从零开始造Spring之单元测试

刘欣

微信：onlyliuxin97

微信公众号：码农翻身(coderising)

单元测试的误区

- 编译通过不就行了吗?
- **测试是测试小组的事情**，程序员应该做些更有趣，更有创意的事情，比如学习一些新技术
- **我写完代码已经手工的测试过了,为什么还得写单元测试**?
- **我用调试器把所有的分支都走过了，肯定没问题，提交**!
- 管它什么单元测试， 时间已经不多了， 经理催的紧，先做个丑陋的修改，让它工作，有时间再改吧。
- **写测试太麻烦了，得准备一大堆数据，不值**
- 那些测试运行的时间是在太长了!

什么是单元测试?

- 单元测试是**开发人员**编写的一小段代码，用于检验被测代码的一个有明确功能的小模块是否正确
  - 通常是用来判断某个类和函数的行为
  - 白盒测试
  - **开发人员**是最大的收益者

例子

```
public class Calculator {
    public int evaluate(String expr){

        //对expr进行解析，执行运算，实现代码略
        int result = …
        return result;
    }

}
```

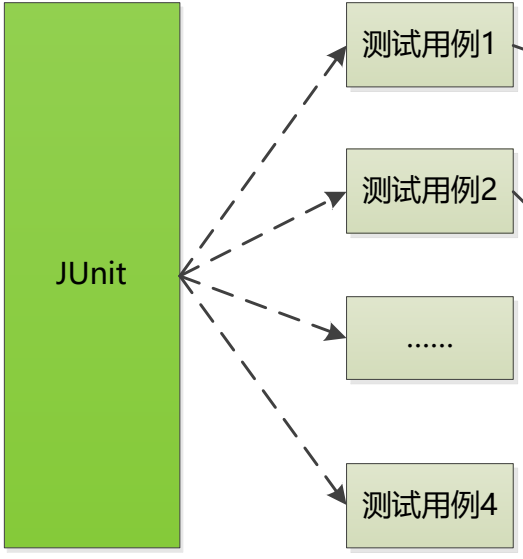
“10+20-5” -> 25

“(20-10)*5+6/3” -> 52

人肉测试

```
public static void main(String[] args){
    Calculator cal = new Calculator();
    int result = cal.evalute("10+20-5");
    System.out.println(result);

    result = cal.evalute("(20-10)*5+6/3");
    System.out.println(result);

}
```

单元测试框架

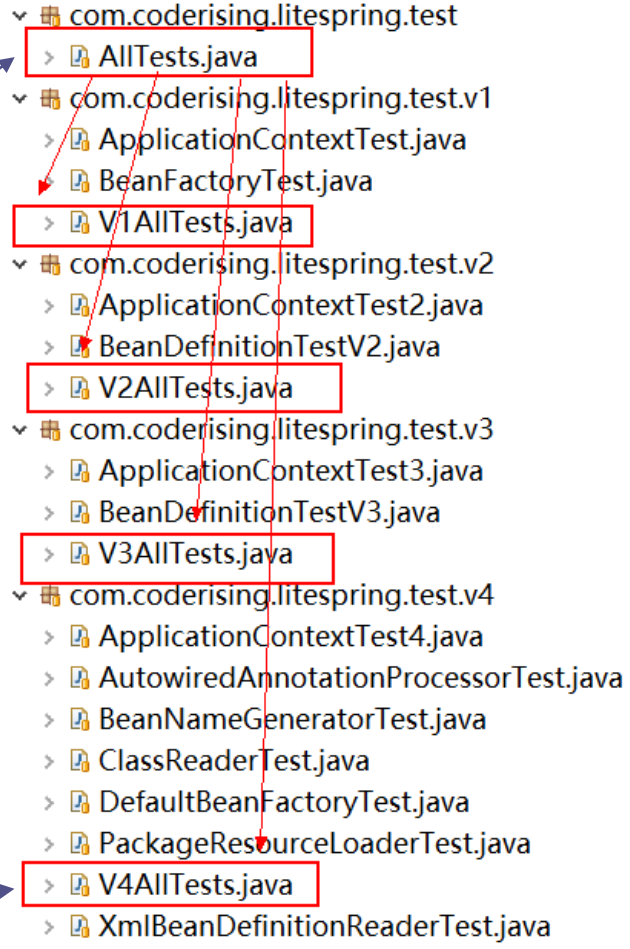



```java
public class CalculatorTest {

        @Test
        public void testEvaluate1() {
                Calculator cal = new Calculator();
                int result = cal.evalute("10+20-5");
                Assert.assertEquals(25,result);

        }
        @Test
        public void testEvaluate2(){
                Calculator cal = new Calculator();
                int result = cal.evalute("(20-10)*5+6/3");
                Assert.assertEquals(52,result);
        }
}
```

