一、谈谈MyBatis的启动过程

这个题目的主要目的是考察大家对MyBatis的理解深度，如果单纯的是做

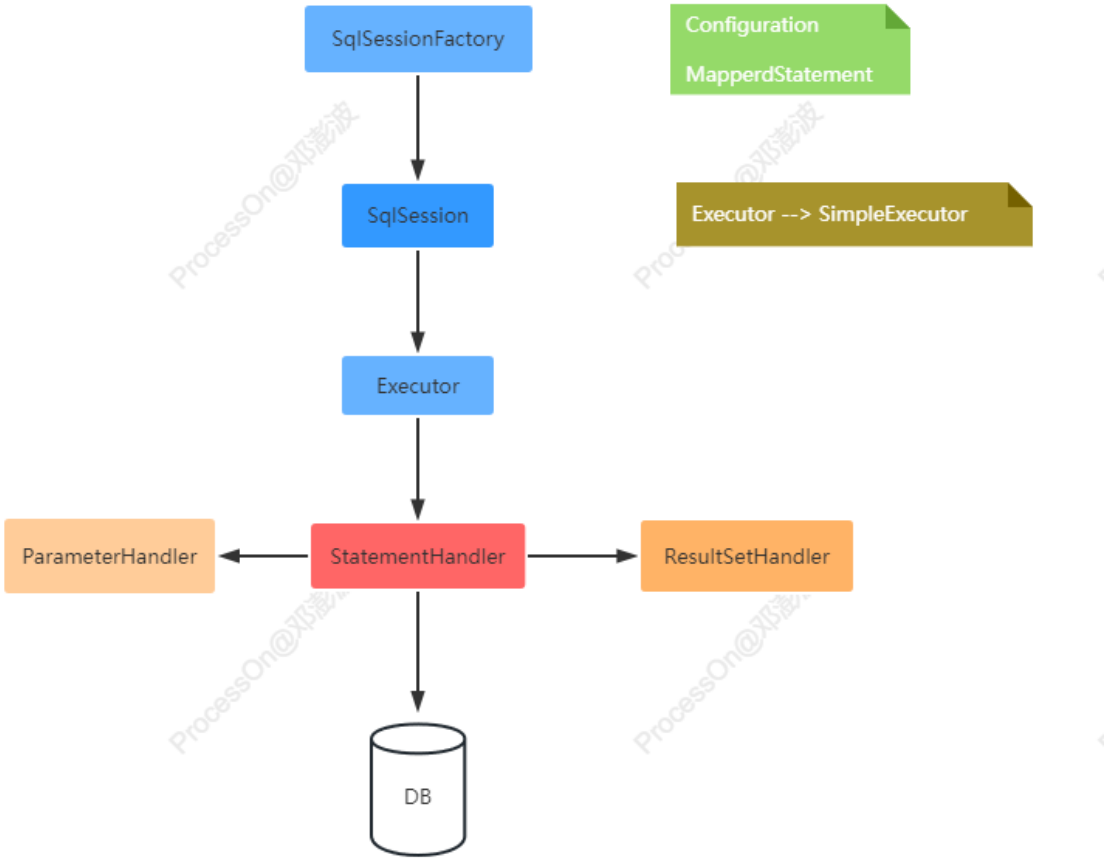
`CRUD`的开发，可能就忽略了。首先我们需要介绍清楚MyBatis执行启动的操作代码

```
@Test
public void start() throws Exception{
    // 1. 加载全局配置文件
    InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
    // 2.获取SqlSessionFactory
    // DefaultSqlSessionFactory
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
    // 3.获取SqlSession对象 DefaultSqlSession  Executor--> SimpleExecutor -->
CachingExecutor --> 插件的逻辑植入
    SqlSession sqlSession = factory.openSession();
    // 4.通过sqlSession的API接口实现数据库操作
    List<User> list =
sqlSession.selectList("com.bobo.vip.mapper.UserMapper.selectUserById",2);
    for (User user : list) {
        System.out.println(user);
    }
    // 关闭会话
    sqlSession.close();
}
```

具体的操作过程如下：

1. 加载配置文件：MyBatis的配置文件是一个XML文件，包含了数据库连接信息、映射文件的位置等配置信息。在启动过程中，MyBatis会读取并解析这个配置文件。
2. 创建SqlSessionFactory对象：SqlSessionFactory是MyBatis的核心对象，用于创建SqlSession对象。在启动过程中，MyBatis会根据配置文件中的信息，创建一个SqlSessionFactory对象。
3. 创建SqlSession对象：SqlSession是MyBatis的会话对象，用于执行数据库操作。在启动过程中，MyBatis会根据SqlSessionFactory对象，创建一个SqlSession对象。
4. 加载映射文件：映射文件是MyBatis的另一个重要配置，用于定义SQL语句与Java方法之间的映射关系。在启动过程中，MyBatis会根据配置文件中的信息，加载映射文件。
5. 初始化Mapper接口：Mapper接口是用于执行SQL语句的Java接口，在启动过程中，MyBatis会根据映射文件中的信息，动态生成Mapper接口的实现类。
6. 完成启动：启动过程完成后，就可以使用SqlSession对象执行数据库操作了。

当然这块的过程相对还是比较简单的。面试官可能会在这个基础上做相关的扩展。可以结合下面的图分析



也就是`SqlSessionFactory`对象的构建和`SqlSession`对象创建的核心过程。已经具体的数据库操作的请求是如何实现的。这块也是面试官比较感兴趣的内容。

- SqlSessionFactory：全局配置文件的加载解析和映射文件的加载解析
- SqlSession：相关的核心Executor和拦截器的实例化
- Executor：处理具体的请求涉及到缓存处理。分页扩展以及Sql解析和参数解析等

二、谈谈MyBatis中的缓存设计

3.1 缓存的作用

MyBatis缓存的作用是提高数据库访问性能。它将查询结果缓存到内存中，当下次有相同的查询请求时，直接从缓存中取出结果，避免了再次访问数据库，从而提高了查询的响应速度。

3.2 一级缓存

一级缓存是SqlSession级别的缓存，它默认是开启的。当同一个SqlSession执行相同的SQL语句时，会先从缓存中查找，如果找到了对应的结果，则直接返回缓存中的结果，而不会再次访问数据库。

要关闭一级缓存：

```
<setting name="localCacheScope" value="STATEMENT"/>
```

3.3 二级缓存

二级缓存是Mapper级别的缓存，它默认是关闭的。当不同的SqlSession执行相同的SQL语句时，如果开启了二级缓存，则会先从缓存中查找，如果找到了对应的结果，则直接返回缓存中的结果，而不会再次访问数据库。

要开启二级缓存：

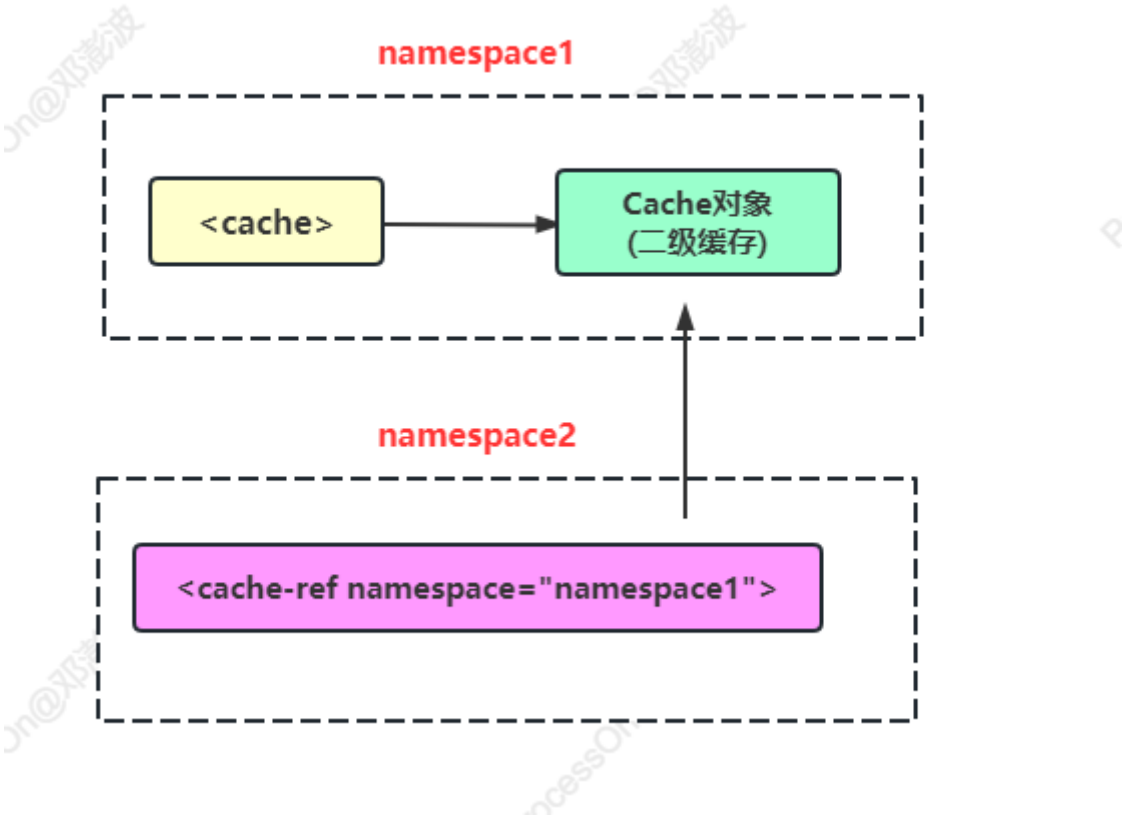
首先是在全局配置文件中设置：

```
<!-- 控制全局缓存（二级缓存），默认 true-->
<setting name="cacheEnabled" value="true"/>
```

