Google Test 简介与基本用法

写在最前：

首先谢谢大家支持MATRIX，欢迎大家来看我们的博客

这篇博客介绍了Google Test的一些特性和基本用法，也是我的学习笔记。这篇博客从单元测试和xUnit开始讲起，然后再过渡到Google Test

---

单元测试

> 针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作

程序模块可以是函数式编程中的函数和过程，也可以是面向对象编程中类的方法。通常每当我们完成了一个模块，我们都会对它进行测试以确保模块功能的完好。比如，后台完成了一个插入数据库的api，那么当我们使用这个api进行数据插入时，我们希望数据能够被正确的插入进数据库，于是我们使用这个api先尝试着插入几条数据，然后再在数据库中查看是否正确插入，这便是单元测试的工作原理。

单元测试一个非常重要的特点是，各个模块是独立的，A模块功能的正确不应该依赖于B模块（因为我们只对A模块进行单元测试）。比如插入数据库的api模块不应该与制造数据的模块同时在同一单元测试中测试，出现依赖关系的时候应该考虑是否应该重新划分模块，将A，B模块集成为一个模块，如将产生数据和插入数据库集成为一个模块（对于这种情况不建议合并），或者通过模拟(mock)来模拟B的功能（在数据库插入的api测试中mock接口，并产生假数据）。

---

xUnit

> xUnit是一系列单元测试框架的集合(x代表语言，如JUnit中J代表Java)，xUnit测试框架是面向对象的，并且他们有一套共同的架构

xUnit的架构[1]

- Test runner
  执行所有测试的可执行程序

- Test case
  基本类，所有的测试都继承于test case，对于同一个单元可以有多个test case来进行测试，每个测试可以覆盖不同的方面。比如在测试插入数据库api时，一个case对应插入正确数据，另一个case对应插入错误数据等
- Test fixtures
  可以理解为测试所需环境。在每个测试前为测试而搭建的环境（preconditions）。例如在在测试插入数据库api时，fixture可以为连接数据库
- Test suites
  共享同一fixture（precondition）的一系列测试，test suit里面的测试顺序不重要（每个测试都是独立的，互不依赖），例如在上述测试中的两个test case就应该放入同一test suit下
- Test execution
  对于每一个test case，都以以下步骤执行
  - `setup() // 搭建测试环境`
  - `test() // 执行测试`
  - `teardown() // 恢复测试环境，以免影响到下一个测试`
- Test result formatter
  输出测试结果，可以是plain text也可以是xml或其他格式
- Assertions
  Assertion（断言）是用来测试函数返回结果是否为期望值的函数或宏，遇到非期望值时，一般会抛出异常，终止当前测试

