

《SpringBoot开发》讲义

讲师：朱德福

版本：v1.0

---

### 教学目标：

1. 掌握Spring 注解开发
2. 了解Spring Boot的发展历程；
3. 掌握Spring Boot项目的两种创建方式；
4. 掌握Spring Boot项目组成结构；
5. 了解Spring Boot项目打包与部署。

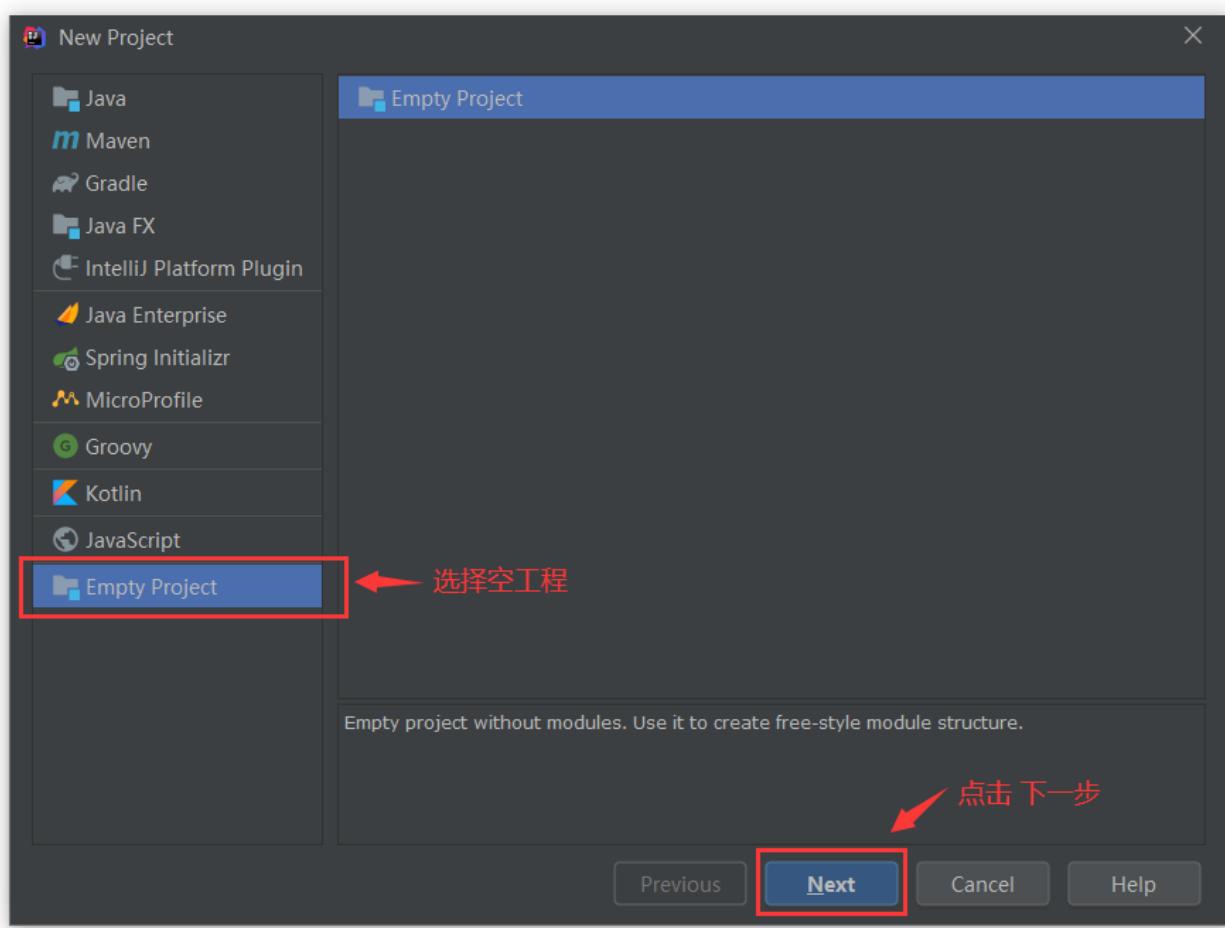
## 一. 基础准备

---

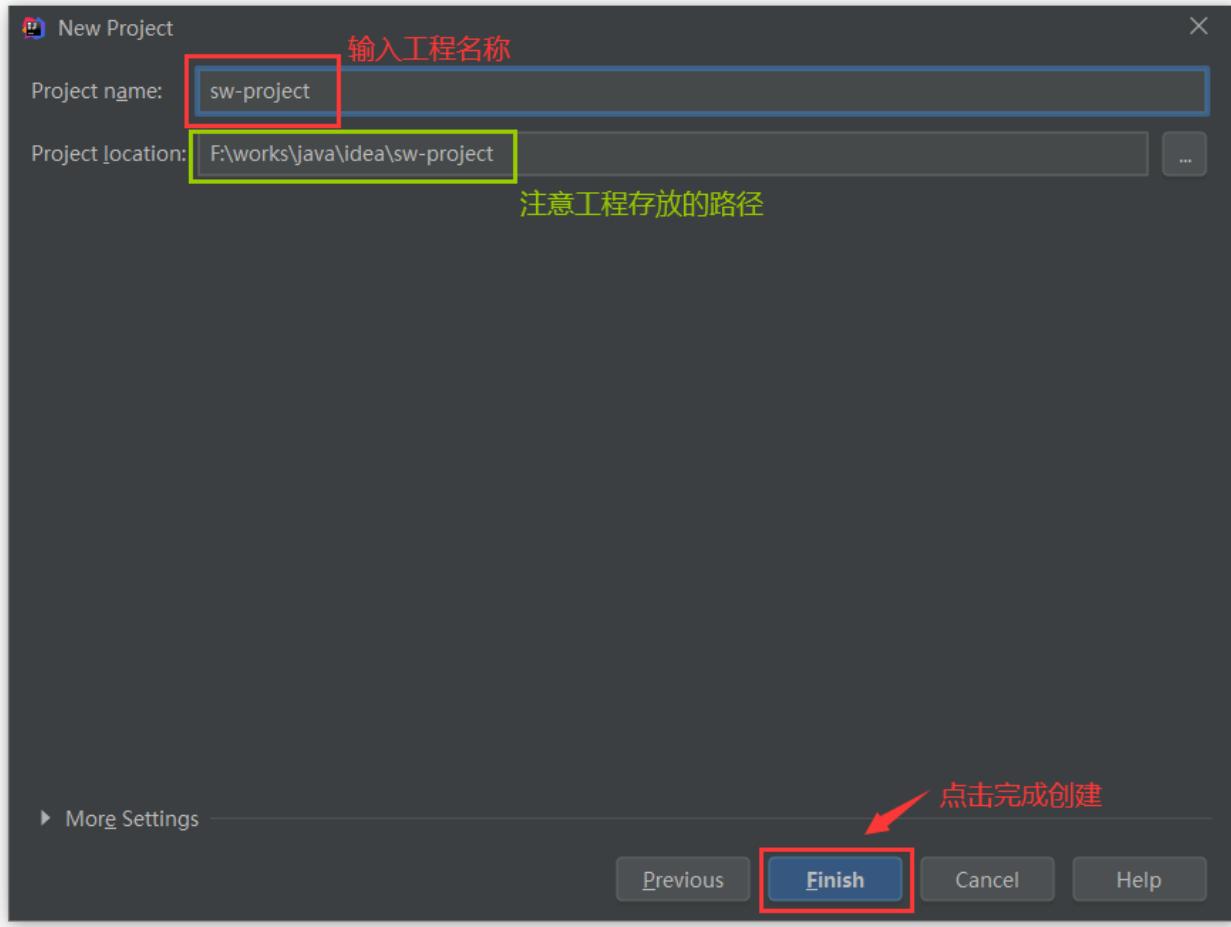
### 1.1 Maven工程

本阶段课程的所有演示项目都是基于Maven构建的，大家需要熟练掌握如何在 **IDEA** 中创建Maven工程。

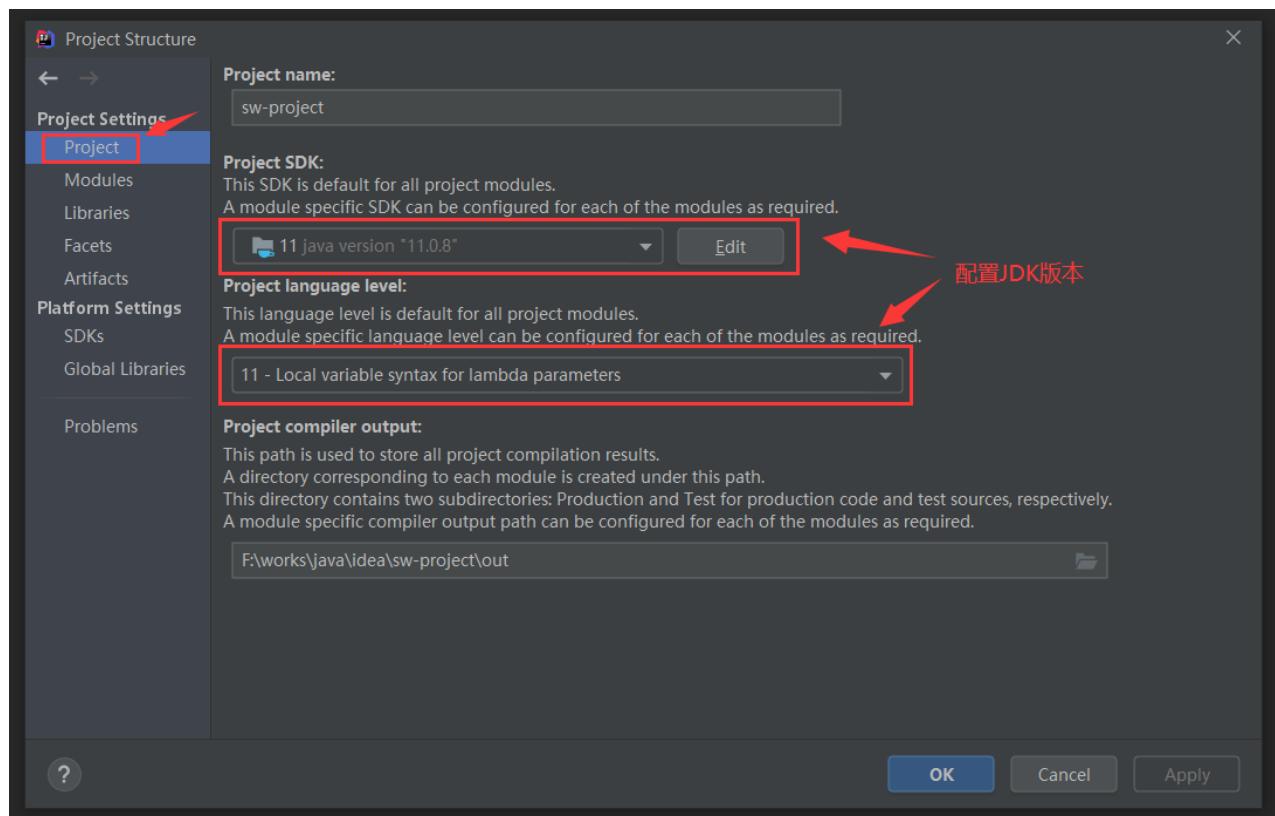
#### 1.1.1 创建空工程



注意修改工程存放的路径，第一次使用IDEA时需要手动调整，往后IDEA将自动记忆。

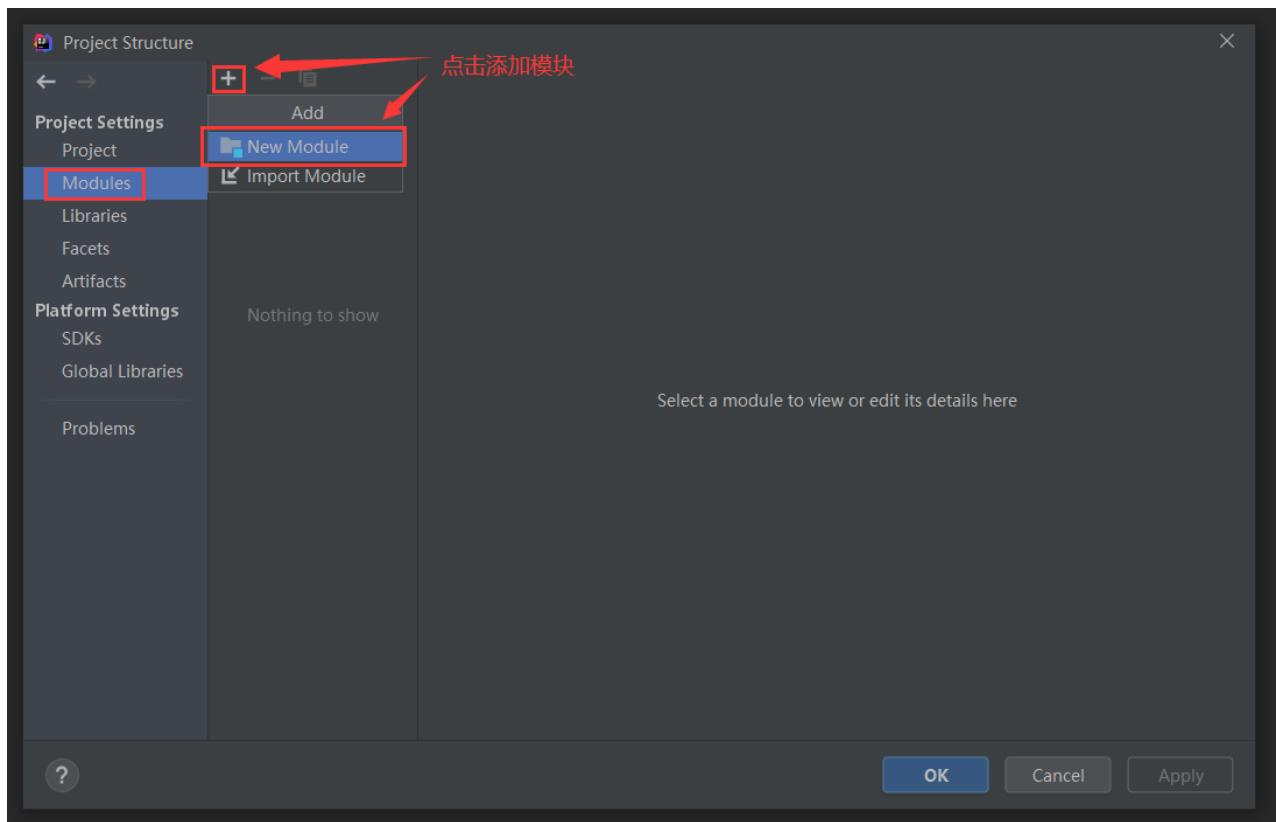


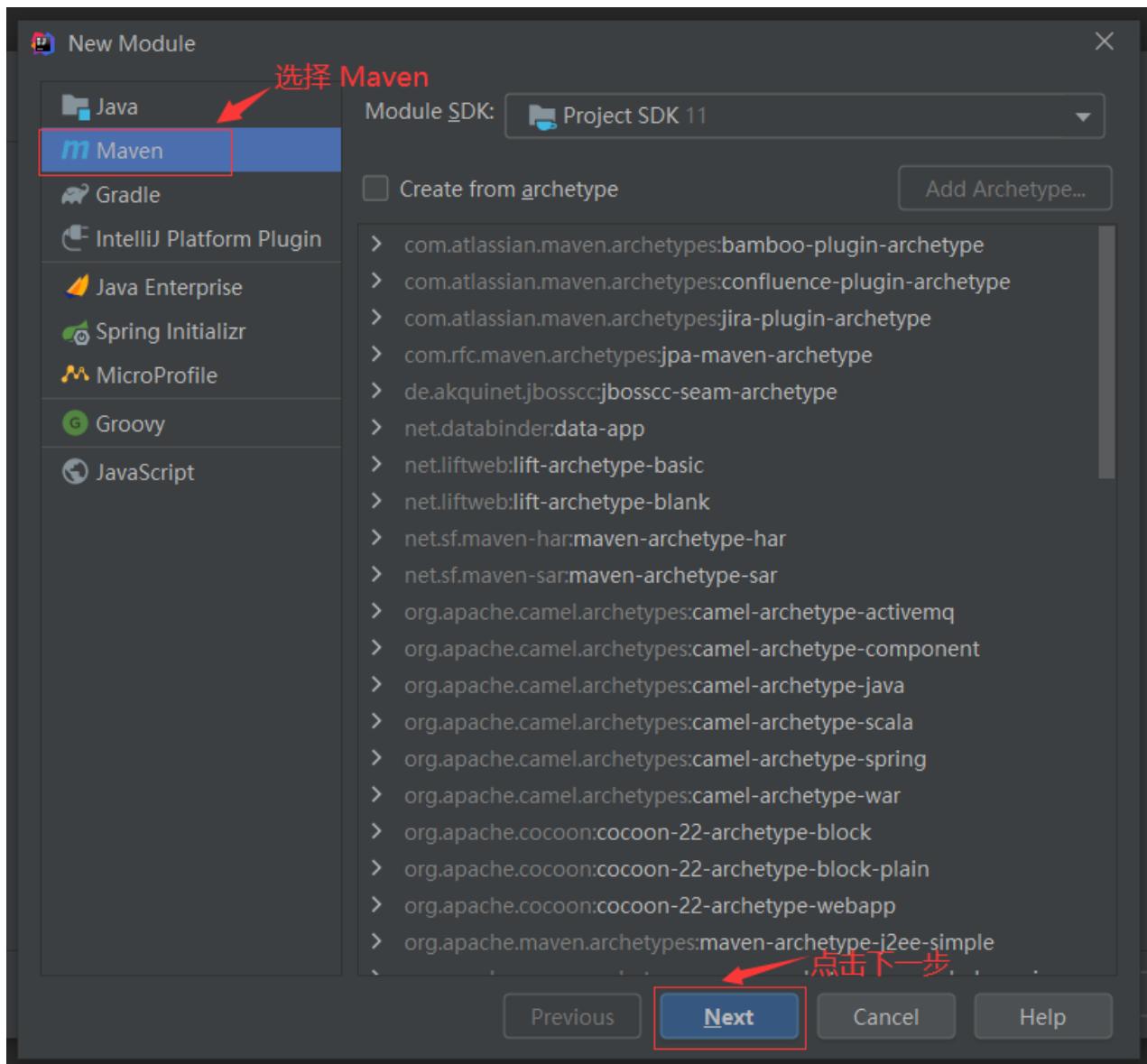
创建完成后，将自动打开工程配置窗口，在这里可以修改工程的JDK版本：



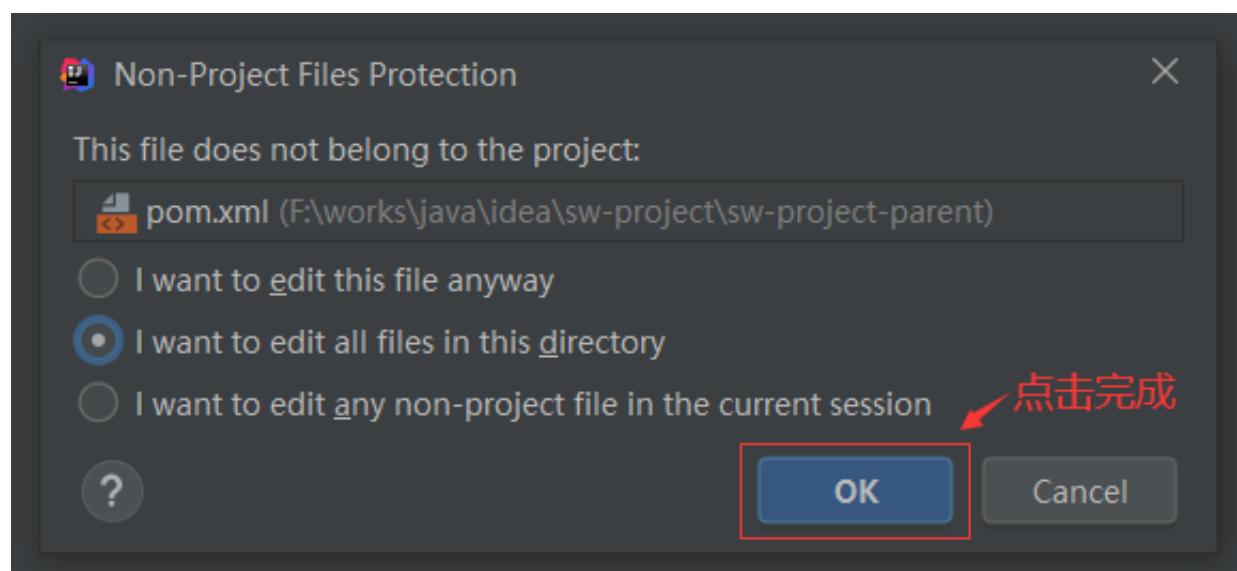
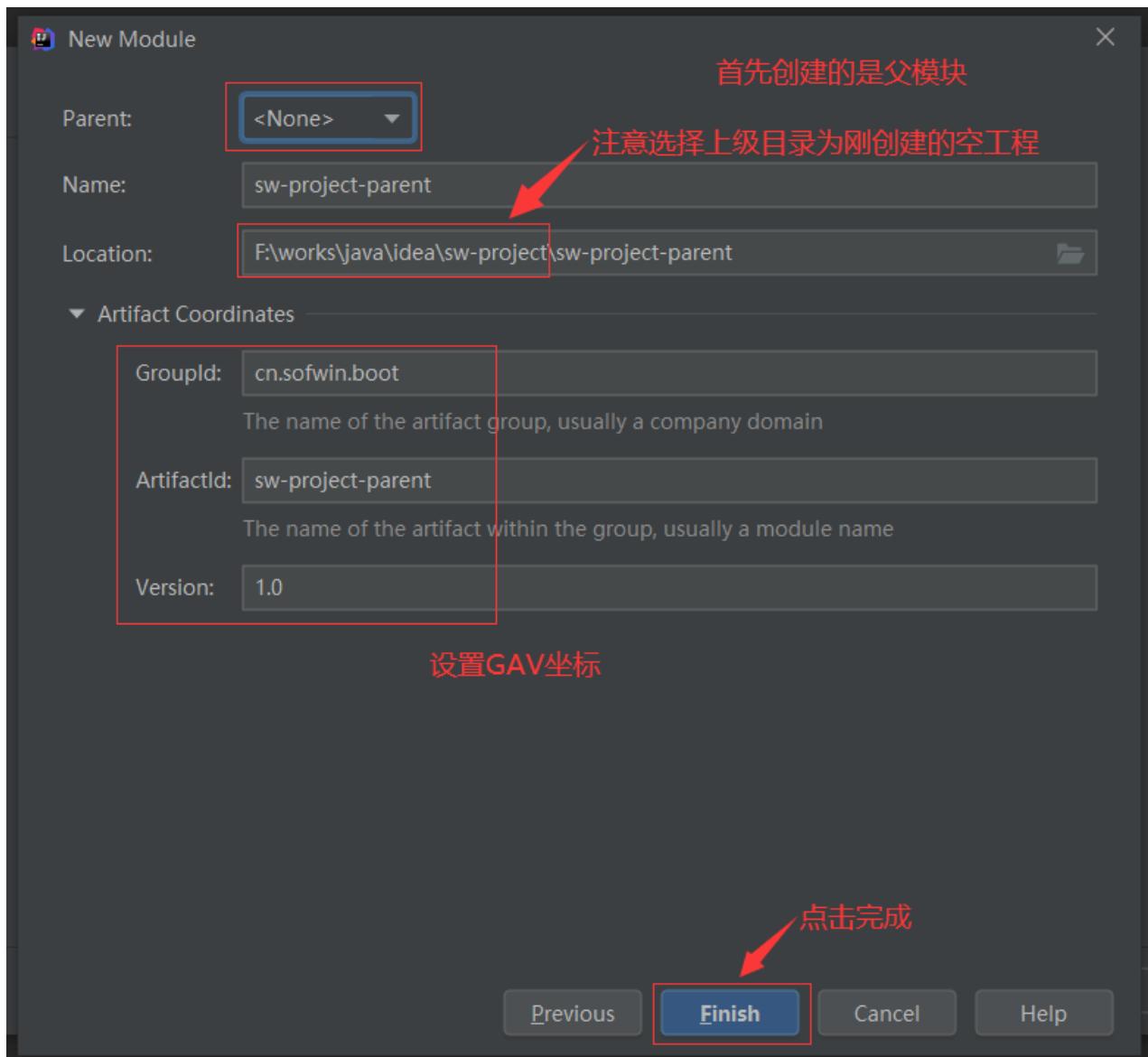
## 1.1.2 创建父模块

点击左侧的 **Modules** 菜单，开始添加父模块：

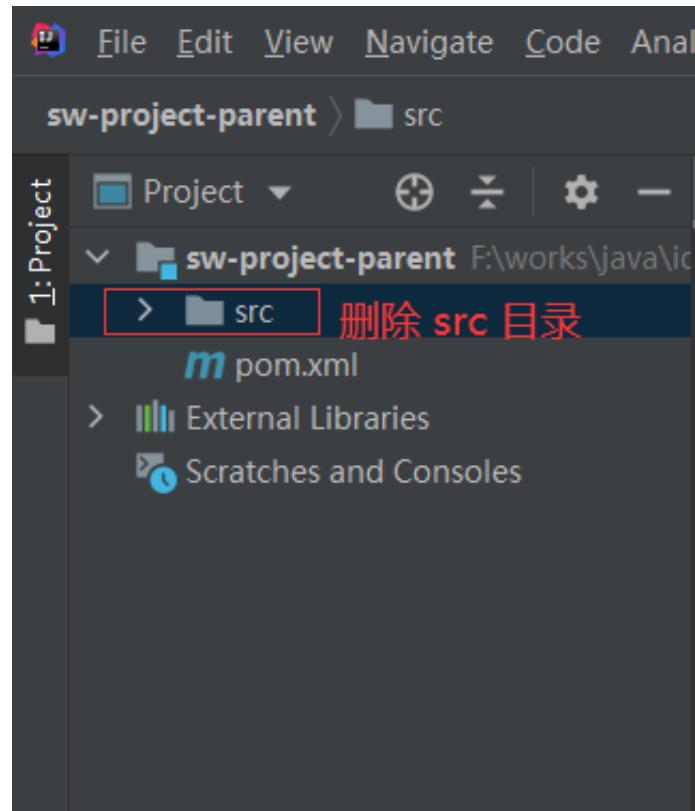




注意模块存放的路径地址，IDEA会自动忽略空工程的目录，我们需要手动补全：



父模块主要用于组件版本的依赖管理，不需要编写Java代码。因此，我们可以将 `src` 目录删除：



最后我们添加基础组件的依赖：

```
11     <!-- 将父模块的打包方式改为POM -->
12     <packaging>pom</packaging>          将父模块的打包方式修改为 POM
13
14     <!-- 添加组件依赖配置 -->
15     <properties>
16         <spring.version>5.2.8.RELEASE</spring.version>
17         <junit.version>4.12</junit.version>
18         <java.version>11</java.version>
19         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
20         <maven-dependency-plugin.version>3.1.2</maven-dependency-plugin.version>
21         <maven-compiler-plugin.version>3.8.1</maven-compiler-plugin.version>
22         <maven-resources-plugin.version>3.2.0</maven-resources-plugin.version>
23         <maven-install-plugin.version>2.5.2</maven-install-plugin.version>
24         <maven-source-plugin.version>3.2.1</maven-source-plugin.version>
25         <maven-jar-plugin.version>3.2.0</maven-jar-plugin.version>
26     </properties>
27
28     <dependencyManagement>
29         <dependencies>
30             <!-- Spring版本依赖 -->
```

