

## CoderZh的技术博客

一个程序员的思考与总结(请移步至：<http://blog.coderzh.com/>)

[博客园](#)[首页](#)[联系](#)[订阅](#)[管理](#)

## 公告

DigitalOcean优惠码

这里的博客将不再更新，最新博客

请移步至：

我的独立博客：

<http://blog.coderzh.com/>



微信公众号：

hacker-thinking

昵称：CoderZh

园龄：11年1个月

粉丝：790

关注：10

+加关注

## 搜索

 找找看

## 随笔分类

Agile(2)

Android(3)

ASP.NET(3)

C#(20)

C/C++(24)

Cocos2d-x(1)

Emacs(2)

随笔-234 文章-10 评论-2047

## 玩转Google开源C++单元测试框架Google Test系列(gtest)之七 - 深入解析gtest

## 一、前言

“深入解析”对我来说的确有些难度，所以我尽量将我学习到和观察到的gtest内部实现介绍给大家。本文算是抛砖引玉吧，只能是对gtest的整体结构的一些介绍，想要了解更多细节最好的办法还是看gtest源码，如果你看过gtest源码，你会发现里面的注释非常的详细！好了，下面就开始了解gtest吧。

## 二、从TEST宏开始


前面的文章已经介绍过TEST宏的用法了，通过TEST宏，我们可以非法简单、方便的编写测试案例，比如：

```
TEST(FooTest, Demo)
{
    EXPECT_EQ(1, 1);
}
```

我们先不去看TEST宏的定义，而是先使用/P参数将TEST展开。如果使用的是Visual Studio的话：

1. 选中需要展开的代码文件，右键 - 属性 - C/C++ - Preprocessor
2. Generate Preprocessed File 设置 Without Line Numbers (/EP /P) 或 With Line Numbers (/P)
3. 关闭属性对话框，右键选中需要展开的文件，右键菜单中点击：Compile

编译过后，会在源代码目录生成一个后缀为.i的文件，比如我对上面的代码进行展开，展开后的内容为：

```

class FooTest_Demo_Test : public ::testing::Test
{
public:
    FooTest_Demo_Test() {}
private:
```

[Google App Engine\(7\)](#)[JAVA\(3\)](#)[Linux\(1\)](#)[Lua\(2\)](#)[Python\(66\)](#)[Ubuntu\(9\)](#)[VBS\(4\)](#)[安全性测试\(9\)](#)[测试生活感悟\(7\)](#)[程序人生\(15\)](#)[代码安全\(3\)](#)[单元测试\(19\)](#)[公告\(13\)](#)[每周总结\(4\)](#)[软件测试\(30\)](#)[设计模式](#)[性能测试\(7\)](#)[学习笔记\(27\)](#)

## 随笔档案

[2015年9月 \(1\)](#)[2015年8月 \(2\)](#)[2015年6月 \(4\)](#)[2015年5月 \(2\)](#)[2015年4月 \(5\)](#)[2015年3月 \(1\)](#)[2014年5月 \(2\)](#)[2014年4月 \(2\)](#)[2011年5月 \(1\)](#)[2011年3月 \(1\)](#)[2011年1月 \(1\)](#)[2010年12月 \(3\)](#)[2010年11月 \(3\)](#)[2010年10月 \(2\)](#)

```
virtual void TestBody();
static ::testing::TestInfo* const test_info_;
FooTest_Demo_Test(const FooTest_Demo_Test &);
void operator=(const FooTest_Demo_Test &);
};

::testing::TestInfo* const FooTest_Demo_Test
::test_info_ =
::testing::internal::MakeAndRegisterTestInfo(
    "FooTest", "Demo", "", "",
    (::testing::internal::GetTestTypeId()),
    ::testing::Test::SetUpTestCase,
    ::testing::Test::TearDownTestCase,
    new ::testing::internal::TestFactoryImpl< FooTest_Demo_Test>);

void FooTest_Demo_Test::TestBody()
{
    switch (0)
    case 0:
        if (const ::testing::AssertionResult
            gtest_ar =
                (::testing::internal:: EqHelper<(sizeof(::testing::internal::IsNullLiteralHelper(1)) == 1)>::Compare("1", "1", 1, 1)))
            ;
        else
            ::testing::internal::AssertHelper(
                ::testing::TPRT_NONFATAL_FAILURE,
                ".\\gtest_demo.cpp",
                9,
                gtest_ar.failure_message()
            ) = ::testing::Message();
}
```



