OTL 使用指南

长春孔辉汽车科技有限公司

文档修改历史

日期	版本	作者	修改内容	评审号	变更控制号	发布日期
2010-7-14	0.1		初稿			
2010-07-18	0.2		增加编程实践相关的内容，主要通过总结 OCS 租费开发中 OTL 的使用经验，整理出 13 章最佳实践。			
2010-07-28	0.3		增加 13.3 小节与开源项目 ORAPP 的性能对比			

目　　录

1 OTL 简介

OTL 是 Oracle, Odbc and DB2-CLI Template Library 的缩写，是一个 C++编译中操控关系数据库的模板库，它目前几乎支持当前所有的各种主流数据库，例如 Oracle, MS SQL Server, Sybase, Informix, MySQL, DB2, Interbase / Firebird, PostgreSQL, SQLite, SAP/DB, TimesTen, MS ACCESS 等等。

OTL 中直接操作 Oracle 主要是通过 Oracle 提供的 OCI 接口进行，操作 DB2 数据库则是通过 CLI 接口进行，至于 MS 的数据库和其它一些数据库，OTL 只提供了 ODBC 的操作方式。当然 Oracle 和 DB2 也可以由 OTL 间接使用 ODBC 的方式进行操纵。

在 MS Windows and Unix 平台下，OTL 目前支持的数据库版本主要有：Oracle 7 (直接使用 OCI7), Oracle 8 (直接使用 OCI8), Oracle 8i (直接使用 OCI8i), Oracle 9i (直接使用 OCI9i), Oracle 10g (直接使用 OCI10g), DB2 (直接使用 DB2 CLI), ODBC 3.x ,ODBC 2.5。目前 OTL 的最新版本为 4.0，参见 http://otl.sourceforge.net/，下载地址 http://otl.sourceforge.net/otlv4_h.zip。

2 编译 OTL

OTL 是一个集成库，它包含了一个模板流框架(template stream framework)以及适配 OCI7, OCI8, OCI8i, OCI9i, OCI10g, ODBC 2.5, ODBC 3.x, DB2 CLI 和 Informix CLI 的适配器(OTL-adapters)。编译时需要使用相应的宏定义向编译器指明底层数据库 API 的类型。例如，如果底层使用 ORACLE10g 的 API，则需要使用宏定义”#define OTL_ORA10G”。

另外，也可以使用相应的宏定义控制编译器对 OTL 的编译。例如，如果需要和 ACE 库一起编译可以使用宏定义”#define OTL_ACE”，如果需要 OTL 为所分配并处理的字符串以空字符结尾成为 C 风格字符串则可以使用宏定义”#define OTL_ADD_NULL_TERMINATOR_TO_STRING_SIZE”等。

所有的相关宏请参见 http://otl.sourceforge.net/otl3_compile.htm。

3 基本使用

OTL 的一般使用步骤包括：

(1) 使用宏指明底层数据库 API 类型和控制编译器对 OTL 的编译。例如：

```
#define OTL_ORA9I          // Compile OTL 4.0/OCI9i
#define OTL_UNICODE        // Enable Unicode OTL for OCI9i
```

(2) 创建 otl_connect 对象，该对象一般为全局共享的。

(3) 调用 otl_connect 的静态方法 otl_initialize()初始化 OTL 环境。

(4) 调用 otl_connect 的 rlogon()方法连接数据库。

(5) 创建 otl_stream()对象，该对象一般为局部的。

(6) 调用 otl_stream 的 open()方法打开 SQL 进行解析。

(7) 使用 otl_stream 的<<操作符绑定 SQL 中的变量。

(8) 使用 otl_stream 的>>操作符读取返回结果。

(9) 调用 otl_connect 的 logoff()方法从数据库断开。

下面将通过一个较为全面的示例说明使用 OTL 连接数据库、创建表和存储过程、调用存储过程、查询记录以及插入记录、从数据库断开的具体代码实现。

```
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <vector>

#define OTL_ORA9I             // Compile OTL 4.0/OCI9i
//#define OTL_UNICODE      // Enable Unicode OTL for OCI9i
#include "otlv4.h"                // include the OTL 4.0 header file

using namespace std;

/**
  *连接数据库
  */
int OTLConnect (const char* pszConnStr, otl_connect& db)
{
    try
    {
        otl_connect::otl_initialize(); // initialize OCI environment
        db.rlogon(pszConnStr);
        db.auto_commit_off ( );
        printf ( "CONNECT: OK!\n" );
```

```
    }
    catch(otl_exception& p)
    {    // intercept OTL exceptions
        printf ( "Connect Error: (%s) (%s) (%s)\n",p.msg, p.stm_text, p.var_info );
        return -1;
    }
    return 0;
}

/**
  *从数据库断开
  */
int OTLDisconnect (otl_connect& db)
{
    db.commit ( );
    db.logoff();

    printf ( "DISCONNECT: OK!\n" );
    return 0;
}

/**
  *创建数据库表和存储过程
  */
int OTLExec ( otl_connect& db)
{
    try
    {
        int nCnt = 0;
        char strSql[] = "SELECT count(0) FROM user_tables "
                        " WHERE table_name = 'TEST_FTP' ";

        otl_stream otlCur (1, (const char*)strSql, db );
```

```
        otlCur >> nCnt;

        if ( nCnt == 0 )
        {
            char strDDL[] =
                "create table TEST_FTP                "
                "(                                                "
                "    AREA_ID              VARCHAR2(100) not null,       "
                "    FTP_FILE_NAME        VARCHAR2(100) not null,       "
                "    FTP_TIME             VARCHAR2(14),                 "
                "    FTP_BEGIN_TIME       VARCHAR2(14),                 "
                "    FTP_END_TIME         VARCHAR2(14),                 "
                "    FTP_MOD_TIME         date,                         "
                "    FTP_SIZE             NUMBER(8),                    "
                "    FTP_SOURCE_PATH VARCHAR2(100),                     "
                "    FTP_LOCAL_PATH       VARCHAR2(100),                "
                "    FTP_RESULT           VARCHAR2(4),                  "
                "    FTP_REDO             VARCHAR2(1)                   )";

            otl_cursor::direct_exec ( db, (const char*)strDDL );
        }

        char strSqlProc[] = "SELECT count(0) from user_objects "
                    "  WHERE object_type = 'PROCEDURE' and object_name =
'PR_REMOVE_FTP' ";
        otl_stream otlCurProc (1, (const char*)strSqlProc, db );
        otlCurProc >> nCnt;

        if ( nCnt == 0 )
        {
            char strProc[] =
            "CREATE OR REPLACE procedure pr_remove_ftp            "
            "    ( area in varchar2, out_flag out varchar )     "
```

```
                "AS                                                    "
                "strtmp varchar2(32);                                  "
                "BEGIN                                                     "
                "    strtmp := area||'%';                                  "
                "    DELETE FROM TEST_FTP where area_id LIKE strtmp;       "
                "    out_flag := 'OK';                                 "
                "END;                                                      ";

                otl_cursor::direct_exec ( db, (const char*)strProc );
            }

        }
        catch(otl_exception& p)
        {     // intercept OTL exceptions
            printf ( "EXECUTE Error: (%s) (%s) (%s)\n",p.msg, p.stm_text, p.var_info );
        }
        return 0;
}

/**
   *调用存储过程
   */
int OTLProcedure (otl_connect& db )
{
        try
        {

            char szData[64],    szData1[64],    szData2[64], szData3[64];
            int nSize = 0;
            char strSql[] = " BEGIN "
                            "  pr_remove_ftp ( :area<char[100],in>, :out<char[100],out> ); "
                            " END; ";
            otl_stream otlCur (1, (const char*)strSql, db );
```

```
        otlCur.set_commit ( 0 );

        strcpy ( szData, "AREA" );
        memset ( szData1, 0, sizeof(szData1) );
        memset ( szData2, 0, sizeof(szData2) );
        memset ( szData3, 0, sizeof(szData3) );

        otlCur << szData;
        otlCur >> szData1;

        printf ( "PROCEDURE: %s!\n", szData1 );
    }
    catch(otl_exception& p)
    { // intercept OTL exceptions
        printf ( "PROCEDURE Error: (%s) (%s) (%s)\n",p.msg, p.stm_text, p.var_info );
    }
    return 0;
}

/**
  *查询记录
  */
int OTLSelect (otl_connect& db)
{
    try
    {
        char szData[64],    szData1[64],    szData2[64], szData3[64], szRedo[2];
        int nSize;
        char strSql[] = " SELECT area_id, ftp_time, ftp_file_name, "
                        " to_char(ftp_mod_time, 'YYYY-MM-DD HH24:MI:SS'), ftp_size "
                        "    FROM TEST_FTP "
                        " WHERE ftp_redo = :ftp_redo<char[2]>" ;
        otl_stream otlCur (1, (const char*)strSql, db );
```

```
        strcpy ( szRedo, "Y" );
        otlCur << szRedo;
        while ( !otlCur.eof() )
        {
            memset ( szData, 0, sizeof(szData) );
            otlCur >> szData;
            otlCur >> szData1;
            otlCur >> szData2;
            otlCur >> szData3;
            otlCur >> nSize;
            printf ( "SELECT: (%s %s %s %s %d)\n",
                    szData, szData1, szData2, szData3, nSize );
        }
    }
    catch(otl_exception& p)
    { // intercept OTL exceptions
        printf ( "Select Error: (%s) (%s) (%s)\n",p.msg, p.stm_text, p.var_info );
    }
    return 0;
}

/**
  *插入记录
  */
int OTLInsert (otl_connect& db)
{
    try
    {
        char szData[64],    szData1[64],    szData2[9], szData3[64], szRedo[2];
        int nSize;
        char strSql[] = " INSERT into TEST_FTP "
                        " ( area_id, ftp_file_name, ftp_time, ftp_mod_time, ftp_size, ftp_redo )"
```

```
                    " VALUES ( :area_id<char[100]>, "
                    "     :ftp_file_name<char[100]>, "
                    "     to_char(sysdate,'YYYYMMDDHH24MISS'), "
                    "     to_date(:ftp_mod_time<char[20]>,'YYYYMMDD'), "
                    "     :ftp_size<int>, "
                    "     :ftp_redo<char[2]> ) ";
    otl_stream otlCur (1, (const char*)strSql, db );

    otlCur.set_commit ( 0 );

    for ( int i = 1; i < 10; i ++ )
    {
        sprintf ( szData, "AREA_%d", i );
        sprintf ( szData1, "FILE_NAME_%d", i );
        if ( i < 5 )
        {
            sprintf ( szData2, "20070415" );
            strcpy ( szRedo, "Y" );
        }
        else
        {
            sprintf ( szData2, "20070416" );
            strcpy ( szRedo, "N" );
        }

        memset ( szData3, 0, sizeof(szData3) );
        nSize = i * 100;

        otlCur << szData << szData1 << szData2 << nSize << szRedo;
    }

    printf ( "INSERT: OK!\n" );
}
```

```
    catch(otl_exception& p)
    { // intercept OTL exceptions
        printf ( "INSERT Error: (%s) (%s) (%s)\n",p.msg, p.stm_text, p.var_info );
    }
    return 0;
}

/**
  *主函数
  */
int main ( int argc, char *argv[] )
{
    otl_connect db;
    char szConn[64];



    if ( argc >= 2 )
        strcpy ( szConn, argv[1] );
    else
    {
        printf ( "otltest conn_str" );
        return -1;
    }

    if ( OTLConnect ( szConn, db ) < 0 )
        return 0;
    OTLExec ( db );
    OTLProcedure ( db );
    OTLInsert ( db );
    OTLSelect ( db );
    OTLDisconnect ( db );

    return 0;
```

```
}
```

4 OTL 流的概念

OTL 设计者认为，任何 SQL 语句、PL/SQL 块或存储过程调用都被输入和输出变量特征化。例如：

- 一个 SELECT 语句在其 WHERE 子句中拥有标量的输入变量，而在其 SELECT 子句则定义了输出的列，如果 SELECT 语句返回的是多行记录则输出列是个向量参数。
- 一个 INSERT 和 UPDATE 语句需要将数据写入表中，它们拥有输入参数。另外，一个 DELETE 语句由于需要指明删除记录的类型，同样拥有输入。工业强度的数据库服务器通常也支持批量操作，例如批量的查询、更新、删除和插入，因此 INSERT/UPDATE/DELETE 语句的参数在批量操作的情况下也可能是向量。
- 一个存储过程可能含有输入和(或)输出参数。通常存储过程的参数是标量，但是也有特例，例如返回的是引用游标(Oracle)或者记录集(MS SQL SERVER 或者 Sybase)。
- 一个 PL/SQL 块可能含有输入和(或)输出参数，这些参数可能是标量也可能是向量。

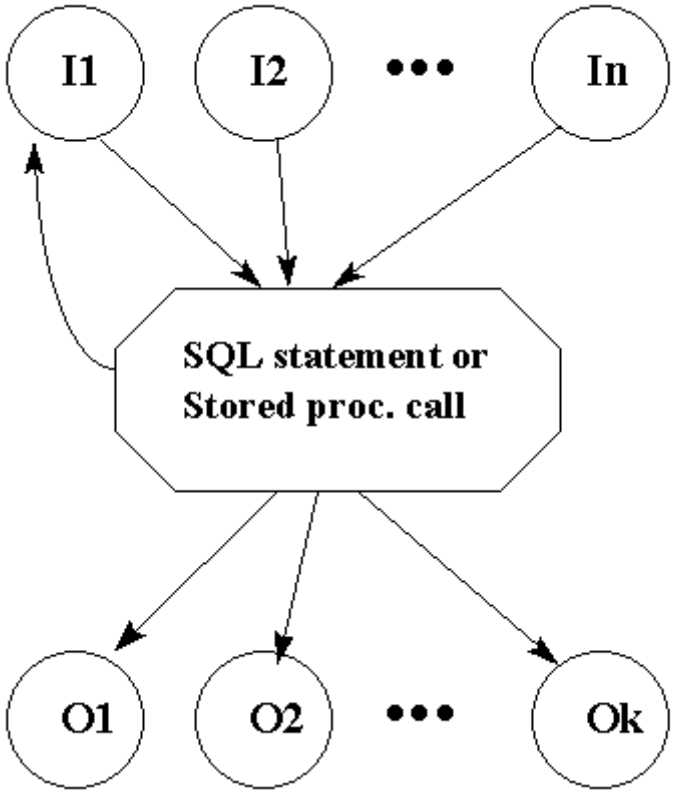

图 4-1 OTL 的流

因此，任何的 SQL 或者其程序上的扩展在交互过程中都可以如图 4-1 所示看作拥有输入和输出的黑盒。OTL 通过将数据流和 SQL 的概念联合起来，用 otl_stream 类表达

这种抽象。

由于 SQL 语句可能以批量的方式执行，otl_stream 是一个缓冲流。它拥有两个独立的缓冲区：输入和输出。输入缓冲区由所有集中到一起的输入参数组成，输出缓冲区则由所有集中到一起的输出变量组成。

OTL 流和 C++的缓冲流很相似。一个 SQL 语句或存储过程调用被当作一个普通的缓冲流被打开。OTL 流的操作逻辑和 C++流操作逻辑基本相同，但是 OTL 流的输出和输出缓冲区可能重叠。

OTL 流拥有 flush()方法在输入缓冲区写满的时候将其自动刷新，也含有一系列的<<和>>运算符来读和写不同数据类型的对象。它最重要的优点是为任何类型的 SQL 语句和存储过程调用提供了统一的接口。应用开发者能够通过熟悉少量的语法和函数名称像使用 C++流一样来使用 OTL 流。

在 OTL 流的内部拥有一个小型的解析器来解析所声明的绑定变量以及绑定变量的数据类型。因此，免去了使用特殊的绑定函数来绑定已声明的 C/C++主机变量(host variables)。由于所有必须的缓冲区在 OTL 流中会自动创建，因此 OTL 仅仅需要被打开来进行读和写相应的数值。

OTL 流接口要求使用 OTL 异常。OTL 流操作都能可能抛掷 otl_exception 异常。因此为了拦截异常并阻止程序异常终止，必须使用 try/catch 块来包裹 OTL 流的使用代码。

OTL 流的实现 otl_stream 具有较高的自动化功能，当 OTL 流的所有的输入变量被定义好(也就是输入缓冲区被填满)，它会触发 OTL 流中的黑盒来执行。在黑盒执行的过程中输出缓冲区被填充。在执行完成后，输出缓冲区中的值能够从 OTL 流中被读取。如果执行的是一个 SELECT 语句并且返回多于输出缓冲区大小的行，那么在输出缓冲区的内容被读取后，OTL 会自动读取下一批行记录到输出缓冲区。

