# white_smile的专栏





## gtest调研手册

2014-02-17 09:37 510人阅读 评论(0)

分类： 综合（4）



### 1.下载与安装

gtest是google编写的一个c++测试框架，具有轻便、灵活、跨平台等特点。其下载地址：http://code.google.com/p/googletest/downloads/list，现在最新的版本为gtest-1.6.0.zip。

下载完成后，在终端对gtest进行解压安装：

```
unzip gtest-1.6.0.zip

cd gtest-1.6.0

g++ -I./include -I./ -c ./src/gtest-all.cc
```

个人资料

白矮星

访问： 58711次
积分： 990
等级： BLOG 3
排名： 千里之外

原创： 37篇 转载： 30篇
译文： 0篇 评论： 1条

文章搜索

关闭

ar -rv libgtest.a gtest-all.o

运行上述命令后，确认生成了库文件libgtest.a，这个库是后续测试程序需要链接的库。

## 2. 测试实例

### 2.1 配置项目环境

首先把/gtest-1.6.0/include/gtest下的头文件引入eclipse工程中，其次把

安装过程中生成的库文件引入，同时把pthread库也引入到工程中。

### 2.2 测试代码

头文件CMax.h：

```cpp
#ifndefCMAX_H_
#defineCMAX_H_
intmax(int a, int b)
{
    returna > b ? a : b;
}
#endif
```

测试文件CMax_test.cpp:

```cpp
#include"gtest/gtest.h"
#include"CMax.h"
TEST(CMAX,max)
{
    EXPECT_EQ(2,max(2,-1));
```

cxf与axis2的比较 (2203)
LDAP使用说明文档 (2081)
HighCharts导出图片和P( (1651)
WdatePicker日历控件使用 (1473)
页面关闭和刷新onbefore (1466)
CoovaChilli-1.3.0编译时出 (1347)
Oracle切换当前的数据库 (1332)

评论排行

cxf与axis2的比较 (1)
eclipse window 选项下没 (0)
SpringMVC配置方案 (0)
HighCharts导出图片和P( (0)
frameset 给主框架添加滚 (0)
cppunit调研手册 (0)
gtest调研手册 (0)
Jibx使用文档 (0)
无法加载/WEB-INF/prop (0)
解决div遮挡select的问题 (0)

推荐文章

* CSDN日报20170828——《4个方法快速打造你的阅读清单》
* Android检查更新下载安装
* 动手打造史上最简单的Recycleview 侧滑菜单
* TCP网络通讯如何解决分包粘包问题

```
06.     EXPECT_EQ(3,max(2,3));
07. }
08. intmain(int argc, char** argv)
09. {
10.    ::testing::InitGoogleTest(&argc,argv);
11.     returnRUN_ALL_TESTS();
12. }
```

编译CMax_test.cpp，其运行结果如下：

[==========] Running 1 test from 1 testcase.

[----------]Global test environment set-up.

[----------]1 test from CMAX

[RUN      ] CMAX.max

[       OK ] CMAX.max (0 ms)

[----------]1 test from CMAX (0 ms total)

[----------]Global test environment tear-down

[==========]1 test from 1 test case ran. (0 ms total)

[  PASSED ] 1 test.

::testing::InitGoogleTest(&argc,argv): gtest的测试案例允许接收一系列的命令行参数，因此，我们将命令行参数传递给gtest，进行一些初始化操作。

关闭

RUN_ALL_TESTS()： 运行所有测试。

在测试用例中我们使用了TEST这个宏，它有两个参数，官方的对这两个参数的解释为：[TestCaseName，TestName]。

对检查点的检查，我们上面使用到了EXPECT_EQ这个宏，这个宏用来比较两个数字是否相等。Google还包装了一系列EXPECT_*和ASSERT_*的宏，下文将对gtest的断言等进行详细的概括。

**3.gtest断言**

3.1 断言简介

gtest中，断言的宏可以理解为分为两类，一类是ASSERT系列，一类是EXPECT系列。一个直观的解

(1) ASSERT_* 系列的断言，当检查点失败时，退出当前函数。

(2) EXPECT_* 系列的断言，当检查点失败时，继续往下执行。

3.2 示例

```
//int型比较，预期值：3，实际值：Add(1,2)
EXPECT_EQ(3 , Add (1,2) )
```

如果把预期值改成4,则会出现如下测试结果:

```
../src/CMax_Test.cpp:12:Failure
Valueof: add(1,2)
```

Actual: 3

Expected: 4

如果对自动输出的出错信息不满意的话，还可以通过操作符 << 将一些自定义的信息输出，通常，这对于调试或是对一些检查点的补充说明来说是很有用的，例子如下：

如果不使用 << 操作符自定义输出的话：

```cpp
for(int i = 0 ; i < x.size() ; i++)
{
    EXPECT_EQ(x[i],y[i]);
}
```

