

项目课程系列之 Atcrowdfunding

尚硅谷 JavaEE 教研组

版本：V1.0

第一章 Spring Boot 介绍

1.1 概述

Spring Boot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。习惯优于配置

1.2 为什么使用 Spring Boot

J2EE 笨重的开发、繁多的配置、低下的开发效率、复杂的部署流程、第三方技术集成难度大。

1.3 Spring Boot 是什么

一站式整合所有应用框架的框架；并且完美整合 Spring 技术栈：<https://spring.io/projects>

Spring Boot 来简化 Spring 应用开发，约定大于配置，去繁从简，just run 就能创建一个独立的，产品级别的应用

1.4 Spring boot 优点

- 快速创建独立运行的 Spring 项目以及与主流框架集成
- 使用嵌入式的 Servlet 容器，应用无需打成 WAR 包
- starters 自动依赖与版本控制
- 大量的自动配置，简化开发，也可修改默认值
- 无需配置 XML，无代码生成，开箱即用
- 准生产环境的运行时应用监控
- 与云计算的天然集成

1.5 环境要求

<https://docs.spring.io/spring-boot/docs/2.0.7.RELEASE/reference/htmlsingle/#getting-started-system-requirements>

<https://docs.spring.io/spring-boot/docs/2.0.7.RELEASE/reference/htmlsingle/#common-application-properties>

9. System Requirements

Spring Boot 2.0.7.RELEASE requires Java 8 or 9 and Spring Framework 5.0.11.RELEASE or above

Explicit build support is provided for the following build tools:

| Build Tool | Version |
|------------|---------|
| Maven | 3.2+ |
| Gradle | 4.x |

9.1 Servlet Containers

Spring Boot supports the following embedded servlet containers:

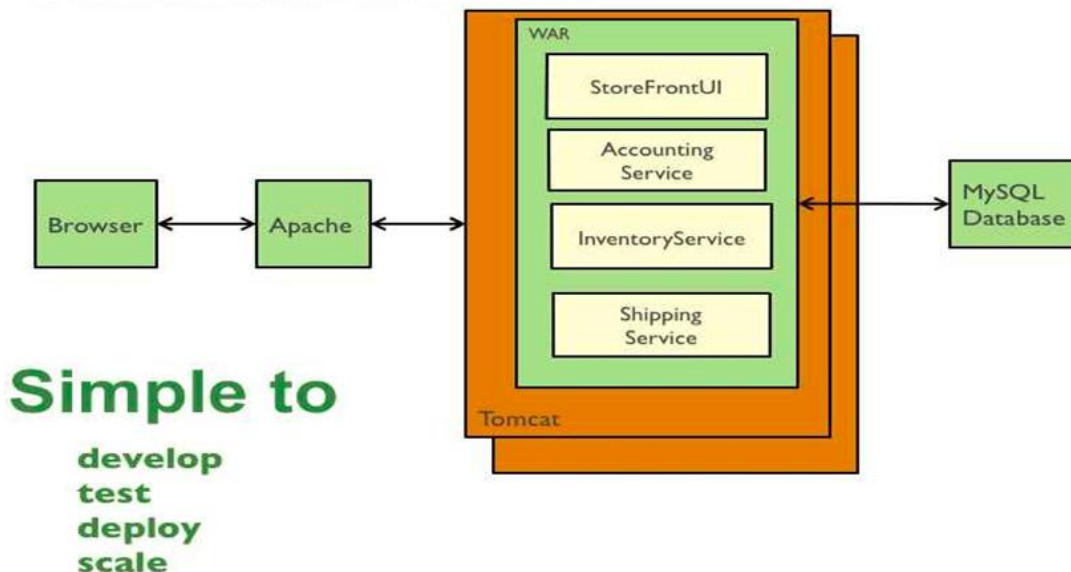
| Name | Servlet Version |
|--------------|-----------------|
| Tomcat 8.5 | 3.1 |
| Jetty 9.4 | 3.1 |
| Undertow 1.4 | 3.1 |

You can also deploy Spring Boot applications to any Servlet 3.1+ compatible container.

第二章 项目架构-单体应用

2.1 单体应用

Traditional web application architecture



2.2 单体应用（monolith application）

- 就是将应用程序的所有功能都打包成一个独立的单元，可以是 JAR、WAR、EAR 或其它归档格式。
- Mvn 命令: `package -Dmaven.test.skip=true`

2.3 单体应用有如下优点

- 为人所熟知：现有的大部分工具、应用服务器、框架和脚本都是这种应用程序；
- IDE 友好：像 NetBeans、Eclipse、IntelliJ 这些开发环境都是针对开发、部署、调试这样的单个应用而设计的；
- 便于共享：单个归档文件包含所有功能，便于在团队之间以及不同的部署阶段之间共享；
- 易于测试：单体应用一旦部署，所有的服务或特性就都可以使用了，这简化了测试过程，因为没有额外的依赖，每项测试都可以在部署完成后立刻开始；
- 容易部署：只需将单个归档文件复制到单个目录下。

2.4 单体应用的一些不足

- **不够灵活**：对应用程序做任何细微的修改都需要将整个应用程序重新构建、重新部署。开发人员需要等到整个应用程序部署完成后才能看到变化。如果多个开发人员共同开发一个应用程序，那么还要等待其他开发人员完成了各自的开发。这降低了团队的灵活性和功能交付频率；
- **妨碍持续交付**：单体应用可能会比较大，构建和部署时间也相应地比较长，不利于频繁部署，阻碍持续交付。在移动应用开发中，这个问题会显得尤为严重；
- **受技术栈限制**：对于这类应用，技术是在开发之前经过慎重评估后选定的，每个团队成员都必须使用相同的开发语言、持久化存储及消息系统，而且要使用类似的工具，无法根据具体的场景做出其它选择；
- **技术债务**：“不坏不修（Not broken, don't fix）”，这在软件开发中非常常见，单体应用尤其如此。系统设计或写好的代码难以修改，因为应用程序的其它部分可能会以意料之外的方式使用它。随着时间推移、人员更迭，这必然会增加应用程序的技术债务。

第三章 项目架构-微服务

3.1 微服务

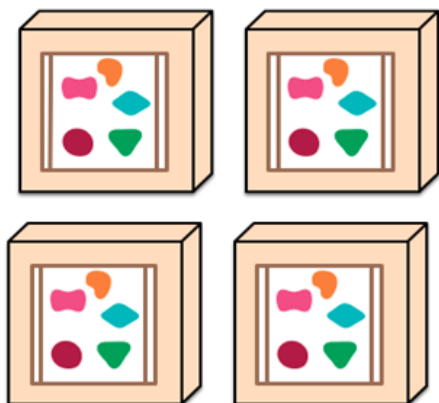
一个单体应用程序把它所有的功能放在一个单一进程中...



