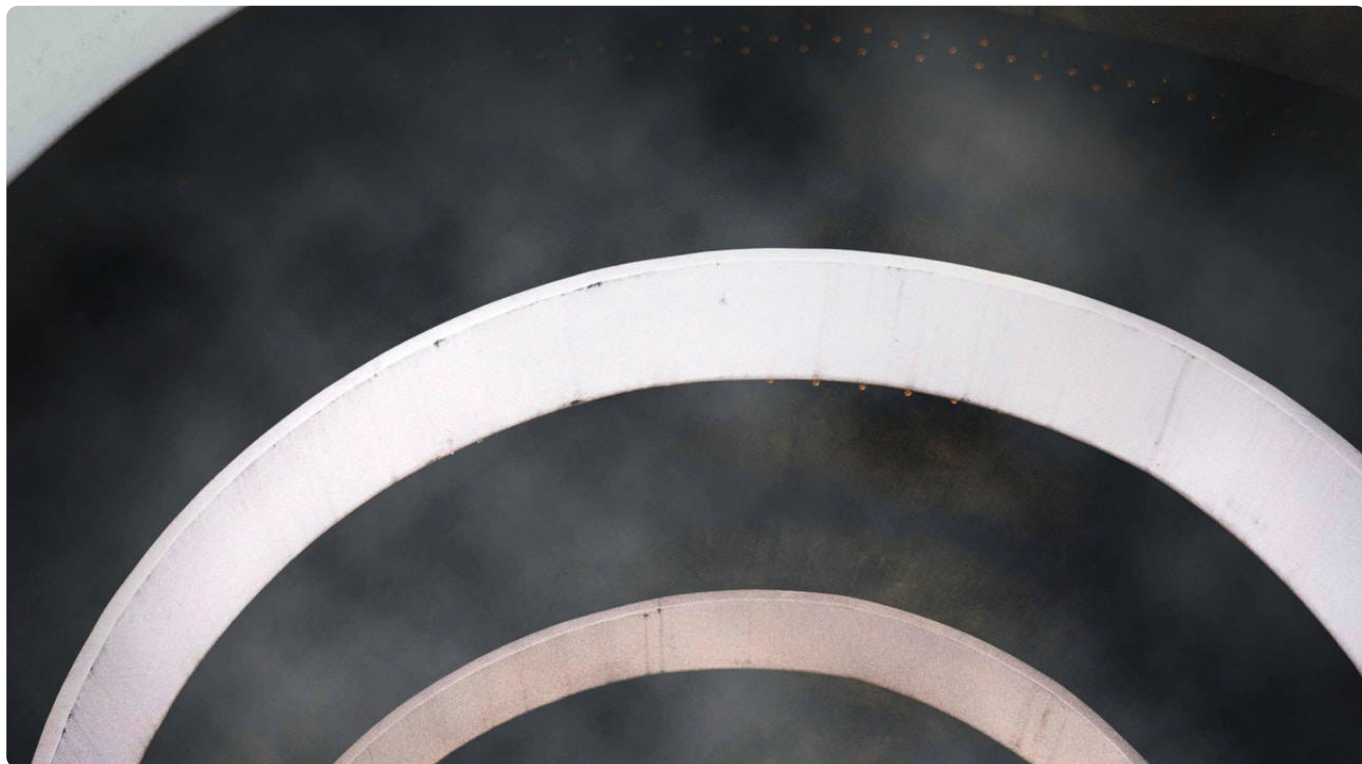


## 29 | 理论三：什么是代码的可测试性？如何写出可测试性好的代码？

2020-01-08 王争

设计模式之美

[进入课程 >](#)



**讲述：冯永吉**

时长 18:22 大小 14.73M



在上一节课中，我们对单元测试做了介绍，讲了“什么是单元测试？为什么要编写单元测试？如何编写单元测试？实践中单元测试为什么难贯彻执行？”这样几个问题。

实际上，写单元测试并不难，也不需要太多技巧，相反，写出可测试的代码反倒是件非常有挑战的事情。所以，今天，我们就再来聊一聊代码的可测试性，主要包括这样几个问题：

什么是代码的可测试性？

如何写出可测试的代码？

有哪些常见的不好测试的代码？

话不多说，让我们正式开始今天的学习吧！

## 编写可测试代码案例实战

刚刚提到的这几个关于代码可测试性的问题，我准备通过一个实战案例来讲解。具体的被测测试代码如下所示。

其中，Transaction 是经过我抽象简化之后的一个电商系统的交易类，用来记录每笔订单交易的情况。Transaction 类中的 execute() 函数负责执行转账操作，将钱从买家的钱包转到卖家的钱包中。真正的转账操作是通过调用 WalletRpcService RPC 服务来完成的。除此之外，代码中还涉及一个分布式锁 DistributedLock 单例类，用来避免 Transaction 并发执行，导致用户的钱被重复转出。

