编写存储过程

在第6章 "优化查询以获得良好性能 "中，我们了解了很多关于优化器以及系统中正在进行的优化。 本章将介绍存储过程，以及如何有效和方便地使用它们。你将了解到存储过程是由什么组成的，哪些语言可以使用，以及你如何能够很好地加速事情。除此之外，你还会被介绍到PL/pgSQL的一些更高级的功能。本章将涵盖以下主题。

- 理解存储过程语言
- 探索各种存储过程语言
- 改进函数
- 为各种目的使用函数

在本章结束时，你将能够编写良好、高效的程序

1 理解存储过程语言

当涉及到存储过程和函数时，PostgreSQL与其他数据库系统有相当大的不同。大多数数据库引擎强迫你使用某种编程语言来编写服务器端代码。Microsoft SQL Server提供Transact-SQL，而Oracle鼓励你使用PL/SQL。PostgreSQL不强迫你使用某种语言；相反，它允许你决定你最熟悉和喜欢的语言。

PostgreSQL之所以如此灵活，从历史意义上讲，其实也很有意思。许多年前，最著名的PostgreSQL开发者之一，Jan Wieck，在PostgreSQL的早期就写了无数的补丁，提出了使用工具命令语言（Tcl）作为服务器端编程语言的想法。问题是，没有人愿意使用Tcl，也没有人愿意在数据库引擎中使用这些东西。解决这个问题的办法是使语言界面非常灵活，基本上任何语言都可以很容易地与PostgreSQL集成。在这一点上，CREATE LANGUAGE子句诞生了。下面是CREATE LANGUAGE的语法。

```
test=# \h CREATE LANGUAGE
Command: CREATE LANGUAGE
Description: define a new procedural language
Syntax:
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
 HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ]
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
URL: https://www.postgresql.org/docs/13/sql-createlanguage.html
```

现在，许多不同的语言都可以用来编写函数和存储过程。为PostgreSQL增加的灵活性确实得到了回报；我们现在可以从丰富的编程语言中选择。

PostgreSQL到底是如何处理语言的？如果我们看一下CREATE LANGUAGE子句的语法，我们会看到一些关键字。

- HANDLER：这个函数实际上是PostgreSQL和你想使用的任何外部语言之间的粘合剂。它负责将PostgreSQL的数据结构映射到语言所需要的东西上，并帮助传递代码。
- VALIDATOR：这是基础设施的警察。如果它是可用的，它将负责向终端用户提供可口的语法错误。许多语言能够在实际执行代码之前解析它。PostgreSQL可以利用这一点，在你创建一个函数的时候告诉你这个函数是否正确。不幸的是，不是所有的语言都能做到这一点，所以在某些情况下，你仍然会在运行时出现问题。
- INLINE：如果存在这个，PostgreSQL将能够利用这个处理函数来运行匿名代码块。

1.1 了解存储过程与函数的基本原理

在我们深入了解存储过程的结构之前，有必要先谈谈一般的函数和过程。传统上，存储过程这一术语实际上是用来谈论一个函数的。因此，我们必须了解函数和存储过程之间的区别。

一个函数是普通SQL语句的一部分，不允许启动或提交事务。下面是一个例子。

```
SELECT func(id) FROM large_table;
```

假设func(id)被调用了5000万次。如果你使用名为Commit的函数，到底应该发生什么？不可能简单地在查询过程中结束一个事务并启动一个新的事务。整个交易的完整性、一致性等概念都会被违反。

相反，一个存储过程能够控制事务，甚至能够一个接一个地运行多个事务。但是，你不能在一个SELECT语句中运行它。相反，你必须调用CALL。下面的列表显示了CALL命令的语法。

```
test=# \h CALL
Command: CALL
Description: invoke a procedure
Syntax:
CALL name ( [ argument ] [, ...] )
URL: https://www.postgresql.org/docs/13/sql-call.html
```

因此，功能和程序之间有一个基本的区别。你在互联网上找到的术语并不总是很清楚。然而，你必须意识到这些重要的区别。在PostgreSQL中，函数从一开始就已经存在了。 然而，本节所概述的过程的概念是新的，在PostgreSQL 11中才被引入。在本章中，我们将详细介绍函数和过程。让我们从了解函数的结构开始。

1.2 函数的剖析

在我们研究具体的语言之前，们将看一下典型函数的剖析。出于演示的目的，让我们看看下面这个函数，它只是将两个数字相加。

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int)
RETURNS int AS
'
 SELECT $1 + $2;
' LANGUAGE 'sql';
CREATE FUNCTION
```

首先要注意的是，这个函数是用SQL写的。 PostgreSQL需要知道我们使用的是哪种语言，所以我们必须在定义中指定。

请注意，函数的代码是以字符串（'）的形式传递给PostgreSQL的。 这是值得注意的，因为它允许一个函数成为一个执行机器的黑匣子。

在其他数据库引擎中，函数的代码不是一个字符串，而是直接连接到语句中。这个简单的抽象层给了PostgreSQL函数管理器所有的力量。在字符串里面，你基本上可以使用你所选择的编程语言所能提供的一切。

在这个例子中，我们将简单地把传递给函数的两个数字相加。有两个整数变量在使用。这里重要的部分是，PostgreSQL为你提供了函数重载。换句话说，mysum(int, int)函数与mysum(int8, int8)函数是不一样的。

PostgreSQL认为这些东西是两个不同的函数。函数重载是一个很好的功能；但是，如果你的参数列表恰好时常变化，你必须非常小心，不要意外地部署太多的函数。一定要确保不再需要的函数真的被删除。

CREATE OR REPLACE FUNCTION子句不会改变参数列表。因此，只有在签名不改变的情况下，你才能使用它。它要么会出错，要么会简单地部署一个新函数。 让我们运行mysum函数。

```
test=# SELECT mysum(10, 20);
 mysum
-------
 30
(1 row)
```

这里的结果是30，这其实并不令人惊讶。在介绍完这些函数之后，重要的是要关注下一个主要话题：引用。

1.3 介绍美元引号

将代码作为字符串传递给PostgreSQL是非常灵活的。然而，使用单引号可能是一个问题。在许多编程语言中，单引号经常出现。为了能够使用这些引号，人们必须在把字符串传递给PostgreSQL时转义它们。许多年来，这一直是标准程序。幸运的是，那些旧时代已经过去了，现在有新的手段可以将代码传递给PostgreSQL。其中之一是美元引号，如下面的代码所示。

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int)
RETURNS int AS
$$
 SELECT $1 + $2;
$$ LANGUAGE 'sql';
CREATE FUNCTION
```

你可以不使用引号来开始和结束字符串，而直接使用$$。 目前，有两种语言已经为$$赋予了意义。在Perl以及Bash脚本中，$代表进程ID。为了克服这个小障碍，我们几乎可以在任何东西前面使用$来开始和结束字符串。下面的例子说明了这一方法的作用。

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int) RETURNS int AS
$body$
 SELECT $1 + $2;
$body$ LANGUAGE 'sql';
CREATE FUNCTION
```

所有这些灵活性使你能够一劳永逸地克服引号的问题。只要开始字符串和结束字符串相匹配，就完全不会有任何问题。

1.4 使用匿名代码块

到目前为止，我们已经编写了尽可能简单的存储过程，也学会了如何执行代码。然而，代码的执行不仅仅是完整的函数。除了函数之外，PostgreSQL还允许使用匿名代码块。这个想法是为了运行只需要一次的代码。这种代码执行在处理管理任务时特别有用。匿名代码块不接受参数，也不会永久地保存在数据库中，因为它们没有名字。

下面是一个简单的例子，显示了一个匿名代码块的运行情况。

```
test=# DO
$$
 BEGIN
 RAISE NOTICE 'current time: %', now();
 END;
$$ LANGUAGE 'plpgsql';
NOTICE: current time: 2020-09-01 09:52:39.279219+02
CONTEXT: PL/pgSQL function inline_code_block line 3 at RAISE
DO
```

在这个例子中，代码只发出了一条信息，然后就退出了。同样，代码块必须知道它使用哪种语言。这个字符串被传递给PostgreSQL，使用简单的美元引号

1.5 使用函数和事务

如你所知，PostgreSQL在用户区暴露的所有东西都是一个事务。当然，如果你正在编写函数，也是如此。一个函数总是你所处的事务的一部分。它不是独立的，就像一个操作者或任何其他操作。

这是一个例子：

