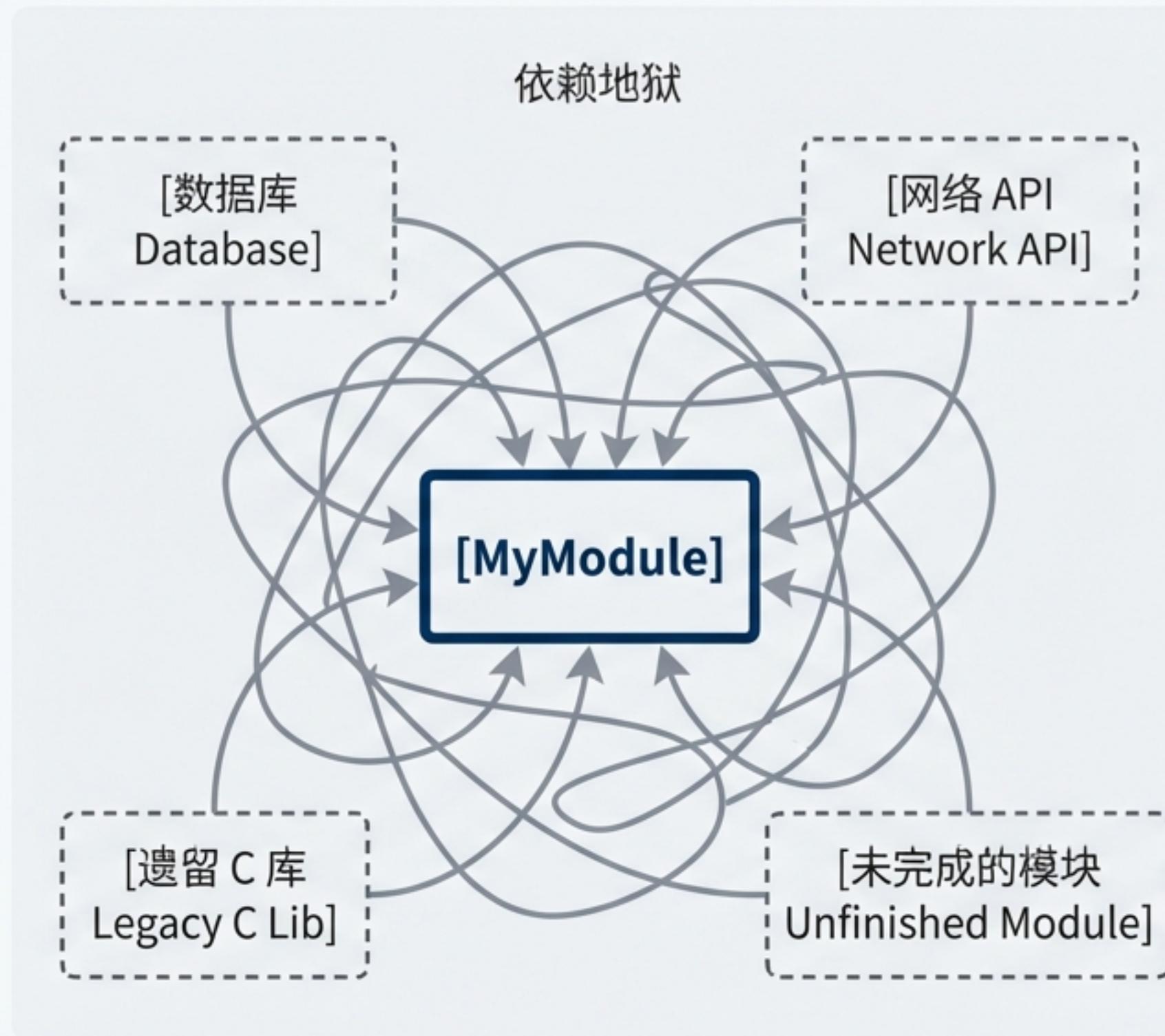


驾驭 C++ 单元测试

GoogleTest 与 MockCpp 双剑合璧深度解析

困境：如何隔离依赖，聚焦核心逻辑？



如何只测试 `MyModule` 的核心业务逻辑？

当依赖项不稳定、未完成或行为复杂时怎么办？

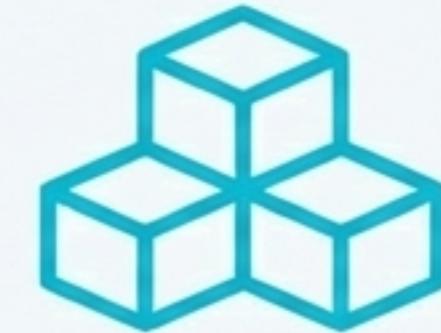
答案：Mock (模拟) —— 为你的依赖创建“行为替身”。

GoogleTest：我们测试体系的坚实基石



丰富的断言

`EXPECT_EQ`, `ASSERT_TRUE` 等。提供清晰的失败信息，`ASSERT_*` 失败会终止当前测试。



测试夹具 (Fixtures)

