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？

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。

```
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 productId, String orderId) {
    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, sellerId, amount);
      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：

```
public void testExecute() {
  Long buyerId = 123L;
  Long sellerId = 234L;
  Long productId = 345L;
  Long orderId = 456L;
  Transction transaction = new Transaction(null, buyerId, sellerId, productId, orderId);
  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。

```
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：

```
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：

```
public void testExecute() {
  Long buyerId = 123L;
  Long sellerId = 234L;
  Long productId = 345L;
  Long orderId = 456L;
  Transction transaction = new Transaction(null, buyerId, sellerId, productId, orderId);
  // 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：

```
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。

```
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, orderId);
  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, orderId);
  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;
      }
    //...
  }
}
```

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, orderId) {
    protected boolean isExpired() {
      return true;
    }
  };
  boolean actualResult = transaction.execute();
  assertFalse(actualResult);
  assertEquals(STATUS.EXPIRED, transaction.getStatus());
}
```

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。

```
  public Transaction(String preAssignedId, Long buyerId, Long sellerId, Long productId, String orderId) {
    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：

```
public Transaction(String preAssignedId, Long buyerId, Long sellerId, Long productId, String orderId) {
  //...
  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。

```
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。

```
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。

精选留言：

- 安静的boy 2020-01-08 09:02:57
  这节满满的干货👍👍👍 [9赞]

- 失火的夏天 2020-01-08 08:29:39
  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。 [9赞]

- 辣么大 2020-01-08 09:22:17
  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
  2、未决行为：例时间、随机数。将未决行为重新封装，测试时mock，使用匿名类。

  关于讨论1：需要mock的情况id会写入数据库的话，测试后需要恢复现场。曾经遇到过这么一个情况，id是通过一张表维护的，大于0，在代码中id的数据类型是Integer（遗留代码），由于测试时没有恢复现场，导致测试数据库中id增加过快，超过了代码中Integer的表示范围，而产生了意想不到的问题。 [8赞]

- 下雨天 2020-01-08 17:59:54
  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！

  2.贫血模型实体类 [2赞]

- QQ怪 2020-01-08 16:34:08
  看到一半，我就来评论，老师收下我的膝盖，太强了 [2赞]

作者回复2020-01-08 21:06:29
😁感谢认可！

- 逍遥思 2020-01-08 15:04:45
1. 不会影响可测试性，因为 generateTransactionId 并不需要依赖什么外部服务，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赞]

- 桂城老托尼 2020-01-08 08:04:53
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?
2.有行为的对象不适合在类中new，尽量使用依赖注入，依赖接口编程，而不是具体的实现。 数据对象适合在类中new 比如各种model do vo info 。
一家之言欢迎讨论指正。 [2赞]

- 平风造雨 2020-01-08 12:45:02
// 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;
this.currentTimeMillis = getCurrentTimeMillis();
}

protected long getCurrentTimeMillis(){
return System.currentTimeMillis();
}
public long caculateDelayDays() {
if(dueTime.getTime() >= currentTimeMillis){
return 0;
}
long delayTime = currentTimeMillis - dueTime.getTime();
long delayDays = delayTime / 86400_000;
return delayDays;
}
@Test
public void testCaculateDelayDays(){
TimeZone timeZone = TimeZone.getTimeZone("Asia/ShangHai");
Calendar calendar = Calendar.getInstance(timeZone);
calendar.clear();
calendar.set(2020, Calendar.FEBRUARY,1,0,0,0);
Date dueTime = calendar.getTime();
Demo demo = new DemoClassOne(dueTime);
Assert.assertEquals(demo.caculateDelayDays(), 0);
calendar.clear();
calendar.set(2019, Calendar.DECEMBER, 31, 0,0,0);
dueTime = calendar.getTime();
demo = new DemoClassOne(dueTime);

