Spring Boot

Spring 旨在简化 J2EE 企业应用程序**Enterprise JavaBean（EJB）** 开发，**Spring Boot**旨在简化 Spring 开发（减少配置文件，开箱即用！）

Java SE & EE , **Simple** --> Spring Framework , **Simpler** --> Spring Boot , **Simplest**

优点

1. 开发基于 Spring 的应用程序很容易。
2. Spring Boot 项目所需的开发或工程时间明显减少，通常会提高整体生产力。
3. Spring Boot 不需要编写大量样板代码、XML 配置和注释。
4. Spring 引导应用程序可以很容易地与 Spring 生态系统集成，如 Spring JDBC、Spring ORM、Spring Data、Spring Security 等。
5. Spring Boot 遵循“固执己见的默认配置”，以减少开发工作（默认配置可以修改）。
6. Spring Boot 应用程序提供嵌入式 HTTP 服务器，如 Tomcat 和 Jetty，可以轻松地开发和测试 web 应用程序。（这点很赞！普通运行 Java 程序的方式就能运行基于 Spring Boot web 项目，省事很多）
7. Spring Boot 提供命令行接口(CLI)工具，用于开发和测试 Spring Boot 应用程序，如 Java 或 Groovy。
8. Spring Boot 提供了多种插件，可以使用内置工具(如 Maven 和 Gradle)开发和测试 Spring Boot 应用程序。

Spring Boot Starters

Spring Boot Starters 是一系列依赖关系的集合

在没有 Spring Boot Starters 之前，我们开发 REST 服务或 Web 应用程序时; 我们需要使用像 Spring MVC，Tomcat 和 Jackson 这样的库，这些依赖我们需要手动一个一个添加。但是，有了 Spring Boot Starters 我们只需要一个只需添加一个**spring-boot-starter-web**一个依赖就可以了

内嵌容器

Spring Boot支持内嵌Servlet容器，Tomcat(默认),Jetty,Undertow

修改默认内嵌容器，修改pom.xml即可

```
<!--从web启动器依赖中排除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>
```

```
<!--添加Jetty依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

注解

@SpringBootApplication

`@SpringBootApplication`可以看作是`@EnableAutoConfiguration`、`@ComponentScan`、`@Configuration`注解的集合。

- `@EnableAutoConfiguration`： 启用 SpringBoot 的自动配置机制
- `@ComponentScan`： 扫描被`@Component`(`@Service`,`@Controller`)注解的 bean，注解默认会扫描该类所在的包下所有的类。
- `@Configuration`： 允许在上下文中注册额外的 bean 或导入其他配置类

常用注解

Spring Bean 相关:

- `@Autowired`：自动导入对象到类中，被注入进的类同样要被 Spring 容器管理。
- `@RestController`：`@RestController`注解是`@Controller`和`@ResponseBody`的合集,表示这是个控制器 bean,并且是将函数的返回值直 接填入 HTTP 响应体中,是 REST 风格的控制器。
- `@Component`： 通用的注解，可标注任意类为 `Spring` 组件。如果一个 Bean 不知道属于哪个层，可以使用`@Component` 注解标注。
- `@Repository`：对应持久层即 Dao 层，主要用于数据库相关操作。
- `@Service`：对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao 层。
- `@Controller`：对应 Spring MVC 控制层，主要用于接受用户请求并调用 Service 层返回数据给前端页面。

处理常见的 HTTP 请求类型:

- `@GetMapping`：GET 请求、
- `@PostMapping`：POST 请求。
- `@PutMapping`：PUT 请求。
- `@DeleteMapping`：DELETE 请求。

前后端传值:

- `@RequestParam`以及`@Pathvairable ： @PathVariable`用于获取路径参数，`@RequestParam`用于获取查询参数。
- `@RequestBody`： 用于读取 Request 请求（可能是 POST,PUT,DELETE,GET 请求）的 body 部分并且 Content-Type 为 `application/json` 格式的数据，接收到数据之后会自动将数据绑定到 Java 对象上去。系统会使用`HttpMessageConverter`或者自定义的`HttpMessageConverter`将请求的 body 中的 json 字符串转换为 java 对象。

SpringBoot自动加载

自动装配，Spring Framework 早就实现了这个功能。Spring Boot 只是在其基础上，通过 SPI 的方式，做了进一步优化。

引入 starter 之后，我们通过少量注解和一些简单的配置就能使用第三方组件提供的功能

Spring Boot 通过`@EnableAutoConfiguration`开启自动装配，通过 SpringFactoriesLoader 最终加载`META-INF/spring.factories`中的自动配置类实现自动装配，自动配置类其实就是通过`@Conditional`按需加载的配置类，想要其生效必须引入`spring-boot-starter-xxx`包实现起步依赖

扩展

先了解SPI(https://zhuanlan.zhihu.com/p/28909673)

> SPI 全称为 (Service Provider Interface) ,是JDK内置的一种服务提供发现机制

通用点讲，SPI就是为某个接口寻找服务实现的机制

SPI概念

调用方 -- 接口 -- 实现方

当接口与实现都在实现方，通常称为api

当接口在调用方， 实现在独立的包中，spi的特征

SPI案例，DriverManager(jdbc里管理和注册不同数据库driver的工具类)

针对一个数据库，可能会存在着不同的数据库驱动实现，典型的SPI特征

SPRING应用的地方也很多，component-scan标签，scope概念，ConfigurableBeanFactory等等

SPI核心方法 ServiceLoader.load

配置

Spring Boot 常用的两种配置文件，application.properties`或者`application.yml

YAML 配置的方式有一个缺点，那就是不支持`@PropertySource`注解导入自定义的 YAML 配置

获取注解方式

1. @Value，不建议使用