5 主要类及方法说明

5-1 OTL 主要类说明

类名	说明
otl_connect	负责创建和处理连接对象以及事务管理。
otl_stream	OTL 流概念(参见第 4 小节)的具体实现。任何具有输入输出的 SQL 语句，匿名的 PL/SQL 块或者存储过程能够使用 otl_stream 类进行 C++编程。

	一般传统的数据库 API 拥有绑定主机变量到 SQL 语句中占位符的函数。因此，开发者需要在程序中声明 host array，解析 SQL 语句，调用绑定函数，填充输入变量，执行 SQL 语句，读输出变量等。这些操作结束后又继续填充输入变量，执行 SQL 语句，读输出变量。 以上的所有事情能够在 otl_stream 中全部自动完成。otl_stream 在保证性能的情况下提供了完全自动的与数据库的交互。 otl_stream 的性能主要被缓冲区大小 arr_size 一个参数控制。缓冲区大小定义了插入表的逻辑行以及与数据库一次往反交互(one round-trip to the database)过程中从表或视图中查询的逻辑行。
otl_exception	可能代表数据库错误也可能代表 OTL 自身的错误。OTL 函数如果在使用底层的数据库 API 时返回非 0 的错误码，则会产生 otl_exception 类型的异常。

~~4.1~~ otl_stream 的主要方法

5.1

5-2 类 otl_stream 的主要方法说明

主要方法	说明
otl_stream(const int arr_size, const char* sqlstm, otl_connect& db, const char* ref_cur_placeholder=0, const char* sqlstm_label=0);	构造函数，负责创建 otl_stream 对象并调用 open()方法。 参数 arr_size 为流的缓冲区大小， 参数 db 为 otl_connect 连接对象的引用， 参数 ref_cur_placeholder 为 reference cursor 的占位符名称， 参数 sqlstm_label 为 SQL 语句的标签，用于取代异常描述消息 otl_exception::stm 中默认填充的 SQL 语句。
void open(const int arr_size,	打开 SQL 语句对其进行解析，所有的输入和输出变量都在流内部别动态分配，并且自动

const char* sqlstm, otl_connect& db, const char* ref_cur_placeholder=0, const char* sqlstm_label=0);	绑定到占位符。
void close(void); #ifdef OTL_STREAM_POOLING_ON void close(constbool save_in_stream_pool=true); #endif	关闭流。如果使用了流缓冲池，则可以使用带 save_in_stream_pool 参数的 close()函数。如果参数 save_in_stream_pool 为 true 则流并不会真正关闭，而是被缓冲池回收。
int good(void);	测试流是否打开。
int eof(void);	测试是否所有的数据都已经从流中读取。
void rewind(void);	重绕流。如果流不含有任何输入变量，该方法将强制流执行 SQL 语句。

operator int(void);	流到 int 的转换操作符，返回从!eof()获得的流状态。它允许运算符>>返回!EOF 的流状态并且能够在 while()循环中像下面的代码那样使用:
	while(s>>f1>>f2) {
	cout<<"f1="<<f1<<", f2="<<f2<<endl;
	}

int is_null(void);	判断是否从流中获得 NULL
void flush(void); void flush // **OTL/OCI8,8i,9i,10g only** (const int row_offset=0, const bool force_flush=false);	刷新流的输出缓冲区。这实际上意味着批量执行 SQL 语句，其执行的批量和流输出缓冲区中已经填充的 SQL 数相等。当输出缓冲区被填满时，缓冲区将被自动刷新。如果流的 auto_commit 标志被置上，则在刷新完毕后当前事务被提交。 OTL/OCI8,8i,9i,10g 拥有另外一个版本的 flush()方法。能够通过参数 row_offset 指定缓

	冲区刷新的开始位置，通过参数 force_flush 则能够指定是否在出现错误抛出 otl_exception 的情况下仍然强制继续刷新，忽略之前的错误。
long get_rpc(void);	获得处理的行数(Rows Processed Count, RPC)。
void set_column_type(const int column_ndx, const int col_type, const int col_size=0);	设置 SELECT 输出列的数据类型。 参数 column_ndx 为列的索引，如 1,2,3… 参数 col_type 为 OTL 定义的数据类型常量（参见 11.2 小节） 参数 col_size 为新数据类型的大小，只被 otl_var_char 类型使用。
void set_commit(int auto_commit=0);	设置流的 auto_commit 标志。默认情况下，该标志被置上，即当输出缓冲区刷新时，当前的事务被自动提交。 注意流的 auto_commit 标志和数据库库的自动提交模型没有任何关系。 如果需要默认方式为不自动提交的流，则可以使用 otl_stream 的子类 otl_nocommit_stream。
void set_flush(const bool auto_flush=true);	设置 auto_flush 标志。默认情况下 auto_flush 的值为 true, 即如果缓冲区出现脏数据则在流的析构函数中刷新缓冲区。如果自动刷新标志被关闭，则需要使用 close()方法或者 flush()方法对流进行刷新。 注意该函数仅仅能够设置流的析构函数中是否自动刷新，并不是通常意义上的缓冲区刷新。
int setBufSize(const int buf_size);	设置流缓冲区的大小。

~~4.2~~ 5.2 otl_connect 的主要方法

4-3 类 otl_connect 的主要方法说明

主要方法	说明
static int otl_initialize(const int threaded_mode=0);	初始化 OTL 环境。需要在程序最开始连接数据库之前调用一次。 参数 threaded_mode 指明程序是否运行在多线程环境，注意由于 OTL 并没有使用同步锁或者临界段，线程安全并不能够自动得到保证。
otl_connect(const char* connect_str, const int auto_commit=0);	构造函数。 参数 connect_str 为连接字符串,OTL/OCIx 风格的连接字符串为： “USER/PASSWORD”(本地 Oracle 连接) “USER/PASSWORD@TNS_ALLAS”(通过 SQL*Net 进行的远程连接) 参数 auto_commit 指明是否每一个在连接中执行的 SQL 语句都会自动提交。如果需要自动提交则为 1，默认情况下为 0 表示不需要自动提交。注意该 auto_commit 参数和 otl_stream 的自动提交没有任何关系。
void rlogon(const char* connect_str, const int auto_commit=0);	连接数据库。参数同构造函数。
void logoff(void);	断开数据库连接。
static int otl_terminate(void);	终止 Oracle 8i/9i 的 OCI 环境。需要在程序最后的数据库连接进行关闭后调用一次。该方法仅仅是 OCI Terminate()调用的包装。通常在多线程环境中，为了终止主

	线程的控制，该方法需要被调用使得进程能够从 OCI 客户端的共享内存中脱离以及做其他事情。
void cancel(void);// ***OTL/OCI8/8i/9i*** only	取消连接对象或者数据库会话中的正在执行或者活动的操作或数据库调用。

void commit(void);
#if defined(OTL_ORA10G_R2)
void commit_nowait(void);
#endif
	同步或异步的方式提交事务。

void rollback(void);	回滚事务。
void auto_commit_off(void); void auto_commit_on(void);	设置 otl_connect 对象的 auto_commit 标志。
void set_stream_pool_size(const int max_size =otl_max_default_pool_size);	如果使用了流缓冲池，则该方法重新分配被默认的流缓冲池和之前的 set_stream_pool_size()调用分配的所有资源。
void set_character_set(const int char_set=SQLCS_IMPLICIT);	如果使用了 UNICODE，则该方法设置默认或国家的字符集： SQLCS_IMPLICIT 为数据库默认字符集。 SQLCS_NCHAR 为数据库国家的字符集。
otl_connect& operator<<(const char* str);	发送字符串到 otl_connect 对象。如果该 otl_connect 对象还没有连接到数据库则字符串为"userid/passwd@db"格式的连接字符串，它使得 otl_connect 对象能够连接数据库。如果该 otl_connect 对象已经连接到数据库则字符串为静态 SQL 语句，该语句被马上执行。
otl_connect& operator<<=(const char* str);	发送字符串到 otl_connect 对象。otl_connect 对象将保存该字符串并被下一个>>操作符使用。该字符串是一个拥有占

	位符并且能够发送到 otl_stream 对象的 SQL 语句。
otl_connect& operator>>(otl_stream& s);	发送之前使用操作符<<=保存的 SQL 语句到 otl_stream 对象。它使得该 SQL 语句被 otl_stream 打开。 注意如果并没有被>>=操作符保存的字符串，则字符串"*** INVALID COMMAND ***"被发送到 otl_stream，最终会将导致解析错误，抛掷 otl_exception 异常。
long direct_exec(const char *sqlstm, int ignore_error = otl_exception::enabled);	直接执行静态的 SQL 语句，返回处理的行数。
void syntax_check(const char *sqlstm);	解析静态的 SQL 语句，如果出现 SQL 错误将抛掷 otl_exception 异常。
void server_attach(const char* tnsname=0, const char* xa_server_external_name=0, const char* xa_server_internal_name=0, #if defined(OTL_ORA_OCI_ENV_CREATE) bool threaded_mode=false #endif);	附加到 Oralcle。
void server_detach(void);	从 Oracle 分离。
void session_begin (const char* username, const char* password, const int auto_commit=0, const int session_mode=OCI_DEFAULT);	开始 Oracle8 会话。
void session_end(void);	结束 Oracle8 会话。

6 SQL 的变量绑定和常量 SQL

6.1 SQL 的变量绑定

OTL 拥有一个小型的解析器，负责在流的内部动态的为 SQL 语句、PL/SQL 块或存储过程调用中声明的绑定变量分配空间。OTL 将 Oracle 传统的使用命名符号作为占位符的变量绑定机制进行了扩展，增加了数据类型说明，例如：

INSERT INTO my_table2 values(**:employee_id<int>**,**:supervisor_name<char[32]**>)

OTL 占位符中的支持的数据类型如表 6-1 所示。

表 6-1 OTL 占位符中支持的数据类型

bigint	64-bit signed integer, for binding with BIGINT table columns (or stored procedure parameters) in MS SQL Server, DB2, MySQL, PostrgeSQL, etc. ODBC, and DB2 CLI support this kind bind variables natively, so does OTL. OCIs do not have native support for 64-bit integers, so OTL has to emulate it via string (<char[XXX]>) bind variables internally and does string-to-bigint and bigint-to-string conversion.
blob	for Oracle 8/9; BLOB
char[length**]** OTL 4.0.118 and higher: **char(**length**)**	null terminated string; length is database dependent; for Oracle in [3,32545]; for ODBC it depends on the database backend and the ODBC driver; for DB2-CLI >2. In Unicode OTL, this type of bind variable declaration means a null terminated Unicode character string (two bytes per character). The *length* field of this declarator needs to include an extra byte / Unicode character, in order to accomodate the null terminator itself (for example char[11] can be used in binding with a VARCHAR(9) column), unless #define OTL_ADD_NULL_TERMINATOR_TO_STRING_SIZE is enabled.
charz	Same as char[] for OTL_ORA7, OTL_ORA8,

	OTL_ORA8I, OTL_ORA9I, OTL_ORA10G. Should be used only when PL/SQL tables of type CHAR(XXX) are used. charz is actually a workaround for the following Oracle error: *PLS-00418: array bind type must match PL/SQL table row type.* Normally, the internal OCI datatype that is used to bind VARCHAR2/CHAR table columns / scalar PL/SQL procedure parameters works fine, except for PL/SQL tables of CHAR(XXX). PL/SQL engine does not like what OTL tries to bind with a PL/SQL table of CHAR(XXX). *charz[]* should be used instead of char[] in cases like that.
clob	for Oracle 8/9: CLOB, NCLOB
db2date	for DB2 DATEs; should be used in the binding of a placeholder with a DB2 DATE column in case of both
db2time	for DB2 TIMEs; should be used in the binding of a placeholder with a DB2 TIME column in case of both OTL/DB2-CLI and OTL/ODBC for DB2; requires otl_datetime as a data container. See example 91 for more detail.
double	8-byte floating point number
float	4-byte floating point number
int	32-bit signed int
ltz_timestamp	Oracle 9i TIMESTAMP WITH LOCAL TIME ZONE, in a combination with #define OTL_ORA_TIMESTAMP, and otl_datetime
nchar[length]	Same as char[] + otl_connect::set_character_set(SQLCS_NCHAR) for Oracle 8i/9i/10g only, under #define OTL_UNICODE., or #define OTL_ORA_UTF8. nchar[] is required only when both VARCHAR2/CHAR and NVARCHAR2/NCHAR need to be declared in the same SQL statement, or PL/SQL block.

nclob	Same as clob + otl_connect::set_character_set(SQLCS_NCHAR) for Oracle 8i/9i/10g only, under #define OTL_UNICODE, or #define OTL_ORA_UTF8. nclob is required only when both CLOB and NCLOB need to be declared in the same SQL statement, or PL/SQL block.
raw[length**]**	
raw_long	
short	short int (16-bit signed integer)
timestamp	MS SQL Server/Sybase DATETIME, DB2 TIMESTAMP, Oracle DATE, Oracle 9i TIMESTAMP (when #define OTL_ORA_TIMESTAMP is enabled) ; it requires TIMESTAMP_STRUCT (OTL/ODBC, OTL/DB2-CLI), or otl_datetime (ODBC, DB2-CLI, and OCIx). OTL/DB2-CLI and OTL/ODBC for DB2; requires otl_datetime as a data container. See example 91 for more detail
tz_timestamp	Oracle 9i TIMESTAMP WITH TIME ZONE, in a combination with #define OTL_ORA_TIMESTAMP, and otl_datetime
unsigned	unsigned int (32-bit unsigned integer)
varchar_long	for Oracle 7: LONG; for Oracle 8/9: LONG; for ODBC: SQL_LONGVARCHAR; for DB2: CLOB

为了区分 PL/SQL 块或存储过程中的输入和输出变量，OTL 引入了以下限定词：

- in – 输入变量
- out – 输出变量
- inout – 输入输出变量

其用法如下的 Oracle 示例代码片断所示。

```
BEGIN
  :rc<int,out> := my_func(:salary<float,in>,
                  :ID<int,inout>,
                  :name<char[32],out>
```

```
                    );
END;
```

6.2 常量 SQL

如果 SQL 语句 、PL/SQL 块或存储过程调用中不含有任何绑定变量，则可以称之为静态的。OTL 包含了静态方法执行静态语句，例如：

```
otl_cursor::direct_exec
   (db, // connect object
    "create table test_tab(f1 number, f2 varchar2(30))"
    );   // create table


  otl_cursor::direct_exec
   (db, // connect object
    "drop table test_tab", // SQL statement or PL/SQL block
    otl_exception::disabled // disable OTL exceptions,
                            // in other words, ignore any
                            // database error
   ); // drop table
```

otl_cursor 是 OTL4.0 的一个 internal class。OTL 虽然并不推荐使用低级别的类，但是 otl_cursor 的 direct_exec()方法是一个特例。该方法的返回值可能为：

- -1, 如果 otl_exception 异常被禁止使用(第二个参数被设置成 otl_exception::disabled)，并且底层的 API 返回了错误。
- >=0, 如果成功执行 SQL 命令，在执行 INSERT、DELETE 或 UPDATE 语句时实际返回的是已处理行数。

7 迭代器

7.1 OTL 流的读迭代器

OTL 提供了模板类 otl_stream_read_iterator 来扩展 OTL 流接口以支持迭代，该模板类提供了类似 JDBC 的传统 getter 接口，可以使用名称访问返回列。

