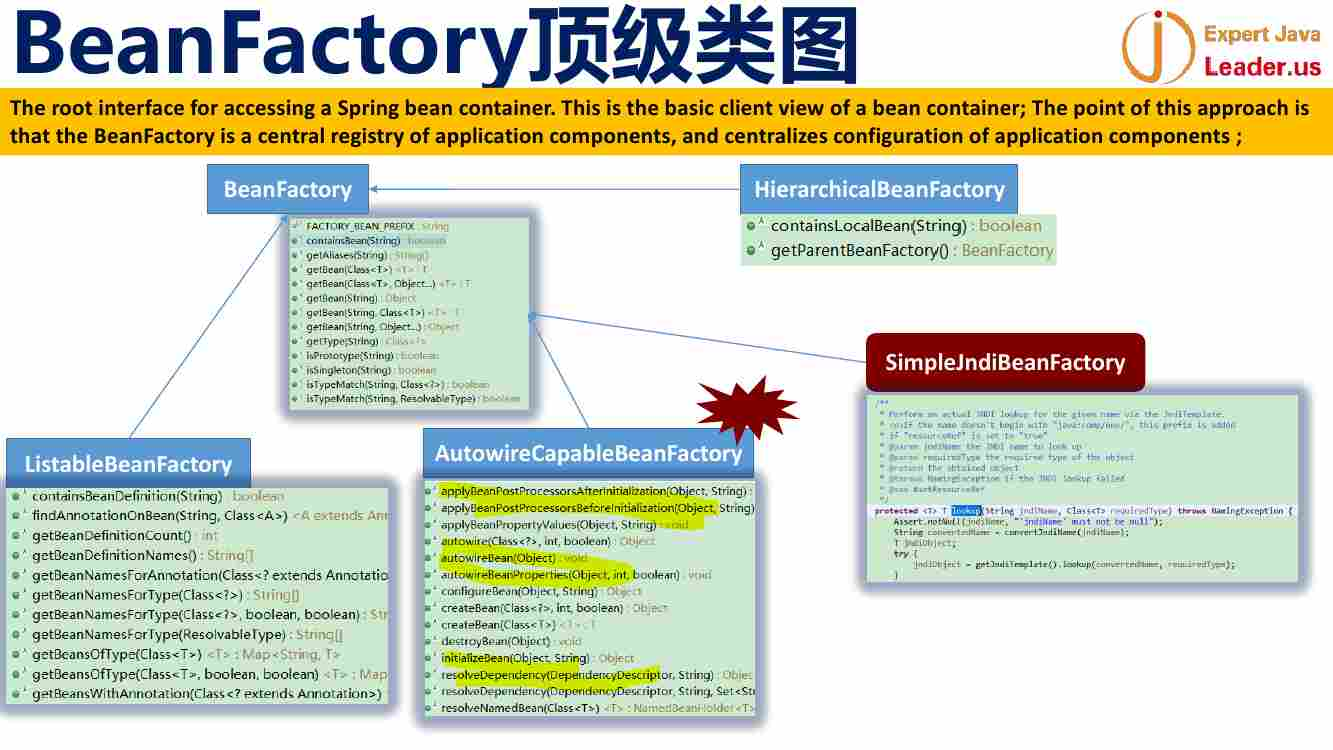


Spring核心比较多 但是 一节课不够 课下自己分析去



Spring最早是BeanFactory 来得到bean 顶级类图 是这样的 是接口

向下顶级的 ---- 第一个顶级的是 继承关系的BeanFactory ----- HierachicalBeanFactory 类似于java中的classloader 父类 子类 ------ java中很多成功的设计都是树状结构 这里面 HierachicalBeanFactory 就是 --- 也是 得到一个getParentBeanFactory() ---- 这个就是父的加载一部分 子的加载一部分 组成一个层级关系

还有一个LIstable ---为了查询bean ---- 可以containsbeanDefinitions ----- 为了做查询 查询是为了装配 并且找符合的bean 进行加工

---- BeanFactory最重要的一个接口

AutowireCapableBeanFactory 这个也非常重要 因为Spring就是最终能自动装配 --- 目标是装配bean 加工bean ---- 在进行bean的装配 复杂的操作 都在这里面完成

另外就是一个SimpleJndiBeanFactory---- 这个实在JDNI中配置的 ----- 这些Bean都来自于JNDI ----- 用的不多



看这个继承关系

SingletomBeanResgitry --- -是为了能够把一个单例的bean进行统一存放的接口

共享的bean 为了异地存储 ---- 相当于beanFactory

ConfigurableBF ---- 扩展了Hierachical 加了一些Bean的方法 可配置----- -Spring 这里面有addBeanPostProcessor ----- Springbean中最强大的地方还有就是可以添加BeanPostProcessor --- 这种扩展机制 可以在bean的装配过程中 人为进行干预

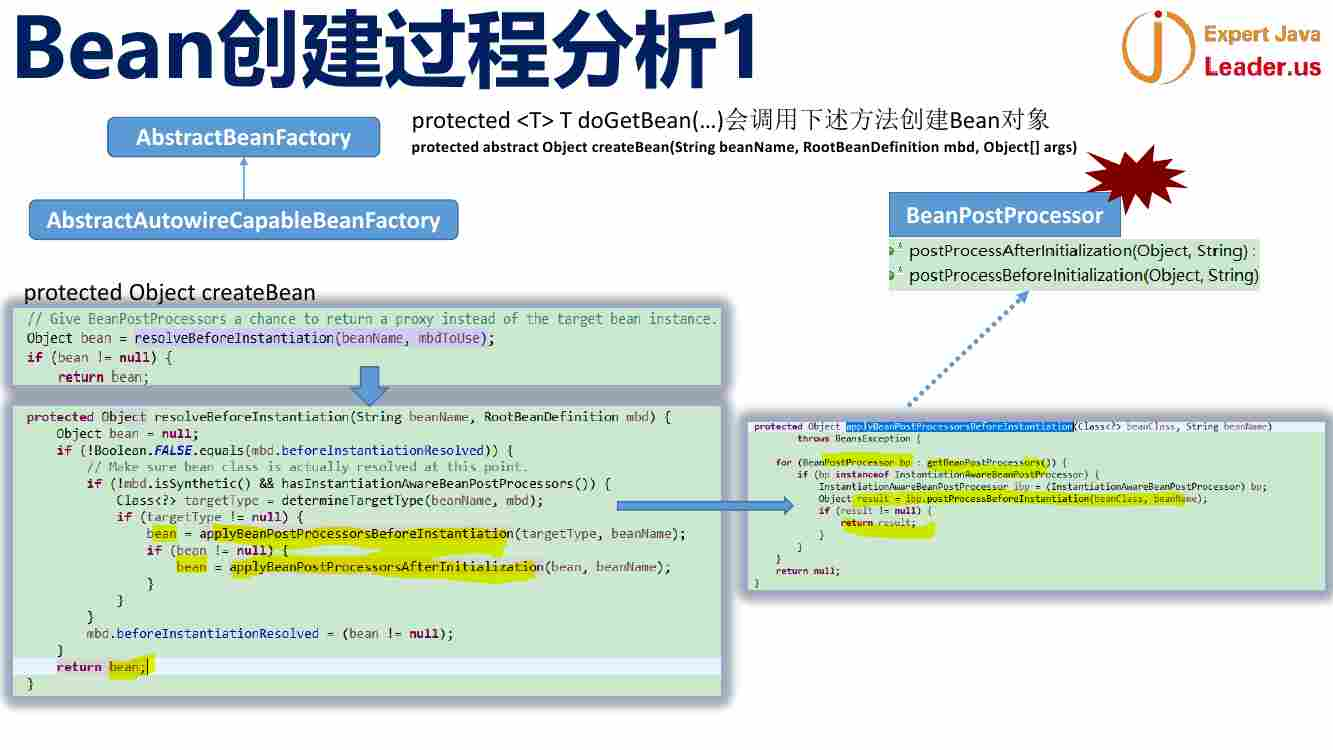
这里面还有addPropertyEditorResigtrar ---- 这个是类型转化