展开后，我们观察到：

1. TEST宏展开后，是一个继承自testing::Test的类。

2010年9月 (6)  
 2010年8月 (2)  
 2010年7月 (4)  
 2010年6月 (3)  
 2010年5月 (4)  
 2010年4月 (9)  
 2010年3月 (6)  
 2010年2月 (3)  
 2010年1月 (16)  
 2009年12月 (6)  
 2009年11月 (3)  
 2009年10月 (4)  
 2009年9月 (3)  
 2009年8月 (2)  
 2009年7月 (7)  
 2009年6月 (2)  
 2009年4月 (12)  
 2009年3月 (5)  
 2009年2月 (2)  
 2009年1月 (3)  
 2008年12月 (7)  
 2008年11月 (9)  
 2008年9月 (8)  
 2008年8月 (7)  
 2008年7月 (8)  
 2008年6月 (9)  
 2008年5月 (33)  
 2008年4月 (6)  
 2008年2月 (1)  
 2007年12月 (3)  
 2007年11月 (3)  
 2007年10月 (7)  
 2007年9月 (1)

- 我们在TEST宏里面写的测试代码，其实是被放到了类的TestBody方法中。
  - 通过静态变量test\_info\_，调用MakeAndRegisterTestInfo对测试案例进行注册。
- 如下图：

```

switch (0)
case 0:
    if (const ::testing::AssertionResult
        gtest_ar =
            (::testing::internal::EqHelper(sizeof(::testing::internal::IsNullLiteralHelper)) == 1) ? ::Compare("1", "1", 1, 1))
        ;
    else
        ::testing::internal::AssertionHelper(
            ::testing::internal::AssertionHelper(
                "gtest_demo.cpp",
                9,
                gtest_ar.failure_message()
            ) = ::testing::Message();

```

```

TEST(FooTest, Demo)
{
    EXPECT_EQ(1, 1);
}

```

```

class FooTest_Demo_Test : public ::testing::Test
{
public:
    FooTest_Demo_Test() {}
private:
    virtual void TestBody();
    static ::testing::TestInfo* const test_info_;
    FooTest_Demo_Test(const FooTest_Demo_Test &);
    void operator=(const FooTest_Demo_Test &);
};

::testing::TestInfo* const FooTest_Demo_Test
    ::test_info_ =
        ::testing::internal::MakeAndRegisterTestInfo(
            "FooTest", "Demo", "", "",
            (::testing::internal::GetTestTypeId()),
            ::testing::Test::SetUpTestCase,
            ::testing::Test::TearDownTestCase,
            new ::testing::internal::TestFactoryImpl< FooTest_Demo_Test>);

void FooTest_Demo_Test::TestBody()
{
    EXPECT_EQ(1, 1);
}

```

创建并注册TestInfo

上面关键的方法就是MakeAndRegisterTestInfo了，我们跳到MakeAndRegisterTestInfo函数中：

```

// 创建一个 TestInfo 对象并注册到 Google Test;
// 返回创建的TestInfo对象
//
// 参数:
//

