

方亮的专栏

目录视图

摘要视图

RSS 订阅

个人资料



breaksoftware

访问: 673871次

积分: 8696

等级:

BLDG

6

排名: 第2191名

原创: 201篇

转载: 1篇

译文: 0篇

评论: 441条

文章搜索

博客专栏

利器

IT项目研发过程中的利器

文章:4篇

阅读:442

Libev

libev源码解析

文章:6篇

阅读:676

PE

PE文件和COFF文件格式分析

文章:11篇

阅读:28979

动态链接库

DLLMain中不当操作导致死锁问题的分析

文章:9篇

阅读:36501

WMI

WMI技术介绍和应用

文章:24篇

阅读:110414

Google Test

GTest源码解析

文章:11篇

阅读:25231

文章分类

DLLMain中的做与不做 (9)

赠书 | 异步2周年,技术图书免费送 每周荐书:渗透测试,K8s、架构(评论送书) 项目管理+代码托管+文档协作,开发更流畅

Google Test(GTest)使用方法和源码解析——模板类测试技术分析和应用

2016-04-08 00:03

3969人阅读

评论(2)

收藏

举报

分类: GTest使用方法和源码解析 (10)

版权声明:本文为博主原创文章,未经博主允许不得转载。

目录(?)

[+]

写C++难免会遇到模板问题,如果要针对一个模板类进行测试,似乎之前博文中介绍的方式只一个特化类型后再进行测试。其实GTest提供了两种测试模板类的方法,本文将介绍方法的使用原理。(转载请注明出于breaksoftware的csdn博客)

应用

GTest将这两种方法叫做:Typed Tests和Type-Parameterized Tests。我觉得可能叫做简单模式通用。先不管这些名字吧,我们看看怎么使用

简单模式(Typed Tests)

首先我们要定义一个模板类。但是为了使用GTest框架,它要继承于Test类

[cpp]

```
01. template <typename T>
02. class TypeTest : public Test {
03. protected:
04.     bool CheckData() {
05.         if (typeid(T) == typeid(bool)) {
06.             return false;
07.         }
08.         else {
09.             return true;
10.         }
11.     }
12. };
```

我们只实现了一个返回bool类型的模板类型判断函数CheckData。然后我们使用下列方式定义一个类型,类型的模板参数是我们需要传递给TypeTest的模板类型

[cpp]

```
01. typedef testing::Types<int, long> IntegerTypes
```

接下来我们使用TYPED_TEST_CASE宏注册一个测试用例

[cpp]

```
01. TYPED_TEST_CASE(TypeTest, IntegerTypes);
```

最后我们使用TYPED_TEST_P定义一个测试特例

[cpp]

```
01. TYPED_TEST_P(TypeTest, Verify) {
02.     EXPECT_TRUE(CheckData());
03. };
```

如此我们可以对TypeTest<int>和TypeTest<long>进行测试。我们看下结果

[plain]

```
01. [-----] 1 test from TypeTest/0, where TypeParam = int
02. [ RUN      ] TypeTest/0.Verify
03. [       OK ] TypeTest/0.Verify (2454 ms)
04. [-----] 1 test from TypeTest/0 (2455 ms total)
```

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项目开发过程中的利器 (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“应月 (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实现的 (10)

推荐文章

* CSDN日报20170817——《如果
不从事编程,我可以做什么?》

```
05. [-----] 1 test from TypeTest/1, where TypeParam = long
06. [ RUN      ] TypeTest/1.Verify
07. [          OK ] TypeTest/1.Verify (4843 ms)
08. [-----] 1 test from TypeTest/1 (11093 ms total)
```

高级模式

如果我们还要对TypeTest使用其他类型作为模板参数进行测试,可能要这么写

```
[cpp]
01. typedef testing::Types<float, double> FloatTypes;
02. TYPED_TEST_CASE(TypeTest, FloatTypes); // compile error
```

但是编译会报错,因为TYPED_TEST_CASE中定义的变量重定义了(之前TYPED_TEST_CASE(TypeTest, IntegerTypes);中定义的)。这个时候我们就要使用高级模式

首先我们需要声明一下测试用例类

```
[cpp]
01. TYPED_TEST_CASE_P(TypeTest);
```

然后使用TYPED_TEST_P定义一个测试实体

```
[cpp]
01. TYPED_TEST_P(TypeTest, Verify) {
02.     EXPECT_TRUE(CheckData());
03. };
```

