框架

- IoC
  - 什么是IoC
    - Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。
    - 传统情况下，直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC是有专门一个容器来创建这些对象，即由Ioc容器来控制对象的创建，控制了外部资源获取。也就是说控制权由我们原来的对象反转为IoC容器了。
  - IoC的作用
    - 有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。
  - 解释
    - spring是一个ioc容器。ioc容器实际上就是个map（key，value），里面存的是各种对象（在xml里配置的bean节点||repository、service、controller、component），在项目启动的时候会读取配置文件里面的bean节点，根据全限定类名使用反射new对象放到map里；扫描到打上上述注解的类还是通过反射new对象放到map里。
    - 这个时候map里就有各种对象了，接下来我们在代码里需要用到里面的对象时，再通过DI注入（autowired、resource等注解，xml里bean节点内的ref属性，项目启动的时候会读取xml节点ref属性根据id注入，也会扫描这些注解，根据类型或id注入；id就是对象名）。
  - 控制反转和依赖注入
    - 依赖注入和控制反转是对同一件事情的不同描述，从某个方面讲，就是它们描述的角度不同。
    - 依赖注入是从应用程序的角度在描述，可以把依赖注入描述完整点：应用程序依赖容器创建并注入它所需要的外部资源；
    - 而控制反转是从容器的角度在描述，描述完整点：容器控制应用程序，由容器反向的向应用程序注入应用程序所需要的外部资源。
  - 核心
    - 底层的核心技术，反射，他会通过反射的技术，直接根据你的类去自己构建对应的对象出来，用的就是反射技术
- Spring容器
  - 初始化过程
    - 1.定位并加载配置文件

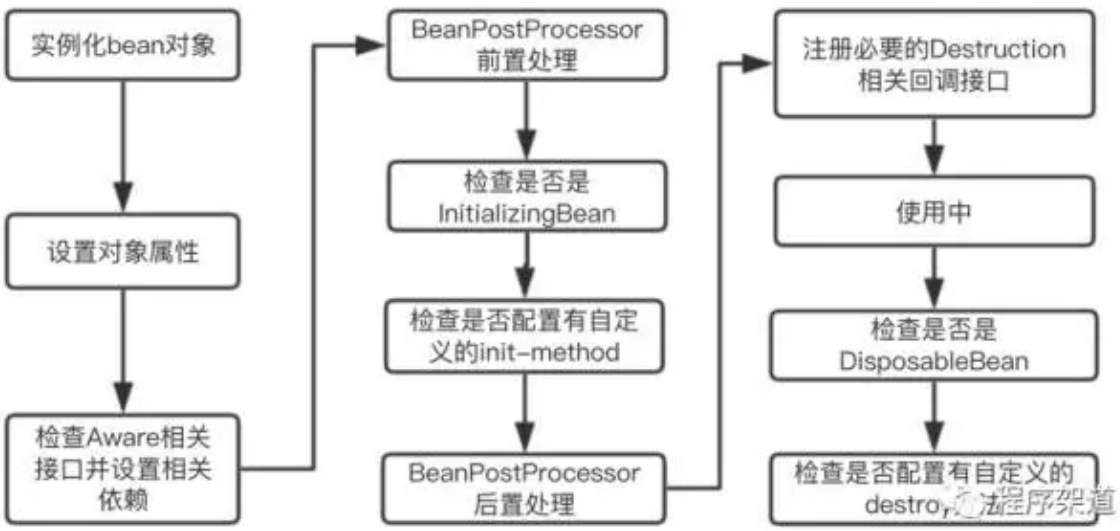
    - 2.解析配置文件中的bean节点，一个bean节点对应一个BeanDefinition对象（这个对象会保存我们在Bean节点内配置的所有内容，比如id，全限定类名，依赖值等等）
    - 3.根据上一步的BeanDefinition集合生成（BeanDefinition对象内包含生成这个对象所需要的所有参数）所有非懒加载的单例对象，其余的会在使用的时候再实例化对应的对象。
    - 4.依赖注入
    - 5.后置处理
-
