gtest快速上手 (/a/1190000002454946)

spacewander (/u/spacewander) **3.2k** 2014年12月31日 发布

因为最近在写的一些C++代码，需要给它写单元测试，所以就得去找一个C++的测试框架。正好之前实验室的同学有推荐过gtest，所以就不纠结了，直接去gtest的项目主页看。

gtest好处都有啥？这个可真说不出来，毕竟C++方面的测试框架，我就只用过gtest。对比于其他语言的测试框架，比如javascript的mocha、jasmine，ruby的minitest，gtest差强人意。不过前面举的例子，都是动态语言的测试框架（我没用过JUnit）。在测试的方面，动态语言对于静态语言具有巨大的优势，因为动态语言可以轻易地hack到待测试方法的内部。何况C++连反射都没有……

不过，总体来说，gtest已经能够满足我对测试的要求了。好了，外话就此打住，开始进入正文。

安装

安装可以见项目的README。这里摘抄兼意译一下：

1. 在下下来的gtest源码文件夹外创建一个构建用的文件夹，这里就用 `lib` 好了。

举个例子， 如果你把 `gtest` 下载到 `~/code` 文件下了，那么就 `mkdir ../lib` 。

1. 切到该文件夹下，对gtest源码所在的目录使用cmake。接着会产生一份Makefile/VC项目文件/Xcode项目文件（会出现哪一个取决于你用的系统）。剩下来的，就只是编译而已。

依上面的例子为例，就是

```
cd ../lib
cmake ../gtest-xxxx
make # 或者build一个VC/Xcode项目
```

最后会生成一些东西，其中有一个 `libgtest.a` 的静态库（具体内容取决于你的系统，这是Linux上的情况），这就是我们想要的。

1. gtest在使用时，依赖这个静态库和gtest的头文件。在编译测试代码时，使用

```
$(CC) -isystem $(GTESTHEADERS) -pthread $(CCFLAGS) $(SRC) $(GTESTLIB)
```

其中 `GTESTHEADERS` 是gtest头文件所在的文件夹，位于下载下来的代码的 `include` 路径； `GTESTLIB` 是编译出的静态库。如果不需要跨平台，可以考虑把gtest头文件和静态库一起随项目发布，这样别人就无需再安装gtest了。

是不是挺麻烦的？习惯了就好……C++需要的，不是那么多奇技淫巧，而是一个好用的包管理器。

快速上手

下面就对比着其他测试框架，来一次快速上手：

首先，一个测试框架至少由两个部分组成，**测试结构**和**断言方法**（对不起这两个词都是我在刚刚半分钟内创造的）。

测试结构：测试用例的组织方式（测试用例、测试函数）、运行方式、钩子函数。

断言方法：可用的断言，比如 `expect().toBe()` 和 `assert_respond_to` 等等。

那我们就从这两方面介绍。

测试结构

如果你用过动态语言的测试框架，你会注意到测试函数是这样组织的：

```
class XXXTest ...
  def test_xxx
  def test_yyy
```

或者

```
describe ...
  it xxx
  it yyy
```

gtest的组织方式也不例外。不过它并不是将测试函数嵌套到测验用例之下。看一下示例就明白：

```
// test.cpp
#include "../lib/gtest.h" // 注意要把gtest.h给include进来
TEST(Gtest, testItWorks)
{
    EXPECT_EQ(2, 1 + 1);
}
```

这里，Gtest相当于XXXTest类，而testItWorks则是一个要运行的测试函数。

那么怎么让它运行呢？我们还需要另外一个文件，实现一个运行测试的main函数。姑且叫它 `main.cpp` 好了：

```
 #include "../lib/gtest.h" // 注意要把gtest.h给include进来

int main(int argc, char** argv)
{
    testing::InitGoogleTest(&argc, argv);

    // Runs all tests using Google Test.
    return RUN_ALL_TESTS();
 }
```

接下来就是编译了（路径什么的得按照你的实际情况来）：

```
g++ -isystem ../lib -pthread test.cpp main.cpp ../lib/libgtest.a
```

运行生成的 a.out 文件，你看，测试结果出来了：

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from Gtest
[ RUN      ] Gtest.testItWorks
[       OK ] Gtest.testItWorks (0 ms)
[----------] 1 test from Gtest (0 ms total)

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

如果你只是想运行部分测试，运行时添加 --gtest_filter 选项。如 '--gtest_filter=Gtest.*' 就是只运行Gtest下的测试函数。

接下来我们讲讲钩子函数。一般来说，在每个测试函数之前，往往会做一些通用的操作，比如准备fixture等等。有时在测试函数结束后，还需要做扫尾的任务。这时候就需要使用钩子函数了。

先讲一下如何设置fixture。Fixture就是在每个测试函数运行之前，事先准备好的被测试对象。
在gtest中，需要定义一个类来添加fixture。

```
#include <vector>
#include "../lib/gtest.h"

class Gtest : public ::testing::Test
{
protected:
    std::vector<int> v;
};

TEST_F(Gtest, testItWorks)
{
    EXPECT_EQ(2, 1 + 1);
}

TEST_F(Gtest, testVector)
{
    EXPECT_NE(1, v.size());
}
```