```

## 系列文章

Python天天美味系列  
攻击方式学习系列  
瘦客户端那些事  
玩转gtest系列

## 读书笔记

Python网络编程  
xUnit Test Patterns  
卓有成效的程序员

## 友情链接

## 积分与排名

积分 - 547334  
排名 - 200

## 最新评论

1. Re:gtest参数化测试代码示例  
博客园的链接改了，是这个地址：  
--canbeing
2. Re:玩转Google开源C++单元测试框架Google Test系列(gtest)之一 - 初识gtest  
@xiao\_1bai编译的目标就是生成lib文件，你已经成功了。现在可以在你的项目引用gtestd.lib...  
--cnbloghzc
3. Re:ViEmuVS2013-3.2.1 破解  
安装失败，提示：  
所需要的.NET Framework 没有  
--xiake007
4. Re:玩转Google开源C++单元测试框架Google Test系列(gtest)之七 - 深入解析gtest

```
// test_case_name:      测试案例的名称
// name:                测试的名称
// test_case_comment:   测试案例的注释信息
// comment:             测试的注释信息
// fixture_class_id:    test fixture类的ID
// set_up_tc:           事件函数SetUpTestCases的函数地址
// tear_down_tc:        事件函数TearDownTestCases的函数地址
// factory:             工厂对象，用于创建测试对象(Test)

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



我们看到，上面创建了一个TestInfo对象，然后通过AddTestInfo注册了这个对象。TestInfo对象到底是一个什么样的东西呢？

TestInfo对象主要用于包含如下信息：

1. 测试案例名称 ( testcase name )
2. 测试名称 ( test name )
3. 该案例是否需要执行
4. 执行案例时，用于创建Test对象的函数指针
5. 测试结果

我们还看到，TestInfo的构造函数中，非常重要的一个参数就是工厂对象，它主要负责在运行测试案例时创建出Test对象。我们看到我们上面的例子的factory为：

谢谢，作者哥哥，这么多章，最精彩这章。领教了，谢谢

--\$JackChen

5. Re:最常用的Emacs的基本操作

如果Emacs入手都算有些难度。。。那VIM怎么办？

--震灵

6. Re:PyQt4学习资料汇总

大神，我下了你的财务管理系统，那个数据库文件提示打不开，怎么解决啊~~急用~~

--lzgst

7. Re:ViEmuVS2013-3.2.1 破解

为啥我的注册表中没有whole

tomato

--蓝域小兵

8. Re:玩转Google开源C++单元测试框架Google Test系列(gtest)之三 - 事件机制

由于没有加TEST 宏，输出结果如下：[=====] Running 0 tests from 0 test cases.[=====] 0 tests from 0 test c.....

--喜马拉雅

9. Re:从CEGUI源码看代码规范

好一个singleton！

--chaosink

10. Re:使用UI Automation库用于UI自动化测试

mark

--大恒爸爸

阅读排行榜

```
new ::testing::internal::TestFactoryImpl< FooTest_Demo_Test>
```

我们明白了，Test对象原来就是TEST宏展开后的那个类的对象(FooTest\_Demo\_Test)，再看看TestFactoryImpl的实现：

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

这个对象工厂够简单吧，嗯，Simple is better。当我们需要创建一个测试对象(Test)时，调用factory的CreateTest()方法就可以了。

创建了TestInfo对象后，再通过下面的方法对TestInfo对象进行注册：

```
GetUnitTestImpl()->AddTestInfo(set_up_tc, tear_down_tc, test_info);
```

GetUnitTestImpl()是获取UnitTestImpl对象：

```
inline UnitTestImpl* GetUnitTestImpl() {
    return UnitTest::GetInstance()->impl();
}
```

其中UnitTest是一个单件(Singleton)，整个进程空间只有一个实例，通过UnitTest::GetInstance()获取单件的实例。上面的代码看到，UnitTestImpl对象是最终是从UnitTest对象中获取的。那么UnitTestImpl到底是一个什么样的东西呢？可以这样理解：

UnitTestImpl是一个在UnitTest内部使用的，为执行单元测试案例而提供了一系列实现的那么一个类。（自己归纳的，可能不准确）

我们上面的AddTestInfo就是其中的一个实现，负责注册TestInfo实例：



```
// 添加TestInfo对象到整个单元测试中
//
```

1. 玩转Google开源C++单元测试框架Google Test系列(gtest)(总)(221480)
2. 玩转Google开源C++单元测试框架Google Test系列(gtest)之一 - 初识gtest(146539)
3. 玩转Google开源C++单元测试框架Google Test系列(gtest)之二 - 断言(105010)
4. 玩转Google开源C++单元测试框架Google Test系列(gtest)之三 - 事件机制(68517)
5. 玩转Google开源C++单元测试框架Google Test系列(gtest)之六 - 运行参数(54532)
6. 玩转Google开源C++单元测试框架Google Test系列(gtest)之四 - 参数化(54400)
7. C# 中使用JSON - DataContractJsonSerializer(52402)
8. PyQt4学习资料汇总(49575)
9. 玩转Google开源C++单元测试框架Google Test系列(gtest)之七 - 深入解析gtest(49311)
10. 代码覆盖率浅谈(47554)

#### 评论排行榜

1. PyQt4学习资料汇总(152)
2. 开源Granados介绍 - SSH连接远程Linux服务器(C#)(66)
3. (原创)攻击方式学习之(1) - 跨站式脚本(Cross-Site Scripting) (49)
4. 三年之痒(44)

