课程目标

1、了解什么是一级、二级、三级缓存，为什么设计三级缓存?

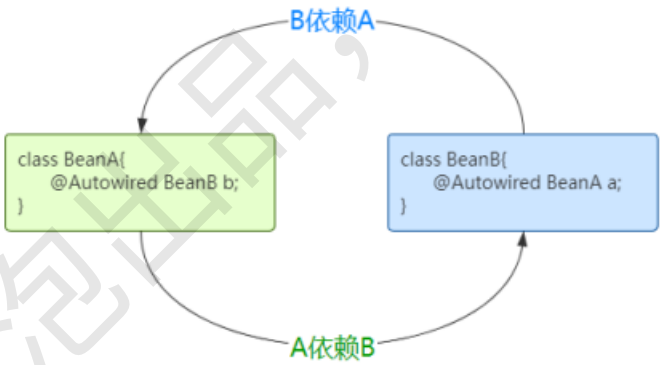
2、掌握 Spring 解决循环依赖问题的原理。

内容定位

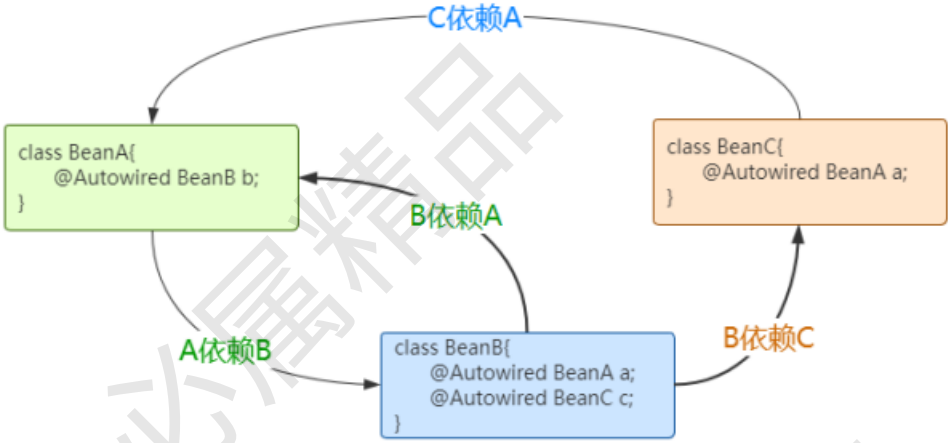
在完全掌握 Spring IoC 原理的基础上，理解 Spring 内部对依赖注入的处理原理。

什么是循环依赖?

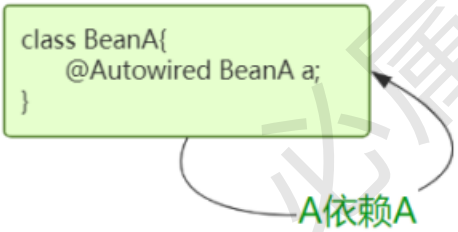
如下图所示:



BeanA 类依赖了 BeanB 类，同时 BeanB 类又依赖了 BeanA 类。这种依赖关系形成了一个闭环，我们把这种依赖关系就称之为循环依赖。同理，再如下图的情况:



上图中，BeanA 类依赖了 BeanB 类，BeanB 类依赖了 BeanC 类，BeanC 类依赖了 BeanA 类，如此，也形成了一个依赖闭环。再比如：



上图中，自己引用了自己，自己和自己形成了依赖关系。同样也是一个依赖闭环。那么，如果出现此类循环依赖的情况，会出现什么问题呢?

