

解压后GTest的目录如上图所示，在根目录下创建文件夹build，并执行cmake：

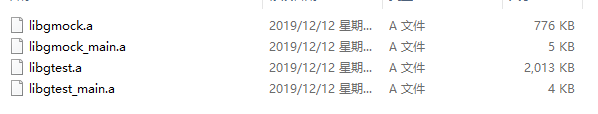
mkdir build

cd build

cmake ../

make

编译完成后会在 build/lib下生成对应于gtest与gmock相关的库文件：



可以根据需求将库加入到工程引用的库目录或者移动到系统的公共库目录（/usr/lib）。

需要引入的头文件位于：

googlemock\include

googletest\include

【gtest】

在gtest中有两个概念的定义testsuit和testcase，一个testsuit可以由若干个testcase构成。

最基本的测试单元为一个测试用例，可以通过如下方式定义：

TEST(TestSuiteName, TestName)

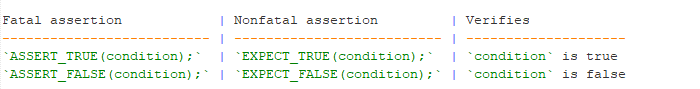
{

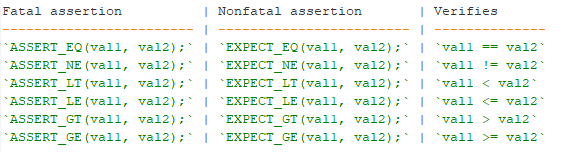
}

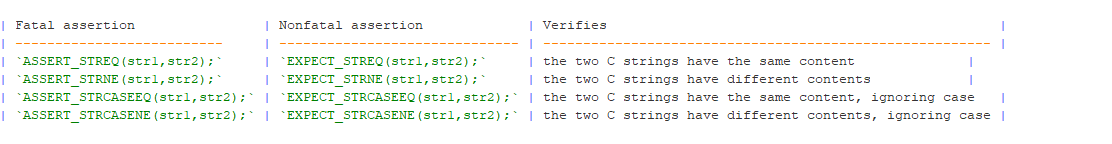
这个宏第一个参数就是所属的testsuit的名字，第二个参数代表测试用例的名称。

Gtest是通过testsuit的名称来组织测试的结果，因此如果一组测试用例在逻辑上属于一组的话，就需要在设定参数时将他们的testsuit名称设置为一致。

进行测试就需要对结果进行一些校验：相等、大于、小于等，以检查执行结果与期望是否一致，在gtest中有两类校验宏：ASSERT\_\*与EXPECT\_\*，ASSERT与EXPECT具有相同的校验能力，唯一的区别就是ASSERT\_\*失败会产生一个错误，testcase会被中断，而EXPECT\_\*失败后不会产生错误，testcase可以继续运行。在编写testcase可以根据具体的情况来选用合适的宏。EXPECT可以让你在一个testcase内检查多次输入返回，而ASSERT可以在一些无法继续执行的情况下提前结束testcase。







如果需要在多个测试用例之间共享一部分数据，可以考虑使用TESTFIXTURE（TEST\_F宏）

TEST\_F第一个参数就是创建的TestFixtrue类，第二个参数是测试用例的名称。

TestFixtrue类的创建方式：

1. 创建一个类继承与 ::testing::test，类内部以protected声明
2. 在类内部定义你需要访问的类对象，如果需要请为类实现一个构造函数或者SetUp方法，进行需要访问类对象的初始化；
3. 如果需要请实现析构函数或者TearDown来进行相关资源的释放。

4． 可以在类中实现一些方法用来支撑测试。

Gtest会为每个使用TESF\_F宏的测试用例生成一个单独的TestFixtrue类对象，gtest创建一个TestFixtrue类对象，调用SetUp完成初始化交由Test\_F使用，在使用完以后，会调用TearDown完成资源释放然后析构。

TestFixtrue类的Construct/Destruct，SetUp/TearDown伴随着改类的创建与释放，在声明周期内调用顺序为: Construct SetUp TearDown Destruct。如果没有特殊的需求，原则上建议仅使用 Construct/Destruct进行对象资源的创建与释放，具体的优点如下：

\* By initializing a member variable in the constructor, we have the option to

make it `const`, which helps prevent accidental changes to its value and

makes the tests more obviously correct.

