OCI 学习资料

--Oracle8 及以后版本的 OCI

1. 简介

Oracle 调用接口(Oracle Call Interface)是一个让我们通过函数调用来访问 Oracle 数据库和控制 SQL 语句执行各个阶段的应用程序编程接口(API)。OCI 支持 C 和 C++的数据类型、调用惯例、语法和语义。

1.1 创建一个 OCI 应用程序

我们编译和连接一个 OCI 程序的方法与编译和连接一个非数据库应用程序的方法相同。不需要独立的预处理或者预编译步骤。

1.2 OCI 的组成部分

OCI 具有如下功能：

- 能够安全地支持大量用户的灵活的、多线程 API 集合。
- 为管理数据库访问、处理 SQL 语句和管理 Oracle 数据库对象的 SQL 访问函数。
- 管理 Oracle 类型的数据属性的数据类型映射和操作函数。
- 不经 SQL 语句直接向数据库加载数据的数据加载函数。

1.3 封装的接口

所有的 OCI 函数使用的数据结构都以被称为句柄的不透明的接口之形式封装。句柄是指向 OCI 库分配的保存着上下文信息、连接信息、错误信息或者关于 SQL 及 PL/SQL 的绑定信息的不透明指针。客户端分配一定类型的句柄，通过已经定义好的接口来填充一个或者多个句柄，并通过这些句柄向服务器发送请求。应用程序可以通过访问函数来访问句柄中包含的相关信息。

2. OCI 基本编程

这部分介绍 OCI 编程中涉及到的基本概念。

2.1 OCI 编程概要

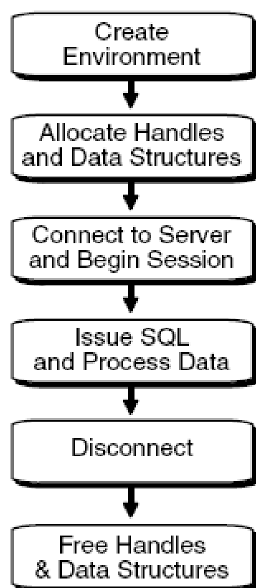
一个 OCI 应用程序的基本目标就是代表多个用户进行数据库操作。

OCI 使用以下基本编程顺序：

1. 初始化 OCI 编程环境和线程。
2. 分配必要的句柄，并且设置服务器连接和用户会话。
3. 通过在服务器上执行 SQL 语句来交换数据，并且执行必要的应用程序数据处理。
4. 执行准备好的语句或者准备即将要执行的语句。
5. 终止用户会话并且断开服务器连接。
6. 释放句柄。

图 2-1 显示了一个 OCI 应用程序的编程步骤。

图 2-1



这幅图及其所列出的步骤提供了一个 OCI 编程步骤的简单概括。根据程序的功能，变化是完全可能发生的。包含管理多个会话、事务和使用对象的更复杂的 OCI 应用程序需要另外的步骤。

所有的 OCI 函数调用都在一个环境中执行。在一个 OCI 进程中可以有多个环境。如果一个环境需要任何进程级别的初始化，则其自动进行。

注意：在一个 OCI 应用程序中可以有多个活动连接和语句。

2.2 OCI 数据结构

句柄(Handles)和描述符(descriptors)是 OCI 应用程序中定义的不透明的数据结构。它们可以被直接分配、通过特殊的分配函数或者可以被 OCI 函数隐式地分配。

7.X 升级注意：以前写过 7.XOCI 应用程序的程序员必须熟悉这些被大多数 OCI 函数使用的数据结构。

句柄和描述符保存有关数据、连接、或者应用程序行为的信息。

2.3 句柄

几乎每一个 OCI 函数的参数列表中都包含有一个或者多个句柄。一个句柄是一个指向一个 OCI 库分配的存储区域的不透明的指针。我们使用句柄来保存上下文信息或者连接信息，(例如，一个环境或者服务句柄)，或者它可以保存关于 OCI 函数或者数据的信息(例如，一个错误句柄或者描述句柄)。句柄可以使编程更简单，因为 OCI 库会维持这些数据而不是应用程序。

大多数 OCI 应用程序需要使用句柄中的信息。获取属性值和设置属性值的 OCI 函数，OCIAttrGet()和 OCIAttrSet()，获取和设置这些信息。

表 2-1 列出了 OCI 中定义的句柄。并列出了与每一种句柄类型相对应的 C 数据类型和 OCI 函数中使用的用来识别 OCI 调用中的句柄类型的句柄类型常量。

表 2-1 OCI 句柄类型

Description	C Data Type	Handle Type Constant
OCI environment handle	OCIEnv	OCI_HTYPE_ENV
OCI error handle	OCIError	OCI_HTYPE_ERROR
OCI service context handle	OCISvcCtx	OCI_HTYPE_SVCCTX
OCI statement handle	OCIStmt	OCI_HTYPE_STMT
OCI bind handle	OCIBind	OCI_HTYPE_BIND
OCI define handle	OCIDefine	OCI_HTYPE_DEFINE
OCI describe handle	OCIDescribe	OCI_HTYPE_DESCRIBE
OCI server handle	OCIServer	OCI_HTYPE_SERVER
OCI user session handle	OCISession	OCI_HTYPE_SESSION
OCI authentication information handle	OCIAuthInfo	OCI_HTYPE_AUTHINFO
OCI connection pool handle	OCICPool	OCI_HTYPE_CPOOL
OCI session pool handle	OCISPool	OCI_HTYPE_SPOOL
OCI transaction handle	OCITrans	OCI_HTYPE_TRANS
OCI complex object retrieval (COR) handle	OCIComplexObject	OCI_HTYPE_COMPLEXOBJECT
OCI thread handle	OCIThreadHandle	Not applicable
OCI subscription handle	OCISubscription	OCI_HTYPE_SUBSCRIPTION
OCI direct path context handle	OCIDirPathCtx	OCI_HTYPE_DIRPATH_CTX
OCI direct path function context handle	OCIDirPathFuncCtx	OCI_HTYPE_DIRPATH_FN_CTX
OCI direct path column array handle	OCIDirPathColArray	OCI_HTYPE_DIRPATH_COLUMN_ARRAY
OCI direct path stream handle	OCIDirPathStream	OCI_HTYPE_DIRPATH_STREAM
OCI process handle	OCIProcess	OCI_HTYPE_PROC
OCI administration handle	OCIAdmin	OCI_HTYPE_ADMIN
OCI HA event handle	OCIEvent	Not applicable

分配和释放句柄

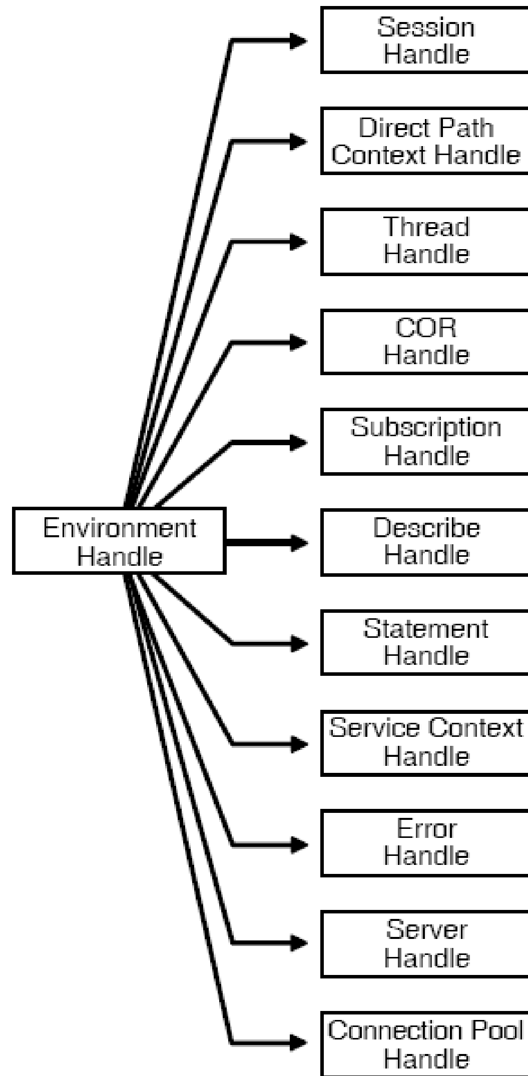
我们的应用程序要为一个特定的环境句柄分配所有的句柄(除了绑定句柄 bind handle、定义句柄 define handle 和线程句柄 thread handles)。我们把环境句柄作为一个参数传递至句柄分配函数。然后，被分配的句柄就特定于那个特定的环境。

绑定句柄和定义句柄是为一个语句句柄分配的，并且包含了关于那个句柄所代表的语句的信息。

注意：绑定句柄和定义句柄是由 OCI 库隐式分配的，不需要用户分配。

图 2-2 显示了不同类型的句柄的层次

图 2-2 句柄的层次



环境句柄是由 OCIEnvCreate()或者 OCIEnvNlsCreate()函数分配和初始化的，所有的 OCI 程序都需要执行它们中的任一个。

所有的用户分配的句柄都通过 OCI 句柄分配函数来初始化，即 OCIHandleAlloc()函数。

句柄类型包括：线程句柄、描述句柄、语句句柄、服务上下文句柄、错误句柄和服务器句柄等。

线程句柄通过 OCIThreadHndInit()函数分配。

一个应用程序必须释放所有的不再使用的句柄。OCIHandleFree()函数释放所有的句柄。

注意：当一个父句柄被释放后，所有与之相连的子句柄也被释放并且再也不能被使用。例如，当一个语句句柄释放后，任何与之相连的绑定和定义句柄也都被释放。

句柄减少了对全局变量的需要。句柄也使错误报告更容易。一个错误句柄用来返回错误和诊断信息。

2.3.1环境句柄

环境句柄提供了一个所有的 OCI 函数被调用的上下文。每一个环境句柄包含一个支持快速访问的内存缓存。所有的环境句柄下的内存分配都是通过这个缓存完成的。如果多个线程想要在同一个环境句柄下分配内存，则它们对缓存的访问是序列化的。当多个线程共享一个单独的环境句柄时，它们会阻塞对缓存的访问。

环境句柄作为 OCIHandleAlloc()函数的 parent 参数来分配其他句柄类型。绑定句柄和定

义句柄是隐式分配的。

2.3.2错误句柄

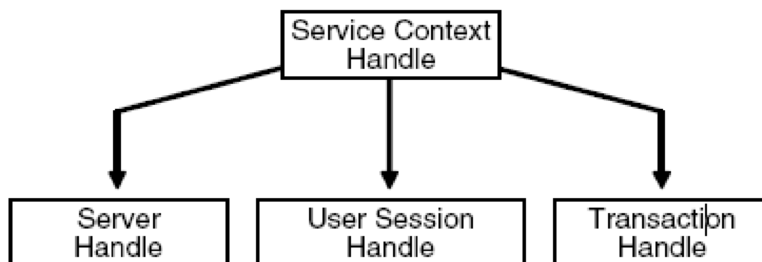
错误句柄作为一个参数传递至大多数 OCI 函数。错误句柄维持着关于一个 OCI 操作中所发生的错误的信息。如果一个函数调用中发生了一个错误，错误句柄可以传给 OCIErrorGet()函数来获取关于那个错误的额外信息。

由于大多数 OCI 函数都需要一个错误句柄作为一个参数，分配错误句柄是一个 OCI 应用程序中的前几个步骤之一。

2.3.3 服务上下文句柄和相关的句柄

一个服务上下文句柄定义了决定 OCI 调用的上下文的属性。服务上下午句柄包含 3 个句柄作为它的属性，分别代表一个服务器连接、一个用户会话和一个事务。图 2-3 列出了这些属性

图 2-3



• 一个服务器句柄 Server Handle 识别对一个数据库的连接。

• 一个用户会话句柄 User Session Handle 定义了一个用户的角色和权限以及函数执行的操作上下文。

• 一个事务句柄定义了 SQL 操作所属的事务。

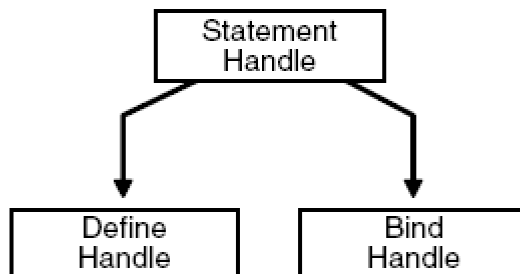
在需要更复杂的会话管理的应用程序中，服务上下午句柄必须被显式地分配，并且服务器句柄和用户会话句柄必须通过 OCIAttrSet()函数被显式地设置到服务上下午句柄中。OCIServerAttach()函数和 OCISessionBegin()函数分别初始化服务器和用户会话句柄。

当应用程序对数据库做修改时，OCI 自动隐式分配的事务句柄正确地工作。

2.3.4 语句句柄、绑定句柄和定义句柄

一个语句句柄是识别 SQL 或者 PL/SQL 语句以及与之相连的属性的上下文，如图 2-4 所示

图 2-4



关于输入绑定变量和输出绑定变量的信息储存于绑定句柄中 bind handle。OCI 库为每一个与 OCIBindByName()函数或者 OCIBindByPos()函数相连的占位符分配一个绑定句柄。用户一定不可以分配绑定句柄。它们是由绑定函数隐式分配的。

查询返回的数据根据定义句柄(define handle)的描述来转换和获取。OCI 库为每一个

OCIDefineByPos()函数中所定义的输出变量分配一个定义句柄。用户一定不可以分配定义句柄。它们是由定义函数隐式分配的。

绑定句柄和定义句柄是由 OCI 库显式分配的，并且如果绑定或者定义操作是重复的话，绑定句柄和定义句柄是被重复使用的。当语句句柄被释放或者当一个新的语句被准备到语句句柄上后，绑定句柄或者定义句柄就被释放。显式分配绑定或者定义句柄可能导致内存泄露。显式释放绑定句柄或者定义句柄可能导致程序异常终止。

2.3.5 描述句柄

描述句柄 describe handle 由 OCI 描述函数所使用，OCIDescribeAny()。这个函数获取关于数据库中的数据库对象的信息(例如,函数或者过程)。这个函数有一个参数是描述句柄，以及关于被描述的对象的附加信息。当函数执行完成后，描述句柄就存有关于那个对象的信息。然后 OCI 应用程序可以通过参数描述符的属性获取描述信息。

2.3.6 句柄属性

所有的 OCI 句柄都有表示存储在那个句柄中的数据的属性。我们可以通过属性获取函数 OCIAttrGet()来获取句柄属性值，并且我们可以通过 OCIAttrSet()函数来设置句柄的属性值。

例如，下面的代码显示了通过写入会话句柄的 OCI_ATTR_USERNAME 属性来设置会话句柄中的用户名。

```
text    username[] = “hr”;
err = OCIAttrSet((dvoid*)mysessp,        OCI_HTYPE_SESSION,     (dvoid*)username,
               (ub4)strlen((char*)username),       OCI_ATTR_USERNAME,  (OCIError*)myerrhp);
```

一些 OCI 函数要求在这些函数被调用前，一些特定的句柄属性值被设置。例如，调用 OCISessionBegin()函数来设置用户的登录会话，在这个函数被调用之前，必须设置用户会话句柄的用户名和密码属性。

其他的 OCI 函数执行完成后，会在句柄属性中提供有用的返回数据。例如，当 OCIStmtExecute()函数被调用来执行 SQL 查询时，关于所选字段的描述信息被返回到语句句柄中。

```
ub4    parmcnt;
/*获取选择列表中的字段数量*/
Err = OCIAttrGet((dvoid*)stmhp,    (ub4)OCI_HTYPE_STMT,        (dvoid*)&parmcnt, (ub4)0,
                        (ub4)OCI_ATTR_PARM_COUNT,            errhp);
```

2.3.7 OCI 描述符

OCI 描述符 descriptor 和定位符 locator 是维护特殊数据信息的不透明数据结构。表 2-2 列出了它们以及对应的 C 数据结构和还有在 OCIDescriptorAlloc()函数中所用到的 OCI 类型常量。OCIDescriptorFree()函数释放描述符和定位符。

表 2-2

Description	C Data Type	OCI Type Constant
Snapshot descriptor	OCISnapshot	OCI_DTYPE_SNAP
Result set descriptor	OCIResult	OCI_DTYPE_RSET
LOB data type locator	OCILobLocator	OCI_DTYPE_LOB
BFILE data type locator	OCILobLocator	OCI_DTYPE_FILE
Read-only parameter descriptor	OCIParam	OCI_DTYPE_PARAM
ROWID descriptor	OCIRowid	OCI_DTYPE_ROWID
ANSI DATE descriptor	OCIDateTime	OCI_DTYPE_DATE
TIMESTAMP descriptor	OCIDateTime	OCI_DTYPE_TIMESTAMP
TIMESTAMP WITH TIME ZONE descriptor	OCIDateTime	OCI_DTYPE_TIMESTAMP_TZ
TIMESTAMP WITH LOCAL TIME ZONE descriptor	OCIDateTime	OCI_DTYPE_TIMESTAMP_LTZ
INTERVAL YEAR TO MONTH descriptor	OCIInterval	OCI_DTYPE_INTERVAL_YM
INTERVAL DAY TO SECOND descriptor	OCIInterval	OCI_DTYPE_INTERVAL_DS
User callback descriptor	OCIUcb	OCI_DTYPE_UCB
Distinguished names of the database servers in a registration request	OCIServerDNs	OCI_DTYPE_SRVDN
Complex object descriptor	OCIComplexObjectComp	OCI_DTYPE_COMPLEXOBJECTCOMP
Advanced queuing enqueue options	OCIAQEnqOptions	OCI_DTYPE_AQENQ_OPTIONS
Advanced queuing dequeue options	OCIAQDeqOptions	OCI_DTYPE_AQDEQ_OPTIONS
Advanced queuing message properties	OCIAQMsgProperties	OCI_DTYPE_AQMSG_PROPERTIES
Advanced queuing agent	OCIAQAgent	OCI_DTYPE_AQAGENT
Advanced queuing notification	OCIAQNotify	OCI_DTYPE_AQNFY
Advanced queuing listen options	OCIAQListenOpts	OCI_DTYPE_AQLIS_OPTIONS
Advanced queuing message properties	OCIAQLisMsgProps	OCI_DTYPE_AQLIS_MSG_PROPERTIES
Change notification	None	OCI_DTYPE_CHDES
Table change	None	OCI_DTYPE_TABLE_CHDES
Row change	None	OCI_DTYPE_ROW_CHDES

