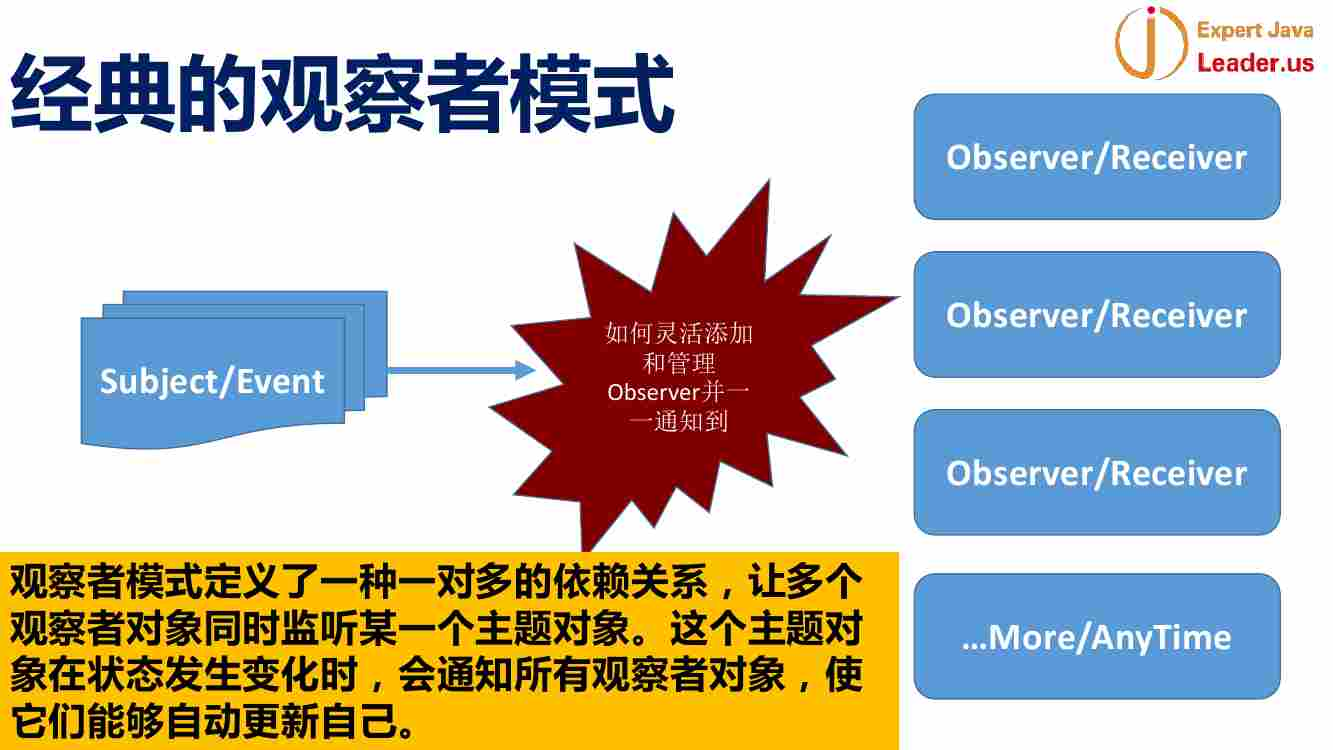




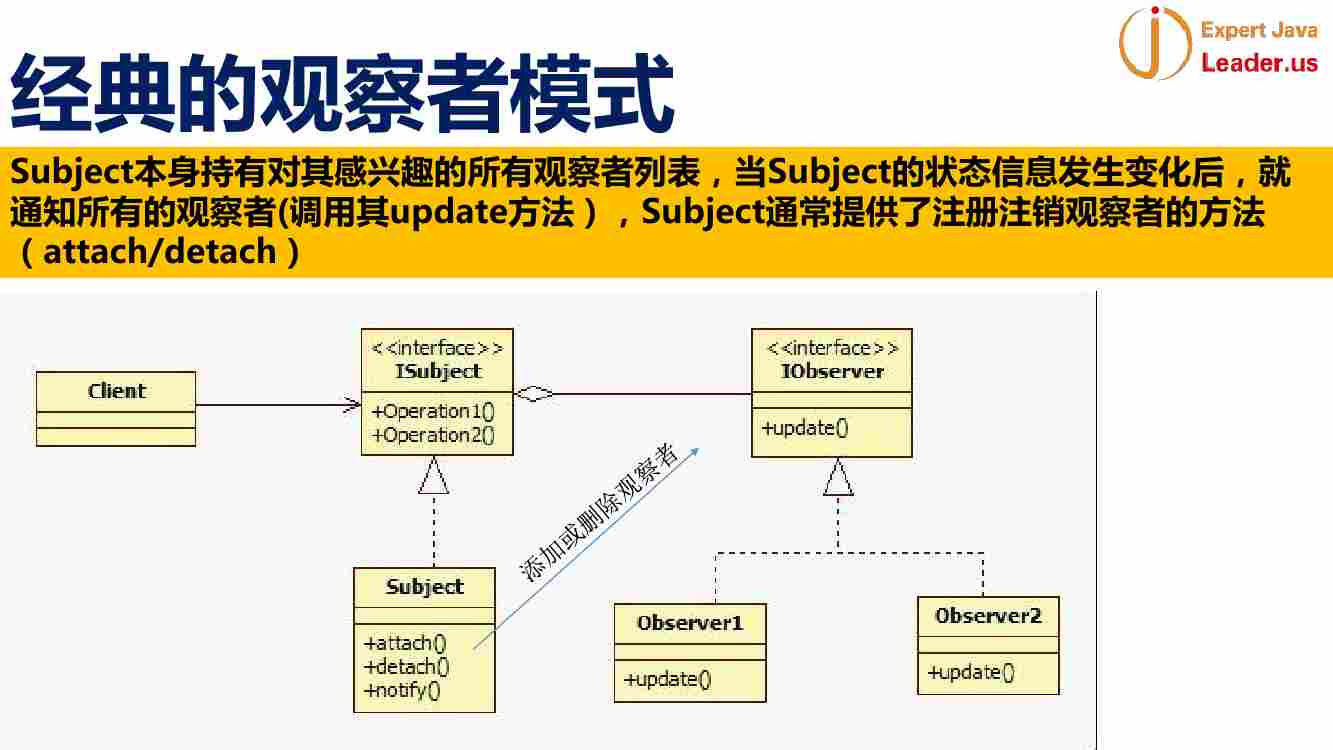
通过DIY一个简单的框架 来掌握Spring



Spring中用到了观察者模式 观察者模式定义了一个消息主题 可以被多个观察者或者消息接收者监听到 并进行响应

可以动态增加或者移除观察者 让这个消息或者事件的产生者和接收者分离开 进行了解耦合

---- 这是一个影响大的模式



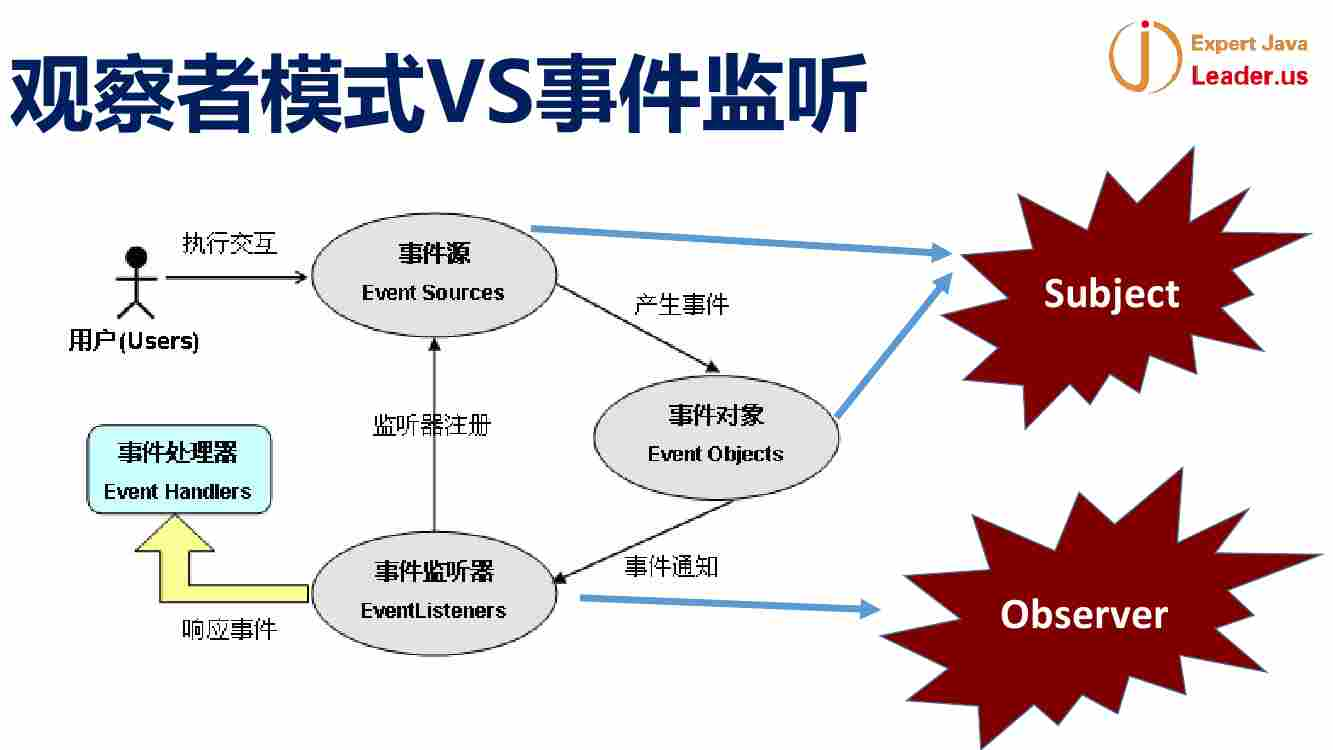
Subject中有一个attach()和detach() 增加或者删除一个观察者

