奋飞的菜鸟的ChinaUnix博客

暂无签名

奋飞的菜鸟

博客访问： 14841
博文数量： 60
博客积分： 1200
博客等级： 中尉
技术积分： 535
用 户 组： 普通用户
注册时间： 2011-04-26 21:13

加关注 短消息
论坛 加好友

文章分类

全部博文（60）
计划安排（1）
Project（26）
C/C++（7）
算法与数据结构（5）
TCP/网络编程（6）
Linux（13）
未分配的博文（2）

文章存档

2012年（16）
2011年（44）

我的朋友


edaworld

laohuang

609854

athsonxy


pannian

最近访客


2007robo

转：google mock C++单元测试框架

2012-03-12 09:33:59

分类： C/C++

Content

Google Mock 入门概述什么是Mock?
Mock，更确切地说应该是Mock Object。它究竟是什么？它有什么作用？在这里，我也只能先说说我的理解。 比如当我们在单元测试、模块的接口测试时，当这个模块需要依赖另外一个/几个类，而这时这些个类还没有开发好（那名开发同学比较懒，呵呵），这时我们就可以定义了Mock对象来模拟那些类的行为。
说得更直白一些，就是自己实现一个假的依赖类，对这个类的方法你想要什么行为就可以有什么行为，你想让这个方法返回什么结果就可以返回怎么样的结果。
但这时很多同学往往会提出一个问题："那既然是我自己实现一个假的依赖类"，那和那些市面上的Mock框架有什么关系啊？
这个其实是这样的，这些个Mock框架可以帮助你比较方便、比较轻松地实现这些个假的依赖类。毕竟，如果你实现这么一个假的依赖类的时间花费过场的话，那我还不如等待那位懒惰的同学吧。

Google Mock概述
Google Mock（简称**gmock**）是Google在2008年推出的一套针对C++的Mock框架，它灵感取自于jMock、EasyMock、harcreat。它提供了以下这些特性：

- 轻松地创建mock类
- 支持丰富的匹配器（Matcher）和行为（Action）
- 支持有序、无序、部分有序的期望行为的定义
- 多平台的支持

参考文档

- 新人手册
- Cheat Sheet
- Cheat Sheet中文翻译
- Cookbook

Google Mock使用最简单的例子
我比较喜欢举例来说明这些个、那些个玩意，因此我们先来看看Google Mock就简单的用法和作用。

首先，那个懒惰的同学已经定义好了这么一个接口（万幸，他至少把接口定义好了）：

FooInterface.h

```
1. #ifndef FOOINTERFACE_H_
2. #define FOOINTERFACE_H_
3.
4. #include <string>
5.
6. namespace seamless {
7.
8. class FooInterface {
9. public:
10.     virtual ~FooInterface() {}
11.
12. public:
13.     virtual std::string getArbitraryString() = 0;
14. };
15.
16. } // namespace seamless
17.
18. #endif // FOOINTERFACE_H_
```

这里需要注意几点：

FooInterface的析构函数~FooInterface()必须是virtual的
在第13行，我们得把getArbitraryString定义为纯虚函数。其实getArbitraryString()也不一定得是纯虚函数，这点我们后面会提到.

现在我们用Google Mock来定义Mock类 **FooMock.h**

```
1. #ifndef MOCKFOO_H_
2. #define MOCKFOO_H_
3.
4. #include <gmock/gmock.h>
5. #include <string>
6. #include "FooInterface.h"
7.
8. namespace seamless {
9.
10. class MockFoo: public FooInterface {
11. public:
12.     MOCK_METHOD0(getArbitraryString, std::string());
13. };
14.
15. } // namespace seamless
16.
17. #endif // MOCKFOO_H_
```

我们稍微来解释一下这个Mock类的定义：

第10行我们的MockFoo类继承懒同学的FooInterface
第22行我们定义使用gmock中的一个宏（Macro）MOCK_METHOD0来定义MockFoo中的**getArbitraryString**。Google Mock是需要你根据不同的形参个数来使用不同的Mock Method，我这里getArbitraryString没有函数，就是MOCK_METHOD0了，同理，如果是一个形参，就是MOCK_METHOD1了，以此往下。

FooMain.cc

```
1. #include <cstdlib>
2. #include <gmock/gmock.h>
3. #include <gtest/gtest.h>
4. #include <iostream>
5. #include <string>
6.
```

```
#include "MockFoo.h"

using namespace seamless;
using namespace std;

using ::testing::Return;

int main(int argc, char** argv) {
        ::testing::InitGoogleMock(&argc, argv);

        string value = "Hello World!";
        MockFoo mockFoo;
        EXPECT_CALL(mockFoo, getArbitraryString()).Times(1).
                WillOnce(Return(value));
        string returnValue = mockFoo.getArbitraryString();
        cout << "Returned Value: " << returnValue << endl;

        return EXIT_SUCCESS;
}
```