- IoC容器想要管理各个业务对象以及它们之间的依赖关系，需要通过某种途径来记录和管理这些信息：BeanDefinition
- BeanDefinition
    - 容器中的每一个bean都会有一个对应的BeanDefinition实例，该实例负责保存bean对象的所有必要信息，包括bean对象的class类型、是否是抽象类、构造方法和参数、其它属性等等。当客户端向容器请求相应对象时，容器就会通过这些信息为客户端返回一个完整可用的bean实例。
- BeanDefinitionRegistry和 BeanFactory
    - BeanDefinitionRegistry抽象出bean的注册逻辑，而BeanFactory则抽象出了bean的管理逻辑，而各个BeanFactory的实现类就具体承担了bean的注册以及管理工作。
    - DefaultListableBeanFactory作为一个比较通用的BeanFactory实现，它同时也实现了BeanDefinitionRegistry接口，因此它就承担了Bean的注册管理工作。
- Spring IoC容器的整个工作流程大致可以分为两个阶段：
    - 1、容器启动阶段
        - 容器启动时，会通过某种途径加载 ConfigurationMetaData。除了代码方式比较直接外，在大部分情况下，容器需要依赖某些工具类，比如：BeanDefinitionReader，BeanDefinitionReader会对加载的 ConfigurationMetaData进行解析和分析，并将分析后的信息**组装为相应的BeanDefinition**，最后把这些保存了bean定义的BeanDefinition，**注册到相应的BeanDefinitionRegistry**，这样容器的启动工作就完成了。这个阶段主要完成一些准备性工作，更侧重于bean对象管理信息的收集，当然一些验证性或者辅助性的工作也在这一阶段完成。
    - 2、Bean的实例化阶段
        - 经过第一阶段，所有bean定义都通过BeanDefinition的方式注册到BeanDefinitionRegistry中，当某个请求通过容器的getBean方法请求某个对象，或者因为依赖关系容器需要隐式的调用getBean时，就会触发第二阶段的活动：容器会首先检查所请求的对象之前是否已经实例化完成。如果没有，则会根据注册的BeanDefinition所提供的信息实例化被请求对象，并为其注入依赖。当该对象装配完毕后，容器会立即将其返回给请求方法使用。
        - BeanFactory只是Spring IoC容器的一种实现，如果没有特殊指定，它采用采用延迟初始化策略：只有当访问容器中的某个对象时，才对该对象进行初始化和依赖注入操作。而在实际场景下，我们更多的使用另外一种类型的容器：

