开源 C++单元测试框架 Google Test 指南

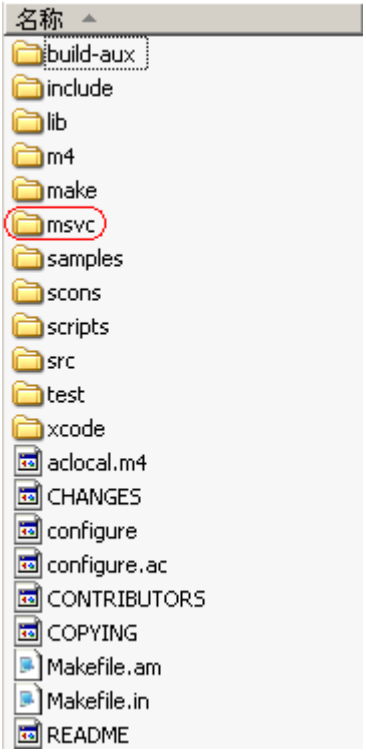
CoderZh

Google Test

Google test 是针对 c/c++的开源测试项目。采用的协议是 BSD license，有很多著名的开源项目采用了它，包括 Chromium（谷歌浏览器开发版）。

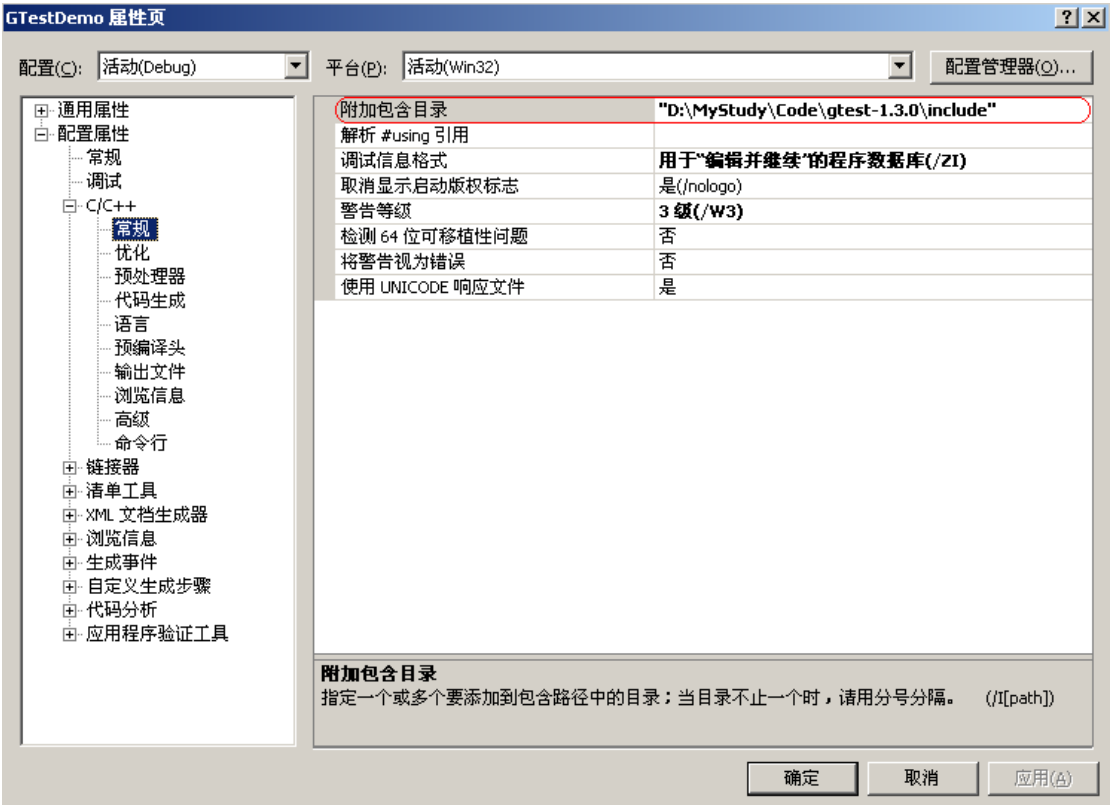
安装配置

下载主页：
http://code.google.com/p/googletest/
官方资料文档：
http://code.google.com/p/googletest/wiki/GoogleTestPrimer
http://code.google.com/p/googletest/wiki/GoogleTestAdvancedGuide
当前的最新版本是 1.5。包含 3 种 tar.bz2，tar.gz 和 zip 格式。解压后的目录结构：



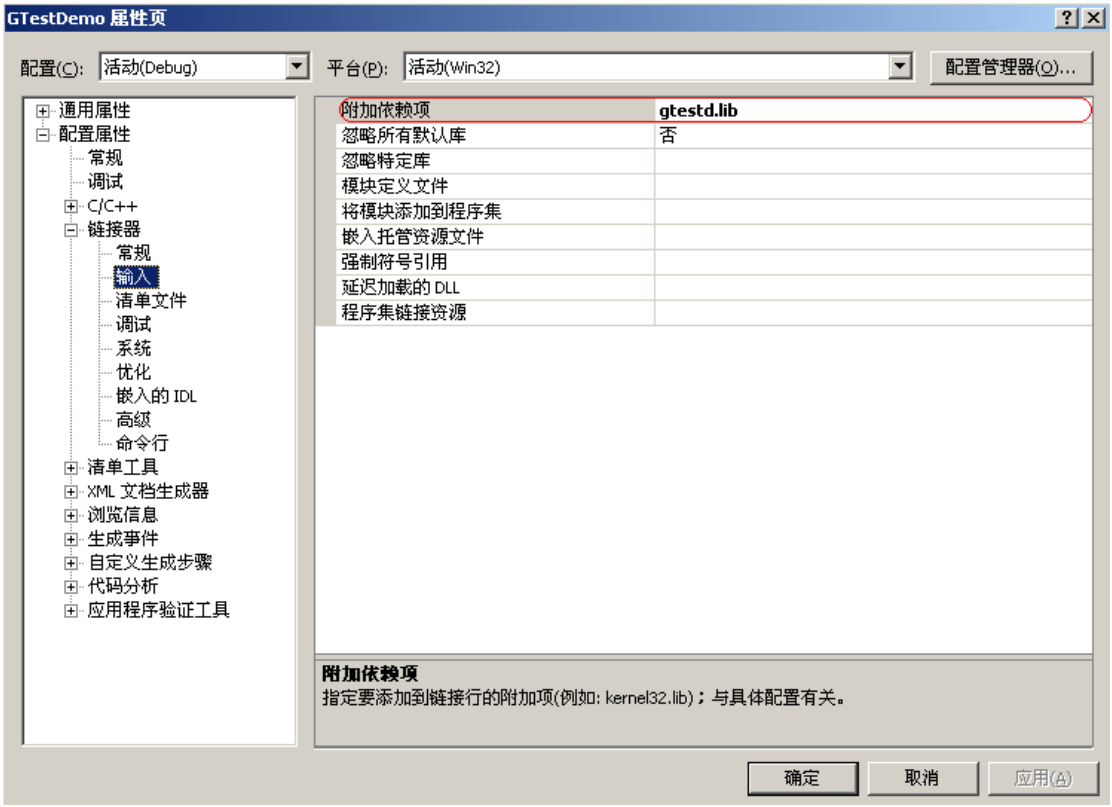
其中的 msvc 就是 VS 的工程目录，可以直接打开进行编译（vs2008 则需要进行工程升级转化），生成相应的 lib 静态库文件。在 vs 中需要在工程中设置 3 个地方，和 ACE 的设置一样：