最后我们运行编译，得到的结果如下：

```
Returned Value: Hello World!
```

在这里：

第15行，初始化一个Google Mock
第18行，声明一个MockFoo的对象：mockFoo
第19行，是为MockFoo的getArbitraryString()方法定义一个期望行为，其中Times(1)的意思是运行一次，WillOnce(Return(value))的意思是第一次运行时把value作为getArbitraryString()方法的返回值。

这就是我们最简单的使用Google Mock的例子了，使用起来的确比较简便吧。

典型的流程
通过上述的例子，已经可以看出使用Mock类的一般流程如下：

引入你要用到的Google Mock名称. 除宏或其它特别提到的之外所有Google Mock名称都位于*testing*命名空间之下.
建立模拟对象(Mock Objects).
可选的,设置模拟对象的默认动作.
在模拟对象上设置你的预期(它们怎样被调用,应该怎样回应?).

自定义方法/成员函数的期望行为
从上述的例子中可以看出，当我们针对懒同学的接口定义好了Mock类后，在单元测试/主程序中使用这个Mock类中的方法时最关键的就是对期望行为的定义。
对方法期望行为的定义的语法格式如下：

```
EXPECT_CALL(mock_object, method(matcher1, matcher2, ...))
    .With(multi_argument_matcher)
    .Times(cardinality)
    .InSequence(sequences)
    .After(expectations)
    .WillOnce(action)
    .WillRepeatedly(action)
    .RetiresOnSaturation();
```

解释一下这些参数（虽然很多我也没弄明白）：

第1行的mock_object就是你的Mock类的对象
第1行的method(matcher1, matcher2, ...)中的method就是你Mock类中的某个方法名，比如上述的getArbitraryString;而matcher（匹配器）的意思是定义方法参数的类型，我们待会详细介绍。
第3行的Times(cardinality)的意思是之前定义的method运行几次。至于cardinality的定义，我也会在后面详细介绍。
第4行的InSequence(sequences)的意思是定义这个方法被执行顺序（优先级），我会再后面举例说明。
第6行WillOnce(action)是定义一次调用时所产生的行为，比如定义该方法返回怎么样的值等等。

第7行WillRepeatedly(action)的意思是缺省/重复行为。

我稍微先举个例子来说明一下，后面有针对更为详细的说明：

```
1. EXPECT_CALL(mockTurtle, getX()).Times(testing::AtLeast(5)).
2.         WillOnce(testing::Return(100)).WillOnce(testing::Return(150)).
3.         WillRepeatedly(testing::Return(200))
```

这个期望行为的定义的意思是：

调用mockTurtle的getX()方法
这个方法会至少调用5次
第一次被调用时返回100
第2次被调用时返回150
从第3次被调用开始每次都返回200

Matcher（匹配器）
Matcher用于定义Mock类中的方法的形参的值（当然，如果你的方法不需要形参时，可以保持match为空。），它有以下几种类型：（更详细的介绍可以参见Google Mock Wiki上的Matcher介绍）
通配符

_	可以代表任意类型
A() or An()	可以是type类型的任意值

这里的_和*A*包括下面的那个匹配符都在Google Mock的*::testing*这个命名空间下，大家要用时需要先引入那个命名空间

一般比较

Eq(value) 或者 value	argument == value，method中的形参必须是value
Ge(value)	argument >= value，method中的形参必须大于等于value
Gt(value)	argument > value
Le(value)	argument <= value
Lt(value)	argument < value
Ne(value)	argument != value
IsNull()	method的形参必须是NULL指针
NotNull()	argument is a non-null pointer
Ref(variable)	形参是variable的引用
TypedEq(value)	形参的类型必须是type类型，而且值必须是value

浮点数的比较

DoubleEq(a_double)	形参是一个double类型，比如值近似于a_double，两个NaN是不相等的
FloatEq(a_float)	同上，只不过类型是float
NanSensitiveDoubleEq(a_double)	形参是一个double类型，比如值近似于a_double，两个NaN是相等的，这个是用户所希望的方式
NanSensitiveFloatEq(a_float)	同上，只不过形参是float

字符串匹配
这里的字符串即可以是C风格的字符串，也可以是C++风格的。

ContainsRegex(string)	形参匹配给定的正则表达式
EndsWith(suffix)	形参以suffix截尾

HasSubstr(string)	形参有string这个子串
MatchesRegex(string)	从第一个字符到最后一个字符都完全匹配给定的正则表达式.
StartsWith(prefix)	形参以prefix开始
StrCaseEq(string)	参数等于string，并且忽略大小写
StrCaseNe(string)	参数不是string，并且忽略大小写
StrEq(string)	参数等于string
StrNe(string)	参数不等于string

容器的匹配
很多STL的容器的比较都支持==这样的操作，对于这样的容器可以使用上述的**Eq(container)**来比较。但如果你想写得更为灵活，可以使用下面的这些容器匹配方法：