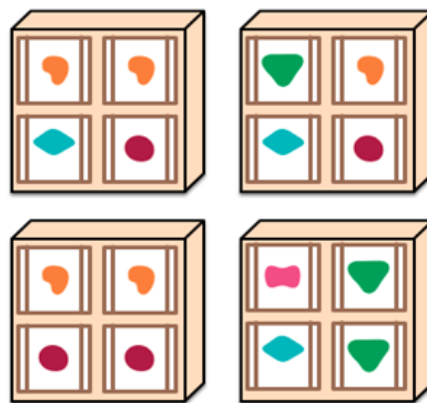
一个微服务架构把每个功能元素放进一个独立的服务中...



...并且通过在多个服务器上复制这个单体进行扩展



...并且通过跨服务器分发这些服务进行扩展，只在需要时才复制。



3.2 微服务的样子

<https://www.martinfowler.com/articles/microservices.html> 微服务 microservices

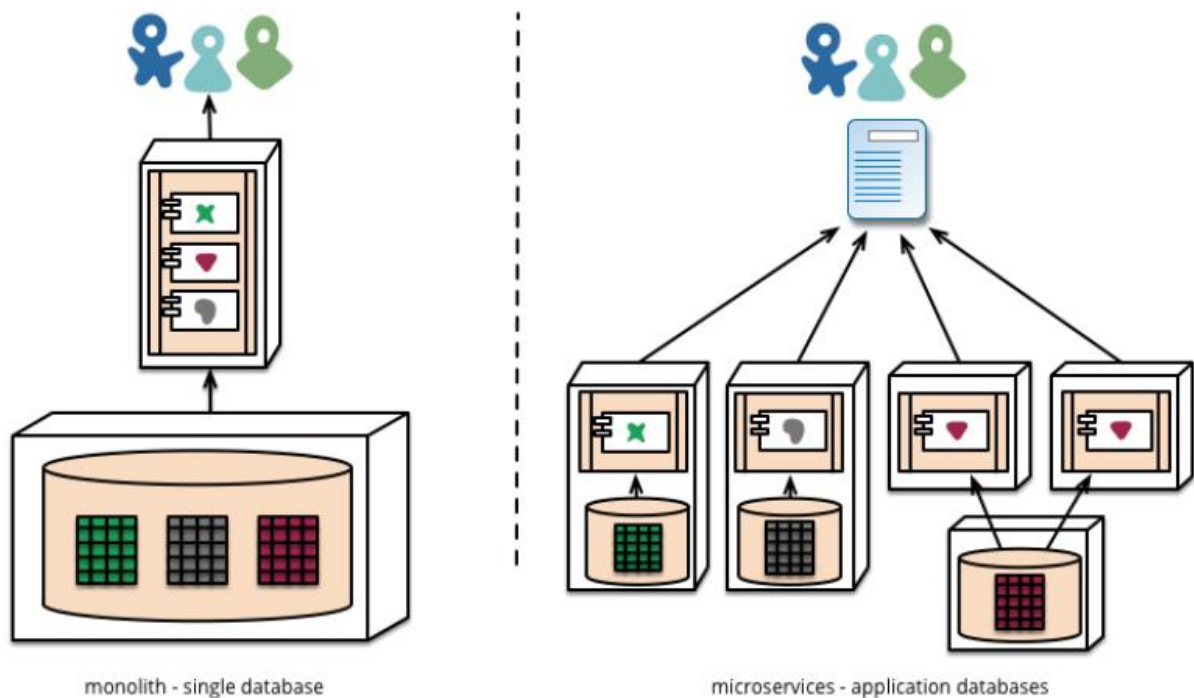
<http://blog.cuicc.com/blog/2015/07/22/microservices/>