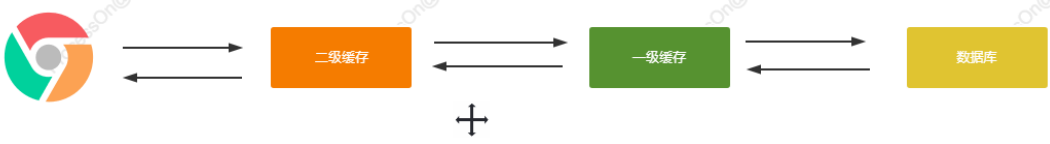
然后需要在具体的映射文件中设置 `cache` 标签

```
<cache />
<!--<cache type="org.mybatis.caches.redis.RedisCache"
       eviction="FIFO"
       flushInterval="60000"
       size="512"
       readOnly="true"/>-->
```

如果映射文件中的某个方法不想开启缓存可以设置 `useCache="false"` 处理

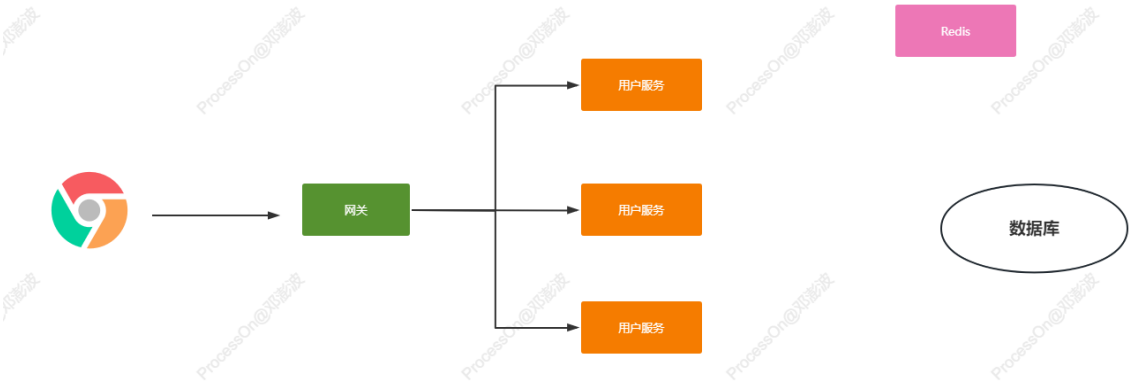


当我们开启了二级缓存后。查询操作是先走二级缓存还是先走一级缓存？



3.4 三级缓存

三级缓存是在分布式环境下把存储在内存中的数据持久化到第三方数据源的过程。比如Redis中。这块需要重写 `Cache` 接口



三、如何解决SqlSession的数据不安全的问题

1 SqlSession为什么是数据不安全的？

在MyBatis中，SqlSession是一个线程不安全的对象，主要原因如下：

1. SqlSession的底层实现是基于JDBC的Connection对象，而Connection对象是非线程安全的，因此SqlSession也是非线程安全的。
2. SqlSession中包含了数据库连接和事务相关的操作，如果多个线程共享同一个SqlSession实例，可能会导致数据的 `不一致性` 或者 `事务的混乱`。
3. SqlSession中的 `缓存机制` 也是基于当前线程的，如果多个线程共享同一个SqlSession实例，可能会导致缓存的数据混乱或者不一致。

2 如何解决这个问题？

为了保证数据的安全性和一致性，通常建议在每个线程中使用独立的SqlSession实例，可以通过工厂模式创建新的SqlSession对象，或者使用MyBatis提供的线程安全的SqlSessionFactory实例来创建SqlSession。另外，可以使用ThreadLocal来保证每个线程中使用的SqlSession对象是唯一的。

四、谈谈你对Spring的理解

1 Spring框架的发展历史

Spring框架的发展历史可以追溯到2002年，当时由Rod Johnson所著的《Expert One-on-One J2EE Development without EJB》一书中首次提出了Spring的概念和理念。

2003年，Spring Framework 1.0发布，成为了一个开源的JavaEE应用程序框架。Spring框架的设计目标是简化开发，并提供了对JavaEE平台的全面支持，包括依赖注入、面向切面编程、声明式事务管理等功能。

随着Spring的不断发展，2004年发布了Spring Framework 1.1版本，引入了许多新特性和改进。2006年发布了Spring Framework 2.0版本，其中最重要的改进是引入了基于注解的配置方式，使得开发者可以更加方便地配置和管理Spring应用程序。

2010年，Spring Framework 3.0发布，引入了许多重要的改进和新特性，包括对Java 5和Java 6的全面支持、对RESTful Web服务的支持、对JavaEE 6规范的支持等。

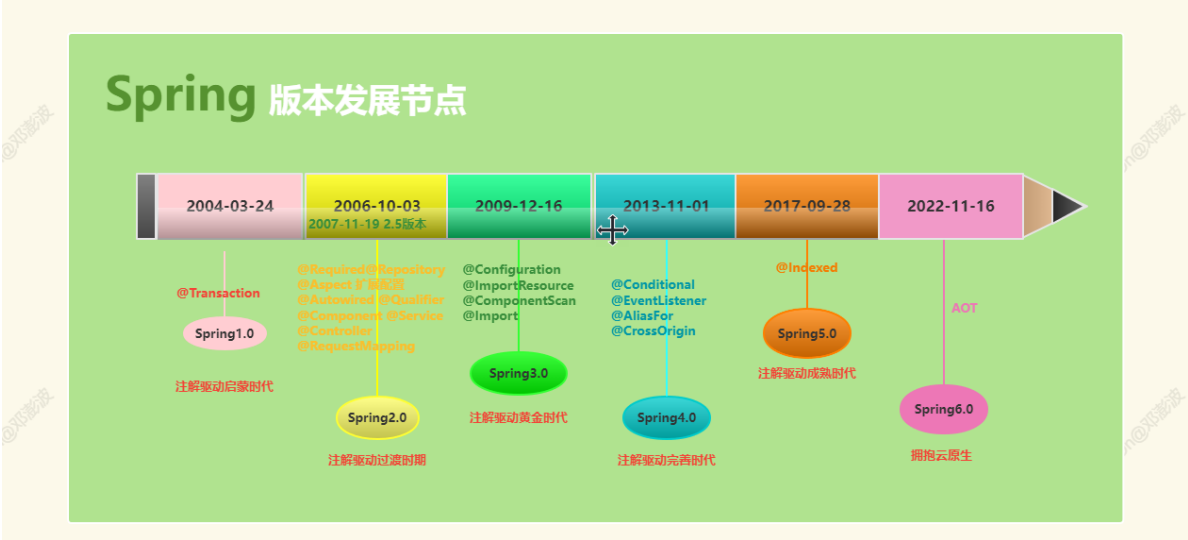
2013年，Spring Framework 4.0发布，其中最重要的改进是对Java 8的全面支持，包括支持Lambda表达式、函数式接口等。

2014年，Spring Framework 4.1发布，引入了许多新功能和改进，包括对WebSocket的支持、对Servlet 3.1规范的支持等。

2016年，Spring Framework 4.3发布，引入了一些新特性和改进，包括对HTTP/2的支持、对JDK 9的支持等。

2017年，Spring Framework 5.0发布，其中最重要的改进是对Reactive编程模型的支持，引入了Spring WebFlux和Reactor等组件。

2022年，Spring Framework 6.0发布，核心的内容是对JDK基线要求提升到了JDK17同时引入了AOT的内容开始拥抱云原生。



2 Spring框架的作用

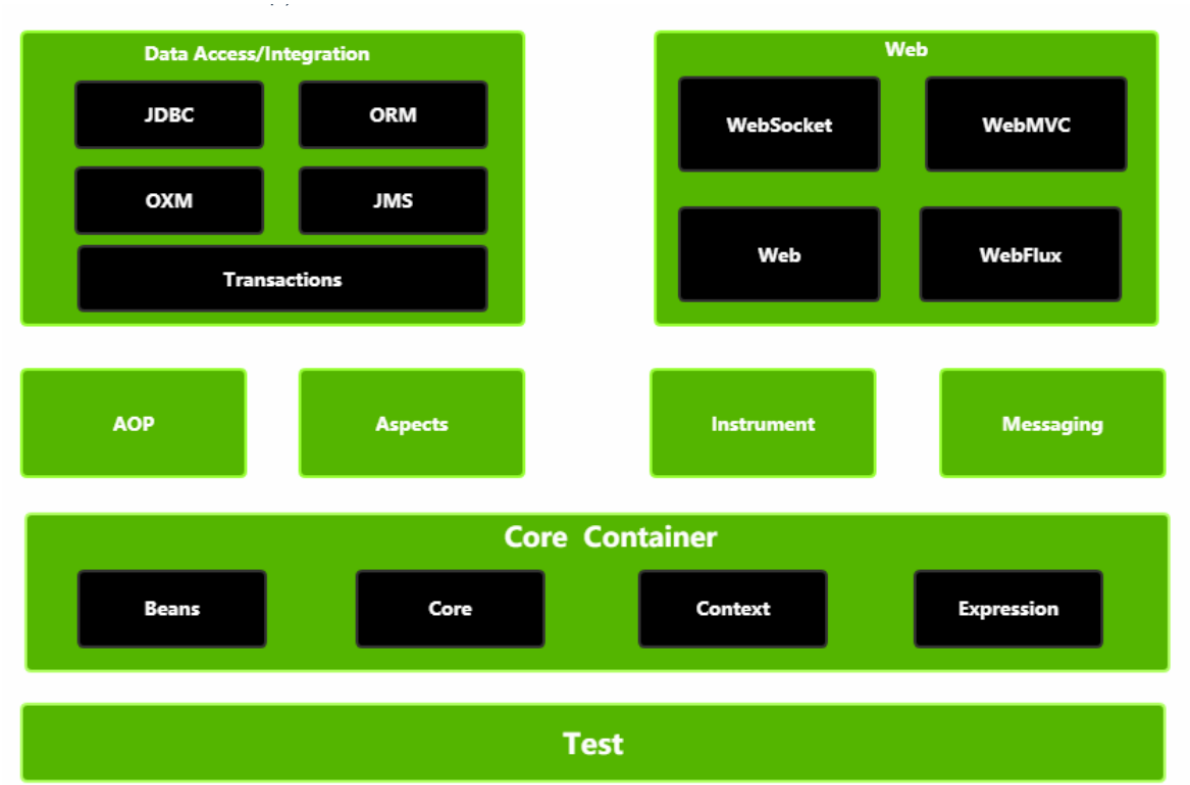
Spring框架是一个开源的Java应用框架，它是为了简化企业级Java应用的开发而设计的。Spring框架提供了一系列的工具和组件，帮助开发者构建可扩展、灵活、高效的企业级应用。

