图是不安全的。

```
CREATE VIEW phone_number AS
   SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

这个视图看起来是安全的，因为规则系统会把任何phone_number上的SELECT重写成 phone_data上的SELECT，并且增加限制使得只有phone 不以 412 开头的项才被处理。但是如果用户可以创建自己的函数，那就不难让规划器在NOT LIKE表达式之前先执行用户自定义函数。如下所示。

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
   RAISE NOTICE '% => %', $1, $2;
   RETURN true;
END
$$ LANGUAGE pluxsql COST 0.00000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

phone_data表中的每一个人和电话号码会被打印成一个NOTICE，因为规划器会选择在执行NOT LIKE之前先执行tricky，因为前者的开销大。 即使禁止用户自定义一个新函数，内置函数也可以用在类似的攻击中（例如，大部分造型函数会在它们产生的错误信息中包含它们的输入值）。

类似的考虑应用于更新规则。在前一节的示例中，示例数据库中表的所有者可以把shoelace视图上的SELECT、INSERT、UPDATE和DELETE权限授予其他人，但对shoelace_log只有SELECT权限。写日志项的规则动作将仍然可以被成功地执行，并且其它用户可以看到日志项。但他们不能创建伪造的项，并且他们也不能操纵或移除现有的项。在这种情况下，不可能通过让规划器改变操作的顺序来推翻规则，因为引用shoelace_log的唯一规则是无限制的INSERT。在更复杂的情景中，这可能不正确。

当需要对一个视图提供行级安全时，security_barrier属性应该被应用到该视图。这会阻止恶意选择的函数和操作符通过行被传递，直到视图完成其工作。例如，如果前文所示的视图被创建成这样，它就是安全的。

```
CREATE VIEW phone_number WITH (security_barrier) AS
   SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

Views created with the 使用security_barrier创建的视图的性能会远差于没有使用该选项的视图。通常，没有办法来避免这种现状：如果最快的候选计划可能在安全性上折衷，它就必须被拒绝。出于该原因，这个选在在默认情况下是没有启用的。

当处理没有副作用的函数时，查询规划器有更多的灵活性。这类函数被称为LEAKPROOF，并且包括很多简单常用的操作符，例如很多等于操作符。查询规划器可以安全地允许这类函数在查询

执行过程中的任何点被计算，因为在用户不可见的行上调用它们将不会泄露关于不可见行的任何信息。更进一步，不接收参数或者不从安全屏障视图得到任何参数的函数不必被标记为LEAKPROOF以便被下推，因为它们从来不会从该视图接收数据。相反，一个可能会基于接收到的参数值抛出错误的函数（例如在溢出或被零除事件中抛出错误的函数）不是防泄漏的，并且如果它被应用在安全性视图的行过滤器之前，它可能会提供有关不可见行的有效信息。

有一点很重要的是理解：即使一个视图使用security_barrier选项创建，它也只在不可见元组不会被传递给可能不安全的函数的前提下才是安全的。用户可能也有其他方式来推断不可见数据；例如，他们可以使用EXPLAIN看到查询计划，或者针对视图来测量查询的运行时间。一个恶意攻击者可能有能力推断有关不可见数据的总量，或者甚至得到有关数据分布的某些信息或最常用值（因为这些东西可以影响计划的运行时间；或者甚至计划的选择，因为它们也被反映在优化器的统计数据中）。如果这类“隐通道”攻击很重要，那么授予任何到该数据的访问都可能是不明智的。

# 4.6. 规则和命令状态

UXsinoDB服务器为它收到的每个命令返回一个命令状态字符串，例如INSERT 149592 1。没有涉及规则时这很简单，但是查询被规则重写时会发生什么呢？

规则对命令状态的影响如下所示。

- 如果没有查询的无条件INSTEAD规则，那么原始给出的查询将会被执行，并且它的命令状态将像平常一样被返回（但是请注意如果存在任何有条件INSTEAD规则，那么它们的反条件将被加到原始查询中。这样可能会减少它处理的行数，并且报告的状态将受影响）。
- 如果有查询的任何无条件INSTEAD规则，那么原始查询将完全不被执行。在这种情况下，服务器将返回由服务器将返回由INSTEAD规则（有条件的或无条件的）插入的最后一条和原始查询命令类型（INSERT、UPDATE或DELETE）相同的查询的命令状态。如果任何规则添加的查询都不符合这些要求，那么返回的命令状态显示原始查询类型并且行计数和 OID 域为零。

通过为任何想要的INSTEAD规则指定在活动规则中排名最后的规则名，程序员可以确保该规则都是在第二种情况里设置命令状态的规则，因为它会被最后一个应用。

# 4.7. 规则 vs 触发器

许多触发器可以干的事情同样也可以用UXsinoDB规则系统来实现。目前不能用规则来实现的东西之一是某些约束，特别是外键。 可以放置一个合格的规则在一列上，这个规则在列的值没有出现在另一个表中时把命令重写成NOTHING。但是这样做数据就会被不声不响地丢弃，因此也不是一个好主意。如果要求检查值的有效性，并且在出现无效值的情况下应该生成一个错误消息，这种需求就必须要用触发器来完成。

在本章中，关注于使用规则来更新视图。本章中所有的更新规则的示例都可以使用视图上的INSTEAD OF触发器来实现。编写这类触发器通常比编写规则要容易，特别是在要求使用复杂逻辑来执行更新的情况下。

对于两者都可实现的情况，哪个更好取决于对数据库的使用。触发器为每一个受影响的行都执行一次。规则修改查询树或生成一个额外的查询。所以如果在一个语句中影响到很多行， 一个发出额外查询的规则通常可能会比一个触发器快，因为触发器对每一个行都要被调用，并且每次被调用时都需要重新判断要做什么样的操作。不过，触发器方法从概念上要远比规则方法简单，并且很容易让新人上手。

下面展示一个示例，该示例说明了在同种情况下两种选择的比较。这里有两个表，如下所示。

```
CREATE TABLE computer (
    hostname        text,    -- 被索引
    manufacturer    text     -- 被索引
);

CREATE TABLE software (
    software        text,    -- 被索引
    hostname        text     -- 被索引
);
```

两个表都有数千行，并且在*hostname*上的索引是唯一的。规则或触发器应该实现一个约束，该约束从software中删除引用已删除计算机的行。 触发器可以用下面这条命令。

```
DELETE FROM software WHERE hostname = $1;
```

因为触发器会为每一个从computer中删除的独立行调用一次， 那么它可以准备并且保存这个命令的规划，把*hostname*作为参数传入。规则如下所示。

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

现在看看不同类型的删除。如下这种情况中，表computer被使用索引（快速）扫描，并且由触发器发出的命令也将使用一个索引扫描（同样快速）。

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

来自规则的额外查询，如下所示。

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                     AND software.hostname = computer.hostname;
```

由于已经建立了合适的索引，规划器将创建一个规划，所以在触发器和规则的实现之间没有太多的速度差别。

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

在接下来的删除中，想要去掉所有 2000 个*hostname*以old开头的计算机。有两个命令可以来做这件事。

- 命令一

  ```
  DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
  ```

  被规则增加的命令如下所示。

  ```
  DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
  ```

```
                AND software.hostname = computer.hostname;
```

计划如下所示。

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

- 命令二

```
DELETE FROM computer WHERE hostname ~ '^old';
```

它会为规划增加的命令产生下面的执行计划。

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

这表明，当有多个条件表达式被使用AND组合在一起时，规划器不能认识到表computer中*hostname*上的条件也可以被用于一个software上的索引扫描， 而在该命令的正则表达式版本中正是这样做的。触发器将为要被删除的 2000 个旧计算机中的每一个调用，并且会导致在computer上的一次索引扫描和software上的 2000 次索引扫描。采用规则的实现将会使用两个使用索引的命令来完成。 并且在顺序扫描情况下规则是否仍将更快是取决于software表的总体大小的。即使所有的索引块都将很快地进入高速缓存，通过 SPI 管理器执行来自触发器的 2000 个命令也要花不少时间。

想要看到最后一个命令，如下所示。

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

同样，这也会导致很多行被从computer中删除。所以触发器同样会通过执行器运行很多命令。规则生成的命令如下所示。

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
                AND software.hostname = computer.hostname;
```

这个命令的计划又将是在两个索引扫描上的嵌套循环，只不过使用了computer上的另一个索引，如下所示。

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

在任何这些情况之一，来自规则系统的额外命令都或多或少与命令中影响的行数无关。

概括来说，规则只有在其动作导致了大而且糟糕的条件连接时才会明显地慢于触发器，这种情况下规划器将没有什么办法。

# 第 5 章 过程语言

UXsinoDB允许使用除了 SQL 和 C 之外的其他语言编写用户定义的函数。这些其他的语言通常被称作*过程语言*（PL）。对于一个用过程语言编写的函数，数据库服务器没有关于如何解释该函数的源文本的内建知识。因此，这个任务被交给一个了解语言细节的特殊处理器。该处理器能够自己处理所有的解析、语法分析、执行工作，或者它可以作为一种UXsinoDB和编程语言既有实现之间的“粘合剂”。就像任何其他 C 函数一样，处理器本身是一个编译到共享对象并且按需载入的 C 语言函数。

在UXsinoDB的标准发布中当前有四种过程语言可用：PL/uxSQL（第 6 章 PL/uxSQL - SQL过程语言）、 PL/Tcl（第 7 章 PL/Tcl - Tcl 过程语言、PL/Perl（第 8 章 PL/Perl - Perl 过程语言）以及 PL/Python（第 9 章 PL/Python - Python 过程语言。还有其他过程语言可用，但是它们没有被包括在核心发布中。

## 5.1. 安装过程语言

在每一个要使用过程语言的数据库中都必须“安装”相应的过程语言。不过安装在数据库template1中的过程语言会被后续创建的数据库自动继承，因为template1中与过程语言相关的项会被**CREATE DATABASE**复制。因此，数据库管理员可以决定在哪些数据库中可以使用哪些语言，并且按照选择让一些语言默认可用。

对于标准发布所提供的语言，只需要执行**CREATE EXTENSION** *language_name*来把该语言安装在当前数据库中。 下文所述的手工过程主要是为了安装没有被包装成扩展的语言。

### 过程 5.1. 手工安装过程语言

安装一个过程语言到一个数据库中包括五个步骤，且必须由一个数据库超级用户来执行。在大部分情况下，所需的 SQL 命令应该被打包成一个“扩展”的安装脚本，这样可以用**CREATE EXTENSION**来执行它们。

1. 用于语言处理器的共享对象必须被编译并安装到一个合适的库目录中。这和编译和安装常规的用户定义 C 函数一样，参见第 1.10.5 节 “编译和链接动态载入的函数”。通常，语言处理器将依赖于一个实际提供编程语言引擎的外部库，如果是这样，那些外部库也应该被安装。

2. 处理器必须用下面的命令声明。

   ```
   CREATE FUNCTION handler_function_name()
     RETURNS language_handler
     AS 'path-to-shared-object'
     LANGUAGE C;
   ```

   特殊的返回类型language_handler告诉数据库系统，这个函数不返回已定义的SQL数据类型，并且不能直接在SQL语句中使用。

3. （可选的） 可选地，语言处理器能提供一个“内联”处理器函数来执行用这种语言编写的匿名代码块（DO命令）。如果该语言提供了一个内联函数，用类似下面的命令声明它。

   ```
   CREATE FUNCTION inline_function_name(internal)
     RETURNS void
     AS 'path-to-shared-object'
   ```

```
  LANGUAGE C;
```

4. （可选的） 可选地，语言处理器能提供一个“验证器”函数用来检查一个函数定义的正确性而无需实际执行它。如果验证器函数存在，它将被CREATE FUNCTION调用。如果该语言提供了一个验证器函数，用类似下面的命令声明它。

```
CREATE FUNCTION validator_function_name(oid)
  RETURNS void
  AS 'path-to-shared-object'
  LANGUAGE C STRICT;
```

5. 最后，PL 必须用下面的命令声明。

```
CREATE [TRUSTED] LANGUAGE language_name
  HANDLER handler_function_name
  [INLINE inline_function_name]
  [VALIDATOR validator_function_name] ;
```

可选的关键词TRUSTED指定，如果用户不具有访问数据的权限，该语言不会授予对数据的访问。可信的语言是为普通数据库用户（没有超级用户特权）设计的并且允许他们安全地创建函数和过程。由于 PL 函数是在数据库内部执行的，TRUSTED标志只应被给予不允许访问数据库服务器内部或文件系统的语言。语言 PL/uxSQL、 PL/Tcl以及 PL/Perl被认为是可信的，语言 PL/TclU、 PL/PerlU以及 PL/PythonU是被设计用来提供无限制功能的并且不应该被标记为可信。

例 5.1 “PL/Perl的手工安装”展示了手工安装过程如何安装语言PL/Perl。

## 例 5.1. PL/Perl的手工安装

下列命令告诉数据库服务器在哪里寻找用于PL/Perl语言调用处理器函数的共享对象。

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
  '$libdir/plperl' LANGUAGE C;
```

PL/Perl有一个内联处理器函数和一个验证器函数，因此要声明它们，如下所示。

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
  '$libdir/plperl' LANGUAGE C;

CREATE FUNCTION plperl_validator(oid) RETURNS void AS
  '$libdir/plperl' LANGUAGE C STRICT;
```

命令

```
CREATE TRUSTED LANGUAGE plperl
  HANDLER plperl_call_handler
  INLINE plperl_inline_handler
  VALIDATOR plperl_validator;
```

则定义了前面声明的函数会为语言属性为plperl的函数及过程所调用。

在一个默认的UXsinoDB安装中，用于PL/uxSQL语言的处理器会被编译并且安装到“library”目录，此外PL/uxSQL语言本身会被安装在所有的数据库中。如果配置了Tcl支持，用于PL/Tcl和PL/TclU的处理器会被编译并且安装到库目录中，但语言本身默认不会被安装在任何数据库中。同样地，PL/Perl和PL/PerlU处理器在配置了 Perl 支持时被编译和安装，并且PL/PythonU处理器在配置了 Python 支持时被安装，但是这些语言默认都不会被安装。

# 第 6 章 PL/uxSQL - SQL过程语言

## 6.1. 综述

PL/uxSQL是一种用于UXsinoDB数据库系统的可载入的过程语言。PL/uxSQL的设计目标是创建一种这样的可载入过程语言。

- 可以被用来创建函数和触发器过程，
- 对SQL语言增加控制结构，
- 可以执行复杂计算，
- 继承所有用户定义类型、函数和操作符，
- 可以被定义为受服务器信任，
- 便于使用。

用PL/uxSQL创建的函数可以被用在任何可以使用内建函数的地方。例如，可以创建复杂条件的计算函数并且后面用它们来定义操作符或把它们用于索引表达式。

PL/uxSQL是默认被安装的。但是它仍然是一种可载入模块，因此特别关注安全性的管理员可以选择移除它。

### 6.1.1. 使用PL/uxSQL的优点

SQL被UXsinoDB和大多数其他关系数据库用作查询语言。它是可移植的并且容易学习。但是每一个SQL语句必须由数据库服务器单独执行。

这说明客户端应用必须发送每一个查询到数据库服务器、等待它被处理、接收并处理结果、做一些计算，然后发送更多查询给服务器。如果客户端和数据库服务器不在同一台机器上，所有这些会引起进程间通信并且将带来网络负担。

通过PL/uxSQL，可以将一整块计算和一系列查询分组在数据库服务器内部，这样就有了一种过程语言的能力并且使 SQL 更易用，但是节省了相当多的客户端/服务器通信开销。

- 客户端和服务器之间的额外往返通信被消除
- 客户端不需要的中间结果不必被整理或者在服务器和客户端之间传送
- 多轮的查询解析可以被避免

与不使用存储函数的应用相比，这能够导致可观的性能提升。

还有，通过PL/uxSQL可以使用 SQL 中所有的数据类型、操作符和函数。

### 6.1.2. 支持的参数和结果数据类型

PL/uxSQL编写的函数可以接受服务器支持的任何标量或数组数据类型作为参数，并且它们能够返回任何这些类型的结果。它们也能接受或返回任何用名称指定的组合类型（行类型）。还可以声明一个PL/uxSQL函数为接受record，这表示任意组合类型都将作为输入，或者声明为返回record，表示结果是一种行类型，它的列由调用查询中的说明确定。

PL/uxSQL函数可以通过使用VARIADIC标记被声明为接受数量不定的参数。如第 1.5.5 节 “带有可变数量参数的SQL函数”中所讨论的，它的工作方式和 SQL 函数一样。

PL/uxSQL函数也可以被声明为接受并返回多态类型anyelement、anyarray、anynonarray、anyenum以及anyrange。如第 1.2.5 节 “多态类型”中所讨论的，由一个多态函数处理的实际数据类型会随着调用改变。在第 6.3.1 节 “声明函数参数”中展示了一个示例。

PL/uxSQL函数还能够被声明为返回一个任意（可作为一个单一实例返回的）数据类型的“集合”（或表）。这样的一个函数通过为结果集的每个期望元素执行**RETURN NEXT**来产生输出，或者通过使用**RETURN QUERY**来输出一个查询计算的结果。

最后，如果一个PL/uxSQL函数没有可用的返回值，它可以被声明为返回void（另外一种选择是，在那种情况下它可以被写作一个过程）。

PL/uxSQL函数也能够被声明为用输出参数代替返回类型的一个显式说明。这没有为该语言增加任何基础功能，但是它常常很方便，特别是对于要返回多个值的情况。RETURNS TABLE符号也可以被用来替代RETURNS SETOF。

在第 6.3.1 节 “声明函数参数”和第 6.6.1 节 “从一个函数返回”中有详细的示例。

# 6.2. PL/uxSQL的结构

通过执行**CREATE FUNCTION**命令，以PL/uxSQL写成的函数可以被定义到服务器中。这种命令一般如下所示，

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'function body text'
LANGUAGE pluxsql;
```

就目前**CREATE FUNCTION**所关心的来说，函数体就是简单的一个字符串。通常在写函数体时，使用美元符号引用通常比使用普通单引号语法更有帮助。如果没有美元引用，函数体中的任何单引号或者反斜线必须通过双写来转义。这一章中几乎所有的示例都在其函数体中使用美元符号引用。

PL/uxSQL是一种块结构的语言。一个函数体的完整文本必须是一个块。一个块的定义如下所示。

```
[ <<label>> ]
[ DECLARE
   declarations ]
BEGIN
   statements
END [ label ];
```

在一个块中的每一个声明和每一个语句都由一个分号终止。如上所示，出现在另一个块中的块必须有一个分号在END之后。不过最后一个结束函数体的END不需要一个分号。

> **提示**
>
> 一种常见的错误是直接在BEGIN之后写一个分号。这是不正确的并且将会导致一个语法错误。

如果想要标识一个块以便在一个EXIT语句中使用或者标识在该块中声明的变量名，那么*label*是唯一需要的。如果一个标签在END之后被给定，它必须匹配在块开始处的标签。

所有的关键词都是大小写无关的。除非被双引号引用，标识符会被隐式地转换为小写形式，就像它们在普通 SQL 命令中。

PL/uxSQL代码中的注释和普通 SQL 中的一样。一个双连字符（--）开始一段注释，它延伸到该行的末尾。一个/*开始一段块注释，它会延伸到匹配*/出现的位置。块注释可以嵌套。

一个块的语句节中的任何语句可以是一个子块。子块可以被用来逻辑分组或者将变量局部化为语句的一个小组。在子块的持续期间，在一个子块中声明的变量会掩盖外层块中相同名称的变量。但是如果用块的标签限定外层变量的名字，仍然可以访问它们。如下所示。

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
  quantity integer := 30;
BEGIN
  RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
  quantity := 50;
  --
  -- 创建一个子块
  --
  DECLARE
    quantity integer := 80;
  BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
    RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
  END;

  RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50

  RETURN quantity;
END;
$$ LANGUAGE pluxsql;
```

> **注意**
>
> 在任何PL/uxSQL函数体的外部确实有一个隐藏的“外层块”包围着。这个块提供了该函数参数（如果有）的声明，以及某些诸如FOUND之类特殊变量（见第 6.5.5 节 “获得结果状态”）。外层块被标上函数的名称，这意味着参数和特殊变量可以用该函数的名称限定。

重要的是不要把PL/uxSQL中用来分组语句的**BEGIN/END**与用于事务控制的同名 SQL 命令弄混。PL/uxSQL的**BEGIN/END**只用于分组，它们不会开始或结束一个事务。有关PL/uxSQL中管理事务的信息，请参见第 6.8 节 “事务管理”。此外，一个包含EXCEPTION子句的块实际上会形成一个子事务，它可以被回滚而不影响外层事务。详见第 6.6.8 节 “俘获错误”。

# 6.3. 声明

在一个块中使用的所有变量必须在该块的声明小节中声明（唯一的例外是在一个整数范围上迭代的FOR循环变量会被自动声明为一个整数变量，并且相似地在一个游标结果上迭代的FOR循环变量会被自动地声明为一个记录变量）。

PL/uxSQL变量可以是任意 SQL 数据类型，例如integer、varchar和char。

这里是变量声明的一些示例，如下所示。

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

一个变量声明的一般语法，如下所示。

*name* [ CONSTANT ] *type* [ COLLATE *collation_name* ] [ NOT NULL ] [ { DEFAULT | := | = } *expression* ];

如果给定DEFAULT子句，它会指定进入该块时分配给该变量的初始值。如果没有给出DEFAULT子句，则该变量被初始化为SQL空值。 CONSTANT选项阻止该变量在初始化之后被赋值，这样它的值在块的持续期内保持不变。COLLATE选项指定用于该变量的一个排序规则（见 第 6.3.6 节 “PL/uxSQL变量的排序规则”。如果指定了NOT NULL，对该变量赋值为空值会导致一个运行时错误。所有被声明为NOT NULL的变量必须被指定一个非空默认值。等号（=）可以被用来代替 PL/SQL-兼容的:=。

一个变量的默认值会在每次进入该块时被计算并且赋值给该变量（不是每次函数调用只计算一次）。因此，例如将now()赋值给类型为timestamp的一个变量将会导致该变量具有当前函数调用的时间，而不是该函数被预编译的时间。

示例

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

## 6.3.1. 声明函数参数

传递给函数的参数被命名为标识符$1、$2等等。可选地，能够为$*n*参数名声明别名来增加可读性。不管是别名还是数字标识符都能用来引用参数值。

有两种方式来创建一个别名。比较好的方式是在CREATE FUNCTION命令中为参数给定一个名称。如下所示。

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
  RETURN subtotal * 0.06;
END;
$$ LANGUAGE pluxsql;
```

另一种方式是显式地使用声明语法声明一个别名。

*name* ALIAS FOR $*n*;

使用这种风格的同一个示例，如下所示。

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
   subtotal ALIAS FOR $1;
BEGIN
   RETURN subtotal * 0.06;
END;
$$ LANGUAGE pluxsql;
```

> **注意**
>
> 这两个示例并非完全等效。在第一种情况中，subtotal可以被引用为sales_tax.subtotal，但在第二种情况中它不能这样引用（如果为内层块附加了一个标签，subtotal则可以用那个标签限定）。

示例

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
   v_string ALIAS FOR $1;
   index ALIAS FOR $2;
BEGIN
   -- 这里是一些使用 v_string 和 index 的计算
END;
$$ LANGUAGE pluxsql;
```

```
CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
   RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE pluxsql;
```

当一个PL/uxSQL函数被声明为带有输出参数，输出参数可以用普通输入参数相同的方式被给定$*n*名称以及可选的别名。一个输出参数实际上是一个最初为 NULL 的变量，它应当在函数的执行期间被赋值。该参数的最终值就是要被返回的东西。例如，sales-tax也可以使用如下方式。

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
   tax := subtotal * 0.06;
END;
$$ LANGUAGE pluxsql;
```

注意忽略了RETURNS real — 也可以包括它，但是那将是冗余。

当返回多个值时，输出参数最有用。如下所示。

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
   sum := x + y;
```

```
    prod := x * y;
END;
$$ LANGUAGE pluxsql;
```

如第 1.5.4 节 “带有输出参数的SQL函数”中所讨论的，这实际上为该函数的结果创建了一个匿名记录类型。如果给定了一个RETURNS子句，它必须RETURNS record。

声明一个PL/uxSQL函数的另一种方式是用RETURNS TABLE，如下所示。

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales AS s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE pluxsql;
```

这和声明一个或多个OUT参数并且指定RETURNS SETOF *sometype*完全等效。

当一个PL/uxSQL函数的返回类型被声明为一个多态类型（anyelement、anyarray、anynonarray、anyenum或anyrange），一个特殊参数$0会被创建。它的数据类型是该函数的实际返回类型，这是从实际输入类型（第 1.2.5 节“多态类型”）推演得来。$0被初始化为空并且不能被该函数修改，因此它能够被用来保持可能需要的返回值，不过这不是必须的。$0也可以被给定一个别名。例如，这个函数工作在任何具有一个+操作符的数据类型上：

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE pluxsql;
```

通过声明一个或多个输出参数为多态类型可以得到同样的效果。在这种情况下，不使用特殊的$0参数，输出参数本身就用作相同的目的。如下所示。

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE pluxsql;
```

## 6.3.2. ALIAS

*newname* ALIAS FOR *oldname*;

ALIAS语法比前一节中建议的更一般化：可以为任意变量声明一个别名，而不只是函数参数。其主要实际用途是为预先决定了名称的变量分配一个不同的名称，例如在一个触发器过程中的NEW或OLD。

示例

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

因为ALIAS创造了两种不同的方式来命名相同的对象，如果对其使用不加限制就会导致混淆。最好只把它用来覆盖预先决定的名称。

## 6.3.3. 复制类型

*variable*%TYPE

%TYPE提供了一个变量或表列的数据类型。可以用它来声明将保持数据库值的变量。例如，如果在users中有一个名为user_id的列。要定义一个与users.user_id具有相同数据类型的变量，如下所示。

```
user_id users.user_id%TYPE;
```

通过使用%TYPE，不需要知道要引用的结构的实际数据类型，而且最重要地，如果被引用项的数据类型在未来被改变（例如把user_id的类型从integer改为real），不需要改变函数定义。

%TYPE在多态函数中特别有价值，因为内部变量所需的数据类型能在两次调用时改变。可以把%TYPE应用在函数的参数或结果占位符上来创建合适的变量。

## 6.3.4. 行类型

*name table_name*%ROWTYPE;
*name composite_type_name*;

一个组合类型的变量被称为一个行变量（或行类型变量）。这样一个变量可以保持一个SELECT或FOR查询结果的一整行，前提是查询的列集合匹配该变量被声明的类型。该行值的各个域可以使用通常的点号标记访问，例如rowvar.field。

通过使用*table_name*%ROWTYPE标记，一个行变量可以被声明为具有和一个现有表或视图的行相同的类型。它也可以通过给定一个组合类型名称来声明（因为每一个表都有一个相关联的具有相同名称的组合类型，所以在UXsinoDB中实际上写不写%ROWTYPE都没有关系。但是带有%ROWTYPE的形式可移植性更好）。

一个函数的参数可以是组合类型（完整的表行）。在这种情况下，相应的标识符$*n*将是一个行变量，并且可以从中选择域，例如$1.user_id。

这里是一个使用组合类型的示例。table1和table2是已有的表，它们至少有以下提到的域，如下所示。

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
  t2_row table2%ROWTYPE;
BEGIN
  SELECT * INTO t2_row FROM table2 WHERE ... ;
  RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE pluxsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

## 6.3.5. 记录类型

*name* RECORD;

记录变量和行类型变量类似，但是它们没有预定义的结构。它们采用在一个SELECT或FOR命令期间为其赋值的行的真实行结构。一个记录变量的子结构能在每次它被赋值时改变。这样的结果是直到一个记录变量第一次被赋值之前，它都没有子结构，并且任何尝试访问其中一个域都会导致一个运行时错误。

注意RECORD并非一个真正的数据类型，只是一个占位符。也应该认识到当一个PL/uxSQL函数被声明为返回类型record，这与一个记录变量并不是完全相同的概念，即便这样一个函数可能会用一个记录变量来保持其结果。在两种情况下，编写函数时都不知道真实的行结构，但是对于一个返回record的函数，当调用查询被解析时就已经决定了真正的结构，而一个行变量能够随时改变它的行结构。

## 6.3.6. PL/uxSQL变量的排序规则

当一个PL/uxSQL函数有一个或多个可排序数据类型的参数时，为每一次函数调用都会基于赋值给实参的排序规则来确定出一个排序规则。如果一个排序规则被成功地确定（即在参数之间隐式排序规则没有冲突），那么所有的可排序参数会被当做隐式具有那个排序规则。这将在函数中影响行为受到排序规则影响的操作。例如，考虑less_than的第一次使用将会采用*text_field_1*和*text_field_2*共同的排序规则进行比较，而第二次使用将采用C排序规则。如下所示。

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
  RETURN a < b;
END;
$$ LANGUAGE pluxsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;
```

此外，被确定的排序规则也被假定为任何可排序数据类型本地变量的排序规则。因此，当这个函数被写为以下形式时，它工作将不会有什么不同。

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
  local_a text := a;
  local_b text := b;
```

```
BEGIN
   RETURN local_a < local_b;
END;
$$ LANGUAGE pluxsql;
```

如果没有可排序数据类型的参数，或者不能为它们确定共同的排序规则，那么参数和本地变量会使用它们数据类型的默认排序规则（通常是数据库的默认排序规则，但是可能不同于域类型的变量）。

通过在一个可排序数据类型的本地变量的声明中包括COLLATE选项，可以为它指定一个不同的排序规则，如下所示。

```
DECLARE
   local_a text COLLATE "en_US";
```

这个选项会覆盖根据上述规则被给予该变量的排序规则。

还有，如果一个函数想要强制在一个特定操作中使用一个特定排序规则，当然可以在该函数内部写一个显式的COLLATE子句。如下所示。

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
   RETURN a < b COLLATE "C";
END;
$$ LANGUAGE pluxsql;
```

这会覆盖表达式中使用的表列、参数或本地变量相关的排序规则，就像在纯 SQL 命令中发生的一样。

# 6.4. 表达式

PL/uxSQL语句中用到的所有表达式会被服务器的主SQL执行器处理。例如，当写一个这样的PL/uxSQL语句时

IF *expression* THEN ...

PL/uxSQL将通过给主 SQL 引擎发送一个查询来计算该表达式，如下所示。

SELECT *expression*

如第 6.11.1 节 “变量替换”中所详细讨论的，在构造该SELECT命令时，PL/uxSQL变量名的每一次出现会被参数所替换。这允许SELECT的查询计划仅被准备一次并且被重用于之后的对于该变量不同值的计算。因此，在一个表达式第一次被使用时实际发生的本质上是一个PREPARE命令。例如，如果已经声明了两个整数变量x和y，并且写了

IF x < y THEN ...

在现象之后发生的等效于

PREPARE *statement_name*(integer, integer) AS SELECT $1 < $2;

并且然后为每一次IF语句的执行，这个预备语句都会被EXECUTE，执行时使用变量的当前值作为参数值。通常这些细节对于一个PL/uxSQL用户并不重要，但是在尝试诊断一个问题时了解它们很有用。更多信息可见第 6.11.2 节 “计划缓存”。

# 6.5. 基本语句

在这一节和接下来的小节中，会描述PL/uxSQL能明确理解的所有语句类型。任何不被识别为这些语句类型之一的被假定为是一个 SQL 命令，并且会被发送给主数据库引擎执行，具体如第 6.5.2 节 “执行一个没有结果的命令”和第 6.5.3 节 “执行一个有单一行结果的查询”中所述。

## 6.5.1. 赋值

为一个PL/uxSQL变量赋一个值，如下所示。

*variable* { := | = } *expression*;

正如以前所解释的，这样一个语句中的表达式被以一个 SQLSELECT命令被发送到主数据库引擎的方式计算。该表达式必须得到一个单一值（如果该变量是一个行或记录变量，它可能是一个行值）。该目标变量可以是一个简单变量（可以选择用一个块名限定）、一个行或记录变量的域或是一个简单变量或域的数组元素。等号（=）可以被用来代替 PL/SQL-兼容的 :=。

如果该表达式的结果数据类型不匹配变量的数据类型，该值将被强制为变量的类型，就好像做了赋值造型一样。如果没有用于所涉及到的数据类型的赋值造型可用，PL/uxSQL解释器将尝试以文本的方式转换结果值，也就是在应用结果类型的输出函数之后再应用变量类型的输入函数。注意如果结果值的字符串形式无法被输入函数所接受，这可能会导致由输入函数产生的运行时错误。

示例

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

## 6.5.2. 执行一个没有结果的命令

对于任何不返回行的 SQL 命令（例如没有一个RETURNING子句的INSERT），可以通过把该命令直接写在一个 PL/uxSQL 函数中执行它。

任何出现在该命令文本中的PL/uxSQL变量名被当作一个参数，并且接着该变量的当前值被提供为运行时该参数的值。这与早前描述的对表达式的处理完全相似，详见第 6.11.1 节 “变量替换”。

当以这种方式执行一个 SQL 命令时，如第 6.11.2 节 “计划缓存”中讨论的，PL/uxSQL会为该命令缓存并重用执行计划。

有时候计算一个表达式或SELECT查询但抛弃其结果是有用的，例如调用一个有副作用但是没有有用的结果值的函数。在PL/uxSQL中要这样做，可使用PERFORM语句，如下所示。

PERFORM *query*;

这会执行*query*并且丢弃掉结果。以写一个SQL SELECT命令相同的方式写该*query*，并且将初始的关键词SELECT替换为PERFORM。对于WITH查询，使用PERFORM并且接着把该查询放在圆括号中（在这种情况中，该查询只能返回一行）。PL/uxSQL变量将被替换到该查询中，正像对不返回结果的命令所作的那样，并且计划被以相同的方式被缓存。还有，如果该查询产生至少一行，特殊变量FOUND会被设置为真，而如果它不产生行则设置为假（见第 6.5.5 节 “获得结果状态”）。

> **注意**
>
> 可能期望直接写SELECT能实现这个结果，但是当前唯一被接受的方式是PERFORM。一个能返回行的 SQL 命令（例如SELECT）将被当成一个错误拒绝，除非它像下一节中讨论的有一个INTO子句。

示例

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

## 6.5.3. 执行一个有单一行结果的查询

一个产生单一行（可能有多个列）的 SQL 命令的结果可以被赋值给一个记录变量、行类型变量或标量变量列表。这通过书写基础 SQL 命令并增加一个INTO子句来达成。如下所示。

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

其中*target*可以是一个记录变量、一个行变量或一个有逗号分隔的简单变量和记录/行域列表。PL/uxSQL变量将被替换到该查询的剩余部分中，并且计划会被缓存，正如之前描述的对不返回行的命令所做的。这对SELECT、带有RETURNING的INSERT/UPDATE/DELETE以及返回行集结果的工具命令（例如EXPLAIN）。除了INTO子句，SQL 命令和它在PL/uxSQL之外的写法一样。

> **提示**
>
> 注意带INTO的SELECT的这种解释和UXsinoDB常规的SELECT INTO命令有很大的不同，后者的INTO目标是一个新创建的表。如果想要在一个PL/uxSQL函数中从一个SELECT的结果创建一个表，请使用语法CREATE TABLE ... AS SELECT。

如果一行或一个变量列表被用作目标，该查询的结果列必须完全匹配该结果的结构，包括数量和数据类型，否则会发生一个运行时错误。当一个记录变量是目标时，它会自动地把自身配置成查询结果列组成的行类型。

INTO子句几乎可以出现在 SQL 命令中的任何位置。通常它被写成刚好在SELECT命令中的*select_expressions*列表之前或之后，或者在其他命令类型的命令最后。推荐遵循这种惯例，以防PL/uxSQL的解析器在未来的版本中变得更严格。

如果STRICT没有在INTO子句中被指定，那么*target*将被设置为该查询返回的第一个行，或者在该查询不返回行时设置为空（注意除非使用了ORDER BY，否则“第一行”的界定并不清楚）。第一行之后的任何结果行都会被抛弃。可以检查特殊的FOUND变量（见第 6.5.5 节 “获得结果状态”）来确定是否返回了一行，如下所示。

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
   RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

如果指定了STRICT选项，该查询必须刚好返回一行或者将会报告一个运行时错误，该错误可能是NO_DATA_FOUND（没有行）或TOO_MANY_ROWS（多于一行）。如果希望捕捉该错误，可以使用一个异常块，如下所示。

```
BEGIN
   SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         RAISE EXCEPTION 'employee % not found', myname;
      WHEN TOO_MANY_ROWS THEN
         RAISE EXCEPTION 'employee % not unique', myname;
END;
```

成功执行一个带STRICT的命令总是会将FOUND置为真。

对于带有RETURNING的INSERT/UPDATE/DELETE，即使没有指定STRICT，PL/uxSQL也会针对多于一个返回行的情况报告一个错误。这是因为没有类似于ORDER BY的选项可以用来决定应该返回哪个被影响的行。

如果为该函数启用了If  print_strict_params，那么当因为STRICT的要求没有被满足而抛出一个错误时，该错误消息的DETAIL将包括传递给该查询的参数信息。可以通过设置pluxsql.print_strict_params为所有函数更改print_strict_params设置，但是只有修改后被编译的函数才会生效。也可以使用一个编译器选项来为一个函数启用它，如下所示。

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
#print_strict_params on
DECLARE
userid int;
BEGIN
   SELECT users.userid INTO STRICT userid
      FROM users WHERE users.username = get_userid.username;
   RETURN userid;
END
$$ LANGUAGE pluxsql;
```

失败时，这个函数会产生如下错误信息。

```
ERROR:  query returned no rows
DETAIL:  parameters: $1 = 'nosuchuser'
CONTEXT:  PL/uxSQL function get_userid(text) line 6 at SQL statement
```

