个人资料

breaksoftware

访问：673860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——结果统计机制分析

2016-04-07 23:54 1698人阅读 评论(0) 收藏 举报

分类： GTest使用方法和源码解析（10）

在分析源码之前，我们先看一个例子。以《Google Test(GTest)使用方法和源码解析——概况》 [illegible] 实例代码为基准，修改最后一个"局部测试"结果为错误。(转载请指明出于breaksoftware的csdn博客

```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(0, _m_list[2]);
}
```

然后我们观察其输出，从下面的结果我们可以分析出GTest帮我们统计了：

- 有多少测试用例
- 一个测试用例中有多少测试特例
- 一个测试用例中有多少测试特例成功
- 一个测试用例中有多少测试特例失败
- 失败的原因、位置、期待结果、实际结果

```plain
Running main() from gtest_main.cc
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from ListTest
[ RUN      ] ListTest.FirstElement
[       OK ] ListTest.FirstElement (0 ms)
[ RUN      ] ListTest.SecondElement
[       OK ] ListTest.SecondElement (0 ms)
[ RUN      ] ListTest.ThirdElement
../samples/sample11_unittest.cc:86: Failure
      Expected: 0
To be equal to: _m_list[2]
      Which is: 13
[  FAILED  ] ListTest.ThirdElement (0 ms)
[----------] 3 tests from ListTest (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 2 tests.
[  FAILED  ] 1 test, listed below:
```

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)
展开

阅读排行

使用WinHttp接口实现HT (35595)
WMI技术介绍和应用—— (18359)
如何定制一款12306抢票 (13984)
一种准标准CSV格式的介 (12486)
一种精确从文本中提取UI (12203)
实现HTTP协议Get、Post (11999)
分析两种Dump(崩溃日志 (11576)
一种解决运行程序报"应用 (11171)
实现HTTP协议Get、Post (11158)
反汇编算法介绍和应用— (10676)

评论排行

使用WinHttp接口实现HT (33)
使用VC实现一个"智能"自 (27)
WMI技术介绍和应用—— (23)
WMI技术介绍和应用—— (20)
实现HTTP协议Get、Post (20)
如何定制一款12306抢票 (17)
在windows程序中嵌入Lu (15)
一个分析"文件夹"选择框 (13)
反汇编算法介绍和应用— (12)
使用VC内嵌Python实现自 (10)

推荐文章

* CSDN日报20170817——《如果不从事编程，我可以做什么？》

```
21. [  FAILED  ] ListTest.ThirdElement
22.
23.  1 FAILED TEST
```

在《Google Test(GTest)使用方法和源码解析——自动调度机制分析》一文中，我们分析了，测试用例对象指针将保存在类UnitTestImpl中

```
[cpp]

01. // The vector of TestCases in their original order.  It owns the
02. // elements in the vector.
03.   std::vector<TestCase*> test_cases_;
```

那么结果的统计，肯定也是针对这个vector变量的。实际也是如此，我们在代码中找到如下函数，从函数注释，我们就可以知道其对应于上面输出中那个结果的统计

```
[cpp]

01. // Gets the number of successful test cases.
02. int UnitTestImpl::successful_test_case_count() const {
03.   return CountIf(test_cases_, TestCasePassed);
04. }
05.
06. // Gets the number of failed test cases.
07. int UnitTestImpl::failed_test_case_count() const {
08.   return CountIf(test_cases_, TestCaseFailed);
09. }
10.
11. // Gets the number of all test cases.
12. int UnitTestImpl::total_test_case_count() const {
13.   return static_cast<int>(test_cases_.size());
14. }
15.
16. // Gets the number of all test cases that contain at least one test
17. // that should run.
18. int UnitTestImpl::test_case_to_run_count() const {
19.   return CountIf(test_cases_, ShouldRunTestCase);
20. }
21.
22. // Gets the number of successful tests.
23. int UnitTestImpl::successful_test_count() const {
24.   return SumOverTestCaseList(test_cases_, &TestCase::successful_test_count);
25. }
26.
27. // Gets the number of failed tests.
28. int UnitTestImpl::failed_test_count() const {
29.   return SumOverTestCaseList(test_cases_, &TestCase::failed_test_count);
30. }
31.
32. // Gets the number of disabled tests that will be reported in the XML report.
33. int UnitTestImpl::reportable_disabled_test_count() const {
34.   return SumOverTestCaseList(test_cases_,
35.                              &TestCase::reportable_disabled_test_count);
36. }
37.
38. // Gets the number of disabled tests.
39. int UnitTestImpl::disabled_test_count() const {
40.   return SumOverTestCaseList(test_cases_, &TestCase::disabled_test_count);
41. }
42.
43. // Gets the number of tests to be printed in the XML report.
44. int UnitTestImpl::reportable_test_count() const {
45.   return SumOverTestCaseList(test_cases_, &TestCase::reportable_test_count);
46. }
47.
48. // Gets the number of all tests.
49. int UnitTestImpl::total_test_count() const {
50.   return SumOverTestCaseList(test_cases_, &TestCase::total_test_count);
51. }
52.
53. // Gets the number of tests that should run.
54. int UnitTestImpl::test_to_run_count() const {
55.   return SumOverTestCaseList(test_cases_, &TestCase::test_to_run_count);
56. }
```

