(注意:如果您不理解编译器错误，一定要咨询谷歌Mock Doctor。)

# 什么是谷歌c++ mock框架?

当您编写原型或测试时，完全依赖真实对象通常是不可行或不明智的。模拟对象实现了与真实对象相同的接口(因此可以将其作为一个对象使用)，但是允许您在运行时指定如何使用它以及它应该做什么(将调用哪些方法?在顺序?多少次?凭什么参数?他们会返回什么?等等)。

**注意:术语“伪对象”和“模拟对象”很容易混淆。**在测试驱动开发(TDD)社区中，伪造和模仿实际上意味着非常不同的事情:

* **伪对象具有可工作的实现，但通常会采取一些捷径(可能是降低操作的成本)，这使得它们不适合生产。**内存中的文件系统就是一个伪造的例子。
* **模拟是带有预期的预编程对象，它形成预期接收的调用的规范。**

如果这一切对您来说太抽象，不要担心——要记住的最重要的事情是，mock允许您检查它自身和使用它的代码之间的交互。当您开始使用mock时，假货和mock之间的区别将变得更加明显。

**谷歌c++ Mock框架(简称谷歌Mock)是一个库(有时我们也称它为“框架”，以使它听起来很酷)，用于创建和使用模拟类。**它对c++的作用就像jMock和EasyMock对Java的作用一样。

使用谷歌Mock包括三个基本步骤:

1. 使用一些简单的宏来描述要模拟的接口，它们将扩展到模拟类的实现;
2. 创建一些模拟对象，并使用直观的语法指定其期望和行为;
3. 使用模拟对象的练习代码。谷歌Mock将在出现任何违反预期的情况时立即捕获。

# 为什么谷歌模拟?

虽然mock对象可以帮助您删除测试中不必要的依赖项，并使它们快速可靠，但是在c++中手动使用mock是困难的:

* 必须有人实现模拟。这项工作通常是乏味和容易出错的。难怪人们要走很远的路来避免它。
* 那些手工编写的模拟的质量有点，呃，难以预测。您可能会看到一些非常完善的应用程序，但是您也可能会看到一些在匆忙中被破坏的应用程序，并且具有各种特殊的限制。
* 从使用一个mock获得的知识不会转移到下一个mock。

相反，Java和Python程序员有一些很好的模拟框架，可以自动创建模拟。因此，模拟是一种行之有效的技术，在这些社区被广泛采用。拥有正确的工具绝对会带来不同。

谷歌Mock是用来帮助c++程序员的。它的灵感来自jMock和EasyMock，但在设计时考虑到了c++的细节。如果以下任何一个问题困扰着你，那就是你的朋友:

* 你被一个次优的设计困住了，并且希望在太晚之前做更多的原型设计，但是c++的原型设计绝不是“快速的”。
* 您的测试速度很慢，因为它们依赖太多的库或使用昂贵的资源(例如数据库)。
* 您的测试是脆弱的，因为它们使用的一些资源是不可靠的(例如网络)。
* 你想要测试你的代码如何处理一个失败(例如一个文件校验和错误)，但这并不容易。
* 你需要确保你的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，你不得不在行动结束时观察其副作用，这是很尴尬的。
* 你想要“模拟”你的依赖，除了他们还没有模拟实现;而且，坦率地说，你不会对那些手写的嘲笑感到兴奋。

我们鼓励您使用谷歌Mock作为:

* 一个设计工具，因为它可以让你更早、更频繁地试验你的界面设计。更多的迭代导致更好的设计!
* 一个测试工具，用于减少您的测试的出站依赖关系，并探测您的模块及其协作者之间的交互。

# 开始

使用谷歌Mock很容易!在您的c++源文件中，只需包含“gtest/gtest”。h”和“gmock / gmock。你已经准备好出发了。

# 一箱假海龟

让我们看一个例子。假设您正在开发一个图形程序，它依赖于一个类似于logo的API来进行绘图。如何测试它是否正确?好吧，你可以运行它，并将屏幕与金色屏幕快照进行比较，但让我们承认:这样的测试运行起来很昂贵，而且很脆弱(如果你只是升级到一个新的显卡，它有更好的抗锯齿性能，那该怎么办?)突然之间，你必须更新你所有的黄金形象。如果你所有的考试都像这样，那就太痛苦了。幸运的是，你学习了依赖注入，知道该怎么做:不是让你的应用程序直接与绘图API对话，而是将API封装在一个接口中(比如，Turtle)，并编写到该接口的代码:

