# Spring Boot

## 1.Spring Boot 项目的搭建：

### 1.1.注入依赖：在pom.xml文件中注入以下以来：

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.3.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <!--spring boot web 开发 -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>
```

### 1.2.启动类：

> 注解说明:
> @SpringBootApplication：
>
> > 发现@SpringBootApplication是一个复合注解，包括@ComponentScan，和@SpringBootConfiguration，@EnableAutoConfiguration.
> >
> > - @SpringBootConfiguration继承自@Configuration，二者功能也一致，标注当前类是配置类，并会将当前类内声明的一个或多个以@Bean注解标记的方法的实例纳入到srping容器中，并且实例名就是方法名。
> > - @EnableAutoConfiguration的作用启动自动的配置，@EnableAutoConfiguration注解的意思就是Springboot根据你添加的jar包来配置你项目的默认配置，比如根据spring-boot-starter-web ，来判断你的项目是否需要添加了webmvc和tomcat，就会自动的帮你配置web项目中所需要的默认配置。在下面博客会具体分析这个注解，快速入门的demo实际没有用到该注解。
> > - @ComponentScan，扫描当前包及其子包下被@Component，@Controller，@Service，@Repository注解标记的类并纳入到spring容器中进行管理。是以前的context:component-scan（以前使用在xml中使用的标签，用来扫描包配置的平行支持）。所以本demo中的User为何会被spring容器管理。

## 2.Spring Boot + Redis 缓存处理

### 2.1.Redis介绍：

Redis是一款开源的、高性能的键-值存储（key-value store）。它常被称作是一款数据结构服务器（data structure server）。

### 2.2.依赖注入：

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

### 2.3.添加配置信息：

```
# REDIS (RedisProperties)
# Redis数据库索引（默认为0）
spring.redis.database=0
# Redis服务器地址
spring.redis.host=192.168.0.58
# Redis服务器连接端口
spring.redis.port=6379
# Redis服务器连接密码（默认为空）
spring.redis.password=
# 连接池最大连接数（使用负值表示没有限制）
spring.redis.pool.max-active=8
# 连接池最大阻塞等待时间（使用负值表示没有限制）
spring.redis.pool.max-wait=-1
# 连接池中的最大空闲连接
spring.redis.pool.max-idle=8
# 连接池中的最小空闲连接
spring.redis.pool.min-idle=0
# 连接超时时间（毫秒）
spring.redis.timeout=0
```

## 3. thymeleaf模板引擎：

### 3.1.表达式语法：

| 表达式： | **1.**变量表达式 | **2.**选择或星号表达式 | **3.**消息表达式 | **4.URL**表达式 |
|---|---|---|---|---|

- 1.变量表达式:
  变量表达式即OGNL表达式或Spring EL表达式(在Spring术语中也叫model attributes)。如下所示：

  ```
  ${session.user.name}
  ```

- 2.选择或星号表达式
  选择表达式很像变量表达式，不过它们用一个预先选择的对象来代替上下文变量容器(map)来执行，如

下：

```
*{customer.name}
```

- 3.URL表达式
  允许我们从一个外部文件获取区域文字信息(.properties)，用Key索引Value，还可以提供一组参数(可选).

```
#{main.title}
#{message.entrycreated(${entryId})}
```

- 4.URL表达式
  URL表达式指的是把一个有用的上下文或回话信息添加到URL，这个过程经常被叫做URL重写。