下面描述了各种描述符类型的主要目的。

- OCISnapshot—用于语句执行
- OCILobLocator—用于 LOB(OCI_DTYPE_LOB)或者 BFILE(OCI_DTYPE_FILE)函数
- OCIParam—用于描述函数
- OCIRowid—用于绑定或者定义 ROWID 值

2.3.7.1 LOB 和 BFILE 定位符

一个大对象(LOB)是保存二进制大数据或者字符大对象(CLOB)数据的 Oracle 数据类型。在数据库中，一个不透明的叫做 LOB 定位符的数据结构保存在一个数据库行的 LOB 字段中。定位符作为实际 LOB 值的指针，实际的 LOB 值保存在其他的地方。

OCI LOB 定位符用于对 LOB(BLOB 或者 CLOB)类型或者 FILE(BFILE)类型进行 OCI 操作。OCILobXXX 函数使用 LOB 定位符作为参数而不是 LOB 数值。OCI LOB 函数不使用实际的 LOB 数据作为参数。它们使用 LOB 定位符作为参数并且通过引用它们来操作 LOB 数据。

通过向 OCIDescriptorAlloc()函数传递 OCI_DTYPE_LOB 作为类型参数来为 BLOBs 或者 CLOBs 分配 LOB 定位符，传递 OCI_DTYPE_FILE 作为类型参数为 BFILEs 分配 LOB 定位符。

注意：这两种 LOB 定位符类型是不能互换的。当绑定或者定义一个 BLOB 或者 CLOB 时，应用程序必须注意定位符是否是通过 OCI_DTYPE_LOB 分配的。同样，当绑定或者定义一个 BFILE 时，应用程序必须确保使用 OCI_DTYPE_FILE 分配定位符。

一个 OCI 应用程序可以使用一个包含有 LOB 字段或者属性的 SQL 语句从 Oracle 数据库

获取LOB定位符。这种情况下，应用程序需要首先分配LBO定位符然后用它来定义一个输出变量。与此类似，一个LOB定位符可以被用来作为绑定操作的一部分以创建一个LOB和一个SQL语句中的占位符的联结。

2.3.7.2 参数描述符

OCI应用程序使用参数描述符获取关于所选字段或者数据库对象的信息。通过描述操作获取这种信息。

参数描述符是唯一的不通过 OCIDescriptorAlloc()函数分配的描述符。我们只有通过描述句柄、语句句柄或者复杂对象获取句柄来获取它，使用OCIParamGet()函数。

3. OCI编程步骤

下面描述开发一个OCI应用程序的细节。

3.1 OCI环境初始化

这部分描述如何初始化OCI环境，设立一个服务器的连接，并且授权一个用户对数据库执行操作。

首先，初始化OCI环境时有三个主要步骤：

- **创建OCI环境**
- **分配句柄和描述符**
- **应用程序初始化、连接和创建会话**

3.1.1 创建OCI环境

每一个OCI函数调用都是在OCIEnvCreate()函数所创建的环境中执行的。这个函数必须在其他OCI函数执行前被调用。唯一的例外是设置OCI共享模式的进程级属性。

OCIEnvCreate()函数的模式(mode)参数指定应用程序是否能够：

- 运行在多线程环境中(mode=OCI_THREADED).
- 使用对象(mode=OCI_OBJECT).Use the AQ subscription registration

程序也可以选择不使用这些模式而使用默认模式(mode=OCI_DEFAULT)或者它们的联合，使用垂直条(|)将它们分开。例如，如果mode=(OCI_THREADED|OCI_OBJECT)，然后应用程序运行在多线程环境中并且使用对象。

我们可以为每个OCI环境指明用户定义的内存管理函数。

3.1.2 分配句柄和描述符

Oracle数据库提供了用于分配和释放句柄及描述符的OCI函数。在向OCI函数传递句柄之前我们必须使用OCIHandleAlloc()函数分配句柄，除非OCI函数，比如OCIBindByPos()，自动为我们分配句柄。

我们使用OCIHandleAlloc()函数来分配表2-1列出的句柄。

3.1.3 应用程序初始化、连接和创建会话

一个应用程序必须调用OCIEnvCreate()函数来初始化OCI环境句柄。

沿着这一步，应用程序有两个选项来设置一个服务器连接和开始一个用户会话：单用户、单连接或者多会话多连接。

注意：应该使用OCIEnvCreate()函数而不是OCIInitialize()函数和OCIEnvInit()函数。OCIInitialize()和OCIEnvInit()函数是向后兼容的函数。

3.1.3.1 单用户、单连接

如果应用程序在任何时刻仅为各个数据库连接维持一个单用户会话，这个选项就是简化的登录函数。

当一个应用程序调用OCILogon()函数时，OCI库初始化传递给它的服务上下午句柄，并且创建一个到用户指定的数据库的连接。

下面的例子显示了OCILogon()函数的用法：

OCILogon(envhp, errhp, &svchp, (text*) “hr”, nameLen, (text*) “hr”, passwdLen, (text*) “oracledb”, dbnameLen);

这个函数的参数包括服务上下午句柄(它将被初始化)，用户名，用户的密码，还有用来设置连接的数据库名。这个函数会隐式地分配服务器句柄和用户会话句柄。

如果应用程序使用这个登录函数，则服务上下文句柄、服务器句柄和用户会话句柄将成为只读的。应用程序不能通过改变服务上下午句柄的相关属性来转换会话或事务。

使用 OCILogon()函数初始化它的会话和认证的应用程序必须调用 OCILogoff()函数来终止它们。

3.1.3.2 多会话或者多连接

这个选项使用显式的服务器连接和开始会话函数来维持一个数据库连接之上的多用户会话和连接。服务器连接和开始会话的函数分别是：

・OCIServerAttach()—创建一个到数据源执行 OCI 操作的访问路径。

・OCISessionBegin()—为一个用户设置一个针对某一特定服务器的会话。为了在数据库服务器上执行数据库操作，必须执行这个函数。

这两个函数设置一个能够使我们执行 SQL 或者 PL/SQL 语句的执行环境。

3.1.4 在 OCI 中处理 SQL 语句

在 OCI 中处理 SQL 语句的细节在后面描述。

3.2 提交或者回滚事务

应用程序通过调用 OCITransCommit()函数来提交对数据库的修改。这个函数使用一个服务上下文句柄作为它的一个参数。与此服务上下文句柄相连的事务将被提交。应用程序可以显式地创建事务或者当应用程序修改数据库时隐式地创建事务。

注意：如果使用 OCIExecute()函数的 OCI_COMMIT_ON_SUCCESS 模式，应用程序可以在每个语句执行后选择性地提交事务，这样可以节省一个额外的网络开销。

使用 OCITransRollback()函数来回滚事务。

如果一个应用程序以非正常方式同 Oracle 断开连接，比如网络连接断开，并且 OCITransCommit()函数没有被调用，则所有活动的事务都会被自动回滚。

3.3 终止应用程序

一个 OCI 程序需要在它终止前执行如下步骤：

1. 调用 OCISessionEnd()函数删除每一个会话的用户会话。
2. 调用 OCIServerDetach()函数删除每一个数据源的访问。
3. 调用 OCIHandleFree()函数来释放每一个句柄。
4. 删除环境句柄，环境句柄会释放所有与之相连的句柄。

注意：当一个父句柄被释放后，任何与其相连的句柄都会被自动释放。

对 OCIServerDetach()函数和 OCISessionEnd()函数的调用不是必须的，但是建议进行调用。如果应用程序终止并且没有调用 OCITransCommit()函数没有被调用，则任何之前的事务都会被自动回滚。

3.4 OCI 中的错误处理

OCI 函数有一个返回代码集合，表 2-3 列出了 OCI 返回代码，这些返回代码指出了函数的成功或者失败，比如 OCI_SUCCESS 或者 OCI_ERROR，或者应用程序需要的其他信息，比如 OCI_NEED_DATA 或者 OCI_STILL_EXECUTING。大多数 OCI 函数会返回这些返回代码中的一个。

表 2-3 OCI 返回代码

OCI Return Code	Description
OCI_SUCCESS	The function completed successfully.
OCI_SUCCESS_WITH_INFO	The function completed successfully; a call to `OCIErrorGet()` returns additional diagnostic information. This may include warnings.
OCI_NO_DATA	The function completed, and there is no further data.
OCI_ERROR	The function failed; a call to `OCIErrorGet()` returns additional information.
OCI_INVALID_HANDLE	An invalid handle was passed as a parameter or a user callback is passed an invalid handle or invalid context. No further diagnostics are available.
OCI_NEED_DATA	The application must provide runtime data.
OCI_STILL_EXECUTING	The service context was established in nonblocking mode, and the current operation could not be completed immediately. The operation must be called again to complete. `OCIErrorGet()` returns ORA-03123 as the error code.
OCI_CONTINUE	This code is returned only from a callback function. It indicates that the callback function wants the OCI library to resume its normal processing.

如果返回代码提示发生了一个错误，应用程序可以调用 OCIErrorGet()函数来获取 Oracle 的错误代码和信息。OCIErrorGet()函数的一个参数是引起错误的错误句柄。

数据的返回代码和错误代码

表 2-4 列出了当数据获取时正常、为空或者被截短时的 OCI 返回代码。错误号码、指示器变量和字段返回代码。

表 2-4 返回和错我代码

State of Data	Return Code	Indicator - not provided	Indicator - provided
not null or truncated	not provided	OCI_SUCCESS error = 0	OCI_SUCCESS error = 0 indicator = 0

State of Data	Return Code	Indicator - not provided	Indicator - provided
not null or truncated	provided	OCI_SUCCESS error = 0 return code = 0	OCI_SUCCESS error = 0 indicator = 0 return code = 0
null data	not provided	OCI_ERROR error = 1405	OCI_SUCCESS error = 0 indicator = -1
null data	provided	OCI_ERROR error = 1405 return code = 1405	OCI_SUCCESS error = 0 indicator = -1 return code = 1405
truncated data	not provided	OCI_ERROR error = 1406	OCI_ERROR error = 1406 indicator = data_len
truncated data	provided	OCI_SUCCESS_WITH_INFO error = 24345 return code = 1405	OCI_SUCCESS_WITH_INFO error = 24345 indicator = data_len return code = 1406

3.5 编码建议 Coding Guidelines

这部分描述编写 OCI 应用程序时，一些附加的问题。

3.5.1参数类型 Parameter Types

OCI 函数使用很多不同类型的参数，包括整数、句柄和字符串。

地址参数 Address Parameters

地址参数用来将变量的地址传递至 Oracle 服务器。

整型参数 Integer Parameters

二进制整数和短的二进制整型参数的大小依赖于系统。

字符串参数 Character String Parameters

字符串参数是一种特殊类型的地址参数。每个需要一个字符串参数的 OCI 函数都需要一个字符串长度参数。

设置一个字段值为空 Inserting Nulls into a Column

我们可以有多种方式向一个数据库字段中插入空值。

1． 在 INSERT 或者 UPDATE 语句中使用 NULL。例如，下面的 SQL 语句

```
INSERT INTO emp (ename, empno, deptno)
       VALUES (NULL, 8010, 20)
```

会使 ENAME 字段为 NULL。

2． 在 OCI 绑定函数中使用指示器变量。

3.5.2 指示器变量 Indicator Variables

每一个定义和绑定OCI函数都有一个参数与一个指示器变量相连或者一个指示器变量数组相连。

C 语言没有 NULL 值的概念，因此我们将一个输入变量 input variable 联系到指示器变量

indicator variable 来指明占位符 place holder 是否为 NULL 值。当数据传到 Oracle 时，这些指示器变量的值决定一个数据库字段是否被设为空值。

对于输出变量 output variable，指示器变量指示 Oracle 返回的值是否为 NULL 或者被截短。在 OCIStmtFetch()函数获取到 NULL 或者 OCIStmtExecute()函数执行时出现 truncation 时，OCI 函数返回 OCI_SUCCESS。输出指示器变量被赋值。

指示器变量的类型是 sb2。在指示器变量数组中，单独的指示器元素为 sb2 类型。

3.5.2.1 输入 Input

对于输入宿主变量，OCI 应用程序可以像指示器变量赋予如下值：

表 3-5 输入指示器变量值 Input Indicator Values

Input Indicator Value	Action Taken by Oracle
-1	Oracle assigns a NULL to the column, ignoring the value of the input variable.
>=0	Oracle assigns the value of the input variable to the column.

当指示器变量值为-1 时，Oracle 忽略输入变量的值，将字段值设置为 NULL。

当指示器变量值大于等于 0 时，Oracle 将指示器变量的值赋给字段。

3.5.2.2 输出 Output

输出时，Oracle 可以赋予指示器变量如下值：

表 3-6 输出指示器变量值 Output Indicator Values

Output Indicator Value	Meaning
-2	The length of the item is greater than the length of the output variable; the item has been truncated. Additionally, the original length is longer than the maximum data length that can be returned in the sb2 indicator variable.
-1	The selected value is null, and the value of the output variable is unchanged.
0	Oracle assigned an intact value to the host variable.
>0	The length of the item is greater than the length of the output variable; the item has been truncated. The positive value returned in the indicator variable is the actual length before truncation.

指示器变量值为-2 时，字段值的长度大于输出变量的长度；字段值被截短。另外，字段值的原始长度大于 sb2 类型的指示器变量所能保存的最大数值。

指示器变量值为-1 时，所选字段的值为 NULL，并且输出变量的值不变。

指示器变量值为 0 时，Oracle 将完整的字段值赋给了宿主变量。

指示器变量值大于 0 时，字段值的长度大于输出变量的长度，字段被截短。指示器变量中返回的正数是字段值被截短前的真实长度。

3.6

4 数据类型

这部分介绍 OCI 应用程序中使用的 Oracle 外部数据类型的参考。也讨论 Oracle 数据类型以及当我们的应用程序和 Oracle 传递数据时繁盛的内部数据类型和外部数据类型之间的转换。

4.1 Oracle 数据类型

一个 OCI 程序的主要目的之一就是同一个 Oracle 服务器通信。OCI 应用程序可以通过 SQL SELECT 查询从数据库表中获取数据，或者通过 INSERT、UPDATE、或者 DELETE 语句修改表中的数据。

在数据库内部，值保存在表的字段中。在内部，Oracle 通过特定格式的内部数据类型 internal datatypes 来表示数据。内部数据类型包括 NUMBER、CHAR 和 DATE。

通常，OCI 应用程序不与数据的内部数据类型工作，而是与宿主主演预先定义的数据类型工作。当数据在 OCI 应用程序和数据库表中传递时，OCI 库在内部数据类型和外部数据类型之间进行转换。

外部数据类型 external datatypes 是宿主语言在 OCI 头文件中定义的数据类型。当一个 OCI 应用程序绑定输入变量时，绑定参数中有一个参数用来说明变量的外部数据类型代码 (SQLT 代码)。同样地，当定义调用中指明输出变量时，必须说明获取到的数据的外部表示。

OCI 可以进行大范围的 Oracle 和 OCI 应用程序间的数据类型转换。OCI 外部数据类型多于 Oracle 内部数据类型。

4.1.1 使用外部数据类型代码 External Datatype Codes

外部数据类型代码告知 Oracle 我们应用程序中的宿主变量被如何表示。这决定了当数据被返回到输出变量时，如何进行转换或者输入变量的数据如何转换成 Oracle 的字段值。例如，如果我们想要将一个 Oracle 字段中的 NUMBER 转换成变长字符数组，我们需要在 OCIDefineByPos()函数中指定 VARCHAR2 外部数据类型代码。

为转换绑定变量到一个 Oracle 字段值，需要指明对应于绑定变量的外部数据类型代码。例如，如果我们想要输入一个字符串比如 02-FEB-65 到一个 DATE 字段，将数据类型指定为字符串并且将长度参数设置为 9.

