

Spring Boot 进阶

不才陈某

Watermark

问候一下世界

前言

相信从事Java开发的朋友都听说过 `SSM` 框架，这还算年轻的，老点的甚至经历过 `SSH`，说起来有点恐怖，哈哈。比如我就是经历过 `SSH` 那个时代末流，没办法，很无奈。

当然无论是`SSM`还是`SSH`都不是今天的重点，今天要说的是 `Spring Boot`，一个令人眼前一亮的框架，从大的说，`Spring Boot`取代了 `SSM` 中的 `SS` 的角色。

今天这篇文章就来谈谈`Spring Boot`，这个我第一次使用直呼 `爽` 的框架。

什么是`Spring Boot`？

`Spring Boot` 是由 Pivotal 团队提供的全新框架。`Spring Boot` 是所有基于 `Spring Framework 5.0` 开发的项目的起点。`Spring Boot` 的设计是为了让你尽可能快的跑起来 `Spring` 应用程序并且尽可能减少你的配置文件。

`Spring Boot` 的设计目的简单一句话：简化`Spring`应用的初始搭建以及开发过程。

从最根本上来讲，`Spring Boot` 就是一些库的集合，它能够被任意项目的构建系统所使用。它使用 “**约定大于配置**”（项目中存在大量的配置，此外还内置一个习惯性的配置）的理念让你的项目快速运行起来。

约定大于配置这个如何理解？其实简单的来说就是`Spring Boot`在搭建之初就内置了许多实际开发中的常用配置，只有少部分的配置需要开发人员自己去配置。

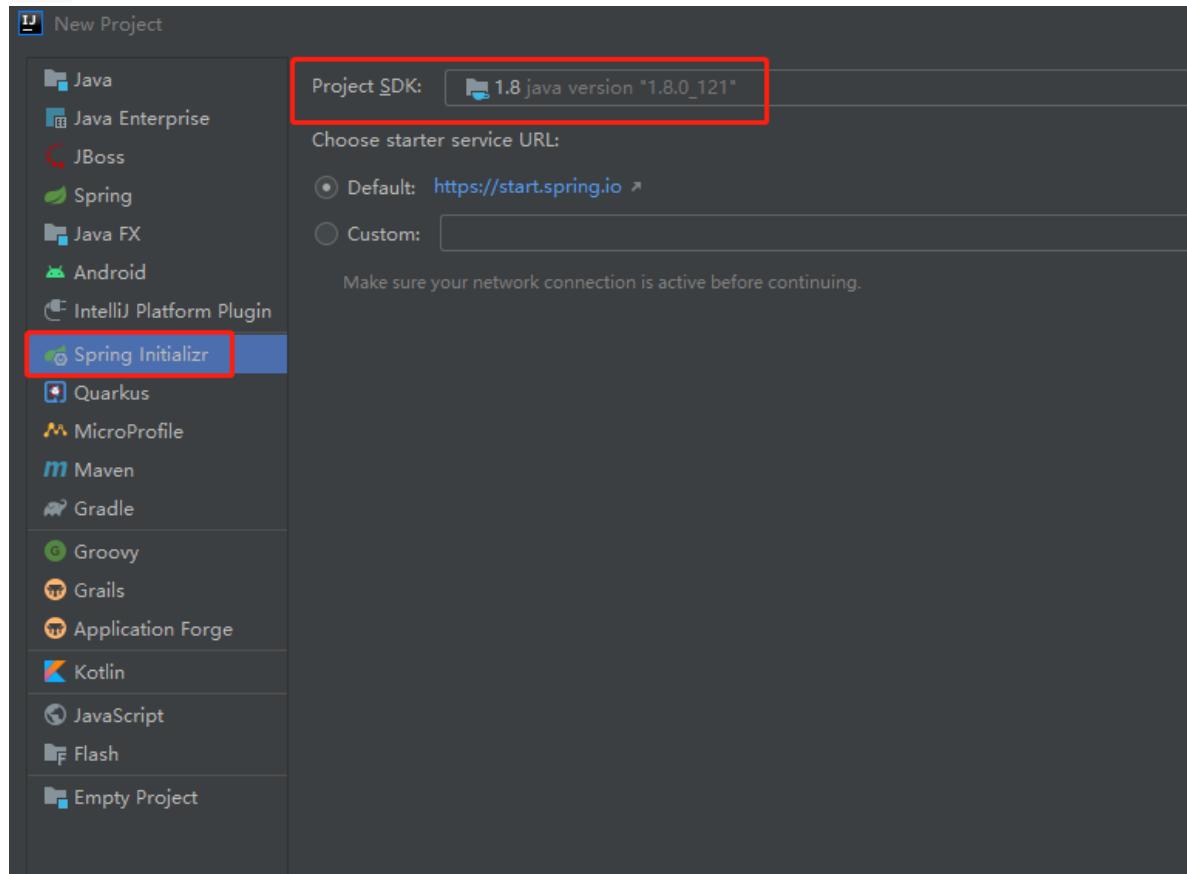
如何搭建一个`Spring Boot`项目？

其实搭建一个SpringBoot项目有很多种方式，最常见的两种方式如下：

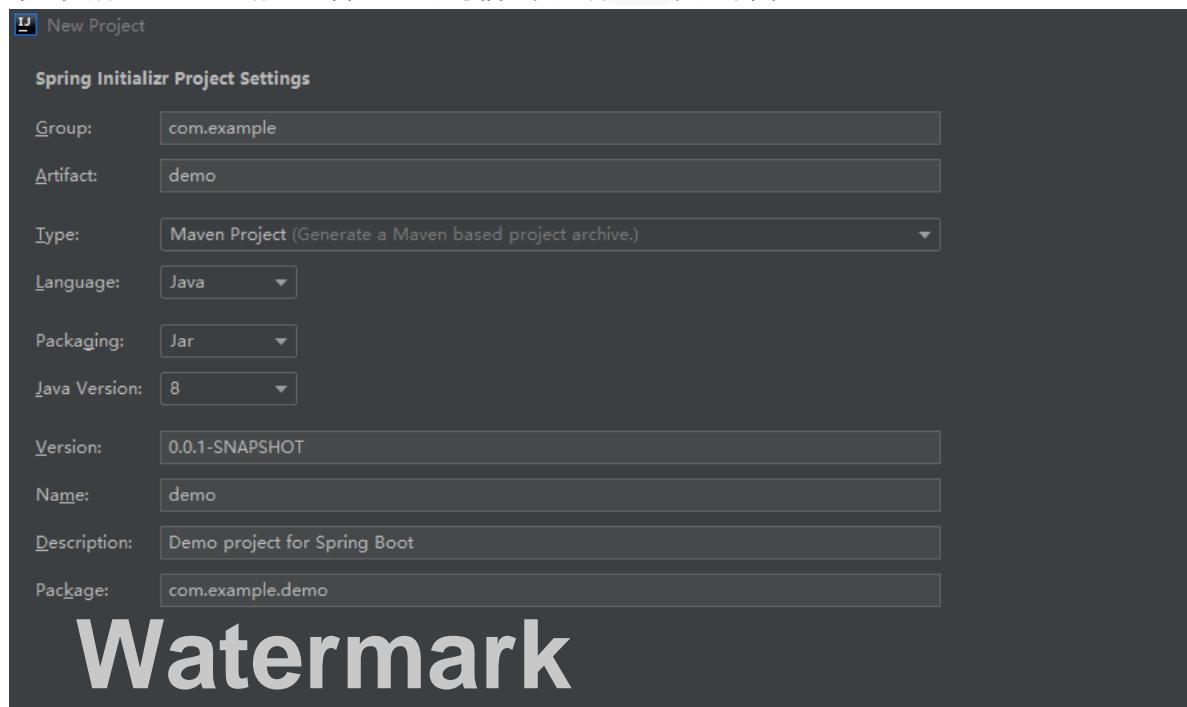
1. 创建Maven项目，自己引入依赖，创建启动类和配置文件。
2. 直接IDEA中的 Spring Initializr 创建项目。

第一种方式不适合入门的朋友玩，今天演示第二种方式搭建一个Spring Boot项目。

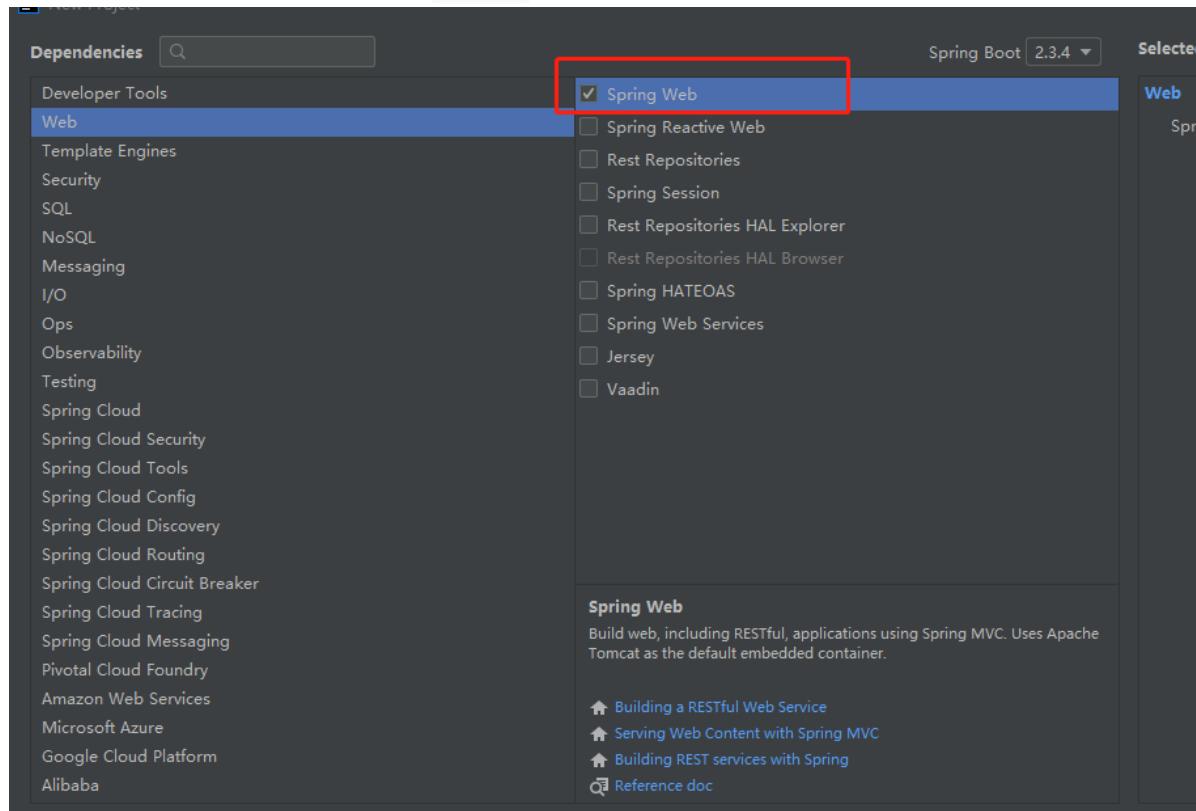
第一步在IDEA中选择 File-->NEW-->Project，选择 Spring Initializr，指定 JDK 版本 1.8，然后 Next。如下图：



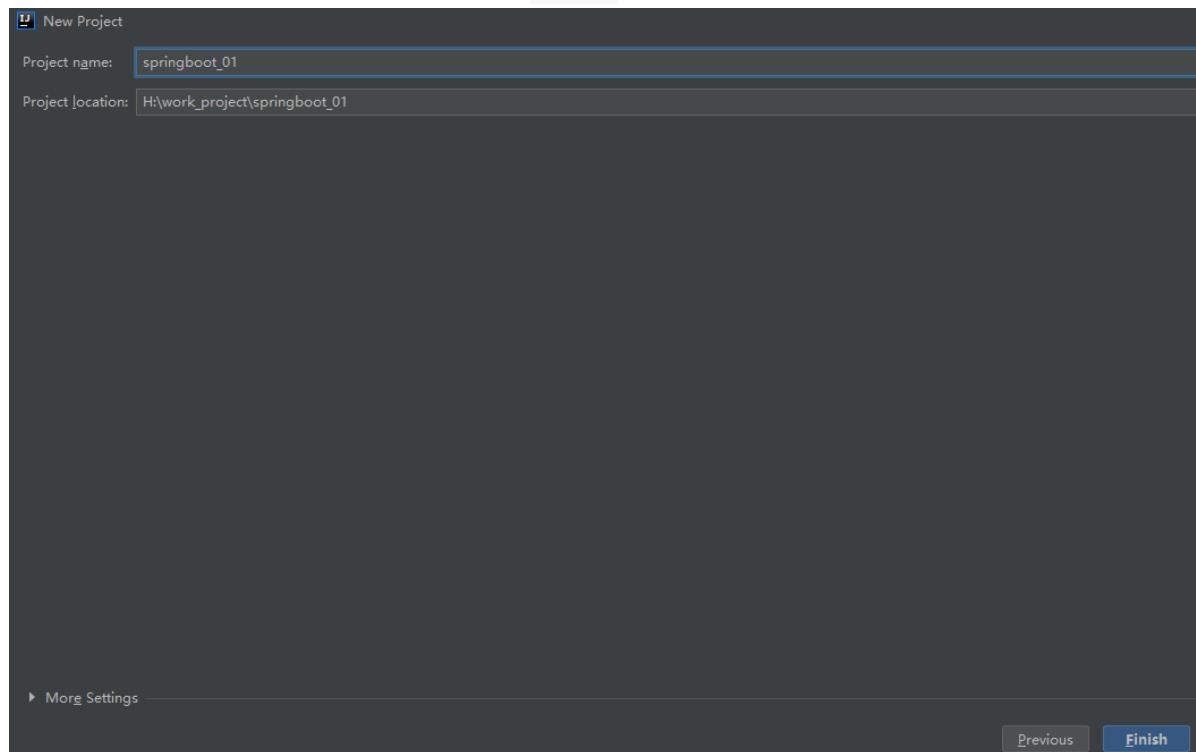
第二步指定Maven坐标、包名、JDK版等信息，然后 Next，如下图：



第三步选择自己所需要的依赖、Spring Boot的版本，Spring Boot与各个框架适配都是以 **starter** 方式，这里我们选择WEB开发所需的 **starter** 即可，如下图：

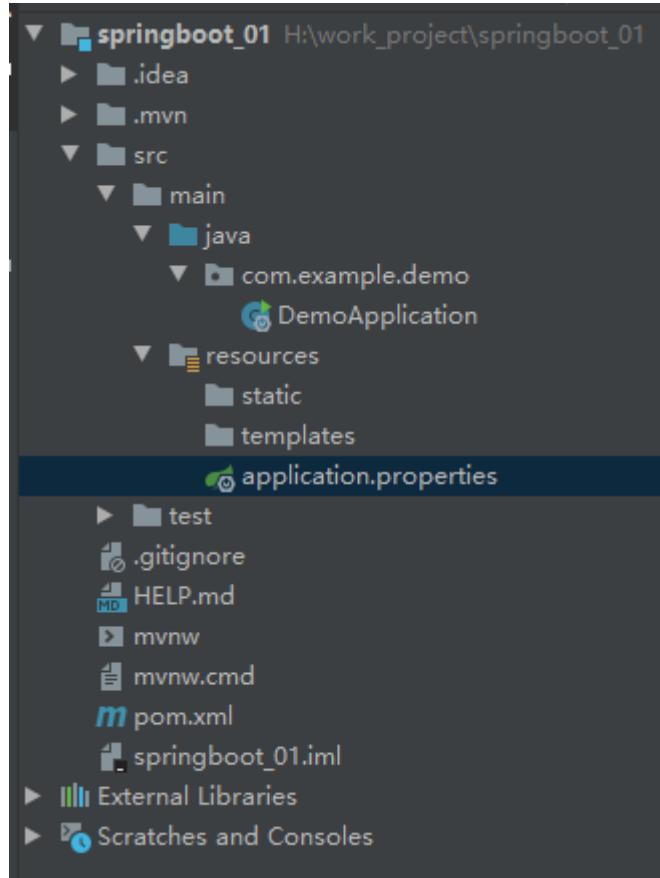


第四步指定项目的名称，路径即可完成，点击 **Finish** 等待创建成功，如下图：



Watermark

创建成功的项目如下图：



其中的 `DemoApplication` 是项目的启动类，里面有一个 `main()` 方法就是用来启动Spring Boot。
`application.properties` 是Spring Boot的配置文件。

此时可以启动项目，在 `DemoApplication` 运行 `main` 方法即可启动，启动成功如下图：

```
com.example.demo.DemoApplication      : Starting DemoApplication on DESKTOP-KJ3N2KS with PID 14308 (H:\work
com.example.demo.DemoApplication      : No active profile set, falling back to default profiles: default
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
o.apache.catalina.core.StandardService : Starting service [Tomcat]
org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.38]
o.a.c.c.C.[Tomcat].[localhost].[]       : Initializing Spring embedded WebApplicationContext
w.s.c.ServletWebServerApplicationContext: Root WebApplicationContext: initialization completed in 2566 ms
o.s.s.concurrent.ThreadPoolTaskExecutor: Initializing ExecutorService 'applicationTaskExecutor'
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
com.example.demo.DemoApplication      : Started DemoApplication in 4.949 seconds (JVM running for 8.863)
```

由于SpringBoot默认内置了Tomcat，因此启动的默认端口就是 8080。

第一个程序 Hello World

学习任何一种技术总是要问候一下世界，哈哈.....

既然是WEB开发，就写个接口吧，前面创建的时候已经引用了 WEB 的 `starter`，如果没有引用，则可以在 `pom.xml` 引入以下依赖：

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

- 下面写一个 `HelloWorldController` 如下：

```
package com.example.demo.controller;

@RestController
public class HelloWorldController {
    @RequestMapping("/hello")
    public String helloWorld() {
        return "Hello World";
    }
}
```

`@RestController`：标记这是一个 controller，是 `@Controller` 和 `@ResponseBody` 这两个注解的集合。

`@RequestMapping`：指定一个映射

以上两个注解都是Spring中的，这里就不再细说了。

由于内置的Tomcat默认端口是 8080，所以启动项目，访问 <http://127.0.0.1:8080/hello> 即可。

依赖解读

Spring Boot项目中的 `pom.xml` 中有这么一个依赖，如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.4.RELEASE</version>
    <relativePath/>
</parent>
```

`<parent>` 这个标签都知道什么意思，`父亲` 是吧，这么个标签主要的作用就是用于版本控制。这也就是引入的 WEB 模块 `starter` 的时候不用指定版本号 `<version>` 标签的原因，因为在 `spring-boot-starter-parent` 中已经指定了，类似于一种继承的关系，父亲已经为你提供了，你只需要选择用不用就行。

为什么引入 `spring-boot-starter-web` 就能使用 Spring mvc 的功能呢？

这确实是个难以理解的问题，为了理解这个问题，我们不妨看一下 `spring-boot-starter-web` 这个启动器都依赖了什么？如下：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
        <version>2.3.4.RELEASE</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-json</artifactId>
        <version>2.3.4.RELEASE</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <version>2.3.4.RELEASE</version>
        <scope>compile</scope>
    </dependency>
```

```
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.2.9.RELEASE</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.9.RELEASE</version>
    <scope>compile</scope>
</dependency>
</dependencies>
```

看到这应该明白了吧，`spring-boot-starter-web` 这个 `starter` 中其实内部引入了 `Spring`、`springmvc`、`tomcat` 的相关依赖，当然能够直接使用 Spring MVC 相关的功能了。

什么是配置文件？

前面说过 `application.properties` 是 Spring Boot 的配置文件，那么这个配置文件究竟是配置什么的呢？

其实 Spring Boot 为了能够适配每一个组件，都会提供一个 `starter`，但是这些启动器的一些信息不能在内部写死啊，比如数据库的用户名、密码等，肯定要由开发人员指定啊，于是就统一写在了一个 `Properties` 类中，在 Spring Boot 启动的时候根据 `前缀名+属性名称` 从配置文件中读取，比如 `WebMvcProperties`，其中定义了一些 Spring Mvc 相关的配置，前缀是 `spring.mvc`。如下：

```
@ConfigurationProperties(prefix = "spring.mvc")
public class WebMvcProperties {
```

那么我们需要修改 Spring Mvc 相关的配置，只需要在 `application.properties` 文件中指定 `spring.mvc.xxxx=xxxx` 即可。

其实配置文件这块还是有许多道道儿的，后面文章会详细介绍。

什么是启动类？

前面说过启动类是 `DemoApplication`，源码如下：

```
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

`@SpringBootApplication` 是什么？其实一眼看上去，这个类在平常不过了，唯一显眼的就是 `@SpringBootApplication` 这个注解了，当然主要的作用还真是它。这个注解的源码如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {}
```

我滴乖乖儿，注解叠加啊，完全是由 `@SpringBootConfiguration`、`@EnableAutoConfiguration`、`@ComponentScan` 这三个注解叠加而来。

`ComponentScan`：这个注解并不陌生，Spring中的注解，包扫描的注解，这个注解的作用就是在项目启动的时候扫描**启动类的同类级以及下级包中的Bean**。

`@SpringBootConfiguration`：这个注解使Spring Boot的注解，源码如下：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {
    @AliasFor(
        annotation = Configuration.class
    )
    boolean proxyBeanMethods() default true;
}
```

从源码可以看出，`@SpringBootConfiguration` 完全就是的 `@Configuration` 注解，`@Configuration` 是 Spring中的注解，表示该类是一个配置类，因此我们可以在启动类中做一些配置类可以做的事，比如注入一个 Bean。

`@EnableAutoConfiguration`：这个注解看到这个名字就知道怎么回事了，直接翻译码，**开启自动配置**，真如其名，源码如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

又是一个熟悉的注解 `@Import`，什么功能呢？**快速导入Bean到IOC容器中**，有三种方式，这里用的是其中一种 `ImportSelector` 方式。不是本文重点，不再细说。

`@EnableAutoConfiguration` 这个注解的作用也就一目了然了，无非就是 `@Import` 的一种形式而已，在项目启动的时候向IOC容器中快速注入 Bean 而已。

好了，启动类就先介绍到这，后续讲到源码文章才能更清楚的了解到这个类的强大之处。

Watermark

如何进行单元测试？

Spring Boot项目创建之处为我们提供了一个单元测试的类，如下：

```
@SpringBootTest  
class DemoApplicationTests {  
  
    @Test  
    void contextLoads() {  
    }  
  
}
```

`@SpringBootTest`：这个注解指定这个类是单元测试的类。

在这个类中能够自动的获取IOC容器中的Bean，比如：

```
@SpringBootTest  
class DemoApplicationTests {  
  
    @Autowired  
    private HelloWorldController helloWorldController;
```

简单的介绍下而已，实际开发中用不到，随着项目越来越大，启动的时间越来越长，谁会傻到启动一个测试方法来检验代码，纯粹浪费时间。

总结

作为Spring Boot的第一弹，写到这儿就结束了，没什么的深入的内容，只是简单的对Spring Boot做了初步的了解。

本文使用的开发工具是 [IDEA](#)，有需要 [2020](#) 版本的公众号回复关键词 [IDEA2020](#)，有需要IDEA破解包的回复关键词 [IDEA破解包](#)



配置文件怎么造? Watermark 前言

自从用了Spring Boot，个人最喜欢的就是Spring Boot的配置文件了，和Spring比起，Spring Boot更加灵活，修改的某些配置也是更加得心应手。

Spring Boot 官方提供了两种常用的配置文件格式，分别是 `properties`、`YML` 格式。相比于 `properties` 来说，`YML` 更加年轻，层级也是更加分明。

今天这篇文章就来介绍一下Spring Boot的配置文件的语法以及如何从配置文件中取值。

properties格式简介

常见的一种配置文件格式，Spring中也是用这种格式，语法结构很简单，结构为：`key=value`。具体如下：

```
userinfo.name=myjszl
userinfo.age=25
userinfo.active=true
userinfo.created-date=2018/03/31 16:54:30
userinfo.map.k1=v1
userinfo.map.k2=v2
```

上述配置文件中对应的实体类如下：

```
@Data
@Override
public class UserInfo {
    private String name;
    private Integer age;
    private Boolean active;
    private Map<String, Object> map;
    private Date createdDate;
    private List<String> hobbies;
}
```

结构很简单，无非就是 `key=value` 这种形式，也是在开发中用的比较多的一种格式。

YML格式简介

以空格的缩进程度来控制层级关系。空格的个数并不重要，只要左边空格对齐则视为同一个层级。注意不能用 `tab` 代替空格。且大小写敏感。支持字面值，对象，数组三种数据结构，也支持复合结构。

字面值：字符串，布尔类型，数值，日期。字符串默认不加引号，单引号会转义特殊字符。日期格式支持 `yyyy/MM/dd HH:mm:ss`

对象：由键值对组成，形如 `key:(空格)value` 的数据组成。冒号后面的空格是必须要有的，每组键值对占用一行，且缩进的程度要一致，也可以使用行内写法： `{k1: v1, ..., kn: vn}`

数组：由形如 `- (空格)value` 的数据组成。短横线后面的空格是必须要有的，每组数据占用一行，且缩进的程度要一致，也可以使用行内写法： `[1, 2, ..., n]`

复合结构：上面三种数据结构任意组合

如何使用

在 `src/resources` 文件夹下创建一个 `application.yml` 文件。支持的类型主要有字符串，带特殊字符的字符串，布尔类型，数值，集合，行内集合，行内对象，集合对象这几种常用的数据格式。

具体的示例如下：

```
userinfo:  
  age: 25  
  name: myjszl  
  active: true  
  created-date: 2018/03/31 16:54:30  
  map: {k1: v1, k2: v2}  
  hobbies:  
    - one  
    - two  
    - three
```

上述配置文件对应的实体类如下：

```
@Data  
@ToString  
public class UserInfo {  
    private String name;  
    private Integer age;  
    private Boolean active;  
    private Map<String, Object> map;  
    private Date createdDate;  
    private List<String> hobbies;  
}
```

总结

YML是一种新式的格式，层级鲜明，个人比较喜欢使用的一种格式，注意如下：

1. 字符串可以不加引号，若加双引号则输出特殊字符，若不加或加单引号则转义特殊字符
2. 数组类型，短横线后面要有空格；对象类型，冒号后面要有空格
3. YAML是以空格缩进的程度来控制层级关系，但不能用tab键代替空格，大小写敏感

如何从配置文件取值？

一切的配置都是为了取值，Spring Boot也是提供了几种取值的方式，下面一一介绍。

@ConfigurationProperties

这个注解用于从配置文件中取值，支持复杂的数据类型，但是不支持 `SPEL` 表达式。

该注解中有一个属性 `prefix`，用于指定获配置的前缀，毕竟配置文件中的属性很多，也有很多重名的，必须用一个前缀来区分下。

该注解可以表示在类上也可以表示在方法上，这也注定了它有两种获取值的方式。

1. 标注在实体类上

这种方式用于从实体类上取值，并且赋值到对应的属性。使用如下：

```
/**  
 * @Component : 注入到IOC容器中  
 * @ConfigurationProperties: 从配置文件中读取文件  
 */  
@Component  
@ConfigurationProperties(prefix = "userinfo")  
@Data  
@ToString  
public class UserInfo {  
    private String name;  
    private Integer age;  
    private Boolean active;  
    private Map<String, Object> map;  
    private Date createdDate;  
    private List<String> hobbies;  
}
```

标注在配置类中的方法上

标注在配置类上的方法上，同样是从配置文件中取值赋值到返回值的属性中。使用如下：

```
/**  
 * @Bean : 将返回的结果注入到IOC容器中  
 * @ConfigurationProperties : 从配置文件中取值  
 * @return  
 */  
@ConfigurationProperties(prefix = "userinfo")  
@Bean  
public UserInfo userInfo(){  
    return new UserInfo();  
}
```

总结

`@ConfigurationProperties` 注解能够很轻松的从配置文件中取值，优点如下：

1. 支持批量的注入属性，只需要指定一个前缀 `prefix`
2. 支持复杂的数据类型，比如 `List`、`Map`
3. 对属性名匹配的要求较低，比如 `user-name`，`user_name`，`userName`，`USER_NAME` 都可以取值
4. 支持JAVA的JSR303数据校验

注意： `@ConfigurationProperties` 这个注解仅仅是支持从Spring Boot的默认配置文件中取值，比如 `application.properties`、`application.yml`。

@Value

`@Value` 这个注解估计很熟悉了，Spring中从属性取值的注解，支持 `SPEL` 表达式，不支持复杂的数据类型，比如 `List`、`Map`。

```
@Value("${userinfo.name}")  
private String UserName;
```

如何从自定义配置文件中取值？

Spring Boot在启动的时候会自动加载 `application.xxx` 和 `bootstrap.xxx`，但是为了区分，有时候需要自定义一个配置文件，那么如何从自定义的配置文件中取值呢？此时就需要配合 `@PropertySource` 这个注解使用了。

只需要在配置类上标注 `@PropertySource` 并指定你自定义的配置文件即可完成。如下：

```
@SpringBootApplication  
@PropertySource(value = {"classpath:custom.properties"})  
public class DemoApplication {
```

`value` 属性是一个数组，可以指定多个配置文件同时引入。

`@PropertySource` 默认加载 `xxx.properties` 类型的配置文件，不能加载 `YML` 格式的配置文件，怎么破？？？

如何加载自定义YML格式的配置文件？

`@PropertySource` 注解有一个属性 `factory`，默认值是 `PropertySourceFactory.class`，这个就是用来加载 `properties` 格式的配置文件，我们可以自定义一个用来加载 `YML` 格式的配置文件，如下：

```
import org.springframework.beans.factory.config.YamlPropertiesFactoryBean;  
import org.springframework.core.env.PropertiesPropertySource;  
import org.springframework.core.env.PropertySource;  
import org.springframework.core.io.support.DefaultPropertySourceFactory;  
import org.springframework.core.io.support.EncodedResource;  
  
import java.io.IOException;  
import java.util.Properties;  
  
public class YmlConfigFactory extends DefaultPropertySourceFactory {  
    @Override  
    public PropertySource<?> createPropertySource(String name, EncodedResource resource) throws  
    IOException {  
        String sourceName = name != null ? name : resource.getResource().getFilename();  
        if (!resource.getResource().exists()) {  
            return new PropertiesPropertySource(sourceName, new Properties());  
        } else if (sourceName.endsWith(".yml") || sourceName.endsWith(".yaml")) {  
            Properties propertiesFromYaml = loadYml(resource);  
            return new PropertiesPropertySource(sourceName, propertiesFromYaml);  
        } else {  
            return super.createPropertySource(name, resource);  
        }  
    }  
  
    private Properties loadYml(EncodedResource resource) throws IOException {  
        YamlPropertiesFactoryBean factory = new YamlPropertiesFactoryBean();  
        factory.setResources(resource.getResource());  
        factory.afterPropertiesSet();  
        return factory.getObject();  
    }  
}
```

Watermark

此时只需要将 `factory` 属性指定为 `YmlConfigFactory` 即可，如下：

```
@SpringBootApplication  
 @PropertySource(value = {"classpath:custom.yml"}, factory = YmlConfigFactory.class)  
 public class DemoApplication {
```

总结

`@PropertySource` 指定加载自定义的配置文件，默认只能加载 `properties` 格式，但是可以指定 `factory` 属性来加载 `YML` 格式的配置文件。

总结

以上内容介绍了Spring Boot中的配置文件的语法以及如何从配置文件中取值，这个内容很重要，作者也是尽可能讲的通俗易懂，希望读者能够有所收获。

一文带你搞懂日志如何配置？

前言

日志通常不会在需求阶段作为一个功能单独提出来，也不会在产品方案中看到它的细节。但是，这丝毫不影响它在任何一个系统中的重要的地位。

今天就来介绍一下Spring Boot中的日志如何配置。

Spring Boot 版本

本文基于的Spring Boot的版本是 `2.3.4.RELEASE`。

日志级别

几种常见的日志级别由低到高分为：`TRACE < DEBUG < INFO < WARN < ERROR < FATAL`。

如何理解这个日志级别呢？很简单，如果项目中的日志级别设置为 `INFO`，那么比它更低级别的日志信息就看不到了，即是 `TRACE`、`DEBUG` 日志将会不显示。

日志框架有哪些？

常见的日志框架有 `log4j`、`logback`、`log4j2`，

`log4j` 这个日志框架显示是耳熟能详了，在Spring开发中是经常使用，但是据说log4j官方已经不再更新了，而且在性能上比 `logback`、`log4j2` 差了很多。

`logback` 是由 `log4j` 创始人设计的另外一个开源日志框架，`logback` 相比于 `log4j` 性能提升了 10 以上，初始化内存加载也更小了。作为的 Spring Boot 默认的日志框架肯定是有不小的优势。

`log4j2` 晚于 `logback` 推出，官网介绍性能比 `logback` 高，但谁知道是不是王婆卖瓜自卖自夸，坊间流传，`log4j2` 在很多思想理念上都是照抄 `logback`，因此即便 `log4j2` 是 Apache 官方项目，Spring 等许多框架项目没有将它纳入主流。**此处完全是作者道听途说，不必当真，题外话而已。**

日志框架很多，究竟如何选择能够适应现在的项目开发，当然不是普通程序员考虑的，但是为了更高的追求，至少应该了解一下，哈哈。

Spring Boot 日志框架

Spring Boot 默认的日志框架是 `logback`，既然 Spring Boot 能够将其纳入的默认的日志系统，肯定是有一定的考量的，因此实际开发过程中还是不要更换。

原则上需要使用 `logback`，需要添加以下依赖，但是既然是默认的日志框架，当然不用重新引入依赖了。

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

Spring Boot 中默认的日志级别是 `INFO`，启动项目日志打印如下：

```
2020-09-28 13:03:30.937  INFO 13500 --- [           main] com.example.demo.DemoApplication      : Starting Demo
2020-09-28 13:03:30.948  INFO 13500 --- [           main] com.example.demo.DemoApplication      : No active profile found, falling back to default profiles: DEFAULT
2020-09-28 13:03:32.848  INFO 13500 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized
2020-09-28 13:03:32.864  INFO 13500 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-09-28 13:03:32.864  INFO 13500 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: [Tomcat]
2020-09-28 13:03:33.082  INFO 13500 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Servlets
2020-09-28 13:03:33.082  INFO 13500 --- [           main] w.s.c.ServletWebServerApplicationContext : Root Web Application Context: started in 0.019 seconds (refreshing)
2020-09-28 13:03:33.350  INFO 13500 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService [tomcat-executor]
2020-09-28 13:03:33.579  INFO 13500 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s) 8080 (http) with context path ''
2020-09-28 13:03:33.615  INFO 13500 --- [           main] com.example.demo.DemoApplication      : Started Demo in 0.029 seconds (refreshing)
```

从上图可以看出，输出的日志的默认元素如下：

1. 时间日期：精确到毫秒
2. 日志级别：ERROR, WARN, INFO, DEBUG , TRACE
3. 进程ID
4. 分隔符：— 标识实际日志的开始
5. 线程名：方括号括起来（可能会截断控制台输出）
6. Logger名：通常使用源代码的类名
7. 日志内容

代码中如何使用日志？

在业务中肯定需要追溯日志，那么如何在自己的业务中输出日志呢？其实常用的有两种方式，下面一一介绍。

第一种其实也是很早之前常用的一种方式，只需要在代码添加如下：

```
private final Logger logger = LoggerFactory.getLogger(DemoApplicationTests.class);
```

这种方式虽然比较通用，但如果每个类中都调用一下岂不是很low。别着急，lombok为我们解决了这个难题。

要想使用lombok，需要添加如下依赖：

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

使用也很简单，只需要在类上标注一个注解 `@Slf4j` 即可，如下：

```
@Slf4j
class DemoApplicationTests {
    @Test
    public void test() {
        log.debug("输出DEBUG日志.....");
    }
}
```

如何定制日志级别？

Spring Boot中默认的日志级别是INFO，但是可以自己定制日志级别，如下：

```
logging.level.root=DEBUG
```

上面是将所有的日志的级别都改成了 `DEBUG`，Spring Boot还支持 `package` 级别的日志级别调整，格式为：`logging.level.xxx=xxx`，如下：

```
logging.level.com.example.demo=INFO
```

那么完整的配置如下：

```
logging.level.root=DEBUG
logging.level.com.example.demo=INFO
```

日志如何输出到文件中？

Spring Boot中日志默认是输出到控制台的，但是在生产环境中显示不可行的，因此需要配置日志输出到日志文件中。

其中有两个重要配置如下：

1. `logging.file.path`：指定日志文件的路径
2. `logging.file.name`：日志的文件名，默认为 `spring.log`

注意：官方文档说这两个属性不能同时配置，否则不生效，因此只需要配置一个即可。

指定输出的文件为当前项目路径的 `logs` 文件夹下，默认生成的日志文件为 `spring.log`，如下：

```
logging.file.path=./logs
```

日志文件中还有一些其他的属性，比如日志文件的最大size，保留几天的日志等等，下面会介绍到。

如何定制日志格式？

默认的日志格式在第一张图已经看到了，有时我们需要定制自己需要的日志输出格式，这样在排查日志的时候能够一目了然。

定制日志格式有两个配置，分别是控制台的输出格式和文件中的日志输出格式，如下：

1. `logging.pattern.console`：控制台的输出格式
2. `logging.pattern.file`：日志文件的输出格式

例如配置如下：

```
logging.pattern.console=%d{yyyy/MM/dd HH:mm:ss} [%thread] %-5level %logger - %msg%n
logging.pattern.file=%d{yyyy/MM/dd HH:mm} [%thread] %-5level %logger - %msg%n
```

上面的配置编码的含义如下：

%d{HH:mm:ss, SSS}——日志输出时间

%thread——输出日志的进程名字，这在Web应用以及异步任务处理中很有用

%-5level——日志级别，并且使用5个字符靠左对齐

%logger——日志输出者的名字

%msg——日志消息

%n——平台的换行符

如何自定义日志配置？

Spring Boot官方文档指出，根据不同的日志系统，可以按照如下的日志配置文件名就能够被正确加载，如下：

1. Logback : logback-spring.xml, logback-spring.groovy, logback.xml, logback.groovy
2. Log4j : log4j-spring.properties, log4j-spring.xml, log4j.properties, log4j.xml
3. Log4j2 : log4j2-spring.xml, log4j2.xml
4. JDK (Java Util Logging) : logging.properties

Spring Boot官方推荐优先使用带有-spring的文件名作为你的日志配置。因此只需要在 `src/resources` 文件夹下创建 `logback-spring.xml` 即可，配置文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="60 seconds" debug="false">
    <!-- 定义日志存放目录 -->
    <property name="logPath" value="logs"/>
    <!-- 日志输出的格式-->
    <property name="PATTERN" value="%d{yyyy-MM-dd HH:mm:ss, SSS} [%t-%L] %-5level %logger{36} %L %M - %msg%xEx%n"/>
    <contextName>logback</contextName>

    <!--输出到控制台 ConsoleAppender-->
    <appender name="consoleLog" class="ch.qos.logback.core.ConsoleAppender">
        <!--展示格式 layout-->
        <layout class="ch.qos.logback.classic.PatternLayout">
            <pattern>${PATTERN}</pattern>
        </layout>
        <!--过滤器，只有过滤到指定级别的日志信息才会输出，如果level为ERROR，那么控制台只会输出
        ERROR日志-->
        <!-- filter less than or equal to level -->
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">-->
        <!-- <level>ERRR</level> -->
        <!-- </filter>-->
    </appender>

    <!--正常的日志文件，输出到文件中-->
```

```

<appender name="fileDEBUGLog" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <!--如果只是想要 Info 级别的日志，只是过滤 info 还是会输出 Error 日志，因为 Error 的级别高，所以我们使用下面的策略，可以避免输出 Error 的日志-->
    <filter class="ch.qos.logback.classic.filter.LevelFilter">
        <!--过滤 Error-->
        <level>Error</level>
        <!--匹配到就禁止-->
        <onMatch>DENY</onMatch>
        <!--没有匹配到就允许-->
        <onMismatch>ACCEPT</onMismatch>
    </filter>

    <!--日志名称，如果没有File 属性，那么只会使用FileNamePattern的文件路径规则
        如果同时有<File>和<FileNamePattern>，那么当天日志是<File>，明天会自动把今天
        的日志改名为今天的日期。即，<File> 的日志都是当天的。
        -->
    <File>${logPath}/log_demo.log</File>
    <!--滚动策略，按照时间滚动 TimeBasedRollingPolicy-->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <!--文件路径, 定义了日志的切分方式——把每一天的日志归档到一个文件中, 以防止日志填满整个
        磁盘空间-->
        <FileNamePattern>${logPath}/log_demo_%d{yyyy-MM-dd}.log</FileNamePattern>
        <!--只保留最近90天的日志-->
        <maxHistory>90</maxHistory>
        <!--用来指定日志文件的上限大小，那么到了这个值，就会删除旧的日志-->
        <!--<totalSizeCap>1GB</totalSizeCap>-->
    </rollingPolicy>
    <!--日志输出编码格式化-->
    <encoder>
        <charset>UTF-8</charset>
        <pattern>${PATTERN}</pattern>
    </encoder>
</appender>

<!--输出ERROR日志到指定的文件中-->
<appender name="fileErrorLog" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <!--如果只是想要 Error 级别的日志，那么需要过滤一下，默认是 info 级别的，ThresholdFilter-->
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
        <level>Error</level>
    </filter>
    <!--日志名称，如果没有File 属性，那么只会使用FileNamePattern的文件路径规则
        如果同时有<File>和<FileNamePattern>，那么当天日志是<File>，明天会自动把今天
        的日志改名为今天的日期。即，<File> 的日志都是当天的。
        -->
    <File>${logPath}/error.log</File>
    <!--滚动策略，按照时间滚动 TimeBasedRollingPolicy-->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <!--文件路径, 定义了日志的切分方式——把每一天的日志归档到一个文件中, 以防止日志填满整个
        磁盘空间-->
        <FileNamePattern>${logPath}/error_%d{yyyy-MM-dd}.log</FileNamePattern>
        <!--只保留最近90天的日志-->
        <maxHistory>90</maxHistory>
        <!--用来指定日志文件的上限大小，那么到了这个值，就会删除旧的日志-->
        <!--<totalSizeCap>1GB</totalSizeCap>-->
    </rollingPolicy>
    <!--日志输出编码格式化-->
    <encoder>
        <charset>UTF-8</charset>

