occiwrapper 使用指南

开发 *occiwrapper* 这个库，主要是为了封装一个跨平台的 *OCCI* 的访问组件，方使 *C++*开发者们灵活地操作 *oracle* 数据库。为了方便使用，组件中的接口形式参考的 *POCO* 库的使用方式。

使用如下的形式执行 *SQL* 语句：

```
occiwrapper::Session s = SessionInstance( connection );
s << "truncate table TBL_TEST ", now;
```

通过 *session* 对象维护一个到 *oracle* 的会话。类似于流的操作方式，向 *session* 中传入 *SQL* 语句，并执行。

在 *oracle* 参数绑定方面，可以直接将 *C++*变量绑定给 *oracle* 参数中，在以后的文档说明中详细介绍。

```
occiwrapper::Session s = SessionInstance( info );
struct tm tm_value;
s << "insert into TBL_TEST( date_value ) values ( :1 )", use( tm_value ), now;
```

同时，对于 *vector* 等容量变量，可以灵活地直接绑定到 *oracle* 的绑定变量上，同时也可以灵活的将 *select* 语句返回的结果绑定到容器中，而且对于使用者来讲，并不用关心类型的对应关系。为了提高存取的性能，写入和读取都采入批量操作的方式，同时采用智能指针自动管理内存缓冲池，最大限度地解放了 *oracle* 开发者。

组件中使用的很多技术都是在工作中一些经验的积累与总结，由于自己认识 *oracle* 不够深刻，对 *C++*的理解也可能不够深入，希望大家多多讨论。对于大家发现的 Bug，我也会尽快修改，同时热情欢迎大家积极参与库的修改。

在此深深感谢我的母校西安交通大学、南京 841 所。感谢我的父母、爱人、同学、朋友，你们带给了我幸福的生活。还有我逝去的哈里、活着的蛋蛋，臭臭的小小。

对于您提交的 Bug，我们会第一时间进行修改，也欢迎您的加入！

联系作者：CUCmehp@foxmail.com
QQ 交流群：348648166

1. 下载和安装

1.1 第三方组件依赖

本组件依赖 *oracle* 公司的 *occi*（*Oracle C++ Calling Interface*）动态库支持。连接不同的 *oracle* 版本，须要选择不同的 *occi* 库版本。

关于 *occi* 的详细介绍，请参见 *oracle* 公司官方文档。

1.2 开发环境

在 *Windows* 系统下，可以通过 *Visual Studio 2008 SP1* 或者 *Visual Studio 2010* 进行编译。

在 *linux* 系统下，可以通过 *vi* 编辑器和 *g++*编译器进行编译。

1.3 运行环境

本组件可跨平台支持 *Windows* 和 *Linux* 操作系统。

1.4 数据库环境

test 工程中所有测试用例用到的数据库表、存储过程、函数，都可以通过 *db* 文件夹中的脚本创建。

方法：

⑴ 通过具有 *system* 权限的用户执行脚本 *create_user.sql*。

⑵ 使用用户 *occiwapper*，密码为 *occiwrapper* 执行 *occiwrapper.sql*。

注：例示用例中使用 *occiwrapper* 做为 *oracle* 的用户名。

1.5 测试

所有的测试代码存放在源码的 *test* 目录下，在 *windows*、*linux(suse 11*，*openSuse12*，*redhat 6* 企业版）下进行了相关的测试。数据库连接信息的配置文件放在 *db_config.ini* 中。

在程序在 *oracle 11gR2* 和 *12cR1*（*oracle 12.1*）对应的 occi 版本下进行了测试。

2. 编译说明

2.1 Windows

- 使用 *Visual Studio 2008 SP1* 工具

使用 *Visual Studio* 直接打开工程 ***occi_wrapper_vs2008.sln***，选择 *Debug* 和 *Release* 进行编译。

注：要求 *vs2008 sp1* 以上，是由于 *occiwrapper* 中使用了 *tr1* 库，而 *vs2008* 从 *SP1* 版本以上，开始支持 *tr1* 库。

- Visual Studio 2010

使用 *Visual Studio* 直接打开工程 **occi_wrapper_vs2010.sln**，选择 *Debug* 和 *Release* 进行编译。