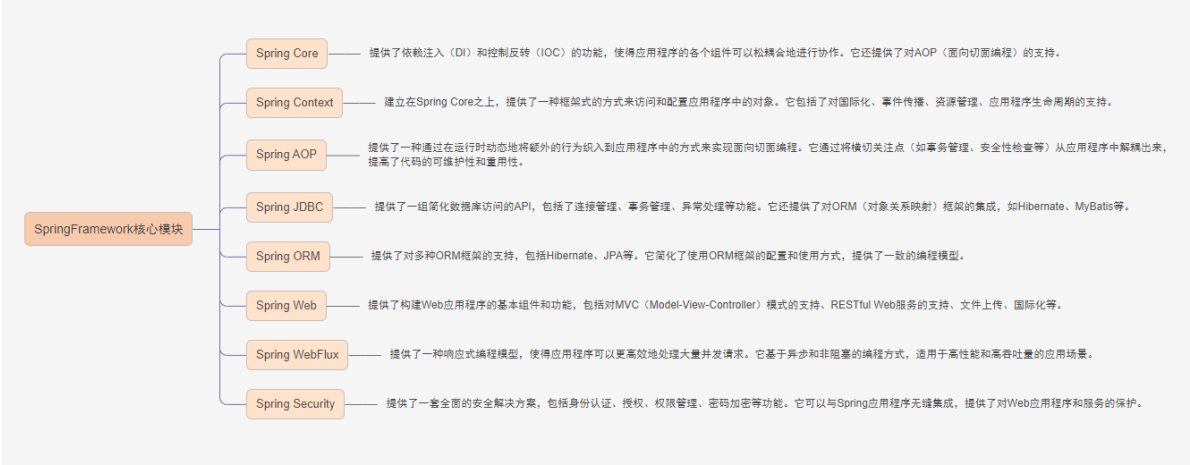
Spring框架的主要特点包括：

1. 轻量级：Spring框架的核心容器非常轻量，不依赖于其他框架或库，可以在任何Java应用中使用。
2. 控制反转（IoC）：Spring框架通过控制反转将对象的创建和依赖关系的管理交给了框架来处理，开发者只需关注业务逻辑的实现。
3. 面向切面编程（AOP）：Spring框架支持面向切面编程，可以将与业务逻辑无关的横切关注点（如事务管理、日志记录等）从业务逻辑中分离出来，提高代码的模块化和可重用性。
4. 容器：Spring框架提供了一个容器来管理和组织应用中的对象，可以通过配置文件或注解的方式管理对象的生命周期和依赖关系。
5. 集成：Spring框架可以与其他框架和库（如Hibernate、MyBatis、JPA等）无缝集成，简化开发者的工作。
6. 松耦合：Spring框架的组件之间是松耦合的，可以独立开发和测试各个组件，提高代码的可维护性和可测试性。
7. 可扩展性：Spring框架提供了扩展机制，可以根据需求定制和扩展框架的功能。

3 Spring框架的核心模块

经典的核心模块图片：





五、谈谈你对IoC的理解

1 IoC的理解

Spring中的IoC（Inversion of Control）是一种设计原则，通过该原则，对象的创建和依赖关系的管理被转移到了容器中，从而降低了对象之间的耦合性。我对Spring中的IoC的理解如下：

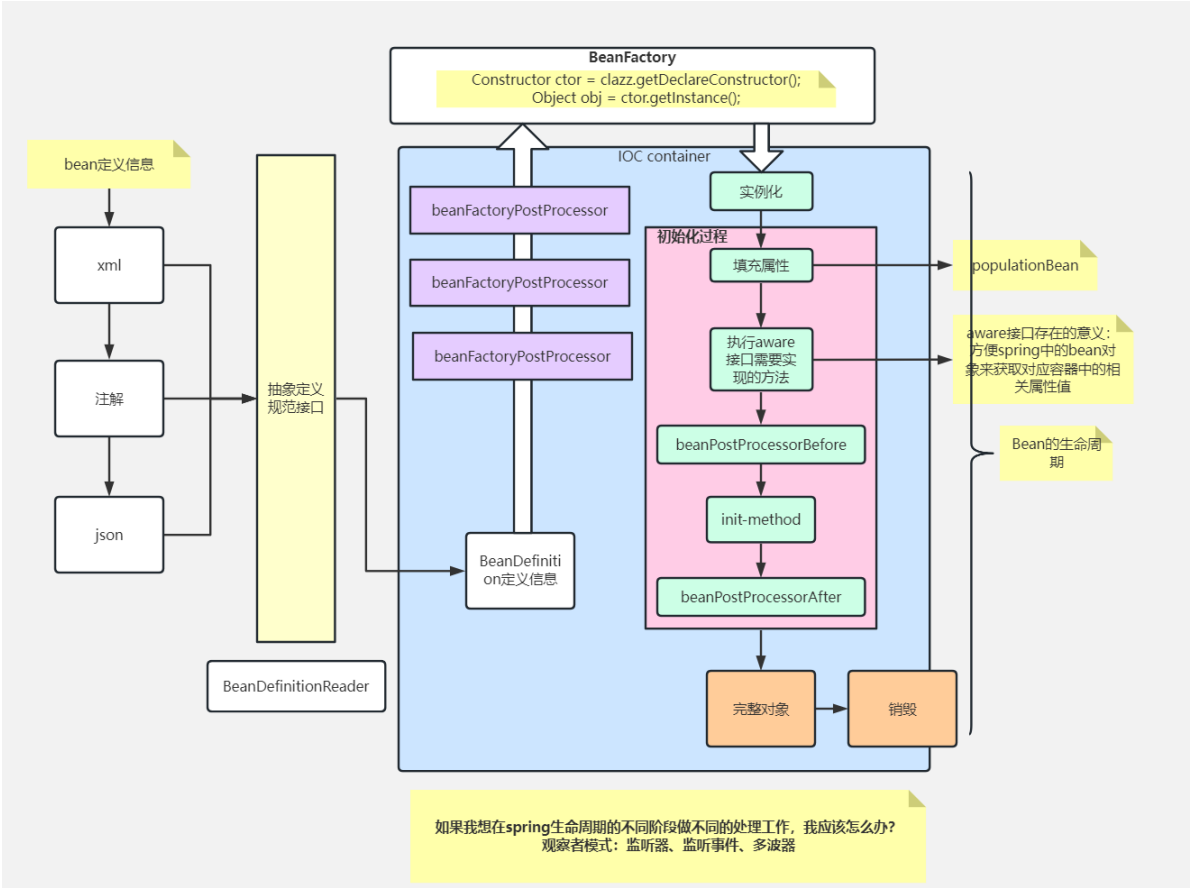
1. 控制反转：传统的对象创建和依赖关系的管理由程序员手动完成，而在IoC中，这些工作被交给了容器来完成。容器负责创建对象，并管理对象之间的依赖关系，将对象注入到其他对象中。
2. 容器：IoC的核心是容器，它负责创建和管理对象。容器通过读取配置文件或注解来了解对象之间的依赖关系，并根据这些信息创建对象，并将它们注入到其他对象中。
3. 依赖注入：容器通过依赖注入的方式来实现对象之间的解耦。依赖注入可以通过构造函数、Setter方法或注解来完成。通过依赖注入，对象只需要关注自己的业务逻辑，而不需要关心如何获取依赖对象。
4. 松耦合：IoC使得对象之间的依赖关系变得松耦合，对象只需要依赖抽象而不依赖具体实现。这使得代码更加灵活和可维护，可以方便地替换依赖的对象。
5. 可测试性：IoC使得代码更容易进行单元测试。由于对象的依赖关系被注入，可以方便地使用模拟对象来进行测试，减少了对外部资源的依赖。