测试人员不会知道是何时出错的，出错时 i 的值是多少。

而如果使用 << 操作符将一些重要信息输出的话：

```cpp
for (int i = 0; i < x.size(); ++i)
{
     EXPECT_EQ(x[i],y[i])<<"Vectors x and y differ at index "<< i;
}
```

从输出结果中就可以定位到在 i = 多少时出现了错误。这样的输出结果看起来更加容易理解。

### 3.3 布尔值检查

| Fatal assertion | Nonfatal assertion | Verifies |
| --- | --- | --- |
| ASSERT_TRUE(*condition*) | EXPECT_TRUE(*condition*) | *condition* **is true** |
| ASSERT_FALSE(*condition*) | EXPECT_FALSE(*condition*) | *condition* **is false** |

### 3.4 数值类型的检查

| Fatal assertion | Nonfatal assertion | Verifies |
| --- | --- | --- |
| ASSERT_EQ(*expected*,*actual*) | EXPECT_EQ(*expected*,*actual*) | *expected*==*actual* |
| ASSERT_NE(*val1*,*val2*) | EXPECT_NE(*val1*, *val2*) | *val1*!= *val2* |
| ASSERT_LT(*val1*,*val2*) | EXPECT_LT(*val1*, *val2*) | *val1* < *val2* |
| ASSERT_LE(*val1*,*val2*) | EXPECT_LE(*val1*, *val2*) | *val1* <= *val2* |
| ASSERT_GT(*val1*,*val2*) | EXPECT_GT(*val1*, *val2*) | *val1* >*val2* |
| ASSERT_GE(*val1*,*val2*) | EXPECT_GE(*val1*, *val2*) | *val1* >=*val2* |

关闭

3.5字符串类型的检查

| Fatal assertion | Nonfatal assertion | Verifies |
| --- | --- | --- |
| ASSERT_STREQ(*expected*,*actual*) | EXPECT_STREQ(*expected*,*actual*) | **the two C strings have the same content** |
| ASSERT_STRNE(*str1*, *str2*) | EXPECT_STRNE(*str1*, *str2*) | **the two C stri different cc** |
| ASSERT_STRCASEEQ(*expected*,*actual*) | EXPECT_STRCASEEQ(*expected*,*actual*) | **the two C string same content, i case** |
| ASSERT_STRCASENE(*str1*, *str2*) | EXPECT_STRCASENE(*str1*, *str2*) | **the two C string case** |

**4.gtest参数化**

**4.1值参数化测试**

**在设计测试案例时，经常需要考虑给被测函数传入不同的值的情况。我们之前的做法通常是写一个通用方法，然后编写在测试案例调用它。即使使用了通用方法，这样的工作也是有很多重复性的。Google考虑到了这个问题，**

关闭

**并且提供了一个灵活的参数化测试的方案。**

**被测试的函数：**

```cpp
boolIsPrime(int n)
{
    if (n <= 1) return false;
    if (n % 2 == 0) return n == 2;

    for (int i = 3; ; i += 2) {
        if (i > n/i) break;
        if (n % i == 0) return false;
    }
    return true;
}
```

**假如要编写判断结果为true的测试案例，需要传入一系列数值让函数IsPrime()去判断是否为true(其实即使传入再多值也无法确保函数正确)，可以这样编写如下的测试案例：**

```cpp
TEST(IsPrimeTest,HandleTrueReturn)
{
    EXPECT_TRUE(IsPrime(3));
    EXPECT_TRUE(IsPrime(5));
    EXPECT_TRUE(IsPrime(11));
    EXPECT_TRUE(IsPrime(23));
    EXPECT_TRUE(IsPrime(17));
}
```

关闭

我们注意到，在这个测试案例中，我至少复制粘贴了4次，随着测试次数的增多无疑会变得很繁琐。而事实上，gtest可以是这样解决问题的。

（1）告诉gtest你的参数类型是什么。可以添加一个类：testing::TestWithParam<T>，其中T就是你需要参数化的参数类型，比如上面的例子，我们需要参数化一个int型的参数。

```cpp
classIsPrimeParamTest :public::testing::TestWithParam<int>
{

};
```

(2)告诉gtest你拿到参数的值后，具体做些什么样的测试。这里，我们使用一个新的:TEST_P，关于这义，可以理解为"parameterized"。在TEST_P宏里，使GetParam()获取当前的参数的具体值。

```cpp
TEST_P(IsPrimeParamTest,HandleTrueReturn)
{
    int n = GetParam();
    EXPECT_TRUE(IsPrime(n));
}
```

关闭

(3)告诉gtest你想要测试的参数范围是什么。可以使用INSTANTIATE_TEST_CASE_P这宏来告诉gtest你要测试的参数范围： INSTANTIATE_TEST_CASE_P(TrueReturn,IsPrimeParamTest,testing::Values(3,5,11,23,17));

