Google C++ TestingFramework(gtest)
c++单元测试培训

２０２１年１２月

目录

什么是gtest：

Google C++ TestingFramework(简称gtest)是一个跨平台的(Liunx、Mac OS X、Windows 、Cygwin 、Windows CE and Symbian) **C++单元测试框架**，由google公司发布。gtest是为在不同平台上为编写C++测试而生成的。它提供了丰富的断言、致命和非致命判断、参数化、”死亡测试”等等。已被应用于多个开源项目及Google内部项目中，知名的例子包括ChromeWeb浏览器、LLVM 编译器架构、ProtocolBuffers数据交换格式及工具等。

优秀的C/C++单元测试框架并不算少，相比之下gtest仍具有明显优势。与CppUnit比，gtest需要使用的头文件，函数和宏更集中，并支持测试用例的自动注册。与CxxUnit比，gtest不要求Python等外部工具的存在。与Boost.Test相比，gtest更简洁容易上手，实用性也并不逊色。

gtest安装(linux)：

下载gtest，Release 1.11.0

```
git clone https://github.com/google/googletest
```

gtest编译

```
cd googletest
```

生成Makefile文件(先安装cmake，brew install cmake)，继续输入命令编译:

```
cmake CMakeLists.txt
```

执行make，生成两个静态库：libgtest.a libgtest_main.a

```
make
```

拷贝到系统目录，注意，如果下诉目录位置在不同版本位置有变动，用find . -name "libgtest*.a" 找到位置

```
sudo cp libgtest*.a  /usr/lib
sudo cp –a include/gtest /usr/include
```

gtest简单示例

示例代码：

```
#include <gtest/gtest.h>
int add(int a, int b)
{
	return a + b;
}
TEST(testAddFunction, testAdd)
{
	EXPECT_EQ(add(2, 3),5);
}
int main(int argc, char **argv)
{
	testing::InitGoogleTest(&argc, argv);
	return RUN_ALL_TESTS();
}
```

编译&&运行

```
./a.out
```

执行结果如下：

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from testAddFunction
[ RUN      ] testAddFunction.testAdd
[       OK ] testAddFunction.testAdd (0 ms)
[----------] 1 test from testAddFunction (0 ms total)

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

分析：

1：使用了TEST这个宏，它有两个参数：

test_case_name第一个参数是测试用例名,通常是取测试函数名或者测试类名。

test_name 第二个参数是测试名这个随便取，但最好取有意义的名称。

当测试完成后显示的测试结果将以"测试用例名.测试名"的形式给出。

2：使用了EXPECT_EQ这个宏，这个宏用来比较两个数字是否相等。

3：函数：InitGoogleTest 函数原型

```
GTEST_API_ void InitGoogleTest(int* argc, char** argv);
 // support UNICODE mode.
GTEST_API_ void InitGoogleTest(int* argc, wchar_t** argv);
```

作用：

InitGoogleTest的实现是由InitGoogleTestImpl实现的，在test::internal命名空间下：

1. 判断是否初始化过以及参数是否为空；
2. 保存所有的传入参数到g_argvs中；
3. 检验参数的合法性及打印help信息；
4. 解析的参数对UnitTestImpl进行相关初始化操作。

4：“RUN_ALL_TESTS()” ：运行所有测试案例

断言：

断言（assert）是一个宏，该宏在＜assert＞中当使用assert时候，给他个参数，即一个判读为真的表达式。）

gtest中，断言的宏可以理解为分为两类，一类是ASSERT系列，一类是EXPECT系列。一个直观的解释就是：

1. ASSERT_* 系列的断言，当检查点失败时，退出当前测试套件。
2. EXPECT_* 系列的断言，当检查点失败时，继续往下执行。

1：布尔值检查

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_TRUE(*condition*);	EXPECT_TRUE(*condition*);	*condition*is true
ASSERT_FALSE(*condition*);	EXPECT_FALSE(*condition*);	*condition*is false

2、数值型数据检查

Fatal assertion	Nonfatal assertion	Verifies
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*
ASSERT_GE(*val1*, *val2*);	EXPECT_GE(*val1*, *val2*);	*val1*>=*val2*

3、字符串检查

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_STREQ(*expected_str*, *actual_str*);	EXPECT_STREQ(*expected_str*, *actual_str*);	the two C strings have the same content
ASSERT_STRNE(*str1*, *str2*);	EXPECT_STRNE(*str1*, *str2*);	the two C strings have different content
ASSERT_STRCASEEQ(*expected_str*, *actual_str*);	EXPECT_STRCASEEQ(*expected_str*, *actual_str*);	the two C strings have the same content, ignoring case
ASSERT_STRCASEEQ(*expected_str*, *actual_str*);	EXPECT_STRCASEEQ(*expected_str*, *actual_str*);	the two C strings have the same content, ignoring case
ASSERT_STRCASENE(*str1*, *str2*);	EXPECT_STRCASENE(*str1*, *str2*);	the two C strings have different content, ignoring case

