一. Google Test

1. 环境搭载

1.1 下载编译

（1） 下载 googletest-release-1.8.x: https://github.com/google/googletest

（2） 编译生成：gtest.lib、gtest_main.lib

1.2 配置

（1） 项目中包含 include 文件夹，链接 gtest.lib 或 gtest_main.lib。

（2） 添加#include”gtest.h”

1.3 运行

（1） 依赖 gtest_main.lib 提供了测试应用程序入口点的默认实现

（2） 依赖 gtest.lib 需要自己写套路化的 main 函数以启动测试

```
int main(int argc, char* argv[])
{

    ::testing::InitGoogleTest(&argc, argv);
    RUN_ALL_TESTS();
    return 0;
}
```

（3） 编写 TEST， TEST 宏的作用是创建一个简单测试，它定义了一个测试函数，在这个函数里可以使用任何 C++代码并使用提供的断言来进行检查。

```
TEST(test_case_name, test_name)
{
  //测试语句
}
```

- test_case_name 是测试用例名,通常是取测试函数名或者测试类名
- test_name 第二个参数是测试名可随便取
- 测试结果将以"测试用例名.测试名"的形式显示

2. 断言

Google Test 断言通过类似于函数调用的宏来实现，通过断言行为来测试一个类或函数。Gtest 中断言的宏都是成对出现，被分为两个系列：

（1） ASSERT_*系列：该系列断言当检查失败时，产生致命错误并中止当前调用它的函数执行，不再执行该测试用例中的后续流程。致命断言(FAIL* and ASSERT_*) 只能在无返回值的函数里使用。

（2） EXPECT_*系列：该系列断言检查失败时，不影响后续流程，继续往下执行

例如：

当 15 行的 ASSERT 系列断言检查失败时，不再执行 EXPECT_EQ(3,add(2,3))

```
13  TEST(AddTest, Int)
14  {
15      ASSERT_EQ(3,add(2,3));
16      EXPECT_EQ(1,add(4,5));
17  }
```

```
[ RUN      ] AddTest.Int
D:\UnitTest\dcaxFile\dcaxFileSystemTest.cpp(15): error:       Expected: 3
To be equal to: add(2,3)
      Which is: 5
[  FAILED  ] AddTest.Int (2 ms)
```

当 15 行的 EXPECT 系列断言检查失败时，继续执行 EXPECT_EQ(1,add(4,5))

```
13  TEST(AddTest, Int)
14  {
15      EXPECT_EQ(3,add(2,3));
16      EXPECT_EQ(1,add(4,5));
17  }
```

```
[ RUN      ] AddTest.Int
D:\UnitTest\dcaxFile\dcaxFileSystemTest.cpp(15): error:       Expected: 3
To be equal to: add(2,3)
      Which is: 5
D:\UnitTest\dcaxFile\dcaxFileSystemTest.cpp(16): error:       Expected: 1
To be equal to: add(4,5)
      Which is: 9
[  FAILED  ] AddTest.Int (2 ms)
```

2.1 基础断言

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_TRUE(condition);	EXPECT_TRUE(condition);	condition is true
ASSERT_FALSE(condition);	EXPECT_FALSE(condition);	condition is false

例如：

```
TEST(Route, Iterator)
{
    //iterator begin() const;
    auto it=p1.begin();
    EXPECT_TRUE(*it=="D:");
```

2.2 二进制比较

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_EQ(val1,val2);	EXPECT_EQ(val1,val2);	val1 == val2
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

例如：

```
//path& repalce_extension(const path& replacement=path());
m_p3.replace_extension(".xml");
EXPECT_EQ("D:\\filetest\\test.xml", m_p3.string());
```

2.3 字符串比较

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_STREQ(str1,str2);	EXPECT_STREQ(str1,str2);	the two C strings have the same content
ASSERT_STRNE(str1,str2);	EXPECT_STRNE(str1,str2);	the two C strings have different content
ASSERT_STRCASEEQ(str1,str2);	EXPECT_STRCASEEQ(str1,str2);	the two C strings have the same content, ignoring case
ASSERT_STRCASENE(str1,str2);	EXPECT_STRCASENE(str1,str2);	the two C strings have different content, ignoring case

