教你轻松学会单元测试

→李军亮(lijunliang@baidu.com)

目录

1 基本概念

- **单元测试**：最低级别的测试，测试对象为函数或类，是开发者的行为。
- **模块测试**：功能级别的测试，测试对象为可执行程序。
- **联调测试**：接口级别的测试，测试对象为模块程序之间的接口。
- **系统测试**：系统级别的测试，测试对象为多模块组成的一个系统。
  如性能测试，大数据对比，压力测试都属于系统测试。

2 单测意义

- **开发收益**：保证代码整体结构良好，使其有很好的可读性，可维护性，可测性。
- **质量收益**：提高代码内部质量，可覆盖黑盒不可见功能点，避免给后人埋下坑。
- **效率收益**：缩短项目周期，使 bug 暴露在开发阶段，减少问题调试和定位代价。

3 单测范围

单元测试是开发阶段的测试，主要是针对代码内部函数功能而进行的。一般除如下所述的几方面函数外，其它函数均应有相应的单测用例来保证其正确性。

- 接口函数：负责与上下游交互。
- 流程性函数：负责一一调用其它函数。
- 框架调度函数：负责调度模块内部各功能子模块(mods)。

4 单测用例开发原则

在开发单元测试用例的时候，需要遵循如下几个原则：

- **四步曲**：构造数据、执行被测函数、校验结果、清理数据。
- **运行快**：一般要求 UT 用例平均运行时间在毫秒级。
- **单一性**：一个 UT 用例只负责一个场景/行为的测试。
- **独立性**：用例之间无耦合，执行顺序不同，但结果相同。
- **稳定性**：避免随机因素，UT 用例在多次运行时，结果不变。
- **准入性**：编译阶段自动运行，若有一个 UT 用例 fail，则视为编译失败。
- **自描述**：UT 用例易读，易理解，与产品代码一样，要求良好的可读性。

5 教你在新模块中开展单测

本文基于 comake2/Btest，以简单的实例，介绍一个新模块如何开始开发和单测。假设我们需要开发的新模块信息如下所述。（**如何为老模块引入单测，亦可参考下文所述**）

svn 路径：app/ecom/im/sample

编译依赖：public/odict@odict_1-1-2-1_PD_BL，lib2-64/wordseg@wordseg_1-2-5-1_PD_BL

头文件名：sample.h，adapter.h

源文件名：sample.cpp，adapter.cpp

在正式开发新模块前，需要在开发机上先安装 comake2 和 Btest。

5.1 安装 comake2

comake2 是一种通过编写 COMAKE 文件，来帮助用户管理编译依赖以及编译环境的开发工具，是目前 scm 主推的一种代码编译管理工具。新模块基本都是用 comake2 进行开发，很多老模块也慢慢地切到 comake2 管理。

- comake2 能够帮助用户生成较可读可调试的 Makefile
- comake2 能够自动帮助用户搭建编译环境
- comake2 能够允许用户实施持续集成开发

WIKI：http://wiki.babel.baidu.com/twiki/bin/view/Com/Main/Comake2#comake2 使用说明

comake2 属于 scmtools 工具集，安装 scmtools 即可。安装步骤如下：

（通过执行 which comake2，若查找成功，则可不必再安装。scmtools 会自动进行升级。）

ROOT 用户安装 scmtools，这样该开发机其他用户亦可使用。**[推荐]** scp getprod@product.scm.baidu.com:/data/prod-64/scm/compile-optim/latest/client/usr/bin/install.sh . 输入密码：getprod sh install.sh /home/scmtools echo "PATH=/home/scmtools/usr/bin:\${PATH}" >> /etc/profile.d/scmtools.sh
普通用户安装 scmtools。 scp getprod@product.scm.baidu.com:/data/prod-64/scm/compile-optim/latest/client/usr/bin/install.sh . 输入密码：getprod sh install.sh /home/$USER/scmtools echo "PATH=/home/$USER/scmtools/usr/bin:\${PATH}" >> /home/$USER/.bash_profile

按照上述步骤操作后，开发用户重新登陆或 source ~/ .bash_profile 后，通过执行 which comake2，若输出结果类似为/home/scmtools/usr/bin/comake2，便表示安装成功。

5.2 安装 Btest

Btest 是基于 google test(简称 Gtest)的二次开发，已被广泛应用在公司各 C/C++产品模块的单元测试上。其需要先安装 svn，再安装 Btest，具体步骤如下：