> > ApplicationContext，它构建在BeanFactory之上，属于更高级的容器，除了具有BeanFactory的所有能力之外，还提供对事件监听机制以及国际化的支持等。它管理的bean，在容器启动时全部完成初始化和依赖注入操作。

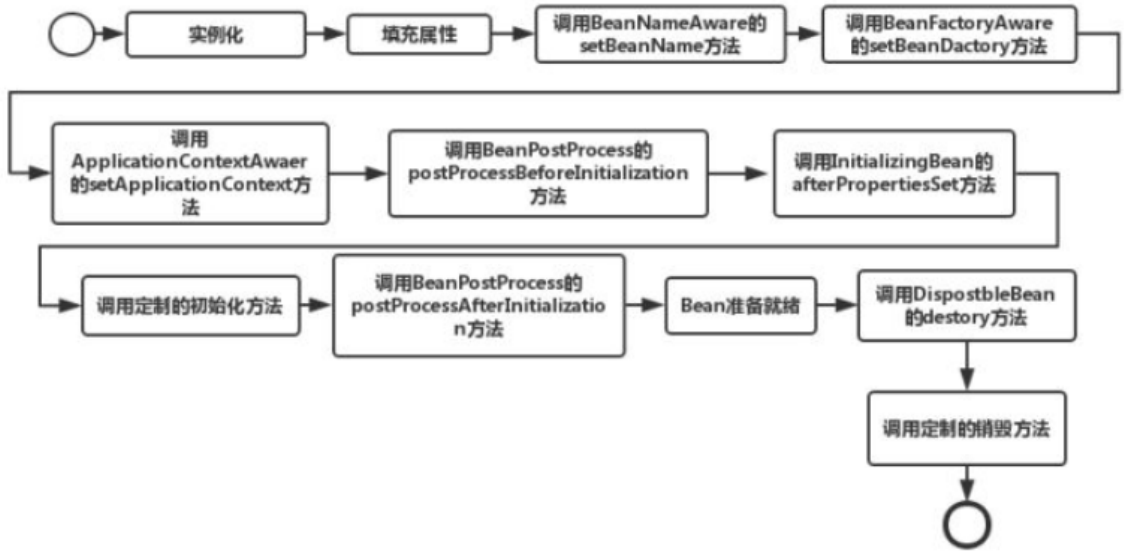
- AOP
  - AOP就是面向切面编程，目的是让你的业务逻辑去关注自己本身的业务。**这种在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。**（还可以在编译期、类加载期织入，比如AspectJ）
  - AOP解决的问题
    - 除了所谓的业务代码，还存在数量相当的公共代码，类似日志、安全验证、事物、异常处理等问题。这部分代码重要但是与我们编写程序要实现的功能没有关系，具有功能相似、重用性高、使用场景分散等特点。我们姑且称它们为共性问题。
    - AOP的就是为了解决这类共性问题，将散落在程序中的公共部分提取出来，以切面的形式切入业务逻辑中，使程序员只专注于业务的开发，从事务提交等与业务无关的问题中解脱出来。
  - jdk动态代理
    - 类有接口的时候，spring aop会使用jdk动态代理，生成一个跟你实现**同样接口**的一个代理类，构造一个实例对象出来
    - 通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是 java.lang.reflect.Proxy类。
    - 通俗理解：实现和目标类相同的接口，伪装成了和目标类一样的类（实现了同一接口，咱是兄弟了），也就逃过了类型检查，到java运行期的时候，利用多态的后期绑定，伪装类（代理类）就变成了接口的真正实现。代理类后来还是会把业务具体方法让目标类实现，只不过在这之前自己就处理好了其他事情（写日志，安全检查，事务等）。
  - cglib代理
    - 某个类没有实现接口的时候，spring aop会改用cglib来生成动态代理，生成你的类的一个子类，他可以动态生成字节码，覆盖你的一些方法，在方法里加入增强的代码
    - 如果目标类没有实现接口，那么 Spring AOP 会选择使用 CGLIB 来动态代理目标类。
    - CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB 是通过继承的方式做的动态代理，因此如果某个类被标记为 final，那么它是无法使用 CGLIB 做动态代理的。
    - 通俗理解：如果目标类就没实现某一接口呢，那jdk代理还怎么伪装！我就压根没有机会让你搞出这个长的一样的兄弟，那么就用第2种代理方式，创建一个目标类的子类，生个儿子，让儿子伪装我。spring使用CGLIB库生成目标类的一个子类，spring织入通知，并且把对这个子类的调用委托到目标类，在这之前子类还会实现目标类不关心的方法（写日志，安全检查，事务等）。
  - 通知
- 常用注解

- @Component 最普通的组件，表明可以被注入到 spring 容器中进行管理其他三个注解是这个注解的拓展，@Component+特定功能
- @Repository 表示和数据库相关的操作
- @Service 用于逻辑业务层
- @Controller 作用于表现层，spring-mvc 的注解
- @Autowired 注入bean
- @Configuration 声明当前类为配置类
- @ComponentScan：表示启用组件扫描
  - Spring会自动扫描所有通过注解配置的bean，然后将其注册到IOC容器中。我们可以通过 basePackages等属性来指定 @ComponentScan 自动扫描的范围，如果不指定，默认从声明 @ComponentScan 所在类的 package进行扫描。正因为如此，SpringBoot的启动类都默认在 src/main/java下。
- @Import 注解用于导入配置类
- @Conditional 按条件初始化
  - 表示在满足某种条件后才初始化一个bean或者启用某些配置。它一般用在由@Component、 @Service、 @Configuration等注解标识的类上面，或者由 @Bean标记的方法上。如果一个 @Configuration类标记了 @Conditional，则该类中所有标识了 @Bean的方法和 @Import注解导入的相关类将遵从这些条件。
  - 在Spring里可以很方便的编写你自己的条件类，所要做的就是实现 Condition接口，并覆盖它的 matches()方法。举个例子，下面的简单条件类表示只有在 Classpath里存在 JdbcTemplate类时才生效：

