* [Spring的优点](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#spring%E7%9A%84%E4%BC%98%E7%82%B9)
* [Spring 用到了哪些设计模式？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#spring-%E7%94%A8%E5%88%B0%E4%BA%86%E5%93%AA%E4%BA%9B%E8%AE%BE%E8%AE%A1%E6%A8%A1%E5%BC%8F)
* [什么是AOP？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#%E4%BB%80%E4%B9%88%E6%98%AFaop)
* [AOP有哪些实现方式？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#aop%E6%9C%89%E5%93%AA%E4%BA%9B%E5%AE%9E%E7%8E%B0%E6%96%B9%E5%BC%8F)
* [JDK动态代理和CGLIB动态代理的区别？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#jdk%E5%8A%A8%E6%80%81%E4%BB%A3%E7%90%86%E5%92%8Ccglib%E5%8A%A8%E6%80%81%E4%BB%A3%E7%90%86%E7%9A%84%E5%8C%BA%E5%88%AB)
* [Spring AOP相关术语](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#spring-aop%E7%9B%B8%E5%85%B3%E6%9C%AF%E8%AF%AD)
* [Spring通知有哪些类型？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#spring%E9%80%9A%E7%9F%A5%E6%9C%89%E5%93%AA%E4%BA%9B%E7%B1%BB%E5%9E%8B)
* [什么是IOC？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#%E4%BB%80%E4%B9%88%E6%98%AFioc)
* [IOC的优点是什么？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#ioc%E7%9A%84%E4%BC%98%E7%82%B9%E6%98%AF%E4%BB%80%E4%B9%88)
* [什么是依赖注入？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#%E4%BB%80%E4%B9%88%E6%98%AF%E4%BE%9D%E8%B5%96%E6%B3%A8%E5%85%A5)
* [IOC容器初始化过程？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#ioc%E5%AE%B9%E5%99%A8%E5%88%9D%E5%A7%8B%E5%8C%96%E8%BF%87%E7%A8%8B)
* [Bean的生命周期](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#bean%E7%9A%84%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F)
* [BeanFactory和FactoryBean的区别？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#beanfactory%E5%92%8Cfactorybean%E7%9A%84%E5%8C%BA%E5%88%AB)
* [Bean注入容器有哪些方式？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#bean%E6%B3%A8%E5%85%A5%E5%AE%B9%E5%99%A8%E6%9C%89%E5%93%AA%E4%BA%9B%E6%96%B9%E5%BC%8F)
* [Bean的作用域](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#bean%E7%9A%84%E4%BD%9C%E7%94%A8%E5%9F%9F)
* [Spring自动装配的方式有哪些？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#spring%E8%87%AA%E5%8A%A8%E8%A3%85%E9%85%8D%E7%9A%84%E6%96%B9%E5%BC%8F%E6%9C%89%E5%93%AA%E4%BA%9B)
* [@Autowired和@Resource的区别？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#autowired%E5%92%8Cresource%E7%9A%84%E5%8C%BA%E5%88%AB)
* [@Qualifier 注解有什么作用](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#qualifier-%E6%B3%A8%E8%A7%A3%E6%9C%89%E4%BB%80%E4%B9%88%E4%BD%9C%E7%94%A8)
* [@Bean和@Component有什么区别？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#bean%E5%92%8Ccomponent%E6%9C%89%E4%BB%80%E4%B9%88%E5%8C%BA%E5%88%AB)
* [@Component、@Controller、@Repositor和@Service 的区别？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#componentcontrollerrepositor%E5%92%8Cservice-%E7%9A%84%E5%8C%BA%E5%88%AB)
* [Spring 事务实现方式有哪些？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#spring-%E4%BA%8B%E5%8A%A1%E5%AE%9E%E7%8E%B0%E6%96%B9%E5%BC%8F%E6%9C%89%E5%93%AA%E4%BA%9B)
* [有哪些事务传播行为？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#%E6%9C%89%E5%93%AA%E4%BA%9B%E4%BA%8B%E5%8A%A1%E4%BC%A0%E6%92%AD%E8%A1%8C%E4%B8%BA)
* [Spring怎么解决循环依赖的问题？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#spring%E6%80%8E%E4%B9%88%E8%A7%A3%E5%86%B3%E5%BE%AA%E7%8E%AF%E4%BE%9D%E8%B5%96%E7%9A%84%E9%97%AE%E9%A2%98)
* [Spring启动过程](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#spring%E5%90%AF%E5%8A%A8%E8%BF%87%E7%A8%8B)
* [Spring 的单例 Bean 是否有线程安全问题？](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E9%9D%A2%E8%AF%95%E9%A2%98.md#spring-%E7%9A%84%E5%8D%95%E4%BE%8B-bean-%E6%98%AF%E5%90%A6%E6%9C%89%E7%BA%BF%E7%A8%8B%E5%AE%89%E5%85%A8%E9%97%AE%E9%A2%98)

**Spring的优点**

* **轻量**，基本版本大约2MB。
* 通过控制反转和依赖注入实现**松耦合**。
* 支持**面向切面**的编程，并且把应用业务逻辑和系统服务分开。
* 通过切面和模板减少样板式代码。
* 方便集成各种优秀框架。内部提供了对各种优秀框架的直接支持（如：Hibernate、MyBatis等）。
* 方便程序的测试。Spring支持Junit4，添加注解便可以测试Spring程序。

**Spring 用到了哪些设计模式？**

1、**简单工厂模式**：BeanFactory就是简单工厂模式的体现，根据传入一个唯一标识来获得 Bean 对象。

@Override

public Object getBean(String name) throws BeansException {

assertBeanFactoryActive();

return getBeanFactory().getBean(name);

}

2、**工厂方法模式**：FactoryBean就是典型的工厂方法模式。spring在使用getBean()调用获得该bean时，会自动调用该bean的getObject()方法。每个 Bean 都会对应一个 FactoryBean，如 SqlSessionFactory 对应 SqlSessionFactoryBean。

3、**单例模式**：一个类仅有一个实例，提供一个访问它的全局访问点。Spring 创建 Bean 实例默认是单例的。

4、**适配器模式**：SpringMVC中的适配器HandlerAdatper。由于应用会有多个Controller实现，如果需要直接调用Controller方法，那么需要先判断是由哪一个Controller处理请求，然后调用相应的方法。当增加新的 Controller，需要修改原来的逻辑，违反了开闭原则（对修改关闭，对扩展开放）。

为此，Spring提供了一个适配器接口，每一种 Controller 对应一种 HandlerAdapter 实现类，当请求过来，SpringMVC会调用getHandler()获取相应的Controller，然后获取该Controller对应的 HandlerAdapter，最后调用HandlerAdapter的handle()方法处理请求，实际上调用的是Controller的handleRequest()。每次添加新的 Controller 时，只需要增加一个适配器类就可以，无需修改原有的逻辑。

常用的处理器适配器：SimpleControllerHandlerAdapter，HttpRequestHandlerAdapter，AnnotationMethodHandlerAdapter。

// Determine handler for the current request.

mappedHandler = getHandler(processedRequest);

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

// Actually invoke the handler.

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

public class HttpRequestHandlerAdapter implements HandlerAdapter {

@Override

public boolean supports(Object handler) {//handler是被适配的对象，这里使用的是对象的适配器模式

return (handler instanceof HttpRequestHandler);

}

@Override

@Nullable

public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)

throws Exception {

((HttpRequestHandler) handler).handleRequest(request, response);

return null;

}

}

5、**代理模式**：spring 的 aop 使用了动态代理，有两种方式JdkDynamicAopProxy和Cglib2AopProxy。

6、**观察者模式**：spring 中 observer 模式常用的地方是 listener 的实现，如ApplicationListener。

7、**模板模式**： Spring 中 jdbcTemplate、hibernateTemplate 等，就使用到了模板模式。

**什么是AOP？**

面向切面编程，作为面向对象的一种补充，将公共逻辑（事务管理、日志、缓存等）封装成切面，跟业务代码进行分离，可以减少系统的重复代码和降低模块之间的耦合度。切面就是那些与业务无关，但所有业务模块都会调用的公共逻辑。

**AOP有哪些实现方式？**

AOP有两种实现方式：静态代理和动态代理。

**静态代理**

静态代理：代理类在编译阶段生成，在编译阶段将通知织入Java字节码中，也称编译时增强。AspectJ使用的是静态代理。

缺点：代理对象需要与目标对象实现一样的接口，并且实现接口的方法，会有冗余代码。同时，一旦接口增加方法，目标对象与代理对象都要维护。

**动态代理**

动态代理：代理类在程序运行时创建，AOP框架不会去修改字节码，而是在内存中临时生成一个代理对象，在运行期间对业务方法进行增强，不会生成新类。

**JDK动态代理和CGLIB动态代理的区别？**

Spring AOP中的动态代理主要有两种方式：JDK动态代理和CGLIB动态代理。

**JDK动态代理**

如果目标类实现了接口，Spring AOP会选择使用JDK动态代理目标类。代理类根据目标类实现的接口动态生成，不需要自己编写，生成的动态代理类和目标类都实现相同的接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。

缺点：目标类必须有实现的接口。如果某个类没有实现接口，那么这个类就不能用JDK动态代理。

**CGLIB来动态代理**

通过继承实现。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）可以在运行时动态生成类的字节码，动态创建目标类的子类对象，在子类对象中增强目标类。

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。

优点：目标类不需要实现特定的接口，更加灵活。

什么时候采用哪种动态代理？

1. 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
2. 如果目标对象实现了接口，可以强制使用CGLIB实现AOP
3. 如果目标对象没有实现了接口，必须采用CGLIB库

**两者的区别**：

1. jdk动态代理使用jdk中的类Proxy来创建代理对象，它使用反射技术来实现，不需要导入其他依赖。cglib需要引入相关依赖：asm.jar，它使用字节码增强技术来实现。
2. 当目标类实现了接口的时候Spring Aop默认使用jdk动态代理方式来增强方法，没有实现接口的时候使用cglib动态代理方式增强方法。

**Spring AOP相关术语**

（1）**切面（Aspect）**：切面是通知和切点的结合。通知和切点共同定义了切面的全部内容。

（2）连接点（Join point）：指方法，在Spring AOP中，一个连接点总是代表一个方法的执行。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

（3）通知（Advice）：在AOP术语中，切面的工作被称为通知。

（4）切入点（Pointcut）：切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。

（5）引入（Introduction）：引入允许我们向现有类添加新方法或属性。

（6）目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。它通常是一个代理对象。

（7）织入（Weaving）：织入是把切面应用到目标对象并创建新的代理对象的过程。在目标对象的生命周期里有以下时间点可以进行织入：

* 编译期：切面在目标类编译时被织入。AspectJ的织入编译器是以这种方式织入切面的。
* 类加载期：切面在目标类加载到JVM时被织入。需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ5的加载时织入就支持以这种方式织入切面。
* 运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。SpringAOP就是以这种方式织入切面。

**Spring通知有哪些类型？**

在AOP术语中，切面的工作被称为通知。通知实际上是程序运行时要通过Spring AOP框架来触发的代码段。

Spring切面可以应用5种类型的通知：

1. 前置通知（Before）：在目标方法被调用之前调用通知功能；
2. 后置通知（After）：在目标方法完成之后调用通知，此时不会关心方法的输出是什么；
3. 返回通知（After-returning ）：在目标方法成功执行之后调用通知；
4. 异常通知（After-throwing）：在目标方法抛出异常后调用通知；
5. 环绕通知（Around）：通知包裹了被通知的方法，在被通知的方法调用之前和调用之后执行自定义的逻辑。