- **安装 svn**

a） 确认 svn 是否存在，直接输入命令 svn，若存在该命令，则跳过安装。

b） 按照步骤安装 svn，安装步骤参见：

http://wiki.babel.baidu.com/twiki/bin/view/Com/Test/Auction-svn-readme

c） 确认 svn 安装成功

- **安装 Btest**

a） 确认 Btest 是否存在，直接输入命令 autorun.sh，若存在该命令，则跳过安装。

b） 按照步骤安装 Btest，安装步骤参见：

http://wiki.babel.baidu.com/twiki/bin/view/Com/Test/Auction-btest-readme

c） 确认 Btest 安装成功

5.3 建立代码组织结构

```
mkdir –p sample/app/ecom/im/sample
cd sample/app/ecom/im/sample
mkdir make                  ##make 用于存放 COMAKE 文件和自动生成的 Makefile
mkdir include               ##include 用于存在头文件 sample.h，adapater.h
mkdir src                   ##src 用于存放源文件 sample.cpp，adapter.cpp
mkdir unittest              ##unitest 用于存放 UT 用例 test_sample.cpp，test_adapter.cpp
comake2 -C make/ -S         ##生成 COMAKE 文件模板，若 make 下已有 COMAKE，则忽略
touch include/sample.h include/adapter.h
touch src/sample.cpp src/adapter.cpp
touch unittest/test_sample.cpp unittest/test_adapter.cpp
```

5.4 撰写 COMAKE 文件

```
#edit-mode: -*- python -*-
#coding:gbk
#工作路径.
WORKROOT('../../../../../')
#使用硬链接 copy.
CopyUsingHardLink(True)
#C 预处理器参数.
CPPFLAGS('-D_GNU_SOURCE -D__STDC_LIMIT_MACROS -DVERSION=\\\"1.0.0.0\\\" -D__INLINE__="inline"')
#C 编译参数.
CFLAGS('-g -pipe -W -Wall -fPIC')
#C++编译参数.
CXXFLAGS('-g -pipe -W -Wall -fPIC')
#IDL 编译参数
IDLFLAGS('--compack')
#UBRPC 编译参数
UBRPCFLAGS('--compack')
#头文件路径.
INCPATHS('.. ../include ../output ../output/include')
#链接参数.
LDFLAGS('-lpthread -lcrypto -lrt')
#依赖模块
CONFIGS('public/odict@odict_1-1-2-1_PD_BL')
CONFIGS('lib2-64/wordseg@wordseg_1-2-5-1_PD_BL')
```

```
CONFIGS('third-64/gtest@gtest_1-4-0-500_PD_BL') ##gtest 是 UT 调度框架，提供了基础校验接口，如 EXPECT_EQ
CONFIGS('app-test/ecom/im/common/utlib@trunk')##utlib 是一个构造数据，校验结果的公共库
user_sources=GLOB('../src/*.cpp')
user_headers=GLOB('../include/*.h')
#可执行文件
Application('sample', Sources(user_sources), OutputPath('./output/bin/'))
#静态库,用于 UT 用例联编
StaticLibrary('utsample', Sources(user_sources,
                CppFlags('-D_UNIT_TEST_ -Dprivate=public -Dprotected=public')),HeaderFiles(user_headers))
#编译 UT 用例
ut_sources=glob.glob('../unittest/test_*.cpp')
import string
ut_target_list = []
for test in ut_sources:
        ut_target = os.path.splitext(test)[0]
        Application(ut_target,Sources(test,
        CppFlags('-D__INLINE__="" -Dprivate=public -Dprotected=public -D__64BIT__')),
        HeaderFiles(user_headers),
        Libraries('./libutsample.a','../../../../../app-test/ecom/im/common/utlib/output/lib/hookmon.so'),
        LinkFlags('-lpthread -lcrypto -lrt'))
        ut_target_list.append(ut_target)
TARGET('ut',PhonyMode(True),Prefixes(string.join(ut_target_list)),ShellCommands('autorun.sh -u -p "../unittest/*"'))
```

5.5 搭建编译开发环境

```
comake2 -C make -U    ##下载依赖代码
comake2 -C make -B    ##编译这些依赖代码
comake2 -C make -P    ##生成 Makefile
```

5.6 开发产品代码

- **adapter.h**

```
#ifndef __ADAPTER_H__
#define __ADAPTER_H__
#define MAX_Q_NUM 1024
struct queue{
        unsigned int num;
        int weight[MAX_Q_NUM];
};
class Adapter{
public:
        virtual int GetWeight(const char* query, queue* q);
};
#endif
```

