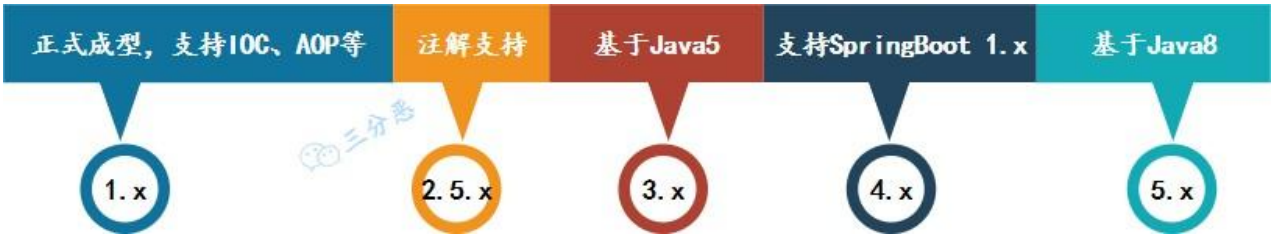
> 图文详解 35 道 Spring 面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：楼仔，作者：三分恶，戳原文链接。

基础

1.Spring 是什么？特性？有哪些模块？

一句话概括：Spring **是一个轻量级、非入侵式的控制反转** (IoC) **和面向切面** (AOP) **的框架。**

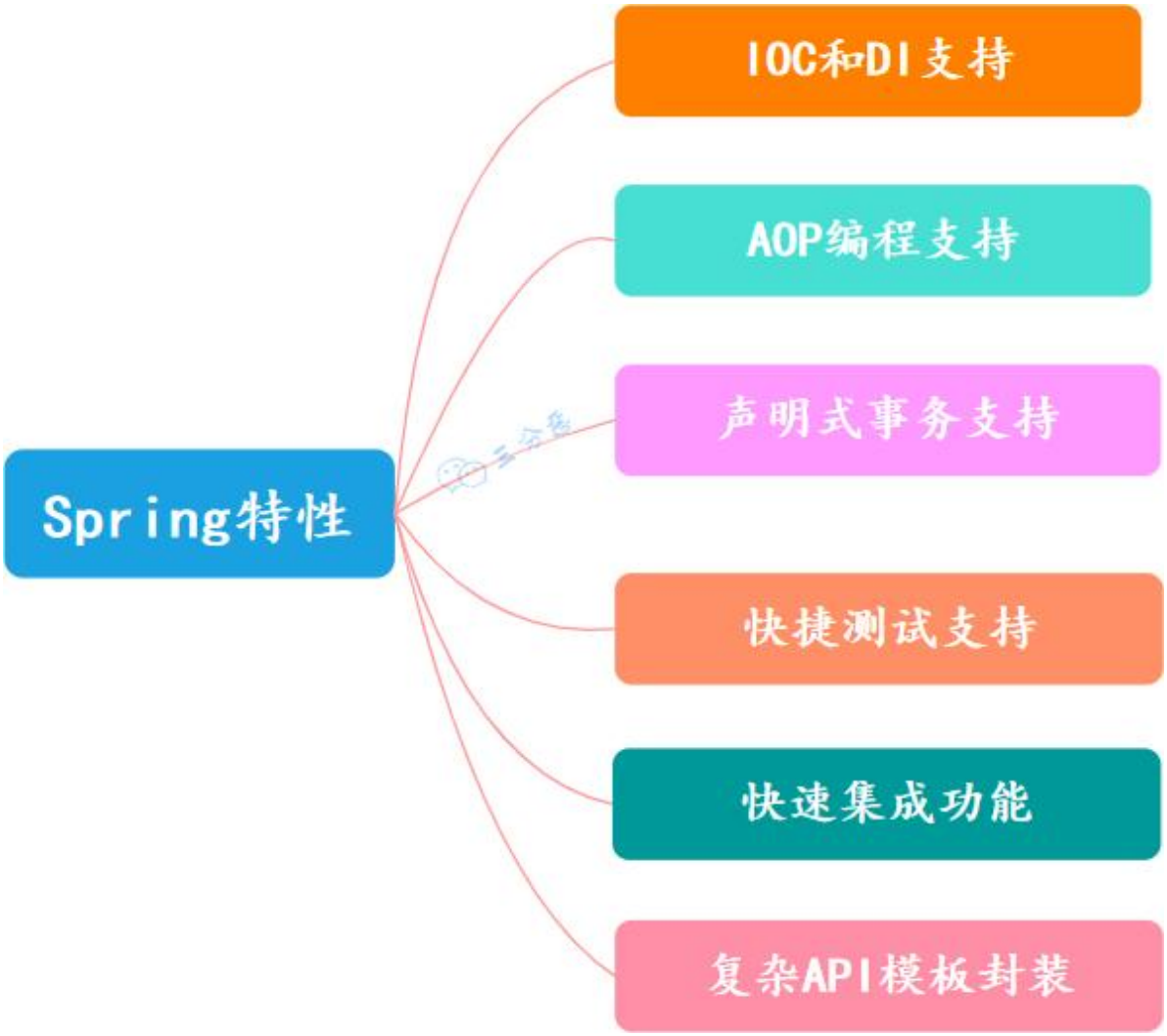
2003 年，一个音乐家 Rod Johnson 决定发展一个轻量级的 Java 开发框架， Spring作为 Java 战场的龙骑兵渐渐崛起，并淘汰了EJB这个传统的重装骑兵。



到了现在，企业级开发的标配基本就是 Spring5 + Spring Boot 2 + JDK 8

> Spring 有哪些特性呢？

Spring 有很多优点：



1. IOC 和 DI 的支持

Spring 的核心就是一个大的工厂容器，可以维护所有对象的创建和依赖关系，Spring 工厂用于生成 Bean，并且管理 Bean 的生命周期，实现**高内聚低耦合**的设计理念。

2. AOP 编程的支持

Spring 提供了**面向切面编程**，可以方便的实现对程序进行权限拦截、运行监控等切面功能。

3. 声明式事务的支持

支持通过配置就来完成对事务的管理，而不需要通过硬编码的方式，以前重复的一些事务提交、回滚的 JDBC 代码，都可以不用自己写了。

4. 快捷测试的支持

Spring 对 Junit 提供支持，可以通过**注解**快捷地测试 Spring 程序。

5. 快速集成功能

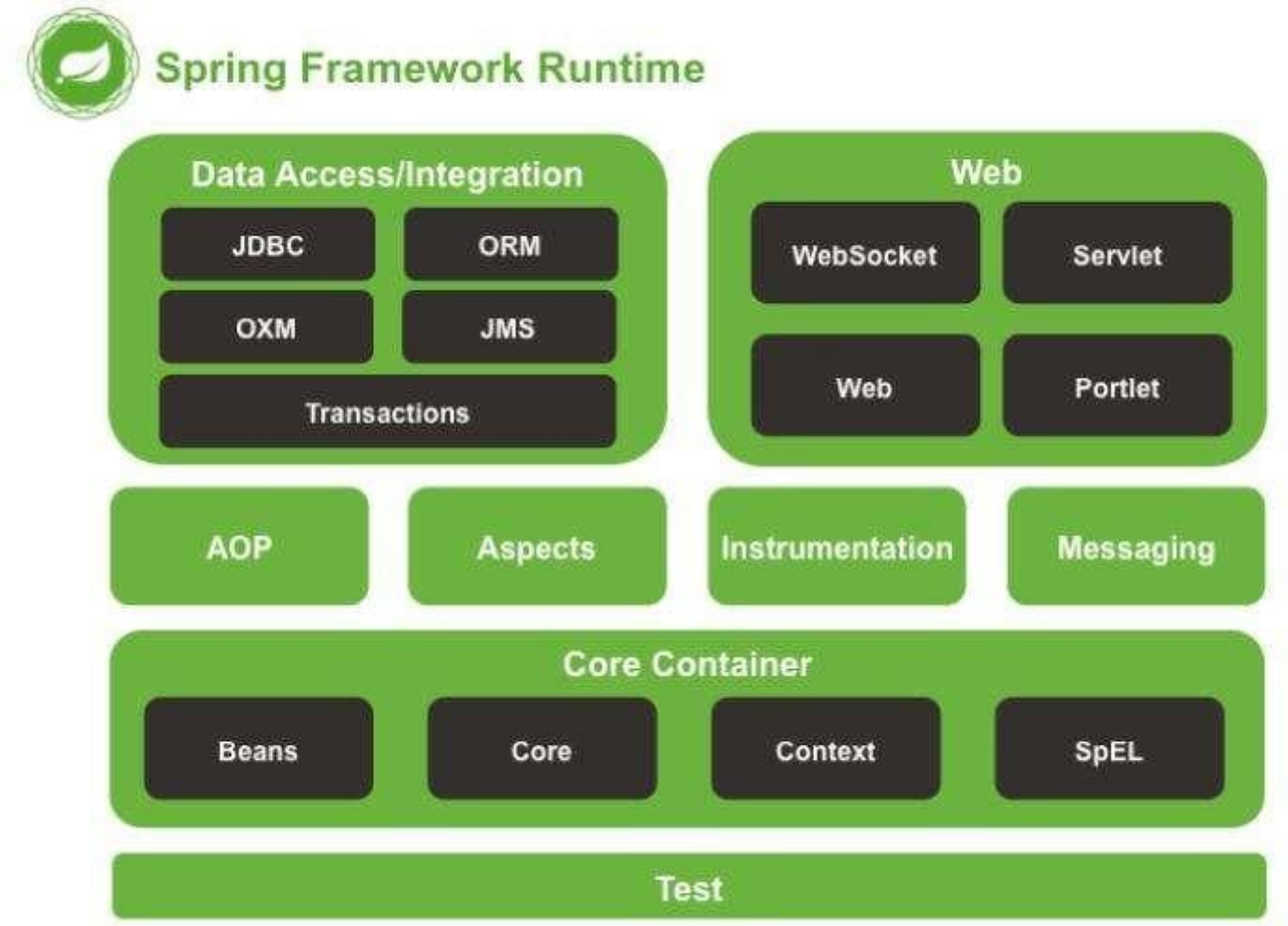
方便集成各种优秀框架，Spring 不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz 等）的直接支持。

6. 复杂 API 模板封装

Spring 对 JavaEE 开发中非常难用的一些 API（JDBC、JavaMail、远程调用等）都提供了模板化的封装，这些封装 API 的提供使得应用难度大大降低。

2.Spring 有哪些模块呢？

Spring 框架是分模块存在，除了最核心的Spring Core Container是必要模块之外，其他模块都是可选 ，大约有 20 多个模块。

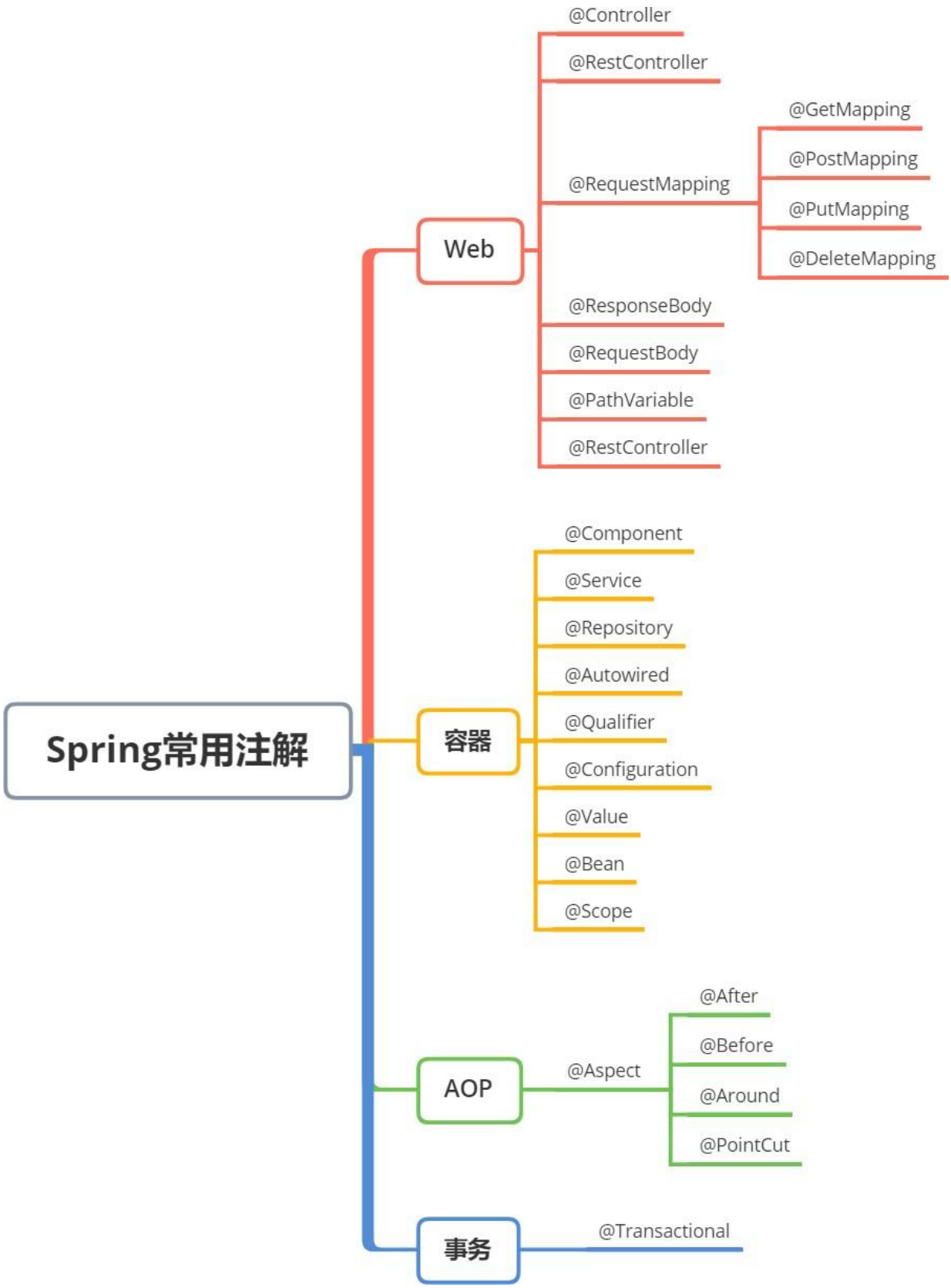


最主要的七大模块：

1. Spring Core：Spring 核心，它是框架最基础的部分，提供 IOC 和依赖注入 DI 特性。
2. Spring Context：Spring 上下文容器，它是 BeanFactory 功能加强的一个子接口。
3. Spring Web：它提供 Web 应用开发的支持。
4. Spring MVC：它针对 Web 应用中 MVC 思想的实现。
5. Spring DAO：提供对 JDBC 抽象层，简化了 JDBC 编码，同时，编码更具有健壮性。
6. Spring ORM：它支持用于流行的 ORM 框架的整合，比如：Spring + Hibernate、Spring + iBatis、Spring + JDO 的整合等。
7. Spring AOP：即面向切面编程，它提供了与 AOP 联盟兼容的编程实现。

3.Spring 有哪些常用注解呢？

Spring 有很多模块，甚至广义的 SpringBoot、SpringCloud 也算是 Spring 的一部分，我们来分模块，按功能来看一下一些常用的注解：



Web:

- @Controller：组合注解（组合了@Component 注解），应用在 MVC 层（控制层）。

- @RestController：该注解为一个组合注解，相当于@Controller 和@ResponseBody 的组合，注解在类上，意味着，该 Controller 的所有方法都默认加上了@ResponseBody。
- @RequestMapping：用于映射 Web 请求，包括访问路径和参数。如果是 Restful 风格接口，还可以根据请求类型使用不同的注解：
  - @GetMapping @PostMapping
  - @PutMapping @DeleteMapping
  - @ResponseBody：支持将返回值放在 response 内，而不是一个页面，通常用户返回 json 数据。
- @RequestBody：允许 request 的参数在 request 体中，而不是在直接连接在地址后面。
- @PathVariable：用于接收路径参数，比如 @RequestMapping(“/hello/{name}”)申明的路径，将注解放在参数中前，即可获取该值，通常作为 Restful 的接口实现方法。
- @RestController：该注解为一个组合注解，相当于@Controller 和@ResponseBody 的组合，注解在类上，意味着，该 Controller 的所有方法都默认加上了@ResponseBody。

容器

- @Component：表示一个带注释的类是一个“组件”，成为 Spring 管理的 Bean。当使用基于注解的配置和类路径扫描时，这些类被视为自动检测的候选对象。同时@Component 还是一个元注解。 @Service：组合注解（组合了@Component 注解），应用在 service 层（业务逻辑层）。 @Repository：组合注解（组合了@Component 注解），应用在 dao 层（数据访问层）。 @Autowired：Spring 提供的工具（由 Spring 的依赖注入工具（BeanPostProcessor、 BeanFactoryPostProcessor）自动注入）。
- @Qualifier：该注解通常跟 @Autowired 一起使用，当想对注入的过程做更多的控制，@Qualifier 可帮助配置，比如两个以上相同类型的 Bean 时 Spring 无法抉择，用到此注解 @Configuration：声明当前类是一个配置类（相当于一个 Spring 配置的 xml 文件）

  `#{}` `\${}`

- @Value：可用在字段，构造器参数跟方法参数，指定一个默认值，支持 两个方式。一般将 SpringbBoot 中的 application.properties 配置的属性值赋值给变量。
- @Bean：注解在方法上，声明当前方法的返回值为一个 Bean。返回的 Bean 对应的类中可以定义 init()方法和 destroy()方法，然后在 @Bean(initMethod=”init”,destroyMethod=”destroy”)定义，在构造之后执行 init，在销毁之前执行 destroy。
- @Scope:定义我们采用什么模式去创建 Bean（方法上，得有@Bean） 其设置类型包括：Singleton 、Prototype、Request 、 Session、GlobalSession。

AOP:

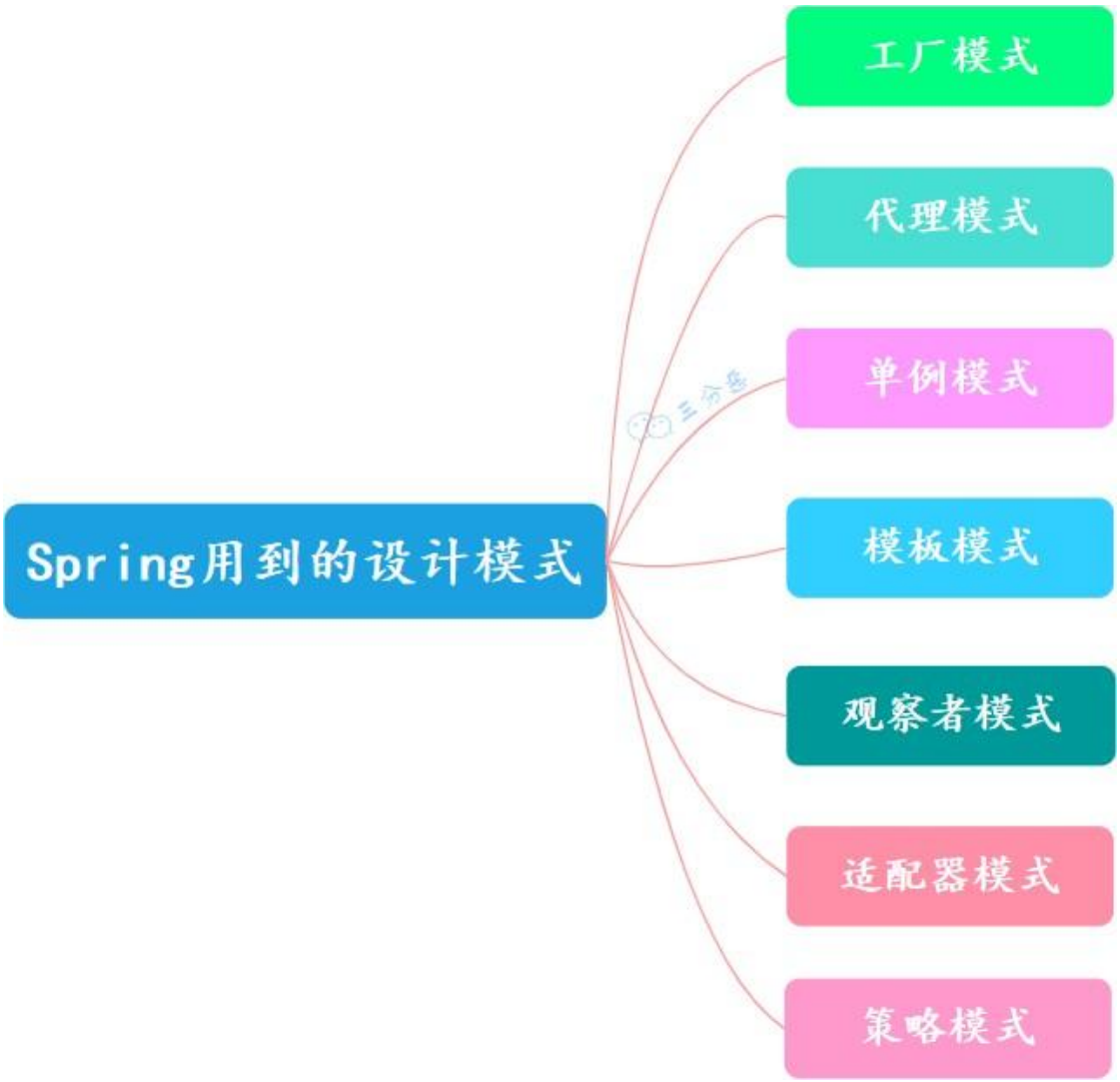
- @Aspect:声明一个切面（类上） 使用@After、@Before、@Around 定义建言（advice），可直接将拦截规则（切点）作为参数。
  - `@After` ：在方法执行之后执行（方法上）。
  - `@Before：` 在方法执行之前执行（方法上）。
  - `@Around：` 在方法执行之前与之后执行（方法上）。
  - `@PointCut：` 声明切点 在 java 配置类中使用@EnableAspectJAutoProxy 注解开启 Spring 对 AspectJ 代理的支持（类上）。

事务：

- @Transactional：在要开启事务的方法上使用@Transactional 注解，即可声明式开启事务。

4.Spring 中应用了哪些设计模式呢？

Spring 框架中广泛使用了不同类型的设计模式，下面我们来看看到底有哪些设计模式?



