# 名词

* Group Id：项目的组织机构，也是包的目录结构，一般都是域名的倒序，比如 com.shiyanlou.demo。
* Atifact Id ：项目实际的名字，比如 spring3-Example。
* Packaging ：可在 Group Id 后加 .Atifact Id。
* Version ：项目版本号比如 1.0-SNAPSHOT。
* Bundle ： 捆绑 集成。
* Spring Aop中常用的三个概念
  + Advices ：向程序内部注入的代码,表示一个 method 执行前或执行后的动作。
  + Pointcut ：根据 method 的名字或者正则表达式去拦截一个 method,注入 Advice 的位置，切入点，一般为某方法。
  + Advisor ： Advice 和 Pointcut 组成的独立的单元，并且能够传给 proxy factory 对象，以便将 Advice 和 Pointcut 分开实现灵活配置。

# 笔记：

1. 使用Maven自动构建spring项目，在pom文件中配置核心spring\_core和spring\_context核心组件。Bean之间的关联配置可以实现功能之间的松耦合。
2. 用户在使用 Spring 所提供的各种功能之前， 必须在 Spring IoC 容器中装配好 Bean ，并建立 Bean 和 Bean 之间的关联关系。
3. JavaBean 也允许大写字母起头的属性变量名，不过必须满足: 变量的前两个字母要么全部大写，要么全部小写。
4. 加入<context:component-scan ……>标签自动扫描对应文件夹下被@Component注释的bean，名称默认为类名首字母小写。
5. 在项目中，我们可以将所有自动扫描组件都用 @Component 注释，Spring 将会扫描所有用 @Component 注释过得组件。 实际上，@Repository 、 @Service 、 @Controller 三种注释是为了加强代码的阅读性而创造的，可以在不同的应用层中，用不同的注释。
6. Maven默认使用jdk1.5标准，在Pom文件中添加如下properties修改：

<maven.compiler.source>1.8</maven.compiler.source>  
<maven.compiler.target>1.8</maven.compiler.target>

1. Java中抛出异常之后将不再执行，，若要异常后继续执行下面语句，可以通过try、catch、finally实现。

# Nginx

（静态资源、请求）代理服务器

nginx.cnf主要配置：

upstream tomcat\_server {

server localhost:8080;

}

server {

listen 80;

server\_name localhost;

location ~ .\*.(gif|jpg|jpeg|png|bmp|swf|ioc)$

{ root static/source; }

location ~ .\*\.(js)$ {

root static/js;

}

location ~ .\*\.(css)$ {

root static/css;

}

location / {

proxy\_pass http://tomcat\_server;

}

Springboot向Nginx请求html资源路径调配和返回视图等问题（待研究）

}

# Spring框架

## 1.1 概念：

* **应用程序**：能完成我们所需要功能的成品，比如购物网站、OA 系统。
* **框架**：能完成一定功能的半成品，比如我们可以使用框架进行购物网站开发；框架做一部分功能，我们自己做一部分功能，辅助高效工作。而且框架规定了你在开发应用程序时的整体架构，提供了一些基础功能，还规定了类和对象的如何创建、如何协作等，从而简化我们的代码编写，让我们专注于业务逻辑开发。
* **非侵入式设计**：从框架角度可以这样理解，无需继承框架提供的类，这种设计就可以看作是非侵入式设计，如果继承了这些框架类，就是侵入设计，如果以后想更换框架，之前写过的代码几乎无法重用，如果非侵入式设计则之前写过的代码仍然可以继续使用。
* **轻量级与重量级**：轻量级是相对于重量级而言的，轻量级一般就是非入侵性的、所依赖的东西非常少、资源占用非常少、部署简单等等，其实就是比较容易使用，而重量级正好相反。
* **POJO** ： POJO （ Plain Old Java Objects ）简单的 Java 对象。它可以包含业务逻辑或持久化逻辑，但不担当任何特殊角色且不继承或不实现任何其它 Java 框架的类或接口。
* **容器**：在日常生活中容器就是一种盛放东西的器具，从程序设计角度看就是装对象的的对象，因为存在放入、拿出等操作，所以容器还要管理对象的生命周期。
* **反转控制**：即 Inversion of Control ，缩写为 IoC ，控制反转还有一个名字叫做依赖注入（ Dependency Injection ），就是由容器控制程序之间的关系，而非传统实现中，由程序代码直接操控。
* **Bean** ：一般指容器管理对象，在 Spring 中指 Spring IoC 容器管理对象。