1. **设置 gtest 的头文件**



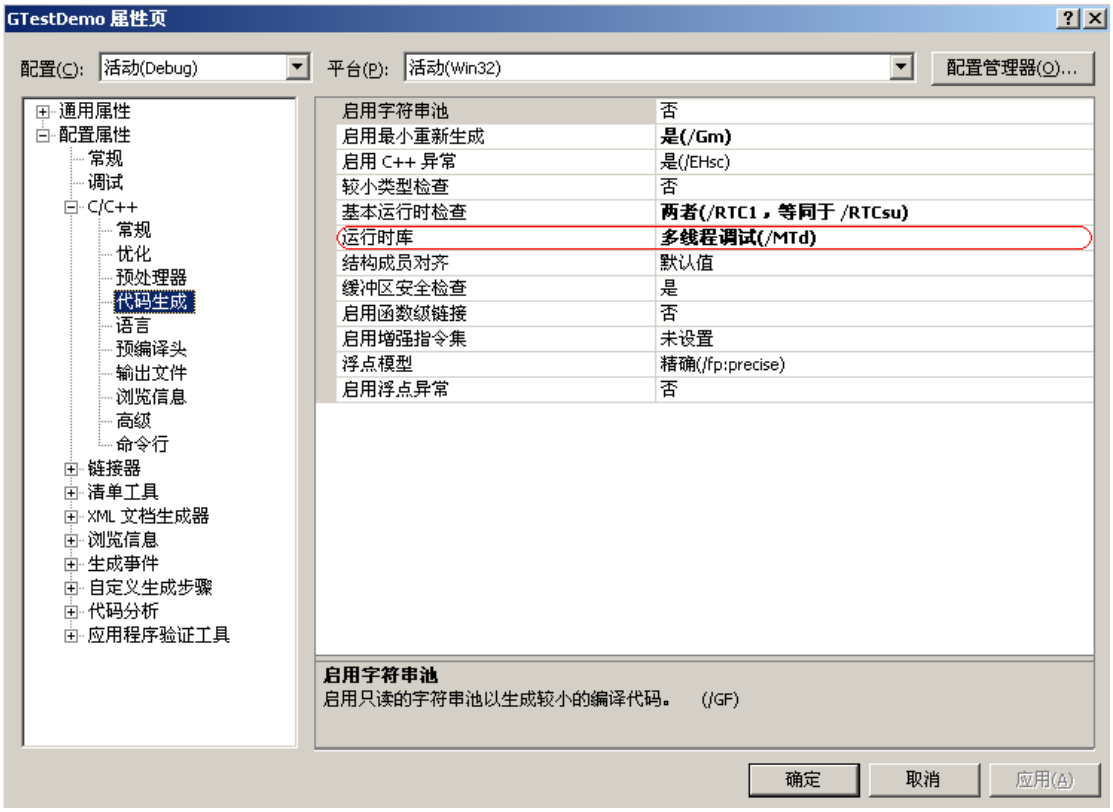
注：如果测试代码需要上库，附加包含目录建议设置为相对路径。

2. 设置 gtest 的 lib 文件



注：如果测试代码需要上库，附加依赖项建议不要带绝对路径。

3. **设置运行时的多线程库支持**



如果是 Release 版本，Runtime Library 设为/MT。当然，其实你也可以选择动态链接（/MD），前提是你之前编译的 gtest 也使用了同样是/MD 选项。

如果是在 Linux 下，就比较方便，和普通的开源软件一样，采用

1. **./configure –prefix=/your install path （如果不带参数默认为/usr/local 下面）**
2. **make**
3. **make install**

然后就可以在工程中进行使用（如果指定了安装目录，则需要-I 和-L 来指明，同时也在最后的 link 加上-lpthread –lgtest）

下面是一个简易的写法：

```
g++ -g -I/data/gtest/include -L/data/gtest/lib -o test main.cpp -lgtest -lpthread
```

By the way: 我在 192.168.100.119 上采用的是默认安装，所以直接加上-lgtest 和-lpthread 就可以了

简单例子

如果需要使用 gtest，则需要包含

```
#include "gtest/gtest.h"
```

下面是一个简单例子：

```cpp
#include "gtest/gtest.h"

int myadd(int a, int b)
{
    return a+b;
}

TEST(Test_myadd, IsReturnAdd)
{
    EXPECT_EQ(2, myadd(4, 10));
    EXPECT_EQ(6, myadd(30, 18));
}

int main(int argc, char* argv[])
{
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

```

编译运行的结果：

```
lsj@Suse-lsj:~/code/test/gtest_main> ./test
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from Test_myadd
[ RUN      ] Test_myadd.IsReturnAdd
main.cpp:10: Failure
Value of: myadd(4, 10)
  Actual: 14
Expected: 2
main.cpp:11: Failure
Value of: myadd(30, 18)
  Actual: 48
Expected: 6
[  FAILED  ] Test_myadd.IsReturnAdd (0 ms)
[----------] 1 test from Test_myadd (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (1 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] Test_myadd.IsReturnAdd

 1 FAILED TEST
```