```

```

        <pattern>${PATTERN}</pattern>
    </encoder>
</appender>

<!--指定最基础的日志输出级别--&gt;
&lt;root level="DEBUG"&gt;
    &lt;!--appender将会添加到这个logger--&gt;
    &lt;appender-ref ref="consoleLog"/&gt;
    &lt;appender-ref ref="fileDEBUGLog"/&gt;
    &lt;appender-ref ref="fileErrorLog"/&gt;
&lt;/root&gt;

<!--      定义指定package的日志级别--&gt;
&lt;logger name="org.springframework" level="DEBUG"&gt;&lt;/logger&gt;
&lt;logger name="org.mybatis" level="DEBUG"&gt;&lt;/logger&gt;
&lt;logger name="java.sql.Connection" level="DEBUG"&gt;&lt;/logger&gt;
&lt;logger name="java.sql.Statement" level="DEBUG"&gt;&lt;/logger&gt;
&lt;logger name="java.sql.PreparedStatement" level="DEBUG"&gt;&lt;/logger&gt;
&lt;logger name="io.lettuce.*" level="INFO"&gt;&lt;/logger&gt;
&lt;logger name="io.netty.*" level="ERROR"&gt;&lt;/logger&gt;
&lt;logger name="com.rabbitmq.*" level="DEBUG"&gt;&lt;/logger&gt;
&lt;logger name="org.springframework.amqp.*" level="DEBUG"&gt;&lt;/logger&gt;
&lt;logger name="org.springframework.scheduling.*" level="DEBUG"&gt;&lt;/logger&gt;
&lt;!--定义com.xxx..xx..xx包下的日志信息不上传，直接输出到fileDEBUGLog和fileErrorLog这两个appender中，日志级别为DEBUG--&gt;
&lt;logger name="com.xxx.xxx.xx" additivity="false" level="DEBUG"&gt;
    &lt;appender-ref ref="fileDEBUGLog"/&gt;
    &lt;appender-ref ref="fileErrorLog"/&gt;
&lt;/logger&gt;

&lt;/configuration&gt;
</pre>

```

当然，如果就不想用Spring Boot推荐的名字，想自己定制也行，只需要在配置文件中指定配置文件名即可，如下：

```
logging.config=classpath:logging-config.xml
```

懵逼了，一堆配置什么意思？别着急，下面一一介绍。

configuration节点

这是一个根节点，其中的各个属性如下：

1. **scan**：当此属性设置为true时，配置文件如果发生改变，将会被重新加载，默认值为true。
2. **scanPeriod**：设置监测配置文件是否有修改的时间间隔，如果没有给出时间单位，默认单位是毫秒。当scan为true时，此属性生效。默认的时间间隔为1分钟。
3. **debug**：当此属性设置为true时，将打印出logback内部日志信息，实时查看logback运行状态。默认值为false。

Watermark

root节点

这是一个必须节点，用来指定基础的日志级别，只有一个 `level` 属性，默认值是 `DEBUG`。
该节点可以包含零个或者多个元素，子节点是 `appender-ref`，标记这个 `appender` 将会添加到这个logger 中。

contextName节点

标识一个上下文名称，默認為default，一般用不到

property节点

标记一个上下文变量，属性有name和value，定义变量之后可以使用 `{}$` 来获取。

appender节点

用来格式化日志输出节点，有两个属性 `name` 和 `class`，`class`用来指定哪种输出策略，常用就是[控制台输出策略](#)和[文件输出策略](#)。

这个节点很重要，通常的日志文件需要定义三个appender，分别是控制台输出，常规日志文件输出，异常日志文件输出。

该节点有几个重要的子节点，如下：

1. `filter`：日志输出拦截器，没有特殊定制一般使用系统自带的即可，但是如果要将日志分开，比如将ERROR级别的日志输出到一个文件中，将除了 `ERROR` 级别的日志输出到另外一个文件中，此时就要拦截 `ERROR` 级别的日志了。
2. `encoder`：和pattern节点组合用于具体输出的日志格式和编码方式。
3. `file`：节点用来指明日志文件的输出位置，可以是绝对路径也可以是相对路径
4. `rollingPolicy`：日志回滚策略，在这里我们用了`TimeBasedRollingPolicy`，基于时间的回滚策略，有以下子节点`fileNamePattern`，必要节点，可以用来设置指定时间的日志归档。
5. `maxHistory`：可选节点，控制保留的归档文件的最大数量，超出数量就删除旧文件，例如设置为30的话，则30天之后，旧的日志就会被删除
6. `totalSizeCap`：可选节点，用来指定日志文件的上限大小，例如设置为3GB的话，那么到了这个值，就会删除旧的日志

logger节点

可选节点，用来具体指明包的日志输出级别，它将会覆盖root的输出级别。

该节点有几个重要的属性如下：

1. `name`：指定的包名
2. `level`：可选，日志的级别
3. `additivity`：可选，默認為true，将此logger的信息向上级传递，将有root节点定义日志打印。如果设置为false，将不会上传，此时需要定义一个 `appender-ref` 节点才会输出。

总结

Watermark

Spring Boot的日志选型以及如何自定义日志配置就介绍到这里，如果觉得有所收获，不妨点个关注，分享一波，将是对作者最大的鼓励！！！

一文带你搞懂日志框架如何切换？

前言

首先要感谢一下读者朋友们的支持，你们每一个的赞都是对陈某最大的肯定，陈某也会一如既往的加油，奥利给！！！

留给中国队的时间不...
作为java初学者，spring boot的初学者，很是受用，感谢分享
17小时前

不才陈某 Lv2 (作者) 公众号: 码猿技术专栏 @ ...
多谢肯定，持续更新中 🤝
5小时前

言归正传，上一篇文章写了Spring Boot的默认日志框架Logback的基本配置，有兴趣的可以看看：
[Spring Boot第三弹，一文带你搞懂日志如何配置？](#)。

今天就来介绍一下Spring Boot如何无感的切换日志框架？

Spring Boot 版本

本文基于的Spring Boot的版本是 [2.3.4.RELEASE](#)。

什么是日志门面？

前面介绍的日志框架都是基于日志门面 [SLF4j](#) 即简单日志门面（Simple Logging Facade for Java），
SLF4j并不是一个真正的日志实现，而是一个抽象层，它允许你在后台使用任意一个日志实现。

使用了slf4j后，对于应用程序来说，无论底层的日志框架如何变，应用程序不需要修改任意一行代码，
就可以直接上线了。

如果对SLF4j比较感兴趣的可以去官网看看：[SLF4j官网](#)

如何做到无感知切换？

SLF4j是日志门面，无论什么日志框架都是基于SLF4j的API实现，因此无论是代码打印日志还是
Lombok注解形式打印日志，都要使用的SLF4j的API，而不是日志框架的API，这样才能解耦，做到无
感知。因为最终切换的框架只是对于SLF4j的实现，并不是切换SLF4j。

其实这一条在阿里开发手册中也是明确指出了，如下：

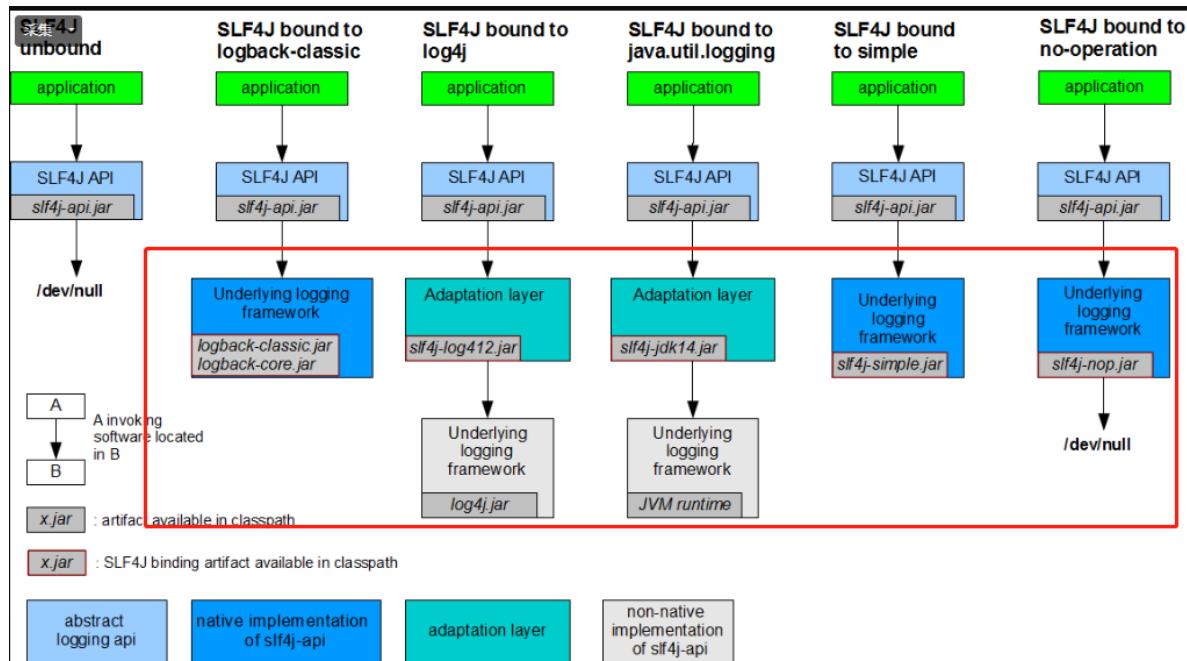
1. 【强制】应用中不可直接使用日志系统（Log4j、Logback）中的 API，而应依赖使用日志框架 SLF4J 中的 API，使用门面模式的日志框架，有利于维护和各个类的日志处理方式统一。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
private static final Logger logger = LoggerFactory.getLogger(Abc.class);
```

如何切换？

Spring Boot默认是 Logback 日志框架，如果需要切换到其他的日志框架应该如何做？

首先我们先看官网的一张图，一切都在图中，如下：



SLF4j只是一个门面，共有两大特性。一是静态绑定、二是桥接。

什么是静态绑定？：我们以 log4j 为例。首先我们的application中会使用slf4j的api进行日志记录。我们引入适配层的jar包 slf4j-log4j12.jar 及底层日志框架实现 log4j.jar 。简单的说适配层做的事情就是把 slf4j 的api转化成 log4j 的api。通过这样的方式来屏蔽底层框架实现细节。

什么是桥接？：比如你的application中使用了 slf4j，并绑定了 logback。但是项目中引入了一个 A.jar，A.jar 使用的日志框架是 log4j。那么有没有方法让 slf4j 来接管这个 A.jar 包中使用 log4j 输出的日志呢？这就用到了桥接包。你只需要引入 log4j-over-slf4j.jar 并删除 log4j.jar 就可以实现 slf4j 对 A.jar 中 log4j 的接管。听起来有些不可思议。你可能会想如果删除 log4j.jar 那 A.jar 不会报编译错误嘛？答案是不会。因为 log4j-over-slf4j.jar 实现了 log4j 几乎所有 public 的 API。但关键方法都被改写了。不再是简单的输出日志，而是将日志输出指令委托给 slf4j。

下面就以 log4j2 为例，切换Spring Boot的日志框架为 Log4j2 。

引入依赖

Spring Boot 默认使用 Logback 日志框架，如果想要切换 Log4j2 肯定是要将 Logback 的依赖移除，只需要排除 web 模块即可。启动器即可，如下：

```

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <!-- 去掉springboot默认日志框架logback -->
        <exclusions>
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>

```

排除了默认的logback依赖，肯定是需要引入 log4j2 的依赖，其实 log4j2 为了与 Spring Boot 适配也做了个启动器，不需要在引入其他的jar包了，只需要添加如下依赖即可：

```

<!-- 引入log4j2依赖 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>

```

指定配置文件

Spring Boot 官方文档已经给出了默认两个log4j2的配置的名称，分别为： `log4j2-spring.xml`，`log4j2.xml`，但是建议使用 `log4j2-spring.xml`，因为 Spring Boot 会做一些扩展，行吧，就整这个放在 `src/resources` 文件夹下即可。

另外上篇文章也说过，如果不使用默认的配置名称，则需要在 `application.properties` 指定配置文件，如下：

```
logging.config=classpath:logging-config.xml
```

日志如何配置？

其实 log4j2 的一些配置和logback很相似，这里就不再一一介绍，有兴趣的可以去官网查查，直接贴出一些即用的配置，如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!--Configuration后面的status，这个用于设置log4j2自身内部的信息输出，可以不设置，当设置成trace时，你会看到log4j2内部各种详细输出-->
<!--monitorInterval: Log4j能够自动检测修改配置 文件和重新配置本身，设置间隔秒数-->
<configuration monitorInterval="5">
    <!--日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL -->

    <!--变量配置-->
    <Properties>
        <!-- 格式化输出: %date表示日期, %thread表示线程名, %-5level: 级别从左显示5个字符宽度 %msg: 日志消息, %n是换行符-->
        <!-- %logger{36} 表示 Logger 名字最长36个字符 -->
        <property name="LOG_PATTERN" value="%date{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n" />
        <!-- 定义日志存储的路径 -->
        <property name="FILE_PATH" value="更换为你的日志路径" />
        <property name="FILE_NAME" value="更换为你的项目名" />
    </Properties>

```

```
<appenders>
```

```
    <console name="Console" target="SYSTEM_OUT">
        <!--输出日志的格式-->
        <PatternLayout pattern="${LOG_PATTERN}" />
        <!--控制台只输出level及其以上级别的信息（onMatch），其他的直接拒绝（onMismatch）-->
        <ThresholdFilter level="info" onMatch="ACCEPT" onMismatch="DENY"/>
    </console>

    <!--文件会打印出所有信息，这个log每次运行程序会自动清空，由append属性决定，适合临时测试用-->
    <File name="Filelog" fileName="${FILE_PATH}/test.log" append="false">
        <PatternLayout pattern="${LOG_PATTERN}" />
    </File>

    <!-- 这个会打印出所有的info及以下级别的信息，每次大小超过size，则这size大小的日志会自动存入按年份-月份建立的文件夹下面并进行压缩，作为存档-->
    <RollingFile name="RollingFileInfo" fileName="${FILE_PATH}/info.log"
filePattern="${FILE_PATH}/${FILE_NAME}-INFO-%d{yyyy-MM-dd}_%i.log.gz">
        <!--控制台只输出level及以上级别的信息（onMatch），其他的直接拒绝（onMismatch）-->
        <ThresholdFilter level="info" onMatch="ACCEPT" onMismatch="DENY"/>
        <PatternLayout pattern="${LOG_PATTERN}" />
        <Policies>
            <!--interval属性用来指定多久滚动一次，默认是1 hour-->
            <TimeBasedTriggeringPolicy interval="1"/>
            <SizeBasedTriggeringPolicy size="10MB"/>
        </Policies>
        <!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7个文件开始覆盖-->
        <DefaultRolloverStrategy max="15"/>
    </RollingFile>

    <!-- 这个会打印出所有的warn及以下级别的信息，每次大小超过size，则这size大小的日志会自动存入按年份-月份建立的文件夹下面并进行压缩，作为存档-->
    <RollingFile name="RollingFileWarn" fileName="${FILE_PATH}/warn.log"
filePattern="${FILE_PATH}/${FILE_NAME}-WARN-%d{yyyy-MM-dd}_%i.log.gz">
        <!--控制台只输出level及以上级别的信息（onMatch），其他的直接拒绝（onMismatch）-->
        <ThresholdFilter level="warn" onMatch="ACCEPT" onMismatch="DENY"/>
        <PatternLayout pattern="${LOG_PATTERN}" />
        <Policies>
            <!--interval属性用来指定多久滚动一次，默认是1 hour-->
            <TimeBasedTriggeringPolicy interval="1"/>
            <SizeBasedTriggeringPolicy size="10MB"/>
        </Policies>
        <!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7个文件开始覆盖-->
        <DefaultRolloverStrategy max="15"/>
    </RollingFile>

    <!-- 这个会打印出所有的error及以下级别的信息，每次大小超过size，则这size大小的日志会自动存入按年份-月份建立的文件夹下面并进行压缩，作为存档-->
    <RollingFile name="RollingFileError" fileName="${FILE_PATH}/error.log"
filePattern="${FILE_PATH}/${FILE_NAME}-ERROR-%d{yyyy-MM-dd}_%i.log.gz">
        <!--控制台只输出level及以上级别的信息（onMatch），其他的直接拒绝（onMismatch）-->
        <ThresholdFilter level="error" onMatch="ACCEPT" onMismatch="DENY"/>
        <PatternLayout pattern="${LOG_PATTERN}" />
        <Policies>
            <!--interval属性用来指定多久滚动一次，默认是1 hour-->
            <TimeBasedTriggeringPolicy interval="1"/>
            <SizeBasedTriggeringPolicy size="10MB"/>
        </Policies>

```

```

        </Policies>
        <!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7个文件开始覆盖-->
        <DefaultRolloverStrategy max="15"/>
    </RollingFile>

</appenders>

<!--Logger节点用来单独指定日志的形式，比如要为指定包下的class指定不同的日志级别等。-->
<!--然后定义loggers，只有定义了logger并引入的appender，appender才会生效-->
<loggers>

    <!--过滤掉spring和mybatis的一些无用的DEBUG信息-->
    <logger name="org.mybatis" level="info" additivity="false">
        <AppenderRef ref="Console"/>
    </logger>
    <!--监控系统信息-->
    <!--若是additivity设为false，则子Logger只会在自己的appender里输出，而不会在父Logger的
    appender里输出。-->
    <Logger name="org.springframework" level="info" additivity="false">
        <AppenderRef ref="Console"/>
    </Logger>

    <root level="info">
        <appender-ref ref="Console"/>
        <appender-ref ref="Filelog"/>
        <appender-ref ref="RollingFileInfo"/>
        <appender-ref ref="RollingFileWarn"/>
        <appender-ref ref="RollingFileError"/>
    </root>
</loggers>
</configuration>

```

上面的配置中如果需要使用的话，需要改掉全局变量中的日志路径和项目名称，如下部分：

```

<property name="FILE_PATH" value="更换为你的日志路径" />
<property name="FILE_NAME" value="更换为你的项目名" />

```

总结

本篇文章介绍了Spring Boot如何切换日志框架以及SLF4j一些内容，如果有所收获点点在看关注分享一波，谢谢！！！

WEB开发初了解~

前言 Watermark

今天是Spring Boot专栏的第五篇文章，相信大家看了前四篇文章对Spring Boot已经有了初步的了解，今天这篇文章就来介绍一下Spring Boot的重要功能WEB开发。

Spring Boot 版本

本文基于的Spring Boot的版本是 2.3.4.RELEASE。

前提条件（必须注意）

Spring Boot的WEB开发有自己的启动器和自动配置，最好采用Spring Boot的一套配置，这里千万不要在任何一个配置类上添加 `@EnableWebMvc` 这个注解，具体原因会单独一篇文章讲述。

此篇文章所有的内容都是在没有标注 `@EnableWebMvc` 这个注解的前提下。

添加依赖

Spring Boot对web模块有一个启动器，只需要在pom.xml中引入即可，如下：

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

这个依赖看似只是引入了一个依赖，其实内部引入了Spring, Spring MVC的相关依赖，Spring Boot的启动器就是这么神奇，后面的文章会介绍启动器的原理和如何自定义启动器。

第一个接口开发

假设这么一个需求，需要根据用户的ID获取用户信息，我们应该如何写接口呢？

其实和Spring MVC开发步骤一样，写一个controller，各种注解骚操作搞起，如下：

```
@RestController
@RequestMapping("/user")
public class UserController {
    @GetMapping("/{id}")
    public Object getById(@PathVariable("id") String id) {
        return User.builder()
            .id(id)
            .name("不才陈某")
            .age(18)
            .birthday(new Date())
            .build();
    }
}
```

这样一个接口就已经完成了，启动项目访问 <http://localhost:8080/user/1> 即可得到如下的结果：

Watermark

```
{
    "id": 1,
    "age": 18,
    "birthday": 1601454650860,
    "name": "不才陈某"
}
```

如何自定义tomcat的端口?

Spring Boot其实默认内嵌了Tomcat，当然默认的口号也是 8080，如果需要修改的话，只需要在配置文件中添加如下一行配置即可：

```
server.port=9090
```

如何自定义项目路径?

在配置文件中添加如下配置即可：

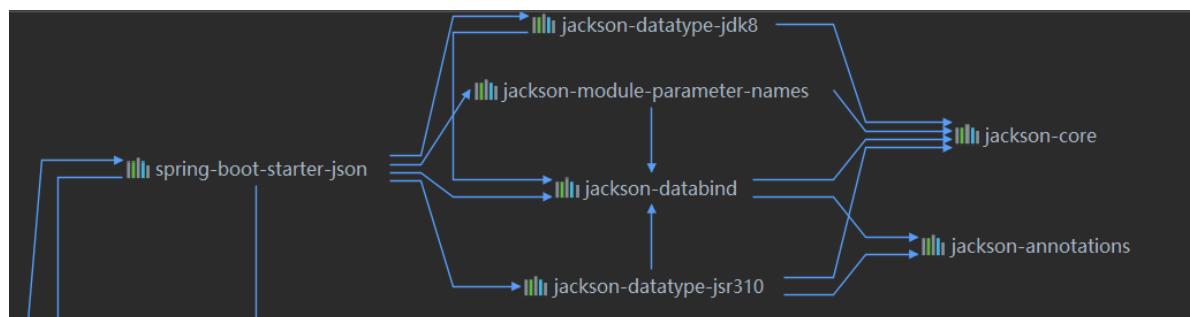
```
server.servlet.context-path=/springboot01
```

以上的端口和项目路径改了之后，只需要访问 <http://localhost:9090/springboot01/user/1> 即可。

JSON格式化

在前后端分离的项目中大部分的接口基本都是返回JSON字符串，因此对返回的JSON也是需要定制一下，比如**日期的格式**，**NULL值是否返回**等等内容。

Spring Boot默认是使用Jackson对返回结果进行处理，在引入WEB启动器的时候会引入相关的依赖，如下图：



同样是引入了一个启动器，则意味着我们既可以在配置文件中修改配置，也可以在配置类中重写其中的配置。Jackson的自动配置类是 [JacksonAutoConfiguration](#)

日期格式的设置

上面的例子中日期的返回结果其实是一个时间戳，那么我们需要返回格式为 `yyyy-MM-dd HH:mm:ss`。

可以在配置文件 `application.properties` 中设置指定的格式，这属于**全局配置**，如下：

```
spring.jackson.date-format= yyyy-MM-dd HH:mm:ss  
spring.jackson.time-zone= GMT+8
```

也可以在实体属性中标注 `@JsonFormat` 这个注解，属于局部配置，会覆盖全局配置，如下：

```
@JsonFormat(pattern = "yyyy-MM-dd HH:mm", timezone = "GMT+8")  
private Date birthday;
```

上述日期格式标注之后返回的就是按指定格式的日期，如下：

```
{  
    "id": "1",  
    "age": 18,  
    "birthday": "2020-09-30 17:21",  
    "name": "不才陈某"  
}
```

其他属性的配置

Jackson还有很多的属性可以配置，这里就不再一一介绍了，所有的配置前缀都是 `spring.jackson`。

如何在配置类配置？

前面说过在引入WEB模块的时候还引入了JackSon的启动器，这是个好东西，这也是Spring Boot的好处之一，自动配置类中所需的一些配置既可以在全局配置文件 `application.properties` 中配置也可以在配置类中重新注入某个Bean而达到修改默认配置的效果。

在JackSon自动配置类 `JacksonAutoConfiguration` 中有如下一段代码：

```
@Bean  
@Primary  
@ConditionalOnMissingBean  
ObjectMapper jacksonObjectMapper(Jackson2ObjectMapperBuilder builder) {  
    return builder.createXmlMapper(false).build();  
}
```

这一段代码可能初学者比较懵逼了，什么意思呢？别着急，`@Bean` 这个注解无非就是注入一个Bean到IOC容器中，`@Primary` 这个注解自不用说了，剩下的就是 `@ConditionalOnMissingBean` 这个注解了，什么意思呢？

其实仔细研究过Spring Boot的源码的朋友都知道，类似这种 `@Conditionalxxx` 的注解还有很多，这里就不再深入讲了，后期的文章会介绍。

`@ConditionalOnMissingBean` 这个注解的意思很简单，就是当IOC容器中没有指定Bean的时候才会注入，言下之意就是当容器中不存在 `ObjectMapper` 这个Bean会使用这里生成的，类似于一种生效的条件。

言外之意就是只需要自定义一个 `ObjectMapper` 然后注入到IOC容器中，那么这个自动配置类 `JacksonAutoConfiguration` 中注入的将会失效，也就达到了覆盖的作用了。

因此只需要定义一个配置类，注入 `ObjectMapper` 即可，如下：

```
/**  
 * 自定义jackson序列化与反序列规则，增加相关格式（全局配置）  
 */  
@Configuration  
public class JacksonConfig {  
    @Bean  
    @Primary  
    public ObjectMapper jacksonObjectMapper(Jackson2ObjectMapperBuilder builder) {  
        builder.locale(Locale.CHINA);  
        builder.timeZone(TimeZone.getTimeZone(ZoneId.systemDefault()));  
        builder.dateFormat(DateTimeFormat.NORM_DATETIME_PATTERN);  
        builder.modules(new CustomTimeModule());  
  
        ObjectMapper objectMapper = builder.createXmlMapper(false).build();  
  
        objectMapper.setSerializationInclusion(JsonInclude.Include.NON_EMPTY);  
    }  
}
```

```
//遇到未知属性的时候抛出异常，//为true 会抛出异常
objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
// 允许出现特殊字符和转义符
objectMapper.configure(JsonParser.Feature.ALLOW_UNQUOTED_CONTROL_CHARS, true);
// 允许出现单引号
objectMapper.configure(JsonParser.Feature.ALLOW_SINGLE_QUOTES, true);

objectMapper.registerModule(new CustomTimeModule());

return objectMapper;
}

}
```

上面只是个例子，关于 `ObjectMapper` 中的一些内容感兴趣的可以自己查查相关资料。

总结

这篇文章算是WEB开发的入门，介绍了如何定义接口，返回JSON如何定制等内容，如果觉得有所收获点点关注在看分享一波！！！

拦截器如何配置，看这儿~

前言

上篇文章讲了Spring Boot的WEB开发基础内容，相信读者朋友们已经有了初步的了解，知道如何写一个接口。

今天这篇文章来介绍一下拦截器在Spring Boot中如何自定义以及配置。

Spring Boot 版本

本文基于的Spring Boot的版本是 `2.3.4.RELEASE`。

什么是拦截器？

Spring MVC中的拦截器（`Interceptor`）类似于Servlet中的过滤器（`Filter`），它主要用于拦截用户请求并作相应的处理。例如通过拦截器可以进行权限验证、记录请求信息的日志、判断用户是否登录等。

如何自定义一个拦截器？

自定义一个拦截器非常简单，只需要实现 `HandlerInterceptor` 这个接口即可，该接口有三个可以实现的方法，如下：

1. `preHandle()` 方法：该方法会在控制器方法前执行，其返回值表示是否知道如何写一个接口。中断后续操作。当其返回值为 `true` 时，表示继续向下执行；当其返回值为 `false` 时，会中断后续的所有操作（包括调用下一个拦截器和控制器类中的方法执行等）。
2. `postHandle()` 方法：该方法会在控制器方法调用之后，且解析视图之前执行。可以通过此方法对请求域中的模型和视图做出进一步的修改。
3. `afterCompletion()` 方法：该方法会在整个请求完成，即视图渲染结束之后执行。可以通过此方法实现一些资源清理、记录日志信息等工作。

如何使其在Spring Boot中生效？

其实想要在Spring Boot生效其实很简单，只需要定义一个配置类，实现 `WebMvcConfigurer` 这个接口，并且实现其中的 `addInterceptors()` 方法即可，代码演示如下：

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Autowired
    private XXX xxx;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //不拦截的uri
        final String[] commonExclude = {};
        registry.addInterceptor(xxx).excludePathPatterns(commonExclude);
    }
}
```

举个栗子

开发中可能会经常遇到短时间内由于用户的重复点击导致几秒之内重复的请求，可能就是在这几秒之内由于各种问题，比如 网络，事务的隔离性 等等问题导致了数据的重复等问题，因此在日常开发中必须规避这类的重复请求操作，今天就用拦截器简单的处理一下这个问题。

思路

在接口执行之前先对指定接口（比如标注某个 `注解` 的接口）进行判断，如果在指定的时间内（比如 5 秒）已经请求过一次了，则返回重复提交的信息给调用者。

Watermark

根据什么判断这个接口已经请求了？

根据项目的架构可能判断的条件也是不同的，比如 IP地址、用户唯一标识、请求参数、请求URI 等等其中的某一个或者多个的组合。

这个具体的信息存放在哪里？

由于是短时间 内甚至是瞬间并且要保证 定时失效，肯定不能存在事务性数据库中了，因此常用的几种数据库中只有 Redis 比较合适了。

如何实现？

第一步，先自定义一个注解，可以标注在类或者方法上，如下：

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface RepeatSubmit {
    /**
     * 默认失效时间5秒
     */
    long seconds() default 5;
}
```

第二步，创建一个拦截器，注入到IOC容器中，实现的思路很简单，判断controller的类或者方法上是否标注了 @RepeatSubmit 这个注解，如果标注了，则拦截判断，否则跳过，代码如下：

```
/**
 * 重复请求的拦截器
 * @Component: 该注解将其注入到IOC容器中
 */
@Component
public class RepeatSubmitInterceptor implements HandlerInterceptor {

    /**
     * Redis的API
     */

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    /**
     * preHandler方法，在controller方法之前执行
     *
     * 判断条件仅仅是用了uri，实际开发中根据实际情况组合一个唯一识别的条件。
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
        throws Exception {
        if (handler instanceof HandlerMethod) {
            //只拦截标注了@RepeatSubmit该注解
            HandlerMethod method=(HandlerMethod) handler;
            //标注在方法上的@RepeatSubmit
            RepeatSubmit repeatSubmitByMethod =
                AnnotationUtils.findAnnotation(method.getMethod(), RepeatSubmit.class);
            //标注在controler类上的@RepeatSubmit
            RepeatSubmit repeatSubmitByCls =
                AnnotationUtils.findAnnotation(method.getMethod().getDeclaringClass(), RepeatSubmit.class);
            //没有限制重复提交，直接跳过
        }
    }
}
```

```

        if (Objects.isNull(repeatSubmitByMethod) && Objects.isNull(repeatSubmitByCls))
            return true;

        // todo: 组合判断条件, 这里仅仅是演示, 实际项目中根据架构组合条件
        //请求的URI
        String uri = request.getRequestURI();

        //存在即返回false, 不存在即返回true
        Boolean ifAbsent = stringRedisTemplate.opsForValue().setIfAbsent(uri, "", 
Objects.nonNull(repeatSubmitByMethod) ? repeatSubmitByMethod.seconds() : repeatSubmitByCls.seconds(),
TimeUnit.SECONDS);

        //如果存在, 表示已经请求过了, 直接抛出异常, 由全局异常进行处理返回指定信息
        if (ifAbsent!=null&&!ifAbsent)
            throw new RepeatSubmitException();
    }
    return true;
}
}

```

第三步，在Spring Boot中配置这个拦截器，代码如下：

```

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Autowired
    private RepeatSubmitInterceptor repeatSubmitInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //不拦截的uri
        final String[] commonExclude = {" /error", " /files/**"};
        registry.addInterceptor(repeatSubmitInterceptor).excludePathPatterns(commonExclude);
    }
}

```

OK，拦截器已经配置完成，只需要在需要拦截的接口上标注 `@RepeatSubmit` 这个注解即可，如下：

```

@RestController
@RequestMapping("/user")
//标注了@RepeatSubmit注解, 全部的接口都需要拦截
@RepeatSubmit
public class LoginController {

    @RequestMapping("/login")
    public String login() {
        return "login success";
    }
}

```

此时，请求这个URI: <http://localhost:8080/springboot-demo/user/login> 在5秒之内只能请求一次。

注意：方法在客户端的压测时间会严重影响服务器上的时间，因为如下一段代码：

```

Boolean ifAbsent = stringRedisTemplate.opsForValue().setIfAbsent(uri, "", 
Objects.nonNull(repeatSubmitByMethod) ? repeatSubmitByMethod.seconds() : repeatSubmitByCls.seconds(),
TimeUnit.SECONDS);

```

这段代码的失效时间先取值 `repeatSubmitByMethod` 中配置的，如果为null，则取值 `repeatSubmitByCls` 配置的。

总结

至此，拦截器的内容就介绍完了，其实配置起来很简单，没什么重要的内容。

上述例子中的 源代码 有需要的朋友公众号 码猿技术专栏 内回复关键词 拦截器 即可获取。



过滤器如何配置，一波梭哈~

前言

上篇文章介绍了Spring Boot中如何配置拦截器，今天这篇文章就来讲讲类似于拦截器的一个组件：过滤器。

其实在实际开发中过滤器真的接触的不多，但是在应用中却是不可或缺的角色，值得花费一个章节专门介绍一下。

Spring Boot 版本

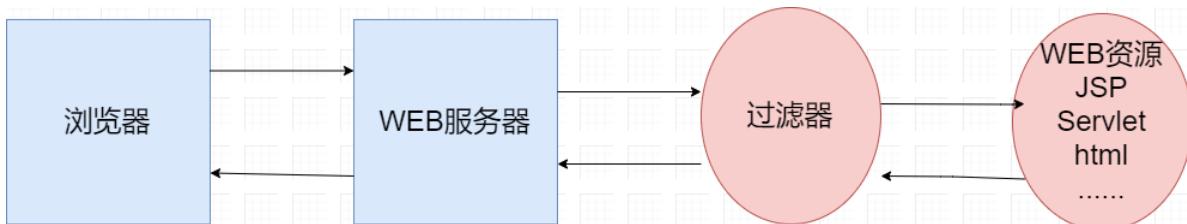
本文基于的Spring Boot的版本是 2.3.4.RELEASE。

什么是过滤器？

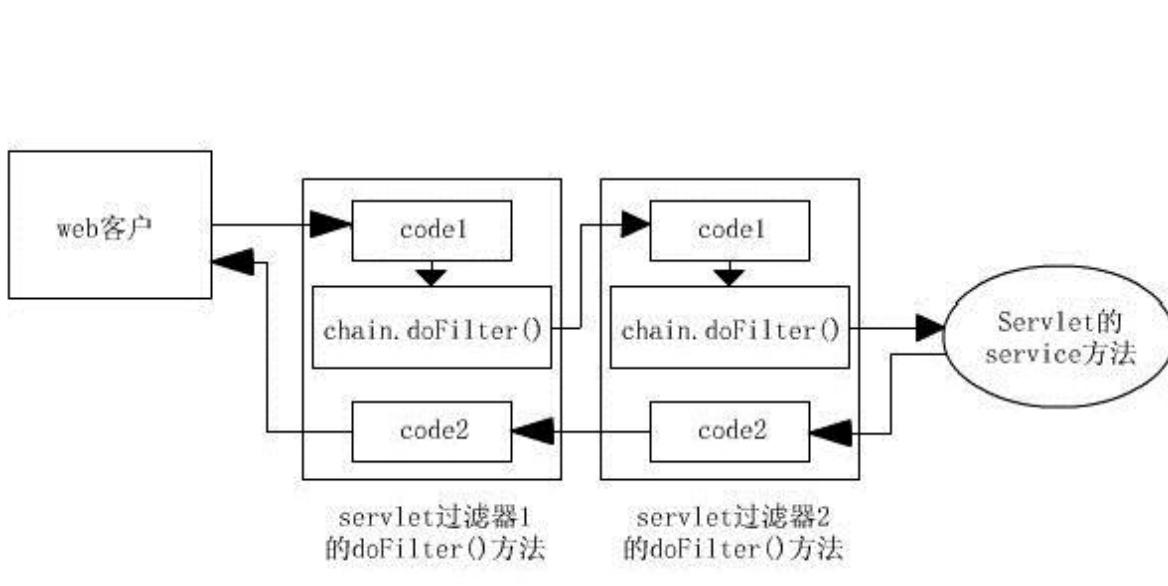
Filter，即之为过滤器，是Servlet技术中最实用的技术，WEB开发人员通过Filter技术，对web服务器管理的所有web资源：例如JSP，Servlet，静态图片文件或静态HTML文件进行拦截，从而实现一些特殊功能。例如实现 URL级别的权限控制、过滤敏感词汇、压缩响应信息等一些高级功能。

Filter的执行原理

当客户端发出Web资源的请求时，Web服务器根据应用程序配置文件设置的过滤规则进行检查，若客户请求满足过滤规则，则对客户请求 / 响应进行拦截，对请求头和请求数据进行检查或改动，并依次通过过滤器链，最后把请求 / 响应交给请求的Web资源处理。请求信息在过滤器链中可以被修改，也可以根据条件让请求不发往资源处理器，并直接向客户机发回一个响应。当资源处理器完成了对资源的处理后，响应信息将逐级逆向返回。同样在这个过程中，用户可以修改响应信息，从而完成一定的任务，如下图：



服务器会按照过滤器定义的先后循序组装成 **一条链**，然后一次执行其中的 `doFilter()` 方法。（注：这一点 **Filter** 和 **Servlet** 是不一样的）执行的顺序就如下图所示，执行第一个过滤器的 `chain.doFilter()` 之前的代码，第二个过滤器的 `chain.doFilter()` 之前的代码，请求的资源，第二个过滤器的 `chain.doFilter()` 之后的代码，第一个过滤器的 `chain.doFilter()` 之后的代码，最后返回响应。



如何自定义一个Filter?

这个问题其实不是Spring Boot这个章节应该介绍的了，在Spring MVC中就应该会的内容，只需要实现 `javax.servlet.Filter` 这个接口，重写其中的方法。实例如下：

```
@Component
public class CrosFilter implements Filter {

    //重写其中的doFilter方法
    @Override
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws
    IOException, ServletException {
        //继续执行下一个过滤器
        chain.doFilter(req, response);
    }
}
```

Spring Boot如何配置Filter?

自定义好了过滤器当然要使其在Spring Boot中生效了，Spring Boot配置Filter有两种方式，其实都很简单，下面一一介绍。

配置类中使用@Bean注入【推荐使用】

其实很简单，只需要将 `FilterRegistrationBean` 这个实例注入到IOC容器中即可，如下：

```
@Configuration
public class FilterConfig {
    @Autowired
    private Filter1 filter1;

    @Autowired
    private Filter2 filter2;

    /**
     * 注入Filter1
     * @return
     */
    @Bean
    public FilterRegistrationBean filter1() {
        FilterRegistrationBean registration = new FilterRegistrationBean();
        registration.setFilter(filter1);
        registration.addUrlPatterns("/*");
        registration.setName("filter1");
        //设置优先级别
        registration.setOrder(1);
        return registration;
    }

    /**
     * 注入Filter2
     * @return
     */
    @Bean
    public FilterRegistrationBean filter2() {
        FilterRegistrationBean registration = new FilterRegistrationBean();
        registration.setFilter(filter2);
        registration.addUrlPatterns("/*");
        registration.setName("filter2");
        //设置优先级别
        registration.setOrder(2);
        return registration;
    }
}
```

注意：设置的优先级别决定了过滤器的执行顺序。

Watermark

使用@WebFilter

`@WebFilter` 是Servlet3.0的一个注解，用于标注一个Filter，Spring Boot也是支持这种方式，只需要在自定义的Filter上标注该注解即可，如下：

```
@WebFilter(filterName = "crosFilter", urlPatterns = {"/*"})
public class CrosFilter implements Filter {

    //重写其中的doFilter方法
    @Override
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws
    IOException, ServletException {
        //继续执行下一个过滤器
        chain.doFilter(req, response);
    }
}
```

要想 `@WebFilter` 注解生效，需要在配置类上标注另外一个注解 `@ServletComponentScan` 用于扫描使其生效，如下：

```
@SpringBootApplication
@ServletComponentScan(value = {"com.example.springbootintercept.filter"})
public class SpringbootApplication {}
```

至此，配置就完成了，启动项目，即可正常运行。

举个栗子

对于前后端分离的项目来说跨域是一个难题，什么是跨域问题？如何造成的？这个不是本章的重点。

对于跨域问题有多中解决方案，比如JSONP，网关支持等等。**关于跨域的问题以及Spring Boot如何优雅的解决跨域问题？将会在后续文章中介绍。**今天主要介绍如何使用过滤器来解决跨域问题。

其实原理很简单，只需要在请求头中添加相应支持跨域的内容即可，如下代码仅仅是简单的演示下，针对细致的内容还需自己完善，比如白名单等等。

```
@Component
public class CrosFilter implements Filter {

    //重写其中的doFilter方法
    @Override
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws
    IOException, ServletException {
        HttpServletResponse response = (HttpServletResponse) res;
        response.setHeader("Access-Control-Allow-Origin", "*");
        response.setHeader("Access-Control-Allow-Credentials", "true");
        response.setHeader("Access-Control-Allow-Methods", "POST, GET, OPTIONS, DELETE");
        response.setHeader("Access-Control-Max-Age", "3600");
        response.setHeader("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type,
        Accept");
        //继续执行下一个过滤器
        chain.doFilter(req, response);
    }
}
```

配置类中注入 `FilterRegistrationBean`，如下代码：

```

@Configuration
public class FilterConfig {
    @Autowired
    private CrosFilter crosFilter;

    /**
     * 注入crosFilter
     * @return
     */
    @Bean
    public FilterRegistrationBean crosFilter() {
        FilterRegistrationBean registration = new FilterRegistrationBean();
        registration.setFilter(crosFilter);
        registration.addUrlPatterns("/*");
        registration.setName("crosFilter");
        //设置优先级别
        registration.setOrder(Ordered.HIGHEST_PRECEDENCE);
        return registration;
    }
}

```

至此，配置完成，相关细致功能还需自己润色。

总结

过滤器内容相对简单些，但是在实际开发中不可或缺，比如常用的权限控制框架 Shiro，Spring Security，内部都是使用过滤器，了解一下对以后的深入学习有着固本的作用。

另外作者的第一本PDF书籍已经整理好了，由浅入深的详细介绍了Mybatis基础以及底层源码，有需要的朋友公众号回复关键词 Mybatis进阶 即可获取，目录如下：



如果有所收获，不妨关注在看支持一下，持续连载中.....