```
// 参数:
//
//  set_up_tc:      事件函数SetUpTestCases的函数地址
//  tear_down_tc:  事件函数TearDownTestCases的函数地址
//  test_info:      TestInfo对象
void AddTestInfo(Test::SetUpTestCaseFunc set_up_tc,
                 Test::TearDownTestCaseFunc tear_down_tc,
                 TestInfo * test_info) {
// 处理死亡测试的代码, 先不关注它
if (original_working_dir_.IsEmpty()) {
    original_working_dir_.Set(FilePath::GetCurrentDir());
    if (original_working_dir_.IsEmpty()) {
        printf("%s\n", "Failed to get the current working directory.");
        abort();
    }
}
// 获取或创建了一个TestCase对象, 并将testinfo添加到TestCase对象中。
GetTestCase(test_info->test_case_name(),
            test_info->test_case_comment(),
            set_up_tc,
            tear_down_tc)->AddTestInfo(test_info);
}
```

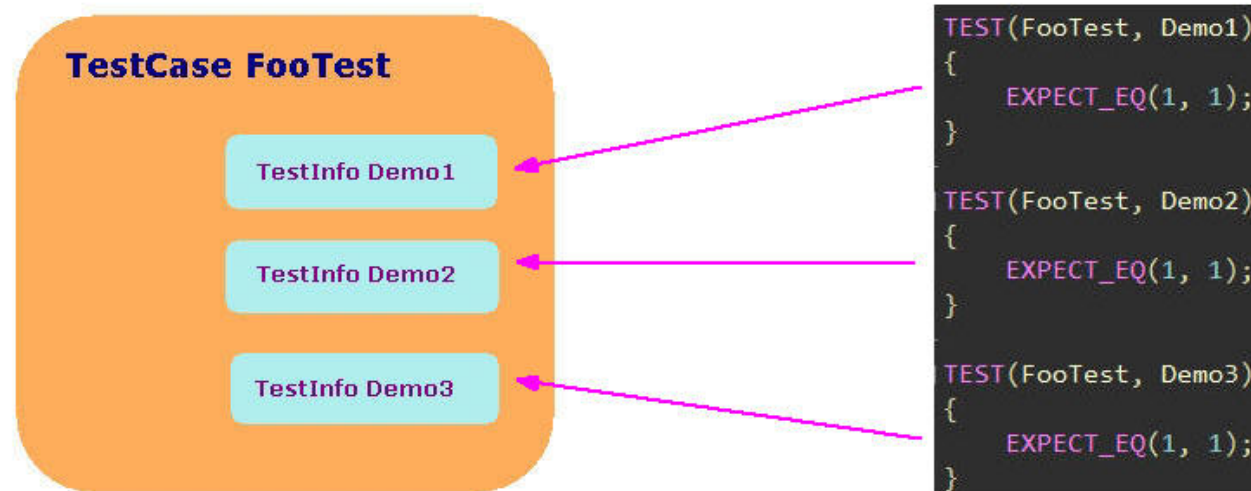
我们看到, [TestCase](#)对象出来了, 并通过AddTestInfo添加了一个TestInfo对象。这时, 似乎豁然开朗了:

1. TEST宏中的两个参数, 第一个参数testcase\_name, 就是TestCase对象的名称, 第二个参数test\_name就是Test对象的名称。而TestInfo包含了一个测试案例的一系列信息。
2. 一个TestCase对象对应一个或多个TestInfo对象。

5. NancyBlog - 我的Google App Engine Blog(42)
6. 创业三年来的一些感想 - 游戏篇(40)
7. CCNET+MSBuild+SVN实时构建的优化总结(40)
8. CoderZh首款Python联机对战游戏 - NancyTetris1.0倾情发布 (一) (37)
9. 代码安全系列(1) - Log的注入(35)
10. 程序员的信仰(35)

### 推荐排行榜

1. 玩转Google开源C++单元测试框架Google Test系列(gtest)(总)(24)
2. 创业三年来的一些感想 - 游戏篇(14)
3. 程序员的共鸣 - 读《卓有成效的程序员》(12)
4. 代码覆盖率浅谈(12)
5. 《xUnit Test Patterns》学习笔记 5 - xUnit基础(10)
6. 三年之痒(9)
7. 玩转Google开源C++单元测试框架Google Test系列(gtest)之一 - 初识gtest(9)
8. 优美的测试代码 - 行为驱动开发(BDD)(8)
9. Python天天美味(总)(7)
10. PyQt4学习资料汇总(6)



我们来看看TestCase的创建过程(UnitTestImpl::GetTestCase)：



