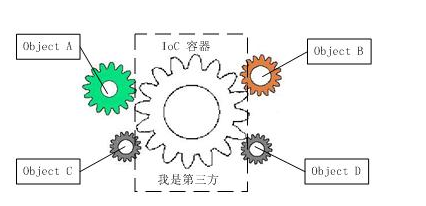
# Spring

## IoC

Spring 简化Java企业应用的开发，最核心的部分：IoC、AOP

**控制反转（又叫依赖注入）**（Inversion of Control, IoC）：IOC理论，用来实现**对象**之间的“解耦”，目前这个理论已经被成功地应用到实践当中，很多的J2EE项目均采用了IOC框架产品Spring。借助于“第三方”实现具有依赖关系的对象之间的解耦，如下图：



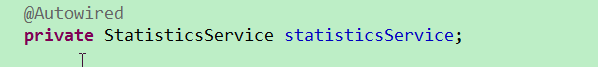
大家看到了吧，由于引进了中间位置的“第三方”，也就是IOC容器，使得A、B、C、D这4个对象没有了耦合关系，齿轮之间的传动全部依靠“第三方”了，全部对象的控制权全部上缴给“第三方”IOC容器，所以，IOC容器成了整个系统的关键核心，它起到了一种类似“粘合剂”的作用，把系统中的所有对象粘合在一起发挥作用，如果没有这个“粘合剂”，对象与对象之间会彼此失去联系，这就是有人把IOC容器比喻成“粘合剂”的由来。

软件系统在没有引入IOC容器之前，如图1所示，对象A依赖于对象B，那么对象A在初始化或者运行到某一点的时候，自己必须主动去创建对象B或者使用已经创建的对象B。无论是创建还是使用对象B，控制权都在自己手上。

软件系统在引入IOC容器之后，这种情形就完全改变了，如图3所示，由于IOC容器的加入，对象A与对象B之间失去了直接联系，所以，当对象A运行到需要对象B的时候，**IOC容器会主动创建**一个对象B注入到对象A需要的地方。

通过前后的对比，我们不难看出来：对象A获得依赖对象B的过程,由主动行为变为了被动行为，控制权颠倒过来了，这就是“控制反转”这个名称的由来。

IOC中最基本的技术就是“**反射**(Reflection)”编程，通俗来讲就是根据给出的类名（字符串方式）来动态地生成对象



IOC又叫依赖注入（Dependence Injection ,DI）。2004年，Martin Fowler探讨了同一个问题，既然IOC是控制反转，那么到底是“哪些方面的控制被反转了呢？”，经过详细地分析和论证后，他得出了答案：“获得依赖对象的过程被反转了”。控制被反转之后，获得依赖对象的过程由自身管理变为了由IOC容器主动注入。于是，他给“控制反转”取了一个更合适的名字叫做“依赖注入（Dependency Injection）”。他的这个答案，实际上给出了实现IOC的方法：注入。所谓依赖注入，就是由IOC容器在运行期间，动态地将某种依赖关系注入到对象之中。

所以，依赖注入(DI)和控制反转(IOC)是从不同的角度的描述的同一件事情，就是指通过引入IOC容器，利用依赖关系注入的方式，实现对象之间的解耦。

**使用IOC框架应该注意什么**

使用IOC框架产品能够给我们的开发过程带来很大的好处，但是也要充分认识引入IOC框架的缺点，做到心中有数，杜绝滥用框架。

第一、软件系统中由于引入了第三方IOC容器，生成对象的步骤变得有些复杂，本来是两者之间的事情，又凭空多出一道手续，所以，我们在刚开始使用IOC框架的时候，会感觉系统变得不太直观。所以，引入了一个全新的框架，就会增加团队成员学习和认识的培训成本，并且在以后的运行维护中，还得让新加入者具备同样的知识体系。

第二、由于IOC容器生成对象是通过反射方式，在**运行效率上有一定的损耗**。如果你要追求运行效率的话，就必须对此进行权衡。

第三、具体到IOC框架产品(比如：Spring)来讲，需要进行大量的配制工作，比较繁琐，对于一些小的项目而言，客观上也可能加大一些工作成本。

第四、IOC框架产品本身的成熟度需要进行评估，如果引入一个不成熟的IOC框架产品，那么会影响到整个项目，所以这也是一个隐性的风险。

我们大体可以得出这样的结论：一些工作量不大的项目或者产品，不太适合使用IOC框架产品。另外，如果团队成员的知识能力欠缺，对于IOC框架产品缺乏深入的理解，也不要贸然引入。最后，特别强调运行效率的项目或者产品，也不太适合引入IOC框架产品，象WEB2.0网站就是这种情况。

## AOP

**和面向切面编程**（aspect-oriented programming, AOP）。

### @Pointcut

定义一个方法为切点里面的内容为一个表达式,需要一个切点来处理他们两者,我们可以这么配置：

@Pointcut(value = "controllerAspect()||controllerDemo()")

### @Before

在切点前执行方法,内容为指定的切点

### @After

在切点后,return前执行,

### @AfterReturning

在切入点,return后执行,如果想对某些方法的返回参数进行处理,可以在这操作

### @Around

环绕切点,在进入切点前,跟切点后执行

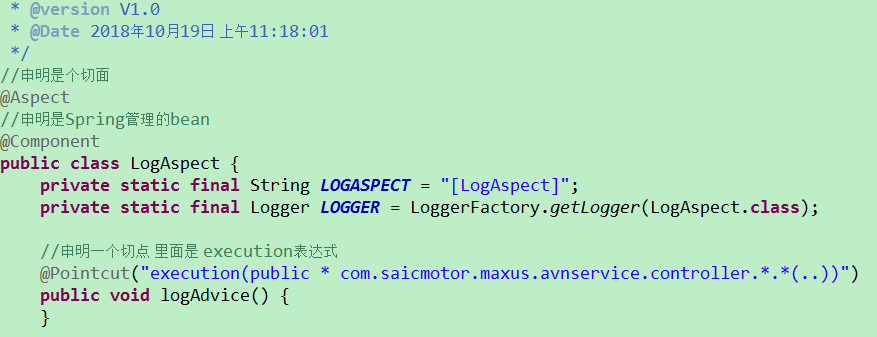
### @AfterThrowing

在切点后抛出异常进行处理

### @order(i)

标记切点的优先级,i越小,优先级越高

日志切面举例：

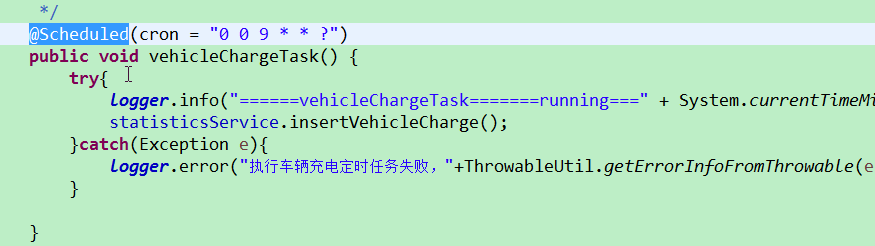




## @Scheduled cron

**Spring 定时任务之 @Scheduled cron表达式**

一个cron表达式有至少6个（也可能7个）有空格分隔的时间元素。



按顺序依次为

秒（0~59）

分钟（0~59）

小时（0~23）

天（月）（0~31，但是你需要考虑你月的天数）

月（0~11）

天（星期）（1~7 1=SUN 或 SUN，MON，TUE，WED，THU，FRI，SAT）

7.年份（1970－2099）

其中每个元素可以是一个值(如6),一个连续区间(9-12),一个间隔时间(8-18/4)(/表示每隔4小时),一个列表(1,3,5),通配符。由于"月份中的日期"和"星期中的日期"这两个元素互斥的,必须要对其中一个设置?.