教你Spring Boot如何扩展、接管MVC?

前言

自从用了Spring Boot是否有一个感觉，以前MVC的配置都很少用到了，比如视图解析器，拦截器，过滤器等等，这也正是Spring Boot好处之一。

但是往往Spring Boot提供默认的配置不一定适合实际的需求，因此需要能够定制MVC的相关功能，这篇文章就介绍一下如何扩展和全面接管MVC。

Spring Boot 版本

本文基于的Spring Boot的版本是 2.3.4.RELEASE。

如何扩展MVC?

在这里需要声明一个前提：配置类上没有标注 `@EnableWebMvc` 并且没有任何一个配置类继承了 `WebMvcConfigurerSupport`。至于具体原因，下文会详细解释。

扩展MVC其实很简单，只需要以下步骤：

1. 创建一个MVC的配置类，并且标注 `@Configuration` 注解。
2. 实现 `WebMvcConfigurer` 这个接口，并且实现需要的方法。

`WebMvcConfigurer` 这个接口中定义了MVC相关的各种组件，比如拦截器，视图解析器等等的定制方法，需要定制什么功能，只需要实现即可。

在Spring Boot之前的版本还可以继承一个抽象类 `WebMvcConfigurerAdapter`，不过在 2.3.4.RELEASE 这个版本中被弃用了，如下：

```
Watermark  
@Deprecated  
public abstract class WebMvcConfigurerAdapter implements WebMvcConfigurer {}
```

举个栗子：现在要添加一个拦截器，使其在Spring Boot中生效，此时就可以在MVC的配置类重写 `addInterceptors()` 方法，如下：

```

/**
 * MVC扩展的配置类，实现WebMvcConfigurer接口
 */
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Autowired
    private RepeatSubmitInterceptor repeatSubmitInterceptor;

    /**
     * 重写addInterceptors方法，注入自定义的拦截器
     */
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(repeatSubmitInterceptor).excludePathPatterns("/error");
    }
}

```

操作很简单，除了拦截器，还可以定制视图解析，资源映射处理器等等相关的功能，和Spring MVC很类似，只不过Spring MVC是在 XML 文件中配置，Spring Boot是在配置类中配置而已。

什么都不配置为什么依然能运行MVC相关的功能？

早期的SSM架构中想要搭建一个MVC其实挺复杂的，需要配置视图解析器，资源映射处理器，DispatcherServlet 等等才能正常运行，但是为什么Spring Boot仅仅是添加一个 WEB 模块依赖即能正常运行呢？依赖如下：

```

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

```

其实这已经涉及到了Spring Boot高级的知识点了，在这里就简单的说一下，Spring Boot的每一个 starter 都会有一个自动配置类，什么是自动配置类呢？**自动配置类就是在Spring Boot项目启动的时候会自动加载的类，能够在启动期间就配置一些默认的配置。** WEB 模块的自动配置类是 WebMvcAutoConfiguration。

WebMvcAutoConfiguration 这个配置类中还含有如下一个子配置类 WebMvcAutoConfigurationAdapter，如下：

```

@Configuration(proxyBeanMethods = false)
@Import(EnableWebMvcConfiguration.class)
@EnableConfigurationProperties({ WebMvcProperties.class, ResourceProperties.class })
@Order(0)
public static class WebMvcAutoConfigurationAdapter implements WebMvcConfigurer {}

```

WebMvcAutoConfigurationAdapter 这个子配置类实现了 WebMvcConfigurer 这个接口，这个正是MVC扩展接口，这个就很清楚了。**自动配置类是在项目启动的时候就加载的，因此Spring Boot会在项目启动时加载 WebMvcAutoConfigurationAdapter 这个MVC扩展配置类，提前完成一些默认的配置（比如内置了默认的视图解析器，资源映射处理器等）**，这也就是为什么没有配置什么MVC相关的东西依然能够运行。

如何全面接管MVC? 【不推荐】

全面接管MVC是什么意思呢？全面接管的意思就是不需要Spring Boot自动配置，而是全部使用自定义的配置。

全面接管MVC其实很简单，只需要在配置类上添加一个 `@EnableWebMvc` 注解即可。 还是添加拦截器，例子如下：

```
/*
 * @EnableWebMvc: 全面接管MVC，导致自动配置类失效
 */
@Configuration
@EnableWebMvc
public class WebMvcConfig implements WebMvcConfigurer {
    @Autowired
    private RepeatSubmitInterceptor repeatSubmitInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //添加拦截器
        registry.addInterceptor(repeatSubmitInterceptor).excludePathPatterns("/error");
    }
}
```

一个注解就能全面接口MVC，是不是很爽，不过，不建议使用。

为什么`@EnableWebMvc`一个注解就能够全面接管MVC?

Watermark

what? ? ? 为什么呢？上面刚说过自动配置类 `WebMvcAutoConfiguration` 会在项目启动期间加载一些默认的配置，这会怎么添加一个 `@EnableWebMvc` 注解就不行了呢？



其实很简单，`@EnableWebMvc` 源码如下：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import(DelegatingWebMvcConfiguration.class)
public @interface EnableWebMvc { }
```

其实重要的就是这个 `@Import(DelegatingWebMvcConfiguration.class)` 注解了，Spring中的注解，快速导入一个配置类 `DelegatingWebMvcConfiguration`，源码如下：

```
@Configuration(proxyBeanMethods = false)
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {}
```

明白了，`@EnableWebMvc` 这个注解实际上就是导入了一个 `WebMvcConfigurationSupport` 子类型的配置类而已。

而WEB模块的自动配置类有这么一行注解

`@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)`，源码如下：

Watermark

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class, TaskExecutionAutoConfiguration.class,
    ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {

```

这个注解 `@ConditionalOnMissingBean` 什么意义呢？简单的说就是IOC容器中没有指定的 Bean 这个配置才会生效。

一切都已经揭晓了，`@EnableWebMvc` 导入了一个 `WebMvcConfigurationSupport` 类型的配置类，导致了自动配置类 `WebMvcAutoConfiguration` 标注的

`@@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)` 判断为 `false` 了，从而自动配置类失效了。

Spring Boot相关资料

前期有很多的小伙伴私信我，觉得看文章太枯燥了，有些东西也不能理解的透彻，有没有好的视频课程分享，前几天特意回家找了找资源，总算找到了适合入门学习的完整视频教程，从Spring Boot初级入门到高级整合，讲解的非常全面，一些目录如下：

名称
1、缓存-JSR107简介.avi
2、缓存-Spring缓存抽象简介.avi
3、缓存-基本环境搭建.avi
4、缓存-@Cacheable初体验.avi
5、缓存-缓存工作原理&@Cacheable运行流程.avi
6、缓存-@Cacheable其他属性.avi
7、缓存-@CachePut.avi
8、缓存-@CacheEvict.avi
9、缓存-@Caching&@CacheConfig.avi
10、缓存-搭建redis环境&测试.avi
11、缓存-RedisTemplate&序列化机制.avi
12、缓存-自定义CacheManager.avi
13、消息-JMS&AMQP简介.avi
14、消息-RabbitMQ基本概念简介.avi
15、消息-RabbitMQ运行机制.avi
16、消息-RabbitMQ安装测试.avi
17、消息-RabbitTemplate发送接受消息&序列化机...
18、消息-@RabbitListener&@EnableRabbit.avi
19、消息-AmqpAdmin管理组件的使用.avi
20、检索-Elasticsearch简介&安装.avi
21、检索-Elasticsearch快速入门.avi
22、检索-Spring Boot整合Jest操作ES.avi
23、检索-整合SpringDataElasticsearch.avi

这些资料全部免费提供，我的文章也是尽量跟着视频大纲匹配，希望小伙伴们能够系统完整的学习Spring Boot。公众号【码猿技术专栏】回复关键词 `Spring Boot初级` 和 `Spring Boot高级` 分别获取初级和高级的视频教程。

总结

扩展和全面接管MVC都很简单，但是不推荐全面接管MVC，一旦全面接管了，WEB模块的这个 **starter** 将没有任何意义，一些全局配置文件中与MVC相关的配置也将会失效。

另外作者的第一本PDF书籍已经整理好了，由浅入深的详细介绍了Mybatis基础以及底层源码，有需要的朋友公众号回复关键词 **Mybatis进阶** 即可获取，目录如下：



满屏的try-catch，你不瘳得慌？

前言

软件开发过程中不可避免各种的异常，解决的异常，一直就是在解决异常的路上永不停歇，如果你的代码中再出现 `try() {...} catch() {...} finally{...}` 代码块，你还有心情看下去吗？自己不觉得恶心吗？

冗余的代码往往回丧失写代码的动力，每天搬砖似的写代码，真的很难受。今天这篇文章教你如何去掉满屏的 `try() {...} catch() {...} finally{...}`，解放你的双手。

Spring Boot 版本

本文基于的Spring Boot的版本是 2.3.4.RELEASE。

全局统一异常处理的前世今生

早在 Spring 3.x 就已经提出了 `@ControllerAdvice`，可以与 `@ExceptionHandler`、`@InitBinder`、`@ModelAttribute` 等注解注解配套使用，这几个此处就不再详细解释了。

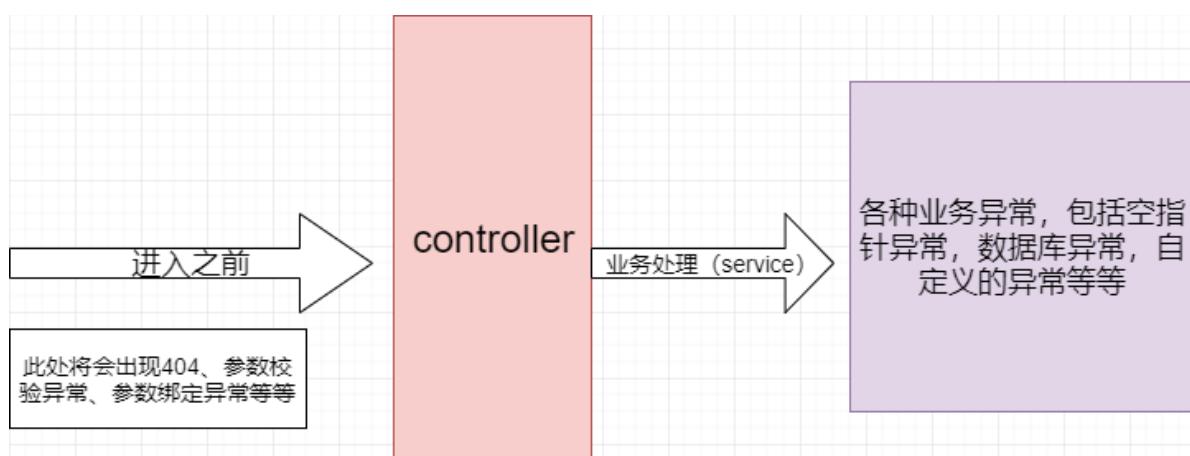
这几个注解小眼一瞟只有 `@ExceptionHandler` 与异常有关啊，翻译过来就是 异常处理器。其实异常的处理可以分为两类，分别是 局部异常处理 和 全局异常处理。

局部异常处理：`@ExceptionHandler` 和 `@Controller` 注解搭配使用，只有指定的controller层出现了异常才会被 `@ExceptionHandler` 捕获到，实际生产中怕是有成百上千个controller了吧，显然这种方式不合适。

全局异常处理：既然局部异常处理不合适了，自然有人站出来解决问题了，于是就有了 `@ControllerAdvice` 这个注解的横空出世了，`@ControllerAdvice` 搭配 `@ExceptionHandler` 彻底解决了全局统一异常处理。当然后面还出现了 `@RestControllerAdvice` 这个注解，其实就是 `@ControllerAdvice` 和 `@ResponseBody` 结晶。

Spring Boot的异常如何分类？

Java中的异常就很多，更别说Spring Boot中的异常了，这里不再根据传统意义上Java的异常进行分类了，而是按照 controller 进行分类，分为 进入controller前的异常 和 业务层的异常，如下图：



进入controller之前异常一般是 `javax.servlet.ServletException` 类型的异常，因此在全局异常处理的时候需要统一处理。几个常见的异常如下：

1. `NoHandlerFoundException`：客户端的请求没有找到对应的controller，将会抛出 404 异常。
2. `HttpRequestMethodNotSupportedException`：若匹配到了（匹配结果是一个列表，不同的是http方法不同，如：Get、Post等），则尝试将请求的http方法与列表的控制器做匹配，若没有对应http方法的控制器，则抛该异常。
3. `HttpMediaTypeNotSupportedError`：然后再对请求头与控制器支持的做比较，比如 `content-type` 请求头，若控制器的参数签名包含注解 `@RequestBody`，但是请求的 `content-type` 请求头的值没有包含 `application/json`，那么会抛该异常（当然，不止这种情况会抛这个异常）
4. `MissingPathVariable`：未检测到路径参数。比如url为：/user/{userId}，参数签名包含 `@PathVariable("userId")`，当请求的url为/user，在没有明确定义url为/user的情况下，会被判定

为：缺少路径参数

如何统一异常处理？

在统一异常处理之前其实还有许多东西需要优化的，比如统一结果返回的形式。当然这里不再细说了，不属于本文范畴。

统一异常处理很简单，这里以前后端分离的项目为例，步骤如下：

1. 新建一个统一异常处理的一个类
2. 类上标注 `@RestControllerAdvice` 这一个注解，或者同时标注 `@ControllerAdvice` 和 `@ResponseBody` 这两个注解。
3. 在方法上标注 `@ExceptionHandler` 注解，并且指定需要捕获的异常，可以同时捕获多个。

下面是作者随便配置一个demo，如下：

```
/**  
 * 全局统一的异常处理，简单的配置下，根据自己的业务要求详细配置  
 */  
@RestControllerAdvice  
@Slf4j  
public class GlobalExceptionHandler {  
  
    /**  
     * 重复请求的异常  
     * @param ex  
     * @return  
     */  
    @ExceptionHandler(RepeatSubmitException.class)  
    public ResultResponse onException(RepeatSubmitException ex) {  
        //打印日志  
        log.error(ex.getMessage());  
        //todo 日志入库等等操作  
  
        //统一结果返回  
        return new ResultResponse(ResultCodeEnum.CODE_NOT_REPEAT_SUBMIT);  
    }  
  
    /**  
     * 自定义的业务上的异常  
     */  
    @ExceptionHandler(ServiceException.class)  
    public ResultResponse onException(ServiceException ex) {  
        //打印日志  
        log.error(ex.getMessage());  
        //todo 日志入库等等操作  
  
        //统一结果返回  
        return new ResultResponse(ResultCodeEnum.CODE_SERVICE_FAIL);  
    }  
  
    /**  
     * 捕获一些进入controller之前的异常，有些4xx的状态码统一设置为200  
     */
```

```
* @param ex
* @return
*/
@ExceptionHandler({HttpRequestMethodNotSupportedException.class,
    HttpMediaTypeNotSupportedException.class, MediaTypeNotAcceptableException.class,
    MissingPathVariableException.class, MissingServletRequestParameterException.class,
    ServletRequestBindingException.class, ConversionNotSupportedException.class,
    TypeMismatchException.class, HttpMessageNotReadableException.class,
    HttpMessageNotWritableException.class,
    MissingServletRequestPartException.class, BindException.class,
    NoHandlerFoundException.class, AsyncRequestTimeoutException.class})
public ResultResponse onException(Exception ex) {
    //打印日志
    log.error(ex.getMessage());
    //todo 日志入库等等操作

    //统一结果返回
    return new ResultResponse(ResultCodeEnum.CODE_FAIL);
}
}
```

注意：上面的只是一个例子，实际开发中还有许多的异常需要捕获，比如 TOKEN失效、过期等等异常，如果整合了其他的框架，还要注意这些框架抛出的异常，比如 Shiro，Spring Security 等等框架。

异常匹配的顺序是什么？

Watermark

有些朋友可能疑惑了，如果我同时捕获了父类和子类，那么到底能够被那个异常处理器捕获呢？比如 `Exception` 和 `ServiceException`。



此时可能就疑惑了，这里先揭晓一下答案，当然是 `ServiceException` 的异常处理器捕获了，精确匹配，如果没有 `ServiceException` 的异常处理器才会轮到它的父亲，父亲没有才会到祖父。总之一句话，精准匹配，找那个关系最近的。

为什么呢？这可不是凭空瞎说的，源码为证，出处

`org.springframework.web.method.annotation.ExceptionHandlerMethodResolver#getMappedMethod`，如下：

```
@Nullable
private Method getMappedMethod(Class<? extends Throwable> exceptionType) {
    List<Class<? extends Throwable>> matches = new ArrayList<>();
    //遍历异常处理器中定义的异常类型
    for (Class<? extends Throwable> mappedException : this.mappedMethods.keySet()) {
        //是否是抛出异常的父类，如果是添加到集合中
        if (mappedException.isAssignableFrom(exceptionType)) {
            //添加到集合中
            matches.add(mappedException);
        }
    }
    //如果集合不为空，则按照规则进行排序
    if (!matches.isEmpty()) {
        matches.sort(new ExceptionDepthComparator(exceptionType));
        return this.mappedMethods.get(matches.get(0));
    }
    else {
        return null;
    }
}
```

}

在初次异常处理的时候会执行上述的代码找到最匹配的那个异常处理器方法，后续都是直接从缓存中（一个 Map 结构， key 是异常类型， value 是异常处理器方法）。

别着急，上面代码最精华的地方就是对 matches 进行排序的代码了，我们来看看 ExceptionDepthComparator 这个比较器的关键代码，如下：

```
//递归调用，获取深度，depth值越小越精准匹配
private int getDepth(Class<?> declaredException, Class<?> exceptionToMatch, int depth) {
    //如果匹配了，返回
    if (exceptionToMatch.equals(declaredException)) {
        // Found it!
        return depth;
    }
    // 递归结束的条件，最大限度了
    if (exceptionToMatch == Throwable.class) {
        return Integer.MAX_VALUE;
    }
    //继续匹配父类
    return getDepth(declaredException, exceptionToMatch.getSuperclass(), depth + 1);
}
```

精髓全在这里了，一个递归搞定，计算深度， depth 初始值为0。值越小，匹配度越高越精准。

总结

全局异常的文章万万千，能够讲清楚的能有几篇呢？只出最精的文章，做最野的程序员，如果觉得不错的，关注分享走一波，谢谢支持！！！

另外作者的第一本PDF书籍已经整理好了，由浅入深的详细介绍了Mybatis基础以及底层源码，有需要的朋友公众号回复关键词 Mybatis进阶 即可获取，目录如下：





码猿技术专栏 ★

互联网的百科全书，公众号发起人不才陈某，多年老司机，就职于蚂蚁金服。



扫一扫关注一下

优质资源分享！Spring Boot 入门到放弃！！！

前言

最近不知不觉写Spring Boot专栏已经写了九篇文章了，从最底层的项目搭建到源码解析以及高级整合的部分，作者一直在精心准备文章，定时更新，有兴趣的可以看我的专栏[Spring Boot进阶](#)。

有些读者反映文章更新的有点慢，想要尽快的入门Spring Boot，想我推荐一个Spring Boot的视频教程，最好能够和我的文章大纲有契合的地方。于是作者回家找了一套个人觉得很好的视频教程，今天在这里免费分享给大家。

视频目录

整个教程分为[基础](#)和[高级整合](#)两个部分，每套教程都有完整的[代码](#)和上课的[课件](#)，可以说是非常完整的。

基础的部分目录如下：

1. 入门-课程简介
2. 入门-Spring Boot简介
3. 入门-微服务简介
4. 入门-环境准备
5. 入门-springboot-helloworld
6. 入门-HelloWorld细节-场景启动器（starter）
7. 入门-HelloWorld细节-自动配置
8. 入门-使用向导快速创建Spring Boot应用
9. 配置-yaml简介
10. 配置-yaml语法
11. 配置 yaml 配置文件加载
12. 配置-properties配置文件编码问题
13. 配置-@ConfigurationProperties与@Value区别
14. 配置-@PropertySource.@ImportResource.@Bean
15. 配置-配置文件占位符

16. 配置-Profile多环境支持
17. 配置-配置文件的加载位置
18. 配置-外部配置加载顺序
19. 配置-自动配置原理
20. 配置-@Conditional&自动配置报告
21. 日志-日志框架分类和选择
22. 日志-slf4j使用原理
23. 日志-其他日志框架统一转换为slf4j
24. 日志-SpringBoot日志关系
25. 日志-SpringBoot默认配置
26. 日志-指定日志文件和日志Profile功能
27. 日志-切换日志框架
28. web开发-简介
29. web开发-webjars&静态资源映射规则
30. web开发-引入thymeleaf
31. web开发-thymeleaf语法
32. web开发-SpringMVC自动配置原理
33. web开发-扩展与全面接管SpringMVC
34. web开发-【实验】-引入资源
35. web开发-【实验】-国际化
36. web开发-【实验】-登陆&拦截器
37. web开发-【实验】-Restful实验要求
38. web开发-【实验】-员工列表-公共页抽取
39. web开发-【实验】-员工列表-链接高亮&列表完成
40. web开发-【实验】-员工添加-来到添加页面
41. web开发-【实验】-员工添加-添加完成
42. web开发-【实验】-员工修改-重用页面&修改完成
43. web开发-【实验】-员工删除-删除成功
44. web开发-错误处理原理&定制错误页面
45. web开发-定制错误数据
46. web开发-嵌入式Servlet容器配置修改
47. web开发-注册servlet三大组件
48. web开发-切换其他嵌入式Servlet容器
49. web开发-嵌入式Servlet容器自动配置原理
50. web开发-嵌入式Servlet容器启动原理
51. web开发-使用外部Servlet容器&JSP支持
52. web开发-外部Servlet容器启动SpringBoot应用原理
53. Docker-简介
54. Docker-核心概念
55. Docker-linux环境准备
56. Docker-docker安装&启动&停止
57. Docker-docker镜像操作常用命令
58. Docker-docker容器操作常用命令
59. Docker-docker安装MySQL
60. 数据访问-简介
61. 数据访问-JDBC&自动配置原理
62. 数据访问-整合MyBatis -基础环境搭建
63. 数据访问-整合MyBatis (一) -基础环境搭建
64. 数据访问-整合MyBatis (二) -注解版MyBatis
65. 数据访问-整合MyBatis (二) -配置版MyBatis
66. 数据访问-SpringData JPA简介
67. 数据访问-整合JPA

68. 原理-第一步：创建SpringApplication
69. 原理-第二步：启动应用
70. 原理-事件监听机制相关测试
71. 原理-自定义starter
72. 结束语

高级整合的部分目录如下：

0. 尚硅谷SpringBoot高级 源码.课件
1. 缓存-JSR107简介
2. 缓存-Spring缓存抽象简介
3. 缓存-基本环境搭建
4. 缓存-@Cacheable初体验
5. 缓存-缓存工作原理&@Cacheable运行流程
6. 缓存-@Cacheable其他属性
7. 缓存-@CachePut
8. 缓存-@CacheEvict
9. 缓存-@Caching&@CacheConfig
10. 缓存-搭建redis环境&测试
11. 缓存-RedisTemplate&序列化机制
12. 缓存-自定义CacheManager
13. 消息-JMS&AMQP简介
14. 消息-RabbitMQ基本概念简介
15. 消息-RabbitMQ运行机制
16. 消息-RabbitMQ安装测试
17. 消息-RabbitTemplate发送接受消息&序列化机制
18. 消息-@RabbitListener&@EnableRabbit
19. 消息-AmqpAdmin管理组件的使用
20. 检索-Elasticsearch简介&安装
21. 检索-Elasticsearch快速入门
22. 检索-SpringBoot整合Jest操作ES
23. 检索-整合SpringDataElasticsearch
24. 任务-异步任务
25. 任务-定时任务
26. 任务-邮件任务
27. 安全-测试环境搭建
28. 安全-登录&认证&授权
29. 安全-权限控制&注销
30. 安全-记住我&定制登陆页
31. 分布式-dubbo简介
32. 分布式-docker安装zookeeper
33. 分布式-SpringBoot.Dubbo.Zookeeper整合
34. 分布式-SpringCloud-Eureka注册中心
35. 分布式-服务注册
36. 分布式-服务发现&消费
37. 热部署-devtools开发热部署
38. 监管-监管端点测试
39. 监管-自定义端点
40. 监管-自定义HealthIndicator

如何获取

分别回复关键词 `Spring Boot初级` 和 `Spring Boot高级` 即可获取两套视频教程。

纯属个人分享，如果有侵权立即删除。

总结

作者的专栏文章大致和这个视频教程的大纲相同，有兴趣的可以结合着视频学习一波。



这类注解都不知道，还好意思说用过Spring Boot

前言

不知道大家在使用Spring Boot开发的日常中有没有用过`@Conditionalxxx`注解，比如`@ConditionalOnMissingBean`。相信看过Spring Boot源码的朋友一定不陌生。

`@Conditionalxxx`这类注解表示某种判断条件成立时才会执行相关操作。掌握该类注解，有助于日常开发，框架的搭建。

今天这篇文章就从前世今生介绍一下该类注解。

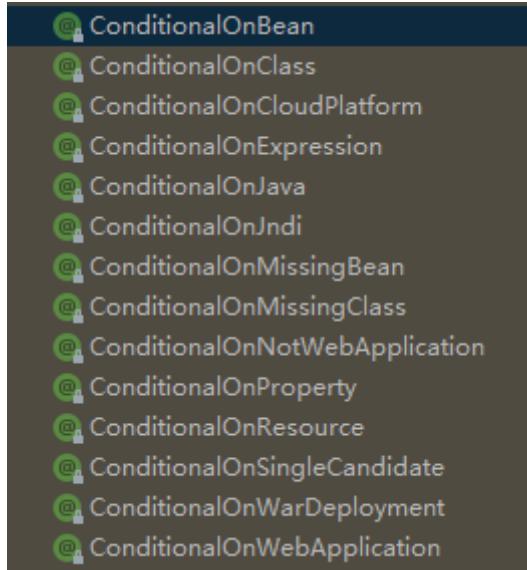
Spring Boot 版本 Watermark

本文基于的Spring Boot的版本是 `2.3.4.RELEASE`。

@Conditional

`@Conditional` 注解是从 Spring4.0 才有的，可以用在任何类型或者方法上面，通过 `@Conditional` 注解可以配置一些条件判断，当所有条件都满足的时候，被 `@Conditional` 标注的目标才会被 Spring 容器处理。

`@Conditional` 的使用很广，比如控制某个 Bean 是否需要注册，在 Spring Boot 中的变形很多，比如 `@ConditionalOnMissingBean`、`@ConditionalOnBean` 等等，如下：



该注解的源码其实很简单，只有一个属性 `value`，表示判断的条件（一个或者多个），是 `org.springframework.context.annotation.Condition` 类型，源码如下：

```
@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Conditional {  
  
    /**  
     * All {@link Condition} classes that must {@linkplain Condition#matches match}  
     * in order for the component to be registered.  
     */  
    Class<? extends Condition>[] value();  
}
```

`@Conditional` 注解实现的原理很简单，就是通过 `org.springframework.context.annotation.Condition` 这个接口判断是否应该执行操作。

Condition 接口

`@Conditional` 注解判断条件与否取决于 `value` 属性指定的 `Condition` 实现，其中有一个 `matches()` 方法，返回 `true` 表示条件成立，反之不成立，接口如下：

```
@FunctionalInterface  
public interface Condition {  
    boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata);  
}
```

`matches` 中的两个参数如下：

1. `context` : 条件上下文, `ConditionContext` 接口类型的, 可以用来获取容器中上下文信息。
2. `metadata` : 用来获取被 `@Conditional` 标注的对象上的所有注解信息

ConditionContext接口

这个接口很重要, 能够从中获取Spring上下文的很多信息, 比如 `ConfigurableListableBeanFactory`, 源码如下:

```
public interface ConditionContext {  
  
    /**  
     * 返回bean定义注册器, 可以通过注册器获取bean定义的各种配置信息  
     */  
    BeanDefinitionRegistry getRegistry();  
  
    /**  
     * 返回ConfigurableListableBeanFactory类型的bean工厂, 相当于一个ioc容器对象  
     */  
    @Nullable  
    ConfigurableListableBeanFactory getBeanFactory();  
  
    /**  
     * 返回当前spring容器的环境配置信息对象  
     */  
    Environment getEnvironment();  
  
    /**  
     * 返回资源加载器  
     */  
    ResourceLoader getResourceLoader();  
  
    /**  
     * 返回类加载器  
     */  
    @Nullable  
    ClassLoader getClassLoader();  
}
```

如何自定义Condition?

举个栗子: 假设有这样一个需求, 需要根据运行环境注入不同的 Bean , Windows 环境和 Linux 环境注入不同的 Bean 。

实现很简单, 分别定义不同环境的判断条件, 实现 `org.springframework.context.annotation.Condition` 即可。

windows环境的判断条件源码如下:

```
/**  
 * 操作系统的匹配条件, 如果是 windows 系统, 则返回 true  
 */  
public class WindowsCondition implements Condition {  
    @Override  
    public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata metadata) {  
        //获取当前环境信息
```

```

        Environment environment = conditionContext.getEnvironment();
        //获得当前系统名
        String property = environment.getProperty("os.name");
        //包含Windows则说明是windows系统，返回true
        if (property.contains("Windows")){
            return true;
        }
        return false;
    }
}

```

Linux环境判断源码如下：

```

/**
 * 操作系统的匹配条件，如果是windows系统，则返回true
 */
public class LinuxCondition implements Condition {
    @Override
    public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata metadata) {
        Environment environment = conditionContext.getEnvironment();

        String property = environment.getProperty("os.name");
        if (property.contains("Linux")){
            return true;
        }
        return false;
    }
}

```

配置类中结合 @Bean 注入不同的Bean，如下：

```

@Configuration
public class CustomConfig {

    /**
     * 在Windows环境下注入的Bean为winP
     * @return
     */
    @Bean("winP")
    @Conditional(value = {WindowsCondition.class})
    public Person personWin() {
        return new Person();
    }

    /**
     * 在Linux环境下注入的Bean为LinuxP
     * @return
     */
    @Bean("LinuxP")
    @Conditional(value = {LinuxCondition.class})
    public Person personLinux() {
        return new Person();
    }
}

```

简单的测试一下，如下：

```

@SpringBootTest
class SpringbootInterceptApplicationTests {

    @Autowired(required = false)
    @Qualifier(value = "winP")
    private Person winP;

    @Autowired(required = false)
    @Qualifier(value = "LinuxP")
    private Person linP;

    @Test
    void contextLoads() {
        System.out.println(winP);
        System.out.println(linP);
    }
}

```

Windows环境下执行单元测试，输出如下：

```

com.example.springbootintercept.domain.Person@885e7ff
null

```

很显然，判断生效了，Windows环境下只注入了 **WINP**。

条件判断在什么时候执行？

条件判断的执行分为两个阶段，如下：

1. **配置类解析阶段(ConfigurationPhase.PARSE_CONFIGURATION)**：在这个阶段会得到一批配置类的信息和一些需要注册的 Bean。
2. **Bean注册阶段(ConfigurationPhase.REGISTER_BEAN)**：将配置类解析阶段得到的配置类和需要注册的Bean注入到容器中。

默认都是配置解析阶段，其实也就够用了，但是在Spring Boot中使用了 **ConfigurationCondition**，这个接口可以自定义执行阶段，比如 **@ConditionalOnMissingBean** 都是在Bean注册阶段执行，因为需要从容器中判断Bean。

这个两个阶段有什么不同呢？：其实很简单的，配置类解析阶段只是将需要加载配置类和一些 Bean（被 **@Conditional** 注解过滤掉之后）收集起来，而Bean注册阶段是将的收集来的Bean和配置类注入到容器中，如果在配置类解析阶段执行 **Condition** 接口的 **matches()** 接口去判断某些 Bean是否存在IOC容器中，这个显然是不行的，因为这些Bean还未注册到容器中。

什么是配置类，有哪些？：类上被 **@Component**、**@ComponentScan**、**@Import**、**@ImportResource**、**@Configuration** 标注的以及类中方法有 **@Bean** 的方法。如何判断配置类，在源码中有单独的方法：

```
org.springframework.context.annotation.ConfigurationClassUtils#isConfigurationCandidate。
```

ConfigurationCondition接口

这个接口相比于 `@Condition` 接口就多了一个 `getConfigurationPhase()` 方法，可以自定义执行阶段。源码如下：

```
public interface ConfigurationCondition extends Condition {

    /**
     * 条件判断的阶段，是在解析配置类的时候过滤还是在创建bean的时候过滤
     */
    ConfigurationPhase getConfigurationPhase();

    /**
     * 表示阶段的枚举：2个值
     */
    enum ConfigurationPhase {
        /**
         * 配置类解析阶段，如果条件为false，配置类将不会被解析
         */
        PARSE_CONFIGURATION,
        /**
         * bean注册阶段，如果为false，bean将不会被注册
         */
        REGISTER_BEAN
    }
}
```

这个接口在需要指定执行阶段的时候可以实现，比如需要根据某个Bean是否在IOC容器中来注入指定的Bean，则需要指定执行阶段为**Bean的注册阶段**（`ConfigurationPhase.REGISTER_BEAN`）。

多个Condition的执行顺序

`@Conditional` 中的 `Condition` 判断条件可以指定多个，默认是按照先后顺序执行，如下：

```
class Condition1 implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        System.out.println(this.getClass().getName());
        return true;
    }
}

class Condition2 implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        System.out.println(this.getClass().getName());
        return true;
    }
}

class Condition3 implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        System.out.println(this.getClass().getName());
    }
}
```

```

        return true;
    }
}

@Configuration
@Conditional({Condition1.class, Condition2.class, Condition3.class})
public class MainConfig5 {
}

```

上述例子会依次按照 Condition1、Condition2、Condition3 执行。

默认按照先后顺序执行，但是当我们需要指定顺序呢？很简单，有如下三种方式：

1. 实现 PriorityOrdered 接口，指定优先级
2. 实现 Ordered 接口接口，指定优先级
3. 使用 @Order 注解来指定优先级

例子如下：

```

@Order(1)
class Condition1 implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        System.out.println(this.getClass().getName());
        return true;
    }
}

class Condition2 implements Condition, Ordered {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        System.out.println(this.getClass().getName());
        return true;
    }

    @Override
    public int getOrder() {
        return 0;
    }
}

class Condition3 implements Condition, PriorityOrdered {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        System.out.println(this.getClass().getName());
        return true;
    }

    @Override
    public int getOrder() {
        return 1000;
    }
}

@Configuration
@Conditional({Condition1.class, Condition2.class, Condition3.class})
public class MainConfig6 {
}

```

根据排序的规则，`PriorityOrdered` 的会排在前面，然后会再按照 `order` 升序，最后可以顺序是：
`Condition3->Condition2->Condition1`

Spring Boot中常用的一些注解

Spring Boot中大量使用了这些注解，常见的注解如下：

1. `@ConditionalOnBean`：当容器中有指定Bean的条件下进行实例化。
2. `@ConditionalOnMissingBean`：当容器里没有指定Bean的条件下进行实例化。
3. `@ConditionalOnClass`：当classpath类路径下有指定类的条件下进行实例化。
4. `@ConditionalOnMissingClass`：当类路径下没有指定类的条件下进行实例化。
5. `@ConditionalOnWebApplication`：当项目是一个Web项目时进行实例化。
6. `@ConditionalOnNotWebApplication`：当项目不是一个Web项目时进行实例化。
7. `@ConditionalOnProperty`：当指定的属性有指定的值时进行实例化。
8. `@ConditionalOnExpression`：基于SpEL表达式的条件判断。
9. `@ConditionalOnJava`：当JVM版本为指定的版本范围时触发实例化。
10. `@ConditionalOnResource`：当类路径下有指定的资源时触发实例化。
11. `@ConditionalOnJndi`：在JNDI存在的条件下触发实例化。
12. `@ConditionalOnSingleCandidate`：当指定的Bean在容器中只有一个，或者有多个但是指定了首选的Bean时触发实例化。

比如在 `WEB` 模块的自动配置类 `WebMvcAutoConfiguration` 下有这样一段代码：

```
@Bean  
 @ConditionalOnMissingBean  
 public InternalResourceViewResolver defaultViewResolver() {  
     InternalResourceViewResolver resolver = new InternalResourceViewResolver();  
     resolver.setPrefix(this.mvcProperties.getView().getPrefix());  
     resolver.setSuffix(this.mvcProperties.getView().getSuffix());  
     return resolver;  
 }
```

常见的 `@Bean` 和 `@ConditionalOnMissingBean` 注解结合使用，意思是当容器中没有 `InternalResourceViewResolver` 这种类型的Bean才会注入。这样写有什么好处呢？好处很明显，可以让开发者自定义需要的视图解析器，如果没有自定义，则使用默认的，这就是Spring Boot为自定义配置提供的便利。

总结

`@Conditional` 注解在Spring Boot中演变的注解很多，需要着重了解，特别是后期框架整合的时候会大量涉及。

Watermark



Spring Boot整合多点套路，少走点弯路

前言

网上有很多文章都在说 Spring Boot 如何整合 xxx，有文章教你为什么这么整合吗？整合了千万个框架，其实套路就那么几个，干嘛要学千万个，不如来这学习几个套路轻松整合，它不香吗？？？

今天写这篇文章的目的就是想从思想上教给大家几个套路，不用提到整合什么就去百度了，自己尝试去亲手整合一个。

Spring Boot 版本

本文基于的Spring Boot的版本是 2.3.4.RELEASE。

1. 找到自动配置类

Spring Boot 在整合任何一个组件的时候都会先添加一个依赖 `starter`，比如整合的Mybatis有一个 `mybatis-spring-boot-starter`，依赖如下：

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
```

每一个 `starter` 基本都会有一个自动配置类，命名方式也是类似的，格式为：`xxxAutoConfiguration`，比如Mybatis的自动配置类就是 `MybatisAutoConfiguration`，Redis 的自动配置类是 `RedisAutoConfiguration`，WEB 模块的自动配置类是 `WebMvcAutoConfiguration`。

2. 注意`@Conditionalxxx`注解

`@Conditionalxxx` 标注在配置类上或者结合 `@Bean` 标注在方法上，究竟是什么意思，在上一篇文章 [这类注解都不知道，还好意思说会Spring Boot](#) 已经从表层到底层深入的讲了一遍，不理解的可以查阅一下。

首先需要注意自动配置类上的 `@Conditionalxxx` 注解，这个是自动配置类生效的条件。

比如 `WebMvcAutoConfiguration` 类上标了一个如下注解：

```
@ConditionalOnMissingBean (WebMvcConfigurationSupport.class)
```

以上这行代码的意思就是当前IOC容器中没有 `WebMvcConfigurationSupport` 这个类的实例时自动配置类才会生效，这也就是在配置类上标注 `@EnableWebMvc` 会导致自动配置类 `WebMvcAutoConfiguration` 失效的原因。

其次需要注意方法上的 `@Conditionalxxx` 注解，Spring Boot会在自动配置类中结合 `@Bean` 和 `@Conditionalxxx` 注解提供一些组件运行的默认配置，但是利用 `@Conditionalxxx`（在特定条件下生效）注解的 `条件性`，方便开发者覆盖这些配置。

比如在Mybatis的自动配置类 `MybatisAutoConfiguration` 中有如下一个方法：

```
@Bean  
 @ConditionalOnMissingBean  
 public SqlSessionFactory sqlSessionFactory (DataSource dataSource) throws Exception {}
```

以上这个方法不用看方法体的内容，只看方法上的注解。`@Bean` 这个注解的意思是注入一个 Bean 到 IOC 容器中，`@ConditionalOnMissingBean` 这个注解就是一个条件判断了，表示当 `SqlSessionFactory` 类型的对象在 IOC 容器 中不存在才会注入。

哦？领悟到了吧，**言外之意就是如果开发者需要定制 `SqlSessionFactory`，则可以自己的创建一个 `SqlSessionFactory` 类型的对象并且注入到IOC容器中即能覆盖自动配置类中的。** 比如在Mybatis配置多数据源的时候就需要定制一个 `SqlSessionFactory` 而不是使用自动配置类中的。

总之，一定要注意自动配置类上或者方法上的 `@Conditionalxxx` 注解，这个注解表示某种特定条件。

下面列出了常用的几种注解，如下：

1. `@ConditionalOnBean`：当容器中有指定Bean的条件下进行实例化。
2. `@ConditionalOnMissingBean`：当容器里没有指定Bean的条件下进行实例化。
3. `@ConditionalOnClass`：当classpath类路径下有指定类的条件下进行实例化。
4. `@ConditionalOnMissingClass`：当类路径下没有指定类的条件下进行实例化。
5. `@ConditionalOnApplication`：当项目是一个Web项目时进行实例化。
6. `@ConditionalOnNotWebApplication`：当项目不是一个Web项目时进行实例化。
7. `@ConditionalOnProperty`：当指定的属性有指定的值时进行实例化。
8. `@ConditionalOnExpression`：基于SpEL表达式的条件判断。
9. `@ConditionalOnJava`：当JVM版本为指定的版本范围时触发实例化。

10. `@ConditionalOnResource` : 当类路径下有指定的资源时触发实例化。
11. `@ConditionalOnJndi` : 在JNDI存在的条件下触发实例化。
12. `@ConditionalOnSingleCandidate` : 当指定的Bean在容器中只有一个，或者有多个但是指定了首选的Bean时触发实例化。

3. 注意`EnableConfigurationProperties`注解

`EnableConfigurationProperties` 这个注解常标注在配置类上，使得 `@ConfigurationProperties` 标注的配置文件生效，这样就可以在全局配置文件（`application.xxx`）配置指定前缀的属性了。

在Redis的自动配置类 `RedisAutoConfiguration` 上方标注如下一行代码：

