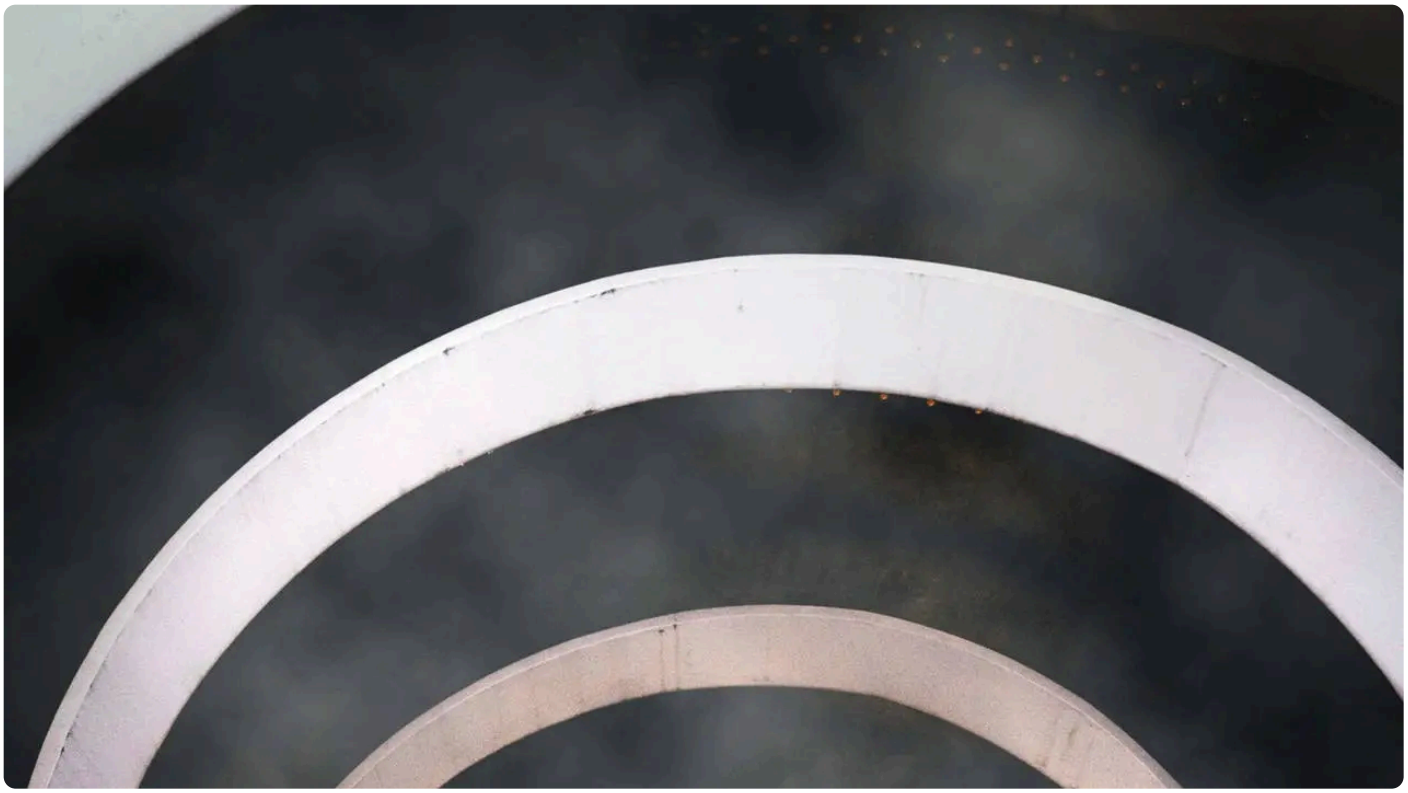
29 | 理论三：什么是代码的可测试性？如何写出可测试性好的代码？

王争 · 设计模式之美



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

```
public class Transaction {
  private String id;
  private Long buyerId;
  private Long sellerId;
  private Long productId;
  private String orderId;
  private Long createTimestamp;
  private Double amount;
  private STATUS status;
  private String walletTransactionId;

  // ...get() methods...

  public Transaction(String preAssignedId, Long buyerId, Long sellerId, Long prod
    if (preAssignedId != null && !preAssignedId.isEmpty()) {
      this.id = preAssignedId;
    } else {
      this.id = IdGenerator.generateTransactionId();
    }
    if (!this.id.startWith("t_")) {
      this.id = "t_" + preAssignedId;
    }
    this.buyerId = buyerId;
    this.sellerId = sellerId;
    this.productId = productId;
    this.orderId = orderId;
    this.status = STATUS.TO_BE_EXECUTD;
    this.createTimestamp = System.currentTimestamp();
  }

  public boolean execute() throws InvalidTransactionException {
    if ((buyerId == null || (sellerId == null || amount < 0.0) {
      throw new InvalidTransactionException(...);
    }
```

```
    if (status == STATUS.EXECUTED) return true;
    boolean isLocked = false;
    try {
      isLocked = RedisDistributedLock.getSingletonIntance().lockTransction(id);
      if (!isLocked) {
        return false; // 锁定未成功，返回false，job兜底执行
      }
      if (status == STATUS.EXECUTED) return true; // double check
      long executionInvokedTimestamp = System.currentTimestamp();
      if (executionInvokedTimestamp - createdTimestap > 14days) {
        this.status = STATUS.EXPIRED;
        return false;
      }
      WalletRpcService walletRpcService = new WalletRpcService();
      String walletTransactionId = walletRpcService.moveMoney(id, buyerId, seller
      if (walletTransactionId != null) {
        this.walletTransactionId = walletTransactionId;
        this.status = STATUS.EXECUTED;
        return true;
      } else {
        this.status = STATUS.FAILED;
        return false;
      }
    } finally {
      if (isLocked) {
       RedisDistributedLock.getSingletonIntance().unlockTransction(id);
      }
    }
  }
}
```

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

```
public void testExecute() {
  Long buyerId = 123L;
  Long sellerId = 234L;
  Long productId = 345L;
  Long orderId = 456L;