当notify可以通知所有的观察者 观察者有自己的业务响应

可以动态增加或者删除观察者

具备灵活的扩展能力

添加或者删除不同的Observer进行处理



事件监听这个模式也是一个观察者模式 在UI中用的非常多 用户会产生一个button click等等

--- 事件源 Event Sources 对应 的起始是Event Objects ----- 会把事件源包装成一个Event Objects

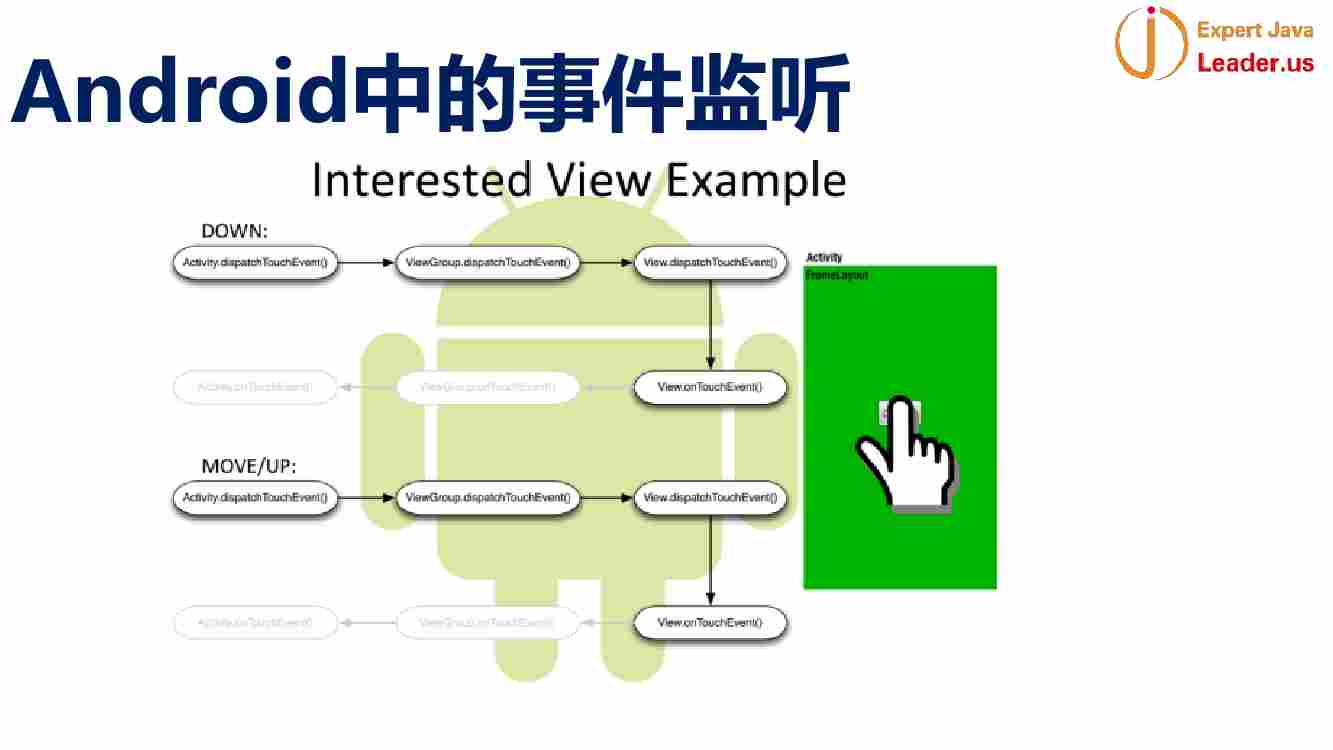
\*\*\* 比如 会把事件鼠标点击了哪一个按钮 或者 鼠标移动了多少个像素等等 产生一个Event Object