接下来使用REGISTER_TYPED_TEST_CASE_P注册测试用例

```
[cpp]
01. REGISTER_TYPED_TEST_CASE_P(TypeTest, Verify);
```

最后使用INSTANTIATE_TYPED_TEST_CASE_P宏创建每个测试特例

```
[cpp]
01. INSTANTIATE_TYPED_TEST_CASE_P(IntegerCheck, TypeTest, IntegerTypes);
02. INSTANTIATE_TYPED_TEST_CASE_P(FloatCheck, TypeTest, FloatTypes);
03. INSTANTIATE_TYPED_TEST_CASE_P(BoolCheck, TypeTest, bool);
```

上面三行测试了三组类型,其输出是

```
[plain]
01. [-----] 1 test from IntegerCheck/TypeTest/0, where TypeParam = int
02. [ RUN      ] IntegerCheck/TypeTest/0.Verify
03. [          OK ] IntegerCheck/TypeTest/0.Verify (702 ms)
04. [-----] 1 test from IntegerCheck/TypeTest/0 (703 ms total)
05.
06. [-----] 1 test from IntegerCheck/TypeTest/1, where TypeParam = long
07. [ RUN      ] IntegerCheck/TypeTest/1.Verify
08. [          OK ] IntegerCheck/TypeTest/1.Verify (400 ms)
09. [-----] 1 test from IntegerCheck/TypeTest/1 (403 ms total)
10.
11. [-----] 1 test from FloatCheck/TypeTest/0, where TypeParam = float
12. [ RUN      ] FloatCheck/TypeTest/0.Verify
13. [          OK ] FloatCheck/TypeTest/0.Verify (451 ms)
14. [-----] 1 test from FloatCheck/TypeTest/0 (453 ms total)
15.
16. [-----] 1 test from FloatCheck/TypeTest/1, where TypeParam = double
17. [ RUN      ] FloatCheck/TypeTest/1.Verify
18. [          OK ] FloatCheck/TypeTest/1.Verify (403 ms)
19. [-----] 1 test from FloatCheck/TypeTest/1 (405 ms total)
20.
21. [-----] 1 test from BoolCheck/TypeTest/0, where TypeParam = bool
22. [ RUN      ] BoolCheck/TypeTest/0.Verify
23. ..\test\gtest_unittest.cc(117): error: Value of: CheckData()
24.     Actual: false
25.     Expected: true
26. [  FAILED  ] BoolCheck/TypeTest/0.Verify, where TypeParam = bool (5440 ms)
27. [-----] 1 test from BoolCheck/TypeTest/0 (6633 ms total)
```

* 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“遥知之”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\$ 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\$ 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 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...