**什么是IOC？**

IOC：控制反转，由Spring容器管理bean的整个生命周期。通过反射实现对其他对象的控制，包括初始化、创建、销毁等，解放手动创建对象的过程，同时降低类之间的耦合度。

IOC的好处：降低了类之间的耦合，对象创建和初始化交给Spring容器管理，在需要的时候只需向容器进行申请。

**IOC的优点是什么？**

* IOC 和依赖注入降低了应用的代码量。
* 松耦合。
* 支持加载服务时的饿汉式初始化和懒加载。

**什么是依赖注入？**

在Spring创建对象的过程中，把对象依赖的属性注入到对象中。依赖注入主要有两种方式：构造器注入和属性注入。

**IOC容器初始化过程？**

ioc 容器初始化过程：BeanDefinition 的资源定位、解析和注册。

1. 从XML中读取配置文件。
2. 将bean标签解析成 BeanDefinition，如解析 property 元素， 并注入到 BeanDefinition 实例中。
3. 将 BeanDefinition 注册到容器 BeanDefinitionMap 中。
4. BeanFactory 根据 BeanDefinition 的定义信息创建实例化和初始化 bean。

单例bean的初始化以及依赖注入一般都在容器初始化阶段进行，只有懒加载（lazy-init为true）的单例bean是在应用第一次调用getBean()时进行初始化和依赖注入。

// AbstractApplicationContext

// Instantiate all remaining (non-lazy-init) singletons.

finishBeanFactoryInitialization(beanFactory);

多例bean 在容器启动时不实例化，即使设置 lazy-init 为 false 也没用，只有调用了getBean()才进行实例化。

loadBeanDefinitions采用了模板模式，具体加载 BeanDefinition 的逻辑由各个子类完成。

**Bean的生命周期**

1.调用bean的构造方法创建Bean

2.通过反射调用setter方法进行属性的依赖注入

3.如果Bean实现了BeanNameAware接口，Spring将调用setBeanName()，设置 Bean的name（xml文件中bean标签的id）

4.如果Bean实现了BeanFactoryAware接口，Spring将调用setBeanFactory()把bean factory设置给Bean

5.如果Bean实现了ApplicationContextAware接口，Spring容器将调用setApplicationContext()给Bean设置ApplictionContext

6.如果存在BeanPostProcessor，Spring将调用它们的postProcessBeforeInitialization（预初始化）方法，在Bean初始化前对其进行处理

7.如果Bean实现了InitializingBean接口，Spring将调用它的afterPropertiesSet方法，然后调用xml定义的 init-method 方法，两个方法作用类似，都是在初始化 bean 的时候执行

8.如果存在BeanPostProcessor，Spring将调用它们的postProcessAfterInitialization（后初始化）方法，在Bean初始化后对其进行处理

9.Bean初始化完成，供应用使用，直到应用被销毁

10.如果Bean实现了DisposableBean接口，Spring将调用它的destory方法，然后调用在xml中定义的 destory-method方法，这两个方法作用类似，都是在Bean实例销毁前执行

public interface BeanPostProcessor {

@Nullable

default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

return bean;

}

@Nullable

default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

return bean;

}

}

public interface InitializingBean {

void afterPropertiesSet() throws Exception;

}

**BeanFactory和FactoryBean的区别？**

BeanFactory：管理Bean的容器，Spring中生成的Bean都是由这个接口的实现来管理的。

FactoryBean：通常是用来创建比较复杂的bean，一般的bean 直接用xml配置即可，但如果一个bean的创建过程中涉及到很多其他的bean 和复杂的逻辑，直接用xml配置比较麻烦，这时可以考虑用FactoryBean，可以隐藏实例化复杂Bean的细节。

当配置文件中bean标签的class属性配置的实现类是FactoryBean时，通过 getBean()方法返回的不是FactoryBean本身，而是调用FactoryBean#getObject()方法所返回的对象，相当于FactoryBean#getObject()代理了getBean()方法。如果想得到FactoryBean必须使用 '&' + beanName 的方式获取。

Mybatis 提供了 SqlSessionFactoryBean，可以简化 SqlSessionFactory 的配置：

public class SqlSessionFactoryBean implements FactoryBean<SqlSessionFactory>, InitializingBean, ApplicationListener<ApplicationEvent> {

@Override

public void afterPropertiesSet() throws Exception {

notNull(dataSource, "Property 'dataSource' is required");

notNull(sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is required");

state((configuration == null && configLocation == null) || !(configuration != null && configLocation != null),

"Property 'configuration' and 'configLocation' can not specified with together");

this.sqlSessionFactory = buildSqlSessionFactory();

}

protected SqlSessionFactory buildSqlSessionFactory() throws IOException {

//复杂逻辑

}

@Override

public SqlSessionFactory getObject() throws Exception {

if (this.sqlSessionFactory == null) {

afterPropertiesSet();

}

return this.sqlSessionFactory;

}

}

在 xml 配置 SqlSessionFactoryBean：

<bean id="tradeSqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="trade" />

<property name="mapperLocations" value="classpath\*:mapper/trade/\*Mapper.xml" />

<property name="configLocation" value="classpath:mybatis-config.xml" />

<property name="typeAliasesPackage" value="com.bytebeats.mybatis3.domain.trade" />

</bean>

Spring 将会在应用启动时创建 SqlSessionFactory，并使用 sqlSessionFactory 这个名字存储起来。

**Bean注入容器有哪些方式？**

将普通类交给Spring容器管理，通常有以下方法：

1、使用@Configuration与@Bean注解

2、使用@Controller、@Service、@Repository、@Component 注解标注该类，然后启用@ComponentScan自动扫描

3、使用@Import 方法。使用@Import注解把bean导入到当前容器中，代码如下：

//@SpringBootApplication

@ComponentScan

/\*把用到的资源导入到当前容器中\*/

@Import({Dog.class, Cat.class})

public class App {

public static void main(String[] args) throws Exception {

ConfigurableApplicationContext context = SpringApplication.run(App.class, args);

System.out.println(context.getBean(Dog.class));

System.out.println(context.getBean(Cat.class));

context.close();

}

}

**Bean的作用域**

1、singleton：单例，Spring中的bean默认都是单例的。

2、prototype：每次请求都会创建一个新的bean实例。

3、request：每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。

4、session：每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP session内有效。

5、global-session：全局session作用域。

**Spring自动装配的方式有哪些？**

Spring的自动装配有三种模式：byType(根据类型)，byName(根据名称)、constructor(根据构造函数)。

**byType**

找到与依赖类型相同的bean注入到另外的bean中，这个过程需要借助setter注入来完成，因此必须存在set方法，否则注入失败。

当xml文件中存在多个相同类型名称不同的实例Bean时，Spring容器依赖注入仍然会失败，因为存在多种适合的选项，Spring容器无法知道该注入那种，此时我们需要为Spring容器提供帮助，指定注入那个Bean实例。可以通过＜bean＞标签的autowire-candidate设置为false来过滤那些不需要注入的实例Bean

<bean id="userDao" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<!-- autowire-candidate="false" 过滤该类型 -->

<bean id="userDao2" autowire-candidate="false" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<!-- byType 根据类型自动装配userDao-->

<bean id="userService" autowire="byType" class="com.zejian.spring.springIoc.service.impl.UserServiceImpl" />

**byName**

将属性名与bean名称进行匹配，如果找到则注入依赖bean。

<bean id="userDao" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<bean id="userDao2" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<!-- byName 根据名称自动装配，找到UserServiceImpl名为 userDao属性并注入-->

<bean id="userService" autowire="byName" class="com.zejian.spring.springIoc.service.impl.UserServiceImpl" />

**constructor**

存在单个实例则优先按类型进行参数匹配（无论名称是否匹配），当存在多个类型相同实例时，按名称优先匹配，如果没有找到对应名称，则注入失败，此时可以使用autowire-candidate=”false” 过滤来解决。

<!--只存在userDao2，userDao3 无法成功注入-->

<bean id="userDao2" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<bean id="userDao3" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<bean id="userService" autowire="constructor" class="com.zejian.spring.springIoc.service.impl.UserServiceImpl" />

@Autowired 可以传递了一个required=false的属性，false指明当userDao实例存在就注入不存就忽略，如果为true，就必须注入，若userDao实例不存在，就抛出异常。由于默认情况下@Autowired是按类型匹配的(byType)，如果需要按名称(byName)匹配的话，可以使用@Qualifier注解与@Autowired结合。

public class UserServiceImpl implements UserService {

//标注成员变量

@Autowired

@Qualifier("userDao1")

private UserDao userDao;

}

**byName模式 xml 配置：**

<!-- 根据@Qualifier("userDao1")自动识别 -->

<bean id="userDao1" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<bean id="userDao2" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<bean id="userService" class="com.zejian.spring.springIoc.service.impl.UserServiceImpl" />

@Resource，默认按 byName模式自动注入。@Resource有两个中重要的属性：name和type。Spring容器对于@Resource注解的name属性解析为bean的名字，type属性则解析为bean的类型。因此使用name属性，则按byName模式的自动注入策略，如果使用type属性则按 byType模式自动注入策略。倘若既不指定name也不指定type属性，Spring容器将通过反射技术默认按byName模式注入。

@Resource(name=“userDao”)

private UserDao userDao;//用于成员变量

//也可以用于set方法标注

@Resource(name=“userDao”)

public void setUserDao(UserDao userDao) {

this.userDao= userDao;

}

上述两种自动装配的依赖注入并不适合简单值类型，如int、boolean、long、String以及Enum等，对于这些类型，Spring容器也提供了@Value注入的方式。@Value接收一个String的值，该值指定了将要被注入到内置的java类型属性值，Spring 容器会做好类型转换。一般情况下@Value会与properties文件结合使用。

jdbc.properties文件如下：

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://127.0.0.1:3306/test?characterEncoding=UTF-8&allowMultiQueries=true

jdbc.username=root

jdbc.password=root

利用注解@Value获取jdbc.url和jdbc.username的值，实现如下：

public class UserServiceImpl implements UserService {

//占位符方式

@Value("${jdbc.url}")

private String url;

//SpEL表达方式，其中代表xml配置文件中的id值configProperties

@Value("#{configProperties['jdbc.username']}")

private String userName;

}

xml配置文件：

<!--基于占位符方式 配置单个properties -->

<!--<context:property-placeholder location="conf/jdbc.properties"/>-->

<!--基于占位符方式 配置多个properties -->

<bean id="propertyConfigurer" class="org.springframework.beans.factory.config.PreferencesPlaceholderConfigurer">

<property name="location" value="conf/jdbc.properties"/>

</bean>

<!--基于SpEL表达式 配置多个properties id值为configProperties 提供java代码中使用 -->

<bean id="configProperties" class="org.springframework.beans.factory.config.PropertiesFactoryBean">

<property name="locations">

<list>

<value>classpath:/conf/jdbc.properties</value>

</list>

</property>

</bean>

**@Autowired和@Resource的区别？**

@Autowired注解是按照类型（byType）装配依赖对象的,但是存在多个类型⼀致的bean，⽆法通过byType注⼊时，就会再使⽤byName来注⼊，如果还是⽆法判断注⼊哪个bean则会UnsatisfiedDependencyException。 @Resource会⾸先按照byName来装配，如果找不到bean，会⾃动byType再找⼀次。

**@Qualifier 注解有什么作用**

当需要创建多个相同类型的 bean 并希望仅使用属性装配其中一个 bean 时，可以使用@Qualifier 注解和 @Autowired 通过指定应该装配哪个 bean 来消除歧义。

