# 安装方法

gtest development package:

sudo apt-get install libgtest-dev #安装源代码到/usr/src/gtest

注意这一步只是安装源代码到/usr/src/gtest，需要用cmake构建Makefile然后再make生成静态库。

sudo apt-get install cmake  #安装cmake

cd /usr/src/gtest

sudo cmake CMakeLists.txt #记得加sudo 否则下一步会有可能报错

sudo make #生成libgtest\_main.a 和 libgtest.a两个静态链接库

sudo cp \*.a /usr/lib/  #拷贝生成的库到/usr/lib/

# MOCK\_METHODn()

在MockTurtle的public:部分，写上MOCK\_METHODn(); (如果你要Mock一个const

函数，就写MOCK\_CONST\_METHODn )，其中n是函数中的参数个数

函数名作为宏的第一个参数，然后将函数定义

中除函数名以外的部分作为宏的第二个参数。

#include "gmock/gmock.h" // Brings in Google Mock.

class MockTurtle : public Turtle {

public:

...

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());

};

你不需要再在别的地方去定义这些Mock函数了，MOCK\_METHOD\*宏会帮你产生这些函

数定义。

必须将Mock函数定义( MOCK\_METHOD\* )放到Mock类的public:部分中，无论被

Mock的函数在基类中是public，protected，还是private。这样做是为了让ON\_CALL

和EXPECT\_CALL可以从Mock类外引用Mock函数。( 是的，C++允许子类改变一个基类虚函数的访问权限)

# EXPECT\_CALL()

在Goolge Mock中，我们用EXPECT\_CALL()宏来设置一个Mock函数上的期望。一般

语法是：

EXPECT\_CALL(mock\_object, method(matchers))

.Times(cardinality)

.WillOnce(action)

.WillRepeatedly(action);

这个宏有两个参数：第一个是Mock对象，第二个参数是函数和它的参数。注意两个参

数是用逗号( , )分隑的，而丌是句号( . )。

在EXPECT\_CALL()之后第一个我们可以指定的子句是Times()。我们称Times的参数

为cardinality，因为它是指这个函数应该被调用多少次

Times()子句可以省略。如果你省略Times()，Google Mock会推断出cardinality

的值是什么。这个规则很容易记：

 如果在EXPECT\_CALL中既没有WillOnce()也没有WillRepeatedly()，那推断出的

cardinality就是Times(1)。

 如果有n个WillOnce()，但没有WillRepeatedl()，其中n >= 1，那么cardinality就是Times(n)。

 如果有n个WillOnce()，和一个WillRepeatedly()，其中n >= 0，那么cardinality就是Times(AtLeast(n))。

EXPECT\_CALL(turtle, GetX())

.Times(5)

.WillOnce(Return(100))

.WillOnce(Return(150))

.WillRepeatedly(Return(200));

说明如下

公布答案，turtle对象的GetX()方法会被调用5次，它第一次返回100，第二次返回

150，然后每次返回200。许多人喜欢称这种语法方式为特定领域语言( Domain-Specific Language (DSL)

参数的模糊处理

EXPECT\_CALL(turtle, Forward(100));) )。

有时你可能丌想指定的太精确( 还记得前面测试丌应太严格吗？指定的太精确会导致

测试健壮性丌足，并影响测试的本意。所以我们鼓励你叧指定那些必须要指定的参数，丌要多，也丌要少 )

EXPECT\_CALL(turtle, Forward(\_)); //“\_”作为参数，它的意义是“任意”参数。

EXPECT\_CALL(turtle, Forward(Ge(100))); //至少前迚至少100个单位

Actions: Mock函数的返回值

如果Mock函数的返回类型是一个指针戒是内置类型，那这个函数是有默认行为

的( 一个void函数直接返回，bool函数返回false，其它函数返回0 )。如果你丌想改

变它，那这种行为就会被应用。

如果一个Mock函数没有默认行为，戒默认行为丌适合你，你可以用WillOnce

来指定每一次的返回值是什么，最后可以选用WillRepeatedly来结束。

EXPECT\_CALL(turtle, GetX())

.WillOnce(Return(100))

.WillOnce(Return(200))

.WillOnce(Return(300));

//上面的意思是turtle.GetX()会被调用恰好3次，并分别返回100，200，300。

using ::testing::Return;...

EXPECT\_CALL(turtle, GetY())

.WillOnce(Return(100))

.WillOnce(Return(200))

.WillRepeatedly(Return(300));

//上面的意思是指turtle.GetY()将至少被调用2次，第一次返回100，第二次返回200，从第三次以后都返回300。

在WillOnce()里除了写Return()我们还能做些什么呢？你可以用

ReturnRef( variable )，或是调用一个预先定义好的函数，在cheatsheet#Action中找吧。

|`Return()`|Return from a `void` mock function.|

|:---------|:----------------------------------|

|`Return(value)`|Return `value`.