> **注意**
>
> STRICT选项匹配 Oracle PL/SQL 的SELECT INTO和相关语句的行为。

对于要处理来自于一个 SQL 查询的结果行的情况，请见第 6.6.6 节 “通过查询结果循环”。

## 6.5.4. 执行动态命令

很多时候将想要在PL/uxSQL函数中产生动态命令，也就是每次执行中会涉及到不同表或不同数据类型的命令。PL/uxSQL通常对于命令所做的缓存计划尝试（如第 6.11.2 节 “计划缓存”中讨论）在这种情境下无法工作。要处理这一类问题，需要提供EXECUTE语句，如下所示。

EXECUTE *command-string* [ INTO [STRICT] *target* ] [ USING *expression* [, ... ] ];

其中*command-string*是一个能得到一个包含要被执行命令字符串（类型text）的表达式。可选的*target*是一个记录变量、一个行变量或者一个逗号分隔的简单变量以及记录/行域的列表，该命令的结果将存储在其中。可选的USING表达式提供要被插入到该命令中的值。

在计算得到的命令字符串中，不会做PL/uxSQL变量的替换。任何所需的变量值必须在命令字符串被构造时被插入其中，或者可以使用下面描述的参数。

还有，对于通过EXECUTE执行的命令不会有计划被缓存。该命令反而在每次运行时都会被做计划。因此，该命令字符串可以在执行不同表和列上动作的函数中被动态创建。

INTO子句指定一个返回行的 SQL 命令的结果应该被赋值到哪里。如果提供了一个行或变量列表，它必须完全匹配查询结果的结构（当使用一个记录变量时，它会自动把它自己配置为匹配结果结构）。如果返回多个行，只有第一个行会被赋值给INTO变量。如果没有返回行，NULL 会被赋值给INTO变量。如果没有指定INTO变量，该查询结果会被抛弃。

如果给出了STRICT选项，除非该查询刚好产生一行，否则将会报告一个错误。

命令字符串可以使用参数值，它们在命令中用$1、$2等引用。这些符号引用在USING子句中提供的值。这种方法常常更适合于把数据值作为文本插入到命令字符串中：它避免了将该值转换为文本以及转换回来的运行时负荷，并且它更不容易被 SQL 注入攻击，因为不需要引用或转义。如下所示。

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
  INTO c
  USING checked_user, checked_date;
```

需要注意的是，参数符号只能用于数据值 — 如果想要使用动态决定的表名或列名，必须将它们以文本形式插入到命令字符串中。例如，如果前面的那个查询需要在一个动态选择的表上执行，如下所示。

```
EXECUTE 'SELECT count(*) FROM '
   || quote_ident(tabname)
   || ' WHERE inserted_by = $1 AND inserted <= $2'
  INTO c
  USING checked_user, checked_date;
```

一种更干净的方法是为表名或者列名使用format()的 %I规范（被新行分隔的字符串会被串接起来），如下所示。

```
EXECUTE format('SELECT count(*) FROM %I '
```

```
  'WHERE inserted_by = $1 AND inserted <= $2', tabname)
  INTO c
  USING checked_user, checked_date;
```

另一个关于参数符号的限制是，它们只能在SELECT、INSERT、UPDATE和DELETE命令中工作。在另一种语句类型（通常被称为实用语句）中，即使值是数据值，也必须将它们以文本形式插入。

在上面第一个示例中，带有一个简单的常量命令字符串和一些USING参数的EXECUTE命令在功能上等效于直接用PL/uxSQL写的命令，并且允许自动发生PL/uxSQL变量替换。重要的不同之处在于，EXECUTE会在每一次执行时根据当前的参数值重新规划该命令，而PL/uxSQL则是创建一个通用计划并且将其缓存以便重用。在最佳计划强依赖于参数值的情况中，使用EXECUTE来明确地保证不会选择一个通用计划是很有帮助的。

EXECUTE目前不支持SELECT INTO。但是可以执行一个纯的SELECT命令并且指定INTO作为EXECUTE本身的一部分。

> **注意**
>
> PL/uxSQL中的EXECUTE语句与UXsinoDB服务器支持的 SQL 语句无关。服务器的EXECUTE语句不能直接在PL/uxSQL函数中使用（并且也没有必要）。

## 例 6.1. 在动态查询中引用值

在使用动态命令时经常不得不处理单引号的转义。推荐在函数体中使用美元符号引用来引用固定的文本（如果有使用美元符界定的老代码，请参见第 6.12.1 节 “处理引号”中的概述，这样在把上述代码转换成更合理的模式时会省力些）。

动态值需要被小心地处理，因为它们可能包含引号字符。一个使用format()的示例（这假设用美元符号引用了函数体，因此引号不需要被双写），如下所示。

```
EXECUTE format('UPDATE tbl SET %I = $1 '
  'WHERE key = $2', colname) USING newvalue, keyvalue;
```

还可以直接调用引用函数：

```
EXECUTE 'UPDATE tbl SET '
      || quote_ident(colname)
      || ' = '
      || quote_literal(newvalue)
      || ' WHERE key = '
      || quote_literal(keyvalue);
```

这个示例展示了quote_ident和quote_literal函数的使用。为了安全，在进行一个动态查询中的插入之前，包含列或表标识符的表达式应该通过quote_ident被传递。如果表达式包含在被构造出的命令中应该是字符串的值时，它应该通过quote_literal被传递。这些函数采取适当的步骤来分别返回被封闭在双引号或单引号中的文本，其中任何嵌入的特殊字符都会被正确地转义。

因为quote_literal被标记为STRICT，当用一个空参数调用时，它总是会返回空。在上面的示例中，如果newvalue或keyvalue为空，整个动态查询字符串会变成空，导致从EXECUTE得到一个错误。可以通过使用quote_nullable函数来避免这种问题，它工作起来和quote_literal相同，除了用空参数调用时会返回一个字符串NULL。如下所示。

```
EXECUTE 'UPDATE tbl SET '
    || quote_ident(colname)
    || ' = '
    || quote_nullable(newvalue)
    || ' WHERE key = '
    || quote_nullable(keyvalue);
```

如果正在处理的参数值可能为空，那么通常应该用quote_nullable来代替quote_literal。

通常，必须小心地确保查询中的空值不会递送意料之外的结果。例如如果keyvalue为空，下面的WHERE子句永远不会成功，因为在=操作符中使用空操作数得到的结果总是为空，如下所示。

```
'WHERE key = ' || quote_nullable(keyvalue)
```

如果想让空和一个普通键值一样工作，使用用下命令。（目前，IS NOT DISTINCT FROM的处理效率不如=，因此只有在非常必要时才这样做。

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

请注意美元符号引用只对引用固定文本有用。建议不要尝试如下示例。因为如果newvalue的内容碰巧含有$$，那么这段代码就会出问题。同样的缺点可能适用于选择的任何其他美元符号引用定界符。因此，要想安全地引用事先不知道的文本，必须恰当地使用quote_literal、quote_nullable或quote_ident。

```
EXECUTE 'UPDATE tbl SET '
    || quote_ident(colname)
    || ' = $$'
    || newvalue
    || '$$ WHERE key = '
    || quote_literal(keyvalue);
```

动态 SQL 语句也可以使用format函数来安全地构造。如下所示。

```
EXECUTE format('UPDATE tbl SET %I = %L '
  'WHERE key = %L', colname, newvalue, keyvalue);
```

%I等效于quote_ident并且 %L等效于quote_nullable。 format函数可以和 USING子句一起使用，如下所示。

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
  USING newvalue, keyvalue;
```

这种形式更好，因为变量被以它们天然的数据类型格式处理，而不是无条件地把它们转换成文本并且通过%L引用它们。这也效率更高。

动态命令和EXECUTE的一个更大的示例可以在例 6.10 “从PL/SQL移植一个创建另一个函数的函数到PL/uxSQL”中找到，它会构建并且执行一个CREATE FUNCTION命令来定义一个新的函数。

# 6.5.5. 获得结果状态

有好几种方法可以判断一条命令的效果。第一种方法是使用GET DIAGNOSTICS命令，其形式如下所示。

GET [ CURRENT ] DIAGNOSTICS *variable* { = | := } *item* [ , ... ];

这条命令允许检索系统状态指示符。CURRENT是一个噪声词（参见第 6.6.8.1 节 “得到有关一个错误的信息”中的GET STACKED DIAGNOSTICS）。每个*item*是一个关键字， 它标识一个要被赋予给指定##的状态值（变量应具有正确的数据类型来接收状态值）。表 6.1 “可用的诊断项”中展示了当前可用的状态项。冒号等号（:=）可以被用来取代 SQL 标准的=符号。如下所示。

GET DIAGNOSTICS integer_var = ROW_COUNT;

表 6.1. 可用的诊断项

| 名称 | 类型 | 描述 |
|---|---|---|
| ROW_COUNT | bigint | 最近的SQL命令处理的行数 |
| UX_CONTEXT | text | 描述当前调用栈的文本行（见第 6.6.9 节 “获得执行位置信息”） |

第二种判断命令效果的方法是检查一个名为FOUND的boolean类型的特殊变量。在每一次PL/uxSQL函数调用时，FOUND开始都为假。它的值会被下面的每一种类型的语句设置。

- 如果一个SELECT INTO语句赋值了一行，它将把FOUND设置为真，如果没有返回行则将之设置为假。
- 如果一个PERFORM语句生成（并且抛弃）一行或多行，它将把FOUND设置为真，如果没有产生行则将之设置为假。
- 如果UPDATE、INSERT以及DELETE语句影响了至少一行，它们会把FOUND设置为真，如果没有影响行则将之设置为假。
- 如果一个FETCH语句返回了一行，它将把FOUND设置为真，如果没有返回行则将之设置为假。
- 如果一个MOVE语句成功地重定位了游标，它将会把FOUND设置为真，否则设置为假。
- 如果一个FOR或FOREACH语句迭代了一次或多次，它将会把FOUND设置为真，否则设置为假。当循环退出时，FOUND用这种方式设置；在循环执行中，尽管FOUND可能被循环体中的其他语句的执行所改变，但它不会被循环语句修改。
- 如果查询返回至少一行，RETURN QUERY和RETURN QUERY EXECUTE语句会把FOUND设为真， 如果没有返回行则设置为假。

其他的PL/uxSQL语句不会改变FOUND的状态。尤其需要注意的一点是：EXECUTE会修改GET DIAGNOSTICS的输出，但不会修改FOUND的输出。

FOUND是每个PL/uxSQL函数的局部变量；任何对它的修改只影响当前的函数。

### 6.5.6. 什么也不做

有时一个什么也不做的占位语句也很有用。例如，它能够指示 if/then/else 链中故意留出的空分支。可以使用**NULL**语句达到这个目的。

```
NULL;
```

例如，下面的两段代码是等价的。

```
BEGIN
   y := x / 0;
EXCEPTION
   WHEN division_by_zero THEN
      NULL; -- 忽略错误
END;
```

```
BEGIN
   y := x / 0;
EXCEPTION
   WHEN division_by_zero THEN  -- 忽略错误
END;
```

究竟使用哪一种取决于各人的喜好。

> **注意**
>
> 在 Oracle 的 PL/SQL 中，不允许出现空语句列表，并且因此在这种情况下必须使用**NULL**语句。而PL/uxSQL允许什么也不写。

## 6.6. 控制结构

控制结构可能是PL/uxSQL中最有用的（以及最重要）的部分了。利用PL/uxSQL的控制结构，可以非常灵活而且强大的操纵UXsinoDB的数据。

### 6.6.1. 从一个函数返回

有两个命令可以从函数中返回数据：**RETURN**和**RETURN NEXT**。

#### 6.6.1.1. RETURN

```
RETURN expression;
```

带有一个表达式的**RETURN**用于终止函数并把*expression*的值返回给调用者。这种形式被用于不返回集合的PL/uxSQL函数。

如果一个函数返回一个标量类型，表达式的结果将被自动转换成函数的返回类型。但是要返回一个复合（行）值，必须写一个正好产生所需列集合的表达式。这可能需要使用显式造型。

如果声明带输出参数的函数，那么就只需要写不带表达式的RETURN。输出参数变量的当前值将被返回。

如果声明函数返回void，一个RETURN语句可以被用来提前退出函数；但是不要在RETURN后面写一个表达式。

一个函数的返回值不能是未定义。如果控制到达了函数最顶层的块而没有碰到一个RETURN语句，那么会发生一个运行时错误。不过，这个限制不适用于带输出参数的函数以及返回void的函数。在这些情况中，如果顶层的块结束，将自动执行一个RETURN语句。

示例

```
-- 返回一个标量类型的函数
RETURN 1 + 2;
RETURN scalar_var;

-- 返回一个组合类型的函数
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- 必须把列造型成正确的类型
```

### 6.6.1.2. RETURN NEXT以及RETURN QUERY

RETURN NEXT *expression*;
RETURN QUERY *query*;
RETURN QUERY EXECUTE *command-string* [ USING *expression* [, ... ] ];

当一个PL/uxSQL函数被声明为返回SETOF *sometype*，那么遵循的过程则略有不同。在这种情况下，要返回的个体项被用一个RETURN NEXT或者RETURN QUERY命令的序列指定，并且接着会用一个不带参数的最终RETURN命令来指示这个函数已经完成执行。RETURN NEXT可以被用于标量和复合数据类型；对于复合类型，将返回一个完整的结果“表”。RETURN QUERY将执行一个查询的结果追加到一个函数的结果集中。在一个单一的返回集合的函数中，RETURN NEXT和RETURN QUERY可以被随意地混合，这样它们的结果将被串接起来。

RETURN NEXT和RETURN QUERY实际上不会从函数中返回 — 它们简单地向函数的结果集中追加零或多行。然后会继续执行PL/uxSQL函数中的下一条语句。随着后继的RETURN NEXT和RETURN QUERY命令的执行，结果集就建立起来了。最后一个RETURN（应该没有参数）会导致控制退出该函数（或者可以让控制到达函数的结尾）。

RETURN QUERY有一种变体RETURN QUERY EXECUTE，它可以动态指定要被执行的查询。可以通过USING向计算出的查询字符串插入参数表达式，这和在EXECUTE命令中的方式相同。

如果声明函数带有输出参数，只需要写不带表达式的RETURN NEXT。在每一次执行时，输出参数变量的当前值将被保存下来用于最终返回为结果的一行。注意为了创建一个带有输出参数的集合返回函数，在有多个输出参数时，必须声明函数为返回SETOF record；或者如果只有一个类型为*sometype*的输出参数时，声明函数为SETOF *sometype*。

使用RETURN NEXT的函数，如下所示。

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');
```

```
CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
   r foo%rowtype;
BEGIN
   FOR r IN
      SELECT * FROM foo WHERE fooid > 0
   LOOP
      -- 这里可以做一些处理
      RETURN NEXT r; -- 返回 SELECT 的当前行
   END LOOP;
   RETURN;
END
$BODY$
LANGUAGE pluxsql;

SELECT * FROM get_all_foo();
```

使用**RETURN QUERY**的函数，如下所示。

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
   RETURN QUERY SELECT flightid
                  FROM flight
                 WHERE flightdate >= $1
                   AND flightdate < ($1 + 1);

   -- 因为执行还未结束，可以检查是否有行被返回
   -- 如果没有就抛出异常。
   IF NOT FOUND THEN
      RAISE EXCEPTION 'No flight at %.', $1;
   END IF;

   RETURN;
 END
$BODY$
LANGUAGE pluxsql;

-- 返回可用的航班或者在没有可用航班时抛出异常。
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

> **注意**
>
> 如上所述，目前**RETURN NEXT**和**RETURN QUERY**的实现在从函数返回之前会把整个结果集都保存起来。这意味着如果一个PL/uxSQL函数生成一个非常大的结果集，性能可能会很差：数据将被写到磁盘上以避免内存耗尽，但是函数本身在整个结果集都生成之前不会退出。将来的PL/uxSQL版本可能会允许用户定义没有这种限制的集合返回函数。目前，数据开始被写入到磁盘的时机由配置变量work_mem控制。拥有足够内存来存储大型结果集的管理员可以考虑增大这个参数。

## 6.6.2. 从过程中返回

过程没有返回值。因此，过程的结束可以不用RETURN语句。 如果想用一个RETURN语句提前退出代码，只需写一个没有表达式的RETURN。

如果过程有输出参数，那么输出参数最终的值会被返回给调用者。

## 6.6.3. 调用存储过程

PL/uxSQL函数，存储过程或DO块可以使用CALL调存储用过程。输出参数的处理方式与纯SQL中CALL的工作方式不同。存储过程的每个INOUT参数必须和CALL语句中的变量对应，并且无论存储过程返回什么，都会在返回后赋值给该变量。如下所示。

```
CREATE PROCEDURE triple(INOUT x int)
LANGUAGE pluxsql
AS $$
BEGIN
   x := x * 3;
END;
$$;

DO $$
DECLARE myvar int := 5;
BEGIN
  CALL triple(myvar);
  RAISE NOTICE 'myvar = %', myvar;  -- prints 15
END
$$;
```

## 6.6.4. 条件

IF和CASE语句根据某种条件执行二选其一的命令。PL/uxSQL有三种形式的IF，如下所示。

- IF ... THEN ... END IF
- IF ... THEN ... ELSE ... END IF
- IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF

以及两种形式的CASE：

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

### 6.6.4.1. IF-THEN

```
IF boolean-expression THEN
   statements
END IF;
```

IF-THEN语句是IF的最简单形式。 如果条件为真，在THEN和END IF之间的语句将被执行。否则，将忽略它们。

示例

```
IF v_user_id <> 0 THEN
   UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

### 6.6.4.2. IF-THEN-ELSE

```
IF boolean-expression THEN
   statements
ELSE
   statements
END IF;
```

IF-THEN-ELSE语句对IF-THEN进行了增加，它能够指定一组在条件不为真时应该被执行的语句（注意这也包括条件为 NULL 的情况）。

示例

```
IF parentid IS NULL OR parentid = ''
THEN
   RETURN fullname;
ELSE
   RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
   INSERT INTO users_count (count) VALUES (v_count);
   RETURN 't';
ELSE
   RETURN 'f';
END IF;
```

### 6.6.4.3. IF-THEN-ELSIF

```
IF boolean-expression THEN
   statements
[ ELSIF boolean-expression THEN
   statements
[ ELSIF boolean-expression THEN
   statements
   ...
]
]
[ ELSE
   statements ]
```

```
END IF;
```

有时会有多于两种选择。IF-THEN-ELSIF则提供了一个简便的方法来检查多个条件。IF条件会被一个接一个测试，直到找到第一个为真的。然后执行相关语句，然后控制会被交给END IF之后的下一个语句（后续的任何IF条件不会被测试）。如果没有一个IF条件为真，那么ELSE块（如果有）将被执行。

示例

```
IF number = 0 THEN
   result := 'zero';
ELSIF number > 0 THEN
   result := 'positive';
ELSIF number < 0 THEN
   result := 'negative';
ELSE
   -- 嗯，唯一的其他可能性是数字为空
   result := 'NULL';
END IF;
```

关键词ELSIF也可以被拼写成ELSEIF。

另一个可以完成相同任务的方法是嵌套IF-THEN-ELSE语句，如下所示。

```
IF demo_row.sex = 'm' THEN
   pretty_sex := 'man';
ELSE
   IF demo_row.sex = 'f' THEN
      pretty_sex := 'woman';
   END IF;
END IF;
```

不过，这种方法需要为每个IF都写一个匹配的END IF，因此当有很多选择时，这种方法比使用ELSIF要麻烦得多。

#### 6.6.4.4. 简单CASE

```
CASE search-expression
   WHEN expression [, expression [ ... ]] THEN
     statements
  [ WHEN expression [, expression [ ... ]] THEN
     statements
   ... ]
  [ ELSE
     statements ]
END CASE;
```

CASE的简单形式提供了基于操作数等值判断的有条件执行。*search-expression*会被计算（一次）并且一个接一个地与WHEN子句中的每个*expression*比较。如果找到一个匹配，那么相应的*statements*会被执行，并且接着控制会被交给END CASE之后的下一个语句（后续的WHEN表达式不会被计算）。如果没有找到匹配，ELSE ##会被执行。但是如果ELSE不存在，将会抛出一个CASE_NOT_FOUND异常。

示例

```
CASE x
   WHEN 1, 2 THEN
      msg := 'one or two';
   ELSE
      msg := 'other value than one or two';
END CASE;
```

#### 6.6.4.5. 搜索CASE

```
CASE
   WHEN boolean-expression THEN
     statements
 [ WHEN boolean-expression THEN
     statements
   ... ]
 [ ELSE
     statements ]
END CASE;
```

CASE的搜索形式基于布尔表达式真假的有条件执行。每一个WHEN子句的*boolean-expression*会被依次计算，直到找到一个得到真的。然后相应的*statements*会被执行，并且接下来控制会被传递给END CASE之后的下一个语句（后续的WHEN表达式不会被计算）。如果没有找到为真的结果，ELSE *statements*会被执行。但是如果ELSE不存在，那么将会抛出一个CASE_NOT_FOUND异常。

示例

```
CASE
   WHEN x BETWEEN 0 AND 10 THEN
      msg := 'value is between zero and ten';
   WHEN x BETWEEN 11 AND 20 THEN
      msg := 'value is between eleven and twenty';
END CASE;
```

这种形式的CASE整体上等价于IF-THEN-ELSIF，不同之处在于CASE到达一个被忽略的ELSE子句时会导致一个错误而不是什么也不做。

### 6.6.5. 简单循环

使用LOOP、EXIT、CONTINUE、WHILE、FOR、FORALL和FOREACH语句，可以安排PL/uxSQL重复一系列命令。

#### 6.6.5.1. **LOOP**

```
[ <<label>> ]
LOOP
   statements
```

END LOOP [ *label* ];

LOOP定义一个无条件的循环，它会无限重复直到被EXIT或RETURN语句终止。可选的*label*可以被EXIT和CONTINUE语句用在嵌套循环中指定这些语句引用的是哪一层循环。

### 6.6.5.2. EXIT

EXIT [ *label* ] [ WHEN *boolean-expression* ];

如果没有给出*label*，那么最内层的循环会被终止，然后跟在END LOOP后面的语句会被执行。如果给出了*label*，那么它必须是当前或者更高层的嵌套循环或者语句块的标签。然后该命名循环或块就会被终止，并且控制会转移到该循环/块相应的END之后的语句上。

如果指定了WHEN，只有*boolean-expression*为真时才会发生循环退出。否则，控制会转移到EXIT之后的语句。

EXIT可以被用在所有类型的循环中，它并不限于在无条件循环中使用。

在和BEGIN块一起使用时，EXIT会把控制交给块结束后的下一个语句。需要注意的是，一个标签必须被用于这个目的；一个没有被标记的EXIT永远无法被认为与一个BEGIN块匹配（这种状况已经开始改变。这可能允许一个未被标记的EXIT匹配一个BEGIN块）。

示例

```
LOOP
   -- 一些计算
   IF count > 0 THEN
      EXIT;  -- 退出循环
   END IF;
END LOOP;

LOOP
   -- 一些计算
   EXIT WHEN count > 0;  -- 和前一个示例相同的结果
END LOOP;

<<ablock>>
BEGIN
   -- 一些计算
   IF stocks > 100000 THEN
      EXIT ablock;  -- 导致从 BEGIN 块中退出
   END IF;
   -- 当stocks > 100000时，这里的计算将被跳过
END;
```

### 6.6.5.3. CONTINUE

CONTINUE [ *label* ] [ WHEN *boolean-expression* ];

如果没有给出*label*，最内层循环的下一次迭代会开始。也就是，循环体中剩余的所有语句将被跳过，并且控制会返回到循环控制表达式（如果有）来决定是否需要另一次循环迭代。如果*label*存在，它指定应该继续执行的循环的标签。

如果指定了WHEN，该循环的下一次迭代只有在*boolean-expression*为真时才会开始。否则，控制会传递给CONTINUE后面的语句。

CONTINUE可以被用在所有类型的循环中，它并不限于在无条件循环中使用。

示例

```
LOOP
   -- 一些计算
   EXIT WHEN count > 100;
   CONTINUE WHEN count < 50;
   -- 一些用于 count IN [50 .. 100] 的计算
END LOOP;
```

#### 6.6.5.4. WHILE

```
[ <<label>> ]
WHILE boolean-expression LOOP
   statements
END LOOP [ label ];
```

只要*boolean-expression*被计算为真，WHILE语句就会重复一个语句序列。在每次进入到循环体之前都会检查该表达式。

如下所示。

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
   -- 这里是一些计算
END LOOP;

WHILE NOT done LOOP
   -- 这里是一些计算
END LOOP;
```

#### 6.6.5.5. FOR（整型变体）

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
   statements
END LOOP [ label ];
```

这种形式的FOR会创建一个在一个整数范围上迭代的循环。变量*name*会自动定义为类型integer并且只在循环内存在（任何该变量名的现有定义在此循环内都将被忽略）。给出范围上下界的两个表达式在进入循环的时候计算一次。如果没有指定BY子句，迭代步长为 1，否则步长是BY中指定的值，该值也只在循环进入时计算一次。如果指定了REVERSE，那么在每次迭代后步长值会被减除而不是增加。

整数FOR循环，如下所示。

```
FOR i IN 1..10 LOOP
```

```
   -- 我在循环中将取值 1,2,3,4,5,6,7,8,9,10
END LOOP;

FOR i IN REVERSE 10..1 LOOP
   -- 我在循环中将取值 10,9,8,7,6,5,4,3,2,1
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
   -- 我在循环中将取值 10,8,6,4,2
END LOOP;
```

如果下界大于上界（或者在REVERSE情况下是小于），循环体根本不会被执行。而且不会抛出任何错误。

如果一个*label*被附加到FOR循环，那么整数循环变量可以用一个使用那个*label*的限定名引用。

#### 6.6.5.6. FORALL

```
[ <<label>> ]
FORALL name IN lower_bound .. upper_bound
   dml_statement
```

表 6.2. FORALL 参数说明

| 参数 | 说明 |
| --- | --- |
| name | 一个无需声明的标识符，作为下标使用。 |
| lower_bound .. upper_bound | 数字表达式，来指定一组连续有效的索引数字下限和上限。 |
| dml_statement | sql语句，只支持 update、insert、delete、merge、execute语句。 |

FORALL示例，如下所示。

```
FORALL i IN 1..100
 insert into t1 values(i);
```

### 6.6.6. 通过查询结果循环

使用一种不同类型的FOR循环，可以通过一个查询的结果进行迭代并且操纵相应的数据。语法如下所示。

```
[ <<label>> ]
FOR target IN query LOOP
   statements
END LOOP [ label ];
```

*target*是一个记录变量、行变量或者逗号分隔的标量变量列表。*target*被连续不断被赋予来自*query*的每一行，并且循环体将为每一行执行一次。如下所示。

```
CREATE FUNCTION refresh_mviews() RETURNS integer AS $$
DECLARE
  mviews RECORD;
BEGIN
  RAISE NOTICE 'Refreshing all materialized views...';

  FOR mviews IN
    SELECT n.nspname AS mv_schema,
         c.relname AS mv_name,
         ux_catalog.ux_get_userbyid(c.relowner) AS owner
     FROM ux_catalog.ux_class c
  LEFT JOIN ux_catalog.ux_namespace n ON (n.oid = c.relnamespace)
    WHERE c.relkind = 'm'
   ORDER BY 1
  LOOP

    -- Now "mviews" has one record with information about the materialized view

    RAISE NOTICE 'Refreshing materialized view %.% (owner: %)...',
            quote_ident(mviews.mv_schema),
            quote_ident(mviews.mv_name),
            quote_ident(mviews.owner);
    EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I', mviews.mv_schema,
mviews.mv_name);
  END LOOP;

  RAISE NOTICE 'Done refreshing materialized views.';
  RETURN 1;
END;
$$ LANGUAGE pluxsql;
```

如果循环被一个EXIT语句终止，那么在循环之后仍然可以访问最后被赋予的行值。

在这类FOR语句中使用的*query*可以是任何返回行给调用者的 SQL 命令：最常见的是 SELECT，但也可以使用带有RETURNING子句的INSERT、UPDATE或DELETE。一些EXPLAIN之类的功能性命令也可以用在这里。

PL/uxSQL变量会被替换到查询文本中，并且如第 6.11.1 节 “变量替换”和第 6.11.2 节 “计划缓存”中详细讨论的，查询计划会被缓存以用于可能的重用。

FOR-IN-EXECUTE语句是在行上迭代的另一种方式，如下所示。

```
[ <<label>> ]
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
  statements
END LOOP [ label ];
```

这个示例类似前面的形式，只不过源查询被指定为一个字符串表达式，在每次进入FOR循环时都会计算它并且重新规划。这允许程序员在一个预先规划好了的命令的速度和一个动态命令的灵活性之间进行选择，就像一个纯EXECUTE语句那样。在使用EXECUTE时，可以通过USING将参数值插入到动态命令中。

另一种指定要对其结果迭代的查询的方式是将它声明为一个游标。这会在第 6.7.4 节 “通过一个游标的结果循环”中描述。

# 6.6.7. 通过数组循环

FOREACH循环很像一个FOR循环，但不是通过一个 SQL 查询返回的行进行迭代，它通过一个数组值的元素来迭代（通常，FOREACH意味着通过一个组合值表达式的部件迭代；用于通过除数组之外组合类型进行循环的变体可能会在未来被加入）。在一个数组上循环的FOREACH语句如下所示。

```
[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

如果没有SLICE，或者如果没有指定SLICE 0，循环会通过计算*expression*得到的数组的个体元素进行迭代。*target*变量被逐一赋予每一个元素值，并且循环体会为每一个元素执行。通过整数数组的元素循环，如下所示。

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE pluxsql;
```

元素会被按照存储顺序访问，而不管数组的维度数。尽管*target*通常只是一个单一变量，当通过一个组合值（记录）的数组循环时，它可以是一个变量列表。在那种情况下，对每一个数组元素，变量会被从组合值的连续列赋值。

通过一个正SLICE值，FOREACH通过数组的切片而不是单一元素迭代。SLICE值必须是一个不大于数组维度数的整数常量。*target*变量必须是一个数组，并且它接收数组值的连续切片，其中每一个切片都有SLICE指定的维度数。通过一维切片迭代，如下所示。

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE pluxsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
```

```
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

## 6.6.8. 俘获错误

默认情况下，任何在PL/uxSQL函数中发生的错误会中止该函数的执行，而且实际上会中止其周围的事务。可以使用一个带有EXCEPTION子句的BEGIN块俘获错误并且从中恢复。其语法是BEGIN块通常的语法的一个扩展，如下所示。

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
        handler_statements
      ... ]
END;
```

如果没有发生错误，这种形式的块只是简单地执行所有*statements*， 并且接着控制转到END之后的下一个语句。但是如果在*statements*内发生了一个错误，则会放弃对*statements*的进一步处理，然后控制会转到EXCEPTION列表。系统会在列表中寻找匹配所发生错误的第一个*condition*。如果找到一个匹配，则执行对应的*handler_statements*，并且接着把控制转到END之后的下一个语句。如果没有找到匹配，该错误就会传播出去，就好像根本没有EXCEPTION一样：错误可以被一个带有EXCEPTION的闭合块捕捉，如果没有EXCEPTION则中止该函数的处理。

*condition*的名字可以是错误代码中的任何名字。一个分类名匹配其中所有的错误。特殊的条件名OTHERS匹配除了QUERY_CANCELED和ASSERT_FAILURE之外的所有错误类型（虽然通常并不明智，还是可以用名字捕获这两种错误类型）。条件名是大小写无关的。一个错误条件也可以通过SQLSTATE代码指定，例如以下语句是等价的。

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

如果在选中的*handler_statements*内发生了新的错误，那么它不能被这个EXCEPTION子句捕获，而是被传播出去。一个外层的EXCEPTION子句可以捕获它。

当一个错误被EXCEPTION捕获时，PL/uxSQL函数的局部变量会保持错误发生时的值，但是该块中所有对持久数据库状态的改变都会被回滚。例如，思考下如下语句。

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
```

```
    RAISE NOTICE 'caught division_by_zero';
    RETURN x;
END;
```

当控制到达对y赋值的地方时，它会带着一个division_by_zero错误失败。这个错误将被EXCEPTION子句捕获。而在RETURN语句中返回的值将是x增加过后的值。但是UPDATE命令的效果将已经被回滚。不过，在该块之前的INSERT将不会被回滚，因此最终的结果是数据库包含Tom Jones但不包含Joe Jones。

> **提示**
>
> 进入和退出一个包含EXCEPTION子句的块要比不包含EXCEPTION的块开销大的多。因此，只在必要的时候使用EXCEPTION。

### 例 6.2.UPDATE/INSERT的异常

这个示例使用异常处理来酌情执行UPDATE或INSERT。推荐应用使用带有ON CONFLICT DO UPDATE的INSERT 而不是真正使用这种模式。下面示例主要是为了展示PL/uxSQL如何控制流程。

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
  LOOP
    -- 首先尝试更新见
    UPDATE db SET b = data WHERE a = key;
    IF found THEN
      RETURN;
    END IF;
    -- 不在这里，那么尝试插入该键
    -- 如果其他某人并发地插入同一个键，
    -- 可能得到一个唯一键失败
    BEGIN
      INSERT INTO db(a,b) VALUES (key, data);
      RETURN;
    EXCEPTION WHEN unique_violation THEN
      -- 什么也不做，并且循环再次尝试 UPDATE
    END;
  END LOOP;
END;
$$
LANGUAGE pluxsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

这段代码假定unique_violation错误是INSERT造成，并且不是由该表上一个触发器函数中的INSERT导致。如果在该表上有多于一个唯一索引，也可能会发生不正确的行为，因为不管哪个索引导致该错误它都将重试该操作。通过接下来要讨论的特性来检查被捕获的错误是否为所预期的会更安全。

### 6.6.8.1. 得到有关一个错误的信息

异常处理器经常被用来标识发生的特定错误。有两种方法来得到PL/uxSQL中当前异常的信息：特殊变量和GET STACKED DIAGNOSTICS命令。

在一个异常处理器内，特殊变量SQLSTATE包含了对应于被抛出异常的错误代码。特殊变量SQLERRM包含与该异常相关的错误消息。这些变量在异常处理器外是未定义的。

在一个异常处理器内，也可以用GET STACKED DIAGNOSTICS命令检索有关当前异常的信息，该命令的形式如下所示。

GET STACKED DIAGNOSTICS *variable* { = | := } *item* [ , ... ];

每个*item*是一个关键词，它标识一个被赋予给指定##（应该具有接收该值的正确数据类型）的状态值。表 6.3 “错误诊断项”中显示了当前可用的状态项。

表 6.3. 错误诊断项

| 名称 | 类型 | 描述 |
|---|---|---|
| RETURNED_SQLSTATE | text | 该异常的 SQLSTATE 错误代码 |
| COLUMN_NAME | text | 与异常相关的列名 |
| CONSTRAINT_NAME | text | 与异常相关的约束名 |
| UX_DATATYPE_NAME | text | 与异常相关的数据类型名 |
| MESSAGE_TEXT | text | 该异常的主要消息的文本 |
| TABLE_NAME | text | 与异常相关的表名 |
| SCHEMA_NAME | text | 与异常相关的模式名 |
| UX_EXCEPTION_DETAIL | text | 该异常的详细消息文本（如果有） |
| UX_EXCEPTION_HINT | text | 该异常的提示消息文本（如果有） |
| UX_EXCEPTION_CONTEXT | text | 描述产生异常时调用栈的文本行（见第 6.6.9 节 “获得执行位置信息”） |

如果异常没有为一个项设置值，将返回一个空字符串。

示例

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;
BEGIN
  -- 某些可能导致异常的处理
  ...
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = UX_EXCEPTION_DETAIL,
                          text_var3 = UX_EXCEPTION_HINT;
END;
```

### 6.6.9. 获得执行位置信息

GET DIAGNOSTICS（之前在第 6.5.5 节 “获得结果状态”中描述）命令检索有关当前执行状态的信息（反之上文讨论的GET STACKED DIAGNOSTICS命令会把有关执行状态的信息报告成一个以前的错误）。它的UX_CONTEXT状态项可用于标识当前执行位置。状态项UX_CONTEXT将返回一个文本字符串，其中有描述该调用栈的多行文本。第一行会指向当前函数以及当前正在执行GET DIAGNOSTICS的命令。第二行及其后的行表示调用栈中更上层的调用函数。如下所示。

```
CREATE OR REPLACE FUNCTION outer_func() RETURNS integer AS $$
BEGIN
  RETURN inner_func();
END;
$$ LANGUAGE pluxsql;

CREATE OR REPLACE FUNCTION inner_func() RETURNS integer AS $$
DECLARE
  stack text;
BEGIN
  GET DIAGNOSTICS stack = UX_CONTEXT;
  RAISE NOTICE E'--- Call Stack ---\n%', stack;
  RETURN 1;
END;
$$ LANGUAGE pluxsql;

SELECT outer_func();

NOTICE:  --- Call Stack ---
PL/uxSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/uxSQL function outer_func() line 3 at RETURN
CONTEXT:  PL/uxSQL function outer_func() line 3 at RETURN
 outer_func
 ------------
          1
(1 row)
```

GET STACKED DIAGNOSTICS ... UX_EXCEPTION_CONTEXT返回同类的栈跟踪，但是它描述检测到错误的位置而不是当前位置。

## 6.7. 游标

和一次执行整个查询不同，可以建立一个游标来封装该查询，并且接着一次读取该查询结果的一些行。这样做的原因之一是在结果中包含大量行时避免内存不足（不过，PL/uxSQL用户通常不需要担心这些，因为FOR循环在内部会自动使用一个游标来避免内存问题）。一种更有趣的用法是返回一个函数已经创建的游标的引用，允许调用者读取行。这提供了一种有效的方法从函数中返回大型行集。

### 6.7.1. 声明游标变量

所有在PL/uxSQL中对游标的访问都会通过游标变量，它总是特殊的数据类型refcursor。创建游标变量的一种方法是把它声明为一个类型为refcursor的变量。另外一种方法是使用游标声明语法，如下所示。

*name* [ [ NO ] SCROLL ] CURSOR [ ( *arguments* ) ] FOR *query*;

（为了对Oracle的兼容性，可以用IS替代FOR）。如果指定了SCROLL，那么游标可以反向滚动；如果指定了NO SCROLL，那么反向取的动作会被拒绝；如果二者都没有被指定，那么能否进行反向取就取决于查询。如果指定了*arguments*， 那么它是一个逗号分隔的*name datatype*对的列表， 它们定义在给定查询中要被参数值替换的名称。实际用于替换这些名字的值将在游标被打开之后指定。

示例