原理解析

简单模式

我们先从typedef testing::Types<int, long> IntegerTypes;这句开始分析。Types定义非常有意思, 我截取部分来看看

```
[cpp]
01. struct Types0 {};
02.
03. // Type lists of length 1, 2, 3, and so on.
04.
05. template <typename T1>
06. struct Types1 {
07.     typedef T1 Head;
08.     typedef Types0 Tail;
09. };
10. template <typename T1, typename T2>
11. struct Types2 {
12.     typedef T1 Head;
13.     typedef Types1<T2> Tail;
14. };
15.
16. template <typename T1, typename T2, typename T3>
17. struct Types3 {
18.     typedef T1 Head;
19.     typedef Types2<T2, T3> Tail;
20. };
21.
22. .....
23.
24. template <typename T1 = internal::None, typename T2 = internal::None,
25.     typename T3 = internal::None, typename T4 = internal::None,
26.     typename T5 = internal::None, typename T6 = internal::None,
27.     typename T7 = internal::None, typename T8 = internal::None,
28.     typename T9 = internal::None, typename T10 = internal::None,
29.     typename T11 = internal::None, typename T12 = internal::None,
30.     typename T13 = internal::None, typename T14 = internal::None,
31.     typename T15 = internal::None, typename T16 = internal::None,
32.     typename T17 = internal::None, typename T18 = internal::None,
33.     typename T19 = internal::None, typename T20 = internal::None,
34.     typename T21 = internal::None, typename T22 = internal::None,
35.     typename T23 = internal::None, typename T24 = internal::None,
36.     typename T25 = internal::None, typename T26 = internal::None,
37.     typename T27 = internal::None, typename T28 = internal::None,
38.     typename T29 = internal::None, typename T30 = internal::None,
39.     typename T31 = internal::None, typename T32 = internal::None,
40.     typename T33 = internal::None, typename T34 = internal::None,
41.     typename T35 = internal::None, typename T36 = internal::None,
42.     typename T37 = internal::None, typename T38 = internal::None,
43.     typename T39 = internal::None, typename T40 = internal::None,
44.     typename T41 = internal::None, typename T42 = internal::None,
45.     typename T43 = internal::None, typename T44 = internal::None,
46.     typename T45 = internal::None, typename T46 = internal::None,
47.     typename T47 = internal::None, typename T48 = internal::None,
48.     typename T49 = internal::None, typename T50 = internal::None>
49. struct Types {
50.     typedef internal::Types50<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
51.         T13, T14, T15, T16, T17, T18, T19, T20, T21, T22, T23, T24, T25, T26,
52.         T27, T28, T29, T30, T31, T32, T33, T34, T35, T36, T37, T38, T39, T40,
53.         T41, T42, T43, T44, T45, T46, T47, T48, T49, T50> type;
54. };
55.
56. template <>
57. struct Types<internal::None, internal::None, internal::None, internal::None,
58.     internal::None, internal::None, internal::None, internal::None,
59.     internal::None, internal::None, internal::None, internal::None,
60.     internal::None, internal::None, internal::None, internal::None,
61.     internal::None, internal::None, internal::None, internal::None,
62.     internal::None, internal::None, internal::None, internal::None,
63.     internal::None, internal::None, internal::None, internal::None,
64.     internal::None, internal::None, internal::None, internal::None,
65.     internal::None, internal::None, internal::None, internal::None,
66.     internal::None, internal::None, internal::None, internal::None,
67.     internal::None, internal::None, internal::None, internal::None,
68.     internal::None, internal::None, internal::None, internal::None,
69.     internal::None, internal::None> {
70.     typedef internal::Types0 type;
71. };
```

```

72.  template <typename T1>
73.  struct Types<T1, internal::None, internal::None, internal::None,
74.      internal::None, internal::None, internal::None, internal::None,
75.      internal::None, internal::None, internal::None, internal::None,
76.      internal::None, internal::None, internal::None, internal::None,
77.      internal::None, internal::None, internal::None, internal::None,
78.      internal::None, internal::None, internal::None, internal::None,
79.      internal::None, internal::None, internal::None, internal::None,
80.      internal::None, internal::None, internal::None, internal::None,
81.      internal::None, internal::None, internal::None, internal::None,
82.      internal::None, internal::None, internal::None, internal::None,
83.      internal::None, internal::None, internal::None, internal::None,
84.      internal::None, internal::None, internal::None, internal::None,
85.      internal::None, internal::None> {
86.      typedef internal::Types1<T1> type;
87.  };
88.  template <typename T1, typename T2>
89.  struct Types<T1, T2, internal::None, internal::None, internal::None,
90.      internal::None, internal::None, internal::None, internal::None,
91.      internal::None, internal::None, internal::None, internal::None,
92.      internal::None, internal::None, internal::None, internal::None,
93.      internal::None, internal::None, internal::None, internal::None,
94.      internal::None, internal::None, internal::None, internal::None,
95.      internal::None, internal::None, internal::None, internal::None,
96.      internal::None, internal::None, internal::None, internal::None,
97.      internal::None, internal::None, internal::None, internal::None,
98.      internal::None, internal::None, internal::None, internal::None,
99.      internal::None, internal::None, internal::None, internal::None,
100.     internal::None, internal::None, internal::None, internal::None,
101.     internal::None> {
102.     typedef internal::Types2<T1, T2> type;
103. };

```

这段代码很长,但是其定义了我们用于罗列类型的结构体Types。它是一个递归定义,即Types3{Types2, Types2依赖于Types1, Types1依赖于Types0.....。每个模板类都会将自己模板列表的第一个模板别名为Head,剩下的类型别名为Tail。未来我们将看到这两个类型的使用。

我们再看下TYPED_TEST_CASE的实现

```

[cpp]
01.  # define TYPED_TEST_CASE(CaseName, Types) \
02.      typedef ::testing::internal::TypeList< Types >::type \
03.      GTEST_TYPE_PARAMS_(CaseName)
04.  # define GTEST_TYPE_PARAMS_(TestCaseName) gtest_type_params_##TestCaseName##_