下面来依次解释：

myadd 是待测试函数名，TEST 是作为 gTest 的一次测试（其实它是由 gTest 包装过的一个宏），第一个参数 Test_myadd 是**测试用例名**，第二个参数 IsReturnAdd 是**测试名**（这两个参数都是自己任意定义的）。在随后的测试结果中将以“**测试用例名.测试名**”的形式呈现

```
[ RUN      ] Test_myadd.IsReturnAdd
```

EXPECT_EQ 用于测试两个数据是否相等。第一个参数可以是你提前预定义的数据，第二个为测试函数名。

main 主函数中：

```
int main(int argc, char* argv[])
{
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

testing::InitGoogleTest 用来处理程序的命令行参数。RUN_ALL_TEST 也是一个宏，用来运行所有的测试用例（本例中就只有一个 TEST）。测试结果英文也很清晰，我就不画蛇添足了。

最后再补充一点，编译后的二进制文件支持 gtest 的命令行参数，可以将数据直接转化为 xml

Z:\code\test\gtest_main\test_detail.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
- <testsuites tests="1" failures="1" disabled="0" errors="0" time="0" name="AllTests">
  - <testsuite name="Test_myadd" tests="1" failures="1" disabled="0" errors="0" time="0">
    - <testcase name="IsReturnAdd" status="run" time="0" classname="Test_myadd">
      - <failure message="Value of: myadd(4, 10) Actual: 14 Expected: 2" type="">
        - <![CDATA[
            main.cpp:10
            Value of: myadd(4, 10)
              Actual: 14
            Expected: 2
          ]]>
        </failure>
      - <failure message="Value of: myadd(30, 18) Actual: 48 Expected: 6" type="">
        - <![CDATA[
            main.cpp:11
            Value of: myadd(30, 18)
              Actual: 48
            Expected: 6
          ]]>
        </failure>
      </testcase>
    </testsuite>
  </testsuites>
```

断言

gtest 采用了大量的宏来包装断言，此断言不同于 c 语言的断言（assert），按照使用方法分为 2 类：

1. **ASSERT 系列（ASSERT_*系列的断言，当检查点失败时，退出当前函数，并非退出当前案例）；**
2. **EXPECT 系列（EXPECT_*系列的断言，当检查点失败时，继续往下运行）**

按照常用功能依次分为 12 类，平常主要用到的是以下几类：

1. **布尔值比较**
2. **数值型数据比较**
3. **字符串比较**
4. **浮点数比较**
5. **近似数比较**
6. **异常检测**
7. **自定义格式函数与参数检查**

布尔值比较		
ASSERT_TRUE(condition)	EXPECT_TRUE(condition)	condition == true
ASSERT_FALSE(condition)	EXPECT_FALSE(condition)	condition == false
数值型数据比较		
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)	val2 >= val2
字符串比较		
ASSERT_STREQ(str1, str2)	EXPECT_STREQ(str1, str2)	两个 C 字符串内容相同（同时支持 char *和 wchar_t *类型）
ASSERT_STRNE(str1, str2)	EXPECT_STRNE(str1, str2)	两个 C 字符串内容不同（同时支持 char *和 wchar_t *类型）
ASSERT_STRCASEEQ(str1,str2)	EXPECT_STRCASEEQ(str1,str2)	两个 C 字符串内容相同，忽略大小写（只支持 char *类型）
ASSERT_STRCASENE(str1,str2)	EXPECT_STRCASENE(str1,str2)	两个 C 字符串内容不同，忽略大小写（只支持 char *类型）
浮点数比较		
ASSERT_FLOAT_EQ(val1,val2)	EXPECT_FLOAT_EQ(val1,val2)	the two float values are almost equal
ASSERT_DOUBLE_EQ(val1,val2)	EXPECT_DOUBLE_EQ(val1,val2)	the two double values are almost equal
近似数比较		
ASSERT_NEAR(val1, val2, abs_error)	EXPECT_NEAR(val1, val2, abs_error)	两个数值 val1 和 val2 的绝对值差不超过 abs_error
异常检查		
ASSERT_THROW(statement, exception_type)	EXPECT_THROW(statement, exception_type)	抛出指定类型异常
ASSERT_THROW(statement)	EXPECT_THROW(statement)	抛出任意类型异常
ASSERT_NO_THROW(statement)	EXPECT_NO_THROW(statement)	不抛异常

函数值与参数检查（目前最多只支持 5 个参数）		
ASSERT_PRED1(pred1, val1)	EXPECT_PRED1(pred1, val1)	pred1(val1) returns true
ASSERT_PRED2(pred2, val1, val2)	EXPECT_PRED2(pred2, val1, val2)	pred2(val1, val2) returns true
Windows HRESULT		
ASSERT_HRESULT_SUCCEEDED (expression)	EXPECT_HRESULT_SUCCEEDED (expression)	expression is a success HRESULT
ASSERT_HRESULT_FAILED (expression)	EXPECT_HRESULT_FAILED (expression)	expression is a failure HRESULT
自定义格式函数与参数检查（目前最多支持 5 个参数）		
ASSERT_PRED_FORMAT1(pred1, val1)	EXPECT_PRED_FORMAT1(pred1, val1)	pred1(val1) is successful
ASSERT_PRED_FORMAT1(pred1, val1, val2)	EXPECT_PRED_FORMAT1(pred1, val1, val2)	pred2(val1, val2) is successful

下面将用一个实例来演示：

我们编写了一个 Configure 的 class，提供了 3 个对外的接口方法:

1. **get_size(void)**
2. **add_item(string str)**
3. **get_item(int index)**

现在需要对其进行测试，那么就应该依次有这 3 个文件：

1. **Configure.h**
2. **Configure.cpp**
3. **main.cpp**

首先是 Configure.h：

```
#ifndef _CONFIGURE_H_
#define _CONFIGURE_H_

#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class Configure
{
public:
    int get_size(void);
    int add_item(string str);
    string get_item(int index);
private:
    vector<string> m_v_Items;
};

#endif
```

接着是 Configure.cpp：

```
#include "Configure.h"

int
Configure::add_item(string str)
{
    vector<string>::const_iterator pIter = find(m_v_Items.begi
n(), m_v_Items.end(), str);
    if(m_v_Items.end() != pIter){
        return pIter - m_v_Items.begin();
    }
    m_v_Items.push_back(str);
    return m_v_Items.size() - 1;
}

string
Configure::get_item(int index)
{
    if(m_v_Items.size() < index){
        return "";
    }else{
        return m_v_Items.at(index);
    }
}

int
Configure::get_size(void)
{
    return m_v_Items.size();
}
```

