零声教育 Mark 老师 QQ: 2548898954

googletest 简介

什么是一个好的测试?

1. Tests should be *independent* and *repeatable*. It's a pain to debug a test that succeeds or fails as a result of other tests. googletest isolates the tests by running each of them on a different object. When a test fails, googletest allows you to run it in isolation for quick debugging.

   测试应该是**独立的和可重复的。**调试一个由于其他测试而成功或失败的测试是一件痛苦的事情。*googletest* 通过在不同的对象上运行测试来隔离测试。当测试失败时，*googletest* 允许您单独运行它以快速调试。

2. Tests should be well *organized* and reflect the structure of the tested code. googletest groups related tests into test suites that can share data and subroutines. This common pattern is easy to recognize and makes tests easy to maintain. Such consistency is especially helpful when people switch projects and start to work on a new code base.

   测试应该很好地"组织"，并反映出**测试代码的结构。***googletest* 将相关测试分组到可以共享数据和子例程的测试套件中。这种通用模式很容易识别，并使测试易于维护。当人们切换项目并开始在新的代码库上工作时，这种一致性尤其有用。

3. Tests should be *portable* and *reusable*. Google has a lot of code that is platform-neutral; its tests should also be platform-neutral. googletest works on different OSes, with different compilers, with or without exceptions, so googletest tests can work with a variety of configurations.

   测试应该是**可移植的和可重用的。**谷歌有许多与平台无关的代码；它的测试也应该是平台中立的。*googletest* 可以在不同的操作系统上工作，使用不同的编译器，所以 *googletest* 测试可以在多种配置下工作。

4. When tests fail, they should provide as much *information* about the problem as possible. googletest doesn't stop at the first test failure. Instead, it only stops the current test and continues with the next. You can also set up tests that report non-fatal failures after which the current test continues. Thus, you can detect and fix multiple bugs in a single run-edit-compile cycle.

   当测试失败时，他们应该**提供尽可能多的关于问题的信息。***googletest* 不会在第一次测试失败时停止。相反，它只停止当前的测试并继续下一个测试。还可以设置报告非致命失败的测试，在此之后当前测试将继续进行。因此，您可以在一个运行-编辑-编译周期中检测和修复多个错误。

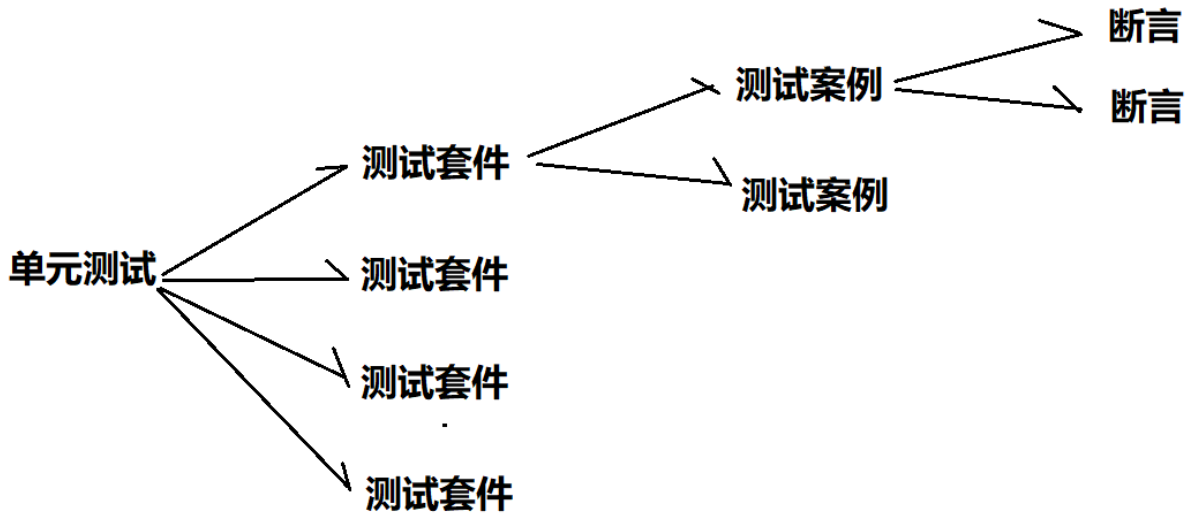
5. The testing framework should liberate test writers from housekeeping chores and let them focus on the test *content*. googletest automatically keeps track of all tests defined, and doesn't require the user to enumerate them in order to run them.