成百上千个测试用例该怎么组织?

```
@RunWith(Suite.class)
@SuiteClasses({
    V1AllTests.class,
    V2AllTests.class,
    V3AllTests.class,
    ……
    })
public class AllTests {

}
```

```
@RunWith(Suite.class)
@SuiteClasses({
    ClassReaderTest.class
    ApplicationContextText4.class
    ……
    })
public class V4AllTests {
}
```



JUnit常用的几种断言

Assert.**assertEquals**(expected, actual);

Assert.**assertTrue**(condition);

Assert.**assertNotNull**(object);

Assert.**assertArrayEquals**(expecteds, actuals);

如何对Exception进行测试

```
@Test
public void testEvaluateWrongExpression(){
    Calculator cal = new Calculator();
    try{
        int result = cal.evalute("10/0");
    }catch(ArithmeticException e){
        //代码应该进入这个分支
        return;
    }
    //如果走到这里，说明计算器实现得不正确
    Assert.fail();
}
@Test(expected=ArithmeticException.class)
public void testEvaluateWrongExpression(){
    Calculator cal = new Calculator();
    int result = cal.evalute("10/0");
}
```

两个特殊的方法

```
public class CalculatorTest {
    @Before
    public void setUp(){
        //每个测试用例执行前都会被调用一次
    }
    @After
    public void tearDown(){
        //每个测试用例执行后都会被调用一次
    }
    @Test
    public void testEvaluate1(){
        //测试用例的代码
    }
    @Test
    public void testEvaluate2(){
        //测试用例的代码
    }
}
```

setup()
testEvaluate1()
tearDown()

setUp()
testEvaluate2()
tearDown()

setUp()
……
tearDown()

两个更特殊的方法

```
public class CalculatorTest {
    @BeforeClass
    public static void beforeClass(){
        //对于CalculatorTest来说，只会在开
        始之前执行一次
    }
    @AfterClass
    public static void afterClass(){
        //对于CalculatorTest来说，只会在结
        束之后执行一次
    }
    ......
}
```

beforeClass()
setup()
testEvaluate1()
tearDown()

setUp()
testEvaluate2()
tearDown()

setUp()
……
tearDown()
afterClass()

单元测试的优点

- **验证行为**
  - 保证代码的正确性
  - 回归测试：即使到项目后期，我们仍然有勇气去增加新功能，修改程序结构，而不用担心破坏重要功能
  - 给重构带来保证
- **设计行为**
  - **测试驱动**迫使我们从调用者的角度去观察和思考问题，迫使我们把代码设计成可测试的，松耦合的。
- **文档行为**
  - 单元测试是一种无价的文档，精确的描述了代码的行为，是如何使用函数和类的最佳文档。

单元测试是个团队行为

- 互相帮助，互相扶持，共同前进
  - **你运行别人的测试用例：验证你的代码修改**
  - **别人运行你的测试用例：验证别人的代码修改**

No

Yes

单元测试的原则

- 测试代码和被测代码是同等重要的，需要被同时维护
  - 测试代码不是附属品！
  - 不但要重构代码，也要重构单元测试！
- 单元测试一定是隔离的
  - 一个测试用例的运行结果不能影响其他测试用例
  - 测试用不能互相依赖，应该能够以任何次序执行
- 单元测试一定是可以重复执行的
  - 不能依赖环境的变化
- 保持单元测试的简单性和可读性

单元测试的原则

- 尽量对接口进行测试
- 单元测试应该可以迅速执行
  - 给程序员提供及时的反馈
  - 使用Mock对象对数据库，网络的依赖进行解耦
- 自动化单元测试
  - 集成到build过程中去

使用Mock对象

- 真实的对象不易构造
  - 例如httpservlet 必须在servlet容器中才能创建出来
- 真实的对象非常复杂
  - 如jdbc中的Connection, ResultSet
- 真实的对象的行为具有不确定性，难于控制他们的输出或者返回结果
- 真实的对象的有些行为难于触发， 例如硬盘已满，网络连接断开
- 真实的对象可能还不存在，例如依赖的另外一个模块还没开发完毕

使用Mock对象

- 使用Mock 对象“替代”或者“冒充” 真实模块和被测试对象进行交互
  - 开发人员可以精确的定制期待的行为
- 对TDD提供有力的支持
  - 帮助你发现对象的角色和职责
  - 对接口编程，而不是对实现编程