## 1.2 优点：

* **非常轻量级的容器**：以集中的、自动化的方式进行应用程序对象创建和装配，除此之外还会负责管理对象生命周期，能组合成复杂的应用程序。Spring 容器是非侵入式的（不需要依赖任何 Spring 特定类），而且完全采用 POJOs 进行开发，使应用程序更容易测试、更容易管理。而且核心 JAR 包非常小，Spring3.0.5 不到 1 M ，而且不需要依赖任何应用服务器，可以部署在任何环境（ Java SE 或 Java EE ）。
* **AOP**： AOP 是 Aspect Oriented Programming 的缩写，意思是面向切面编程。提供从另一个角度来考虑程序结构以完善面向对象编程（ OOP ），即可以通过在编译期间、装载期间或运行期间实现在不修改源代码的情况下给程序动态添加功能的一种技术。通俗点说就是把可重用的功能提取出来，然后将这些通用功能在合适的时候织入到应用程序中；比如安全，日志记录，这些都是通用的功能，我们可以把它们提取出来，然后在程序执行的合适地方植入这些代码并执行它们，从而完成需要的功能并复用了这些功能。
* **简单的数据库事务管理**：在使用数据库的应用程序当中，自己管理数据库事务是一项很让人头疼的事，而且很容易出现错误，Spring 支持可插入的事务管理支持，而且无需 JavaEE 环境支持，通过 Spring 管理事务可以把我们从事务管理中解放出来来专注业务逻辑。
* **JDBC 抽象及 ORM （对象关系映射）框架支持**： Spring 使 JDBC 更加容易使用；提供 DAO（数据访问对象）支持，非常方便集成第三方 ORM 框架，比如 Hibernate 等；并且完全支持 Spring 事务和使用 Spring 提供的一致的异常体系。
* **灵活的 Web 层支持**： Spring 本身提供一套非常强大的 MVC 框架，而且可以非常容易的与第三方 MVC 框架集成，比如 Struts 等。
* **简化各种技术集成**：提供对 Java Mai 、任务调度、 JMX 、 JMS 、 JNDI 、 EJB 、动态语言、远程访问、 Web Service 等的集成。

## 1.3 IOC的概念

***Ioc—Inversion of Control，***即“控制反转”，它不是什么技术，而是一种设计思想。在 Java 开发中， Ioc 意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。如何理解好 Ioc 呢？理解好 Ioc 的关键是要明确“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”，那我们来深入分析一下：

谁控制谁，控制什么：**传统 Java SE 程序设计，我们直接在对象内部通过 new 进行创建对象，是程序主动去创建依赖对象；而 IoC 是有专门一个容器来创建这些对象，即由 Ioc 容器来控制对象的创建；谁控制谁？当然是 IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。**

为何是反转，哪些方面反转了：有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。**传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。**

其实 IoC 对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在 IoC/DI 思想中，应用程序就变成被动的了，被动的等待 IoC 容器来创建并注入它所需要的资源了。

***DI—Dependency Injection，***即“依赖注入”：是组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解 DI 的关键是：“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，那我们来深入分析一下：

* 谁依赖于谁：当然是某个容器管理对象依赖于 IoC 容器；“被注入对象的对象”依赖于“依赖对象”；
* 为什么需要依赖：容器管理对象需要 IoC 容器来提供对象需要的外部资源；
* 谁注入谁：很明显是 IoC 容器注入某个对象，也就是注入“依赖对象”；
* 注入了什么：就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）。

IoC 和 DI 由什么关系呢？其实它们是同一个概念的不同角度描述，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以 2004 年大师级人物 Martin Fowler 又给出了一个新的名字：“依赖注入”，相对 IoC 而言，“依赖注入”明确描述了“被注入对象依赖 IoC 容器配置依赖对象”。

## 2.1 Spring Bean Scopes – Bean的作用域：

1. singleton — 单例模式（默认），由 IOC 容器返回一个唯一的 bean 实例。
2. prototype — 原型模式，被请求时，每次返回一个新的 bean 实例。
3. request — 每个 HTTP Request 请求返回一个唯一的 Bean 实例。
4. session — 每个 HTTP Session 返回一个唯一的 Bean 实例。
5. globalSession — Http Session 全局 Bean 实例。

## 2.2 Spring Collections – 集合类型的Bean

1. List —— <list/>

<property name ="lists">  
 <list>  
 <value>1</value>  
 <ref bean = "personBean"/>  
 <bean class="com.legolas.learning.innerbean.Person">  
 <property name="name" value="legolas"/>  
 <property name = "address" value = "wuhan"/>  
 <property name ="age" value="25"/>  
 </bean>  
 </list>  
</property>

1. Set —— <set/>

与list类似

1. Map —— <map/>

<property name="maps">  
 <map>  
 <entry key="Key 1" value="1" /><!--一个 entry 就是一个 Map 元素-->  
 <entry key="Key 2" value-ref="personBean" />  
 <entry key="Key 3">  
 <bean class="com.legolas.learning.innerbean.Person">  
 <property name="name" value="shiyanlouMap" />  
 <property name="address" value="chengdu" />  
 <property name="age" value="25" />  
 </bean>  
 </entry>  
 </map>  
</property>

