# gmock简介

gmock是google公司推出的一款开源的白盒测试工具。gmock是个很强大的东西，测试一个模块的时候，可能涉及到和其他模块交互，可以将模块之间的接口mock起来，模拟交互过程。其作用就类似白盒测试中的打桩的概念。

下面几个例子说明了打桩在白盒测试中的重要性：

1、比如银行业务，需要测试业务模块。此时，不可能去操作真实的数据库，并且搭建新的数据库可能比较复杂或者耗时。那么就可以用gmock将数据库接口打桩，来模拟数据库操作。

2、比如要测试A模块，A模块需要调用B模块的函数。如果B模块还没有实现，此时，就可以用gmock将B模块的某些接口打桩。这样就可以让A模块的测试继续进行下去。

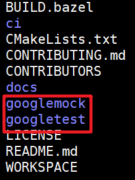
3、比如网关设备，在用gtest测试device模块的时候，必须有真实的设备才能让测试进行下去。如果用gmock模拟一套sdk接口，那么无需真实的设备也能让测试进行下去。

# 安装gmock

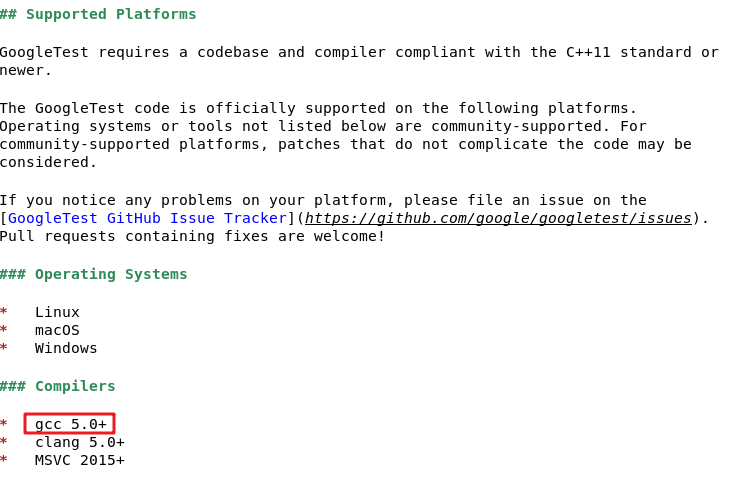
gmock是google的开源工具，google的官网上是可以下载源码,从github也可以获取到源码。

git clone https://github.com/google/googletest.git

下载后内容如下：

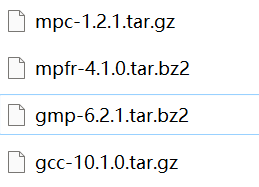


可以看到里面既有gtest也有gmock。大部分情况下，gmock是配合gtest来做白盒测试的。注意编译该项目时，要求gcc版本在5.0以上



因为自己机器的gcc是4.8，所以把gcc也升级了。这里用的gcc版本是10.1

## 1.准备gcc相关源码包



## 2.安装gcc10

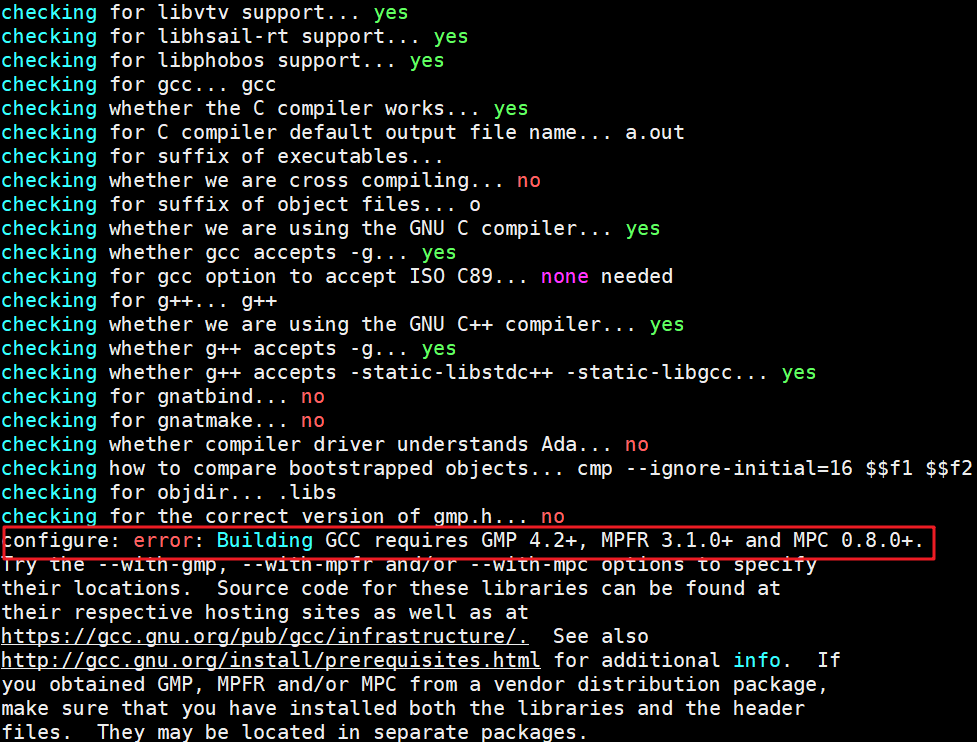
解压源码包

tar zxvf gcc-10.1.0.tar.gz

mkdir build

cd build

../configure --prefix=/usr/local/gcc-10.1.0/ --enable-checking=release --enable-languages=c,c++ --disable-multilib