*

4、异常检查

Fatal assertion	Nonfatal assertion	Verifies
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

5、Predicate Assertions

在使用EXPECT_TRUE或ASSERT_TRUE时，有时希望能够输出更加详细的信息，比如检查一个函数的返回值TRUE还是FALSE时，希望能够输出传入的参数是什么，以便失败后好跟踪。因此提供了如下的断言：

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_PRED1(*pred1, val1*);	EXPECT_PRED1(*pred1, val1*);	*pred1(val1)*returns true
ASSERT_PRED2(*pred2, val1, val2*);	EXPECT_PRED2(*pred2, val1, val2*);	*pred2(val1, val2)*returns true
…	…	…

6、浮点型检查

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_FLOAT_EQ(*expected, actual*);	EXPECT_FLOAT_EQ(*expected, actual*);	the two floatvalues are almost equal
ASSERT_DOUBLE_EQ(*expected, actual*);	EXPECT_DOUBLE_EQ(*expected, actual*);	the two doublevalues are almost equal

对相近两个数的比较：

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_NEAR(*val1, val2, abs_error*);	EXPECT_NEAR(*val1, val2, abs_error*);	the difference between *val1*and *val2*doesn't exceed the given absolute error

事件机制

gtest单元测试程序框架如下图：

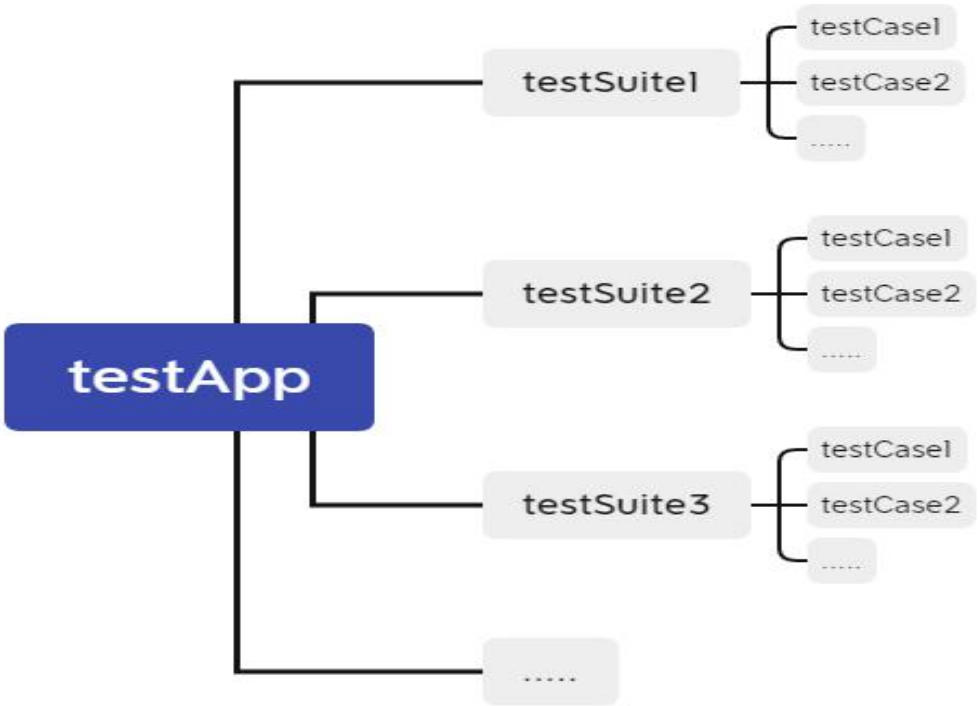


图 1

gtest提供了三种事件机制，通过事件机制我们可以在案例之间共享一些通用方法，共享资源。

我们现在测试vector容器的两个方法empty和size

TEST测试：

示例：

testVector.cpp

测试结果：

```
root@lilongiu-VirtualBox:~/study/g_test/test# g++  testVector.cpp -lgtest -lpthread
root@lilongiu-VirtualBox:~/study/g_test/test# ./a.out
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from testVector
[ RUN      ] testVector.isEmpty
[       OK ] testVector.isEmpty (0 ms)
[ RUN      ] testVector.size
[       OK ] testVector.size (0 ms)
[----------] 2 tests from testVector (0 ms total)

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

通过上述代码我们可以知道在TEST宏定义中我们编写了重复代码，但是通过事件机制，我们可以消除重复。

1：个体事件机制：个体事件机制（针对的是对应图 1 的testCase（测试用例）），实现个体事件需要写一个类，继承testing::Test，然后实现两个方法SetUp()和TearDown()。

a. SetUp()方法在每个TestCase之前执行
b. TearDown()方法在每个TestCase之后执行

示例：

testCaseVector.cpp

测试结果：