STREQ 和 STRNE 同时支持 char*和 wchar_t*类型的，STRCASEEQ 和 STRCASENE 只接收 char*。

2.4 浮点数对比

（1） 默认两数差值在 4ULP 内近似相等：

Fatal assertion	Nonfatal assertion	Verifies
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

（2） 自定义精度

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_NEAR(val1, val2, abs_error);	EXPECT_NEAR(val1, val2, abs_error);	the difference between val1 and val2 doesn't exceed the given absolute error

例如：

```
ASSERT_NEAR(-1.0f, -1.1f, 0.2f);
ASSERT_NEAR(2.0f, 3.0f, 1.0f);
```

2.5 成功失败断言

（1） 返回成功：SUCCEED( )

（2） 返回失败：

Fatal assertion	Nonfatal assertion
FAIL();	ADD_FAILURE();

使用 FAIL()宏标记致命错误（同 ASSERT_*），ADD_FAILURE()宏标记非致命错误（同 EXPECT_*）。

例如：

```
if (check)
{
    ADD_FAILURE() << "Sorry"; // None Fatal Asserton, 继续往下执行。

    FAIL(); // Fatal Assertion, 不往下执行该案例。
}
SUCCEED();
```

2.6 类型对比断言

::testing::StaticAssertTypeEq<T, T>()，编译期类型检查，该断言针对模板

例如：

```
template <typename T>
class Foo {
public:
    void Bar() { ::testing::StaticAssertTypeEq<int, T>(); }
};

void test()
{
    Foo <bool> foo;
    foo.Bar();//实例化后报错
}
```

2.7 异常断言

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_THROW(statement, exception_type);	EXPECT_THROW(statement, exception_type);	statement throws an exception of the given type
ASSERT_ANY_THROW(statement);	EXPECT_ANY_THROW(statement);	statement throws an exception of any type
ASSERT_NO_THROW(statement);	EXPECT_NO_THROW(statement);	statement doesn't throw any exception

例如：

```
void ThrowException(int n) {
    switch (n) {
    case 0:
        throw 0;
    case 1:
        throw "const char*";
    case 2:
        throw 1.1f;
    case 3:
        return;
    }
}

TEST(ThrowException, Check)
{
    EXPECT_THROW(ThrowException(0), int);//返回int类型异常
    EXPECT_THROW(ThrowException(1), const char*);
    ASSERT_ANY_THROW(ThrowException(2)); //返回任意异常
    ASSERT_NO_THROW(ThrowException(3)); //无异常
}
```

2.8 谓语断言

在使用 EXPECT/ASSERT_TRUE/FALSE 时，断言失败时只能显示 TRUE 或 FALSE。为了能够输出更加详细的信息，gtest 中提供了三种方法：

（1）如下宏能够输出传入的参数是什么，以便失败后好跟踪。

Fatal assertion	Nonfatal assertion	Verifies
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

宏的第一个参数是 bool 函数，后面的一至多个参数是 bool 函数的变量，目前，GTest 最多支持 5 个参数的版本 ASSERT/EXPECT_PRED5 宏。

例如：

相比于 EXPECT_TRUE，EXPECT_PRED2 在断言失败时，输出了具体的 a,b 参数值。

```cpp
bool compare(int a, int b)
{
    return a > b;
}

TEST(Compare, demo)
{
    int a = 1;
    int b = 6;
    EXPECT_TRUE(compare(a,b));
    EXPECT_PRED2(compare,a,b);
}
```

```
[ RUN      ] Compare.demo
D:\UnitTest\dcaxFile\dcaxFileSystemTest.cpp(12): error: Value of: compare(a,b)
  Actual: false
Expected: true
D:\UnitTest\dcaxFile\dcaxFileSystemTest.cpp(13): error: compare(a, b) evaluates to false, where
a evaluates to 1
b evaluates to 6
[  FAILED  ] Compare.demo (2 ms)
```

