# 极速开发框架SpringBoot及源码分析

主讲：Cat 老师

**北京动力节点教育科技有限公司**  
**2020 • 北京**

**动力节点版权所有，禁止私自传播**

## 快速回顾

What is SpringBoot？

SpringBoot开发运行环境要求

springboot程序的几种创建方式

SpringBoot开发java程序

SpringBoot开发web程序

SpringBoot main方法分析

SpringBoot项目约定的代码结构

SpringBoot 项目pom.xml继承结构

SpringBoot 项目pom.xml修改默认版本

SpringBoot 整合mybatis

SpringBoot 整合jsp视图展示（也支持一些模板技术）

SpringBoot 项目pom.xml热部署插件devtools

JRebel （这是idea插件，收费的，需要破解）

SpringBoot开发最佳实践

SpringBoot敏感配置信息加密

SpringBoot数据库连接池

SpringBoot跨域资源共享CORS

SpringBoot静态资源处理

SpringBoot核心配置文件（Duration/DataSize单位类型、随机值、占位符、多环境、4种配置文件读取方式）

SpringBoot线程池异步调用（@Async、手动编码）

SpringBoot集成Swagger（OAS、springfox、swagger的关系）

## SpringBoot集成websocket

WebSocket协议是由HTML5定义的，基于TCP协议实现的一种网络协议，通过该协议服务器可以主动向客户端发送信息；

WebSocket 协议在2008年诞生，2011年成为W3C国际标准；

我们已经有了 HTTP 协议，为什么出现一个websocket协议？

http协议是短连接，因为请求之后，都会关闭连接，下次重新请求数据，需要再次打开链接；

WebSocket协议是一种长连接，只需要通过一次请求来初始化连接，然后所有的请求和响应都是通过这个TCP连接进行通讯；

所以HTTP协议通信只能是客户端向服务器发出请求，服务器返回响应结果，HTTP 协议做不到服务器主动向客户端推送信息，而websocket能实现服务器和客户端全双工通信；

何谓全双工

信息只能单向传送为单工；信息能双向传送但不能同时双向传送称为半双工，信息能够同时双向传送则称为全双工；

基本实现原理

WebSocket协议基于TCP协议实现，客户端和服务器只需要做一个握手的动作之后，形成了一条基于客户端和服务器之间的快速通道，之后客户端与服务端之间便可以进行多次数据帧双向传输；

这样实现的目的是客户端和服务器进行频繁双向通信时，可以使服务器避免频繁创建HTTP连接，节约资源，提高工作效率和资源利用率；

**传统Web推送实现**

在没有WebSocket协议之前，服务器如何向浏览器端推送消息？

此时，通常的实现方式是在页面通过Ajax定时轮询，比如每隔1秒中向服务器发送一次HTTP请求，询问服务器是否有新消息，服务器返回结果；

这种形式缺点很明显，浏览器需要不断的向服务器发出HTTP请求，而HTTP请求包含较长的头部，有效信息相对较少，反复的无效请求占用了大量的带宽和 CPU 资源，造成很大的浪费，所以，WebSocket 应运而生；

HTML5定义的WebSocket协议，能更好的节省服务器资源和带宽，并且能够更实时地进行通讯；

WebSocket协议本质上是一个基于TCP的协议，因此与HTTP协议没有什么关系；

**WebSocket的特点**

全双工通信，客户端和服务器可以双向平等通信；

建立在TCP协议之上，服务器端的实现比较容易；

数据格式比较轻量，性能开销小，通信高效；

可以发送文本，也可以发送二进制数据；

通信具有更强的实时性；

协议标识符是ws，服务器地址就是：ws://www.abc.com/some/path

http协议： http://

**websocket业务场景**

WebSocket聊天室；

股票实时价格显示等应用；

即时通讯、游戏、可视化大屏展示等领域；

企业内部管理通讯等功能，主要通讯协议websocket；

web网页聊天、客服系统实现；

系统提醒、用户上下线提醒、客户端同步，实时数据更新，多屏幕同步，用户在线状态，消息通知，扫描二维码登录/二维码支付，弹幕、各类信息提醒，在线选座，实时监控大屏等等；

## Java中的WebSocket API

在Java EE 7中Java语言开始支持websocket协议，Java EE 7中定义了一套Websocket API规范，也就是一系列接口，没有实现，位于包javax.websocket下，包含客户端API和服务端API，WebSocket的Java API 只是规范，具体实现需要web容器（比如tomcat就实现了Java websocket api）、Java EE服务器或者框架提供；

现在最新版本是Oracle Java EE 8； 以后的Java EE版本升级是eclipse去做，以后Oracle就不管Java EE的维护和升级；

|  |  |
| --- | --- |
| [javax.websocket](mk:@MSITStore:G:\\vip\\02-第二期\\03-源码分析专题\\03-SpringBoot\\JavaEE7_API.CHM::/javax/websocket/package-summary.html) | This package contains all the WebSocket APIs common to both the client and server side. |
| [javax.websocket.server](mk:@MSITStore:G:\\vip\\02-第二期\\03-源码分析专题\\03-SpringBoot\\JavaEE7_API.CHM::/javax/websocket/server/package-summary.html) | This package contains all the WebSocket APIs used only by server side applications. |

