OCCI 编程

前 言

开发基于 Oracle 数据库的应用程序，我们可以选择多种工具，不仅可以用一般的数据库开发技术，诸如 ADO(ActiveX Data Objects)、ODBC(Open DataBaseConnectivity)等等，同时，也可以用 Oracle 公司提供的专门的开发工具，诸如 Pro C_C++，OCI(Oracle Call Intedace)，OCCI(Oracle C++ Call Intedace)等等。比较这几种方式，前者因为是通用技术，开发起来比较容易，但是有一个致命的弱点就是诸如 ADO 之类的通用技术的速度太慢，如果我们要开发管理海量数据的数据库，比如影像数据库，那么，这种速度我们是不能忍受的。而 OCCI 虽然开发起来难度大一些，但是它的速度极快，而且是一种底层接口，几乎可以操纵 Oracle 数据库的任何对象。

目 录

1、OCCI 入门(Introduction to OCCI)

1.1、OCCI 综述(Overview of OCCI)

OCCI是一个提供了C++应用程序使用ORACLE数据库中数据的API接口，OCCI能够使C++编程者最大限度利用ORACLE数据库的操作，包括SQL语句处理和对象处理
OCCI提供以下内容

- 通过有效的利用系统内存和网络连接使应用程序达到最高性能
- 可升级应用程序服务于不断增长的用户数和请求数
- 运用ORACLE数据库对象，包括客户端允许使用的数据库对象，全面支持应用开发
- 简单的用户验证和密码管理
- n-tiered体系架构验证
- 对在two-tier clent/server环境中或者是multitiered环境中，采用一致的接口进行动态连接管理和事物管理
- 封闭和接口不透明处理
  OCCI一个提供访问标准数据的库文件和能够以C++应用程序运行时链接的运态链接库的形式retrieval函数.这就消除了需要嵌入SQL或者PL/SQL包括的第三代语言。

1.1.1、使用 OCCI 的好处(Benefits of OCCI)

OCCI 提供其它访问 ORACLE 数据库所不具备的重要优势

- 利用c++和对象导向编程范例
- 简单易用
- 与JDBC比较相近，容易学会
- 可以象操作C++实例一样操作用户定义的数据库对象

1.1.2、建立 OCCI 应用程序(Building an OCCI Application)

如图1-1所示，你可以象编译和链接一个没有数据库的应用程序一样来编译和链接一个OCCI应用程序

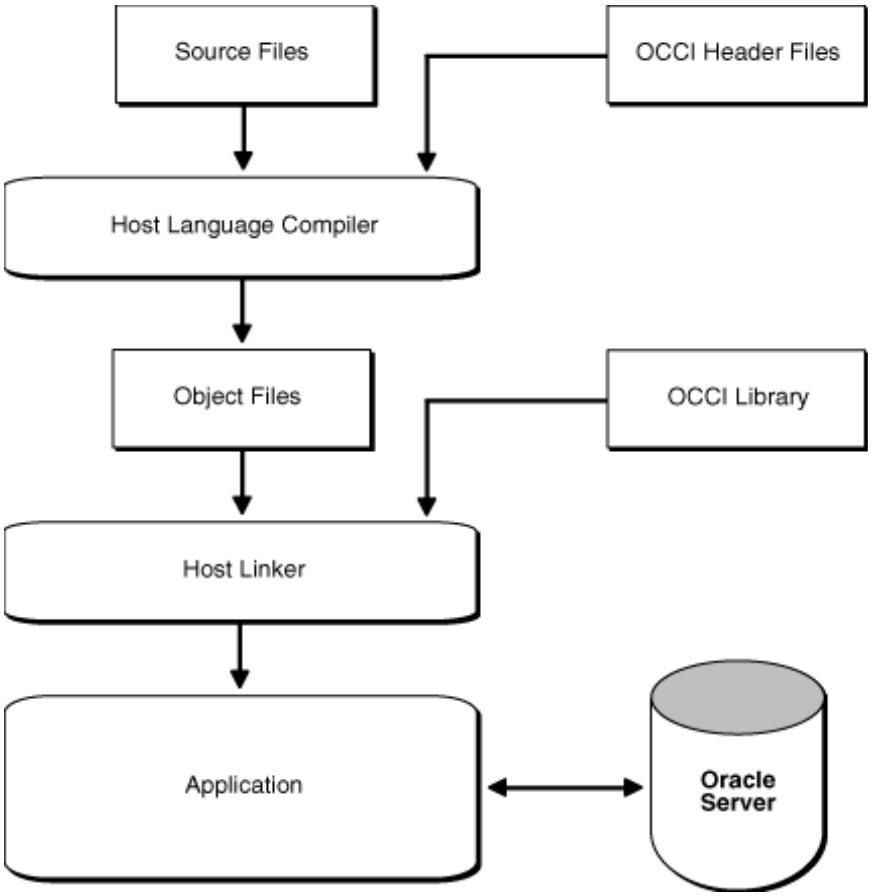

图 1-1 OCCI开发步骤

ORACLE支持大多数流行的第三方编译。链接一个OCCI程序的细节由于系统不同而不同，在某些平台上，除OCCI库外，可能需要包括其它的库文件链接你的OCCI程序。

1.1.3、OCCI 的功能(Functionality of OCCI)

OCCI提供下列功能

- API设计可升级，多线程应用能够提供大用户量的安全使用
- 提供SQL访问函数，管理数据库访问，执行SQL语句，和重新得到操作ORACLE数据库服务的对象
- 为了象处理属性一样处理ORACLE类型，提供数据类型映射和处理函数
- 先进的消息管理队列

⬧XA compliance for distributed transaction support XA符合分布式支持
⬧Statement caching of SQL and PL/SQL queries Statement缓存SQL和PL/SQL查询
⬧Connection pooling for managing multiple connections 可管理多种多样连接的连接池
⬧Globalization and Unicode support to customize applications for international and regional language requirement 全面的Unicode支持国际化和区域化定制程序
⬧Object Type Translator Utility事物类型翻译功能
⬧Transparent Application Failover support 清晰的程序错误处理支持

1.1.4、程式化与非程式化的元素(Procedural and Nonprocedural Elements)

Oracle C++ Call Interface (OCCI) enables you to develop scalable, multithreaded applications on multitiered architectures that combine nonprocedural data access power of structured query language (SQL) with the procedural capabilities of C++. In a nonprocedural language program, the set of data to be operated on is specified,but what operations will be performed, or how the operations are to be carried out, is not specified. The nonprocedural nature of SQL makes it an easy language to learn
and use to perform database transactions. It is also the standard language used to access and manipulate data in modern relational and object-relational database systems.
Oracle C++ 调用接口(OCCI)允许你用程式化的C++和非程式化的具有强大数据操作能力的SQL引擎开发在多种多样架构平台上的可升级、多线程程序。在非程式化语言的程序中，结果集是特定的，但是操作执行或者抛出操作，是非特定的。非程式化的SQL使其容易学习和迎来执行数据库事务。它同时也是当前关系模型数据库和对象-关系模型数据库操作数据和产生数据的标准语句。

In a procedural language program, the execution of most statements depends on previous or subsequent statements and on control structures, such as loops or conditional branches, which are not available in SQL. The procedural nature of these languages makes them more complex than SQL, but it also makes them very flexible and powerful.
在程式化语言程序中，大多数语句的执行依靠先前的或者子程式语句和控制结构，例如循环和条件分支，这些在SQL不适用。这些程式化模型的语言致使它们远比SQL复杂，但是也使得它们非常灵活和强大。

The combination of both nonprocedural and procedural language elements in an OCCI program provides easy access to an Oracle database in a structured programming environment.
程式化与非程式化元素的联合使得OCCI程序在程式化的环境中非常容易操作Oracle数据库。

OCCI supports all SQL data definition, data manipulation, query, and transaction control facilities that are available through an Oracle database server. For example, an OCCI program can run a query against an Oracle database. The queries can require the program to supply data to the database by using input (bind) variables, as follows:
OCCI通过一个Oracle数据库服务器支持所有SQL数据定义，数据控制。查新，事务控制能力。例如，一个OCC程序可以执行一个Oracle数据库的查询。查询可以通过程序绑定变量为Oracle数据库提供数据，如下所示:

```
SELECT name FROM employees WHERE empno = :empnumber
```

In this SQL statement, *empnumber* is a placeholder for a value that will be supplied by the application.
在这条SQL语句中，*empnumber是程序所提供数据值的一个占位符。*

In an OCCI application, you can also take advantage of PL/SQL, Oracle's procedural extension to SQL. The applications you develop can be more powerful and flexible than applications written in SQL alone. OCCI also provides facilities for accessing and manipulating objects in an Oracle database server.
在 OCCI 程序中，你也可以使用 SQL 的延伸 PL/SQL，Oracle 存储过程。这样你开发的程序将会比只使用 SQL 功能更强大和灵活。OCCI 也提供了 Oracle 数据库服务器操作和控制事务的能力。

1.2、SQL 语句执行(Processing of SQL Statements)

1.3、PL/SQL 概述(Overview of PL/SQL)

1.4、特殊的 OCCI/SQL 条款(Special OCCI/SQL Terms)

2、安装和升级(Installation and Upgrading)

首先，为了防止某些动态链接库出问题，建议在安装了 Oracle 客户端的机器上进行开发、运行。
其次，使用 OCCI 开发的程序，需要使用 Oracle 客户端的 tnsnames.ora 这个配置文件，所以在开发前需要使用 netca 来配置好相关内容。
第三，Linux 下的系统环境变量需要设置好。需要设置的环境变量包括 ORACLE_HOME、ORACLE_SID、TNS_ADMIN，其中 TNS_ADMIN 指定到 tnsnames.ora 所在的文件夹。

3、编程相关(Relational Programming)