说明：源码中 *include/occi_11g* 目录下，存放着 *oracle 11.2.0.2 64* 位数据库对应的头文件，在 *lib_vs2008* 和 *lib_vs2010* 目录下，存放着 *11.2.0.2* 版数据库对应的 *lib* 库文件。在 *bin_vs2008* 和 *bin_vs2010* 目录下，存放着对应的 *dll* 文件。

若需要使用其它版本的 *occi* 库版本，则将上述文件替换为需要使用的 *occi* 库文件。

2.2 Linux

linux 下使用，需要先下载 *oracle occi* 库对应的头文件和动态库文件。在 *oracle* 数据库的 *OCI/lib* 目录下，通常可以找到这些文件。若未安装 *oracle* 数据库，则可以从官网上下载：

http://www.oracle.com/technetwork/topics/linuxx86-64soft-092277.html

头文件对应文件包为：instantclient-sdk-linux.x64-*.*.*.*.*.zip

库文件放在 instantclient-basic-linux.x64-*.*.*.*.*.zip

只需要选择需要的 *oracle* 版本，下载对就的文件就可以，然后解压，记得将 *so* 文件放在系统运行时扫描的目录，如*/usr/local/lib*、*/usr/lib* 等目录，也可以将 *so* 的目录添加到*/etc/ld.so.conf* 配置文件中。

下载 *occiwrapper.tar.gz* 包，并在 *linux* 下解压，执行以下命令：

```
tar -zxvf occiwrapper.tar.gz
```

genConfigure.sh 生成可执行的权限，使用以下命令：

```
chmod +x genConfigure.sh
```

运行此命令，运行的格式如下：

```
./genConfigure.sh --occi-include=occi_include_path_value --occi-lib=occi_lib_path_value
```

其中 *occi_include_path_value* 对应着 *occi* 头文件的目录，*occi_lib_path_value* 对应着 *occi* 库文件的目录，比如头文件放在 */u01/install/oracle_client/instantclient_12_1/sdk/include/*, 库文件存放在 */u01/install/oracle_client/instantclient_12_1/*, 则执行的命令为：

```
./genConfigure.sh
--occi-include=/u01/install/oracle_client/instantclient_12_1/sdk/include/
--occi-lib=/u01/install/oracle_client/instantclient_12_1/
```

成功执行后，目录下会生成 *configure* 文件，然后执行*./configure*，默认安装在*/usr/local* 目录下，也可以自己指定安装的目录，执行：

```
./configure –prefix=/usr/local/occiwrapper
```

编译程序，执行 *make* 命令。

```
make
```

最后进行安装，执行

```
make install
```

执行成功后，在*/usr/local/occiwrapper* 目录下，存放着生成的 *include* 和 *lib* 文件，测试文件存放在 *bin* 目录下，*bin* 目录下的 *db_config.ini* 为测试库对应的配置信息。要执行 *test* 程序，需先在 *oracle* 数据库中执行 *db* 文件夹下的脚本。

3. 使用指南

本部分主要介绍调用 *occiwrapper* 组件的方法，包括以下部分：

- *oracle* 连接；
- 创建一个会话；
- 执行 *DDL* 语句；
- 执行简单插入；
- 使用绑定变量插入；
- 执行 *update* 操作；
- *Commit* 与 *Rollback*；
- 执行存储过程；
- 调用函数；
- 执行 *select* 语句并保存结果；
- 处理 *oracle* 中的日期类型
- 关于 *NULL* 值的处理；

3.1 *Oracle* 连接

occiwrapper 组件通过类 *occiwrapper::ConnectionInfo* 结构来保存 *oracle* 连接信息。

```
occiwrapper::ConnectionInfo info;
info.ip = "127.0.0.1";
info.port = 1521
info.username = "occiwrapper";
info.password = "occiwrapper";
info.sid = "orcl";
```

要建立一个 oracle 连接，需要先申请一个 *oracle* 的 *Environment* 变量，函数内部通过调用 *OCCI* 的 *createEnvironment* 函数创建 *Environment*。

```
oracle::occi::Environment::createEnvironment( oracle::occi::Environment
    ::THREADED_MUTEXED );
```

参数取值默认使用了 *THREADED_MUTEXED*，关于 *OCCI* 创建 *Environment* 的参数有如下定义：

- ***DEFAULT:*** *not thread safe, not in object mode;*
- ***THREADED_MUTEXED:*** *thread safe, mutexed internally by OCCI;*
- ***THREADED_UN-MUTEXED:*** *thread safe, client responsible for mutexing;*
- ***OBJECT:*** *uses object features*

occiwrapper 组件通过封装 *Environment*，将 *Environment* 的创建与销毁同发者隔离起来，开发者不需要关心何时去关闭 *Environment*。