```
// 查找并返回一个指定名称的TestCase对象。如果对象不存在，则创建一个并返回
//
// 参数：
//
// test_case_name: 测试案例名称
// set_up_tc:      事件函数SetUpTestCases的函数地址
// tear_down_tc:   事件函数TearDownTestCases的函数地址
TestCase* UnitTestImpl::GetTestCase(const char* test_case_name,
                                     const char* comment,
                                     Test::SetUpTestCaseFunc set_up_tc,
                                     Test::TearDownTestCaseFunc tear_down_tc) {

    // 从test_cases里查找指定名称的TestCase
    internal::ListNode<TestCase*>* node = test_cases_.FindIf(
        TestCaseNameIs(test_case_name));

    if (node == NULL) {
        // 没找到，我们来创建一个
        TestCase* const test_case =
            new TestCase(test_case_name, comment, set_up_tc, tear_down_tc);
```



```
// 判断是否为死亡测试案例
if (internal::UnitOptions::MatchesFilter(String(test_case_name),
                                           kDeathTestCaseFilter)) {

    // 是的话, 将该案例插入到最后一个死亡测试案例后
    node = test_cases_.InsertAfter(last_death_test_case_, test_case);
    last_death_test_case_ = node;
} else {
    // 否则, 添加到test_cases最后。
    test_cases_.PushBack(test_case);
    node = test_cases_.Last();
}

// 返回TestCase对象
return node->element();
}
```



### 三、回过头看看TEST宏的定义

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

同时也看看TEST\_F宏

```
#define TEST_F(test_fixture, test_name)\
    GTEST_TEST_(test_fixture, test_name, test_fixture, \
                 ::testing::internal::GetTypeId<test_fixture>())
```

都是使用了GTEST\_TEST\_宏, 在看看这个宏如何定义的:






```

#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_DISALLOW_COPY_AND_ASSIGN_(\
        GTEST_TEST_CLASS_NAME_(test_case_name, test_name));\
};\
\
::testing::TestInfo* const GTEST_TEST_CLASS_NAME_(test_case_name, test_name)\
::test_info_ =\
    ::testing::internal::MakeAndRegisterTestInfo(\
        #test_case_name, #test_name, "", "", \
        (parent_id), \
        parent_class::SetUpTestCase, \
        parent_class::TearDownTestCase, \
        new ::testing::internal::TestFactoryImpl<\
            GTEST_TEST_CLASS_NAME_(test_case_name, test_name)>);\
void GTEST_TEST_CLASS_NAME_(test_case_name, test_name)::TestBody()

```



不需要多解释了，和我们上面展开看到的差不多，不过这里比较明确的看到了，我们在TEST宏里写的就是TestBody里的东西。这里再补充说明一下里面的GTEST\_DISALLOW\_COPY\_AND\_ASSIGN\_宏，我们上面的例子看出，这个宏展开后：

```

FooTest_Demo_Test(const FooTest_Demo_Test &);
void operator=(const FooTest_Demo_Test &);

```

正如这个宏的名字一样，它是用于防止对对象进行拷贝和赋值操作的。

#### 四、再来了解RUN\_ALL\_TESTS宏

我们的测试案例的运行就是通过这个宏发起的。RUN\_ALL\_TEST的定义非常简单：

```
#define RUN_ALL_TESTS()\n (::testing::UnitTest::GetInstance()->Run())
```

我们又看到了熟悉的::testing::UnitTest::GetInstance(), 看来案例的执行时从UnitTest的Run方法开始的, 我提取了一些Run中的关键代码, 如下:



```
int UnitTest::Run() {\n    __try {\n        return impl_->RunAllTests();\n    } __except(internal::UnitTestOptions::GTestShouldProcessSEH(\n        GetExceptionCode())) {\n        printf("Exception thrown with code 0x%x.\\nFAIL\\n", GetExceptionCode());\n        fflush(stdout);\n        return 1;\n    }\n    return impl_->RunAllTests();\n}
```



我们又看到了熟悉的impl ( UnitTestImpl ), 具体案例该怎么执行, 还是得靠UnitTestImpl。