- **adapter.cpp**

```
#include "adapter.h"
int Adapter::GetWeight(const char* query, queue* q)
{
        //根据 query 查询出其 q 个数和对应 weight，这里实现省略，用如下代码代替。
        q->num = 2;
        q->weight[0] = 100;
        q->weight[1] = 200;
        return q->num;
```

```
}
```

- **sample.h**

```
#ifndef __SAMPLE_H__
#define __SAMPLE_H__
#include "stdio.h"
#include "adapter.h"
class Sample{
public:
	int GetAverage(const char* query);
private:
	Adapter *ada;
};
#endif
```

- **sample.cpp**

```
#include "sample.h"
int Sample::GetAverage(const char* query)
{
	queue q;
	if(ada->GetWeight(query, &q))
	{
		int sum = 0;
		for(int i= 0; i < q.num; i++)
		{
			sum += q.weight[i];
		}
		return sum/q.num;
	}
	return -1;
}
#ifdef _UNIT_TEST_ //编译 UT 用例时，走 ut_main，避免出现 UT 用例编译时出现两个 main 函数。
int ut_main() //该做法会修改产品代码，一般不建议采用。建议将 main 函数放到一个独立文件中。
#else
int main()
#endif
{
	printf("This is a sample.\n");
	return 0;
}
```

5.7 撰写 UT 用例

- **test_sample.cpp**

```
#include "sample.h" //包含被测函数或类的头文件
#include "gtest/gtest.h"
int main(int argc, char **argv)
{
	testing::InitGoogleTest(&argc, argv);
	return RUN_ALL_TESTS(); //顺序执行下述所有以 TEST/TEST_F/TEST_P 开头的 UT 用例
}
class test_sample_suite : public ::testing::Test{
protected:
virtual void SetUp() //test_sample_suite 每个 case 运行前运行，用于统一初始化构造数据。
	{
		testobj = new Sample();
```

```
        testobj->ada = new Adapter();
    };
    virtual void TearDown()//test_sample_suite 每个 case 运行结束后运行，用于统一清理数据。
    {
        delete testobj->ada;
        delete testobj;
    };
    Sample *testobj;
};
TEST_F(test_sample_suite, test_GetAverage)
{
    ASSERT_EQ(150, testobj->GetAverage("XIANHUA"));//执行被测函数，同时校验结果
}
```

5.8 编译并运行 UT

```
make -C make -s        ##编译产品代码和 UT 用例，并运行 UT 用例
cd unittest
autorun.sh -u          ##运行当前目录下所有的 UT 用例
./test_sample          ##单独运行 test_sample 用例
./test_sample --gtest_filter="test_sample_suite.test_GetAverage" ##选择运行某个具体用例
./test_sample --gtest_filter="test_sample_suite.* " ##选择运行某批具体用例
```

6 公共接口

6.1 结果断言

Gtest 提供了很多结果校验的宏接口，断言的宏可以分为两类，一类是 ASSERT 系列，一类是 EXPECT 系列。一个直观的解释就是：

- **ASSERT_*：**当检查点失败时，退出当前函数（注意：并非退出当前案例）。
- **EXPECT_*：**当检查点失败时，继续往下执行。

如常用的宏有：EXPECT_EQ，EXPECT_STREQ，EXPECT_TRUE，EXPECT_FALSE，ASSERT_EQ。

更详细的宏接口介绍：http://www.cnblogs.com/yylqinghao/archive/2010/04/15/1712837.html

6.2 用例宏函数

UT 用例开头的宏接口有三种：**TEST，TEST_F 和 TEST_P**，其都能被 RUN_ALL_TESTS()函数自动调度到。

- TEST(case_name，case_name)

  这是最简单的 UT 用例开头，其两个参数仅仅表示用例名字。

- TEST_F(Testsuite，case_name)

  当一批用例存在很大的相似性时，可通过定义一个 Testsuite 类，其中实现公共的

数据构造，清理数据的功能，如常见的 SetUp()和 TearDown()函数。

- TEST_P(class_name，case_name)

  用于参数化测试。详见：http://www.cnblogs.com/yylqinghao/archive/2010/04/15/1712843.html

6.3 事件触发机制

Gtest 提供了三种不同级别的事件触发机制：全局事件、TestSuite 事件和 TestCase 事件

- **全局事件**：全局的，所有案例执行前后。
- **TestSuite 事件**：TestSuite 级别，在某一批案例第一个案例前，最后一个案例执行后。
- **TestCase 事件**：TestCase 级别的，每个 TestCase 前后。