请注意有两点变化，一个是我们定义了Gtest类，另一个是我们把 TEST 改成 TEST_F 。通过定义与测试用例同名的类，我们可以把fixture放在它的protected域或者public域，这样每个测试函数都能用到。如果想每个测试函数共用同一个fixture，把它设置为static的类变量。另外，**如果定义了使用fixture的类，就要使用 TEST_F 而不是 TEST** 。

接下看看两个钩子函数，SetUp和Teardown。

修改 Gtest 定义为：

```
#include <iostream>

class Gtest : public ::testing::Test
{
protected:
    virtual void SetUp()
    {
        std::cout << "Hi \n";
    }
    virtual void TearDown()
    {
        std::cout << "Bye \n";
    }
    std::vector<int> v;
};
```

对应的输出是这样：

```
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from Gtest
[ RUN      ] Gtest.testItWorks
Hi
Bye
[       OK ] Gtest.testItWorks (0 ms)
[ RUN      ] Gtest.testVector
Hi
Bye
[       OK ] Gtest.testVector (0 ms)
[----------] 2 tests from Gtest (0 ms total)

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

断言方法

断言方法的丰富程度，决定了写测试代码时，需不需要做很多hack。
这里直接把各种方法贴出来：

基本断言

出错会停止测试的断言	出错不会停止测试的断言	意思
ASSERT_EQ(expected, actual)	EXPECT_EQ(expected, actual)	val1 == val2
ASSERT_NE(val1, val2)	EXPECT_NE(val1, val2)	val1 != val2
ASSERT_LT(val1, val2)	EXPECT_LT(val1, val2)	val1 < val2
ASSERT_LE(val1, val2)	EXPECT_LE(val1, val2)	val1 <= val2
ASSERT_GT(val1, val2)	EXPECT_GT(val1, val2)	val1 > val2
ASSERT_GE(val1, val2)	EXPECT_GE(val1, val2)	val1 >= val2
ASSERT_TRUE(condition)	EXPECT_TRUE(condition)	condition为true
ASSERT_FALSE(condition)	EXPECT_FALSE(condition)	condition为false

适用于C Style字符串的断言

出错会停止测试的断言	出错不会停止测试的断言	意思
ASSERT_STREQ(expected_str, actual_str)	EXPECT_STREQ(expected_str, actual_str)	两个字符串相等
ASSERT_STRNE(str1, str2)	EXPECT_STRNE(str1, str2)	两个字符串不等
ASSERT_STRCASEEQ(expected_str,actual_str)	EXPECT_STRCASEEQ(expected_str,actual_str)	两个字符串相等，忽略大小写
ASSERT_STRCASENE(str1, str2)	EXPECT_STRCASENE(str1, str2)	两个字符串不等，忽略大小写

适用于浮点数比较的断言

出错会停止测试的断言	出错不会停止测试的断言	意思
ASSERT_FLOAT_EQ(expected, actual)	EXPECT_FLOAT_EQ(expected, actual)	float类型近似相等
ASSERT_DOUBLE_EQ(expected, actual)	EXPECT_DOUBLE_EQ(expected, actual)	double类型近似相等

近似相等的断言

出错会停止测试的断言	出错不会停止测试的断言	意思
ASSERT_NEAR(val1, val2, abs_error)	EXPECT_NEAR(val1, val2, abs_error)	val1和val2的差的绝对值小于abs_error

是否抛出异常的断言

出错会停止测试的断言	出错不会停止测试的断言	意思
ASSERT_THROW(statement, exception_type)	EXPECT_THROW(statement, exception_type)	代码块会抛出指定异常
ASSERT_ANY_THROW(statement)	EXPECT_ANY_THROW(statement)	代码块会抛出异常

ASSERT_NO_THROW(statement)	EXPECT_NO_THROW(statement)	代码块不会抛出异常

借用官方示例：

```
ASSERT_THROW(Foo(5),bar_exception);

EXPECT_NO_THROW({
 int n=5;
 Bar(&n);
});
```

是不是感觉缺少了什么？

缺了比较C Style数组的断言呀！

有一次我想找个如何比较两个int类型数组的断言，却怎么也找不到。网上的解决办法，都是需要引入另外一个Google Mock框架。我不想再增加别的依赖，所以最后hack一下，把int类型数组转换成 `vector<int>` 。

我介绍的大概就这么多，最后借用gtest的README上的一句话：

> Happy testing!

c++ (/t/c%2B%2B/blogs) gtest (/t/gtest/blogs)

链接 (/a/1190000002454946) 更多

你可能感兴趣的文章

googletest (/a/1190000000393616) 1 收藏，748 浏览

C++判断Windows当前主题 (/a/1190000000573755) 561 浏览

判断一段程序是由C 编译程序还是由C++编译程序编译的 (/a/1190000000450516) 925 浏览

讨论区