个人资料

breaksoftware

访问：673868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——参数自动填充技术分析和应用

2016-04-08 00:02 4131人阅读 评论(0) 收藏 举报

分类： GTest使用方法和源码解析（10）

目录(?) [+]

在我们设计测试用例时，我们需要考虑很多场景。每个场景都可能要细致地考虑到到各个参数。希望使用函数IsPrime检测10000以内字的数字，难道我们要写一万行代码么？（转载请指明出于bre csdn博客）

```cpp
EXPECT_TRUE(IsPrime(0));
EXPECT_TRUE(IsPrime(1));
EXPECT_TRUE(IsPrime(2));
......
EXPECT_TRUE(IsPrime(9999));
```

这种写法明显是不合理的。GTest框架当然也会考虑到这点，它设计了一套自动生成上述检测的机制，让我们用很少的代码就可以解决这个问题。

参数自动填充机制应用

我们先从应用的角度讲解其使用。首先我们设计一个需要被测试的类

```cpp
class Bis {
public:
    bool Even(int n) {
        if (n % 2 == 0) {
            return true;
        }
        else {
            return false;
        }
    };

    bool Suc(bool suc) {
        return suc;
    }
};
```

该类暴露了两个返回bool类型的方法：Even用于判断是否是偶数；Suc只是返回传入的参数。

由于GTest要求提供测试的类要继承于::testing::Test，于是我们定义一个代理类，它只是继承于::testing::Test和Bis，代理Bis完成相关调用。

```cpp
class TestClass :
    public Bis,
    public ::testing::Test {
};
```

bool型入参

Suc函数的入参类型是bool，于是我们可以新建一个测试用例类，让它继承于template <typename T> class WithParamInterface模板类，并把模板指定为bool

```cpp
class CheckBisSuc :
```

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？》

```
02.     public TestClass,
03.     public ::testing::WithParamInterface<bool>
04. {
05. };
```

我们再设置一个测试特例，在特例中使用GetParam()方法获取框架指定的参数

```
[cpp]
01. TEST_P(CheckBisSuc, Test) {
02.     EXPECT_TRUE(Suc(GetParam()));
03. }
```

最后，我们使用INSTANTIATE_TEST_CASE_P宏向框架注册"定制化测试"

```
[cpp]
01. INSTANTIATE_TEST_CASE_P(TestBisBool, CheckBisSuc, Bool());
```

该宏的第一个参数是测试前缀，第二个参数是测试类名，第三个参数是参数生成规则。如此我们就相当于执行了

```
[cpp]
01. EXPECT_TRUE(Suc(true));
02. EXPECT_TRUE(Suc(false));
```

可**选择**入参

我们再看下针对Even函数的测试。我们要定义一个继承于template <typename T> class WithF 模板类的类CheckBisEven，用于指定Even的入参类型为int

```
[cpp]
01. class CheckBisEven :
02.     public TestClass,
03.     public ::testing::WithParamInterface<int>
04. {
05. };
```

然后我们建立一个针对该类的测试特例

```
[cpp]
01. TEST_P(CheckBisEven, Test) {
02.     EXPECT_TRUE(Even(GetParam()));
03. }
```

最后我们可以使用Range、Values或者ValuesIn的方式指定Even的参数值

```
[cpp]
01. INSTANTIATE_TEST_CASE_P(TestBisValuesRange, CheckBisEven, Range(0, 9, 2));
02.
03. INSTANTIATE_TEST_CASE_P(TestBisValues, CheckBisEven, Values(11, 12, 13, 14));
04.
05. int values[] = {0, 1};
06. INSTANTIATE_TEST_CASE_P(TestBisValuesIn, CheckBisEven, ValuesIn(values));
07.
08. int moreValues[] = {0,1,2,3,4,5,6,7,8,9,10};
09. vector<int> IntVecValues(moreValues, moreValues + sizeof(moreValues));
10. INSTANTIATE_TEST_CASE_P(TestBisValuesInVector, CheckBisEven, ValuesIn(IntVecValues));
```

Range的第一个参数是起始参数值，第二个值是结束参数值，第三个参数是递增值。于是Range这组测试测试的是0、2、4、6、8这些入参。如果第三个参数没有，则默认是递增1。