总而言之，Spring中的IoC通过控制反转、依赖注入和松耦合的方式，实现了对象的创建和依赖关系的管理，提高了代码的灵活性、可维护性和可测试性。

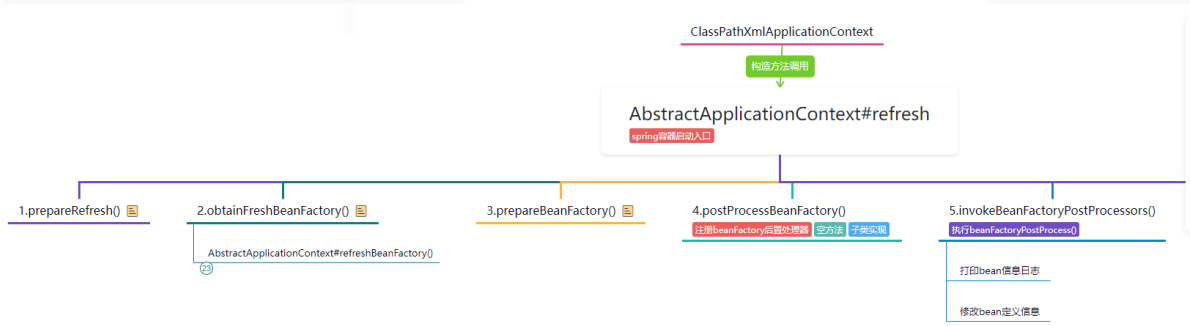
2 IoC的实现原理

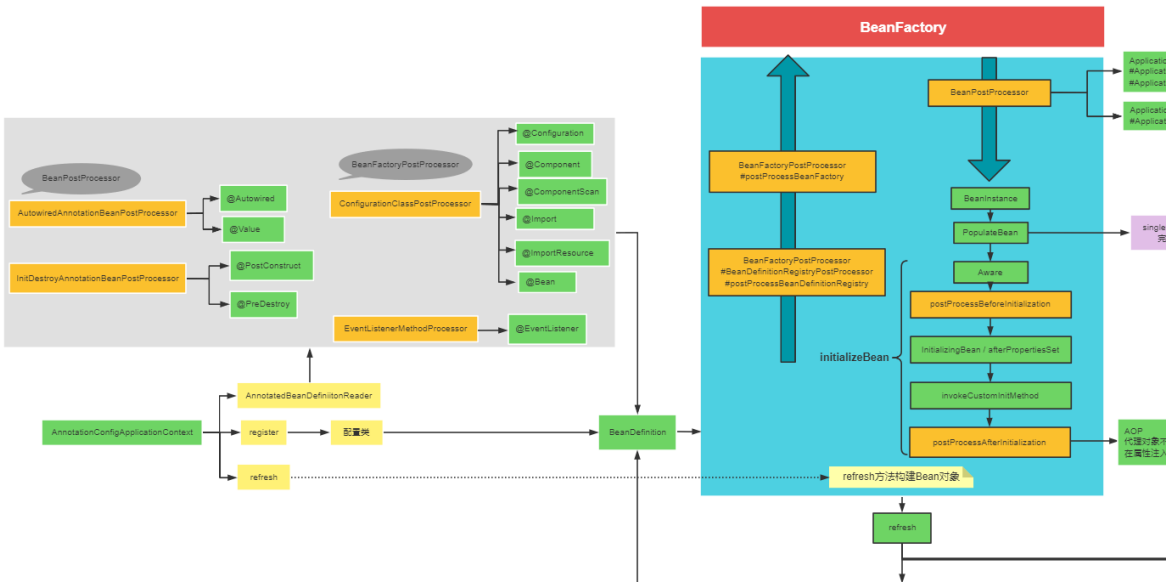
可以从比较粗的角度介绍写IoC中的核心对象的作用：

- `BeanDefinition`:BeanDefinition是Spring中定义Bean的元数据信息的接口。它包含了Bean的类名、属性、构造函数参数等信息，通过BeanDefinition可以告诉容器如何创建和配置Bean。
- `BeanDefinitionRegistry`：BeanDefinitionRegistry提供了对BeanDefinition的管理和操作功能，是Spring IoC容器中用于注册和管理Bean的核心接口。
- `BeanFactory`：BeanFactory是Spring IoC容器的核心接口，负责实例化、配置和管理应用中的对象（Bean）。它是IoC容器的基础，提供了一种获取Bean的机制，可以通过Bean的名称或类型来获取Bean的实例。
- `ApplicationContext`：ApplicationContext是BeanFactory的子接口，它是Spring中更高级的IoC容器。除了提供BeanFactory的功能外，它还提供了更多的企业级功能，例如国际化支持、事件发布、资源加载等。
- `BeanPostProcessor`：BeanPostProcessor是Spring中的一个扩展接口，用于在Bean实例化和依赖注入的过程中对Bean进行增强处理。通过实现BeanPostProcessor接口，可以在Bean的初始化前后对Bean进行自定义操作。
- `BeanWrapper`：BeanWrapper是Spring中对Bean对象的一种封装，提供了对Bean属性的访问和设置的方法。BeanWrapper可以对Bean的属性进行类型转换和验证等操作。



当然这块面试官还可能会继续深入的需要你详细介绍IoC的过程。这时可以参考下面的结构图来查看了

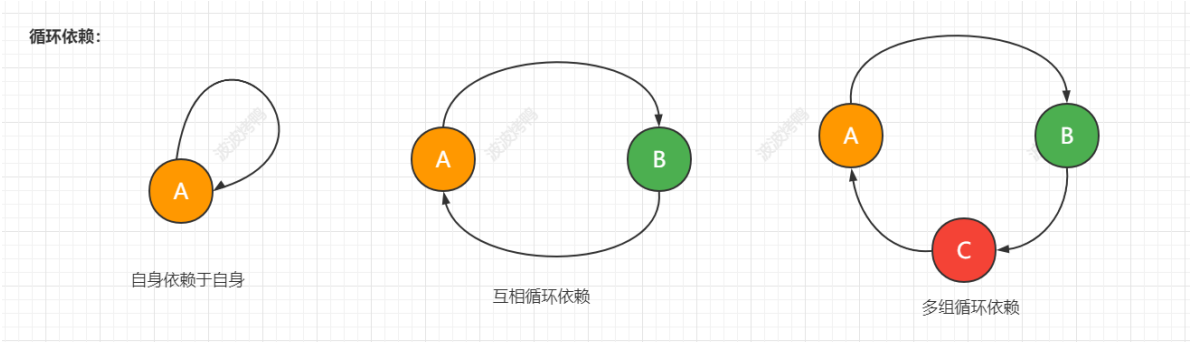




六、Spring中是如何解决循环依赖的

3.1 什么是循环依赖

看下图



上图是循环依赖的三种情况，虽然方式有点不一样，但是循环依赖的本质是一样的，就你的完整创建要依赖与我，我的完整创建也依赖于你。相互依赖从而没法完整创建造成失败。我们通过代码演示加强下理解

```
public class CircularTest {

    public static void main(String[] args) {
        new CircularTest1();
    }
}
class CircularTest1{
    private CircularTest2 circularTest2 = new CircularTest2();
}

class CircularTest2{
    private CircularTest1 circularTest1 = new CircularTest1();
}
```

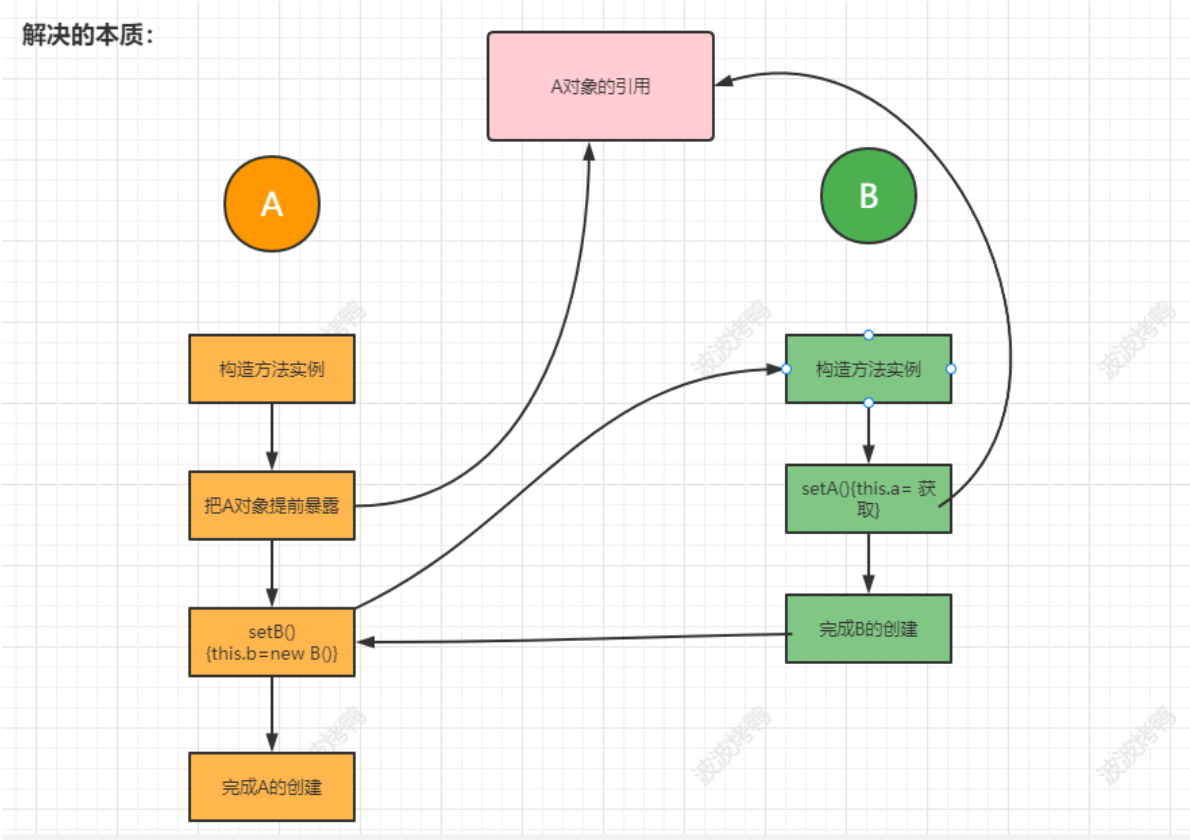
执行后出现了 StackOverflowError 错误

```
D:\software\java\jdk8\bin\java.exe ...
Exception in thread "main" java.lang.StackOverflowError Create breakpoint
    at com.bobo.circular.CircularTest2.<init>(CircularTest.java:14)
    at com.bobo.circular.CircularTest1.<init>(CircularTest.java:10)
    at com.bobo.circular.CircularTest2.<init>(CircularTest.java:14)
    at com.bobo.circular.CircularTest1.<init>(CircularTest.java:10)
    at com.bobo.circular.CircularTest2.<init>(CircularTest.java:14)
    at com.bobo.circular.CircularTest1.<init>(CircularTest.java:10)
    at com.bobo.circular.CircularTest2.<init>(CircularTest.java:14)
    at com.bobo.circular.CircularTest1.<init>(CircularTest.java:10)
```