4.2 内部数据类型

表 4-1 列出了 Oracle 的内部数据类型，以及各个类型的最大内部长度和类型代码。

表 4-1 Oracle 内部数据类型

Internal Oracle Datatype	Maximum Internal Length	Datatype Code
VARCHAR2, NVARCHAR2	4000 bytes	1
NUMBER	21 bytes	2
LONG	2^31-1 bytes (2 gigabytes)	8
DATE	7 bytes	12
RAW	2000 bytes	23
LONG RAW	2^31-1 bytes	24
ROWID	10 bytes	69
CHAR, NCHAR	2000 bytes	96
BINARY_FLOAT	4 bytes	100
BINARY_DOUBLE	8 bytes	101
User-defined type (object type, VARRAY, Nested Table)	N/A	108
REF	N/A	111
CLOB, NCLOB	128 terabytes	112
BLOB	128 terabytes	113
BFILE	maximum operating system file size	114
TIMESTAMP	11 bytes	180
TIMESTAMP WITH TIME ZONE	13 bytes	181
INTERVAL YEAR TO MONTH	5 bytes	182
INTERVAL DAY TO SECOND	11 bytes	183
UROWID	3950 bytes	208
TIMESTAMP WITH LOCAL TIME ZONE	11 bytes	231

4.3 外部数据类型

表 4-2 列出了外部数据类型的类型代码。为每一种数据类型，表中都列出了 Oracle 的内部数据被正常转换为的 C 语言中的程序变量类型。

表 4-2 外部数据类型和代码

EXTERNAL DATATYPE	CODE	PROGRAM VARIABLE	OCI DEFINED CONSTANT
VARCHAR2	1	char[n]	SQLT_CHR
NUMBER	2	unsigned char[21]	SQLT_NUM
8-bit signed INTEGER	3	signed char	SQLT_INT
16-bit signed INTEGER	3	signed short, signed int	SQLT_INT
32-bit signed INTEGER	3	signed int, signed long	SQLT_INT
FLOAT	4	float, double	SQLT_FLT
NULL-terminated STRING	5	char[n+1]	SQLT_STR
VARNUM	6	char[22]	SQLT_VNU
LONG	8	char[n]	SQLT_LNG
VARCHAR	9	char[n+sizeof(short integer)]	SQLT_VCS
DATE	12	char[7]	SQLT_DAT
VARRAW	15	unsigned char[n+sizeof(short integer)]	SQLT_VBI

EXTERNAL DATATYPE	CODE	PROGRAM VARIABLE	OCI DEFINED CONSTANT
native float	21	float	SQLT_BFLOAT
native double	22	double	SQLT_BDOUBLE
RAW	23	unsigned char[n]	SQLT_BIN
LONG RAW	24	unsigned char[n]	SQLT_LBI
UNSIGNED INT	68	unsigned	SQLT_UIN
LONG VARCHAR	94	char[n+sizeof(integer)]	SQLT_LVC
LONG VARRAW	95	unsigned char[n+sizeof(integer)]	SQLT_LVB
CHAR	96	char[n]	SQLT_AFC
CHARZ	97	char[n+1]	SQLT_AVC
ROWID descriptor	104	OCIRowid *	SQLT_RDD
NAMED DATATYPE	108	struct	SQLT_NTY
REF	110	OCIRef	SQLT_REF
Character LOB descriptor	112	OCILobLocator (see note 2)	SQLT_CLOB
Binary LOB descriptor	113	OCILobLocator (see note 2)	SQLT_BLOB
Binary FILE descriptor	114	OCILobLocator	SQLT_FILE
OCI STRING type	155	OCIString	SQLT_VST (see note 1)
OCI DATE type	156	OCIDate *	SQLT_ODT (see note 1)
ANSI DATE descriptor	184	OCIDateTime *	SQLT_DATE
TIMESTAMP descriptor	187	OCIDateTime *	SQLT_TIMESTAMP
TIMESTAMP WITH TIME ZONE descriptor	188	OCIDateTime *	SQLT_TIMESTAMP_TZ
INTERVAL YEAR TO MONTH descriptor	189	OCIInterval *	SQLT_INTERVAL_YM
INTERVAL DAY TO SECOND descriptor	190	OCIInterval *	SQLT_INTERVAL_DS
TIMESTAMP WITH LOCAL TIME ZONE descriptor	232	OCIDateTime *	SQLT_TIMESTAMP_LTZ

4.3.1 VARCHAR2

VARCHAR2 数据类型是一个变长字符串，其最大长度为 4000 字节。

输入 Input

OCIBindByName()函数或者 OCIBindByPos()函数中的 value_sz 参数决定了可变长字符串的长度。

如果 value_sz 参数大于 0，Oracle 从程序中的缓冲区地址开始读取其所指定的字节数。如果 value_sz 参数为 0，Oracle 将绑定变量看做 NULL，不考虑其真实内容。如果我们向一个具有 NOT NULL 约束的字段插入 NULL 值，Oracle 会报错，并且改行不会被插入。

输出 Output

在 OCIDefineByPos()函数的 value_sz 参数中指定想要获取的返回值的长度。如果 value_sz 为 0，则没有数据被返回。

为检测返回值是否为空或者字符截短是否发生，就要在 OCIDefineByPos()函数中使用指示器变量参数。如果我们没有指定指示器变量参数并且返回一个空值，获取函数会返回错误代码 OCI_SUCCESS_WITH_INFO。

4.3.2 STRING

NULL 结尾的 String 格式类似于 VARCHAR2 格式，除了 string 格式会包含一个 NULL 终止符。这个数据类型对 C 语言非常有用。

输入 Input

如果 NULL 终止符没有在指定的长度中发现，Oracle 会产生如下错误

ORA-01480: trailing NULL missing from STR bind value

如果没有指定长度，则 OCI 使用隐含的最大长度—4000.

输出 Output

在被返回的最后一个字符后面添加一个 NULL 终止符。如果字符串超出了指定的字段长度，则其会被截短并且输出变量的最后一个字符包含 NULL 终止符。

4.3.3 DATE

Date 以二进制格式保存，包含 7 个字节，如表 4-3 所示。

表 4-3 Data 数据类型的格式

Byte	1	2	3	4	5	6	7
Meaning	Century	Year	Month	Day	Hour	Minute	Second
Example (for 30-NOV-1992, 3:17 PM)	119	192	11	30	16	18	1

4.3.4 LOB 定位符 Locator

当一个 OCI 应用程序发出的 SQL 查询中包括一个 LOB 字段时，查询返回的是 LOB 定位符，而不是实际的 LOB 字段值。在 OCI 中，LOB 定位符映射 OCILobLocator 类型的变量。

OCI 中的 LOB 相关函数使用一个 LOB 定位符作为它们的参数。这些 OCI 函数假定 LOB 定位符已经被创建，不论其指向的 LOB 是否含有数据。

绑定操作和定义操作都是对 LOB 定位符进行操作，通过 OCIDescriptorAlloc()函数来分配 LOB 定位符。

绑定和定义 LOB 的数据类型代码如下：

- SQLT_BLOB—一个二进制 LOB 数据类型。
- SQLT_CLOB—一个字符型 LOB 数据类型。

5 在 OCI 程序中使用 SQL 语句

这一节讨论使用 OCI 处理 SQL 语句是所涉及的概念和步骤。

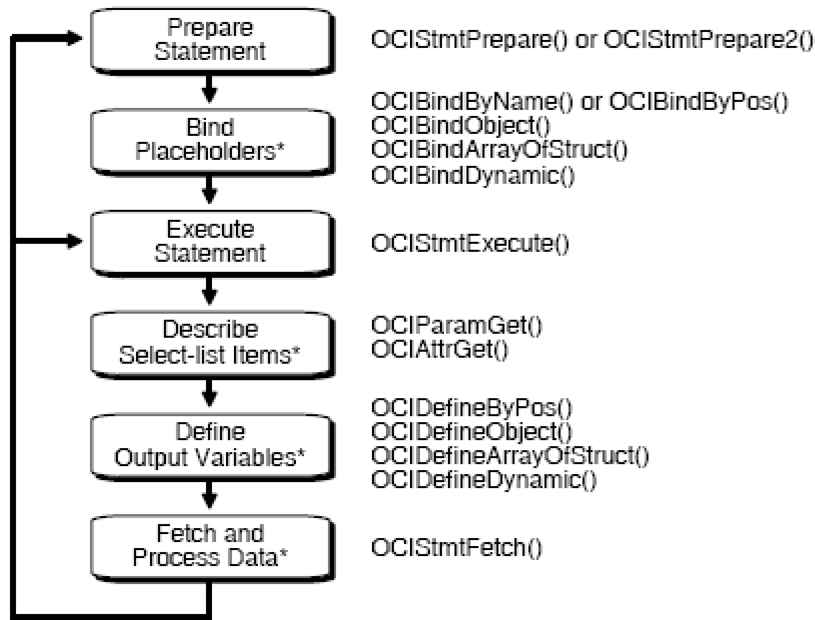
5.1 SQL 语句处理概要

一个 OCI 程序的通常任务就是接受和处理 SQL 语句。这部分概述一下 SQL 处理中所涉及

的步骤。

一旦我们分配了必要的句柄并且连接至服务器，按照图 5-1 中的步骤来处理 SQL 语句。

图 5-1 处理 SQL 语句的步骤



1. 准备语句。使用 OCIStmtPrepare()函数或者 OCIStmtPrepare2()函数来定义一个应用程序请求。
2. 如果有必要的话，需要绑定占位符。对于 DML 语句和有输入变量的查询，要执行使用以下的函数执行一个或者多个绑定。
   - OCIBindByPos()
   - OCIBindByName()
   - OCIBindObject()
   - OCIBindDynamic()
   - OCIBindArrayOfStruct()
3. 也可以使用 OCIStmtPrepare2()函数来准备一个语句进行执行。OCIStmtPrepare2()函数是 OCIStmtPrepare()函数的增强版，其引入了对语句缓存的支持。
4. 执行。调用 OCIStmtExecute()函数来执行语句。对于 DDL 语句则不需要其他步骤。
5. 如有必要，需要进行描述。如有必要使用 OCIParamGet()函数和 OCIAttrGet()函数来描述所选字段。这是一个可选步骤。
6. 如有必要，需要进行定义。对于查询，执行一个或者多个 OCIDefineByPos()、OCIDefineObject()、OCIDefineDynamic()或者 OCIDefineArrayOfStruct()函数来为 SQL 语句中的各个所选字段定义输出变量。
7. 如有必要，需要进行获取操作。对于查询，调用 OCIStmtFetch()来获取查询的结果。

按照这些步骤，应用程序可以释放已经分配的句柄，然后断开同服务器的连接，或者处理另外的 SQL 语句。

5.2 准备语句

通过使用语句准备函数和任何必要的绑定函数，SQL 和 PL/SQL 语句为执行做好准备。

应用程序指明一个 SQL 或者 PL/SQL 语句并且将语句中的占位符绑定至执行是所用的数据。客户端库为准备的语句分配空间。

一个应用程序调用 OCIStmtPrepare()函数来请求一个 SQL 或者 PL/SQL 语句为执行做准备并且向其传递一个之前分配的语句句柄。这是一个本地调用，不需要同服务器的通讯。此时，在语句和特定的服务器间没有联结。

在 OCI 中绑定占位符

大多数 DML 语句，以及一些查询(比如那些带有 Where 子句的)，需要程序将 SQL 或者 PL/SQL 中的一部分数据传递给 Oracle。这个数据可以使常量或者字符串，在我们的程序编译时已经是已知的。例如，下面的项数据库中增加一条员工记录的 SQL 语句：

```
INSERT INTO emp VALUE
    (2365, 'BESTRY', 'PROGRAMER', 2000, 20)
```

5.3 执行语句

一个 OCI 应用程序通过调用 OCIStmtExecute()函数来执行已经准备好的语句。

当一个 OCI 应用程序执行查询时，它从数据库中获取那些符合查询条件的数据。在数据库内部，数据以 Oracle 定义的格式储存。当结果返回后，OCI 应用程序可以请求将数据转换成一种特定的宿主语言格式，并将其储存在一个特定的输出变量或者缓冲区中。

对于查询中的所选字段中的每一个字段，OCI 程序都必须定义一个接收查询到的结果的输出变量。定义步骤说明了缓冲区的地址和将要获取到的数据的类型。

注意：如果一个 SELECT 语句的输出变量在调用 OCIStmtExecute()函数前进行了定义，OCIStmtExecute()函数中的 iters 参数指明的行数将会直接获取到已经定义的输出缓冲区中，并且等于预获取行数的其他行会被预获取到。如果没有其他行，则不需要调用 OCIStmtFetch()就可以完成获取操作。

5.4 获取查询结果

如果一个 OCI 应用程序已经处理了一个查询。通常会在语句执行完成后调用 OCIStmtFetch()函数或者 OCIStmtFetch2()函数来获取查询结果。Oracle 鼓励使用支持可滚动游标 scrollable cursors 的 OCIStmtFetch2()函数。

获取到的数据被放置到已经在定义操作中指明的输出变量中。

6 绑定和定义操作

6.1 OCI 中的绑定概述

本小节扩展绑定和定义的基本概念，并且提供我们在 OCI 应用程序中可以使用的绑定和定义操作的更详细的信息。另外，本节将讨论结构体数组的使用，以及其他的设计绑定、定义和字符转换的议题。

例如，如下 INSERT 语句

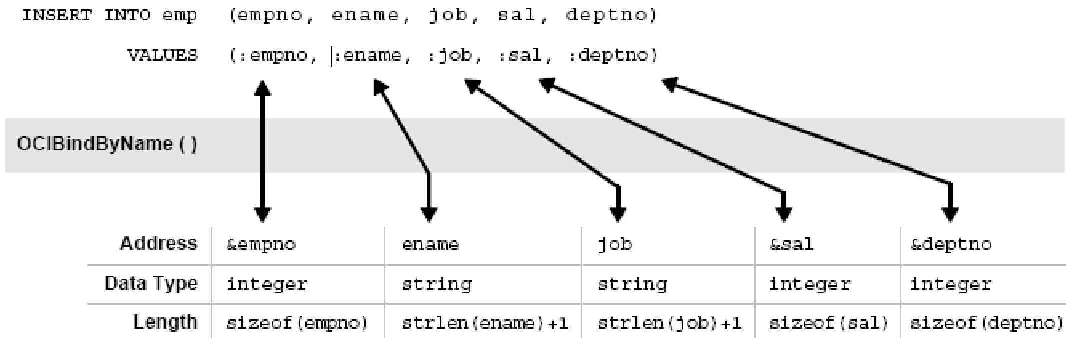
```
INSERT INTO emp VALUES
        (:empno, :ename, :job, :sal, :deptno)
```

还有如下变量声明

```
text *ename, *job;
sword empno, sal, deptno;
```

绑定步骤再占位符名字和程序变量之间建立结合。绑定也说明了程序变量的数据类型和长度，如图 6-1 所示

图 6-1 使用 OCIBindByName()函数来结合占位符和程序变量



如果我们只改变绑定变量的值，则没有必要为了重新执行语句而进行重新绑定。因为绑定是按引用绑定，只要变量地址和句柄地址保持不变，我们就可以重新执行引用那个变量的语句而不需进行重新绑定。

6.1.1 按名字绑定和按位置绑定

上述例子是一个按名字绑定的例子。语句中的每个占位符都有一个名字与其相结合，比如'ename'或者'sal'。当这个语句已经准备好并且占位符已经和应用程序中的一个变量值相结合时，结合是通过 OCIBindByName()函数设立的。

第二种绑定是按位置进行绑定。在按位置绑定中，通过占位符在语句中的位置而不是名字来进行绑定。为了进行绑定，使用 OCIBindByPos()函数对输入值和占位符的位置建立结合。

使用之前的例子进行按位置绑定：

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

这 5 个占位符分别通过调用 OCIBindByPos()函数进行绑定，并且将占位符在语句中的位置传递到 OCIBindByPos()函数的位置 position 参数中。例如，调用 OCIBindByPos()函数并传递位置参数 1，:empno 占位符将会被绑定，:ename 则传递位置参数 2。

在重复绑定的情况下，仅有一个单独绑定调用是必要的。考虑如下的 SQL 语句，这个语句查询数据库中提成和收入均大于某一数量的员工。

```
SELECT empno FROM emp
    WHERE sal > :some_value
    AND comm > :some_value
```

通过调用一次 OCIBindByName()函数进行按名字绑定，OCI 程序就可以完成该语句的绑定操作。在这种情况下，第二个占位符继承第一个占位符的绑定信息。

6.1.2 OCI 绑定的步骤

绑定占位符需要多个步骤。下面的例子显示了绑定一个 SQL 语句中的各个占位符的句柄分配和绑定。

```
/*  通过调用 OCIStmtPrepare()函数，将 SQL 语句结合到 stmthp(语句句柄)*/
text* insert = (text*) “INSERT INTO emp (empno, ename, job, sal, deptno) \
    VALUES (:empno, :ename, :job, :sal, :deptno)”;
. . .
/* 绑定 SQL 语句中的占位符，每一个占位符都需要一个绑定句柄*/
checker(errhp, OCIBindByName( stmthp, &bnd1p, errhp, (text *) ":ENAME",
    strlen(":ENAME"), ( ub1 *) ename, enamelen+1, SOLT_STR, (dvoid *) 0,
    (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT));
checkerr(errhp, OCIBindByName( stmthp, &bnd2p, errhp, (text *) ":JOB",
    strlen(":JOB"), (ub1 *) job, joblen+1, SQLT_STR, (dvoid *)&job_ind,
```

```
        (ub2 *) 0,      (ub2) 0,      (ub4) 0,      (ub4 *) 0,          OCI_DEFAULT));
    checkerr(errhp,    OCIBindByName(   stmthp,    &bnd3p,     errhp,    (text *) ":SAL",
        strlen(":SAL"),      (ub1 *) &sal,      (sword) sizeof(sal),       SQLT_INT,
        (dvoid *) &sal_ind,     (ub2 *) 0,     (ub2) 0,      (ub4) 0,      (ub4 *) 0,
        OCI_DEFAULT));
    checkerr(errhp,    OCIBindByName(   stmthp,    &bnd4p,    errhp,   (text *) ":DEPTNO",
        strlen(":DEPTNO"),      (ub1 *) &deptno,   (sword) sizeof(deptno),      SQLT_INT,
        (dvoid *) 0,    (ub2 *) 0,      (ub2) 0,   (ub4) 0,   (ub4 *) 0,       OCI_DEFAULT));
    checkerr(errhp,    OCIBindByName(   stmthp,   &bnd5p,      errhp, (   text *) ":EMPNO",
        strlen(":EMPNO"),       (ub1 *) &empno,    (sword) sizeof(empno),       SQLT_INT,
        (dvoid *) 0,    (ub2 *) 0,      (ub2) 0,   (ub4) 0,   (ub4 *) 0,      OCI_DEFAULT));
```

