Mock 的基本概念和方法

作者 ： 余祖波(livelylittlefish@gmail.com)
Blog: http://blog.csdn.net/livelylittlefish, http://www.abo321.org

Content

0. 序言

在软件开发中，我们不可避免的要调用一些外部或者系统级别的接口，然而，我们在测试时，也许这些接口或环境并不存在。比如在对我们自己的模块做单元测试时，发现自己的模块依赖的别的模块或接口还没有建立好，如何测试？

Mock 概念应运而生，最开始在 Java 领域，后来各种语言或开发领域均引入该概念。

Mock 实际上就是一种模拟和控制外部或者系统级别对象或接口的方法。因此，我们在做测试时，尤其是单元测试或覆盖测试时，不必与真实环境交互即可完成对自己的模块业务逻辑的测试，或许自己的模块需要依赖外部环境。

因此，我们可以总结

Mock 的本质是：模拟(mock)你的(代码)，来测我的(代码)。

在这里，别人的(代码)，或者与硬件相关的(代码)，或者暂时未完成的(代码)，统称为你的(代码)。

关于单元测试，各种软件工程书籍，http://en.wikipedia.org/wiki/Unit_testing，及其链接有较详细的解释。

关于 Mock 对象，《测试驱动开发-Test-Driven Development》第 7 章，其笔记可参考 MOCK object-第 7 章，http://www.mockobjects.com，http://en.wikipedia.org/wiki/Mock_object，等有较详细的解释。

1. 本文议题

在本文中，笔者将以文件操作为例，讲述基本的 mock 概念和方法。本例中，你的代码 your_file.h/.c 如下。

```
/*
 * your_file.h
 */
#ifndef _YOUR_FILE_H_
#define _YOUR_FILE_H_

#include <stdio.h>

FILE* your_file_open(char *fname);
void your_file_close(FILE* fp);

#endif
```

Your_file.c 是你的代码本来应该有的功能，如打开和关闭文件。

```
/*
 * your_file.c
 */
#include "your_file.h"

FILE* your_file_open(char *fname)
{
    FILE *fp = NULL;

    fp = fopen(fname, "r");
    if (fp == NULL)
    {
        printf("Fail to open file!\n");
        return 0;
    }

    printf("Succeed!\n");
    return fp;
}

void your_file_close(FILE* fp)
{
    fclose(fp);
}
```

首先，做如下假设：

(1) 由于某种原因，这个.c 文件(your_file.c)还有问题；

(2) 或者，这个.c 文件(your_file.c)还没有完成；

(3) 除此以外，我的业务逻辑代码(my_business.c)依赖 your_file.c/.h；

(4) 而且，此时我们需要对 my_business.c 做单元测试，例如，要测试其中的某几个函数等；

那么，在这种情况下，我们应该如何测试自己的业务逻辑？即如何测试 my_business.c 文件？——这将是本文的主要内容。

my_business.c 业务逻辑如下。为了方便，将 main()放在该文件中，实际应用中，main()应该在 main.c 或者别的启动文件中。

```
/*
 * my bussiness
 */

#include "your_file.h"
#include <stdio.h>

int read_file()
{
    FILE* fp = your_file_open("data.txt");
    //assert(fp != 0);

    /*
     * here my business start, for example, read data from the file.
     * for test, only print the fp.
     */
    printf("%s, %d: file handle = 0x%x\n", __FUNCTION__, __LINE__, (unsigned int)fp);

    your_file_close(fp);
    return 0;
}

int main(/* int argc, char **argv */)
{
    read_file();
    return 0;
}
```

注：本文实验对 Win32 平台和 Linux 平台均适用。对于 **make 文件**，Linux 平台为 makefile，win32 平台为 make.bat。

如果代码所在目录下有 data.txt 文件，其运行结果如下。

```
# ./my_bussiness
Succeed!
read_file, 16: file handle = 0x8ba2008
```

2. 应该做什么？

在假设的情况下，现在的问题是，your_file.c 可能还没有完成或者其他原因不能使用，而且还要对 my_business.c 中的 read_file()进行测试，且 my_business.c 依赖 your_file.c，怎么办？

由 Mock 基本概念可知，我们要做的就是模拟(mock)your_file.c 文件的功能。

3. 如何做？

3.1 方案一

方法：模拟 your_file.h/.c 文件，并修改其中的函数名

(1) 建立模拟文件

模拟 your_file.h/.c 文件，将其功能的简单实现放在 mock_your_file.h/.c 文件中，如下。

```
/*
 * mock_your_file.h
 */
#ifndef _MOCK_YOUR_FILE_H_
#define _MOCK_YOUR_FILE_H_

#include <stdio.h>

FILE* mock_your_file_open(char *fname);
void mock_your_file_close(FILE* fp);

#endif
```