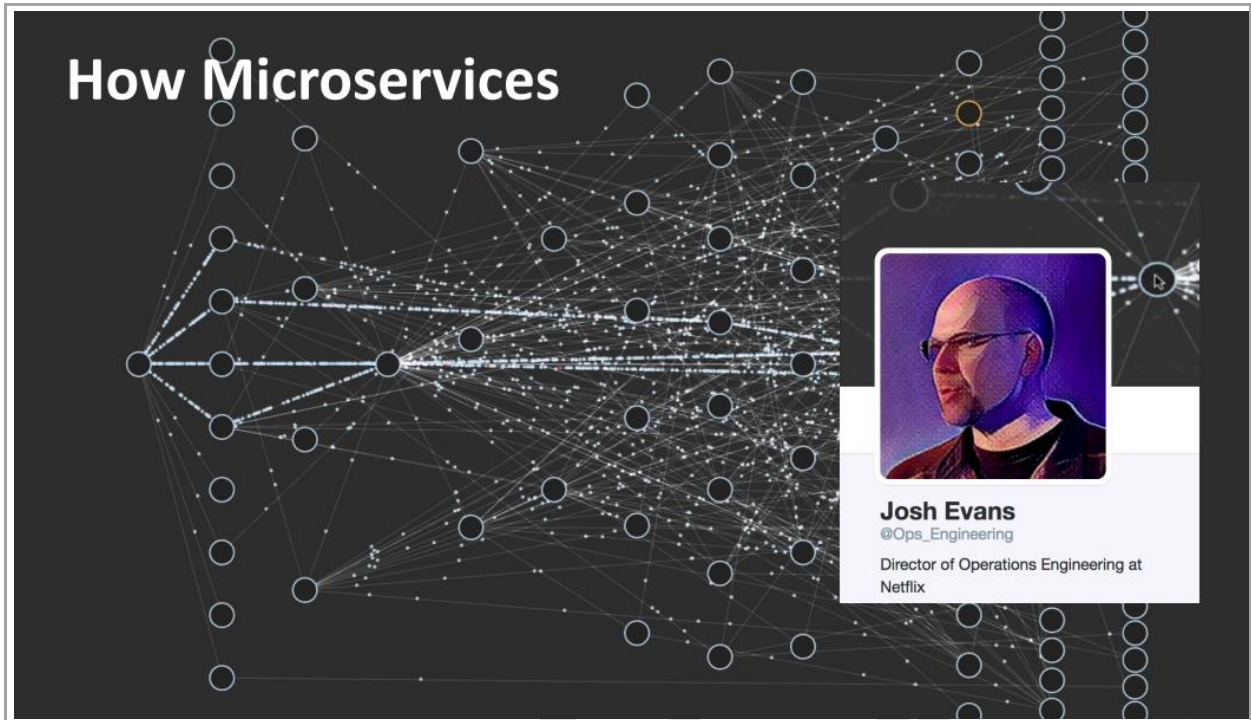
【RPC（远程过程调用）】：

1、dubbo【传输数据没用 HTTP 协议；dubbo 协议组装数据】/zookeeper

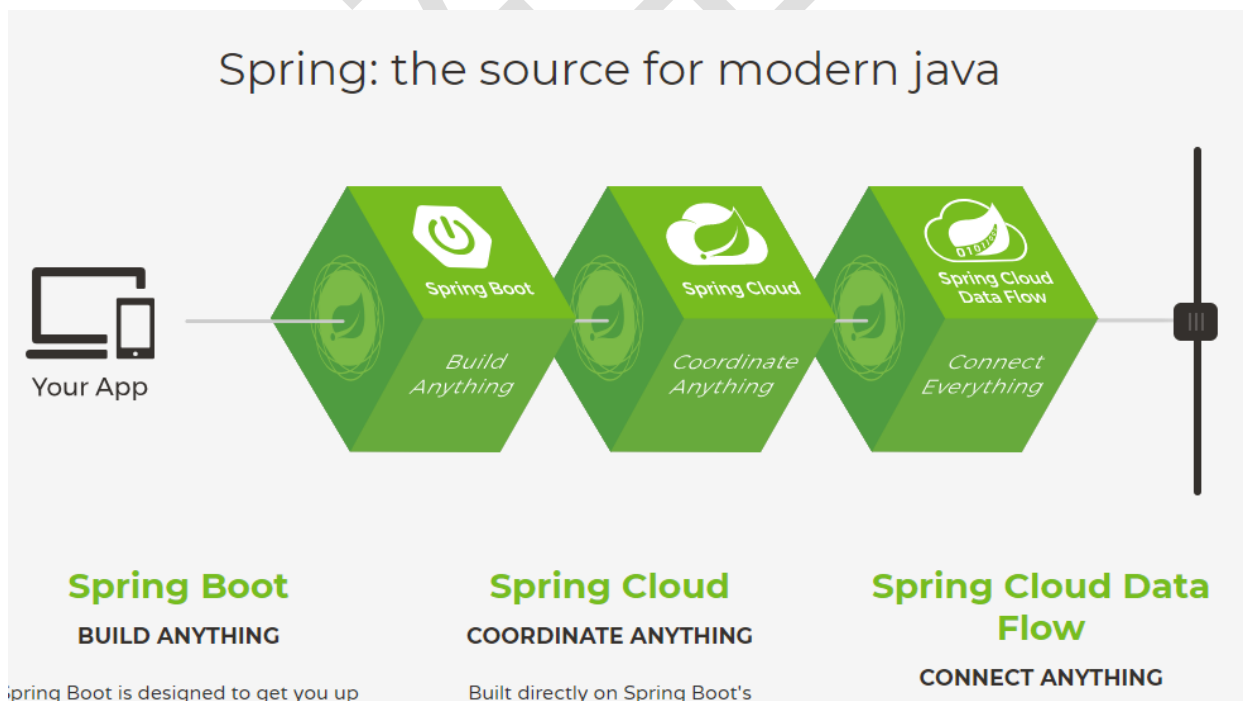
2、SpringCloud: HTTP+JSON

In short(简言之), the microservice **architectural style**【架构风格】[1] is an approach to developing a **single application as a suite of small services**【独立应用变成一套小服务】, each running in its own process and communicating with lightweight(轻量级沟通) mechanisms(每一个都运行在自己的进程内(容器)), often an HTTP resource API(用 HTTP, 将功能写成能接受请求). These services are built around business capabilities (独立业务能力) and independently deployable by fully automated deployment machinery (应该自动化独立部署). There is a bare minimum of centralized management of these services (应该有一个能管理这些服务的中心), which may be written in different programming languages (独立开发语言) and use different data storage technologies (独立的数据存储)





3.3 Spring 官网



第四章 SpringBoot-HelloWorld 初体验

如何完成 页面发送/hello 请求，服务器响应 "OK" 字符串；

4.1 创建 maven 工程

4.2 引入如下依赖

```
<!-- Inherit defaults from Spring Boot -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.3.RELEASE</version>
</parent>

<!-- Add typical dependencies for a web application -->
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

4.3 增加控制器

```
@Controller
public class HelloController {
    @ResponseBody
    @GetMapping("/hello")
    public String handle01(){
        return "OK!+哈哈";
    }
}
```

4.4 编写主程序

```
@SpringBootApplication
```



```
public class MainApplication {  
    public static void main(String[] args) {  
        //Spring 应用跑起来...  
        SpringApplication.run(MainApplication.class, args);  
    }  
}
```

4.5 运行访问

<http://localhost:8080/hello>

第五章 SpringBoot-原理-简化依赖和配置

5.1 依赖管理

```
<project xmlns="http://maven.apache.org/POM/4.0.0"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
    http://maven.apache.org/xsd/maven-4.0.0.xsd">  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>com.atguigu</groupId>  
  <artifactId>spring-boot-01</artifactId>  
  <version>0.0.1-SNAPSHOT</version>  
  
