一、 Spring

1. 什么是 Spring 框架?

Spring 是一种轻量级开发框架，旨在提高开发人员的开发效率以及系统的可维护性。

我们一般说 Spring 框架指的都是 Spring Framework，它是很多模块的集合，使用这些模块可以很方便地协助我们进行开发。这些模块是：核心容器、数据访问/集成,、Web、AOP（面向切面编程）、工具、消息和测试模块。比如：核心容器中的 Core 组件是 Spring 所有组件的核心，Beans 组件和 Context 组件是实现 IOC 和依赖注入的基础，AOP 组件用来实现面向切面编程。

2. 列举一些重要的 Spring 模块?

Spring Core： 基础,可以说 Spring 其他所有的功能都需要依赖于该类库。主要提供 IoC 依赖注入功能。

Spring AOP ： 提供了面向切面的编程实现。

Spring JDBC : Java 数据库连接。

Spring JMS ： Java 消息服务。

Spring ORM : 用于支持 Hibernate 等 ORM 工具。

Spring Web : 为创建 Web 应用程序提供支持。

Spring Test : 提供了对 JUnit 和 TestNG 测试的支持。

3. Spring 中的设计模式

工厂设计模式 : Spring 使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象。

代理设计模式 : Spring AOP 功能的实现。

单例设计模式 : Spring 中的 Bean 默认都是单例的。

模板方法模式 : Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。

包装器设计模式 : 我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。

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。

4. IOC

1) 什么是 Ioc 及其优点

IoC（Inverse of Control:控制反转）是一种设计思想，就是 将原本在程序中手动创建对象的控制权，交由 Spring 框架来管理。IoC 容器是 Spring 用来实现 IoC 的载体， IoC 容器实际上就是个 Map（key，value）,Map 中存放的是各种对象。

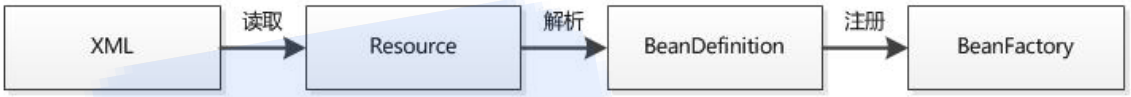
将对象之间的相互依赖关系交给 IoC 容器来管理，并由 IoC 容器完成对象的注入。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。 IoC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。

在实际项目中一个 Service 类可能有几百甚至上千个类作为它的底层，假如我们需要实例化这个 Service，你可能要每次都要搞清这个 Service 所有底层类的构造函数，这可能会把人逼疯。如果利用 IoC 的话，你只需要配置好，然后在需要的地方引用就行了，这大大增加了项目的可维护性且降低了开发难度。

Spring 时代我们一般通过 XML 文件来配置 Bean，后来开发人员觉得 XML 文件来配置不太好，于是 SpringBoot 注解配置就慢慢开始流行起来。

2) IOC 容器的初始化过程

IOC 容器的初始化主要包括 Resource 定位，载入和注册三个步骤，接下来我们依次介绍。



Resource 资源定位：

Resouce 定位是指 BeanDefinition 的资源定位，也就是 IOC 容器找数据的过程。Spring 中使用外部资源来描述一个 Bean 对象，IOC 容器第一步就是需要定位 Resource 外部资源。由 ResourceLoader 通过统一的 Resource 接口来完成定位。

BeanDefinition 的载入：

载入过程就是把定义好的 Bean 表示成 IOC 容器内部的数据结构，即 BeanDefinition。在配置文件中每一个 Bean 都对应着一个 BeanDefinition 对象。

通过 BeanDefinitionReader 读取，解析 Resource 定位的资源，将用户定义好的 Bean 表示成 IOC 容器的内部数据结构 BeanDefinition。

在 IOC 容器内部维护着一个 BeanDefinitionMap 的数据结构，通过 BeanDefinitionMap，IOC 容器可以对 Bean 进行更好的管理。

BeanDefinition 的注册：

注册是将前面的 BeanDefition 保存到 Map 中的过程，通过 BeanDefinitionRegistry 接口来实现注册。

3) Spring 提供了多种依赖注入的方式

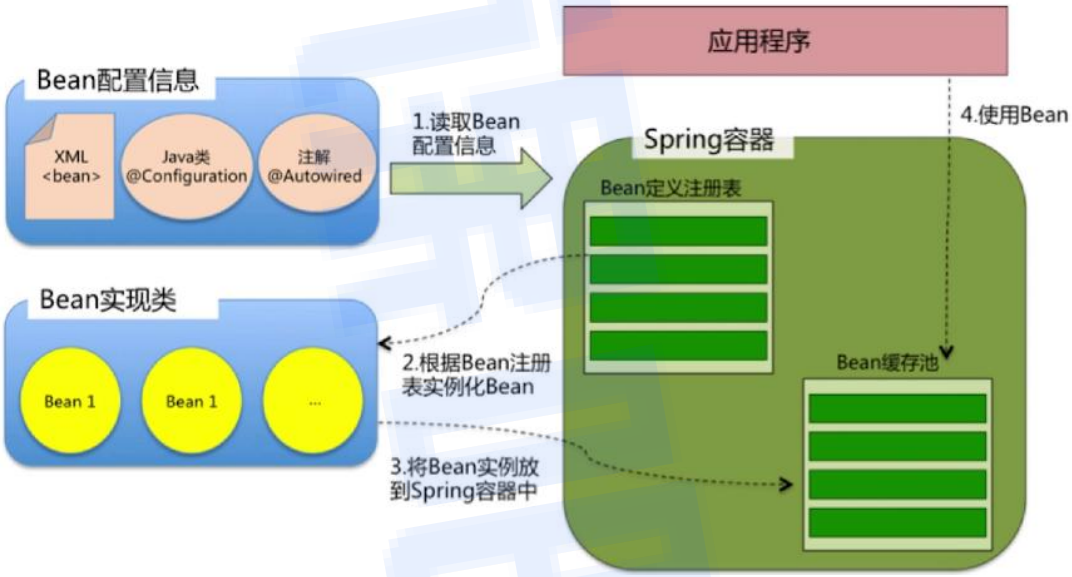
Set 注入

构造器注入

接口注入

注解注入

Spring 启动时读取应用程序提供的 Bean 配置信息，并在 Spring 容器中生成一份相应的 Bean 配置注册表，然后根据这张注册表实例化 Bean，装配好 Bean 之间的依赖关系，为上层应用提供准备就绪的运行环境。其中 Bean 缓存池为 HashMap 实现



4) Spring 中，如何给对象的属性赋值?

1. 通过构造函数
2. 通过 set 方法给属性注入值
3. 自动装配

byName 通过参数名自动装配，如果一个 bean 的 name 和另外一个 bean 的 property 相同就自动装配。

byType 通过参数的数据类型自动自动装配，如果一个 bean 的数据类型和另外一个 bean 的 property 属性的数据类型兼容，就自动装配必须确保该类型在 IOC 容器中只有一个对象；否则报错。

4.注解

使用注解步骤：

1）先引入 context 名称空间

xmlns:context="http://www.springframework.org/schema/context"

2）开启注解扫描

<context:component-scan base-package="cn.itcast.e_anno2"></context:component-scan>

3）使用注解

通过注解的方式，把对象加入 ioc 容器。

创建对象以及处理对象依赖关系，相关的注解：

@Component 指定把一个对象加入 IOC 容器

@Repository 作用同@Component； 在持久层使用

@Service 作用同@Component； 在业务逻辑层使用

@Controller 作用同@Component； 在控制层使用

@Resource 或@Autowired 属性注入

@Qualifier 指定注入 bean 的名称，防止在容器中有重名的 bean

总结：

1） 使用注解，可以简化配置，且可以把对象加入 IOC 容器,及处理依赖关系(DI)

2） 注解可以和 XML 配置一起使用。

3） @Resource 的作用相当于@Autowired，只不过@Autowired 按 byType 自动注入，而 @Resource 默认按 byName 自动注入

