# SSM相关面试题

## 1 Spring

### 1.1 什么是Spring IOC 和DI？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

① 控制反转(IOC)：Spring容器使用了工厂模式为我们创建了所需要的对象，我们使用时不需要自己去创建，直接调用Spring为我们提供的对象即可，这就是控制反转的思想。

② 依赖注入(DI)：Spring使用Java Bean对象的Set方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程就是依赖注入的基本思想。

### 1.2 有哪些不同类型的依赖注入实现方式？

> 难易程度：☆☆
>
> 出现频率：☆☆

依赖注入分为Setter方 法注入和构造器注入

构造器依赖注入：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每 个参数代表一个对其他类的依赖。

Setter方法注入：Setter方法注入是容器通过调用无参构造器或无参static工厂 方法实例化bean之 后，调用该bean的setter方法，即实现了基于setter的依赖注入。

## 1.3 Spring支持的几种bean的作用域

> 难易程度：☆☆☆
>
> 出现频率：☆☆

Spring框架支持以下五种bean的作用域：

singleton : bean在每个Spring ioc 容器中只有一个实例。

prototype：一个bean的定义可以有多个实例。

request：每次http请求都会创建一个bean，该作用域仅在基于web的Spring ApplicationContext情形下有效。

session：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的 Spring ApplicationContext情形下有效。

application：属于应用程序域，应用程序启动时bean创建，应用程序销毁时bean销毁。该作用域仅在基于web的ServletContext.

## 1.4 Spring框架中的单例bean是线程安全的吗？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

不是线程安全的

当多用户同时请求一个服务时，容器会给每一个请求分配一个线程，这是多个线程会并发执行该请求对应的业务逻辑（成员方法），如果该处理逻辑中有对该单列状态的修改（体现为该单例的成员属性），则必须考虑线程同步问题。

Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。

Spring bean并没有可变的状态(比如Service类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。

如果你的bean有多种状态的话（比如 View Model对象），就需要自行保证线程安全。最浅显的解决办法就是将多态bean的作用由“**singleton**”变更为“**prototype**”。

## 1.5 Spring自动装配 bean 有哪些方式？

> 难易程度：☆☆
>
> 出现频率：☆☆

spring的自动装配功能的定义：无须在Spring配置文件中描述javaBean之间的依赖关系（如配置`<property>、<constructor-arg>`）。

自动装配模式：

1、no：这是 Spring 框架的默认设置，在该设置下自动装配是关闭的，开发者需要自行在 bean 定义中用标签明确的设置依赖关系 。

2、byName：该选项可以根据bean名称设置依赖关系 。 当向一个bean中自动装配一个属性时，容器将根据bean的名称自动在在配置文件中查询一个匹配的bean。 如果找到的话，就装配这个属性，如果没找到的话就报错 。

3、byType：该选项可以根据 bean 类型设置依赖关系 。 当向一个 bean 中自动装配一个属性时，容器将根据 bean 的类型自动在在配置文件中查询一个匹配的bean。 如果找到的话，就装配这个属性，如果没找到的话就报错 。

4、constructor ：构造器的自动装配和byType模式类似，但是仅仅适用于与有构造器相同参数的bean，如果在容器中没有找到与构造器参数类型一致的bean ，那么将会抛出异常 。

5、default：该模式自动探测使用构造器自动装配或者byType自动装配 。 首先会尝试找合适的带参数的构造器，如果找到的话就是用构造器自动装配，如果在bean内部没有找到相应的 构造器或者是无参构造器，容器就会自动选择 byTpe 的自动装配方式 。

比如如下注入：

```
<bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl" />
<bean id="bookService" class="com.itheima.service.impl.BookServiceImpl">
    <property name="bookDao" ref="bookDao"/>
</bean>
```

可以改造为：

```
<bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl" />
<bean id="bookService" class="com.itheima.service.impl.BookServiceImpl" autowire="byType" />
```

## 1.6 你用过哪些重要的Spring注解？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

1、@Component- 用于服务类。

> @Service
>
> @Repository
>
> @Controller

2、**@Autowired** - 用于在 spring bean 中自动装配依赖项。通过类型来实现自动注入bean。和@Qualifier注解配合使用可以实现根据name注入bean。

3、@Qualifier - 和@Autowired一块使用，在同一类型的bean有多个的情况下可以实现根据name注入的需求。

4、@Scope - 用于配置 spring bean 的范围。

5、@Configuration，@ComponentScan 和 @Bean - 用于基于 java 的配置。

6、@Aspect，@Before，@After，@Around，@Pointcut - 用于切面编程（AOP）

## 1.7 Spring中的事务是如何实现的

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

Spring支持**编程式事务管理**和**声明式事务管理**两种方式！

**编程式事务控制**：需要使用TransactionTemplate来进行实现，这种方式实现对业务代码有侵入性，因此在项目中很少被使用到。

**声明式事务管理**：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

目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就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。

## 1.8 Spring中事务失效的场景？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

因为Spring事务是基于代理来实现的，所以某个加了@Transactional的方法只有是被代理对象调用时， 那么这个注解才会生效，如果使用的是目标对象调用，那么@Transactional会失效

同时如果某个方法是private的，那么@Transactional也会失效，因为底层cglib是基于父子类来实现 的，子类是不能重载父类的private方法的，所以无法很好的利用代理，也会导致@Transactianal失效

如果在业务中对异常进行了捕获处理，出现异常后Spring框架无法感知到异常，@Transactional也会失效

@Transactional中的配置的Rollback的默认是异常是：runTimeException,更改为Exception

## 1.9 说一下Spring的事务传播行为

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

1. PROPAGATION_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就 加入该事务，该设置是最常用的设置。
2. PROPAGATION_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不 存在事务，就以非事务执行。
3. PROPAGATION_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前 不存在事务，就抛出异常。
4. PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。
5. PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前 事务挂起。
6. PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。
7. PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则 按REQUIRED属性执行。

## 1.10 什么是AOP，你们项目中有没有使用到AOP

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

AOP一般称为面向切面编程，作为面向对象的一种补充，用于 将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时 提高了系统的可维护性。

在我们的项目中我们自己写AOP的场景其实很少，但是我们使用的很多框架的功能底层都是AOP ，例如 ：

1、统一日志处理

2、spring中内置的事务处理

> 记录操作日志
>
> 需求：是谁，在什么时间，修改了数据（修改之前和修改之后），删除了什么数据，新增了什么数据
>
> 要求：方法命名要规范
>
> 实现步骤：
>
> 1，定义切面类
>
> 2，使用环绕通知，根据方法名和参数，记录到表中

## 1.11 JDK动态代理和CGLIB动态代理的区别

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

Spring中AOP底层的实现是基于**动态代理**进行实现的。

常见的动态代理技术有两种：JDK的动态代理和CGLIB。

两者的区别如下所示：

1、JDK动态代理只能对实现了接口的类生成代理，而不能针对类

2、Cglib是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法进行增强，但是因为采用的是**继承**，所以该类或方法最好不要声明为final，对于final类或方法，是无法继承的。

Spring如何选择是用JDK还是cglib？

1、当bean实现接口时，会用JDK代理模式

2、当bean没有实现接口，会用cglib实现

3、可以强制使用cglib

- 在springboot项目可以配置以下注解，强制使用cglib

  `@EnableAspectJAutoProxy(proxyTargetClass = true)`

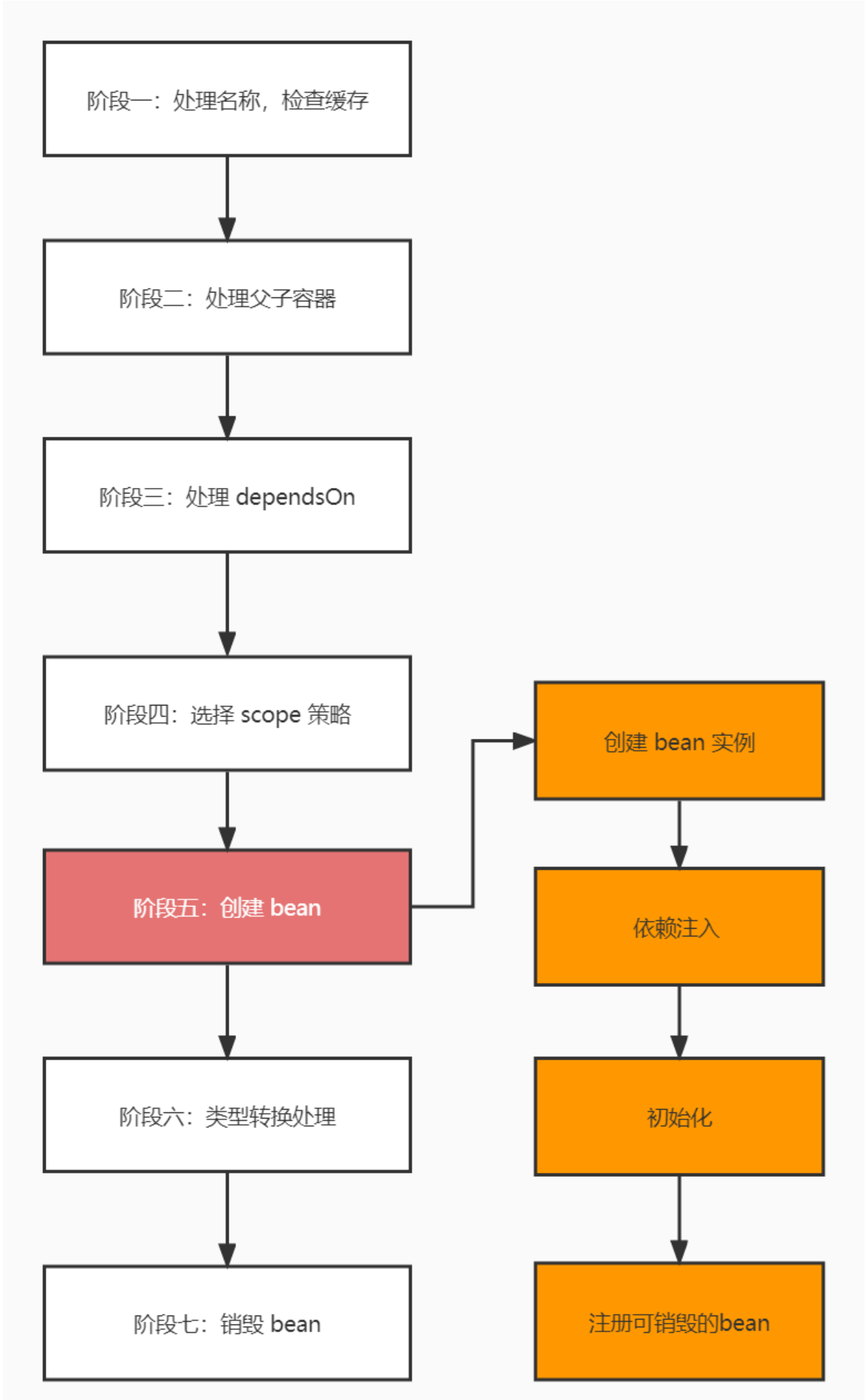
```
@SpringBootApplication
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class WemediaApplication {
```