往下有一个AutowireCapableBF 和 AbstractBF

再往下 有一个非常基础的类就是AbstractAutoCapableBF ---- 主要的BF的一个实现 具备一个能够自动装配的BF 把所有的结构都实现了 是一个继承关系 是ConfigurableBF

上面的AbstractBF 采用了模板设计模式 把重要的流程设计好了 把具体的给子类做 这个addBeanProstProcessor等等都做好了

AACBF里面就是createbean 是真正创建bean的方法 创建之前 尝试看是否要真正创建一个bean 因为很多时候 并不是创建真正的bean 而是创建一个proxy 因此给Proxy一个机会 代理对象



刚才说了 AbstractBF是一个抽象模板的设计模式 对流程的控制 具体是子类实现

ABF里面 都有一个doGetBean ----- 里面不存在 就创建bean (doGetbean字面意思就是查询Bean ---- 这个方法在查询bean的时候 用的非常多) ------ 调用的时候 发现没有这个Bean 就创建这个bean ------ 里面调用抽象的createBean 这个是抽象方法

这个createbean 在AACBF中的实现 ---- 尝试让其他工具先去创建Bean ----- 调用了resolvebeaforeInstanLIstion ---- 下面这段代码

这里面重要的就是 bean = applyBeanPostProcessorBeforeInstantiation ---- 翻译过来就是在实例化（Bean）之前 对bean使用PostProcessor ---- 可以让PostBeanProcessor去创建这个bean【因为 你这里面 方法返回的是bean】

创建完这个bean之后 继续调用 bean = applyBeanPostProcessorAfterInitialization ---- 也就是在创建这个bean之后 也是允许BeanPostProcessor来修改bean的

所以 最后返回这个bean --- 所以 proxy也这么创建的

这个beanPostProcessor是一个接口 在bean初始化之前 和bean初始化之后分别调用 ----- 具体的beanpostprocessor 就实现一个 【重新截图一下】

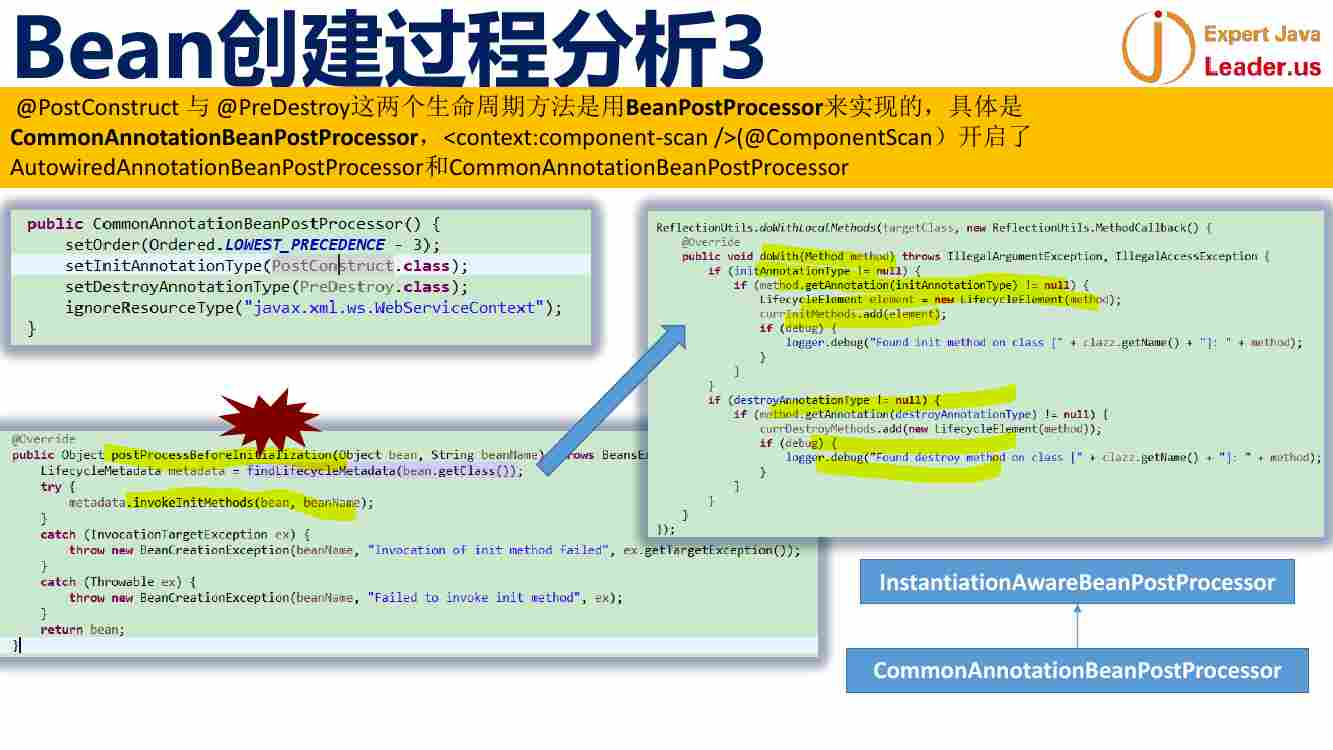
就是 获取了所有的BeanPostProcessor ---- 然后 是InstantiationAwareBeanPostPorcessor这种类型的 就使用InstantiationAwareBeanPostProcessor来修改bean ----- 因此 BeanPostProcessor的重要性 ---- 可能创建 替换我们创建的Bean