- JavaBean的生命周期
  - 图解1



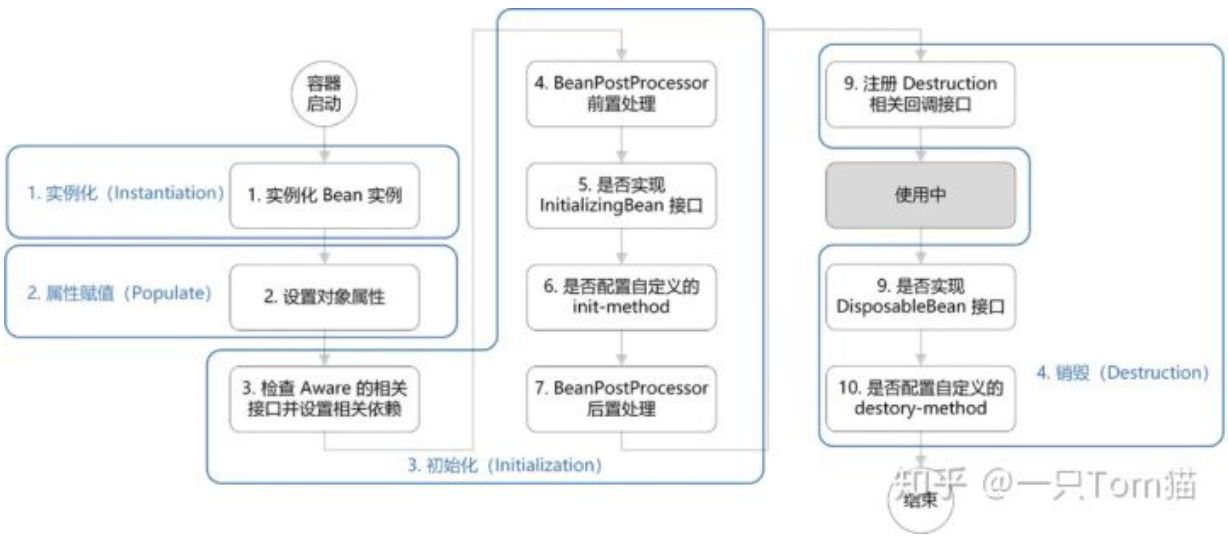
  - 图解2



- 1.Spring对Bean进行实例化（相当于程序中的new Xx()）
- 2.Spring将值和Bean的引用注入进Bean对应的属性中
- 3.如果Bean实现了BeanNameAware接口，Spring将Bean的ID传递给setBeanName()方法（实现BeanNameAware主要是为了通过Bean的引用来获得Bean的ID，一般业务中是很少有用到Bean的ID的）
- 4.如果Bean实现了BeanFactoryAware接口，Spring将调用setBeanFactory(BeanFactory bf)方法并把BeanFactory容器实例作为参数传入。（实现BeanFactoryAware 主要目的是为了获取Spring容器，如Bean通过Spring容器发布事件等）
- 5.如果Bean实现了ApplicationContextAwaer接口，Spring容器将调用setApplicationContext(ApplicationContext ctx)方法，把y应用上下文作为参数传入.(作用与BeanFactory类似都是为了获取Spring容器，不同的是Spring容器在调用setApplicationContext方法时会把它自己作为setApplicationContext 的参数传入，而Spring容器在调用setBeanDactory前需要程序员自己指定（注入）setBeanFactory里的参数BeanFactory)
- 6.如果Bean实现了BeanPostProcess接口，Spring将调用它们的postProcessBeforeInitialization（预初始化）方法（作用是在Bean实例创建成功后进行增强处理，如对Bean进行修改，增加某个功能）
- 7.如果Bean实现了InitializingBean接口，Spring将调用它们的afterPropertiesSet方法，作用与在配置文件中对Bean使用init-method声明初始化的作用一样，都是在Bean的全部属性设置成功后执行的初始化方法。
- 8.如果Bean实现了BeanPostProcess接口，Spring将调用它们的postProcessAfterInitialization（后初始化）方法（作用与6的一样，只不过6是在Bean初始化前执行的，而这个是在Bean初始化后执行的，时机不同)
- 9.经过以上的工作后，Bean将一直驻留在应用上下文中给应用使用，直到应用上下文被销毁
- 10.如果Bean实现了DispostbleBean接口，Spring将调用它的destory方法，作用与在配置文件中对Bean使用destory-method属性的作用一样，都是在Bean实例销毁前执行的方法。
- 其他版本