5) bean 对象创建的细节

1) 对象创建： 单例/多例

scope="singleton"，默认值， 即默认是单例【service/dao/工具类】

scope="prototype"，多例； 【Action 对象】

Spring 的单例 bean 不是线程安全的（可以用局部变量使线程安全）

2) 什么时候创建?

scope="prototype" 在用到对象的时候，才创建对象。

scope="singleton" 在启动(容器初始化之前)， 就已经创建了 bean，且整个应用只有一个。

3)是否延迟创建

lazy-init="false" 默认为 false， 不延迟创建，即在启动时候就创建对象

lazy-init="true" 延迟初始化， 在用到对象的时候才创建对象（只对单例有效）

4) 创建对象之后，初始化/销毁

init-method="init_user" 【对应对象的 init_user 方法，对象创建之后执行】

destroy-method="destroy_user"【在调用容器对象的 destriy 方法时候执行,(容器用实现类)】

6) Bean 的作用域?

singleton：唯一 bean 实例，Spring 中的 bean 默认都是单例的。

prototype：每次请求都会创建一个新的 bean 实例。

request：每次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP request 内有效。

session：每次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP session 内有效。

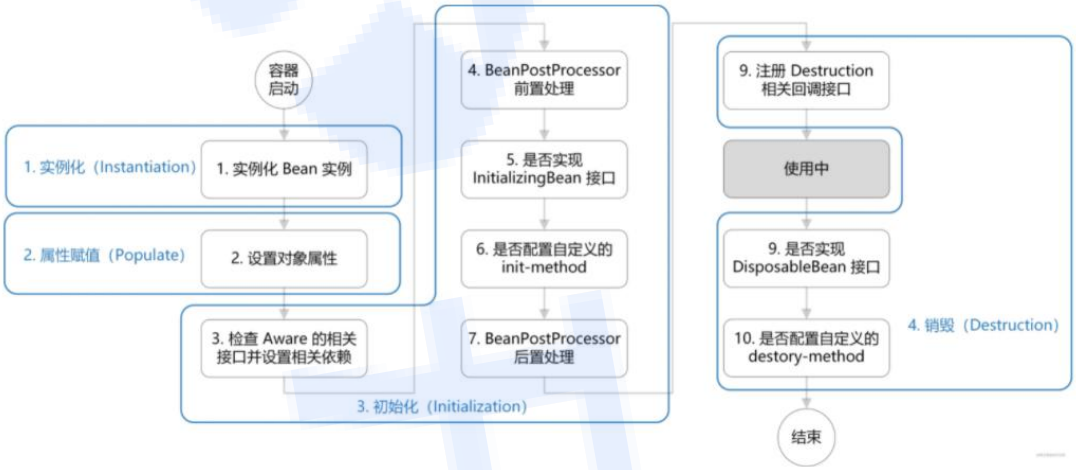
global-session： 全局 session 作用域，仅仅在基于 portlet 的 web 应用中才有意义，Spring5

已经没有了。Portlet 是能够生成语义代码(例如：HTML)片段的小型 Java Web 插件。它们基于 portlet 容器，可以像 servlet 一样处理 HTTP 请求。但是，与 servlet 不同，每个 portlet 都有不同的会话

7) Bean 的生命周期

Bean 的生命周期概括起来就是 4 个阶段：

实例化（Instantiation）
属性赋值（Populate）
初始化（Initialization）
销毁（Destruction）



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：第 3~7 步，步骤较多，其中第 5、6 步为初始化操作，第 3、4 步为在初始化前执行，第 7 步在初始化后执行，该阶段结束，才能被用户使用；
销毁：第 8~10 步，第 8 步不是真正意义上的销毁（还没使用呢），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。

总结

首先是实例化、属性赋值、初始化、销毁这 4 个大阶段；
再是初始化的具体操作，有 Aware 接口的依赖注入、BeanPostProcessor 在初始化前后的处理以及 InitializingBean 和 init-method 的初始化操作；
销毁的具体操作，有注册相关销毁回调接口，最后通过 DisposableBean 和 destory-method 进行销毁。

8) BeanFactory 和 FactoryBean 的区别：

BeanFactory：Bean 工厂，是一个工厂(Factory)， 是 Spring IOC 容器的最顶层接口，它的作用是管理 Bean，即实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。
FactoryBean：工厂 Bean，是一个 Bean，作用是产生其他 Bean 实例，需要提供一个工厂方法，该方法用来返回其他 Bean 实例。

9) BeanFactory 和 ApplicationContext 的区别？

BeanFactory 是 Spring 里面最顶层的接口，包含了各种 Bean 的定义，读取 Bean 配置文档，管理 Bean 的加载、实例化，控制 Bean 的生命周期，维护 Bean 之间的依赖关系。
ApplicationContext 接口是 BeanFactory 的派生，除了提供 BeanFactory 所具有的功能外，还提供了更完整的框架功能：
继承了 MessageSource，支持国际化。

提供了统一的资源文件访问方式。
提供在 Listener 中注册 Bean 的事件。
提供同时加载多个配置文件的功能。
载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的 web 层。

BeanFactory 和 ApplicationContext 的优缺点分析：

BeanFactory 的优缺点：
优点：应用启动的时候占用资源很少，对资源要求较高的应用，比较有优势；
缺点：运行速度会相对来说慢一些。而且有可能会出现空指针异常的错误，而且通过 Bean 工厂创建的 Bean 生命周期会简单一些。
ApplicationContext 的优缺点：
优点：所有的 Bean 在启动的时候都进行了加载，系统运行的速度快；在系统启动的时候，可以发现系统中的配置问题。
缺点：把费时的操作放到系统启动中完成，所有的对象都可以预加载，缺点就是内存占用较大。

5. AOP

答： AOP，面向切面编程是指当需要在某一个方法之前或者之后做一些额外的操作，比如说日志记录，权限判断，异常统计等，可以利用 AOP 将功能代码从业务逻辑代码中分离出来。

AOP 中有如下的操作术语：

Pointcut(切入点)：所谓切入点是指我们要对哪些 Joinpoint 进行拦截的定义（业务方法）
Joinpoint(连接点)： 连接点可以说是切点的全集。切点是连接点的子集。也可以理解为，连接点是我们没有定义那个 select 开头规则时，满足条件的全部的方法。

```
@Aspect
public class Logging {

   @Pointcut("execution(* com.yiibai.*.*(..))")
   private void selectAll(){}

}
```

Advice(通知/增强)：所谓通知是指拦截到 Joinpoint 之后所要做的事情就是通知。
Aspect(切面)：是切入点和通知（引介）的结合
Introduction(引介)：引介是一种特殊的通知在不修改类代码的前提下，Introduction 可以在运行期为类动态地添加一些方法或属性
Target(目标对象)：代（dai）理的目标对象(要增强的类)
Weaving(织入)：是把增强应用到目标的过程，把 advice 应用到 target 的过程
Proxy（代(dai)理）：一个类被 AOP 织入增强后，就产生一个结果代（dai）理类
我们来简单理解下重要概念。切入点就是在类里边可以有很多方法被增强，比如实际操作中，只是增强了个别方法，则定义实际被增强的某个方法为切入点；通知/增强 就是指增强的逻辑，比如扩展日志功能，这个日志功能称为增强；切面就是把增强应用到具体方法上面的过程称为切面。

Spring 中的 AOP 主要有两种实现方式：

使用 JDK 动态代理实现，使用 java.lang.reflection.Proxy 类来处理
使用 cglib 来实现
两种实现方式的不同之处：
JDK 动态代理，只能对实现了接口的类生成代理，而不是针对类，该目标类型实现的接口都将被代理。原理是通过在运行期间创建一个接口的实现类来完成对目标对象的代理。实现步骤大概如下：