  Transction transaction = new Transaction(null, buyerId, sellerId, productId, or
  boolean executedResult = transaction.execute();
  assertTrue(executedResult);
}
```

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。

> 如果要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，那我们就需要将被测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数据。

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数据，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

```
public class MockWalletRpcServiceOne extends WalletRpcService {
  public String moveMoney(Long id, Long fromUserId, Long toUserId, Double amount)
    return "123bac";
  }
}

public class MockWalletRpcServiceTwo extends WalletRpcService {
  public String moveMoney(Long id, Long fromUserId, Long toUserId, Double amount)
    return null;
  }
}
```

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

```
public class Transaction {
  //...
  // 添加一个成员变量及其set方法
  private WalletRpcService walletRpcService;

  public void setWalletRpcService(WalletRpcService walletRpcService) {
    this.walletRpcService = walletRpcService;
  }
  // ...
  public boolean execute() {
    // ...
    // 删除下面这一行代码
    // WalletRpcService walletRpcService = new WalletRpcService();
    // ...
  }
}
```

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

```
public void testExecute() {
  Long buyerId = 123L;
  Long sellerId = 234L;
  Long productId = 345L;
  Long orderId = 456L;
  Transction transaction = new Transaction(null, buyerId, sellerId, productId, or
  // 使用mock对象来替代真正的RPC服务
  transaction.setWalletRpcService(new MockWalletRpcServiceOne()):
```

```
  boolean executedResult = transaction.execute();
  assertTrue(executedResult);
  assertEquals(STATUS.EXECUTED, transaction.getStatus());
}
```

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

```
public class TransactionLock {
  public boolean lock(String id) {
    return RedisDistributedLock.getSingletonIntance().lockTransction(id);
  }

  public void unlock() {
    RedisDistributedLock.getSingletonIntance().unlockTransction(id);
  }
}

public class Transaction {
  //...
  private TransactionLock lock;

  public void setTransactionLock(TransactionLock lock) {
    this.lock = lock;
  }

  public boolean execute() {
    //...
    try {
      isLocked = lock.lock();
```

```
      //...
    } finally {
      if (isLocked) {
        lock.unlock();
      }
    }
    //...
  }
`
```