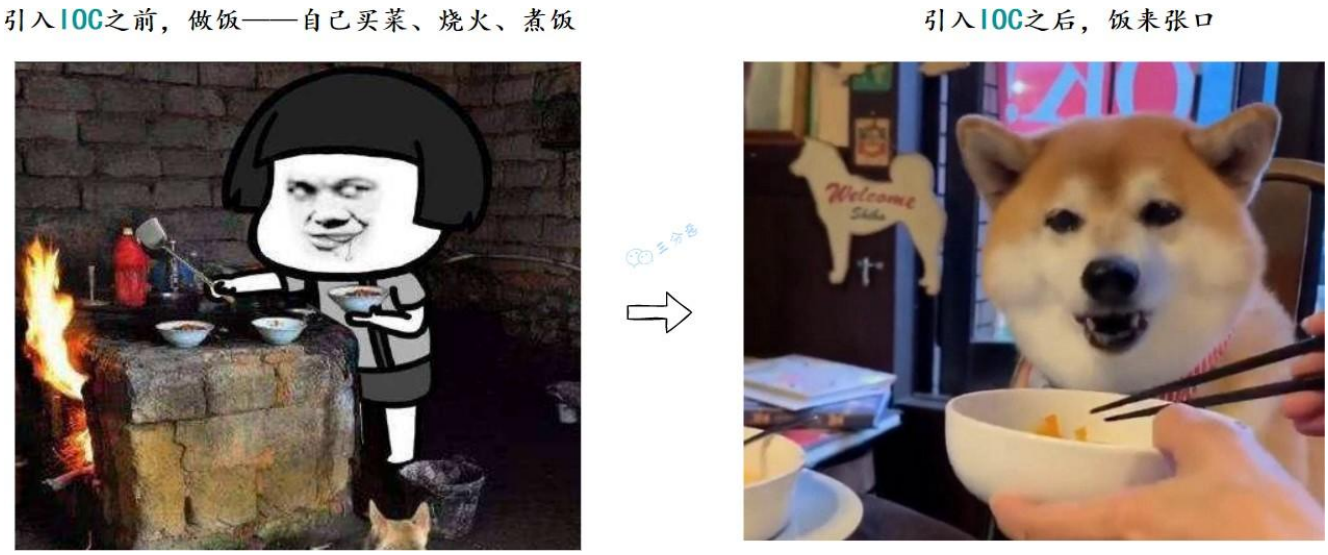
1. **工厂模式**：Spring 容器本质是一个大工厂，使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象。
2. **代理模式**：Spring AOP 功能功能就是通过代理模式来实现的，分为动态代理和静态代理。
3. **单例模式**：Spring 中的 Bean 默认都是单例的，这样有利于容器对 Bean 的管理。
4. **模板模式**：Spring 中 JdbcTemplate、RestTemplate 等以 Template 结尾的对数据库、网络等等进行操作的模板类，就使用到了模板模式。
5. **观察者模式**: Spring 事件驱动模型就是观察者模式很经典的一个应用。
6. **适配器模式**：Spring AOP 的增强或通知(Advice)使用到了适配器模式、Spring MVC 中也是用到了适配器模式适配 Controller。
7. **策略模式**：Spring 中有一个 Resource 接口，它的不同实现类，会根据不同的策略去访问资源。

IOC

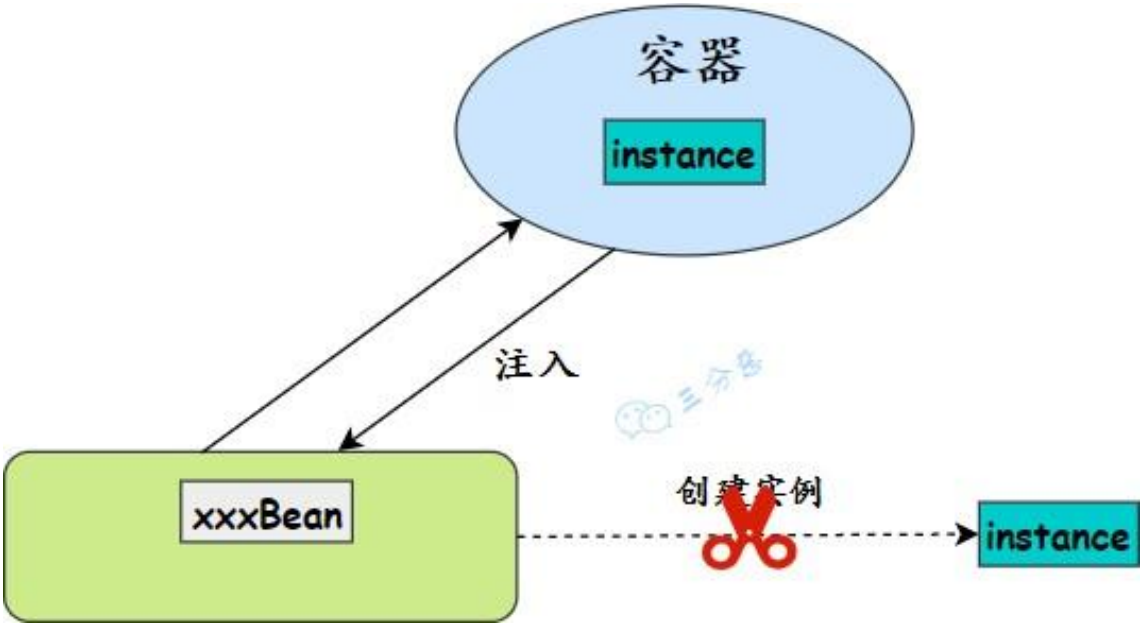
5.说一说什么是 IOC？什么是 DI?

Java 是面向对象的编程语言，一个个实例对象相互合作组成了业务逻辑，原来，我们都是在代码里创建对象和对象的依赖。

所谓的IOC（控制反转）：就是由容器来负责控制对象的生命周期和对象间的关系。以前是我们想要什么，就自己创建什么，现在是我们需要什么，容器就给我们送来什么。



也就是说，控制对象生命周期的不再是引用它的对象，而是容器。对具体对象，以前是它控制其它对象，现在所有对象都被容器控制，所以这就叫**控制反转**。



DI（**依赖注入**）：指的是容器在实例化对象的时候把它依赖的类注入给它。有的说法 IOC 和 DI 是一回事，有的说法是 IOC 是思想，DI 是 IOC 的实现。

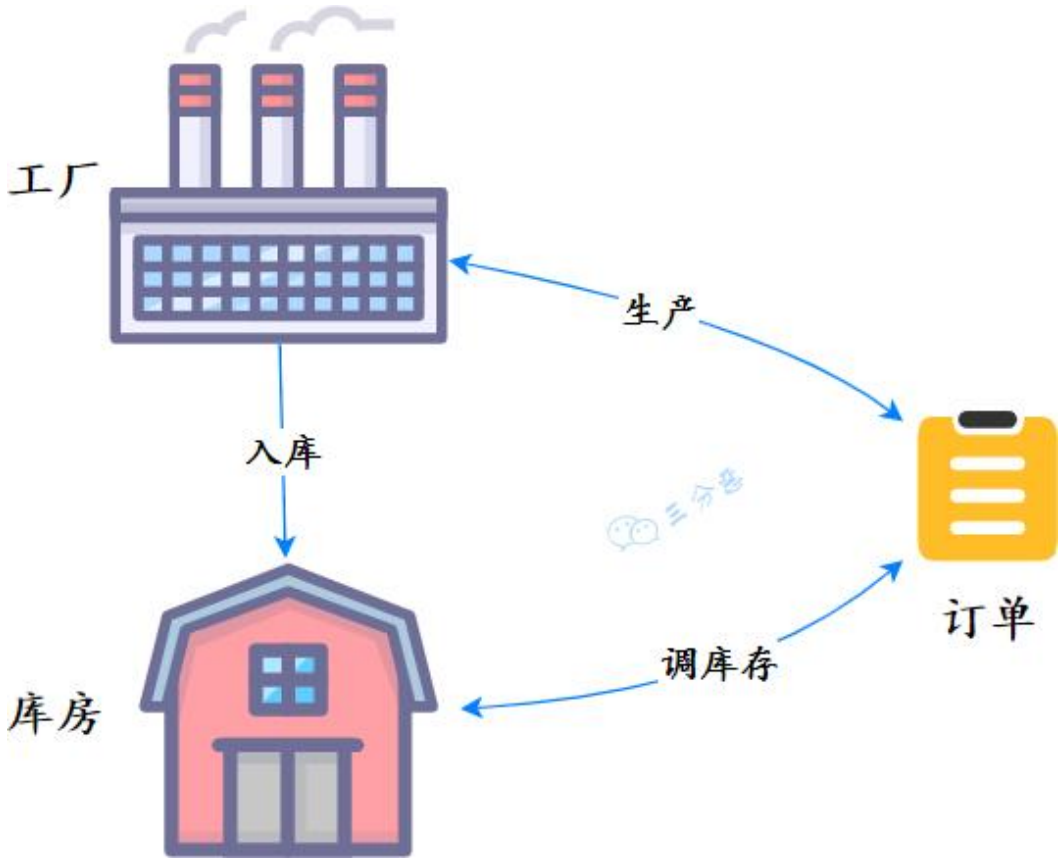
> 为什么要使用 IOC 呢？

最主要的是两个字**解耦**，硬编码会造成对象间的过度耦合，使用 IOC 之后，我们可以不用关心对象间的依赖，专心开发应用就行。

6.能简单说一下 Spring IOC 的实现机制吗？

PS:这道题老三在面试中被问到过，问法是“**你有自己实现过简单的** Spring **吗？**”

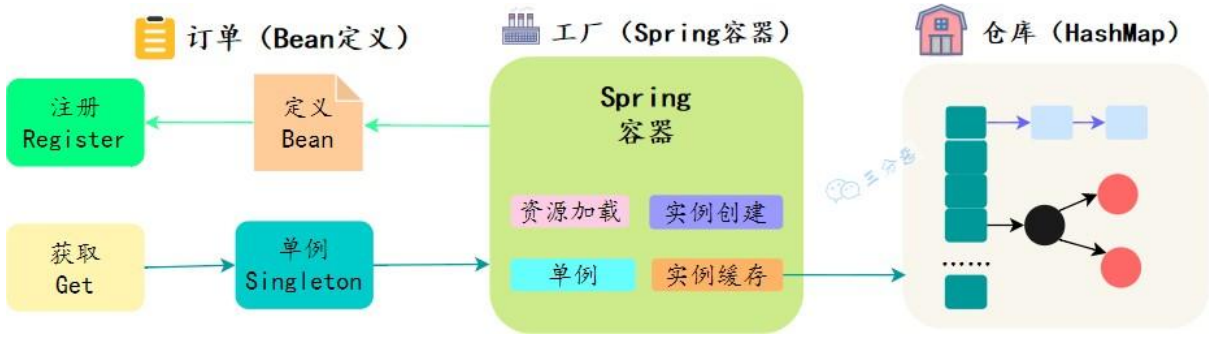
Spring 的 IOC 本质就是一个大工厂，我们想想一个工厂是怎么运行的呢？



- **生产产品**：一个工厂最核心的功能就是生产产品。在 Spring 里，不用 Bean 自己来实例化，而是交给 Spring，应该怎么实现呢？——答案毫无疑问，**反射**。
  那么这个厂子的生产管理是怎么做的？你应该也知道——**工厂模式**。
- **库存产品**：工厂一般都是有库房的，用来库存产品，毕竟生产的产品不能立马就拉走。Spring 我们都知道是一个容器，这个容器里存的就是对象，不能每次来取对象，都得现场来反射创建对象，得把创建出的对象存起来。
- **订单处理**：还有最重要的一点，工厂根据什么来提供产品呢？订单。这些订单可能五花八门，有线上签签的、有到工厂签的、还有工厂销售上门签的……最后经过处理，指导工厂的出货。

  在 Spring 里，也有这样的订单，它就是我们 bean 的定义和依赖关系，可以是 xml 形式，也可以是我们最熟悉的注解形式。

我们简单地实现一个 mini 版的 Spring IOC：



Bean **定义：**

Bean　通过一个配置文件定义，把它解析成一个类型。

- beans.properties

  偷懒，这里直接用了最方便解析的 properties，这里直接用一个 <key,value>类型的配置来代表 Bean 的定义，其中 key 是 beanName，value 是 class

  ```
  userDao:cn.fighter3.bean.UserDao
  ```

- BeanDefinition.java

  bean 定义类，配置文件中 bean 定义对应的实体

  ```
  public class BeanDefinition {

      private String beanName;

      private Class beanClass;
       //省略getter、setter
   }
  ```

- ResourceLoader.java

  资源加载器，用来完成配置文件中配置的加载

  ```
  public class ResourceLoader {


      public static Map<String, BeanDefinition> getResource() {
          Map<String, BeanDefinition> beanDefinitionMap = new HashMap<>(16); Properties properties = new Properties();
          try {
              InputStream inputStream = ResourceLoader.class.getResourceAsStream("/beans.properties");
              properties.load(inputStream);
              Iterator<String> it = properties.stringPropertyNames().iterator(); while (it.hasNext()) {
                  String key = it.next();
                  String className = properties.getProperty(key); BeanDefinition beanDefinition = new
                  BeanDefinition(); beanDefinition.setBeanName(key);
                  Class clazz = Class.forName(className);
                  beanDefinition.setBeanClass(clazz); beanDefinitionMap.put(key,
                  beanDefinition);
              }
              inputStream.close();
          } catch (IOException | ClassNotFoundException e) { e.printStackTrace();
          }
  ```

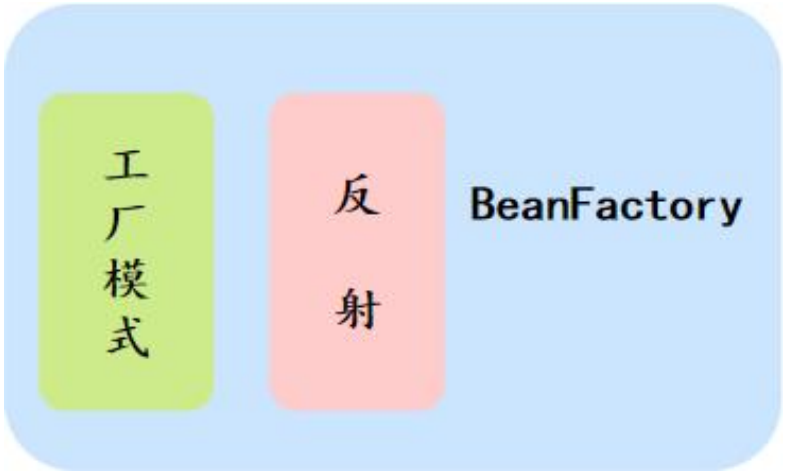
```
        return beanDefinitionMap;
    }

}
```

- BeanRegister.java

  对象注册器，这里用于单例bean 的缓存，我们大幅简化，默认所有 bean 都是单例的。可以看到所谓单例注册，也很简单，不过是往 HashMap 里存对象。

```
public class BeanRegister {

    //单例Bean缓存
    private Map<String, Object> singletonMap = new HashMap<>(32);

    /**
     * 获取单例Bean
     *
     * @param beanName bean名称
     * @return
     */
    public Object getSingletonBean(String beanName)
        { return singletonMap.get(beanName);
    }

    /**
     * 注  单例bean
     *
     * @param beanName
     * @param bean
     */
    public void registerSingletonBean(String beanName, Object bean)
        { if (singletonMap.containsKey(beanName)) {
            return;
        }
        singletonMap.put(beanName, bean);
    }

}
```

- BeanFactory.java



- 对象工厂，我们最**核心**的一个类，在它初始化的时候，创建了bean 注册器，完成了资源的加载。获
- 取 bean 的时候，先从单例缓存中取，如果没有取到，就创建并注册一个 bean

```
public class BeanFactory {

    private Map<String, BeanDefinition> beanDefinitionMap = new HashMap<>();

    private BeanRegister beanRegister;

    public BeanFactory() {
        //创建bean注 器
        beanRegister = new BeanRegister();
        //加载资源
        this.beanDefinitionMap = new ResourceLoader().getResource();
    }

    /**
     * 获取bean
     *
     * @param beanName bean名称
     * @return
     */
    public Object getBean(String beanName) {
        //  bean缓存中取
        Object bean = beanRegister.getSingletonBean(beanName);
        if (bean != null) {
            return bean;
        }
        //根据bean定义，创建bean
        return createBean(beanDefinitionMap.get(beanName));
    }

    /**
     * 创建Bean
```

```
     *
     * @param beanDefinition bean定义
     * @return
     */
    private Object createBean(BeanDefinition beanDefinition)
        { try {
            Object bean = beanDefinition.getBeanClass().newInstance();
            //缓存bean
            beanRegister.registerSingletonBean(beanDefinition.getBeanName(), bean);
            return bean;
        } catch (InstantiationException | IllegalAccessException e)
            { e.printStackTrace();
        }
        return null;
    }
}
```

- 测试
  - UserDao.java

    我们的 Bean 类，很简单

    ```
    public class UserDao {

        public void queryUserInfo(){
            System.out.println("A good man.");
        }
    }
    ```

  - 单元测试

    ```
    public class ApiTest
        { @Test
        public void test_BeanFactory() {
            //1.创建bean工  (同时完成了加载资源、创建注  单例bean注  器的操作)
            BeanFactory beanFactory = new BeanFactory();

            //2.第一次获取bean (通过反射创建bean，缓存bean)
            UserDao userDao1 = (UserDao) beanFactory.getBean("userDao");
            userDao1.queryUserInfo();

            //3.第二次获取bean (  缓存中获取bean)
            UserDao userDao2 = (UserDao) beanFactory.getBean("userDao");
            userDao2.queryUserInfo();
        }
    }
    ```

  - 运行结果

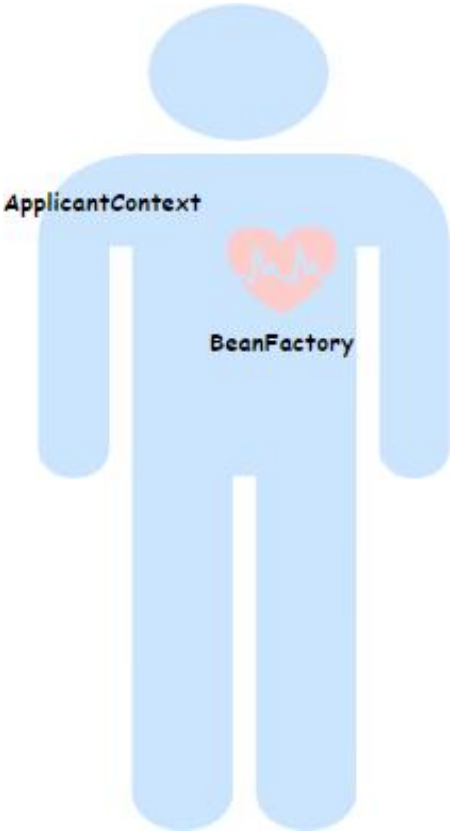
```
A good man.
A good man.
```

至此，我们一个乞丐+破船版的 Spring 就完成了，代码也比较完整，有条件的可以跑一下。

PS:因为时间+篇幅的限制，这个 demo 比较简陋，没有面向接口、没有解耦、边界检查、异常处理……健壮性、扩展性都有很大的不足，感兴趣可以学习参考[15]。

7.说说 BeanFactory 和 ApplicantContext?

可以这么形容，BeanFactory 是 Spring 的“心脏”，ApplicantContext 是完整的“身躯”。

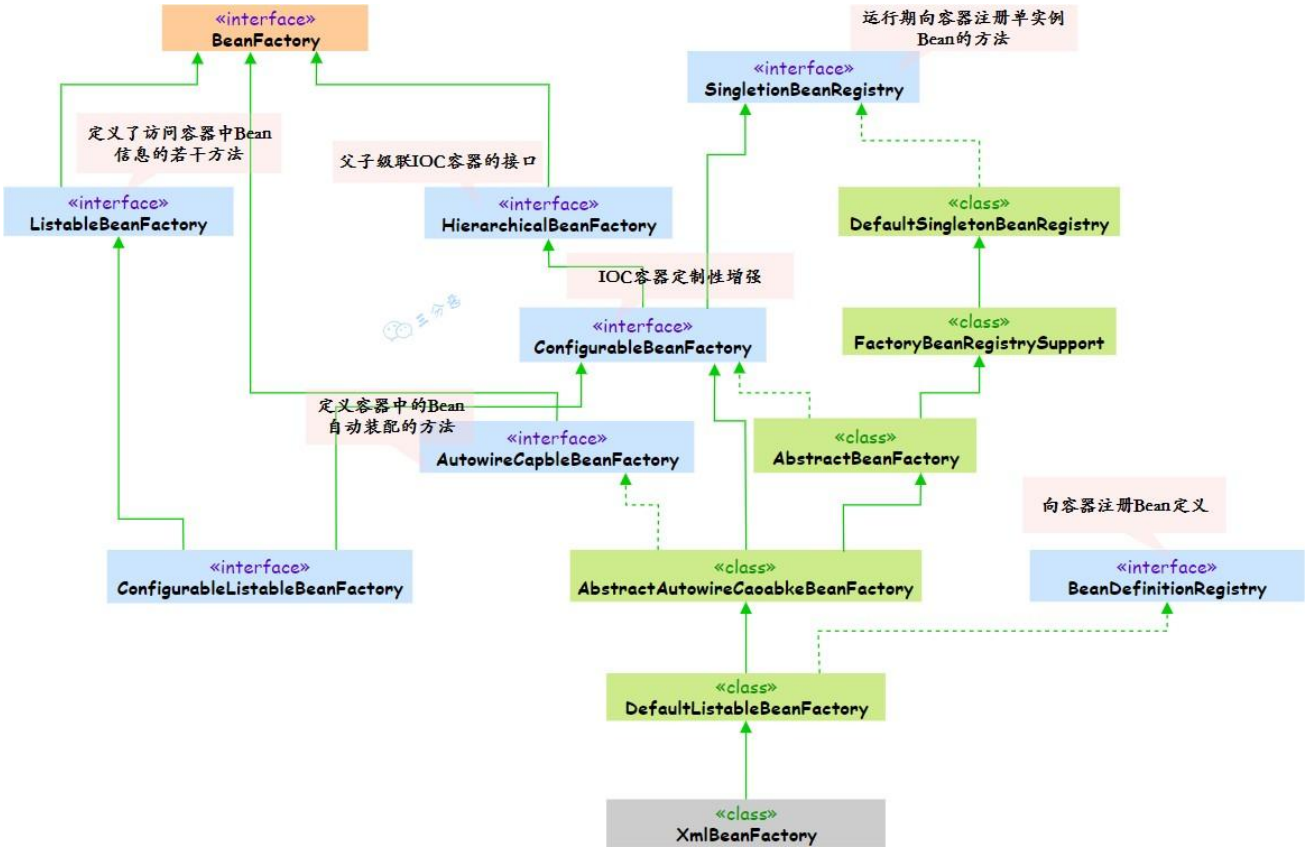


- BeanFactory（Bean 工厂）是 Spring 框架的基础设施，面向 Spring 本身。
- ApplicantContext（应用上下文）建立在 BeanFactoty 基础上，面向使用 Spring 框架的开发者。

BeanFactory 接口

BeanFactory 是类的通用工厂，可以创建并管理各种类的对象。

Spring 为 BeanFactory 提供了很多种实现，最常用的是 XmlBeanFactory，但在 Spring 3.2 中已被废弃，建议使用 XmlBeanDefinitionReader、DefaultListableBeanFactory。