6.1.3 OCI 绑定 LOBs

有两种方式来绑定 LOBs。

• 绑定 LOB 定位符 locator,而不是实际的 LOB 值。在这种情况下，LOB 数值是通过向 OCILOB 函数传递 LOB 定位符完成的。

• 直接绑定 LOB 数值，而不是使用 LOB 定位符。

绑定 LOB 定位符

一个单独的定位符或者一个定位符数组都可以在一个单独的绑定调用中绑定。在这两种情况下，应用程序必须传递 LOB 定位符的地址而不是定位符本身。例如，如果一个应用程序已经准备了一个 SQL 语句。

```
INSERT  INTO   some_table  VALUES  (:one_lob)
```

one_lob 是一个与 LOB 字段对应的绑定变量，并且已经做出了如下声明：

```
OCILobLocator * one_lob;
```

然后，下面的调用将会被用来绑定占位符和执行语句。

```
/* 初始化单个定位符*/
One_lob  =  OCIDescriptorAlloc(…OCI_DTYPE_LOB…);
. . .
/* 传递定位符的地址*/
OCIBindByName(…(dvoid*)&one_lob, . . .SQLT_CLOB, . . .);
OCIStmtExecute(…, 1, …);  /* 1 是迭代参数*/
```

在描述符可以使用之前，我们必须调用 OCIDescriptorAlloc()函数来分配描述符。

6.2 OCI 定义概述

查询语句将从数据库中获取的数据返回到我们的应用程序中。当处理一个查询的时候，我们必须为所选字段中的每一个字段定义一个输出变量。定义步骤决定了返回值储存在什么地方和以什么形式储存。

例如，如果我们的应用程序处理如下语句，我们需要定义两个输出变量，一个用来接收 ename 字段返回的值，一个用来接收 ssn 字段返回的数值。

```
SELECT    name,     ssn    FROM    employess
    WHERE   empno   =    :empnum
```

如果我们只对获取 name 字段值感兴趣，则我们可以不为 ssn 字段定义输出变量。

OCI 在客户端本地处理定义调用。除了说明保存结果的缓冲区的位置，定义步骤还决定了当数据返回到应用程序时如何进行数据转换。

OCIDefineByPos()函数的 dty 参数说明输出变量的数据类型。当数据获取至输出变量时，OCI 能够执行大范围的数据转换。例如，Oracle 的 DATE 类型的内部数据可以自动转换为 String

数据类型。

6.2.1 OCI 定义的步骤

通过调用 OCIDefineByPos()函数来完成基本的定义。这一步建立了所选字段和输出变量的结合。

下面的例子显示了在执行完成后一个输出变量被定义。

```
SELECT department_name FROM departments WHERE department_id = :dept_input
/*这个输入变量已经被绑定，并且其数据来自于下面的用户输入*/
Printf("Enter employee dept:");
Scanf("%d", &deptno);
/* 执行语句。如果 OCIStmtExecute()返回 OCI_NO_DATA，则表示没有数据匹配该查询，然后该部门号即为无效部门号*/
if ( (status = OCIStmtExecute(svchp, stmthp, errhp, 0, 0, (OCISnapshot *) 0,
(OCISnapshot *) 0, OCI_DEFAULT)) && (status != OCI_NO_DATA))
{
    checkerr(errhp, status);
    return OCI_ERROR;
}
if (status == OCI_NO_DATA) {
    printf("The dept you entered doesn't exist.\n");
    return 0;
}
/* 下面的语句描述所选字段dname，并且返回它的数据长度*/
checkerr(errhp, OCIParamGet((dvoid *)stmthp, (ub4) OCI_HTYPE_STMT, errhp,
(dvoid **)&parmdp, (ub4) 1));
checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
(dvoid*) &deptlen, (ub4 *) &sizelen, (ub4) OCI_ATTR_DATA_SIZE,
(OCIError *) errhp ));
/* 使用返回的dname的长度来分配输出缓冲区，并且定义输出变量，如果定义函数返回错误则退出应用程序 */
dept = (text *) malloc( (int) deptlen + 1 );
if (status = OCIDefineByPos(stmthp, &defnp, errhp, 1, (dvoid *) dept,
(sb4) deptlen+1, SQLT_STR, (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, OCI_DEFAULT))
{
    checkerr(errhp, status);
    return OCI_ERROR;
}
```

6.2.2 OCI 定义 LOB 输出变量

有两种方式定义 LOBs:

• 定义一个 LOB 定位符，而不是真实的 LOB 数值。这种情况下，OCI LOB 函数通过 LOB 定位符来读取或者写入 LOB 数值。

• 不使用 LOB 定位符，直接定义一个 LOB 值。

定义 LOB 定位符

应用程序必须传递 LOB 定位符的地址而不是定位符本身。例如，如果应用程序已经准

备了下面的 SQL 语句。

SELECT lob1 FROM some_table;

其中 lob1 是 LOB 字段并且 one_lob 是对应于 LOB 字段的定义变量。其声明如下：

OCILobLocator* one_lob;

下面的步骤绑定占位符，并执行语句。

```
/*初始化单个定位符*/
One_lob = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
...
/*传递定位符的地址*/
OCIDefineByPos(...1, ..., (dvoid*) &one_lob, ...SQLT_CLOB, . . .);
OCIStmtExecute(..., 1, ...);
```

在描述符可以使用之前，我们必须调用 OCIDescriptorAlloc()函数来分配描述符。

6.2.3

6.3

7 OCI 相关函数

这一节介绍 Oracle 的 OCI 相关函数。

7.1 连接、认证和初始化函数

这一小节描述 OCI 的连接 connect、授权 authorize 和初始化 initialize 函数。表 7-1 列出了这些函数

表 7-1 连接、授权和初始化函数

Function	Purpose
OCIConnectionPoolCreate() on page 15-5	Initializes the connection pool.
OCIConnectionPoolDestroy() on page 15-8	Destroys the connection pool.
OCIEnvCreate() on page 15-9	Creates and initializes an OCI environment.
OCIEnvNlsCreate() on page 15-14	Creates and initializes an environment for OCI functions to work under. Allows you to set character set id and national character set id at environment creation time.
OCIEnvInit() on page 15-12	Initialize an environment handle.
OCIInitialize() on page 15-18	Initialize OCI process environment.
OCILogon() on page 15-22	Simplified single-session logon.
OCILogon2() on page 15-24	This function is used to create a logon session in various modes.
OCIServerAttach() on page 15-27	Attach to a server; initialize server context handle.
OCIServerDetach() on page 15-30	Detach from a server; uninitialize server context handle.
OCISessionBegin() on page 15-31	Authenticate a user.
OCISessionEnd() on page 15-35	Terminate a user session.
OCISessionGet() on page 15-36	Get a session from a session pool.
OCISessionPoolCreate() on page 15-40	Initializes a session pool.
OCISessionPoolDestroy() on page 15-44	Destroys a session pool.
OCISessionRelease() on page 15-45	Releases a session.
OCITerminate() on page 15-47	Detaches from a shared memory subsystem.

7.1.1 OCIEnvCreate()函数

作用：为执行 OCI 函数的执行创建和初始化一个环境。

原型：

```
Sword    OCIEnvCreate(    OCIEnv                    **envhp,
                          Ub4                       mode,
                          CONST    dvoid            *ctxp,
                          CONST    dvoid            *(*malocfp)
                                                    (dvoid* ctxp,
                                                     Size_t    size),
                          CONST    dvoid            *(*ralocfp)
                                                    (dvoid* ctxp,
                                                     dvoid* memptr,
                                                    size_t    newsize),
                          CONST    void             (*mfreefp)
                                                    (dvoid* ctxp,
                                                     dvoid *memptr),
                          size_t                    xtramem_sz,
                          dvoid                     **usrmempp);
```

参数：

envhpp: 指向环境句柄的指针。

mode: 指明初始化的模式。有效的模式有：

• OCI_DEFAULT—默认值，其为非 UTF-16 编码。

• OCI_THREADED—使用线程环境。

• OCI_OBJECT—使用对象特性。

• OCI_EVENT—利用出版-订阅特性。

• OCI_NO_UCB—禁止调用动态回调函数 OCIEnvCallback。当环境创建时，默认允许调用 OCIEnvCallback 函数。

•OCI_ENV_NO_MUTEX—在此模式中没有互斥。所有的在环境句柄中完成，或者环境句柄派生的句柄上完成的 OCI 调用都必须被序列化。

• OCI_NEW_LENGTH_SEMANTICS—

ctxp: 指明用户为内存回调函数定义的上下文。

malocfp:指明用户定义的内存分配函数。如果为 OCI_THREADED 模式，这个内存分配函数必须是支持多线程安全访问的。

ctxp:指明用户定义的内存分配函数的上下文指针。

size:指明被用户定义的内存分配函数分配的内存大小。

ralocfp:指明用户定义的内存再分配函数。如果为 OCI_THREADED 模式，这个内存分配函数必须是支持多线程安全访问的。

ctxp:用户定义的内存再分配函数的上下文指针。

memp:内存块的指针。

newsize:被分配的内存的新大小。

mfreefp:说明用户定义的内存释放函数。如果为 OCI_THREADED 模式，这个内存释放函数必须是支持多线程安全访问的。

ctxp:用户定义的内存释放函数的上下文指针。

memptr:被释放的内存的指针。

xtramemsz:说明用户分配的内存的数量。

usrmempp:为用户返回 xtramemsz 大小的内存的指针。

注释：

这个函数为在用户指定模式下的所有的OCI函数调用创建一个环境。

这个函数返回一个环境句柄，这个环境句柄被其他的OCI函数使用。在OCI中可以有多个环境，各有其环境模式。这个函数含执行各种模式所需要的进程级初始化。

7.1.2 OCIServerAttach 函数

作用：为 OCI 操作创建一个对数据源访问路径。

原型：

```
Sword    OCIServerAttach(   OCIServer       *srvhp,
                            OCIError        *errhp,
                            CONST text      *dblink,
                            Sb4             dblink_len,
                            Ub4             mode);
```

参数：

srvhp:一个未被初始化的服务器句柄，这个服务器句柄被这个函数初始化。传递一个已经被初始化的句柄会产生错误。

errhp:错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

dblink:说明使用的数据库服务器。这个参数指向一个字符串，这个字符串说明了连接字符串。如果连接字符串为 NULL，则这个函数调用会连至默认主机。

dblink_len:其为 dblink 所指向的字符串的长度。

mode:说明操作模式。有效的模式有：

- OCI_DEFAULT：为默认模式。
- OCI_CPOOL:使用连接池。

注释：

这个函数用来创建一个 OCI 应用程序和一个特定服务器之间的连接。

这个函数初始化服务器句柄，这个服务器句柄必须是已经被 OCIHandleAlloc()函数分配的。通过调用 OCIAttrSet()函数可以讲这个函数初始化的服务器上下文句柄连接到一个服务上下午句柄。一个设立了这个连接，就可以对服务器执行 OCI 操作。

如果一个应用程序对多个服务器执行操作，则可以维持多个服务器上下文句柄。可以对服务上下文句柄当前绑定的任何一个服务器上下文句柄进行 OCI 操作。

当 OCIServerAttach()函数成功执行后，会启动一个 Oracle 隐蔽进程 shadow process。为清除这个 Oracle 隐蔽程序，必须调用 OCISessionEnd()函数和 OCIServerDetach()函数。否则，随着隐蔽进程的累积会引起 Unix 系统耗尽进程。如果数据库重启并且没有足够的进程，数据库就无法启动。

例子：

下面的例子示范了 OCIServerAttach()函数的使用。这段代码分配了服务器句柄，调用了 attach 函数，分配了服务上下午句柄并且设置了服务上下文句柄中的服务器上下文句柄。

```
/*分配服务器上下文句柄*/
OCIHandleAlloc( (dvoid *) envhp,    (dvoid **) &srvhp,    (ub4)OCI_HTYPE_SERVER,
                0,        (dvoid **) 0);
/*初始化服务器上下文句柄*/
OCIServerAttach( srvhp,    errhp,    (text *) 0,    (sb4) 0,    (ub4) OCI_DEFAULT);
/*分配服务上下文句柄*/
OCIHandleAlloc( (dvoid *) envhp,    (dvoid **) &svchp,    (ub4)OCI_HTYPE_SVCCTX,
```

```
0,          (dvoid **) 0);
/*设置服务上下文句柄中的服务器上下文属性*/
OCIAttrSet( (dvoid *) svchp,     (ub4) OCI_HTYPE_SVCCTX,          (dvoid *) srvhp,
             (ub4) 0,           (ub4) OCI_ATTR_SERVER,             (OCIError *) errhp);
```

7.1.3 OCIServerDetach()函数

作用：删除 OCI 操作对一个数据源的访问。

原型：

```
Sword    OCIServerDetach(  OCIServer          *srvhp,
                           OCIError           *errhp,
                           Ub4                mode);
```

参数：

srvhp:一个已经初始化的服务器上下文句柄，该句柄会被重置为未初始化状态。该句柄不会被释放。

errhp:可以传递给 OCIErrorGet()函数来获取诊断信息的错误句柄。

mode:说明操作的不同模式。仅有的有效模式是 OCI_DEFAULT。

注释：

这个函数会删除 OCI 操作的数据源访问路径。

7.1.4 OCISessionBegin()函数

作用：创建一个用户会话并开始一个用户会话。

原型：

```
Sword    OCISessionBegin(  OCISvcCtx          *svchp,
                           OCIError           *errhp,
                           OCISession         *usrhp,
                           Ub4                credt,
                           Ub4                mode);
```

参数：

svchp:服务上下午句柄。在 svchp 中必须有一个有效的服务器上下文句柄。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

usrhp:一个用户会话上下文的句柄，函数初始化这个句柄。

credt:说明用来设置用户会话的凭据。有效值是:

• OCI_CRED_RDBMS—使用用户名和密码作为凭据来进行认证。在调用此函数前必须设置用户会话句柄中的 OCI_ATTR_USERNAME 和 OCI_ATTR_PASSWORD 属性。

• OCI_CRED_EXT—使用外部认证。不使用用户名或者密码。

mode:指定操作模式。有效模式为:

•OCI_DEFAULT—默认模式，在此模式中，返回的用户会话上下文句柄只能设置到 svchp 中已经设置的服务器上下文句柄上。

• OCI_MIGRATE—

• OCI_SYSDBA--DBA 模式，在这种模式下用户被授予 SYSDBA 权限。

• OCI_SYSOPER—操作员模式，在这种模式下用户被授予 SYSOPER 权限。

• OCI_PRELIM_AUTH—。

注释：

OCISessionBegin()函数用来向服务器认证一个用户。

OCISessionBegin()函数只支持对服务上下文句柄中的服务器句柄所指明的 Oracle 服

务器进行用户认证。

例子：

下面的例子表明了 OCISessionBegin()函数的用法。这段代码分配了用户会话句柄，设置了用户名和密码属性，调用 OCISessionBegin()函数并且将用户会话设置到服务上下文中。

```
/*分配用户会话句柄*/
OCIHandleAlloc((dvoid *)envhp,   (dvoid **)&usrhp,   (ub4)OCI_HTYPE_SESSION,
        (size_t) 0,       (dvoid **) 0);
/*设置用户会话句柄的用户名句柄*/
OCIAttrSet((dvoid *)usrhp,   (ub4)OCI_HTYPE_SESSION,       (dvoid *)"hr",
        (ub4)strlen("hr"),        OCI_ATTR_USERNAME,       errhp);
/*设置用户会话句柄的密码属性*/
OCIAttrSet((dvoid *)usrhp,   (ub4)OCI_HTYPE_SESSION,   (dvoid *)"hr",
        (ub4)strlen("hr"),        OCI_ATTR_PASSWORD,       errhp);
/*调用OCISessionBegin()函数，创建一个用户会话并开始一个用户会话*/
checkerr(errhp,    OCISessionBegin (svchp,     errhp,    usrhp,    OCI_CRED_RDBMS,
        OCI_DEFAULT));
/*设置服务上下午句柄的用户会话属性*/
OCIAttrSet((dvoid *)svchp,   (ub4)OCI_HTYPE_SVCCTX,        (dvoid *)usrhp,
        (ub4)0,        OCI_ATTR_SESSION,       errhp);
```

7.1.5 OCISessionEnd()函数

作用：终止 OCISessionBegin()函数所创建的用户会话。

原型：

```
Sword   OCISessionEnd(   OCISvcCtx      *svchp,
                         OCIError       *errhp,
                         OCISession     *usrhp,
                         Ub4            mode);
```

参数：

Svchp:服务上下文句柄。必须已经有一个有效的服务器上下文句柄和用户会话句柄连接到了 svchp。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

usrhp: 取消对该用户的认证。如果这个参数为 NULL，服务上下文句柄中的用户被取消认证。

mode: 唯一有效的模式是 OCI_DEFAULT。

7.2 句柄和描述符函数

这一节描述 OCI 句柄和描述符函数。

表 7-2 句柄和描述符函数

Function	Purpose
OCIAttrGet() on page 15-49	Get the attributes of a handle
OCIAttrSet() on page 15-52	Set an attribute of a handle or descriptor
OCIDescriptorAlloc() on page 15-54	Allocate and initialize a descriptor or LOB locator
OCIDescriptorFree() on page 15-57	Free a previously allocated descriptor
OCIHandleAlloc() on page 15-59	Allocate and initialize a handle
OCIHandleFree() on page 15-62	Free a previously allocated handle
OCIParamGet() on page 15-64	Get a parameter descriptor
OCIParamSet() on page 15-66	Set parameter descriptor in COR handle