1. Properties —— <props/>

<property name="pros"><!-- Properties 类型类似于Map 类型的特例，Map 元素的键值可以对应任何类型的对象，但是Properties只能是字符串-->  
 <props>  
 <prop key="admin">admin@nospam.com</prop>  
 <prop key="support">support@nospam.com</prop>  
 </props>  
</property>

## 3.1 Spring的注解配置

注解是为 Spring 容器提供 Bean 定义的信息，表现形为把 XML 定义的信息通过类注解描述出来。众所周知，Spring容器三大要素：Bean 定义、 Bean 实现类以及 Spring 框架。如果采用 XML 配置，Bean 定义和 Bean 实现类本身分离，而采用注解配置，Bean 定义在 Bean 实现类上注解就可以实现。

* @Component

被此注解标注的类将被 Spring 容器自动识别，自动生成 Bean 定义

@Component("shiyanlou")

public class shiyanlou{

}

**有四种自动扫描组件的注释类型**

@Component ——表示一个自动扫描 component

@Repository ——表示持久化层的 DAO component

@Service ——表示业务逻辑层的 Service component

@Controller ——表示表示层的 Controller component

* @Autowired

@Autowired 可以用来装配 bean，都可以写在字段上，或者方法上。使用 @Autowired，首先要在在 applicationContext.xml 中加入 <bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>@Autowired 默认按类型装配，默认情况下必须要求依赖对象必须存在，如果要允许 null 值，可以设置它的 required 属性为 false 。例如：

@Autowired()

@Qualifier("shiyanlouDao")

private ShiyanlouDao shiyanlouDao;

* @Configuration

通过使用注释 @Configuration 告诉 Spring ，这个 Class 是 Spring 的核心配置文件，并且通过使用注释 @Bean 定义 bean ，举例说明：

@Configuration

public class AppConfig {

@Bean(name="animal")

public IAnimal getAnimal(){

return new Dog();

}

}

App.java中：

context = new AnnotationConfigApplicationContext(AppConfig.class);

IAnimal obj = (IAnimal) context.getBean("animal");

ApplicationContext.xml中：

<bean id="animal" class="com.lei.demo.java\_config.Dog">

## 4.1 自动扫描中过滤组件

* Filter Component – include(包含该组件)

配置SpringFiltering.xml，通过regex命名规则过滤，只要包含Dao和Service（ Dao.，Service. ）关键字的，都将被检查注册到Spring容器中:

<context:component-scan base-package="com.legolas.springauto" > <context:include-filter type="regex" expression="com.legolas.springauto.dao.\*Dao.\*" />

<context:include-filter type="regex" expression="com.legolas.springauto.service.\*Service.\*" /> </context:component-scan>

* Filter Component – exclude(排除该组件)

以下配置排除用 @Service 注释过的组件：

<context:component-scan base-package=" com.legolas.springauto " >

<context:exclude-filter type="annotation"

expression="org.springframework.stereotype.Service" />

</context:component-scan>

以下配置排除包含 Dao 关键字的组件：

<context:component-scan base-package=" com.legolas.springauto " >

<context:exclude-filter type="regex"

expression=" com.legolas.springauto.dao.\*Dao.\*" />

</context:component-scan>

## 4.2 自动装配 Bean

所谓自动装配，就是将一个 Bean 注入到其他 Bean 的 Property 中，类似于以下：

<bean id="customer" class="com.lei.common.Customer" autowire="byName" />

默认情况下，需要通过 ref 来装配 bean ，如下：

<bean id="customer" class="com.shiyanlou.spring.autowire.common.Customer">

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

</bean>

<bean id="person" class="com.shiyanlou.spring.autowire.common.Person" />

Spring 支持 5 种自动装配模式，如下：

no —— 默认情况下，不自动装配，通过 ref attribute手动设定。

buName —— 根据 Property 的 Name 自动装配，如果一个 bean 的 name ，和另一个 bean 中的 Property 的 name 相同，则自动装配这个 bean 到 Property 中。

byType —— 根据 Property 的数据类型（ Type ）自动装配，如果一个 bean 的数据类型，兼容另一个 bean 中 Property 的数据类型，则自动装配，有两种相同数据类型的 bean 被配置，将抛出 UnsatisfiedDependencyException 异常。

constructor —— 根据构造函数参数的数据类型，进行 byType 模式的自动装配。

autodetect —— 如果发现默认的构造函数，用 constructor 模式，否则，用 byType 模式。

## 5.1 Spring AOP

Spring AOP 即 Aspect-oriented programming，面向切面编程，是作为面向对象编程的一种补充，专门用于处理系统中分布于各个模块（不同方法）中的交叉关注点的问题。简单地说，就是一个拦截器（ interceptor ）拦截一些处理过程。

