Spring测试先行



测试先行

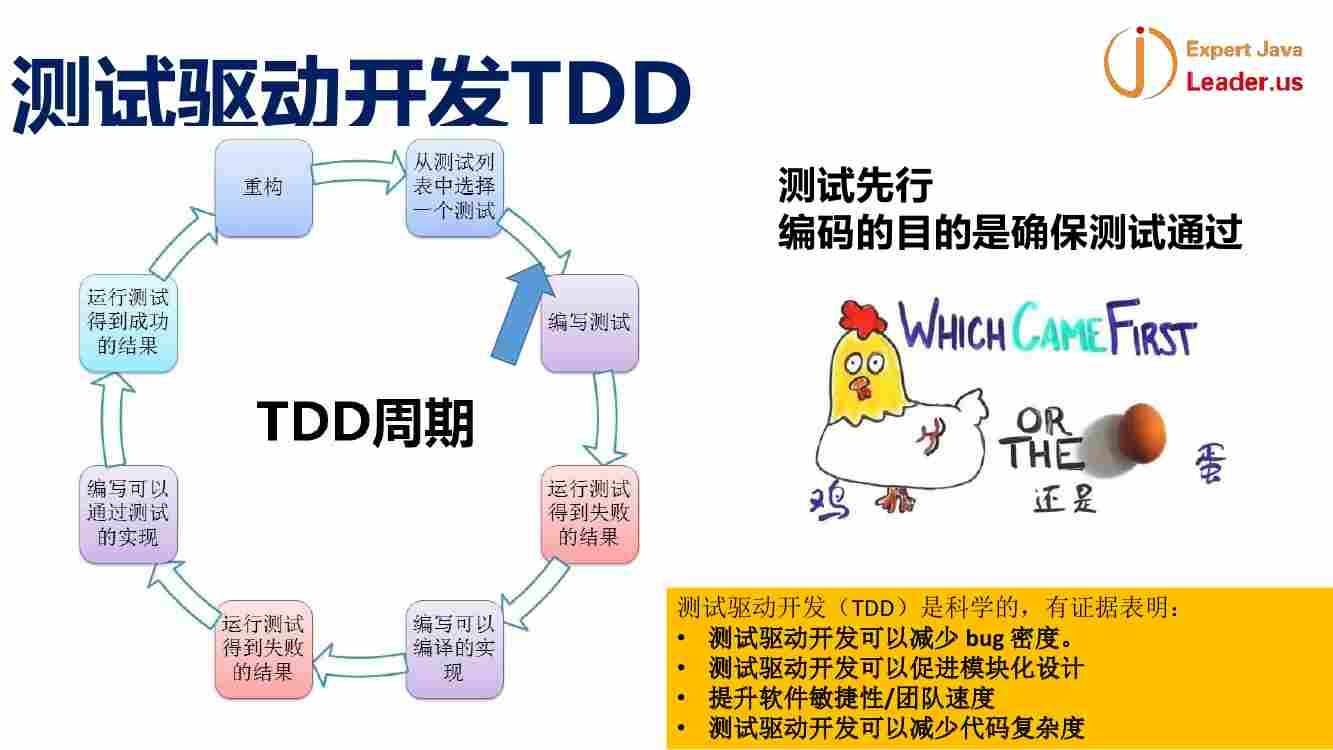
测试驱动开发 ----- spring在这做了**很多工作** 专门提供了一套复杂的API 来支持测试先行

----- 写单元测试 写产品代码

----- 重构单元测试 重构产品代码

=========== 这些尽量一次只做一件

下面这个周期是测试驱动开发 是闭环 开始看不到开始 最开始从编写测试开始



测试驱动开发 的周期是一个闭环 --- 闭环从哪里开始呢？

测试先行的说法 开始做的是单元测试 ---- 会发现 这些方法都无法通过 修改代码

然后代码修改完了 单元测试还是无法通过 ---- 继续改

也就是编译测试 首先失败

鼓励你先写**空的方法**

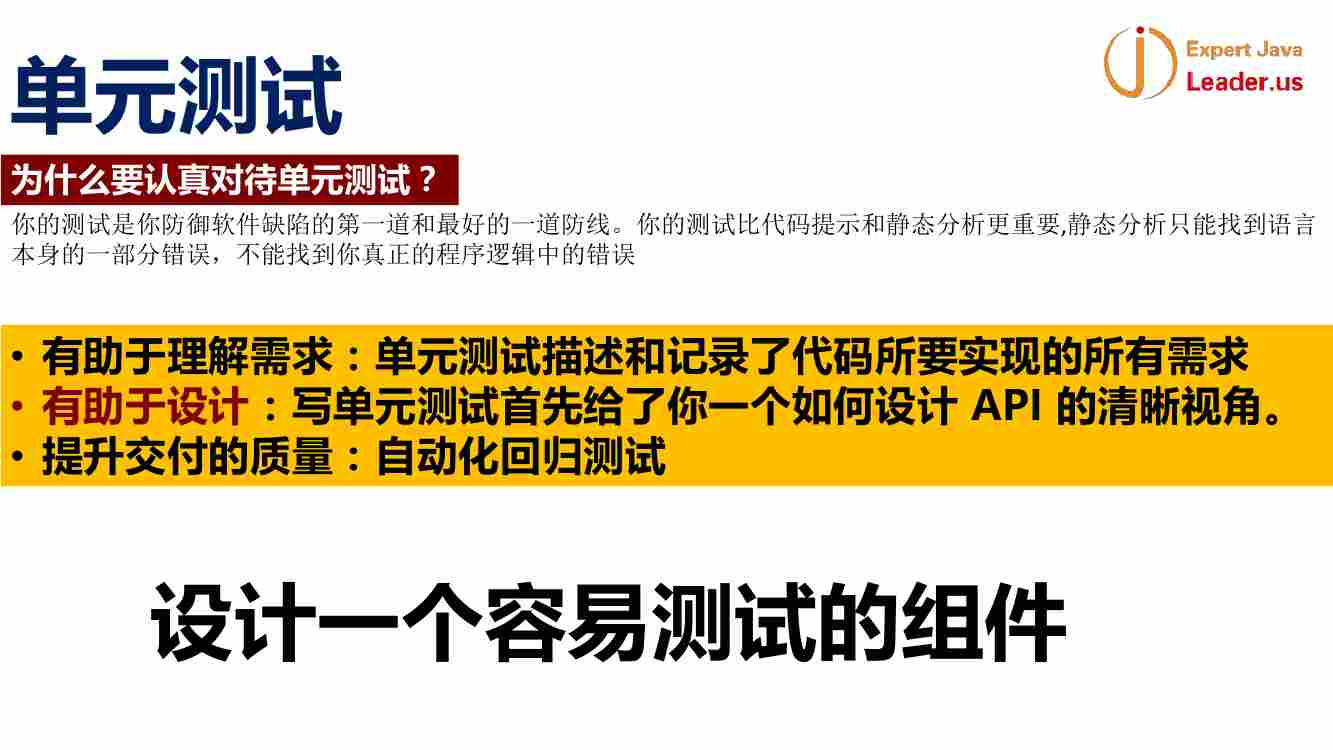
先是单元测试的代码都通过 表明你的工作是完成了

单元测试代码都通过了 表明完成了

黄色的框 ----- 证明是正确的（科学依据）

1. ---测试是比较难 也就是 测试从页面发起 --- 走一个流程 才能走到你这里 --- **前面的过程是人工的 并且时间长** ---- 如果这里的代码改动比较多 如果能写一个单元测试 是非常值得的 **绕过前面的手动过程** 让你的时间得到节省 ----- 这个就是敏捷度 提升速度
2. --- 减少bug ---- **单元测试就是减少bug提供的**
3. --- (1)中说了 提升敏捷度 就是快了
4. --- 减少代码的复杂度 ----发现你的**代码不好测试** 这样你的代码的模块设计有问题 ----- 这就是模块**依赖的太多** 这些**类不好去创建**