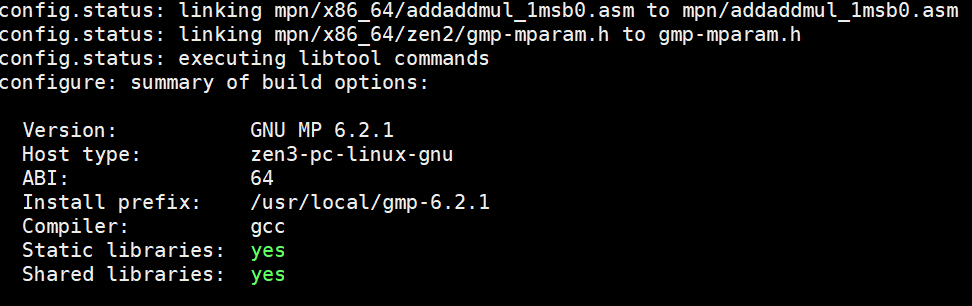
配置错误，提示需要GMP 4.2+，MPFR 3.1+，MPC 0.8+

## 3.安装GMP 6.2.1

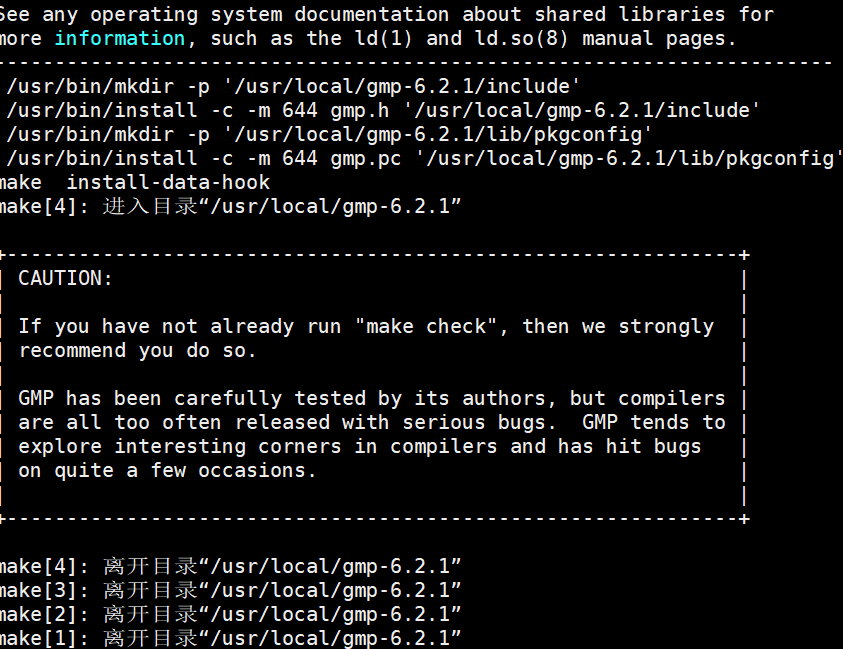
tar jxf gmp-6.2.1.tar.bz2

mv gmp-6.2.1 /usr/local/

./configure --prefix=/usr/local/gmp-6.2.1



make && make install

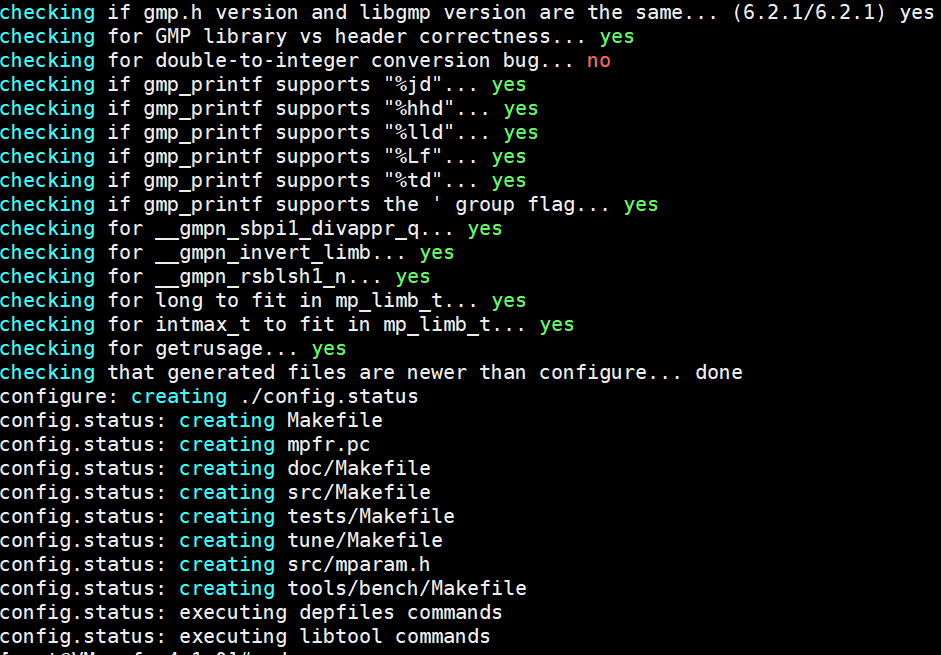


## 4.安装MPFR 4.1.0

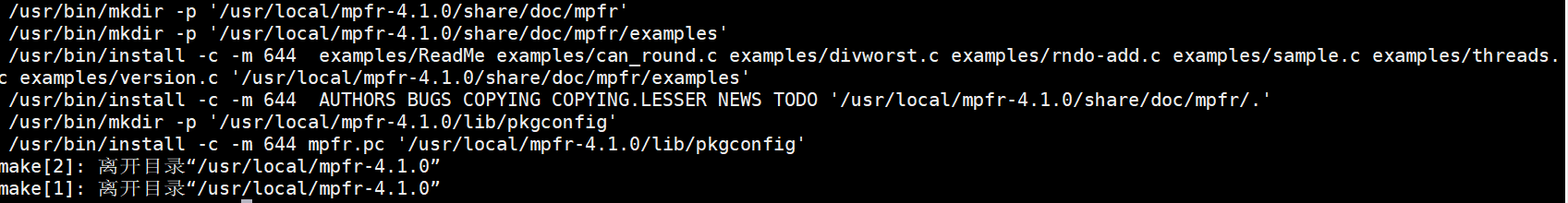
tar jxf mpfr-4.1.0.tar.bz2