BeanFactory 接口位于类结构树的顶端，它最主要的方法就是 getBean(String var1)，这个方法从容器中返回特定名称的 Bean。

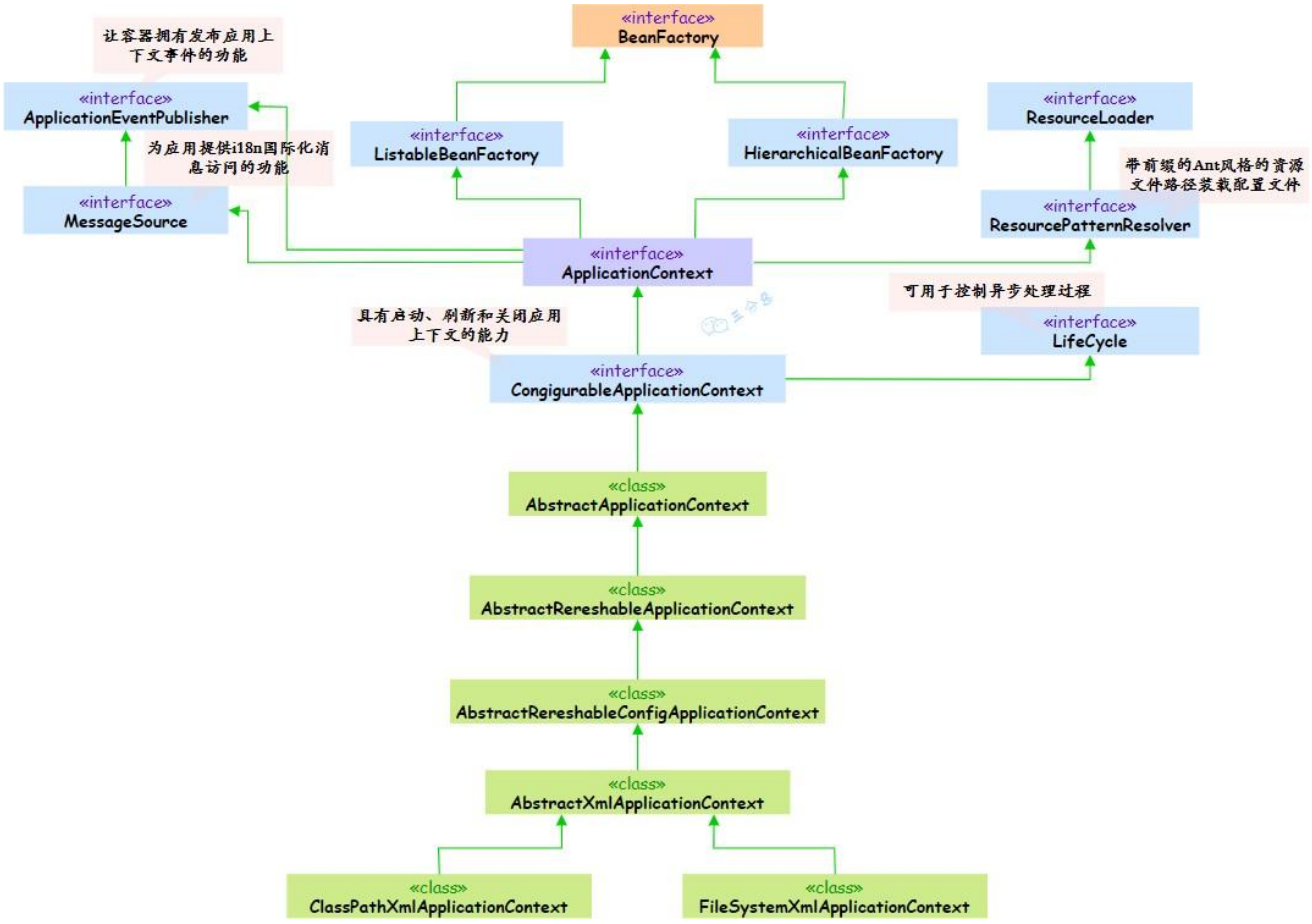
BeanFactory 的功能通过其它的接口得到了不断的扩展，比如 AbstractAutowireCapableBeanFactory 定义了将容器中的 Bean 按照某种规则（比如按名字匹配、按类型匹配等）进行自动装配的方法。

这里看一个 XMLBeanFactory（已过期） 获取 bean 的例子：

```
public class HelloWorldApp{
   public static void main(String[] args) {
      BeanFactory factory = new XmlBeanFactory (new ClassPathResource("beans.xml"));
      HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");
      obj.getMessage();
   }
}
```

ApplicationContext 接口

ApplicationContext 由 BeanFactory 派生而来，提供了更多面向实际应用的功能。可以这么说，使用 BeanFactory 就是手动档，使用 ApplicationContext 就是自动档。



ApplicationContext 继承了HierachicalBeanFactory 和 ListableBeanFactory 接口，在此基础上，还通过其他的接口扩展了BeanFactory 的功能，包括：

- Bean instantiation/wiring
- Bean 的实例化/串联
- 自动的 BeanPostProcessor 注册
- 自动的 BeanFactoryPostProcessor 注册方便
- 的 MessageSource 访问（i18n）
- ApplicationEvent 的发布与 BeanFactory 懒加载的方式不同，它是预加载，所以，每一个 bean 都在 ApplicationContext 启动之后实例化这

是 ApplicationContext 的使用例子：

```
public class HelloWorldApp{
   public static void main(String[] args) {
      ApplicationContext context=new ClassPathXmlApplicationContext("beans.xml");
      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
      obj.getMessage();
   }
}
```

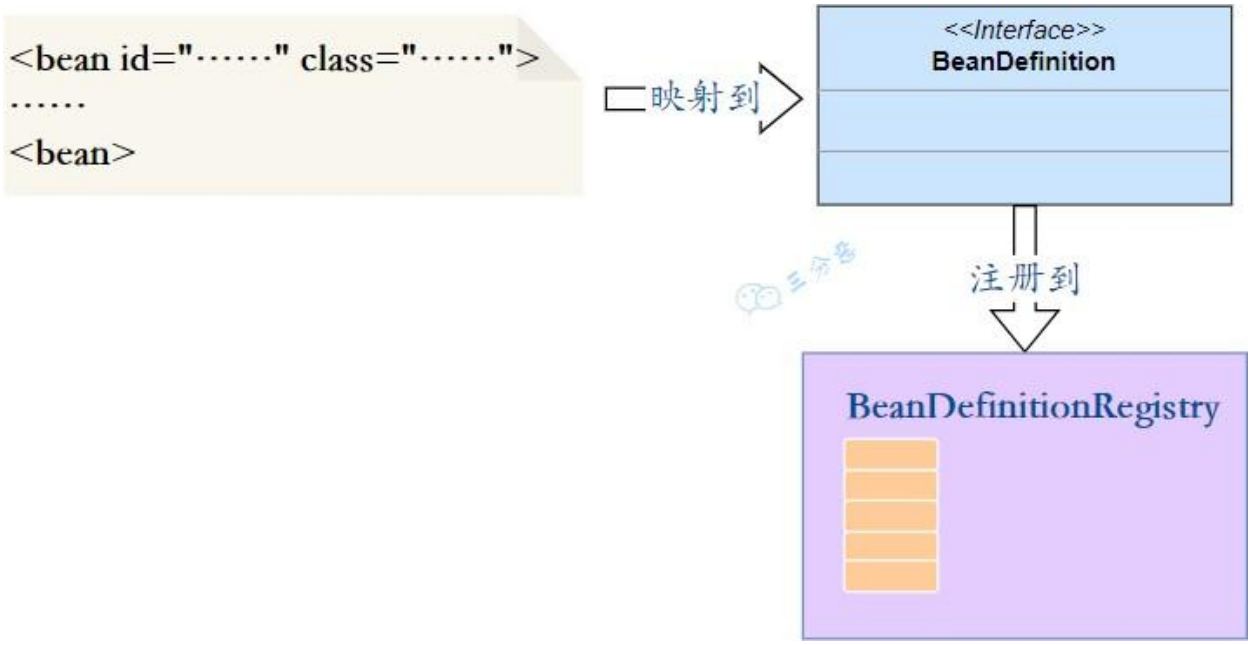
ApplicationContext 包含 BeanFactory 的所有特性，通常推荐使用前者。

8.你知道 Spring 容器启动阶段会干什么吗？

PS：这道题老三面试被问到过

Spring 的 IOC 容器工作的过程，其实可以划分为两个阶段：**容器启动阶段**和Bean **实例化阶段**。其中容器启动阶段主要做的工作是加载和解析配置文件，保存到对应的 Bean 定义中。

容器启动开始，首先会通过某种途径加载 Congiguration MetaData，在大部分情况下，容器需要依赖某些工具类（BeanDefinitionReader）对加载的 Congiguration MetaData 进行解析和分析，并将分析后的信息组为相应的 BeanDefinition。



最后把这些保存了 Bean 定义必要信息的 BeanDefinition，注册到相应的 BeanDefinitionRegistry，这样容器启动就完成了。

9.能说一下 Spring Bean 生命周期吗？

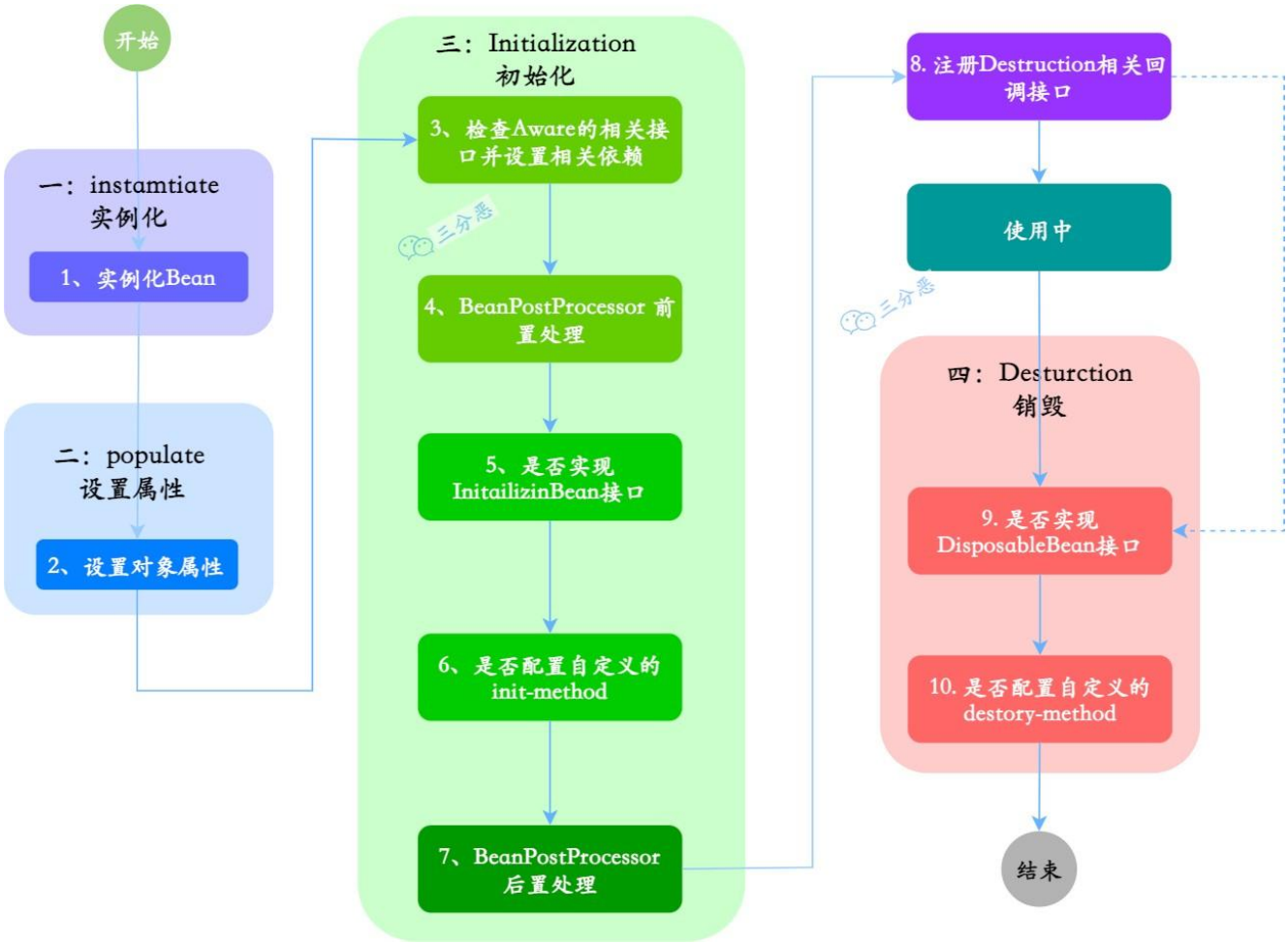
可以看看：Spring Bean 生命周期，好像人的一生。。

在 Spring 中，基本容器BeanFactory 和扩展容器ApplicationContext 的实例化时机不太一样，BeanFactory 采用的是延迟初始化的方式，也就是只有在第一次 getBean()的时候，才会实例化 Bean；ApplicationContext 启动之后会实例化所有的 Bean 定义。

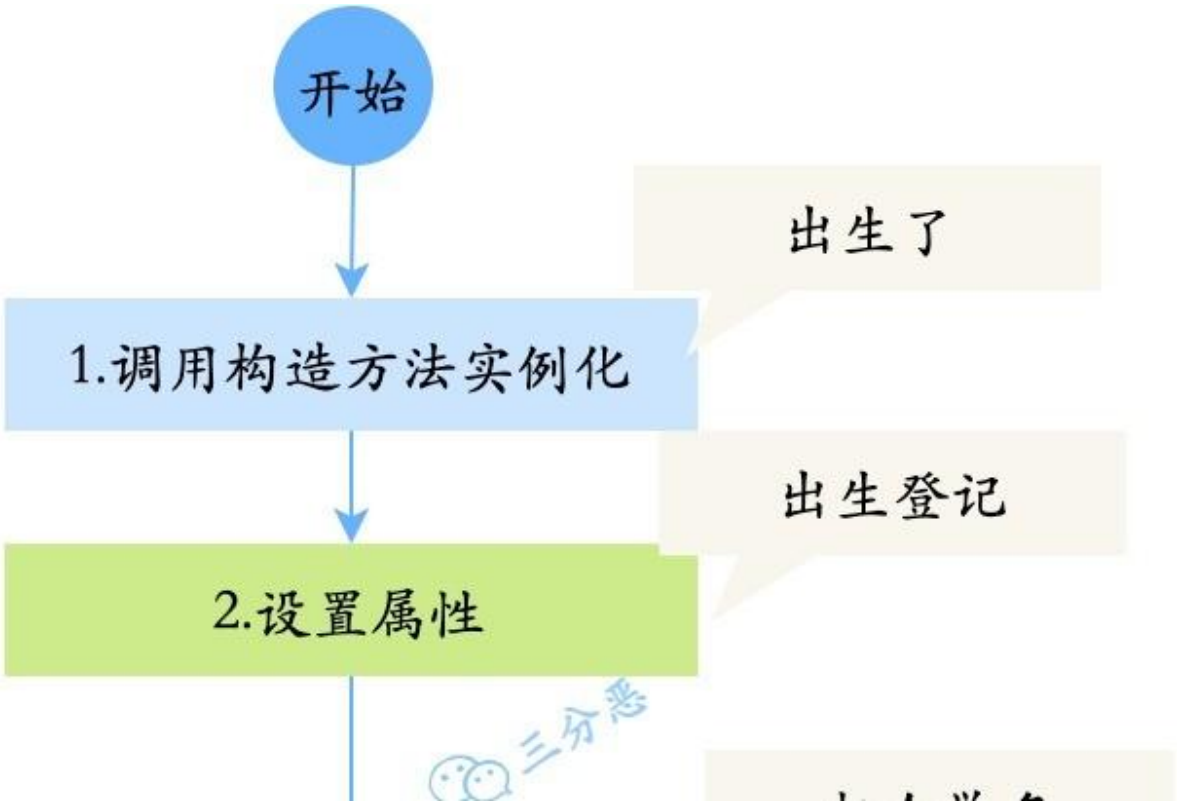
Spring IOC 中 Bean 的生命周期大致分为四个阶段：**实例化**（Instantiation）、**属性赋值**（Populate）、**初始化**（Initialization）、**销毁**（Destruction）。

我们再来看一个稍微详细一些的过程：

- **实例化**：第 1 步，实例化一个 Bean 对象
- **属性赋值**：第 2 步，为 Bean 设置相关属性和依赖
- **初始化**：初始化的阶段的步骤比较多，5、6 步是真正的初始化，第 3、4 步为在初始化前执行，第 7 步在初始化后执行，初始化完成之后，Bean 就可以被使用了
- **销毁**：第 8~10 步，第 8 步其实也可以算到销毁阶段，但不是真正意义上的销毁，而是先在使用前注册了销毁的相关调用接口，为了后面第 9、10 步真正销毁 Bean 时再执行相应的方法



简单总结一下，Bean 生命周期里初始化的过程相对步骤会多一些，比如前置、后置的处理。最后通过一个实例来看一下具体的细节：




起个学名
3.调用BeanNameAware
#setBeanName方法
选个学校
4.调用BeanFactoryAware
#setBeanFactory方法
报名
5.BeanPostProcessor
#postProcessBeforeInitialization方法
入学登记
6.InitializingBean
#afterPropertiesSet方法
努力学习
7.自定义init方法
毕业拿证
8.BeanPostProcessor
#postProcessAfterInitialization方法
工作
Bean使用中

- 定义一个PersonBean类，实现DisposableBean, InitializingBean, BeanFactoryAware, BeanNameAware这 4 个接口，同时还有自定义的init-method和 destroy-method。

```
public class PersonBean implements InitializingBean, BeanFactoryAware, BeanNameAware,
DisposableBean {

    /**
     * 身份证号
     */
    private Integer no;

    /**
     * 姓名
     */
    private String name;

    public PersonBean() {
        System.out.println("1.调用构造方法：我出生了！");
    }

    public Integer getNo() {
        return no;
    }

    public void setNo(Integer no) {
        this.no = no;
    }

    public String getName() {
        return name;
```

```
    }

    public void setName(String name) { this.name = name;
        System.out.println("2.设置属性：我的名字叫"+name);
    }

    @Override
    public void setBeanName(String s) {
        System.out.println("3.调用BeanNameAware#setBeanName方法:我要上学了，起了个学名");
    }

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException { System.out.println("4.调用
        BeanFactoryAware#setBeanFactory方法：选好学校了");
    }

    @Override
    public void afterPropertiesSet() throws Exception { System.out.println("6.InitializingBean#afterPropertiesSet方法：入学登记");
    }

    public void init() {
        System.out.println("7.自定义init方法：努力上学ing");
    }

    @Override
    public void destroy() throws Exception { System.out.println("9.DisposableBean#destroy方法：平淡的一生落幕了");
    }

    public void destroyMethod() {
        System.out.println("10.自定义destroy方法:睡了，别想叫醒我");
    }

    public void work(){
        System.out.println("Bean使用中：工作，只有对社会没有用的人才放假。。");
    }

}
```

- 定义一个`MyBeanPostProcessor`实现`BeanPostProcessor`接口。

```
public class MyBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
```

```
        System.out.println("5.BeanPostProcessor.postProcessBeforeInitialization方法：到学校报名啦");
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("8.BeanPostProcessor#postProcessAfterInitialization方法：终毕业，  到毕业证啦！");
        return bean;
    }
}
```

- 配置文件，指定init-method和 destroy-method属性

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="myBeanPostProcessor"
class="cn.fighter3.spring.life.MyBeanPostProcessor" />
    <bean name="personBean" class="cn.fighter3.spring.life.PersonBean"
          init-method="init" destroy-method="destroyMethod">
        <property name="idNo" value= "80669865"/>
        <property name="name" value="张铁钢" />
    </bean>