从测试角度 就是 依赖什么 ---- 接口 和 依赖是有问题的 ----- 促进模块设计 并 减小复杂度



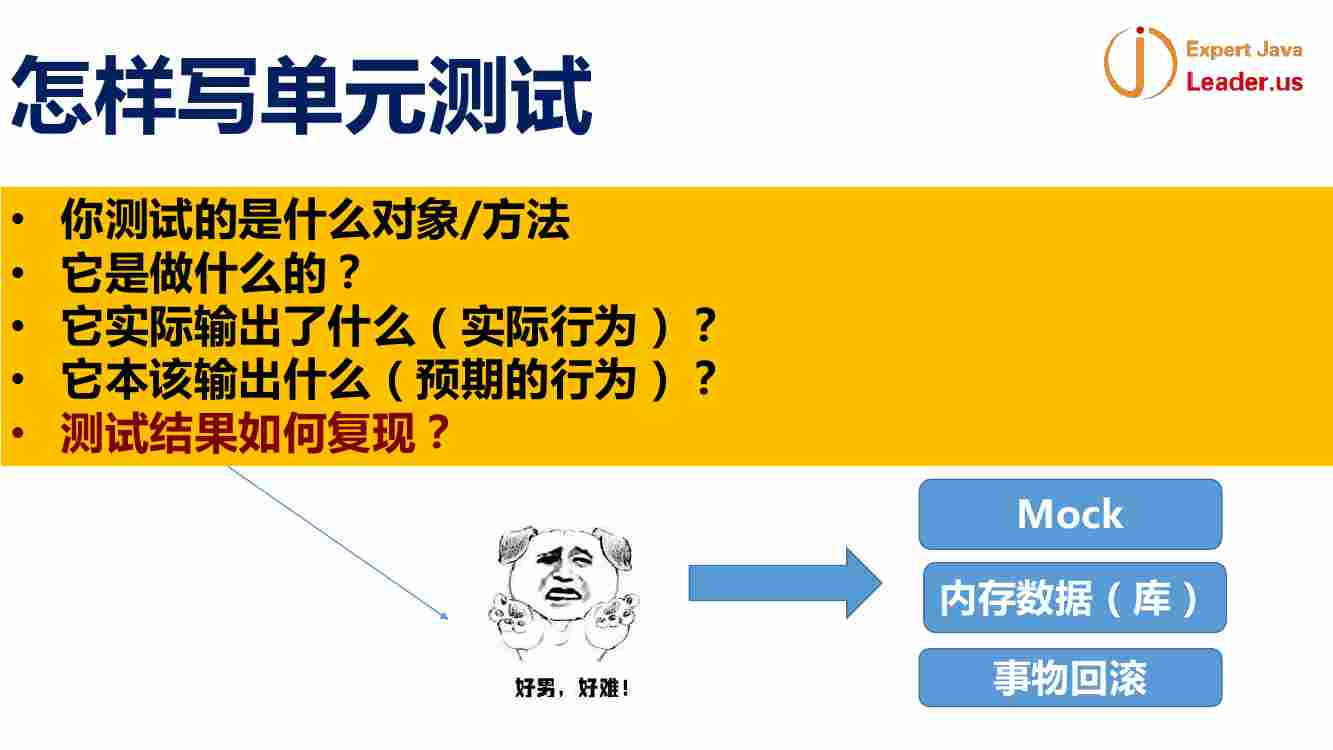
上面说了

**单元测试有助于理解需求** ----- 知道怎么测 怎么测的对的时候 -🡪 知道你的方法怎么实现 否则不知道你的方法怎么去写

写单元测试 也对需求更加了解 **特别是边界问题** ---- 你的组件 到底**要**做什么 不关心细节 **关心入口和出口 ----- 组件本身的边界**

**有助于设计** ----- 知道**依赖的组件** ---- 剩下就是PPT所说

最后就是进行 **自动化回归测试** --- 组件容易测试 设计就合理 否则 如果组件过于复杂 或者 依赖太多 就会难于测试



如何确保结果重现 这个非常难 ---- Spring做了很多工作

最后一点非常难 就是 如何复现 ----- Spring在这做了很多工作

我们的很多程序和持久存储有关 ---- 持久存储非常难办 本身有事务 **大家共用一个环境 --- 测试过程会改变数据 如何复现是一个很难办的事情**

----- 这里面有几种方式：

对**环境依赖达到最小** ----- 的确就是在查数据库 **但是 要认定 你们的记录不会被改变**（比如 Mock和内存数据库都能做到）

最后一个是事务回滚

------ - **最重要的是Mock ---- 假定你的条件都满足**

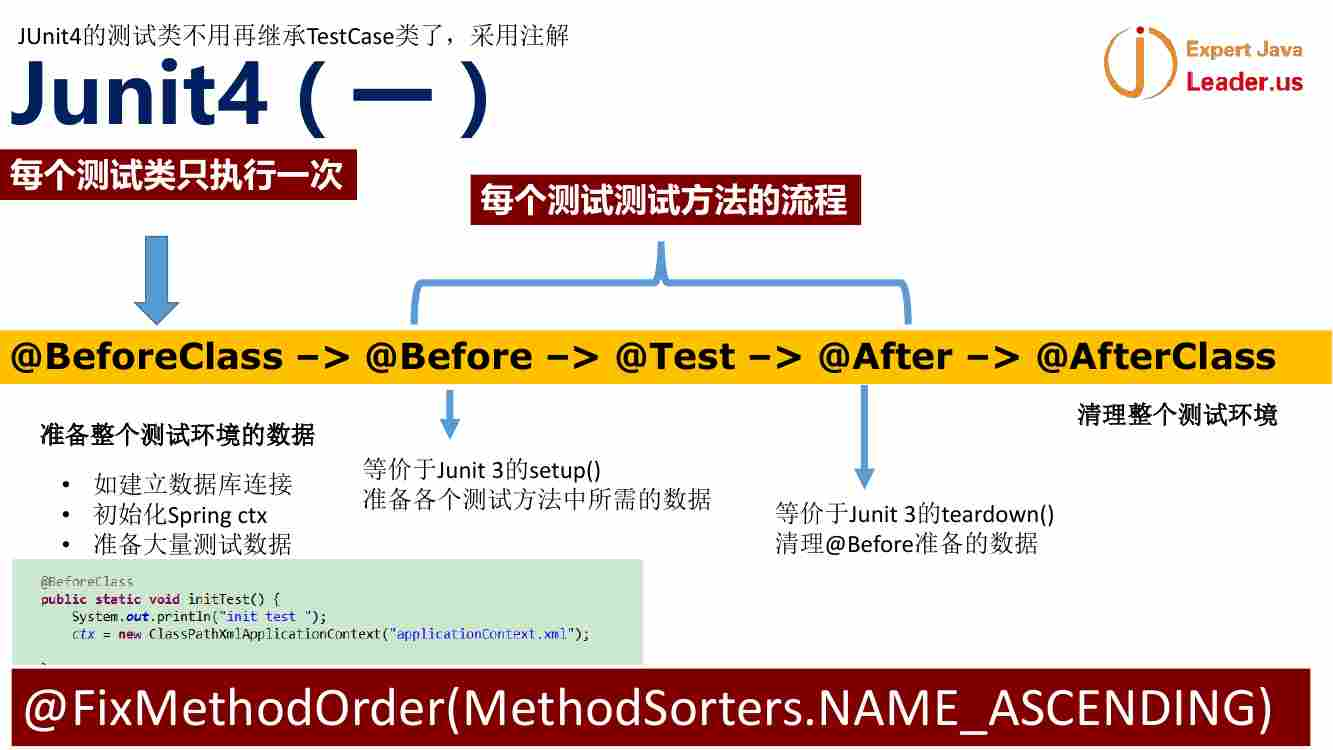