 复制代码

```
1 public class Transaction {
2     private String id;
3     private Long buyerId;
4     private Long sellerId;
5     private Long productId;
6     private String orderId;
7     private Long createTimeStamp;
8     private Double amount;
9     private STATUS status;
10    private String walletTransactionId;
11
12    // ...get() methods...
13
14    public Transaction(String preAssignedId, Long buyerId, Long sellerId, Long p
15        if (preAssignedId != null && !preAssignedId.isEmpty()) {
16            this.id = preAssignedId;
17        } else {
18            this.id = IdGenerator.generateTransactionId();
19        }
20        if (!this.id.startsWith("t_")) {
21            this.id = "t_" + preAssignedId;
22        }
23        this.buyerId = buyerId;
24        this.sellerId = sellerId;
25        this.productId = productId;
26        this.orderId = orderId;
27        this.status = STATUS.TO_BE_EXECUTD;
28        this.createTimeStamp = System.currentTimeMillis();
29    }
30
31    public boolean execute() throws InvalidTransactionException {
```

```

32     if ((buyerId == null || (sellerId == null || amount < 0.0)) {
33         throw new InvalidTransactionException(...);
34     }
35     if (status == STATUS.EXECUTED) return true;
36     boolean isLocked = false;
37     try {
38         isLocked = RedisDistributedLock.getSingletonIntance().lockTransction(id)
39         if (!isLocked) {
40             return false; // 锁定未成功, 返回 false, job 兜底执行
41         }
42         if (status == STATUS.EXECUTED) return true; // double check
43         long executionInvokedTimestamp = System.currentTimeMillis();
44         if (executionInvokedTimestamp - createdTimestamp > 14days) {
45             this.status = STATUS.EXPIRED;
46             return false;
47         }
48         WalletRpcService walletRpcService = new WalletRpcService();
49         String walletTransactionId = walletRpcService.moveMoney(id, buyerId, sel
50         if (walletTransactionId != null) {
51             this.walletTransactionId = walletTransactionId;
52             this.status = STATUS.EXECUTED;
53             return true;
54         } else {
55             this.status = STATUS.FAILED;
56             return false;
57         }
58     } finally {
59         if (isLocked) {
60             RedisDistributedLock.getSingletonIntance().unlockTransction(id);
61         }
62     }
63 }
64 }

```

对比上一节课中的 Text 类的代码，这段代码要复杂很多。如果让你给这段代码编写单元测试，你会如何来写呢？你可以先试着思考一下，然后再来看我下面的分析。

在 Transaction 类中，主要逻辑集中在 execute() 函数中，所以它是我们测试的重点对象。为了尽可能全面覆盖各种正常和异常情况，针对这个函数，我设计了下面 6 个测试用例。

1. 正常情况下，交易执行成功，回填用于对账（交易与钱包的交易流水）用的 walletTransactionId，交易状态设置为 EXECUTED，函数返回 true。
2. buyerId、sellerId 为 null、amount 小于 0，返回 InvalidTransactionException。

3. 交易已过期 (createTimestamp 超过 14 天) , 交易状态设置为 EXPIRED, 返回 false。
4. 交易已经执行了 (status==EXECUTED) , 不再重复执行转钱逻辑, 返回 true。
5. 钱包 (WalletRpcService) 转钱失败, 交易状态设置为 FAILED, 函数返回 false。
6. 交易正在执行着, 不会被重复执行, 函数直接返回 false。

测试用例设计完了。现在看起来似乎一切进展顺利。但是, 事实是, 当我们将测试用例落实到具体的代码实现时, 你就会发现有很多行不通的地方。对于上面的测试用例, 第 2 个实现起来非常简单, 我就不做介绍了。我们重点来看其中的 1 和 3。测试用例 4、5、6 跟 3 类似, 留给你自己来实现。

现在, 我们就来看测试用例 1 的代码实现。具体如下所示:

 复制代码

```
1 public void testExecute() {
2     Long buyerId = 123L;
3     Long sellerId = 234L;
4     Long productId = 345L;
5     Long orderId = 456L;
6     Transaction transaction = new Transaction(null, buyerId, sellerId, productId,
7     boolean executedResult = transaction.execute();
8     assertTrue(executedResult);
9 }
```

execute() 函数的执行依赖两个外部的服务, 一个是 RedisDistributedLock, 一个 WalletRpcService。这就导致上面的单元测试代码存在下面几个问题。

如果让这个单元测试能够运行, 我们需要搭建 Redis 服务和 Wallet RPC 服务。搭建和维护的成本比较高。

我们还需要保证将伪造的 transaction 数据发送给 Wallet RPC 服务之后, 能够正确返回我们期望的结果, 然而 Wallet RPC 服务有可能是第三方 (另一个团队开发维护的) 的服务, 并不是我们可控的。换句话说, 并不是我们想让它返回什么数据就返回什么。


Transaction 的执行跟 Redis、RPC 服务通信, 需要走网络, 耗时可能会比较长, 对单元测试本身的执行性能也会有影响。

网络的中断、超时、Redis、RPC 服务的不可用，都会影响单元测试的执行。

我们回到单元测试的定义上来看一下。单元测试主要是测试程序员自己编写的代码逻辑的正确性，并非是端到端的集成测试，它不需要测试所依赖的外部系统（分布式锁、Wallet RPC 服务）的逻辑正确性。所以，如果代码中依赖了外部系统或者不可控组件，比如，需要依赖数据库、网络通信、文件系统等，那我们就需要将测试代码与外部系统解依赖，而这种解依赖的方法就叫作“mock”。所谓的 mock 就是用一个“假”的服务替换真正的服务。mock 的服务完全在我们的控制之下，模拟输出我们想要的结果。

那如何来 mock 服务呢？mock 的方式主要有两种，手动 mock 和利用框架 mock。利用框架 mock 仅仅是为了简化代码编写，每个框架的 mock 方式都不大一样。我们这里只展示手动 mock。

我们通过继承 WalletRpcService 类，并且重写其中的 moveMoney() 函数的方式来实现 mock。具体的代码实现如下所示。通过 mock 的方式，我们可以让 moveMoney() 返回任意我们想要的结果，完全在我们的控制范围内，并且不需要真正进行网络通信。

 复制代码

```
1 public class MockWalletRpcServiceOne extends WalletRpcService {
2     public String moveMoney(Long id, Long fromUserId, Long toUserId, Double amount) {
3         return "123bac";
4     }
5 }
6
7 public class MockWalletRpcServiceTwo extends WalletRpcService {
8     public String moveMoney(Long id, Long fromUserId, Long toUserId, Double amount) {
9         return null;
10    }
11 }
```

现在我们再来看，如何用 MockWalletRpcServiceOne、MockWalletRpcServiceTwo 来替换代码中的真正的 WalletRpcService 呢？

因为 WalletRpcService 是在 execute() 函数中通过 new 的方式创建的，我们无法动态地对其进行替换。也就是说，Transaction 类中的 execute() 方法的可测试性很差，需要通过重构来让其变得更容易测试。该如何重构这段代码呢？

在 [第 19 节](#) 中，我们讲到，依赖注入是实现代码可测试性的最有效的手段。我们可以应用依赖注入，将 WalletRpcService 对象的创建反转给上层逻辑，在外部创建好之后，再注入到 Transaction 类中。重构之后的 Transaction 类的代码如下所示：

 复制代码

```
1 public class Transaction {
2     //...
3     // 添加一个成员变量及其 set 方法
4     private WalletRpcService walletRpcService;
5
6     public void setWalletRpcService(WalletRpcService walletRpcService) {
7         this.walletRpcService = walletRpcService;
8     }
9     // ...
10    public boolean execute() {
11        // ...
12        // 删除下面这一行代码
13        // WalletRpcService walletRpcService = new WalletRpcService();
14        // ...
15    }
16 }
```

现在，我们就可以在单元测试中，非常容易地将 WalletRpcService 替换成 MockWalletRpcServiceOne 或 WalletRpcServiceTwo 了。重构之后的代码对应的单元测试如下所示：