完整的配置如下：

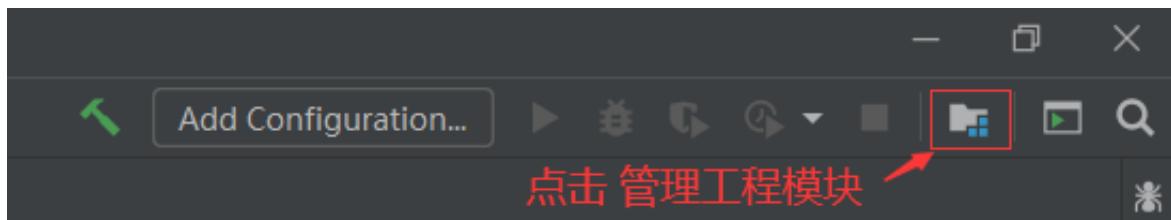
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5             http://maven.apache.org/xsd/maven-4.0.0.xsd">
6
7     <modelVersion>4.0.0</modelVersion>
8
9     <groupId>cn.sofwin.boot</groupId>
10    <artifactId>sw-project-parent</artifactId>
11    <version>1.0</version>
12
13    <!-- 将父模块的打包方式改为POM -->
14    <packaging>pom</packaging>
15
16    <!-- 添加组件依赖配置 -->
17    <properties>
18        <!-- SpringFramework -->
19        <spring.version>5.2.11.RELEASE</spring.version>
20
21        <!-- 通用组件 -->
22        <lombok.version>1.18.16</lombok.version>
23        <junit.version>4.12</junit.version>
24        <java.version>11</java.version>
25        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
26        <maven-dependency-plugin.version>3.1.2</maven-dependency-plugin.version>
27        <maven-compiler-plugin.version>3.8.1</maven-compiler-plugin.version>
28        <maven-resources-plugin.version>3.2.0</maven-resources-plugin.version>
29        <maven-install-plugin.version>2.5.2</maven-install-plugin.version>
30        <maven-source-plugin.version>3.2.1</maven-source-plugin.version>
31        <maven-jar-plugin.version>3.2.0</maven-jar-plugin.version>
32    </properties>
33
34    <dependencyManagement>
35        <dependencies>
36            <!-- Spring版本依赖 -->
37            <dependency>
38                <groupId>org.springframework</groupId>
39                <artifactId>spring-context</artifactId>
40                <version>${spring.version}</version>
41            </dependency>
42
43            <!-- Lombok 依赖 -->
44            <dependency>
45                <groupId>org.projectlombok</groupId>
46                <artifactId>lombok</artifactId>
47                <version>${lombok.version}</version>
48                <scope>provided</scope>
49            </dependency>
50
51            <!-- Junit -->
52            <dependency>
53                <groupId>junit</groupId>
54                <artifactId>junit</artifactId>
55                <version>${junit.version}</version>
56                <scope>test</scope>
```

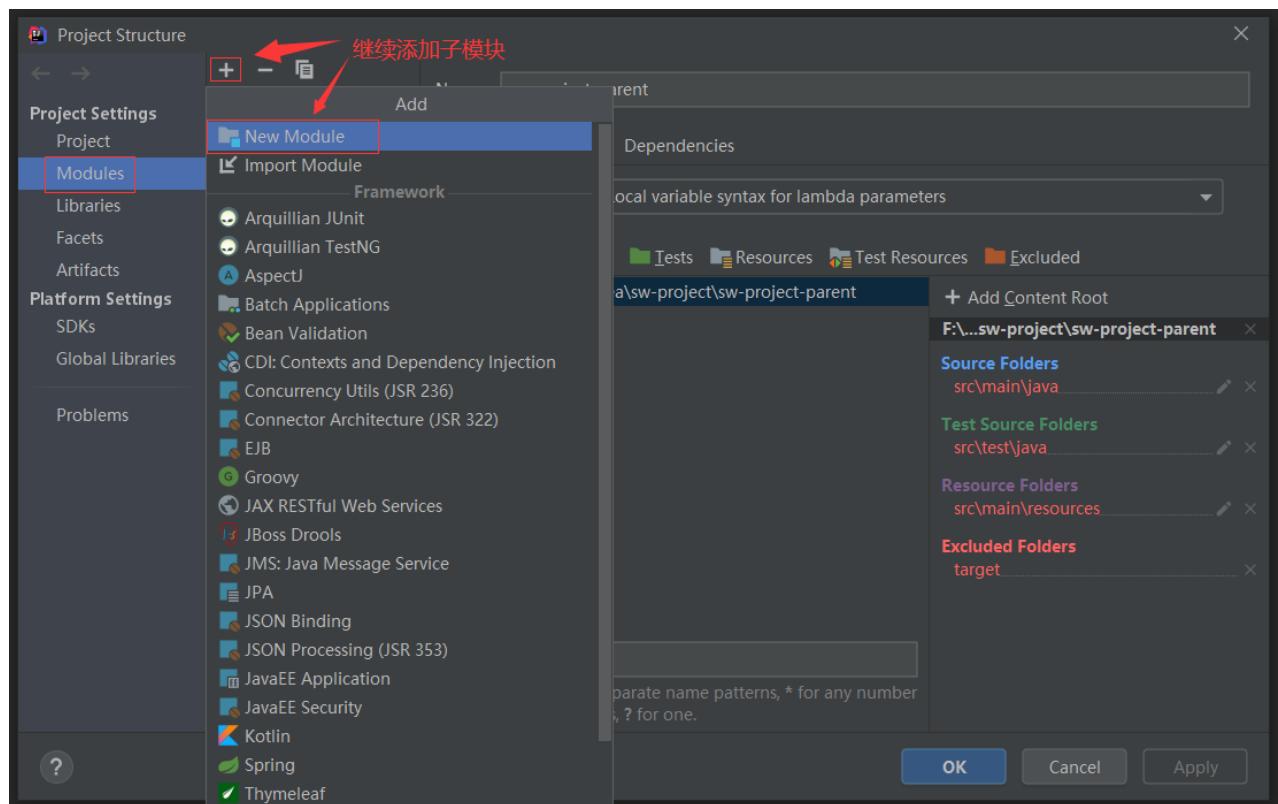
```
55      </dependency>
56    </dependencies>
57  </dependencyManagement>
58</project>
```

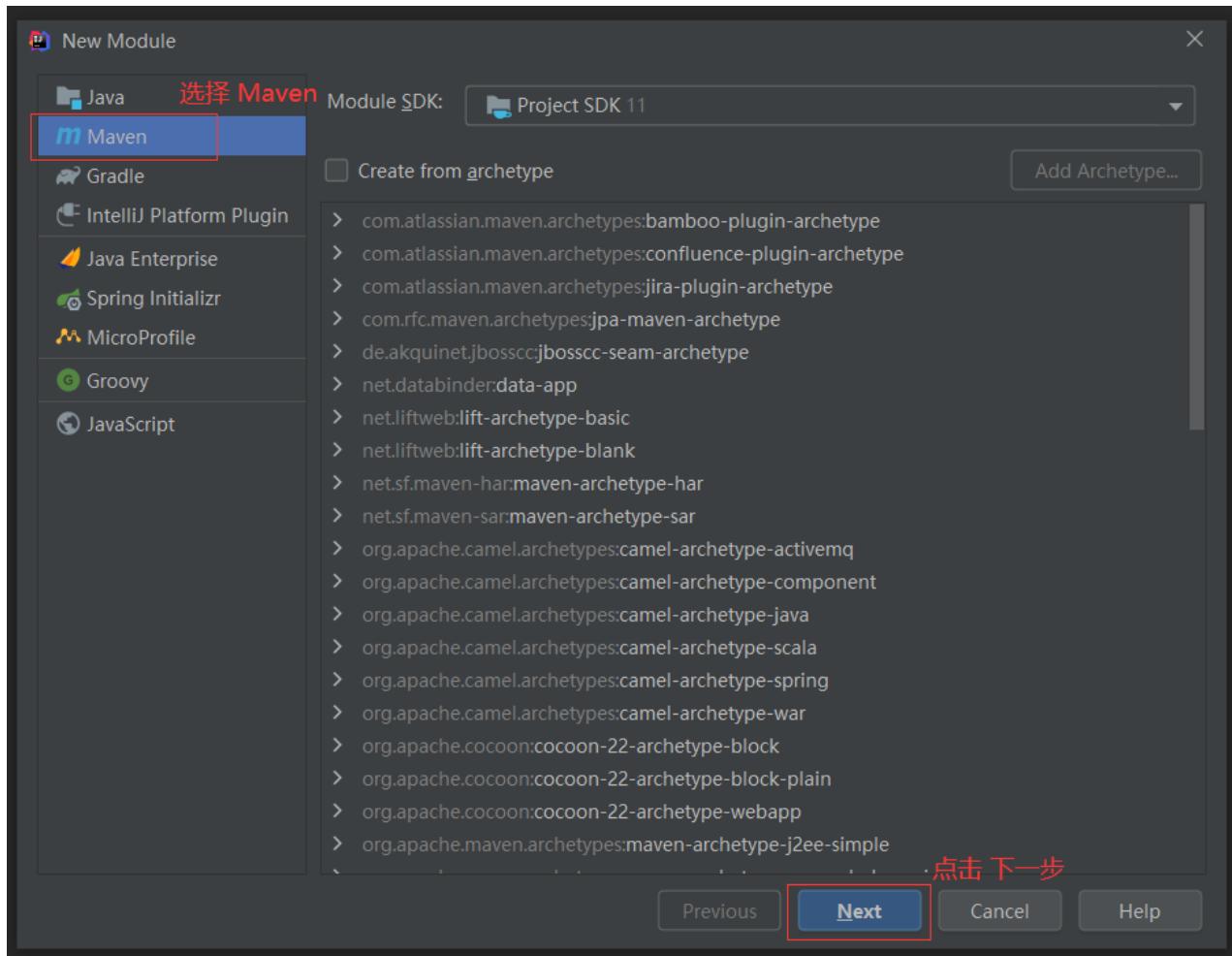
### 1.1.3 创建子模块

点击 IDEA 窗口右上角的 图标，如图：

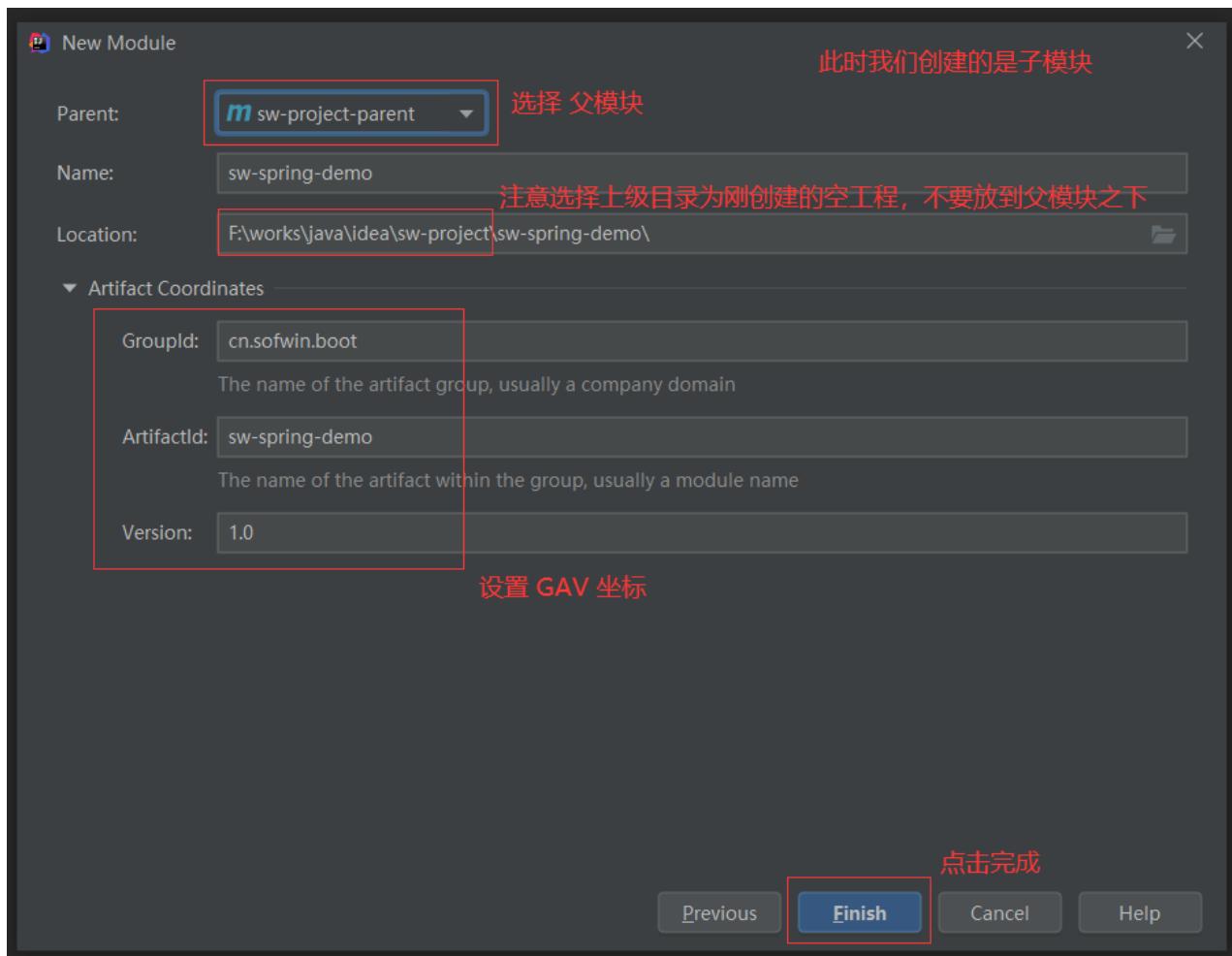


选择 Modules 菜单，开始添加子模块：





注意：①设置 **Parent** 为刚创建的父模块；②修改子模块的上级目录为空工程目录（IDEA默认将子模块存放在父模块的目录之下）



创建完成后，可以在子模块的 POM 文件中添加组件依赖：

```
5 <parent>
6   <artifactId>sw-project-parent</artifactId>
7   <groupId>cn.sofwin.boot</groupId>
8   <version>1.0</version>
9   <relativePath>../sw-project-parent/pom.xml</relativePath>
10 </parent>
11 <modelVersion>4.0.0</modelVersion>
12 <artifactId>sw-spring-demo</artifactId>
13
14 <dependencies>
15   <!-- Spring版本依赖 -->
16   <dependency>
17     <groupId>org.springframework</groupId>
18     <artifactId>spring-context</artifactId>
19   </dependency>
20
21   <!-- Junit -->
22   <dependency>
23     <groupId>junit</groupId>
24     <artifactId>junit</artifactId>
25   </dependency>
26 </dependencies>
27 </project>
```

添加 组件依赖

后期如果还要添加子模块，重复此操作。

## 课堂练习

1. 创建一个名称为：`sw-springboot-project` 的空工程；
2. 在工程中添加 `sw-spring-parent` 父模块；
3. 在父模块中配置 `spring-context`、`junit` 依赖的版本；
4. 创建子模块：`sw-spring-test`，子模块依赖父模块；
5. 在子模块中添加 `junit` 测试用例，输出 `九九乘法表`。

## 1.2 Spring 注解开发

添加子模块：`sw-spring-basic`

### 1.2.1 添加依赖

```
1 <dependencies>
2     <!-- Spring版本依赖 -->
3     <dependency>
4         <groupId>org.springframework</groupId>
5         <artifactId>spring-context</artifactId>
6     </dependency>
7
8     <!-- Lombok 依赖 -->
9     <dependency>
10        <groupId>org.projectlombok</groupId>
11        <artifactId>lombok</artifactId>
12        <scope>provided</scope>
13    </dependency>
14
15    <!-- Junit -->
16    <dependency>
17        <groupId>junit</groupId>
18        <artifactId>junit</artifactId>
19        <scope>test</scope>
20    </dependency>
21 </dependencies>
```

## 1.2.2 添加实体类

```
1  public class User {  
2      private Integer id;  
3  
4      private String username;  
5  
6      private String password;  
7  
8      private String nickname;  
9  
10     public User() {  
11         }  
12  
13     public Integer getId() {  
14         return id;  
15     }  
16  
17     public void setId(Integer id) {  
18         this.id = id;  
19     }  
20  
21     public String getUsername() {  
22         return username;  
23     }  
24  
25     public void setUsername(String username) {  
26         this.username = username;  
27     }  
28  
29     public String getPassword() {  
30         return password;  
31     }  
32  
33     public void setPassword(String password) {  
34         this.password = password;  
35     }  
36  
37     public String getNickname() {  
38         return nickname;  
39     }  
40  
41     public void setNickname(String nickname) {  
42         this.nickname = nickname;  
43     }  
44  
45     @Override  
46     public String toString() {  
47         return "User{" +  
48                 "id=" + id +  
49                 ", username='" + username + '\'' +
```

```
50             ", password='" + password + '\'' +  
51             ", nickname='" + nickname + '\'' +  
52         '}';  
53     }  
54 }
```

### 1.2.3 添加配置类

```
1  @Configuration  
2  public class AppConfig {  
3  
4      @Bean  
5      public User user(){  
6          User user = new User();  
7          user.setId(1);  
8          user.setUsername("zhangsan");  
9          user.setPassword("123456");  
10         user.setNickname("张三");  
11  
12         return user;  
13     }  
14 }
```

### 1.2.4 测试用例

```
1  public class SpringTest {  
2      @Test  
3      public void testGetBean(){  
4          ApplicationContext act = new  
5              AnnotationConfigApplicationContext(AppConfig.class);  
6          // User user = act.getBean(User.class);  
7          User user = act.getBean("user", User.class);  
8  
9          System.out.println(user);  
10     }
```

### 1.2.5 自动导入

```
1  @Configuration  
2  @ComponentScan("cn.sofwin")  
3  public class AppConfig {  
4  
5      @Bean  
6      public User user(){  
7          User user = new User();
```

```
8     user.setId(1);
9     user.setUsername("zhangsan");
10    user.setPassword("123456");
11    user.setNickname("张三");
12
13    return user;
14 }
15 }
```

## 1.2.6 添加服务组件

添加服务类: `cn.sofwin.spring.service.UserService`

注意: 需要放在在被扫描的 `cn.sofwin` 包或其子包下

```
1 @Service
2 public class UserService {
3     public void sayHello(String msg){
4         System.out.println("Hello " + msg);
5     }
6 }
```

## 1.2.7 测试用例

```
1 @Test
2 public void testScan(){
3     ApplicationContext act = new
4     AnnotationConfigApplicationContext(AppConfig.class);
5     UserService userService = act.getBean(UserService.class);
6     userService.sayHello("软赢科技");
7 }
```

## 1.2.8 扩展配置

1. `@Lazy` 配置bean懒加载
2. `@Scope` 配置bean的作用域（单例、原型）
3. `@Conditional` 配置bean的导入条件
4. `@Import` 在配置类中快速导入组件
5. 通过 `ImportSelector` 的实现类在 `@Import` 注解中配置选择器  
`ImportSelector` 的实现类:

```
1  public class SwImportSelector implements ImportSelector {
2      @Override
3      public String[] selectImports(AnnotationMetadata importingClassMetadata)
4      {
5          // return new String[]{"com.so.xxx"}; // 配置类的全类名
6          return new String[0];
7      }
8 }
```

在 `@Import` 注解中使用:

```
1  @Configuration
2  @Import({User.class, SwImportSelector.class})
3  public class AppConfig {
4      ...
5 }
```

## 6. 通过 `FactoryBean` 导入组件

创建工厂 Bean:

```
1  public class SwFactoryBean implements FactoryBean<User> {
2      @Override
3      public User getObject() throws Exception {
4          return new User().setId(1111);
5      }
6
7      @Override
8      public Class<?> getObjectType() {
9          return User.class;
10     }
11 }
```

在配置类中导入:

```
1  @Bean
2  public SwFactoryBean swFactoryBean(){
3      return new SwFactoryBean();
4 }
```

获取 Bean:

```
1  @Test
2  public void testFactoryBean(){
3      ApplicationContext context = new
4          AnnotationConfigApplicationContext(AppConfig.class);
5      // 注意：虽然我们是获取swFactoryBean，但他的返回类型是User
6      User user = context.getBean("swFactoryBean", User.class);
7      // 如果在id前面添加&字符，则代表获取工厂类本身
8      SwFactoryBean swFactoryBean = context.getBean("&swFactoryBean",
9          SwFactoryBean.class);
10 }
```

## 1.2.9 生命周期

Bean的生命周期主要包含：创建对象 --> 初始化 --> 销毁对象。

在Spring中由IOC容器管理Bean生命周期，我们可以自定义初始化、销毁的回调方法，当容器执行相关动作之前将会进行回调。

自定义回调方法：

### 1. 指定初始化和销毁方法

创建实体类：

```
1  @Data
2  @Accessors(chain = true)
3  public class Computer {
4
5      private String brand;
6
7      private String name;
8
9      public Computer() {
10         System.out.println("计算机创建中...");
11     }
12
13     // 定义初始化方法
14     public void init(){
15         System.out.println("计算机初始化中...");
16     }
17
18     // 定义销毁方法
19     public void destroy(){
20         System.out.println("计算机销毁中...");
21     }
22 }
```

配置类中指定方法：

```
1 // 指定初始化、销毁方法
2 @Bean(initMethod = "init", destroyMethod = "destroy")
3 public Computer computer(){
4     return new Computer();
5 }
```

2. 实体类实现 `InitializingBean`、`DisposableBean` 接口，此时不需要显示的在配置类中指定初始化、销毁方法
3. 在实体类的方法上添加注解 `@PostConstruct`、`@PreDestroy`，以标明初始化、销毁时调用。此时不需要显示的在配置类中指定初始化、销毁方法。如：

注意：这两个注解是 JSR-250 规范提供的，Spring 默认没有包含这个依赖的 jar 包。使用之前需要添加依赖：

```
1 <dependency>
2     <groupId>javax.annotation</groupId>
3     <artifactId>javax.annotation-api</artifactId>
4     <version>1.3.2</version>
5 </dependency>
```

示例：

```
1 @Data
2 @Accessors(chain = true)
3 public class Computer {
4
5     private String brand;
6
7     private String name;
8
9     public Computer() {
10         System.out.println("计算机创建中... ");
11     }
12
13     // 定义初始化方法
14     @PostConstruct
15     public void init(){
16         System.out.println("计算机初始化中... ");
17     }
18
19     // 定义销毁方法
20     @PreDestroy
21     public void destroy(){
22         System.out.println("计算机销毁中... ");
23     }
24 }
```

4. 使用 `BeanPostProcessor` 后置处理器

`BeanPostProcessor` 接口中有两个方法：

- `postProcessBeforeInitialization`：在所有初始化方法执行之前调用

- `postProcessAfterInitialization`：在所有初始化方法执行之后调用

自定义后置处理器：

```

1  @Component // 将该后置处理器导入到IOC容器中
2  public class SwBeanPostProcessor implements BeanPostProcessor {
3      @Override
4      public Object postProcessBeforeInitialization(Object bean, String
5          beanName) throws BeansException {
6              System.out.println("postProcessBeforeInitialization..." + beanName);
7              return bean;
8      }
9
10     @Override
11     public Object postProcessAfterInitialization(Object bean, String
12         beanName) throws BeansException {
13         System.out.println("postProcessAfterInitialization..." + beanName);
14         return bean;
15     }
16 }
```

## 1.2.10 属性赋值

可以使用 `@Value` 注解对Bean的属性进行赋值，如：

`@Value` 注解中的值可以是：字面值常量、SpringEL表达式、及配置文件中的值

注意：如果要使用外部的配置文件，需要在配置类中导入配置文件，如：

```

1  @Configuration
2  @PropertySource("classpath:/app.properties")
3  public class AppConfig4PropertyValue {
4      ...
5  }
```

配置文件：`app.properties`

```

1  user.id=1
2  user.username=wangwu
3  user.password=123456
4  user.nickname=王五
```

实体类：

```

1  @Data
2  @Accessors(chain = true)
3  public class User {
4
5      // @Value("11") // 1. 字面常量
6      // @Value("#{22 + 33}") // 2. SpringEL表达式
7      @Value("${user.id}") // 3. 加载配置文件中的值
```

```
8     private Integer id;  
9  
10    @Value("${user.username}")  
11    private String username;  
12  
13    @Value("${user.password}")  
14    private String password;  
15  
16    @Value("${user.nickname}")  
17    private String nickname;  
18 }
```

## 课堂练习

基于注解的Spring，实现MVC模式的、命令行版的用户注册、登录功能。

要求：

1. 无需连接数据库，通过集合类在内存中存储用户注册信息；
2. 通过配置文件，配置默认用户信息（zhangsan、123456）；
3. 要分别实现 **Service**、**Dao** 层
4. 通过自动扫描注入相应的 **service** 和 **dao**。

## 二. Spring Boot基础

### 2.1 Spring Boot简介

#### 2.1.1 Spring Boot是什么

由Rod Johnson主导开发的Spring项目在2004年推出了1.0稳定版本，并从此彻底改变了JavaEE开发的世界。在早期的1.x版本中，由于JDK并不支持注解，Spring只能使用XML。JDK升级到5.0版时，开始加入了注解的新特性。此时Spring团队内部也分为两派：一派是使用XML的支持者；一派是使用注解的支持者。几经讨论，Spring 2.x版本开始少量的引入注解。但功能不够强大。绝大部分情况下还是以XML为主、注解为辅。

Spring更新到3.0版后，则引入了更多的注解。Spring团队内部也产生了严重分歧：到底是使用XML还是使用注解？喜欢注解的人认为XML太过繁琐、臃肿，喜欢XML的人认为注解分散的到处都是、难以控制。最终，大家达成一个共识：业务类使用注解（如：@Controller、@Service等），通用配置使用XML（如：数据库、缓存、框架整合等）。

但随着注解的功能不断增强，Spring对XML的依赖越来越少，到了Spring4.x，甚至已经可以完全脱离XML了。此时，Spring团队在原有的基础上基于注解简化了Spring框架的开发，Spring Boot项目孕育而生。所以，Spring Boot并不是为了取代Spring，而是让Spring框架开发更加容易、快速。

Spring Boot于2014年发布1.0版本（Spring 4.0），2018年发布了2.0版（Spring 5.0）。

资料与链接：

Spring官网：<https://spring.io>

Spring Boot：<https://spring.io/projects/spring-boot>

Spring Cloud：<https://spring.io/projects/spring-cloud>

### 2.1.2 Spring Boot的优点

- 创建独立的Spring应用程序；
- 嵌入Tomcat、jetty或Undertow，无需部署WAR；
- 允许通过Maven按需配置starter；
- 尽可能地自动配置Spring；
- 提供运行时应用监控；
- 没有代码生成、对XML没有要求配置；
- 是Spring Cloud微服务框架的基石。

### 2.1.3 Spring Boot的缺点

入门容易、精通难。如果对Spring注解、框架底层不了解的话，就很难理解Spring Boot的原理。

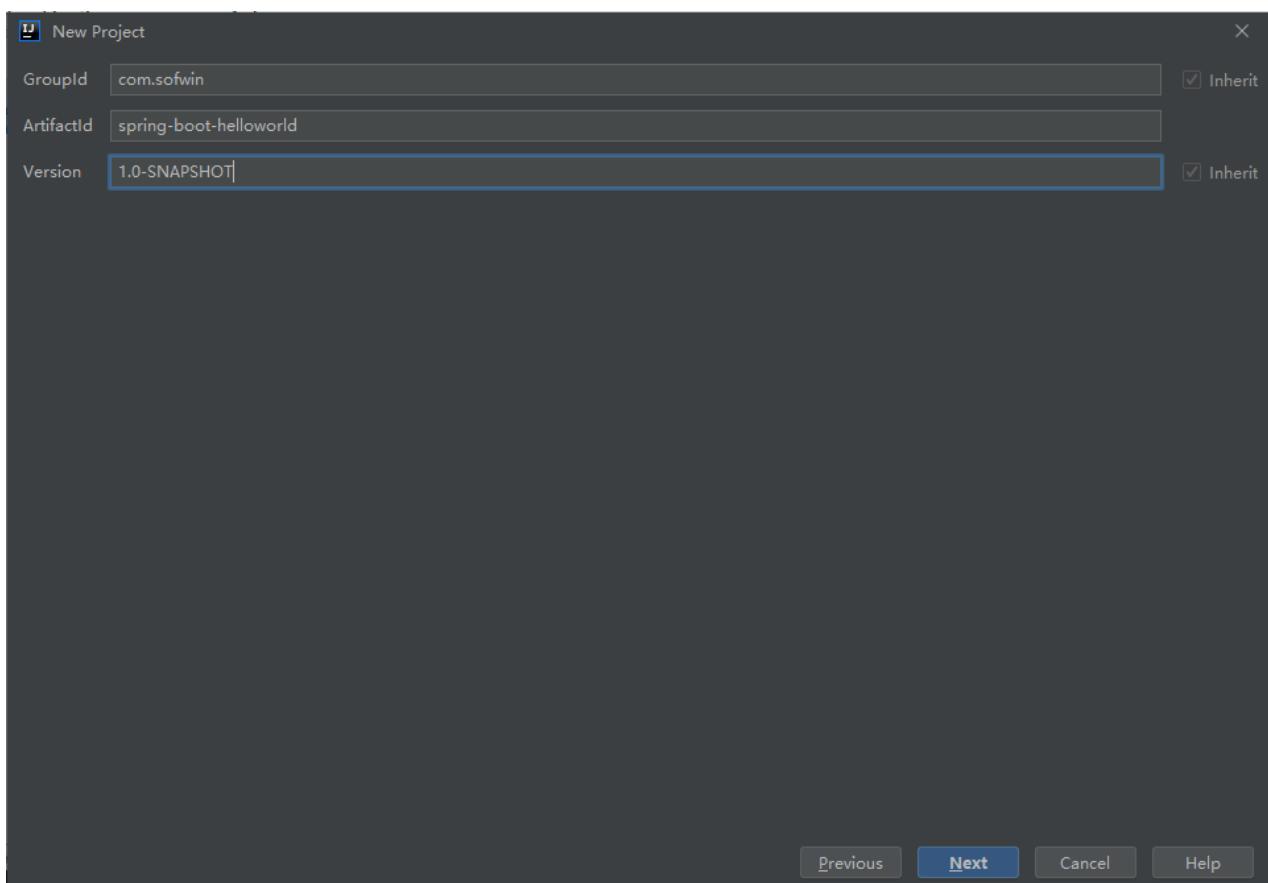
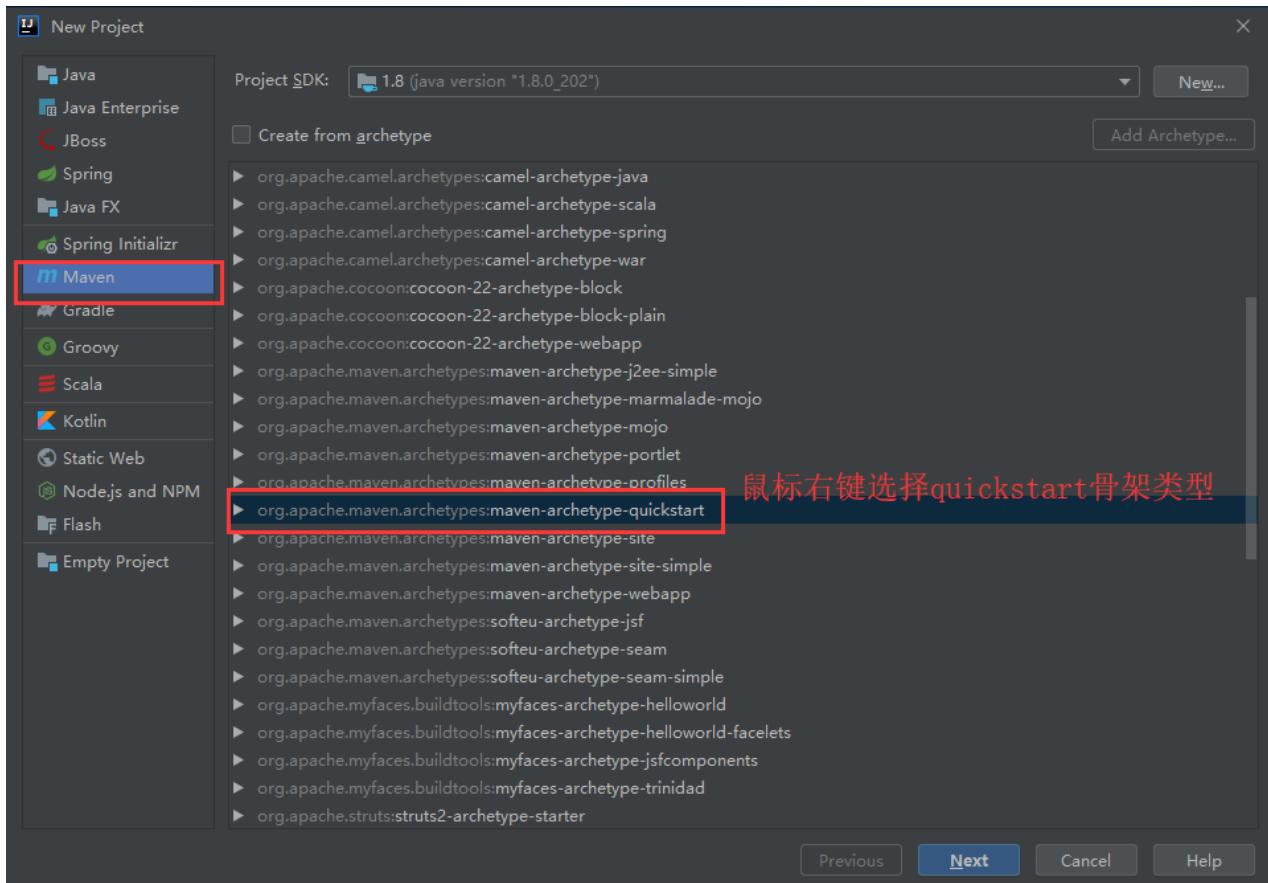
## 2.2 Spring Boot项目

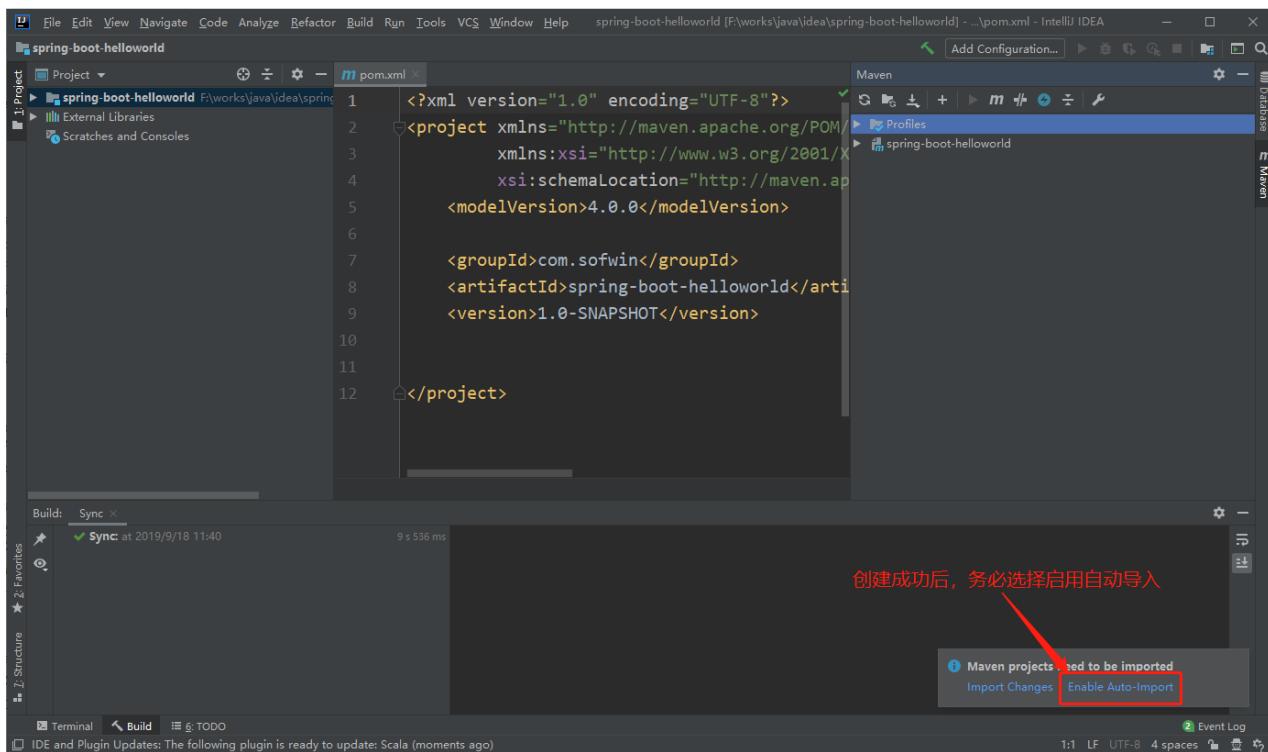
### 2.2.1 手动创建

开发流程：

1. 创建一个Maven工程
2. 导入Spring Boot相关依赖
3. 创建启动类
4. 编写业务类
5. 运行项目

1. 创建Maven工程：





## 2. 导入Spring Boot依赖

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5               http://maven.apache.org/xsd/maven-4.0.0.xsd">
6      <modelVersion>4.0.0</modelVersion>
7
8      <groupId>cn.sofwin</groupId>
9      <artifactId>spring-boot-helloworld</artifactId>
10     <version>1.0</version>
11
12     <!-- 1. 继承 Spring Boot 父项目 -->
13     <parent>
14         <groupId>org.springframework.boot</groupId>
15         <artifactId>spring-boot-starter-parent</artifactId>
16         <version>2.3.6.RELEASE</version>
17     </parent>
18
19     <!-- 2. 管理依赖, 导入starter (启动器) -->
20     <dependencies>
21         <dependency>
22             <groupId>org.springframework.boot</groupId>
23             <artifactId>spring-boot-starter-web</artifactId>
24         </dependency>
25     </dependencies>
26
27     <!-- 3. 打包工具 -->
28     <build>
```

```
28      <plugins>
29          <plugin>
30              <groupId>org.springframework.boot</groupId>
31              <artifactId>spring-boot-maven-plugin</artifactId>
32          </plugin>
33      </plugins>
34  </build>
35 </project>
```

