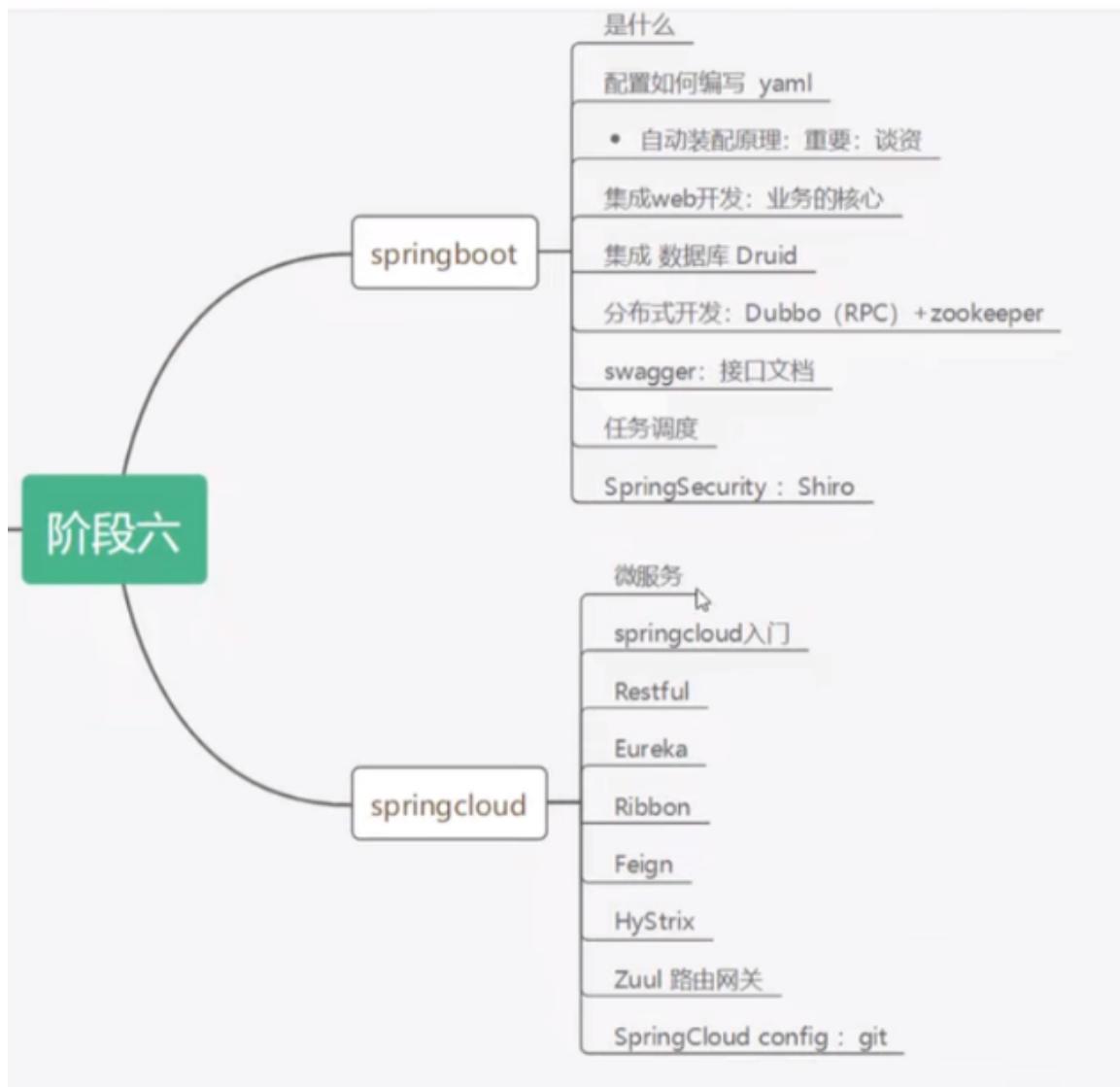


# SpringBoot



## 什么是Spring

Spring是一个开源框架，2003年兴起的一个轻量级的Java开发框架，作者：Rod Johnson。

Spring是为了解决企业级应用开发的复杂性而创建的，简化开发。

## Spring是如何简化Java开发的

为了降低Java开发的复杂性，Spring采用了以下4种关键策略：

- 1、基于POJO的轻量级和最小侵入性编程；
- 2、通过IOC，依赖注入（DI）和面向接口实现松耦合；
- 3、基于切面（AOP）和惯例进行声明式编程；
- 4、通过切面和模版减少样式代码；

## 什么是SpringBoot

学过javaweb的同学就知道，开发一个web应用，从最初开始接触Servlet结合Tomcat，跑出一个Hello World程序，是要经历特别多的步骤；后来就用了框架Struts，再后来是SpringMVC，到了现在的SpringBoot，过一两年又会有其他web框架出现；不知道你们有没经历过框架不断的演进，然后自己开发项目所有的技术也再不断的变化、改造，反正我是都经历过了，哈哈。言归正传，什么是SpringBoot呢，就是一个javaweb的开发框架，和SpringMVC类似，对比其他javaweb框架的好处，官方说是简化开发，约定大于配置，you can "just run"，能迅速的开发web应用，几行代码开发一个http接口。

所有的技术框架的发展似乎都遵循了一条主线规律：从一个复杂应用场景衍生一种规范框架，人们只需要进行各种配置而不需要自己去实现它，这时候强大的配置功能成了优点；发展到一定程度之后，人们根据实际生产应用情况，选取其中实用功能和设计精华，重构出一些轻量级的框架；之后为了提高开发效率，嫌弃原先的各类配置过于麻烦，于是开始提倡“约定大于配置”，进而衍生出一些一站式的解决方案。

是的这就是Java企业级应用->J2EE->spring->springboot的过程。

随着 Spring 不断的发展，涉及的领域越来越多，项目整合开发需要配合各种各样的文件，慢慢变得不那么易用简单，违背了最初的理念，甚至人称配置地狱。Spring Boot 正是在这样的一个背景下被抽象出来的开发框架，目的为了让大家更容易的使用 Spring、更容易的集成各种常用的中间件、开源软件；

Spring Boot 基于 Spring 开发，Spring Boot 本身并不提供 Spring 框架的核心特性以及扩展功能，只是用于快速、敏捷地开发新一代基于 Spring 框架的应用程序。也就是说，它并不是用来替代 Spring 的解决方案，而是和 Spring 框架紧密结合用于提升 Spring 开发者体验的工具。Spring Boot 以**约定大于配置的核心思想**，默认帮我们进行了很多设置，多数 Spring Boot 应用只需要很少的 Spring 配置。同时它集成了大量常用第三方库配置（例如 Redis、MongoDB、Jpa、RabbitMQ、Quartz 等等），Spring Boot 应用中这些第三方库几乎可以零配置的开箱即用，

简单来说就是Spring Boot其实不是什么新的框架，它默认配置了很多框架的使用方式，就像maven整合了所有的jar包，spring boot整合了所有的框架。

Spring Boot 出生名门，从一开始就站在一个比较高的起点，又经过这几年的发展，生态足够完善，Spring Boot 已经当之无愧成为 Java 领域最热门的技术。

Spring Boot的主要优点：

- 为所有Spring开发者更快的入门
- **开箱即用**，提供各种默认配置来简化项目配置
- 内嵌式容器简化Web项目
- 没有冗余代码生成和XML配置的要求

## 微服务

### 什么是微服务？

微服务是一种架构风格，**它要求我们在开发一个应用的时候，这个应用必须构建成一系列小服务的组合；可以通过http的方式进行互通。**要说微服务架构，先得说说过去我们的单体应用架构。

### 单体应用架构

所谓单体应用架构 (**all in one**) 是指，我们将一个应用的中的所有应用服务都封装在一个应用中。

无论是ERP、CRM或是其他什么系统，你都把数据库访问，web访问，等等各个功能放到一个war包内。

- 这样做好处是，易于开发和测试；也十分方便部署；当需要扩展时，只需要将war复制多份，然后放到多个服务器上，再做个负载均衡就可以了。
- 单体应用架构的缺点是，哪怕我要修改一个非常小的地方，我都需要停掉整个服务，重新打包、部署这个应用war包。特别是对于一个大型应用，我们不可能吧所有内容都放在一个应用里面，我们如何维护、如何分工合作都是问题。

### 微服务架构

**all in one**的架构方式，我们把所有的功能单元放在一个应用里面。然后我们把整个应用部署到服务器上。如果负载能力不行，我们将整个应用进行水平复制，进行扩展，然后在负载均衡。

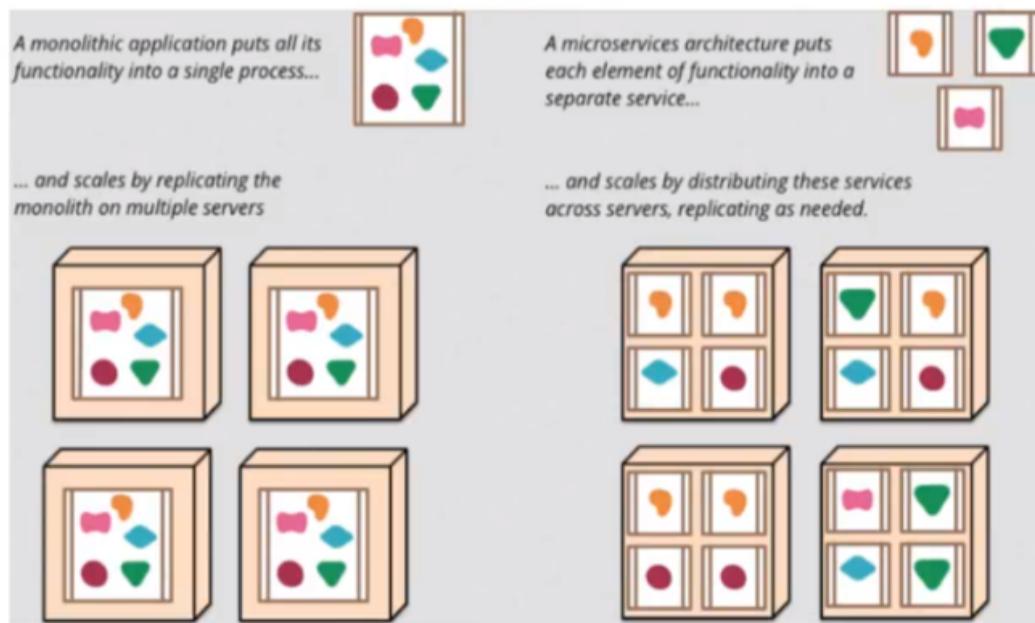
服务器上。如果负载能力不行，我们将整个应用进行水平复制，进行扩展，然后在负载均衡。

[复制](#) [发送文字到手机](#)

所谓微服务架构，**就是打破之前all in one的架构方式**，把每个功能元素独立出来。把独立出来的功能元素的动态组合，需要的功能元素才去拿来组合，需要多一些时可以整合多个功能元素。所以微服务架构是对功能元素进行复制，而没有对整个应用进行复制。

这样做的好处是：

1. 节省了调用资源。
2. 每个功能元素的服务都是一个可替换的、可独立升级的软件代码。



Martin Flower 于 2014 年 3 月 25 日写的《Microservices》，详细的阐述了什么是微服务。

- 原文地址：<http://martinfowler.com/articles/microservices.html>
- 翻译：<https://www.cnblogs.com/liuning8023/p/4493156.html>