\* In case we need to subclass the test fixture class, the subclass'

constructor is guaranteed to call the base class' constructor \*first\*, and

the subclass' destructor is guaranteed to call the base class' destructor

\*afterward\*. With `SetUp()/TearDown()`, a subclass may make the mistake of

forgetting to call the base class' `SetUp()/TearDown()` or call them at the

wrong time.

一方面是可以在构造函数对成员进行const修饰，从而避免了意外的值改变导致的错误；另一方面如果有类继承了该TextFixtrue类，在构造/析构的时候可以保证调用到父类的构造/析构方法，而不能保证及时调用SetUp与TearDown方法

也有一些不得不使用SetUp/TearDown的场景：

1. C++不允许在构造、析构方法中调用虚函数，因此只能转移到成员函数中调用
2. 不能在析构、构造方法中调用ASSERT\_\*宏，如果创建对象动作造成了错误，可以通过ASSERT\_\*阻止错误的扩散
3. 如果TearDown有可能抛出异常，就不能把功能放在析构中。

当你完成测试用例的编写以后，可以通过RUN\_ALL\_TESTS统一调用运行。

在测试工程的main方法中需要调用：testing::InitGoogleTest(&argc,argv)完成对gtest组件的初始化。

【GGtest进阶】

SUCCEED()

产生一个成功标识，但是并不会直接让整个testcase成功。可以理解为相当于某个bool类型的Assert或者expect成功，不会影响整个testcase最后的结果

FAIL()

产生一个严重错误

ADD\_FAILURE();

ADD\_FAILURE\_AT("file\_path", line\_number)

产生非严重错误。

以上这些可以运用到控制流程中，当测试结果的判断不仅仅是TRUE或者FLASE

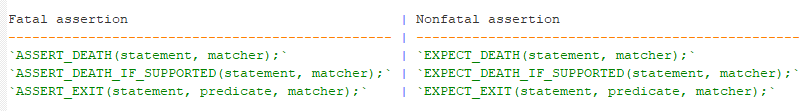
可以测试一个调用是否正确的抛出了异常

`ASSERT\_THROW(statement, exception\_type);` | `EXPECT\_THROW(statement, exception\_type);` | `statement` throws an exception of the given type

`ASSERT\_ANY\_THROW(statement);` | `EXPECT\_ANY\_THROW(statement);` | `statement` throws an exception of any type

`ASSERT\_NO\_THROW(statement);` | `EXPECT\_NO\_THROW(statement);` | `statement` doesn't throw any exception

DEATH TEST



在一些情况下，某些测试输入会造成程序出错甚至崩溃，可以通过\*\_DEATH来进行监听。

if a piece of code throws an exception, we don't consider it "death"

for the purpose of death tests, as the caller of the code could catch the

exception and avoid the crash. If you want to verify exceptions thrown by your

code

如果程序只是抛出了异常，我们不认为这是一个DEATH\_TEST，对应的，需要使用异常监听的相关判断去监听

DEATH\_TEST会创建一个新的进程去执行DEATH Expression 原来的进程并不会死亡，在创建的子进程死亡以后，会返回一些stderr信息，如果需要进行匹配，则在调用

\*\_DEATH或者\*\_EXIT的时候需要提供一个合适的正则表达式matcher进行匹配，如果接受任意的则直接传递一个空的字符串“”

使断言的错误信息更加丰富：

Gtest提供的断言系列很丰富但是不能涵盖用户所有的使用场景，在一些场景，由于没有合适的宏（所有的断言都是以宏的形式提供），我们会使用EXPECT\_TRUE去检查一个复杂的表达式，但是没有办法在出错时获取到是哪个部分产生了错误，所以用户需要自己去构建错误信息，有时候这种构建的代价非常大。因此gtest提供了以下的方案来解决这个问题：