Contains(e)	在method的形参中，只要有其中一个元素等于e
Each(e)	参数各个元素都等于e
ElementsAre(e0, e1, …, en)	形参有n+1的元素，并且挨个匹配
ElementsAreArray(array) 或者 ElementsAreArray(array, count)	和ElementsAre()类似，除了预期值/匹配器来源于一个C风格数组
ContainerEq(container)	类型Eq(container)，就是输出结果有点不一样，这里输出结果会带上哪些个元素不被包含在另一个容器中
Pointwise(m, container)	

上述的一些匹配器都比较简单，我就随便打包举几最简单的例子演示一下吧：　　我稍微修改一下之前的**Foo.h**和**MockFoo.h**， **MockFoo.h** 增加了2个方法

```
#ifndef MOCKFOO_H_
#define MOCKFOO_H_

#include <gmock/gmock.h>
#include <string>
#include <vector>
#include "FooInterface.h"

namespace seamless {

class MockFoo: public FooInterface {
public:
        MOCK_METHOD0(getArbitraryString, std::string());
        MOCK_METHOD1(setValue, void(std::string& value));
        MOCK_METHOD2(setDoubleValues, void(int x, int y));
};

} // namespace seamless

#endif // MOCKFOO_H_
```

FooMain.h

```
#include <cstdlib>
#include <gmock/gmock.h>
#include <iostream>
#include <string>

#include "MockFoo.h"

using namespace seamless;
using namespace std;

using ::testing::Assign;
using ::testing::Eq;
using ::testing::Ge;
```

```
14. using ::testing::Return;
15.
16. int main(int argc, char** argv) {
17.         ::testing::InitGoogleMock(&argc, argv);
18.
19.         string value = "Hello World!";
20.         MockFoo mockFoo;
21.
22.         EXPECT_CALL(mockFoo, setValue(testing::_));
23.         mockFoo.setValue(value);
24.
25.         // 这里我故意犯错
26.         EXPECT_CALL(mockFoo, setDoubleValues(Eq(1), Ge(1)));
27.         mockFoo.setDoubleValues(1, 0);
28.
29.         return EXIT_SUCCESS;
30. }
```

第22行，让setValue的形参可以传入任意参数
另外，我在第26~27行故意犯了个错（为了说明上述这些匹配器的作用），我之前明明让setDoubleValues第二个参数得大于等于1,但我实际传入时却传入一个0。这时程序运行时就报错了：

```
unknown file: Failure

Unexpected mock function call – returning directly.
Function call: setDoubleValues(1, 0)
Google Mock tried the following 1 expectation, but it didn't match:

FooMain.cc:35: EXPECT_CALL(mockFoo, setDoubleValues(Eq(1), Ge(1)))…
Expected arg #1: is >= 1
Actual: 0
Expected: to be called once
Actual: never called – unsatisfied and active
FooMain.cc:35: Failure
Actual function call count doesn't match EXPECT_CALL(mockFoo, setDoubleValues(Eq(1), Ge(1)))…
Expected: to be called once
Actual: never called – unsatisfied and active
```

上述的那些匹配器都比较简单，下面我们来看看那些比较复杂的匹配吧。
成员匹配器

Field(&class::field, m)	argument.field (或 argument->field, 当argument是一个指针时)与匹配器m匹配, 这里的argument是一个class类的实例.
Key(e)	形参（argument）比较是一个类似map这样的容器，然后argument.first的值等于e
Pair(m1, m2)	形参（argument）必须是一个pair，并且argument.first等于m1，argument.second等于m2.
Property(&class::property, m)	argument.property()(或argument->property(),当argument是一个指针时)与匹配器m匹配, 这里的argument是一个class类的实例.

还是举例说明一下：

```
1. TEST(TestField, Simple) {
2.         MockFoo mockFoo;
3.         Bar bar;
4.         EXPECT_CALL(mockFoo, get(Field(&Bar::num, Ge(0)))).Times(1);
5.         mockFoo.get(bar);
6. }
7.
8. int main(int argc, char** argv) {
9.         ::testing::InitGoogleMock(&argc, argv);
```

```
10.     return RUN_ALL_TESTS();
11. }
```

这里我们使用Google Test来写个测试用例，这样看得比较清楚。

第5行，我们定义了一个**Field(&Bar::num, Ge(0))**，以说明Bar的成员变量num必须大于等于0。

上面这个是正确的例子，我们为了说明**Field**的作用，传入一个bar.num = -1试试。

```
1. TEST(TestField, Simple) {
2.     MockFoo mockFoo;
3.     Bar bar;
4.     bar.num = -1;
5.     EXPECT_CALL(mockFoo, get(Field(&Bar::num, Ge(0)))).Times(1);
6.     mockFoo.get(bar);
7. }
```

运行是出错了：

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from TestField
[ RUN ] TestField.Simple
unknown file: Failure

