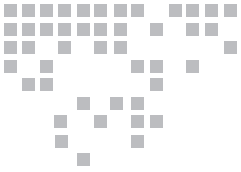

Chapter 3 第 3 章

测试替身

本章内容包括：

- 我们能用测试替身做些什么
- 哪些测试替身可供选择
- 使用测试替身的指南

自从我们开始用类和方法来构建软件时，桩（stub）或哑元（dummy）的概念也差不多存在了。过去这类工具主要用于占位，直到真正的事物准备好——它允许你在周边代码就位之前就能编译和执行某段代码。

在现代开发者测试的上下文中，这些对象具有了更多的不同目的。除了允许在某些依赖缺失的情况下编译执行代码之外，崇尚测试的程序员还创建了一系列“仅供测试”的工具，用于隔离被测代码、加速执行测试、使随机行为变得确定、模拟特殊情况，以及使测试能够访问隐藏信息。

满足这些目的的各种对象具有相似之处，但又有所区别，我们统称为测试替身（test double）。[⊖]

我们先探讨开发者采用测试替身的理由。理解了使用测试替身的潜在好处后，我们看看各种可供选择的类型。最后，我们以几个使用测试替身的简单指南来结束本章。

但是现在，我们问问自己，它对我意味着什么？

⊖ 尽管测试替身术语最初是由 Manning 的作家同行 J. B. Rainsberger 介绍给我的。但我相信是 Gerard Meszaros 和他的《xUnit Test Patterns：Refactoring Test Code》一书（Addison Wesley，2007）将此术语及相关分类在软件社区发扬光大。

3.1 测试替身的威力

甘地（Mahatma Gandhi）说过："改变世界从自身做起"。（Be the change you want to see in the world.）测试替身响应了甘地的召唤，成为你在代码中希望见到的变化。牵强附会？容我慢慢道来。

代码是一个大集合。它是指代其他代码的代码网络。每一块都有预定义的行为——作为程序员的你定义了那些行为。某些行为是原子的，包含在单个类或方法中。某些行为意味着不同代码块之间的交互。

为了时不时地验证一段代码的行为符合你的期望，最好的选择是替换其周围的代码，使你获得对环境的完整控制，从而在其中测试你的代码。你有效地将被测代码与其协作者隔离开，以便进行测试，如图 3.1 所示。

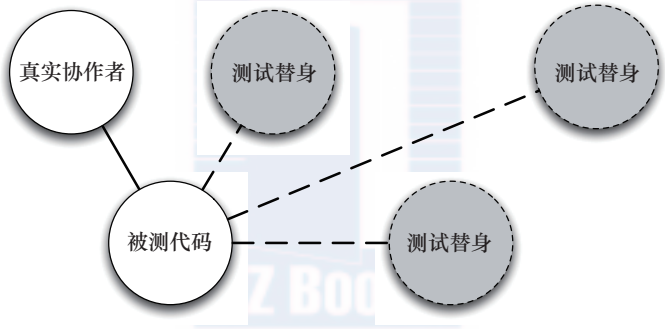

图 3.1 测试替身帮助你隔离被测代码，这样你就能测试其行为的各个方面。本例中，我们用测试替身替换掉了四个协作者中的三个，将测试范围确定为被测代码本身的行为和协作，外加一个特定的协作者

这是引入测试替身的最根本原因——将被测代码与周围隔离开。此外，如本章开头所述，还存在许多其他原因。我们认为"仅供测试"的工具是为了：

- 隔离被测代码
- 加速执行测试
- 使执行变得确定
- 模拟特殊情况
- 访问隐藏信息

存在多种类型的测试替身可供实现这些效果。多数效果可以用一种测试替身实现，而有些则只匹配于某种特定类型。3.2 节会再次讨论这些问题。现在，我想对列出的理由建立共识——在第一时间获得测试替身的理由，以及使用它们的目的。

3.1.1 隔离被测代码

讨论在面向对象编程语言的上下文中隔离被测代码时，我们的世界包含两种东西：

- 被测代码
- 与被测代码交互的代码

当我们说要“隔离被测代码”时，意味着将需要测试的代码与所有其他代码隔离开来。如此一来，我们不仅使测试更加有针对性和容易理解，还更容易建立测试。实际上，“所有其他代码”包括了从被测代码中调用的代码。代码清单 3.1 通过一个简单的例子来展示。

代码清单 3.1 被测代码（Car）及其协作者（Engine 和 Route）

```
public class Car {
   private Engine engine;

   public Car(Engine engine) {
      this.engine = engine;
   }

   public void start() {
      engine.start();
   }

   public void drive(Route route) {
      for (Directions directions : route.directions()) {
         directions.follow();
      }
   }

   public void stop() {
      engine.stop();
   }
}
```


