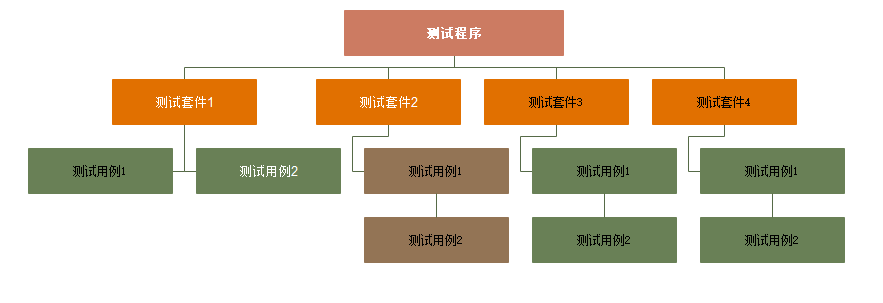
# gtest简介

gtest是一个**跨平台的**(Liunx、Mac OS X、Windows、Cygwin、Windows CE and Symbian)C++[单元测试](https://so.csdn.net/so/search?q=%E5%8D%95%E5%85%83%E6%B5%8B%E8%AF%95&spm=1001.2101.3001.7020)框架，由google公司发布。gtest是为在不同平台上为编写C++测试而生成的。它提供了丰富的断言、致命和非致命判断、参数化、“死亡测试”等等。

# gtest原理

gtest主要由一系列的宏和事件实现。

1. 宏：有TEST和TEST\_F宏，TEST宏针对简单的测试用例，TEST\_F宏针对需要做初始化和资源回收的测试用例，有点像类似C++的构造函数和析构函数，两个宏都是把参数展开后拼成一个类。
2. 事件：分为三种事件



测试程序：一个进程，全局事件在该层

测试套件：一系列测试用例的集合，SetUpTestCase事件在该层

测试用例：一个测试用例的事件体现在每个用例前后的SetUp和TearDown

每个事件可用于下级事件的数据共享以及测试前后的数据处理

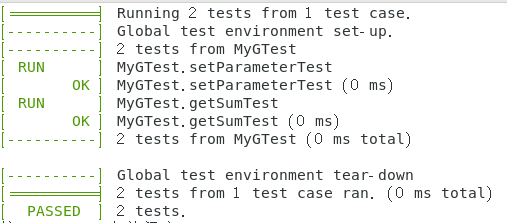
## gtest事件机制

“事件” 本质是gtest框架提供了一个机会, 可以使用这样的机会来执行定制的代码, 来给测试用例准备/清理数据。gtest提供了多种事件机制，总结一下gtest的事件一共有三种：  
**1、TestSuite事件**  
需要写一个类，继承testing::Test，然后实现两个静态方法：SetUpTestCase方法在第一个TestCase之前执行；TearDownTestCase方法在最后一个TestCase之后执行。