### 3. 创建启动类

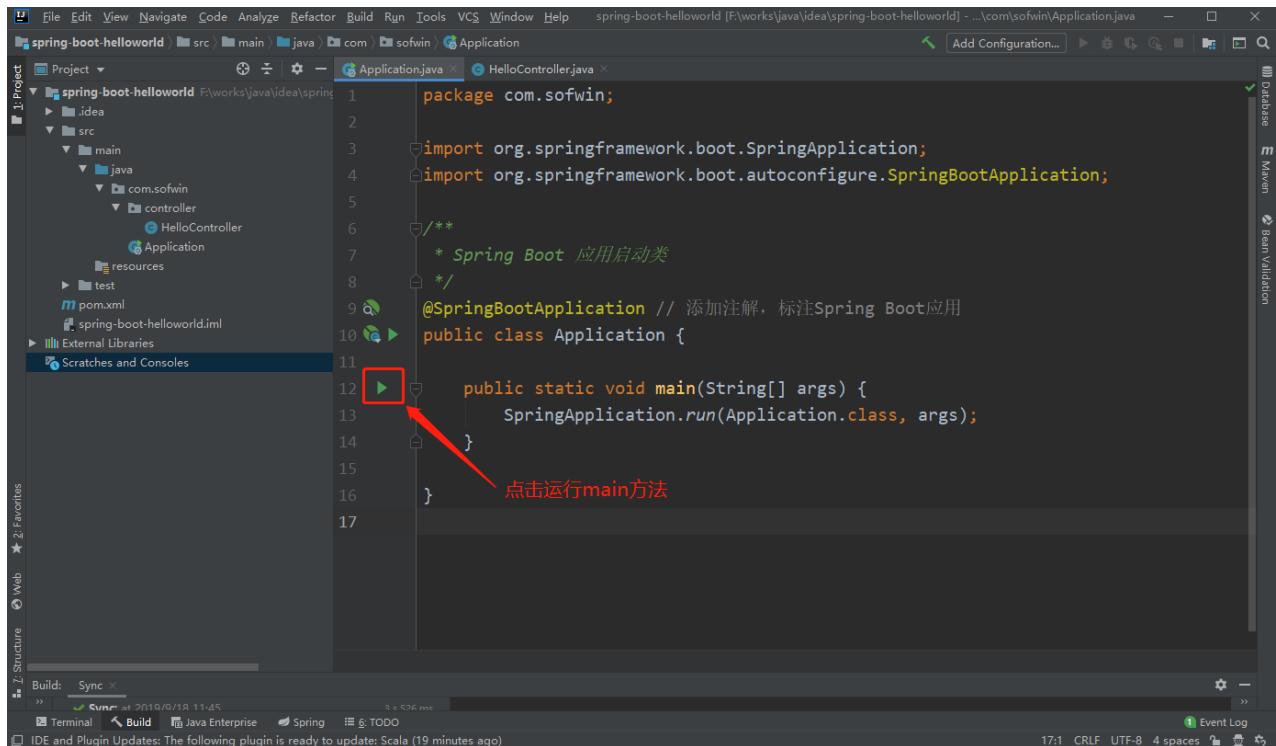
```
1 package cn.sofwin;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 /**
7  * Spring Boot 应用启动类
8 */
9 @SpringBootApplication // 添加注解, 标注Spring Boot应用
10 public class Application {
11
12     public static void main(String[] args) {
13         SpringApplication.run(Application.class, args);
14     }
15
16 }
```

### 4. 编写业务类

```
1 package cn.sofwin.controller;
2
3 import org.springframework.web.bind.annotation.RequestMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
7 public class HelloController {
8
9     @RequestMapping("/hello")
10    public String hello(){
11        return "Hello World!";
12    }
13
14 }
```

### 5. 运行程序

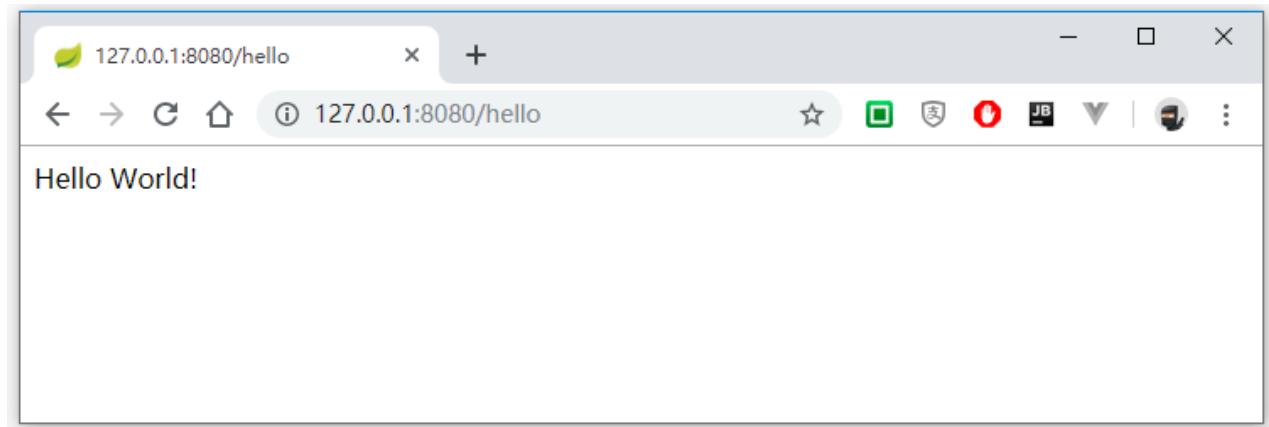
运行启动类的 `main` 方法，即可启动应用。



```
package com.sofwin;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
/**
 * Spring Boot 应用启动类
 */
@SpringBootApplication // 添加注解，标注Spring Boot应用
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

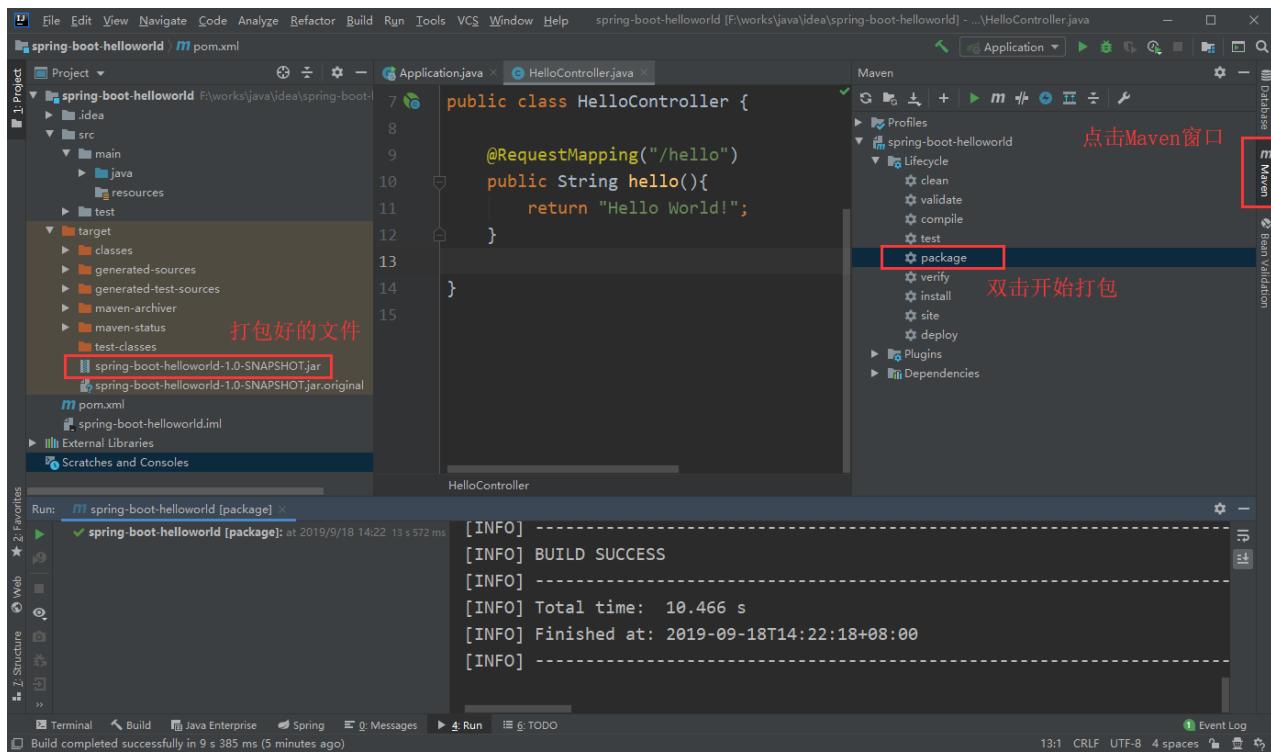
点击运行main方法

访问测试：



## 6. 打包部署

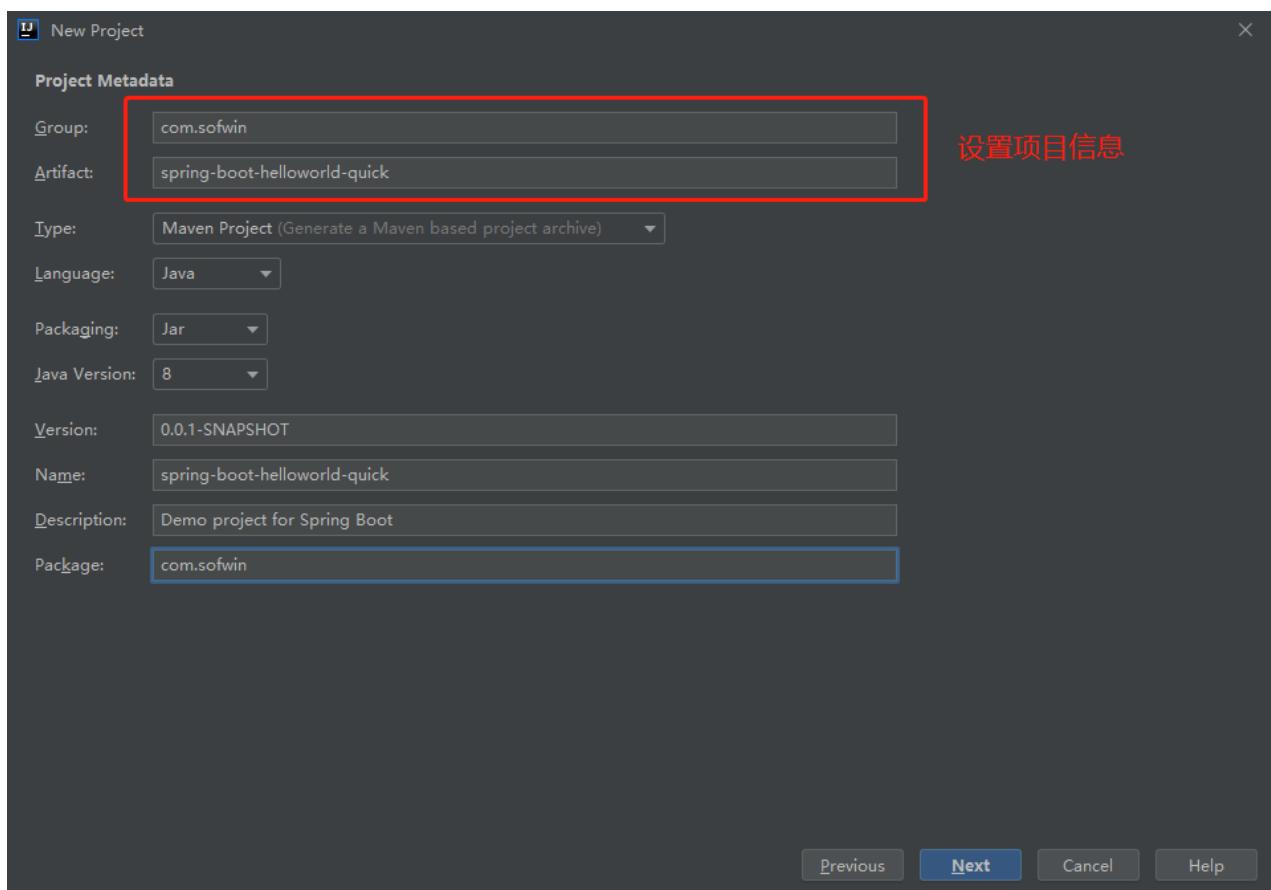
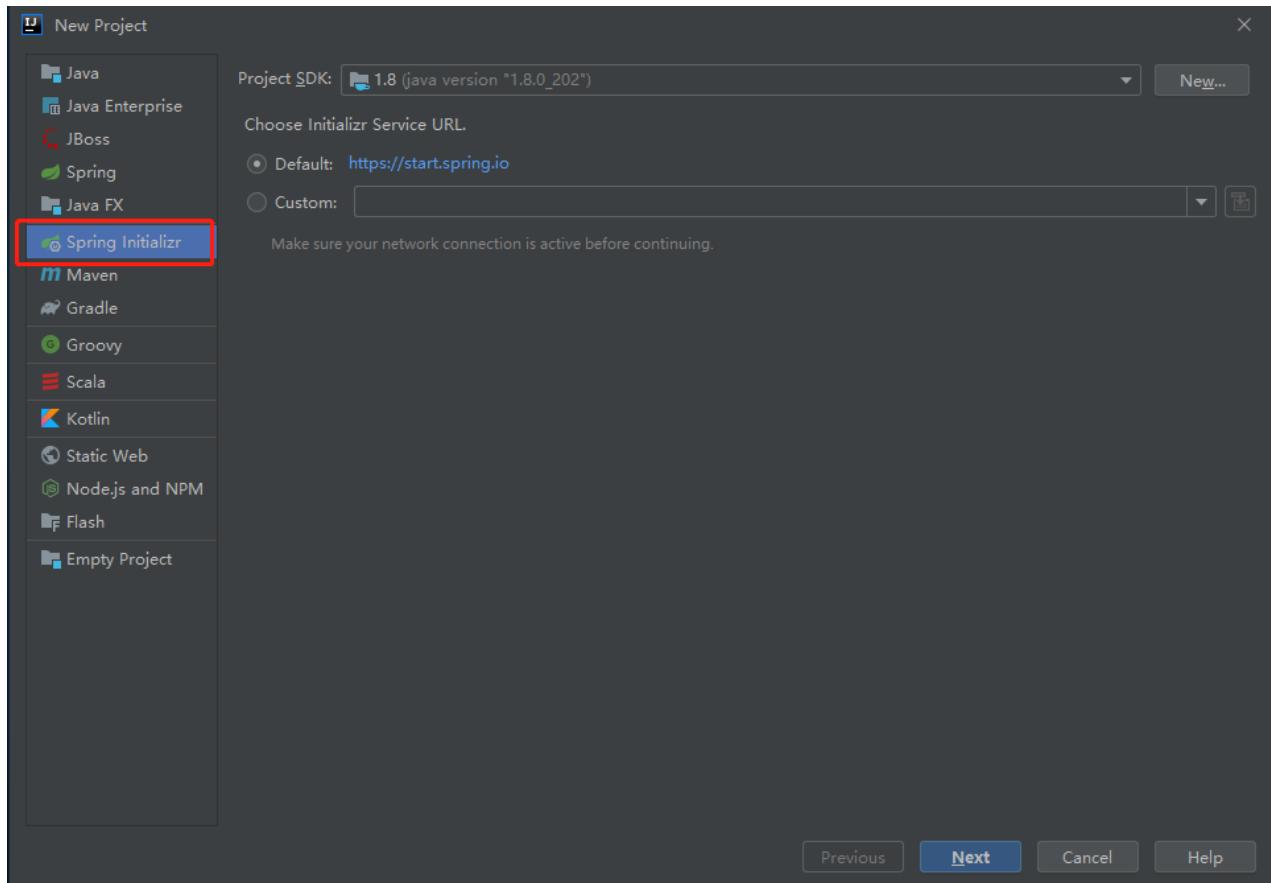
项目打包：

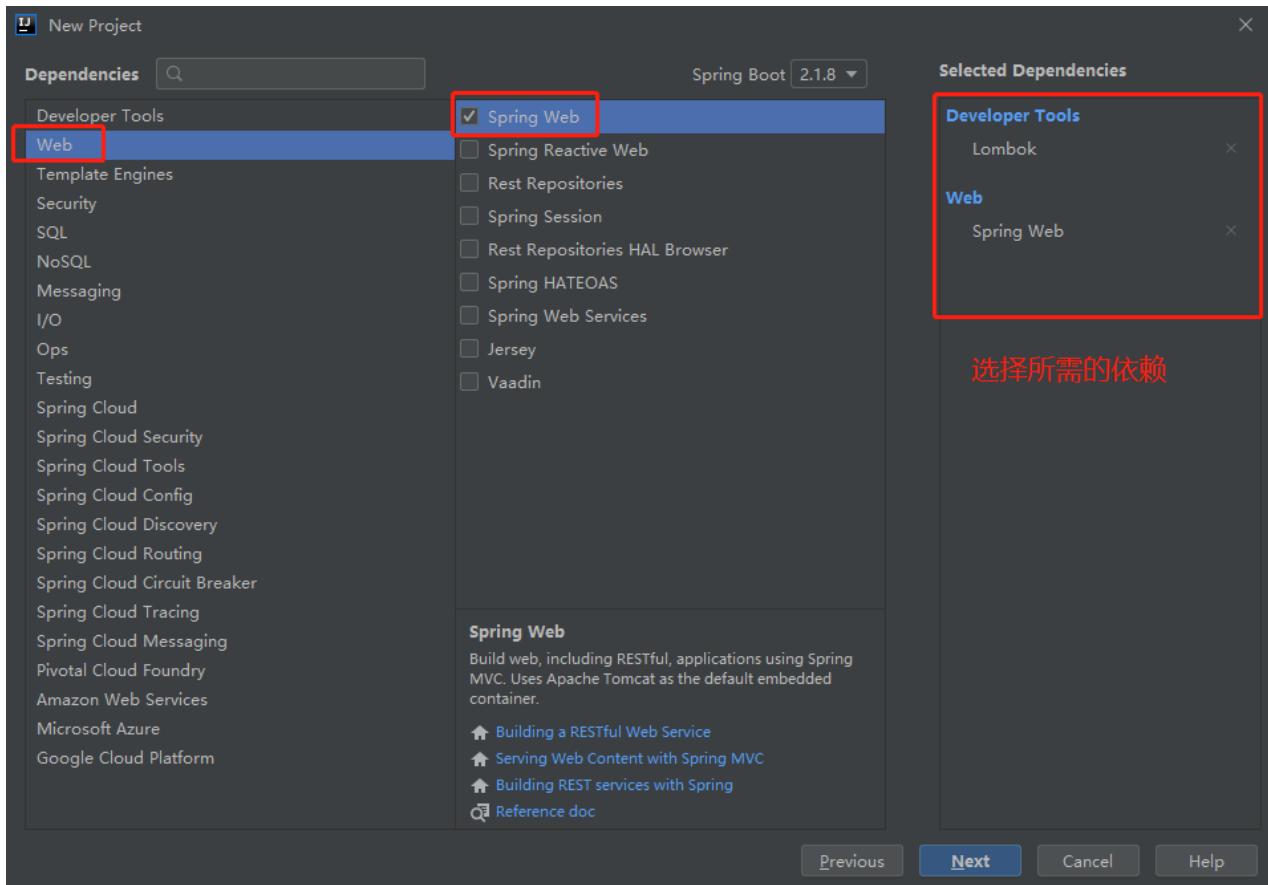


## 项目部署：

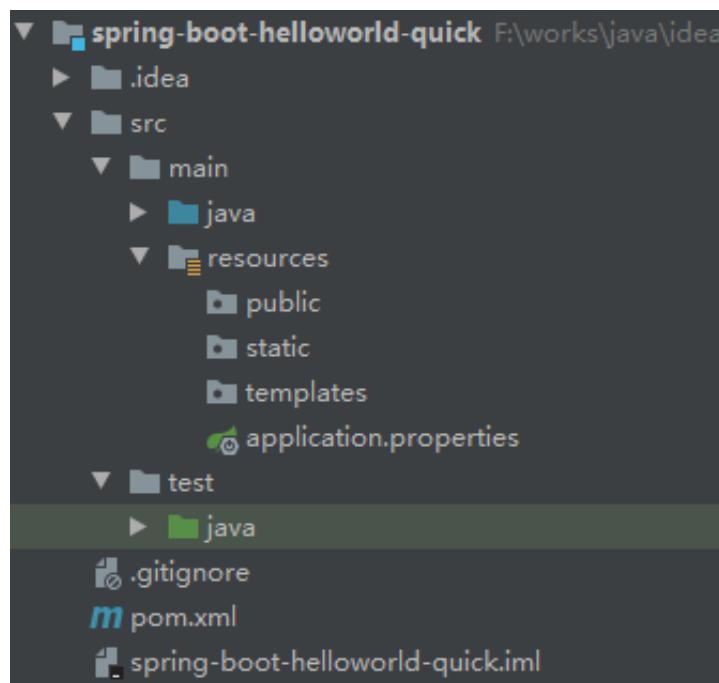
将打包好的文件 `spring-boot-helloworld-1.0-SNAPSHOT.jar` 拷贝到部署的目录，运行命令：`java -jar spring-boot-helloworld-1.0-SNAPSHOT.jar` 即可启动。

## 2.2.2 向导创建





### 2.2.3 项目结构



```
1   └──spring-boot-helloworld-quick
2       └──src
3           ├──main
4               ├──java
5               └──resources
6                   ├──public
7                   ├──static
8                   └──templates
9           └──test
10              ├──java
11              └──resources
```

#### resources 文件夹说明：

- **public:** 用于存放html、css、js等静态文件；
- **static:** 用于存放css、js等静态文件；
- **templates:** 用于存放模板文件；
- **application.properties:** 项目配置文件。

**test** 目录中也可以包含 **resources** 文件夹，但只对测试用例有效。

## 2.2.4 项目分析

可以注意到：在这个 **hello world** 项目中，没有任何配置。是如何实现的呢？我们来逐步探究一下这背后的原理。

### 2.2.4.1 父项目

在 **pom.xml** 文件中，我们的项目继承了Spring Boot父项目：

```
1  <!-- 1. 继承 Spring Boot 父项目 -->
2  <parent>
3      <groupId>org.springframework.boot</groupId>
4      <artifactId>spring-boot-starter-parent</artifactId>
5      <version>2.3.6.RELEASE</version>
6  </parent>
```

而 **spring-boot-starter-parent** 的父项目是：**spring-boot-dependencies**

```
1  <parent>
2      <groupId>org.springframework.boot</groupId>
3      <artifactId>spring-boot-dependencies</artifactId>
4      <version>2.3.6.RELEASE</version>
5      <relativePath>../../spring-boot-dependencies</relativePath>
6  </parent>
```

**spring-boot-dependencies** 中定义了Spring Boot依赖的各个组件的版本：

```
1  ...
2  <properties>
3      <activemq.version>5.15.13</activemq.version>
```

```
4      <antlr2.version>2.7.7</antlr2.version>
5      <appengine-sdk.version>1.9.83</appengine-sdk.version>
6      <artemis.version>2.12.0</artemis.version>
7      <aspectj.version>1.9.6</aspectj.version>
8      <assertj.version>3.16.1</assertj.version>
9      <atomikos.version>4.0.6</atomikos.version>
10     <awaitility.version>4.0.3</awaitility.version>
11     <bitronix.version>2.1.4</bitronix.version>
12     <build-helper-maven-plugin.version>3.1.0</build-helper-maven-plugin.version>
13     <byte-buddy.version>1.10.18</byte-buddy.version>
14     <caffeine.version>2.8.6</caffeine.version>
15     <cassandra-driver.version>4.6.1</cassandra-driver.version>
16     <classmate.version>1.5.1</classmate.version>
17     <commons-codec.version>1.14</commons-codec.version>
18     <commons-dbcop2.version>2.7.0</commons-dbcop2.version>
19     <commons-lang3.version>3.10</commons-lang3.version>
20     <commons-pool.version>1.6</commons-pool.version>
21     <commons-pool2.version>2.8.1</commons-pool2.version>
22     ...
...
```

`spring-boot-dependencies` 是Spring Boot版本仲裁中心，在这里定义过的依赖，我们导入时不需要写版本号。

#### 2.2.4.2 启动器

在 `pom.xml` 文件中，我们的导入了 `spring-boot-starter-web` 场景启动器的依赖：

```
1  <!-- 2. 管理依赖，导入starter（启动器） -->
2  <dependencies>
3      <dependency>
4          <groupId>org.springframework.boot</groupId>
5          <artifactId>spring-boot-starter-web</artifactId>
6      </dependency>
7  </dependencies>
```

`spring-boot-starter-web` 导入了该场景启动器的所有组件依赖。

Spring Boot官方抽象、提取了众多应用场景，封装成了一个个场景启动器。我们在开发过程中，只需导入不同的驱动器即可。详情可参考：<https://docs.spring.io/spring-boot/docs/2.3.6.RELEASE/reference/html/using-spring-boot.html#using-boot-starter>。

#### 2.2.4.3 启动类

```
1  /**
2   * Spring Boot 应用启动类
3   */
4 @SpringBootApplication // 添加注解, 标注Spring Boot应用
5 public class Application {
6
7     public static void main(String[] args) {
8         SpringApplication.run(Application.class, args);
9     }
10
11 }
```

`@SpringBootApplication` 是用于标注启动类的注解（启动类也是主配置类），我们只需要运行启动类的 `main` 方法就能启动、运行应用程序。`@SpringBootApplication` 是一个组合注解，由 `@SpringBootConfiguration` 及 `@EnableAutoConfiguration` 组合而成：

```
1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Inherited
5  @SpringBootConfiguration
6  @EnableAutoConfiguration
7  @ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class),
8          @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
9  public @interface SpringBootApplication {
10      ...
11 }
```

### `@SpringBootConfiguration:`

其中 `@SpringBootConfiguration` 标注在某个类上时，表示这个类时Spring Boot的配置类。它包含了 `@Configuration` 注解。

```
1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Configuration
5  public @interface SpringBootConfiguration {
6
7 }
```

`@Configuration` 是Spring底层提供的用于标注配置类的注解。

### `@EnableAutoConfiguration:`

`@EnableAutoConfiguration` 注解用于告诉Spring Boot开启自动配置，它包含了：`@AutoConfigurationPackage`、`@Import(AutoConfigurationImportSelector.class)`。

```
1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Inherited
5  @AutoConfigurationPackage
6  @Import(AutoConfigurationImportSelector.class)
7  public @interface EnableAutoConfiguration {
8      String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
9
10     Class<?>[] exclude() default {};
11
12     String[] excludeName() default {};
13 }
```

其中 `@AutoConfigurationPackage` 通过 `@Import(AutoConfigurationPackages.Registrar.class)` 给容器中导入组件，`@Import` 是 Spring底层提供的注解，可以根据逻辑规则导入所需的组件。

```
1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Inherited
5  @Import(AutoConfigurationPackages.Registrar.class)
6  public @interface AutoConfigurationPackage {
7
8 }
```

而 `AutoConfigurationPackages.Registrar` 则会扫描主配置类（即被 `@SpringBootApplication` 标注的类）所在包及其下所有子包，将里面所有组件导入到IOC容器中去。

```
1  static class Registrar implements ImportBeanDefinitionRegistrar,
DeterminableImports {
2
3      @Override
4      public void registerBeanDefinitions(AnnotationMetadata metadata,
BeanDefinitionRegistry registry) {
5          register(registry, new PackageImport(metadata).getPackageName());
6      }
7
8      @Override
9      public Set<Object> determineImports(AnnotationMetadata metadata) {
10         return Collections.singleton(new PackageImport(metadata));
11     }
12
13 }
```

所以，如果我们定义的组件不在启动类所长的包和子包中，则该组件不会被Spring扫描、导入。

`@EnableAutoConfiguration` 中还包含了：`@Import(AutoConfigurationImportSelector.class)`，`AutoConfigurationImportSelector`会给IOC容器中导入很多自动配置类（`XXXAutoConfiguration`），也即给容器中导入当前所有场景启动器所有组件，并配置好。这样就免去了我们手动配置的工作。

Spring Boot在启动时，会搜索类路径下 `"META-INF/spring.factories"` 文件，获取 `EnableAutoConfiguration` 配置项的值，并将这些值作为自动配置类导入到IOC容器。

## 2.3 SpringBoot配置

教学目标：

1. 了解两种配置文件的区别；
2. 掌握yml配置文件的编写；
3. 掌握多环境配置文件；
4. 了解SpringBoot自动配置原理。

### 2.3.1 配置文件类型

配置文件的作用是用来修改Spring Boot的默认配置项。

Spring Boot的配置文件名都是 `application`，但有两种类型：

- `application.properties`
- `application.yml`

`properties` 配置文件我们都比较熟悉，这里我们主要看看 `yml` 配置文件：YAML。

YAML 是 "YAML Ain't a Markup Language" (YAML不是一种标记语言) 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" (仍是一种标记语言)，但为了强调这种语言以数据做为中心，而不是以标记语言为重点，而用反向缩略语重命名。

YAML：以数据为中心，比xml、json等更适合作配置文件。

YAML：配置示例：

```
1 server:  
2   port: 8080  
3   tomcat:  
4     uri-encoding: UTF-8
```

YAML 基本语法：

- K:(空格)V 表示一个键值对，需要注意冒号后面必须要有一个空格；
- 以空格（2个）的缩进来控制层级关系，左对齐的一列配置都属于同一个层级；
- 属性名、属性值，对大小写敏感；

值的写法：

- 字面值：数字、字符串（不需要加双引号）等  
k: v 直接写，如果字符串强加引号，注意：双引号不会转义字符（即会按规则解析换号等操作），单引号则会转义字符（即按文本内容原样输出）；
- 对象、Map：

以YAML缩进风格写，把对象的属性当成下一层配置，如

```
1 user:  
2   username: 张三  
3   age: 22
```

或用花括号写在同一行，如：

```
1 user: {username: zhansgan, age: 22}
```

- 数组、集合：

用减号加值表示元素，如：

```
1 pets:  
2   - cat  
3   - dog  
4   - pig
```

或用方括号写在同一行，如：

```
1 pets: [cat, dog, pig]
```

项目中可以同时存在 `yml` 两种类型文件，但如果两个文件中有相同的配置项，则以 `properties` 文件为主。

练习：

尝试yml配置文件的多种配置

## 2.3.2 配置文件值注入

pom文件添加依赖：

```
1 <dependency>  
2   <groupId>org.springframework.boot</groupId>  
3   <artifactId>spring-boot-configuration-processor</artifactId>  
4   <optional>true</optional>  
5 </dependency>
```

准备配置文件 `application.yml`：

```
1 server:
2     port: 8080
3
4 person:
5     user-name: 张三
6     age: 22
7     books:
8         - 计算机网络
9         - Java编程思想
10    phone:
11        brand: apple
12        price: 7000.00
```

## 准备配置类：

```
1 @Data
2 public class Phone {
3     private String brand;
4     private Double price;
5 }
```

## 注意：

- 只有被IOC容器管理的组件才能使用自动配置功能
- 通过 `@ConfigurationProperties` 注解配置组件属性与配置文件的映射

```
1 /**
2  * @ConfigurationProperties: 告诉Spring Boot, 将本实例的所有属性和配置文件中相关的配置进行
3  * 绑定
4  *      prefix = "person": 指定配置的前缀
5  */
6 @Component
7 @ConfigurationProperties(prefix = "person")
8 public class Person {
9     private String userName;
10    private Integer age;
11    private List<String> books;
12    private Phone phone;
13
14    public Person() {
15    }
16
17    public String getUserName() {
18        return userName;
19    }
20
21    public void setUserName(String userName) {
22        this.userName = userName;
23    }
24}
```

```
23
24     public Integer getAge() {
25         return age;
26     }
27
28     public void setAge(Integer age) {
29         this.age = age;
30     }
31
32     public List<String> getBooks() {
33         return books;
34     }
35
36     public void setBooks(List<String> books) {
37         this.books = books;
38     }
39
40     public Phone getPhone() {
41         return phone;
42     }
43
44     public void setPhone(Phone phone) {
45         this.phone = phone;
46     }
47
48     @Override
49     public String toString() {
50         return "Person{" +
51                 "userName='" + userName + '\'' +
52                 ", age=" + age +
53                 ", books=" + books +
54                 ", phone=" + phone +
55                 '}';
56     }
57 }
```

测试类：

```

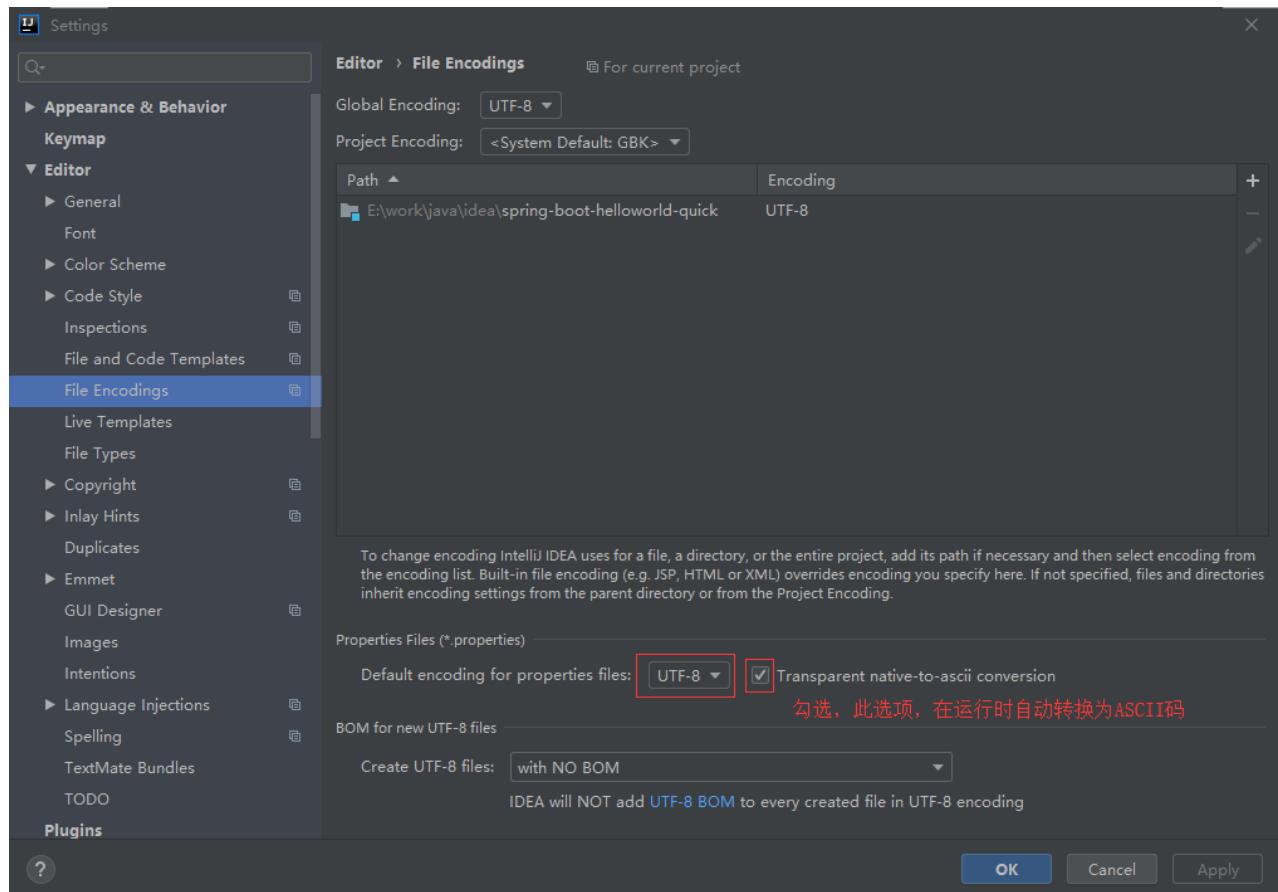
1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class ApplicationTests {
4
5      @Autowired
6      Person person;
7
8      @Test
9      public void contextLoads() {
10         System.out.println(person);
11     }
12
13 }

```

通过测试，我们可以看到SpringBoot会自动完成属性、配置的绑定。

另外，我们也可以观察到：配置文件中的 `user-name` 配置可以绑定到 `userName` 属性，这称之为松绑定（松散语法）。允许将类似：`user-name`、`user_name`、`USER_NAME` 等绑定到 `userName` 属性。

如果出现中文乱码，需要修改IDEA配置：



我们也可以直接通过 `@Value` 注解手动配置属性值：

