个人资料

breaksoftware

访问：673857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——自动调度机制分析

2016-04-07 23:53 1830人阅读 评论(4) 收藏 举报

分类： GTest使用方法和源码解析（10）

目录(?) [+]

在《Google Test(GTest)使用方法和源码解析——概况》一文中，我们简单介绍了下GTest的使用。本篇博文开始，我们将深入代码，研究这些特性的实现。(转载请指明出于breaksoftware的csdn博客)

测试用例的自动保存

当使用一组宏构成测试代码后，我们并没有发现调用它们的地方。GTest框架实际上是通过这些宏将测试用例保存到类中，然后逐个去执行的。我们先查看TEST宏的实现

```cpp
#define GTEST_TEST(test_case_name, test_name)\
  GTEST_TEST_(test_case_name, test_name, \
              ::testing::Test, ::testing::internal::GetTestTypeId())

// Define this macro to 1 to omit the definition of TEST(), which
// is a generic name and clashes with some other libraries.
#if !GTEST_DONT_DEFINE_TEST
# define TEST(test_case_name, test_name) GTEST_TEST(test_case_name, test_name)
#endif
```

可见它只是对GTEST_TEST_宏的再次封装。GTEST_TEST_宏不仅要求传入测试用例和测试实例名，还要传入Test类名和其ID。我们将GTEST_TEST_的实现拆成三段分析

```cpp
// Helper macro for defining tests.
#define GTEST_TEST_(test_case_name, test_name, parent_class, parent_id)\
class GTEST_TEST_CLASS_NAME_(test_case_name, test_name) : public parent_class {\
 public:\
  GTEST_TEST_CLASS_NAME_(test_case_name, test_name)() {}\
 private:\
  virtual void TestBody();\
  static ::testing::TestInfo* const test_info_ GTEST_ATTRIBUTE_UNUSED_;\
  GTEST_DISALLOW_COPY_AND_ASSIGN_(\
      GTEST_TEST_CLASS_NAME_(test_case_name, test_name));\
};\
\
```

首先使用宏GTEST_TEST_CLASS_NAME_生成类名。该类暴露了一个空的默认构造函数、一个私有的虚函数TestBody、一个静态变量test_info_和一个私有的赋值运算符(将运算符=私有化，限制类对象的赋值和拷贝行为)。

静态变量test_info的作用非常有意思，它利用"静态变量在程序运行前被初始化"的特性，抢在main函数执行之前，执行一段代码，从而有机会将测试用例放置于一个固定的位置。这个是"自动"保存测试用例的本质所在。

```cpp
::testing::TestInfo* const GTEST_TEST_CLASS_NAME_(test_case_name, test_name)\
  ::test_info_ =\
    ::testing::internal::MakeAndRegisterTestInfo(\
        #test_case_name, #test_name, NULL, NULL, \
        ::testing::internal::CodeLocation(__FILE__, __LINE__), \
        (parent_id), \
        parent_class::SetUpTestCase, \
        parent_class::TearDownTestCase, \
        new ::testing::internal::TestFactoryImpl<\
            GTEST_TEST_CLASS_NAME_(test_case_name, test_name)>);\
```

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 (35547)
WMI技术介绍和应用—— (18337)
如何定制一款12306抢票 (13982)
一种准标准CSV格式的介 (12471)
一种精确从文本中提取UI (12192)
实现HTTP协议Get、Post (11969)
分析两种Dump(崩溃日志 (11565)
一种解决运行程序报"应用 (11166)
实现HTTP协议Get、Post (11128)
反汇编算法介绍和应用— (10673)

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 (10)

推荐文章

* CSDN日报20170817——《如果不从事编程，我可以做什么？》

我们先跳过这段代码，看完GTEST_TEST_宏的实现，其最后一行是

```cpp
void GTEST_TEST_CLASS_NAME_(test_case_name, test_name)::TestBody()
```