## 1.12 spring的bean的生命周期

> 难易程度：☆☆☆☆☆
>
> 出现频率：☆☆☆

bean 的生命周期从调用 beanFactory 的 getBean 开始，到这个 bean 被销毁，可以总结为以下七个阶段：

1. 处理名称，检查缓存
2. 处理父子容器
3. 处理 dependsOn
4. 选择 scope 策略
5. 创建 bean (关键阶段)
6. 类型转换处理
7. 销毁 bean


阶段一：处理名称，检查缓存
阶段二：处理父子容器
阶段三：处理 dependsOn
阶段四：选择 scope 策略
阶段五：创建 bean
阶段六：类型转换处理
阶段七：销毁 bean
创建 bean 实例
依赖注入
初始化
注册可销毁的bean

spring框架在创建的bean的时候都会调用AbstractBeanFactory类中的doGetBean方法，下面是截取了部分

```
protected <T> T doGetBean(final String name, @Nullable final
Class<T> requiredType,
        @Nullable final Object[] args, boolean typeCheckOnly) throws
BeansException {
    //获取beanName，这边有三种形式，一个是原始的beanName，一个是加了&
的，一个是别名
    final String beanName = transformedBeanName(name);
    Object bean;

    // Eagerly check singleton cache for manually registered
singletons.
    // 是否已经创建了
    Object sharedInstance = getSingleton(beanName);
    //已经创建了，且没有构造参数，进入这个方法，如果有构造参数，往else
走，也就是说不获取bean，而直接创建bean
    if (sharedInstance != null && args == null) {
        if (logger.isTraceEnabled()) {
            if (isSingletonCurrentlyInCreation(beanName)) {
                logger.trace("Returning eagerly cached instance of
singleton bean '" + beanName +
                        "' that is not fully initialized yet - a
consequence of a circular reference");
            }
            else {
                logger.trace("Returning cached instance of singleton
bean '" + beanName + "'");
            }
        }
        // 如果是普通bean，直接返回，是FactoryBean，返回他的getObject
        bean = getObjectForBeanInstance(sharedInstance, name,
beanName, null);
    }

    else {
        // Fail if we're already creating this bean instance:
        // We're assumably within a circular reference.
        // 没创建过bean或者是多例的情况或者有参数的情况
```

```
        // 创建过Prototype的bean，会循环引用，抛出异常，单例才尝试解决循环依赖的问题
        if (isPrototypeCurrentlyInCreation(beanName)) {
            throw new BeanCurrentlyInCreationException(beanName);
        }

        // Check if bean definition exists in this factory.
        BeanFactory parentBeanFactory = getParentBeanFactory();
        // 父容器存在，本地没有当前beanName，从父容器取
        if (parentBeanFactory != null &&
!containsBeanDefinition(beanName)) {
            // Not found -> check parent.
            // 处理后，如果是加&，就补上&
            String nameToLookup = originalBeanName(name);
            if (parentBeanFactory instanceof AbstractBeanFactory) {
                return ((AbstractBeanFactory)
parentBeanFactory).doGetBean(
                        nameToLookup, requiredType, args,
typeCheckOnly);
            }
            else if (args != null) {
                // Delegation to parent with explicit args.
                return (T) parentBeanFactory.getBean(nameToLookup,
args);
            }
            else if (requiredType != null) {
                // No args -> delegate to standard getBean method.
                return parentBeanFactory.getBean(nameToLookup,
requiredType);
            }
            else {
                return (T) parentBeanFactory.getBean(nameToLookup);
            }
        }

        if (!typeCheckOnly) {
            // typeCheckOnly为false，将beanName放入alreadyCreated中
            markBeanAsCreated(beanName);
        }

        try {
```

```
            // 获取BeanDefinition
            final RootBeanDefinition mbd =
getMergedLocalBeanDefinition(beanName);
            // 抽象类检查
            checkMergedBeanDefinition(mbd, beanName, args);

            // Guarantee initialization of beans that the current
bean depends on.
            // 如果有依赖的情况，先初始化依赖的bean
            String[] dependsOn = mbd.getDependsOn();
            if (dependsOn != null) {
                for (String dep : dependsOn) {
                    // 检查是否循环依赖，a依赖b，b依赖a。包括传递的依
赖，比如a依赖b，b依赖c，c依赖a
                    if (isDependent(beanName, dep)) {
                        throw new
BeanCreationException(mbd.getResourceDescription(), beanName,
                                "Circular depends-on relationship
between '" + beanName + "' and '" + dep + "'");
                    }
                    // 注册依赖关系
                    registerDependentBean(dep, beanName);
                    try {
                        // 初始化依赖的bean
                        getBean(dep);
                    }
                    catch (NoSuchBeanDefinitionException ex) {
                        throw new
BeanCreationException(mbd.getResourceDescription(), beanName,
                                "'" + beanName + "' depends on
missing bean '" + dep + "'", ex);
                    }
                }
            }

            // Create bean instance.
            // 如果是单例
            if (mbd.isSingleton()) {
                sharedInstance = getSingleton(beanName, () -> {
                    try {
                        // 创建bean
```

```
                        return createBean(beanName, mbd, args);
                    }
                    catch (BeansException ex) {
                        // Explicitly remove instance from singleton
cache: It might have been put there
                        // eagerly by the creation process, to allow
for circular reference resolution.
                        // Also remove any beans that received a
temporary reference to the bean.
                        destroySingleton(beanName);
                        throw ex;
                    }
                });
                // 如果是普通bean，直接返回，是FactoryBean，返回他的
getObject
                bean = getObjectForBeanInstance(sharedInstance,
name, beanName, mbd);
            }

            else if (mbd.isPrototype()) {
                // It's a prototype -> create a new instance.
                Object prototypeInstance = null;
                try {
                    // 加入prototypesCurrentlyInCreation，说明正在创
建
                    beforePrototypeCreation(beanName);
                    //创建bean
                    prototypeInstance = createBean(beanName, mbd,
args);
                }
                finally {
                    // 移除prototypesCurrentlyInCreation，说明已经创
建结束
                    afterPrototypeCreation(beanName);
                }
                // 如果是普通bean，直接返回，是FactoryBean，返回他的
getObject
                bean = getObjectForBeanInstance(prototypeInstance,
name, beanName, mbd);
            }
```

```
            else {
                String scopeName = mbd.getScope();
                final Scope scope = this.scopes.get(scopeName);
                if (scope == null) {
                    throw new IllegalStateException("No Scope
registered for scope name '" + scopeName + "'");
                }
                try {
                    Object scopedInstance = scope.get(beanName, () -
> {
                        beforePrototypeCreation(beanName);
                        try {
                            return createBean(beanName, mbd, args);
                        }
                        finally {
                            afterPrototypeCreation(beanName);
                        }
                    });
                    bean = getObjectForBeanInstance(scopedInstance,
name, beanName, mbd);
                }
                catch (IllegalStateException ex) {
                    throw new BeanCreationException(beanName,
                            "Scope '" + scopeName + "' is not active
for the current thread; consider " +
                            "defining a scoped proxy for this bean
if you intend to refer to it from a singleton",
                            ex);
                }
            }
        }
        catch (BeansException ex) {
            cleanupAfterBeanCreationFailure(beanName);
            throw ex;
        }
    }

    // Check if required type matches the type of the actual bean
instance.
    if (requiredType != null && !requiredType.isInstance(bean)) {
        try {
```

```
            T convertedBean =
getTypeConverter().convertIfNecessary(bean, requiredType);
            if (convertedBean == null) {
                throw new BeanNotOfRequiredTypeException(name,
requiredType, bean.getClass());
            }
            return convertedBean;
        }
        catch (TypeMismatchException ex) {
            if (logger.isTraceEnabled()) {
                logger.trace("Failed to convert bean '" + name + "'
to required type '" +
                        ClassUtils.getQualifiedName(requiredType) +
"'", ex);
            }
            throw new BeanNotOfRequiredTypeException(name,
requiredType, bean.getClass());
        }
    }
    return (T) bean;
}
```

**1. 处理名称，检查缓存**

- 这一步会处理别名，将别名解析为实际名称
- 对 FactoryBean 也会特殊处理，如果以 & 开头表示要获取 FactoryBean 本身，否则表示要获取其产品
- 这里针对单例对象会检查一级、二级、三级缓存
  - singletonFactories 三级缓存，存放单例工厂对象
  - earlySingletonObjects 二级缓存，存放单例工厂的产品对象
    - 如果发生循环依赖，产品是代理；无循环依赖，产品是原始对象
  - singletonObjects 一级缓存，存放单例成品对象

**2. 处理父子容器**

- 如果当前容器根据名字找不到这个 bean，此时若父容器存在，则执行父容器的 getBean 流程
- 父子容器的 bean 名称可以重复

### 3. 处理 dependsOn

- 如果当前 bean 有通过 dependsOn 指定了非显式依赖的 bean，这一步会提前创建这些 dependsOn 的 bean
- 所谓非显式依赖，就是指两个 bean 之间不存在直接依赖关系，但需要控制它们的创建先后顺序

### 4. 选择 scope 策略

- 对于 singleton scope，首先到单例池去获取 bean，如果有则直接返回，没有再进入创建流程
- 对于 prototype scope，每次都会进入创建流程
- 对于自定义 scope，例如 request，首先到 request 域获取 bean，如果有则直接返回，没有再进入创建流程

### 5.1 创建 bean - 创建 bean 实例

| 要点 | 总结 |
| --- | --- |
| AutowiredAnnotationBeanPostProcessor | ① 优先选择带 @Autowired 注解的构造；② 若有唯一的带参构造，也会入选 |
| 采用默认构造 | 如果上面的后处理器和 BeanDefiniation 都没找到构造，采用默认构造，即使是私有的 |

### 5.2 创建 bean - 依赖注入

| 要点 | 总结 |
| --- | --- |
| AutowiredAnnotationBeanPostProcessor | 识别 @Autowired 及 @Value 标注的成员，封装为 InjectionMetadata 进行依赖注入 |
| CommonAnnotationBeanPostProcessor | 识别 @Resource 标注的成员，封装为 InjectionMetadata 进行依赖注入 |
| AUTOWIRE_BY_NAME | 根据成员名字找 bean 对象，修改 mbd 的 propertyValues，不会考虑简单类型的成员 |
| AUTOWIRE_BY_TYPE | 根据成员类型执行 resolveDependency 找到依赖注入的值，修改 mbd 的 propertyValues |

---

| 要点 | 总结 |
| --- | --- |
| applyPropertyValues | 根据 mbd 的 propertyValues 进行依赖注入（即xml中 `<property name ref\|value/>`） |

### 5.3 创建 bean - 初始化

| 要点 | 总结 |
| --- | --- |
| 内置 Aware 接口的装配 | 包括 BeanNameAware，BeanFactoryAware 等 |
| 扩展 Aware 接口的装配 | 由 ApplicationContextAwareProcessor 解析，执行时机在 postProcessBeforeInitialization |
| @PostConstruct | 由 CommonAnnotationBeanPostProcessor 解析，执行时机在 postProcessBeforeInitialization |
| InitializingBean | 通过接口回调执行初始化 |
| initMethod | 根据 BeanDefinition 得到的初始化方法执行初始化，即 `<bean init-method>` 或 @Bean(initMethod) |
| 创建 aop 代理 | 由 AnnotationAwareAspectJAutoProxyCreator 创建，执行时机在 postProcessAfterInitialization |