龟类{…  
  
virtual ~Turtle() {} virtual void PenUp() = 0;  
  
虚拟空间PenDown() = 0;  
虚虚前向(int距离)= 0;  
虚拟空转(int度)= 0;  
虚虚GoTo(int x, int y) = 0;  
虚拟int GetX() const = 0;  
int () const = 0;};

(注意,海龟的析构函数必须是虚拟的,是所有类的理由你打算继承——否则派生类的析构函数不会被称为指针,当你删除一个对象通过一个基地,你会得到损坏程序内存泄漏等州。)

您可以使用PenUp()和PenDown()控制乌龟的移动是否会留下痕迹，使用Forward()、Turn()和GoTo()控制它的移动。最后，GetX()和GetY()告诉您turtle的当前位置。

您的程序通常将使用此接口的实际实现。在测试中，可以使用模拟实现。这允许您轻松地检查程序调用的绘图原语、参数和顺序。以这种方式编写的测试更加健壮(它们不会崩溃，因为您的新机器以不同的方式进行了反走样)，更容易阅读和维护(测试的目的是在代码中表示的，而不是在一些二进制映像中)，并且运行速度快得多。

# 编写模拟类

如果幸运的话，您需要使用的模拟已经由一些不错的人实现了。但是，如果您发现自己处于编写模拟类的位置，请放松—谷歌mock将这个任务变成了一个有趣的游戏!(几乎)。

## 如何定义它

以Turtle界面为例，以下是您需要遵循的简单步骤:

1. 从Turtle派生出MockTurtle类。
2. 以Turtle的一个虚拟函数为例(虽然可以使用模板模拟非虚拟方法，但它涉及的内容要多得多)。数一数它有多少个参数。
3. 在public:部分的子类，写MOCK\_METHODn();(或MOCK\_CONST\_METHODn ();如果您正在模拟一个const方法)，其中n是参数的数量;如果你算错了，那是你的耻辱，编译器错误会告诉你。
4. 接下来是有趣的部分:将函数签名作为宏的第一个参数，剪切并粘贴函数名，剩下的作为第二个参数(如果您好奇，这是函数的类型)。
5. 重复此操作，直到要模拟的所有虚拟函数都完成。

在这个过程之后，你应该有这样的东西:

# include“gmock / gmock。h" //引入了谷歌Mock。  
素甲鱼:公立甲鱼{公立:…  
  
  
MOCK\_METHOD0 (PenUp无效());  
MOCK\_METHOD0 (PenDown无效());  
MOCK\_METHOD1(向前,空白距离(int));  
MOCK\_METHOD1(又无效(int度));  
MOCK\_METHOD2(GoTo, void(int x, int y));  
int MOCK\_CONST\_METHOD0 (GetX ());  
MOCK\_CONST\_METHOD0 (GetY int ());};

您不需要在其他地方定义这些模拟方法—MOCK\_METHOD\*宏将为您生成这些定义。就这么简单!一旦您熟悉了它，您就可以比源代码控制系统处理签入的速度更快地生成模拟类。

**提示:如果这对您来说工作量太大，那么您可以找到gmock\_gen。**py工具在谷歌Mock的脚本/生成器/目录(由cppclean项目提供)中非常有用。这个命令行工具要求您安装Python 2.4。您给它一个c++文件和其中定义的抽象类的名称，它将为您打印模拟类的定义。由于c++语言的复杂性，这个脚本可能不是总能工作，但是当它工作时，它可能非常方便。要了解更多细节，请阅读用户文档。

## 放在哪里

定义模拟类时，需要决定将其定义放在何处。有些人把它放在\*\_test.cc中。当被模拟的接口(比如Foo)由同一个人或团队拥有时，这是没有问题的。否则，当Foo的所有者更改它时，您的测试可能会中断。(你不能指望Foo的维护者去修复每一个使用Foo的测试，对吧?)