```
root@lilongiu-VirtualBox:~/study/g_test/test# g++  testCaseVector.cpp -lgtest -lpthread
root@lilongiu-VirtualBox:~/study/g_test/test# ./a.out
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from testVector
[ RUN      ] testVector.isEmpty
Setup()
Teardown()
[       OK ] testVector.isEmpty (0 ms)
[ RUN      ] testVector.size
Setup()
Teardown()
[       OK ] testVector.size (0 ms)
[----------] 2 tests from testVector (0 ms total)

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

2：局部事件机制：局部事件机制（针对的是对应图 1 testSuite（测试套件）），实现局部事件需要写一个类，继承testing::Test，然后实现两个静态方法SetUpTestCase()和TearDownTestCase()。

a. SetUpTestCase() 方法在同一个testSuite的第一个TestCase之前执行

b. TearDownTestCase() 方法在同一个testSuite的最后一个TestCase之后执行

示例：

testSuiteVector.cpp

测试结果：

```
root@lilongiu-VirtualBox:~/study/g_test/test# g++  testSuiteVector.cpp -lgtest -lpthread
root@lilongiu-VirtualBox:~/study/g_test/test# ./a.out
[==========] Running 3 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 2 tests from testVector
setUpTestCase()
[ RUN      ] testVector.isEmpty
[       OK ] testVector.isEmpty (0 ms)
[ RUN      ] testVector.size
[       OK ] testVector.size (0 ms)
tearDownTestCase()
[----------] 2 tests from testVector (0 ms total)

[----------] 1 test from testcase
[ RUN      ] testcase.case0
[       OK ] testcase.case0 (0 ms)
[----------] 1 test from testcase (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 2 test suites ran. (0 ms total)
[  PASSED  ] 3 tests.
```

3： 全局事件机制：全局事件机制（针对的是对应图 1 的testApp），要实现全局事件，必须写一个类，继承testing::Environment类，实现里面的SetUp TearDown方法。

a. SetUp()方法在所有案例执行前执行

b. TearDown()方法在所有案例执行后执行

示例：testGlobleVector.cpp

测试结果：

```
root@lilongiu-VirtualBox:~/study/g_test/test# g++  testGlobleVector.cpp -lgtest -lpthread
root@lilongiu-VirtualBox:~/study/g_test/test# ./a.out
[==========] Running 3 tests from 2 test suites.
[----------] Global test environment set-up.
 golble Setup()
[----------] 2 tests from testVector
setUpTestCase()
[ RUN      ] testVector.isEmpty
[       OK ] testVector.isEmpty (0 ms)
[ RUN      ] testVector.size
[       OK ] testVector.size (0 ms)
tearDownTestCase()
[----------] 2 tests from testVector (0 ms total)

[----------] 1 test from testcase
[ RUN      ] testcase.case0
[       OK ] testcase.case0 (0 ms)
[----------] 1 test from testcase (0 ms total)

[----------] Global test environment tear-down
golble Teardown()
[==========] 3 tests from 2 test suites ran. (0 ms total)
[  PASSED  ] 3 tests.
```

参数化：

在设计测试案例时，经常需要考虑给被测函数传入不同的值的情况。我们之前的做法通常是写一个通用方法，然后编写在测试案例调用它。即使使用了通用方法，这样的工作也是有很多重复性的，程序员都懒，都希望能够少写代码，多复用代码。Google的程序员也一样，他们考虑到了这个问题，并且提供了一个灵活的参数化测试的方案。

我们用一个函数来判断一个整形数是不是质数

1：不用参数化的方法

示例：

test.cpp

2:使用参数化测试

示例：

testParaInit.cpp

测试结果：

```
root@lilongiu-VirtualBox:~/study/g_test/test# g++  testParaInit.cpp -lgtest -lpthread
root@lilongiu-VirtualBox:~/study/g_test/test# ./a.out
[==========] Running 5 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 5 tests from TrueReturn/IsPrimeParamTest
[ RUN      ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/0
[       OK ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/0 (0 ms)
[ RUN      ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/1
[       OK ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/1 (0 ms)
[ RUN      ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/2
[       OK ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/2 (0 ms)
[ RUN      ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/3
[       OK ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/3 (0 ms)
[ RUN      ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/4
[       OK ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/4 (0 ms)
[----------] 5 tests from TrueReturn/IsPrimeParamTest (0 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 5 tests.
```