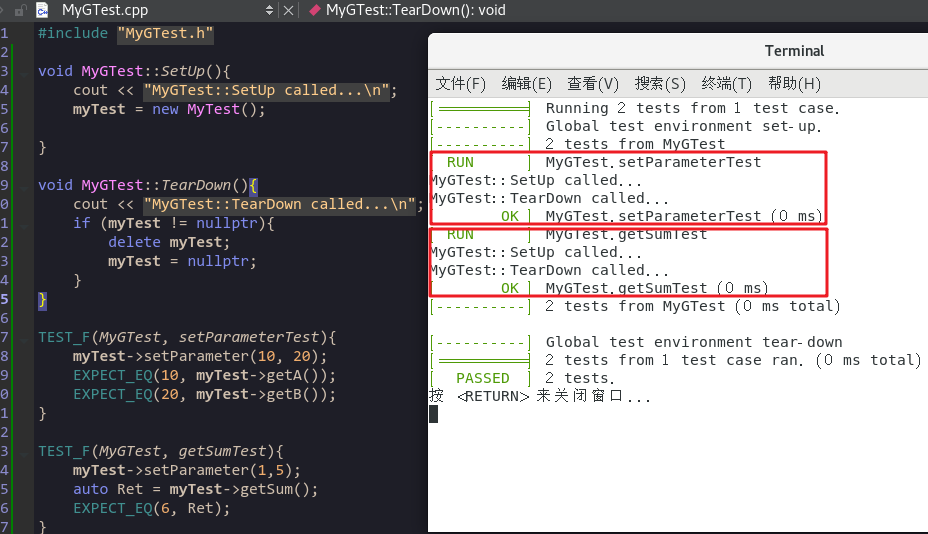
如下是一个例子：



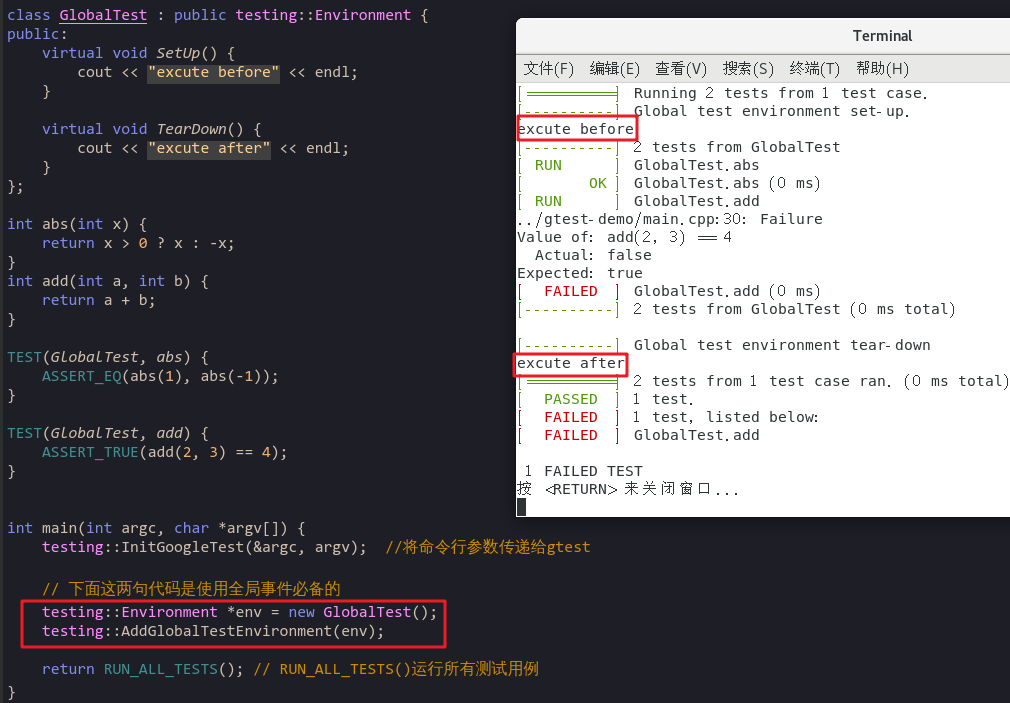
**运行结果如下：**

  
**2、TestCase事件**  
是挂在每个用例执行前后的，需要实现的是SetUp方法和TearDown方法。SetUp方法在每个TestCase之前执行；TearDown方法在每个TestCase之后执行。

在1中的SetUp和TearDown中添加打印语句：

  
**3、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。SetUp方法在所有测试用例执行前执行；TearDown方法在所有测试用例执行后执行。  
除了要继承testing::Environment类，还要定义一个该全局环境的一个对象并将该对象添加到全局测试环境中。

全局事件可以按照下列方式来使用：



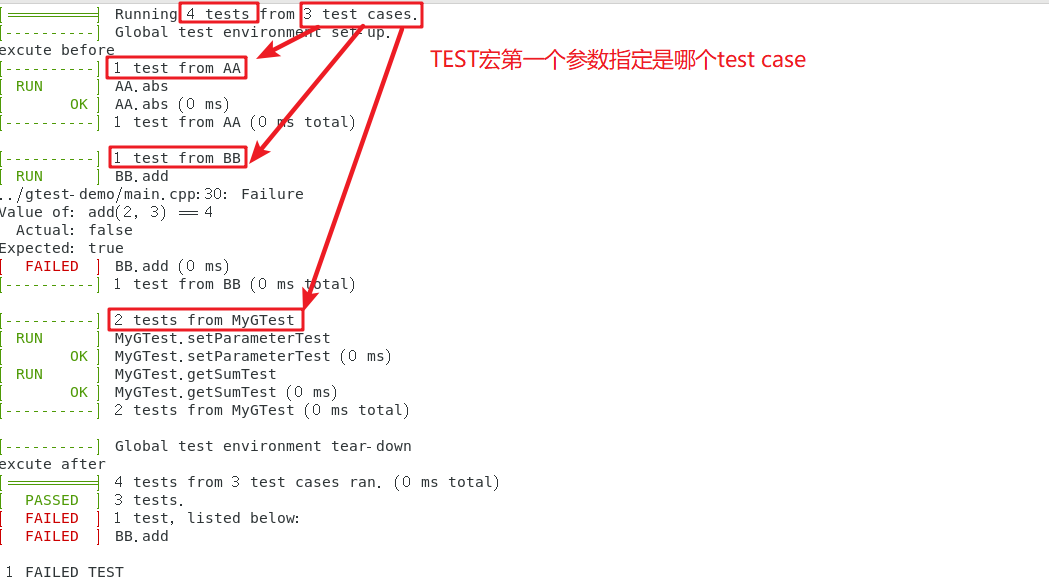
# gtest内置的宏

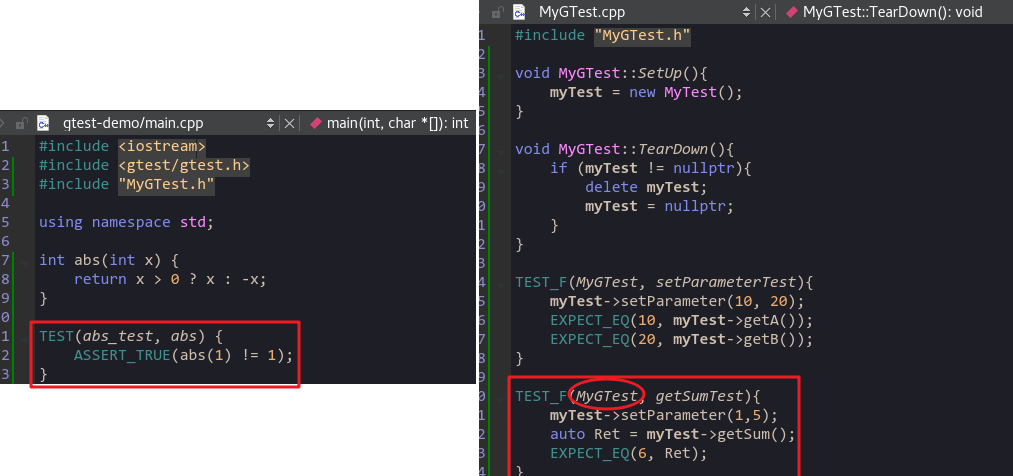
TEST(test\_case\_name, test\_name)