Values中罗列的是将被选择作为参数的值。

ValuesIn的参数是个容器或者容器的起始迭代器和结束迭代器。

参数**组**合

参数组合要求编译器支持tr/tuple，所以一些不支持tr库的编译器将无法使用该功能。

什么是参数组合？顾名思义，就是将不同参数集组合在一起衍生出多维的数据。比如（true,false）和（1,2）可以组合成（true，1）、（true，2）、（false，1）和（false，2）等四种参数组合，然后我们使用这四组数据进行测试。

我们看个例子，首先我们要定义一个待测类。需要注意的是，它继承了模板类TestWithParam，且模板参数是组合的类型::testing::tuple<bool, int>。这个类并没有继承Bis，而是让Bis成为其成员变量，在checkData函数中检测Bis的各个函数

* Android自定义EditText：你需要一款简单实用的SuperEditText(一键删除&自定义样式)

* 从JDK源码角度看Integer

* 微信小程序——智能小秘"通知之"源码分享（语义理解基于olami）

* 多线程中断机制

* 做自由职业者是怎样的体验

最新**评论**

使用WinHttp接口实现HTTP协议(
breaksoftware: @qq_34534425:你过谦了。多总结、多练习、多借鉴就好了。

使用WinHttp接口实现HTTP协议(
qq_34534425: 代码真心nb，感觉自己写的就是渣渣

朴素、Select、Poll和Epoll网络编程
zhangcunli8499: @Breaksoftware:多谢

朴素、Select、Poll和Epoll网络编程
breaksoftware: @zhangcunli8499:这篇http://blog.csdn.net/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...

```cpp
class CombineTest :
    public TestWithParam< ::testing::tuple<bool, int> > {
protected:
    bool checkData() {
        bool suc = ::testing::get<0>(GetParam());
        int n = ::testing::get<1>(GetParam());
        return bis.Suc(suc) &&  bis.Even(n);
    }
private:
    Bis bis;
};
```

然后我们定义一个（true，false）和（1，2，3，4）组合测试

```cpp
TEST_P(CombineTest, Test) {
    EXPECT_TRUE(checkData());
}

INSTANTIATE_TEST_CASE_P(TestBisValuesCombine, CombineTest, Combine(Bool(), Va
```

如何我们便可以衍生出8组测试。我们看下部分测试结果输出

```plain
[----------] 8 tests from TestBisValuesCombine/CombineTest
......
[ RUN      ] TestBisValuesCombine/CombineTest.Test/6
[       OK ] TestBisValuesCombine/CombineTest.Test/6 (0 ms)
[ RUN      ] TestBisValuesCombine/CombineTest.Test/7
../samples/sample11_unittest.cc:175: Failure
Value of: checkData()
  Actual: false
Expected: true
[  FAILED  ] TestBisValuesCombine/CombineTest.Test/7, where GetParam() = (true, 3) (1 ms)
[----------] 8 tests from TestBisValuesCombine/CombineTest (2 ms total)
```

上例中TestBisValuesCombine/CombineTest是最终的测试用例名，Test/6和Test/7是其下两个测试特例名。

我们最后把参数生成函数罗列下

Range(begin, end[, step])	Yields values {begin, begin+step, begin+step+step, ...}. The values do not include end. ste defaults to 1.
Values(v1, v2, ..., vN)	Yields values {v1, v2, ..., vN}.
ValuesIn(container) and ValuesIn(begin, end)	Yields values from a C-style array, an STL-style container, or an iterator range [begin, end container, begin, and end can be expressions whose values are determined at run time.
Bool()	Yields sequence {false, true}.
Combine(g1, g2, ..., gN)	Yields all combinations (the Cartesian product for the math savvy) of the values generated the N generators. This is only available if your system provides the <tr1/tuple> header. If y sure your system does, and Google Test disagrees, you can override it by defining GTEST_HAS_TR1_TUPLE=1. See comments in include/gtest/internal/gtest-port.h for mo information.