```
test=# SELECT now(), mysum(id, id) FROM generate_series(1, 3) AS id;
 now | mysum
-------------------------------+-------
 2020-09-01 09:52:55.966338+02 | 2
 2020-09-01 09:52:55.966338+02 | 4
 2020-09-01 09:52:55.966338+02 | 6
(3 rows)
```

所有三个函数调用都发生在同一个事务中。这一点很重要，因为它意味着你不能在一个函数内做太多的事务性流程控制。当第二个函数调用提交时会发生什么？它就是不能工作。

然而，Oracle有一种机制，允许自主交易。 这个想法是，即使一个事务回滚，有些部分可能仍然需要，它们应该被保留。一个经典的例子是这样的。

1. 启动一个查找秘密数据的功能。
2. 在文件中添加一个日志行，说明有人修改了这个重要的秘密数据。
3. 提交日志行，但回滚修改。
4. 保存信息，说明有人试图改变数据。

为了解决类似这样的问题，可以使用自主事务。 这个想法是为了能够在主事务中独立提交一个事务。在这种情况下，日志表中的条目将占上风，而变化将被回滚。

从PostgreSQL 11.0开始，自主事务并没有实现。 然而，已经有一些补丁在流传，以实现这一功能。这些功能什么时候能进入核心版本还有待观察。

为了给你一个印象，让你知道事情很可能是如何运作的，这里有一个基于第一批补丁的代码片段。

```
...
AS
$$
DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
 FOR i IN 0..9 LOOP
 START TRANSACTION;
 INSERT INTO test1 VALUES (i);
 IF i % 2 = 0 THEN
 COMMIT;
 ELSE
 ROLLBACK;
 END IF;
 END LOOP;
 RETURN 42;
END;
$$;
...
```

这个例子的重点是告诉你，我们可以决定是提交还是回滚自主事务。

2 探索各种存储过程语言

正如我们在本章中已经说过的，PostgreSQL让你有能力用各种语言编写函数和存储过程。以下是可用的选项，并与PostgreSQL核心一起提供。

- SQL PL/pgSQL
- PL/Perl and PL/PerlU
- PL/Python
- PL/Tcl and PL/TclU

SQL是编写函数的明显选择，应该尽可能地使用它，因为它给了优化器最大的自由。然而，如果你想写稍微复杂一点的代码，PL/pgSQL可能是你的选择语言。

PL/pgSQL提供了流程控制和更多的功能。在这一章中，我们将展示PL/pgSQL的一些更高级和不为人知的特性，但请记住，这一章并不意味着是PL/pgSQL的完整教程。

核心部分包含在Perl中运行服务器端函数的代码。基本上，这里的逻辑是一样的。代码将以字符串的形式传递并由Perl执行。 请记住，PostgreSQL不会说Perl，它只是有代码将东西传给外部编程语言。

也许你已经注意到Perl和Tcl有两种风格：可信的（PL/Perl和PL/Tcl）和不可信的（PL/PerlU和PL/TclU）。受信任和不受信任的语言之间的区别实际上是一个重要的区别。在PostgreSQL中，一种语言被直接加载到数据库连接中。因此，该语言能够做相当多的讨厌的事情。为了摆脱安全问题，人们发明了可信语言的概念。这个想法是，可信语言被限制在语言的核心部分。它不可能做以下事情。

- 包括库
- 打开网络套接字
- 执行任何类型的系统调用，这将包括打开文件

Perl提供了一种叫做污点模式的东西，它被用来在PostgreSQL中实现这一功能。Perl会自动将自己限制在信任模式下，如果即将发生安全漏洞，就会出错。在非信任模式下，一切皆有可能，因此只有超级用户可以运行非信任的代码。

如果你想运行受信任以及不受信任的代码，你必须激活两种语言，即plperl和plperlu（分别是pltcl和pltclu）。

目前，Python 只能作为一种不受信任的语言使用；因此，当涉及到一般的安全问题时，管理员必须非常小心，因为在不受信任模式下运行的函数可以绕过所有由 PostgreSQL 执行的安全机制。只要记住，Python 是作为你的数据库连接的一部分来运行的，而绝不是对安全负责。

让我们开始讨论本章中最令人期待的话题。

2.1 介绍 PL/pgSQL

在这一节中，你将被介绍到PL/pgSQL的一些更高级的特性，这些特性对于编写正确和高效的代码非常重要。 请注意，这不是对编程的初学者介绍，也不是对PL/pgSQL的一般介绍。

2.1.1 处理引号和字符串格式

数据库编程中最重要的事情之一是引号。如果你不使用正确的引号，你肯定会陷入SQL注入的麻烦中，并打开不可接受的安全漏洞。

什么是SQL注入？让我们考虑下面的例子。

```
CREATE FUNCTION broken(text) RETURNS void AS
$$
DECLARE
 v_sql text;
BEGIN
 v_sql := 'SELECT schemaname
 FROM pg_tables
 WHERE tablename = ''' || $1 || '''';
 RAISE NOTICE 'v_sql: %', v_sql;
 RETURN;
END;
$$ LANGUAGE 'plpgsql';
```

在这个例子中，SQL代码被简单地粘贴在一起，而不必担心安全问题。我们在这里所做的就是使用||操作符来连接字符串。如果人们运行正常的查询，这就很好。考虑一下下面的例子，显示了一些破损的代码。

```
SELECT broken('t_test');
```

然而，我们必须准备好应对那些试图利用你的系统的人。考虑一下下面的例子。

```
SELECT broken('''; DROP TABLE t_test; ');
```

带着这个参数运行函数会显示一个问题。下面的代码显示了经典的SQL注入。

```
NOTICE: v_sql: SELECT schemaname FROM pg_tables
WHERE tablename = ''; DROP TABLE t_test; '
CONTEXT: PL/pgSQL function broken(text) line 6 at RAISE
 broken
--------
(1 row)
```

当你只想做一个查询时，删除一个表并不是一件理想的事情。让你的应用程序的安全取决于传递给你的语句的参数，这绝对是不可接受的。 为了避免SQL注入，PostgreSQL提供了各种函数；这些函数应该随时使用，以确保你的安全保持不变。

```
test=# SELECT quote_literal(E'o''reilly'), quote_ident(E'o''reilly');
 quote_literal | quote_ident
----------------+-------------
 'o''reilly' | "o'reilly" (1 row)
```

quote_literal函数将以这样的方式转义一个字符串，即不会再发生任何坏事。它将在字符串周围加上所有的引号，并转义字符串中有问题的字符。因此，不需要手动开始和结束字符串。 这里显示的第二个函数是quote_ident。它可以用来正确引用对象名称。注意，使用的是双引号，这正是处理表名所需要的。下面的例子显示了如何使用复杂的名称。

```
test=# CREATE TABLE "Some stupid name" ("ID" int);
CREATE TABLE
test=# \d "Some stupid name"
 Table "public.Some stupid name"
 Column | Type | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 ID | integer | | |
```

通常情况下，PostgreSQL中所有的表名都是小写的。然而，如果使用了双引号，对象名称可以包含大写字母。一般来说，使用这种技巧并不是一个好主意，因为你将不得不一直使用双引号，这可能有点不方便。

现在你已经有了一个关于引号的基本介绍，重要的是看一下NULL值是如何处理的。下面的代码显示了quote_literal函数是如何处理NULL的。

```
test=# SELECT quote_literal(NULL);
 quote_literal
---------------
 (1 row)
```

如果你在一个NULL值上调用quote_literal函数，它将直接返回NULL。在这种情况下，没有必要照顾到引号。 PostgreSQL提供了更多的函数来明确地处理NULL值。

```
test=# SELECT quote_nullable(123), quote_nullable(NULL);
 quote_nullable | quote_nullable
-----------------+----------------
 '123' | NULL (1 row)
```

它不仅可以引用字符串和对象名称；还可以使用PL/pgSQL上的格式化和准备整个查询。这里的美妙之处在于，你可以使用格式化函数来向语句中添加参数。 下面是它的工作原理。

```
CREATE FUNCTION simple_format() RETURNS text AS
$$
DECLARE
 v_string text;
 v_result text;
BEGIN
 v_string := format('SELECT schemaname|| '' .'' || tablename
 FROM pg_tables
 WHERE %I = $1
 AND %I = $2', 'schemaname', 'tablename');
 EXECUTE v_string USING 'public', 't_test' INTO v_result;
 RAISE NOTICE 'result: %', v_result;
 RETURN v_string;
END;
$$ LANGUAGE 'plpgsql';
```

字段的名称被传递给format函数。最后，EXECUTE语句的USING子句是为了将参数添加到查询中，然后执行。同样，这里的好处是不可能发生SQL注入。

下面是调用simple_format函数时发生的情况。