本章描述运用OCCI进行C++应用程序编程处理存储在关系统数据库中数据的基本要素。

本章包括以下主题：

- 连接数据库
- 共享数据库连接
- 执行SQL DDL和DML语句

- 在OCCI环境中SQL语句的类型
- 执行SQL查询
- 执行动态语句
- 提交事物
- 捕捉语句
- 处理异常

3.1、连接数据库(Connecting to a Database)

关于应用程序连接数据库你有许多不同的选择

3.1.1、创建和终结一个环境(Creating and Terminating an Environment)

所有的OCCI进程处理都与Environment类有关。一个OCCI环境提供应用模式和用户定义内存管理函数，下面代码示例显示如何创建一个OCCI环境：

```
Environment *env = Environment::createEnvironment();
```

所有的OCCI对象都有createxxx 方法来创建（连接，连接池，声明）必须被明确的终止，适当的时候，你必须显示的终止环境，下面的代码示例如何终止一个OCCI环境

```
Environment::terminateEnvironment(env);
```

另外，一个OCCI环境应该有一个比下列对象类型有更大的范围，如Agent, Bytes, Date, Message, IntervalDS, IntervalYM, Subscription 和 Timestamp。这个规则不适用于BFile, Blob和Clob对象。这个规则的示例如下：

```
const string userName = "SCOTT";
const string password = "TIGER";
const string connectString = "";
Environment *env = Environment::createEnvironment();
{
    Connection *conn = env->createConnection(
    userName, password, connectString);
    Statement *stmt = conn->createStatement(
    "SELECT blobcol FROM mytable");
    ResultSet *rs = stmt->executeQuery();
    rs->next();
    Blob b = rs->getBlob(1);
    cout << "Length of BLOB : " << b.length();
     .
     .
     .
     stmt->closeResultSet(rs);
     conn->terminateStatement(stmt);
     env->terminateConnection(conn);
}
```

```
Environment::terminateEnvironment(env);
```

这个应用需要访问全局对象，如静态或者全局变量，这些对象必须在环境变量终止前设置成NULL，在前面提到的例子中，如果b是一个全局变量，那么b.setNull()调用优先在terminateEnvirorment()之前调用。

你可以指定应用程序中createEnviroment方法的参数

- 运行的线程环境(THREADED_MUTEXED 或者 THREADED_UNMUTEXED)
- 运用对象
  这个模式允许你对每个环境独立设定

3.1.2、打开和关闭一个连接(Opening and Closing a Connection)

Environment 类是一个创建 Connection 对象的类工厂，首先你需要创建一个 Environment 实例，然后通过 createConnection()方法能够让用户连接到数据库。
下面的代码示例创建了一个 Environment 实例，然后用它创建一个数据库连接，用户名为 scott,密码 tiger.

```
Environment *env = Environment::createEnvironment();
Connection *conn = env->createConnection("scott", "tiger");
```

你必须在工作完成以后用如下代码所示的terminateConnection()方法显示关闭连接。另外，OCCI环境也应该被 显示终止
你必须记住所有的对象(Refs, Bfiles, Produces, Consumers等等)，在Connection终止前，那些内部范围的实例，必须被显示的终止。

```
env->terminateConnection(conn);
Environment::terminateEnvironment(env);
```

3.2、共享数据库连接-连接池(Pooling Connections)

本节讨论如何利用 OCCI 的连接池特性，包括以下信息内容：

- Creating a Connection Pool
- Stateless Connection Pooling

这两个连接池的主要区别是 StatelessConnectionPools 用于应用程序不需要考虑状态，这些应用能够通过运用预鉴别连接提高性能

3.2.1、使用连接池(Using Connection Pools)

许多中级应用程序，数据库的连接要求能够处理大量的进程，由于每个进程只存活较短的时间，所以为每个进程打开数据库连接导致的结果是连接的利用率低和较低的性能。
通过使用连接池特性，应用程序可以创建一个少量的连接应对大量的进程，这将有助于你更有效的利用数据库的资源。

3.2.1.1、创建一个连接池(Creating a Connection Pool)

为了创建一个连接池，可以用createConnectionPool()方法

```
virtual ConnectionPool* createConnectionPool(
const string &poolUserName,
```

```
const string &poolPassword,
const string &connectString ="",
unsigned int minConn =0,
unsigned int maxConn =1,
unsigned int incrConn =1) = 0;
```

在上面的示例中可以使用下列参数：

- poolUserName: 连接池的属主
- poolPassword: 获得进入连接池的密码
- onnectString:指定连接池与数据库服务关联的数据库名
- ninConn:连接池创建后开放的最少连接数
- maxConn:连接池所能支持的最大连接数，当连接池中的连接数达到最大值时，一个OOCI方法需要一个连接时，必须等到有一个连接释放，除非在连接池中设置了setErrorOnBusy()错误调度
- incrConn:当所有的连接都处理忙碌状态时，又有新的请求需要一个连接，可以设置额外的连接数。这个增量只有在当所有的连接数小于最大连接数时，才会允许在连接池中打开。

下面的代码例子示范你如何创建一个连接池

```
const string connectString = "";
unsigned int maxConn = 5;
unsigned int minConn = 3;
unsigned int incrConn = 2;
ConnectionPool *connPool = env->createConnectionPool(
poolUserName,
poolPassword,
connectString,
minConn,
maxConn,
incrConn);
```

你当然也可以动态的配置这些属性。这允许你应用能够随时读取当前加载和适当的调整这些属性（打开连接的数量和忙碌连接的数量）。另外，你可以使用setTimeOut()方法，当空闲时间大于指定时间时使连接终止。OCCI会周期性的终止空闲连接来保持打开连接数为最合适的值。一个环境有几个连接池没有限制，在一个OCCI环境中可以有多个连接，这些连接可以指向一个或多个数据库，这将有助于开发需要负载平衡的应用。

3.2.1.2、代理连接(Proxy Connections)

If you authorize the connection pool user to act as a proxy for other connections, then no password is required to log in database users who use one of the connections in the connection pool.

如果你授权一个连接池的用户作为另外连接的代理，这时登录连接池中的连接的用户无需密码验证。

A proxy connection can be created by using either of the following methods:

创建一个代理连接可以用下面方法中的任意一种:

```
ConnectionPool->createProxyConnection(
```

```
const string &username,
Connection::ProxyType proxyType = Connection::PROXY_DEFAULT);
```

or

```
ConnectionPool->createProxyConnection(
const string &username,
string roles[],
int numRoles,
Connection::ProxyType proxyType = Connection::PROXY_DEFAULT);
```

The following parameters are used in the previous method example:
下面是前面事例方法中用到的参数信息：

- `roles[]`:The roles array specifies a list of roles to be activated after the proxy connection is activated for the client
角色数组决定了代理连接生效后客户端充当的角色列表。
- `Connection::ProxyType proxyType = Connection::PROXY_DEFAULT`:
The enumeration `Connection::ProxyType` lists constants representing the various ways of achieving proxy authentication. `PROXY_DEFAULT` is used to indicate that `name` represents a database username and is the only proxy authentication mode currently supported.
枚举类型 `Connection::ProxyType` 列举出了完成代理验证的所有方式。
`PROXY_DEFAULT` 用来表明 `name` 代表一个数据库用户和当前仅仅支持的代理验证模式。

3.2.2、无状态连接池(Stateless Connection Pooling)

Stateless Connection Pooling is specifically designed for use in applications that require short connection times and don't need to deal with state considerations. The primary benefit of Stateless Connection Pooling is increased performance, since the time consuming connection and authentication protocols are eliminated. Stateless Connection Pools create and maintain a group of stateless, authenticated connection to the database that can be used by multiple threads. Once a thread finishes
using its connection, it should release the connection back to the pool. If no connections are available, new ones are generated. Thus, the number of connections in the pool can increase dynamically.
无状态连接池设计目的是为那些短时连接、不需要关系状态的应用程序服务的。无状态连接池的第一个好处是可以增加效率，因为连接时间消耗和验证协议是可以忽略不计的。无状态连接池创建一组可以被多线程使用的无状态的、已经验证的数据库连接。一旦一个线程完成了它所使用的连接，必须释放该连接到连接池。如果没有连接可用，一个新的连接将被创建。因此连接数在连接池中是动态增加的。
Some of the connections in the pool may be tagged with specific properties. The user may request a default connection, set certain attributes, such as Globalization Support settings, then tag it and return it to the pool. When a connection with same attributes is needed, a request for a connection with the

same tag can be made, and one of several connections in the pool with the same tag can be reused. The tag on a connection can be changed or reset.
在连接池中的一些连接也可以增加一些特殊属性。用户可能需求一个默认的连接，设置特定属性，例如支持全局设置，这时标记它并返回到连接池中。当需要一个具有相同属性的连接时，具有与之属性相同的标记请求可以被设置，具有相同属性标记的几个连接中的一个可以被复用。在连接上的标记可以改变或者重置。
Proxy connections may also be created and maintained through the Stateless Connection Pooling interface. Stateless connection pooling improves the scalability of the mid-tier applications by multiplexing the connections. However, connections from a `StatelessConnectionPool` should not be used for long transactions, as holding connections for long periods leads to reduced concurrency.
代理链接的创建和获取也可以通过无状态连接池的接口来实现。无状态连接池通过多元化连接提升了中间层应用程序的可升级性。尽管如此，一个来自于`StatelessConnectionPool`无状态连接池的连接还是不能用于长连接事务。因为长时间的把持一个连接会导致并发性降低。

There are two types of stateless connection pools:
有两种类型的无状态连接池：
- A homogeneous pool is one in which all the connections will be authenticated with the username and password provided at the time of creation of the pool. Therefore, all connections will have the same authentication context. Proxy connections are not allowed in such pools.

同性质的连接池　　在创建连接池的时候其中所有的连接都通过相同的用户名和密码进行验证。所以所有的连接有相同的验证内容。代理链接是不允许发生在这种连接池中的。
- Different connections can be authenticated by different usernames in heterogeneous pools. Proxy connections can also exist in heterogeneous pools, provided the necessary privileges for creating them are granted on the server.