 复制代码


```
1 public void testExecute() {
2     Long buyerId = 123L;
3     Long sellerId = 234L;
4     Long productId = 345L;
5     Long orderId = 456L;
6     Transaction transaction = new Transaction(null, buyerId, sellerId, productId,
7     // 使用 mock 对象来替代真正的 RPC 服务
8     transaction.setWalletRpcService(new MockWalletRpcServiceOne());
9     boolean executedResult = transaction.execute();
10    assertTrue(executedResult);
11    assertEquals(STATUS.EXECUTED, transaction.getStatus());
12 }
```

WalletRpcService 的 mock 和替换问题解决了，我们再来看 RedisDistributedLock。它的 mock 和替换要复杂一些，主要是因为 RedisDistributedLock 是一个单例类。单例相当

于一个全局变量，我们无法 mock（无法继承和重写方法），也无法通过依赖注入的方式来替换。

如果 RedisDistributedLock 是我们自己维护的，可以自由修改、重构，那我们可以将其改为非单例的模式，或者定义一个接口，比如 IDistributedLock，让 RedisDistributedLock 实现这个接口。这样我们就可以像前面 WalletRpcService 的替换方式那样，替换 RedisDistributedLock 为 MockRedisDistributedLock 了。但如果 RedisDistributedLock 不是我们维护的，我们无权去修改这部分代码，这个时候该怎么办呢？


我们可以对 transaction 上锁这部分逻辑重新封装一下。具体代码实现如下所示：

 复制代码

```
1 public class TransactionLock {
2     public boolean lock(String id) {
3         return RedisDistributedLock.getSingletonIntance().lockTransction(id);
4     }
5
6     public void unlock() {
7         RedisDistributedLock.getSingletonIntance().unlockTransction(id);
8     }
9 }
10
11 public class Transaction {
12     //...
13     private TransactionLock lock;
14
15     public void setTransactionLock(TransactionLock lock) {
16         this.lock = lock;
17     }
18
19     public boolean execute() {
20         //...
21         try {
22             isLocked = lock.lock();
23             //...
24         } finally {
25             if (isLocked) {
26                 lock.unlock();
27             }
28         }
29         //...
30     }
31 }
```




针对重构过的代码，我们的单元测试代码修改为下面这个样子。这样，我们就能在单元测试代码中隔离真正的 RedisDistributedLock 分布式锁这部分逻辑了。

 复制代码

```
1 public void testExecute() {
2     Long buyerId = 123L;
3     Long sellerId = 234L;
4     Long productId = 345L;
5     Long orderId = 456L;
6
7     TransactionLock mockLock = new TransactionLock() {
8         public boolean lock(String id) {
9             return true;
10        }
11
12        public void unlock() {}
13    };
14
15    Transaction transaction = new Transaction(null, buyerId, sellerId, productId,
16    transaction.setWalletRpcService(new MockWalletRpcServiceOne());
17    transaction.setTransactionLock(mockLock);
18    boolean executedResult = transaction.execute();
19    assertTrue(executedResult);
20    assertEquals(STATUS.EXECUTED, transaction.getStatus());
21 }
```

至此，测试用例 1 就算写好了。我们通过依赖注入和 mock，让单元测试代码不依赖任何不可控的外部服务。你可以照着这个思路，自己写一下测试用例 4、5、6。

现在，我们再来看测试用例 3：交易已过期（createTimestamp 超过 14 天），交易状态设置为 EXPIRED，返回 false。针对这个单元测试用例，我们还是先把代码写出来，然后再来分析。

 复制代码

```
1 public void testExecute_with_TransactionIsExpired() {
2     Long buyerId = 123L;
3     Long sellerId = 234L;
4     Long productId = 345L;
5     Long orderId = 456L;
6     Transaction transaction = new Transaction(null, buyerId, sellerId, productId,
7     transaction.setCreatedTimestamp(System.currentTimeMillis() - 14days);
8     boolean actualResult = transaction.execute();
9     assertFalse(actualResult);
10    assertEquals(STATUS.EXPIRED, transaction.getStatus());
11 }
```



上面的代码看似没有任何问题。我们将 transaction 的创建时间 createdTimestamp 设置为 14 天前，也就是说，当单元测试代码运行的时候，transaction 一定是处于过期状态。但是，如果在 Transaction 类中，并没有暴露修改 createdTimestamp 成员变量的 set 方法（也就是没有定义 setCreatedTimestamp() 函数）呢？

