

Spring

Bean

■ *bean*是单例的

singleton

■ *bean*不是线程安全的

并没有可变的状态 （不能修改的类）

某种程度上线程安全

但是可变的成员变量要考虑线程安全

■ 生命周期

把xml信息传递到`beandefinition`

构造函数 实例化`bean`

——》依赖注入

--》Aware接口 -----(`beannamaware`,`beanfactoryaware`,`applicationaware`) 三个要记住

--->`BeanPostPrecessor` #before

--->初始化方法---(`initializingbean` , 自定义化方法)

--->`BeanPostProcessro`#after (---->AOP-->动态代理) 这里可以功能增强

--->销毁`bean`

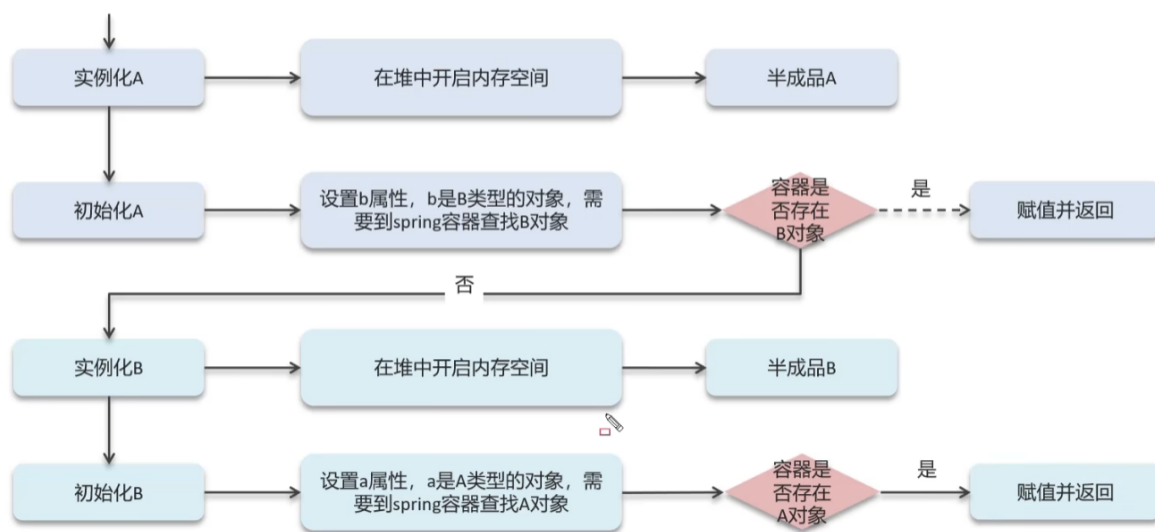
循环引用

A用到B B用到A

可能会死循环

初始化的循环依赖

什么是Spring的循环依赖?



三级缓存解决循环依赖 (都是concurrenthashmap, 三级是普通hashmap)

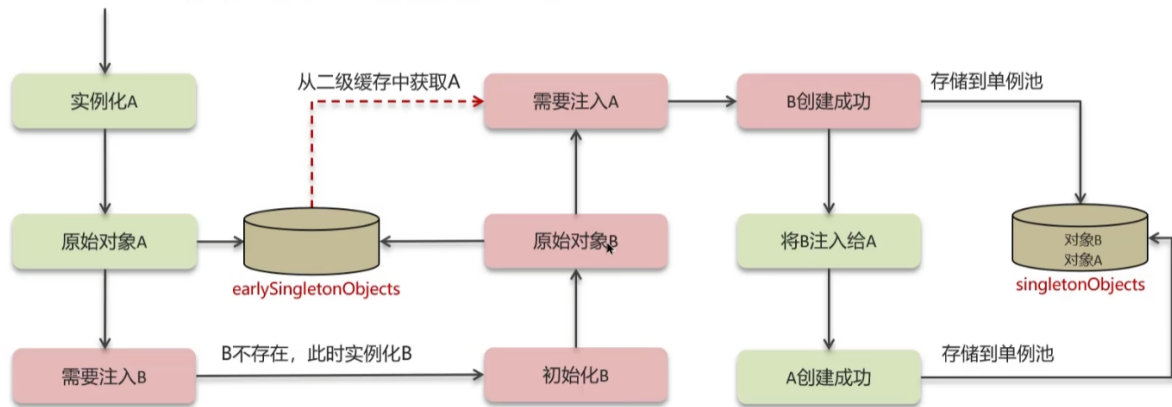
一级缓存 singletonObjects 单例池, 缓存已经经历了完整的生命周期, 已经初始化完成的bean对象 (为了实现单例模式)