这行要在类外提供TestBody函数的实现。我们要注意下，这个只是函数的一部分，即它只是包含了函数返回类型、函数名，而真正的函数实体是在TEST宏之后的{}内的，如

```cpp
TEST(FactorialTest, Zero) {
  EXPECT_EQ(1, Factorial(0));
}
```

这段代码最后应该如下，它实际上是测试逻辑的主体。

```cpp
……
void GTEST_TEST_CLASS_NAME_(test_case_name, test_name)::TestBody() {
  EXPECT_EQ(1, Factorial(0));
}
```

可以说TEST宏的写法只是一种类函数的写法，而实际它"偷梁换柱"，实现了测试的实体。

我们再看下test_info_的初始化逻辑，它调用了::testing::internal::MakeAndRegisterTestInfo函下最后一个参数，它是一个模板类，模板是当前类名。同时从名字上看，它也是一个工厂类。该类继承TestFactoryBase，并重载了CreateTest方法——它只是new出了一个模板类对象，并返回

```cpp
template <class TestClass>
class TestFactoryImpl : public TestFactoryBase {
 public:
  virtual Test* CreateTest() { return new TestClass; }
};
```

MakeAndRegisterTestInfo函数的实现也非常简单：它new出一个TestInfo类对象，并调用UnitTestImpl单例的AddTestInfo方法，将其保存起来。

```cpp
TestInfo* MakeAndRegisterTestInfo(
    const char* test_case_name,
    const char* name,
    const char* type_param,
    const char* value_param,
    CodeLocation code_location,
    TypeId fixture_class_id,
    SetUpTestCaseFunc set_up_tc,
    TearDownTestCaseFunc tear_down_tc,
    TestFactoryBase* factory) {
  TestInfo* const test_info =
      new TestInfo(test_case_name, name, type_param, value_param,
                   code_location, fixture_class_id, factory);
  GetUnitTestImpl()->AddTestInfo(set_up_tc, tear_down_tc, test_info);
  return test_info;
}
```

AddTestInfo试图通过测试用例名等信息获取测试用例，然后调用测试用例对象去新增一个测试特例——test_info。这样我们在此就将测试用例和测试特例的关系在代码中找到了关联。

```cpp
GetTestCase(test_info->test_case_name(),
                test_info->type_param(),
                set_up_tc,
                tear_down_tc)->AddTestInfo(test_info);
```

但是如果第一次调用TEST宏，是不会有测试用例类的，那么其中新建测试用例对象，并保存到UnitTestImpl类单例对象的test_cases_中的逻辑是在GetTestCase函数实现中

```cpp
TestCase* UnitTestImpl::GetTestCase(const char* test_case_name,
```

```
02.                                     const char* type_param,
03.                                     Test::SetUpTestCaseFunc set_up_tc,
04.                                     Test::TearDownTestCaseFunc tear_down_tc) {
05.   // Can we find a TestCase with the given name?
06.   const std::vector<TestCase*>::const_iterator test_case =
07.       std::find_if(test_cases_.begin(), test_cases_.end(),
08.                    TestCaseNameIs(test_case_name));
09.
10.   if (test_case != test_cases_.end())
11.     return *test_case;
12.
13.   // No.  Let's create one.
14.   TestCase* const new_test_case =
15.       new TestCase(test_case_name, type_param, set_up_tc, tear_down_tc);
16.
17.   // Is this a death test case?
18.   if (internal::UnitTestOptions::MatchesFilter(test_case_name,
19.                                                kDeathTestCaseFilter)) {
20.     ++last_death_test_case_;
21.     test_cases_.insert(test_cases_.begin() + last_death_test_case_,
22.                        new_test_case);
23.   } else {
24.     test_cases_.push_back(new_test_case);
25.   }
26.
27.   test_case_indices_.push_back(static_cast<int>(test_case_indices_.size()));
28.   return new_test_case;
29. }
```

正如我们所料，在没有找到测试实例对象指针的情况下，新建了一个TestCase测试用例对象，并[illegible]到了test_cases_中。如此我们就解释了，测试用例是如何被保存的了。