### 5.4 创建 bean - 注册可销毁 bean

在这一步判断并登记可销毁 bean

- 判断依据
  - 如果实现了 DisposableBean 或 AutoCloseable 接口，则为可销毁 bean
  - 如果自定义了 destroyMethod，则为可销毁 bean
  - 如果采用 @Bean 没有指定 destroyMethod，则采用自动推断方式获取销毁方法名（close，shutdown）
  - 如果有 @PreDestroy 标注的方法
- 存储位置
  - singleton scope 的可销毁 bean 会存储于 beanFactory 的成员当中
  - 自定义 scope 的可销毁 bean 会存储于对应的域对象当中
  - prototype scope 不会存储，需要自己找到此对象销毁

- 存储时都会封装为 DisposableBeanAdapter 类型对销毁方法的调用进行适配

**6. 类型转换处理**

- 如果 getBean 的 requiredType 参数与实际得到的对象类型不同，会尝试进行类型转换

**7. 销毁 bean**

- 销毁时机
  - singleton bean 的销毁在 ApplicationContext.close 时，此时会找到所有 DisposableBean 的名字，逐一销毁
  - 自定义 scope bean 的销毁在作用域对象生命周期结束时
  - prototype bean 的销毁可以通过自己手动调用 AutowireCapableBeanFactory.destroyBean 方法执行销毁
- 同一 bean 中不同形式销毁方法的调用次序
  - 优先后处理器销毁，即 @PreDestroy
  - 其次 DisposableBean 接口销毁
  - 最后 destroyMethod 销毁（包括自定义名称，推断名称，AutoCloseable 接口 多选一）

## 1.13 spring中的循环引用

> 难易程度：☆☆☆☆☆
>
> 出现频率：☆☆☆

### 1.13.1 什么是Spring的循环依赖？

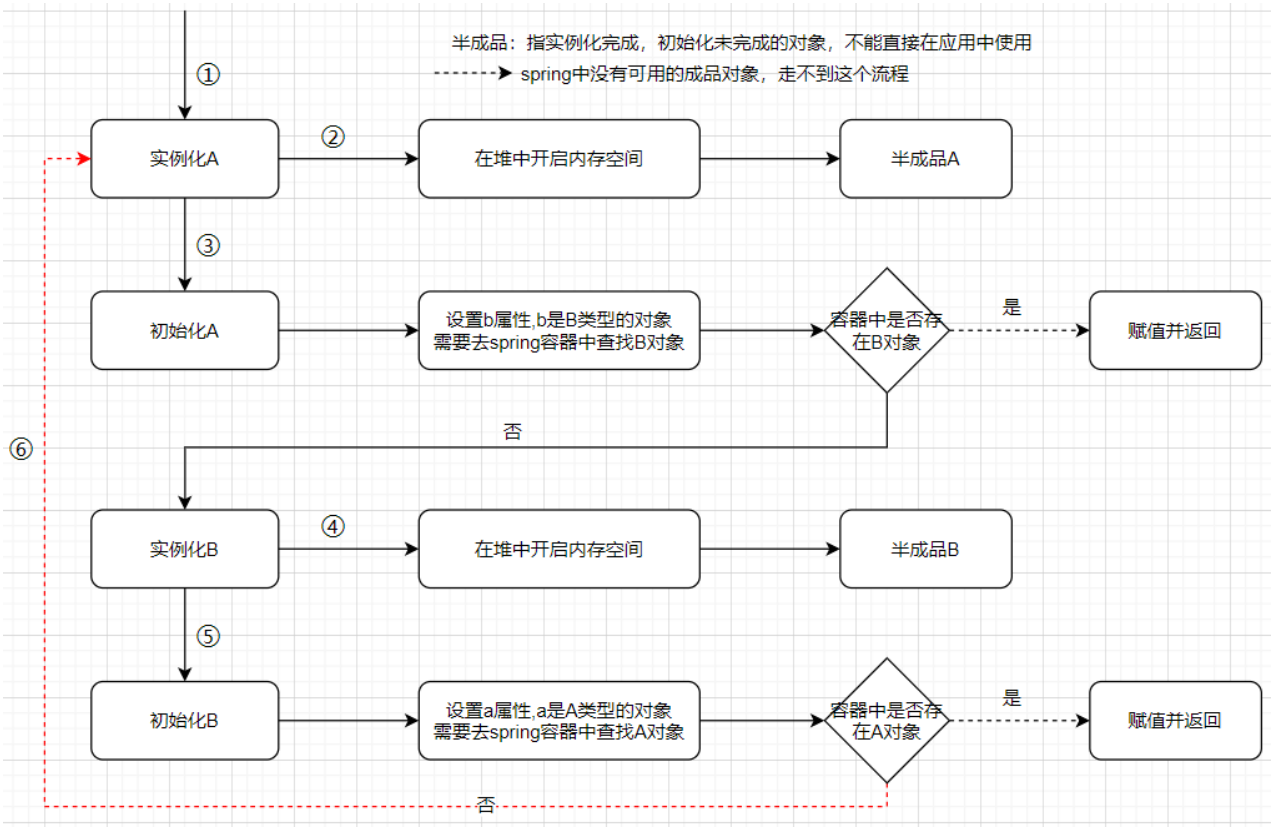
简单的来说就是A依赖B的同时，B依赖A。在创建A对象的同时需要使用的B对象，在创建B对象的同时需要使用到A对象。如下代码所示：

```
@Component
public class A {

    public A(){
        System.out.println("A的构造方法执行了...");
```

```
    }

    private B b;

    @Autowired
    public void setB(B b) {
        this.b = b;
        System.out.println("给A注入B");
    }
}
@Component
public class B {

    public B(){
        System.out.println("B的构造方法执行了...");
    }

    private A a;

    @Autowired
    public void setA(A a) {
        this.a = a;
        System.out.println("给B注入了A");
    }

}
```