  详见：http://www.cnblogs.com/coderzh/archive/2009/04/06/1430396.html

6.4 自定义接口

除了 Gtest 已有的宏接口外，往往我们需要根据本模块的特点自定义一些公共的结果校验和数据构造接口，最好形成 LIB，便于所有的 UT 用例调用，也方便其它模块的单测时使用。如 app-test/ecom/im/common/utlib 就是我们 IM 组自定义的一些常用单测接口。下面简单介绍下其中接口使用方法。

- **ut_add_file**：用于创建某文件。
- **ut_touch_file**：用于改变文件时间戳，便于重载。任何时候调用，时间戳必然改变。

  (间隔>1s，touch 才可以改变时间戳，故在 UT 中不适合用 touch 命令)
- **ut_del_file**：用于删除某文件。
- **ut_get_free_port**：用于获得一个未被占用的端口号。
- **odict_odb_add**：插入一个 node 到指定字典中。
- **UT_NODE_EXPECT**：用于校验字典中某个 KEY 是否存在。
- **UT_VALUE_EXPECT**：用于校验字典中某个 KEY 对应的 VALUE 正确性。

  也用于校验切词 token 字面正确性。
- **ut_open_log**：用于打开日志文件，默认是名字 ut.log 和 ut.log.wf，日志级别=16。

具体使用方法见 app-test/ecom/im/common/utli/include/ut_common.h。附简单实例如下。

```
int main(int argc, char **argv)
{
    ut_open_log(); //打开日志文件，将用例中间输出的 log 定向到 ut.log*，避免输出到终端上。
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
TEST(test_route_handle_t, test_reload_success)
{
    route_handle_t hdl;
    const char* content="
        [@communication]\n\
```

```
            communication_name : imas2imbs\n\
            default_path : imbs0”
    ut_add_file(content, “./conf/ubc.conf”);
    ASSERT_EQ(0, hdl.init("./conf", "ubc.conf", "./", mapping_file));
    EXPECT_EQ(0u,   hdl._mapp._routes_idx);
    ut_touch_file(mapping_file);                    //改变文件时间戳
    ASSERT_EQ(0, hdl.reload("./", mapping_file));//开始重载
    EXPECT_EQ(1u,   hdl._mapp._routes_idx);
    time_t t3 = hdl._mapp._last_routes_mtime;
    EXPECT_NE(t2, t3);
    ut_del_file(“./conf/ubc.conf”)
}
```

```
sodict_build_t * data_dict;
odict_odb_add(data_dict, 111, 222, 333, 444);
UT_NODE_EXPECT(data_dict, 111, 222, ODB_SEEK_OK);
UT_VALUE_EXPECT(data_dict, 111, 222, 333, 444);
token_t *seg_tok
UT_VALUE_EXPECT(seg_tok, 4, "中华", "人民", "共和", "国");
```

7 单测技术

7.1 如何测试 private 函数

方法 1[推荐]	UT Makefile 中定义	-Dprivate=public -Dprotected=public
方法 2	UT 用例 test_***.cpp 中定义	#define private public #define protected public

7.2 如何管理 UT 中的大数据

一般单元测试中是不会出现大数据的，但当确实用到了，也是不建议跟代码一起 CI 到代码库中。因为这会增加代码量，加重代码库的负担。如 UT 用例需要加载 worddict，一般做法是在 UT 用例运行前，先判断本地是否有 worddict，若无则从稳定的提供方去自动下载 ftp://getprod:getprod@product.scm.baidu.com:/data/prod-32/ps/se/worddict/worddict_1-2-9_BL/output/bin data/worddict。这里请注重两个原则：

- **动态性**：要求运行前动态获取，避免对本地运行环境的依赖。
- **稳定性**：要求提供方足够稳定，若使用 wget 时，记得加上重试参数-t。

7.3 MOCK 技术

如下一段代码，当需要单测函数 GetAverage ()，而又不想关心 GetWeight 内部具体实现时，我们就需要通过一个简单的函数 m_GetWeight 去 mock 掉 GetWeight，让 UT 用例在执行 GetAverage 函数的时候，调用自己写的 m_GetWeight 函数。如此该 UT 用例就只跟 GetAverage 函数有关，跟 GetWeight 函数无关。GetWeight 函数自然由另外的 UT 用例去覆

