轻松编写 C++ 单元测试

介绍全新单元测试框架组合： googletest 与 googlemock

googletest 与 googlemock 是 Google 公司于 2008 年发布的两套用于单元测试的应用框架，本文将向读者介绍如何应用这两套应用框架轻松编写 C++ 单元测试代码。以下讨论基于 gtest-1.2.1 及 gmock-1.0.0 。

熊伟（ Wayne Xiong ），华中科技大学硕士，曾用网名 Bill David、大卫、大笨熊等。精于 C++，后转入 JAVA 阵营，曾就职于 Lucent、BEA（ Oracle ）等公司，从事电信及 J2EE 应用平台的设计开发；现为 Adobe 公司高级软件工程师，主要从事 Flash Media Server 及 RIA 相关应用的设计开发。可以通过 billdavidcn@hotmail.com 或博客 http://blog.csdn.net/billdavid 与他联系。

2009 年 5 月 21 日

单元测试概述

测试并不只是测试工程师的责任，对于开发工程师，为了保证发布给测试环节的代码具有足够好的质量（ Quality ），为所编写的功能代码编写适量的单元测试是十分必要的。

单元测试（ Unit Test ，模块测试）是开发者编写的一小段代码，用于检验被测代码的一个很小的、很明确的功能是否正确，通过编写单元测试可以在编码阶段发现程序编码错误，甚至是程序设计错误。

单元测试不但可以增加开发者对于所完成代码的自信，同时，好的单元测试用例往往可以在回归测试的过程中，很好地保证之前所发生的修改没有破坏已有的程序逻辑。因此，单元测试不但不会成为开发者的负担，反而可以在保证开发质量的情况下，加速迭代开发的过程。

对于单元测试框架，目前最为大家所熟知的是 JUnit 及其针对各语言的衍生产品， C++ 语言所对应的 JUnit 系单元测试框架就是 CppUnit 。但是由于 CppUnit 的设计严格继承自 JUnit ，而没有充分考虑 C++ 与 Java 固有的差异（主要是由于 C++ 没有反射机制，而这是 JUnit 设计的基础），在 C++ 中使用 CppUnit 进行单元测试显得十分繁琐，这一定程度上制约了 CppUnit 的普及。笔者在这里要跟大家介绍的是一套由 google 发布的开源单元测试框架（ Testing Framework ）： googletest 。

应用 googletest 编写单元测试代码

googletest 是由 Google 公司发布，且遵循 New BSD License （可用作商业用途）的开源项目，并且 googletest 可以支持绝大多数大家所熟知的平台。与 CppUnit 不同的是： googletest 可以自动记录下所有定义好的测试，不需要用户通过列举来指明哪些测试需要运行。

定义单元测试

在应用 googletest 编写单元测试时，使用 TEST() 宏来声明测试函数。如：

清单 1. 用 TEST() 宏声明测试函数

```
TEST(GlobalConfigurationTest, configurationDataTest)
 TEST(GlobalConfigurationTest, noConfigureFileTest)
```

分别针对同一程序单元 GlobalConfiguration 声明了两个不同的测试（ Test ）函数，以分别对配置数据进行检查（ configurationDataTest ），以及测试没有配置文件的特殊情况（ noConfigureFileTest ）。

实现单元测试

针对同一程序单元设计出不同的测试场景后（即划分出不同的 Test 后），开发者就可以编写单元测试分别实现这些测试场景了。

在 googletest 中实现单元测试，可通过 ASSERT_* 和 EXPECT_* 断言来对程序运行结果进行检查。ASSERT_* 版本的断言失败时会产生致命失败，并结束当前函数； EXPECT_* 版本的断言失败时产生非致命失败，但不会中止当前函数。因此， ASSERT_* 常常被用于后续测试逻辑强制依赖的处理结果的断言，如创建对象后检查指针是否为空，若为空，则后续对象方法调用会失败；而 EXPECT_* 则用于即使失败也不会影响后续测试逻辑的处理结果的断言，如某个方法返回结果的多个属性的检查。