定义一个实现接口 InvocationHandler 的类
通过构造函数，注入被代理类
实现 invoke（ Object proxy, Method method, Object[ ] args）方法
在主函数中获得被代理类的类加载器
使用 Proxy.newProxyInstance( )产生一个代理对象
通过代理对象调用各种方法
cglib 主要针对类实现代理，对是否实现接口无要求。原理是对指定的类生成一个子类，覆盖其中的方法，因为是继承，所以被代理的类或方法不可以声明为 final 类型。实现步骤大概如下：
定义一个实现了 MethodInterceptor 接口的类
实现其 intercept（）方法，在其中调用 proxy.invokeSuper( )
Spring AOP 对这两种代理方式的选择：
如果目标对象实现了接口，默认情况下会采用 JDK 的动态代理实现 AOP，也可以强制使用 cglib 实现 AOP；
如果目标对象没有实现接口，必须采用 cglib 库，Spring 会自动在 JDK 动态代理和 cglib 之间转换。

6. 事务

1) Spring 的事务控制

Spring 支持编程式事务(不用)和声明式事务两种方式：
编程式事务管理：使用 TransactionTemplate 实现。
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。
事务选择：
声明式事务管理要优于编程式事务管理，这正是 Spring 倡导的非侵入式的开发方式，使业务代码不受污染，只要加上注解就可以获得完全的事务支持。唯一不足之处是声明式事务的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。
当多个 Spring 事务存在的时候，Spring 定义了下边的 7 个传播行为来处理这些事务行为：
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。
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。
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。
PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。
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。
PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。
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。

2) 如何管理事务，事务是只读的还是读写的，对于查询的 find()是只读，对保存 save()是读写？

如果一次执行单条查询语句，则没有必要启用事务支持，数据库默认支持--执行期间的读一致性；
如果一次执行多条查询语句，例如统计查询，报表查询，在这种场景下，多条查询 SQL 必须保证整体的读一致性，否则，在前条 SQL 查询之后，后条 SQL 查询之前，数据被其他用户改变，则该

次整体的统计查询将会出现读数据不一致的状态，此时应该启用事务支持。

read-only="true"表示该事务为只读事务，比如上面说的多条查询的这种情况可以使用只读事务，由于只读事务不存在数据的修改，因此数据库将会为只读事务提供一些优化手段，例如 Oracle 对于只读事务，不启动回滚段，不记录回滚 log。

指定只读事务的办法为：

bean 配置文件中，prop 属性增加“read-Only”

或者用注解方式@Transactional(readOnly=true)

Spring 中设置只读事务是利用上面两种方式（根据实际情况）

在将事务设置成只读后，相当于将数据库设置成只读数据库，此时若要进行写的操作，会出现错误。

3) Spring 的事务是如何配置的？

先配置事务管理器 TransactionManager,不同的框架有不同属性。

再配置事务通知和属性,通过 tx:advice。

配置<aop:config>,设置那些方法或者类需要加入事务。

4) 事务并发会引起什么问题,怎么解决？

事务并发会引起脏读,幻读,不可重复读等问题,设定事务的隔离级别可以解决。

(1) 不可重复读是读取了其他事务更改的数据，针对 update 操作

解决：使用行级锁，锁定该行，事务 A 多次读取操作完成后才释放该锁，这个时候才允许其他事务更改刚才的数据。

(2) 幻读是读取了其他事务新增的数据，针对 insert 和 delete 操作

解决：使用表级锁，锁定整张表，事务 A 多次读取数据总量之后才释放该锁，这个时候才允许其他事务新增数据。

7. Spring 的单元测试

在编写完自己的功能模块后，为了保证代码的准确性，一般都会使用 junit 进行单元测试，当时使用的是 junit4 这种基于注解的方式来进行单元测试。为了和 spring 集成获取配置的 bean,通常使用 @RunWith 来加载 springjunit 这个核心类，使用 @ContextConfiguration 来加载相关的配置的文件，通过 @Resource 按名字来注入具体的 bean,最后在需要测试的方法上面加上@Test 来进行单元测试。并且在编写单元测试的时候还要遵守一定的原则如：

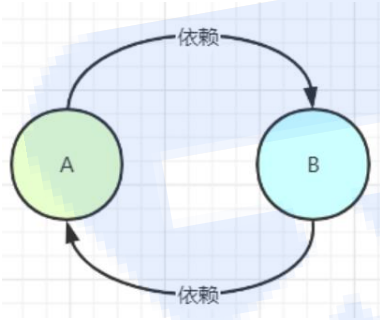
源代码和测试代码需要分开；测试类和目标源代码的类应该位于同一个包下面，即它们的包名应该一样；

测试的类名之前或之后加上@Test，测试的方法名通常也以 test 开头。

```
@RunWith(SpringJUnit4ClassRunner.class)//运行 spring 相关环境 相当于 spring 监听功能
@ContextConfiguration(locations={"classpath:spring-common.xml","classpath:spring-datasource.x
ml"})//读取 spring 配置文件 不识别* 只能识别具体文件 多个配置文件使用 string 数据传递
public class TestSpring {
  //注入 Service 层
    private @Resource UserService userService;
    @Test
    public void testFind(){
        List<User> userList = userService.findAllUserInfo();
        for (User user : userList) {
            System.err.println(user.toString());
        }
    }
}
```

8. Spring 中的循环依赖（源码级）

1) 什么是循环依赖？




```
@Component
public class A {
    // A中注入了B
    @Autowired
    private B b;
}

@Component
public class B {
    // B中也注入了A
    @Autowired
    private A a;
}
```

或

```
// 自己依赖自己
@Component
public class A {
    // A中注入了A
    @Autowired
    private A a;
}
```

2) 什么情况下循环依赖可以被处理？

原型(Prototype)的场景是不支持循环依赖的，我们能解决的就是默认单例的属性注入场景。

Spring 解决循环依赖是有前置条件的：

出现循环依赖的 Bean 必须要是单例

依赖注入的方式不能全是构造器注入的方式。用代码来说

```java
@Component
public class A {
//    @Autowired
//    private B b;
    public A(B b) {

    }
}


@Component
public class B {

//    @Autowired
//    private A a;

    public B(A a){

    }
}
```

A 中注入 B 的方式是通过构造器，B 中注入 A 的方式也是通过构造器，这个时候循环依赖是无法被解决，如果你的项目中有两个这样相互依赖的 Bean，在启动时就会报出以下错误：

```
name 'a': Requested bean is currently in creation: Is there an unresolvable circular reference?
```

循环依赖的解决情况跟注入方式的关系：

依赖情况	依赖注入方式	循环依赖是否被解决
AB相互依赖（循环依赖）	均采用setter方法注入	是
AB相互依赖（循环依赖）	均采用构造器注入	否
AB相互依赖（循环依赖）	A中注入B的方式为setter方法，B中注入A的方式为构造器	是
AB相互依赖（循环依赖）	B中注入A的方式为setter方法，A中注入B的方式为构造器	否

3) Spring 是如何解决的循环依赖？

Spring 的单例对象的初始化主要分为三步：实例化 -> 填充属性 -> 初始化 Bean

1、createBeanInstance：实例化，其实也就是调用对象的构造方法实例化对象

2、populateBean：填充属性，这一步主要是多 bean 的依赖属性进行填充

3、initializeBean：调用 spring xml 中的 init 方法。

源码简析：

```
singletonObject = singletonFactory.getObject(); // 创建Bean

doCreatedBean(){
    instanceWrapper = createBeanInstance(beanName, mdb, args); // 实例化，创建新的实例
     populateBean(beanName, mbd, instanceWrapper); //填充属性：填充准备、自动填充（byName/byType）、填充属性
     exposedObject = initializeBean(beanName, exposedObject, mbd);//初始化
}

addSingleton(beanName, singletonObject); // 加入单例池
```

从上面讲述的单例 bean 初始化步骤，可以知道，循环依赖主要发生在第一、第二步。也就是构造器循环依赖和 field 循环依赖。

那么我们要解决循环引用也应该从初始化过程着手，对于单例来说，在 Spring 容器整个生命周期内，有且只有一个对象，所以很容易想到这个对象应该存在 Cache 中，Spring 为了解决单例的循环依赖问题，使用了三级缓存。

看源码，三级缓存主要指：

```
/** Cache of singleton objects: bean name --> bean instance */
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(256)

/** Cache of singleton factories: bean name --> ObjectFactory */
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<String, ObjectFac

/** Cache of early singleton objects: bean name --> bean instance */
private final Map<String, Object> earlySingletonObjects = new HashMap<String, Object>(16);
```