```
@EnableConfigurationProperties(RedisProperties.class)
```

这行代码有意思了，我们可以看看 `RedisProperties` 的源码，如下：

```
@ConfigurationProperties(prefix = "spring.redis")
public class RedisProperties {
    private int database = 0;
    private String url;
    private String host = "localhost";
    private String password;
    ...
}
```

`@ConfigurationProperties` 这个注解指定了全局配置文件中以 `spring.redis.xxx` 为前缀的配置都会映射到 `RedisProperties` 的指定属性中，其实 `RedisProperties` 这个类中定义了Redis的一些所需属性，比如 `host`，`IP地址`，`密码` 等等。

`@EnableConfigurationProperties` 注解就是使得指定的配置生效，能够将全局配置文件中配置的属性映射到相关类的属性中。

为什么要注意 `@EnableConfigurationProperties` 这个注解呢？

引入一个组件后往往需要改些配置，我们都知道在全局配置文件中可以修改，但是不知道前缀是什么，可以改哪些属性，因此找到 `@EnableConfigurationProperties` 这个注解后就能找到对应的配置前缀以及可以修改的属性了。

4. 注意`@Import`注解

这个注解有点牛逼了，`Spring 3.x` 中就已经有的一个注解，大致的意思的就是快速导入一个Bean或者配置类到IOC容器中。这个注解有很多妙用，后续会单独写篇文章介绍下。

`@Import` 这个注解通常标注在自动配置类上方，并且一般都是导入一个或者多个配置类。

比如 `RabbitMQ` 的自动配置类 `RabbitAutoConfiguration` 上有如下一行代码：

```
@Import(RabbitAnnotationDrivenConfiguration.class)
```

这行代码的作用就是添加了 `RabbitAnnotationDrivenConfiguration` 这个配置类，使得Spring Boot在加载到自动配置类的时候能够一起加载。

比如Redis的自动配置类 `RedisAutoConfiguration` 上有如下一行代码：

```
@Import({ LettuceConnectionConfiguration.class, JedisConnectionConfiguration.class })
```

这个 `@Import` 同时引入了 `Lettuce` 和 `Jedis` 两个配置类了，因此如果你的Redis需要使用Jedis作为连接池的话，想要知道Jedis都要配置什么，此时就应该看看 `JedisConnectionConfiguration` 这个配置类了。

总结： `@Import` 标注在自动配置类上方，一般都是快速导入一个或者多个配置类，因此如果自动配置类没有配置一些东西时，一定要看看 `@Import` 这个注解导入的配置类。

5. 注意`@AutoConfigurexxx`注解

`@AutoConfigurexxx` 这类注解决定了自动配置类的加载顺序，比如 `AutoConfigureAfter`（在指定自动配置类之后）、`AutoConfigureBefore`（在指定自动配置类之前）、`AutoConfigureOrder`（指定自动配置类的优先级）。

为什么要注意顺序呢？因为某些组件往往之间是相互依赖的，比如 `Mybatis` 和 `DataSource`，肯定要先将数据源相关的东西配置成功才能配置 `Mybatis` 吧。`@AutoConfigurexxx` 这类注解正是解决了组件之间相互依赖的问题。

比如 `MybatisAutoConfiguration` 上方标注了如下一行代码：

```
@AutoConfigureAfter (DataSourceAutoConfiguration.class)
```

这个行代码意思很简单，就是 `MybatisAutoConfiguration` 这个自动配置在 `DataSourceAutoConfiguration` 这个之后加载，因为你需要我，多么简单的理由。

好了，这下明白了吧，以后别犯傻问：**为什么Mybatis配置好了，启动会报错？**这个问题先看看数据源有没有配置成功吧。

6. 注意内部静态配置类

有些自动配置类比较简单没那么多套路，比如 `RedisAutoConfiguration` 这个自动配置类中就定义了两个注入Bean的方法，其他的没了。

但是有些自动配置类就没那么单纯了，中间能嵌套 n 个静态配置类，比如 `WebMvcAutoConfiguration`，类中还嵌套了 `WebMvcAutoConfigurationAdapter`、`EnableWebMvcConfiguration`、`ResourceChainCustomizerConfiguration` 这三个配置类。如果你光看 `WebMvcAutoConfiguration` 这个自动配置类好像没配置什么，但是其内部却是大有乾坤啊。

总结：一定要自动配置类的内部嵌套的配置类，真是大有乾坤啊。

总结

以上总结了六种自动配置的套路，希望你能够即读即用，摆脱百度，自己也能独立整合组件。

总之，Spring Boot整合xxx组件的文章很多，相信大家也看的比较懵，其实套路都是一样，学会陈某分享的套路，让你少走弯路！！！



Spring Boot与多数据源那点事儿~

前言

大约在19年的这个时候，老同事公司在做医疗系统，需要和 [HIS](#) 系统对接一些信息，比如患者、医护、医嘱、科室等信息。但是起初并不知道如何与HIS无缝对接，于是向我取经。

最终经过讨论采用了 [视图对接](#) 的方式，大致就是HIS系统提供视图，他们进行对接。

写这篇文章的目的

这篇文章将会涉及到Spring Boot 与Mybatis、数据库整合，类似于整合Mybatis与数据库的文章其实网上很多，作者此前也写过一篇文章详细的介绍了一些整合的套路：[Spring Boot 整合多点套路，少走点弯路~](#)，有兴趣的可以看看。

什么是多数据源？

最常见的单一应用中最多涉及到一个数据库，即是一个数据源（[Datasource](#)）。那么顾名思义，多数据源就是在一个单一应用中涉及到了两个及以上的数据库了。

其实在配置数据源的时候就已经很明确这个定义了，如以下代码：

Watermark

```
@Bean(name = "dataSource")
public DataSource dataSource() {
    DruidDataSource druidDataSource = new DruidDataSource();
    druidDataSource.setUrl(url);
    druidDataSource.setUsername(username);
    druidDataSource.setDriverClassName(driverClassName);
    druidDataSource.setPassword(password);
    return druidDataSource;
}
```

`url`、`username`、`password`这三个属性已经唯一确定了一个数据库了，`DataSource`则是依赖这三个创建出来的。则多数据源即是配置多个`DataSource`（暂且这么理解）。

何时用到多数据源？

正如前言介绍到的一个场景，相信大多数做过医疗系统的都会和`HIS`打交道，为了简化护士以及医生的操作流程，必须要将必要的信息从`HIS`系统对接过来，据我了解的大致有两种方案如下：

1. `HIS` 提供视图，比如医护视图、患者视图等，而此时其他系统只需要定时的从`HIS`视图中读取数据同步到自己数据库中即可。
2. `HIS` 提供接口，无论是`webService`还是`HTTP`形式都是可行的，此时其他系统只需要按照要求调接口即可。

很明显第一种方案涉及到了至少两个数据库了，一个是`HIS`数据库，一个自己系统的数据库，在单一应用中必然需要用到**多数据源的切换**才能达到目的。

当然多数据源的使用场景还是有很多的，以上只是简单的一个场景。

整合单一的数据源

本文使用阿里的数据库连接池`druid`，添加依赖如下：

```
<!--druid连接池-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.9</version>
</dependency>
```

阿里的数据库连接池非常强大，比如**数据监控**、**数据库加密**等等内容，本文仅仅演示与Spring Boot整合的过程，一些其他的功能后续可以自己研究添加。

Druid连接池的`starter`的自动配置类是`DruidDataSourceAutoConfigure`，类上标注如下一行注解：

```
@EnableConfigurationProperties({DruidStatProperties.class, DataSourceProperties.class})
```

Watermark

`@EnableConfigurationProperties`这个注解使得配置文件中的配置生效并且映射到指定类的属性。

`DruidStatProperties`中指定的前缀是`spring.datasource.druid`，这个配置主要是用来设置连接池的一些参数。

`DataSourceProperties` 中指定的前缀是 `spring.datasource`，这个主要是用来设置数据库的 `url`、`username`、`password` 等信息。

因此我们只需要在全局配置文件中指定**数据库的一些配置以及连接池的一些配置信息即可**，前缀分别是 `spring.datasource.druid`、`spring.datasource`，以下是个人随便配置的(`application.properties`)：

```
spring.datasource.url=jdbc:mysql://120.26.101.xxx:3306/xxx?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull&useSSL=false&allowMultiQueries=true&serverTimezone=Asia/Shanghai
spring.datasource.username=root
spring.datasource.password=xxxx
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
#初始化连接大小
spring.datasource.druid.initial-size=0
#连接池最大使用连接数量
spring.datasource.druid.max-active=20
#连接池最小空闲
spring.datasource.druid.min-idle=0
#获取连接最大等待时间
spring.datasource.druid.max-wait=6000
spring.datasource.druid.validation-query=SELECT 1
#spring.datasource.druid.validation-query-timeout=6000
spring.datasource.druid.test-on-borrow=false
spring.datasource.druid.test-on-return=false
spring.datasource.druid.test-while-idle=true
#配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒
spring.datasource.druid.time-between-eviction-runs-millis=60000
#置一个连接在池中最小生存的时间，单位是毫秒
spring.datasource.druid.min-evictable-idle-time-millis=25200000
#spring.datasource.druid.max-evictable-idle-time-millis=
#打开removeAbandoned功能，多少时间内必须关闭连接
spring.datasource.druid.removeAbandoned=true
#1800秒，也就是30分钟
spring.datasource.druid.remove-abandoned-timeout=1800
#<!-- 1800秒，也就是30分钟 --&gt;
spring.datasource.druid.log-abandoned=true
spring.datasource.druid.filters=mergeStat</pre>
```

在全局配置文件 `application.properties` 文件中配置以上的信息即可注入一个数据源到Spring Boot中。其实这仅仅是一种方式，下面介绍另外一种方式。

在自动配置类中 `DruidDataSourceAutoConfigure` 中有如下一段代码：

```
@Bean(initMethod = "init")
@ConditionalOnMissingBean
public DataSource dataSource() {
    LOGGER.info("Init DruidDataSource");
    return new DruidDataSourceWrapper();
}
```

`@ConditionalOnMissingBean` 和 `@Bean` 这两个注解的结合，意味着我们可以覆盖，只需要提前在 IOC 中注入一个 `DataSource` 类型的 Bean 即可。

因此我们在自定义的配置类中定义如下配置即可：

```

/**
 * @Bean: 向IOC容器中注入一个Bean
 * @ConfigurationProperties: 使得配置文件中以spring.datasource为前缀的属性映射到Bean的属性中
 * @return
 */
@ConfigurationProperties(prefix = "spring.datasource")
@Bean
public DataSource dataSource() {
    //做一些其他的自定义配置, 比如密码加密等.....
    return new DruidDataSource();
}

```

以上介绍了两种数据源的配置方式，第一种比较简单，第二种适合扩展，按需选择。

整合Mybatis

Spring Boot 整合Mybatis其实很简单，简单的几步就搞定，首先添加依赖：

```

<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>

```

第二步找到自动配置类 `MybatisAutoConfiguration`，有如下一行代码：

```
@EnableConfigurationProperties(MybatisProperties.class)
```

老套路了，全局配置文件中配置前缀为 `mybatis` 的配置将会映射到该类中的属性。

可配置的东西很多，比如 `XML文件的位置`、`类型处理器` 等等，如下简单的配置：

```

mybatis.type-handlers-package=com.demo.typehandler
mybatis.configuration.map-underscore-to-camel-case=true

```

如果需要通过包扫描的方式注入Mapper，则需要在配置类上加入一个注解：`@MapperScan`，其中的 `value` 属性指定需要扫描的包。

直接在全局配置文件配置各种属性是一种比较简单的方式，其实的任何组件的整合都有不少于两种的配置方式，下面来介绍下配置类如何配置。

`MybatisAutoConfiguration` 自动配置类有如下一段代码：

```

@Bean
@ConditionalOnMissingBean
public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {}

```

`@ConditionalOnMissingBean` 和 `@ExceptionHandler` 真是绝配档了，意味着我们又可以覆盖，只需要在IOC容器中注入 `SqlSessionFactory`（Mybatis六剑客之一生产者）。

在自定义配置类中注入即可，如下：

```
/**  
 * 注入SqlSessionFactory  
 */  
@Bean("sqlSessionFactory1")  
public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {  
    SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();  
    sqlSessionFactoryBean.setDataSource(dataSource);  
    sqlSessionFactoryBean.setMapperLocations(new  
        PathMatchingResourcePatternResolver().getResources("classpath*:mapper/**/*.xml"));  
    org.apache.ibatis.session.Configuration configuration = new  
        org.apache.ibatis.session.Configuration();  
    // 自动将数据库中的下划线转换为驼峰格式  
    configuration.setMapUnderscoreToCamelCase(true);  
    configuration.setDefaultFetchSize(100);  
    configuration.setDefaultStatementTimeout(30);  
    sqlSessionFactoryBean.setConfiguration(configuration);  
    return sqlSessionFactoryBean.getObject();  
}
```

以上介绍了配置Mybatis的两种方式，其实在大多数场景中使用第一种已经够用了，至于为什么介绍第二种呢？当然是为了多数据源的整合而做准备了。

在 `MybatisAutoConfiguration` 中有一行很重要的代码，如下：

```
@ConditionalOnSingleCandidate(DataSource.class)
```

`@ConditionalOnSingleCandidate` 这个注解的意思是当IOC容器中只有一个候选Bean的实例才会生效。

这行代码标注在Mybatis的自动配置类中有何含义呢？下面介绍，哈哈哈~



你仿佛在特意逗我笑

多数据源如何整合？

上文留下的问题：为什么的Mybatis自动配置上标注如下一行代码：

```
@ConditionalOnSingleCandidate(dataSource.class)
```

以上这行代码的言外之意：当IOC容器中只有一个数据源DataSource，这个自动配置类才会生效。

哦？照这样搞，多数据源是不能用Mybatis吗？

可能大家会有一个误解，认为多数据源就是多个的 `DataSource` 并存的，当然这样说也不是不正确。

多数据源的情况下并不是多个数据源并存的，Spring提供了 `AbstractRoutingDataSource` 这样一个抽象类，使得能够在多数据源的情况下任意切换，相当于一个**动态路由**的作用，作者称之为 **动态数据源**。因此Mybatis只需要配置这个动态数据源即可。

什么是动态数据源？

动态数据源简单的说就是能够自由切换的数据源，类似于一个动态路由的感觉，Spring 提供了一个抽象类 `AbstractRoutingDataSource`，这个抽象类中哟一个属性，如下：

```
private Map<Object, Object> targetDataSources;
```

`targetDataSources` 是一个 `Map` 结构，所有需要切换的数据源都存放在其中，根据指定的 `KEY` 进行切换。当然还有一个默认的数据源。

`AbstractRoutingDataSource` 这个抽象类中有一个抽象方法需要子类实现，如下：

```
protected abstract Object determineCurrentLookupKey();
```

`determineCurrentLookupKey()` 这个方法的返回值决定了需要切换的数据源的 `KEY`，就是根据这个 `KEY` 从 `targetDataSources` 取值（数据源）。

数据源切换如何保证线程隔离？

数据源属于一个公共的资源，在多线程的情况下如何保证线程隔离呢？不能我这边切换了影响其他线程的执行。

说到线程隔离，自然会想到 `ThreadLocal` 了，将切换数据源的 `KEY`（用于从 `targetDataSources` 中取值）存储在 `ThreadLocal` 中，执行结束之后清除即可。

单独封装了一个 `DataSourceHolder`，内部使用 `ThreadLocal` 隔离线程，代码如下：

```
/*
 * 使用ThreadLocal存储切换数据源后的KEY
 */
public class DataSourceHolder {
    private static final ThreadLocal<String> dataSources = new InheritableThreadLocal();
    //设置数据源
    public static void setDataSource(String datasource) {
```

```

        dataSources.set(dataSource);
    }

    //获取数据源
    public static String getDataSource() {
        return dataSources.get();
    }

    //清除数据源
    public static void clearDataSource() {
        dataSources.remove();
    }
}

```

如何构造一个动态数据源？

上文说过只需继承一个抽象类 `AbstractRoutingDataSource`，重写其中的一个方法 `determineCurrentLookupKey()` 即可。代码如下：

```

/**
 * 动态数据源，继承AbstractRoutingDataSource
 */
public class DynamicDataSource extends AbstractRoutingDataSource {

    /**
     * 返回需要使用的数据源的key，将会按照这个KEY从Map获取对应的数据源（切换）
     * @return
     */
    @Override
    protected Object determineCurrentLookupKey() {
        //从ThreadLocal中取出KEY
        return DataSourceHolder.getDataSource();
    }

    /**
     * 构造方法填充Map，构建多数据源
     */
    public DynamicDataSource(DataSource defaultTargetDataSource, Map<Object, Object>
targetDataSources) {
        //默认的数据源，可以作为主数据源
        super.setDefaultTargetDataSource(defaultTargetDataSource);
        //目标数据源
        super.setTargetDataSources(targetDataSources);
        //执行afterPropertiesSet方法，完成属性的设置
        super.afterPropertiesSet();
    }
}

```

上述代码很简单，分析如下：

1. 一个多参的构造方法，指定了默认的数据源和目标数据源。
2. 重写 `determineCurrentLookupKey()` 方法，返回数据源对应的 KEY，这里是直接从 `ThreadLocal` 中取值，就是上文封装的 `DataSourceHolder`。

定义一个注解

为了操作方便且低耦合，不能每次需要切换的数据源的时候都要手动调一下接口吧，可以定义一个切换数据源的注解，如下：

```
/**  
 * 切换数据源的注解  
 */  
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface SwitchSource {  
  
    /**  
     * 默认切换的数据源KEY  
     */  
    String DEFAULT_NAME = "hisDataSource";  
  
    /**  
     * 需要切换到数据的KEY  
     */  
    String value() default DEFAULT_NAME;  
}
```

注解中只有一个 `value` 属性，指定了需要切换数据源的 `KEY`。

有注解还不行，当然还要有切面，代码如下：

```
@Aspect  
//优先级设置到最高  
@Order(Ordered.HIGHEST_PRECEDENCE)  
@Component  
@Slf4j  
public class DataSourceAspect {  
  
    @Pointcut("@annotation(SwitchSource)")  
    public void pointcut() {  
    }  
  
    /**  
     * 在方法执行之前切换到指定的数据源  
     * @param joinPoint  
     */  
    @Before(value = "pointcut()")  
    public void beforeOpt(JoinPoint joinPoint) {  
        /*因为是对注解进行切面，所以这边无需做过多判定，直接获取注解的值，进行环绕，将数据源设置成远方，然后结束后，清楚当前线程数据源*/  
        Method method = ((MethodSignature) joinPoint.getSignature()).getMethod();  
        SwitchSource switchSource = method.getAnnotation(SwitchSource.class);  
        log.info("[Switch DataSource]: " + switchSource.value());  
        DataSourceHolder.setDataSource(switchSource.value());  
    }  
  
    /**  
     * 方法执行之后清除掉ThreadLocal中存储的KEY，这样动态数据源会使用默认的数据源  
     */  
    @After(value = "pointcut()")
```

```

    public void afterOpt() {
        DataSourceHolder.clearDataSource();
        log.info("[Switch Default DataSource]");
    }
}

```

这个 ASPECT 很容易理解，`beforeOpt()` 在方法之前执行，取值 `@SwitchSource` 中 `value` 属性设置到 `ThreadLocal` 中；`afterOpt()` 方法在方法执行之后执行，清除掉 `ThreadLocal` 中的 `KEY`，保证了如果不切换数据源，则用默认的数据源。

如何与Mybatis整合？

单一数据源与Mybatis整合上文已经详细讲解了，数据源 `DataSource` 作为参数构建了 `SqlSessionFactory`，同样的思想，只需要把这个数据源换成动态数据源即可。注入的代码如下：

```

/**
 * 创建动态数据源的SqlSessionFactory，传入的是动态数据源
 * @Primary这个注解很重要，如果项目中存在多个SqlSessionFactory，这个注解一定要加上
 */
@Primary
@Bean("sqlSessionFactory2")
public SqlSessionFactory sqlSessionFactoryBean(DynamicDataSource dynamicDataSource) throws
Exception {
    SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();
    sqlSessionFactoryBean.setDataSource(dynamicDataSource);
    org.apache.ibatis.session.Configuration configuration = new
org.apache.ibatis.session.Configuration();
    configuration.setMapUnderscoreToCamelCase(true);
    configuration.setDefaultFetchSize(100);
    configuration.setDefaultStatementTimeout(30);
    sqlSessionFactoryBean.setConfiguration(configuration);
    return sqlSessionFactoryBean.getObject();
}

```

与Mybatis整合很简单，只需要把数据源替换成自定义的动态数据源 `DynamicDataSource`。

那么动态数据源如何注入到IOC容器中呢？看上文自定义的 `DynamicDataSource` 构造方法，肯定需要两个数据源了，因此必须先注入两个或者多个数据源到IOC容器中，如下：

```

/**
 * @Bean：向IOC容器中注入一个Bean
 * @ConfigurationProperties：使得配置文件中以spring.datasource为前缀的属性映射到Bean的属性中
 */
@ConfigurationProperties(prefix = "spring.datasource")
@Bean("dataSource")
public DataSource dataSource() {
    return new DruidDataSource();
}

/**
 * 在IOC容器中再另外加一个数据源
 * 全局配置文件中前缀是spring.datasource.his
 */
@Bean(name = SwitchSource.DEFAULT_NAME)
@ConfigurationProperties(prefix = "spring.datasource.his")
public DataSource hisDataSource() {
}

```

```
        return DataSourceBuilder.create().build();
    }
```

以上构建的两个数据源，一个是**默认的数据源**，一个是**需要切换到的数据源**（`targetDataSources`），这样就组成了动态数据源了。数据源的一些信息，比如 `url`，`username` 需要自己在全局配置文件中根据指定的前缀配置即可，代码不再贴出。

动态数据源的注入代码如下：

```
/**
 * 创建动态数据源的SqlSessionFactory，传入的是动态数据源
 * @Primary这个注解很重要，如果项目中存在多个SqlSessionFactory，这个注解一定要加上
 */
@Primary
@Bean("sqlSessionFactory2")
public SqlSessionFactory sqlSessionFactoryBean(DynamicDataSource dynamicDataSource) throws
Exception {
    SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();
    sqlSessionFactoryBean.setDataSource(dynamicDataSource);
    org.apache.ibatis.session.Configuration configuration = new
org.apache.ibatis.session.Configuration();
    configuration.setMapUnderscoreToCamelCase(true);
    configuration.setDefaultFetchSize(100);
    configuration.setDefaultStatementTimeout(30);
    sqlSessionFactoryBean.setConfiguration(configuration);
    return sqlSessionFactoryBean.getObject();
}
```

这里还有一个问题：IOC中存在多个数据源了，那么事务管理器怎么办呢？它也懵逼了，到底选择哪个数据源呢？因此事务管理器肯定还是要重新配置的。

事务管理器此时管理的数据源将是动态数据源 `DynamicDataSource`，配置如下：

```
/**
 * 重写事务管理器，管理动态数据源
 */
@Primary
@Bean(value = "transactionManager2")
public PlatformTransactionManager annotationDrivenTransactionManager(DynamicDataSource dataSource)
{
    return new DataSourceTransactionManager(dataSource);
}
```

至此，Mybatis与多数据源的整合就完成了。

演示

使用也很简单，在需要切换数据源的方法上方标注 `@SwitchSource` 切换到指定的数据源即可，如下：

Watermark

```
//不开启事务  
@Transactional(propagation = Propagation.NOT_SUPPORTED)  
//切换到HIS的数据源  
@SwitchSource  
@Override  
public List<DeptInfo> list() {  
    return hisDeptInfoMapper.listDept();  
}
```

这样只要执行到这方法将会切换到 HIS 的数据源，方法执行结束之后将会清除，执行默认的数据源。

总结

本篇文章讲了Spring Boot与单数据源、Mybatis、多数据源之间的整合，希望这篇文章能够帮助读者理解多数据源的整合，虽说用的不多，但是在有些领域仍然是比较重要的。

原创不易，点点赞分享一波，谢谢支持~

源码已经上传，需要源码的朋友公众号回复关键词 多数据源。



整合JSR303实现数据校验

前言 Watermark

不知不觉 Spring Boot 专栏文章已经写到第十四章了，无论写的好与不好，作者都在尽力写的详细，写得与其它的文章不同，每一章都不是浅尝辄止。如果前面的文章没有看过的朋友，[点击这里前往](#)。

今天介绍一下 Spring Boot 如何优雅的整合 JSR-303 进行参数校验，说到参数校验可能都用过，但是你真的会用吗？网上的教程很多，大多是简单的介绍。

什么是 JSR-303?

JSR-303 是 JAVA EE 6 中的一项子规范，叫做 Bean Validation。

Bean Validation 为 JavaBean 验证定义了相应的 元数据模型 和 API。缺省的元数据是 Java Annotations，通过使用 XML 可以对原有的元数据信息进行覆盖和扩展。在应用程序中，通过使用 Bean Validation 或是你自己定义的 constraint，例如 @NotNull，@Max，@ZipCode，就可以确保数据模型（JavaBean）的正确性。constraint 可以附加到字段，getter 方法，类或者接口上面。对于一些特定的需求，用户可以很容易的开发定制化的 constraint。Bean Validation 是一个运行时的数据验证框架，在验证之后验证的错误信息会被马上返回。

添加依赖

Spring Boot整合JSR-303只需要添加一个 starter 即可，如下：

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

内嵌的注解有哪些？

Bean Validation 内嵌的注解很多，基本实际开发中已经够用了，注解如下：

Watermark

注解	详细信息
@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(value)	被注释的元素必须符合指定的正则表达式

以上是 Bean Validation 的内嵌的注解，但是 Hibernate Validator 在原有的基础上也内嵌了几个注解，如下。

注解	详细信息
@Email	被注释的元素必须是电子邮箱地址
@Length	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	被注释的元素必须在合适的范围内

如何使用？

参数校验分为简单校验、嵌套校验、分组校验。

简单校验

简单的校验即是没有嵌套属性，直接在需要的元素上标注约束注解即可。如下：

```
@Data
public class ArticleDO {
    @NotNull(message = "文章id不能为空")
    @Min(value = 1,message = "文章ID不能为负数")
    private Integer id;
```

```

    @NotBlank(message = "文章内容不能为空")
    private String content;

    @NotBlank(message = "作者Id不能为空")
    private String authorId;

    @Future(message = "提交时间不能为过去时间")
    private Date submitTime;
}

```

同一个属性可以指定多个约束，比如 `@NotNull` 和 `@MAX`，其中的 `message` 属性指定了约束条件不满足时的提示信息。

以上约束标记完成之后，要想完成校验，需要在 `controller` 层的接口标注 `@Valid` 注解以及声明一个 `BindingResult` 类型的参数来接收校验的结果。

下面简单的演示下添加文章的接口，如下：

```

/**
 * 添加文章
 */
@PostMapping("/add")
public String add(@Valid @RequestBody ArticleDTO articleDTO, BindingResult bindingResult) throws
JsonProcessingException {
    //如果有错误提示信息
    if (bindingResult.hasErrors()) {
        Map<String , String> map = new HashMap<>();
        bindingResult.getFieldErrors().forEach( (item) -> {
            String message = item.getDefaultMessage();
            String field = item.getField();
            map.put( field , message );
        } );
        //返回提示信息
        return objectMapper.writeValueAsString(map);
    }
    return "success";
}

```

仅仅在属性上添加了约束注解还不行，还需在接口参数上标注 `@Valid` 注解并且声明一个 `BindingResult` 类型的参数来接收校验结果。

分组校验

举个栗子：上传文章不需要传文章 `ID`，但是修改文章需要上传文章 `ID`，并且用的都是同一个 `DTO` 接收参数，此时的约束条件该如何写呢？

此时就需要对这个文章 `ID` 进行分组校验，上传文章接口是一个分组，不需要执行 `@NotNull` 校验，修改文章的接口是一个分组，需要执行 `@NotNull` 的校验。

所有的校验注解都有一个 `groups` 属性用来指定分组，`Class<?>[]` 类型，没有实际意义，因此只需要定义一个或者多个接口用来区分即可。

```

public class ArticleDTO {

    /**
     * 文章ID只在修改的时候需要检验，因此指定groups为修改的分组
     */
    @NotNull(message = "文章id不能为空", groups = UpdateArticleDTO.class)
    @Min(value = 1, message = "文章ID不能为负数", groups = UpdateArticleDTO.class)
    private Integer id;

    /**
     * 文章内容添加和修改都是必须校验的，groups需要指定两个分组
     */
    @NotBlank(message = "文章内容不能为空", groups = {AddArticleDTO.class, UpdateArticleDTO.class})
    private String content;

    @NotBlank(message = "作者Id不能为空", groups = AddArticleDTO.class)
    private String authorId;

    /**
     * 提交时间是添加和修改都需要校验的，因此指定groups两个
     */
    @Future(message = "提交时间不能为过去时间", groups = {AddArticleDTO.class, UpdateArticleDTO.class})
    private Date submitTime;

    //修改文章的分组
    public interface UpdateArticleDTO {}

    //添加文章的分组
    public interface AddArticleDTO {}

}

```

JSR303本身的 `@Valid` 并不支持分组校验，但是Spring在其基础提供了一个注解 `@Validated` 支持分组校验。`@Validated` 这个注解 `value` 属性指定需要校验的分组。

```

/**
 * 添加文章
 * @Validated: 这个注解指定校验的分组信息
 */
@PostMapping("/add")
public String add(@Validated(value = ArticleDTO.AddArticleDTO.class) @RequestBody ArticleDTO
articleDTO, BindingResult bindingResult) throws JsonProcessingException {
    //如果有错误提示信息
    if (bindingResult.hasErrors()) {
        Map<String , String> map = new HashMap<>();
        bindingResult.getFieldErrors().forEach( item -> {
            String message = item.getDefaultMessage();
            String field = item.getField();
            map.put( field , message );
        });
        /**
         * 返回提示信息
         */
        ObjectMapper mapper.writeValueAsString(map);
    }
    return "success";
}

```

嵌套校验

嵌套校验简单的解释就是一个实体中包含另外一个实体，并且这两个或者多个实体都需要校验。

举个栗子：文章可以有一个或者多个分类，作者在提交文章的时候必须指定文章分类，而分类是单独一个实体，有 分类ID、名称 等等。大致的结构如下：

```
public class ArticleDTO {  
    ... 文章的一些属性.....  
  
    //分类的信息  
    private CategoryDTO categoryDTO;  
}
```

此时文章和分类的属性都需要校验，这种就叫做嵌套校验。

嵌套校验很简单，只需要在嵌套的实体属性标注 `@Valid` 注解，则其中的属性也将得到校验，否则不会校验。

如下文章分类实体类校验：

```
/**  
 * 文章分类  
 */  
@Data  
public class CategoryDTO {  
    @NotNull(message = "分类ID不能为空")  
    @Min(value = 1, message = "分类ID不能为负数")  
    private Integer id;  
  
    @NotBlank(message = "分类名称不能为空")  
    private String name;  
}
```

文章的实体类中有个嵌套的文章分类 `CategoryDTO` 属性，需要使用 `@Valid` 标注才能嵌套校验，如下：

```
@Data  
public class ArticleDTO {  
  
    @NotBlank(message = "文章内容不能为空")  
    private String content;  
  
    @NotBlank(message = "作者Id不能为空")  
    private String authorId;  
  
    @Future(message = "提交时间不能为过去时间")  
    private Date submitTime;  
  
    /**  
     * @Valid这个注解指定CategoryDTO中的属性也需要校验  
     */  
    @Valid  
    @NotNull(message = "分类不能为空")  
    private CategoryDTO categoryDTO;  
}
```

`Controller` 层的添加文章的接口同上，需要使用 `@Valid` 或者 `@Validated` 标注入参，同时需要定义一个 `BindingResult` 的参数接收校验结果。

嵌套校验针对**分组查询**仍然生效，如果嵌套的实体类（比如 `CategoryDTO`）中的校验的属性和接口中 `@Validated` 注解指定的分组不同，则不会校验。

`JSR-303` 针对**集合**的嵌套校验也是可行的，比如 `List` 的嵌套校验，同样需要在属性上标注一个 `@Valid` 注解才会生效，如下：

```
@Data  
public class ArticleDTO {  
    /**  
     * @Valid这个注解标注在集合上，将会针对集合中每个元素进行校验  
     */  
    @Valid  
    @Size(min = 1, message = "至少一个分类")  
    @NotNull(message = "分类不能为空")  
    private List<CategoryDTO> categoryDTOS;  
}
```

总结：嵌套校验只需要在需要校验的元素（单个或者集合）上添加 `@Valid` 注解，接口层需要使用 `@Valid` 或者 `@Validated` 注解标注入参。

如何接收校验结果？

接收校验的结果的方式很多，不过实际开发中最好选择一个优雅的方式，下面介绍常见的两种方式。

BindingResult 接收

这种方式需要在 `Controller` 层的每个接口方法参数中指定，Validator会将校验的信息自动封装到其中。这也是上面例子中一直用的方式。如下：

```
@PostMapping("/add")  
public String add(@Valid @RequestBody ArticleDTO articleDTO, BindingResult bindingResult) {}
```

这种方式的弊端很明显，每个接口方法参数都要声明，同时每个方法都要处理校验信息，显然不现实，舍弃。

此种方式还有一个优化的方案：使用 `AOP`，在 `Controller` 接口方法执行之前处理 `BindingResult` 的消息提示，不过这种方案仍然**不推荐使用**。

全局异常捕捉

参数在校验失败的时候会抛出的 `MethodArgumentNotValidException` 或者 `BindException` 两种异常，可以在全局的异常处理器中捕捉到这两种异常，将提示信息或者自定义信息返回给客户端。

全局异常捕捉是写过一个打了无数补丁的可以看 **满屏的try-catch，你不疼得慌？**。

作者这里就不再详细的贴出其他的异常捕获了，仅仅贴一下参数校验的异常捕获（**仅仅举个例子，具体的返回信息需要自己封装**），如下：

```
@RestControllerAdvice
```

```

public class ExceptionRsHandler {

    @Autowired
    private ObjectMapper objectMapper;

    /**
     * 参数校验异常步骤
     */
    @ExceptionHandler(value= {MethodArgumentNotValidException.class , BindException.class})
    public String onException(Exception e) throws JsonProcessingException {
        BindingResult bindingResult = null;
        if (e instanceof MethodArgumentNotValidException) {
            bindingResult = ((MethodArgumentNotValidException)e).getBindingResult();
        } else if (e instanceof BindException) {
            bindingResult = ((BindException)e).getBindingResult();
        }
        Map<String, String> errorMap = new HashMap<>(16);
        bindingResult.getFieldErrors().forEach((fieldError)→
            errorMap.put(fieldError.getField(), fieldError.getDefaultMessage())
        );
        return objectMapper.writeValueAsString(errorMap);
    }
}

```

spring-boot-starter-validation做了什么？

这个启动器的自动配置类是 `ValidationAutoConfiguration`，最重要的代码就是注入了一个 `Validator`（校验器）的实现类，代码如下：

```

@Bean
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
@ConditionalOnMissingBean(Validator.class)
public static LocalValidatorFactoryBean defaultValidator() {
    LocalValidatorFactoryBean factoryBean = new LocalValidatorFactoryBean();
    MessageInterpolatorFactory interpolatorFactory = new MessageInterpolatorFactory();
    factoryBean.setMessageInterpolator(interpolatorFactory.getObject());
    return factoryBean;
}

```

这个有什么用呢？`Validator` 这个接口定义了校验的方法，如下：

```

<T> Set<ConstraintViolation<T>> validate(T object, Class<?>... groups);

<T> Set<ConstraintViolation<T>> validateProperty(T object,
    String propertyName,
    Class<?>... groups);

<T> Set<ConstraintViolation<T>> validateValue(Class<T> beanType,
    String propertyName,
    Object value,
    Class<?>... groups);

.....

```

这个 Validator 可以用来自定义实现自己的校验逻辑，有些大公司完全不用JSR-303提供的 @Valid 注解，而是有一套自己的实现，其实本质就是利用 Validator 这个接口的实现。

如何自定义校验？

虽说在日常的开发中内置的约束注解已经够用了，但是仍然有些时候不能满足需求，需要自定义一些校验约束。

举个栗子：有这样一个例子，传入的数字要在列举的值范围内，否则校验失败。

自定义校验注解

首先需要自定义一个校验注解，如下：

```
@Documented
@Constraint(validatedBy = { EnumValuesConstraintValidator.class })
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@NotNull(message = "不能为空")
public @interface EnumValues {
    /**
     * 提示消息
     */
    String message() default "传入的值不在范围内";

    /**
     * 分组
     * @return
     */
    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * 可以传入的值
     * @return
     */
    int[] values() default { };
}
```

根据 Bean Validation API 规范的要求有如下三个属性是必须的：

1. `message`：定义消息模板，校验失败时输出
2. `groups`：用于校验分组
3. `payload`：Bean Validation API 的使用者可以通过此属性来给约束条件指定严重级别。这个属性并不被API自身所使用。

除了以上三个必须要的属性，添加了一个 `values` 属性用来接收限制的范围。

该校验注解上标注的如下一行语句。

```
@Constraint(validatedBy = { EnumValuesConstraintValidator.class })
```

这个 `@Constraint` 注解指定了通过哪个校验器去校验。

自定义校验注解可以复用内嵌的注解，比如 `@EnumValues` 注解头上标注了一个 `@NotNull` 注解，这样 `@EnumValues` 就兼具了 `@NotNull` 的功能。

自定义校验器

`@Constraint` 注解指定了校验器为 `EnumValuesConstraintValidator`，因此需要自定义一个。

自定义校验器需要实现 `ConstraintValidator<A extends Annotation, T>` 这个接口，第一个泛型是 `校验注解`，第二个是 `参数类型`。代码如下：

```
/*
 * 校验器
 */
public class EnumValuesConstraintValidator implements ConstraintValidator<EnumValues, Integer> {
    /**
     * 存储枚举的值
     */
    private Set<Integer> ints=new HashSet<>();

    /**
     * 初始化方法
     * @param enumValues 校验的注解
     */
    @Override
    public void initialize(EnumValues enumValues) {
        for (int value : enumValues.values()) {
            ints.add(value);
        }
    }

    /**
     *
     * @param value 入参传的值
     * @param context
     * @return
     */
    @Override
    public boolean isValid(Integer value, ConstraintValidatorContext context) {
        //判断是否包含这个值
        return ints.contains(value);
    }
}
```

如果约束注解需要对其他数据类型进行校验，则可以的自定义对应数据类型的校验器，然后在约束注解头上的 `@Constraint` 注解中指定其他的校验器。

演示

校验注解在验证器定义成功之后即可使用，如下：

```
@Data
public class AuthorDTO {
    @EnumValues(values = {1,2}, message = "性别只能传入1或者2")
    private Integer gender;
}
```

总结

数据校验作为客户端和服务端的一道屏障，有着重要的作用，通过这篇文章希望能够对 JSR-303 数据校验有着全面的认识。

另外作者的第一本PDF书籍已经整理好了，由浅入深的详细介绍了Mybatis基础以及底层源码，有需要的朋友公众号回复关键词 **Mybatis进阶** 即可获取，目录如下：



Watermark
如何开启远程调试？

前言

上周末一个朋友庆生，无意间听他说起了近况，说公司项目太多了，每天一堆BUG需要修复，项目来回切换启动，真是挺烦的。

随着项目越来越多，特别是身处外包公司的朋友，每天可能需要切换两三个项目，难道一有问题就本地启动项目调试？

今天这篇文章就来介绍一下什么是远程调试，[Spring Boot](#) 如何开启远程调试？

什么是远程调试？

所谓的远程调试就是服务端程序运行在一台远程服务器上，我们可以在本地服务端的代码（**前提是本地的代码必须和远程服务器运行的代码一致**）中设置断点，每当有请求到远程服务器时时能够在本地知道远程服务端的此时的内部状态。

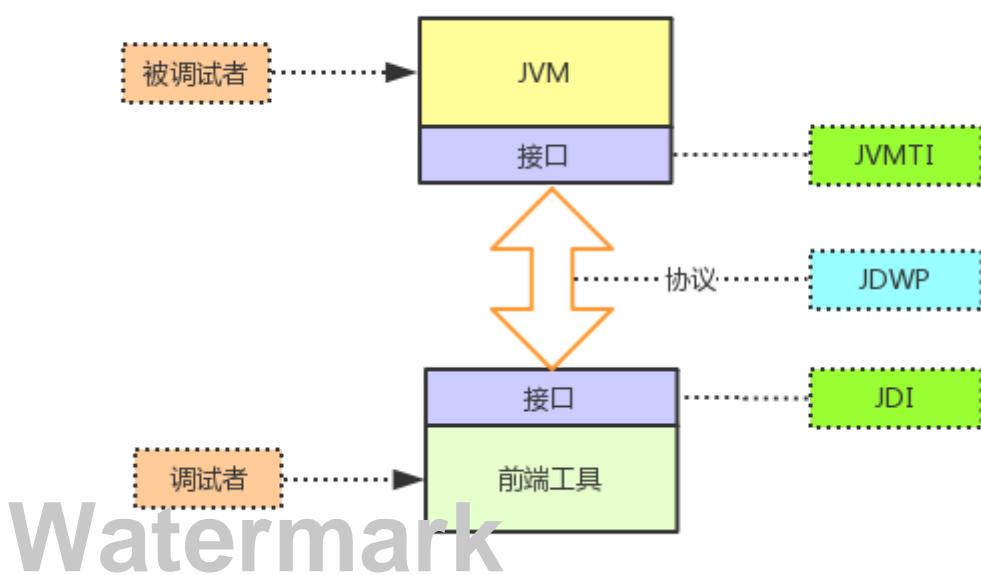
简单的意思：本地无需启动项目的状态下能够实时调试服务端的代码。

为什么要远程调试？

随着项目的体量越来越大，启动的时间的也是随之增长，何必为了调试一个BUG花费十分钟的时间去启动项目呢？你不怕老大骂你啊？

什么是JPDA？

JPDA ([Java Platform Debugger Architecture](#))，即 Java 平台调试体系，具体结构图如下图所示：



其中实现调试功能的主要协议是 JDWP 协议，在 [Java SE 5](#) 以前版本，JVM 端的实现接口是 JVMPPI ([Java Virtual Machine Profiler Interface](#))，而在 [Java SE 5](#) 及以后版本，使用 JVMTI ([Java Virtual Machine Tool Interface](#)) 来替代 JVMPPI。

因此，如果你使用的是 Java SE 5 之前的版本，则使用的调试命令格式如下：