可以看到，我们将该函数名改了。

```
/*
 * mock_your_file.c
 */
#include "mock_your_file.h"

FILE* mock_your_file_open(char *fname)
{
    printf("Succeed!\n");
    return (FILE*)0x94f9008;
}

void mock_your_file_close(FILE* fp)
{
    fp = 0;
}
```

可以看到，在打开文件函数中，我们只是返回一个硬编码的指针。现在，mock_your_file.c 是一个单独的模块，用来模拟 your_file.c 的功能。

(2) 修改业务逻辑中的调用

函数名改了，我们需要修改 my_business.c 中的调用。如下。

```
/*
 * my bussiness
 */

#include "mock_your_file.h"  //该包含文件也要修改

void read_file()
{
    FILE* fp = mock_your_file_open("data.txt");  //新的函数名
    //assert(fp != 0);

    /*
     * here my business start, for example, read data from the file.
     * for test, only print the fp.
     */
    printf("%s, %d: file handle = 0x%x\n", __FUNCTION__, __LINE__, (unsigned int)fp);

    mock_your_file_close(fp);  //新的函数名
}

int main(/* int argc, char **argv */)
{
    read_file();
    return 0;
}
```

(3) 修改 make 文件

当然，还需要修改 makefile 文件。如下。

```
CXX = gcc
CXXFLAGS += -g -Wall -Wextra

TESTS = my_bussiness

all : $(TESTS)

clean :
     rm -f $(TESTS) *.o

my_bussiness.o: my_bussiness.c
     $(CXX) $(CXXFLAGS) -c $^

#your_file.o: your_file.c
mock_your_file.o: mock_your_file.c
     $(CXX) $(CXXFLAGS) -c $^

$(TESTS): my_bussiness.o mock_your_file.o
     $(CXX) $(CXXFLAGS) $^ -o $@
```

可以看到，以前的 your_file.c 就不再使用了，换成 mock 文件。

如果是 Win32 平台，其 make.bat 文件也要修改。如下。

```
@echo off

echo start to compile all examples
echo.

cl /wd 4530 /nologo my_bussiness.c mock_your_file.c
echo.

del *.obj

echo done. bye.
pause
```

至此，就达到了模拟 your_file.h/.c 的目的。

(4) 讨论

实际上，这是一个较为笨重的方法，因为该方案需要修改的东西太多，比如要修改文件名，函数名，还要修改 my_business.c 文件中的调用，以及 make 文件，比较麻烦。

一个稍微简单点且不需要修改这么多内容的方法：在模拟文件 mock_your_file.h/.c 文件中不修改函数名，那么 my_business.c 就不需要改动，只修改 make 文件即可。

你甚至可以通过目录隔离的方式，不修改模拟文件名、函数名、make 文件等，唯一要做的仅仅是修改模拟的函数的内容即可。但这导致代码可能有两套，维护也麻烦。

等等，这些方法都是比较肤浅的方法，属于体力活，但实现简单。那么，有没有稍微好一些的方法呢？

3.2 方案二

方法：使用编译预处理的宏定义，让将 fopen 函数换个指向，即实际上模拟 your_open_file()调用的 fopen 函数。

(1) 建立模拟文件

研究 your_open_file()函数的代码发现，其调用的 fopen()函数返回的 FILE*实际上作为 your_open_file()函数的返回值返回。那么，能不能模拟 fopen()函数呢？——of course!

该方案重新编写的 mock 文件如下。

```
/*
 * mock_your_file.h
 */
```

```
#ifndef _MOCK_YOUR_FILE_H_
#define _MOCK_YOUR_FILE_H_

#include <stdio.h>

FILE* mock_fopen(const char *fname, const char* option);
void mock_fclose(FILE* fp);

#endif
```

```
/*
 * mock_your_file.c
 */
#include "mock_your_file.h"

FILE* mock_fopen(const char *fname, const char* option)
{
    return (FILE*)0x94f9008;
}

void mock_fclose(FILE* fp)
{
    fp = 0;
}
```

让 mock_fopen()函数模拟 fopen()，直接返回 FILE*。

其他的文件不需要修改，且 your_file.h/.c 文件仍然使用(区别于方案一)，但要修改 make 文件。

(2) 基本思想

该方案的基本思想是：使用编译预处理的宏定义功能进行(符号)常量定义，即将 fopen 看作一个符号常量，定义该常量的值为模拟函数 mock_fopen；因为 mock_your_file.c 也会被编译，因此，链接时 your_file.c 中对 fopen 的调用便转为对 mock_your_file.c 中的 mock_fopen 的调用。

What a good idea!