## 如何构建微服务

一个大型系统的微服务架构，就像一个复杂交织的神经网络，每一个神经元就是一个功能元素，它们各自完成自己的功能，然后通过http相互请求调用。比如一个电商系统，查缓存、连数据库、浏览页面、结账、支付等服务都是一个个独立的功能服务，都被微化了，它们作为一个个微服务共同构建了一个庞大的系统。如果修改其中的一个功能，只需要更新升级其中一个功能服务单元即可。

但是这种庞大的系统架构给部署和运维带来很大的难度。于是，spring为我们带来了构建大型分布式微服务的全套、全程产品：

- 构建一个个功能独立的微服务应用单元，可以使用springboot，可以帮我们快速构建一个应用；
- 大型分布式网络服务的调用，这部分由spring cloud来完成，实现分布；
- 在分布式中间，进行流式数据计算、批处理，我们有spring cloud data flow。
- spring为我们想清楚了整个从开始构建应用到大型分布式应用全流程方案。



第一个SpringBoot程序：

springboot的主入口

```
package com.example.springboot;
```

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

// 程序的主入口, 是Spring的一个组件
@SpringBootApplication
public class SpringbootApplication {

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

}

```

springboot的pom.xml文件

```

<?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
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.5</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>springboot</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>springboot</name>
  <description>springboot</description>
  <properties>
    <java.version>1.8</java.version>
  </properties>
  <dependencies>
    <!--web依赖: 集成了Tomcat, 配置了DispatcherServlet, xml。。。-->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!--springboot的依赖都是使用spring-boot-starter开头的-->
    <!--单元测试-->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

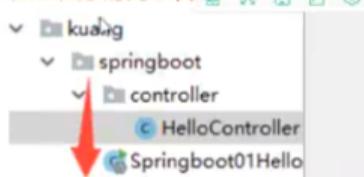
  <build>
    <!--打包jar包插件-->
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>

```

- 项目元数据信息：创建时候输入的Project Metadata部分，也就是Maven项目的基本元素，包括：groupId、artifactId、version、name、description等
- parent：继承spring-boot-starter-parent的依赖管理，控制版本与打包等内容
- dependencies：项目具体依赖，这里包含了spring-boot-starter-web用于实现HTTP接口（该依赖中包含了Spring MVC），官网对它的描述是：使用Spring MVC构建Web（包括RESTful）应用程序的入门者，使用Tomcat作为默认嵌入式容器。；spring-boot-starter-test用于编写单元测试的依赖包。更多功能模块的使用我们将在后面逐步展开。
- build：构建配置部分。默认使用了spring-boot-maven-plugin，配合spring-boot-starter-parent就可以把Spring Boot应用打包成JAR来直接运行。

## 编写HTTP接口

1. 在主程序的同级目录下新建controller包【一定要在同级目录下，否则识别不到】



2. 在包中新建一个Controller类

```
package com.kuang.springboot.controller;

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

@RestController
public class HelloController {

    @RequestMapping("/hello")
    public String hello() {
        return "Hello World";
    }
}
```

3. 编写完毕后，从主程序启动项目，浏览器发起请求，看页面返回：

- 控制台输出了SpringBoot 的 banner
- 控制台输出了 Tomcat 访问的端口号！
- 访问 hello 请求，字符串成功返回！

springboot的重点是自动装配，而自动装配的重点在pom.xml中的spring-boot-starter-parent—>spring-boot-dependencies(负责导入依赖，版本管理，资源导出和插件打包)即springboot的核心依赖在父工程中。我们在引入springboot依赖的时候，不需要指定版本，就是因为父工程spring-boot-dependencies中有一个版本仓库。

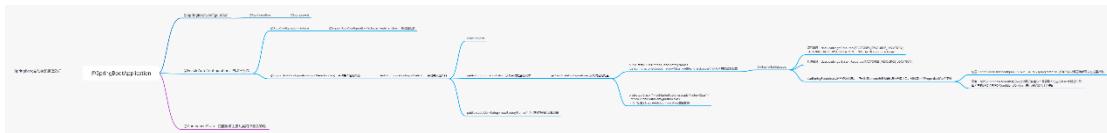
启动器：说白了就是SpringBoot的启动场景，比如spring-starter-web，引入这个启动器，他就会帮我们自动导入web环境所有的依赖。

springboot会将所有的功能场景都变成一个个的启动器。

如果我们需要使用什么功能，只需要找到对应的启动器(spring-starter-)就可以了。

## 主程序中的@SpringBootApplication注解

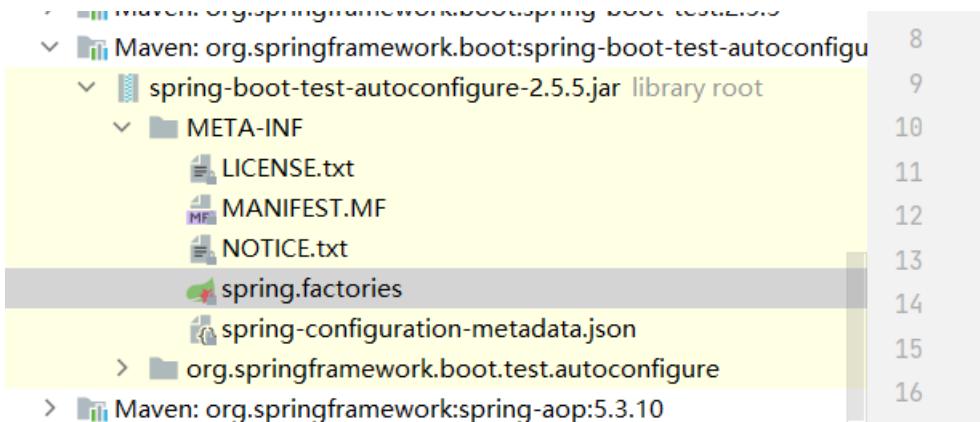
springboot自动装配原理分析：



### • 注解

```
1  @SpringBootConfiguration : springboot的配置
2      @Configuration: spring配置类
3      @Component: 说明这也是一个spring的组件
4
5
6  @EnableAutoConfiguration : 自动配置
7      @AutoConfigurationPackage : 自动配置包
8          @Import(AutoConfigurationPackages.Registrar.class) : 自动配置`包注册`
9          @Import(AutoConfigurationImportSelector.class): 自动配置导入选择
10
11 //获取所有的配置
12 List<String> configurations = getCandidateConfigurations(annotationMetadata,
13 attributes);
14
```

springboot实现自动装配的核心文件：



结论：springboot所有的自动配置都是在启动的时候扫描并加载：spring.factories保存了所有的自动装配类，但是不一定生效，要判断注解@ConditionalOnXXXX中的内容是否生效，并且需要导入了对应的start，就有对应的启动器了，有了启动器，我们的自动装配就会生效，才能配置成功。

1. springboot在启动的时候，会从类路径下的/META-INF/spring.factories获取指定的值；
2. 将这些自动配置的类导入容器，自动配置就会生效，帮我们自动配置；
3. 以前需要我们自动配置的东西，现在springboot帮我们做了

4. 整个JavaEE、解决方案和自动配置的东西都在spring-boot-autoconfigure-2.5.5.jar包下；
5. 它会把我们需要导入的组件，以全限定类名的方式返回，这些组件就会被添加到容器中；
6. 容器中也会存在许多的xxxAutoConfiguration的bean对象，正是这些类给容器导入了这个场景的所有组件，并自动配置；
7. 有了自动配置类，我们就不用手动编写了。

springboot启动类：

7. 有了自动配置类，免去了我们手动编写配置注入功能组件等的工作；

### Run

我最初以为就是运行了一个main方法，没想到却开启了一个服务；

```
@SpringBootApplication
public class SpringbootDemo02Application {

    public static void main(String[] args) {
        //该方法返回一个ConfigurableApplicationContext对象
        //参数一：应用入口的类    参数类：命令行参数
        SpringApplication.run(SpringbootDemo02Application.class, args);
    }
}
```

### SpringApplication.run分析

分析该方法主要分两部分，一部分是SpringApplication的实例化，二是run方法的执行；

### SpringApplication

这个类主要做了以下四件事情

1. 推断应用的类型是普通的项目还是Web项目
2. 查找并加载所有可用初始化器，设置到initializers属性中
3. 找出所有的应用程序监听器，设置到listeners属性中
4. 推断并设置main方法的定义类，找到运行的主类

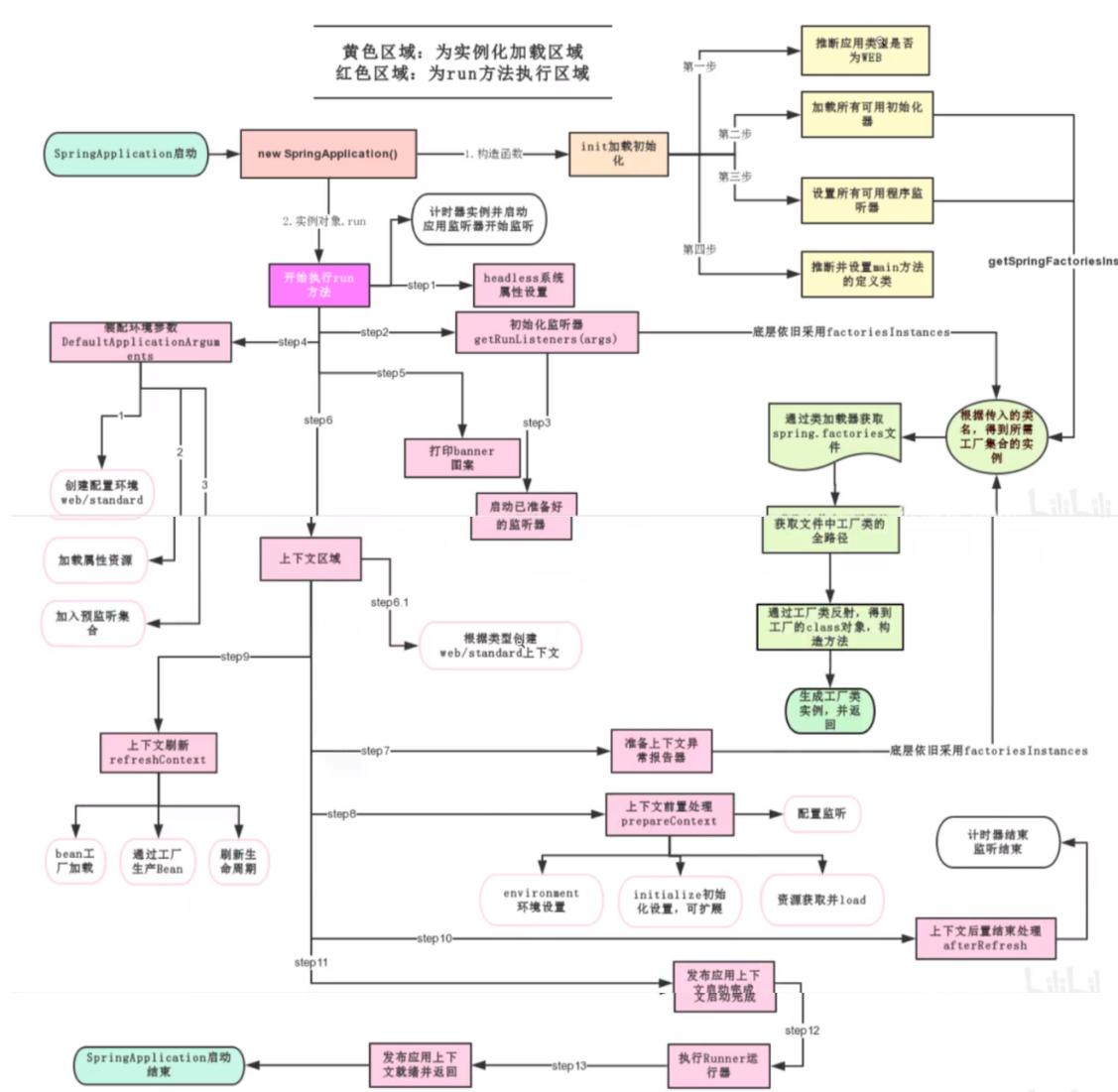
## 查看构造器