前面BF创建中 会调用InstantiationAwareBeanPostProcessor ---- 然后AutowiredAnnotationBeanPostProcessor刚好是这个InstantiationAwaurebeanPostProcessor刚好是这个的一个实现 ---- 就是对于注解这种注入方式 进行处理的 刚好是这个AutowiredAnnotationBeanPostProcessor ---- 这个BeanPostProcessor里面【重新截图】刚好是对于@Autowired 和@value这两个注解 --- 把这两个注解 关注起来 ---value这个注解也会由这个操作

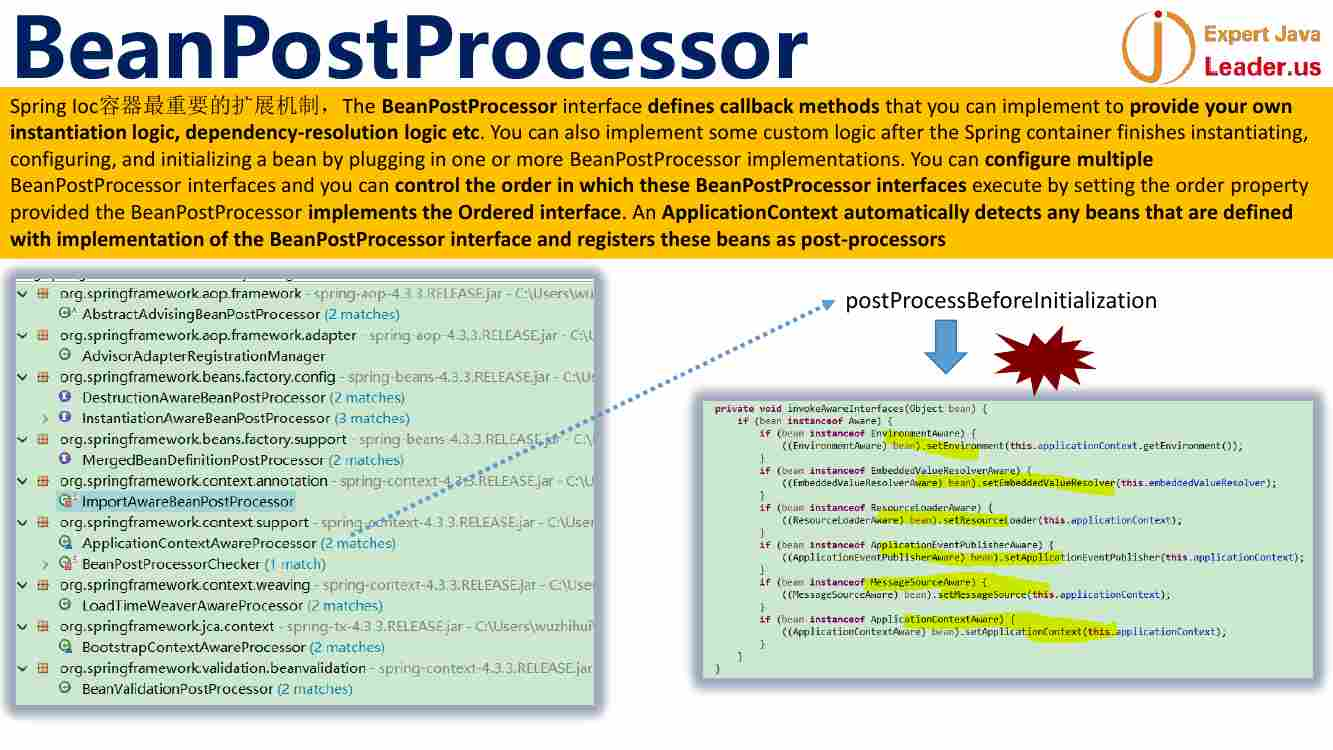
在AACBF中 有一个 调用populatebean方法 ---- 在代码中调用postProcessPV ----- 这里面 就是metadata.inject 进行注入 如果注解失败 报错

因此 Autowired 和Value这两个 都这个postProcessor处理 --- 进行自动装配



CommonAnnotationBeanPostPorcessor 这个是在Bean初始化之前调用的 这个对bean的lifecycle处理的 ------ 处理的是@PostConstruct和@PreDestroy

在这个PP中 setInitAnnotationType(PostConstruct.calss) 和 setDestroyAnnotationType(PreDestroy.class) 这个 ---- lifyCycle就是PostConstruct 和PreDestroy ------- 这个就在postProcessBeforeInitialization里面 调用了metadata.invokeInitMethods中调用 这个里面调用了init方法 ------- 下面是destroy方法进行处理 这里面用了Reflection进行代理 ------ 刚才了这两个Processor完成了自动装箱 和 生命周期的控制



Bean工厂创建的过程中 可以通过这个进行大量的控制 搜一下 很多地方都用到了 比如 aop里面 ---- 比如实现了spring aware接口 用到了这个pp 其他的 LocalTimeWeaver这个 加载之前 把这个class重新编制 也用到了 看一下ApplicationContext 这个里面 做了一个扩展 加载了所有的aware的接口 右边的图 其实 **BeanPP和aware接口 合在一起**  是Spring中的重要的扩展机制 ------- 可以实现不同aware接口 和pp交给spring进行触发



定义一些Aware接口 然后给这个bean出一套BeanPostProcessor 处理类 --- 这些类负责把拥有这些Aware接口进行特殊的处理 ----- 比如你定义一套LogWare接口 然后set一个自己控制的log 都可以