参数自**动**填充机制解析

该机制和之前介绍的各种技术都不同，所以我们还要从函数注册、自动调用等基础方面去解析。

注册

之前的博文中，我们都是使用TEST宏。它帮我们完成了测试类的注册和测试实体的组织（详见《Google Test(GTest)使用方法和源码解析——自动调度机制分析》）。本节我们使用的都是TEST_P宏，其实现方式和TEST宏有类似的地方

- 都定义了一个测试类
- 都声明了一个虚方法——TestBody
- 都将赋值符设置为私有
- 都在末尾定了TestBody函数体的一部分，要求用户去填充测试实体

```cpp
# define TEST_P(test_case_name, test_name) \
```

```
  class GTEST_TEST_CLASS_NAME_(test_case_name, test_name) \
      : public test_case_name { \
   public: \
    GTEST_TEST_CLASS_NAME_(test_case_name, test_name)() {} \
    virtual void TestBody(); \
   private:
    GTEST_DISALLOW_COPY_AND_ASSIGN_(\
        GTEST_TEST_CLASS_NAME_(test_case_name, test_name));   \
......
    void GTEST_TEST_CLASS_NAME_(test_case_name, test_name)::TestBody()
```

不同的地方便是TestBody方法由私有变成公有，还有就是类的注册

```
[cpp]

private: \
    static int AddToRegistry() { \
      ::testing::UnitTest::GetInstance()->parameterized_test_registry(). \
          GetTestCasePatternHolder<test_case_name>(\
              #test_case_name, \
              ::testing::internal::CodeLocation(\
                  __FILE__, __LINE__))->AddTestPattern(\
                      #test_case_name, \
                      #test_name, \
                      new ::testing::internal::TestMetaFactory< \
                          GTEST_TEST_CLASS_NAME_(\
                              test_case_name, test_name)>()); \
      return 0; \
    } \
    static int gtest_registering_dummy_ GTEST_ATTRIBUTE_UNUSED_; \
......
  }; \
  int GTEST_TEST_CLASS_NAME_(test_case_name, \
                             test_name)::gtest_registering_dummy_ = \
      GTEST_TEST_CLASS_NAME_(test_case_name, test_name)::AddToRegistry(); \
```

TEST_P宏暴露出来的静态变量gtest_registering_dummy_明显只是一个辅助，它的真正目的只是为了让其可以在main函数之前初始化，并在初始化函数中完成类的注册。而注册函数也是实现在TEST_P定义的类的内部，但是是个静态成员函数。

注册过程中，单例UnitTest调用了parameterized_test_registry方法返回一个ParameterizedTestCaseRegistry对象引用。它是参数自动填充机制类（之后称Parameterized类）的注册场所。其内部变量test_case_infos_保存了所有Parameterized类对象的指针

```
[cpp]

private:
 typedef ::std::vector<ParameterizedTestCaseInfoBase*> TestCaseInfoContainer;
 TestCaseInfoContainer test_case_infos_;
```

该类还暴露了一个非常重要的方法GetTestCasePatternHolder，它用于返回一个测试用例对象指针

```
[cpp]

template <class TestCase>
  ParameterizedTestCaseInfo<TestCase>* GetTestCasePatternHolder(
      const char* test_case_name,
      CodeLocation code_location) {
    ParameterizedTestCaseInfo<TestCase>* typed_test_info = NULL;
    for (TestCaseInfoContainer::iterator it = test_case_infos_.begin();
         it != test_case_infos_.end(); ++it) {
      if ((*it)->GetTestCaseName() == test_case_name) {
        if ((*it)->GetTestCaseTypeId() != GetTypeId<TestCase>()) {
          // Complain about incorrect usage of Google Test facilities
          // and terminate the program since we cannot guaranty correct
          // test case setup and tear-down in this case.
          ReportInvalidTestCaseType(test_case_name, code_location);
          posix::Abort();
        } else {
          // At this point we are sure that the object we found is of the same
          // type we are looking for, so we downcast it to that type
          // without further checks.
          typed_test_info = CheckedDowncastToActualType<
              ParameterizedTestCaseInfo<TestCase> >(*it);
        }
        break;
      }
    }
```

```
    if (typed_test_info == NULL) {
      typed_test_info = new ParameterizedTestCaseInfo<TestCase>(
          test_case_name, code_location);
      test_case_infos_.push_back(typed_test_info);
    }
    return typed_test_info;
  }
```