**@Bean和@Component有什么区别？**

都是使用注解定义 Bean。@Bean 是使用 Java 代码装配 Bean，@Component 是自动装配 Bean。

@Component 注解用在类上，表明一个类会作为组件类，并告知Spring要为这个类创建bean，每个类对应一个 Bean。

@Bean 注解用在方法上，表示这个方法会返回一个 Bean。@Bean 需要在配置类中使用，即类上需要加上@Configuration注解。

@Component

public class Student {

private String name = "lkm";

public String getName() {

return name;

}

}

@Configuration

public class WebSocketConfig {

@Bean

public Student student(){

return new Student();

}

}

@Bean 注解更加灵活。当需要将第三方类装配到 Spring 容器中，因为没办法源代码上添加@Component注解，只能使用@Bean 注解的方式，当然也可以使用 xml 的方式。

**@Component、@Controller、@Repositor和@Service 的区别？**

@Component：最普通的组件，可以被注入到spring容器进行管理。

@Controller：将类标记为 Spring Web MVC 控制器。

@Service：将类标记为业务层组件。

@Repository：将类标记为数据访问组件，即DAO组件。

**Spring 事务实现方式有哪些？**

事务就是一系列的操作原子执行。Spring事务机制主要包括声明式事务和编程式事务。

* 编程式事务：通过编程的方式管理事务，这种方式带来了很大的灵活性，但很难维护。
* 声明式事务：将事务管理代码从业务方法中分离出来，通过aop进行封装。Spring声明式事务使得我们无需要去处理获得连接、关闭连接、事务提交和回滚等这些操作。使用 @Transactional 注解开启声明式事务。

@Transactional相关属性如下：

| **属性** | **类型** | **描述** |
| --- | --- | --- |
| value | String | 可选的限定描述符，指定使用的事务管理器 |
| propagation | enum: Propagation | 可选的事务传播行为设置 |
| isolation | enum: Isolation | 可选的事务隔离级别设置 |
| readOnly | boolean | 读写或只读事务，默认读写 |
| timeout | int (in seconds granularity) | 事务超时时间设置 |
| rollbackFor | Class对象数组，必须继承自Throwable | 导致事务回滚的异常类数组 |
| rollbackForClassName | 类名数组，必须继承自Throwable | 导致事务回滚的异常类名字数组 |
| noRollbackFor | Class对象数组，必须继承自Throwable | 不会导致事务回滚的异常类数组 |
| noRollbackForClassName | 类名数组，必须继承自Throwable | 不会导致事务回滚的异常类名字数组 |

**有哪些事务传播行为？**

在TransactionDefinition接口中定义了七个事务传播行为：

1. PROPAGATION\_**REQUIRED**如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。如果嵌套调用的两个方法都加了事务注解，并且运行在相同线程中，则这两个方法使用相同的事务中。如果运行在不同线程中，则会开启新的事务。
2. PROPAGATION\_**SUPPORTS**如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行。
3. PROPAGATION**\_MANDATORY**如果已经存在一个事务，支持当前事务。如果不存在事务，则抛出异常IllegalTransactionStateException。
4. PROPAGATION\_**REQUIRES\_NEW** 总是开启一个新的事务。需要使用JtaTransactionManager作为事务管理器。
5. PROPAGATION\_**NOT\_SUPPORTED** 总是非事务地执行，并挂起任何存在的事务。需要使用JtaTransactionManager作为事务管理器。
6. PROPAGATION\_**NEVER**总是非事务地执行，如果存在一个活动事务，则抛出异常。
7. PROPAGATION\_**NESTED**如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务, 则按PROPAGATION\_REQUIRED 属性执行。

**PROPAGATION\_NESTED 与PROPAGATION\_REQUIRES\_NEW的区别:**

使用PROPAGATION\_REQUIRES\_NEW时，内层事务与外层事务是两个独立的事务。一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。

使用PROPAGATION\_NESTED时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。

**Spring怎么解决循环依赖的问题？**

构造器注入的循环依赖：**Spring处理不了，直接抛出BeanCurrentlylnCreationException异常**。

**单例模式**下属性注入的循环依赖：**通过三级缓存处理循环依赖**。

非单例循环依赖：无法处理。

下面分析单例模式下属性注入的循环依赖是怎么处理的：

首先，Spring单例对象的初始化大略分为三步：

1. createBeanInstance：实例化bean，使用构造方法创建对象，为对象分配内存。
2. populateBean：进行依赖注入。
3. initializeBean：初始化bean。

Spring为了解决单例的循环依赖问题，使用了三级缓存：

singletonObjects：完成了初始化的单例对象map，bean name --> bean instance

earlySingletonObjects ：完成实例化未初始化的单例对象map，bean name --> bean instance

singletonFactories ： 单例对象工厂map，bean name --> ObjectFactory，单例对象实例化完成之后会加入singletonFactories。

在调用createBeanInstance进行实例化之后，会调用addSingletonFactory，将单例对象放到singletonFactories中。

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);

}

}

}

假如A依赖了B的实例对象，同时B也依赖A的实例对象。

1. A首先完成了实例化，并且将自己添加到singletonFactories中
2. 接着进行依赖注入，发现自己依赖对象B，此时就尝试去get(B)
3. 发现B还没有被实例化，对B进行实例化
4. 然后B在初始化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和二级缓存earlySingletonObjects没找到，尝试三级缓存singletonFactories，由于A初始化时将自己添加到了singletonFactories，所以B可以拿到A对象，然后将A从三级缓存中移到二级缓存中
5. B拿到A对象后顺利完成了初始化，然后将自己放入到一级缓存singletonObjects中
6. 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

由此看出，属性注入的循环依赖主要是通过将实例化完成的bean添加到singletonFactories来实现的。而使用构造器依赖注入的bean在实例化的时候会进行依赖注入，不会被添加到singletonFactories中。比如A和B都是通过构造器依赖注入，A在调用构造器进行实例化的时候，发现自己依赖B，B没有被实例化，就会对B进行实例化，此时A未实例化完成，不会被添加到singtonFactories。而B依赖于A，B会去三级缓存寻找A对象，发现不存在，于是又会实例化A，A实例化了两次，从而导致抛异常。

总结：1、利用缓存识别已经遍历过的节点； 2、利用Java引用，先提前设置对象地址，后完善对象。

**Spring启动过程**

1. 读取web.xml文件。
2. 创建 ServletContext，为 ioc 容器提供宿主环境。
3. 触发容器初始化事件，调用 contextLoaderListener.contextInitialized()方法，在这个方法会初始化一个应用上下文WebApplicationContext，即 Spring 的 ioc 容器。ioc 容器初始化完成之后，会被存储到 ServletContext 中。
4. 初始化web.xml中配置的Servlet。如DispatcherServlet，用于匹配、处理每个servlet请求。

**Spring 的单例 Bean 是否有线程安全问题？**

当多个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，这时多个线程会并发执行该请求对应的业务逻辑，如果业务逻辑有对单例状态的修改（体现为此单例的成员属性），则必须考虑线程安全问题。

若每个线程中对全局变量、静态变量只有读操作，而无写操作，那么不会有线程安全问题；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则就可能影响线程安全。

**无状态bean和有状态bean**

* 有实例变量的bean，可以保存数据，是非线程安全的。
* 没有实例变量的对象。不能保存数据，是线程安全的。

在Spring中无状态的Bean适合用单例模式，这样可以共享实例提高性能。有状态的Bean在多线程环境下不安全，一般用Prototype模式或者使用ThreadLocal解决线程安全问题。

* [AOP](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#aop)
  + [静态代理](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#%E9%9D%99%E6%80%81%E4%BB%A3%E7%90%86)
  + [动态代理](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#%E5%8A%A8%E6%80%81%E4%BB%A3%E7%90%86)
  + [Spring AOP动态代理](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#spring-aop%E5%8A%A8%E6%80%81%E4%BB%A3%E7%90%86)
    - [实现原理](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#%E5%AE%9E%E7%8E%B0%E5%8E%9F%E7%90%86)
* [IOC](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#ioc)
  + [ioc容器](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#ioc%E5%AE%B9%E5%99%A8)
  + [BeanDefinition](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#beandefinition)
  + [容器初始化](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#%E5%AE%B9%E5%99%A8%E5%88%9D%E5%A7%8B%E5%8C%96)
  + [Bean生命周期](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#bean%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F)
  + [Aware接口](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#aware%E6%8E%A5%E5%8F%A3)
  + [BeanFactory和FactoryBean](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#beanfactory%E5%92%8Cfactorybean)
    - [FactoryBean使用](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#factorybean%E4%BD%BF%E7%94%A8)
  + [bean注入容器的方法](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#bean%E6%B3%A8%E5%85%A5%E5%AE%B9%E5%99%A8%E7%9A%84%E6%96%B9%E6%B3%95)
* [bean的作用域](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#bean%E7%9A%84%E4%BD%9C%E7%94%A8%E5%9F%9F)
* [事务](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#%E4%BA%8B%E5%8A%A1)
  + [@Transactional](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#transactional)
  + [事务传播行为](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#%E4%BA%8B%E5%8A%A1%E4%BC%A0%E6%92%AD%E8%A1%8C%E4%B8%BA)
* [循环依赖](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#%E5%BE%AA%E7%8E%AF%E4%BE%9D%E8%B5%96)
  + [初始化](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#%E5%88%9D%E5%A7%8B%E5%8C%96)
  + [三级缓存](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#%E4%B8%89%E7%BA%A7%E7%BC%93%E5%AD%98)
* [Spring启动过程](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#spring%E5%90%AF%E5%8A%A8%E8%BF%87%E7%A8%8B)
* [Spring Bean线程安全](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/Spring%E6%80%BB%E7%BB%93.md#spring-bean%E7%BA%BF%E7%A8%8B%E5%AE%89%E5%85%A8)

## AOP

面向切面编程，作为面向对象的一种补充，将公共逻辑（事务管理、日志、缓存等）封装成切面，跟业务代码进行分离，可以减少系统的重复代码和降低模块之间的耦合度。切面就是那些与业务无关，但所有业务模块都会调用的公共逻辑。

AOP有两种实现方式：静态代理和动态代理。

### 静态代理

静态代理：代理类在编译阶段生成，在编译阶段将通知织入Java字节码中，也称编译时增强。AspectJ使用的是静态代理。

缺点：代理对象需要与目标对象实现一样的接口，并且实现接口的方法，会有冗余代码。同时，一旦接口增加方法，目标对象与代理对象都要维护。

### 动态代理

动态代理：代理类在程序运行时创建，AOP框架不会去修改字节码，而是在内存中临时生成一个代理对象，在运行期间对业务方法进行增强，不会生成新类。

### Spring AOP动态代理

Spring AOP中的动态代理主要有两种方式：**JDK动态代理**和**CGLIB动态代理**。

1. JDK动态代理（生成的代理类实现了接口）。如果目标类实现了接口，Spring AOP会选择使用JDK动态代理目标类。代理类根据目标类实现的接口动态生成，不需要自己编写，生成的动态代理类和目标类都实现相同的接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。

缺点：目标类必须有实现的接口。如果某个类没有实现接口，那么这个类就不能用JDK动态代理。

1. CGLIB来动态代理（通过继承）。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）可以在运行时动态生成类的字节码，动态创建目标类的子类对象，在子类对象中增强目标类。

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。

优点：目标类不需要实现特定的接口，更加灵活。

什么时候采用哪种动态代理？

1. 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
2. 如果目标对象实现了接口，可以强制使用CGLIB实现AOP
3. 如果目标对象没有实现了接口，必须采用CGLIB库

区别：

1. jdk动态代理使用jdk中的类Proxy来创建代理对象，它使用反射技术来实现，不需要导入其他依赖。cglib需要引入相关依赖：asm.jar，它使用字节码增强技术来实现。
2. 当目标类实现了接口的时候Spring Aop默认使用jdk动态代理方式来增强方法，没有实现接口的时候使用cglib动态代理方式增强方法。