`TEST_F`, `SetUp()`、`TearDown()`。高效管理测试上下文和共享资源。



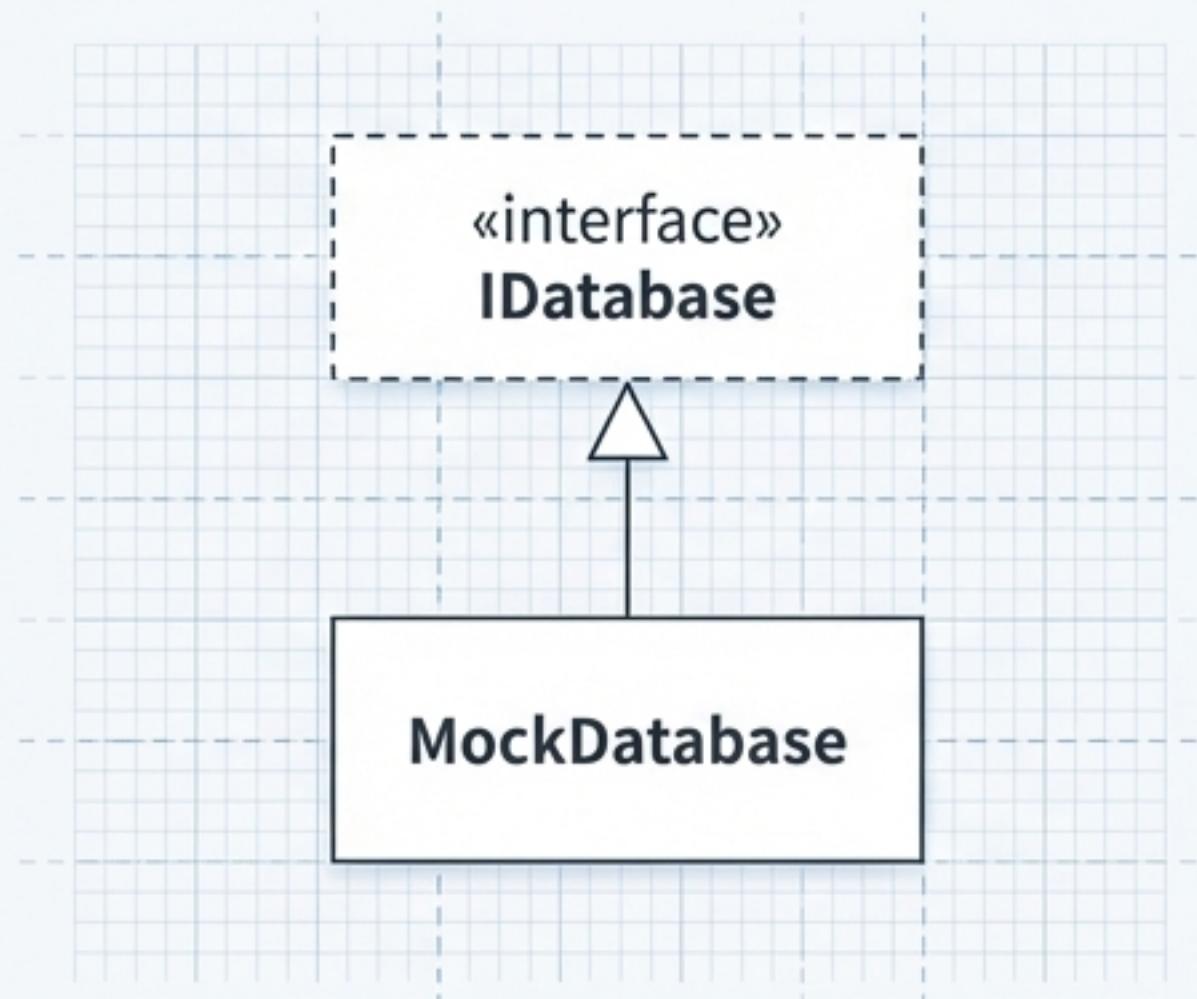
参数化测试

`TEST_P`。用一套逻辑，验证多组数据，避免代码冗余。

GoogleTest 是 C++ 测试的“标准答案”，为我们接下来的 Mock 探索铺平道路。

模拟依赖：两条截然不同的哲学路径

GMock: 继承与多态的“架构师”之道



面向对象 · 类型安全 · 编译时检查 · 依赖注入

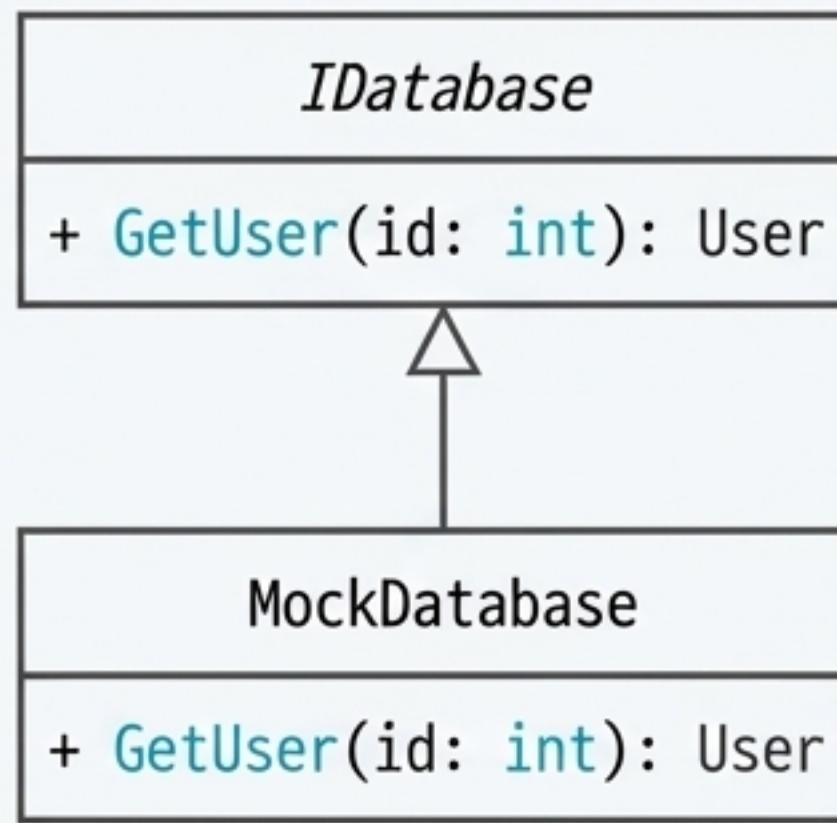
MockCpp: 运行时 Hook 的“外科医生”之术



运行时修改 · 零代码侵入 · 支持C函数/静态函数 · 强大灵活

GMock 之道：基于接口的优雅设计

理念



GMock 利用 C++ 的多态特性，通过继承虚函数来实现 Mock。前提是代码遵循面向接口的设计。

代码实战

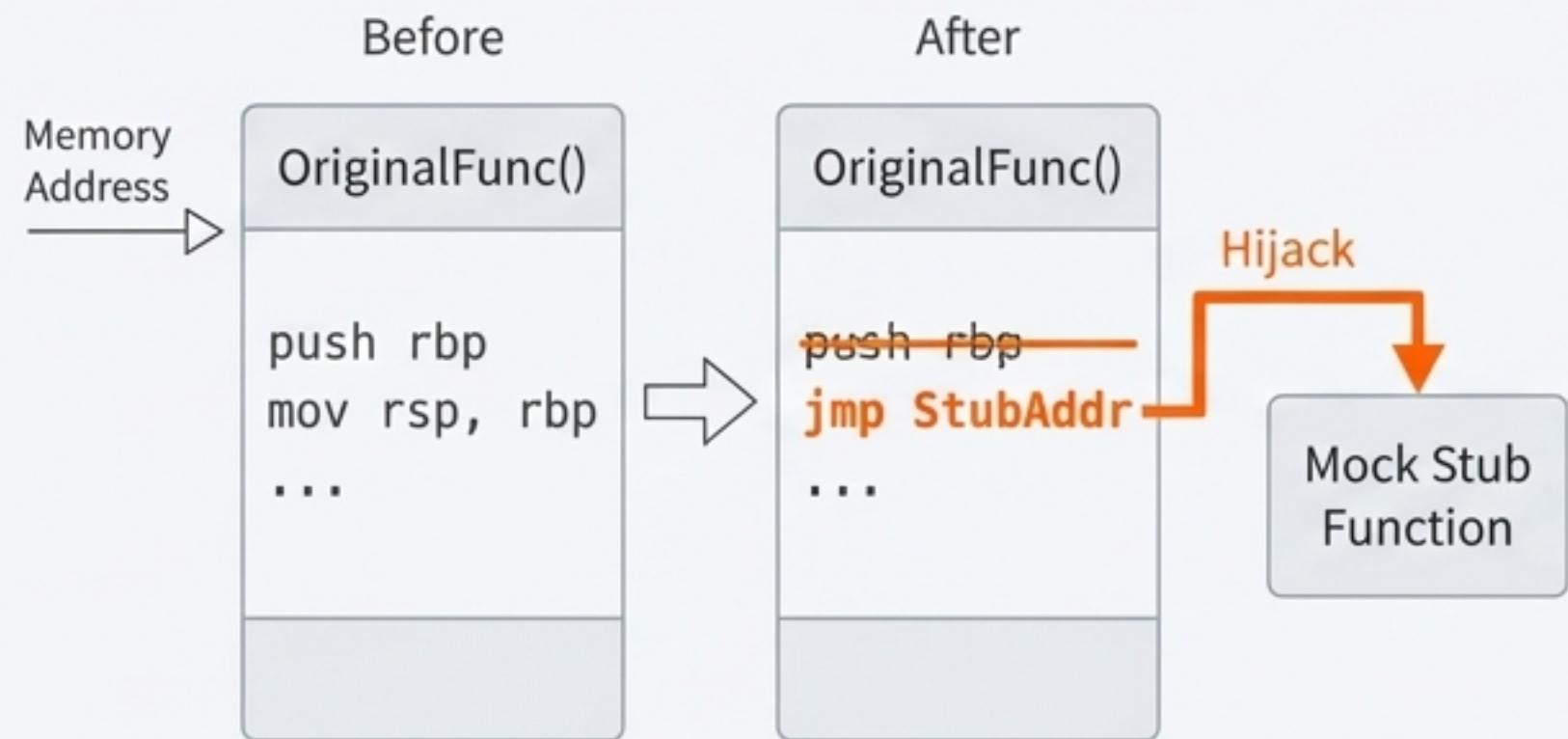
```
// 1. 定义 Mock 类，继承接口
class MockDB : public IDatabase {
public:
    MOCK_METHOD(User, GetUser, (int id), (override));
};

// 2. 在测试中设置期望
TEST(UserRepoTest, FindUser) {
    MockDB mock_db;