你可能会说，如果没有 createTimestamp 的 set 方法，我就重新添加一个呗！实际上，这违反了类的封装特性。在 Transaction 类的设计中，createTimestamp 是在交易生成时（也就是构造函数中）自动获取的系统时间，本来就不应该人为地轻易修改，所以，暴露 createTimestamp 的 set 方法，虽然带来了灵活性，但也带来了不可控性。因为，我们无法控制使用者是否会调用 set 方法重设 createTimestamp，而重设 createTimestamp 并非我们的预期行为。

那如果没有针对 createTimestamp 的 set 方法，那测试用例 3 又该如何实现呢？实际上，这是一类比较常见的问题，就是代码中包含跟“时间”有关的“未决行为”逻辑。我们一般的处理方式是将这种未决行为逻辑重新封装。针对 Transaction 类，我们只需要将交易是否过期的逻辑，封装到 isExpired() 函数中即可，具体的代码实现如下所示：

[📄 复制代码](#)

```
1 public class Transaction {
2
3     protected boolean isExpired() {
4         long executionInvokedTimestamp = System.currentTimeMillis();
5         return executionInvokedTimestamp - createdTimestamp > 14days;
6     }
7
8     public boolean execute() throws InvalidTransactionException {
9         //...
10        if (isExpired()) {
11            this.status = STATUS.EXPIRED;
12            return false;
13        }
14        //...
15    }
16 }
```

针对重构之后的代码，测试用例 3 的代码实现如下所示：

```
1 public void testExecute_with_TransactionIsExpired() {
2     Long buyerId = 123L;
3     Long sellerId = 234L;
4     Long productId = 345L;
5     Long orderId = 456L;
6     Transaction transaction = new Transaction(null, buyerId, sellerId, productId,
7         protected boolean isExpired() {
8             return true;
9         }
10    };
11    boolean actualResult = transaction.execute();
12    assertFalse(actualResult);
13    assertEquals(STATUS.EXPIRED, transaction.getStatus());
14 }
```

通过重构，Transaction 代码的可测试性提高了。之前罗列的所有测试用例，现在我们都顺利实现了。不过，Transaction 类的构造函数的设计还有点不妥。为了方便你查看，我把构造函数的代码重新 copy 了一份贴到这里。

```
1 public Transaction(String preAssignedId, Long buyerId, Long sellerId, Long p
2     if (preAssignedId != null && !preAssignedId.isEmpty()) {
3         this.id = preAssignedId;
4     } else {
5         this.id = IdGenerator.generateTransactionId();
6     }
7     if (!this.id.startsWith("t_")) {
8         this.id = "t_" + preAssignedId;
9     }
10    this.buyerId = buyerId;
11    this.sellerId = sellerId;
12    this.productId = productId;
13    this.orderId = orderId;
14    this.status = STATUS.TO_BE_EXECUTD;
15    this.createTimestamp = System.currentTimeMillis();
16 }
```

我们发现，构造函数中并非只包含简单赋值操作。交易 id 的赋值逻辑稍微复杂。我们最好也要测试一下，以保证这部分逻辑的正确性。为了方便测试，我们可以把 id 赋值这部分逻辑单独抽象到一个函数中，具体的代码实现如下所示：

```
1 public Transaction(String preAssignedId, Long buyerId, Long sellerId, Long p
2     //...
3     fillTransactionId(preAssignId);
4     //...
5 }
6
7 protected void fillTransactionId(String preAssignedId) {
8     if (preAssignedId != null && !preAssignedId.isEmpty()) {
9         this.id = preAssignedId;
10    } else {
11        this.id = IdGenerator.generateTransactionId();
12    }
13    if (!this.id.startsWith("t_")) {
14        this.id = "t_" + preAssignedId;
15    }
16 }
```

到此为止，我们一步一步将 Transaction 从不可测试代码重构成了测试性良好的代码。不过，你可能还会有疑问，Transaction 类中 isExpired() 函数就不用测试了吗？对于 isExpired() 函数，逻辑非常简单，肉眼就能判定是否有 bug，是可以不用写单元测试的。

实际上，可测试性差的代码，本身代码设计得也不够好，很多地方都没有遵守我们之前讲到的设计原则和思想，比如“基于接口而非实现编程”思想、依赖反转原则等。重构之后的代码，不仅可测试性更好，而且从代码设计的角度来说，也遵从了经典的设计原则和思想。这也印证了我们之前说过的，代码的可测试性可以从侧面上反应代码设计是否合理。除此之外，在平时的开发中，我们也要多思考一下，这样编写代码，是否容易编写单元测试，这也利于我们设计出好的代码。