来看 这个事件源和事件对象相当于一个Subject

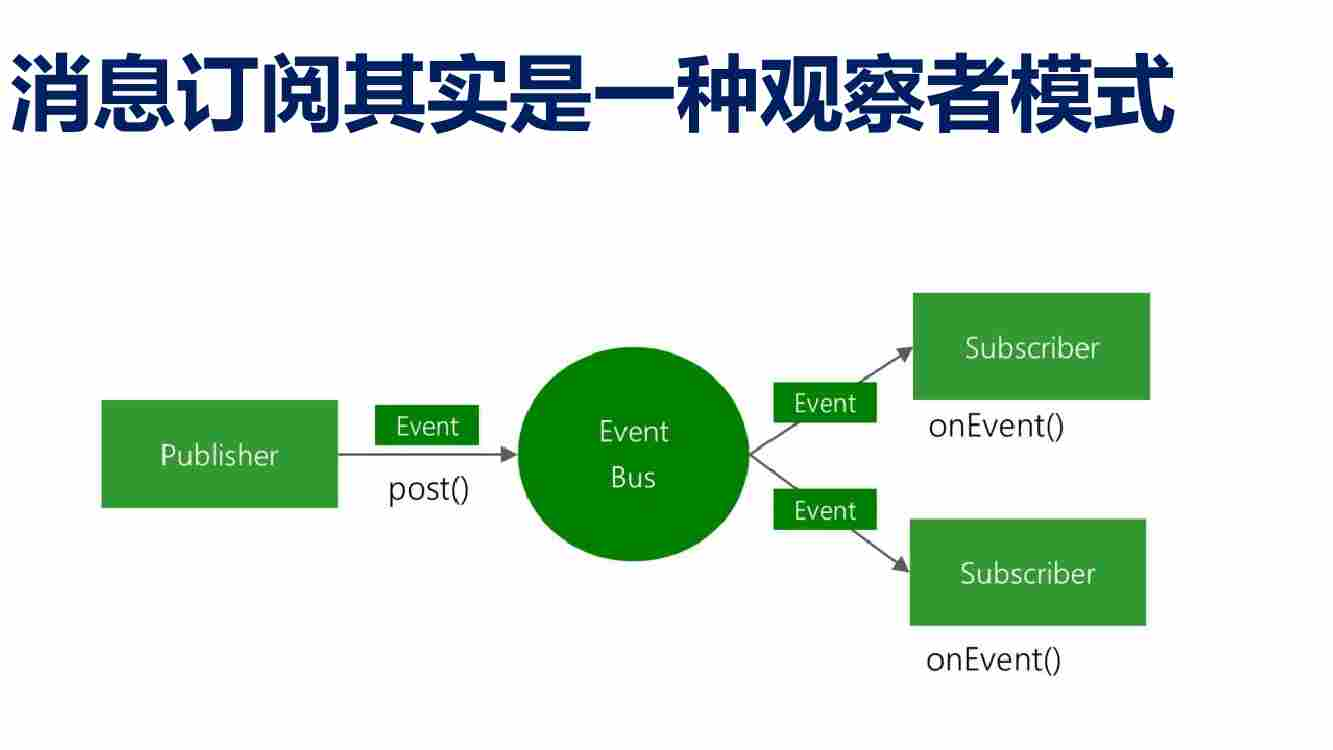
Subject是可以监听消息的接收者的 这里面的EventListener就相当于Observer 【英文的字面意思 观察者 事件监听者 监听不就是一种特殊的观察么】

整个过程就是 Event Sources会生成Event Objects 之后 会调用通知方法 通知所有在Event Sources中注册的Event Listeners

通知之后 每个EventListener就会按照自己的EventHandler中的回调方法进行业务处理



安卓中的UI也是这样的 ----- 不同的listener做一个判断



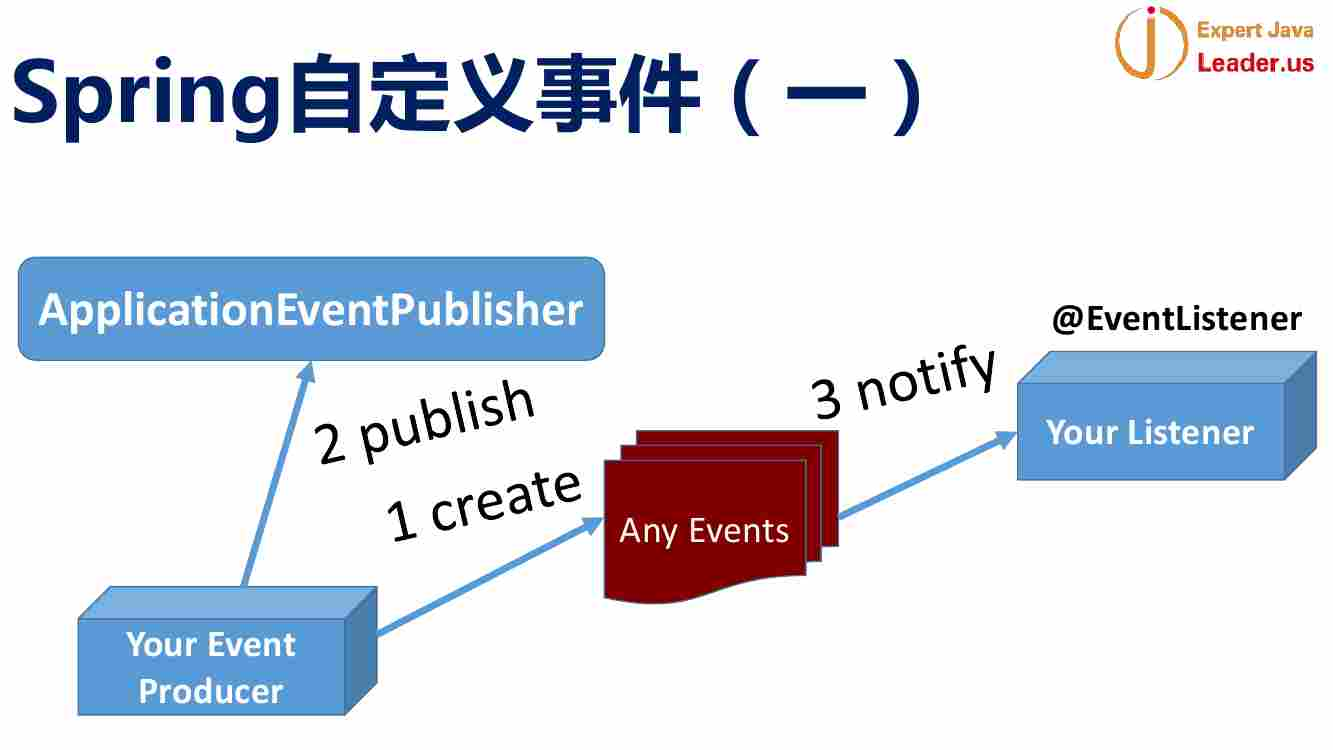
一个消息的产生者 Publisher 产生一个Event 放到Event总线上 然后 把这个Event发送到订阅者身上 ----- 这也是一种观察者模式



Spring容器中 某一个Bean产生了一个消息 可以发布publish到spring容器中 ----- 其他bean注册并监听了这些消息 ----- 消息生成的时候 会被通知