#### 实现原理

[Spring aop实现原理](https://www.zhihu.com/question/23641679)

Spring会为目标对象生成代理对象。当调用代理对象方法的时候，会触发CglibAopProxy.intercept()，然后将目标对象的增强包装成拦截器，形成拦截器链，最后执行全部拦截器和目标方法。

## IOC

[IOC基础](https://juejin.im/entry/588083111b69e60059034e3d#comment) | [IOC demo](https://juejin.im/post/5b399eb1e51d4553156c0525#heading-3)

IOC：控制反转，由Spring容器管理bean的整个生命周期。通过反射实现对其他对象的控制，包括初始化、创建、销毁等，解放手动创建对象的过程，同时降低类之间的耦合度。

IOC的好处：降低了类之间的耦合，对象创建和初始化交给Spring容器管理，在需要的时候只需向容器进行申请。

DI(依赖注入)：在Spring创建对象的过程中，把对象依赖的属性注入到对象中。有两种方式：构造器注入和属性注入。

### ioc容器

Spring主要有两种ioc容器，实现了BeanFactory接口的简单容器和ApplicationContext高级容器。

* BeanFactory ：延迟注入(使用到某个 bean 的时候才会注入),相比于BeanFactory 来说会占用更少的内存，程序启动速度更快。BeanFactory提供了最基本的ioc容器的功能（最基本的依赖注入支持）。
* ApplicationContext ：容器启动的时候，一次性创建所有 bean 。ApplicationContext 扩展了 BeanFactory ，除了有BeanFactory的功能还有额外更多功能，所以一般开发人员使用ApplicationContext会更多。

ApplicationContext 提供了 BeanFactory 没有的新特性：

1. 支持多语言版本；
2. 支持多种途径获取 Bean 定义信息；
3. 支持应用事件，方便管理 Bean；

DefaultListableBeanFactory 实现了 ioc 容器的基本功能，其他 ioc 容器如 XmlBeanFactory 和 ApplicationContext 都是通过持有或扩展 DefaultListableBeanFactory 获得基本的 ioc 容器的功能。

### BeanDefinition

BeanDefinition 用于管理Spring应用的对象和对象之间的依赖关系，是对象依赖关系的数据抽象。

### 容器初始化

ioc 容器初始化过程：BeanDefinition 的资源定位、解析和注册。

1. 从XML中读取配置文件。
2. 将bean标签解析成 BeanDefinition，如解析 property 元素， 并注入到 BeanDefinition 实例中。
3. 将 BeanDefinition 注册到容器 BeanDefinitionMap 中。
4. BeanFactory 根据 BeanDefinition 的定义信息创建实例化和初始化 bean。

单例bean的初始化以及依赖注入一般都在容器初始化阶段进行，只有懒加载（lazy-init为true）的单例bean是在应用第一次调用getBean()时进行初始化和依赖注入。

// AbstractApplicationContext

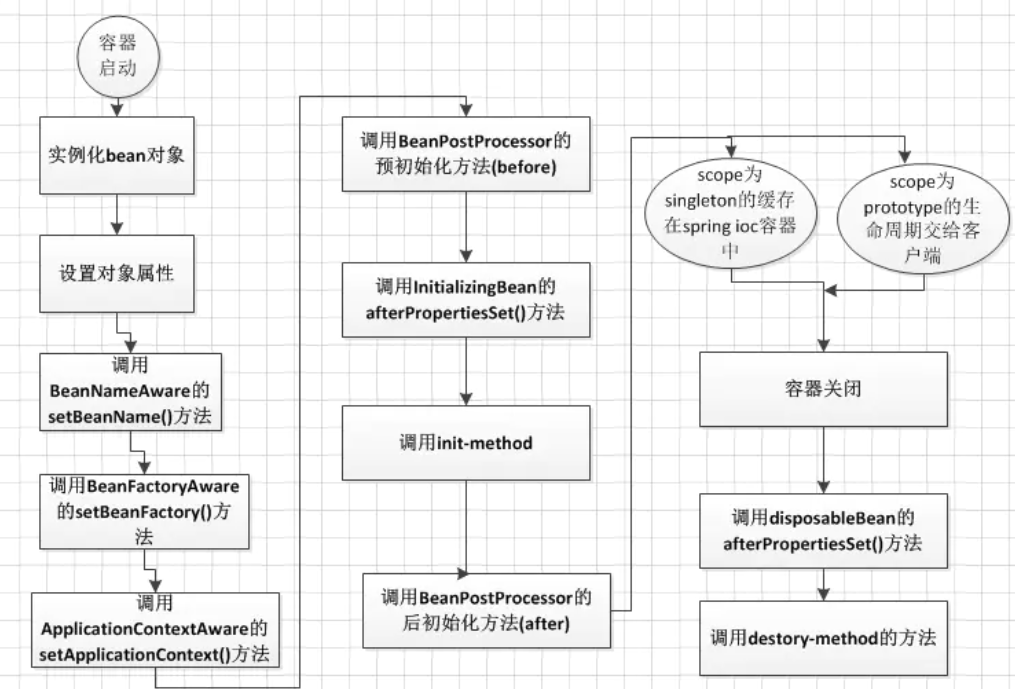
// Instantiate all remaining (non-lazy-init) singletons.

finishBeanFactoryInitialization(beanFactory);

多例bean 在容器启动时不实例化，即使设置 lazy-init 为 false 也没用，只有调用了getBean()才进行实例化。

loadBeanDefinitions 采用了模板模式，具体加载 BeanDefinition 的逻辑由各个子类完成。

### Bean生命周期



1.对Bean进行实例化

2.依赖注入

3.如果Bean实现了BeanNameAware接口，Spring将调用setBeanName()，设置 Bean id（xml文件中bean标签的id）

4.如果Bean实现了BeanFactoryAware接口，Spring将调用setBeanFactory()

5.如果Bean实现了ApplicationContextAware接口，Spring容器将调用setApplicationContext()

6.如果存在BeanPostProcessor，Spring将调用它们的postProcessBeforeInitialization（预初始化）方法，在Bean初始化前对其进行处理

7.如果Bean实现了InitializingBean接口，Spring将调用它的afterPropertiesSet方法，然后调用xml定义的 init-method 方法，两个方法作用类似，都是在初始化 bean 的时候执行

8.如果存在BeanPostProcessor，Spring将调用它们的postProcessAfterInitialization（后初始化）方法，在Bean初始化后对其进行处理

9.Bean初始化完成，供应用使用，直到应用被销毁

10.如果Bean实现了DisposableBean接口，Spring将调用它的destory方法，然后调用在xml中定义的 destory-method 方法，这两个方法作用类似，都是在Bean实例销毁前执行。

public interface BeanPostProcessor {

@Nullable

default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

return bean;

}

@Nullable

default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

return bean;

}

}

public interface InitializingBean {

void afterPropertiesSet() throws Exception;

}

### Aware接口

对于应用程序来说，应该尽量减少对Sping Api的耦合程度，然而有些时候为了运用Spring所提供的一些功能，有必要让Bean了解Spring容器对其进行管理的细节信息，如让Bean知道在容器中是以那个名称被管理的，或者让Bean知道BeanFactory或者ApplicationContext的存在，也就是让该Bean可以取得BeanFactory或者ApplicationContext的实例，如果Bean可以意识到这些对象，那么就可以在Bean的某些动作发生时，做一些如事件发布等操作。

BeanNameAware：通过调用setBeanName()可以让bean获取自身的id属性。

ApplicationContextAware：通过调用 setApplicationContext() 设置应用上下文实例，从而可以直接在 Bean 中使用应用上下文的服务。

ApplicationEventPublisherAware：通过调用 setApplicationEventPublisher() 给 Bean 设置事件发布器，从而可以在 Bean 中发布应用上下文的事件。

### BeanFactory和FactoryBean

**BeanFactory**：管理Bean的容器，Spring中生成的Bean都是由这个接口的实现来管理的。

**FactoryBean**：通常是用来创建比较复杂的bean，一般的bean 直接用xml配置即可，但如果一个bean的创建过程中涉及到很多其他的bean 和复杂的逻辑，直接用xml配置比较麻烦，这时可以考虑用FactoryBean，可以隐藏实例化复杂Bean的细节。

当配置文件中bean标签的class属性配置的实现类是FactoryBean时，通过 getBean()方法返回的不是FactoryBean本身，而是调用FactoryBean#getObject()方法所返回的对象，相当于FactoryBean#getObject()代理了getBean()方法。如果想得到FactoryBean必须使用 '&' + beanName 的方式获取。

Mybatis 提供了 SqlSessionFactoryBean，可以简化 SqlSessionFactory 的配置：

public class SqlSessionFactoryBean implements FactoryBean<SqlSessionFactory>, InitializingBean, ApplicationListener<ApplicationEvent> {

@Override

public void afterPropertiesSet() throws Exception {

notNull(dataSource, "Property 'dataSource' is required");

notNull(sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is required");

state((configuration == null && configLocation == null) || !(configuration != null && configLocation != null),

"Property 'configuration' and 'configLocation' can not specified with together");

this.sqlSessionFactory = buildSqlSessionFactory();

}

protected SqlSessionFactory buildSqlSessionFactory() throws IOException {

//复杂逻辑

}

@Override

public SqlSessionFactory getObject() throws Exception {

if (this.sqlSessionFactory == null) {

afterPropertiesSet();

}

return this.sqlSessionFactory;

}

}

在 xml 配置 SqlSessionFactoryBean：

<bean id="tradeSqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="trade" />

<property name="mapperLocations" value="classpath\*:mapper/trade/\*Mapper.xml" />

<property name="configLocation" value="classpath:mybatis-config.xml" />

<property name="typeAliasesPackage" value="com.bytebeats.mybatis3.domain.trade" />

</bean>

Spring 将会在应用启动时创建 SqlSessionFactory，并使用 sqlSessionFactory 这个名字存储起来。

#### FactoryBean使用

[FactoryBean使用](https://www.cnblogs.com/davidwang456/p/3688250.html)

如果使用传统方式配置下面Car的<bean>时，Car的每个属性分别对应一个<property>元素标签。

public class Car {

private int maxSpeed ;

private String brand ;

private double price ;

}

如果用FactoryBean的方式实现就会灵活一些，下例通过逗号分割符的方式一次性地为Car的所有属性指定配置值：

public class CarFactoryBean implements FactoryBean<Car> {

private String carInfo ;

public Car getObject () throws Exception {

Car car = new Car () ;

String [] infos = carInfo .split ( "," ) ;

car.setBrand ( infos [ 0 ]) ;

car.setMaxSpeed ( Integer. valueOf ( infos [ 1 ])) ;

car.setPrice ( Double. valueOf ( infos [ 2 ])) ;

return car;

}

public Class<Car> getObjectType () {

return Car. class ;

}

public boolean isSingleton () {

return false ;

}

public String getCarInfo () {

return this . carInfo ;

}

// 接受逗号分割符设置属性信息

public void setCarInfo ( String carInfo ) {

this . carInfocarInfo = carInfo;

}

}

xml 配置 CarFactoryBean：

<bean id="car" class="com.test.factorybean.CarFactoryBean" carInfo="超级跑车,400,2000000"/>

当调用getBean("car") 时，Spring通过反射机制发现CarFactoryBean实现了FactoryBean的接口，这时Spring容器就调用接口方法CarFactoryBean#getObject()方法返回。如果希望获取CarFactoryBean的实例，则需要在使getBean(beanName) 方法时在beanName前显示的加上 "&" 前缀，例如getBean("&car")。

### bean注入容器的方法

将普通类交给Spring容器管理，通常有以下方法：

1、使用 @Configuration与@Bean 注解

2、使用@Controller @Service @Repository @Component 注解标注该类，然后启用@ComponentScan自动扫描

3、使用@Import 方法

@Import注解把bean导入到当前容器中。

//@SpringBootApplication

@ComponentScan

/\*把用到的资源导入到当前容器中\*/

@Import({Dog.class, Cat.class})

public class App {

public static void main(String[] args) throws Exception {

ConfigurableApplicationContext context = SpringApplication.run(App.class, args);

System.out.println(context.getBean(Dog.class));

System.out.println(context.getBean(Cat.class));

context.close();

}

}

## bean的作用域

Spring创建bean默认是单例，每一个Bean的实例只会被创建一次，通过getBean()获取的是同一个Bean的实例。可使用＜bean＞标签的scope属性来指定一个Bean的作用域。

<!-- 作用域：prototype -->

<bean name="accountDao" scope="prototype" class="com.zejian.spring.springIoc.dao.impl.AccountDaoImpl"/>

通过注解来声明作用域：

@Scope("prototype")

public class AccountDaoImpl {

//......