这里面好多Aware

ClassLoaderWare ---类加载的时候 可以获取CL

BFA 可以获取BF

ACA 可以获取ApplicationContext

环境变量 不用注解 也可以使用这个EnviromentAware

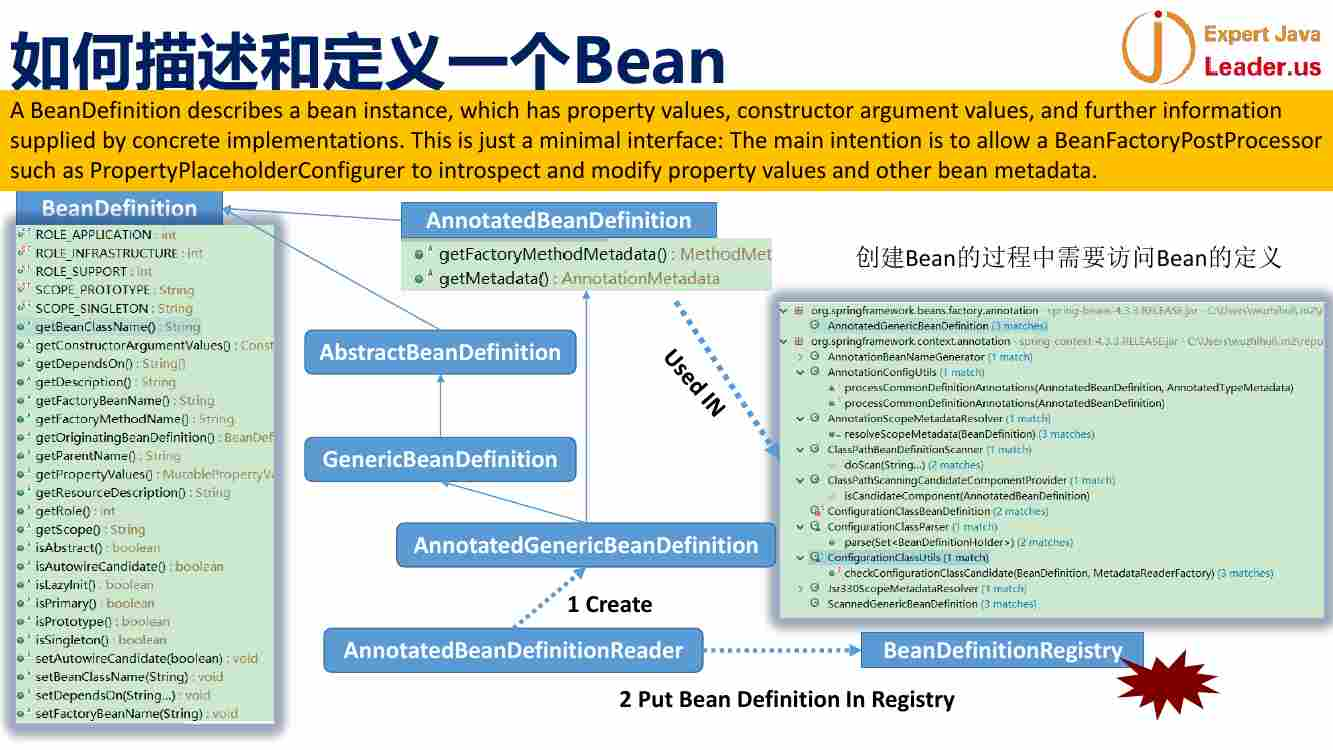
等等

Spring扩展机制 就会说 Aware接口+BPP

但是 Aware不建议使用 因为 有入侵性 ----- Aware不是为了程序而用的 而是为了 你的应用程序

所以 建议使用注解方式 而不是Aware接口

如果想扩展spring容器本身 可以使用Aware



**Bean的创建是依赖于一种描述的** 就是Bean的元数据 就是一种数据的元数据 比如 Class就是Object的元数据

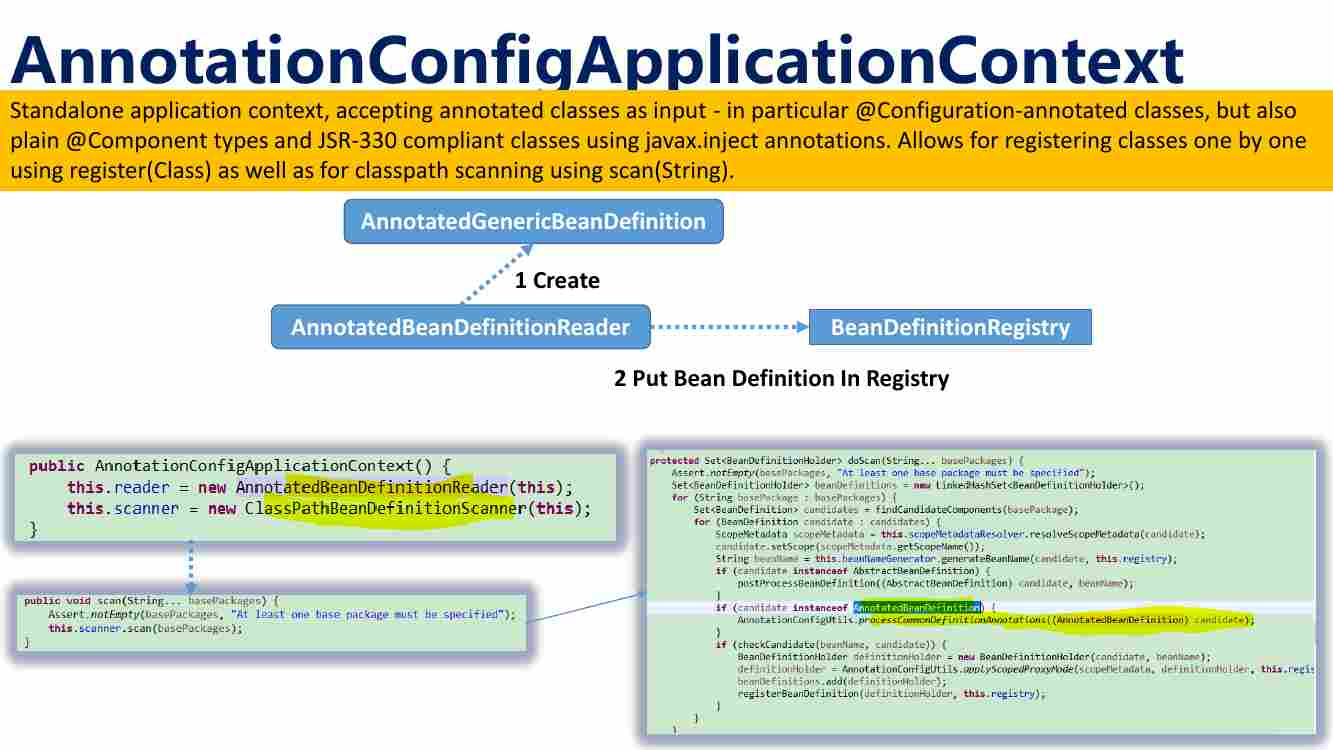
Bea的元数据 就是BeanDefinition ----- 我们的xml中对于Bean的描述 就是BeanDefinition ---后来使用了注解 做了扩展 使用的是AnnotationBD --- 注解的信息都扫描到AnnotatedBeanDefinition 【可以自己做程序 进行验证】

这里面比较复杂 有一个AbstractBDA ---- GenericDB ----AnnotatedBD 这种复杂的继承关系

右边是 创建bean的过程 是从BD来创建Bean的

问题就是 如何得到这些BD ---- 有一个Reader ------ Reader的Class负责对每一个要创建的bean 创建一个BD 创建过程 直接使用BD ----- Bean的定义 有一个BeanDRegistry ----- 是Reader把所有的Bean的定义放到这个注册中心去

下面是AnnotationConfiApplictionCOntext 是Ac基于注解模式的实现 ------ 我们注解模式 驱动一个AC 就是这个类 --------- 这个是一个单利的AC ---具备了classpath扫描功能 scan ---扫描所有的class信息 然后把Bean的定义都读到 然后再进行处理



构造方法有reader scanner两个成员 ------ Reader读一个带注解的bean 得到带注解的Bean的定义 BD ----- Scanner得到所有的要处理的Bean 两者配合 然后 把这个Bean的BD放到beanDefinitionRegistry中 ------ AnnotationGenericBD ---- 找到所有的Component