TEST\_F(test\_fixture,test\_name) //多个测试场景需要相同数据配置的情况，用TEST\_F。TEST\_F test fixture，测试夹具，测试套，承担了一个注册的功能（为测试类的类名）。

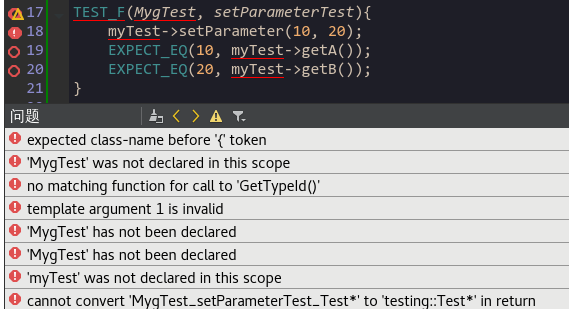
TEST\_F宏的使用需要按照C++的风格定义类，这个类需要继承自testing::Test并且重写这四个成员函数：SetUpTestCase、TearDownTestCase、SetUp、TearDown。如果有多个测试用例，那么SetUpTestCase就会在所有用例之前执行，TearDownTestCase会在所有用例之后执行；SetUp会在单个用例之前执行，TearDown会在单个用例之后执行。

【注意】使用TEST宏时，test\_case\_name不要求是当前测试类的类名，而使用TEST\_F宏时，第一个参数必须是当前测试类的类名。





TEST\_F宏第一个参数不是当前测试类的类名时，编译错误：



TEST宏的作用是创建一个简单测试，它定义了一个测试函数，在这个函数里可以使用任何C++代码并使用提供的断言来进行检查。

# gtest单元测试断言

gtest中断言的宏可以分为两类：一类是ASSERT宏，另一类是EXPECT宏。

1、ASSERT\_系列：如果当前点检测失败则退出当前函数  
2、EXPECT\_系列：如果当前点检测失败则继续往下执行

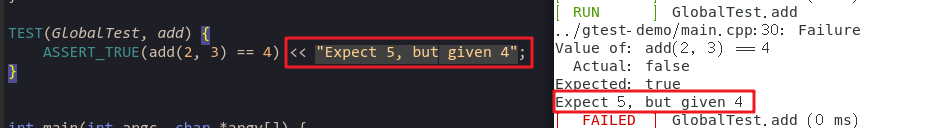
如果对自动输出的错误信息不满意的话，也可以通过operator<<在失败的时候打印日志，将一些自定义的信息输出。

常用ASSERT\_系列：

|  |  |  |  |
| --- | --- | --- | --- |
| **宏类型** | **宏名称** | **功能** | **描述** |
| bool值检查 | ASSERT\_TRUE(条件) | 判断条件是否为true | 期待结果是true |
| ASSERT\_FALSE(条件) | 判断条件是否为false | 期待结果是false |
| 数值型数据检查 | ASSERT\_EQ(参数1，参数2) | equal | 相等为true |
| ASSERT\_NE(参数1，参数2) | not equal | 不等于返回true |
| ASSERT\_LT(参数1，参数2) | less than | 小于返回true |
| ASSERT\_GT(参数1，参数2) | greater than | 大于返回true |
| ASSERT\_LE(参数1，参数2) | less equal | 小于等于返回true |
| ASSERT\_GE(参数1，参数2) | greater equal | 大于等于返回true |
| ASSERT\_FLOAT\_EQ(expected, actual) | equal | 两个float值相同返回true |
| ASSERT\_DOUBLE\_EQ(expected, actual) | equal | 两个double值相同返回true |
| 字符串检查 | ASSERT\_STREQ(expected\_str, actual\_str) | 判断字符串是否相等 | 两个C风格的字符串相等返回true |
| ASSERT\_STRNE(expected, actual) | 判断字符串不相等 | 两个C风格的字符串不相等时返回true |
| ASSERT\_STRCASEEQ(expected\_str, actual\_str) | 判断字符串是否相等(忽略大小写) |  |
| ASSERT\_STRCASENE(str1, str2) | 判断字符串不相等(忽略大小写) |  |