  <!-- 用来做依赖管理，几乎将我们用到的所有的依赖的版本都声明好了；版本仲裁中心 -->  
  <parent>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-parent</artifactId>  
    <version>2.0.7.RELEASE</version>  
  </parent>  
  
  <!--  
    spring-boot-starter-xxx: 场景启动器;  
    1)、我们现在开发 web 程序就引用 web 场景 spring-boot-starter-web;  
    SpringBoot 自定引入这个场景所需要的所有依赖;  
    2)、所有支持的场合都在这里
```



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

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

<https://docs.spring.io/spring-boot/docs/2.0.7.RELEASE/reference/htmlsingle/>

III. Using Spring Boot

13. Build Systems

13.1. Dependency Management

13.2. Maven

- 13.2.1. Inheriting the Starter Parent
- 13.2.2. Using Spring Boot without the Parent POM
- 13.2.3. Using the Spring Boot Maven Plugin

13.3. Gradle

13.4. Ant

13.5. Starters

第六章 SpringBoot-原理-简化部署

6.1 引入 springboot 插件

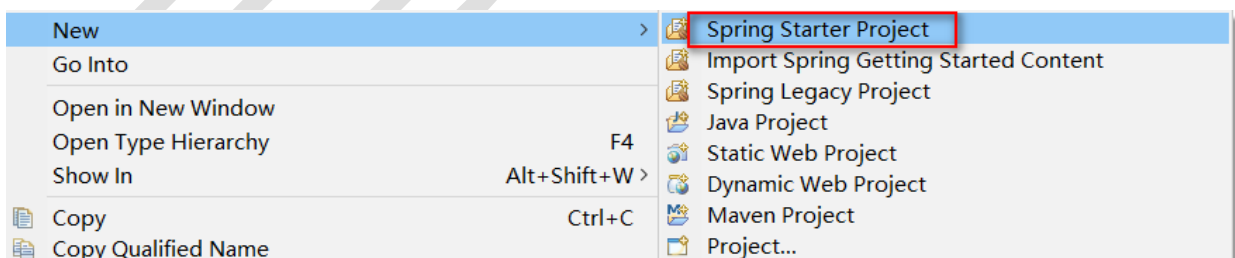
```
<!-- 引入 springboot 插件；打包插件 -->
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