例如，当一 个method 被执行，Spring AOP 能够劫持正在运行的 method ，在 method 执行前或者后加入一些额外的功能。

在配置文件中加入新的 bean 配置 HijackBeforeMethod ，然后创建一个新的代理（ proxy ）使用 Spring proxy 之前，必须添加 CGLIB2 类库，命名为 customerServiceProxy 。target 定义你想劫持哪个 bean； interceptorNames 定义想用哪个 class ( advice )劫持 target:

<bean **id**="hijackBeforeMethodBean" **class**="com.legolas.springaop.HijackBeforeMethod" />  
  
<bean **id**="customerServiceProxy" **class**="org.springframework.aop.framework.ProxyFactoryBean">  
 <property **name**="target" **ref**="customerService" />  
 <property **name**="interceptorNames">  
 <list>  
 <value>hijackBeforeMethodBean</value>  
 </list>  
 </property>  
</bean>

常用的有四种类型的通知：

* 1. Before Advice 每一个 customerService 的 method 运行前，都将先执行 HijackBeforeMethod 的 before 方法。
  2. After Returning Advice method 运行后，直到返回结果后，才运行HijackAfterMethod的代码，如果没有返回结果，将不运行切入的代码。
  3. Afetr Throwing Advice 目标method运行时，抛出 IllegalArgumentException 异常后，运行切入的方法。
  4. Around Advice 结合了以上 3 种形式的 Advice ，创建一个实现了接口MethodInterceptor 的 class ，你必须通过 methodInvocation.proceed() 来调用原来的方法，即通过调用 methodInvocation.proceed() 来调用 CustomerService 中的每一个方法。也可以不调用原方法。

大多数的 Spring 开发者只用 Around Advice ，因为它能够实现所有类型的 Advice 。

在实际的项目开发中，我们还是要尽量选择适合的 Advice 。

在以上的例子中，CustomerService 中的所有方法都被自动拦截，但是大多数情况下，我们不需要拦截一个 class 中的所有方法，而是拦截符合条件的方法。这时，我们就需要用到 Pointcut and Advice 。

Pointcut & Advisor：

<bean **id**="customerPointcut" **class**="org.springframework.aop.support.NameMatchMethodPointcut">  
<property **name**="mappedName" **value**="printName"/>  
</bean>  
<bean **id**="customerAdvisor"  
**class**="org.springframework.aop.support.DefaultPointcutAdvisor">  
 <property **name**="pointcut" **ref**="customerPointcut" />  
 <property **name**="advice" **ref**="hijackAroundMethodBean" />  
</bean>

Pointcut配置需要拦截的方法，advisor配置将pointcut和advice关联起来。

使用NameMatchMethodPointcutAdvisor可以省去pointcut配置，在advisor中完成。

但是不利于程序的解耦。

Pointcut - Regular exxpression match：

<bean id="customerAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<property name="patterns">

<list>

<value>.\*URL.\*</value>

</list>

</property>

<property name="advice" ref="hijackAroundMethodBean" />

</bean>

使用正则式可以拦截名字中包含 URL 字符的 method 了，在实际工作中，你可以用它来管理 DAO 层，例如，你可以用 .\*DAO.\* 来拦截所有 DAO 层中的相关业务。

## 5.2 自动创建Proxy

方法1：

创建 BeanNameAutoProxyCreator

<bean

class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">

<property name="beanNames">

<list><value>\*Service</value></list>

</property>

<property name="interceptorNames">

<list><value>customerAdvisor</value></list>

</property>

</bean>

以上配置中只要 bean 的 id 符合 \*Service ，就会自动创建 proxy，并与customerAdvisor。

## 6 AspectJ（注解式拦截）

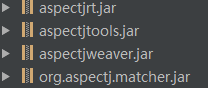
AspectJ 是基于注释（ Annotation ）的，所以需要 JDK5.0 以上的支持。

AspectJ 支持的注释类型如下：

* @Before
* @After
* @AfterReturning
* @AfterThrowing
* @Around

引入4个aspectJ相关jar包，在PointcutsDefinition类中定义切入点，LoggingAspectJ类中定义切入方法。然后配置xml文件：

<aop:aspectj-autoproxy/>  
<bean **id**="customerBo" **class**="com.legolas.springaspectj.CustomerBo"/>  
<bean **id**="logAspect" **class**="com.legolas.springaspectj.LoggingAspectJ" />



