个人资料

breaksoftware

访问：673823次
积分：8696
等级：BLOG 6
排名：第2191名

原创：201篇
转载：1篇
译文：0篇
评论：441条

Google Test(GTest)使用方法和源码解析——概况

2016-04-07 23:53 4508人阅读 评论(0) 收藏 举报

分类： GTest使用方法和源码解析（10）

GTest是很多开源工程的测试框架。虽然介绍它的博文非常多，但是我觉得可以深入到源码层来解析它的实现原理以及使用方法。这样我们不仅可以在开源工程中学习到实用知识，还能学习到一些思想和技巧。我觉得有时候思想和技巧是更重要的。（转载请指明出于breaksoftware的csdn博客）

我们即将要分析的是GTest1.7版本。我们可以通过https://github.com/google/googletest.Git得到代码。

官方文档见：

- https://github.com/google/googletest/blob/master/googletest/docs/Primer.md
- https://github.com/google/googletest/blob/master/googletest/docs/AdvancedGuide.md

我们先大致熟悉一下GTest的特性。GTest和很多开源工程一样，并不只是针对特定的平台，否则其使用范围将大打折扣，所以GTest具有很好的移植特性和可复用性，我们以工程中的代码为例

```cpp
template <class T, typename Result>
Result HandleSehExceptionsInMethodIfSupported(
    T* object, Result (T::*method)(), const char* location) {
#if GTEST_HAS_SEH
  __try {
    return (object->*method)();
  } __except (internal::UnitTestOptions::GTestShouldProcessSEH(  // NOLINT
      GetExceptionCode())) {
    std::string* exception_message = FormatSehExceptionMessage(
        GetExceptionCode(), location);
    internal::ReportFailureInUnknownLocation(TestPartResult::kFatalFailure,
                                             *exception_message);
    delete exception_message;
    return static_cast<Result>(0);
  }
#else
  (void)location;
  return (object->*method)();
#endif  // GTEST_HAS_SEH
}
```

这段代码只是为了执行模板类T对象的method函数指针指向的方法。其核心就是(object->*method)()这句，但是它却使用了20行的代码去实现，就是为了解决平台的兼容问题。从名字我们可以看出它为了兼容SEH机制——结构化异常处理——一种windows系统上提供给用户处理异常的机制。而这种机制在linux系统上没有。这个函数是GTest为移植特性所做工作的一个很好的代表，我们将在之后的源码介绍中经常见到它的身影。

我们编码时，有时候我们不仅考究逻辑的严谨，还非常注意编码的风格和布局的优美。其实代码就像一件作品，一个不负责任的作者可能是毫无章法的涂鸦手法，而有些有着一定境界的程序员可能就会按照自己的“画风”去绘制——他的“画风”可能你并不喜欢，但是那种风格却是独立和鲜明的，甚至是有一定道理的。而且如果一旦“画风”确定，对于临摹者来说只要照着这样的套路去做，而不用自己发挥自己的风格，这对一个库的发展也是非常有益的。GTest同样有着良好的**组织结构**，我们以其自带的Sample1为例

```cpp
// Tests factorial of negative numbers.
TEST(FactorialTest, Negative) {
  EXPECT_EQ(1, Factorial(-5));
  EXPECT_EQ(1, Factorial(-1));
  EXPECT_GT(Factorial(-10), 0);
```

WMI技术介绍和应用 (24)
Apache服务搭建和插件实现 (7)
网络编程模型的分析、实现和对比 (6)
GTest使用方法和源码解析 (11)
PE文件结构和相关应用 (11)
windows安全 (9)
网络通信 (5)
沙箱 (7)
内嵌及定制Lua引擎技术 (3)
IE控件及应用 (7)
反汇编 (15)
开源项目 (16)
C++ (15)
界面库 (3)
python (11)
疑难杂症 (24)
PHP (8)
Redis (8)
IT项目研发过程中的利器 (4)
libev源码解析 (6)