注意 这个是在Spring容器中 一个bean产生的消息 被其他的Bean接收

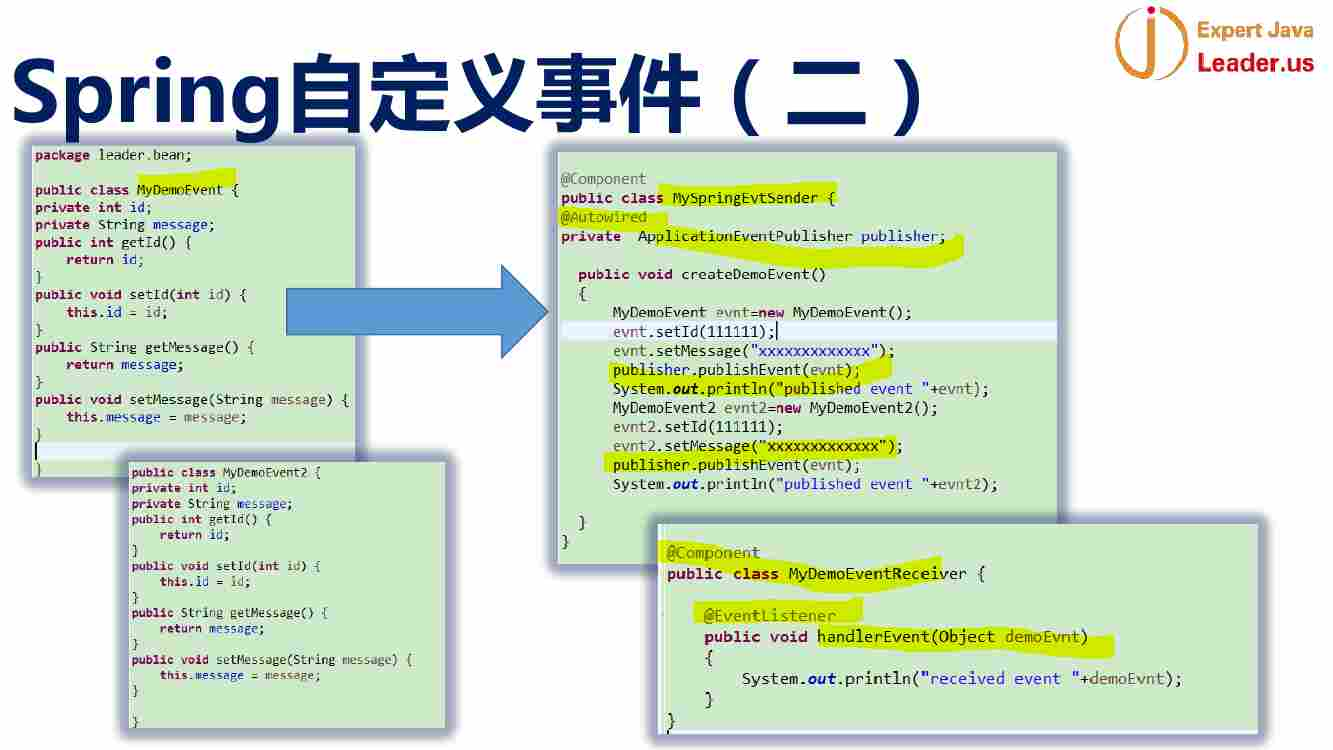
这个消息不是JMS消息 是内部简单的消息



这个spring自定义的事件发布的过程 大体流程：

首先 某一个Bean(Your Event Producer)生成了一个事件 Any Events ------ 在这Spring容器中 把这个消息交给ApplicationEventPubslisher进行广播 ------ 之后 监听了这个事件Any Events的其他组件bean就会感知到这个消息

我们的观察者使用@EventLIstener进行了标示

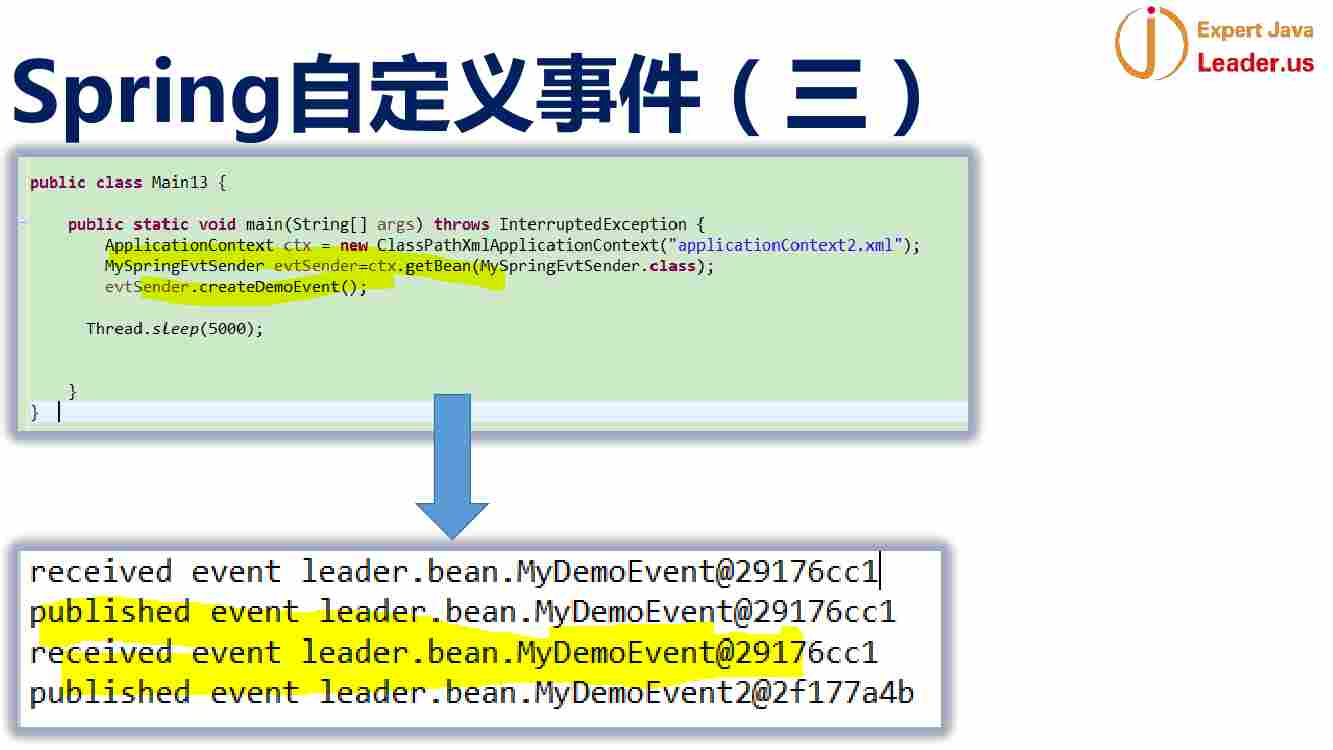


定义了Event对象 POJO ----- 在我们的Bean Producer中的某个方法中  **持有了一个ApplicationEventPublisher** Autowired之后 ----- 在我们的业务方法中 生成我们的消息 然后通过这个EventPubisher把消息发布出去

我们的监听者 或者观察者 要在接到Event发生的回调方法上直接注解 注上 @EventListener

这个@EventLIstener会被Spring进行了纳入

【按理说 我们这里面 publisher本身就是subject ----- 但是 为了不侵入 所以 直接聚合就可以了 -------- @EventListener标记的就是Observer 就是观察者 ----- Spring容器初始化的时候 会被注入到Subject中 ------- 这样持有ApplicationEventPublisher的类就相当于外部调用的subject的业务类而已 ----- 这就对上了】