```
DECLARE
  curs1 refcursor;
  curs2 CURSOR FOR SELECT * FROM tenk1;
  curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

所有这三个变量都是refcursor类型，但是第一个可以用于任何查询，而第二个已经被绑定了一个完全指定的查询，并且最后一个被绑定了一个参数化查询。（游标被打开时，key将被一个整数参数值替换）。变量curs1被称为未绑定，因为它没有被绑定到任何特定查询。

## 6.7.2. 打开游标

在一个游标可以被用来检索行之前，它必需先被打开（这是和 SQL 命令DECLARE CURSOR等效的操作）。PL/uxSQL有三种形式的OPEN命令，其中两种用于未绑定游标变量，另外一种用于已绑定的游标变量。

> **注意**
>
> 可以通过第 6.7.4 节 “通过一个游标的结果循环”中描述的FOR语句在不显式打开游标的情况下使用已绑定的游标变量。

### 6.7.2.1. OPEN FOR *query*

OPEN *unbound_cursorvar* [ [ NO ] SCROLL ] FOR *query*;

该游标变量被打开并且被给定要执行的查询。游标不能是已经打开的，并且它必需已经被声明为一个未绑定的游标变量（也就是声明为一个简单的refcursor变量）。该查询必须是一条SELECT或者其它返回行的东西（例如EXPLAIN）。该查询会按照其它PL/uxSQL中的 SQL命令同等的方式对待：先代换PL/uxSQL变量名，并且执行计划会被缓存用于可能的重用。当一个PL/uxSQL变量被替换到游标查询中时，替换的值是在OPEN时它所具有的值。对该变量后续的改变不会影响游标的行为。对于一个已经绑定的游标，SCROLL和NO SCROLL选项具有相同的含义。

示例

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

### 6.7.2.2. OPEN FOR EXECUTE

OPEN *unbound_cursorvar* [ [ NO ] SCROLL ] FOR EXECUTE *query_string*
[ USING *expression* [, ... ] ];

打开游标变量并且执行指定的查询。该游标不能是已打开的，并且必须已经被声明为一个未绑定的游标变量（也就是声明为一个简单的refcursor变量）。该查询以和EXECUTE中相同的方式被指定为一个字符串表达式。照例，这提供了灵活性，因此查询计划可以在两次运行之间变化（见第 6.11.2 节 “计划缓存”），并且它也意味着在该命令字符串上还没有完成变量替换。正如EXECUTE，可以通过format()和USING将参数值插入到动态命令中。SCROLL和NO SCROLL选项具有和已绑定游标相同的含义。

在如下示例中，表名被通过format()插入到查询中。col1的比较值被通过一个USING参数插入，所以它不需要引用。

```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tabname) USING
 keyvalue;
```

### 6.7.2.3. 打开一个已绑定的游标

OPEN *bound_cursorvar* [ ( [ *argument_name* := ] *argument_value* [, ...] ) ];

这种形式的OPEN被用于打开一个游标变量，它的查询是在声明时绑定的。该游标不能是已经打开的。当且仅当该游标被声明为接收参数时，才必需出现一个实际参数值表达式的列表。这些值将被替换到命令中。

一个已绑定游标的查询计划总是被认为是可缓存的，在这种情况中没有EXECUTE的等效形式。注意SCROLL和NO SCROLL不能在OPEN中指定，因为游标的滚动行为已经被确定。

使用位置或命名记号可以传递参数值。在位置记号中，所有参数都必须按照顺序指定。在命名记号中，每一个参数的名字被使用:=指定以将它和参数表达式分隔开。

示例（这些示例使用上面示例中的游标声明）。

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

因为在一个已绑定游标的查询上已经完成了变量替换，实际有两种方式将值传到游标中：给OPEN一个显式参数，或者在查询中隐式引用一个PL/uxSQL变量。不过，只有在已绑定游标之前声明的变量才将会被替换到游标中。在两种情况下，要被传递的值都是在OPEN时确定的。例如，得到上例中curs3相同效果的另一种方式，如下所示。

```
DECLARE
   key integer;
   curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
   key := 42;
   OPEN curs4;
```

## 6.7.3. 使用游标

一旦一个游标已经被打开，那么就可以用这里描述的语句操作它。

这些操作不需要发生在打开该游标开始操作的同一个函数中。可以从一个函数返回一个refcursor值，并且让调用者在该游标上操作（在内部，refcursor值只是一个包含该游标活动查询的所谓入口的字符串名称。这个名字可以被传递、赋予给其它refcursor变量等等，而不用担心扰乱入口）。

所有入口会在事务的结尾被隐式地关闭。因此一个refcursor值只能在该事务结束前用于引用一个打开的游标。

#### 6.7.3.1. FETCH

FETCH [ *direction* { FROM | IN } ] *cursor* INTO *target*;

就像SELECT INTO一样，FETCH从游标中检索下一行到目标中，目标可以是一个行变量、记录变量或者逗号分隔的简单变量列表。如果没有下一行，目标会被设置为 NULL。与SELECT INTO一样，可以检查特殊变量FOUND来看一行是否被获得。

*direction*子句可以是 SQL FETCH命令中允许的任何变体，除了那些能够取得多于一行的。即它可以是 NEXT、 PRIOR、 FIRST、 LAST、 ABSOLUTE *count*、 RELATIVE *count*、 FORWARD或者 BACKWARD。 省略*direction*和指定NEXT是一样的。在使用*count*的形式中，*count*可以是任意的整数值表达式（与SQL命令FETCH不一样，FETCH仅允许整数常量）。除非游标被使用SCROLL选项声明或打开，否则要求反向移动的*direction*值很可能会失败。

*cursor*必须是一个引用已打开游标入口的refcursor变量名。

示例

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

#### 6.7.3.2. MOVE

MOVE [ *direction* { FROM | IN } ] *cursor*;

MOVE重新定位一个游标而不检索任何数据。MOVE的工作方式与FETCH命令很相似，但是MOVE只是重新定位游标并且不返回至移动到的行。与SELECT INTO一样，可以检查特殊变量FOUND来看要移动到的行是否存在。

示例

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

#### 6.7.3.3. UPDATE/DELETE WHERE CURRENT OF

UPDATE *table* SET ... WHERE CURRENT OF *cursor*;

DELETE FROM *table* WHERE CURRENT OF *cursor*;

当一个游标被定位到一个表行上时，使用该游标标识该行就可以对它进行更新或删除。对于游标的查询可以是什么是有限制的（尤其是不能有分组），并且最好在游标中使用FOR UPDATE。详见《优炫数据库参考手册 V2.1》“SQL命令”章节中的**DECLARE**命令。

一个示例

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

#### 6.7.3.4. CLOSE

CLOSE *cursor*;

**CLOSE**关闭一个已打开游标的底层入口。这样就可以在事务结束之前释放资源，或者释放掉该游标变量以便再次打开。

一个示例

```
CLOSE curs1;
```

#### 6.7.3.5. 返回游标

PL/uxSQL函数可以向调用者返回游标。这对于返回多行或多列（特别是巨大的结果集）很有用。要想这么做，该函数打开游标并且把该游标的名字返回给调用者（或者简单的使用调用者指定的或已知的入口名打开游标）。调用者接着可以从游标中取得行。游标可以由调用者关闭，或者是在事务关闭时自行关闭。

用于一个游标的入口名可以由编程者指定或者自动生成。要指定一个入口名，只需要在打开refcursor变量之前简单地为它赋予一个字符串。**OPEN**将把refcursor变量的字符串值用作底层入口的名字。不过，如果refcursor变量为空，**OPEN**会自动生成一个与任何现有入口不冲突的名称，并且将它赋予给refcursor变量。

> **注意**
>
> 一个已绑定的游标变量被初始化为表示其名称的字符串值，因此入口的名字和游标变量名相同，除非程序员在打开游标之前通过赋值覆盖了这个名字。但是一个未绑定的游标变量最初默认为空值，因此它会收到一个自动生成的唯一名字，除非被覆盖。

下面的示例显示了一个调用者提供游标名字的方法。

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
   OPEN $1 FOR SELECT col FROM test;
   RETURN $1;
END;
' LANGUAGE pluxsql;
```

```
BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

下面的示例使用了自动游标名生成。

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
   ref refcursor;
BEGIN
   OPEN ref FOR SELECT col FROM test;
   RETURN ref;
END;
' LANGUAGE pluxsql;

-- 需要在一个事务中使用游标。
BEGIN;
SELECT reffunc2();

     reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

下面的示例展示了从一个函数中返回多个游标的一种方法。

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
   OPEN $1 FOR SELECT * FROM table_1;
   RETURN NEXT $1;
   OPEN $2 FOR SELECT * FROM table_2;
   RETURN NEXT $2;
END;
$$ LANGUAGE pluxsql;

-- 需要在一个事务中使用游标。
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

## 6.7.4. 通过一个游标的结果循环

有一种FOR语句的变体，它允许通过游标返回的行进行迭代。语法结构如下所示。

[ <<*label*>> ]
FOR *recordvar* IN *bound_cursorvar* [ ( [ *argument_name* := ] *argument_value* [, ...] ) ] LOOP
  *statements*
END LOOP [ *label* ];

该游标变量必须在声明时已经被绑定到某个查询，并且它不能已经被打开。FOR语句会自动打开游标，并且在退出循环时自动关闭游标。当且仅当游标被声明要使用参数时，才必须出现一个实际参数值表达式的列表。这些值会被替换到查询中，采用OPEN期间的方式（见第 6.7.2.3 节“打开一个已绑定的游标”）。

变量*recordvar*会被自动定义为record类型，并且只存在于循环内部（循环中该变量名任何已有定义都会被忽略）。每一个由游标返回的行都会被陆续地赋值给这个记录变量并且执行循环体。

# 6.8. 事务管理

在由CALL命令调用的过程中以及匿名代码块（DO命令）中，可以用命令COMMIT和ROLLBACK结束事务。在一个事务被使用这些命令结束后，一个新的事务会被自动开始，因此没有单独的START TRANSACTION命令（注意BEGIN和END在PL/uxSQL中有不同的含义）。

这里是一个简单的示例。

```
CREATE PROCEDURE transaction_test1()
LANGUAGE pluxsql
AS $$
BEGIN
  FOR i IN 0..9 LOOP
    INSERT INTO test1 (a) VALUES (i);
    IF i % 2 = 0 THEN
      COMMIT;
    ELSE
      ROLLBACK;
    END IF;
  END LOOP;
END
$$;

CALL transaction_test1();
```

新事务开始时具有默认事务特征，如事务隔离级别。在循环中提交事务的情况下，可能需要以与前一个事务相同的特征来自动启动新事务。 命令COMMIT AND CHAIN和ROLLBACK AND CHAIN可以完成此操作。

只有在从顶层调用的CALL或DO中才能进行事务控制，在没有任何其他中间命令的嵌套CALL或DO调用中也能进行事务控制。例如，如果调用栈是CALL proc1() [rarr ] CALL proc2() [rarr ] CALL proc3()，那么第二个和第三个过程可以执行事务控制动作。但是如果调用栈是CALL proc1() [rarr ] SELECT func2() [rarr ] CALL proc3()，则最后一个过程不能做事务控制，因为中间有SELECT。

对于游标循环有特殊的考虑。如下所示。

```
CREATE PROCEDURE transaction_test2()
```

```
LANGUAGE pluxsql
AS $$
DECLARE
  r RECORD;
BEGIN
  FOR r IN SELECT * FROM test2 ORDER BY x LOOP
    INSERT INTO test1 (a) VALUES (r.x);
    COMMIT;
  END LOOP;
END;
$$;

CALL transaction_test2();
```

通常，游标会在事务提交时被自动关闭。但是，一个作为循环的组成部分创建的游标会自动被第一个COMMIT或ROLLBACK转变成一个可保持游标。这意味着该游标在第一个COMMIT或ROLLBACK处会被完全计算出来，而不是逐行被计算。该游标在循环后仍会被自动删除，因此这通常对用户是不可见的。

有非只读命令（UPDATE ... RETURNING）驱动的游标循环中不允许有事务命令。

事务在一个具有异常处理部分的块中不能被结束。

# 6.9. 错误和消息

## 6.9.1. 报告错误和消息

使用RAISE语句报告消息以及抛出错误。

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

*level*选项指定了错误的严重性。允许的级别有DEBUG、LOG、INFO、NOTICE, WARNING以及EXCEPTION，默认级别是EXCEPTION。EXCEPTION会抛出一个错误（通常会中止当前事务）。其他级别仅仅是产生不同优先级的消息。不管一个特定优先级的消息是被报告给客户端、还是写到服务器日志、亦或是二者同时都做，这都由log_min_messages和client_min_messages配置变量控制。

如果有*level*，在它后面可以写一个*format*（它必须是一个简单字符串而不是表达式）。该格式字符串指定要被报告的错误消息文本。在格式字符串后面可以跟上可选的要被插入到该消息的参数表达式。在格式字符串中，%会被下一个可选参数的值所替换。写%%可以发出一个字面的%。参数的数量必须匹配格式字符串中%占位符的数量，否则在函数编译期间就会发生错误。

在这个示例中，v_job_id的值将替换字符串中的%。

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

通过写一个后面跟着*option = expression*项的USING，可以为错误报告附加一些额外信息。每一个*expression*可以是任意字符串值的表达式。允许如下*option*关键词。

- MESSAGE

  设置错误消息文本。这个选项可以被用于在USING之前包括一个格式字符串的RAISE形式。

- DETAIL

  提供一个错误的细节消息。

- HINT

  提供一个提示消息。

- ERRCODE

  指定要报告的错误代码（SQLSTATE），或者直接作为一个五字符的 SQLSTATE 代码。

- COLUMN CONSTRAINT DATATYPE TABLE SCHEMA

  提供一个相关对象的名称。

用给定的错误消息和提示中止事务，如下所示。

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
    USING HINT = 'Please check your user ID';
```

设置 SQLSTATE 有两种等价的方法，如下所示。

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

还有第二种RAISE语法，在其中主要参数是要被报告的条件名或 SQLSTATE，如下所示。

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

在这种语法中，USING能被用来提供一个自定义的错误消息、细节或提示。另一种做前面的示例的方法如下所示。

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

仍有另一种变体是写RAISE USING或者RAISE *level* USING并且把所有其他东西都放在USING列表中。

RAISE的最后一种变体根本没有参数。这种形式只能被用在一个BEGIN块的EXCEPTION子句中，它导致当前正在被处理的错误被重新抛出。

> **注意**
>
> 没有参数的RAISE被解释为重新抛出来自包含活动异常处理器的块的错误。因此一个嵌套在那个处理器中的EXCEPTION子句无法捕捉它，即使RAISE位于嵌套EXCEPTION子句的块中也是这样。这种行为很奇怪，也并不兼容 Oracle 的 PL/SQL。

如果在一个RAISE EXCEPTION命令中没有指定条件名以及 SQLSTATE，默认是使用ERRCODE_RAISE_EXCEPTION (P0001)。如果没有指定消息文本，默认是使用条件名或 SQLSTATE 作为消息文本。

> **注意**
>
> 当用 SQLSTATE 代码指定一个错误代码时，不会受到预定义错误代码的限制，而是可以选择任何由五位以及大写 ASCII 字母构成的错误代码，只有00000不能使用。推荐尽量避免抛出以三个零结尾的错误代码，因为这些是分类代码并且只能用来捕获整个类别。

## 6.9.2. 检查断言

ASSERT语句是一种向 PL/uxSQL函数中插入调试检查的方便方法。

ASSERT *condition* [ , *message* ];

*condition*是一个布尔表达式，它被期望总是计算为真。如果确实如此，ASSERT语句不会再做什么。但如果结果是假或者空，那么将发生一个ASSERT_FAILURE异常（如果在计算*condition*时发生错误，它会被报告为一个普通错误）。

如果提供了可选的*message*，它是一个结果（如果非空）被用来替换默认错误消息文本“assertion failed”的表达式（如果*condition*失败）。*message*表达式在断言成功的普通情况下不会被计算。

通过配置参数pluxsql.check_asserts可以启用或者禁用断言测试，这个参数接受布尔值且默认为on。如果这个参数为off，则ASSERT语句什么也不做。

注意ASSERT是为了检测程序的 bug，而不是报告普通的错误情况。如果要报告普通错误，请使用前面介绍的RAISE语句。

# 6.10. 触发器函数

PL/uxSQL可以被用来在数据更改或者数据库事件上定义触发器函数。触发器函数用CREATE FUNCTION命令创建，它被声明为一个没有参数并且返回类型为trigger（对于数据更改触发器）或者event_trigger（对于数据库事件触发器）的函数。名为UX_*something*的特殊局部变量将被自动创建用以描述触发该调用的条件。

## 6.10.1. 数据改变的触发器

一个数据更改触发器被声明为一个没有参数并且返回类型为trigger的函数。注意，如下所述，即便该函数准备接收一些在CREATE TRIGGER中指定的参数 — 这类参数通过TG_ARGV传递，也必须把它声明为没有参数。

当一个PL/uxSQL函数当做触发器调用时，在顶层块会自动创建一些特殊变量。如下所示。

- NEW

  数据类型是RECORD；该变量为行级触发器中的INSERT/UPDATE操作保持新数据行。在语句级别的触发器以及DELETE操作，这个变量是null。

- OLD

  数据类型是RECORD；该变量为行级触发器中的UPDATE/DELETE操作保持新数据行。在语句级别的触发器以及INSERT操作，这个变量是null。

- TG_NAME

  数据类型是name；该变量包含实际触发的触发器名。

- TG_WHEN

  数据类型是text；是值为BEFORE、AFTER或INSTEAD OF的一个字符串，取决于触发器的定义。

- TG_LEVEL

  数据类型是text；是值为ROW或STATEMENT的一个字符串，取决于触发器的定义。

- TG_OP

  数据类型是text；是值为INSERT、UPDATE、DELETE或TRUNCATE的一个字符串，它说明触发器是为哪个操作引发。

- TG_RELID

  数据类型是oid；是导致触发器调用的表的对象 ID。

- TG_RELNAME

  数据类型是name；是导致触发器调用的表的名称。现在已经被废弃，并且可能在未来的一个发行中消失。使用TG_TABLE_NAME替代。

- TG_TABLE_NAME

  数据类型是name；是导致触发器调用的表的名称。

- TG_TABLE_SCHEMA

  数据类型是name；是导致触发器调用的表所在的模式名。

- TG_NARGS

  数据类型是integer；在CREATE TRIGGER语句中给触发器函数的参数数量。

- TG_ARGV[]

  数据类型是text数组；来自CREATE TRIGGER语句的参数。索引从 0 开始记数。非法索引（小于 0 或者大于等于tg_nargs）会导致返回一个空值。

一个触发器函数必须返回NULL或者是一个与触发器为之引发的表结构完全相同的记录/行值。

BEFORE引发的行级触发器可以返回一个空来告诉触发器管理器跳过对该行剩下的操作（即后续的触发器将不再被引发，并且不会对该行发生INSERT/UPDATE/DELETE)。如果返回了一个非空值，那么对该行值会继续操作。返回不同于原始NEW的行值将修改将要被插入或更新的行。因此，如果该触发器函数想要触发动作正常成功而不修改行值，NEW（或者另一个相等的值）必须被返回。要修改将被存储的行，可以直接在NEW中替换单一值并且返回修改后的NEW，或者

构建一个全新的记录/行来返回。在一个DELETE上的前触发器情况下，返回值没有直接效果，但是它必须为非空以允许触发器动作继续下去。注意NEW在DELETE触发器中是空值，因此返回它通常没有意义。在DELETE中的常用方法是返回OLD.

INSTEAD OF触发器（总是行级触发器，并且可能只被用于视图）能够返回空来表示它们没有执行任何更新，并且对该行剩余的操作可以被跳过（即后续的触发器不会被引发，并且该行不会被计入外围INSERT/UPDATE/DELETE的行影响状态中）。否则一个非空值应该被返回用以表示该触发器执行了所请求的操作。对于INSERT 和UPDATE操作，返回值应该是NEW，触发器函数可能对它进行了修改来支持INSERT RETURNING和UPDATE RETURNING（这也将影响被传递给任何后续触发器的行值，或者被传递给带有ON CONFLICT DO UPDATE的INSERT语句中一个特殊的EXCLUDED别名引用）。对于DELETE操作，返回值应该是OLD。

一个AFTER行级触发器或一个BEFORE或AFTER语句级触发器的返回值总是会被忽略，它可能也是空。不过，任何这些类型的触发器可能仍会通过抛出一个错误来中止整个操作。

例 6.3 “一个 PL/uxSQL 触发器函数”展示了PL/uxSQL中一个触发器函数的示例。

### 例 6.3. 一个 PL/uxSQL 触发器函数

这个示例触发器保证：任何时候一个行在表中被插入或更新时，当前用户名和时间也会被标记在该行中。并且它会检查给出了一个雇员的姓名以及薪水是一个正值。

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- 检查给出了 empname 以及 salary
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- 谁会倒贴钱为了工作？
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- 记住谁在什么时候改变了工资单
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE pluxsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```

另一种记录对表的改变的方法涉及到创建一个新表来为每一个发生的插入、更新或删除保持一行。这种方法可以被认为是对一个表的改变的审计。例 6.4 “一个用于审计的 PL/uxSQL 触发器函数”展示了PL/uxSQL中一个审计触发器函数的示例。

### 例 6.4. 一个用于审计的 PL/uxSQL 触发器函数

这个示例触发器保证了在emp表上的任何插入、更新或删除一行的动作都被记录（即审计）在emp_audit表中。当前时间和用户名会被记录到行中，还有在其上执行的操作类型。

```
CREATE TABLE emp (
    empname          text NOT NULL,
    salary           integer
);

CREATE TABLE emp_audit(
    operation        char(1)   NOT NULL,
    stamp            timestamp NOT NULL,
    userid           text      NOT NULL,
    empname          text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- 在 emp_audit 中创建一行来反映 emp 上执行的动作，
        -- 使用特殊变量 TG_OP 来得到操作。
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
        END IF;
        RETURN NULL; -- 因为这是一个 AFTER 触发器，结果被忽略
    END;
$emp_audit$ LANGUAGE pluxsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE FUNCTION process_emp_audit();
```

前一个示例的一种变体使用一个视图将主表连接到审计表来展示每一项最后被修改是什么时间。这种方法还是记录了对于表修改的完整审查跟踪，但是也提供了审查跟踪的一个简化视图，只为每一个项显示从审查跟踪生成的最后修改时间戳。例 6.5 “一个用于审计的 PL/uxSQL 视图触发器函数”展示了在PL/uxSQL中一个视图上审计触发器的示例。

### 例 6.5. 一个用于审计的 PL/uxSQL 视图触发器函数

这个示例在视图上使用了一个触发器让它变得可更新，并且确保视图中一行的任何插入、更新或删除被记录（即审计）在emp_audit表中。当前时间和用户名会被与执行的操作类型一起记录，并且该视图会显示每一行的最后修改时间。

```
CREATE TABLE emp (
   empname          text PRIMARY KEY,
   salary           integer
);

CREATE TABLE emp_audit(
   operation         char(1)   NOT NULL,
   userid            text      NOT NULL,
   empname           text      NOT NULL,
   salary            integer,
   stamp             timestamp NOT NULL
);

CREATE VIEW emp_view AS
   SELECT e.empname,
         e.salary,
         max(ea.stamp) AS last_updated
     FROM emp e
     LEFT JOIN emp_audit ea ON ea.empname = e.empname
    GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
   BEGIN
      --
      -- 执行 emp 上所要求的操作，并且在 emp_audit 中创建一行来反映对 emp 的改变。
      --
      IF (TG_OP = 'DELETE') THEN
         DELETE FROM emp WHERE empname = OLD.empname;
         IF NOT FOUND THEN RETURN NULL; END IF;

         OLD.last_updated = now();
         INSERT INTO emp_audit VALUES('D', user, OLD.*);
         RETURN OLD;
      ELSIF (TG_OP = 'UPDATE') THEN
         UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
         IF NOT FOUND THEN RETURN NULL; END IF;

         NEW.last_updated = now();
         INSERT INTO emp_audit VALUES('U', user, NEW.*);
         RETURN NEW;
      ELSIF (TG_OP = 'INSERT') THEN
         INSERT INTO emp VALUES(NEW.empname, NEW.salary);

         NEW.last_updated = now();
         INSERT INTO emp_audit VALUES('I', user, NEW.*);
         RETURN NEW;
      END IF;
   END;
$$ LANGUAGE pluxsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
   FOR EACH ROW EXECUTE FUNCTION update_emp_view();
```

触发器的一种用法是维护一个表的另一个汇总表。作为结果的汇总表可以用来在特定查询中替代原始表 — 通常会大量减少运行时间。这种技术常用于数据仓库中，在其中被度量或被观察数据的表（称为事实表）可能会极度大。例 6.6 “一个 PL/uxSQL 用于维护汇总表的触发器函数”展示了PL/uxSQL中一个为数据仓库事实表维护汇总表的触发器函数的示例。

## 例 6.6. 一个 PL/uxSQL 用于维护汇总表的触发器函数

这里详述的模式有一部分是基于 Ralph Kimball 所作的**The Data Warehouse Toolkit**中的**Grocery Store**示例。

```
--
-- 主表 - 时间维度和销售事实。
--
CREATE TABLE time_dimension (
    time_key                integer NOT NULL,
    day_of_week             integer NOT NULL,
    day_of_month            integer NOT NULL,
    month                   integer NOT NULL,
    quarter                 integer NOT NULL,
    year                    integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                integer NOT NULL,
    product_key             integer NOT NULL,
    store_key               integer NOT NULL,
    amount_sold             numeric(12,2) NOT NULL,
    units_sold              integer NOT NULL,
    amount_cost             numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- 汇总表 - 按时间汇总销售
--
CREATE TABLE sales_summary_bytime (
    time_key                integer NOT NULL,
    amount_sold             numeric(15,2) NOT NULL,
    units_sold              numeric(12) NOT NULL,
    amount_cost             numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- 在 UPDATE、INSERT、DELETE 时修改汇总列的函数和触发器。
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
```

```
  delta_amount_cost       numeric(15,2);
BEGIN

  -- 算出增量/减量数。
  IF (TG_OP = 'DELETE') THEN

    delta_time_key = OLD.time_key;
    delta_amount_sold = -1 * OLD.amount_sold;
    delta_units_sold = -1 * OLD.units_sold;
    delta_amount_cost = -1 * OLD.amount_cost;

  ELSIF (TG_OP = 'UPDATE') THEN

    -- 禁止更改 the time_key 的更新-
    -- （可能不会太麻烦，因为大部分的更改是用 DELETE + INSERT 完成的）。
    IF ( OLD.time_key != NEW.time_key) THEN
      RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                OLD.time_key, NEW.time_key;
    END IF;

    delta_time_key = OLD.time_key;
    delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
    delta_units_sold = NEW.units_sold - OLD.units_sold;
    delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

  ELSIF (TG_OP = 'INSERT') THEN

    delta_time_key = NEW.time_key;
    delta_amount_sold = NEW.amount_sold;
    delta_units_sold = NEW.units_sold;
    delta_amount_cost = NEW.amount_cost;

  END IF;


  -- 插入或更新带有新值的汇总行。
  <<insert_update>>
  LOOP
    UPDATE sales_summary_bytime
      SET amount_sold = amount_sold + delta_amount_sold,
        units_sold = units_sold + delta_units_sold,
        amount_cost = amount_cost + delta_amount_cost
      WHERE time_key = delta_time_key;

    EXIT insert_update WHEN found;

    BEGIN
      INSERT INTO sales_summary_bytime (
            time_key,
            amount_sold,
            units_sold,
            amount_cost)
        VALUES (
            delta_time_key,
```

```
                delta_amount_sold,
                delta_units_sold,
                delta_amount_cost
               );

        EXIT insert_update;

      EXCEPTION
        WHEN UNIQUE_VIOLATION THEN
          -- 什么也不做
      END;
    END LOOP insert_update;

    RETURN NULL;

  END;
$maint_sales_summary_bytime$ LANGUAGE pluxsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
  FOR EACH ROW EXECUTE FUNCTION maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

AFTER也可以利用*传递表*来观察被触发语句更改的整个行集合。CREATE TRIGGER命令会为一个或者两个传递表分配名字，然后函数可以引用那些名字，就好像它们是只读的临时表一样。例 6.7 “用传递表进行审计”展示了一个示例。

## 例 6.7. 用传递表进行审计

这个示例产生和例 6.4 “一个用于审计的 PL/uxSQL 触发器函数”相同的结果，但并未使用一个为每一行都触发的触发器，而是在把相关信息收集到一个传递表中之后用了一个只为每个语句引发一次的触发器。当调用语句修改了很多行时，这种方法明显比行触发器方法快。注意必须为每一种事件建立一个单独的触发器声明，因为每种情况的REFERENCING子句必须不同。但是这并不能阻止使用单一的触发器函数（实际上，使用三个单独的函数会更好，因为可以避免在TG_OP上的运行时测试）。

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
```

```
    userid          text     NOT NULL,
    empname         text     NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- 在emp_audit中创建行来反映在emp上执行的操作，
        -- 利用特殊变量TG_OP来区分操作。
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit
                SELECT 'D', now(), user, o.* FROM old_table o;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit
                SELECT 'U', now(), user, n.* FROM new_table n;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit
                SELECT 'I', now(), user, n.* FROM new_table n;
        END IF;
        RETURN NULL; -- 由于这是一个AFTER触发器，所以结果被忽略
    END;
$emp_audit$ LANGUAGE pluxsql;

CREATE TRIGGER emp_audit_ins
    AFTER INSERT ON emp
    REFERENCING NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_upd
    AFTER UPDATE ON emp
    REFERENCING OLD TABLE AS old_table NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_del
    AFTER DELETE ON emp
    REFERENCING OLD TABLE AS old_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
```

## 6.10.2. 事件触发器

PL/uxSQL可以被用来定义事件触发器。UXsinoDB要求一个可以作为事件触发器调用的函数必须被声明为没有参数并且返回类型为event_trigger。

当一个PL/uxSQL函数被作为一个事件触发器调用，在顶层块中会自动创建一些特殊变量。如下所示。

TG_EVENT

数据类型是text；它是一个表示引发触发器的事件的字符串。

TG_TAG

数据类型是text；它是一个变量，包含了该触发器为之引发的命令标签。

例 6.8 “一个 PL/uxSQL 事件触发器函数”展示了PL/uxSQL中一个事件触发器函数的示例。

例 6.8. 一个 PL/uxSQL 事件触发器函数

这个示例触发器在受支持命令每一次被执行时会简单地抛出一个NOTICE消息。

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
   RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE pluxsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE FUNCTION snitch();
```

# 6.11. PL/uxSQL的内部

这一节讨论了一些PL/uxSQL用户应该知道的一些重要的实现细节。

## 6.11.1. 变量替换

一个PL/uxSQL函数中的 SQL 语句和表达式能够引用该函数的变量和参数。在现象背后，PL/uxSQL会为这些引用替换查询参数。只有在语法上允许一个参数或列引用的地方才会替换参数。作为一种极端情况，一般不建议使用如下示例。

```
INSERT INTO foo (foo) VALUES (foo);
```

foo的第一次出现在语法上必须是一个表名， 因此它将不会被替换，即使该函数有一个名为foo的变量。第二次出现必须是该表的一列的名称，因此它也将不会被替换。只有第三次出现是对该函数变量引用的候选。

因为变量名在语法上与表列的名字没什么区别，在也引用表的语句中会有歧义：一个给定的名字意味着一个表列或一个变量？把前一个示例进行修改，如下所示。

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

这里，dest和src必须是表名，并且col必须是dest的一列，但是foo和bar可能该函数的变量或者src的列。

默认情况下，如果一个 SQL 语句中的名称可能引用一个变量或者一个表列，PL/uxSQL将报告一个错误。修复这种问题的方法很多：可以重命名变量或列来，或者对有歧义的引用加以限定，或者告诉PL/uxSQL要引用哪种解释。

最简单的解决方案是重命名变量或列。一种常用的编码规则是为PL/uxSQL变量使用一种不同于列名的命名习惯。例如，如果将函数变量统一地命名为v_*something*，而列名不会开始于v_，就不会发生冲突。

另外可以限定有歧义的引用让它们变清晰。在上面的示例中，src.foo将是对表列的一种无歧义的引用。要创建对一个变量的无歧义引用，在一个被标记的块中声明它并且使用块的标签（见第 6.2 节 “PL/uxSQL的结构”。如下所示。

```
<<block>>
DECLARE
  foo int;
BEGIN
  foo := ...;
  INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

这里block.foo表示变量，即使在src中有一个列foo。函数参数以及诸如FOUND的特殊变量，都能通过函数的名称被限定，因为它们被隐式地声明在一个带有该函数名称的外层块中。

有时候在一个大型的PL/uxSQL代码体中修复所有的有歧义引用是不现实的。在这种情况下，可以指定PL/uxSQL应该将有歧义的引用作为变量或表列（这与某些其他系统兼容，例如Oracle）解决。

要在系统范围内改变这种行为，将配置参数pluxsql.variable_conflict设置为error、use_variable或者use_column（这里error是出厂设置）之一。这个参数会影响PL/uxSQL函数中语句的后续编译，但是不会影响在当前会话中已经编译过的语句。因为改变这个设置能够导致PL/uxSQL函数中行为的意想不到的改变，所以只能由一个超级用户来更改它。

可以对逐个函数设置该行为，做法是在函数文本的开始插入这些特殊命令之一，如下所示。

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

这些命令只影响它们所属的函数，并且会覆盖pluxsql.variable_conflict的设置。如下所示。

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
  #variable_conflict use_variable
  DECLARE
    curtime timestamp := now();
  BEGIN
    UPDATE users SET last_modified = curtime, comment = comment
      WHERE users.id = id;
  END;
$$ LANGUAGE pluxsql;
```

在UPDATE命令中，curtime、comment以及id将引用该函数的变量和参数，不管users有没有这些名称的列。注意，不得不在WHERE子句中对users.id的引用加以限定，以便让它引用表列。但是不需要在UPDATE列表中把对comment的引用限定为一个目标，因为语法上那必须是users的一列。可以用下面的方式写一个相同的不依赖于variable_conflict设置的函数，如下所示。

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
  <<fn>>
  DECLARE
    curtime timestamp := now();
  BEGIN
    UPDATE users SET last_modified = fn.curtime, comment = stamp_user.comment
      WHERE users.id = stamp_user.id;
  END;
$$ LANGUAGE pluxsql;
```

被交给EXECUTE或其变体的命令字符串中不会发生变量替换。如果需要插入一个变化值到这样一个命令中，在构建该字符串值时就这样做，或者使用USING，如第 6.5.4 节 “执行动态命令”中所阐明的。

当前变量替换只能在SELECT、INSERT、UPDATE和DELETE命令中工作，因为主 SQL 引擎只允许查询参数在这些命令中。要在其他语句类型（通常被称为实用语句）中使用一个非常量名称或值，必须将实用语句构建为一个字符串并且EXECUTE它。

## 6.11.2. 计划缓存

在函数被第一次调用时（在每个会话中），PL/uxSQL解释器解析函数的源文本并且产生一个内部的二进制指令树。该指令树完全翻译了PL/uxSQL语句结构，但是该函数中使用的SQL表达式以及SQL命令并没有被立即翻译。

作为该函数中每一个表达式和第一次被执行的SQL命令，PL/uxSQL解释器使用SPI管理器的SPI_prepare函数解析并且分析该命令来创建一个预备语句。对于那个表达式或命令的后续访问将会重用该预备语句。因此，一个带有很少被访问的条件性代码路径的函数将永远不会发生分析那些在当前会话中永远不被执行的命令的开销。一个缺点是在一个特定表达式或命令中的错误将不能被检测到，直到函数的该部分在执行时被到达（不重要的语法错误在初始的解析中就会被检测到，但是任何更深层次的东西将只有在执行时才能检测到）。

PL/uxSQL（或者更准确地说是 SPI 管理器）能进一步尝试缓冲与任何特定预备语句相关的执行计划。如果没有使用一个已缓存的计划，那么每次访问该语句时都会生成一个新的执行计划，并且当前的参数值（也就是PL/uxSQL的变量值）可以被用来优化被选中的计划。如果该语句没有参数，或者要被执行很多次，SPI 管理器将考虑创建一个不依赖特定参数值的一般计划并且将其缓存用于重用。通常只有在执行计划对其中引用的PL/uxSQL变量值不那么敏感时，才会这样做。如果这样做，每一次生成的计划就是纯利。关于预备语句的行为请详见《优炫数据库参考手册 V2.1》“SQL命令”章节中的PREPARE。

由于PL/uxSQL保存预备语句并且有时候以这种方式保存执行计划，直接出现在一个PL/uxSQL函数中的 SQL 命令必须在每次执行时引用相同的表和列。也就是说，不能在一个 SQL 命令中把一个参数用作表或列的名字。要绕过这种限制，可以构建PL/uxSQL EXECUTE使用的动态命令，但是会付出在每次执行时需要执行新解析分析以及构建新执行计划的代价。

记录变量的易变天性在这个关系中带来了另一个问题。当一个记录变量的域被用在表达式或语句中时，域的数据类型不能在该函数的调用之间改变，因为每一个表达式被分析时都将使用第一次到达该表达式时存在的数据类型。必要时，可以用EXECUTE来绕过这个问题。

如果同一个函数被用作一个服务于多个表的触发器，PL/uxSQL会为每一个这样的表独立地准备并缓存语句 — 也就是对每一种触发器函数和表的组合都会有一个缓存，而不是每个函数一个缓存。这减轻了数据类型变化带来的问题。例如，一个触发器函数将能够成功地使用一个名为key的列工作，即使该列正好在不同的表中有不同的类型。

同样，具有多态参数类型的函数也会为它们已经被调用的每一种实参类型组合都保留一个独立的缓存，这样数据类型差异不会导致意想不到的失败。

语句缓存有时可能在解释时间敏感的值时产生意外的效果。例如这两个函数做的事情就有区别，如下所示。

- 在logfunc1中，UXsinoDB的主解析器在分析INSERT时就知道字符串'now'应该被解释为timestamp，因为logtable的目标列是这种类型。因此，在INSERT被分析时'now'将被转换为一个timestamp常量，并且在该会话的生命周期内被用于所有对logfunc1的调用。一般不建议这样使用，建议使用now()或current_timestamp函数。

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
  BEGIN
    INSERT INTO logtable VALUES (logtxt, 'now');
  END;
$$ LANGUAGE pluxsql;
```