### 1.13.2 出现循环依赖以后会有什么问题？

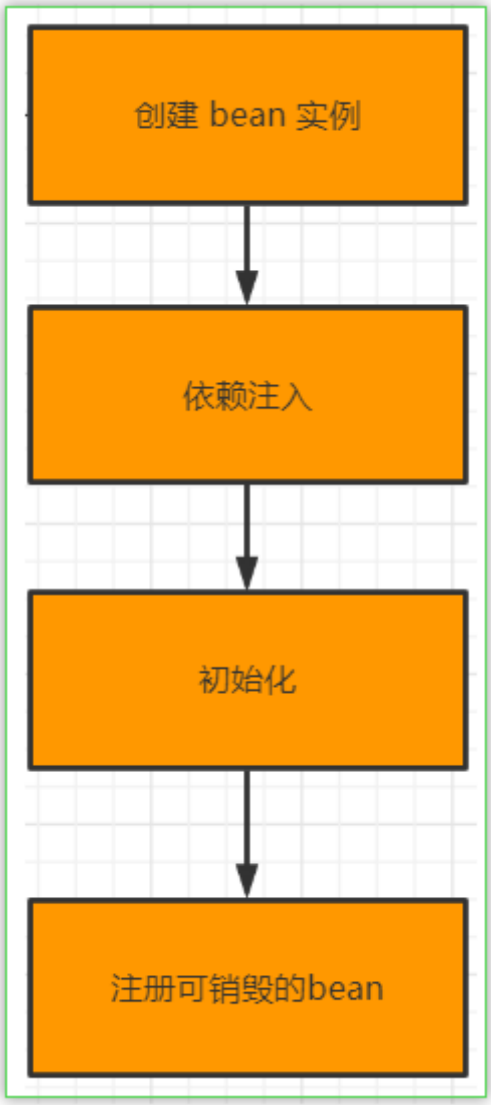
对象的创建过程会产生死循环，如下所示：



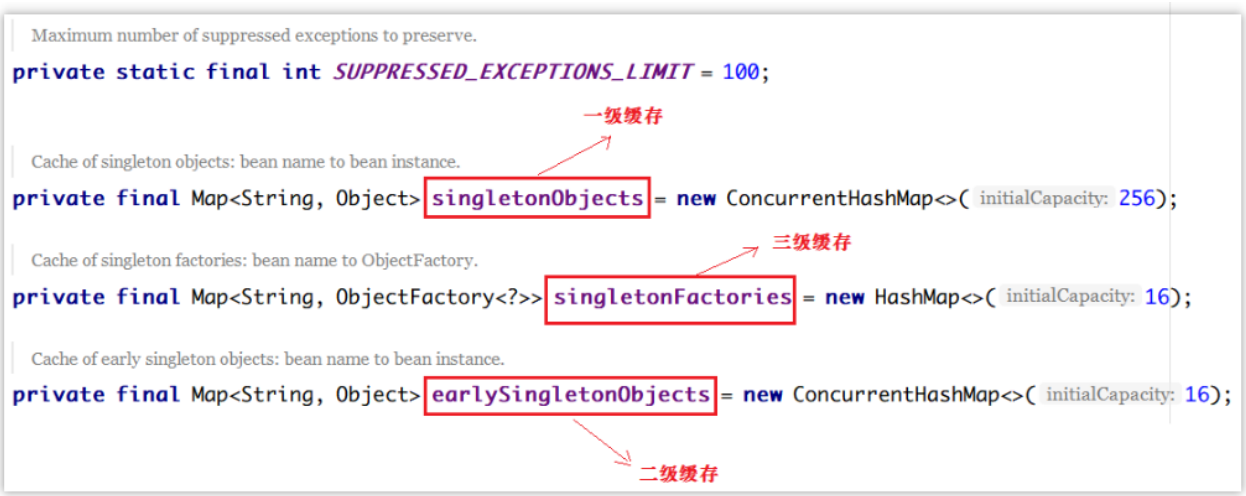

在spring中通过某些机制(三级缓存)帮开发者解决了部分循环依赖的问题。

### 1.13.3 spring如何解决循环依赖的？

spring的Bean生命周期创建bean的过程回顾：



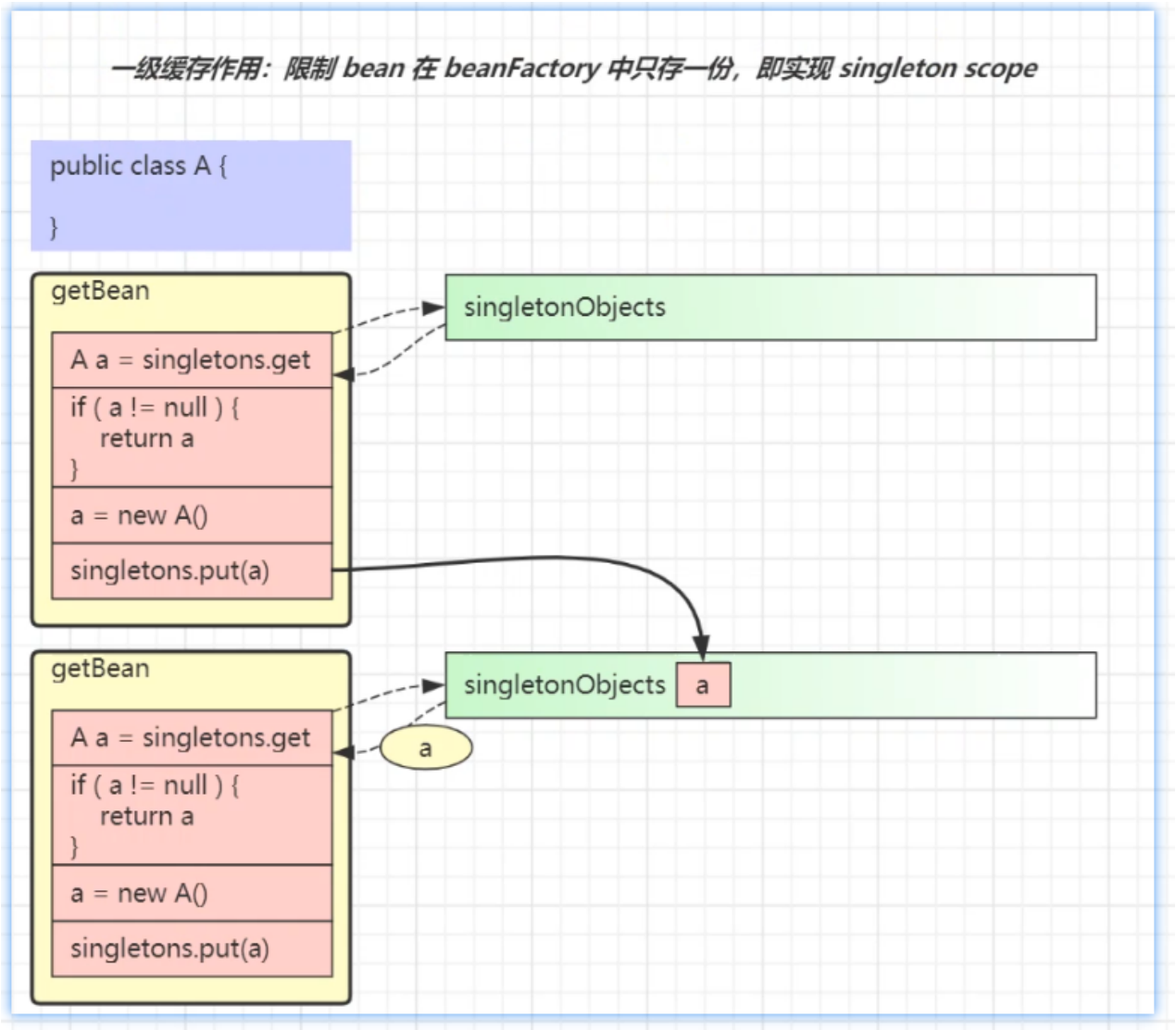

Spring解决循环依赖是通过三级缓存，对应的三级缓存如下所示：



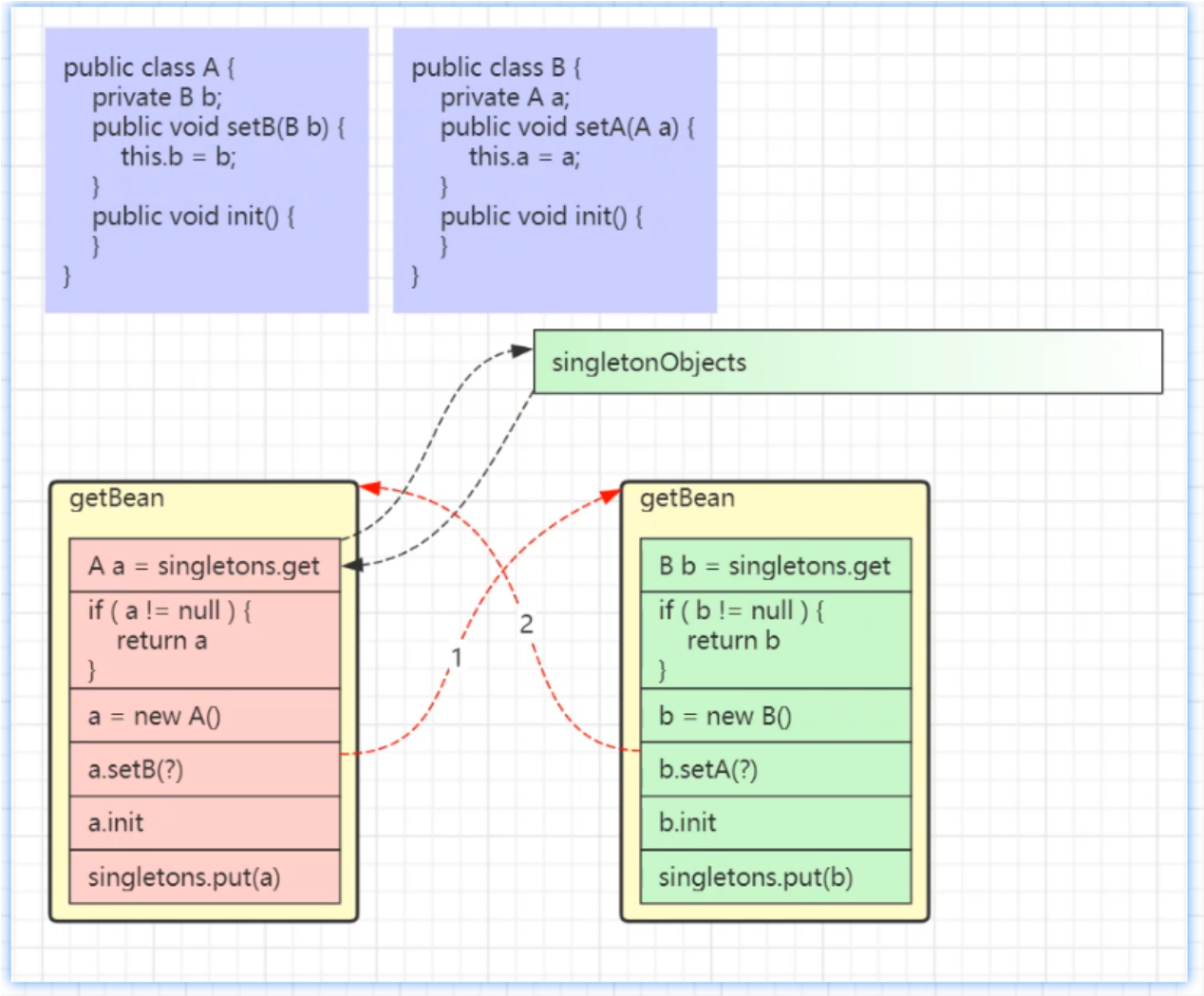

| 缓存 | 源码名称 | 作用 |
| --- | --- | --- |
| 一级缓存 | singletonObjects | 单例池; 缓存已经经历了完整声明周期, 已经初始化完成的bean对象 |

| 缓存 | 源码名称 | 作用 |
| --- | --- | --- |
| 二级缓存 | earlySingletonObjects | 缓存早期的bean对象(生命周期还没有走完) |
| 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 |

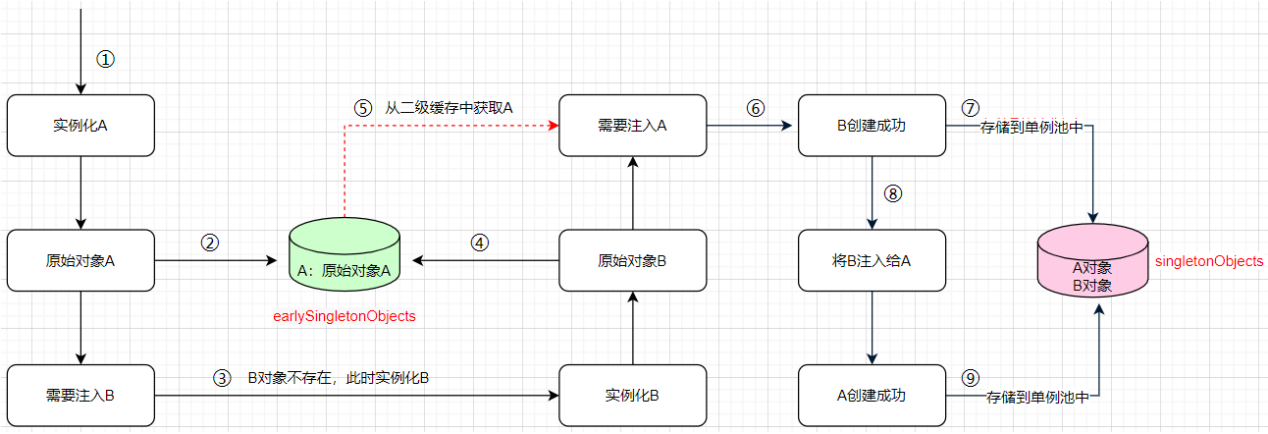
一级缓存作用：




一级缓存解决不了循环依赖问题




二级缓存的作用：如果要想打破上述的循环，就需要一个中间人的参与，这个中间人就是缓存。




步骤如下所示：

1、实例化A得到A的原始对象

2、将A的原始对象存储到二级缓存(earlySingletonObjects)中

3、需要注入B，B对象在一级缓存中不存在，此时实例化B，得到原始对象B

4、将B的原始对象存储到二级缓存中

5、需要注入A，从二级缓存中获取A的原始对象

6、B对象创建成功

7、将B对象加入到一级缓存中

8、将B注入给A，A创建成功

9、将A对象添加到一级缓存中

三级缓存的作用：

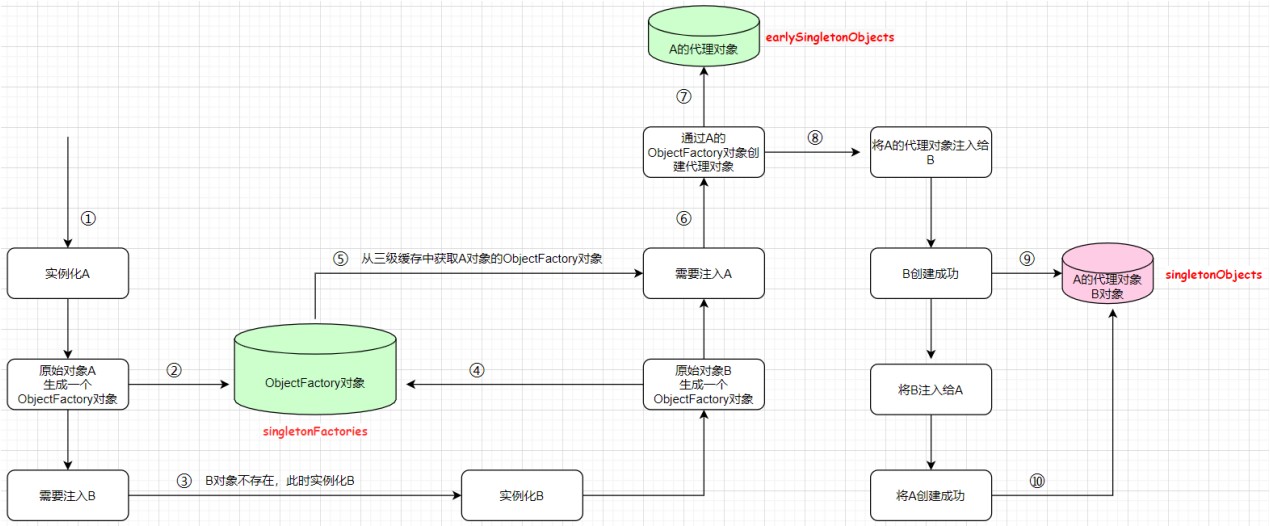
从上面这个分析过程中可以得出，只需要一个缓存就能解决循环依赖了，那么为什么Spring中还需要singletonFactories ？