Google Test

> Google Test，全称为Google's C++ test framework，是一个基于**xUnit**的C++测试框架

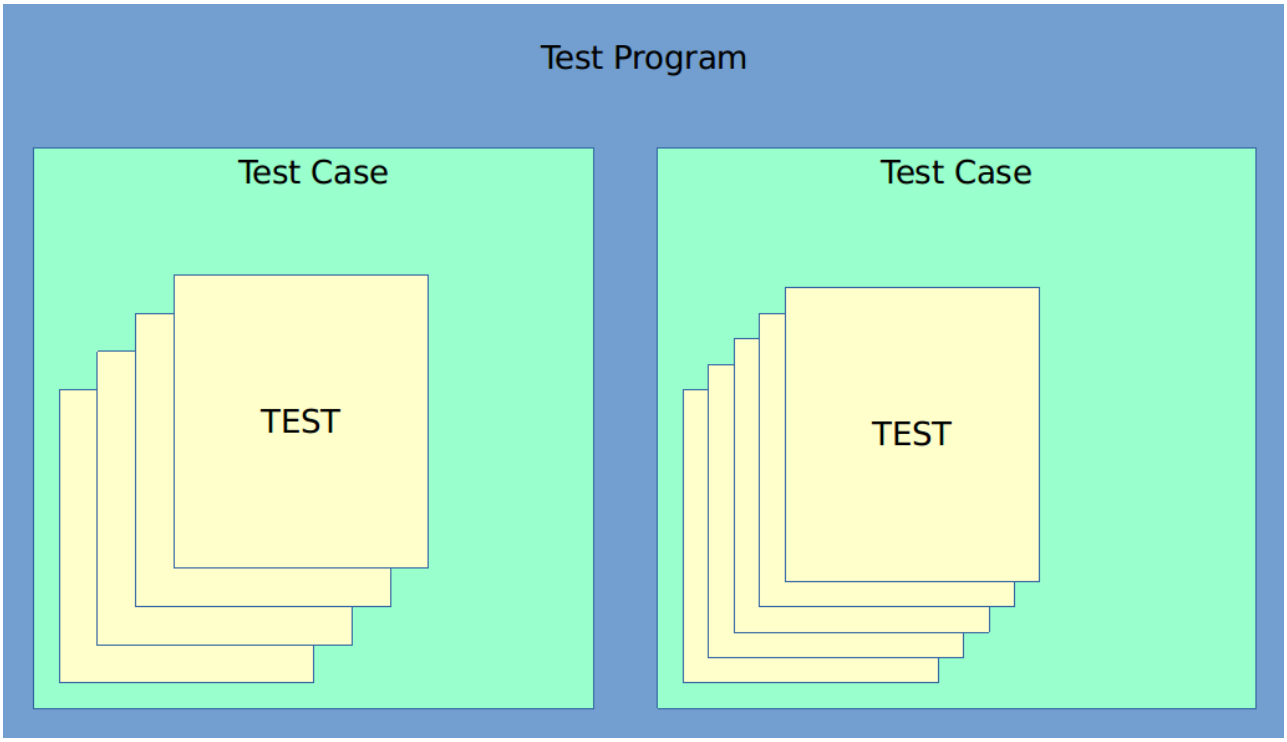
如果你熟悉xUnit，你会很快上手，如果不熟悉，请参见上面xUnit的介绍。

对于Google Test，以一篇博客实在难以详尽的说明其特性和使用方法。这里对Google Test做一个简短的说明和介绍。

首先附上Google Test的网站（github）：https://github.com/google/googletest
在这里面可以找到Google Test的所有内容，推荐里面的常用内容：

- Google Test Primer
- AdvancedGuide
- Samples

Google Test程序框架[2]



Quick Look

假设我们实现了一个除法函数，其定义和实现如下：

```
//  division.h

#ifndef DIVISION_H
#define DIVISION_H

#include <stdexcept>

namespace myns {
    float divide(float dividend, float divisor) throw(std::domain_error);
}

#endif /* DIVISION_H */
```

```cpp
//  division.cpp

#include <stdexcept>
#include <iostream>
#include "division.h"

using namespace std;

float myns::divide(float dividend, float divisor) throw(domain_error) {
    if (divisor == 0) {
        throw domain_error("Divisor cannot be 0");
    }
    return dividend / divisor;
}
```

那么测试文件可以这么写：

```cpp
//  test.cpp

#include <gtest/gtest.h>
#include <stdexcept>
#include "division.h"

TEST(divisionTest, divisorNotZero) {
    EXPECT_EQ(2, myns::divide(8, 4));
}

TEST(divisionTest, divisorZero) {
    EXPECT_THROW(myns::divide(1, 0), std::domain_error);
}
```

编译命令：

```
g++ division.cpp test.cpp -o test -std=c++11 -lgtest -lgtest_main -lpthread
```

执行结果：

```
Running main() from gtest_main.cc
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from divisionTest
[ RUN      ] divisionTest.divisorNotZero
[       OK ] divisionTest.divisorNotZero (0 ms)
[ RUN      ] divisionTest.divisorZero
[       OK ] divisionTest.divisorZero (0 ms)
[----------] 2 tests from divisionTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 2 tests.
```

假设我们没有实现抛出异常的部分，即实现代码为：

```
float myns::divide(float dividend, float divisor) throw(std::domain_error)
 {
    // if (divisor == 0) {
    //     throw domain_error("Divisor cannot be 0");
    // }
    return dividend / divisor;
}
```

那么测试结果为：

```
Running main() from gtest_main.cc
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from divisionTest
[ RUN      ] divisionTest.divisorNotZero
[       OK ] divisionTest.divisorNotZero (0 ms)
[ RUN      ] divisionTest.divisorZero
test.cpp:10: Failure
Expected: myns::divide(1, 0) throws an exception of type std::domain_error.
  Actual: it throws nothing.
[  FAILED  ] divisionTest.divisorZero (0 ms)
[----------] 2 tests from divisionTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] divisionTest.divisorZero

 1 FAILED TEST
```

对Quick Look的一些说明

- `TEST()`

  在Google Test中，每一个Test case都由`TEST()`宏来实现，其接收两个参数，第一个为test case name，第二个为test name（测试名字都是自己定义的，但是要按照要求定义）。正如Quick Look中的两个测试，`TEST(divisionTest, divisorNotZero)`，`TEST(divisionTest, divisorZero)`，第一个参数表明是对divide这个函数进行测试，第二个表示测试的测试的方面。这些名字会反应在生成的报告上。

  有一点要注意的是，传入的名字不能带`_`字符，因为这是一个宏定义，用户传入的名字会插入在Google Test内部定义的变量中，而这个变量是用`_`连接的。在运行test时用gdb查看调用栈可以看到：

```
#1  0x00000000004050d5 in divisionTest_divisorNotZero_Test::TestBody
(this=0x6661c0) at test.cpp:6
```