通过类 *Connection* 可以方便的管理 *oracle* 的数据库连接，该类屏封了默认构造函数和拷贝构造函数。只能通过静态方法 *GetConnection* 得到一个数据库连接。

以下示例完整的给出了，如何创建一个 oracle 连接。

```
shared_ptr< occiwrapper::Environment > pEnv =
        occiwrapper::Environment::CreateEnvironment();
assert( pEnv->CreateEnvironment() != NULL );
occiwrapper::ConnectionInfo info;
info.ip = "127.0.0.1";
info.username = "occiwrapper";
info.password = "occiwrapper";
info.sid = "orcl";
assert( occiwrapper::Connection::GetConnection(
        shared_ptr< occiwrapper::Environment >( pEnv ), info ) != NULL );
```


联系作者：CUCmehp@foxmail.com
QQ 交流群：348648166

上述代码中，*shared_ptr* 为 ***C++ 0x*** 标准中提出的 ***tr1*** 库定义的智能指针，关于 ***C++ 0x*** 和 ***tr1*** 库，在此不做介绍，大家可以参见维基百科。

- 英文：http://en.wikipedia.org/wiki/C%2B%2B11
- 中文：http://zh.wikipedia.org/wiki/C%2B%2B11

occiwrapper 使用连接池对 *oracle* 数据库连接进行管理，对于相同的数据库连接进行复用；同时，通过连接池，对 *oracle* 连接进行管理，以下代码给出了如何使用 *occiwrapper* 的连接池，取得 *oracle* 连接。

```
occiwrapper::ConnectionInfo info;
occiwrapper::ConnectionPool connPool;
info.ip = "127.0.0.1";
info.username = "occiwrapper";
info.password = "occiwrapper";
info.sid = "orcl";
shared_ptr< occiwrapper::Connection > p = connPool.GetConnection( info );
assert( p != NULL);
assert( p->GetValidity() == occiwrapper::VALID );
shared_ptr< occiwrapper::Connection > other = connPool.GetConnection( info );
assert( connPool.GetConnMapSize() == 1 );
```

3.2 创建一个会话

Session 类用来管理客户端与 *oracle* 服务器之间的会话连接。通过会话连接，客户端可以灵活的创建出若干 *Statement* 来执行 *SQL* 命令。

occiwrapper 通过类 *SessionFactory* 来创建 *session* 会话，*SessionFactory* 是一个以单件形式存在的工厂类，可以动态的创建出 *Session* 对象。同时，*SessionFactory* 中内置了一个连接池对象管理 *oracle* 连接。关于连接池对象，可以参考 3.2 节创建一个会话中的介绍。

以下代码给出了如何使用 *SessionFactory* 创建一个 *Session* 对象。

```
occiwrapper::ConnectionInfo info;
occiwrapper::Session s =
        occiwrapper::SessionFactory::Instance().Create( info, false );
```

为了方使创建 Session，也可以使用宏定义。

```
occiwrapper::Session s = SessionInstance( info );
```

3.3 执行 DDL 语句

利用 *Session* 对象可以方便地执行 *DDL* 命令，如下示例演示如何创建一张表，再将它清空，最后删除该表。

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper",
        "occiwrapper", "orcl" );
occiwrapper::Session s = SessionInstance( info );
bool bRet = false;
string strErrMsg = "";
s << "create table tbl_test( x int )", now, bRet, strErrMsg;
s << "truncate table tbl_test", now, bRet, strErrMsg;
s << "drop table tbl_test", now, bRet, strErrMsg;
```

3.4 执行简单插入

对于简单的 *insert* 操作，指不使用绑定变量的 *insert* 操作。如何利用 *occiwrapper* 进行绑定变量插入，将在下两节中进行介绍。本节只介绍执行最简单的 *SQL* 语句。

例如，对于一张已知的表 *TBL_TEST1*，该表只含有一个整数字段 *X*。以下的代码数字 *10004* 插入到该表中。

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper",
        "occiwrapper", "orcl" );
occiwrapper::Session s = SessionInstance( info );
bool bRet = false;
string strErrMsg = "";
s << "insert into tbl_test1( x ) values ( 10004 )", now, bRet, strErrMsg;
```