（2）在 bool 断言中提供更加详细的信息，可以写一个断言函数返回的是断言结果对象而不是 bool 变量，并断言结果对象中使用<<操作符插入消息。

例如：

```cpp
::testing::AssertionResult IsEven(int n) {
        if ((n % 2) == 0)
            return ::testing::AssertionSuccess();
        else
            return ::testing::AssertionFailure() << n << " is odd";
}

TEST(IsEven, int)
{
    EXPECT_TRUE(IsEven(5));
    EXPECT_TRUE(IsEven(3));
}
```

```
D:\UnitTest\dcaxFile\dcaxFileSystemTest.cpp(20): error: Value of: IsEven(5)
  Actual: false (5 is odd)
Expected: true
D:\UnitTest\dcaxFile\dcaxFileSystemTest.cpp(21): error: Value of: IsEven(3)
  Actual: false (3 is odd)
Expected: true
[  FAILED  ] IsEven.int (1 ms)
```

（3）还可以通过断言格式化，自定义输出更详尽的信息

Fatal 断言	非 Fatal 断言	验证
ASSERT_PRED_FORMAT1(p_format1, val1);	EXPECT_PRED_FORMAT1(p_format1, val1);	p_format1(val1)成功
ASSERT_PRED_FORMAT2(p_format2, val1, val2);	EXPECT_PRED_FORMAT2(p_format2, val1, val2);	p_format2(val1, val2) 成功

例如：

```cpp
testing::AssertionResult AssertFoo(const char* m_expr, const char* n_expr, const char* k_expr,
                                   int m, int n, int k)
{
    if (Dec(m, n) == k)   //Dec函数实现m/n
        return testing::AssertionSuccess();
    testing::Message msg;
    msg << m_expr << " 除" << n_expr << "是: " << Dec(m, n) << " 而不是: " << k_expr;
    return testing::AssertionFailure(msg);
}

TEST(AssertFooTest, HandleFail)
{
    EXPECT_PRED_FORMAT3(AssertFoo,6,3,3);
}
```

```
[ RUN      ] AssertFooTest.HandleFail
D:\UnitTest\dcaxFile\dcaxFileSystemTest.cpp(20): error: 6 除3是: 2 而不是: 3
[  FAILED  ] AssertFooTest.HandleFail (2 ms)
```

2.9 子过程中使用断言

如下，当在子过程 Sub 中使用断言（ASSERT_EQ），在父过程中多次调用子过程（Sub）时，若断言失败，输出报告仅指向子过程（cpp(4)），无法区分是哪次调用出错。添加宏 SCOPED_TRACE，可添加标记位。

```
3  void Sub(int n) {
4      ASSERT_EQ(1, n);
5  }
6
7  TEST(SubTest, Test1) {
8      {
9          //SCOPED_TRACE("A");//SCOPED_TRACE宏用于标记位置
10         Sub(2);
11     }
12     Sub(3);
13 }
```

```
[ RUN      ] SubTest.Test1
D:\UnitTest\dcaxFile\dcaxFileSystemTest.cpp(4): error:       Expected: 1
To be equal to: n
      Which is: 2
[  FAILED  ] SubTest.Test1 (2 ms)
[----------] 1 test from SubTest (2 ms total)
```

```
[ RUN      ] SubTest.Test1
D:\UnitTest\dcaxFile\dcaxFileSystemTest.cpp(4): error:       Expected: 1
To be equal to: n
      Which is: 2
Google Test trace:
D:\UnitTest\dcaxFile\dcaxFileSystemTest.cpp(9): A
[  FAILED  ] SubTest.Test1 (1 ms)
[----------] 1 test from SubTest (1 ms total)
```