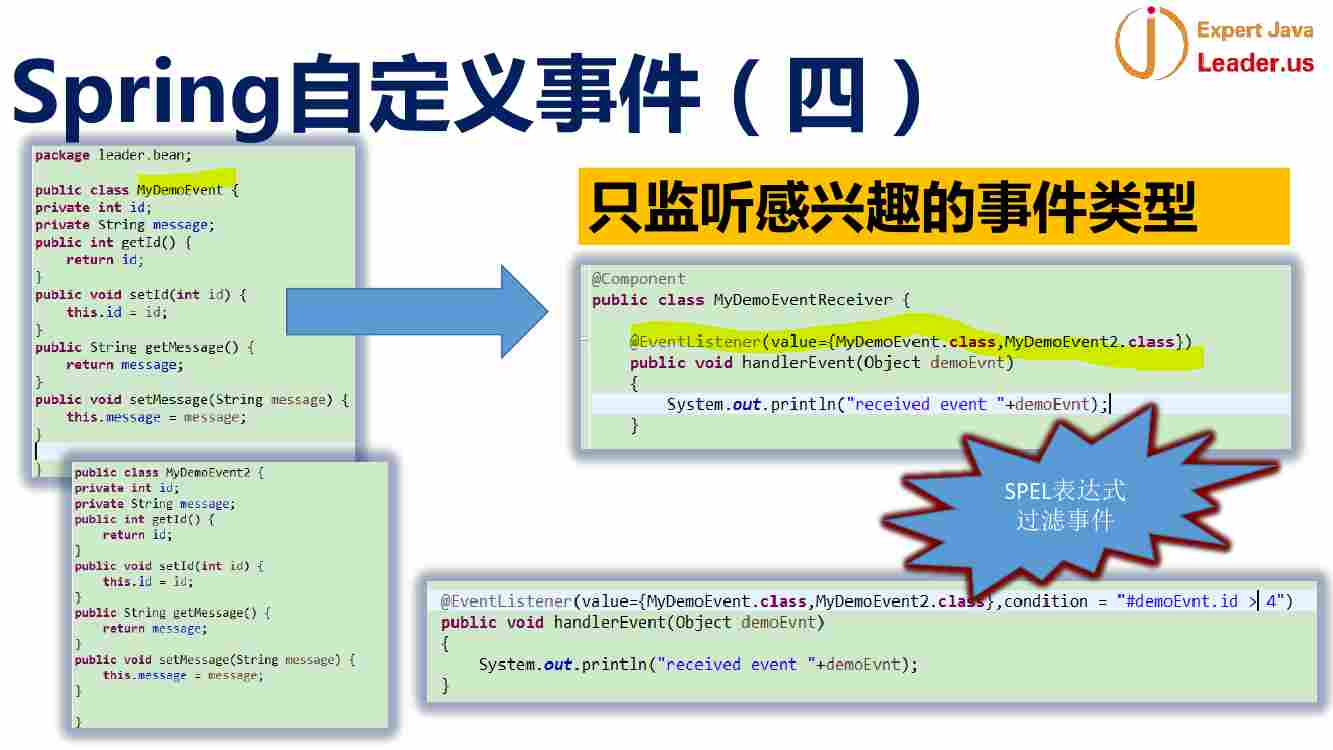


代码中 结果可以看到 publish 然后有人接收到

这样在Spring中消息发送非常简单

消息机制非常重要 是异步的

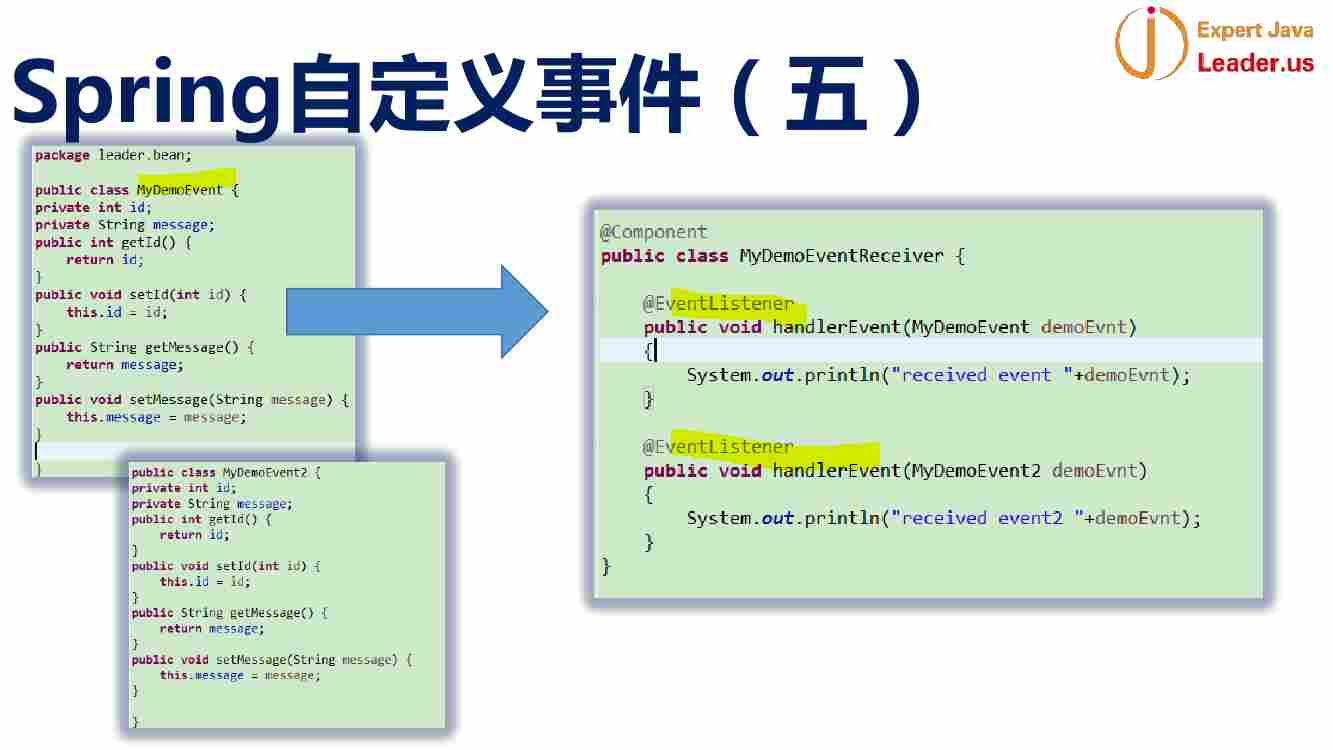
我们产生订单的时候 把这个做成消息 其他的bean去处理这个业务



传入具体的class就能管写特定的Publisher ----- 上面只处理那两种事件

加入SpringEL之后 可以表示 处理这两类消息中 id>4的

非常灵活



这个在高端面试中会问到 做一个IOC容器

做一个这样的容器 会对内部的机制有深入的了解



Spring发展有10年了

这个大世界

02年开始了 04年第一个版本出现

09年之后被收购了 现在已经4.x了

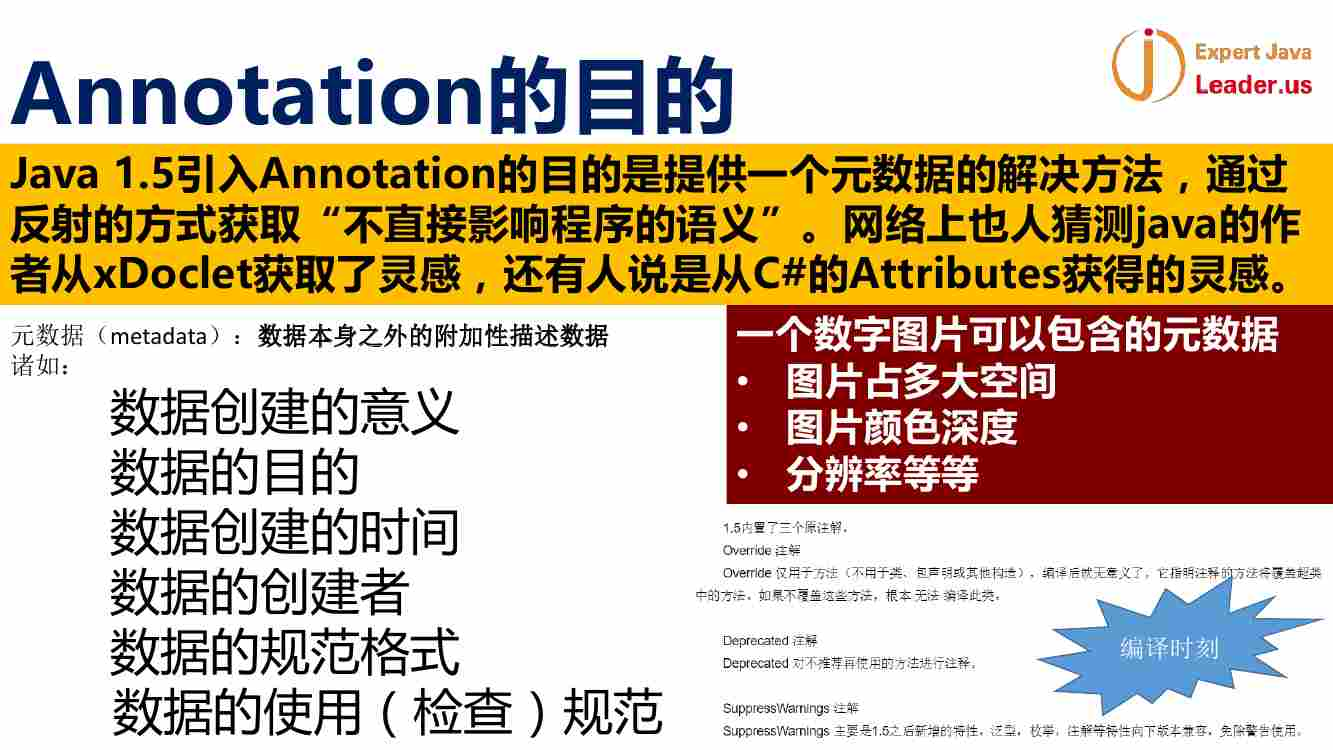
最后2.0做了改变 conext:来配置

2.5往注解的方式靠拢

3.0进一步靠拢

最后4.0支持了Java8的用法

我们全部使用了注解



所以我们看一下基于注解的框架

Annotation是JDK5中引入的 提供了元数据

数据库中一个表 有两部分 定义 每一个字段 字段类型 真实数据是表中一行行的记录

元数据是描述数据的一种数据

有更多的引申过程

Java中认为Class是Object的元数据 Class描述了Object

MetaData非常重要

Ppt中 创建意义 目的 作者 等等