常用的AnnotationConfigApplicationContext 构造---- reader得到所有的bean ----- scanner获取到所有的注解信息 然后 reader就把这些注解信息变成BD 最后放到Registry中

这个doScan中 做了什么事情呢

If(candidate instanceof AnnotatedBeanDefinition){

AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition)candidate);

}

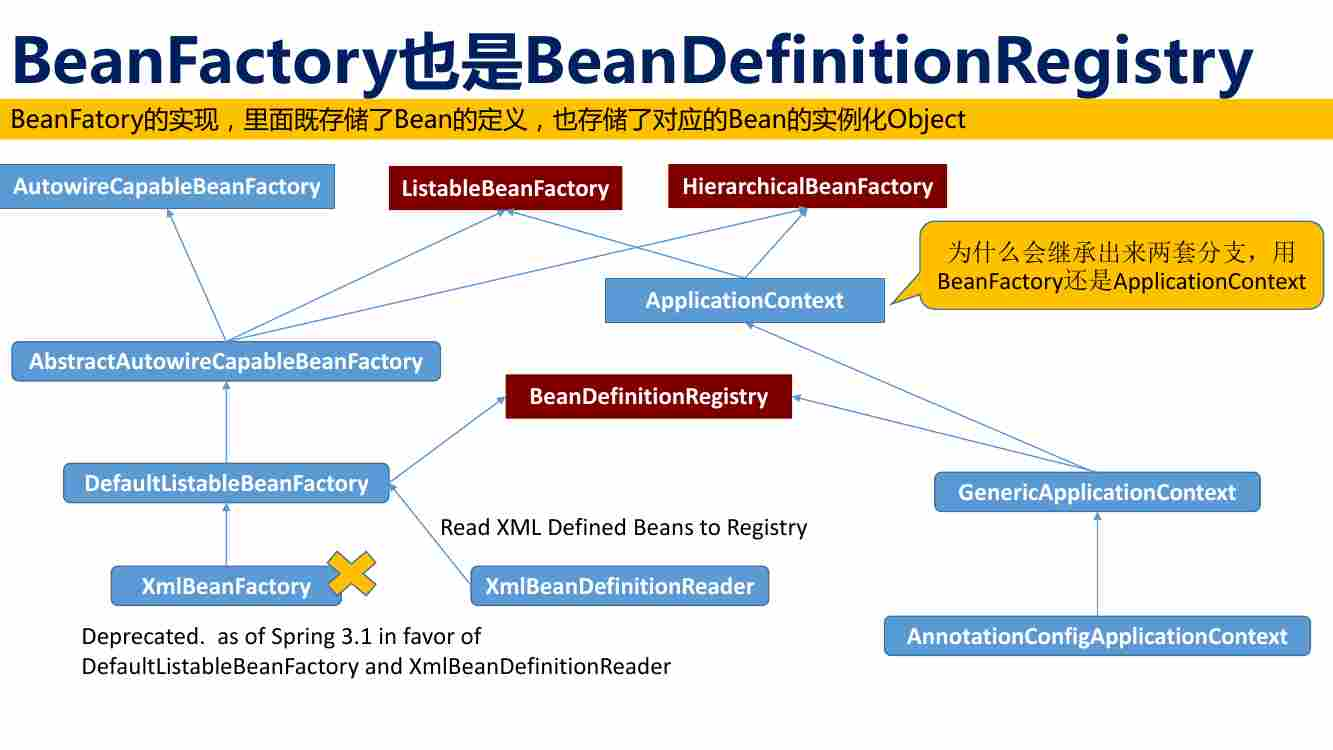
If(checkCandidate(beanName, candidate)){

beanDefininitions.add(defubutauibHolder);

registerBeanDefinition(definitionHolder, this.registry);

} 注册了Bean

处理注解定义 这个过程么有生成bean Bean的生成在其他的地方



正确的理解 就是BeanFactory也是BDRegistry【为啥？ 下面说了 BF的实现 里面既存储了Bean的定义BD 也存储了对应的Bean的实例化Object】 ----- 看这个关系

BF中有一个Listable和Capable 这两个重要 ------ 这个图中 ListableBF和可继承的HierachicalBF （最上面红色标记的 ---- 集成关系中 上面的 也就是箭头指向的 就是傅雷 所以 这两个是重要的父类）

这两个父类下面 右边的分支就是ApplicationContext 是我们经常用到的

AC往下走 就是GenericApplictionContext和AnnotationConfigApplicationContext 这里用到了BeanDefinitionRegistry 实际上 GenericAC也是BeanDefiniationRegistry的一个接口实现【这个不就证明了 BF也是BDResgitry么？ 可以自己证明看】

BF 这边 往下是AACBF 往下是DefaultListableBF 这里面也用到了DBR 但是 实际上 是BDR这个接口的实现

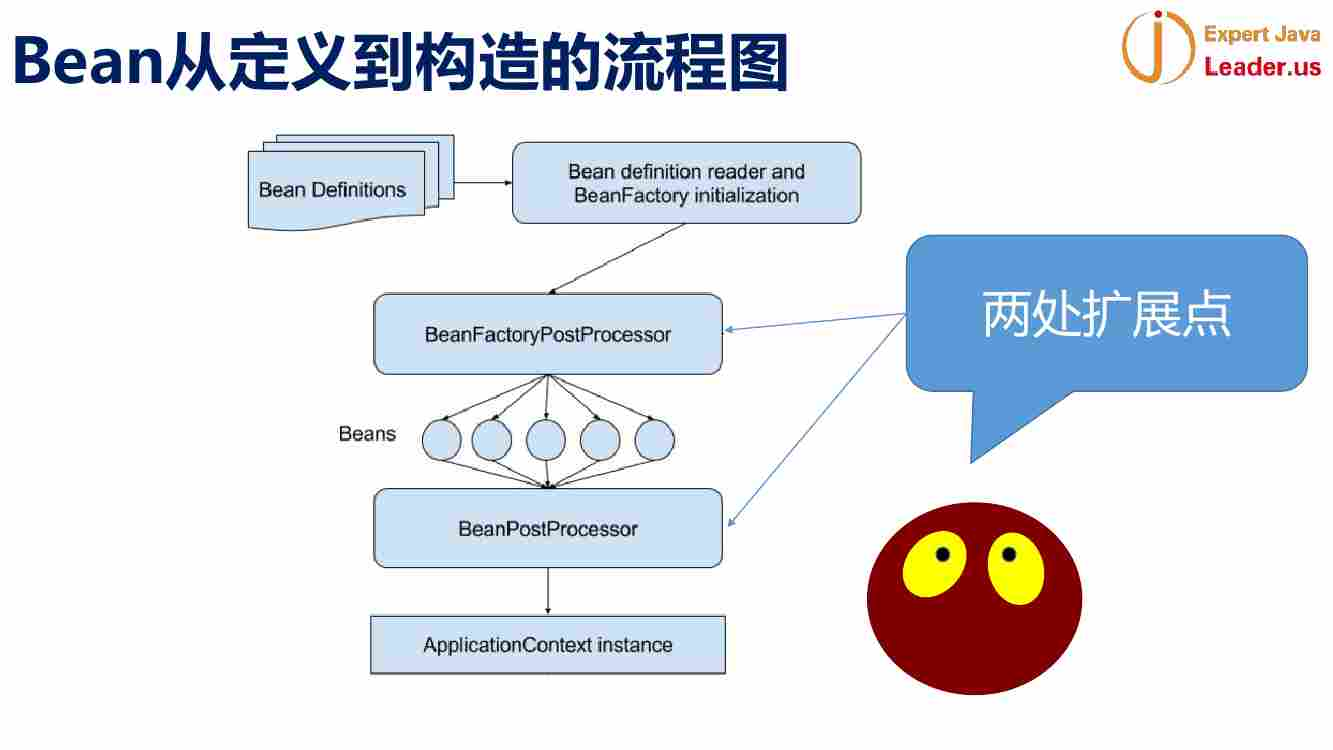
真正的BF 和AC 同是都是DBR 就是BD的注册表 ----- 所以 这两个里面 放了Bean本身 和 Bean本身的定义