    EXPECT_CALL(mock_db, GetUser(101))
        .WillOnce(Return(User("Alice")));
    // ... a call to code that uses mock_db ...
}
```

The code snippet demonstrates how to define a Mock class (`MockDB`) that inherits from the `IDatabase` interface. It shows the declaration of a virtual method `GetUser` using the `MOCK_METHOD` macro. In the test case (`TEST`), it uses the `EXPECT_CALL` macro to set up expectations for the `GetUser` method, specifying the expected argument value (101) and the return value (a user object with name "Alice").

MockCpp 之术：深入运行时的“精准手术”

原理图示



MockCpp 直接修改函数入口的机器指令，强制调用跳转到我们的桩函数。

代码实战

```
// 无需继承或接口，直接 Mock 任意 C 函数或静态函数
int add(int a, int b);

TEST(LegacyTest, MockAdd) {
    MOCKER(add)
        .stubs()
        .with(eq(2), eq(3))
        .will(returnValue(999));

    int result = add(2, 3); // result is 999
    ASSERT_EQ(999, result);
}
```

通过 MOCKER 宏，直接对目标函数 add 进行 Hook，设定当输入参数为 2 和 3 时，强制返回 999，无需修改源代码或依赖接口。

揭秘 MockCpp：三步“偷天换日”



Step 1: 定位函数地址

`&OriginalFunc`

Step 2: 解锁内存权限

.text 段默认是只读的。为写入
指令，必须调用 `mprotect()`
将包含函数地址的内存页权限修
改为 可读 | 可写 | 可执行。

Step 3: 注入跳转指令

将一段 `jmp` 指令的机器码
(14字节) 通过 `memcpy` 写入
`OriginalFunc` 的起始地址，
该指令会无条件跳转到我们的
Mock Stub。

眼见为实：调试器下的“真相”

修改前

```
(lldb) disassemble --name Func
```

...

```
pushq %rbp  
movq %rsp, %rbp
```

修改后

```
(执行 `AddrModifier` 构造后再次执行)
```

...

```
jmpq *(%rip) ; <+6>
```

...