googletest 中定义了如下的断言：

表 1： googletest 定义的断言（ Assert ）

基本断言	二进制比较	字符串比较
ASSERT_TRUE(*condition*); EXPECT_TRUE(*condition*); *condition*为真 ASSERT_FALSE(*condition*); EXPECT_FALSE(*condition*); *condition*为假	ASSERT_EQ(*expected*,*actual*); EXPECT_EQ(*expected*,*actual*); *expected*==*actual* ASSERT_NE(*val1*,*val2*); EXPECT_NE(*val1*,*val2*); *val1*!=*val2* ASSERT_LT(*val1*,*val2*); EXPECT_LT(*val1*,*val2*); *val1*<*val2* ASSERT_LE(*val1*,*val2*); EXPECT_LE(*val1*,*val2*); *val1*<=*val2* ASSERT_GT(*val1*,*val2*); EXPECT_GT(*val1*,*val2*); *val1*>*val2* ASSERT_GE(*val1*,*val2*); EXPECT_GE(*val1*,*val2*); *val1*>=*val2*	ASSERT_STREQ(*expected_str*,*actual_str*); EXPECT_STREQ(*expected_str*,*actual_str*); 两个 C 字符串有相同的内容 ASSERT_STRNE(*str1*,*str2*); EXPECT_STRNE(*str1*,*str2*); 两个 C 字符串有不同的内容 ASSERT_STRCASEEQ(*expected_str*,*actual_str*); EXPECT_STRCASEEQ(*expected_str*,*actual_str*); 两个 C 字符串有相同的内容，忽略大小写 ASSERT_STRCASENE(*str1*,*str2*); EXPECT_STRCASENE(*str1*,*str2*); 两个 C 字符串有不同的内容，忽略大小写

下面的实例演示了上面部分断言的使用：

清单 2. 一个较完整的 googletest 单元测试实例

```
// Configure.h
 #pragma once

 #include <string>
 #include <vector>

 class Configure
 {
 private:
    std::vector<std::string> vItems;

 public:
    int addItem(std::string str);

    std::string getItem(int index);

    int getSize();
 };

 // Configure.cpp
 #include "Configure.h"

 #include <algorithm>

 /**
 * @brief Add an item to configuration store. Duplicate item will be ignored
 * @param str item to be stored
 * @return the index of added configuration item
 */
 int Configure::addItem(std::string str)
 {
```

```
std::vector<std::string>::const_iterator vi=std::find(vItems.begin(), vItems.end(), str);
    if (vi != vItems.end())
        return vi - vItems.begin();

    vItems.push_back(str);
    return vItems.size() - 1;
 }

 /**
 * @brief Return the configure item at specified index.
 * If the index is out of range, "" will be returned
 * @param index the index of item
 * @return the item at specified index
 */
 std::string Configure::getItem(int index)
 {
    if (index >= vItems.size())
        return "";
    else
        return vItems.at(index);
 }

 /// Retrieve the information about how many configuration items we have had
 int Configure::getSize()
 {
    return vItems.size();
 }

 // ConfigureTest.cpp
 #include <gtest/gtest.h>

 #include "Configure.h"

 TEST(ConfigureTest, addItem)
 {
    // do some initialization
    Configure* pc = new Configure();

    // validate the pointer is not null
    ASSERT_TRUE(pc != NULL);

    // call the method we want to test
    pc->addItem("A");
    pc->addItem("B");
    pc->addItem("A");

    // validate the result after operation
    EXPECT_EQ(pc->getSize(), 2);
    EXPECT_STREQ(pc->getItem(0).c_str(), "A");
    EXPECT_STREQ(pc->getItem(1).c_str(), "B");
    EXPECT_STREQ(pc->getItem(10).c_str(), "");

    delete pc;
 }
```

运行单元测试

在实现完单元测试的测试逻辑后，可以通过 RUN_ALL_TESTS() 来运行它们，如果所有测试成功，该函数返回 0，否则会返回 1 。 RUN_ALL_TESTS() 会运行你链接到的所有测试——它们可以来自不同的测试案例，甚至是来自不同的文件。