BF继续向下 就是XmlBeanFactory =----- 从XML中创建Bean的工厂类 但是 后来发现这个XmlBeanFactory没什么用 就单独搞出来一个XmlBeanDefinitionReader 这个XmlBeanDefinitionReader 就可以把XML中的内容读出BD 然后放到Registry中 而不需要单独构建一个xmlbeanfactory了 这样就足够了 ------ 所以 后来XmlBeanFactory就被取消了【源码中去观察】

所以 最后 这个 XmlBeanDefinitionReader就直接继承了DefaultListtableBeanFactory ----- 然后这个DefaultListtableBF还是DBRegistry的实现类 这样 就是XMLBeanDefinitionReader获取到BD之后 就直接放到DefaultListableBF中去了！！！ 所以 不用单独的XmlBeanFactory 用DefaultListableBF代替了XmlBF 【前面估计也是 有一个Annotation的Reader 然后 把Annotation标注的类的BD 都放到DefaultListableBF中 也没有单独的AnnotationBF。。。】

Spring复杂在 发展过程 修订了很多 设计

真正创建的Bean的时候 需要注册表



Bean Definition可能存储在xml中 可能被注解标注出来 也可能存储在class文件中 ---- 都通过BD Reader进行读取 读取完放到Bean Definition Registry中 因为这个Bean Factory前面说过了 实际上也是 BeanDefinitionRegistry的接口实现 前面的结论就是 BF 也是BDRegistry ------- 所以 把BeanDefinition放到BF中 就相当于放到了注册表中

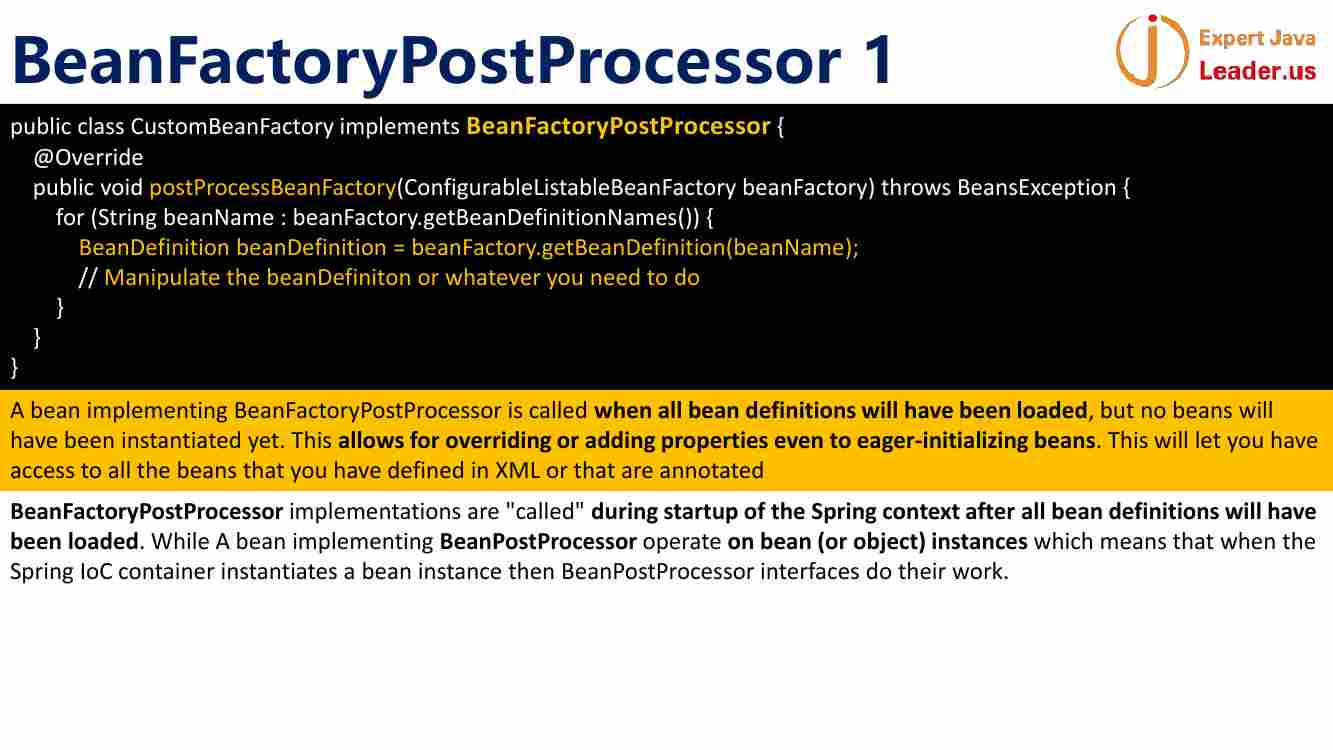
创建一个bean 的时候 首先是BeanfatoryhPP ---- 这个类的名字误导人 ---- 实际上很简单 就是提供了修改BD的机会 就是 原来BD定义的是ClassA 但是通过这个BeanFactoryPP可以把ClassA 修改成ClassB

可以把BeanDefinition 通过这个BFPP进行修改

产生Bean的过程中 可以通过BeanPostProcessor进行加工

【这张图需要重新画　　一个 是 BeanFatoryPostProcessor和BeanPostProcessor ----- 一个是处理BF的 另一个是处理bean的 但是 实际上 因为BeanFactory本身还是BeanDefinitionRegistry 所以 两个PostProcessor 严格说 一个是改变BD的 一个改变bean的】