Session 执行的结果会被保存 *bRet* 中，若执行出错，在 *strErrMsg* 中输出出错原因。

3.5 使用绑定变量进行插入

绑定变量是 *oracle* 编程中一个重要的使用技巧。通过绑定变量的使用，能够显著的提高多次执行同一条 *SQL* 语句的性能。在此不在赘述。

下面分两类进行介绍，包括绑定简单变量和绑定容器。

- 绑定简单变量

绑定简单变量，将简单数据类型变量，如 *int, float, double, string, struct tm* 等绑定到 *oracle* 的绑定变量上。

下表给出了 *occiwrapper* 支持的绑定类型

表 *1 Windows* 下 *occiwrapper* 支持的绑定类型定义

occiwrapper 类型定义	操作系统类型
Int8	signed char
UInt8	unsigned char
Int16	signed short
UInt16	unsigned short
Int32	int
UInt32	unsigned int
Int64	signed __int64
UInt64	unsigned __int64
float	float
double	double
struct tm	struct tm
std::string	std::string

表 *2* Linux 下 *occiwrapper* 支持的绑定类型定义

occiwrapper 类型定义	操作系统类型
Int8	signed char
UInt8	unsigned char
Int16	signed short
UInt16	unsigned short
Int32	int
UInt32	unsigned int
Int64	signed long
UInt64	signed long long

float	float
double	double
struct tm	struct tm
std::string	std::string

occiwrapper 通过 *use* 关键字进行简单变量绑定。如执行 *use(1)*，将数字 *1* 绑定到对应的绑定变量上; 执行 *use("hello world")*将字符串 *hello world* 绑定到对应的绑定变量上。

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper",
        "occiwrapper", "orcl" );
occiwrapper::Session s = SessionInstance( info );
bool bRet = false;
string strErrMsg = "";
s << "insert into tbl_test1( x ) values ( :1 )", use( 2 ), now, result, err_msg;
cout << "result: " << result << endl << "error message: " << err_msg << endl;
assert( result );

s << "insert into test_string( id, string_val ) values( :1, :2 )", use( 2 ),
use( "CUCMehp" ), now, result, err_msg;
cout << "result: " << result << endl << "error message: " << err_msg << endl;
assert( result );

s << "insert into test_number( id, number_value ) values( :1, :2 )", use( 1 ),
use( 3.5 ), now, result, err_msg;
cout << "result: " << result << endl << "error message: " << err_msg << endl;
assert( result );
```

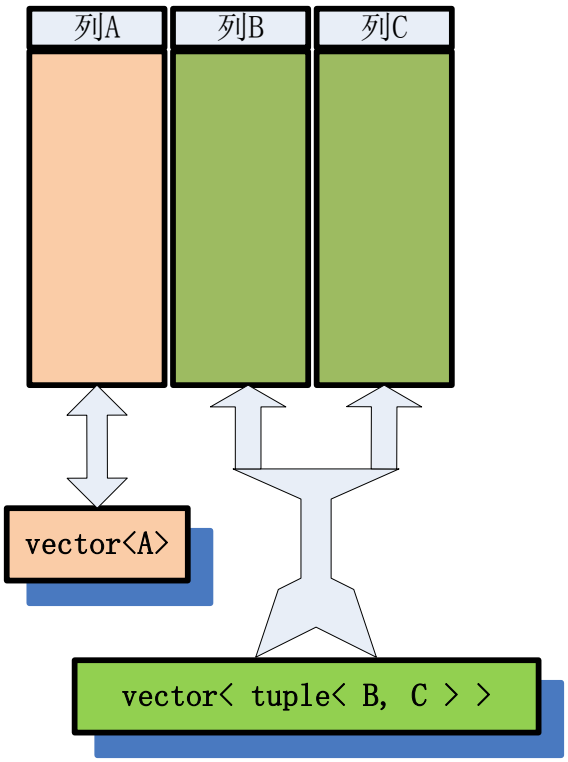
对于同一个 *Session* 也可以多次进行绑定，进行复用，以下代码向表 *TBL_TEST1* 中插入 *0* 到 *9*。

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper",
        "occiwrapper", "orcl" );
occiwrapper::Session s = SessionInstance( info );
bool bRet;
string strErrMsg;
s << "truncate table tbl_test1", now;
occiwrapper::Statement stmt = s << "insert into tbl_test1( x ) values( :1 )";
for( int i = 0; i < 10; ++ i )
{
    stmt, use( i ), now, bRet, strErrMsg;
    assert( bRet );
}
```