```

public SpringApplication(ResourceLoader resourceLoader, Class... primarySources) {
    this.sources = new LinkedHashSet();
    this.bannerMode = Mode.CONSOLE;
    this.logStartupInfo = true;
    this.addCommandLineProperties = true;
    this.addConversionService = true;
    this.headless = true;
    this.registerShutdownHook = true;
    this.additionalProfiles = new HashSet();
    this.isCustomEnvironment = false;
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet(Arrays.asList(primarySources));
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    this.setInitializers(this.getSpringFactoriesInstances(ApplicationContextInitializer.class));
    this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.class));
    this.mainApplicationClass = this.deduceMainApplicationClass();
}

```

## run方法



## SpringBoot：配置文件及自动配置原理

### 配置文件

SpringBoot使用一个全局的配置文件， 配置文件名称是固定的

- application.properties
  - 语法结构： key=value
- application.yml
  - 语法结构： key: 空格 value

**配置文件的作用：**修改SpringBoot自动配置的默认值，因为SpringBoot在底层都给我们自动配置好了；

### YAML

YAML是 "YAML Ain't a Markup Language" (YAML不是一种置标语言) 的递归缩写。

在开发的这种语言时， YAML 的意思其实是："Yet Another Markup Language" (仍是一种置标语言)

YAML A Markup Language : 是一个标记语言

YAML isnot Markup Language : 不是一个标记语言

**标记语言**

### YAML语法

**基础语法：**

k:(空格) v

以此来表示一对键值对（空格不能省略）；以空格的缩进来控制层级关系，只要是左边对齐的一列数据都是同一个层级的。

注意：属性和值的大小写都是十分敏感的。例子：

```
server:  
  port: 8081  
  path: /hello
```

### 值的写法

**字面量：普通的值 [ 数字，布尔值，字符串 ]**

k: v

字面量直接写在后面就可以， 字符串默认不用加上双引号或者单引号；

"" 双引号，不会转义字符串里面的特殊字符， 特殊字符会作为本身想表示的意思；

比如： name: "kuang \n shen" 输出： kuang 换行 shen

以前的配置文件，大多数都是使用xml来配置；比如一个简单的端口配置，我们来对比下yaml和xml

yaml配置：

```
server:  
    port: 8080
```

xml配置：

```
<server>  
    <port>8081</port>  
</server>
```

Springboot程序实现：

```
<!--导入配置文件处理器-->  
<!--<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-configuration-processor</artifactId>  
    <optional>true</optional>  
</dependency>-->
```

在springboot的主程序的同级目录下建包，只有这样，主程序才会对这些类生效，才能扫描到这些类里面的组件；

编写实体类：

```
package com.example.pojo;  
  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import lombok.ToString;  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.stereotype.Component;  
  
import java.util.Date;  
import java.util.List;  
import java.util.Map;  
  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@ToString  
@Component  
@ConfigurationProperties(prefix = "person")  
/**  
 * @ConfigurationProperties注解的作用：将配置文件中配置的每一个属性的值，映射到这个  
组件中；  
 * 告诉springboot将本类中的所有属性和配置文件中相关的配置进行绑定  
 * 参数 prefix="person"；将配置文件中的person与下面的所有属性一一对应。  
 * 只有这个组件是容器中的组件，才能使用容器提供的@ConfigurationProperties功能。  
 */  
public class Person {  
  
    private String name;  
    private Integer age;  
    private Boolean happy;  
    private Date birth;
```

```

private Map<String, Object> maps;
private List<Object> lists;
private Dog dog;

}

@Data
@NoArgsConstructor
@AllArgsConstructor
@ToString @Component
public class Dog {
    @Value("夯实")
    private String name;
    @Value("3")
    private Integer age; }

```

若要使用properties配置文件可能导入时存在乱码现象，需要IDEA中进行调整，我们这里直接使用yml文件，将默认的application.properties后缀修改为yml。

yml可以直接给实体类赋值，通过配置文件application.yaml实现属性注入：

编写配置文件application.yaml

```

# yaml可以注入到我们的配置类中
# yaml对空格的要求极高
# 普通的key: value
name: 张三

# 对象
student:
  name: liu
  age: 18

# 行内写法
stu: {name: wang, age: 3}

# 数组
pets:
  - cat
  - dog
  - pig

pet: [cat, dog, pig]

# yaml可以注入到我们的配置类中

person:
  name: 张三${random.uuid}
  age: ${random.int(100)}
  happy: false
  birth: 2021/12/02
  maps: {k1: v1, k2: v2}
  lists: [wang, xiang, tai]
  hello: xiao
  dog:
    name: ${person.hello:hello}_夯实 # ${person.hello:hello}: 若person值不存在就取默认值hello
    age: 3

```

编写测试类：

```

@SpringBootTest
class SpringbootApplicationTests {
    @Autowired
    private Person person;

```

```

    @Test
    void contextLoads() {
        System.out.println(person); // 输出: Person(name=Bob, age=46, happy=false,
        // birth=Thu Dec 02 00:00:00 CST 2021,
        // maps={k1=v1, k2=v2},
        // lists=[wang, xiang, tai],
        // dog=Dog{name='xiao_秀昊', age=3})
    }
}

```

配置文件除了yml还有我们之前常用的properties，我们没有讲，properties配置文件在写中文的时候，会有乱码，我们需要去IDEA中设置编码格式为UTF-8；

settings-->FileEncodings中配置；



还有，我们的类和配置文件直接关联着，我们使用的是@configurationProperties的方式，还有一种方式是使用@value

```

    @Component //注册bean
public class Person {
    //直接使用@value
    @Value("${person.name}") //从配置文件中取值
    private String name;
    @Value("#{11*2}") //#{SPEL} Spring表达式
    private Integer age;
    @Value("true") //字面量
    private Boolean happy;
}

```

这个使用起来并不友好！我们需要为每个属性单独注解赋值，比较麻烦；我们来看个功能对比图

	@ConfigurationProperties	@Value
功能	批量注入配置文件中的属性	一个个指定
松散绑定（松散语法）	支持	不支持
SpEL	不支持	支持
JSR303数据校验	支持	不支持
复杂类型封装	支持	不支持

- cp只需要写一次即可，value则需要每个字段都添加
- 松散绑定：这个什么意思呢？比如我的yml中写的last-name，这个和lastName是一样的，-后面跟着的字母默认是大写的。这就是松散绑定
- JSR303数据校验，这个就是我们在字段上增加一层过滤器验证，可以保证数据的合法性
- 复杂类型封装，yml中可以封装对象，使用@value就不支持

结论：

- 配置yml和配置properties都可以获取到值，强烈推荐yml
- 如果我们在某个业务中，只需要获取配置文件中的某个值，可以使用一下@value
- 如果说，我们专门编写了一个JavaBean来和配置文件进行映射，就直接使用@configurationProperties，不要犹豫！

## JSR-303校验：

JSR-303 是JAVA EE 6 中的一项子规范，叫做Bean Validation，Hibernate Validator 是 Bean Validation 的参考实现。Hibernate Validator 提供了 JSR 303 规范中所有内置 constraint 的实现，除此之外还有一些附加的 constraint。

导入依赖：

```
<!-- JSR303校验 -->
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-validator -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.16.Final</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-validator-
annotation-processor -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator-annotation-processor</artifactId>
    <version>6.0.16.Final</version>
</dependency>
<!-- https://mvnrepository.com/artifact/com.fasterxml/classmate -->
<dependency>
    <groupId>com.fasterxml</groupId>
    <artifactId>classmate</artifactId>
    <version>1.5.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.jboss.logging/jboss-logging -->
<dependency>
    <groupId>org.jboss.logging</groupId>
    <artifactId>jboss-logging</artifactId>
    <version>3.4.0.Final</version>
</dependency>
```

### Bean Validation 中内置的 constraint

空检查 @Null 验证对象是否为null @NotNull 验证对象是否不为null，无法查检长度为0的字符串 @NotBlank 检查约束字符串是不是Null还有被Trim的长度是否大于0, 只对字符串, 且会去掉前后空格。 @NotEmpty 检查约束元素是否为NULL或者是EMPTY. Boolean检查 @AssertTrue 验证 Boolean 对象是否为 true @AssertFalse 验证 Boolean 对象是否为 false 长度检查 @Size(min=, max=) 验证对象 (Array, Collection, Map, String) 长度是否在给定的范围之内 @Length(min=, max=) Validates that the annotated string is between min and max included. 日期检查 @Past 验证 Date 和 Calendar 对象是否在当前时间之前，验证成立的话被注释的元素一定是一个过去的日期 @Future 验证 Date 和 Calendar 对象是否在当前时间之后，验证成立的话被注释的元素一定是一个将来的日期 @Pattern 验证 String 对象是否符合正则表达式的规则，被注释的元素符合制定的正则表达式, regexp: 正则表达式 flags: 指定 Pattern.Flag 的数组，表示正则表达式的相关选项。数值检查 建议使用在String, Integer类型，不建议使用在int类型上，因为表单值为“”时无法转换为int，但可以转换为String为“”，Integer为null @Min 验证 Number 和 String 对象是否大等于指定的值 @Max 验证 Number 和 String 对象是否小等于指定的值 @DecimalMax 被标注的值必须不大于约束中指定的最大值. 这个约束的参数是一个通过 BigDecimal 定义的最大值的字符串表示. 小数存在精度 @DecimalMin 被标注的值必须不小于约束中指定的最小值. 这个约束的参数是一个通过 BigDecimal 定义的最小值的字符串表示. 小数存在精度 @Digits 验证 Number 和 String 的构成是否合法 @Digits(integer=, fraction=)

验证字符串是否是符合指定格式的数字，interger指定整数精度，fraction指定小数精度。

@Range(min=, max=) 被指定的元素必须在合适的范围内

@Range(min=10000, max=50000, message=" range. bean. wage" ) @Valid 递归的对关联对象进行校验，如果关联对象是个集合或者数组，那么对其中的元素进行递归校验，如果是一个map，则对其中的值部分进行校验。(是否进行递归验证) @CreditCardNumber信用卡验证 @Email 验证是否是邮件地址，如果为null, 不进行验证，算通过验证。 @ScriptAssert(lang=, script=, alias=) @URL(protocol=, host=, port=, regexp=, flags=)

Hibernate Validator 附加的 constraint

Constraint	详细信息
@Email	被注释的元素必须是电子邮箱地址
@Length	被注释的字符串的大小必须在指定的范围内
@NotEmpty	被注释的字符串的必须非空
@Range	被注释的元素必须在合适的范围内

## 多环境配置：

### 多环境切换

profile是Spring对不同环境提供不同配置功能的支持，可以通过激活不同的环境版本，实现快速切换环境；

#### 方式一：多配置文件

我们在主配置文件编写的时候，文件名可以是 application-{profile}.properties/yml，用来指定多个环境版本；

例如：application-test.properties 代表测试环境配置 application-dev.properties 代表开发环境配置

但是Springboot并不会直接启动这些配置文件，它默认使用application.properties主配置文件；

我们需要通过一个配置来选择需要激活的环境；

#比如在配置文件中指定使用dev环境，我们可以通过设置不同的端口号进行测试；

#我们启动SpringBoot，就可以看到已经切换到dev下的配置了；

spring.profiles.active=dev

可以创建application.yaml配置文件的位置，file是项目目录，classpath是resource目录，生效优先级为1 > 2 > 3 > 4

**注意：如果yml和properties同时都配置了端口，并且没有激活其他环境， 默认会使用properties配置文件的！**

## 配置文件加载位置

springboot 启动会扫描以下位置的application.properties或者application.yml文件作为Spring boot的默认配置文件

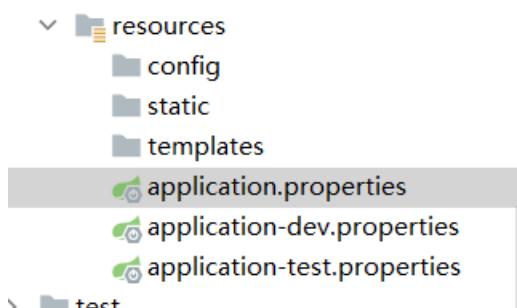
- 优先级1：项目路径下的config文件夹配置文件
- 优先级2：项目路径下配置文件
- 优先级3：资源路径下的config文件夹配置文件
- 优先级4：资源路径下配置文件

优先级由高到底，高优先级的配置会覆盖低优先级的配置；

**SpringBoot会从这四个位置全部加载主配置文件；互补配置；**

properties文件实现多环境配置，在application.properties中编写：profile是“配置”的意思