<aop:aspectj-autoproxy/> 启动 AspectJ 支持，这样 Spring 会自动寻找用 @Aspect 注释过的类。

# SpringBoot

Springboot使用习惯优于配置，很容易创建一个独立运行（运行jar，内嵌servlet容器）、准生产级别的基于spring框架的项目。Springboot可选择内嵌Tomcat、Jetty、或者Undertow

## RestController&RequestMapping

* RestController是一个组合注解，组合了@Controller和@ResponseBody（返回数据），意味着开发一个和页面交互数据控制时，需要使用此注解，若没有此注解，则需要在自己代码中加入@Controller和@ResponseBody两个注解。
* RequestMapping(value,produces,charset)

Value:访问方法入口

Produces：数据类型：text/plain、application/json

Charset:字符类型UTF-8

## 拦截器配置

可让普通的Bean实现HandlerInterceptor接口或继承HandlerInterceptorAdapter类来实现自定义拦截器。

## 注解

* @SpringBootApplication:
* 是SpringBoot项目的核心注解（源码可见为组合注解），主要组合了@Configuration、@EnableAutoConfiguration、@ComponentScan，主要目的用于开启自动配置，若不使用该配置，可在入口类上直接使用这三个注解。
* Springboot会自动扫描@SpringBootApplication所在类的同级包，以及下级包里的Bean（若为jpa项目还可以扫描注解@Entity的实体类）所以建议入口类放置位置在groupId+arctifactId组合的包名下。
* 关闭特定的自动配置应该使用@SpringBootApplication注解的exclude参数。如：

@SpringBootApplication（exclude = {DataSourceAutoConfiguration.class}）

* @EnableWebMVC

在configuration中添加@EnableWebMvc之后，默认静态路径全部失效。（待研究）

* @EnableScheduling

@Scheduled(fixedRate = 5000)注解用在某方法上创建定时任务，fixedRate是该任务执行周期，@EnableScheduling注解用于启动定时任务。

* @Profile

@Bean

@Profile(“dev/prod”)

为bean在不同环境下使用提供不同的配置支持。

在@Autowired之前使用@ActiveProfiles(“dev/prod”)激活。

## 配置文件

* Springboot使用一个全局的配置文件application.properties或者application.yml放置在resources目录或者类路径的/config下
* SpringBoot提倡零配置，但是在实际项目中如果要求必须的xml配置，可通过Spring提供的@ImportResource来加载。
* Model组件中除了使用@Value注解为属性逐一注入配置，还可以通过@ConfigurationProperties注解来指定属性文件，自动完成注入工作。

//prefix指定注入的属性前缀,locations指定文件位置  
@ConfigurationProperties(prefix = "author",locations = "classpath:properties/author.properties")

* 配置日志，默认情况下，SpringBoot使用Logback作为日志框架：

配置日志级别：logging.file=D:/mylog/log.log

配置日志文件：格式为logging.level.包名=级别（DEBUG、ERROW、INFO、WARN……）

* 通过在配置文件中添加debug=true，来显示启用或不启用哪些自动配置。

## 自定义SpringBoot的自动配置类

SpringBoot实战P159

## Thymeleaf模板引擎

* SpringBoot不支持使用JSP页面，因为JSP在内嵌的Servlet容器上运行不支持以jar形式运行。而Thymeleaf提供了完美的SpringMVC支持。
* 在SpringMVC中若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。
* Thymeleaf为我们定义好了。在SpringMVC中集成Thymeleaf引擎变得十分简单。
* Spring-boot-starter-thymeleaf会自动包含Spring-boot-starter-web。
* Restcontroller会使thymeleaf的解析失效，导致直接返回字符串。使用Controller正常
* 使用Thymeleaf模板时，html中标签规则必须严格按照Thymeleaf要求编写，在实践中因为引入样式文件少写了rel属性，导致样式无法显示。

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

## 自动配置的静态资源

在自动配置类中，addResourceHandlers方法将类路径下的/static、/public、/resources、/META-INF/resources文件夹下的静态文件直接映射为/\*\*可以通过[http://localhost:8080/\*\*](http://localhost:8080/**)直接来访问。

静态首页的支持：把静态index.html放置在上述静态路径下，访问根目录时会自动映射。

webjar的META-INF/resources/webjars下的静态资源映射为/webjar/\*\*。

## 接管SpringBoot的web配置：

* 通过一个配置类@Configuration加上@EnableWebMvc注解来实现完全控制自己的MVC配置。
* 定义一个配置类，并继承WebMvcConfigurerAdaper，可以增加自己额外配置的时候保留Spring的自动配置。（我们的配置和SpringBoot自动配置是同时有效的）。
* 常用配置：
  + 创建InternalResourceViewResolver，自定义defaultViewResolver
  + 添加自定义静态路径：addResourceHandler（不建议，似乎会覆盖默认路径）
  + 设置自定义访问路径addViewControllers