以下是使用读迭代器的示例代码。

```
#include <iostream>
using namespace std;
```

```
#include <stdio.h>
//#define OTL_ORA7 // Compile OTL 4.0/OCI7
//#define OTL_ORA8 // Compile OTL 4.0/OCI8
//#define OTL_ORA8I // Compile OTL 4.0/OCI8i
#define OTL_ORA9I // Compile OTL 4.0/OCI9i
//#define OTL_ORA10G // Compile OTL 4.0/OCI10g
#define OTL_STREAM_READ_ITERATOR_ON
#define OTL_STL
#include <otlv4.h> // include the OTL 4.0 header file

otl_connect db; // connect object

void insert()
// insert rows into table
{
  otl_stream o(50, // buffer size
                   "insert into test_tab values(:f1<int>,:f2<char[31]>)", // SQL statement
                   db // connect object
                  );
  char tmp[32];

  for(int i=1;i<=100;++i){
    sprintf(tmp,"Name%d",i);
    o<<i<<tmp;
  }
}

void select()
{
  otl_stream i(50, // buffer size
                   "select * from test_tab "
              "where f1>=:f11<int> and f1<=:f12<int>*2",
                       // SELECT statement
                   db // connect object
```

```
                    );
    // create select stream

 int f1;
 char f2[31];
 otl_stream_read_iterator<otl_stream,otl_exception,otl_lob_stream> rs;

 rs.attach(i); // attach the iterator "rs" to the stream "i".
 i<<8<<8; // assigning :f11 = 8, :f12 = 8
    // SELECT automatically executes when all input variables are
    // assigned. First portion of output rows is fetched to the buffer

 while(rs.next_row()){ // while not end-of-data
     rs.get("F2",f2);
     rs.get("F1",f1);
     cout<<"f1="<<f1<<", f2="<<f2<<endl;
 }

 rs.detach(); // detach the itertor from the stream

 i<<4<<4; // assigning :f11 = 4, :f12 = 4
    // SELECT automatically executes when all input variables are
    // assigned. First portion of output rows is fetched to the buffer

 while(!i.eof()){ // while not end-of-data
     i>>f1>>f2;
     cout<<"f1="<<f1<<", f2="<<f2<<endl;
 }
}

int main()
{
 otl_connect::otl_initialize(); // initialize OCI environment
```

```
 try{

  db.rlogon("scott/tiger"); // connect to Oracle

  otl_cursor::direct_exec
   (
    db,
    "drop table test_tab",
    otl_exception::disabled // disable OTL exceptions
   ); // drop table

  otl_cursor::direct_exec
   (
    db,
    "create table test_tab(f1 number, f2 varchar2(30))"
    );    // create table

  insert(); // insert records into table
  select(); // select records from table

 }

 catch(otl_exception& p){ // intercept OTL exceptions
  cerr<<p.msg<<endl; // print out error message
  cerr<<p.stm_text<<endl; // print out SQL that caused the error
  cerr<<p.var_info<<endl; // print out the variable that caused the error
 }

 db.logoff(); // disconnect from Oracle

 return 0;

}
```

输出结果：

```
f1=8, f2=Name8
f1=9, f2=Name9
f1=10, f2=Name10
f1=11, f2=Name11
f1=12, f2=Name12
f1=13, f2=Name13
f1=14, f2=Name14
f1=15, f2=Name15
f1=16, f2=Name16
f1=4, f2=Name4
f1=5, f2=Name5
f1=6, f2=Name6
f1=7, f2=Name7
f1=8, f2=Name8
```

7.2 STL 兼容的迭代器

OTL 将泛型编程和 Oracle 紧密结合以构筑小容量的、可靠的、高性能并且容易维护的 C++数据库应用，为此分别提供了两个 STL 兼容的迭代器：otl_output_iterator<T>和 otl_input_iterator<T, Distance>。

otl_output_iterator<T>是一种输出迭代器(Output Iterator)，它将类型为 T 的对象输出到 otl_stream。其构造函数为 otl_output_iterator(otl_stream& s)。

otl_input_iterator<T, Distance>是一种输入迭代器(Input Iterator)，它从 otl_stream 中读出将类型为 T 的对象，另外一个模板参数 Distance 为 otl_input_iterator 的指针偏移类型。当流的末尾到达时，otl_input_iterator 会得到一个特殊的值即 past-the-end 迭代器。

otl_input_iterator 的构造函数分别为 otl_output_iterator(otl_stream& s)和 otl_output_iterator()，其中无参数的默认构造函数 otl_output_iterator()将创建一个 past-the-end 迭代器用以指示 otl_input_iterator 到达流尾。

以下是使用 STL 兼容迭代器的示例代码。

```
#include <iostream>
#include <vector>
#include <iterator>
#include <string>
#define OTL_ORA8 // Compile OTL 4.0/OCI8
#define OTL_STL // Turn on STL features
#define OTL_ANSI_CPP // Turn on ANSI C++ typecasts
#include <otlv4.h> // include the OTL 4.0 header file

using namespace std;

otl_connect db; // connect object

// row container class
```

```
class row{
public:
    int f1;
    string f2;

    // default constructor
    row(){f1=0;}

    // destructor
    ~row(){}

    // copy constructor
    row(const row& row)
    {
        f1=row.f1;
        f2=row.f2;
    }

    // assignment operator
    row& operator=(const row& row)
    {
        f1=row.f1;
        f2=row.f2;
        return *this;
    }
};

// redefined operator>> for reading row& from otl_stream
otl_stream& operator>>(otl_stream& s, row& row)
{
    s>>row.f1>>row.f2;
    return s;
}

// redefined operator<< for writing row& into otl_stream
otl_stream& operator<<(otl_stream& s, const row& row)
{
    s<<row.f1<<row.f2;
    return s;
}

// redefined operator<< writing row& into ostream
ostream& operator<<(ostream& s, const row& row)
{
    s<<"f1="<<row.f1<<", f2="<<row.f2;
    return s;
}

void insert()
// insert rows into table
```

```
{
  otl_stream o(50, // buffer size
                   "insert into test_tab values(:f1<int>,:f2<char[31]>)",
                       // SQL statement
                   db // connect object
                  );

  row r; // single row buffer
  vector<row> vo; // vector of rows

  // populate the vector
  for(int i=1;i<=100;++i){
      r.f1=i;
      r.f2="NameXXX";
      vo.push_back(r);
  }

  cout<<"vo.size="<<vo.size()<<endl;

  // insert vector into table
  copy(vo.begin(), vo.end(), otl_output_iterator<row>(o) );
}

void select()
{
  otl_stream i(50, // buffer size
                   "select * from test_tab where f1>=:f<int> and f1<=:f*2",
                       // SELECT statement
                   db // connect object
                  );
     // create select stream

  vector<row> v; // vector of rows

  // assigning :f = 8
  i<<8;

     // SELECT automatically executes when all input variables are
     // assigned. First portion of out rows is fetched to the buffer

  // copy all rows to be fetched into the vector
  copy(otl_input_iterator<row,ptrdiff_t>(i),
          otl_input_iterator<row,ptrdiff_t>(),
          back_inserter(v));

  cout<<"Size="<<v.size()<<endl;

  // send the vector to cout
  copy(v.begin(), v.end(), ostream_iterator<row>(cout, "\n"));
```

```
  // clean up the vector
  v.erase(v.begin(),v.end());

  i<<4; // assigning :f = 4
     // SELECT automatically executes when all input variables are
     // assigned. First portion of out rows is fetched to the buffer

  // copy all rows to be fetched to the vector
  copy(otl_input_iterator<row,ptrdiff_t>(i),
          otl_input_iterator<row,ptrdiff_t>(),
          back_inserter(v));

  cout<<"Size="<<v.size()<<endl;

  // send the vector to cout
  copy(v.begin(), v.end(), ostream_iterator<row>(cout, "\n"));

}

int main()
{
  otl_connect::otl_initialize(); // initialize OCI environment
  try{

    db.rlogon("scott/tiger"); // connect to Oracle

    otl_cursor::direct_exec
      (
        db,
        "drop table test_tab",
        otl_exception::disabled // disable OTL exceptions
      ); // drop table

    otl_cursor::direct_exec
      (
        db,
        "create table test_tab(f1 number, f2 varchar2(30))"
        );    // create table

    insert(); // insert records into table
    select(); // select records from table

  }

  catch(otl_exception& p){ // intercept OTL exceptions
    cerr<<p.msg<<endl; // print out error message
    cerr<<p.stm_text<<endl; // print out SQL that caused the error
    cerr<<p.var_info<<endl; // print out the variable that caused the error
  }
```

```
  db.logoff(); // disconnect from Oracle

  return 0;

}
```

输出结果：

f1=8, f2=Name8

f1=9, f2=Name9

f1=10, f2=Name10

f1=11, f2=Name11

f1=12, f2=Name12

f1=13, f2=Name13

f1=14, f2=Name14

f1=15, f2=Name15

f1=16, f2=Name16

f1=4, f2=Name4

f1=5, f2=Name5

f1=6, f2=Name6

f1=7, f2=Name7

f1=8, f2=Name8

8 资源池

8.1 连接缓冲池

对于 Oracle 数据库 API，OTL 的 otl_connect 类提供了 server_attach()、server_detached()、session_begin()、session_end()四个方法(见 4.2 小节)，它们可以联合使用以创建类似连接缓冲池机制。使用 session_begin()比直接使用 rlogon()快大约 50 到 100 倍。

以下是联合使用 otl_connect 的以上四个方法创建连接缓冲池机制的示例代码。

```
#include <iostream>
using namespace std;


#include <stdio.h>
#define OTL_ORA8 // Compile OTL 4.0/OCI8
```

```
#include <otlv4.h> // include the OTL 4.0 header file

otl_connect db; // connect object

void insert()
// insert rows into table
{
  otl_stream o(50, // buffer size
                   "insert into test_tab values(:f1<float>,:f2<char[31]>)",
                       // SQL statement
                   db // connect object
                  );
  char tmp[32];

  for(int i=1;i<=100;++i){
    sprintf(tmp,"Name%d",i);
    o<<(float)i<<tmp;
  }
}

void select()
{
  otl_stream i(50, // buffer size
                   "select * from test_tab where f1>=:f<int> and f1<=:f*2",
                       // SELECT statement
                   db // connect object
                  );
     // create select stream

  float f1;
  char f2[31];

  i<<4; // assigning :f = 4
```

```
    // SELECT automatically executes when all input variables are
    // assigned. First portion of output rows is fetched to the buffer

  while(!i.eof()){ // while not end-of-data
   i>>f1>>f2;
   cout<<"f1="<<f1<<", f2="<<f2<<endl;
  }

}

int main()
{
  otl_connect::otl_initialize(); // initialize OCI environment
  try{

   db.rlogon("scott/tiger"); // connect to Oracle

   otl_cursor::direct_exec
    (
     db,
     "drop table test_tab",
     otl_exception::disabled // disable OTL exceptions
    ); // drop table

   otl_cursor::direct_exec
    (
     db,
     "create table test_tab(f1 number, f2 varchar2(30))"
     );    // create table

   insert(); // insert records into table

   db.logoff(); // disconnect from Oracle
```

```
  db.server_attach(); // attach to the local Oracle server
                      // In order to connect to a remote server,
                      // a TNS alias needs to be specified

  for(int i=1;i<=100;++i){
      cout<<"Session begin ==> "<<i<<endl;
      db.session_begin("scott","tiger");
      // begin session; this function is much faster
      // than rlogon() and should be used (see the Oracle
      // manuals for more detail) in high-speed processing
      // systems, possibly with thousands of users.
      // this technique can be used instead of traditional
      // connection pooling.

      select(); // select records from table

      cout<<"Session end ==> "<<i<<endl;
      db.session_end(); // end session
  }

  db.server_detach(); // detach from the Oracle server
  }
  catch(otl_exception& p){ // intercept OTL exceptions
     cerr<<p.msg<<endl; // print out error message
     cerr<<p.stm_text<<endl; // print out SQL that caused the error
     cerr<<p.var_info<<endl; // print out the variable that caused the error
  }

  db.logoff(); // make sure that the program gets disconnected from Oracle

return 0;
```

```
}
```

输出结果：

```
Session begin ==> XXX
f1=4, f2=Name4
f1=5, f2=Name5
f1=6, f2=Name6
f1=7, f2=Name7
f1=8, f2=Name8
Session end ==> XXX
```

8.2 OTL 流缓冲池

流缓冲池是 OTL 的一个新机制，当 otl_stream 实例关闭时，otl_stream 变量实例将被保存到流缓冲池中，使得程序能够继续重用。otl_stream 的每次实例化将触发数据库后台对 OTL 流中 SQL 语句的重新解析，这是相对耗时的操作，而流缓冲池机制将减少这方面的耗时并简化编码技术。

缓冲池中的流可以是局部变量也可以是分配在堆上的动态变量。流之间的相似型和流缓冲区大小以及 SQL 语句文本有关，拥有相同缓冲区大小和 SQL 语句文本的流将保存在缓冲池中相同的桶中如图 8-1 所示。由于流缓冲池的底层使用 STL 的 map 和 vector 实现，因此使用时应该用”#define OTL_STL”向编译器指明。