```

它只是对该测试用例的参数列表类的type做了一个别名,展开代码就是

```

[cpp]
01.  typedef ::testing::internal::TypeList< IntegerTypes >::type gtest_type_params_TypeTest_

```

TypeList是个模板类,它将Types<T>别名为type

```

[cpp]
01.  template <typename T>
02.  struct TypeList {
03.      typedef Types1<T> type;
04.  };

```

对应于我们的例子就是

```

[cpp]
01.  struct TypeList<IntegerTypes> {
02.      typedef Types1<IntegerTypes> type;
03.  };

```

于是整体展开就是

```

[cpp]
01.  typedef ::testing::internal::Types1<IntegerTypes> gtest_type_params_TypeTest_

```

最后我们看下TYPED_TEST的宏实现。它和之前博文介绍的TEST宏有如下相同之处:

- 定义了私有的虚方法TestBody
- 执行了注册逻辑
- 末尾声明了TestBody函数部分, 便于开发者填充测试实体

相同的地方我们就不说了, 我们看下不同的

```
[cpp]
01. # define TYPED_TEST(CaseName, TestName) \
02.     template <typename gtest_TypeParam> \
03.     class GTEST_TEST_CLASS_NAME_(CaseName, TestName) \
04.     : public CaseName<gtest_TypeParam> { \
05.     private: \
06.         typedef CaseName<gtest_TypeParam> TestFixture; \
07.         typedef gtest_TypeParam_ TypeParam; \
08.         virtual void TestBody(); \
09.     }; \
```

它继承于一个模板类, 模板类的类名是我们通过TYPED_TEST传入的测试用例类。同时它将父类、模板类进行了别名操作。用我们的例子展开代码即是

```
[cpp]
01. template <typename T>
02.     class TypeTest_Verify_Test
03.     : public TypeTest<T> {
04.     private:
05.         typedef TypeTest<T> TestFixture;
06.         typedef T TypeParam;
07.         virtual void TestBody();
08.     }; \
```

最后一个傀儡变量被初始化

```
[cpp]
01. bool gtest_##CaseName##_##TestName##_registered_ GTEST_ATTRIBUTE_UNUSED_ = \
02.     ::testing::internal::TypeParameterizedTest< \
03.         CaseName, \
04.         ::testing::internal::TemplateSel< \
05.             GTEST_TEST_CLASS_NAME_(CaseName, TestName)>, \
06.             GTEST_TYPE_PARAMS_(CaseName)>::Register(\
07.         "", ::testing::internal::CodeLocation(__FILE__, __LINE__), \
08.         #CaseName, #TestName, 0); \
```

我们把代码展开

```
[cpp]
01. bool gtest_TypeTest_Verify_registered_ GTEST_ATTRIBUTE_UNUSED_ =
02.     ::testing::internal::TypeParameterizedTest<TypeTest,
03.         ::testing::internal::TemplateSel<TypeTest_Verify_Test>,
04.         gtest_type_params_TypeTest_
05.     >::Register(
06.         "",
07.         ::testing::internal::CodeLocation(__FILE__, __LINE__),
08.         'TypeTest',
09.         'Verify',
10.         0);
```

我们看下TypeParameterizedTest类的Register实现

```
[cpp]
01. template <GTEST_TEMPLATE_ Fixture, class TestSel, typename Types>
02.     class TypeParameterizedTest {
03.     public:
04.         // 'index' is the index of the test in the type list 'Types'
05.         // specified in INSTANTIATE_TYPED_TEST_CASE_P(Prefix, TestCase,
06.         // Types). Valid values for 'index' are [0, N - 1] where N is the
07.         // length of Types.
08.         static bool Register(const char* prefix,
09.                             CodeLocation code_location,
10.                             const char* case_name, const char* test_names,
11.                             int index) {
12.             typedef typename Types::Head Type;
```

```

13.     typedef Fixture<Type> FixtureClass;
14.     typedef typename GTEST_BIND_(TestSel, Type) TestClass;
15.
16.     // First, registers the first type-parameterized test in the type
17.     // list.
18.     MakeAndRegisterTestInfo(
19.         (std::string(prefix) + (prefix[0] == '\0' ? "" : "/") + case_name + "/"
20.          + StreamableToString(index)).c_str(),
21.         StripTrailingSpaces(GetPrefixUntilComma(test_names)).c_str(),
22.         GetTypeName<Type>().c_str(),
23.         NULL, // No value parameter.
24.         code_location,
25.         GetTypeId<FixtureClass>(),
26.         TestClass::SetUpTestCase,
27.         TestClass::TearDownTestCase,
28.         new TestFactoryImpl<TestClass>);
29.
30.     // Next, recurses (at compile time) with the tail of the type list.
31.     return TypeParameterizedTest<Fixture, TestSel, typename Types::Tail>
32.         ::Register(prefix, code_location, case_name, test_names, index + 1);
33. }
34. };