EXPECT\_系列，具有ASSERT系列类似的宏结构。

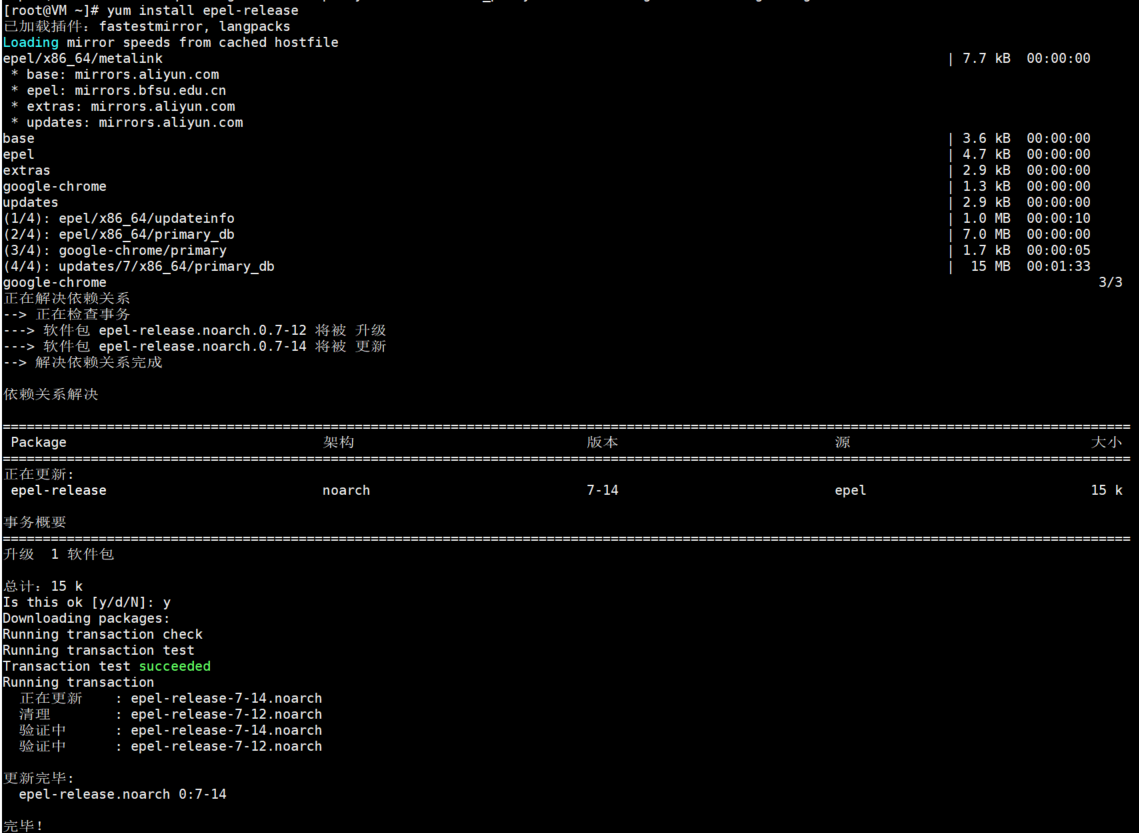
如下是自定义信息输出：



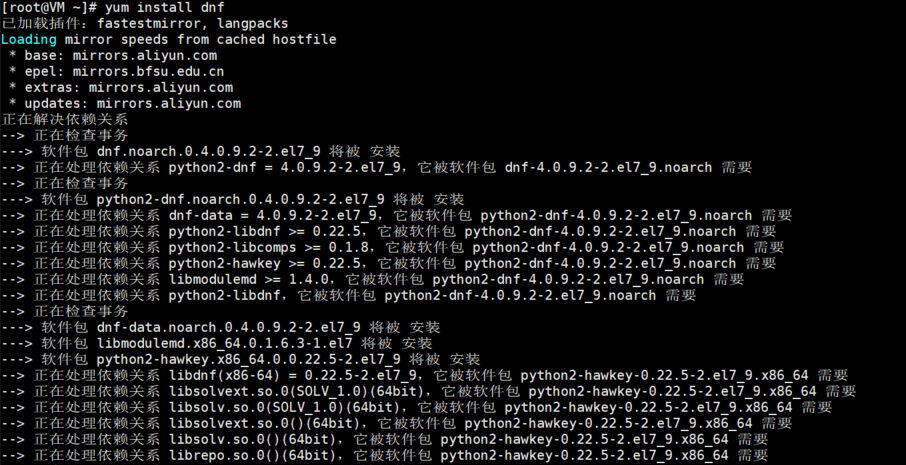
# gtest安装

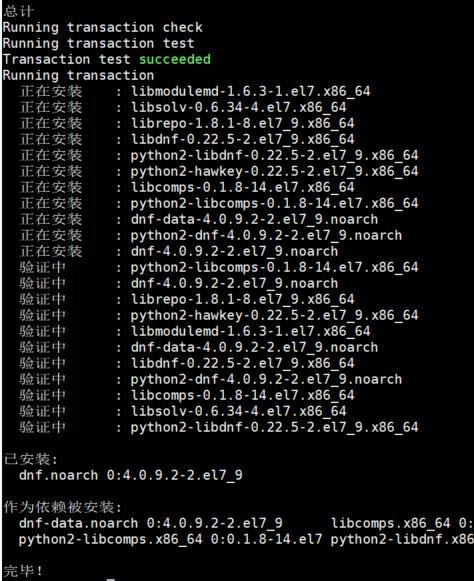
## 1、更新dnf源

sudo yum install epel-release



sudo yum install dnf

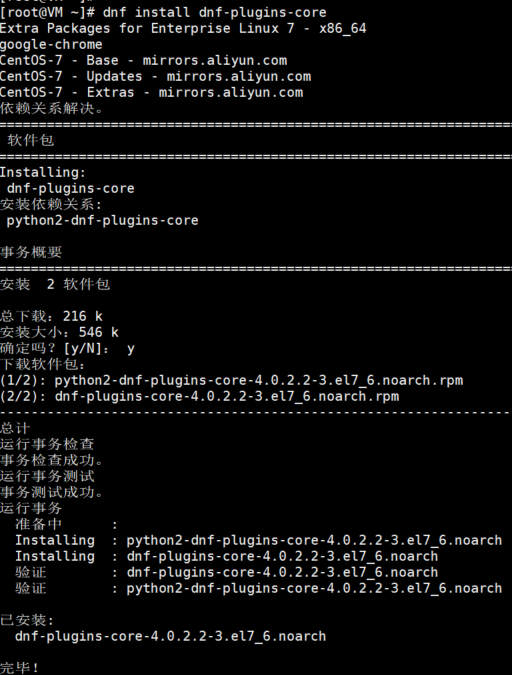




## 2、安装gtest

sudo dnf install dnf-plugins-core

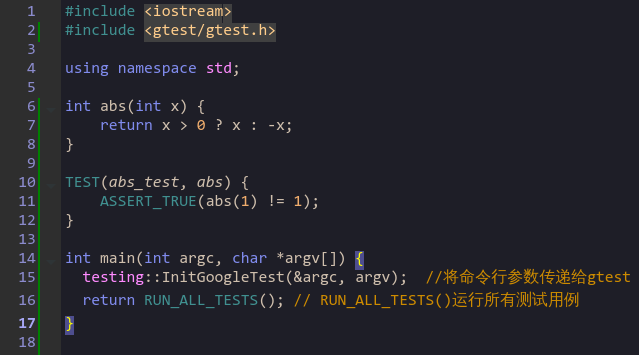
sudo dnf install gtest gtest-devel

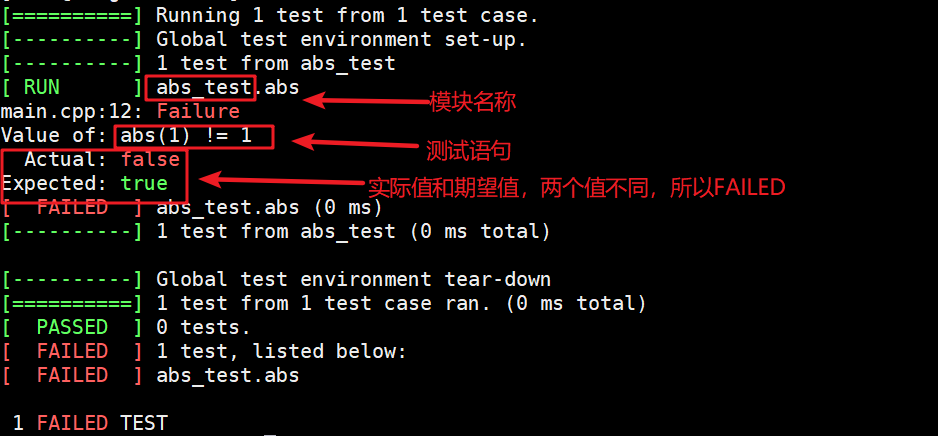