Mock Object的例子

```
public class URLParser{
    public void parse(HttpServletRequest request){
        String startRow = request.getParameter(“startRow”);
        String endRow = request.getParameter(“endRow”);
        … do some business logic…

    }
}
```

- 方法一：开发人员写一个对接口HttpServletRequest的实现类，然后实现getParamter方法
  - 不得不实现几十个无用的空方法

使用Mock对象

```
//step 1: 创建mock 对象
MockControl control = MockControl.createControl(HttpServletRequest.class);
HttpServletRequest request = (HttpServletRequest) control.getMock();

//step2: 设置并记录mock对象的行为
request.getParameter("startRow");
control.setReturnValue("10");
request.getParameter("endRow");
control.setReturnValue("20");

// step3: 转换为回放模式
control.replay();

// step 4: 测试代码
URLParser parser = new URLParser(request);
parser.parse();
Assert xxx
```

对遗留代码进行测试

- 遗留代码不可避免
  - 虽然TDD是很有效的编程方法，但是我们的工作很少从第一行代码开始。
- 遗留代码不是坏代码
  - 它是可以工作的软件/组件，但是
  - 在设计和开发式没有考虑“可测试性”
- 遗留代码难于测试
  - 长久失修，导致业务逻辑难于理解
  - 依赖的资源太多，导致测试无从下手
  - 不敢修改，害怕牵一发而动全身