```

1  @Component
2  //@ConfigurationProperties(prefix = "person")
3  public class Person {
4
5      @Value("${person.user-name}") // 手动绑定配置
6      private String userName;
7
8      @Value("#{11*22}") // 使用SpEL表达式
9      private Integer age;
10     private List<String> books;
11     private Phone phone;
12     ...
13 }

```

### **@ConfigurationProperties 与 @Value 取值的区别:**

	<b>@ConfigurationProperties</b>	<b>@Value</b>
功能	批量注入配置文件中的属性	一个一个地指定
松散绑定	支持	支持
SpEL	不支持	支持
JSP303数据校验	支持	不支持
复杂类型封装	支持	不支持

一般来说，@ConfigurationProperties用于做全局配置，@Value用于临时取值。

### **@PropertySource 注解可用于指定加载的配置文件:**

```

1  @Component
2  @PropertySource(value = {"classpath:person.properties"})
3  @ConfigurationProperties(prefix = "person")
4  public class Person {
5
6      @Value("${person.user-name}") // 手动绑定配置
7      private String userName;
8
9      @Value("#{11*22}") // 使用Spring表达式
10     private Integer age;
11     private List<String> books;
12     private Phone phone;
13     ...
14 }

```

Spring Boot没有Spring的XML配置文件，`@ImportSource`注解可用于指定加载Spring的配置文件。

Spring Boot不建议使用Spring的XML配置文件，推荐使用`@Configuration`以注解的方式配置Bean.

### 2.3.3 配置文件占位符

Spring Boot允许在配置文件中使用占位符，如： `${random.int}`

### 2.3.4 多环境配置文件

#### 1. 多Profile文件

配置文件名可以是：`application-dev.yml`、`application-test.yml`、`application-prod.yml`

#### 2. yml支持多文档块

```
1  server:
2      port: 8080
3  spring:
4      profiles:
5          active: dev
6
7  ---
8  spring:
9      profiles: dev
10
11 server:
12     port: 8081
13
14 ---
15 spring:
16     profiles: test
17
18 server:
19     port: 8082
20
21 ---
22 spring:
23     profiles: prod
24
25 server:
26     port: 8083
```

#### 3. 激活指定的profile

`spring.profiles.active=dev`

或

```
1  spring:  
2    profiles:  
3      active: dev
```

也可以在运行时，指定命令行参数：`--spring.profiles.active=dev`；

也可以指定JVM参数：`-Dspring.profiles.active=dev`

### 2.3.5 配置文件加载顺序

Spring Boot会一次扫描以下路径，来查找配置文件：

- file:./config/
- file:./
- classpath:/config/
- classpath:/

以上路径，优先级从高到低。所有配置文件都会被加载，但高优先级的配置文件可能会覆盖低优先级的配置。

可以通过 `spring.config.location=F:/xxx/xxx.properties` 来指定配置文件，所有配置文件都会起作用，形成互补配置。

项目打包之后，还可以通过命令行参数指定配置文件 `--spring.config.location=F:/xxx/xxx.properties`。

Spring Boot配置文件的加载顺序：

1. 命令行参数
2. 来自java:comp/env的JNDI属性
3. Java系统属性（`System.getProperties()`）
4. 操作系统环境变量
5. `RandomValuePropertySource`配置的`random.*`属性值
6. jar包外部的`application-{profile}.properties`或`application.yml`(带`spring.profile`)配置文件
7. jar包内部的`application-{profile}.properties`或`application.yml`(带`spring.profile`)配置文件
8. jar包外部的`application.properties`或`application.yml`(不带`spring.profile`)配置文件
9. jar包内部的`application.properties`或`application.yml`(不带`spring.profile`)配置文件
10. `@Configuration`注解类上的`@PropertySource`
11. 通过`SpringApplication.setDefaultProperties`指定的默认属性

以上顺序优先级从高到低，多个配置文件可以互补。具体可参考：<https://docs.spring.io/spring-boot/docs/2.3.6.RELEASE/reference/html/spring-boot-features.html#boot-features-external-config>

配置文件能够配置的属性列表，可参考：<https://docs.spring.io/spring-boot/docs/2.3.6.RELEASE/reference/html/appendix-application-properties.html#common-application-properties>

## 2.3.6 自动配置原理

扫描配置自动配置类：

1. Spring Boot启动时，加载启动类（主配置类）。启动类注解 `@SpringBootApplication` 中包含了 `@EnableAutoConfiguration`，从而开启了自动配置；
2. `@EnableAutoConfiguration` 注解包含了 `@Import(AutoConfigurationImportSelector.class)`，即通过 `AutoConfigurationImportSelector` 选择器导入组件；
3. `AutoConfigurationImportSelector` 类中定义了 `selectImports` 方法，它会继续调用 `getAutoConfigurationEntry` 方法；
4. `getAutoConfigurationEntry` 方法会调用 `List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);` 来获取配置列表；
5. `getCandidateConfigurations` 方法又通过 `List<String> configurations = SpringFactoriesLoader.loadFactoryNames(SpringFactoriesLoaderFactoryClass(), getBeanClassLoader());` 获取配置列表；
6. 追踪后我们会发现，`SpringFactoriesLoader.loadFactoryNames` 会扫描所有类路径下所有 jar包的 `META-INF/spring.factories` 文件，并封装成 properties 对象；
7. `AutoConfigurationImportSelector` 会从 properties 对象中获取 key 为 `org.springframework.boot.autoconfigure.EnableAutoConfiguration` 的配置。

小结：Spring Boot 会扫描类路径下所有 `META-INF/spring.factories` 文件，并将其中所有 `org.springframework.boot.autoconfigure.EnableAutoConfiguration` 配置项的值导入到 IOC 容器中。

```
1 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
2 org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfigura-
tion,\ 
3 org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\ 
4 org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\ 
5 org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\ 
6 org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\ 
7 org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\ 
8 ...
```

每一个配置类进行自动配置，例如：

```
1 @Configuration // 这是一个配置类
2 @EnableConfigurationProperties(HttpProperties.class) // 启用指定类的
ConfigurationProperties 功能，参见：HttpProperties 源码
3 @ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.SERVLET) // 
判断当前项目是否为 web 应用，如果是则配置生效
4 @ConditionalOnClass(CharacterEncodingFilter.class) // 判断项目中是否包含
CharacterEncodingFilter 这个类，如果有则配置生效
5 @ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled",
matchIfMissing = true) // 判断配置文件中是否包含 spring.http.encoding 配置项
6 public class HttpEncodingAutoConfiguration {
7     private final HttpProperties.Encoding properties;
```

```

8
9     /**
10      * 只有一个带参构造器时，会自动注入参数
11      */
12     public HttpEncodingAutoConfiguration(HttpProperties properties) {
13         this.properties = properties.getEncoding();
14     }
15
16     /**
17      * 向IOC容器中添加组件
18      */
19     @Bean
20     @ConditionalOnMissingBean // 容器中不包含characterEncodingFilter对象则生效
21     public CharacterEncodingFilter characterEncodingFilter() {
22         CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();
23         filter.setEncoding(this.properties.getCharset().name());
24
25         filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));
26
27         filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));
28         return filter;
29     }
30     ...
31 }
32 // 配置文件能够配置的参数，都定义在`XXXProperties`类中。
33 @ConfigurationProperties(prefix = "spring.http") // 从配置文件中获取指定值与属性进行绑定
34 public class HttpProperties {
35     ...
36 }
```

也即：根据当前不同的判断条件，来决定配置类是否生效。一旦这个配置类生效，他就会自动向IOC容器中添加组件，而这些组件的属性是和配置文件的配置绑定的。从而实现了自动配置。

## 总结：

1. Spring Boot启动时会加载大量的自动配置类：`XXXAutoConfiguration`；
2. 这些自动配置类的属性有默认值，大部分情况下能满足我们的使用需求
3. 当我们需要修改自动配置类的属性时，只需要在配置文件中添加相关配置项，而这些配置项都包含在 `XXXXXXProperties` 类中。

## 2.4 SpringBoot日志

教学目标：

1. 了解常见日志框架的关系；
2. 掌握SL4J适配原理；
3. 掌握SpringBoot日志配置；
4. 了解Lombok日志操作。

### 2.4.1 日志框架

常见日志框架：

JUL、JCL、JBoss-logging、logback、log4j、log4j2、slf4j

框架分类：

日志门面（抽象层）	日志实现
JCL(Jakarta Commons Logging)、SLF4J(Simple Logging Facade for Java)、jboss-logging	Log4j、Logback、Log4j2、JUL(java.util.logging)

在项目中使用日志框架，就是挑选一个日志门面、在挑选一个日志实现。

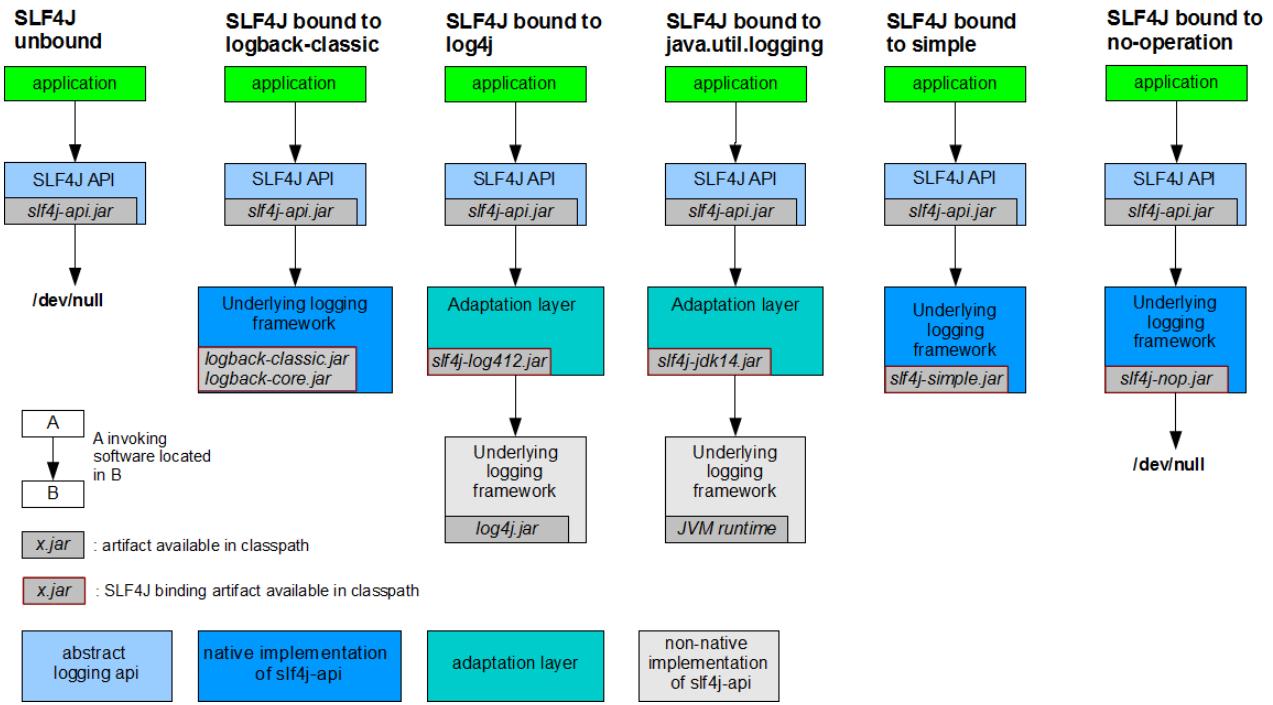
SLF4J、Log4j、Logback 是同一作者的作品，Log4j2是apache基金会的项目。

Spring框架底层默认使用的是JCL，即apache基金会的 **commons logging**。

Spring Boot 默认使用 Logback。

记录日志时，我们应该使用 **日志门面** 中定义的方法，而具体的实现由项目中导入的实现框架来决定。

SL4J适配图：



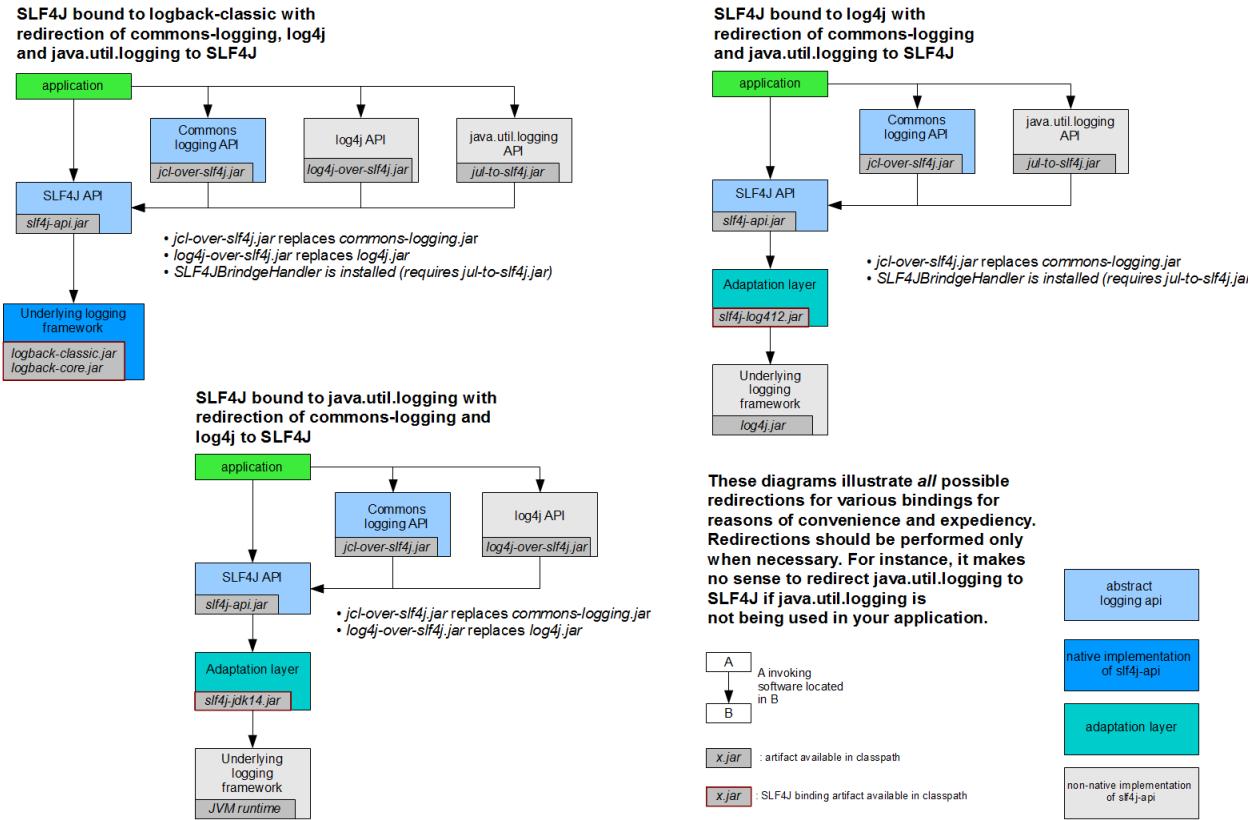
每个日志实现框架都有自己的配置文件，开发过程中要根据选择的实现框架编写对应的的配置文件。

### 日志冲突：

由于不同的框架（Spring、Hibernate、MyBatis）默认使用了不同的日志框架，如果不做处理可能会导致冲突和混乱。

### 解决办法：

1. 在maven的pom文件中排除原有的日志框架依赖；
2. 用中间包替换原有日志实现包；
3. 导入我们需要的日志实现框架依赖



## Spring Boot日志分析：

当前2.x版本的Spring Boot是基于Spring 5.x的。Spring4.x用的是原生的jcl日志组件，Spring5.x用的自定义的 `spring-jcl` 日志组件。在 `spring-boot-starter-logging` 中Spring对日志框架做了类似sl4j所做的适配工作。

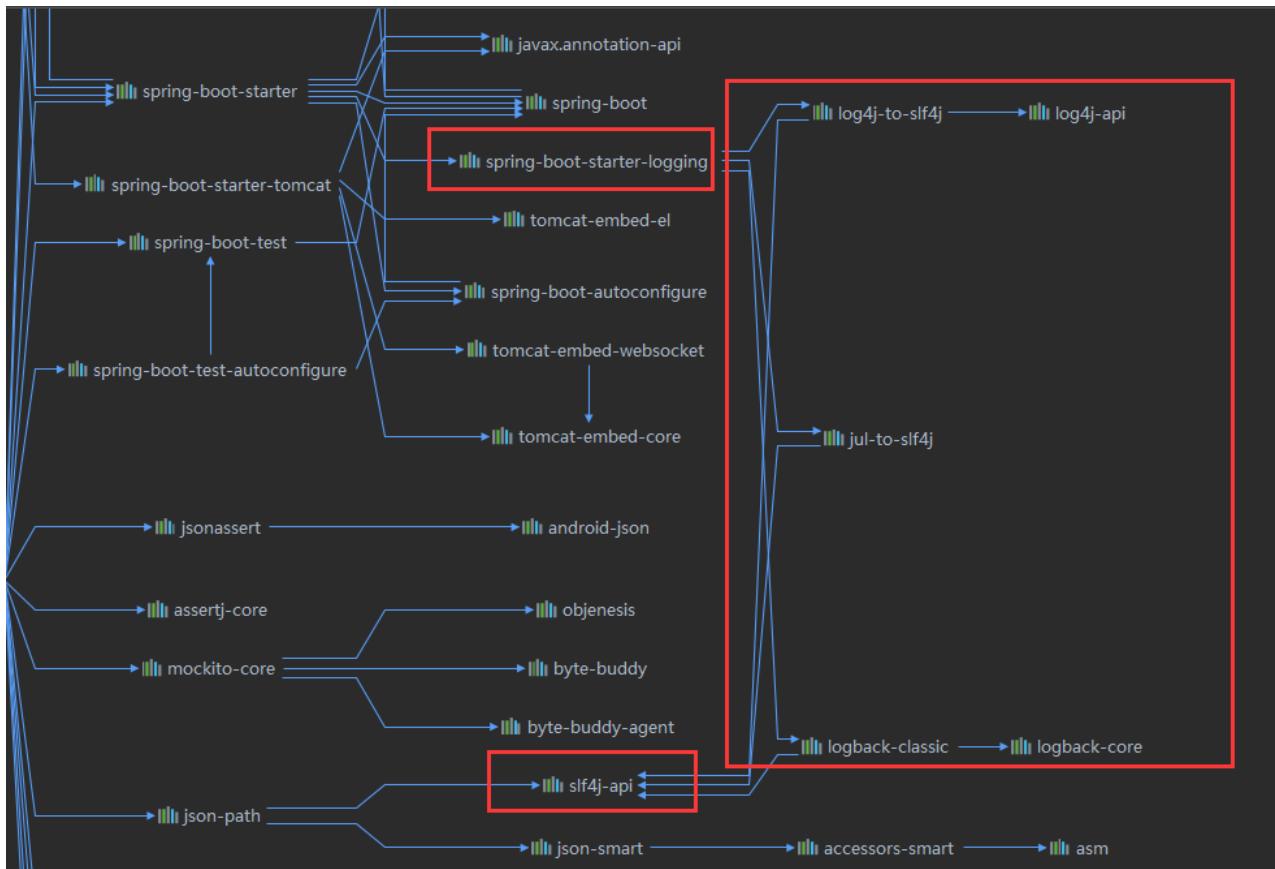
Spring Boot通过 `spring-boot-starter-logging` 启动器来管理日志框架：

```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-logging</artifactId>
4 </dependency>

```

`spring-boot-starter-logging` 导入了 `jul-to-slf4j`、`log4j-to-slf4j` 等中间包，最终使用 `logback-classic` 的日志实现。



## 2.4.2 日志配置

Spring Boot已经配置好了日志框架， 默认是info级别：

```

1 import org.slf4j.Logger;
2 import org.slf4j.LoggerFactory;
3
4 public class HelloWorld {
5     public static void main(String[] args) {
6         // 日志记录器
7         Logger logger = LoggerFactory.getLogger(HelloWorld.class);
8
9         logger.info("Hello World");
10    }
11 }

```

可以在配置文件中，修改日志输出级别。如全局日志输出级别：

```
1 logging.level.root: debug
```

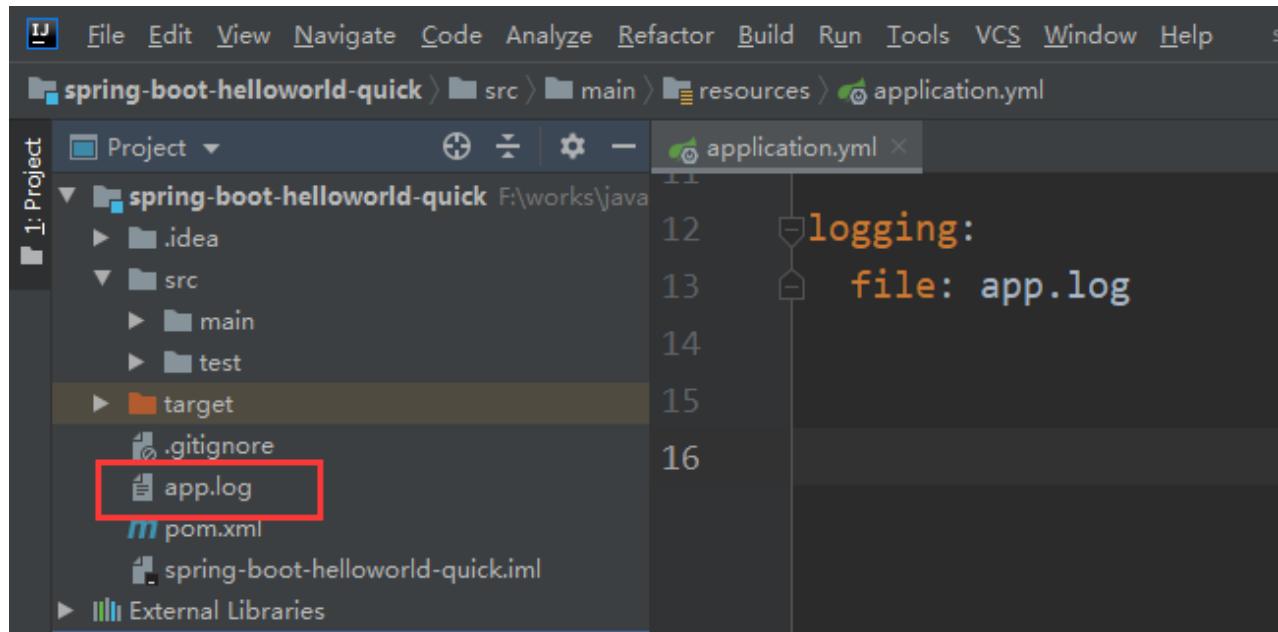
也可以配置指定包名下的日志输出级别，如：

```
1 logging.level.cn.sofwin: debug
```

日志文件：

```
1  logging:  
2    file: app.log
```

将会在当前项目文件夹下创建日志文件，如：



完整路径的日志文件：

```
1  logging:  
2    file: f:/log/app.log
```

Spring Boot 将会把日志输出到指定的目录。

日志目录：

```
1  logging:  
2    path: f:/log
```

此时，Spring Boot 将会向指定的目录输出日志，文件名为： `spring.log` 。

建议：

- 推荐仅配置日志目录，而不配置日志文件，以保持默认的日志文件名（方便后续日志分析）；

日志格式：

```
1  logging:
2    pattern:
3      console: '%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} : %msg%n'
4      file: '%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} : %msg%n'
```

## 自定义配置文件

在类路径下，放置日志实现框架的配置文件即可，如：`logback.xml`。此时，Spring Boot将不再使用默认的日志配置。参考：<https://docs.spring.io/spring-boot/docs/2.3.6.RELEASE/reference/html/spring-boot-features.html#boot-features-custom-log-configuration>

### 2.4.3 课后作业

创建一个SpringBoot演示项目：`sw-boot-test03`

要求：

1. 将所有日志文件输出到：`f:/app/log` 目录
2. 配置日志格式，需要输出类名、方法名以及行号
3. 使用 `lombok` 输入日志

## 2.5 SpringBoot数据访问

教学目标：

- 1. 掌握JDBC与数据源的配置与使用；
- 2. 掌握SpringBoot集成MyBatis；
- 3. 掌握SpringBoot集成Redis；
- 4. 了解数据库分库、分表。

### 2.5.1 JDBC

1. 引入Maven依赖

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-jdbc</artifactId>
4 </dependency>
5
6 <dependency>
7   <groupId>mysql</groupId>
8     <artifactId>mysql-connector-java</artifactId>
9     <scope>runtime</scope>
10 </dependency>
```

## 2. 修改配置文件，添加以下配置：

```
1 spring:
2   datasource:
3     username: root
4     password: 123456
5     url: jdbc:mysql://127.0.0.1:3306/sw_jdbc_demo
6     driver-class-name: com.mysql.cj.jdbc.Driver
7     # hikari数据库连接池
8     hikari:
9       minimum-idle: 2 #最小空闲连接数量
10      idle-timeout: 180000 #空闲连接存活最大时间， 默认600000 (10分钟)
11      maximum-pool-size: 5 #连接池最大连接数， 默认是10
12      auto-commit: true #此属性控制从池返回的连接的默认自动提交行为， 默认值：true
13      max-lifetime: 1800000 #此属性控制池中连接的最长生命周期， 值0表示无限生命周期， 默认1800000即30分钟
14      connection-timeout: 30000 #数据库连接超时时间， 默认30秒， 即30000
15      connection-test-query: SELECT 1
```

## 3. 测试连接

```
1 @RunWith(SpringRunner.class)
2 @SpringBootTest
3 public class SpringBootApplicationTests {
4
5   /**
6    * SpringBoot 1.x 默认使用的是：org.apache.tomcat.jdbc.pool.DataSource
7    * SpringBoot 2.x 默认使用的是：com.zaxxer.hikari.HikariDataSource
8    * 可以在配置文件中修改，本实例中，使用了Druid数据源
9    */
10  @Autowired
11  DataSource dataSource;
12
13  @Autowired
14  JdbcTemplate jdbcTemplate;
15
16  /**
17   * 测试基础JDBC
18   * @throws SQLException
19   */
20  @Test
```

```

21     public void testJdbc() throws SQLException {
22         Connection conn = dataSource.getConnection();
23         Statement stmt = conn.createStatement();
24         ResultSet rs = stmt.executeQuery("select * from user");
25         while (rs.next()){
26             System.out.println(rs.getString("username"));
27         }
28     }
29
30     /**
31      * 测试JdbcTemplate
32      */
33     @Test
34     public void testJdbcTemplate() {
35         List<Map<String, Object>> users = jdbcTemplate.queryForList("select
36         * from user;");
37         System.out.println(users);
38     }

```

## 2.5.2 MyBatis

### 2.5.2.1 注解版

#### 1. 创建测试数据库及表

```

1   CREATE TABLE user(
2       id INT NOT NULL AUTO_INCREMENT COMMENT '序号',
3       username VARCHAR(64) NOT NULL COMMENT '用户名',
4       password VARCHAR(64) NOT NULL COMMENT '密码',
5       nickname VARCHAR(64) COMMENT '昵称',
6       add_time DATETIME COMMENT '添加时间',
7       PRIMARY KEY (id)
8   );

```

#### 2. 添加Maven依赖

```

1 <!-- MyBatis启动器 -->
2 <dependency>
3     <groupId>org.mybatis.spring.boot</groupId>
4     <artifactId>mybatis-spring-boot-starter</artifactId>
5 </dependency>
6
7 <!-- 数据库驱动 -->
8 <dependency>
9     <groupId>mysql</groupId>
10    <artifactId>mysql-connector-java</artifactId>
11    <scope>runtime</scope>
12 </dependency>
13
14 <!-- Spring测试 -->

```

```
15 <dependency>
16     <groupId>org.springframework.boot</groupId>
17     <artifactId>spring-boot-starter-test</artifactId>
18     <scope>test</scope>
19 </dependency>
```

注意： `mybatis-spring-boot-starter` 依赖了 `spring-boot-starter-jdbc`，因此，我们无需重复引入 `spring-boot-starter-jdbc` 启动器。

### 3. 修改配置文件，在主配置文件中，配置数据源、日志：

```
1 spring:
2   datasource:
3     username: root
4     password: 123456
5     url: jdbc:mysql://127.0.0.1:3306/sw_jdbc_demo
6     driver-class-name: com.mysql.cj.jdbc.Driver
7     # hikari数据库连接池
8     hikari:
9       minimum-idle: 2 #最小空闲连接数量
10      idle-timeout: 180000 #空闲连接存活最大时间，默认600000（10分钟）
11      maximum-pool-size: 5 #连接池最大连接数，默认是10
12      auto-commit: true #此属性控制从池返回的连接的默认自动提交行为，默认值：true
13      max-lifetime: 1800000 #此属性控制池中连接的最长生命周期，值0表示无限生命周期， 默认1800000即30分钟
14      connection-timeout: 30000 #数据库连接超时时间，默认30秒，即30000
15      connection-test-query: SELECT 1
16
17    # 配置打印MyBatis执行的SQL
18    logging.level.cn.sofwin.hello.dao: debug
```

### 4. 创建实体类

```
1 @Data
2 public class User {
3     private Integer id;
4     private String username;
5     private String password;
6     private String nickname;
7 }
```

### 5. 创建mapper

```
1 @Mapper
2 public interface UserDao {
3
4     @Select("select * from user")
5     List<User> selectAll();
6
7     @Select("select * from user where id = #{id}")
8     User selectById(Integer id);
9 }
```

```
10     @Options(useGeneratedKeys = true, keyProperty = "id")
11     @Insert("insert into user(username, password, nickname) values (#
12     {username}, #{password}, #{nickname})")
13     void insert(User user);
14
15     @Update("update user set password = #{password}, nickname = #{nickname}
16     where id = #{id}")
17     void update(User user);
18
19     @Delete("select from user where id = #{id}")
20     void deleteById(Integer id);
21 }
```

## 6. 编写测试类

```
1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class SpringBootApplicationTests {
4
5      @Autowired
6      UserDao userDao;
7
8      @Test
9      public void testMyBatis()  {
10         List<User> userList = userDao.selectAll();
11         System.out.println(userList);
12     }
13
14 }
```

注意：需要在 `application.yml` 配置文件中配置jdbc数据源。

注解版的MyBatis无需任何额外的配置就可以使用。这是因为 **MybatisAutoConfiguration** 已经帮我们完成了自动配置。

但在实际的开发过程中，我们是需要将业务 `sql` 独立出来，放置到一个xml文件中的。这样做的好处是：①更加集中的管理SQL；②后期维护线上代码，如果仅仅是修改业务SQL，我们不需要重新编译、打包整个项目。

### 2.5.2.2 映射文件版

要将SQL文件独立出来，我们需要做一些调整：

1. 添加新的测试表

```
1  CREATE TABLE article(
2      id INT NOT NULL AUTO_INCREMENT,
3      title VARCHAR(64),
4      content TEXT,
5      add_time DATETIME,
6      read_count INT,
7      PRIMARY KEY (id)
8  );
```

## 2. 修改配置

① 启动类添加 `@MapperScan` 注解，以指定dao接口的扫描路径。这样我们的DAO接口就不要添加 `@Mapper` 注解了，Spring Boot会自动扫描。

```
1  @MapperScan("cn.sofwin.**.dao")
2  @SpringBootApplication
3  public class SpringBootApplication {
4
5      ...
6  }
```

②修改 `application.yml` 项目配置文件，添加MyBatis配置，已指定xml文件的扫描路径

```
1  # MyBatis 框架配置
2  mybatis:
3      mapper-locations: classpath:mapper/*.xml # SQL映射文件的扫描路径
4      configuration:
5          map-underscore-to-camel-case: true # 自动将下划线分隔的标识符转换为驼峰规则
```

## 3. 创建实体类

```
1  @Data
2  public class Article {
3      private Integer id;
4      private String title;
5      private String content;
6      private Date addTime;
7      private Integer readCount;
8  }
```

## 4. 创建DAO接口

```
1 public interface ArticleDao {  
2  
3     List<Article> selectAll();  
4  
5     Article selectById(Integer id);  
6  
7     void insert(Article article);  
8  
9     void update(Article article);  
10  
11    void deleteById(Integer id);  
12  
13 }
```