7.2.1 OCIHandleAlloc()函数

作用：返回一个被分配的和初始化的句柄的指针。

原型：

```
sword    OCIHandleAlloc(    CONST dvoid    *parenth,
                            dvoid          **hndlpp,
                            ub4            type,
                            size_t         xtramem_sz,
                            dvoid          **usrmempp);
```

参数：

parenth:一个环境句柄。

hndlpp:返回一个句柄。

type:指定要被分配的句柄的类型。允许的类型是：

- OCI_HTYPE_ENV—分配 C 类型 OCIEnv 的环境句柄。
- OCI_HTYPE_ERROR—分配 C 的 OCIError 类型的错误报告句柄。
- OCI_HTYPE_SVCCTX—分配 C 的 OCISvcCtx 类型的服务上下午句柄。
- OCI_HTYPE_STME—分配 C 的 OCISvcCtx 类型的语句句柄。
- OCI_HTYPE_DESCRIBE—分配 C 的 OCIDescribe 类型的描述句柄。
- OCI_HTYPE_SERVER—分配 C 类型为 OCIServer 的服务器上下文句柄。
- OCI_HTYPE_SESSION—分配 C 类型为 OCISession 的用户会话句柄。
- OCI_HTYPE_TRANS—分配 C 类型为 OCITrans 的事务上下文句柄。

xtramem_sz:指定被分配的用户内存。

usrmempp:返回被分配的 xtramem_sz 大小的内存的指针。

注释：

这个函数返回一个参数 type 中指定的类型的句柄指针。执行成功后返回一个 non-NULL 类型的句柄。所有被分配的句柄都相关于作为输入参数的父句柄 parent handle—环境句柄。

当有错误发生时，不提供诊断信息。这个函数执行成功后返回 OCI_SUCCESS，当有错误发生时返回 OCI_INVALID_HANDLE。

将句柄传至 OCI 函数前，必须使用 OCIHandleAlloc()函数来分配句柄。

调用 OCIEnvInit 函数来分配和初始化环境句柄。

7.2.2 OCIHandleFree()函数

作用：这个函数显式地释放一个句柄。

原型：

```
sword    OCIHandleFree(    dvoid    *hndlp,
                           ub4      type);
```

参数：

hndlp: OCIHandleAlloc()函数分配的句柄。

type:说明要被释放的句柄的类型。相关的类型有：

- OCI_HTYPE_ENV—分配 C 类型 OCIEnv 的环境句柄。
- OCI_HTYPE_ERROR—分配 C 的 OCIError 类型的错误报告句柄。
- OCI_HTYPE_SVCCTX—分配 C 的 OCISvcCtx 类型的服务上下午句柄。
- OCI_HTYPE_STME—分配 C 的 OCISvcCtx 类型的语句句柄。
- OCI_HTYPE_DESCRIBE—分配 C 的 OCIDescribe 类型的描述句柄。
- OCI_HTYPE_SERVER—分配 C 类型为 OCIServer 的服务器上下文句柄。
- OCI_HTYPE_SESSION—分配 C 类型为 OCISession 的用户会话句柄。
- OCI_HTYPE_TRANS—分配 C 类型为 OCITrans 的事务上下文句柄。

注释：

这个函数释放 type 指定类型的句柄的存储空间。

这个函数返回 OCI_SUCCESS 或者 OCI_INVALID_HANDLE。

所有的句柄都可以被显式地释放。如果父句柄被释放，则 OCI 会释放其子句柄的存储空间。

7.2.3 OCIAttrGet()函数

作用：用来获取一个句柄的属性。

原型：

```
Sword    OCIAttrGet(    CONST   dvoid        *trgthndlp,
                        Ub4                  trghndltyp,
                        Dvoid                *attributep,
                        Ub4                  *sizep,
                        Ub4                  attrtype,
                        OCIError             *errhp);
```

参数：

trgthndlp:句柄的指针。实际的句柄可以是一个语句句柄、一个会话句柄等等。

trgthndltyp:说明句柄的类型。

attributep:属性值的指针。

sizep:属性值的大小。因为 attributep 是 dvoid 类型的指针，所以 size 总是以字节为单位。因为非字符串属性的大小是 OCI 库已知的，所以这个参数可以置为 NULL。对于 text*类型的参数，该参数必须是一个 ub4 类型的指针，其提供字符串的长度。

attrtype:获取的属性的类型。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

注释：

这个函数用来获取一个句柄的特定属性。

7.2.4 OCIAttrSet()函数

作用：这个函数用来设置一个句柄或者描述符的属性。

原型：

```
sword    OCIAttrSet(    dvoid       *trgthndlp,
                        ub4         trghndltyp,
                        dvoid       *attributep,
                        ub4         size,
```

```
                    ub4           attrtype,
                    OCIError      *errhp);
```

参数：

trgthndlp:要修改的句柄的指针。

trghndltyp:句柄的类型。

attributep:属性值的指针。属性值会被复制到目标句柄中。如果属性值为指针，则会复制指针而不会复制指针的内容。

size:属性值的大小。对 OCI 已经知道的属性，可以直接传递 0。对 text*类型的属性，则需要传递 ub4 类型的字符串的长度。

attrtype:被设置的属性的类型。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

7.2.5 OCIDescriptorAlloc()函数

作用：为保存描述符或者 LOB 定位符分配空间。

原型：

```
sword   OCIDescriptorAlloc(CONST   dvoid        *parenth,
                           dvoid                **descpp,
                           ub4                  type,
                           size_t               xtramem_sz,
                           dvoid                **usrmempp);
```

参数：

parenth:一个环境句柄。

descpp:返回一个描述符或者 LOB 定位符。

type:指定描述符的类型或者 LOB 定位符。

- OCI_DTYPE_SNAP—
- OCI_DTYPE_LOB—用于分配 OCILobLocator 类型的 LOB 定位符(对于 BLOB 或者 CLOB).
- OCI_DTYPE_FILE—用于 OCILobLocator 类型的 FILE 值类型的定位符。

xtramem_sz:指定应用程序为用户分配的内存的数量。

usrmempp:返回 xtramem_sz 大小的内存的指针。

注释：

返回 type 参数中指定的类型的描述符的指针。执行成功后返回非空的描述符或者 LOB 定位符。

如果执行成功，函数返回 OCI_SUCCESS，否则返回 OCI_INVALIDE_HANDLE。

7.2.6 OCIDescriptorFree()函数

作用；这个函数用来释放之前分配的描述符。

原型：

```
sword   OCIDescriptorFree(   dvoid        *descpp,
                             ub4          type);
```

参数：

descpp:一个被分配的描述符。

type:指出要被释放的存储空间的类型。

注释：

这个函数释放同一个描述符相连的存储空间。返回 OCI_SUCCESS 或者 OCI_INVALID_HANDLE。所有的描述符都可以被显式地释放，但是，当环境句柄被释放

后，其描述符也会被释放。

7.2.7 OCIParamGet()函数

作用：返回一个描述句柄或者语句句柄中指定位置的参数描述符。

原型：

```
sword   OCIParamGet(   CONST dvoid   *hndlp,
                       ub4           htype,
                       OCIError      *errhp,
                       dvoid         **parmdpp,
                       ub4           pos);
```

参数：

hndlp:一个语句句柄或者描述句柄。OCIParamGet()函数将会为这个句柄返回一个参数描述符。

htype:指定传递至参数 hndlp 的句柄的类型。有效的类型是：

- OCI_DTYPE_PARAM,参数描述符。
- OCI_HTYPE_COMPLEXOBJECT，复杂对象获取句柄。
- OCI_HTYPE_STMT，语句句柄。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

parmdpp:参数 pos 指定的位置的参数描述符。

pos:说明语句句柄或者描述句柄中的位置。将会为这个位置返回一个参数描述符。

注释：

这个函数返回描述句柄或者语句句柄中指定位置的参数描述符。参数描述符总是在 OCI 库内部进行分配的。可以通过 OCIDescriptorFree()函数来释放它们。

7.3 绑定、定义和描述函数

这部分描述绑定、定义和描述函数。

表 7-3 绑定、定义和描述函数

Function	Purpose
OCIBindArrayOfStruct() on page 15-69	Set skip parameters for static array bind
OCIBindByName() on page 15-71	Bind by name
OCIBindByPos() on page 15-77	Bind by position
OCIBindDynamic() on page 15-82	Sets additional attributes after bind with OCI_DATA_AT_EXEC mode
OCIBindObject() on page 15-87	Set additional attributes for bind of named datatype
OCIDefineArrayOfStruct() on page 15-90	Set additional attributes for static array define
OCIDefineByPos() on page 15-92	Define an output variable association
OCIDefineDynamic() on page 15-97	Sets additional attributes for define in OCI_DYNAMIC_FETCH mode
OCIDefineObject() on page 15-100	Set additional attributes for define of named datatype
OCIDescribeAny() on page 15-102	Describe existing schema objects
OCIStmtGetBindInfo() on page 15-105	Get bind and indicator variable names and handle

7.3.1 OCIBindByName()函数

作用：创建程序变量和一个 SQL 语句中的占位符之间的结合。

原型：

```
sword   OCIBindByName(   OCIStmt   *stmtp,
```

```
OCIBind          **bindpp,
OCIError         *errhp,
CONST text       *placeholder,
sb4              placeh_len,
dvoid            *valuep,
sb4              value_sz,
ub2              dty,
dvoid            *indp,
ub2              *alenp,
ub2              *rcodep,
ub4              maxarr_len,
ub4              *curelep,
ub4              mode);
```

参数：

stmtp:被处理的 SQL 或者 PL/SQL 语句的语句句柄。

bindpp:保存这个函数隐式分配的绑定句柄的指针。绑定句柄维护了某个特定的输入值的所有的绑定信息。当语句句柄被释放时，这个绑定句柄会被隐式地释放。作为输入/输出参数，在输入时，指针的是必须为 NULL 或者是一个有效的绑定句柄。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

placeholder:由名字来指定的占位符，其对应着同语句句柄结合的语句中的一个变量。

placeh_len:占位符的长度。以字节为单位。

valuep: 指向 dty 参数制定的类型的数值的指针。

value_sz:万能指针 valuep 指向的数值的长度。对于客户应用程序不知道其大小的描述符、定位符或者 REFs，我们使用 sizeof(OCILobLocator*)。

dty：指定被绑定的数值的数据类型。

Indp:指示器变量的指针。对于除 SQLT_NTY 之外的数据类型，该指针为指向 sb2 类型的指针或者一个 sb2 的数组。

alenp:数组袁术的实际长度的指针。

rcodep:指向一个字段级返回代码数组的指针。

maxarr_len:

curelep:

mode:模式。有效的模式有

- OCI_DEFAULT—默认模式。
- OCI_BIND_SOFT。
- OCI_DATA_AT_EXEC。

注释：

这个函数用来执行基本的绑定操作。绑定会在一个程序变量的地址和一个 SQL 语句或者 PL/SQL 块中的占位符之间建立结合。这个函数也指定了被绑定的数据的类型，也说明了在运行时提供数据的方式。

这个函数会隐式地分配绑定句柄，绑定句柄由 bindpp 参数指明。如果**bindpp 是一个非空指针，OCI 假定这个指针指向一个之前分配的有效的句柄。

7.3.2 OCIBindByPos()函数

作用：创建程序变量和一个 SQL 语句中的占位符之间的结合。

原型：

```
sword    OCIBindByPos(OCIStmt*          *stmtp,
                      OCIBind           **bindpp,
                      OCIError          *errhp,
                      ub4               position,
                      dvoid             *valuep,
                      Sb4               value_sz,
                      ub2               dty,
                      dvoid             *indp,
                      ub2               *alenp,
                      ub2               *rcodep,
                      ub4               maxarr_len,
                      ub4               *curelep,
                      ub4               mode);
```

参数：

stmtp:被处理的 SQL 或者 PL/SQL 语句的语句句柄。

bindpp:保存这个函数隐式分配的绑定句柄的指针。绑定句柄维护了某个特定的输入值的所有的绑定信息。当语句句柄被释放时，这个绑定句柄会被隐式地释放。作为输入/输出参数，在输入时，指针的是必须为 NULL 或者是一个有效的绑定句柄。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

position:说明占位符的位置。

valuep: 指向 dty 参数制定的类型的数值的指针。

value_sz:万能指针 valuep 指向的数值的长度。对于客户应用程序不知道其大小的描述符、定位符或者 REFs，我们使用 sizeof(OCILobLocator*)。

dty：指定被绑定的数值的数据类型。

Indp:指示器变量的指针。对于除 SQLT_NTY 之外的数据类型，该指针为指向 sb2 类型的指针或者一个 sb2 的数组。

alenp:数组衷术的实际长度的指针。

rcodep:指向一个字段级返回代码数组的指针。

maxarr_len:

curelep:

mode:模式。有效的模式有

注释：

这个函数用来执行基本的绑定操作。绑定会在一个程序变量的地址和一个 SQL 语句或者 PL/SQL 块中的占位符之间建立结合。这个函数也指定了被绑定的数据的类型，也说明了在运行时提供数据的方式。

这个函数会隐式地分配绑定句柄，绑定句柄由 bindpp 参数指明。如果**bindpp 是一个非空指针，OCI 假定这个指针指向一个之前分配的有效的句柄。

7.3.3 OCIDefineByPos()函数

作用：建立所选字段中的一个字段和输出缓冲区的结合。

原型：

```
Sword    OCIDefineByPos(    OCIStmt       *stmtp,
                            OCIDefine     **defnpp,
```

```
OCIError    *errhp,
Ub4         position,
Dvoid       *valuep,
Sb4         value_sz,
Ub2         dty,
Dvoid       *indp,
Ub2         *rlenp,
Ub2         *rcodep,
Ub4         mode);
```

参数：

stmtp:请求 SQL 查询的语句句柄。

defnpp:指向定义句柄指针的指针。如果这个参数为 NULL，这个函数将会隐式地分配定义句柄。在重新定义的情况下，一个非 NULL 句柄可以传递到这个参数。这个句柄用来保存某个字段的定义信息。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

position:这个值在所选字段中的位置。位置是从 1 开始的，并且从左向右递增。

valuep:指向 dty 参数所指明的类型的缓冲区或者缓冲区数组的指针。对于 LOB 字段，缓冲区指针必须是 OCILobLocator 类型的 LOB 定位符的指针。

value_sz: valuep 指针所指向的缓冲区的大小。

dty:数据类型。用于指明所选字段如何转换为程序变量。

indp:指示器变量或者数组的指针。对于标准变量，该指针为指向 sb2 类型的的指针或者是一个 sb2 类型的数组。

rlenp:指向获取到的数据的长度的数组的指针。Rlenp 中的每个元素都是数据的长度。

rcodep:字段级返回代码数组的指针。

mode:模式。有效的模式为：

- OCI_DEFAULT—默认模式。
- OCI_DEFINE_SOFT.
- OCI_DYNAMIC_FETCH.

注释：

这个函数定义一个将要从 Oracle 服务器中获取数据的输出缓冲区。这个函数会为所选字段隐式地分配定义句柄。如果*defnpp 为一个非空指针，OCI 假设这个指针指向一个之前通过 OCIHandleAlloc()函数或者 OCIDefineByPos()函数分配的有效的句柄。

获取 LOB 字段类型时，缓冲区指针必须是一个指向 OCILobLocator 类型的 LOB 定位符的指针，通过 OCIDescriptorAlloc()函数来分配它。LOB 字段返回 LOB 定位符，而不是 LOB 数据。

7.4 语句函数

这部分描述语句函数。表 7-4 列出了语句函数。

表 7-4 语句函数

Function	Purpose
OCIStmtExecute() on page 16-5	Send statements to server for execution
OCIStmtFetch() on page 16-9	Fetch rows from a query (deprecated)
OCIStmtFetch2() on page 16-11	Fetch rows from a query
OCIStmtGetPieceInfo() on page 16-14	Get piece information for piecewise operations
OCIStmtPrepare() on page 16-16	Prepares a SQL or PL/SQL statement for execution.
OCIStmtPrepare2() on page 16-18	Prepares a SQL or PL/SQL statement for execution.
OCIStmtRelease() on page 16-20	Releases the statement handle.
OCIStmtSetPieceInfo() on page 16-21	Set piece information for piecewise operations

7.4.1 OCIStmtPrepare()函数

作用：为执行 SQL 或者 PL/SQL 语句而做准备。

原型：

```
sword    OCIStmtPrepare(  OCIStmt          *stmtp,
                          OCIError         *errhp,
                          CONST    text    *stmt,
                          ub4              stmt_len,
                          ub4              language,
                          ub4              mode);
```

参数：

stmtp:语句句柄。与这个语句句柄相连的语句将会被执行。

errhp:错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

stmt:语句。被执行的 SQL 或者 PL/SQL 语句。必须是一个 NULL 结尾的字符串。

stmt_len:语句的长度。

language:解析语句的语法。可能的值有：

- OCI_V7_SYNTAX—V7 Oracle 解析语法。
- OCI_NTV_SYNTAX—解析的语法依赖于服务器的版本。

mode:模式。唯一可能的值是：

- OCI_DEFAULT—默认模式。

注释：

一个 OCI 应用程序使用这个函数来为执行一个 SQL 或者 PL/SQL 语句做准备。OCIStmtPrepare()函数也定义了一个应用程序请求。

这个函数并不创建语句句柄和数据库服务器之间的连接。

7.4.2 OCIStmtExecute()函数

作用：连接一个应用程序请求值服务器。

原型：

```
sword    OCIStmtExecute(  OCISvcCtx               *svchp,
                          OCIStmt                 *stmtp,
                          OCIError                *errhp,
                          ub4                     iters,
                          ub4                     rowoff,
                          CONST    OCISnapshot    *snap_in,
                          OCISnapshot             *snap_out,
```

ub4 mode);