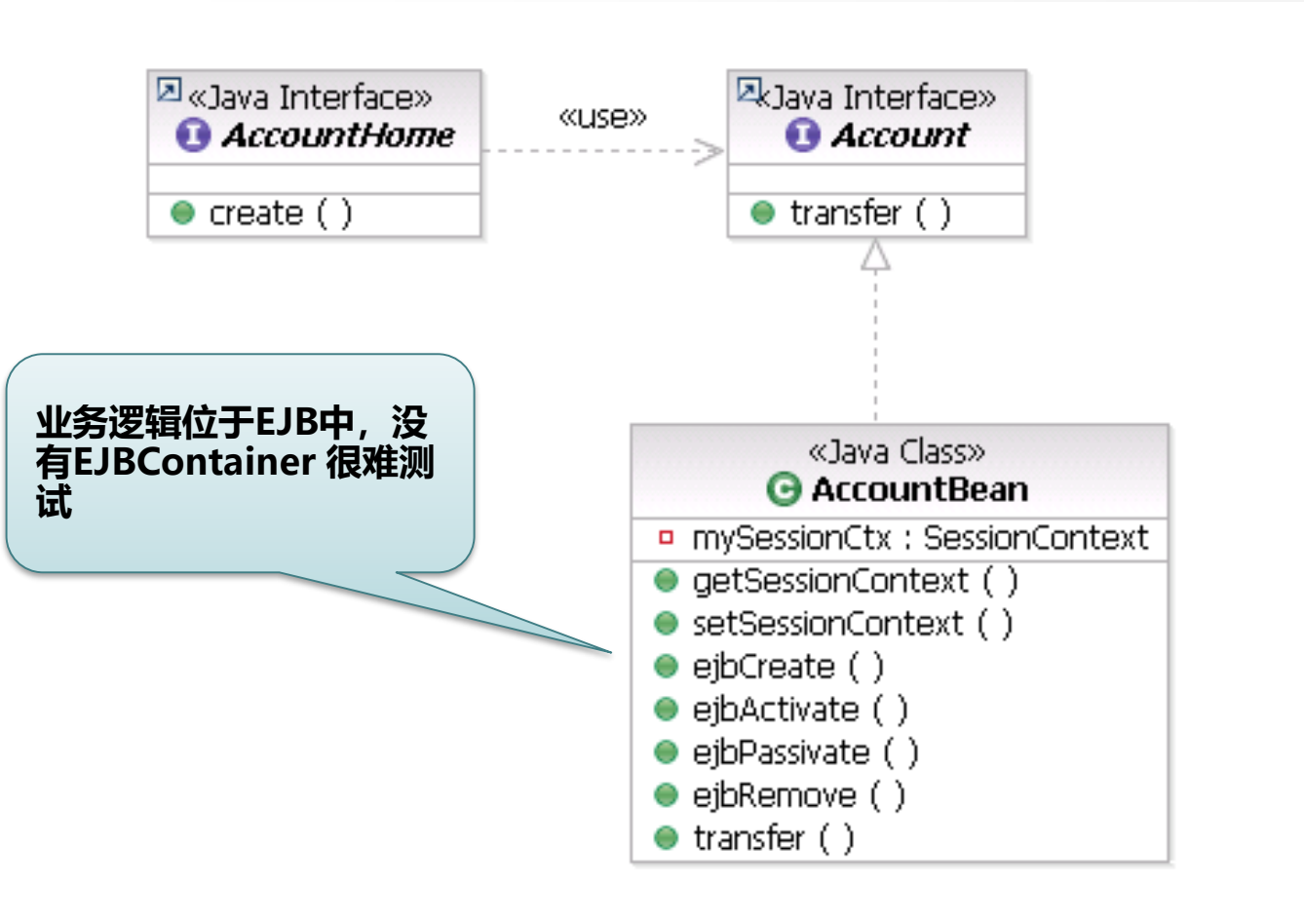
处理遗留代码的策略

- 重构代码，提高可测试性
- 使用Mock Object解除依赖
- 测试分解
  - 先写粗粒度的测试代码，然后编写细粒度的代码
  - Package -> Class -> method