```
test=# SELECT simple_format ();
NOTICE: result: public .t_test
 simple_format
-----------------------------------------------
 SELECT schemaname|| ' .' || tablename +
 FROM pg_tables +
 WHERE schemaname = $1+
 AND tablename = $2
(1 row)
```

正如你所看到的，调试信息正确地显示了表，包括模式，并正确地返回了查询。然而，格式函数可以做得更多。下面是一些例子。

```
test=# SELECT format('Hello, %s %s','PostgreSQL', 13);
 format
----------------------
 Hello, PostgreSQL 13
(1 row)
test=# SELECT format('Hello, %s %10s','PostgreSQL', 13);
 format
----------------------
 Hello, PostgreSQL 13
(1 row)
```

format能够使用格式选项，如例子所示。%10s意味着我们要添加的字符串将被填充。空白会被添加。

在某些情况下，可能需要不止一次地使用一个变量。下面的例子显示了两个参数，它们被不止一次地添加到我们想要创建的字符串中。你可以做的是用1美元、2美元等来识别参数列表中的条目。

```
test=# SELECT format('%1$s, %1$s, %2$s', 'one', 'two');
 format
---------------
 one, one, two
(1 row)
```

format是一个非常强大的功能，当你想避免SQL注入时，这个功能超级重要，你应该好好利用这个强大的功能。

2.1.2 管理作用域

在处理了一般的引用和基本安全（SQL注入）之后，我们将关注另一个重要的话题：作用域。 就像大多数流行的编程语言一样，PL/pgSQL使用变量，这取决于其上下文。变量是在一个函数的DECLARE语句中定义的。然而，PL/pgSQL允许你嵌套一个DECLARE语句。

```
CREATE FUNCTION scope_test () RETURNS int AS
$$
DECLARE
 i int := 0;
BEGIN
 RAISE NOTICE 'i1: %', i;
 DECLARE
 i int;
 BEGIN
 RAISE NOTICE 'i2: %', i;
```

```
 END;
 RETURN i;
END;
$$ LANGUAGE 'plpgsql';
```

在DECLARE语句中，定义了i变量并给它分配了一个值。然后，i被显示出来。然后，第二个DECLARE语句开始。它包含了一个额外的i的化身，它没有被分配一个值。因此，其值将是NULL。注意，PostgreSQL现在将显示内部的i。

```
test=# SELECT scope_test();
NOTICE: i1: 0
NOTICE: i2: <NULL>
 scope_test
------------
 0
(1 row)
```

正如所料，调试信息将显示0和NULL。 PostgreSQL允许你使用各种各样的技巧。然而，我们强烈建议你保持你的代码简单和易读。

2.1.3 了解高级错误处理

对于编程语言，在每个程序和每个模块中，错误处理是一件重要的事情。每件事情都会偶尔出错，因此，正确和专业地处理错误是至关重要的，也是关键性的。在PL/pgSQL中，你可以使用EXCEPTION块来处理错误。我们的想法是，如果BEGIN块做错了什么，EXCEPTION块会处理它，并正确地处理问题。就像许多其他语言，如Java，你可以对不同类型的错误做出反应，并分别捕捉它们。

在下面的例子中，代码可能会遇到一个除以零的问题。我们的目标是抓住这个错误并作出相应的反应。

```
CREATE FUNCTION error_test1(int, int) RETURNS int AS
$$
BEGIN
 RAISE NOTICE 'debug message: % / %', $1, $2;
 BEGIN
 RETURN $1 / $2;
 EXCEPTION
 WHEN division_by_zero THEN
 RAISE NOTICE 'division by zero detected: %', sqlerrm;
 WHEN others THEN
 RAISE NOTICE 'some other error: %', sqlerrm;
 END;
 RAISE NOTICE 'all errors handled';
 RETURN 0;
END;
$$ LANGUAGE 'plpgsql';
```

BEGIN块显然可以抛出一个错误，因为有可能出现除以0的情况。然而，EXCEPTION块抓住了我们正在看的错误，同时也处理了所有其他可能意外出现的潜在问题。

从技术上讲，这或多或少与保存点相同，因此，错误不会导致整个事务完全失败。只有导致错误的区块才会受到小型回滚的影响。

通过检查sqlerrm变量，你也可以直接接触到错误信息本身。让我们运行这段代码。

```
test=# SELECT error_test1(9, 0);
NOTICE: debug message: 9 / 0
NOTICE: division by zero detected: division by zero
NOTICE: all errors handled
 error_test1
-------------
 0
(1 row)
```

PostgreSQL捕获了这个异常，并在EXCEPTION块中显示了这个消息。它还很友好的向我们展示了出错的那一行。这使得调试和修复代码变得更加容易，如果它被破坏的话。

在某些情况下，引发你自己的异常也是有意义的。正如你所期望的，这是很容易做到的。

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

除此之外，PostgreSQL提供了许多预定义的错误代码和异常。下面的页面包含这些错误信息的完整列表：https://www.postgresql.org/docs/13/static/errcodes-appendix.html。

2.1.4 使用 GET DIAGNOSTICS

许多过去使用过Oracle的人可能对GET DIAGNOSTICS子句很熟悉。GET DIAGNOSTICS子句背后的想法是让用户看到系统中正在发生的事情。虽然对于习惯于现代代码的人来说，这种语法可能显得有些奇怪，但它仍然是一个有价值的工具，可以使你的应用程序变得更好。

从我的观点来看，GET DIAGNOSTICS子句有两个主要任务可以使用。

- 检查行数
- 获取上下文信息并获得回溯

检查行数绝对是你在日常编程中需要的。如果你想调试应用程序，提取上下文信息将很有用。

下面的例子显示了如何在你的代码中使用GET DIAGNOSTICS子句。

```
CREATE FUNCTION get_diag() RETURNS int AS
$$
DECLARE
 rc int;
 _sqlstate text;
 _message text;
 _context text;
BEGIN
 EXECUTE 'SELECT * FROM generate_series(1, 10)';
 GET DIAGNOSTICS rc = ROW_COUNT;
 RAISE NOTICE 'row count: %', rc;
 SELECT rc / 0;
EXCEPTION
 WHEN OTHERS THEN
 GET STACKED DIAGNOSTICS
 _sqlstate = returned_sqlstate,
 _message = message_text,
 _context = pg_exception_context;
 RAISE NOTICE 'sqlstate: %, message: %, context: [%]',
 _sqlstate,
 _message,
 replace( _context, E'n', ' <- ' );
```

```
 RETURN rc;
END;
$$ LANGUAGE 'plpgsql';
```

声明这些变量后的第一件事是执行一个SQL语句，并向GET DIAGNOSTICS子句询问行数，然后在调试信息中显示。然后，该函数强制PL/pgSQL出错。一旦发生这种情况，我们将使用GET DIAGNOSTICS子句从服务器上提取信息来显示。

下面是我们调用get_diag函数时的情况。

```
test=# SELECT get_diag();
NOTICE: row count: 10
CONTEXT: PL/pgSQL function get_diag() line 12 at RAISE
NOTICE: sqlstate: 22012,
message: division by zero,
context: [SQL statement "SELECT rc / 0"
<- PL/pgSQL function get_diag() line 14 at
SQL statement]
CONTEXT: PL/pgSQL function get_diag() line 22 at RAISE
 get_diag
----------
 10
(1 row)
```

正如你所看到的，GET DIAGNOSTICS子句给了我们关于系统中活动的详细信息。

2.1.5 使用游标来获取分块的数据

如果你执行SQL，数据库将计算出结果并将其发送给你的应用程序。一旦整个结果集被发送至客户端，应用程序就可以继续做它的工作。问题是这样的：如果结果集大到无法再装入内存，会发生什么？如果数据库返回100亿行怎么办？客户端应用程序通常不能一次处理这么多数据，实际上，它也不应该这样做。解决这个问题的方法是游标。游标背后的想法是，只有在需要时（调用FETCH时）才会产生数据。因此，在数据库实际生成数据的时候，应用程序已经可以开始消耗数据了。除此之外，执行这一操作所需的内存要少得多。

当涉及到PL/pgSQL时，游标也起到了重要作用。每当你在一个结果集上循环时，PostgreSQL会在内部自动使用游标。 这样做的好处是，你的应用程序的内存消耗将大大减少，而且由于处理的数据量很大，几乎不可能出现内存耗尽的情况。使用游标的方式有多种。

下面是一个最简单的例子，在一个函数内使用游标。

```
CREATE OR REPLACE FUNCTION c(int)
 RETURNS setof text AS
$$
DECLARE
 v_rec record;
BEGIN
 FOR v_rec IN SELECT tablename
 FROM pg_tables
 LIMIT $1
 LOOP
 RETURN NEXT v_rec.tablename;
 END LOOP;
 RETURN;
END;
```