```

这段代码非常重要, 我们看到核心函数MakeAndRegisterTestInfo, 它把我们测试的对象加入到中。具体它的原理和实现可以参看《Google Test(GTest)使用方法和源码解析——自动调度机制分

第12行别名为Types::Head为Type。Types是传入的模板类, 以我们的例子为例, 其传入的就是::testing::internal::Types1<IntegerTypes>。我们在介绍Types模板类时提到过Head别名, 它是该模板类第一个模板参数类型。对应于我们的例子就是typedef testing::Types<int, long> IntegerTypes;中的int类型。

第13行使用12行别名的类型, 特化了我们传入的测试用例类, 即该行对应于

```

[cpp]
01.     typedef TypeTest<int> FixtureClass;

```

第14行对测试特例类使用了int类型进行特化

```

[cpp]
01.     template <GTEST_TEMPLATE_ Tmpl>
02.     struct TemplateSel {
03.         template <typename T>
04.         struct Bind {
05.             typedef Tmpl<T> type;
06.         };
07.     };
08.
09.     # define GTEST_BIND_(TmplSel, T) \
10.         TmplSel::template Bind<T>::type

```

即对应于TypeTest_Verify_Test<T>::type

```

[cpp]
01.     typedef typename TypeTest_Verify_Test<T> TestClass;

```

如此MakeAndRegisterTestInfo函数中的参数就比较明确了。

这段代码中还有个非常重要的一个递归调用

```

[cpp]
01.     // Next, recurses (at compile time) with the tail of the type list.
02.     return TypeParameterizedTest<Fixture, TestSel, typename Types::Tail>
03.         ::Register(prefix, code_location, case_name, test_names, index + 1);

```

这次调用, 最后一个模板参数传递了Types::Tail。它在编译期间将触发编译器进行类型推导, 如同抽丝剥茧般, 使用typedef testing::Types<int, long> IntegerTypes;中每个模板类型对TypeParameterizedTest::Register进行特化。从而可以在运行期间可以对每个类型进行注册。

高级模式

我们先看下TYPED_TEST_CASE_P宏的实现

```

[cpp]
01.     # define TYPED_TEST_CASE_P(CaseName) \
02.         static ::testing::internal::TypedTestCasePState \

```

```
03. |         GTEST_TYPED_TEST_CASE_P_STATE_(CaseName)
```

它定义了一个TypedTestCasePState类型的全局变量,对应于我们例子就是

```
[cpp]
```

```
01. | static ::testing::internal::TypedTestCasePState gtest_typed_test_case_p_state_TypeTest_;
```

TypedTestCasePState类暴露了AddTestName方法用于保存测试用例和测试特例名 再看下
REGISTER_TYPED_TEST_CASE_P宏的实现

```
[cpp]
```

```
01. | # define REGISTER_TYPED_TEST_CASE_P(CaseName, ...) \
02. |     namespace GTEST_CASE_NAMESPACE_(CaseName) { \
03. |         typedef ::testing::internal::Templates<__VA_ARGS__>::type gtest_AllTests_; \
04. |     } \
05. |     static const char* const GTEST_REGISTERED_TEST_NAMES_(CaseName) = \
06. |         GTEST_TYPED_TEST_CASE_P_STATE_(CaseName).VerifyRegisteredTestNames(\
07. |             __FILE__, __LINE__, #__VA_ARGS__)
```