这三级缓存分别指：

（1）singletonFactories ： 单例对象工厂的 cache（映射创建 Bean 的原始工厂）

（2）earlySingletonObjects ：提前暴光的单例对象的 Cache（映射 Bean 的早期引用，也就是说在这个 Map 里的 Bean 不是完整的，甚至还不能称之为“Bean”，只是一个 Instance）

（3）singletonObjects：单例对象的 cache（它是我们最熟悉的朋友，俗称“单例池”“容器”，缓存创建完成单例 Bean 的地方）

这里的缓存指的就是上面三个 Map 对象：

singletonObjects（一级 Map）：里面保存了所有已经初始化好的单例 Bean，也就是会保存 Spring IoC 容器中所有单例的 Spring Bean

earlySingletonObjects（二级 Map），里面会保存从 三级 Map 获取到的正在初始化的 Bean

singletonFactories（三级 Map），里面保存了正在初始化的 Bean 对应的 ObjectFactory 实现类，调用其 getObject() 方法返回正在初始化的 Bean 对象（仅实例化还没完全初始化好）

我们在创建 bean 的时候，首先想到的是从 cache 中获取这个单例的 bean，这个缓存就是 singletonObjects。主要调用方法就就是：

```
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        synchronized (this.singletonObjects) {
            singletonObject = this.earlySingletonObjects.get(beanName);
            if (singletonObject == null && allowEarlyReference) {
                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    singletonObject = singletonFactory.getObject();
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return (singletonObject != NULL_OBJECT ? singletonObject : null);
}
```

1. 【一级 Map】从单例缓存 `singletonObjects` 中获取 beanName 对应的 Bean
2. 如果**一级 Map**中不存在，且当前 beanName 正在创建
    1. 对 `singletonObjects` 加锁
    2. 【二级 Map】从 `earlySingletonObjects` 集合中获取，里面会保存从 **三级 Map** 获取到的正在初始化的 Bean
    3. 如果**二级 Map** 中不存在，且允许提前创建
        1. 【三级 Map】从 `singletonFactories` 中获取对应的 ObjectFactory 实现类，如果从**三级 Map** 中存在对应的对象，则进行下面的处理
        2. 调用 ObjectFactory#getOject() 方法，获取目标 Bean 对象（早期半成品）
        3. 将目标对象放入**二级 Map**
        4. 从**三级 Map**移除 beanName
3. 返回从缓存中获取的对象

上面的代码需要解释两个参数：

（1）isSingletonCurrentlyInCreation()：判断当前单例 bean 是否正在创建中，也就是没有初始化完成,比如 A 的构造器依赖了 B 对象,所以得先去创建 B 对象，或者在 A 的 populateBean 过程中依赖了 B 对象，得先去创建 B 对象，这时的 A 就是处于创建中的状态。

（2）allowEarlyReference：是否允许从 singletonFactories 中通过 getObject 拿到对象

分析 getSingleton() 的整个过程，Spring 首先从一级缓存 singletonObjects 中获取。如果获取不到，并且对象正在创建中，就再从二级缓存 earlySingletonObjects 中获取。如果还是获取不到且允许 singletonFactories 通过 getObject() 获取，就从三级缓存 singletonFactory.getObject()(三级缓存)获取，如果获取到了则：

```
this.earlySingletonObjects.put(beanName, singletonObject);
this.singletonFactories.remove(beanName);
```

从 singletonFactories 中移除，并放入 earlySingletonObjects 中。其实也就是从三级缓存移动到了二级缓存。

从上面三级缓存的分析，Spring 解决循环依赖的诀窍就在于 singletonFactories 这个三级 cache。这个 cache 的类型是 ObjectFactory，定义如下：

```
public interface ObjectFactory<T> {
    T getObject() throws BeansException;
}
```

这个接口在下面被引用

```
protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(singletonFactory, "Singleton factory must not be null");
    synchronized (this.singletonObjects) {
        if (!this.singletonObjects.containsKey(beanName)) {
            this.singletonFactories.put(beanName, singletonFactory);
            this.earlySingletonObjects.remove(beanName);
            this.registeredSingletons.add(beanName);
        }
    }
}
```

这里就是解决循环依赖的关键，这段代码发生在 createBeanInstance 之后，也就是说单例对象此时已经被创建出来(调用了构造器)。

这个对象已经被生产出来了，虽然还不完美（还没有进行初始化的第二步和第三步），但是已经能被人认出来了（根据对象引用能定位到堆中的对象），所以 Spring 此时将这个对象提前曝光出来让大家认识，让大家使用。

这样做有什么好处呢？

让我们来分析一下“A 的某个 field 或者 setter 依赖了 B 的实例对象，同时 B 的某个 field 或者 setter 依赖了 A 的实例对象”这种循环依赖的情况。

A 首先完成了初始化的第一步，并且将自己提前曝光到 singletonFactories 中，此时进行初始化的第二步，发现自己依赖对象 B，此时就尝试去 get(B)，发现 B 还没有被 create，所以走 create 流程，B 在初始化第一步的时候发现自己依赖了对象 A，于是尝试 get(A)，尝试一级缓存 singletonObjects(肯定没有，因为 A 还没初始化完全)，尝试二级缓存 earlySingletonObjects（也没有），尝试三级缓存 singletonFactories，由于 A 通过 ObjectFactory 将自己提前曝光了，所以 B 能够通过 ObjectFactory.getObject 拿到 A 对象(虽然 A 还没有初始化完全，但是总比没有好呀)，B 拿到 A 对象后顺利完成了初始化阶段 1、2、3，完全初始化之后将自己放入到一级缓存 singletonObjects 中。此时返回 A 中，A 此时能拿到 B 的对象顺利完成自己的初始化阶段 2、3，最终 A 也完成了初始化，进去了一级缓存 singletonObjects 中，而且更加幸运的是，由于 B 拿到了 A 的对象引用，所以 B 现在 hold 住的 A 对象完成了初始化。

Spring 通过三级缓存加上“提前曝光”机制，配合 Java 的对象引用原理，比较完美地解决了 field 属性的循环依赖问题！

知道了这个原理时候，肯定就知道为什么 Spring 不能解决“A 的构造方法中依赖了 B 的实例对象，同时 B 的构造方法中依赖了 A 的实例对象”这类问题了！因为加入 singletonFactories 三级缓存的前提是执行了构造器，所以构造器的循环依赖没法解决。

4) **为什么需要上面的二级 Map ？**

因为通过 三级 Map 获取 Bean 会有相关 SmartInstantiationAwareBeanPostProcessor#getEarlyBeanReference(..) 的处理，避免重复处理，处理后返回的可能是一个代理对象。

例如在循环依赖中一个 Bean 可能被多个 Bean 依赖， A -> B（也依赖 A） -> C -> A，当你获取 A 这个 Bean 时，后续 B 和 C 都要注入 A，没有上面的 二级 Map 的话，三级 Map 保存的 ObjectFactory 实现类会被调用两次，会重复处理，可能出现问题，这样做在性能上也有所提升。

5) **为什么不直接调用这个 ObjectFactory#getObject() 方法放入 二级 Map 中，而需要上面的三级 Map？**

对于不涉及到 AOP 的 Bean 确实可以不需要 singletonFactories（三级 Map），但是 Spring

AOP 就是 Spring 体系中的一员，如果没有 singletonFactories（三级 Map），意味着 Bean 在实例化后就要完成 AOP 代理，这样违背了 Spring 的设计原则。Spring 是通过 AnnotationAwareAspectJAutoProxyCreator 这个后置处理器在完全创建好 Bean 后来完成 AOP 代理，而不是在实例化后就立马进行 AOP 代理。如果出现了循环依赖，那没有办法，只有给 Bean 先创建代理对象，但是在没有出现循环依赖的情况下，设计之初就是让 Bean 在完全创建好后才完成 AOP 代理。