6.2 打包：package

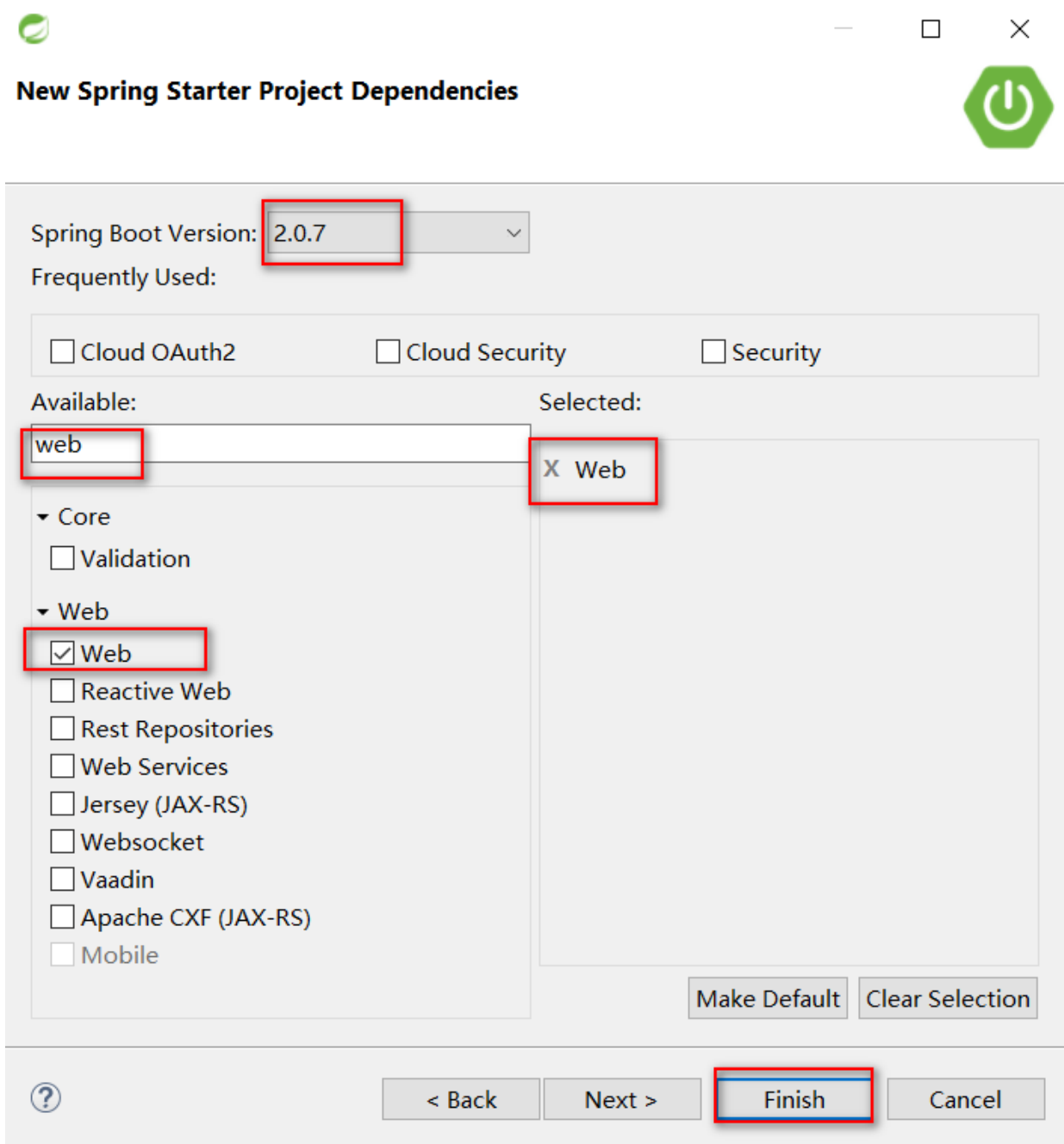
6.3 java -jar

第七章 快速创建 SpringBoot 应用

7.1 创建 Spring Starter Project; 必须联网创建



7.2 选择版本，引入需要的依赖



The image shows the 'New Spring Starter Project Dependencies' dialog box. It has a title bar with a green icon, a minus button, a maximize button, and a close button. The title is 'New Spring Starter Project Dependencies' and there is a green power button icon on the right. The main area is divided into two sections: 'Available:' and 'Selected:'. In the 'Available:' section, there is a search bar containing 'web'. Below it, there are two expandable sections: 'Core' and 'Web'. The 'Web' section is expanded, and the 'Web' checkbox is checked. In the 'Selected:' section, there is a list containing 'X Web'. At the bottom of the 'Available:' section, there are buttons for 'Make Default' and 'Clear Selection'. At the bottom of the dialog, there are buttons for '< Back', 'Next >', 'Finish', and 'Cancel'. The 'Finish' button is highlighted with a red box.

Spring Boot Version: 2.0.7

Frequently Used:

☐ Cloud OAuth2 ☐ Cloud Security ☐ Security

Available: Selected:

web X Web

▼ Core

☐ Validation

▼ Web

☒ Web

☐ Reactive Web

☐ Rest Repositories

☐ Web Services

☐ Jersey (JAX-RS)

☐ Websocket

☐ Vaadin

☐ Apache CXF (JAX-RS)

☐ Mobile

Make Default Clear Selection

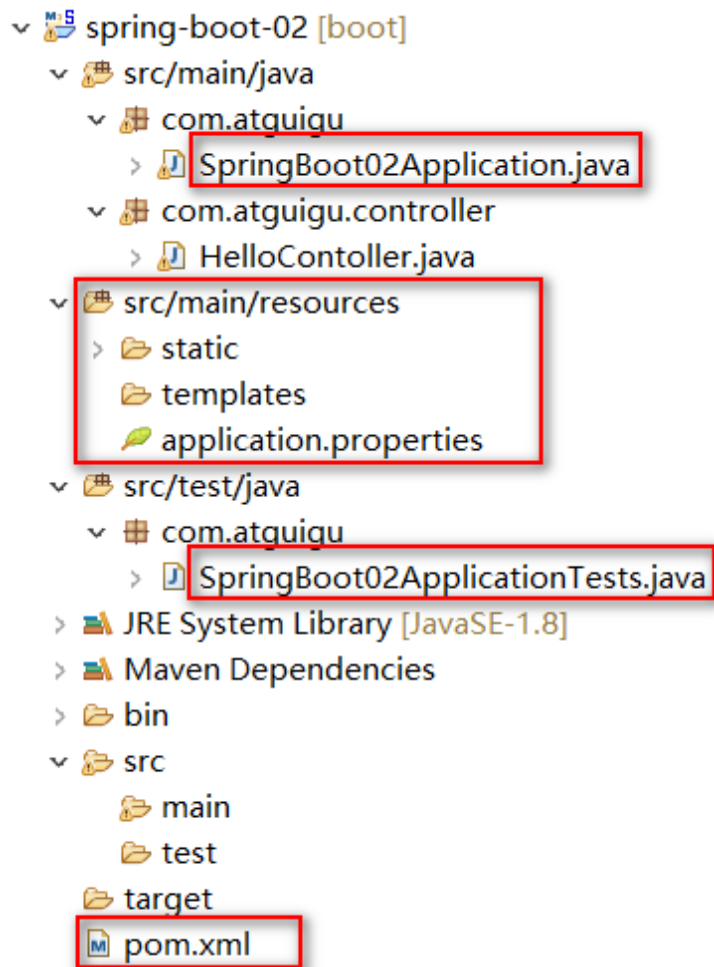
? < Back Next > Finish Cancel

7.3 项目结构

- 自动生成主程序类，用于启动项目
- 自动生成静态资源目录及属性配置文件
- 自动生成测试类

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

- 自动增加 pom.xml 相关依赖配置



7.4 增加控制器类

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;
```

```
//@ResponseBody
```

```
//@Controller
```

```
@RestController
```

```
public class HelloController {
```

```
    @GetMapping("/hello")
```

```
    public String handle01(){
```

```
    return "OK!+哈哈";  
}  
  
}
```

7.5 运行测试

- <http://localhost:8080/hello>
- 在 static 文件下存放 java.jpg 图片， <http://localhost:8080/java.jpg>
- 在 application.properties 设置
- 端口， server.port=8081， <http://localhost:8081/hello>
- 上下文路径(一般不指定)， server.servlet.context-path=/a ， <http://localhost:8081/a/hello>
- server.servlet.session.timeout=60
- server.tomcat.max-threads=800
- server.tomcat.uri-encoding=UTF-8

第八章 SpringBoot-yml 配置文件

8.1 配置文件

- SpringBoot 使用一个全局的配置文件，配置文件名是固定的；
- **application.properties**
- **application.yml**
 - 配置文件的作用：修改 SpringBoot 自动配置的默认值；SpringBoot 在底层都给我们自动配置好；
 - YAML（YAML Ain't Markup Language）
 - YAML A Markup Language：是一个标记语言
 - YAML isn't Markup Language：不是一个标记语言；
- 标记语言：
 - 以前的配置文件：大多都使用的是 xxxx.xml 文件；
 - YAML：以数据为中心，比 json、xml 等更适合做配置文件；
 - YAML：配置例子