比如 对于图片占多大空间 尺度大小 分辨率

Java中内置了三个元注解

@Override ----子类覆盖了父类的方法 编译之后 就没有意义了

@Deprecated 某一个方法不适用了 调用者方式 Java就会警告

@SuppressWarning的时候 一直警告



注解有时候属于源码的一部分 有时候不属于

这里面有三个元注解 @Documented @Retention @Target

@Documented

@Retention 注解在源码级别呆 还是Class 还是JVM中

一般情况都是Runtime

Spring中的@Retention大量都是RUNTIME的 Runtime会伴随这class加载到class中

@Target标示这个注解用在哪里的 方法 还是类 还是其他

@Inherited标示子类是否能够继承父类的注解

注意的

在一个Inreface上面注解 子类是无法看到的 这个是很特殊的

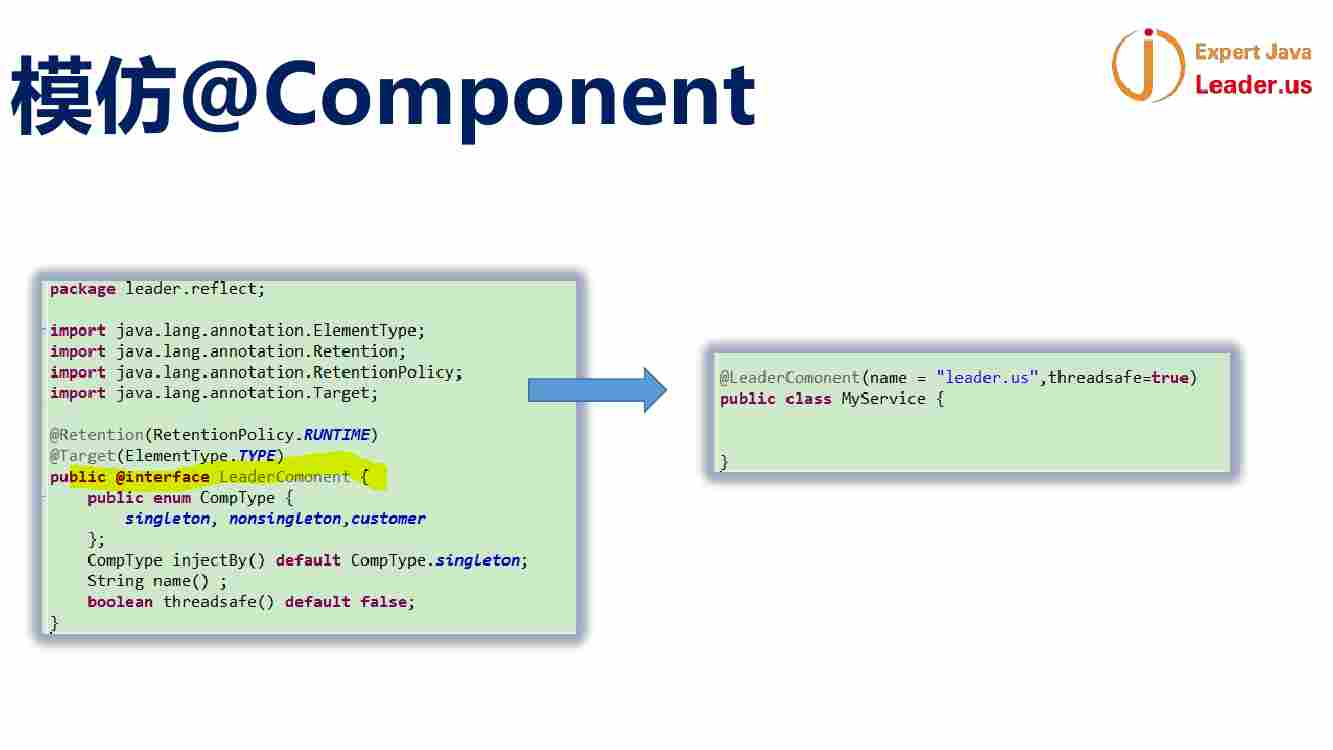
Spring中的很多注解都是继承的

注解的定义方式@interface

注解里面可以定义很多属性

注解本身就是包括本身的特殊的一个类

注解本身代表了某些属性的数据结构



假设有一个注解 模仿Component注解 定义在ELMENT上 定义在类上 这里面是不能定义在方法上

LeadComponent这个注解是一定在RUNTIME 并且定义在RUNTIME的

比如 注入的方式 injectBy() 名称 和 threadSafe()