- Assertions

  Google Test的断言非常丰富，在Quick Look中使用了两个断言：`EXPECT_EQ()`和`EXPECT_THROW()`。其中`EXPECT_EQ()`会判断传入的两个值是否相等，若相等，则这次断言通过，不等，则失败。通常，第一个参数为期望值，第二个为实际测试的值，这个结果会最终反映在report中，用Expected和Actual表示。`EXPECT_THROW()`同理，第一个参数为一个表达式，第二个为一个异常，并期望这个表达式会抛出这个异常。

Google Test中断言分为两类，一类是`EXPECT_*`，一类是`ASSERT_*`。`EXPECT_*`在失败时会继续这个`TEST()`，而`ASSERT_*`在失败时会终止该`TEST()`。一般情况下，`ASSERT_*`用于致命错误，或者是这次断言失败`TEST()`中剩下的测试都没有意义的情况。

Google Test完整的Assertion表可以参看开头给出的Primer和AdvancedGuide的链接（有机会再写吧XD）

- 编译命令

写好的测试代码在编译时需要加上`-std=c++11 -lpthread -lgtest -lgtest_main`参数，Google Test安装时会安装`libgtest.a`和`libgtest_main.a`（默认的main函数，这样就不用写main函数了）静态库。同时，Google Test是一个多线程程序，需要使用线程库。（实际测试的时候Google Test并不会开多线程让测试同时进行，因为这涉及到线程安全和死亡测试等问题，同时，也不要在非pthread库支持的平台下多线程测试[3]）

一些重要的特性

`main()`

上面的例子使用了Goolge Test自带的main静态库，当然我们也可以自己写main函数，下面是Google Test自带的main函数：

```
int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Google Test首先接收外部传参进行初始化（具体参数可以在运行测试程序时传入`--help`参数查看），`RUN_ALL_TESTS()`会收集所有的`TEST`并执行，因此实现的`TEST`不用手动注册了（后面讲到全局`SetUp`和`TearDown`时需要手动注册）

Fixture

Fixture的目的是减少代码冗余，如果你的一系列测试中都需要在相同的环境运行，或者需要相同的测试数据，那么你不用在每次`TEST`的开头手动搭建环境和创建数据，而可以将这些配置都写在fixture里面。

在Google Test中，创建一个fixture首先需要定义一个类，类名即为test case的名字，并从`::testing::Test`继承，其中的所有成员都需要声明为`public`或`protected`，其中你可以定义以下内容：

- `virtual void SetUp()`（注意是大写的U）
- `virtual void TearDown()`
- `static void SetUpTestCase()`
- `static void TearDownTestCase()`
- 以及在测试中需要用到的变量

回顾xUnit中Test execution的执行顺序，每次执行`TEST()`的都会先执行`SetUp()`，每次`TEST()`执行完后都会执行`TearDown()`,继承`::testing::Test`后重写这两个函数可以自定义其功能，对于开头数据库的例子，连接数据库的步骤就可以写在`SetUp()`中。

`SetUpTestCase()`和`TearDownTestCase()`同理，不过这个函数是作用在Test Case上的。它们分别会在Test Case中第一个`TEST()`的`SetUp()`之前和最后一个`TEST()`的`TearDown()`之后执行，作为整个Test Case的环境维护。这两个函数可以用于构造开销较大，在每个`TEST()`中都要用到的数据，以达到数据在`TEST()`之间共享的目的。要注意的是，数据共享并不意味着两个**`TEST()`**之间能够相互影响，原则上**`TEST()`**之间是应该互相独立的，其执行顺序不同也不应该影响测试结果，这意味着，当你在一个**`TEST()`**中修改了数据，你应该在测试结束时将其恢复。

在每一个`TEST()`中都需要用到的变量可以直接声明在fixture中，在测试中可以直接使用`SetUpTestCase()`和`TearDownTestCase()`中操作的变量需要声明为`static`。

所以，执行顺序为：`SetUpTestCase()->SetUp()->TEST()->TearDown()->...->SetUp()->TEST()->TearDown()->TearDownTestCase()`，若有多个Test Case，则继续下一个Case。

最后，要使用这个**fixture**，你需要使用**`TEST_F()`**宏，并将**fixture**类的名字传入第一个参数。

举个例子，现在修改上面`test.cpp`的代码，添加了`setup`和`teardown`，并添加了一些数据

```cpp
//  test.cpp

#include <gtest/gtest.h>
#include <stdexcept>
#include <iostream>
#include "division.h"

class divisionTest: public ::testing::Test {
protected:
    static void SetUpTestCase() {
        std::cout << "SetUpTestCase\n";
    }
    static void TearDownTestCase() {
        std::cout << "TearDownTestCase\n";
    }
    virtual void SetUp() {
        data1 = 8;
        data2 = 4;
        data3 = 0;
        std::cout << "SetUp\n";
    }
    virtual void TearDown() {
        std::cout << "TearDown\n";
    }
    int data1, data2, data3;
};