最后是主函数调用：

```cpp
#include "gtest/gtest.h"
#include "Configure.h"

TEST(ConfigureTest, add_Item)
{
    //初始化
    Configure* p = new Configure();

    //布尔值判断
    ASSERT_TRUE(NULL != p);

    //压入测试数据
    p->add_item("A");
    p->add_item("B");
    p->add_item("C");

    //进行测试
    EXPECT_EQ(p->get_size(), 2);
    EXPECT_STREQ(p->get_item(0).c_str(), "A");
    EXPECT_STREQ(p->get_item(1).c_str(), "B");
    EXPECT_STREQ(p->get_item(10).c_str(), "");

    //清理
    if(NULL != p){
        delete p;
        p = NULL;
    }
    p = NULL;
}

int main(int argc, char* argv[])
{
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

```

进行编译后的执行：

```
lsj@Suse-lsj:~/code/test/gtest_assert> ./test
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from ConfigureTest
[ RUN      ] ConfigureTest.add_Item
main.cpp:18: Failure
Value of: 2
Expected: p->get_size()
Which is: 3
[  FAILED  ] ConfigureTest.add_Item (0 ms)
[----------] 1 test from ConfigureTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (1 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] ConfigureTest.add_Item

 1 FAILED TEST
```

事件机制

Gtest 提供了多种事件机制方便在测试用例之前或者完成以后进行一些操作，按照使用方法分为 3 类：

1. **全局事件：在所有测试用例执行之前和完成之后生效。可以在全局事件中完成一些测试环境的初始化和资源回收工作，比如预留内存申请/回收，组件对象初始化/析构等。**
2. **TestSuite 级别：在指定的测试套第一个测试用例之前，最后一个测试用例之后。如果根据子模块定义测试套，那么就可以在 TestSuite 事件中完成一些子模块的线程、消息队列等的初始化和资源回收工作。**
3. **TestCase 级别：在每个测试用例执行前后，即在每个测试代码的断言前后进行执行。**

以下分别是 3 类事件的用法：

全局事件：必须通过继承 testing::Environment 类，实现里面的 SetUp 和 TearDown 方法：

1. **SetUp()方法在所有用例执行前执行；**
2. **TearDown()方法在所有用例执行后执行；**

```
class TestEnvironment : public testing::Environment
{
public:
    virtual void SetUp()
    {
        std::cout << "Test TestEnvironment SetUP." << std::endl;
    }
    virtual void TearDown()
    {
        std::cout << "Test TestEnvironment TearDown." << std::endl;
    }
};
```

完成继承类方法实现以后，还需要告诉 gtest 添加全局事件，我们需要在 main 函数中通过 testing::AddGlobalTestEnvironment 方法添加该全局事件。如果需要增加全局事件，也可以写多个继承类，然后将事件都添加到测试用例之前。

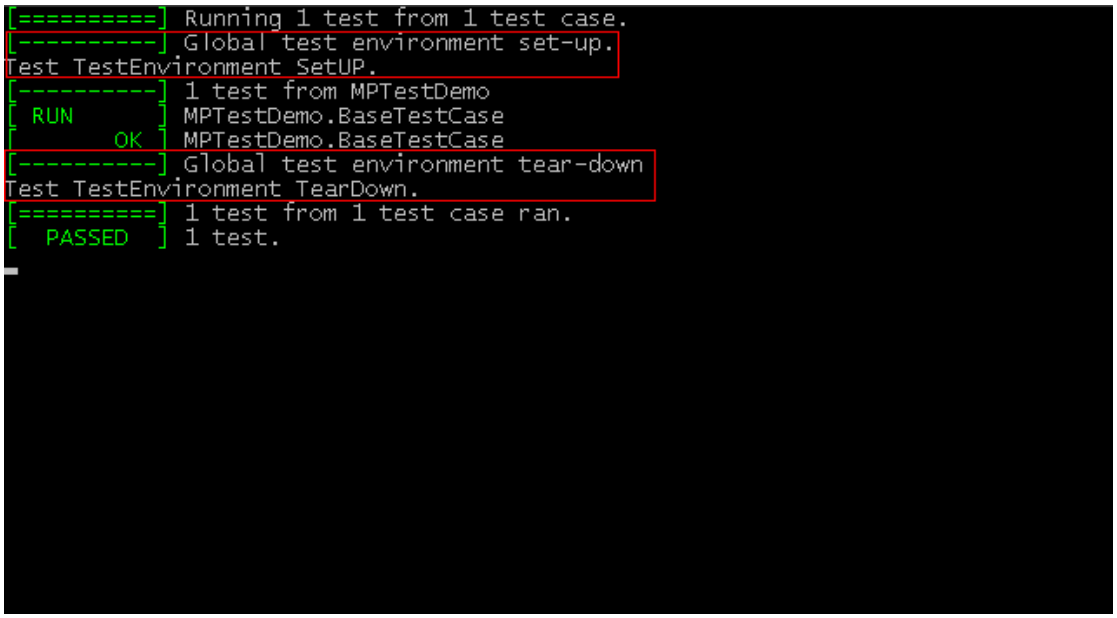
```
int main(int argc, char** argv)
{
    int m_ui_result = 0;

    // 添加全局事件
    testing::AddGlobalTestEnvironment(new TestEnvironment);
    testing::InitGoogleTest(&argc, argv);

    m_ui_result = RUN_ALL_TESTS();

    return m_ui_result;
}
```

以下是运行全局事件以后的显示结果：



TestSuite 事件：需要通过继承 testing::Test 类，实现里面的 SetUpTestCase 和 TearDownTestCase 两个静态方法：