上面的就是最基本的循环依赖的场景，你需要我，我需要你，然后就报错了。而且上面的这种设计情况我们是没有办法解决的。那么针对这种场景我们应该要怎么设计呢？这个是关键！

问题分析：

首先我们要明确一点就是如果这个对象A还没创建成功，在创建的过程中要依赖另一个对象B，而另一个对象B也是在创建中要依赖对象A，这种肯定是无解的，这时我们就要转换思路，我们先把A创建出来，但是还没有完成初始化操作，也就是这是一个半成品的对象，然后在赋值的时候先把A暴露出来，然后创建B，让B创建完成后找到暴露的A完成整体的实例化，这时再把B交给A完成A的后续操作，从而揭开了循环依赖的密码。也就是如下图：



3.2 怎么解决

明白了上面的本质后，我们可以自己来尝试解决下：

先来把上面的案例改为set/get来依赖关联

```
public class CircularTest {

    public static void main(String[] args) throws Exception{
        System.out.println(getBean(CircularTest1.class).getCircularTest2());
        System.out.println(getBean(CircularTest2.class).getCircularTest1());
    }

    private static <T> T getBean(Class<T> beanClass) throws Exception{
        // 1.获取 实例对象
        Object obj = beanClass.newInstance();
        // 2.完成属性填充
        Field[] declaredFields = obj.getClass().getDeclaredFields();
        // 遍历处理
        for (Field field : declaredFields) {
            field.setAccessible(true); // 针对private修饰
            // 获取成员变量 对应的类对象
            Class<?> fieldClass = field.getType();
            // 获取对应的 beanName
            String fieldBeanName = fieldClass.getSimpleName().toLowerCase();
            // 给成员变量赋值 如果 singletonObjects 中有半成品就获取，否则创建对象
            field.set(obj,getBean(fieldClass));
```

```
        }
        return (T) obj;
    }
}

class CircularTest1{
    private CircularTest2 circularTest2;

    public CircularTest2 getCircularTest2() {
        return circularTest2;
    }

    public void setCircularTest2(CircularTest2 circularTest2) {
        this.circularTest2 = circularTest2;
    }
}

class CircularTest2{
    private CircularTest1 circularTest1;

    public CircularTest1 getCircularTest1() {
        return circularTest1;
    }

    public void setCircularTest1(CircularTest1 circularTest1) {
        this.circularTest1 = circularTest1;
    }
}
```

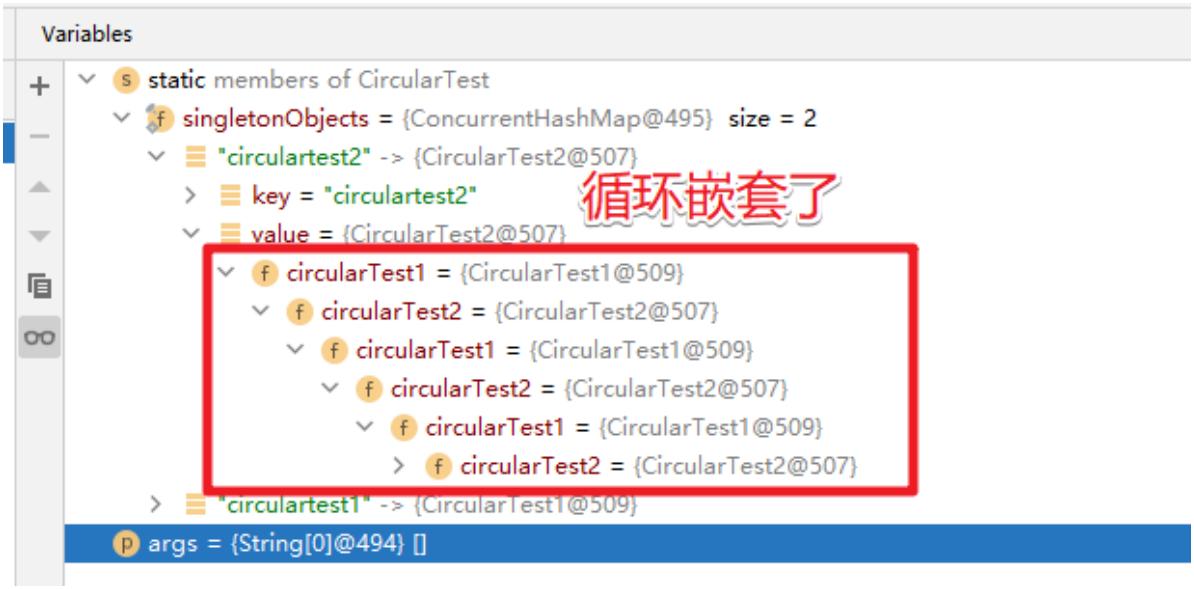
然后我们再通过把对象实例化和成员变量赋值拆解开来处理。从而解决循环依赖的问题

```
public class CircularTest {
    // 保存提前暴露的对象，也就是半成品的对象
    private final static Map<String,Object> singletonObjects = new
ConcurrentHashMap<>();

    public static void main(String[] args) throws Exception{
        System.out.println(getBean(CircularTest1.class).getCircularTest2());
        System.out.println(getBean(CircularTest2.class).getCircularTest1());
    }

    private static <T> T getBean(Class<T> beanClass) throws Exception{
        //1.获取类对象对应的名称
        String beanName = beanClass.getSimpleName().toLowerCase();
        // 2.根据名称去 singletonObjects 中查看是否有半成品的对象
        if(singletonObjects.containsKey(beanName)){
            return (T) singletonObjects.get(beanName);
        }
        // 3. singletonObjects 没有半成品的对象，那么就反射实例化对象
        Object obj = beanClass.newInstance();
        // 还没有完整的创建完这个对象就把这个对象存储在了 singletonObjects中
        singletonObjects.put(beanName,obj);
        // 属性填充来补全对象
        Field[] declaredFields = obj.getClass().getDeclaredFields();
        // 遍历处理
        for (Field field : declaredFields) {
            field.setAccessible(true); // 针对private修饰
```

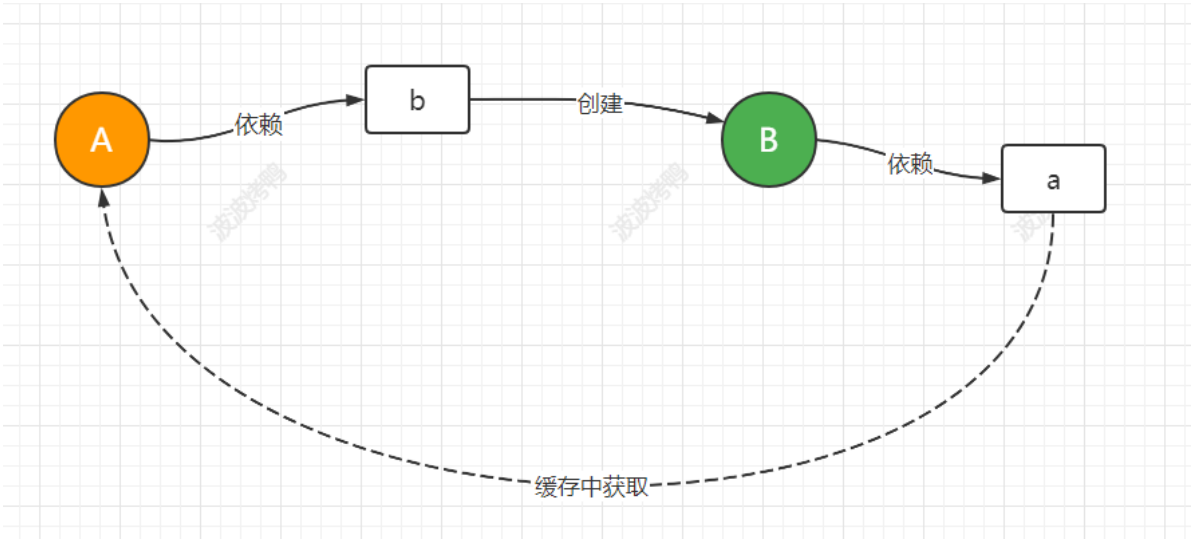
```
            // 获取成员变量 对应的类对象
            Class<?> fieldClass = field.getType();
            // 获取对应的 beanName
            String fieldBeanName = fieldClass.getSimpleName().toLowerCase();
            // 给成员变量赋值 如果 singletonObjects 中有半成品就获取，否则创建对象
            field.set(obj,singletonObjects.containsKey(fieldBeanName)?
                    singletonObjects.get(fieldBeanName):getBean(fieldClass));
        }
        return (T) obj;
    }
}

class CircularTest1{
    private CircularTest2 circularTest2;

    public CircularTest2 getCircularTest2() {
        return circularTest2;
    }

    public void setCircularTest2(CircularTest2 circularTest2) {
        this.circularTest2 = circularTest2;
    }
}

class CircularTest2{
    private CircularTest1 circularTest1;

    public CircularTest1 getCircularTest1() {
        return circularTest1;
    }

    public void setCircularTest1(CircularTest1 circularTest1) {
        this.circularTest1 = circularTest1;
    }
}
```