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

```
public void testExecute() {
  Long buyerId = 123L;
  Long sellerId = 234L;
  Long productId = 345L;
  Long orderId = 456L;

  TransactionLock mockLock = new TransactionLock() {
    public boolean lock(String id) {
      return true;
    }

    public void unlock() {}
  };

  Transction transaction = new Transaction(null, buyerId, sellerId, productId, or
  transaction.setWalletRpcService(new MockWalletRpcServiceOne());
  transaction.setTransactionLock(mockLock);
  boolean executedResult = transaction.execute();
  assertTrue(executedResult);
  assertEquals(STATUS.EXECUTED, transaction.getStatus());
}
```

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。

```
public void testExecute_with_TransactionIsExpired() {
  Long buyerId = 123L;
  Long sellerId = 234L;
  Long productId = 345L;
  Long orderId = 456L;
  Transction transaction = new Transaction(null, buyerId, sellerId, productId, or
  transaction.setCreatedTimestamp(System.currentTimestamp() - 14days);
  boolean actualResult = transaction.execute();
  assertFalse(actualResult);
  assertEquals(STATUS.EXPIRED, transaction.getStatus());
}
```

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：

```
public class Transaction {

  protected boolean isExpired() {
    long executionInvokedTimestamp = System.currentTimestamp();
    return executionInvokedTimestamp - createdTimestamp > 14days;
  }

  public boolean execute() throws InvalidTransactionException {
    //...
      if (isExpired()) {
        this.status = STATUS.EXPIRED;
        return false;
      }
    //...
  }
}
```

复制代码

针对重构之后的代码，测试用例 3 的代码实现如下所示：

```
public void testExecute_with_TransactionIsExpired() {
  Long buyerId = 123L;
  Long sellerId = 234L;
  Long productId = 345L;
  Long orderId = 456L;
  Transction transaction = new Transaction(null, buyerId, sellerId, productId, or
    protected boolean isExpired() {
      return true;
    }
  };
  boolean actualResult = transaction.execute();
  assertFalse(actualResult);
  assertEquals(STATUS.EXPIRED, transaction.getStatus());
}
```

复制代码

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。

```
```

复制代码

```
public Transaction(String preAssignedId, Long buyerId, Long sellerId, Long prod
  if (preAssignedId != null && !preAssignedId.isEmpty()) {
    this.id = preAssignedId;
  } else {
    this.id = IdGenerator.generateTransactionId();
  }
  if (!this.id.startWith("t_")) {
    this.id = "t_" + preAssignedId;
  }
  this.buyerId = buyerId;
  this.sellerId = sellerId;
  this.productId = productId;
  this.orderId = orderId;
  this.status = STATUS.TO_BE_EXECUTD;
  this.createTimestamp = System.currentTimestamp();
}
```

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：

复制代码

```
public Transaction(String preAssignedId, Long buyerId, Long sellerId, Long prod
  //...
  fillTransactionId(preAssignId);
  //...
}

protected void fillTransactionId(String preAssignedId) {
  if (preAssignedId != null && !preAssignedId.isEmpty()) {
    this.id = preAssignedId;
  } else {
    this.id = IdGenerator.generateTransactionId();
  }
  if (!this.id.startWith("t_")) {
    this.id = "t_" + preAssignedId;
  }
}
```

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，这也有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，试着重构一下下面的代码，并且为它编写单元测试。

复制代码

```
public class Demo {
  public long caculateDelayDays(Date dueTime) {
    long currentTimestamp = System.currentTimeMillis();
    if (dueTime.getTime() >= currentTimestamp) {
      return 0;
    }
    long delayTime = currentTimestamp - dueTime.getTime();
    long delayDays = delayTime / 86400;
    return delayDays;
  }
}
```