在Java中服务端的实现

1、Tomcat：java中的websocket实现，需要tomcat 7.0.47+以上才支持，

Java EE7的支持；

2、Spring的websocket，需要Spring 4.x，所以springboot也可以用；

## Websocket开发相关注解及API方法

@ServerEndpoint("/websocket/{uid}")

申明这是一个websocket服务；

需要指定访问该服务的地址，在地址中可以指定参数，需要通过{}进行占位；

@OnOpen

用法：public void onOpen(Session session, @PathParam("uid") String uid) throws IOException{}

该方法将在建立连接后执行，会传入session对象，就是客户端与服务端建立的长连接通道，通过@PathParam获取url中声明的参数；

@OnClose

用法：public void onClose() {}

该方法是在连接关闭后执行；

@OnMessage

用法：public void onMessage(String message, Session session) throws IOException {}

该方法用于接收客户端发送的消息；

message：发来的消息数据；

session：会话对象（也是长连接通道）；

发送消息到客户端；

用法：session.getBasicRemote().sendText("hello,websocket.");

通过session进行消息发送；

## 前端技术对websocket的支持

Websocket是html5规范，主流浏览器都支持；（某些老浏览器不支持）

jQuery、vueJS、React JS、angularjs等都可以支持webscoket对象；

底层是javascript支持的一个webscoket的js对象，通过这个对象可以建立websocket的连接：*ws://localhost:8080/websocket/12345*

## 第三方商业方案goeasy

GoEasy实时Web推送，支持后台推送和前台推送两种：后台推送可以选择Java SDK、Restful API支持所有开发语言；前台推送：JS推送，由于它支持websocket 和polling两种连接方式所以兼顾大多数主流浏览器，低版本的IE浏览器也是支持的；

地址：<http://goeasy.io/>

WebSocket有很多优点，弥补了http协议的不足，但是如果服务端维护很多长连接也是很耗费资源的，服务器集群带来了复杂性以及存在老版本浏览器兼容性问题，各大厂目前大多数还是基于轮询的方式实现的，比如：扫码登录、支付成功通知、秒杀成功通知等；

在秒杀场景中，用轮询即可，前端轮询一般不可能穿透到后端数据库查询，一般是采用高性能的缓存标记来判定秒杀是否成功，相对于websocket长连接以及其不确定因素，轮询依然是相对比较合适的方案；

Websocket各浏览器的支持情况：

<https://caniuse.com/#search=websocket>

作业：实现秒杀成功后给页面推送秒杀结果；

## 注册Servlets, Filters, Listeners作为Spring Beans

使用嵌入式Servlet容器时，可以通过使用Spring Bean或扫描Servlet组件来从Servlet规范中注册Servlet，过滤器和所有监听器（例HttpSessionListener）；

@Bean

public ServletRegistrationBean heServletRegistrationBean(){

ServletRegistrationBean registration = new ServletRegistrationBean(new HeServlet(), "/heServlet");

return registration;

}

@Bean

public FilterRegistrationBean heFilterRegistration() {

FilterRegistrationBean registration = new FilterRegistrationBean(new HeFilter());

registration.addUrlPatterns("/\*");

return registration;

}

@ServletComponentScan

使用@ServletComponentScan启用自动注册带有@ WebServlet，@ WebFilter和@WebListener的类；

默认情况下，如果上下文仅包含单个Servlet，则将其映射到/，对于多个servlet bean，bean名称用作路径前缀；

如果基于约定的映射不够灵活，则可以使用ServletRegistrationBean，FilterRegistrationBean和ServletListenerRegistrationBean类进行完全控制；

具体参考代码和视频演示；

## SpringBoot使用拦截器

1、按照Spring mvc的方式编写一个拦截器类；

2、编写一个配置类implements WebMvcConfigurer接口

3、为该配置类添加@Configuration注解，标注此类为一个配置类，让Spring boot 扫描到；

4、覆盖其中的方法并添加已经编写好的拦截器：

@Override

public void addInterceptors(InterceptorRegistry registry) {

//对来自 /api/\*\* 链接的请求进行拦截，对登录请求/api/login不进行拦截

registry.addInterceptor(new LoginInterceptor())

.addPathPatterns("/api/\*\*")

.excludePathPatterns("/api/login");

}

## SpringBoot内嵌Web服务器

Spring Boot 默认是使用Tomcat 来作为内嵌的Servlet 容器运行我们的web程序，我们也可以将Servlet容器切换为其他服务器，比如Undertow 、Jetty服务器，只需要通过pom.xml文件中配置即可；

Undertow 是什么？

在早些年，有一款基于 Java EE 的免费开源的应用服务器Jboss (也叫Jboss AS) ，由 Jboss 社区维护（另外注意 Jboss 旗下还有另一款商用应用服务 JBoss EAP，由Radhat提供商业支持）；

Jboss 服务器的核心服务仅支持 EJB，不支持 JSP 和 Servelt，所以 Jboss 之前一直在内部绑定 Tomcat 作为 Web 容器，但后来Jboss不想一直绑定tomcat作为web容器，所以到 Jboss 8.0（改名叫 wildfly）的版本后，Jboss 终于有了一款属于自己的 Web 容器“Undertow”；