Spring 的设计是让 Bean 在完全创建好后才完成 AOP 代理，因为创建的代理对象需要关联目标对象，在拦截处理的过程中需要根据目标对象执行被拦截的方法，所以这个目标对象最好是一个“成熟态”，而不是仅实例化还未初始化的一个对象。

二、 Mybatis

1. MyBatis

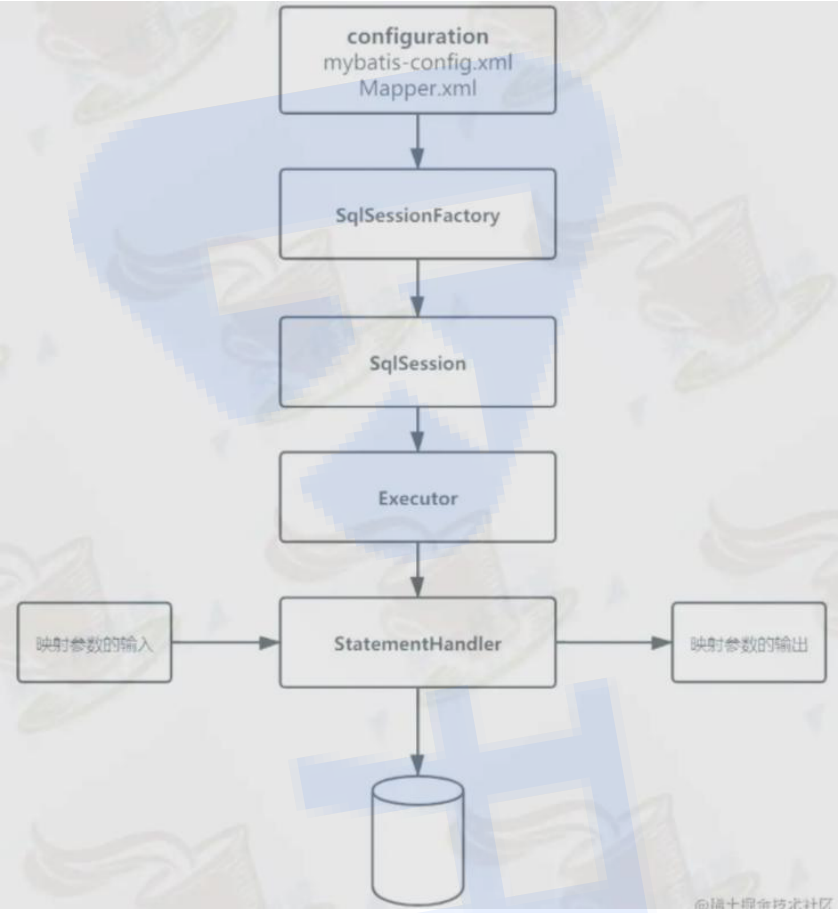
MyBatis 是一个半 ORM（对象关系映射）框架，内部封装了 JDBC，开发时只需要关注 SQL 语句本身，不需要花费精力去处理加载驱动、创建连接、创建 statement 等繁杂的过程。通过直接编写原生态 SQL，可以严格控制 SQL 语句的执行性能，灵活度高(支持动态 SQL 语句)。

MyBatis 使用 XML 或注解来配置和映射原生信息，将 POJO 映射成数据库中的记录，避免了 JDBC 代码手动设置参数以及获取结果集的繁琐步骤。

MyBatis 通过 xml 文件或注解的方式将要执行的各种 statement 配置起来，并通过 Java 对象和 statement 中 SQL 的动态参数进行映射生成最终执行的 SQL 语句，最后由 MyBatis 框架执行 SQL 语句，并将结果映射为 Java 对象并返回。

2. Mybatis 工作流程

每一个 Mybatis 的应用程序都以一个 SqlSessionFactory 对象的实例为核心。首先用字节流通过 Resource 将配置文件读入，然后通过 SqlSessionFactoryBuilder().build 方法创建 SqlSessionFactory，然后再通过 SqlSessionFactory.openSession()方法创建一个 SqlSession 为每一个数据库事务服务。经历了 Mybatis 初始化 ->创建 SqlSession ->运行 SQL 语句，返回结果三个过程



3. Mapper.xml 中 statement 中属性含义

1) id:sql 语句唯一标识
2) parameterType：指定输入参数类型，mybatis 通过 ognl 从输入对象中获取参数值拼 接在 sql 中。
3) resultType：指定输出结果类型，mybatis 将 sql 查询结果的一行记录数据映射为 resultType 指定类型的对象。
4) resultMap：resultType 可以指定 pojo 将查询结果映射为 pojo，但需要 pojo 的属性名和 sql 查询的列名一致方可映射成功。如果 sql 查询字段名和 pojo 的属性名不一致，可以通过 resultMap 将字段名和属性名作一个对应关系 resultMap 实质上还需要将查询结果映射到 pojo 对象中。 resultMap 可以实现将查询结果映射为复杂类型的 pojo，比如在查询结果映射对象中包括 pojo 和 list 实现一对一查询和一对多查询。
5) #{}表示一个占位符号，通过#{}可以实现 preparedStatement 向占位符中设置值，自动进行 java 类型和 jdbc 类型转换，#{}可以有效防止 sql 注入。 #{}可以接收简单类型值或 pojo 属性值。 如果 parameterType 传输单个简单类型值，#{}括号中可以是 value 或其它名称。一般能用#的就别用$.
6) ${}表示拼接 sql 串，通过${}可以将 parameterType 传入的内容拼接在 sql 中且不进 行 jdbc 类型转换， ${}可以接收简单类型值或 pojo 属性值，如果 parameterType 传输单个简单类型值，${}括号中只能是 value。

4. 简述 MyBatis 的单个参数、多个参数如何传递及如何取值。

MyBatis 传递单个参数，如果是普通类型(String+8 个基本)的，取值时在#{}中可以任意指定，如果是对象类型的，则在#{}中使用对象的属性名来取值

MyBatis 传递多个参数，默认情况下，MyBatis 会对多个参数进行封装 Map. 取值时在#{}可以使用 0 1 2 .. 或者是 param1 param2..

MyBatis 传递多个参数，建议使用命名参数，在 Mapper 接口的方法的形参前面使用@Param() 来指定封装 Map 时用的 key. 取值时在#{}中使用@Param 指定的 key.

5. #和$ 有什么区别？

MyBatis 中我们能用#号就尽量不要使用$符号，它们的区别主要体现在下边几点：

#符号将传入的数据都当做一个字符串，会对自动传入的数据加一个双引号

$符号将传入的数据直接显示在生成的 SQL 语句中。

#符号存在预编译的过程，对问号赋值，防止 SQL 注入。

$符号是直译的方式，一般用在 order by ${列名}语句中。

6. Mybatis 中自主主键如何获取

数据库为 MySql 时：

```
<insert id="insert" parameterType="com.test.User" keyProperty="userId"
useGeneratedKeys="true" >
```

“keyProperty”表示返回的 id 要保存到对象的那个属性中，”useGeneratedKeys”表示主键 id 为自增长模式。MySQL 中做以上配置就可以了。

数据库为 Oracle 时：

由于 Oracle 没有自增长一说法，只有序列这种模仿自增的形式，所以不能再使用”useGeneratedKeys”属性。而是使用<selectKey>将 ID 获取并赋值到对象的属性中，insert 插入操作时正常插入 id。

```
<insert id="insert" parameterType="com.test.User">
  <selectKey resultType="INTEGER" order="BEFORE" keyProperty="userId">
      SELECT SEQ_USER.NEXTVAL as userId from DUAL
  </selectKey>
insert into user (user_id, user_name, modified, state)
values (#{userId,jdbcType=INTEGER}, #{userName,jdbcType=VARCHAR},
        #{modified,jdbcType=TIMESTAMP}, #{state,jdbcType=INTEGER})
</insert>
```

7. Mybatis 中 uuid 主键如何获取

需要增加通过 select uuid()得到 uuid 值

要将 User 中的 id 改成 String 类型，并且将 User 表中的 id 字段改为 varchar(36)

8. Mapper 接口开发方法

通常 Mybatis 开发 Dao 方法有两种。即原始 Dao 开发方法和 Mapper 接口开发方法。

Mapper 接口开发方法只需要程序员编写 Mapper 接口（相当于 Dao 接口），由 Mybatis 框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边 Dao 接口实现类方法。

Mapper 接口开发需要遵循以下规范：

1) Mapper.xml 文件中的 namespace 与 mapper 接口的类路径相同。
2) Mapper 接口方法名和 Mapper.xml 中定义的每个 statement 的 id 相同
3) Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 类型相同
4) Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 类型相同

接口绑定有两种实现方式,一种是通过注解绑定,就是在接口的方法上面加上@Select@Update 等注解里面包含 Sql 语句来绑定,另外一种就是通过 xml 里面写 SQL 来绑定,在这种情况下,要指定 xml 映射文件里面的 namespace 必须为接口的全路径名.