(3) 修改 make 文件

该方案的 make 文件，Linux 平台如下。

```
CXX = gcc
CXXFLAGS += -g -Wall -Wextra

TESTS = my_business

MOCK_FLAG = -Dfopen=mock_fopen -Dfclose=mock_fclose  #定义符号常量

all : $(TESTS)

clean :
```

```
	rm -f $(TESTS) *.o

your_file.o: your_file.c
	$(CXX) $(CXXFLAGS) $(MOCK_FLAG) -c $^  #编译 your_file.c 时使用该常量

mock_your_file.o: mock_your_file.c
	$(CXX) $(CXXFLAGS) -c $^

my_business.o: my_business.c
	$(CXX) $(CXXFLAGS) -c $^

$(TESTS): your_file.o mock_your_file.o my_business.o
	$(CXX) $(CXXFLAGS) $^ -o $@
```

可以看出，3 个源文件均参与编译、链接，区别于方案一(your_file.c 不再参与)。

win32 平台的 make.bat 文件为，

```
@echo off

echo start to compile all examples
echo.

cl /wd 4996 /nologo /Dfopen=mock_fopen /Dfclose=mock_fclose my_business.c
mock_your_file.c your_file.c
echo.

del *.obj

echo done. bye.
pause
```

(4) 讨论

该方案比较于方案一提出的各种“肤浅”方案，要巧妙的多。其关键之处就是利用编译器的编译预处理的宏定义功能，定义符号常量。将 fopen 看作符号常量，其值定义为 mock_fopen，链接时对符号的 resolve 处理，即会将对 fopen 的调用转为对 mock_fopen 的调用。

从 your_file.c 编译后的.o 文件的符号表中也能看出端倪。

```
# objdump -t your_file.o

your_file.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 your_file.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .debug_abbrev  00000000 .debug_abbrev
00000000 l    d  .debug_info    00000000 .debug_info
00000000 l    d  .debug_line    00000000 .debug_line
00000000 l    d  .rodata        00000000 .rodata
00000000 l    d  .debug_frame   00000000 .debug_frame
00000000 l    d  .debug_loc     00000000 .debug_loc
```

```
00000000 l    d  .debug_pubnames        00000000 .debug_pubnames
00000000 l    d  .debug_aranges 00000000 .debug_aranges
00000000 l    d  .debug_str     00000000 .debug_str
00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
00000000 l    d  .comment       00000000 .comment
00000000 g     F .text  00000055 your_file_open
00000000         *UND*  00000000 mock_fopen
00000000         *UND*  00000000 puts
00000055 g     F .text  00000013 your_file_close
00000000         *UND*  00000000 mock_fclose
```

如果没有 MOCK_FLAG，其编译后的符号表和上述符号表的唯一差别就是这两个符号，分别为 fopen 和 fclose。

(5) 该方案的变种

将该符号定义放在 Your_file.c 文件中，即-D 命令行方式的另一种方式。

```
/*
 * your_file.c
 */
#include "your_file.h"
#include "mock_your_file.h"
#define fopen mock_fopen
#define fclose mock_fclose

FILE* your_file_open(char *fname)
{
    FILE *fp = NULL;

    fp = fopen(fname, "r");
    if (fp == NULL)
    {
        printf("Fail to open file!\n");
        return 0;
    }

    printf("Succeed!\n");
    return fp;
}

void your_file_close(FILE* fp)
{
    fclose(fp);
}
```

注 1：该变种方法的 make 文件，需要将 makefile/make.bat 中的 MOCK_FLAG 及其使用均删除。

注 2：参考(3)的 makefile，编译时只是对 your_file.c 使用了 MOCK_FLAG，因此，只能将宏定义放在 your_file.c 文件。

注 3：单步执行会发现，对 fopen 的调用被 resolve 到对 mock_fopen 的调用；同样地，对 fclose 的调用被 resolve 到对 mock_fclose 的调用。