因此，运行 googletest 编写的单元测试的一种比较简单可行的方法是：

- 为每一个被测试的 class 分别创建一个测试文件，并在该文件中编写针对这一 class 的单元测试；
- 编写一个 Main.cpp 文件，并在其中包含以下代码，以运行所有单元测试：

清单 3. 初始化 googletest 并运行所有测试

```
#include <gtest/gtest.h>

 int main(int argc, char** argv) {
    testing::InitGoogleTest(&argc, argv);

    // Runs all tests using Google Test.
    return RUN_ALL_TESTS();
 }
```

- 最后，将所有测试代码及 Main.cpp 编译并链接到目标程序中。

此外，在运行可执行目标程序时，可以使用 --gtest_filter 来指定要执行的测试用例，如：

- ./foo_test 没有指定filter，运行所有测试；
- ./foo_test --gtest_filter=* 指定filter为*，运行所有测试；
- ./foo_test --gtest_filter=FooTest.* 运行测试用例FooTest的所有测试；
- ./foo_test --gtest_filter=*Null*:*Constructor* 运行所有全名（即测试用例名 + “ . ” + 测试名，如 GlobalConfigurationTest.noConfigureFileTest）含有"Null"或"Constructor"的测试；
- ./foo_test --gtest_filter=FooTest.*-FooTest.Bar 运行测试用例FooTest的所有测试，但不包括 FooTest.Bar。

这一特性在包含大量测试用例的项目中会十分有用。

应用 googlemock 编写 Mock Objects

很多 C++ 程序员对于 Mock Objects （模拟对象）可能比较陌生，模拟对象主要用于模拟整个应用程序的一部分。在单元测试用例编写过程中，常常需要编写模拟对象来隔离被测试单元的“下游”或“上游”程序逻辑或环境，从而达到对需要测试的部分进行隔离测试的目的。

例如，要对一个使用数据库的对象进行单元测试，安装、配置、启动数据库、运行测试，然后再卸装数据库的方式，不但很麻烦，过于耗时，而且容易由于环境因素造成测试失败，达不到单元测试的目的。模仿对象提供了解决这一问题的方法：模仿对象符合实际对象的接口，但只包含用来“欺骗”测试对象并跟踪其行为的必要代码。因此，其实现往往比实际实现类简单很多。

为了配合单元测试中对 Mocking Framework 的需要， Google 开发并于 2008 年底开放了： googlemock 。与 googletest 一样， googlemock 也是遵循 New BSD License （可用作商业用途）的开源项目，并且 googlemock 也可以支持绝大多数大家所熟知的平台。

注 1：在 Windows 平台上编译 googlemock

对于 Linux 平台开发者而言，编译 googlemock 可能不会遇到什么麻烦；但是对于 Windows 平台的开发者，由于 Visual Studio 还没有提供 tuple （ C++0x TR1 中新增的数据类型）的实现，编译 googlemock 需要为其指定一个 tuple 类型的实现。著名的开源 C++ 程序库 boost 已经提供了 tr1 的实现，因此，在 Windows 平台下可以使用 boost 来编译 googlemock 。为此，需要修改 %GMOCK_DIR%/msvc/gmock_config.vsprops ，设定其中 BoostDir 到 boost 所在的目录，如：

```
<UserMacro
    Name="BoostDir"
    Value="$(BOOST_DIR)"
 />
```

其中 BOOST_DIR 是一个环境变量，其值为 boost 库解压后所在目录。

对于不希望在自己的开发环境上解包 boost 库的开发者，在 googlemock 的网站上还提供了一个从 boost 库中单独提取出来的 tr1 的实现，可将其下载后将解压目录下的 boost 目录拷贝到 %GMOCK_DIR% 下（这种情况下，请勿修改上面的配置项；建议对 boost 不甚了解的开发者采用后面这种方式）。

在应用 googlemock 来编写 Mock 类辅助单元测试时，需要：

- 编写一个 Mock Class （如 class MockTurtle ），派生自待 Mock 的抽象类（如 class Turtle ）；
- 对于原抽象类中各待 Mock 的 virtual 方法，计算出其参数个数 n ；