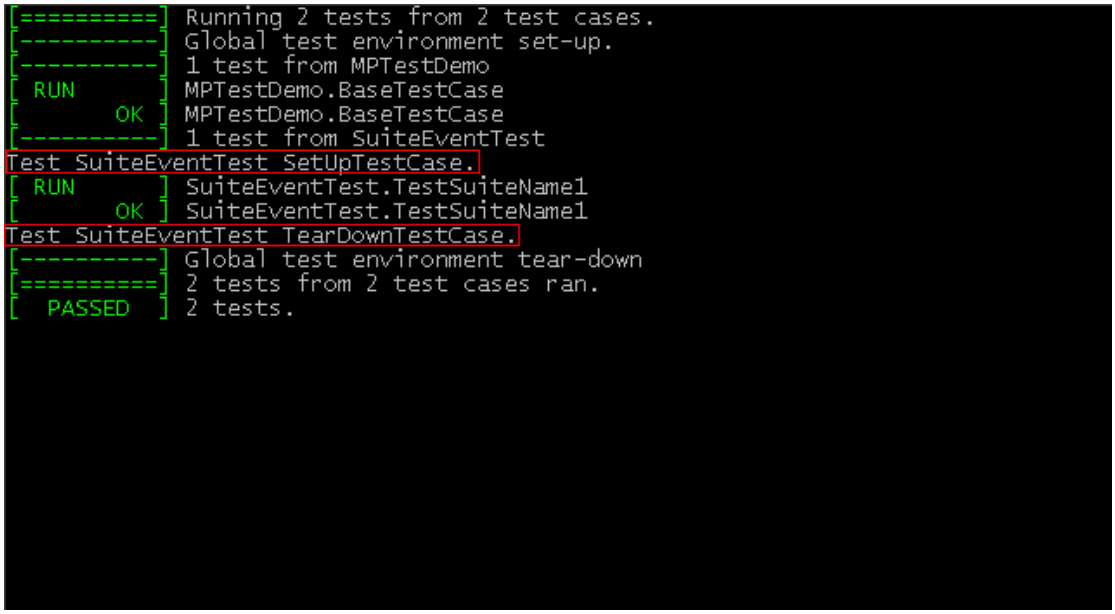
1. **SetUpTestCase()方法在 TestSuit 的第一个 TestCase 之前执行；**
2. **TearDownTestCase()方法在 TestSuite 的最后一个 TestCase 之后执行；**

```
class SuiteEventTest : public testing::Test
{
public :
    static void SetUpTestCase()
    {
        std::cout << "Test SuiteEventTest SetUpTestCase." << std::endl;
    }
    static void TearDownTestCase()
    {
        std::cout << "Test SuiteEventTest TearDownTestCase." << std::endl;
    }
};
```

在编写测试用例时，需要使用 TEST_F 宏，第一个参数必须是上面的继承类名字，代表一个 TestSuite。

```
// 测试套事件
TEST_F(SuiteEventTest, TestSuiteName1)
{
    // 这里添加测试套用例TestCaseName1
    return;
}
```

以下是运行 TestSuite 事件以后的显示结果：



TestCase 事件：与 TestSuite 事件实现方法相同，需要通过继承 testing::Test 类，但是只需要实现里面的 SetUp 和 TearDown 两个方法：

1. **SetUp()方法在每个 TestCase 之前执行；**
2. **TearDown()方法在每个 TestCase 之后执行；**

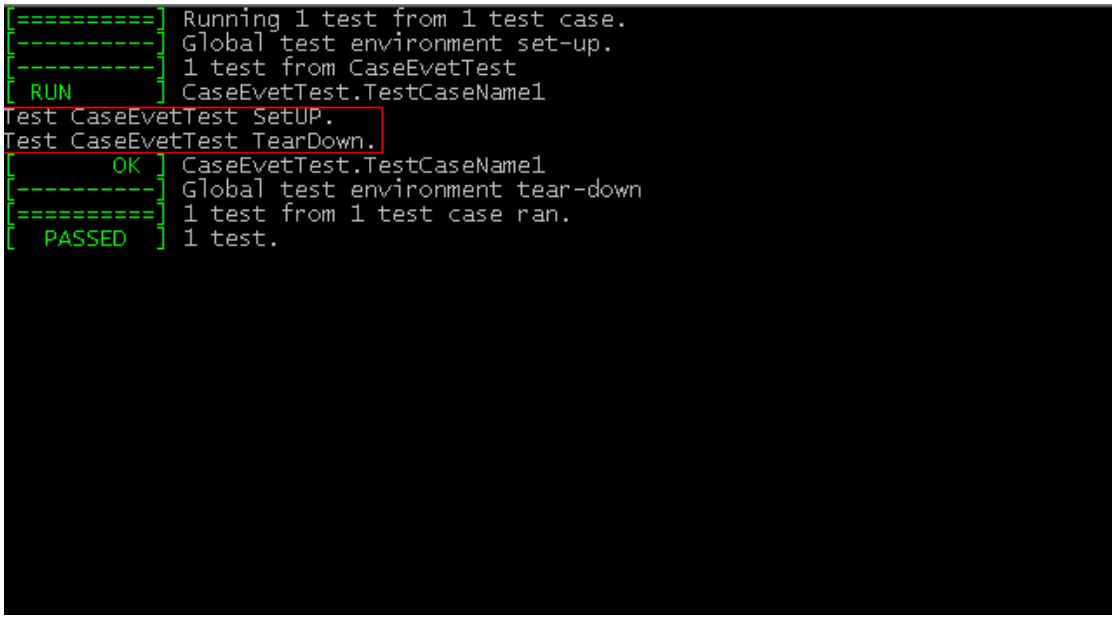
```
class CaseEvetTest : public testing::Test
{
public :
    virtual void SetUp()
    {
        std::cout << "Test CaseEvetTest SetUP." << std::endl;
    }
    virtual void TearDown()
    {
        std::cout << "Test CaseEvetTest TearDown." << std::endl;
    }
};
```

在编写测试用例时，需要使用 TEST_F 宏，第一个参数必须是上面的继承类名字，代表一个 TestSuite，并且在测试套中添加测试用例。

```
// 测试用例事件
TEST_F(CaseEvetTest, TestCaseName1)
{
    EXPECT_EQ(1, (1 + 1));

    return;
}
```

以下是运行 TestSuite 事件以后的显示结果：



参数化

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，以前的做法一般是写一个通用方法，然后编写在测试案例调用它，即使使用了通用方法，也需要很多重复性的工作。以下是一般的测试方法，如果需要测试 N 个数字，则需要拷贝复制粘贴 N 次：

```
TEST(ParameterTest, TestParamName1)
{
    c_test_class *p_c_testcase = new(nothrow) c_test_class();

    EXPECT_TRUE(p_c_testcase->is_prime(20));
    EXPECT_TRUE(p_c_testcase->is_prime(21));
    EXPECT_TRUE(p_c_testcase->is_prime(22));
    EXPECT_TRUE(p_c_testcase->is_prime(23));
    EXPECT_TRUE(p_c_testcase->is_prime(24));
    EXPECT_TRUE(p_c_testcase->is_prime(25));

    if (NULL != p_c_testcase)
    {
        delete p_c_testcase;
        p_c_testcase = NULL;
    }
}
```

gTest 在这里提供了一个灵活的函数参数化测试的方案：

1. **告诉 gtest 参数类型：必须添加一个继承类 testing::TestWithParam<T>，其中 T 就是需要参数化的参数类型。以上面为例，需要参数化一个 int 型的参数；**

```
// int参数化测试
class ParameterTest : public::testing::TestWithParam<int>
{
public :
};
```