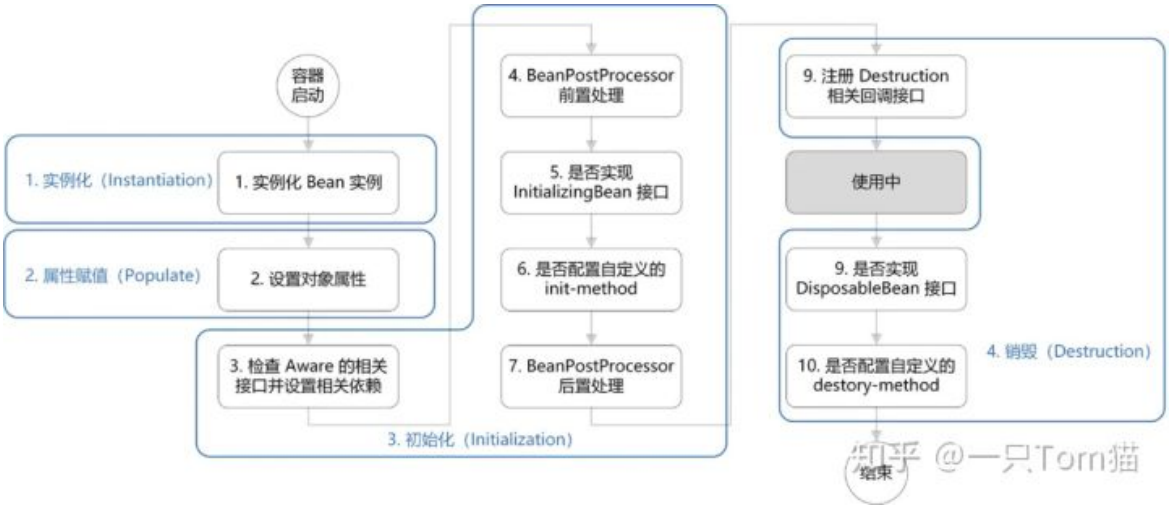
- （1）实例化Bean：如果要使用一个bean的话
- （2）设置对象属性（依赖注入）：他需要去看看，你的这个bean依赖了谁，把你依赖的bean也创建出来，给你进行一个注入
- （3）处理Aware接口：
  - 如果这个Bean已经实现了ApplicationContextAware接口，spring容器就会调用我们的bean的setApplicationContext(ApplicationContext)方法，传入Spring上下文，把spring容器给传递给这个bean
- （4）BeanPostProcessor：
  - 如果我们想在bean实例构建好了之后，此时我们想要对Bean进行一些自定义的处理，那么可以让Bean实现了BeanPostProcessor接口，那将会调用postProcessBeforeInitialization(Object obj, String s)方法。
- （5）InitializingBean 与 init-method：
  - 如果Bean在Spring配置文件中配置了 init-method 属性，则会自动调用其配置的初始化方法。
- （6）如果这个Bean实现了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法
- （7）DisposableBean：
  - 当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用其实现的destroy()方法；
- （8）destroy-method：