参数：

svchp:服务上下文句柄。

stmtp:语句句柄。它定义了语句并且连接数据到服务器。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

iters:对于非 SELECT 语句，语句执行的次数等于 iters-rowoff。

对于 SELECT 语句，如果 iters 为非 0，则必须已经完成了语句句柄的定义操作。语句的执行将获取 iters 行数据进入预定义的缓冲区中并且根据预获取行数 prefetch row count 预获取更多行。如果我们不知道 SELECT 语句会返回多少行，则把 iters 设置为 0。

对于非 SELECT 语句，如果 iters=0，则函数返回一个错误。

rowoff:起始索引。关于多行执行的数组绑定中的起始索引。

snap_in:该参数为可选参数。如果使用该参数，其必须指向一个 OCI_DTYPE_SNAP 类型的快照描述符。

snap_out:该参数为可选参数。如果使用该参数，其必须指向 OCI_DTYPE_SNAP 类型的描述符。

mode:模式包括

• OCI_BATCH_ERRORS

• OCI_COMMIT_ON_SUCCESS—当一个语句在此模式下执行时，如果语句成功执行，则执行完成后当前的事务会被提交。

• OCI_DEFAULT—默认模式，使用此模式来调用 OCIStmtExecute()函数。它会隐式地返回关于所选字段的描述信息。

• OCI_DESCRIBE_ONLY—描述模式。这个模式用于在执行前获取查询的描述信息。以此模式调用 OCIStmtExecute()函数并不执行语句，而是返回所选字段的描述信息。为提高性能，建议用户使用默认模式。

• OCI_EXACT_FETCH—

• OCI_PARSE_ONLY—

• OCI_STMT_SCROLLABLE_READONLY--

注释：

这个函数用来执行一个已经准备好的 SQL 语句。调用这个函数时，应用程序与服务器建立请求。

7.4.3 OCIStmtFetch()函数

作用：获取查询的结果。

原型：

```
Sword    OCIStmtFetch( OCIStmt          *stmtp,
                       OCIError         *errhp,
                       Ub4              nrows,
                       Ub2              orientation,
                       Ub4              mode);
```

参数：

Stmtp:语句句柄。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

nrows:从当前位置获取的行数。

orientation:在 9.0 版本以前，唯一可以接受的值是 OCI_FETCH_NEXT。

mode:模式。传入 OCI_DEFAULT。

注释：

如果预获取的行数足够的话，这个获取函数是一个本地调用。

如果读取的是 LOB 字段，获取到的是 LOB 定位符。

当获取不到数据时这个函数返回 OCI_NO_DATA。

7.5 LOB 函数

这部分描述使用 LOB 定位的 LOB 函数。

7.5.1 OCILobGetLength()函数

作用：获取 LOB 的长度。

原型：

```
Sword    OCILobGetLength(  OCISvcCtx          *svchp,
                           OCIError           *errhp,
                           OCILobLocator      *locp,
                           Ub4                *lenp);
```

参数：

svchp:服务上下文句柄。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

locp:引用 LOB 的 LOB 定位符。

lenp:如果 LOB 不为空，则其为 LOB 的长度。对于字符型 LOB，它是字符的数量，对于二进制 LOB 和 BFILE，它是 LOB 的字节数量。

注释：

获取 LOB 的长度。空的 LOB 的长度为 0。

7.5.2 OCILobRead()函数

作用：读取 LOB 或者 BFILE 的一部分到缓冲区。

原型：

```
sword    OCILobRead(  OCISvcCtx             *svchp,
                      OCIError              *errhp,
                      OCILobLocator         *locp,
                      ub4                   *amtp,
                      ub4                   offset,
                      dvoid                 *bufp,
                      ub4                   bufl,
                      dvoid                 *ctxp,
                      OCICallbackLobRead    (cbfp)
                                            (dvoid              *ctxp,
                                            CONST    dvoid      *bufp,
                                            Ub4                 len,
                                            Ub1                 piece)
                      ub2                   csid,
                      ub1                   csfrm);
```

参数：

svchp:服务上下午句柄。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

locp:引用 LOB 或者 BFILE 的一个 LOB 或者 BFILE 定位符。

amtp:字节的或者字符的数量。如下表所示

LOB or BFILE	Input	Output with fixed-width client-side character set	Output with varying-width client-side character set
BLOBs and BFILEs	bytes	bytes	bytes
CLOBs and NCLOBs	characters	characters	bytes (1)

offset:表示从 LOB 值开始处的偏移量。

bufp:保存 LOB 值的缓冲区的指针。假定缓冲区的长度为 bufl。

bufl:缓冲区的长度。

ctxp:回调函数的上下文指针。可以为 NULL。

cbfp:

ctxp:回调函数的上下文。可以为 NULL。

Bufp:

Len:

Piece:

Csid:

Csfrm:

注释：

这个从 LOB 或者 BFILE 中读取一部分到一个缓冲区中。从空的 LOB 或者 BFILE 中读取值是一个错误。

7.5.3 OCILobWrite()函数

作用：将缓冲区中的数据写入到 LOB 中。

原型：

```
Sword    OCILobWrite(  OCISvcCtx              *svchp,
                       OCIError               *errhp,
                       OCILobLocator          *locp,
                       ub4                    *amtp,
                       ub4                    offset,
                       dvoid                  *bufp,
                       ub4                    buflen,
                       ub1                    piece,
                       dvoid                  *ctxp,
                       OCICallbackLobWrite    (cbfp)
                                              (dvoid    *ctxp,
                                              Dvoid     *bufp,
                                              Dvoid     *bufp,
                                              Ub4       *lenp,
                                              Ub1       *piecep)
                       ub2                    csid,
```

```
                              ub1                    csfrm);
```

参数：

svchp:服务上下午句柄。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

locp:引用 LOB 或者 BFILE 的一个 LOB 或者 BFILE 定位符。

amtp:字节的或者字符的数量。如下表所示

LOB or BFILE	Input	Output with fixed-width client-side character set	Output with varying-width client-side character set
BLOBs and BFILEs	bytes	bytes	bytes
CLOBs and NCLOBs	characters	characters	bytes (1)

offset:表示从 LOB 值开始处的偏移量。

bufp:保存 LOB 值的缓冲区的指针。假定缓冲区的长度为 bufl。

bufl:缓冲区的长度。

ctxp:回调函数的上下文指针。可以为 NULL。

cbfp:

ctxp:回调函数的上下文。可以为 NULL。

Bufp:

Len:

Piece:

Csid:

Csfrm:

注释：

将缓冲区中的数据写入到 LOB 字段中。

7.6 事务函数

这部分描述事务相关函数。表 7-5 列出了所有的事务相关函数。

表 7-5 事务相关函数

Function	Purpose
OCITransCommit() on page 16-198	Commit a transaction on a service context
OCITransDetach() on page 16-201	Detach a transaction from a service context
OCITransForget() on page 16-203	Forget a prepared global transaction
OCITransMultiPrepare() on page 16-204	Prepare a transaction with multiple branches in a single cell
OCITransPrepare() on page 16-205	Prepare a global transaction for commit
OCITransRollback() on page 16-206	Roll back a transaction
OCITransStart() on page 16-207	Start a transaction on a service context

7.6.1 OCITransCommit()函数

作用：提交与指定的服务上下午联系的事务。

原型：

```
Sword    `OCITransCommit(  OCISvcCtx          *svchp,
                           OCIError           *errhp,
```

```
                                    Ub4              flags);
```

参数：

Svchp:服务上下午句柄。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

flags:如果事务不是分布式的，则 flags 可以被忽略，将 OCI_DEFAULT 作为其值。

注释：

与当前的服务上下午句柄相连的事务被提交。

7.6.2 OCITransRollback()函数

作用：回滚当前事务。

原型：

```
Sword   OCITransRollback(   dvoid            *svchp,
                            OCIError         *errhp,
                            Ub4              flags);
```

参数：

svchp:服务上下文句柄。服务上下午句柄中的事务被回滚。

errhp: 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

flags:必须为此参数传递 OCI_DEFAULT。

注释：

该函数回滚当前事务。

7.7 线程管理函数

这部分描述线程管理函数，表 16-9 列出了所有的线程管理函数

表 16-9 线程管理函数 Thread Management functions

Function	Purpose
OCIThreadClose() on page 16-170	Closes a thread handle
OCIThreadCreate() on page 16-171	Creates a new thread
OCIThreadHandleGet() on page 16-173	Retrieves the OCIThreadHandle of the thread in which it is called
OCIThreadHndDestroy() on page 16-174	Destroys and deallocates the thread handle
OCIThreadHndInit() on page 16-175	Allocates and initializes the thread handle
OCIThreadIdDestroy() on page 16-176	Destroys and deallocates a thread id
OCIThreadIdGet() on page 16-177	Retrieves the OCIThreadId of the thread in which it is called
OCIThreadIdInit() on page 16-178	Allocate and initialize the thread id
OCIThreadIdNull() on page 16-179	Determines whether or not a given OCIThreadId is the NULL thread ID
OCIThreadIdSame() on page 16-180	Determines whether or not two OCIThreadIds represent the same thread
OCIThreadIdSet() on page 16-181	Sets one OCIThreadId to another
OCIThreadIdSetNull() on page 16-182	Sets the NULL thread ID to a given OCIThreadId
OCIThreadInit() on page 16-183	Initializes OCIThread context
OCIThreadIsMulti() on page 16-184	Tells the caller whether the application is running in a multithreaded environment or a single-threaded environment
OCIThreadJoin() on page 16-185	Allows the calling thread to join with another thread
OCIThreadKeyDestroy() on page 16-186	Destroy and deallocate the key pointed to by key
OCIThreadKeyGet() on page 16-187	Gets the calling threads current value for a key
OCIThreadKeyInit() on page 16-188	Creates a key
OCIThreadKeySet() on page 16-190	Sets the calling threads value for a key
OCIThreadMutexAcquire() on page 16-191	Acquires a mutex for the thread in which it is called

Function	Purpose
OCIThreadMutexDestroy() on page 16-192	Destroys and deallocate a mutex
OCIThreadMutexInit() on page 16-193	Allocates and initializes a mutex
OCIThreadMutexRelease() on page 16-194	Releases a mutex
OCIThreadProcessInit() on page 16-195	Performs OCIThread process initialization
OCIThreadTerm() on page 16-196	Releases the OCIThread context

7.7.1 OCIThreadProcessInit()函数

作用：执行 OCIThread 进程初始化。

原型：

```
Void        OCIThreadProcessInit();
```

注释：

是否需要调用这个函数依赖于 OCI 线程如何被使用。

在一个单线程应用程序中，可以调用这个函数也可以不调用这个函数。如果它被调用了，则对它的第一次调用必须早于任何其他的 OCIThread 函数。之后的对 OCIThreadProcessInit()函数的调用将没有任何限制：它们没有任何作用。

在一个多线程应用程序中，这个函数必须被调用。对它的第一次调用必须早于任何其他的 OCIThread 函数调用。之后的对 OCIThreadProcessInit()函数的调用将没有任何限制：它们没有任何作用。

7.7.2 OCIThreadInit()函数

作用：初始化 OCIThread 上下文

原型：

```
Sword    OCIThreadInit( dvoid        *hndl,
                        OCIError     *err);
```

参数：

hndl: OCI 环境句柄或者用户会化句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

注释：

OCIThreadInit()函数第一次被调用时，它初始化 OCI 线程上下文。它以依赖于系统的方式来保存一个指向该上下文的指针。之后对 OCIThreadInit()函数的调用会返回同样的上下文。

每一个对 OCIThreadInit()函数的调用都匹配一个对 OCIThreadTerm()函数的调用。

7.7.3 OCIThreadTerm()函数

作用：释放 OCIThread 上下文。

原型：

```
Sword    OCIThreadTerm(  dvoid        *hndl,
                         OCIError     *err);
```

参数：

hndl: OCI 环境句柄或者用户会化句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

注释：

这个函数必须与每一个对 OCIThreadInit()函数的调用相匹配。

对 OCIThreadTerm()函数的并发调用是安全的。OCIThreadTerm()函数不做任何事情直到它被调用的次数和 OCIThreadInit()函数被调用的次数相同。当调用次数相同时，它终止 OCIThread 层并且释放分配给上下文的内存。一旦这发生了，该上下文就不能被重用。需要调用 OCIThreadInit()函数来获取一个新的 OCIThread 上下文。

7.7.4 OCIThreadIdInit()函数

作用：分配和初始化线程 ID

原型：

```
Sword    OCIThreadIdInit(  dvoid          *hndl,
                           OCIError       *err,
                           OCIThreadId    **tid);
```

参数：

hndl:OCI 环境句柄或者用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

tid:要初始化的线程 ID Thread ID 的指针。

7.7.5 OCIThreadIdGet()函数

作用：获取调用该函数的线程的 OCIThreadId。

原型：

```
Sword    OCIThreadIdGet(  dvoid        *hndl,
                          OCIError     *err,
```

```
                    OCIThreadId    *tid);
```

参数：

hndl:OCI 环境句柄或者用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

tid:其指向保存调用线程的 ID 的地方。

注释：

tid 应该被 OCIThradIdInit()函数来初始化。当 OCIThread 用于单线程环境时，OCIThreadIdGet()函数将始终放置相同的值在 tie 所指向的地方。重要的并不是这个数值。重要的是这个值不为 NULL，thread ID 总是相同的值。

7.7.6 OCIThreadIdNull()函数

作用：查看一个给定的 OCIThreadId 是否是 NULL thread ID

原型：

```
Sword    OCIThreadIdNull( dvoid          *hndl,
                         OCIError       *err,
                         OCIThreadId    *tid,
                         boolean        *result);
```

参数：

hndl:OCI 环境句柄或者用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

tid:所要检测的 OCIThreadIde 的指针。

result:指向结果的指针。

注释：

如果线程 ID 为 NULL 线程 ID，result 被设置为 TRUE。否则，result 被设置为 FALSE。tid 应该被 OCIThreadIdInit()初始化。

7.7.7 OCIThreadIdSame()函数

作用：检测两个 OCIThreadId 是否代表同一个线程

原型：

```
Sword    OCIThreadIdSame( dvoid          *hndl,
                         OCIError       *err,
                         OCIThreadId    *tid1,
                         OCIThreadId    *tid2,
                         Boolean        *result);
```

参数：

hndl:OCI 环境句柄或者用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

注释：

7.7.8 OCIThreadMutexInit()函数

作用：分配和初始化一个互斥 mutex。

原型：

```
Sword    OCIThreadMutexInit(    dvoid              *hndl,
```

```
OCIError        *err,
OCIThreadMutex  **mutex);
```

参数：

hndl:OCI 环境句柄或者用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

mutex:要初始化的互斥。

注释：

所有的互斥在使用前必须被初始化。

多个线程不能同时初始化同一个互斥。并且，在一个互斥被销毁前，其不能被重新初始化。

7.7.9 OCIThreadMutexDestroy()函数

作用：销毁互斥并释放其所占用的空间。

原型：

```
Sword   OCIThreadMutexDestroy(   dvoid           *hndl,
                                 OCIError        *err,
                                 OCIThreadMutex  **mutex);
```

参数：

hndl:OCI 环境句柄或者用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

mutex:要销毁的互斥。

注释：

每一个不再被使用的互斥一定要被销毁。

销毁一个未被初始化的互斥或者一个当前被一个线程持有的互斥并不是错误的。互斥的销毁不能与其他对互斥的操作并发执行。一个互斥在其被销毁后，其不能再被使用。

7.7.10 OCIThreadMutexAcquire()函数

作用：为调用这个函数的线程获取互斥

原型：

```
Sword   OCIThreadMutexAcquire(dvoid          *hndl,
                              OCIError       *err,
                              OCIThreadMutex *mutex);
```

参数：

hndl:OCI 环境句柄或者用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

mutex:所要获取的互斥。

注释：

如果互斥被其他线程持有，则该线程会被阻塞直到它能够获取互斥。

获取一个未初始化的线程是违规的。

7.7.11 OCIThreadMutexRelease()函数

作用：释放一个互斥 mutex

原型：

```
Sword   OCIThreadMutexRelease( dvoid          *hndl,
```

```
OCIError        *err,
OCIThreadMutex  *mutex);
```

参数：

hndl:OCI 环境句柄或者用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

mutex:要释放的互斥。

注释：

如果有线程阻塞在这个互斥中，它们中的一个线程会获得这个互斥并且成为不阻塞的。释放一个未初始化的互斥的违规的。一个线程释放其不拥有的互斥也使违规的。

7.7.12 OCIThreadIdSetNull()函数

作用：对给定的 OCIThreadId 设置其为 NULL 线程 ID。

原型：

```
Sword    OCIThreadIdSetNull(    dvoid         *hndl,
                                OCIError      *err,
                                OCIThreadId   *tid);
```

参数：

hndl:OCI 环境句柄或者用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

OCIThreadId:指向 OCIThreadId，其会被赋值为 NULL 线程 ID。

注释：

Tid 应该已经被 OCIThreadIdInit()函数初始化。

7.7.13 OCIThreadIdSet()函数

作用：将源 OCIThreadId 的值赋给另一个 OCIThreadId

原型：

```
Sword    OCIThreadIdSet(    dvoid         *hndl,
                            OCIError      *err,
                            OCIThreadId   *tidDest,
                            OCIThreadId   *tidSrc);
```