If the type of `value` is different to the mock function's return type, `value` is converted to the latter type <i>at the time the expectation is set</i>, not when the action is executed.|

|`ReturnArg<N>()`|Return the `N`-th (0-based) argument.|

|`ReturnNew<T>(a1, ..., ak)`|Return `new T(a1, ..., ak)`; a different object is created each time.|

|`ReturnNull()`|Return a null pointer. |

|`ReturnPointee(ptr)`|Return the value pointed to by `ptr`.|

|`ReturnRef(variable)`|Return a reference to `variable`. |

|`ReturnRefOfCopy(value)`|Return a reference to a copy of `value`; the copy lives as long as the action.|

# 帮助文档列表

* [ForDummies](local://base_request.html/ForDummies.md) -- start here if you are new to Google Mock.
* [CheatSheet](local://base_request.html/CheatSheet.md) -- a quick reference.
* [CookBook](local://base_request.html/CookBook.md) -- recipes for doing various tasks using Google Mock.
* [FrequentlyAskedQuestions](local://base_request.html/FrequentlyAskedQuestions.md) -- check here before asking a question on the mailing list.

To contribute code to Google Mock, read:

* [DevGuide](local://base_request.html/DevGuide.md) -- read this *before* writing your first patch.
* [Pump Manual](local://base_request.html/../googletest/docs/PumpManual.md) -- how we generate some of Google Mock's source files.

int n = 100;

EXPECT\_CALL(turtle, GetX())

.Times(4)

.WillRepeatedly(Return(n++));

它并不是依次返回100，101，102...，而是每次都返回100，因为n++叧会被求一次

值。类似的，Return(new Foo)当EXPECT\_CALL()求值时叧会创建一个Foo对象，所以它会每次都返回相同的指针。如果你希望每次都看到丌同的结果，你需要定义一个自定义行为，我们将在CookBook中指导你。

重要提示：EXPECT\_CALL()叧对行为子句求一次值，尽管这个行为可能出现很多次。

所以你必须小心这种副作用。

using ::testing::Return;...

EXPECT\_CALL(turtle, GetY())

.Times(4)

.WillOnce(Return(100));

turtle.GetY()第一次返回100，以后每次都返回0，因为0是默认行为的返回值。

每次调用都会消耗一个WillOnce()子句，消耗完后，就会使用默认行为。

Mock一个模板类，需要在MOCK\_\*宏后加上\_T：

template <typename Elem>

class StackInterface {

...

// Must be virtual as we'll inherit from StackInterface.

virtual ~StackInterface();

virtual int GetSize() const = 0;

virtual void Push(const Elem& x) = 0;

};

template <typename Elem>

class MockStack : public StackInterface<Elem> {

...

MOCK\_CONST\_METHOD0\_T(GetSize, int());

MOCK\_METHOD1\_T(Push, void(const Elem& x));

};

Mocking Non-virtual Methods

Mock非虚函数时并不与真实的类共享一个公共的基类，你的Mock类与真实类将毫无

关系，但两者所定义的函数却是一致的。Mock非虚函数与Mock虚函数的语法是一致的

要想出一种在正式代码中使用ConcretePacketStream，在测试代码中

使用MockPacketStream的方法。因为函数是非虚的，而两个类也是毫无关系的，所以你必须在编译时( 而不是运行时 )决定你使用的类。

其中一种方法是模板化需要用Packet Stream的代码。具体一点，你在代码中使用一

个针对packet stream模板参数。在正式代码中，你可以用ConcretePacketStream来

实例化，在测试中你用MockPacketStream来实例化。

Delegating Calls to a Fake

有时你已经有一个对某一接口的Fake实现了

# Gtest断言

### 快速入门:

Google Test提供了两种断言形式，

一种以**ASSERT\_\***开头， Fatal

另一种以**EXPECT\_\***开头，NonFatal

它们的区别是**ASSERT\_\*** 一旦失败立马退出，而**EXPECT\_\***还能继续下去。

#### 断言列表:

##### 真假条件测试:

|  |  |  |
| --- | --- | --- |
| **致命断言Fatal assertion** | **非致命断言 Nonfatal assertion** | **验证条件 Verifies** |
| ASSERT\_TRUE(condition); | EXPECT\_TRUE(condition); | condition为真 |
| ASSERT\_FALSE(condition); | EXPECT\_FALSE(condition); | condition 为假 |