```
$$ LANGUAGE 'plpgsql';
```

这段代码很有趣，有两个原因。首先，它是一个集合返回函数（SRF）。它产生了整个列，而不仅仅是一个单行。实现这一目的的方法是使用变量集，而不仅仅是数据类型。 RETURN NEXT子句将建立起结果集，直到我们到达终点。RETURN子句将告诉PostgreSQL我们要离开这个函数，并且我们有了结果。

第二个重要的问题是，在游标上循环将自动创建一个内部游标。换句话说，不需要担心有可能耗尽内存。PostgreSQL将对查询进行优化，试图尽可能快地产生前10%的数据（由cursor_tuple_fraction变量定义）。下面是这个查询将返回的内容。

```
test=# SELECT * FROM c(3);
 c
--------------
 t_test
 pg_statistic
 pg_type
(3 rows)
```

在这个例子中，只是会有一个随机表的列表。如果你那边的结果不一样，这在一定程度上是可以预期的。

在我看来，你刚才看到的是在PL/pgSQL中使用隐式游标的最频繁和最常见的方式。

下面的例子显示了一个较老的机制，许多有Oracle背景的人可能知道这个机制。

```
CREATE OR REPLACE FUNCTION d(int)
 RETURNS setof text AS
$$
DECLARE
 v_cur refcursor;
 v_data text;
BEGIN
 OPEN v_cur FOR
 SELECT tablename
 FROM pg_tables
 LIMIT $1;
 WHILE true LOOP
 FETCH v_cur INTO v_data;
 IF FOUND THEN
 RETURN NEXT v_data;
 ELSE
 RETURN;
 END IF;
 END LOOP;
END;
$$ LANGUAGE 'plpgsql';
```

在这个例子中，游标被明确地声明和打开。在里面，循环数据被显式地获取并返回给调用者。基本上，查询做的是同样的事情。这只是一个关于开发人员实际喜欢什么语法的品味问题。

你还觉得你对游标的了解还不够吗？ 还有，这里有第三个选项，可以做完全相同的事情。

```
CREATE OR REPLACE FUNCTION e(int)
 RETURNS setof text AS
$$
DECLARE
```

```
 v_cur CURSOR (param1 int) FOR
 SELECT tablename
 FROM pg_tables
 LIMIT param1;
 v_data text;
BEGIN
 OPEN v_cur ($1);
 WHILE true LOOP
 FETCH v_cur INTO v_data;
 IF FOUND THEN
 RETURN NEXT v_data;
 ELSE
 RETURN;
 END IF;
 END LOOP;
END;
$$ LANGUAGE 'plpgsql';
```

在这种情况下，游标被输入一个整数参数，这个参数直接来自于函数调用（$1）。 有时，游标并没有被存储过程本身使用，而是被返回供以后使用。在这种情况下，你可以返回一个简单的游标作为返回值。

```
CREATE OR REPLACE FUNCTION cursor_test(c refcursor)
 RETURNS refcursor AS
$$
BEGIN
 OPEN c FOR SELECT *
 FROM generate_series(1, 10) AS id;
 RETURN c;
END;
$$ LANGUAGE plpgsql;
```

这里的逻辑很简单。游标的名称被传递给函数。然后，游标被打开并返回。这里的好处是，游标背后的查询可以在运行中创建，并动态地编译。

应用程序可以像其他应用程序一样从游标中获取信息。下面是它的工作原理。

```
test=# BEGIN;
BEGIN
test=# SELECT cursor_test('mytest');
 cursor_test
-------------
 mytest
(1 row)
 test=# FETCH NEXT FROM mytest;
 id
----
 1
(1 row)
test=# FETCH NEXT FROM mytest;
 id
----
 2
(1 row)
```

请注意，它只在使用事务块时起作用。

在本节中，我们已经了解到游标只会在数据被消耗时产生数据。这对大多数查询来说是正确的。然而，这个例子有一个问题；只要使用SRF，整个结果就必须被物化。它不是即时创建的，而是一次性的。其原因是SQL必须能够重新扫描一个关系，这在普通表的情况下很容易做到。然而，对于函数，情况是不同的。 因此，一个SRF总是被计算和物化，使得这个例子中的游标完全没有用。换句话说，我们在编写函数时需要小心。在某些情况下，危险就隐藏在巧妙的细节中。

2.1.6 使用复合类型

在大多数其他数据库系统中，存储过程只用于原始数据类型，比如整数、数字、varchar等等。 然而，PostgreSQL是非常不同的。我们可以使用所有可用的数据类型。这包括原始的、复合的和自定义的数据类型。就数据类型而言，根本没有任何限制。为了充分释放PostgreSQL的力量，复合类型是非常重要的，并且经常被在互联网上发现的扩展所使用。下面的例子显示了如何将复合类型传递给一个函数，以及如何在内部使用它。最后，复合类型将被再次返回。

```
CREATE TYPE my_cool_type AS (s text, t text);
CREATE FUNCTION f(my_cool_type)
 RETURNS my_cool_type AS
$$
DECLARE
 v_row my_cool_type;
BEGIN
 RAISE NOTICE 'schema: (%) / table: (%)'
 , $1.s, $1.t;
 SELECT schemaname, tablename
 INTO v_row
 FROM pg_tables
 WHERE tablename = trim($1.t)
 AND schemaname = trim($1.s)
 LIMIT 1;
 RETURN v_row;
END;
$$ LANGUAGE 'plpgsql';
```

这里的主要问题是，你可以简单地使用1.field_name来访问复合类型。返回复合类型也不难。

你只需要像其他数据类型一样，临时组装复合类型的变量并返回它。你甚至可以轻松地使用数组，甚至更复杂的结构。

下面的代码显示了PostgreSQL将返回什么。

```
test=# SELECT (f).s, (f).t
 FROM f ('("public", "t_test")'::my_cool_type);
NOTICE: schema: (public) / table: ( t_test)
 s | t
--------+--------
 public | t_test
(1 row)
```

2.1.7 在 PL/pgSQL 中编写触发器

如果你想对数据库中发生的某些事件作出反应，服务器端代码就特别受欢迎。触发器允许你在一个表发生INSERT、UPDATE、DELETE或TRUNCATE子句时调用一个函数。被触发器调用的函数可以修改表中发生变化的数据，或者简单地执行一个必要的操作。

在PostgreSQL中，触发器多年来变得更加强大，它们现在提供了一系列丰富的功能。

```
test=# \h CREATE TRIGGER
Command: CREATE TRIGGER
Description: define a new trigger
Syntax:
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR
... ] }
 ON table_name
 [ FROM referenced_table_name ]
 [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ]
]
 [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ]
]
 [ FOR [ EACH ] { ROW | STATEMENT } ]
 [ WHEN ( condition ) ]
 EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
where event can be one of:
 INSERT
 UPDATE [ OF column_name [, ... ] ]
 DELETE
 TRUNCATE
URL: https://www.postgresql.org/docs/13/sql-createtrigger.html
```

首先要注意的是，触发器总是为一个表或一个视图触发，并调用一个函数。它有一个名字，可以在一个事件之前或之后发生。PostgreSQL的魅力在于，你可以在一个表中有无数个触发器。虽然这对PostgreSQL的铁杆用户来说并不奇怪，但我想指出，这在许多昂贵的商业数据库引擎中是不可能的，这些引擎目前仍在世界各地使用。 如果同一个表有多个触发器，那么多年前在PostgreSQL 7.3中引入的以下规则将是有用的：触发器是按字母顺序触发。首先，所有这些BEFORE触发器都是按字母顺序发生的。然后，PostgreSQL执行触发器被触发的行操作，并继续按字母顺序在触发器之后执行。换句话说，触发器的执行顺序是绝对确定的，而且触发器的数量基本上是无限的。

触发器可以在实际修改发生之前或之后修改数据。一般来说，这是一个验证数据的好方法，如果违反了一些自定义限制，就会出错。下面的例子显示了一个在INSERT子句中被触发的触发器，它改变了被添加到表中的数据。

```
CREATE TABLE t_sensor (
 id serial,
 ts timestamp,
 temperature numeric
);
```

我们的表只是存储了几个值。现在的目标是一旦有行插入，就调用一个函数。