</beans>
```

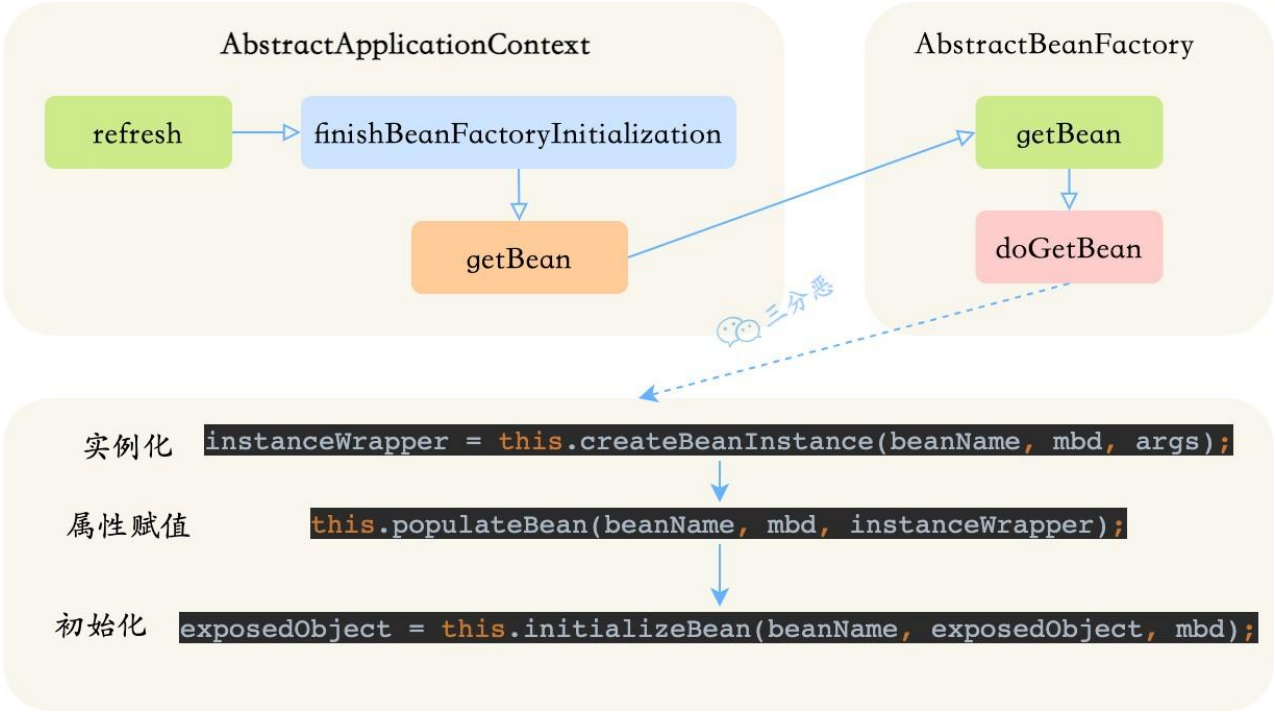
- 测试

```
public class Main {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("spring-config.xml");
        PersonBean personBean = (PersonBean) context.getBean("personBean");
        personBean.work();
        ((ClassPathXmlApplicationContext) context).destroy();
    }
}
```

- 运行结果：

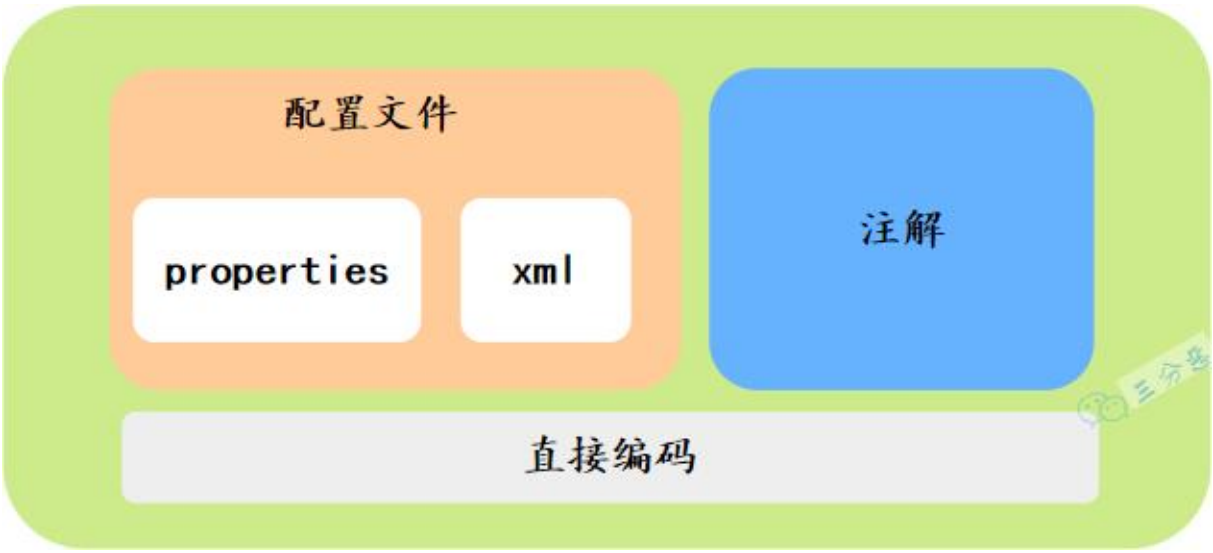
```
1. 调用构造方法：我出生了！
2. 设置属性：我的名字叫张铁钢
3. 调用BeanNameAware#setBeanName方法:我要上学了，起了 学名
4. 调用BeanFactoryAware#setBeanFactory方法：选好学校了
5. BeanPostProcessor#postProcessBeforeInitialization方法：到学校报名啦
6. InitializingBean#afterPropertiesSet方法：入学登记
7. 自定义init方法：努力上学ing
8. BeanPostProcessor#postProcessAfterInitialization方法：终 毕业， 到毕业证啦！
Bean使用中：工作，只有对社会没有用的人才放 。。
9. DisposableBean#destroy方法：平淡的一生落幕了
10. 自定义destroy方法:睡了，别想叫醒我
```

关于源码，Bean 创建过程可以查看`AbstractBeanFactory#doGetBean`方法，在这个方法里可以看到 Bean 的实例化，赋值、初始化的过程，至于最终的销毁，可以看看`ConfigurableApplicationContext#close()`。



10. Bean 定义和依赖定义有哪些方式？

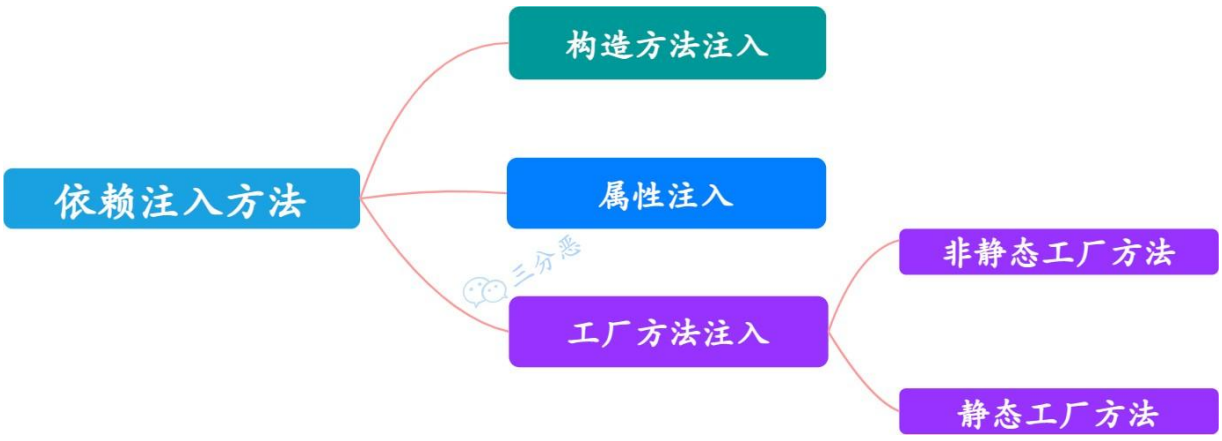
有三种方式：**直接编码方式、配置文件方式、注解方式**。



- 直接编码方式：我们一般接触不到直接编码的方式，但其实其它的方式最终都要通过直接编码来实现。
- 配置文件方式：通过 xml、propreties 类型的配置文件，配置相应的依赖关系，Spring 读取配置文件，完成依赖关系的注入。
- 注解方式：注解方式应该是我们用的最多的一种方式了，在相应的地方使用注解修饰，Spring 会扫描注解，完成依赖关系的注入。

11.有哪些依赖注入的方法？

Spring 支持**构造方法注入**、**属性注入**、**工厂方法注入**,其中工厂方法注入，又可以分为**静态工厂方法注入**和**非静态工厂方法注入**。



- **构造方法注入**

  通过调用类的构造方法，将接口实现类通过构造方法变量传入

```
public CatDaoImpl(String
  message){ this. message = message;
}
```

```
<bean id="CatDaoImpl" class="com.CatDaoImpl">
  <constructor-arg value=" message "></constructor-arg>
</bean>
```

- **属性注入**

  通过 Setter 方法完成调用类所需依赖的注入

```
 public class Id {
    private int id;

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }
}
```

```
<bean id="id" class="com.id ">
  <property name="id" value="123"></property>
</bean>
```

- **工厂方法注入**
  - **静态工厂注入**

    静态工厂顾名思义，就是通过调用静态工厂的方法来获取自己需要的对象，为了让 Spring 管理所有对象，我们不能直接通过"工程类.静态方法()"来获取对象，而是依然通过 Spring 注入的形式获取：

```
public class DaoFactory { //静态工

   public static final FactoryDao getStaticFactoryDaoImpl(){
      return new StaticFacotryDaoImpl();
   }
}

public class SpringAction {

 //注入对象
 private FactoryDao staticFactoryDao;

 //注入对象的 set 方法
 public void setStaticFactoryDao(FactoryDao staticFactoryDao)
     { this.staticFactoryDao = staticFactoryDao;
 }

}
```

```
//factory-method="getStaticFactoryDaoImpl"指定调用哪  工  方法
 <bean name="springAction" class=" SpringAction" >
   <!--使用静态工  的方法注入对象,对应下面的配置文件-->
   <property name="staticFactoryDao" ref="staticFactoryDao"></property>
 </bean>

 <!--此处获取对象的方式是  工  类中获取静态方法-->
<bean name="staticFactoryDao" class="DaoFactory"
  factory-method="getStaticFactoryDaoImpl"></bean>
```

- 非静态工厂注入

非静态工厂，也叫实例工厂，意思是工厂方法不是静态的，所以我们需要首先 new 一个工厂实例，再调用普通的实例方法。

```
//非静态工
public class DaoFactory {
   public FactoryDao
     getFactoryDaoImpl(){ return new
     FactoryDaoImpl();
   }

public class SpringAction {
  //注入对象
  private FactoryDao factoryDao;

  public void setFactoryDao(FactoryDao factoryDao) {
    this.factoryDao = factoryDao;
  }
}
```

```
 <bean name="springAction" class="SpringAction">
   <!--使用非静态工  的方法注入对象,对应下面的配置文件-->
   <property name="factoryDao" ref="factoryDao"></property>
 </bean>

 <!--此处获取对象的方式是  工  类中获取实例方法-->
 <bean name="daoFactory" class="com.DaoFactory"></bean>