二级缓存 earlySingletonObjects 缓存早期的bean对象 (生命周期还没走完) (半成品)

三级缓存 singletonFactories 缓存的是ObjectFactory, 表示对象工厂, 用来创建某个对象的

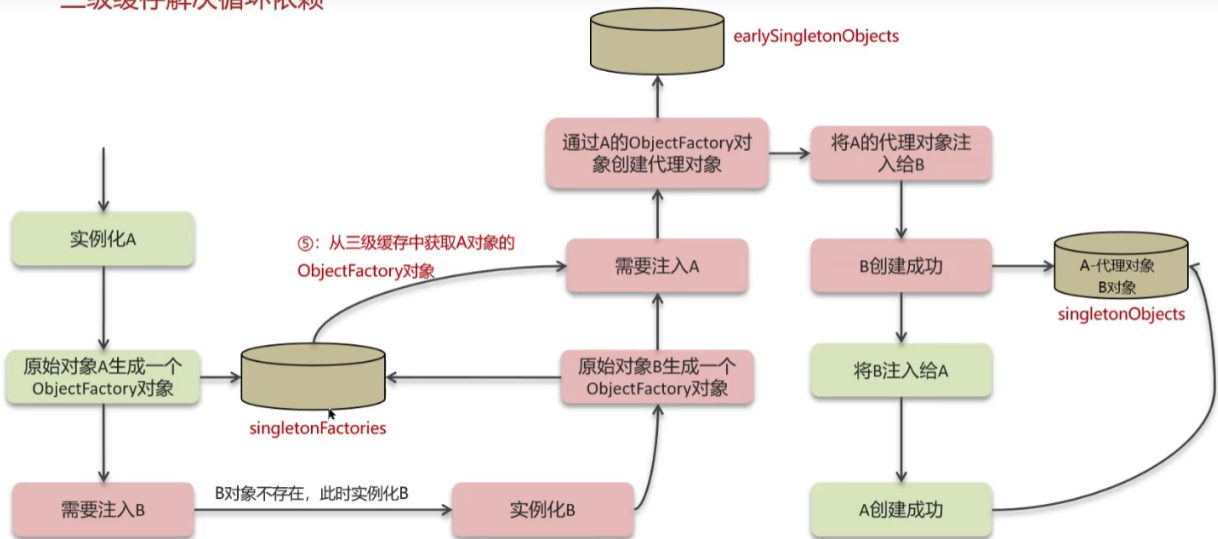
二级缓存解决普通循环问题

如果想要打破循环依赖, 就需要一个中间人的参与, 这个中间人就是二级缓存。



如果 A是代理对象 那就需要三级缓存解决

三级缓存解决循环依赖



对比二级缓存解决 其实就是存了个单例的工厂 然后用工厂拿到A 再把A放到早期单例池

例如事务就要代理对象 @transaction

如果构造方法出了循环依赖 那就用懒加载 @Lazy

AOP

切面

基于动态代理

@Around 是完全控制（代理）

，通常需要先使用 @Pointcut 注解来声明一个切点函数，然后再通过其他通知注解（如 @Before、@After、@Around 等）来引用这个切点函数。

```
@Pointcut("execution(* com.example.service.UserService.*(..))")
public void logPointcut() {}

@Before("logPointcut()") // 引用刚才定义的切点函数
```

事务的本质是AOP 开启事务再执行 失败就回滚

事务失效的场景

@Transactional

情况1：异常捕获处理

让try catch给捕获 但是没有throw 导致spring没处理和回滚

情况2：抛出检查异常

在方法上定义了throws IO异常

Spring默认回滚非检查异常(Runtime)