## 其他常见的 Anti-Patterns

刚刚我们通过一个实战案例，讲解了如何利用依赖注入来提高代码的可测试性，以及编写单元测试中最复杂的一部分内容：如何通过 mock、二次封装等方式解依赖外部服务。现在，我们再来总结一下，有哪些典型的、常见的测试性不好的代码，也就是我们常说的 Anti-Patterns。

### 1. 未决行为

所谓的未决行为逻辑就是，代码的输出是随机或者说不确定的，比如，跟时间、随机数有关的代码。对于这一点，在刚刚的实战案例中我们已经讲到，你可以利用刚才讲到的方法，试

着重重构一下下面的代码，并且为它编写单元测试。


 复制代码

```
1 public class Demo {
2     public long caculateDelayDays(Date dueTime) {
3         long currentTimestamp = System.currentTimeMillis();
4         if (dueTime.getTime() >= currentTimestamp) {
5             return 0;
6         }
7         long delayTime = currentTimestamp - dueTime.getTime();
8         long delayDays = delayTime / 86400;
9         return delayDays;
10    }
11 }
```

## 2. 全局变量

前面我们讲过，全局变量是一种面向过程的编程风格，有种种弊端。实际上，滥用全局变量也让编写单元测试变得困难。我举个例子来解释一下。

RangeLimiter 表示一个  $[-5, 5]$  的区间，position 初始在 0 位置，move() 函数负责移动 position。其中，position 是一个静态全局变量。RangeLimiterTest 类是为其设计的单元测试，不过，这里面存在很大的问题，你可以先自己分析一下。

 复制代码

```
1 public class RangeLimiter {
2     private static AtomicInteger position = new AtomicInteger(0);
3     public static final int MAX_LIMIT = 5;
4     public static final int MIN_LIMIT = -5;
5
6     public boolean move(int delta) {
7         int currentPos = position.addAndGet(delta);
8         boolean betweenRange = (currentPos <= MAX_LIMIT) && (currentPos >= MIN_LIMIT);
9         return betweenRange;
10    }
11 }
12
13 public class RangeLimiterTest {
14     public void testMove_betweenRange() {
15         RangeLimiter rangeLimiter = new RangeLimiter();
16         assertTrue(rangeLimiter.move(1));
17         assertTrue(rangeLimiter.move(3));
18         assertTrue(rangeLimiter.move(-5));
19     }
20 }
```

```
20     public void testMove_exceedRange() {
21         RangeLimiter rangeLimiter = new RangeLimiter();
22         assertFalse(rangeLimiter.move(6));
23     }
24 }
25
```

上面的单元测试有可能会运行失败。假设单元测试框架顺序依次执行 `testMove_betweenRange()` 和 `testMove_exceedRange()` 两个测试用例。在第一个测试用例执行完成之后，`position` 的值变成了 `-1`；再执行第二个测试用例的时候，`position` 变成了 `5`，`move()` 函数返回 `true`，`assertFalse` 语句判定失败。所以，第二个测试用例运行失败。

当然，如果 `RangeLimiter` 类有暴露重设（reset）`position` 值的函数，我们可以在每次执行单元测试用例之前，把 `position` 重设为 `0`，这样就能解决刚刚的问题。

不过，每个单元测试框架执行单元测试用例的方式可能是不同的。有的是顺序执行，有的是并发执行。对于并发执行的情况，即便我们每次都把 `position` 重设为 `0`，也并不奏效。如果两个测试用例并发执行，第 16、17、18、23 这四行代码可能会交叉执行，影响到 `move()` 函数的执行结果。

### 3. 静态方法

前面我们也提到，静态方法跟全局变量一样，也是一种面向过程的编程思维。在代码中调用静态方法，有时候会导致代码不易测试。主要原因是静态方法也很难 mock。但是，这个要分情况来看。只有在这个静态方法执行耗时太长、依赖外部资源、逻辑复杂、行为未决等情况下，我们才需要在单元测试中 mock 这个静态方法。除此之外，如果只是类似 `Math.abs()` 这样的简单静态方法，并不会影响代码的可测试性，因为本身并不需要 mock。