<bean name="factoryDao" factory-bean="daoFactory" factory-
method="getFactoryDaoImpl"></bean>
```

12. Spring 有哪些自动装配的方式？

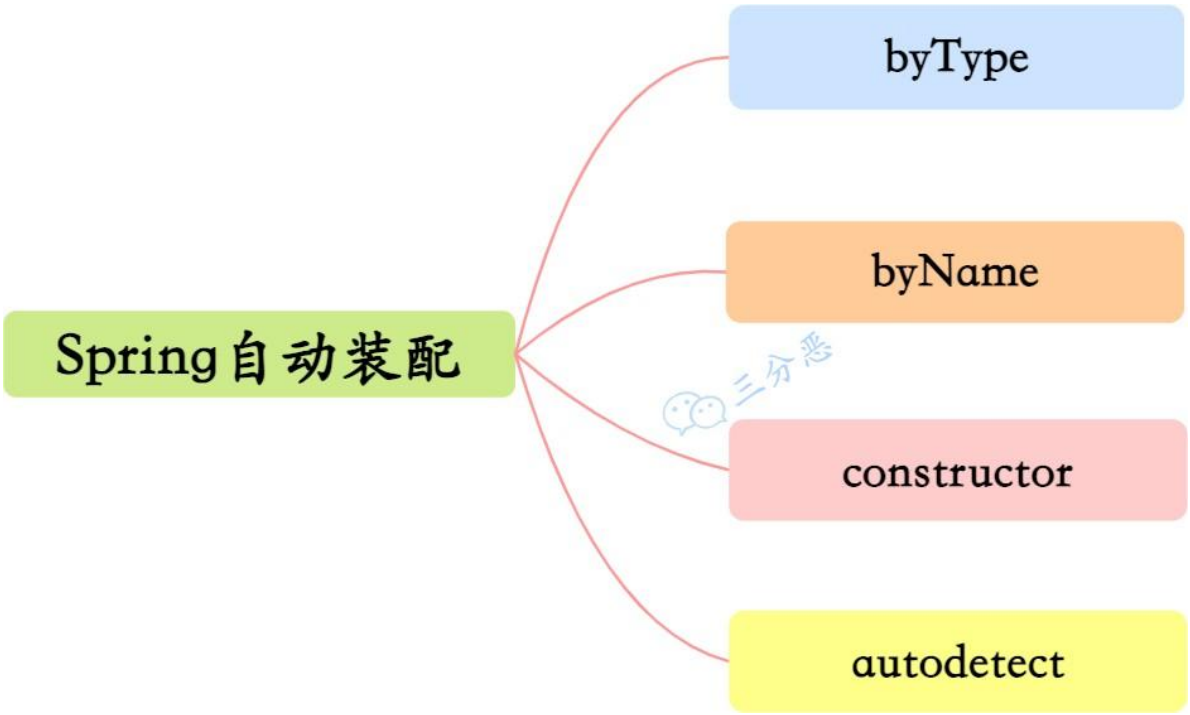
> 什么是自动装配？

Spring IOC 容器知道所有 Bean 的配置信息，此外，通过 Java 反射机制还可以获知实现类的结构信息，如构造方法的结构、属性等信息。掌握所有 Bean 的这些信息后，Spring IOC 容器就可以按照某种规则对容器中的 Bean 进行自动装配，而无须通过显式的方式进行依赖配置。

Spring 提供的这种方式，可以按照某些规则进行 Bean 的自动装配，`<bean>`元素提供了一个指定自动装配类型的属性：`autowire="<自动装配类型>"`

> Spring 提供了哪几种自动装配类型？

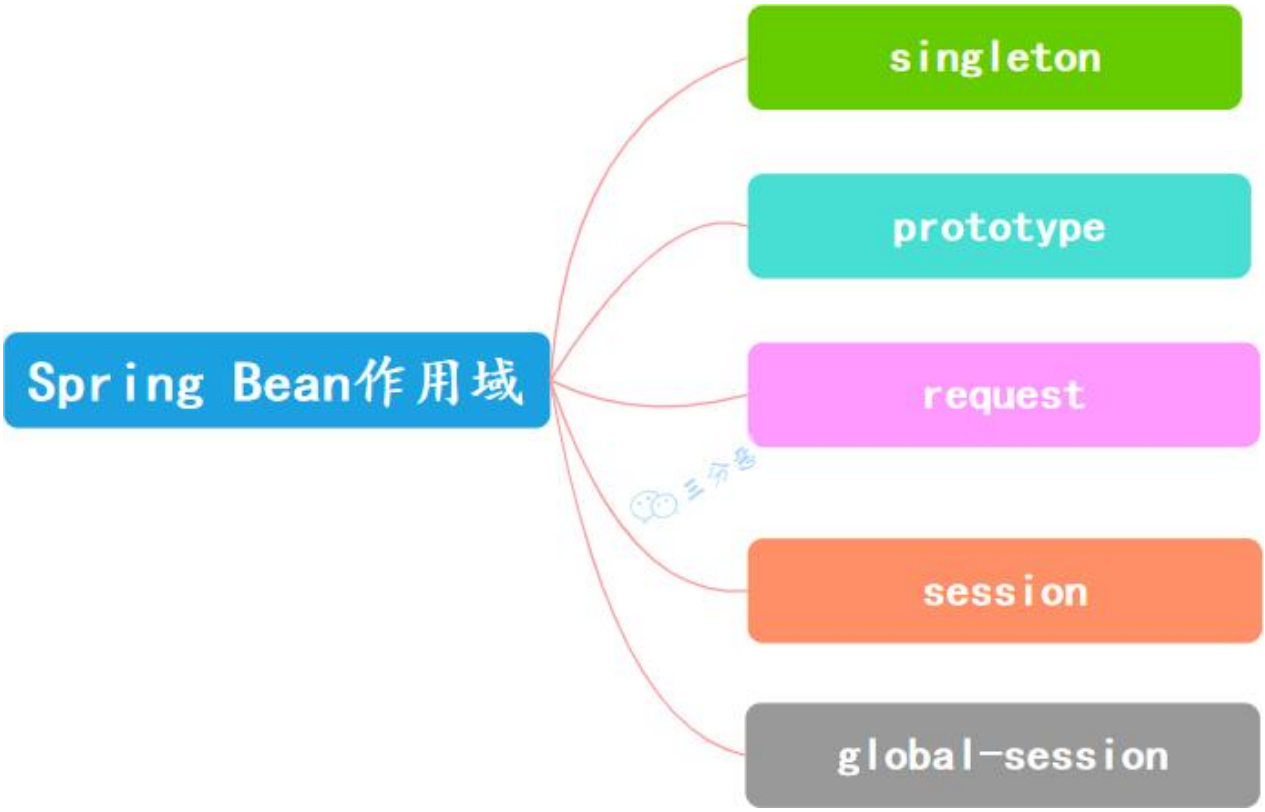
Spring 提供了4 种自动装配类型：



- byName：根据名称进行自动匹配，假设 Boss 又一个名为 car 的属性，如果容器中刚好有一个名为 car 的 bean，Spring 就会自动将其装配给 Boss 的 car 属性
- byType：根据类型进行自动匹配，假设 Boss 有一个 Car 类型的属性，如果容器中刚好有一个 Car 类型的 Bean，Spring 就会自动将其装配给 Boss 这个属性
- constructor：与 byType 类似，只不过它是针对构造函数注入而言的。如果 Boss 有一个构造函数，构造函数包含一个 Car 类型的入参，如果容器中有一个 Car 类型的 Bean，则 Spring 将自动把这个 Bean 作为 Boss构造函数的入参；如果容器中没有找到和构造函数入参匹配类型的 Bean，则 Spring 将抛出异常。 autodetect：根据 Bean 的
- 自省机制决定采用 byType 还是 constructor 进行自动装配，如果 Bean 提供了默认的构造函数，则采用 byType，否则采用 constructor。

13. Spring 中的 Bean 的作用域有哪些?

Spring 的 Bean 主要支持五种作用域：



- singleton : 在 Spring 容器仅存在一个 Bean 实例，Bean 以单实例的方式存在，是 Bean 默认的作用域。
- prototype : 每次从容器重调用 Bean 时，都会返回一个新的实例。以下

三个作用域于只在 Web 应用中适用：

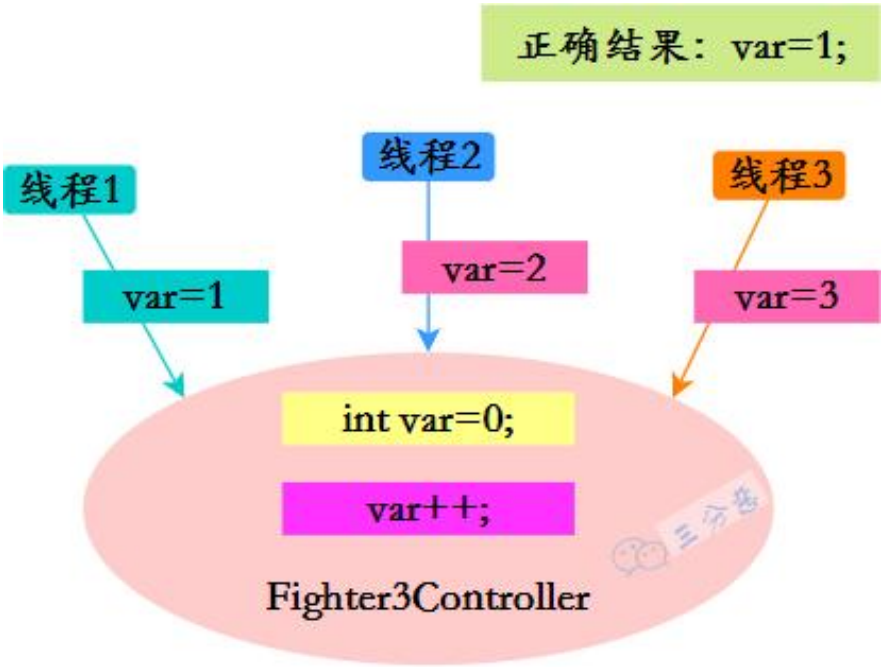
- request : 每一次 HTTP 请求都会产生一个新的 Bean，该 Bean 仅在当前 HTTP Request 内有效。 session : 同一个
- HTTP Session 共享一个 Bean，不同的 HTTP Session 使用不同的 Bean。 globalSession：同一个全局 Session 共享一
- 个 Bean，只用于基于 Protlet 的 Web 应用，Spring5 中已经不存在了。

14. Spring 中的单例 Bean 会存在线程安全问题吗？

首先结论在这：Spring 中的单例 Bean**不是线程安全的**。

因为单例Bean，是全局只有一个 Bean，所有线程共享。如果说单例Bean，是一个无状态的，也就是线程中的操作不会对 Bean 中的成员变量执行**查询**以外的操作，那么这个单例Bean 是线程安全的。比如 Spring mvc 的 Controller、Service、Dao 等，这些 Bean 大多是无状态的，只关注于方法本身。

假如这个 Bean 是有状态的，也就是会对 Bean 中的成员变量进行写操作，那么可能就存在线程安全的问题。



> 单例 Bean 线程安全问题怎么解决呢？

常见的有这么些解决办法：

1. 将 Bean 定义为多例

   这样每一个线程请求过来都会创建一个新的 Bean，但是这样容器就不好管理 Bean，不能这么办。

2. 在 Bean 对象中尽量避免定义可变的成员变量削

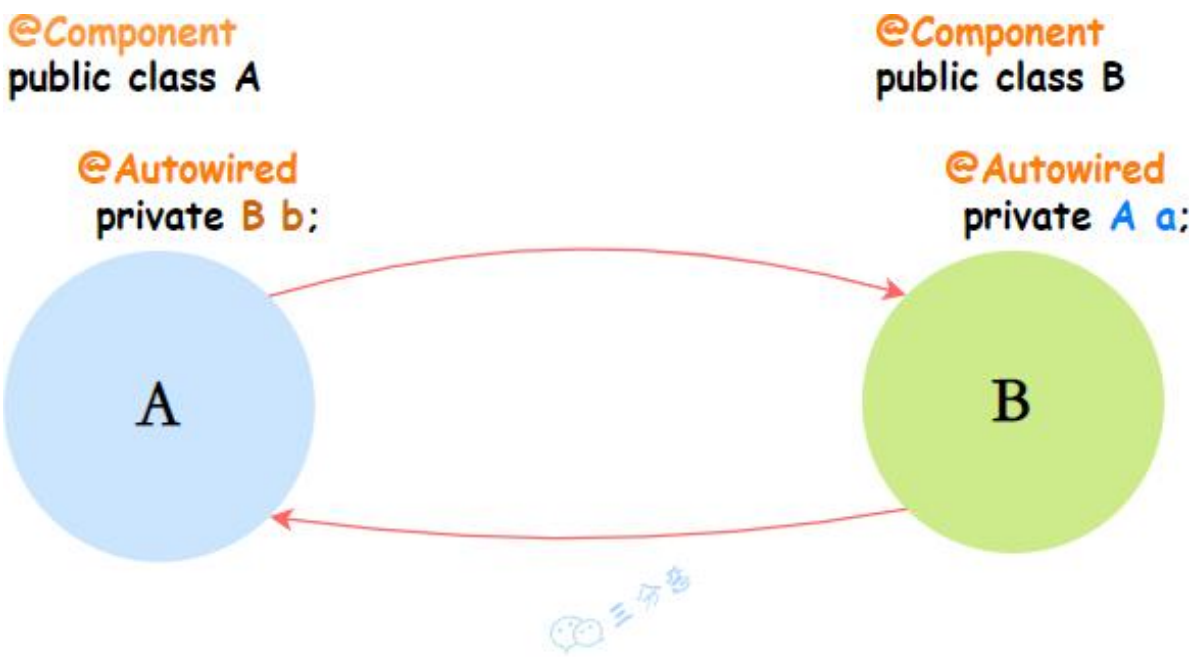
   足适履了属于是，也不能这么干。

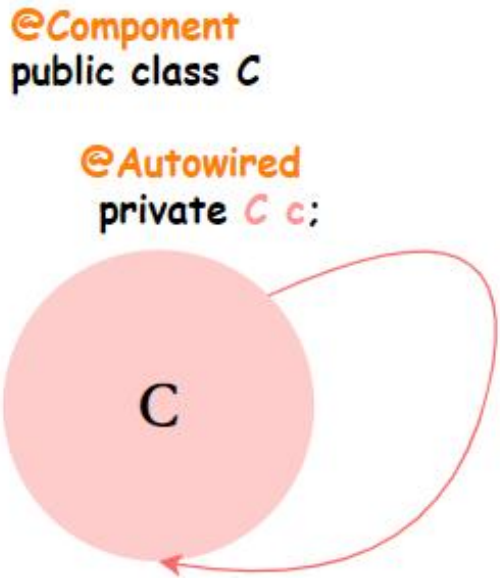
3. 将 Bean 中的成员变量保存在 ThreadLocal 中 ⭐

   我们知道 ThredLoca 能保证多线程下变量的隔离，可以在类中定义一个 ThreadLocal 成员变量，将需要的可变成员变量保存在 ThreadLocal 里，这是推荐的一种方式。

15. 说说循环依赖?

> 什么是循环依赖？





Spring 循环依赖：简单说就是自己依赖自己，或者和别的 Bean 相互依赖。

只有单例的 Bean 才存在循环依赖的情况，**原型**(Prototype)情况下，Spring 会直接抛出异常。原因很简单，AB 循环依赖，A 实例化的时候，发现依赖 B，创建 B 实例，创建 B 的时候发现需要 A，创建 A1 实例……无限套娃，直接把系统干垮。

> Spring 可以解决哪些情况的循环依赖？

Spring 不支持基于构造器注入的循环依赖，但是假如 AB 循环依赖，如果一个是构造器注入，一个是 setter 注入呢？

看看几种情形：

循环依赖

依赖情况	依赖注入方式	是否支持
AB循环依赖	AB均采用构造器注入	✗
AB循环依赖	AB均采用setter方法注入	✓
AB循环依赖	AB均采用属性自动注入	✓
AB循环依赖	A中注入的B为setter注入，B中注入的A为构造器注入	✓
AB循环依赖	B中注入的A为setter注入，A中注入的B为构造器注入	✗

第四种可以而第五种不可以的原因是 Spring 在创建 Bean 时默认会根据自然排序进行创建，所以 A 会先于 B 进行创建。

所以简单总结，当循环依赖的实例都采用 setter 方法注入的时候，Spring 可以支持，都采用构造器注入的时候，不支持，构造器注入和 setter 注入同时存在的时候，看天。

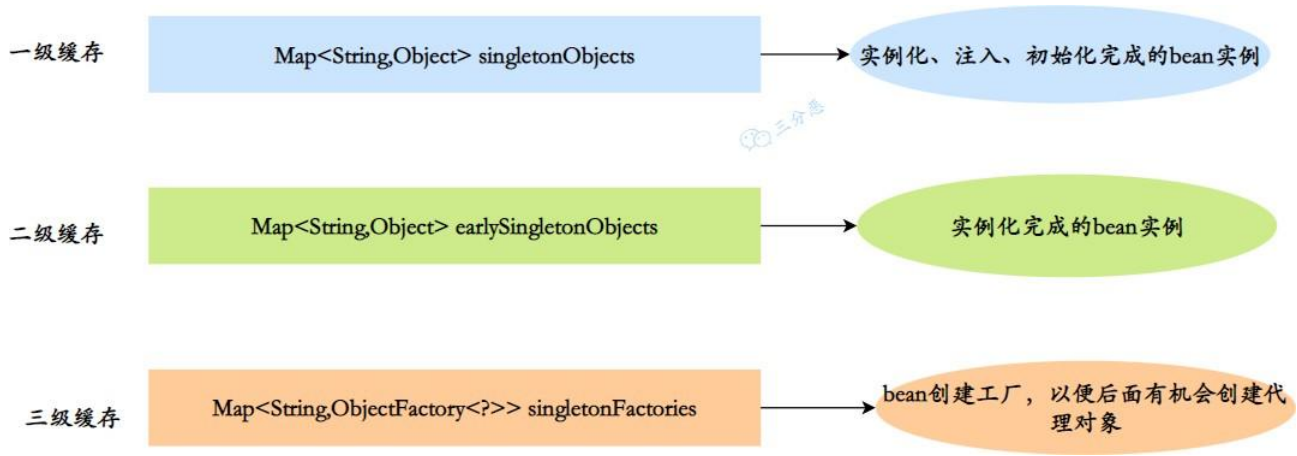
16. 那 Spring 怎么解决循环依赖的呢？

> PS：其实正确答案是开发人员做好设计，别让 Bean 循环依赖，但是没办法，面试官不想听这个。我

们都知道，单例Bean 初始化完成，要经历三步：

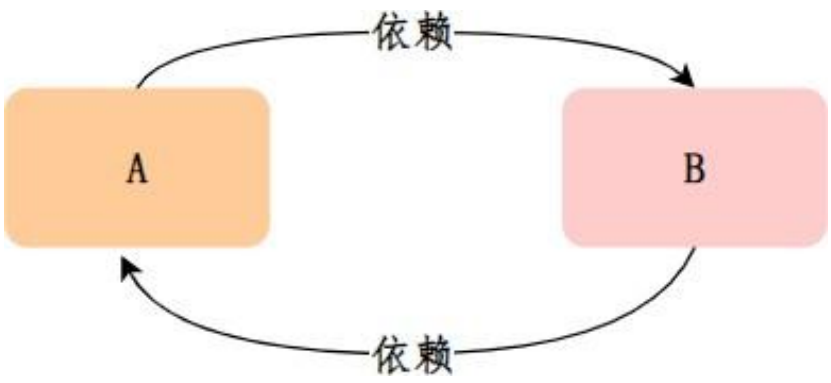
注入就发生在第二步，**属性赋值**，结合这个过程，Spring 通过**三级缓存**解决了循环依赖：

1. 一级缓存：`Map<String,Object>` singletonObjects，单例池，用于保存实例化、属性赋值（注入）、初始化完成的 bean 实例
2. 二级缓存：`Map<String,Object>` earlySingletonObjects，早期曝光对象，用于保存实例化完成的 bean 实例
3. 三级缓存：`Map<String,ObjectFactory<?>>` singletonFactories，早期曝光对象工厂，用于保存 bean 创建工厂，以便于后面扩展有机会创建代理对象。



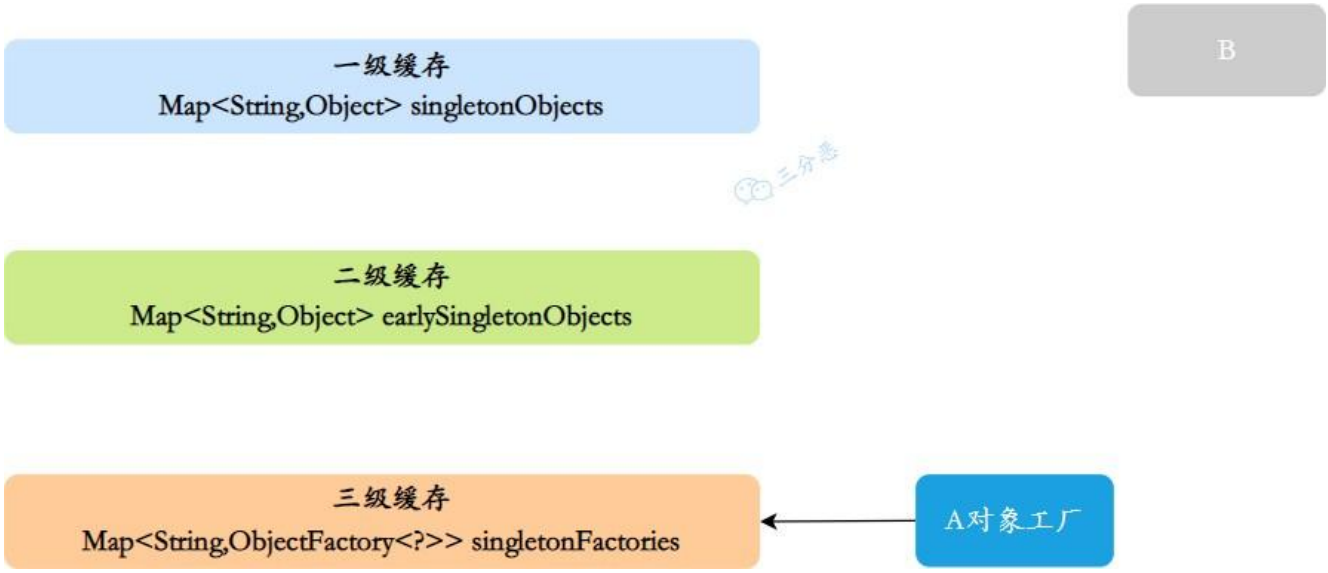
我们来看一下三级缓存解决循环依赖的过程：

当 A、B 两个类发生循环依赖时：

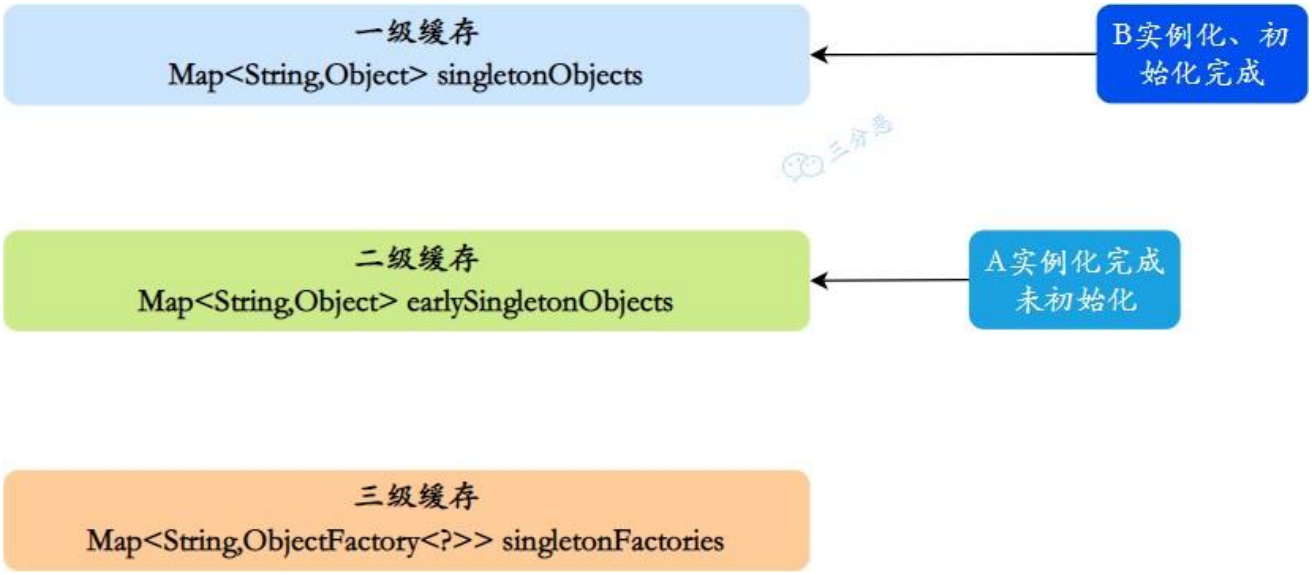


A 实例的初始化过程：

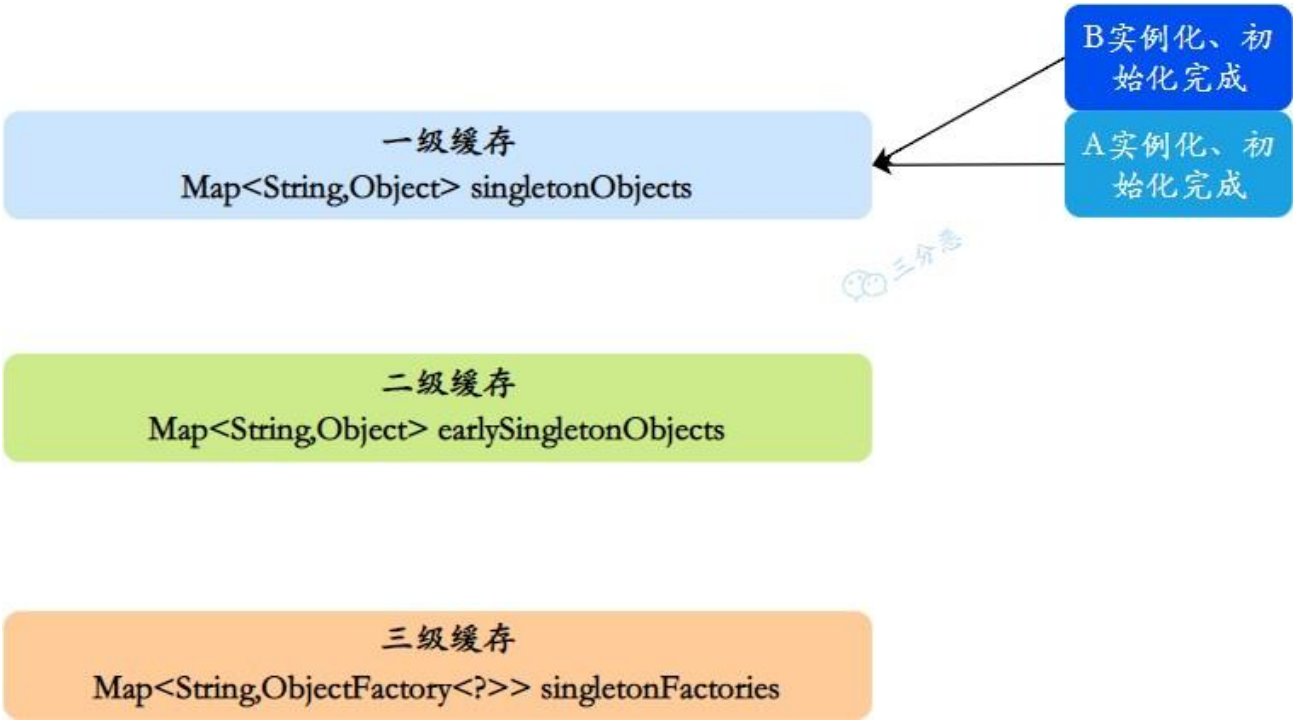
1. 创建 A 实例，实例化的时候把 A 对象工厂放入三级缓存，表示 A 开始实例化了，虽然我这个对象还不完整，但是先曝光出来让大家知道



2. A 注入属性时，发现依赖 B，此时 B 还没有被创建出来，所以去实例化 B
3. 同样，B 注入属性时发现依赖 A，它就会从缓存里找 A 对象。依次从一级到三级缓存查询 A，从三级缓存通过对象工厂拿到 A，发现 A 虽然不太完善，但是存在，把 A 放入二级缓存，同时删除三级缓存中的 A，此时，B 已经实例化并且初始化完成，把 B 放入一级缓存。



4. 接着 A 继续属性赋值，顺利从一级缓存拿到实例化且初始化完成的 B 对象，A 对象创建也完成，删除二级缓存中的 A，同时把 A 放入一级缓存
5. 最后，一级缓存中保存着实例化、初始化都完成的 A、B 对象



所以，我们就知道为什么 Spring 能解决 setter 注入的循环依赖了，因为实例化和属性赋值是分开的，所以里面有操作的空间。如果都是构造器注入的化，那么都得在实例化这一步完成注入，所以自然是无法支持了。

17. 为什么要三级缓存？二级不行吗？

不行，主要是为了**生成代理对象**。如果是没有代理的情况下，使用二级缓存解决循环依赖也是 OK 的。但是如果存在代理，三级没有问题，二级就不行了。

因为三级缓存中放的是生成具体对象的匿名内部类，获取 Object 的时候，它可以生成代理对象，也可以返回普通对象。使用三级缓存主要是为了保证不管什么时候使用的都是一个对象。

假设只有二级缓存的情况，往二级缓存中放的显示一个普通的 Bean 对象，Bean 初始化过程中，通过 BeanPostProcessor 去生成代理对象之后，覆盖掉二级缓存中的普通 Bean 对象，那么可能就导致取到的 Bean 对象不一致了。

18. @Autowired 的实现原理？

实现@Autowired 的关键是：AutowiredAnnotationBeanPostProcessor

在 Bean 的初始化阶段，会通过 Bean 后置处理器来进行一些前置和后置的处理。实

现@Autowired 的功能，也是通过后置处理器来完成的。这个后置处理器就是 AutowiredAnnotationBeanPostProcessor。

- Spring 在创建 bean 的过程中，最终会调用到 doCreateBean()方法，在 doCreateBean()方法中会调用 populateBean()方法，来为 bean 进行属性填充，完成自动装配等工作。

- 在 populateBean()方法中一共调用了两次后置处理器，第一次是为了判断是否需要属性填充，如果不需要进行属性填充，那么就会直接进行return，如果需要进行属性填充，那么方法就会继续向下执行，后面会进行第二次后置处理器的调用，这个时候，就会调用到 AutowiredAnnotationBeanPostProcessor 的 postProcessPropertyValues()方法，在该方法中就会进行@Autowired 注解的解析，然后实现自动装配。

```
/**
* 属性赋值
**/
protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {
                //.....
                if (hasInstAwareBpps) { if (pvs ==
                        null) {
                                pvs = mbd.getPropertyValues();
                        }

                        PropertyValues pvsToUse; for(Iterator var9 =
this.getBeanPostProcessorCache().instantiationAware.iterator(); var9.hasNext(); pvs = pvsToUse) {
                                InstantiationAwareBeanPostProcessor bp = (InstantiationAwareBeanPostProcessor)var9.next();
                                pvsToUse = bp.postProcessProperties((PropertyValues)pvs, bw.getWrappedInstance(), beanName);
                                if (pvsToUse == null) {
                                        if (filteredPds == null) { filteredPds =
this.filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
                                        }
                                        //执行后处理器，填充属性，完成自动装配
                                        //调用InstantiationAwareBeanPostProcessor的
postProcessPropertyValues()方法
                                        pvsToUse = bp.postProcessPropertyValues((PropertyValues)pvs, filteredPds,
bw.getWrappedInstance(), beanName);
                                        if (pvsToUse == null) { return;
                                        }
                                }
                        }
                }
                //.....
    }
```

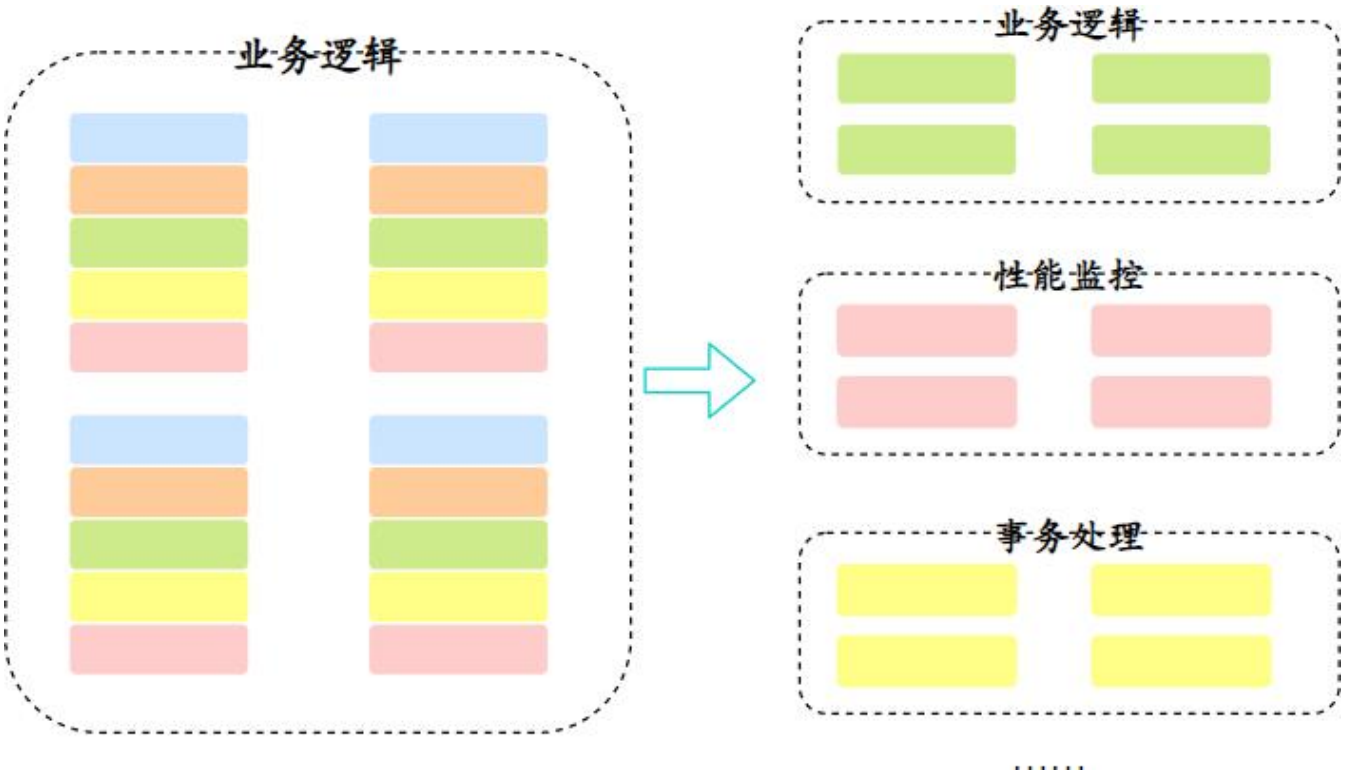
- postProcessorPropertyValues()方法的源码如下，在该方法中，会先调用 findAutowiringMetadata()方法解析出 bean 中带有@Autowired 注解、@Inject 和@Value 注解的属性和方法。然后调用 metadata.inject()方法，进行属性填充。