架构师 vs. 外科医生：何时用谁？

特性 (Feature)	GMock (架构师)	MockCpp (外科医生)
Mock 机制	编译时 (继承虚函数)	运行时 (指令 Hook)
适用对象	C++ 虚函数 / 接口	C/C++ 任意函数 (包括静态/全局)
代码侵入性	需要面向接口设计 (高)	几乎为零 (低)
学习曲线	语法稍复杂	链式调用，较直观
杀手锏	强大的类型匹配与检查	处理遗留代码 / C 函数
选择建议	新项目，设计良好	存量代码，技术债多

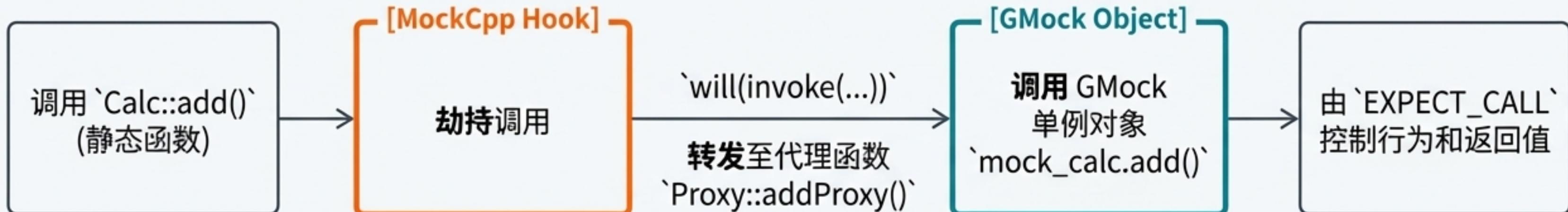
高级战术：当“外科医生”遇上“架构师”

核心问题

GMock 语法优雅，但无法 Mock 静态函数 `Calc::add()`。MockCpp 可以，但语法和功能不如 GMock 丰富。怎么办？

解决方案

用 MockCpp 做一个‘管道’，将静态函数的调用转发给 GMock 对象！



代码实现：优雅的代理模式

第一步：用 MockCpp “一劳永逸”地设置转发

```
// CalcMock 是一个包含 MOCK_METHOD 的单例类  
// 这个 MOCKER 只需要在 Test Fixture 的 SetUp 中设置一次  
MOCKER(Calc::add)  
    .defaults()  
    .will(invoker(CalcMock::addProxy));
```