-------- 你的数据都在内存中 **不变化**

------- 回滚你的事务 这样 不就回到原来的数据中了么 ----- 数据就不变了

还有就是你真的使用数据库的时候 ----- 你要**保证你的系统有真的初始化的数据** ----- **这个大家都认可 是基准 是基准数据库** 基准数据库不能随便修改 单元测试就按照这个基准数据库进行

***这样你的单元测试 都会按照这个基准进行测试 ---- 对一个项目必须保证基础数据的维护***

----- **这个很多的大项目都会忽视 ---- 如果没有 后面很多事情都不好做 基础数据不准备好 否则 后面很多测试不出来**



**这里面几个注解 整个生命周期 就按照上面的**

测试类在创建的时候 @BeforeClass就执行一次 --- **这里面 Spring可以new一个Context**

@Before 每个@Test被调用 都会调用@Before这里面的内容 ---- Before里面可以准备数据 这里面可以初始化数据

@After 测试之后 可以清理数据

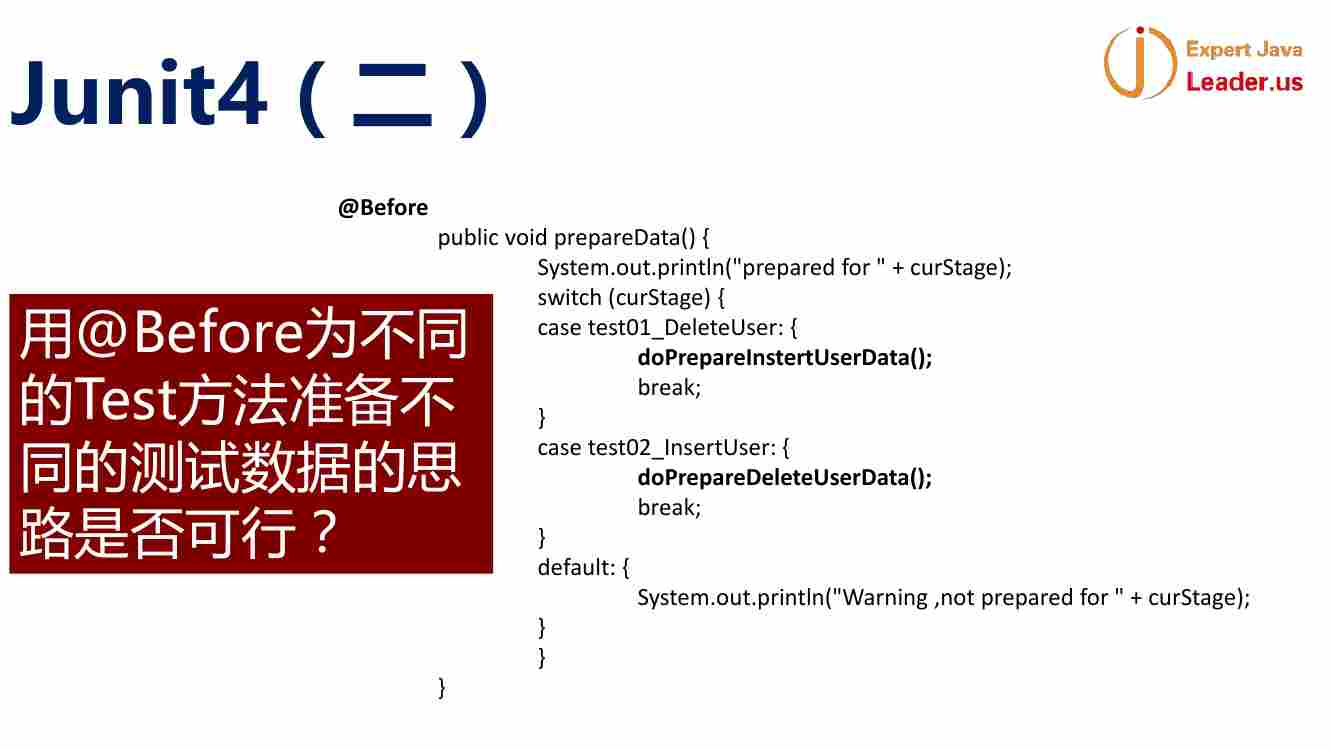
**关于测试里面的顺序问题**

都加了@Test注解 @FixMethodOrder(**MethodSorters.NAME\_ASCENDING**)

如果测试方法有先后的依赖关系 前面过来 要通过这种方式保证 --- 这个只能够**按照名字排序 ---- test01 test02 …. test0N 这个是保证顺序的**