}

容器在创建完一个prototype实例后，就不会去管理这个bean了，会把它交给应用自己去管理。

一般情况下，对有状态的bean应该使用prototype作用域，而对无状态的bean则应该使用singleton作用域。所谓有状态就是该bean保存有自己的信息，不能共享，否则会造成线程安全问题。而无状态则不保存信息，可以共享，spring中大部分bean都是单例的，整个生命周期过程只会存在一个。

request作用域：对于每次HTTP请求到达应用程序，Spring容器会创建一个全新的Request作用域的bean实例，且该bean实例仅在当前HTTP request内有效，整个请求过程也只会使用相同的bean实例，而其他请求HTTP请求则创建新bean的实例，当处理请求结束，request作用域的bean实例将被销毁。

session 作用域：每当创建一个新的HTTP Session时就会创建一个Session作用域的Bean，并该实例bean伴随着会话的结束（session过期）而销毁。

## 事务

事务就是一系列的操作原子执行。

Spring事务机制主要包括声明式事务和编程式事务。

声明式事务将事务管理代码从业务方法中分离出来，通过aop进行封装。用 @Transactional 注解开启声明式事务。

Spring声明式事务使得我们无需要去处理获得连接、关闭连接、事务提交和回滚等这些操作。

### @Transactional

| **属性** | **类型** | **描述** |
| --- | --- | --- |
| value | String | 可选的限定描述符，指定使用的事务管理器 |
| propagation | enum: Propagation | 可选的事务传播行为设置 |
| isolation | enum: Isolation | 可选的事务隔离级别设置 |
| readOnly | boolean | 读写或只读事务，默认读写 |
| timeout | int (in seconds granularity) | 事务超时时间设置 |
| rollbackFor | Class对象数组，必须继承自Throwable | 导致事务回滚的异常类数组 |
| rollbackForClassName | 类名数组，必须继承自Throwable | 导致事务回滚的异常类名字数组 |
| noRollbackFor | Class对象数组，必须继承自Throwable | 不会导致事务回滚的异常类数组 |
| noRollbackForClassName | 类名数组，必须继承自Throwable | 不会导致事务回滚的异常类名字数组 |

### 事务传播行为

在TransactionDefinition接口中定义了七个事务传播行为：

1. PROPAGATION\_REQUIRED 如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。如果嵌套调用的两个方法都加了事务注解，并且运行在相同线程中，则这两个方法使用相同的事务中。如果运行在不同线程中，则会开启新的事务。
2. PROPAGATION\_SUPPORTS 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行。
3. PROPAGATION\_MANDATORY 如果已经存在一个事务，支持当前事务。如果不存在事务，则抛出异常IllegalTransactionStateException。
4. PROPAGATION\_REQUIRES\_NEW 总是开启一个新的事务。需要使用JtaTransactionManager作为事务管理器。
5. PROPAGATION\_NOT\_SUPPORTED 总是非事务地执行，并挂起任何存在的事务。需要使用JtaTransactionManager作为事务管理器。
6. PROPAGATION\_NEVER 总是非事务地执行，如果存在一个活动事务，则抛出异常。
7. PROPAGATION\_NESTED 如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务, 则按PROPAGATION\_REQUIRED 属性执行。

**PROPAGATION\_NESTED 与PROPAGATION\_REQUIRES\_NEW的区别:**

使用PROPAGATION\_REQUIRES\_NEW时，内层事务与外层事务是两个独立的事务。一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。

使用PROPAGATION\_NESTED时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。

## 循环依赖

[图解spring循环依赖](https://juejin.im/post/5e927e27f265da47c8012ed9#heading-5) | [循环依赖](https://blog.csdn.net/u010853261/article/details/77940767)

构造器注入的循环依赖：spring处理不了，直接抛出BeanCurrentlylnCreationException异常。

单例模式下属性注入的循环依赖：通过三级缓存处理循环依赖。

非单例循环依赖：无法处理。

### 初始化

spring单例对象的初始化大略分为三步：

1. createBeanInstance：实例化bean，使用构造方法创建对象，为对象分配内存。
2. populateBean：进行依赖注入。
3. initializeBean：初始化bean。

this.singletonsCurrentlyInCreation.add(String beanName)将当前正在创建的bean id记录在缓存中，如果在记录的过程中发现自己已经在缓存中，则说明存在循环依赖，将抛出BeanCurrentlylnCreationException 异常表示循环依赖。创建完成的bean将会从缓存中清除。

### 三级缓存

Spring为了解决单例的循环依赖问题，使用了三级缓存。

singletonObjects：完成了初始化的单例对象map，bean name --> bean instance

earlySingletonObjects ：完成实例化未初始化的单例对象map，bean name --> bean instance

singletonFactories ： 单例对象工厂map，bean name --> ObjectFactory，单例对象实例化完成之后会加入singletonFactories。

在调用createBeanInstance进行实例化之后，会调用addSingletonFactory，将单例对象放到singletonFactories中。

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);

}

}

}

假如A依赖了B的实例对象，同时B也依赖A的实例对象。

1. A首先完成了实例化，并且将自己添加到singletonFactories中
2. 接着进行依赖注入，发现自己依赖对象B，此时就尝试去get(B)
3. 发现B还没有被实例化，对B进行实例化
4. 然后B在初始化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和二级缓存earlySingletonObjects没找到，尝试三级缓存singletonFactories，由于A初始化时将自己添加到了singletonFactories，所以B可以拿到A对象，然后将A从三级缓存中移到二级缓存中
5. B拿到A对象后顺利完成了初始化，然后将自己放入到一级缓存singletonObjects中
6. 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

由此看出，属性注入的循环依赖主要是通过将实例化完成的bean添加到singletonFactories来实现的。而使用构造器依赖注入的bean在实例化的时候会进行依赖注入，不会被添加到singletonFactories中。比如A和B都是通过构造器依赖注入，A在调用构造器进行实例化的时候，发现自己依赖B，B没有被实例化，就会对B进行实例化，此时A未实例化完成，不会被添加到singtonFactories。而B依赖于A，B会去三级缓存寻找A对象，发现不存在，于是又会实例化A，A实例化了两次，从而导致抛异常。

总结：1、利用缓存识别已经遍历过的节点； 2、利用Java引用，先提前设置对象地址，后完善对象。

## Spring启动过程

1. 读取web.xml文件。
2. 创建 ServletContext，为 ioc 容器提供宿主环境。
3. 触发容器初始化事件，调用 contextLoaderListener.contextInitialized()方法，在这个方法会初始化一个应用上下文WebApplicationContext，即 Spring 的 ioc 容器。ioc 容器初始化完成之后，会被存储到 ServletContext 中。
4. public void contextInitialized(ServletContextEvent event) {
5. initWebApplicationContext(event.getServletContext());

}

1. 初始化web.xml中配置的Servlet。如DispatcherServlet，用于匹配、处理每个servlet请求。

## Spring Bean线程安全

Spring Bean默认是单例的，大部分的Spring bean没有可变的状态（比如Service类和DAO类），是线程安全的。如果Bean带有状态，可以将bean设置为prototype或者使用ThreadLocal确保线程安全。

* [Spring Boot基础](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#spring-boot%E5%9F%BA%E7%A1%80)
  + [特点](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E7%89%B9%E7%82%B9)
* [Spring Boot核心](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#spring-boot%E6%A0%B8%E5%BF%83)
  + [基本配置](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E5%9F%BA%E6%9C%AC%E9%85%8D%E7%BD%AE)
  + [外部配置](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E5%A4%96%E9%83%A8%E9%85%8D%E7%BD%AE)
  + [日志配置](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E6%97%A5%E5%BF%97%E9%85%8D%E7%BD%AE)
  + [Profile配置](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#profile%E9%85%8D%E7%BD%AE)
* [Spring Boot的Web开发](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#spring-boot%E7%9A%84web%E5%BC%80%E5%8F%91)
  + [Thymeleaf 模板引擎](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#thymeleaf-%E6%A8%A1%E6%9D%BF%E5%BC%95%E6%93%8E)
    - [Thymeleaf 基础知识](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#thymeleaf-%E5%9F%BA%E7%A1%80%E7%9F%A5%E8%AF%86)
    - [与 Spring MVC 集成](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E4%B8%8E-spring-mvc-%E9%9B%86%E6%88%90)
    - [Spring Boot 的 Thymeleaf 支持](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#spring-boot-%E7%9A%84-thymeleaf-%E6%94%AF%E6%8C%81)
    - [实战](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E5%AE%9E%E6%88%98)
  + [Web 相关配置](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#web-%E7%9B%B8%E5%85%B3%E9%85%8D%E7%BD%AE)
    - [Spring Boot 提供的自动配置](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#spring-boot-%E6%8F%90%E4%BE%9B%E7%9A%84%E8%87%AA%E5%8A%A8%E9%85%8D%E7%BD%AE)
    - [实现自己的 MVC 配置](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E5%AE%9E%E7%8E%B0%E8%87%AA%E5%B7%B1%E7%9A%84-mvc-%E9%85%8D%E7%BD%AE)
    - [注册 Servlet、Filter、Listener](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E6%B3%A8%E5%86%8C-servletfilterlistener)
  + [Tomcat配置](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#tomcat%E9%85%8D%E7%BD%AE)
    - [配置 Tomcat](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E9%85%8D%E7%BD%AE-tomcat)
    - [替换 Tomcat](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E6%9B%BF%E6%8D%A2-tomcat)
* [Spring Boot 的数据访问](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#spring-boot-%E7%9A%84%E6%95%B0%E6%8D%AE%E8%AE%BF%E9%97%AE)
  + [Docker 常用命令及参数](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#docker-%E5%B8%B8%E7%94%A8%E5%91%BD%E4%BB%A4%E5%8F%8A%E5%8F%82%E6%95%B0)
    - [Docker 镜像命令](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#docker-%E9%95%9C%E5%83%8F%E5%91%BD%E4%BB%A4)
    - [Docker 容器命令](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#docker-%E5%AE%B9%E5%99%A8%E5%91%BD%E4%BB%A4)
  + [Spring Boot 对 Spring Data JPA 的支持](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#spring-boot-%E5%AF%B9-spring-data-jpa-%E7%9A%84%E6%94%AF%E6%8C%81)
    - [JDBC 的自动配置](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#jdbc-%E7%9A%84%E8%87%AA%E5%8A%A8%E9%85%8D%E7%BD%AE)
    - [对 JPA 的自动配置](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E5%AF%B9-jpa-%E7%9A%84%E8%87%AA%E5%8A%A8%E9%85%8D%E7%BD%AE)
    - [对 Spring Data JPA 的自动配置](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E5%AF%B9-spring-data-jpa-%E7%9A%84%E8%87%AA%E5%8A%A8%E9%85%8D%E7%BD%AE)
* [@Value原理](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#value%E5%8E%9F%E7%90%86)
* [启动过程](https://github.com/Tyson0314/Java-learning/blob/master/%E6%A1%86%E6%9E%B6/SpringBoot%E5%AE%9E%E6%88%98.md#%E5%90%AF%E5%8A%A8%E8%BF%87%E7%A8%8B)