2. 参数类型确定以后，需要使用一个新的宏 TSET_P 进行测试用例，在 TEST_P 宏里，使用 GetParam()方法获取当前的参数的具体值；

```
// 测试参数化
TEST_P(ParameterTest, TestParamName2)
{
    c_test_class *p_c_testcase = new(nothrow) c_test_class();

    int n = GetParam();
    EXPECT_TRUE(p_c_testcase->is_prime(n));

    if (NULL != p_c_testcase)
    {
        delete p_c_testcase;
        p_c_testcase = NULL;
    }
}
```

3. 使用 INSTANTIATE_TEST_CASE_P 宏来确定测试的参数取值范围；

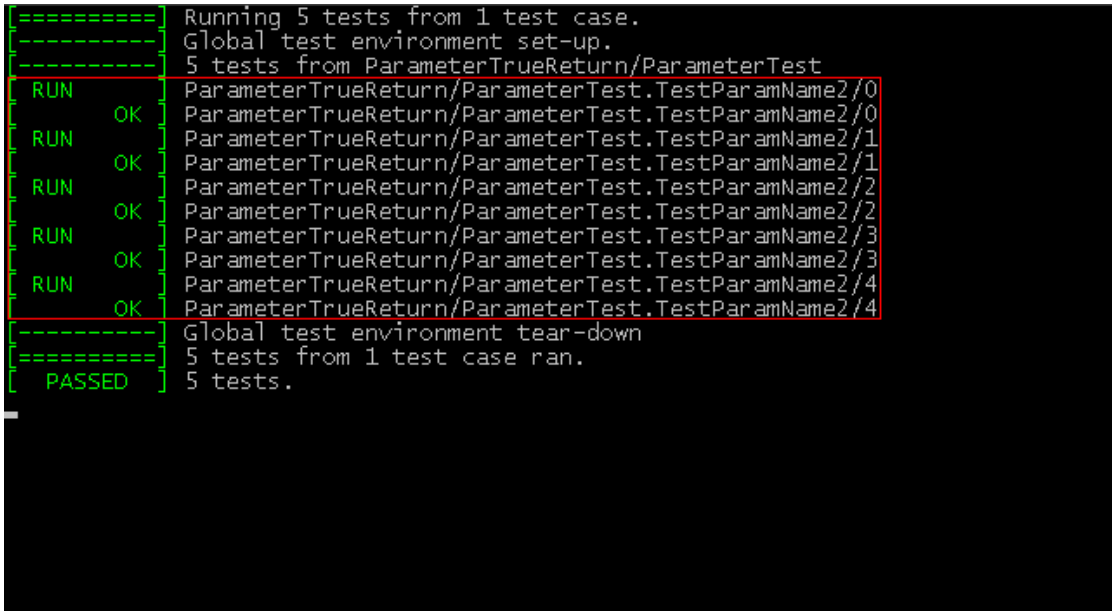
```
INSTANTIATE_TEST_CASE_P(ParameterTrueReturn, ParameterTest, testing::Values(3, 5, 11, 23, 17));
```

其中：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，如：ParameterTest ；第三个参数是可以理解为参数生成器，上面的例子使用 testing::Values 表示使用括号内的参数。

Google 提供了以下一系列的参数生成的函数：

Range(begin, end[, step])	范围在 begin 和 end 之间，步长为 step，不包括 end 边界
Values(v1, v2, ..., vN)	从 v1, v2 到 vN 的值
ValuesIn(container), ValuesIn(begin, end)	从一个 C 类型的数组或者 STL 容器或者迭代器中取值
Bool()	取 false 和 true 两个值
Combine(g1, g2, ..., gN)	将 g1, g2, ..., gN 进行排列组合，g1, g2, ..., gN 本身是一个参数生成器，每次分别从 g1, g2, ..., gN 中各取出一个值，组合成一个元组(Tuple)作为一个参数 说明：这个功能只在提供了<tr1/tuple>头的系统中有效。gtest 会自动去判断是否支持 tr/tuple，如果系统确实支持，而 gtest 判断错误的话，可以重新定义宏 GTEST_HAS_TR1_TUPLE = 1。

以下是执行函数参数化用例以后的显示结果，最终输出测试结果命名规则为：



ParameterTrueReturn/ParameterTest.TestParamName2/ValuesIndex

除了测试用例可以参数化以外，gtest 还提供了针对各种不同类型数据时的方案，以及参数化类型的方案。下面以 gtest 的例子为例进行介绍：

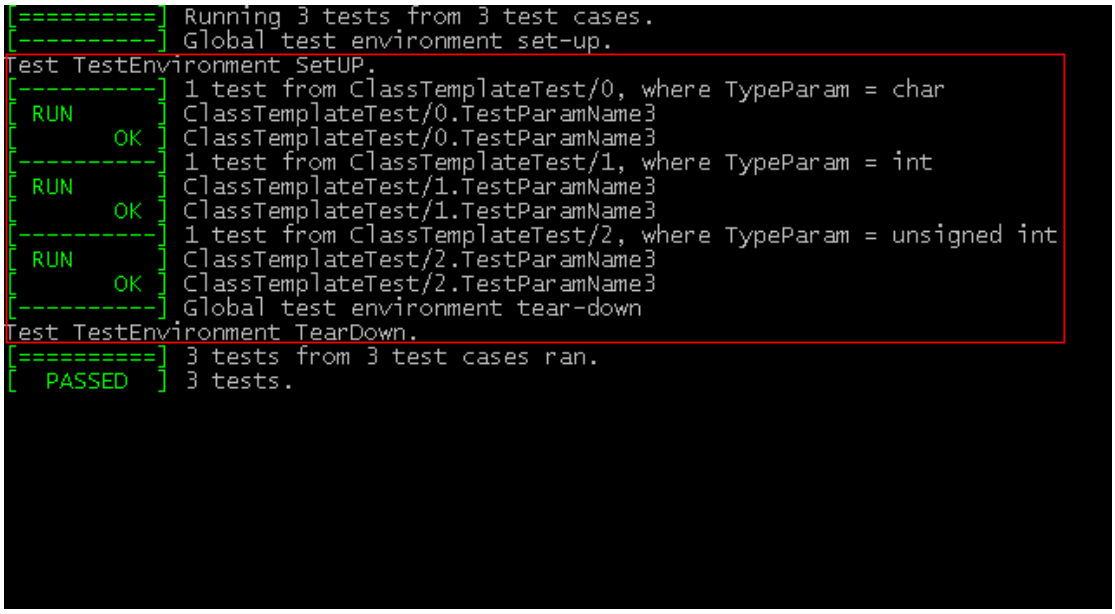
1. **首先需要定义一个模版类，从 testing::Test 类继承；**

```
// 类型参数化测试
template <typename T>
class ClassTemplateTest : public testing::Test
{
public:
    typedef std::list<T> List;
    static T m_s_shared;
    T m_t_test;
};
```