注 4：若将该宏定义放在 my_business.c 文件中，则达不到目的，为什么？读者可自行思考。

4. 小结

本文通过例子讲述了 Mock 的基本概念和方法，并提出了两种 mock 方案，比较而言，方案一较为肤浅，但实现简单；虽然方案二相比方案一利用了编译预处理的宏定义的技巧，但若文件过多，mock 文件也会多，从而导致 make 文件的维护工作增加，或者要添加很多宏定义，也难以维护。

那么有没有一种比较好的方法，能自动产生一些文件或者目录，供测试使用呢？答案是“有，一定有，因为，这个世界从来不缺聪明的发明人”。笔者将在“Mock 的基本概念和方法(续)”一文讲解使用 Cmock、Unity 等工具的方法。

Reference

http://www.mockobjects.com
http://en.wikipedia.org/wiki/Mock_object
http://en.wikipedia.org/wiki/Unit_testing
<TDD:驱动测试开发>,第 7 章

本文例子目录

F:\study\12.MyCode\mock\2011-4-12.sample0
F:\study\12.MyCode\mock\2011-4-12.sample1
F:\study\12.MyCode\mock\2011-4-12.sample1.2
F:\study\12.MyCode\mock\2011-4-12.sample2
F:\study\12.MyCode\mock\2011-4-12.sample2.2
方案二的参考例子 F:\study\12.MyCode\mock\2011-04-12.mock_test

Mock 的基本概念和方法(续)

作者 ： 余祖波(livelylittlefish@gmail.com)
Blog: http://blog.csdn.net/livelylittlefish, http://www.abo321.org

Content

0. 序

在 **Mock 的基本概念和方法**一文中，笔者在结尾提出了一个问题，有没有一种比较好的方法或者工具，能自动产生一些文件或者目录，供测试使用？

有，一定有，因为，这个世界从来不缺聪明的发明人。

那么，本文仍以 **Mock 的基本概念和方法**一文中的例子为例，重点讲述使用 Cmock、Unity 等工具进行单元测试的方法。

注：本文的实验，CUnit 不是必须的。

1. 平台

Cygwin, or Linux

工作目录:

```
Linux : # cd /usr/src/cmock_sample
Cygwin: $ cd /usr/src/cmock_sample
```

2. 第三方库

本文实验用到的第三方库分别为：

cmock_2_0_204.zip：
http://sourceforge.net/projects/cmock/files/cmock/cmock2.0/cmock_2_0_204.zip
unity_2_0_113.zip：
http://sourceforge.net/projects/unity/files/unity/unity2.0/unity_2_0_113.zip
ruby-1.9-stable.tar.gz：http://ftp.ruby-lang.org/pub/ruby/ruby-1.9-stable.tar.gz

实际上，cmock_2_0_204.zip 在其 vendor 目录中包含了 Unity。

3. 如何使用 CMock 和 Unity?

以下将以 cygwin 平台为例。

3.1 修改业务代码

该例子中的业务代码文件：my_business.c
为例使用 CMock 和 Unity 工具，需要修改 my_business.c 文件，将其中的 main 函数删除，并增加 my_business.h 文件。如下。

```
/*
 * my bussiness.h
 */

int read_file();
```

my_business.c 文件内容。

```
/*
 * my bussiness
 */
#include "my_business.h"
#include <stdio.h>

int read_file()
{
    FILE* fp = your_file_open("data.txt");
    //assert(fp != 0);

    /*
     * here my business start, for example, read data from the file.
```

```
     * for test, only print the fp.
     */
    printf("%s, %d: file handle = 0x%x\n", __FUNCTION__, __LINE__, (unsigned int)fp);

    your_file_close(fp);
    return 0;
}
```

注：实际上，或许某些场合不需要该头文件，但为了本文说明问题的需要，提供该文件。后面的 Unity 工具将根据该文件自动生成 mock 文件。

3.2 编写单元测试

要对业务代码做单元测试，应该事先编写好单元测试代码，文件为 my_business_unittest.c，如下。

```
/*
 * my bussiness unittest
 */

#include "my_business.h"
#include "unity.h"

void setUp(void)
{
}

void tearDown(void)
{
}

void test_read_file()
{
    TEST_ASSERT_EQUAL(0, read_file());
}

void test_read_file2()
{
    TEST_ASSERT_NOT_EQUAL(1, read_file());
}
```

3.3 如何 Mock 并测试?

