编写优美的 GTest 测试案例

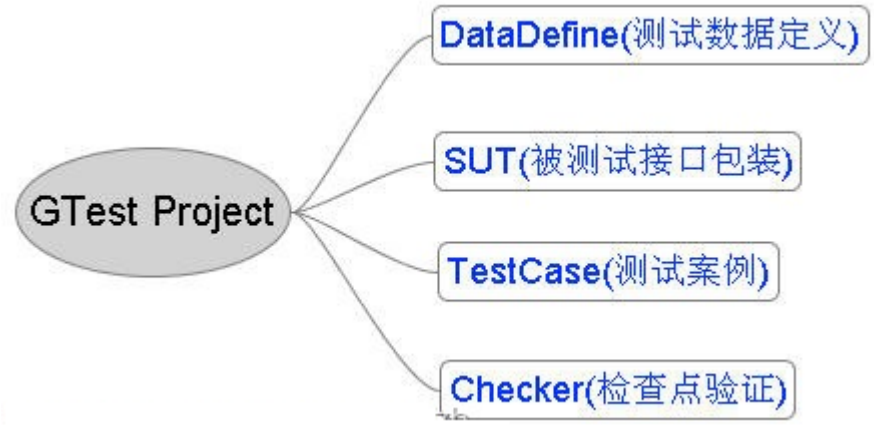
使用 gtest 也有很长一段时间了，这期间也积累了一些经验，所以分享一下。GTest 为我们提供了便捷的测试框架，让我们只需要关注案例本身。如何在 GTest 框架下写出优美的测试案例，我觉得必须要做到：

1. 案例的层次结构一定要清晰
2. 案例的检查点一定要明确
3. 案例失败时一定要能精确的定位问题
4. 案例执行结果一定要稳定
5. 案例执行的时间一定不能太长
6. 案例一定不能对测试环境造成破坏
7. 案例一定独立，不能与其他案例有先后关系的依赖
8. 案例的命名一定清晰，容易理解

案例的可维护性也是非常重要，如果做到上面的 8 点，自然也就做到了可维护性。下面来分享一下我对于上面 8 点的经验：

1．案例的层次结构一定要清晰

所谓层次结构，至少要让人一眼就能分辨出被测代码和测试代码。简单的说，就是知道你在测什么。由于是进行接口测试，我已经习惯了如下的案例层次：



DataDefine

我会将测试案例所需要的数据，以及数据之间的联系全部在预先定义好。测试数据与案例逻辑的分离，有利于维护和扩展测试案例。同时，GTest 先天就支持测试数据参数化，为测试数据的分离提供了进一步的便捷。什么是测试数据参数化？就是你可以预先定义好一批各种各样的数据，而你只需要编写一个测试案例的逻辑代码，gtest 会将定义好的数据逐个套入测试案例中进行执行。具体的做法请见：玩转 Google 开源 C++单元测试框架 Google Test 系列(gtest)之四 - 参数化

SUT

SUT，即 system under test，表明你的测试对象是什么，它可以是一个类(CUT)，对象(OUT)，函数(MUT)，甚至可以是整个应用程序(AUT)。我单独将这个层次划分出来，主要有两个目的：

- 明确的表示出你的测试对象是什么
- 为复杂调用对象包装简单调用接口

明确表示测试对象是什么，便于之后对测试案例的维护和对测试案例的理解。同时，对于一些被测对象，你想要调用它需要经过一系列烦琐的过程，这时，就需要将这一烦琐的调用过程隐藏起来，而只关注被测对象的输入和输出。

TestCase

测试工程中，必须非常明确的表示出哪些是测试案例，哪些是其他的辅助文件。通常，我们会在测试案例的文件名加上 Test 前缀(或者后缀)。我建议，将所有的测试案例文件或代码放在最显眼的地方，让所有看到你的测试工程的人，第一眼看到的就是测试案例，这很重要。

Checker

对于一个复杂系统的接口测试，仅仅坚持输入和输出是远远不够的。比如测试一个写数据库的函数，函数的返回值告诉你数据已经成功写入是远远不够的，你必须亲身去数据库中查个究竟才行。因此，对于某一类的测试案例，我们可以抽象出一些通用的检查点代码。