盖，如此不仅使得 GetAverage 函数更易单测，也保证了 UT 用例的稳定性。

```
class Sample:{
public:
      int GetAverage(const char* query);
private:
      virtual int GetWeight(const char* query, queue* q);
}
int Sample::GetAverage(const char* query)
{
       queue q;
       if(GetWeight(query, &q))
       {
             int sum = 0;
             for(int i = 0; i < q.num; i++)
             {
                   sum += q.weight[i];
             }
             return sum/q.num;
       }
       return -1;
}
```

下面介绍四种常见 MOCK：Stub 技术，宏替换，虚函数重载，HOOK 技术，LIB 重新联编等，并附上部分实例。

7.3.1 Stub 技术

针对与上下游之间的网络通信，数据发送/接收等功能做单测时，会用到 stub 技术。可以开发一个简单的桩程序 stub 来代替下游模块，并提供接口用来在 UT 用例中设置 stub 返回的结果数据。但考虑该方法具有一定开发成本，使用起来不是很方便，加上，网络通信，数据发送/接收等功能的测试更像是接口，联调类测试，放在单测中进行本身就不是很合适。故一般不建议采用，而这里也不举例阐述。

7.3.2 宏替换

宏替换 MOCK 是通过 C/C++中#ifdef 条件编译，来实现产品程序和 UT 用例执行不同的函数。如下示例：（宏替换会修改产品代码，使得产品代码可读性变差，一般不建议采用。）

```
//产品代码如下编写，测试代码 makefile 中使用-D_UNIT_TEST_宏定义
#ifdef _UNIT_TEST_
#define GetWeight m_GetWeight //此函数在测试代码中实现。
#endif
int Sample::GetAverage(const char* query){
       queue q;
       if(GetWeight(query, &q))
       {
            ……
       }
}
#ifdef _UNIT_TEST_
#undef GetWeight
#endif
```

7.3.3 虚函数重载

virtual 虚函数重载的特性，为我们实现 MOCK 提供了天然的接口。如下给出三种场景的 UT 示例：**子类化重写**和**参数/成员变量式适配器**。

缺点：a）对象指针可以被 MOCK，对象不能被 MOCK。

b）被 mock 函数必须是 virtual 函数。

7.3.3.1 子类化重写

```
class Sample:
{
public:
    int GetAverage(const char* query);
private:
    virtual int GetWeight(const char* query, queue* q);
}
int Sample::GetAverage(const char* query)
{
    queue q;
    if(GetWeight(query, &q))
    {
        int sum = 0;
        for(int i = 0; I < q.num; i++)
        {
            sum += q.weight[i];
        }
        return sum/q.num;
    }
    return -1;
}
```

```
class TestSample:public Sample
{
private:
    virtual int GetWeight(const char* query, queue* q);
}
int TestSample::GetWeight(const char* query, queue* q)
{
    if(strcmp(query, ”鲜花”) == 0){
        q->num = 0;
        return 0
    }
    q->num = 2;
    q->weight[0] = 8;
    q->weight[1] = 4;
    return 2;
}
TEST(test_ GetAverage, return_zero_when_qnum_eq_0)
{
    TestSample *testobj = new TestSample();
    ASSERT_EQ(-1, testobj->GetAverage(“鲜花”));
    delete testobj;
}
```

如上定义了一个子类 TestSample，其重新实现了 GetWeight 函数，当 UT 用例中执行 testobj->GetAverage 时，调用的是 sample::GetAverage 函数，但 GetAverage 内部调用的 GetWeight 却是子类 TestSample:: GetWeight 函数。

7.3.3.2 参数适配器

```
class Sample:
{
public:
    int GetAverage(const char* query, Adapter *ada);
}
class Adapter:
{
private:
    virtual int GetWeight(const char* query, queue* q);
}
int Sample::GetAverage(const char* query,Adapter *ada)
```

```
class MockAdapter: public Adapter
{
private:
    virtual int GetWeight(const char* query, queue* q);
}
int MockAdapter::GetWeight(const char* query, queue* q)
{
    if(strcmp(query, ”鲜花”) == 0){
        q->num = 0;
        return 0
    }
```

```
{
        queue q;
        if(ada->GetWeight(query, &q))
        {
              int sum = 0;
              for(int i = 0; i < q.num; i++)
              {
                    sum += q.weight[i];
              }
              return sum/q.num;
        }
        return -1;
}
```

```
        q->num = 2;
        q->weight[0] = 8;
        q->weight[1] = 4;
        return 2;
}
TEST(test_ GetAverage, return_6_when_qnum_eq_2)
        {
        Sample *testobj = new Sample();
        MockAdapter *ada = new MockAdapter();
        ASSERT_EQ(6, testobj->GetAverage("电脑"), ada);
        delete testobj;
        delete ada;
}
```