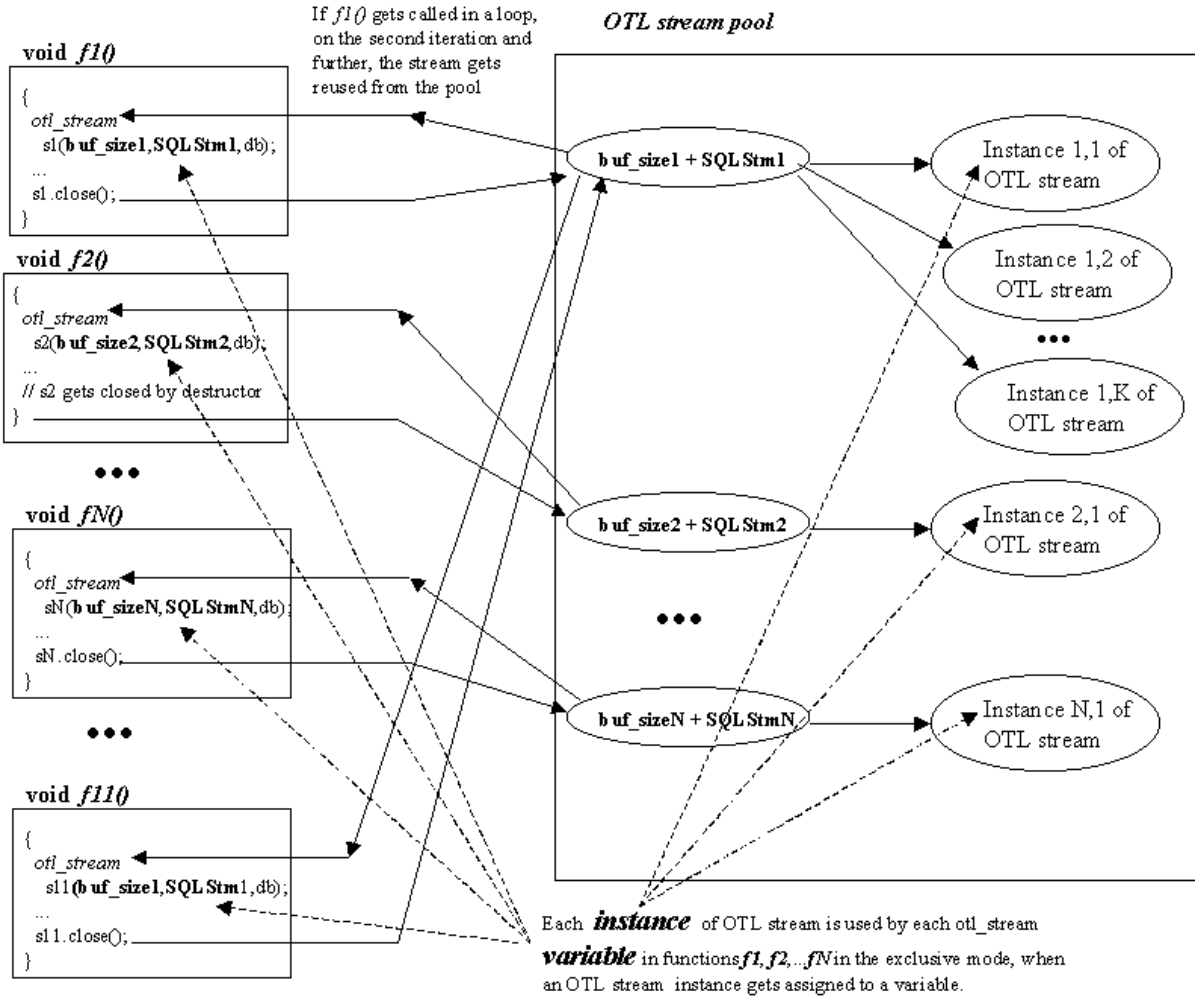

图 8-1 OTL 的流缓冲池

以下是采用流缓冲池机制的示例代码。

```
#include <iostream>
using namespace std;


#include <stdio.h>
// Uncomment the line below when OCI7 is used with OTL
// #define OTL_ORA7 // Compile OTL 4.0/OCI7
#define OTL_ORA8 // Compile OTL 4.0/OCI8
#define OTL_STL // turn on OTL in the STL compliance mode
#define OTL_STREAM_POOLING_ON
 // turn on OTL stream pooling.
 // #define OTL_STREAM_POOLING_ON line
 // can be commented out the number of iterations in
```

```
 // the select() loop can be increased, and the difference
 // in performace with and without OTL_STREAM_POOLING_ON can
 // be benchmarked. The difference should grow with the overall
 // number of streams to be used in one program.

#include <otlv4.h> // include the OTL 4.0 header file

otl_connect db; // connect object

void insert()
// insert rows into table
{
 otl_stream o(50, // buffer size
              "insert into test_tab values(:f1<int>,:f2<char[31]>)",
                // SQL statement
              db // connect object
             );
 char tmp[32];

 for(int i=1;i<=100;++i){
   sprintf(tmp,"Name%d",i);
   o<<i<<tmp;
 }
#ifdef OTL_STREAM_POOLING_ON
 o.close(false); // do not save the stream in the stream pool.
                 // in other words, destroy it on the spot, since
                 // the stream is not going to be reused later.
#else
 o.close();
#endif
}

void select()
```

```
{ // when this function is called in a loop,
  // on the second iteration of the loop the streams i1, i2 will
  // will get the instances of the OTL stream from the stream
  // pool, "fast reopen", so to speak.

 otl_stream i1(50, // buffer size
               "select * from test_tab where f1>=:f11<int> and f1<=:f12<int>*2",
                 // SELECT statement
              db // connect object
             );
   // create select stream

 otl_stream i2(33, // buffer size
               "select f1,f2 from test_tab where f1>=:f11<int> and f1<=:f12<int>*2",
                 // SELECT statement
              db // connect object
             );
   // create select stream

// i1 and i2 are NOT similar, because their buffer sizes as well
// as SQL statements are not equal. It will generate two entry points in the
// OTL stream pool.

 int f1;
 char f2[31];

 i1<<2<<2; // assigning :f11 = 2, :f12 = 2
   // SELECT automatically executes when all input variables are
   // assigned. First portion of output rows is fetched to the buffer

 while(!i1.eof()){ // while not end-of-data
  i1>>f1>>f2;
  cout<<"I1==> f1="<<f1<<", f2="<<f2<<endl;
```

```
 }

 i2<<3<<3; // assigning :f11 = 2, :f12 = 2
    // SELECT automatically executes when all input variables are
    // assigned. First portion of output rows is fetched to the buffer

 while(!i2.eof()){ // while not end-of-data
   i2>>f1>>f2;
   cout<<"I2==> f1="<<f1<<", f2="<<f2<<endl;
 }

} // destructors of i1, i2 will call the close()
   // function for both of the streams and the OTL stream
   // instances will be placed in the stream pool.

int main()
{
 otl_connect::otl_initialize(); // initialize the environment
 try{

   db.rlogon("scott/tiger"); // connect to the database
#ifdef OTL_STREAM_POOLING_ON
   db.set_stream_pool_size(2);
    // set the maximum stream pool size and actually initializes
    // the stream pool.
    // if this function is not called, the stream pool
    // gets initialized anyway, with the default size of 32 entries.
#endif

   otl_cursor::direct_exec
    (
     db,
     "drop table test_tab",
```

```
    otl_exception::disabled // disable OTL exceptions
   ); // drop table

  otl_cursor::direct_exec
   (
    db,
    "create table test_tab(f1 int, f2 varchar(30))"
    );   // create table

  insert(); // insert records into table
  for(int i=1;i<=10; ++i){
    cout<<"==================> Iteration: "<<i<<endl;
    select(); // select records from table
  }
 }

 catch(otl_exception& p){ // intercept OTL exceptions
  cerr<<p.msg<<endl; // print out error message
  cerr<<p.stm_text<<endl; // print out SQL that caused the error
  cerr<<p.var_info<<endl; // print out the variable that caused the error
 }

 db.logoff(); // disconnect from the database

 return 0;

}
```

输出结果：

```
==================> Iteration: 1
I1==> f1=2, f2=Name2
I1==> f1=3, f2=Name3
I1==> f1=4, f2=Name4
I2==> f1=3, f2=Name3
I2==> f1=4, f2=Name4
I2==> f1=5, f2=Name5
I2==> f1=6, f2=Name6
```

```
==================> Iteration: 2
I1==> f1=2, f2=Name2
I1==> f1=3, f2=Name3
I1==> f1=4, f2=Name4
I2==> f1=3, f2=Name3
I2==> f1=4, f2=Name4
I2==> f1=5, f2=Name5
I2==> f1=6, f2=Name6
==================> Iteration: 3
I1==> f1=2, f2=Name2
I1==> f1=3, f2=Name3
I1==> f1=4, f2=Name4
I2==> f1=3, f2=Name3
I2==> f1=4, f2=Name4
I2==> f1=5, f2=Name5
I2==> f1=6, f2=Name6
==================> Iteration: 4
I1==> f1=2, f2=Name2
I1==> f1=3, f2=Name3
I1==> f1=4, f2=Name4
I2==> f1=3, f2=Name3
I2==> f1=4, f2=Name4
I2==> f1=5, f2=Name5
I2==> f1=6, f2=Name6
==================> Iteration: 5
I1==> f1=2, f2=Name2
I1==> f1=3, f2=Name3
I1==> f1=4, f2=Name4
I2==> f1=3, f2=Name3
I2==> f1=4, f2=Name4
I2==> f1=5, f2=Name5
I2==> f1=6, f2=Name6
==================> Iteration: 6
I1==> f1=2, f2=Name2
I1==> f1=3, f2=Name3
I1==> f1=4, f2=Name4
I2==> f1=3, f2=Name3
I2==> f1=4, f2=Name4
I2==> f1=5, f2=Name5
I2==> f1=6, f2=Name6
==================> Iteration: 7
I1==> f1=2, f2=Name2
I1==> f1=3, f2=Name3
I1==> f1=4, f2=Name4
I2==> f1=3, f2=Name3
I2==> f1=4, f2=Name4
I2==> f1=5, f2=Name5
I2==> f1=6, f2=Name6
==================> Iteration: 8
I1==> f1=2, f2=Name2
```

```
I1==> f1=3, f2=Name3
I1==> f1=4, f2=Name4
I2==> f1=3, f2=Name3
I2==> f1=4, f2=Name4
I2==> f1=5, f2=Name5
I2==> f1=6, f2=Name6
==================> Iteration: 9
I1==> f1=2, f2=Name2
I1==> f1=3, f2=Name3
I1==> f1=4, f2=Name4
I2==> f1=3, f2=Name3
I2==> f1=4, f2=Name4
I2==> f1=5, f2=Name5
I2==> f1=6, f2=Name6
==================> Iteration: 10
I1==> f1=2, f2=Name2
I1==> f1=3, f2=Name3
I1==> f1=4, f2=Name4
I2==> f1=3, f2=Name3
I2==> f1=4, f2=Name4
I2==> f1=5, f2=Name5
I2==> f1=6, f2=Name6
```

9 操作大型对象

OTL 提供了 otl_long string、otl_long_unicode_string 以及 otl_lob_stream 三个类操作大型对象。其中 otl_long string、otl_long_unicode_string 用来存储大型对象，otl_lob_stream 用来读写大型对象。

9.1 大型对象的存储

9.1.1 otl_long_string

OTL 提供了类 otl_long_string 来存放 ANSI 字符集编码的大型对象，其定义如下。

```
class otl_long_string{
public:
    unsigned char* v;
    otl_long_string(
          const int buffer_size=32760,
          const int input_length=0
    );
    otl_long_string(const void* external_buffer,
          const int buffer_size,
          const int input_length=0
   );
    void set_len(const int len=0);
    void set_last_piece(const bool last_piece=false);
    int len(void);
```

```
    unsigned char& operator[](int ndx);

  otl_long_string& operator=(const otl_long_string&);
  otl_long_string(const otl_long_string&);
}; // end of otl_long_string
```

otl_long_string 的成员变量 v 存放大型对象的缓存起始位置。构造函数中的参数说明如下：

- buffer_size 参数指明存放大型对象的缓存大小，默认为 32760，可以通过 otl_connect 的 set_max_long_size()方法来改变默认的大小值 。
- input_length 参数则指明实际输入的大小，如果该参数被设定则 set_len()成员方法就没有必要使用了。
- 另外，如果使用参数 external_buffer 则 otl_long_string 不再为大型对象实际分配存储空间，而是直接使用用户传入的以 external_buffer 为起始地址的缓存。

9.1.2 otl_long_unicode_string

OTL 提供了类 otl_long_string 来存放 UNICODE 字符集编码的大型对象，其定义如下。

```
class otl_long_unicode_string: public otl_long_string{
public:
  otl_long_unicode_string(
        const int buffer_size=32760,
        const int input_leng
  );

  otl_long_unicode_string(
        const void* external_buffer,
        const int buffer_size,
        const int input_length=0
   );

  void set_len(const int len=0);
   int len(void);
  unsigned short& operator[](int ndx);
}; // end of otl_long_unicode_string
```

otl_long_unicode_string 的成员变量、方法以及构造函数和父类相同，但是注意缓冲区大小为 UNICODE 字符数量而不是字节数。

9.2 大型对象的读写

大型对象的读写通过类 otl_lob_stream 实现，其定义如下。

```
class otl_lob_stream {
public:
```

```
    void set_len(const int alen);
    otl_lob_stream& operator<<(const std::string& s);
     otl_lob_stream& operator<<(const ACE_TString& s);

    otl_lob_stream& operator>>(std::string& s);
     otl_lob_stream& operator>>(ACE_TString& s);

     void setStringBuffer(const int chunk_size);
     otl_lob_stream& operator<<(const otl_long_string& s);
    otl_lob_stream& operator<<(const otl_long_unicode_string& s);
    otl_lob_stream& operator>>(otl_long_string& s);

     otl_lob_stream& operator>>(otl_long_unicode_string& s);

    int len(void);
    int eof(void);
    void close(void);
    bool is_initialized(void);
}; // end of otl_lob_stream
```

otl_lob_stream 重载了<<和>>运算符来操作存放在 std::string、otl_long_string、otl_long_unicode_string 以及 ACE_TString 中的大型对象。

以下是操作大型对象的示例代码。包括 INSERT、UPDATE、SELECT 操作。注意在使用 INSERT 和 UPDATE 时流缓冲区的大小必须为 1，另外必须将 otl_stream 的 auto_commit 标志设置为 false。初始化 otl_lob_stream 是通过将 otl_lob_stream 对象作为 otl_stream 的<<操作符参数来实现的。

```
#include <iostream>
using namespace std;

#include <stdio.h>
#define OTL_ORA8 // Compile OTL 4.0/OCI8
#include <otlv4.h> // include the OTL 4.0 header file

otl_connect db; // connect object

void insert()
// insert rows into table
{otl_long_string f2(60000); // define long string variable
 otl_stream o(1, // buffer size has to be set to 1 for operations with LOBs
                  "insert into test_tab values(:f1<int>,empty_clob()) "
                  "returning f2 into :f2<clob> ", // SQL statement
                  db // connect object
                 );
 o.set_commit(0); // setting stream "auto-commit" to "off". It is required
                   // when LOB stream mode is used.
```

```
 otl_lob_stream lob; // LOB stream for reading/writing unlimited number
                     // of bytes regardless of the buffer size.

 for(int i=1;i<=20;++i){
     for(int j=0;j<50000;++j)
         f2[j]='*';
     f2[50000]='?';
     f2.set_len(50001);

     o<<i;

     o<<lob; // Initialize otl_lob_stream by writing it
             // into otl_stream. Weird, isn't it?

     lob.set_len(50001+23123); // setting the total    size of
                               // the CLOB to be written.
     // It is required for compatibility
     // with earlier releases of OCI8: OCI8.0.3, OCI8.0.4.

     lob<<f2; // writing first chunk of the CLOB into lob

     f2[23122]='?';
     f2.set_len(23123); // setting the size of the second chunk

     lob<<f2; // writing the second chunk of the CLOB into lob
     lob.close(); // closing the otl_lob_stream
 }

 db.commit(); // committing transaction.
}
void update()
// insert rows in table
{
 otl_long_string f2(6200); // define long string variable

 otl_stream o(1, // buffer size has to be set to 1 for operations with LOBs
                 "update test_tab "
                 "      set f2=empty_clob() "
                 "where f1=:f1<int> "
                 "returning f2 into :f2<clob> ",
                     // SQL statement
                 db // connect object
                );

  otl_lob_stream lob;

  o.set_commit(0); // setting stream "auto-commit" to "off".


 for(int j=0;j<6000;++j){
```

```
      f2[j]='#';
  }

  f2[6000]='?';
  f2.set_len(6001);

  o<<5;
  o<<lob; // Initialize otl_lob_stream by writing it
              // into otl_stream.

  lob.set_len(6001*4); // setting the total size of of the CLOB to be written
  for(int i=1;i<=4;++i)
      lob<<f2; // writing chunks of the CLOB into the otl_lob_stream

  lob.close(); // closing the otl_lob_stream

  db.commit(); // committing transaction

}

void select()
{
  otl_long_string f2(20000); // define long string variable

  otl_stream i(10, // buffer size. To read CLOBs, it can be set to a size greater than 1
                   "select * from test_tab where f1>=:f<int> and f1<=:f*2",
                       // SELECT statement
                   db // connect object
                  );
     // create select stream

  float f1;
  otl_lob_stream lob; // Stream for reading CLOB

  i<<4; // assigning :f = 4
     // SELECT automatically executes when all input variables are
     // assigned. First portion of output rows is fetched to the buffer

  while(!i.eof()){ // while not end-of-data
      i>>f1;
      cout<<"f1="<<f1<<endl;
      i>>lob; // initializing CLOB stream by reading the CLOB reference
                // into the otl_lob_stream from the otl_stream.
      int n=0;
      while(!lob.eof()){ // read while not "end-of-file" -- end of CLOB
          ++n;
          lob>>f2; // reading a chunk of CLOB
          cout<<"      chunk #"<<n;
          cout<<", f2="<<f2[0]<<f2[f2.len()-1]<<", len="<<f2.len()<<endl;
      }
```

```
      lob.close(); // closing the otl_lob_stream. This step may be skipped.
  }
}

int main()
{
  otl_connect::otl_initialize(); // initialize OCI environment
  try{

    db.rlogon("scott/tiger"); // connect to Oracle

    otl_cursor::direct_exec
      (
       db,
       "drop table test_tab",
       otl_exception::disabled // disable OTL exceptions
      ); // drop table

    otl_cursor::direct_exec
      (
       db,
       "create table test_tab(f1 number, f2 clob)"
       );    // create table

    insert(); // insert records into table
    update(); // update records in table
    select(); // select records from table

  }

  catch(otl_exception& p){ // intercept OTL exceptions
    cerr<<p.msg<<endl; // print out error message
    cerr<<p.stm_text<<endl; // print out SQL that caused the error
    cerr<<p.var_info<<endl; // print out the variable that caused the error
  }

  db.logoff(); // disconnect from Oracle

  return 0;

}
```

输出结果：

```
f1=4
     chunk #1, f2=**, len=20000
     chunk #2, f2=**, len=20000
     chunk #3, f2=**, len=20000
     chunk #4, f2=*?, len=13124
f1=5
     chunk #1, f2=##, len=20000
```

```
    chunk #2, f2=#?, len=4004
f1=6
    chunk #1, f2=**, len=20000
    chunk #2, f2=**, len=20000
    chunk #3, f2=**, len=20000
    chunk #4, f2=*?, len=13124
f1=7
    chunk #1, f2=**, len=20000
    chunk #2, f2=**, len=20000
    chunk #3, f2=**, len=20000
    chunk #4, f2=*?, len=13124
f1=8
    chunk #1, f2=**, len=20000
    chunk #2, f2=**, len=20000
    chunk #3, f2=**, len=20000
    chunk #4, f2=*?, len=13124
```