2. **然后再定义需要测试的具体数据类型，比如下面定义了测试 char, int 和 unsigned int 类型；**

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

3. **以下是执行类型参数化用例以后的显示结果；**



我们在这里使用函数参数化或者类型参数化，基本上就可以满足正常单元测试需要。

死亡测试

“死亡测试”名字比较恐怖，这里的“死亡”是指程序的崩溃。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。

死亡测试宏定义		
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

由于有些异常只在 Debug 下抛出，因此还提供了*_DEBUG_DEATH，用来处理 Debug 和 Realease 下的不同。

简单来说，通过*_DEATH(statement, regex)和*_EXIT(statement, predicate, regex)，我们可以非常方便的编写导致崩溃的测试案例，并且在不影响其他案例执行的情况下，对崩溃案例的结果进行检查。

以下是*_DEATH 用法介绍：

1. **statement 是被测试的代码语句，这里可以使用表达式，也可以直接调用函数结果；**
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；**
3. **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。**

```
// 死亡测试
TEST(FunctionDeathTest, TestDeathName1)
{
    c_test_class *p_c_testcase = new(nothrow) c_test_class();

    EXPECT_DEATH(p_c_testcase->get_throw(NULL), "Test FunctionDeathTest.");

    if (NULL != p_c_testcase)
    {
        delete p_c_testcase;
        p_c_testcase = NULL;
    }
}
```

以上代码中，通过执行 get_throw(NULL)抛出异常来结束程序，在运行过程中 gtest 执行用例出错以后没有直接抛出异常，而是捕获了此异常信息，以下是运行结果：

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from FunctionDeathTest
[ RUN      ] FunctionDeathTest.TestDeathName1
d:\mystudy\demo\gtestdemo\gtestdemo.cpp(192): error: Death test: p_c_testcase->g
et_throw(0)
    Result: died but not with expected error.
  Expected: Test FunctionDeathTest.
Actual msg:
[  FAILED  ] FunctionDeathTest.TestDeathName1
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran.
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] FunctionDeathTest.TestDeathName1

 1 FAILED TEST
```

其中 stderr 内容通过正则表达式输出：

在 POSIX 系统（Linux, Cygwin, 和 Mac）中，gtest 的死亡测试中使用的是 POSIX 风格的正则表达式（关于 POSIX 风格的正则表达式资料请参照相关文档），宏定义 **GTEST_USES_POSIX_RE = 1**；

在 Windows 系统中，gtest 的死亡测试中使用的是 gtest 自己实现的简单的正则表达式语法。相比 POSIX 风格，gtest 的简单正则表达式少了很多内容，比如 ("x|y"), ("(xy)"), ("[xy]") 和 ("x{5,7}")都不支持，宏定义 **GTEST_USES_SIMPLE_RE=1**。

\\d	匹配数字
\\D	匹配非数字
\\f	匹配 \f
\\n	匹配 \n
\\r	匹配 \r
\\s	匹配所有 ASCII 字符（包括 whitespace，\n）
\\S	匹配非空格字符
\\t	匹配 \t
\\v	匹配 \v

\\w	匹配所有字母，下划线和数字
\\W	匹配所有\\w 无法匹配的字母
\\c	matches any literal character c, which must be a punctuation
.	matches any single character except \n
A?	matches 0 or 1 occurrences of A
A*	matches 0 or many occurrences of A
A+	matches 1 or many occurrences of A
^	matches the beginning of a string (not that of each line)
$	matches the end of a string (not that of each line)
xy	matches x followed by y

死亡测试有以下两种运行方式：

1. **fast 方式（默认的方式）：**
   testing::FLAGS_gtest_death_test_style = "fast";
2. **threadsafe 方式：**
   testing::FLAGS_gtest_death_test_style = "threadsafe";

可以在 main() 里为所有的死亡测试设置测试形式，也可以为某次测试单独设置。gtest 会在每次测试之前保存这个标记并在测试完成后恢复，所以可以不需要去管理这部分工作。
以下是所有死亡测试的设置方式：

```
int main(int argc, char** argv)
{
    int m_ui_result = 0;

    // 输出XML参数
    testing::GTEST_FLAG(output) = "xml:";
    // 添加全局事件
    testing::AddGlobalTestEnvironment(new TestEnvironment);
    testing::InitGoogleTest(&argc, argv);
    // 以fast方式运行死亡测试
    testing::FLAGS_gtest_death_test_style = "fast";

    m_ui_result = RUN_ALL_TESTS();

    return m_ui_result;
}
```

以下是测试套死亡测试的设置方式：

```
// 死亡测试
TEST(FunctionDeathTest, TestDeathName1)
{
    // 在测试套中以threadsafe方式运行死亡测试
    testing::FLAGS_gtest_death_test_style = "threadsafe";

    c_test_class *p_c_testcase = new(nothrow) c_test_class();

    EXPECT_DEATH(p_c_testcase->get_throw(NULL), "Test FunctionDeathTest.");

    if (NULL != p_c_testcase)
    {
        delete p_c_testcase;
        p_c_testcase = NULL;
    }
}
```

注意事项：

1. **不要在死亡测试里释放内存；**
2. **不要在父进程里再次释放内存；**
3. **不要在程序中使用内存堆检查。**

运行参数

使用 gtest 编写的测试案例通常本身就是一个可执行文件，因此运行起来非常方便。同时，gtest 也为我们提供了一系列的运行参数（环境变量、命令行参数或代码里指定），使得我们可以对案例的执行进行一些有效的控制。gtest 提供了三种设置的途径：

1. **系统环境变量**
2. **命令行参数**
3. **代码中指定 FLAG**

其优先级原则是，最后设置的那个会生效。通常情况下的优先级顺序为：命令行参数 > 代码中指定 FLAG > 系统环境变量。由于在 gtest 工程 main 函数中，gtest 通过 testing::InitGoogleTest 方法直接处理输入参数，因此测试用例可以处理命令行参数。