0 0 10,14,16 \* \* ? 每天上午10点，下午2点，4点

0 0/30 9-17 \* \* ? 朝九晚五工作时间内每半小时

0 0 12 ? \* WED 表示每个星期三中午12点

"0 0 12 \* \* ?" 每天中午12点触发

"0 15 10 ? \* \*" 每天上午10:15触发

"0 15 10 \* \* ?" 每天上午10:15触发

"0 15 10 \* \* ? \*" 每天上午10:15触发

"0 15 10 \* \* ? 2005" 2005年的每天上午10:15触发

"0 \* 14 \* \* ?" 在每天下午2点到下午2:59期间的每1分钟触发

"0 0/5 14 \* \* ?" 在每天下午2点到下午2:55期间的每5分钟触发

"0 0/5 14,18 \* \* ?" 在每天下午2点到2:55期间和下午6点到6:55期间的每5分钟触发

"0 0-5 14 \* \* ?" 在每天下午2点到下午2:05期间的每1分钟触发

"0 10,44 14 ? 3 WED" 每年三月的星期三的下午2:10和2:44触发

"0 15 10 ? \* MON-FRI" 周一至周五的上午10:15触发

"0 15 10 15 \* ?" 每月15日上午10:15触发

"0 15 10 L \* ?" 每月最后一日的上午10:15触发

"0 15 10 ? \* 6L" 每月的最后一个星期五上午10:15触发

"0 15 10 ? \* 6L 2002-2005" 2002年至2005年的每月的最后一个星期五上午10:15触发

"0 15 10 ? \* 6#3" 每月的第三个星期五上午10:15触发

有些子表达式能包含一些范围或列表

例如：子表达式（天（星期））可以为 “MON-FRI”，“MON，WED，FRI”，“MON-WED,SAT”

“\*”字符代表所有可能的值

因此，“\*”在子表达式（月）里表示每个月的含义，“\*”在子表达式（天（星期））表示星期的每一天

“/”字符用来指定数值的增量

例如：在子表达式（分钟）里的“0/15”表示从第0分钟开始，每15分钟

在子表达式（分钟）里的“3/20”表示从第3分钟开始，每20分钟（它和“3，23，43”）的含义一样

“？”字符仅被用于天（月）和天（星期）两个子表达式，表示不指定值

当2个子表达式其中之一被指定了值以后，为了避免冲突，需要将另一个子表达式的值设为“？”

“L” 字符仅被用于天（月）和天（星期）两个子表达式，它是单词“last”的缩写

但是它在两个子表达式里的含义是不同的。

在天（月）子表达式中，“L”表示一个月的最后一天

在天（星期）自表达式中，“L”表示一个星期的最后一天，也就是SAT

如果在“L”前有具体的内容，它就具有其他的含义了

例如：“6L”表示这个月的倒数第６天，“ＦＲＩＬ”表示这个月的最一个星期五

注意：在使用“L”参数时，不要指定列表或范围，因为这会导致问题

POJO = "Plain Old Java Object"，是MartinFowler等发明的一个术语，用来表示普通的Java对象，不是JavaBean, EntityBean 或者 SessionBean

# Spring设计模式

## 简单工厂

又叫做静态工厂方法（StaticFactory Method）模式，但不属于23种GOF设计模式之一

## 工厂模式（Factory Method）

通常由应用程序直接使用new创建新的对象，为了将对象的创建和使用相分离，采用工厂模式,即应用程序将对象的创建及初始化职责交给工厂对象

## 单例模式（Singleton）

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

spring中的单例模式完成了后半句话，即提供了全局的访问点BeanFactory。但没有从构造器级别去控制单例，这是因为spring管理的是是任意的java对象。

核心提示点：Spring下默认的bean均为singleton，可以通过singleton=“true|false” 或者 scope=“？”来指定

Controller类@Scope(value = "prototype")

## 适配器模式（Adapter）

比如说Springmvc的HandlerInterceptorAdapter就是个接口适配器，实现了AsyncHandlerInterceptor(HandlerInterceptor的子类)，ThemeChangeInterceptor继承至HandlerInterceptorAdapter，只需要重写关注的方法即可，不相关的方法完全可以忽略。

## 包装器（Decorator）

## 代理模式（Proxy）

静态代理在Spring websocket里面非常常见，比如说WebsocketHandlerDecorator，因为它不方便直接操作开发者自定义的WebSocketHandler，因此使用了代理模式。另外，cglib、jdk动态代理在spring中也是非常常见

## 观察者（Observer）

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

## 策略（Strategy）

比如SpringMVC的HandlerMethodArgumentResolver接口，使用HandlerMethodArgumentResolverComposite(实现HandlerMethodArgumentResolver接口)，通过遍历内部所有的HandlerMethodArgumentResolver(当然有缓存 机制)，选择support返回true的实例，并把接口的调用交给该实例处理

其实，我们在写业务代码的时候，很多童鞋喜欢用Map保存策略实现类，其实这样并不好，不够灵活，应该像spring这样提供一个support方法，可以支持更加复杂的逻辑判断

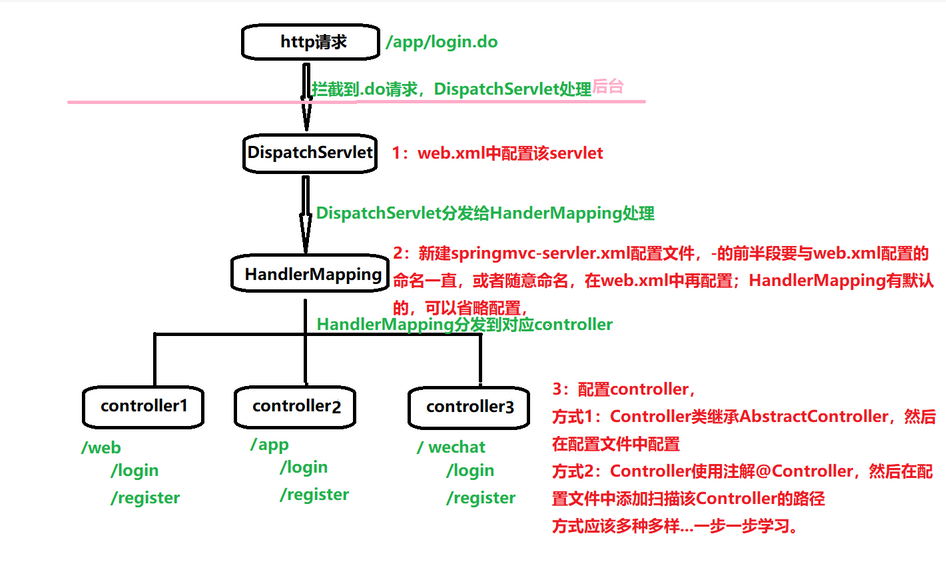
## 模板方法（Template Method）

父类定义了骨架（调用哪些方法及顺序），某些特定方法由子类实现

最大的好处：代码复用，减少重复代码。除了子类要实现的特定方法，其他方法及方法调用顺序都在父类中预先写好了。

# Spring-MVC

## SpimgMVC工作原理



SpringMVC流程

1、 用户发送请求至前端控制器DispatcherServlet。

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

3、 处理器映射器找到具体的处理器(可以根据xml配置、注解进行查找)，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经过适配调用具体的处理器(Controller，也叫后端控制器)。