```
CREATE OR REPLACE FUNCTION trig_func()
RETURNS trigger AS
$$
 BEGIN
 IF NEW.temperature < -273
 THEN
 NEW.temperature := 0;
 END IF;
 RETURN NEW;
 END;
$$ LANGUAGE 'plpgsql';
```

正如我们之前所说，触发器将总是调用一个函数，这允许你使用漂亮的抽象代码。这里重要的一点是，触发器函数必须返回一个触发器。要访问你要插入的行，你可以访问NEW变量。

INSERT和UPDATE触发器总是提供一个NEW变量。UPDATE和DELETE会提供一个叫做OLD的变量。这些变量包含你将要修改的行。

在我的例子中，代码检查温度是否过低。如果是的话，这个值是不行的，它会被动态地调整。为了确保修改后的行能够被使用，仅仅返回NEW。如果在这个触发器之后还有第二个触发器被调用，那么下一个函数调用将已经看到修改过的行。

在下一步，可以使用CREATE TRIGGER命令来创建触发器。

```
CREATE TRIGGER sensor_trig
 BEFORE INSERT ON t_sensor
 FOR EACH ROW
 EXECUTE PROCEDURE trig_func();
```

这是触发器将执行的操作：

```
test=# INSERT INTO t_sensor (ts, temperature)
 VALUES ('2017-05-04 14:43', -300) RETURNING *;
 id | ts | temperature
-----+---------------------+-------------
 1 | 2017-05-04 14:43:00 | 0
(1 row)
INSERT 0 1
```

正如你所看到的，该值已被正确调整。表中的内容显示温度为0。

如果你正在使用触发器，你应该意识到这样一个事实，即一个触发器对自己了解很多。它可以访问一些变量，这些变量允许你编写更复杂的代码，从而实现更好的抽象性。

让我们先删除触发器。

```
test=# DROP TRIGGER sensor_trig ON t_sensor;
DROP TRIGGER
```

触发器被成功删除。 然后，可以添加一个新的函数。

```
CREATE OR REPLACE FUNCTION trig_demo()
 RETURNS trigger AS
$$
BEGIN
 RAISE NOTICE 'TG_NAME: %', TG_NAME;
 RAISE NOTICE 'TG_RELNAME: %', TG_RELNAME;
 RAISE NOTICE 'TG_TABLE_SCHEMA: %', TG_TABLE_SCHEMA;
 RAISE NOTICE 'TG_TABLE_NAME: %', TG_TABLE_NAME;
 RAISE NOTICE 'TG_WHEN: %', TG_WHEN;
 RAISE NOTICE 'TG_LEVEL: %', TG_LEVEL;
 RAISE NOTICE 'TG_OP: %', TG_OP;
 RAISE NOTICE 'TG_NARGS: %', TG_NARGS;
-- RAISE NOTICE 'TG_ARGV: %', TG_NAME;
 RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER sensor_trig
 BEFORE INSERT ON t_sensor
 FOR EACH ROW
 EXECUTE PROCEDURE trig_demo();
```

这里使用的所有变量都是预定义的，默认是可用的。我们的代码所做的就是显示它们，以便我们可以看到它们的内容。

```
test=# INSERT INTO t_sensor (ts, temperature)
 VALUES ('2017-05-04 14:43', -300) RETURNING *;
NOTICE: TG_NAME: demo_trigger
NOTICE: TG_RELNAME: t_sensor
NOTICE: TG_TABLE_SCHEMA: public
NOTICE: TG_TABLE_NAME: t_sensor
NOTICE: TG_WHEN: BEFORE
NOTICE: TG_LEVEL: ROW
NOTICE: TG_OP: INSERT
NOTICE: TG_NARGS: 0
 id | ts | temperature
-----+---------------------+-------------
 2 | 2017-05-04 14:43:00 | -300
(1 row)
INSERT 0 1
```

我们在这里看到的是，触发器知道它的名字，它被触发的表，以及很多其他的东西。为了在不同的表中应用类似的操作，这些变量有助于避免重复的代码，只需编写一个函数。然后，这可以用于我们感兴趣的所有表。

到目前为止，我们已经看到了简单的行级触发器，它在每个语句中被触发一次。然而，随着PostgreSQL 10.0的推出，有一些新的功能。语句级的触发器已经存在了一段时间了。然而，它不可能访问被触发器改变的数据。在PostgreSQL 10.0中，这个问题已经得到了解决，现在可以利用过渡表，其中包含所有的变化。

下面的代码包含一个完整的例子，显示了如何使用过渡表。

```
CREATE OR REPLACE FUNCTION transition_trigger()
 RETURNS TRIGGER AS $$
 DECLARE
 v_record record;
 BEGIN
 IF (TG_OP = 'INSERT') THEN
 RAISE NOTICE 'new data: ';
 FOR v_record IN SELECT * FROM new_table
 LOOP
 RAISE NOTICE '%', v_record;
 END LOOP;
 ELSE
 RAISE NOTICE 'old data: ';
 FOR v_record IN SELECT * FROM old_table
 LOOP
 RAISE NOTICE '%', v_record;
 END LOOP;
 END IF;
 RETURN NULL; -- result is ignored since this is an AFTER trigger
 END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER transition_test_trigger_ins
 AFTER INSERT ON t_sensor
 REFERENCING NEW TABLE AS new_table
 FOR EACH STATEMENT EXECUTE PROCEDURE transition_trigger();
CREATE TRIGGER transition_test_trigger_del
 AFTER DELETE ON t_sensor
 REFERENCING OLD TABLE AS old_table
 FOR EACH STATEMENT EXECUTE PROCEDURE transition_trigger();
```

在这种情况下，我们需要两个触发器定义，因为我们不能把所有东西都挤到一个定义里。在触发器函数内部，过渡表很容易使用：它可以像普通表一样被访问。

让我们通过插入一些数据来测试一下触发器的代码。

```
INSERT INTO t_sensor
 SELECT *, now(), random() * 20
 FROM generate_series(1, 5);
DELETE FROM t_sensor;
```

在我的例子中，代码将简单地为过渡表中的每个条目发出NOTICE。

```
NOTICE: new data:
NOTICE: (1,"2017-10-04 15:47:14.129151",10.4552665632218)
NOTICE: (2,"2017-10-04 15:47:14.129151",12.8670312650502)
NOTICE: (3,"2017-10-04 15:47:14.129151",14.3934494629502)
NOTICE: (4,"2017-10-04 15:47:14.129151",4.35718866065145)
NOTICE: (5,"2017-10-04 15:47:14.129151",10.9121138229966)
INSERT 0 5
NOTICE: old data:
NOTICE: (1,"2017-10-04 15:47:14.129151",10.4552665632218)
NOTICE: (2,"2017-10-04 15:47:14.129151",12.8670312650502)
NOTICE: (3,"2017-10-04 15:47:14.129151",14.3934494629502)
NOTICE: (4,"2017-10-04 15:47:14.129151",4.35718866065145)
NOTICE: (5,"2017-10-04 15:47:14.129151",10.9121138229966)
DELETE 5
```

请记住，使用数十亿行的过渡表不一定是个好主意。PostgreSQL确实是可扩展的，但在某些时候，有必要看到对性能也有影响。

2.1.8 在 PL/pgSQL 中编写存储过程

现在，让我们继续前进，学习如何编写存储过程。在本节中，你将学习如何编写真正的存储过程，这是PostgreSQL 11中引入的。要创建一个存储过程，你必须使用CREATE PROCEDURE。这个命令的语法与CREATE FUNCTION非常相似。只是有一些细微的差别，可以从下面的语法定义中看到。

```
test=# \h CREATE PROCEDURE
Command: CREATE PROCEDURE
Description: define a new procedure
Syntax:
CREATE [ OR REPLACE ] PROCEDURE
 name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [,
...] ] )
 { LANGUAGE lang_name
 | TRANSFORM { FOR TYPE type_name } [, ... ]
```

```
 | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
 | SET configuration_parameter { TO value | = value | FROM CURRENT }
 | AS 'definition'
 | AS 'obj_file', 'link_symbol'
 } ...
URL: https://www.postgresql.org/docs/13/sql-createprocedure.html
```

下面的例子显示了一个运行两个事务的存储过程。 第一个事务是COMMIT，因此创建了两个表。第二个存储过程是ROLLBACK。

```
test=# CREATE PROCEDURE test_proc()
 LANGUAGE plpgsql
AS $$
 BEGIN
 CREATE TABLE a (aid int); CREATE TABLE b (bid int);
 COMMIT;
 CREATE TABLE c (cid int);
 ROLLBACK;
 END;
$$;
CREATE PROCEDURE
```

正如我们所看到的，存储过程能够进行明确的事务处理。存储过程背后的想法是能够运行批处理工作和其他操作，这在函数中很难做到。

为了运行存储过程，你必须使用CALL，如下面的例子所示。