Dao 接口的工作原理是 JDK 动态代理，MyBatis 运行时会使用 JDK 动态代理为 Dao 接口生成代理 proxy 对象，代理对象 proxy 会拦截接口方法，转而执行 MappedStatement 所代表的

sql，然后将 sql 执行结果返回。

9. 不同的映射文件 xml 中的 id 值可以重复吗？

MyBatis 使用全限名+方法名来寻找对应的 MapperStatement，那么看起来我们是不可以定义相同的 id 值的。但是注意需要分情况讨论该问题。

不同的 xml 映射文件，如果配置了 namespace，那么 id 可以重复

如果没有配置 namespace，那么 id 不能重复

10. Mybatis 能执行一对一、一对多的关联查询吗？都有哪些实现方式，以及它们之间的区别？

Mybatis 不仅可以执行一对一、一对多的关联查询，还可以执行多对一，多对多的关联查询，多对一查询，其实就是一对一查询，只需要把 selectOne()修改为 selectList()即可；多对多查询，其实就是一对多查询，只需要把 selectOne()修改为 selectList()即可。

关联对象查询，有两种实现方式，一种是单独发送一个 sql 去查询关联对象，赋给主对象，然后返回主对象。另一种是使用嵌套查询，嵌套查询的含义为使用 join 查询，一部分列是 A 对象的属性值，另外一部分列是关联对象 B 的属性值，好处是只发一个 sql 查询，就可以把主对象和其关联对象查出来。

11. 动态 sql

动态 SQL 是指可以根据不同的参数信息来动态拼接的不确定的 SQL 叫做动态 SQL。

MyBatis 动态 SQL 的主要元素有：if、choose/when/otherwise、trim、where、set、foreach 等。

通过 mybatis 提供的各种标签方法实现动态拼接 sql

foreach 标签:循环传入的集合参数

collection:传入的集合的变量名称

item:每次循环将循环出的数据放入这个变量中

open:循环开始拼接的字符串

close:循环结束拼接的字符串

separator:循环中拼接的分隔符

where 标签作用:会自动向 sql 语句中添加 where 关键字,会去掉第一个条件的 and 关键字。

include 标签：调用 sql 条件

12. Mybatis 如何完成 MySQL 的批量操作,举例说明

MyBatis 完成 MySQL 的批量操作主要是通过<foreache>标签来拼装相应的 SQL 语句.

例如:

```
<insert id="insertBatch" >
    insert into tbl_employee(last_name,email,gender,d_id) values
    <foreach collection="emps" item="curr_emp" separator=",">
        (#{curr_emp.lastName},#{curr_emp.email},#{curr_emp.gender},#{curr_emp.dept.id})
    </foreach>
</insert>
```

13. Mybatis 缓存机制

mybatis 是自动开启一级缓存的,sqlsession 级别

mybatis 的二级缓存需要手动开启的,mapper 级别(application 应用),

通常项目中,对于字典表(大量查询,很少修改)的数据,可以使用 mybatis 的二级缓存机制,提高查询效率。比如省市县、汽车品牌、配件类别、企业信息等

如何使用 Mybatis 的二级缓存

1：mybatis 的核心配置文件当中,需要手动开启二级缓存

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

2:在指定 mapper 文件当中,是否启用二级缓存,以及二级缓存的实现类(实现 cache 接口)

3:如果 Mapper 启动二级缓存,那么该 mapper 对应的实体类,必须实现序列化接口

mybatis 的二级缓存通常是使用 ehcache,但是可以使用 redis 来充当二级缓存容器

14. 如何开启 MyBatis 的延迟加载?

只需要在 mybatis-config.xml 设置即可打开延迟缓存功能，完整配置文件如下

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

15. Mybatis 逆向工程

使用官方网站自动生成工具 mybatis-generator-core-1.3.2 来生成 pojo 类和 mapper 映射文件及 mapper 接口。

作用:mybatis 官方提供逆向工程,可以使用它通过数据库中的表来自动生成 Mapper 接口和映射文件(单表增删改查)和 pojo 类.

16. mybatis 使用了哪些设计模式

sqlSessionFactoryBuilder 建造者模式

sqlSessionFactory 单例 singleTon

sqlSession 工厂模式 beanFactory

mapper 动态代理 springaop

springmvc handlerAdapter 适配器模式

17. Mybatis 是如何进行分页的？分页插件的原理是什么？

1）Mybatis 使用 RowBounds 对象进行分页，可以直接编写 sql 实现分页，也可以使用 Mybatis 的分页插件 PageHelper。

2）分页插件的原理：实现 Mybatis 提供的接口 Interceptor，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql。

select * from student，拦截 sql 后重写为：select t.* from （select * from student）t limit 0，10

三、 SpringMVC

1. 适配器模式

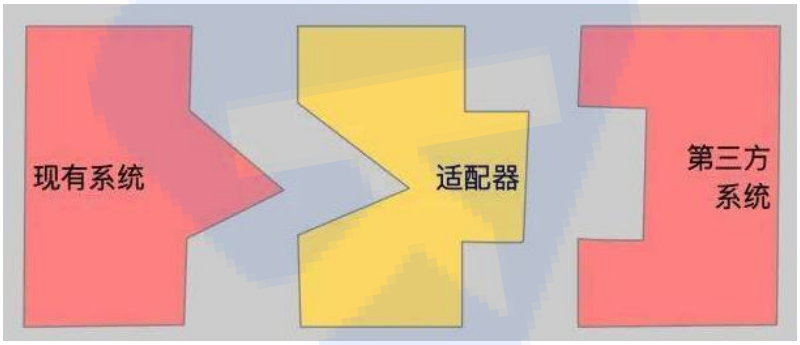
假设我们在做一套股票看盘系统，数据提供方给我们提供 XML 格式数据，我们获取数据用来显示，随着系统的迭代，我们要整合一些第三方系统的对外数据，但是他们只提供获取 JSON 格式的数据接口。