(1) 生成 mock 文件

业务代码 my_business.h/.c 依赖 your_file.h/.c，根据 Cmock/Unity 框架，要根据 your_file.h 自动生成 mock 文件。Cmock 框架提供了该自动生成的功能，是用 ruby 写的 scripts，在./cmock/lib 目录下，在运行这些脚本时，使用 ruby 命令，因此需要第三方库 ruby。完成该自动生成功能的脚本是./cmock/lib/cmock.rb，可以处理 1 个或多个.h 文件。

```
$ mkdir mocks  //默认情况下，生成的 mock 文件保存在该目录，故需事先建立好
$ ruby /usr/src/cmock/lib/cmock.rb your_file.h  //运行该脚本生成 mock 文件
```

```
Creating mock for your_file...
$ cd mocks
$ ls
Mockyour_file.c  Mockyour_file.h  //自动生成的 mock 文件
$ cd ..
```

此处，笔者仅提供 Mockyour_file.h 的内容，Mockyour_file.c 文件太大，请参考附录。

```
/* AUTOGENERATED FILE. DO NOT EDIT. */
#ifndef _MOCKYOUR_FILE_H
#define _MOCKYOUR_FILE_H

#include "your_file.h"

void Mockyour_file_Init(void);
void Mockyour_file_Destroy(void);
void Mockyour_file_Verify(void);

#define your_file_open_ExpectAndReturn(fname, cmock_retval)
your_file_open_CMockExpectAndReturn(__LINE__, fname, cmock_retval)
void your_file_open_CMockExpectAndReturn(UNITY_LINE_TYPE cmock_line, char* fname, FILE*
cmock_to_return);
#define your_file_close_Expect(fp) your_file_close_CMockExpect(__LINE__, fp)
void your_file_close_CMockExpect(UNITY_LINE_TYPE cmock_line, FILE* fp);

#endif
```

(2) 生成单元测试 runner

Unity 工具提供了根据单元测试文件自动生成 runner 文件的功能，该 runner 文件中包含 main()函数，这就是单元测试程序启动的入口。这也是为什么要修改 my_business.c，将其中的 main()删除的原因。

What a great idea! Ruby 真的是一门很好的语言，是时候学习了。

这将节省开发人员很多的时间，使开发人员将精力集中在单元测试本身和设计测试用例上。

```
$ ruby /usr/src/unity/auto/generate_test_runner.rb my_business_unittest.c my_business_runner.c
$ ls
data.txt  makefile  my_business.c  my_business_runner.c    your_file.c
make.bat  mocks     my_business.h  my_business_unittest.c  your_file.h
```

(3) 拷贝或修改生成的 mock 文件

因为生成的 mock 文件放在 mocks 目录，且在 Mockyour_file.h 中包含了 your_file.h，但 mocks 目录中并没有 your_file.h 文件，而是在上一级目录中，因此需要将其拷贝到上一级目录，或者修改 Mockyour_file.h 文件中包含 your_file.h 的路径。笔者选择拷贝。

```
$ cp mocks/Mockyour_file.h Mockyour_file.h
$ cp mocks/Mockyour_file.c Mockyour_file.c
```

自此，要进行单元测试所需的.h/.c 文件全部建立，小结一下，看看有哪些文件。

```
你的代码文件：your_file.h/.c          (该文件在测试时不再使用)
mocked 文件 ：Mockyour_file.h/.c     (在测试时要依赖该文件)
业务代码文件：my_business.h/.c
业务测试文件：my_business_unittest.c
测试运行文件：my_business_runner.c  (main()函数即在其中)
```

(4) build

本文的实验，笔者编写了 makefile(Linux 平台/Cygwin 环境)或者 make.bat(win32 平台)，让 build 更加快速和可控。

使用 CMokc 和 Unity 进行单元测试和 Mock，一定会依赖 CMock 和 Unity，因此这两个工具的源代码 cmock.c 和 unity.c 也一定要编译到并链接到目标文件。那么对于 CMock 和 Unity，我们需要或者依赖哪些文件呢？

不要被 CMock 和 Unity 的内容吓到！虽然其中有很多目录和文件，但其核心(框架)文件很少，而这些核心文件正是 build 需要的。

```
CMock: ./cmock/src/cmock.c  //CMock 的 core 文件在./cmock/src 目录
       ./cmock/src/cmock.h
Unity: ./unity/src/unity.c  //Unity 的 core 文件在./unity/src 目录
       ./unity/src/unity.h
       ./unity/src/unity_internals.h
```