```
server:  
  port: 8081
```

- XML:

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

8.2 YAML 语法

8.2.1 YAML 基本语法

- 使用缩进表示层级关系
- 缩进时不允许使用 Tab 键，只允许使用空格。
- 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可
大小写敏感

8.2.2 YAML 支持的三种数据结构

- 对象：键值对的集合
- 数组：一组按次序排列的值
- 字面量：单个的、不可再分的值

8.3 值的写法

8.3.1 字面量：普通的值（数字，字符串，布尔）

- k: v: 字面直接来写；
- 字符串默认不用加上单引号或者双引号；
- "": 双引号；不会转义字符串里面的特殊字符；特殊字符会作为本身想表示的意思
- name: "zhangsan \n lisi": 输出；zhangsan 换行 lisi
- ": 单引号；会转义特殊字符，特殊字符最终只是一个普通的字符串数据
- name: 'zhangsan \n lisi': 输出；zhangsan \n lisi

8.3.2 对象、Map（属性和值）（键值对）：

- k: v: 在下一行来写对象的属性和值的关系；注意缩进
- 对象还是 k: v 的方式

```
friends:
  lastName: zhangsan
```

```
age: 20
```

- 行内写法:
- friends: {lastName: zhangsan,age: 18}

8.3.3 数组 (List、Set)

用- 值表示数组中的一个元素

```
pets:
```

- cat
- dog
- pig

行内写法

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

第九章 SpringBoot-自动配置原理

9.1 自动配置原理

- SpringBoot 所有的东西都自动配置好了;
 - 1)、**spring-boot-autoconfigure-2.0.7.RELEASE.jar** (自动配置包)
 - 2)、好多的场景全部自动配置好
- 自动配置原理:
 - 1)、主程序类标注了 **@SpringBootApplication** 注解相当于标注了 **@EnableAutoConfiguration**
 - 2)、**@EnableAutoConfiguration** 开启 SpringBoot 的自动配置功能
- 就会自动的将所有的自动配置类导进来
如: **HttpEncodingAutoConfiguration** (http 编码的自动配置)
 - 1)、**@ConditionalOnXX** 根据当前系统环境判断我这个类的所有配置是否需要生效
 - 2)、会发现这些配置类中使用 **@Bean** 给容器中放了好多组件, 这些组件就生效;
 - 3)、这些组件会从一个类中 (配置文件属性值的封装类) 获取到它应该使用的值是什么。比如 **HttpEncodingProperties** 获取 charset
 - 4)、这写配置文件值的封装类都是和配置文件一一绑定
@ConfigurationProperties(prefix = "spring.http.encoding")
HttpEncodingProperties
- 使用心得:
 - 1)、SpringBoot 帮我们配好了所有的场景
 - 2)、SpringBoot 中会有很多的 **xxxxAutoConfigurion** (帮我们给容器中自动配好组件)
 - 3)、**xxxxAutoConfigurion** 给容器中配组件的时候, 组件默认的属性一般都是从 **xxxProperties** 中获取这些属性的值

4)、xxxProperties 是和配置文件绑定的（属性一一对应）

5)、我们改掉这些默认配置即可；

6)、如果默认的组件我们不用；

@Bean

@ConditionalOnMissingBean：容器中没这个组件

public InternalResourceViewResolver defaultViewResolver()

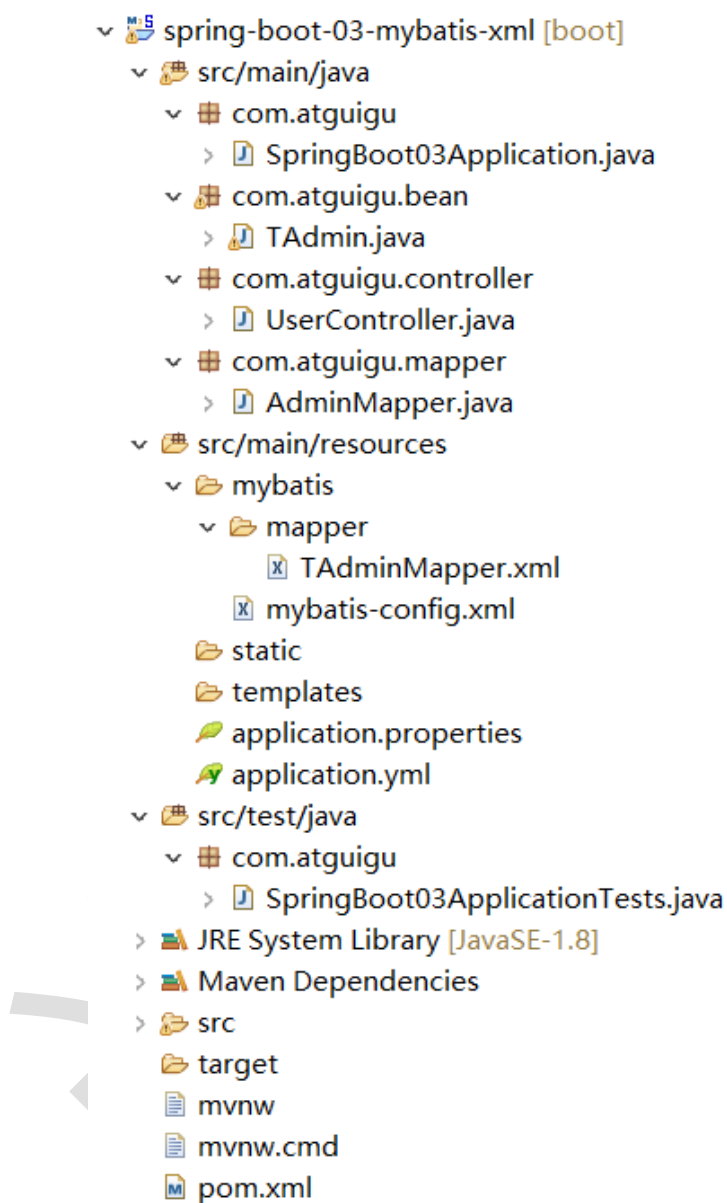
SpringBoot 的一个最大策略：自定义组件用自己的，否则，使用默认的。

9.2 自定义配置类

视图解析需要将 thymeleaf 模块引入项目中。

```
@Configuration
public class AppConfig {
    @Bean
    public InternalResourceViewResolver internalResourceViewResolver(){
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/templates/");
        resolver.setSuffix(".html");
        return resolver;
    }
}
```

第十章 SpringBoot-整合 mybatis-配置版



10.1 创建 Spring Starter Project

增加 web,jdbc,mybatis,mysql 相关组件

10.2 增加 application.yml