# gtest-demo

## 简单用法





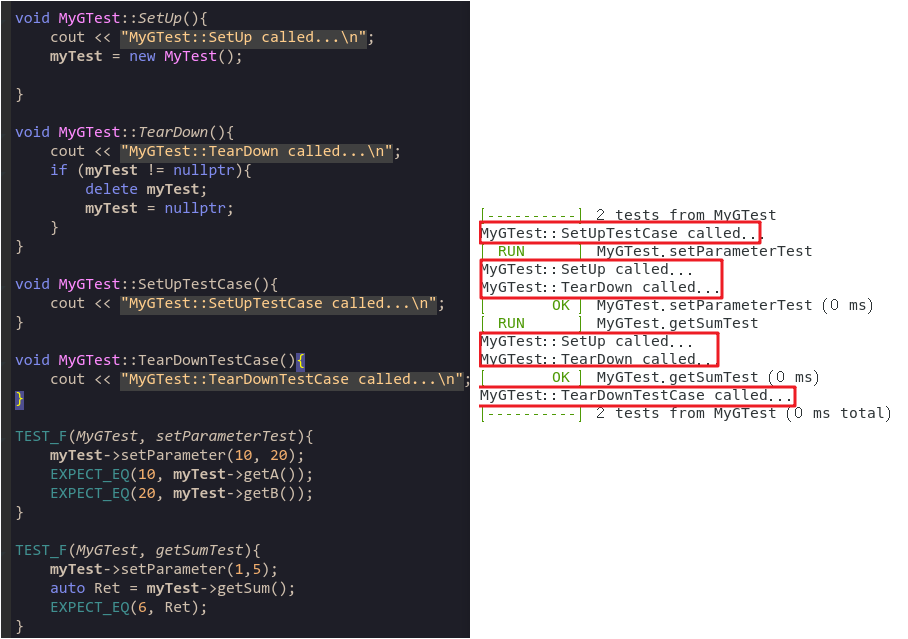
## 作为测试类使用

### 1、定义一个要被测试的类

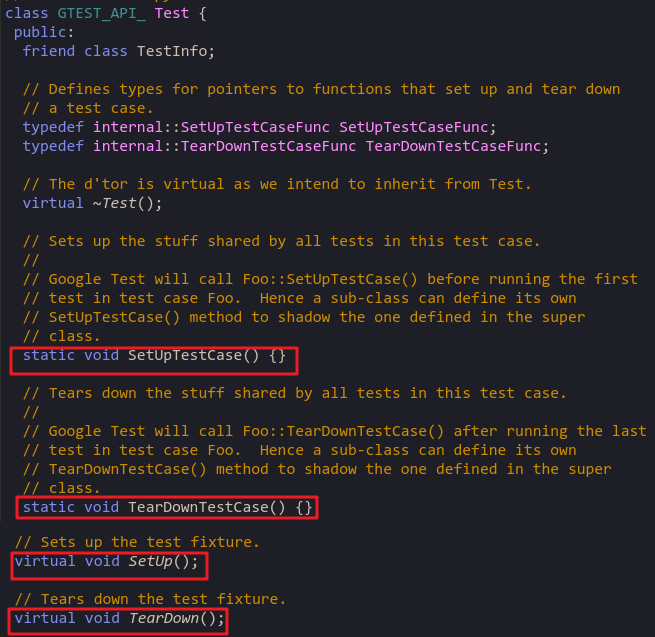


### 2、定义测试类

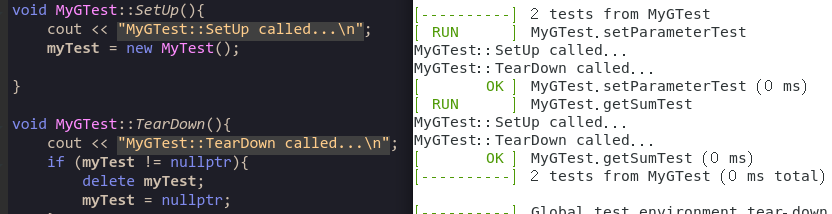
把要测试的类作为测试类的成员变量，重写SetUpTestCase、TearDownTestCase、SetUp、TearDown成员函数。

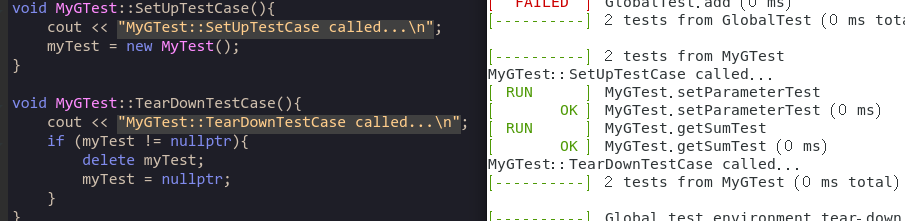


实测SetUpTestCase、TearDownTestCase、SetUp、TearDown四个函数只用重写其中的一对即可(如SetUp和TearDown、SetUpTestCase和TearDownTestCase均可)。可以按实际需求选择要重写的函数。注意SetUpTestCase和TearDownTestCase是static函数，SetUp和TearDown是虚函数。