综上，编写的 makefile 文件和 make.bat 文件如下。makefile 文件适用于 cygwin 和 Linux 平台。

```
CXX = gcc
CXXFLAGS += -g -Wall -Wextra

TESTS = my_business

CMOCK_DIR = /usr/src/cmock
CMOCK_SRC = $(CMOCK_DIR)/src
UNITY_DIR = /usr/src/unity
UNITY_SRC = $(UNITY_DIR)/src

ifeq ($(OS), Windows_NT)
     TARGET_EXTENSION=.exe
else
     TARGET_EXTENSION=.out
endif

TARGET = $(TESTS)$(TARGET_EXTENSION)

CLEANUP = rm -f $(TARGET) *.o
```

```
all : $(TARGET)

clean :
	$(CLEANUP)

unity.o: $(UNITY_SRC)/unity.c
	$(CXX) $(CXXFLAGS) -I$(UNITY_SRC) -c $^

cmock.o: $(CMOCK_SRC)/cmock.c
	$(CXX) $(CXXFLAGS) -I$(CMOCK_SRC) -I$(UNITY_SRC) -c $^

mockyour_file.o: mockyour_file.c
	$(CXX) $(CXXFLAGS) -I$(CMOCK_SRC) -I$(UNITY_SRC) -c $^

my_business.o: my_business.c
	$(CXX) $(CXXFLAGS) -c $^

my_business_unittest.o: my_business_unittest.c
	$(CXX) $(CXXFLAGS) -I$(UNITY_SRC) -c $^

my_business_runner.o: my_business_runner.c
	$(CXX) $(CXXFLAGS) -I$(UNITY_SRC) -c $^

$(TARGET): unity.o cmock.o mockyour_file.o my_business.o my_business_unittest.o
my_business_runner.o
	$(CXX) $(CXXFLAGS) $^ -o $@
```

(5) 讨论

这真的是一种前无古人的方法，CMock 能自动生成 mock 文件，让你进行单元测试；Unity 能自动生成单元测试 runner，让你的单元测试跑起来。CMock 和 Unity 的出现，为广大开发者节省了宝贵的时间，使你的精力集中于单元测试本身和测试用例的设计上。

4. 总结

本文，笔者以一个例子讨论了使用 CMock 和 Unity 进行 mock 和单元测试的方法和过程，主要关注如何使用这两个工具并 build，build 时，要注意 CMock 和 Unity 的 cmock.c 和 unity.c 也要编译并链接到目标文件。

本文重点如下。

- 生成 mock 文件
- 生成单元测试 runner
- build

广大软件开发者们，如果要做 C 项目的单元测试，尤其是嵌入式软件的单元测试，那就去大胆尝试 CMock 和 Unity 吧！

Reference

CMock 简介：CMock Summary.pdf
Unity 简介：Unity Summary.pdf
CMock 设计原理：Functionality_and_Design_of_the_CMock_Framework.pdf
如何 mocking 嵌入式软件：Mocking the Embedded World - Test-Driven Development, Continuous Integration, and Design Patterns.pdf

http://sourceforge.net/apps/trac/cmock
http://sourceforge.net/apps/trac/unity
http://sourceforge.net/projects/unity/forums/forum/770030/topic/4067272
http://sourceforge.net/projects/unity/forums/forum/770030/topic/3795145
http://sourceforge.net/apps/trac/cmock/wiki/EclipseIde
http://sourceforge.net/apps/mediawiki/unity/index.php?title=Eclipse_IDE_Integration
http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks （各种单元测试工具比较）
http://sourceforge.net/projects/unity/forums/forum/770030/topic/4067272 （Cmock 和 Unity 的关系）
http://meekrosoft.wordpress.com/2010/01/29/unit-test-embedded-software-in-3-easy-steps （嵌入式软件单元测试的 3 个步骤）

http://www.lulu.com/product/paperback/embedded-testing-with-unity-and-cmock/14408590?productTrackingContext=search_results/search_shelf/center/1
<Embedded Testing with Unity and Cmock>, By Mark VanderVoord.