如你所见，这个例子包含了汽车（Car）、汽车引擎（Engine）和由一系列方向（Directions）组成的路径（Route）。假设现在你想要测试汽车。我们总共有四个类，其中一个是被测代码（Car），两个是协作者（Engine 和 Route）。为什么 Directions 不是协作者？某种意义上，Car 引用和调用了 Directions 上的方法。但是还有另一个角度去观察这个场景。我们看看图 3.2 能否帮助澄清这个观点。

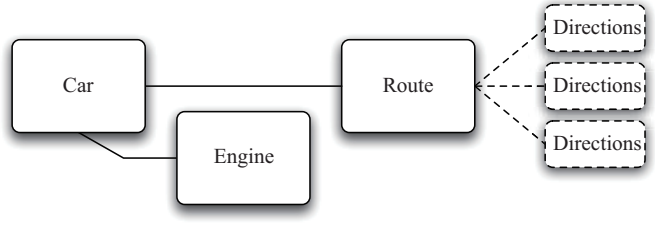

图 3.2 Car 直接使用了 Engine 和 Route，而只是间接地使用了 Directions

如果我们从 Car 的方法中引用的类来关注高一级的抽象层次，并站在 Car 的角度，我们看到的会是 Car 通过 Route 来获取和访问 Directions（如图 3.2）。因此，用测试替身替换 Engine 和 Route，即可将 Car 与其所有的协作者都隔离开。由于我们用伪实现替换了 Route，因此完全控制了向 Car 提供的各种 Directions。

既然你明白了基本原则，即如何通过一些测试替身进行替换从而获得控制，我们再来看看用它们还能做哪些好玩儿的事情。

3.1.2 加速执行测试

替换掉真实协作者会带来一个愉悦的副作用，那就是测试替身的实现经常比真实事物执行得要快。有时，测试替身的速度不只是副作用，而是使用测试替身的主要原因。

考虑图 3.2 中的驾驶例子。假设初始化 Route 要涉及加权图搜索算法，以便找出汽车（Car）当前位置与目的地之间的最短路径。由于今日街道和高速公路网络的复杂性，计算需要花一点时间。尽管折腾一次算法可能还比较快，但即使小小的延迟也会积少成多。如果每个测试都初始化一次 Route，你可能会在这个算法上消耗好几秒甚至几分钟的 CPU 周期——当开发者运行自动化测试来获得快速反馈时，几分钟就等于永远。

放置一个测试替身，令它总是返回预先计算好的通往终点的路径，这样就会避免不必要的等待，而且测试运行得更快了。太棒了。但有些地方还是需要那些缓慢的 Route 算法——在单独有针对性的测试中——但你不希望到处都运行缓慢的算法。

尽管速度总是一件好事，但它不总是最重要的事情。毕竟，如果方向开错了，再快的车也没用。

3.1.3 使执行变得确定

我曾听过著名励志演讲家 Tony Robbins 讲到过惊喜，尽管我们都说自己喜欢惊喜，但我们只喜欢那些自己想要的惊喜。没错，对于软件也一样，特别是当谈到测试代码时。

测试就是指定行为，并验证行为符合规范。只要代码具有完全确定性，并且其逻辑不包含一丝随机性，这就是简单而直接的。其实，为了使代码（和测试）具有确定性，你就需要能够针对同样的代码重复地运行测试，并总是得到相同的结果。

很多时候，你的生产代码需要包含随机性因素，或者其他因素造成重复执行的结果不唯一。例如，如果你开发一个掷骰子的 Craps 游戏，你最好让骰子的结果不能预测——这就是随机。[⊖]

⊖ Craps 是一个掷骰子游戏，玩家对两只骰子的结果下注。这就是电影中黑帮玩的游戏，他们蹲在街角，守着一堆钞票在掷骰子。

或许不确定行为的最典型情形就是依赖于时间的行为。回到 Car 的例子，它向 Route 请求 Directions，想象一下用来计算路径的算法会涉及时间，以及流量、限速等，如代码清单 3.2 所示。

代码清单 3.2 有时候，代码行为天生就是不确定的

```
public class Route {
   private Clock clock = new Clock();
   private ShortestPath algorithm = new ShortestPath();

   public Collection<Directions> directions() {
      if (clock.isRushHour()) {
          return algorithm.avoidBusyIntersections();
      }
      return algorithm.calculateRouteBetween(...);
   }
}
```

在高峰时间计算出的路径大不相同！

这样的话，如果在不同时间执行测试，你如何确保路径算法的正确性？毕竟，算法肯定是从某个时钟获取了时间，尽管在下午 3：40 或 3：50 时算法可能建议走高速公路，但如果现在是下午 3：50，那么最佳结果可能突然就变成了走洲际公路，因为高速公路的晚高峰开始了。