## Spring Boot基础

理念：习惯优于配置，内置习惯性配置，无需手动进行配置。使用Spring boot可以很快创建一个独立运行、准生产级别的基于Spring框架的项目，不需要或者只需很少的Spring配置。

### 特点

* 内置servlet容器，不需要在服务器部署 tomcat。只需要将项目打成 jar 包，使用 java -jar xxx.jar一键式启动项目；
* SpringBoot提供了starter，把常用库聚合在一起，简化复杂的环境配置，快速搭建spring应用环境。

## Spring Boot核心

### 基本配置

Spring Boot通常有个Application入口类：

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@SpringBootApplication

public class SpringbootDemoApplication {

public static void main(String[] args) {

SpringApplication.run(SpringbootDemoApplication.class, args);

}

}

@SpringBootApplication是Spring Boot的核心注解，它是组合注解，源码如下：

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan(

excludeFilters = {@Filter(

type = FilterType.CUSTOM,

classes = {TypeExcludeFilter.class}

), @Filter(

type = FilterType.CUSTOM,

classes = {AutoConfigurationExcludeFilter.class}

)}

)

public @interface SpringBootApplication {

}

@SpringBootApplication注解组合了@Configuration、@EnableAutoConfiguration和@ComponentScan注解，若不使用@SpringBootApplication注解，则可以在入口类上直接使用@Configuration、@EnableAutoConfiguration和@ComponentScan。

@EnableAutoConfiguration作用是让Spring Boot根据类路径中的jar包依赖为当前项目进行自动配置。例如，添加了spring-boot-starter-web依赖，会自动添加Tomcat和Spring MVC依赖，那么Spring Boot会对Tomcat和Spring MVC进行自动配置。

@Configuration标注在类上，相当于把该类作为spring的xml配置文件中的<beans>，作用是配置spring容器。

**关闭特定的自动配置**

使用@SpringBootApplication的exclude参数关闭特定的自动配置。

@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})

**Spring Boot的配置文件**

Spring Boot使用一个全局的配置文件application.properties或application.yml。这个全局配置文件可以对一些默认配置的配置值进行修改。如修改Tomcat默认的端口号，并将默认的访问路径"/"修改为"/hello"：

server.port=9090

server.context-path=/hello

**starter pom**

Spring Boot为我们提供了简化企业级开发绝大多数场景的starter pom，只要使用了应用场景所需要的starter pom，相应的配置就可以消除，就可以得到Spring Boot为我们提供的自动配置的bean。

**xml配置**

Spring Boot提倡零配置，但实际项目中可能需要使用xml配置，此时可以通过Spring提供的@ImportResource来加载xml配置。

@ImportResource({"classpath:xxx-context.xml", "classpath:yyy-context.xml"})

### 外部配置

**命令行参数配置**

Spring Boot是基于jar包运行的，打成jar包的程序可以直接通过下面的命令运行：

java -jar xx.jar

可以通过以下命令修改端口号：

java -jar xx.jar --server.port=9090

**常规属性配置**

在Spring Boot里，我们只需在application.properties定义属性，直接使用@Value注入即可。

application.properties增加属性：

book.author=tyson

book.name=life

修改入口类：

import org.springframework.beans.factory.annotation.Value;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@SpringBootApplication

public class SpringbootDemoApplication {

@Value("${book.author}")

private String bookAuthor;

@Value("${book.name}")

private String bookName;

@RequestMapping("/")

public String index() {

return "book name: " + bookName + ", written by: " + bookAuthor;

}

public static void main(String[] args) {

SpringApplication.run(SpringbootDemoApplication.class, args);

}

}

@RestController=@Controller + @ResponseBody，注解的类里面的方法以json格式输出。

**类型安全的配置**

Spring Boot提供了基于类型安全的配置方式，通过@ConfigurationProperties将配置文件application.properties中配置的属性值映射到当前类的属性中，从而实现类型安全的配置。

类型安全的bean：

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.stereotype.Component;

@ConfigurationProperties(prefix = "book")

public class BookConfig {

private String name;

private String author;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getAuthor() {

return author;

}

public void setAuthor(String author) {

this.author = author;

}

}

通过@ConfigurationProperties加载properties文件内的配置，通过prefix属性指定前缀。

检验代码：

import com.tyson.springbootdemo.config.BookConfig;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@SpringBootApplication

@EnableConfigurationProperties({BookConfig.class})

public class SpringbootDemoApplication {

@Autowired

public BookConfig bookConfig;

@RequestMapping("/")

public String index() {

return "book name: " + bookConfig.getName() + ", written by: " + bookConfig.getAuthor();

}

public static void main(String[] args) {

SpringApplication.run(SpringbootDemoApplication.class, args);

}

}

@EnableConfigurationProperties注解将带有@ConfigurationProperties注解的类注入为Spring容器的Bean。

### 日志配置

Spring Boot 支持 Log4J、Logback、Java Util Logging、Log4J2 作为日志框架，无论使用哪种日志框架，Spring Boot 已为当前使用日志框架的控制台输出及文件输出做好了配置。默认情况下，Spring Boot 使用 Logback 作为日志框架，日志级别为 INFO。

配置日志文件：

logging.path=H:/log/

logging.file=springbootdemo.log

配置日志级别，格式为 logging.level.包名=级别：

logging.level.root=INFO #root级别，项目所有日志

logging.level.org.springframework.web=DEBUG #package级别

配置日志样式：

logging.pattern.console=%d{yyyy/MM/dd-HH:mm:ss} [%thread] %-5level %logger- %msg%n

logging.pattern.file=%d{yyyy/MM/dd-HH:mm} [%thread] %-5level %logger- %msg%n

### Profile配置

Profile 是 Spring 用来针对不同环境对不同配置提供支持的，全局Profile配置使用 application-{profile}.properties（如application-prod.properties）。通过在 application.properties 中设置spring.profiles.active=prod 来制定活动的Profile。

假如有生产和开发环境，生产环境下端口号为80，开发环境下端口号为8888。配置文件如下：

application-prod.properties:

server.port=80

application-dev.properties

server.port=8888

application.properties 增加：

spring.profiles.active=prod

启动程序结果为：

2019-03-03 09:17:08.003 INFO 17812 --- [ main] c.t.s.SpringbootDemoApplication : The following profiles are active: prod

2019-03-03 09:17:11.007 INFO 17812 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8888 (http)

## Spring Boot的Web开发

spring-boot-starter-web 为我们提供了嵌入的 tomcat 和 Spring MVC 的依赖。

### Thymeleaf 模板引擎

JSP 在内嵌的 Servlet 容器上运行会存在一些问题（内嵌 Tomcat、Jetty 不支持以 jar 形式运行 JSP，Undertow 不支持 JSP）。Spring Boot 提供了大量的模板引擎，包含 FreeMarker、Groovy、Thymeleaf、Velocity 和 Mustache，Spring Boot 中推荐使用 Thymeleaf 作为模板引擎，因为 Thymeleaf 提供了完美的 Spring MVC 的支持。

#### Thymeleaf 基础知识

Thymeleaf 是 Java 类库，它是一个 xml/xhtml/html5 的模板引擎，可以作为 MVC 的 Web 应用的 view 层。Thymeleaf 还提供了额外的模块与 Spring MVC 集成，使用 Thymeleaf 完全可以替代 JSP。

1. **引入 Thymeleaf**

基本的 Thymeleaf 模板页面，引入了 Bootstrap（作为样式控制）和 jQuery（DOM 操作）。

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

<link th:src="@{bootstrap/css/bootstrap.min.css}" rel="stylesheet">

<link th:src="@{bootstrap/css/bootstrap-theme.min.css}" rel="stylesheet">

<title>Demo</title>

</head>

<body>

<script th:src="@{jquery-1.10.2.min.js}" type="text/javascript"></script>

<script th:src="@{bootstrap/js/bootstrap.min.js}"></script>

</body>

</html>

通过 xmlns:th="[http://www.thymeleaf.org](http://www.thymeleaf.org/)" 命名空间，将静态页面转换为动态视图。需要进行动态处理的元素需使用 “th:” 为前缀。通过 “@{}” 引用 Web 静态资源，这在 JSP 下是极易出错的。

1. **访问 model 中的数据**

通过 “${}” 访问 model 中的属性，这和 JSP 很相似。需要处理的动态内容需要加上 “th:” 前缀。

<div class="panel panel-primary">

<div class="panel-heading">

<h3 class="panel-title">访问model</h3>

</div>

<div class="panel-body">

<span th:text="${person.name}"></span>

</div>

</div>

1. **model 中的数据迭代**

<div class="panel panel-primary">

<div class="panel-heading">

<h3 class="panel-title">列表</h3>

</div>

<div class="panel-body">

<ul class="list-group">

<li class="list-group-item" th:each="person:${people}">

<span th:text="${person.name}"></span>

<span th:text="${person.age}"></span>

</li>

</ul>

</div>

</div>

1. **数据判断**

通过${not #lists.isEmpty(people)}表达式判断 people 是否为空。Thymeleaf 还支持 >、<、==、!= 等作为比较条件，同时也支持将 SpringEL 表达式语言用于条件中。

<div th:if="${not #lists.isEmpty(people)}">

<div class="panel panel-primary">

<div class="panel-heading">

<h3 class="panel-title">列表</h3>

</div>

<div class="panel-body">

<ul class="list-group">

<li class="list-group-item" th:each="person:${people}">

<span th:text="${person.name}"></span>

<span th:text="${person.age}"></span>

</li>

</ul>

</div>

</div>

</div>

1. 在 JavaScript 中访问 model

<script th:inline="javascript">

var single = [[${person}]];

console.log(single.name+"/"+single.age)

</script>

通过 th:inline="javascript"添加到 script 标签，这样 JavaScript 代码即可访问 model 中的属性；

通过“[[${person}]]”格式可以获得实际的值。

如果需要在 html 代码里访问 model 中的属性，比如我们需要在列表后面单击每一行后面的按钮获得 model 中的值，可做如下处理：

<li class="list-group-item" th:each="person:${people}">

<span th:text="${person.name}"></span>

<span th:text="${person.age}"></span>

<button class="btn"

th:onclick="'getName(\'' + ${person.name} + '\');'">获得名字</button>

</li>

#### 与 Spring MVC 集成

在 Spring MVC 中，若我们需要集成一个模板引擎的话，需要定义 ViewResolver，而 ViewResolver 需要定义一个 View。在 Spring MVC 中集成 Thymeleaf 非常简单，Thymeleaf 为我们定义好了 org.thymeleaf.spring4.view .ThymeleafView 和 org.thymeleaf.spring4.view.ThymeleafViewResolver（默认使用 ThymeleafView 作为 View）。Thymeleaf 给我们提供了一个 SpringTemplateEngine 类，用来驱动 Spring MVC 下使用 Thymeleaf 模板引擎，另外提供了一个 TemplateResolver 用来设置通用的模板引擎（包含前缀、后缀等）。