基于上面的场景想一个问题：如果A的原始对象注入给B的属性之后，A的原始对象进行了AOP产生了一个代理对象，此时就会出现，对于A而言，它的Bean对象其实应该是AOP之后的代理对象，而B的a属性对应的并不是AOP之后的代理对象，这就产生了冲突。 也就是说， 最终单例池中存放的A对象（代理对象）和B依赖的A对象不是同一个。

所以在该场景下， 上述提到的二级缓存就解决不了了。那这个时候Spring就利用了第三级缓存**singletonFactories**来解决这个问题。

singletonFactories中存的是某个beanName对应的**ObjectFactory**，在bean的生命周期中，生成完原始对象之后，就会构造一个ObjectFactory存入singletonFactories中，后期其他的Bean可以通过调用该ObjectFactory对象的getObject方法获取对应的Bean。

整体的解决循环依赖问题的思路如下所示：




步骤如下所示：

1、实例化A，得到原始对象A，并且同时生成一个原始对象A对应的ObjectFactory对象

2、将ObjectFactory对象存储到三级缓存中

3、需要注入B，发现B对象在一级缓存和二级缓存都不存在，并且三级缓存中也不存在B对象所对应的ObjectFactory对象

4、实例化B，得到原始对象B，并且同时生成一个原始对象B对应的ObjectFactory对象，然后将该ObjectFactory对象也存储到三级缓存中

5、需要注入A，发现A对象在一级缓存和二级缓存都不存在，但是三级缓存中存在A对象所对应的ObjectFactory对象

6、通过A对象所对应的ObjectFactory对象创建A对象的代理对象

7、将A对象的代理对象存储到二级缓存中

8、将A对象的代理对象注入给B，B对象执行后面的生命周期阶段，最终B对象创建成功

9、将B对象存储到一级缓存中

10、将B对象注入给A，A对象执行后面的生命周期阶段，最终A对象创建成功，将二级缓存的A的代理对象存储到一级缓存中

注意：

1、后面的生命周期阶段会按照本身的逻辑进行AOP, 在进行AOP之前会判断是否已经进行了AOP，如果已经进行了AOP就不会进行AOP操作了。

2、singletonFactories : 缓存的是一个ObjectFactory，主要用来去生成原始对象进行了AOP之后得到的代理对象，在每个Bean的生成过程中，都会提前暴露一个工厂，这个工厂可能用到，也可能用不到，如果没有出现循环依赖依赖本bean，那么这个工厂无用，本bean按照自己的生命周期执行，执行完后直接把本bean放入singletonObjects中即可，如果出现了循环依赖依赖了本bean，则另外那个bean执行ObjectFactory提交得到一个AOP之后的代理对象(如果没有AOP，则直接得到一个原始对象)。

### 1.13.4 只有一级缓存和三级缓存是否可行？

不行，每次从三级缓存中拿到ObjectFactory对象，执行getObject()方法又会产生**新的代理对象**，因为A是单例的，所有这里我们要借助二级缓存来解决这个问题，将执行了objectFactory.getObject()产生的对象放到二级缓存中去，后面去二级缓存中拿，没必要再执行一遍objectFactory.getObject()方法再产生一个新的代理对象，保证始终只有一个代理对象。

总结：所以如果没有AOP的话确实可以两级缓存就可以解决循环依赖的问题，如果加上AOP，两级缓存是无法解决的，不可能每次执行objectFactory.getObject()方法都给我产生一个新的代理对象，所以还要借助另外一个缓存来保存产生的代理对象。

### 1.13.5 构造方法出现了循环依赖怎么解决？

Spring中大部分的循环依赖已经帮助我们解决掉了，但是有一些循环依赖还需要我们程序员自己进行解决。如下所示：

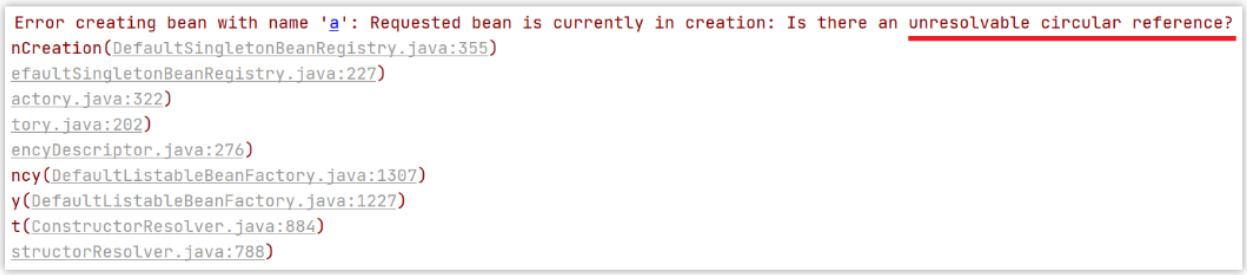
```
@Component
public class A {
```

```
    // B成员变量
    private B b;

    public A(B b){
        System.out.println("A的构造方法执行了...");
        this.b = b ;
    }
}

@Component
public class B {

    // A成员变量
    private A a;

    public B(A a){
        System.out.println("B的构造方法执行了...");
        this.a = a ;
    }

}
```

main方法程序：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SpringConfig.class)
public class AccountServiceTest {

    @Autowired
    private A a ;

    @Test
    public void testTransfer() throws Exception {
        System.out.println(a);
    }

}
```

控制台输出：




解决方案：使用@Lazy注解

```
@Component
public class A {

    // B成员变量
    private B b;

    public A(@Lazy B b){
        System.out.println("A的构造方法执行了...");
        this.b = b ;
    }
}
```

在构造参数前面加了@Lazy注解之后, 就不会真正的注入真实对象, 该注入对象会被延迟加载， 此时注入的是一个代理对象 。

# 2 SpringMVC

## 2.1 Spring MVC中的拦截器和Servlet中的filter有什么区别？

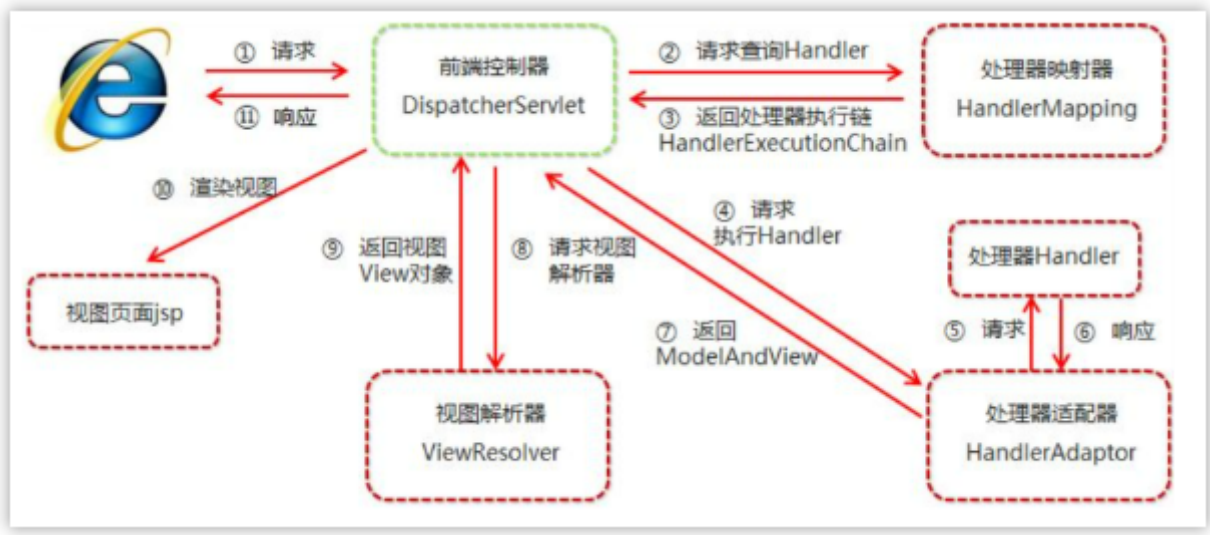
> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

过滤器：依赖于servlet容器，在实现上基于函数回调，可以对几乎**所有请求**进行过滤

拦截器：依赖于web框架，在SpringMVC中就是依赖于SpringMVC框架，属于面向切面编程（AOP）的一种运用。

## 2.2 SpringMVC的执行流程知道嘛

> 难易程度：☆☆☆☆☆
>
> 出现频率：☆☆☆




具体流程如下所示：

1、用户发送出请求到前端控制器DispatcherServlet。

2、DispatcherServlet收到请求调用HandlerMapping（处理器映射器）。

3、HandlerMapping找到具体的处理器(可查找xml配置或注解配置)，生成处理器对象及处理器拦截器(如果有)，再一起返回给DispatcherServlet。

4、DispatcherServlet调用HandlerAdapter（处理器适配器）。

5、HandlerAdapter经过适配调用具体的处理器（Handler/Controller）。

---

6、Controller执行完成返回ModelAndView对象。

7、HandlerAdapter将Controller执行结果ModelAndView返回给DispatcherServlet。

8、DispatcherServlet将ModelAndView传给ViewReslover（视图解析器）。

9、ViewReslover解析后返回具体View（视图）。

10、DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。

11、DispatcherServlet响应用户。

## 2.3 Spring MVC常用的注解有哪些？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

1、@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径。

2、@RequestBody：注解实现接收http请求的json数据，将json转换为java对象。

3、@RequestParam：指定请求参数的名称

4、@PathViriable：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

5、@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。

6、@RequestHeader：获取指定的请求头数据

## 2.4 Spring MVC怎么处理异常？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

可以直接使用Spring MVC中的全局异常处理器对异常进行统一处理，此时Contoller方法只需要编写业务逻辑代码，不用考虑异常处理代码。

开发一个全局异常处理器需要使用到两个注解：@Controlleradvice 、@ExceptionHandler

如下所示：