##### 数据对比测试:

|  |  |  |
| --- | --- | --- |
| **致命断言** | **非致命断言** | **验证条件** |
| ASSERT\_EQ(期望值, 实际值); | EXPECT\_EQ(期望值, 实际值); | 期望值 == 实际值 |
| 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 |

##### 字符串(针对C形式的字符串，即char\*或wchar\_t\*)对比测试:

|  |  |  |
| --- | --- | --- |
| **致命断言** | **非致命断言** | **验证条件** |
| 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 |

# 夹具 Fixture

夹具是加工时用来迅速紧固工件，使[机床](http://baike.baidu.com/view/9884.htm)、刀具、工件保持正确相对位置的工艺装置。也就是说Workholding工装夹具是机械加工不可缺少的部件，在机床技术向高速、高效、精密、复合、智能、环保方向发展的带动下，夹具技术正朝着高精、高效、模块、组合、通用、经济方向发展。

夹具又称[卡具](http://baike.baidu.com/view/830186.htm)。从广义上说，在工艺过程中的任何工序，用来迅速、方便、安全地安装工件的装置，都可称为夹具。例如[焊接夹具](http://baike.baidu.com/view/1144333.htm)、检验夹具、装配夹具、[机床夹具](http://baike.baidu.com/view/1772369.htm)等。其中机床夹具最为常见，常简称为夹具。

在机床上加工工件时，为使工件的表面能达到图纸规定的尺寸、几何形状以及与其他表面的相互[位置精度](http://baike.baidu.com/view/286866.htm)等技术要求，加工前必须将工件装好（定位）、夹牢（夹紧）。

应用机床夹具，有利于保证工件的[加工精度](http://baike.baidu.com/view/1535088.htm)、稳定产品质量；有利于提高劳动生产率和降低成本；有利于改善工人劳动条件，保证安全生产；有利于扩大机床工艺范围，实现“一机多用”。

夹具通常由定位元件（确定工件在夹具中的正确位置）、夹紧装置、[对刀](http://baike.baidu.com/view/1251881.htm)引导元件(确定刀具与工件的相对位置或导引刀具方向)、[分度装置](http://baike.baidu.com/view/1671062.htm)（使工件在一次安装中能完成数个[工位](http://baike.baidu.com/view/998319.htm)的加工，有回转分度装置和直线移动分度装置两类）、连接元件以及夹具体（夹具底座）等组成。

夹具在电子厂商使用也是非常高的，在生产中为了提高生产效率和产品质量，在生产的中段和后段就常用工装夹具来进行[功能测试](http://baike.baidu.com/view/651435.htm)或者辅助装配（能装配出固定的外形及高度等）。实际在电子生产制造厂中的用途是非常广，因为是非标定制的在，所以只有想不到而没有做不到的。

Sometimes you have tests that share common setup steps or data

Instead of repeatedly creating the same test setup multiple times, create a test fixture to share configurations

* + Derive a class from ::testing::Test
  + Create any objects you need
  + Create SetUp() and TearDown() functions as needed
  + Use the TEST\_F() macro instead of TEST()
  + The test fixture will run its constructors and run SetUp() prior to running each TEST\_F() test, and will call TearDown() and the destructor after running each test.

TEST\_F() tests can access objects that are members of the class.

例子可参考SerManager里面的Fixture\_test

Populate the class fixture definition as needed if you want to share setups between tests.

# 添加TEST\_F步骤

* Adding real tests:
  + You may rename unit\_test.cpp if you want so you can group related tests.
    - Make sure to update CMakeLists.txt if you rename or add additional unit test source files.
  + Add any necessary #include statements.
  + In unit\_test.cpp, rename the “TemplateTest” class to use a more descriptive name for your test.
  + Populate the class fixture definition as needed if you want to share setups between tests.
  + Remove the two generic TEST\_F() functions that came with the template.
  + Create new unit test functions using TEST\_F(), using the new name of your test class as the first parameter and a descriptive identifier of the test as the second parameter

例：TEST\_F(Fixture\_test, FirstAdd )

# 面向UnitTest的设计

Design for UnitTest

* Some code is easy to write unit test for:
  + An example is a function that takes all of its data as parameters, does a calculation, and returns a result without using global data.

理想化的函数： 参数输入-Process-返回值输出 多入单出系统

* Some code is harder to write unit tests for:
  + Testing private members of classes is harder since the private methods are not accessible by the test code.

类的私有成员

* + - Workarounds include changing making the test class a friend class or by conditional compilation to change the visibility.
  + Static functions are not exported to the test

文件作用域的私有函数

* + - One workaround is to use STATIC instead of the static keyword for the function prototype. During a target build, STATIC would be set to “static” and during the unit test STATIC would be “”. The unit test could then include an “extern” function prototype. See the AsBuiltInfo unit test for an example.