它使用TYPED_TEST_CASE_P定义的TypedTestCasePState类对象方法VerifyRegisteredT
注册的测试用例名,并别名了一个类型。以上两个宏都是和测试用例名称注册有关。接下来我们看下
TYPED_TEST_P的实现

```
[cpp]
```

```
01. | # define TYPED_TEST_P(CaseName, TestName) \
02. |     namespace GTEST_CASE_NAMESPACE_(CaseName) { \
03. |         template <typename gtest_TypeParam> \
04. |         class TestName : public CaseName<gtest_TypeParam> { \
05. |             private: \
06. |                 typedef CaseName<gtest_TypeParam> TestFixture; \
07. |                 typedef gtest_TypeParam_ TypeParam; \
08. |                 virtual void TestBody(); \
09. |         }; \
10. |         static bool gtest_##TestName##_defined_ GTEST_ATTRIBUTE_UNUSED_ = \
11. |             GTEST_TYPED_TEST_CASE_P_STATE_(CaseName).AddTestName(\
12. |                 __FILE__, __LINE__, #CaseName, #TestName); \
13. |     } \
14. |     template <typename gtest_TypeParam> \
15. |     void GTEST_CASE_NAMESPACE_(CaseName)::TestName<gtest_TypeParam>::TestBody()
```

它和我们之前介绍的TYPED_TEST实现是相似的。不同点是:

- 直接使用传入的测试特例名作为类名
- 调用TYPED_TEST_CASE_P定义的TypedTestCasePState类对象AddTestName对测试用例和测试特例名进行注册
- 将测试特例类和傀儡变量初始化过程控制在一个和测试用例名相关的命名空间中

最后我们看下INSTANTIATE_TYPED_TEST_CASE_P的实现

```
[cpp]
```

```
01. | # define INSTANTIATE_TYPED_TEST_CASE_P(Prefix, CaseName, Types) \
02. |     bool gtest_##Prefix##_##CaseName GTEST_ATTRIBUTE_UNUSED_ = \
03. |         ::testing::internal::TypeParameterizedTestCase<CaseName, \
04. |             GTEST_CASE_NAMESPACE_(CaseName)::gtest_AllTests_, \
05. |             ::testing::internal::TypeList< Types >::type>::Register(\
06. |                 #Prefix, \
07. |                 ::testing::internal::CodeLocation(__FILE__, __LINE__), \
08. |                 >EST_TYPED_TEST_CASE_P_STATE_(CaseName), \
09. |                 #CaseName, GTEST_REGISTERED_TEST_NAMES_(CaseName))
```

可以说,套路和简单模式的注册方式是一样的。不一样的是它调用了TypeParameterizedTestCase类的
Register,而不是TypeParameterizedTest的Register。还有就是Register的第二个参数是在
REGISTER_TYPED_TEST_CASE_P别名的类型。我们看下这个类型的相关代码

```
[cpp]
```

```
01. | # define REGISTER_TYPED_TEST_CASE_P(CaseName, ...) \
02. |     namespace GTEST_CASE_NAMESPACE_(CaseName) { \
03. |         typedef ::testing::internal::Templates<__VA_ARGS__>::type gtest_AllTests_; \
04. |     } \
```

```
[cpp]
01. // Template lists of length 1, 2, 3, and so on.
02.
03. template <GTEST_TEMPLATE_ T1>
04. struct Templates1 {
05.     typedef TemplateSel<T1> Head;
06.     typedef Templates0 Tail;
07. };
08. template <GTEST_TEMPLATE_ T1, GTEST_TEMPLATE_ T2>
09. struct Templates2 {
10.     typedef TemplateSel<T1> Head;
11.     typedef Templates1<T2> Tail;
12. };

[cpp]
01. template <>
02. struct Templates<NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
03.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
04.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
05.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
06.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
07.     NoneT> {
08.     typedef Templates0 type;
09. };
10. template <GTEST_TEMPLATE_ T1>
11. struct Templates<T1, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
12.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
13.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
14.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
15.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
16.     NoneT> {
17.     typedef Templates1<T1> type;
18. };
19. template <GTEST_TEMPLATE_ T1, GTEST_TEMPLATE_ T2>
20. struct Templates<T1, T2, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
21.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
22.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
23.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
24.     NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT, NoneT,
25.     NoneT> {
26.     typedef Templates2<T1, T2> type;
27. };

```

可以见得这个类型和之前的Types是类似的,用于在编译期间通过编译器推导特例出注册方法。需要注意的是这个地方推导的不是模板类的类型,而是测试特例类。我们在讲解TYPED_TEST_P时提过,宏中直接使用传入的测试特例名作为类名,这是有原因的。原因就是在这儿要一个推导。可能这么说还不太明白,我们看下高级模式另外一种用法

```
[cpp]
01. TYPED_TEST_P(TypeTest, Verify) {
02.     EXPECT_TRUE(CheckData());
03. };
04.
05. TYPED_TEST_P(TypeTest, Verify2) {
06.     EXPECT_FALSE(CheckData());
07. };
08.
09. REGISTER_TYPED_TEST_CASE_P(TypeTest, Verify, Verify2);