   测试框架应该将测试编写者从日常琐事中解放出来，让他们专注于测试"内容"。*googletest* **自动跟踪所有定义的测试**，并且不要求用户为了运行它们而枚举它们。

6. Tests should be *fast*. With googletest, you can reuse shared resources across tests and pay for the set-up/tear-down only once, without making tests depend on each other.

   测试应该是快速的。使用 *googletest*，您可以在**测试之间重用共享资源**，并且只需要为设置/拆除支付一次费用，而无需使测试彼此依赖。

测试层次关系



环境准备

下载

```
git clone https://github.com/google/googletest.git
# 或者
wget https://github.com/google/googletest/releases/tag/release-1.11.0
```

安装

```
cd googletest
cmake CMakeLists.txt
make
sudo make install
```

重要文件

googletest

```
# 头文件
gtest/gtest.h

# 不带 main 静态库
libgtest.a

# 带 main 静态库
libgtest_main.a
```

> 当不想写 `main` 函数的时候，可以直接引入 *libgtest_main.a*;
>
> ```
> g++ sample.cc -o sample -lgtest -lgtest_main -lpthread
> g++ sample.cc -o sample -lgmock -lgmock_main -lpthread
> ```
>
> 否则
>
> ```
> g++ sample.cc -o sample -lgtest -lpthread
> ```

googlemock

```
# 头文件
gmock/gmock.h

# 不带 main 静态库
libgmock.a

# 带 main 静态库
libgmock_main.a
```

断言

> 断言成对出现，它们测试相同的东西，但对当前函数有不同的影响。`ASSERT_*`版本在失败时产生致命失败，并中止当前测试案例。`EXPECT_*`版本生成非致命失败，它不会中止当前函数。通常首选`EXPECT_*`，因为它们允许在测试中报告一个以上的失败。但是，如果在有问题的断言失败时继续没有意义，则应该使用`ASSERT_*`。
>
> 所有断言宏都支持输出流，也就是当出现错误的时候，我们可以通过流输出更详细的信息；注意**编码问题**，经流输出的信息会**自动转换为 UTF-8**；

```
EXPECT_TRUE(my_condition) << "My condition is not true";
```

明确指定成功或者失败

> 有时候我们测试案例当中的条件太复杂，不能使用断言，那么自己写判断语句；自己返回 成功或者失败；‘
>
> `SUCCEED()` 或者 `FAIL()`

布尔条件

> `EXPECT_TRUE( condition )`
> `ASSERT_TRUE( condition )`
>
> `EXPECT_FALSE( condition )`
> `ASSERT_FALSE( condition )`

二元比较

`val1 = val2`:

> `EXPECT_EQ( val1 , val2 )`
> `ASSERT_EQ( val1 , val2 )`

`val1 != val2`:

> `EXPECT_NE( val1 , val2 )`
> `ASSERT_NE( val1 , val2 )`
>
> 注意：比较空指针的时候；
>
> 使用 `EXPECT_NE( ptr , nullptr)` 而不是 `EXPECT_NE( ptr , NULL)`。

`val1 < val2`:

> `EXPECT_LT( val1 , val2 )`
> `ASSERT_LT( val1 , val2 )`

`val1 <= val2`:

> `EXPECT_LE( val1 , val2 )`
> `ASSERT_LE( val1 , val2 )`

`val1 > val2`:

> `EXPECT_GT( val1 , val2 )`
> `ASSERT_GT( val1 , val2 )`

`val1 >= val2`:

> `EXPECT_GE( val1 , val2 )`
> `ASSERT_GE( val1 , val2 )`

谓词断言

> 谓词断言能比 `EXPECT_TRUE` 提供更详细的错误消息；

`EXPECT_PRED1( pred , val1 )` \
`EXPECT_PRED2( pred , val1 , val2 )` \
`EXPECT_PRED3( pred , val1 , val2 , val3 )` \
`EXPECT_PRED4( pred , val1 , val2 , val3 , val4 )` \
`EXPECT_PRED5( pred , val1 , val2 , val3 , val4 , val5 )`