循环依赖问题复现

定义依赖关系

我们继续扩展前面章节的内容，给 ModifyService 增加一个属性，代码如下：

```
@GPService
public class ModifyService implements IModifyService {

    @GPAutowired private QueryService queryService;

    ...

}
```

给 QueryService 增加一个属性，代码如下：

```
@GPService
@Slf4j
public class QueryService implements IQueryService {
```

```
    @GPAutowired private ModifyService modifyService;

    ...

}
```

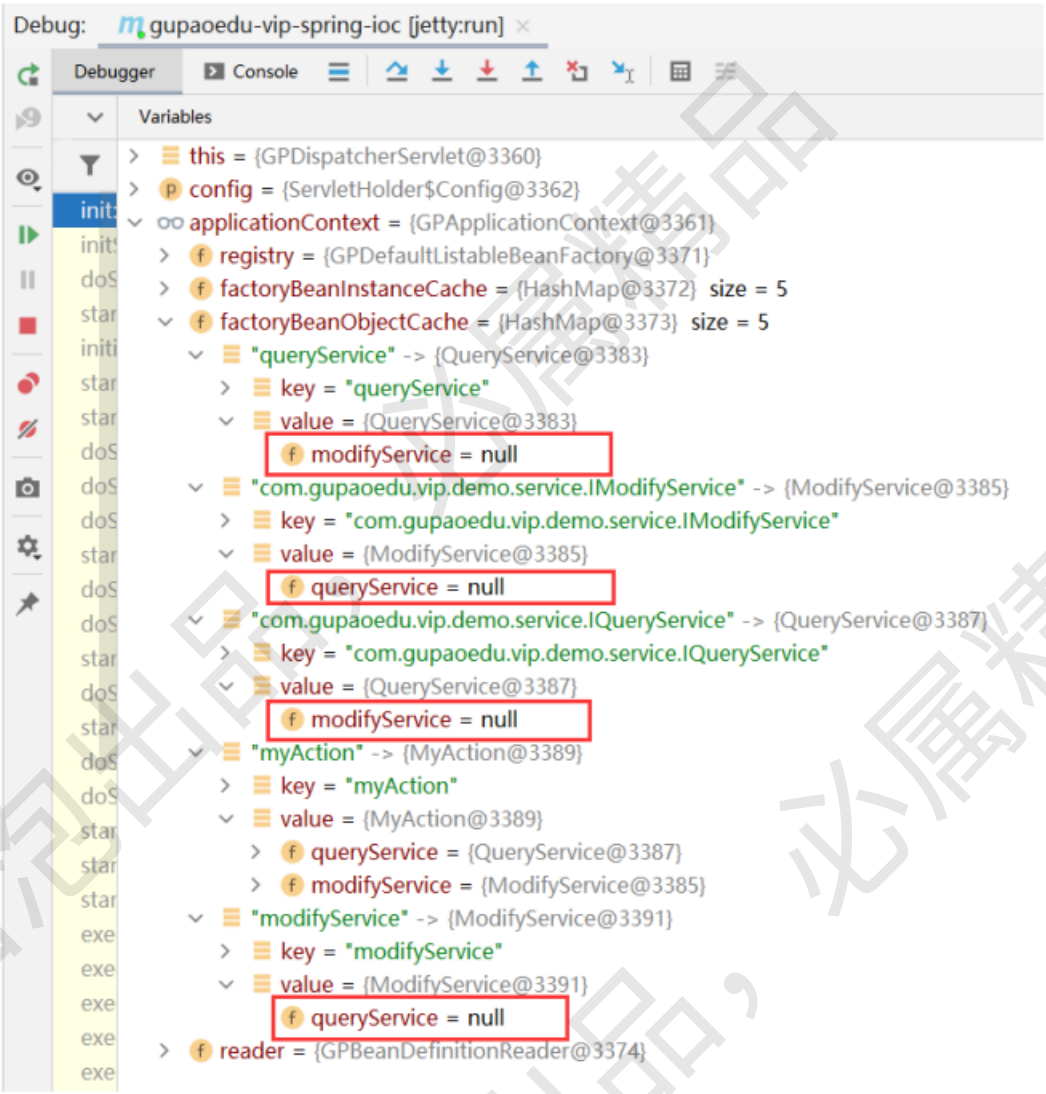
如此，ModifyService 依赖了 QueryService，同时 QueryService 也依赖了 ModifyService，形成了依赖闭环。那么这种情况下会出现什么问题呢?