- 在logfunc2中，UXsinoDB的主解析器不知道'now'应该变成什么类型并且因此返回一个text类型的包含字符串now的数据值。在确定对本地变量curtime的赋值期间，PL/uxSQL解释器通过调用用于转换的text_out以及timestamp_in函数将这个字符串造型为timestamp类型。因此，计算出来的时间戳会按照程序员的期待在每次执行时更新。虽然这正好符合预期，但是它的效率很差，因此使用now()函数仍然是一种更好的方案。

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
  DECLARE
    curtime timestamp;
  BEGIN
    curtime := 'now';
    INSERT INTO logtable VALUES (logtxt, curtime);
  END;
$$ LANGUAGE pluxsql;
```

# 6.12. PL/uxSQL开发提示

在PL/uxSQL中进行开发的一种好方法是使用自己选择的文本编辑器来创建函数，并且在另一个窗口中使用uxsql来载入并且测试那些函数。如果正在这样做，使用CREATE OR REPLACE FUNCTION来编写函数是一个好主意。用那种方式只需要重载该文件来更新函数的定义。如下所示。

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
    ....
$$ LANGUAGE pluxsql;
```

在运行uxsql期间，可以用下面的命令载入或者重载这样一个函数定义文件。

```
\i filename.sql
```

并且接着立即发出 SQL 命令来测试该函数。

另一种在PL/uxSQL中开发的方式是用一个 GUI 数据库访问工具，它能方便对过程语言的开发。这种工具的一个示例是uxdbAdmin。这些工具通常提供方便的特性，例如转义单引号以及便于重新创建和调试函数。

## 6.12.1. 处理引号

一个PL/uxSQL函数的代码在一个CREATE FUNCTION中被指定为一个字符串。如果用通常的方式把该字符串写在单引号中间，那么该函数体中的任何单引号都必须被双写；同样任何反斜线也必须被双写（假定使用了转义字符串语法）。双写引号最多有点冗长，并且在更复杂的情况中代码会变得完全无法理解，因为很容易发现需要半打或者更多相邻的引号。推荐把函数体写成一个“美元引用”的字符串。在美元引用方法中，从不需要双写任何引号。但是要注意为需要的每一层嵌套选择一个不同的美元引用定界符。例如，CREATE FUNCTION命令如下所示。

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
        ....
$PROC$ LANGUAGE pluxsql;
```

在这里面，可以在 SQL 命令中为简单字符串使用引号并且用$$来界定被组装成字符串的 SQL 命令片段。如果需要引用包括$$的文本，可以使用$Q$等等。

下列图表展示了在写没有美元引用的引号时需要做什么。在将之前用美元引用的代码翻译成更容易理解的代码时，它们会有所帮助。

- 1 个引号

  用来开始和结束函数体，如下所示。

  ```
  CREATE FUNCTION foo() RETURNS integer AS '
          ....
  ' LANGUAGE pluxsql;
  ```

  在一个单引号引用的函数体中的任何位置，引号必须成对出现。

- 2 个引号

  用于函数体内的字符串，如下所示。

  ```
  a_output := "Blah";
  SELECT * FROM users WHERE f_name="foobar";
  ```

  在美元引用方法中，如下所示。

  ```
  a_output := 'Blah';
  SELECT * FROM users WHERE f_name='foobar';
  ```

  这恰好就是PL/uxSQL在两种情况中会看到的。

- 4 个引号

  当函数内的一个字符串常量中需要一个单引号时，如下所示。

  ```
  a_output := a_output || " AND name LIKE ""foobar"" AND xyz"
  ```

  实际会被追加到a_output的值将是： AND name LIKE 'foobar' AND xyz。

  使用美元引用方法，要注意在这周围的任何美元引用定界符不只是$$，如下所示。

  ```
  a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
  ```

- 6 个引号

  当在函数体内的一个字符串中的一个单引号与该字符串常量末尾相邻，如下所示。

```
a_output := a_output || " AND name LIKE ""foobar"""
```

被追加到a_output的值则将是： AND name LIKE 'foobar'。

在美元引用方法中，进行如下使用。

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

- 10 个引号

  当在一个字符串常量（占 8 个引号）中有两个单引号时并且这会挨着该字符串常量的末尾（另外 2 个）。如果正在写一个产生其他函数的函数（如例 6.10 “从PL/SQL移植一个创建另一个函数的函数到PL/uxSQL”中），将很可能只需要这种。如下所示。

```
a_output := a_output || " if v_" ||
   referrer_keys.kind || " like """""""
   || referrer_keys.key_string || """""""
   then return """  || referrer_keys.referrer_type
   || """; end if;";
```

a_output的值将是下面内容。

```
if v_... like "..." then return "..."; end if;
```

在美元引用方法中，进行如下使用。

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
   || referrer_keys.key_string || $$'
   then return '$$  || referrer_keys.referrer_type
   || $$'; end if;$$;
```

这里假定只需要把单引号放在a_output中，因为在使用前它将被再引用。

## 6.12.2. 额外的编译时和运行时检查

为了辅助用户在一些简单但常见的问题产生危害之前找到它们，PL/uxSQL提供了额外的检查。当被启用时，根据配置，它们可以在一个函数的编译期间被用来发出WARNING或者ERROR。一个已经收到了WARNING的函数可以被继续执行而不会产生进一步的消息，因此建议在一个单独的开发环境中进行测试。

根据需要设置 pluxsql.extra_warnings 或 pluxsql.extra_errors，适当情况下，在开发和/或测试环境中可以设置为 "all"。

这些附加检查通过配置变量启用， pluxsql.extra_warnings用于警告，pluxsql.extra_errors 用于错误。 两者都可以设置为逗号分隔的检查列表，"none" 或 "all"。 默认值为"none"。当前可用的检查列表如下所示。

- shadowed_variables

  检查声明是否遮盖了以前定义的变量。

- strict_multi_assignment

某些PL/uxSQL命令允许一次将值分配给多个变量，例如SELECT INTO。 通常，目标变量的数量和源变量的数量应匹配，尽管PL/uxSQL将使用NULL来处理缺失的值和被忽略的额外变量。 启用此检查将导致 PL/uxSQL在目标变量数和源变量数不同时引发WARNING或ERROR。

- too_many_rows

  启用此检查将导致PL/uxSQL检查在使用INTO子句时给定查询是否返回多行。 由于INTO语句只会使用一行，让查询返回多行通常会效率低下和/或不确定性，因此很可能会出现错误。

下面的示例显示了pluxsql.extra_warnings 设置为shadowed_variables的效果。

```
SET pluxsql.extra_warnings TO 'shadowed_variables';

CREATE FUNCTION foo(f1 int) RETURNS int AS $$
DECLARE
f1 int;
BEGIN
RETURN f1;
END
$$ LANGUAGE pluxsql;
WARNING:  variable "f1" shadows a previously defined variable
LINE 3: f1 int;
        ^
CREATE FUNCTION
```

下面的示例显示了将pluxsql.extra_warnings 设置为strict_multi_assignment：

```
SET pluxsql.extra_warnings TO 'strict_multi_assignment';

CREATE OR REPLACE FUNCTION public.foo()
 RETURNS void
 LANGUAGE pluxsql
AS $$
DECLARE
  x int;
  y int;
BEGIN
  SELECT 1 INTO x, y;
  SELECT 1, 2 INTO x, y;
  SELECT 1, 2, 3 INTO x, y;
END;
$$;

SELECT foo();
WARNING:  number of source and target fields in assignment does not match
DETAIL:  strict_multi_assignment check of extra_warnings is active.
HINT:  Make sure the query returns the exact list of columns.
WARNING:  number of source and target fields in assignment does not match
DETAIL:  strict_multi_assignment check of extra_warnings is active.
HINT:  Make sure the query returns the exact list of columns.

 foo
```

---

(1 row)

# 6.13. 从Oracle PL/SQL 移植

这一节解释了UXsinoDB的PL/uxSQL语言和 Oracle 的PL/SQL语言之间的差别，用以帮助那些从Oracle®向UXsinoDB移植应用的人。

PL/uxSQL与 PL/SQL 在许多方面都非常类似。它是一种块结构的、命令式的语言并且所有变量必须先被声明。赋值、循环和条件则很类似。在从PL/SQL向PL/uxSQL移植时必须遵守一些事情，如下所示。

- 如果一个 SQL 命令中使用的名字可能是一个表的列名或者是对一个函数中变量的引用，那么PL/SQL会将它当作一个列名。如第 6.11.1 节 “变量替换”中所述，这对应的是PL/uxSQL的pluxsql.variable_conflict = use_column行为（不是默认行为）。通常最好是首先避免这种歧义，但如果不得不移植依赖于该行为的大量代码，那么设置variable_conflict将是最好的方案。

- 在UXsinoDB中，函数体必须写成字符串文本。因此需要使用美元符引用或者转义函数体中的单引号（见第 6.12.1 节 “处理引号”）。

- 数据类型名称常常需要翻译。例如，在 Oracle 中字符串值通常被声明为类型varchar2，这并非 SQL 标准类型。在UXsinoDB中则要使用类型varchar或者text来替代。类似地，要把类型number替换成numeric，或者在适当的时候使用某种其他数字数据类型。

- 应该用模式把函数组织成不同的分组，而不是用包。

- 因为没有包，所以也没有包级别的变量。可以在临时表里保存会话级别的状态。

- 带有REVERSE的整数FOR循环的工作方式不同：PL/SQL中是从第二个数向第一个数倒数，而PL/uxSQL是从第一个数向第二个数倒数，因此在移植时需要交换循环边界。不幸的是这种不兼容性是不太可能改变的（见第 6.6.5.5 节 FOR（整型变体）”）。

- 查询上的FOR循环（不是游标）的工作方式同样不同：目标变量必须已经被声明，而PL/SQL总是会隐式地声明它们。但是这样做的优点是在退出循环后，变量值仍然可以访问。

- 在使用游标变量方面，存在一些记法差异。

## 6.13.1. 移植示例

例 6.9 “从PL/SQL移植一个简单的函数到PL/uxSQL”展示了如何从PL/SQL移植一个简单的函数到PL/uxSQL中。

### 例 6.9. 从PL/SQL移植一个简单的函数到PL/uxSQL

这里有一个Oracle PL/SQL函数示例，如下所示。

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar2,
                                    v_version varchar2)
RETURN varchar2 IS
BEGIN
  IF v_version IS NULL THEN
    RETURN v_name;
```

```
  END IF;
  RETURN v_name || '/' || v_version;
END;
/
show errors;
```

过一遍这个函数并且看看与PL/uxSQL相比有什么样的不同，如下所示。

- 类型名称varchar2被改成了varchar或者text。在这一节的示例中，将使用varchar，但如果不需要特定的字符串长度限制，text常常是更好的选择。
- 在函数原型中（不是函数体中）的RETURN关键字在UXsinoDB中变成了RETURNS。还有，IS变成了AS，并且还需要增加一个LANGUAGE子句，因为PL/uxSQL并非唯一可用的函数语言。
- 在UXsinoDB中，函数体被认为是一个字符串，所以需要使用引号或者美元符号包围它。这代替了Oracle 方法中的用于终止的/。
- 在UXsinoDB中没有show errors命令，并且也不需要这个命令，因为错误是自动报告的。

这个函数被移植到UXsinoDB后，结果如下所示。

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                    v_version varchar)
RETURNS varchar AS $$
BEGIN
  IF v_version IS NULL THEN
    RETURN v_name;
  END IF;
  RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE pluxsql;
```

例　6.10 “从PL/SQL移植一个创建另一个函数的函数到PL/uxSQL”展示了如何移植一个会创建另一个函数的函数，以及如何处理引号问题。

### 例　6.10.　从PL/SQL移植一个创建另一个函数的函数到PL/uxSQL

下面的过程从一个SELECT语句抓取行，并且为了效率而构建一个带有IF语句中结果的大型函数。

Oracle 版本如下所示。

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
  CURSOR referrer_keys IS
    SELECT * FROM cs_referrer_keys
    ORDER BY try_order;
  func_cmd VARCHAR(4000);
BEGIN
  func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR2,
          v_domain IN VARCHAR2, v_url IN VARCHAR2) RETURN VARCHAR2 IS BEGIN';

  FOR referrer_key IN referrer_keys LOOP
    func_cmd := func_cmd ||
```

```
        ' IF v_' || referrer_key.kind
        || ' LIKE ''' || referrer_key.key_string
        || ''' THEN RETURN ''' || referrer_key.referrer_type
        || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

UXsinoDB版本如下所示。

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ' || quote_literal(referrer_key.key_string)
          || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
          || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
      'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                            v_domain varchar,
                                            v_url varchar)
        RETURNS varchar AS '
      || quote_literal(func_body)
      || ' LANGUAGE pluxsql;' ;

    EXECUTE func_cmd;
END;
$func$ LANGUAGE pluxsql;
```

请注意函数体是如何被单独构建并且通过quote_literal被传递以双写其中的任何引号。需要这个技术是因为无法安全地使用美元引用定义新函数：不确定从*referrer_key.key_string*域中来的什么字符串会被插入（这里假定*referrer_key.kind*可以确信总是为host、domain或者url，但是*referrer_key.key_string*可能是任何东西，特别是它可能包含美元符号）。这个函数实际上是在Oracle 的原版上的改进，因为当*referrer_key.key_string*或者*referrer_key.referrer_type*包含引号时，它将不会生成坏掉的代码。

例 6.11 “从PL/SQL移植一个带有字符串操作以及OUT参数的过程到PL/uxSQL”展示了如何移植一个带有OUT参数和字符串处理的函数。UXsinoDB没有内建的instr函数，但是可以用其它函数的组合来创建一个。在第 6.13.3 节 “附录”中有一个instr的PL/uxSQL实现，可以用它让移植变得更容易。

## 例 6.11. 从PL/SQL移植一个带有字符串操作以及OUT参数的过程到PL/uxSQL

下面的Oracle PL/SQL 过程被用来解析一个 URL 并且返回一些元素（主机、路径和查询）。

Oracle 版本如下所示。

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR2,
    v_host OUT VARCHAR2,  -- 这将被传回去
    v_path OUT VARCHAR2,  -- 这个也是
    v_query OUT VARCHAR2) -- 还有这个
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

这里是一种到PL/uxSQL的可能翻译，如下所示。

```
CREATE OR REPLACE FUNCTION cs_parse_url(
```

```
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- 这将被传递回去
    v_path OUT VARCHAR,  -- 这个也是
    v_query OUT VARCHAR) -- 以及这个
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE pluxsql;
```

这个函数可以使用如下方式。

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

例　6.12 “从PL/SQL移植一个过程到PL/uxSQL”展示了如何移植一个使用了多种 Oracle 特性的过程。

## 例　6.12.　从PL/SQL移植一个过程到PL/uxSQL

Oracle 版本如下所示。

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;
```

```
    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- 释放锁
        raise_application_error(-20000,
                'Unable to create a new job: a job is currently running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN dup_val_on_index THEN NULL; -- 如果已经存在也不用担心
    END;
    COMMIT;
END;
/
show errors
```

如何将这个过程移植到PL/uxSQL，如下所示。

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- 释放锁
        RAISE EXCEPTION 'Unable to create a new job: a job is currently running'; --1
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN --2
            -- 如果已经存在不要担心
    END;
    COMMIT;
END;
$$ LANGUAGE pluxsql;
```

1. RAISE的语法与 Oracle 的语句相当不同，尽管基本的形式RAISE *exception_name*工作起来是相似的。

2. PL/uxSQL所支持的异常名称不同于 Oracle。内建的异常名称集合要更大。目前没有办法声明用户定义的异常名称，尽管能够抛出用户选择的 SQLSTATE 值。

# 6.13.2. 其他要关注的事项

这一节解释了在移植 Oracle PL/SQL函数到UXsinoDB中时要关注的一些其他问题。

## 6.13.2.1. 异常后隐式回滚

在PL/uxSQL，当一个异常被EXCEPTION子句捕获之后，从该块的BEGIN以来的所有数据库改变都会被自动回滚。也就是，该行为等效于在 Oracle 中用下面的代码得到的效果。

```
BEGIN
  SAVEPOINT s1;
  ... 代码 ...
EXCEPTION
  WHEN ... THEN
    ROLLBACK TO s1;
    ... 代码 ...
  WHEN ... THEN
    ROLLBACK TO s1;
    ... 代码 ...
END;
```

如果正在翻译一个使用这种风格的SAVEPOINT以及ROLLBACK TO的 Oracle 过程，只要忽略掉SAVEPOINT以及ROLLBACK TO。如果 Oracle 过程是以不同的方法使用SAVEPOINT以及ROLLBACK TO。

## 6.13.2.2. EXECUTE

PL/uxSQL的EXECUTE与PL/SQL中的工作相似，但是必须要记住按照第 6.5.4 节 “执行动态命令”中所述地使用quote_literal以及quote_ident。EXECUTE 'SELECT * FROM $1';类型的结构将无法可靠地工作除非使用这些函数。

## 6.13.2.3. 优化 PL/uxSQL 函数

UXsinoDB提供了两种函数创建修饰符来优化执行：“volatility”（对于给定的相同参数，函数是否总是返回相同的结果）以及“strictness”（如果任何参数为空，函数是否返回空）。

在利用这些优化属性时，CREATE FUNCTION语句如下所示。

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE pluxsql STRICT IMMUTABLE;
```

# 6.13.3. 附录

这一节包含了一组 Oracle 兼容的instr函数代码，可以用它来简化移植工作。

```
--
-- instr 函数模仿 Oracle 的对应函数
-- 语法: instr(string1, string2 [, n [, m]])
-- 其中 [] 表示可选参数。
```

```
--
-- 从第n个字符开始搜索string1，要求找到string2的第m次出现。
-- 如果n为负，则从后向前搜索，从string1的末尾开始的第abs(n)个字符开始。
-- 如果没有传n，假定它为1（从第1个字符开始搜索）。
-- 如果没有传m，假定它为1（找第1次出现）。
-- 在string1中返回string2的开始索引，如果没有找到string2则为0。
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
BEGIN
   RETURN instr($1, $2, 1);
END;
$$ LANGUAGE pluxsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                beg_index integer)
RETURNS integer AS $$
DECLARE
   pos integer NOT NULL DEFAULT 0;
   temp_str varchar;
   beg integer;
   length integer;
   ss_length integer;
BEGIN
   IF beg_index > 0 THEN
      temp_str := substring(string FROM beg_index);
      pos := position(string_to_search_for IN temp_str);

      IF pos = 0 THEN
         RETURN 0;
      ELSE
         RETURN pos + beg_index - 1;
      END IF;
   ELSIF beg_index < 0 THEN
      ss_length := char_length(string_to_search_for);
      length := char_length(string);
      beg := length + 1 + beg_index;

      WHILE beg > 0 LOOP
         temp_str := substring(string FROM beg FOR ss_length);
         IF string_to_search_for = temp_str THEN
            RETURN beg;
         END IF;

         beg := beg - 1;
      END LOOP;

      RETURN 0;
   ELSE
      RETURN 0;
   END IF;
END;
$$ LANGUAGE pluxsql STRICT IMMUTABLE;
```

```
CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
   pos integer NOT NULL DEFAULT 0;
   occur_number integer NOT NULL DEFAULT 0;
   temp_str varchar;
   beg integer;
   i integer;
   length integer;
   ss_length integer;
BEGIN
   IF occur_index <= 0 THEN
      RAISE 'argument "%" is out of range', occur_index
       USING ERRCODE = '22003';
   END IF;

   IF beg_index > 0 THEN
      beg := beg_index - 1;
      FOR i IN 1..occur_index LOOP
         temp_str := substring(string FROM beg + 1);
         pos := position(string_to_search_for IN temp_str);
         IF pos = 0 THEN
            RETURN 0;
         END IF;
         beg := beg + pos;
      END LOOP;

      RETURN beg;
   ELSIF beg_index < 0 THEN
      ss_length := char_length(string_to_search_for);
      length := char_length(string);
      beg := length + 1 + beg_index;

      WHILE beg > 0 LOOP
         temp_str := substring(string FROM beg FOR ss_length);
         IF string_to_search_for = temp_str THEN
            occur_number := occur_number + 1;
            IF occur_number = occur_index THEN
               RETURN beg;
            END IF;
         END IF;

         beg := beg - 1;
      END LOOP;

      RETURN 0;
   ELSE
      RETURN 0;
   END IF;
END;
$$ LANGUAGE pluxsql STRICT IMMUTABLE;
```

# 第 7 章 PL/Tcl - Tcl 过程语言

PL/Tcl 是一种用于UXsinoDB数据库系统的可载入过程语言， 它可以让Tcl 语言被用来编写UXsinoDB函数。

## 7.1. 概述

PL/Tcl 提供了大部分函数编写者在 C 语言中能够获得的能力，虽然有一些限制，但是却额外提供了 Tcl 中强大的字符串处理库。

一种强制性的好限制是所有被执行的东西都处于 Tcl 解释器的安全上下文中。除了安全 Tcl 的有限的命令集合之外，只有几个通过 SPI 访问数据库的命令以及通过elog()产生消息的命令。PL/Tcl 没有提供访问数据库服务器内部或者在UXsinoDB服务器进程权限之下得到 OS-级访问的方法，而 C 函数是可以那样做的。因此，非特权数据库用户可以使用这种语言，它不会给予他们无限制的权利。

其他值得注意的实现限制是 Tcl 函数不能被用来创建新数据类型的输入/输出函数。

有时候要编写不受安全 Tcl 限制的 Tcl 函数。例如，可能想要一个能发送电子邮件的 Tcl 函数。要处理这些情况，可以使用一种PL/Tcl的变体，它被称为PL/TclU（用于非可信 Tcl）。它其实是完全相同的一种语言，不过它使用了一个完整的 Tcl 解释器。如果使用了PL/TclU，它必须被安装为一种非可信的过程语言，这样只有数据库超级用户可以用它来创建函数。PL/TclU函数的编写者必须注意该函数不能被用来做其设计目的之外的事情，因为该函数能做一个作为数据库管理员登录的用户可以做的任何事情。

如果在安装过程的配置步骤中指定了 Tcl 支持，PL/Tcl以及PL/TclU调用处理器的共享对象代码会被自动编译并且被安装在UXsinoDB的库目录中。要在一个特定数据库中安装PL/Tcl或者PL/TclU，请使用**CREATE EXTENSION**命令，例如CREATE EXTENSION pltcl或者CREATE EXTENSION pltclu。

## 7.2. PL/Tcl 函数和参数

要用PL/Tcl语言创建函数，使用标准的**CREATE FUNCTION**语法，如下所示。

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
  # PL/Tcl function body
$$ LANGUAGE pltcl;
```

PL/TclU的函数是一样的语法，只是语言被指定为pltclu。

函数的主体就是一个 Tcl 脚本。当函数被调用时，参数值会被以变量名$1 ... $*n*传递给该Tcl脚本。结果会以常见的方式通过一个return语句从 Tcl 脚本中返回。在一个过程中，Tcl代码的返回值会被忽略。

例如，一个返回两个整数值中较大值的函数可以进行如下定义。

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
  if {$1 > $2} {return $1}
  return $2
$$ LANGUAGE pltcl STRICT;
```

注意子句STRICT，不用去操心空输入值：如果空值被传入，函数根本就不会被调用，而是自动地返回一个空结果。

在非严格函数中，如果一个参数的实际值为空，对应的$*n*变量将被设置为一个空串。为了检测一个特定参数是否为空，可使用函数argisnull。例如，想要带有一个空参数和一个非空参数并且返回非空参数的tcl_max，如下所示。

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
   if {[argisnull 1]} {
      if {[argisnull 2]} { return_null }
      return $2
   }
   if {[argisnull 2]} { return $1 }
   if {$1 > $2} {return $1}
   return $2
$$ LANGUAGE pltcl;
```

如上所述，要从一个 PL/Tcl 函数返回空值，可执行return_null。不管函数是严格还是非严格都可以这样做。

组合类型参数会被作为 Tcl 数组传递给函数。该数组的元素名就是组合类型的属性值。如果被传入行的一个属性为空值，它不会出现在数组中。如下所示，

```
CREATE TABLE employee (
   name text,
   salary integer,
   age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
   if {200000.0 < $1(salary)} {
      return "t"
   }
   if {$1(age) < 30 && 100000.0 < $1(salary)} {
      return "t"
   }
   return "f"
$$ LANGUAGE pltcl;
```

PL/Tcl函数也能返回组合类型的结果。要返回组合类型结果，Tcl代码必须返回匹配预期结果类型的“列名/值”对的列表。任何从该列表中省略的列名将被返回为空，如果有预期之外的列名则会报出错误。如下所示。

```
CREATE FUNCTION square_cube(in int, out squared int, out cubed int) AS $$
   return [list squared [expr {$1 * $1}] cubed [expr {$1 * $1 * $1}]]
$$ LANGUAGE pltcl;
```

过程的输出参数以相同的方式返回，如下所示。

```
CREATE PROCEDURE tcl_triple(INOUT a integer, INOUT b integer) AS $$
   return [list a [expr {$1 * 3}] b [expr {$2 * 3}]]
```

```
$$ LANGUAGE pltcl;

CALL tcl_triple(5, 10);
```

> **提示**
>
> 结果列表可以用Tcl的array get命令从想得到的元组的数组表示中造出。如下所示。
>
> ```
> CREATE FUNCTION raise_pay(employee, delta int) RETURNS employee AS
>  $$
>    set 1(salary) [expr {$1(salary) + $2}]
>    return [array get 1]
> $$ LANGUAGE pltcl;
> ```

PL/Tcl函数能够返回集合。要返回集合，Tcl代码应该对每一个要返回的行调用一次return_next，在返回标量类型时传入合适的值或者在返回组合类型时传入“列名/值”堆的列表。返回标量类型，如下所示。

```
CREATE FUNCTION sequence(int, int) RETURNS SETOF int AS $$
   for {set i $1} {$i < $2} {incr i} {
      return_next $i
   }
$$ LANGUAGE pltcl;
```

返回组合类型，如下所示。

```
CREATE FUNCTION table_of_squares(int, int) RETURNS TABLE (x int, x2 int) AS $$
   for {set i $1} {$i < $2} {incr i} {
      return_next [list x $i x2 [expr {$i * $i}]]
   }
$$ LANGUAGE pltcl;
```

# 7.3. PL/Tcl 中的数据值

提供给 PL/Tcl 函数代码的参数值是输入参数简单转换而成的文本形式（就像被SELECT语句显示的那样）。反过来，return和return_next命令将接受任何字符串，只要它是该函数声明的返回类型的可接受的输入格式，或者是组合结果类型的指定列的可接受输入格式。

# 7.4. PL/Tcl 中的全局数据

有时候需要在同一个函数的两次调用间保持某些全局数据或者在不同的函数之间共享全局数据。在 PL/Tcl 中这很容易做到，但是必须了解一些限制。

由于安全性原因，PL/Tcl 会为一个 SQL 角色建立一个单独的 Tcl 解释器来执行该角色调用的函数。这可以避免一个用户无意或者恶意地干涉另一个用户的 PL/Tcl 函数的行为。对任何“全局” Tcl 变量，每一个这样的解释器都有其自身的值。因此，当且仅当两个 PL/Tcl 函数由用一个 SQL 角色执行时，它们才能共享相同的全局变量。在使用单个会话执行多个 SQL 角色的

代码（通过SECURITY DEFINER函数、使用SET ROLE等）的应用中，需要采取显式的步骤以保证 PL/Tcl 函数能共享数据。要这样做，需要确保要通信的函数都属于同一个用户，并且把它们标记为SECURITY DEFINER。当然，要小心这样的函数被滥用。

在一个会话中使用的所有 PL/TclU 函数都在同一个 Tcl 解释器中执行，这当然与用于 PL/Tcl 函数的解释器不同。因此，在 PL/TclU 函数之间会自动地共享全局数据。这并不是一种安全性风险，因为所有的 PL/TclU 函数都在同样的信任级别上执行，即都以数据库超级用户的级别执行。

为了保护 PL/Tcl 函数不会无意间彼此干扰，通过upvar命令可以建立一个对每个函数可用的全局数组。这个变量的全局名称是该函数的内部名称，并且本地名称为GD。推荐使用GD来保持一个函数的持久私有数据。在多个函数之间共享的值使用常规的 Tcl 全局变量（注意GD数组只在一个特定的解释器中是全局的，因此它们不会绕过上文提到的安全性限制）。

下文的spi_execp示例中有一个使用GD的示例。

# 7.5. 从 PL/Tcl 访问数据库

下面的命令可以用来从 PL/Tcl 函数体中访问数据库。

- spi_exec [-count *n*] [-array *name*] *command* [*loop-body*]

  执行一个以字符串给出的 SQL 命令。命令中错误将会导致错误发生。否则，spi_exec的返回值是被命令处理的行数（选择、插入、更新或者删除），如果命令是一条功能性语句则返回零。此外，如果命令是一条SELECT语句，被选中的列的值会被放在上文所述的 Tcl 变量中。

  可选的-count值告诉spi_exec命令中要处理的最大行数。这种效果类似于用游标建立一个查询然后使用FETCH *n*。

  如果命令是一条SELECT语句，结果列的值会被放在以列名命名的 Tcl 变量中。如果给出了-array选项，列值会被存储在所提及的关联数组的元素中，而列名则被用作数组索引。此外，结果中的当前行号（从零开始记）被存储在数组元素“.tupno”中，除非这个名字在结果中已经被用作一个列名。

  如果命令是一条SELECT语句并且没有给出*loop-body*脚本，则只有结果的第一行被存储在 Tcl 变量或数组元素中。如果结果中有剩余的行，它们会被忽略。如果查询不返回行则不存储任何东西（这种情况可以通过spi_exec的结果检测到）。如下所示。

  ```
  spi_exec "SELECT count(*) AS cnt FROM ux_proc"
  ```

  将把 Tcl 变量$cnt设置为ux_proc系统目录中的行数。

  如果给出了可选的*loop-body*参数，它会是一个 Tcl 脚本，对查询结果中的每一行都要执行这个脚本（如果给出的查询不是SELECT则忽略*loop-body*）。在每次迭代前当前行的列值会被存储在 Tcl 变量或数组元素中。如下所示。

  ```
  spi_exec -array C "SELECT * FROM ux_class" {
    elog DEBUG "have table $C(relname)"
  }
  ```

  会对ux_class的每一行打印一段日志消息。这种特性工作起来类似于其他的 Tcl 循环结构。特别是continue和break的动作方式与在循环体中的通常方式相同。

如果一个查询结果的一列为空，为它准备的目标变量不会被建立，而是会被“unset”。

- spi_prepare *query typelist*

  为后面的执行准备并且保存一个查询计划。保存下来的计划将在当前会话的生命期内保持存在。

  查询可以使用参数，也就是占位符。在计划真正被执行时将会为占位符提供值。在查询字符串中，可以用符号$1 ... $*n*引用参数。如果查询使用了参数，参数类型的名称必须以一个 Tcl 列表的形式给出（如果不使用参数，可以为*typelist*写一个空列表）。

  从spi_prepare返回的值是一个查询 ID，在后续的spi_execp调用中需要用到这个 ID。示例可见spi_execp。

- spi_execp [-count *n*] [-array *name*] [-nulls *string*] *queryid* [*value-list*] [*loop-body*]

  执行一个之前用spi_prepare准备的查询。*queryid*是spi_prepare返回的 ID。如果查询引用参数，则必须提供一个*value-list*。这是一个参数实际值的 Tcl 列表。这个列表必须和之前传给spi_prepare的参数类型列表具有相同的长度。如果查询没有参数则可省略*value-list*。

  -nulls的值可选，它是一个空格和'n'字符构成的串，它告诉spi_execp哪些参数是空值。如果给出这个值，它必须正好和*value-list*长度相等。如果没有给出这个值，所有的参数值都是空。

  除了指定查询及其参数的方法，spi_execp和spi_exec很像。-count、-array以及*loop-body*选项是相同的，并且结果值也一样。

  使用预备计划的 PL/Tcl 函数，如下所示。

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
  if {![ info exists GD(plan) ]} {
    # 第一次调用时准备保存的计划
    set GD(plan) [ spi_prepare \
        "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \$2" \
        [ list int4 int4 ] ]
  }
  spi_execp -count 1 $GD(plan) [ list $1 $2 ]
  return $cnt
$$ LANGUAGE pltcl;
```

  需要在给spi_prepare的查询字符串里放上反斜线来确保$*n*标记会被原样传递给spi_prepare，并且不会被 Tcl 变量替换。

- subtransaction *command*

  *command*中包含的Tcl脚本会被在一个SQL子事务中执行。如果该脚本返回一个错误，那么整个子事务会在把错误返回到外围的Tcl代码之前就回滚。更多细节和示例请参见第 7.9 节 “PL/Tcl中的显式子事务”。

- quote *string*

  在给定的字符串中双写所有单引号和反斜线字符。这可以被用来引用字符串，以便它们能被安全地插入到传给spi_exec或者spi_prepare的 SQL 命令字符串中。例如，考虑这样的 SQL 命令字符串，如下所示。

"SELECT '$val' AS ret"

这里的 Tcl 变量val实际上包含doesn't。这将会导致最终的命令串，如下所示。

SELECT 'doesn't' AS ret

这种命令串会导致spi_exec或spi_prepare期间的解析错误。要正确地工作，提交的命令，如下所示。

SELECT 'doesn''t' AS ret

在 PL/Tcl 中可以进行如下操作。

"SELECT '[ quote $val ]' AS ret"

spi_execp的一个好处是不必这样引用参数值，因为参数值不会被作为 SQL 命令串的一部分被解析。

- elog *level msg*

  发出一段日志或者错误消息。可能的级别是DEBUG、LOG、INFO、NOTICE、WARNING、ERROR以及FATAL。ERROR产生一个错误情况。如果周围的 Tcl 代码没有捕捉它，错误会传播到调用查询，导致当前事务或者子事务被中止。这实际上与 Tcl 的error命令相同。FATAL中止事务并且导致当前会话关闭（可能在 PL/Tcl 函数中没有很好的理由来使用这种错误级别，但是为了完整性还是提供了这种级别）。其他级别只产生不同优先级的消息。一个特定级别的消息是被报告给客户端、写入到服务器日志或者两者都做，是由配置变量log_min_messages和client_min_messages所控制。详见第 7.8 节 “PL/Tcl 中的错误处理”。

# 7.6. PL/Tcl 中的触发器函数

触发器函数也可以用 PL/Tcl 编写。UXsinoDB要求能作为触发器被调用的函数必须被声明为没有参数并且返回类型为trigger。

来自于触发器管理器的信息通过下列变量被传递给函数体，如下所示。

- $TG_name

  CREATE TRIGGER语句中触发器的名字。

- $TG_relid

  导致触发器函数被调用的表的对象 ID。

- $TG_table_name

  导致触发器函数被调用的表的名字。

- $TG_table_schema

  导致触发器函数被调用的表所在的模式。

- $TG_relatts

  表列名的 Tcl 列表，前面放上一个空列表元素。因此用Tcl的lsearch命令在该列表中查找一个列名返回的元素编号会从 1 开始（对于第一列），这和UXsinoDB中的自定义编号是同样的方式（空列表元数也出现在被删除的列的位置上，这样其右边的列的属性编号才是正确的）。

- $TG_when

  可以为BEFORE、AFTER或者INSTEAD OF，具体的选择取决于触发器事件的类型。

- $TG_level

  可以为ROW或者STATEMENT，取决于触发器事件的类型。

- $TG_op

  可以为INSERT、UPDATE、DELETE或者TRUNCATE，取决于触发器事件的类型。

- $NEW

  对于INSERT或者UPDATE动作是一个包含着新表行值的关联数组，对于DELETE为空。该数组以列名为索引。为空的列不会出现在数组中。对于语句级触发器这个变量不会被设置。

- $OLD

  对于UPDATE或者DELETE动作是一个包含着新表行值的关联数组，对于INSERT为空。该数组以列名为索引。为空的列不会出现在数组中。对于语句级触发器这个变量不会被设置。