文章存档

2017年08月 (7)
2017年07月 (4)
2017年05月 (9)
2017年02月 (1)
2016年12月 (10)
2016年11月 (3)
2016年10月 (5)
2016年06月 (6)
2016年05月 (3)
2016年04月 (11)
2016年02月 (5)
2016年01月 (3)
2015年12月 (1)
2015年11月 (1)
2015年09月 (1)
2015年06月 (2)
2015年05月 (1)
2015年03月 (2)
2015年02月 (4)
2014年12月 (1)
2014年09月 (3)
2014年08月 (4)
2014年07月 (4)
2014年06月 (1)
2014年04月 (4)
2013年12月 (4)
2013年10月 (4)
2013年04月 (2)
2013年03月 (4)
2013年02月 (6)
2013年01月 (12)
2012年12月 (4)
2012年11月 (13)
2012年09月 (5)
2012年08月 (4)
2012年07月 (4)
2012年06月 (8)
2012年05月 (1)
2012年04月 (5)
2012年01月 (1)
2011年12月 (3)

```
06. }
07.
08. // Tests factorial of 0.
09. TEST(FactorialTest, Zero) {
10.   EXPECT_EQ(1, Factorial(0));
11. }
12.
13. // Tests factorial of positive numbers.
14. TEST(FactorialTest, Positive) {
15.   EXPECT_EQ(1, Factorial(1));
16.   EXPECT_EQ(2, Factorial(2));
17.   EXPECT_EQ(6, Factorial(3));
18.   EXPECT_EQ(40320, Factorial(8));
19. }
```

这段代码是一套完整的测试代码。可以观察发现，每个逻辑使用一个TEST宏控制，其内部也是一系列EXPECT_*宏堆砌。先不论其他风格，单从整齐有规律的书写方式上来说，GTest也算是一个便于结构性编码的样板。我们使用者只要照着这样的样板去编写测试用例，是非常方便的，这也将大大降低我们使用GTest库的门槛。

TEST宏是一个很重要的宏，它构成一个测试特例。现在有必要介绍下其构成，TEST宏的第一个参数是“测试用例名”，第二个参数是“测试特例名”。测试用例（Test Case）是为某个特殊目标而编制的一组测试输入、执行条件以及预期结果，以便测试某个程序路径或核实是否满足某个特定需求（引百度百科），测试特例是测试用例下的一组测试。以以上代码为例，三段TEST宏构成的是一个测试用例——测试用例名是FactorialTest（阶乘方法检测，测试Factorial函数），该用例覆盖了三种测试特例——Negative、Zero和Positive——即检测输入参数是负数、0和正数这三种特例情况。

我们再看一组检测素数的测试用例

```
[cpp]

01. TEST(IsPrimeTest, Negative) {
02.   // This test belongs to the IsPrimeTest test case.
03.   EXPECT_FALSE(IsPrime(-1));
04.   EXPECT_FALSE(IsPrime(-2));
05.   EXPECT_FALSE(IsPrime(INT_MIN));
06. }
07.
08. // Tests some trivial cases.
09. TEST(IsPrimeTest, Trivial) {
10.   EXPECT_FALSE(IsPrime(0));
11.   EXPECT_FALSE(IsPrime(1));
12.   EXPECT_TRUE(IsPrime(2));
13.   EXPECT_TRUE(IsPrime(3));
14. }
15.
16. // Tests positive input.
17. TEST(IsPrimeTest, Positive) {
18.   EXPECT_FALSE(IsPrime(4));
19.   EXPECT_TRUE(IsPrime(5));
20.   EXPECT_FALSE(IsPrime(6));
21.   EXPECT_TRUE(IsPrime(23));
22. }
```

这组测试用例的名是IsPrimeTest（测试IsPrime函数），三个测试特例是Negative（错误结果场景）、Trivial（有对有错的场景）和Positive（正确结果场景）。

对于测试用例名和测试特例名，不能有下划线（_）。因为GTest源码中需要使用下划线把它们连接成一个独立的类名

```
[cpp]

01. // Expands to the name of the class that implements the given test.
02. #define GTEST_TEST_CLASS_NAME_(test_case_name, test_name) \
03.   test_case_name##_##test_name##_Test
```

这样也就要求，我们不能有相同的“测试用例名和特例名”的组合——否则类名重合。

测试用例名和测试特例名的分开，使得我们编写的测试代码有着更加清晰的结构——即有相关性也有独立性。相关性是通过相同的测试用例名联系的，而独立性通过不同的测试特例名体现的。我们通过这段测试代码的运行结果查看一下这两个特性

```
[plain]

01. Running main() from gtest_main.cc
02. [==========] Running 6 tests from 2 test cases.
03. [----------] Global test environment set-up.
```

```
[----------] 3 tests from FactorialTest
[ RUN      ] FactorialTest.Negative
[       OK ] FactorialTest.Negative (0 ms)
[ RUN      ] FactorialTest.Zero
[       OK ] FactorialTest.Zero (0 ms)
[ RUN      ] FactorialTest.Positive
[       OK ] FactorialTest.Positive (0 ms)
[----------] 3 tests from FactorialTest (0 ms total)

[----------] 3 tests from IsPrimeTest
[ RUN      ] IsPrimeTest.Negative
[       OK ] IsPrimeTest.Negative (0 ms)
[ RUN      ] IsPrimeTest.Trivial
[       OK ] IsPrimeTest.Trivial (0 ms)
[ RUN      ] IsPrimeTest.Positive
[       OK ] IsPrimeTest.Positive (0 ms)
[----------] 3 tests from IsPrimeTest (0 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 2 test cases ran. (14 ms total)
[  PASSED  ] 6 tests.
```