```
@Value("${tokenizeFlag: false}")
private Boolean tokenizeFlag;

@Value("#{'${prdCodes}'.split(',')}")
private List<String> prdCodes;
```

2. @ConfigurationProperties读取并与bean绑定

```
@Component
@ConfigurationProperties(prefix = "library")
```

配置文件优先级

src同级目录config下的application.yml > src/resources目录下的config下的application.yml > resources下的

Spring Boot 监控系统实际运行状况

可以使用 Spring Boot Actuator 来对 Spring Boot 项目进行简单的监控

集成了这个模块之后，你的 Spring Boot 应用程序就自带了一些开箱即用的获取程序运行时的内部状态信息的 API。

比如通过 GET 方法访问 `/health` 接口，你就可以获取应用程序的健康指标。

Spring Boot请求参数校验

Spring Boot 程序做请求参数校验的话只需要 `spring-boot-starter-web` 依赖就够了，它的子依赖包含了我们所需要的东西

校验注解

JSR 提供的校验注解:

- `@Null` 被注释的元素必须为 null
- `@NotNull` 被注释的元素必须不为 null
- `@AssertTrue` 被注释的元素必须为 true
- `@AssertFalse` 被注释的元素必须为 false
- `@Min(value)` 被注释的元素必须是一个数字，其值必须大于等于指定的最小值
- `@Max(value)` 被注释的元素必须是一个数字，其值必须小于等于指定的最大值
- `@DecimalMin(value)` 被注释的元素必须是一个数字，其值必须大于等于指定的最小值
- `@DecimalMax(value)` 被注释的元素必须是一个数字，其值必须小于等于指定的最大值
- `@Size(max=, min=)` 被注释的元素的大小必须在指定的范围内
- `@Digits (integer, fraction)` 被注释的元素必须是一个数字，其值必须在可接受的范围内
- `@Past` 被注释的元素必须是一个过去的日期
- `@Future` 被注释的元素必须是一个将来的日期
- `@Pattern(regex=,flag=)` 被注释的元素必须符合指定的正则表达式

Hibernate Validator 提供的校验注解:

- `@NotBlank(message =)` 验证字符串非 null，且长度必须大于 0
- `@Email` 被注释的元素必须是电子邮箱地址
- `@Length(min=,max=)` 被注释的字符串的大小必须在指定的范围内
- `@NotEmpty` 被注释的字符串的必须非空
- `@Range(min=,max=,message=)` 被注释的元素必须在合适的范围内

Spring Boot全局异常处理

可以使用 `@ControllerAdvice` 和 `@ExceptionHandler` 处理全局异常

Spring Boot实现定时任务

使用 `@Scheduled` 注解就能很方便地创建一个定时任务（SpringBoot 中我们只需要在启动类上加上 `@EnableScheduling` 注解），`@EnableScheduling` 注解的作用是发现注解 `@Scheduled` 的任务并在后台执行该任务

Hystrixs 隔离实现

信号量，线程池

因为信号量同步，无法支持超时，一般选用线程池，多了一层，需要配置好

线程池最佳实践

线程池资源优化演进

在生产环境中部署一个短路器，一开始需要将一些关键配置设置的大一些，比如timeout超时时长，线程池大小，或信号量容量，然后逐渐优化这些配置，直到在一个生产系统中运作良好。

（1）一开始先不要设置timeout超时时长，默认就是1000ms，也就是1s
（2）一开始也不要设置线程池大小，默认就是10
（3）直接部署hystrix到生产环境，如果运行的很良好，那么就让它这样运行好了
（4）让hystrix应用，24小时运行在生产环境中
（5）依赖标准的监控和报警机制来捕获到系统的异常运行情况
（6）在24小时之后，看一下调用延迟的占比，以及流量，来计算出让短路器生效的最小的配置数字
（7）直接对hystrix配置进行热修改，然后继续在hystrix dashboard上监控
（8）看看修改配置后的系统表现有没有改善

HystrixThreadPool线程数量设定计算公式

每秒的高峰访问次数 * 99%的访问延时 + 缓冲数量

假设一个依赖的服务的高峰访问次数30QPS，99%的请求延时实践在200ms，那么我们可以设定的线程数量=30*0.2+4=10个线程

HystrixCommand线程超时timeout设定

结合线程池数量配置，设置能够完成执行高峰时QPS任务的timeout数值。

> 默认线程超时为1s。假设一个依赖服务的高峰访问时，99.5%的请求延时在250ms，假设再允许服务重试一次消耗50ms，那么这个依赖服务的延时设置为300ms就可以了，这样一个线程最多延时300ms，也就是这个线程每秒能执行三次依赖服务，线程池中10个线程，整好能处理30QPS。如果把延时设置为400ms，那么一个线程每秒最多执行2次请求，10个线程在1s内最多执行20个请求，剩余的10个请求就会被线程池阻塞住，再来新的请求也继续会被阻塞住，如果等待队列已满，那么新来的请求就会被直接Reject。所以400ms的超时设置就不能满足30QPS的请求次数要求。

生产环境中动态分配线程池资源

线程池动态配置选项

//允许线程池自动从coreSize扩容到maximumSize

.withAllowMaximumSizeToDivergeFromCoreSize(true)

//default 10，默认线程池大小

.withCoreSize(10)

//最大线程数

.withMaximumSize(30)

//空闲线程存活时间1分钟，超时后线程池数量自动恢复到coreSize大小

.withKeepAliveTimeMinutes(1)