该方法是个模板方法，模板是我们通过TEST_P传入的测试用例类。它通过我们传入的测试用例名和代码所在行数等信息，创建一个或者返回一个已存在的ParameterizedTestCaseInfo<T>*类型的数据，其指向了符合以上信息的测试用例对象。这个对象内部保存了一系列测试特例类指针

```
[cpp]

typedef ::std::vector<linked_ptr<TestInfo> > TestInfoContainer;
TestInfoContainer tests_;
```

TEST_P宏通过该对象，调用AddTestPattern方法向测试用例对象新增当前测试特例对象

```
[cpp]

void AddTestPattern(const char* test_case_name,
                    const char* test_base_name,
                    TestMetaFactoryBase<ParamType>* meta_factory) {
  tests_.push_back(linked_ptr<TestInfo>(new TestInfo(test_case_name,
                                                     test_base_name,
                                                     meta_factory)));
}
```

这个过程和TEST宏的思路基本一致，不同的是它引入了很多模板。但是需要注意的是，这并不是

行队列中插入测试用例或者测试测试特例信息的地方，这只是中间临时保存的过程。

我们再看看INSTANTIATE_TEST_CASE_P的实现，它首先定义了一个返回参数生成器的函数

```
[cpp]

# define INSTANTIATE_TEST_CASE_P(prefix, test_case_name, generator, ...) \
  ::testing::internal::ParamGenerator<test_case_name::ParamType> \
      gtest_##prefix##test_case_name##_EvalGenerator_() { return generator; } \
```

这个函数非常重要，之后我们就靠它生成参数。

然后定义了一个返回参数名称的函数

```
[cpp]

::std::string gtest_##prefix##test_case_name##_EvalGenerateName_( \
      const ::testing::TestParamInfo<test_case_name::ParamType>& info) { \
    return ::testing::internal::GetParamNameGen<test_case_name::ParamType> \
        (__VA_ARGS__)(info); \
  } \
```

最后它定义了一个全局傀儡变量，在其初始化时，抢在main函数执行之前注册相关信息

```
[cpp]

int gtest_##prefix##test_case_name##_dummy_ GTEST_ATTRIBUTE_UNUSED_ = \
    ::testing::UnitTest::GetInstance()->parameterized_test_registry(). \
        GetTestCasePatternHolder<test_case_name>(\
            #test_case_name, \
            ::testing::internal::CodeLocation(\
                __FILE__, __LINE__))->AddTestCaseInstantiation(\
                    #prefix, \
                    >est_##prefix##test_case_name##_EvalGenerator_, \
                    >est_##prefix##test_case_name##_EvalGenerateName_, \
                    __FILE__, __LINE__)
```

可见它也是通过测试用例的类名获取我们之前通过TEST_P创建的测试用例类对象，然后调用AddTestCaseInstantiation方法，传入参数生成函数指针（参数生成器）和参数名生成函数指针。通过这些信息，将新建一个定制化（Instantiation）的测试对象——Instantiationinfo。AddTestCaseInstantiation将该定制化测试对象保存到template <class TestCase> class ParameterizedTestCaseInfo类里成员变量中。

```
[cpp]

// INSTANTIATE_TEST_CASE_P macro uses AddGenerator() to record information
// about a generator.
```

```
int AddTestCaseInstantiation(const string& instantiation_name,
                             GeneratorCreationFunc* func,
                             ParamNameGeneratorFunc* name_func,
                             const char* file,
                             int line) {
  instantiations_.push_back(
      InstantiationInfo(instantiation_name, func, name_func, file, line));
  return 0;  // Return value used only to run this method in namespace scope.
}
typedef ::std::vector<InstantiationInfo> InstantiationContainer;
InstantiationContainer instantiations_;
```

至此，我们把所有在main函数之前执行的操作给看完了。但是仍然没有发现GTest框架是如何将这些临时信息保存到执行队列中的，更没有看到调度的代码。

归类及再注册

最后我们在main函数的testing::InitGoogleTest(&argc, argv);中发现如下代码

```cpp
GetUnitTestImpl()->PostFlagParsingInit();
```