从上图中的案例名称大概能够看出案例的命名规则大概为：

prefix/test_case_name.test.name/index

死亡测试：

通常在测试过程中，我们需要考虑各种各样的输入，有的输入可能直接导致程序崩溃，这时我们就需要检查程序是否按照预期的方式挂掉，这也就是所谓的“死亡测试”。gtest的死亡测试能做到在一个安全的环境下执行崩溃的测试案例，同时又对崩溃结果进行验证。

使用的宏：

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_DEATH(*statement, regex*`);`	EXPECT_DEATH(*statement, regex*`);`	*statement*crashes with the given error
ASSERT_EXIT(*statement, predicate, regex*`);`	EXPECT_EXIT(*statement, predicate, regex*`);`	*statement*exits with the given error and its exit code matches *predicate*

由于有些异常只在Debug下抛出，因此还提供了*_DEBUG_DEATH，用来处理Debug和Realease下的不同。

*_DEATH(statement, regex`)

1. statement是被测试的代码语句
2. regex是一个正则表达式，用来匹配异常时在stderr中输出的内容

示例：

testDeath.cpp

测试结果：

```
root@lilongiu-VirtualBox:~/study/g_test/test# g++  testDeath.cpp -lgtest -lpthread
root@lilongiu-VirtualBox:~/study/g_test/test# ./a.out
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from FooDeathTest
[ RUN      ] FooDeathTest.Demo
[       OK ] FooDeathTest.Demo (876 ms)
[----------] 1 test from FooDeathTest (876 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (877 ms total)
[  PASSED  ] 1 test.
```

提示：编写死亡测试案例时，TEST的第一个参数，即testcase_name，请使用DeathTest后缀。原因是gtest会优先运行死亡测试案例，应该是为线程安全考虑。

*_EXIT(statement, predicate, regex`)

1. statement是被测试的代码语句
2. predicate 在这里必须是一个委托，接收int型参数，并返回bool。只有当返回值为true时，死亡测试案例才算通过。gtest提供了一些常用的predicate：
   testing::ExitedWithCode(exit_code)
   如果程序正常退出并且退出码与exit_code相同则返回 true
   testing::KilledBySignal(signal_number) // Windows下不支持
   如果程序被signal_number信号kill的话就返回true
3. regex是一个正则表达式，用来匹配异常时在stderr中输出的内容
   这里， 要说明的是，*_DEATH其实是对*_EXIT进行的一次包装，*_DEATH的predicate判断进程是否以非0退出码退出或被一个信号杀死。

例子：

```
TEST(ExitDeathTest, Demo)
{
   EXPECT_EXIT(_exit(1),  testing::ExitedWithCode(1),  "");
}
```

*_DEBUG_DEATH

定义：

```
#ifdef NDEBUG
#define EXPECT_DEBUG_DEATH(statement, regex) \
  do { statement; } while (false)
#define ASSERT_DEBUG_DEATH(statement, regex) \
  do { statement; } while (false)
#else
#define EXPECT_DEBUG_DEATH(statement, regex) \
  EXPECT_DEATH(statement, regex)
#define ASSERT_DEBUG_DEATH(statement, regex) \
  ASSERT_DEATH(statement, regex)
#endif
```

可以看到，在Debug版和Release版本下， *_DEBUG_DEATH的定义不一样。因为很多异常只会在Debug版本下抛出，而在Realease版本下不会抛出，所以针对Debug和Release分别做了不同的处理。

运行参数

使用gtest编写的测试案例通常本身就是一个可执行文件，因此运行起来非常方便。同时，gtest也为我们提供了一系列的运行参数（环境变量、命令行参数或代码里指定），使得我们可以对案例的执行进行一些有效的控制。

对于运行参数，gtest提供了三种设置的途径：

1. 系统环境变量
2. 命令行参数
3. 代码中指定FLAG

因为提供了三种途径，就会有优先级的问题， 有一个原则是，最后设置的那个会生效。不过总结一下，通常情况下，比较理想的优先级为：

命令行参数 > 代码中指定FLAG > 系统环境变量

为什么我们编写的测试案例能够处理这些命令行参数呢？是因为我们在main函数中，将命令行参数交给了gtest，由gtest来搞定命令行参数的问题。

示例：将输出保存到xml 

testParaInit.cpp

测试结果： 
a.out.xml

深入解析gtest：

1: TEST

2：RUN_ALL_TESTS宏

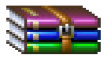
TEST宏.png

打造自己的单元测试框架：

示例：

testmyGtest.cpp

下载资料：


gtest教程.rar


gtest-1.3.0.zip

推荐链接：（死亡测试）
（事件机制）
（Gtest各种测试示例）


谢谢观看！