Appendix 1: Cmock installation

```
# cd /usr/src
# wget http://sourceforge.net/projects/cmock/files/cmock/cmock2.0/cmock_2_0_204.zip
# unzip -x cmock_2_0_204.zip
# cd cmock
```

编译需要 ruby 的构建系统 rake，有待研究。

Appendix 2: Unity installation

```
# cd /usr/src
# wget http://sourceforge.net/projects/unity/files/unity/unity2.0/unity_2_0_113.zip
# unzip -x unity_2_0_113.zip
# cd unity
# make
# cd examples
# make
```

Appendix 3: Ruby installation

```
$ cd /usr/src
$ wget http://ftp.ruby-lang.org/pub/ruby/ruby-1.9-stable.tar.gz
$ tar.exe  -zxvf ruby-1.9-stable.tar.gz
$ cd ruby-1.9.2-p180
$ ./configure && make && make install
```

Appendix 4: Mockyour_file.h/.c

文件名：Mockyour_file.h

```
/* AUTOGENERATED FILE. DO NOT EDIT. */

//=======Test Runner Used To Run Each Test Below=====
#define RUN_TEST(TestFunc, TestLineNum) \
{ \
  Unity.CurrentTestName = #TestFunc; \
  Unity.CurrentTestLineNumber = TestLineNum; \
  Unity.NumberOfTests++; \
  if (TEST_PROTECT()) \
  { \
      setUp(); \
      TestFunc(); \
  } \
  if (TEST_PROTECT() && !TEST_IS_IGNORED) \
  { \
    tearDown(); \
```

```
  } \
  UnityConcludeTest(); \
}

//=======Automagically Detected Files To Include=====
#include "unity.h"
#include <setjmp.h>
#include <stdio.h>

//=======External Functions This Runner Calls=====
extern void setUp(void);
extern void tearDown(void);
extern void test_read_file();
extern void test_read_file2();

//=======Test Reset Option=====
void resetTest()
{
  tearDown();
  setUp();
}

//=======MAIN=====
int main(void)
{
  Unity.TestFile = "my_business_unittest.c";
  UnityBegin();
  RUN_TEST(test_read_file, 16);
  RUN_TEST(test_read_file2, 21);

  return (UnityEnd());
}
```