- bean的生命周期
  - 由IoC容器管理的那些对象我们就叫它Bean， Bean就是由Spring容器初始化、装配及管理的对象
  - 图视



  - 4个阶段
    - 1）实例化（Instantiation）
    - 2）属性赋值（Populate）

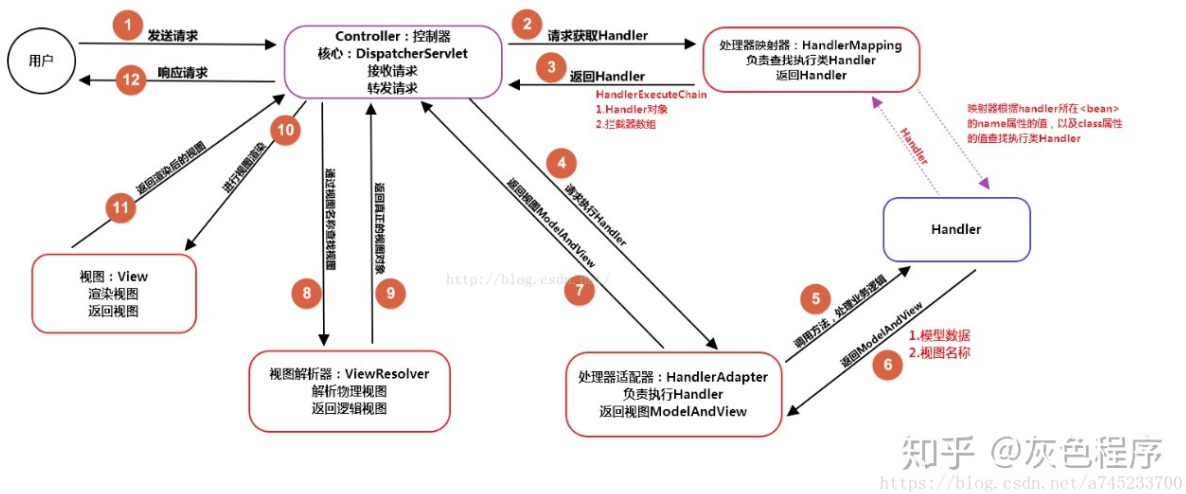
    - 3）初始化（Initiallization）
    - 4）销毁（Destruction）
- 具体流程
    - 实例化：第 1 步，实例化一个 bean 对象；
    - 属性赋值：第 2 步，为 bean 设置相关属性和依赖；
    - 初始化：第 3~7 步，步骤较多，其中第 5、6 步为初始化操作，第 3、4 步为在初始化前执行，第 7 步在初始化后执行，该阶段结束，才能被用户使用；
    - 销毁：第 8~10步，第8步不是真正意义上的销毁（还没使用呢），而是先在使用前注册了销毁的相关调用接口，为了后面第9、10步真正销毁 bean 时再执行相应的方法。
    - 流程图



- 描述
    - bean的生命周期主要可以分为四个部分，也就是实例化createBeanInstance、属性赋值populateBean、初始化initalizeBean和销毁Destruction，在这个过程中会出现一些对bean的拓展
    - 在初始化的过程中
        - 首先会检查是否实现了相关的Aware接口，如果实现了的话会设置相关依赖，比如说：
            - BeanNameAware：注入当前 bean 对应 beanName；
            - BeanClassLoaderAware：注入加载当前 bean 的 ClassLoader；
            - BeanFactoryAware：注入 当前BeanFactory容器 的引用。
        - 然后是一个BeanPostProcessor 前置处理
            - 对于 ApplicationContext 类型的容器，也提供了 Aware 接口，只不过这些 Aware 接口的注入实现，是通过 BeanPostProcessor 的方式注入的，但其作用仍是注入依赖。
            - EnvironmentAware：注入 Enviroment，一般用于获取配置属性；
            - ApplicationContextAware（ResourceLoader、ApplicationEventPublisherAware、MessageSourceAware）：注入 ApplicationContext 容器本身。

        - 若实现 InitializingBean 接口，调用 afterPropertiesSet() 方法
        - 若配置自定义的 init-method方法来指定初始化的方法，则执行
        - 然后使用BeanPostProcessor后置处理
    - 在销毁过程中，
        - 若实现 DisposableBean 接口，则执行 destory()方法
        - 若配置自定义的 detory-method 方法，执行
- Spring Bean的作用域
    - singleton（默认）
        - 单例模式，在整个Spring IoC容器中，使用singleton定义的Bean将只有一个实例
    - prototype
        - 原型模式，为每一个bean请求提供一个实例（通常不用）
    - request
        - 对于每次HTTP请求，使用request定义的Bean都将产生一个新实例，即每次HTTP请求将会产生不同的Bean实例。在请求完成以后，bean会失效并被垃圾回收器回收
        - 只有在Web应用中使用Spring时，该作用域才有效
    - session
        - 与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效
    - globalsession
        - 每个全局的HTTP Session，使用session定义的Bean都将产生一个新实例。同样只有在Web应用中使用Spring时，该作用域才有效
- Spring事务
    - 事务实现原理
        - 加一个@Transactional注解，此时就spring会使用AOP思想
        - 对你的这个方法在执行之前，先去开启事务，执行完毕之后，根据你方法是否报错，来决定回滚还是提交事务
    - 自己通过代码实现：编程式事务
    - 通过注解声明：声明式事务（通过代理实现）
    - 事务传播机制
        - requered：强制要求要有一个物理事务。如果没有已经存在的事务，就专门打开一个事务用于当前范围。
        - requires_new：总是使用一个独立的物理事务用于每一个受影响的逻辑事务范围，从来不参与到一个已存在的外围事务范围。
        - nested：有事务就创建一个子事务（嵌套事务），没有就创建一个普通事务
        - support：支持事务执行，没有就普通执行