运行程序你会发现问题完美的解决了

在上面的方法中的核心是getBean方法，Test1 创建后填充属性时依赖Test2，那么就去创建 Test2，在创建 Test2 开始填充时发现依赖于 Test1，但此时 Test1 这个半成品对象已经存放在缓存到 `singletonObjects` 中了，所以Test2可以正常创建，在通过递归把 Test1 也创建完整了。

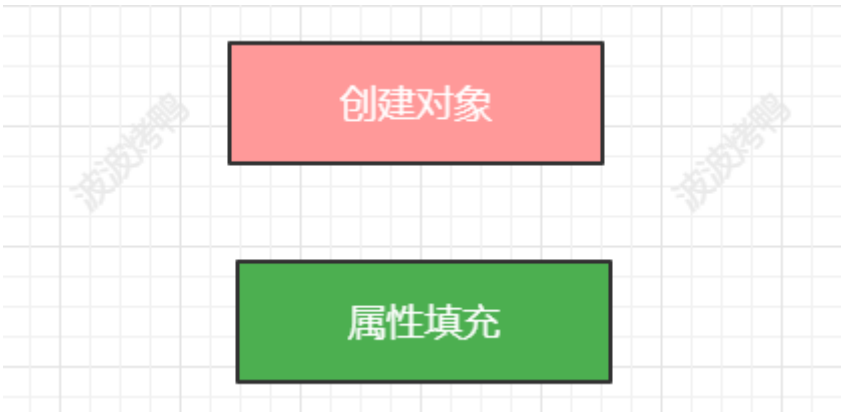


最后总结下该案例解决的本质：

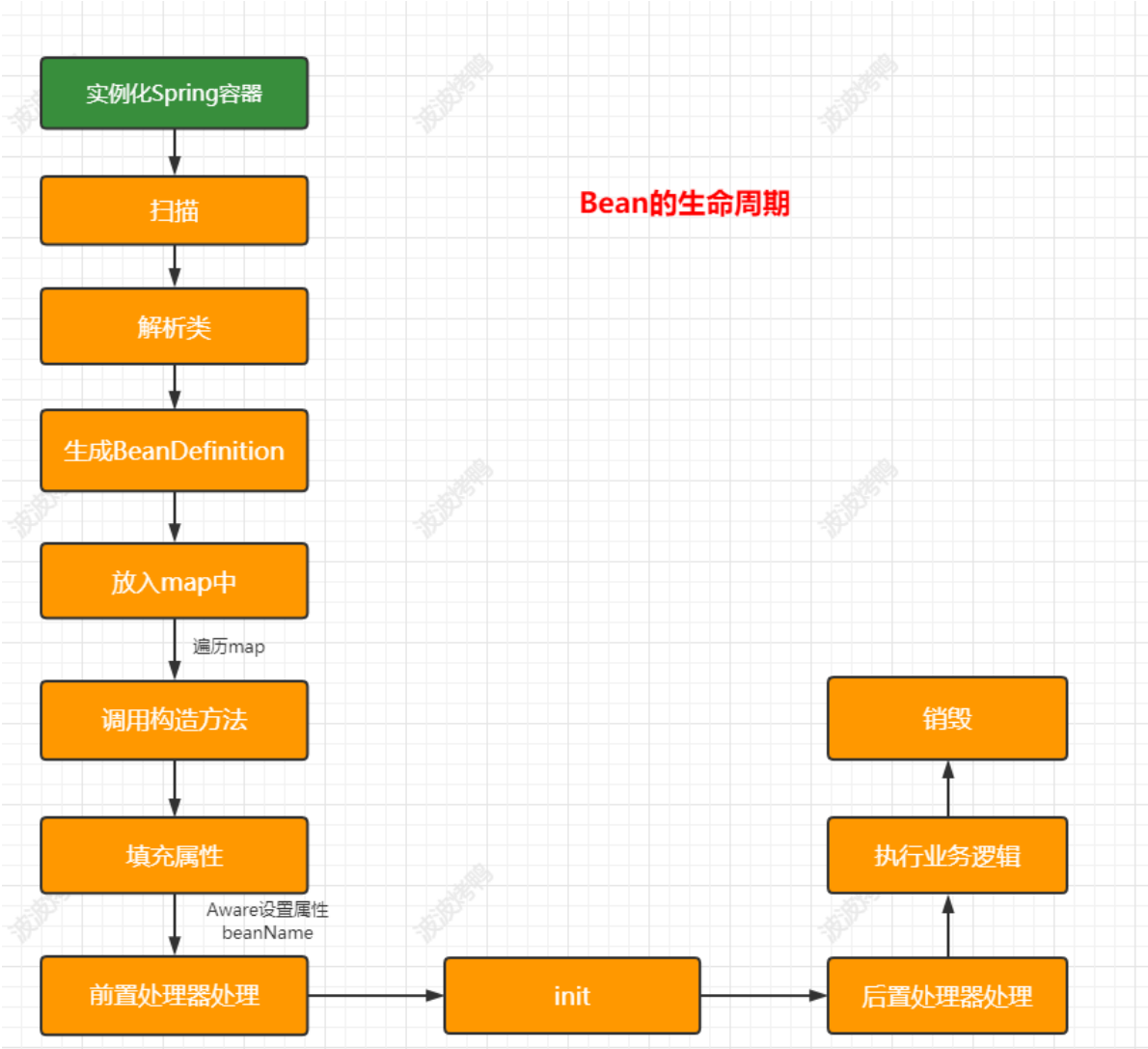


3.3 Spring循环依赖

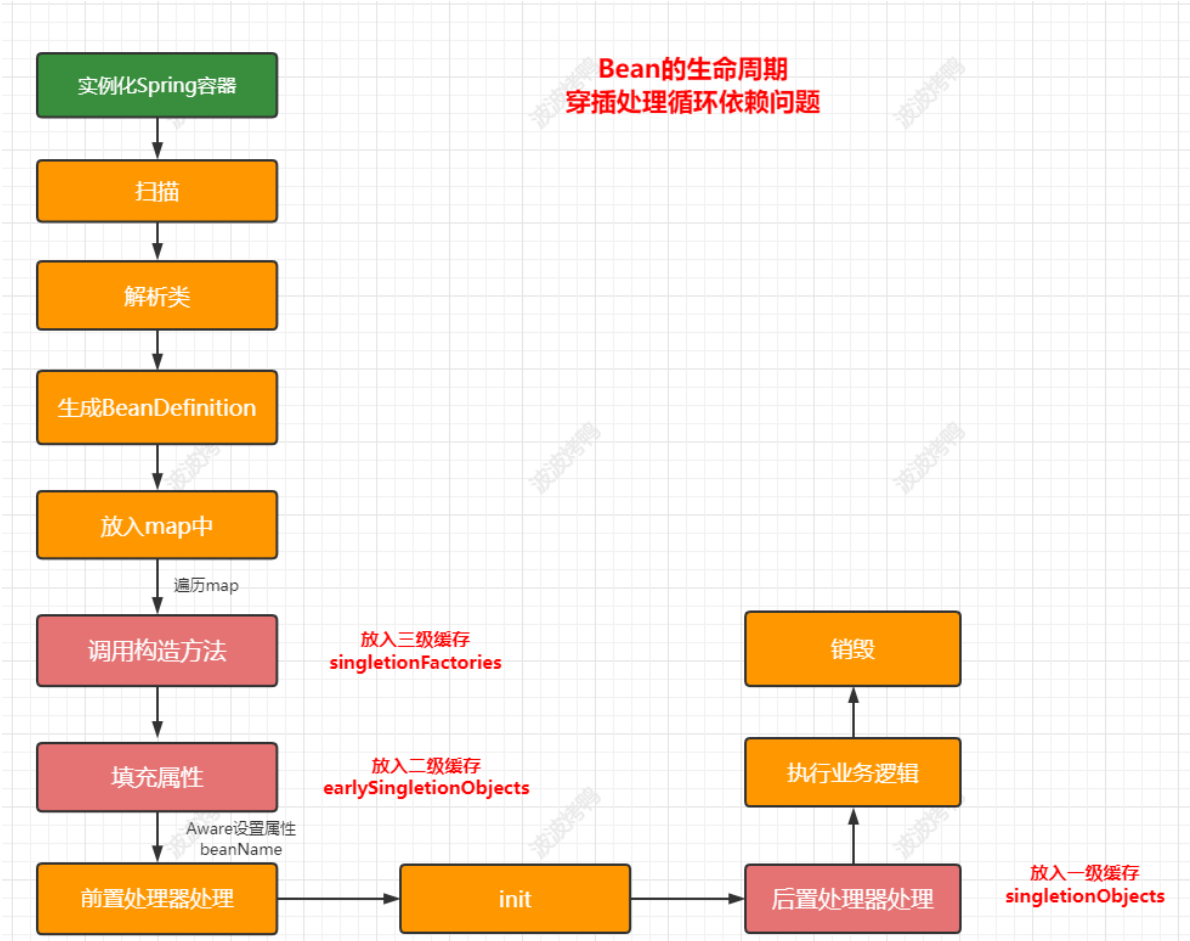
然后我们再来看看Spring中是如何解决循环依赖问题的呢？刚刚上面的案例中的对象的生命周期的核心就两个