mv mpfr-4.1.0 /usr/local/

./configure --prefix=/usr/local/mpfr-4.1.0 --with-gmp=/usr/local/gmp-6.2.1



make && make install

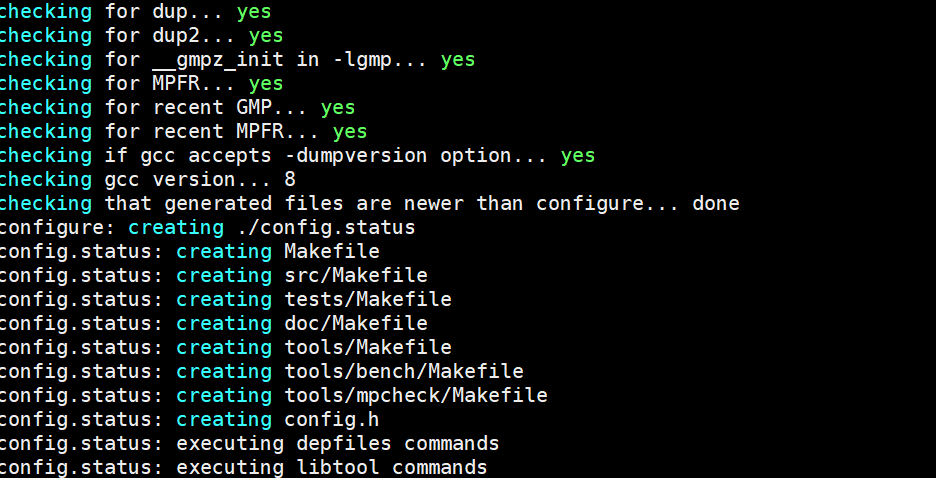


## 5.安装MPC 1.2.1

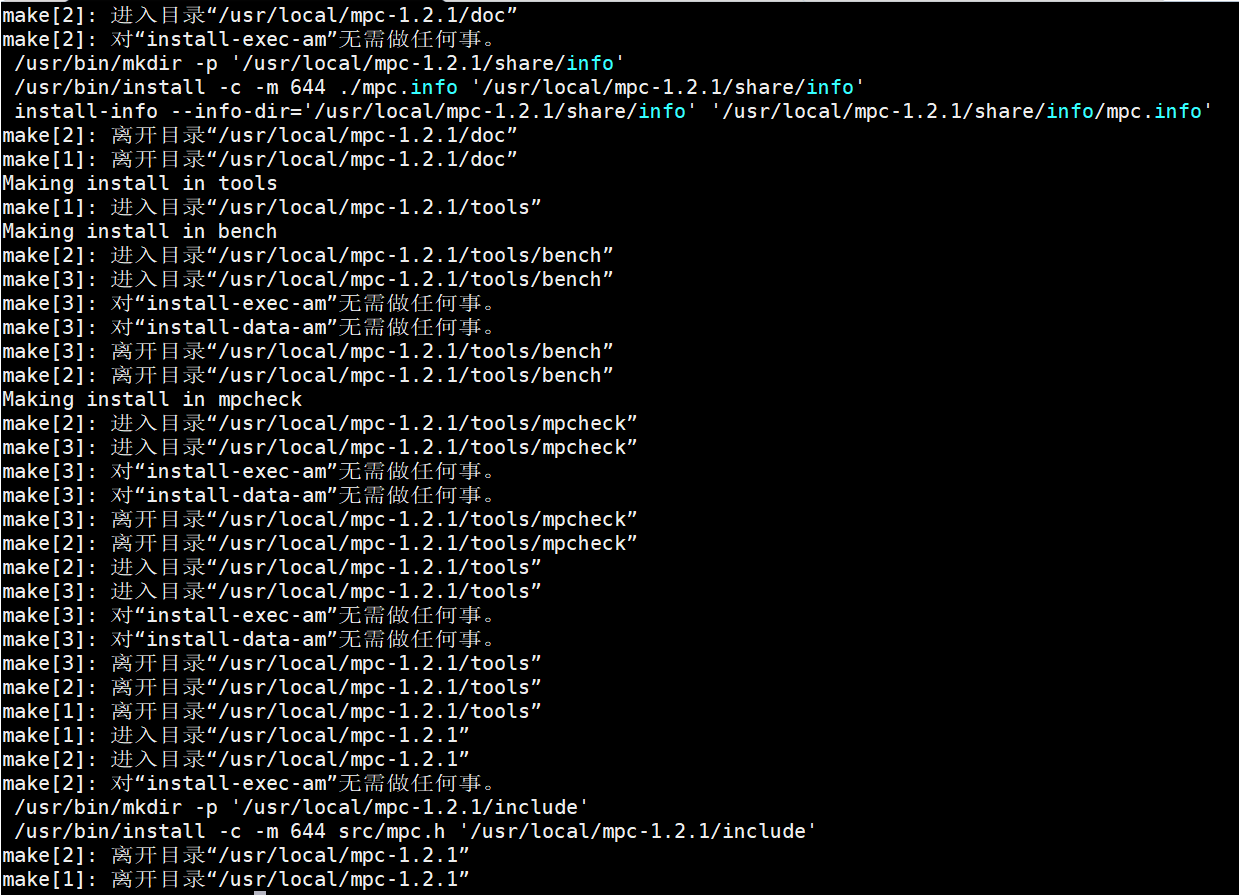
tar zxf mpc-1.2.1.tar.gz

mv mpc-1.2.1 /usr/local/

./configure --prefix=/usr/local/mpc-1.2.1 --with-gmp=/usr/local/gmp-6.2.1 --with-mpfr=/usr/local/mpfr-4.1.0