测试替身也可以对这类不确定行为伸出援手。例如，当你的骰子变成可以作弊的测试替身，并能产出一串已知的点数序列时，Craps 游戏的特定实例突然就变得容易模拟了。相类似，如果你用一个固定时刻的测试替身来替换掉系统时钟，你就更容易去描述某个日志文件的预期输出。

控制你的协作者，并在精确设置被测场景时能够消除所有变量，这是使执行变得确定的关键。说到场景，测试替身也能模拟正常情况下不会发生的情况。

3.1.4 模拟特殊情况

我们编写的大多数软件往往是简单粗暴的——至少在某种意义上，大多数代码都是确定的。因此，通过实例化合适的对象图（object graph），并将其作为参数传入被测代码，我们可以重建几乎任何的情况。当我们从“1 Infinite Loop，Cupertino，CA”出发，设置“1600 Amphitheatre Parkway，Mountain View，CA”为终点，然后说 drive()（开车），那么我们可以测试代码清单 3.1 中 Car 最终应该停在正确的地方。

我们无法仅用 API 和产品代码的特性来创建某些情况。假设我们的 Route 通过互联网从 Google 地图来获取路线方向。若是请求方向时互联网连接不幸中断，这种情况下该如何测试 Route 的表现依然正常？

通过禁用计算机的网络接口进行测试，其缺点在于你无法伪造这类网络连接错误，但是若将某处替换为测试替身的话，则可以在请求连接时抛出一个异常。[⊖]

⊖ 虽然编个程序让乐高智力风暴（Lego Mindstorms）机器人去拔掉你的网线肯定是件非常棒的事情。

3.1.5 暴露隐藏的信息

采用测试替身的最后一个（也很重要的）理由，是令我们的测试访问到无法访问的信息。特别是在 Java 上下文中，“暴露信息”首先想到的是允许测试能够读写其他对象的私有成员。尽管有时你决定去那样做[⊖]，但这里的信息指的却是被测代码与其协作者之间的交互。

我们再用可靠的 Car 例子来帮助你掌握这种动态。这是从代码清单 3.1 中复制的 Car 类中的代码片段：

```
public class Car {
   private Engine engine;

   public void start() {
      engine.start();
   }

   // rest omitted for clarity
}
```

如你所见，当某人启动汽车 Car 的时候，汽车 Car 启动它的引擎 Engine。你如何测试它真的发生了？你可以向测试代码暴露私有成员，并为 Engine 增加一个新方法用于判定引擎是否启动了。但是如果你不想那么做的话呢？要是你不想仅仅为了测试而弄乱生产代码呢？

现在你大概猜到了，答案就是测试替身。通过将 Car 的 Engine 替换为测试替身，可以向测试代码中添加仅供测试的方法，避免增加一个永远不会在生产环境中使用的 isRunning() 方法而弄乱你的生产代码。测试代码如代码清单 3.3 所示。

代码清单 3.3 测试替身可以提供内幕消息

```
public class CarTest {
   @Test
   public void engineIsStartedWhenCarStarts() {
      TestEngine engine = new TestEngine();
      new Car(engine).start();
      assertTrue(engine.isRunning());
    }
}

public class TestEngine extends Engine {
   private boolean isRunning;

   public void start() {
      isRunning = true;
   }

   public boolean isRunning() {
      return isRunning;
   }
}
```

❶ 测试替身来帮忙

❷ 方法仅存在于 TestEngine，而非 Engine

⊖ 通常来说，从外部来探测对象的内部并不是一个好主意。每当我觉得需要做类似的事情时，都表明我的设计中隐藏了某个缺失的抽象。

如你所见，我们的示例测试用测试替身❶来配置 Car，㊀启动汽车，使用测试替身来验证引擎如愿启动❷。强调一下，isRunning() 不是 Engine 的方法——它是我们添加到 TestEngine 上的，用于揭示正常 Engine 所不能暴露的信息。㊁