3. TEST_F 宏

当多种不同情况的测试 Case 中需要使用相同一份的测试数据时，采用 TEST 宏进行测试将会为不同的测试 case 各创建一份数据，而采用 TEST_F 宏将会共用一份可避免重复拷贝。

```
TEST_F(TestFixtureName,TestCaseName)
{
    //测试语句
}
```

- TestFixtureName 只能是定义的测试夹具名
- 第二个参数是测试名可随便取

3.1 test_fixture（）

（1）Test Fixtures 类继承于::testing::Test 类。

（2）在 Test Fixtures 类内部使用 public 或者 protected 描述其成员，以保证实际执行的测试子类可以使用类的成员变量。

（3）在构造函数或者继承于::testing::Test 类中的 SetUp()方法中，创建、初始化我们需要构造的数据。

（4）在析构函数或继承于::testing::Test 类中的 TearDown()方法中，实现资源释放。

```
class Path :public testing::Test
{

    void SetUp() override
    {
        m_p1 = sf::path(path1);
        m_p2 = sf::path(m_p1);
        m_p3 = sf::path(str);
        m_p4 = sf::path(path2);
    }

public:
    sf::path m_p;
    sf::path m_p1;
    sf::path m_p2;
    sf::path m_p3;
    sf::path m_p4;
};
```

例如：

```
TEST_F(Path, Clear)
{
    //void clear() noexcept;
    m_p1.clear();
    EXPECT_EQ("", m_p1.string());
}

TEST_F(Path,FileName)
{
    //bool has_filename() const; path filename() const;
    EXPECT_TRUE(m_p3.has_filename()&&m_p3.filename()=="log.txt");

    //path& remove_filename();
    m_p4.remove_filename();
    EXPECT_EQ("D:\\filetest\\", m_p4.string());

    //path& replace_filename();
    m_p3.replace_filename("test.txt");
```

3.2 testcase 事件

如 test_fixture(path)中，实现的是 SetUp（）方法和 TearDown（）方法。每个测试特例都要新建一个 test_fixture 对象，并在测试特例结束时销毁。

例如：

```
class TestCase :public testing::Test
{
public:

    void SetUp() {
        cout << "SetUp() is running" << endl;
    }

    void TearDown() {
        cout << "TearDown()" << endl;
    }
};
```

```
TEST_F(TestCase, Test1)
{

}

TEST_F(TestCase, Test2)
{

}
```

```
[----------] 2 tests from TestCase
[ RUN      ] TestCase.Test1
SetUp() is running
TearDown()
[       OK ] TestCase.Test1 (1 ms)
[ RUN      ] TestCase.Test2
SetUp() is running
TearDown()
[       OK ] TestCase.Test2 (0 ms)
[----------] 2 tests from TestCase (3 ms total)
```

3.3 testsuit 事件

testsuit 事件同样继承 testing::Test，实现的是 SetUpTestCase()和 TearDownTestCase()。在测试用例开始时执行，全部结束时才销毁。

例如：

```
class TestSuit :public testing::Test
{
public:

    static void SetUpTestCase()
    {
        cout << "SetUpTestCase()" << endl;
    }

    static void TearDownTestCase()
    {
        cout << "TearDownTestCase()" << endl;
    }
};
```

```
TEST_F(TestSuit, Test1)
{

}

TEST_F(TestSuit, Test2)
{

}
```

```
[----------] 2 tests from TestSuit
SetUpTestCase()
[ RUN      ] TestSuit.Test1
[       OK ] TestSuit.Test1 (0 ms)
[ RUN      ] TestSuit.Test2
[       OK ] TestSuit.Test2 (0 ms)
TearDownTestCase()
[----------] 2 tests from TestSuit (0 ms total)
```

3.4 全局事件

全局事件的类需继承 testing::Environment 类，实现里面的 SetUp()和 TearDown()方法，此外，还需在 main 函数中通过 testing::AddGlobalTestEnvironment 添加这个全局事件。