问题复现

我们来运行调试一下之前的代码，在 GPApplicationContext 初始化后打上断点，我们来跟踪一下 IoC 容器里面的情况，如下图：

```
@Override
public void init(ServletConfig config) throws ServletException {

    applicationContext = new GPApplicationContext(config.getInitParameter( s: "contextConfigLocation"));

    //========== MVC功能 ==========
    //5、初始化HandlerMapping
    doInitHandlerMapping();

    System.out.println("GP Spring framework is init.");

}
```

启动项目，我们发现只要是有循环依赖关系的属性并没有自动赋值，而没有循环依赖关系的属性均有自动赋值，如下图所示：

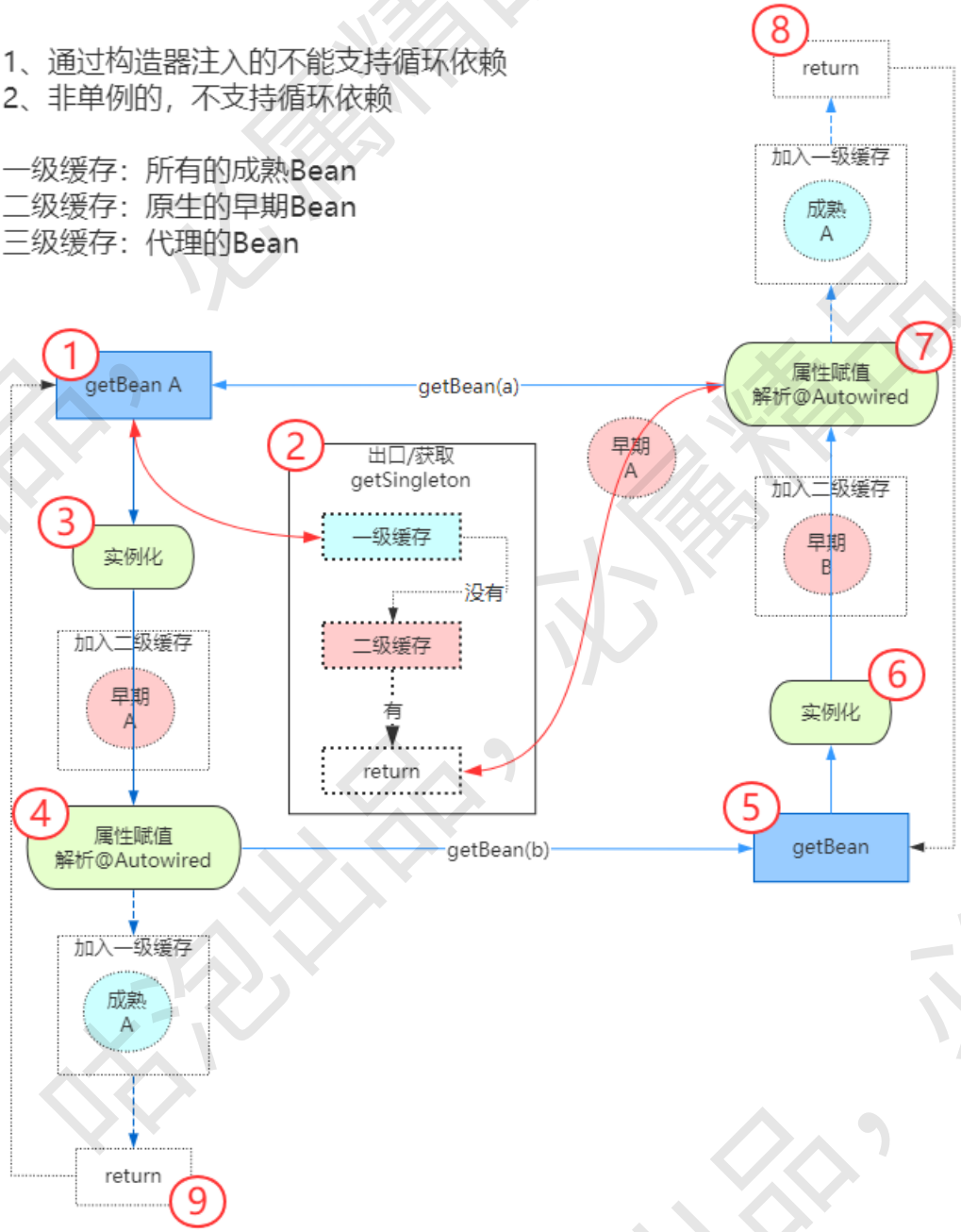


这种情况是怎么造成的呢？我们分析原因之后发现，因为，IoC 容器对 Bean 的初始化是根据 BeanDefinition 循环迭代，有一定的顺序。这样，在执行依赖注入时，需要自动赋值的属性对应的对象有可能还没初始化，没有初始化也就没有对应的实例可以注入。于是，就出现我们看到的情况。

使用缓存解决循环依赖问题

1、通过构造器注入的不能支持循环依赖
2、非单例的，不支持循环依赖

一级缓存：所有的成熟Bean
二级缓存：原生的早期Bean
三级缓存：代理的Bean



定义缓存

具体代码如下：

```
// 循环依赖的标识---当前正在创建的实例 bean
    private Set<String> singletonsCurrectlyInCreation = new HashSet<String>();

    //一级缓存
    private Map<String, Object> singletonObjects = new HashMap<String, Object>();
```

```
// 二级缓存：为了将成熟的bean和纯净的bean分离．避免读取到不完整的bean.
private Map<String, Object> earlySingletonObjects = new HashMap<String, Object>();
```

判断循环依赖

增加 getSingleton()方法：

```
/**
 * 判断是否是循环引用的出口.
 * @param beanName
 * @return
 */
private Object getSingleton(String beanName,GPBeanDefinition beanDefinition) {

    //先去一级缓存里拿，
    Object bean = singletonObjects.get(beanName);
    // 一级缓存中没有，但是正在创建的bean标识中有，说明是循环依赖
    if (bean == null && singletonsCurrentlyInCreation.contains(beanName)) {

        bean = earlySingletonObjects.get(beanName);
        // 如果二级缓存中没有，就从三级缓存中拿
        if (bean == null) {
            // 从三级缓存中取
            Object object = instantiateBean(beanName,beanDefinition);