```
@ControllerAdvice
public class ExceptionAdvice {

    @ExceptionHandler(BusinessException.class)
    public ResponseEntity handlerException(BusinessException be) {
        be.printStackTrace();
        ErrorResult errorResult = be.getErrorResult();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(errorResult);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity handlerException1(Exception be) {
        be.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(ErrorResult.error());
    }
}
```

# 3 Mybatis

## 3.1 Mybatis #{}和${}的区别

> 难易程度：☆☆
>
> 出现频率：☆☆☆☆☆

1、#{}是预编译处理，${}是字符串替换。

2、Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；

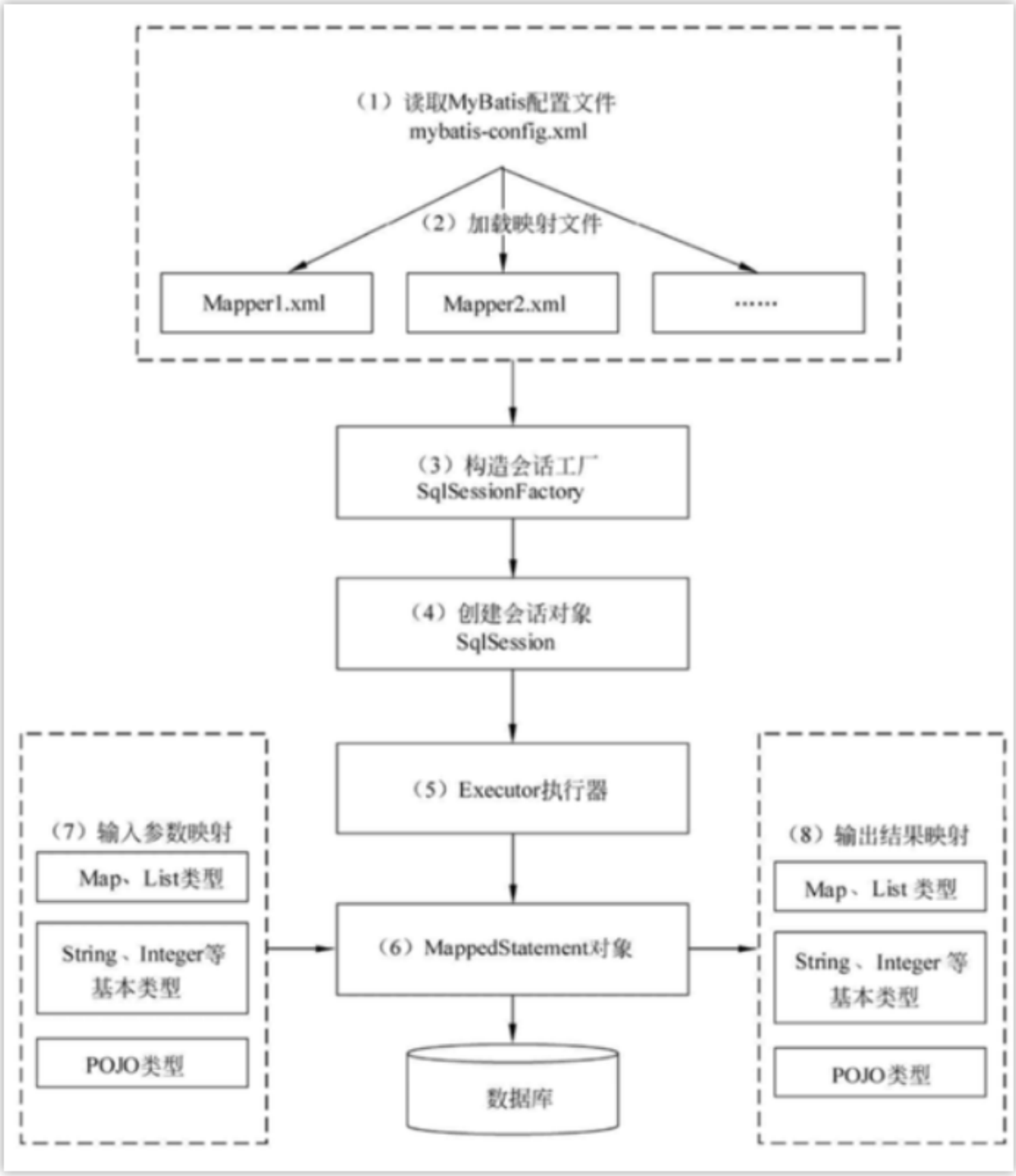
3、Mybatis在处理${}时，就是把${}替换成变量的值。

4、使用#{}可以有效的防止SQL注入，提高系统安全性。

## 3.2 MyBatis执行流程

> 难易程度：☆☆☆☆

出现频率：☆☆




1、读取 MyBatis 配置文件：mybatis-config.xml 为 MyBatis 的全局配置文件，配置了 MyBatis 的运行环境等信息，例如数据库连接信息。

2、加载映射文件。映射文件即 SQL 映射文件，该文件中配置了操作数据库的 SQL 语句，需要在MyBatis 配置文件 mybatis-config.xml 中加载。

mybatis-config.xml 文件可以加载多个映射文件，每个文件对应数据库中的一张表。

3、构造会话工厂：通过 MyBatis 的环境等配置信息构建会话工厂 SqlSessionFactory。

4、创建会话对象：由会话工厂创建 SqlSession 对象，该对象中包含了执行 SQL 语句的所有方法。

5、Executor执行器：MyBatis底层定义了一个Executor 接口来操作数据库，它将根据 SqlSession 传递的参数动态地生成需要执行的 SQL 语句，同时负责查询缓存的维护。

6、MappedStatement 对象：在 Executor 接口的执行方法中有一个 MappedStatement 类型的参数，该参数是对映射信息的封装，用于存储要映射的 SQL 语句的 id、参数等信息。

7、输入参数映射：输入参数类型可以是 Map、List 等集合类型，也可以是基本数据类型和 POJO 类型。输入参数映射过程类似于 JDBC 对 preparedStatement 对象设置参数的过程。

8、输出结果映射：输出结果类型可以是 Map、 List 等集合类型，也可以是基本数据类型和 POJO 类型。输出结果映射过程类似于 JDBC 对结果集的解析过程。

## 3.3 Mybatis 如何获取生成的主键

> 难易程度：☆☆☆
>
> 出现频率：☆☆

使用insert标签中的useGeneratedKeys和keyProperty 属性。使用方式如下所示：

```
<insert id = "saveUser" useGeneratedKeys = "true" keyProperty="id">
    insert into tb_user(user_name,password,gender,addr) values (#
{username},#{password},#{gender},#{addr})
</insert>
```

属性说明：

1、useGeneratedKeys：是够获取自动增长的主键值。true表示获取。

2、keyProperty ：指定将获取到的主键值封装到哪儿个属性里

## 3.4 当实体类中的属性名和表中的字段名不一样，怎么办

> 难易程度：☆☆☆
>
> 出现频率：☆☆

第1种： 通过在查询的SQL语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。

```
<select id="getOrder" parameterType="int"
resultType="com.heima.pojo.Order">
    select order_id id,order_no orderno,order_price price from
orders
</select>
```

第2种： 通过`<resultMap>`来映射字段名和实体类属性名的一一对应的关系。

```
<select id="getOrder" parameterType="int" resultMap =
"orderResultMap">
    select order_id id,order_no orderno,order_price price from
orders
</select>
<resultMap type="com.heima.pojo.Order" id="orderResultMap">
    <!-- 用id属性来映射主键字段 -->
    <id property="id" column="order_id">

    <!-- 用result属性来映射非主键字段，property为实体类属性名，column为
数据库表中的属性 -->
    <result property ="orderno" column="order_no" />
    <result property ="price" column="order_price" />
</resultMap>
```

第3种，开启mybatis驼峰命名自动匹配映射

```
<settings>
    <setting name="mapUnderscoreToCamelCase" value="true" /> <!-- 开
启驼峰命名自动映射 -->
</settings>
```

## 3.5 Mybatis如何实现多表查询

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆

Mybatis是新多表查询的方式也有二种：

第一种是：编写多表关联查询的SQL语句，使用ResultMap建立结果集映射，在ResultMap中建立多表结果集映射的标签有 `association` 和 `collection`

```
<resultMap id="Account_User_Map" type="com.heima.entity.Account">
    <id property="id" column="id"></id>
    <result property="uid" column="uid"></result>
    <result property="money" column="money"></result>

    <association property="user">
        <id property="id" column="uid"></id>
        <result property="username" column="username"></result>
        <result property="birthday" column="birthday"></result>
        <result property="sex" column="sex"></result>
        <result property="address" column="address"></result>
    </association>

</resultMap>

<!--public Account findByIdWithUser(Integer id);-->
<select id="findByIdWithUser" resultMap="Account_User_Map">
    select  a.*,  username, birthday, sex, address  from account a , user u where a.UID = u.id and a.ID = #{id} ;
</select>
```

第二种是：将多表查询分解为多个单表查询，使用ResultMap表的子标签 `association` 和 `collection` 标签的 `select` 属性指定另外一条SQL的定义去执行，然后执行结果会被自动封装

```
<resultMap id="Account_User_Map" type="com.heima.entity.Account"
autoMapping="true">
    <id property="id" column="id"></id>

    <association property="user"
select="com.heima.dao.UserDao.findById" column="uid"
fetchType="lazy"></association>
</resultMap>

<!--public Account findByIdWithUser(Integer id);-->
<select id="findByIdWithUser" resultMap="Account_User_Map">
    select * from account where  id = #{id}
</select>
```

## 3.6 Mybatis都有哪些动态sql？能简述一下动 态sql的执行原理吗？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆

Mybatis动态sql可以让我们在Xml映射文件内，以标签的形式编写动态sql，完成逻辑判断和动态 拼接sql的功能，Mybatis提供了9种动态sql标签 trim|where|set|foreach|if|choose|when|otherwise|bind。

其执行原理为，使用OGNL从sql参数对象中计算表达式的值，根据表达式的值动态拼接sql，以此 来完成动态sql的功能。

## 3.7 Mybatis是否支持延迟加载？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

Mybatis仅支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载**lazyLoadingEnabled**=true|false。

```
<settings>
    <setting name = "lazyLoadingEnabled" value = "true"> <!-- 开启延迟加载 -->
</settings>
```

默认情况下延迟加载是关闭的。

实现原理：

它的原理是，使用CGLIB创建**目标对象的代理对象**，当调用目标方法时，进入拦截器方法，比如调用order.getUser().getUserName()，拦截器invoke()方

法发现order.getUser()是null值，那么就会单独发送事先保存好的查询关联User对象的sql，把User查询上来，然后调用order.setUser(user)，于是order的对象user属性就有值了，接着完成order.getUser().getUserName()方法的调用。

## 3.8 如何使用Mybatis实现批量插入？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

批量插入数据：

1、mybatis的接口方法参数需要定义为集合类型List

```
public abstract void saveUsers(List<User> users);
```

2、在映射文件中通过forEach标签遍历集合，获取每一个元素作为insert语句的参数值

```xml
<!-- 批量插入用户 -->
<insert id="savaUsers" parameterType="java.util.List">
    insert into tb_user(user_name,password)
    values
    <foreach
collection="list",item="user",index="index",separator=",">
        (#{user.userName},#{user.password})
    </foreach>
</insert>
```

## 3.9 Mybatis的一级、二级缓存？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

1、一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当Session进行flush或close之后，该Session中的所有Cache就将清空，默认打开一级缓存。如下所示：

```java
//1. 加载mybatis的核心配置文件，获取 SqlSessionFactory
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

//2. 获取SqlSession对象，用它来执行sql
SqlSession sqlSession = sqlSessionFactory.openSession();
//3. 执行sql
//3.1 获取UserMapper接口的代理对象
UserMapper userMapper1 = sqlSession.getMapper(UserMapper.class);
UserMapper userMapper2 = sqlSession.getMapper(UserMapper.class);
//根据id查询数据
User user1 = userMapper1.selectById(1);
User user2 = userMapper2.selectById(1);
//输出
System.out.println(user1);
System.out.println(user2);
//4. 释放资源
sqlSession.close();
```

使用同一个sqlSession对象获取两次UserMapper对象，进行了两次用户数据的查询。控制台的输出结果如下所示：