因此，经验法则是:如果您需要模拟Foo，而它是由其他人拥有的，那么在Foo的包中定义模拟类(最好是在一个测试子包中，这样您就可以清楚地将生产代码和测试工具分开)，并将其放在mock\_foo.h中。然后每个人都可以引用mock\_foo。h从他们的测试。如果Foo发生了变化，那么只需要修改MockFoo的一个副本，并且只需要修改那些依赖于变化的方法的测试。

另一种方法是:您可以在Foo和代码之上引入一个瘦层FooAdaptor到这个新接口。因为您拥有FooAdaptor，所以可以更轻松地吸收Foo中的更改。虽然这在一开始需要更多的工作，但是仔细选择adaptor接口可以使您的代码更容易编写和更具可读性(从长远来看是一个优势)，因为您可以选择FooAdaptor来更好地适应您的特定域。

# 在测试中使用模拟

一旦有了模拟类，就很容易使用它。典型的工作流程为:

1. 从测试命名空间中导入谷歌模拟名称，这样就可以不限定地使用它们(每个文件只需执行一次)。请记住，名称空间是一个好主意，而且对您的健康有好处。
2. 创建一些模拟对象。
3. 指定您对它们的期望(一个方法将被调用多少次?凭什么参数?它应该做什么?等等)。
4. 练习一些使用模拟的代码;可以选择使用谷歌测试断言检查结果。如果调用mock方法的次数超过预期，或者参数错误，那么您将立即得到一个错误。
5. 当一个mock被销毁时，谷歌mock将自动检查它上的所有期望是否都得到了满足。

这里有一个例子:

# include“路径/ /素甲鱼。h gmock / gmock # include。  
h”#包括“gt / gt。  
h”至少测试使用::::;  
{仿龟龟;  
  
  
// #2 EXPECT\_CALL(turtle, PenDown()) // #3 .Times(至少(1));  
  
  
  
画家画家(乌龟);/ / # 4 EXPECT\_TRUE(画家。  
  
{//在运行测试之前，必须执行以下代码来初始化谷歌Mock //(和谷歌测试)。  
  
  
  
  
  
::测试::InitGoogleMock(命令行参数个数,argv);  
返回RUN\_ALL\_TESTS ();}

您可能已经猜到，这个测试检查PenDown()至少被调用一次。如果painter对象没有调用这个方法，你的测试就会失败，结果会是这样的:

路径/ / my\_test。cc:119: FailureActual function call count不符合这个期望:Actually: never called;Expected:至少调用一次。

**提示1:如果您从Emacs缓冲区运行测试，您可以在错误消息中显示的行号上点击<Enter>，以直接跳到失败的期望。**

**技巧2:如果从未删除模拟对象，则不会进行最终验证。**因此，当您在堆上分配模拟时，最好在测试中使用堆泄漏检查器。

**重要提示:谷歌Mock要求在调用模拟函数之前设置预期值，否则行为是未定义的。**特别是，不能交叉使用EXPECT\_CALL()s和对模拟函数的调用。

这意味着应该将EXPECT\_CALL()读取为预期将来会发生调用，而不是预期已经发生调用。为什么谷歌模拟是这样工作的?嗯，预先指定期望允许谷歌Mock在上下文(堆栈跟踪等)仍然可用时，在冲突发生时报告冲突。这使得调试更加容易。

不可否认，这个测试是人为设计的，并没有多大作用。即使不使用谷歌Mock，也可以轻松实现相同的效果。然而，我们很快就会揭示，谷歌Mock允许您对模拟做更多的事情。

## 在任何测试框架中使用谷歌Mock

如果您想使用谷歌测试(如CppUnit或CxxTest)以外的测试框架，只需将上一节中的main()函数更改为:

下面的代码行导致谷歌Mock在失败时抛出异常，//将被您的测试框架解释为测试失败。  
  
  
::测试::GTEST\_FLAG (throw\_on\_failure) = true;  
::测试::InitGoogleMock(命令行参数个数,argv);  
…无论您的测试框架需要什么…

这种方法有一个缺陷:有时它使谷歌模拟从模拟对象的析构函数抛出异常。对于某些编译器，这有时会导致测试程序崩溃。您仍然可以注意到测试失败了，但这并不是一个优雅的失败。

更好的解决方案是使用谷歌测试的事件监听器API向测试框架报告测试失败。您需要实现事件监听器接口的OnTestPartResult()方法，但它应该是简单的。

如果这工作量太大，我们建议您坚持使用谷歌测试，它可以无缝地与谷歌模拟一起工作(实际上，它在技术上是谷歌模拟的一部分)。如果有你不能使用谷歌测试的原因，请告诉我们。

