GTest入门

Yeolar 2014-12-21 22:01

GTest是Google的一套用于编写C++测试的框架，可以运行在很多平台上（包括Linux、Mac OS X、Windows、Cygwin等等）。基于xUnit架构。支持很多好用的特性，包括自动识别测试、丰富的断言、断言自定义、死亡测试、非终止的失败、生成XML报告等等。

目录

GTest有哪些优点？

好的测试应该有下面的这些特点，我们看看GTest是如何满足要求的。

1. 测试应该是独立的、可重复的。一个测试的结果不应该作为另一个测试的前提。GTest中每个测试运行在独立的对象中。如果某个测试失败了，可以单独地调试它。
2. 测试应该是有清晰的结构的。GTest的测试有很好的组织结构，易于维护。
3. 测试应该是可移植和可复用的。有很多代码是不依赖平台的，因此它们的测试也需要不依赖于平台。GTest可以在多种操作系统、多种编译器下工作，有很好的可移植性。

4. 测试失败时，应该给出尽可能详尽的信息。GTest在遇到失败时并不停止接下来的测试，而且还可以选择使用非终止的失败来继续执行当前的测试。这样一次可以测试尽可能多的问题。
5. 测试框架应该避免让开发者维护测试框架相关的东西。GTest可以自动识别定义的全部测试，你不需要一一列举它们。
6. 测试应该够快。GTest在满足测试独立的前提下，允许你复用共享数据，它们只需创建一次。

GTest采用的是xUnit架构，你会发现和JUnit、PyUnit很类似，所以上手非常快。

搭建一个新的测试工程

使用GTest需要把它编译成库，然后链接到你的测试上。GTest的包里已经提供了一些常用的构建系统的构建文件：

- msvc/ - Visual Studio
- xcode/ - Mac Xcode
- make/ - GNU make
- codegear/ - Borland C++ Builder
- CMakeLists.txt - CMake（还有autotools脚本，不过已不推荐使用）

如果你的构建环境不是这些，那就研究一下 make/Makefile 文件吧。

创建你的测试工程时，需要确保 GTEST_ROOT/include 在头文件查找路径中。

可以参考GTest中自带的测试的构建方法。

在Debian系统中，可以使用包管理器中的GTest。注意只有 libgtest-dev 包，为了方便可以编译一份静态库放到系统的库路径下。

```
# cd /usr/src/gtest/
# mkdir build && cd build
# cmake ..
# make
# cp libgtest* /usr/local/lib/
# cd .. && rm -r build
```

然后，如果使用CMake中自带的 FindGTest.cmake，可以像下面这样写CMake脚本：

```
enable_testing()
find_package(GTest REQUIRED)
include_directories(${GTEST_INCLUDE_DIRS})

add_executable(foo foo.cc foo_unittest.cc)
target_link_libraries(foo gtest_main gtest pthread)

add_test(FooTest foo)
```

基本概念

使用GTest，可以从写断言开始。断言的结果有三种：成功、非终止的失败、终止的失败。终止的失败会终止当前的测试函数。

在被测试的代码崩溃或者断言失败时，测试失败，否则成功。

一个测试用例可以包含多个测试。编写良好的测试应该分组放到不同的测试用例中。测试用例中的测试需要共享数据时，可以使用测试类。

一个测试程序可以包含多个测试用例。

断言

GTest的断言是和函数类似的宏。断言失败时，GTest打印断言的所在文件和行号，以及失败信息。可以添加自定义的失败信息。

断言有两种。 ASSERT_* 的版本会终止当前函数， EXPECT_* 的版本则不会终止。一般来说 EXPECT_* 更常用，因为它可以输出测试的多个失败。如果你确实需要在断言失败时终止当前函数，那么就选择 ASSERT_* 。

因为 ASSERT_* 会立即终止，所以可能跳过了后面的清理工作，这可能导致内存泄漏。一般来说这没什么关系，只要在内存检测工具报错的时候知道原因就可以。

添加自定义的失败信息，只需要像流那样使用 << 添加即可。比如：

```
ASSERT_EQ(x.size(), y.size()) << "Vectors x and y are of unequal length";

for (int i = 0; i < x.size(); ++i) {
  EXPECT_EQ(x[i], y[i]) << "Vectors x and y differ at index " << i;
}
```

任何可以传给 ostream 的东西都可以传给断言宏。注意对于C字符串或者 string ，如果传给断言的是宽字符串（ wchar_t*, TCHAR*, std::wstring ），它会被转换成UTF-8。