函数的定义和声明默认情况下是extern的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。

* + Some code is dependent on hardware or on functions that perform a complex or long operation

硬件依赖或执行复杂操作

* + - Workarounds could include creating “mock” classes that are used only for testing that simulate the complex classes. This is more of an advanced topic and is supported by GMock, but is outside the scope of this document.

Harder but not impossible(CAN NOT)， always has a workaround.

优化你的代码 以便于单元测试

* Try to compartmentalize your software.
* Put the logic that you want to unit test into its own function with few dependencies。 减少依赖
  + Example: if you have a feature that must read a file and then perform some action on the data that was read, move the processing of that data into a separate function from the file read operation and just pass in a pointer/copy of the data that was read.
  + That way, your unit test can be written to supply its own data instead of depending on the external files.
* The easiest functions to unit test involve only the data you pass into the function and output a result.

硬件依赖 昂贵操作 低概率故障 Mock化

* Create stub classes to replace hardware dependencies or to replace “expensive” operations, or to force conditions that would normally be hard to reproduce
  + Example: Create a stub class that produces MOST error messages.

Testable Design

FICC Properties

F: Fast

I: Isolated

C: Configuration-free

C: Consistent

If you think of tests as a user of your system, designing for testability becomes a way of thinking.

If you were doing test-driven development, you’d have no choice but to write a testable system, because in TDD the tests come first and largely determine the API design of the system, forcing it to be something that the tests can work with.

|  |  |  |
| --- | --- | --- |
| **Design Guideline** | **Benefit** | **Comments** |
| Make methods virtual by default | This allows you to override the methods in a derived  class for testing. Overriding allows for changing behavior or breaking a call to an external dependency. | virtual才能override，才好mock |
| Use interface-based designs | This allows you to use polymorphism to replace dependencies in the system with your own stubs or mocks. | 基于接口的设计 |
| Make classes nonsealed by default. | You can’t override anything virtual if the class is  sealed (final in Java). | No final |
| Avoid instantiating concrete classes inside methods with logic. Get instances of classes from helper methods, factories, inversion of control containers such as Unity, or other places, but don’t directly create them. | This allows you to serve up your own fake instances  of classes to methods that require them, instead of  being tied down to working with an internal produc-  tion instance of a class. | 不要在富含逻辑处理的方法中，实例化具体的类。 |
| Avoid direct calls to static methods. Prefer calls  to instance methods that later call statics. | This allows you to break calls to static methods by  overriding instance methods. (You won’t be able to  override static methods.) | 不要直接调用Static方法 |
| Avoid constructors and static constructors that  do logic. | Overriding constructors is difficult to implement.  Keeping constructors simple will simplify the job of  inheriting from a class in your tests. | 构造函数不要有复杂逻辑 |
| Separate singleton logic from singleton holders | If you have a singleton, have a way to replace its  instance so you can inject a stub singleton or reset it. | 单例模式下的测试 |

Use interface-based designs

Identifying “roles” in the application and abstracting them under interfaces is an important part of the design process.

An abstract class shouldn’t call concrete classes,

and concrete classes shouldn’t call concrete classes either, unless they’re data objects (objects holding data, with no behavior).

This allows you to have multiple seams in the application where you could intervene and provide your own implementation.

# Gtest事件机制

gtest提供了多种事件机制，非常方便我们在案例之前或之后做一些操作。事件一共有3种：

1. 全局的，所有案例执行前后。

2. TestSuite级别的，在某一批案例中第一个案例前，最后一个案例执行后。

3. TestCase级别的，每个TestCase前后。

## 全局事件

要实现全局事件，必须写一个类，继承testing::Environment类，实现里面的SetUp和TearDown方法。

1. SetUp()方法在所有案例执行前执行

2. TearDown()方法在所有案例执行后执行

当然，这样还不够，我们还需要告诉gtest添加这个全局事件，我们需要在main函数中通过testing::AddGlobalTestEnvironment方法将事件挂进来，也就是说，我们可以写很多个这样的类，然后将他们的事件都挂上去。

int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    testing::AddGlobalTestEnvironment(new FooEnvironment);  
    testing::InitGoogleTest(&argc, argv);  
    return RUN\_ALL\_TESTS();  
}

## TestSuit事件

我们需要写一个类，继承testing::Test，然后实现两个静态方法

1. SetUpTestCase() 方法在第一个TestCase之前执行

2. TearDownTestCase() 方法在最后一个TestCase之后执行

## TestCase事件

TestCase事件是挂在每个案例执行前后的，实现方式和上面的几乎一样，不过需要实现的是SetUp方法和TearDown方法：

1. SetUp()方法在每个TestCase之前执行

2. TearDown()方法在每个TestCase之后执行