从输出结果上，我们看到GTest框架将我们相同测试用例名的场景合并在一起，不同测试特例名的场景分开展现。而且我们还发现GTest有自**动统计结**果、自**动**格式化**输**出**结**果、自**动调**度**执**行等特性。这些特性也将是之后博文分析的重点。

虽然上例中，所有的执行都是正确的，但是如果以上测试中发生一个错误，也不能影响其他测试——不同测试用例不相互影响、相同测试用例不同测试特例不相互影响。我们称之为独立性。除了独立性，也不失灵活性，测试测试特例中可以通过不同宏（ASSERT_*类宏会影响之后执行，EXPECT_*类宏不会）控制是否影响之后的执行。

如果我们编写的测试用例组（如上例是两组）中一组发生了错误，我们希望出错的那组不用执行组再执行一遍。一般情况下，我们可能需要去删除执行正确的那段测试代码，但是这种方式非常不优美——需要编译或者忘记恢复代码。GTest框架可以让我们通过在程序参数控制执行哪个测试用例，比如我们希望只执行Factorial测试，就可以这样调用程序

```plain
./sample1_unittest --gtest_filter=Factorial*
```

我们可以将以上特性称之为**选择**性**测试**。

最后一个特性便是**预处**理。我们测试时，往往要构造复杂的数据。如果我们在每个测试特例中都要构造一遍数据，将是非常繁琐和不美观的。GTest提供了一种提前构建数据的方式。我们以如下代码为例

```cpp
class ListTest : public testing::Test {
 protected:
  virtual void SetUp() {
      _m_list[0] = 11;
      _m_list[1] = 12;
      _m_list[2] = 13;
  }
  int _m_list[3];
};
TEST_F(ListTest, FirstElement) {
  EXPECT_EQ(11, _m_list[0]);
}

TEST_F(ListTest, SecondElement) {
  EXPECT_EQ(12, _m_list[1]);
}

TEST_F(ListTest, ThirdElement) {
  EXPECT_EQ(13, _m_list[2]);
}
```

我们让ListTest类继承于GTest提供的基类testing::Test，并重载SetUp方法。这样我们每次执行ListTest的一个测试特例时，SetUp方法都会执行一次，从而将数据准备完毕。这样我们只要在一个类中构建好数据就行了。这儿需要注意一下TEST_F宏，它的第一参数要求是类名——即ListTest——不像TEST宏的第一个参数我们可以随便命名。

/breaksoftwa...

朴素、Select、Poll和Epoll网络编程

zhangcunli8499: 哥们，能传一下完整的代码吗？

C++拾趣——类构造函数的隐式转

breaksoftware: @wuchalilun:多谢鼓励，其实我就想写出点不一样的地方，哈哈。

C++拾趣——类构造函数的隐式转

Ray_Chang_988: 其他相关的explicit的介绍文章也看了，基本上explicit的作用也都解释清楚了，但是它们都没...

Redis源码解析——字典结构

breaksoftware: @u011548018:多谢鼓励

Redis源码解析——字典结构

生无可恋只能打怪升级: 就冲这图也得点1024个赞

WMI技术介绍和应用——查询系统

breaksoftware: @hobbyonline:我认为这种属性的信息不准确是很正常的，因为它的正确与否不会影响到系统在不同...

顶 踩

2 0

上一篇 WMI技术介绍和应用——总结（完）

下一篇 Google Test(GTest)使用方法和源码解析——自动调度机制分析

相关文章推荐

- Google Test(GTest)使用方法和源码解析——私有…
- 【直播】机器学习之凸优化--马博士
- Google Mock(Gmock)简单使用和源码分析——源…
- 【直播】计算机视觉原理及实战--屈教授
- Google Test(GTest)使用方法和源码解析——断言…
- 机器学习&数据挖掘7周实训--韦玮
- Google Test(GTest)使用方法和源码解析——Liste…
- 机器学习之数学基础系列--AI100
- Google Test(GTest)使用方法和源码解析——死亡…
- 【套餐】2017软考系统集成项目管理工程师顺利通…
- 用google mock模拟C++对象
- 【课程】深入探究Linux/VxWorks的设备树--宋宝华
- python3编写简易统计服务器
- Google Test(GTest)使用方法和源码解析——结果…
- Google Test(GTest)使用方法和源码解析——自定…
- Google Test(GTest)使用方法和源码解析——预处…

查看评论

暂无评论

发表评论

用 户 名： GreatProgramer

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场