而Spring创建Bean的生命周期中涉及到的方法就很多了。下面是简单列举了对应的方法

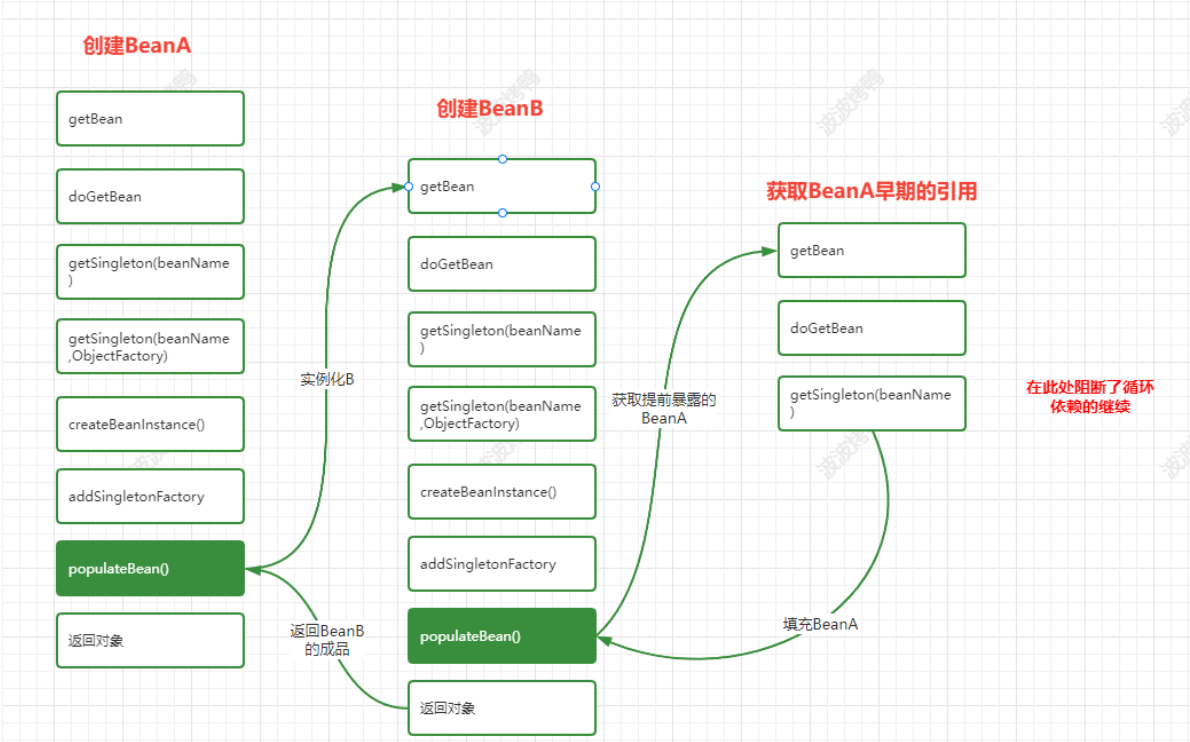


基于前面案例的了解，我们知道肯定需要在调用构造方法方法创建完成后再暴露对象，在Spring中提供了三级缓存来处理这个事情，对应的处理节点如下图：



- 一级缓存：存储的是 成品Bean对象 存储的所有的单例对象 其实可以说和循环依赖没有关系
- 二级缓存：存储的是 半成品对象 -->是解决循环依赖的关键，如果不去考虑AOP代理增加的情况，只有二级缓存的情况下也是可以解决循环依赖的，也就是不需要三级缓存。
- 三级缓存：三级缓存存在的意义是解决AOP增强对象的原因，存储的是一个Lambda表达式(内部类)--》ObjectFactory

对应到源码中具体处理循环依赖的流程如下：



上面就是在Spring的生命周期方法中和循环依赖出现相关的流程了。那么源码中的具体处理是怎么样的呢？我们继续往下面看。

首先在调用构造方法的后会放入到三级缓存中

```
        // Eagerly cache singletons to be able to resolve circular references
        // even when triggered by lifecycle interfaces like BeanFactoryAware.
        // 判断当前bean是否需要提前曝光：单例&允许循环依赖&当前bean正在创建中，检测循环依赖
        boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
                isSingletonCurrentlyInCreation(beanName));
        if (earlySingletonExposure) {
            if (logger.isTraceEnabled()) {
                logger.trace("Eagerly caching bean '" + beanName +
                        "' to allow for resolving potential circular references");
            }
            // 为避免后期循环依赖，可以在bean初始化完成前将创建实例的ObjectFactory加入工厂
            addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));

            //只保留二级缓存，不向三级缓存中存放对象
            earlySingletonObjects.put(beanName,bean);
            registeredSingletons.add(beanName);
//
//          synchronized (this.singletonObjects) {
```

下面就是放入三级缓存的逻辑

```
    protected void addSingletonFactory(String beanName, ObjectFactory<?>
singletonFactory) {
        Assert.notNull(singletonFactory, "Singleton factory must not be null");
        // 使用singletonObjects进行加锁，保证线程安全
        synchronized (this.singletonObjects) {
            // 如果单例对象的高速缓存【beam名称-bean实例】没有beanName的对象
            if (!this.singletonObjects.containsKey(beanName)) {
                // 将beanName,singletonFactory放到单例工厂的缓存【bean名称 -
ObjectFactory】
                this.singletonFactories.put(beanName, singletonFactory);
                // 从早期单例对象的高速缓存【bean名称-bean实例】 移除beanName的相关缓存对
象
                this.earlySingletonObjects.remove(beanName);
                // 将beanName添加已注册的单例集中
                this.registeredSingletons.add(beanName);
            }
        }
    }
```

然后在填充属性的时候会存入二级缓存中

```
earlySingletonObjects.put(beanName,bean);
registeredSingletons.add(beanName);
```

最后把创建的对象保存在了一级缓存中

```
    protected void addSingleton(String beanName, Object singletonObject) {
        synchronized (this.singletonObjects) {
            // 将映射关系添加到单例对象的高速缓存中
            this.singletonObjects.put(beanName, singletonObject);
            // 移除beanName在单例工厂缓存中的数据
            this.singletonFactories.remove(beanName);
            // 移除beanName在早期单例对象的高速缓存的数据
            this.earlySingletonObjects.remove(beanName);
            // 将beanName添加到已注册的单例集中
            this.registeredSingletons.add(beanName);
        }
    }
```