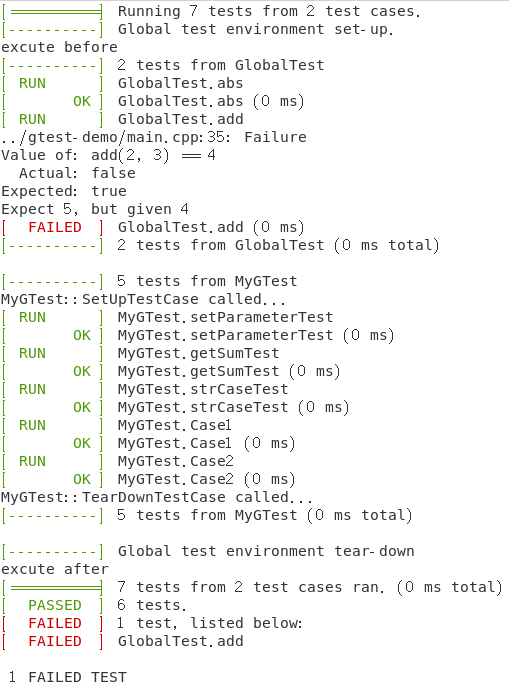
重写任意一对即可：





### 3、执行所有用例





# gtest测试类的私有成员

使用gtest编写单元测试，一个最常见的问题是对类私有成员的测试与验证。理想情况下，我们希望在测试中，类中所有的数据与方法都是可以访问的；而在产品代码中，只暴露实现定义好的接口。

[gtest官方文档](http://code.google.com/p/googletest/wiki/GoogleTestAdvancedGuide#Private_Class_Members)中也提到了对私有成员的处理，方法有两种：一是使用friend关键字；二是重构采用Pimpl模式，公共类中只暴露接口，而实现类中暴露所有细节(public)，测试时包含实现类即可。当然，还有另外一种简单暴力的方法，要达到的目的无非是在产品代码中，该pubic的是public，该private的还是private；而在测试代码中，全部都是public。加宏编译，将private/protected全部改为public。