在不想改变原有代码逻辑的情况下，如何解决呢？

这时候我们就可以创建一个「适配器」。这是一个特殊的对象， 能够转换对象接口， 使其能

与其他对象进行交互。

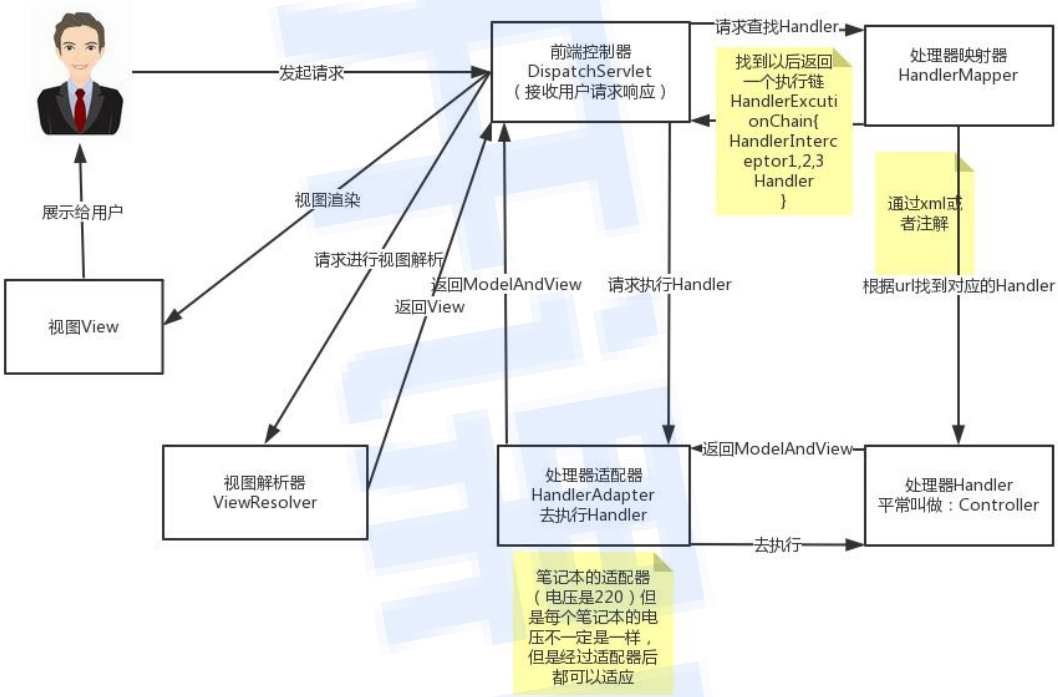
适配器模式通过封装对象将复杂的转换过程隐藏于幕后。被封装的对象甚至察觉不到适配器的存在。



2. SpringMVC 运行原理

SpringMVC 是一种轻量级的 Web 层框架，是一个基于请求驱动的 Web 框架，使用了“前端控制器（DispatcherServlet）”模型来进行设计，再根据请求映射规则分发给相应的页面控制器进行处理。消息处理依次经过的组件如下：

DispatcherServlet、HandlerMapping、HandlerAdapter，返回一个 ModelAndView 逻辑视图名、ViewResolver、View



客户端（浏览器）发送请求，直接请求到 DispatcherServlet。

DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。

解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理。

HandlerAdapter 会根据 Handler 来调用真正的处理器来处理请求，并处理相应的业务逻辑。

处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是个逻辑上的 View。

ViewResolver 会根据逻辑 View 查找实际的 View。

DispaterServlet 把返回的 Model 传给 View（视图渲染）。

把 View 返回给请求者（浏览器）

3. **Spring MVC 常用的注解**

springMVC 中用到过的注解有

@RequestParam 它的作用是接受前台传递的参数并且可以通过 defaultValue 属性对其设置默认值；在 SpringMVC 进行文件上传的时候也会通过@RequestParam 和 MultipartFile 结合使用。

@Autowired 注解和@Resource 注解的作用都是为了进行属性注入，但@Autowired 默认是按照类型进行匹配，它是 Spring 提供的注解，@Resource 默认按照名字进行匹配，它是 java 提供的注解。

在进行 restful 接口编程时还会用到@pathvariable 注解从路径中获取参数信息以及用到

@ResponseBody 注解将实体类自动转换为指定的 json 格式,

@RequestBody 将前台传递过来的 json 格式的数据转换为对应的 javabean。

除此之外还有@Controller，@Service,@Repository 分别在控制层，业务逻辑层和持久层的实现类型添加。

@RequestMapping 注解在控制层的方法上添加从而将指定 url 和方法对应起来。

@RestController：相当于 @Controller 加 @ResponseBody 的组合效果

@Configuration：用于定义配置类

@PathVariable 用于获取路径参数

@RequestParam 用于获取查询参数。

@Value 读取比较简单的配置信息：

4. **如何使用 SpringMVC 完成 JSON 操作**

配置 MappingJacksonHttpMessageConverter

如果使用 mvc:annotation-driven 就不需要配置。

使用 @RequestBody 注解作为请求参数或 ResponseBody 作为返回值

还有常用的 fastjson,jackson 等第三方组件。

5. **mvc:annotation-driven**

使用 mvc:annotation-driven 代替注解映射器和注解适配器配置器

mvc:annotation-driven 默认加载很多的参数绑定方法，比如 json 转换解析器就默认加载了，如果使用 mvc:annotation-driven 就不用配置

RequestMappingHandlerMapping 和 RequestMappingHandlerAdapter

6. **SpringMVC 怎么样设定重定向和转发的？**

转发：在返回值前面加"forward:"，譬如"forward:user.do?name=method4"

重定向：在返回值前面加"redirect:"，譬如"redirect:http://www.baidu.com"

7. **SpringMvc 怎么和 AJAX 相互调用的？**

通过 Jackson 框架就可以把 Java 里面的对象直接转化成 Js 可以识别的 Json 对象。步骤如下 ：

（1）加入 Jackson.jar

（2）在配置文件中配置 json 的映射

（3）在接受 Ajax 方法里面可以直接返回 Object,List 等,但方法前面要加上@ResponseBody 注解。

8. **restful**

一种软件架构风格，设计风格而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。（强调以资源为导向）

所谓"资源"，就是网络上的一个实体，或者说是网络上的一个具体信息。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实在。你可以用一个 URI（统一资源定位符）指向它，每种资源对应一个特定的 URI。要获取这个资源，访问它的 URI 就可以，

因此 URI 就成了每一个资源的地址或独一无二的识别符。

资源定位：互联网所有的事物都是资源，要求 url 中没有动词，只有名词。没有参数

Url 格式：http://blog.csdn.net/beat_the_world/article/details/45621673

（使用 restful 后，utl 中不能用？传参或.action）

资源操作：使用 put、delete、post、get，使用不同方法对资源进行操作。分别对应添加、删除、修改、查询。一般使用时还是 post 和 get。Put 和 Delete 几乎不使用。

客户端用到的手段，只能是 HTTP 协议。具体来说，就是 HTTP 协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。

它们分别对应四种基本操作：GET 用来获取资源，POST 用来新建资源（也可以用于更新资源），PUT 用来更新资源，DELETE 用来删除资源。

URI 的设计只要负责把资源通过合理方式暴露出来就可以了。对资源的操作与它无关，操作是通过 HTTP 动词体现，所以 REST 通过 URI 暴露资源时，会强调不要在 URI 中出现动词。

比如：左边是错误的设计，而右边是正确的

GET /rest/api/getDogs --> GET /rest/api/dogs 获取所有小狗狗

GET /rest/api/addDogs --> PUT /rest/api/dog 添加一个小狗狗

GET /rest/api/editDogs/:dog_id --> POST /rest/api/dogs/:dog_id 修改一个小 狗狗

GET /rest/api/deleteDogs/:dog_id --> DELETE /rest/api/dogs/:dog_id 删除一 个小狗狗

左边的这种设计，很明显不符合 REST 风格，上面已经说了，URI 只负责准确无误的暴露资源，而 getDogs/addDogs…已经包含了对资源的操作，这是不对的。相反右边却满足了，它的操作是使用标准的 HTTP 动词来体现。

9. SpringMvc 里面拦截器是怎么实现的？

定义一个类,实现了 handlerInterceptor 接口,重写 preHandle postHandle afterCompletion 三个方法,之后在 springmvc 的配置文件当中使用<mvc:interceptors>里面可以配置多个 interceptor

preHandle

此方法的执行时机是在控制器方法执行之前，所以我们通常是使用此方法对请求部分进行增强。同时由于结果视图还没有创建生成，所以此时我们可以指定响应的视图。

postHandle

此方法的执行时机是在控制器方法执行之后，结果视图创建生成之前。所以通常是使用此方法对响应部分进行增强。因为结果视图没有生成，所以我们此时仍然可以控制响应结果。

afterCompletion

此方法的执行时机是在结果视图创建生成之后，展示到浏览器之前。所以此方法执行时，本次请求要准备的数据具已生成完毕，且结果视图也已创建完成，所以我们可以利用此方法进行清理操作。同时，我们也无法控制响应结果集内容。