```
# 这是默认使用的配置环境  
# springboot的多环境配置，可以选择激活那一个配置文件  
spring.profiles.active=dev
```



yaml文件实现多环境配置：

### 方式二：yml的多文档块

和properties配置文件中一样，但是使用yml去实现不需要创建多个配置文件，更加方便了

```
server:  
  port: 8081  
#选择要激活那个环境块  
spring:  
  profiles:  
    active: prod  
  
---  
  
server:  
  port: 8083  
#配置环境的名称  
spring:  
  profiles: dev  
  
# 默认使用第一套配置，可以在第一套配置里设置激活使用那一套配置  
server:  
  port: 8081
```

```
spring:  
  profiles:  
    active: dev  
---  
server:  
  port: 8082  
spring:  
  profiles: dev  
---  
server:  
  port: 8083  
spring:  
  profiles: test
```

我们还可以通过spring.config.location来改变默认的配置文件位置

项目打包好以后，我们可以使用命令行参数的形式，启动项目的时候来指定配置文件的新位置；这种情况，一般是后期运维做的多，相同配置，外部指定的配置文件优先级最高

```
java -jar spring-boot-config.jar --spring.config.location=F:/application.properties
```

外部加载配置文件的方式十分多，我们选择最常用的即可，在开发的资源文件中进行配置！

@Conditional派生注解：

@Conditional扩展注解	作用（判断是否满足当前指定条件）
@ConditionalOnJava	系统的java版本是否符合要求
@ConditionalOnBean	容器中存在指定Bean；
@ConditionalOnMissingBean	容器中不存在指定Bean；
@ConditionalOnExpression	满足SpEL表达式指定
@ConditionalOnClass	系统中有指定的类
@ConditionalOnMissingClass	系统中没有指定的类
@ConditionalOnSingleCandidate	容器中只有一个指定的Bean，或者这个Bean是首选Bean
@ConditionalOnProperty	系统中指定的属性是否有指定的值
@ConditionalOnResource	类路径下是否存在指定资源文件
@ConditionalOnWebApplication	当前是web环境
@ConditionalOnNotWebApplication	当前不是web环境
@ConditionalOnJndi	JNDI存在指定项

作用：必须是@Conditional指定的条件成立，才给容器中添加组件，配置里面的所有内容才生效。

@EnableAutoConfiguration实现的关键在于引入了AutoConfigurationImportSelector，其核心逻辑为selectImports方法，借助AutoConfigurationImportSelector，它可以帮助SpringBoot应用将所有符合条件的@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器。

每一个这样的 xxxAutoConfiguration类都是容器中的一个组件，最后都加入到容器中；用他们来做自动配置；

3. 每一个自动配置类可以进行自动配置功能；

4. 我们以**HttpEncodingAutoConfiguration** (Http编码自动配置) 为例解释自动配置原理；

**一句话总结：根据当前不同的条件判断，决定这个配置类是否生效！**

一旦这个配置类生效；这个配置类就会给容器中添加各种组件；这些组件的属性是从对应的 properties类中获取的，**这些类里面的每一个属性又是和配置文件绑定的**；

5. 所有在配置文件中能配置的属性都是在xxxxProperties类中封装者’；配置文件能配置什么就可以参照某个功能对应的这个属性类

**精髓：**

1) 、SpringBoot启动会加载大量的自动配置类

2) 、我们看我们需要的功能有没有在SpringBoot默认写好的自动配置类当中；

3) 、我们再来看这个自动配置类中到底配置了哪些组件；（只要我们要用的组件存在在其中，我们就不需要再手动配置了）

4) 、给容器中自动配置类添加组件的时候，会从properties类中获取某些属性。我们只需要在配置文件中指定这些属性的值即可；

**xxxxAutoConfigurartion:** 自动配置类；给容器中添加组件

**xxxxProperties:** 封装配置文件中相关属性；

那么多的自动配置类，必须在一定的条件下才能生效；也就是说，我们加载了这么多的配置类，但不是所有的都生效了。

我们怎么知道哪些自动配置类生效；我们可以通过启用 **debug=true**属性；来让控制台打印自动配置报告，这样我们就可以很方便的知道哪些自动配置类生效；

```
#开启springboot的调试类  
debug=true
```

**Positive matches:** (自动配置类启用的：正匹配)

**Negative matches:** (没有启动，没有匹配成功的自动配置类：负匹配)

**Unconditional classes:** (没有条件的类)

输出的日志我们可以在这里看下：

```
■ 控制台打印日志
```

```
@SpringBootApplication -> AutoConfigurationImportSelector -> selectImports ->  
getAutoConfigurationEntry -> getCandidateConfigurations ->  
SpringFactoriesLoader.loadFactoryNames(EnableAutoConfiguration, beanClassLoader) ->  
loadSpringFactories->url:ClassLoader.getResources("META-INF/spring.factories")  
url:spring.factories -> xxxAutoConfiguration <-  
@EnableConfigurationProperties(xxxProperties.class) <- xxxProperties <-
```

@ConfigurationProperties(prefix="xxx.xxx") 装配我们在application.yaml中写的配置

```
getSpringFactoriesLoaderFactoryClass() -> EnableAutoConfiguration  
getBeanClassLoader() -> beanClassLoader
```

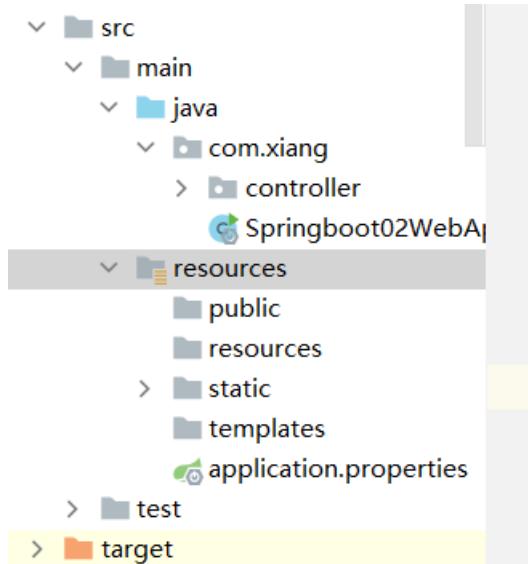
要解决的问题：

- 导入静态资源.....
- 首页！
- jsp， 模板引擎 Thymeleaf
- 装配扩展SpringMVC
- 增删改查
- 拦截器
- 国际化！

## 静态资源导入探究：

从WebMvcAutoConfiguration.java中的addResourceHandlers方法中可以看到静态资源导入的原理。

只要是“/\*\*”的url请求都会去：1.”classpath:/META-INF/resources/“；2.”classpath:/resources/“；3.”classpath:/static/“；4.”classpath:/public/“ 目录下去找静态资源。



图片中的资源目录resources就是target文件中的classpath。templates文件夹只能通过controller来跳转访问。

访问优先级依次是：resources > static > public

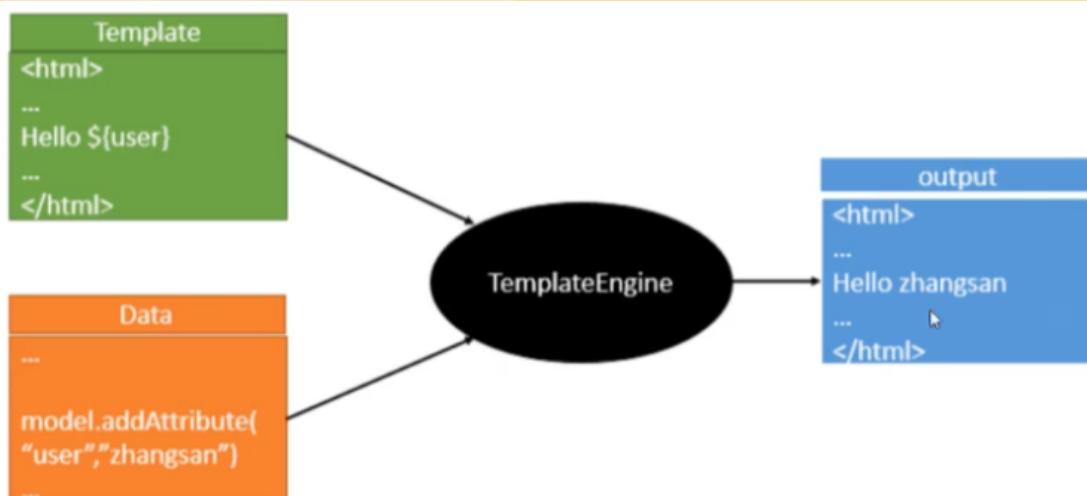
总结：在springboot中，我们可以使用以下方式处理静态资源：

1. web jars
2. public, static, /\*\*, resources

前端交给我们的页面，是html页面。如果我们以前开发，我们需要把他们转成jsp页面，jsp好处就是当我们查出一些数据转发到JSP页面以后，我们可以用jsp轻松实现数据的显示，及交互等。jsp支持非常强大的功能，包括能写Java代码，但是呢，我们现在的这种情况，SpringBoot这个项目首先是以jar的方式，不是war，像第二，我们用的还是嵌入式的Tomcat，所以呢，他现在默认是不支持jsp的。

那不支持jsp，如果我们直接用纯静态页面的方式，那给我们开发会带来非常大的麻烦，那怎么办呢，SpringBoot推荐你可以来使用模板引擎。

那么这模板引擎，我们其实大家听到很多，其实jsp就是一个模板引擎，还有以用的比较多的freemarker，包括SpringBoot给我们推荐的Thymeleaf，模板引擎有非常多，但再多的模板引擎，他们的思想都是一样的，什么样一个思想呢我们来看一下这张图。



模板引擎的作用就是我们来写一个页面模板，比如有些值呢，是动态的，我们写一些表达式。而这些值，从哪来呢，我们来组装一些数据，我们把这些数据找到。然后把这个模板和这个数据交给我们模板引擎，模板引擎按照我们这个数据帮你把这表达式解析、填充到我们指定的位置，然后把这个数据最终生成一个我们想要的内容给我们写出去，这就是我们这个模板引擎，不管是jsp还是其他模板引擎，都是这个思想。只不过呢，就是说不同模板引擎之间，他们可能这个语法有点不一样。其他的我就不介绍了，我主要来介绍一下SpringBoot给我们推荐的Thymeleaf模板引擎，这模板引擎呢，是一个高级语言的模板引擎，他的这个语法更简单。而且呢，功能更强大。

我们呢，就来看一下这个模板引擎，那既然要看这个模板引擎。首先，我们来看SpringBoot里边怎么用。

第一步：引入thymeleaf，怎么引入呢，对于springboot来说，什么事情不都是一个start的事情嘛，我们去在项目中引入一下。给大家三个网址：

- 1、Thymeleaf 官网：<https://www.thymeleaf.org/>
- 2、Thymeleaf 在Github 的主页：<https://github.com/thymeleaf/thymeleaf>
- 3、Spring官方文档：“<https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/htmlsingle/#using-boot-starter>” , 找到我们对应的版本

要使用模板引擎需要导入依赖：