## 1.加宏编译

在测试类头文件中添加：

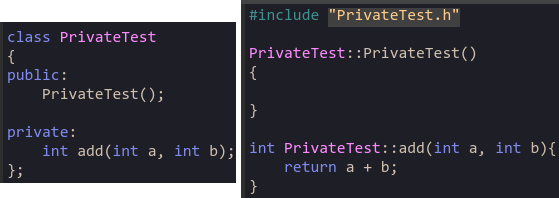
#ifdef GTEST\_API\_

#define private public

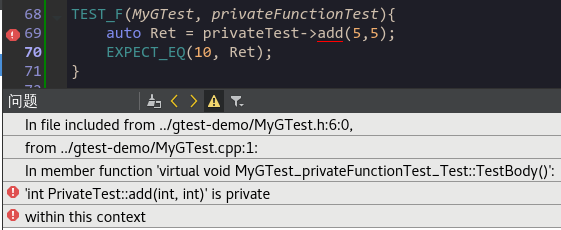
#define protected public

#endif

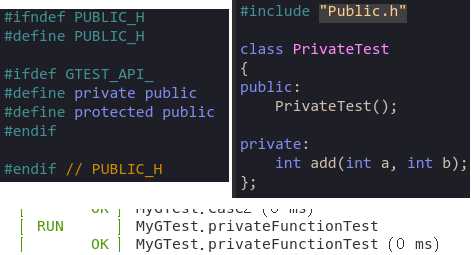
下面是一个测试private方法的例子：



如果直接测试，由于private方法无法被访问，编译失败



加入宏后，重新编译：



优点：可以非常方便的使用被测试类中的任何成员

缺点：一是使得访问私有成员过于容易，从而导致写出来的test访问私有成员的可能性增大。测试访问了私有成员，依赖于实现，这会让单元测试成为重构的负担，而不是保证。二是这种方法是不标准的C++用法: C++标准不允许重定义关键词，所以这种方法即使此时在当前的编译器上是可行的，也不能保证将来，或者在其他编译器上可行；public, protected private等访问修饰符可能会影响对象成员的布局，这样当测试是直接链接到产品代码时会有问题。

如果是以纯源代码的方式使用产品类的，这个方法没有任何问题；但如果是通过静态库，或者动态库的方式使用产品类的，在测试代码中若直接使用这些库，编译是没有问题，因为全部伪装成了public，但在链接的时候，因为private的成员是没有从库中导出来的，必然会出现链接错误。此时有两个方案：一是以GTEST的方式重新编译库；二是直接将产品源代码编译进你的测试工程中去。

所以，除非对以上利弊十分清楚并且可以接受，不然，还是使用官方的FRIEND\_TEST要更好一些。

## 2.FRIEND\_TEST

摘自官方文档：

Private class members are only accessible from within the class or by friends. To access a class' private members, you can declare your test fixture as a friend to the class and define accessors in your fixture. Tests using the fixture can then access the private members of your production class via the accessors in the fixture. Note that even though your fixture is a friend to your production class, your tests are not automatically friends to it, as they are technically defined in sub-classes of the fixture.( 类的私有成员只能从类内部或友元函数访问。要访问类的私有成员，可以将测试夹具声明为类的友元，并在fixture中定义访问器。然后，使用fixture的测试可以通过fixture中的访问器访问生产类的私有成员。请注意，即使你的fixture是生产类的友元，你的测试也不会自动成为它的友元，因为它们是在fixture的子类中定义的。)

FRIEND\_TEST(TestCaseName, TestName);

展开后如下：

#define FRIEND\_TEST(test\_case\_name, test\_name)\

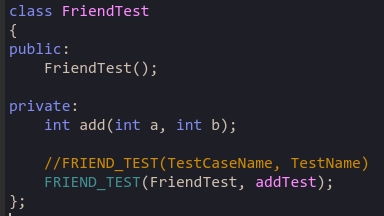
friend class test\_case\_name##\_##test\_name##\_Test

gtest通过FRIEND\_TEST(TestCaseName, TestName)声明友元类，即TestCaseName为友元类，而TestName为具体的case名。

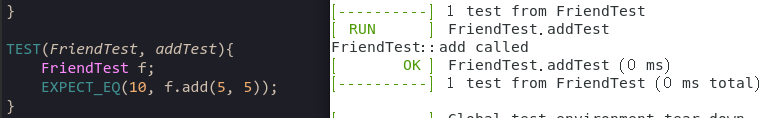
要使用FRIEND\_TEST，在被测试的类中添加：

FRIEND\_TEST(TestCaseName, TestName)