例如：

```
class TestGlobal :public testing::Environment
{
public:

    void SetUp() {
        cout << "Environment SetUP" << std::endl;
    }

    void TearDown() {
        cout << "Environment TearDown" << std::endl;
    }
};
```

```
int main(int argc, char* argv[])
{
    testing::AddGlobalTestEnvironment(new  TestGlobal);
    ::testing::InitGoogleTest(&argc, argv);
    RUN_ALL_TESTS();
    return 0;
}
```

```
[----------] Global test environment set-up.
Environment SetUP
[----------] 2 tests from Path
[ RUN      ] Path.PathStructure
[       OK ] Path.PathStructure (1 ms)
[ RUN      ] Path.PathOperate
[       OK ] Path.PathOperate (0 ms)
[----------] 2 tests from Path (2 ms total)

[----------] Global test environment tear-down
Environment TearDown
```

4. TEST_P 宏（值参数化）

有时为了完整的测试一个函数的正确性，我们需要在给定各种不同输入的情况下，测试函数的输出是否正确，但是这种方式重复且工作量大，如下：

```
TEST(IsPrimeTest, Positive)
{
    //IsPrime判断传入的整形变量是否为素数
    EXPECT_FALSE(IsPrime(0));
    EXPECT_FALSE(IsPrime(1));
    EXPECT_TRUE(IsPrime(2));
    EXPECT_TRUE(IsPrime(3));
}
```

通过类 TestWithParam 可将值参数化，减少代码量，使用类 TestWithParam 必须使用宏 TEST_P 来编写具体的测试用例。同 TEST_F:

```
TEST_P(TestFixtureName,TestCaseName)
{
    //测试语句
}
```

- TestFixtureName 只能是定义的测试夹具名
- 第二个参数是测试名可随便取

4.1 TestWithParm

类 testing::TestWithParm 的基础使用如下，分为三步:

(1) 定义 test fixture：test fixture 继承自类 testing::TestWithParam，继承时需要注明参数类型。

(2) 使用宏 TEST_P 编写具体的测试用例。

(3) 使用 INSTANTIATE_TEST_CASE_P(TestCaseName,TestFixtureName,Generator)明确待使用的参数。

例如：

```
class IsPrimeTest : public testing::TestWithParam<int> {};
TEST_P(IsPrimeTest, basicTest)
{
    int n;
    n = GetParam();
    EXPECT_FALSE(IsPrime(n));
}
INSTANTIATE_TEST_CASE_P(IsPrime, IsPrimeTest, testing::Values(0, 1, 2, 3));
```

对于 GetParam( )函数，它可以定义在 test fixture 中、也可以定义在 TEST_P 中。GetParam( )从 testing::Values( )中获取数据，数据类型和 test fixture 定义时的类型一致：示例中是 int。

4.2 Generator

Generator 相当于参数生成器，上面的例子使用了 testing::Values 函数，gtest 中提供的函数还有：

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)	这个比较强悍，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，如果你的系统确实支持，而gtest判断错误的话，你可以重新定义宏GTEST_HAS_TR1_TUPLE=1。

5. 类型参数化

当需要测试的数据具有多种类型或同一接口具有多个实现时，可将类型参数化后进行测试。

5.1 TYPE_TEST

此种情况下，已知所有需要测试的类型。具体实现如下：

（1） 定义一个 test_fixture，继承自类 testing::Test。

（2） 定义需要测试到的具体数据类型，typedef::testing::Types< 数据类型列表>TypeList。

（3） 使用宏 TYPED_TEST_CASE(TestFixtureName,TypeList)。同 TEST_F, TestFixtureName 必须与夹具名相同。