```
int UnitTestImpl::RunAllTests() {\n\n    // ...\n\n    printer->OnUnitTestStart(parent_);\n\n    // 计时\n    const TimeInMillis start = GetTimeInMillis();\n\n    printer->OnGlobalSetUpStart(parent_);\n    // 执行全局的SetUp事件
```

```
environments_.ForEach(SetupEnvironment);
printer->OnGlobalSetUpEnd(parent_);

// 全局的Setup事件执行成功的话
if (!Test::HasFatalFailure()) {
    // 执行每个测试案例
    test_cases_.ForEach(TestCase::RunTestCase);
}

// 执行全局的TearDown事件
printer->OnGlobalTearDownStart(parent_);
environments_in_reverse_order_.ForEach(TearDownEnvironment);
printer->OnGlobalTearDownEnd(parent_);

elapsed_time_ = GetTimeInMillis() - start;

// 执行完成
printer->OnUnitTestEnd(parent_);

// Gets the result and clears it.
if (!Passed()) {
    failed = true;
}
ClearResult();

// 返回测试结果
return failed ? 1 : 0;
}
```



上面，我们很开心的看到了我们前面讲到的全局事件的调用。environments\_是一个Environment的链表结构（List），它的内容是我们在main中通过：

```
testing::AddGlobalTestEnvironment(new FooEnvironment);
```

添加进去的。test\_cases\_我们之前也了解过了，是一个TestCase的链表结构（List）。gtest实现了一个链表，并且提供了一个Foreach方法，迭代调用某个函数，并将里面的元素作为函数的参数：



```
template <typename F> // F is the type of the function/functor
void ForEach(F functor) const {
    for ( const ListNode<E> * node = Head();
          node != NULL;
          node = node->next() ) {
        functor(node->element());
    }
}
```



因此，我们关注一下：environments\_.ForEach(SetupEnvironment)，其实是迭代调用了SetupEnvironment函数：

```
static void SetupEnvironment(Environment* env) { env->Setup(); }
```

最终调用了我们定义的Setup()函数。

再看看test\_cases\_.ForEach(TestCase::RunTestCase)的TestCase::RunTestCase实现：

```
static void RunTestCase(TestCase * test_case) { test_case->Run(); }
```

再看TestCase的Run实现：



```
void TestCase::Run() {
    if (!should_run_) return;

    internal::UnitTestImpl* const impl = internal::GetUnitTestImpl();
    impl->set_current_test_case(this);
}
```

```

UnitTestEventListenerInterface * const result_printer =
impl->result_printer();

result_printer->OnTestCaseStart(this);
impl->os_stack_trace_getter()->UponLeavingGTest();
// 哈! SetUpTestCases事件在这里调用
set_up_tc_();

const internal::TimeInMillis start = internal::GetTimeInMillis();
// 嗯, 前面分析的一个TestCase对应多个TestInfo, 因此, 在这里迭代对TestInfo调用RunTest方法
test_info_list_->ForEach(internal::TestInfoImpl::RunTest);
elapsed_time_ = internal::GetTimeInMillis() - start;

impl->os_stack_trace_getter()->UponLeavingGTest();
// TearDownTestCases事件在这里调用
tear_down_tc_();
result_printer->OnTestCaseEnd(this);
impl->set_current_test_case(NULL);
}

```



第二种事件机制又浮出我们眼前, 非常兴奋。可以看出, SetUpTestCases和TearDownTestCaess是在一个TestCase之前和之后调用的。接着看test\_info\_list\_->ForEach(internal::TestInfoImpl::RunTest) :

```

static void RunTest(TestInfo * test_info) {
    test_info->impl()->Run();
}

```

哦? TestInfo也有一个impl? 看来我们之前漏掉了点东西, 和UnitTest很类似, TestInfo内部也有一个主管各种实现的类, 那就是TestInfoImpl, 它在TestInfo的构造函数中创建了出来(还记得前面讲的TestInfo的创建过程吗?):




```

TestInfo::TestInfo(const char* test_case_name,
                  const char* name,
                  const char* test_case_comment,
                  const char* comment,
                  internal::TypeId fixture_class_id,
                  internal::TestFactoryBase* factory) {

```

```
impl_ = new internal::TestInfoImpl(this, test_case_name, name,
                                   test_case_comment, comment,
                                   fixture_class_id, factory);
}
```



因此，案例的执行还得看TestInfoImpl的Run()方法，同样，我简化一下，只列出关键部分的代码：



```
void TestInfoImpl::Run() {
    // ...