6、 Controller执行完成返回ModelAndView。

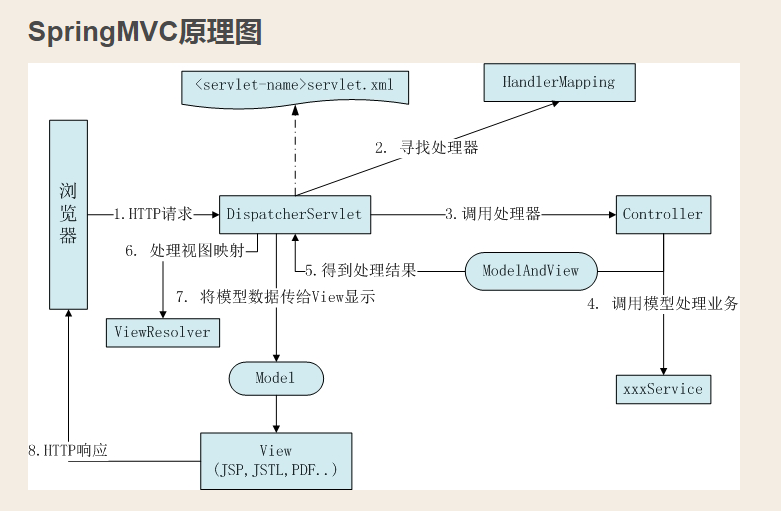
7、HandlerAdapter将 controller执行结果ModelAndView返回给DispatcherServlet。

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

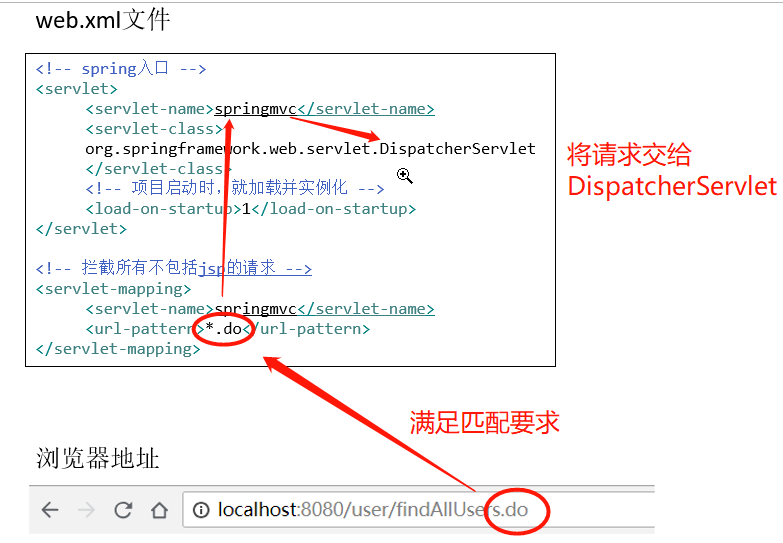
9、 ViewReslover解析后返回具体View。

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

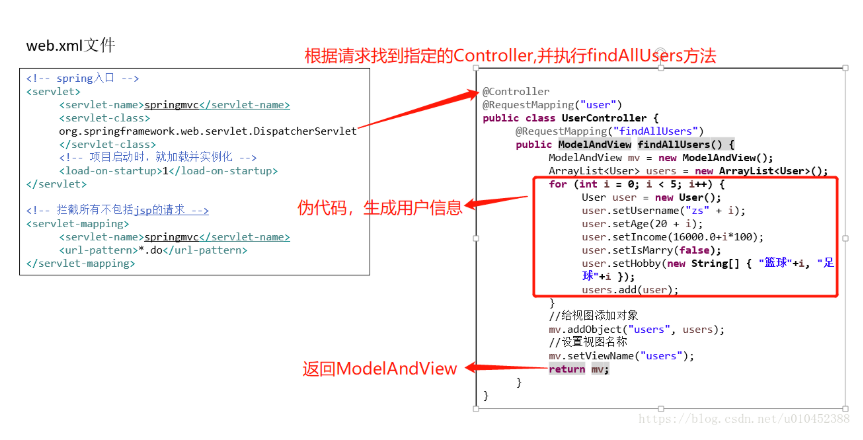
11、 DispatcherServlet响应用户



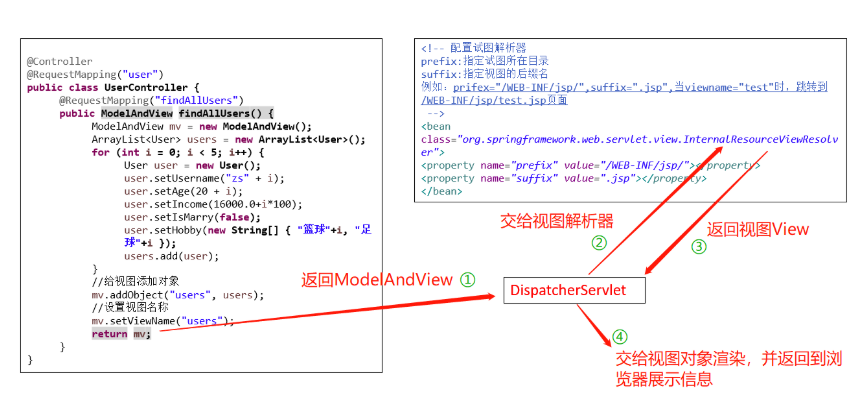
流浪器请求：

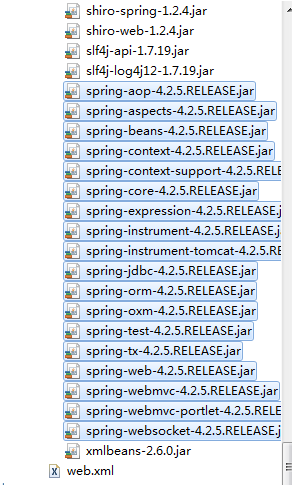
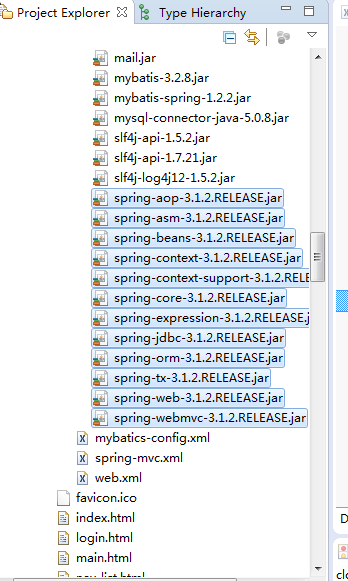


找到Controller



解析视图对象，返回流浪器





除了spring.jar文件，Spring还包括有其它13个独立的jar包，各自包含着对应的Spring组件，用户可以根据自己的需要来选择组合自己的jar包，而不必引入整个spring.jar的所有类文件。

1. spring-aop.jar

这个jar文件包含在应用中使用Spring的AOP特性时所需的类。使用基于AOP的Spring特性，如声明型事务管理（Declarative Transaction Management），也要在应用里包含这个jar包。

1. spring-beans.jar

这个jar文件是所有应用都要用到的，它包含访问配置文件、创建和管理bean以及进行Inversion of Control / Dependency Injection（IoC/DI）操作相关的所有类。如果应用只需基本的IoC/DI支持，引入spring-core.jar及spring-beans.jar文件就可以了。