混杂的连接池　　不同的连接通过不同用户名进行验证。代理链接也可以存在于这种混在的连接池中，在创建时通过服务器授予必须的特权。

Example 3–1 illustrates a basic usage scenario for connection pools.
Example 3–2 presents the usage scenario for creating and using a homogeneous stateless connection
pool, while Example 3–3 covers the use of heterogeneous pools.

Example 3–1 演示了连接池基本用法的描述；
Example 3–2 介绍了创建和使用一个同性质连接池的用法；
Example 3–3 报道了混杂连接池的用法。

Example 3–1 Usage Scenario for a StatelessConnectionPool
Because the pool size is dynamic, in response to changing user requirements, up to the
specified maximum number of connections. Assume that a stateless connection pool is

created with following parameters:
因为连接池大小是动态改变的，在回复用户请求时到达了特定的最大连接数。假设创建连接池时传递了一下一些参数：

- `minConn = 5`
- `incrConn = 2`
- `maxConn = 10`

Five connections are opened when the pool is created:
当连接池创建时5个连接已经被打开:

- `openConn = 5`

Using `get[AnyTagged][Proxy]Connection()` methods, the user consumes all 5 open connection:
用 `get[AnyTagged][Proxy]Connection()` 方法, 用户使用了全部的5个已打开连接:

- `openConn = 5`
- `busyConn = 5`

When the user wants another connection, the pool will open 2 new connections and
return one of them to the user
当一个用户请求另外一个连接时，连接池会打开2个连接，返回其中一个给用户

- `openConn = 7`
- `busyConn = 6`

The upper limit for the number of connections that can be pooled is `maxConn` specified at the time of creation of the pool. The user can also modify the pool parameters after the pool is created using the call to `setPoolSize()` method.
连接池中被轮询的连接数上限可以在创建连接池时通过 `maxConn`设定。在创建连接池后用户也可以通过`setPoolSize()`函数更改连接池参数。
If a heterogenous pool is created, the `incrConn` and `minConn` arguments are ignored.
如果创建的是混杂连接池，那么`incrConn` 和 `minConn` 参数将被忽略。

Example 3-2 How to Create and Use a Homogeneous Stateless Connection Pool
To create a homogeneous stateless connection pool, follow these basic steps and pseudocode commands:
创建同性质连接池时，可参照下面的基本步骤和伪代码:
1. Create a stateless connection pool in the `HOMOGENEOUS` mode of the `Environment` with a createStatelessConnectionPool() call.
创建 `HOMOGENEOUS` 性质的连接池需要调用`Environment`的 createStatelessConnectionPool()函数。

```
StatelessConnectionPool *scp =
env->createStatelessConnectionPool(
username, passwd, connectString, maxCon, minCon, incrCon,
StatelessConnectionPool::HOMOGENEOUS );
```

2. Get a new or existing connection from the pool by calling the getConnection() method.

获取一个新的或者已经存在于连接池中的连接需要调用getConnection()函数。

```
Connection *conn=scp->getConnection(tag);
```

During the execution of this call, the pool is searched for a connection with a matching tag. If such a connection exists, it is returned to the user. Otherwise, an untagged connection authenticated by the pool username and password is returned.

在调用运行期间，连接池会寻找一个匹配的连接标记。如果这样的一个连接存在，它将返回给用户。否则一个没有标记的连接将通过用户名和密码验证并返回。

Alternatively, you can obtain a connection with getAnyTaggedConnection() call that has been overloaded for heterogeneous pools. It will return a connection with a non-matching tag if neither a matching tag or `NULL` tag connections are available. You should verify the tag returned by a getTag() call on `Connection`.

做为选择，你可以通过getAnyTaggedConnection()获取一个连接。在没有一个匹配的和空闲的标记连接可用时，它将返回一个无匹配标记的连接。你可以通过getTag()更改这个返回的连接的标记。

```
Connection *conn=scp->getAnyTaggedConnection(tag);
string tag=conn->getTag();
```

3.使用连接(Use the connection).

4. Release the connection to the `StatelessConnectionPool` through the releaseConnection() call.

通过releaseConnection() 释放 StatelessConnectionPool 中的一个连接。

```
scp->releaseConnection(conn, tag);
```

An empty tag, "", untags the `Connection`. 一个空的标记, "", 不标记`Connection`.

You have an option of retrieving the connection from the `StatelessConnectionPool` using the same `tag` parameter value in a getConnection() call.

你有这样的一个选择，使用相同的 `tag` 标记参数值，通过getConnection()来挽回`StatelessConnectionPool`中的一个连接。

```
Connection *conn=scp->getConnection(tag);
```

Instead of returning the `Connection` to the `StatelessConnectionPool`, you may wish to destroy it using the terminateConnection() call.

你可能希望通过销毁terminateConnection()它，而不是把连接放回连接池`StatelessConnectionPool`中。

```
scp->terminateConnection(conn);
```

5. Destroy the pool through a terminateStatelessConnectionPool() call on the `Environment` object.

通过Environment实例调用terminateStatelessConnectionPool()来销毁连接池。

```
env->terminateStatelessConnectionPool(scp);
```

Example 3-3 How to Create and Use a Heterogeneous Stateless Connection Pool

To create a heterogeneous stateless connection pool, follow these basic steps and pseudocode commands:

创建混杂连接池时，可参照下面的基本步骤和伪代码:

1. Create a stateless connection pool in the `HETEROGENEOUS` mode of the

Environment with a createStatelessConnectionPool() call.

创建 HETEROGENEOUS 性质的连接池需要调用Environment的createStatelessConnectionPool()函数。

```
StatelessConnectionPool *scp =
env->createStatelessConnectionPool(
username, passwd, connectString, maxCon, minCon, incrCon,
StatelessConnectionPool::HETEROGENEOUS);
```

2. Get a new or existing connection from the pool by calling the getConnection() method of the StatelessConnectionPool that is overloaded for the heterogeneous pool option.

获取一个新的或者已经存在于StatelessConnectionPool连接池中的连接需要调用getConnection()函数。

```
Connection *conn=scp->getConnection(username, passwd, tag);
```

During the execution of this call, the heterogeneous pool is searched for a connection with a matching tag. If such a connection exists, it is returned to the user. Otherwise, an appropriately authenticated untagged connection with a NULL tag is returned.

在调用运行期间，连接池会寻找一个匹配的连接标记。如果这样的一个连接存在，它将返回给用户。否则一个没有标记的连接将验证并返回。

Alternatively, you can obtain a connection with getAnyTaggedConnection() call. It will return a connection with a non-matching tag if neither a matching tag or NULL tag connections are available. You should verify the tag returned by a getTag() call on Connection.

做为选择，你可以通过etAnyTaggedConnection()获取一个连接。在没有一个匹配的和空闲的标记连接可用时，它将返回一个无匹配标记的连接。你可以通过getTag()更改这个返回的连接的标记。

```
Connection *conn=scp->getAnyTaggedConnection(username, passwd, tag);
string tag=conn->getTag();
```

You may also wish to use proxy connections by getProxyConnection() or getAnyTaggedProxyConnection() calls on the StatelessConnectionPool.

你可能希望通过getProxyConnection()或者getAnyTaggedProxyConnection()来使用StatelessConnectionPool中的一个代理连接。

```
Connection *pcon = scp->getProxyConnection(proxyName, roles{},
nuRoles, tag, proxyType);
Connection *pcon = scp->getAnyTaggedProxyConnection( proxyName, tag,
proxyType);
```

3. **使用连接(Use the connection)**.

4. Release the connection to the StatelessConnectionPool through the releaseConnection() call.

通过releaseConnection() 释放 StatelessConnectionPool 中的一个连接。`scp->releaseConnection(conn, tag);`

An empty tag, "", untags the Connection. 一个空的标记, "", 不标记Connection.

You have an option of retrieving the connection from the `StatelessConnectionPool` using the same `tag` parameter value in a getConnection() call.
你有这样的一个选择，使用相同的 `tag` 标记参数值，通过getConnection()来挽回`StatelessConnectionPool`中的一个连接。

```
Connection *conn=scp->getConnection(tag);
```

Instead of returning the `Connection` to the `StatelessConnectionPool`, you may wish to destroy it using the terminateConnection()terminateStatelessConnectionPool() call.
你可能希望通过销毁terminateConnection()它，而不是把连接放回连接池`StatelessConnectionPool`中。

```
scp->terminateConnection(conn);
```

5. Destroy the pool through a terminateStatelessConnectionPool() call on the `Environment` object.
通过`Environment`实例调用terminateStatelessConnectionPool()来销毁连接池。

```
env->terminateStatelessConnectionPool(scp);
```

3.2.3、数据库常驻连接池(Database Resident Connection Pooling)

3.2.3.1、管理数据库常驻连接池(Administrating Database Resident Connection Pools)

3.2.3.1、使用数据库常驻连接池(Using Database Resident Connection Pools)

3.3、执行 DDL SQL 和 DML 语句(Executing SQL DDL and DML Statements)

SQL是关系型数据库的通用语言，在OCCI中，你可以用Statement类执行ＳＱＬ命令。

3.3.1、创建一个 Statement 对象(Creating a Statement Object)

通过调用Connection对象的createStatement()方法可以创建一个Statement对象，如下所示：

Statement *stmt = conn->createStatement();

创建一个对象执行SQL命令

3.3.2、创建一个执行 SQL 命令的对象(Creating a Statement Object that Executes SQL Commands)

当创建完Statement对象后，可以调用Statement中的execute(), executeUpdate(), executeArrayUpdate(), executeQuery()方法执行SQL命令，这些方法的目的分述如下：