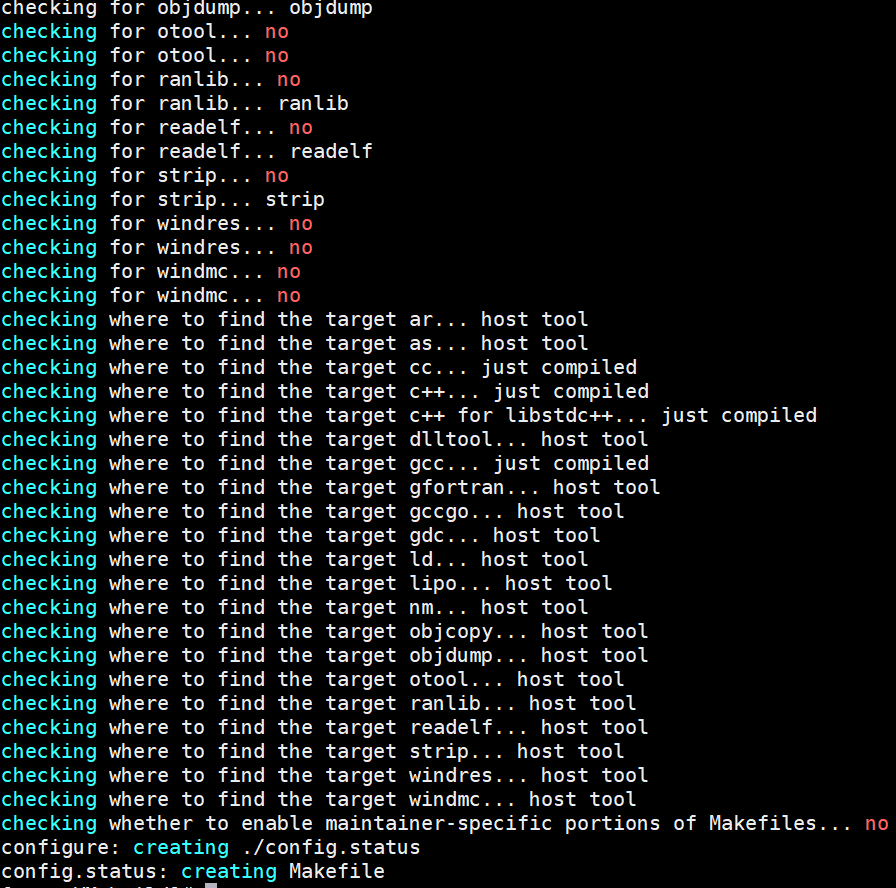
make && make install



## 6.重新安装gcc10

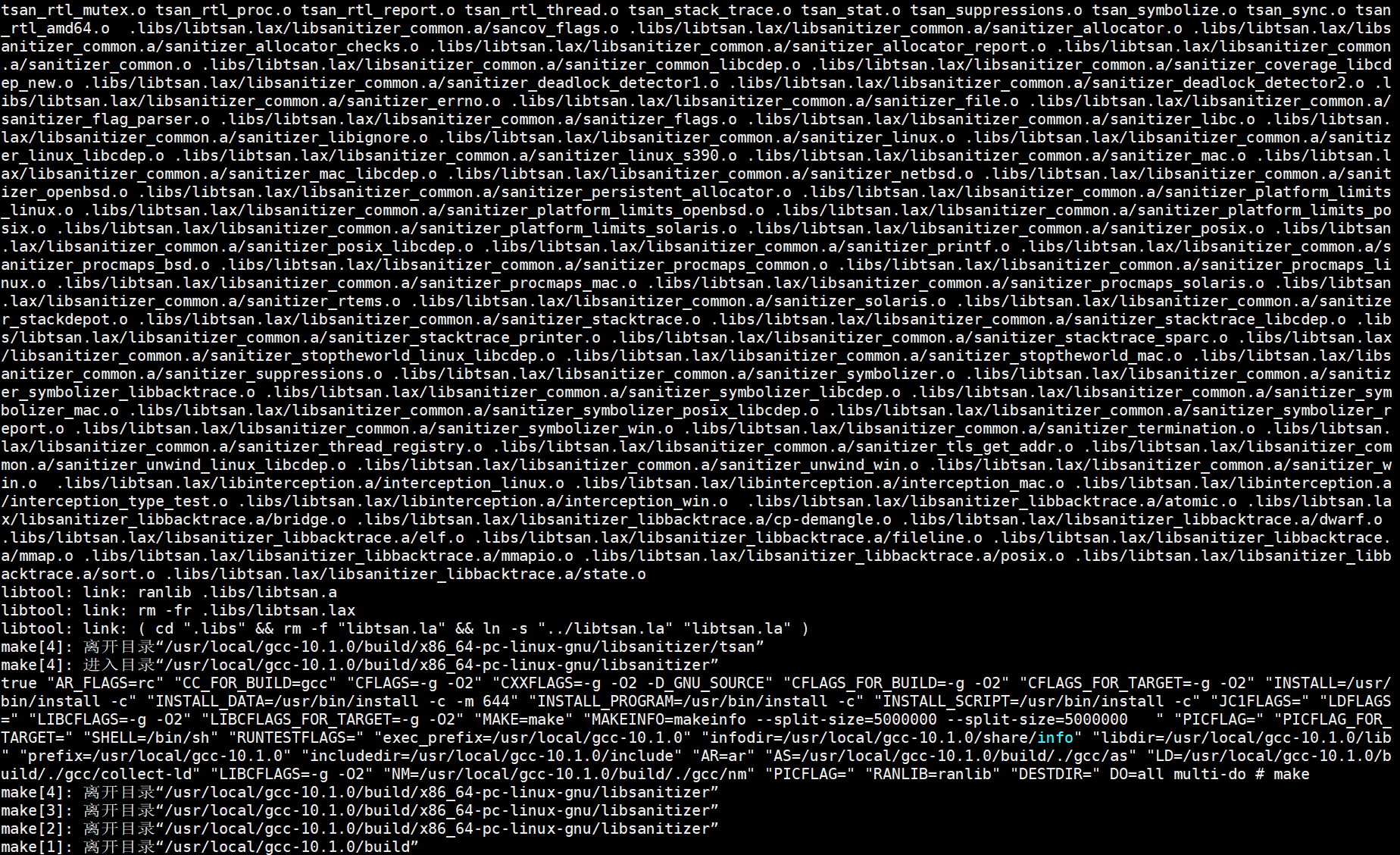
cd /usr/local/gcc-10.1.0/

./configure --prefix=/usr/local/gcc-10.1.0/ --enable-checking=release --enable-languages=c,c++ --disable-multilib --with-gmp=/usr/local/gmp-6.2.1 --with-mpfr=/usr/local/mpfr-4.1.0 --with-mpc=/usr/local/mpc-1.2.1