10 国际化

OTL 的国际化支持主要是通过支持编码类型为 UNICODE 或 UTF8 的字符串操作实现。

10.1 使用 UNICODE 字符串

可以通过宏”#define OTL_UNICODE”指示 OTL 内部使用 UNICODE 字符串。以下是使用 UNICODE 字符串的示例代码。

示例代码中使用 unsigned short 类型数组存放 UNICODE 字符串，由于 otl_stream 的操作符<<并不支持 unsigned short*类型，因此在变量绑定使用<<操作符时，将其强制转换成 unsigned char*类型进行操作。与此类似，由于 otl_stream 的操作符>>并不支持 unsigned short*类型，读取结果使用>>操作符时也是将其强制转换成 unsigned char*类型进行操作。

```
#include <iostream>
using namespace std;


#include <stdio.h>
#define OTL_ORA9I // Compile OTL 4.0/OCI9i
#define OTL_UNICODE // Enable Unicode OTL for OCI9i
#include <otlv4.h> // include the OTL 4.0 header file


otl_connect db; // connect object
```

```
void insert()
// insert rows into table
{
 otl_stream o(50, // buffer size
              "insert into test_tab values(:f1<float>,:f2<char[31]>)",
                // SQL statement
              db // connect object
             );
 char tmp[32];
 unsigned short tmp2[32]; // Null terminated Unicode character array.

 for(int i=1;i<=100;++i){
   sprintf(tmp,"Name%d",i);
   unsigned short* c2=tmp2;
   char* c1=tmp;
// Unicode's first 128 characters are ASCII (0..127), so
// all is needed for converting ASCII into Unicode is as follows:
   while(*c1){
    *c2=(unsigned char)*c1;
    ++c1; ++c2;
   }
   *c2=0; // target Unicode string is null terminated,
              // only the null terminator is a two-byte character,
              // not one-byte
   o<<(float)i;
   o<<(unsigned char*)tmp2;
    // overloaded operator<<(const unsigned char*) in the case of Unicode
    // OTL accepts a pointer to a Unicode character array.
    // operator<<(const unsigned short*) wasn't overloaded
    // in order to avoid ambiguity in C++ type casting.
 }
```

```
}

void select()
{
 otl_stream i(50, // buffer size
              "select * from test_tab where f1>=:f<int> and f1<=:f*2", // SELECT statement
              db // connect object
             );
   // create select stream

 float f1;
 unsigned short f2[32];

 i<<8; // assigning :f = 8
   // SELECT automatically executes when all input variables are
   // assigned. First portion of output rows is fetched to the buffer

 while(!i.eof()){ // while not end-of-data
      i>>f1;
      i>>(unsigned char*)f2;
      // overloaded operator>>(unsigned char*) in the case of Unicode
      // OTL accepts a pointer to a Unicode chracter array.
      // operator>>(unsigned short*) wasn't overloaded
      // in order to avoid ambiguity in C++ type casting.
      cout<<"f1="<<f1<<", f2=";
       // Unicode's first 128 characters are ASCII, so in order
       // to convert Unicode back to ASCII all is needed is
       // as follows:
       for(int j=0;f2[j]!=0;++j){
            cout<<(char)f2[j];
       }
       cout<<endl;
}
```

```
 i<<4; // assigning :f = 4
    // SELECT automatically executes when all input variables are
    // assigned. First portion of output rows is fetched to the buffer

 while(!i.eof()){ // while not end-of-data
   i>>f1>>(unsigned char*)f2;
   cout<<"f1="<<f1<<", f2=";
    for(int j=0;f2[j]!=0;++j){
      cout<<(char)f2[j];
    }
    cout<<endl;
 }

}

int main()
{
 otl_connect::otl_initialize(); // initialize OCI environment
 try{

   db.rlogon("scott/tiger"); // connect to Oracle

   otl_cursor::direct_exec
    (
     db,
     "drop table test_tab",
     otl_exception::disabled // disable OTL exceptions
    ); // drop table

   otl_cursor::direct_exec
    (
     db,
```

```
    "create table test_tab(f1 number, f2 varchar2(30))"
    );   // create table

  insert(); // insert records into table
  select(); // select records from table

}

catch(otl_exception& p){ // intercept OTL exceptions
  cerr<<p.msg<<endl; // print out error message
  cerr<<p.stm_text<<endl; // print out SQL that caused the error
  cerr<<p.var_info<<endl; // print out the variable that caused the error
}

db.logoff(); // disconnect from Oracle

return 0;

}
```

输出结果：

```
f1=8, f2=Name8
f1=9, f2=Name9
f1=10, f2=Name10
f1=11, f2=Name11
f1=12, f2=Name12
f1=13, f2=Name13
f1=14, f2=Name14
f1=15, f2=Name15
f1=16, f2=Name16
f1=4, f2=Name4
f1=5, f2=Name5
f1=6, f2=Name6
f1=7, f2=Name7
f1=8, f2=Name8
```

10.2 使用 UTF8 字符串

可以通过宏”#define OTL_ORA_UTF8”指示 OTL 内部使用 UTF8 字符串。以下是使用 UTF 字符串的示例代码。

示例代码中使用 unsigned char 类型数组存放 UTF8 字符串，在使用 otl_stream 的操作符<<进行变量绑定时，通过转型并使用 copy()函数将其转成 char*类型进行操作。另外需要注意到环境变量的设定 NLS_LANG=.AL32UTF8。

```
#include <iostream>
using namespace std;

#include <stdio.h>
#define OTL_ORA9I // Compile OTL 4.0/OCI9i
#define OTL_ORA_UTF8 // Enable UTF8 OTL for OCI9i
#include <otlv4.h> // include the OTL 4.0 header file

otl_connect db; // connect object

// Sample UTF8 based string
unsigned char utf8_sample[]=
{0x61,0x62,0x63,0xd0,0x9e,0xd0,0x9b,0xd0,
 0xac,0xd0,0x93,0xd0,0x90,0x0};

void insert()
// insert rows into table
{
 otl_stream o(50, // buffer size
              "insert into test_tab values(:f1<int>,:f2<char[31]>)",
                // SQL statement
              db // connect object
             );

 unsigned char tmp[31];

 for(int i=1;i<=100;++i){
     strcpy(reinterpret_cast<char*>(tmp),reinterpret_cast<const char*>(utf8_sample));
    o<<i;
    o<<tmp;
```

```
 }

}

void select()
{
 otl_stream i(50, // buffer size
              "select * from test_tab where f1>=:f<int> and f1<=:f*2",
                // SELECT statement
              db // connect object
             );
   // create select stream

 int f1;
 unsigned char f2[31];

 i<<8; // assigning :f = 8
   // SELECT automatically executes when all input variables are
   // assigned. First portion of output rows is fetched to the buffer

 while(!i.eof()){ // while not end-of-data
    i>>f1;
    i>>f2;
    cout<<"f1="<<f1<<", f2=";
    for(int j=0;f2[j]!=0;++j)
        printf("%2x ", f2[j]);
    cout<<endl;
 }

}

int main()
{
```

```
 putenv(const_cast<char*>("NLS_LANG=.AL32UTF8"));
 // set your Oracle Client NLS_LANG
 // if its default was set to something else
otl_connect::otl_initialize(); // initialize OCI environment
try{

  db.rlogon("scott/tiger"); // connect to Oracle

  otl_cursor::direct_exec
   (
    db,
    "drop table test_tab",
    otl_exception::disabled // disable OTL exceptions
   ); // drop table

  otl_cursor::direct_exec
   (
    db,
    "create table test_tab(f1 number, f2 varchar2(30))"
    );   // create table

  insert(); // insert records into table
  select(); // select records from table

}

catch(otl_exception& p){ // intercept OTL exceptions
  cerr<<p.msg<<endl; // print out error message
  cerr<<p.stm_text<<endl; // print out SQL that caused the error
  cerr<<p.var_info<<endl; // print out the variable that caused the error
}

db.logoff(); // disconnect from Oracle
```

```
 return 0;
}
```

输出结果：

```
f1=8, f2=61 62 63 d0 9e d0 9b d0 ac d0 93 d0 90
f1=9, f2=61 62 63 d0 9e d0 9b d0 ac d0 93 d0 90
f1=10, f2=61 62 63 d0 9e d0 9b d0 ac d0 93 d0 90
f1=11, f2=61 62 63 d0 9e d0 9b d0 ac d0 93 d0 90
f1=12, f2=61 62 63 d0 9e d0 9b d0 ac d0 93 d0 90
f1=13, f2=61 62 63 d0 9e d0 9b d0 ac d0 93 d0 90
f1=14, f2=61 62 63 d0 9e d0 9b d0 ac d0 93 d0 90
f1=15, f2=61 62 63 d0 9e d0 9b d0 ac d0 93 d0 90
f1=16, f2=61 62 63 d0 9e d0 9b d0 ac d0 93 d0 90
```

11 Reference Cursor 流

OTL 为 OTL/OCI8/8i/9i/10g 提供了类 otl_refcur_stream，它可以从 reference cursor 类型的绑定变量中读取结果行，其定义如下所示。

```
class otl_refcur_stream {
public:
 void set_column_type(const int column_ndx,
                      const int col_type,
                      const int col_size=0);
 void set_all_column_types(const unsigned mask=0);

 void rewind(void);
 int is_null(void);
 int eof(void);
 void close(void);

 otl_column_desc* describe_select(int& desc_len);

  otl_var_desc* describe_out_vars(int& desc_len);
  otl_var_desc* describe_next_out_var(void);

 otl_refcur_stream& operator>>(char& c);
 otl_refcur_stream& operator>>(unsigned char& c);
 otl_refcur_stream& operator>>(char* s);
 otl_refcur_stream& operator>>(unsigned char* s);
 otl_refcur_stream& operator>>(int& n);
 otl_refcur_stream& operator>>(unsigned& u);
 otl_refcur_stream& operator>>(short& sh);
 otl_refcur_stream& operator>>(long int& l);
 otl_refcur_stream& operator>>(float& f);
 otl_refcur_stream& operator>>(double& d);

  //for large whole number
```

```
 otl_refcur_stream& operator>>(OTL_BIGINT& n);

 //for LOBs
otl_refcur_stream& operator>>(otl_long_string& s);
otl_refcur_stream& operator>>(otl_datetime& dt);
otl_refcur_stream& operator>>(otl_lob_stream& lob);
otl_refcur_stream& operator>>(std::string& s);

 //for UNICODE
otl_stream& operator>>(unsigned char* s);
otl_stream& operator>>(otl_long_unicode_string& s);

}; // end of otl_refcur_stream
```

otl_refcur_stream 的初始化通过将 otl_refcur_stream 对象作为 otl_stream 的>>操作符参数来实现。以下是其基本使用的示例代码。

```
#include <iostream>
using namespace std;

#include <stdio.h>
#define OTL_ORA8 // Compile OTL 4.0/OCI8
//#define OTL_ORA8I // Compile OTL 4.0/OCI8i
//#define OTL_ORA9I // Compile OTL 4.0/OCI9i
#include <otlv4.h> // include the OTL 4.0 header file


otl_connect db; // connect object


void insert()
// insert rows into table
{
 otl_stream o(50, // buffer size
              "insert into test_tab values(:f1<float>,:f2<char[31]>)", // SQL statement
              db // connect object
             );
 char tmp[32];


 for(int i=1;i<=100;++i){
   sprintf(tmp,"Name%d",i);
   o<<(float)i<<tmp;
```

```
 }
}

void select()
{
  // :cur is a bind variable name, refcur -- its type,
 // out -- output parameter, 50 -- the buffer size when this
 // reference cursor will be attached to otl_refcur_stream
 otl_stream i(1, // buffer size
                 "begin "
                 " open :cur<refcur,out[50]> for "
                 "   select * "
                 "   from test_tab "
                 "   where f1>=:f<int,in> and f1<=:f*2; "
                 "end;", // PL/SQL block returns a referenced cursor
                 db // connect object
             );
   // create select stream with referenced cursor

  i.set_commit(0); // set stream "auto-commit" to OFF.

 float f1;
 char f2[31];
 otl_refcur_stream s; // reference cursor stream for reading rows.

 i<<8; // assigning :f = 8
 i>>s; // initializing the refrence cursor stream with the output
         // reference cursor.

 while(!s.eof()){ // while not end-of-data
     s>>f1>>f2;
     cout<<"f1="<<f1<<", f2="<<f2<<endl;
}
```

```
 s.close(); // closing the reference cursor

 i<<4; // assigning :f = 4
 i>>s;

 while(!s.eof()){ // while not end-of-data
      s>>f1>>f2;
      cout<<"f1="<<f1<<", f2="<<f2<<endl;
 }
 // there is no need to explicitly calls s.close() since s's destructor
 // will take care of closing the stream
}

int main()
{
 otl_connect::otl_initialize(); // initialize OCI environment
 try{

  db.rlogon("scott/tiger"); // connect to Oracle

  otl_cursor::direct_exec
   (
    db,
    "drop table test_tab",
    otl_exception::disabled // disable OTL exceptions
   ); // drop table

  otl_cursor::direct_exec
   (
    db,
    "create table test_tab(f1 number, f2 varchjar2(30))"
    );   // create table
```

```
  insert(); // insert records into table
  select(); // select records from table

 }
 catch(otl_exception& p){ // intercept OTL exceptions
  cerr<<p.msg<<endl; // print out error message
  cerr<<p.stm_text<<endl; // print out SQL that caused the error
  cerr<<p.var_info<<endl; // print out the variable that caused the error
 }

 db.logoff(); // disconnect from Oracle

 return 0;

}
```

输出结果：

```
f1=8, f2=Name8
f1=9, f2=Name9
f1=10, f2=Name10
f1=11, f2=Name11
f1=12, f2=Name12
f1=13, f2=Name13
f1=14, f2=Name14
f1=15, f2=Name15
f1=16, f2=Name16
f1=4, f2=Name4
f1=5, f2=Name5
f1=6, f2=Name6
f1=7, f2=Name7
f1=8, f2=Name8
```

12 杂项

12.1 使用 otl_nocommit_stream 避免 SQL 执行成功后立刻提交事务

otl_stream 提供了方法 set_commit()方法(见 4.1 小节)设置 auto_commit 标志。该标志控制 SQL 执行成功后，是否立刻提交当前事务。auto_commit 默认为 true，即提交事务。可以通过设置 auto_commit 标志值为 false 避免这种情况。

otl_stream 的派生类 otl_nocimmit_stream 提供了 SQL 执行成功后不立刻提交事务的

功能，该类通过将 auto_commit 标记默认设置为 true，以简化实现该功能时的代码编写。

以下是使用 otl_nocommit_stream 类避免 SQL 执行成功后立刻提交事务的示例代码。

```
#include <iostream>
using namespace std;

#include <stdio.h>
#define OTL_ORA8 // Compile OTL 4.0/OCI8
#include <otlv4.h> // include the OTL 4.0 header file

otl_connect db; // connect object

void insert()
// insert rows into table
{
 otl_nocommit_stream o
      (50, // buffer size
       "insert into test_tab values(:f1<float>,:f2<char[31]>)", // SQL statement
       db // connect object
      );
 char tmp[32];

 for(int i=1;i<=100;++i){
  sprintf(tmp,"Name%d",i);
  o<<(float)i<<tmp;
 }

 o.flush();
 db.commit();

}

void select()
{
 otl_stream i(50, // buffer size
                  "select * from test_tab where f1>=:f<int> and f1<=:f*2",
                      // SELECT statement
                  db // connect object
                 );
    // create select stream

 float f1;
 char f2[31];

 i<<8; // assigning :f = 8
    // SELECT automatically executes when all input variables are
    // assigned. First portion of output rows is fetched to the buffer

 while(!i.eof()){ // while not end-of-data
  i>>f1>>f2;
```

```
    cout<<"f1="<<f1<<", f2="<<f2<<endl;
  }

  i<<4; // assigning :f = 4
      // SELECT automatically executes when all input variables are
      // assigned. First portion of output rows is fetched to the buffer

  while(!i.eof()){ // while not end-of-data
    i>>f1>>f2;
    cout<<"f1="<<f1<<", f2="<<f2<<endl;
  }

}

int main()
{
  otl_connect::otl_initialize(); // initialize OCI environment
  try{

    db.rlogon("scott/tiger"); // connect to Oracle

    otl_cursor::direct_exec
      (
        db,
        "drop table test_tab",
        otl_exception::disabled // disable OTL exceptions
      ); // drop table

    otl_cursor::direct_exec
      (
        db,
        "create table test_tab(f1 number, f2 varchar2(30))"
        );    // create table

    insert(); // insert records into table
    select(); // select records from table

  }

  catch(otl_exception& p){ // intercept OTL exceptions
    cerr<<p.msg<<endl; // print out error message
    cerr<<p.stm_text<<endl; // print out SQL that caused the error
    cerr<<p.var_info<<endl; // print out the variable that caused the error
  }

  db.logoff(); // disconnect from Oracle

  return 0;

}
```

输出结果：

```
f1=8, f2=Name8
f1=9, f2=Name9
f1=10, f2=Name10
f1=11, f2=Name11
f1=12, f2=Name12
f1=13, f2=Name13
f1=14, f2=Name14
f1=15, f2=Name15
f1=16, f2=Name16
f1=4, f2=Name4
f1=5, f2=Name5
f1=6, f2=Name6
f1=7, f2=Name7
f1=8, f2=Name8
```

12.2 SELECT 中的数据类型映射覆写

otl_stream 在执行 SELECT 语句返回结果列时，具有如下表 12-1 所示的数据库数据类型(Database datatype)到默认数据类型(Default datatype)的映射。

某些情况下往往需要对这种默认映射进行覆写，即将数据库数据类型映射为非默认的数据类型。例如当读取超过 16 位的数字时，将其映射为字符串类型往往更方便。为此，otl_stream 提供了 set_colunm_type()方法(参见 4.1 小节)进行数据类型覆写。