Unexpected mock function call – returning directly.
Function call: get(@0xbff335bc 4-byte object )
Google Mock tried the following 1 expectation, but it didn't match:

FooMain.cc:34: EXPECT_CALL(mockFoo, get(Field(&Bar::num, Ge(0))))…
Expected arg #0: is an object whose given field is >= 0
Actual: 4-byte object , whose given field is -1
Expected: to be called once
Actual: never called – unsatisfied and active
FooMain.cc:34: Failure
Actual function call count doesn't match EXPECT_CALL(mockFoo, get(Field(&Bar::num, Ge(0))))
…
Expected: to be called once
Actual: never called – unsatisfied and active
[ FAILED ] TestField.Simple (0 ms)
[----------] 1 test from TestField (0 ms total)

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

1 FAILED TEST
```

匹配函数或函数对象的返回值

ResultOf(f, m)	f(argument) 与匹配器m匹配, 这里的f是一个函数或函数对象.

指针匹配器

Pointee(m)	argument (不论是智能指针还是原始指针) 指向的值与匹配器m匹配.

复合匹配器

AllOf(m1, m2, …, mn)	argument 匹配所有的匹配器m1到mn
AnyOf(m1, m2, …, mn)	argument 至少匹配m1到mn中的一个

Not(m)	argument 不与匹配器m匹配

```
1. EXPECT_CALL(foo, DoThis(AllOf(Gt(5), Ne(10))));
```

传入的参数必须 >5 并且 <= 10

```
1. EXPECT_CALL(foo, DoThat(Not(HasSubstr("blah")), NULL));
```

第一个参数不包含“blah”这个子串

基数（Cardinalities）
基数用于Times()中来指定模拟函数将被调用多少次|

AnyNumber()	函数可以被调用任意次.
AtLeast(n)	预计至少调用n次.
AtMost(n)	预计至多调用n次.
Between(m, n)	预计调用次数在m和n(包括n)之间.
Exactly(n) 或 n	预计精确调用n次. 特别是, 当n为0时,函数应该永远不被调用.

行为（Actions）
Actions（行为）用于指定Mock类的方法所期望模拟的行为：比如返回什么样的值、对引用、指针赋上怎么样个值，等等。 值的返回

Return()	让Mock方法返回一个void结果
Return(value)	返回值value
ReturnNull()	返回一个NULL指针
ReturnRef(variable)	返回variable的引用.
ReturnPointee(ptr)	返回一个指向ptr的指针

另一面的作用（**Side Effects**）

Assign(&variable, value)	将value分配给variable

使用函数或者函数对象（**Functor**）作为行为

Invoke(f)	使用模拟函数的参数调用f, 这里的f可以是全局/静态函数或函数对象.
Invoke(object_pointer, &class::method)	使用模拟函数的参数调用object_pointer对象的mothod方法.

复合动作

DoAll(a1, a2, ..., an)	每次发动时执行a1到an的所有动作.
IgnoreResult(a)	执行动作a并忽略它的返回值. a不能返回void.

这里我举个例子来解释一下**DoAll()**的作用，我个人认为这个DoAll()还是挺实用的。例如有一个Mock方法：

```
1. virtual int getParamter(std::string* name,  std::string* value) = 0
```

对于这个方法，我这回需要操作的结果是将name指向value的地址，并且得到方法的返回值。
类似这样的需求，我们就可以这样定义期望过程：

```
1. TEST(SimpleTest, F1) {
2.     std::string* a = new std::string("yes");
3.     std::string* b = new std::string("hello");
4.     MockIParameter mockIParameter;
5.     EXPECT_CALL(mockIParameter, getParamter(testing::_, testing::_)).Times(1). \
6.         WillOnce(testing::DoAll(testing::Assign(&a, b), testing::Return(1)));
```

```
7.     mockIParameter.getParamter(a, b);
8.  }
```

这时就用上了我们的DoAll()了，它将Assign()和Return()结合起来了。

序列（Sequences）
默认时，对于定义要的期望行为是无序（**Unordered**）的，即当我定义好了如下的期望行为：

```
1.      MockFoo mockFoo;
2.      EXPECT_CALL(mockFoo, getSize()).WillOnce(Return(1));
3.      EXPECT_CALL(mockFoo, getValue()).WillOnce(Return(string("Hello World")));
```

对于这样的期望行为的定义，我何时调用**mockFoo.getValue()**或者何时**mockFoo.getSize()**都可以的。

但有时候我们需要定义有序的（**Ordered**）的调用方式，即序列 **(Sequences)** 指定预期的顺序. 在同一序列里的所有预期调用必须按它们指定的顺序发生; 反之则可以是任意顺序.