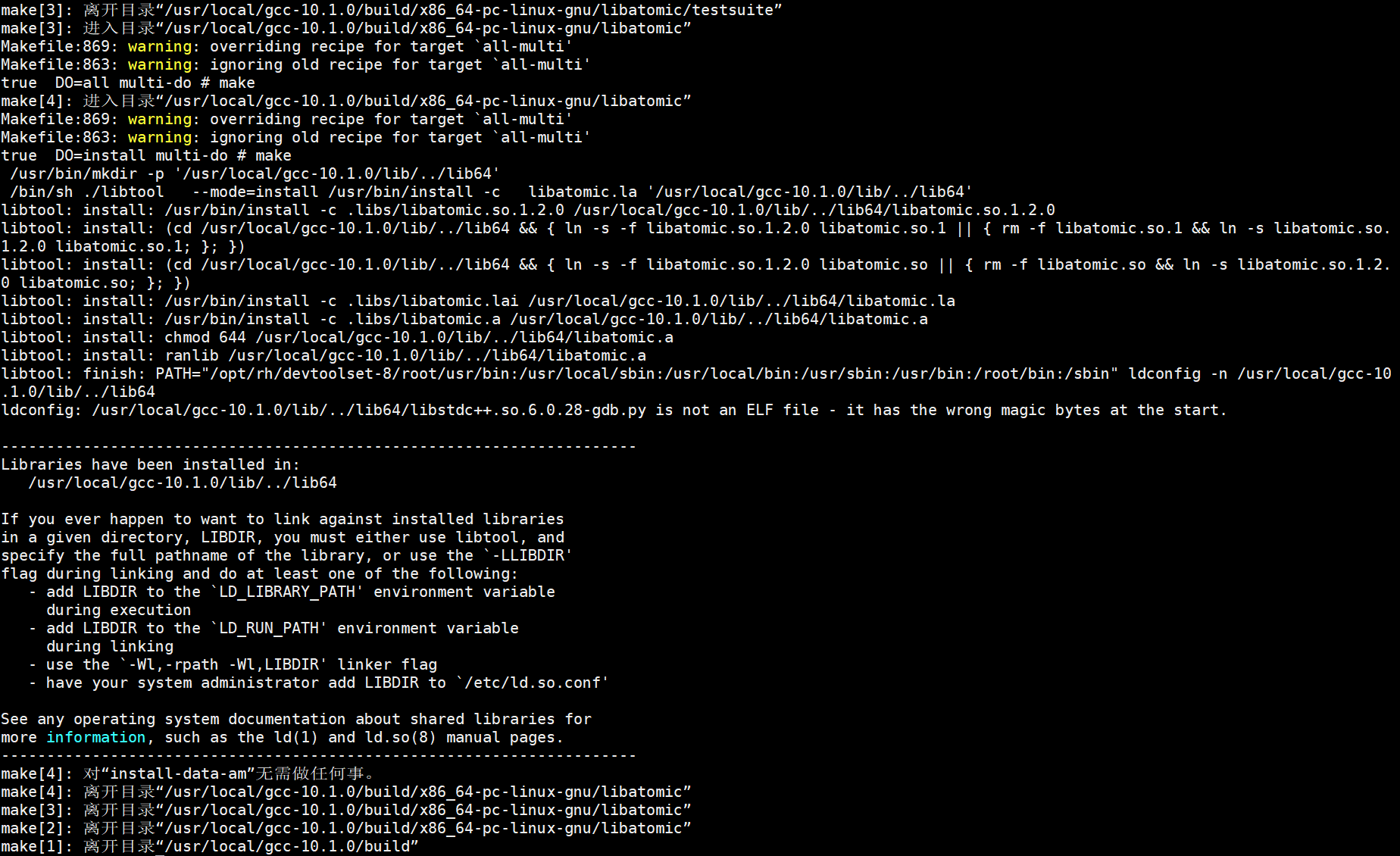


编译源码，安装。此处使用并行编译

make -j8



make install



## 7.gcc版本更新

cd /usr/local/bin

mv /usr/local/bin/gcc /usr/local/bin/gcc493

mv /usr/local/bin/g++ /usr/local/bin/g++493

mv /usr/local/bin/c++ /usr/local/bin/c++493

mv /usr/local/bin/cpp /usr/local/bin/cpp493

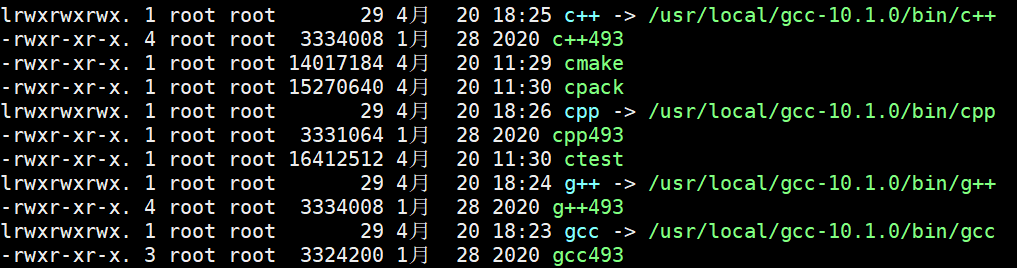
创建新版本软连接：

ln -s /usr/local/gcc-10.1.0/bin/gcc /usr/local/bin/gcc

ln -s /usr/local/gcc-10.1.0/bin/g++ /usr/local/bin/g++

ln -s /usr/local/gcc-10.1.0/bin/c++ /usr/local/bin/c++

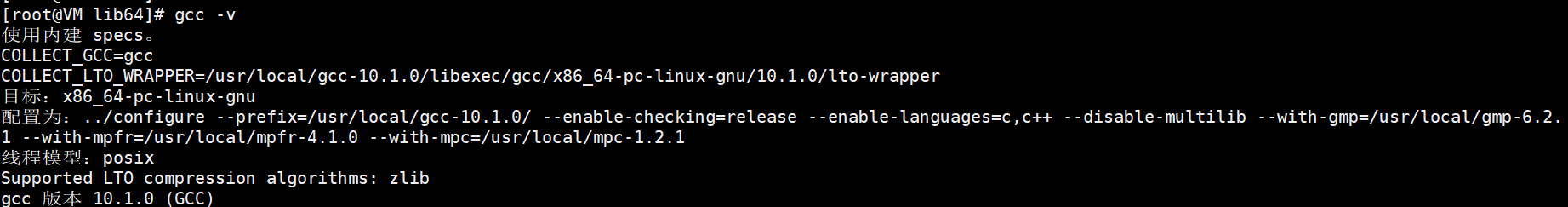
ln -s /usr/local/gcc-10.1.0/bin/cpp /usr/local/bin/cpp



mv /usr/lib64/libstdc++.so.6 libstdc++.so.6.bak

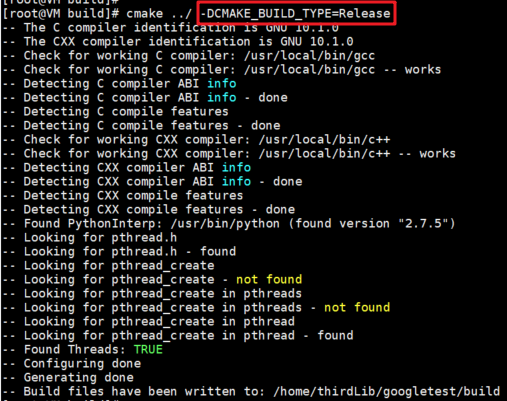
ln -s /usr/local/gcc-10.1.0/lib64/libstdc++.so.6.0.28 /usr/lib64/libstdc++.so.6