```
[DEBUG]  [main] c.i.m.U.selectById - ==>  Preparing: select * from tb_user where id = ?;
[DEBUG]  [main] c.i.m.U.selectById - ==> Parameters: 1(Integer)
[DEBUG]  [main] c.i.m.U.selectById - <==      Total: 1
User{id=1, username='zhangsan', password='123', gender='男', addr='北京'}
User{id=1, username='zhangsan', password='123', gender='男', addr='北京'}
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connect
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@27e47833]
[DEBUG]  [main] o.a.i.d.p.PooledDataSource - Returned connection 669284403 to pool.
```

只执行了一次sql语句。说明第二次查询的时候使用的是缓存数据。

2、二级缓存：二级缓存是基于namespace和mapper的作用域起作用的，不是依赖于SQL session，默认也是采用 PerpetualCache，HashMap 存储。

如下代码：

```java
//1. 加载mybatis的核心配置文件，获取 SqlSessionFactory
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
//2. 获取SqlSession对象，用它来执行sql
//开启第一个会话
SqlSession sqlSession1 = sqlSessionFactory.openSession();
//3. 执行sql
//3.1 获取UserMapper接口的代理对象
UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);
User user1 = userMapper1.selectById(1);
System.out.println(user1);
sqlSession1.close();

//开启第二个会话
SqlSession sqlSession2 = sqlSessionFactory.openSession();

UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);
//根据id查询数据
User user2 = userMapper2.selectById(1);
System.out.println(user2);
sqlSession2.close();
```

当执行完sqlSession1.close()方法时一级缓存就一斤被清空掉了。再次获取了一个新的sqlSession对象，那么此时就需要再次查询数据，因此控制台的输

出如下所示：

```
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@27e47833]
[DEBUG]  [main] c.i.m.U.selectById - ==> Preparing: select * from tb_user where id = ?;
[DEBUG]  [main] c.i.m.U.selectById - ==> Parameters: 1(Integer)
[DEBUG]  [main] c.i.m.U.selectById - <==      Total: 1
User{id=1, username='zhangsan', password='123', gender='男', addr='北京'}
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@27e47833]
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@27e47833]
[DEBUG]  [main] o.a.i.d.p.PooledDataSource - Returned connection 669284403 to pool.
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Opening JDBC Connection
[DEBUG]  [main] o.a.i.d.p.PooledDataSource - Checked out connection 669284403 from pool.
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@27e47833]
[DEBUG]  [main] c.i.m.U.selectById - ==> Preparing: select * from tb_user where id = ?;
[DEBUG]  [main] c.i.m.U.selectById - ==> Parameters: 1(Integer)
[DEBUG]  [main] c.i.m.U.selectById - <==      Total: 1
User{id=1, username='zhangsan', password='123', gender='男', addr='北京'}
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@27e47833]
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@27e47833]
[DEBUG]  [main] o.a.i.d.p.PooledDataSource - Returned connection 669284403 to pool.
```

可以看到进行了两次查询。

默认情况下二级缓存并没有开启，要想使用二级缓存，那么就需要开启二级缓存，如下所示：

① 全局配置文件

```
<settings>
    <setting name="cacheEnabled" value="true"/> <!-- 开启二级缓存 -->
</settings>
```

② 映射文件

使用 `<cache/>` 标签让当前mapper生效二级缓存

```
<mapper namespace="com.itheima.mapper.UserMapper">

    <cache/>  <!-- 二级缓存生效 -->

    <select id="selectAll" resultType="user">
        select *
        from tb_user;
    </select>
    <select id="selectById" resultType="user">
        select *
        from tb_user where id = #{id};
```