    UnitTestEventListenerInterface* const result_printer =
        impl->result_printer();
    result_printer->OnTestStart(parent_);
    // 开始计时
    const TimeInMillis start = GetTimeInMillis();

    Test* test = NULL;

    __try {
        // 我们的对象工厂，使用CreateTest()生成Test对象
        test = factory_->CreateTest();
    } __except(internal::UnitTestOptions::GTestShouldProcessSEH(
        GetExceptionCode())) {
        AddExceptionThrownFailure(GetExceptionCode(),
                                   "the test fixture's constructor");

        return;
    }

    // 如果Test对象创建成功

    if (!Test::HasFatalFailure()) {

        // 调用Test对象的Run()方法，执行测试案例

        test->Run();
    }
}
```

```
// 执行完毕，删除Test对象
impl->os_stack_trace_getter()->UponLeavingGTest();
delete test;
test = NULL;

// 停止计时
result_.set_elapsed_time(GetTimeInMillis() - start);
result_printer->OnTestEnd(parent_);

}
```



上面看到了我们前面讲到的对象工厂factory，通过factory的CreateTest()方法，创建Test对象，然后执行案例又是通过Test对象的Run()方法：



```
void Test::Run() {
    if (!HasSameFixtureClass()) return;

    internal::UnitTestImpl* const impl = internal::GetUnitTestImpl();
    impl->os_stack_trace_getter()->UponLeavingGTest();
    __try {
        // Yeah! 每个案例的SetUp事件在这里调用
        SetUp();
    } __except(internal::UnitTestOptions::GTestShouldProcessSEH(
        GetExceptionCode())) {
        AddExceptionThrownFailure(GetExceptionCode(), "SetUp()");
    }

    // We will run the test only if SetUp() had no fatal failure.
    if (!HasFatalFailure()) {
        impl->os_stack_trace_getter()->UponLeavingGTest();
        __try {
            // 哈哈!! 千辛万苦，我们定义在TEST宏里的东西终于被调用了!
            TestBody();
        } __except(internal::UnitTestOptions::GTestShouldProcessSEH(
```



```
        GetExceptionCode())) {
            AddExceptionThrownFailure(GetExceptionCode(), "the test body");
        }
    }