```
  public PropertyValues postProcessProperties(PropertyValues pvs, Object bean, String
beanName) {
      //@Autowired注解、@Inject和@Value注解的属性和方法
      InjectionMetadata metadata = this.findAutowiringMetadata(beanName,
bean.getClass(), pvs);

      try {
          //属性填充
          metadata.inject(bean, beanName, pvs);
          return pvs;
      } catch (BeanCreationException var6)
          { throw var6;
      } catch (Throwable var7) {
          throw new BeanCreationException(beanName, "Injection of autowired
dependencies failed", var7);
      }
  }
```

AOP

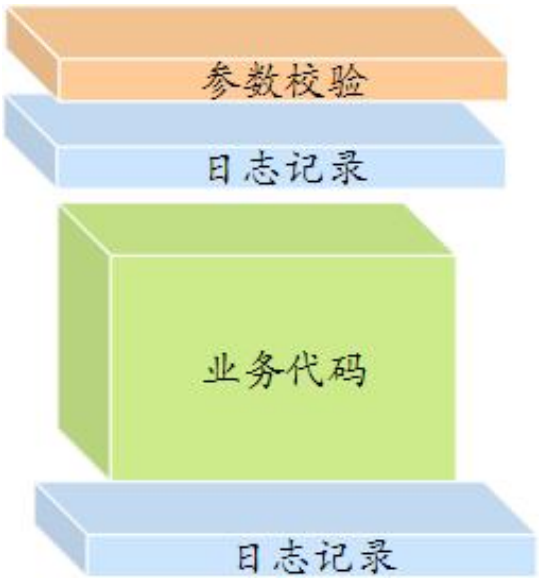
19. 说说什么是 AOP?

AOP：面向切面编程。简单说，就是把一些业务逻辑中的相同的代码抽取到一个独立的模块中，让业务逻辑更加清爽。



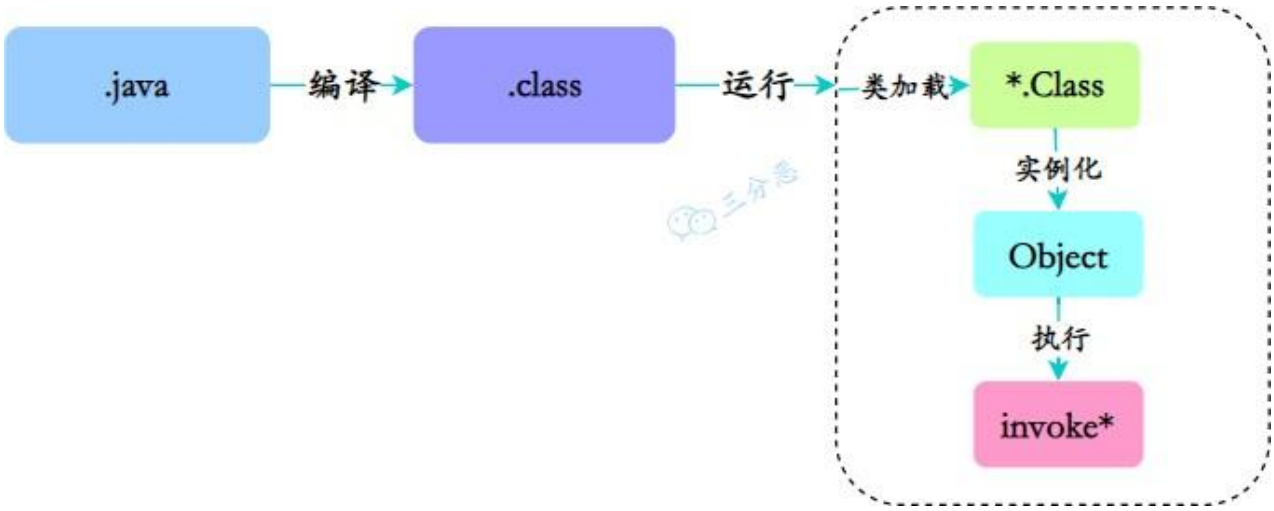
具体来说，假如我现在要 crud 写一堆业务，可是如何业务代码前后前后进行打印日志和参数的校验呢？

我们可以把`日志记录`和`数据校验`可重用的功能模块分离出来，然后在程序的执行的合适的地方动态地植入这些代码并执行。这样就简化了代码的书写。



业务逻辑代码中没有参和通用逻辑的代码，业务模块更简洁，只包含核心业务代码。实现了业务逻辑和通用逻辑的代码分离，便于维护和升级，降低了业务逻辑和通用逻辑的耦合性。

AOP 可以将遍布应用各处的功能分离出来形成可重用的组件。在编译期间、装载期间或运行期间实现在不修改源代码的情况下给程序动态添加功能。从而实现对业务逻辑的隔离，提高代码的模块化能力。



AOP 的核心其实就是**动态代理**，如果是实现了接口的话就会使用 JDK 动态代理，否则使用 CGLIB 代理，主要应用于处理一些具有横切性质的系统级服务，如日志收集、事务管理、安全检查、缓存、对象池管理等。

> AOP **有哪些核心概念？**

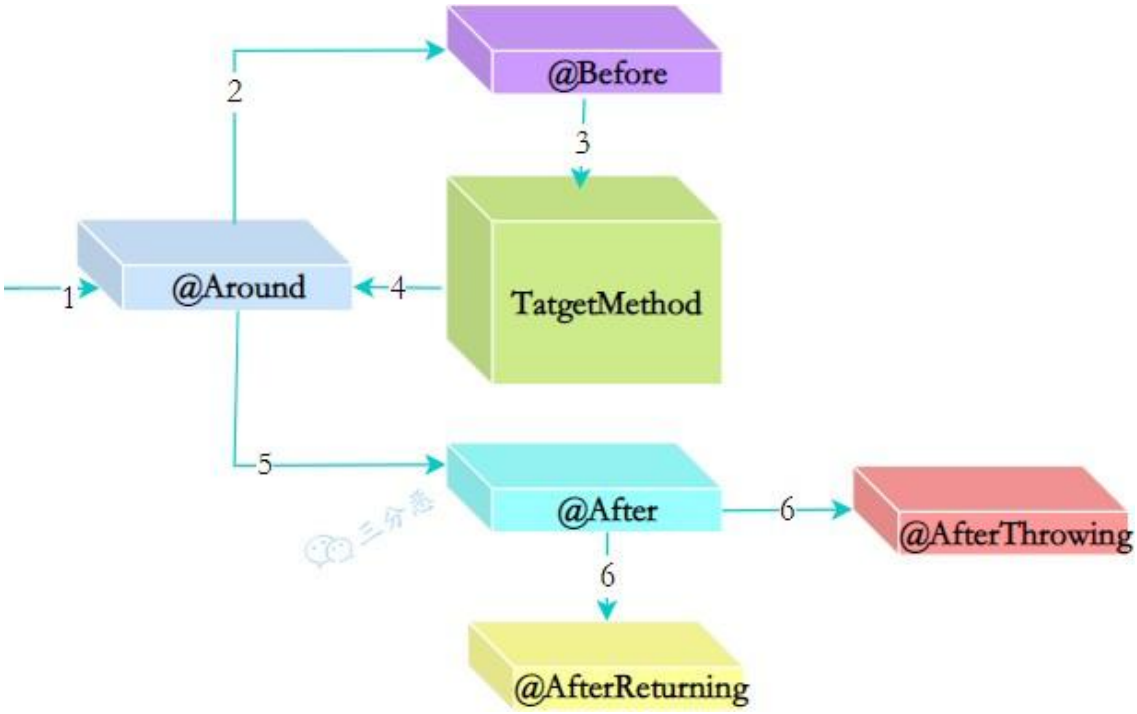
- **切面**（Aspect）：类是对物体特征的抽象，切面就是对横切关注点的抽象
- **连接点**（Joinpoint）：被拦截到的点，因为 Spring 只支持方法类型的连接点，所以在 Spring 中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器
- **切点**（Pointcut）：对连接点进行拦截的定位
- **通知**（Advice）：所谓通知指的就是指拦截到连接点之后要执行的代码，也可以称作**增强**
- **目标对象**（Target）：代理的目标对象
- **织入**（Weabing）：织入是将增强添加到目标类的具体连接点上的过程。编
  - 译期织入：切面在目标类编译时被织入
  - 类加载期织入：切面在目标类加载到 JVM 时被织入。需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。
  - 运行期织入：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP 容器会为目标对象动态地创建一个代理对象。SpringAOP 就是以这种方式织入切面。

    Spring 采用运行期织入，而 AspectJ 采用编译期织入和类加载器织入。
- **引介**（introduction）：引介是一种特殊的增强，可以动态地为类添加一些属性和方法

> AOP **有哪些环绕方式？**

AOP 一般有 5 **种**环绕方式：

- 前置通知 (@Before)
- 返回通知 (@AfterReturning)异
- 常通知 (@AfterThrowing)后置
- 通知 (@After)
- 环绕通知 (@Around)



多个切面的情况下，可以通过 @Order 指定先后顺序，数字越小，优先级越高。

20. 说说你平时有用到 AOP 吗？

PS：这道题老三的同事面试候选人的时候问到了，候选人说了一堆 AOP 原理，同事就势来一句，你能现场写一下 AOP 的应用吗？结果——场面一度很尴尬。虽然我对面试写这种百度就能出来的东西持保留意见，但是还是加上了这一问，毕竟招人最后都是要撸代码的。

这里给出一个小例子，SpringBoot 项目中，利用 AOP 打印接口的入参和出参日志，以及执行时间，还是比较快捷的。

- 引入依赖：引入 AOP 依赖

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
```

- 自定义注解：自定义一个注解作为切点

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Documented
public @interface WebLog {
}
```

- 配置 AOP 切面：
  - @Aspect：标识切面

- @Pointcut：设置切点，这里以自定义注解为切点，定义切点有很多其它种方式，自定义注解是比较常用的一种。
- @Before：在切点之前织入，打印了一些入参信息
- @Around：环绕切点，打印返回参数和接口执行时间

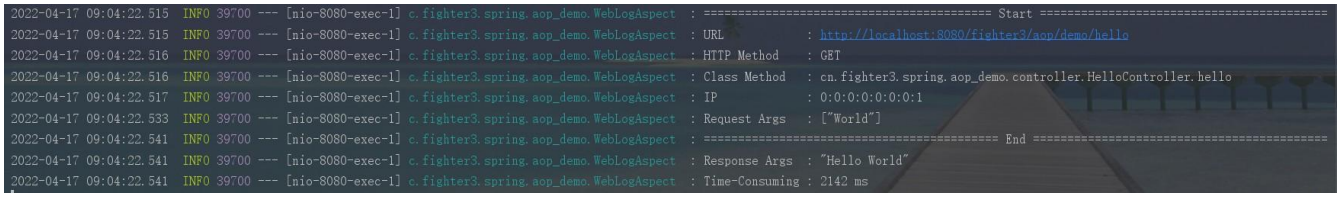
```
@Aspect @Component
public class WebLogAspect {

    private final static Logger logger                                    =
LoggerFactory.getLogger(WebLogAspect.class);

    /**
     * 以自定义 @WebLog 注解为切点
     **/ @Pointcut("@annotation(cn.fighter3.spring.aop_demo.WebLog)") public void webLog() {}

    /**
     * 在切点之前织入
     */ @Before("webLog()")
    public void doBefore(JoinPoint joinPoint) throws Throwable {
        // 开始打印请求日志
        ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();
        HttpServletRequest request = attributes.getRequest();
        // 打印请求相关参数
        logger.info("========================================== Start
==========================================");
        // 打印请求 url
        logger.info("URL                    : {}", request.getRequestURL().toString());
        // 打印 Http method
        logger.info("HTTP Method            : {}", request.getMethod());
        // 打印调用 controller 的全路径以及执行方法
        logger.info("Class Method           : {}.{}",
joinPoint.getSignature().getDeclaringTypeName(),
joinPoint.getSignature().getName());
        // 打印请求的 IP
        logger.info("IP                     : {}", request.getRemoteAddr());
        // 打印请求入参
        logger.info("Request Args           : {}",new
ObjectMapper().writeValueAsString(joinPoint.getArgs()));
    }

    /**
     * 在切点之后织入
     * @throws Throwable
     */
```

```
    @After("webLog()")
    public void doAfter() throws Throwable {
        // 结束后打 分隔线，方便查看
        logger.info("========================================== End
==========================================");
    }

    /**
     * 环绕
     */
    @Around("webLog()")
    public Object doAround(ProceedingJoinPoint proceedingJoinPoint) throws
Throwable {
        //开始时间
        long startTime = System.currentTimeMillis();
        Object result = proceedingJoinPoint.proceed();
        // 打印出参
        logger.info("Response Args  : {}", new
ObjectMapper().writeValueAsString(result));
        // 执行耗时
        logger.info("Time-Consuming : {} ms", System.currentTimeMillis() -
startTime);
        return result;
    }

}
```

- 使用：只需要在接口上加上自定义注解

```
    @GetMapping("/hello")
    @WebLog(desc = "这是一 欢迎接口")
    public String hello(String name){
        return "Hello "+name;
    }
```

- 执行结果：可以看到日志打印了入参、出参和执行时间



21. 说说 JDK 动态代理和 CGLIB 代理？

Spring 的 AOP 是通过动态代理来实现的，动态代理主要有两种方式 JDK 动态代理和 Cglib 动态代理，这两种动态代理的使用和原理有些不同。

JDK **动态代理**

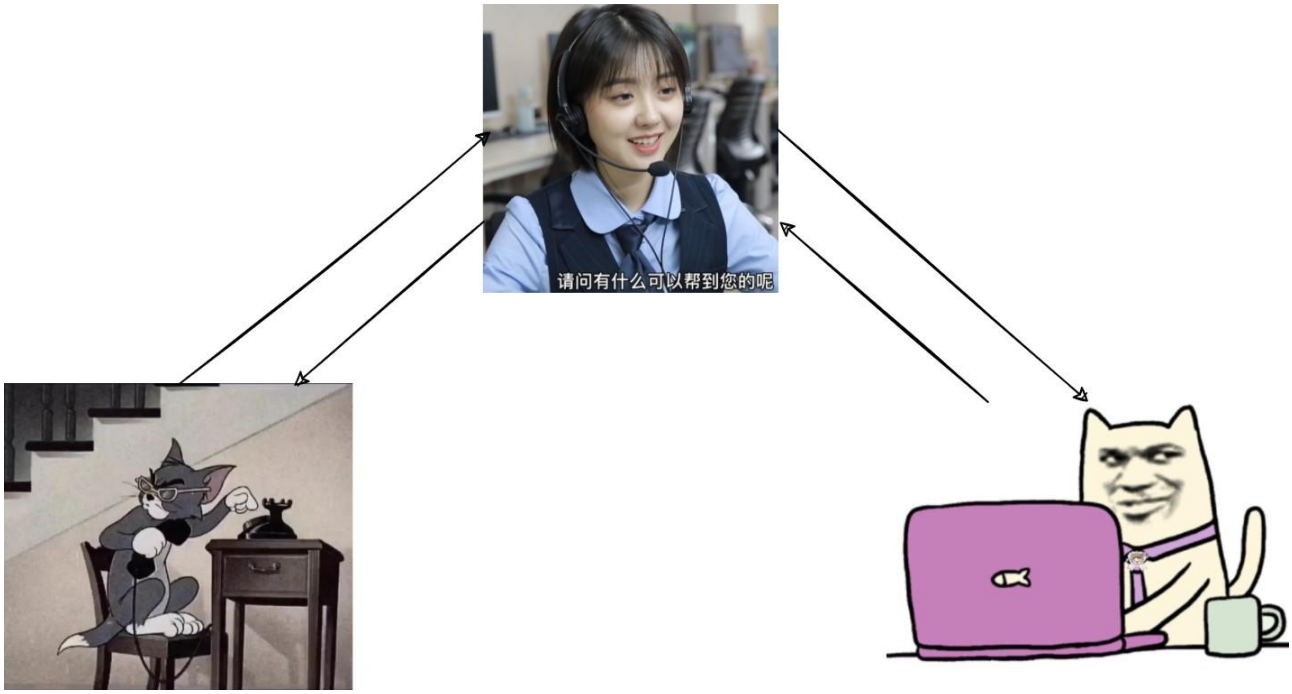
1. Interface：对于 JDK 动态代理，目标类需要实现一个 Interface。

2. InvocationHandler：InvocationHandler 是一个接口，可以通过实现这个接口，定义横切逻辑，再通过反射机制（invoke）调用目标类的代码，在次过程，可能包装逻辑，对目标方法进行前置后置处理。
3. Proxy：Proxy 利用 InvocationHandler 动态创建一个符合目标类实现的接口的实例，生成目标类的代理对象。

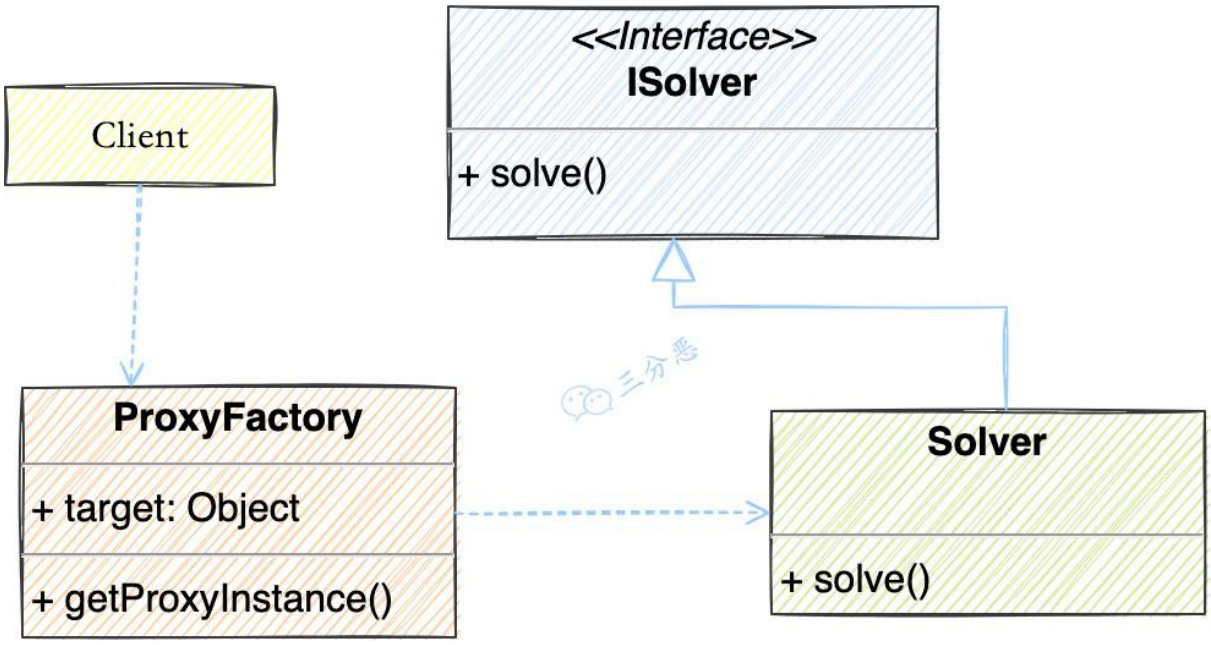
CgLib **动态代理**

1. 使用 JDK 创建代理有一大限制，它只能为接口创建代理实例，而 CgLib 动态代理就没有这个限制。
2. CgLib 动态代理是使用字节码处理框架 ASM，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。
3. CgLib 创建的动态代理对象性能比 JDK 创建的动态代理对象的性能高不少，但是 CGLib 在创建代理对象时所花费的时间却比 JDK 多得多，所以对于单例的对象，因为无需频繁创建对象，用 CGLib 合适，反之，使用 JDK 方式要更为合适一些。同时，由于 CGLib 由于是采用动态创建子类的方法，对于 final 方法，无法进行代理。

我们来看一个常见的小场景，客服中转，解决用户问题：



JDK **动态代理实现：**



- 接口

```
public interface ISolver
    { void solve();
}
```

- 目标类:需要实现对应接口

```
public class Solver implements ISolver
    { @Override
    public void solve() {
        System.out.println("疯狂 头发解 问题.....");
    }
}
```

- 态代理工厂:ProxyFactory，直接用反射方式生成一个目标对象的代理对象，这里用了一个匿名内部类方式重写 InvocationHandler 方法，实现接口重写也差不多

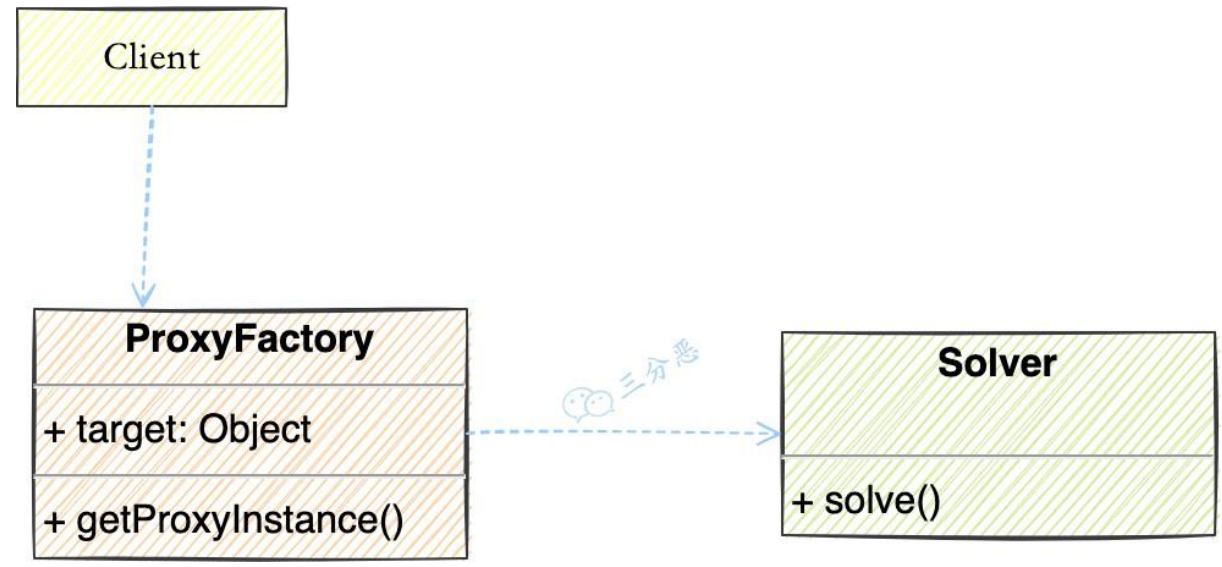
```
public class ProxyFactory {

    // 维护一 目标对象
    private Object target;

    public ProxyFactory(Object target) {
        this.target = target;
    }

    // 为目标对象生成代理对象
    public Object getProxyInstance() {
```

```
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(),
                new InvocationHandler()
                    { @Override
                    public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                        System.out.println("请问有什么可以帮到您？");

                        // 调用目标对象方法
                        Object returnValue = method.invoke(target, args);

                        System.out.println("问题  经解  啦！");
                        return null;
                    }
                });
    }
}
```

- 客户端：Client，生成一个代理对象实例，通过代理对象调用目标对象方法

```
public class Client {
    public static void main(String[] args) {
        //目标对象:程序员
        ISolver developer = new Solver();
        //代理：客服小姐姐
        ISolver csProxy = (ISolver) new ProxyFactory(developer).getProxyInstance();
        //目标方法：解  问题
        csProxy.solve();
    }
}
```

Cglib 动态代理实现：



- 目标类：Solver，这里目标类不用再实现接口。

```
public class Solver {

    public void solve() {
        System.out.println("疯狂  头发解  问题……");
    }
}
```

- 动态代理工厂：

```
public class ProxyFactory implements MethodInterceptor {

   //维护一  目标对象
    private Object target;

    public ProxyFactory(Object target) {
        this.target = target;
    }

    //为目标对象生成代理对象
    public Object getProxyInstance() {
        //工具类
        Enhancer en = new Enhancer();
        //设置父类
        en.setSuperclass(target.getClass());
        //设置回调函数
        en.setCallback(this);
        //创建子类对象代理
        return en.create();
    }


    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy
proxy) throws Throwable {
        System.out.println("请问有什么可以帮到您？");
        // 执行目标对象的方法
        Object returnValue = method.invoke(target, args);
        System.out.println("问题  经解  啦！");
        return null;
    }