```
Assert.assertEquals(demo.caculateDelayDays(), 1);
}

public static class DemoClassOne extends Demo {
public DemoClassOne(Date dueTime) {
super(dueTime);
}
@Override
protected long getCurrentTimeMillis() {
TimeZone timeZone = TimeZone.getTimeZone("Asia/ShangHai");
Calendar calendar = Calendar.getInstance(timeZone);
calendar.clear();
calendar.set(2020, Calendar.JANUARY,1,0,0,0);
return calendar.getTimeInMillis();
}
} [1赞]
```

- Jesse 2020-01-08 10:23:03
  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。 [1赞]

- Ken张云忠 2020-01-10 08:11:03
  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？
  如果generateTransactionId()依赖了外部服务,如redis/mysql/zookeeper等,就会影响代码的可测试性,在写单元测试时就需要mock这个函数;
  如果generateTransactionId()没有依赖外部服务就不会影响代码的可测试性,写单元测试时也不需要mock这个函数.

  2.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？
  不是.
  没有依赖外部服务的对象可以在类内部创建,并且不会影响代码的可测试性.
  例如new Date()没有依赖外部服务,只是调用了本机操作系统的时间函数.

- 达文西 2020-01-09 20:58:12
  内容都是干货,不够看啊

- whistleman 2020-01-09 14:09:13
  一直搞不懂单元测试怎么写，于是就不喜欢写，跟着这节课敲了代码，学到了好多。感觉知道怎么去写，感觉自己一下变强了，争哥太强了，666

- #HEAVEN 2020-01-08 22:58:08
  你好，没有找到作者邮箱，想问一个问题；
  作者在开发一个需求的时候是怎样的一个流程，设计做到那种程度？
  比如说一般我会做1. 需求分析，列出哪些需求case； 2. 列出这些case需要开发哪些功能点；3. 主要涉及

到哪些类，结构如何组织；4. 主要类的主要职责等；5. 开始code了；
在开发的过程中也会遇到一些问题，比如，有时候有些类的职责或者结构开始的设计不太合理，需要一些修改；这个时候我就在怀疑，是不是前期做的设计不够充分造成的。也看到一些书上会把类的属性、方法都设计出来，还有主要流程case的序列图；但是这样做耗时较多，很多时候项目日程不允许。
像问一下，作者在开发中设计阶段有哪些流程，做到什么程度？
留个邮箱方便交流就更好了

- 苟麒睿 2020-01-08 21:36:14
对于IdGenerator.generateTransactionId()，虽然是未决行为，个人认为只是生成一个id了话，并不会包含非常复杂的逻辑操作，应该就跟Math.abs()类似，不需要进行mock

- 再见孙悟空 2020-01-08 21:15:06
今天老师讲的为了更好的单元测试而进行的重构，原来工作中无形间已经用到了。在对接三方 api 时，有时候缺少必要的参数信息，我们只能模拟调通，这时候我们就写一个类继承原始类，重写原方法，返回自己需要的数据，不过还有很多做的不足，例如对于不确定数据的mock 没有抽成方法等，持续学习，老师棒！

- 斐波那契 2020-01-08 20:09:37
感觉那个createtimestamp那边 如果没有set方法应该可以用反射去修改这个属性

思考题1 可以不mock 因为执行idgenerator之前有逻辑判断的 只要传入进去的参数不满足条件就不会走 其次对于id开头添加t_这个逻辑跟id生成器没有关系 只要保证造出来到id没有t_开头就可以测试

思考题2 其实最近在写一个需求 我就用了内部类 也觉得并没有破坏测试性 我这个内部类主要是为了隐藏某个接口的实现 不想被调用者在使用外部类时滥用我的每一个接口实现方法 起到一个保护作用 对于测试性 完全可以通过不同的外部类参数来进行调整 其实对于内部类的可测试性来讲 只要外部类有足够的参数来控制内部类就可以 对于内部类调用第三方的情况 只要外部类有参数可以注入就可以用mock来修改内部类的实现

- Jxin 2020-01-08 12:57:38
1.栏主好像提过，要谈谈分层对于可测试性的影响，不知是不是我记错了，这篇没提到哈。

回答问题
1.交易id这东西，是全局唯一的。不该被mock，mock了不仅没用，反而可能会有其他问题（如果有引入唯一键检验相关机制的话，比如幂等啥的）。

2.值对象可以new，因为值对象不会有涉及改动自身属性的方法，也就是说它通常是不可变的，所以也没什么检验的意义。而实体领域模型不一定可以new，因为其方法会改变自身属性，而对这些属性变动，有时候我们需要校验。而贫血实体dto或do之类的，一般也可以new，因为它只承接属性，场景类似值对象，只需要关心方法返回的dto或vo的值即可，无需关心方法内部是new还是注入的（对于方法而言，除了类成员属性的注入，方法入参也算注入吧）。

- 李小四 2020-01-08 11:41:09
设计模式_29:

1. 我认为静态方法```IdGenerator.generateTransactionId()```不需要mock，因为它不会很耗时(如果实现比较正常)，也没有未决行为，除非对于id有特殊的要求，否则不需要mock。

2. 这道题我想不清楚，想看看王争老师和大家的看法。

- 此鱼不得水 2020-01-08 11:22:11
  1. 未决行为 中提到的单测，可以把不确定的变量‘当前时间’提取出来作为入参

- 守拙 2020-01-08 10:38:14
  课堂讨论:

  Q1.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？

  Answer1:

  理论上讲fill()方法由于内部静态方法的使用,及id生成的未决行为,影响可测试性.

  解决方法是为fill()方法添加一个形参,generateId,如下:

  void fillTransactionId(@Nullable String preAssignedId, @Nullable String generateId)

  但这样做会影响封装性.fill()方法内部逻辑简单,对可测试性的影响是微不足道的.除非测试问题直指fill()方法,否则个人倾向于不做修改.

  Q2.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？

  Answer2: 内部类或静态内部类, 局部类的对象可以在类内部通过new 的方式初始化.它们是外部类行为的一部分,仅为外部类自己使用,不影响测试性.

  对于未决行为方法的改造:

  before:

  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;

 }

}
```

after:

```
public class Demo {

public long caculateDelayDays(Date dueTime, Date currentTime){

 long currentTimestamp = currentTime.getTime();

 if (dueTime.getTime() >= currentTimestamp) { return 0; }

 long delayTime = currentTimestamp - dueTime.getTime();

 long delayDays = delayTime / 86400;

 return delayDays;

 }

}
```