```
test=# CALL test_proc();
CALL
```

前两个表已经提交由于过程中的回滚，第三个表尚未创建

```
test=# \d
List of relations
 Schema | Name | Type | Owner
--------+------+-------+-------
 public | a | table | hs
 public | b | table | hs
(2 rows)
```

程序是PostgreSQL 11中引入的最重要的功能之一，它们对软件开发的效率做出了重大贡献。

2.2 介绍PL/Perl

关于PL/pgSQL，还有很多要讲的。然而，在一本书中不可能涵盖所有的内容，所以现在是时候转向下一个过程性语言了。PL/Perl已经被很多人采纳，成为字符串处理的理想语言。正如你可能知道的那样，Perl以其字符串操作能力而闻名，因此在这么多年后仍然相当流行。

要启用PL/Perl，你有两个选择。

```
test=# create extension plperl;
CREATE EXTENSION
test=# create extension plperlu;
CREATE EXTENSION
```

你可以部署受信任或不受信任的Perl。如果你想要这两种语言，你必须同时启用这两种语言。

为了向你展示PL/Perl是如何工作的，我实现了一个简单解析电子邮件地址并返回真或假的函数。下面是它的工作原理。

```
test=# CREATE OR REPLACE FUNCTION verify_email(text)
RETURNS boolean AS
$$
if ($_[0] =~ /^[a-z0-9.]+@[a-z0-9.-]+$/)
{
 return true;
}
return false;
$$ LANGUAGE 'plperl';
CREATE FUNCTION
```

一个测试参数被传递给函数。在函数内部，所有这些输入参数都可以用$_访问。在这个例子中，正则表达式被执行，而函数被返回。

该函数可以被调用，就像用其他语言编写的其他过程一样。下面的列表显示了如何调用该函数。

```
test=# SELECT verify_email('hs@cybertec.at');
 verify_email
--------------
 t
(1 row)
test=# SELECT verify_email('totally wrong');
 verify_email
--------------
 f
(1 row)
```

前面的列表显示，该函数正确验证了代码。 请记住，如果你在一个受信任的函数内部，你就不能加载包，等等。例如，如果你想用w命令来查找单词，Perl会在内部加载utf8.pm，当然，这是不允许的。

2.2.1 利用PL/Perl进行数据类型抽象

正如本章已经说过的，PostgreSQL中的函数是相当通用的，可以在许多不同的情况下使用。如果你想使用函数来提高你的数据质量，你可以使用CREATE DOMAIN子句。

```
test=# \h CREATE DOMAIN
Command: CREATE DOMAIN
Description: define a new domain
Syntax:
CREATE DOMAIN name [ AS ] data_type
 [ COLLATE collation ]
 [ DEFAULT expression ]
 [ constraint [ ... ] ]
where constraint is:
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
URL: https://www.postgresql.org/docs/13/sql-createdomain.html
```

在这个例子中，PL/Perl函数将被用来创建一个叫做email的域，而这个域又可以作为一个数据类型使用。 下面的代码显示了如何创建该域。

```
test=# CREATE DOMAIN email AS text
 CHECK (verify_email(VALUE) = true);
CREATE DOMAIN
```

CREATE DOMAIN命令创建了额外的类型，并自动应用检查约束，以确保该限制在整个数据库中被一致使用。

正如我们之前提到的，域的功能就像一个正常的数据类型。

```
test=# CREATE TABLE t_email (id serial, data email);
CREATE TABLE
```

这个Perl函数确保没有任何违反我们检查的东西可以被插入数据库，下面的例子成功地证明了这一点。

```
test=# INSERT INTO t_email (data)
 VALUES ('somewhere@example.com');
INSERT 0 1
test=# INSERT INTO t_email (data)
 VALUES ('somewhere_wrong_example.com');
ERROR: value for domain email violates check constraint "email_check"
```

Perl可能是一个很好的选择，可以进行字符串计算，但是，像往常一样，你必须决定你是否想让这些代码直接进入数据库，或者不直接进入。

2.2.2 在 PL/Perl 和 PL/PerlU 之间做出选择

到目前为止，Perl代码还没有引起任何与安全有关的问题，因为我们所做的只是使用正则表达式。这里的问题是，如果有人试图在Perl函数里面做一些讨厌的事情怎么办？正如我们已经说过的，PL/Perl会简单地出错，你可以在下一个列表中看到。

```
test=# CREATE OR REPLACE FUNCTION test_security()
RETURNS boolean AS
$$
use strict;
my $fp = open("/etc/password", "r");
return false;
$$ LANGUAGE 'plperl';
ERROR: 'open' trapped by operation mask at line
CONTEXT: compilation of PL/Perl function "test_security"
```

该列表显示，PL/Perl在你试图创建函数时就会报错。一个错误会立即显示出来。 如果你真的想在Perl中运行不受信任的代码，你必须使用PL/PerlU。

```
test=# CREATE OR REPLACE FUNCTION first_line()
RETURNS text AS
$$
open(my $fh, '<:encoding(UTF-8)', "/etc/passwd")
 or elog(NOTICE, "Could not open file '$filename' $!");
my $row = <$fh>;
close($fh);
return $row;
$$ LANGUAGE 'plperlu';
CREATE FUNCTION
```

该程序保持不变。它返回一个字符串。然而，它被允许做任何事情。唯一的区别是，该函数被标记为plperlu。

结果有些出乎意料

```
test=# SELECT first_line();
 first_line
---------------------------------
 root:x:0:0:root:/root:/bin/bash+
(1 row)
```

第一行将按预期显示。

2.2.3 使用 SPI 接口

偶尔，你的Perl程序要进行一些数据库工作。 记住，这个函数是数据库连接的一部分。因此，实际创建一个数据库连接是没有意义的。为了与数据库对话，PostgreSQL服务器基础设施提供了服务器编程接口（SPI），这是一个C接口，你可以用它来与数据库内部对话。所有帮助你运行服务器端代码的过程性语言都使用这个接口来向你暴露功能。PL/Perl也是这样做的，在本节中，你将学习如何使用SPI接口周围的Perl包装器。

你可能想做的最重要的事情是简单地运行SQL，并检索出被取走的行数。spi_exec_query函数正是用来做这个的。传递给该函数的第一个参数是查询参数。第二个参数是你实际想要检索的行数。为了简单起见，我决定获取所有的行。下面的代码显示了一个例子。

```
test=# CREATE OR REPLACE FUNCTION spi_sample(int)
RETURNS void AS
$$
my $rv = spi_exec_query(" SELECT *
 FROM generate_series(1, $_[0])", $_[0]
);
elog(NOTICE, "rows fetched: " . $rv->{processed});
elog(NOTICE, "status: " . $rv->{status});
return;
$$ LANGUAGE 'plperl';
```

SPI将执行查询并显示行数。这里重要的是，所有的存储过程语言都提供了一个发送日志信息的方法。在PL/Perl的情况下，这个函数被称为elog，它需要两个参数。第一个参数定义了消息的重要性（INFO, NOTICE, WARNING, ERROR，等等），第二个参数包含了实际的消息。

下面的消息显示了查询返回的内容。

```
test=# SELECT spi_sample(9);
NOTICE: rows fetched: 9
NOTICE: status: SPI_OK_SELECT
 spi_sample
------------
(1 row)
```

对SPI的调用工作得很好，并且显示了状态。

2.2.4 使用 SPI 设置返回函数

在许多情况下，你并不只是想执行一些SQL语句，然后忘记它。 在大多数情况下，一个存储过程会在结果上循环，并对其进行处理。下面的例子将展示你如何在一个查询的输出上进行循环。除此之外，我还决定加强这个例子，让这个函数返回一个复合数据类型。在Perl中使用复合类型是非常容易的，因为你可以简单地将数据塞进一个哈希值并返回。

return_next函数将逐渐建立起结果集，直到函数以一个简单的返回语句结束。

下面代码中的例子生成了一个由随机值组成的表。

```
CREATE TYPE random_type AS (a float8, b float8);
CREATE OR REPLACE FUNCTION spi_srf_perl(int)
 RETURNS setof random_type AS
$$
my $rv = spi_query("SELECT random() AS a,
 random() AS b
 FROM generate_series(1, $_[0])");
while (defined (my $row = spi_fetchrow($rv)))
{
 elog(NOTICE, "data: " .
 $row->{a} . " / " . $row->{b});
 return_next({a => $row->{a},
 b => $row->{b}});
}
return;
$$ LANGUAGE 'plperl';
CREATE FUNCTION
```

首先，spi_query函数被执行，然后使用spi_fetchrow函数启动一个循环。在这个循环中，复合类型将被组装起来并塞进结果集中。 正如预期的那样，该函数将返回一组随机值。