测试特例的保存

接着上例的分析，如下代码将测试特例信息通过TestCase类的AddTestInfo方法保存起来

```
[cpp]

01. GetTestCase(test_info->test_case_name(),
02.                test_info->type_param(),
03.                set_up_tc,
04.                tear_down_tc)->AddTestInfo(test_info);
```

其中AddTestInfo的实现如下

```
[cpp]

01. void TestCase::AddTestInfo(TestInfo * test_info) {
02.   test_info_list_.push_back(test_info);
03.   test_indices_.push_back(static_cast<int>(test_indices_.size()));
04. }
```

可见test_info_list_中保存了测试特例信息。

调度的**实现**

在之前的测试代码中，我们并没有发现main函数。但是C/C++语言要求程序必须要有程序入口，那Main函数呢？其实GTest为了让我们可以更简单的使用它，为我们编写了一个main函数，它位于src目录下gtest_main.cc文件中

```
[cpp]

01. GTEST_API_ int main(int argc, char **argv) {
02.   printf("Running main() from gtest_main.cc\n");
03.   testing::InitGoogleTest(&argc, argv);
04.   return RUN_ALL_TESTS();
05. }
```

Makefile文件编译了该文件，并将其链接到可执行文件中。这样我们的程序就有了入口。那么这个main函数又是如何将执行流程引到我们的代码中的呢？代码之前了无秘密。短短的这几行，只有04行才可能是我们的代码入口。（03行将程序入参传递给了Gtest库，从而实现了《Google Test(GTest)使用方法和源码解析——概况》中所述的"选择性测试"）。很显然，它的名字——RUN_ALL_TESTS也暴露了它的功能。我们来看下其实现

```
[cpp]

01. inline int RUN_ALL_TESTS() {
02.   return ::testing::UnitTest::GetInstance()->Run();
03. }
```

它最终调用了UnitTest类的单例（GetInstance）的Run方法。UnitTest类的单例是个很重要的对象，它在源码中各处可见，它是连接各个逻辑的重要一环。我们再看下Run方法的核心实现（去除平台差异后）

```cpp
return internal::HandleExceptionsInMethodIfSupported(
      impl(),
      &internal::UnitTestImpl::RunAllTests,
      "auxiliary test code (environments or event listeners)") ? 0 : 1;
```

impl()方法返回了一个UnitTestImpl对象指针impl_，它是在UniTes类的构造函数中生成的（HandleExceptionsInMethodIfSupported函数见《Google Test(GTest)使用方法和源码解析——概况》分析）

```cpp
UnitTest::UnitTest() {
  impl_ = new internal::UnitTestImpl(this);
}
```

UnitTestImpl类的RunAllTest方法中，核心的调度代码只有这几行

```cpp
for (int test_index = 0; test_index < total_test_case_count(); test_index++)
     GetMutableTestCase(test_index)->Run();
}
```

GetMutableTestCase方法逐个返回UnitTestImpl对象成员变量test_cases_中的元素——各个测试用例类对象指针，然后调用测试用例的Run方法。

```cpp
std::vector<TestCase*> test_cases_;
```

测试用例类TestCase的Run方法逻辑也是类似的，它将逐个获取其下的测试特例信息，并调用其Run方法

```cpp
for (int i = 0; i < total_test_count(); i++) {
  GetMutableTestInfo(i)->Run();
}
```