```

这儿的Verify和Verify2都是测试特例名,于是通过REGISTER_TYPED_TEST_CASE_P操作后,就变成

```
[cpp]
01. typedef ::testing::internal::Templates<Verify,Verfiy2>::type gtest_AllTests_;

```

最后在下面注册函数中,触发对该函数使用Verify和Verfiy2进行特化的操作。

```
[cpp]
01. template <GTEST_TEMPLATE_ Fixture, typename Tests, typename Types>
02. class TypeParameterizedTestCase {
03. public:
04.     static bool Register(const char* prefix, CodeLocation code_location,
05.         const TypedTestCasePState* state,
06.         const char* case_name, const char* test_names) {

```



```
07.         std::string test_name = StripTrailingSpaces(  
08.             GetPrefixUntilComma(test_names));  
09.         if (!state->TestExists(test_name)) {  
10.             fprintf(stderr, "Failed to get code location for test %s.%s at %s.",  
11.                 case_name, test_name.c_str(),  
12.                 FormatFileLocation(code_location.file.c_str(),  
13.                     code_location.line).c_str());  
14.             fflush(stderr);  
15.             posix::Abort();  
16.         }  
17.         const CodeLocation& test_location = state->GetCodeLocation(test_name);  
18.  
19.         typedef typename Tests::Head Head;  
20.  
21.         // First, register the first test in 'Test' for each type in 'Types'.  
22.         TypeParameterizedTest<Fixture, Head, Types>::Register(  
23.             prefix, test_location, case_name, test_names, 0);  
24.  
25.         // Next, recurses (at compile time) with the tail of the test list.  
26.         return TypeParameterizedTestCase<Fixture, typename Tests::Tail, Types>  
27.             ::Register(prefix, code_location, state,  
28.                 case_name, SkipComma(test_names));  
29.     }  
30. };
```

在TypeParameterizedTestCase类的Register方法中我们看到有对TypeParameterizedTest的两种调用方式。一个测试特例下的类型推导是在TypeParameterizedTest的Register中完成的，而不同测试特例的推导则在TypeParameterizedTestCase类的Register方法中完成的。

顶 踩
1 0

上一篇 [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](#)

下一篇 [Google Test\(GTest\)使用方法和源码解析——死亡测试技术分析和应用](#)

相关文章推荐

- [Google Test\(GTest\)使用方法和源码解析——概况](#)
- [Google Test\(GTest\)使用方法和源码解析——断言...](#)
- [【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...](#)
- [如何用googletest写单元测试](#)
- [Google Test\(GTest\)使用方法和源码解析——结果...](#)
- [【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](#)
- [Google Test\(GTest\)使用方法和源码解析——参数...](#)
- [gtest实现架构简单分析](#)
- [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——私有...](#)
- [如何编译google test的例子？](#)
- [Google Test\(GTest\)使用方法和源码解析——死亡...](#)
- [机器学习之数学基础系列--AI100](#)
- [gtest安装与使用示例](#)

查看评论

1楼 [七月不是一月](#) 2016-05-25 14:18发表



简单模式测试特例是不是应该是TYPED_TEST？

还有一个问题想请教一下，当文中的高级模式有多个类似CheckData()的函数需要测试，而且不同的CheckData()对应不同的Types是不是就没法测试了？因为注册只能注册一次，但是没办法区分不同的类似CheckData()的函数，请问是这样么？

Re: [breaksoftware](#) 2016-05-26 03:21发表




回复七月不是一月：简单模式或者高级模式都是我取的名字，实际上还是应该叫Typed Tests和Type-Parameterized Tests。不是，你看下高级模式的用例。GTest在命名时，可以让你指定至少两个信息，这多个信息在一起组成了测试的实际名称，所以还是有办法区分的。你所要测试的固定的那个函数可以只是这些命名环节中的一个因素，通过修改其他因素就可以了。

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

[公司简介](#) | [招贤纳士](#) | [广告服务](#) | [联系方式](#) | [版权声明](#) | [法律顾问](#) | [问题报告](#) | [合作伙伴](#) | [论坛反馈](#)

[网站客服](#) [杂志客服](#) [微博客服](#) webmaster@csdn.net 400-660-0108 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2017, CSDN.NET, All Rights Reserved 