CountIf函数返回符合条件的测试用例个数，SumOverTestCaseList函数返回符合条件的所有测试特例的个数。其实现也非常简单，我们以CountIf为例

```cpp
template <class Container, typename Predicate>
inline int CountIf(const Container& c, Predicate predicate) {
  // Implemented as an explicit loop since std::count_if() in libCstd on
  // Solaris has a non-standard signature.
  int count = 0;
  for (typename Container::const_iterator it = c.begin(); it != c.end(); ++it) {
    if (predicate(*it))
      ++count;
  }
  return count;
}
```

这种写法的一个好处就是我们封装了函数调用迭代器中元素，从而不用到处都是遍历。然后我们将重点放在传入的函数指针，以TestCaseFailed为例

```cpp
// Returns true iff the test case failed.
static bool TestCaseFailed(const TestCase* test_case) {
  return test_case->should_run() && test_case->Failed();
}
```

它和TestCasePassed区别就是将test_case调用的Failed函数变成Passed函数。而TestCaseF是对Failed函数取反，所以最终还是调用到Failed中，我们看下其实现

```html
bool Failed() const { return failed_test_count() > 0; }
int TestCase::failed_test_count() const {
  return CountIf(test_info_list_, TestFailed);
}
```

可见TestCase测试用例对象最终还是要对其下的测试特例对象指针逐个调用TestFailed

```cpp
// Returns true iff test failed.
static bool TestFailed(const TestInfo* test_info) {
  return test_info->should_run() && test_info->result()->Failed();
}
```

经过这层传递，最终逻辑运行到TestResult的Failed函数中

```cpp
bool TestResult::Failed() const {
  for (int i = 0; i < total_part_count(); ++i) {
    if (GetTestPartResult(i).failed())
      return true;
  }
  return false;
}
```

GetTestPartResult获取的一个测试特例中"局部测试"的结果。比如

```cpp
TEST(IsPrimeTest, Negative) {
  // This test belongs to the IsPrimeTest test case.
  EXPECT_FALSE(IsPrime(-1));
  EXPECT_FALSE(IsPrime(-2));
  EXPECT_FALSE(IsPrime(INT_MIN));
}
```

这个测试特例中有三个"局部测试"（3、4和5行）。它们的结果保存在TestResult的（实际上并不是所有情况都保存，我们将在之后分析）

```cpp
// The vector of TestPartResults
  std::vector<TestPartResult> test_part_results_;
```

现在我们看到了数据的统计逻辑，接下来我们需要关注源码是如何将结果填充到test_part_results_中的。

在源码中，TestResult只提供了AddTestPartResult方法用于保存"局部测试"结果。而调用该方法的地方只有一

处

```cpp
void DefaultGlobalTestPartResultReporter::ReportTestPartResult(
    const TestPartResult& result) {
  unit_test_->current_test_result()->AddTestPartResult(result);
  unit_test_->listeners()->repeater()->OnTestPartResult(result);
}
```

其调用逻辑最终会归于如下逻辑

```cpp
void AssertHelper::operator=(const Message& message) const {
  UnitTest::GetInstance()->
    AddTestPartResult(data_->type, data_->file, data_->line,
                      AppendUserMessage(data_->message, message),
                      UnitTest::GetInstance()->impl()
                      ->CurrentOsStackTraceExceptTop(1)
                      // Skips the stack frame for this function itself.
                      );  // NOLINT
}
```