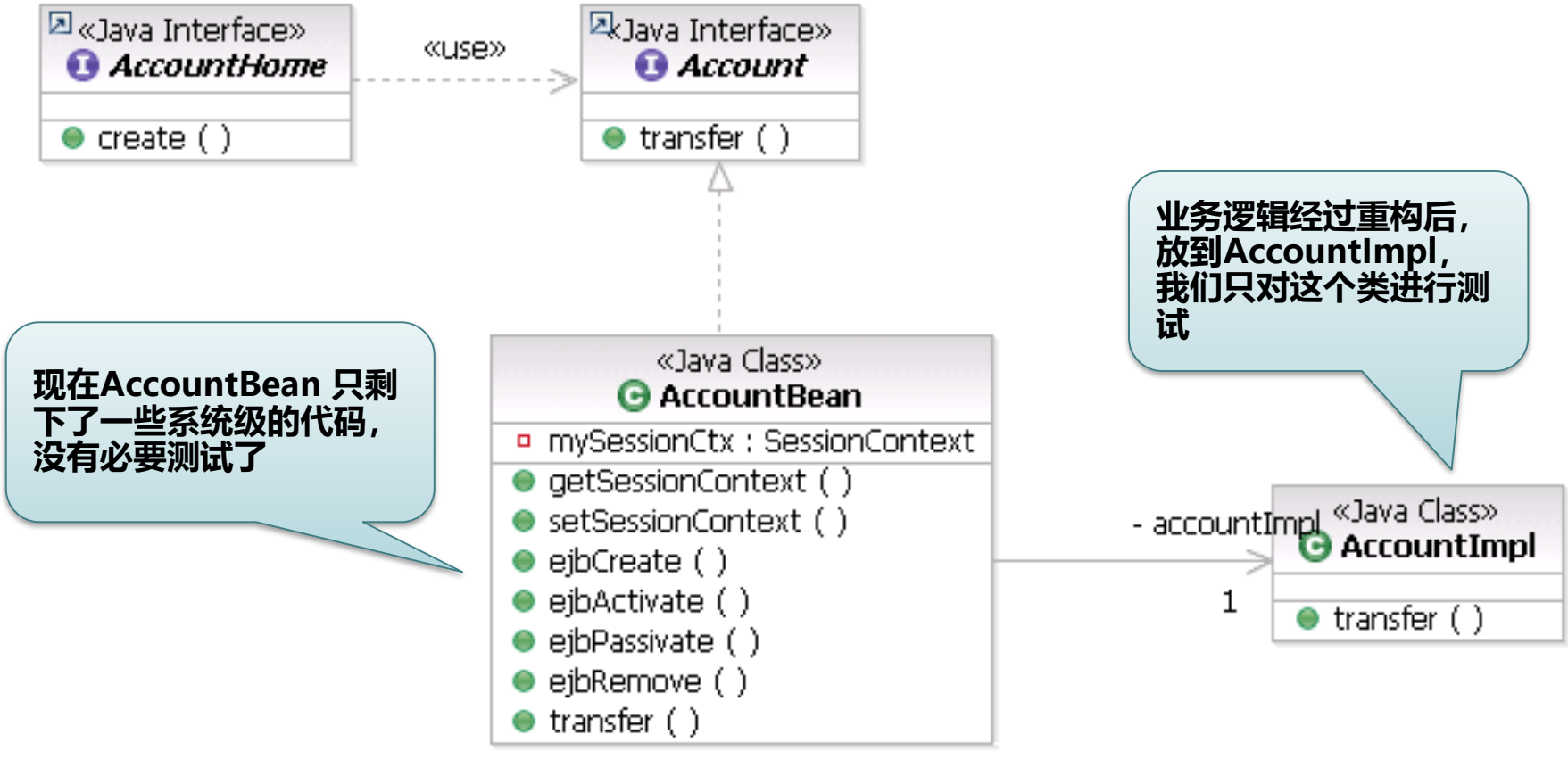
处理遗留代码的步骤

- 1.确定要测试的类和函数
- 2.解除依赖
- 3.编写测试用例
- 4.重构代码

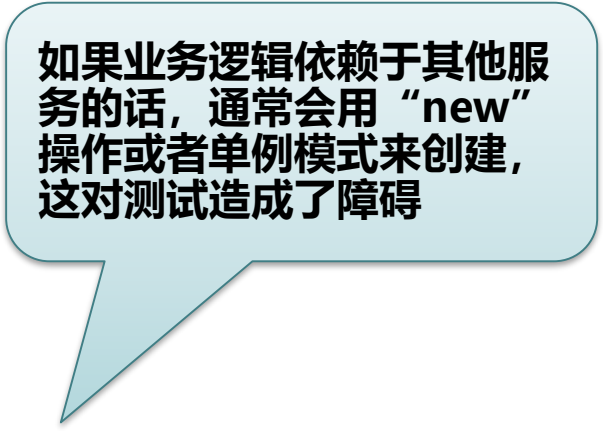
重构的例子


«Java Interface»
AccountHome
create ( )
«use»
«Java Interface»
Account
transfer ( )
业务逻辑位于EJB中，没有EJBContainer 很难测试
«Java Class»
AccountBean
mySessionCtx : SessionContext
getSessionContext ( )
setSessionContext ( )
ejbCreate ( )
ejbActivate ( )
ejbPassivate ( )
ejbRemove ( )
transfer ( )

进行重构


«Java Interface»
AccountHome
create ( )
«use»
«Java Interface»
Account
transfer ( )
«Java Class»
AccountBean
mySessionCtx : SessionContext
getSessionContext ( )
setSessionContext ( )
ejbCreate ( )
ejbActivate ( )
ejbPassivate ( )
ejbRemove ( )
transfer ( )
- accountImpl
1
«Java Class»
AccountImpl
transfer ( )
现在AccountBean 只剩下了一些系统级的代码，没有必要测试了
业务逻辑经过重构后，放到AccountImpl，我们只对这个类进行测试

重构的例子



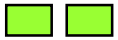

```
public class Account{
    public void transfer(){
        TransactionManager txManager = new TransactoinManagerImpl();
        txManager.begin();
        .... business logic ....
        txManager.commit();
    }
}
```

重构的例子

```
public class Account{
   public void transfer(){

      TransactionManager txManager = getTransactionManager();
      txManager.begin();
      .... business logic ....
      txManager.commit();
   }
   // create a new method to get the TransactionManager
   protected TransactionManager getTransactionManager(){
      return new TransactionManagerImpl();
   }
}

// 在测试之前，创建一个Acount子类,重写 getTransactionManger() 方法来提供一个Mock实现
Account accout = new Account(){
    protected TransactionManager getTransactionManager(){
            return new MockTransactionManager()
        }
}
// 测试业务逻辑， Mock实现会被调用
account.transfer();
```

单元测试的运行

- 1分钟
  - 经常运行一个小的单元测试套件集合来验证自己的代码



- 10 分钟
  - 运行其他人的单元测试套件，确保自己的代码不会影响其他人

- 每隔几小时
  - 运行整个系统的单元测试

好的单元测试

- 简单
  - 防止过度的Setup,否则不知道是测试用例的错误，还是业务逻辑的错误
- 隔离
- 可重复
  - 防止在一台机器上可以运行，在另外一台机器上失败
  - 防止今天成功，明天失败
- 运行快
  - 防止长时间的运行
- 代码覆盖面广
  - 防止测试通过，但是没测到什么代码

可以考虑在Review代码的同时对单元测试进行Review

使用code coverage工具

name	class, %	method, %	block, %	line, %
default package	98% (118/120)	66% (318/483)	81% (15517/19107)	77% (2651.4/3430)

```
539         // Resolve occurrences of "/../" in the normalized path
540         while (true)
541         {
542             int index = normalized.indexOf("/../");
543             if (index < 0)
544                 break;
545             if (index == 0)
546                 return (null);  // Trying to go outside our context
547             int index2 = normalized.lastIndexOf('/', index - 1);
548             normalized = normalized.substring(0, index2) +
549                 normalized.substring(index + 3);
550         }
551
552         // Return the normalized path that we have completed
553         return (normalized);
```