}
```

- 客户端：Client

```
public class Client {
    public static void main(String[] args) {
        //目标对象:程序员
        Solver developer = new Solver();
        //代理：客服小姐姐
        Solver csProxy = (Solver) new ProxyFactory(developer).getProxyInstance();
        //目标方法：解  问题
        csProxy.solve();
    }
}
```

22. 说说 Spring AOP 和 AspectJ AOP 区别?

Spring AOP

Spring AOP 属于`运行时增强`，主要具有如下特点：

1. 基于动态代理来实现，默认如果使用接口的，用 JDK 提供的动态代理实现，如果是方法则使用 CGLIB 实现
2. Spring AOP 需要依赖 IOC 容器来管理，并且只能作用于 Spring 容器，使用纯 Java 代码实现
3. 在性能上，由于 Spring AOP 是基于**动态代理**来实现的，在容器启动时需要生成代理实例，在方法调用上也会增加栈的深度，使得 Spring AOP 的性能不如 AspectJ 的那么好。
4. Spring AOP 致力于解决企业级开发中最普遍的 AOP(方法织入)。

AspectJ

AspectJ 是一个易用的功能强大的 AOP 框架，属于`编译时增强`，可以单独使用，也可以整合到其它框架中，是 AOP 编程的完全解决方案。AspectJ 需要用到单独的编译器 ajc。

AspectJ 属于**静态织入**，通过修改代码来实现，在实际运行之前就完成了织入，所以说它生成的类是没有额外运行时开销的，一般有如下几个织入的时机：

1. 编译期织入（Compile-time weaving）：如类 A 使用 AspectJ 添加了一个属性，类 B 引用了它，这个场景就需要编译期的时候就进行织入，否则没法编译类 B。
2. 编译后织入（Post-compile weaving）：也就是已经生成了.class 文件，或已经打成 jar 包了，这种情况我们需要增强处理的话，就要用到编译后织入。
3. 类加载后织入（Load-time weaving）：指的是在加载类的时候进行织入，要实现这个时期的织入，有几种常见的方法

整体对比如下：

Spring AOP	AspectJ
在纯Java中实现	用Java编程语言扩展实现
编译器javac	一般需要ajc
只可运行时织入	支持编译时、编译后、加载时织入
仅支持方法级编织	可编织字段、方法、构造函数、静态初始值等
只可在Spring管理的Bean上实现	可在所有域对象实现
仅支持方法执行切入点	支持所有切入点
比AspectJ慢很多	速度比AOP快很多
易学习使用	比AOP更复杂
创建目标对象的代理，切面在代理中执行	执行程序前，各方面直接织入代码中

事务

Spring 事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，Spring 是无法提供事务功能的。 Spring 只提供统一事务管理接口，具体实现都是由各数据库自己实现，数据库事务的提交和回滚是通过数据库自己的事务机制实现。

23. Spring 事务的种类？

Spring 支持编程式事务管理和声明式事务管理两种方式：

1. 编程式事务

编程式事务管理使用 TransactionTemplate，需要显式执行事务。

2. 声明式事务
3. 声明式事务管理建立在 AOP 之上的。其本质是通过 AOP 功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前启动一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务
4. 优点是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过 @Transactional 注解的方式，便可以将事务规则应用到业务逻辑中，减少业务代码的污染。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

24. Spring 的事务隔离级别？

Spring 的接口 TransactionDefinition 中定义了表示隔离级别的常量，当然其实主要还是对应数据库的事务隔离级别：

1. ISOLATION_DEFAULT：使用后端数据库默认的隔离界别，MySQL 默认可重复读，Oracle 默认读已提交。
2. ISOLATION_READ_UNCOMMITTED：读未提交
3. ISOLATION_READ_COMMITTED：读已提交
4. ISOLATION_REPEATABLE_READ：可重复读
5. ISOLATION_SERIALIZABLE：串行化

25. Spring 的事务传播机制？

Spring 事务的传播机制说的是，当多个事务同时存在的时候——一般指的是多个事务方法相互调用时，Spring 如何处理这些事务的行为。

事务传播机制是使用简单的 ThreadLocal 实现的，所以，如果调用的方法是在新线程调用的，事务传播实际上是会失效的。

7种事务传播机制

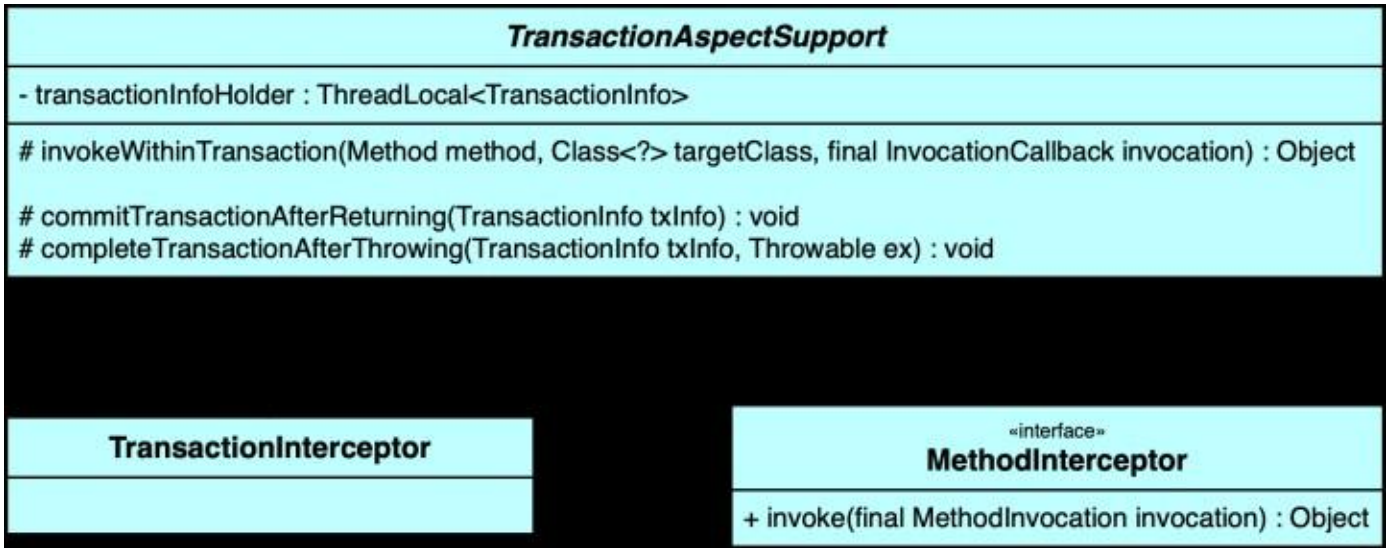
事务传播机制	描述
REQUIERD	默认，如果没有当前事务，就新建一个事务，如果存在一个事务，就加入到这个事务中
SUPPORTS	支持当前事务，如果没有当前事务，就以非事务方法执行
MANDATORY	使用当前事务，如果没有当前事务，就抛出异常
REQUIRED_NEW	新建事务，如果存在当前事务，就把当前事务挂起
NOT_SUPPORTED	以非事务方式执行操作，如果当前事务存在则抛出异常
NESTED	如果当前存在事务，则在事务内执行，如果当前没有事务，则执行与REQUIRED类似操作

Spring 默认的事务传播行为是 PROPAFATION_REQUIRED，它适合绝大多数情况，如果多个 ServiceX#methodX()都工作在事务环境下（均被 Spring 事务增强），且程序中存在调用链 Service1#method1()->Service2#method2()->Service3#method3()，那么这 3 个服务类的三个方法通过 Spring 的事务传播机制都工作在同一个事务中。

26. 声明式事务实现原理了解吗？

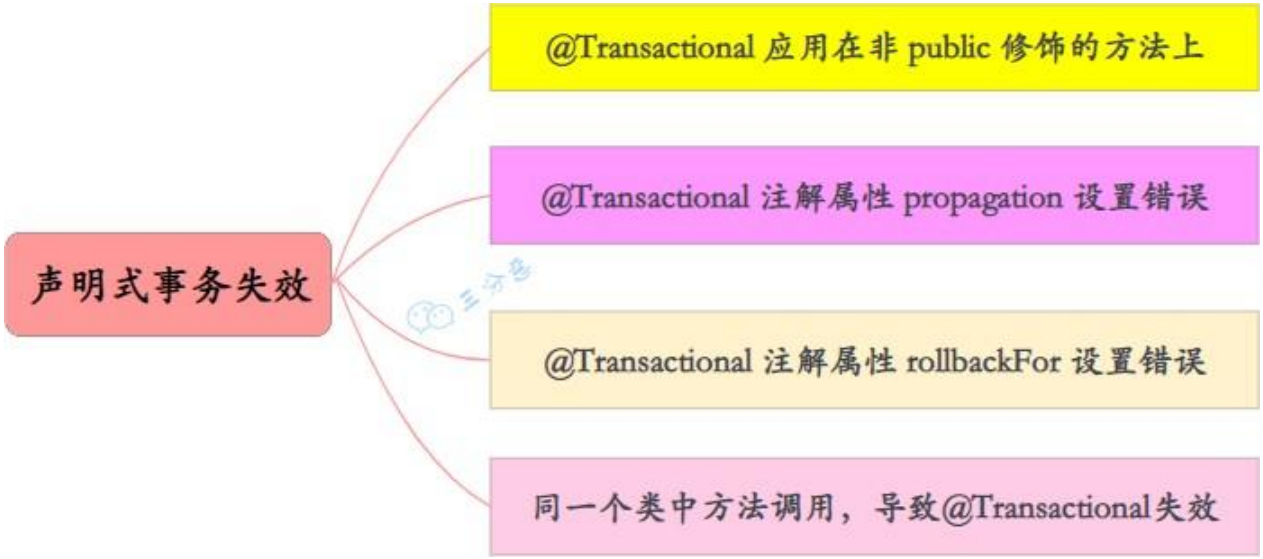
就是通过 AOP/动态代理。

- **在 Bean 初始化阶段创建代理对象**：Spring 容器在初始化每个单例bean 的时候，会遍历容器中的所有 BeanPostProcessor 实现类，并执行其 postProcessAfterInitialization 方法，在执行 AbstractAutoProxyCreator 类的 postProcessAfterInitialization 方法时会遍历容器中所有的切面，查找与当前实例化 bean 匹配的切面，这里会获取事务属性切面，查找 @Transactional 注解及其属性值，然后根据得到的切面创建一个代理对象，默认是使用 JDK 动态代理创建代理，如果目标类是接口，则使用 JDK 动态代 理，否则使用 Cglib。
- **在执行目标方法时进行事务增强操作**：当通过代理对象调用 Bean 方法的时候，会触发对应的 AOP 增强拦截器，声明式事务是一种环绕增强，对应接口为MethodInterceptor，事务增强对该接口的实现为 TransactionInterceptor，类图如下：



事务拦截器TransactionInterceptor在 invoke方法中，通过调用父类TransactionAspectSupport的 invokeWithinTransaction方法进行事务处理，包括开启事务、事务提交、异常回滚。

27.声明式事务在哪些情况下会失效？



1、@Transactional **应用在非** public **修饰的方法上**

如果 Transactional 注解应用在非 public 修饰的方法上，Transactional 将会失效。

是因为在 Spring AOP 代理时，TransactionInterceptor （事务拦截器）在目标方法执行前后进行拦截，DynamicAdvisedInterceptor（CglibAopProxy 的内部类）的 intercept 方法 或 JdkDynamicAopProxy 的 invoke方法会间接调用 AbstractFallbackTransactionAttributeSource 的 computeTransactionAttribute方法，获取 Transactional 注解的事务配置信息。

```
protected TransactionAttribute computeTransactionAttribute(Method method,
    Class<?> targetClass) {
        // Don't allow no-public methods as required.
        if (allowPublicMethodsOnly() && !Modifier.isPublic(method.getModifiers()))
        { return null;
}
```

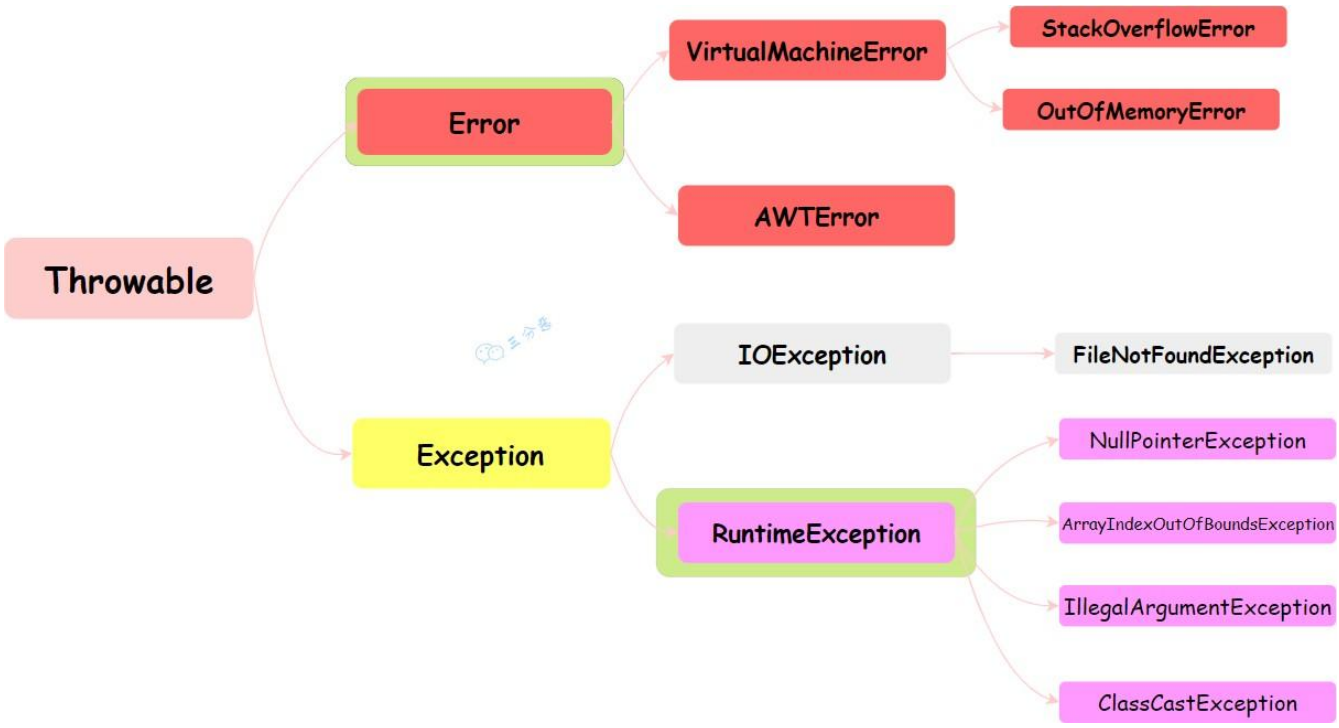
此方法会检查目标方法的修饰符是否为 public，不是 public 则不会获取@Transactional 的属性配置信息。

2、@Transactional **注解属性** propagation **设置错误**

- TransactionDefinition.PROPAGATION_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。 TransactionDefinition.PROPAGATION_NOT_SUPPORTED：以非事务方式
- 运行，如果当前存在事务，则把当前事务挂起。
  TransactionDefinition.PROPAGATION_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。

3、@Transactional **注解属性** rollbackFor **设置错误**

rollbackFor 可以指定能够触发事务回滚的异常类型。Spring 默认抛出了未检查 unchecked 异常（继承自 RuntimeException 的异常）或者 Error 才回滚事务，其他异常不会触发回滚事务。




```
// 希望自定义的异常可以进行回滚
@Transactional(propagation= Propagation.REQUIRED,rollbackFor= MyException.class
```

若在目标方法中抛出的异常是 rollbackFor 指定的异常的子类，事务同样会回滚。

4、**同一个类中方法调用，导致**@Transactional **失效**

开发中避免不了会对同一个类里面的方法调用，比如有一个类 Test，它的一个方法 A，A 再调用本类的方法 B（不论方法 B 是用 public 还是 private 修饰），但方法 A 没有声明注解事务，而 B 方法有。则外部调用方法 A 之后，方法 B 的事务是不会起作用的。这也是经常犯错误的一个地方。

那为啥会出现这种情况？其实这还是由于使用 Spring AOP 代理造成的，因为只有当事务方法被当前类以外的代码调用时，才会由 Spring 生成的代理对象来管理。

```
//@Transactional
    @GetMapping("/test")
    private Integer A() throws Exception {
        CityInfoDict cityInfoDict = new CityInfoDict();
        cityInfoDict.setCityName("2");
        /**
         * B   入字段为 3的数据
         */
        this.insertB();
       /**
        * A   入字段为 2的数据
        */
       int insert = cityInfoDictMapper.insert(cityInfoDict);
       return insert;
    }

    @Transactional()
    public Integer insertB() throws Exception
        { CityInfoDict cityInfoDict = new
        CityInfoDict(); cityInfoDict.setCityName("3");
        cityInfoDict.setParentCityId(3);

        return cityInfoDictMapper.insert(cityInfoDict);
    }
```

这种情况是最常见的一种@Transactional 注解失效场景

```
@Transactional
private Integer A() throws Exception
    { int insert = 0;
    try {
        CityInfoDict cityInfoDict = new CityInfoDict();
        cityInfoDict.setCityName("2");
        cityInfoDict.setParentCityId(2);
        /**
         * A   入字段为 2的数据
         */
        insert = cityInfoDictMapper.insert(cityInfoDict);
        /**
         * B   入字段为 3的数据
        */
```

```
            b.insertB();
        } catch (Exception e)
            { e.printStackTrace();
        }
    }
```

如果 B 方法内部抛了异常，而 A 方法此时 try catch 了B 方法的异常，那这个事务就不能正常回滚了，会抛出异常：

```
org.springframework.transaction.UnexpectedRollbackException: Transaction rolled back
because it has been marked as rollback-only
```

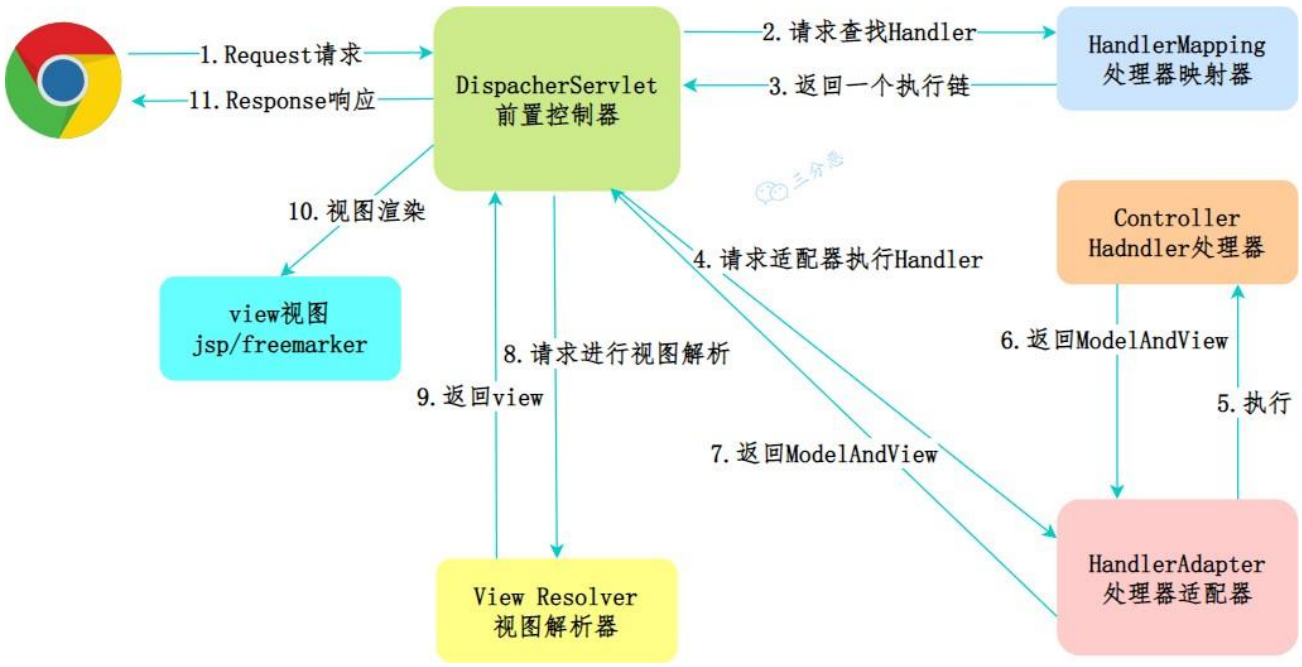
MVC

28. Spring MVC 的核心组件？

1. DispatcherServlet：前置控制器，是整个流程控制的**核心**，控制其他组件的执行，进行统一调度，降低组件之间的耦合性，相当于总指挥。
2. Handler：处理器，完成具体的业务逻辑，相当于 Servlet 或 Action。
3. HandlerMapping：DispatcherServlet 接收到请求之后，通过 HandlerMapping 将不同的请求映射到不同的 Handler。
4. HandlerInterceptor：处理器拦截器，是一个接口，如果需要完成一些拦截处理，可以实现该接口。
5. HandlerExecutionChain：处理器执行链，包括两部分内容：Handler 和 HandlerInterceptor（系统会有一个默认的 HandlerInterceptor，如果需要额外设置拦截，可以添加拦截器）。
6. HandlerAdapter：处理器适配器，Handler 执行业务方法之前，需要进行一系列的操作，包括表单数据的验证、数据类型的转换、将表单数据封装到 JavaBean 等，这些操作都是由 HandlerApater 来完成，开发者只需将注意力集中业务逻辑的处理上，DispatcherServlet 通过 HandlerAdapter 执行不同的 Handler。
7. ModelAndView：装载了模型数据和视图信息，作为 Handler 的处理结果，返回给 DispatcherServlet。
8. ViewResolver：视图解析器，DispatcheServlet 通过它将逻辑视图解析为物理视图，最终将渲染结果响应给

客户端。

29. Spring MVC 的工作流程？



1. 客户端向服务端发送一次请求，这个请求会先到前端控制器 DispatcherServlet(也叫中央控制器)。
2. DispatcherServlet 接收到请求后会调用 HandlerMapping 处理器映射器。由此得知，该请求该由哪个 Controller 来处理（并未调用 Controller，只是得知）
3. DispatcherServlet 调用 HandlerAdapter 处理器适配器，告诉处理器适配器应该要去执行哪个 Controller
4. HandlerAdapter 处理器适配器去执行 Controller 并得到 ModelAndView(数据和视图)，并层层返回给 DispatcherServlet
5. DispatcherServlet 将 ModelAndView 交给 ViewReslover 视图解析器解析，然后返回真正的视图。
6. DispatcherServlet 将模型数据填充到视图中
7. DispatcherServlet 将结果响应给客户端

Spring MVC 虽然整体流程复杂，但是实际开发中很简单，大部分的组件不需要开发人员创建和管理，只需要通过配置文件的方式完成配置即可，真正需要开发人员进行处理的只有 Handler（Controller）、View、Model。

当然我们现在大部分的开发都是前后端分离，Restful 风格接口，后端只需要返回 Json 数据就行了。

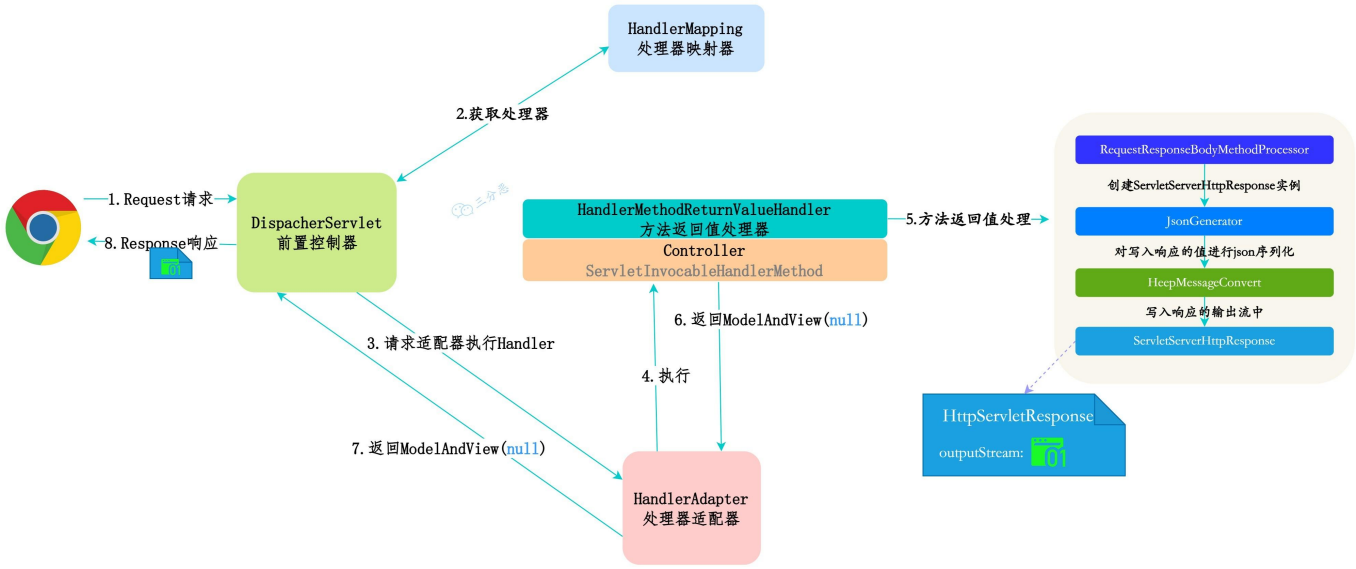
30. SpringMVC Restful 风格的接口的流程是什么样的呢？

PS:这是一道全新的八股，毕竟 ModelAndView 这种方式应该没人用了吧？现在都是前后端分离接口，八股也该更新换代了。

我们都知道 Restful 接口，响应格式是 json，这就用到了一个常用注解：@ResponseBody