设置@Transactional(rollback=Exception.class) 直接顶格 处理所有异常（实现exception的异常）

情况三：非public方法导致的失效

spring默认处理public 与代理有关

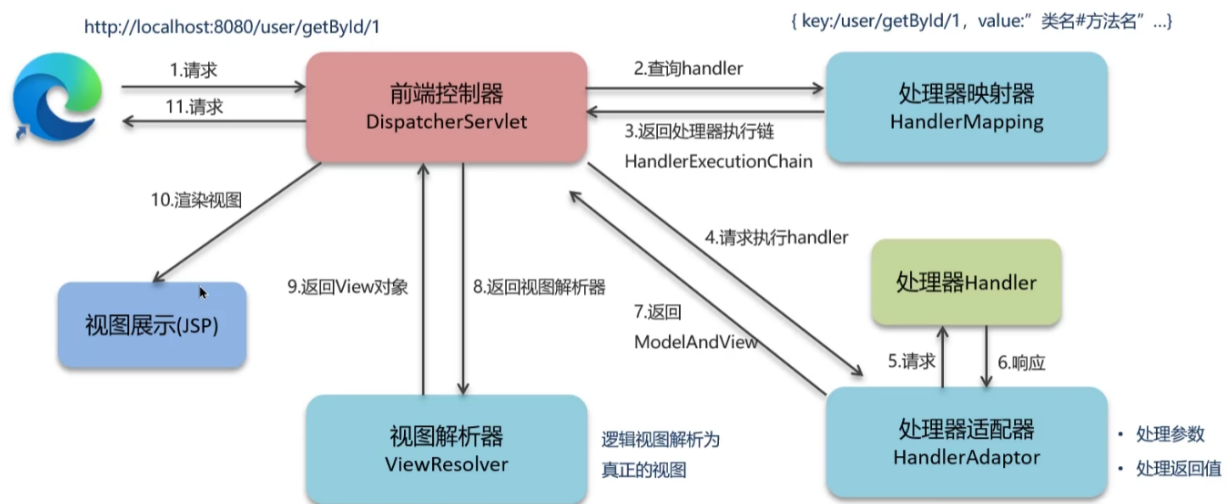
java默认是default 方法具有包私有访问权限，只有同一包中的类能够访问 不能被包外的类访问。

SpringMVC

执行流程

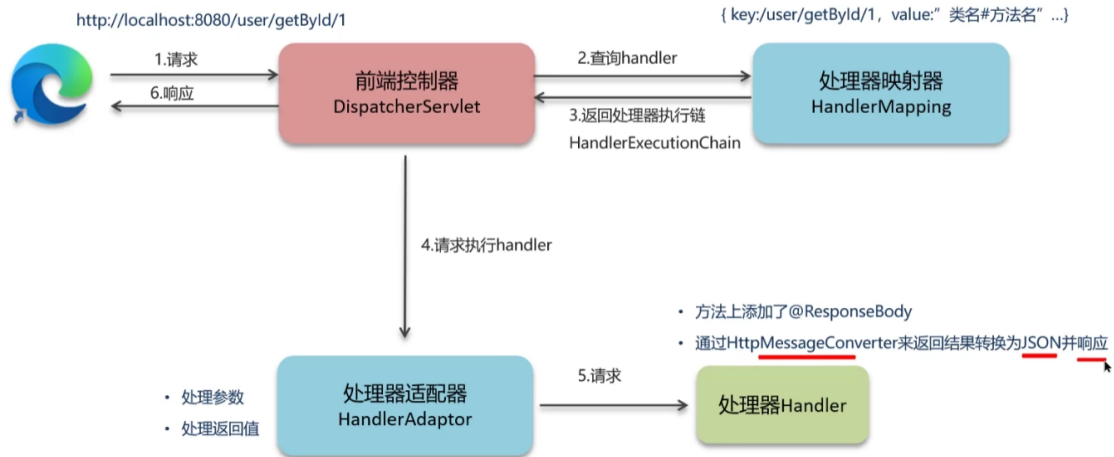
视图阶段

视图阶段 (JSP)



这一部分看PDF

前后端分离阶段（接口开发，异步请求）



其实就是把后面的视图省掉了

自动配置原理

@SpringBootApplication包含以下三个注解

@SpringBootConfiguration

@ComponentScane

@EnableAutoConfiguration

它实际上是 `@Import(AutoConfigurationImportSelector.class)` 的一种简化写法，意味着它导入了 `AutoConfigurationImportSelector` 类，这个类负责从所有自动配置类中挑选出与当前应用环境匹配的配置。

读取引用的jar包,component和查meta-inf/spring.factories

@ConditionalOnClass

判断是否有对应的类，如果有则加载 把配置类的所有bean放入容器

Spinrg三件套常见注解

Spring

自己背吧

注解	说明
<u>@Component</u> 、 <u>@Controller</u> 、 <u>@Service</u> 、 <u>@Repository</u>	使用在类上用于实例化Bean
<u>@Autowired</u>	使用在字段上用于根据类型依赖注入
<u>@Qualifier</u>	结合@Autowired一起使用用于根据名称进行依赖注入
@Scope	标注Bean的作用范围
@Configuration	指定当前类是一个 Spring 配置类，当创建容器时会从该类上加载注解
@ComponentScan	用于指定 Spring 在初始化容器时要扫描的包
@Bean	使用在方法上，标注将该方法的返回值存储到Spring容器中
@Import	使用@Import导入的类会被Spring加载到IOC容器中
@Aspect、@Before、@After、@Around、@Pointcut	用于切面编程（AOP）