PostFlagParsingInit最终将会调用到

```cpp
void UnitTestImpl::RegisterParameterizedTests() {
#if GTEST_HAS_PARAM_TEST
  if (!parameterized_tests_registered_) {
    parameterized_test_registry_.RegisterTests();
    parameterized_tests_registered_ = true;
  }
#endif
}
```

parameterized_test_registry_就是之前在TEST_P和INSTANTIATE_TEST_CASE_P宏中使用到的::testing::UnitTest::GetInstance()->parameterized_test_registry()的返回值，我们看看RegisterTests()里干了什么

```cpp
void RegisterTests() {
  for (TestCaseInfoContainer::iterator it = test_case_infos_.begin();
       it != test_case_infos_.end(); ++it) {
    (*it)->RegisterTests();
  }
}
```

它遍历了所有通过TEST_P保存的测试用例对象（ParameterizedTestCaseInfo<T>），然后逐个调用其RegisterTests方法

```cpp
virtual void RegisterTests() {
   for (typename TestInfoContainer::iterator test_it = tests_.begin();
        test_it != tests_.end(); ++test_it) {
     linked_ptr<TestInfo> test_info = *test_it;
```

一开始它枚举了所有之前通过TEST_P保存的测试特例对象，之后都会对该对象进行操作

```cpp
for (typename InstantiationContainer::iterator gen_it =
        instantiations_.begin(); gen_it != instantiations_.end();
        ++gen_it) {
  const string& instantiation_name = gen_it->name;
  ParamGenerator<ParamType> generator((*gen_it->generator)());
  ParamNameGeneratorFunc* name_func = gen_it->name_func;
  const char* file = gen_it->file;
  int line = gen_it->line;

  string test_case_name;
  if ( !instantiation_name.empty() )
    test_case_name = instantiation_name + "/";
  test_case_name += test_info->test_case_base_name;

  size_t i = 0;
  std::set<std::string> test_param_names;
```

然后枚举该测试用例中所有通过INSTANTIATE_TEST_CASE_P宏保存的定制化测试对象，并准备好相关数据供之后使用。

```cpp
for (typename ParamGenerator<ParamType>::iterator param_it =
         generator.begin();
     param_it != generator.end(); ++param_it, ++i) {
  Message test_name_stream;

  std::string param_name = name_func(
      TestParamInfo<ParamType>(*param_it, i));

  GTEST_CHECK_(IsValidParamName(param_name))
      << "Parameterized test name '" << param_name
      << "' is invalid, in " << file
      << " line " << line << std::endl;

  GTEST_CHECK_(test_param_names.count(param_name) == 0)
      << "Duplicate parameterized test name '" << param_name
      << "', in " << file << " line " << line << std::endl;

  test_param_names.insert(param_name);

  test_name_stream << test_info->test_base_name << "/" << param_name;
```

这段代码遍历参数生成器，并使用参数名生成器把所有参数转换成一个string类型数据，插入到待输出的内容中

调度

RegisterTests函数最后将调用如下过程

```cpp
      MakeAndRegisterTestInfo(
          test_case_name.c_str(),
          test_name_stream.GetString().c_str(),
          NULL,  // No type parameter.
          PrintToString(*param_it).c_str(),
          code_location_,
          GetTestCaseTypeId(),
          TestCase::SetUpTestCase,
          TestCase::TearDownTestCase,
          test_info->test_meta_factory->CreateTestFactory(*param_it));
    }  // for param_it
  }  // for gen_it
}  // for test_it
 // RegisterTests
```

MakeAndRegisterTestInfo函数在之前的博文中做过分析，它将所有测试用例和测试特例保存到GTest框架的可执行队列中，从而完成调度前的所有准备工作。至于调度及MakeAndRegisterTestInfo的细节可以参见《Google Test(GTest)使用方法和源码解析——自动调度机制分析》。

说了这么多理论，我们以之前的例子为例

```cpp
TEST_P(CheckBisEven, Test) {
    EXPECT_TRUE(Even(GetParam()));
}
```

```
04.
05.  int values[] = {0, 1};
06.  INSTANTIATE_TEST_CASE_P(TestBisValuesIn, CheckBisEven, ValuesIn(values));
07.  INSTANTIATE_TEST_CASE_P(TestBisValues, CheckBisEven, Values(11, 12, 13, 14));
```