12-1 otl_stream 中的数据类型映射覆写

Database datatype	Default datatype	Datatype override
NUMBER (Oracle)	otl_var_double	otl_var_char, otl_var_int, otl_var_float, otl_var_short, otl_var_unsigned_int
NUMERIC, FLOAT, REAL, MONEY, DECIMAL (MS SQL Server, Sybase, DB2)	otl_var_double	otl_var_char, otl_var_int, otl_var_float, otl_var_short, otl_var_unsigned_int, otl_var_long_int
INT (MS SQL Server, Sybase, DB2)	otl_var_int	otl_var_char, otl_var_double, otl_var_float, otl_var_short, otl_var_unsigned_int, otl_var_long_int
SMALLINT, TINYINT (MS SQL Server, Sybase, DB2)	otl_var_short	otl_var_char, otl_var_int, otl_var_float, otl_var_double, otl_var_unsigned_int, otl_var_long_int
DATE (Oracle), DATETIME (MS SQL Server, Sybase)	otl_timestamp	otl_var_char
LONG (Oracle)	otl_var_varchar_long	otl_var_char (<=32000 bytes)

TEXT (MS SQL Server, Sybase)	otl_var_varchar_long	otl_var_char(<= max. size of varchar, e.g. <=8000 in MS SQL 7.0)

以下是 SELECT 中的数据类型覆写的示例代码。

```
#include <iostream>
using namespace std;

#include <stdio.h>
#define OTL_ORA8 // Compile OTL 4.0/OCI8
#include <otlv4.h> // include the OTL 4.0 header file


otl_connect db; // connect object


void insert()
// insert rows into table
{
 otl_stream o(50, // buffer size
              "insert into test_tab values(12345678900000000+:f1<int>,:f2<char[31]>)",
                // SQL statement
              db // connect object
             );
 char tmp[32];

 for(int i=1;i<=100;++i){
   sprintf(tmp,"Name%d",i);
   o<<i<<tmp;
 }
}

void select()
{
 otl_stream i;

 i.set_column_type(1,otl_var_char,40); // use a string(40) instead of default double
```

```
i.open(50, // buffer size
       "select * from test_tab "
       "where f1>=12345678900000000+:f<int> "
       "   and f1<=12345678900000000+:f*2",
         // SELECT statement
       db // connect object
      );
  // create select stream

char f1[40];
char f2[31];

i<<8; // assigning :f = 8
  // SELECT automatically executes when all input variables are
  // assigned. First portion of output rows is fetched to the buffer

while(!i.eof()){ // while not end-of-data
 i>>f1>>f2;
 cout<<"f1="<<f1<<", f2="<<f2<<endl;
}

i<<4; // assigning :f = 4
  // SELECT automatically executes when all input variables are
  // assigned. First portion of output rows is fetched to the buffer

while(!i.eof()){ // while not end-of-data
 i>>f1>>f2;
 cout<<"f1="<<f1<<", f2="<<f2<<endl;
}

}

int main()
```

```
{
 otl_connect::otl_initialize(); // initialize OCI environment
 try{

  db.rlogon("scott/tiger"); // connect to Oracle

  otl_cursor::direct_exec
   (
    db,
    "drop table test_tab",
    otl_exception::disabled // disable OTL exceptions
   ); // drop table

  otl_cursor::direct_exec
   (
    db,
    "create table test_tab(f1 number, f2 varchar2(30))"
    );   // create table

  insert(); // insert records into table
  select(); // select records from table

 }

 catch(otl_exception& p){ // intercept OTL exceptions
  cerr<<p.msg<<endl; // print out error message
  cerr<<p.stm_text<<endl; // print out SQL that caused the error
  cerr<<p.var_info<<endl; // print out the variable that caused the error
 }

 db.logoff(); // disconnect from Oracle

 return 0;
```

```
}
```

输出结果：

```
f1=12345678900000008, f2=Name8
f1=12345678900000009, f2=Name9
f1=12345678900000010, f2=Name10
f1=12345678900000011, f2=Name11
f1=12345678900000012, f2=Name12
f1=12345678900000013, f2=Name13
f1=12345678900000014, f2=Name14
f1=12345678900000015, f2=Name15
f1=12345678900000016, f2=Name16
f1=12345678900000004, f2=Name4
f1=12345678900000005, f2=Name5
f1=12345678900000006, f2=Name6
f1=12345678900000007, f2=Name7
f1=12345678900000008, f2=Name8
```

12.3 使用 OTL tracing 跟踪 OTL 的方法调用

OTL 可以通过宏定义打开内部的跟踪机制，方便程序的调试。通过宏 OTL_TRACE_LEVEL 指明跟踪的级别，通过宏 OTL_TRACE_STREAM 指明跟踪消息的输出流，通过宏 OTL_TRACE_LINE_PREFIX 指明跟踪消息的输出头。

以下是使用 OTL tracing 来跟踪 OTL 方法调用的示例代码。

```
#include <iostream>
using namespace std;
#include <stdio.h>
unsigned int my_trace_level=
    0x1 | // 1st level of tracing
    0x2 | // 2nd level of tracing
    0x4 | // 3rd level of tracing
    0x8 | // 4th level of tracing
    0x10; // 5th level of tracing
// each level of tracing is represented by its own bit,
// so levels of tracing can be combined in an arbitrary order.

#define OTL_TRACE_LEVEL my_trace_level
    // enables OTL tracing, and uses my_trace_level as a trace control variable.

#define OTL_TRACE_STREAM cerr
```

```
    // directs all OTL tracing to cerr

#define OTL_TRACE_LINE_PREFIX "MY OTL TRACE ==> "
    // redefines the default OTL trace line prefix. This #define is optional

#define OTL_ORA9I // Compile OTL 4.0/OCI9i
// #define OTL_ORA8I // Compile OTL 4.0/OCI8i
// #define OTL_ORA8 // Compile OTL 4.0/OCI8
#include <otlv4.h> // include the OTL 4.0 header file

otl_connect db; // connect object

void insert()
// insert rows into table
{
 otl_stream o(10, // buffer size
                  "insert into test_tab values(:f1<int>,:f2<char[31]>)", // SQL statement
                  db // connect object
                 );
 char tmp[32];

 for(int i=1;i<=23;++i){
  sprintf(tmp,"Name%d",i);
  o<<i<<tmp;
 }
}

void select()
{
 otl_stream i(5, // buffer size
                  "select * from test_tab where f1>=:f<int> and f1<=:ff<int>*2",
                      // SELECT statement
                  db // connect object
```

```
                    );
    // create select stream

 float f1;
 char f2[31];

 i<<8<<8; // assigning :f = 8; :ff = 8
    // SELECT automatically executes when all input variables are
    // assigned. First portion of output rows is fetched to the buffer

 while(!i.eof()){ // while not end-of-data
  i>>f1>>f2;
  cout<<"f1="<<f1<<", f2="<<f2<<endl;
 }

}

int main()
{
 otl_connect::otl_initialize(); // initialize OCI environment
 try{

  db.rlogon("scott/tiger"); // connect to the database

  otl_cursor::direct_exec
   (
    db,
    "drop table test_tab",
    otl_exception::disabled // disable OTL exceptions
   ); // drop table

  otl_cursor::direct_exec
   (
```

```
      db,
      "create table test_tab(f1 int, f2 varchar(30))"
      );    // create table

   insert(); // insert records into table
   select(); // select records from table

  }
  catch(otl_exception& p){ // intercept OTL exceptions
   cerr<<p.msg<<endl; // print out error message
   cerr<<p.stm_text<<endl; // print out SQL that caused the error
   cerr<<p.var_info<<endl; // print out the variable that caused the error
  }

  db.logoff(); // disconnect from the database

  return 0;
}
```

输出结果：

```
MY OTL TRACE ==> otl_connect(this=004332D4)::rlogon(connect_str="scott/*****",
auto_commit=0);
MY OTL TRACE ==> otl_cursor::direct_exec(connect=004332CC,sqlstm="drop table
test_tab",exception_enabled=0);
MY OTL TRACE ==> otl_cursor::direct_exec(connect=004332CC,sqlstm="create table
test_tab(f1 int, f2 varchar(30))",exception_enabled=1);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::open(buffer_size=10, sqlstm=insert into
test_tab values(:f1<int>,:f2<char[31]>), connect=004332CC);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=1);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name1");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=2);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name2");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=3);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name3");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
```

```
placeholder=:f1, value=4);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name4");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=5);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name5");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=6);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name6");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=7);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name7");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=8);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name8");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=9);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name9");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=10);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name10");
MY OTL TRACE ==> otl_stream, executing SQL Stm=insert into test_tab
values(:f1         ,:f2                  ), current batch size=10, row offset=0
MY OTL TRACE ==> otl_connect(this=004332CC)::commit();
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=11);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name11");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=12);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name12");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=13);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name13");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=14);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name14");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=15);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name15");
```

```
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=16);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name16");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=17);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name17");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=18);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name18");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=19);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name19");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=20);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name20");
MY OTL TRACE ==> otl_stream, executing SQL Stm=insert into test_tab
values(:f1          ,:f2                    ), current batch size=10, row offset=0
MY OTL TRACE ==> otl_connect(this=004332CC)::commit();
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=21);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name21");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=22);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name22");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4,
placeholder=:f1, value=23);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(char*: ftype=1,
placeholder=:f2, value="Name23");
MY OTL TRACE ==> otl_stream(this=0012FEFC)::close();
MY OTL TRACE ==> otl_stream, executing SQL Stm=insert into test_tab
values(:f1          ,:f2                    ), current batch size=3, row offset=0
MY OTL TRACE ==> otl_connect(this=004332CC)::commit();
MY OTL TRACE ==> otl_stream(this=0012FEFC)::open(buffer_size=5, sqlstm=select *
from test_tab where f1>=:f<int> and f1<=:ff<int>*2, connect=004332CC);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4, placeholder=:f,
value=8);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator <<(int: ftype=4, placeholder=:ff,
value=8);
MY OTL TRACE ==> otl_stream, executing SQL Stm=select * from test_tab where f1>=:f
and f1<=:ff          *2, buffer size=5
MY OTL TRACE ==> otl_stream, fetched the first batch of rows, SQL Stm=select * from
test_tab where f1>=:f            and f1<=:ff          *2, RPC=5
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(float& : ftype=2,
```

```
placeholder=F1, value=8);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(char* : ftype=1,
placeholder=F2, value="Name8");
f1=8, f2=Name8
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(float& : ftype=2,
placeholder=F1, value=9);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(char* : ftype=1,
placeholder=F2, value="Name9");
f1=9, f2=Name9
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(float& : ftype=2,
placeholder=F1, value=10);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(char* : ftype=1,
placeholder=F2, value="Name10");
f1=10, f2=Name10
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(float& : ftype=2,
placeholder=F1, value=11);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(char* : ftype=1,
placeholder=F2, value="Name11");
f1=11, f2=Name11
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(float& : ftype=2,
placeholder=F1, value=12);
MY OTL TRACE ==> otl_stream, fetched the next batch of rows, SQL Stm=select * from
test_tab where f1>=:f           and f1<=:ff         *2, RPC=9
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(char* : ftype=1,
placeholder=F2, value="Name12");
f1=12, f2=Name12
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(float& : ftype=2,
placeholder=F1, value=13);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(char* : ftype=1,
placeholder=F2, value="Name13");
f1=13, f2=Name13
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(float& : ftype=2,
placeholder=F1, value=14);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(char* : ftype=1,
placeholder=F2, value="Name14");
f1=14, f2=Name14
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(float& : ftype=2,
placeholder=F1, value=15);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(char* : ftype=1,
placeholder=F2, value="Name15");
f1=15, f2=Name15
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(float& : ftype=2,
placeholder=F1, value=16);
MY OTL TRACE ==> otl_stream(this=0012FEFC)::operator >>(char* : ftype=1,
placeholder=F2, value="Name16");
f1=16, f2=Name16
MY OTL TRACE ==> otl_stream(this=0012FEFC)::close();
MY OTL TRACE ==> otl_connect(this=004332CC)::logoff();
```

12.4 获取已处理行数(Rows Processed Count)

otl_stream 提供了 get_rpc()方法(参见 4.1 小节)获取已处理行数。另外，类 otl_cursor 的 direct_exec()方法如果处理成功也返回已处理行数。

已处理行数和 INSERT、UPDATE 以及 DELETE 语句有关。对于 INSERT 语句而言，已处理行数可能小于等于流的缓冲区大小。对于 UPDATE 以及 DELETE 而言，则和多少行被更新或删除相关。

以下是获取已处理行数的示例代码。

```
#include <iostream>
using namespace std;

#include <stdio.h>
#define OTL_ORA8 // Compile OTL 4.0/OCI8
#include <otlv4.h> // include the OTL 4.0 header file


otl_connect db; // connect object


void insert()
// insert rows into table
{
 otl_stream o(200, // buffer size
              "insert into test_tab values(:f1<float>,:f2<char[31]>)",
                // SQL statement
              db // connect object
             );
 char tmp[32];

 for(int i=1;i<=123;++i){
   sprintf(tmp,"Name%d",i);
   o<<(float)i<<tmp;
 }
 o.flush();


 cout<<"Rows inserted: "<<o.get_rpc()<<endl;
```

```
}

void delete_rows()
{
  long rpc=otl_cursor::direct_exec(db,"delete from test_tab where f1>=95");

  cout<<"Rows deleted: "<<rpc<<endl;
}

int main()
{
 otl_connect::otl_initialize(); // initialize OCI environment
 try{

  db.rlogon("scott/tiger"); // connect to Oracle

  otl_cursor::direct_exec
   (
    db,
    "drop table test_tab",
    otl_exception::disabled // disable OTL exceptions
   ); // drop table

  otl_cursor::direct_exec
   (
    db,
    "create table test_tab(f1 number, f2 varchar2(30))"
    );   // create table

  insert(); // insert records into table
  delete_rows(); // select records from table

 }
```

```
 catch(otl_exception& p){ // intercept OTL exceptions
   cerr<<p.msg<<endl; // print out error message
   cerr<<p.stm_text<<endl; // print out SQL that caused the error
   cerr<<p.var_info<<endl; // print out the variable that caused the error
 }

 db.logoff(); // disconnect from Oracle

 return 0;

}
```

输出结果：

```
Rows inserted: 123
Rows deleted: 29
```

12.5 使用 otl_connect 的重载运算符<<, <<=, >>

otl_connect 重载了<<, <<=和>>运算符(参见 4.2 小节)，用于简化编程操作。其中操作符<<发送字符串到 otl_connect 对象。如果该 otl_connect 对象还没有连接到数据库则字符串为"userid/passwd@db"格式的连接字符串，它使得 otl_connect 对象能够立刻连接数据库。如果该 otl_connect 对象已经连接到数据库则字符串被当作为静态 SQL 语句立刻执行。

操作符<<=发送字符串到 otl_connect 对象。otl_connect 对象将保存该字符串并被下一个>>操作符使用。该字符串是一个拥有占位符并且能够发送到 otl_stream 对象的 SQL 语句。操作符>>取出之前使用操作符<<=保存的 SQL 语句并则发送到 otl_stream 对象。它使得该 SQL 语句被 otl_stream 打开。

以下是使用 otl_connect 重载的<<, <<=和>>运算符示例代码。

```
#include <iostream>
using namespace std;

#include <stdio.h>
// #define OTL_ORA8 // Compile OTL 4.0/OCI8
// #define OTL_ORA8I // Compile OTL 4.0/OCI8i
```

```
// #define OTL_ORA9I // Compile OTL 4.0/OCI9I
// #define OTL_ORA10G // Compile OTL 4.0/OCI10g
#define OTL_ORA10G_R2 // Compile OTL 4.0/OCI10gR2
#include <otlv4.h> // include the OTL 4.0 header file

otl_connect db; // connect object

void insert()
// insert rows into table
{
  otl_stream o;
  o.setBufSize(50);

// Send a message (SQL statement) to the otl_connect object.
  db<<="insert into test_tab values(:f1<int>,:f2<char[31]>)";

// Send a message (SQL statement) from the connect object
// to the otl_stream object.

  db>>o;

// By and large, this is all syntactical sugar, but "some like it hot".

  char tmp[32];

  for(int i=1;i<=100;++i){
    sprintf(tmp,"Name%d",i);
    o<<i<<tmp;
  }
}

void select()
{
```

```
  otl_stream i;

  i.setBufSize(50);

// Send a message (SQL statement) to the otl_connect object.
  db<<="select * from test_tab where f1>=:f11<int> and f1<=:f12<int>*2";

// Send a message (SQL statement) from the connect object
// to the otl_stream object.

  db>>i;

// By and large, this is all syntactical sugar, but "some like it hot".

  int f1;
  char f2[31];

  i<<8<<8; // assigning :f11 = 8, :f12 = 8
     // SELECT automatically executes when all input variables are
     // assigned. First portion of output rows is fetched to the buffer

  while(!i.eof()){ // while not end-of-data
    i>>f1>>f2;
    cout<<"f1="<<f1<<", f2="<<f2<<endl;
  }

  i<<4<<4; // assigning :f11 = 8, :f12 = 8
     // SELECT automatically executes when all input variables are
     // assigned. First portion of output rows is fetched to the buffer

  while(!i.eof()){ // while not end-of-data
    i>>f1>>f2;
    cout<<"f1="<<f1<<", f2="<<f2<<endl;
```

```
  }

}

int main()
{
  otl_connect::otl_initialize(); // initialize the database API environment
  try{

    db<<"scott/tiger"; // connect to the database

    // Send SQL statements to the connect obejct for immediate execution.
    // Ignore any exception for the first statement.
    try{ db<<"drop table test_tab"; } catch(otl_exception&){}
    db<<"create table test_tab(f1 int, f2 varchar(30))";

    insert(); // insert records into table
    select(); // select records from table

  }

  catch(otl_exception& p){ // intercept OTL exceptions
    cerr<<p.msg<<endl; // print out error message
    cerr<<p.stm_text<<endl; // print out SQL that caused the error
    cerr<<p.var_info<<endl; // print out the variable that caused the error
  }

  db.logoff(); // disconnect from the database

  return 0;

}
```