```
@GetMapping("/user")
@ResponseBody
public User user(){
    return new User(1,"张三");
}
```

加入了这个注解后，整体的流程上和使用 ModelAndView 大体上相同，但是细节上有一些不同：



1. 客户端向服务端发送一次请求，这个请求会先到前端控制器 DispatcherServlet
2. DispatcherServlet 接收到请求后会调用 HandlerMapping 处理器映射器。由此得知，该请求该由哪个 Controller 来处理
3. DispatcherServlet 调用 HandlerAdapter 处理器适配器，告诉处理器适配器应该要去执行哪个 Controller
4. Controller 被封装成了 ServletInvocableHandlerMethod，HandlerAdapter 处理器适配器去执行 invokeAndHandle 方法，完成对 Controller 的请求处理
5. HandlerAdapter 执行完对 Controller 的请求，会调用 HandlerMethodReturnValueHandler 去处理返回值，主要的过程：

   5.1. 调用 RequestResponseBodyMethodProcessor，创建 ServletServerHttpResponse（Spring 对原生 ServerHttpResponse 的封装）实例

   5.2. 使用 HttpMessageConverter 的 write 方法，将返回值写入 ServletServerHttpResponse 的 OutputStream 输出流中

   5.3. 在写入的过程中，会使用 JsonGenerator（默认使用 Jackson 框架）对返回值进行 Json 序列化
6. 执行完请求后，返回的 ModealAndView 为 null，ServletServerHttpResponse 里也已经写入了响应，所以不用关心 View 的处理

31.介绍一下 SpringBoot，有哪些优点？

Spring Boot 基于 Spring 开发，Spirng Boot 本身并不提供 Spring 框架的核心特性以及扩展功能，只是用于快 速、敏捷地开发新一代基于 Spring 框架的应用程序。它并不是用来替代 Spring 的解决方案，而是和 Spring 框架紧密结合用于提升 Spring 开发者体验的工具。

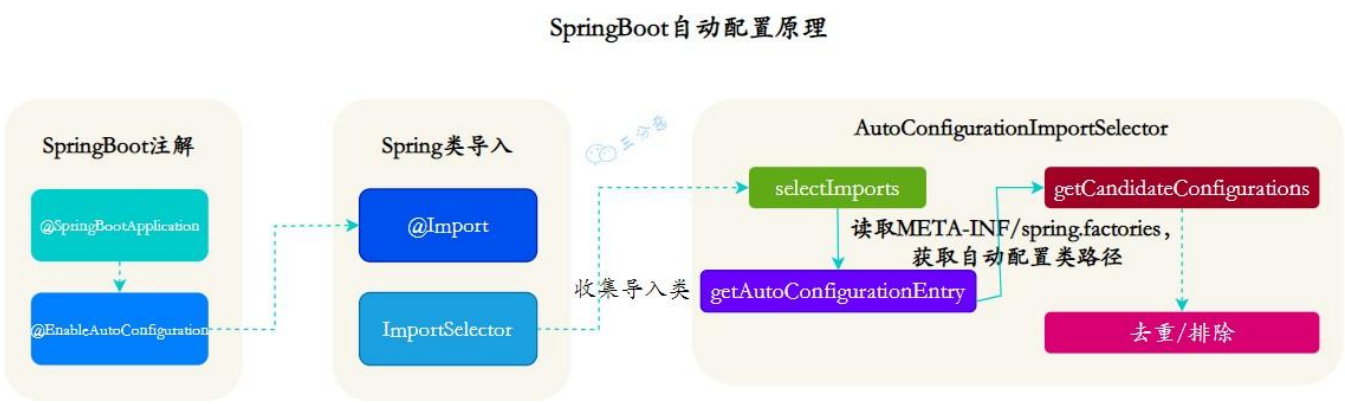
Spring Boot 以约定大于配置核心思想开展工作，相比 Spring 具有如下优势：

1. Spring Boot 可以快速创建独立的 Spring 应用程序。
2. Spring Boot 内嵌了如 Tomcat，Jetty 和 Undertow 这样的容器，也就是说可以直接跑起来，用不着再做部署工作了。
3. Spring Boot 无需再像 Spring 一样使用一堆繁琐的 xml 文件配置。
4. Spring Boot 可以自动配置(核心)Spring。SpringBoot 将原有的 XML 配置改为 Java 配置，将 bean 注入改为使用注解注入的方式(@Autowire)，并将多个 xml、properties 配置浓缩在一个 appliaction.yml 配置文件 中。
5. Spring Boot 提供了一些现有的功能，如量度工具，表单数据验证以及一些外部配置这样的一些第三方功能。

6. Spring Boot 可以快速整合常用依赖（开发库，例如 spring-webmvc、jackson-json、validation-api 和 tomcat 等），提供的 POM 可以简化 Maven 的配置。当我们引入核心依赖时，SpringBoot 会自引入其他依赖。

32. SpringBoot 自动配置原理了解吗？

SpringBoot 开启自动配置的注解是`@EnableAutoConfiguration` ，启动类上的注解`@SpringBootApplication` 是一个复合注解，包含了@EnableAutoConfiguration：



- `EnableAutoConfiguration` 只是一个简单的注解，自动装配核心功能的实现实际是通过 `AutoConfigurationImportSelector`类

```
@AutoConfigurationPackage //将main同级的包下的所有组件注 到容器中
@Import({AutoConfigurationImportSelector.class}) //加载自动装配类
xxxAutoconfiguration
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    Class<?>[] exclude() default {};

    String[] excludeName() default {};
}
```

- `AutoConfigurationImportSelector`实现了`ImportSelector`接口，这个接口的作用就是收集需要导入的配置类，配合 @Import() 就可以将相应的类导入到 Spring 容器中
- 获取注入类的方法是 selectImports()，它实际调用的是`getAutoConfigurationEntry`，这个方法是获取自动装配类的关键，主要流程可以分为这么几步：
  1. 获取注解的属性，用于后面的排除
  2. **获取所有需要自动装配的配置类的路径**：这一步是最关键的，从 META-INF/spring.factories 获取自动配置类的路径
  3. 去掉重复的配置类和需要排除的重复类，把需要自动加载的配置类的路径存储起来

```
    protected AutoConfigurationImportSelector.AutoConfigurationEntry
getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
        if (!this.isEnabled(annotationMetadata)) {
```

```
                return EMPTY_ENTRY;
            } else {
                //1.获取到注解的属性
                AnnotationAttributes attributes = this.getAttributes(annotationMetadata);
                //2.获取需要自动装配的所有配置类，读取META-INF/spring.factories，获取自动配置类路径 List<String>
                configurations =
this.getCandidateConfigurations(annotationMetadata, attributes);
                //3.1.移除重复的配置
                configurations = this.removeDuplicates(configurations);
                //3.2.处理需要排除的配置
                Set<String> exclusions = this.getExclusions(annotationMetadata, attributes);
                this.checkExcludedClasses(configurations, exclusions); configurations.removeAll(exclusions);
                configurations = this.getConfigurationClassFilter().filter(configurations);
                this.fireAutoConfigurationImportEvents(configurations, exclusions);
                return new AutoConfigurationImportSelector.AutoConfigurationEntry(configurations, exclusions);
            }
        }
```

33. 如何自定义一个 SpringBoot Srarter?

知道了自动配置原理，创建一个自定义 SpringBoot Starter 也很简单。

1. 创建一个项目，命名为 demo-spring-boot-starter，引入 SpringBoot 相关依赖

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
            <optional>true</optional>
        </dependency>
```

2. 编写配置文件

   这里定义了属性配置的前缀

```
@ConfigurationProperties(prefix = "hello")
public class HelloProperties {

    private String name;

    //省略getter、setter
}
```

3. 自动装配

   创建自动配置类 HelloPropertiesConfigure

```
@Configuration
@EnableConfigurationProperties(HelloProperties.class)
public class HelloPropertiesConfigure {
}
```

4. 配置自动类

   在 /resources/META-INF/spring.factories 文件中添加自动配置类路径

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
  cn.fighter3.demo.starter.configure.HelloPropertiesConfigure
```

5. 测试

   - 创建一个工程，引入自定义 starter 依赖

```
        <dependency>
            <groupId>cn.fighter3</groupId>
            <artifactId>demo-spring-boot-starter</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
```

   - 在配置文件里添加配置

```
hello.name=张こ
```

   - 测试类

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class HelloTest
    { @Autowired
    HelloProperties helloProperties;

    @Test
    public void hello(){
        System.out.println("你好，"+helloProperties.getName());
    }
}
```

- 运行结果

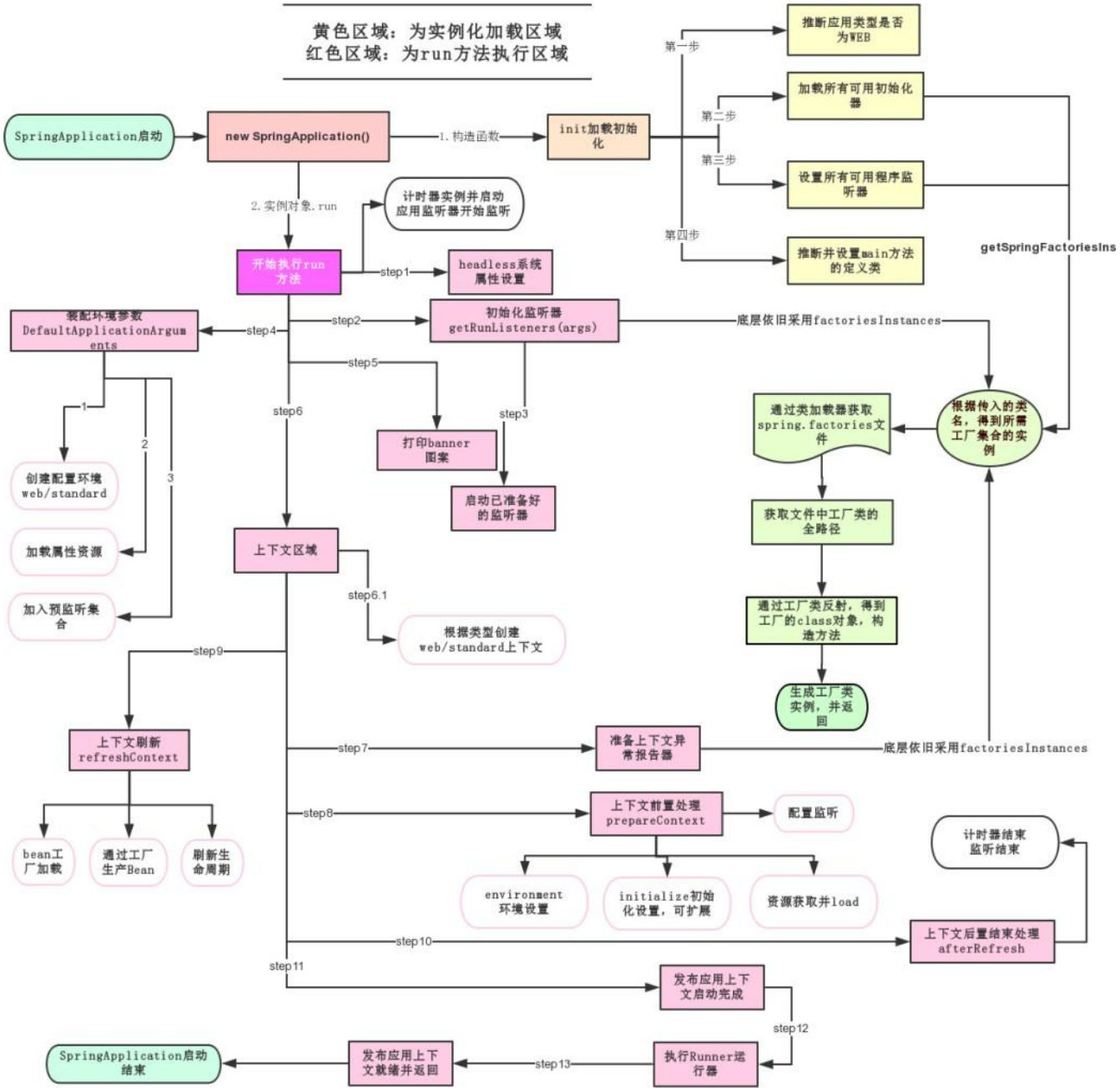
至此，随手写的一个自定义 SpringBoot-Starter 就完成了，虽然比较简单，但是完成了主要的自动装配的能力。

34. Springboot 启动原理?

SpringApplication 这个类主要做了以下四件事情：

1. 推断应用的类型是普通的项目还是 Web 项目
2. 查找并加载所有可用初始化器， 设置到 initializers 属性中
3. 找出所有的应用程序监听器，设置到 listeners 属性中
4. 推断并设置 main 方法的定义类，找到运行的主类

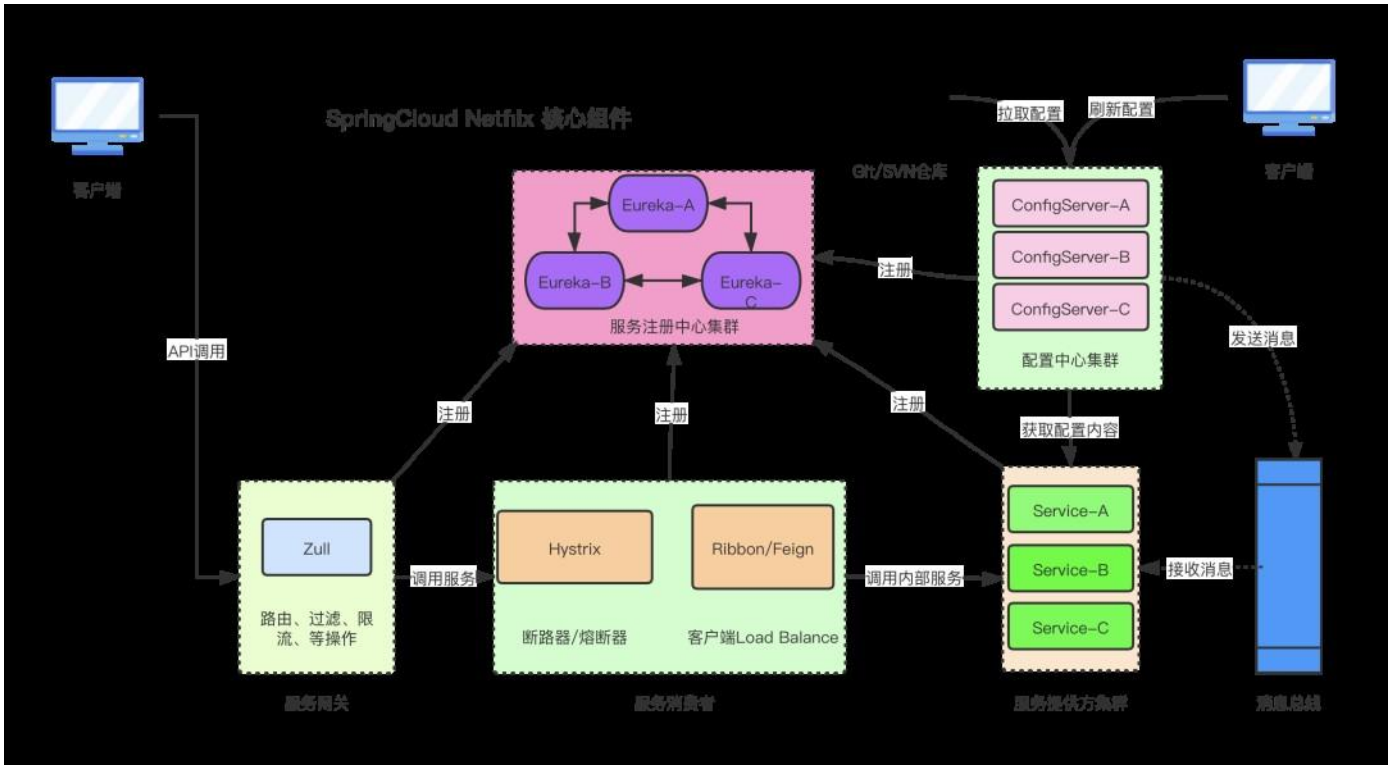
SpringBoot 启动大致流程如下 ：



Spring Cloud

35. 对 SpringCloud 了解多少？

SpringCloud 是 Spring 官方推出的微服务治理框架。



什么是微服务？

1. 2014 年Martin Fowler 提出的一种新的架构形式。微服务架构是一种**架构模式**，提倡将单一应用程序划分成一组小的服务，服务之间相互协调，互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务之间采用轻量级的通信机制(如 HTTP 或 Dubbo)互相协作，每个服务都围绕着具体的业务进行构建，并且能够被独立的部署到生产环境中，另外，应尽量避免统一的，集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具(如 Maven)对其进行构建。
2. 微服务化的核心就是将传统的一站式应用，根据业务拆分成一个一个的服务，彻底地去耦合，每一个微服务提供单个业务功能的服务，一个服务做一件事情，从技术角度看就是一种小而独立的处理过程，类似进程的概念，能够自行单独启动或销毁，拥有自己独立的数据库。

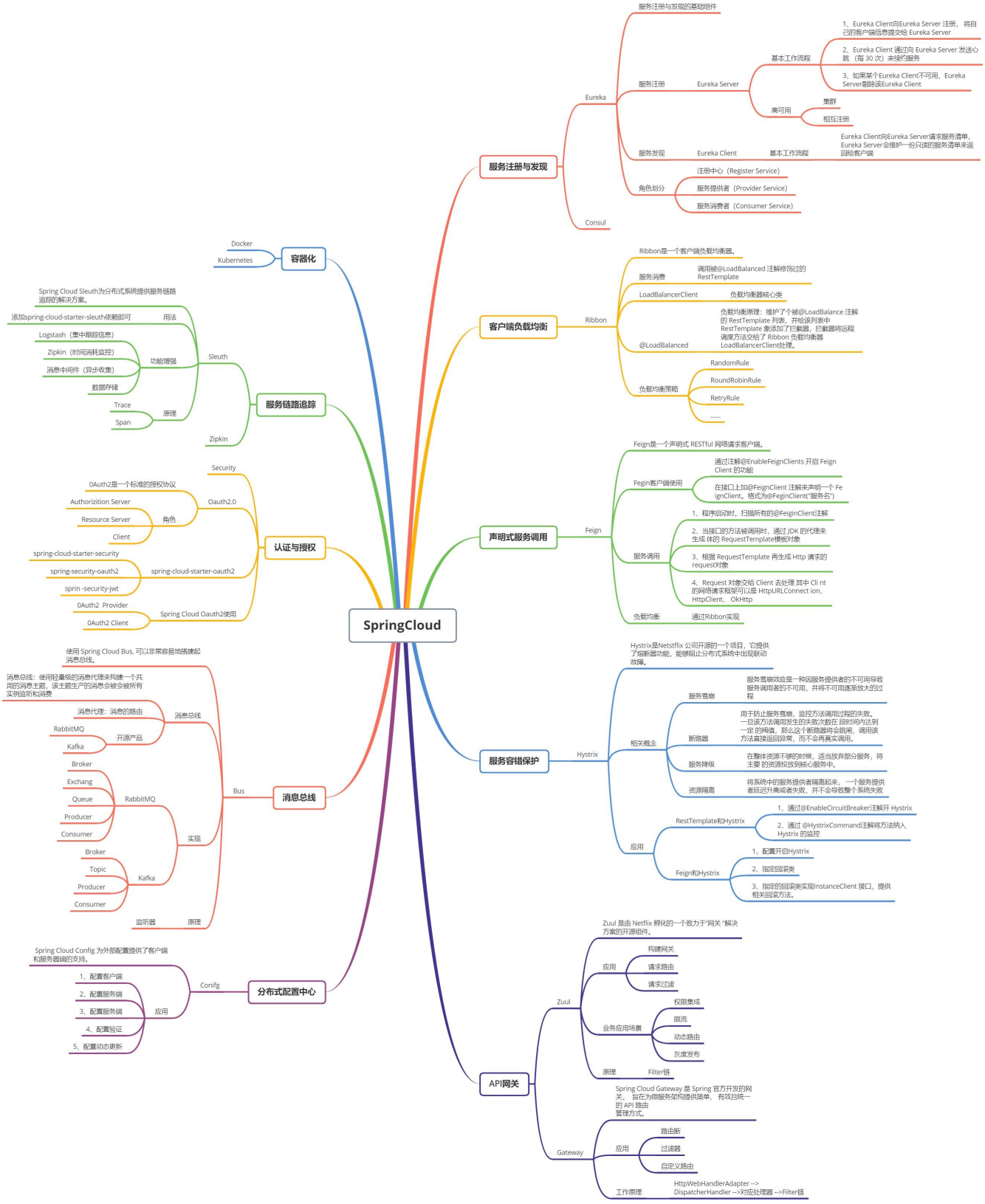
微服务架构主要要解决哪些问题？

1. 服务很多，客户端怎么访问，如何提供对外网关?
2. 这么多服务，服务之间如何通信? HTTP 还是 RPC?
3. 这么多服务，如何治理？服务的注册和发现。
4. 服务挂了怎么办？熔断机制。

有哪些主流微服务框架？

1. Spring Cloud Netflix
2. Spring Cloud Alibaba
3. SpringBoot + Dubbo + ZooKeeper

SpringCloud 有哪些核心组件？



PS:微服务后面有机会再扩展，其实面试一般都是结合项目去问。

> 图文详解 35 道 Spring 面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：楼仔，作者：三分恶，戳原文链接。