1. spring-context.jar

这个jar文件为Spring核心提供了大量扩展。可以找到使用Spring ApplicationContext特性时所需的全部类，JDNI所需的全部类，UI方面的用来与模板（Templating）引擎如Velocity、FreeMarker、JasperReports集成的类，以及校验Validation方面的相关类。

1. spring-core.jar

这个jar文件包含Spring框架基本的核心工具类，Spring其它组件要都要使用到这个包里的类，是其它组件的基本核心，当然你也可以在自己的应用系统中使用这些工具类。

1. spring-dao.jar

这个jar文件包含Spring DAO、Spring Transaction进行数据访问的所有类。为了使用声明型事务支持，还需在自己的应用里包含spring-aop.jar。

1. spring-hibernate.jar

这个jar文件包含Spring对Hibernate 2及Hibernate 3进行封装的所有类。

1. spring-jdbc.jar

这个jar文件包含对Spring对JDBC数据访问进行封装的所有类。

1. spring-orm.jar

这个jar文件包含Spring对DAO特性集进行了扩展，使其支持 iBATIS、JDO、OJB、TopLink，因为Hibernate已经独立成包了，现在不包含在这个包里了。这个jar文件里大部分的类都要依赖spring-dao.jar里的类，用这个包时你需要同时包含spring-dao.jar包。

1. spring-remoting.jar

这个jar文件包含支持EJB、JMS、远程调用Remoting（RMI、Hessian、Burlap、Http Invoker、JAX-RPC）方面的类。

1. spring-support.jar

这个jar文件包含支持缓存Cache（ehcache）、JCA、JMX、邮件服务（Java Mail、COS Mail）、任务计划Scheduling（Timer、Quartz）方面的类。

1. spring-web.jar

　　这个jar文件包含Web应用开发时，用到Spring框架时所需的核心类，包括自动载入WebApplicationContext特性的类、Struts与JSF集成类、文件上传的支持类、Filter类和大量工具辅助类。

1. spring-webmvc.jar

　　这个jar文件包含Spring MVC框架相关的所有类。包含国际化、标签、Theme、视图展现的FreeMarker、JasperReports、Tiles、Velocity、XSLT相关类。当然，如果你的应用使用了独立的MVC框架，则无需这个JAR文件里的任何类。

1. spring-mock.jar

　　这个jar文件包含Spring一整套mock类来辅助应用的测试。Spring测试套件使用了其中大量mock类，这样测试就更加简单。模拟HttpServletRequest和HttpServletResponse类在Web应用单元测试是很方便的。

如何选择这些发布包，决定选用哪些发布包其实相当简单。

 如果你正在构建Web应用并将全程使用Spring，那么最好就使用单个全部的spring.jar文件；

 如果你的应用仅仅用到简单的Inversion of Control / Dependency Injection（IoC/DI）容器，那么只需spring-core.jar与spring-beans.jar即可；

 如果你对发布的大小要求很高，那么就得精挑细选了，只取包含自己所需特性的jar文件了。

采用独立的发布包你可以避免包含自己的应用不需要的全部类。当然你可以采用其它的一些工具来设法令整个应用包变小，节省空间的重点在于准确地找出自己所需的Spring依赖类，然后合并所需的类与包就可以了。

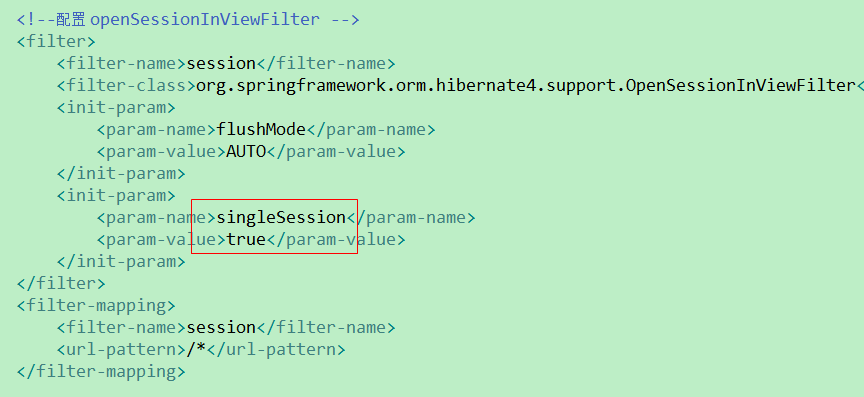
## 配置web.xml

### OpenSessionInViewFilter过滤器

作用是延长hibernate中Session的生命周期到页面，使用Proxy时获取数据

说明一下Open Session in View的作用，就是允许在每次的整个request的过程中使用同一个hibernate session,可以在这个request任何时期lazy loading数据。

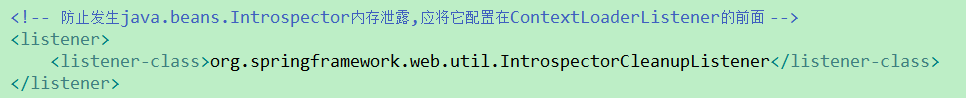
如果是singleSession=false的话，就不会在每次的整个request的过程中使用同一个hibernate session,而是每个数据访问都会产生各自的session,等于没有Open Session in View. OpenSessionInViewFilter默认是不会对session 进行flush的,并且flush mode 是 never



### IntrospectorCleanupListener监听器

1、此监听器主要用于解决java.beans.Introspector导致的内存泄漏的问题

2、此监听器应该配置在web.xml中与Spring相关监听器中的第一个位置(也要在ContextLoaderListener的前面)  
3、JDK中的java.beans.Introspector类的用途是发现Java类是否符合JavaBean规范如果有的框架或程序用到了Introspector类,那么就会启用一个系统级别的缓存,此缓存会存放一些曾加载并分析过的JavaBean的引用。当Web服务器关闭时,由于此缓存中存放着这些JavaBean的引用,所以垃圾回收器无法回收Web容器中的JavaBean对象,最后导致内存变大。而org.springframework.web.util.IntrospectorCleanupListener就是专门用来处理Introspector内存泄漏问题的辅助类。  
 IntrospectorCleanupListener会在Web服务器停止时清理Introspector缓存,使那些Javabean能被垃圾回收器正确回收。Spring自身不会出现这种问题，因为Spring在加载并分析完一个类之后会马上刷新  
     JavaBeans Introspector缓存,这就保证Spring中不会出现这种内存泄漏的问题。但有些程序和框架在使用了JavaBeans Introspector之后,没有进行清理工作(如  Quartz,Struts),最后导致内存泄漏。