```
1.  using ::testing::Return;
2.  using ::testing::Sequence;
3.
4.  int main(int argc, char **argv) {
5.      ::testing::InitGoogleMock(&argc, argv);
6.
7.      Sequence s1, s2;
8.      MockFoo mockFoo;
9.      EXPECT_CALL(mockFoo, getSize()).InSequence(s1, s2).WillOnce(Return(1));
10.     EXPECT_CALL(mockFoo, getValue()).InSequence(s1).WillOnce(Return(
11.         string("Hello World!")));
12.     cout << "First:\t" << mockFoo.getSize() << endl;
13.     cout << "Second:\t" << mockFoo.getValue() << endl;
14.
15.     return EXIT_SUCCESS;
16. }
```

首先在第8行建立两个序列：s1、s2。
然后在第11行中，**EXPECT_CALL(mockFoo, getSize()).InSequence(s1, s2)**说明**getSize()**的行为优先于s1、s2.
而第12行时，**EXPECT_CALL(mockFoo, getValue()).InSequence(s1)**说明**getValue()**的行为在序列s1中。

得到的结果如下：

```
First: 1
Second: Hello World!
```

当我尝试一下把**mockFoo.getSize()**和**mockFoo.getValue()**的调用对调时试试：

```
1.      cout << "Second:\t" << mockFoo.getValue() << endl;
2.      cout << "First:\t" << mockFoo.getSize() << endl;
```

得到如下的错误信息：

```
unknown file: Failure

Unexpected mock function call – returning default value.
Function call: getValue()
Returns: ""
Google Mock tried the following 1 expectation, but it didn't match:

FooMain.cc:29: EXPECT_CALL(mockFoo, getValue())…
Expected: all pre-requisites are satisfied
Actual: the following immediate pre-requisites are not satisfied:
FooMain.cc:28: pre-requisite #0
(end of pre-requisites)
Expected: to be called once
Actual: never called – unsatisfied and active
```

```
Second:
First: 1
FooMain.cc:29: Failure
Actual function call count doesn't match EXPECT_CALL(mockFoo, getValue())...
Expected: to be called once
Actual: never called – unsatisfied and active
```

另外，我们还有一个偷懒的方法，就是不要这么傻乎乎地定义这些个**Sequence s1, s2**的序列，而根据我定义期望行为（**EXPECT_CALL**）的顺序而自动地识别调用顺序，这种方式可能更为地通用。

```
using ::testing::InSequence;
using ::testing::Return;

int main(int argc, char **argv) {
        ::testing::InitGoogleMock(&argc, argv);

        InSequence dummy;
        MockFoo mockFoo;
        EXPECT_CALL(mockFoo, getSize()).WillOnce(Return(1));
        EXPECT_CALL(mockFoo, getValue()).WillOnce(Return(string("Hello World")));

        cout << "First:\t" << mockFoo.getSize() << endl;
        cout << "Second:\t" << mockFoo.getValue() << endl;

        return EXIT_SUCCESS;
}
```

Mock实践

下面我从我在工作中参与的项目中选取了一个实际的例子来实践Mock。

这个例子的背景是用于搜索引擎的：

> 引擎接收一个查询的Query，比如http://127.0.0.1/search?q=mp3&retailwholesale=0&isuse_alipay=1
> 引擎接收到这个Query后，将解析这个Query，将Query的Segment（如q=mp3、retail_wholesale=0放到一个数据结构中）
> 引擎会调用另外内部模块具体根据这些Segment来处理相应的业务逻辑。

由于Google Mock不能Mock模版方法，因此我稍微更改了一下原本的接口，以便演示：

我改过的例子

我们先来看看引擎定义好的接口们：

VariantField.h 一个联合体，用于保存Query中的Segment的值

```
#ifndef VARIANTFIELD_H_
#define VARIANTFIELD_H_

#include <boost/cstdint.hpp>

namespace seamless {

union VariantField
{
    const char * strVal;
    int32_t intVal;
};

} // namespace mlr_isearch_api

#endif // VARIANTFIELD_H_
```

IParameterInterface.h 提供一个接口，用于得到Query中的各个Segment的值

```
#ifndef IPARAMETERINTERFACE_H_
#define IPARAMETERINTERFACE_H_

#include <boost/cstdint.hpp>

#include "VariantField.h"
```

```
7.
8.  namespace seamless {
9.
10. class IParameterInterface {
11. public:
12.         virtual ~IParameterInterface() {};
13.
14. public:
15.         virtual int32_t getParameter(const char* name,  VariantField*& value) = 0;
16. };
17.
18. }  // namespace
19.
20. #endif // IPARAMETERINTERFACE_H_
```

IAPIProviderInterface.h 一个统一的外部接口

```
1.  #ifndef IAPIPROVIDERINTERFACE_H_
2.  #define IAPIPROVIDERINTERFACE_H_
3.
4.  #include <boost/cstdint.hpp>
5.
6.  #include "IParameterInterface.h"
7.  #include "VariantField.h"
8.
9.  namespace seamless {
10.
11. class IAPIProviderInterface {
12. public:
13.         IAPIProviderInterface() {}
14.         virtual ~IAPIProviderInterface() {}
15.
16. public:
17.         virtual IParameterInterface* getParameterInterface() = 0;
18. };
19.
20. }
21.
22. #endif // IAPIPROVIDERINTERFACE_H_
```

引擎定义好的接口就以上三个，下面是引擎中的一个模块用于根据Query中的Segment接合业务处理的。**Rank.h** 头文件