执行成功之后就可以查看当前使用的gcc版本了

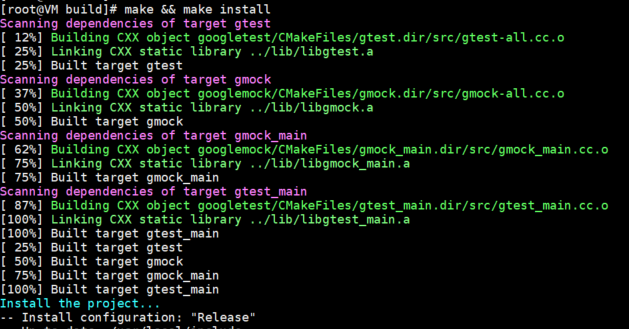


## 8.编译gmock

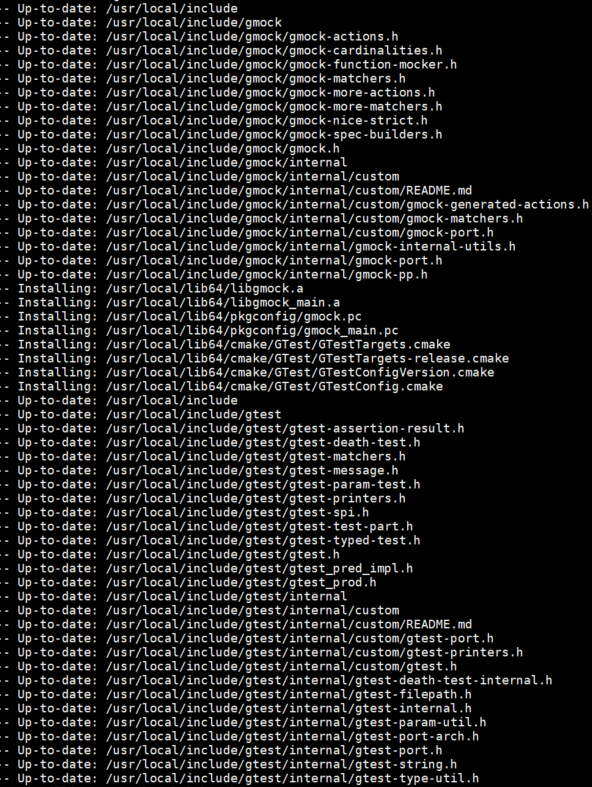
生成Makefile



编译安装



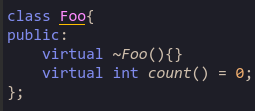
头文件也会被自动配置到对应目录下：



# gmock实例

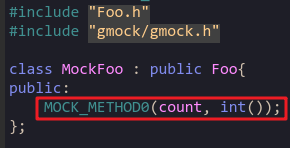
假设有一个抽象类Foo

//Foo.h

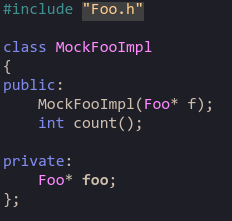


现在要测试其中的纯虚函数count(),不能直接测试。借助gmock的MOCK\_METHOD0，通过gmock来mock（模拟）Foo的方法。

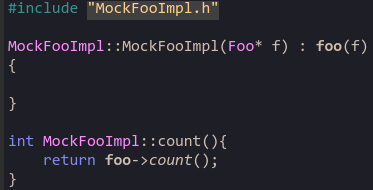
//MockFoo.h



// MockFooImpl.h

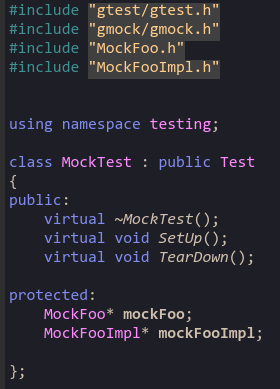


// MockFooImpl.cpp

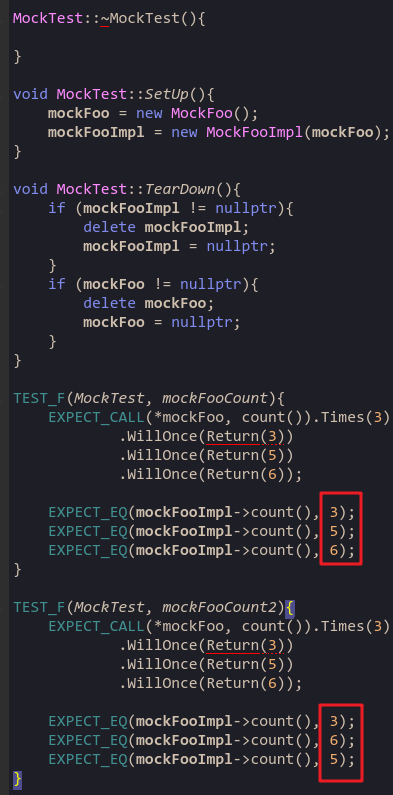


对模拟出来的类进行测试

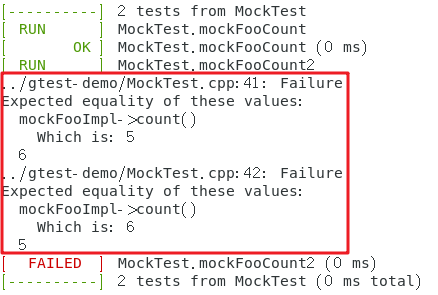
//MockTest.h



//MockTest.cpp



运行结果



可以看到，对count的三次调用，返回结果依次是3,5,6。修改一下顺序就会导致测试失败。

# 深入理解mock的定义

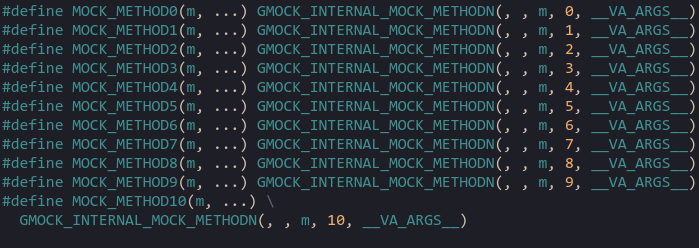
在单元测试、模块的接口测试时，当一个模块需要依赖另外一个或几个类，而这时所依赖的类还没有开发好，就可以定义Mock对象来模拟那些类的行为。也就是自己实现一个假的依赖类，对这个类的方法想要什么行为就可以有什么行为，想让这个方法返回什么结果就返回什么样的结果。（便捷的模拟对象的方法。）

gmock 依赖C++的多态机制进行工作，只有虚函数才能被mock, 非虚函数不能被mock, 这一点告诉我们，如果想要在代码中使用gmock，那么在类的设计中，最好采用接口隔离，对于c++来说也就是采用纯虚类型，因为c++本身没有接口类型。

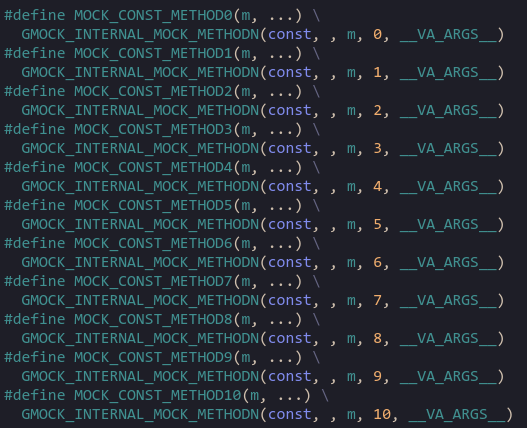
当一个文件中多个方法需要调用一个未完成或者虚函数做单元测试时，就需要将这个未完成或者使用比较麻烦的虚方法Mock，因为不希望为了执行单元测试，专门搭建一个服务器与XXXClient交互，成本太高。只需要Mock这个被调用的虚方法，调用方法不需要Mock，如果没有返回值可以使用SetArgReferee<0>()，参数从0开始。