## Tomcat配置

* Tomcat的所有属性都在ServerProperties配置类中做了定义，我们只需在application.properties中配置。
* 通用的Servlet容器配置以“server”为前缀，Tomcat特有配置以“server.tomcat”为前缀。
* 通用配置：继承EmbeddedServletContainerCustomizer接口，重写customize方法。

特定配置：新建配置类，在配置类中新建一个Bean

TomcatEmbeddedServletContainerFactory factory;

factory.addErrorPages(new ErrorPage(HttpStatus.NOT\_FOUND,” /404.html));

factory.addErrorPages(new ErrorPage(HttpStatus.INTERNAL\_SERVER\_ERROR

,” /500.html));

……

factory.setPort(8080);

return factory;

其中，404放在resources下的static下就可以。

* 替换tomcat，依赖改为spring-boot-starter-jetty就可以。

## SSL配置

SSL安全套接层，是为网络通信提供安全及数据完整性的一种安全协议。SSL在网络传输层对网络连接进行加密，位于TCP/IP协议与各种应用层协议之间，SSL协议分为两层：

SSL记录协议：建立在可靠的传输协议之上（如TCP），为高层协议提供数据封装、压缩、加密、等基本功能的支持

SSL握手协议：建立在记录协议之上，用于实际数据传输开始之前，通信双方进行身份认证、协商加密算法、交换加密密钥等。

在基于B/S的web应用中，是通过Https来实现SSL的，Https简单来说就是在Http下加入SSL层，就是Http的安全版。

* 使用SSL首先需要一个证书
  + jdk里有工具keytool（证书管理工具）可以用来生成自签名的证书。命令：

keytool –genkey –alias tomcat -keyalg RSA

-genkey:生成密钥文件 -alias：密钥别名 -keyalg：加密方式

（加密方式不可默认，默认的加密类型访问时被认定为不安全类型）。

* + 将生成的证书放到项目根目录下（是项目的根目录！）下，配置

server.ssl.key-store=.keystore

server.ssl.keystore-password=123456

server.ssl.keyStoreType=JKS

server.ssl.keyAlias=tomcat

* http转向https：需要配置TomcatEmbeddedServletContainerFactory，并添加Tomcat的connector来实现。

## Websocket

Websocket是通过一个socket 双工异步通信，直接使用Websocket或者SockJS协议开发特别繁琐，所以我们使用它的子协议STOMP，它使用一个基于帧frame的格式来定义消息。与Http的request和response类似（有类似与@RequestMapping的@MessageMapping）。

* 导入spring-boot-starter-websocket，html中引入sockjs.js和stomp.js。
* 新建WebsocketConfig配置类，继承AbstractWebSocketMessageBrokerConfigurer，
  + 重写registerStompEndpoints方法，注册STOMP协议节点，映射指定URL，注册一个STOMP的endpoint，并指定使用SockJS协议。
  + 重写configureMessageBroker，广播式配置一个/path消息代理。
  + Controller中使用@MessageMapping(“/path1”)指定访问路径，@SendTo(“/path/path2”)指定返回路径。
  + Html中new SockJS(“/endpoint”)；连接SockJS的endpoint。Stomp.over(socket)使用STOMP子协议，在stompClient.connect中实现stompClient.subscribe(“/path/path2”,function(response){ })方法，订阅服务端发来的消息。stompClient.send(“/path1”,{},JSON.stringfy(json));向服务端发送消息。
* 若要实现点对点传输，首先引入spring-boot-starter-security依赖。
  + 新建WebSecurityConfigy类，继承WebSecurityConfigureAdapter，重写其中的configure方法。配置拦截跳转页面、设置用户……
  + 在WebsocketConfig中注册STOMP的endpoint，添加消息代理
  + Controller中使用提供的MessagingTemplate类的convertAndSendToUser方法，控制消息发送的目标。
  + Html中stomp.subscribe()方法中的接收目标地址，比convertAndSendToUser中配置要多一个/user。

## BootStrap和AngularJS

单页面应用（SPA），在单页面应用中所有资源都是按需动态加载到页面上的，且不需要服务端控制页面转向。

响应式设计（RWD），指不同设备访问相同页面得到不同页面视图。

数据导向：数据导向是对于页面导向而言的，页面上的数据获得是通过消费后台的REST服务来实现的，而不是通过服务器渲染的动态页面（如JSP）来实现的。

AngularJS与jquery的比较：

在jQuery中，你首先设计了一个page，然后再去动态修改它的内容，这是因为jQuery被设计用来进行扩展并在这个前提下大幅度地增加和修改内容，但是在angularjs中，你必须在心中先设计好你的架构，