现在你理解了使用测试替身的最常见原因。现在该看看不同类型的测试替身了，以及它们各自所具有的优势。

3.2 测试替身的类型

你见过了使用测试替身的各种原因，我们也暗示了有多种测试替身可供选择。我们来仔细看看那些类型吧。图 3.3 展示了这把大伞下的四种对象。

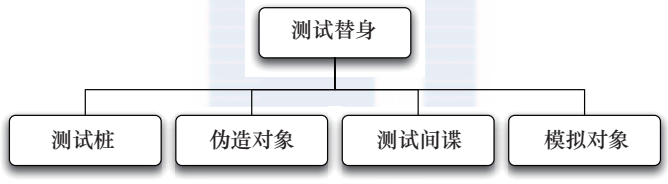

图 3.3 测试替身这把大伞之下聚集了四种对象

既然我们已经制定了测试替身的分类，现在就来认识一下它们，并了解相互的区别，以及运用它们的典型目的。我们先从最简单的开始。

3.2.1 测试桩通常是短小的

我这样来定义它：**桩**（名词），截断的或非常短的物体。

这衍生出测试桩的精确定义。测试桩（简称桩或 Stub）的目的是用最简单的可能实现来代替真实实现。最基本的实现例子就是一个对象的所有方法都只有一行，且它们各自返回一个适当的默认值。

假如你负责的代码应当对自己的操作生成一段审计日志，并通过叫做 Logger 的接口写入远程日志服务器。假如 Logger 接口仅仅定义了一个方法来产生此类日志，那么 Logger 接口的桩看起来是这样：

```
public class LoggerStub implements Logger {
    public void log(LogLevel level, String message) { }
}
```

㊀ 这种通过构造函数传递依赖的风格称为构造函数注入。

㊁ 说到测试替身扩展真实对象的接口，没有理由说你不能在 TestEngine 上增加一个类似 assertIsRunning() 的方法！

有没有注意到 log() 方法其实什么都没做？这是桩的典型例子——什么都不做。毕竟，你正是对真实 Logger 实现打桩，因为你在测试时完全不在乎日志，那么又何必真写日志呢？但是有时候什么都不做也不行。例如，如果 Logger 接口还定义了一个方法来确定当前设置的日志级别（Log Level），那么桩实现看起来可能是这样：

```
public class LoggerStub implements Logger {
   public void log(LogLevel level, String message) {
      // still a no-op
   }

   public LogLevel getLogLevel() {
      return LogLevel.WARN;  // hard-coded return value
   }
}
```

我们在这个类中硬编码了 getLogLevel() 方法，它总是返回 LogLevel.WARN。有没有搞错？大部分情况下这绝对没问题。毕竟，我们有三个充分的理由来使用测试桩代替真实 Logger 实现：

1. 我们的测试不关心被测代码所写的日志。
2. 我们没有运行日志服务器，所以测试会悲剧地失败。
3. 我们也不希望测试套件在控制台中输出大量字节（更别提将所有数据写入文件了）。

简而言之，Logger 桩实现完美地满足了我们的需要。

有时候，简单的硬编码返回语句和一堆空的 void 方法还不够。有时候你至少需要填充一些行为，而有时候你需要测试替身根据收到的消息种类来表现出不同的行为。这些情况下，你会借助**伪造对象**。

3.2.2 伪造对象做事不产生副作用

比起 Stub，伪造对象（简称 Fake）是一种更加复杂的测试替身。Stub 可以返回硬编码值，而每个测试可能需要有差异地实例化来返回不同值，以模拟不同的场景。Fake 更像是真实事物的简单版本，优化地伪造真实事物的行为，但是没有副作用或使用真实事物的其他后果。

持久化对象是采用 Fake 的典型例子。假设应用程序架构是这样的：一些存储对象提供持久化服务，它们知道如何存储和查找指定的对象类型。这种存储对象可能提供的 API 如下：

```
public interface UserRepository {
   void save(User user);
   User findById(long id);
   User findByUsername(String username);
}
```

对于使用存储对象的应用程序，如果没有这种测试替身，测试全都将试图访问真实的数据库。要是对 UserRepository 接口打桩，令其精确地返回测试所需，你就会感觉好一些。但是模拟更复杂的场景肯定会越发复杂。另一方面，由于 UserRepository 接口足够简单，以至于你可以实现一个愚蠢而简单的内存数据库，它只提供基本的数据类型。代码清单 3.4 提供了一个例子。

代码清单 3.4 实现伪造对象不见得那么困难

```
public class FakeUserRepository implements UserRepository {
   private Collection<User> users = new ArrayList<User>();

   public void save(User user) {
      if (findById(user.getId()) == null) {
         users.add(user);
      }
   }

   public User findById(long id) {
      for (User user : users) {
         if (user.getId() == id) return user;
      }
      return null;
   }

   public User findByUsername(String username) {
      for (User user : users) {
         if (user.getUsername().equals(username)) {
            return user;
         }
      }
      return null;
   }
}
```

用这种另类实现来替换真实事物的优点在于，它像只鸭子那样嘎嘎叫，还能摇摆，但它摇摆得比真鸭子要快——即使每次查找一个 User 时都循环一个包含 50 个条目的列表。