SpringMVC

注解	说明
@RequestMapping	用于映射请求路径，可以定义在类上和方法上。用于类上，则表示类中的所有的方法都是以该地址作为父路
@RequestBody	注解实现接收http请求的json数据，将json转换为java对象
@RequestParam	指定请求参数的名称
@PathVariable	从请求路径下中获取请求参数(/user/{id})，传递给方法的形式参数
@ResponseBody	注解实现将controller方法返回对象转化为json对象响应给客户端
@RequestHeader	获取指定的请求头数据
@RestController	@Controller + @ResponseBody

Springboot

注解	说明
<u>@SpringBootConfiguration</u>	组合了- <u>@Configuration</u> 注解，实现配置文件的功能
<u>@EnableAutoConfiguration</u>	打开自动配置的功能，也可以关闭某个自动配置的选
<u>@ComponentScan</u>	Spring组件扫描

Mybatis

■ 执行流程

mybatis-config.xml

配置

指定mapper

有springboot就不用这些了

构建会话工厂---》创建会话SqlSession 项目与sql的会话----》执行器(里面有映射信息)--
-》MappedStatement对象 ----->数据库

■ 延迟加载

默认不开启

什么是延迟加载？

简单讲就是按需加载 fetchtype可以设置为lazy （局部修改）

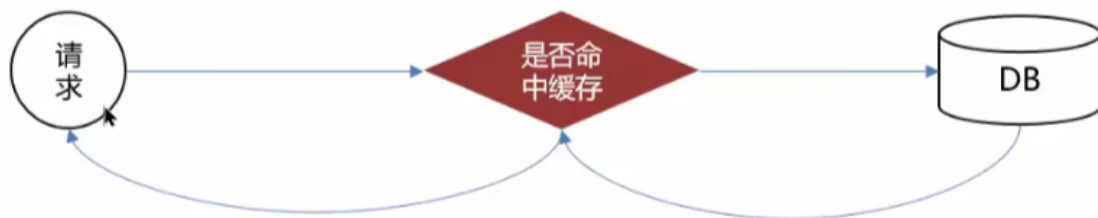
全局修改:lazyloadingenabled

用的cglib代理对象

延迟加载的底层原理知道吗？

1. 使用CGLIB创建目标对象的代理对象
2. 当调用目标方法时，进入拦截器invoke方法，发现目标方法是null值，执行sql查询
3. 获取数据以后，调用set方法设置属性值，再继续查询目标方法，就有值了

缓存



- 本地缓存，基于PerpetualCache，本质是一个HashMap
- 一级缓存：作用域是session级别
- 二级缓存：作用域是namespace和mapper的作用域，不依赖于session

一级缓存：单一sqlsession

二级缓存：不同会话之间的

二级缓存默认关闭 可以开启 要记得打上标签

一级缓存和二级缓存namespaces增删改后 二级缓存select中的缓存会被清空

只有会话提交或者关闭以后 一级缓存中的数据才会存到二级缓存

Spring代理相关

Spring 默认优先使用 **JDK** 动态代理，但如果目标类没有实现接口，Spring 只能使用 **CGLIB**。其核心原因在于 **JDK** 动态代理的限制 和 **CGLIB** 的继承机制。

CGLIB 代理的本质是 继承目标类并重写方法，但 **final** 方法不能被重写，所以当 CGLIB 试图对 **final** 方法进行代理增强时，可能会导致 代理对象内部依赖的 **Spring** 组件未被正确初始化，从而引发 空指针异常（**NullPointerException, NPE**）。

CGLIB 代理是通过 创建目标类的子类 来实现的，但 **final** 方法不能被子类重写，所以当外部调用该 **final** 方法时，实际上调用的是 目标类（父类）自身的方法，而代理对象的依赖注入（**DI**）在 **Spring** 中是针对代理类进行的，目标类本身并没有被 **Spring** 进行 **DI**，因此会导致 **null** 引用，从而引发 空指针异常。