- 绑定容器

绑定容器是指把表中的若干列与一个或多个容器(*1.0.0* 版本仅支持 *vector* 容器)相绑定。

如下图所示，一个表有 *A*、*B*、*C* 三列，可以将列 *A* 的数据与变量 *vector<A>* 进行绑定，将列 *B*、列 *C* 的数据绑定到 *vector< tuple< B, C> >*变量中，*tuple* 为元组类型，其定义可以参见 *C++ 0x* 标准，定义在库 *tr1* 中。



利用 *occiwrapper* 的 *batched_use* 关键字可以灵活的实现上述绑定，以下示例给出了将数组*{20,21,22,23,24}*所生成的 *vector*，批量插入到表 *TBL_TEST1* 中。

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper", "occiwrapper",
        "orcl" );
occiwrapper::Session s = SessionInstance( info );
bool bRet;
string strErrMsg;
int a[5] = { 20, 21, 22, 23, 24 };
vector< int > vec( a, a + 5 );
s << "insert into tbl_test1( x ) values ( :1 )", batched_use( vec ), now, bRet, strErrMsg;
assert(bRet);
```

以下示例给出了如何绑定到两个 *vector* 或者一个 *vector<tuple>*结构中。表 *TEST_STRING* 中含有两个字段 *ID*、*STRING_VAL*，分别为 *integer* 和 *varchar2* 类型。示例中，首先将一个 *vector<int>*变量和 *vector<string>*变量绑定到这两列上，进行插入操作。然后，将一个 *vector< tuple<int,string> >*变量绑定到这两列上。

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper", "occiwrapper",
        "orcl" );
occiwrapper::Session s = SessionInstance( info );
bool bRet;
string strErrMsg;
int a[5] = { 20, 21, 22, 23, 24 };
vector< int > vec( a, a + 5 );
string strArray[5] = { "message1", "message2", "message3", "message4",
"message5" };
vector< string > vecStr( strArray, strArray + 5 );
s << "insert into test_string( id, string_val ) values ( :1, :2 )", batched_use( vec ),
batched_use( vecStr ), now, bRet, strErrMsg;
assert( bRet );
s << "truncate table test_string", now, bRet;
assert( bRet );
vector< tuple< int, string > > vecTuple;
for( size_t i = 0; i < 5; ++ i )
{
    vecTuple.push_back( make_tuple( vec[ i ], vecStr[ i ] ) );
}
s << "insert into test_string( id, string_val ) values ( :1, :2 )", batched_use( vecTuple ),
now, bRet, strErrMsg;
assert( bRet );
```

3.6 执行 *update* 操作

执行 *update* 操作的方式，跟执行 *DDL* 和 *select* 语句的方式基本相同，同样可以使用绑定变量。以下给出了简单的例子。

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper", "occiwrapper",
        "orcl" );
occiwrapper::Session s = SessionInstance( info );
bool bRet = false;
string strErrMsg = "";
s << "truncate table tbl_test1", now, bRet, strErrMsg;
s << "insert into tbl_test1( x ) values ( 10004 )", now, bRet, strErrMsg;
s << "update tbl_test1 set x = :1 where x = 10004", use( 10005 ), now, bRet, strErrMsg;
```

示例代码先向表 *TBL_TEST1* 中的 *X* 字段插入数值 *10004*，然后将数值修改为 *10005*。

3.7 Commit 与 Rollback

Occiwrapper 支持两种形式的 *commit* 操作。在创建 *Session* 时，可以指定 *Commit* 的类型，*isAutoCommit* 参数为 *true*，则表示每次操作后，自动进行 *commit*，否则需要手工 *commit* 或者 *rollback*，操作才能生效。

```
occiwrapper::Session occiwrapper::SessionFactory::Create( const ConnectionInfo&
        connInfo, bool bAutoCommit )
```

以下代码给出了示例，