**如果有测试有依赖关系 现在只有按照名字进行排序的测试 一般不建议测试之间有相互依赖 否则很容易测试失败**

不建议测试方法之间有依赖

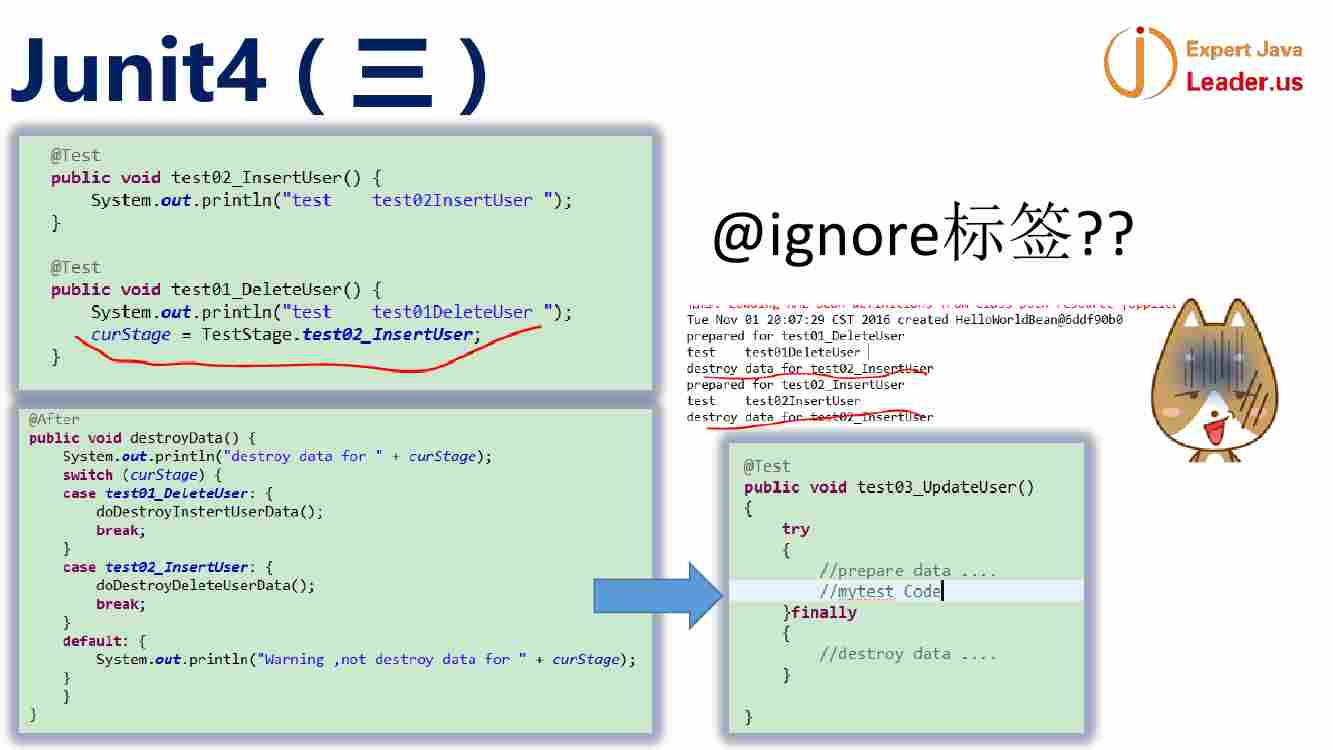
【**这个思想非常好 就是为不同的Test方法准备不同的数据 后面给否定了**】

@Before

这里面加了一个状态标签 curStage

如果是test1 为test1准备数据 如果是test2 就为test2准备数据 ---- 感觉好像是可以的 告诉下一个阶段是哪一个阶段 进入之前 准备新的数据

Destroy 也按照这个阶段走



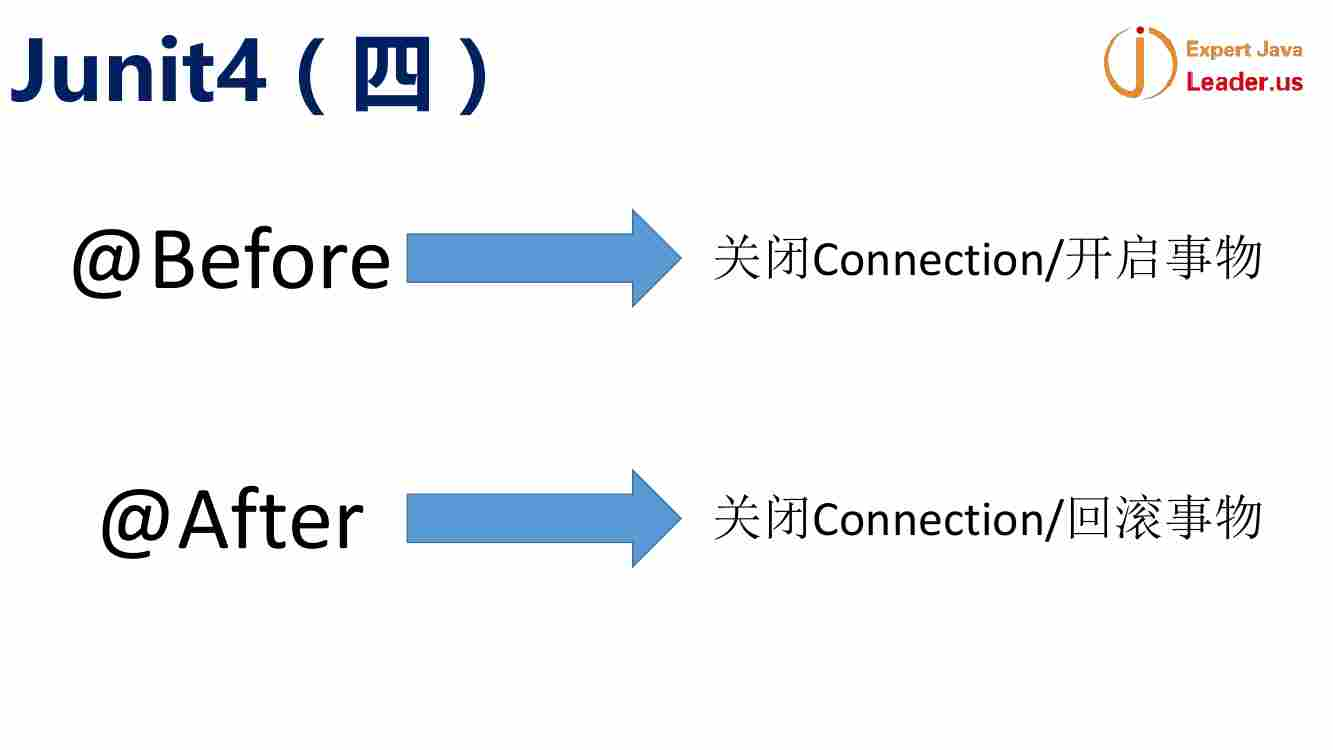
但是 如果这个状态写错了 那么 这个数据的准备就乱套了 哪个方法一旦ignore 就会乱掉