- execute():执行所有的非特殊statement类型
- executeUpdate():执行DML和DDL语句
- executeQuery():执行一个查询
- executeArrayUpdate():执行复合 D M L 语句

3.3.2.1、创建一个数据库表(Creating a Database Table)

用executeUpdate()方法,下面代码示例显示如何创建一个数据库表

```
stmt->executeUpdate("CREATE TABLE basket_tab
(fruit varchar2(30), quantity number)");
```

3.3.2.2、往数据库表中插入数据(Inserting Values into a Database Table)

同样，你可以通过调用executeUpdate方法执行SQL INSERT语句：

```
stmt->executeUpdate("insert into basket_table
values('MANGOES',3)");
```

执行executeUpdate()方法返回该SQL语句影响的记录条数。

3.3.3、重新使用一个 Statement 对象(Reusing the Statement Object)

你可以使用一个Statement对象执行多次SQL语句。举例说明：用不同的参数执行相同的语句，通过Statement对象中的setSQL方法，你可以指定不同的参数。

```
stmt->setSQL("INSERT INTO basket_tab values(:1, :2)");
```

运用这个方法，你可以根据需要多次执行这个INSERT语句了。如果你希望执行另一个不同的SQL语句，只需简单的重设一下statement对象，举例说明：

```
stmt->setSQL("SELECT * FROM basket_tab where quantity >= :1");
```

通过这种方法使用的OCCI对象和他们的资源不需要分配和释放，你可以在任何时间用getSQL()方法重新获取当前的statement对象

3.3.4、终止一个 Statement 对象(Terminating a Statement Object)

你必须显示的终止或取消一个Statement对象

Connection::conn->terminateStatement(Statement *stmt);

3.4、在 OCCI 环境中的 SQL 语句的类型(Types of SQL Statements in the OCCI Environment)

在OCCI环境中允许使用三种类型的SQL语句

- 用SQL命令和指定值的标准SQL语句
- 带有参数或绑定变量的参数语句
- 调用PL/SQL存储过程的请求语句

Statement方法可以被细分为应用的所有语句，带参数的语句，和调用的语句。

标准的语句是带参数语句的父类，参数的语句又是调用的父类。

3.4.1、标准语句(Standard Statements)

前面的章节描述了DDL和DML命令，举例说明：

stmt->executeUpdate(“create table basket_tab (fruit varchar2(30), quantity number)”);

stmt->executeUpdate(“insert into basket_tab values(‘mangoes’, ‘3’)”);

这是标准语句中明确定义语句值的示例。所以，在这些例子中，建表语句指定了表名为basket_tab，插入语句约定了插入的值为(’mangoes’, 3)。

3.4.2、参数化的语句(Parameterized Statements)

你可以通过在同一个语句中设定占位符，用于执行不同的参数，语句中变量是可以输入的。

这些语句参考参数化的语句，参数化的语句可以接受来自用户或程序的输入。

举例说明：假定你需要执行一个INSERT语句用不同的参数，你必须通过Statement对象中的setSQL()方法来指定语句：

stmt->setSQL(“INSERT INTO basket_tab values(:1, :2)”);

然后可以调用setxxx()方法指定特定的参数，xxx代表参数的类型。下面的例子中使用setString()和setInt()方法指定上面例子中的第一和第二个参数。

插入一条记录：

stmt->setString(1, “Bananas”); //第一个参数

stmt->setInt(2, 5); //第二个参数
完成参数设定后，就可以插入一条记录了

stmt->executeUpdate();
插入另一条记录

stmt->setString(1, “apples”); //value for first parameter
stmt->setInt(2, 9); //value for second parameter
完成参数指定后，可以再次使用下面的语句执行

stmt->executeUpdate();

If your application is executing the same statement repeatedly, then avoid changing the input parameter types because a rebind is performed each time the input type changes.
如果你的程序重复执行一个相同的语句，应当避免输入参数的更改，因为输入参数改变时需要重新绑定参数。

3.4.3、可调用语句(Callable Statements)

PL/SQL stored procedures, as their name suggests, are procedures that are stored on the database server for reuse by an application. By using OCCI, a call to a procedure which contains other SQL statements is referred to as a **callable statement**.
PL/SQL 存数过程，顾名思义，就是存储在数据库服务器上的可以被程序重复的过程。通过OCCI，调用具有其他SQL语句的存数过程被称为一个**callable statement**
For example, suppose you wish to call a procedure `countFruit()`, that returns the quantity of a specified kind of fruit. To specify the input parameters of a PL/SQL stored procedure, call the `set`*XXX*`()` methods of the `Statement` class as you would for parameterized statements.
例如，假定你希望一个存储过程`countFruit()`，并让它返回一种特殊水果的质量。通过你想参数化的`Statement`类的`set`*XXX*`()`函数来敲定存储过程的输入参数。

```
stmt->setSQL("BEGIN countFruit(:1, :2); END:");
int quantity;
stmt->setString(1, "Apples"); // specify first (IN) parameter of procedure
```

However, before calling a stored procedure, you need to specify the type and size of any `OUT` parameters by calling the `registerOutParam()` method. For `IN/OUT`

parameters, use the `set`*XXX*`()` methods to pass in the parameter, and `get`*XXX*`()` methods to retrieve the results.
尽管如此，在调用存储过程之前，你需要你通过调用`registerOutParam()`方法来确定任何`OUT`输出参数。因为输入/输出参数，通过`set`*XXX*`()`来传入参数，通过`get`*XXX*`()`来获取参数值。

```
stmt->registerOutParam(2, Type::OCCIINT, sizeof(quantity));
// specify type and size of the second (OUT) parameter
```

You now execute the statement by calling the procedure:
现在你可以调用存储过程来执行相应的sql语句:

```
stmt->executeUpdate(); // call the procedure
```

Finally, you obtain the output parameters by calling the relevant `get`*xxx()* method:
最后，你需要通过调用`get`*xxx()*函数来获取输出参数:

```
quantity = stmt->getInt(2); // get value of the second (OUT) parameter
```

3.4.3.1、以数组作为参数的可调用语句(Callable Statements with Arrays as Parameters)

A PL/SQL stored procedure executed through a callable statement can have array of values as parameters. The number of elements in the array and the dimension of elements in the array are specified through the `setDataBufferArray()` method.
一个PL/SQL 存储过程的执行可能不要一组可调用语句的参数值。数组中的元素数量和元素模型通过`setDataBufferArray()`来确定。

The following example shows the `setDataBufferArray()` method:
下面展示了`setDataBufferArray()`方法:

```
void setDataBufferArray(
unsigned int paramIndex,
void *buffer,
Type type,
ub4 arraySize,
ub4 *arrayLength,
sb4 elementSize,
ub2 *elementLength,
sb2 *ind = NULL,
ub2 *rc = NULL);
```

The following parameters are used in the previous method example:
下面的参数是前面方法中的用到的参数：

- `paramIndex`: Parameter number 参数数目
- `buffer`: Data buffer containing an array of values 数组的数据缓冲区
- `Type`: Type of data in the data buffer 数据缓冲区中的数据类型
- `arraySize`: Maximum number of elements in the array 数组中元素的最大数目
- `arrayLength`: Number of elements in the array 数组中元素的数目

- elementSize: Size of the current element in the array 数组中当前元素的大小
- elementLength: Pointer to an array of lengths. elementLength[i] has the current length of the ith element of the array
指向数组的大小. elementLength[i]表示数组中的第i个元素
- ind: 消息指针
- rc: 返回值

3.4.4、流化的读和写(Streamed Reads and Writes)

OCCI supports a streaming interface for insertion and retrieval of very large columns by breaking the data into a series of small chunks. This approach minimizes client-side memory requirements. This streaming interface can be used with parameterized statements such as SELECT and various DML commands, and with callable statements in PL/SQL blocks. The datatypes supported by streams are BLOB, CLOB, LONG, LONG RAW, RAW, and VARCHAR2.
OCCI支持流接口插入和检索巨大的列转换为一系列小的部分。这种方法最小化了客户端内存需求。流接口支持诸如SELECT等参数化了的语句和多种多样的DML命令，和可调用PL/SQL语句块。可被流支持的数据类型有BLOB, CLOB, LONG, LONG RAW, RAW, VARCHAR2。

三种数据流：
- A **writable** stream corresponds to a bind variable in a SELECT/DML statement or an IN argument in a callable statement.
可写数据流和SELECT/DML语句中的绑定参数或者可调用语句中的输入参数相当。
- A **readable** stream corresponds to a fetched column value in a SELECT statement or an OUT argument in a callable statement.
可读数据流和SELECT语句获取的列值或者可调用语句的输出参数相当。
- A **bidirectional** stream corresponds to an IN/OUT bind variable.
双向数据流和绑定的输入输出参数相当。

Methods of the Stream Class support the stream interface：
Stream Class流类的方法支持流接口：

The getStream() method of the Statement Class returns a stream object that supports reading and writing for DML and callable statements:
Statement Class的getStream()法返回一个支持DML读写的、可被调用的流对象：
- For writing, it passes data to a bind variable or to an IN or IN/OUT argument
- For reading, it fetches data from an OUT or IN/OUT argument
写的时候，它传数据给一个绑定变量或者一个IN输入或者IN/OUT输入/输出出参数
读的时候，它从一个OUT输出或者IN/OUT输入/输出参数中获得数据。

The getStream() method of the ResultSet Class returns a stream object that can be used for reading data.