```
java -Xdebug -Xrunjdwp:...
```

如果你使用的是 Java SE 5 之后的版本，则使用的命令格式如下：

```
java -agentlib:jdwp=...
```

如何开启调试？

由于现在使用的大多数都是 Java SE 5 之后的版本，则之前的就忽略了。

日常开发中最常见的开启远程调试的命令如下：

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=9093 -jar xxx.jar
```

前面的 `java -agentlib:jdwp=` 是基础命令，后面跟着的一串命令则是可选的参数，具体什么意思呢？下面详细介绍。

transport

指定运行的被调试应用和调试者之间的通信协议，有如下可选值：

1. `dt_socket`：采用 `socket` 方式连接（常用）
2. `dt_shmem`：采用共享内存的方式连接，支持有限，仅仅支持 windows 平台

server

指定当前应用作为调试服务端还是客户端，默认的值为 `y`（客户端）。

如果你想将当前应用作为被调试应用，设置该值为 `y`；如果你想将当前应用作为客户端，作为调试的发起者，设置该值为 `n`。

suspend

当前应用启动后，是否阻塞应用直到被连接，默认值为 `y`（阻塞）。

大部分情况下这个值应该为 `n`，即不需要阻塞等待连接。一个可能为 `y` 的应用场景是，你的程序在启动时出现了一个故障，为了调试，必须等到调试方连接上来后程序再启动。

address

对外暴露的端口，默认值是 `8000`

注意：此端口不能和项目同一个端口，且未被占用以及对外开放。

onthrow

这个参数的意思是当程序抛出指定异常时，则中断调试。

Watermark

onuncaught

当程序抛出未捕获异常时，是否中断调试，默认值为 `n`。

launch

当调试中断时，执行的程序。

timeout

超时时间，单位 `ms` (毫秒)

当 `suspend = y` 时，该值表示等待连接的超时；当 `suspend = n` 时，该值表示连接后的使用超时。

常用的命令

下面列举几个常用的参考命令，这样更加方便理解。

1. 以 `Socket` 方式监听 `8000` 端口，程序启动阻塞（`suspend` 的默认值为 `y`）直到被连接，命令如下：

```
-agentlib:jdwp=transport=dt_socket,server=y,address=8000
```

2. 以 `Socket` 方式监听 `8000` 端口，当程序启动后 `5` 秒无调试者连接的话终止，程序启动阻塞（`suspend` 的默认值为 `y`）直到被连接。

```
-agentlib:jdwp=transport=dt_socket,server=y,address=localhost:8000,timeout=5000
```

3. 选择可用的共享内存连接地址并使用 `stdout` 打印，程序启动不阻塞。

```
-agentlib:jdwp=transport=dt_shmem,server=y,suspend=n
```

4. 以 `socket` 方式连接到 `myhost:8000` 上的调试程序，在连接成功前启动阻塞。

```
-agentlib:jdwp=transport=dt_socket,address=myhost:8000
```

5. 以 `Socket` 方式监听 `8000` 端口，程序启动阻塞（`suspend` 的默认值为 `y`）直到被连接。当抛出 `IOException` 时中断调试，转而执行 `usr/local/bin/debugstub` 程序。

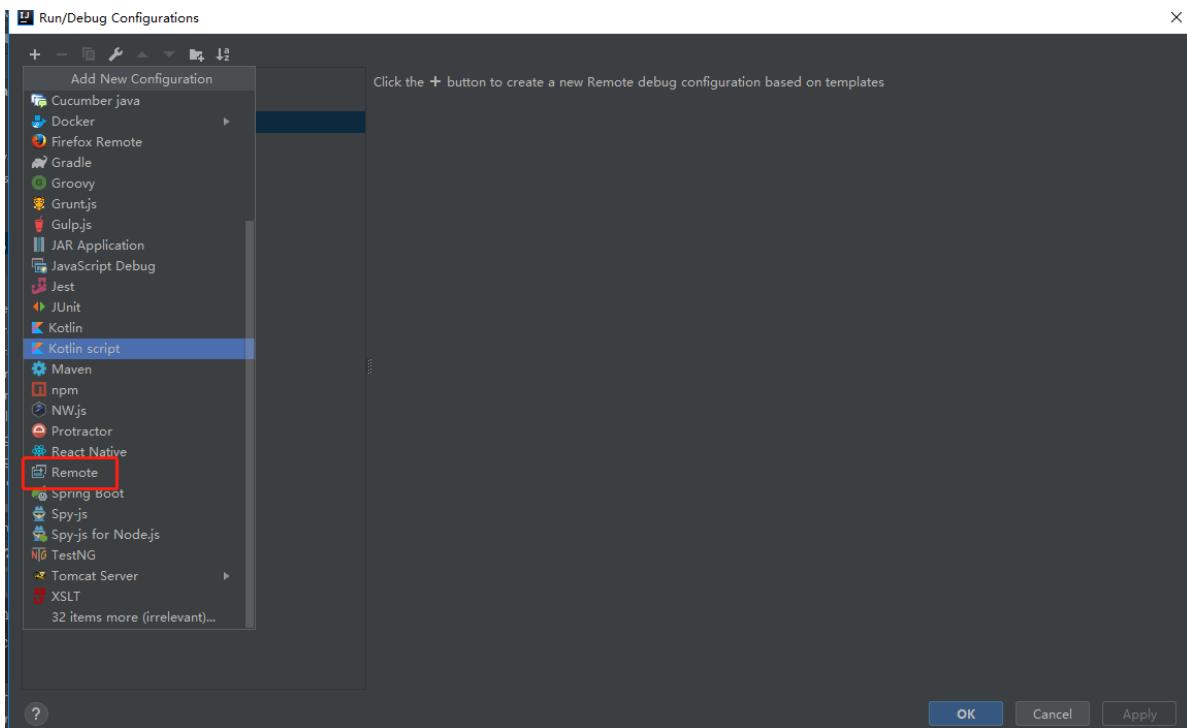
```
-agentlib:jdwp=transport=dt_socket,server=y,address=8000,onthrow=java.io.IOException,launch=/usr/local/bin/debugstub
```

IDEA如何开启远程调试？

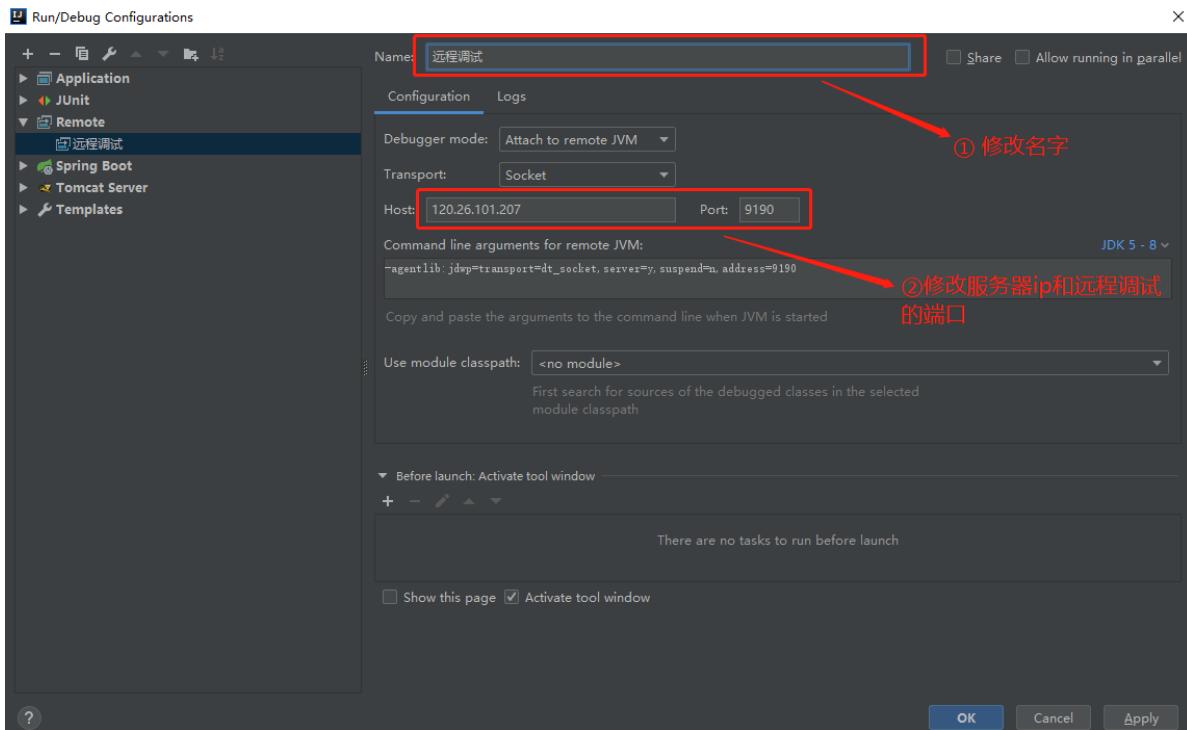
首先的将打包后的 `Spring Boot` 项目在服务器上运行，执行如下命令（各种参数根据实际情况自己配置）：

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=9193 -jar debug-demo.jar
```

项目启动成功后，点击 `Edit Configurations`，在弹框中点击 `+` 号，然后选择 `Remote`。

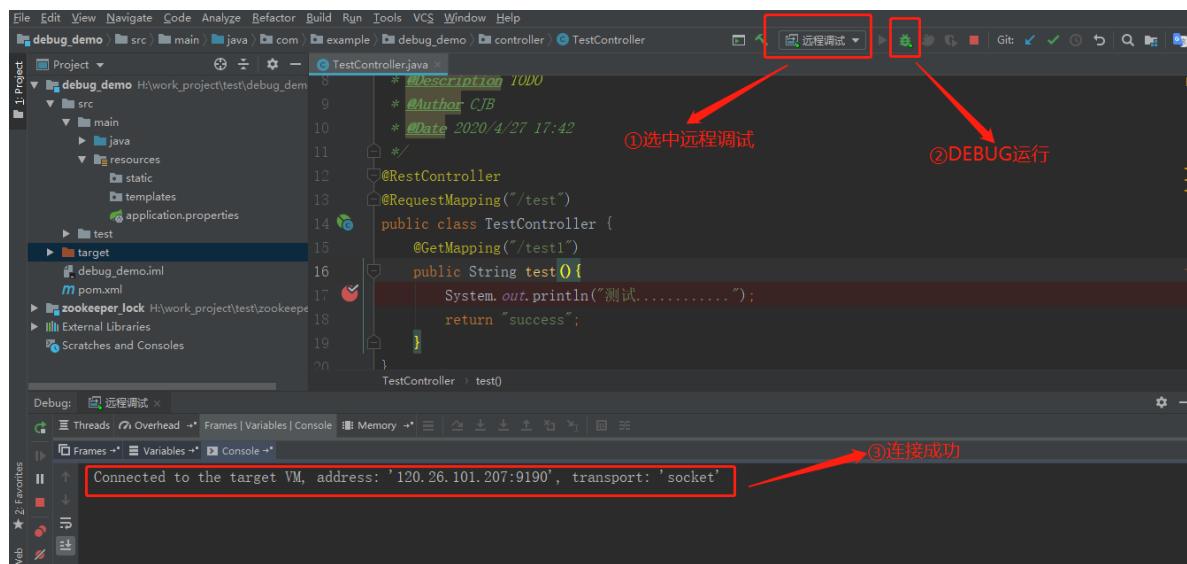


然后填写服务器的地址及端口，点击 **OK** 即可。

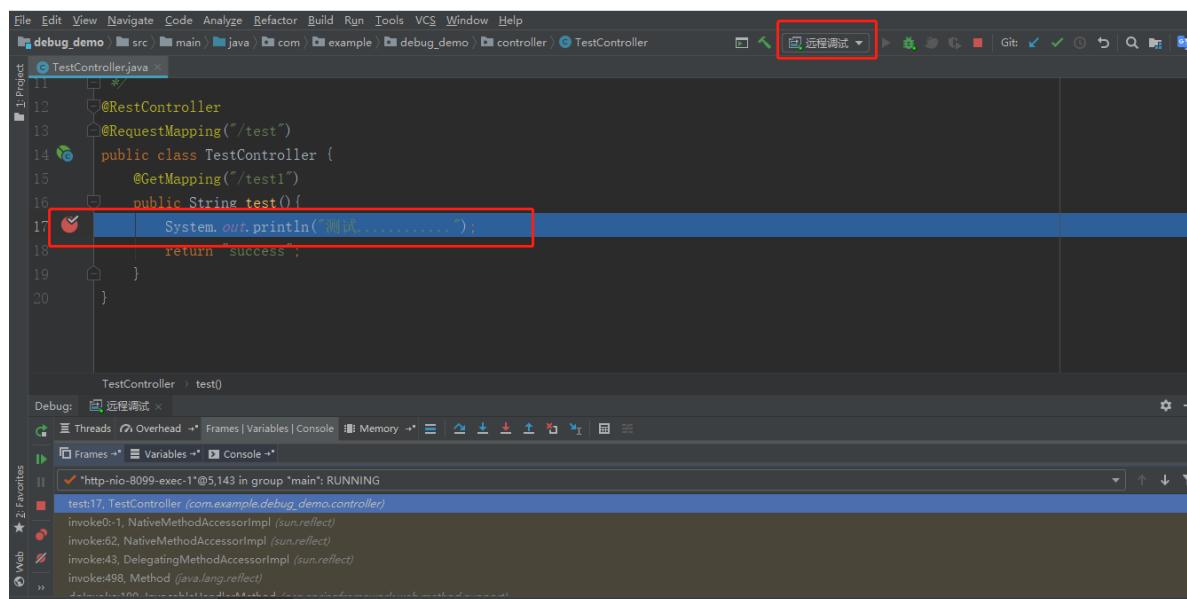


以上步骤配置完成后，点击DEBUG调试运行即可。

Watermark



配置完毕后点击保存即可，因为我配置的 `suspend=n`，因此服务端程序无需阻塞等待我们的连接。我们点击 IDEA 调试按钮，当我访问某一接口时，能够正常调试。

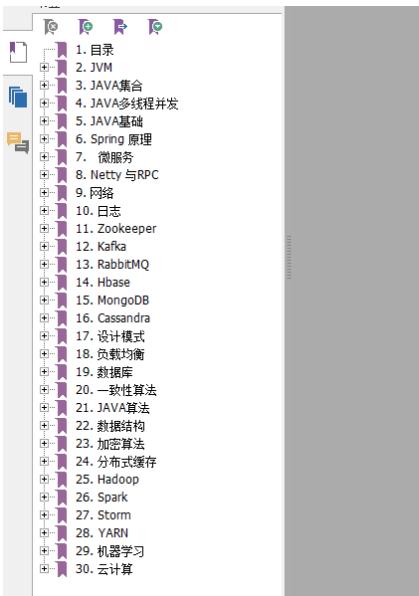


总结

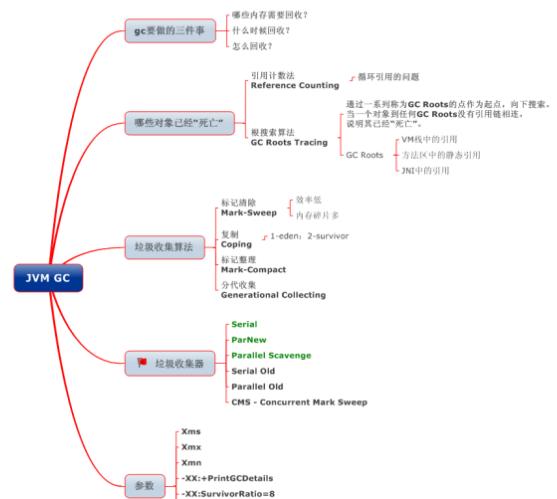
每天一个小知识，今天你学到了吗？

另外作者为大家准备接近 10M 的面筋，涵盖后端的各个层面，老规矩，公众号内回复 [Java面试宝典](#) 即可获取。

Watermark



2.4. 垃圾回收与算法



热部署只知道devtools吗？JRebel不香吗？

前言

Spring Boot 中的热部署相信大家用的最多的就是 devtools，没办法，官推的。

JRebel 相对于 devtools，个人觉得无论是加载速度还是使用便捷，JRebel 完胜。

作为前辈级别的开发利器，JRebel 真的值得开一章节来好好介绍下。

JRebel收费怎么破？

前面作者单独写过一篇激活 JRebel 的文章教程，没钱的可以去看看：[撸了个反向代理工具，搞一搞 JRebel](#)。

特此声明：作者支持原版，不差钱的建议装个原版的，毕竟这么好的工具值得。

什么是本地热部署？

传统的开发中，项目在启动过程中代码有所改动是不会重新编译运行的，而是要关闭项目重新启动后修改的代码才会生效。

本地热部署则是能够在项目运行中感知到特定文件代码的修改而使项目不重新启动就能生效。

什么是远程热部署？

远程热部署的 **远程** 两字指的是**远程服务器**，平时开发中，只要本地代码改动了，必须要重新打包上传服务器重新启动之后才会生效，**你这样干过吗？.....**



宝宝，我懂你

远程热部署则是本地代码改变之后，不用重新打包上传服务器重启项目就能生效，本地改变之后能够自动改变服务器上的项目代码。

有些人听到这里懵逼了，这是什么鬼？还有这么神奇的东西.....

Watermark

一脸懵逼 你说啥？



JRebel和devtools的区别

前辈和后辈的比较其实没什么可比性，如果不是JRebel收费了，绝对是所有程序员的首选。但还是要说说他们之间的区别，如下：

1. JRebel 加载的速度优于 devtools
2. JRebel不仅仅局限于Spring Boot项目，可以用在任何的Java项目中。
3. devtools 方式的热部署在功能上有限制，方法内的修改可以实现热部署，但新增的方法或者修改方法参数之后热部署是不生效的。

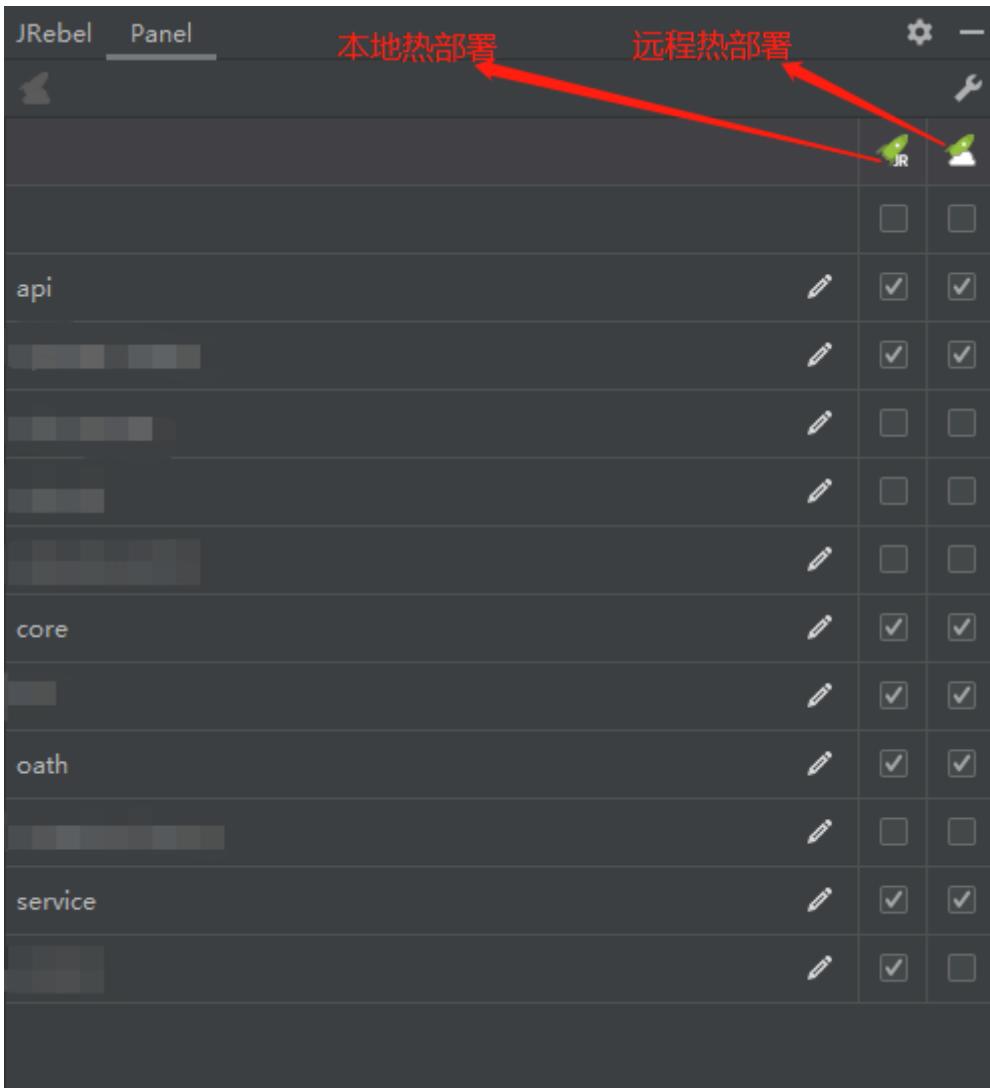
如何安装JRebel?

本地热部署只需要在 IDEA 中装一个JRebel的插件，远程热部署需要在服务器上装一个JRebel，这两种方式在上一篇文章都介绍过，不会的可以去看看：[撸了个反向代理工具，搞一搞JRebel](#)。

如何本地热部署？

JRebel 插件安装完成之后，将 IDEA 中的 自动编译 开启，然后找到 IDEA 中的 JRebel 的工具面板，将所需要热部署的项目或者模块勾选上即可，如下图：

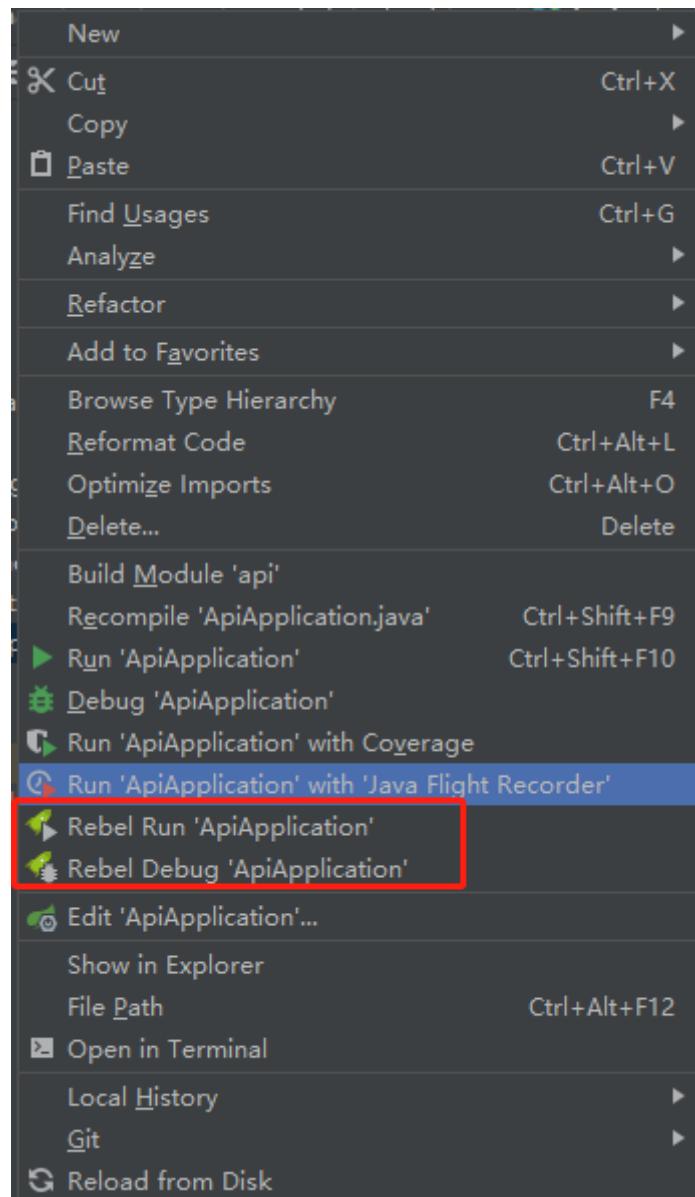
Watermark



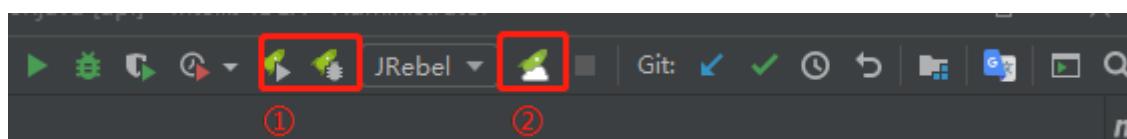
勾选成功之后将会在项目或者模块的 `src/resource` 下生成一个 `rebel.xml` 文件。

此时在 `Spring Boot` 的主启动类上右键，将会出现以 `JRebel` 启动的选项，如下图：

Watermark



当然在 IDEA 的右上角也存在启动的按钮，如下图：



① 是本地启动和 DEBUG 模式启动，② 是远程热部署的时候更新按钮。

此时就已经配置成功了，如果勾选的项目或者模块出现了改变，按 **CTRL+SHIFT+F9** 则会自动重新编译加载改变的部分，不用再重新启动项目了。

如何远程热部署？

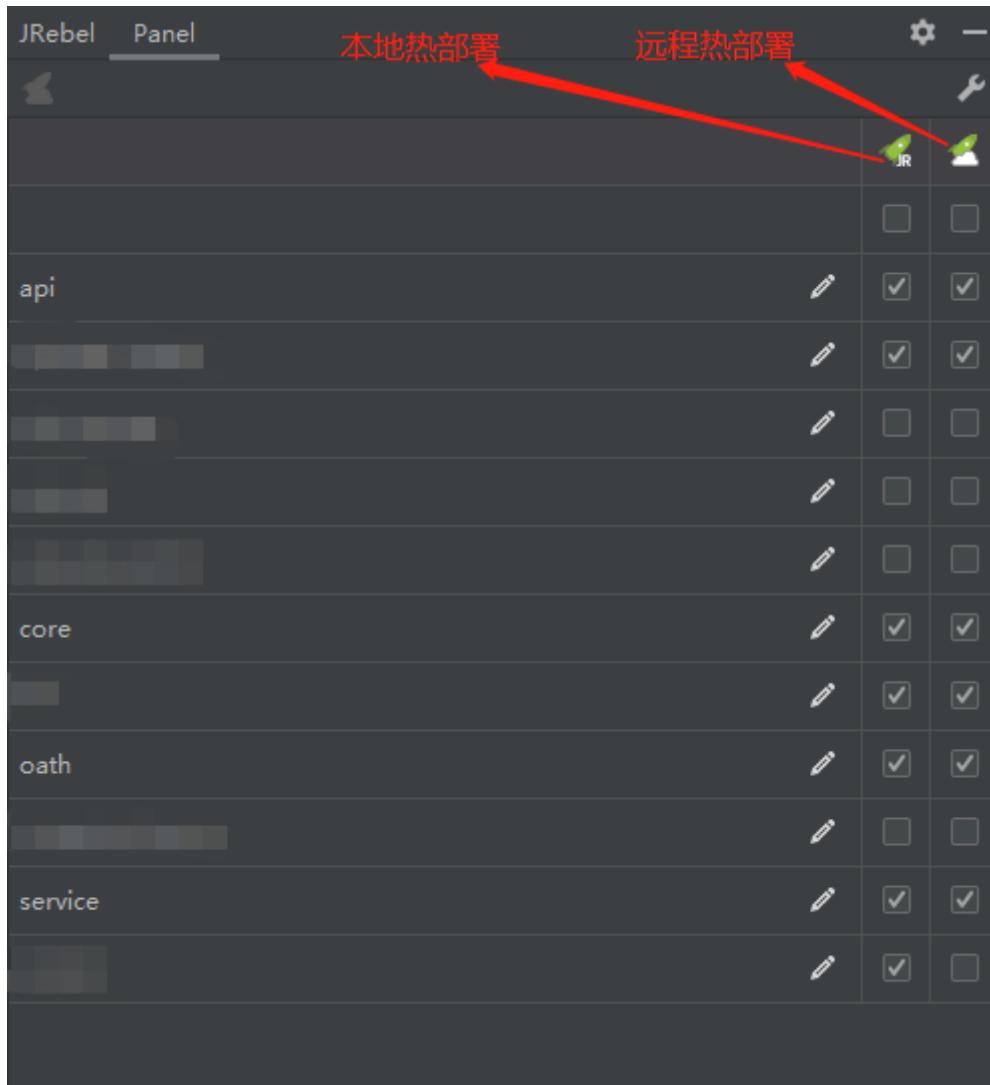
远程热部署需要在服务器上安装并激活 **JRebel**，参照上篇文章：[撸了个反向代理工具，搞一搞 JRebel](#)

激活成功后需要设置远程连接的密码，在 **JRebel** 的根目录下执行以下命令：

```
java -jar jrebel.jar -set-remote-password 123456789
```

此处设置的 123456789 则是远程的密码，在 IDEA 连接服务器的时候需要。

服务器配置成功后，在IDEA中JRebel的面板中设置远程热部署的模块，如下图：



勾选成功后，将会在 `src/resource` 下生成一个 `rebel-remote.xml` 文件。

此时将 Spring Boot 项目打包成一个 Jar，上传到服务器，执行以下命令启动项目：

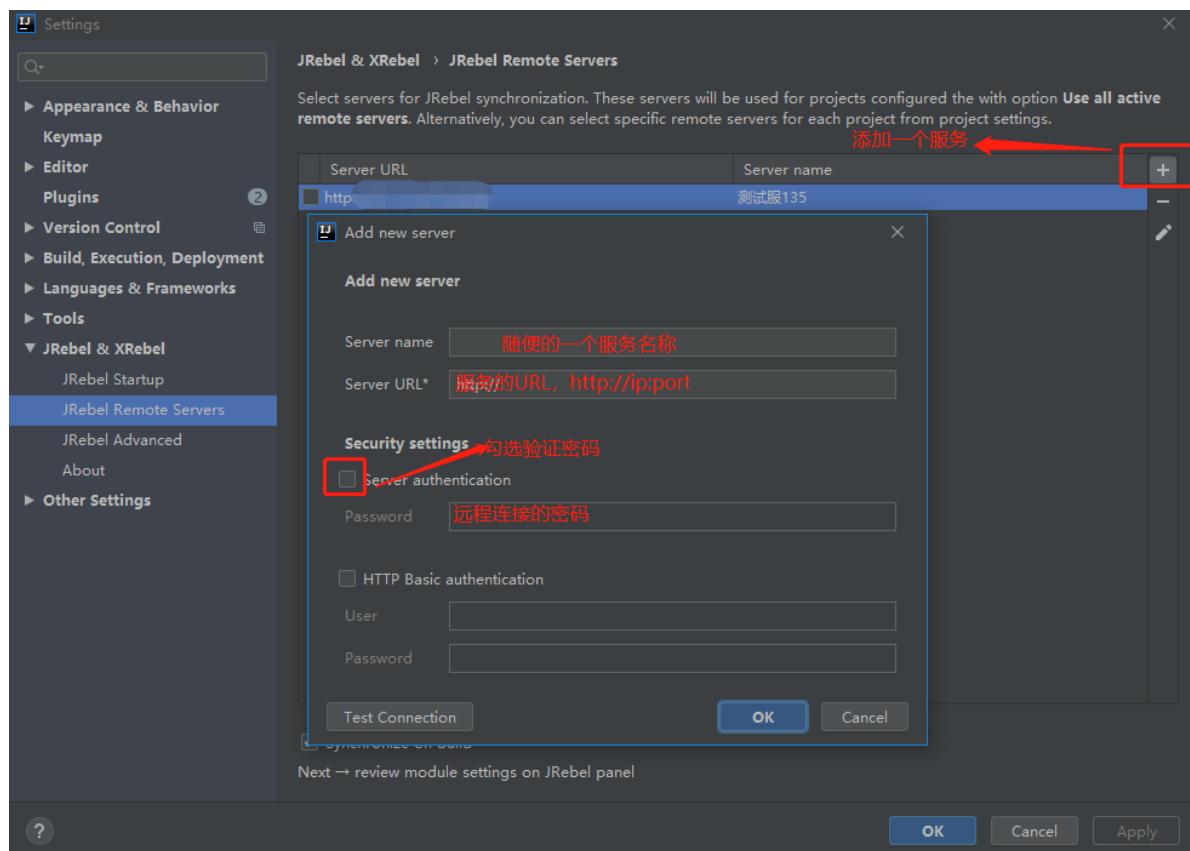
```
nohup java -agentpath:/usr/local/jrebel/lib/libjrebel64.so -Drebel.remoting_plugin=true -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=9083 -jar xxx.jar &
```

`libjrebel64.so` 这个文件是 JRebel 的 `lib` 目录下的文件。

`-Xdebug` 之后，`-jar` 之前的命令是开启远程调试的，如果不需要的可以去掉，不知道远程调试的，可以看：**惊呆了！Spring Boot还能开启远程调试~**。

项目启动成功后，服务器上的配置就完成了。

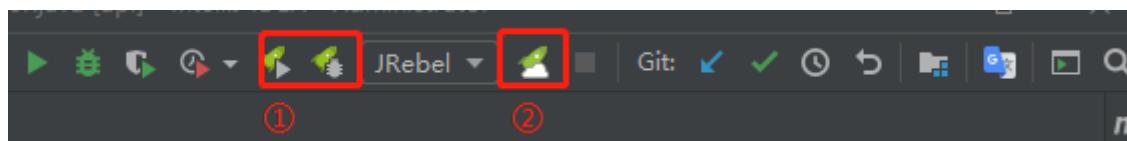
此时在IDEA中需要设置连接到刚才启动的项目，打开 `File->setting->JRebel&XRebel->JRebel Remote Servers`，如下图



步骤如下：

1. 点击 + 号添加一个服务
2. 填写信息
 - **server name** 随便起个服务的名字
 - **server URL** 格式：`http://ip:port`，这里的 `ip` 是服务器的IP，`port` 是项目端口号。
 - 远程密码则是上文设置的 JRebel 的密码 `123456789`。
3. 点击 **OK**，即可添加成功。

以上设置成功后，点击右上角的远程部署按钮，下图中的 ② 号按钮，则会自动更新服务器上已启动项目的代码使之本地修改在服务端自动生效：



在 **JRebel Console** 这个面板中将会打印出远程热部署更新的日志信息，如下图：

```

JRebel Console
[2020-11-01 23:09:59] [Project service, ] Connecting to server to sync project
[2020-11-01 23:09:59] [Project service, ] Project requires full synchronization
[2020-11-01 23:09:59] [Project service, ] Using directory 'H:\work_project'
[2020-11-01 23:09:59] [Project service, ] Upload succeeded in 137 ms. Transaction took 609 ms.
[2020-11-01 23:09:59] [Project oath, ] Connecting to server to sync project
[2020-11-01 23:09:59] [Project oath, ] Project requires full synchronization
[2020-11-01 23:09:59] [Project oath, ] Using directory 'H:\work_project'
[2020-11-01 23:10:00] [Project oath, ] Upload succeeded in 100 ms. Transaction took 340 ms.
[2020-11-01 23:10:00] [Project his, ] Connecting to server to sync project
[2020-11-01 23:10:00] [Project his, ] Project requires full synchronization

```

只要本地有了更改，点击远程热部署按钮，则会自动上传代码到服务器端并实时更新，不用重新启动项目。

多模块开发的一个坑

如果是多模块开发，比如分为 `api`（最终的 Jar 包），`core`（核心包），`service`（业务层的包），最终打包运行在服务器端的是 `api` 这个模块，其余两个模块都是属于依赖模块，虽然在 JRebel 远程热部署选项中都勾选了，但是它们的代码更改并不会在服务端生效。

这个如何解决呢？很简单，在 `api` 项目下的 `rebel-remote.xml` 文件中将其余两个模块添加进去，默认的如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<rebel-remote xmlns="http://www.zeroturnaround.com/rebel/remote">
    <id>xx.xx.xx.api</id>
</rebel-remote>
```

添加之后的代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<rebel-remote xmlns="http://www.zeroturnaround.com/rebel/remote">
    <id>xx.xxxx.xx.api</id>
    <id>xx.xx.xx.service</id>
    <id>xx.xx.xx.core</id>
</rebel-remote>
```

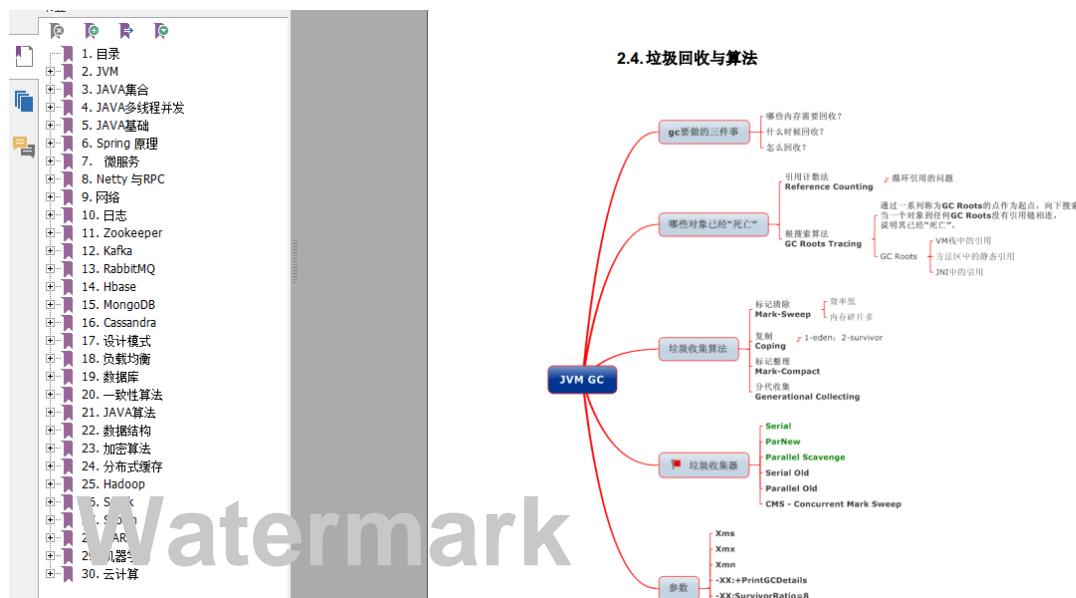
以上的 `<id>` 标签中指定的是模块的包名（package）。

总结

作为热部署界的前辈，JRebel 依然是敌得过后浪，果然是姜还是老的辣.....

希望这篇文章介绍的 JRebel 能够提高读者们的开发效率，反正我是提高了，哈哈~

另外作者为大家准备接近 10M 的面筋，涵盖后端的各个层面，老规矩，公众号内回复 Java面试宝典 即可获取。





整合Swagger3.0

前言

最近频繁被 Swagger 3.0 刷屏，官方表示这是一个突破性的变更，有很多的亮点，我还真不太相信，今天来带大家尝尝鲜，看看这碗汤到底鲜不鲜....

官方文档如何说？

该项目开源在 [Github](https://github.com/springfox/springfox) 上，地址：<https://github.com/springfox/springfox>。

Swagger 3.0 有何改动？官方文档总结如下几点：

1. 删除了对 `springfox-swagger2` 的依赖
2. 删除所有 `@EnableSwagger2...` 注解
3. 添加了 `springfox-boot-starter` 依赖项
4. 移除了 `guava` 等第三方依赖
5. 文档访问地址改变了，改成了 <http://ip:port/project/swagger-ui/index.html>。

姑且看到这里，各位小伙伴们有何看法？

The screenshot shows the Swagger UI interface. At the top, there's a header with the Swagger logo and a dropdown menu labeled "Select a definition" with "后台管理" selected. Below the header, the title "API的标题" is displayed with "2.0 OAS3" badges. A URL "http://localhost:8080/demo/v3/api-docs?group=后台管理" is shown. The main content area contains a "描述" (Description) section with placeholder text "这是一条描述" and "联系人姓名 - Website" followed by "Send email to 联系人姓名". Below this, there's a "Servers" dropdown set to "http://localhost:8080 - Inferred Url" and an "Authorize" button with a lock icon. Under the "article-controller" section, there's a "Article Controller" entry with a "POST /demo/article/add 添加文章" button.

既然人家更新出来了，咱不能不捧场，下面就介绍下 Spring Boot 如何整合 Swagger 3.0 吧。

Spring Boot版本说明

作者使用 Spring Boot 的版本是 2.3.5.RELEASE

添加依赖

Swagger 3.0 已经有了与Spring Boot整合的启动器，只需要添加以下依赖：

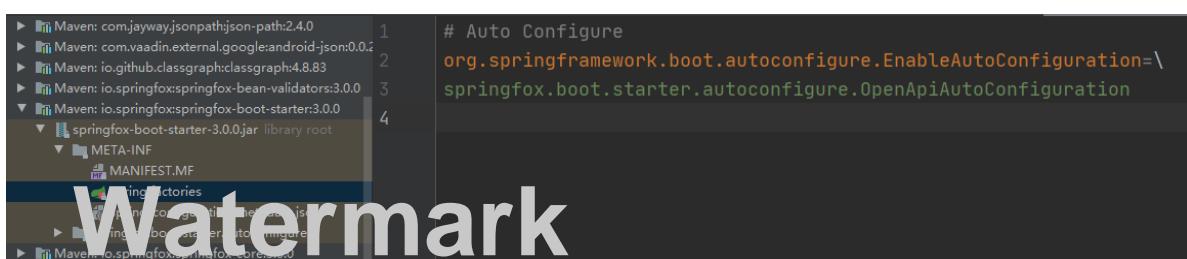
```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
```

springfox-boot-starter做了什么？

Swagger 3.0 主推的一大特色就是这个启动器，那么这个启动器做了什么呢？

记住：启动器的一切逻辑都在自动配置类中。

找到 `springfox-boot-starter` 的自动配置类，在 `/META-INF/spring.factories` 文件中，如下：



从上图可以知道，自动配置类就是 `OpenApiAutoConfiguration`，源码如下：