```
<!--Thymeleaf, 我们都是基于3.x开发的-->
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring5</artifactId>
</dependency>
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-java8time</artifactId>
</dependency>
```

如果springboot使用的是1.x版本，就导入2.x版本的thymeleaf

Thymeleaf的使用：

- Simple expressions:
  - Variable Expressions: \${...}
  - Selection Variable Expressions: \*{...}
  - Message Expressions: #{...}
  - Link URL Expressions: @{...}
  - Fragment Expressions: ~{...}
- Literals
  - Text literals: 'one text' , 'Another one!' ,...
  - Number literals: 0 , 34 , 3.0 , 12.3 ,...
  - Boolean literals: true , false
  - Null literal: null
  - Literal tokens: one , sometext , main ,...
- Text operations:
  - String concatenation: +
  - Literal substitutions: |The name is \${name}|
- Arithmetic operations:
  - Binary operators: + , - , \* , / , %
  - Minus sign (unary operator): -

- Boolean operations:
    - Binary operators: and , or
    - Boolean negation (unary operator): ! , not
  - Comparisons and equality:
    - Comparators: > , < , >= , <= ( gt , lt , ge , le )
    - Equality operators: == , != ( eq , ne )
  - Conditional operators:
    - If-then: (if) ? (then)
    - If-then-else: (if) ? (then) : (else)
    - Default: (value) ?: (defaultvalue)
  - Special tokens:
- 

- No-Operation: \_

Order	Feature	Attributes
1	Fragment inclusion [jsp: include]	th:insert th:replace
2	Fragment iteration	th:each
3	Conditional evaluation	th:if th:unless th:switch th:case
4	Local variable definition	th:object th:with
5	General attribute modification	th:attr th:attrprepend th:attrappend
6	Specific attribute modification	th:value th:href th:src ...
7	Text (tag body modification)	th:text th:utext
8	Fragment specification	th:fragment
9	Fragment removal	th:remove

扩展springmvc自动装配

The auto-configuration adds the following features on top of Spring's defaults:

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (covered later in this document).
- Automatic registration of `Converter`, `GenericConverter`, and `Formatter` beans.
- Support for `HttpMessageConverters` (covered later in this document).
- Automatic registration of `MessageCodesResolver` (covered later in this document).
- Static `index.html` support.
- Custom `Favicon` support (covered later in this document).
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered later in this document).

If you want to keep Spring Boot MVC features and you want to add additional MVC configuration (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but without `@EnableWebMvc`. If you wish to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, you can declare a `WebMvcRegistrationsAdapter` instance to provide such components.

If you want to take complete control of Spring MVC, you can add your own `@Configuration` annotated with `@EnableWebMvc`.

## 1. 首页配置：

1. 注意点，所有页面的静态资源都需要使用thymeleaf接管；
2. url: @{}

## 2. 页面国际化：

1. 我们需要配置 i18n文件
2. 我们如果需要在项目中进行按钮自动切换，我们需要自定义一个组件 `LocaleResolver`
3. 记得将自己写的组件配置到 spring容器 `@Bean`
4. #{}

## 3. 登录 + 拦截器

## 4. 员工列表展示

### 1. 提取公共页面

1. `th:fragment="sidebar"`
2. `th:replace="~{commons/commons::topbar}"`
3. 如果要传递参数，可以直接使用 () 传参，接收判断即可！

### 2.

```
# 1. 前端搞定： 页面长什么样子： 数据  
# 2. 设计数据库（数据库设计难点！）  
# 3. 前端让他能够自动运行， 独立化工程  
# 4. 数据接口如何对接： json， 对象 all in one !  
# 5. 前后端联调测试！
```

- 1. 有一套自己熟悉的后台模板：工作必要！x-admin
- 2. 前端界面：至少自己能够通过前端框架，组合出来一个网站页面
  - index
  - about
  - blog
  - post
  - user
- 3. 让这个网站能够独立运行！

一个月！

## 上周回顾

- SpringBoot是什么？
- 微服务
- HelloWorld~
- 探究源码~ 自动装配原理~
- 配置 yaml
- 多文档环境切换
- 静态资源映射
- Thymeleaf th:xxx
- SpringBoot 如何扩展MVC javaconfig~
- 如何修改SpringBoot的默认配置~
- CRUD
- 国际化
- 拦截器
- 定制首页，错误页~

这周：

- JDBC
- Mybatis：重点
- Druid：重点
- Shiro：安全：重点
- Spring Security：安全：重点
- 异步任务~，邮件发送，定时任务
- Swagger
- Dubbo + Zookeeper

## SpringBoot整合JDBC：

### 简介

对于数据访问层，无论是 SQL(关系型数据库) 还是 NOSQL(非关系型数据库)，Spring Boot 底层都是采用 Spring Data 的方式进行统一处理。

Spring Boot 底层都是采用 Spring Data 的方式进行统一处理各种数据库，Spring Data 也是 Spring 中与 Spring Boot、Spring Cloud 等齐名的知名项目。

Spring Data 官网：<https://spring.io/projects/spring-data>

数据库相关的启动器：可以参考官方文档：<https://docs.spring.io/spring-boot/docs/2.1.7.RELEASE/reference/htmlsingle/#using-boot-starter>

pom.xml

```
<?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
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
```

```

<version>2.5.5</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.xiang</groupId>
<artifactId>springboot-03-data</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>springboot-03-data</name>
<description>Demo project for Spring Boot</description>
<properties>
    <java.version>1.8</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>

```

application.yml:

```

spring:
  datasource:
    username: root
    password: 123456
    # 若因时区报错，就增加一个时区的配置 &serverTimezone=UTC
    url: jdbc:mysql://localhost:3306/mybatis?useUnicode=true&characterEncoding=utf8
    driver-class-name: com.mysql.cj.jdbc.Driver

```

测试：

```

package com.xiang;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

```

```

@SpringBootTest
class Springboot03DataApplicationTests {

    @Autowired
    DataSource dataSource;

    @Test
    void contextLoads() throws SQLException {
        // 查看默认数据源: com.zaxxer.hikari.HikariDataSource
        System.out.println(dataSource.getClass());
        // 获得数据库连接
        Connection connection = dataSource.getConnection();
        System.out.println(connection);

        // 关闭
        connection.close();
    }
}

```

JdbcController:

```

package com.xiang.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.Map;

@RestController
public class JdbcController {

    @Autowired
    JdbcTemplate jdbcTemplate;

    // 查询数据库中的所有信息，在没有实体类的情况下，可以使用Map获取数据
    @RequestMapping("userList")
    public List<Map<String, Object>> userList() {
        String sql = "select * from user";
        List<Map<String, Object>> maps = jdbcTemplate.queryForList(sql);
        return maps;
    }

    @GetMapping("/addUser")
    public String addUser() { // 不需要手动提交事务了
        String sql = "insert into mybatis.user (id, name, pwd) values(4, '李四', '456')";
        jdbcTemplate.update(sql);
        return "add-success";
    }
}

```

```

@GetMapping("/updateUser/{id}")
public String updateUser(@PathVariable int id) {
    String sql = "update mybatis.user set name=?, pwd=? where id=" + id;
    //封装
    Object[] objects = new Object[2];
    objects[0] = "李4";
    objects[1] = "654";
    jdbcTemplate.update(sql, objects);
    return "update-success";
}

@GetMapping("/deleteUser/{id}")
public String deleteUser(@PathVariable("id") int id) {
    String sql = "delete from mybatis.user where id = ?";
    jdbcTemplate.update(sql, id);
    return "delete-success";
}