（4） 使用 TYPED_TEST(TestFixtureName, TestName) 编写一个类型测试。

例如：

测试以下重载接口，已知数据类型有 int、string、double：

```
class  Item {
public:
    int  addItems(int  area, vector < int >  value);
    int  addItems(int  area, vector < std::string >  value);
    int  addItems(int  area, vector < double >  value);

};
```

```
template < typename T >
class  ItemTest : public  ::testing::Test {
public:
    ItemTest() {
        if (typeid(T) == typeid(int)) {
            _value.push_back(1);
            _value.push_back(2);
            _value.push_back(3);
        }
        else   if (typeid(T) == typeid(string)) {
            _value.push_back(string(" Hello "));
            _value.push_back(string(" world "));
        }
    }
    std::vector < boost::any >  _value;
    typedef std::vector < T >  List;
};
```

```
using  testing::Types;

typedef ::testing::Types <   int, string, double >  MyTypes;
TYPED_TEST_CASE(ItemTest, MyTypes);

TYPED_TEST(ItemTest, test01)
{
    Item t;
    typename TestFixture::List values;
    for (int i = 0; i < this->_value.size(); ++i) {
        values.push_back(any_cast <TypeParam> (this->_value[i]));
    }
    EXPECT_TRUE(t.addItems(0, values));
}
```

5.2 TYPED_TEST_P

当需要参数化的类型未知时，实现如下：

（1） 同上，定义一个 test_fixture，继承自类 testing::Test。

（2） 使用宏 TYPED_TEST_CASE_P(TestFixtureName) 声明测试用例。

（3） 使用宏 TYPED_TEST_P(TestFixtureName,TestName)编写测试案例。

（4） 相比 5.1 还需使用宏 REGISTER_TYPED_TEST_CASE_P(TestFixtureName, TestName1,TestName2…)注册实例。

例如：针对类 Queue

```
template <class T>
class Queue {
public:
    Queue();
    void Enqueue(const T& element);
    T* Dequeue(); // Returns NULL if the queue is empty.
    size_t size() const;
    ...
};
```

通过以下代码实现了未知类型的测试模板，可将以下测试代码写进一个.h 中，以供后续具体实例化时调用：

```
template <class T>
Queue<T>* CreateQueue();

template <class T>
class QueueTest :public testing::Test
{
protected:
    QueueTest() :queue(CreateQueue<T>()) {}
    virtual ~QueueTest() { delete queue; }
    Queue<T>* const queue;
};//test_fixture

TYPED_TEST_CASE_P(QueueTest); //声明测试用例

TYPED_TEST_P(QueueTest, DefaultConstructor)
{
    EXPECT_EQ(0u, this->queue->Size());
}//测试案例

REGISTER_TYPED_TEST_CASE_P(QueueTest, DefaultConstructor);//注册实例
```

假设要测试 int 和 char 类型可在.cpp 中添加：

```
using testing::Types;
typedef Types<int, char> Implementations;
INSTANTIATE_TYPED_TEST_CASE_P(QueueInt_Char, QueueTest, Implementations);
```

6. 死亡测试

在测试过程中，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，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 会优先运行死亡测试案例，在写案例时，TEST 的第一个参数必须加 DeathTest 后缀。

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*

（1） statement 是被测试的逻辑表达式，它可以是个函数，可以是个对象的方法，也可以是几个表达式的组合。

（2） predicate 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。 常用的有：

1）testing::ExitedWithCode(exit_code) 如果程序正常退出并且退出码与 exit_code 相同则返回 true

2）testing::KilledBySignal(signal_number) // Windows 下不支持 如果程序被 signal_number 信号 kill 的话就返回 true

（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。POSIX 系统和 Windows 下，gtest 使用不同的正则表达式语法。根据宏区分

1）POSIX 风格：GTEST_USES_POSIX_RE = 1

2）Simple 风格：GTEST_USES_SIMPLE_RE=1

POSIX 使用的更复杂，Windows 的部分如下：

	matches any literal character c
\\d	matches any decimal digit
\\D	matches any character that's not a decimal digit
\\f	matches \f
\\n	matches \n
\\r	matches \r
\\s	matches any ASCII whitespace, including \n
\\S	matches any character that's not a whitespace
\\t	matches \t
\\v	matches \v
\\w	matches any letter, _, or decimal digit
\\W	matches any character that \\w doesn't match
\\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