TEST_F(divisionTest, divisorNotZero) {
    std::cout << "Test: divisorNotZero\n";
    EXPECT_EQ(2, myns::divide(data1, data2));
}

TEST_F(divisionTest, divisorZero) {
    std::cout << "Test: divisorZero\n";
    EXPECT_THROW(myns::divide(1, data3), std::domain_error);
}
```

运行结果:

```
Running main() from gtest_main.cc
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from divisionTest
SetUpTestCase
[ RUN      ] divisionTest.divisorNotZero
SetUp
Test: divisorNotZero
TearDown
[       OK ] divisionTest.divisorNotZero (0 ms)
[ RUN      ] divisionTest.divisorZero
SetUp
Test: divisorZero
TearDown
[       OK ] divisionTest.divisorZero (0 ms)
TearDownTestCase
[----------] 2 tests from divisionTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 2 tests.
```

从上面的运行结果可以很直观的看出执行顺序。

既然Test Case也有`setup`和`teardown`，那么对于整个程序是否也有类似的方法呢？有的，在AdvancedGuide中有介绍。和创建fixture类似，首先你需要继承`::testing::Environment`这个类，并重写`virtual void SetUp()`和`virtual void TearDown()`这两个方法。但是与fixture不同的是，你需要注册这个environment，在`RUN_ALL_TESTS()`之前调用`Environment *AddGlobalTestEnvironment(Environment* env)`来注册。

当然，你可以注册多个Environment，他们会按照注册顺序依次执行。若你的设置了重复次数，那么每次重复时都会执行。

举个例子，重写main函数：

```cpp
//  main.cpp

#include <gtest/gtest.h>
#include <iostream>
#include "division.h"

using namespace std;

class myEnv_1: public ::testing::Environment {
public:
    virtual void SetUp() {
        cout << "Global setup 1\n";
    }
    virtual void TearDown() {
        cout << "Global teardown 1\n";
    }
};


class myEnv_2: public ::testing::Environment {
public:
    virtual void SetUp() {
        cout << "Global setup 2\n";
    }
    virtual void TearDown() {
        cout << "Global teardown 2\n";
    }
};

int main(int argc, char **argv) {
    AddGlobalTestEnvironment(new myEnv_1);
    AddGlobalTestEnvironment(new myEnv_2);
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

运行结果：

```
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
Global setup 1
Global setup 2
[----------] 2 tests from divisionTest
SetUpTestCase
[ RUN      ] divisionTest.divisorNotZero
SetUp
Test: divisorNotZero
TearDown
[       OK ] divisionTest.divisorNotZero (0 ms)
[ RUN      ] divisionTest.divisorZero
SetUp
Test: divisorZero
TearDown
[       OK ] divisionTest.divisorZero (0 ms)
TearDownTestCase
[----------] 2 tests from divisionTest (0 ms total)

[----------] Global test environment tear-down
Global teardown 2
Global teardown 1
[==========] 2 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 2 tests.
```

自定义报错信息

如果在上面fixture例子中没有实现抛出异常的代码，那么错误报告会是这样：

```
test.cpp:37: Failure
Expected: myns::divide(1, data3) throws an exception of type std::domain_er
ror.
  Actual: it throws nothing.
```

你会发现报错信息中出现了`data3`这个变量但是并没有显示出它的值，这非常不友好，也不容易让我们定位到错误。Google Test的所有断言都可以接收字符串流并显示在报告中。对此，你可以自定义报错信息来查看`data3`变量。下面是一个例子：

```
TEST_F(divisionTest, divisorZero) {
    std::cout << "Test: divisorZero\n";
    ASSERT_THROW(myns::divide(1, data3), std::domain_error)
        << "With data3 = " << data3;
}
```

报错信息：

```
Expected: myns::divide(1, data3) throws an exception of type std::domain_er
ror.
  Actual: it throws nothing.
With data3 = 0
```

写在最后：

了解了上面涉及到的特性就已经可以对大多数程序写测试了。当然如果想要写出高质量的单元测试，一些高级特性也是必须要了解的，建议在写之前先浏览一遍Assertion表。

最后还有些内容本文没有覆盖到，但是依旧也很重要的有：Assertion的详细介绍（包括自定义断言等），测试程序的传参（可以设置测试行为，如：生成xml报告，设置测试次数，随机测试顺序等）

---

1. 参见wikipedia里xUnit架构介绍 ↩
2. 在Google Test中的test case对应xUnit中的test suit，命名规则不同吧 ↩
3. 参见Google Test Primer中最后一段 ↩