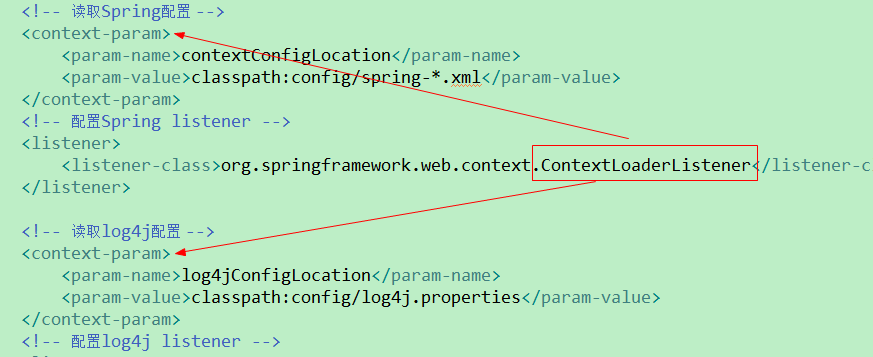


### ContextLoaderListener(Spring listener)

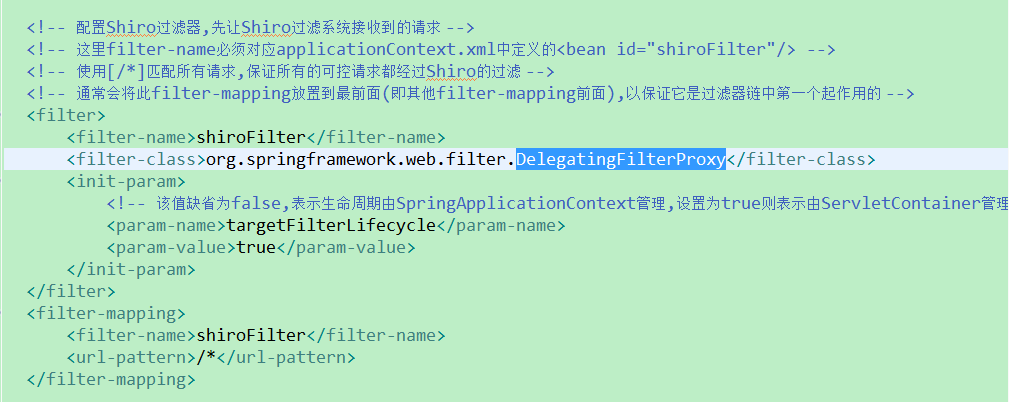
ContextLoaderListener属于Spring框架，其实现了ContextLoaderListener接口，本质是一个listener，当服务启动，Context初始化和销毁时被调用。

ContextLoaderListener会读取web.xml中定义的context-param标签

当然，你可以不定义，ContextLoaderListener会有一个默认的地址去寻找文件。ContextLoaderListener其主要功能是初始化Ioc容器，这样，我们在项目中管理bean容器就可以通过ApplicationContext实例化，而不必去new；还有，当你用到了Shiro 、Dubbo等其他需要Sping配置的框架，其需要在ContextLoaderListener中加载，因此需要定义在该文件下。



### Shiro过滤器



### 声明DispatchServlet前置控制器，监听请求

其继承了HttpServlet，本质是一个Servlet，Servlet初始化以及相关url访问时会被调用。

web.xml中classpath:和classpath\*: 有什么区别?

classpath：只会到你的class路径中查找找文件;

classpath\*：不仅包含class路径，还包括jar文件中(class路径)进行查找.

有时候会用模糊匹配的方式配置多配置文件。

但是如果配置文件是在jar包里，模糊匹配就找不到了。可以用逗号隔开的方式配置多个配置文件。

存放位置：

1：src下面

需要在web.xml中定义如下：

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

< /context-param>

2：WEB-INF下面

需要在web.xml中定义如下：

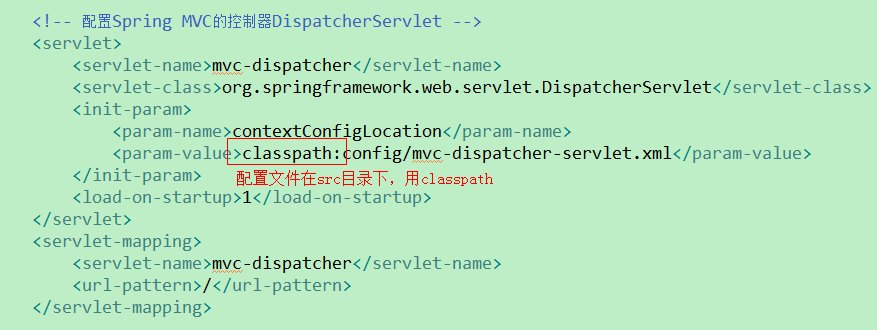
<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/applicationContext\*.xml</param-value>

</context-param>





### 过滤器CharacterEncodingFilter（filter-name）



### 其它



### 创建Controller



### 新建springmvc-servlet.xml并配置



# Spring-Cloud

分布式微服务架构下的一站式解决方案，是各个微服务架构落地技术的集合体，俗称微服务栈解决方案。

Spring Cloud Eureka是Spring Cloud Netflix项目下的服务治理模块。而Spring Cloud Netflix项目是Spring Cloud的子项目之一，主要内容是对Netflix公司一系列开源产品的包装，它为Spring Boot应用提供了自配置的Netflix OSS整合。通过一些简单的注解，开发者就可以快速的在应用中配置一下常用模块并构建庞大的分布式系统。它主要提供的模块包括：服务发现（Eureka），断路器（Hystrix），智能路由（Zuul），客户端负载均衡（Ribbon）等。

boot就是医院里面一个一个的科室，而Cloud就是把boot组合起来的提供对外的综合医院。

springboot可以单独使用，它不依赖于springcloud

1. 而springcloud必然依赖于springboot，属于依赖关系。

2. Springboot专注于快速方便的开发单个个体微服务。

3. SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来，为各个微服务之间提供，配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服务。

# Spring-boot

用来简化spring应用的初始搭建，使用特定方式配置（properties）

## 起步依赖

Spring提供了一系列starter来简化Maven配置。其核心原理也就是Maven和Gradle的依赖传递方案。当我们在我们的pom文件中增加对某个starter的依赖时，该starter的依赖也会自动的传递性被依赖进来。而且，很多starter也依赖了其他的starter。例如web starter就依赖了tomcat starter，并且大多数starter基本都依赖了spring-boot-starter。说明见[Spring Boot Starter](#Starter)

spring-boot-starter-web-services - SOAP Web Services

spring-boot-starter-web - Web 和 RESTful 应用程序

spring-boot-starter-test - 单元测试和集成测试

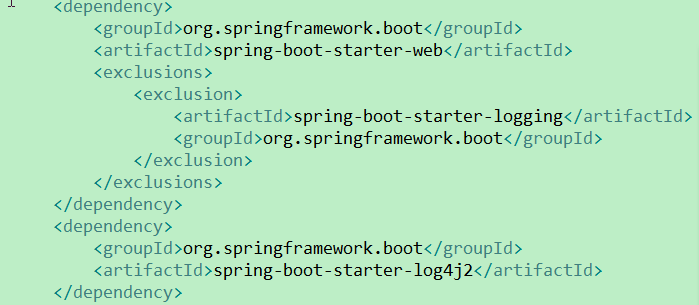
spring-boot-starter-jdbc - 传统的 JDBC

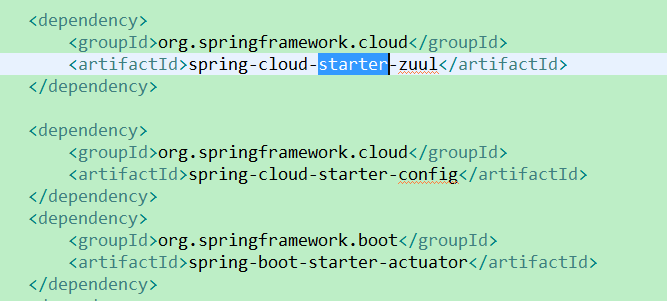
spring-boot-starter-hateoas - 为服务添加 HATEOAS 功能