### 4. 复杂继承

我们前面提到，相比组合关系，继承关系的代码结构更加耦合、不灵活，更加不易扩展、不易维护。实际上，继承关系也更加难测试。这也印证了代码的可测试性跟代码质量的相关性。

如果父类需要 mock 某个依赖对象才能进行单元测试，那所有的子类、子类的子类.....在编写单元测试的时候，都要 mock 这个依赖对象。对于层次很深（在继承关系类图中表现为纵向深度）、结构复杂（在继承关系类图中表现为横向广度）的继承关系，越底层的子类要 mock 的对象可能就会越多，这样就会导致，底层子类在写单元测试的时候，要一个一个 mock 很多依赖对象，而且还需要查看父类代码，去了解该如何 mock 这些依赖对象。

如果我们利用组合而非继承来组织类之间的关系，类之间的结构层次比较扁平，在编写单元测试的时候，只需要 mock 类所组合依赖的对象即可。

## 5. 高耦合代码

如果一个类职责很重，需要依赖十几个外部对象才能完成工作，代码高度耦合，那我们在编写单元测试的时候，可能需要 mock 这十几个依赖的对象。不管是从代码设计的角度来说，还是从编写单元测试的角度来说，这都是不合理的。

## 重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

### 1. 什么是代码的可测试性？

粗略地讲，所谓代码的可测试性，就是针对代码编写单元测试的难易程度。对于一段代码，如果很难为其编写单元测试，或者单元测试写起来很费劲，需要依靠单元测试框架中很高级的特性，那往往就意味着代码设计得不够合理，代码的可测试性不好。

### 2. 编写可测试性代码的最有效手段

依赖注入是编写可测试性代码的最有效手段。通过依赖注入，我们在编写单元测试的时候，可以通过 mock 的方法解依赖外部服务，这也是我们在编写单元测试的过程中最有技术挑战的地方。

### 3. 常见的 Anti-Patterns

常见的测试不友好的代码有下面这 5 种：

代码中包含未决行为逻辑

滥用可变全局变量

滥用静态方法


使用复杂的继承关系

高度耦合的代码

## 课堂讨论

1. 实战案例中的 `void fillTransactionId(String preAssignedId)` 函数中包含一处静态函数调用：`IdGenerator.generateTransactionId()`，这是否会影响到代码的可测试性？在写单元测试的时候，我们是否需要 mock 这个函数？
2. 我们今天讲到，依赖注入是提高代码可测试性的最有效的手段。所以，依赖注入，就是不要在类内部通过 `new` 的方式创建对象，而是要通过外部创建好之后传递给类使用。那是不是所有的对象都不能在类内部创建呢？哪种类型的对象可以在类内部创建并且不影响代码的可测试性？你能举几个例子吗？

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

点击参加小程序学习打卡 

# 8个月，攻克设计模式



扫一扫参与小程序打卡



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。



上一篇 28 | 理论二：为了保证重构不出错，有哪些非常能落地的技术手段？

下一篇 30 | 理论四：如何通过封装、抽象、模块化、中间层等解耦代码？

## 精选留言 (36)

写留言



安静的boy

2020-01-08

这节满满的干货👍👍👍

展开



17



失火的夏天

2020-01-08

思考题1，该方法逻辑就是填充一个ID，基本都是内部实现的一个id生成器，可以不用重写。一定要重写也行，自己弄一个自增id实现就行了。

思考题2，提供方法的类不要new，也就是我们常说的service类，这个是要依赖注入的。提供属性的类，比如vo，bo，entity这些就可以new。

展开



12



辣么大

2020-01-08

参考争哥今天的代码写了例子中的测试（可运行）：

<https://github.com/gdhucoder/Algorithms4/tree/master/designpattern/u29>

今天学习到了高级的单元测试方法：

1、依赖外部单例：将单例封装...

展开

2

10



QQ怪

2020-01-08

看到一半，我就来评论，老师收下我的膝盖，太强了

展开

作者回复: 🙏 感谢认可!



4

**桂城老托尼**

2020-01-08

感谢争哥分享

课后讨论1.id的生成逻辑有点没看懂，单纯从代码覆盖上看，fillTransactionId 未覆盖完全，需要mock下这个静态方法，当然也有其他分支逻辑可以覆盖。

id没有在execute方法中不是核心属性(mock方法的入参)，不影响execute的可测试性。id的生成用静态方法真的好么？ ...