引入依赖：

<!--thymeleaf-->

<dependency>

<groupId>org.thymeleaf</groupId>

<artifactId>thymeleaf-spring4</artifactId>

<version>${thymeleaf.version}</version>

</dependency>

<dependency>

<groupId>org.thymeleaf</groupId>

<artifactId>thymeleaf</artifactId>

<version>${thymeleaf.version}</version>

</dependency>

xml 配置：

<!--thymeleaf-->

<bean id="templateResolver" class="org.thymeleaf.templateresolver.ServletContextTemplateResolver">

<property name="prefix" value="/WEB-INF/templates/" />

<property name="suffix" value=".html" />

<property name="templateMode" value="HTML5" />

<property name="cacheable" value="false" />

<property name="characterEncoding" value="UTF-8"/>

</bean>

<bean id="templateEngine" class="org.thymeleaf.spring4.SpringTemplateEngine">

<property name="templateResolver" ref="templateResolver"/>

</bean>

<bean class="org.thymeleaf.spring4.view.ThymeleafViewResolver">

<property name="templateEngine" ref="templateEngine"/>

<property name="characterEncoding" value="UTF-8" />

</bean>

或者使用 JavaConfig 配置：

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.thymeleaf.spring4.SpringTemplateEngine;

import org.thymeleaf.spring4.view.ThymeleafViewResolver;

import org.thymeleaf.templateresolver.ServletContextTemplateResolver;

import org.thymeleaf.templateresolver.TemplateResolver;

@Configuration

public class ThymeleafConfig {

@Bean

public TemplateResolver templateResolver() {

TemplateResolver templateResolver = new ServletContextTemplateResolver();

templateResolver.setPrefix("/WEB-INF/template");

templateResolver.setSuffix(".html");

templateResolver.setTemplateMode("HTML5");

return templateResolver;

}

@Bean

public SpringTemplateEngine templateEngine(TemplateResolver templateResolver) {

SpringTemplateEngine templateEngine = new SpringTemplateEngine();

templateEngine.setTemplateResolver(templateResolver);

return templateEngine;

}

@Bean

public ThymeleafViewResolver thymeleafViewResolver(SpringTemplateEngine templateEngine) {

ThymeleafViewResolver thymeleafViewResolver = new ThymeleafViewResolver();

thymeleafViewResolver.setTemplateEngine(templateEngine);

//thymeleafViewResolver.setViewClass(ThymeleafView.class);

return thymeleafViewResolver;

}

}

#### Spring Boot 的 Thymeleaf 支持

Spring Boot 通过 org.springframework.boot.autoconfigure.thymeleaf 包对 Thymeleaf 进行了自动配置，通过ThymeleafAutoConfiguration 类对集成所需的 Bean 进行自动配置，包括 templateResolver、templateEngine 和 thymeleafViewResolvers 的配置。通过 ThymeleafProperties 来设置属性以及默认配置。

@ConfigurationProperties(

prefix = "spring.thymeleaf"

)

public class ThymeleafProperties {

private static final Charset DEFAULT\_ENCODING;

public static final String DEFAULT\_PREFIX = "classpath:/templates/";

public static final String DEFAULT\_SUFFIX = ".html";

private boolean checkTemplate = true;

private boolean checkTemplateLocation = true;

private String prefix = "classpath:/templates/";

private String suffix = ".html";

private String mode = "HTML";

private Charset encoding;

private boolean cache;

private Integer templateResolverOrder;

private String[] viewNames;

private String[] excludedViewNames;

private boolean enableSpringElCompiler;

private boolean renderHiddenMarkersBeforeCheckboxes;

private boolean enabled;

private final ThymeleafProperties.Servlet servlet;

private final ThymeleafProperties.Reactive reactive;

...

}

#### 实战

1. **引入依赖**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

引入 Thymeleaf 之后，Controller才能根据逻辑视图名转发到相应的视图。

1. **JavaBean**

public class Person {

private String name;

private Integer age;

private String address;

public Person() {}

//setter和getter

}

1. **页面**

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

<meta name="viewport" content="width=device-width, initial-scale=1"/>

<link th:href="@{bootstrap/css/bootstrap.min.css}" rel="stylesheet">

<link th:href="@{bootstrap/css/bootstrap-theme.min.css}" rel="stylesheet"/>

<title>index</title>

</head>

<body>

<div class="panel panel-primary">

<div class="panel-heading">

<h3 class="panel-title">welcome</h3>

</div>

<div class="panel-body">

<span th:text="${person.name}"/>

</div>

</div>

</body>

</html>

页面引入了Bootstrap和jQuery，这些静态文件放置在src/main/resources/static下面。

1. **Controller**

import com.tyson.springbootdemo.pojo.Person;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.RequestMapping;

**@Controller**

**@RequestMapping("/person")**

public class PersonController {

@RequestMapping("/hi")

public String index(Model model) {

Person p = new Person();

p.setName("tyson");

model.addAttribute("person", p);

return "index";

}

@RequestMapping("/")

public String home() {

return "home";

}

}

### Web 相关配置

#### Spring Boot 提供的自动配置

WebMvcAutoConfiguration 及 WebMvcProperties 定义了 Web 相关的自动配置。

1. **自动配置的ViewResolver**

@Bean

@ConditionalOnMissingBean

public InternalResourceViewResolver defaultViewResolver() {

InternalResourceViewResolver resolver = new InternalResourceViewResolver();

resolver.setPrefix(this.mvcProperties.getView().getPrefix());

resolver.setSuffix(this.mvcProperties.getView().getSuffix());

return resolver;

}

@Bean

@ConditionalOnBean({View.class})

@ConditionalOnMissingBean

public BeanNameViewResolver beanNameViewResolver() {

BeanNameViewResolver resolver = new BeanNameViewResolver();

resolver.setOrder(2147483637);

return resolver;

}

1. **自动配置的静态资源**

将类路径下的/static、/resources、/public 和 /META-INF/resources 文件夹下的静态文件直接映射为/\*\*，可以通过 [http://localhost:8080/\\*\\*访问。](http://localhost:8080/%5C*%5C*%E8%AE%BF%E9%97%AE%E3%80%82)

把 webjar 的 /META-INF/resources/webjars/ 下的静态文件映射为/webjar/\*\*，可以通过 [http://localhost:8080/webjar/\\*\\*访问。](http://localhost:8080/webjar/%5C*%5C*%E8%AE%BF%E9%97%AE%E3%80%82)

1. **自动配置的 Formatter 和 Converter**

public void addFormatters(FormatterRegistry registry) {

Iterator var2 = this.getBeansOfType(Converter.class).iterator();

while(var2.hasNext()) {

Converter<?, ?> converter = (Converter)var2.next();

registry.addConverter(converter);

}

var2 = this.getBeansOfType(GenericConverter.class).iterator();

while(var2.hasNext()) {

GenericConverter converter = (GenericConverter)var2.next();

registry.addConverter(converter);

}

var2 = this.getBeansOfType(Formatter.class).iterator();

while(var2.hasNext()) {

Formatter<?> formatter = (Formatter)var2.next();

registry.addFormatter(formatter);

}

}

只要我们定义了 Converter、GenericConverter 和 Formatter 接口的实现类的 Bean，这些 Bean 就会自动注册到 Spring MVC 中。

1. **静态首页的支持**

**把静态首页放到如下目录：**

* **classpath:/META-INF/resources/index.html**
* classpath:/resources/index.html
* classpath:/static/index.html
* classpath:/public/index.html

当我们访问应用根目录 <http://localhost:8080/> 时，会直接映射。

#### 实现自己的 MVC 配置

当 Spring Boot 提供的 Spring MVC 不符合要求时，可以通过一个配置类（注解有@Configuration 的类）加上@EnableWebMvc 注解来实现完全自己控制的 MVC 配置。

要想保留 Spring Boot 提供的 MVC 配置，同时增加额外的配置，可以通过定义一个配置类并继承 WebMvcConfigurerAdapter，无需使用@EnableWebMvc注解。

import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

public class WebMvcConfig implements WebMvcConfigurer {

@Override

public void addViewControllers(ViewControllerRegistry registry) {

registry.addViewController("/xx").setViewName("xx");

}

}

重写的 addViewControllers 方法并不会覆盖 WebMvcAutocConfiguration 中的 addViewControllers（此方法中，Spring Boot 将“/”映射到 index.html），即我们自己的配置和 Spring Boot 的自动配置同时生效。

#### 注册 Servlet、Filter、Listener

当使用嵌入式的 Servlet 容器时，通过将 Servlet、Filter 和 Listener 声明为 Spring Bean 达到注册的效果；或者注册 ServletRegistrationBean、FilterRegistrationBean 和 ServletRegistrationBean 的 Bean。

直接注册 bean：

@Bean

public xxServlet xxServlet() {

return new XxServlet();

}

通过 RegistrationBean 示例：

@Bean

public ServletRegistrationBean servletRegistrationBean() {

return new ServletRegistrationBean(new XxServlet(), "/xx/\*");

}

@Bean

public FilterRegistrationBean filterRegistrationBean() {

FilterRegistrationBean registrationBean = new FilterRegistrationBean();

registrationBean.setFilter(new YyFilter());

registrationBean.setOrder(2);

return registrtionBean;

}

@Bean

public ServletListenerRegistrationBean<ZzListener> zzListenerServletRegistrationBean() {

return new ServletListenerRegistrationBean<ZzListener>(new ZzListener());

}

### Tomcat配置

#### 配置 Tomcat

关于 Tomcat 的所有属性都在 org.springframework.boot.autoconfigure.web.ServerPr=8080operties 配置类中做了定义，只需在 application.properties 配置即可。通用的 Servlet 容器配置都以 "server" 作为前缀，而 Tomcat 特有的配置都以 "server.tomcat" 作为前缀。

#配置Servlet容器

server.port=8080

server.session-timeout=3600 #以秒为单位

server.context-path=/

#配置Tomcat

server.tomcat.uri-encoding=UTF-8

server.tomcat.compression=off #是否开启压缩，默认是关闭

#### 替换 Tomcat

Spring Boot 默认使用 Tomcat 作为内嵌的 Servlet 容器，如果要使用 Jetty 或者 Undertow 为容器，只需修改 spring-boot-start-web 的依赖即可。

1. 替换为 Jetty

在 pom.xml中，将 spring-boot-starter-web 的依赖由 spring-boot-starter-tomcat 替换为 spring-boot-starter-jetty：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<!-- 使用Jetty，需要在spring-boot-starter-web排除spring-boot-starter-tomcat，因为SpringBoot默认使用tomcat -->

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

<!-- Jetty适合长连接应用，就是聊天类的长连接 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

启动 Spring Boot，控制台输出效果如下：

INFO 21724 --- [ main] o.s.b.web.embedded.jetty.JettyWebServer : Jetty started on port(s) 8080 (http/1.1) with context path '/'

1. 替换为 Undertow

将 spring-boot-starter-web 的依赖由 spring-boot-starter-tomcat 替换为 spring-boot-starter-undertow：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boo</groupId>

<artifactId>spring-boot-starter-undertow</artifactId>

</dependency>

## Spring Boot 的数据访问

### Docker 常用命令及参数

#### Docker 镜像命令

（1）Docker 镜像检索

docker search redis

（2）镜像下载

docker pull redis

（3）镜像列表

docker images

（4）删除镜像

删除指定镜像：

docker rmi image-id

删除所有镜像：

docker rmi ${docker images -q}

#### Docker 容器命令

（1）容器基本操作

运行容器，Docker 会为我们生成唯一的标识。

docker run --name container-name -d image-name

（2）容器列表

查看运行中的容器列表：

docker ps

查看运行和停止状态的容器：

docker ps -a

（3）停止和启动容器

停止容器：

docker stop container-name/container-id

启动容器：

docker start container-name/container-id

端口映射：Docker 容器中运行的软件所使用的端口，在本机和本机的局域网是不能访问的，需要将 Docker 容器的端口映射到当前主机的端口上，这样我们在本机和本机的局域网才能访问该软件。

映射容器的6379端口到虚拟机的6378端口上：

docker run -d -p 6378:6379 --name port-redis resdis

删除单个容器：

docker rm container-id

删除所有容器：

docker rm ${docker ps -a -q}

查看容器日志：

docker logs container-name/container-id

登录容器：

docker exec -it container-id/container-name bash

登录后可以在容器中进行常规的 Linux 系统操作命令。

### Spring Boot 对 Spring Data JPA 的支持

JPA 是一个基于 O/R 映射的标准规范（不提供实现）。Spring Data JPA 是 Spring Data 的一个子项目，它通过提供基于 JPA 的Repository 极大地减少了 JPA 作为数据访问方案的代码量。

#### JDBC 的自动配置

spring-boot-starter-data-jpa 依赖于 spring-boot-starter-jdbc，Spring Boot 对 JDBC 做了一些自动配置，源码在 org.springframework.boot.autoconfigure.jdbc 下。

在 DataSourceAutoConfiguration 类中配置了对 DataSource 的自动配置，通过"spring.datasource"为前缀的属性自动配置 dataSource；Spring Boot 开启了注解事务的支持（@EnableTransactionManagement）；还配置了一个jdbcTemplate。以下是 JdbcProperties 类的源码：

@ConfigurationProperties(

prefix = "spring.jdbc"

)

public class JdbcProperties {

private final JdbcProperties.Template template = new JdbcProperties.Template();

public JdbcProperties() {

}

public JdbcProperties.Template getTemplate() {

return this.template;

}

public static class Template {

private int fetchSize = -1;

private int maxRows = -1;

@DurationUnit(ChronoUnit.SECONDS)

private Duration queryTimeout;

...