```
@Configuration
@EnableConfigurationProperties(SpringfoxConfigurationProperties.class)
```

```

@ConditionalOnProperty(value = "springfox.documentation.enabled", havingValue = "true", matchIfMissing =
true)
@Import({
    OpenApiDocumentationConfiguration.class,
    SpringDataRestConfiguration.class,
    BeanValidatorPluginsConfiguration.class,
    Swagger2DocumentationConfiguration.class,
    SwaggerUiWebFluxConfiguration.class,
    SwaggerUiWebMvcConfiguration.class
})
@AutoConfigureAfter({ WebMvcAutoConfiguration.class, JacksonAutoConfiguration.class,
    HttpMessageConvertersAutoConfiguration.class, RepositoryRestMvcAutoConfiguration.class })
public class OpenApiAutoConfiguration {
}

```

敢情这个自动配置类啥也没干，就光导入了几个配置类(`@Import`)以及开启了属性配置(`@EnableConfigurationProperties`)。



纳尼

重点：记住 `OpenApiDocumentationConfiguration` 这个配置类，初步看来这是个BUG，本人也不想深入，里面的代码写的实在拙劣，注释都不写。

撸起袖子就是干？

说真的，还是和以前一样，真的没什么太大的改变，按照文档的步骤一步步来。

定制一个基本的文档示例

一切的东西还是需要配置类手动配置，说真的，我以为会在全局配置文件中自己配置就行了。哎，想多了。配置类如下：

```

@EnableOpenApi
@Configuration
@EnableConfigurationProperties(value = {SwaggerProperties.class})
public class SwaggerOn {
    /**
     * 配置属性
     */
    @Autowired
    private SwaggerProperties properties;
}

```

```

@Bean
public Docket frontApi() {
    return new Docket(DocumentationType.OAS_30)
        //是否开启，根据环境配置
        .enable(properties.getFront().getEnable())
        .groupName(properties.getFront().getGroupName())
        .apiInfo(frontApiInfo())
        .select()
        //指定扫描的包

    .apis(RequestHandlerSelectors.basePackage(properties.getFront().getBasePackage()))
        .paths(PathSelectors.any())
        .build();
}

/**
 * 前台API信息
 */
private ApiInfo frontApiInfo() {
    return new ApiInfoBuilder()
        .title(properties.getFront().getTitle())
        .description(properties.getFront().getDescription())
        .version(properties.getFront().getVersion())
        .contact(      //添加开发者的一些信息
            new Contact(properties.getFront().getContactName(),
properties.getFront().getContactUrl(),
properties.getFront().getContactEmail()))
        .build();
}
}

```

`@EnableOpenApi` 这个注解文档解释如下：

Indicates that Swagger support should be enabled.
This should be applied to a Spring java config and should have an accompanying '@Configuration' annotation.
Loads all required beans defined in @see SpringSwaggerConfig

什么意思呢？大致意思就是只有在配置类标注了 `@EnableOpenApi` 这个注解才会生成 Swagger 文档。

`@EnableConfigurationProperties` 这个注解使开启自定义的属性配置，这是作者自定义的 Swagger 配置。

总之还是和之前一样配置，根据官方文档要求，需要在配置类上加一个 `@EnableOpenApi` 注解。

文档如何分组？

我们都知道，一个项目可能分为 前台，后台，APP 端，小程序端 每个端的接口可能还相同，不可能全部放在一起吧，肯定是要区分开的。

因此，实下开发中文档肯定是要分组的。

分组其实很简单，`Swagger` 向 `Docket` 中注入一个 `Docket` 即为一个组的文档，其中有个 `groupName()` 方法指定分组的名称。

因此只需要注入多个 `Docket` 指定不同的组名即可，当然，这些文档的标题、描述、扫描的路径都是可以不同定制的。

如下配置两个 Docket，分为前台和后台，配置类如下：

```
@EnableOpenApi
@Configuration
@EnableConfigurationProperties(value = {SwaggerProperties.class})
public class SwaggerConfig {

    /**
     * 配置属性
     */
    @Autowired
    private SwaggerProperties properties;

    @Bean
    public Docket frontApi() {
        return new Docket(DocumentationType.OAS_30)
            //是否开启，根据环境配置
            .enable(properties.getFront().getEnable())
            .groupName(properties.getFront().getGroupName())
            .apiInfo(frontApiInfo())
            .select()
            //指定扫描的包

        .apis(RequestHandlerSelectors.basePackage(properties.getFront().getBasePackage()))
        .paths(PathSelectors.any())
        .build();
    }

    /**
     * 前台API信息
     */
    private ApiInfo frontApiInfo() {
        return new ApiInfoBuilder()
            .title(properties.getFront().getTitle())
            .description(properties.getFront().getDescription())
            .version(properties.getFront().getVersion())
            .contact(          //添加开发者的一些信息
                new Contact(properties.getFront().getContactName(),
                properties.getFront().getContactUrl(),
                properties.getFront().getContactEmail()))
            .build();
    }

    /**
     * 后台API
     */
    @Bean
    public Docket backApi() {
        return new Docket(DocumentationType.OAS_30)
            //是否开启，根据环境配置
            .enable(properties.getBack().getEnable())
            .groupName("后台管理")
            .apiInfo(backApiInfo())
            .select()
            .apis(RequestHandlerSelectors.basePackage(properties.getBack().getBasePackage()))
            .paths(PathSelectors.any())
            .build();
    }
}
```

```

    /**
     * 后台API信息
     */
    private ApiInfo backApiInfo() {
        return new ApiInfoBuilder()
            .title(properties.getBack().getTitle())
            .description(properties.getBack().getDescription())
            .version(properties.getBack().getVersion())
            .contact(      //添加开发者的一些信息
                new Contact(properties.getBack().getContactName(),
                properties.getBack().getContactUrl(),
                properties.getBack().getContactEmail()))
            .build();
    }

}

```

属性配置文件 `SwaggerProperties` 如下，分为前台和后台两个不同属性的配置：

```

    /**
     * swagger的属性配置类
     */
    @ConfigurationProperties(prefix = "spring.swagger")
    @Data
    public class SwaggerProperties {

        /**
         * 前台接口配置
         */
        private SwaggerEntity front;

        /**
         * 后台接口配置
         */
        private SwaggerEntity back;

        @Data
        public static class SwaggerEntity {
            private String groupName;
            private String basePackage;
            private String title;
            private String description;
            private String contactName;
            private String contactEmail;
            private String contactUrl;
            private String version;
            private Boolean enable;
        }
    }
}

```

此时的文档截图如下，可以看到有了两个不同的分组：

Watermark

如何添加授权信息？

现在项目API肯定都需要权限认证，否则不能访问，比如请求携带一个 TOKEN。

在Swagger中也是可以配置认证信息，这样在每次请求将会默认携带上。

在 Docket 中有如下两个方法指定授权信息，分别是 `securitySchemes()` 和 `securityContexts()`。在配置类中的配置如下，在构建Docket的时候设置进去即可：

```

@Bean
public Docket frontApi() {
    RequestParameter parameter = new RequestParameterBuilder()
        .name("platform")
        .description("请求头")
        .in(ParameterType.HEADER)
        .required(true)
        .build();

    List<RequestParameter> parameters = Collections.singletonList(parameter);
    return new Docket(DocumentationType.OAS_30)
        //是否开启，根据环境配置
        .enable(properties.getFront().getEnable())
        .groupName(properties.getFront().getGroupName())
        .apiInfo(frontApiInfo())
        .select()
        //指定扫描的包

    .apis(RequestHandlerSelectors.basePackage(properties.getFront().getBasePackage()))
        .paths(PathSelectors.any())
        .build()
        .securitySchemes(securitySchemes())
        .securityContexts(securityContexts());
}

/**
 * 设置授权信息
 */
private List<SecurityScheme> securitySchemes() {
    ApiKey apiKey = new ApiKey("BASE_TOKEN", "token", In.HEADER.toValue());
    return Collections.singletonList(apiKey);
}

/**
 * 授权信息全局应用
 */
private List<SecurityContext> securityContexts() {
    return Collections.singletonList(
        SecurityContext.builder()

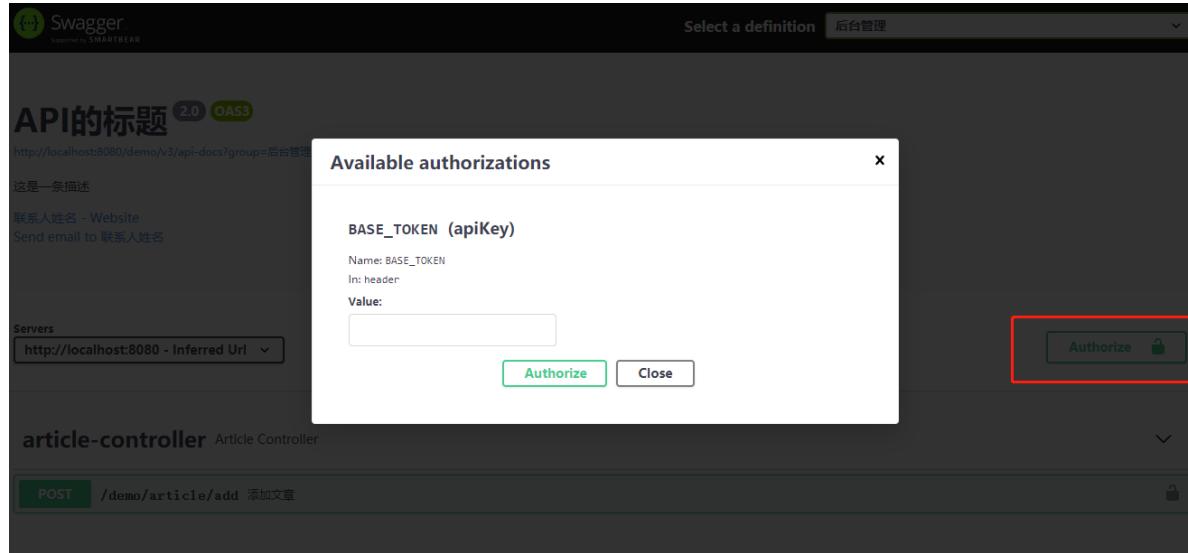
```

```

        . securityReferences(Collections.singletonList(new
SecurityReference("BASE_TOKEN", new AuthorizationScope[] {new AuthorizationScope("global", "")})))
        . build()
    );
}

```

以上配置成功后，在Swagger文档的页面中将会有 **Authorize** 按钮，只需要将请求头添加进去即可。如下图：



如何携带公共的请求参数？

不同的架构可能发请求的时候除了携带 **TOKEN**，还会携带不同的参数，比如请求的平台，版本等等，这些每个请求都要携带的参数称之为公共参数。

那么如何在 **Swagger** 中定义公共的参数呢？比如在请求头中携带。

在 **Docket** 中的方法 `globalRequestParameters()` 可以设置公共的请求参数，接收的参数是一个 `List<RequestParameter>`，因此只需要构建一个 `RequestParameter` 集合即可，如下：

```

@Bean
public Docket frontApi() {
    //构建一个公共请求参数platform，放在在header
    RequestParameter parameter = new RequestParameterBuilder()
        //参数名称
        .name("platform")
        //描述
        .description("请求的平台")
        //放在header中
        .in(ParameterType.HEADER)
        //是否必传
        .required(true)
        .build();
    //构建一个请求参数集合
    List<RequestParameter> parameters = Collections.singletonList(parameter);
    return new Docket(DocumentationType.OAS_30)
        .build()
        .globalRequestParameters(parameters);
}

```

以上配置完成，将会在每个接口中看到一个请求头，如下图：

The screenshot shows the Swagger UI interface for a POST request to the endpoint `/demo/article/add`. The `Parameters` section lists three fields:

- `authord`: `string (query)`, value: `11`
- `id`: `integer($int32) (query)`, value: `11`
- `platform * required (header)`, value: `platform - 请求头`

The third parameter, `platform`, is highlighted with a red border.

粗略是一个BUG

作者在介绍自动配置类的时候提到了一嘴，现在来简单分析下。

`OpenApiAutoConfiguration` 这个自动配置类中已经导入 `OpenApiDocumentationConfiguration` 这个配置类，如下一段代码：

```
@Import({
    OpenApiDocumentationConfiguration.class,
    .....
})
```

`@EnableOpenApi` 的源码如下：

```
@Retention(value = java.lang.annotation.RetentionPolicy.RUNTIME)
@Target(value = {java.lang.annotation.ElementType.TYPE})
@Documented
@Import(OpenApiDocumentationConfiguration.class)
public @interface EnableOpenApi {
}
```

从源码可以看出：`@EnableOpenApi` 这个注解的作用就是导入 `OpenApiDocumentationConfiguration` 这个配置类，纳尼？？？

既然已经在自动配置类 `OpenApiAutoConfiguration` 导入了，那么无论需不需要在配置类上标注 `@EnableOpenApi` 注解不都会开启 `Swagger` 支持吗？

测试一下：不在配置类上标注 `@EnableOpenApi` 这个注解，看看是否 `Swagger` 运行正常。结果在意料之中，还是能够正常运行。

总结：作者只是大致分析了下，这可能是个 `BUG` 亦或是后续有其他的目的，至于结果如此，不想验证了，没什么意思。

总结 Watermark

这篇文章也是尝了个鲜，个人感觉不太香，有点失望。你喜欢吗？

Spring Boot 整合的源码已经上传，需要的朋友回复关键词**Swagger3.0**获取。

另外作者的第一本 PDF 书籍已经整理好了，由浅入深的详细介绍了Mybatis基础以及底层源码，有需要的朋友公众号回复关键词**Mybatis进阶**即可获取，目录如下：



Jenkins部署Spring Boot 项目

Watermark

自动持续集成不知道大家伙有没有听说过，有用过类似的工具吗？

简而言之，自动持续集成的工作主要是能对项目进行构建、自动化测试和发布。

今天这篇文章就来讲解常用的持续集成的工具 Jenkins 以及如何自动构建 Spring Boot 项目。

如何安装Jenkins?

Jenkins 是Java开发的一套工具，可以直接下载 war 包部署在 Tomcat 上，但是今天作者用最方便、最流行的 Docker 安装。

环境准备

在开始安装之前需要准备以下环境和工具：

1. 一台服务器，当然没有的话可以用自己的电脑，作者的服务器型号是 Ubuntu 。
2. JDK 环境安装，作者的版本是 1.8，至于如何安装，网上很多教程。
3. 准备 maven 环境，官网下载一个安装包，放在指定的目录下即可。
4. Git 环境安装，网上教程很多。
5. 代码托管平台，比如 Github 、 GitLab 等。

开始安装Jenkins

Docker 安装 Jenkins 非常方便，只要跟着作者的步骤一步步操作，一定能够安装成功。

Docker环境安装

每个型号服务器安装的方式各不相同，读者可以根据自己的型号安装，网上教程很多。

拉取镜像

我这里安装的版本是 jenkins/jenkins:2.222.3-centos，可以去这里获取你需要的版本：

https://hub.docker.com/_/jenkins?tab=tags。执行如下命令安装：

```
docker pull jenkins/jenkins:2.222.3-centos
```

创建本地数据卷

在本地创建一个数据卷挂载docker容器中的数据卷，我创建的是 /data/jenkins_home/，命令如下：

```
mkdir -p /data/jenkins_home/
```

需要修改下目录权限，因为当映射本地数据卷时， /data/jenkins_home/ 目录的拥有者为 root 用户，而容器中 jenkins 用户的 uid 为 1000。

```
chown -R 1000:1000 /data/jenkins_home/
```

创建容器

除了需要挂载上面创建的 /data/jenkins_home/ 以外，还需要挂载 maven 、 jdk 的根目录。启动命令如下：

```
docker run -d --name jenkins -p 8040:8080 -p 50000:50000 -v /data/jenkins_home:/var/jenkins_home -v /usr/local/jdk:/usr/local/jdk -v /usr/local/maven/jenkins/jenkins:2.222.3-centos
```

以上命令解析如下：

1. -d：后台运行容器
2. --name：指定容器启动的名称

3. `-p` : 指定映射的端口, 这里是将服务器的 8040 端口映射到容器的 8080 以及 50000 映射到容器的 50000 。
注意: 8040 和 50000 一定要是开放的且未被占用, 如果用的是云服务器, 还需要在管理平台开放对应的规则。
4. `-v` : 挂载本地的数据卷到 docker 容器中, **注意:** 需要将 JDK 和 maven 的所在的目录挂载。

初始化配置

容器启动成功, 则需要配置 Jenkins , 安装一些插件、配置远程推送等等。

访问首页

容器创建成功, 访问 `http://ip:8040` , 如果出现以下页面表示安装成功:



Please wait while Jenkins is getting ready to work ...

Your browser will reload automatically when Jenkins is ready.

输入管理员密码

启动成功, 则会要求输入密码, 如下图:

解锁 Jenkins

为了确保管理员安全地安装 Jenkins , 密码已写入到日志中 ([不知道在哪里?](#)) 该文件在服务器上 :

`/var/jenkins_home/secrets/initialAdminPassword`

请从本地复制密码并粘贴到下面。

管理员密码

这里要求输入的是管理的密码, 提示是在 `/var/jenkins_home/secrets/initialAdminPassword` , 但是我们已经将 `/var/jenkins_home` 这个文件夹挂载到本地目录了, 因此只需要去挂载的目录 `/data/jenkins_home/secrets/initialAdminPassword` 文件中找。

输入密码, 点击继续。

安装插件

初始化安装只需要安装社区推荐的一些插件即可，如下图：

自定义Jenkins

插件通过附加特性来扩展Jenkins以满足不同的需求。

安装推荐的插件

安装Jenkins社区推荐的插件。

选择插件来安装

选择并安装最适合的插件。

这里选择 安装推荐的插件，然后 Jenkins 会自动开始安装。

注意： 如果出现想插件安装很慢的问题，找到 `/data/jenkins_home/updates/default.json` 文件，替换的内容如下：

1. 将 `updates.jenkins-ci.org/download` 替换为 `mirrors.tuna.tsinghua.edu.cn/jenkins`
2. 将 `www.google.com` 替换为 `www.baidu.com`。

执行以下两条命令：

```
sed -i 's/www.google.com/www.baidu.com/g' default.json  
sed -i 's/updates.jenkins-ci.org\/download/mirrors.tuna.tsinghua.edu.cn\/jenkins/g' default.json
```

Watermark

新手入门

✓ Timestamper	✓ Credentials Binding	✓ Email Extension	✓ Localization: Chinese (Simplified)	** branch API ** Pipeline: Multibranch ** Authentication Tokens API ** Docker Commons ** Durable Task ** Pipeline: Nodes and Processes ** Pipeline: Basic Steps ** Docker Pipeline ** Pipeline: Stage Tags Metadata ** Pipeline: Declarative Agent API ** Pipeline: Declarative ** Lockable Resources Pipeline Pipeline: Stage View Ant ** GitHub API Git ** GitHub GitHub Branch Source SSH Build Agents Pipeline: GitHub Groovy Libraries Matrix Authorization Strategy Mailer OWASP Markup Formatter Gradle PAM Authentication Git ** Resource Disposer Workspace Cleanup ** MapDB API Subversion LDAP Folders ** - 需要依赖
✓ Build Timeout	✓ Pipeline	✓ Pipeline: Stage View	✓ Ant	
✓ GitHub Branch Source	✓ SSH Build Agents	✓ Pipeline: GitHub Groovy Libraries	✓ Matrix Authorization Strategy	
✓ Mailer	✓ OWASP Markup Formatter	✓ Gradle	✓ PAM Authentication	
✓ Git	✓ Workspace Cleanup	✓ Subversion	✓ LDAP	
✓ Folders				

全部安装完成，继续下一步。

创建管理员

随便创建一个管理员，按要求填写信息，如下图：

创建第一个管理员用户

用户名:

密码:

确认密码:

全名:

电子邮件地址:

实例配置

配置自己的服务器 IP 和端口，如下图：

实例配置

Jenkins URL:

http://172.16.22.30:8040/

Jenkins URL 用于给各种Jenkins资源提供绝对路径链接的根地址。这意味着对于很多Jenkins特色是需要正确设置的，例如：邮件通知、PR状态更新以及提供给构建步骤的BUILD_URL环境变量。

推荐的默认值显示在尚未保存，如果可能的话这是根据当前请求生成的。最佳实践是要设置这个值，用户可能会需要用到。这将会避免在分享或者查看链接时的困惑。

配置完成

按照以上步骤，配置完成后自动跳转到如下界面：

The screenshot shows the Jenkins dashboard. At the top right, it says "欢迎来到 Jenkins!" and "开始创建一个新任务。". On the left, there is a sidebar with links: "新建Item", "用户列表", "构建历史", "Manage Jenkins", "My Views", "凭据", "Lockable Resources", and "新建视图". Below the sidebar, there are two collapsed sections: "构建队列" (which shows "队列中没有构建任务") and "构建执行状态" (which shows "1 空闲" and "2 空闲").

构建Spring Boot 项目

在构建之前还需要配置一些开发环境，比如 [JDK](#)，[Maven](#) 等环境。

配置JDK、maven、Git环境

Jenkins 集成需要用到 [maven](#)、[JDK](#)、[Git](#) 环境，下面介绍如何配置。

首先打开 [系统管理 > 全局工具配置](#)，如下图



管理Jenkins

- 系统配置**
配置全局设置和路径
- 全局安全配置**
Jenkins 安全，定义谁可以访问或使用系统。
- 凭据配置**
配置凭据的提供者和类型
- 全局工具配置**
工具配置，包括它们的位置和自动安装器

分别配置 `JDK` , `Git` , `Maven` 的路径，根据你的实际路径来填写。

注意：这里的 `JDK` 、 `Git` 、 `Maven` 环境一定要挂载到 `docker` 容器中，否则会出现以下提示：

```
xxxx is not a directory on the Jenkins master (but perhaps it exists on some agents)
```

JDK

JDK 安装

新增 JDK

JDK

别名

`jdk 1.8`

`JAVA_HOME`

`/usr/java/jdk1.8.0_181`

自动安装

Git

Git installations

Git

Name

`git`

Path to Git executable

`/usr/bin/git`

自动安装

Maven

Maven 安装

新增 Maven

Maven

Name

`maven 3.6.0`

`MAVEN_HOME`

`/usr/maven/apache-maven-3.6.0/`

自动安装

配置成功后，点击保存。

安装插件

除了初始化配置中安装的插件外，还需要安装如下几个插件：

1. Maven Integration
2. Publish Over SSH

打开 系统管理 -> 插件管理，选择 可选插件，勾选中 Maven Integration 和 Publish Over SSH，点击 直接安装。



在安装界面勾选上安装完成后重启 Jenkins。

安装/更新 插件中

准备

- Checking internet connectivity
- Checking update center connectivity

Publish Over SSH

 等待

Loading plugin extensions

 Pending

重启 Jenkins

 等待

→ [返回首页](#)
(返回首页使用已经安装好的插件)

→ 安装完成后重启Jenkins(空闲时)

添加 SSH Server

SSH Server 是用来连接部署服务器的，用于在项目构建完成后将你的应用推送到服务器中并执行相应的脚本。

打开 系统管理 -> 系统配置，找到 Publish Over SSH 部分，选择 新增

Watermark

Publish over SSH

Jenkins SSH Key

Passphrase

Path to key

Key

Disable exec

SSH Servers

新增

点击 **高级** 展开配置

SSH Servers	
Name	test135
Hostname	47.111.0.135
Username	root
Remote Directory	
高级...	
Test Configuration	
删除	
新增	

最终配置如下：

SSH Servers	
Name	test135 名称，随便
Hostname	47.111.0.135 服务器IP
Username	root 远程登录的用户名
Remote Directory	
<input checked="" type="checkbox"/> Use password authentication, or use a different key	勾选
Passphrase / Password root用户的密码
Path to key	
Key	
Jump host	

保存

应用

配置完成后可点击 **Test Configuration** 测试连接，出现 **success** 则连接成功。

添加凭据

凭据 是用来从 **Git** 仓库拉取代码的，打开 **凭据** -> **系统** -> **全局凭据** -> **添加凭据**



这里配置的是 **Github**，直接使用 **用户名** 和 **密码**，如下图：

类型 Username with password → 选择用户名密码模式

范围 全局 (Jenkins, nodes, items, all child items, etc)

用户名 cm

密码

ID Github

描述 Github凭据

确定

创建成功，点击保存。

新建Maven项目

以上配置完成后即可开始构建了，首先需要新建一个 **Maven** 项目，步骤如下。

创建任务

Watermark

首页点击 新建任务 -> 构建一个maven项目 , 如下图:

The screenshot shows the Jenkins 'New Item' creation interface. At the top, there is a text input field labeled '输入一个任务名称' (Name随便) containing 'test_1'. Below it, there is a note: '» 必填项' (Required). The interface lists several project types: '构建一个自由风格的软件项目' (Free-style software project), which is described as Jenkins's main function; '构建一个maven项目' (Build a maven project), which is highlighted with a red border; '流水线' (Pipeline), described as suitable for long-running tasks across multiple nodes; '构建一个多配置项目' (Multi-configuration project), described as suitable for multi-environment testing; '文件夹' (Folder), described as a container for nested configurations; and 'GitHub 组织' (GitHub Organization), which scans all repositories in a GitHub organization for matching markers. The 'GitHub 组织' section includes a '确定' (Confirm) button.

源码管理

在源码管理中, 选择 Git , 填写 仓库地址 , 选择之前添加的 凭据 。

The screenshot shows the Jenkins Source Management configuration screen for a Git repository. It has two options: '无' (None) and 'Git'. The 'Git' option is selected, indicated by a blue radio button. Below this, there is a 'Repositories' section where a single repository is defined. The 'Repository URL' is set to 'https://github.com/.../vagger3.0-demo.git'. The 'Credentials' dropdown shows '***** (github的凭据)' with a '添加' (Add) button next to it. A red arrow points from the text 'github的仓库地址' to the Repository URL field. In the 'Branches to build' section, there is a field '指定分支 (为空时代表any)' (Specify branch (empty means any)) containing '/master'. A red arrow points from the text '分支, 可以选择分支构建' to this field. At the bottom left, there are '保存' (Save) and '应用' (Apply) buttons. The status bar at the bottom indicates '(自动)' (Automatic).

构建环境

勾选 Add timestamps to the Console Output , 代码构建的过程中会将日志打印出来。

Watermark

构建环境

- Delete workspace before build starts
- Use secret text(s) or file(s)
- Send files or execute commands over SSH before the build starts
- Send files or execute commands over SSH after the build runs
- Abort the build if it's stuck
- Add timestamps to the Console Output
- Inspect build log for published Gradle build scans
- With Ant

构建命令

在 Build 中，填写 Root POM 和 Goals and options，也就是你构建项目的命令。

Build

Root POM: pom.xml

Goals and options: clean package -Pprod -U 

Post Steps

选择 Run only if build succeeds，添加 Post 步骤，选择 Send files or execute commands over SSH。

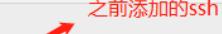
Post Steps

Run only if build succeeds Run only if build succeeds or is unstable Run regardless of build result
Should the post-build steps run only for successful builds, etc.

Send files or execute commands over SSH

SSH Publishers

SSH Server

Name: test135 

Transfers

Transfer Set

Source files: target/swagger-demo-0.0.1-SNAPSHOT.jar

Remove prefix: target/ 

Remote directory: /data/test 

Exec command: /data/test/deploy.sh 

All of the transfer fields (except for Exec timeout) support substitution of Jenkins environment variables

保 应用 Watermark

上图各个选项解析如下：

1. name :选择前面添加的 SSH Server
2. Source files :要推送的文件

3. Remove prefix :文件路径中要去掉的前缀,
4. Remote directory :要推送到目标服务器上的哪个目录下
5. Exec command :目标服务器上要执行的脚本

Exec command 指定了需要执行的脚本，如下：

```
# jdk环境, 如果全局配置了, 可以省略
export JAVA_HOME=/xx/xx/jdk
export JRE_HOME=/xx/xx/jdk/jre
export CLASSPATH=/xx/xx/jdk/lib
export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH

# jenkins编译之后的jar包位置, 在挂载docker的目录下
JAR_PATH=/data/jenkins_home/workspace/test/target
# 自定义的jar包位置
DIR=/data/test

## jar包的名称
JARFILE=swagger-demo-0.0.1-SNAPSHOT.jar

if [ ! -d $DIR/backup ];then
    mkdir -p $DIR/backup
fi

ps -ef | grep $JARFILE | grep -v grep | awk '{print $2}' | xargs kill -9

if [ -f $DIR/backup/$JARFILE ]; then
    rm -f $DIR/backup/$JARFILE
fi

mv $JAR_PATH/$JARFILE $DIR/backup/$JARFILE

java -jar $DIR/backup/$JARFILE > out.log &
if [ $? = 0 ];then
    sleep 30
    tail -n 50 out.log
fi

cd $DIR/backup/
ls -lt|awk 'NR>5{print $NF}'|xargs rm -rf
```

以上脚本大致的意思就是将 kill 原有的进程，启动新构建 jar 包。

脚本可以自己定制，比如备份 Jar 等操作。

构建任务

项目新建之后，一切都已准备就绪，点击 **立即构建** 可以开始构建任务，控制台可以看到 log 输出，如果构建失败，在 log 中会输出原因。

The screenshot shows the Jenkins interface for a Maven project. On the left, there's a sidebar with various options: '返回面板' (Back to Panel), '状态' (Status), '修改记录' (Edit History), '工作空间' (Workspace), '立即构建' (Build Now) which is circled in red, '删除 Maven project' (Delete Maven project), '配置' (Configure), '模块' (Modules), and '重命名' (Rename). Below this is a 'Build History' section with three builds listed: '#3' (2020-4-29 上午9:46), '#2', and '#1'. A red arrow points from the '#1' entry to a tooltip that says '控制台输出' (Console Output) and '控制台可以查看构建log' (Console can view build log). To the right of the build history are two icons: '工作区' (Workspace) and '最新修改' (Latest Changes).

任务构建过程会执行脚本启动项目。

如何构建托管在GitLab的项目？

上文介绍的例子是构建 Github 仓库的项目，但是企业中一般都是私服的 GitLab，那么又该如何配置呢？

其实原理是一样的，只是在构建任务的时候选择的是 GitLab 的凭据，下面将详细介绍。

安装插件

在 系统管理 -> 插件管理 -> 可选插件 中搜索 GitLab Plugin 并安装。

添加GitLab API token

首先打开 凭据 -> 系统 -> 全局凭据 -> 添加凭据，如下图：

The screenshot shows the Jenkins 'Global Credentials (unrestricted)' configuration page. It has fields for '类型' (Type) set to 'GitLab API token', '范围' (Scope) set to '全局 (Jenkins, nodes, items, all child items, etc.)', 'API token' (containing 'gitlab私服的token'), 'ID' (containing 'gitlabToken'), and '描述' (Description) (containing 'gitlab的API Token'). Red arrows point from the text descriptions on the left to the corresponding fields: one arrow points to the '类型' field with the label '类型选择Gitlab API token', another to the 'API token' field with 'gitlab私服的token', another to the 'ID' field with '任意起名', and another to the '描述' field with 'gitlab的API Token'.

上图中的 API token 如何获取呢？

打开 GitLab (例如公司内网的 GitLab 网站) , 点击个人设置菜单下的 setting , 再点击 Account , 复制 Private token , 如下:

The screenshot shows the 'Account' tab selected in the top navigation bar. A blue banner at the top states 'Some options are unavailable for LDAP accounts'. Below it, the 'Private Tokens' section is shown. A note says 'Keep these tokens secret, anyone with access to them can interact with GitLab as if they were you.' A text input field contains the token 'j-ssNZ-EDViBY9p92xcz'. A descriptive note below the input field says 'Your private token is used to access the API and Atom feeds without username/password authentication.'

上图的 Private token 则是 API token , 填上即可。

配置GitLab插件

打开 系统管理 -> 系统配置 -> GitLab , 如下图:

The screenshot shows the 'GitLab' configuration screen. Under 'GitLab connections', there is a section for 'Connection name' (set to 'GitLab'), 'GitLab host URL' (set to 'http://私服地址:80'), and 'Credentials' (a dropdown set to 'GitLab API token' with a note '选择你配置的gitlab API token'). Below these fields are 'Success' and 'Test Connection' buttons, and a red '删除' (Delete) button.

配置成功后, 点击 Test Connection , 如果提示 Success 则配置成功。

新建任务

新建一个Maven任务, 配置的步骤和上文相同, 唯一区别则是配置 Git 仓库地址的地方, 如下图:

The screenshot shows the 'Source Code Management' configuration screen. Under 'Repositories', there is a 'Repository URL' field containing 'http://私服地址:80/rock/gradle.git' and a 'Credentials' dropdown set to '**** (gitlab120)'. Red arrows point from the text 'gitlab仓库的项目地址' to the repository URL and from 'gitlab的凭证' to the credentials dropdown. At the bottom, there are '保存' (Save) and '应用' (Apply) buttons.

仓库地址和凭据需要填写 Gitlab 相对应的。

后续操作

后续一些操作，比如构建项目，控制台输出等操作，都是和 GitHub 操作相同，不再赘述了。

多模块项目如何构建？

如果你的多模块不是通过私服仓库依赖的，那么在构建打包是有先后顺序的，在新建任务的时候需要配置 Build 的 maven 命令，如下图：

The screenshot shows the 'Build' configuration section of a Jenkins job. It includes fields for 'Root POM' (set to 'pom.xml'), 'Goals and options' (containing '-P test -pl api -am clean package', which is highlighted with a red box), 'MAVEN_OPTS', and several checkboxes under 'MAVEN_OPTS'. One checkbox, 'Enable triggering of downstream projects', is checked and has a sub-option 'Block downstream trigger when building' also checked.

上图中的 Goals and options 中的命令就是构建 api 这个模块的命令，至于这个命令是什么意思，前面有单独一篇文章介绍过，请看[一次打包引发的思考，原来maven还能这么玩~](#)。

总结

本文详细的介绍了如何从零安装部署一个 Jenkins，这下又能吹牛了，哈哈....

如果觉得不错，点个赞不迷路~

另外作者的第一本 PDF 书籍已经整理好了，由浅入深的详细介绍了Mybatis基础以及底层源码，有需要的朋友公众号回复关键词**Mybatis进阶**即可获取，目录如下：



码了两本书的陈某，邀你一起进步



码猿技术专栏

回复以下关键词获取专栏书籍

▶ Mybatis 进阶

▶ Spring Boot 进阶

写最精的文章，做最野的程序员

Spring Boot 第十九弹，Spring Boot 多环境切换配置

前言

日常开发中至少有三个环境，分别是开发环境（`dev`），测试环境（`test`），生产环境（`prod`）。

不同的环境的各种配置都不相同，比如数据库，端口，IP 地址等信息。

那么这么多环境如何区分，如何打包呢？

本篇文章就来介绍一下 `Spring Boot` 中多环境如何配置，如何打包。

Spring Boot 自带的多环境配置

Spring Boot 对多环境整合已经有了很好的支持，能够在打包，运行间自由切换环境。

那么如何配置呢？下面将会逐步介绍。

创建不同环境的配置文件

既然每个环境的配置都不相同，我们把不同环境的配置放在不同的配置文件中，因此需要创建三个不同的配置文件，分别是 `application-dev.properties`、`application-test.properties`、`application-prod.properties`。

注意：配置文件的名称一定要是 `application-name.properties` 或者 `application-name.yml` 格式。这个 `name` 可以自定义，主要用于区分。

此时整个项目中就有四个配置文件，加上 `application.properties`。

指定运行的环境

虽然你创建了各个环境的配置文件，但是 `Spring Boot` 仍然不知道你要运行哪个环境，有以下两种方式指定：

配置文件中指定

在 `application.properties` 或者 `application.yml` 文件中指定，内容如下：

```
# 指定运行环境为测试环境  
spring.profiles.active=test
```

以上配置有什么作用呢？

如果没有指定运行的环境，`Spring Boot` 默认会加载 `application.properties` 文件，而这个的文件又告诉 `Spring Boot` 去找 `test` 环境的配置文件。

运行 jar 的时候指定

`Spring Boot` 内置的环境切换能够在运行 `Jar` 包的时候指定环境，命令如下：

```
java -jar xxx.jar --spring.profiles.active=test
```

以上命令指定了运行的环境是 `test`，是不是很方便呢？

Maven 的多环境配置

`Maven` 本身也提供了对多环境的支持，不仅仅支持 `Spring Boot` 项目，只要是基于 `Maven` 的项目都可以配置。

`Maven` 对于多环境的支持在功能方面更加强大，支持 `JDK版本`、`资源文件`、`操作系统` 等等因素来选择环境。

如何配置呢？下面逐一介绍。

创建多环境配置文件

创建不同环境的配置文件，分别是 `application-dev.properties`、`application-test.properties`、`application-prod.properties`。

加上默认的配置文件 `application.properties` 同样是四个配置文件。

定义激活的变量

需要将 `Maven` 激活的环境作用于 `Spring Boot`，实际还是利用了 `spring.profiles.active` 这个属性，只是现在这个属性的取值将是取值于 `Maven`。配置如下：

```
spring.profiles.active=@profile.active@
```

`profile.active` 实际上就是一个变量，在 `maven` 打包的时候指定的 `-P test` 传入的就是值。

pom 文件中定义 profiles

需要在 maven 的 pom.xml 文件中定义不同环境的 profile，如下：

```
<!--定义三种开发环境-->
<profiles>
    <profile>
        <!--不同环境的唯一id-->
        <id>dev</id>
        <activation>
            <!--默认激活开发环境-->
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <!--profile.active对应application.yml中的@profile.active@-->
            <profile.active>dev</profile.active>
        </properties>
    </profile>

    <!--测试环境-->
    <profile>
        <id>test</id>
        <properties>
            <profile.active>test</profile.active>
        </properties>
    </profile>

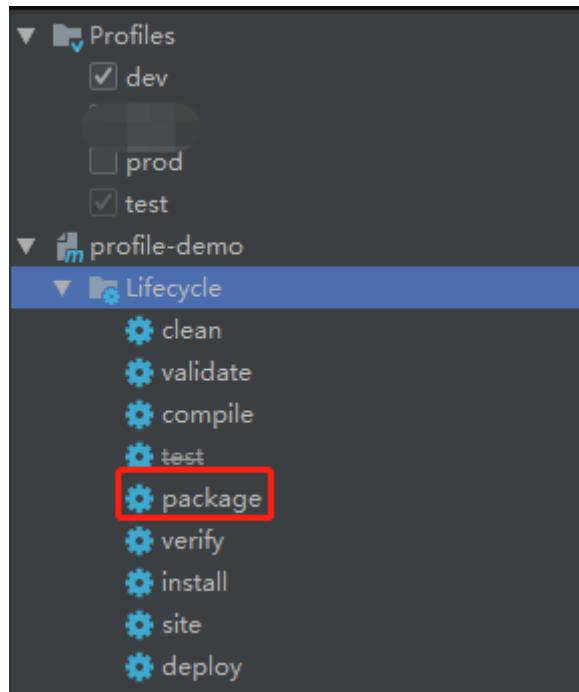
    <!--生产环境-->
    <profile>
        <id>prod</id>
        <properties>
            <profile.active>prod</profile.active>
        </properties>
    </profile>
</profiles>
```

标签 <profile.active> 正是对应着配置文件中的 @profile.active@ 。

<activeByDefault> 标签指定了默认激活的环境，则是打包的时候不指定 -P 选项默认选择的环境。

以上配置完成后，将会在IDEA的右侧 Maven 选项卡中出现以下内容：

Watermark



可以选择打包的环境，然后点击 `package` 即可。

或者在项目的根目录下用命令打包，不过需要使用 `-P` 指定环境，如下：

```
mvn clean package package -P test
```

maven 中的 `profile` 的激活条件还可以根据 `jdk`、`操作系统`、`文件存在或者缺失` 来激活。这些内容都是在 `<activation>` 标签中配置，如下：

```
<!--activation用来指定激活方式，可以根据jdk环境，环境变量，文件的存在或缺失-->
<activation>
    <!--配置默认激活-->
    <activeByDefault>true</activeByDefault>

    <!--通过jdk版本-->
    <!--当jdk环境版本为1.8时，此profile被激活-->
    <jdk>1.8</jdk>
    <!--当jdk环境版本1.8或以上时，此profile被激活-->
    <jdk>[1.8,)</jdk>

    <!--根据当前操作系统-->
    <os>
        <name>Windows XP</name>
        <family>Windows</family>
        <arch>x86</arch>
        <version>5.1.2600</version>
    </os>
</activation>
```

资源过滤

如果你不配置这一步，将会在任何环境下打包都会带上全部的配置文件，但是我们可以配置只保留对应环境下的配置文件，这样安全性更高。

这一步配置很简单，只需要在 `pom.xml` 文件中指定 `<resource>` 过滤的条件即可，如下：

```
<build>
    <resources>
```

```
<!--排除配置文件-->
<resource>
    <directory>src/main/resources</directory>
    <!--先排除所有的配置文件-->
    <excludes>
        <!--使用通配符，当然可以定义多个exclude标签进行排除-->
        <exclude>application*.properties</exclude>
    </excludes>
</resource>