}

```

可以看出 Spring Boot 2.1.7 默认使用 com.zaxxer.hikari.HikariDataSource 数据源，而以前版本，如 Spring Boot 1.5 默认使用 org.apache.tomcat.jdbc.pool.DataSource 作为数据源；

HikariDataSource 号称 Java WEB 当前速度最快的数据源，相比于传统的 C3P0、DBCP、Tomcat jdbc 等连接池更加优秀；

## 自定义数据源 DruidDataSource

### DRUID 简介

Druid 是阿里巴巴开源平台上一个数据库连接池实现，结合了 C3P0、DBCP、PROXOOL 等 DB 池的优点，同时加入了日志监控。

Druid 可以很好的监控 DB 池连接和 SQL 的执行情况，天生就是针对监控而生的 DB 连接池。

Spring Boot 2.0 以上默认使用 Hikari 数据源，可以说 Hikari 与 Druid 都是当前 Java Web 上最优秀的数据源，我们来重点介绍 Spring Boot 如何集成 Druid 数据源，如何实现数据库监控。

DruidDataSource 配置兼容 DBCP，但个别配置的语意有所区别。

**Druid参数配置：**

**name** : 配置这个属性的意义在于，如果存在多个数据源，监控的时候可以通过名字来区分开来。如果没有配置，将会生成一个名字，格式是：“DataSource-” + System.identityHashCode(this). 另外配置此属性至少在 1.0.5 版本中是不起作用的，强行设置 name 会出错。

**url**: 连接数据库的 url，不同数据库不一样。例如：

mysql : jdbc:mysql://10.20.153.104:3306/druid2

oracle : jdbc:oracle:thin:@10.20.149.85:1521:ocnauto

**username**: 连接数据库的用户名

**password**:连接数据库的密码。如果你不希望密码直接写在配置文件中，可以使用ConfigFilter

**driverClassName**: 缺省值-根据url自动识别 这一项可配可不配，如果不配置druid会根据url自动识别dbType，然后选择相应的driverClassName

**initialSize**: 缺省值-0 初始化时建立物理连接的个数。初始化发生在显示调用init方法，或者第一次getConnection时

**maxActive**: 缺省值- 8 最大连接池数量

**maxIdle**: 缺省值- 8 已经不再使用，配置了也没有效果的

**minIdle**: 最小连接池数量

**maxWait**:获取连接时最大等待时间，单位毫秒。配置了maxWait之后，缺省启用公平锁，并发效率会有所下降，如果需要可以通过配置useUnfairLock属性为true使用非公平锁。

**poolPreparedStatements**: 缺省值-false 是否缓存preparedStatement，也就是PSCache。PSCache对支持游标的数据库性能提升巨大，比如说oracle。在mysql下建议关闭。

**maxPoolPreparedStatementPerConnectionSize**: 缺省值- -1 要启用PSCache，必须配置大于0，当大于0时，poolPreparedStatements自动触发修改为true。在Druid中，不会存在Oracle下PSCache占用内存过多的问题，可以把这个数值配置大一些，比如说100

**validationQuery**:用来检测连接是否有效的sql，要求是一个查询语句，常用select 'x'。如果validationQuery为null，testOnBorrow、testOnReturn、testWhileIdle都不会起作用

**validationQueryTimeout**:单位：秒，检测连接是否有效的超时时间。底层调用jdbc Statement对象的void setQueryTimeout(int seconds)方法

**testOnBorrow**: 缺省值-true 申请连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能。

**testOnReturn**: 缺省值-false 归还连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能。

**testWhileIdle**:缺省值-false 建议配置为true，不影响性能，并且保证安全性。申请连接的时候检测，如果空闲时间大于

timeBetweenEvictionRunsMillis，执行validationQuery检测连接是否有效。

keepAlive:缺省值-false(1.0.28版本)连接池中的minIdle数量以内的连接，空闲时间超过minEvictableIdleTimeMillis(**最小可驱逐空闲时间**毫秒)，则会执行keepAlive操作。

timeBetweenEvictionRunsMillis: 缺省值- 1分钟 (1.0.14) 有两个含义：

1) Destroy线程会检测连接的间隔时间，如果连接空闲时间大于等于minEvictableIdleTimeMillis则关闭物理连接。

2) testWhileIdle的判断依据，详细看testWhileIdle属性的说明

numTestsPerEvictionRun:缺省值- 30分钟 不再使用，一个DruidDataSource只支持一个EvictionRun

minEvictableIdleTimeMillis: 连接保持空闲而不被驱逐的最长时间

connectionInitSqls: 物理连接初始化的时候执行的sql

exceptionSorter:缺省值-根据dbtype自动识别 当数据库抛出一些不可恢复的异常时，抛弃连接

filters: 属性类型是字符串，通过别名的方式配置扩展插件，常用的插件有：

监控统计用的filter:stat

日志用的filter:log4j

防御sql注入的filter:wall

proxyFilters:类型是List<com.alibaba.druid.filter.Filter>，如果同时配置了filters和proxyFilters，是组合关系，并非替换关系

## 引入数据源

第一步需要在应用的 pom.xml 文件中添加上 Druid 数据源依赖，而这个依赖可以从 Maven 仓库官网 [Maven Repository](#) 中获取



```
<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.12</version>
</dependency>
```

## Druid:

导入依赖:

```
<properties>
    <java.version>1.8</java.version>
    <log4j2.version>2.15.0</log4j2.version>
</properties>
<dependencies>      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<!-- druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.8</version>
</dependency>

<!-- log4j-->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

在换成druid数据源之前测试spring默认的数据源[查看默认数据源](#):

`com.zaxxer.hikari.HikariDataSource`

在test文件夹下:

```
package com.xiang;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

@SpringBootTest
class Springboot03DataApplicationTests {

    @Autowired
    DataSource dataSource;

    @Test
    void contextLoads() throws SQLException {
        // 查看默认数据源: com.zaxxer.hikari.HikariDataSource
        // 配置自己的数据源: com.alibaba.druid.pool.DruidDataSource
        System.out.println(dataSource.getClass());

        // 获得数据库连接
        Connection connection = dataSource.getConnection();
        System.out.println(connection);

        // xxxTemplate: 是SpringBoot已经配置好的模板bean, 拿来即用
        // 关闭
        connection.close();
    }
}
```

在application.yml中连接数据库:

```
spring:
  # 数据源配置
  datasource:
    username: root
    password: 123456
    # 若因时区报错, 就增加一个时区的配置 &serverTimezone=UTC
    url: jdbc:mysql://localhost:3306/mybatis?useUnicode=true&characterEncoding=utf8
    driver-class-name: com.mysql.cj.jdbc.Driver
```

controller文件夹下:

```
package com.xiang.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.Map;
```

```

@RestController
public class JdbcController {

    @Autowired
    JdbcTemplate jdbcTemplate;

    // 查询数据库中的所有信息，在没有实体类的情况下，可以使用Map获取数据
    @RequestMapping("userList")
    public List<Map<String, Object>> userList() {
        String sql = "select * from user";
        List<Map<String, Object>> maps = jdbcTemplate.queryForList(sql);
        return maps;
    }

    @GetMapping("/addUser")
    public String addUser() { // 不需要手动提交事务了
        String sql = "insert into mybatis.user (id, name, pwd) values(4, '李
四', '456')";
        jdbcTemplate.update(sql);
        return "add-success";
    }

    @GetMapping("/updateUser/{id}")
    public String updateUser(@PathVariable int id) {
        String sql = "update mybatis.user set name=?, pwd=? where id=" + id;
        // 封装
        Object[] objects = new Object[2];
        objects[0] = "李四";
        objects[1] = "654";
        jdbcTemplate.update(sql, objects);
        return "update-success";
    }

    @GetMapping("/deleteUser/{id}")
    public String deleteUser(@PathVariable("id") int id) {
        String sql = "delete from mybatis.user where id = ?";
        jdbcTemplate.update(sql, id);
        return "delete-success";
    }
}

```

切换成druid，并在application.yml中配置druid

```

spring:
  # 数据源配置
  datasource:
    username: root
    password: 123456
    # 若因时区报错，就增加一个时区的配置 &serverTimezone=UTC
    url: jdbc:mysql://localhost:3306/mybatis?useUnicode=true&characterEncoding=utf8
    driver-class-name: com.mysql.cj.jdbc.Driver

    # SpringBoot默认是不注入这些属性值的，需要自己绑定
    # druid数据源专有配置
    type: com.alibaba.druid.pool.DruidDataSource
    # 连接池配置
    druid:
      # 初始化大小，最小，最大
      initialSize: 5
      minIdle: 5
      maxActive: 20
      # 配置获取连接等待超时的时间

```

```

maxWait: 60000
# 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位毫秒
timeBetweenEvictionRunsMillis: 60000
# 配置一个连接在池中最小生存时间
minEvictableIdleTimeMillis: 300000
validationQuery: SELECT 1 FROM DUAL
testWhileIdle: true
testOnBorrow: false
testOnReturn: false
# 打开 PSCache，并且指定每个连接上 PSCache 的大小
poolPreparedStatements: true
# max-pool-prepared-statement-per-connection-size: 20

# 配置监控统计拦截的filters, stat: 监控统计、log4j: 日志记录、wall: 防御sql
注入
# 如果运行时报错 java.lang.ClassNotFoundException: org.apache.log4j.Priority
# 则导入log4j依赖即可, Maven地址:
https://mvnrepository.com/artifact/log4j/log4j
filters: stat, wall, log4j
maxPoolPreparedStatementPerConnectionSize: 20
useGlobalDataSourceStat: true
connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500

# 通过 connection-properties 属性打开 mergeSql 功能; 慢 SQL 记录
connection-properties:
druid.stat.mergeSql|=true;druid.stat.slowSqlMillis|=5000

#以下配置已在DruidConfig文件中实现
# # 配置 DruidStatFilter
# web-stat-filter:
#   enabled: true
#   url-pattern: /*
#   exclusions: *.js, *.gif, *.jpg, *.bmp, *.png, *.css, *.ico, /druid/*
# # 配置 DruidStatViewServlet
# stat-view-servlet:
#   url-pattern: /druid/*
#   # IP 白名单, 没有配置或者为空, 则允许所有访问
#   allow: 127.0.0.1
#   # IP 黑名单, 若白名单也存在, 则优先使用
#   deny: 192.168.31.253
#   # 禁用 HTML 中 Reset All 按钮
#   reset-enable: false
#   # 登录用户名/密码
#   login-username: root
#   login-password: 123456
#

```

在config文件夹下编写DruidConfig配置类

```

package com.xiang.config;

import com.alibaba.druid.pool.DruidDataSource;
import com.alibaba.druid.support.http.StatViewServlet;
import com.alibaba.druid.support.http.WebStatFilter;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.boot.web.servlet.ServletRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

@Configuration
public class DruidConfig {

```

```

@ConfigurationProperties(prefix = "spring.datasource")
@Bean
public DataSource druidDataSource() {
    return new DruidDataSource();
}

// 后台监控
@Bean // 因为SpringBoot内置了servlet容器，所以没有web.xml，所以使用
ServletRegistrationBean替代
public ServletRegistrationBean StatViewServlet() {
    ServletRegistrationBean<StatViewServlet> bean = new
ServletRegistrationBean<>(new StatViewServlet(), "/druid/*");
    // 后台需要有人登录，账号密码配置
    HashMap<String, String> initParameters = new HashMap<>();

    // 增加配置
    initParameters.put("loginUsername", "admin"); // 登录key是固定的
    loginUsername, loginPassword
    initParameters.put("loginPassword", "123456");

    // 允许谁能访问
    initParameters.put("allow", "");

    // 禁止谁访问
    initParameters.put("kuangshen", "192.168.11.23");

    bean.setInitParameters(initParameters); // 设置初始化参数
    return bean;
}

// filter
@Bean
public FilterRegistrationBean webStatFilter() {

    FilterRegistrationBean filterBean = new FilterRegistrationBean();
    filterBean.setFilter(new WebStatFilter());

    // 过滤哪些请求
    Map<String, String> initParameters = new HashMap<>();
    // 这些东西不进行统计
    initParameters.put("exclusions", "*.*.js,*.*.css,/druid/*");
    filterBean.setInitParameters(initParameters);
    return filterBean;
}
}

```

用浏览器访问localhost:8080/druid即可查看druid监控的日志信息

quartz: 任务调度

stat: (监控) 统计

## SpringBoot整合Mybatis:

导入依赖:

```

<dependency>
    <groupId>org.projectlombok</groupId>

```

```
<artifactId>lombok</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

在pojo目录下编写实体类：

```
package com.xiang.pojo;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {

    private int id;
    private String name;
    private String pwd;

}
```

在application.yml中配置数据源和mybatis：

```
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.url=jdbc:mysql://localhost:3306/mybatis?
serverTimezone=UTC&useUnicode=true&characterEncoding=utf8
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# 整合mybatis
mybatis.type-aliases-package=com.xiang.pojo
mybatis.mapper-locationsclasspath:mybatis/mapper/*.xml
```

在mapper目录下编写接口：

```
package com.xiang.mapper;

import com.xiang.pojo.User;
import org.apache.ibatis.annotations.Mapper;
import org.springframework.stereotype.Repository;

import java.util.List;

@Mapper
@Repository
public interface UserMapper {

    List<User> queryUserList();
```

```

User queryUserById(int id);

int addUser(User user);

int updateUser(User user);

int deleteUser(int id);