}

}

#### 对 JPA 的自动配置

Spring Boot 对 JPA 的自动配置放置在 org.springframework.boot.autoconfiguration.orm.jpa 下，从HibernateJpaAutoConfiguration 可以看出，Spring Boot 默认的 JPA 实现是Hibernate。

配置 JPA 可以在 application.properties 中使用 spring.jpa 为前缀的属性来配置。

以下是 JpaProperties 类的源码：

@ConfigurationProperties(

prefix = "spring.jpa"

)

public class JpaProperties {

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

private final List<String> mappingResources = new ArrayList();

private String databasePlatform;

private Database database;

private boolean generateDdl = false;

private boolean showSql = false;

private Boolean openInView;

...

}

在 JpaBaseConfiguration 类中，Spring Boot 为我们创建了 transactionManager、jpaVendorAdapter、entityManagerFactory 等 bean。JpaBaseConfiguration 还有 getPackagesToScan 方法，可以自动扫描有@Entity 注解的实体类。

#### 对 Spring Data JPA 的自动配置

Spring Boot 对 Spring Data JPA 的自动配置放置在 org.springframework.boot.autoconfigure.data.jpa 中。JpaRepositoriesAutoConfiguration 是依赖于 HibernateJpaAutoConfiguration 配置的，且 Spring Boot 自动开启了对 Spring Data JPA 的支持，无需在配置类显式声明@EnableJpaRepositories。

在 Spring Boot 下使用 Spring Data JPA，首先在项目的 maven 依赖里添加 spring-boot-starter-data-jpa，然后只需定义DataSource、实体类和数据访问层，在需要使用数据访问的地方注入数据访问层的 Bean 即可。

## @Value原理

@Value的解析就是在bean初始化阶段。BeanPostProcessor定义了bean初始化前后用户可以对bean进行操作的接口方法，它的一个重要实现类AutowiredAnnotationBeanPostProcessor为bean中的@Autowired和@Value注解的注入功能提供支持。

## 启动过程

准备Environment——发布事件——创建上下文、bean——刷新上下文——结束。

构造SpringApplication的时候会进行初始化的工作，初始化的时候会做以下几件事： 判断运行环境类型，有三种运行环境：NONE 非 web 的运行环境、SERVLET 普通 web 的运行环境、REACTIVE 响应式 web 的运行环境 加载 spring.factories 配置文件, 并设置 ApplicationContextInitializer 加载配置文件, 设置 ApplicationListener

SpringApplication构造完成之后调用run方法，启动SpringApplication，run方法执行的时候会做以下几件事： 构造一个StopWatch，观察SpringApplication的执行 找出SpringApplicationRunListener，用于监听SpringApplication run方法的执行。监听的过程中会封装SpringApplicationEvent事件，然后使用ApplicationEventMulticaster广播出去，应用程序监听器ApplicationListener会监听到这些事件 发布starting事件 加载配置资源到environment，包括命令行参数、application.yml等 发布environmentPrepared事件 创建并初始化ApplicationContext，设置environment，加载配置 refresh ApplicationContext

* 设置beanFactory
* 调用BeanFactoryPostProcessors
* 初始化消息源
* 初始化事件广播器（initApplicationEventMulticaster）
* 调用onRefresh()方法，默认是空实现
* 注册监听器
* 实例化non-lazy-init单例
* 完成refresh
* 发布ContextRefreshedEvent事件

发布started事件，启动结束

# 自动配置原理

在 application.properties 中设置属性 debug=true，可以在控制台查看已启用和未启用的自动配置。

@SpringBootApplication是@Configuration、@EnableAutoConfiguration和@ComponentScan的组合。

@Configuration表示该类是Java配置类。

@ComponentScan开启自动扫描符合条件的bean（添加了@Controller、@Service等注解）。

@EnableAutoConfiguration会根据类路径中的jar依赖为项目进行自动配置，比如添加了spring-boot-starter-web依赖，会自动添加Tomcat和Spring MVC的依赖，然后Spring Boot会对Tomcat和Spring MVC进行自动配置（spring.factories EnableAutoConfiguration配置了WebMvcAutoConfiguration）。

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@AutoConfigurationPackage

@Import(EnableAutoConfigurationImportSelector.class)

public @interface EnableAutoConfiguration {

}

EnableAutoConfiguration主要由 @AutoConfigurationPackage，@Import(EnableAutoConfigurationImportSelector.class)这两个注解组成的。

@AutoConfigurationPackage用于将启动类所在的包里面的所有组件注册到spring容器。

@Import 将EnableAutoConfigurationImportSelector注入到spring容器中，EnableAutoConfigurationImportSelector通过SpringFactoriesLoader从类路径下去读取META-INF/spring.factories文件信息，此文件中有一个key为org.springframework.boot.autoconfigure.EnableAutoConfiguration，定义了一组需要自动配置的bean。

# Auto Configure

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\

org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\

org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\

org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\

org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\

这些配置类不是都会被加载，会根据xxxAutoConfiguration上的@ConditionalOnClass等条件判断是否加载，符合条件才会将相应的组件被加载到spring容器。（比如mybatis-spring-boot-starter，会自动配置sqlSessionFactory、sqlSessionTemplate、dataSource等mybatis所需的组件）

@Configuration

@ConditionalOnClass({ EnableAspectJAutoProxy.class, Aspect.class, Advice.class,

AnnotatedElement.class }) //类路径存在EnableAspectJAutoProxy等类文件，才会加载此配置类

@ConditionalOnProperty(prefix = "spring.aop", name = "auto", havingValue = "true", matchIfMissing = true)

public class AopAutoConfiguration {

@Configuration

@EnableAspectJAutoProxy(proxyTargetClass = false)

@ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue = "false", matchIfMissing = false)

public static class JdkDynamicAutoProxyConfiguration {

}

@Configuration

@EnableAspectJAutoProxy(proxyTargetClass = true)

@ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue = "true", matchIfMissing = true)

public static class CglibAutoProxyConfiguration {

}

}

全局配置文件中的属性如何生效，比如：server.port=8081，是如何生效的？

@ConfigurationProperties的作用就是将配置文件的属性绑定到对应的bean上。全局配置的属性如：server.port等，通过@ConfigurationProperties注解，绑定到对应的XxxxProperties bean，通过这个 bean 获取相应的属性（serverProperties.getPort()）。

//server.port = 8080

@ConfigurationProperties(prefix = "server", ignoreUnknownFields = true)

public class ServerProperties {

private Integer port;

private InetAddress address;

@NestedConfigurationProperty

private final ErrorProperties error = new ErrorProperties();

private Boolean useForwardHeaders;

private String serverHeader;

//...

}

# 实现自动配置

实现当某个类存在时，自动配置这个类的bean，并且可以在application.properties中配置bean的属性。

**（1）新建Maven项目spring-boot-starter-hello，修改pom.xml如下：**

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.tyson</groupId>

<artifactId>spring-boot-starter-hello</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-autoconfigure</artifactId>

<version>1.3.0.M1</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

</dependency>

</dependencies>

</project>

**（2）属性配置**

public class HelloService {

private String msg;

public String getMsg() {

return msg;

}

public void setMsg(String msg) {

this.msg = msg;

}

public String sayHello() {

return "hello" + msg;

}

}

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix="hello")

public class HelloServiceProperties {

private static final String MSG = "world";

private String msg = MSG;

public String getMsg() {

return msg;

}

public void setMsg(String msg) {

this.msg = msg;

}

}

**（3）自动配置类**

import com.tyson.service.HelloService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;

import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;

import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;

import org.springframework.boot.context.properties.EnableConfigurationProperties;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

@EnableConfigurationProperties(HelloServiceProperties.class) //1

@ConditionalOnClass(HelloService.class) //2

@ConditionalOnProperty(prefix="hello", value = "enabled", matchIfMissing = true) //3

public class HelloServiceAutoConfiguration {

@Autowired

private HelloServiceProperties helloServiceProperties;

@Bean

@ConditionalOnMissingBean(HelloService.class) //4

public HelloService helloService() {

HelloService helloService = new HelloService();

helloService.setMsg(helloServiceProperties.getMsg());

return helloService;

}

}

1. @EnableConfigurationProperties 注解开启属性注入，将带有@ConfigurationProperties 注解的类注入为Spring 容器的 Bean。
2. 当 HelloService 在类路径的条件下。
3. 当设置 hello=enabled 的情况下，如果没有设置则默认为 true，即条件符合。
4. 当容器没有这个 Bean 的时候。

**（4）注册配置**

想要自动配置生效，需要注册自动配置类。在 src/main/resources 下新建 META-INF/spring.factories。添加以下内容：

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

com.tyson.config.HelloServiceAutoConfiguration

"\"是为了换行后仍然能读到属性。若有多个自动配置，则用逗号隔开。

（5）使用starter

在 Spring Boot 项目的 pom.xml 中添加：

<dependency>

<groupId>com.tyson</groupId>

<artifactId>spring-boot-starter-hello</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

运行类如下：

import com.tyson.service.HelloService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@SpringBootApplication

public class SpringbootDemoApplication {

@Autowired

public HelloService helloService;

@RequestMapping("/")

public String index() {

return helloService.getMsg();

}

public static void main(String[] args) {

SpringApplication.run(SpringbootDemoApplication.class, args);

}

}

在项目中没有配置 HelloService bean，但是我们可以注入这个bean，这是通过自动配置实现的。

在 application.properties 中添加 debug 属性，运行配置类，在控制台可以看到：

HelloServiceAutoConfiguration matched:

- @ConditionalOnClass found required class 'com.tyson.service.HelloService' (OnClassCondition)

- @ConditionalOnProperty (hello.enabled) matched (OnPropertyCondition)

HelloServiceAutoConfiguration#helloService matched:

- @ConditionalOnMissingBean (types: com.tyson.service.HelloService; SearchStrategy: all) did not find any beans (OnBeanCondition)

可以在 application.properties 中配置 msg 的内容：

hello.msg=大彬