```
occiwrapper::Session s = occiwrapper::SessionFactory::Instance().Create( info,
        false );
s << "truncate table tbl_test1", now;
s.Commit();
bool bRet = false;
string strErrMsg = "";
s << "insert into tbl_test1( x ) values ( 10005 )", now, bRet, strErrMsg;
s.Rollback();
occiwrapper::Statement stmt2 = ( s << "insert into tbl_test1( x ) values ( 10005 )" );
for ( int i = 0; i < 2; ++i)
{
    stmt2.execute();
}
s.Commit();
```

首先创建一个不会自动提交的 *Session* 对象，然后向表中插入数值 *10005*，但并不提交，反而撤销。此时，用 *PL/SQL* 工具查看该表，发现表为空。连续执行两次 *insert* 操作后，提交。此时，表中有两条记录。

3.8 执行存储过程

对于 *oracle* 存储过程，有输入参数与输出参数之分，输入参数是向 *oracle* 中传入的参数，而输出参数是从 *oracle* 中传出的参数。在使用 *use* 关键字是，附加 *PAR_IN* 指定参数为输入参数，附加 *PAR_OUT* 指定参数为输出参数。

在下面的例子中，存储过程 *p_test_procedure* 含有两个参数，第一个参数为 *int* 类型，第二个参数为 *varchar2*，函数实现了将 *int* 参数转化为 *string* 后，由参数 2 输出。

下面的代码块给出了上述过程。

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper",
        "occiwrapper", "orcl" );
occiwrapper::Session s = SessionInstance( info );
bool bRet;
string strErrMsg;
int nParIn = 1000;
string strParOut = "";
s << "begin p_test_procedure( :1,:2 ); end;", use( nParIn,
        occiwrapper::PAR_IN ), use( strParOut, occiwrapper::PAR_OUT ),
        now, bRet, strErrMsg;
assert( bRet );
assert( strParOut == "1000" );
```

3.9 调用函数

调用函数的方法与调用存储过程类似，只不过函数有返回值，类型为输出类型。

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper",
        "occiwrapper", "orcl" );
occiwrapper::Session s = SessionInstance( info );
bool bRet;
string strErrMsg = “”;
int a = 0;
s << "begin :1 := f_ins_tbl_test1( :2 ); end;", use( a,
        occiwrapper::PAR_OUT ), use( 2, occiwrapper::PAR_IN ) , now,
        bRet, strErrMsg;
assert( bRet );
int b;
s << "begin :1 := f_test2( :2, :3 ); end;", use( a, occiwrapper::PAR_OUT ),
        use( 2 ), use( b, occiwrapper::PAR_OUT ), now, bRet, strErrMsg;
assert( bRet );
```

3.10 执行 select 语句并保存结果

select 操作是 *SQL* 操作中最常见的操作之一，将 *oracle* 中的数据取到内存中。*occiwrapper* 通过关键字 *into* 实现将数据库中的一个字段绑定到一个容器中。

```
vector< string > vStr;
vector< struct tm > vDate;
vector< occiwrapper::Int32 > vInt;
vector< float > vFloat;
vector< double > vDouble;
s << "select string_value, date_value, int_value, float_value, number_value
        from test_batched_table", into( vStr ), into( vDate ), into( vInt ),
        into( vFloat ), into( vDouble ), now, bRet, strErrMsg;
assert( bRet );
```

上述代码段将表 *tbl_batched_table* 中的数据读取，并存到 5 组 *vector* 变量中。

occiwrapper 内部采用了批量读取的方式，提高了读取的性能，同时内部自动管理批量绑定时所需要内存空间。

当表中数据量比较大，只取若干条数据时，可以使用 Limit 关键字。

下列代码段给出了示例：