最后 构建Application Context



这里面给的是ConfigurableBeanfactory

这里面可以修改bean的定义

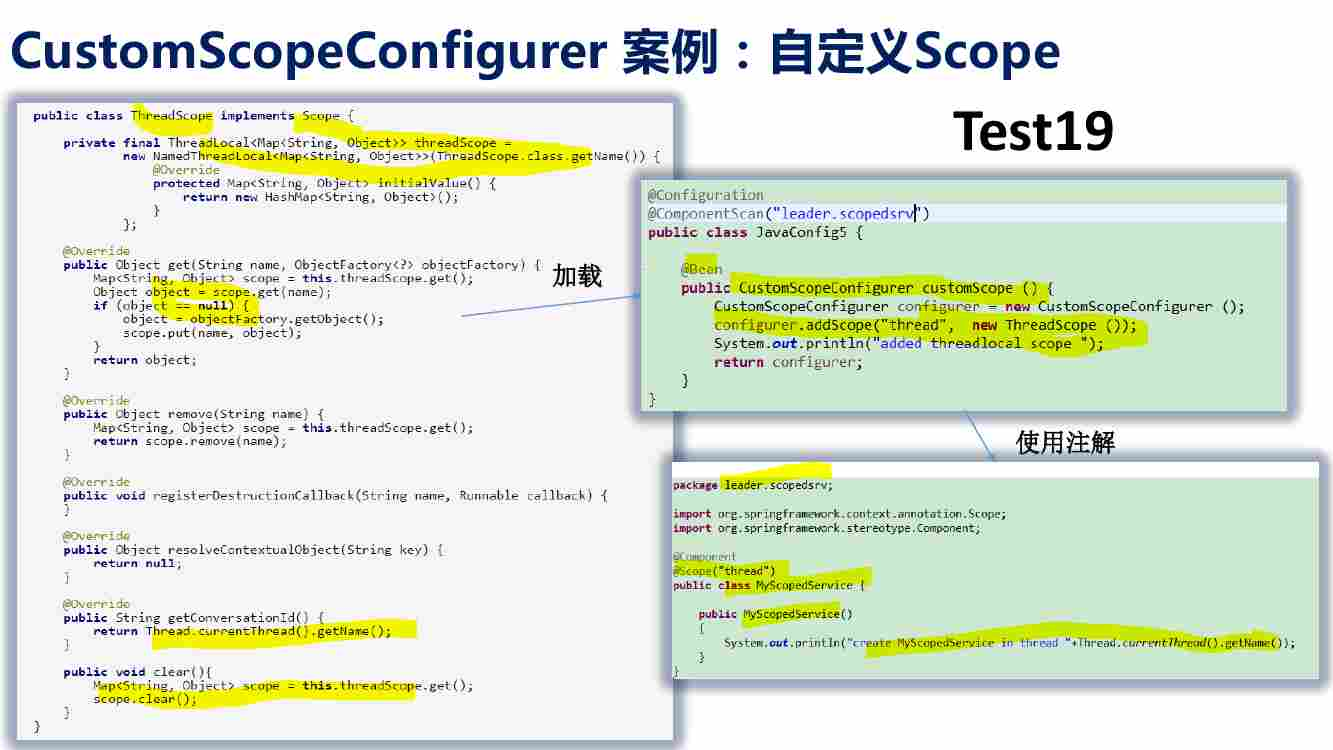
前面读到Registry之后 可以在这修改BD　这时候　BD还没有创建

在哪用到的　下面：



AspectJWeaving也用到了

这里面 有一个CustomScopeConfiguer 也是Bean定义的时候 加载的 我们利用这个 做一个有价值的练习



Scope 定义一个ThreadLocal中的bean ---- ThreadLocal中 一个线程过来的时候 会创建这个Bean 然后 另外一个线程过来的时候 会重新创建这个bean

也就是不同的线程 访问到这个bean的时候 可以拥有自己的 这个在Spring中是没有提供的 我们可以模仿【可以写作业】

先实现一个ThreadScope 有一个thradScope 这里面是ThreadLocal <Map>

【需要解释Scope】

每次都从Thread中获取的

GetConversionId这里面 给出来这个名字

Scope不负责对象的创建 BF去创建 但是 什么时候创建Scope去创建的

Test19这个代码里面 构造一个Bean ----然后 搞一个JavaConfig5

最后的关键代码在这里

这里面 @Scope(“thread”) 怎么和之前的对应呢？

在JavaConfig5这个类中

Configurer.addScope(“thread”, new ThreadScope()); 【//Spring 本身 本身的配置 不就是xml中的对应关系么！！！所以 这里面 就是 “thread”, new ThreadScope() ---- xml配置 就是 XML Configuration ----所以 类名 就i是 Conigurer.addScopexxxx 】

把这个ScopeCOnfigure开启 加入自己的Scope的实现

这样 后面的

@Component

@Scope(“thread”)

Public class MyScopeService{

Public MyScopeService(){

System.out.println(“create MyScopedService in thread ”+Thread.currentThread.getName());