# gmock常用的宏

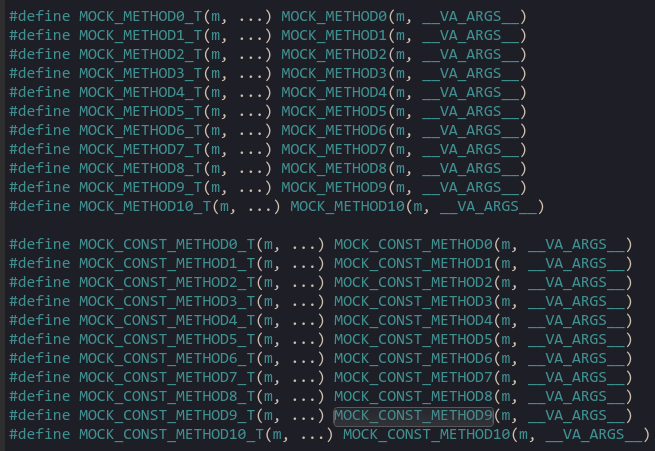
**MOCK\_METHOD系列**



MOCK\_METHODx系列宏，顾名思义，用于模拟（mock）函数方法的，但是它仅限于模拟类成员方法。  
    1. 其中x表示参数的个数。  
    比如我们要mock一个类成员函数：int foo(int param)，那么声明mock的时候会使用MOCK\_METHOD1宏，因为这个函数只有一个参数。如果有10个参数，那么就使用MOCK\_METHOD10。Gmock最大只支持10个参数的函数mock。如果函数超出10个参数，首先建议重构你的代码，因为函数参数最好不要超过10个。如果无法避免参数过多的情况，可以将几个有意义的参数作为一个组，并将它们聚合到一个结构中，然后将该结构的实例作为参数传递给该方法。  
    2. 这一系列宏接收两个参数，第一个参数为函数名，第二个参数为函数类型。  
    函数类型：函数原型剔除掉函数名剩下的部分，如int strcmp(const char cmp0, const char \*cmp1)这个函数，它的函数类型就是这个原型剔除掉函数名strcmp以后剩下的部分：int(const char \*cmp0, const char \*cmp1)，由于在函数声明和类型声明的时候，具体的参数是可以不给出的，只需给出类型，所以这个strcmp的函数类型可以简写为：int(const char\*, const char \*)。  
    3. Mock的函数是const类型时，需要使用MOCK\_CONST\_METHODx系列宏。



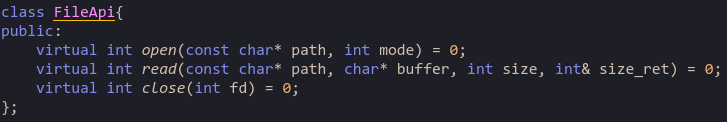
类似的，gmock中还提供了对模板类Template的mock。用到的宏是MOCK\_METHODx\_T、MOCK\_CONST\_METHODx\_T。

  
    4. 使用了MOCK\_METHODx宏声明函数的效果。  
使用MOCK\_METHODx宏声明的类成员函数，可以使用EXPECT\_CALL等一系列宏来指定这些函数在被调用时的动作。比如你期望函数无论何时在被调用时都返回0，又比如你期望函数在被调用时设置函数第二个参数的值为一个你指定的值（函数第二个参数是一个输出参数，引用类型或者指针），又比如你期望函数在第n次被调用时给函数第二个参数设置一个特殊的值等。这就是mock的真实存在：我们在做一个类的单元测试时，这个类所依赖的其他类的具体实现我们无需关心，我们只关心我们调用的那些类函数返回各种值的时候，我们要测试的这个类做出正确的处理。

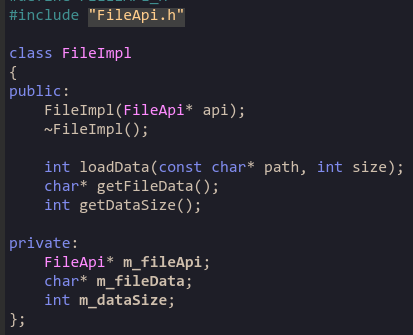
# mock抽象类

假设有一个对文件操作(打开、读取、关闭)的抽象类FileApi，类FileImpl里面的loadData调用了抽象类的接口，但抽象类的方法还没有实现，现在要对loadData进行测试。