`ASSERT_PRED1( pred , val1 ) \`
`ASSERT_PRED2( pred , val1 , val2 ) \`
`ASSERT_PRED3( pred , val1 , val2 , val3 ) \`
`ASSERT_PRED4( pred , val1 , val2 , val3 , val4 ) \`
`ASSERT_PRED5( pred , val1 , val2 , val3 , val4 , val5 )`

```
// Returns true if m and n have no common divisors except 1.
bool MutuallyPrime(int m, int n) { ... }
...
const int a = 3;
const int b = 4;
const int c = 10;
...
EXPECT_PRED2(MutuallyPrime, a, b);  // Succeeds
EXPECT_PRED2(MutuallyPrime, b, c);  // Fails
```

能得到错误信息：

```
MutuallyPrime(b, c) is false, where
b is 4
c is 10
```

googletest

函数测试以及类测试

```
#define TEST(test_suite_name,test_name)
```

test fixture(测试夹具)

> 用相同的数据配置来测试多个测试案例。

```
// 定义类型，继承自 testing::Test
class TestFixtureSmpl : public testing::Test {
protected:
    void SetUp() {} // 测试夹具测试前调用的函数 -- 做初始化的工作
    void TearDown() {} // 测试夹具测试后调用的函数 -- 做清理的工作
};

// 需要在 TEST_F 中书写测试用例
#define TEST_F(test_fixture,test_name)

// 如果需要复用测试夹具，只需要继承自 TestFixtureSmpl
class TestFixtureSmpl_v2 : public TestFixtureSmpl {
};
```

类型参数化

基础

> 有时候相同的接口，有多个实现，下面是复用测试代码流程；
>
> 复用测试案例

```
using testing::Test;
using testing::Types;
// 先申明测试夹具
template <class T>
class TestFixtureSmpl : public testing::Test {
protected:
    void SetUp() {} // 测试夹具测试前调用的函数 -- 做初始化的工作
    void TearDown() {} // 测试夹具测试后调用的函数 -- 做清理的工作
};
// 枚举测试类型
typedef Types<Class1, Class2, class3, ...> Implementations;
// #define TYPED_TEST_SUITE(CaseName,Types,__VA_ARGS__...)
// 注意 casename 一定要与测试夹具的名字一致
TYPED_TEST_SUITE(TestFixtureSmpl, Implementations);
// #define TYPED_TEST(CaseName,TestName)
// 开始测试， CaseName 要与 TYPED_TEST_SUITE 一致
TYPED_TEST(TestFixtureSmpl, TestName)
```

进阶

> 有时候你写了某个接口，期望其他人实现它，你可能想写一系列测试，确保其他人的实现满足你的测试；

```
// 首先声明测试类型参数化(_P 是 parameterized or pattern)
// #define TYPED_TEST_SUITE_P(SuiteName)
TYPED_TEST_SUITE_P(TestFixtureSmpl);
// 书写测试, suiteName 与上面一致
// #define TYPED_TEST_P(SuiteName,TestName)
TYPED_TEST_P(TestFixtureSmpl,TestName)
// 枚举所有测试
// #define REGISTER_TYPED_TEST_SUITE_P(SuiteName,__VA_ARGS__...)
REGISTER_TYPED_TEST_SUITE_P(TestFixtureSmpl,
   TestName1,TestName2,...)
// 上面定义的是抽象测试类型
// 其他人实现功能后，开始测试，假如实现了 OnTheFlyPrimeTable 和
PreCalculatedPrimeTable
typedef Types<OnTheFlyPrimeTable, PreCalculatedPrimeTable>
    PrimeTableImplementations;
// #define
INSTANTIATE_TYPED_TEST_SUITE_P(Prefix,SuiteName,Types,__VA_ARGS__...)
INSTANTIATE_TYPED_TEST_SUITE_P(
    instance_name,
    testcase,
    typelist...)
```