}

```

在resource/mybatis/mapper目录下编写UserMapper.xml文件：

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xiang.mapper.UserMapper">

    <select id="queryUserList" resultType="user">
        select * from user
    </select>

    <select id="queryUserById" resultType="user">
        select *
        from user
        where id=#{id};
    </select>

    <insert id="addUser" parameterType="user">
        insert into user (id, name, pwd) values (#{id}, #{name}, #{pwd})
    </insert>

    <update id="updateUser" parameterType="user">
        update user set name=#{name}, pwd=#{pwd} where id = #{id}
    </update>

    <delete id="deleteUser" parameterType="int">
        delete from user where id = #{id}
    </delete>

</mapper>

```

在controller目录下编写UserController：

```

package com.xiang.controller;

import com.xiang.mapper.UserMapper;
import com.xiang.pojo.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
public class UserController {

    @Autowired
    private UserMapper userMapper;

    @GetMapping("/queryUserList")

```

```

public List<User> queryUserList() {
    List<User> users = userMapper.queryUserList();
    for (User user : users) {
        System.out.println(user);
    }
    return users;
}

@GetMapping("/queryUserById/{id}")
public User queryUserById(@PathVariable("id") Integer id) {
    User user = userMapper.queryUserById(id);
    return user;
}

@GetMapping("/addUser{id} {name} {pwd}")
public int addUser(@RequestParam("id") Integer id, @RequestParam("name") String name, @RequestParam("pwd") String pwd) {
    int i = userMapper.addUser(new User(id, name, pwd));
    return i;
}

@GetMapping("/updateUser/{id} /{name} /{pwd}")
public int updateUser(@PathVariable("id") Integer id, @PathVariable("name") String name, @PathVariable("pwd") String pwd) {
    int i = userMapper.updateUser(new User(id, name, pwd));
    return i;
}

@GetMapping("/deleteUser/{id}")
public int deleteUser(@PathVariable("id") Integer id) {
    int i = userMapper.deleteUser(id);
    return i;
}
}

```

启动主程序访问链接看数据库，测试效果

## SpringBoot Security:

## 简介

Spring Security 是针对Spring项目的安全框架，也是Spring Boot底层安全模块默认的技术选型，他可以实现强大的Web安全控制，对于安全控制，我们仅需要引入 spring-boot-starter-security 模块，进行少量的配置，即可实现强大的安全管理！

记住几个类：

- WebSecurityConfigurerAdapter：自定义Security策略
- AuthenticationManagerBuilder：自定义认证策略
- @EnableWebSecurity：开启WebSecurity模式， @Enablexxxx 开启某个功能

Spring Security的两个主要目标是“认证”和“授权”（访问控制）。

“认证”（Authentication）

“授权”（Authorization）

这个概念是通用的，而不是只在Spring Security 中存在。

参考官网：<https://spring.io/projects/spring-security>， 查看我们自己项目中的版本，找到对应的帮助文档：

<https://docs.spring.io/spring-security/site/docs/5.2.0.RELEASE/reference/htmlsingle>

导入依赖：

```
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity5</artifactId>
    <version>3.0.4.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring5</artifactId>
</dependency>

<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-java8time</artifactId>
</dependency>
```

关闭thymeleaf

# 关闭模板引擎thymeleaf的缓存，方便随时调试

spring.thymeleaf.cache=false

在controller目录下编写网站路由：

```
package com.xiang.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class RouterController {

    @RequestMapping({"/", "/index", "index.html"})
    public String index() {
        return "index";
    }

    @RequestMapping("/toLogin")
    public String toLogin() {
        return "views/login";
    }

    @RequestMapping("/level1/{id}")
    public String level1(@PathVariable("id") int id) {
        return "views/level1/" + id;
    }

    @RequestMapping("/level2/{id}")
    public String level2(@PathVariable("id") int id) {
        return "views/level2/" + id;
    }

    @RequestMapping("/level3/{id}")
    public String level3(@PathVariable("id") int id) {
        return "views/level3/" + id;
    }
}
```

在config目录下编写security配置类

```
package com.xiang.config;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@EnableWebSecurity // 开启WebSecurity模式
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    // 授权
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // 首页所有人可以访问，功能页只有拥有权限的人，才能访问
        // 请求授权的规则，链式编程
        http.authorizeRequests()
            .antMatchers("/").permitAll()
            .antMatchers("/level1/**").hasRole("vip1")
    }
}
```

```

        .antMatchers("/level2/**").hasRole("vip2")
        .antMatchers("/level3/**").hasRole("vip3");
    // 若没有权限，则默认会跳到登录页面，需要开启登录的页面 loginPage设置登录页面 loginProcessingUrl设置登录表单提交页面 usernameParameter接收name=user的表单输入作为username,
    http.formLogin().loginPage("/toLogin").usernameParameter("user").passwordParameter("pwd").loginProcessingUrl("/login"); // 定制登录页

    // 防止网站攻击，最好通过post发送登出请求，登出失败可能存在的原因：使用get请求登出
    http.csrf().disable(); // 关闭csrf(跨站请求伪造)功能

    // 注销，开启了注销功能
    // http.logout().deleteCookies("remove").invalidateHttpSession(true);
    http.logout().logoutSuccessUrl("/");

    // 开启记住账号功能，保存cookie，默认保存两周 rememberMeParameter自定义接收前端参数
    http.rememberMe().rememberMeParameter("remember");
}

// 认证
// 在Spring Security 5.0+ 新增了很多的加密方法
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    // auth.jdbcAuthentication() 这些数据正常应该从数据库中读取 passwordEncoder
    // 密码编码 new BCryptPasswordEncoder() 设置密码加密方式
    auth.inMemoryAuthentication().passwordEncoder(new BCryptPasswordEncoder())
        .withUser("admin").password(new BCryptPasswordEncoder().encode("123456")).roles("vip1", "vip2", "vip3")
        .and()
        .withUser("xiang").password(new BCryptPasswordEncoder().encode("123456")).roles("vip2", "vip3")
        .and()
        .withUser("guest").password(new BCryptPasswordEncoder().encode("123456")).roles("vip1");
}
}

```

## 9.2 JDBC Authentication

You can find the updates to support JDBC based authentication. The example below assumes that you have already defined a `DataSource` within your application. The `jdbc-javaconfig` sample provides a complete example of using JDBC based authentication.

```

@.Autowired
private DataSource dataSource;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    // ensure the passwords are encoded properly
    UserBuilder users = User.withDefaultPasswordEncoder();
    auth
        .jdbcAuthentication()
        .dataSource(dataSource)
        .withDefaultSchema()
        .withUser(users.username("user").password("password").roles("USER"))
        .withUser(users.username("admin").password("password").roles("USER", "ADMIN"));
}

```

## 首页index.html

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">
    <title>首页</title>
    <!--semantic-ui-->

```

```

<link href="https://cdn.bootcss.com/semantic-ui/2.4.1/semantic.min.css"
rel="stylesheet">
<link th:href="@{/qinjiang/css/qinstyle.css}" rel="stylesheet">
</head>
</body>

<!--主容器--&gt;
&lt;div class="ui container"&gt;

&lt;div class="ui segment" id="index-header-nav" th:fragment="nav-menu"&gt;
    &lt;div class="ui secondary menu"&gt;
        &lt;a class="item" th:href="@{/index}"&gt;首页&lt;/a&gt;

        &lt;!--登录注销--&gt;
        &lt;div class="right menu"&gt;
            &lt;!--如果用户未登录, 显示登录图标--&gt;
            &lt;div sec:authorize="!isAuthenticated()"&gt;
                &lt;a class="item" th:href="@{/toLogin}"&gt;
                    &lt;i class="address card icon"&gt;&lt;/i&gt; 登录
                &lt;/a&gt;
            &lt;/div&gt;
            &lt;!--如果已登录, 显示用户名和用户权限--&gt;
            &lt;div sec:authorize="isAuthenticated()"&gt;
                &lt;a class="item"&gt;
                    用户名: &lt;span sec:authentication="name"&gt;&lt;/span&gt; &lt;!--获得用户名--&gt; &ampnbsp;
                    角色: &lt;span sec:authentication="principal authorities"&gt;
                &lt;/span&gt; &lt;!--获得用户权限 principal当前用户 (当事人)--&gt;
                &lt;/a&gt;
            &lt;/div&gt;

            &lt;!--如果已登录, 显示注销图标--&gt;
            &lt;!--注销 @{/logout}由Spring security提供, 让用户登出, 并invalidating session--&gt;
            &lt;div sec:authorize="isAuthenticated()"&gt;
                &lt;a class="item" th:href="@{/logout}"&gt;
                    &lt;i class="sign-out icon"&gt;&lt;/i&gt; 注销
                &lt;/a&gt;
            &lt;/div&gt;
            &lt;!--已登录
            &lt;a th:href="@{/toUserCenter}"&gt;
                &lt;i class="address card icon"&gt;&lt;/i&gt; admin
            &lt;/a&gt;
            --&gt;
        &lt;/div&gt;
    &lt;/div&gt;
&lt;/div&gt;

&lt;div class="ui segment" style="text-align: center"&gt;
    &lt;h3&gt;Spring Security Study&lt;/h3&gt;
&lt;/div&gt;

&lt;div&gt;
    &lt;br&gt;
    &lt;div class="ui three column stackable grid"&gt;
        &lt;!--根据不同用户显示不同板块, 若登录的用户有vip1的权限就会显示, 否则隐藏(不显示)--&gt;
        &lt;div class="column" sec:authorize="hasRole('vip1')"&gt;
            &lt;div class="ui raised segment"&gt;
                &lt;div class="ui"&gt;
                    &lt;div class="content"&gt;
                        &lt;h5 class="content"&gt;Level 1&lt;/h5&gt;
                        &lt;hr&gt;
</pre>

```

```

</i> Level-1-1</a></div>
</i> Level-1-2</a></div>
</i> Level-1-3</a></div>
</div>
</div>
</div>
</div>

<div class="column" sec:authorize="hasRole(' vip2' )">
    <div class="ui raised segment">
        <div class="ui">
            <div class="content">
                <h5 class="content">Level 2</h5>
                <hr>
                <div><a th:href="@{/level2/1}"><i class="bullhorn icon">
</i> Level-2-1</a></div>
</i> Level-2-2</a></div>
</i> Level-2-3</a></div>
                </div>
            </div>
        </div>
    </div>
</div>

<div class="column" sec:authorize="hasRole(' vip3' )">
    <div class="ui raised segment">
        <div class="ui">
            <div class="content">
                <h5 class="content">Level 3</h5>
                <hr>
                <div><a th:href="@{/level3/1}"><i class="bullhorn icon">
</i> Level-3-1</a></div>
                <div><a th:href="@{/level3/2}"><i class="bullhorn icon">
</i> Level-3-2</a></div>
                <div><a th:href="@{/level3/3}"><i class="bullhorn icon">
</i> Level-3-3</a></div>
                </div>
            </div>
        </div>
    </div>
</div>

</div>
</div>

<script th:src="@{/xiang/js/jquery-3.1.1.min.js}"></script>
<script th:src="@{/xiang/js/semantic.min.js}"></script>

```

编写login.html:

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, maximum-
scale=1">