【@ignore是 忽视测试方法 】

有一个问题： 就是我的上一个test为下一个test准备数据【这里面的CurStage】

但是 如果某一个@test加了@ignore标签 这个状态就不对了 这个代码就绕不清楚了 状态在哪写错了

所以 **使用@Before和@After为每个测试方法准备数据是不可行的**



那么一般情况下 @Before和@After可以分别做 开启Connection /**开启 回滚事务**【PPT都写成关闭了 并且 事务 不是事物】

假设自己控制事务 @Before和After 【回滚事务 的意思 就是 还原数据库中的基础数据 为下一次 或者别的测试做准备】

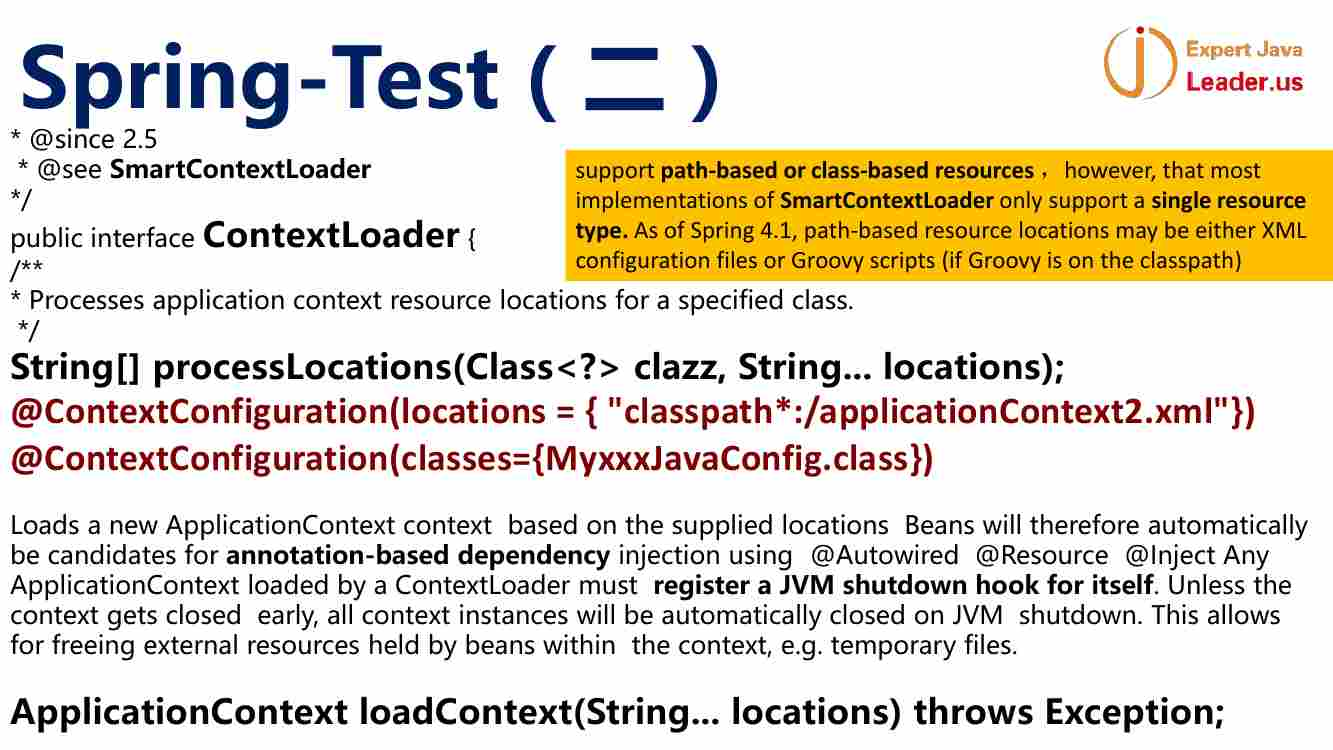
如果做跟状态有关 具体的方法 有关 就非常难了 ---- 也就是很难为特定的方法准备特定的数据 ------ 这两个Junit注解做的事情

下面Spring引进了SpringContext



Spring-Test 为了单元测试 做了很多 引入了Spring-test这个子工程 为单元测试做了很多 定义了很多API ---------- 为了application context加载 和 事务回滚 都做了很多 实现了一套机制

为了模拟很多环境 ---- 比如测试Servlet 都提供了 Servlet上下文 也提供了 是很复杂的 【后面有待仔细研究 这次先算了】



这个**ContextLoader*不是spring-core中***的一部分 而***是Spring-test***中的一部分

说的是 我要加载上下文 定义了一个ContextLoader接口 支持的是class-based或者path-based

[ppt需要好好看一下]

**有趣的是 这段源码中的两个注解不能混合在一起 不能是一部分class 另一部分path**

然后 这个C**ontextLoader做的是 为了单元测试 加载我的vSPring Context** 这个是**非常复杂**的

就是**ContextLoader的loadContext返回的是一个ApplicationContext 这里面涉及到测试过程**

这里面涉及到测试过程中 **上下文的关闭的过程 作为一个Loader的概念**

【PPT自己看】

还有个SmartContextLoader这个就是ContextLoader最顶级的实现类了

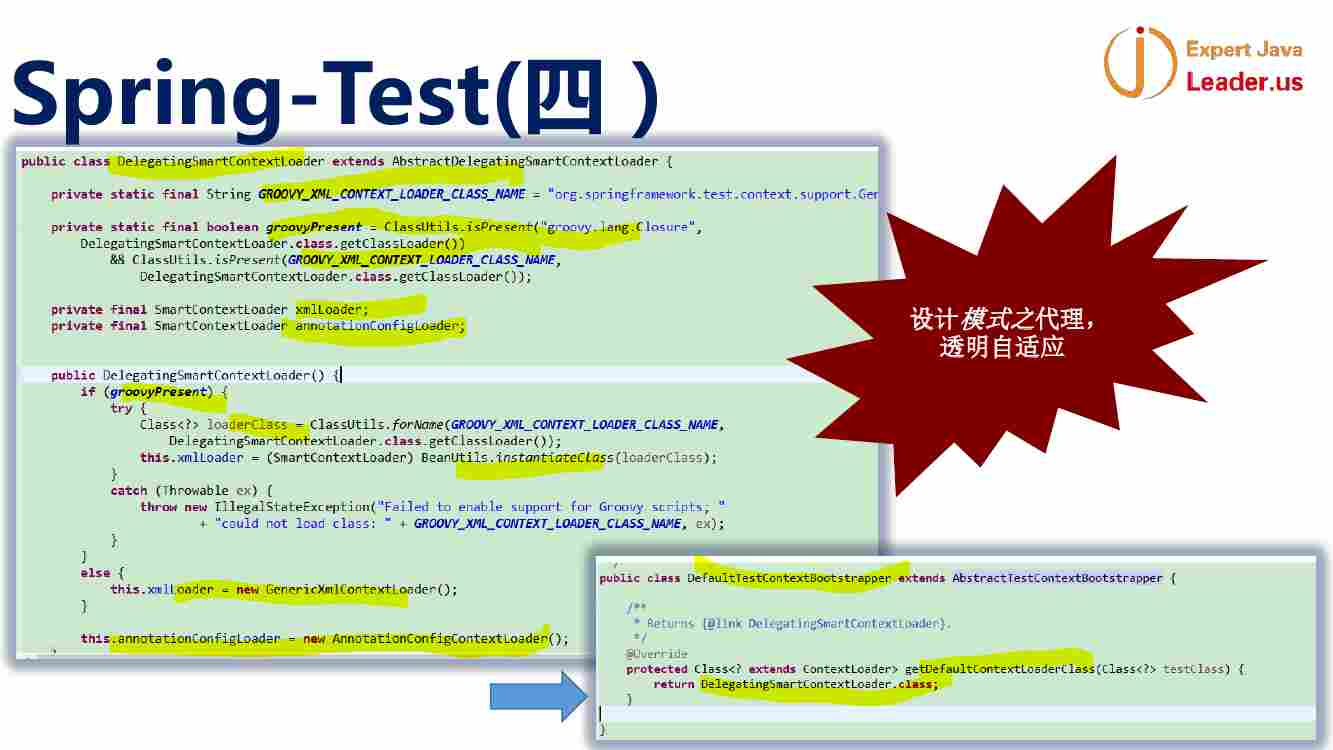
看看这个里面继承了ActiveProfiles ----- 也就是前面的Profile的配置 等等 支持的

各种ContextLoader

代理 注解 XML web代理 注解配置 Web方式也有三种【和后面的ApplicationContext很像 对应的 web的也有三种】

有三种Web的ContextLoader

这些ContextLoader都不是为了实际使用的 都是为了测试



DelegatingSmartContextLoader 非常有趣

这里面比较有趣 就是 DelegatingSmartCotnextLoader 这里面 如果**发现有一个groovy的类库** ------ 类存在的 **就加载了Groovy的ClassLoader**

也就是支持这种语法的

如果不是 做一个xmlContextLoader 然后 加载了AnnotationConfigContextLoader

这个ContextLoader优先支持Groovy 如果不存在 xml 和Annotation

这个里面 **就是smart体现在 各种情况 我都支持 --- groovy我也支持 xml的我也支持 annotation的我也支持**

---**透明自适应** 的意思 就是 你不知道发生了什么 就都支持了 ---- 没看到的 都支持 代理不仅仅是转发 可以做很多事情 做的非常智能 对外屏蔽了很多复杂的逻辑