```
test=# SELECT * FROM spi_srf_perl(3);
NOTICE: data: 0.154673356097192 / 0.278830723837018
CONTEXT: PL/Perl function "spi_srf_perl"
NOTICE: data: 0.615888888947666 / 0.632620786316693
CONTEXT: PL/Perl function "spi_srf_perl"
NOTICE: data: 0.910436692181975 / 0.753427186980844
CONTEXT: PL/Perl function "spi_srf_perl"
 a_col | b_col
-------------------+-------------------
 0.154673356097192 | 0.278830723837018
 0.615888888947666 | 0.632620786316693
 0.910436692181975 | 0.753427186980844
(3 rows)
```

请记住，返回集合的函数必须是物化的，以便整个结果集可以被存储在内存中。

2.2.5 在 PL/Perl 中转义和支持函数

到目前为止，我们只使用了整数，所以SQL注入或特殊表名都不是问题。基本上，可以使用以下函数。

- quote_literal: quote_nullable: 返回一个字符串的引用，作为一个字符串字面。引用一个字符串。
- quote_ident: 引述SQL标识符（对象名称等）。
- decode_bytea: 对PostgreSQL的字节数组字段进行解码。
- encode_bytea: 对数据进行编码，并把它变成一个字节数组。
- encode_literal_array: 这是对一个字词阵列的编码。
- encode_typed_literal: 将一个Perl变量转换为作为第二个参数传递的数据类型的值，并返回该值的字符串表示。
- encode_array_constructor: 将引用的数组的内容以数组构造器的格式返回为字符串。
- looks_like_number: 如果一个字符串看起来像一个数字，则返回true。
- is_array_ref: 如果某个东西是一个数组引用，则返回true。

这些函数总是可用的，它们可以被直接调用，而不需要包含一个库。

2.2.6 跨函数调用共享数据

有时，有必要在不同的通话中分享数据。基础设施有办法真正做到这一点。在Perl中，一个哈希值可以用来存储任何需要的数据。看一下下面的例子吧。

```
CREATE FUNCTION perl_shared(text) RETURNS int AS
$$
if ( !defined $_SHARED{$_[0]} )
{
 $_SHARED{$_[0]} = 0;
}
else
{
 $_SHARED{$_[0]}++;
}
return $_SHARED{$_[0]};
$$ LANGUAGE 'plperl';
```

一旦我们发现传给函数的键还不在那里，$_SHARED变量就会被初始化为0。对于其他的调用，1会被添加到计数器中，留给我们的是以下输出。

```
test=# SELECT perl_shared('some_key') FROM generate_series(1, 3);
 perl_shared
-------------
 0
 1
 2
(3 rows)
```

如果是比较复杂的语句，开发者通常不知道函数将以什么顺序被调用。牢记这一点很重要。在大多数情况下，你不能依赖执行顺序。

2.2.7 在 Perl 中编写触发器

每一种与PostgreSQL核心一起提供的存储过程语言都允许你用该语言编写触发器。当然，这也适用于Perl。由于本章篇幅有限，我决定不包括用Perl编写触发器的例子，而是让你看PostgreSQL的官方文档：https://www.postgresql.org/docs/10/static/plperl-triggers.html。基本上，用Perl写一个触发器和用PL/pgSQL写一个触发器没有什么区别。所有预定义的变量都到位了，至于返回值，规则适用于每一种存储过程语言。

2.3 介绍 PL/Python

如果你碰巧不是Perl专家，PL/Python可能是适合你的东西。Python已经成为PostgreSQL基础设施的一部分很长时间了，因此是一个坚实的、经过良好测试的实现。

谈到PL/Python，有一件事你必须牢记。PL/Python只能作为一种不受信任的语言使用。从安全的角度来看，在任何时候都必须记住这一点。

要启用PL/Python，你可以从你的命令行中运行以下一行，并测试你想用PL/Python使用的数据库的名称：

```
createlang plpythonu test
```

一旦语言被启用，就可以写代码了。

另外，你也可以使用CREATE LANGUAGE子句。另外，请记住，为了使用服务器端语言，需要包含这些语言的PostgreSQL包（postgresql-plpython-$(VERSIONNUMBER)，等等）。

2.3.1 编写简单的 PL/Python 代码

在本节中，你将学习如何编写简单的Python程序。我们在这里要讨论的例子很简单：如果你在奥地利开车拜访客户，你可以在每公里的费用中扣除42欧分，以减少你的所得税。因此，该函数所做的是，获取公里数，并返回我们可以从税单中扣除的金额。下面是它的工作原理。

```
CREATE OR REPLACE FUNCTION calculate_deduction(km float)
 RETURNS numeric AS
$$
if km <= 0:
 elog(ERROR, 'invalid number of kilometers')
else:
 return km * 0.42
$$ LANGUAGE 'plpythonu';
```

该函数确保只接受正值。最后，结果被计算并返回。正如你所看到的，Python函数传递给PostgreSQL的方式与Perl或PL/pgSQL并没有什么不同。

2.3.2 使用 SPI 接口

像所有程序性语言一样，PL/Python让你可以访问SPI接口。下面的例子显示了如何将数字加起来。

```
CREATE FUNCTION add_numbers(rows_desired integer)
 RETURNS integer AS
$$
mysum = 0
```

```
cursor = plpy.cursor("SELECT * FROM
 generate_series(1, %d) AS id" % (rows_desired))
while True:
 rows = cursor.fetch(rows_desired)
 if not rows:
 break
 for row in rows:
 mysum += row['id']
return mysum
$$ LANGUAGE 'plpythonu';
```

当你尝试这个例子时，要确保对光标的调用实际上是单行的。Python 是关于缩进的，所以如果你的代码是由一行或两行组成的，确实有区别。

一旦游标被创建，我们就可以在它上面循环，把这些数字加起来。这些行里面的列可以很容易地用列名来引用。

调用该函数将返回所需的结果。

```
test=# SELECT add_numbers(10);
 add_numbers
-------------
 55
(1 row)
```

如果你想检查一个SQL语句的结果集，PL/Python提供了各种函数，允许你从结果中检索更多的信息。同样，这些函数是SPI在C语言层面上提供的包装。 下面的函数更仔细地检查了一个结果。

```
CREATE OR REPLACE FUNCTION result_diag(rows_desired integer)
 RETURNS integer AS
$$
rv = plpy.execute("SELECT *
 FROM generate_series(1, %d) AS id" % (rows_desired))
plpy.notice(rv.nrows())
plpy.notice(rv.status())
plpy.notice(rv.colnames())
plpy.notice(rv.coltypes())
plpy.notice(rv.coltypmods())
plpy.notice(rv. str ())
return 0
$$ LANGUAGE 'plpythonu';
```

nrows()函数将显示行的数量。status()函数告诉我们一切工作是否顺利。colnames()函数返回一个列的列表。coltypes()函数返回结果集中数据类型的对象ID。23是整数的内部数字，如下面的代码所示。

```
test=# SELECT typname FROM pg_type WHERE oid = 23;
 typname
---------
 int4
(1 row)
```

然后是tymod。考虑像varchar(20)这样的东西：类型的配置部分就是tymod的全部内容。 最后，有一个函数将整个东西作为一个字符串返回，用于调试目的。调用该函数将返回以下结果。

```
test=# SELECT result_diag(3);
NOTICE: 3
NOTICE: 5
NOTICE: ['id']
NOTICE: [23]
NOTICE: [-1]
NOTICE: <PLyResult status=5 nrows=3 rows=[{'id': 1},
 {'id': 2}, {'id': 3}]>
result_diag
-------------
 0
(1 row)
```

该清单显示了我们的诊断函数返回的内容。在SPI接口中还有很多函数可以帮助你执行SQL。

2.3.3 处理错误

偶尔，你可能要捕捉一个错误。当然，这在Python中也是可能的。下面的例子显示了这是如何进行的。

```
CREATE OR REPLACE FUNCTION trial_error()
 RETURNS text AS
$$
try:
 rv = plpy.execute("SELECT surely_a_syntax_error")
except plpy.SPIError:
 return "we caught the error" else:
else:
 return "all fine"
$$ LANGUAGE 'plpythonu';
```

你可以使用一个正常的try或except块，并访问plpy来处理你想捕捉的错误。然后该函数可以正常返回而不破坏你的事务，如下所示。

```
test=# SELECT trial_error();
 trial_error
---------------------
 we caught the error
(1 row)
```

记住，PL/Python可以完全访问PostgreSQL的内部结构。 因此，它也可以将各种错误暴露给你的程序。下面是一个例子。