3.4 疑问点

这些疑问点也是面试官喜欢问的问题点

为什么需要三级缓存

三级缓存主要处理的是AOP的代理对象，存储的是一个ObjectFactory

三级缓存考虑的是代理对象，而二级缓存考虑的是性能-从三级缓存的工厂里创建出对象，再扔到二级缓存（这样就不用每次都要从工厂里拿）

没有三级环境能解决吗？

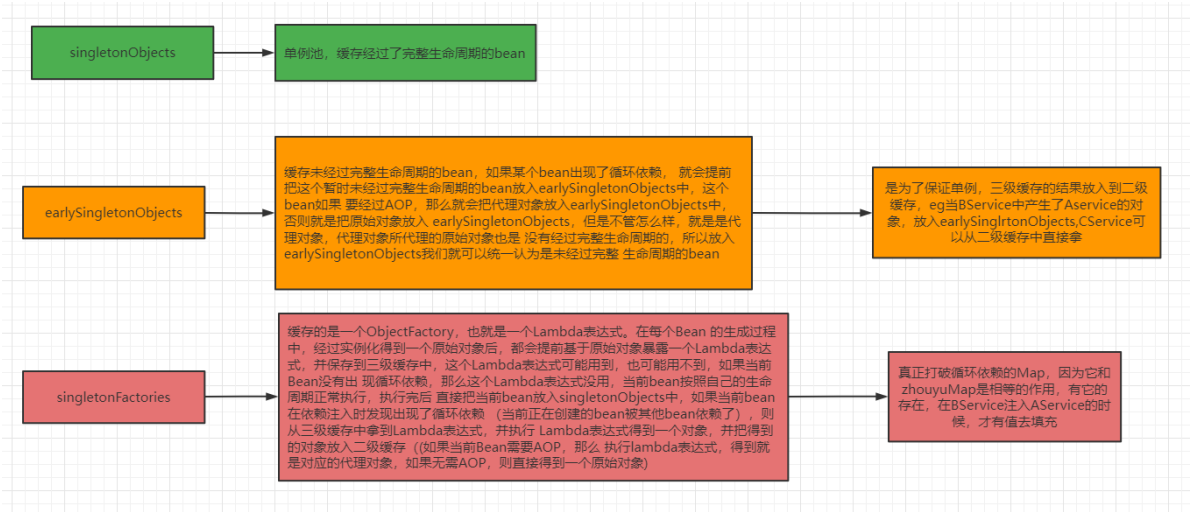
没有三级缓存是可以解决循环依赖问题的

三级缓存分别什么作用

一级缓存：正式对象

二级缓存：半成品对象

三级缓存：工厂



Spring中Bean有两种方式：单例、原型、构造注入和设值注入

- 单例：可以解决循环依赖问题
- 原型：不能解决循环依赖问题
- 构造注入：不能解决循环依赖问题
- 设置注入：可以解决循环依赖问题

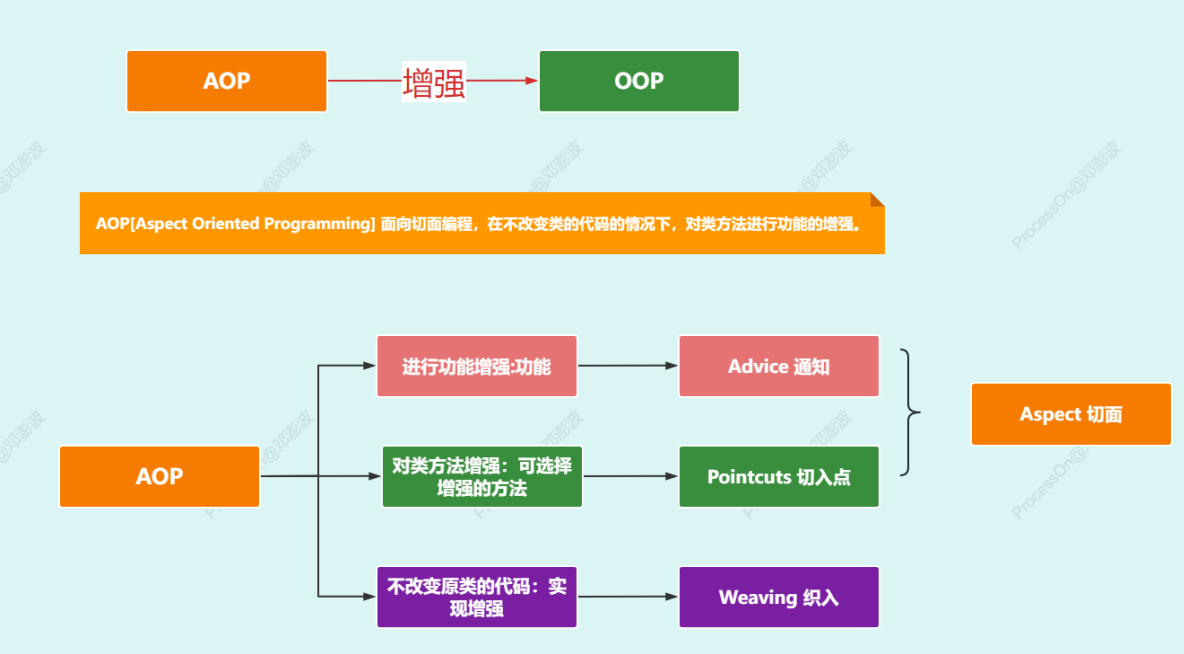
该问题的回答方式：

1. 循环依赖的原因和解决方案
2. Spring中循环依赖的解决方案

七、谈谈你对AOP的理解

5.1 AOP的概念

`AOP`（面向切面编程）是一种软件设计思想，旨在通过将横切关注点与主要业务逻辑分离，提高代码的可维护性和可重用性。是 `OOP`(面向对象)的有效补充。



在传统的面向对象编程中，代码的功能通常被分散在多个类中，这种分散导致了代码的重复和散乱。AOP的目标是通过将这些横切关注点（如日志记录、权限验证、事务管理等）从核心业务逻辑中分离出来，并将其封装成可复用的模块，从而实现代码的解耦和重用。

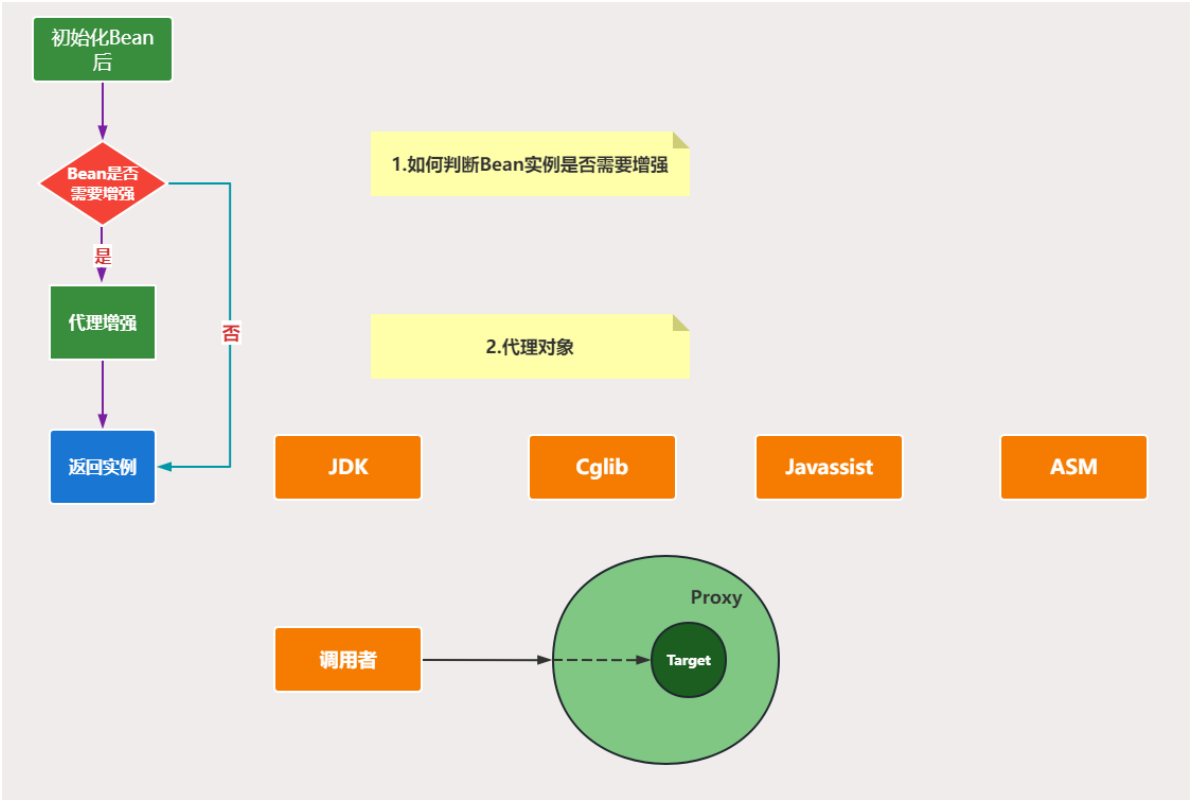
AOP的关键概念是切面（Aspect），切面是横切关注点的模块化，它定义了在何处（连接点）以及何时（切点）应用特定的横切关注点。切面可以通过通知（Advice）来实现具体的横切功能，通常包括前置通知（Before）、后置通知（After）、异常通知（AfterThrowing）和返回通知（AfterReturning）等。

5.2 AOP的实现方式

https://blog.csdn.net/qq_38526573/article/details/86441916

5.3 AOP源码分析

https://dpb-bobokaoya-sm.blog.csdn.net/article/details/127263992



八、谈谈你对自动装配原理的理解

8.1 回答的关键

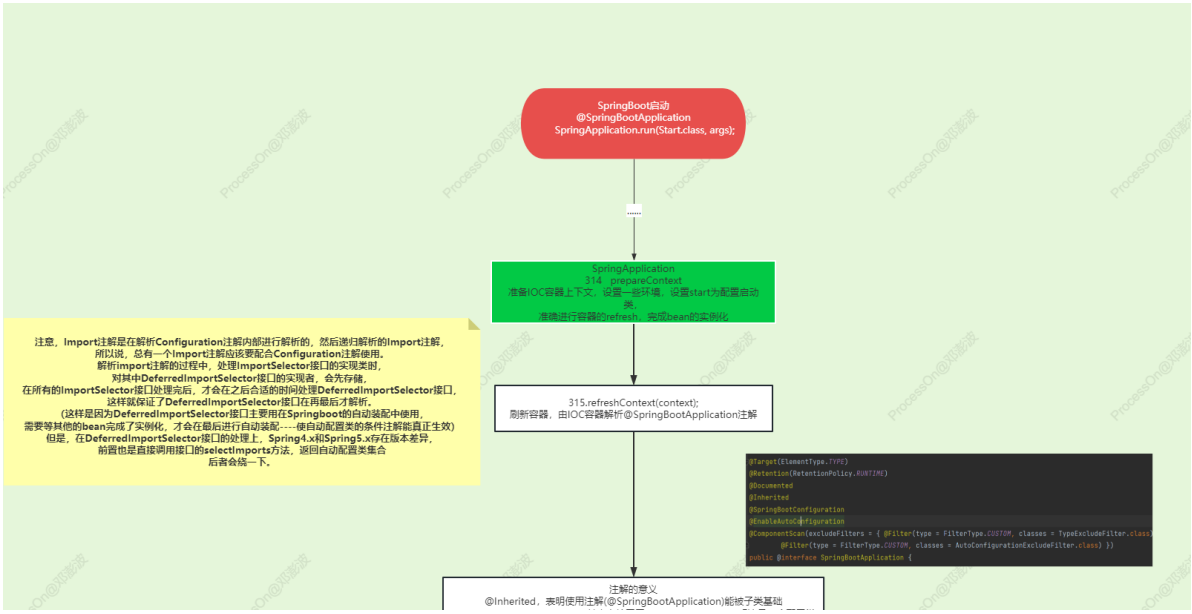
- 约定优于配置
- 自动扫描
- 本质就是Spring的脚手架
- @EnableAutoConfiguration
- spring.factories
- 条件注解

8.2 源码的梳理

源码梳理从四个方面介绍

- run方法：Spring容器的初始化
- @SpringBootApplication注解:自动扫描、自动装配
- 两者之间是怎么串联的
- 自定义starter

https://www.processon.com/view/link/64c06b8938c1420369f6958a


SpringBoot启动
@SpringBootApplication
SpringApplication.run(Start.class, args);
.....
SpringApplication
314 prepareContext
准备IOC容器上下文，设置一些环境，设置start为配置启动类，
准确进行容器的refresh，完成bean的实例化
注意，Import注解是在解析Configuration注解内部进行解析的，然后递归解析的Import注解，
所以说，总有一个Import注解应该要配合Configuration注解使用。
解析import注解的过程中，处理ImportSelector接口的实现类时，
对其中DeferredImportSelector接口的实现者，会先存储，
在所有的ImportSelector接口处理完后，才会在之后合适的时间处理DeferredImportSelector接口，
这样就保证了DeferredImportSelector接口在再最后才解析。
(这样是因为DeferredImportSelector接口主要用在Springboot的自动装配中使用，
需要等其他的bean完成了实例化，才会在最后进行自动装配----使自动配置类的条件注解能真正生效)
但是，在DeferredImportSelector接口的处理上，Spring4.x和Spring5.x存在版本差异，
前置也是直接调用接口的selectImports方法，返回自动配置类集合
后者会绕一下。
315.refreshContext(context);
刷新容器，由IOC容器解析@SpringBootApplication注解
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class)
@Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
注解的意义
@Inherited，表明使用注解(@SpringBootApplication)能被子类基础