输出结果：

```
f1=8, f2=Name8
f1=9, f2=Name9
f1=10, f2=Name10
f1=11, f2=Name11
f1=12, f2=Name12
f1=13, f2=Name13
f1=14, f2=Name14
f1=15, f2=Name15
f1=16, f2=Name16
f1=4, f2=Name4
f1=5, f2=Name5
f1=6, f2=Name6
f1=7, f2=Name7
f1=8, f2=Name8
```

12.6 手工刷新 otl_stream 缓冲区

otl_stream 提供了方法 set_flush()方法(见 4.1 小节)设置 auto_flush 标志。该标志仅仅控制析构函数中由于输出缓冲区数据变脏引起的刷新，并不能控制缓冲区填满引起自动刷新。auto_flush 标志默认为 true 即进行刷新。

提供 set_flush()方法控制 otl_stream 析构函数中的刷新操作和 C++的异常机制有关。C++异常机制退出作用域时，局部对象会由于堆栈回退发生析构。如果在析构函数中又产生异常将导致 std::terminate()被调用，从而使程序中止。

otl_stream 析构函数中的输出缓冲区刷新在某些情况下极有可能导致这种级联异常发生，因此 otl_stream 提供了 set_flush()方法进行控制。如果 auto_flush 被设置成 false，则必须手工调用 otl_stream 的 flush()或 close()方法进行刷新。当然，前提是输出缓冲区还没有填满，因为缓冲区填满的自动刷新是不能通过 set_flush()控制的。

以下是手工刷新 otl_stream 缓冲区的示例代码。

```
#include <iostream>
#include <stdio.h>

// Uncomment the line below when OCI7 is used with OTL
// #define OTL_ORA7 // Compile OTL 4.0/OCI7

using namespace std;

#define OTL_ORA8 // Compile OTL 4.0/OCI8
#include <otlv4.h> // include the OTL 4.0 header file

otl_connect db; // connect object

void insert1()
// insert rows into table
```

```
{
  otl_stream o; // define an otl_stream variable

  o.set_flush(false); // set the auto-flush flag to OFF.

  o.open(200, // buffer size
           "insert into test_tab values(:f1<float>,:f2<char[31]>)", // SQL statement
           db // connect object
          );

  char tmp[32];
  for(int i=1;i<=100;++i){
       sprintf(tmp,"Name%d",i);
       o<<(float)i<<tmp;
       if(i%55==0)
            throw "Throwing an exception";
  }

  o.flush();      // when the auto-flush flag is OFF, an explicit flush
                  // of the stream buffer is required in case of successful
                  // completion of execution of the INSERT statement.
                  // In case of a raised exception, the stream buffer would not be flushed.
}

void insert2()
// insert rows into table
{
  otl_stream o; // define an otl_stream variable


  o.set_flush(false); // set the auto-flush flag to OFF.


  o.open(200, // buffer size
           "insert into test_tab values(:f1<float>,:f2<char[31]>)", // SQL statement
           db // connect object
          );
  char tmp[32];


  for(int i=1;i<=100;++i){
       sprintf(tmp,"Name%d",i);
```

```
      o<<(float)i<<tmp;
      //if(i%55==0)
      //      throw "Throwing an exception";
  }
  o.flush();      // when the auto-flush flag is OFF, an explicit flush
                  // of the stream buffer is required in case of successful
                  // completion of execution of the INSERT statement.
                  // In case of a raised exception, the stream buffer would not be flushed.
}

void select()
{
  otl_stream i(50, // buffer size
                    "select * from test_tab where f1>=:f<int> and f1<=:f*2",
                        // SELECT statement
                    db // connect object
                   );
     // create select stream

  float f1;
  char f2[31];

  i<<8; // assigning :f = 8
     // SELECT automatically executes when all input variables are
     // assigned. First portion of output rows is fetched to the buffer

  while(!i.eof()){ // while not end-of-data
    i>>f1>>f2;
    cout<<"f1="<<f1<<", f2="<<f2<<endl;
  }

  i<<4; // assigning :f = 4
     // SELECT automatically executes when all input variables are
```

```
    // assigned. First portion of output rows is fetched to the buffer

  while(!i.eof()){ // while not end-of-data
   i>>f1>>f2;
   cout<<"f1="<<f1<<", f2="<<f2<<endl;
  }

}

int main()
{
  otl_connect::otl_initialize(); // initialize OCI environment
  try{

   db.rlogon("scott/tiger"); // connect to Oracle

   otl_cursor::direct_exec
    (
        db,
        "drop table test_tab",
        otl_exception::disabled // disable OTL exceptions
    ); // drop table

   otl_cursor::direct_exec
    (
        db,
        "create table test_tab(f1 number, f2 varchar2(30))"
    ); // create table

   try{
       insert1(); // insert records into table
   }catch(const char* p){
       cout<<p<<endl;
```

```
    }
    cout<<"Selecting the first time around:"<<endl;
    select(); // select records from table

    insert2();
    cout<<"Selecting the second time around:"<<endl;
    select();

  }

  catch(otl_exception& p){ // intercept OTL exceptions
       cerr<<p.msg<<endl; // print out error message
       cerr<<p.stm_text<<endl; // print out SQL that caused the error
       cerr<<p.var_info<<endl; // print out the variable that caused the error
  }

  db.logoff(); // disconnect from Oracle

  return 0;
}
```

输出结果：

```
Throwing an exception
Selecting the first time around:
Selecting the second time around:
f1=8, f2=Name8
f1=9, f2=Name9
f1=10, f2=Name10
f1=11, f2=Name11
f1=12, f2=Name12
f1=13, f2=Name13
f1=14, f2=Name14
f1=15, f2=Name15
f1=16, f2=Name16
f1=4, f2=Name4
f1=5, f2=Name5
f1=6, f2=Name6
f1=7, f2=Name7
f1=8, f2=Name8
```

12.7 忽略 INSERT 操作时的重复键异常

底层操作为 Oracle API 时，otl_stream 提供了特殊版本的 flush()方法(参见 4.1 小节)，该方法可以设置刷新的起始行以及忽略产生错误时抛出的 otl_exception 异常继续刷新。在进行 INSERT 操作有时候需要忽略重复键值引起的错误继续处理，这时可以利用该 flush()方法进行处理。

以下是忽略 INSERT 操作时的重复键异常示例代码。

```
#include <iostream>
using namespace std;

#include <stdio.h>
//#define OTL_ORA8I // Compile OTL 4.0/OCI8
#define OTL_ORA8I // Compile OTL 4.0/OCI8i
// #define OTL_ORA9I // Compile OTL 4.0/OCI9i
#include <otlv4.h> // include the OTL 4.0 header file


otl_connect db; // connect object


void insert()
// insert rows into table
{
 otl_stream o(10, // make the buffer size larger than the actual
                  // row set to inserted, so that the stream will not
                  // flush the buffer automatically
              "insert into test_tab values(:f1<int>,:f2<char[31]>)", // SQL statement
              db // connect object
             );

 o.set_commit(0); // set stream's auto-commit to OFF.

 long total_rpc=0; // total "rows processed count"
 long rpc=0; // rows successfully processed in one flush() call
 int iters=0; // number of rows to be bypassed

 try{
```

```
  o<<1<<"Line1"; // Enter one row into the stream
  o<<1<<"Line1"; // Enter the same data into the stream
                 // and cause a "duplicate key" error.
  o<<2<<"Line2"; // Enter one row into the stream
  o<<3<<"Line3"; // Enter one row into the stream
  o<<4<<"Line4"; // Enter one row into the stream
  o<<4<<"Line4"; // Enter the same data into the stream
            // and cause a "duplicate key" error.
  o.flush();
}catch(otl_exception& p){
     if(p.code==1){ // ORA-0001: ...duplicate key...
          ++iters;
          rpc=o.get_rpc();
          total_rpc=rpc;
          do{
               try{
                    cout<<"TOTAL_RPC="<<total_rpc<<", RPC="<<rpc<<endl;
                    o.flush(total_rpc+iters, // bypass the duplicate row and start
                                             // with the rows after that
                            true // force buffer flushing regardless
                    );
                    rpc=0;
               }catch(otl_exception& p2){
                    if(p2.code==1){ // ORA-0001: ... duplicate key ...
                         ++iters;
                         rpc=o.get_rpc();
                         total_rpc+=rpc;
                    }else
                         throw;
               }
          }while(rpc>0);
  }else
     throw; // re-throw the exception to the outer catch block.
```

```
 }

 db.commit(); // commit transaction

}

void select()
{
 otl_stream i(10, // buffer size
              "select * from test_tab",
                // SELECT statement
              db // connect object
             );
   // create select stream

 int f1;
 char f2[31];

 while(!i.eof()){ // while not end-of-data
  i>>f1>>f2;
  cout<<"f1="<<f1<<", f2="<<f2<<endl;
 }

}

int main()
{
 otl_connect::otl_initialize(); // initialize OCI environment
 try{

  db.rlogon("scott/tiger"); // connect to Oracle

  otl_cursor::direct_exec
```

```
  (
   db,
   "drop table test_tab",
   otl_exception::disabled // disable OTL exceptions
  ); // drop table

 otl_cursor::direct_exec
  (
   db,
   "create table test_tab(f1 number, f2 varchar2(30))"
   );   // create table

 otl_cursor::direct_exec
  (
   db,
   "create unique index ind001 on test_tab(f1)"
   );   // create unique index

 insert(); // insert records into table
 select(); // select records from table

}

catch(otl_exception& p){ // intercept OTL exceptions
 cerr<<p.msg<<endl; // print out error message
 cerr<<p.stm_text<<endl; // print out SQL that caused the error
 cerr<<p.var_info<<endl; // print out the variable that caused the error
}

db.logoff(); // disconnect from Oracle

return 0;
```

```
}
```

输出结果：

```
TOTAL_RPC=1, RPC=1
TOTAL_RPC=4, RPC=3
f1=1, f2=Line1
f1=2, f2=Line2
f1=3, f2=Line3
f1=4, f2=Line4
```

12.8 使用模板 otl_value<T>创建数据容器

otl_value<T>是一个 OTL 模板类，能够基于标量的数据类型包括 int, unsigned, long, short, float, double, otl_datetime (OTL date&time container), std::string (STL string class)来创建派生的数据容器。

派生的数据容器具有内建的 NULL 指示器功能，即容器内部拥有一个 NULL 指示器域并且能够从一个操作传递到另一个操作。例如从 SELECT 语句读取并装入容器 otl_value<int>的值是 NULL，在该值被写入 INSERT 语句后，otl_value<int>容器中的 NULL 指示器域仍然保存该 NULL 值并且 NULL 值也从 SELECT 传递到了 INSERT。

模板 otl_value<T> 的使用可以通过”#define OTL_STL” 或者 ”#define OTL_VALUE_TEMPLATE_ON”激活。其定义如下：

```
template<class TData>
class otl_value{
public:
    TData v;
    bool ind;

    otl_value(); // default constructor

    otl_value(const otl_value<TData>& var); // copy constructor
    otl_value(const TData& var); // copy constructor
    otl_value(const otl_null& var); // copy constructor

    otl_value<TData>& operator=(const otl_value<TData>& var); //assignment operator
    otl_value<TData>& operator=(const TData& var); // assignment operator
    otl_value<TData>& operator=(const otl_null& var); // assignment operator

    bool is_null(void) const;
    void set_null(void);
    void set_non_null(void);
}; // end of otl_value

template <class TData>
otl_stream& operator<<(otl_stream& s, const otl_value<TData>& var);
```

```
template <class TData>
otl_stream& operator>>(otl_stream& s, otl_value<TData>& var);

template <class TData> std::ostream&
operator<<(std::ostream& s, const otl_value<TData>& var);
```

其中方法 set_null()和 set_non_null()将 otl_value 设置成 NULL 或者非 NULL，这两个方法必须直接操作 public 数据成员 TData v 或 bool ind 的时候才使用。数据成员 v 为标量数据类型值的存放容器，成员 ind 则为 NULL 指示器。

以下为使用 otl_value<T>模板的示例代码。

```
#define OTL_ORA8 // Compile OTL 4.0/OCI8
#define OTL_STL // Turn on STL features and otl_value<T>
#define OTL_ANSI_CPP // Turn on ANSI C++ typecasts
#include <otlv4.h> // include the OTL 4.0 header file

using namespace std;

otl_connect db; // connect object

void insert()
// insert rows into table
{
 otl_stream o(50, // buffer size
                  "insert into test_tab "
                  "values(:f1<int>,:f2<char[31]>,:f3<timestamp>)",
                      // SQL statement
                  db // connect object
                 );

 otl_value<string> f2; // otl_value container of STL string
 otl_value<otl_datetime> f3; // container of otl_datetime


 for(int i=1;i<=100;++i){

  if(i%2==0)
      f2="NameXXX";
  else
      f2=otl_null(); // Assign otl_null() to f2

  if(i%3==0){
      // Assign a value to f3 via the .v field directly
      f3.v.year=2001;
      f3.v.month=1;
      f3.v.day=1;
      f3.v.hour=13;
      f3.v.minute=10;
```

```
        f3.v.second=5;
        f3.set_non_null(); // Set f3 as a "non-NULL"
    }else
        f3.set_null(); // Set f3 as null via .set_null() function

    o<<i<<f2<<f3;

  }
}

void select()
{
  otl_stream i(50, // buffer size
                   "select * from test_tab where f1>=:f<int> and f1<=:f*2",
                        // SELECT statement
                   db // connect object
                  );
     // create select stream

  int f1;
  otl_value<string> f2;
  otl_value<otl_datetime> f3;


  i<<8; // assigning :f = 8
     // SELECT automatically executes when all input variables are
     // assigned. First portion of output rows is fetched to the buffer

  while(!i.eof()){ // while not end-of-data
      i>>f1>>f2>>f3;
      cout<<"f1="<<f1<<", f2="<<f2<<", f3="<<f3<<endl;
  }

  i<<4; // assigning :f = 4
     // SELECT automatically executes when all input variables are
     // assigned. First portion of output rows is fetched to the buffer

  while(!i.eof()){ // while not end-of-data
    i>>f1>>f2>>f3;
    cout<<"f1="<<f1<<", f2="<<f2<<", f3="<<f3<<endl;
  }

}

int main()
{
  otl_connect::otl_initialize(); // initialize OCI environment
  try{

    db.rlogon("scott/tiger"); // connect to Oracle
```

```
  otl_cursor::direct_exec
   (
    db,
    "drop table test_tab",
    otl_exception::disabled // disable OTL exceptions
   ); // drop table

  otl_cursor::direct_exec
   (
    db,
    "create table test_tab(f1 number, f2 varchar2(30), f3 date)"
    );    // create table

  insert(); // insert records into table
  select(); // select records from table

 }

 catch(otl_exception& p){ // intercept OTL exceptions
  cerr<<p.msg<<endl; // print out error message
  cerr<<p.stm_text<<endl; // print out SQL that caused the error
  cerr<<p.var_info<<endl; // print out the variable that caused the error
 }

 db.logoff(); // disconnect from Oracle

 return 0;

}
```

输出结果：

```
f1=8, f2=NameXXX, f3=NULL
f1=9, f2=NULL, f3=1/1/2001 13:10:5
f1=10, f2=NameXXX, f3=NULL
f1=11, f2=NULL, f3=NULL
f1=12, f2=NameXXX, f3=1/1/2001 13:10:5
f1=13, f2=NULL, f3=NULL
f1=14, f2=NameXXX, f3=NULL
f1=15, f2=NULL, f3=1/1/2001 13:10:5
f1=16, f2=NameXXX, f3=NULL
f1=4, f2=NameXXX, f3=NULL
f1=5, f2=NULL, f3=NULL
f1=6, f2=NameXXX, f3=1/1/2001 13:10:5
f1=7, f2=NULL, f3=NULL
f1=8, f2=NameXXX, f3=NULL
```