spring-boot-starter-security - 使用 SpringSecurity 进行身份验证和授权

spring-boot-starter-data-jpa - 带有 Hibeernate 的 Spring Data JPA

spring-boot-starter-data-rest - 使用 Spring Data REST 公布简单的 REST 服务





## 启动入口

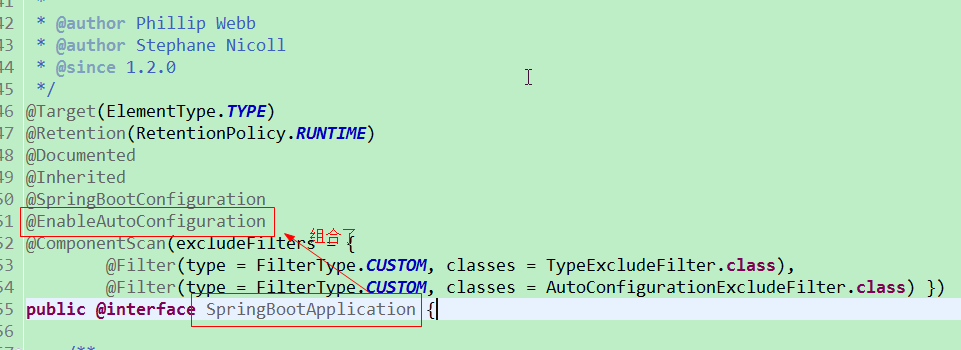
Spring Boot应用通常有一个名为\*Application的入口类，入口类中有一个main方法，这个方法其实就是一个标准的Java应用的入口方法。一般在main方法中使用SpringApplication.run()来启动整个应用。值得注意的是，这个入口类要使用@SpringBootApplication注解声明。@SpringBootApplication是Spring Boot的核心注解，他是一个**组合注解**。

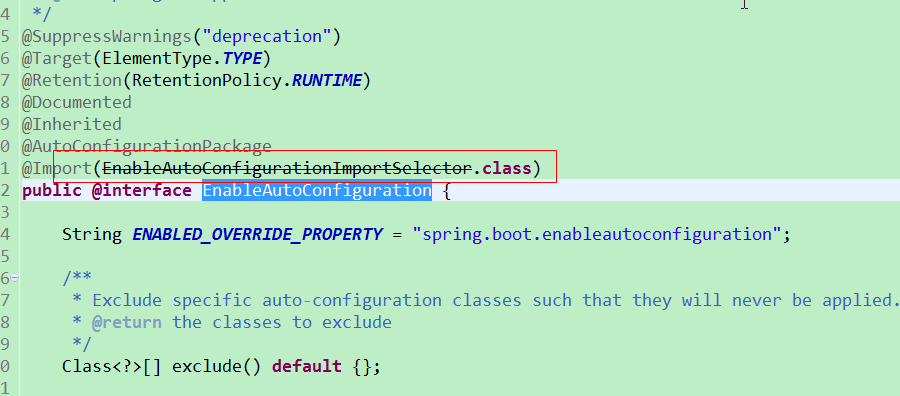


@SpringBootApplication是一个组合注解，它主要包含@SpringBootConfiguration、@EnableAutoConfiguration等几个注解。也就是说可以直接在启动类中使用这些注解来代替@ SpringBootApplication注解。 关于Spring Boot中的Spring自动化配置主要是@EnableAutoConfiguration的功劳。该注解可以让Spring Boot根据类路径中的jar包依赖为当前项目进行自动配置。

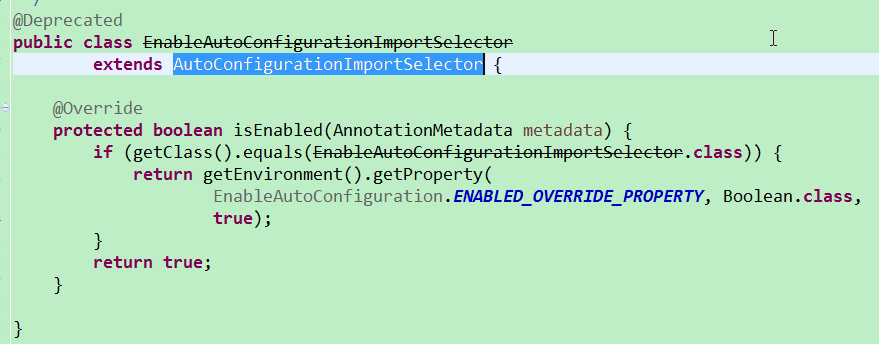
## 自动配置

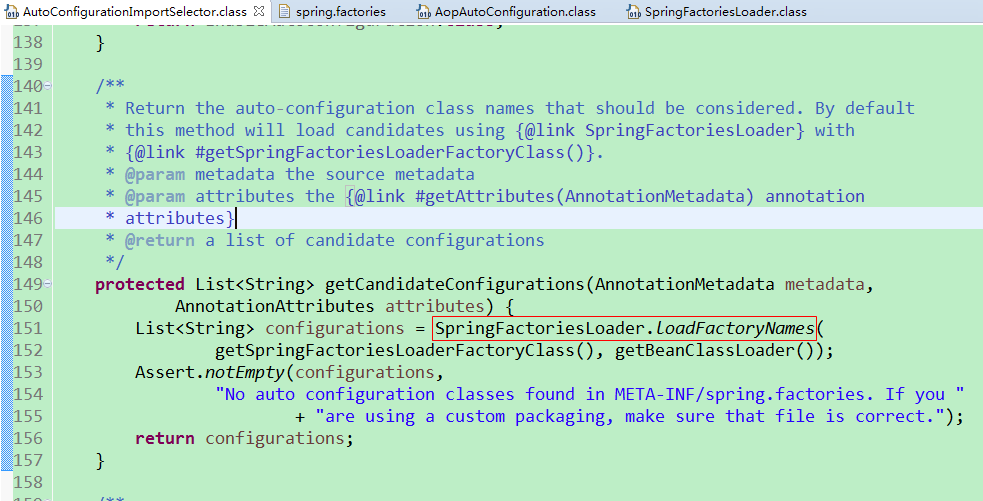
当一个mvc jar添加到程序中，怎么不用写配置文件，自动配置beans。Spring Boot 提供了配置应用程序和框架所需要的基本配置。这就是自动配置。Spring Boot的自动化配置主要是通过@EnableAutoConfiguration来实现的,@SpringBootApplication组合了@EnableAutoConfiguration