## 5. 创建SQL映射文件

首先在类路径（即 `resources` 目录）下创建 `mapper` 文件夹，用于放置所有SQL映射文件。然后我们新建 `ArticleDao.xml` 文件：

```
1 <?xml version="1.0" encoding="UTF-8"?>  
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
3   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
4 <mapper namespace="cn.sofwin.hello.dao.ArticleDao">  
5  
6     <insert id="insert" useGeneratedKeys="true" keyProperty="id">  
7         insert into  
8             article(title, content, add_time, read_count)  
9             values  
10            (#{{title}, #{{content}, #{{addTime}, #{{readCount}}})  
11        </insert>  
12  
13        <select id="selectAll" resultType="cn.sofwin.hello.entity.Article">  
14            select * from article  
15        </select>  
16  
17        <select id="selectById" resultType="cn.sofwin.hello.entity.Article">  
18            select * from article where id = #{{id}}  
19        </select>  
20  
21        <update id="update">  
22            update  
23                article  
24                set  
25                    title = #{{title}},  
26                    content = #{{content}},  
27                    read_count = #{{readCount}}  
28                where  
29                    id = #{{id}}  
30            </update>  
31  
32        <delete id="deleteById">  
33            delete from article where id = #{{id}}
```

```
33      </delete>
34
35  </mapper>
```

## 6. 编写测试类

```
1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class SpringBootApplicationTests {
4
5      @Autowired
6      ArticleDao articleDao;
7
8      @Test
9      public void testMyBatis()  {
10         Article article = new Article();
11         article.setTitle("测试文章");
12         article.setContent("从前有座山、山里有座庙... ");
13         article.setAddTime(new Date());
14         article.setReadCount(1);
15
16         articleDao.insert(article);
17
18         List<Article> articleList = articleDao.selectAll();
19         System.out.println(articleList);
20     }
21 }
```

注意： 映射文件的编写规范请参考MyBatis官方文档。

### 2.5.3 Spring Data Jpa

#### 概念介绍

JPA ( Java Persistence API ) 即Java持久化API，是Sun官方在JDK5.0后提出的Java持久化规范 (JSR 338)，这些接口所在包为 `javax.persistence`，详细内容可参考 <https://github.com/javaee/jpa-spec> )

#### Spring Data

Spring Data 项目的目的是为了简化构建基于 Spring 框架应用的数据访问技术，包括非关系数据库、Map-Reduce 框架、云数据服务等等；另外也包含对关系数据库的访问支持。

##### 1. SpringData特点

SpringData为我们提供使用统一的API来对数据访问层进行操作；这主要是Spring Data Commons项目来实现的。Spring Data Commons让我们在使用关系型或者非关系型数据访问技术时都基于Spring提供的统一标准，标准包含了CRUD（创建、获取、更新、删除）、查询、排序和分页的相关操作。

##### 2. SpringData提供了统一的Repository接口

Repository<T, ID extends Serializable>: 统一接口

RevisionRepository<T, ID extends Serializable, N extends Number & Comparable>: 基于乐观锁机制

CrudRepository<T, ID extends Serializable>: 基本CRUD操作

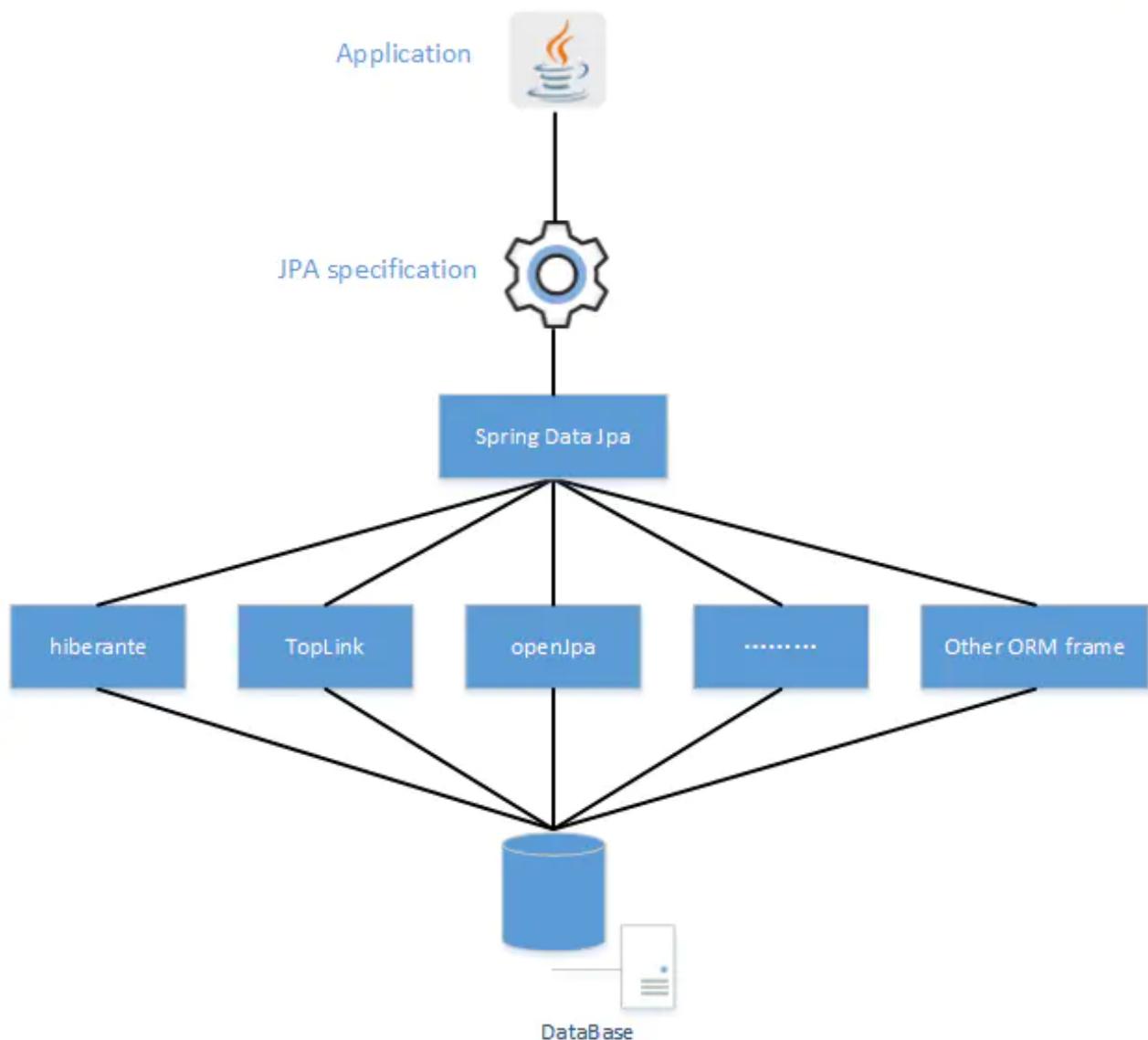
PagingAndSortingRepository<T, ID extends Serializable>: 基本CRUD及分页

### 3. 提供数据访问模板类 xxxTemplate

如: MongoTemplate、 RedisTemplate等

## Spring Data JPA

Spring Data JPA是Spring Data家族的一部分，可以轻松实现基于JPA的存储库。此模块处理对基于JPA的数据访问层的增强支持。它使构建使用数据访问技术的Spring驱动应用程序变得更加容易。



## 整合步骤

### 1. 导入依赖

在 `pom.xml` 配置文件中添加以下依赖:

```

1 <!-- JPA启动器 -->
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-data-jpa</artifactId>
5 </dependency>
6
7 <!-- 数据库驱动 -->
8 <dependency>
9   <groupId>mysql</groupId>
10    <artifactId>mysql-connector-java</artifactId>
11    <scope>runtime</scope>
12 </dependency>
13
14 <!-- Spring测试 -->
15 <dependency>
16   <groupId>org.springframework.boot</groupId>
17     <artifactId>spring-boot-starter-test</artifactId>
18     <scope>test</scope>
19 </dependency>

```

## 2. 修改配置

在主配置文件中添加以下配置：

```

1 spring:
2   datasource:
3     username: root
4     password: 123456
5     url: jdbc:mysql://127.0.0.1:3306/sw_jdbc_demo
6     driver-class-name: com.mysql.cj.jdbc.Driver
7     # hikari数据库连接池
8     hikari:
9       minimum-idle: 2 #最小空闲连接数量
10      idle-timeout: 180000 #空闲连接存活最大时间, 默认600000 (10分钟)
11      maximum-pool-size: 5 #连接池最大连接数, 默认是10
12      auto-commit: true #此属性控制从池返回的连接的默认自动提交行为, 默认值: true
13      max-lifetime: 1800000 #此属性控制池中连接的最长生命周期, 值0表示无限生命周期,
14        默认1800000即30分钟
15      connection-timeout: 30000 #数据库连接超时时间, 默认30秒, 即30000
16      connection-test-query: SELECT 1
17      # JPA配置
18      jpa:
19        show-sql: true
20        hibernate:
21          ddl-auto: none
22      # 配置打印MyBatis执行的SQL
23      logging.level.cn.sofwin.hello.dao: debug

```

## 3. 添加实体类

```
1  @Entity
2  @Table(name = "user")
3  public class User {
4      @Id // 标记为主键
5      @GeneratedValue(strategy = GenerationType.IDENTITY) // 使用自增长策略
6      private Integer id;
7
8      @Column(name = "username") // 标记为字段，并设置字段名
9      private String username;
10
11     @Column(name = "password")
12     private String password;
13
14     @Column // 如果属性名和字段名一致，则可省略字段名
15     private String nickname;
16
17     public User() {
18     }
19
20     public Integer getId() {
21         return id;
22     }
23
24     public void setId(Integer id) {
25         this.id = id;
26     }
27
28     public String getUsername() {
29         return username;
30     }
31
32     public void setUsername(String username) {
33         this.username = username;
34     }
35
36     public String getPassword() {
37         return password;
38     }
39
40     public void setPassword(String password) {
41         this.password = password;
42     }
43
44     public String getNickname() {
45         return nickname;
46     }
47
48     public void setNickname(String nickname) {
49         this.nickname = nickname;
50     }
51 }
```

JPA自带的几种主键生成策略：

- TABLE： 使用一个特定的数据库表格来保存主键
- SEQUENCE： 根据底层数据库的序列来生成主键， 条件是数据库支持序列。这个值要与 generator一起使用， generator指定生成主键使用的生成器（可能是oracle中自己编写的序列）
- IDENTITY： 主键由数据库自动生成（主要是支持自动增长的数据库， 如mysql）
- AUTO： 主键由程序控制， 也是 GenerationType的默认值

4. 添加DAO， 只需创建一个继承了 `JpaRepository` 接口的子接口：

```
1  public interface UserDao extends JpaRepository<User, Integer> {  
2  }
```

`JpaRepository` 接口中的两个泛型， 分别代表： 实体类的类型、 主键的类型

5. 测试用例

```
1  @RunWith(SpringRunner.class)  
2  @SpringBootTest  
3  public class JpaTest {  
4  
5      @Autowired  
6      UserDao userDao;  
7  
8      @Test  
9      @Transactional  
10     public void testQueryById(){  
11         User user = userDao.getOne(1);  
12         System.out.println(user.getUsername());  
13     }  
14  
15     @Test  
16     @Transactional  
17     public void testQueryByUsername(){  
18         User user = userDao.findByUsername("admin");  
19         System.out.println(user.getId() + ":" + user.getUsername());  
20     }  
21  
22 }
```

Spring Data Jpa 最大的优势是可以自动生成SQL。语法规则可以参考：<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.repositories>

## 2.5.4 Redis

Redis是一款高效的Key-Value数据库。安装教程：[Redis安装与配置](#)

在Java Web开发中，Redis是重要的缓存中间件。在Spring Boot中操作Redis也非常简单，主要有以下步骤：

1. 引入Maven依赖，添加 `spring-boot-starter-data-redis` 启动器

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-data-redis</artifactId>
4 </dependency>
```

2. 修改配置文件，添加Redis服务器配置。

注意：SpringBoot 1.x版本中默认的redis连接池采用的是jedis；而在2.x版本中，已经切换为lettuce。

```
1 spring:
2   # Redis 配置
3   redis:
4     host: 127.0.0.1
5     port: 6379
6     password: xw2016
7   #   lettuce:
8   #     pool:
9   #       min-idle: 0
10  #       max-idle: 8
11  #       max-wait: 1ms
12  #       max-active: 8
13  #       shutdown-timeout: 100ms
```

3. 编写测试程序

```
1 @RunWith(SpringRunner.class)
2 @SpringBootTest
3 public class SpringBootApplicationTests {
4
5   @Autowired
6   RedisTemplate redisTemplate; // 操作k-v对象
7
8   @Autowired
9   StringRedisTemplate stringRedisTemplate; // 操作k-v字符串
10
11  @Test
12  public void testRedisTemplate(){
13    stringRedisTemplate.opsForValue().set("name", "软赢科技");
14    String name = stringRedisTemplate.opsForValue().get("name");
15    System.out.println(name);
16  }
17}
```

**RedisTemplate** 默认会使用jdk序列化机制对key、value进行序列化，然后保存到redis中。如果我们希望以json方式进行序列化，我们需要做一些修改。

添加Redis配置：

```
1  @Configuration
2  public class RedisConfig {
3      @Bean
4      public static RedisTemplate<String, Object>
5          redisTemplate(LettuceConnectionFactory lettuceConnectionFactory){
6              // 设置序列化
7              Jackson2JsonRedisSerializer<Object> jackson2JsonRedisSerializer = new
8                  Jackson2JsonRedisSerializer<Object>(
9                      Object.class);
10             ObjectMapper om = new ObjectMapper();
11             om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
12             om.activateDefaultTyping(LaissezFaireSubTypeValidator.instance,
13                 ObjectMapper.DefaultTyping.NON_FINAL);
14             jackson2JsonRedisSerializer.setObjectMapper(om);
15
16             // 配置redisTemplate
17             RedisTemplate<String, Object> redisTemplate = new RedisTemplate<String,
18                 Object>();
19             redisTemplate.setConnectionFactory(lettuceConnectionFactory);
20             RedisSerializer<?> stringSerializer = new StringRedisSerializer();
21             redisTemplate.setKeySerializer(stringSerializer); // key序列化
22             redisTemplate.setValueSerializer(jackson2JsonRedisSerializer); // value序列
23             化
24             redisTemplate.setHashKeySerializer(stringSerializer); // Hash key序列化
25             redisTemplate.setHashValueSerializer(jackson2JsonRedisSerializer); // Hash
26             value序列化
27             redisTemplate.afterPropertiesSet();
28             return redisTemplate;
29         }
30     }
```

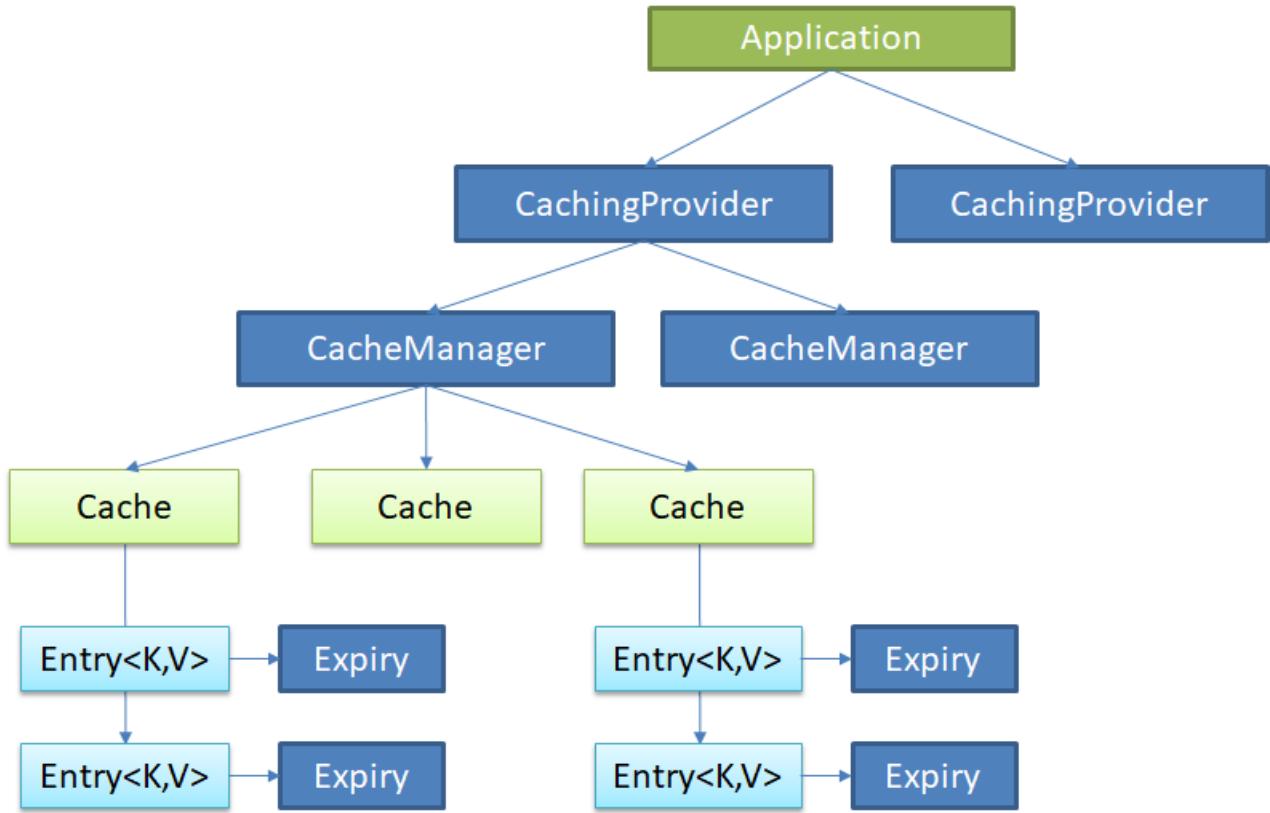
## 2.6 缓存

注意： 这里使用的是Spring本地缓存，适合单体应用，不适合分布式应用。分布式应用，应该使用集中式缓存，如：Redis。

Java Caching定义了5个核心接口，分别是：

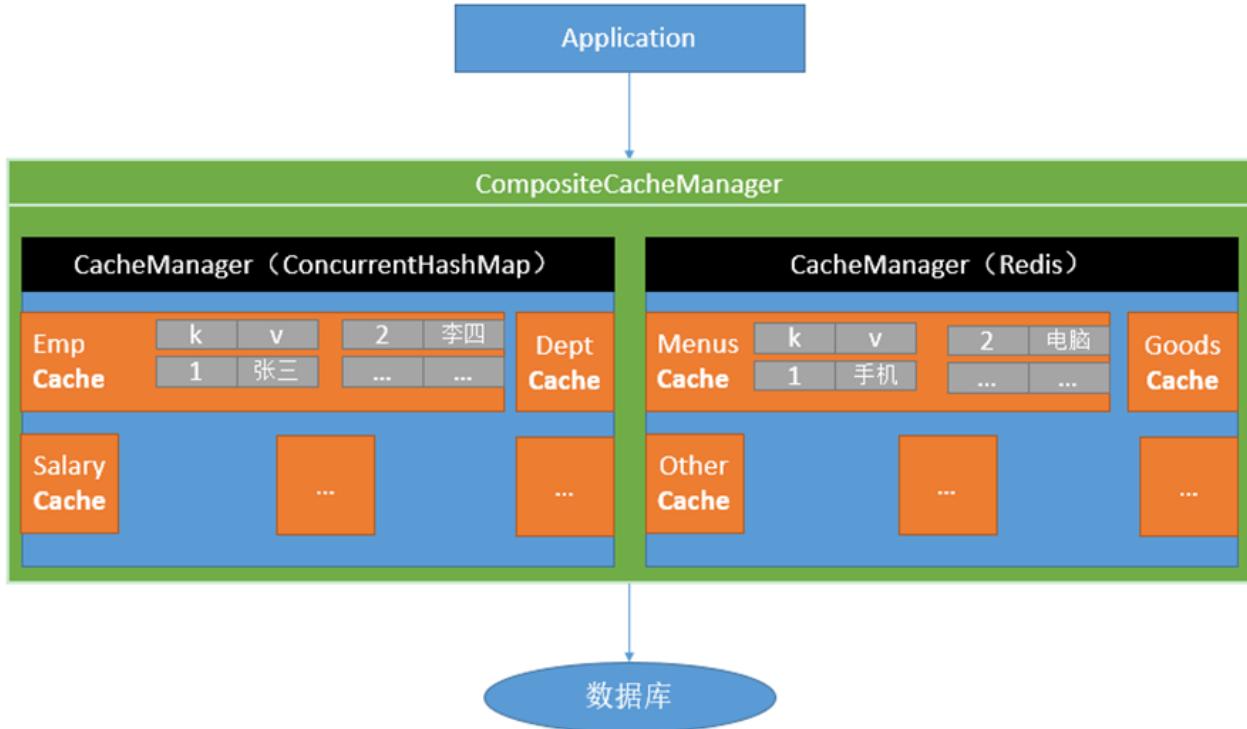
- **CachingProvider** 定义了创建、配置、获取、管理和控制多个**CacheManager**。一个应用可以在运行期访问多个CachingProvider。
- **CacheManager** 定义了创建、配置、获取、管理和控制多个唯一命名的**Cache**，这些Cache存在于**CacheManager**的上下文中。一个**CacheManager**仅被一个CachingProvider所拥有。

- **Cache** 是一个类似Map的数据结构并临时存储以Key为索引的值。一个Cache仅被一个CacheManager 所拥有。
- **Entry** 是一个存储在Cache中的key-value对。
- **Expiry** 每一个存储在Cache中的条目有一个定义的有效期。一旦超过这个时间，条目为过期的状态。一旦过期，条目将不可访问、更新和删除。缓存有效期可以通过ExpiryPolicy设置。



Spring从3.1开始定义了org.springframework.cache.Cache和org.springframework.cache.CacheManager接口来统一不同的缓存技术；并支持使用JCache（JSR-107）注解简化我们开发。

- Cache接口为缓存的组件规范定义，包含缓存的各种操作集合；
- Cache接口下Spring提供了各种xxxCache的实现；如RedisCache, EhCacheCache , ConcurrentMapCache等；
- 每次调用需要缓存功能的方法时，Spring会检查指定参数的指定的目标方法是否已经被调用过；如果有就直接从缓存中获取方法调用后的结果，如果没有就调用方法并缓存结果后返回给用户。下次调用直接从缓存中获取。
- 使用Spring缓存抽象时我们需要关注以下两点：
  - 确定方法需要被缓存以及他们的缓存策略
  - 从缓存中读取之前缓存存储的数据



### 概念及注解:

<b>Cache</b>	缓存接口，定义缓存操作。实现有：RedisCache、EhCacheCache、ConcurrentMapCache等
<b>CacheManager</b>	缓存管理器，管理各种缓存（Cache）组件
<b>@Cacheable</b>	主要针对方法配置，能够根据方法的请求参数对其结果进行缓存
<b>@CacheEvict</b>	清空缓存
<b>@CachePut</b>	保证方法被调用，又希望结果被缓存。
<b>@EnableCaching</b>	开启基于注解的缓存
<b>keyGenerator</b>	缓存数据时key生成策略
<b>serialize</b>	缓存数据时value序列化策略

<code>@Cacheable/@CachePut/@CacheEvict</code> 主要的参数		
value	缓存的名称，在 spring 配置文件中定义，必须指定至少一个	例如： <code>@Cacheable(value="mycache") 或者 @Cacheable(value={"cache1","cache2"})</code>
key	缓存的 key，可以为空，如果指定要按照 SpEL 表达式编写，如果不指定，则缺省按照方法的所有参数进行组合	例如： <code>@Cacheable(value="testcache", key="#userName")</code>
condition	缓存的条件，可以为空，使用 SpEL 编写，返回 true 或者 false，只有为 true 才进行缓存/清除缓存，在调用方法之前之后都能判断	例如： <code>@Cacheable(value="testcache", condition="#userName.length()&gt;2")</code>
allEntries (@CacheEvict )	是否清空所有缓存内容，缺省为 false，如果指定为 true，则方法调用后将立即清空所有缓存	例如： <code>@CacheEvict(value="testcache", allEntries=true)</code>
beforeInvocation (@CacheEvict)	是否在方法执行前就清空，缺省为 false，如果指定为 true，则在方法还没有执行的时候就清空缓存，缺省情况下，如果方法执行抛出异常，则不会清空缓存	例如： <code>@CacheEvict(value="testcache", beforeInvocation=true)</code>
unless (@CachePut) (@Cacheable)	用于否决缓存的，不像 condition，该表达式只在方法执行之后判断，此时可以拿到返回值 result 进行判断。条件为 true 不会缓存， false 才缓存	例如： <code>@Cacheable(value="testcache", unless="#result == null")</code>

## 2.6.1 开启缓存

在启动类上添加 `@EnableCaching` 注解：

```

1  @SpringBootApplication
2  @EnableCaching
3  @MapperScan("cn.sofwin.**.dao")
4  public class Application {
5
6      public static void main(String[] args) {
7          SpringApplication.run(Application.class, args);
8      }
9
10 }
```

## 2.6.2 在业务方法上添加注解

### 1. 启用缓存: `@Cacheable`

`@Cacheable` 会先检查缓存，如果缓存中有数据，则直接返回，不再调用方法；否则先调用方法，然后将结果放置到缓存中。

```
1  /**
2   * CacheManager可以管理多个缓存组件，每个缓存组件都有自己唯一的名字（相当于数据库中的表）
3   * Cacheable有几个属性：
4   *      value/cacheName: 指定缓存的名字；
5   *      key: 缓存数据时使用的key（相当于主键字段名），如果不指定则使用方法参数，可以使用SpEL
6   *      @return
7   */
8 @Cacheable(cacheNames = "user", key = "#id")
9 // @Cacheable(cacheNames = "user", key = "#root.methodName + '[' + #id +
10 public User get(Integer id) {
11     return userDao.selectOne(id);
12 }
```

### 2. 更新缓存: `@CachePut`

`@CachePut` 会先执行方法，将返回值放置到缓存。他不会事先检查缓存。一般用于更新操作。

```
1  @CachePut(cacheNames = "user", key = "#user.id")
2  @Override
3  public User update(User user) {
4      userDao.update(user);
5      return user;
6 }
```

### 3. 清除缓存: `@CacheEvict`

`@CacheEvict` 用于清除缓存，一般用户删除操作

```
1  @CacheEvict(cacheNames = "user", key = "#id")
2  @Override
3  public void delete(Integer id) {
4      userDao.deleteById(id);
5 }
```

## 2.7 SpringBoot页面视图

教学目标:

- 掌握Spring Boot中web项目的创建；
- 掌握静态资源映射规则；
- 掌握Thymeleaf模板引擎的使用。

Spring Boot不仅可以编写Controller控制器接口，也可以开发Web页面，官方推荐使用模板引擎（如：thymleaf、Freemarker等）。

## 2.7.1 静态资源的映射规则

在 `WebMvcAutoConfiguration` 类中定义了资源处理器的逻辑：

```
1  @Override
2  public void addResourceHandlers(ResourceHandlerRegistry registry) {
3      if (!this.resourceProperties.isAddMappings()) {
4          logger.debug("Default resource handling disabled");
5          return;
6      }
7      Duration cachePeriod = this.resourceProperties.getCache().getPeriod();
8      CacheControl cacheControl =
9          this.resourceProperties.getCache().getCachecontrol().toHttpCacheControl();
10     if (!registry.hasMappingForPattern("/webjars/**")) {
11
12         customizeResourceHandlerRegistration(registry.addResourceHandler("/webjars/**")
13             .addResourceLocations("classpath:/META-INF/resources/webjars/")
14             .setCachePeriod(getSeconds(cachePeriod)).setCacheControl(cacheControl));
15     }
16     String staticPathPattern = this.mvcProperties.getStaticPathPattern();
17     if (!registry.hasMappingForPattern(staticPathPattern)) {
18
19         customizeResourceHandlerRegistration(registry.addResourceHandler(staticPathPatte
rn)
20             .addResourceLocations(getResourceLocations(this.resourceProperties.getStaticLocat
ions())))
21             .setCachePeriod(getSeconds(cachePeriod)).setCacheControl(cacheControl));
22     }
23 }
```

其中定义的规则如下：

1. `ResourceProperties` 类中定义了静态资源的目录及相关配置项

```
1  @ConfigurationProperties(prefix = "spring.resources", ignoreUnknownFields =  
2    false)  
3  public class ResourceProperties {  
4  
4      private static final String[] CLASSPATH_RESOURCE_LOCATIONS = {  
5          "classpath:/META-INF/resources/",  
6              "classpath:/resources/", "classpath:/static/",  
7          "classpath:/public/" };  
8  
8      ...  
9  }
```

即包含以下路径：

```
1  "classpath:/META-INF/resources/"  
2  "classpath:/resources/"  
3  "classpath:/static/"  
4  "classpath:/public/"  
5  "/" 当前系统的根路径
```

也可以在配置文件中，指定静态资源目录，如：

```
1  spring:  
2      resources:  
3          static-locations: classpath:/site/,classpath:/hello/
```

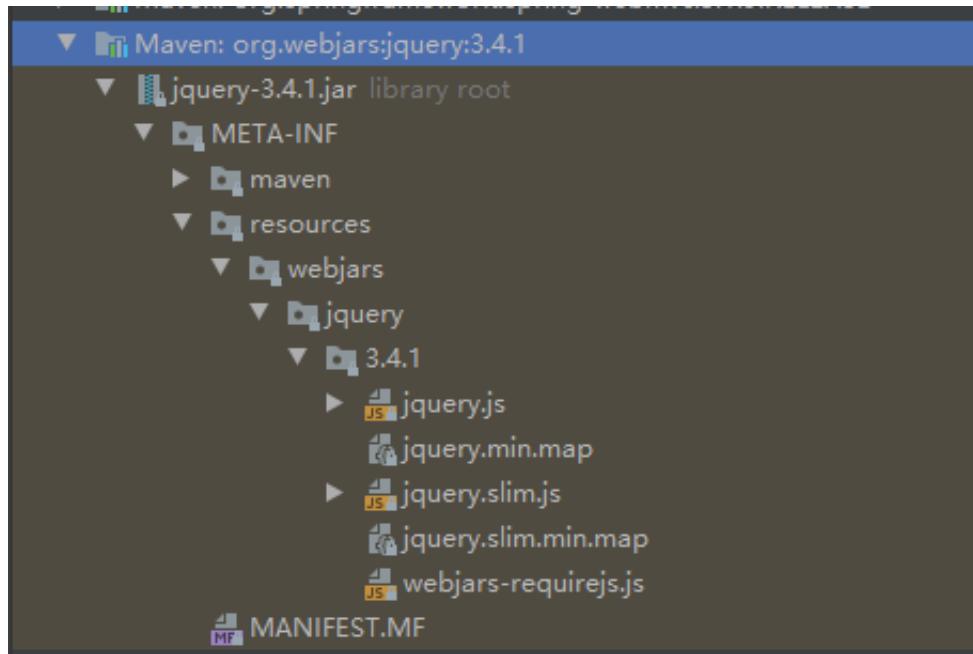
注意：使用过程中，访问地址不需要添加 `/resource`、`/static`、`/public` 等前缀。

- 所有 `/webjars/**` 请求，都从 `classpath:/META-INF/resources/webjars/` 目录中寻找相关资源。

所谓 `webjars` 即通过 `maven` 以jar包的方式引入静态资源，参考：<https://www.webjars.org/>。

以导入jquery为例，首先添加 `maven` 依赖：

```
1  <dependency>  
2      <groupId>org.webjars</groupId>  
3      <artifactId>jquery</artifactId>  
4      <version>3.5.1</version>  
5  </dependency>
```



我们就可以通过路径：<http://localhost:8080/webjars/jquery/3.4.1/jquery.js> 来引用 jquery文件。

## 2.7.2 Thymeleaf模板引擎

市面上常见的模板引擎有：JSP、Velocity、Freemarker、Thymeleaf等，Spring Boot官方推荐使用模板 **Thymeleaf** 引擎。

模板引擎的思想是：

- 开发时：将模板和数据分离；
- 运行时：模板引擎会将模板和数据整合，生成最终文档。

**千万注意：** 模板 + 数据才能生成文档，因此我们不要直接访问模板，而是通过访问Controller，让Spring Boot自动调用视图解析器（模板引擎）来渲染、生成页面文档。

官方网站：<https://www.thymeleaf.org/>

### 开发流程

1. 添加场景启动器：

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-thymeleaf</artifactId>
4 </dependency>
```

Spring Boot 已经对Thymeleaf添加了自动配置规则：

```
1 @ConfigurationProperties(prefix = "spring.thymeleaf")
2 public class ThymeleafProperties {
3
4     private static final Charset DEFAULT_ENCODING = StandardCharsets.UTF_8;
5
6     public static final String DEFAULT_PREFIX = "classpath:/templates/";
7
8     public static final String DEFAULT_SUFFIX = ".html";
9
10    ...
11 }
```

可以看到，我们需要把模板放置到 `classpath:/templates/` 目录下，默认的模板文件扩展名是 `.html`。

## 2. 添加控制器：

```
1 @Controller
2 public class HelloController {
3
4     @RequestMapping("/hello")
5     public String hello(){
6         return "hello";
7     }
8 }
```

由于我们已经添加了Thymeleaf的maven依赖，Spring Boot将会使用Thymeleaf的视图解析器来渲染页面。

## 3. 添加模板页面：

在 `classpath:/templates` 目录下添加 `hello.html` 文档。

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta charset="UTF-8">
5     <title>Hello</title>
6 </head>
7 <body>
8     <h1>Hello</h1>
9 </body>
10 </html>
```

启动项目，即可通过 `/hello` 地址来访问页面。

## Thymeleaf 语法

### 示例代码：

- 1.) 添加新的请求映射，在请求作用域中添加属性值