- 在 Mock Class 类中，使用 MOCK_METHODn() （对于 const 方法则需用 MOCK_CONST_METHODn() ）宏来声明相应的 Mock 方法，其中第一个参数为待 Mock 方法的方法名，第二个参数为待 Mock 方法的类型。如下：

清单 4. 使用 MOCK_METHODn 声明 Mock 方法

```
#include <gmock/gmock.h>  // Brings in Google Mock.

 class MockTurtle : public Turtle {
    MOCK_METHOD0(PenUp, void());
    MOCK_METHOD0(PenDown, void());
    MOCK_METHOD1(Forward, void(int distance));
    MOCK_METHOD1(Turn, void(int degrees));
    MOCK_METHOD2(GoTo, void(int x, int y));
    MOCK_CONST_METHOD0(GetX, int());
    MOCK_CONST_METHOD0(GetY, int());
 };
```

- 在完成上述工作后，就可以开始编写相应的单元测试用例了。在编写单元测试时，可通过 ON_CALL 宏来指定 Mock 方法被调用时的行为，或 EXPECT_CALL 宏来指定 Mock 方法被调用的次数、被调用时需执行的操作等，并对执行结果进行检查。如下：

清单 5. 使用 ON_CALL 及 EXPECT_CALL 宏

```
using testing::Return;                              // #1，必要的声明

 TEST(BarTest, DoesThis) {
    MockFoo foo;                                    // #2，创建 Mock 对象

    ON_CALL(foo, GetSize())                         // #3，设定 Mock 对象默认的行为（可选）
        .WillByDefault(Return(1));
    // ... other default actions ...

    EXPECT_CALL(foo, Describe(5))                   // #4，设定期望对象被访问的方式及其响应
        .Times(3)
        .WillRepeatedly(Return("Category 5"));
    // ... other expectations ...

    EXPECT_EQ("good", MyProductionFunction(&foo));
    // #5，操作 Mock 对象并使用 googletest 提供的断言验证处理结果
 }
 // #6，当 Mock 对象被析构时， googlemock 会对结果进行验证以判断其行为是否与所有设定的预期一致
```

其中， WillByDefault 用于指定 Mock 方法被调用时的默认行为； Return 用于指定方法被调用时的返回值；Times 用于指定方法被调用的次数； WillRepeatedly 用于指定方法被调用时重复的行为。

对于未通过 EXPECT_CALL 声明而被调用的方法，或不满足 EXPECT_CALL 设定条件的 Mock 方法调用，googlemock 会输出警告信息。对于前一种情况下的警告信息，如果开发者并不关心这些信息，可以使用 Adapter 类模板 NiceMock 避免收到这一类警告信息。如下：

清单 6. 使用 NiceMock 模板

```
testing::NiceMock<MockFoo> nice_foo;
```

在笔者开发的应用中，被测试单元会通过初始化时传入的上层应用的接口指针，产生大量的处理成功或者失败的消息给上层应用，而开发者在编写单元测试时并不关心这些消息的内容，通过使用 NiceMock 可以避免为不关心的方法编写 Mock 代码（注意：这些方法仍需在 Mock 类中声明，否则 Mock 类会被当作 abstract class 而无法实例化）。

与 googletest 一样，在编写完单元测试后，也需要编写一个如下的入口函数来执行所有的测试：

清单 7. 初始化 googlemock 并运行所有测试

```
#include <gtest/gtest.h>
 #include <gmock/gmock.h>

 int main(int argc, char** argv) {
    testing::InitGoogleMock(&argc, argv);

    // Runs all tests using Google Test.
    return RUN_ALL_TESTS();
 }
```

下面的代码演示了如何使用 googlemock 来创建 Mock Objects 并设定其行为，从而达到对核心类 AccountService 的 transfer （转账）方法进行单元测试的目的。由于 AccountManager 类的具体实现涉及数据库等复杂的外部环境，不便直接使用，因此，在编写单元测试时，我们用 MockAccountManager 替换了具体的 AccountManager 实现。