- $args

  在CREATE TRIGGER语句中对过程给出的参数的 Tcl 列表。在过程体中也可以用$1 ... $*n*来访问这些参数。

一个触发器函数的返回值可以是字符串OK或者SKIP，或者是一个“列名/值”对的列表。如果返回值是OK，引发该触发器的操作（INSERT/UPDATE/DELETE）将正常继续下去。SKIP告诉触发器管理器悄悄地抑制对这一行的该操作。如果返回一个列表，它告诉PL/Tcl返回一个被修改的行给触发器管理器，这个被修改的行的内容由列表中的列名和值指定。该列表中没有提到的任何列会被设置为空。返回被修改行只对行级BEFORE INSERT或UPDATE触发器有意义，对它们来说这个被修改的行将被插入而不是插入$NEW中给出的行。返回被修改行还对行级INSTEAD OF INSERT或UPDATE触发器有意义，其中被返回的行被用作INSERT RETURNING或UPDATE RETURNING子句的源数据。在行级BEFORE DELETE或INSTEAD OF DELETE触发器中，返回一个被修改行的效果和返回OK的效果相同，即操作继续。对所有其他类型的触发器来说，触发器返回值会被忽略。

> **提示**
>
> 结果列表可以用Tcl的array get命令从被修改的元组的数组表示中造出。

这里有一个触发器函数的示例，它用一个表中的整数值来跟踪在行上被执行的更新数。对于被插入的新行，该值被初始化为 0 并且之后在每一次更新操作时被加一。

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
  switch $TG_op {
```

```
    INSERT {
      set NEW($1) 0
    }
    UPDATE {
      set NEW($1) $OLD($1)
      incr NEW($1)
    }
    default {
      return OK
    }
  }
  return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
  FOR EACH ROW EXECUTE FUNCTION trigfunc_modcount('modcnt');
```

注意触发器函数本身不知道列名，列名由触发器参数提供。这让触发器函数可以被重用于不同的表。

# 7.7. PL/Tcl 中的事件触发器函数

事件触发器函数也可以用 PL/Tcl 编写。UXsinoDB要求能作为事件触发器被调用的函数必须被声明为没有参数并且返回类型为event_trigger。

来自于触发器管理器的信息通过下列变量被传递给函数体，如下所示。

$TG_event

触发器为其引发的事件名。

$TG_tag

触发器为其引发的命令标签。

触发器函数的返回值被忽略。

这里是一个事件触发器函数的示例，它在所支持的命令每次执行时简单地产生一个NOTICE消息，如下所示。

```
CREATE OR REPLACE FUNCTION tclsnitch() RETURNS event_trigger AS $$
  elog NOTICE "tclsnitch: $TG_event $TG_tag"
$$ LANGUAGE pltcl;

CREATE EVENT TRIGGER tcl_a_snitch ON ddl_command_start EXECUTE FUNCTION tclsnitch();
```

# 7.8. PL/Tcl 中的错误处理

PL/Tcl 函数中的 Tcl 代码或者从 PL/Tcl 函数中调用的代码可以抛出一个错误，错误可以由执行某些非法操作产生或者通过使用 Tcl error命令或者 PL/Tcl 的elog命令产生。Tcl 中可以使用

Tcl catch命令捕获这类错误。如果一个错误没有被捕捉但是被允许传播到该PL/Tcl函数执行的顶层，它会在该函数的调用查询中被报告为一个SQL错误。

相反，在 PL/Tcl 的spi_exec、spi_prepare以及spi_execp命令中发生的SQL错误会被报告为 Tcl 错误，因此它们也可以被 Tcl 的catch命令捕获（这些PL/Tcl命令中的每一个都在一个子事务中运行它的SQL操作，该子事务在错误时会被回滚，这样任何部分完成的操作也会被自动清除）。同样地，如果一个错误被传播到顶层而没有被捕获，它会转变成SQL错误。

Tcl 提供了一个errorCode变量，它表示有关于一个错误的附加信息，它的格式易于 Tcl 程序解释。该变量的内容符合 Tcl 列表格式，第一个词标识报告该错误的子系统或者库，之后的内容则留给子系统或者库来填充。对于 PL/Tcl 命令报告的数据库错误，第一个词是UXDB，第二个词是 UXsinoDB 的版本号，剩下的部分是域名称/域值构成的对，它们提供有关该错误的详细信息。域SQLSTATE、condition以及message总是会被提供。可能出现的域包括 detail、hint、context、 schema、table、column、 datatype、constraint、statement、cursor_position、 filename、lineno以及 funcname。

使用 PL/Tcl 的errorCode信息的一种便捷方式是把它载入到一个数组中，这样域名称就变成了数组下标，如下所示。

```
if {[catch { spi_exec $sql_command }]} {
   if {[lindex $::errorCode 0] == "UXDB"} {
      array set errorArray $::errorCode
      if {$errorArray(condition) == "undefined_table"} {
         # deal with missing table
      } else {
         # deal with some other type of SQL error
      }
   }
}
```

（双冒号显式地指定errorCode是一个全局变量）。

# 7.9. PL/Tcl中的显式子事务

从第 7.8 节 “PL/Tcl 中的错误处理”中介绍的数据库访问导致的错误中恢复可能导致一种不可取的情况，其中一些操作在它们中的一个失败前成功完成，并且在从错误中恢复过来后数据还处于一种不一致的状态。PL/Tcl以显式子事务的形式为这类问题提供了一个解决方案。

考虑一个在两个账户间实现转账的函数，如下所示。

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
   if [catch {
      spi_exec "UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'"
      spi_exec "UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'"
   } errormsg] {
      set result [format "error transferring funds: %s" $errormsg]
   } else {
      set result "funds transferred successfully"
   }
   spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

如果第二个UPDATE语句导致一个异常，这个函数将记下该失败，但是第一个UPDATE的结果将被提交。换句话说，资金将从Joe的账户中被取走，但不会被转到Mary的账户。这是因为每个spi_exec都是一个单独的子事务，并且那些子事务中只有一个被回滚。

为了处理这类情况，可以把多个数据库操作包裹在一个显式子事务中，它将作为一个整体成功完成或者回滚。PL/Tcl提供了一个subtransaction命令来做这件事情。编写函数如下所示。

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
   if [catch {
      subtransaction {
         spi_exec "UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'"
         spi_exec "UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'"
      }
   } errormsg] {
      set result [format "error transferring funds: %s" $errormsg]
   } else {
      set result "funds transferred successfully"
   }
   spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

注意，为了实现这个目的仍要求使用catch。否则错误将传播到该函数的顶层，导致想要对operations表的插入被阻止。subtransaction命令不会捕捉错误，它仅确保报告错误时在其范围内执行的所有数据库操作将被一起回滚。

一个显式子事务的回滚发生在包含它的Tcl代码报告任何错误时，而不仅仅是数据库访问导致的错误。因此一个subtransaction命令中发生的常规Tcl异常也将导致该子事务被回滚。不过，无错误退出到包含子事务的Tcl代码外面（例如，由于return）不会导致回滚。

# 7.10. 事务管理

在从顶层调用的过程中或者从顶层调用的匿名代码块（DO命令）中，可以控制事务。要提交当前的事务，可调用commit。要回滚当前事务，可调用rollback（注意不能通过spi_exec或类似的函数运行SQL命令COMMIT或者ROLLBACK。这类工作必须用这些函数完成）。在事务结束以后，一个新的事务会自动开始，因此没有独立的函数用来开始新事务。

示例

```
CREATE PROCEDURE transaction_test1()
LANGUAGE pltcl
AS $$
for {set i 0} {$i < 10} {incr i} {
   spi_exec "INSERT INTO test1 (a) VALUES ($i)"
   if {$i % 2 == 0} {
      commit
   } else {
      rollback
   }
}
$$;

CALL transaction_test1();
```

当一个显式的子事务处于活跃状态时，事务不能被结束。

# 7.11. PL/Tcl配置

这一节列举影响PL/Tcl的配置参数。

- pltcl.start_proc (string)

  如果被设置为一个非空字符串，这个参数指定一个无参数PL/Tcl函数的名称（可能是方案限定的），只要为PL/Tcl创建一个新的Tcl解释器，就会执行这个函数。这样一个函数可以执行针对会话的初始化，例如载入额外的Tcl代码。当一个PL/Tcl在一个数据库会话中被第一次执行时，或者由于一个PL/Tcl函数被一个新的SQL角色调用而必须创建一个额外的解释器时，一个新的Tcl解释器会被创建。

  被引用的函数必须用pltcl语言编写，并且不能被标记为SECURITY DEFINER（这些限制确保它运行在它应该要初始化的解释器中）。当前用户也必须有权限调用它。

  如果该函数带着一个错误失败，它将中止导致新解释器创建的函数并且把错误传播到调用查询，进而导致当前事务或子事务被中止。在Tcl中已完成的任何动作将不会被撤销，不过，那个解释器将不会被再次使用。如果该语言被再次使用，则初始化将在一个全新的Tcl解释器中被再次尝试。

  只有超级用户能够更改这个设置。尽管这个设置能在会话中更改，但这种更改将不会影响已经被创建的Tcl解释器。

- pltclu.start_proc (string)

  这个参数与pltcl.start_proc几乎一模一样，只不过它适用于PL/TclU。被引用的函数必须用pltclu语言编写。

# 7.12. Tcl 过程名

在UXsinoDB，同一个函数名可以被用于不同的函数定义，只要它们的参数个数或者类型不同。不过，Tcl 要求所有过程名必须能区分。PL/Tcl 通过让内部 Tcl 过程名称包含该函数在系统表ux_proc中的对象 ID 作为名称的一部分来解决这样的限制。因此，具有相同名称和不同参数类型的UXsinoDB函数也将是不同的 Tcl 过程。这对 PL/Tcl 程序员来说通常不需要关心，但是在调试时可见。

# 第 8 章 PL/Perl - Perl 过程语言

PL/Perl 是一种可载入过程语言，它允许用Perl 编程语言编写UXsinoDB函数。

使用 PL/Perl 的主要优势它允许在存储函数中使用大量 Perl 的“串整理”操作符和函数。使用 Perl 解析复杂串比使用 PL/uxSQL 中提供的串函数和控制结构要更容易。

要在一个特定数据库中安装 PL/Perl，使用CREATE EXTENSION plperl。

> **提示**
>
> 如果把语言安装在template1中，所有后续创建的数据库都将自动地安装有该语言。

> **注意**
>
> 使用源码包安装的用户必须在安装过程中开启对 PL/Perl 的编译。使用二进制包安装的用户可能会在独立的子包中找到 PL/Perl。

## 8.1. PL/Perl 函数和参数

要用 PL/Perl 语言创建一个函数，可使用标准的CREATE FUNCTION语法。

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
  # PL/Perl 函数体
$$ LANGUAGE plperl;
```

函数的主体就是普通的 Perl 代码。事实上，PL/Perl 的粘合代码会把它包裹在一个 Perl 子程序中。一个 PL/Perl 函数会在一种标量上下文中被调用，因此它无法返回列表。如下文所述，可以通过返回引用来返回非标量值（数组、记录和集合）。

在一个PL/Perl过程中，任何从Perl代码返回的值都会被忽略。

PL/Perl 也支持用DO语句调用的匿名代码块，如下所示。

```
DO $$
  # PL/Perl 代码
$$ LANGUAGE plperl;
```

一个匿名代码块没有参数，并且它返回的任何值都会被抛弃。否则其行为就像一个函数。

> **注意**
>
> 在 Perl 中使用命名嵌套子程序是有危险的，特别是当它们在作用域内引用局部变量时。因为 PL/Perl 函数被包装成一个子程序，任何放在其中的命名子程序都会被嵌套。总之，创建通过 coderef 调用的匿名子程序要安全得多。更多信息可见 perldiag手册页 中的Variable "%s" will not stay shared以及 Variable "%s" is not available，或者在互联网上 搜索“perl nested named subroutine”。

CREATE FUNCTION命令的语法要求函数体被写作一个字符串常量。通常对字符串常量使用美元引用最方便。如果选择使用转义字符串语法E''，必须双写任何在函数体中使用的单引号（'）和反斜线（\）。

参数和结果的处理和在任何其他 Perl 子程序中一样：参数被传递到@_中，并且结果值用return返回或者把函数中计算的最后一个表达式作为结果值。

例如，一个返回两个整数值中较大值的函数可以进行如下定义。

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
   if ($_[0] > $_[1]) { return $_[0]; }
   return $_[1];
$$ LANGUAGE plperl;
```

> **注意**
>
> 参数将被从数据库的编码转换到 PL/Perl 中使用的 UTF-8，返回时再从 UTF-8 转回到数据库编码。

如果一个 SQL 空值被传给一个函数，在 Perl 中该参数值将呈现为“undefined”。上述函数定义对于空输入的行为不太好（实际上，它会把它们当作零）。可以为函数定义增加STRICT让UXsinoDB干得更合理：如果空值被传入，函数将根本不会被调用，而只是自动返回一个空结果。另外一种方式，可以在函数体中检查未定义的输入。例如，想让带有一个空参数或者一个非空参数的perl_max返回非空参数而不是空值，如下所示。

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
   my ($x, $y) = @_;
   if (not defined $x) {
      return undef if not defined $y;
      return $y;
   }
   return $x if not defined $y;
   return $x if $x > $y;
   return $y;
$$ LANGUAGE plperl;
```

如上所述，要从一个 PL/Perl 函数返回一个 SQL 空值，就返回一个未定义值。不管函数是严格的还是非严格的都可以这样做。

一个非引用的函数参数中的任何东西都是一个串，是相关数据类型的标准UXsinoDB外部文本表达。在普通数字或文本类型的情况下，Perl 将会做正确的事情并且程序员通常不需要操心。不过，在其他情况下将需要被转换成在 Perl 中更可用的形式。例如，decode_bytea函数可以被用来把类型bytea的参数转换成未转义的二进制形式。

类似地，回传给UXsinoDB的值必须是外部文本表达格式。例如，encode_bytea函数可以被用来转义二进制数据得到类型bytea的返回值。

Perl 可以把UXsinoDB数组返回为对 Perl 数组的引用。如下所示。

```
CREATE OR REPLACE function returns_array()
RETURNS text[][] AS $$
   return [['a''b','c,d'],['e\\f','g']];
```

```
$$ LANGUAGE plperl;

select returns_array();
```

Perl 把UXsinoDB数组作为被 bless 过的UXsinoDB::InServer::ARRAY对象传递。这个对象可以被当作 一个数组引用或者一个串，允许为了向后兼容性与之前版本的UXsinoDB编写的 Perl 代码一起运行。如下所示。

```
CREATE OR REPLACE FUNCTION concat_array_elements(text[]) RETURNS TEXT AS $$
   my $arg = shift;
   my $result = "";
   return undef if (!defined $arg);

   # as an array reference
   for (@$arg) {
      $result .= $_;
   }

   # also works as a string
   $result .= $arg;

   return $result;
$$ LANGUAGE plperl;

SELECT concat_array_elements(ARRAY['PL','/','Perl']);
```

> **注意**
>
> 多维数组被以一种对每一个 Perl 程序员都公认的方法表示为对较低维引用数组的引用。

组合类型参数被作为哈希的引用传递给函数。哈希的键是组合类型的属性名。如下所示。

```
CREATE TABLE employee (
   name text,
   basesalary integer,
   bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
   my ($emp) = @_;
   return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT name, empcomp(employee.*) FROM employee;
```

PL/Perl 函数可以使用相同的方法返回组合类型：返回具有所要求属性的哈希的引用。如下所示。

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
```

```
    return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

任何所要求结果数据类型中不存在于哈希中的列将被返回为空值。

类似的，过程的输出参数也可以被返回为哈希引用，如下所示。

```
CREATE PROCEDURE perl_triple(INOUT a integer, INOUT b integer) AS $$
    my ($a, $b) = @_;
    return {a => $a * 3, b => $b * 3};
$$ LANGUAGE plperl;

CALL perl_triple(5, 10);
```

PL/Perl 函数也能返回标量或者组合类型集合。为了加速启动并且避免在内存中让整个结果集排队等候，通常希望能一次返回一行。可以按下文所说的用return_next来这样做。注意在最后一次return_next后，必须放上return或者return undef（后者更好）。

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF testrowperl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'UXsinoDB' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
$$ LANGUAGE plperl;
```

对于小结果集，可以返回到一个数组的引用，该数组分别包含用于简单类型、数组类型和组合类型的标量、数组引用或者哈希引用。把整个结果集作为数组引用返回，如下所示。

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
    return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
    return [
        { f1 => 1, f2 => 'Hello', f3 => 'World' },
        { f1 => 2, f2 => 'Hello', f3 => 'UXsinoDB' },
        { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
```

```
    ];
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```

如果想要对代码使用strict编译指示，有几种选项可用。对于临时的全局使用，可以SETplperl.use_strict为真。这将影响后续PL/Perl函数的编译，但是对当前会话中已经编译过的函数没有影响。对于持久的全局使用，可以在uxsinodb.conf文件中设置plperl.use_strict为真。

对于在特定函数中的持久使用，可以简单地把**use strict;**放在函数体的顶层。

如果 Perl 版本是 5.10.0 或者更高，也可以使用feature编译指示。

# 8.2. PL/Perl 中的数据值

提供给 PL/Perl 函数代码的参数值是被转换成文本形式的输入参数（就像它们被SELECT语句显示的那样）。反过来，return和return_next命令将接受任何该函数返回类型可接受的输入格式的串。

# 8.3. 内建函数

## 8.3.1. 从 PL/Perl 访问数据库

可以通过下列函数从 Perl 函数中访问数据库本身。

- spi_exec_query(*query* [, *max-rows*])

  spi_exec_query执行一个 SQL 命令并且以哈希引用数组的引用的形式返回整个行集。只有在知道结果集相对较小时才应该使用这个命令。带有可选最大行数的查询（SELECT命令），如下所示。

  ```
  $rv = spi_exec_query('SELECT * FROM my_table', 5);
  ```

  这会从表my_table. 返回最多 5 行。如果my_table有一个列my_column，可以从结果的$i行得到值，如下所示。

  ```
  $foo = $rv->{rows}[$i]->{my_column};
  ```

  可以这样访问从一个SELECT查询返回的总行数，如下所示。

  ```
  $nrows = $rv->{processed}
  ```

  这里是使用不同命令类型的一个示例。如下所示。

  ```
  $query = "INSERT INTO my_table VALUES (1, 'test')";
  $rv = spi_exec_query($query);
  ```

  可以这样访问命令状态（例如SPI_OK_INSERT），如下所示。

```
$res = $rv->{status};
```

得到受影响的行数，如下所示。

```
$nrows = $rv->{processed};
```

示例

```
CREATE TABLE test (
    i int,
    v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};
    foreach my $rn (0 .. $nrows - 1) {
        my $row = $rv->{rows}[$rn];
        $row->{i} += 200 if defined($row->{i});
        $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
        return_next($row);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

- spi_query(*command*)  spi_fetchrow(*cursor*)  spi_cursor_close(*cursor*)

  spi_query和spi_fetchrow结对用于可能比较大的行集合，或者用于希望在行到达时返回的情况。spi_fetchrow只和spi_query一起工作。怎样使用它们，如下所示。

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
    use Digest::MD5 qw(md5_hex);
    my $file = '/usr/share/dict/words';
    my $t = localtime;
    elog(NOTICE, "opening file $file at $t" );
    open my $fh, '<', $file # ooh, it's a file access!
        or elog(ERROR, "cannot open $file for reading: $!");
    my @words = <$fh>;
    close $fh;
    $t = localtime;
    elog(NOTICE, "closed file $file at $t");
    chomp(@words);
```

```
    my $row;
    my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
    while (defined ($row = spi_fetchrow($sth))) {
        return_next({
            the_num => $row->{a},
            the_text => md5_hex($words[rand @words])
        });
    }
    return;
$$ LANGUAGE plperlu;

SELECT * from lotsa_md5(500);
```

通常，spi_fetchrow应该重复执行直到它返回undef（表示没有更多行要读取）。当spi_fetchrow返回undef时，spi_query返回的游标会自动被释放。如果不想读取所有的行，可以调用spi_cursor_close来释放游标。如果没有这样做会导致内存泄露。

- spi_prepare(*command*, *argument types*) spi_query_prepared(*plan*, *arguments*) spi_exec_prepared(*plan* [, *attributes*], *arguments*) spi_freeplan(*plan*)

  spi_prepare、spi_query_prepared、spi_exec_prepared和spi_freeplan为预备查询实现了相同的功能。spi_prepare接受一个查询字符串，其中包括编好号的参数占位符（$1、$2 等）以及参数类型的字符串列表，如下所示。

```
$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name = $2',
                    'INTEGER', 'TEXT');
```

  一旦通过调用spi_prepare准备好一个查询计划，就可以在spi_exec_prepared（返回的结果和spi_exec_query相同）或者spi_query_prepared（返回的结果和spi_query一样，后面会被传给spi_fetchrow）中用该计划来取代字符串查询。 spi_exec_prepared可选的第二个参数是属性的哈希引用，当前唯一支持的属性是limit，它限定了一个查询返回的最大行数。

  预备查询的有点是可以把一个准备好的计划用于多次查询执行。不再需要该计划后，可以用spi_freeplan释放它，如下所示。

```
CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
    $_SHARED{my_plan} = spi_prepare('SELECT (now() + $1)::date AS now',
                        'INTERVAL');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
    return spi_exec_prepared(
        $_SHARED{my_plan},
        $_[0]
    )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
    spi_freeplan( $_SHARED{my_plan});
    undef $_SHARED{my_plan};
$$ LANGUAGE plperl;
```

```
SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();

 add_time  |  add_time  |  add_time
------------+------------+------------
 2005-12-10 | 2005-12-11 | 2005-12-12
```

注意spi_prepare中的参数下标通过 $1、$2、$3 等定义，这样避免了用双引号来声明查询串（容易导致难以捕捉的缺陷）。

另一个展示spi_exec_prepared中可选参数用法的示例，如下所示。

```
CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS address
                FROM generate_series(1,3) AS id;

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
      $_SHARED{plan} = spi_prepare('SELECT * FROM hosts
                            WHERE address << $1', 'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts AS $$
      return spi_exec_prepared(
            $_SHARED{plan},
            {limit => 2},
            $_[0]
      )->{rows};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS $$
      spi_freeplan($_SHARED{plan});
      undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();

   query_hosts
-----------------
 (1,192.168.1.1)
 (2,192.168.1.2)
(2 rows)
```

- spi_commit()  spi_rollback()

  提交或者回滚当前事务。只能在从顶层调用的过程或者匿名代码块（DO命令）中调用这个函数（注意不能通过spi_exec_query或者类似的函数运行SQL命令COMMIT或者ROLLBACK。这样的工作只能使用这些函数完成）。在一个事务结束后，一个新的事务会自动开始，因此没有单独的函数来开始新事务。

  示例

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plperl
AS $$
foreach my $i (0..9) {
    spi_exec_query("INSERT INTO test1 (a) VALUES ($i)");
    if ($i % 2 == 0) {
        spi_commit();
    } else {
        spi_rollback();
    }
}
$$;

CALL transaction_test1();
```

## 8.3.2. PL/Perl 中的工具函数

- elog(*level*, *msg*)

  发出一个日志或者错误消息。可用的级别有DEBUG、LOG、INFO、NOTICE、WARNING以及ERROR。ERROR产生一种错误情况，如果它没有被周围的 Perl 代码捕获，错误会传播到调用查询中，导致当前事务或者子事务被中止。这实际上和 Perl 的die 命令相同。其他级别只产生不同优先级的消息。特定优先级的消息是被报告给客户端、写到服务器日志或者两者都做由配置变量log_min_messages和 client_min_messages控制。

- quote_literal(*string*)

  返回给定字符串的被适当引用后的形式，这种形式能被用作 SQL 语句字符串中的字符串。嵌入的引号和反斜线会被正确地双写。注意对 undef 输入quote_literal会返回 undef。如果参数可能是 undef，quote_nullable通常更合适。

- quote_nullable(*string*)

  返回给定字符串的被适当引用后的形式，这种形式能被用作 SQL 语句字符串中的字符串。或者在参数为 undef 时，返回未引用的串 "NULL"。嵌入的引号和反斜线会被正确地双写。

- quote_ident(*string*)

  返回给定字符串的被适当引用后的形式，这种形式能被用作 SQL 语句字符串中的标识符。只有在必要时才增加引号（即，如果串包含非标识符字符或者是大小写折叠的）。嵌入的引号会被正确地双写。

- decode_bytea(*string*)

  返回由给定串的内容（应该用bytea编码）表示的未转义二进制数据。

- encode_bytea(*string*)

  返回给定串的二进制数据内容的bytea编码形式。

- encode_array_literal(*array*) encode_array_literal(*array*, *delimiter*)

  把被引用的数组的内容返回成数组文字格式。如果它不是一个数组的引用，则不加修改地返回参数值。如果没有指定定界符或者定界符为 undef，则默认把",用作数组文字的元素之间的定界符。

- encode_typed_literal(*value*, *typename*)

  把一个 Perl 变量转换为由第二个参数传入的数据类型的值，并且返回该值的字符串表达。它能正确地处理嵌套数组和组合类型的值。

- encode_array_constructor(*array*)

  把被引用数组的内容返回为数组构造器格式的一个串。其中的个体值用quote_nullable引用。如果参数不是一个数组引用，则返回用quote_nullable引用的该参数值。

- looks_like_number(*string*)

  如果给定串的内容对于 Perl 看起来像是数字则返回真，否则返回假。如果参数是 undef 则返回 undef。前导和结尾的空格会被忽略。Inf和Infinity被视作数字。

- is_array_ref(*argument*)

  如果给定参数可以被当作一个数组引用对待则返回真值，即该参数的定义为ARRAY或者UXsinoDB::InServer::ARRAY时返回真。否则返回假。

# 8.4. PL/Perl 中的全局值

可以在函数调用之间或者当前会话的生命期中用全局哈希%_SHARED来存储数据，包括代码引用。

这是共享数据的示例，如下所示。

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
   if ($_SHARED{$_[0]} = $_[1]) {
      return 'ok';
   } else {
      return "cannot set shared variable $_[0] to $_[1]";
   }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
   return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl!  How''s tricks?');
SELECT get_var('sample');
```

这是一个使用代码引用的稍微复杂一点的示例，如下所示。

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
   $_SHARED{myquote} = sub {
      my $arg = shift;
      $arg =~ s/(['\\])/\\$1/g;
      return "'$arg'";
   };
$$ LANGUAGE plperl;
```

```
SELECT myfuncs(); /* 初始化函数 */

/* 设置一个使用引用函数的函数 */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
   my $text_to_quote = shift;
   my $qfunc = $_SHARED{myquote};
   return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

（可以把上面的代码用一行 return $_SHARED{myquote}->($_[0]);替换，但是代码可读性差）。

处于安全原因，PL/Perl 一个 SQL 角色独立的 Perl 解释器中执行该角色调用的任何一个函数。这可以避免一个用户无意或者恶意地干涉另一个用户的PL/Perl 函数的行为。每一个这样的解释器都具有其自身的%_SHARED变量值和其他全局状态。因此，只有当两个 PL/Perl 函数是由同一个 SQL 角色执行时，它们才能共享同一个 %_SHARED值。在使用单个会话执行多个 SQL 角色的代码（通过SECURITY DEFINER函数、使用SET ROLE等）的应用中，需要采取显式的步骤以保证 PL/Perl函数能够通过%_SHARED共享数据。要这样做，需要确保要通信的函数都属于同一个用户，并且把它们标记为 SECURITY DEFINER。当然，要小心这样的函数被滥用。

# 8.5. 可信的和不可信的 PL/Perl

通常，PL/Perl 被作为一种“可信的”编程语言安装，其名称为plperl。在这种设置下，为了保持安全性禁用了某些 Perl 操作。一般来说，被限制的操作是那些与环境交互的操作。它们包括文件处理操作、require以及use（外部模块）。没有办法像 C 函数那样访问数据库服务器进程的内部或者用服务器进程的权限得到 OS 级别的访问。因此，任何没有特权的数据库用户也被允许使用这种语言。

下面示例中的函数将无法工作，因为出于安全原因不允许它做文件操作。这个函数的创建会失败，因为验证器会捕捉到它使用了禁用的操作。

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
   my $tmpfile = "/tmp/badfile";
   open my $fh, '>', $tmpfile
      or elog(ERROR, qq{could not open the file "$tmpfile": $!});
   print $fh "Testing writing to a file\n";
   close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
   return 1;
$$ LANGUAGE plperl;
```

有些时候需要编写不受限制的 Perl 函数。例如，想要一个能发送电子邮件的 Perl 函数。要处理这些情况，可以把 PL/Perl 安装成一种“不可信的”语言（通常被称作PL/PerlU）。在这种情况下整个 Perl 语言的特性都可以使用。在安装语言时，用语言名称plperlu将会选择不可信的 PL/Perl 变体。

PL/PerlU函数的编写者必须注意该函数不能被用来做其设计目的之外的事情，因为该函数能做一个作为数据库管理员登录的用户可以做的任何事情。注意数据库系统只允许数据库超级用户用不可信语言创建函数。

如果上述函数是一个超级用户用语言plperlu创建的，则可以执行成功。

以和plperl语言同样的方式，可以用plperlu编写 Perl 中的匿名代码块，这样的代码块能够使用受限的操作，不过调用者必须是超级用户。

> **注意**
>
> 虽然对每个 SQL 角色会在一个独立的 Perl 解释器中运行PL/Perl函数，但是在一个给定会话中执行的所有PL/PerlU函数都运行在一个 Perl 解释器中（与用于任何PL/Perl函数的解释器不同）。这允许PL/PerlU函数自由地共享数据，但是PL/Perl和PL/PerlU函数之间不会发生任何交流。

> **注意**
>
> Perl 不支持一个进程中的多个解释器，除非编译它时使用了合适的标志，即usemultiplicity或者useithreads（ usemultiplicity会更好，除非确实需要使用线程。更多细节，请见perlembed手册页）。 如果PL/Perl用的是一份没有这样编译的 Perl 拷贝，那么在每个会话中只能有一个 Perl 解释器，并且因此任一会话只能要么执行PL/PerlU函数，要么执行同一个 SQL 角色调用的PL/Perl函数。

# 8.6. PL/Perl 触发器

PL/Perl 可以被用来编写触发器函数。在触发器函数中，哈希引用$_TD包含有关当前触发器事件的信息。$_TD是一个全局变量，对触发器的每一次调用它都会得到一个独立的本地值。$_TD哈希引用的域包括如下内容。

- $_TD->{new}{foo}

  列foo的NEW值

- $_TD->{old}{foo}

  列foo的OLD值

- $_TD->{name}

  要被调用的触发器的名称

- $_TD->{event}

  触发器事件：INSERT、UPDATE、DELETE、TRUNCATE或者UNKNOWN

- $_TD->{when}

  什么时候调用触发器：BEFORE、AFTER、INSTEAD OF或者UNKNOWN

- $_TD->{level}

  触发器级别：ROW、STATEMENT或者UNKNOWN

- $_TD->{relid}

  触发器定义在其上的表的 OID

- $_TD->{table_name}

触发器定义在其上的表的名称

- $_TD->{relname}

  触发器定义在其上的表的名称。这已经被废弃，并且可能会在未来的发布中被移除。请使用 $_TD->{table_name}。

- $_TD->{table_schema}

  触发器定义在其上的表所在的模式的名称

- $_TD->{argc}

  触发器函数的参数数目

- @{$_TD->{args}}

  触发器函数的参数。如果$_TD->{argc}为 0 则不存在

行级触发器可以返回下列之一：

- return;

  执行操作

- "SKIP"

  不执行操作

- "MODIFY"

  指示触发器函数修改了NEW行

这个事件触发器函数的示例，展示了上文所说的一些东西，如下所示。

```
CREATE TABLE test (
   i int,
   v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
   if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
      return "SKIP";   # skip INSERT/UPDATE command
   } elsif ($_TD->{new}{v} ne "immortal") {
      $_TD->{new}{v} .= "(modified by trigger)";
      return "MODIFY"; # 修改行并且执行 INSERT/UPDATE 命令
   } else {
      return;          # 执行 INSERT/UPDATE 命令
   }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
   BEFORE INSERT OR UPDATE ON test
   FOR EACH ROW EXECUTE FUNCTION valid_id();
```

# 8.7. PL/Perl 事件触发器

PL/Perl 可以被用来编写事件触发器函数。在事件触发器函数中，哈希引用$_TD包含有关当前触发器事件的信息。$_TD是一个全局变量，对触发器的每一次调用它都会得到一个独立的本地值。$_TD哈希引用的域，包括如下内容。

- $_TD->{event}

  触发器为其触发的事件名称。

- $_TD->{tag}

  触发器为其触发的命令标签。

触发器函数的返回值会被忽略。

这个事件触发器函数的示例，展示了上文所说的一些内容，如下所示。

```
CREATE OR REPLACE FUNCTION perlsnitch() RETURNS event_trigger AS $$
  elog(NOTICE, "perlsnitch: " . $_TD->{event} . " " . $_TD->{tag} . " ");
$$ LANGUAGE plperl;

CREATE EVENT TRIGGER perl_a_snitch
   ON ddl_command_start
   EXECUTE FUNCTION perlsnitch();
```

# 8.8. PL/Perl 下面的东西

## 8.8.1. 配置

这一节列出了影响PL/Perl的配置参数。

- plperl.on_init (string)

  指定当第一次初始化一个 Perl 解释器时要执行的 Perl 代码，这会在具体用于plperl或plperlu之前做完。当这段代码被执行时 SPI 函数不可用。如果该代码由于错误失败，它将中止解释器的初始化并且把错误传播到调用查询，最终导致当前事务或者子事务被中止。

  该 Perl 代码被限制为一个单一的字符串。更长的代码可以放在一个模块中，然后由on_init字符串载入。如下所示。

  ```
  plperl.on_init = 'require "plperlinit.pl"'
  plperl.on_init = 'use lib "/my/app"; use MyApp::UxInit;'
  ```

  任何被plperl.on_init载入的模块（不管是直接还是间接）都可以被plperl使用。这可能会导致安全性风险。要看哪些模块已经被载入，如下所示。

  ```
  DO 'elog(WARNING, join ", ", sort keys %INC)' LANGUAGE plperl;
  ```

如果 plperl 库被包括在shared_preload_libraries中，那么初始化将发生在 uxmaster 中，在这种情况下要特别地考虑对uxmaster 带来的不稳定风险。使用这种特性的主要原因是，plperl.on_init载入的 Perl 模块只需要在 uxmaster 开始时被载入，并且在数据库会话中不需要任何工作就立刻可用。不过，要记住这只免除了一个数据库会话中使用的第一个 Perl 解释器的负载 — 不管是 PL/PerlU 还是用于第一个 SQL 角色调用 PL/Perl 函数的 PL/Perl。在一个数据库会话中创建的任何额外的 Perl 解释器将不得不重新执行plperl.on_init。还有，在 Windows 上无论从什么里面进行预先载入，都不会有这种节约，因为在 uxmaster 进程中创建的 Perl 解释器不会传播到子进程中。

这个参数只能在uxsinodb.conf文件或者服务器命令中设置。

- plperl.on_plperl_init (string)

  plperl.on_plperlu_init (string)

  这些参数分别指定当为plperl或plperlu专门准备好一个 Perl 解释器时要执行的 Perl 代码。当一个 PL/Perl 或者 PL/PerlU 函数 第一次在一个数据库会话中被执行时会发生这种动作，或者由于调用其他语言或者新的 SQL 角色调用 PL/Perl 函数导致创建额外的解释器时也会发生这种动作。这些初始化跟随着plperl.on_init所作的初始化。当这段代码被执行时，SPI 函数不可用。plperl.on_plperl_init中的 Perl代码在“锁闭”解释器之后被执行，因此它只能执行可信的操作。

  如果该代码由于错误失败，它将中止初始化并且把错误传播到调用查询，最终导致当前事务或者子事务被中止。在 Perl 中已经完成的任何动作都不会被撤销。不过，该解释器将不能被再次使用。如果再次使用该语言，将在一个新鲜的 Perl 解释器中再次尝试初始化。

  只有超级用户能够更改这些设置。尽管这些设置可以在会话中被修改，但是这类更改将不会影响已经被用来执行函数的 Perl 解释器。

- plperl.use_strict (boolean)

  如果被设置为真，则后续的 PL/Perl 函数编译将会启用strict编译指示。这个参数不影响当前会话中已编译的函数。

## 8.8.2. 限制和缺失的特性

PL/Perl 中目前缺少下列特性，但是欢迎大家对此作出贡献。

- PL/Perl 函数不能直接调用彼此。
- SPI 还没有被完全实现。
- 如果正在使用spi_exec_query取一个非常大的数据集，应该注意它们都会进入到内存中。可以按先前所述，通过使用spi_query/spi_fetchrow来避免发生这类情况。

  如果一个集合返回函数通过return把一个大型的行集合返回给 UXsinoDB，同样会发生这种情况。同样如前所述，可以为每一个要返回的行使用return_next来避免这种问题。

- 当会话正常结束（而不是由于致命错误结束）时，任何已经定义的END块将被执行。当前不会执行其他动作。特别地，此时文件句柄不会被自动刷写并且对象不会被自动销毁。

# 第 9 章 PL/Python - Python 过程语言

PL/Python过程语言允许用Python 语言编写UXsinoDB函数。

要在一个特定数据库中安装 PL/Python，请使用CREATE EXTENSION plpythonu（参见第 9.1 节“Python 2 vs. Python 3”）。

> **提示**
>
> 如果把一种语言安装在template1中，所有后续创建的数据库都会自动安装该语言。

PL/Python 只是一种“不可信”语言，这意味着它没有提供任何方法来限制用户在其中的所作所为，并且因此被命名为plpythonu。如果未来开发出在 Python 中的安全执行机制，可能会出现一种可信的变体plpython。不可信 PL/Python 中函数的编写者必须注意该函数不能被用来做任何它不应该做的事情，因为它将能做以数据库管理员用户登录能做的事情。只有超级用户能够创建plpythonu等不可信语言中的函数。

> **注意**
>
> 源码包的用户必须在安装过程中特别地启用 PL/Python 的编译（更多信息请参见安装指导）。二进制包的用户可以在一个单独的子包中找到 PL/Python。