基本的断言

下面的断言执行基本的true/false条件测试。

终止断言	非终止断言	验证
ASSERT_TRUE(condition);	EXPECT_TRUE(condition);	condition为true
ASSERT_FALSE(condition);	EXPECT_FALSE(condition);	condition为false

Linux、Windows、Mac可用。

二元比较

下面的断言比较两个值。

终止断言	非终止断言	验证
ASSERT_EQ(expected, actual);	EXPECT_EQ(expected, actual);	expected == actual
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

终止断言	非终止断言	验证
ASSERT_GE(val1, val2);	EXPECT_GE(val1, val2);	val1 >= val2

失败时，GTest会打印val1和val2。在 ASSERT_EQ* 和 EXPECT_EQ* （包括后面其他的等价性断言）中，应该把想要测试的表达式放在actual的位置，把期望值放在expected的位置，因为GTest会据此优化失败信息。

参数需要支持比较操作符，1.6.0版本以下的GTest还需要支持 << 操作符。

这些断言还支持定义了对应比较操作符的用户自定义类型。 ASSERT_* 宏能在打印结果的同时打印操作数。

参数只用一次，所以不用担心参数有副作用，但是仍要注意参数的执行顺序是不确定的。

ASSERT_EQ() 执行指针比较。如果是两个C字符串，测试的是它们是否有相同的内存地址。比较C字符串可以使用 ASSERT_STREQ() ，判断C字符串是否为 NULL 使用 ASSERT_STREQ(NULL, c_string) 。对于 string 对象的比较，仍然使用 ASSERT_EQ 。

这些宏对于宽字符串对象同样有效（ wstring ）。

Linux、Windows、Mac可用。

字符串比较

下面的断言比较两个C字符串。比较两个 string 对象，使用 EXPECT_EQ, EXPECT_NE 那些。

终止断言	非终止断言	验证
ASSERT_STREQ(expected_str, actual_str);	EXPECT_STREQ(expected_str, actual_str);	两个C字符串内容相同
ASSERT_STRNE(str1, str2);	EXPECT_STRNE(str1, str2);	两个C字符串内容不同
ASSERT_STRCASEEQ(expected_str, actual_str);	EXPECT_STRCASEEQ(expected_str, actual_str);	忽略大小写，两个C字符串内容相同

终止断言	非终止断言	验证
ASSERT_STRCASENE(str1, str2);	EXPECT_STRCASENE(str1, str2);	忽略大小写，两个C字符串内容不同

注意断言名字中的 CASE 代表是忽略大小写的。

STREQ 和 STRNE 也支持宽C字符串（ wchar_t* ）。比较失败的话，会以UTF-8编码来打印。

NULL 指针和空字符串是不同的。

Linux、Windows、Mac可用。

简单的测试

创建一个测试：

1. 使用 TEST() 宏定义和命名一个测试函数。它们就是普通的C++的无返回值函数。
2. 函数中可以使用任何C++表达式，以及GTest中的断言。
3. 如果任一断言失败了（终止或非终止的），或者如果测试崩溃了，该测试失败；反之成功。

```
TEST(test_case_name, test_name) {
 ... test body ...
}
```

TEST() 的第一个参数是测试用例名，第二个参数是测试用例中的测试名。必须是有效的C++标识符，并且不能包含下划线。测试的全名由这两个名字组成。不同测试用例的测试的名字可以相同。

比如下面这个整数函数：

```
int Factorial(int n); // Returns the factorial of n
```

它的测试用例可能是像这样：

```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput) {
  EXPECT_EQ(1, Factorial(0));
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
  EXPECT_EQ(1, Factorial(1));
  EXPECT_EQ(2, Factorial(2));
  EXPECT_EQ(6, Factorial(3));
  EXPECT_EQ(40320, Factorial(8));
}
```

GTest会把测试结果按照测试用例来组织，所以相关的测试应该放在相同的测试用例中，即第一个参数相同。

Linux、Windows、Mac可用。

捆绑测试：给多个测试使用相同的数据配置

如果有多个测试使用类似的数据，可以使用捆绑测试（text fixture）。它允许几个不同的测试复用相同的配置。

创建一个捆绑：

1. 从 ::testing::Test 派生一个类。使用 protected: 或 public: ，因为我们需要能从子类访问捆绑的成员。
2. 在类中声明任何你想用的对象。
3. 如果有必要，实现默认构造函数或者 SetUp() 函数来为测试准备数据。
4. 如果有必要，实现一个析构函数或者 TearDown() 函数来释放在 SetUp() 中分配的资源。
5. 如果需要，定义用于共享的子例程。