事件

> 可以通过 *googletest* 的事件机制，在测试前后进行埋点处理；
>
> 模板模式

```
// The interface for tracing execution of tests. The methods are organized
in
// the order the corresponding events are fired.
class TestEventListener {
 public:
  virtual ~TestEventListener() {}

  // Fired before any test activity starts.
  virtual void OnTestProgramStart(const UnitTest& unit_test) = 0;

  // Fired before each iteration of tests starts.  There may be more than
  // one iteration if GTEST_FLAG(repeat) is set. iteration is the iteration
  // index, starting from 0.
  virtual void OnTestIterationStart(const UnitTest& unit_test,
                                    int iteration) = 0;
  // Fired before environment set-up for each iteration of tests starts.
  virtual void OnEnvironmentsSetUpStart(const UnitTest& unit_test) = 0;

  // Fired after environment set-up for each iteration of tests ends.
  virtual void OnEnvironmentsSetUpEnd(const UnitTest& unit_test) = 0;

  // Fired before the test suite starts.
  virtual void OnTestSuiteStart(const TestSuite& /*test_suite*/) {}

  //  Legacy API is deprecated but still available
#ifndef GTEST_REMOVE_LEGACY_TEST_CASEAPI_
  virtual void OnTestCaseStart(const TestCase& /*test_case*/) {}
#endif  //  GTEST_REMOVE_LEGACY_TEST_CASEAPI_

  // Fired before the test starts.
  virtual void OnTestStart(const TestInfo& test_info) = 0;

  // Fired after a failed assertion or a SUCCEED() invocation.
  // If you want to throw an exception from this function to skip to the
next
  // TEST, it must be AssertionException defined above, or inherited from
it.
  virtual void OnTestPartResult(const TestPartResult& test_part_result) = 0;

  // Fired after the test ends.
  virtual void OnTestEnd(const TestInfo& test_info) = 0;

  // Fired after the test suite ends.
  virtual void OnTestSuiteEnd(const TestSuite& /*test_suite*/) {}

//  Legacy API is deprecated but still available
#ifndef GTEST_REMOVE_LEGACY_TEST_CASEAPI_
  virtual void OnTestCaseEnd(const TestCase& /*test_case*/) {}
#endif  //  GTEST_REMOVE_LEGACY_TEST_CASEAPI_

  // Fired before environment tear-down for each iteration of tests starts.
```

```
  virtual void OnEnvironmentsTearDownStart(const UnitTest& unit_test) = 0;

  // Fired after environment tear-down for each iteration of tests ends.
  virtual void OnEnvironmentsTearDownEnd(const UnitTest& unit_test) = 0;

  // Fired after each iteration of tests finishes.
  virtual void OnTestIterationEnd(const UnitTest& unit_test,
                                  int iteration) = 0;

  // Fired after all test activities have ended.
  virtual void OnTestProgramEnd(const UnitTest& unit_test) = 0;
};
```

内存泄露

> 怎么产生？1. 忘记释放了；2. 因为逻辑bug，跳过了释放流程；
>
> `new` 是 *c++* 中的操作符；
>
> 1. 调用 `operator new` 分配内存；
> 2. 调用构造函数在步骤 *1* 返回的内存地址生成类对象；
>
> 可以通过重载 `new` 来修改 *1* 的功能；
>
> `delete` 与 `new` 类似；只是是先调用析构函数，再释放内存；

```
// 重载操作符 new 和 delete，接着用类的静态成员来统计调用 new 和 delete的次数
class CLeakMem {
public:
  // ...
  void* operator new(size_t allocation_size) {
    allocated_++;
    return malloc(allocation_size);
  }
  void operator delete(void* block, size_t /* allocation_size */) {
    allocated_--;
    free(block);
  }
private:
  static int allocated_;
};
int CLeakMem::allocated_ = 0;
```

检测

```
class LeakChecker : public EmptyTestEventListener {
 private:
  // Called before a test starts.
  void OnTestStart(const TestInfo& /* test_info */) override {
    initially_allocated_ = CLeakMem::allocated();
  }

  // Called after a test ends.
  void OnTestEnd(const TestInfo& /* test_info */) override {
    int difference = CLeakMem::allocated() - initially_allocated_;

    // You can generate a failure in any event handler except
```

```
    // OnTestPartResult. Just use an appropriate Google Test assertion to do
    // it.
    EXPECT_LE(difference, 0) << "Leaked " << difference << " unit(s) of
class!";
  }

  int initially_allocated_;
};
```

googlemock

> 当你写一个原型或测试，往往不能完全的依赖真实对象。一个 *mock* 对象实现与一个真实对象相同的接口，但让你在运行时指定它时，如何使用？它应该做什么？（哪些方法将被调用？什么顺序？多少次？有什么参数？会返回什么？等）
>
> 可以模拟检查它自己和调用者之间的交互；
>
> mock 用于创建模拟类和使用它们；
>
> - 使用一些简单的宏描述你想要模拟的接口，他们将扩展到你的 *mock* 类的实现；
> - 创建一些模拟对象，并使用直观的语法指定其期望和行为；
> - 练习使用模拟对象的代码。 *googlemock* 会在出现任何违反期望的情况时立即处理。

注意

> *googlemock* 依赖 *googletest*；调用 `InitGoogleMock` 时会自动调用 `InitGoogleTest`；

什么时候使用?