第一个参数是测试案例的前缀，可以任意取。

第二个参数是测试案例的名称，需要和之前定义的参数化的类的名称相同：IsPrimeParamTest

第三个参数是可以理解为参数生成器，上面的例子使用test::Values表示使用括号内的参数。Google提供了一系列的参数生成的函数：

| | |
|---|---|
| Range(begin, end[, step]) | 范围在begin--end之间，步长为step，不包括end |
| Values(v1, v2, ..., Vn) | v1,v2到Vn的值 |
| ValuesIn(container) and ValuesIn(begin,end) | 从一个C类型的数组或是STL容器，或是迭代器中取值 |
| Bool() | 取false 和 true 两个值 |
| Combine(g1, g2, ..., Gn) | 它将g1,g2,...Gn进行排列组合，g1,g2,...Gn本身是一个参数生成器，每次 g1,g2,...Gn中各取出一个值，组合成一个元组(Tuple)作为一个参 |

gtest值参数化完整案例：

parameter.h：

关闭

```cpp
bool IsPrime(int n)
{
    if(n <= 1)
        returnfalse;
    if(n % 2 == 0)
        returnn == 2;
    for(int i = 3;; i += 2)
    {
        if(i > n / i)
            break;
        if(n % i == 0)
            returnfalse;
     }
```

```
14.     returntrue;
15.  }
16.  class IsPrimeParamTest : public::testing::TestWithParam<int>
17.  {
18.  };
```

MainTest.cpp：

```
[cpp]

01.  TEST_P(IsPrimeParamTest, TrueReturn)
02.   {
03.         int n = GetParam();
04.         EXPECT_TRUE(IsPrime(n));
05.   }
06.
07.   INSTANTIATE_TEST_CASE_P(TrueReturn,IsPrimeParamTest,         testing::Values(3,5 11 13))
08.
09.   int main(int argc, char** argv)
10.   {
11.       ::testing::InitGoogleTest(&argc,argv);
12.       returnRUN_ALL_TESTS();
13.   }
```

关闭

4.2 gest类型参数化测试

除了值参数化测试外，gtest还提供了类型参数化测试，以应付各种不同类型数据时的方案。

类型参数化测试步骤:

(1)首先定义一个模版类，继承testing::Test。

```cpp
template<typename T> class FooTest : public testing::Test
   {
        public:
      typedef std::list<T>List;
      T value_;
   };
```

（2）接着定义需要测试的具体数据类型，比如下面定义了char,int和unsigned int:

```
typedef testing::Types<char,int, unsigned int> MyTypes;
TYPED_TEST_CASE(FooTest, MyTypes);
```

(3) 使用宏TYPED_TEST来完成测试案例。在声明模版的数据类型时，使用TypeParam 。

```cpp
TYPED_TEST(FooTest,DoesBlah)
{
    TypeParamn = this->value_;
    typenameTestFixture::List values;
    values.push_back(n);
}
```

关闭

5.gtest死亡测试

这里的死亡指的的是程序的崩溃。通常在测试过程中，我们需要考虑各种各样的输入，有的输入可能直接导致程序崩溃，这时我们就需要检查程序是否按照预期的方式挂掉，这也就是所谓的死亡测试。gtest的死亡测试能做到在一个安全的环境下执行崩溃的测试案例，同时又对崩溃结果进行验证。

gtest的死亡测试所用的宏如下表所示：

| Fatal assertion | Nonfatal assertion | Verifies |
| --- | --- | --- |
| ASSERT_DEATH(*statement, regex`*) | EXPECT_DEATH(*statement, regex`*) | *statement* **crashes with the given error** |
| ASSERT_EXIT(*statement, predicate, regex`*) | EXPECT_EXIT(*statement, predicate, regex`*) | *statement* **exits with the given error and its exit code matches***predicate* |

5.1_DEATH(statement, regex`)

(1) statement是被测试的代码语句。

(2) regex是一个正则表达式，用来匹配异常时在stderr中输出的内容。

例子：

关闭

```cpp
void Foo()
{
    int *pInt = 0;
    *pInt = 42 ;
}
TEST(FooDeathTest, Demo)
{
    EXPECT_DEATH(Foo(), "");
}
```

注意：编写死亡测试案例时，TEST的第一个参数，即testcase_name，请使用DeathTest后缀。原因是gtest会优先运行死亡测试案例，应该是为线程安全考虑。

5.2 *_EXIT(*statement*, *predicate*,*regex*`)

（1）statement是被测试的代码语句。

(2）predicate 在这里必须是一个委托，接收int型参数，并返回bool。只有当返回值为true时，死亡测试案例才算通过。

(3）regex是一个正则表达式，用来匹配异常时在stderr中输出的内容。

要说明的是，*_DEATH其实是对*_EXIT进行的一次包装。



上一篇 Android环境下Ksoap连接Axis2

下一篇 cppunit调研手册