参数：

hndl:OCI 环境句柄或者用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

tidDest:目标 ThreadId，指向所要设置的 OCIThreadId 的位置。

tidSrc:源 ThreadId,指向来源 OCIThreadId。

注释：

tid 应该被 OCIThreadIdInit()函数初始化。

7.7.14 OCIThreadCreate()函数

作用：创建新的线程。

原型：

```
Sword    OCIThreadCreate(   dvoid            *hndl,
                            OCIError         *err,
                            Void(*start)     (dvoid*),
```

```
Dvoid              *arg,
OCIThreadId        *tid,
OCIThreadHandle    *tHnd);
```

参数：

hndl:OCI 环境句柄或者 OCI 用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

start:新的线程开始执行的函数。

arg:传递给 start 所指向的函数的参数。

tid:如果不为 NULL，则为新线程获取该 ID。

tHnd:如果不为 NULL，则为新线程获取该句柄。

注释：

新线程通过执行对 start 所指向的函数并获取 arg 参数开始运行。当那个函数返回时，新的线程将终止。线程函数不应返回一个值而应该接收一个参数。只有在 tHnd 为非空的情况下，每一个 OCIThreadCreate()函数的调用必须匹配一个 OCIThreadClose()函数的调用。

tid 应该被 OCIThreadIdInit()函数初始化并且 tHnd 应该被 OCIThreadHndInit()函数初始化。

7.7.15 OCIThreadClose()函数

作用：关闭一个线程句柄。

原型：

```
Sword    OCIThreadClose(    dvoid              *hndl,
                            OCIError           *err,
                            OCIThreadHandle    *tHnd);
```

参数：

hndl:OCI 环境句柄或者 OCI 用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

tHnd:所要关闭的 OCIThread 句柄。

注释：

tHnd 应该被 OCIThreadHndInit()函数初始化。在调用 OCIThreadClose()函数后，由同一个 OCIThreadCreate()调用返回的线程句柄和线程 ID 都变为无效。

7.7.16 OCIThreadHndInit()函数

作用：分配和初始化线程句柄

原型：

```
Sword    OCIThreadHndInit(  dvoid              *hndl,
                            OCIError           *err,
                            OCIThreadHandle    **thnd);
```

参数：

hndl:OCI 环境句柄或者 OCI 用户会话句柄。

err: OCI 错误句柄，当有错误发生时，我们可以将该错误句柄传递至 OCIErrorGet()函数获取诊断信息。

tHnd:所要关闭的 OCIThread 句柄的指针的地址。

7.8

8. 句柄和描述符属性

这部分描述OCI句柄和描述符的属性，可以通过OCIAttrGet()函数来读取属性，OCIAttrSet()函数来设置属性。

各种句柄的属性可以被读取或者修改。

8.1 服务上下文句柄

8.1.1 OCI_ATTR_SERVER 属性

模式：读写模式

作用：当读取时，返回指向服务上下文句柄的服务器上下文的指针。修改时，会设置服务上下午的服务器上下文属性。

属性值的数据类型：**OCIServer**/OCIServer***

8.1.2 OCI_ATTR_SESSION 属性

模式：读写模式

作用：当读取时，返回服务上下文句柄的认证上下文属性的指针。当修改时，设置服务上下文句柄的认证上下文属性。

属性值的数据类型：**OCISession**/OCISession***

8.2 用户会话句柄

8.2.1 OCI_ATTR_USERNAME 属性

模式：写模式

作用：指定用于用户认证的用户名。

属性值的数据类型：**OraText***

8.2.2 OCI_ATTR_PASSWORD 属性

模式：写模式

作用：指定用于用户认证的密码。

属性值的数据类型：**OraText***

9. OCI 示例程序

Oracle 提供了表明 OCI 使用方式的示例程序。表 9-1 列出了这些示例程序

表 9-1 OCI 示例程序

Program Name	Features Illustrated
cdemo81.c	Using basic SQL processing with release 8 and later functionality.
cdemo82.c	Performing basic processing of user-defined objects.

Program Name	Features Illustrated
cdemocor.c	Using complex object retrieval (COR) to improve performance.
cdemodr1.c, cdemodr2.c, cdemodr3.c	Using INSERT/UPDATE/DELETE statements with RETURNING clause used with basic datatypes, LOBs and REFs.
cdemodsa.c	Describing information about a table.
cdemodsc.c	Describing information about an object type.
cdemofo.c	Registering and operating application failover callbacks.
cdemolb.c	Create and insert LOB data and then read, write, copy, append and trim the data.
cdemolb2.c	Writing and reading of CLOB/BLOB columns with stream mode and callback functions.
cdemolbs.c	Writing and reading to LOBs with the LOB buffering system.
cdemobj.c	Pinning and navigation of REF object.
cdemorid.c	Using INSERT, UPDATE, DELETE statements and fetches to get multiple rowids in one round-trip.
cdemoses.c	Using session switching and migration.
cdemothr.c	Using the OCIThread package.
cdemosyev.c	Registering predefined subscriptions and specifying a callback function to be invoked for client notifications (for more information about Advanced Queuing, see *Oracle Streams Advanced Queuing User's Guide and Reference*).
ociaqdemo00.c, ociaqdemo01.c, ociaqdemo02.c	Advanced queuing.
cdemodp.c, cdemodp_lip.c	Loading data with the direct path load functions.
cdemdpco.c	Loading a column object with the direct path load functions.
cdemdpno.c	Loading a nested column object with the direct path load functions.
cdemdpin.c	Loading derived type (inheritance) - direct path.
cdemdpit.c	Loading an object table with inheritance - direct path.
cdemdpro.c	Loading a reference with the direct path load functions.
cdemdpss.c	Loading SQL strings with the direct path load functions.

Program Name	Features Illustrated
cdemoucb.c, cdemoucbl.c	Using static and dynamic user callbacks.
cdemoupk.c, cdemoup1.c, cdemoup2.c	Using dynamic user callbacks with multiple packages.
cdemodt.c	Datetime and interval example.
cdemosc.c	Scrollable cursor.
cdemol2l.c	Accesses LOBs using the LONG API.
cdemoin1.c	Inheritance demo which modifies an inherited type in a table and displays a record from the table.
cdemoin2.c	Inheritance demo to do attribute substitutability.
cdemoin3.c	Inheritance demo that describes an object, inherited types, object tables, and a sub-table.
cdemoanydata1.c	Anydata demo. Inserts and selects rows to and from anydata table.
cdemoanydata2.c	Anydata demo. Creates a type piecewise using OCITypeBeginCreate() and then describes the new type created.
cdemosp.c	Session pooling.
cdemocp.c	Connection pooling.
cdemocpproxy.c	Connection pooling with proxy functionality.
cdemostc.c	Statement caching.
cdemouni.c	Program for OCI UTF16 API.
nchdemo1.c	Shows nchar implicit conversion feature and codepoint feature.

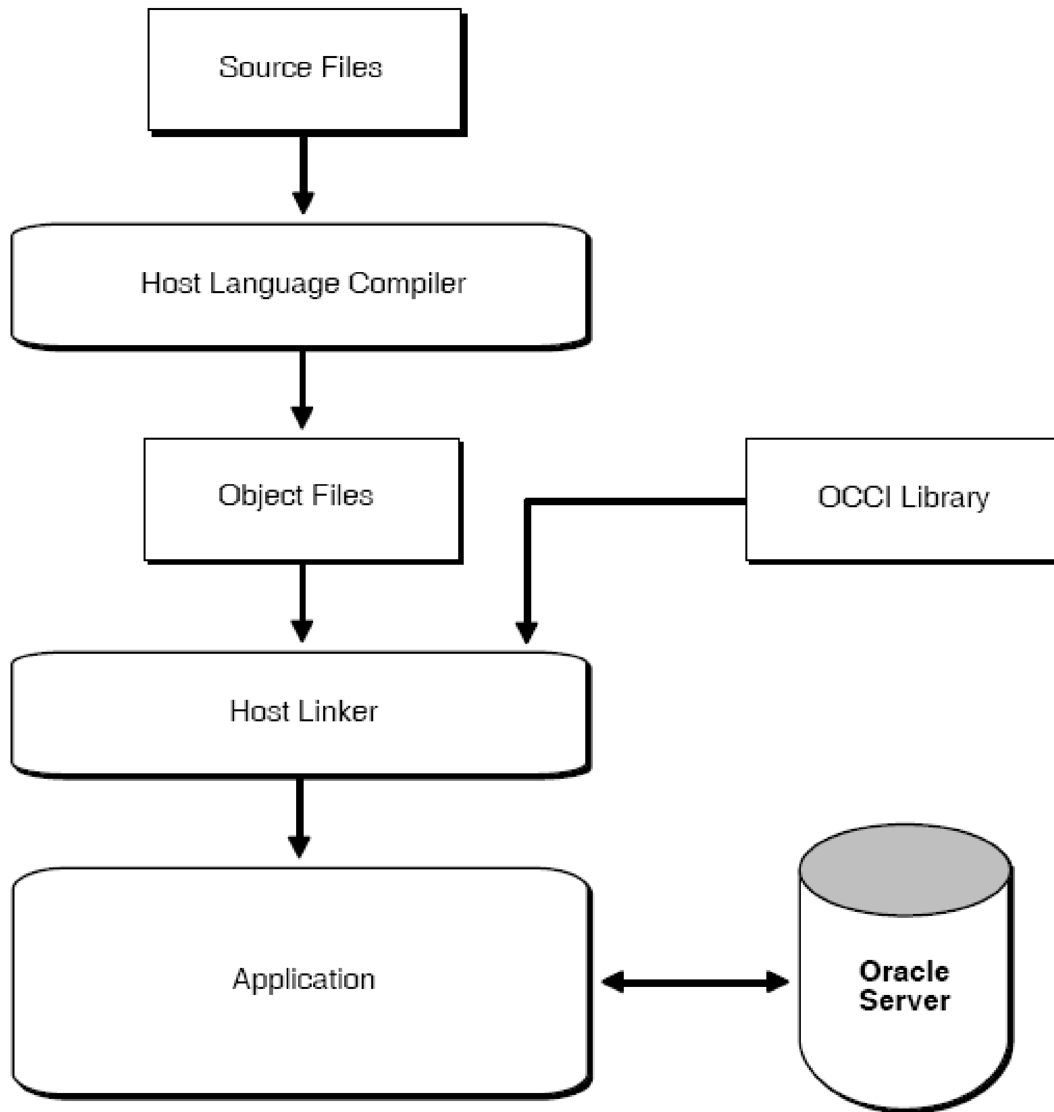
10. Oracle C++ Call Interface

Oracle C++ Call Interface 是用于 C++程序操作 Oracle 数据库的 API。OCCI 使 C++程序员能够利用 Oracle 数据库操作手段，包括 SQL 语句处理和对象操作。

10.1 创建一个 OCII 应用程序

编译和连接 OCII 程序的方式与编译连接非数据库应用程序的方式一样。如图 10-1 所示：

图 10-1



10.2 编程方法

这部分描述使用 OCCI 开发 C++应用程序来操作关系型数据库的基本知识。

10.2.1 连接至数据库

10.2.1.1 创建和终止一个环境

所有的 OCCI 操作都发生在环境类 Environment class 的上下文中。OCCI 环境提供了应用程序模式和用户指定的内存管理函数。下列代码显示了如何创建 OCCI 环境。

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

所有由 createXXX 函数创建的 OCCI 对象都必须显式地终止，在合适的时候，我们也可以显式地终止环境。下列代码显示了如何终止 OCCI 环境。

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

另外 OCCI 环境的作用于要大于任何一个在 OCCI 环境中创建的对象的作用域。下面的代码显示了这个概念：

```
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);
```

我们可以使用 CreateEnvironment 函数的模式 mode 参数来指定我们的应用程序的模式：

- 运行在多线程环境中(THREAD_MUTEXED 或者 THREADED_UNMUTEXED)
- 使用对象(OBJECT)
- 使用共享数据结构(SHARED)

10.2.1.2 打开和关闭一个连接

环境类 Environment class 是创建连接类 Connection class 对象的工厂类。我们首先创建一个环境实例，然后通过 createConnection 方法让用户连接到数据库服务器。

下面的代码创建了一个环境实例并且用它为用户 scott 密码 tiger 创建了一个数据库连接。

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

在会话结束时，我们需要使用 terminateConnection 方法关闭连接。另外 OCCI 环境应该被显式地终止。

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

10.3 执行 SQL、DDL 和 DML 语句

在 OCCI 中，我们通过语句类 Statement Class 来执行 SQL 命令。

10.3.1 创建一个环境句柄

使用连接对象 connection 的 createStatement 方法创建语句句柄 Statement Handle，如下所示：

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

10.3.2 创建语句句柄来执行 SQL 命令

创建语句句柄之后，通过调用 execute、executeUpdate、executeArrayUpdate 或者 executeQuery 方法来执行 SQL 命令。这些方法用于以下方面：

- execute:执行非特定类型的语句
- executeUpdate:执行 DML 和 DDL 语句
- executeQuery:执行一个查询
- executeArrayUpdate：执行多个 DML 语句

创建数据库表

使用 executeUpdate 方法，下面的代码显示了如何创建数据库表：

```
stmt->executeUpdate(“CREATE  TABLE  basket_tab
                (fruit   VARCHAR2(30),  quantity   NUMBER)”);
```

向数据库表中插入数据

同样，我们可以通过 executeUpdate 方法来执行一个 SQL INSERT 语句。

```
Stmt->executeUpdate(“INSERT  INTO  basket_tab  VALUES(‘MANGOES’,3)”);
```

重复使用语句句柄

我们可以重复使用一个语句句柄来执行多次 SQL 语句。例如，使用不同的参数来重复执行相同的语句，通过语句句柄的 setSQL 方法来说明所要执行的语句。

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

我们可移植性这个 SQL 语句我们想要的次数。如果我们想执行其他 SQL 语句，只需重置语句句柄。例如：

```
stmt->setSQL(“SELECT   *   FROM   basket_tab   WHERE   quantity   >= :1”);
```

我们可以通过 getSQL 方法获取当前语句句柄的内容。

终止语句句柄

我们需要显式地终止和释放一个语句：

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

10.4 OCCI 环境中的 SQL 语句的种类

在一个 OCCI 环境中有三种 SQL 语句。

- 标准语句—含有指定值的 SQL 指令。
- 参数化语句—有参数或者绑定变量的语句。
- 可调用语句—调用保存的 PL/SQL 语句。

标准语句是参数化语句的全集，参数化语句是可调用语句的全集。

10.4.1 标准语句

上面的两个例子描述了 DDL 和 DML 命令。例如

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

和

```
stmt->executeUpdate(“INSERT INTO basket_tab
      VALUES(‘MANGOES’, 3)”);
```

这两个例子都是关于标准语句的例子。CREATE TABLE 语句说明了表名(basket_tab)，INSERT 语句说明了要被插入的值(‘MANGOES’,3)。

10.4.2 参数化语句

我们通过为语句中的输入变量设置占位符来执行同样的语句。因为这些语句能够通过参数来接收输入。

例如，如果我们想使用不同的参数来执行 INSERT 语句。我们首先使用语句句柄的 setSQL 方法来指明语句。

```
Stmt->setSQL(“INSERT   IINTO   basket_tab   VALUES(:1,   :2)”);
```

然后调用 setxxx 方法来指明参数，xxx 代表参数的类型。下面的例子调用了 setString 和 setInt 方法来为第一个参数和第二个参数输入值。

插入一行

```
Stmt->setString(1, “Banana”);          //第一个参数的值
Stmt->setInt(2, 5)                     //第二个参数的值
```

指明参数后，我们将这些值插入到数据库中。

```
Stmt->executeUpdate();            //执行语句
```

插入另一行数据：

```
Stmt->setString(1, "Apples");     //第一个参数的值
Stmt->setInt(2, 9);               //第二个参数的值
```

再插入一行数据。

```
stmt->executeUpdate();            //执行语句
```

10.5 执行 SQL 查询

查询返回的结果为结果集 result set。

10.6 OCCI 类和方法

表 10-1 列出了 OCCI 类。

表 10-1 OCCI 类

Class	Description
Bfile Class on page 8-5	Provides access to a SQL BFILE value.
Blob Class on page 8-13	Provides access to a SQL BLOB value.
Bytes Class on page 8-24	Examines individual bytes of a sequence for comparing bytes, searching bytes, and extracting bytes.
Clob Class on page 8-27	Provides access to a SQL CLOB value.
Connection Class on page 8-40	Represents a connection with a specific database.
ConnectionPool Class on page 8-45	Represents a connection pool with a specific database.
Date Class on page 8-51	Specifies abstraction for SQL DATE data items. Also provides formatting and parsing operations to support the OCCI escape syntax for date values.
Environment Class on page 8-64	Provides an OCCI environment to manager memory and other resources of OCCI objects. An OCCI driver manager maps to an OCI environment handle.
IntervalDS Class on page 8-70	Represents a period of time in terms of days, hours, minutes, and seconds.
IntervalYM Class on page 8-83	Represents a period of time in terms of year and months.
Map Class on page 8-95	Used to store the mapping of the SQL structured type to C++ classes.
MetaData Class on page 8-97	Used to determine types and properties of columns in a `ResultSet`, that of existing schema objects in the database, or the database as a whole.
Number Class on page 8-104	Provides control over rounding behavior.