展开 ▾



3

**达文西**

2020-01-09

内容都是干货,不够看啊

展开 ▾



2

**下雨天**

2020-01-08

问题回答：

1. IdGenerator.generateTransactionId()有未决行为逻辑，但不是说有未决行为就一定影响可测试性，前提是需要看未决行为是否有测试必要性，此处生成一个随机数(类似 System.currentTimeMillis())，测试意义不大！

...

展开 ▾



2

**逍遥思**

2020-01-08

1. 不会影响可测试性，因为 generateTransactionId 并不需要什么外部服务，所以也不需要 mock

2. 不是。不依赖外部服务的类就可以内部创建，比如 String



2

**平风造雨**

2020-01-08

// 抽取了当前时间获取的逻辑，方便测试

```
private long currentTimeMillis;
```

```
private Date dueTime;
public Demo(Date dueTime){
    this.dueTime = dueTime;...
```

展开 ▾



1



**Jesse**

2020-01-08

思考题1, 该方法产生一个唯一的ID,我认为不需要mock。

思考题2, 我觉得如果对象有行为, 并且行为与外部系统交互或者执行的结果具有不确定性, 就需要依赖注入来完成测试。如果对象的行为是可预测的并且唯一的, 可以直接new。

展开 ▾



1



**蒋先生**

2020-01-13

收获很多, 要是文章结尾有个完整代码就更好了, 不然现在还得一点点去复制才能看到全貌。



**jaryoung**

2020-01-13

思考1, 无需实现, 随机生成的ID不应该影响业务主流程, 除非ID有业务含义就另当别论。

思考二, 如果类本身是当前类的内部类?

展开 ▾



**饭粒**

2020-01-11

读完这篇感觉一下就认识了代码的可测试性。依赖注入提高代码的可测试性, 那 spring 里推荐使用 setter 方法形式的 @autowired 注入 bean 更好哎。

然后我认为 IdGenerator.generateTransactionId() 不需要 mock, 它的功能应该就是生成一个全局唯一的 id, 对 Transaction 而言功能简单, 不影响测试性, 本身的实现逻辑不在 Transaction 测试。

展开 ▾



**小美**

2020-01-10

争哥，想请教一个问题。我们在mock RedisDistributedLock的时候，将其封装成TransactionLock，这样原Transaction类中调用RedisDistributedLock的地方都改成了调用TransactionLock。

这样不就破坏了原有类的实现吗？如果这样的mock很多，最后忘记修改，会不会有问题？

展开 ∨



小美

2020-01-10

争哥，想请教一个问题。我们在mock RedisDistributedLock的时候，将其封装成TransactionLock，这样原Transaction类中调用RedisDistributedLock的地方都改成了调用TransactionLock。

这样不就破坏了原有类的实现吗？如果这样的mock很多，最后忘记修改，会不会有问题？

展开 ∨



L 🐱

2020-01-10

关于思考题: 思考题一 IdGenerator.generateTransactionId() 只是生成一个 id 生成器, 不需要mock 这个函数, 就生成一个随机数, 没多大测试意义.

思考题二 有实现方法的类应该用依赖注入减少依赖, entity bo vo 等bean 可以 new 不需要依赖注入

展开 ∨



堵车

2020-01-10

请问单元测试在什么时候做比较好？是一个模块写完再测试还是一个方法写完再测试？文中获取redis锁用静态方法，一行代码就搞定了，而且几乎所有的分布式锁都是这样的写法，用起来很方便，后期几乎不会变动。为了可测试性，每个业务都封装了一个lock,是不是过度解耦了。

展开 ∨



花儿少年

2020-01-10

mock的通常是外部服务，id生成器如果是外部服务那就需要mock，不过通常来说id生成不会影响业务逻辑。

什么时间可以new，需要取决于new的对象是否有外部依赖，有则需要注入，否则可以new

展开 ▾



石仔

2020-01-10

继续优化Demo:

```
public class Demo {  
    private long currentTimestamp;  
    private long dueTimestamp;  
    private long delayDays;...
```

展开 ▾



石仔

2020-01-10

[原则]是:不依赖网络,IO或者第三方服务的就可以不需要mock.

- 1.本实战案例 并没有要对TransactionId进行验证,所以只要它能正常生成就完成了它的任务不需要mock.
- 2.类内部通过 new 的方式创建对象,只要这个对象的创建不违反[原则]而且不需要在后续测试依赖到就可以放在内部new来创建....

展开 ▾