## 9.1. Python 2 vs. Python 3

PL/Python 同时支持 Python 2 和 Python 3 两种语言变体（UXsinoDB 安装指导可能包含了所支持的 Python 次版本的更精确的信息）。因为 Python 2 和 Python 3 语言变体在某些重要的方面并不兼容，PL/Python 使用了下列命名和转换模式以避免混淆它们。

- 名为plpython2u的 UXsinoDB 语言实现了基于 Python 2 语言变体的 PL/Python。
- 名为plpython3u的 UXsinoDB 语言实现了基于 Python 3 语言变体的 PL/Python。
- 名为plpythonu的语言实现了基于默认 Python语言变体（当前是 Python 2）的 PL/Python（这种默认与任何本地 Python 安装所认为的“默认”无关，例如/usr/bin/python）。在遥远的未来，UXsinoDB 的发布中可能会把这种默认语言变体改成 Python 3，这取决于 Python 社区迁移到 Python 3 的进度。

这种模式类似于PEP 394中关于python命令的命名和转换的推荐。

到底是 Python 2 还是 Python 3 的 PL/Python 可用或是两者都可用，取决于编译配置或者被安装的包。

> **提示**
>
> 如果是编译安装，则取决于在安装期间找到的 Python 版本或者用PYTHON环境变量显式设置的版本。要在一个安装中让两种变体的 PL/Python 都可用，源代码树必须被配置和编译两次。

这产生了下列的使用和迁移策略。

- 现有用户以及对 Python 3 不感兴趣的用户使用plpythonu语言并且在可预见的未来不必做出任何改变。推荐通过迁移到 Python 2.6/2.7 逐步地让代码“经得起未来的考验”以简化最终迁移到 Python 3 的工作。

  实际上，很多 PL/Python 函数可以用很少或者不做修改就迁移到 Python 3。

- 对于代码严重依赖于 Python 2 并且不打算做改变的用户可以使用plpython2u语言。这将在很长时间内都有效，直到 UXsinoDB 完全删除掉对 Python 2 的支持。

- 想投入 Python 3 的怀抱的用户可以使用plpython3u语言，在当前的标准下这将一直有效。在遥远的未来，当 Python 3 成为默认以后，出于审美的原因，“3”可能会被移除。

- 想要构建一个只有 Python 3 的操作系统环境的冒险者们，可以更改ux_pltemplate的内容让plpythonu等价于plpython3u，记住这将会让他们的安装与世界的其他大部分东西都不兼容。

有关移植到 Python 3 的更多信息还可见文档What's New In Python 3.0。

不允许在同一个会话中使用基于 Python 2 的 PL/Python 以及基于 Python 3 的 PL/Python，因为动态模块中的符号会冲突，这会导致 UXsinoDB 服务器进程的崩溃。在一个会话中有一个检查来阻止混淆 Python 的主版本，如果检测到不匹配会中断会话。不过，可以在同一个数据库中对不同的会话使用两种 PL/Python 变体。

# 9.2. PL/Python 函数

PL/Python 中的函数通过标准的CREATE FUNCTION语法声明，如下所示。

```
CREATE FUNCTION funcname (argument-list)
  RETURNS return-type
AS $$
  # PL/Python 函数体
$$ LANGUAGE plpythonu;
```

函数体就是一个 Python 脚本。当函数被调用时，它的参数被当做列表args的元素传递，命名参数也被作为普通变量传递给 Python 脚本。使用命名参数通常可读性更好。Python 代码会以通常的方式返回结果，即使用return或者yield（在结果集合语句的情况中）。如果没有提供一个返回值，Python 会返回默认的None。PL/Python会把 Python 的None翻译成 SQL 空值。在一个过程中，Python代码的结果必须是None（通常实现为结束过程时不写return语句或者使用不带参数的return），否则将会发生错误。

例如，一个返回两个整数中较大的整数的函数可以定义成如下内容。

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

作为该函数定义给出的 Python 代码会被转换成一个 Python 函数。例如上面的代码会得到如下结果。

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

假定 23456 是UXsinoDB分配给这个函数的 OID。

参数被设置为全局变量。由于 Python 的可见范围规则，这会导致一种后果：在函数内不能把一个参数变量重新赋予给一个涉及该变量名称本身的表达式的值，除非在该代码块中重新声明该变量为全局的。例如，如下代码无法使用。

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip() # 错误
  return x
$$ LANGUAGE plpythonu;
```

因为对x的赋值让x成为了整个代码块的一个局部变量，并且因此该赋值操作右边的x引用的是一个还未赋值的局部变量x，而不是 PL/Python 函数的参数。通过使用global语句，可以让上面的代码正常使用。

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip() # 现在好了
  return x
$$ LANGUAGE plpythonu;
```

但是不建议依赖于这类 PL/Python 的实现细节。最好把函数参数当作是只读。

# 9.3. 数据值

一般来讲，PL/Python 的目标是提供在 UXsinoDB 和 Python 世界之间的一种“自然的”映射。这包括下面介绍的数据映射规则。

## 9.3.1. 数据类型映射

在调用一个 PL/Python 函数时，它的参数会被从 UXsinoDB 的数据类型转换成相应的 Python 类型，如下所示。

- UXsinoDB 的boolean被转换成 Python 的bool。
- UXsinoDB 的smallint和int被转换成 Python 的int。 UXsinoDB 的bigint和oid被转换成 Python 2 的long或者 Python 3 的int。
- UXsinoDB 的real和double被转换成 Python 的float。
- UXsinoDB 的numeric被转换成 Python 的Decimal。如果存在cdecimal包，则会从其中导入该类型。否则将使用来自标准库的decimal.Decimal。cdecimal比decimal要更快。不过，在 Python 3.3 以及更高的版本中，cdecimal已经被整合到了标准库中（也是用decimal这个名字），因此也就不再有什么区别。

- UXsinoDB 的bytea被转换成 Python 的str（Python 2）和bytes（Python 3）。在 Python 2 中，串应该被当做没有任何字符编码的字节序列对待。
- 包括 UXsinoDB 字符串类型在内的所有其他数据类型会被转换成一个 Python 的str。在 Python 2 中，这个串将用 UXsinoDB 服务器编码；在 Python 3 中，它将和所有串一样使用 Unicode。
- 对于非标量数据类型，请见下文。

当一个 PL/Python 函数返回时，会按照下列规则把它的返回值转换成该函数声明的 UXsinoDB 返回数据类型。

- 当 UXsinoDB 返回类型是boolean时，返回值会被根据Python规则计算真假。也就是说，0 和空串是假，但是要特别注意'f'是真。
- 当 UXsinoDB 返回类型是bytea时，返回值会被使用相应的 Python 内建机制转换成串（Python 2）或者字节（Python 3），结果将被转换成bytea。
- 对于所有其他 UXsinoDB 返回类型，返回值被使用 Python 内建的str转换成一个串，并且结果会被传递给 UXsinoDB 数据类型的输入函数（如果该 Python 值是一个float，它会被用内建的repr而不是str转换，这是为了避免精度损失）。

  当 Python 2 的串被传递给 UXsinoDB 时，它们被要求是 UXsinoDB 服务器编码。在当前服务器编码中不可用的串将会产生错误，但是并非所有的编码失配都能被检测到，因此当没有正确地将串编码时，垃圾数据仍然会产生。Unicode 串会被自动地转换为正确的编码，因此使用 Unicode 串更加安全并且更加方便。在 Python 3 中，所有串都是 Unicode 串。
- 对于非标量数据类型，请见下文。

注意所声明的 UXsinoDB 返回类型和实际返回对象的 Python 数据类型之间的逻辑失配不会被标志，无论怎样该值都会被转换。

## 9.3.2. Null, None

如果一个 SQL 空值被传递给一个函数，该参数值将作为 Python 中的None出现。例如，第 9.2 节 “PL/Python 函数”中展示的pymax的函数定义对于空值输入将会返回错误的回答。为函数定义增加STRICT让UXsinoDB做得更加合理：如果一个空值被传入，该函数将根本不会被调用，而只是自动地返回一个空结果。此外，可以在函数体中检查空输入，如下所示。

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

如前所示，要从一个 PL/Python 函数返回一个 SQL 空值，可返回值None。不管该函数严格与否都可以这样做。

## 9.3.3. 数组、列表

SQL 数组会被作为一个 Python 列表传递到 PL/Python 中。要从一个 PL/Python 函数中返回出一个 SQL 数组值，可返回一个Python列表，如下所示。

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return [1, 2, 3, 4, 5]
$$ LANGUAGE plpythonu;

SELECT return_arr();
 return_arr
-------------
 {1,2,3,4,5}
(1 row)
```

多维数组被当做嵌套的Python列表传入PL/Python。例如，一个2维数组是一个列表的列表。在把一个多维SQL数组从PL/Python函数返回出去时，每一层的内层列表都必须是相同的尺寸。如下所示。

```
CREATE FUNCTION test_type_conversion_array_int4(x int4[]) RETURNS int4[] AS $$
plpy.info(x, type(x))
return x
$$ LANGUAGE plpythonu;

SELECT * FROM test_type_conversion_array_int4(ARRAY[[1,2,3],[4,5,6]]);
INFO:  ([[1, 2, 3], [4, 5, 6]], <type 'list'>)
 test_type_conversion_array_int4
---------------------------------
 {{1,2,3},{4,5,6}}
(1 row)
```

UXsinoDB当不支持多维数组时，也接受元组之类的其他Python序列。不过，它们总是被当做一维数组，因为它们会和组合类型混淆。出于同样的原因，当一个组合类型被用在多维数组中时，它必须被表示为一个元组而不是一个列表。

注意在 Python 中，串是序列，这可能产生与 Python 程序员所熟悉的不同的效果，如下所示。

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpythonu;

SELECT return_str_arr();
 return_str_arr
----------------
 {h,e,l,l,o}
(1 row)
```

## 9.3.4. 组合类型

组合类型参数被作为 Python 映射传递给函数。映射的元素名称就是组合类型的属性名。如果被传递的行中有一个属性是空值，在映射中它的值是None。如下所示。

```
CREATE TABLE employee (
  name text,
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpythonu;
```

有多种方式从一个 Python 函数返回行或者组合类型，如下所示。

```
CREATE TYPE named_value AS (
  name   text,
  value  integer
);
```

一个组合结果可以被返回为多种类型，如下所示。

序列类型（一个元组或者列表，但不是集合，因为 集合不能被索引）

被返回的序列对象必须具有和组合结果类型的域个数相同的项。索引号为 0 的项被分配给组合类型的第一个域，为 1 的项给第二个域，以此类推。如下所示。

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return ( name, value )
  # or alternatively, as tuple: return [ name, value ]
$$ LANGUAGE plpythonu;
```

要为任意列返回一个 SQL 空，应在对应的位置插入None。

当一个组合类型的数组被返回时，它不能被返回为列表，因为会弄不清该Python列表究竟是表示一个组合类型还是另一个数组维度。

映射（字典）

用列名作为键从映射中检索每一个结果类型列的值。如下所示。

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return { "name": name, "value": value }
$$ LANGUAGE plpythonu;
```

任何额外的字典键/值对都会被忽略。丢失的键会被当做错误。要为任意列返回一个 SQL 空，应用相应的列名作为键插入None。

对象（任何提供方法__getattr__的对象）

这和映射的运作方式相同。如下所示。

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  class named_value:
    def __init__ (self, n, v):
      self.name = n
      self.value = v
  return named_value(name, value)

  # 或简单地
  class nv: pass
  nv.name = name
  nv.value = value
  return nv
$$ LANGUAGE plpythonu;
```

也支持具有OUT参数的函数。如下所示。

```
CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
return (1, 2)
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple();
```

过程的输出参数会以同样的方式传回。如下所示。

```
CREATE PROCEDURE python_triple(INOUT a integer, INOUT b integer) AS $$
return (a * 3, b * 3)
$$ LANGUAGE plpythonu;

CALL python_triple(5, 10);
```

## 9.3.5. 集合返回函数

PL/Python函数也能返回标量类型或者组合类型的集合。有多种方法可以做到这一点，因为被返回的对象在内部会被转变成一个迭代器。假设有组合类型，如下所示。

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

可从以下类型返回集合结果：

序列类型（元组、列表、集合）

```
CREATE FUNCTION greet (how text)
```

```
 RETURNS SETOF greeting
AS $$
 # 把包含列表的元组返回为组合类型
 # 所有其他组合也能行
 return ( [ how, "World" ], [ how, "UXsinoDB" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

迭代器（任何提供__iter__以及 next方法的对象）

```
CREATE FUNCTION greet (how text)
 RETURNS SETOF greeting
AS $$
 class producer:
  def __init__ (self, how, who):
   self.how = how
   self.who = who
   self.ndx = -1

  def __iter__ (self):
   return self

  def next (self):
   self.ndx += 1
   if self.ndx == len(self.who):
    raise StopIteration
   return ( self.how, self.who[self.ndx] )

 return producer(how, [ "World", "UXsinoDB", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

发生器（yield）

```
CREATE FUNCTION greet (how text)
 RETURNS SETOF greeting
AS $$
 for who in [ "World", "UXsinoDB", "PL/Python" ]:
  yield ( how, who )
$$ LANGUAGE plpythonu;
```

也支持有OUT参数的集合返回函数（使用RETURNS SETOF record）。如下所示。

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT integer) RETURNS SETOF
 record AS $$
return [(1, 2)] * n
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple_setof(3);
```

# 9.4. 共享数据

在对同一个函数的重复调用之间可用全局字典SD来存储私有数据。全局字典GD是公共数据，它对一个会话中的所有 Python 函数都可用，使用起来需要注意。

在 Python 解释器中每一个函数都会得到自己的执行环境，因此来自myfunc的全局数据和函数参数对myfunc2不可用。例外是GD字典中的数据。

# 9.5. 匿名代码块

PL/Python 也支持用DO语句调用的匿名代码块，如下所示。

```
DO $$
  # PL/Python 代码
$$ LANGUAGE plpythonu;
```

匿名代码块没有参数，并且任何可能返回的值都会被丢弃。否则，其行为就像一个函数。

# 9.6. 触发器函数

当函数被用作触发器时，字典TD包含触发器相关的值，如下所示。

TD["event"]

包含字符串型的事件：INSERT、UPDATE、DELETE或者TRUNCATE。

TD["when"]

包含BEFORE、AFTER或者INSTEAD OF之一。

TD["level"]

包含ROW或者STATEMENT。

TD["new"]
TD["old"]

对于行级触发器，这些域的一个或者两个包含相应的触发器行，这取决于触发器事件是什么。

TD["name"]

包含触发器的名称。

TD["table_name"]

包含该触发器发生其上的表名。

TD["table_schema"]

包含该触发器发生其上的表所属的模式名。

TD["relid"]

包含该触发器发生其上的表的 OID。

TD["args"]

如果CREATE TRIGGER命令包括参数，它们可以通过TD["args"][0]至TD["args"][*n*-1]使用。

如果TD["when"]是BEFORE或者INSTEAD OF并且TD["level"]是ROW，可以从 Python 函数返回None或者"OK"来表示行没有被修改。返回"SKIP"可以中止事件，或者在TD["event"]为INSERT或UPDATE时可以返回"MODIFY"以表示已经修改了新行。否则返回值会被忽略。

# 9.7. 数据库访问

PL/Python 语言模块会自动导入一个被称为plpy的 Python 模块。这个模块中的函数和常量在 Python 代码中可以用plpy.*foo*这样的方式访问。

## 9.7.1. 数据库访问函数

plpy模块提供了几个函数来执行数据库命令，如下所示。

plpy.execute(*query* [, *max-rows*])

用一个查询字符串和一个可选的行限制参数调用plpy.execute会让该查询运行并且其结果会被以一个结果对象返回。

结果对象模拟一个列表或者字典对象。可以用行号和列名来访问结果对象。如下所示。

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

会从my_table中返回 5 行。如果my_table有一列是my_column，可以这样来访问它：

```
foo = rv[i]["my_column"]
```

可以用内建的len函数获得返回的行数。

结果对象有这些额外的方法，如下所示。

nrows()

返回被该命令处理的行数。注意这不一定与返回的行数相同。例如，UPDATE命令将会设置这个值但是不返回任何行（除非使用RETURNING）。

status()

SPI_execute()的返回值。

colnames()
coltypes()
coltypmods()

分别返回一个列名列表、列类型 OID 列表以及列的类型相关的类型修饰符列表。

在来自于不产生结果集合的命令的结果对象上调用这些方法会产生异常，例如不带RETURNING的UPDATE或者DROP TABLE。但是在包含的行数为零的结果集合上使用这些方法是 OK 的。

__str__()

也定义了标准的__str__方法，例如可以使用plpy.debug(rv)来调试查询执行结果。

结果对象可以被修改。

注意调用plpy.execute将会导致整个结果集合被读入到内存中。只有当确信结果集相对比较小时才应使用这个函数。在取得大型结果时，如果不想冒着耗尽内存的风险，应使用plpy.cursor而不是plpy.execute。

plpy.prepare(*query* [, *argtypes*])
plpy.execute(*plan* [, *arguments* [, *max-rows*]])

plpy.prepare为一个查询准备执行计划。它的参数是一个查询串和一个参数类型列表（如果查询中有参数引用）。如下所示。

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1", ["text"])
```

text是要为$1传递的变量的类型。如果不想给查询传递任何参数，第二个参数就是可选的。

在准备好一个语句后，可以使用函数plpy.execute的一种变体来运行它。

```
rv = plpy.execute(plan, ["name"], 5)
```

把计划作为第一个参数传递（而不是查询字符串），并且把要替换到查询中的值列表作为第二个参数传递。如果查询不需要任何参数，则第二个参数是可选的。和前面一样，第三个参数是可选的，它用来指定行数限制。

另外，可以在计划对象上调用execute方法，如下所示。

```
rv = plan.execute(["name"], 5)
```

查询参数以及结果行域会按照第 9.3 节 “数据值”中所述在 UXsinoDB 和 Python 数据类型之间转换。

在使用 PL/Python 模块准备一个计划时，它会被自动保存。其含义可以阅读 SPI 文档（第 10 章 服务器编程接口）。为了有效在函数调用之间利用这种特性，需要使用一种持久化存储字典SD或者GD（见第 9.4 节 “共享数据”）。如下所示。

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
  if "plan" in SD:
    plan = SD["plan"]
  else:
    plan = plpy.prepare("SELECT 1")
    SD["plan"] = plan
  # 函数的剩余部分
$$ LANGUAGE plpythonu;
```

plpy.cursor(*query*)
plpy.cursor(*plan* [, *arguments*])

plpy.cursor函数接受和plpy.execute相同的参数（行数限制除外）并且返回一个游标对象，它允许以较小的块来处理大型的结果集合。和plpy.execute一样（行数限制除外），既可以使用一个查询字符串，也可以使用带有参数列表的计划对象，或者cursor函数可以作为计划对象的一个方法来调用。

游标对象提供了一种fetch方法，它接受一个整数参数并返回一个结果对象。每次调用fetch，返回的对象将包含下一批行，行数不会超过参数值。一旦所有的行都被消耗掉，fetch会开始

返回一个空的结果对象。游标对象也提供一种迭代器接口，它一次得到一行直到所有行被耗尽。用这种方法取得的数据不会被作为结果对象返回，而是以字典的形式返回，每一个字典对应于一个结果行。

一个大型表中针对上述两种方式处理数据，如下所示。

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
        odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
        if row['num'] % 2:
            odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
odd = 0
plan = plpy.prepare("select num from largetable where num % $1 <> 0", ["integer"])
rows = list(plpy.cursor(plan, [2]))  # or: = list(plan.cursor([2]))

return len(rows)
$$ LANGUAGE plpythonu;
```

游标会被自动丢弃掉。但是如果想要显式地释放游标所持有的所有资源，可使用close方法。一旦被关闭，就再也不能从游标中取得数据。

> **提示**
>
> 不要把plpy.cursor创建的游标对象与Python Database API specification定义的 DB-API 游标弄混。除了名字之外，它们之间没有任何共同点。

## 9.7.2. 捕捉错误

访问数据库的函数可能会碰到错误，这将导致函数中止并且产生异常。plpy.execute和plpy.prepare都能产生plpy.SPIError的一个子类的实例，这默认将终止该函数。通过使用try/except结构，这种错误可以像其他 Python 异常一样被处理。如下所示。

```
CREATE FUNCTION try_adding_joe() RETURNS text AS $$
```

```
    try:
        plpy.execute("INSERT INTO users(username) VALUES ('joe')")
    except plpy.SPIError:
        return "something went wrong"
    else:
        return "Joe added"
$$ LANGUAGE plpythonu;
```

产生的异常的实际类对应于特定的导致该错误的情况。模块plpy.spiexceptions为每一种UXsinoDB情况定义了一个异常类，并且根据情况的名称命名。例如：division_by_zero变成DivisionByZero，unique_violation变成UniqueViolation，fdw_error变成FdwError，等等等等。这些异常类的每一种都是从SPIError继承而来。这种分离让处理特定错误更加容易，如下所示。

```
CREATE FUNCTION insert_fraction(numerator int, denominator int) RETURNS text AS $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 / $2)", ["int", "int"])
    plpy.execute(plan, [numerator, denominator])
except spiexceptions.DivisionByZero:
    return "denominator cannot equal zero"
except spiexceptions.UniqueViolation:
    return "already have that fraction"
except plpy.SPIError, e:
    return "other error, SQLSTATE %s" % e.sqlstate
else:
    return "fraction inserted"
$$ LANGUAGE plpythonu;
```

注意因为所有来自于plpy.spiexceptions模块的异常都继承自SPIError，一个处理它的except子句将捕捉任何数据库访问错误。

作为另一种处理不同错误情况的方法，可以捕捉SPIError异常并且在except块中通过查看异常对象的sqlstate属性来判断错误情况。这种属性是包含着“SQLSTATE”错误代码的一个字符串值。这种方法提供了近乎相同的功能。

# 9.8. 显式子事务

按第 9.7.2 节 “捕捉错误”中所述的从数据库访问导致的错误中恢复可能导致不好的情况：某些操作在其中一个操作失败之前已经成功，并且在从错误中恢复后这些操作的数据形成了一种不一致的状态。PL/Python 通过显式子事务的形式为这种问题提供了一套解决方案。

## 9.8.1. 子事务上下文管理器

考虑一个实现在两个账户间进行转账的函数，如下所示。

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
try:
    plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
    plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
```

```
else:
   result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

如果第二个UPDATE语句导致产生一个异常，这个函数将会报告该错误，但是第一个UPDATE的结果却不会被提交。换句话说，资金将从 Joe 的账户中收回，而不会转移到 Mary 的账户中。

为了避免这类问题，可以把plpy.execute包裹在显式子事务中。plpy模块提供了一种助手对象来管理用plpy.subtransaction()函数创建的显式子事务。这个函数创建的对象实现了上下文管理器接口。通过使用显式子事务，如下所示。

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
try:
   with plpy.subtransaction():
      plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
      plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError, e:
   result = "error transferring funds: %s" % e.args
else:
   result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

注意仍需使用try/catch。否则异常会传播到 Python 栈的顶层并且将导致整个函数以一个UXsinoDB错误中止，这样不会有任何行被插入到operations表。子事务上下文管理器不会捕捉错误，它只确保在其范围内执行的所有数据库操作将被原子性地提交或者回滚。在任何类型的异常（并非只是数据库访问产生的错误）退出时，会发生子事务块回滚。在显式子事务块内部产生的常规 Python 异常也会导致子事务被回滚。

## 9.8.2. 更旧的 Python 版本

Python 2.6 中默认可用的是使用with关键词的上下文管理器语法。如果 PL/Python 用的是一种较老的 Python 版本，仍然可以使用显式子事务，尽管不是那么透明。可以使用别名enter和exit调用子事务管理器的__enter__和__exit__函数。转移资金的示例函数，如下所示。

```
CREATE FUNCTION transfer_funds_old() RETURNS void AS $$
try:
   subxact = plpy.subtransaction()
   subxact.enter()
   try:
      plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
      plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
   except:
      import sys
      subxact.exit(*sys.exc_info())
      raise
   else:
      subxact.exit(None, None, None)
except plpy.SPIError, e:
```

```
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"

plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

> **注意**
>
> 尽管 Python 2.5 中实现了上下文管理器，要在那个版本中使用with语法，需要使用一个future 语句。不过，由于实现细节的原因，不能在 PL/Python 函数中使用 future 语句。

# 9.9. 事务管理

在从顶层调用的过程中或者从顶层调用的匿名代码块（DO命令）中，可以控制事务。要提交当前的事务，可调用plpy.commit()。要回滚当前事务，可调用plpy.rollback()（注意不能通过plpy.execute或类似的函数运行SQL命令COMMIT或者ROLLBACK。这类工作必须用这些函数完成）。在事务结束以后，一个新的事务会自动开始，因此没有独立的函数用来开始新事务。

如下所示。

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plpythonu
AS $$
for i in range(0, 10):
    plpy.execute("INSERT INTO test1 (a) VALUES (%d)" % i)
    if i % 2 == 0:
        plpy.commit()
    else:
        plpy.rollback()
$$;

CALL transaction_test1();
```

当一个显式的子事务处于活跃状态时，事务不能被结束。

# 9.10. 实用函数

plpy模块也提供了函数

plpy.debug(*msg, **kwargs*)
plpy.log(*msg, **kwargs*)
plpy.info(*msg, **kwargs*)
plpy.notice(*msg, **kwargs*)
plpy.warning(*msg, **kwargs*)
plpy.error(*msg, **kwargs*)
plpy.fatal(*msg, **kwargs*)

plpy.error和plpy.fatal实际上会产生一个 Python 异常（如果没被捕捉），它会被传播到调用查询中导致当前事务或者子事务被中止。raise plpy.Error(*msg*)和raise plpy.Fatal(*msg*)分别等效于调

用plpy.error(*msg*)和plpy.fatal(*msg*)，不过raise形式不允许传递关键词参数。其他函数只生成不同优先级的消息。一个特定优先级的消息是被报告给客户端、写入服务器日志还是两者都做，由log_min_messages和client_min_messages配置变量控制。

*msg*参数被给定位一个位置参数。为了向后兼容，可以给出多于一个位置参数。在那种情况下，位置参数形成的元组的字符串表达将会变成报告给客户端的消息。

下列 keyword-only 参数会被接受。

detail
hint
sqlstate
schema_name
table_name
column_name
datatype_name
constraint_name
作为 keyword-only 参数传递的对象的字符串表达可以用来丰富报告给客户端的消息。如下所示。

```
CREATE FUNCTION raise_custom_exception() RETURNS void AS $$
plpy.error("custom exception message",
        detail="some info about exception",
        hint="hint for users")
$$ LANGUAGE plpythonu;

=# SELECT raise_custom_exception();
ERROR:  plpy.Error: custom exception message
DETAIL:  some info about exception
HINT:  hint for users
CONTEXT:  Traceback (most recent call last):
  PL/Python function "raise_custom_exception", line 4, in <module>
    hint="hint for users")
PL/Python function "raise_custom_exception"
```

另一组工具函数是plpy.quote_literal(*string*)、plpy.quote_nullable(*string*)以及plpy.quote_ident(*string*)。它们等效于内建引用函数。在构建临时查询时它们能派上用场。例 6.1 “在动态查询中引用值”中动态 SQL 的 PL/Python 等效体如下所示。

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

# 9.11. 环境变量

某些 Python 解释器接受的环境变量也能被用来影响 PL/Python 行为。它们需要在主 UXsinoDB 服务器进程的环境中设置，例如在一个启动脚本中设置。可用的环境变量取决于 Python 的版本，细节可见 Python 文档。在编写这份文档时，下面的环境变量可以对 PL/Python 产生影响（假定有一个合适的 Python 版本）。

- PYTHONHOME

- PYTHONPATH
- PYTHONY2K
- PYTHONOPTIMIZE
- PYTHONDEBUG
- PYTHONVERBOSE
- PYTHONCASEOK
- PYTHONDONTWRITEBYTECODE
- PYTHONIOENCODING
- PYTHONUSERBASE
- PYTHONHASHSEED

（Python 的实现细节似乎超出了 PL/Python 的控制范围，某些列在python手册页上的环境变量只在命令行解释器中有效，但在嵌入式 Python 解释器中无效）。

# 第 10 章 服务器编程接口

服务器编程接口（SPI）给予用户定义C函数编写者在其函数内运行SQL命令的能力。SPI是一组接口函数，它们可以简化对解析器、规划器和执行器的访问。SPI也做一些内存管理。

> **注意**
>
> 可用的过程语言提供了多种方法从函数中执行 SQL 命令。大部分这些设施都是基于 SPI 的，因此这个文档也对那些语言的用户有用。

注意如果一个通过 SPI 调用的命令失败，那么控制将会返回到C函数中。当然啦，C函数所在的事务或者子事务将被回滚（这可能看起来令人惊讶，因为据文档所说 SPI 函数大多数都有错误返回约定。但是那些约定只适用于在 SPI 函数本身内部检测到的错误）。通过在可能失败的 SPI 调用周围建立自己的子事务可以在错误之后恢复控制。

SPI成功时返回一个非负结果（要么通过一个返回的整数值，要么如下所述放在全局变量SPI_result中）。错误时，将会返回一个负结果或者NULL。

使用 SPI 的源代码文件必须包括头文件executor/spi.h。

## 10.1. 接口函数

# 名称

SPI_connect, SPI_connect_ext — 连接一个C函数到 SPI 管理器

# 大纲

int SPI_connect(void)

int SPI_connect_ext(int *options*)

# 描述

SPI_connect从一个C函数调用中打开一个到 SPI 管理器的连接。如果想要通过 SPI 执行命令，必须调用这个函数。有一些功能性 SPI 函数可以从未连接的C函数中调用。

SPI_connect_ext会做同样的事情，但是有一个允许传递选项标志的参数。当前有如下选项值可用。

- SPI_OPT_NONATOMIC

  设置SPI连接为nonatomic，这表示允许事务控制调用SPI_commit、SPI_rollback以及SPI_start_transaction。否则，调用这些函数将立即导致错误。

SPI_connect()等效于SPI_connect_ext(0)。

# 返回值

SPI_OK_CONNECT

成功时

SPI_ERROR_CONNECT

错误时

# 名称

SPI_finish — 将一个C函数从 SPI 管理器断开

# 大纲

int SPI_finish(void)

# 描述

SPI_finish关闭一个到 SPI 管理器的现有连接。必须在完成C函数的当前调用中所需的 SPI 操作之后必须调用这个函数。不过，如果通过elog(ERROR)中断了事务，无须担心这个函数的调用。在那种情况下，SPI 将自己自动进行清理。

# 返回值

SPI_OK_FINISH

如果正确地断开连接

SPI_ERROR_UNCONNECTED

如果从一个未连接的C函数中调用

# 名称

SPI_execute — 执行一个命令

# 大纲

int SPI_execute(const char * *command*, bool *read_only*, long *count*)

# 描述

SPI_execute执行指定的 SQL 命令以获得*count*行。如果*read_only*为true，该命令必须是只读的，并且执行开销也会有所降低。

只能从一个已连接的C函数中调用这个函数。

如果*count*为零，那么该命令会为其所适用的所有行执行。如果*count*大于零，那么会检索不超过*count*行，当到达该计数时执行会停止，这很像为查询增加一个LIMIT子句。如下所示。

SPI_execute("SELECT * FROM foo", true, 5);

会从表中检索至多 5 行。注意这样一个限制只有当命令真正返回行时才有效。如下所示。

SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);

插入所有来自于bar的行，而忽略*count*参数。不过，通过

SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);

将插入至多 5 行，因为在第五个RETURNING结果行被检索到后执行就会停止。

可以在一个字符串中传递多个命令，SPI_execute会返回最后一个被执行的命令的结果。 *count*限制单独适用于每一个命令（即便只有最后一个结果会被实际返回）。该限制 不适用于由规则产生的任何隐藏命令。

当*read_only*是false时， SPI_execute增加命令计数器并且在执行字符串中每一个命令之前 计算一个新的snapshot。如果当前事务隔离级别是SERIALIZABLE或REPEATABLE READ， 该快照并不会实际改变。但是在READ COMMITTED模式中，快照更新允许每个命令看到来自其他会话中新近已提交事务 的结果。当命令正在修改数据库时，这对一致性行为非常重要。

当*read_only*是true时， SPI_execute不更新快照或者命令计数器，并且它只允许纯 SELECT命令出现在命令字符串中。这些命令被使用之前为周围查询 建立的快照来执行。这种执行模式要比读/写模式更快，因为消除了每个命令跟新快照的开销。 它也允许建立真正stable的函数：因为连续执行将会使用同一个快照，因此结果不会有改变。

在一个使用 SPI 的单一函数中混合只读和读写命令通常是不明智的， 这样可能会导致非常令人困惑的行为，因为只读查询将看不到任何 由读写查询完成的数据库更新结果。

被执行的（最后一个）命令的实际行数使用全局变量SPI_processed返回。如果该函数的返回值是SPI_OK_SELECT、SPI_OK_INSERT_RETURNING、SPI_OK_DELETE_RETURNING或者SPI_OK_UPDATE_RETURNING，那么可以使用全局指针SPITupleTable *SPI_tuptable来访问结果行。

某些工具命令（例如EXPLAIN）也返回行集合，并且在这些情况中SPI_tuptable也会包含该结果。某些工具命令（COPY、CREATE TABLE AS）不返回一个行集合，因此SPI_tuptable为 NULL，但是它们仍然会在SPI_processed中返回被处理的行数。

结构SPITupleTable被定义，如下所示。

```
typedef struct
{
   MemoryContext tuptabcxt;   /* 结果表的内存上下文 */
   uint64     alloced;      /* 已分配值的数量 */
   uint64     free;         /* 空限值的数量 */
   TupleDesc   tupdesc;      /* 行描述符 */
   HeapTuple  *vals;         /* 行 */
} SPITupleTable;
```

*vals*是一个行指针的数组（可用项的数量由SPI_processed给出）。 *tupdesc*是一个行描述符，可以把它传递给 SPI 函数来处理行。*tuptabcxt*、 *alloced*和*free*是不准备给 SPI 调用者使用的内部域。

SPI_finish释放在当前的C函数中已分配的所有SPITupleTable。如果已经用完了一个结果表，可以通过调用SPI_freetuptable提早释放它。

## 参数

const char * *command*

包含要执行命令的字符串

bool *read_only*

对只读执行为true

long *count*

要返回的最大行数，或者用0表示没有限制

## 返回值

如果命令的执行成功，那么将会返回下列（非负）值之一。

SPI_OK_SELECT

如果执行了一个SELECT（但不是SELECT INTO）

SPI_OK_SELINTO

如果执行了一个SELECT INTO

SPI_OK_INSERT

如果执行了一个INSERT

SPI_OK_DELETE

如果执行了一个DELETE

SPI_OK_UPDATE

如果执行了一个UPDATE

SPI_OK_INSERT_RETURNING

如果执行了一个INSERT RETURNING

SPI_OK_DELETE_RETURNING

如果执行了一个DELETE RETURNING

SPI_OK_UPDATE_RETURNING

如果执行了一个UPDATE RETURNING

SPI_OK_UTILITY

如果执行了一个工具命令（例如CREATE TABLE）

SPI_OK_REWRITTEN

如果该命令被一个规则重写成了另一类命令（例如UPDATE变成了一个INSERT）

发生错误时，将会返回下列负值之一。

SPI_ERROR_ARGUMENT

如果*command*为NULL或者*count*小于 0

SPI_ERROR_COPY

如果尝试COPY TO stdout或者COPY FROM stdin

SPI_ERROR_TRANSACTION

如果尝试了一个事务操纵命令（ BEGIN、 COMMIT、 ROLLBACK、 SAVEPOINT、 PREPARE TRANSACTION、 COMMIT PREPARED、 ROLLBACK PREPARED或者其他变体）

SPI_ERROR_OPUNKNOWN

如果命令类型位置（不应该会发生）

SPI_ERROR_UNCONNECTED

如果从未连接的C函数中调用

## 注解

所有 SPI 查询执行函数都会设置SPI_processed和SPI_tuptable（只是指针，而不是结构的内容）。如果需要在以后访问SPI_execute或另一个查询执行函数的结果表，请将这两个全局变量保存到本地的C函数变量中。

# 名称

SPI_exec — 执行一个读/写命令

# 大纲

int SPI_exec(const char * *command*, long *count*)

# 描述

SPI_exec和 SPI_execute相同，但后者的 *read_only*参数的值总是取 false。

# 参数

const char * *command*

包含要执行的命令的字符串

long *count*

要返回的最大行数，0表示没有限制

# 返回值

见SPI_execute。

# 名称

SPI_execute_with_args — 用线外参数执行一个命令

# 大纲

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

# 描述

SPI_execute_with_args执行一个可能包括 对外部提供的参数引用的命令。命令文本用 $*n*引用一个参数，并且调用 会为每一个这种符号指定数据类型和值。 *read_only*和 *count*的解释与SPI_execute中相同。

相对于SPI_execute，这个例程的主要优点是数据值可以被插入到命令中而无需冗长的引用/转义，并且因此减少了 SQL 注入攻击的风险。

可以通过在SPI_prepare后面跟上SPI_execute_plan达到相似的结果。但是，使用这个函数时查询计划总是被定制成提供的指定参数值。对于一次 性的查询执行，这个函数应该更好。如果同样的命令需要用很多不同的参数执行，两种方法都可能会更快，这取决于重新做规划的代价与定制计划带来的好处之间的对比。

# 参数

const char * *command*

命令字符串

int *nargs*

输入参数的数量（$1、$2等等）。

Oid * *argtypes*

一个长度为*nargs*的数组， 包含参数的数据类型的OID

Datum * *values*

一个长度为*nargs*的数组， 包含实际的参数值

const char * *nulls*

一个长度为*nargs*的数组， 描述哪些参数为空值

如果*nulls*为NULL， 那么SPI_execute_with_args会假设没有参数 为空值。否则，如果对应的参数值为非空， *nulls* 数组的每一个项都应该是' '；如果对应参数值为空， *nulls*数组的项应为'n'（在后 面的情况中，对应的*values*项中的值没有 关系）。注意*nulls*不是一个文本字符串， 它只是一个数组：它不需要一个'\0'终止符。

bool *read_only*

对只读执行是true

long *count*

要返回的最大行数，0表示没有限制

## 返回值

该返回值和SPI_execute一样。

如果成功SPI_execute会设置 SPI_processed和 SPI_tuptable。

# 名称

SPI_prepare — 准备一个语句，但不执行它

# 大纲

SPIPlanPtr SPI_prepare(const char * *command*, int *nargs*, Oid * *argtypes*)

# 描述

SPI_prepare为指定的命令创建并且返回一个预备语句，但是并不执行该命令。该预备语句会在稍后使用SPI_execute_plan重复执行。

当相同的或者相似的命令要被重复执行时，通常来说只执行一次解析分析是有利的，并且更有利的是重用该命令的执行计划。SPI_prepare把一个命令字符串转换成一个预 备语句，它包装了解析分析的结果。如果发现为每一次执行都生成一个定制计划没有帮助，该预备语句也提供了一个地方缓存执行计划。

一个预备命令可以被一般化为在一个普通命令中应该出现常量的地方写上参数（$1、$2等等）。参数的实际值在 SPI_execute_plan被调用时指定。这让该预备语句可以比没有参数的形式用户与更广泛的情况。

SPI_prepare返回的语句只能在当前的C过程调用中使用，因为SPI_finish会释放为这样一个语句分配的内存。但是可以使用函数SPI_keepplan或SPI_saveplan把该语句保存更久。

# 参数

const char * *command*

命令字符串

int *nargs*

输入参数（$1、$2等等）的数量

Oid * *argtypes*

一个数组指针，它指向的数组包含参数的数据类型的 OID

# 返回值

SPI_prepare返回一个指向SPIPlan 的非空指针，它是一个表示一个预备语句的不透明结构。发生错误时， 将会返回NULL，并且 SPI_result将被设置为一个也被 SPI_execute使用的错误码，不过当 *command*为NULL、 或者*nargs*小于零、或者*nargs*大于 零但是*argtypes*为NULL 时它会被设置为SPI_ERROR_ARGUMENT。

# 注解

如果没有定义参数，在第一次使用SPI_execute_plan时将会创建一个一般的计划，并且把它用于所有的后续执行。如果有参数，SPI_execute_plan的前几次使用将根据提供的参数值产生定制计划。在使用同一个预备语句足够多次后，SPI_execute_plan将构建一个一般计划，并且如果它并不比定制计划昂贵太多， SPI_execute_plan将开始使用一般计划来取代每次都进行重新规划。如

果这种默认的行为不合适，可以通过传
递CURSOR_OPT_GENERIC_PLAN或CURSOR_OPT_CUSTOM_PLAN标志
给SPI_prepare_cursor，以分别强制使用一般或者定制计划。

尽管一个预备语句的要点是避免对语句的重复解析分析以及规划，只要语句中用到的数据库对象从上一次使用该预备语句以来经历过定义性（DDL）改变， UXsinoDB将会强制重新分析和重新规划该语句。还有，如果search_path的值从一个改变成下一个，该语句将 会使用新的search_path进行重新解析（后一种行为是从UXsinoDB 9.3 开始的新行为）。

这个函数只能从一个已连接的C函数调用。

SPIPlanPtr被声明为spi.h中的一种不透明结构类型的指针。尝试直接访问其内容是不明智的，因为那会让代码更有可能会在未来版本的UXsinoDB中无法运行。

SPIPlanPtr这个名字多少有点历史原因，因为该数据结构不再需要包含一个执行计划。

# 名称

SPI_prepare_cursor — 预备一个语句，但是不执行它

# 大纲

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,
                              Oid * argtypes, int cursorOptions)
```