清单 8. 待测试的程序逻辑

```
// Account.h
 // basic application data class
 #pragma once

 #include <string>

 class Account
 {
 private:
    std::string accountId;

    long balance;

 public:
    Account();

    Account(const std::string& accountId, long initialBalance);

    void debit(long amount);

    void credit(long amount);

    long getBalance() const;

    std::string getAccountId() const;
 };

 // Account.cpp
 #include "Account.h"

 Account::Account()
 {
 }

 Account::Account(const std::string& accountId, long initialBalance)
 {
    this->accountId = accountId;
    this->balance = initialBalance;
 }

 void Account::debit(long amount)
 {
    this->balance -= amount;
 }

 void Account::credit(long amount)
 {
    this->balance += amount;
 }
```

```
long Account::getBalance() const
{
   return this->balance;
}

std::string Account::getAccountId() const
{
   return accountId;
}

// AccountManager.h
// the interface of external services which should be mocked
#pragma once

#include <string>

#include "Account.h"

class AccountManager
{
public:
   virtual Account findAccountForUser(const std::string& userId) = 0;

   virtual void updateAccount(const Account& account) = 0;
};

// AccountService.h
// the class to be tested
#pragma once

#include <string>

#include "Account.h"
#include "AccountManager.h"

class AccountService
{
private:
   AccountManager* pAccountManager;

public:
   AccountService();

   void setAccountManager(AccountManager* pManager);
   void transfer(const std::string& senderId,
              const std::string& beneficiaryId, long amount);
};

// AccountService.cpp
#include "AccountService.h"

AccountService::AccountService()
{
   this->pAccountManager = NULL;
}

void AccountService::setAccountManager(AccountManager* pManager)
{
   this->pAccountManager = pManager;
}

void AccountService::transfer(const std::string& senderId,
                  const std::string& beneficiaryId, long amount)
{
   Account sender = this->pAccountManager->findAccountForUser(senderId);

   Account beneficiary = this->pAccountManager->findAccountForUser(beneficiaryId);

   sender.debit(amount);

   beneficiary.credit(amount);
```

```
   this->pAccountManager->updateAccount(sender);

   this->pAccountManager->updateAccount(beneficiary);
}
```

清单 9. 相应的单元测试

```
// AccountServiceTest.cpp
 // code to test AccountService
 #include <map>
 #include <string>

 #include <gtest/gtest.h>
 #include <gmock/gmock.h>

 #include "../Account.h"
 #include "../AccountService.h"
 #include "../AccountManager.h"

 // MockAccountManager, mock AccountManager with googlemock
 class MockAccountManager : public AccountManager
 {
 public:
    MOCK_METHOD1(findAccountForUser, Account(const std::string&));

    MOCK_METHOD1(updateAccount, void(const Account&));
 };

 // A facility class acts as an external DB
 class AccountHelper
 {
 private:
    std::map<std::string, Account> mAccount;
             // an internal map to store all Accounts for test

 public:
    AccountHelper(std::map<std::string, Account>& mAccount);

    void updateAccount(const Account& account);

    Account findAccountForUser(const std::string& userId);
 };

 AccountHelper::AccountHelper(std::map<std::string, Account>& mAccount)
 {
    this->mAccount = mAccount;
 }

 void AccountHelper::updateAccount(const Account& account)
 {
    this->mAccount[account.getAccountId()] = account;
 }

 Account AccountHelper::findAccountForUser(const std::string& userId)
 {
    if (this->mAccount.find(userId) != this->mAccount.end())
        return this->mAccount[userId];
    else
        return Account();
 }

 // Test case to test AccountService
 TEST(AccountServiceTest, transferTest)
 {
    std::map<std::string, Account> mAccount;
    mAccount["A"] = Account("A", 3000);
    mAccount["B"] = Account("B", 2000);
    AccountHelper helper(mAccount);

    MockAccountManager* pManager = new MockAccountManager();
```

```
    // specify the behavior of MockAccountManager
    // always invoke AccountHelper::findAccountForUser
     // when AccountManager::findAccountForUser is invoked
    EXPECT_CALL(*pManager, findAccountForUser(testing::_)).WillRepeatedly(
        testing::Invoke(&helper, &AccountHelper::findAccountForUser));

    // always invoke AccountHelper::updateAccount
    //when AccountManager::updateAccount is invoked
    EXPECT_CALL(*pManager, updateAccount(testing::_)).WillRepeatedly(
        testing::Invoke(&helper, &AccountHelper::updateAccount));

    AccountService as;
    // inject the MockAccountManager object into AccountService
    as.setAccountManager(pManager);

    // operate AccountService
    as.transfer("A", "B", 1005);

    // check the balance of Account("A") and Account("B") to
    //verify that AccountService has done the right job
    EXPECT_EQ(1995, helper.findAccountForUser("A").getBalance());
    EXPECT_EQ(3005, helper.findAccountForUser("B").getBalance());

    delete pManager;
}

// Main.cpp
#include <gtest/gtest.h>
#include <gmock/gmock.h>

int main(int argc, char** argv) {
    testing::InitGoogleMock(&argc, argv);

    // Runs all tests using Google Test.
    return RUN_ALL_TESTS();
}
```