使用场景：

用户登录判断，在执行 Action 的前面判断是否已经登录，如果没有登录的跳转到登录页面。

用户权限判断，在执行 Action 的前面判断是否具有，如果没有权限就给出提示信息。

操作日志

在 Spring MVC 的配置文件中，添加 ，用于排除拦截目录，完整配置的示例代码如下：

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**" />
        <!-- 排除拦截地址 -->
        <mvc:exclude-mapping path="/api/**" />
        <bean class="com.learning.core.MyInteceptor">
</bean>
    </mvc:interceptor>
</mvc:interceptors>
```

四、 SpringBoot

1. 什么是 Spring Boot?

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器 starter，开发者能快速上手。

SpringBoot 的优点包括可以独立运行，简化了配置，可以实现自动配置，无代码生成以及 XML 配置，并且可以进行应用监控。

以前如果必须启动一个新的 Spring 项目，我们必须添加构建路径或 maven 依赖项，配置 application server，添加 Spring 配置。因此，启动一个新的 Spring 项目需要大量的工作，因为我们目前必须从头开始做所有事情。Spring Boot 是这个问题的解决方案。Spring boot 构建在现有 Spring 框架之上。使用 Spring boot，可以避免以前必须执行的所有样板代码和配置。

2. Spring Boot 的优点是什么?

编码：减少开发、测试的时间和工作量。

配置：使用 JavaConfig 有助于避免使用 XML。没有 web.xml 文件，只需添加带 @ configuration 注释的类。

避免大量 maven 导入和各种版本冲突。

部署：不需要单独的 Web 服务器。这意味着您不再需要启动 Tomcat 或其他任何东西。

3. SpringBoot、SpringMVC 和 Spring 区别

spring boot 只是一个配置工具,整合工具,辅助工具.

springmvc 是框架,项目中实际运行的代码

Spring 框架就像一个家族，有众多衍生产品例如 boot、security、jpa 等等。但他们的基础都是 Spring 的 ioc 和 aop，ioc 提供了依赖注入的容器， aop 解决了面向横切面的编程，然后在此两者的基础上实现了其他延伸产品的高级功能。

4. SpringBoot 的核心注解

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

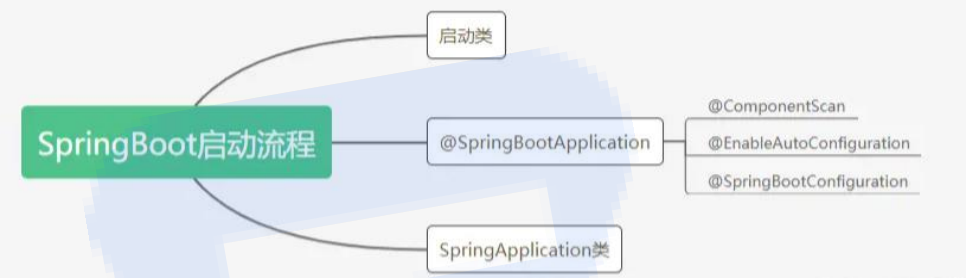
@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项。如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

@ComponentScan：Spring 组件扫描。

5. Spring Boot 自动配置原理是什么

springboot 中只需要有@SpringBootApplication 这个注解，有了它马上就能够让整个应用跑起来。实际上它只是一个组合注解，@Configuration 配置类，@ComponentScan 类，包扫描，@EnableAutoConfiguration 根据需求自动加载相关的 bean 这三个注解。



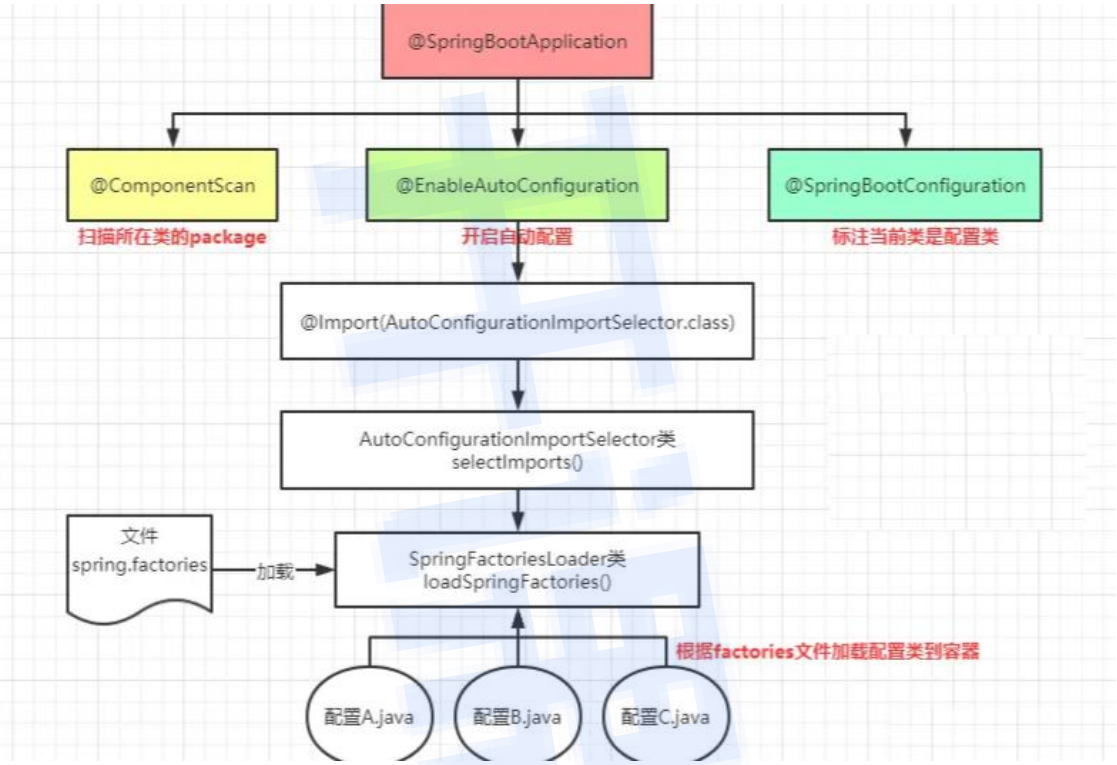
@EnableAutoConfiguration 这个注解开启自动配置，它的作用：

利用 EnableAutoConfigurationImportSelector 给容器中导入一些组件

这个类父类有一个方法：selectImports()，这个方法返回 configurations ：

List configurations = getCandidateConfigurations(annotationMetadata, attributes);获取候选的配置

将类路径下 META-INF/spring.factories 里面配置的所有 EnableAutoConfiguration 的值加入到了容器中，加载某个组件的时候根据注解的条件判断每个加入的组件是否生效，如果生效就把类的属性和配置文件绑定起来，这时就读取配置文件的值加载组件。



6. **Spring Boot 提供了两种常用的配置文件**

   properties 文件和 yml 文件

7. **常见的 starter 有哪些?**

   Spring Boot Starters 是一系列依赖关系的集合，因为它的存在，项目的依赖之间的关系对我们来说变的更加简单了

   spring-boot-starter-web：Web 开发支持

   spring-boot-starter-data-jpa：JPA 操作数据库支持

   spring-boot-starter-data-redis：Redis 操作支持

   spring-boot-starter-data-solr：Solr 权限支持

   mybatis-spring-boot-starter：MyBatis 框架支持

8. **Spring Boot 有哪几种读取配置的方式?**

Spring Boot 可以通过 @Value、@Environment、@ConfigurationProperties 这三种方式来读取

9. SpringBoot 热部署使用什么？

devtools

热部署的实现原理主要依赖 java 的类加载机制，在实现方式可以概括为在容器启动的时候起一条后台线程，定时的检测类文件的时间戳变化，如果类的时间戳变掉了，则重新加载整个应用的 class 文件，同时重启服务，重新部署。

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-devtools</artifactId>
<optional>true</optional>
</dependency>
```

热加载是在运行时重新加载 class 文件，不会重启服务。

10. 如何使用 Spring Boot 实现异常处理

使用 @ControllerAdvice 和@ExceptionHandler 处理全局异常

11. SpringBoot 如何实现打包

进入项目目录在控制台输入 mvn clean package，clean 是清空已存在的项目包，package 进行打包或是直接在 idea 中用 maven 的命令 install