测试桩和伪造对象往往是救命稻草，你可以在测试时用它们替换掉缓慢的真实事物，以及鞭长莫及的依赖。然而，这两种基本的测试替身不总是够用。有时你发现自己面对一堵墙，希望自己能像千里眼一样看透它——为了验证代码行为符合预期。那些情况下，你可能会求助于**测试间谍**。

3.2.3 测试间谍偷取秘密

你如何测试下列方法？

```
public String concat(String first, String second) { ... }
```

大多数人会说，把这个那个传进去，然后检查返回值是什么的。那可能没问题。毕竟正确的返回值是你最关心的。那么，下列方法又如何测试？

```
public void filter(List<?> list, Predicate<?> predicate) { ... }
```

这里并没有返回值可以用来断言。这个方法所做的事情是接收一个列表和一个谓词（predicate），过滤列表中不满足谓词的条目。换句话说，验证这个方法正常工作的唯一方式就是事后检查列表。这就像警察卧底，然后汇报她看到的一切。[⊖]通常你不用测试替身也能做

⊖ 我承认，我电视看多了，但是测试间谍就像那样。

到这一点。这个例子中你可以询问 List 对象，看它是否包含你所期望的条目。

至于测试替身——我们正在讨论的测试间谍（简称 Spy）——的方便之处在于，当没有对象作为参数传入时，通过它们的 API 也能揭示你想要了解的知识。代码清单 3.5 显示了这样一个例子。

代码清单 3.5 测试间谍的例子——参数没有为测试提供足够的情报

```
public class DLog {
   private final DLogTarget[] targets;

   public DLog(DLogTarget... targets) {
      this.targets = targets;
   }

   public void write(Level level, String message) {
      for (DLogTarget each : targets) {
         each.write(level, message);
      }
   }
}

public interface DLogTarget {
   void write(Level level, String message);
}
```

❶ 向 DLog 提供了一些 DLogTarget

❷ 每个 target 接收到相同的消息

❸ DLogTarget 仅仅定义了 write() 方法

我们先来看看上述代码清单中的场景。被测对象是一个分布式的日志对象 DLog，代表了一组 DLogTarget ❶。当向 DLog 写入时，你应该向所有 DLogTarget 写入相同的消息❷。从测试的角度来看，事情有点尴尬，你无法知道指定的消息是否被写入，因为 DLogTarget 接口只定义了一个方法 write() ❸，而且 DLogTarget、ConsoleTarget 和 RemoteTarget 的真实实现也都没有提供任何方法。

测试间谍登场了。代码清单 3.6 展示了一个精明的程序员如何鞭打他的女特工去干活。

代码清单 3.6 测试间谍通常很容易实现

```
public class DLogTest {
   @Test
   public void writesEachMessageToAllTargets() throws Exception {
      SpyTarget spy1 = new SpyTarget();
      SpyTarget spy2 = new SpyTarget();
      DLog log = new DLog(spy1, spy2);
      log.write(Level.INFO, "message");
      assertTrue(spy1.received(Level.INFO, "message"));
      assertTrue(spy2.received(Level.INFO, "message"));
   }

   private class SpyTarget implements DLogTarget {
      private List<String> log = new ArrayList<String>();

      @Override
      public void write(Level level, String message) {
         log.add(concatenated(level, message));
      }
```

❶ 间谍潜入

❸ 间谍汇报

❷ 令 write() 留下蛛丝马迹

```
        boolean received(Level level, String message) {
            return log.contains(concatenated(level, message));
        }

        private String concatenated(Level level, String message) {
            return level.getName() + ": " + message;
        }
    }
}
```

让测试来问话

这就是测试间谍的一切。像其他测试替身一样，你将它们传入❶。然后你令测试间谍❷记录已发送的消息，并❸让测试询问测试间谍是否收到指定消息。干得漂亮！

简而言之，测试间谍是一种测试替身，它用于记录过去发生的情况，这样测试在事后就知道所发生的一切。有时我们进一步利用这个概念，于是测试间谍就变成了全能的**模拟对象**。如果测试间谍像个卧底警察，那么模拟对象就像渗入暴民的远程控制机器人。这可能需要一些解释……

3.2.4 模拟对象反对惊喜

模拟对象（简称 Mock）是特殊的 Spy。它是一个在特定情景下可配置行为的对象。例如，UserRepository 接口的模拟对象可能被告之：当带着参数 123 调用 findById() 时要返回 null，而当带着参数 124 调用 findById() 时要返回 User 的一个实例。在这一点上，我们主要讨论的是根据参数来对特定的方法调用打桩。