第1行将新建名为CheckBisEven的测试用例。并在该测试用例下新建并保存一个名为CheckBisEven_Test_Test的测试特例。

第6、7行将在CheckBisEven测试用例下新增两个定制化测试对象。至此main函数之前的数据保存工作完毕，但是数据保存在一个临时区域。

testing::InitGoogleTest方法遍历所有的测试用例对象。针对每个测试用例，又遍历其测试特例对象。对每个测试特例对象，再遍历这个测试用例中保存的定制化测试对象（上例中有两个定制化测试对象）。使用定制化测试对象生成参数，通过MakeAndRegisterTestInfo方法将重新组织关系的测试用例和被参数化的测试特例保存到GTest的可执行队列中。从而在之后被框架自动调度起来。

为了区分之前的测试特例，MakeAndRegisterTestInfo使用了新的测试用例和测试特例名。测试用例名的生成规则是（INSTANTIATE_TEST_CASE_P宏的第一个参数/INSTANTIATE_TEST_CASE_P的第二个参数）——上例是TestBisValuesIn/CheckBisEven和TestBisValues/CheckBisEven，测试特例名的生成规则是（TEST_P的第二个参数/当前参数值）——上例是Test/0、Test/1、Test/11、Test/12、Test/13、Test/14。于是上例就会生成两] 有2个和4个测试特例。每个参数是一个特例。这些才是框架执行的测试对象。

参数传递

通过上面分析，我们可以得知，TEST_P定义的测试类，可能分属于两个不同的测试的特例（上 用例TestBisValuesIn/CheckBisEven和TestBisValues/CheckBisEven）。于是我们在MakeAndReg 调用中看到

```
[cpp]
01.  test_info->test_meta_factory->CreateTestFactory(*param_it)
```

这行代码是通过类厂，新建了一个特例对象。

我们将重点放到类厂的实现，这将有助于我们发现参数是怎么传递的。

测试用例信息的类中保存了一系列TestInfo对象，每个TestInfo对象都有一个用于“生成携带参数的对象”的类厂——test_meta_factory

```
[cpp]
01.  template <class TestCase>
02.  class ParameterizedTestCaseInfo : public ParameterizedTestCaseInfoBase {
03.  .......
04.   typedef typename TestCase::ParamType ParamType;
05.  .......
06.   struct TestInfo {
07.      TestInfo(const char* a_test_case_base_name,
08.               const char* a_test_base_name,
09.               TestMetaFactoryBase<ParamType>* a_test_meta_factory) :
10.          ......,
11.          test_meta_factory(a_test_meta_factory) {}
12.
13.  .......
14.      const scoped_ptr<TestMetaFactoryBase<ParamType> > test_meta_factory;
15.    };
16.  ......
17.  }
```

注意下ParamType，它是我们传入的模板类的一个属性，即测试用例类的属性，但是我们好像没有定义过它。其实我们是通过继承template <typename T> class WithParamInterface类来设置该属性的

```
[cpp]
01.  template <typename T>
02.  class WithParamInterface {
03.   public:
04.    typedef T ParamType;
```

回顾下我们测试用例类的设计

```
[cpp]
01.  class CheckBisEven :
02.      public TestClass,
03.      public ::testing::WithParamInterface<int>
04.  {
```

```
};
```

可见该用例的ParamType就是我们指定的int。框架在不知道我们指定了哪个类型的情况下，选择了一个替代符实现之后逻辑的，这在模板类设计中经常见到。

我们再回到类厂的实现上来。test_meta_factory是在TEST_P宏中使用下列方法新建的

```
[cpp]
new ::testing::internal::TestMetaFactory< GTEST_TEST_CLASS_NAME_(test_case_name, test_name)>
```

TestMetaFactory的定义如下

```
[cpp]
template <class TestCase>
class TestMetaFactory
    : public TestMetaFactoryBase<typename TestCase::ParamType> {
 public:
  typedef typename TestCase::ParamType ParamType;
......
};
```

它是个模板类，继承于另一个模板类TestMetaFactoryBase，TestMetaFactoryBase的模板是T模板的ParamType属性，对应于上例就是int。TestMetaFactoryBase类是个接口类，没什么好说的。