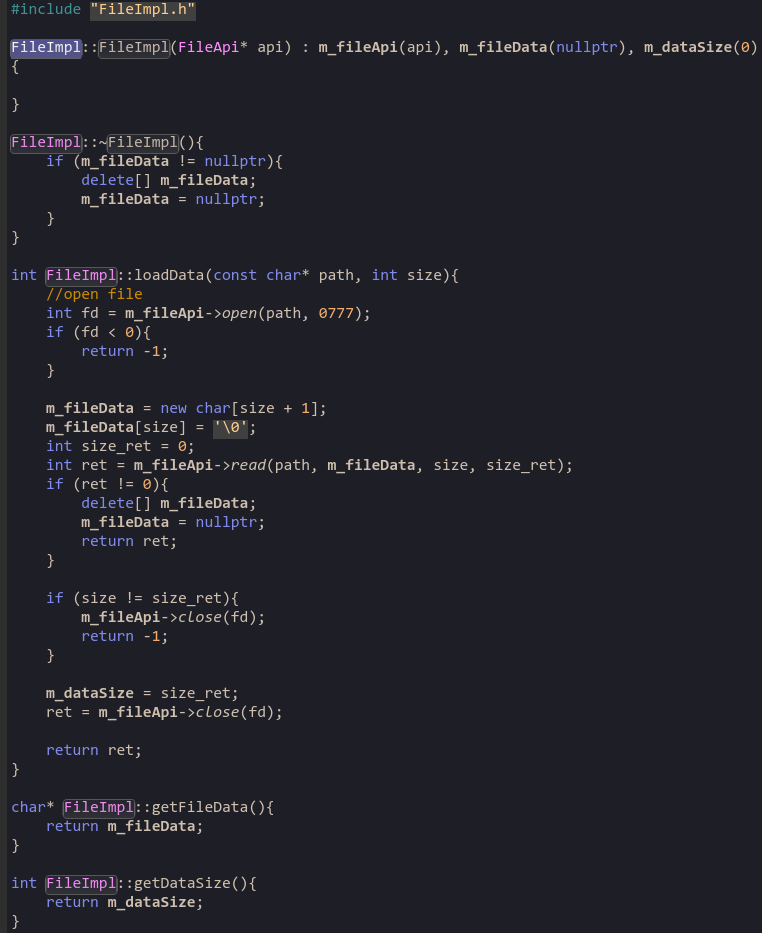
//FileApi.h



//FileImpl.h

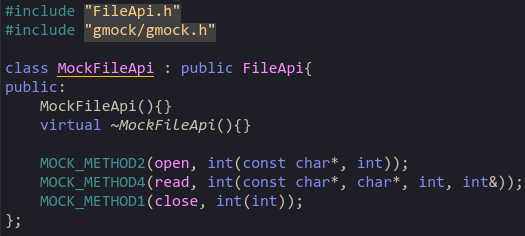


//FileImpl.cpp

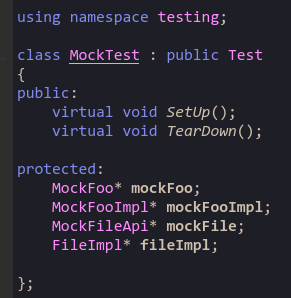


现在，mock一个类MockFileApi，去模拟抽象类的功能。

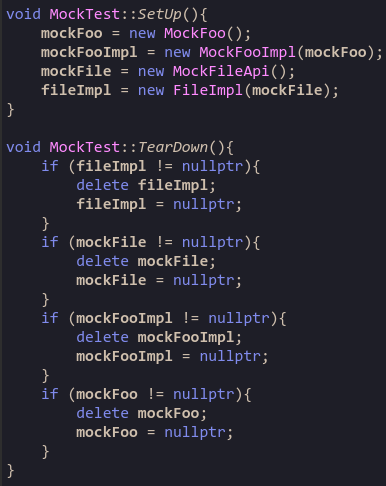
//MockFileApi.h



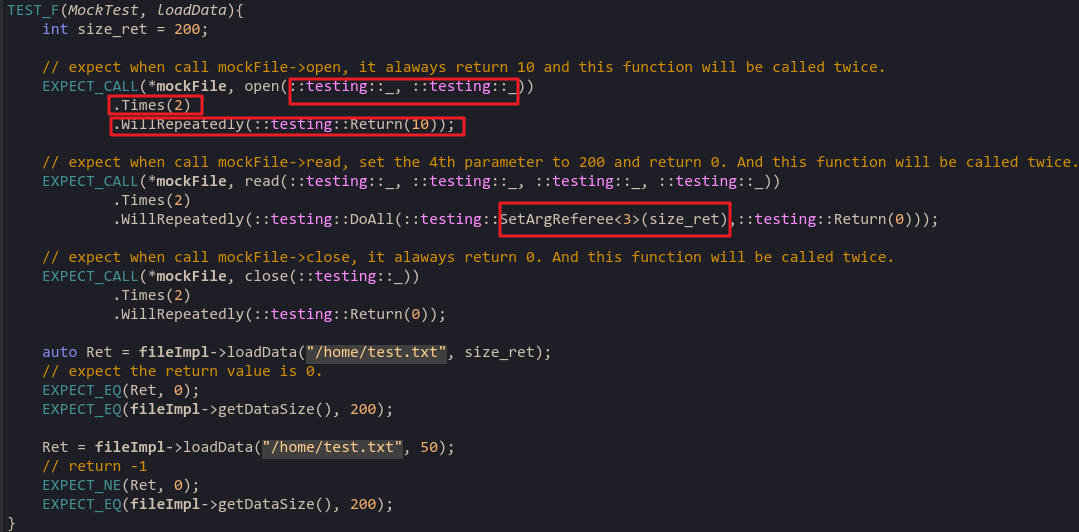
//MockTest.h



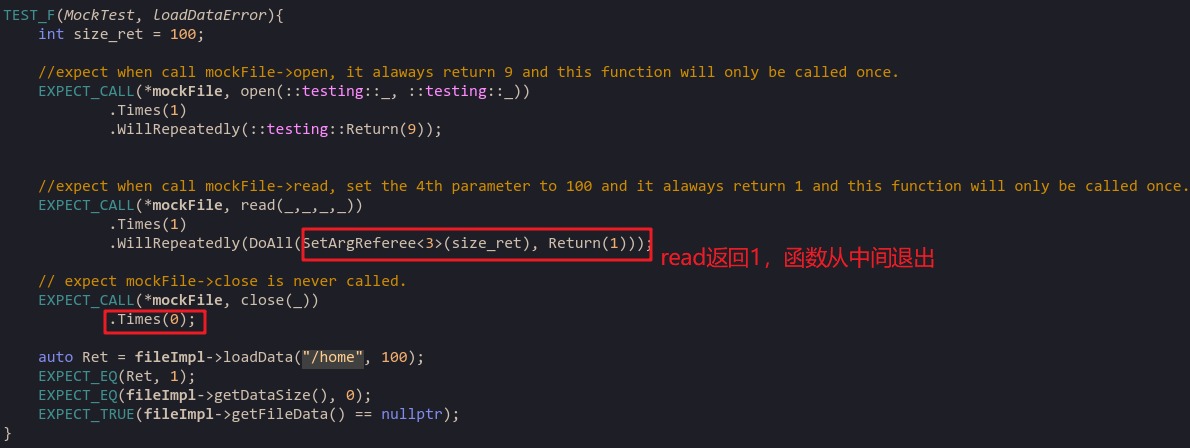
// MockTest.cpp



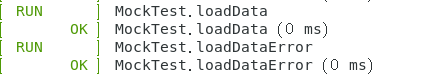
正常调用过程：



模拟异常过程：

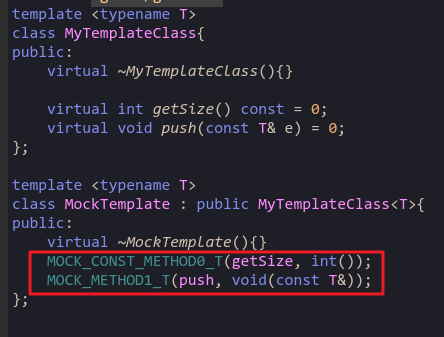


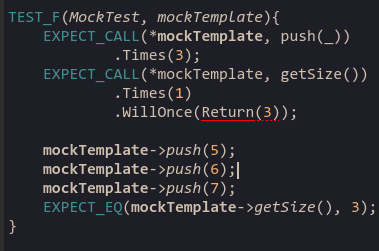
运行结果如下：



# mock模板类

// MyTemplateClass.h





执行结果如下：