# 设定的期望

成功使用模拟对象的关键是对其设置正确的期望。如果您将期望设置得太严格，那么您的测试将由于不相关的更改而失败。如果你把它们放得太松，虫子就会溜进来。您想要做得恰到好处，这样您的测试就可以准确地捕获您想要捕获的bug。谷歌Mock为您提供了“恰到好处”的必要方法。

## 一般的语法

在谷歌模拟中，我们使用EXPECT\_CALL()宏来设置模拟方法上的期望。一般语法是:

(mock\_object, method(matchers)) .Times(cardinality) .WillOnce(action) . will(action);

该宏有两个参数:首先是模拟对象，然后是方法及其参数。注意，两者之间用逗号(，)分隔，而不是句号(.)。(为什么要用逗号?答案是，出于技术原因，这是必要的。)

宏后面可以跟着一些可选的子句，它们提供了关于期望的更多信息。在接下来的几节中，我们将讨论每个子句是如何工作的。

这种语法旨在使期望读起来像英语。例如，你可能会猜到

测试使用::::返回;…  
(Return(100)) .WillOnce(Return(150)) . will(Return(200));

说明turtle对象的GetX()方法将被调用5次，第一次返回100次，第二次返回150次，每次返回200次。有些人喜欢将这种语法风格称为领域特定语言(DSL)。

**注意:为什么我们要使用宏来做这个?**它有两个目的:首先，它使期望很容易识别(通过grep或人工阅读器);其次，它允许谷歌Mock在消息中包含失败期望的源文件位置，从而简化调试。

## 牧场主:我们期待什么样的争论?

当模拟函数接受参数时，我们必须指定需要哪些参数;例如:

//预计乌龟前进100个单位。  
EXPECT\_CALL(海龟,远期(100));

有时你可能不想说得太具体(还记得关于测试太严格的说法吗?过多的规格说明导致了测试的脆弱，模糊了测试的意图。因此，我们建议您只指定必要的内容—不多不少。如果你想检查Forward()是否会被调用，但是对它的实参不感兴趣，就把\_写成实参，它的意思是“一切照常”:

测试使用::::\_;…  
  
//希望乌龟继续前进。  
EXPECT\_CALL(乌龟向前(\_));

\_是我们所说的matchers的一个实例。匹配器类似于谓词，可以测试一个参数是否符合我们的期望。只要需要函数参数，就可以在EXPECT\_CALL()中使用匹配器。

在备忘单中可以找到内置的匹配器列表。例如，这里是Ge(大于或等于)匹配器:

测试使用::::通用;…  
EXPECT\_CALL(乌龟向前(通用电气(100)));

这检查海龟将被告知前进至少100单位。

## 基数:它将被调用多少次?

我们可以在EXPECT\_CALL()之后指定的第一个子句是Times()。我们称它的参数为基数，因为它表示调用应该发生多少次。它允许我们重复一个期望很多次，而不需要写很多次。更重要的是，基数可以是“模糊的”，就像匹配器一样。这允许用户准确地表达测试的意图。

一个有趣的特例是乘以(0)您可能已经猜到—这意味着根本不应该使用给定的参数调用该函数，并且无论何时(错误地)调用该函数，谷歌Mock都会报告谷歌测试失败。

我们前面已经看到(n)作为模糊基数的一个例子。有关您可以使用的内置基数列表，请参阅备忘单。

可以省略Times()子句。**如果省略Times()，则谷歌Mock将为您推断基数。**这些规则很容易记住:

* 如果在EXPECT\_CALL()中既没有WillOnce()，也没有will()，则推断的基数是乘以(1)。
* 如果有n个WillOnce()，但没有will()，其中n个>= 1，则基数是乘以(n)。
* 如果有n个WillOnce()和一个will()，其中n个>= 0，则基数是乘以(至少(n))。

**快速测试:如果一个函数被期望调用两次，但实际上调用了四次，你认为会发生什么?**

## 行动:它应该做什么?

还记得模拟对象实际上没有一个工作实现吗?作为用户，我们必须告诉它调用方法时应该做什么。这在谷歌模拟中很简单。

首先,如果一个模拟函数的返回类型是内置类型或指针,函数有一个默认的动作(void函数就会返回,bool函数将返回false,和其他函数将返回0)。此外,在c++中11个及以上,模拟函数的返回类型是default-constructible(即有一个默认的构造函数)有一个默认的返回一个默认构造价值的行动。如果你什么都不说，就会使用这种行为。