```
1  @Controller
2  public class HelloController {
3      /**
4      * 携带数据的请求
5      * @return
6      */
7      @RequestMapping("/info")
8      public String info(Map<String, Object> map){
9          map.put("msg", "你好");
10         return "info";
11     }
12
13 }
```

## 2.) 添加模板页面

```
1  <!DOCTYPE html>
2  <html xmlns:th="http://www.thymeleaf.org">
3  <head>
4      <meta charset="UTF-8">
5      <title>Hello</title>
6  </head>
7  <body>
8      <h1>info</h1>
9      <p th:text="${msg}"></p>
10 </body>
11 </html>
```

注意：在编写模板页面过程中，为了让IDEA能智能提示代码以提升开发效率，我们可以在 `<html>` 标签中添加 `xmlns:th="http://www.thymeleaf.org"` 名称空间。当然，这不是必须的。

### 基础语法：

1. Thymeleaf模板文档中，在html标签元素的任意属性前面添加 `th:` 前缀，即表示交由Thymeleaf引擎来处理这些属性。如：`th:id`、`th:class`、`th:src`、`th:href` 等；
2. `th:text`、`th:utext` 都可以用于设置标签内容，区别在于 `th:utext` 会将内容中的文本当前html文档进行渲染；
3. `th:if`、`th:unless` 进行条件判断；
4. `th:switch`、`th:case` 进行多条件判断，`th:case='*'` 代表默认；
5. `th:each` 进行迭代输出；
6. 在 `JavaScript` 中获取值：

```
1  <script th:inline="javascript">
2      var username = [${age}];
3  </script>
```

## 表达式语法:

```
1  Simple expressions:  
2      # 表达式语法  
3  Variable Expressions: ${...}  
4      # 变量取值, OGNL表达式  
5          #1. 获取对象属性、调用方法  
6          #2. 使用内置对象  
7              #ctx : the context object.  
8              #vars: the context variables.  
9              #locale : the context locale.  
10             #request : (only in Web Contexts) the HttpServletRequest object.  
11             #response : (only in Web Contexts) the HttpServletResponse object.  
12             #session : (only in Web Contexts) the HttpSession object.  
13             #servletContext : (only in Web Contexts) the ServletContext object.  
14          #3. 使用内置工具对象  
15              #execInfo : information about the template being processed.  
16              #messages : methods for obtaining externalized messages inside  
variables expressions, in the same way as they would be obtained using #{...}  
syntax.  
17              #uris : methods for escaping parts of URLs/URIs  
18              #conversions : methods for executing the configured conversion service  
(if any).  
19              #dates : methods for java.util.Date objects: formatting, component  
extraction, etc.  
20              #calendars : analogous to #dates , but for java.util.Calendar  
objects.  
21              #numbers : methods for formatting numeric objects.  
22              #strings : methods for String objects: contains, startsWith,  
prepend/append, etc.  
23              #objects : methods for objects in general.  
24              #bools : methods for boolean evaluation.  
25              #arrays : methods for arrays.  
26              #lists : methods for lists.  
27              #sets : methods for sets.  
28              #maps : methods for maps.  
29              #aggregates : methods for creating aggregates on arrays or  
collections.  
30              #ids : methods for dealing with id attributes that might be repeated  
(for example, as a result of an iteration).  
31  
32      Selection Variable Expressions: *{...}  
33      # 变量选择表达式, 功能与${...}相同, 区别在于, 他需要在父标签中存在th:object声明的变量, *  
就代表引用这个对象  
34  
35      Message Expressions: #{...}  
36      # 用于获取国际化内容  
37  
38      Link URL Expressions: @{...}  
39      # 用于动态定义URL链接  
40  
41      Fragment Expressions: ~{...}
```

```

42      # 片段引用表达式
43
44  Literals:
45      # 字面量
46      Text literals: 'one text' , 'Another one!' , ...
47      Number literals: 0 , 34 , 3.0 , 12.3 , ...
48      Boolean literals: true , false
49      Null literal: null
50      Literal tokens: one , sometext , main , ...
51
52  Text operations:
53      # 文本操作
54      String concatenation: +
55      Literal substitutions: |The name is ${name}|
56
57  Arithmetic operations:
58      # 数学运算
59      Binary operators: + , - , * , / , %
60      Minus sign (unary operator): -
61
62  Boolean operations:
63      # 布尔运算
64      Binary operators: and , or
65      Boolean negation (unary operator): ! , not
66
67  Comparisons and equality:
68      # 比较运算
69      Comparators: > , < , >= , <= ( gt , lt , ge , le )
70      Equality operators: == , != ( eq , ne )
71
72  Conditional operators:
73      # 条件运算
74      If-then: (if) ? (then)
75      If-then-else: (if) ? (then) : (else)
76      Default: (value) ?: (defaultvalue)
77
78  Special tokens:
79      # 特殊操作
80      No-Operation: _

```

## 2.8 . 配置SpringMVC

教学目标:

1. 掌握SpringMVC自动配置原理;
2. 掌握RESTful风格的API接口开发;
3. 掌握拦截器与异常处理;
4. 了解SpringBoot中Servlet、Filter的创建。

## 2.8.1 SpringMVC自动配置

文档地址：<https://docs.spring.io/spring-boot/docs/2.3.6.RELEASE/reference/html/spring-boot-features.html#boot-features-spring-mvc>

Spring Boot 自动配置好了SpringMVC

以下是Spring Boot对SpringMVC的默认配置:

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
  - 配置了视图解析器，`ContentNegotiatingViewResolver` 会自动获取容器中所有视图解析器。我们可以向容器中添加自己的视图解析器。
- Support for serving static resources, including support for WebJars (covered [later in this document](#)).
  - 静态资源映射
- Automatic registration of `Converter`, `GenericConverter`, and `Formatter` beans.
  - 参数类型转换器、格式化器
- Support for `HttpMessageConverters`
  - 消息转换器，一般用于json格式转换
- Automatic registration of `MessageCodesResolver` .
  - 用于错误代码生成规则
- Static `index.html` support.
  - 首页映射
- Custom `Favicon` support
  - 网站图片
- Automatic use of a `ConfigurableWebBindingInitializer` bean.
  - 初始化WebDataBinder

Spring Boot 配置模式:

1. 自动配置：Spring Boot在自动配置时，会优先扫描、使用用户自定义的组件，如果用户没有配置，才使用默认的组件。
2. 扩展配置：当我们想在自动配置的基础上，添加一点自己的功能（如拦截器），则可以编写一个`@Configuration`，这个类要实现`WebMvcConfigurer`接口，且不能包含`@EnableWebMvc`注解。**原理：**`WebMvcAutoConfigurationAdapter`会自动扫描、装配`WebMvcConfigurer`的实例。
3. Spring Boot中会有很多`XXXConfigurer`，这些都可以帮助我们进行扩展配置。

## 2.8.2 配置拦截器

### 1. 自定义拦截器

```
1  @Slf4j
2  @Component
3  public class SimpleInterceptor implements HandlerInterceptor {
```

```
4
5     @Override
6     public boolean preHandle(HttpServletRequest request, HttpServletResponse
7         response, Object handler) throws Exception {
8
9         log.info("=====拦截请求=====");
10        log.info("path: {}", request.getServletPath());
11        log.info("=====");
12        return true; // true代表继续后续流程, false则终止向后执行
13    }
14
15    @Override
16    public void postHandle(HttpServletRequest request, HttpServletResponse
17        response, Object handler, ModelAndView modelAndView) throws Exception {
18
19    }
20
21    @Override
22    public void afterCompletion(HttpServletRequest request,
23        HttpServletResponse response, Object handler, Exception ex) throws Exception
24    {
25
26    }
27}
```

## 2. 添加拦截器配置

```
1  @Configuration
2  public class MvcConfig implements WebMvcConfigurer {
3
4      @Autowired
5      SimpleInterceptor simpleInterceptor;
6
7      /**
8       * 配置拦截器
9       * @param registry
10      */
11     @Override
12     public void addInterceptors(InterceptorRegistry registry) {
13         registry.addInterceptor(simpleInterceptor)
14             .addPathPatterns("/interceptor/**")
15             .excludePathPatterns("/interceptor/exclude");
16     }
17 }
```

## 2.8.3 错误处理

### Spring Boot默认错误处理

默认情况下，Spring Boot会根据客户端的类型，返回不同的错误信息。对于浏览器会返回错误页面，对于其他客户端则会返回错误的JSON信息。

原理：

由 `ErrorMvcAutoConfiguration` 完成自动配置，它向容器中添加了以下组件：

1. `DefaultErrorAttributes`

封装错误信息，如：`timestamp`、`status`、`error`、`exception`、`message` 等

2. `BasicErrorController`

处理 `/error` 请求，会根据Http请求头中的 `Accept` 信息来判断是返回 `html` 还是 `json`

```
1  @Controller
2  @RequestMapping("${server.error.path:${error.path:/error}}")
3  public class BasicErrorController extends AbstractErrorController {
4      ...
5  }
```

对于返回 `html` 的请求，将会调用 `DefaultErrorViewResolver` 来解析视图；

3. `ErrorResponseCustomizer`

系统出现错误后，转到跳转到 `/error` 请求进行处理。

4. `DefaultErrorViewResolver`

错误页面解析

步骤：

一旦系统出现4XX、5XX之类的错误，`ErrorResponseCustomizer`就会生效，以定制错误响应规则。

定制错误页面：

1. 使用模板引擎时，在 `classpath:/templates/error/` 文件夹下，按 `状态码.html` 的形式添加错误页面。如：`classpath:/templates/error/404.html`，  
`classpath:/templates/error/502.html`，或使用通配符  
`classpath:/templates/error/4xx.html`、`classpath:/templates/error/5xx.html`；
2. 没有模板引擎时，可以将 `error` 文件夹放置在静态资源目录下。

定制JSON信息：

添加异常处理器

```
1  /**
2  * 异常处理器
3  */
```

```
4  @ControllerAdvice
5  public class MyExceptionHandler {
6
7      @ResponseBody
8      @ExceptionHandler(Exception.class) // 建议写特定异常的子类
9      public Map<String, Object> handleException(Exception e){
10         Map<String, Object> map = new HashMap<>();
11         map.put("code", 100);
12         map.put("msg", e.getMessage());
13         return map;
14     }
15 }
16 }
```

## 2.9 AOP编程

步骤：

1. 导入AOP模块依赖
2. 定义切面：通知方法
3. 开启基于注解的AOP

### 2.9.1 导入Maven 依赖

```
1 <dependency>
2     <groupId>org.springframework</groupId>
3     <artifactId>spring-aop</artifactId>
4 </dependency>
5
6 <dependency>
7     <groupId>org.aspectj</groupId>
8     <artifactId>aspectjweaver</artifactId>
9 </dependency>
```

### 2.9.2 定义切面

```
1 /**
2  * Controller 日志切面
3  */
4 @Aspect // 切面注解
5 @Component // 需要导入到IOC容器
6 public class LogAspect {
```

```
8     private static Logger log = LoggerFactory.getLogger(LogAspect.class);
9
10    /**
11     * 定义切入点
12     */
13    @Pointcut("execution(* net.letcode..controller.*.*(..))")
14    public void pointCut(){}
15
16    /**
17     * 前置通知
18     * 在目标方法之前执行
19     */
20    @Before("pointCut()")
21    public void doBefore() {
22        log.info("LogAspect before...");
23    }
24
25    /**
26     * 后置通知
27     * 在目标方法之后执行，不管是否有异常抛出
28     */
29    @After("pointCut()")
30    public void doEnd() {
31        log.info("LogAspect end...");
32    }
33
34    /**
35     * 返回通知
36     * 在目标方法正常执行之后执行，有异常则不执行
37     */
38    @AfterReturning("pointCut()")
39    public void doReturn() {
40        log.info("LogAspect return...");
41    }
42
43    /**
44     * 异常通知
45     * 在目标方法抛出异常之后执行，没有异常则不执行
46     */
47    @AfterThrowing("pointCut()")
48    public void doException() {
49        log.info("LogAspect exception...");
50    }
51
52    /**
53     * 环绕通知
54     * 最底层的方法，需要手动推进目标方法执行
55     */
56    // @Around("pointCut()")
57    public void doRound(){
58        log.info("LogAspect before...");
59    }
```

```
60  
61     }
```

## 切入点表达式

在上述代码中，切入点表达式的定义是：`execution(* net.letcode..controller.*.*(..))`，其中各部分含义如下：

标识符	含义
<code>execution ()</code>	表达式的主体
第一个 <code>*</code> 符号	表示返回值的类型， <code>*</code> 代表所有返回类型
<code>net.letcode..controller</code>	AOP 所切的服务的包名，即需要进行横切的业务类
包名后面的 <code>..</code>	表示当前包及子包
第二个 <code>*</code>	表示类名， <code>*</code> 表示所有类
最后的 <code>.*(..)</code>	第一个 <code>.</code> 表示任何方法名，括号内为参数类型， <code>..</code> 代表任何类型参数

## 2.9.3 开启基于注解的AOP

在启动类上添加注解即可：

```
1  @SpringBootApplication  
2  @EnableAspectJAutoProxy // 启用AspectJ自动代理，启用基于注解的AOP模式  
3  @EnableTransactionManagement  
4  @MapperScan("net.letcode.**.dao")  
5  public class Application {  
6      public static void main(String[] args) {  
7          SpringApplication.run(Application.class, args);  
8      }  
9  }
```

## 2.10 任务管理

## 2.10.1 异步任务

在Java应用中，绝大多数情况下都是通过同步的方式来实现交互处理的；但是在处理与第三方系统交互的时候，容易造成响应迟缓的情况。在Spring 3.x之后，已经内置了@Async来完美解决这个问题。

创建服务：

```
1  @Service
2  public class AsyncService {
3
4      public void asyncTask(){
5          System.out.println("任务开始...");
6          try {
7              Thread.sleep(10000);
8          } catch (InterruptedException e) {
9              e.printStackTrace();
10         }
11         System.out.println("任务结束...");
12     }
13
14 }
```

创建测试类：

```
1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class AsyncServiceTest {
4
5      @Autowired
6      AsyncService asyncService;
7
8      @Test
9      public void testTask(){
10
11         System.out.println("测试调用开始...");
12
13         asyncService.asyncTask();
14
15         System.out.println("测试调用结束...");
16
17         // 防止主线程退出
18         try {
19             new CountDownLatch(1).await();
20         } catch (InterruptedException e) {
21             e.printStackTrace();
22         }
23     }
24
25 }
```

这是传统的同步调用方式，我们卡到的结果是：

```
1 测试调用开始...
2 任务开始...
3 任务结束...
4 测试调用结束...
```

我们将服务类中的方法进行改造，添加注解 `@Async`：

```
1 @Service
2 public class AsyncService {
3
4     @Async
5     public void asyncTask(){
6         System.out.println("任务开始...");
7         try {
8             Thread.sleep(10000);
9         } catch (InterruptedException e) {
10             e.printStackTrace();
11         }
12         System.out.println("任务结束...");
13     }
14
15 }
```

同时，为启动类添加注解 `@EnableAsync`：

```
1 @EnableAsync
2 @SpringBootApplication
3 public class Application {
4
5     public static void main(String[] args) {
6         SpringApplication.run(Application.class);
7     }
8
9 }
```

再次运行测试用例，我们看到的结果是：

```
1 测试调用开始...
2 测试调用结束...
3 任务开始...
4 任务结束...
```

此时我们可以看到，调用是异步的。

## 2.10.2 定时任务

我们经常需要执行一些定时任务，比如需要在每天凌晨时候，分析一次前一天的日志信息。Spring为我们提供了异步执行任务调度的方式，提供TaskExecutor、TaskScheduler接口。

创建服务：

```
1  @Service
2  public class TimerService {
3
4      /**
5       * conn 表达式规则：
6       *      second, minute, hour, day of month, month, and day of week
7       *      如：
8       *      0 * * * * MON-FRI
9       *      代表：
10      *      周一 (MON) 到周六 (SAT) 的整秒运行 (即每分钟运行一次)
11      */
12      @Scheduled(cron = "0 * * * * MON-SAT")
13      public void timerTask(){
14          System.out.println("执行定时任务...");
```

启动类添加注解 `@EnableScheduling`：

```
1  @EnableScheduling
2  @SpringBootApplication
3  public class Application {
4
5      public static void main(String[] args) {
6          SpringApplication.run(Application.class);
7      }
8
9  }
```

启动项目，我们可以在控制台看到每分钟的0秒都会执行一次：

```
1  执行定时任务...
2  执行定时任务...
```

Cron表达式：

字段	允许值	允许的特殊字符
秒	0-59	, - * /
分	0-59	, - * /
小时	0-23	, - * /
日期	1-31	, - * ? / L W C
月份	1-12	, - * /
星期	0-7或SUN-SAT 0,7是SUN	, - * ? / L C #

特殊字符	代表含义
,	枚举
-	区间
*	任意
/	步长
?	日/星期冲突匹配
L	最后
W	工作日
C	和calendar联系后计算过的值
#	星期, 4#2, 第2个星期四

## 2.11 健康监控

我们经常需要了解当前应用的运行情况，通过引入spring-boot-starter-actuator，可以使用Spring Boot为我们提供的准生产环境下的应用监控和管理功能。我们可以通过HTTP, JMX, SSH协议来进行操作，自动得到审计、健康及指标信息等。

## 2.11.1 基本使用

### 1. 添加Maven依赖:

```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-web</artifactId>
5   </dependency>
6
7   <dependency>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-actuator</artifactId>
10    </dependency>
11  </dependencies>
```

### 2. 添加配置文件:

Spring Boot 2.X 中，Actuator 默认只开放 health 和 info 两个端点，添加以下配置可以开放完整的监控端点。

```
1 management:
2   server:
3     # 修改端口
4     port: 9090
5   endpoints:
6     web:
7       # 禁用所有
8       enabled-by-default: false
9       # 放开所有端点
10      exposure:
11        include: "*"
12      # 修改根路径
13      base-path: /manager
14      # 修改端点的访问路径
15      path-mapping:
16        beans: mbeans
```

### 3. 可以查看健康健康的链接：访问：<http://127.0.0.1:8080/actuator>

```
1 {
2   "_links": {
3     "self": {
4       "href": "http://127.0.0.1:8080/actuator",
5       "templated": false
6     },
7     "beans": {
8       "href": "http://127.0.0.1:8080/actuator/beans",
9       "templated": false
10    },
11  }
```

```
11     "caches-cache": {
12         "href": "http://127.0.0.1:8080/actuator/caches/{cache}",
13         "templated": true
14     },
15     "caches": {
16         "href": "http://127.0.0.1:8080/actuator/caches",
17         "templated": false
18     },
19     "health": {
20         "href": "http://127.0.0.1:8080/actuator/health",
21         "templated": false
22     },
23     "health-path": {
24         "href": "http://127.0.0.1:8080/actuator/health/{*path}",
25         "templated": true
26     },
27     "info": {
28         "href": "http://127.0.0.1:8080/actuator/info",
29         "templated": false
30     },
31     "conditions": {
32         "href": "http://127.0.0.1:8080/actuator/conditions",
33         "templated": false
34     },
35     "configprops": {
36         "href": "http://127.0.0.1:8080/actuator/configprops",
37         "templated": false
38     },
39     "env": {
40         "href": "http://127.0.0.1:8080/actuator/env",
41         "templated": false
42     },
43     "env-toMatch": {
44         "href": "http://127.0.0.1:8080/actuator/env/{toMatch}",
45         "templated": true
46     },
47     "loggers": {
48         "href": "http://127.0.0.1:8080/actuator/loggers",
49         "templated": false
50     },
51     "loggers-name": {
52         "href": "http://127.0.0.1:8080/actuator/loggers/{name}",
53         "templated": true
54     },
55     "heapdump": {
56         "href": "http://127.0.0.1:8080/actuator/heapdump",
57         "templated": false
58     },
59     "threaddump": {
60         "href": "http://127.0.0.1:8080/actuator/threaddump",
61         "templated": false
62     },
```

```
63         "metrics-requiredMetricName": {
64             "href": "http://127.0.0.1:8080/actuator/metrics/{requiredMetricName}",
65             "templated": true
66         },
67         "metrics": {
68             "href": "http://127.0.0.1:8080/actuator/metrics",
69             "templated": false
70         },
71         "scheduledtasks": {
72             "href": "http://127.0.0.1:8080/actuator/scheduledtasks",
73             "templated": false
74         },
75         "mappings": {
76             "href": "http://127.0.0.1:8080/actuator/mappings",
77             "templated": false
78         }
79     }
80 }
```

#### 4. 常见的监控端点

端点	描述
auditevents	获取当前应用暴露的审计事件信息
beans	获取应用中所有的 Spring Beans 的完整关系列表
caches	获取公开可以用的缓存
conditions	获取自动配置条件信息，记录哪些自动配置条件通过和没通过的原因
configprops	获取所有配置属性，包括默认配置，显示一个所有 @ConfigurationProperties 的整理列版本
env	获取所有环境变量
flyway	获取已应用的所有Flyway数据库迁移信息，需要一个或多个 Flyway Bean
liquibase	获取已应用的所有Liquibase数据库迁移。需要一个或多个 Liquibase Bean
health	获取应用程序健康指标（运行状况信息）
httptrace	获取HTTP跟踪信息（默认情况下，最近100个HTTP请求-响应交换）。需要 HttpTraceRepository Bean
info	获取应用程序信息
integrationgraph	显示 Spring Integration 图。需要依赖 spring-integration-core
loggers	显示和修改应用程序中日志的配置
logfile	返回日志文件的内容（如果已设置logging.file.name或logging.file.path属性）
metrics	获取系统度量指标信息
mappings	显示所有@RequestMapping路径的整理列表
scheduledtasks	显示应用程序中的计划任务
sessions	允许从Spring Session支持的会话存储中检索和删除用户会话。需要使用 Spring Session的基于Servlet的Web应用程序
shutdown	关闭应用，要求endpoints.shutdown.enabled设置为true， 默认为 false
threaddump	获取系统线程转储信息
headdump	返回hprof堆转储文件
jolokia	通过HTTP公开JMX bean（当Jolokia在类路径上时，不适用于 WebFlux）。需要依赖 jolokia-core
prometheus	以Prometheus服务器可以抓取的格式公开指标。需要依赖 micrometer-registry-prometheus

## 2.11.2 定制端点

定制端点一般通过endpoints+端点名+属性名来设置，如：

```
1 management:
2   endpoint:
3     # 显示完整的健康状态
4     health:
5       show-details: always
6     beans:
7       enabled: false
```

## 三. 前后端分离开发

教学目标：

1. 了解前后端分离开发的优势；
2. 掌握Postman工具的使用；
3. 掌握Vue、Axios等前端框架的使用；
4. 掌握跨域请求的处理。

在没有实行前后端分离时，我们会看到：

**开发人员：**前端、后端技术日益专业化、复杂化，一个人很难在短时间内同时精通前端的开发技术。专业的事情应该交给专业的人去做，开发人员应该在各自擅长的领域不断深造；

**项目主管：**开发团队偏后台，项目功能很强大，但界面很丑，难以满足领导、客户的需求；产品需求天天变，改个页面加个版本。

**运维人员：**一套代码打天下，CDN搞不了，服务器永远不够用。加个活动页面也要全服上线。

**公司领导：**全栈工程师太难找了，好不容易招来一个还怕跑。

人类社会的本质是分工协作！

我们需要将前端与后台分离：前端工程师专注页面开发，后端工程师专注业务与数据，前、后端通过Ajax异步调用完成数据交互。

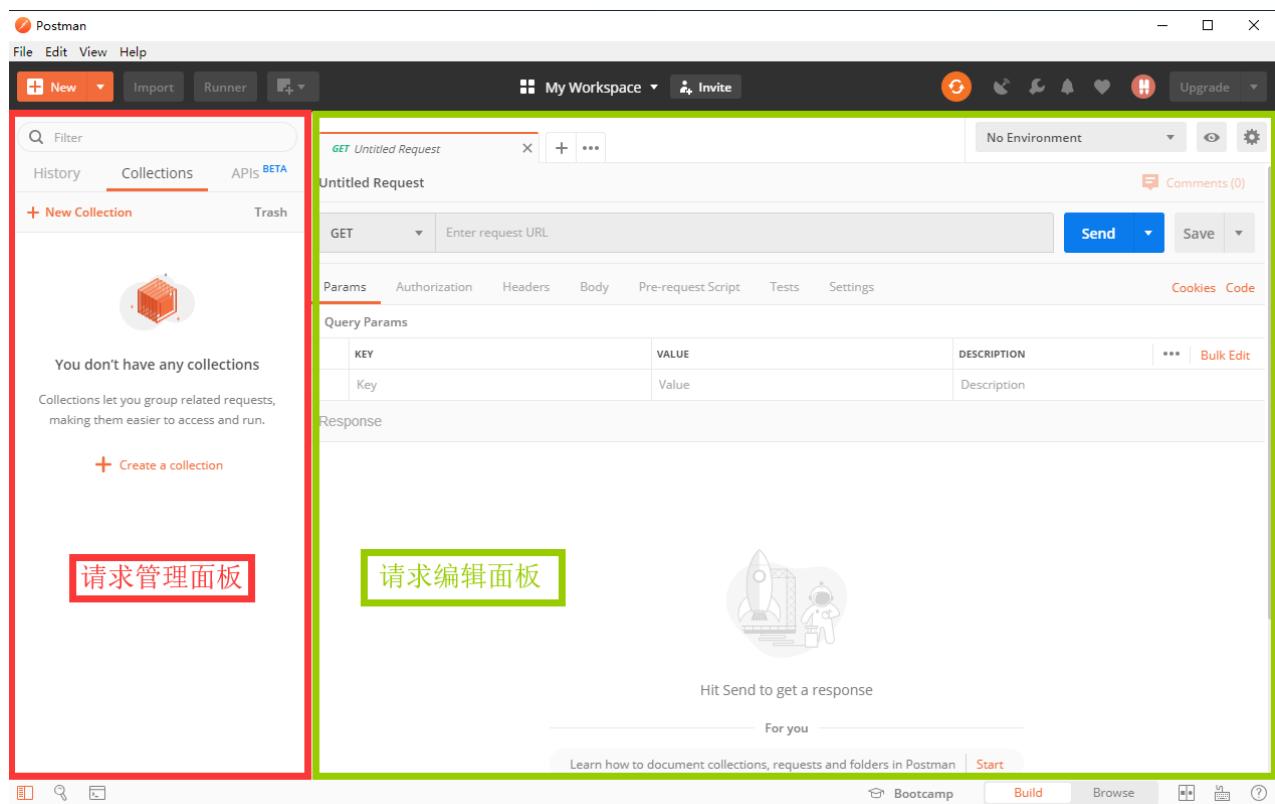
### 3.1 Postman

前后端分离后，后端工程师只需要对外提供数据接口，而不用关心页面视图了。但这也带了一些麻烦：没有了页面，我们该如何测试新开发的接口是否有BUG呢？这时候就该Postman闪亮登场啦！

Postman一款非常流行的API调试工具。Postman早起是作为chrome浏览器插件存在的，现在官方提供了独立的安装包，不再依赖于Chrome浏览器了。推荐大家使用独立安装包的方式安装。

下载地址：<https://www.getpostman.com/downloads>

安装很简单，保持默认即可。运行Postman可以看到：



在左侧面板可以集中管理接口列表，右侧是具体请求的编辑、测试面板：

The screenshot shows the Postman application interface. On the left, there's a sidebar with options like 'New Collection' (highlighted with a red box), 'Share Collection', 'Manage Roles', 'Rename', 'Edit', 'Create a fork', 'Merge changes', 'Add Request' (highlighted with a red box), 'Add Folder', 'Duplicate', 'Export', 'Monitor Collection', 'Mock Collection', 'Publish Docs', 'Remove from workspace', 'Delete', and 'Del'. A red arrow points from the 'Add Request' button to the 'Params' section of a request configuration window on the right. Another red arrow points from the 'Add Request' button to the 'Query Params' section. The main window shows a 'GET Untitled Request' with 'Enter request URL' set to 'GET' and 'No Environment'. It also includes sections for 'Params', 'Authorization', 'Headers', 'Body', 'Pre-request Script', 'Tests', 'Settings', 'Cookies', and 'Code'. A central area says 'Hit Send to get a response'.

## 发起测试请求：

This screenshot shows a completed API request in Postman. The request is named '登录接口测试' and is of type 'POST' (highlighted with a yellow box). The 'Request Address' (highlighted with a green box) is 'http://127.0.0.1/user/login?username=zahnsgan&password=123456'. The 'Query Params' section (highlighted with a blue box) contains two entries: 'username' with value 'zahnsgan' and 'password' with value '123456'. Below the request details, the 'Body' tab is selected, showing a JSON response: { "msg": "登录成功!", "code": 0 }. The status bar at the bottom indicates 'Status: 200 OK', 'Time: 115ms', and 'Size: 164 B'. A pink box highlights the entire response body area.

## 3.2 Axios

Axios是近年来备受推崇的一个网络请求库，它以基于Promise的方式封装了浏览器的XMLHttpRequest和服务器端node http请求，使得我们可以用es6推荐的异步方式处理网络请求。

发起Get请求：

```
1 // Make a request for a user with a given ID
2 axios.get('/user?ID=12345')
3   .then(function (response) {
4     // handle success
5     console.log(response);
6   })
7   .catch(function (error) {
8     // handle error
9     console.log(error);
10  })
11  .finally(function () {
12    // always executed
13  });
14
15 // Optionally the request above could also be done as
16 axios.get('/user', {
17   params: {
18     ID: 12345
19   }
20 })
21   .then(function (response) {
22     console.log(response);
23   })
24   .catch(function (error) {
25     console.log(error);
26   })
27   .finally(function () {
28     // always executed
29   });
30
31 // Want to use async/await? Add the `async` keyword to your outer
32 // function/method.
33 async function getUser() {
34   try {
35     const response = await axios.get('/user?ID=12345');
36     console.log(response);
37   } catch (error) {
38     console.error(error);
39   }
}
```

发起Post请求：

```
1 axios.post('/user', {
2     firstName: 'Fred',
3     lastName: 'Flintstone'
4 })
5     .then(function (response) {
6         console.log(response);
7     })
8     .catch(function (error) {
9         console.log(error);
10    });

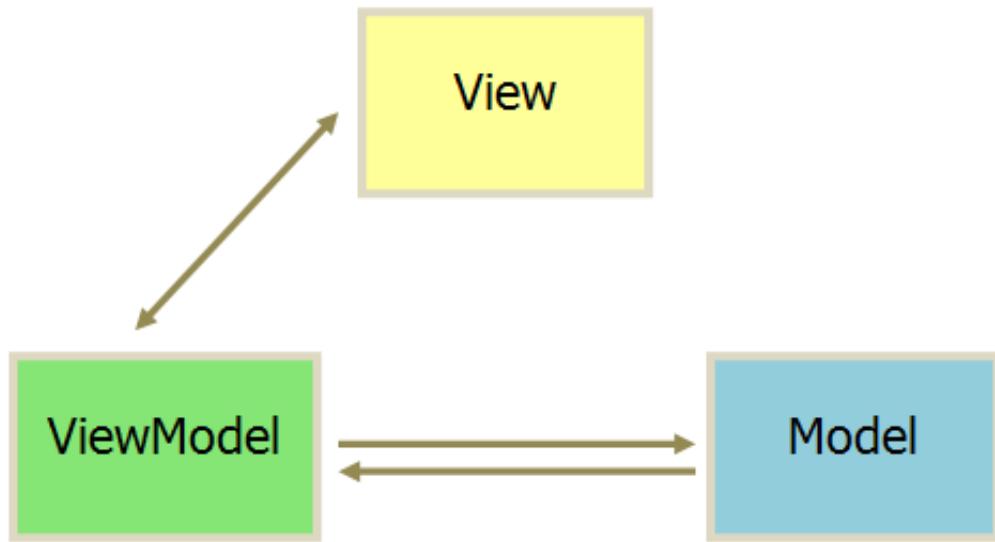
```

原生的HTTP请求：

```
1 // Send a POST request
2 axios({
3     method: 'post',
4     url: '/user/12345',
5     data: {
6         firstName: 'Fred',
7         lastName: 'Flintstone'
8     }
9 }).then(function (response) {
10     console.log(response);
11 });

```

### 3.3 VueJS



MVVM是Model-View-ViewModel的简写，本质上就是MVC的改进版。它采用双向绑定（data-binding）：View的变动，自动反映在ViewModel，反之亦然。

Vue是一套实现了MVVM模式的用于构建用户界面的渐进式JavaScript框架。Vue的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。

### 3.3.1 单文件应用

官网地址：<https://cn.vuejs.org>

Vue是一款前端的开发框架，开发流程是：①在HTML文档中设置挂载点（标签元素）；②在JavaScript中初始化Vue实例，添加到挂载点；③开发页面组件。

#### 1. 设置挂载点

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <title>Vue开发示例</title>
6  </head>
7  <body>
8      <!-- 挂载点其实就是一个标签元素 -->
9      <div id="app">
10     ...