ResultSet Class的getStream()方法返回一个可用来读取数据的流对象。

The `status()` method of these classes determines the status of the streaming operation.
这些类的`status()`方法用来确定流操作的状态。

3.4.4.1、流模型中的绑定数据; SELECT/DML 和 PL/SQL(Binding Data in a Streaming Mode; SELECT/DML and PL/SQL)

To bind data in a streaming mode, follow these steps and review Example 3–4:
在一个流模型中绑定参数，参照下面的步骤并参阅Example 3–4:

1. Create a `SELECT/DML` or PL/SQL statement with appropriate bind placeholders.
创建一个带有合理占位符的`SELECT/DML`和PL/SQL语句。

2. Call the setBinaryStreamMode() or setCharacterStreamMode() method of the Statement Class for each bind position that will be used in the streaming mode. If the bind position is a PL/SQL `IN` or `IN/OUT` argument type, this by calling the three-argument versions of these methods and setting the `inArg` parameter to `TRUE`.
对流模型中用到的任意一个绑定位置调用Statement Class的setBinaryStreamMode()或者setCharacterStreamMode()方法。如果绑定位置是PL/SQL `IN`或者`IN/OUT`参数类型，通过调用这些方法的具有三个参数版本的方法并设置参数`inArg`的值为`TRUE`指出这种状况。

3. Execute the statement; the status() method of the Statement Class will return `NEEDS_STREAM_DATA`.
执行语句; Statement Class的status()方法将返回`NEEDS_STREAM_DATA`。

4. Obtain the stream object through a getStream() method of the Statement Class.
调用Statement Class的getStream()方法获取流对象。

5. Use writeBuffer() and writeLastBuffer() methods of the Stream Class to write data.
调用Stream Class的writeBuffer()或者writeLastBuffer()方法写入数据。

6. Close the stream with closeStream() method of the Statement Class.
调用Statement Class的closeStream()方法关闭流。

7. After all streams are closed, the status() method of the Statement Class will change to an appropriate value, such as `UPDATE_COUNT_AVAILABLE`.
所有流关闭后，Statement Class的status()方法的状态值将更改至一个合理的值，例如`UPDATE_COUNT_AVAILABLE`。

Example 3-4 How to Bind Data in a Streaming Mode
Example 3-4如何在流模型中绑定数据

```
Statement *stmt = conn->createStatement(
"Insert Into testtab(longcol) values (:1)"); //longcol is LONG type column
stmt->setCharacterStreamMode(1, 100000);
stmt->executeUpdate();
Stream *instream = stmt->getStream(1);
char buffer[1000];
instream->writeBuffer(buffer, len); //write data
instream->writeLastBuffer(buffer, len); //repeat
stmt->closeStream(instream); //stmt->status() is
//UPDATE_COUNT_AVAILABLE
Statement *stmt = conn->createStatement("BEGIN testproc(:1); END;");
//if the argument type to testproc is IN or IN/OUT then pass TRUE to
//setCharacterStreamMode or setBinaryStreamMode
stmt->setBinaryStreamMode(1, 100000, TRUE);
```

3.4.4.2、在流模型中获取数据: PL/SQL(Fetching Data in a Streaming Mode: PL/SQL)

To fetch data from a streaming mode, follow these steps and review Example 3–5:
在流模型中获取数据，请参照斜面步骤并参阅Example 3–5:

1. Create a `SELECT`/DML statement with appropriate bind placeholders.
   创建一个带有合理占位符的`SELECT`/`DML`和PL/SQL语句。

2. Call the setBinaryStreamMode() or setCharacterStreamMode() method of the Statement Class for each bind position into which data will be retrieved from the streaming mode.
对流模型中用到的任意一个获取数据的绑定位置调用Statement Class的setBinaryStreamMode()或者setCharacterStreamMode()方法。

3. Execute the statement; the status() method of the Statement Class will return `STREAM_DATA_AVAILABLE`.
执行语句; Statement Class的status()方法将返回`STREAM_DATA_AVAILABLE`。

4. Obtain the stream object through a getStream() method of the Statement Class.
调用Statement Class的getStream()方法获取流对象。

5. Use readBuffer() and readLastBuffer() methods of the Stream Class to read data.
调用Stream Class的readBuffer()或者readLastBuffer()方法读取数据。

6. Close the stream with closeStream() method of the Statement Class.
调用Statement Class的closeStream()方法关闭流。

Example 3-5 How to Fetch Data in a Streaming Mode Using PL/SQL
Example 3-5如何用PL/SQL流模型获取数据

```
Statement *stmt = conn->createStatement("BEGIN testproc(:1); END;");
//argument 1 is OUT type
stmt->setCharacterStreamMode(1, 100000);
stmt->execute();
Stream *outarg = stmt->getStream(1);
//use Stream::readBuffer/readLastBuffer to read data
```

3.4.4.3、在 ResultSet 结果集流模型中获取数据(Fetching Data in Streaming Mode: ResultSet)

Executing SQL Queries and Example 3–7 on page 3-16 provide an explanation of how to use the streaming interface with result sets.
执行3-16页上的SQL查询，它说明了如何使用流接口在结果集中获取数据

3.4.4.4、和多重流一起工作(Working with Multiple Streams)

If you have to work with multiple read and write streams, you have to ensure that the read or write of one stream is completed prior to reading or writing on another stream. To determine stream position, use the getCurrentStreamParam() method of the Statement Class or the getCurrentStreamColumn() method of the ResultSet Class. The status() method of the Stream Class will return `READY_FOR_READ` if there is data in the stream available for reading, or it will return `INACTIVE` if all the data has been read, as described in Table 12–44. The application can then read the next streaming column. Example 3–6 demonstrates how to read and write with two concurrent
streams.
如果你的工作中用到多重的读写流，你必须保证在读写另外一个流前读写一个流式完整的。确定当前流位置，用Statement Class的getCurrentStreamParam()方法或者ResultSet Class的getCurrentStreamColumn()方法。如果流中依然有可读的数据，Stream Class的status()方法将返回`READY_FOR_READ`，否则如果数据已经读取，将返回`INACTIVE`，如Table 12–44所描述。这时程序就可以读取下一列数据了。Example 3–6演示了如何同时在两个流中读取和写入数据。

Example 3-6 How to Read and Write with Multiple Streams

```
Statement *stmt = conn->createStatement(
"Insert into testtab(longcol1, longcol2) values (:1,:2)");
//longcol1 AND longcol2 are 2 columns inserted in streaming mode
```

```
stmt->setBinaryStreamMode(1, 100000);
stmt->setBinaryStreamMode(2, 100000);
stmt->executeUpdate();
Stream *col1 = stmt->getStream(1);
Stream *col2 = stmt->getStream(2);
col1->writeBuffer(buffer, len); //first stream
... //complete writing col1 stream
col1->writeLastBuffer(buffer, len); //finish first stream and move to col2
col2->writeBuffer(buffer, len); //second stream
//reading multiple streams
stmt = conn->createStatement("select longcol1, longcol2 from testtab");
ResultSet *rs = stmt->executeQuery();
rs->setBinaryStreamMode(1, 100000);
rs->setBinaryStreamMode(2, 100000);
while (rs->next())
{
Stream *s1 = rs->getStream(1)
while (s1->status() == Stream::READY_FOR_READ)
{
s1->readBuffer(buffer,size); //process
} //first streaming column done
rs->closeStream(s1);
//move onto next column. rs->getCurrentStreamColumn() will return 2
Stream *s2 = rs->getStream(2)
while (s2->status() == Stream::READY_FOR_READ)
{
s2->readBuffer(buffer,size); //process
} //close the stream
rs->closeStream(s2);
}
```

3.4.5、行更改迭代(Modifying Rows Iteratively)

While you can issue the `executeUpdate` method repeatedly for each row, OCCI provides an efficient mechanism for sending data for multiple rows in a single network round-trip. To do this, use the `addIteration()` method of the `Statement` class to perform batch operations that modify a different row with each iteration.
如果你遭遇了需要重复对每行执行`executeUpdate`操作，`OCCI`提供了在一个单独网络连接为多行发送数据的高效机制。你需要调用`Statement`的`addIteration()`方法来组合操作行的不同。

To execute INSERT, UPDATE, and DELETE operations iteratively, you must:
重复执行INSERT, UPDATE, 和 DELET操作，你需要：
- Set the maximum number of iterations 设置重复执行的最大次数
- Set the maximum parameter size for variable length parameters 设置长变参的参数最大长度

3.4.5.1、设置最大重复次数(Setting the Maximum Number of Iterations)

For iterative execution, first specify the maximum number of iterations that would be done for the statement by calling the setMaxIterations() method:
为了重读执行，首先需要调用setMaxIterations()方法确定最大执行次数：

```
Statement->setMaxIterations(int maxIterations);
```

You can retrieve the current maximum iterations setting by calling the getMaxIterations() method.
你可以调用getMaxIterations()获取当前最大迭代次数

3.4.5.2、设置参数最大长度(Setting the Maximum Parameter Size)

If the iterative execution involves variable length datatypes, such as string and Bytes, then you must set the maximum parameter size so that OCCI can allocate the maximum size buffer:
如果迭代执行涉及到变长数据类型，例如string和Bytes，这时你必须确定参数最大长度以便OCCI可以申请缓冲区的最大值。

```
Statement->setMaxParamSize(int parameterIndex, int maxParamSize);
```

You do not need to set the maximum parameter size for fixed length datatypes, such as Number and Date, or for parameters that use the setDataBuffer() method.
You can retrieve the current maximum parameter size setting by calling the getMaxParamSize() method.
你不需要设置长度固定数据类型的参数最大值，诸如Number和Date，或者使用setDataBuffer()方法的那些参数。你可以调用getMaxParamSize()方法获取当前参数最大值。

3.4.5.3、执行一个迭代操作(Executing an Iterative Operation)

Once you have set the maximum number of iterations and (if necessary) the maximum parameter size, iterative execution using a parameterized statement is straightforward, as shown in the following example:
一旦你确定了最大迭代次数和参数最大值，可以直接用一个带参数的语句执行迭代，如下所示:

```
stmt->setSQL("INSERT INTO basket_tab VALUES(:1, :2)");
stmt->setString(1, "Apples"); // value for first parameter of first row
stmt->setInt(2, 6); // value for second parameter of first row
stmt->addIteration(); // add the iteration
stmt->setString(1, "Oranges"); // value for first parameter of second row
```

```
stmt->setInt(2, 4); // value for second parameter of second row
stmt->executeUpdate(); // execute statement
```

As shown in the example, you call the `addIteration()` method after each iteration except the last, after which you invoke `executeUpdate()` method. Of course, if you did not have a second row to insert, then you would not need to call the `addIteration()` method or make the subsequent calls to the `set`*xxx*`()` methods.