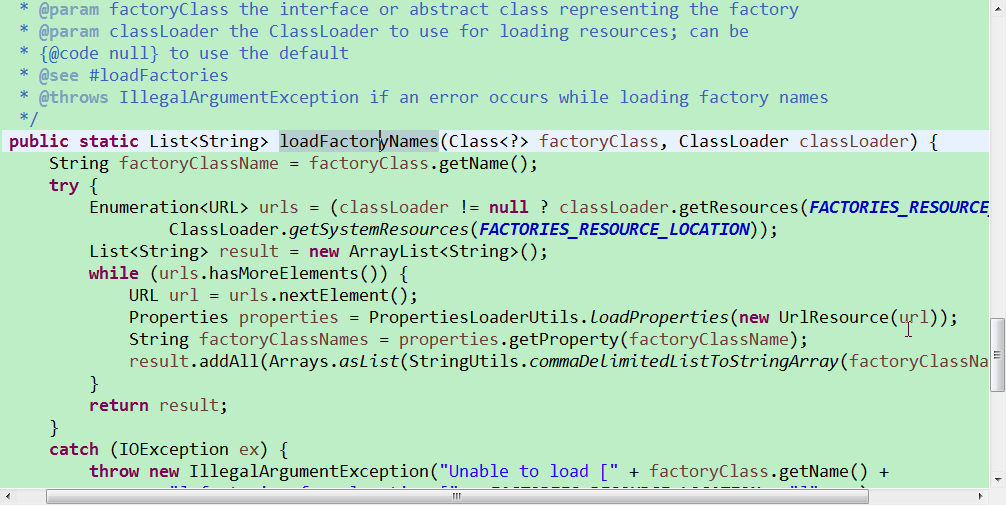


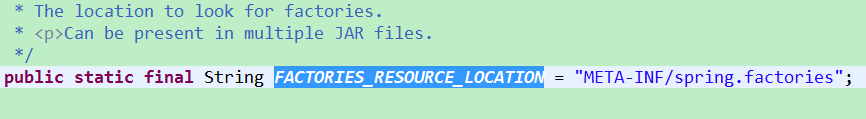
观察@EnableAutoConfiguration可以发现，这里Import了@EnableAutoConfigurationImportSelector，这就是Spring Boot自动化配置的“始作俑者”。之所以能够自动配置bean,是因为继承类org.springframework.boot.autoconfigure.AutoConfigurationImportSelector,该类主要是寻找一些必要的配置类



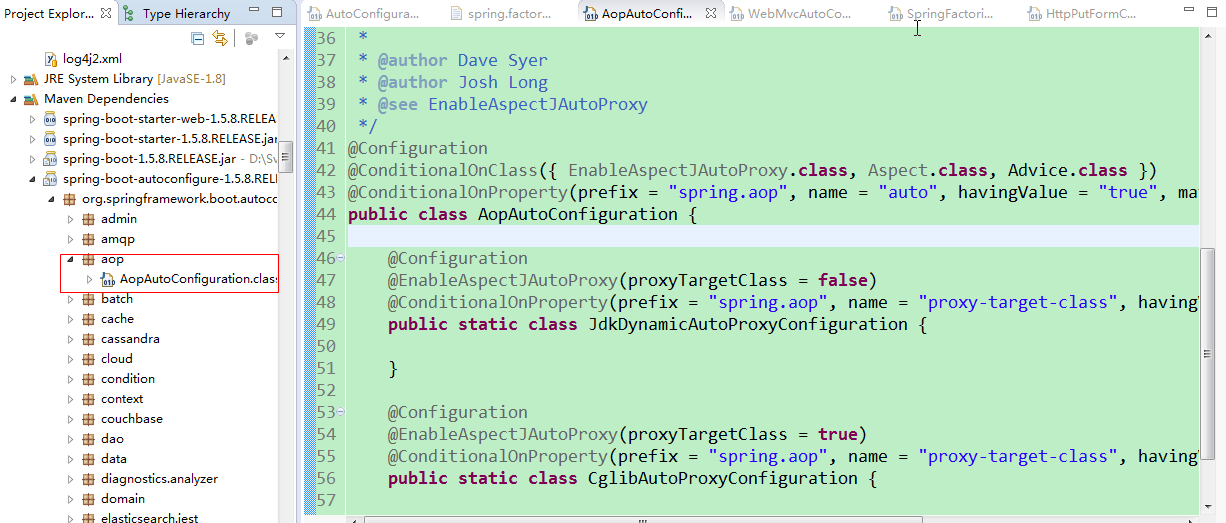


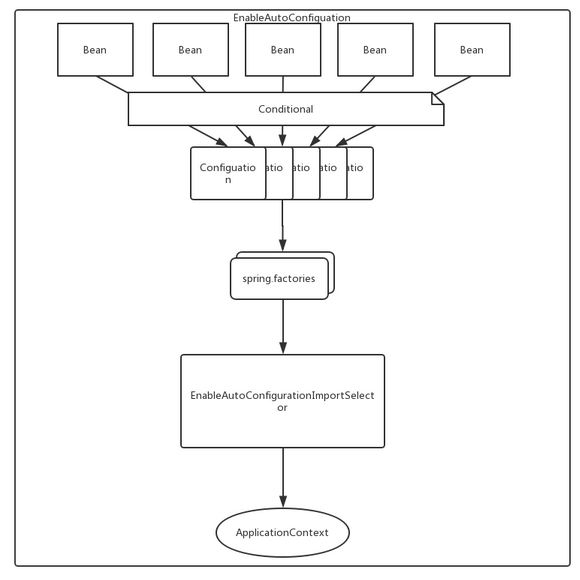
该方法是是返回spring boot约定需要自动配置的类，





以上就是自动配置的源码过程，我们从spring-boot-autoconfigure-1.5.1.RELEASE.jar中的spring.factories文件随便找一个Configuration，看看他是如何自动加载bean的。





通过Spring 4的条件配置决定哪些bean可以被配置，将这些条件定义成具体的Configuation，然后将这些Configuation配置到spring.factories文件中，作为key:org.springframework.boot.autoconfigure.EnableAutoConfiguration的值，这时候，容器在启动的时候，由于使用了EnableAutoConfiguration注解，该注解Import的EnableAutoConfigurationImportSelector会去扫描classpath下的所有spring.factories文件，然后进行bean的自动化配置。

所以，如果我们想要自定义一个starter的话，可以通过以上方式将自定义的starter中的bean自动化配置到Spring的上下文中，从而避免大量的配置。

## 4.4 条件化配置

将Bean交给spring托管很简单(怎么创建Bean)，根据功能在类上添加@Component,@Service,@Controller等等都行，如果是第三方类，也可以通过标有@Configuration的配置类来进行注入。但并不是所有被注入的bean都用得着，无脑注入会浪费资源。springboot提供了条件化配置，只有在满足注入条件才实例化。比如自定义一个ServiceHelloService，希望在spring.profiles.active=local时才加载。



[当@Conditional(HelloCondition.class)](mailto:当@Conditional(HelloCondition.class))条件的值为true的时候，HelloService才会被创建，否则该Bean会被忽略