Class	Description
PObject Class on page 8-130	When defining types, enables specification of persistent or transient instances. Class instances derived from `PObject` can be either persistent or transient. If persistent, a class instance derived from `PObject` inherits from the `PObject` class; if transient, there is no inheritance.
Ref Class on page 8-137	The mapping in C++ for the SQL REF value, which is a reference to a SQL structured type value in the database.
RefAny Class on page 8-144	The mapping in C++ for the SQL REF value, which is a reference to a SQL structured type value in the database.
ResultSet Class on page 8-147	Provides access to a table of data generated by executing an OCCI `Statement`.
SQLException Class on page 8-169	Provides information on database access errors.
Statement Class on page 8-171	Used for executing SQL statements, including both query statements and insert / update / delete statements.
Stream Class on page 8-215	Used to provide streamed data (usually of the LONG datatype) to a prepared DML statement or stored procedure call.
Timestamp Class on page 8-219	Specifies abstraction for SQL TIMESTAMP data items. Also provides formatting and parsing operations to support the OCCI escape syntax for time stamp values.

Overview of OCI Multithreaded Development

Threads are lightweight processes that exist within a larger process. Threads share the same code and data segments but have their own program counters, machine registers, and stacks. Global and static variables are common to all threads, and a mutual exclusivity mechanism is required to manage access to these variables from multiple threads within an application.

Once spawned, threads run asynchronously with respect to one another. They can access common data elements and make OCI calls in any order: Because of this shared access to data elements, a synchronized mechanism is required to maintain the integrity of data being accessed.

The mechanism to manage data access takes the form of mutexes (mutual exclusively locks), that is implemented to ensure that no conflicts arise between multiple threads accessing shared. In OCI, mutexes are granted for each environment handle.

The thread safety feature of the Oracle database server and OCI libraries allows developers to use the OCI in a multithreaded environment. Thread safety ensures code can be reentrant, with multiple threads making OCI calls without side effects.

The OCIThread Package

The OCIThread package provides a number of commonly used threading primitives. It offers a portable interface to threading capabilities native to various operating systems, but does not implement threading on operating system that do not have native threading capability.

OCIThread does not provide a portable implementation, but it serves as a set of portable covers for native multithreaded facilities. Therefore, operating systems that do not have native support for multithreading will only be able to support a limited implementation of the

OCIThread package. As a result, products that rely on all of OCIThread's functionality will not port to all operating systems. Products that must port to all operating systems must use only subset of OCIThread's functionality.

The OCIThread API consists of three main parts. Each part is described briefly here. The following subsections describe each in greater detail.

• **Initialization and Termination**. These calls deal with the initialization and termination of OCIThread context, which is required for other OCIThread calls.

OCIThread only requires that the process initialization function, OCIThreadProcessInit(), is called when OCIThread is being used in a multithreaded application. Failing to call OCIThreadProcessInit() in a single-threaded application is not an error.

Separate calls to OCIThreadInit() will all return the same OCIThread context. Each call to OCIThreadInit() must eventually be matched by a call to OCIThreadTerm().

• **Passive Threading Primitives.** *Passive* threading primitives are used to manipulate mutual exclusion locks (mutex), thread IDs, and thread-specific data keys. These primitives are described as passive because while their specifications allows for the existence of multiple threads, they do not require it. It is possible for these primitives to be implemented according to specification in both single-threaded and multithreaded environments. As a result, OCIThread clients that use only these primitives will not require a multiple-thread environment in order to work correctly. They will be able to work in single-threaded environments without branching code.

• **Passive Threading Primitives.** Active threading primitives deal with the creation, termination, and manipulation of threads. These primitives are described as *active* because they can only be used in true multithreaded environments. Their specification explicitly requires multiple threads. If you need to determine at runtime whether or not you are in a multithreaded environment, call OCIThreadIsMulti() before using an OCIThread active primitives.

Initialization and Termination

The types and functions described in this section are associated with the initialization and termination of the OCIThread package. OCIThread must be initialized before any of its functionality can be used.

The observed behavior of the initialization and termination functions is the same regardless of whether OCIThread is in single-threaded or multithreaded environment. Table 9-1 lists functions for thread initialization and termination.

Table 9–1 Initialization and Termination Multithreading Functions

Function	Purpose
`OCIThreadProcessInit()`	Performs OCIThread process initialization.
`OCIThreadInit()`	Initializes OCIThread context.
`OCIThreadTerm()`	Terminates the OCIThread layer and frees context memory.
`OCIThreadIsMulti()`	Tells the caller whether the application is running in a multithreaded environment or a single-threaded environment.

OCIThread Context

Most calls to OCIThread functions use the OCI environment or user session handle as a parameter. The OCIThread context is part of the OCI environment or user session handle and it

must be initialized by calling OCIThreadInit(). Termination of the OCIThread context occurs by calling OCIThreadTerm().

注意：**The OCIThread context is an opaque data structure. Do not attempt to examine the contents of the context.**

Passive Threading Primitives

The passive threading primitives deal with the manipulation of mutex, thread ID's, and thread-specific data. Since the specifications of these primitives do not require the existence of multiple threads, they can be used both in multithreaded and single-threaded operating systems. Table 9-2 lists functions used to implement passive threading.

Table 9–2 Passive Threading Primitives

Function	Purpose
`OCIThreadMutexInit()`	Allocates and initializes a mutex.
`OCIThreadMutexDestroy()`	Destroys and deallocates a mutex.
`OCIThreadMutexAcquire()`	Acquires a mutex for the thread in which it is called.
`OCIThreadMutexRelease()`	Releases a mutex.
`OCIThreadKeyInit()`	Allocates and initializes a key.
`OCIThreadKeyDestroy()`	Destroys and deallocates a key.
`OCIThreadKeyGet()`	Gets the calling thread's current value for a key.
`OCIThreadKeySet()`	Sets the calling thread's value for a key.
`OCIThreadIdInit()`	Allocates and initializes a thread ID.
`OCIThreadIdDestroy()`	Destroys and deallocates a thread ID.
`OCIThreadIdSet()`	Sets on thread ID to another.
`OCIThreadIdSetNull()`	Nulls a thread ID.
`OCIThreadIdGet()`	Retrieves a thread ID for the thread in which it is called.
`OCIThreadIdSame()`	Determines if two thread IDs represent the same thread.
`OCIThreadIdNull()`	Determines if a thread ID is NULL.

OCIThreadMutex

The OCIThreadMutex datatype is used to represent a mutex. This mutex is used to ensure that:

- **only one thread accesses a given set of data at a time, or**
- **only one thread executes a given critical section of code at a time**

Mutex pointers can be declared as parts of client structures or as stand-alone variables. Before they can be used, they must be initialized using OCIThreadMutexInit(). Once they are no longer needed, they must be destroyed using OCIThreadMutexDestroy().

A thread can acquire a mutex by using OCIThreadMutexAcquire(). This ensures that only one thread at a time is allowed to hold a given mutex. A thread that holds a mutex can release it by calling OCIThreadMutexRelease().

OCIThreadKey

The datatype OCIThreadKey can be thought of as a process-wide variable with a thread-specific value. This means that all threads in a process can use a given key, but each

thread can examine or modify the key independently of the other threads. The value that a thread sees when it examines the key will always be the same as the value that it last set for the key. It will not see any values set for the key by other threads. The datatype of the value held by a key is a dvoid * generic pointer.

Keys can be created using OCIThreadKeyInit(). Key value are initialized to NULL for all threads.

A thread can set a key's value using OCIThreadKeySet(). A thread can get a key's value using OCIThreadKeyGet().

Connection Pooling in OCI

Connection pooling is the use of a group (the pool) of reusable physical connections by several sessions, in order to balance loads. The management of the pool is done by OCI, not the application. Applications that can use connection pooling include middle-tier applications for Web application servers and e-mail servers.

A sample usage of this feature is in a Web application server connected to a back-end Oracle database. Suppose that a Web application server gets several concurrent requests for data from the database server. The application can create a pool (or a set of pools) in each environment during application initialization.

OCI Connection Pooling Concepts

Oracle has several transaction monitor capabilities such as the fine-grained (细颗粒的) management of database sessions and connections. This is done by separating the notion of database sessions (user handles) from connections (server handles). Using these OCI calls for session switching and session migration, it is possible for an application server or transaction monitor to multiplex several sessions over fewer physical connections, thus achieving a high degree of scalability by pooling of connections and back-end Oracle server processes.

The connection pool itself is normally configured with a shared pool of physical connections, translating to a back-end server pool containing an identical number of dedicated server processes.

The number of physical connections is less than the number of database sessions in use by the application. The number of physical connections and back-end server processes are also reduced by using connection pooling. Thus many more database sessions can be multiplexed.

OCI Calls for Connection Pooling

The steps in using connection pooling in your application are:

- **Allocate the Pool Handle**
- **Create the Connection Pool**
- **Logon to the Database**
- **Deal with SGA Limitations in Connection Pooling**
- **Logoff from the Database**
- **Destroy the Connection Pool**

• **Free the Pool Handle**

1. **Allocate the Pool Handle**

Connection pooling requires that the pool handle OCI_HTYPE_CPOOL be allocated by OCIHandleAlloc(). Multiple pools can be created for a given environment handle.

For a single connection pool, here is an allocation example:

```
OCICPool *poolhp;
OCIHandleAlloc((    dvoid*) envhp,      (dvoid**) &poolhp,      OCI_HTYPE_CPOOL,
                    (size_t)0,          (dvoid**)   0);
```

2. **Create the Connection Pool**

The function OCIConnectionPoolCreate() initializes the connection pool handle. It has these IN parameters:

• connMin, the minimum number of connections to be opened when the pool is created.

• **c**onnIncr, the incremental number of connections that can be opened when all the connections are busy and a call needs a connection. This increment is used only when the total number of open connections is less than the maximum number of connections that can be opened in that pool.

• **c**onnMax, the maximum number of connections that can be opened in the pool. When the maximum number of connections are open in the pool, and all the connections are busy, if a call needs a connection, it will wait till it gets one. However, if the OCI_ATTR_CONN_NOWAIT attribute is set for the pool, an error is returned.

• A poolUsername, and poolPasswd, to allow user sessions to transparently migrate between connections in the pool.

• In addition, an attribute OCI_ATTR_CONN_TIMEOUT, can be set to time out the connections in the pool. Connections idle for more than this time are terminated periodically, to maintain a optimum number of open connections. If this attribute is not set, then the connections are never timed out.

All the preceding attributes can be configured dynamically. So the application has the flexibility of reading the current load (number of open connections and number of busy connections) and tuning these attributes appropriately.

If the pool attributes (connMax, connMin, connIncr) are to be changed dynamically, OCIConnectionPoolCreate() must be called with mode set to OCI_CPOOL_REINITIALIZE.

The OUT parameters poolName and poolNameLen will contain values to be used in subsequent OCIServerAttach() and OCILogon2() calls in place of the database name and the database name length arguments.

There is no limit on the number of pools that can be created by an application. Middle tier applications can take advantage of this feature and create multiple pools to connect to the same server or to different servers,

Here is an example of this call:

```
OCIConnectionPoolCreate((OCIEnv*)envhp,
                        (OCIError*) errhp,        (OCICPool*) poolhp,
                        &poolName,   &poolNameLen,
                        (text*)database,     strlen(database),
                        (ub4)conMin,         (ub4)connMax,       (ub4)connIncr,
                        (text*)pooluser,     strlen(pooluser),
```

```
(text*)poolPasswd,        strlen(poolPasswd),
OCI_DEFAULT);
```

Logon to the Database

The application will need to log on to the database for each thread, using one of the following interface.

·OCILogon2()

This is the simplest interface. Use this interface when you need a simple Connection Pool connection and do not need to alter any attributes of the session handle. This interface can also be used to make proxy connections to the database.

Here is an example using OCILogon2():

```
for (i=0;    I < MAXTHREADS;    ++i)
{
    OCILogon2(envhp, errhp, &svchp[i], "hr", 2, "hr", 2 , poolName,
                    poolNameLen,    OCI_LOGON2_CPOOL));
}
```

In order to use this interface to get a proxy connection, set the password parameter to NULL.

・OCISessionGet()

This is the recommended interface. It gives the user the additional option of using external authentication methods, such as certifications, distinguished name, and so on. OCISessionGet() is the recommended uniform function call to retrieve a session.

Here is an example using OCISessionGet():

```
For(I = 0; i< MAXTHREADS;    ++i)
{
    OCISessionGet(envhp,    errhp,    &svchp,    authp,
                    (OraText*) poolName,
                    Strlen(poolName),        NULL, 0, NULL, NULL, NULL,
                    OCI_SESSGET_CPOOL);

}
```

3.

11. OCI 多线程开发

线程是存在于一个更大的进程中的轻量级的进程。线程共享同样的代码和数据段，但是有它们自己的程序计数器，寄存器和栈。线程共享全局和静态变量，互斥锁机制用来管理一个应用程序中的多个线程对这些变量的访问。

11.1 实现线程安全 Implementing Thread Safety

为了利用线程安全的优势，应用程序必须运行在一个线程安全的 thread-safe 操作系统。应用程序通过使用 OCIEnvNlsCreate()函数的 OCI_THREADED 模式来运行多线程环境。

所有后续的对 OCIEnvNlsCreate()函数的调用也要使用 OCI_THREADED 模式。

如果一个应用程序时单线程的，不论操作系统是否为线程安全的 thread-safe，应用程序调用 OCIInitialize()或者 OCIEnvNlsCreate()是必须使用 OCI_DEFAULT。运行在 OCI_THREADED 模式下的单线程应用程序可能运行的效率低一些。

如果一个多线程应用程序运行在一个线程安全的操作系统下，OCI 库将会为应用程序管

理互斥锁。一个应用程序可以不用这个特点而维持其自己的互斥机制通过调用 OCIEnvCreate()时，使 mode 参数为 OCI_NO_MUTEX。

11.2 OCIThread 包 Package

OCIThread 包提供了一些广泛使用的线程原语。它对不同操作系统的线程能力提供了一个可移植的接口。

OCIThread API 有三部分构成。包括：

• 初始化和终止 Initialization 和 Termination。这些函数处理 OCIThread 上下文的初始化和终止，这是其他 OCIThread 调用所必须的。

OCIThread 只需要在一个多线程环境中当 OCITread 被使用时，进程初始化函数 OCIThreadProcessInit()函数被调用。

调用 OCIThreadInit 函数会返回同样的 OCIThread 上下文。每调用一次 OCIThreadInit()函数最后必须对应一个 OCIThreadTerm()函数。

• 被动线程原语 Passive Threading Primitives。被动线程原语用来操作互斥锁、线程 ID、和线程相关数据键。由于它们允许多线程的存在，但并不需要多线程，所以称其为被动的。有可能它们在单线程或者多线程环境中实现。所以仅仅使用这些原语的 OCIThread 客户并不一定需要一个多线程环境。它们可以运行在单线程环境下。

• 主动线程原语 Active Threading Primitives。主动线程原语处理线程的创建、终止和操作。由于它们只能在真正的多线程环境中使用，所以它们被称为主动的。

11.3 初始化和终止 Initialization and Termination

这部分描述与 OCIThread 包的初始化和终止有关的类型和函数。在其功能被使用之前，OCIThread 一定要被初始化。

不论 OCIThread 是在单线程还是多线程环境下，初始化和终止函数的行为都是相同的。表 9-1 列出了线程初始化和终止的函数

表 9-1 初始化和终止多线程函数

Function	Purpose
OCIThreadProcessInit()	Performs OCIThread process initialization.
OCIThreadInit()	Initializes OCIThread context.
OCIThreadTerm()	Terminates the OCIThread layer and frees context memory.
OCIThreadIsMulti()	Tells the caller whether the application is running in a multithreaded environment or a single-threaded environment.

11.3 OCIThread 上下文

大多数 OCIThread 函数使用 OCI 环境句柄或者用户会话句柄作为一个参数。OCIThread 上下文是 OCI 环境或者用户会话句柄的一部分，必须调用 OCIThreadInit()函数来将其初始化。通过调用 OCIThreadTerm()函数来终止 OCIThread 上下文。

注意：OCIThread 上下文是一个不透明的数据结构。不要试图查看上下文中的内容。

11.4 被动线程原语

表 9-2 列出了实现被动线程的函数

表 9-2 被动线程原语

Function	Purpose
OCIThreadMutexInit()	Allocates and initializes a mutex.
OCIThreadMutexDestroy()	Destroys and deallocates a mutex.
OCIThreadMutexAcquire()	Acquires a mutex for the thread in which it is called.
OCIThreadMutexRelease()	Releases a mutex.
OCIThreadKeyInit()	Allocates and initializes a key.
OCIThreadKeyDestroy()	Destroys and deallocates a key.
OCIThreadKeyGet()	Gets the calling thread's current value for a key.
OCIThreadKeySet()	Sets the calling thread's value for a key.
OCIThreadIdInit()	Allocates and initializes a thread ID.
OCIThreadIdDestroy()	Destroys and deallocates a thread ID.
OCIThreadIdSet()	Sets on thread ID to another.
OCIThreadIdSetNull()	Nulls a thread ID.
OCIThreadIdGet()	Retrieves a thread ID for the thread in which it is called.
OCIThreadIdSame()	Determines if two thread IDs represent the same thread.
OCIThreadIdNull()	Determines if a thread ID is NULL.

11.4.1 OCIThreadMutex

OCIThreadMutex 数据类型用来代表一个互斥锁。这个互斥锁用来确保：

- 在某一时刻只有一个线程访问某一集合的数据
- 在某一时刻只有一个线程执行某一个临界区

互斥锁指针可以被声明为客户数据结构的一部分或者一个单独的变量。在它们可以被使用之前，必须使用 OCIThreadMutexInit()函数来初始化它们。一旦不再需要它们，必须使用 OCIThreadMutexDestroy()来销毁它们。

一个线程可以使用 OCIThreadMutexAcquire()函数来获取一个互斥锁。这确保了在某一时刻只有一个线程被允许持有互斥锁。持有互斥锁的线程可以通过调用 OCIThreadMutexRelease()函数来释放互斥锁。

11.4.2 OCIThreadKey

OCIThreadKey

12.