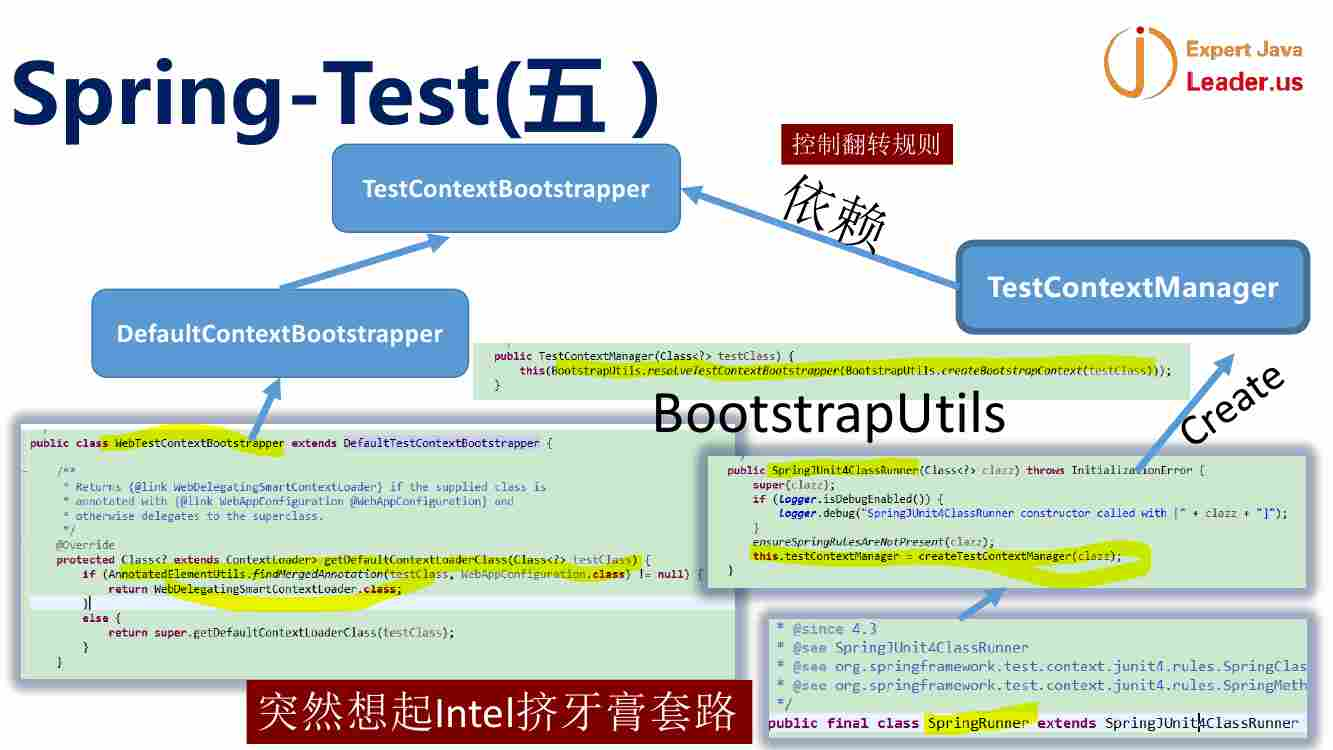
做了很多改变 ----- 就是代理不仅仅能做简单的转发 还能做很多复杂的事情

---- Spring的设计来看 做到了一种智能的情况 也就是 复杂的逻辑 外部看不到 ---- 内部自己自适应切换

下面的这个 **DefaultTestContextBootstrapper** 就产生了一个ContextLoader

这样就创建了Smart的代理 【说的是里面的方法】

--------------------------------------------------------------------- 20:51



这里面有一个DefaultContextBootstraper 这里面扩展了一个WebTestContextBootstrapper ---- 做法就是你的class加了WebApp的注解 就做一个web的Delegate的ContextLoader -----否则到默认的

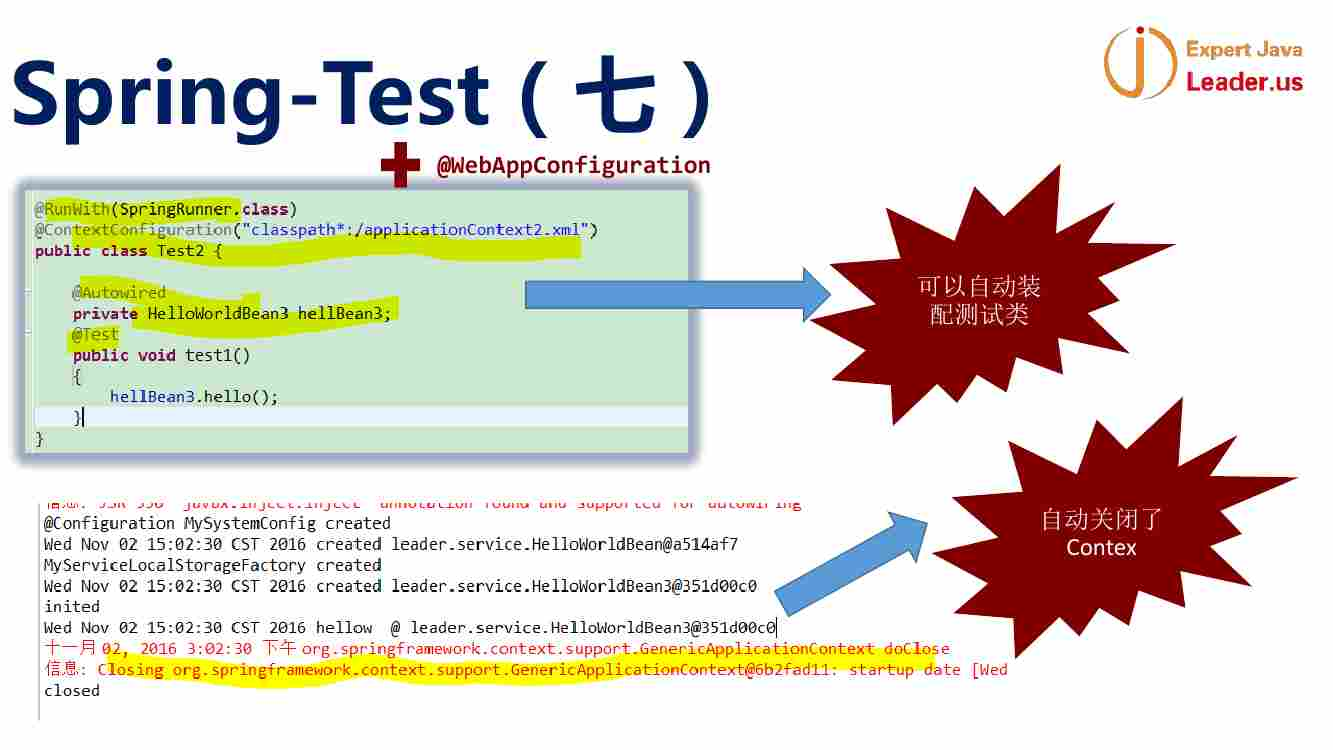
TestContextManager是Spring的 SpringJunit4ClassRunner创建了这个对象 ------ 这地方创建出来一个合适的ContextLoader来加载context