}

}

创建这个类的时候 就会调用这个Scope 【所以 要看Scope在创建中 起到了什么样的作用】

不同线程创建 就是不同的结果

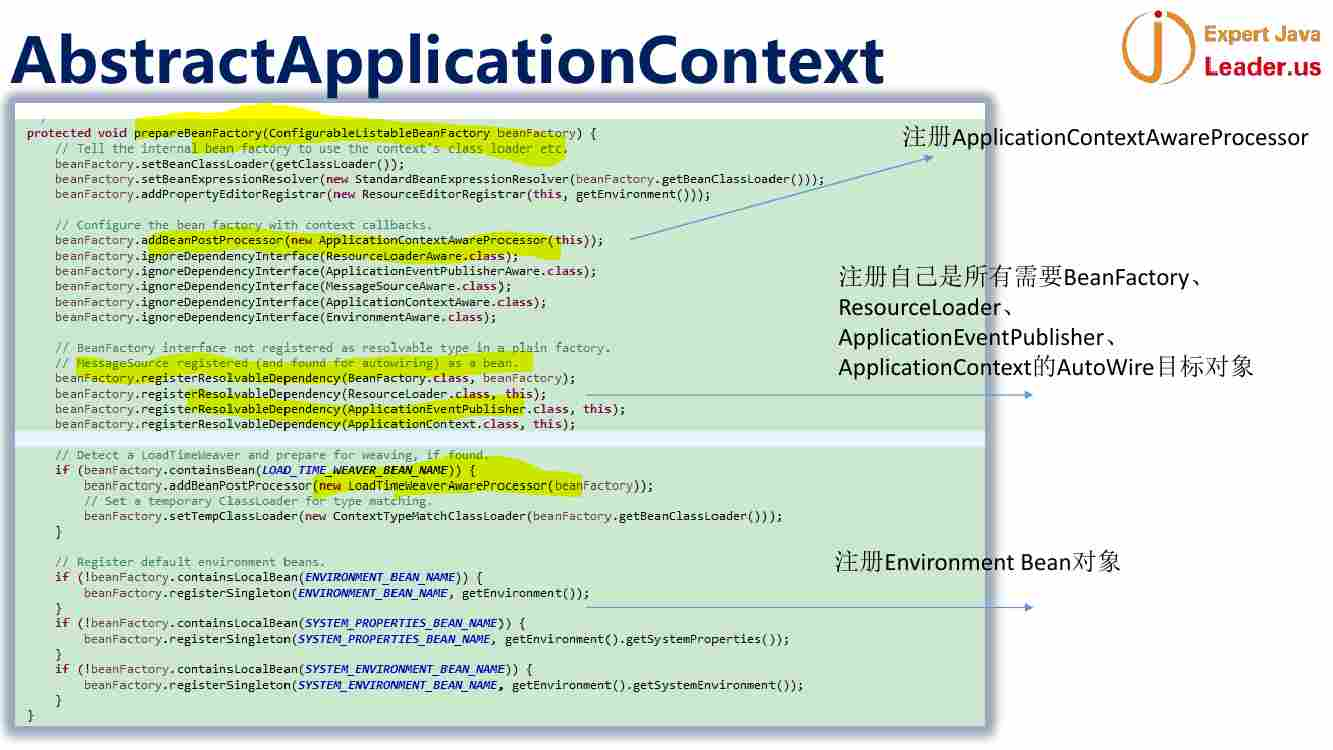


ApplicationContext 除了BF 还有AC 这个PPT讲的区别

BF仅仅是初始化 装箱

AC 是一个加强版的BF 除了做BF的工作 还有做了红色标注的工作

为了程序开发方便做的加强版 这里面 加入了BPP BFPP 和 MesageSource 和 ApplciationEvent Publisher



Prepare的时候 穿了ConfiguarbleListableBF ---- 所以 更体现了AC是BF的加强版 有参数

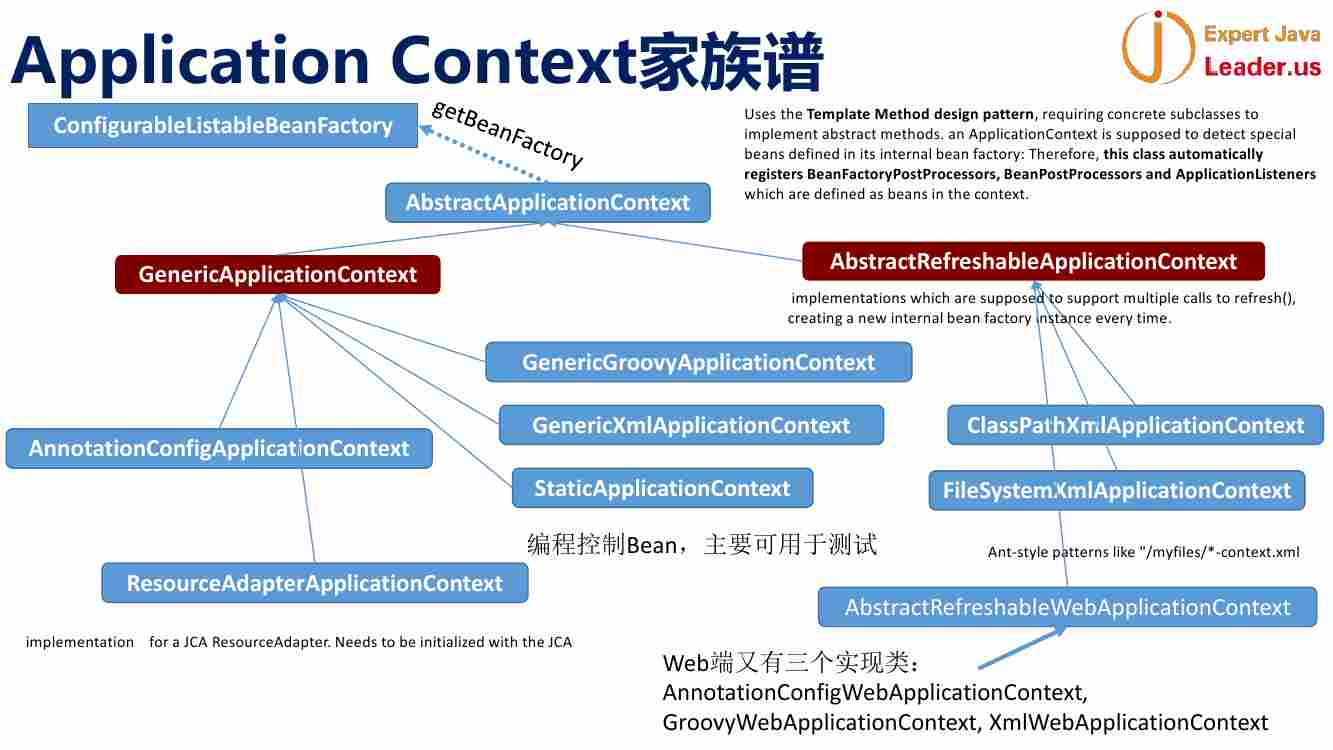
这里面 有上下文 就把Processor注册进去了

也就是说 自己是BF 自己是Pulisher

所以 一个bean需要注入这些的时候 AC就被提供出去了

这里面AspectJ自动加载 也有

AC在BF基础上 把各种常用的功能 集成到AC 是BF的加强版



这个家族也是非常强大的 这里面 AbstractApplicationContext 是虚线 也就是 依赖了ConfiguarableListableneanFactory 前面不就是 参数是这个BF么 就是uml中的依赖 这个AAC 实现 是依赖于BF的体系

往下走 做了几个分支

AAC用了经典的模板设计方式 父类定义了流程 子类实现必要的接口

向下 连个大分支

这个GenericAC和 ARAC

GAC 下面有一个有一个AnnotationConfigAC --- 前面介绍了 ---- 注解读

ResourceAAC用的很少 ----- 资源文件里面读

StaticAC 用于测试 很少用 ---- 程序编写好的 编程控制好的

XML方式用的多 ---- xml中读

Groovy用的多 ----groovy中读

区别就在于 读BD用什么读

ARAC 这个分支 ----- 就是 给J2EE程序用的 也就是J2Se程序整个运行过程是不可能refresh的 但是 webapp中有reload

Reload就是bean可以重新刷新 可以热加载

所以refresh 就可以重新加载 这个就是为了j2ee用的 web用的

往下是Classpath FSAC 还有 web做了一个抽象的AbstractWAC ---

Web端有三个 AnnotationWAC GroovyWAC 和 XMLWAC 这个 和左边的J2se一样的 三个扩展

【GenericAC这个分支 是给J2se用的 右边的 ARefreshableAC是给WEB用的】

【在Spring中 Abstract出现了类 基本上都是模板设计模式 这个要在代码中证明】

所以 Spring弄得特别复杂

【可以看对应的AC 比如 AnnotationConfigAC和AnnotationConfigWebAC怎么用 具体区别】

【把上节课的总结看看 做回顾】