@Conditional指定了一个条件，它的条件是一个Java类——HelloCondition,并要实现Condition接口并重写其中的Matches方法

除了可以自定义一些条件之外，Spring 4本身提供了很多已有的条件供直接使用，如：

@ConditionalOnBean

@ConditionalOnClass

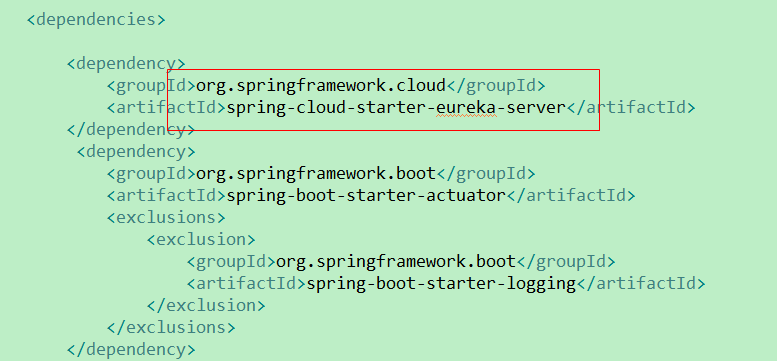
@ConditionalOnExpression

@ConditionalOnMissingBean

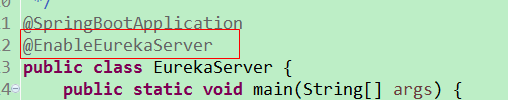
@ConditionalOnMissingClass

@ConditionalOnNotWebApplication

# eureka-server服务注册中心



想要实现一个服务注册中心的功能非常简单，只需要在项目的启动类EurekaServerApplication上使用@EnableEurekaServer注解即可



# Spring一系列比较及说明

### Spring Boot、Spring MVC 和 Spring 有什么区别

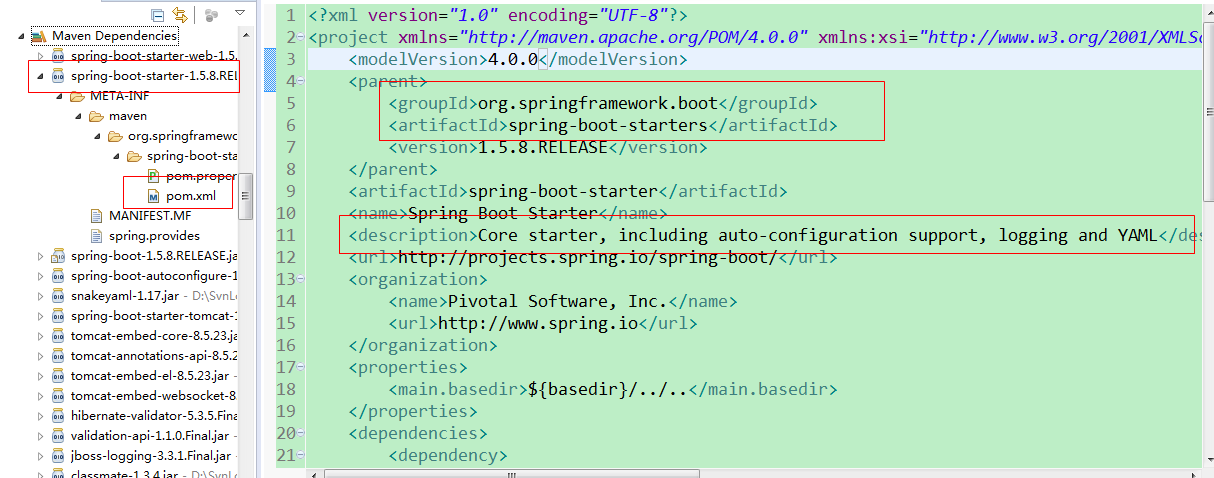
SpringFramework最重要特征是依赖注入（DI），所有 SpringModules 不是依赖注入就是 IOC 控制反转。很好的运用DI和IoC，开发松耦合应用，很容易的进行单元测试。

SpringMVC分离式方法开发WEB应用，通过运用像DispatcherServelet, MoudlAndView, ViewResolver等

SpringBoot通过一个自动配置和启动的项，来解决SpringFramework和SpringMVC大量配置项的问题，使开发者更加关注的是应用程序本身的业务和逻辑，SpringBoot提供了一些非功能性特征

### Spring Boot Starter

jar包，解压包出来会发现里面什么都没有，只定义了一些POM依赖。



web 应用程序或者是公开 REST 服务的应用程序



# javaBean

简单笼统的说就是一个类，一个可复用的类。一个实体类，用来封装对象，这个类里面全部都是属性值，和get，set方法

这个Java类必须具有一个无参的构造函数

属性必须私有化。

私有化的属性必须通过public类型的方法暴露给其它程序，并且方法的命名也必须遵守一定的命名规范

我们经常用的model就是javaBean

# Restful API

RESTful:URL定位资源，用HTTP动词（GET,POST,PUT,DELETE)描述操作。

实际上Restful本身不是一项什么高深的技术，而只是一种编程风格，或者说是一种设计风格。在传统的http接口设计中，我们一般只使用了get和post两个方法，然后用我们自己定义的词汇来表示不同的操作，比如上面查询文章的接口，我们定义了article/list.json来表示查询文章列表，可以通过get或者post方法来访问。而Restful API的设计则通过HTTP的方法来表示CRUD相关的操作。因此，除了get和post方法外，还会用到其他的HTTP方法，如PUT、DELETE、HEAD等，通过不同的HTTP方法来表示不同含义的操作。下面是我设计的一组对文章的增删改查的Restful API：



有三个URL映射是相同的，即都是/article/{id}，这在@Controller标识的类中是不允许出现的。这里的可以通过method来进行区分，可以看出，URL仅仅是标识资源的路劲，而具体的行为由HTTP方法来指定。

我们使用的是@RestController这个注解，而不是@Controller，不过这个注解同样不是Spring boot提供的，而是Spring MVC4（spring-boot-starter-web）中的提供的注解，表示一个支持Restful的控制器。

所以RESTful API就是REST风格的API。 那么在什么场景下使用RESTful API呢？在当今的互联网应用的前端展示媒介很丰富。有手机、有平板电脑还有PC以及其他的展示媒介。那么这些前端接收到的用户请求统一由一个后台来处理并返回给不同的前端肯定是最科学和最经济的方式，RESTful API就是一套协议来规范多种形式的前端和同一个后台的交互方式。

# Netty

IO：面向字节流或字符流的，以流式的方式顺序地从一个 Stream 中读取一个或多个字节, 因此也就不能随意改变读取指针的位置。Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。

NIO：即New IO，引入了 Channel 和 Buffer 的概念，面向缓冲的。从 Channel 中读取数据到 Buffer 中或将数据从 Buffer 中写入到 Channel。可以随意地读取任意位置的数据。使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，而不是保持线程阻塞，等可读取了再读取，次间可以做其它事，所以非阻塞。线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）

NIO主要用到的是块，所以NIO的效率要比IO高很多。在Java API中提供了两套NIO，一套是针对标准输入输出NIO，另一套就是网络编程NIO。

Netty 是一个基于 JAVA NIO 类库的异步通信框架，它的架构特点是：异步非阻塞、基于事件驱动、高性能、高可靠性和高可定制性。

TCP和UDP的socket服务开发，Socket通讯

Gateway使用netty技术接收Tbox的socket连接。

以下是netty的socket Channel定义：

socketChannel.pipeline().addLast(

new IdleStateHandler(idleTime, idleTime, idleTime),

new HexStringEncoder(),

new LengthFieldBasedFrameDecoder(maxFrameLength,

lengthFieldOffset, lengthFieldLength,

lengthAdjustment, initialBytesToStrip),

new OTAResolveServerHandler(tspPlatformClient, kafkaService, streamSender));

其中，idleTime表示是当前socketChannel允许的读写空闲时间，网关目前设定为1分30秒。当创建的socket在设定时间内没有收到任何报文，则断开socket。

报文在传递过程中，可能发生拆包粘包的问题，因此，正确分包才能保障后续报文的处理的准确性。由于OTA协议中定义的报文格式，第三第四个字节为整条报文的长度（见图1），故使用Netty的LengthFieldBasedFrameDecoder(自定义长度解码器)技术，根据长度域来进行报文分包，能够避免粘包和拆包问题。

图1.OTA报文格式

maxFrameLength表示一帧报文最大支持长度，目前设定为2000个字节；

lengthFieldOffset表示长度域的偏移，根据OTA协议，设定为2；

lengthFieldLength表示长度域的字节长度，根据OTA协议，设定为2；

lengthAdjustment表示需要接收的报文长度调整的字节数，由于长度域在第三第四字节，故头部共4个字节需要调整，该值设定为-4；

initialBytesToStrip表示是否从起始字节开始需要跳过一些字节，根据OTA协议，不需要跳过，设定为0。