注 2：上述范例工程详见附件。要编译该工程，请读者自行添加环境变量 GTEST_DIR 、 GMOCK_DIR ，分别指向 googletest 、 googlemock 解压后所在目录；对于 Windows 开发者，还需要将 %GMOCK_DIR%/msvc/gmock_config.vsprops 通过 View->Property Manager 添加到工程中，并将 gmock.lib 拷贝到工程目录下。

通过上面的实例可以看出， googlemock 为开发者设定 Mock 类行为，跟踪程序运行过程及结果，提供了丰富的支持。但与此同时，应用程序也应该尽量降低应用代码间的耦合度，使得单元测试可以很容易对被测试单元进行隔离（如上例中， AccountService 必须提供了相应的方法以支持 AccountManager 的替换）。关于如何通过应用设计模式来降低应用代码间的耦合度，从而编写出易于单元测试的代码，请参考本人的另一篇文章《应用设计模式编写易于单元测试的代码》（ developerWorks ， 2008 年 7 月）。

注 3：此外，开发者也可以直接通过继承被测试类，修改与外围环境相关的方法的实现，达到对其核心方法进行单元测试的目的。但由于这种方法直接改变了被测试类的行为，同时，对被测试类自身的结构有一些要求，因此，适用范围比较小，笔者也并不推荐采用这种原始的 Mock 方式来进行单元测试。

总结

Googletest 与 googlemock 的组合，很大程度上简化了开发者进行 C++ 应用程序单元测试的编码工作，使得单元测试对于 C++ 开发者也可以变得十分轻松；同时， googletest 及 googlemock 目前仍在不断改进中，相信随着其不断发展，这一 C++ 单元测试的全新组合将变得越来越成熟、越来越强大，也越来越易用。

下载

描述	名字	大小
完整的使用 googletest 及 googlemock 编写单元测试的例子	AccountService.tar.gz	4KB
一个较完整的使用 googletest 编写单元测试的例子	ConfigureTest.tar.gz	3KB

参考资料

- “使用模仿对象进行单元测试”（ developerWorks ，2003 年 3 月）：介绍如何使用模仿对象替换合作者以改进单元测试。
- “应用设计模式编写易于单元测试的代码”（ developerWorks ，2008 年 7 月）：介绍如何应用设计模式编写易于单元测试的代码。
- “用 Mock Object 进行独立单元测试”：介绍如何应用 jMock ， EasyMock 对单个的类进行隔离测试。
- “MocksAren ’ t Stubs”（ Martin Fowler ，2007 年 6 月）：介绍 Mock Objects 概念及其与传统 Stubs 测试方式的区别。
- 关于 googletest 的更多信息，请访问其项目主页：http://code.google.com/p/googletest/
- 关于 googlemock 的更多信息，请访问其项目主页：http://code.google.com/p/googlemock/