到此，我们只要关注于AssertHelper的赋值符就行了。但是事情并不像我们想象的那么简单，甚在这儿实现有个缺陷。为什么这么说呢？我们搜索完代码，发现该类的赋值符调用只有一处

```cpp
#define GTEST_MESSAGE_AT_(file, line, message, result_type) \
  ::testing::internal::AssertHelper(result_type, file, line, message) \
    = ::testing::Message()
```

调用GTEST_MESSAGE_AT_的地方只有

```cpp
// Generates a nonfatal failure at the given source file location with
// a generic message.
#define ADD_FAILURE_AT(file, line) \
  GTEST_MESSAGE_AT_(file, line, "Failed", \
                    ::testing::TestPartResult::kNonFatalFailure)
```

和

```cpp
#define GTEST_MESSAGE_(message, result_type) \
  GTEST_MESSAGE_AT_(__FILE__, __LINE__, message, result_type)
```

对ADD_FAILURE_AT的调用只有一处，且只是在出错时。而对GTEST_MESSAGE_的调用则有三处

```cpp
#define GTEST_FATAL_FAILURE_(message) \
  return GTEST_MESSAGE_(message, ::testing::TestPartResult::kFatalFailure)

#define GTEST_NONFATAL_FAILURE_(message) \
  GTEST_MESSAGE_(message, ::testing::TestPartResult::kNonFatalFailure)

#define GTEST_SUCCESS_(message) \
  GTEST_MESSAGE_(message, ::testing::TestPartResult::kSuccess)
```

GTEST_FATAL_FAILURE_和GTEST_NONFATAL_FAILURE_都将在出错时被调用，如EXPECT_EQ在内部是这么调用的

```cpp
#define EXPECT_PRED_FORMAT2(pred_format, v1, v2) \
  GTEST_PRED_FORMAT2_(pred_format, v1, v2, GTEST_NONFATAL_FAILURE_)
```

但是对GTEST_SUCCESS_的调用只有一处

```cpp
// Generates a success with a generic message.
#define GTEST_SUCCEED() GTEST_SUCCESS_("Succeeded")
```

GTEST_SUCCEED并不会出现在每个判断的宏中。比如EXPECT_EQ的实现是

```cpp
#define EXPECT_EQ(val1, val2) \
  EXPECT_PRED_FORMAT2(::testing::internal:: \
                      EqHelper<GTEST_IS_NULL_LITERAL_(val1)>::Compare, \
                      val1, val2)
```

EXPECT_PRED_FORMAT2宏中只处理了出错的情况——调用GTEST_NONFATAL_FAILURE_——从而触发AssertHelper的赋值符——将结果保存到“局部测试”结果集合中。而正确的情况下并不会保存结果到“局部测试”结果集中!!但是TestResult计算局部测试个数的函数注释说明它包含了所有情况的结果

```cpp
// Gets the number of all test parts.  This is the sum of the number
// of successful test parts and the number of failed test parts.
int TestResult::total_part_count() const {
  return static_cast<int>(test_part_results_.size());
}
```

所以，它的注释是错误的!!只有出错的情况会保存“局部测试”错误结果，或者人为调用GTEST_
“局部测试”正确结果，而其他情况不保存。我一直觉得test_part_results_保存的数据有点混乱，没有义。

但是这种混乱的保存为什么不会影响测试结果统计呢？我们再看下TestResult的Failed函数

```cpp
bool TestResult::Failed() const {
  for (int i = 0; i < total_part_count(); ++i) {
    if (GetTestPartResult(i).failed())
      return true;
  }
  return false;
}
```

当我们没有人为调用GTEST_SUCCEED保存“局部测试”正确结果时，test_part_results_只保存了错误结果。如果没有错误结果，total_part_count函数返回0。而从Failed函数返回false，即没有出错。

到此，我们将结果统计的实现讲完了。

上一篇 Google Test(GTest)使用方法和源码解析——自动调度机制分析
下一篇 Google Test(GTest)使用方法和源码解析——Listener技术分析和应用

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