这个用法 有图

这个就模仿了Spring的Component

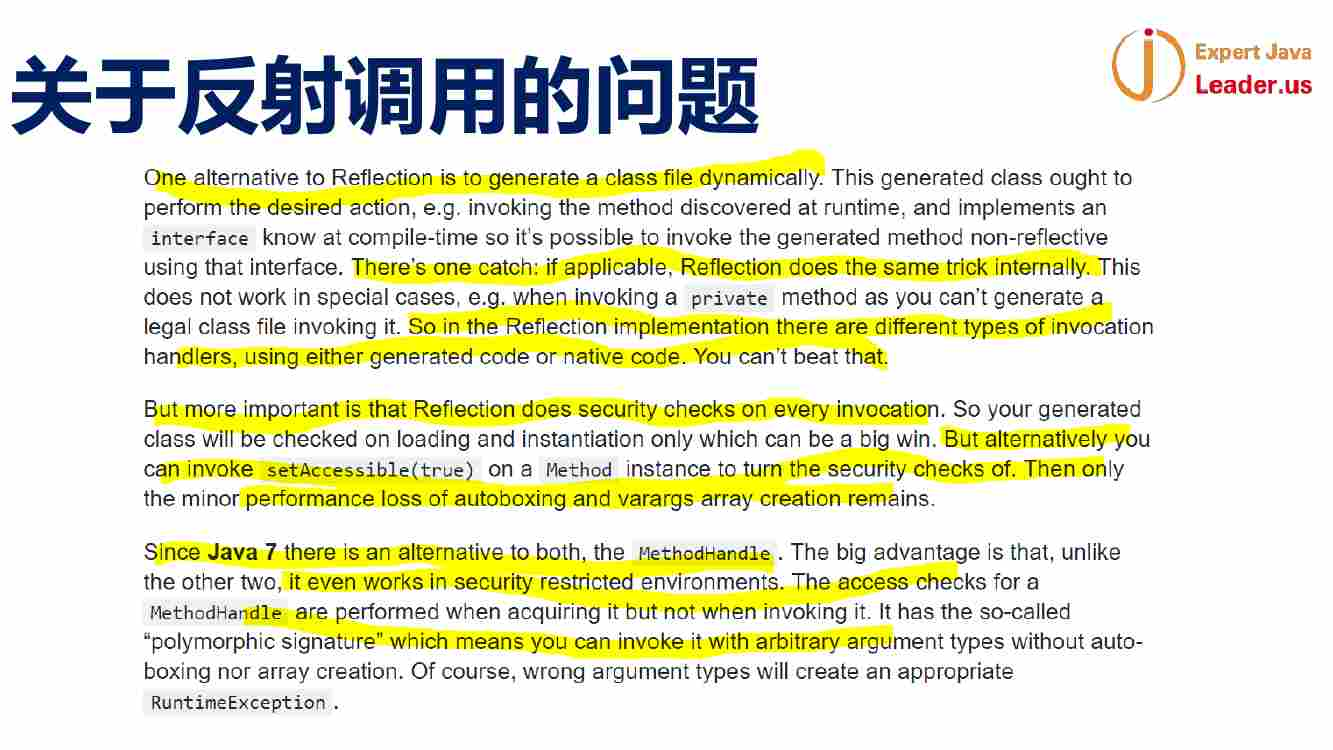


我们是能够通过class来得到注解的 MyService.class.getAnnotations() ---- Annotation[]数组

这里面就是观察 是不是有LeaderComponent这个注解

发现了这个被注解的bean

找到了被托管的bean



反射调用 可以调用某一个类的属性 和方法 ---- 在我们的组件装配的过程中 需要set某一个对象的属性 或者 new这个对象 这个都要用到反射 ----

【PPT上面的英文的翻译

一种替代翻身的就是动态的生成一个文件。这个生成的类应该能执行目标action ---- 也就是在运行期间调用发现的一个方法 并且 能够实现一个在编译过程的接口 ===== 这样 就可以用非反射的方式执行使用那个接口生成的方法。

有一个陷阱就是 如果可以用的话，反射也能够内部做这样的把戏。这个不会再特定的类上面生效 ==== 就是当调用一个私有的方法因为你不能生成一个合法的类文件来调用。【这没太懂】

所以 在反射的实现中，有各种饿养的调用的handler的类型 ---这些要么使用的生成的代码 要么使用的本地代码。你不能违反这个约定。

但是更重要的是，翻身会在每一次调用的时候都会进行安全检查。所以 你的生成的类文件会在加载和实例化的时候被检查 ---- 但是 有一种替代的方式 就是 你在一个方法实例上调用setAccessible(true)来关掉安全检查。但是 一部分自动装箱和可变参数的性能损失还是存在的。

自从Java7之后，上面两种情况有了替代 – 那就是MethodHandler ----- 这个不像是其他两种（生成实现接口的动态类和反射） 这个工作在安全性严格的环境下。这种对于MethodHandler的访问检查 是在获取到的时候就进行 而不是调用的时候。 他有着所谓的多态的签名 ----意味着 你可以使用任意的参数类型 调用 但是 没有自动装箱 也没有数组生成。当然 错误的参数类型会产生一个恰当的RuntimeException

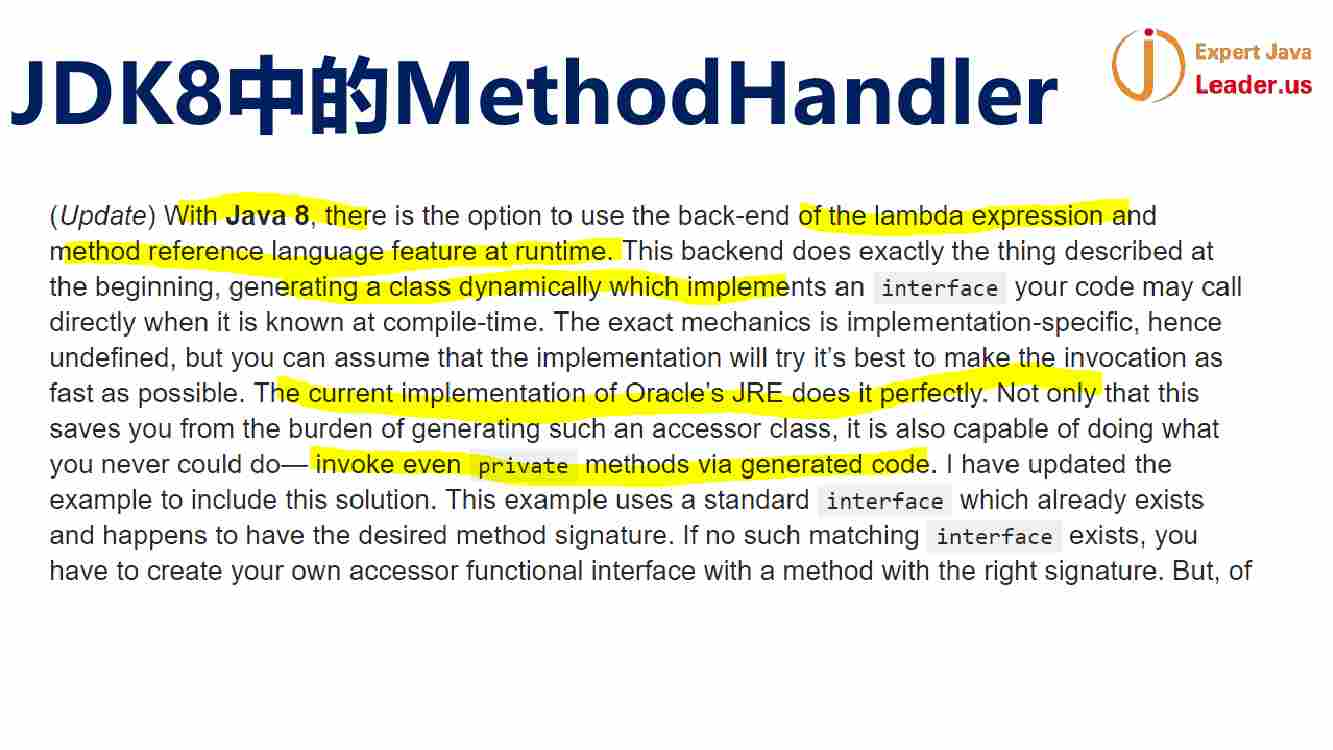
】

反射就是可以去调用某一个类的属性和方法 ---- 因为在组件装配的过程中 因为涉及set到某个属性 和new累的本身 ----反射有一个性能问题 每次都会做一个安全性的检测 虽然可以setAccessible避免 但是 这种反射有缺陷的