# 描述

SPI_prepare_cursor和 SPI_prepare一样，不过它也允许说明规划器的 “游标选项”参数。这是一个位掩码，它的值如 nodes/parsenodes.h中 DeclareCursorStmt的*options*域所示。 SPI_prepare总是把该游标选项取做零。

# 参数

const char * *command*

命令字符串

int *nargs*

输入参数（$1、$2等等）的数量

Oid * *argtypes*

一个数组指针，它指向的数组包含参数的数据类型的 OID

int *cursorOptions*

整数形式的游标选项位掩码，零会导致默认行为

# 返回值

SPI_prepare_cursor具有和 SPI_prepare一样的返回习惯。

# 注解

在*cursorOptions*设置的有用的位包括如下选项。

- CURSOR_OPT_SCROLL
- CURSOR_OPT_NO_SCROLL
- CURSOR_OPT_FAST_PLAN
- CURSOR_OPT_GENERIC_PLAN
- CURSOR_OPT_CUSTOM_PLAN

注意CURSOR_OPT_HOLD被特别地忽略。

# 名称

SPI_prepare_params — 预备一个语句，但是不执行它

# 大纲

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

# 描述

SPI_prepare_params为指定的命令创建并 返回一个预备语句，但是不执行该命令。这个函数等效于 SPI_prepare_cursor，此外调用者可以指定 解析器钩子函数来控制外部参数引用的解析。

# 参数

const char * *command*

命令字符串

ParserSetupHook *parserSetup*

解析器钩子设置函数

void * *parserSetupArg*

用于*parserSetup*的转嫁参数

int *cursorOptions*

整数形式的游标选项位掩码，零会导致默认行为

# 返回值

SPI_prepare_params具有和 SPI_prepare相同的返回习惯。

# 名称

SPI_getargcount — 返回一个由SPI_prepare 准备好的语句所需的参数数量

# 大纲

int SPI_getargcount(SPIPlanPtr *plan*)

# 描述

SPI_getargcount返回执行一个由 SPI_prepare准备好的语句所需的参数数量。

# 参数

SPIPlanPtr *plan*

预备语句（由SPI_prepare返回）

# 返回值

*plan*所期望的参数计数。如果该 *plan*为NULL或者无效， SPI_result会被设置为 SPI_ERROR_ARGUMENT 并且返回 -1。

# 名称

SPI_getargtypeid — 为由SPI_prepare 准备好的一个语句的一个参数返回其数据类型 OID

# 大纲

Oid SPI_getargtypeid(SPIPlanPtr *plan*, int *argIndex*)

# 描述

SPI_getargtypeid返回由 SPI_prepare准备好的一个语句的 第*argIndex*个参数的类型的 OID。 第一个参数的索引为零。

# 参数

SPIPlanPtr *plan*

预备语句（由SPI_prepare返回）

int *argIndex*

参数的索引，从零开始

# 返回值

给定索引处的参数的类型 OID。如果该 *plan*为NULL或者无效， 或者*argIndex*小于零或者小于为*plan*声明的参数数量， SPI_result会被设置为 SPI_ERROR_ARGUMENT并且将会返回 InvalidOid。

# 名称

SPI_is_cursor_plan — 如果一个由SPI_prepare预备好 的语句可以用于SPI_cursor_open则返回true

# 大纲

bool SPI_is_cursor_plan(SPIPlanPtr *plan*)

# 描述

如果一个由SPI_prepare预备好的语句可以被作为一个参数传递给SPI_cursor_open，SPI_is_cursor_plan会返回true。否则返回false。原则是该*plan*表示一个单一命令并且这个命令向其调用者返回元组。例如，只要不含INTO子句，SELECT就被允许，而只有包含一个RETURNING子句时才允许UPDATE。

# 参数

SPIPlanPtr *plan*

预备语句（由SPI_prepare返回）

# 返回值

如果该*plan*能产生一个游标则返回true，否则返回false并且把SPI_result设置为零。如果不可能决定答案 （例如，如果*plan*为NULL或无效，或者在没有连接到 SPI 时调用），那么SPI_result会被设置为一个合适的错误码并且返回false。

# 名称

SPI_execute_plan — 执行一个由SPI_prepare预备好的语句

# 大纲

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                     bool read_only, long count)
```

# 描述

SPI_execute_plan执行一个由 SPI_prepare或其同类方法准备好的语句。 *read_only*和 and *count*的解释和 SPI_execute中相同。

# 参数

SPIPlanPtr *plan*

预备语句（由SPI_prepare返回）

Datum * *values*

一个实际参数值的数组。必须和语句的参数数量等长。

const char * *nulls*

一个描述哪些参数为空值的数组。必须和语句的参数数量等长。

如果*nulls*为NULL， 那么SPI_execute_plan会假设没有参数 为空值。否则，如果对应的参数值为非空， *nulls* 数组的每一个项都应该是' '；如果对应参数值为空， *nulls*数组的项应为'n'（在后 面的情况中，对应的*values*项中的值没有 关系）。注意*nulls*不是一个文本字符串， 它只是一个数组：它不需要一个'\0'终止符。

bool *read_only*

true表示只读执行

long *count*

要返回的行的最大数量，或者用0表示没有限制

# 返回值

返回值和SPI_execute相同，还有下列额外可能的错误（负值）结果如下所示。

SPI_ERROR_ARGUMENT

如果*plan*为NULL 或者非法，或者*count*小于 0

SPI_ERROR_PARAM

如果*values*为NULL但是 *plan*被准备时用了一些参数

成功时，就像在SPI_execute中会设置 SPI_processed和 SPI_tuptable。

# 名称

SPI_execute_plan_with_paramlist — 执行一个由SPI_prepare预备好的语句

# 大纲

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                    ParamListInfo params,
                                    bool read_only,
                                    long count)
```

# 描述

SPI_execute_plan_with_paramlist执行一个由SPI_prepare准备好的语句。这个函数与SPI_execute_plan等效，不过被传递给该查询的参数值的信息以不同的方式呈现。ParamListInfo表现形式更方便于把这种格式的值向下传递。它也支持通过ParamListInfo中指定的钩子函数动态设置参数。

# 参数

SPIPlanPtr *plan*

预备语句（由SPI_prepare返回）

ParamListInfo *params*

包含参数类型和值的数据结构，如果没有则为 NULL

bool *read_only*

true表示只读执行

long *count*

要返回的行的最大数量，或者用0表示没有限制

# 返回值

返回值和SPI_execute_plan相同。

成功时，在SPI_execute_plan中会设置 SPI_processed和 SPI_tuptable。

# 名称

SPI_execp — 以读/写模式执行一个语句

# 大纲

int SPI_execp(SPIPlanPtr *plan*, Datum * *values*, const char * *nulls*, long *count*)

# 描述

SPI_execp与 SPI_execute_plan相同，不过后者的 *read_only*参数总是取false。

# 参数

SPIPlanPtr *plan*

预备语句（由SPI_prepare返回）

Datum * *values*

实际参数值的数组。长度必须等于该语句的参数数量。

const char * *nulls*

描述哪些参数为空值的数据。长度必须等于该语句的参数数量。

如果*nulls*为NULL， 那么SPI_execp会假设没有参数 为空值。否则，如果对应的参数值为非空， *nulls* 数组的每一个项都应该是' '；如果对应参数值为空， *nulls*数组的项应为'n'（在后面的情况中，对应的*values*项中的值没有 关系）。注意*nulls*不是一个文本字符串， 它只是一个数组：它不需要一个'\0'终止符。

long *count*

要返回的行的最大数量，或者用0表示没有限制

# 返回值

见SPI_execute_plan。

成功时，就像在SPI_execute中会设置 SPI_processed和 SPI_tuptable。

# 名称

SPI_cursor_open — 使用由SPI_prepare创建的 语句建立一个游标

# 大纲

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
                       Datum * values, const char * nulls,
                       bool read_only)
```

# 描述

SPI_cursor_open建立一个游标（在内部是一个 portal），该游标将执行由SPI_prepare准备好 的一个语句。参数具有和SPI_execute_plan的 相应参数相同的含义。

使用一个游标而不是直接执行该语句有两个好处。首先，可以一次只取出一些结果行，避免为返回很多行的查询过度使用内存。其次，一个 portal可以比当前的C函数生存更长时间（事实上，它可以生存到当前事务结束）。把 portal 的名称返回给该C函数的调用者提供了一种将一个行集合返回为结果的方法。

被传入的参数数据将被复制到游标的 portal 中，因此在该游标仍然存在时可以释放掉被传入的参数数据。

# 参数

const char * *name*

portal 的名字，或者设置成NULL 让系统选择一个名称

SPIPlanPtr *plan*

预备语句（由SPI_prepare返回）

Datum * *values*

实际参数值的数组。长度必须等于该语句的参数数量。

const char * *nulls*

描述哪些参数是空值的数据。长度必须等于该语句的参数数量。

如果*nulls*为NULL，那么SPI_cursor_open会假设没有参数为空值。否则，如果对应的参数值为非空， *nulls*数组的每一个项都应该是' '；如果对应参数值为空，*nulls*数组的项应为'n'（在后 面的情况中，对应的*values*项中的值没有关系）。注意*nulls*不是一个文本字符串，它只是一个数组：它不需要一个'\0'终止符。

bool *read_only*

true表示只读执行

# 返回值

指向包含该游标的 portal 的指针。注意这里没有错误返回约定，任何错误都将通过elog报告。

# 名称

SPI_cursor_open_with_args — 使用一个查询和参数建立一个游标

# 大纲

```
Portal SPI_cursor_open_with_args(const char *name,
                                 const char *command,
                                 int nargs, Oid *argtypes,
                                 Datum *values, const char *nulls,
                                 bool read_only, int cursorOptions)
```

# 描述

SPI_cursor_open_with_args建立一个将 执行指定查询的游标（在内部是一个 portal）。大部分参数具有和 SPI_prepare_cursor 和SPI_cursor_open中相应参数相同的含 义。

对于一次性的查询执行，这个函数应该比 SPI_prepare_cursor加上其后的 SPI_cursor_open更好。如果相同的命令 要被用很多不同的参数执行，哪种方法更快就要取决于重做计划的 代价与定制计划带来的好处之间谁更有利。

被传入的参数数据将被复制到游标的 portal 中，因此在该游标仍然存在时 可以释放掉被传入的参数数据。

# 参数

const char * *name*

portal 的名字，或者设置成NULL 让系统选择一个名称

const char * *command*

命令字符串

int *nargs*

输入参数的数量（$1、$2等等）

Oid * *argtypes*

一个长度为*nargs*的数组，它包含参数的 数据类型的OID

Datum * *values*

一个长度为*nargs*的数组，它包含实际的参数值

const char * *nulls*

一个长度为*nargs*的数组，它描述哪些参数为空值

如果*nulls*为NULL，那么SPI_cursor_open_with_args会假设没有参数为空值。否则，如果对应的参数值为非空， *nulls*数组的每一个项都应该是' '；如果对应参数值为空，*nulls*数组的项应为'n'（在后 面的情况中，对应的*values*项中的值没有关系）。注意*nulls*不是一个文本字符串，它只是一个数组：它不需要一个'\0'终止符。

bool *read_only*

true表示只读执行

int *cursorOptions*

游标选项的整数型位掩码，为零会产生默认行为

## 返回值

指向包含该游标的 portal 的指针。注意这里没有错误返回约定，任何错误都将通过elog报告。

# 名称

SPI_cursor_open_with_paramlist — 使用参数建立一个游标

# 大纲

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

# 描述

SPI_cursor_open_with_paramlist建立一个游标（在内部是一个 portal），它将执行一个由SPI_prepare准备好的语句。这个函数等效于 SPI_cursor_open，不过被传递给该查询的参数值的信息以不同的方式呈现。ParamListInfo表现形式更方便于把这种格式的值向下传递。它也支持通过 ParamListInfo中指定的钩子函数动态设置参数。

被传入的参数数据将被复制到游标的 portal 中，因此在该游标仍然存在时可以释放掉被传入的参数数据。

# 参数

const char * *name*

portal 的名字，或者设置成NULL 让系统选择一个名称

SPIPlanPtr *plan*

预备语句（由SPI_prepare返回）

ParamListInfo *params*

包含参数类型和值的数据结构，如果没有就为 NULL

bool *read_only*

true表示只读执行

# 返回值

指向包含该游标的 portal 的指针。注意这里没有错误返回约定， 任何错误都将通过elog报告。

# 名称

SPI_cursor_find — 用名称查找一个现有的游标

# 大纲

Portal SPI_cursor_find(const char * *name*)

# 描述

SPI_cursor_find用名称查找一个现有的 portal。 这主要被用于解析由其他某个函数返回的一个油表名称。

# 参数

const char * *name*

该 portal 的名称

# 返回值

带有指定名称的 portal 的指针，如果没有找到就是 NULL

# 名称

SPI_cursor_fetch — 从一个游标取出一些行

# 大纲

void SPI_cursor_fetch(Portal *portal*, bool *forward*, long *count*)

# 描述

SPI_cursor_fetch从一个游标取得一些行。这等效于 SQL 命令FETCH的一个子集（更多功能见SPI_scroll_cursor_fetch）。

# 参数

Portal *portal*

包含该游标的 portal

bool *forward*

为真表示向前获取，为假表示向后获取

long *count*

要取得的最大行数

# 返回值

成功时，就像在SPI_execute中会设置 SPI_processed和 SPI_tuptable。

# 注解

如果该游标的计划不是用CURSOR_OPT_SCROLL选项创建的，向后获取会失败。

# 名称

SPI_cursor_move — 移动一个游标

# 大纲

void SPI_cursor_move(Portal *portal*, bool *forward*, long *count*)

# 描述

SPI_cursor_move跳过一个游标中的一些行。这等效于 SQL 命令**MOVE**的一个子集（更多的功能请见SPI_scroll_cursor_move）。

# 参数

Portal *portal*

包含该游标的 portal

bool *forward*

为真表示前移，为假表示后移

long *count*

要移动的最大行数

# 注解

如果该游标的计划不是用CURSOR_OPT_SCROLL选项创建的，向后移动会失败。

# 名称

SPI_scroll_cursor_fetch — 从一个游标取出一些行

# 大纲

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction,
                             long count)
```

# 描述

SPI_scroll_cursor_fetch从一个游标中取出一些行。这等效于 SQL 命令**FETCH**。

# 参数

Portal *portal*

包含该游标的 portal

FetchDirection *direction*

可取值如下所示。

- FETCH_FORWARD
- FETCH_BACKWARD
- FETCH_ABSOLUTE
- FETCH_RELATIVE

long *count*

FETCH_FORWARD或FETCH_BACKWARD方式中要取出的行数。

FETCH_ABSOLUTE方式中要取出的绝对行号。

FETCH_RELATIVE方式中要取出的相对行号。

# 返回值

成功时，就像在SPI_execute中会设置 SPI_processed和 SPI_tuptable。

# 注解

参数*direction*和*count*的详细解释请见SQL **FETCH**命令。

如果该游标的计划不是用CURSOR_OPT_SCROLL选项创建的，除FETCH_FORWARD之外的方向值会失败。

# 名称

SPI_scroll_cursor_move — 移动一个游标

# 大纲

void SPI_scroll_cursor_move(Portal *portal*, FetchDirection *direction*,
long *count*)

# 描述

SPI_scroll_cursor_move在一个游标中跳过 一定数量的行。这等效于 SQL 命令**MOVE**。

# 参数

Portal *portal*

包含该游标的 portal

FetchDirection *direction*

可取值如下所示。

- FETCH_FORWARD
- FETCH_BACKWARD
- FETCH_ABSOLUTE
- FETCH_RELATIVE

long *count*

FETCH_FORWARD或者 FETCH_BACKWARD方式中要移动的行数。

FETCH_ABSOLUTE方式中要移动到的绝对行号。

FETCH_RELATIVE方式中要移动到的相对行号。

# 返回值

成功时，就像在SPI_execute中会设置 SPI_processed。 SPI_tuptable被设置为NULL， 因为这个函数不需要返回行。

# 注解

参数*direction*和 *count*的详细解释请见SQL**FETCH**命令。

如果该游标的计划不是用CURSOR_OPT_SCROLL选项创建的，除FETCH_FORWARD之外的方向值会失败。

# 名称

SPI_cursor_close — 关闭一个游标

# 大纲

void SPI_cursor_close(Portal *portal*)

# 描述

SPI_cursor_close关闭一个之前创建的游标并且释放它的 portal 存储。

所有打开的游标会在事务结束时自动被关闭。只有在希望尽快释放资源时，才需要调用SPI_cursor_close。

# 参数

Portal *portal*

包含该游标的 portal

# 名称

SPI_keepplan — 保存一个预备语句

# 大纲

int SPI_keepplan(SPIPlanPtr *plan*)

# 描述

SPI_keepplan保存一个被传入的语句（由SPI_prepare准备好），这样它将不会被SPI_finish或者事务管理器释放。这能够在当前会话的后续C函数调用中重用预备语句。

# 参数

SPIPlanPtr *plan*

要保存的预备语句

# 返回值

成功返回 0；如果*plan*为NULL 或者无效则返回SPI_ERROR_ARGUMENT

# 注解

这个函数通过指针调整的方法（不需要数据复制）将被传入的语句重定位到永久存储中。如果后来需要删除它，可以对它使用SPI_freeplan。

# 名称

SPI_saveplan — 保存一个预备语句

# 大纲

SPIPlanPtr SPI_saveplan(SPIPlanPtr *plan*)

# 描述

SPI_saveplan把一个被传入的语句（由SPI_prepare准备好）复制到不会被SPI_finish或者事务管理器释放的内存中。这能够在当前会话的后续C函数调用中重用预备语句。

# 参数

SPIPlanPtr *plan*

要保存的预备语句

# 返回值

要被复制的语句的指针；如果没有成功则返回NULL。 错误时，SPI_result会被这样设置：

SPI_ERROR_ARGUMENT

如果*plan*为NULL或无效

SPI_ERROR_UNCONNECTED

如果从一个未连接的C函数调用

# 注解

原始的被传入的语句不会被释放，因此可能希望在其上执行SPI_freeplan以避免在SPI_finish之前发生内存泄露。

在大部分情况下，SPI_keepplan更适合于 执行这种功能，因为它极大程度上达到了同样的结果而无需物理地 复制该预备语句的数据结构。

# 名称

SPI_register_relation — 建立一个短暂的命名关系，对当前SPI查询名称可用

# 大纲

int SPI_register_relation(EphemeralNamedRelation *enr*)

# 简介

SPI_register_relation用相关信息建立一个短暂的命名关系，它对通过当前SPI连接规划和执行的查询可用。

# 参数

EphemeralNamedRelation *enr*

短暂的命名关系的注册项

# 返回值

如果该命令的执行成功，则将返回下列（非负）值：

SPI_OK_REL_REGISTER

如果该关系已经成功地用名称注册

在出错时，会返回下列负值之一。

SPI_ERROR_ARGUMENT

如果*enr*是NULL或者其name字段是NULL

SPI_ERROR_UNCONNECTED

如果从一个未连接的C函数中调用

SPI_ERROR_REL_DUPLICATE

如果*enr*的name字段中指定的名称已经为这个连接注册

# 名称

SPI_unregister_relation — 从当前连接的注册项中移除一个短暂存在的关系

# 大纲

int SPI_unregister_relation(const char * *name*)

# 简介

SPI_unregister_relation从当前连接的注册项中移除一个短暂存在的关系。

# 参数

const char * *name*

关系的注册项名称

# 返回值

如果该命令的执行成功，则会返回下列（非负）值：

SPI_OK_REL_UNREGISTER

如果tuplestore已经被成功地从注册项中移除

出现错误时，会返回下列负值之一。

SPI_ERROR_ARGUMENT

如果*name*为NULL

SPI_ERROR_UNCONNECTED

如果从一个未连接的C函数中调用

SPI_ERROR_REL_NOT_FOUND

如果没有在当前连接的注册项中找到*name*

## 名称

SPI_register_trigger_data — 临时触发器数据，在 SPI 查询中使可用

## 大纲

int SPI_register_trigger_data(TriggerData *tdata)

## 简介

SPI_register_trigger_data会造出被触发器捕获的任何短暂存在的关系，它们对通过当前SPI连接规划和执行的查询可用。当前，这表示用REFERENCING OLD/NEW TABLE AS ... 子句定义的被AFTER触发器捕获的传递表。这个函数应该被一个PL触发器的处理器函数在连接之后调用。

## 参数

TriggerData *tdata

以fcinfo->context传递给触发器处理器函数的TriggerData对象

## 返回值

如果命令的执行成功，则会返回下列（非负）值：

SPI_OK_TD_REGISTER

如果被捕获的触发器数据（如果有）已经被成功地注册

出现错误时，会返回下列负值之一。

SPI_ERROR_ARGUMENT

如果*tdata*为NULL

SPI_ERROR_UNCONNECTED

如果从一个未连接的C函数中调用

SPI_ERROR_REL_DUPLICATE

如果任何触发器数据瞬时关系的名字已经为这个连接注册过

# 10.2. 接口支持函数

这里描述的函数提供了一个接口从SPI_execute 及其他 SPI 函数返回的结果集中抽取信息。

这一小节中描述的所有函数都可以被用在已连接和未连接的C函数中。

# 名称

SPI_fname — 为指定的列号确定列名

# 大纲

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

# 描述

SPI_fname返回一个指定列的列名的拷贝（当不再需要该列名拷贝后，可以使用pfree释放它）。

# 参数

TupleDesc *rowdesc*

输入行描述

int *colnumber*

列号（从 1 开始计）

# 返回值

列名；如果*colnumber*超出范围则返回NULL。出错时SPI_result会被设置成SPI_ERROR_NOATTRIBUTE。

# 名称

SPI_fnumber — 为一个指定的列名确定列号

# 大纲

int SPI_fnumber(TupleDesc *rowdesc*, const char * *colname*)

# 描述

SPI_fnumber返回指定列名的列号。

如果*colname*引用的是一个系统列（例如， ctid），那么将返回对应的负值列号。调用者应该小心地测试 返回值是不是正好为SPI_ERROR_NOATTRIBUTE 来检测错误；除非系统列应该被拒绝，测试结果是否小于或者等于零这种 方式是不正确的。

# 参数

TupleDesc *rowdesc*

输入行描述

const char * *colname*

列名

# 返回值

列号（用户定义的列从1开始计），如果没有找到所提到的列名则返回SPI_ERROR_NOATTRIBUTE。

# 名称

SPI_getvalue — 返回指定列的字符串值

# 大纲

char * SPI_getvalue(HeapTuple *row*, TupleDesc *rowdesc*, int *colnumber*)

# 描述

SPI_getvalue返回指定列的值的字符串表示。

结果在使用palloc分配的内存中返回（当不再需要该结果时，可以使用pfree释放该内存）。

# 参数

HeapTuple *row*

要检查的输入行

TupleDesc *rowdesc*

输入行描述

int *colnumber*

列号（从 1 开始计）

# 返回值

列值，如果列为空值、*colnumber*超出范围（SPI_result被设置为SPI_ERROR_NOATTRIBUTE）或者没有输出函数 可用（SPI_result被设置为SPI_ERROR_NOOUTFUNC）则返回NULL。

# 名称

SPI_getbinval — 返回指定列的二进制值

# 大纲

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,
                    bool * isnull)
```

# 描述

SPI_getbinval以内部格 式（以Datum类型）返回指定列的值。

这个函数不会为该 datum 分配新空间。在传引用数据类型的情况下， 返回值将是一个被传递行的指针。

# 参数

HeapTuple *row*

要检查的输入行

TupleDesc *rowdesc*

输入行描述

int *colnumber*

列号（从 1 开始计）

bool * *isnull*

列中是否为空值的标志

# 返回值

该列的二进制值会被返回。如果该列为空值，由*isnull*指向的变量将被设置为真，否则会被设置为假。

错误时SPI_result会被设置成SPI_ERROR_NOATTRIBUTE。

# 名称

SPI_gettype — 返回指定列的数据类型名称

# 大纲

char * SPI_gettype(TupleDesc *rowdesc*, int *colnumber*)

# 描述

SPI_gettype返回该指定列的数据类型名称的拷贝（当不再需要该拷贝后，可以使用pfree释放它）。

# 参数

TupleDesc *rowdesc*

输入行描述

int *colnumber*

列号（从 1 开始计）

# 返回值

指定列的数据类型名称，或者在错误时返回NULL。错误时SPI_result会被设置成SPI_ERROR_NOATTRIBUTE。

# 名称

SPI_gettypeid — 返回指定列的数据类型的OID

# 大纲

Oid SPI_gettypeid(TupleDesc *rowdesc*, int *colnumber*)

# 描述

SPI_gettypeid返回该指定列的数据类型的 OID。

# 参数

TupleDesc *rowdesc*

输入行描述

int *colnumber*

列号（从 1 开始计）

# 返回值

指定列的数据类型的OID，或者出错时返回 InvalidOid。出错时， SPI_result会被设置成 SPI_ERROR_NOATTRIBUTE。

# 名称

SPI_getrelname — 返回指定关系的名称

# 大纲

char * SPI_getrelname(Relation *rel*)

# 描述

SPI_getrelname返回该指定关系的名称的拷贝（当不再需要该拷贝后，可以使用pfree释放它）。

# 参数

Relation *rel*

输入关系

# 返回值

指定关系的名称。

# 名称

SPI_getnspname — 返回指定关系的名字空间

# 大纲

char * SPI_getnspname(Relation *rel*)

# 描述

SPI_getnspname返回指定关系所属的名字空间的名称拷贝。这等效于该关系的模式。当用完这个函数的返回值后，应该调用pfree释放它。

# 参数

Relation *rel*

输入关系

# 返回值

指定关系的名字空间的名称。

## 名称

SPI_result_code_string — 将错误代码作为字符串返回

## 大纲

const char * SPI_result_code_string(int *code*);

## 简介

SPI_result_code_string返回之前各种SPI函数返回的或者存储在SPI_result中的结果代码的字符串表示。

## 参数

int *code*

结果代码

## 返回值

结果代码的字符串表示。

# 10.3. 内存管理

UXsinoDB在内存上下文中分配内存，内存上下文为管理在多个不同位置、具有不同生存时间需要的分配提供了一种便捷的方法。 销毁一个上下文会释放所有在其中分配的内存。因此不必跟踪单个对象来避免内存泄露，而是只需要管理数量相对较少的上下文即可。palloc和相关的函数可以从“当前”上下文中分配内存。

SPI_connect创建一个新的内存上下文并且让它成为当前上下文。SPI_finish恢复之前的当前上下文并且销毁由SPI_connect创建的内存上下文。这些动作确保在C函数中分配的内存在C函数退出时被回收，从而避免内存泄露。

不过，如果C函数需要返回一个在已分配内存中的对象（例如一个传引用数据类型的值），不能使用palloc分配内存，或者说至少不能在连接到 SPI 时这样做。如果试着这样做，该对象会被SPI_finish接触分配，那么C函数将无法可靠地工作。要解决这个问题，应使用 SPI_palloc来为要返回的对象分配内存。SPI_palloc会在“上层执行器上下文”中分配内存，也就是当SPI_connect被调用时的当前内存上下文，它才是从C函数中返回的值最适合的上下文。这一节中描述的几个其他实用函数也会返回在上层执行器上下文中创建的对象。

当SPI_connect被调用时，这个C函数的私有上下文（由SPI_connect）会被作为当前上下文。所有用palloc、 repalloc或者 SPI 功能函数（除了这个小节中描述的例外）分配的内存都在这个上下文中。当一个C函数从SPI管理器断开连接时（通过SPI_finish），当前上下文被恢复到上层的执行器上下文，并且在该过程的内存上下文中分配的内存都会被释放，之后再不能被使用。

# 名称

SPI_palloc — 在上层执行器上下文中分配内存

# 大纲

void * SPI_palloc(Size *size*)

# 描述

SPI_palloc在上层的执行器上下文中分配内存。

这个函数只能在连接到SPI时使用。否则，它会报出错误。

# 参数

Size *size*

要分配的存储空间大小（以字节计）

# 返回值

指向具有指定大小的新存储空间的指针

# 名称

SPI_repalloc — 在上层执行器上下文中重分配内存

# 大纲

void * SPI_repalloc(void * *pointer*, Size *size*)

# 描述

SPI_repalloc改变之前用SPI_palloc 分配的内存段的大小。

这个函数不再和普通的repalloc相区别。 保留它只是为了对现有代码保持向后兼容。

# 参数

void * *pointer*

指向要改变的现有存储空间的指针

Size *size*

要分配的存储空间大小（以字节计）

# 返回值

指向具有指定大小的新存储空间的指针，现有区域的内容会被复制到其中

# 名称

SPI_pfree — 在上层执行器上下文中释放内存

# 大纲

void SPI_pfree(void * *pointer*)

# 描述

SPI_pfree释放之前使用 SPI_palloc或者 SPI_repalloc分配的内存。

这个函数不再和普通的pfree相区别。 保留它只是为了对现有代码保持向后兼容。

# 参数

void * *pointer*

指向要释放的现有存储空间的指针

# 名称

SPI_copytuple — 在上层执行器上下文中创建一行的拷贝

# 大纲

HeapTuple SPI_copytuple(HeapTuple *row*)

# 描述

SPI_copytuple在上层执行器上下文中为一行创建一份拷贝。这通常被用来从一个触发器中返回一个被修改的行。在一个被声明为返回组合类型的函数中，应使用 SPI_returntuple。

这个函数只能在连接到SPI时使用。否则，它会返回NULL并且把SPI_result设置为SPI_ERROR_UNCONNECTED。

# 参数

HeapTuple *row*

要拷贝的行

# 返回值

被拷贝的行，或者在出错时返回NULL（错误的内容请参见SPI_result）

# 名称

SPI_returntuple — 准备把一个元组返回为一个 Datum

# 大纲

HeapTupleHeader SPI_returntuple(HeapTuple *row*, TupleDesc *rowdesc*)

# 描述

SPI_returntuple为一个行在上层执行器上下文中创建一个拷贝，把它以一种行类型Datum的形式返回。被返回的指针只需要在返回前通过PointerGetDatum被转换成Datum。

这个函数只能在连接到SPI时使用。否则，它会返回NULL并且把SPI_result设置为SPI_ERROR_UNCONNECTED。

注意这应该被用于声明为要返回组合类型的函数。它不能用于触发器，在触发器中应使用SPI_copytuple来返回一个被修改的行。

# 参数

HeapTuple *row*

要被拷贝的行

TupleDesc *rowdesc*

行的描述符（对大部分有效的缓存，每次都传递相同的描述符）

# 返回值

指向被拷贝行的HeapTupleHeader，或者在出错时返回NULL（错误的内容请参见SPI_result）

# 名称

SPI_modifytuple — 通过替换一个给定行的选定域来创建一行

# 大纲

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
                          int * colnum, Datum * values, const char * nulls)
```

# 描述

SPI_modifytuple创建一个新行，其中选定的列用新值替代，其他列则从输入行中拷贝。输入行本身不被修改。新行被返回在上层的执行器上下文中。

这个函数只能在连接到SPI时使用。否则，它会返回NULL并且把SPI_result设置为SPI_ERROR_UNCONNECTED。

# 参数

Relation *rel*

只被用作该行的行描述符的来源（传递一个关系而不是　一个行描述符是一种令人头痛的设计）。

HeapTuple *row*

要被修改的行

int *ncols*

要被修改的列数

int * *colnum*

一个长度为*ncols*的数组，包含了要被修改的列号 （列号从 1 开始）

Datum * *values*

一个长度为*ncols*的数组，包含了指定列的新值

const char * *nulls*

一个长度为*ncols*的数组，描述哪些新值为空值

如果*nulls*为NULL，那么SPI_modifytuple假定没有新值为空值。否则，如果对应的新值为非空，*nulls*数组的每一项都应该是' '，而如果对应的新值为空值则为'n'（在后一种情况中，对应的*values*项中的新值无关紧要）。注意*nulls*不是一个文本字符串，只是一个数组：它不需要一个'\0'终止符。

# 返回值

修改过的新行，它被分配在上层的执行器上下文中，或者在出错时返回NULL（错误的内容请参见SPI_result）

出错时，SPI_result被设置如下：

SPI_ERROR_ARGUMENT

如果*rel*为NULL，或者 *row*为NULL，或者*ncols* 小于等于 0，或者*colnum*为NULL， 或者*values*为NULL。

SPI_ERROR_NOATTRIBUTE

如果*colnum*包含一个无效的列号（小于等于 0 或者大于 *row*中的列数）。

SPI_ERROR_UNCONNECTED

如果SPI不是活跃状态

# 名称

SPI_freetuple — 释放一个在上层执行器上下文中分配的行

# 大纲

void SPI_freetuple(HeapTuple *row*)

# 描述

SPI_freetuple释放之前在上层执行器上下文中分配的一个行。

这个函数不再和普通的heap_freetuple相区别。保留它只是为了对现有代码保持向后兼容。

# 参数

HeapTuple *row*

要释放的行

# 名称

SPI_freetuptable — 释放一个由SPI_execute 或者类似函数创建的行集合

# 大纲

void SPI_freetuptable(SPITupleTable * *tuptable*)

# 描述

SPI_freetuptable释放一个由之前的 SPI 命令执行函数（例如SPI_execute）创建的行集合。因此，调用这个函数时，常常使用SPI_tuptable作为参数。

如果一个使用SPI的C函数需要执行多个命令并且不想保留早期命令的结果，这个函数就有用了。注意，SPI_finish会释放任何还未释放的行集合。还有，如果在一个使用SPI的C函数的执行中开始了一个子事务并且后来被中止，SPI 会自动释放该子事务运行期间创建的任何行集合。

SPI_freetuptable包含了保护逻辑以避免对于同一行集的重复删除请求。在以前的发布中，重复的删除将会导致崩溃。

# 参数

SPITupleTable * *tuptable*

要释放的行集的指针，NULL 表示什么也不做

## 名称

SPI_freeplan — 释放一个之前保存的预备语句

## 大纲

int SPI_freeplan(SPIPlanPtr *plan*)