```
    </select>

</mapper>
```

运行程序进行测试，控制台输出结果如下所示：

```
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@14fa86ae]
[DEBUG]  [main] c.i.m.U.selectById - ==> Preparing: select * from tb_user where id = ?;
[DEBUG]  [main] c.i.m.U.selectById - ==> Parameters: 1(Integer)
[DEBUG]  [main] c.i.m.U.selectById - <==      Total: 1
User{id=1, username='zhangsan', password='123', gender='男', addr='北京'}
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@14fa86ae]
[DEBUG]  [main] o.a.i.t.j.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@14fa86ae]
[DEBUG]  [main] o.a.i.d.p.PooledDataSource - Returned connection 351962798 to pool.
[DEBUG]  [main] c.i.m.UserMapper - Cache Hit Ratio [com.itheima.mapper.UserMapper]: 0.5
User{id=1, username='zhangsan', password='123', gender='男', addr='北京'}
```

只进行了一次查询，那么就说明数据已经进入到了二级缓存中。

3、对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行了新增、修改、删除操作后，默认该作用域下所有 select 中的缓存将被 clear。

注意事项：

1、二级缓存需要缓存的数据实现Serializable接口

2、只有会话提交或者关闭以后，一级缓存中的数据才会转移到二级缓存中

3、可自定义存储源，如 Ehcache。

# 4 面试现场

## 4.1 Spring

> 面试官：什么是Spring IOC 和DI ?
>
> 候选人：
>
> 嗯，这个spring框架最核心的部分，它是一个思想

控制反转(IOC)是指Spring容器使用了工厂模式为我们创建了所需要的对象，我们使用时不需要自己去创建，直接调用Spring为我们提供的对象即可，这就是控制反转的思想。

关于依赖注入也就是DI，它是Spring使用Java Bean对象的Set方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程就是依赖注入的基本思想。

**面试官**：Spring 有哪些不同类型的依赖注入实现方式？

**候选人**：

嗯，依赖注入共有两种类型，Setter方 法注入和构造器注入

构造器依赖注入：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每 个参数代表一个对其他类的依赖，在项目很少用。

在项目比较常用的是Setter方法注入，Setter方法注入是容器通过调用无参构造器或无参static工厂 方法实例化bean之 后，调用该bean的setter方法，即实现了基于setter的依赖注入。

**面试官**：Spring支持的几种bean的作用域?

**候选人**：

是这样的~，spring框架支持5中作用域。

第一个是singleton （单例）就是在每个Spring ioc 容器中只有一个实例。

第二个是prototype（多例）一个bean的定义可以有多个实例。

还有三个是在web的Spring ApplicationContext情形下有效。 代表不同的域，分别是

request域、Session域、还有一个是比较大的application域(应用域)

**面试官**：Spring框架中的单例bean是线程安全的吗？

**候选人**：

嗯！

不是线程安全的，是这样的

当多用户同时请求一个服务时，容器会给每一个请求分配一个线程，这是多个线程会并发执行该请求对应的业务逻辑（成员方法），如果该处理逻辑中有对该单列状态的修改（体现为该单例的成员属性），则必须考虑线程同步问题。

Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。

比如：我们通常在项目中使用的Spring bean都是不可可变的状态(比如Service类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。

如果你的bean有多种状态的话（比如 View Model对象），就需要自行保证线程安全。最浅显的解决办法就是将多态bean的作用由“**singleton**”变更为“**prototype**”。

**面试官**：Spring自动装配 bean 有哪些方式？

**候选人**：

嗯，我们一般项目中常见的是有这几种

第一是byName，一般是一个接口可能会有多种实现，这个时候需要指定名称才行。

第二是byType，就是根据类型自动到匹配注入，这个是在项目中用的比较多的

第三可以通过constructor ，构造器的自动装配和byType模式类似，通过构造器的参数进行注入

第四是default，该模式自动探测使用构造器自动装配或者byType自动装配

**面试官**：你用过哪些重要的Spring注解？

**候选人**：

嗯，这个就很多了

第一类是：声明bean，有@Component、@Service、@Repository、@Controller

第二类是：依赖注入相关的，有@Autowired、@Qualifier、@Resourse

第三类是：设置作用域 @Scope

第四类是：spring配置相关的，比如@Configuration，@ComponentScan 和 @Bean

第五类是：跟aop相关做增强的注解 @Aspect，@Before，@After，@Around，@Pointcut

**面试官**：Spring中的事务是如何实现的

**候选人**：

嗯，spring中提供了两种事务的控制

第一种是**编程式事务控制**：需要使用TransactionTemplate来进行实现，这种方式实现对业务代码有侵入性，因此在项目中很少被使用到。

第二种是**声明式事务管理**：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加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就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。

**面试官**：Spring中事务失效的场景？

**候选人**：

嗯，这个在项目中有很多种情况，我记得有

第一种情况是在业务层方法上抛出检查异常导致事务不能正确回滚

第二情况是在业务层方法内自己 try-catch 异常导致事务不能正确回滚

还有一种是非 public 方法导致的事务失效

**面试官**：说一下Spring的事务传播行为

候选人：

嗯！这个是spring框架对事务做的增强，一共有7个值

1. PROPAGATION_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就 加入该事务，该设置是最常用的设置。
2. PROPAGATION_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不 存在事务，就以非事务执行。
3. PROPAGATION_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前 不存在事务，就抛出异常。
4. PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。
5. PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前 事务挂起。
6. PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。
7. PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则 按REQUIRED属性执行。

不过，一般项目中用的较多的就是PROPAGATION_REQUIRED，如果复杂点的业务，需要在service的方法上的调用不同的业务可以设置PROPAGATION_REQUIRES_NEW。

面试官：什么是AOP，你们项目中有没有使用到AOP

候选人：

嗯，AOP也是spring的核心之一

AOP一般称为面向切面编程，作为面向对象的一种补充，用于 将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模 块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时 提高了系统的可维护性。

在我们的项目中我们自己写AOP的场景其实很少，但是我们使用的很多框架的功能底层都是AOP，例如

1、统一日志处理

2、spring中内置的事务处理

面试官：JDK动态代理和CGLIB动态代理的区别

候选人：

嗯，知道的

Spring中AOP底层的实现是基于**动态代理**进行实现的。

常见的动态代理技术有两种：JDK的动态代理和CGLIB。

- JDK动态代理只能对实现了接口的类生成代理，而不能针对类
- Cglib是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法进行增强，但是因为采用的是**继承**，所以该类或方法最好不要声明为final，对于final类或方法，是无法继承的。

**面试官**：好的，那Spring如何选择是用JDK还是cglib，这个清楚吗？

**候选人**：

嗯，是这样的

1、当bean实现接口时，会用JDK代理模式

2、当bean没有实现接口，会用cglib实现

3、可以强制使用cglib

**面试官**：spring的bean的生命周期

**候选人**：

嗯，这个步骤比较多，我想一下~~~

spring的bean生命周期共分为了7个阶段

1. 处理名称，检查缓存
2. 处理父子容器
3. 处理 dependsOn
4. 选择 scope 策略
5. 创建 bean（重点说）

这个阶段做了很重要的事情，实例化bean、依赖注入、初始化、注册可销毁 bean，都在这个阶段完成

- 实例化bean的过程中，优先选择带 @Autowired 注解的构造；其次是有唯一的带参构造，最后才是无参构造
- 依赖注入识别 根据我们常用的注解 @Autowired 、 @Value、@Resource 进行依赖注入，其次是根据成员名字和类型找 bean 对象注入，最后是，如果有配置文件

也会根据属性的配置进行注入

- 初始化阶段主要完成的是，如果bean实现了Aware 接口接口，则这些接口中回调方法需要执行。还有就是自定义的一些方法也会这里执行，比如使用注解@PostConstruct的方法。这里还有非常关键的一项，如果当前bean对象有增强，则也是这个阶段进行增强处理
- 注册可销毁的bean阶段，主要是bean是否实现了DisposableBean 等接口，则可以可销毁，还有就是如果自己实现了销毁方法，比如加了@PreDestroy的注解的方法，也可以销毁

6. 类型转换处理

如果 getBean 的 requiredType 参数与实际得到的对象类型不同，会尝试进行类型转换

7. 销毁 bean

不同作用域销毁的方式不同

- singleton bean 的销毁在 ApplicationContext.close 时，此时会找到所有 DisposableBean 的名字，逐一销毁
- 自定义 scope bean 的销毁在作用域对象生命周期结束时
- prototype bean 的销毁可以通过自己手动调用 AutowireCapableBeanFactory.destroyBean 方法执行销毁

**面试官：**什么是Spring的循环依赖？

**候选人：**

简单的来说就是A依赖B的同时，B依赖A。在创建A对象的同时需要使用的B对象，在创建B对象的同时需要使用到A对象

**面试官：**出现循环依赖以后会有什么问题？

**候选人：**

对象的创建过程会产生死循环。

根据spring的bean的生命周期，我们指导A是可以正常实例化的，但在初始化阶段需要注入B对象，这个时候spring会先实例化B，但是B在初始化的时候需要A，然而目前的A还没有创建完成，则会造成A等B创建，B等A创建，就会造成死循环。

**面试官：**spring如何解决循环依赖的？

候选人：

嗯，是这样的

Spring是通过三级缓存解决循环依赖。三级缓存分别是：

- 一级缓存 singletonObjects 单例池; 缓存已经经历了完整声明周期, 已经初始化完成的bean对象
- 二级缓存 earlySingletonObjects 缓存早期的bean对象(生命周期还没有走完)
- 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

整体流程大概这样的：

第一，先实例A对象，同时会创建ObjectFactory对象存入三级缓存singletonFactories

第二，A在初始化的时候需要B对象，这个走B的创建的逻辑

第三，B实例化完成，也会创建ObjectFactory对象存入三级缓存singletonFactories

第四，B需要注入A，通过三级缓存中获取ObjectFactory来生成一个A的对象同时存入二级缓存，这个是有两种情况，一个是可能是A的普通对象，另外一个是A的代理对象，都可以让ObjectFactory来生产对应的对象，这也是三级缓存的关键

第五，B通过从通过二级缓存earlySingletonObjects 获得到A的对象后可以正常注入，B创建成功，存入一级缓存singletonObjects

第六，回到A对象初始化，因为B对象已经创建完成，则可以直接注入B，A创建成功存入一次缓存singletonObjects

第七，二级缓存中的临时对象A清除

面试官：好的，那如果只有一级缓存和三级缓存是否可行？

候选人：

不行的，每次从三级缓存中拿到ObjectFactory对象，执行getObject()方法又会产生新的代理对象，因为A是单例的，所有这里我们要借助二级缓存来解决这个问题，将执行了objectFactory.getObject()产生的对象放到二级缓存中去，后面去二级缓存中拿，没必要再执行一遍objectFactory.getObject()方法

再产生一个新的代理对象，保证始终只有一个代理对象。

所以如果没有AOP的话确实可以两级缓存就可以解决循环依赖的问题，如果加上AOP，两级缓存是无法解决的，不可能每次执行objectFactory.getObject()方法都给我产生一个新的代理对象，所以还要借助另外一个缓存来保存产生的代理对象。

**面试官**：构造方法出现了循环依赖怎么解决？

**候选人**：

嗯，其实spring框架已经解决大部分的循环依赖问题，如果还产生了循环依赖，比如出现了构造注入导致循环依赖，则可以在构造参数前面加了@Lazy注解之后, 就不会真正的注入真实对象, 该注入对象会被延迟加载， 此时注入的是一个代理对象 。

## 4.2 SpringMVC

**面试官**：Spring MVC中的拦截器和Servlet中的filter有什么区别？

**候选人**：

嗯，它们两个功能是类似的

过滤器：依赖于servlet容器，在实现上基于函数回调，可以对几乎**所有请求**进行过滤

拦截器：依赖于web框架，在SpringMVC中就是依赖于SpringMVC框架，属于面向切面编程（AOP）的一种运用。

**面试官**：SpringMVC的执行流程知道嘛

**候选人**：

嗯，这个知道的，它分了好多步骤

1、用户发送出请求到前端控制器DispatcherServlet，这是一个调度中心

2、DispatcherServlet收到请求调用HandlerMapping（处理器映射器）。

3、HandlerMapping找到具体的处理器(可查找xml配置或注解配置)，生成处理器对象及处理器拦截器(如果有)，再一起返回给DispatcherServlet。

4、DispatcherServlet调用HandlerAdapter（处理器适配器）。

5、HandlerAdapter经过适配调用具体的处理器（Handler/Controller）。

6、Controller执行完成返回ModelAndView对象。

7、HandlerAdapter将Controller执行结果ModelAndView返回给DispatcherServlet。

8、DispatcherServlet将ModelAndView传给ViewReslover（视图解析器）。

9、ViewReslover解析后返回具体View（视图）。

10、DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。

11、DispatcherServlet响应用户。

当然现在的开发，基本都是前后端分离的开发的，并没有视图这些，一般都是handler中使用Response直接结果返回

**面试官：**Spring MVC常用的注解有哪些？

**候选人：**

嗯，这个也很多的

有@RequestMapping：用于映射请求路径；@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；@PathViriable：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，还有像@PostMapping、@GetMapping这些。

**面试官：**Spring MVC怎么处理异常？

**候选人：**

嗯，这个异常几乎每个项目都会用到的

可以直接使用Spring MVC中的全局异常处理器对异常进行统一处理，此时Contoller方法只需要编写业务逻辑代码，不用考虑异常处理代码。

开发一个全局异常处理器需要使用到两个注解：@Controlleradvice 、@ExceptionHandler

## 4.3 Mybatis

面试官：Mybatis #{}和${}的区别

候选人：

嗯，这个很常见，主要有这些不同吧。

1、#{}是预编译处理，${}是字符串替换。

2、Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；

3、Mybatis在处理${}时，就是把${}替换成变量的值。

4、使用#{}可以有效的防止SQL注入，提高系统安全性。

面试官：MyBatis执行流程

候选人：

好，这个知道的，不过步骤也很多

1、读取 MyBatis 配置文件：mybatis-config.xml 为 MyBatis 的全局配置文件，配置了 MyBatis 的运行环境等信息，例如数据库连接信息。

2、加载映射文件。映射文件即 SQL 映射文件，该文件中配置了操作数据库的 SQL 语句，需要在MyBatis 配置文件 mybatis-config.xml 中加载。

mybatis-config.xml 文件可以加载多个映射文件，每个文件对应数据库中的一张表。

3、构造会话工厂：通过 MyBatis 的环境等配置信息构建会话工厂 SqlSessionFactory。

4、创建会话对象：由会话工厂创建 SqlSession 对象，该对象中包含了执行 SQL 语句的所有方法。

5、Executor执行器：MyBatis底层定义了一个Executor 接口来操作数据库，它将根据 SqlSession 传递的参数动态地生成需要执行的 SQL 语句，同时负责查询缓存的维护。

6、MappedStatement 对象：在 Executor 接口的执行方法中有一个 MappedStatement 类型的参数，该参数是对映射信息的封装，用于存储要映射的 SQL 语句的 id、参数等信息。

7、输入参数映射：输入参数类型可以是 Map、List 等集合类型，也可以是基本数据类型和 POJO 类型。输入参数映射过程类似于 JDBC 对 preparedStatement 对象设置参数的过程。

8、输出结果映射：输出结果类型可以是 Map、 List 等集合类型，也可以是基本数据类型和 POJO 类型。输出结果映射过程类似于 JDBC 对结果集的解析过程。

现在我们都是使用springboot项目集成mybatis框架使用，像 SqlSessionFactory和SqlSession 这些框架都给封装了，我们现在需要关心主要都是mapper的编写

**面试官：**Mybatis 如何获取生成的主键

**候选人：**

嗯，这个在我们的项目也用过

可以在mapper的insert标签中的useGeneratedKeys和keyProperty 属性，其中useGeneratedKeys是够获取自动增长的主键值。keyProperty 指定将获取到的主键值封装到哪儿个属性里

**面试官：**当实体类中的属性名和表中的字段名不一样 ，怎么办

**候选人：**

嗯，这个也很常见，主要解决的话，有两种方案

第一是：通过在查询的SQL语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。

第二是：通过 `<resultMap>` 来映射字段名和实体类属性名的一一对应的关系。

**面试官：**Mybatis如何实现多表查询

**候选人：**

嗯，这个实现也有两种方案：

第一种是 : 编写多表关联查询的SQL语句 , 使用ResultMap建立结果集映射 , 在ResultMap中建立多表结果集映射的标签有 `association` 和 `collection`

第二种是 : 将多表查询分解为多个单表查询, 使用ResultMap表的子标签 `association` 和 `collection` 标签的 `select` 属性指定另外一条SQL的定义去执行, 然后执行结果会被自动封装

**面试官：**Mybatis都有哪些动态sql？能简述一下动态sql的执行原理吗？

**候选人：**

嗯，我思考一下~~

Mybatis动态sql可以让我们在Xml映射文件内，以标签的形式编写动态sql，完成逻辑判断和动态 拼接sql的功能，Mybatis提供了9种动态sql标签 trim|where|set|foreach|if|choose|when|otherwise|bind。

其执行原理为，使用OGNL从sql参数对象中计算表达式的值，根据表达式的值动态拼接sql，以此 来完成动态sql的功能。

**面试官：**Mybatis是否支持延迟加载？

**候选人：**

是支持的，不过默认是关闭的，如果想要使用，需要添加配置开启。不过Mybatis仅支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载**lazyLoadingEnabled**=true|false。

**面试官：**如何使用Mybatis实现批量插入？

**候选人：**

嗯，这个在项目中也用过，为了提升性能，可以采用在映射文件中通过forEach标签遍历集合，获取每一个元素作为insert语句的参数值

**面试官：**Mybatis的一级、二级缓存？

**候选人：**

嗯~~

mybatis的一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当Session进行flush或close之后，该Session中的所有Cache就将清空，默认打开一级缓存

关于二级缓存需要单独开启

二级缓存是基于namespace和mapper的作用域起作用的，不是依赖于SQL session，默认也是采用 PerpetualCache，HashMap 存储。

如果想要开启二级缓存需要在全局配置文件和映射文件中开启配置才行。

**面试官：**Mybatis的二级缓存什么时候会清理缓存中的数据？

**候选人：**

嗯！！

当某一个作用域(一级缓存 Session/二级缓存Namespaces)的进行了新增、修改、删除操作后，默认该作用域下所有 select 中的缓存将被 clear。