如上定义了一个子类 MockAdapter，UT 用例中 new 一个 MockAdapter 对象，在调用 GetAverage 时，将子类对象传进去。这样 GetAverage 内部调用的 GetWeight 函数，便是自定义实现的 MockAdapter::GetWeight 函数。

7.3.3.3 成员变量适配器

```
class Sample:
{
public:
      int GetAverage(const char* query);
      Adapter *ada;
}
class Adapter:
{
private:
      virtual int GetWeight(const char* query, queue* q);
}
int Sample::GetAverage(const char* query)
{
        queue q;
        if(ada->GetWeight(query, &q))
        {
              int sum = 0;
              for(int i = 0; i < q.num; i++)
              {
                    sum += q.weight[i];
              }
              return sum/q.num;
        }
        return -1;
}
```

```
class MockAdapter: public Adapter
{
private:
      virtual int GetWeight(const char* query, queue* q);
}

int MockAdapter::GetWeight(const char* query, queue* q)
{
      if(strcmp(query, "鲜花") == 0){
            q->num = 0;
            return 0
        }
        q->num = 2;
        q->weight[0] = 8;
        q->weight[1] = 4;
        return 2;
}

TEST(test_ GetAverage, return_zero_when_qnum_eq_0)
{
        Sample *testobj = new Sample();
        testobj->ada = new MockAdapter();
        ASSERT_EQ(4, testobj->GetAverage("电脑"));
        delete testobj;
}
```

如上定义了一个子类 MockAdapter，UT 用例中 new 一个 MockAdapter 对象，并赋值给 testobj->ada。这样 GetAverage 内部 ada 调用的 GetWeight 函数，便是自定义实现的 MockAdapter::GetWeight 函数。

7.3.4 HOOK 技术

HOOK 技术是完全不同于前面几种 MOCK 技术，其不要求被 MOCK 的函数是虚函数，也

不要求修改产品代码，其通过在运行中动态修改函数地址的方式，达到 MOCK 的效果。在 UT 用例 test_***.cpp 中添加如下代码后。当执行到 GetWeight 函数时，就会跳转到 new_GetWeight 函数，在 new_GetWeight 函数里面判断，若满足 query=xianhua，则 return 0，否则就返回到 old_GetWeight 函数执行。old_GetWeight 起到保存 GetWeight 函数地址的作用。

```
int (*old_GetWeight)(Adapter *ada, const char* query, queue* q);
int new_GetWeight(Adapter *ada, const char* query, queue* q);
{
    if(!strcmp(query, “xianhua”))
    {
        q->num=0;
        return 0;
    }
    return old_ GetWeight (ada, query, q);
}
void __attribute__ ((constructor)) hook_init(void)
{
    attach_func("Adapter::GetWeight", (void *)new_GetWeight, (void **)&old_GetWeight);
}
```

使用 HOOK 技术需要在 test_***.cpp 开头 #include "hookmon.h"，编译和运行需要依赖 app-test/ecom/im/common/utlib/output/lib/hookmon.so，其它介绍见：

http://wiki.babel.baidu.com/twiki/bin/view/Com/Main/Testdbg

7.3.5 Lib 重新联编

如被测函数依赖一些复杂的 lib 函数接口，可以通过自实现一个简单的 LIB 桩去替代真正的 LIB。编译产品程序时，链接的是真正 LIB，而编译 UT 用例时则用自实现的 LIB。该种方法不改变函数地址，也不需要虚函数重载，只需要在产品代码和 UT 代码编译时指定链接不同的 LIB 即可，其多应用在稳定且常用的 lib 打桩方面。

7.3.6 GMOCK

大家常听说的 gmock 本质上是基于虚函数重载进行的二次开发。其提供了很多封装好的 MOCK 宏接口，这里不详细介绍，可参考如下链接：

http://com.baidu.com/twiki/bin/view/Test/GoogleMock 的使用

http://com.baidu.com/twiki/bin/view/Test/Gmock 在单元测试中的应用

7.3.7 BMOCK

最近 BTEST 刚推出的 BMOCK 技术，本质上是基于 HOOK 技术的二次开发，其提供了类似 GMOCK 的宏接口，这里不详细介绍，可参考如下链接：

http://wiki.babel.baidu.com/twiki/bin/view/Com/Test/Bmock 使用帮助