<!--根据激活条件引入打包所需的配置和文件-->
<resource>
    <directory>src/main/resources</directory>
    <!--引入所需环境的配置文件-->
    <filtering>true</filtering>
    <includes>
        <include>application.yml</include>
        <!--根据maven选择环境导入配置文件-->
        <include>application-${profile.active}.yml</include>
    </includes>
</resource>
</resources>
</build>
```

上述配置主要分为两个方面，第一是先排除所有配置文件，第二是根据 `profile.active` 动态的引入配置文件。

总结

至此，`Maven` 的多环境打包已经配置完成，相对来说挺简单，既可以在 `IDEA` 中选择环境打包，也同样支持命令 `-P` 指定环境打包。

总结

本文介绍了`Spring Boot` 的两种打包方式，每种方式有各自的优缺点，你更喜欢哪种呢？

源码已经上传，回复关键词 `多环境配置` 获取。

另外作者的第一本 `PDF` 书籍已经整理好了，由浅入深的详细介绍了`Mybatis`基础以及底层源码，有需要的朋友公众号回复关键词 `Mybatis进阶` 即可获取，目录如下：

Watermark



Spring Boot 与注解那些事儿~

前言

注解相信大家都用过，尤其是 Spring Boot 这个框架，比如 `@Controller`。

这篇文章就来介绍下 Spring Boot 中如何自定义一个注解，顺带介绍一下 Spring Boot 与 AOP 如何整合。

Watermark

什么是AOP?

AOP 即是面向切面，是 Spring 的核心功能之一，主要的目的即是针对业务处理过程中的横向拓展，以达到低耦合的效果。

举个栗子，项目中有记录操作日志的需求、或者流程变更是记录变更履历，无非就是插表操作，很简单的一个 save 操作，都是一些记录日志或者其他辅助性的代码。一遍又一遍的重写和调用。不仅浪费了时间，又将项目变得更加的冗余，实在得不偿失。

此时 AOP 的就该出场了，能够在不改变原逻辑的基础上实现相关功能。

AOP的相关概念（面试常客）

要理解 Spring Boot 整合 Aop 的实现，就必须先对面向切面实现的一些 Aop 的概念有所了解，不然也是云里雾里。

切面（Aspect）：一个关注点的模块化。以注解 @Aspect 的形式放在类上方，声明一个切面。

连接点（Joinpoint）：在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候都可以是连接点。

通知（Advice）：通知增强，需要完成的工作叫做通知，就是你写的业务逻辑中需要比如事务、日志等先定义好，然后需要的地方再去用。增强包括如下五个方面：

1. @Before：在切点之前执行
2. @After：在切点方法之后执行
3. @AfterReturning：切点方法返回后执行
4. @AfterThrowing：切点方法抛异常执行
5. @Around：属于环绕增强，能控制切点执行前，执行后，用这个注解后，程序抛异常，会影响 @AfterThrowing 这个注解。

切点（Pointcut）：其实就是筛选出的连接点，匹配连接点的断言，一个类中的所有方法都是连接点，但又不全需要，会筛选出某些作为连接点做为切点。

引入（Introduction）：在不改变一个现有类代码的情况下，为该类添加属性和方法，可以在无需修改现有类的前提下，让它们具有新的行为和状态。其实就是把切面（也就是新方法属性：通知定义的）用到目标类中去。

目标对象（Target Object）：被一个或者多个切面所通知的对象。也被称做被通知（advised）对象。既然 Spring AOP 是通过运行时代理实现的，这个对象永远是一个被代理（proxied）对象。

AOP代理（AOP Proxy）：AOP 框架创建的对象，用来实现切面契约（例如通知方法执行等等）。在 Spring 中，AOP 代理可以是 JDK 动态代理或者 CGLIB 代理。

织入（Weaving）：把切面连接到其它的应用程序类型或者对象上，并创建一个被通知的对象。这些可以在编译时（例如使用 AspectJ 编译器），类加载时和运行时完成。Spring 和其他纯 Java AOP 框架一样，在运行时完成织入。

Spring Boot 如何整合AOP自定义一个注解？

在实际开发中对于一些公共的逻辑需要取出，这时候就需要使用 AOP，比如日志的记录、权限的验证等等，这些功能都可以用注解轻松的完成。

下面介绍如何在 Spring Boot 使用 AOP 定义一个注解。

添加依赖starter

AOP 整合 Spring Boot 有一个 `starter`，只需要添加依赖即可，如下：

```
<!--springboot集成Aop-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

开启AOP

在配置类上标注 `@EnableAspectJAutoProxy` 注解即可开启 AOP，这个注解有什么用呢，源码如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(AspectJAutoProxyRegistrar.class)
public @interface EnableAspectJAutoProxy {}
```

最重要的是如下一行代码：

```
@Import(AspectJAutoProxyRegistrar.class)
```

`@Import` 这个注解很熟悉了吧，快速注入一个类，这里是注入一个 `AnnotationAwareAspectJAutoProxyCreator`。

自定义一个注解

就以日志处理为例子，定义一个日志处理的注解，如下：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface SysLog {
    String value() default "";
}
```

定义一个切面

一个切面的满足条件如下：

1. 类上标注了 `@Aspect` 注解
2. 注入到IOC容器中，比如 `@Component` 注解

定义的日志切面如下：

```
@Component
@Aspect
@Order(Ordered.HIGHEST_PRECEDENCE)
public class SysLogAspect {
```

`@Order` 指定了切面执行的优先级，假如有多个切面，肯定是要有先后的执行顺序，这样才能保证逻辑性。

定义切点表达式

这里需要拦截的肯定是 `@SysLog` 这个注解，只要方法上标注了该注解都将会被拦截，表达式如下：

```
@Pointcut("@annotation(com.example.annotation_demo.annotation.SysLog)")  
public void pointCut() {}
```

添加通知方法

既然是日志记录，肯定是在方法执行前，执行后都需要记录，因此需要定义一个环绕通知，如下：

```
@Around("pointCut()")  
public Object around(ProceedingJoinPoint point) throws Throwable {  
    //逻辑开始时间  
    long beginTime = System.currentTimeMillis();  
  
    //执行方法  
    Object result = point.proceed();  
  
    //todo, 保存日志, 自己完善  
    saveLog(point, beginTime);  
  
    return result;  
}
```

测试

以上配置完成后即可使用，只需要在需要的方法上标注 `@SysLog` 注解即可，如下：

```
@SysLog  
@PostMapping("/add")  
public String add(){  
    return "";  
}
```

使用拦截器如何自定义注解？

使用 AOP 自定义的注解在每个方法上都会被拦截验证，首先效率上就不高。

然而拦截器是在每个 Controller 方法执行之前进行拦截，其他的方法都不会生效，比如 service 方法。

比如权限的验证、防止瞬间重复点击等等需求就适合使用拦截器自定义的注解。

自定义一个注解

就以防止瞬间重复点击的例子来创建一个注解，如下：

```
@Target({ElementType.METHOD, ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface RateLimitAnnotation {  
    /**  
     * 默认失效时间5秒  
     */  
    long seconds() default 5;  
}
```

自定义拦截器

需要在请求执行之前完成验证，逻辑很简单，就是判断方法上有没有标注 `@RepeatSubmit` 注解，代码如下：

```
/*
 * description:重复提交注解的拦截器
 */
@Component
public class RepeatSubmitInterceptor implements HandlerInterceptor {

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {
        if (handler instanceof HandlerMethod) {
            //只拦截标注了@RepeatSubmit该注解
            HandlerMethod handlerMethod = (HandlerMethod) handler;
            //获取controller方法上标注的注解
            RepeatSubmit repeatSubmit =
                    AnnotationUtils.findAnnotation(handlerMethod.getMethod(), RepeatSubmit.class);
            //没有限制重复提交，直接跳过
            if (Objects.isNull(repeatSubmit))
                return true;
            //todo 一个值，标志这个请求的唯一性，比如IP+userId+uri+请求参数
            String flag="";
            //存在即返回false，不存在即返回true
            Boolean ifAbsent = stringRedisTemplate.opsForValue().setIfAbsent(flag, "", repeatSubmit.seconds(), TimeUnit.SECONDS);
            if (ifAbsent!=null&&!ifAbsent)
                //todo: 此处抛出异常，需要在全局异常解析器中捕获
                throw new RepeatSubmitException();
        }
        return true;
    }
}
```

注入的拦截器

将上述自定义的拦截器注入到 `Spring Boot` 中，这里不再演示了，前面教程有介绍过，请看：[Spring Boot 第六弹，拦截器如何配置，看这儿~](#)。

测试

在需要拦截方法上添加 `@RepeatSubmit` 注解即可，如下：

```
@RepeatSubmit
@GetMapping("/add")
public String add() {
    return "add";
}
```

内部调用导致AOP注解失效

这个问题在事务中也是经常被忽略的问题，网上很多人说是 AOP 的 Bug，其实在我看来这真不是一个 BUG，并且也是有办法解决的。

先来看一下失效的案例，如下：

```
public class ArticleServiceImpl {  
    @SysLog  
    public void A() {  
        .....  
    }  
  
    public void B() {  
        this.A();  
    }  
}
```

在上述的代码中，如果执行方法 B，则 @SysLog 注解将会失效。

失效的原因

AOP 使用的是动态代理的机制，它会给类生成一个代理类，事务的相关操作都在代理类上完成。内部方式使用 this 调用方式时，使用的是实例调用，并没有通过代理类调用方法，所以会导致事务失效。

解决方法

其实解决方法有很多，下面将会一一介绍。

1. 引入自身的Bean

在类内部通过 @Autowired 将本身 bean 引入，然后通过调用自身 bean，从而实现使用 AOP 代理操作。代码如下：

```
public class ArticleServiceImpl {  
    /**  
     * 注入自身的Bean  
     */  
    @Autowired  
    private ArticleService articleService;  
  
    @SysLog  
    public void A() {  
        .....  
    }  
  
    public void B() {  
        articleService.A();  
    }  
}
```

Watermark

2. 通过ApplicationContext引入bean

通过 `ApplicationContext` 获取 `bean`，通过 `bean` 调用内部方法，就使用了 `bean` 的代理类。

需要先创建一个 `ApplicationContext` 的工具类获取 `ApplicationContext`，然后才能调用 `getBean()` 方法，代码如下：

```
public class ArticleServiceImpl {  
  
    @SysLog  
    public void A() {  
        .....  
    }  
  
    public void B() {  
        ApplicationContextUtils.getApplicationContext().getBean(ArticleService.class).A();  
    }  
}
```

3. 通过AopContext获取当前类的代理类

此种方法需要设置 `@EnableAspectJAutoProxy` 中的 `exposeProxy` 为 `true`。

使用 `AopContext` 获取当前的代理对象，代码如下：

```
public class ArticleServiceImpl {  
  
    @SysLog  
    public void A() {  
        .....  
    }  
  
    public void B() {  
        ((ArticleService)AopContext.currentProxy()).A();  
    }  
}
```

总结

这篇文章介绍了 `AOP` 的相关概念、`AOP` 实现自定义注解以及拦截器实现自定义注解，都是日常开发中必备的知识点，希望这篇文章对各位有所帮助。

源码已经上传，回复关键词 `AOP注解` 获取。

最后，别忘了点赞哦！！！

另外作者的第一本 `PDF` 书籍已经整理好了，由浅入深的详细介绍了Mybatis基础以及底层源码，有需要的朋友公众号回复关键词 `Mybatis进阶` 即可获取，目录如下：

Watermark



启动过程源码分析

前言

Spring Boot 专栏已经写了五十多天了，前面二十章从基础应用到高级整合避重就轻介绍的都是工作、面试中常见的知识点。

今天开始底层源码介绍的阶段，相对内容比较深一点，作者也尽可能介绍的通俗易懂，层次分明一点。相信读过我写的 Mybatis 专栏的文章都知道，只要跟着作者的步骤，方法一步步研究，其实源码并不难。

这篇文章花了四天时间精雕细琢，力求介绍的通俗易懂，毕竟源码相对难度更高些，希望通过作者拆分讲解能够帮助到读者。

如果没读过作者的前二十篇文章，[点击前往](#)

源码版本

作者 Spring Boot 是基于 2.4.0。每个版本有些变化，读者尽量和我保持一致，以防源码有些出入。

从哪入手？

相信很多人尝试读过 Spring Boot 的源码，但是始终没有找到合适的方法。那是因为你对 Spring Boot 的各个组件、机制不是很了解，研究起来就像大海捞针。

至于从哪入手不是很简单的问题吗，当然主启动类了，即是标注着 `@SpringBootApplication` 注解并且有着 `main()` 方法的类，如下一段代码：

```
@SpringBootApplication
public class AnnotationDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(AnnotationDemoApplication.class, args);
    }
}
```

话不多说，`DEBUG` 伺候，别怕，搞它……



源码如何切分？

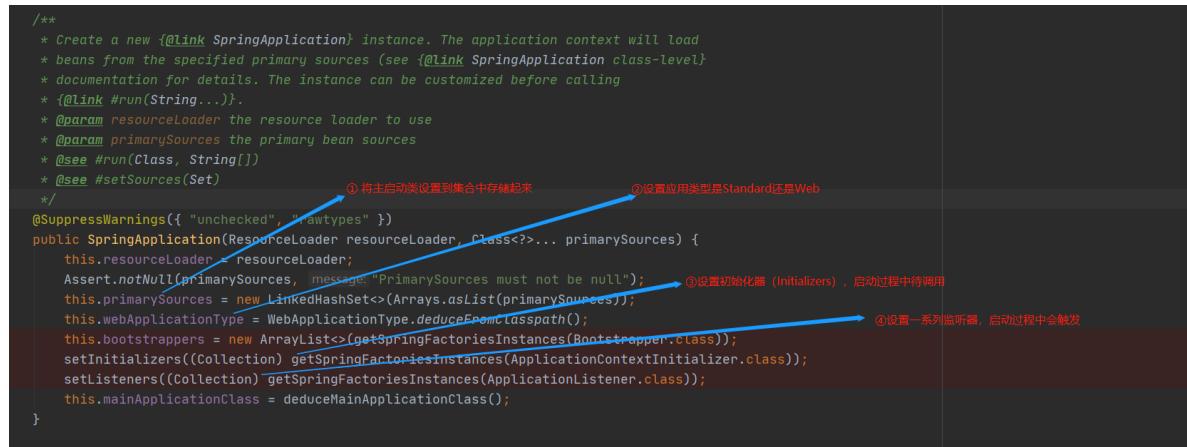
`SpringApplication` 中的静态 `run()` 方法并不是一步完成的，最终执行的源码如下：

```
//org.springframework.context.ConfigurableApplicationContext
public static ConfigurableApplicationContext run(Class<?>[] primarySources, String[] args) {
    return new SpringApplication(primarySources).run(args);
}
```

很显然分为两个步骤，分别是创建 `SpringApplication` 和执行 `run()` 方法，下面将分为这两个部分介绍。

如何创建SpringApplication?

创建即是 new 对象了， DEBUG 跟进代码，最终执行的 SpringApplication 构造方法如下图：



如上图中标注的注释，创建过程重用的其实分为 ②、③、④ 这三个阶段，下面将会一一介绍每个阶段做了什么事。

设置应用类型

这个过程非常重要，直接决定了项目的类型，应用类型分为三种，都在 `WebApplicationType` 这个枚举类中，如下：

1. `NONE`：顾名思义，什么都没有，正常流程走，不额外的启动 web容器，比如 `Tomcat`。
2. `SERVLET`：基于 `servlet` 的web程序，需要启动内嵌的 `servlet` web容器，比如 `Tomcat`。
3. `REACTIVE`：基于 `reactive` 的web程序，需要启动内嵌 `reactive` web容器，作者不是很了解，不便多说。

判断的依据很简单，就是加载对应的类，比如加载了 `DispatcherServlet` 等则会判断是 `Servlet` 的web程序。源码如下：

```
static WebApplicationType deduceFromClasspath() {
    if (ClassUtils.isPresent(WEBFLUX_INDICATOR_CLASS, null) &&
        !ClassUtils.isPresent(WEBMVC_INDICATOR_CLASS, null)
        && !ClassUtils.isPresent(JERSEY_INDICATOR_CLASS, null)) {
        return WebApplicationType.REACTIVE;
    }
    for (String className : SERVLET_INDICATOR_CLASSES) {
        if (!ClassUtils.isPresent(className, null)) {
            return WebApplicationType.NONE;
        }
    }
    return WebApplicationType.SERVLET;
}
```

这里我引入了 `spring-boot-starter-web`，肯定是 `Servlet` 的web程序。

设置初始化器(Initializer)

初始化器 `ApplicationContextInitializer` 是个好东西，用于 `IOC` 容器刷新之前初始化一些组件，比如 `ServletContextApplicationContextInitializer`。

那么如何获取初始化器呢？跟着上图中的代码进入，在 `SpringApplication` 中的如下图中的方法：

```

private <T> Collection<T> getSpringFactoriesInstances(Class<T> type, Class<?>[] parameterTypes, Object... args) {
    ClassLoader classLoader = getClassLoader();
    // Use names and ensure unique to protect against duplicates
    Set<String> names = new LinkedHashSet<>(SpringFactoriesLoader.loadFactoryNames(type, classLoader));
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes, classLoader, args, names);
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}

```

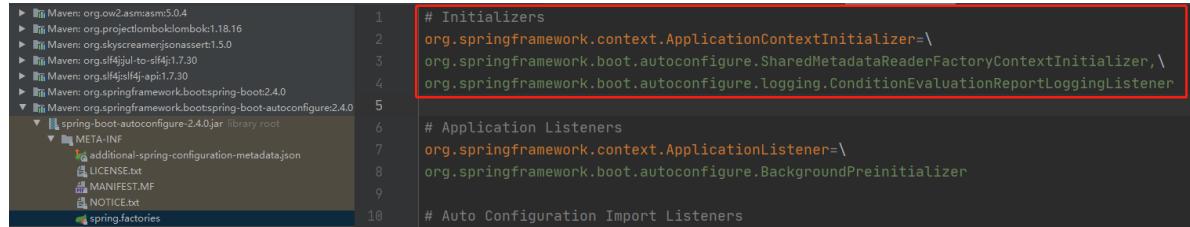
相对重要的就是第一步获取初始化器的名称了，这个肯定是 全类名 了，详细源码肯定在

`loadFactoryNames()` 方法中了，跟着源码进入，最终调用的是

`#SpringFactoriesLoader.loadSpringFactories()` 方法。

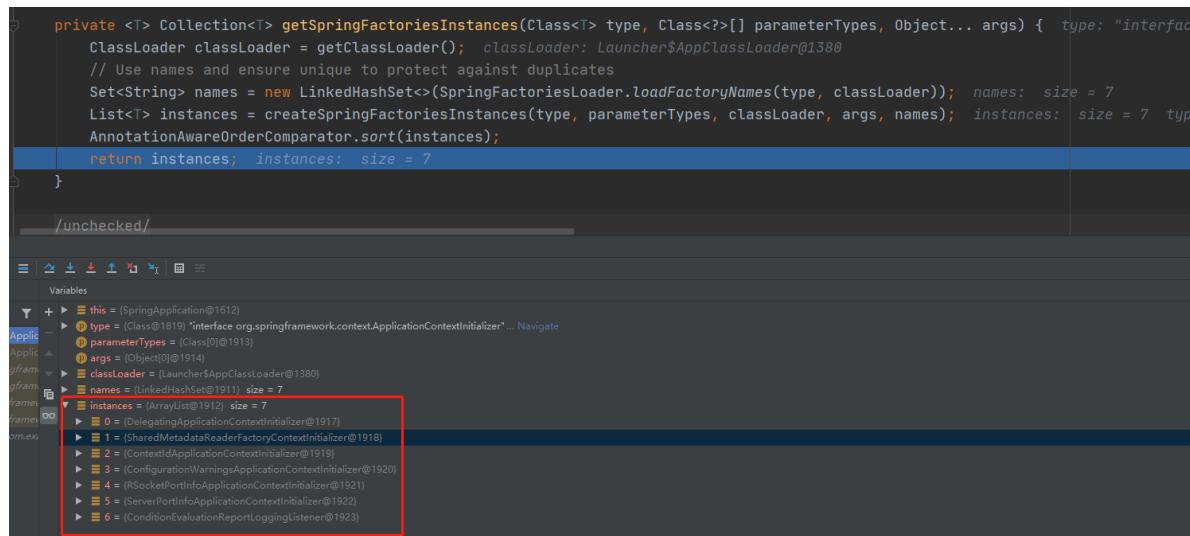
`loadSpringFactories()` 方法就不再详细解释了，其实就是从类路径 `META-INF/spring.factories` 中加载 `ApplicationContextInitializer` 的值。

在 `spring-boot-autoconfigure` 的 `spring.factories` 文件中的值如下图：



上图中的只是一部分初始化器，因为 `spring.factories` 文件不止一个。

下图中是我的 demo 中注入的初始化器，现实项目中并不止这些。



这也告诉我们自定义一个 `ApplicationContextInitializer` 只需要实现接口，在 `spring.factories` 文件中设置即可。

设置监听器(Listener)

监听器（`ApplicationListener`）这个概念在 Spring 中就已经存在，主要用于监听特定的事件（`ApplicationEvent`），比如IOC容器刷新、容器关闭等。

Spring Boot 扩展了 `ApplicationEvent` 建立了 `SpringApplicationEvent` 这个抽象类，主要用于 Spring Boot 启动过程中触发的事件，比如程序启动中、程序启动完成等。如下图：

- `ApplicationContextInitializedEvent` (`org.springframework.boot.context.event`)
- `ApplicationEnvironmentPreparedEvent` (`org.springframework.boot.context.event`)
- `ApplicationFailedEvent` (`org.springframework.boot.context.event`)
- `ApplicationPreparedEvent` (`org.springframework.boot.context.event`)
- `ApplicationReadyEvent` (`org.springframework.boot.context.event`)
- `ApplicationStartedEvent` (`org.springframework.boot.context.event`)
- `ApplicationStartingEvent` (`org.springframework.boot.context.event`)

监听器如何获取？从源码中知道其实和初始化器(`ApplicationContextInitializer`)执行的是同一个方法，同样是从 `META-INF/spring.factories` 文件中获取。

在 `spring-boot-autoconfigure` 的 `spring.factories` 文件中的值如下图：

```

2 org.springframework.context.ApplicationContextInitializer=\n3 org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\n4 org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener\n5\n6 # Application Listeners\n7 org.springframework.context.ApplicationListener=\n8 org.springframework.boot.autoconfigure.BackgroundPreinitializer
9

```

`spring.factories` 文件不止一个，同样监听器也不止以上这些。

作者 `demo` 中注入的一些监听器如下图：

```

private <T> Collection<T> getSpringFactoriesInstances(Class<T> type, Class<?>[] parameterTypes, Object... args) {
    ClassLoader classLoader = getClassLoader(); classLoader: Launcher$AppClassLoader@1380
    // Use names and ensure unique to protect against duplicates
    Set<String> names = new LinkedHashSet<>(SpringFactoriesLoader.loadFactoryNames(type, classLoader)); names: s
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes, classLoader, args, names); instances:
    AnnotationAwareOrderComparator.sort(instances);
    return instances; instances: size = 9
}

```

```

this = (SpringApplication@1612)
type = (Class@1882) 'interface org.springframework.context.ApplicationListener' ... Navigate
parameterTypes = [Class@0]@1983
args = [Object@0]@1984
classLoader = [Launcher$AppClassLoader@1380]
names = [LinkedHashSet@1981] size = 9
instances = [ArrayList@1982] size = 9
  0 = [EnvironmentPostProcessorApplicationListener@1987]
  1 = [AnsiOutputApplicationListener@1988]
  2 = [LoggingApplicationListener@1989]
  3 = [BackgroundPreinitializer@1990]
  4 = [DelegatingApplicationListener@1991]
  5 = [ParentContextCloserApplicationListener@1992]
  6 = [ClearCachesApplicationListener@1993]
  7 = [FileEncodingApplicationListener@1994]
  8 = [LiquibaseServiceLocatorApplicationListener@1995]

```

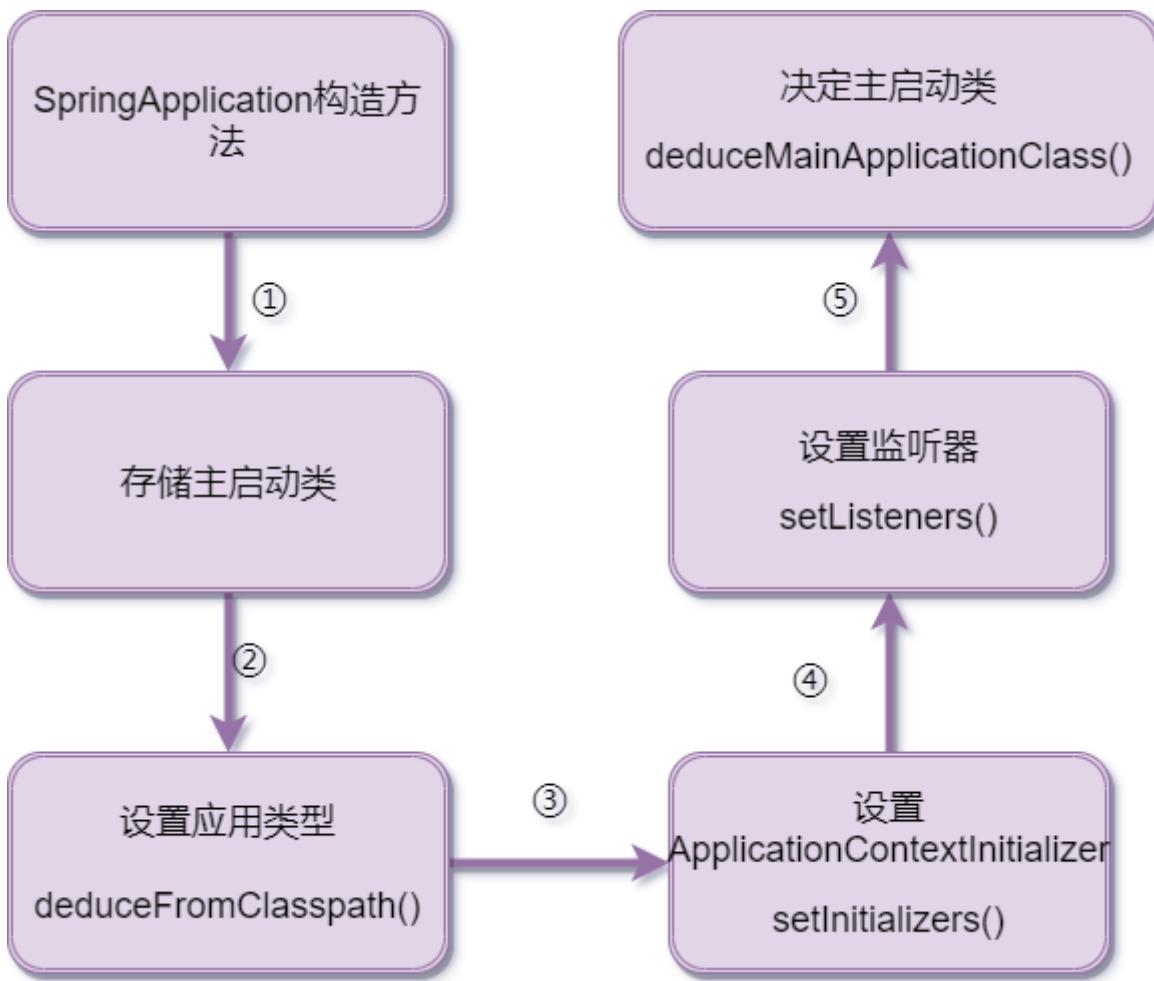
总结

`SpringApplication` 的构建都是为了 `run()` 方法启动做铺垫，构造方法中总共就有几行代码，最重要的部分就是设置应用类型、设置初始化器、设置监听器。

注意： 初始化器和这里的监听器都要放置在 `spring.factories` 文件中才能在这一步骤加载，否则不会生效，因此此时 `IOC容器` 还未创建，即使将其注入到 `IOC容器` 中也是不会生效的。

作者简单的画了张执行流程图，仅供参考，如下：

Watermark



执行run()方法

上面分析了 `SpringApplication` 的构建过程，一切都做好了铺垫，现在到了启动的过程了。

作者根据源码将启动过程分为了8步，下面将会一一介绍。

1. 获取、启动运行过程监听器

`SpringApplicationRunListener` 这个监听器和 `ApplicationListener` 不同，它是用来监听应用程序启动过程的，接口的各个方法含义如下：

```

public interface SpringApplicationRunListener {
    // 在run()方法开始执行时，该方法就立即被调用，可用于在初始化最早期时做一些工作
    void starting();
    // 当environment构建完成，ApplicationContext创建之前，该方法被调用
    void environmentPrepared(ConfigurableEnvironment environment);
    // 当ApplicationContext构建完成时，该方法被调用
    void contextPrepared(ConfigurableApplicationContext context);
    // 在ApplicationContext完成加载，但没有被刷新前，该方法被调用
    void contextLoaded(ConfigurableApplicationContext context);
    // 在ApplicationRun Listener启动后，CommandLineRunners和ApplicationRunner未被调用前，该方法被调用
    void started(ConfigurableApplicationContext context);
    // 在run()方法执行完成前该方法被调用
    void running(ConfigurableApplicationContext context);
    // 当应用运行出错时该方法被调用
    void failed(ConfigurableApplicationContext context, Throwable exception);
}

```

```
}
```

如何获取运行监听器？

在 `SpringApplication#run()` 方法中，源码如下：

```
//从spring.factories中获取监听器
SpringApplicationRunListeners listeners = getRunListeners(args);
```

跟进 `getRunListeners()` 方法，其实还是调用了 `loadFactoryNames()` 方法从 `spring.factories` 文件中获取值，如下：

```
org.springframework.boot.SpringApplicationRunListener=\
org.springframework.boot.context.event.EventPublishingRunListener
```

最终注入的是 `EventPublishingRunListener` 这个实现类，创建实例过程肯定是通过反射了，因此我们看看它的构造方法，如下图：

The screenshot shows the constructor of `EventPublishingRunListener`:

```
public EventPublishingRunListener(SpringApplication application, String[] args) {
    this.application = application;
    this.args = args;
    this.initialMulticaster = new SimpleApplicationEventMulticaster();
    for (ApplicationListener<?> listener : application.getListeners()) {
        this.initialMulticaster.addApplicationListener(listener);
    }
}
```

Annotations on the code:

- A blue arrow points to `this.initialMulticaster = new SimpleApplicationEventMulticaster();` with the text "保存创建好的SpringApplication".
- A blue arrow points to `for (ApplicationListener<?> listener : application.getListeners()) {` with the text "构建一个简单应用事件广播器".
- A blue arrow points to `this.initialMulticaster.addApplicationListener(listener);` with the text "将SpringApplication创建过程中设置的监听器全部添加到广播器中".

这个运行监听器内部有一个事件广播器(`SimpleApplicationEventMulticaster`)，主要用来广播特定的事件(`SpringApplicationEvent`)来触发特定的监听器 `ApplicationListener`。

`EventPublishingRunListener` 中的每个方法用来触发 `SpringApplicationEvent` 中的不同子类。

如何启动运行监听器？

在 `SpringApplication#run()` 方法中，源码如下：

```
//执行starting()方法
listeners.starting(bootstrapContext, this.mainApplicationClass);
```

执行 `SpringApplicationRunListeners` 的 `starting()` 方法，跟进去其实很简单，遍历执行上面获取的运行监听器，这里只有一个 `EventPublishingRunListener`。因此执行的是它的 `starting()` 方法，源码如下图：

```
@Override
public void starting(ConfigurableBootstrapContext bootstrapContext) {
    this.initialMulticaster
        .multicastEvent(new ApplicationStartingEvent(bootstrapContext, this.application, this.args));
}
```

上述源码中逻辑很简单，其实只是执行了 `multicastEvent()` 方法，广播了 `ApplicationStartingEvent` 事件。至于 `multicastEvent()` 内部的逻辑感兴趣的可以看看，其实就是遍历 `ApplicationListener` 的实现类，找到监听 `ApplicationStartingEvent` 这个事件的监听器，执行 `onApplicationEvent()` 方法。

总结

这一步其实就是广播了 `ApplicationStartingEvent` 事件来触发监听这个事件的 `ApplicationListener`。

因此如果自定义了 `ApplicationListener` 并且监听了 `ApplicationStartingEvent` (应用程序开始启动) 事件，则这个监听器将会被触发。

2. 环境构建

这一步主要用于加载系统配置以及用户的自定义配置(`application.properties`)，源码如下，在 `run()` 方法中：

```
ConfigurableEnvironment environment = prepareEnvironment(listeners, bootstrapContext,  
applicationArguments);
```

`prepareEnvironment` 方法内部广播了 `ApplicationEnvironmentPreparedEvent` 事件，源码如下图：

```
private ConfigurableEnvironment prepareEnvironment(SpringApplicationRunListeners listeners,  
DefaultBootstrapContext bootstrapContext, ApplicationArguments applicationArguments) {  
    // Create and configure the environment  
    ConfigurableEnvironment environment = getOrCreateEnvironment();  
    configureEnvironment(environment, applicationArguments.getSourceArgs());  
    ConfigurationPropertySources.attach(environment);  
    listeners.environmentPrepared(bootstrapContext, environment);  
    DefaultPropertiesPropertySource.moveToEnd(environment);  
    configureAdditionalProfiles(environment);  
    bindToSpringApplication(environment);  
    if (!this.isCustomEnvironment) {  
        environment = new EnvironmentConverter(getClassLoader()).convertEnvironmentIfNecessary(environment,  
            deduceEnvironmentClass());  
    }  
    ConfigurationPropertySources.attach(environment);  
    return environment;  
}
```

环境构建这一步加载了系统环境配置、用户自定义配置并且广播了 `ApplicationEnvironmentPreparedEvent` 事件，触发监听器。

3. 创建IOC容器

源码在 `run()` 方法中，如下：

```
context = createApplicationContext();
```

跟进代码，真正执行的是 `ApplicationContextFactory` 方法，如下图：

Watermark

```

/**
 * A default {@link ApplicationContextFactory} implementation that will create an
 * appropriate context for the {@link WebApplicationType}.
 */
ApplicationContextFactory DEFAULT = (webApplicationType) -> {
    try {
        switch (webApplicationType) {
            case SERVLET:
                return new AnnotationConfigServletWebServerApplicationContext();
            case REACTIVE:
                return new AnnotationConfigReactiveWebServerApplicationContext();
            default:
                return new AnnotationConfigApplicationContext();
        }
    } catch (Exception ex) {
        throw new IllegalStateException("Unable create a default ApplicationContext instance, "
                + "you may need a custom ApplicationContextFactory", ex);
    }
};

```

根据 `webApplicationType` 决定创建的类型，很显然，我这里的是 `servlet`，因此创建的是 `AnnotationConfigServletWebServerApplicationContext`。

这一步仅仅是创建了 IOC 容器，未有其他操作。

4. IOC容器的前置处理

这一步真是精华了，在刷新容器之前做准备，其中有一个非常关键的操作：将启动类注入容器，为后续的自动化配置奠定基础。源码如下：

```
prepareContext(context, environment, listeners, applicationArguments, printedBanner);
```

`prepareContext()` 源码解析如下图，内容还是挺多的：

```

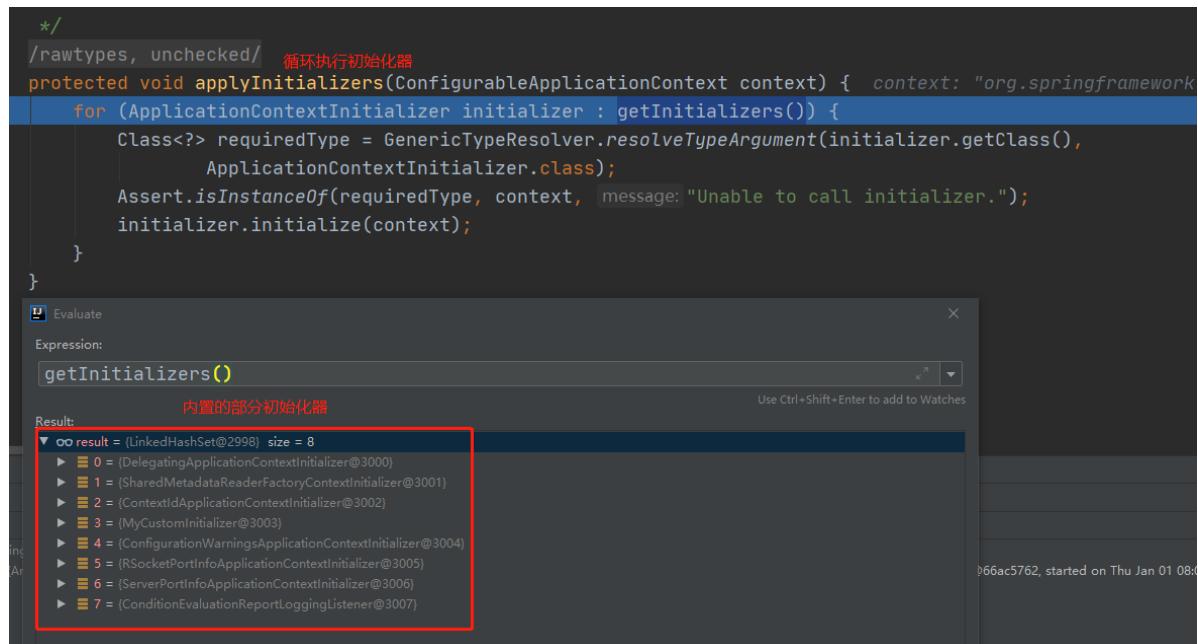
private void prepareContext(DefaultBootstrapContext bootstrapContext, ConfigurableApplicationContext context,
    ConfigurableEnvironment environment, SpringApplicationRunListeners listeners,
    ApplicationArguments applicationArguments, Banner printedBanner) {
    context.setEnvironment(environment);
    postProcessApplicationContext(context);
    applyInitializers(context);
    listeners.contextPrepared(context);
    bootstrapContext.close(context);
    if (this.logStartupInfo) {
        logStartupInfo(isRoot: context.getParent() == null);
        logStartupProfileInfo(context);
    }
    // Add boot specific singleton beans
    ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
    beanFactory.registerSingleton(s: "springApplicationArguments", applicationArguments);
    if (printedBanner != null) {
        beanFactory.registerSingleton(s: "springBootBanner", printedBanner);
    }
    if (beanFactory instanceof DefaultListableBeanFactory) {
        ((DefaultListableBeanFactory) beanFactory)
            .setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
    }
    if (this.lazyInitialization) {    获取启动类指定的参数, 可以是多个
        context.addBeanFactoryPostProcessor(new LazyInitializationBeanFactoryPostProcessor());
    }
    // Load the sources
    Set<Object> sources = getAllSources();
    if (sources.isEmpty()) {
        throw new IllegalStateException("Sources must not be empty");
    }
    load(sources);
    listeners.contextLoaded(context);
}

```

从上图可以看出步骤很多，下面将会详细介绍几个重点的内容。

调用初始化器

在 `SpringApplication` 构建过程中设置的初始化器，从 `spring.factories` 取值的。执行的流程很简单，遍历执行，源码如下图：



```
/*
 * rawtypes, unchecked/ 循环执行初始化器
 */
protected void applyInitializers(ConfigurableApplicationContext context) { context: "org.springframework
    for (ApplicationContextInitializer initializer : getInitializers()) {
        Class<?> requiredType = GenericTypeResolver.resolveTypeArgument(initializer.getClass(),
            ApplicationContextInitializer.class);
        Assert.isInstanceOf(requiredType, context, message: "Unable to call initializer.");
        initializer.initialize(context);
    }
}

```

Evaluate
Expression: `getInitializers()`
Result: 内置的部分初始化器
Result: `result = [LinkedHashSet@2998 size = 8]`

- ▶ 0 = (DelegatingApplicationContextInitializer@3000)
- ▶ 1 = (SharedMetadataReaderFactoryContextInitializer@3001)
- ▶ 2 = (ContextIdApplicationContextInitializer@3002)
- ▶ 3 = (MyCustomInitializer@3003)
- ▶ 4 = (ConfigurationWarningsApplicationContextInitializer@3004)
- ▶ 5 = (RSocketPortInfoApplicationContextInitializer@3005)
- ▶ 6 = (ServerPortInfoApplicationContextInitializer@3006)
- ▶ 7 = (ConditionEvaluationReportLoggingListener@3007)

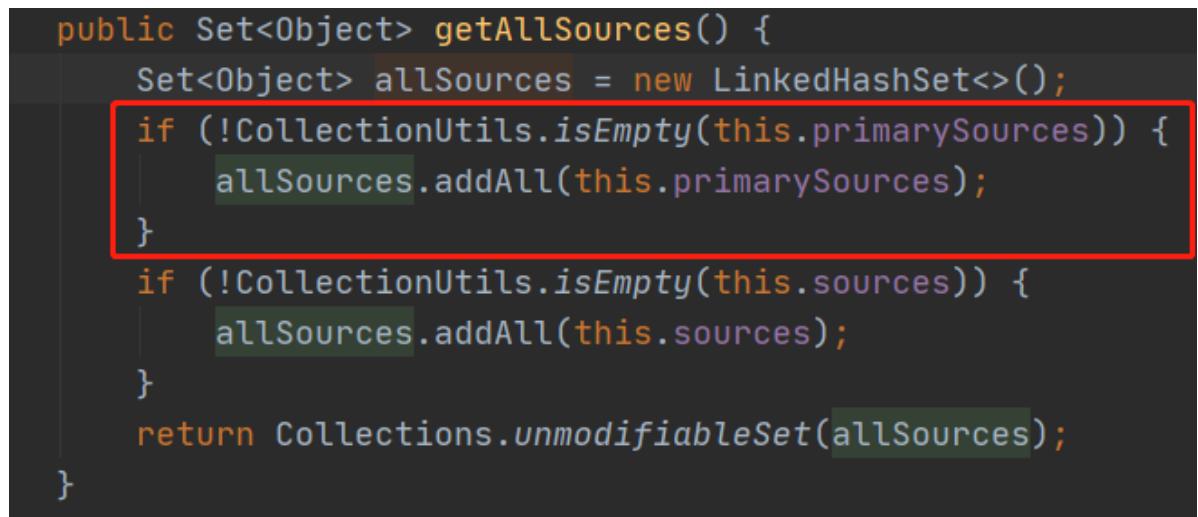
Use Ctrl+Shift+Enter to add to Watches
266ac5762, started on Thu Jan 01 08:00:00 CST 2024