1. 有一个返回类型为bool（或者可以转换为bool）的函数，可以使用：

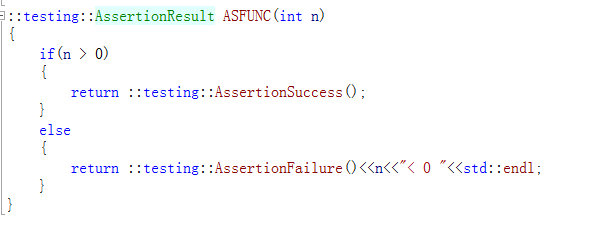
EXPECT\_PRED2(func, a, b);

ASSERT\_PRED2(func, a, b);

EXPECT/ASSERT\_PRED\*中\*表示函数接收的参数，第一个参数为函数名称，剩余的参数为函数接收的参数，数量由上述的\*定义。如果产生失败会有一些默认的错误信息。

1. 使用一个返回AssertionResult的函数

在测试代码中可以创建一个方法，返回AssertionResult：



AssertionResult一般有两种返回情况 AssertionSuccess和AssertionFailure，如上述代码的else分支，可以根据具体的错误信息已流的形式输入到AssertionResult返回对象中。

然后可以使用

EXPECT\_FALSE（ASFUNC（-1））来进行断言测试。

使用这样测试，不再对实际待测试代码的返回值有要求，可以返回任意的值，只需要在新定义的函数中根据不同的值选择返回对应的分支并记录对应的信息。但是这种写法，只是自动生成了错误信息，但是信息表达方式并不受控制。

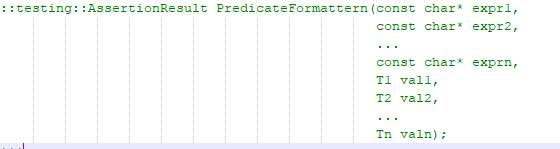
1. Predicate-Formatter就是用来解决2中的问题，并给出了一个统一的解决方案：

ASSERT\_PRED\_FORMAT2(pred\_format2, val1, val2)

EXPECT\_PRED\_FORMAT2(pred\_format2, val1, val2)

和之前的EXPECT\_PRED\*类似，ASSERT/EXPECT\_PRED\_FORMAT\*中\*代表了参数数量

pred\_format2代表了gtest提供的一个函数范式，需要使用该范式将需要测试的函数进行封装，其标准范式说明为：



其中exprn代表了方法参数的表达式，valn代表了实际的参数， Tn代表了类型

如果定一个函数：::testing::AssertionResult Asserttestzero(const char\* expr,int a)

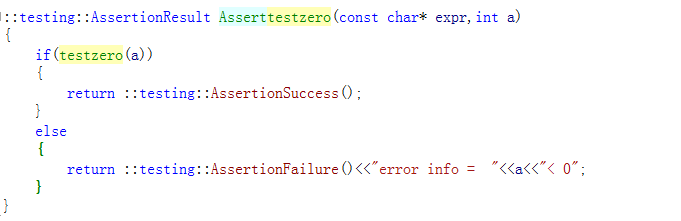
调用测试时：

int a = -1;

EXPECT\_PRED\_FORMAT1(Asserttestzero, a);

此时 expr1 = “a” val1 = 1 T1 = int

举个例子：



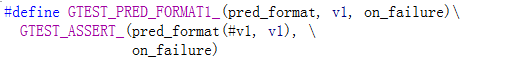
实际在函数内部实现与2中类似，也是根据函数执行结果进行分支返回，但是此处直接调用了用户的函数，避免了对用户的代码的修改，由于之前介绍函数范式中Tn是任意类型，因此Tn可能是个引用，所以这样处理，我们不仅仅可以检查函数的返回值，还可以检查它带出的引用，诸如此类。