```
1.  #ifndef RANK_H_
2.  #define RANK_H_
3.
4.  #include "IAPIProviderInterface.h"
5.
6.  namespace seamless {
7.
8.  class Rank {
9.  public:
10.         virtual ~Rank() {}
11.
12. public:
13.         void processQuery(IAPIProviderInterface* iAPIProvider);
14. };
15.
16. }  // namespace seamless
17.
18. #endif // RANK_H_
```

Rank.cc 实现

```
1.  #include <cstdlib>
2.  #include <cstring>
3.  #include <iostream>
4.  #include <string>
5.  #include "IAPIProviderInterface.h"
6.  #include "IParameterInterface.h"
```

```
#include "VariantField.h"

#include "Rank.h"

using namespace seamless;
using namespace std;

namespace seamless {

void Rank::processQuery(IAPIProviderInterface* iAPIProvider) {
	IParameterInterface* iParameter = iAPIProvider->getParameterInterface();
	if (!iParameter) {
		cerr << "iParameter is NULL" << endl;
		return;
	}

	int32_t isRetailWholesale = 0;
	int32_t isUseAlipay = 0;

	VariantField* value = new VariantField;

	iParameter->getParameter("retail_wholesale", value);
	isRetailWholesale = (strcmp(value->strVal, "0")) ? 1 : 0;

	iParameter->getParameter("is_use_alipay", value);
	isUseAlipay = (strcmp(value->strVal, "0")) ? 1 : 0;

	cout << "isRetailWholesale:\t" << isRetailWholesale << endl;
	cout << "isUseAlipay:\t" << isUseAlipay << endl;

	delete value;
	delete iParameter;
}

} // namespace seamless
```

从上面的例子中可以看出，引擎会传入一个IAPIProviderInterface对象，这个对象调用getParameterInterface()方法来得到Query中的Segment。
因此，我们需要Mock的对象也比较清楚了，就是要模拟引擎将Query的Segment传给这个模块。其实就是让=模拟**iParameter->getParameter**方法：我想让它返回什么样的值就返回什么样的值.

下面我们开始Mock了：

MockIParameterInterface.h 模拟模拟IParameterInterface类

```
#ifndef MOCKIPARAMETERINTERFACE_H_
#define MOCKIPARAMETERINTERFACE_H_

#include <boost/cstdint.hpp>
#include <gmock/gmock.h>

#include "IParameterInterface.h"
#include "VariantField.h"

namespace seamless {

class MockIParameterInterface: public IParameterInterface {
public:
	MOCK_METHOD2(getParameter, int32_t(const char* name,  VariantField*& value));
};

} // namespace seamless

#endif // MOCKIPARAMETERINTERFACE_H_
```

MockIAPIProviderInterface.h 模拟IAPIProviderInterface类

```
1. #ifndef MOCKIAPIPROVIDERINTERFACE_H_
2. #define MOCKIAPIPROVIDERINTERFACE_H_
3.
4. #include <gmock/gmock.h>
5.
6. #include "IAPIProviderInterface.h"
7. #include "IParameterInterface.h"
8.
9. namespace seamless {
10.
11. class MockIAPIProviderInterface: public IAPIProviderInterface{
12. public:
13.         MOCK_METHOD0(getParameterInterface, IParameterInterface*());
14. };
15.
16. } // namespace seamless
17.
18. #endif // MOCKIAPIPROVIDERINTERFACE_H_
```

tester.cc 一个测试程序，试试我们的Mock成果