    - not_support：不支持事务执行，有事务就把事务挂起
    - mandatory：一定要有事务，没有就抛出异常
    - never：不支持事务，有事务就抛出异常
    - 具体问题
        - 一段业务逻辑，方法A调用方法B，我希望的是如果说方法A出错了，此时仅仅回滚方法A，不能回滚方法B，必须得用REQUIRES_NEW，传播机制，让他们俩的事务是不同的
        - 方法A调用方法B，如果出错，方法B只能回滚他自己，方法A可以带着方法B一起回滚，NESTED嵌套事务
- Spring MVC执行流程
    - 图视



    - （1）用户发送请求至前端控制器DispatcherServlet；
    - （2） DispatcherServlet收到请求后，调用HandlerMapping处理器映射器，请求获取Handle；
    - （3）处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet；
    - （4）DispatcherServlet 调用 HandlerAdapter处理器适配器；
    - （5）HandlerAdapter 经过处理器适配器适配调用 具体处理器(Handler，也叫后端控制器)；
    - （6）Handler执行完成返回ModelAndView；
    - （7）HandlerAdapter将Handler执行结果ModelAndView返回给DispatcherServlet；
    - （8）DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行解析；
    - （9）ViewResolver解析后返回具体View；
    - （10）DispatcherServlet对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。
- SpringMVC细节问题

- SpringMVC请求方法
  - get/post/put/delete
  - @RequestMapping(value = "/testRest/{id}", method = RequestMethod.GET) 设置请求方法

- Spring Boot
  - 简省了繁重的配置，提供了各种启动器
- Spring Boot自动加载（过程）
  - 1、Spring Boot应用的启动类一般均位于 src/main/java根路径下。其中@SpringBootApplication开启组件扫描和自动配置，而 SpringApplication.run则负责启动引导应用程序。@SpringBootApplication是一个复合 Annotation，它将三个有用的注解组合在一起。

```
@SpringBootApplication
public class MoonApplication {

    public static void main(String[] args) {
        SpringApplication.run(MoonApplication.class, args);
    }
}
```

  - 2、@SpringBootConfiguration就是 @Configuration，它是Spring框架的注解，标明该类是一个 JavaConfig配置类。而 @ComponentScan启用组件扫描，前文已经详细讲解过，这里着重关注 @EnableAutoConfiguration。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filte (type = FilterTyp .CUSTOM, classes = TypeExcludeFilte .clas )
        r              e                              r                s    ,
       @Filt (type  FilterTy.CUSTOM, classe AutoConfigurationExclude .clas) }
       er    =      pe      s =            Filter                    s    )
public @interface SpringBootApplication {
    // ......
}
```

  - 3、@EnableAutoConfiguration注解表示开启Spring Boot自动配置功能，Spring Boot会根据应用的依赖、自定义的bean、classpath下有没有某个类 等等因素来猜测你需要的bean，然后注册到IOC容器中。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    // ......
}
```

- @Import(EnableAutoConfigurationImportSelector.class)上了，前文说过，@Import注解用于导入类，并将这个类作为一个bean的定义注册到容器中，这里它将把EnableAutoConfigurationImportSelector作为bean注入到容器中，而这个类会将所有符合条件的@Configuration配置都加载到容器中
- 4、这个类会扫描所有的jar包，将所有符合条件的@Configuration配置类注入的容器中

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    // 省略了大部分代码，保留一句核心代码
    // 注意：SpringBoot最近版本中，这句代码被封装在一个单独的方法中
    // SpringFactoriesLoader相关知识请参考前文
    Lis <Strin > factories = ne  ArrayLis <Strin > ne  LinkedHashSe <Strin >(
    t    g                   w   t         g     ( w   t             g
      SpringFactories .loadFactoryN EnableAutoConfigu .cla , thi.beanClassLoade
      Loader          ames(         ration            ss    s   r)));
}
```

```
// 来自 org.springframework.boot.autoconfigure下的META-INF/spring.factories
// 配置的key = EnableAutoConfiguration，与代码中一致
org.springframework.boot.autoconfigure.EnableAutoConfiguration=
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration
.....
```