在Spring4.3的时候 做了一个简化 看到有一个SpringRunner这个标签 ---- 这个SpringRunner这个类来穿件这个TestContextManager ----这个Manager去找一个合适的ContextBootStraper ----- 然后产生一个ContextLoader 来进行单元测试

机制很复杂



看一下这个TestContextManager 里面除了产生合适的Bootstrapper加载Context之外 还有就是加了很多重要的testExecutionListeners 这些Listeners做了很多重要的事情



SpringRunner驱动了ContextManager 然后加载过程中 加载了classpath下面的这个xml进行测试

非常有意思的 这个test类不属于bean 但是可以被注入 autowired

Spring的测试类能让test本身也被autowired

\*\*\*Test类中可以直接引用container中的bean 这个是非常重要的

这个结果中 自身没有明确用到上下文context的引用 但是 能被spring托管 并且注入 这样非常简单 ---- 自动装配 进行测试

这个完成之后 上下文被自动关闭了



Listener在前面的Manager中有一个list

这里面有很多方法 ---- afterTestClass afterTestMethod beforeTestClass beforeTestMethod ---- 再给个就相当于junit中的四个注解

比注解强大在哪？告诉你当前测试的v方法是哪一个方法

这样 我们自己实现一个这个AbstractTestExsecutionListener

实现这里面的方法

然后 我们在Test中 添加一个注解 @TestExecutionListener 让我们的listener加入到test中



运行的时候报了一个空指针

看空指针的堆栈情况

分析代码 意味着Hello3是空 但是 这个是被autowired注解的 为什么为空？

这个只加载了我们的listener 没有加载默认的listener

\*\*\*\*所以 正确的做法 就是 @TestExecutionListeners(listeners = MyTestListener.class, mergeMode=MergeMode.MERGE\_WITH\_DEFAULTS) 非常重要\*\*\*\*\*\*

这样就把默认的listener添加进去了【有监听器 实际上就是观察者 这个观察者模式如何体现的呢？ 自己看下】

Debug的过程中 就可以看到 有一个DependencyInjectionTestExecutionListener [黄色框中的信息]

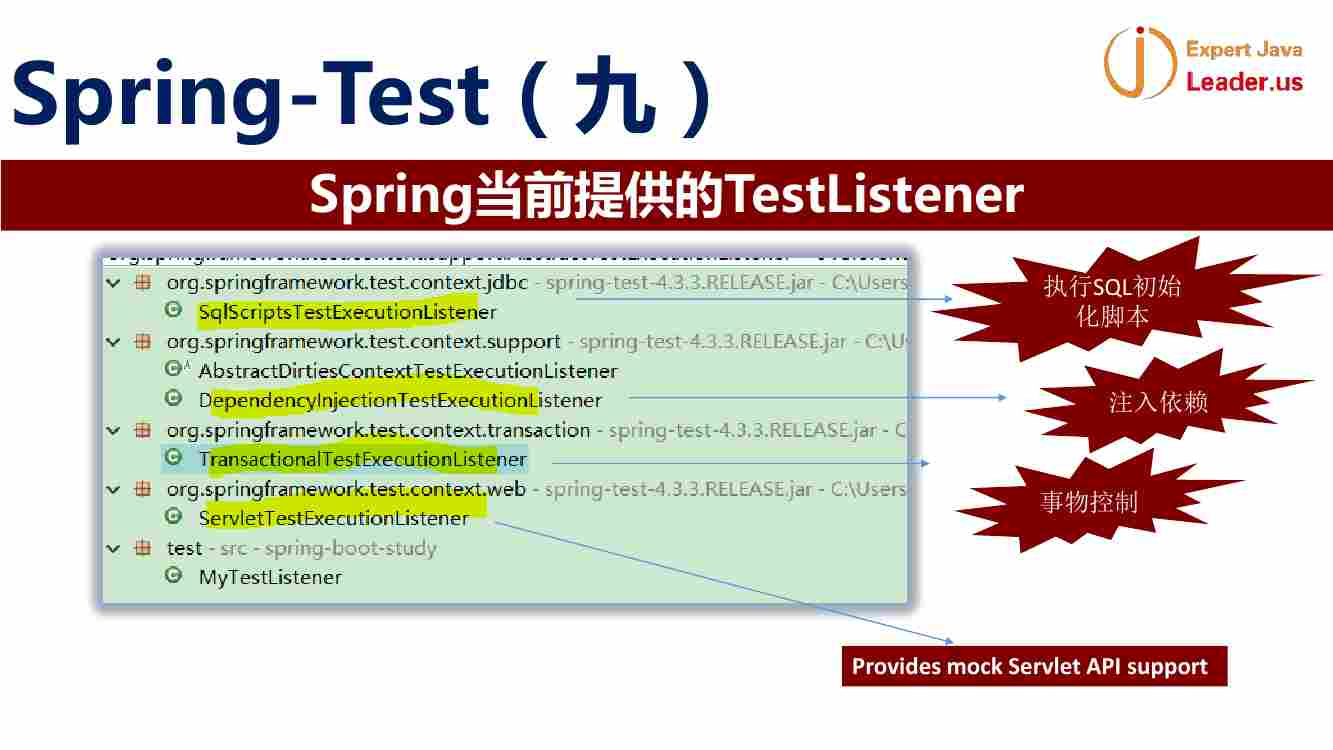
这样@Autowired起作用了

刚才这个没有了 autowired就完蛋了

还有就是TransactionTestExecutionListener是为了让单元测试中的事务进行回滚 ------ 刚才的listener ---- 这个listener里面就有刚才说的before after方法 这样就能让方法开始测试之前开启事务 方法结束 进行事物的回滚 等等

还有一个SqlScripts ---- 前面提到了 希望做一些数据库数据的初始化 这个SqlScriptsTestExecutionListener做的就是这些事 --- 配合一个SQL注解 在测试前和后执行一个SQL脚本

Spring做这个测试框架 背后这些事情 精华就在这些listener!!!!



这个执行成功了 就通过

然后 还有一个ServletTestExectuionListener ---- servlet api需要容器 这里面用mock方式支持了

Sql初始化脚本 后面会用到 --- 可以看看里面的代码



Sql注解 单元测试中写一个sql 完成准备信息

无论有没有事务 都可以执行 --- 没有的时候 通过Datasource去查找



这里面对mock做了一个介绍 有些条件很难准备 可以使用mock做依赖 --- 这里面 datasource的内容就mock出来了

---- 使用的是 DataSource mockedDartaSrc = Mockito.mock(DataSource.class);