从一开始，你就要摒弃“我拥有一个DOM元素并且想让它去做某件事”，代之为“我需要完成什么任务，然后接着设计你的应用，最后再去设计你的视图view层”。

AngularJS1.5之后不支持全局路由配置，只能集成配置。配置方法为

angular.module('actionApp', ['ngRoute'])

.controller(

……

)

.controller(

……

)

.config(

//配置路由，并注入$routeProvider来配置。

)

## Spring Data

Spring Data使用的统一API是通过Spring Data Commons项目来实现的，它是各种Spring Data项目的依赖。

Spring Data Commons让我们在使用关系或非关系型数据库时都使用基于Spring的统一标准该标准包括CRUD（创建、获取、更新、删除）、查询、排序、和分页的相关操作。

Spring Data Commons的一个重要概念：Spring Data Repository抽象。它极大地减少了数据访问层的代码。该抽象的根接口是Repository接口。

Repository子接口CrudRepository定义了和CRUD操作相关的内容，CrudRepository的子接口PagingAndRepository定义了分页和排序相关的内容。

不同数据访问技术提供了不同的Repository，如JpaRepository、MongoRepository……

## Spring Data JPA

JPA（Java Persistence API），jpa是一个基于O/R映射的标准规范，所谓标准规范，只是定义标准规范（如注解、接口），不提供实现，软件提供商可以按照标准规范来实现。JPA主要实现由Hibernate、EclipseLink、OpenJPA等。意味着只要使用JPA来开发无论哪一个开发方式都是一样的。

* 定义数据访问层，只需定义一个继承JpaRepositiry的接口。
* 通过@EnableJpaRepositories注解来kaiqiSpring Data JPA的支持。
* 查询方式：
  + 常规查询：根据属性名来定义查询方法：findBy、Like、And、其中findBy可使用find、read、readBy、query、queryBy、get、getBy来替代。
  + 限制结果数量：用top、first关键字来实现：findFirst10ByName（查询符合条件的前10条数据）
  + JPA的@NameQuery定义查询：，即一个名称映射一个查询语句：在@Entity下添加注解@NameQuery（name=”Person.findByName”,query=”select p from Person p where p.name = ?1”） 然后在数据访问层就可以通过findByName调用该定义查询。
  + 使用@Query查询（使用参数索引）：使用@Query（”select p from Person p where p.address=?1”）定义在数据访问层接口方法上，实现查询。findByAddress(String address)
  + 使用@Query查询（使用命名参数）：使用@Query（”select p from Person p where p.address= ：address”） findByAddress(@Param(“address”) String address)。
  + 更新查询：@Modifying和@Query注解组合：

@Modifying

@Transactional

@Query（"update Person p set p.name = ?1”）

Int setName(String name) //返回int表示更新影响的行数

* + Specification，自构造查询条件。
  + 排序与分页

Sort类以及Page接口和Pageable接口。

使用排序：List<Person> people = personRepository.findByName（“xx”,new Sort(Direction.ASC),”age”）；

使用分页：Page<Person> people =personRepository.findByName(“xx”,new PageRequest(0,10));

* @GeneratedValue注解默认使用主键生成方式为自增。，hibernate会为我们自动生成一个名为HIBERNATE\_SEQUENCE的序列。
* 在实体类中我们没有使用@Table注解，则实体类与表名的映射遵循默认规则：如Person——PERSON，TestPerson——TEST\_PERSON；字段没有使用注解@Column，则按照默认对应：属性名name——NAME；多字母：testName——TEST\_NAME。
* 在实体类中的@Id注解、@NamedQuery注解均为javax.persistence包中的类。而非org.springframework.data.annotation.Id;和org.hibernate.annotation。否则会报denied for EntityManagerFactory错误。
* 自定义Repository实现：P274
* SpringDataJPA的模糊查询写法：

,query="select p from Person p where p.name = ?1 and address like CONCAT('%',?2,'%') "

## Spring Data REST

Spring Data Rest 是基于Spring Data的repository之上，可以将repository自动输出为REST资源。

REST服务只需添加spring-boot-starter-data-jpa和starter-data-rest依赖。

默认访问REST资源的路径是在根目录下的，如果我们要定制跟路径：只需在SpringBoot的application.properties下增加spring.data.rest.base-path=/path

访问节点名，是在实体类之后加s来形成路径。如果要自定义映射的名称，在实体类的Repository上使用@RepositoryRestResource（path=”mypath”）。

## Spring的事务机制

Spring提供了一个@EnableTransactionManagement注解在配置类上来开启声明事务的支持。使用该注解后，Spring容器会自动扫描注解@Trasactional的方法和类。