- 测试很慢，依赖于太多的库或使用昂贵的资源；
- 测试脆弱，使用的一些资源是不可靠的（例如网络）；某个功能由多个网络交互构成
- 测试代码如何处理失败（例如，文件校验和错误），但不容易造成；
- 确保模块以正确的方式与其他模块交互，但是很难观察到交互；因此你希望看到观察行动结束时的副作用；
- 想模拟出复杂的依赖；

编写模拟类

```
#include "gmock/gmock.h"  // Brings in Google Mock.
class Turtle {
public:
    int GetX() const;
}
class MockTurtle : public Turtle {
 public:
  ...
  MOCK_METHOD0(PenUp, void());
  MOCK_METHOD0(PenDown, void());
  MOCK_METHOD1(Forward, void(int distance));
  MOCK_METHOD1(Turn, void(int degrees));
  MOCK_METHOD2(GoTo, void(int x, int y));
  MOCK_CONST_METHOD0(GetX, int());
  MOCK_CONST_METHOD0(GetY, int());
};
```

设置期望

通用语法

```
EXPECT_CALL(mock_object, method(matchers))
    .Times(cardinality)
    .WillOnce(action)
    .WillRepeatedly(action);
// 第二步
EXPECT_CALL(mock_object, method(matchers))
    .Times(2)
    .WillOnce(t1)
    .WillRepeatedly(action);

// 第三步:
mock_object obj;
t1 = obj.func1();
t2 = obj.func2();
```

matchers（期待参数）

```
EXPECT_CALL(turtle, Forward(100));
```

cardinality（调用次数）

```
// turtle::Forward 将预期调用1次
EXPECT_CALL(turtle, Forward(100)).Times(1);
// turtle::Forward 将预期调用至少1次
EXPECT_CALL(turtle, Forward(100)).Times(AtLeast(1))
```

action（满足期望做什么）

```
using ::testing::Return;
// ...
// GetX 第一次调用返回100，第二次调用返回200，第三次返回300
EXPECT_CALL(turtle, GetX())
    .Times(3)
    .WillOnce(Return(100))
    .WillOnce(Return(200))
    .WillOnce(Return(300));

// GetX 第一次调用返回100，第二次调用返回200，第三次返回0
EXPECT_CALL(turtle, GetX())
    .Times(3)
    .WillOnce(Return(100))
    .WillOnce(Return(200));

// GetX 将会返回4次100; WillRepeatedly 中的表达式只会计算一次
int n = 100;
EXPECT_CALL(turtle, GetX())
.Times(4)
.WillRepeatedly(Return(n++));

// #2 将会覆盖 #1;调用第三次将会报错
EXPECT_CALL(turtle, Forward(_));  // #1
```

```
EXPECT_CALL(turtle, Forward(10))  // #2
    .Times(2);

// 将严格按照 PenDown, Forward, PenUp 调用顺序检查
EXPECT_CALL(turtle, PenDown());
EXPECT_CALL(turtle, Forward(100));
EXPECT_CALL(turtle, PenUp());
```

例子

```
class FooInterface {
public:
    virtual ~FooInterface() {}
    virtual std::string getArbitraryString() = 0;

    virtual int getPosition() = 0;
};

class MockFoo : public FooInterface {
public:
    MOCK_METHOD0(getArbitraryString, std::string());
    MOCK_METHOD0(getPosition, int());
};
```

```

#include "stdafx.h"
using namespace seamless;
using namespace std;

using ::testing::Return;

int main(int argc, char** argv) {
    ::testing::InitGoogleMock(&argc, argv);
    int n = 100;
    string value = "Hello World!";

    MockFoo mockFoo;
    EXPECT_CALL(mockFoo, getArbitraryString())
        .Times(2)
        .WillOnce(Return(value))
        .WillOnce(Return(value));

    string returnValue = mockFoo.getArbitraryString();
    cout << "Returned Value: " << returnValue << endl;
    //在这里Times(2)意思是调用两次，但是下边只调用了一次，所以会报出异常
    EXPECT_CALL(mockFoo, getPosition())
        .Times(2)
        .WillRepeatedly(Return(n++));
    int val = mockFoo.getPosition();  // 100
    cout << "Returned Value: " << val << endl;
    //getPosition指定了调用两次，这里只调用了一次，所以运行结果显示出错
    return EXIT_SUCCESS;
}
```

操作流程

- 编写模拟类
- 设置期望
- 书写测试流程