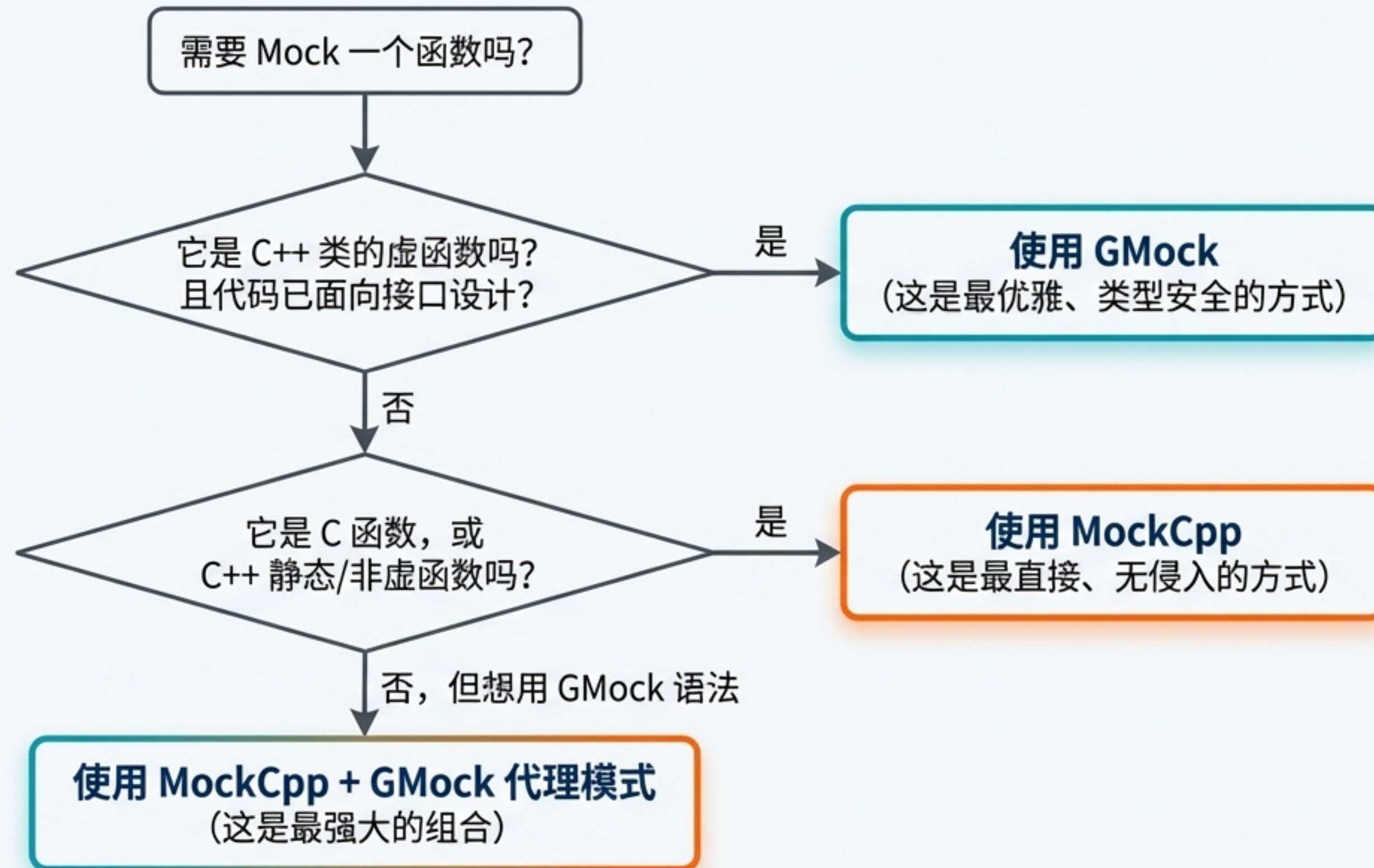
第二步：之后所有测试都用 GMock 语法，灵活控制

```
// 在具体测试用例中，可以多次改变 Mock 行为  
EXPECT_CALL(CalcMock::getInstance(), add(1, 2))  
    .WillOnce(Return(0));  
ASSERT_EQ(0, Calc::add(1, 2));  
  
EXPECT_CALL(CalcMock::getInstance(), add(1, 2))  
    .WillOnce(Return(10));  
ASSERT_EQ(10, Calc::add(1, 2));
```

结论

解决了 MockCpp 无法多次 Mock 同一静态函数的问题，并享受 GMock 强大的语法。

你的 C++ Mock 工具选择指南



框架横评：放眼全局的选择

特性	GTest + MockCpp	GTest + GMock	CMocka
语言	C++	C++	C
Mock 机制	运行时 Hook	虚函数/模板	链接时 --wrap
需要链接选项	✗ (No)	✗ (No)	✓ (Yes)
调用真实函数	不使用 MOCKER 即可	需要 delegate	<code>_real_func()</code>
参数/调用验证	✓ (自动)	✓ (自动)	✗ (手动)
最佳应用场景	遗留系统, C/C++ 混合项目	设计良好的现代 C++ 项目	纯 C 项目

总结：三大核心要点

- 1. GTest 是基石：**提供强大的断言和测试结构。
- 2. GMock vs. MockCpp 是哲学之选：**
 - GMock (架构师)：适用于现代、设计良好的 C++ 代码。
 - MockCpp (外科医生)：是处理遗留代码、C 函数和静态函数的利器。
- 3. 组合出奇迹：**创造性地结合工具，解决最棘手的测试问题。

Q&A

谢谢！

Email: contact@example.com
GitHub: github.com/example



扫描获取演讲源码及示例