Connection也mock出来了

Statement也是mock出来的 【用的函数式编程 后面】

用法简单 给个接口 mock出来一个对象

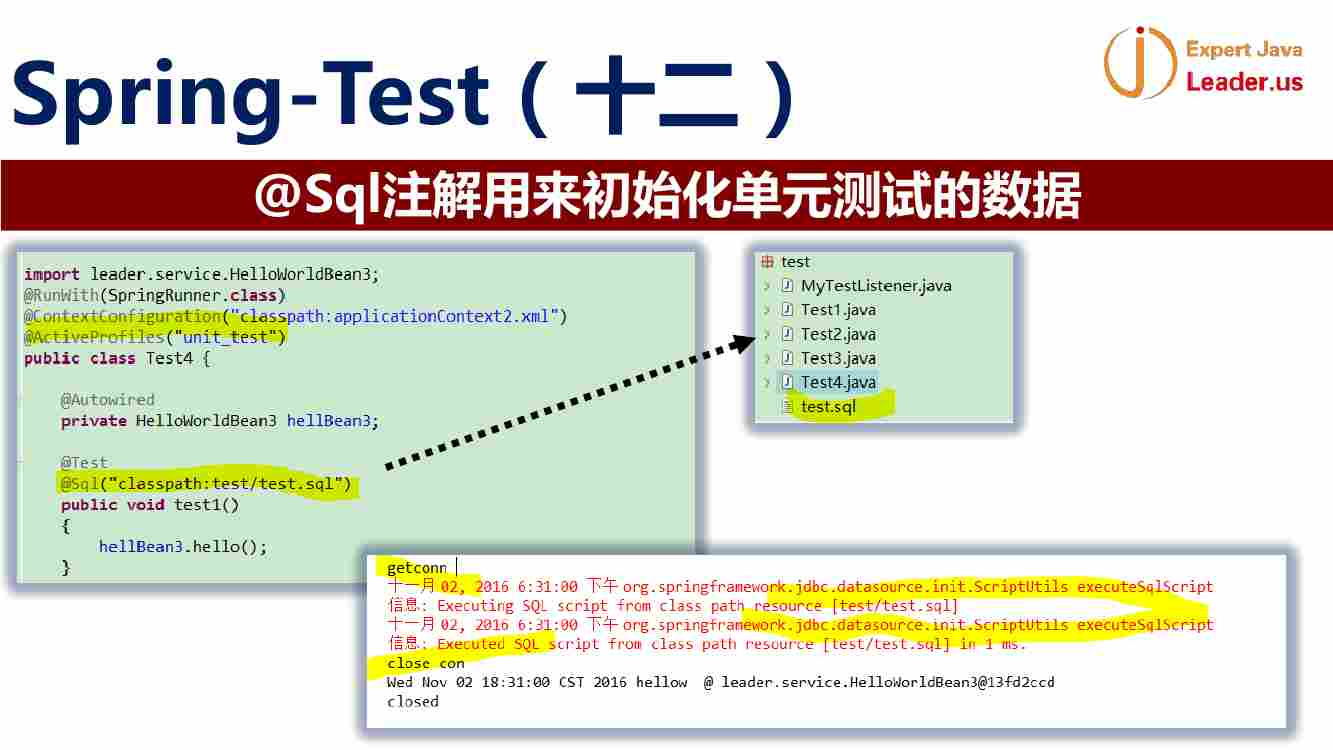
后面把这个装配起来

---- 如果datasource的getConnection()被调用 就打印一句话 得到一个新连接 返回connection

False被调用 没有返回值

后面各种打印出来

这里面操作resultset的时候 还是需要进行mock模拟 ----- 这个清楚到底做了什么事情 不用去看数据库 这段代码非常有用



单元测试的时候 加了Sql注解

可以看到 这个单元测试 有getconn 和 closed 就是分别得到了单元测试的链接 然后执行了test.sql中的内容 最后关闭了sporing context

这样 学会了使用mock的方式创建对象 和 sql注解准备数据



Spring框架下 创建和rollback每一个事务 【PPT自己看】

但是 我们看看

前面有一个TransactionTestExecutionListener来做到的



看这个例子 准备DataSource --- datasource()这个里面的方法 都是mock的

做好这些准备 做一个测试

最后代码中不是想象的 是commited ---和前面的ppt不一样 【意思就是 应该在结束的时候 还原数据 所以 要回滚 所以 回滚是期望的动作】

这是单元测试少了一个信息

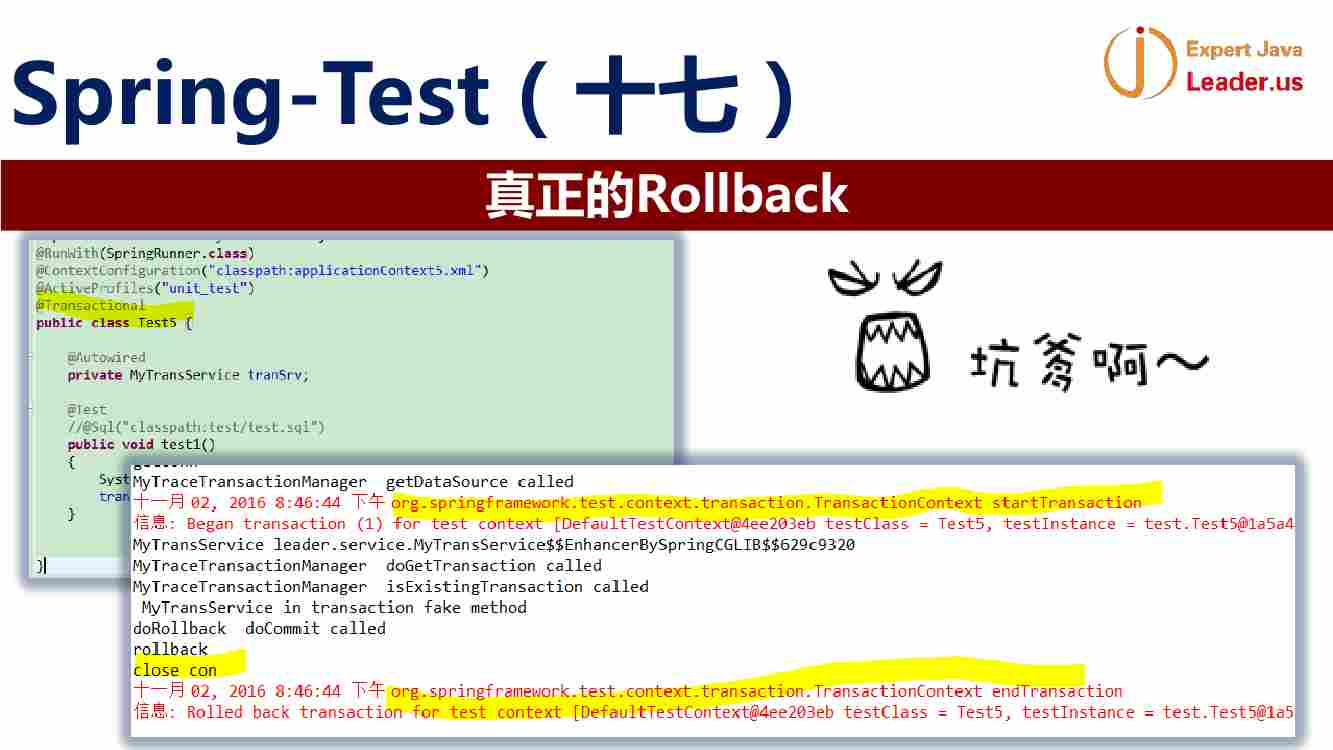


但是 是被commited ---因为测试上面没有添加Transactional的这个标签



可以看到 这里面 这个对象 也被事务包装了 生成了spring的代理类

但是 为什么会被commited呢？



真正只有加了这个@Transactional之后 spring的context才会加这个才会表示 这个是和事务相关的测试 这样 这个时候 才要会把事务取消掉 所以 打印出两个黄色背景的话

背后的秘密就在于 TransactionalTestExecutionListener ---- 去扫描 这个测试上面是不是有@Transactional 是不是有这个注解 如果有 采取rollback 否则什么都不做