正如例子中所示，你可以在除最后一句的任何迭代语句后调用`addIteration()`方法，因为最后一句涉及到`executeUpdate()`方法。当然，如果你没有第二行参数插入，你必须要调用`addIteration()`方法或者调用`set`*xxx*`()`来调用子过程。

3.4.5.4、执行迭代用法提示(Iterative Execution Usage Notes)

- Iterative execution is designed only for use in `INSERT`, `UPDATE` and `DELETE` operations that use either standard or parameterized statements. It cannot be used for callable statements and queries.

迭代执行是只为`INSERT`, `UPDATE`和`DELETE`操作或者用标准的或者参数化的语句进行设计的。他不能用于可调用语句和查询。

- The datatype cannot be changed between iterations. For example, if you use `setInt()` for parameter `1`, then you cannot use `setString()` for the same parameter in a later iteration.

在迭代中数据类型不可更改。例如，如果你用`setInt()`设置了参数1，这时你不能在以后的迭代中用`setString()`设置同样的参数。

3.5、执行 SQL 查询(Executing SQL Queries)

SQL query statements allow your applications to request information from a database based on any constraints specified. A result set is returned as a result of a query.

SQL查询语句允许你的程序请求基于任何特定约束的数据库的信息。查询的结果会以一个结果集返回。

3.5.1、使用结果集(Using the Result Set)

Execution of a database query puts the results of the query into a set of rows called the result set. In OCCI, a SQL `SELECT` statement is executed by the `executeQuery` method of the `Statement` class. This method returns an `ResultSet` object that represents the results of a query.

执行一个数据库查询，会把查询结果放进一个行的集合中，这个集合被称为结果集。在OCCI中，一个SQL查询`SELECT`语句通过调用`Statement`中的`executeQuery`方法来执行。这个方法返回一个`ResultSet`对象来体现查询的结果。

```
ResultSet *rs = stmt->executeQuery("SELECT * FROM basket_tab");
```

Once you have the data in the result set, you can perform operations on it. For example, suppose you wanted to print the contents of this table. The `next()` method of the `ResultSet` is used to fetch data, and the `get`*xxx*`()` methods are used to retrieve the individual columns of the result set, as shown in the following code example:
一旦你的结果集中有数据，你就可以在他上面执行操作。例如，假设你想打印表的内容。`ResultSet`的`next()`方法迎来获取数据，`get`*xxx*`()`方法迎来获取结果集中单独的列，正如下面事例所示那样:

```
cout << "The basket has:" << endl;
while (rs->next())
{
string fruit = rs->getString(1); // get the first column as string
int quantity = rs->getInt(2); // get the second column as int
cout << quantity << " " << fruit << endl;
}
```

The `next()` and `status()` methods of the `ResultSet` class return `Status`, as defined in Table 12–37.
`ResultSet`类的`next()`和`status()`方法返回`Status`，正如表Table 12–37定义的那样。

If data is available for the current row, then the status is `DATA_AVAILABLE`. After all the data has been read, the status changes to `END_OF_FETCH`. If there are any output streams to be read, then the status is `STREAM_DATA_AVAILABLE`, until all the streamed data are read succcessfully.
如果数据针对当前行使可用的，那么状态将是`DATA_AVAILABLE`。当所有的数据读取之后，状态更改为`END_OF_FETCH`。如果有任何的输出参数要读，那么状态将是`STREAM_DATA_AVAILABLE`，直至数据流被成功读取完毕。

Example 3–7 illustrates how to fetch streaming data into a result set, while section "Streamed Reads and Writes" on page 3-11 provides the general backgound.
Example 3–7 演示了如何获取数据里到一个结果集中，在3-11页的"Streamed Reads and Writes"段提供了这个基本背景。

Example 3–7 How to Fetch Data in Streaming Mode Using ResultSet

```
char buffer[4096];
ResultSet *rs = stmt->executeQuery
("SELECT col1, col2 FROM tab1 WHERE col1 = 11");
rs->setCharacterStreamMode(2, 10000);
while (rs->next ())
{
unsigned int length = 0;
unsigned int size = 500;
Stream *stream = rs->getStream (2);
while (stream->status () == Stream::READY_FOR_READ)
{
```

```
length += stream->readBuffer (buffer +length, size);
}
cout << "Read " << length << " bytes into the buffer" << endl;
}
```

3.5.2、特定查询(Specifying the Query)

The `IN` bind variables can be used with queries to specify constraints in the `WHERE` clause of a query. For example, the following program prints only those items that have a minimum quantity of `4`:
查询通过`WHERE`约束条款来敲定其输入变量。例如，下面的程序仅仅打印最小质量为`4` 的项：

```
stmt->setSQL("SELECT * FROM basket_tab WHERE quantity >= :1");
int minimumQuantity = 4;
stmt->setInt(1, minimumQuantity); // set first parameter
ResultSet *rs = stmt->executeQuery();
cout << "The basket has:" << endl;
while (rs->next())
cout << rs->getInt(2) << " " << rs->getString(1) << endl;
```

3.5.3、设置预处理事项优化性能(Optimizing Performance by Setting Prefetch Count)

Although the `ResultSet` method retrieves data one row at a time, the actual fetch of data from the server need not entail a network round-trip for each row queried. To maximize the performance, you can set the number of rows to prefetch in each round-trip to the server.
尽管`ResultSet`得方法每次获取一行数据，事实上从服务器获取数据并不需要针对每行查询引发一次网络来返。为了最大的提高效率，你可以在服务器上设置欲获取的行数。

You effect this either by setting the number of rows to be prefetched through the `setPrefetchRowCount()` method, or by setting the memory size to be used for prefetching through the `setPrefetchMemorySize()` method.
你可以使用`setPrefetchRowCount()`方法设置欲获取行数来使之生效，或者通过`setPrefetchMemorySize()`方法设置欲获取内存大小来使之生效。

If you set both of these attributes, then the specified number of rows are prefetched unless the specified memory limit is reached first. If the specified memory limit is reached first, then the prefetch returns as many rows as will fit in the memory space defined by the call to the `setPrefetchMemorySize()` method.
如果你同时设置了这两个属性，欲获取行数将首先考虑特定的内存的欲获取行数。如果首先达到了内存约束，这时欲获取行数将尽可能多的接近`setPrefetchMemorySize()`方法定义的数目。

By default, prefetching is turned on, and the database fetches an extra row all the time.To turn prefetching off, set both the prefetch row count and memory size to `0`.
默认情况下，欲获取是打开状态的，数据库一直都多获取额外的行。关闭欲获取，需要同时设置欲获取行数和内存都为0。

3.6、执行动态语句(Executing Statements Dynamically)

When you know that you need to execute a DML operation, you use the `executeUpdate` method. Similarly, when you know that you need to execute a query, you use `executeQuery()` method.
当你需要执行DML操作时，你可以用`executeUpdate()`方法。类似的，当你需要执行查询时，你可以用`executeQuery()`方法。

If your application needs to allow for dynamic events and you cannot be sure of which statement will need to be executed at run time, then OCCI provides the `execute()` method. Invoking the `execute()` method returns one of the following statuses:
如果你的程序需要允许动态，并且你在运行时不确定那条语句将要执行，这时OCCI提供了`execute()`方法。`execute()`方法执行返回下面状态的其中一个：

- UNPREPARED
- PREPARED
- RESULT_SET_AVAILABLE
- UPDATE_COUNT_AVAILABLE
- NEEDS_STREAM_DATA
- STREAM_DATA_AVAILABLE

While invoking the `execute()` method will return one of these statuses, you can also interrogate the statement by using the `status` method.
不仅仅是`execute()`返回这些状态中的一个，你还可以使用`status()`方法来获取。

```
Statement stmt = conn->createStatement();
Statement::Status status = stmt->status(); // status is UNPREPARED
stmt->setSQL("select * from emp");
status = stmt->status(); // status is PREPARED
```

If a statement object is created with a SQL string, then it is created in a `PREPARED` state. For example:
如果创建的一个语句是SQL字符串，这时它创建的是`PREPARED`状态。例如：

```
Statement stmt = conn->createStatement("insert into foo(id) values(99)");
Statement::Status status = stmt->status();// status is PREPARED
status = stmt->execute(); // status is UPDATE_COUNT_AVAILABLE
```

When you set another SQL statement on the Statement, the status changes to `PREPARED`. For example:
如果你需要在Statement设置另外一个SQL语句，状态将更改为`PREPARED`。例如：

```
stmt->setSQL("select * from emp"); // status is PREPARED
status = stmt->execute(); // status is RESULT_SET_AVAILABLE
```

3.6.1、状态定义(Status Definitions)

This section describes the possible values of `Status` related to a statement object:
本届描述和语句实体相关的尽可能的状态值:

- UNPREPARED
- PREPARED
- RESULT_SET_AVAILABLE
- UPDATE_COUNT_AVAILABLE
- NEEDS_STREAM_DATA
- STREAM_DATA_AVAILABLE

3.6.1.1、UNPREPARED

If you have not used the `setSQL()` method to attribute a SQL string to a statement object, then the statement is in an `UNPREPARED` state.
如果你没有使用`setSQL()`方法来属性化一个SQL字符串为语句实体，那么这时语句实体是`UNPREPARED`状态。

```
Statement stmt = conn->createStatement();
Statement::Status status = stmt->status(); // status is UNPREPARED
```

3.6.1.2、PREPARED

If a Statement is created with an SQL string, then it is created in a `PREPARED` state. For example:
如果语句实体通过一个SQL字符串来创建，那么它创建后将是`PREPARED`状态。例如：

```
Statement stmt = conn->createStatement("INSERT INTO demo_tab(id)
VALUES(99)");
Statement::Status status = stmt->status(); // status is PREPARED
```

Setting another SQL statement on the Statement will also change the status to `PREPARED`. For example:
设置另外一个SQL语句到语句实体，将会改变状态为`PREPARED`。例如：

```
status = stmt->execute(); // status is UPDATE_COUNT_AVAILABLE
stmt->setSQL("SELECT * FROM demo_tab"); // status is PREPARED
```

3.6.1.3、RESULT_SET_AVAILABLE

A status of `RESULT_SET_AVAILABLE` indicates that a properly formulated query has been executed and the results are accessible through a result set.