用 TEST_F() 代替 TEST() ，这样就可以访问捆绑测试中的对象和子例程了：

```
TEST_F(test_case_name, test_name) {
 ... test body ...
}
```

TEST_F() 的第一个参数也是测试用例名，它必须是捆绑测试类的类名。

别忘了必须在使用捆绑测试类之前定义它，否则编译时会报错“virtual outside class declaration”。

对每个 TEST_F() 定义的测试，GTest会：

1. 在运行时会创建一个新的捆绑测试
2. 用 SetUp() 初始化它
3. 运行测试
4. 调用 TearDown() 清理
5. 删除这个捆绑测试。注意同一测试用例中不同的测试有不同的捆绑测试对象，GTest创建下一个之前会删除前一个，不重用相同的。所以它们是互不影响的。

作为例子，下面给出了一个FIFO队列 —— Queue 的测试：

```
template <typename E> // E is the element type.
class Queue {
 public:
  Queue();
  void Enqueue(const E& element);
  E* Dequeue(); // Returns NULL if the queue is empty.
  size_t size() const;
  ...
};
```

首先定义一个捆绑类。

```
class QueueTest : public ::testing::Test {
 protected:
  virtual void SetUp() {
    q1_.Enqueue(1);
    q2_.Enqueue(2);
    q2_.Enqueue(3);
  }

  // virtual void TearDown() {}

  Queue<int> q0_;
  Queue<int> q1_;
  Queue<int> q2_;
};
```

这个例子中不需要实现 TearDown() ，默认的析构函数就可以完成清理。

现在我们用它和 TEST_F() 写测试。

```
TEST_F(QueueTest, IsEmptyInitially) {
  EXPECT_EQ(0, q0_.size());
}

TEST_F(QueueTest, DequeueWorks) {
  int* n = q0_.Dequeue();
  EXPECT_EQ(NULL, n);

  n = q1_.Dequeue();
  ASSERT_TRUE(n != NULL);
  EXPECT_EQ(1, *n);
  EXPECT_EQ(0, q1_.size());
  delete n;

  n = q2_.Dequeue();
  ASSERT_TRUE(n != NULL);
  EXPECT_EQ(2, *n);
  EXPECT_EQ(1, q2_.size());
  delete n;
}
```

上面同时使用了 ASSERT_* 和 EXPECT_* 断言，注意它们的使用原则。使用 ASSERT_TRUE 的地方是因为如果失败，就不应该继续下面的测试了。

运行测试时：

1. GTest构造一个 QueueTest 对象（就叫 t1 ）。
2. t1.SetUp() 初始化 t1 。
3. 在 t1 上运行第一个测试 IsEmptyInitially 。
4. 测试完成后，由 t1.TearDown() 做清理。
5. t1 被析构。
6. 在另一个 QueueTest 对象上重复上面步骤，运行 DequeueWorks 测试。

Linux、Windows、Mac可用。

调用测试

TEST() 和 TEST_F() 会自动注册测试。因此不需要再把它们重新列一遍。

定义完测试后，可以用 RUN_ALL_TESTS() 来运行它们，如果全部测试都通过了，返回0，否则返回1。注意 RUN_ALL_TESTS() 会运行链接的全部测试，可以来自不同的源文件、不同的测试用例。

RUN_ALL_TESTS() 宏被调用的时候：

1. 保存所有GTest flag的状态。
2. 为第一个测试创建捆绑测试对象。
3. 用 SetUp() 初始化它。
4. 在捆绑对象上运行测试。
5. 用 TearDown() 清理捆绑对象。
6. 删除捆绑对象。
7. 恢复所有GTest flag的状态。
8. 对后面的每个测试重复以上步骤。

如果第2步的捆绑测试的构造函数发生了一个终止失败，第3到5步会被跳过。类似地，第3步发生终止，第4步会被跳过。

> **重要**
>
> 不能忽略 RUN_ALL_TESTS() 的返回值，否则 gcc 会报错。这么设计是因为测试框架是通过退出码来判断测试是否通过，而不是根据输出。所以 main() 函数必须返回 RUN_ALL_TESTS() 的值。