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

```
public class RangeLimiter {
  private static AtomicInteger position = new AtomicInteger(0);
  public static final int MAX_LIMIT = 5;
  public static final int MIN_LIMIT = -5;

  public boolean move(int delta) {
    int currentPos = position.addAndGet(delta);
    boolean betweenRange = (currentPos <= MAX_LIMIT) && (currentPos >= MIN_LIMIT)
    return betweenRange;
  }
}

public class RangeLimiterTest {
  public void testMove_betweenRange() {
    RangeLimiter rangeLimiter = new RangeLimiter();
    assertTrue(rangeLimiter.move(1));
    assertTrue(rangeLimiter.move(3));
    assertTrue(rangeLimiter.move(-5));
  }

  public void testMove_exceedRange() {
    RangeLimiter rangeLimiter = new RangeLimiter();
    assertFalse(rangeLimiter.move(6));
  }
}
```

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。

AI智能总结

本文深入探讨了提高代码可测试性的关键方法，重点讨论了在编写可测试代码时可能遇到的挑战以及解决方法。通过实际案例展示了一个复杂的交易类代码，并提出了针对该类的六个测试用例。文章指出了在执行这些测试用例时可能遇到的问题，包括对外部服务的依赖、网络通信的耗时和不可控因素等。作者通过详细的技术讲解，展示了如何使用依赖注入和mock技术来提高代码的可测试性，使得单元测试不再依赖外部系统，从而更加可靠和高效。此外，文章还强调了良好的代码设计原则和思想对于代码可测试性的重要性，以及在平时的开发中要多思考代码编写是否容易进行单元测试，有利于设计出高质量的代码。总之，本文通过清晰的案例和详细的技术讲解，帮助读者了解了如何应对代码可测试性的挑战，对于软件开发人员具有一定的参考价值。

全部留言 (116)

最新　精选

楊_宵夜
2020-01-20

争歌, 代码中isExpired()方法的修饰符是protected, 如果某些方法从设计原则来说应该设置为private的话, 那么这样的手动mock的方式是否就不适用了呢?
换个角度来提问: 为了维持可测试性, 在代码中加入过多protected的方法, 是否合理呢?

> 作者回复: 也是没办法的事情，理论上应该是private的。所以会有@VisibleForTesting这样的annotation

共 9 条评论>　77

张三丰
2020-05-28

老师，下面这句话不是很理解，如果我的某个接口就是需要依赖很多服务才能把结果正确返回给前端，这时候怎么办？比如查询购物车，需要访问商品服务的商品信息，优惠服务的优惠信息，同时访问价格服务的价格信息等等，这个时候，高度耦合怎么去避免呢？

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。

作者回复: 这个不叫高度耦合吧。不是说耦合很多就是高度耦合，也要看业务需求啊，确实要这么多数据，那必然要以来这么多服务。还有，为了前端获取数据简单，可以用facade模式，包裹一层接口。


21

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

作者回复: 😄 感谢认可！

共 2 条评论>


17

Vincent.X
2020-06-17

手机看代码有老是要拖动，有什么解决的办法吗？？

作者回复: 只能@一下编辑了


5

美美
2020-01-08

有多个通过spring注入的类时，应该怎么做测试呢？

作者回复: 可以借助springtest测试框架来做

共 4 条评论> 4

Mew151
2020-08-26

有一个问题，如果测试方法A()中调用了本类的私有方法B()，这个时候该怎么处理呢?

作者回复: 你指的处理什么呢?

1

J.Smile
2020-01-08

想到一个问题，代码结构扁平化的极端结果可能会造成依赖对象过多吗？这种情况mock不是依然难搞吗

作者回复: ”代码结构扁平化的极端结果“能举个例子吗?

共 2 条评论> 1

qpzm7903
2020-04-26

请问贫血模式的mvc中的service怎么进行单元测试呢

作者回复: 单元测试跟贫不贫血没关系吧

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。

共 2 条评论> 178

安静的boy

2020-01-08

这节满满的干货👍👍👍

61