12.9 使用 OTL 流的读迭代器遍历流返回的 Reference Cursor

otl_stream 的构造函数和 open()方法(参见 4.1 小节)中参数 sqlstm 可以为 SQL，也可

以为 PL/SQL 块或者存储过程调用，如果流返回 reference cursor，则需要在参数 ref_cur_placeholder 中指明占位符号。

以下是使用使用 OTL 流的读迭代器遍历 OTL 流返回的 reference cursor 的示例代码。

```
#include <iostream>
using namespace std;

#include <stdio.h>
// #define OTL_ORA7 // Compile OTL 4.0/OCI7
//#define OTL_ORA8 // Compile OTL 4.0/OCI8
//#define OTL_ORA8I // Compile OTL 4.0/OCI8i
#define OTL_ORA9I // Compile OTL 4.0/OCI9i
//#define OTL_ORA10G // Compile OTL 4.0/OCI10g
#define OTL_STREAM_READ_ITERATOR_ON
#include <otlv4.h> // include the OTL 4.0 header file

otl_connect db; // connect object

void insert()
// insert rows into table
{
  otl_stream o(50, // buffer size
                       "insert into test_tab values(:f1<int>,:f2<char[31]>)", // SQL statement
                       db // connect object
                      );
  char tmp[32];

  for(int i=1;i<=100;++i){
    sprintf(tmp,"Name%d",i);
    o<<i<<tmp;
  }
}

void select()
```

```
{
 otl_stream i(50, // buffer size
                  "begin "
                  "     open :cur1 for "
                  "     select * from test_tab "
                  "     where f1>=:f11<int> "
                  "        and f1<=:f12<int>*2; "
                  "end;", // SELECT statement
                  db, // connect object
                  ":cur1"
                 );
   // create select stream

 int f1;
 char f2[31];
 otl_stream_read_iterator<otl_stream,otl_exception,otl_lob_stream> rs;

 i<<8<<8; // assigning :f11 = 8, :f12 = 8
   // SELECT automatically executes when all input variables are
   // assigned. First portion of output rows is fetched to the buffer

 rs.attach(i); // attach the iterator "rs" to the stream "i"
                  // after the PL/SQL block is executed.
 while(rs.next_row()){ // while not end-of-data
     rs.get(1,f1);
     rs.get(2,f2);
     cout<<"f1="<<f1<<", f2="<<f2<<endl;
 }

 rs.detach(); // detach the itertor from the stream

 i<<4<<4; // assigning :f11 = 4, :f12 = 4
   // SELECT automatically executes when all input variables are
```

```
    // assigned. First portion of output rows is fetched to the buffer

  while(!i.eof()){ // while not end-of-data
      i>>f1>>f2;
      cout<<"f1="<<f1<<", f2="<<f2<<endl;
  }

}

int main()
{
  otl_connect::otl_initialize(); // initialize OCI environment
  try{

    db.rlogon("scott/tiger"); // connect to Oracle

    otl_cursor::direct_exec
     (
      db,
      "drop table test_tab",
      otl_exception::disabled // disable OTL exceptions
     ); // drop table

    otl_cursor::direct_exec
     (
      db,
      "create table test_tab(f1 number, f2 varchar2(30))"
      );    // create table

    insert(); // insert records into table
    select(); // select records from table

  }
```

```
 catch(otl_exception& p){ // intercept OTL exceptions
  cerr<<p.msg<<endl; // print out error message
  cerr<<p.stm_text<<endl; // print out SQL that caused the error
  cerr<<p.var_info<<endl; // print out the variable that caused the error
 }

 db.logoff(); // disconnect from Oracle

 return 0;

}
```

输出结果：

```
f1=8, f2=Name8
f1=9, f2=Name9
f1=10, f2=Name10
f1=11, f2=Name11
f1=12, f2=Name12
f1=13, f2=Name13
f1=14, f2=Name14
f1=15, f2=Name15
f1=16, f2=Name16
f1=4, f2=Name4
f1=5, f2=Name5
f1=6, f2=Name6
f1=7, f2=Name7
f1=8, f2=Name8
```

12.10 使用 Reference Cursor 流从存储过程中返回多个 Referece Cursor

otl_refcur_stream 和 reference cursor 类型的占位符联合，允许在存储过程调用中返回多个 reference cursor。以下为示例代码。

```
#include <iostream>
using namespace std;

#include <stdio.h>
#define OTL_ORA8 // Compile OTL 4.0/OCI8
//#define OTL_ORA8I // Compile OTL 4.0/OCI8i
//#define OTL_ORA9I // Compile OTL 4.0/OCI9i
```

```
#include <otlv4.h> // include the OTL 4.0 header file

otl_connect db; // connect object

void insert()
// insert rows into table
{
 otl_stream o(50, // buffer size
              "insert into test_tab values(:f1<int>,:f2<char[31]>)", // SQL statement
              db // connect object
             );
 char tmp[32];

 for(int i=1;i<=100;++i){
   sprintf(tmp,"Name%d",i);
   o<<i<<tmp;
 }
}

void select()
{
    // :str1 is an output string parameter
    // :cur1, :cur2 are a bind variable names, refcur -- their types,
    // out -- output parameter, 50 -- the buffer size when this
    // reference cursor will be attached to otl_refcur_stream
    otl_stream i(1, // buffer size
                 "begin "
                 " my_pkg.my_proc(:f1<int,in>,:f2<int,in>, "
                 "          :str1<char[100],out>, "
                 "          :cur1<refcur,out[50]>, "
                 "          :cur2<refcur,out[50]>); "
                 "end;", // PL/SQL block that calls a stored procedure
                 db // connect object
```

```
            );

 i.set_commit(0); // set stream "auto-commit" to OFF.
char str1[101];
float f1;
char f2[31];
otl_refcur_stream s1, s2; // reference cursor streams for reading rows.

i<<8<<4; // assigning :f1 = 8, :f2 = 4
i>>str1; // reading :str1 from the stream
i>>s1; // initializing the refeence cursor stream with the output
        // reference cursor :cur1
i>>s2; // initializing the refeence cursor stream with the output
        // reference cursor :cur2

cout<<"STR1="<<str1<<endl;
cout<<"=====> Reading :cur1..."<<endl;
while(!s1.eof()){ // while not end-of-data
    s1>>f1>>f2;
    cout<<"f1="<<f1<<", f2="<<f2<<endl;
}

cout<<"=====> Reading :cur2..."<<endl;
while(!s2.eof()){ // while not end-of-data
    s2>>f1>>f2;
    cout<<"f1="<<f1<<", f2="<<f2<<endl;
}

 s1.close(); // closing the reference cursor
 s2.close(); // closing the reference cursor
}

int main()
```

```
{
 otl_connect::otl_initialize(); // initialize OCI environment
 try{

  db.rlogon("scott/tiger"); // connect to Oracle

  otl_cursor::direct_exec
   (
    db,
    "drop table test_tab",
    otl_exception::disabled // disable OTL exceptions
   ); // drop table

  otl_cursor::direct_exec
   (
    db,
    "create table test_tab(f1 number, f2 varchar2(30))"
    );   // create table

  // create a PL/SQL package
  otl_cursor::direct_exec
   (db,
    "CREATE OR REPLACE PACKAGE my_pkg IS "
    " "
    "   TYPE my_cursor IS REF CURSOR; "
    " "
    "   PROCEDURE my_proc "
    "     (f1_in IN NUMBER, "
    "      f2_in IN NUMBER, "
    "      str1 OUT VARCHAR2, "
    "      cur1 OUT my_cursor, "
    "      cur2 OUT my_cursor); "
    " "
```

```
 "END; "
);

otl_cursor::direct_exec
 (db,
 "CREATE OR REPLACE PACKAGE BODY my_pkg IS "
 " "
 "   PROCEDURE my_proc "
 "     (f1_in IN NUMBER, "
 "      f2_in IN NUMBER, "
 "      str1 OUT VARCHAR2, "
 "      cur1 OUT my_cursor, "
 "      cur2 OUT my_cursor) "
 "   IS "
 "     lv_cur1 my_cursor; "
 "     lv_cur2 my_cursor; "
 "   BEGIN "
 "     OPEN lv_cur1 FOR "
 "       SELECT * FROM test_tab "
 "       WHERE f1>=f1_in    "
 "         AND f1<=f1_in*2; "
 "     OPEN lv_cur2 FOR "
 "       SELECT * FROM test_tab "
 "       WHERE f1>=f2_in    "
 "         AND f1<=f2_in*2; "
 "     str1 := 'Hello, world'; "
 "     cur1 := lv_cur1; "
 "     cur2 := lv_cur2; "
 "   END; "
 "    "
 "END; "
);
```

```
  insert(); // insert records into table
  select(); // select records from table

 }

 catch(otl_exception& p){ // intercept OTL exceptions
  cerr<<p.msg<<endl; // print out error message
  cerr<<p.stm_text<<endl; // print out SQL that caused the error
  cerr<<p.var_info<<endl; // print out the variable that caused the error
 }

 db.logoff(); // disconnect from Oracle

 return 0;

}
```

输出结果：

```
STR1=Hello, world
=====> Reading :cur1...
f1=8, f2=Name8
f1=9, f2=Name9
f1=10, f2=Name10
f1=11, f2=Name11
f1=12, f2=Name12
f1=13, f2=Name13
f1=14, f2=Name14
f1=15, f2=Name15
f1=16, f2=Name16
=====> Reading :cur2...
f1=4, f2=Name4
f1=5, f2=Name5
f1=6, f2=Name6
f1=7, f2=Name7
f1=8, f2=Name8
```

13 最佳实践

13.1 流缓冲区大小的设置

OTL 流的性能主要被缓冲区大小一个参数控制，在创建使用 OTL 流时必须通过构造函数或 open()函数的第一个参数 arr_size 指定，合理设置流缓冲区大小对数据库访问

的性能优化尤其重要。

与 INSERT/DELETE/UPDATE 语句不同，SELECT 语句的执行往往导致若干满足查询条件的记录行返回，OTL 的底层实现将这两类语句的操作执行区分开来，因此 SELECT 语句相关的流缓冲区在作用和使用方式上与其他语句存在很大的差别。

对于 SELECT 语句而言，缓冲区大小主要影响 OTL 底层调用 OCI 接口 OCIStmtFetch()一次性获取结果记录的行数。其底层实现如以下代码片断所示。

```
/ **
 *OTL 底层的 OCI 接口封装类 otl_cur 的 fetch()具体实现，其功能主要是
 *在执行 SELECT 语句后获取返回的记录行。
 *参数：iters 输入参数，从当前位置处一次性提取的记录数
 *       eof_date 输入输出参数，标识是否还有未获取的记录行，0 为是，1 为否
 */
int fetch(const otl_stream_buffer_size_type iters, int& eof_data)
{
    eof_data=0;
    status=OCIStmtFetch(
        cda,                                    //语句句柄
        errhp,                                  //错误句柄
        OTL_SCAST(ub4,iters),                   //从当前位置处开始一次提取的记录数
        OTL_SCAST(ub4,OCI_FETCH_NEXT), //提取方向
        OTL_SCAST(ub4,OCI_DEFAULT)      //模式
    );

    eof_status=status;
    if(status!=OCI_SUCCESS&&
        status!=OCI_SUCCESS_WITH_INFO&&
        status!=OCI_NO_DATA)
        return 0;

    //如果记录被提取完或没有数据则
    //输入输出参数 eof_data 赋值为 1，返回 1
```

```
    if(status==OCI_NO_DATA){
        eof_data=1;
        return 1;
    }
    return 1;
}
```

由于缓冲区大小作为第一参数传递给 fetch()方法进行底层的 OCI 接口调用，因此缓冲区大小决定了从当前位置处开始一次提取的记录数。

对于查询数据量较大的应用，缓冲区大小的合理设置往往是性能优化的关键。表 13-1 为查询 10000 条记录时的缓冲区大小对结果行获取的影响。数据库为 Oracle10g, 操作系统为 HP-UNIX。

13-1 查询 10000 条记录时缓冲区大小不同设置的相应耗时

缓存区大小	耗时(s)
1	0.25
10	0.14
50	0.09
100	0.06
200	0.05
300	0.05
1000	0.05
2000	0.04

备注：(1) 测试环境为 HP, Oracle10g
(2) 耗时仅指 SQL 执行成功后，记录的获取时间

由该表可见，对于查询结果为 10000 条记录的 SELECT 语句而言，缓冲区大小的设置对结果行的获取时间存在几十倍的差距。因此在使用 OTL 流进行数据库查询时，请合理设置缓冲区大小。

注意对于 SELECT 语句来说，缓冲区的大小并不影响其执行时机。当 otl_stream 以 SELECT 语句实例化，并设定好缓冲区大小后，一旦所有声明的绑定变量使用<<操作符被绑定完成，该 SELECT 语句将被 OTL 流立刻执行，与设置的缓存区具体大小并没有关系。

与此相反，对于 INSERT/DELETE/UPDATE 语句，缓冲区大小的设置将影响其执行时机。当缓冲区被 SQL 填满时，OTL 流将自动刷新缓冲区，立刻批量执行缓冲区中所

有的 SQL。因此，当执行语句为 INSERT/DELETE/UPDATE 时，也应该根据需要合理设置缓冲区大小。

13.2 批量操作注意的问题

当使用 OTL 的 SQL 变量绑定进行数据的批量操作时，应该注意 otl_stream 缓冲区大小的合理设置(参见 13.1 小节)以及 OTL 默认的语句执行成功后立刻提交事务。

对于批量操作，OTL 默认的语句执行成功后立刻提交事务的操作方式往往不实用。例如对于批量更新 10000 条记录，缓冲区大小为 1000 的情况，当缓冲区填满时更新操作被执行，执行成功后当前事务被立刻提交，如果在此后的处理中出现错误，根本没有办法通过回滚事务取消之前的更新。防止语句执行后立刻提交事务的方法请参见 12.1 小节。

OTL 流 otl_stream 的这种默认操作方式，主要和底层调用 OCI 接口 OCIStmtExecute() 的相关实现有关。其底层实现如以下代码片断所示。

```
/ **
 *OTL 底层的 OCI 接口封装类 otl_cur 的 exec ()具体实现，其功能主要是
 *在执行 SELECT 语句。
 *参数：iters 输入参数，SQL 语句执行次数
 *       rowoff   输入参数，绑定变量的开始偏移量
 */
int exec(const int iters, const int rowoff)
{
    //如果模式为语句执行成功后立刻提交事务则
    //设置相应的局部变量 mode 值
    ub4 mode;
    if(commit_on_success)
        mode=OCI_COMMIT_ON_SUCCESS;
    else
        mode=OCI_DEFAULT;

    //调用 OCI 接口 OCIStmtExecute()执行 SQL 语句
    status=OCIStmtExecute(
        db->svchp,
```

```
        cda,
        errhp,
        OTL_SCAST(ub4,iters),
        OTL_SCAST(ub4,rowoff),
        0,
        0,
        mode
    );

    //如果执行 SQL 语句错误则返回 0
    if(status!=OCI_SUCCESS)
        return 0;

    return 1;
}
```

由于 otl_stream 默认将语句执行成功立刻提交事务的开关打开，因此在 exec()方法中调用 OCI 接口时，将使用该操作模式。

13.3 与开源项目 ORAPP 的性能对比

ORAPP 是一个基于 C++语言的封装了 OCI 函数，对 Oracle 数据库进行专门访问的类库。表 13-2 所示的性能对比结果的测试环境为 Linux, Oracle10g, OTL 流缓冲区大小为 1000， 可以看出当数据量剧增时 OTL 的性能明显优越。

表 1-2 ORAPP 与 OTL 性能对比

数据量	OTL 耗时(s)	ORAPP 耗时(s)
1000	0.02	0.02
10000	0.03	0.21
100000	0.19	2.04
300000	0.56	5.77

备注：(1) 测试环境为 Linux, Oracle10g, OTL 流缓冲区大小为 1000

(2) 耗时包括 SQL 执行时间和记录获取时间的总和

究其原因，是由于 OTL 存在流缓冲区、流缓冲池等性能保证的机制，对于大数据量的查询 OTL 可以通过设置缓冲池大小，指定每次底层 OCI 调用获取的结果行，几倍甚至几十倍的提高性能。ORAPP 则将每次获取的记录行数硬编码为 1，性能不能保证。