文件名：Mockyour_file.c

```
/* AUTOGENERATED FILE. DO NOT EDIT. */
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include "unity.h"
#include "cmock.h"
#include "Mockyour_file.h"

typedef struct _CMOCK_your_file_open_CALL_INSTANCE
{
  UNITY_LINE_TYPE LineNumber;
  FILE* ReturnVal;
  char* Expected_fname;

} CMOCK_your_file_open_CALL_INSTANCE;

typedef struct _CMOCK_your_file_close_CALL_INSTANCE
{
  UNITY_LINE_TYPE LineNumber;
  FILE* Expected_fp;

} CMOCK_your_file_close_CALL_INSTANCE;

static struct Mockyour_fileInstance
{
  CMOCK_MEM_INDEX_TYPE your_file_open_CallInstance;
  CMOCK_MEM_INDEX_TYPE your_file_close_CallInstance;
} Mock;

extern jmp_buf AbortFrame;

void Mockyour_file_Verify(void)
{
  UNITY_LINE_TYPE cmock_line = TEST_LINE_NUM;
  UNITY_TEST_ASSERT(CMOCK_GUTS_NONE == Mock.your_file_open_CallInstance, cmock_line,
"Function 'your_file_open' called less times than expected.");
  UNITY_TEST_ASSERT(CMOCK_GUTS_NONE == Mock.your_file_close_CallInstance, cmock_line,
"Function 'your_file_close' called less times than expected.");
}

void Mockyour_file_Init(void)
{
  Mockyour_file_Destroy();
}
```

```
void Mockyour_file_Destroy(void)
{
  CMock_Guts_MemFreeAll();
  memset(&Mock, 0, sizeof(Mock));
}

FILE* your_file_open(char* fname)
{
  UNITY_LINE_TYPE cmock_line = TEST_LINE_NUM;
  CMOCK_your_file_open_CALL_INSTANCE* cmock_call_instance =
(CMOCK_your_file_open_CALL_INSTANCE*)CMock_Guts_GetAddressFor(Mock.your_file_open_CallIn
stance);
  Mock.your_file_open_CallInstance =
CMock_Guts_MemNext(Mock.your_file_open_CallInstance);
  UNITY_TEST_ASSERT_NOT_NULL(cmock_call_instance, cmock_line, "Function 'your_file_open'
called more times than expected.");
  cmock_line = cmock_call_instance->LineNumber;
  UNITY_TEST_ASSERT_EQUAL_STRING(cmock_call_instance->Expected_fname, fname, cmock_line,
"Function 'your_file_open' called with unexpected value for argument 'fname'.");
  return cmock_call_instance->ReturnVal;
}

void CMockExpectParameters_your_file_open(CMOCK_your_file_open_CALL_INSTANCE*
cmock_call_instance, char* fname)
{
  cmock_call_instance->Expected_fname = fname;
}

void your_file_open_CMockExpectAndReturn(UNITY_LINE_TYPE cmock_line, char* fname, FILE*
cmock_to_return)
{
  CMOCK_MEM_INDEX_TYPE cmock_guts_index =
CMock_Guts_MemNew(sizeof(CMOCK_your_file_open_CALL_INSTANCE));
  CMOCK_your_file_open_CALL_INSTANCE* cmock_call_instance =
(CMOCK_your_file_open_CALL_INSTANCE*)CMock_Guts_GetAddressFor(cmock_guts_index);
  UNITY_TEST_ASSERT_NOT_NULL(cmock_call_instance, cmock_line, "CMock has run out of
memory. Please allocate more.");
  Mock.your_file_open_CallInstance =
CMock_Guts_MemChain(Mock.your_file_open_CallInstance, cmock_guts_index);
  cmock_call_instance->LineNumber = cmock_line;
  CMockExpectParameters_your_file_open(cmock_call_instance, fname);
  cmock_call_instance->ReturnVal = cmock_to_return;
}

void your_file_close(FILE* fp)
{
  UNITY_LINE_TYPE cmock_line = TEST_LINE_NUM;
  CMOCK_your_file_close_CALL_INSTANCE* cmock_call_instance =
(CMOCK_your_file_close_CALL_INSTANCE*)CMock_Guts_GetAddressFor(Mock.your_file_close_Call
Instance);
  Mock.your_file_close_CallInstance =
CMock_Guts_MemNext(Mock.your_file_close_CallInstance);
  UNITY_TEST_ASSERT_NOT_NULL(cmock_call_instance, cmock_line, "Function
'your_file_close' called more times than expected.");
  cmock_line = cmock_call_instance->LineNumber;
  UNITY_TEST_ASSERT_EQUAL_MEMORY((void*)(cmock_call_instance->Expected_fp), (void*)(fp),
sizeof(FILE), cmock_line, "Function 'your_file_close' called with unexpected value for
argument 'fp'.");
}

void CMockExpectParameters_your_file_close(CMOCK_your_file_close_CALL_INSTANCE*
cmock_call_instance, FILE* fp)
{
  cmock_call_instance->Expected_fp = fp;
}

void your_file_close_CMockExpect(UNITY_LINE_TYPE cmock_line, FILE* fp)
{
  CMOCK_MEM_INDEX_TYPE cmock_guts_index =
CMock_Guts_MemNew(sizeof(CMOCK_your_file_close_CALL_INSTANCE));
  CMOCK_your_file_close_CALL_INSTANCE* cmock_call_instance =
(CMOCK_your_file_close_CALL_INSTANCE*)CMock_Guts_GetAddressFor(cmock_guts_index);
  UNITY_TEST_ASSERT_NOT_NULL(cmock_call_instance, cmock_line, "CMock has run out of
memory. Please allocate more.");
  Mock.your_file_close_CallInstance =
CMock_Guts_MemChain(Mock.your_file_close_CallInstance, cmock_guts_index);
  cmock_call_instance->LineNumber = cmock_line;
  CMockExpectParameters_your_file_close(cmock_call_instance, fp);
}
```

Appendix 5: my_business_runner.c

```
/* AUTOGENERATED FILE. DO NOT EDIT. */

//=======Test Runner Used To Run Each Test Below=====
#define RUN_TEST(TestFunc, TestLineNum) \
{ \
  Unity.CurrentTestName = #TestFunc; \
  Unity.CurrentTestLineNumber = TestLineNum; \
  Unity.NumberOfTests++; \
  if (TEST_PROTECT()) \
  { \
      setUp(); \
      TestFunc(); \
  } \
  if (TEST_PROTECT() && !TEST_IS_IGNORED) \
  { \
    tearDown(); \
  } \
  UnityConcludeTest(); \
}

//=======Automagically Detected Files To Include=====
#include "unity.h"
#include <setjmp.h>
#include <stdio.h>

//=======External Functions This Runner Calls=====
extern void setUp(void);
extern void tearDown(void);
extern void test_read_file();
extern void test_read_file2();


//=======Test Reset Option=====
void resetTest()
{
  tearDown();
  setUp();
}


//=======MAIN=====
int main(void)
{
  Unity.TestFile = "my_business_unittest.c";
  UnityBegin();
  RUN_TEST(test_read_file, 16);
  RUN_TEST(test_read_file2, 21);

  return (UnityEnd());
}
```