整体的看，

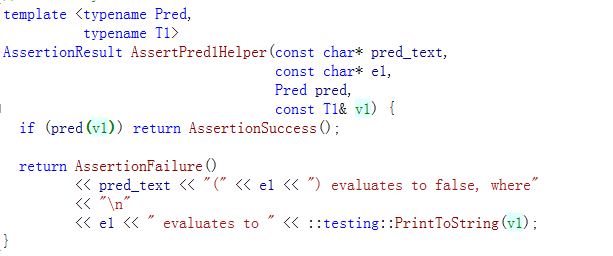
方法1属于特殊情况，函数直接返回了bool；

方法2属于3的简化情况，参考具体实现代码：

EXPECT\_PRED\_FORMAT1的实现如下：



EXPECT\_PRED1的实现



EXPECT\_PRED1中的函数 AssertPred1Helper就相当于在gtest内部实现了一个符合方法3中描述的返回AssertionResult的函数范式的函数。

浮点数比较：

对于浮点数的相等，之前的ASSERT\_EQ可能无法满足需求（主要是因为计算机存储浮点数的方式加上类型的精度控制）

可以使用 \*\_FLOAT\_EQ 或者 \*\_DOUBLE\_EQ

\*\_NEAR可以判断两个浮点数是不是近似相等

由于浮点数有丰富的操作，为了避免因为涉及浮点数就需要额外定义大量的宏，因此提供了

EXPECT\_PRED\_FORMAT2(::testing::FloatLE, val1, val2);

EXPECT\_PRED\_FORMAT2(::testing::DoubleLE, val1, val2);

用于进行浮点数比较

\*\_THAT（value,matcher）用于检查传递给一个mock方法的参数是否符合要求

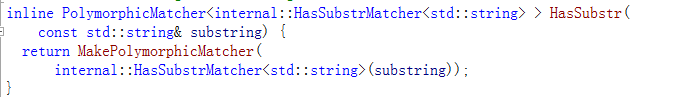
//matcher将在gmock部分进行介绍

可以通过不同的MATHCER来对value进行判断，实现很多字符串断言

在内部 \*\_THAT宏最后调用\*\_PRED\_FORMAT1（::testing::internal::MakePredicateFormatterFromMatcher(matcher), value)，其中MakePredicateFormatterFromMatcher调用PredicateFormatterFromMatcher的移动构造方法创建了对象，PredicateFormatterFromMatcher内部重载了运算符（），最后执行了PredicateFormatterFromMatcher（value）

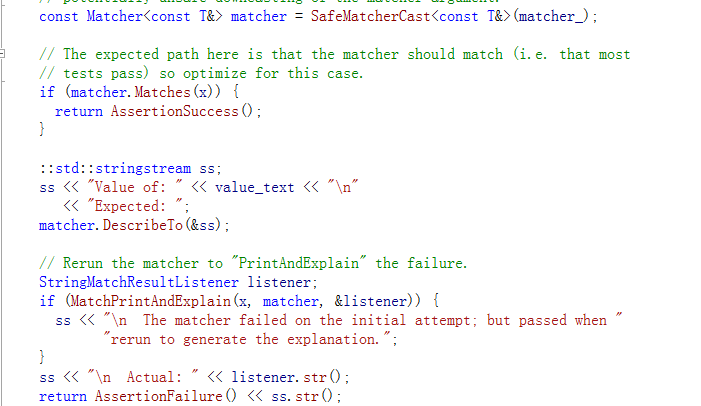
PredicateFormatterFromMatcher是一个模板类，可以传入不同的MATCHER类型，如::HasSubstr;

::testing::HasSubstr的定义



其中inyernal::HasSubStrMatcher是这个matcher实际实现的部分，其内部主要实现了方法MatchAndExplain和DescribeTo。

在之前描述的PredicateFormatterFromMatcher的重载运算符（）中实现如下：



类型检查：

::testing::StaticAssertTypeEq<T1, T2>(); 如果T1,T2不相等会返回编译错误

Gtest 访问测试私有成员变量

FRIEND\_TEST 可以根据情况声明为TEST或者TEST\_F

FRIEND\_TEST（suitename/fixturename,testcase）

动态啊生成测试用例：

It provides the following signature:

```c++

template <typename Factory>

TestInfo\* RegisterTest(const char\* test\_suite\_name, const char\* test\_name,

const char\* type\_param, const char\* value\_param,

const char\* file, int line, Factory factory);