Jboss（内嵌Tomcat） --> wildfly（内嵌Undertow）

Jboss 8.0（改名叫wildfly） <https://wildfly.org> （内嵌 <http://undertow.io>）

认识一下Jetty？

Jetty 是一款由Eclipse开源的Servlet容器，为JSP和servlet提供运行环境，Jetty是使用Java语言编写的；

<https://www.eclipse.org/jetty/>

SpringBoot使用Undertow 或Jetty 服务器，修改pom文件：

1、排除掉Tomcat：

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 <**exclusions**>  
 <**exclusion**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-tomcat</**artifactId**>  
 </**exclusion**>  
 </**exclusions**>  
</**dependency**>

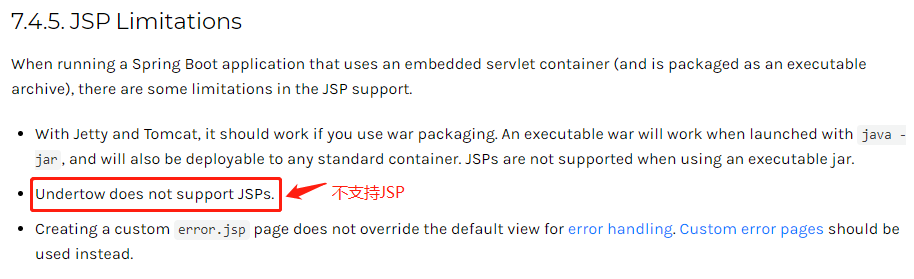
2、添加undertow或者jetty的依赖；

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

启动日志：

Undertow started on port(s) 8090 (http) with context path '/24-springboot'

注意：undertow不支持JSP；



[https://docs.spring.io/spring-boot/docs/2.3.1.RELEASE/reference/html/spring-boot-features.html#boot-features-jsp-limitations](https://docs.spring.io/spring-boot/docs/2.2.1.RELEASE/reference/html/spring-boot-features.html#boot-features-jsp-limitations)

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

启动日志：

Jetty started on port(s) 8090 (http/1.1) with context path '/25-springboot'

Jetty要支持JSP，需再添加几个依赖：

*<!-- jetty -->*<**dependency**>  
 <**groupId**>org.mortbay.jetty</**groupId**>  
 <**artifactId**>jetty</**artifactId**>  
 <**version**>6.1.26</**version**>  
</**dependency**>  
  
*<!-- jsp-2.1 -->*<**dependency**>  
 <**groupId**>org.mortbay.jetty</**groupId**>  
 <**artifactId**>jsp-2.1</**artifactId**>  
 <**version**>6.1.14</**version**>  
</**dependency**>

重新启动后根据控制台输出的日志发现容器已经切换成undertow、jetty；

当运行使用嵌入式Servlet容器的Spring Boot应用程序时，JSP支持存在一些限制；

使用java -jar运行时候，可能会出现不支持JSP；

像Undertow就完全不支持JSP；

## SpringBoot打包部署

### 打war包

1. 程序入口类需扩展继承 SpringBootServletInitializer 类

2、程序入口类覆盖如下方法：

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {

return application.sources(SpringbootApplication.class);

}

3、更新包为war，在 pom.xml 中修改 <packaging>war</packaging>

4、配置springboot打包的插件

<!-- Springboot打包的插件 -->

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

注意pom.xml文件添加build信息：（避免一些资源没有打到包里面去）

<resource>

<!--源文件位置-->

<directory>src/main/java</directory>

<includes>

<!--要把哪些文件编译过去.\*.\*表示所有-->

<include>\*\*/\*.xml</include>

</includes>

</resource>

<resource>

<!--源文件位置-->

<directory>src/main/resources</directory>

<includes>

<!--要把哪些文件编译过去.\*.\*表示所有-->

<include>\*\*/\*.\*</include>

</includes>

</resource>

<resource>

<!--源文件位置-->

<directory>src/main/webapp</directory>

<!--编译到META-INF/resources目录下-->

<targetPath>META-INF/resources</targetPath>

<includes>

<!--要把哪些文件编译过去.\*.\*表示所有-->

<include>\*\*/\*.\*</include>

</includes>

</resource>

1. 在项目中通过 Maven package 命令打一个war包，然后将war包部署到tomcat下运行；

### 打jar包

对Jsp的支持有一些限制，打的jar不支持jsp访问；

1、Spring boot程序打jar包，在pom.xml文件加入如下Spring boot的maven插件：

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

<!--使用1.4.2版本打jar包，jsp页面才能访问，否则访问不到-->

<version>1.4.2.RELEASE</version>

</plugin>

</plugins>

</build>

上面这个插件打jar包请使用1.4.2版本，其他版本有点问题，就是jsp无法访问）

2、在项目中使用 Maven package命令打一个jar包；

3、使用java -jar 运行第2步生成的jar包，从而可以启动 Spring boot 程序；

4、访问第3步运行起来的 spring boot程序；