```
<form th:action="@{/createOrder}">
<a href="main.html" th:href="@{/main}">
```

**3.1.1常用th标签:**

| | | |
|---|---|---|
| th:id | 替换id | `<input th:id="'xxx' + ${collect.id}"/>` |
| th:text | 文本替换 | `<p th:text="${collect.description}">description</p>` |
| th:utext | 支持html的文本替换 | `<p th:utext="${htmlcontent}">conten</p>` |
| th:object | 替换对象 | `<div th:object="${session.user}">` |
| th:value | 属性赋值 | `<input th:value="${user.name}" />` |
| th:with | 变量赋值运算 | `<div th:with="isEven=${prodStat.count}%2==0"></div>` |
| th:style | 设置样式 | `th:style="'display:' + @{(${sitrue} ? 'none' : 'inline-block')} + ''"` |
| th:onclick | 点击事件 | `th:onclick="'getCollect()'"` |
| th:each | 属性赋值 | `tr th:each="user,userStat:${users}">` |
| th:if | 判断条件 | `<a th:if="${userId == collect.userId}" >` |
| th:unless | 和th:if判断相反 | `<a th:href="@{/login}" th:unless=${session.user != null}>Login</a>` |
| th:href | 链接地址 | `<a th:href="@{/login}" th:unless=${session.user != null}>Login</a> />` |
| th:switch | 多路选择 配 | `<div th:switch="${user.role}">` |

| | | |
|---|---|---|
| | 合th:case 使用 | |
| th:case | th:switch的一个分支 | `<p th:case="'admin'">User is an administrator</p>` |
| th:fragment | 布局标签，定义一个代码片段，方便其它地方引用 | `<div th:fragment="alert">` |
| th:include | 布局标签，替换内容到引入的文件 | `<head th:include="layout :: htmlhead" th:with="title='xx'"></head> />` |
| th:replace | 布局标签，替换整个标签到引入的文件 | `<div th:replace="fragments/header :: title"></div>` |
| th:selected | selected选择框 选中 | `th:selected="(${xxx.id} == ${configObj.dd})"` |
| th:src | 图片类地址引入 | `<img class="img-responsive" alt="App Logo" th:src="@{/img/logo.png}" />` |
| th:inline | 定义js脚本可以使用变量 | `<script type="text/javascript" th:inline="javascript">` |
| th:action | 表单提交的地址 | `<form action="subscribe.html" th:action="@{/subscribe}">` |
| th:remove | 删除某个属性 | `<tr th:remove="all">` 1.all:删除包含标签和所有的孩子。2.body:不包含标记删除,但删除其所有的孩子。3.tag:包含标记的删除,但不删除它的孩子。4.all-but-first:删除所有包含标签的孩子,除了第一个。5.none:什么也不做。这个值是有用的动态评估。 |
| th:attr | 设置标签属性，多个属性可以用逗号分隔 | 比如 `th:attr="src=@{/image/aa.jpg},title=#{logo}"`，此标签不太优雅，一般用的比较少。 |

# 4.Spring Boot + JPA

## 4.1.注入依赖

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

```
        <version>2.1.2.RELEASE</version>
    </dependency>
```

> MySQL 驱动名称改动：
>
> ```
>   spring.datasource.driver-class-name=com.mysql.jdbc.Driver
> 变更为:
>   spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
> ```

### 4.2.创建JPA：

创建JPA接口并且继承SpringDataJPA内的接口作为父类

```
// JpaRepository接口（SpringDataJPA提供的简单数据操作接口）
// JpaSpecificationExecutor（SpringDataJPA提供的复杂查询接口）
public interface UserRepository extends
      JpaRepository<UserEntity,Long>,
      JpaSpecificationExecutor<UserEntity>,
      Serializable {
}
```

### 4.3.将JPA注入到需要使用的类中：

```
    @Autowired
    UserRepository userRepository;

     @Override
    public List<UserEntity> getUserList() {
        List<UserEntity> userList= new ArrayList<UserEntity>();
        userList=userRepository.findAll();
        return  userList;
    }
```

### 4.4.@Query注解自定义SQL

```
  @Query(value = "select id,username from fd_boy where username = ?",nativeQuery = true)
    public UserEntity getUserEntiteByName(String names);
```

## 4.热部署

Spring Boot 配置热部署

- 注入依赖

```
<!-- 热部署 -->
      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-devtools</artifactId>
          <optional>true</optional>
      </dependency>
```

- 添加配置信息

```
spring:
devtools:
  restart:
    enabled: true
freemarker:
  cache: true
```