```
int main(int argc, char** argv)
{
    int m_ui_result = 0;

    // 输出XML参数，默认在工程目录下生成XML文件
    testing::GTEST_FLAG(output) = "xml:";
    // 添加全局事件
    testing::AddGlobalTestEnvironment(new TestEnvironment);
    testing::InitGoogleTest(&argc, argv);
    // 以fast方式运行死亡测试
    testing::FLAGS_gtest_death_test_style = "fast";

    m_ui_result = RUN_ALL_TESTS();

    return m_ui_result;
}
```

这样就拥有了接收和响应 gtest 工程命令行参数的能力。如果需要在代码中指定 FLAG，可以使用 testing::GTEST_FLAG 这个宏来设置。比如相对于命令行参数--gtest_output，可以使用 testing::GTEST_FLAG(output) = "xml:";来设置。注意这里不需要加--gtest 前缀了，同时，推荐将这句放置 InitGoogleTest 之前，这样就可以使得对于同样的参数，命令行参数优先级高于代码中指定。

```
int main(int argc, char** argv)
{
    int m_ui_result = 0;

    // 输出XML参数，默认在工程目录下生成XML文件
    testing::GTEST_FLAG(output) = "xml:";
    // 添加全局事件
    testing::AddGlobalTestEnvironment(new TestEnvironment);
    testing::InitGoogleTest(&argc, argv);
    // 以fast方式运行死亡测试
    testing::FLAGS_gtest_death_test_style = "fast";

    m_ui_result = RUN_ALL_TESTS();

    return m_ui_result;
}
```

如果需要 gtest 的设置系统环境变量，必须注意的是：

1. **系统环境变量全大写，比如对于--gtest_output，相应的系统环境变量为 GTEST_OUTPUT**
2. **有一个命令行参数例外，那就是--gtest_list_tests，它是不接受系统环境变量的，只是用来罗列测试案例名称。**

以下是所有命令行参数列表：

1. **测试案例集合：**

命令行参数	说明
--gtest_list_tests	使用这个参数时，将不会执行里面的测试案例，而是输出一个案例的列表。
--gtest_filter	对执行的测试案例进行过滤，支持通配符 ?　单个字符 *　任意字符 -　排除，如-a 表示除了 a :　取或，如 a:b 表示 a 或 b 比如下面的例子： ./foo_test 没有指定过滤条件，运行所有案例； ./foo_test --gtest_filter=* 使用通配符*，表示运行所有案例； ./foo_test --gtest_filter=FooTest.* 运行所有“测试案例名称(testcase_name)”为 FooTest 的案例； ./foo_test --gtest_filter=*Null*:*Constructor* 运行所有“测试案例名称(testcase_name)”或“测试名称(test_name)”包含 Null 或 Constructor 的案例； ./foo_test --gtest_filter=-*DeathTest.* 运行所有非死亡测试案例； ./foo_test --gtest_filter=FooTest.*-FooTest.Bar 运行所有“测

	试案例名称(testcase_name)”为 FooTest 的案例，但是除了 FooTest.Bar 这个案例
--gtest_also_run_disabled_tests	执行案例时，同时也执行被置为无效的测试案例。关于设置测试案例无效的方法为：在测试案例名称或测试名称中添加 DISABLED 前缀。
--gtest_repeat=[COUNT]	设置案例重复运行次数： --gtest_repeat=1000：重复执行 1000 次，即使中途出现错误； --gtest_repeat=-1： 无限次数执行 --gtest_repeat=1000 --gtest_break_on_failure：重复执行 1000 次，并且在第一个错误发生时立即停止，这个功能对调试非常有用。 --gtest_repeat=1000 --gtest_filter=FooBar：重复执行 1000 次测试案例名称为 FooBar 的案例。

2. 测试案例输出

命令行参数	说明
--gtest_color=(yes|no|auto)	输出命令行时是否使用一些五颜六色的颜色，默认是 auto。
--gtest_print_time	输出命令行时是否打印每个测试案例的执行时间，默认是不打印的。
--gtest_output= xml[:DIRECTORY_PATH|:FILE_PATH]	将测试结果输出到一个 xml 中。 --gtest_output=xml: 不指定输出路径时，默认为案例当前路径。 --gtest_output=xml:d:\ 指定输出到某个目录 --gtest_output=xml:d:\foo.xml 指定输出到 d:\foo.xml 如果不是指定了特定的文件路径，gtest 每次输出的报告不会覆盖，而会以数字后缀的方式创建。

3. 对案例的异常处理

命令行参数	说明
--gtest_break_on_failure	调试模式下，当案例失败时停止，方便调试。
--gtest_throw_on_failure	当案例失败时以 C++异常的方式抛出。
--gtest_catch_exceptions	是否捕捉异常。gtest 默认是不捕捉异常的，因此假如测试案例抛了一个异常，很可能会弹出一个对话框，这非常的不友好，同时也阻碍了测试案例的运行。如果想不弹这个框，可以通过设置这个参数来实现。如将 --gtest_catch_exceptions 设置为一个非零的数。 注意：该参数只在 Windows 下有效。

以下是命令行参数列表在使用过程中遇到的一些问题总结：

1. **同时使用--gtest_filter 和--gtest_output=xml:时，在 xml 测试报告中能否只包含过滤后的测试案例的信息。**
2. **有时在代码中设置 testing::GTEST_FLAG(catch_exceptions) = 1 和在命令行中使用**

--gtest_catch_exceptions 结果稍有不同，在代码中设置 FLAG 方式有时候捕捉不了某些异常，但是通过命令行参数的方式一般都不会有问题。最后处理办法是既在代码中设置 FLAG，又在命令行参数中传入 --gtest_catch_exceptions，估计是 gtest 在 catch_exceptions 方面不够稳定的原因导致。

附件

以下是一些素材来源和更详细的 gtest 介绍资料，本文档暂时只提供一些基础的介绍和演示，如果还需要更详细的了解，建议阅读 gtest 源代码或者上网查找更深入详细的资料：

玩转Google开源C++
单元测试框架Googl