如果一旦任何意外发生时 Mock 就立即使测试失败，Mock 就能够变得更加精确。例如，假设我们告诉了模拟对象如何应对带着 123 或 124 的 findById() 调用，它就会严格按照指令工作。对于任何其他的调用——不论是调用不同的方法或者带着另外的参数调用 findById()——Mock 就会抛出异常，直接使测试失败。同样，如果 findById() 被调用太多次，Mock 就会抱怨——除非我们告诉它允许调用任意次数——如果预期的调用没发生，Mock 也会抱怨。

包括 JMock、Mockito 和 EasyMock 在内的模拟对象库已经是成熟的工具了，崇尚测试的程序员可以借助它们获得力量。每个库都有自己的行事风格，但基本上你可以用它们中任何一个来完成所有的工作。

这并非模拟对象库的全面教程，但是我们迅速看看代码清单 3.7 中的例子，它展示了这种库的具体用法。这里我们使用 JMock，因为我碰巧有个项目正在使用 JMock。

代码清单 3.7 JMock 允许你在运行时配置 mock

```
public class TestTranslator {
    protected Mockery context;

    @Before
    public void createMockery() throws Exception {
```

```
        context = new JUnit4Mockery();
    }

    @Test
    public void usesInternetForTranslation() throws Exception {
        final Internet internet = context.mock(Internet.class);
        context.checking(new Expectations() {{
            one(internet).get(with(containsString("langpair=en%7Cfi")));
            will(returnValue("{\"translatedText\":\"kukka\"}"));
        }});
        Translator t = new Translator(internet);
        String translation = t.translate("flower", ENGLISH, FINNISH);
        assertEquals("kukka", translation);
    }

    ...
}
```

在这样一小段测试代码中，这个例子展示了许多模拟对象库用法的典型构造。首先，我们告诉库要为指定接口创建一个模拟对象。

在 context.checking() 中看似笨拙的代码块其实是测试在指导模拟的 Internet，告诉它应该期待哪些交互，以及如何应对这些交互。这种情况下，我们预期测试会带着包含 "langpair=en%7Cfi" 字符串的参数调用 get() 方法一次，对此，mock 应当返回指定字符串。

最终，我们将 Mock 传给被测的 Translator 对象，执行 Translator，然后断言 Translator 为我们的场景提供了正确的翻译。

然而，这并非我们的全部断言。如前所述，Mock 可以严格地判断已经发生的预期交互。在模拟 Internet 的例子中，Mock 严格地断言它确实收到了一次带有指定子字符串参数的 get() 方法调用。

3.3 使用测试替身的指南

测试替身是程序员的工具，就像木匠的锤子和钉子。存在敲钉子的适当方式，当然也有不恰当的方式——最好是能把它们识别出来。

先从我认为最重要的指南开始吧，当你求助于测试替身时要时刻牢记它——从你的工具箱中选择合适的工具。

3.3.1 为测试挑选合适的替身

有许多测试替身可供选择，它们看起来各有千秋。采用它们的最佳条件是什么？到底应该选择哪个？

这里并没有太多的硬性规定，但一般来说你应该因地制宜地混合使用。我是说，某些情

况下你只想要“一个返回 5 的对象”，而其他情况下你特别想知道某个方法被调用过。有时在一个测试中对两者都感兴趣，于是你将 Stub、Fake 和 Mock 一同使用。

前面已经说过，并没有清晰的原则来决定采用哪种方式以得到最可读的测试。但我还是忍不住对如何选择这个问题阐述一些逻辑和启发：

- 如果你关心某些交互，即两个对象之间的方法调用，你可能会需要一个模拟对象 Mock。
- 如果你决定使用 Mock，但测试代码最终看起来不像你想的那样漂亮，那就看看一个手工的简单测试间谍 Spy 能否满足需要。
- 如果你只关心协作对象向被测对象输送的响应，用桩 Stub 就可以。[⊖]
- 如果你想运行一个复杂场景，其中它所依赖的服务或组件无法供测试使用，而你对所有交互打桩的快速尝试却戛然而止，或产出了难以维护的糟糕的测试代码，那就考虑实现一个伪造对象 Fake 吧。
- 如果上述都不能满足你手上的特殊情况，那就抛硬币吧——正面代表 Mock，反面代表 Stub，如果硬币直立，我允许你找一个 Fake 帮你干活。

如果觉得那个列表太难记，别怕。《JUnit Recipes》（Manning，2004）的作者 J. B. Rainsberger 有一个简单的记忆规则，用于选择正确的测试替身类型：Stub 管查询，Mock 管操作。现在我们顺利地得到启发，知道什么时候该用哪种测试替身了，接下来看看如何使用它们。

3.3.2 准备、执行、断言

关于编码约定（convention），我要说几句。问题是各种标准太多了。幸运的是，当你构造单元测试时，存在一个大多数程序员都认为合理的、相当确定的实践。它叫做准备 – 执行 – 断言（Arrange-Act-Assert），这种组织测试的方式基本上是这样的，先准备用于测试的对象，然后触发执行，最后对输出进行断言。