```
spring:
```

```
datasource:
    username: root
    password: root
    url:
jdbc:mysql://192.168.137.3:3306/atcrowdfunding?useSSL=false&useUnicode=true&characterEncoding=UTF-8
    driver-class-name: com.mysql.jdbc.Driver
mybatis:
    config-location: classpath:mybatis/mybatis-config.xml
    mapper-locations: classpath:mybatis/mapper/*.xml
```

10.3 增加实体类：TAdmin

10.4 增加 Mapper 接口

```
public interface AdminMapper {
    public TAdmin getAdminById(Integer id);
}
```

10.5 增加映射配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org/DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.atguigu.mapper.AdminMapper">
    <select id="getAdminById" resultType="com.atguigu.bean.TAdmin">
        SELECT * FROM `t_admin` WHERE id=#{id}
    </select>
</mapper>
```

10.6 增加控制器

```
package com.atguigu.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import com.atguigu.bean.TAdmin;
import com.atguigu.mapper.AdminMapper;

@RestController
public class UserController {

    @Autowired
    AdminMapper adminMapper;

    @GetMapping("/getAdminById")
    public TAdmin getAdminById (Integer id){
        return adminMapper.getAdminById(id);
    }

}
```

10.7 增加 mybatis 主配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org/DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
</configuration>
```

10.8 增加 mapper 扫描

```
/**
 * 和 Mybatis 的整合
 * 1)、在配置文件中指明 mybatis 全局配置文件和 Sql 映射文件的位置
 * 2)、扫描 mybatis 的所有 mapper 接口
 */
@MapperScan("com.atguigu.mapper")
@SpringBootApplication
public class SpringBoot03Application {
    public static void main(String[] args) {
        SpringApplication.run(SpringBoot03Application.class, args);
    }
}
```

第十一章 SpringBoot-整合 mybatis-注解版

拷贝上一个项目进行实验

增加 Mapper 接口,在方法增加相应的注解即可

```
@Mapper //加不加都行
public interface AdminMapper {
    @Select("SELECT * FROM `t_admin` WHERE id=#{id}")
    public TAdmin getAdminById(Integer id);

    @Insert("INSERT INTO t_admin(loginacct,userpswd,username,email,createtime) "
    + "VALUES(#{loginacct},#{userpswd},#{username},#{email},#{createtime})")
    public void insertAdmin(TAdmin admin);
}
```

第十二章 SpringBoot-整合 Druid 数据源

12.1 使用方式一

12.1.1 增依赖

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

12.1.2 配置数据源

```
spring:
  datasource:
    username: root
    password: root
    url:
jdbc:mysql://192.168.137.3:3306/atcrowdfunding?useSSL=false&useUnicode=true&characterEncoding=UTF
```

F-8

driver-class-name: com.mysql.jdbc.Driver

type: com.alibaba.druid.pool.DruidDataSource

12.1.3 测试

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBoot03ApplicationTests {

    @Autowired
    DataSource dataSource;

    /**
     * HikariDataSource 默认数据源，性能很高
     * DruidDataSource 使用很高，很稳定
     */
    @Test
    public void contextLoads() throws SQLException {
        System.out.println(dataSource.getClass());
        Connection connection = dataSource.getConnection();
        System.out.println(connection);
        connection.close();
    }
}
```

12.2 使用方式二

12.2.1 创建数据源

```
@Configuration
public class AppConfig {

    @ConfigurationProperties(prefix = "spring.datasource") //将数据库连接信息直接封装到数据源对象中
    @Bean
    public DataSource dataSource() throws SQLException {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setFilters("stat");//配置监控统计拦截的 filters
        return dataSource;
    }
}
```

12.3 Druid 监控使用情况

12.3.1 增加配置

```
//配置 Druid 的监控
//1、配置一个管理后台的 Servlet
@Bean
public ServletRegistrationBean statViewServlet() {
    ServletRegistrationBean bean = new ServletRegistrationBean(new StatViewServlet(), "/druid/*");
    Map<String, String> initParams = new HashMap<>();
    initParams.put("loginUsername", "admin");
    initParams.put("loginPassword", "123456");
    initParams.put("allow", ""); // 默认就是允许所有访问
    initParams.put("deny", "192.168.15.21"); // 拒绝哪个 ip 访问
    bean.setInitParameters(initParams);
    return bean;
}
//2、配置一个 web 监控的 filter
@Bean
public FilterRegistrationBean webStatFilter() {
    FilterRegistrationBean bean = new FilterRegistrationBean();
    bean.setFilter(new WebStatFilter());
    Map<String, String> initParams = new HashMap<>();
    initParams.put("exclusions", "/*.js,/*.css,/druid/*"); // 排除过滤
    bean.setInitParameters(initParams);
    bean.setUrlPatterns(Arrays.asList("/*"));
    return bean;
}
```

12.3.2 如果无法打印监控语句

可以设置 `dataSource.setFilters("stat");` 或者 `spring.datasource.filters=stat`

第十三章 SpringBoot-整合 Web 组件-注解版

之前的 Web 开发基于 Servlet 2.5 规范（在 web.xml 中配置 Servlet,Filter,Listener）。

现在基于 Servlet 3.0 规范(基于配置类的方式声明对象：@WebServlet @WebFilter @WebListener)

13.1 监听器 @WebListener

```
package com.atguigu.listener;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

@WebListener
public class HelloListener implements ServletContextListener {
    @Override
    public void contextDestroyed(ServletContextEvent arg0) {
        System.out.println("应用销毁了....HelloListener");
    }

    @Override
    public void contextInitialized(ServletContextEvent arg0) {
        System.out.println("应用启动了....HelloListener");
    }
}
```

13.2 过滤器 @WebFilter(urlPatterns="//*")

```
package com.atguigu.filter;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;

@WebFilter(urlPatterns="//*")
public class HelloFilter implements Filter {

    @Override
```

```
public void destroy() {  
  
}  
  
@Override  
public void doFilter(ServletRequest arg0, ServletResponse arg1, FilterChain arg2)  
throws IOException, ServletException {  
    System.out.println("HelloFilter.....放行之前");  
    arg2.doFilter(arg0, arg1);  
    System.out.println("HelloFilter.....放行之后");  
}  
  
@Override  
public void init(FilterConfig arg0) throws ServletException {  
  
}  
  
}
```