@Transactional可以注解在方法和类上，注解在类上表示该类中的所有public方法都是开启事务的。，同时使用，方法级别的注解会覆盖类级别的注解。

* SimpleJpaRepository的源码中可以看到，Spring DataJPA对所有的默认方法都开启了事务支持，且查询类事务默认启用readOnly=true属性。

## Springboot开启热部署添加的依赖：

<dependency>  
<groupId>org.springframework</groupId>  
<artifactId>springloaded</artifactId>  
</dependency>

## 数据缓存Cache

针对不同的缓存技术，需要实现不同的CacheManager，在使用任意一个实现的CacheManager的时候，需注册实现CacheManager的Bean。

声明式缓存注解：

@Cacheable：在方法执行前先查看缓存中是否有数据，没有则调用方法返回值放入缓存。

@CachePut：无论如何都会将方法的返回值放入缓存。

@CacheEvict：将一条或多条数据从缓存中删除。

@Caching：可以通过Caching注解组合多个注解策略在一个方法上。

这些注解都有value属性，指定缓存名称，key指定数据在缓存中存储的键。

//如果没有指定key，则默认方法参数作为key。

开启声明式缓存的支持：配置类上@EnableCaching

SpringBoot自动配置了多个CacheManager的实现。

默认使用ConCurrentMap来实现数据缓存

在Spring Boot中还是要@EnableCaching开启缓存支持。

## 切换缓存技术

使用EhCache作为缓存技术，只需在pom中添加EhCache依赖：

<groupId>net.sf.ehcache</groupId>

<artifactId>ehcache</artifactId>

Ehcache.xml文件放置在类路径下，Spring Boot会自动扫描

Guava:

<groupId>com.google.guava</groupId>

<artifactId>guava</artifactId>

Redis:

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

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

## Spring Security

Spring security是专门针对spring的项目的安全框架，安全框架的两个重要概念：认证和授权。

Spring security为我们提供了过滤器来实现所有安全的功能，我们只需注册一个特殊的DelegatingFilterProxy过滤器到WebApplicationInitializer即可。

在实际应用中我们只需让自己的Initializer类继承AbstractSecurityWebApplicationInitializer抽象类即可。它实现了WebApplicationInitializer接口并通过onStartup方法调用：

insertSpringSecurityFilterChain（servletContext）；

* SpringSecurity的配置只需在一个配置类上注解@EnableWebSecurity并让这个类继承WebSecurityConfigurerAdapter即可。
* 配置用户认证：重写configure（AuthenticationManagerBuilder auth）
* 内存中的用户：auth

.inMemoryAuthentication()

.withUser("hjh").password("123456").roles("ROLE\_ADMIN")

.and()

.withUser("legolas").password("123456").roles("ROLE\_USER");

* JDBC中的用户：auth

.jdbcAuthentication().dataSource(dataSource)

.usersByUsernameQuery(“select username,password,true from myusers where username=?”)

.authoritiesByUsernameQuery(“select username,role from roles where username=?”);

* 配置请求授权：重写configure（HttpSecurity http）
  + Spring Security使用以下下匹配器来匹配请求路径：
    - antMatchers：使用Ant风格的路径匹配
    - regexMatchers：使用正则表达式匹配路径
    - anyRequest：匹配所有路径

http

.authorizeRequests()

.antMatchers("/admin/\*\*”).hasRole(“ROLE\_ADMIN”)

.antMatcher(“/user/\*\*”).hasAnyRole(“ROLE\_ADMIN”,”ROLE\_USER”)

.anyRequest().authenticated();

只有admin可访问/admin/请求路径

只有user可访问/user/请求路径

其它路径均需认证后（登陆后）才可访问。

## Spring Batch

Spring Batch是用来处理大量数据操作的一个框架，主要用来读取大量数据，然后进行一定处理后输出成指定形式。

## JMS (Java Message Service) java消息服务

Spring对JMS和AMQP（一个消息代理的规范），分别来自于spring-jms和spring-rabbit。它们分别需要ConnectionFactory的实现来连接消息代理，并分别提供了JmsTempleate、RabbitTemplate来发送消息。

Spring为JMS、AMQP提供 了@JmsListner、@RabbitListener注解在方法上监听代理发布的消息。分别需要通过@EnableJms、@EnableRabbit开启支持。

* 支持JMS的实现有ActiveMQ HornetQ…，Spring Boot为我们定义ActiveMQConnectionFactory的Bean作为连接，并通过spring.activemq为前缀的属性来配置ActiveMQ的连接属性。
* 对于AMQP它为我们配置了连接的ConnectionFactory和RabbitTemple且默认开启了注解是消息监听，配置前缀为：spring.rabbitmq

使用activeMQ实现，