代码清单 3.8 复制了代码清单 3.7 的测试，我们看它如何符合这种约定来组织测试方法。

代码清单 3.8 准备 – 执行 – 断言使结构变得清晰

```
@Test
public void usesInternetForTranslation() throws Exception {
    final Internet internet = context.mock(Internet.class);
    context.checking(new Expectations() {{
        one(internet).get(with(containsString("langpair=en%7Cfi")));
        will(returnValue("{\"translatedText\":\"kukka\"}"));
    }});
    Translator t = new Translator(internet);
```

❶ 准备

⊖ 如果你决定使用 Stub，但你的测试中也有 Mock，那就考虑使用你选择的模拟对象库来创建 Stub——它也能做到那样，而且会使测试代码更养眼。

```
    String translation = t.translate("flower", ENGLISH, FINNISH);   <-- ❷ 执行
    assertEquals("kukka", translation);                              <-- ❸ 断言
}
```

注意我在三段代码之间增加空白的方式。这用来强调三段代码的不同角色。㊀

测试的前五行是准备❶所要用到的协作对象。虽然其中我们只涉及 Internet 接口的一个 Mock，但是在测试开头设置多个协作者的情况也很常见。然后是被测对象 Translator——对它的实例化也是准备工作的一部分。㊁

下一段代码中，我们调用 translation ❷（被测的翻译功能），最后，不论预期输出是直接输出还是造成的副作用，我们都对它进行断言❸。

给定 – 当 – 那么（Given，When，Then）

行为驱动开发运动所推广的词汇和结构与“准备 – 执行 – 断言”很像：给定（某个上下文），当（发生某些事情），那么（期望某些结果）。这个想法以更加直观的语言来指定预期行为，尽管“准备 – 执行 – 断言”更好记，但“给定 – 当 – 那么”更流畅，使人们更加自然地思考行为（而不是实现细节）。

这种结构相当普遍，它有助于使测试保持专注。如果感觉三部分中某一部分很“大”，那就是一个信号，表明测试可能试图做太多事情，需要更加专注。既然说到这话题，咱们就简单讨论一下测试应该专注什么。

3.3.3 检查行为，而非实现

人都会犯错误。模拟对象库新手常犯的一个错误是过度细致地对 Mock 设置期望。我指的是在测试中，对测试可能涉及的每个对象都做 Mock，每个对象间的方法调用都严格指定。

是的，某种意义上，测试给予我们确定性，只要有任何变更它就会中断并报警。而这也是问题所在——即使是最小的变更，哪怕它与测试所要验证的不相关，也会中断测试。好比在一片口香糖上密密麻麻地敲了许多钉子，使之动弹不得。

这种测试的基本问题是缺乏专注。一个测试应当只测试一件事情，并好好地测试，清晰地沟通自己的意图。看着被测对象，你要问自己到底什么是想要验证的预期行为？至于实现细节，倒是并不需要钉在我们的测试中。

㊀ 尽管空白肯定会被滥用，但一致地使用空白有助于程序员更容易地看清测试的结构。

㊁ 技术上讲，为 Mock 设置期望是一种验证行为——是断言，而不是准备工作。但这里将 Mock 用于 Stub。有人说这是歪门邪道——他们说得对。

预期行为应该配置在 Mock 对象的期望中。应该寻求通过 Stub 或非严格 Mock 来提供实现细节，它们不介意交互从未发生或者发生多次。

检查行为，而非实现。当你掏出喜爱的模拟对象库时，你应该牢牢记住这一点。说到这里……

3.3.4 挑选你的工具

说到模拟对象库，Java 程序员真是占了大便宜——有太多可以选择的。像我之前提到的，你几乎可以用任何先进的库来做同样的事情，但是它们在 API 方面还是有些细微的区别以及一些独特的功能，从而在满足某些特定方向和需求时能够一锤定音。

或许其中最独特的功能就是 Mockito 的**打桩**与**验证**分离。这得细说，接下来看个例子，我用 Mockito 重写了之前采用 JMock 的测试：

```
@Test
public void usesInternetForTranslation() throws Exception {
   final Internet internet = mock(Internet.class);
   when(internet.get(argThat(containsString("langpair=en%7Cfi"))))
               .thenReturn("{\"translatedText\":\"kukka\"}");
   Translator translator = new Translator(internet);
   String result = translator.translate("flower", ENGLISH, FINNISH);
   assertEquals("kukka", result);
}
```