            // 然后将其放入到二级缓存中．因为如果有多次依赖，就去二级缓存中判断．已经有了就不在再次创建了
            earlySingletonObjects.put(beanName, object);


        }
    }
    return bean;
}
```

添加缓存

修改 getBean()方法，在 getBean()方法中添加如下代码：

```
  //Bean的实例化，DI是从而这个方法开始的
   public Object getBean(String beanName){

       //1、先拿到BeanDefinition配置信息
       GPBeanDefinition beanDefinition = regitry.beanDefinitionMap.get(beanName);

       // 增加一个出口．判断实体类是否已经被加载过了
       Object singleton = getSingleton(beanName,beanDefinition);
       if (singleton != null) { return singleton; }

       // 标记bean正在创建
       if (!singletonsCurrentlyInCreation.contains(beanName)) {
           singletonsCurrentlyInCreation.add(beanName);
       }

       //2、反射实例化newInstance();
       Object instance = instantiateBean(beanName,beanDefinition);

       //放入一级缓存
       this.singletonObjects.put(beanName, instance);

       //3、封装成一个叫做BeanWrapper
       GPBeanWrapper beanWrapper = new GPBeanWrapper(instance);
```

```
        //4、执行依赖注入
        populateBean(beanName,beanDefinition,beanWrapper);
        //5、保存到 IoC 容器
        factoryBeanInstanceCache.put(beanName,beanWrapper);

        return beanWrapper.getWrapperInstance();
    }
```

添加依赖注入

修改 populateBean()方法，代码如下：

```
    private void populateBean(String beanName, GPBeanDefinition beanDefinition, GPBeanWrapper beanWrapper) {

        ...

            try {

                //ioc.get(beanName) 相当于通过接口的全名拿到接口的实现的实例
                field.set(instance,getBean(autowiredBeanName));
            } catch (IllegalAccessException e) {
                e.printStackTrace();
                continue;
            }
        ...
    }
```