```
vector< int > vec1;
vector< int > vec2;
// test limit select
s << "select * from tbl_test2 t", into( vec1 ), into( vec2 ),
        occiwrapper::Limit( 3 ), now;
```

occiwrapper 在读取时，并没有提供游标的操作（考虑到游标会影响对于底层的封装性），直接将内存中的容器对象（*vector*）与 *oracle* 的数据表进行了绑定。因此，当数据表中的数据量很大时，*vector* 的插入受限于内存，故本组件更适应单表数据量在百万条以下的应用场景。当然，开发者为了防止插入时 vector 出错内存用尽，如将一个 *1* 亿条记录的数据，分 *100* 次插入到 *100W* 的 *vector* 容量中，每次调用批量入库，这样做也是能够成功的。然而，由于没有提供游标相关的操作，该表无法将 *1* 亿条记录一次绑定到一组 *vector* 中（会 *std bad allocate*

异常），从而形成了无法处理的窘境。因此，本组件更适合单表数据量在百万条以下规模的应用。

3.11 处理 oracle 中的日期类型

为了处理 *oracle* 中的 *Date* 类型，*occiwrapper* 对外处理的类型为 *struct tm* 类型，即 *occiwrapper* 插入和读取的类型都采用 *struct tm* 类型。

示例代码中，给出了如何向表中插入当前时间，并如何将当向时间从 *oracle* 数据库中读到一个 *vector* 数组中。

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper", "occiwrapper",
        "orcl" );
occiwrapper::Session s = SessionInstance( info );
bool bRet;
string strErrMsg;
struct tm tmValue;
time_t nowTime = time( NULL );
localtime_r( &nowTime, &tmValue);
s << "truncate table test_date", now, bRet, strErrMsg;
s << "insert into test_date( date_val ) values ( :1 )", use( tmValue ), now, bRet;
vector< struct tm > vTmDb;
s << "select date_val from test_date", into( vTmDb ), now, bRet, strErrMsg;
```

3.12 关于 NULL 值的处理

当数据库表中取在有空值时，*occiwrapper* 在向容器中插入时，用默认值进行处理。下表给出了各种类型的默认值。

occiwrapper 类型定义	默认值
Int8	*0*
UInt8	*0*
Int16	*0*
UInt16	*0*
Int32	*0*
UInt32	*0*
Int64	*0*
UInt64	*0*

float	*0*
double	*0*
struct tm	*1900-1-1 0:0:0*
std::string	“”

如果不想用上表中的默认值，则推荐在 *select* 语句中使用 *nvl* 函数进行显示指定。

3.13 批量插入

在 3.5 绑定容器一节中，利用绑定容器变量的方式，已经实现了各种数据的批量入库。

本节再给出一个含有多种类型数据批量插入的示例：

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper", "occiwrapper",
        "orcl" );
occiwrapper::Session s = SessionInstance( info );
struct tm tm_value;
time_t now_time = time( NULL );
localtime_s( &tm_value, &now_time );
vector< string > vec0;
vec0.push_back( "123456" );
vec0.push_back( "222222" );
vector< struct tm >   vec1;
vec1.push_back( tm_value );
vec1.push_back( tm_value );
vector< occiwrapper::Int8 > vec2;
vec2.push_back( 1 );
vec2.push_back( 2 );
vector< float > vec3;
vec3.push_back( 0.1 );
vec3.push_back( 0.2 );
vector< double > vec4;
vec4.push_back( 10000 );
vec4.push_back( 100 );
s << "insert into test_batched_table( string_value, date_value, int_value, float_value,
        number_value ) values ( :1, :2, :3, :4, :5 )", batched_use( vec0 ),
        batched_use( vec1 ), batched_use( vec2 ), batched_use( vec3 ),
        batched_use( vec4 ), now, bRet, strErrMsg;
assert( bRet );
```

3.14 批量读取

当一个表含有大量记录时，需要进行批量读取。Occiwrapper 提供了非常便捷的批量读取方式，可以直接把表内的结果读取到容器中，屏闭了复杂的类型绑定和内存操作过程。

在 3.10 执行 select 语句并保存结果一节中，给出了通过一条 SQL 语句，一次把表内所有的数据读取出来。

但实际上，常遇到一张表内的数据量达到百万、千万级别，一次根本读不到内存中（out of memory）。此时，就需要多次分批读取。以下给出一段示例代码。

```
occiwrapper::ConnectionInfo info( "127.0.0.1", 1521, "occiwrapper", "occiwrapper",
        "orcl" );
occiwrapper::Session s = SessionInstance( info );
struct tm tm_value;
time_t now_time = time( NULL );

vector< int > vec1;
vector< int > vec2;
// test limit select, getting all of data to the vectors
occiwrapper::Statement stmt = ( s << "select * from tbl_test2 t", into( vec1 ),
        into( vec2 ), limit( 10000 ) );
do
{
    vec1.clear();
    vec2.clear();
    stmt.Execute();
}
while( stmt.HasNext() );
```

通过执行上述代码，每次通过调用 Execute 操作，从表中取 10000 条记录到 vec1 和 vec2 两个容器中。如果不执行 vec1.clear()操作，下次调用 Execute 操作，会再往 vec1 中追加 10000 条记录。示例中，并没有对数据进行处理，因此，直接进行了清除。