状态RESULT_SET_AVAILABLE用来表明一个执行合理的查询已经构造出，并且可以通过结果集来操作了。
When you set a statement object to a query, it is PREPARED. Once you have executed the query, the statement changes to RESULT_SET_AVAILABLE. For example:
当你设置了一个语句实体来查询，那么它是PREPARED状态。一旦你执行了这个查询，那么它的状态就转化为了RESULT_SET_AVAILABLE。例如：

```
stmt->setSQL("SELECT * from EMP"); // status is PREPARED
status = stmt->execute(); // status is RESULT_SET_AVAILABLE
```

To access the data in the result set, issue the following statement:
获取结果集中的数据，参照下面的指令：

```
ResultSet *rs = Statement->getResultSet();
```

3.6.1.4、UPDATE_COUNT_AVAILABLE

When a DDL or DML statement in a PREPARED state is executed, its state changes to UPDATE_COUNT_AVAILABLE, as shown in the following code example:
当执行一个处于PREPARED状态的DDL或者DML语句，它的状态就转换为UPDATE_COUNT_AVAILABLE，正如下面事例代码所示的那样：

```
Statement stmt = conn->createStatement("INSERT INTO demo_tab(id)
VALUES(99)");
Statemnt::Status status = stmt->status(); // status is PREPARED
status = stmt->execute(); // status is UPDATE_COUNT_AVAILABLE
```

This status refers to the number of rows affected by the execution of the statement. It indicates that:
这个状态设计到执行语句所影响的行数。它表明：

- The statement did not include any input or output streams. 结果集不包含任何输入或者输出流
- The statement was not a query but either a DDL or DML statement. 语句不是查询语句而是DDL或者DML语句。

You can obtain the number of rows affected by issuing the following statement:
你可以使用下面的指令或者所影响的行数:

```
Statement->getUpdateCount();
```

Note that a DDL statement will result in an update count of zero (0). Similarly, an update that does not meet any matching conditions will also produce a count of zero(0). In such a case, you cannot infer the kind of statement that has been executed from the reported status.
注意一个DDL语句所影响的更新行数为0。类似的，一个不匹配任何条件的更细将产生一个0行结果。在这种情况下，你不能论断这种语句已经从报告的状态上执行了。

3.6.1.5、NEEDS_STREAM_DATA

If there are any output streams to be written, the execute does not complete until all the stream data is completely provided. In this case, the status changes to

NEEDS_STREAM_DATA to indicate that a stream must be written. After writing the stream, call the status() method to find out if more stream data should be written, or whether the execution has completed.
如果有任何输出参数等待写入，这个执行将不会成功一直等到流数据已经被完整的提供。这种情况下，状态将更改为NEEDS_STREAM_DATA，来表明一个流必须被写入。在写入流后，调用status()方法会发现更多的流数据需要被写入，或者执行是否已经完成。

In cases where your statement includes multiple streamed parameters, use the getCurrentStreamParam() method to discover which parameter needs to be written. If you are performing an iterative or array execute, the getCurrentStreamIteration() method reveals to which iteration the data is to be written.
Once all the stream data has been processed, the status changes to either RESULT_SET_AVAILABLE or UPDATE_COUNT_AVAILABLE.
在你的语句包含更多的流参数情况下，用getCurrentStreamParam()方法来确定那个参数需要被写入。如果你在执行一个迭代或者数组，getCurrentStreamIteration()方法用来揭示那个迭代数据等待被写入。一旦所有的流数据被进行，那么状态将更改为RESULT_SET_AVAILABLE或者UPDATE_COUNT_AVAILABLE。

3.6.1.6、STREAM_DATA_AVAILABLE

This status indicates that the application requires some stream data to be read in OUT or IN/OUT parameters before the execution can finish. After reading the stream, call the status method to find out if more stream data should be read, or whether the execution has completed.
这个状态表明程序在执行完成前请求一些流数据被读入OUT 或者 IN/OUT参数。当读取完毕后，调用status函数会发现更多的流数据等待被写入，或者执行是否完成。

In cases in which your statement includes multiple streamed parameters, use the getCurrentStreamParam() method to discover which parameter needs to be read.
If you are performing an iterative or array execute, then the getCurrentStreamIteration() method reveals from which iteration the data is to be read.Once all the stream data has been handled, the status changes to UPDATE_COUNT_REMOVE_AVAILABLE. The ResultSet class also has readable streams and it operates similar to the readable streams of the Statement class.
在你的语句包含更多的流参数情况下，用getCurrentStreamParam()方法来确定那个参数需要被写入。如果你在执行一个迭代或者数组，getCurrentStreamIteration()方法用来揭示那个迭代数据等待被写入。一旦所有的流数据被处理，那么状态将更改为UPDATE_COUNT_REMOVE_AVAILABLE。ResultSet类同样有可读流，它的操作非常类似于Statement类的可读流。

3.7、提交事务(Committing a Transaction)

All SQL DML statements are executed in the context of a transaction. An application causes the changes made by these statement to become permanent by either committing the transaction, or undoing them by performing a rollback. While the SQL `COMMIT` and `ROLLBACK` statements can be executed with the `executeUpdate()` method, you can also call the `Connection::commit()` and `Connection::rollback()` methods.

所有的SQL DML语句在一个事务内容执行。一个程序通过这些语句引起的改变将在提交事务后变的稳定或者回滚事务后，不做更改。可以通过执行`executeUpdate()`方法来`COMMIT`提交和`ROLLBACK`回滚语句，你还可以通过调用`Connection::commit()`和`Connection::rollback()`方法。

If you want the DML changes that were made to be committed immediately, you can turn on the auto commit mode of the `Statement` class by issuing the following statement:

如果你想DML改变在提交后立即生效，你可以依照下面的指令`Statement`类的打开自动提交模式：

```
Statement::setAutoCommit(TRUE);
```

Once auto commit is in effect, each change is automatically made permanent. This is similar to issuing a commit right after each execution.To return to the default mode, auto commit off, issue the following statement:

一旦自动提交生效，每样更改会自动变为永久性的。这非常类似于在执行后正确的发布提交。如果要恢复默认模式，自动提交关闭，遵循下面的指令：

```
Statement::setAutoCommit(FALSE);
```

3.8、缓存语句(Caching Statements)

The statement caching feature establishes and manages a cache of statements within a session. It improves performance and scalability of application by efficiently using prepared cursors on the server side and eliminating repetitive statement parsing.

创建语句缓存特性和管理一个缓冲的语句在一个会话中。他能通过高效的使用服务器上的预定义游标和消除语句重复解析来提高性能和应用程序的灵活性。

Statement caching can be used with connection and session pooling, and also without connection pooling. Please review Example 3–8 and Example 3–9 for typical usage scenarios.

语句缓存可以和连接池、对话池一起使用，也可以没有连接池。请查阅Example 3–8 和Example 3–9的这种典型的情节。

Example 3–8 Statement Caching without Connection Pooling 没有连接池的语句缓存

These steps and accompanying pseudocode implement the statement caching feature without use of connection pools:

这些步骤和没有完成的伪指令实现了没有连接池的语句缓存特性：

1. Create a `Connection` by making a createConnection() call on the `Environment` object.

调用`Environment`对象的createConnection()建立一个连接`Connection`。

```
Connection *conn = env->createConnection(username, password, connecstr);
```

2. Enable statement caching on the `Connection` object by using a nonzero `size` parameter in the setStmtCacheSize() call.

调用setStmtCacheSize()并传入一个非0 size参数的值来使能`Connection`连接对象的语句缓存。

```
conn->setStmtCacheSize(10);
```

Subsequent calls to getStmtCacheSize() would determine the size of the cache,while setStmtCacheSize() call changes the size of the statement cache, or disables statement caching if the `size` parameter is set to zero.

随后调用getStmtCacheSize()可以确定缓存的大小，而setStmtCacheSize()将更改语句缓冲区的大小，或者通过设置size参数的值为0禁用语句缓存。

3. Create a `Statement` by making a createStatement() call on the `Connection` object; the `Statement` is returned if it is in the cache already, or a new `Statement` with a `NULL` tag is created for the user.

调用`Connection`连接对象的createStatement()方法来创建一个`Statement`实例；如果缓存区已经准备好将返回，否则一个带有`NULL`空标记的`Statement`实例将被创建并返回给用户：

```
Statement *stmt = conn->createStatement(sql);
```

To retrieve a previously cached tagged statement, use the alternate form of the createStatement() method:

来获取一个先前已经标记的缓存语句，使用更改过的下面的createStatement()。

```
Statement *stmt = conn->createStatement(sql, tag);
```

4. Use the statement to execute SQL commands and obtain results.

用statement来执行SQL命令并获取结果。

5. Return the statement to cache.　　返回语句到缓存中。

```
conn->terminateStatement(stmt, tag);
```

If you don't want to cache this statement, use the disableCaching() call and an alternate from of terminateStatement():

如果你不想缓存这些语句，可以用disableCaching()和terminateStatement()的变型。

```
stmt->disableCaching();
conn->terminateStatement(stmt);
```

If you need to verify whether a statement has been cached, issue an isCached() call on the `Connection` object.