其次，如果模拟函数没有默认操作，或者默认操作不适合您，那么您可以使用一系列WillOnce()子句和一个可选的willrepeat()来指定期望匹配的每次操作。例如,

测试使用::::返回;…  
(Return(100)) .WillOnce(Return(200)) .WillOnce(Return(300));

这就是说，正确地调用turtle.GetX()三次(谷歌Mock从我们已经编写的WillOnce()子句的数量推断出这一点，因为我们没有显式地编写times())，并且将分别返回100、200和300次。

测试使用::::返回;…  
.WillOnce(Return(100)) .WillOnce(Return(200)) . will(Return(300));

gety()至少会被调用两次(谷歌Mock知道这一点，因为我们已经编写了两个WillOnce()子句和一个will()，但是没有显式的Times())，第一次返回100，第二次返回200，第三次返回300。

当然，如果显式地编写了一个Times()，那么谷歌Mock将不会尝试推断基数本身。如果指定的数目大于WillOnce()子句的数目，该怎么办?在使用完所有WillOnce()之后，谷歌Mock将每次执行该函数的默认操作(当然，除非您有一个will().)。

除了Return()，我们还能在WillOnce()里面做什么呢?您可以使用ReturnRef(变量)返回引用，或者调用预定义的函数等。

**重要提示:EXPECT\_CALL()语句只计算操作子句一次，即使操作可能执行多次。**因此，你必须小心副作用。以下可能做不到你想要的:

int = 100;EXPECT\_CALL(turtle, GetX()).Times(4). will(Return(n++));

而不是返回100 101 102…，这个模拟函数总是返回100，因为n++只计算一次。类似地，Return(new Foo)将在执行EXPECT\_CALL()时创建一个新的Foo对象，并且每次返回相同的指针。如果您希望副作用每次都发生，您需要定义一个自定义操作，我们将在CookBook中介绍。

又到了测验时间!你认为以下是什么意思?

测试使用::::返回;…  
EXPECT\_CALL(海龟,GetY())同学(4).WillOnce(返回(100));

显然，turtle.GetY()将被调用四次。但是，如果您认为它每次都会返回100，请三思!请记住，每次调用函数时都将使用一个WillOnce()子句，并在调用后执行默认操作。所以正确的答案是，turtle.GetY()第一次返回100，但是从第二次开始返回0，因为返回0是int函数的默认操作。

## 使用多个预期

到目前为止，我们只展示了单一期望的例子。更实际的做法是，您将在多个模拟方法上指定期望，这些方法可能来自多个模拟对象。

默认情况下，当调用模拟方法时，谷歌mock将以定义期望的相反顺序搜索期望，并在找到匹配参数的活动期望时停止(您可以将其视为“更新的规则覆盖旧的规则”)。如果匹配的期望不能再接受任何调用，您将得到一个违反上限的失败。这里有一个例子:

测试使用::::\_;…  
EXPECT\_CALL(乌龟向前(\_));// #1EXPECT\_CALL(乌龟，前进(10))// #2 .乘以(2);

如果连续三次调用Forward(10)，那么第三次调用时将出现错误，因为最后一个匹配期望(#2)已经饱和。但是，如果将第三个Forward(10)调用替换为Forward(20)，那么就没有问题，因为现在#1是匹配的期望。

**旁注:为什么谷歌模拟搜索与期望的顺序相反的匹配?**原因是，这允许用户在模拟对象的构造函数或测试装置的设置阶段中设置默认的期望，然后通过在测试主体中编写更具体的期望来定制模拟。所以，如果你对同一种方法有两种期望，你会想把一个有更具体的匹配器的方法放在另一个之后，或者更具体的规则会被更普遍的规则所遮蔽。

## 有序调用与无序调用

默认情况下，期望可以匹配调用，即使先前的期望没有得到满足。换句话说，调用不必按照指定期望的顺序发生。

有时，您可能希望所有预期的调用都以严格的顺序发生。在谷歌Mock中这样说很简单:

测试使用::::InSequence;…  
测试(FooTest, DrawsLineSegment){…  
  
{InSequence假;  
  