```
except spiexceptions.DivisionByZero:
 return "found a division by zero"
except spiexceptions.UniqueViolation:
 return "found a unique violation"
except plpy.SPIError, e:
 return "other error, SQLSTATE %s" % e.sqlstate
```

这段代码显示了如何捕捉各种Python错误。在Python中捕捉错误真的很容易，而且可以帮助防止你的函数失败。 在这一节中，你已经了解了Python的错误处理。在下一节中，我们将深入研究，看看我们如何帮助优化器。

3 改进函数

到目前为止，你已经看到如何用各种语言编写基本函数和触发器。当然，还有很多语言被支持。其中最突出的是PL/R（R是一个强大的统计包）和PL/v8（它是基于谷歌JavaScript引擎的）。然而，这些语言已经超出了本章的范围（不管它们是否有用）。

在本节中，我们将重点讨论如何提高一个函数的性能。 有几种方法可以让我们加快处理速度。

- 减少函数调用次数
- 使用缓存计划
- 给优化器以提示

在本节中，将讨论所有这三个方面。让我们从减少函数调用的数量开始。

3.1 减少函数调用次数

在很多情况下，性能不好是因为函数被调用得太频繁了。在我看来--我怎么强调这一点都不为过--太频繁地调用东西是性能不好的主要原因。当你创建一个函数时，你可以从三种类型的函数中选择：易变的、稳定的和不可变的。下面是一个例子。

```
test=# SELECT random(), random();
 random | random
--------------------+--------------------
 0.276252629235387 | 0.710661871358752
(1 row)
test=# SELECT now(), now();
 now | now
-------------------------------+-------------------------------
 2020-09-01 09:58:33.153834+02 | 2020-09-01 09:58:33.153834+02
(1 row)
test=# SELECT pi();
 pi
------------------
 3.14159265358979
(1 row)
```

一个易失性函数意味着该函数不能被优化掉。它必须一次又一次地被执行。一个易失性函数也可能是某个索引不被使用的原因。默认情况下，每个函数都被认为是不稳定的。一个稳定的函数将总是在同一个事务中返回相同的数据。它可以被优化，调用也可以被删除。now()函数就是一个稳定函数的好例子；在同一个事务中，它返回相同的数据。不可变的函数是黄金标准，因为它们允许大多数优化，这是因为如果给它们相同的输入，它们总是返回相同的结果。作为优化函数的第一步，一定要确保它们被正确标记，在定义的末尾加上volatile、stable或immutable。

在下一小节中，你将了解到缓存计划。

3.2 使用缓存计划

在PostgreSQL中，一个查询是通过四个阶段来执行的。

1. 解析器。它检查语法。
2. 重写系统。它负责处理规则。
3. 优化器/规划器。这将优化查询。
4. 执行器。执行由计划器提供的计划。

如果查询时间较短，前三个步骤相对于实际执行时间来说是比较耗时的。因此，对执行计划进行缓存是有意义的。PL/pgSQL基本上在幕后为你自动完成所有的计划缓存。你不需要担心这个问题。PL/Perl和PL/Python将为你提供选择。

SPI接口提供了一些函数，这样你就可以处理和运行准备好的查询，所以程序员可以选择是否应该准备好一个查询。在长查询的情况下，使用未准备的查询实际上是有意义的。短的查询应该总是准备好的，以减少内部开销。

3.3 将成本分配给函数

从优化器的角度来看，一个函数基本上就像一个运算符。PostgreSQL也会以相同的方式处理成本，就像它是一个标准运算符一样。问题是这样的：两个数字相加通常比使用PostGIS提供的函数相交成本线要便宜。问题是，优化器不知道一个函数是便宜还是昂贵。

幸运的是，我们可以告诉优化器使函数更便宜或更昂贵。下面是CREATE FUNCTION的语法。

```
test=# \h CREATE FUNCTION
Command: CREATE FUNCTION
Description: Define a new function
Syntax:
CREATE [ OR REPLACE ] FUNCTION
...
| COST execution_cost
| ROWS result_rows
...
```

COST参数表示你的操作符比标准操作符真正昂贵的程度。它是cpu_operator_cost的一个乘数，不是一个静态值。一般来说，默认值是100，除非该函数是用C语言编写的。现在我们已经了解了所有关于函数的知识，让我们在下一节中进一步探讨它们。

4 为各种目的使用函数

在PostgreSQL中，存储过程几乎可以用于一切。 在本章中，你已经了解了CREATE DOMAIN子句等，但也可以创建你自己的操作符、类型转换，甚至排序规则。

在本节中，你将看到如何创建一个简单的类型转换，以及如何利用它来发挥你的优势。要定义一个类型转换，可以考虑看一下CREATE CAST子句。这个命令的语法在下面的代码中显示。

```
test=# \h CREATE CAST
Command: CREATE CAST
Description: define a new cast
Syntax:
CREATE CAST (source_type AS target_type)
 WITH FUNCTION function_name [ (argument_type [, ...]) ]
 [ AS ASSIGNMENT | AS IMPLICIT ]
CREATE CAST (source_type AS target_type)
 WITHOUT FUNCTION
 [ AS ASSIGNMENT | AS IMPLICIT ]
CREATE CAST (source_type AS target_type)
 WITH INOUT
 [ AS ASSIGNMENT | AS IMPLICIT ]
```

```
URL: https://www.postgresql.org/docs/13/sql-createcast.html
```

使用这个东西是非常简单的。你只需告诉PostgreSQL它应该调用哪个程序，以便将任何类型的数据转换为你想要的数据类型。

在标准的PostgreSQL中，你不能把IP地址转为布尔值。 因此，它是一个很好的例子。首先，存储过程必须被定义。

```
CREATE FUNCTION inet_to_boolean(inet)
RETURNS boolean AS
$$
 BEGIN
 RETURN true;
 END;
$$ LANGUAGE 'plpgsql';
```

为了简单起见，它返回真。然而，你可以使用任何语言的任何代码来进行实际转换。

在下一步，已经可以定义CAST类型了。

```
CREATE CAST (inet AS boolean)
WITH FUNCTION inet_to_boolean(inet) AS IMPLICIT;
```

我们需要做的第一件事是告诉PostgreSQL，我们要把inet转换为boolean。然后，函数被列出，我们告诉PostgreSQL，我们更喜欢隐式转换。 这是一个简单明了的过程，我们可以按照下面的方法来测试转换。

```
test=# SELECT '192.168.0.34'::inet::boolean;
 bool
------
 t
(1 row)
```

类型案例是成功的。 基本上，同样的逻辑也可以应用于定义整理。同样，可以用一个存储过程来执行任何需要做的事情。

```
test=# \h CREATE COLLATION
Command: CREATE COLLATION
Description: define a new collation
Syntax:
CREATE COLLATION [ IF NOT EXISTS ] name (
 [ LOCALE = locale, ]
 [ LC_COLLATE = lc_collate, ]
 [ LC_CTYPE = lc_ctype, ]
 [ PROVIDER = provider, ]
 [ DETERMINISTIC = boolean, ]
 [ VERSION = version ]
)
CREATE COLLATION [ IF NOT EXISTS ] name FROM existing_collation
URL: https://www.postgresql.org/docs/13/sql-createcollation.html
```

CREATE COLLATION的语法真的很简单。虽然创建排序是可能的，但它仍然是很少使用的功能之一。

存储过程和函数提供了很多东西。许多事情都是可能的，而且可以用一种漂亮而有效的方式来完成。

5 总结

在本章中，你已经学会了如何编写存储过程。在理论介绍之后，我们的注意力集中在PL/pgSQL的一些精选特性上。除此之外，你还学习了如何使用PL/Perl和PL/Python，这是PostgreSQL提供的两种重要语言。当然，还有更多的语言可以使用。但是，由于本书范围的限制，无法详细介绍它们。如果你想了解更多，请查看以下网站：https://wiki.postgresql.org/wiki/PL_Matrix。我们还学习了如何改进函数调用，以及如何将其用于其他各种用途，以加快应用程序的速度，并做更多的事情。

在第8章，管理PostgreSQL的安全，你将学习PostgreSQL的安全。你将学习如何在总体上管理用户和权限。在此基础上，你还将学习网络安全。

6 问题

- 函数和存储过程之间的区别是什么?
- 受信任的语言和不受信任的语言之间的区别是什么?
- 一般来说，函数是好是坏?
- 在PostgreSQL中哪些服务器端语言是可用的?
- 什么是触发器?
- 哪些语言可以用来编写函数?
- 哪种语言是最快的?

这些问题的答案可以在 GitHub 仓库中找到 (https://github.com/PacktPublishing/Mastering-PostgreSQL-13-Fourth-Edition)