在MakeAndRegisterTestInfo注册测试特例时，使用了该特例的类厂对象调用CreateTestFact(

```
[cpp]
virtual TestFactoryBase* CreateTestFactory(ParamType parameter) {
  return new ParameterizedTestFactory<TestCase>(parameter);
}
```

它又新建了一个模板类对象指针

```
[cpp]
template <class TestClass>
class ParameterizedTestFactory : public TestFactoryBase {
 public:
  typedef typename TestClass::ParamType ParamType;
  explicit ParameterizedTestFactory(ParamType parameter) :
      parameter_(parameter) {}
.....
 private:
  const ParamType parameter_;

  GTEST_DISALLOW_COPY_AND_ASSIGN_(ParameterizedTestFactory);
};
```

新建的类厂对象最终会保存到TestInfo中，并在测试用例执行前被调用，从而生成对应的测试特例对象。这段逻辑在《Google Test(GTest)使用方法和源码解析——自动调度机制分析》有过分析

```
[cpp]
void TestInfo::Run() {
.......
  Test* const test = internal::HandleExceptionsInMethodIfSupported(
      factory_, &internal::TestFactoryBase::CreateTest,
      "the test fixture's constructor");
.......
}
```

我们再将关注的重点放到ParameterizedTestFactory 的CreateTest方法，它先通过模板类的SetParam方法设置了参数，然后新建并返回了一个模板类对象。

```
[cpp]
virtual Test* CreateTest() {
  TestClass::SetParam(¶meter_);
  return new TestClass();
}
```

我们的测试用例类怎么有SetParam方法？其实这也是在我们继承的WithParamInterface类中实现的

```cpp
template <typename T>
class WithParamInterface {
 public:
  typedef T ParamType;
  virtual ~WithParamInterface() {}

  // The current parameter value. Is also available in the test fixture's
  // constructor. This member function is non-static, even though it only
  // references static data, to reduce the opportunity for incorrect uses
  // like writing 'WithParamInterface<bool>::GetParam()' for a test that
  // uses a fixture whose parameter type is int.
  const ParamType& GetParam() const {
    GTEST_CHECK_(parameter_ != NULL)
        << "GetParam() can only be called inside a value-parameterized test "
        << "-- did you intend to write TEST_P instead of TEST_F?";
    return *parameter_;
  }

 private:
  // Sets parameter value. The caller is responsible for making sure the valu
  // remains alive and unchanged throughout the current test.
  static void SetParam(const ParamType* parameter) {
    parameter_ = parameter;
  }

  // Static value used for accessing parameter during a test lifetime.
  static const ParamType* parameter_;

  // TestClass must be a subclass of WithParamInterface<T> and Test.
  template <class TestClass> friend class internal::ParameterizedTestFactory;
};
```

该类保存了一个静态的全局变量parameter_，它保存了参数的指针。并通过SetParam和GetParam方法设置这个全局参数。

于是参数传递的过程就很明确了：新建TestInfo前，在一个全局区域保存参数，然后通过GetParam方法获取该全局变量，从而实现参数的传递。

其实这儿还有个非常有意思的技术点，就是参数生成器的实现。由于它不是这个框架的重点，而且相关内容也不少，我就不打算在这儿分析了，大家有兴趣可以自己看看。

上一篇 Google Test(GTest)使用方法和源码解析——私有属性代码测试技术分析
下一篇 Google Test(GTest)使用方法和源码解析——模板类测试技术分析和应用

相关文章推荐

- gtest实现架构简单分析
- 如何用googletest写单元测试
- 【直播】机器学习之凸优化--马博士
- 【套餐】2017软考系统集成项目管理工程师顺利通...
- GTest测试私有函数
- 如何编译google test的例子？
- 【直播】计算机视觉原理及实战--屈教授
- 【课程】深入探究Linux/VxWorks的设备树--宋宝华
- 给定N个节点求组成二叉搜索树个数——从一道算...
- 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——模板...
- Google Test(GTest)使用方法和源码解析——概况
- Windows Visual Studio下安装和使用google test(...
- 机器学习之数学基础系列--AI100
- gtest的TEST_F与TEST的区别

查看评论

暂无评论