如果做到上面的分层，那么一个测试案例写出来的结构应该会是这个样子：

```
TEST(TestFoo, JustDemo)
{
    GetTestData(); // 获取测试数据

    CallSUT(); // 调用被测方法

    CheckSomething(); // 检查点验证
}
```

这样的测试案例，一目了然。

2. 案例的检查点一定要明确

一定要明确案例的检查点是什么，并且让检查点尽量集中。有一个不好的习惯就是核心的检查点在分布在多个函数中，需要不断的跳转才能了解到这个案例检查了些什么。好的做法应该是尽量让检查点集中，能够非常清晰的分辨出案例对被测代码做了哪些检查。所以，尽量让 Gtest 的 ASSERT_和 EXPECT_系列的宏放在明显和正确的地方。

3. 案例失败时一定要能精确的定位问题

测试案例失败时，我们通常手忙脚乱。如果一个测试案例 Failed，却不能立即推断是被测代码的 Bug 的话，这个测试案例也有待改进。我们可以在一些复杂的检查点断言中加入一些辅助信息，方便我们定位问题。比如下面这个测试案例：

```
int n = -1;
bool actualResult = Foo::Dosometing(n);
ASSERT_TRUE(actualResult)
```

如果测试案例失败了，会得到下面的信息：

```
Value of: actualResult
Actual: false
Expected:true
```

这样的结果对于我们来说，几乎没有什么用。因为我们根本不知道 actualResult 是什么，以及在什么情况下才会出现非预期值。因此，在断言处多加入一些信息，将有助于定位问题：

```
int n = -1;
bool actualResult = Foo::Dosometing(n);
ASSERT_TRUE(actualResult) << L"Call Foo::Dosometing(n) when n = " <
< n;
```

4. 案例执行结果一定要稳定

要保证测试案例在什么时候、什么情况下执行的结果都是一样的。一个一会成功一会失败的案例是没有意义的。要保证案例稳定性的方法有很多，比如杜绝案例之间的影响，有时候，由于前一个案例执行完后，将一些系统的环境破坏了，导致后面的案例执行失败。在测试某些本身就存在一定几率或延时的系统时，使用超时机制是比较简单的办法。比如，你需要测试一个启动 Windows 服务的方法，如果我们在调用了该方法后立即进行检查，很可能检查点会失败，有时候也许又是通过的。这是因为 Windows 服务由 Stop 状态到 Running 状态，中间还要经过一个 Padding 状态。所以，简单的做法是使用超时机制，隔断时间检查一次，直到超过某个最大忍受时间。

```
ASSERT_TRUE(StartService('xxx'));
int tryTimes = 0;
int status = GetServiceStatus('xxx');
while (status != Running)
{
    if (tryTimes >= 10)
        break;
    ::Sleep(200);
    tryTimes++;
    status = GetServiceStatus('xxx');
}
ASSERT_EQ(Running, status) << "Check the status after StartService('x
xx')";
```

5．案例执行的时间一定不能太长

我们应该尽量让案例能够快速的执行，一方面，我们可以通过优化我们的代码来减少运行时间，比如，减少对重复内容的读取。一方面，对于一些比较耗时的操作，比如文件系统，网络操作，我们可以使用 Mock 对象来替代真实的对象。使用 GMock 是一个不错的选择。

6．案例一定不能对测试环境造成破坏

有的案例需要在特定的环境下来能执行，因此会在案例的初始化时对环境进行一些修改。注意，不管对什么东西进行了修改，一定要保证在案例执行完成的 TearDown 中将这些环境都还原回来。否则有可能对后面的案例造成影响，或者出现一些莫名其妙的错误。

7．案例一定独立，不能与其他案例有先后关系的依赖

任何一个案例都不依赖于其他测试案例，任何一个案例的执行结果都不应该影响到别的案例。任何一个案例都可以单独拿出去正确的执行。所以，不能寄希望于前一个案例所做的环境准备，因为这是不对的。

8．案例的命名一定清晰，容易理解

案例的名字要规范，长不要紧，一定要清晰的表达测试案例的用途。比如，下面的测试案例名称都是不好的：

```
TEST(TestFoo, Test)
TEST(TestFoo, Normal)
TEST(TestFoo, Alright)
```

比如像下面的案例名称就会好一点：

```
TEST(TestFoo, Return_True_When_ParameterN_Larger_Then_Zero)
TEST(TestFoo, Return_False_When_ParameterN_Is_Zero)
```