测试特例的Run方法其核心是

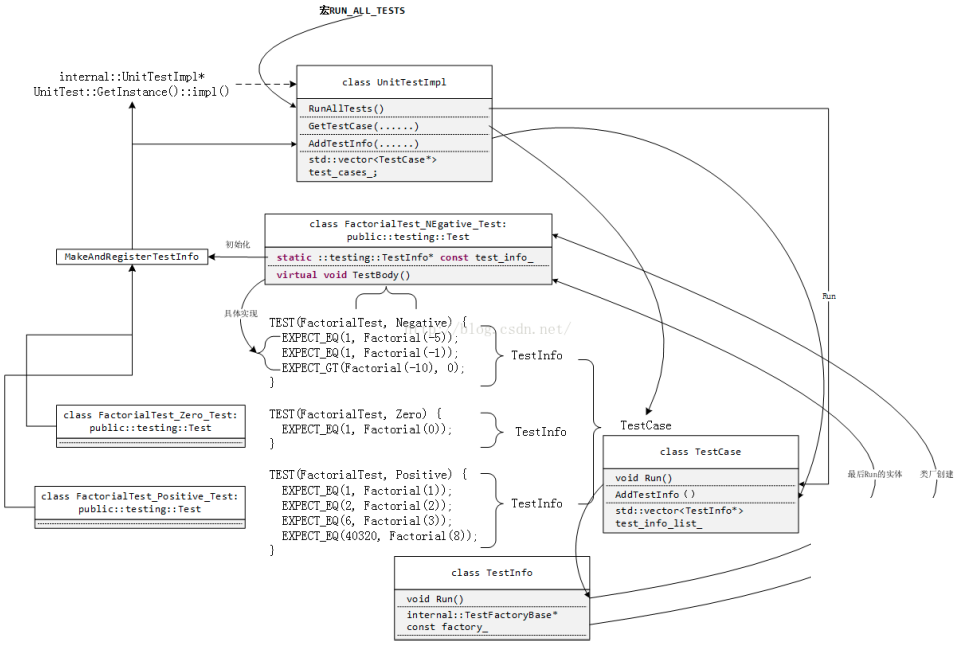
```cpp
Test* const test = internal::HandleExceptionsInMethodIfSupported(
    factory_, &internal::TestFactoryBase::CreateTest,
    "the test fixture's constructor");

if ((test != NULL) && !Test::HasFatalFailure()) {
  test->Run();
}
```

它通过构造函数传入的工厂类对象指针调用其重载的CreateTest方法，new出TEST宏中定义的使用GTEST_TEST_CLASS_NAME_命名（用例名_实例名_TEST）的类（之后称测试用例特例类）的对象指针，然后调用测试用例特例类的父类中的Run方法。由于测试用例特例类继承::testing::Test类后，并没有重载其Run方法，所以其调用的还是Test类的Run方法，而Test类的Run方法实际上只是调用了测试用例特例类重载了的TestBody方法

```cpp
internal::HandleExceptionsInMethodIfSupported(this, &Test::TestBody, "the test body");
```

而TestBody就是我们之前在分析TEST宏时讲解通过“偷梁换柱”实现的虚方法。

如此整个调度的流程就分析清楚了。



顶 踩

0 0

上一篇 Google Test(GTest)使用方法和源码解析——概况

下一篇 Google Test(GTest)使用方法和源码解析——结果统计机制分析

相关文章推荐

- VS2010中使用gtest简单案例
- 【直播】机器学习之凸优化--马博士
- gtest简介及简单使用
- 【直播】计算机视觉原理及实战--屈教授
- gtest框架的介绍与应用
- 机器学习&数据挖掘7周实训--韦玮
- gtest Test_F 和Test 区别
- 机器学习之数学基础系列--AI100
- Google Test(GTest)使用方法和源码解析——概况
- 【套餐】2017软考系统集成项目管理工程师顺利通...
- gtest和gmock入门
- 【课程】深入探究Linux/VxWorks的设备树--宋宝华
- gtest实战练习
- google gtest 快速入门
- Google C++单元测试框架(Gtest)系列教程之五—...
- 玩转Google开源C++单元测试框架Google Test系...

查看评论

2楼 mtcapple 2017-02-09 13:05发表

楼主大牛，把握剖析代码很到位。

Re: breaksoftware 2017-02-09 15:31发表

回复mtcapple：多谢，多多交流。

1楼 demiaowu 2016-11-30 12:12发表

您好，文章写得很棒，想问下您，最后一幅图使用的是什么工具画？挺清晰和漂亮的，谢谢~

Re: breaksoftware 2016-12-01 12:57发表

回复demiaowu:谢谢。使用Visio,纯手工绘制☺

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