```
1. #include <boost/cstdint.hpp>
2. #include <boost/shared_ptr.hpp>
3. #include <cstdlib>
4. #include <gmock/gmock.h>
5.
6. #include "MockIAPIProviderInterface.h"
7. #include "MockIParameterInterface.h"
8. #include "Rank.h"
9.
10. using namespace seamless;
11. using namespace std;
12.
13. using ::testing::_;
14. using ::testing::AtLeast;
15. using ::testing::DoAll;
16. using ::testing::Return;
17. using ::testing::SetArgumentPointee;
18.
19. int main(int argc, char** argv) {
20.         ::testing::InitGoogleMock(&argc, argv);
21.
22.         MockIAPIProviderInterface* iAPIProvider = new MockIAPIProviderInterface;
23.         MockIParameterInterface* iParameter = new MockIParameterInterface;
24.
25.         EXPECT_CALL(*iAPIProvider, getParameterInterface()).Times(AtLeast(1)).
26.                 WillRepeatedly(Return(iParameter));
27.
28.         boost::shared_ptr<VariantField> retailWholesaleValue(new VariantField);
29.         retailWholesaleValue->strVal = "0";
30.
31.         boost::shared_ptr<VariantField> defaultValue(new VariantField);
32.         defaultValue->strVal = "9";
33.
34.         EXPECT_CALL(*iParameter, getParameter(_, _)).Times(AtLeast(1)).
35.                 WillOnce(DoAll(SetArgumentPointee<1>(*retailWholesaleValue), Return(1))).
36.                 WillRepeatedly(DoAll(SetArgumentPointee<1>(*defaultValue), Return(1)));
37.
38.         Rank rank;
39.         rank.processQuery(iAPIProvider);
40.
41.         delete iAPIProvider;
42.
43.         return EXIT_SUCCESS;
44. }
```

第26行，定义一个执行顺序，因此在之前的Rank.cc中，是先调用

iAPIProvider>getParameterInterface，然后再调用iParameter>getParameter，因此我们在下面会先定义MockIAPIProviderInterface.getParameterInterface的期望行为，然后再是其他的。
第27~28行，定义MockIAPIProviderInterface.getParameterInterface的的行为：程序至少被调用一次（Times(AtLeast(1))），每次调用都返回一个iParameter（即MockIParameterInterface*的对象）。
第30~34行，我自己假设了一些Query的Segment的值。即我想达到的效果是Query类似http://127.0.0.1/search?retailwholesale=0&isuse_alipay=9。
第36~38行，我们定义MockIParameterInterface.getParameter的期望行为：这个方法至少被调用一次;第一次被调用时返回1并将第一个形参指向retailWholesaleValue;后续几次被调用时返回1，并指向defaultValue。
第51行，运行Rank类下的processQuery方法。

看看我们的运行成果：

```
isRetailWholesale: 0
isUseAlipay: 1
```

从这个结果验证出我们传入的Query信息是对的，成功Mock!

现实中的例子
就如我之前所说的，上述的那个例子是我改过的，现实项目中哪有这么理想的结构（特别对于那些从来没有**Develop for Debug**思想的同学）。
因此我们来看看上述这个例子中实际的代码：其实只有**IAPIProviderInterface.h**不同，它定义了一个模版函数，用于统一各种类型的接口： **IAPIProviderInterface.h** 真正的IAPIProviderInterface.h，有一个模版函数

```
#ifndef IAPIPROVIDERINTERFACE_H_
#define IAPIPROVIDERINTERFACE_H_

#include <boost/cstdint.hpp>
#include <iostream>

#include "IBaseInterface.h"
#include "IParameterInterface.h"
#include "VariantField.h"

namespace seamless {

class IAPIProviderInterface: public IBaseInterface {
public:
	IAPIProviderInterface() {}
	virtual ~IAPIProviderInterface() {}

public:
	virtual int32_t queryInterface(IBaseInterface*& pInterface) = 0;

	template<typename InterfaceType>
	InterfaceType* getInterface() {
		IBaseInterface* pInterface = NULL;
		if (queryInterface(pInterface)) {
			std::cerr << "Query Interface failed" << std::endl;
		}
		return static_cast<InterfaceType* >(pInterface);
	}
};

}

#endif // IAPIPROVIDERINTERFACE_H_
```

Rank.cc 既然IAPIProviderInterface.h改了，那Rank.cc中对它的调用其实也不是之前那样的。不过其实也就差一行代码：

```
//	IParameterInterface* iParameter = iAPIProvider->getParameterInterface();
	IParameterInterface* iParameter = iAPIProvider->getInterface<IParameterInterface>();
```

因为目前版本（1.5版本）的Google Mock还不支持模版函数，因此我们无法Mock IAPIProviderInterface中的getInterface，那我们现在怎么办？
如果你想做得比较完美的话我暂时也没想出办法，我现在能够想出的办法也只能这样：**IAPIProviderInterface.h** 修改

其中的getInterface，让它根据模版类型，如果是**IParameterInterface**或者**MockIParameterInterface**则就返回一个**MockIParameterInterface**的对象