```

```
11      </div>
12  </body>
13  <script src="vue.min.js"></script>
14  <script>
15    new Vue({
16      el: "#app"; // 通过设置挂载点
17      data: {
18        "number": 123 // 添加数据
19      }
20    });
21  </script>
22  </html>
```

## 2. 初始化Vue实例

```
1  new Vue({
2    el: "#app"; // 通过设置挂载点
3    data: {
4      "number": 123 // 添加数据
5    }
6  });
```

## 3. 开发页面组件

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title>Vue开发示例</title>
6  </head>
7  <body>
8    <div id="app">
9      <!-- 通过指令填充标签文本 -->
10     <span v-text="number"></span>
11   </div>
12 </body>
13 </html>
```

Vue可以非常方便的渲染网页内容，可以实现类似JSP中JSTL的功能，如：

条件渲染：

```
1 <div v-if="type === 'A'">
2   A
3 </div>
4 <div v-else-if="type === 'B'">
5   B
6 </div>
7 <div v-else-if="type === 'C'">
8   C
9 </div>
10 <div v-else>
11   Not A/B/C
12 </div>
```

列表渲染：

```
1 <ul id="example-1">
2   <li v-for="item in items">
3     {{ item.message }}
4   </li>
5 </ul>
```

属性绑定：

```
1 <div
2   class="static"
3   :class="{ active: isActive, 'text-danger': hasError }"
4 ></div>
```

Vue 更重要的是组件化开发，详细内容可以继续查阅官方文档！

### 3.3.2 VUE-CLI

1. 安装vue-cli：

```
1 npm install -g @vue/cli
```

2. 创建项目

```
1 vue create hello-world
```

默认情况下，打包后的文件如：css、js会使用绝对路径。如果要是用相对路径，则：

在项目根目录下，创建 `vue.config.js` 文件，内容如下：

```
1 module.exports = {
2   // 基本路径
3   publicPath: './',
4   // 生产环境是否生成 sourceMap 文件
5   productionSourceMap: false
6 }
```

### 3. 安装 Vue Router

```
1 npm i vue-router
```

在项目 `src` 目录下创建文件: `app.router.js`

```
1 import Vue from 'vue'
2 import VueRouter from 'vue-router'
3 import Login from "@/components/Login";
4 import Home from "@/components/Home";
5
6 Vue.use(VueRouter)
7
8 export default new VueRouter({
9   routes: [
10     {path: "/", component: Login},
11     {path: "/home", component: Home}
12   ]
13 })
```

在 `main.js` 中导入:

```
1 import router from './app.router'
2
3 new Vue({
4   router,
5   render: h => h(App),
6 }).$mount('#app')
```

在 `App.vue` 中添加视图:

```
1 <template>
2   <div id="app">
3     <router-view></router-view>
4   </div>
5 </template>
```

### 4. 安装 vuex

```
1 npm i vuex
```

在项目 `src` 目录下创建文件: `app.store.js`

```
1 import Vue from 'vue'
2 import Vuex from 'vuex'
3
4 Vue.use(Vuex)
```

```
5
6  export default new Vuex.Store({
7    state: {
8      user: { "id": -1, "username": "", "avatar": "" }
9    },
10   mutations: {
11     loginSuccess(state, loginUser){
12       state.user.id = loginUser.id
13       state.user.username = loginUser.username
14       state.user.avatar = loginUser.avatar
15     }
16   }
17 })
```

在 `main.js` 中导入：

```
1 import Vue from 'vue'
2 import App from './App.vue'
3 import router from './app.router'
4 import store from './app.store'
5
6 Vue.config.productionTip = false
7
8 new Vue({
9   router,
10  store,
11  render: h => h(App),
12 }).$mount('#app')
```

## 3.4 跨域请求

前后端分离后，可能会部署在不同的服务器、不同的域名下。受浏览器默认安全限制的影响，不同域名之间是不允许发起异步请求（媒体资源、css、js除外）的。前端项目的Ajax请求将因此而调用失败。

处理跨域请求一般可以通过JSONP来解决，但JSONP有很多限制。在Spring Boot项目中，我们可以通过添加配置来解决。

导入组件：

```
1 @Configuration
2 public class MvcConfig implements WebMvcConfigurer {
3   /**
4    * 支持跨域（新版）
5    * @return
6    */
7   @Override
8   public void addCorsMappings(CorsRegistry registry) {
9     registry.addMapping("/**")
10    .allowedOrigins("*")
```

```
11         .allowedMethods("POST", "GET", "PUT", "OPTIONS", "DELETE")
12         .maxAge(3600)
13         .allowCredentials(true);
14     }
15 }
```

## AXIOS 通用配置：

```
1 // 保持SessionID
2 axios.defaults.withCredentials = true;
3
4 // 设置默认基础的服务端网址
5 axios.defaults.baseURL = 'https://api.example.com';
6
7 // 设置默认的通用请求头
8 axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
9
10 // 设置Post请求头(默认即为 application/json, 可以省略好好改配置)
11 axios.defaults.headers.post['Content-Type'] = 'application/json';
12
13 // 添加请求拦截器
14 axios.interceptors.request.use(function (config) {
15     // 在请求发起之前修改配置
16     return config;
17 }, function (error) {
18     // 在请求失败时执行操作
19     return Promise.reject(error);
20 });
21
22 // 添加响应拦截器
23 axios.interceptors.response.use(function (response) {
24     // 在HTTP响应状态码为2XX时执行
25     // 可以操作响应数据
26     return response;
27 }, function (error) {
28     // 在HTTP响应状态码不为2XX时执行
29     // 可以操作响应的错误信息
30     return Promise.reject(error);
31 });
```

## 四. 上线部署

---

## 4.1 虚拟机

为了后续部署、演示方便，我们可以在当前主机上安装 **VirtualBox** 虚拟化软件。在 **VirtualBox** 中安装虚拟的 **CentOS** 主机。然后在虚拟的 **CentOS** 主机中安装 **Docker**。

**技术选型：**

- VirtualBox-6.x
- CentOS-7-x86\_64-Minimal-x
- jdk-11.x

在安装虚拟机之前，需要在主机的 **BIOS** 中开启 **CPU** 虚拟化。

## 4.2 Docker

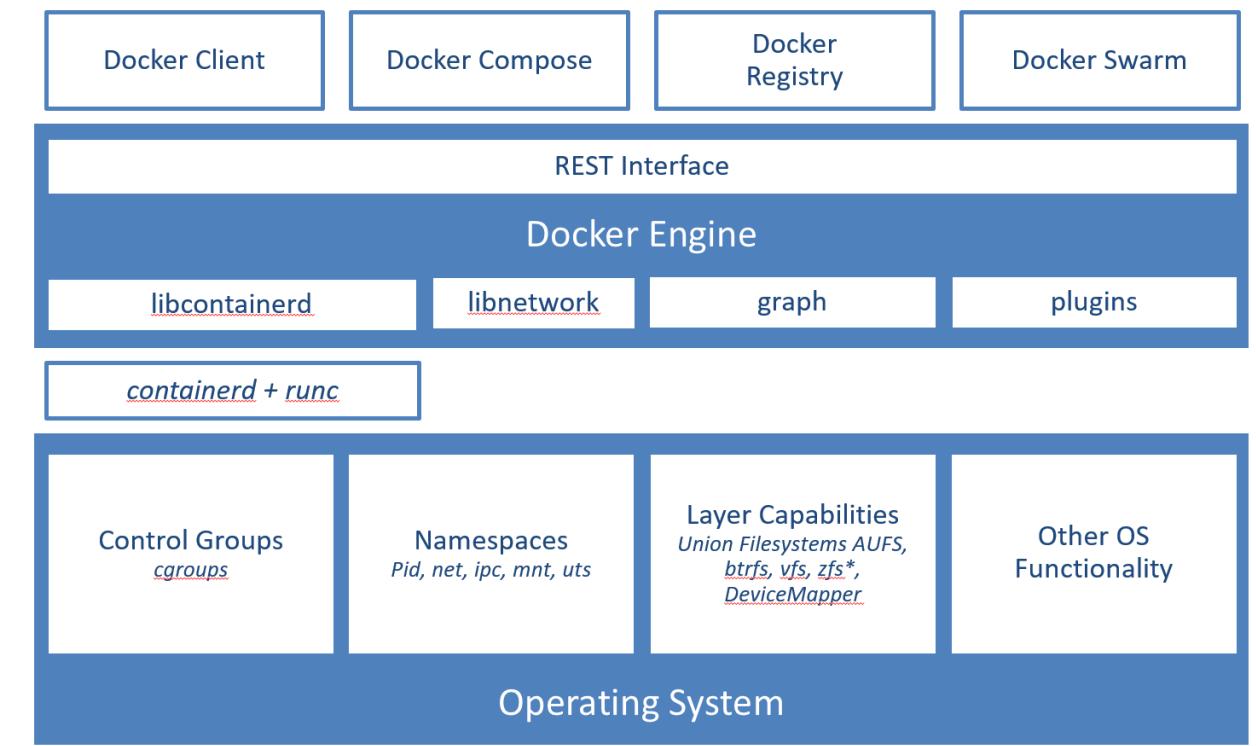
Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化，容器是完全使用沙箱机制，相互之间不会有任何接口。

Docker 最初是 dotCloud 公司创始人 Solomon Hykes 在法国期间发起的一个公司内部项目，它是基于 dotCloud 公司多年云服务技术的一次革新，并于 2013 年 3 月以 Apache 2.0 授权协议开源，主要项目代码在 GitHub 上进行维护。Docker 项目后来还加入了 Linux 基金会，并成立推动开放容器联盟（OCI）。

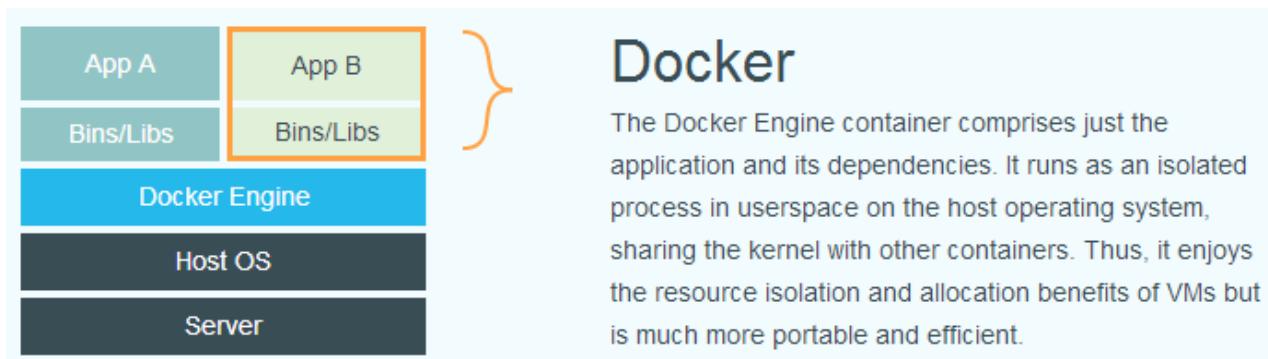
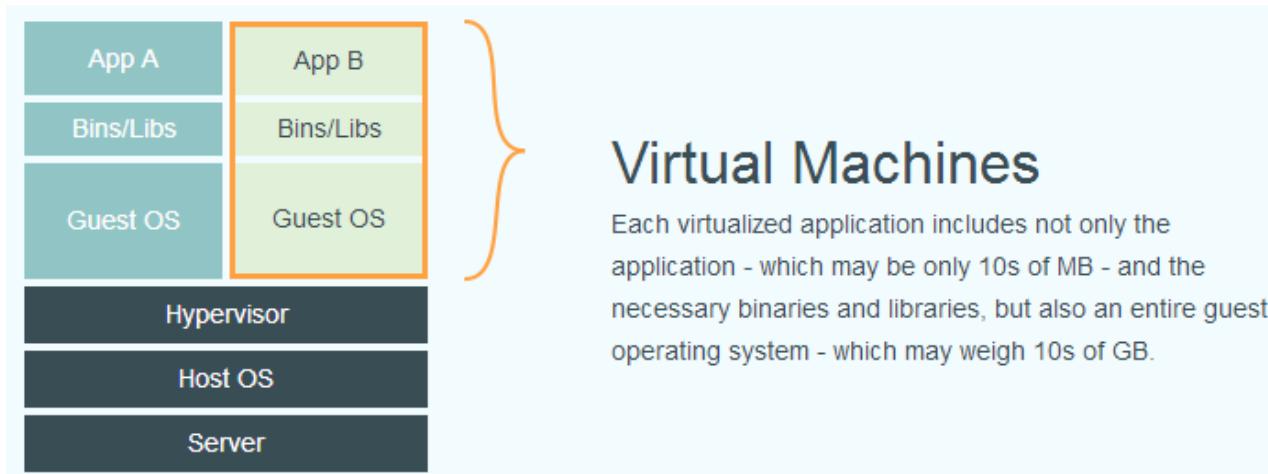
Docker 自开源后受到广泛的关注和讨论，至今其 GitHub 项目已经超过了 5 万 4 千个星标和一万多个 fork。甚至由于 Docker 项目的火爆，在 2013 年底，dotCloud 公司决定改名为 Docker。Docker 最初是在 Ubuntu 12.04 上开发实现的；Red Hat 则从 RHEL 6.5 开始对 Docker 进行支持；Google 也在其 PaaS 产品中广泛应用 Docker。

Docker 使用 Google 公司推出的 Go 语言进行开发实现，基于 Linux 内核的 cgroup、namespace，以及 AUFS 类的 Union FS 等技术，对进程进行封装隔离，属于操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其他的隔离的进程，因此也称其为容器。最初实现是基于 LXC，从 0.7 版本以后开始去除 LXC，转而使用自行开发的 libcontainer，从 1.11 开始，则进一步演进为使用 runC 和 containerd。

# Architecture In Linux



虚拟化与容器化的区别:



## 参考资料：

Docker安装文档：<https://docs.docker.com/install/linux/docker-ce/centos/>

Docker官方仓库：<https://hub.docker.com/>

国内加速：参见阿里云、网易云、DaoCloud等公司产品介绍

### 4.2.1 Docker镜像

要是用Docker部署项目，首先需要制作镜像文件。可以使用Dockerfile文件构建。

准备工作：

1. 拉取最新的CentOS基础镜像

```
1 docker pull centos:7
```

2. 创建工作目录：`/data/works/docker_jdk`

3. 将JDK `jdk-11.0.8_linux-x64_bin.tar.gz` 拷贝至工作目录

在工作目录下创建 `Dockerfile`

```
1 FROM centos:7
2 MAINTAINER LetCode letcode.net
3
4 ADD ./jdk-11.0.8_linux-x64_bin.tar.gz /opt
5
6 ENV JAVA_HOME /opt/jdk-11.0.8
7 ENV JRE_HOME $JAVA_HOME/jre
8 ENV CLASSPATH .:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
9 ENV PATH $PATH:$JAVA_HOME/bin
10
11 EXPOSE 8080
```

执行构建命令：

```
1 docker build -t java:11 .
```

### 4.2.2 容器部署

1. 创建工作目录：`/data/works/sb_test`
2. 拷贝打包好的文件至工作目录，包括：`app.jar`、`app_lib`、`application.yml`
3. 在工作目录创建 `logs` 文件夹，用于存放日志文件（需在 `application.yml` 文件中指定日志路径）
4. 创建启动脚本：`run.sh`，内容如下：

```
1  #!/bin/bash  
2  java -jar /data/works/app.jar
```

5. 为启动脚本添加权限:

```
1  chmod +x ./run.sh
```

6. 运行容器

```
1  docker run -d --name sb_api_test \  
2          -p 8080:8080 \  
3          -v /data/works/api_test:/data/works/ \  
4          java:11 /data/works/run.sh
```

## 五. 中间件

---

### 5.1 消息服务

消息服务的作用：异步、解耦、削峰。

- 异步处理：多应用对消息队列中同一消息进行处理，应用间并发处理消息，相比串行处理，减少处理时间；
- 应用耦合：多应用间通过消息队列对同一消息进行处理，避免调用接口失败导致整个过程失败；
- 限流削峰：广泛应用于秒杀或抢购活动中，避免流量过大导致应用系统挂掉的情况；
- 消息驱动的系统：系统分为消息队列、消息生产者、消息消费者，生产者负责产生消息，消费者(可能有多个)负责对消息进行处理；

消息服务的概念：消息代理（Message Broker）、目的地（Destination）：

当消息发送者发送消息以后，将由消息代理接管，消息代理保证消息传递到指定目的地。消息代理也就是消息服务器（消息队列服务器）。

#### 5.1.1 消息队列

消息队列主要有两种形式的目的地：

1. 队列（queue）：点对点的消息通信；
  - 消息发送者发送消息，消息代理将其放入一个队列中，消息接收者从队列中获取消息内容，消息读取后被移出队列；
  - 消息只有唯一的发送者和接受者，但并不是说只能有一个接收者。即可以有多个接受者，但

他们不能消费同一条消息。

## 2. 主题 (topic) : 发布 (publish) / 订阅 (subscribe) 消息通信。

- 发送者 (发布者) 发送消息到主题, 多个接收者 (订阅者) 监听 (订阅) 这个主题, 那么就会在消息到达时同时收到消息;
- 消息可以有多个接收者, 即多个接受者, 可以消费同一条消息。

## 主要规范:

### 1. JMS

- 基于JVM消息代理的规范;
- ActiveMQ、Kafka、HornetMQ是JMS实现。

### 2. AMQP

- 高级消息队列协议, 也是一个消息代理的规范, 兼容JMS;
- RabbitMQ、RocketMQ是AMQP的实现。

	JMS	AMQP
定义	Java api	网络线级协议
跨语言	否	是
跨平台	否	是
Model	提供两种消息模型: (1) 、Peer-2-Peer (2) 、Pub/sub	提供了五种消息模型: (1) 、direct exchange (2) 、fanout exchange (3) 、topic change (4) 、headers exchange (5) 、system exchange 本质来讲, 后四种和JMS的pub/sub模型没有太大差别, 仅是在路由机制上做了更详细的划分;
支持消息类型	多种消息类型: TextMessage MapMessage BytesMessage StreamMessage ObjectMessage Message (只有消息头和属性)	byte[] 当实际应用时, 有复杂的消息, 可以将消息序列化后发送。
综合评价	JMS 定义了JAVA API层面的标准; 在java体系中, 多个client 均可以通过JMS进行交互, 不需要应用修改代码, 但是其对跨平台的支持较差;	AMQP定义了wire-level层的协议标准; 天然具有跨平台、跨语言特性。

## Spring支持:

- spring-jms提供了对JMS的支持

- spring-rabbit提供了对AMQP的支持
- 需要ConnectionFactory的实现来连接消息代理
- 提供JmsTemplate、RabbitTemplate来发送消息
- @JmsListener (JMS) 、@RabbitListener (AMQP) 注解在方法上监听消息代理发布的消息
- @EnableJms、@EnableRabbit开启支持

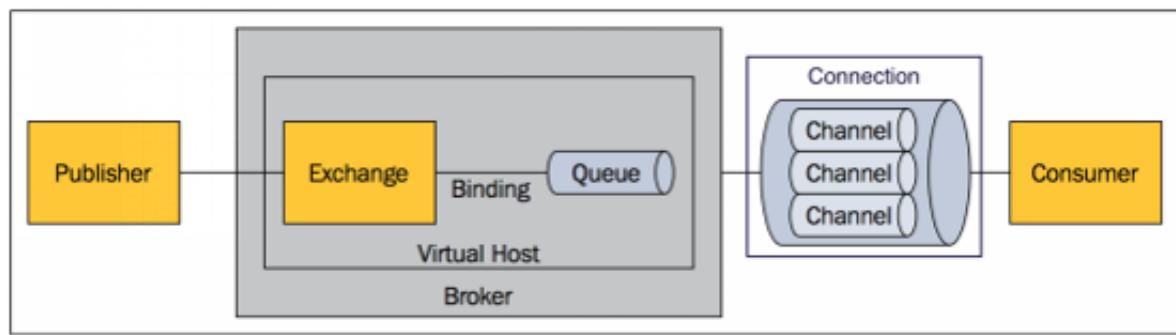
### Spring Boot自动配置：

- JmsAutoConfiguration
- RabbitAutoConfiguration

## 5.1.2 RabbitMQ

RabbitMQ是一个由erlang开发的AMQP(Advanced Message Queue Protocol)的开源实现。

### 5.1.2.1 核心概念



### Message

消息，消息是不具名的，它由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括routing-key（路由键）、priority（相对于其他消息的优先权）、delivery-mode（指出该消息可能需要持久性存储）等。

### Publisher

消息的生产者，也是一个向交换器发布消息的客户端应用程序。

### Exchange

交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列。

Exchange有4种类型：direct(默认), fanout, topic, 和headers, 不同类型的Exchange转发消息的策略有所区别

### Queue

消息队列，用来保存消息直到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。

### Binding

绑定，用于消息队列和交换器之间的关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。  
Exchange 和Queue的绑定可以是多对多的关系。

### Connection

网络连接，比如一个TCP连接。

### Channel

信道，多路复用连接中的一条独立的双向数据流通道。信道是建立在真实的TCP连接内的虚拟连接，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接。

### Consumer

消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。

### Virtual Host

虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身分认证和加密环境的独立服务器域。每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。vhost 是 AMQP 概念的基础，必须在连接时指定，RabbitMQ 默认的 vhost 是 / 。

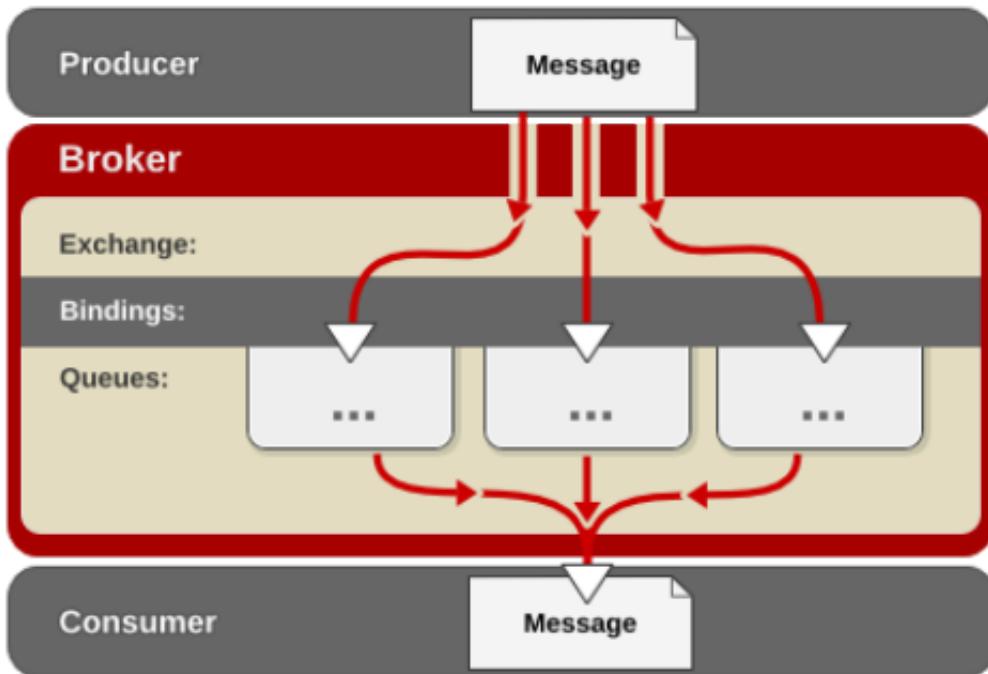
### Broker

表示消息队列服务器实体

### 5.1.2.2 运行机制

AMQP 中增加了 Exchange 和 Binding 的角色。生产者把消息发布到 Exchange 上，消息最终到达队列并被消费者接收，而 Binding 决定交换器的消息应该发送到那个队列。

## Producer Consumer



### Exchange (交换器) 类型：

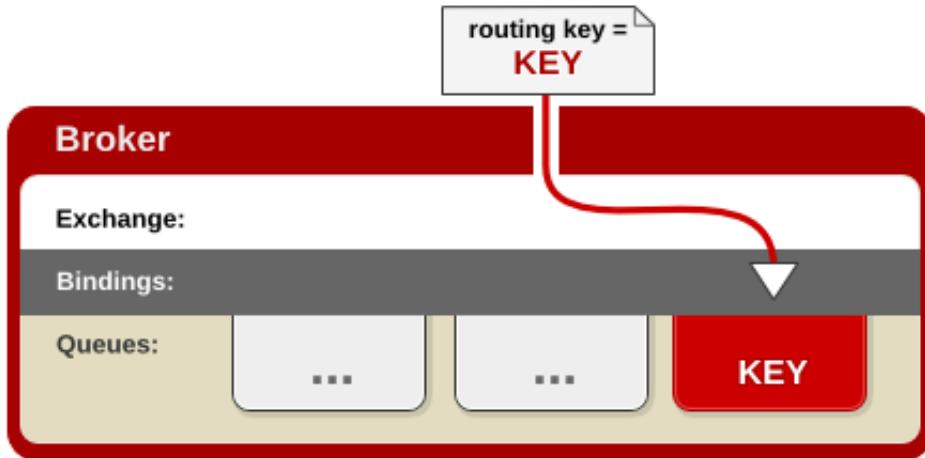
Exchange分发消息时根据类型的不同分发策略有区别，目前共四种类  
型：direct、fanout、topic、headers。

headers 匹配 AMQP 消息的 header 而不是路由键，headers 交换器和 direct 交换器完全一致，但性能差很多，目前几乎用不到了，所以只需关注三种类型。

**direct :**

消息中的路由键（routing key）如果和 Binding 中的 binding key 一致，交换器就将消息发到对应的队列中。路由键与队列名完全匹配，如果一个队列绑定到交换机要求路由键为“dog”，则只转发 routing key 标记为“dog”的消息，不会转发“dog.puppy”，也不会转发“dog.guard”等等。它是完全匹配、单播的模式。

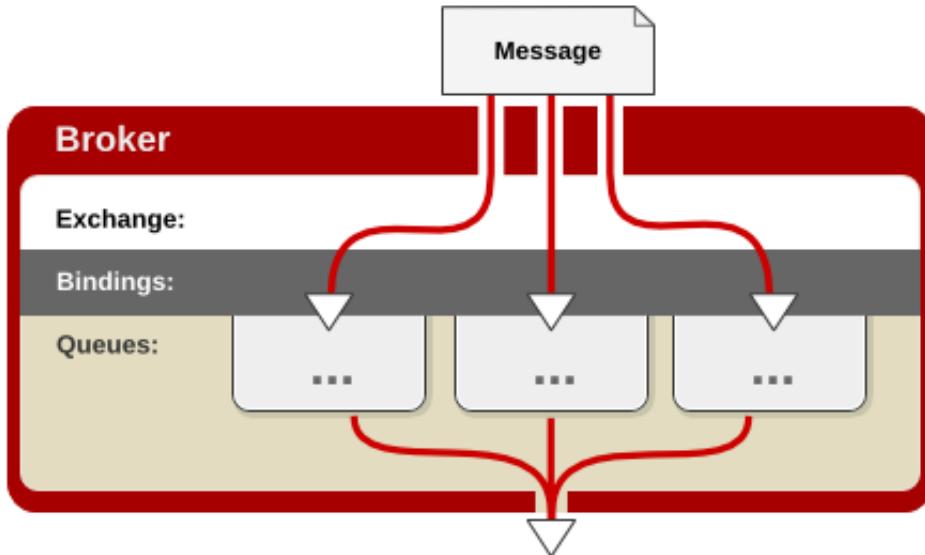
## Direct Exchange



fanout:

每个发到 fanout 类型交换器的消息都会分到所有绑定的队列上去。fanout 交换器不处理路由键，只是简单的将队列绑定到交换器上，每个发送到交换器的消息都会被转发到与该交换器绑定的所有队列上。很像子网广播，每台子网内的主机都获得了一份复制的消息。fanout 类型转发消息是最快的。

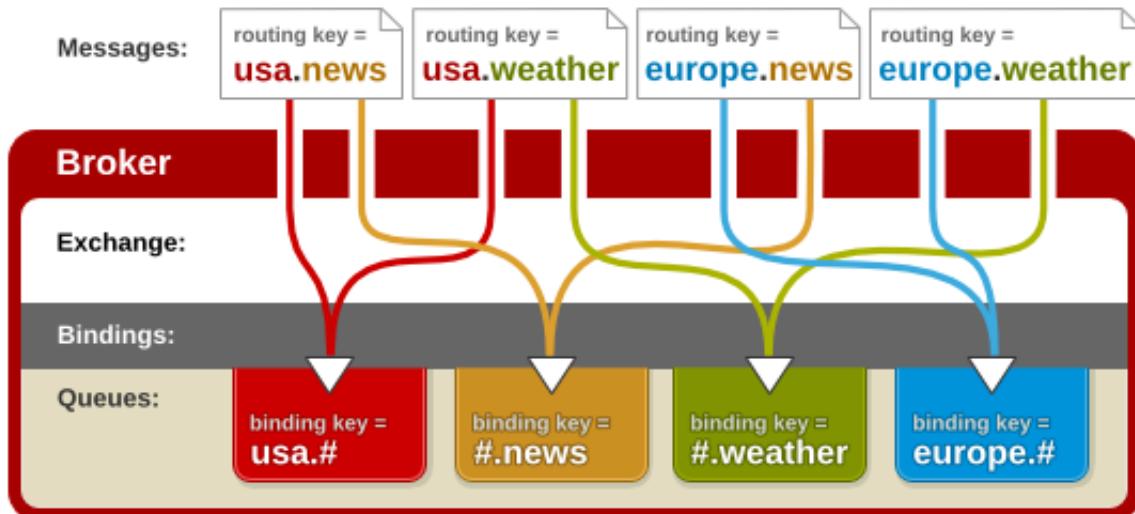
## Fanout Exchange



topic:

topic 交换器通过模式匹配分配消息的路由键属性，将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。它将路由键和绑定键的字符串切分成单词，这些单词之间用点隔开。它同样也会识别两个通配符：符号 # 和符号 \*。# 匹配 0 个或多个单词，\* 匹配一个单词。

## Topic Exchange



### 5.1.2.3 安装RabbitMQ

docker 安装：

拉取镜像：