```

```
<title>登录</title>
<!--semantic-ui-->
<link href="https://cdn.bootcss.com/semantic-ui/2.4.1/semantic.min.css" rel="stylesheet">
</head>
<body>

<!--主容器-->
<div class="ui container">

    <div class="ui segment">

        <div style="text-align: center">
            <h1 class="header">登录</h1>
        </div>

        <div class="ui placeholder segment">
            <div class="ui column very relaxed stackable grid">
                <div class="column">
                    <div class="ui form">
                        <form th:action="@{/login}" method="post">
                            <div class="field">
                                <label>Username</label>
                                <div class="ui left icon input">
                                    <input type="text" placeholder="Username" name="user">
                                    <i class="user icon"></i>
                                </div>
                            </div>
                            <div class="field">
                                <label>Password</label>
                                <div class="ui left icon input">
                                    <input type="password" name="pwd">
                                    <i class="lock icon"></i>
                                </div>
                            </div>
                            <div class="field">
                                <input type="checkbox" name="remember"> 保存账号
                            </div>
                            <input type="submit" class="ui blue submit button"/>
                        </form>
                    </div>
                </div>
            </div>
        </div>
    </div>

    <div style="text-align: center">
        <div class="ui label">
            <i>注册</i>
        </div>
        <br><br>
        <small>blog.baidu.com</small>
    </div>
    <div class="ui segment" style="text-align: center">
        <h3>Spring Security</h3>
    </div>
</div>

</div>

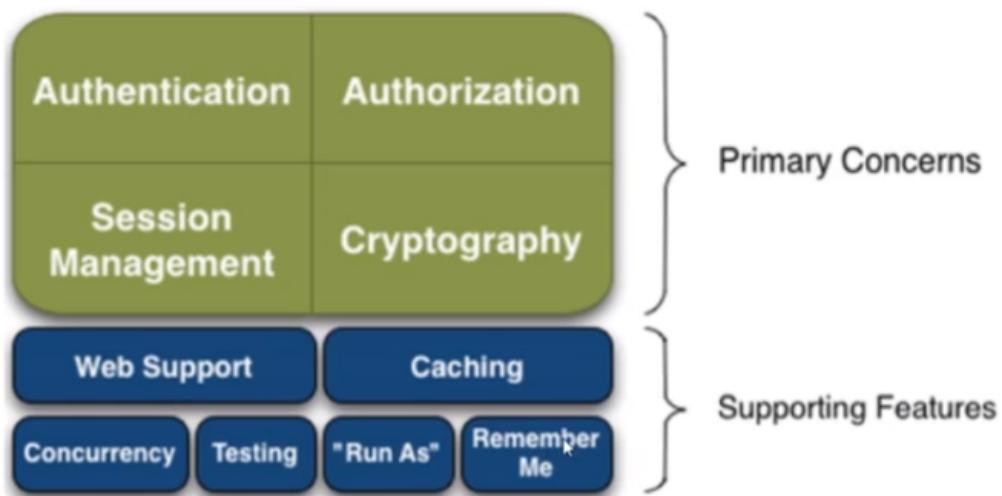
<script th:src="@{/xiang/js/jquery-3.1.1.min.js}"></script>
<script th:src="@{/xiang/js/semantic.min.js}"></script>
```

```
</body>  
</html>
```

## 1.1、什么是Shiro?

- Apache Shiro 是一个Java 的安全（权限）框架。
- Shiro 可以非常容易的开发出足够好的应用，其不仅可以用在JavaSE环境，也可以用在JavaEE环境。
- Shiro可以完成，认证，授权，加密，会话管理，Web集成，缓存等。
- 下载地址：<http://shiro.apache.org/>

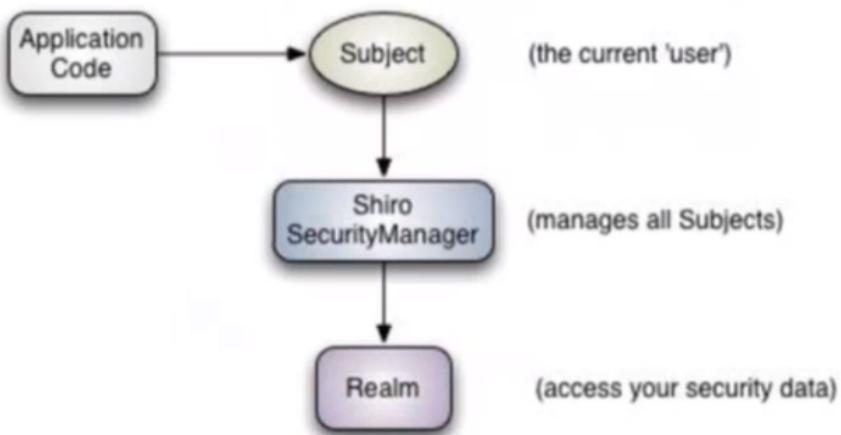
## 1.2、有哪些功能？



- Authentication: 身份认证、登录，验证用户是不是拥有相应的身份；
- Authorization: 授权，即权限验证，验证某个已认证的用户是否拥有某个权限，即判断用户能否进行什么操作，如：验证某个用户是否拥有某个角色，或者细粒度的验证某个用户对某个资源是否具有某个权限！
- Session Manager: 会话管理，即用户登录后就是第一次会话，在没有退出之前，它的所有信息都在会话中；会话可以是普通的JavaSE环境，也可以是Web环境；
- Cryptography: 加密，保护数据的安全性，如密码加密存储到数据库中，而不是明文存储；
- Web Support: Web支持，可以非常容易的集成到Web环境；
- Caching: 缓存，比如用户登录后，其用户信息，拥有的角色、权限不必每次去查，这样可以提高效率
- Concurrency: Shiro支持多线程应用的并发验证，即，如在一个线程中开启另一个线程，能把权限自动的传播过去
- Testing: 提供测试支持；
- Run As: 允许一个用户假装为另一个用户（如果他们允许）的身份进行访问；
- Remember Me: 记住我，这个是非常常见的功能，即一次登录后，下次再来的话不用登录了

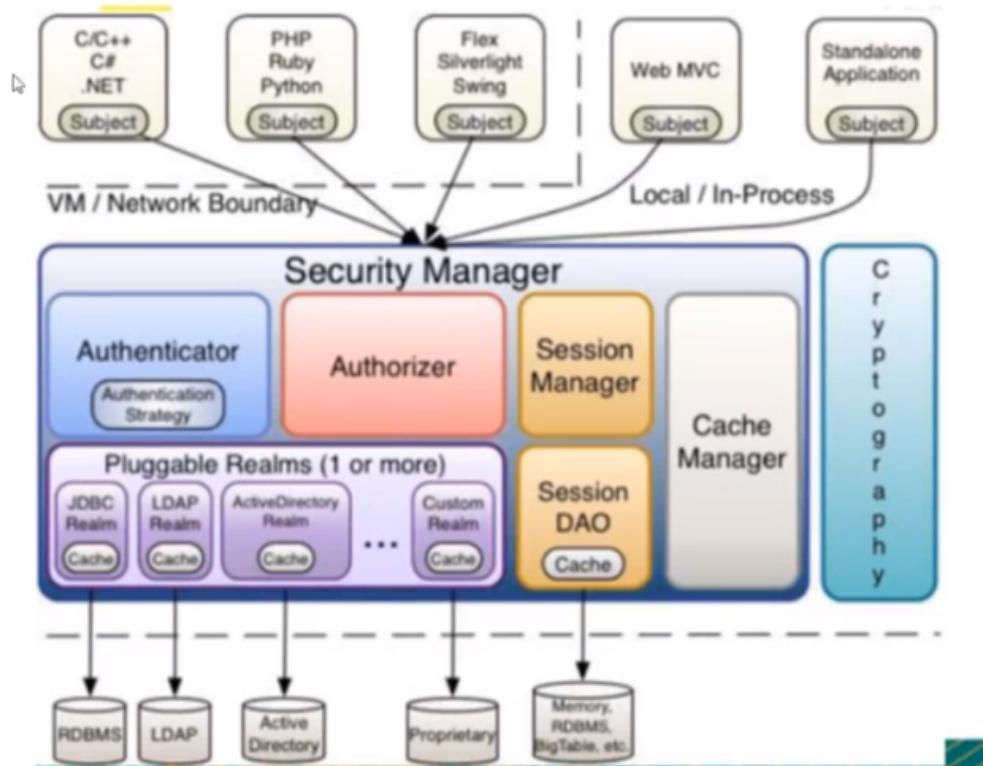
### 1.3、Shiro架构（外部）

从外部来看Shiro，即从应用程序角度来观察如何使用shiro完成工作：



- subject：应用代码直接交互的对象是Subject，也就是说Shiro的对外API核心就是Subject，Subject代表了当前的用户，这个用户不一定是一个具体的人，与当前应用交互的任何东西都是Subject，如网络爬虫，机器人等，与Subject的所有交互都会委托给SecurityManager；Subject其实是一个门面，SecurityManager才是实际的执行者
- SecurityManager：安全管理器，即所有与安全有关的操作都会与SecurityManager交互，并且它管理着所有的Subject，可以看出它是Shiro的核心，它负责与Shiro的其他组件进行交互，它相当于SpringMVC的DispatcherServlet的角色
- subject：应用代码直接交互的对象是Subject，也就是说Shiro的对外API核心就是Subject，Subject代表了当前的用户，这个用户不一定是一个具体的人，与当前应用交互的任何东西都是Subject，如网络爬虫，机器人等，与Subject的所有交互都会委托给SecurityManager；Subject其实是一个门面，SecurityManager才是实际的执行者
- SecurityManager：安全管理器，即所有与安全有关的操作都会与SecurityManager交互，并且它管理着所有的Subject，可以看出它是Shiro的核心，它负责与Shiro的其他组件进行交互，它相当于SpringMVC的DispatcherServlet的角色
- Realm：Shiro从Realm获取安全数据（如用户，角色，权限），也就是说SecurityManager要验证用户身份，那么它需要从Realm获取相应的用户进行比较，来确定用户的身份是否合法；也需要从Realm得到用户相应的角色、权限，进行验证用户的操作是否能够进行，可以把Realm看成DataSource；

## 1.4、Shiro架构（内部）



- Subject: 任何可以与应用交互的‘用户’；
- Security Manager: 相当于SpringMVC中的DispatcherServlet; 是Shiro的心脏，所有具体的交互都通过Security Manager进行控制，它管理者所有的Subject，且负责进行认证，授权，会话，及缓存的管理。
- Authenticator: 负责Subject认证，是一个扩展点，可以自定义实现；可以使用认证策略（Authentication Strategy），即什么情况下算用户认证通过了；
- Authorizer: 授权器，即访问控制器，用来决定主体是否有权限进行相应的操作；即控制着用户能访问应用中的那些功能；
- Realm: 可以有一个或者多个的realm，可以认为是安全实体数据源，即用于获取安全实体的，可以用JDBC实现，也可以是内存实现等等，由用户提供；所以一般在应用中都需要实现自己的realm
- SessionManager: 管理Session生命周期的组件，而Shiro并不仅仅可以用在Web环境，也可以用在普通的JavaSE环境中
- CacheManager: 缓存控制器，来管理如用户，角色，权限等缓存的；因为这些数据基本上很少改变，放到缓存中后可以提高访问的性能；
- Cryptography: 密码模块，Shiro 提高了一些常见的加密组件用于密码加密，解密等

```
1 Subject currentUser = SecurityUtils.getSubject();
2 Session session = currentUser.getSession();
3 currentUser.isAuthenticated()
4 currentUser.getPrincipal()
5 currentUser.hasRole("schwartz")
6 currentUser.isPermitted("lightsaber:wield")
7 currentUser.logout();
```