```
1. #ifndef IAPIPROVIDERINTERFACE_H_
2. #define IAPIPROVIDERINTERFACE_H_
3.
4. #include <boost/cstdint.hpp>
5. #include <iostream>
6.
7. #include "IBaseInterface.h"
8. #include "IParameterInterface.h"
9. #include "VariantField.h"
10.
11. // In order to Mock
12. #include <boost/shared_ptr.hpp>
13. #include <gmock/gmock.h>
14. #include "MockIParameterInterface.h"
15.
16. namespace seamless {
17.
18. class IAPIProviderInterface: public IBaseInterface {
19. public:
20.     IAPIProviderInterface() {}
21.     virtual ~IAPIProviderInterface() {}
22.
23. public:
24.     virtual int32_t queryInterface(IBaseInterface*& pInterface) = 0;
25.
26.     template<typename InterfaceType>
27.     InterfaceType* getInterface() {
28.         IBaseInterface* pInterface = NULL;
29.         if (queryInterface(pInterface) == 0) {
30.             std::cerr << "Query Interface failed" << std::endl;
31.         }
32.
33.         // In order to Mock
34.         if ((typeid(InterfaceType) == typeid(IParameterInterface)) ||
35.             (typeid(InterfaceType) == typeid(MockIParameterInterface))) {
36.             using namespace ::testing;
37.             MockIParameterInterface* iParameter = new MockIParameterInterface;
38.             boost::shared_ptr<VariantField> retailWholesaleValue(new VariantField);
39.             retailWholesaleValue->strVal = "0";
40.
41.             boost::shared_ptr<VariantField> defaultValue(new VariantField);
42.             defaultValue->strVal = "9";
43.
44.             EXPECT_CALL(*iParameter, getParameter(_, _)).Times(AtLeast(1)).
45.                 WillOnce(DoAll(SetArgumentPointee<1>(*retailWholesaleValue), Return(1))).
46.                 WillRepeatedly(DoAll(SetArgumentPointee<1>(*defaultValue), Return(1)));
47.             return static_cast<InterfaceType* >(iParameter);
48.         }
49.         // end of mock
50.
51.         return static_cast<InterfaceType* >(pInterface);
52.     }
53. };
54.
55. }
56.
57. #endif // IAPIPROVIDERINTERFACE_H_
```

第33~49行，判断传入的模版函数的类型，然后定义相应的行为，最后返回一个**MockIParameterInterface**对象

tester.cc

```
int main(int argc, char** argv) {
        ::testing::InitGoogleMock(&argc, argv);

        MockIAPIProviderInterface* iAPIProvider = new MockIAPIProviderInterface;

        InSequence dummy;
        EXPECT_CALL(*iAPIProvider, queryInterface(_)).Times(AtLeast(1)).
                WillRepeatedly(Return(1));

        Rank rank;
        rank.processQuery(iAPIProvider);

        delete iAPIProvider;

        return EXIT_SUCCESS;
}
```

这里的调用就相对简单了，只要一个**MockIAPIProviderInterface**就可以了。

Google Mock Cookbook

这里根据Google Mock Cookbook和我自己试用的一些经验，整理一些试用方面的技巧。

Mock protected、private方法

Google Mock也可以模拟protected和private方法，比较神奇啊（其实从这点上也可以看出，Mock类不是简单地继承原本的接口，然后自己把它提供的方法实现;Mock类其实就等于原本的接口）。

对protected和private方法的Mock和public基本类似，只不过在Mock类中需要将这些方法设置成**public**。

Foo.h 带private方法的接口

```
class Foo {
private:
        virtual void setValue(int value) {};

public:
        int value;
};
```

MockFoo.h

```
class MockFoo: public Foo {
public:
        MOCK_METHOD1(setValue, void(int value));
};
```

Mock 模版类（Template Class）

Google Mock可以Mock模版类，只要在宏MOCK*的后面加上T。

还是类似上述那个例子：

Foo.h 改成模版类

```
template <typename T>
class Foo {
public:
        virtual void setValue(int value) {};

public:
        int value;
};
```

MockFoo.h

```
template <typename T>
class Foo {
public:
        virtual void setValue(int value) {};

public:
        int value;
};
```

Nice Mocks 和 Strict Mocks

当在调用Mock类的方法时，如果之前没有使用EXPECT_CALL来定义该方法的期望行为时，Google Mock在运行时会给

你一些警告信息：

```
GMOCK WARNING:
Uninteresting mock function call – returning default value.
Function call: setValue(1)
Returns: 0
Stack trace
```

对于这种情况，可以使用NiceMock来避免:

```
1.  // MockFoo mockFoo;
2.  NiceMock<MockFoo> mockFoo;
```

使用NiceMock来替代之前的MockFoo。

当然，另外还有一种办法，就是使用StrictMock来将这些调用都标为失败：

```
1.  StrictMock<MockFoo> mockFoo;
```

这时得到的结果：

```
unknown file: Failure
Uninteresting mock function call – returning default value.
Function call: setValue(1)
Returns: 0
```

标签: google mock

上一篇：转：RST攻击
下一篇：64位系统编译svn

0

相关热门文章

TCP窗口行为 | test123 | 关于MYSQL 时间类型存储在数据...
常用IOS开源库整理 | 编写安全代码——小心有符号数... | schedule如何将新程序运行？...
一日为腾讯 终身为企鹅？... | 使用openssl api进行加密解密... | 修改默认端口为222，centos自...
做网站必须遵守的网站策划准则... | 一段自己打印自己的c程序... | 用PHP做一个ftp登录页面...
网站优化选修课 疾速阐发协作... | sql relay的c++接口 | Toad for Oracle工具,为什么在...

给主人留下些什么吧！~~

评论热议

请登录后评论。
登录 注册