13.3 Servlet @WebServlet(urlPatterns="/my")

```
package com.atguigu.servlet;  
  
import java.io.IOException;  
  
import javax.servlet.ServletException;  
import javax.servlet.annotation.WebServlet;  
import javax.servlet.http.HttpServlet;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
  
@WebServlet(urlPatterns="/my")  
public class MyServlet extends HttpServlet {  
  
    @Override  
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,  
        IOException {  
        resp.getWriter().write("MyServlet do.....");  
    }  
  
}
```

13.4 扫描组件 @ServletComponentScan

@ServletComponentScan

```
@MapperScan("com.atguigu.mapper")
@SpringBootApplication
public class SpringBoot03Application {

    public static void main(String[] args) {
        SpringApplication.run(SpringBoot03Application.class, args);
    }

}
```

第十四章 SpringBoot-集成业务层事务

- 扫描 Dao 接口，需要在 AtCrowdfundingApplication 类中增加扫描注解 @MapperScan("com.atguigu.**.dao")及事务管理 @EnableTransactionManagement
- 传统的 SSM 架构中采用的是声明式事务，需要在配置文件中增加 AOP 事务配置，Spring Boot 框架中简化了这种配置，可以在 Service 接口中增加注解 @Transactional

附录 1 SpringBoot 相关模块

表 6-1 官方提供的 starter pom

| 名称 | 描述 |
|--------------------------------------|---|
| spring-boot-starter | Spring Boot 核心 starter，包含自动配置、日志、yaml 配置文件的支持 |
| spring-boot-starter-actuator | 准生产特性，用来监控和管理应用 |
| spring-boot-starter-remote-shell | 提供基于 ssh 协议的监控和管理 |
| spring-boot-starter-amqp | 使用 spring-rabbit 来支持 AMQP |
| spring-boot-starter-aop | 使用 spring-aop 和 AspectJ 支持面向切面编程 |
| spring-boot-starter-batch | 对 Spring Batch 的支持 |
| spring-boot-starter-cache | 对 Spring Cache 抽象的支持 |
| spring-boot-starter-cloud-connectors | 对云平台（Cloud Foundry、Heroku）提供的服务提供简化的连接方式 |

| 名 称 | 描 述 |
|--|--|
| spring-boot-starter-data-elasticsearch | 通过 spring-data-elasticsearch 对 Elasticsearch 支持 |
| spring-boot-starter-data-gemfire | 通过 spring-data-gemfire 对分布式存储 GemFire 的支持 |
| spring-boot-starter-data-jpa | 对 JPA 的支持, 包含 spring-data-jpa、spring-orm 和 Hibernate |
| spring-boot-starter-data-mongodb | 通过 spring-data-mongodb, 对 MongoDB 进行支持 |
| spring-boot-starter-data-rest | 通过 spring-data-rest-webmvc 将 Spring Data repository 暴露为 REST 形式的服务 |
| spring-boot-starter-data-solr | 通过 spring-data-solr 对 Apache Solr 数据检索平台的支持 |
| spring-boot-starter-freemarker | 对 FreeMarker 模板引擎的支持 |
| spring-boot-starter-groovy-templates | 对 Groovy 模板引擎的支持 |
| spring-boot-starter-hateoas | 通过 spring-hateoas 对基于 HATEOAS 的 REST 形式的网络服务的支持 |
| spring-boot-starter-hornetq | 通过 HornetQ 对 JMS 的支持 |
| spring-boot-starter-integration | 对系统集成框架 spring-integration 的支持 |
| spring-boot-starter-jdbc | 对 JDBC 数据库的支持 |
| spring-boot-starter-jersey | 对 Jersey REST 形式的网络服务的支持 |
| spring-boot-starter-jta-atomikos | 通过 Atomikos 对分布式事务的支持 |
| spring-boot-starter-jta-bitronix | 通过 Bitronix 对分布式事务的支持 |
| spring-boot-starter-mail | 对 javax.mail 的支持 |
| spring-boot-starter-mobile | 对 spring-mobile 的支持 |
| spring-boot-starter-mustache | 对 Mustache 模板引擎的支持 |
| spring-boot-starter-redis | 对键值对内存数据库 Redis 的支持, 包含 spring-redis |

| | |
|-------------------------------------|--|
| spring-boot-starter-security | 对 spring-security 的支持 |
| spring-boot-starter-social-facebook | 通过 spring-social-facebook 对 Facebook 的支持 |
| spring-boot-starter-social-linkedin | 通过 spring-social-linkedin 对 Linkedin 的支持 |
| spring-boot-starter-social-twitter | 通过 spring-social-twitter 对 Twitter 的支持 |
| spring-boot-starter-test | 对常用的测试框架 JUnit、Hamcrest 和 Mockito 的支持, 包含 spring-test 模块 |
| spring-boot-starter-thymeleaf | 对 Thymeleaf 模板引擎的支持, 包含于 Spring 整合的配置 |
| spring-boot-starter-velocity | 对 Velocity 模板引擎的支持 |
| spring-boot-starter-web | 对 Web 项目开发的支持, 包含 Tomcat 和 spring-webmvc |
| spring-boot-starter-Tomcat | Spring Boot 默认的 Servlet 容器 Tomcat |
| spring-boot-starter-Jetty | 使用 Jetty 作为 Servlet 容器替换 Tomcat |
| spring-boot-starter-undertow | 使用 Undertow 作为 Servlet 容器替换 Tomcat |
| spring-boot-starter-logging | Spring Boot 默认的日志框架 Logback |
| spring-boot-starter-log4j | 支持使用 Log4J 日志框架 |
| spring-boot-starter-websocket | 对 WebSocket 开发的支持 |
| spring-boot-starter-ws | 对 Spring Web Services 的支持 |

作业

1. 练习联网创建 springboot 项目,以及手动创建 springboot 项目
2. 完成 SpringBoot 项目与 Spring, SpringMVC, MyBatis 的集成开发