  
EXPECT\_CALL(海龟,PenDown ());  
EXPECT\_CALL(海龟,远期(100));  
EXPECT\_CALL(海龟,PenUp ());  
}Foo ();}

通过创建InSequence类型的对象，其作用域中的所有期望都被放入一个sequence中，并且必须按顺序发生。由于我们只是依赖于这个对象的构造函数和析构函数来完成实际的工作，所以它的名称实际上是不相关的。

在本例中，我们测试了Foo()调用三个预期函数的顺序。如果一个调用是无序的，它将是一个错误。

(如果你关心的是一些调用的相对顺序，而不是所有调用的相对顺序呢?可以指定任意的偏序吗?答案是……是的!如果你没有耐心，可以在烹饪书中找到细节。

## 所有的期望都是刚性的(除非另有说明)

现在让我们来做一个小测试，看看你能多好地使用这些模拟的东西。如何测试请求海龟精确地返回原点两次(您想忽略它收到的任何其他指令)?

当你想出答案后，看看我们的，并交换一下意见(首先自己解决——不要作弊!)

测试使用::::\_;…  
// 1 .Times(AnyNumber();EXPECT\_CALL(turtle, GoTo(0,0)) // 2 .Times(2);

假设乌龟。GoTo(0,0)被调用三次。第三次，谷歌Mock将看到参数匹配期望#2(记住，我们总是选择最后一个匹配的期望)。现在，因为我们说过应该只有两个这样的调用，所以谷歌Mock将立即报告一个错误。这基本上就是我们在上面的“使用多重期望”一节中告诉你的。

这个示例显示谷歌Mock中的期望在默认情况下是“粘性的”，因为即使我们达到了它们的调用上限，它们仍然是活动的。这是一条需要记住的重要规则，因为它影响了规范的含义，并且与其他许多mock框架中的实现方式不同(为什么要这样做呢?)因为我们认为我们的规则使一般情况更容易表达和理解。

简单的?让我们看看您是否真正理解了它:下面的代码说明了什么?

测试使用::::返回;…  
  
for (int i = n;我> 0;i——){EXPECT\_CALL(turtle, GetX()) .WillOnce(Return(10\*i));}

getx()将被调用n次，并返回10,20,30，…，连续地，三思而后行!问题是，正如我们所说的，期望是粘性的。因此，第二次调用turtle.GetX()时，最后一个(最新的)EXPECT\_CALL()语句将匹配，并立即导致“超出上限”错误——这段代码不是很有用!

getx()将返回10,20,30，…，就是明确地说，期望是没有粘性的。换句话说，他们一旦饱和就应该退休:

测试使用::::返回;…  
  
for (int i = n;我> 0;i——){EXPECT\_CALL(turtle, GetX()) .WillOnce(Return(10\*i)) . reti谐振饱和();}

而且，还有一种更好的方法:在这种情况下，我们期望调用以特定的顺序发生，并将操作排列起来以匹配顺序。由于顺序在这里很重要，我们应该使用一个序列使它显式:

测试使用::::InSequence;使用::测试::返回;…  
  
  
{InSequence年代;  
  
  
for (int i = 1;我< = n;{EXPECT\_CALL(turtle, GetX()) .WillOnce(Return(10\*i)) . reti谐振饱和();  
  
  
  
}}

顺便说一下，另一种期望可能不具有粘性的情况是当它在序列中——一旦在序列中它之后的另一个期望被使用了，它就会自动退出(并且永远不会被用来匹配任何调用)。

## 无趣的电话

模拟对象可能有许多方法，但并不是所有方法都那么有趣。例如，在某些测试中，我们可能不关心调用GetX()和GetY()的次数。

在谷歌Mock中，如果您对某个方法不感兴趣，就不要对它做任何说明。如果发生了对这个方法的调用，您将在测试输出中看到一个警告，但不会是失败。

# 现在该做什么?

恭喜你!您已经学习了足够多的谷歌Mock知识，可以开始使用它了。现在，您可能想要加入googlemock讨论组，并实际使用谷歌Mock编写一些测试—这将很有趣。嘿，它甚至可能会上瘾——你已经被警告过了。

然后，如果你想提高你的模拟商数，你应该继续学习烹饪书。您可以在这里学习谷歌Mock的许多高级功能，并提高您的享受和测试的幸福程度。