RUN_ALL_TESTS() 只能调用一次。多次调用会导致GTest的一些特性的冲突（比如线程安全的死亡测试）。

Linux、Windows、Mac可用。

编写 main() 函数

可以参考下面的样板：

```
#include "this/package/foo.h"
#include "gtest/gtest.h"

namespace {

// The fixture for testing class Foo.
class FooTest : public ::testing::Test {
 protected:
  // You can remove any or all of the following functions if its body
  // is empty.

  FooTest() {
    // You can do set-up work for each test here.
  }

  virtual ~FooTest() {
    // You can do clean-up work that doesn't throw exceptions here.
  }

  // If the constructor and destructor are not enough for setting up
  // and cleaning up each test, you can define the following methods:

  virtual void SetUp() {
    // Code here will be called immediately after the constructor (right
    // before each test).
  }

  virtual void TearDown() {
    // Code here will be called immediately after each test (right
    // before the destructor).
  }

  // Objects declared here can be used by all tests in the test case for Foo.
};

// Tests that the Foo::Bar() method does Abc.
TEST_F(FooTest, MethodBarDoesAbc) {
  const string input_filepath = "this/package/testdata/myinputfile.dat";
  const string output_filepath = "this/package/testdata/myoutputfile.dat";
```

```
  Foo f;
  EXPECT_EQ(0, f.Bar(input_filepath, output_filepath));
}

// Tests that Foo does Xyz.
TEST_F(FooTest, DoesXyz) {
  // Exercises the Xyz feature of Foo.
}

} // namespace

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

::testing::InitGoogleTest() 函数解析命令行，读取GTest flag，然后删掉它们。这样可以通过各种flag来控制测试程序的行为，细节可以参考 AdvancedGuide (https://code.google.com/p/googletest/wiki/V1_7_AdvancedGuide) 。必须在调用 RUN_ALL_TESTS() 之前调用这个函数，否则flag不会正常初始化。

在Windows上， InitGoogleTest() 也支持宽字符串，因此也可以用于 UNICODE 模式编译的程序。

也许你觉得写这些 main() 函数仍然很麻烦？确实如此，所以GTest中提供了一个基本的 main() 的实现。如果能满足你的需要，只要把你的测试链接上 gtest_main 库即可。

Visual C++用户的注意事项

如果你把测试和 main() 函数分开放在不同的库中，测试不会正常执行。这是Visual C++的一个 bug (https://connect.microsoft.com/feedback/viewfeedback.aspx?FeedbackID=244410&siteid=210) 。定义了测试时，GTest会创建某些静态对象来注册它们。这些对象别处没有引用，但仍然会有构造。当Visual C++链接器发现库在别处没有引用的时候就把它丢弃了。因此需要在主程序中引用带有测试的库以免它被丢了。下面是具体的办法。在库的代码里面声明一个函数：

```
__declspec(dllexport) int PullInMyLibrary() { return 0; }
```

如果测试是放在静态库（而不是DLL）中，就不需要 __declspec(dllexport) 了。现在，在主程序中调用它：

```
int PullInMyLibrary();
static int dummy = PullInMyLibrary();
```

这可以保证测试被引用，并在启动时注册。

如果是在静态库中定义测试，在主程序的链接选项中添加 /OPT:NOREF 。使用MSVC++ IDE的话，在 *.exe project properties/Configuration Properties/Linker/Optimization 中设置 References 为 Keep Unreferenced Data (/OPT:NOREF) 。这会使Visual C++链接器生成可执行文件时不丢弃符号。

还有一个陷阱。如果以静态库的方式使用GTest，测试也需要在静态库中。如果需要放在DLL中，那么GTest也得编译成DLL。否则不能正常工作。

所以最好的办法是：别在库里面写测试。

其他

GTest被设计为线程安全的，只要系统中 pthreads 库可用即可保证这一点。

学习使用GTest可以从它自带的 例子 (https://code.google.com/p/googletest/wiki/V1_7_Samples) 开始。AdvancedGuide (https://code.google.com/p/googletest/wiki/V1_7_AdvancedGuide) 中列举了更多的特性。

GTest的包里面还带了一个元编程框架，可以用来生成重复代码。感兴趣的话，可以参考脚本 scripts/pump.py 和 PumpManual (https://code.google.com/p/googletest/wiki/V1_7_PumpManual) 。

http://www.yeolar.com/note/2014/12/21/gtest/