```
1 docker pull rabbitmq:management
```

运行容器：

```
1 docker run -d --name rabbitmq \
2     -p 5672:5672 \
3     -p 15672:15672 \
4     -e RABBITMQ_VMQ_MEMORY_HIGH_WATERMARK=0.6 \
5     rabbitmq:management
```

注意： `RABBITMQ_VMQ_MEMORY_HIGH_WATERMARK` 参数用于设置内存限制

管理后台：

容器运行成功后，可以访问管理后台：<http://192.168.56.10:15672>

默认用户名和密码都是：`guest`

RabbitMQ Management

RabbitMQ 3.8.7 Erlang 23.0.3

Overview Connections Channels Exchanges Queues Admin

Refreshed 2020-08-29 00:17:18 Refresh every 5 seconds

Virtual host All Cluster rabbit@[a0221ca9b474](#) User guest Log out

**Overview**

Totals

Queued messages last minute ?

Currently idle

Message rates last minute ?

Currently idle

Global counts ?

Connections: 0 Channels: 0 Exchanges: 7 Queues: 0 Consumers: 0

Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats	+/-
rabbit@ <a href="#">a0221ca9b474</a>	99 1048576 available	0 943629 available	545 1048576 available	98 MiB 800 MiB high watermark	142 GiB MiB low watermark	2m 17s	basic disc 2 rss	This node All nodes	

Churn statistics

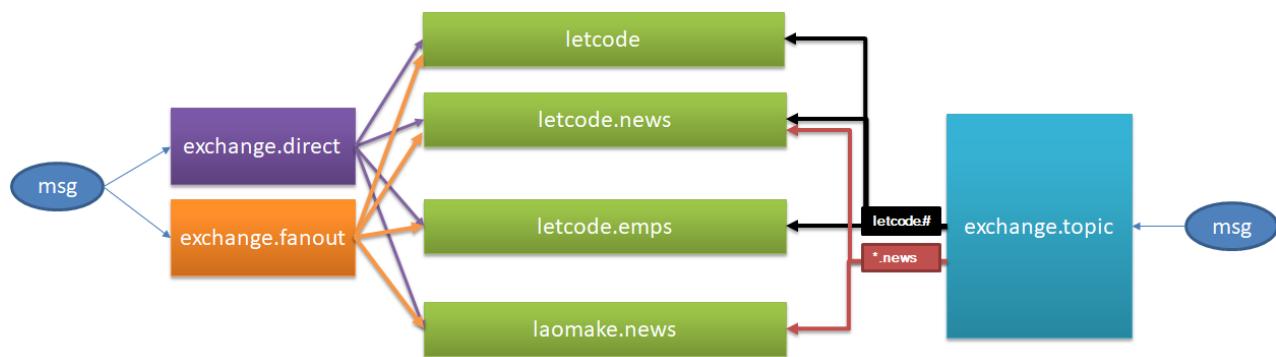
Ports and contexts

Export definitions

Import definitions

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

#### 5.1.2.4 发送消息



首先，我们创建三个交换器：

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D			
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
exchange.direct	direct	D			
exchange.fanout	fanout	D			
exchange.topic	topic	D			

## 创建交换器： exchange.direct

The screenshot shows the RabbitMQ Management interface with the 'Exchanges' tab selected. A red box highlights the 'exchange.direct' row in the list of exchanges. Below the list, a form is open for creating a new exchange:

- Name:** exchange.direct
- Type:** direct (selected from a dropdown)
- Durability:** Durable (selected from a dropdown)
- Auto delete:** No (selected from a dropdown)
- Internal:** No (selected from a dropdown)
- Arguments:** An empty string field with a dropdown menu set to 'String'.

At the bottom of the form, there is a button labeled 'Add exchange'.

## 创建交换器： exchange.fanout

RabbitMQ Management

RabbitMQ 3.8.7 Erlang 23.0.3

Overview Connections Channels Exchanges Queues Admin

Page 1 of 1 - Filter:   Regex ?

Refreshing 2020-08-29 00:24:36 Refresh every 5 seconds

Virtual host / Cluster rabbit@**a0221ca9b474** User guest Log out

Displaying 8 items , page size up to: 100

Name	Type	Features	Message rate in	Message rate out
(AMQP default)	direct	D		
amq.direct	direct	D		
amq.fanout	fanout	D		
amq.headers	headers	D		
amq.match	headers	D		
amq.rabbitmq.trace	topic	D T		
amq.topic	topic	D		
exchange.direct	direct	D		

Add a new exchange

Name: **exchange.fanout** Type: **fanout** Durability: **Durable**

Auto delete: No Internal: No Arguments:  =  String

Add Alternate exchange ?

Add exchange

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

## 创建交换器: exchange.topic

RabbitMQ Management

RabbitMQ 3.8.7 Erlang 23.0.3

Overview Connections Channels Exchanges Queues Admin

Page 1 of 1 - Filter:   Regex ?

Refreshing 2020-08-29 00:25:54 Refresh every 5 seconds

Virtual host / Cluster rabbit@**a0221ca9b474** User guest Log out

Displaying 9 items , page size up to: 100

Name	Type	Features	Message rate in	Message rate out
(AMQP default)	direct	D		
amq.direct	direct	D		
amq.fanout	fanout	D		
amq.headers	headers	D		
amq.match	headers	D		
amq.rabbitmq.trace	topic	D T		
amq.topic	topic	D		
exchange.direct	direct	D		
exchange.fanout	fanout	D		

Add a new exchange

Name: **exchange.topic** Type: **topic** Durability: **Durable**

Auto delete: No Internal: No Arguments:  =  String

Add Alternate exchange ?

Add exchange

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

接下来创建四个消息队列:

## Queues

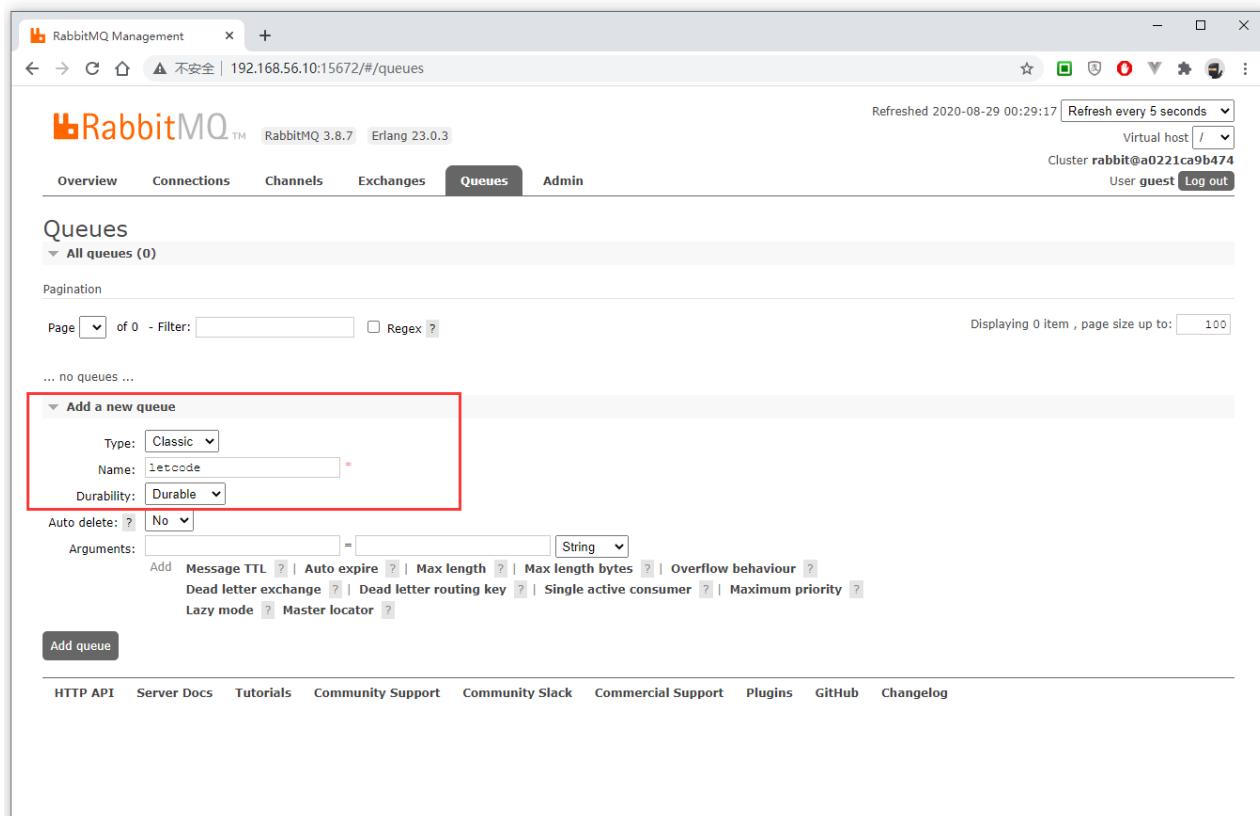
▼ All queues (4)

Pagination

Page  of 1 - Filter:   Regex ?

Overview				Messages			Message rates			+/-
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
laomake.news	classic	D Args	idle	0	0	0				
letcode	classic	D Args	idle	0	0	0				
letcode.emps	classic	D Args	idle	0	0	0				
letcode.news	classic	D Args	idle	0	0	0				

示例：



The screenshot shows the RabbitMQ Management interface at the 'Queues' page. There are four existing queues listed:

- laomake.news (classic, Durable, idle, 0 messages)
- letcode (classic, Durable, idle, 0 messages)
- letcode.emps (classic, Durable, idle, 0 messages)
- letcode.news (classic, Durable, idle, 0 messages)

Below the table, there is a section to "Add a new queue" with the following fields highlighted by a red box:

- Type: Classic
- Name: letcode
- Durability: Durable

Other visible settings include Auto delete: No, Arguments, and various message delivery options like Message TTL, Auto expire, Max length, etc.

交换器绑定消息队列：

为 `exchange.direct` 交换器绑定四个消息队列：

RabbitMQ Management

RabbitMQ 3.8.7 Erlang 23.0.3

Refreshed 2020-08-29 00:35:38 Refresh every 5 seconds

Virtual host /

Cluster rabbit@**a0221ca9b474**

User guest Log out

**Exchange: exchange.direct**

Overview

Message rates last minute ?

Currently idle

Details

Type	direct
Features	durable: true
Policy	

**Bindings**

This exchange

To Routing key Arguments Unbind

laomake.news	laomake.news		Unbind
letcode	letcode		Unbind
letcode.emps	letcode.emps		Unbind
letcode.news	letcode.news		Unbind

Add binding from this exchange

To queue :

## 示例：

RabbitMQ Management

RabbitMQ 3.8.7 Erlang 23.0.3

Refreshed 2020-08-29 00:34:29 Refresh every 5 seconds

Virtual host /

Cluster rabbit@**a0221ca9b474**

User guest Log out

**Exchange: exchange.direct**

Overview

Message rates last minute ?

Currently idle

Details

Type	direct
Features	durable: true
Policy	

**Bindings**

This exchange

... no bindings ...

Add binding from this exchange

To queue :  \*

Routing key:  letcode

Arguments:  =

**Bind**

▶ Publish message

▶ Delete this exchange

为 **exchange.fanout** 交换器绑定四个消息队列：

RabbitMQ Management

不安全 | 192.168.56.10:15672/#/exchanges/%2F/exchange.fanout

Refreshed 2020-08-29 00:37:44 Refresh every 5 seconds

Virtual host / Cluster rabbit@[a0221ca9b474](#)

User guest Log out

## Exchange: exchange.fanout

Overview

Message rates last minute ?

Currently idle

Details

Type	fanout
Features	durable: true
Policy	

Bindings

This exchange

To	Routing key	Arguments	Unbind
laomake.news	laomake.news		Unbind
letcode	letcode		Unbind
letcode.emps	letcode.emps		Unbind
letcode.news	letcode.news		Unbind

Add binding from this exchange

To queue :  \*

为 `exchange.topic` 交换器绑定四个消息队列：

注意修改路由键

RabbitMQ Management

不安全 | 192.168.56.10:15672/#/exchanges/%2F/exchange.topic

Refreshed 2020-08-29 00:41:22 Refresh every 5 seconds

Virtual host / Cluster rabbit@[a0221ca9b474](#)

User guest Log out

## Exchange: exchange.topic

Overview

Type: topic

Features	durable: true
Policy	

Bindings

This exchange

To	Routing key	Arguments	Unbind
laomake.news	*.news		Unbind
laomake.news	laomake.#		Unbind
letcode	letcode.#		Unbind
letcode.emps	letcode.#		Unbind
letcode.news	*.news		Unbind
letcode.news	letcode.#		Unbind

Add binding from this exchange

To queue :  \*

Routing key:

Arguments:  =  String

Bind

发送测试消息：

RabbitMQ Management

RabbitMQ 3.8.7 Erlang 23.0.3

Overview Connections Channels Exchanges Queues Admin

Refreshed 2020-08-29 00:44:03 Refresh every 5 seconds

Virtual host /

Cluster rabbit@[a0221ca9b474](#) User guest Log out

Exchange: exchange.direct

Overview Bindings

Publish message

Routing key: letcode

Headers: ? = String

Properties: ?

Payload: 这是exchange.direct发送给letcode的消息

Publish message

Delete this exchange

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

在消息队列中我们可以看到，已经收到了消息：

RabbitMQ Management

RabbitMQ 3.8.7 Erlang 23.0.3

Overview Connections Channels Exchanges Queues Admin

Refreshed 2020-08-29 00:44:58 Refresh every 5 seconds

Virtual host /

Cluster rabbit@[a0221ca9b474](#) User guest Log out

Queues

All queues (4)

Page 1 of 1 - Filter:   Regex ? Displaying 4 items , page size up to: 100

Overview		Messages			Message rates				
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
laomake.news	classic	D Args	idle	0	0	0			
letcode	classic	D Args	idle	1	0	1	0.00/s		
letcode.emps	classic	D Args	idle	0	0	0			
letcode.news	classic	D Args	idle	0	0	0			

Add a new queue

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

点击进入消息队列，可以获取消息：

The screenshot shows the RabbitMQ Management interface. The top navigation bar includes tabs for Overview, Connections, Channels, Exchanges, Queues (selected), and Admin. Below the navigation is a sub-menu for Publish message, Get messages, Move messages, Delete, Purge, and Runtime Metrics (Advanced). The main content area is titled 'Get messages' with a warning: 'Warning: getting messages from a queue is a destructive action.' It contains fields for Ack Mode (set to 'Nack message requeue true'), Encoding ('Auto string / base64'), and Messages ('1'). A large red box highlights the 'Get Message(s)' button and the resulting message details. The message details show an Exchange: 'exchange.direct', Routing Key: 'letcode', Redelivered: '0', Properties: 'delivery\_mode: 2 headers:', and Payload: '46 bytes'. The payload is described as '这是exchange.direct发送给letcode的消息'. At the bottom of the page are links for HTTP API, Server Docs, Tutorials, Community Support, Community Slack, Commercial Support, Plugins, GitHub, and Changelog.

## 5.1.3 整合消息服务

### 5.1.3.1 基本使用

#### 1. 引入 spring-boot-starter-amqp

```
1 <dependencies>
2     <!-- SpringBoot web启动器 -->
3     <dependency>
4         <groupId>org.springframework.boot</groupId>
5         <artifactId>spring-boot-starter-web</artifactId>
6     </dependency>
7
8     <!-- SpringBoot amqp启动器 -->
9     <dependency>
10        <groupId>org.springframework.boot</groupId>
11        <artifactId>spring-boot-starter-amqp</artifactId>
12    </dependency>
13
14    <!-- SpringBoot 测试启动器 -->
15    <dependency>
16        <groupId>org.springframework.boot</groupId>
17        <artifactId>spring-boot-starter-test</artifactId>
18    </dependency>
```

```
19    </dependencies>
```

## 2. 修改配置文件

```
1  spring:
2    rabbitmq:
3      host: 192.168.56.10
4      port: 5672
5      username: guest
6      password: guest
7      virtual-host: /
```

## 3. 启动类添加注解

```
1  @EnableRabbit
2  @SpringBootApplication
3  public class Application {
4
5    public static void main(String[] args) {
6      SpringApplication.run(Application.class);
7    }
8
9  }
```

## 4. 测试发送

```
1  /**
2   * 消息队列测试
3   */
4  @RunWith(SpringRunner.class)
5  @SpringBootTest
6  public class MqTest {
7
8    @Autowired
9    RabbitTemplate rabbitTemplate;
10
11   @Test
12   public void testDirectSend(){
13     String exchange = "exchange.direct";
14     String routeKey = "letcode";
15
16     Map<String, Object> msg = new HashMap<>();
17     msg.put("title", "测试消息");
18     msg.put("content", "这是exchange.direct发送给letcode的消息");
19
20     rabbitTemplate.convertAndSend(exchange, routeKey, msg);
21   }
22 }
```

## 5. 测试接收

```
1  /**
```

```
2     * 消息队列测试
3     */
4     @RunWith(SpringRunner.class)
5     @SpringBootTest
6     public class MqTest {
7
8         @Autowired
9         RabbitTemplate rabbitTemplate;
10
11        @Test
12        public void testDirectReceive(){
13            String queue = "etcode";
14            Object msg = rabbitTemplate.receiveAndConvert(queue);
15            Map<String, Object> data = (Map<String, Object>) msg;
16            System.out.println(data);
17        }
18    }
```

可以自定义AQMP配置：

```
1  /**
2   * AMQP 自定义配置
3   * @author LetCode
4   * @since 1.0
5   */
6  @Configuration
7  public class AmqpConfig {
8
9      /**
10       * 将消息序列化方式，从默认的JDK序列化转为JSON
11       * @return
12       */
13      @Bean
14      public MessageConverter messageConverter(){
15          return new Jackson2JsonMessageConverter();
16      }
17
18  }
```

### 5.1.3.2 使用监听器

首先要确保启动类添加了 `@EnableRabbit` 注解；

```
1  @EnableRabbit
2  @SpringBootApplication
3  public class Application {
4
5      public static void main(String[] args) {
6          SpringApplication.run(Application.class);
7      }
8
9  }
```

编写服务类：

```
1  @Service
2  public class MqService {
3
4      /**
5       * 自动监听消息
6       * @param msg
7       */
8      @RabbitListener(queues = "letcode")
9      public void receiveMessage(Map<String, Object> msg){
10         System.out.print("收到消息：");
11         System.out.println(msg);
12     }
13
14 }
```

### 5.1.3.3 AmqpAdmin

我们也可以通过代码动态创建交换器、队列、绑定规则，AmqpAdmin 为我们提供了操作方法。

```
1  /**
2   * 消息队列测试
3   */
4  @RunWith(SpringRunner.class)
5  @SpringBootTest
6  public class AmqpAdminTest {
7
8      @Autowired
9      AmqpAdmin amqpAdmin;
10
11     @Test
12     public void testExchange(){
13         amqpAdmin.declareExchange(new DirectExchange("admin.exchange"));
14     }
15
16     @Test
17     public void testCreateQueue(){
18         amqpAdmin.declareQueue(new Queue("admin.queue"));
19     }
20 }
```

```

19      }
20
21      @Test
22      public void testCreateQueue(){
23          amqpAdmin.declareBinding(new Binding("admin.queue",
24              Binding.DestinationType.QUEUE, "admin.exchange", "admin.test", null));
25      }

```

## 5.2 搜索服务

Elasticsearch是一个分布式搜索服务，提供Restful API，底层基于Lucene，采用多shard（分片）的方式保证数据安全，并且提供自动resharding的功能，github等大型的站点也是采用了ElasticSearch作为其搜索服务。

### 5.2.1 安装Elasticsearch

注意： Spring Boot 2.3.X 对应的 ES 版是 7.6.2

Spring Data Release Train	Spring Data Elasticsearch	Elasticsearch	Spring Boot
Neumann[1]	4.0.x[1]	7.6.2	2.3.x[1]
Moore	3.2.x	6.8.6	2.2.x
Lovelace	3.1.x	6.2.2	2.1.x
Kay[2]	3.0.x[2]	5.5.0	2.0.x[2]
Ingalls[2]	2.1.x[2]	2.4.0	1.5.x[2]

拉取镜像：

```
1 docker pull elasticsearch:7.6.2
```

创建目录：

```

1 mkdir -p /data/es/config
2 mkdir -p /data/es/data
3 mkdir -p /data/es/plugins
4 chmod -R 777 /data/es

```

添加配置文件: `/data/es/config/elasticsearch.yml`

```
1 http.host: 0.0.0.0
```

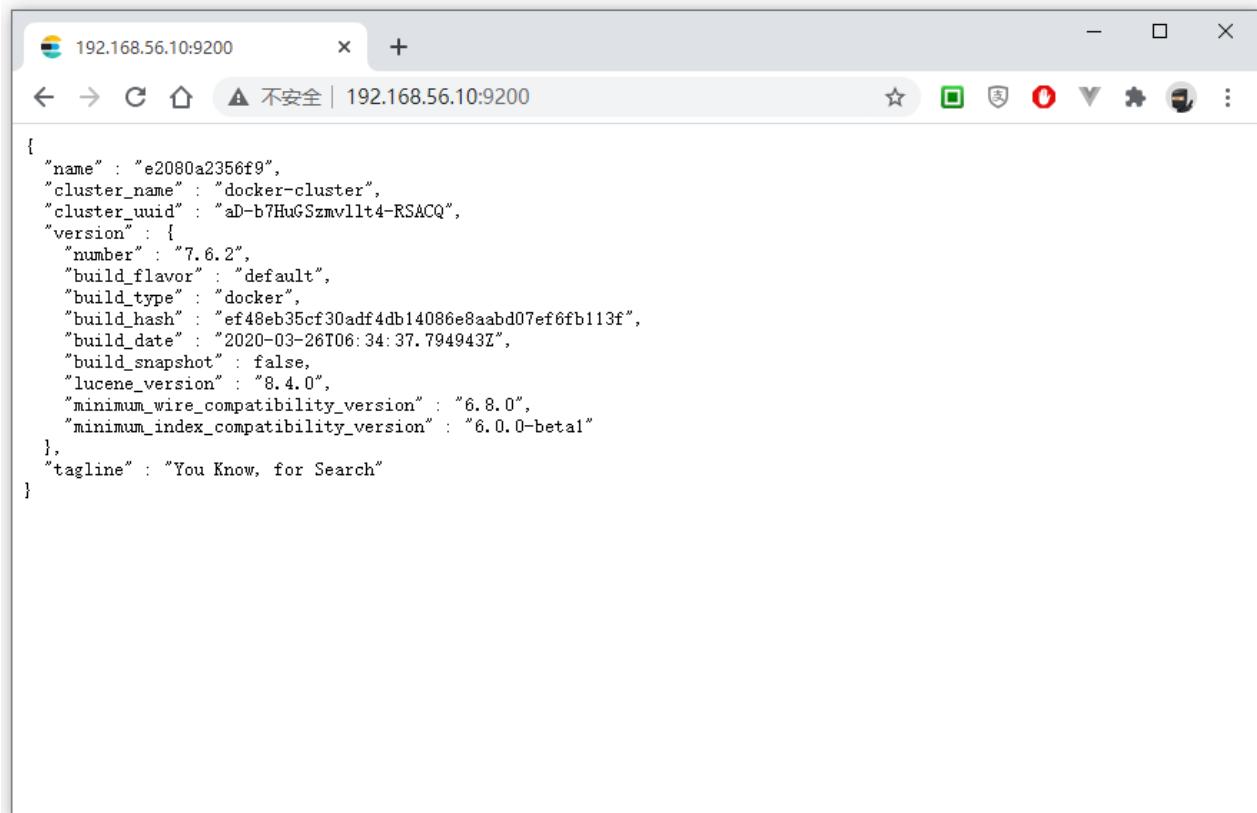
运行容器:

```
1 docker run -d --name es \
2     -p 9200:9200 \
3     -p 9300:9300 \
4     -v
5     /data/es/config/elasticsearch.yml:/usr/share/elasticsearch/config/elasticsearch.yml \
6
7     -v /data/es/data:/usr/share/elasticsearch/data \
8     -v /data/es/plugins:/usr/share/elasticsearch/plugins \
9     -e "discovery.type=single-node" \
10    -e ES_JAVA_OPTS="-Xms256m -Xmx256m" \
11    elasticsearch:7.6.2
```

其中 `-e ES_JAVA_OPTS="-Xms256m -Xmx256m"` 是为了设置初始堆内存和最大堆内存, 否则, ES默认会占用2G堆内存, 导致启动失败。

访问测试:

访问地址 `http://192.168.56.100:9200` 可以看到:



HTTP接口测试：

```
1  ###
2  # 测试
3  GET http://192.168.56.100:9200/
4  Content-Type: application/json
5
```

## 5.2.2 图形化工具

拉取镜像：

```
1 docker pull kibana:7.6.2
```

运行容器：

```
1 docker run -d --name kibana \
2   -p 5601:5601 \
3   -e ELASTICSEARCH_HOSTS=http://172.17.0.1:9200 \
4   kibana:7.6.2
```

访问页面：<http://192.168.56.100:5601>

## 5.2.3 接口调用

官方文档：<https://www.elastic.co/guide/cn/elasticsearch/guide/current/index.html>

以 员工文档 的形式存储为例：

一个文档代表一个员工数据。存储数据到 ElasticSearch 的行为叫做 索引，但在索引一个文档之前，需要确定将文档存储在哪里。

一个 ElasticSearch 集群可以 包含多个 索引，相应的每个索引可以包含多个 类型 。这些不同的类型存储着多个 文档，每个文档又有 多个 属性 。

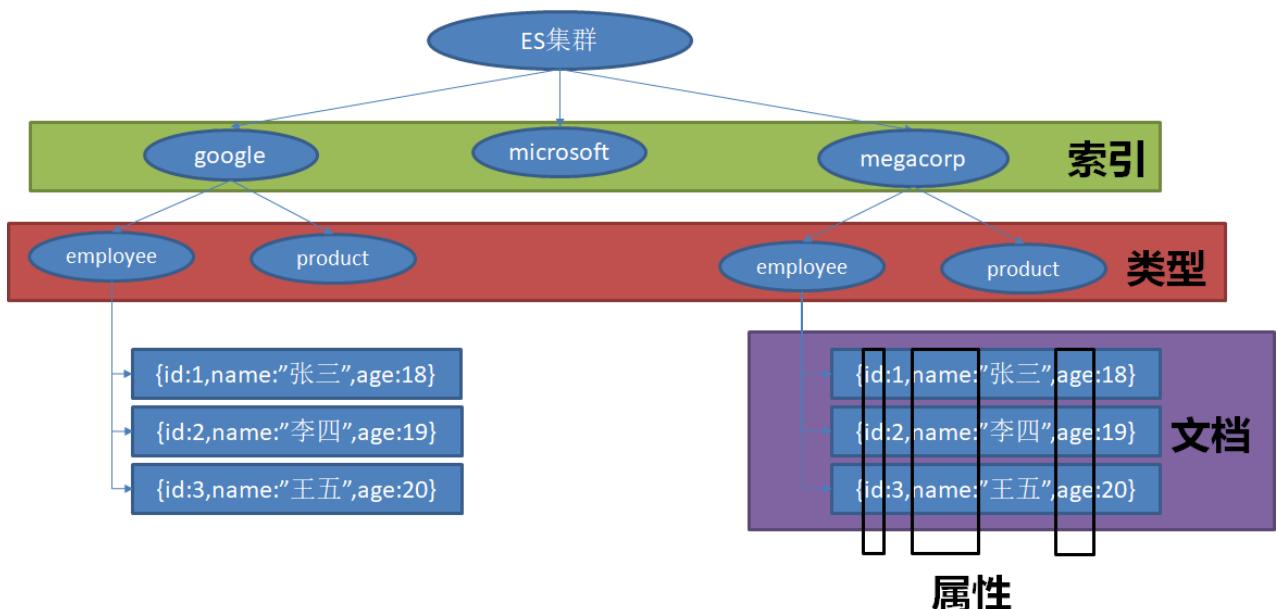
类似关系：

- 索引-数据库
- 类型-表
- 文档-表中的记录
- 属性-列

注意：在ES 7.0 版本中，已经移除了 **类型** 的概念，参考资料：<https://www.cnblogs.com/miracle-luna/p/10998670.html>

移除 **type** 之后：

- 索引操作：PUT {index}/{type}/{id} 需要改成 PUT {index}/\_doc/{id}
- Mapping 操作：PUT {index}/{type}/\_mapping 则变成 PUT {index}/\_mapping
- 所有增删改查操作返回结果里面的关键字 **\_type** 都将被移除
- 父子关系使用 **join** 字段来构建



示例：

可以使用 **PostMan** 发起请求

保存员工信息：

```
1  PUT /megacorp/employee/1
2  {
3      "first_name" : "John",
4      "last_name" : "Smith",
5      "age" : 25,
6      "about" : "I love to go rock climbing",
7      "interests": [ "sports", "music" ]
8  }
```

The screenshot shows the Postman application interface. A collection named "Save Employee" contains a single item named "PUT Save Employee". The request method is PUT, the URL is `http://192.168.56.10:9200/megacorp/employee/1`, and the body is a JSON object representing an employee:

```
1 {
2     "first_name": "John",
3     "last_name": "Smith",
4     "age": 25,
5     "about": "I love to go rock climbing",
6     "interests": [ "sports", "music" ]
7 }
```

The response status is 201 Created, and the response body is a JSON document indicating the creation of a new document:

```
1 {
2     "_index": "megacorp",
3     "_type": "employee",
4     "_id": "1",
5     "_version": 1,
6     "result": "created",
7     "_shards": {
8         "total": 2,
9         "successful": 1,
10        "failed": 0
11    },
12    "created": true
13 }
```

继续参照官网文档，添加两个员工信息

息：[https://www.elastic.co/guide/cn/elasticsearch/guide/current/\\_indexing\\_employee\\_documents.html](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_indexing_employee_documents.html)

根据ID查询信息：

```
1 GET /megacorp/employee/1
```

The screenshot shows the Postman application interface. At the top, there are tabs for 'Save Employee' (PUT) and 'Get Employee' (GET). The 'Get Employee' tab is selected. Below it, the URL is set to 'http://192.168.56.10:9202/megacorp/employee/1'. The 'Params' tab is active, showing a single query parameter 'Key' with value 'Value'. The 'Body' tab shows the JSON response:

```

1 {
2   "_index": "megacorp",
3   "_type": "employee",
4   "_id": "1",
5   "_version": 1,
6   "found": true,
7   "_source": {
8     "first_name": "John",
9     "last_name": "Smith",
10    "age": 25,
11    "about": "I love to go rock climbing",
12    "interests": [
13      "sports",
14      "music"
15    ]
16  }
17 }

```

The status bar at the bottom indicates 'Status: 200 OK'.

查询所有：

```
1 GET /megacorp/employee/_search
```

按条件查询：

```
1 GET /megacorp/employee/_search?q=last_name:Smith
```

## 5.2.4 整合检索

添加Maven依赖：

```

1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-web</artifactId>
5   </dependency>
6
7   <dependency>
8     <groupId>org.elasticsearch.client</groupId>
9     <artifactId>elasticsearch-rest-high-level-client</artifactId>
10  </dependency>
11
12  <dependency>
13    <groupId>org.springframework.boot</groupId>

```

```
14      <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
15    </dependency>
16
17    <dependency>
18      <groupId>org.springframework.boot</groupId>
19      <artifactId>spring-boot-starter-test</artifactId>
20    </dependency>
21  </dependencies>
```

#### 5.2.4.1 RestClient访问

添加配置文件：

```
1  spring:
2    elasticsearch:
3      rest:
4        uris: http://192.168.56.10:9200
```

创建实体类：

```
1  /**
2   * 实体类
3   * @author LetCode
4   * @since 1.0
5   */
6  public class Article {
7
8    private Integer id;
9
10   private String title;
11
12   private String summary;
13
14   private String content;
15
16   public Article() {
17   }
18
19   public Integer getId() {
20     return id;
21   }
22
23   public void setId(Integer id) {
24     this.id = id;
25   }
26
27   public String getTitle() {
28     return title;
29   }
```

```
30
31     public void setTitle(String title) {
32         this.title = title;
33     }
34
35     public String getSummary() {
36         return summary;
37     }
38
39     public void setSummary(String summary) {
40         this.summary = summary;
41     }
42
43     public String getContent() {
44         return content;
45     }
46
47     public void setContent(String content) {
48         this.content = content;
49     }
50 }
```

编写测试类：

```
1  /**
2   * 高版本ES测试
3   * @author LetCode
4   * @since 1.0
5   */
6 @RunWith(SpringRunner.class)
7 @SpringBootTest
8 public class ElasticSearchRestClientTest {
9
10    @Autowired
11    RestHighLevelClient restHighLevelClient;
12
13    /**
14     * 测试保存
15     */
16    @Test
17    public void testSave(){
18        Article article = new Article();
19        article.setId(3);
20        article.setTitle("测试文章3");
21        article.setSummary("这是测试文章，内容很短3");
22        article.setContent("说过了，内容很短！3");
23
24        try {
25            IndexRequest indexRequest = new IndexRequest("letcode");
26            indexRequest.id("3");
```

```
27         indexRequest.source(JsonUtils.toJSONString(article), XContentType.JSON);
28         restHighLevelClient.index(indexRequest, RequestOptions.DEFAULT);
29     } catch (IOException e) {
30         e.printStackTrace();
31     }
32 }
33 */
34 /**
35 * 按ID查找
36 */
37 @Test
38 public void testSearchById(){
39     try {
40         GetRequest getRequest = new GetRequest("etcode", "3");
41         GetResponse response = restHighLevelClient.get(getRequest,
42             RequestOptions.DEFAULT);
43         System.out.println(response.getSource());
44     } catch (IOException e) {
45         e.printStackTrace();
46     }
47 }
48 */
49 /**
50 * 搜索表达式
51 */
52 @Test
53 public void testSearchQuery(){
54     try {
55         BoolQueryBuilder boolBuilder = QueryBuilders.boolQuery();
56
57         // 这里可以根据字段进行搜索, must表示符合条件的, 相反的mustnot表示不符合条件的
58         // matchQuery: 等值查询, wildcardQuery: 模糊查询
59         boolBuilder.must(QueryBuilders.wildcardQuery("title", "*测试*"));
60
61         SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
62         sourceBuilder.query(boolBuilder);
63         sourceBuilder.from(0);
64         sourceBuilder.size(100); // 获取记录数, 默认10
65
66         // 第一个是获取字段, 第二个是过滤的字段, 默认获取全部
67         sourceBuilder.fetchSource(new String[] { "id", "title"}, new String[]
68     });
69
70     SearchRequest searchRequest = new SearchRequest("etcode");
71     searchRequest.source(sourceBuilder);
72     SearchResponse response = restHighLevelClient.search(searchRequest,
73         RequestOptions.DEFAULT);
74     System.out.println("search: " + JsonUtils.toJSONString(response));
75
76     SearchHits hits = response.getHits();
77     SearchHit[] searchHits = hits.getHits();
```

```
76             for (SearchHit hit : searchHits) {
77                 System.out.println("search -> " + hit.getSourceAsString());
78             }
79         } catch (IOException e) {
80             e.printStackTrace();
81         }
82     }
83 }
```

#### 5.2.4.2 Spring Data访问

官方文档: <https://docs.spring.io/spring-data/elasticsearch/docs/4.0.3.RELEASE/reference/html/#preface>

添加配置文件:

```
1 spring:
2   data:
3     elasticsearch:
4       client:
5         reactive:
6           endpoints: [http://192.168.56.10:9200]
```

创建实体类:

```
1 /**
2  * 新闻实体类
3  * @author LetCode
4  * @since 1.0
5 */
6 @Document(indexName = "letcode")
7 public class News {
8
9     @Id
10    private Integer id;
11
12    @Field(type = FieldType.Keyword)
13    private String title;
14
15    @Field(type = FieldType.Keyword)
16    private String summary;
17
18    @Field(type = FieldType.Keyword)
19    private String content;
20
21    public News() {
22    }
```

```
23
24     public Integer getId() {
25         return id;
26     }
27
28     public void setId(Integer id) {
29         this.id = id;
30     }
31
32     public String getTitle() {
33         return title;
34     }
35
36     public void setTitle(String title) {
37         this.title = title;
38     }
39
40     public String getSummary() {
41         return summary;
42     }
43
44     public void setSummary(String summary) {
45         this.summary = summary;
46     }
47
48     public String getContent() {
49         return content;
50     }
51
52     public void setContent(String content) {
53         this.content = content;
54     }
55
56     @Override
57     public String toString() {
58         return "News{" +
59                 "id=" + id +
60                 ", title='" + title + '\'' +
61                 ", summary='" + summary + '\'' +
62                 ", content='" + content + '\'' +
63                 '}';
64     }
65 }
```

## 添加 Repository:

```
1  public interface NewsRepository extends ElasticsearchRepository<News, Integer> {
2 }
```

添加测试类：

```
1  /**
2   * 通过Spring Data访问ElasticSearch
3   * @author LetCode
4   * @since 1.0
5   */
6  @RunWith(SpringRunner.class)
7  @SpringBootTest
8  public class SpringDataTest {
9
10     @Autowired
11     NewsRepository newsRepository;
12
13     @Test
14     public void testSave(){
15         News news = new News();
16         news.setId(11);
17         news.setTitle("这是一个大新闻");
18         news.setSummary("其实也没啥!");
19         news.setContent("这里是新闻的内容！！！");
20
21         newsRepository.save(news);
22     }
23
24     @Test
25     public void testQuery(){
26         News news = newsRepository.findById(11).get();
27         System.out.println(news);
28     }
29
30 }
```

## 六. 阶段练习

基于 `VueJS` + `Axios` + `SpringBoot` + `MySQL` + `Redis` 实现简单的博客系统。

功能要求：

1. 前后端分离开发；
2. 实现用户注册、登录功能，使用 `Token` 标记用户登录状态；
3. 实现文章发表、修改、删除、列表展示、详情展示；
4. 实现热门文章功能，并使用 `Redis` 进行缓存；
5. 实现文章阅读计数功能，使用 `RabbitMQ` 进行异步操作；

6. 实现文章搜索功能，使用 **Elasticsearch** 完成检索。