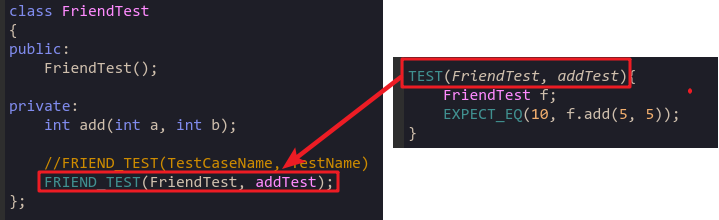
如：



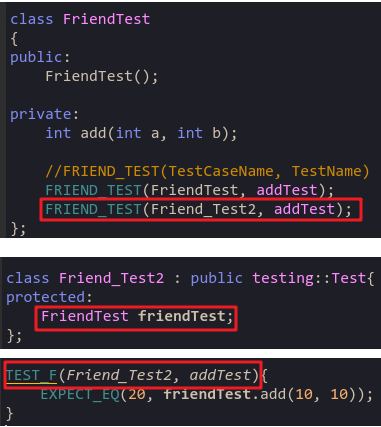
在测试类中包含被测试类的头文件，使用友元调用被测试类的private函数：



注意，对FRIEND\_TEST声明的友元类测试时，当使用TEST宏，TEST宏的两个参数必须和FRIEND\_TEST的宏参数保持一致。



或者，也可以使用TEST\_F宏



上例中给了两种访问私有变量的方式：

**一种是FRIEND\_TEST(FriendTest, addTest)声明的类，然后通过TEST(FriendTest, addTest)实现测试特例实体，实体中需要使用FriendTest的构造函数构造一个对象，然后调用其私有的add方法。**

**一种是FRIEND\_TEST(Friend\_Test2, addTest)声明的类。它使用到了Test Fixtures技术。在Friend\_Test2类中，声明了一个FriendTest对象。然后在TEST\_F(Friend\_Test2, addTest)中直接使用了Friend\_Test2类中的成员变量调用add方法。TEST\_F(Friend\_Test2, addTest)既继承于Friend\_Test2，又是FriendTest的友元类。**

缺点：

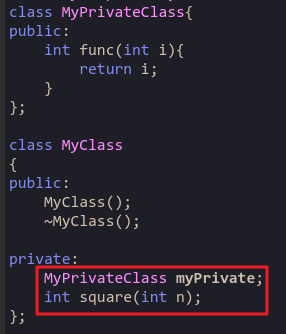
需要通过FRIEND\_TEST宏来新增该类的友元类（就是通过友元类来访问私有成员，所以这是一种侵入式的测试——修改了原来类的结构）。

## 3.使用pimpl重构被测代码

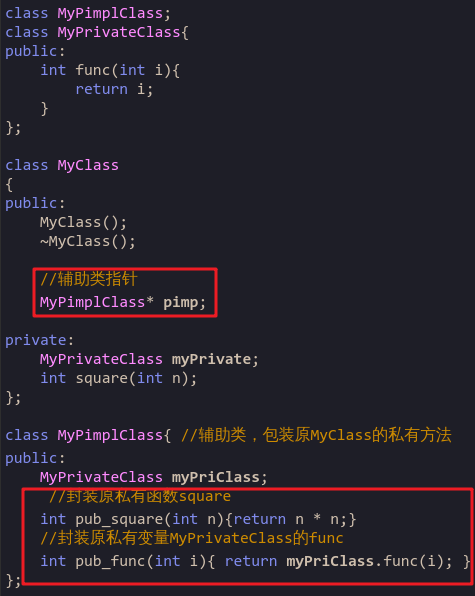
Pimpl主要的作用是解开代码调用接口和具体实现的耦合，具体的实现可以参考[C++程序设计机制——PIMPL机制](https://www.ngui.cc/article/show-347898.html)。利用这种方式打开私有成员，思路是先对被测代码按照pimpl模式进行重构，把具体实现的私有成员全部放到另外一个辅助类，通过指针访问实现的私有成员，然后对这个辅助类进行原类的私有成员测试。

原被测类如下：

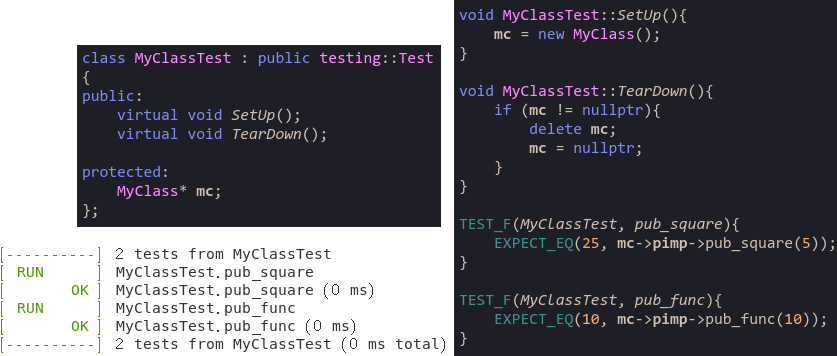
//MyClass.h



修改后的MyClass为：



通过辅助类进行测试：



经过上面的处理，调用新的MyClass类，就可以访问之前的私有成员了，pub\_square ()对应原来的square ()，pub\_func ()对应原MyClass私有成员MyPrivateClass的func ()。也就可以通过调用新的MyClass来访问其原私有成员进行测试了。

      缺点：实现比较复杂。如果采用此方法，需要开发编码时就采用该模式，或者提测后人工对代码进行重构，工作量比较大。

如果只是为了支持测试而做这样的重构，可能没有太大意义。况且，手工重构很麻烦。