Mockito 的 API 比 JMock 更简洁。除此之外，看起来差不多，是不是？是的，只是这个用 Mockito 写的测试仅仅对方法 get() 打桩——即使交互从未发生，它也会成功通过。如果我们真的希望验证 Translator 使用 Internet 的行为，我们就得增加一个对 Mockito API 的调用来进行检查：

```
verify(internet).get(argThat(containsString("langpair=en%7Cfi")));
```

或者，如果感觉不必那么精确：

```
verify(internet).get(anyString());
```

模拟对象库 API 通常是个人喜好问题。但是 Mockito 在测试风格上有一个明显的优势，那就是主要依赖于打桩——在你的特定上下文中这可能是优势，也可能不是。测试代码每天都保持可读、简洁、可维护，这才是关键。这值得停下来权衡一下，明智地选择工具。

咱们再次借用 J. B. 的话来明确 JMock 与 Mockito 在方式和适用条件方面的区别：[㊀]

> 当我想要拯救遗留代码时，我选择 Mockito。当我想要设计新功能时，我选择 JMock。
>
> JMock 与 Mockito 不同的前提假设，使得两者擅长不同的任务。默认情况下，

㊀ J. B. Rainsbergerblog，《JMock v. Mockito，but Not to the Death》，2010.10.5，http://mng.bz/yW2m.

JMock认为测试替身（Mock）期望着客户不会在任何时候调用任何方法。如果你想放宽这个假设，你就得增加一个stub。另一方面，Mockito认为测试替身（也叫Mock）允许客户在任何时候调用任何方法。如果你想加强这个假设，那么你就得验证某个方法的调用。这就是区别所在。

不论你决定选择哪个库，我们的第三个即最后一个测试替身指南全都适用。

3.3.5 注入依赖

为了能够使用测试替身，你需要一种替换真实事物的方法。当涉及依赖时——为了测试目的而替换协作对象——我们的指南建议不要在同一个地方同时实例化和使用它们。在实践中，这意味着将这些对象另存为私有成员，或借助工厂方法来获取它们。

一旦你隔离开依赖，你就需要访问它。你可以用可见性修饰符来破坏封装——将私有（private）内容变成公开（public）或者包级私有（package private）——或使用反射API来将测试替身分配给私有字段。那种方式很快就会变得丑陋。更好的选择是采用依赖注入，从外部将依赖传递给对象，通常使用构造函数注入，正如在Translator例子中那样。㊀

我们对于测试替身说得够多了，我渴望进行第二部分了，接下来对本章学到的东西做一个回顾吧。

3.4 小结

我们从使用测试替身的理由开始，先探讨了测试替身的话题。这些原因背后的共同点是需要隔离被测代码，这样你就能模拟出所有场景，并且测试到代码应该表现出的所有行为。

有时，使用测试替身的理由是希望测试运行得更快。采用简单实现的测试替身，通常比它们替换掉的实现要快一到两倍。有时，被测代码依赖于随机或其他不确定的行为，比如时间。这些情况下，测试替身通过将不可预测变得可预测而使测试变得简单。

对于模拟某些特殊情况并验证对象的预期行为，测试替身可能是仅有可行方式，而不必仅为了可测性而修改产品代码的设计或暴露细节。

除了测试替身带来的好处，我们继续学习了四种测试替身之间的区别：测试桩、伪造对象、测试间谍、模拟对象。极度简约的桩或许最适合切断不相关的协作者。当真实事物难以使用或比较麻烦时，伪造对象提供了闪电般的变换。测试间谍用来访问隐藏的信息和数据，而模拟对象像打了激素的测试间谍，增加了动态配置行为的能力，并验证预期的交互确实会发生。

㊀ 注意，依赖注入并不意味着必须使用一个依赖注入框架。通过构造函数简单地将依赖传入就够了！

我们从各种测试替身使用的时机和场景开始，以一些测试替身的基本使用指南来结束本章。尤其是，当使用模拟对象时，避免过紧地钉住实现是很重要的。你的 Mock 应当验证预期行为，而尽量放过行为的具体实现。

通常，工具可以提供极大的帮助，而且值得为你的特殊需要以及模拟对象的使用风格来评估最合适的库。最后，选择从外部注入依赖，而不是从被测代码内部硬连接它们，将会带来截然不同的可测性。

本章讲了测试替身，第一部分也到此结束。我们现在探讨了编写良好测试的基础。先是理解编写测试的好处，然后了解测试应该具有的属性，接下来操控程序员最基本的测试工具——测试替身——你现在掌握了足够的基本技能，你已经可以开始进一步磨炼了。特别是，你已经准备好开始训练对各种不良测试的感觉，从表现良好到令人头痛，到地板上的臭袜子，甚至是彻头彻尾的维护负担。该是进入第二部分并关注坏味道的时候了。

那正是测试坏味道。