如果你需要核实一个语句是否被缓存，可以是使用`Connection`连接对象的isCached()。

You can choose to tag a statement at release time and then re-use it for another statement with the same tag. The tag will be used to search the cache. An untagged statement, where tag is `NULL`, is a special case of a tagged statement.Two statements are considered different if they only differ in their tags, and if only one of them is tagged.

你可以在释放时选择标记一个语句，并且在以后重复使用同样的标记为另一个语句。这个标记将会选择缓存区。一个没有标记的语句，它的标记是NULL空的，是一个特殊类型的标记语句。有两种语句被认为是不同的，一种是它们的标记不同；另一种是它们是否被标记。

6. Terminate the connection. 关闭连接。

Example 3-9 Statement Caching with Connection Pooling 带有连接池的语句缓存

These steps and accompanying pseudocode implement the statement caching feature with connection pooling:

这些步骤和没有完成的伪指令实现了带有连接池的语句缓存特性：

1. Create a `ConnectionPool` by making a call to the createConnectionPool() of the `Environment` object.

调用`Environment`对象的createConnection()建立一个连接`Connection`。

```
ConnectionPool *conPool = env->createConnectionPool(username,
password,connecstr,minConn, maxConn, incrConn);
```

If using a `StatelessConnectionPool`, call createStatelessConnectionPool() instead. Subsequent operations are the same for `ConnectionPool` and `StatelessConnectionPool` objects.

```
Stateless ConnectionPool *conPool = env->createStatelessConnectionPool(
username, password, connecstr,minConn, maxConn, incrConn, mode);
```

如果使用的是无状态连接池`StatelessConnectionPool`，用createStatelessConnectionPool()来代替。随后的在`ConnectionPool`和`StatelessConnectionPool`上一样的。

2. Enable statement caching for all `Connection`s in the `ConnectionPool` by using a nonzero `size` parameter in the setStmtCacheSize() call.

调用setStmtCacheSize()并传入一个非0 size参数的值来使能`Connection`连接对象的语句缓存。

```
conPool->setStmtCacheSize(10);
```

Subsequent calls to getStmtCacheSize() would determine the size of the cache, while setStmtCacheSize() call changes the size of the statement cache, or disables statement caching if the `size` parameter is set to zero.

随后调用getStmtCacheSize()可以确定缓存的大小，而setStmtCacheSize()将更改语句缓冲区的大小，或者通过设置size参数的值为0禁用语句缓存。

3. Get a `Connection` from the pool by making a createConnection() call on the `ConnectionPool` object; the `Statement` is returned if it is in the cache already,or a new `Statement` with a `NULL` tag is created for the user.

调用`ConnectionPool`连接池对象的createConnection()方法来从连接池中获取一个`Connection`实例；如果缓存区已经准备好将返回，否则一个带有`NULL`空标记的`Statement`实例将被创建并返回给用户：

```
Connection *conn = conPool->createConnection(username, password, connecstr);
```

To retrieve a previously cached tagged statement, use the alternate form of the createStatement() method:

来获取一个先前已经标记的缓存语句，使用更改过的下面的createStatement()。

```
Statement *stmt = conn->createStatement(sql, tag);
```

4. Create a `Statement` by making a createStatement() call on the `Connection` object; the `Statement` is returned if it is in the cache already, or a new `Statement` with a `NULL` tag is created for the user.

调用`Connection`连接对象的createStatement()方法来创建一个`Statement`实例；如果缓存区已经准备好将返回，否则一个带有`NULL`空标记的`Statement`实例将被创建并返回给用户：

```
Statement *stmt = conn->createStatement(sql);
```

To retrieve a previously cached tagged statement, use the alternate form of the createStatement() method:

来获取一个先前已经标记的缓存语句，使用更改过的下面的createStatement()。

```
Statement *stmt = conn->createStatement(sql, tag);
```

5. Use the statement to execute SQL commands and obtain results. 用statement来执行SQL命令并获取结果。

6. Return the statement to cache. 返回语句到缓存中。

```
conn->terminateStatement(stmt, tag);
```

If you don't want to cache this statement, use the disableCaching() call and an alternate from of terminateStatement():

如果你不想缓存这些语句，可以用disableCaching()和terminateStatement()的变型。

```
stmt->disableCaching();
conn->terminateStatement(stmt);
```

If you need to verify whether a statement has been cached, issue an isCached() call on the `Connection` object.

如果你需要核实一个语句是否被缓存，可以是使用`Connection`连接对象的isCached()。

7. Release the connection terminateConnection(). 用terminateConnection()释放这个连接

```
conPool->terminateConnection(conn);
```

3.9、异常处理(Handling Exceptions)

Each OCCI method is capable of generating an exception if it is not successful. This exception is of type `SQLException`. OCCI uses the C++ Standard Template Library (STL), so any exception that can be thrown by the STL can also be thrown by OCCI methods.

每个OCCI函数在执行不成功时都会产生异常。这些异常是`SQLException`类型。`OCCI`使用了`C++`标准模板库，所以任何能够被`STL`跑出的异常也可以被`OCCI`函数抛出。

The STL exceptions are derived from the standard exception class. The `exception::what()` method returns a pointer to the error text. The error text is guaranteed to be valid during the catch block

The SQLException class contains Oracle specific error numbers and messages. It is derived from the standard exception class, so it too can obtain the error text by using the exception::what() method.
STL异常继承自标准异常类。exception::what()返回一个指向错误文本的指针。这些错误文本在捕捉区保证是可用的。SQLException类包含Oracle特殊错误码和信息。它继承自标准异常类，所以他同样可以用exception::what()来获取异常文本。

In addition, the SQLException class has two methods it can use to obtain error information. The getErrorCode() method returns the Oracle error number. The same error text returned by exception::what() can be obtained by the getMessage() method. The getMessage() method returns an STL string so that it can be copied like any other STL string.
除此之外，SQLException还有两个函数可用来获取错误信息。getErrorCode()返回Oracle错误码。错误文本可由getMessage()返回。getMessage()返回一个STL字符串，所以它可以像另外一个STL字符串一样被拷贝。

Based on your error handling strategy, you may choose to handle OCCI exceptions differently from standard exceptions, or you may choose not to distinguish between the two.
基于你的错误处理策略，你可以选择不同于标准异常的OCCI异常处理，或者你选择不区别它们两个。

If you decide that it is not important to distinguish between OCCI exceptions and standard exceptions, your catch block might look similar to the following:
如果你确定区别OCCI异常和标准异常不重要，你的捕捉区看起来应该像下面所示样子:

```
catch (exception &excp)
{
cerr << excp.what() << endl;
}
```

Should you decide to handle OCCI exceptions differently than standard exceptions, your catch block might look like the following:
你需要区别OCCI异常和标准异常的不同，你的捕捉区看起来应该像下面所示样子:

```
catch (SQLException &sqlExcp)
{
cerr <<sqlExcp.getErrorCode << ": " << sqlExcp.getErrorMessage() << endl;
}
catch (exception &excp)
{
cerr << excp.what() << endl;
}
```

In the preceding catch block, SQL exceptions are caught by the first block and non-SQL exceptions are caught by the second block. If the order of these two blocks were to be reversed, SQL exceptions would never be caught. Since

SQLException is derived from the standard exception, the standard exception catch block would handle the SQL exception as well.
在捕捉处理区，SQL异常应该放在首位，而非SQL异常放在第二位。如果这两者位置颠倒，SQL异常将不会被捕捉。因为SQLException继承自标准异常，表春异常捕捉将处理SQL异常。

3.9.1、处理空的 NULL 和截断的数据(Handling Null and Truncated Data)

In general, OCCI does not cause an exception when the data value retrieved by using the get*xxx*() methods of the ResultSet class or Statement class is NULL or truncated. However, this behavior can be changed by calling the setErrorOnNull() method or setErrorOnTruncate() method. If the setError*xxx*() methods are called with causeException=TRUE, then an SQLException is raised when a data value is NULL or truncated.
事实上，OCCI通过调用ResultSet类的get*xxx*()方法来获取数据值将不会产生异常或者Statement是控制或者截除的。尽管如此，调用setErrorOnNull()或者setErrorOnTruncate()将使行为产生改变。如果和causeException=TRUE一起调用setError*xxx*()方法，那么一个SQLException将在数据值为NULL时或者截除时抛出。

The default behavior is not to raise an SQLException. A column or parameter value can also be NULL, as determined by a call to isNull() for a ResultSet or Statement object returning TRUE:
默认的行为时不抛出一个SQLException。一列或者参数值均可以为NULL。可用调用ResultSet或者Statement独享的isNull()返回TRUE来确定。

```
rs->isNull(columnIndex);
stmt->isNull(paramIndex);
```

If the column or parameter value is truncated, it will also return TRUE as determined by a isTruncated() call on a ResultSet or Statement object:
如果一列或者参数是被截除的，那么调用ResultSet或者Statement类的isTruncated()方法来确定将会返回TRUE。

```
rs->isTruncated(columnIndex);
stmt->isTruncated(paramIndex);
```

For data retrieved through the setDataBuffer() method and setDataBufferArray() method, exception handling behavior is controlled by the presence or absence of indicator variables and return code variables as shown in Table 3–1, Table 3–2, and Table 3–3.
因为数据重新获取使用setDataBuffer()和setDataBufferArray()方法，异常处理行为被出席的或者缺席的指示器变量控制，并且返回的代码标量如Table 3–1, Table 3–2, Table 3–3所示。

In Table 3–3, data_len is the actual length of the data that has been truncated if this length is less than or equal to SB2MAXVAL. Otherwise, the indicator is set to -2.
在Table 3–3，data_len是实际的数据长度，如果这个长度小于或者等于SB2MAXVAL将被截除。否则，指示器将被设置为-2。