## 描述

SPI_freeplan释放一个之前由 SPI_prepare返回的或者由 SPI_keepplan、SPI_saveplan 保存的预备语句。

## 参数

SPIPlanPtr *plan*

要释放的语句的指针

## 返回值

成功返回 0；如果*plan*为 NULL或无效则返回 SPI_ERROR_ARGUMENT

# 10.4. 事务管理

不能通过SPI_execute这样的SPI函数运行**COMMIT**和**ROLLBACK**之类的事务控制命令。不过，也有单独的接口函数允许通过SPI进行事务控制。

如果不考虑被调用的上下文，在任意的用户定义的可从SQL调用的函数中开始以及结束事务通常并不是安全和明智的。例如，一个事务位于一个函数内，而该函数是某个SQL命令中的一个复杂SQL表达式的一部分，这样的事务有可能会导致隐蔽的内部错误或者崩溃。这里介绍的接口函数的主要目的是被过程语言的实现用于支持在**CALL**命令调用的SQL层过程中进行事务管理，同时把**CALL**调用的上下文也加以考虑。用C实现的使用SPI的过程可以实现同样的逻辑，但是其细节超出了这份文档的范围。

# 名称

SPI_commit, SPI_commit_and_chain — 提交当前事务

# 大纲

void SPI_commit(void)

void SPI_commit_and_chain(void)

# 简介

SPI_commit提交当前事务。它近似等效于运行SQL命令**COMMIT**。在一个事务被提交后，在进一步的数据库动作被执行前必须使用SPI_start_transaction开始一个新的事务。

SPI_commit_and_chain是相同的，事务特征与刚刚完成的事务相同的新事务会立即启动，类似 SQL 命令 **COMMIT AND CHAIN**。

只有当SPI连接已经在对SPI_connect_ext的调用中被设置为非原子的情况下才能执行这些函数。

# 名称

SPI_rollback, SPI_rollback_and_chain — 终止当前事务

# 大纲

void SPI_rollback(void)

void SPI_rollback_and_chain(void)

# 简介

SPI_rollback回滚当前事务。它近似等效于运行SQL命令**ROLLBACK**。在一个事务被回滚后，在进一步的数据库动作被执行前必须使用SPI_start_transaction开始一个新的事务。

SPI_rollback_and_chain是相同的，事务特征与刚刚完成的事务相同的新事务会立即启动，类似 SQL 命令 **ROLLBACK AND CHAIN**。

只有当SPI连接已经在对SPI_connect_ext的调用中被设置为非原子的情况下才能执行这些函数。

## 名称

SPI_start_transaction — 开始一个新事务

## 大纲

void SPI_start_transaction(void)

## 简介

SPI_start_transaction开始一个新事务。它只能在SPI_commit或SPI_rollback之后被调用，因为在那时没有事务在活动。通常，当一个使用SPI的过程被调用时，会有一个事务已处于活跃状态，因此在关闭当前事务之前尝试启动另一个事务将会导致错误。

只有当SPI连接已经在对SPI_connect_ext的调用中被设置为非原子的情况下才能执行这个函数。

# 10.5. 数据改变的可见性

下列规则主导了使用 SPI 的函数（或者任何其他 C 函数）中数据改变 的可见性：

- 在一个 SQL 命令的执行期间，该命令所作的任何数据更改对该命令本身是不可见的。例如，在如下命令中，被插入的行对SELECT部分不可见。

  INSERT INTO a SELECT * FROM a;

- 一个命令 C 所作的更改对所有在 C 之后开始的命令可见，不管它们是否在 C 之中（在 C 的执行期间）开始还是在 C 完成之后开始。

- 在一个 SQL 命令（或者一个普通函数或者触发器）调用的函数内通过SPI 执行的命令遵循以上哪条规则取决于传递给 SPI 的读/写标志。以只读模式执行的命令遵循第一条规则：它们不能看到调用它们的命令的改变。在读写模式中执行的命令遵循第二条规则：它们能看见目前为止所有的改变。

- 所有的标准过程语言会基于函数的易变性属性设置 SPI 读写模式。STABLE和IMMUTABLE函数的命令会以只读模式完成，而VOLATILE函数的命令会以读写模式完成。虽然 C 函数的作者可以违反这种习惯，但是最好不要那样做。

# 10.6. 示例

这一节包含了 SPI 用法的一个非常简单的示例。C函数execq用一个 SQL 命令作为其第一个参数并且用一个行计数作为第二个参数，使用SPI_exec执行该命令并且返回被该该命令处理过的行的数量。可以在源代码树的src/test/regress/regress.c和spi模块中找到 SPI 的更复杂的示例。

```
#include "uxdb.h"

#include "executor/spi.h"
#include "utils/builtins.h"

UX_MODULE_MAGIC;
```

```
UX_FUNCTION_INFO_V1(execq);

Datum
execq(UX_FUNCTION_ARGS)
{
    char *command;
    int cnt;
    int ret;
    uint64 proc;

    /* 把给定的文本对象转换成一个 C 字符串 */
    command = text_to_cstring(UX_GETARG_TEXT_PP(0));
    cnt = UX_GETARG_INT32(1);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;

    /*
     * 如果取出了一些行，通过 elog(INFO) 打印它们。
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        uint64 j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];
            int i;

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen(buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    UX_RETURN_INT64(proc);
}
```

在把该函数编译到一个共享库中（详见第 1.10.5 节 “编译和链接动态载入的函数”）之后，如下所示，声明该函数。

```
CREATE FUNCTION execq(text, integer) RETURNS int8
   AS 'filename'
   LANGUAGE C STRICT;
```

如下所示，是一个会话实例。

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- inserted by execq
INFO:  EXECQ:  1    -- returned by execq and inserted by upper INSERT

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2    -- 0 + 2，按照所指定的，只有一行被插入

 execq
-------
     3           -- 10 只是最大值，3 是实际的行数
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1              -- 没有行在 a (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  1
INSERT 0 1
=> SELECT * FROM a;
```

```
 x
---
 1
 2              -- 有一行在 in a + 1
(2 rows)

-- 这证明了数据改变可见性规则。

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2              -- 2 行 * 1 （第一行中的 x）
 6              -- 3 rows (2 + 1 被插入) * 2 （第二行中的 x）
(4 rows)            ^^^^^^
                  不同调用中 execq() 的行可见性
```

# 第 11 章 后台工作者进程

UXsinoDB可以被扩展来在独立进程中运行用户提供的代码。这种进程被uxdb启动、停止和监控，这使它们的生命期与服务器的状态紧密联系。这些进程具有选项可以挂接上UXsinoDB的共享内存区域，并且可以从内部连接到数据库。它们也可以连续地运行多个事务，就像一个正常的被客户端连接的服务器进程。同样，通过链接到libpq，它们可以连接到服务器并像一个正常客户端应用工作。

> **警告**
>
> 在使用后台工作者进程时具有相当大的鲁棒性和安全性风险，因为它们由C语言编写，对数据具有无限制的访问权。希望使用包括后台工作者进程在内的模块的管理员必须要极度小心。只有仔细审计过的模块才会被允许运行后台工作者进程。

通过将模块名放在shared_preload_libraries中，可以在UXsinoDB被启动时初始化后台工作者。一个希望运行后台工作者的模块需要通过在其_UX_init()中调用RegisterBackgroundWorker(BackgroundWorker *worker)来注册它。也可以在系统启动后通过调用函数RegisterDynamicBackgroundWorker(BackgroundWorker *worker, BackgroundWorkerHandle **handle)来启动后台工作者。与只能在uxmaster内调用的RegisterBackgroundWorker不同，必须从一个常规后端调用RegisterDynamicBackgroundWorker。

```
typedef void (*bgworker_main_type)(Datum main_arg);
typedef struct BackgroundWorker
{
   char       bgw_name[BGW_MAXLEN];
   char       bgw_type[BGW_MAXLEN];
   int        bgw_flags;
   BgWorkerStartTime bgw_start_time;
   int        bgw_restart_time;      /* in seconds, or BGW_NEVER_RESTART */
   char       bgw_library_name[BGW_MAXLEN];
   char       bgw_function_name[BGW_MAXLEN];
   Datum      bgw_main_arg;
   char       bgw_extra[BGW_EXTRALEN];
   int        bgw_notify_pid;
} BackgroundWorker;
```

*bgw_name*和*bgw_type*是要被用在日志消息、进程列表和类似环境中的字符串。对于同种类型的所有后台工作者，*bgw_type*应该相同，例如这样才可能将进程列表中的这些工作组分组。另一方面，*bgw_name*可以包含有关特定进程的额外信息（通常，*bgw_name*中的字符串在某种程度上也会包含类型，但是并没有严格的要求）。

*bgw_flags*是一个按位与的位掩码，它用于指示模块想要的能力。可能的值如下所示。

BGWORKER_SHMEM_ACCESS

请求共享内存访问。没有共享内存使用权的工作者不能访问任何的UXsinoDB共享数据结构，例如重量级或者轻量级锁、共享缓冲区以及该工作者本身想要创建和使用的任何自定义数据结构。

BGWORKER_BACKEND_DATABASE_CONNECTION

请求建立数据库连接的能力，这样它后面可以通过建立起的连接运行事务和查询。一个使用BGWORKER_BACKEND_DATABASE_CONNECTION来连接一个数据库的后台工作者也必须使用BGWORKER_SHMEM_ACCESS挂接到共享内存，否则工作者启动将会失败。

*bgw_start_time*是服务器状态，在该状态中uxdb会启动该进程，它可以是BgWorkerStart_UxmasterStart（在uxdb本身完成初始化之后立即启动，这种进程不能使用数据库连接）、BgWorkerStart_ConsistentState（当一个热后备中达到一个一致性状态之后立即启动，允许进程连接到数据库并运行只读查询）和BgWorkerStart_RecoveryFinished（在系统进入到正常读写状态后立即启动）之一。注意后两种值在服务器不是一个热后备的情况下是等同的。注意这种设置仅仅表示何时启动进程，当一个不同状态到达时它们不会停止。

*bgw_restart_time*是在崩溃情况下uxdb启动进程之前等待的时间间隔，以秒计。它可以是任何正值，或者BGW_NEVER_RESTART，表示在出现崩溃后不重启进程。

*bgw_library_name*是应该在其中定位后台工作者初始入口点的库名称。所指的库将被工作者进程动态载入并且*bgw_function_name*将被用来标识要调用的函数。如果从核心代码载入一个函数，这必须被设置为"uxdb"。

*bgw_function_name*是一个动态载入库中的一个函数名，该函数将被用作一个新后台工作者的初始入口点。

*bgw_main_arg*是后台工作者主函数的Datum参数。这个主函数应该有一个单一的Datum类型的参数，并且返回void。*bgw_main_arg*将被作为参数传递。此外，全局变量MyBgworkerEntry指向注册时传入的BackgroundWorker结构的一份拷贝，工作者会发现检查这个结构会很有用。

在 Windows （以及任何定义了EXEC_BACKEND的地方）上或者动态后台工作者中，用引用的方式传递Datum是不安全的，只有传值才安全。如果要求一个参数，最安全的方式是传递一个 int32 或者其他的小型值，并且把它当做共享内存中分配的一个数组的索引来使用。如果被传递的是一个cstring或者text这样的值，那么在新的后台工作者进程中该指针将不会有效。

*bgw_extra*可以包含要传递给后台工作者的额外数据。与*bgw_main_arg*不同，这个数据不会被作为一个参数传递给工作者的主函数，而是按照上面所述通过MyBgworkerEntry来访问。

*bgw_notify_pid*是一个UXsinoDB后端进程的PID， 当后台工作者进程启动或者退出时，uxmaster会向这个PID所指的进程发送SIGUSR1。 对于在uxmaster启动时注册的工作者，它应该为0；或者注册该工作者的后端不希望等待该工作者启动时，它也应该为0。否则，它应该被初始化为MyProcPid。

一旦运行起来，进程可以通过调用BackgroundWorkerInitializeConnection(*char *dbname*, *char *username*)或者BackgroundWorkerInitializeConnectionByOid(*Oid dboid*, *Oid useroid*)来连接到一个数据库。这使得该进程可以使用SPI接口运行事务和查询。 如果dbname为NULL或者dboid为InvalidOid，该会话没有连接到任何特定数据库，但共享的目录可以被访问。 如果username为NULL或者useroid为InvalidOid，该进程将以在initdb阶段创建的超级用户身份运行。如果BGWORKER_BYPASS_ALLOWCONN被指定为flags，就可以绕过该限制连接不允许用户连接的数据库。在每一个后台进程中，只能调用两者之一，并且只能调用一次，所以不可能切换数据库。

当控制到达后台工作者的主函数时，信号初始会被阻塞，并且必须被它解除阻塞。这是为了允许进程自定义它的信号处理器。在新进程中可以通过调用BackgroundWorkerUnblockSignals来解除对信号的阻塞，还可以通过调用BackgroundWorkerBlockSignals来阻塞信号。

如果一个后台工作者的*bgw_restart_time*被配置为BGW_NEVER_RESTART，或者它退出时的退出码为0，又或者它是被TerminateBackgroundWorker所终止，它将会被uxmaster在退出时自动解除

注册。否则，它将在等待通过*bgw_restart_time*配置的时间段之后被重新启动，或者在uxmaster因为一次后端失败重新初始化集簇时立刻被重启。需要临时禁止执行的后端应该使用可中断的休眠而不是退出，这可以通过调用WaitLatch()实现。调用该函数时要确保WL_UXMASTER_DEATH标志被设置，并且验证在uxdb本身被终止的紧急情况下产生的快速退出返回码。

当一个后台工作者是通过RegisterDynamicBackgroundWorker函数注册时，后端可以执行该注册以获得有关该工作者的状态信息。希望这样做的后端应该把一个BackgroundWorkerHandle *的地址作为第二个参数传递给RegisterDynamicBackgroundWorker。如果工作者被成功地注册，这个指针将被用一个非透明句柄初始化，它之后会被传递给GetBackgroundWorkerPid(*BackgroundWorkerHandle* *, *pid_t* *)或者TerminateBackgroundWorker(*BackgroundWorkerHandle* *)。GetBackgroundWorkerPid可以被用来测试工作者的状态：返回值为BGWH_NOT_YET_STARTED表示该工作者还未被uxmaster启动；BGWH_STOPPED表示它已经被启动但是不再运行；而BGWH_STARTED表示它正在运行。在最后一种情况下，PID也将被通过第二个参数返回。TerminateBackgroundWorker导致uxmaster发送SIGTERM给工作者（如果它在运行），并且在它不再运行时尽快解除注册。

在某些情况下，一个注册后台工作者的进程可能希望等待该工作者启动起来。其实现方式是：把*bgw_notify_pid*初始化成MyProcPid并且接着把注册时得到的BackgroundWorkerHandle *传递给WaitForBackgroundWorkerStartup(*BackgroundWorkerHandle*handle*, *pid_t* *)函数。这个函数将阻塞直到uxmaster已经尝试启动该后台工作者，或者直到uxmaster死亡。如果后台工作者正在运行，返回值将是BGWH_STARTED，并且其PID将被写入到所提供的地址。否则，返回值将是BGWH_STOPPED或者BGWH_UXMASTER_DIED。

进程也可以等待一个后台工作者关闭，方法是使用WaitForBackgroundWorkerShutdown(*BackgroundWorkerHandle *handle*)函数并且传入注册时得到的BackgroundWorkerHandle *。这个函数将阻塞直至后台工作者退出或者uxmaster死掉。当后台工作者退出时，返回值是BGWH_STOPPED，如果uxmaster死掉则会返回BGWH_UXMASTER_DIED。

如果一个后台工作者通过服务器编程接口（SPI）用NOTIFY命令发送异步通知，在提交外层事务之后它应该显式地调用ProcessCompletedNotifies，这样通知才能被发送出去。如果一个后台工作者通过SPI使用LISTEN注册为接收异步通知，它将记录那些通知，但是对于工作者来说没有程序化的方式可以拦截以及响应那些通知。

src/test/modules/worker_spi模块包含了一个实例，它展示了一些有用的技巧。

注册的后台工作者的最大数量由max_worker_processes限制。

# 第 12 章 逻辑解码

UXsinoDB 提供了方法将所执行的修改通过 SQL 以流的方式传送给外部消费者。这种功能可以被用于多种目的，包括复制方案以及审计。

在流中被送出的更改通过逻辑复制槽标识。

流式传输这些更改的格式由使用的输出插件决定。UXsinoDB 发布中包括了一个示例插件。可以编写额外的插件来扩展可用的格式选择，而无需修改任何核心代码。每一个输出插件都能访问每一个由INSERT产生的新行以及每一个由UPDATE创建的新行版本。UPDATE和DELETE的旧行版本的可用性取决于配置的复制标识（请参见REPLICA IDENTITY）。

可以通过流复制协议（第 12.3 节 “流复制协议接口”）或者通过 SQL 调用函数（第 12.4 节 “逻辑解码的 SQL 接口”）来接收流式传送的更改。也可以编写额外的 接收复制槽输出的模块而无需修改核心代码（第 12.7 节 “逻辑解码输出写入器”）。

## 12.1. 逻辑解码的示例

下面的示例演示了使用 SQL 接口控制逻辑解码。

在使用逻辑解码之前，必须设置wal_level为logical，并且max_replication_slots 必须至少被设置为1。然后，应该作为一个超级用户连接到目标数据库（在下面的示例中是uxdb）。

```
uxdb=# -- 使用输出插件’test_decoding’创建一个名为’regression_slot’的槽
uxdb=# SELECT * FROM ux_create_logical_replication_slot('regression_slot', 'test_decoding');
   slot_name    |    lsn
----------------+-----------
 regression_slot | 0/16B1970
(1 row)

uxdb=# SELECT slot_name, plugin, slot_type, database, active, restart_lsn, confirmed_flush_lsn FROM
 ux_replication_slots;
   slot_name    |    plugin     | slot_type | database | active | restart_lsn | confirmed_flush_lsn
----------------+---------------+-----------+----------+--------+-------------+-----------------
 regression_slot | test_decoding | logical   | uxdb | f      | 0/16A4408   | 0/16A4440
(1 row)

uxdb=# -- 目前还看不到更改
uxdb=# SELECT * FROM ux_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)

uxdb=# CREATE TABLE data(id serial primary key, data text);
CREATE TABLE

uxdb=# -- DDL 没有被复制，因此看到的东西只有事务
uxdb=# SELECT * FROM ux_logical_slot_get_changes('regression_slot', NULL, NULL);
   lsn    | xid |     data
-----------+-------+--------------
 0/BA2DA58 | 10297 | BEGIN 10297
 0/BA5A5A0 | 10297 | COMMIT 10297
```

```
(2 rows)

uxdb=# -- 一单读到更改，它们会被消费掉并且不会在一个后续调用中被发出
uxdb=# SELECT * FROM ux_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)

uxdb=# BEGIN;
uxdb=# INSERT INTO data(data) VALUES('1');
uxdb=# INSERT INTO data(data) VALUES('2');
uxdb=# COMMIT;

uxdb=# SELECT * FROM ux_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    | xid |                       data
-----------+-------+------------------------------------------------------------
 0/BA5A688 | 10298 | BEGIN 10298
 0/BA5A6F0 | 10298 | table public.data: INSERT: id[integer]:1 data[text]:'1'
 0/BA5A7F8 | 10298 | table public.data: INSERT: id[integer]:2 data[text]:'2'
 0/BA5A8A8 | 10298 | COMMIT 10298
(4 rows)

uxdb=# INSERT INTO data(data) VALUES('3');

uxdb=# -- 可以不消费更改而在更改流中先看一看
uxdb=# SELECT * FROM ux_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn    | xid |                       data
-----------+-------+------------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

uxdb=# -- 接下来对 ux_logical_slot_peek_changes() 的调用再次返回相同的更改
uxdb=# SELECT * FROM ux_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn    | xid |                       data
-----------+-------+------------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

uxdb=# -- 可以向输出插件传递选项来影响格式化
uxdb=# SELECT * FROM ux_logical_slot_peek_changes('regression_slot', NULL, NULL, 'include-
timestamp', 'on');
    lsn    | xid |                       data
-----------+-------+------------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299 (at 2017-05-10 12:07:21.272494-04)
(3 rows)

uxdb=# -- 当不再需要一个槽后记住销毁它以停止消耗服务器资源
uxdb=# SELECT ux_drop_replication_slot('regression_slot');
```

```
 ux_drop_replication_slot
----------------------

(1 row)
```

下面的示例展示了如何在流复制协议上使用 UXsinoDB 发布所包括的程序ux_recvlogical来控制逻辑解码。这要求设置客户端认证以允许复制连接，并且把max_wal_senders设置成足够高以允许一个额外的连接。

```
$ ux_recvlogical -d uxdb --slot=test --create-slot
$ ux_recvlogical -d uxdb --slot=test --start -f -
Control+Z
$ uxsql -d uxdb -c "INSERT INTO data(data) VALUES('4');"
$ fg
BEGIN 693
table public.data: INSERT: id[integer]:4 data[text]:'4'
COMMIT 693
Control+C
$ ux_recvlogical -d uxdb --slot=test --drop-slot
```

# 12.2. 逻辑解码概念

## 12.2.1. 逻辑解码

逻辑解码是一种将对数据库表的所有持久更改抽取到一种清晰、易于理解的格式 的处理，这种技术允许在不了解数据库内部状态的详细知识的前提下解释该格式。

在UXsinoDB中，逻辑解码通过解码预写式日志的内容来实现，预写式日志描述了存储 层面上的更改，而逻辑解码则会把更改解码成一种应用相关的形式，例如一个元组 流或者 SQL 语句流。

## 12.2.2. 复制槽

在逻辑复制的环境下，一个槽表示一个更改流，这些更改可以在客户机上以它们在原服务器上产生的顺序被重播。每一个流从一个单一数据库中流式传送更改序列。

> **注意**
>
> UXsinoDB也有流复制槽，但是它们的使用有所不同。

一个复制槽在一个UXsinoDB集簇的所 有数据库之间具有一个唯一的标识符。槽在使用它们的连接之间保持独立并且 对于崩溃是安全的。

在常规操作中，一个逻辑槽只会把每次更改发出一次。只有在检查点时才会持久化每一个槽的当前位置，因此如果发生崩溃，槽可能会回到一个较早的 LSN，这会导致服务器重启时重新发送最近的更改。逻辑解码客户端负责避免多次处理同一消息导致的副作用。客户端可能会希望在解码时记录它们看到的最新的 LSN，并且跳过任何从该 LSN 解码得到的重复数据或者（使用复制协议时的）请求，而不是让服务器来决定开始点。复制进度跟踪特性就是为此服务的，请参见复制源头。

对于同一个数据库可能会存在多个独立的槽。每一个槽有自己的状态，允许不同的消费者从该数据库的更改流中的不同点开始接收更改。对于大多数应用，每一个消费者都将要求一个单独的槽。

逻辑复制槽完全不知道接收者的状态。甚至可能会有多个不同的接收者在不同 时间使用同一个槽，它们将只是从上一个接收者停止消费更改的地方开始得到 更改。但在任一给定时刻，只有一个接收者可以从一个槽中消费更改。

> **注意**
>
> 复制槽可以在崩溃时保持，并且不知道其消费者的状态。即便没有连接使用它们，它们也将阻止移除所需的资源。这会消耗存储，因为只要还有一个复制槽需要，WAL 和来自于系统目录的行就不能被VACUUM移除。在极端情况下这会导致数据库关闭以防止事务ID回卷。因此如果不再需要一个槽，那就应该删除它。

### 12.2.3. 输出插件

输出插件将数据从预写式日志的内部表示转换成复制槽的消费者所需的格式。

### 12.2.4. 导出快照

当使用流复制接口创建一个新的复制槽时（请参见CREATE_REPLICATION_SLOT），一个快照将被导出，在它所显示的数据库状态之后所有的更改都将被包括在更改流中。通过使用SET TRANSACTION SNAPSHOT读取槽被创建时的数据库状态，这可以用来创建一个新的复制。然后这个事务可以被用来及时转储那一点的数据库状态，它后来可以被槽的内容更新而不丢失任何更改。

并非总能够创建快照。特别是在连接到热备时，快照创建将会失败。不要求快照导出的应用可以用NOEXPORT_SNAPSHOT选项来抑制它。

## 12.3. 流复制协议接口

命令如下所示。

- CREATE_REPLICATION_SLOT *slot_name* LOGICAL *output_plugin*
- DROP_REPLICATION_SLOT *slot_name* [ WAIT ]
- START_REPLICATION SLOT *slot_name* LOGICAL ...

被用来创建、删除以及流式传送一个复制槽。这些命令只能在一个复制连接上使用。它们不同通过 SQL 使用。

命令ux_recvlogical可以被用来控制一个流复制连接上的逻辑解码（它在内部使用上述命令）。

## 12.4. 逻辑解码的 SQL 接口

与逻辑解码互动的 SQL 层 API 参见《优炫数据库参考手册 V2.1》中的“函数和操作符章节”的Replication Functions。

同步复制只在使用流复制接口的复制槽上支持。函数接口以及额外的、非核心的接口不支持同步复制。

# 12.5. 与逻辑解码相关的系统目录

ux_replication_slots视图和ux_stat_replication视图分别提供了有关复制槽和流复制连接的当前状态的信息。这些视图适用于物理和逻辑复制。

# 12.6. 逻辑解码输出插件

可以在 UXsinoDB 源码树的contrib/test_decoding子目录中找到一个输出插件的示例。

## 12.6.1. 初始化函数

一个输出插件是通过动态载入一个以输出插件名称作为基础名称的共享库来载入的。将使用普通的库搜索路径来定位该库。为了提供所要求的输出插件回调并且指示该库确实是一个输出插件，需要提供一个名为_UX_output_plugin_init的函数。这个函数会被传入一个结构，其中被填充了各个动作的回调函数指针。

```
typedef struct OutputPluginCallbacks
{
   LogicalDecodeStartupCB startup_cb;
   LogicalDecodeBeginCB begin_cb;
   LogicalDecodeChangeCB change_cb;
   LogicalDecodeTruncateCB truncate_cb;
   LogicalDecodeCommitCB commit_cb;
   LogicalDecodeMessageCB message_cb;
   LogicalDecodeFilterByOriginCB filter_by_origin_cb;
   LogicalDecodeShutdownCB shutdown_cb;
} OutputPluginCallbacks;

typedef void (*LogicalOutputPluginInit) (struct OutputPluginCallbacks *cb);
```

回调函数begin_cb、change_cb以及commit_cb是必需的，
而startup_cb、filter_by_origin_cb、truncate_cb和shutdown_cb是可选的。如果没有设置truncate_cb但是要对一个TRUNCATE进行编码，则该动作将被忽略。

## 12.6.2. 能力

要解码、格式化并且输出更改，输出插件可以使用大部分后端的标准功能，包括调用输出函数。只要访问的关系是initdb在ux_catalog模式中创建的或者被使用，如下所示。

```
ALTER TABLE user_catalog_table SET (user_catalog_table = true);
CREATE TABLE another_catalog_table(data text) WITH (user_catalog_table = true);
```

标记为用户提供的系统表，就允许对关系的只读访问。任何导致事务 ID 分配的动作 都被禁止。其中包括写表、执行 DDL 更改以及调用txid_current()。

## 12.6.3. 输出模式

输出插件回调可以以近乎任意格式向消费者传递数据。对于某些用例，例如通过 SQL 查看更改，以可能包含任何数据的数据类型（例如bytea）返回数据可能会很麻烦。如果输出插件只输

出服务器编码的文本数据，它可以在启动回调中通过把OutputPluginOptions.output_type设置为OUTPUT_PLUGIN_TEXTUAL_OUTPUT替代OUTPUT_PLUGIN_BINARY_OUTPUT来声明这一点。在这种情况下，所有的数据必须是属于服务器的编码，这样一个text数据就能包含它。在启用了断言的编译中会检查这一点。

## 12.6.4. 输出插件回调

一个输出插件需要提供一些回调，它通过它们得到有关更改发生的通知。

并发事务以提交顺序被解码，并且只有属于特定事务的更改会在 begin和commit回调之间被解码。被显式 或隐式回滚的事务不会被解码。成功的检查点被折叠到包含它们的事务中，并且 保持它们在该事务中被执行的顺序。

> **注意**
>
> 只有已经被安全地刷入磁盘的事务将会被解码。当 synchronous_commit被设置为off 时，这会导致一个COMMIT在随后的ux_logical_slot_get_changes()中不会立即被解码。

### 12.6.4.1. 启动回调

只要一个复制槽被创建或者被要求流式传送更改，可选的 startup_cb回调就会被调用，不管有多少更改准备输出。

```
typedef void (*LogicalDecodeStartupCB) (struct LogicalDecodingContext *ctx,
                                        OutputPluginOptions *options,
                                        bool is_init);
```

当复制槽被创建时，is_init参数将为真，否则为假。*options*指向一个输出插件可以设置的选项的结构，如下所示。

```
typedef struct OutputPluginOptions
{
   OutputPluginOutputType output_type;
   bool       receive_rewrites;
} OutputPluginOptions;
```

output_type必须被设置为OUTPUT_PLUGIN_TEXTUAL_OUTPUT或者OUTPUT_PLUGIN_BINARY_OUTPUT。参见第 12.6.3 节 “输出模式”。如果receive_rewrites为真，还将为在某些DDL操作期间的堆重写造成的更改调用输出插件。这些是处理DDL复制的插件感兴趣的事情，但是它们要求特殊的处理。

启动回调应该验证出现在ctx->output_plugin_options中的选项。如果输出插件需要有一个状态，它可以使用ctx->output_plugin_private来存储。

### 12.6.4.2. 关闭回调

只要一个之前活跃的复制槽不再使用，就会调用可选的shutdown_cb回调，它可以被用来释放输出插件私有的资源。该槽并不一定需要被删除，只要其中的流被停止即可。

```
typedef void (*LogicalDecodeShutdownCB) (struct LogicalDecodingContext *ctx);
```

#### 12.6.4.3. 事务开始回调

只要一个已提交事务的开始动作被解码，就会调用必须提供的begin_cb回调。被中止的事务及其内容不会被解码。

```
typedef void (*LogicalDecodeBeginCB) (struct LogicalDecodingContext *ctx,
                                      ReorderBufferTXN *txn);
```

*txn*参数包含有关该事务的元信息，例如该事务被提交的时间戳以及该事务的 XID。

#### 12.6.4.4. 事务结束回调

只要一个已提交事务的提交动作被解码，就会调用必须提供的 commit_cb回调。在此之前，如果有任何被修改 的行，将为所有被修改的行调用change_cb回调。

```
typedef void (*LogicalDecodeCommitCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       XLogRecPtr commit_lsn);
```

#### 12.6.4.5. 更改回调

对于一个事务中的每一个行修改，都将调用必须提供的 change_cb回调，这种修改可能是一个INSERT、UPDATE或者 DELETE。即使原始命令一次修改了多行，该回调也会 为其中的每一行调用一次。

```
typedef void (*LogicalDecodeChangeCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       Relation relation,
                                       ReorderBufferChange *change);
```

*ctx*和*txn*参数与 begin_cb和commit_cb 回调具有相同的内容，但是额外多出一个关系描述符*relation*指向该行所属的关系以及一个结构 *change*描述被传入的行修改。

> **注意**
>
> 只有没有被标记为“不做日志”（UNLOGGED）并且非临时（TEMPORARY or TEMP）的用户定义表中的 更改才能用逻辑解码抽取。

#### 12.6.4.6. 截断回调

truncate_cb回调会为一个TRUNCATE命令被调用。

```
typedef void (*LogicalDecodeTruncateCB) (struct LogicalDecodingContext *ctx,
                                         ReorderBufferTXN *txn,
                                         int nrelations,
                                         Relation relations[],
                                         ReorderBufferChange *change);
```

参数类似于change_cb回调。不过，由于通过外键连接起来的表上的TRUNCATE动作需要一起被执行，这个回调会接收到一个关系的数组而不是单个关系。详情请见对TRUNCATE语句的介绍。

#### 12.6.4.7. 源过滤器回调

可选的filter_by_origin_cb回调被用来决定从*origin_id*重放的数据是否是输出插件感兴趣的数据。

```
typedef bool (*LogicalDecodeFilterByOriginCB) (struct LogicalDecodingContext *ctx,
                                               RepOriginId origin_id);
```

*ctx*参数具有和其他回调相同的内容。对这个回调只有复制源的信息可用。要标志传进来的节点上发生的更改是无关的，返回真，这会导致这些更改被过滤掉，否则返回假。 对于被过滤掉的事务和更改将不会调用其他回调。

在实现级联或者多向复制方案时，这个回调可以派上用场。用源头过滤允许阻止在这样的设置下来回地复制同样的更改。虽然事务和更改也携带了有关源头的信息，通过这个回调过滤明显更有效些。

#### 12.6.4.8. 通用消息回调

只要一个逻辑解码消息被解码出来，可选的message_cb回调就会被调用。

```
typedef void (*LogicalDecodeMessageCB) (struct LogicalDecodingContext *ctx,
                                        ReorderBufferTXN *txn,
                                        XLogRecPtr message_lsn,
                                        bool transactional,
                                        const char *prefix,
                                        Size message_size,
                                        const char *message);
```

*txn*参数包含关于该事务的元信息，如被提交的时间戳和 XID。不过要注意，当消息是非事务性的并且记录该消息的事务中还没有被分配 XID 时，这个参数可以为 NULL。*lsn*是该消息的 WAL 位置。*transactional*说明该消息是否为事务性的。*prefix*是一个任意的空终结的前缀，它当前插件被用来标识感兴趣的消息。最后的*message*参数保存着大小为*message_size*的消息。

应该格外小心确保输出插件用于标识感兴趣消息的前缀是唯一的。建议使用扩展或者输出插件本身的名称。

### 12.6.5. 用于产生输出的函数

在begin_cb、commit_cb或者 change_cb回调中，为了实际产生输出， 输出插件可以把数据写入到ctx->out中的 StringInfo输出缓冲区中。在写出到输出缓冲区之前，必须先 调用OutputPluginPrepareWrite(ctx, last_write)，在完 成写入到缓冲区后，必须调用OutputPluginWrite(ctx, last_write)来执行写出。 *last_write*指出一次特定的写出是否为该回调的最后 一次写出。

下面的示例展示了如何把数据输出给一个输出插件的消费者。

```
OutputPluginPrepareWrite(ctx, true);
appendStringInfo(ctx->out, "BEGIN %u", txn->xid);
OutputPluginWrite(ctx, true);
```

# 12.7. 逻辑解码输出写入器

可以为逻辑解码增加更多输出方法。详情可见src/backend/replication/logical/logicalfuncs.c。 本质上，需要提供三个函数：一个读取 WAL，一个准备写输出，另一个写输出（参见 第 12.6.5 节“用于产生输出的函数”）。

# 12.8. 逻辑解码的同步复制支持

逻辑解码可以被用来构建同步复制方案，该方案具有和流复制的同步复制相同的用户接口。要这样做，流复制接口（见第 12.3 节 “流复制协议接口”）必须被用来流式传出数据。正如流复制客户端所作的一样，逻辑解码的客户端必须发出后备机状态更新 (F)消息。

> **注意**
>
> 一个通过逻辑解码接收更改的同步复制机将工作在一个单一数据库的范围内。因为与之相反，*synchronous_standby_names*当前是服务器范围的，这意味着如果有多于一个数据库被活跃地使用，这种技术将无法正常工作。

# 第 13 章 复制进度追踪

复制源是为了更容易地在逻辑解码上实现逻辑复制解决方案而设计。它们提供了对两种常见问题的解决方案，如下所示。

- 如何安全地跟踪复制进度？
- 如何基于一行的来源更改复制行为？例如，阻止双向复制设置中的循环。

复制源只有两个属性，名称和 OID。名称应该可以被用来在系统间引用该源， 它是一种自由形式的文本。为了避免复制源之间的冲突，可以 在复制源的名称前加上复制解决方案的名称。在空间效率很重要的情况下，OID 被用来避免不得不存储长版本。OID 不能在系统间被共享。

可以使用函数ux_replication_origin_create()创建复制源，使用函数ux_replication_origin_drop()删除复制源，并且在系统目录 ux_replication_origin中查看复制源。

构建一套复制解决方案的一个重要部分是以一种安全的方式跟踪重放进度。当应用 过程或者整个集簇死掉时，需要能够找出数据被成功地复制到了什么地方。对此的 简单解决方案（例如为每一个被重放的事务更新一个表行）有运行时负荷和数据库 膨胀的问题。

通过使用复制源，一个会话可以被标记为从一个远程节点重放（使用ux_replication_origin_session_setup()函数）。此外，可以使用ux_replication_origin_xact_setup()以每一个事务为基础配置每一个源事务的LSN和提交时间戳。如果完成这样的配置，复制过程将保持在一种对崩溃安全的方式中。所有复制源的重放进度可以在ux_replication_origin_status视图中看到。一个源的进度（例如在继续复制时）可以使用ux_replication_origin_progress()（用于任何源）或者ux_replication_origin_session_progress()（用于在当前会话中配置的源）获得。

在比从一个系统复制到另一个系统更复杂的复制拓扑中，另一个问题是很难避免再次复制已经被重放的行。这可能导致复制中的循环和低效。复制源提供了一种可选的机制来识别和阻止这种问题。在使用前一段提到的函数配置时，每一个被传递给输出插件回调（见第 12.6 节 “逻辑解码输出插件”）的由该会话生成的改变和事务会被标记上该会话的复制源。这使得可以在输出插件中以不同的方式对待它们，例如忽略除本地生成的行之外的所有行。此外，filter_by_origin_cb回调可以被用来基于来源过滤逻辑解码改变流。虽然灵活性较低，通过这种回调进行过滤比用输出插件过滤效率更高。