后面是MethodHandler替代反射

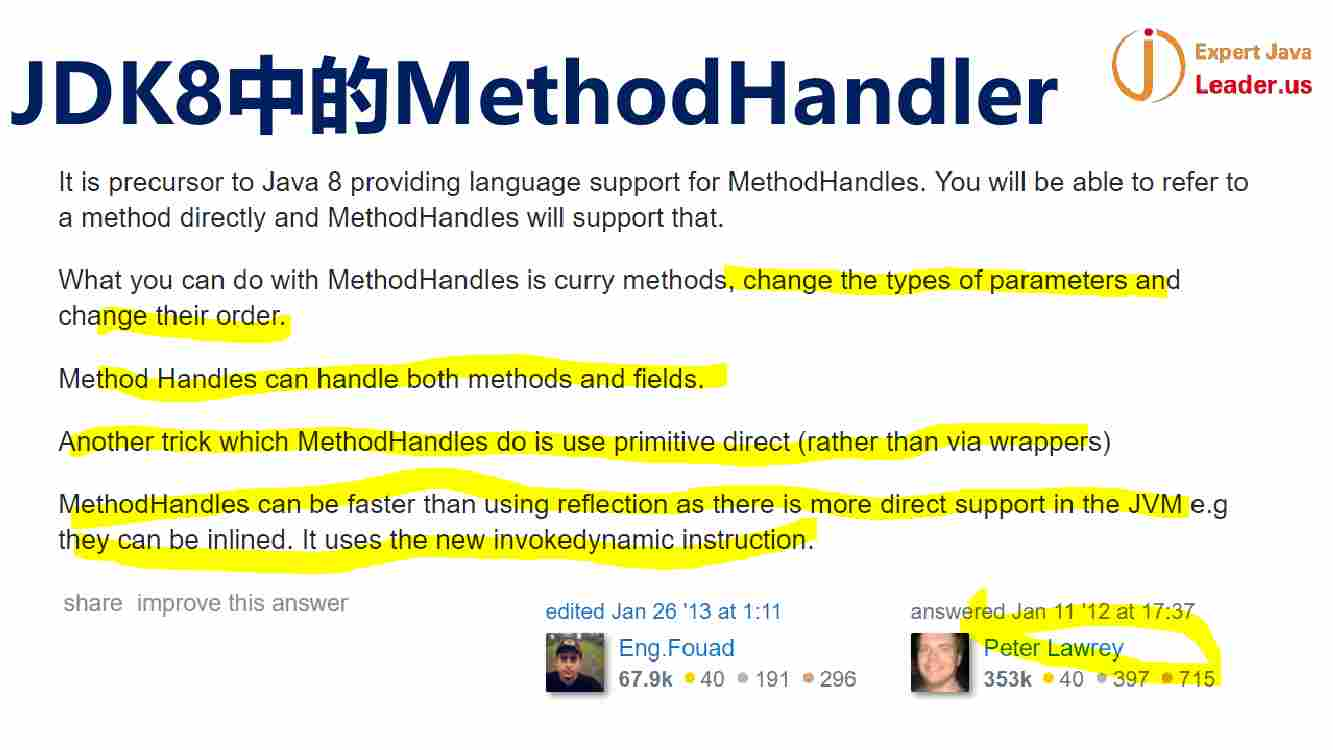
还有就是自动装箱和可变参数的性能损耗。

【MethodHandler要自学一下】



【JDK8中的MethodHandler ------ - 在JDK8中进行了更新 有一个选择就是在运行时使用了lambda表达式 和 方法引用的特性。这个后台可以精确的执行在开始阶段描述的事情。生成一个动态的实现了一个interface的----- 】

JDK8对MethodHandler有改进 可以使用Lambda表达式 另外对动态语言的支持加强了 ---Java中更多强调MethodHandler这种方式 而避免使用古老的反射API



这里说了 Java8中methodHandler已经非常重要了 --- 建议使用 可以做很多事情 改变参数 参数的顺序 – 方法和字段都可以使用methodHandler进行处理 methodHandler底层是直接调用 而不是像反射调用JVM的Wrapper类

MethodHandler比反射更快

Inline ---就是方法的调用（入栈出战的性能损耗）进行展开 ------ 就是说使用MethodHandler替代反射



第一种装配是反射的方式

第二种是methodHandler方式

最后一种是Spring4中DirectFieldAccessor方式

建议都试试



自动扫描一个包下的class 有些的错误



有很多的库可以扫描

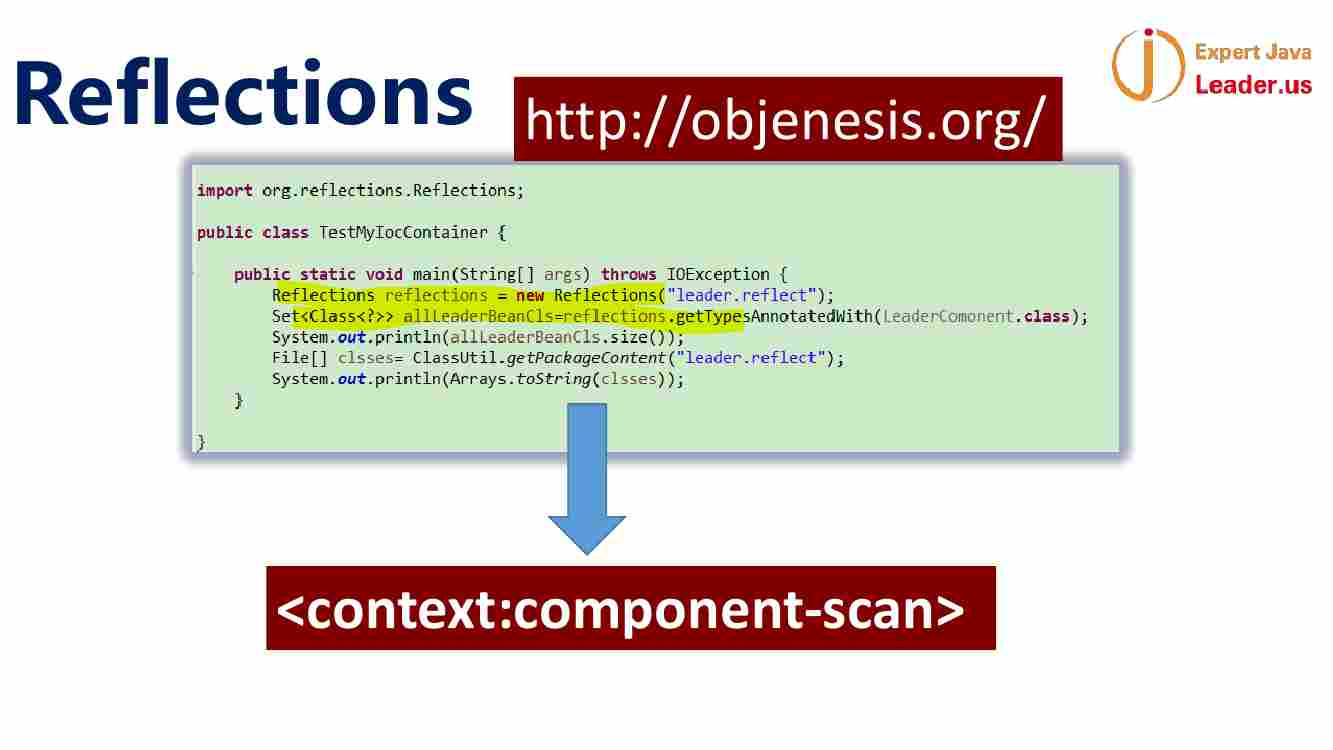


ClassIndex认为传统的扫描的时候很慢 所以 编译的时候扫描的时候 把扫描出来的类放到一个文件夹中 然后直接调用

这里做了一个比较

可以看到ClassIndex是最快的

这些建议 机制 用法都去看看



这里面用的是Reflections这个框架 ---- 执行好扫描的包 然后 就可以扫描 最后 就可以输出了

这部分工作就完成了<context:component-scan>的核心



这样 注解 + autowired注解+Reflections 就完成了自动IOC的框架 这个是大作业