7. 特殊测试

7.1 测试筛选

启动程序时提供命令行参数 --gtest_list_tests 可以显示所有测试的完整名称，以便进行测试过滤。命令行参数--gtest_filter=<匹配模式>可筛选测试，只进行部分测试。

匹配模式语法：

```
[<正匹配单元> { ':' < 正匹配单元 > }]['-' < 负匹配单元 > { ':' < 负匹配单元 > }]
```

匹配单元的特殊字符：

* 匹配 0 到任意个任意字符（默认为*）

? 匹配 1 个任意字符

匹配模式将筛选出所有任意正匹配单元和所有负匹配之外的单元以进行测试。

例如：

`*Null*:*Constructor*` 测试名或测试类/分类中含Null或Constructor的所有测试

`-*DeathTest.*` 运行所有非崩溃测试

`FooTest.*-FooTest.Bar` FooTest测试类/分类中除Bar外的所有测试

7.2 禁止测试

为测试名添加 DISABLED_前缀能暂时禁用该测试

```
TEST(分类名, DISABLED_测试名)
{
    //测试代码
}
```

7.3 重复测试

命令行参数： --gtest_repeat=<>重复数，可进行重复测试。

8. 输出文件

测试结果一般情况下在控制台输出，启动时添加以下命令行参数或添加环境变量可以增加记录到文件的输出。

8.1 xml

(1) 命令行参数：--gtest_out="xml:<文件路径>"

(2) 环境变量：set GTEST_OUTPUT=xml:<文件路径>

8.2 json

(1) 命令行参数：--gtest_out="json:<文件路径>"

(2) 环境变量：set GTEST_OUTPUT=json:<文件路径>

也可省略文件路径：默认为 test_detail.xml 或 test_detail.json

--gtest_output = "xml"
--gtest_output = "json"

8.3 自定义记录

::testing::Test::RecordProperty 记录属性数据到 xml 和 json.

```
RecordProperty(const string& key, int value);
RecordProperty(const string& key, const string& value);
```

二. Google mock

1. 概述

Gmock 就是一个强大的模拟接口的工具，它往往和 GTest 结合在一起使用。使用时同 gtest 一样，需要：

（1） 编译生成 gmock.lib、gmock_main.lib（下载的 gtest 内含 gmock）。

（2） 包含 include 文件夹，并链接 gmock.lib 或 gmock_main.lib

（3） 添加#include”gmock.h”

（4） 依赖 gmock.lib 需要编写如下 main 函数

```
int main(int argc, char* argv[])
{

    ::testing::InitGoogleMock(&argc, argv);
    RUN_ALL_TESTS();
    return 0;
}
```

2. 典型使用

使用 Mock 类的一般流程如下:

(1) 初始化一个 Mock 类，继承自要模拟的类。

(2) 创建 Mock 类对象。

(3) 在对象上设置自定义的预期行为。

例如：

```cpp
class FooInterface {
public:
    virtual ~FooInterface() {}

public:
    virtual std::string getArbitraryString() = 0;
};

//初始化Mock类
class MockFoo : public FooInterface {
public:
    MOCK_METHOD0(getArbitraryString, std::string());
};

using ::testing::Return;
TEST(FooInterface,Mock)
{
string value = "Hello World!";
//创建Mock对象
MockFoo mockFoo;

//设置预期行为
EXPECT_CALL(mockFoo, getArbitraryString()).Times(1).
WillOnce(Return(value));
string returnValue = mockFoo.getArbitraryString();
}
```

Gmock 需要根据形参的个数来使用 MOCK_METHOD，这里 getArbitraryString 没有形参，所以使用 MOCK_METHOD0 来定义，若有一个形参则使用 MOCK_METHOD1，以此类推。

可参考：转一篇小亮同学的 google mock 分享 - welkinwalker - 博客园 (cnblogs.com)