    impl->os_stack_trace_getter()->UponLeavingGTest();
    __try {
        // 每个案例的TearDown事件在这里调用
        TearDown();
    } __except(internal::UnitTestOptions::GTestShouldProcessSEH(
        GetExceptionCode())) {
        AddExceptionThrownFailure(GetExceptionCode(), "TearDown()");
    }
}
```



上面的代码里非常极其以及特别的兴奋的看到了执行测试案例的前后事件，测试案例执行TestBody()的代码。仿佛整个gtest的流程在眼前一目了然了。

#### 四、总结

本文通过分析TEST宏和RUN\_ALL\_TEST宏，了解到了整个gtest运作过程，可以说整个过程简洁而优美。之前读《代码之美》，感触颇深，现在读过gtest代码，再次让我感触深刻。记得很早前，我对设计的理解是“功能越强大越好，设计越复杂越好，那样才显得牛”，渐渐得，我才发现，简单才是最好。我曾总结过自己写代码的设计原则：功能明确，设计简单。了解了gtest代码后，猛然发现gtest不就是这样吗，同时gtest也给了我很多惊喜，因此，我对gtest的评价是：**功能强大，设计简单，使用方便。**

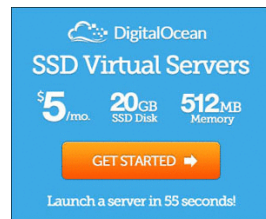
总结一下gtest里的几个关键的对象：

1. UnitTest 单例，总管整个测试，包括测试环境信息，当前执行状态等等。
2. UnitTestImpl UnitTest内部具体功能的实现者。
3. Test 我们自己编写的，或通过TEST，TEST\_F等宏展开后的Test对象，管理着测试案例的前后事件，具体的执行代码TestBody。
4. TestCase 测试案例对象，管理着基于TestCase的前后事件，管理内部多个TestInfo。
5. TestInfo 管理着测试案例的基本信息，包括Test对象的创建方法。
6. TestInfoImpl TestInfo内部具体功能的实现者。

本文还有很多gtest的细节没有分析到，比如运行参数，死亡测试，跨平台处理，断言的宏等等，希望读者自己把源码下载下来慢慢研究。如本文有错误之处，也请大家指出，谢谢！

系列链接：

- 1.玩转Google开源C++单元测试框架Google Test系列(gtest)之一 - 初识gtest
- 2.玩转Google开源C++单元测试框架Google Test系列(gtest)之二 - 断言
- 3.玩转Google开源C++单元测试框架Google Test系列(gtest)之三 - 事件机制
- 4.玩转Google开源C++单元测试框架Google Test系列(gtest)之四 - 参数化
- 5.玩转Google开源C++单元测试框架Google Test系列(gtest)之五 - 死亡测试
- 6.玩转Google开源C++单元测试框架Google Test系列(gtest)之六 - 运行参数
- 7.玩转Google开源C++单元测试框架Google Test系列(gtest)之七 - 深入解析gtest
- 8.玩转Google开源C++单元测试框架Google Test系列(gtest)之八 - 打造自己的单元测试框架



DigitalOcean的VPS主机，稳定、速度快、价格也实惠。可以在上面部署独立网站或各种实用工具。我用了很久了，确实不错，极力推荐。使用这个链接购买可获得10美元优惠。

优惠链接：[DigitalOcean优惠码](#)



微信扫一扫交流

作者：[CoderZh](#)

公众号：hacker-thinking（一个程序员的思考）

独立博客：<http://blog.coderzh.com>

博客园博客将不再更新，请关注我的「微信公众号」或「独立博客」。

作为一个程序员，思考程序的每一行代码，思考生活的每一个细节，思考人生的每一种可能。

文章版权归本人所有，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。

分类: [单元测试, C/C++](#)标签: [Google Test](#)

好文要顶

关注我

收藏该文



CoderZh

关注 - 10

粉丝 - 790

[+加关注](#)

4

0

« 上一篇: [玩转Google开源C++单元测试框架Google Test系列\(gtest\)之六 - 运行参数](#)» 下一篇: [试用了Eric4, 打算在Eric4中使用Pyqt4写个GUI程序](#)

posted @ 2009-04-11 22:23 CoderZh 阅读(49311) 评论(5) 编辑 收藏

## 评论列表

#1楼 2009-04-13 16:26 xjb

分析的很深入，学习ing

支持(0) 反对(0)

#2楼 2010-08-29 22:47 fenghuang

写的很好，赞一个~

另：

UnitTestImpl是一个在UnitTest内部使用的，为执行单元测试案例而提供了一系列实现的那么一个类。（自己归纳的，可能不准确）

-----

这里这么设计是不是主要为了把接口和实现分开，方便外面mock？

#3楼 2013-10-24 17:58 星夜落尘

我对本文的评价：深入浅出，简单优美。

支持(0) 反对(0)

#4楼 2016-10-17 17:48 执迷不悟~

感谢博主分享，讲的很到位

支持(0) 反对(0)

#5楼 2017-07-08 15:22 \$JackChen

谢谢，作者哥哥，这么多章，最精彩这章。领教了，谢谢

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

【推荐】50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库

【推荐】腾讯云上实验室 1小时搭建人工智能应用

【推荐】可嵌入您系统的“在线Excel”！SpreadJS 纯前端表格控件

【推荐】阿里云“全民云计算”优惠升级



#### 最新IT新闻:

- T-Mobile美国和Sprint正商讨合并 未来数周出结果
  - iPhone芯片和屏幕皆出自他手 三星扼住苹果咽喉
  - 贾跃亭已偿还债务200亿元 乐视称将分批还债
  - iOS 11正式版发布：可以升级了！
  - 国行iPhone 8/8 Plus首发开箱！双玻璃果然漂亮
- » 更多新闻...



#### 最新知识库文章:

- Google 及其云智慧
  - 做到这一点，你也可以成为优秀的程序员
  - 写给立志做码农的大学生
  - 架构腐化之谜
  - 学会思考，而不只是编程
- » 更多知识库文章...

站长统计