将自定义的 `ApplicationContextInitializer` 放在 `META-INF/spring.factories` 中，在此时也是会被调用。

加载启动类，注入容器

这一步是将主启动类加载到 `IOC容器` 中，作为后续自动配置的入口。

在 `SpringApplication` 构建过程中将主启动类放置在 `primarySources` 这个集合中，此时的 `getAllSources()` 即是从其中取值，如下图：



```
public Set<Object> getAllSources() {
    Set<Object> allSources = new LinkedHashSet<>();
    if (!CollectionUtils.isEmpty(this.primarySources)) {
        allSources.addAll(this.primarySources);
    }
    if (!CollectionUtils.isEmpty(this.sources)) {
        allSources.addAll(this.sources);
    }
    return Collections.unmodifiableSet(allSources);
}
```

这里取出的就是主启动类，当然你的项目中可能不止一个，接下来就是将其加载到IOC容器中了，源码如下：

跟着代码进去，其实主要逻辑都在 `BeanDefinitionLoader.load()` 方法，如下图：

```

private void load(Class<?> source) { source: "class com.example.annotation_demo.AnnotationDemoAppli
    if (isGroovyPresent() && GroovyBeanDefinitionSource.class.isAssignableFrom(source)) { source: "o
        // Any GroovyLoaders added in beans{} DSL can contribute beans here
        GroovyBeanDefinitionSource loader = BeanUtils.instantiateClass(source, GroovyBeanDefinitionSo
        ((GroovyBeanDefinitionReader) this.groovyReader [NullPointerException] ).beans(loader.getBeans());
    }
    if (isEligible(source)) {
        this.annotatedReader.register(source);
    }
}

```

将主启动类加载到 beanDefinitionMap，后续该启动类将作为开启自动配置化的入口，后续章节详细介绍。

两次广播事件

这一步涉及到了两次事件广播，分别是 ApplicationContextInitializedEvent 和 ApplicationContextPreparedEvent，对应的源码如下：

```

listeners.contextPrepared(context);
load(context, sources.toArray(new Object[0]));

```

5. 刷新容器

刷新容器完全是 Spring 的功能了，比如初始化资源，初始化上下文广播器等，这个就不再详细介绍，有兴趣可以看看 Spring 的源码。

```

protected void refresh(ApplicationContext applicationContext) {
    Assert.isInstanceOf(AbstractApplicationContext.class, applicationContext);
    //调用创建的容器applicationContext中的refresh()方法
    ((AbstractApplicationContext)applicationContext).refresh();
}

public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        /**
         * 刷新上下文环境
         */
        prepareRefresh();

        /**
         * 初始化BeanFactory，解析XML，相当于之前的XmlBeanFactory的操作，
         */
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        /**
         * 为上下文准备BeanFactory，即对BeanFactory的各种功能进行填充，如常用的注解@Autowired
         @Qualifier等
         * 添加ApplicationContextAwareProcessor处理器
         * 在依赖注入忽略实现*Aware的接口，如EnvironmentAware、ApplicationEventPublisherAware等
         * 注册依赖，如一个bean的属性中含有ApplicationEventPublisher(beanFactory)，则会将beanFactory
         的实例注入进去
         */
        createBeanFactory(beanFactory);
    }
}

```

Watermark

```

try {
    /**
     * 提供子类覆盖的额外处理，即子类处理自定义的BeanFactoryPostProcess
     */
}

```

```
postProcessBeanFactory(beanFactory);

    /**
     * 激活各种BeanFactory处理器, 包括BeanDefinitionRegistryBeanFactoryPostProcessor和普通的
     BeanFactoryPostProcessor
        * 执行对应的postProcessBeanDefinitionRegistry方法 和 postProcessBeanFactory方法
        */
    invokeBeanFactoryPostProcessors(beanFactory);

    /**
     * 注册拦截Bean创建的Bean处理器, 即注册BeanPostProcessor, 不是
     BeanFactoryPostProcessor, 注意两者的区别
        * 注意, 这里仅仅是注册, 并不会执行对应的方法, 将在bean的实例化时执行对应的方法
        */
    registerBeanPostProcessors(beanFactory);

    /**
     * 初始化上下文中的资源文件, 如国际化文件的处理等
     */
    initMessageSource();

    /**
     * 初始化上下文事件广播器, 并放入applicationEventMulticaster, 如
     ApplicationEventPublisher
        */
    initApplicationEventMulticaster();

    /**
     * 给子类扩展初始化其他Bean
     */
    onRefresh();

    /**
     * 在所有bean中查找listener bean, 然后注册到广播器中
     */
    registerListeners();

    /**
     * 设置转换器
     * 注册一个默认的属性值解析器
     * 冻结所有的bean定义, 说明注册的bean定义将不能被修改或进一步的处理
     * 初始化剩余的非惰性的bean, 即初始化非延迟加载的bean
     */
    finishBeanFactoryInitialization(beanFactory);

    /**
     * 通过spring的事件发布机制发布ContextRefreshedEvent事件, 以保证对应的监听器做进一步的
     处理
        * 即对那种在spring启动后需要处理的一些类, 这些类实现了
     ApplicationListener<ContextRefreshedEvent>,
            * 这里就是要触发这些类的执行(执行onApplicationEvent方法)
            * 另外, spring的内置Event: ContextClosedEvent、ContextRefreshedEvent、
     ContextStartEvent、ContextStopEvent、RequestHandleEvent
            * 完成初始化, 通知生命周期处理器lifeCycleProcessor刷新过程, 同时发出
     ContextRefreshEvent通知其他人
            */
    finishRefresh();
}
```

```
        finally {
            resetCommonCaches();
        }
    }
}
```

6. IOC容器的后置处理

一个扩展方法，源码如下：

```
afterRefresh(context, applicationArguments);
```

默认为空，如果有自定义需求可以重写，比如打印一些启动结束日志等。

7. 发出结束执行的事件

同样是 `EventPublishingRunListener` 这个监听器，广播 `ApplicationStartedEvent` 事件。

但是这里广播事件和前几次不同，并不是广播给 `SpringApplication` 中的监听器（在构建过程中从 `spring.factories` 文件获取的监听器）。因此在 IOC 容器中注入的监听器（使用 `@Component` 等方式注入的）也能够生效。前面几个事件只有在 `spring.factories` 文件中设置的监听器才会生效。

跟着代码进入，可以看到 `started()` 方法源码如下：

```
@Override
public void started(ConfigurableApplicationContext context) {
    context.publishEvent(new ApplicationStartedEvent(this.application, this.args, context));
    AvailabilityChangeEvent.publish(context, LivenessState.CORRECT);
}
```

这里并没有用事件广播器 `SimpleApplicationEventMulticaster` 广播事件，而是使用 `ConfigurableApplicationContext` 直接在 IOC 容器中发布事件。

8. 执行Runners

Spring Boot 提供了两种 `Runner` 让我们定制一些额外的操作，分别是 `CommandLineRunner` 和 `ApplicationRunner`，关于这两个的区别，后面文章详细介绍。

调用的源码如下：

```
callRunners(context, applicationArguments);
```

跟进代码，其实真正调执行的是如下方法：

Watermark

```

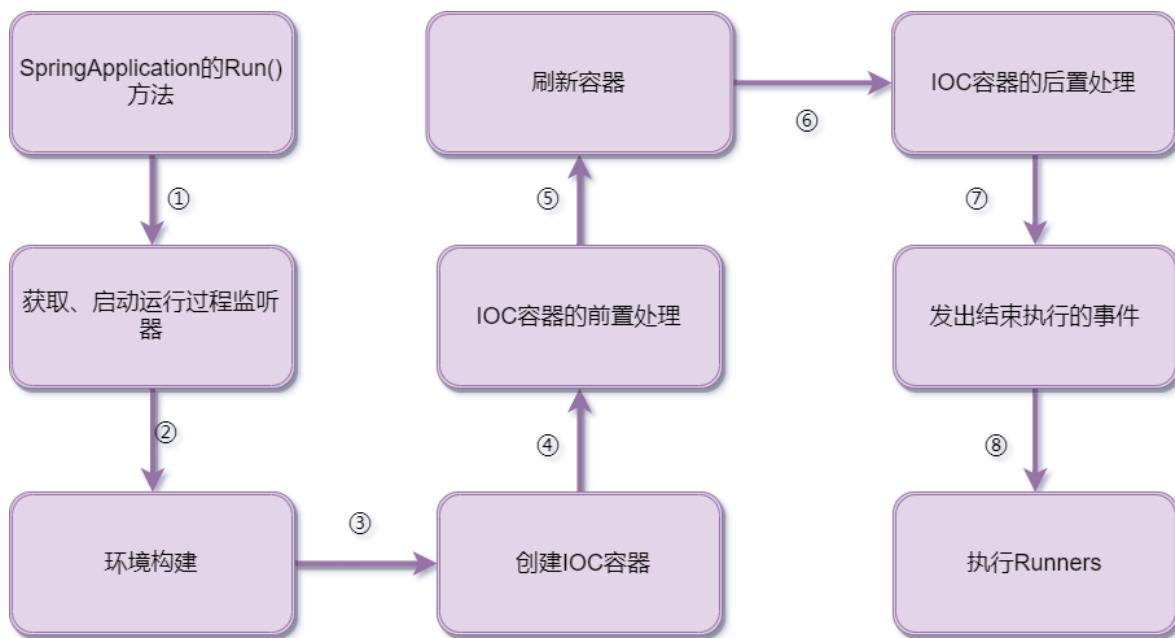
private void callRunners(ApplicationContext context, ApplicationArguments args) {
    List<Object> runners = new ArrayList<>();
    runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());
    runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());
    AnnotationAwareOrderComparator.sort(runners);
    for (Object runner : new LinkedHashSet<>(runners)) {
        if (runner instanceof ApplicationRunner) {
            callRunner((ApplicationRunner) runner, args);
        }
        if (runner instanceof CommandLineRunner) {
            callRunner((CommandLineRunner) runner, args);
        }
    }
}

```

逻辑很简单，从 IOC 容器 中获取，遍历调用。

总结

Spring Boot 启动流程相对简单些，作者将其细分了以上八个步骤，希望能够帮助读者理解，流程图如下：



总结

Spring Boot 启动流程就介绍到这里了，需要重点理解 run() 方法执行的八个步骤以及事件、初始化器、监听器等组件的执行时间点。

作者每一篇文章都很用心，这篇源码解析花了三天时间精雕细琢，力求讲解的通俗易懂，希望能够帮助到你。

另外作者的第一本 PDF 书籍已经整理好了，由浅入深的详细介绍了 Mybatis 基础以及底层源码，有需要的朋友可以[访问我的 Mybatis 排版项目](#)，目录如下：



Spring Boot 自动配置源码解析

前言

为什么 Spring Boot 这么火？因为便捷，开箱即用，但是你思考过为什么会这么便捷吗？传统的SSM架构配置文件至少要写半天，而使用 Spring Boot 之后只需要引入一个 `starter` 之后就能直接使用，why？

原因很简单，每个 `starter` 内部做了工作，比如 Mybatis 的启动器默认内置了可用的 `SqlSessionFactory`。

至于如何内置的？`Spring Boot` 又是如何使其生效的？这篇文章就从源码角度介绍一下`Spring Boot` 的自动配置原理。

源码版本

作者`Spring Boot` 是基于 2.4.0。每个版本有些变化，读者尽量和我保持一致，以防源码有些出入。

@SpringBootApplication干了什么？

这么说吧，这个注解什么也没做，废物，活都交给属下做了，源码如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {}
```

上方标注了三个重要的注解，如下：

1. `@SpringBootConfiguration`：其实就是`@Configuration`，因此主启动类可以当做配置类使用，比如注入`Bean`等。
2. `@EnableAutoConfiguration`：这个注解牛批了，名字就不一样，开启自动配置，哦，关键都在这里了.....
3. `@ComponentScan`：包扫描注解。

经过以上的分析，最终定位了一个注解`@EnableAutoConfiguration`，顾名思义，肯定和自动配置有关，要重点分析下。

@EnableAutoConfiguration干了什么？

想要知道做了什么肯定需要看源码，如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {}
```

上方标注了两个重要的注解，如下：

1. `@AutoConfigurationPackage`：自动配置包注解，默认将主配置类(`@SpringBootApplication`)所在的包及其子包里面的所有组件扫描到 IOC 容器 中。
2. `@Import`：该注解不必多说了，前面文章说过很多次了，这里是导入了`AutoConfigurationImportSelector`，用来注入自动配置类。

以上只是简单的分析了两个注解，下面将会从源码详细的介绍一下。

@AutoConfigurationPackage

这个注解干了什么？这个需要看下源码，如下；

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {}
```

重要的还是 `@Import` 注解，导入了 `AutoConfigurationPackages.Registrar`，这个类是干什么的？源码如下图：

其实就两个方法，但是最重要的就是 `registerBeanDefinitions` 方法，但是这个方法不用看，肯定是注入 Bean，这里的重点是注入哪些 Bean，重点源码如下：

```
//获取扫描的包
new PackageImports(metadata).getPackageNames().toArray(new String[0])
```

跟进代码，主要逻辑都在 `#PackageImports.PackageImports()` 这个构造方法中，源码解析如下图：

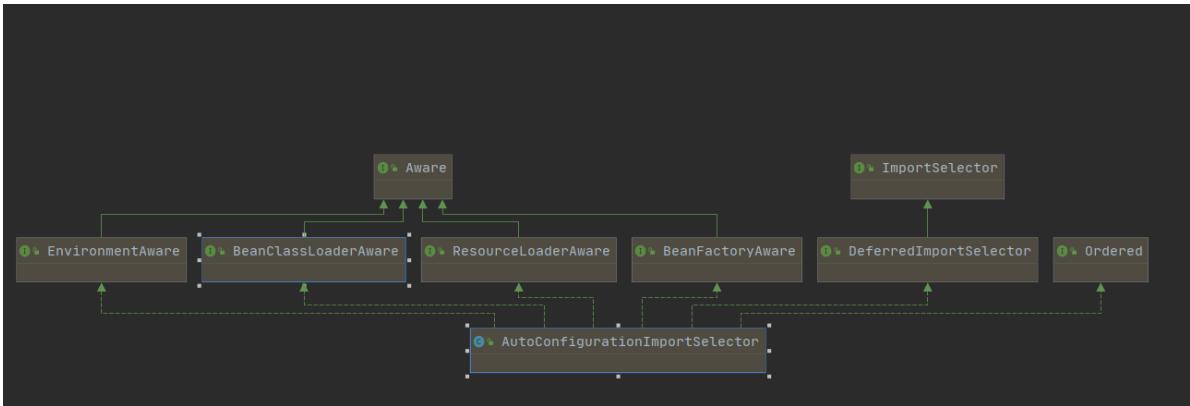
从上面源码分析可以知道，这里扫描的包名是由两部分组成，分别如下：

1. 从 `@AutoConfigurationPackage` 注解中的两个属性解析得来的包名。
2. 注解 `AutoConfigurationPackage` 所在的包名，即是 `@SpringBootApplication` 所在的包名。

`@AutoConfigurationPackage` 将主配置类(`@SpringBootApplication`)所在的包及其子包里面的所有组件扫描到IOC容器中。

@Import(AutoConfigurationImportSelector.class)

这个注解不用多说了，最重要的就是 `AutoConfigurationImportSelector`，我们来看看它的继承关系，如下图：



这个类的继承关系还是挺简单的，实现了 Spring 中的 `xxAware` 注入一些必要的组件，但是最值得关心的是实现了一个 `DeferredImportSelector` 这个接口，这个接口扩展了 `ImportSelector`，也改变了其运行的方式，这个在后面章节会介绍。

注意：这个类会导致一个误区，平时看到 `xxxSelector` 已经有了反射弧了，肯定会在 `selectImports()` 方法上 `DEBUG`，但是这个类压根就没执行该方法，我第一次看也有点怀疑人生了，原来它走的是 `DeferredImportSelector` 的接口方法。

其实该类真正实现逻辑的方法是 `process()` 方法，但是主要加载自动配置类的任务交给了 `getAutoConfigurationEntry()` 方法，具体的逻辑如下图：

```
protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
    configurations = removeDuplicates(configurations); // ④ 去掉重复的自动配置类
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    configurations.removeAll(exclusions);
    configurations = getConfigurationClassFilter().filter(configurations);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return new AutoConfigurationEntry(configurations, exclusions);
}
```

上图的逻辑很简单，先从 `spring.factories` 文件中获取自动配置类，在去掉 `@SpringBootApplication` 中定义排除的自动配置类。

上图中的第 ④ 步就是从 `META-INF/spring.factories` 中加载自动配置类，代码很简单，在上一篇分析启动流程的时候也有很多组件是从 `spring.factories` 文件中加载的，代码都类似。

在 `springboot-autoconfigure` 中的 `spring.factories` 文件内置了很多自动配置类，如下：

Watermark

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\ \
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\ \
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\ \
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\ \
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\ \
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\ \
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\ \
org.springframework.boot.autoconfigure.context.LifecycleAutoConfiguration,\ \
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\ \
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\ \
.....
```

了解了 Spring Boot 如何加载自动配置类，那么自定义一个自动配置类也是很简单的，后续章节教你如何定制自己的自动配置类，里面还是有很多门道的.....

总结

本文从源码角度分析了 Spring Boot 的自动配置是如何加载的，其实分析起来很简单，希望作者的这篇文章能帮助你更深层次的了解 Spring Boot 。

另外作者的第一本 PDF 书籍已经整理好了，由浅入深的详细介绍了Mybatis基础以及底层源码，有需要的朋友公众号回复关键词**Mybatis进阶**即可获取，目录如下：

The screenshot shows a PDF document with a table of contents on the left and the book cover on the right.

Table of Contents:

- ▶ Mybatis入门之基本CRUD
- ▶ Mybatis入门之结果映射
- ▶ Mybatis动态SQL，你真的会了吗？
- ▶ Mybatis几种传参方式，你了解吗？
- ▶ Mybatis中Mapper接口的方法为什么不能重载？
- ▶ Mybatis中的TypeHandler你真的会用吗？
- ▶ Mybatis的插件原理以及如何实现？
- ▶ Mybatis源码阅读之六剑客
- ▶ Mybatis源码如何阅读，教你一招
- ▶ Mybatis如何执行Select语句，你真的知道吗？
- ▶ Mybatis Log plugin破解

Book Cover:

The book cover features a purple and blue abstract background with a forest silhouette at the top. The title "Mybatis 进阶" is written vertically in large white font. On the left side, there is vertical text: "著 / 不才陈某" and "入门到放弃的进阶". At the bottom right, it says "不才陈某". Below the cover, there is a short description: "本书从Mybatis的基础到源码详细的介绍了每个知识点，个人原创，请勿商用。" and the publisher information: "码 猿 技 术 专 栏".

Watermark

码了两本书的陈某，邀你一起进步



码猿技术专栏

回复以下关键词获取专栏书籍

▶ Mybatis 进阶

▶ Spring Boot 进阶

写最精的文章，做最野的程序员

自定义启动器

前言

日常工作中对于 Spring Boot 提供的一些启动器可能已经足够使用了，但是不可避免的需要自定义启动器，比如整合一个陌生的组件，也想要达到开箱即用的效果。

在上一章节从底层源码介绍了 Spring Boot 的自动配置的原理，未读过的朋友建议看一下：[Spring Boot 自动配置源码解析](#)

这篇文章将会介绍如何自定义一个启动器，同时对于自动配置类的执行顺序做一个详细的分析。

如何自定义一个starter?

启动器的核心其实就是自动配置类，在自动配置源码分析的章节已经介绍过，

`AutoConfigurationImportSelector` 是从 `spring.factories` 中加载自动配置类，因此只需要将自定义的自动配置类设置在该文件中即可。

读过源码的朋友都知道自动配置类常用的一些注解，总结如下：

1. `@Configuration`：该注解标志这是一个配置类，**自动配置类可以不加该注解**。
2. `@EnableConfigurationProperties`：这个配置也是经常使用了，使得指定的属性配置生效。一般自动配置类都需要从全局属性配置中读取自定义的配置，这就是一个开关。
3. `@ConditionalOnXXX`：该注解是自动配置类的核心了，自动配置类既要启动时自动配置，又要保证用户自定义的配置覆盖掉自动配置，该注解就是一个条件语句，只有当指定条件成立才会执行某操作。不理解的，请看作者前面的一篇文章：[这类注解都不知道，还说用过Spring Boot~](#)
4. `@AutoConfigureAfter`：指定自动配置类的执行先后顺序，下文详细介绍。
5. `@AutoConfigureBefore`：指定自动配置类的执行先后顺序，下文详细介绍。

6. `@AutoConfigureOrder` : 指定自动配置类的优先级, 下文详细介绍。

有了以上准备, 自定义一个 `starter` 非常简单, 分为两个步骤。

1. 准备自己的自动配置类

启动器的灵魂核心就是自动配置类, 因此需要首先创建一个自动配置类, 如下:

```
@Configuration  
{@AutoConfigureAfter (DataSourceAutoConfiguration.class)  
@AutoConfigureOrder (Ordered.HIGHEST_PRECEDENCE+5)  
@ConditionalOnProperty (prefix = "my.auto", name = "enabled", havingValue = "true", matchIfMissing = true)  
public class MyCustomAutoConfiguration {  
}  
}
```

以上自动配置类只是作者简单的按照格式随手写了一个, 真实开发中需要根据启动器的业务做默认配置。

2. 将自动配置类设置在spring.factories

标注了 `@Configuration` 注解的自动配置类如果不放在 `spring.factories` 文件中, 仅仅是一个普通的配置类而已。想要其成为自动配置类, 需要在 `spring.factories` 文件中设置, 如下:

```
# Auto Configure  
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
com.example.autoconfig.MyCustomAutoConfiguration
```

经过以上的配置, 粗略的启动器完成了, 只需要打包, 然后 `Maven` 引入即可工作。

如何指定自动配置类的执行顺序?

自动配置类需要定义执行顺序吗? 答案: 肯定的。比如 `Mybatis` 的自动配置类, 肯定要在数据源的自动配置类之后执行, 否则如何创建 `SqlSessionFactory` ?

如何自定义自动配置类的执行顺序呢? 此时就需要用到上文提到的三个注解, 如下:

1. `@AutoConfigureAfter` : 当前配置类在指定配置类之后执行
2. `@AutoConfigureBefore` : 当前配置类在指定配置类之前执行
3. `@AutoConfigureOrder` : 指定优先级, 数值越小, 优先级越高。

分享一个经典的误区

对于 `Spring Boot` 不是很了解的人写出的代码真是不堪入目, 曾经看过有人在普通的配置类上使用 `@AutoConfigurexxx` 注解, 如下;

```
@Configuration  
{@AutoConfigureBefore (Config2.class)  
public class Config1{  
}  
}  
Watermark  
@Configuration  
public class Config2{  
}
```

是不是感觉很爽, 原来还能这么指定配置类的执行顺序..... (此处省略一万字)

可能有时候走了狗屎运给你一种错觉还真的配置成功了。实际上这种方式是不可行的，以上三个注解只有针对自动配置类才会生效。

源码分析自动配置类如何排序？

其实关键的代码还是在 `AutoConfigurationImportSelector` 中，将自动配置类从 `spring.factories` 加载出来之后会根据条件排序，在 `selectImports()` 方法中最后一行代码如下：

```
return sortAutoConfigurations(processedConfigurations, getAutoConfigurationMetadata()).stream()
    .map((importClassName) -> new Entry(this.entries.get(importClassName),
importClassName))
    .collect(Collectors.toList());
```

上面的代码则是将排序好的自动配置类返回，跟进代码，发现最终的实现都在

`AutoConfigurationSorter.getInPriorityOrder()` 方法中，逻辑如下图：

```
List<String> getInPriorityOrder(Collection<String> classNames) {
    AutoConfigurationClasses classes = new AutoConfigurationClasses(this.metadataReaderFactory,
        this.autoConfigurationMetadata, classNames);
    List<String> orderedClassNames = new ArrayList<>(classNames);
    // Initially sort alphabetically
    Collections.sort(orderedClassNames); // 按照字母进行排序
    // Then sort by order
    orderedClassNames.sort((o1, o2) -> {
        int i1 = classes.get(o1).getOrder();
        int i2 = classes.get(o2).getOrder();
        return Integer.compare(i1, i2);
    });
    // Then respect @AutoConfigureBefore @AutoConfigureAfter
    orderedClassNames = sortByAnnotation(classes, orderedClassNames);
    return orderedClassNames;
}
```

具体的流程如上图，排序也是按照先后顺序，如下：

1. 先按照字母排序
2. 按照 `@AutoConfigureOrder` 优先级排序
3. 最终按照 `@AutoConfigureAfter`、`@AutoConfigureBefore` 排序

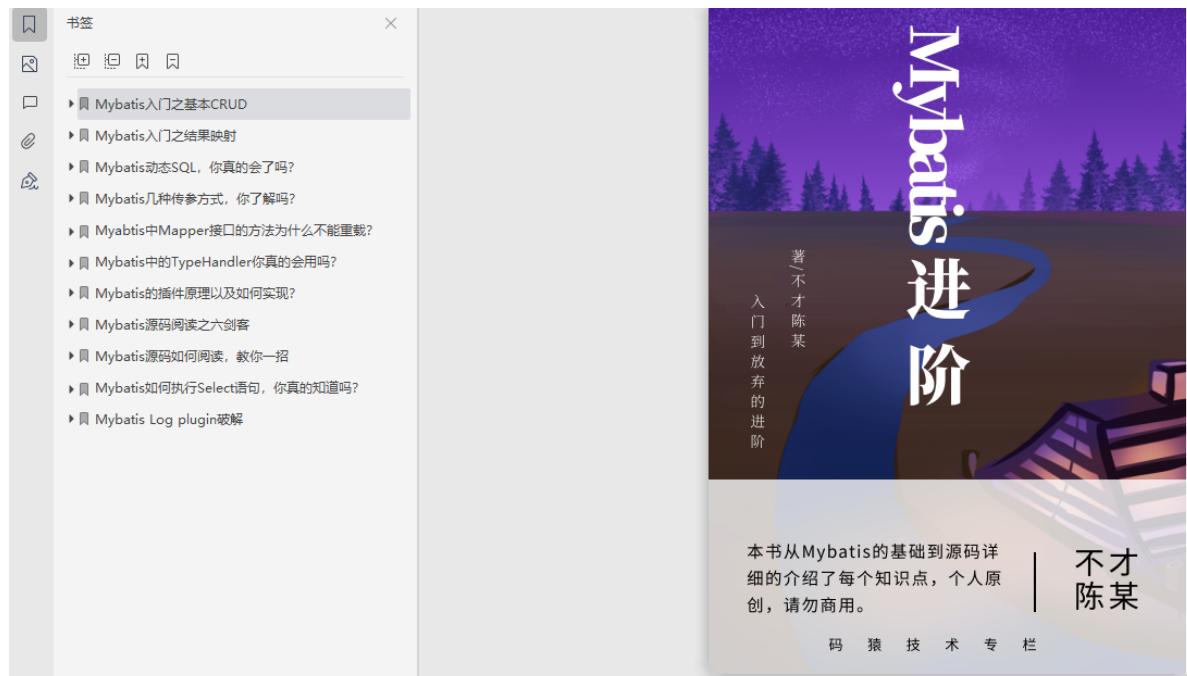
从上面配置的顺序可以知道，最终决定权还是在 `@AutoConfigureAfter`、`@AutoConfigureBefore` 这两个注解。

总结

本文介绍了如何自定义一个启动器以及指定自动配置类的执行顺序，通过作者的介绍，希望读者们能够理解并灵活运用。

另外作者的第一本 PDF 书籍已经整理好了，由浅入深的详细介绍了Mybatis基础以及底层源码，有需要的朋友公众号回复关键词 **Mybatis进阶** 即可获取，目录如下：

Watermark



Spring Boot 配置如何动态刷新？

前言

在实际工作中总是需要在项目启动时做一些初始化的操作，比如初始化线程池、提前加载好加密证书.....

那么经典问题来了，这也是面试官经常会问到的一个问题：有哪些手段在Spring Boot 项目启动的时候做一些事情？

方法有很多种，下面介绍几种常见的方法。

1、监听容器刷新完成扩展点

ApplicationListener<ContextRefreshedEvent>

ApplicationContext事件机制是观察者设计模式实现的，通过ApplicationEvent和ApplicationListener这两个接口实现ApplicationContext的事件机制。

Spring中一些内置的事件如下：

1. `ContextRefreshedEvent`： ApplicationContext 被初始化或刷新时，该事件被发布。这也可以在 ConfigurableApplicationContext 接口中使用 `refresh()` 方法来发生。此处的初始化是指：所有的 Bean 被成功装载，后处理 Bean 被检测并激活，所有 Singleton Bean 被预实例化， ApplicationContext 容器已就绪可用。
2. `ContextStartedEvent`： 当使用 ConfigurableApplicationContext (ApplicationContext 子接口) 接口中的 `start()` 方法启动 ApplicationContext 时，该事件被发布。你可以调查你的数据库，或者你可以在接受到这个事件后重启任何停止的应用程序。
3. `ContextStoppedEvent`： 当使用 ConfigurableApplicationContext 接口中的 `stop()` 停止 ApplicationContext 时，发布这个事件。你可以在接受到这个事件后做必要的清理的工作。
4. `ContextClosedEvent`： 当使用 ConfigurableApplicationContext 接口中的 `close()` 方法关闭 ApplicationContext 时，该事件被发布。一个已关闭的上下文到达生命周期末端；它不能被刷新或重启。
5. `RequestHandledEvent`： 这是一个 web-specific 事件，告诉所有 bean HTTP 请求已经被服务。只能应用于使用 DispatcherServlet 的 Web 应用。在使用 Spring 作为前端的 MVC 控制器时，当 Spring 处理用户请求结束后，系统会自动触发该事件。

好了，了解上面这些内置事件后，我们可以监听 `ContextRefreshedEvent` 在 Spring Boot 启动时完成一些操作，代码如下：

```
@Component
public class TestApplicationListener implements ApplicationListener<ContextRefreshedEvent> {
    @Override
    public void onApplicationEvent(ContextRefreshedEvent contextRefreshedEvent) {
        System.out.println(contextRefreshedEvent);
        System.out.println("TestApplicationListener.....");
    }
}
```

高级玩法

可以自定事件完成一些特定的需求，比如：邮件发送成功之后，做一些业务处理。

1. 自定义 EmailEvent，代码如下：

```
public class EmailEvent extends ApplicationEvent {
    private String address;
    private String text;
    public EmailEvent(Object source, String address, String text) {
        super(source);
        this.address = address;
        this.text = text;
    }
    public EmailEvent(Object source) {
        super(source);
    }
    //.....address和text的setter、getter
}
```

2. 自定义监听器，代码如下：

```
public class EmailNotifier implements ApplicationListener{
    public void onApplicationEvent(ApplicationEvent event) {
        if (event instanceof EmailEvent) {
            EmailEvent emailEvent = (EmailEvent)event;
            System.out.println("邮件地址: " + emailEvent.getAddress());
            System.out.println("邮件内容: " + emailEvent.getText());
        } else {
            System.out.println("容器本身事件: " + event);
        }
    }
}
```

3. 发送邮件后，触发事件，代码如下：

```
public class SpringTest {
    public static void main(String args[]){
        ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
        //创建一个ApplicationEvent对象
        EmailEvent event = new EmailEvent("hello", "abc@163.com", "This is a test");
        //主动触发该事件
        context.publishEvent(event);
    }
}
```

2、SpringBoot 的 CommandLineRunner 接口

当容器初始化完成之后会调用 `CommandLineRunner` 中的 `run()` 方法，同样能够达到容器启动之后完成一些事情。这种方式和 `ApplicationListener` 相比更加灵活，如下：

- 不同的 `CommandLineRunner` 实现可以通过 `@Order()` 指定执行顺序
- 可以接收从控制台输入的参数。

下面自定义一个实现类，代码如下：

```
@Component
@Slf4j
public class CustomCommandLineRunner implements CommandLineRunner {

    /**
     * @param args 接收控制台传入的参数
     */
    @Override
    public void run(String... args) throws Exception {
        log.debug("从控制台接收参数>>>" + Arrays.asList(args));
    }
}
```

运行这个命令如下：

```
java -jar demo.jar aaa bbb ccc
```

以上命令中传入了三个参数，分别是 `aaa`、`bbb`、`ccc`，这三个参数将会被 `run()` 方法接收到。如下图：

```
@Component
@Slf4j
public class CustomCommandLineRunner implements CommandLineRunner {

    /**
     * @param args 接收控制台传入的参数
     */
    @Override
    public void run(String... args) throws Exception {
        log.debug("从控制台接收参数>>>"+ Arrays.asList(args));
    }
}
```

源码分析

读过我的文章的铁粉都应该知道 `CommandLineRunner` 是如何执行的，原文：[头秃系列，二十三张图带你从源码分析Spring Boot 启动流程~](#)

Spring Boot 加载上下文的入口在 `org.springframework.context.ConfigurableApplicationContext()` 这个方法中，如下图：

```
public ConfigurableApplicationContext run(String... args) {
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    ConfigurableApplicationContext context = null;
    Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();
    configureHeadlessProperty();
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();
    try {
        ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
        ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);
        configureIgnoreBeanInfo(environment);
        Banner printedBanner = printBanner(environment);
        context = createApplicationContext();
        exceptionReporters = getSpringFactoriesInstances(SpringBootExceptionReporter.class,
            new Class[] { ConfigurableApplicationContext.class }, context);
        prepareContext(context, environment, listeners, applicationArguments, printedBanner);
        refreshContext(context);
        afterRefresh(context, applicationArguments);
        stopWatch.stop();
        if (this.logStartupInfo) {
            new StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(), stopWatch);
        }
        listeners.started(context);
        callRunners(context, applicationArguments);
    }
    catch (Throwable ex) {
```

调用`CommandLineRunner`在 `callRunners(context, applicationArguments);`这个方法中执行，源码如下图：

Watermark

```
private void callRunners(ApplicationContext context, ApplicationArguments args) {
    List<Object> runners = new ArrayList<>();
    runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());
    runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());
    AnnotationAwareOrderComparator.sort(runners);
    for (Object runner : new LinkedHashSet<>(runners)) {
        if (runner instanceof ApplicationRunner) {
            callRunner((ApplicationRunner) runner, args);
        }
        if (runner instanceof CommandLineRunner) {
            callRunner((CommandLineRunner) runner, args);
        }
    }
}
```

3、SpringBoot 的 ApplicationRunner 接口

`ApplicationRunner` 和 `CommandLineRunner` 都是 Spring Boot 提供的，相对于 `CommandLineRunner` 来说对于控制台传入的参数封装更好一些，可以通过键值对来获取指定的参数，比如 `--version=2.1.0`。

此时运行这个jar命令如下：

```
java -jar demo.jar --version=2.1.0 aaa bbb ccc
```

以上命令传入了四个参数，一个键值对 `version=2.1.0`，另外三个分别是 `aaa`、`bbb`、`ccc`。

同样可以通过 `@Order()` 指定优先级，如下代码：

```
@Component
@Slf4j
public class CustomApplicationRunner implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws Exception {
        log.debug("控制台接收的参数: {}, {}",
        {}, args.getOptionNames(), args.getNonOptionArgs(), args.getSourceArgs());
    }
}
```

通过以上命令运行，结果如下图：

源码分析

和 `CommandLineRunner` 一样，同样在 `callRunners()` 这个方法中执行，源码如下图：

```
private void callRunners(ApplicationContext context, ApplicationArguments args) {
    List<Object> runners = new ArrayList<>();
    runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());
    runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());
    AnnotationAwareOrderComparator.sort(runners);
    for (Object runner : new LinkedHashSet<>(runners)) {
        if (runner instanceof ApplicationRunner) {
            callRunner((ApplicationRunner) runner, args);
        }
        if (runner instanceof CommandLineRunner) {
            callRunner((CommandLineRunner) runner, args);
        }
    }
}
```

4、`@PostConstruct` 注解

前三种针对的是容器的初始化完成之后做的一些事情，`@PostConstruct` 这个注解是针对 Bean 的初始化完成之后做一些事情，比如注册一些监听器...

`@PostConstruct` 注解一般放在 Bean 的方法上，一旦 Bean 初始化完成之后，将会调用这个方法，代码如下：

```
@Component
@Slf4j
public class SimpleExampleBean {

    @PostConstruct
    public void init() {
        log.debug("Bean初始化完成，调用.....");
    }
}
```

5、`@Bean`注解中指定初始化方法

这种方式和 `@PostConstruct` 比较类似，同样是指定一个方法在 Bean 初始化完成之后调用。

新建一个 Bean，代码如下：

```
@Slf4j
public class SimpleExampleBean {

    public void init() {
        log.debug("Bean初始化完成，调用.....");
    }
}
```

在配置类中通过 `@Bean` 实例化这个 Bean，不过 `@Bean` 中的 `initMethod` 这个属性需要指定初始化之后需要执行的方法，如下：

```
@Bean(initMethod = "init")
public SimpleExampleBean simpleExampleBean() {
    return new SimpleExampleBean();
}
```

6、 InitializingBean 接口

InitializingBean 的用法基本上与 @PostConstruct 一致，只不过相应的 Bean 需要实现 afterPropertiesSet 方法，代码如下：

```
@Slf4j
@Component
public class SimpleExampleBean implements InitializingBean {

    @Override
    public void afterPropertiesSet() {
        log.debug("Bean初始化完成，调用.....");
    }
}
```

总结

实现方案有很多，作者只是总结了常用的六种，学会的点个赞。

Springboot 日志、配置文件、接口数据如何脱敏？

一、前言

核心隐私数据无论对于企业还是用户来说尤其重要，因此要想办法杜绝各种隐私数据的泄漏。下面陈某带大家从以下三个方面讲解一下隐私数据如何脱敏，也是日常开发中需要注意的：

1. 配置文件数据脱敏
2. 接口返回数据脱敏
3. 日志文件数据脱敏

文章目录如下
Watermark

Spring Boot 数据脱敏

前言 接口返回数据如何脱敏? 配置文件如何脱敏? 日志文件如何数据脱敏?

二、配置文件如何脱敏?

经常会遇到这样一种情况：项目的配置文件中总有一些敏感信息，比如数据源的url、用户名、密码...这些信息一旦被暴露那么整个数据库都将会被泄漏，那么如何将这些配置隐藏呢？

以前都是手动将加密之后的配置写入到配置文件中，提取的时候再手动解密，当然这是一种思路，也能解决问题，但是每次都要手动加密、解密不觉得麻烦吗？



今天介绍一种方案，让你在无感知的情况下实现配置文件的加密、解密。利用一款开源插件：[jasypt-spring-boot](#)。项目地址如下：

<https://github.com/ulisesbocchio/jasypt-spring-boot>

使用方法很简单，整合Spring Boot 只需要添加一个 [starter](#)。

Watermark

1. 添加依赖

```
<dependency>
    <groupId>com.github.ulisesbocchio</groupId>
    <artifactId>jasypt-spring-boot-starter</artifactId>
    <version>3.0.3</version>
</dependency>
```

2. 配置秘钥

在配置文件中添加一个加密的秘钥（任意），如下：

```
jasypt:
  encryptor:
    password: Y6M9fAJQdU7jNp5MW
```

当然将秘钥直接放在配置文件中也是不安全的，我们可以在项目启动的时候配置秘钥，命令如下：

```
java -jar xxx.jar -Djasypt.encryptor.password=Y6M9fAJQdU7jNp5MW
```

3. 生成加密后的数据

这一步骤是将配置明文进行加密，代码如下：

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

    /**
     * 注入加密方法
     */
    @Autowired
    private StringEncryptor encryptor;

    /**
     * 手动生成密文，此处演示了url, user, password
     */
    @Test
    public void encrypt() {
        String url = encryptor.encrypt("jdbc:mysql://127.0.0.1:3306/test?
useUnicode=true&characterEncoding=UTF-
8&zeroDateTimeBehavior=convertToNull&useSSL=false&allowMultiQueries=true&serverTimezone=Asia/Shang
hai");
        String name = encryptor.encrypt("root");
        String password = encryptor.encrypt("123456");
        System.out.println("database url: " + url);
        System.out.println("database name: " + name);
        System.out.println("database password: " + password);
        Assert.assertTrue(url.length() > 0);
        Assert.assertTrue(name.length() > 0);
        Assert.assertTrue(password.length() > 0);
    }
}
```

上述代码对数据源的url、user、password进行了明文加密，输出的结果如下：

```

database url:
szkFDG56WcAOzG2utv0m2aoAvNFH5g3DXz0o6.joZjT26Y5WNA+1Z+pQFpyhFBokqOp2jsFtB+P9b3gB601rfas3dSfvS8Bgo3MyP1noj
JgVp6gCVi+B/XUs0keXPn+pbX/19Hr1UN1LeEweHS/LCRZs1hWJCsIXTwZo1P1pXRv3Vyhf20EzzKLm3mIAYj51CrEaN3w5cMiCES1lwv
KUhpAJVz/uXQJ1spLUAMuXCKKrXM/6dSRnWyTtdFRost5cChEU9uRjw5M+8HU3BLemtcK0vM8iYDjEi5zDbZtwxD3hA=


database name: L8I2RqYPptEtQNL4x8VhRVakSUdlsTGzEND/3TOOnVTYPWe0ZnWsW0/5JdUsw9u1m


database password: EJYCSbBL8Pmf2HubIH7dHhpfdZcLyJCEGMR9jAV3apJtvFtx9TVdhUPsAxjQ2pnJ

```

4. 将加密后的密文写入配置

jasypt 默认使用 `ENC()` 包裹，此时的数据源配置如下：

```

spring:
  datasource:
    # 数据源基本配置
    username: ENC(L8I2RqYPptEtQNL4x8VhRVakSUdlsTGzEND/3TOOnVTYPWe0ZnWsW0/5JdUsw9u1m)
    password: ENC(EJYCSbBL8Pmf2HubIH7dHhpfdZcLyJCEGMR9jAV3apJtvFtx9TVdhUPsAxjQ2pnJ)
    driver-class-name: com.mysql.jdbc.Driver
    url:
      ENC(szkFDG56WcAOzG2utv0m2aoAvNFH5g3DXz0o6.joZjT26Y5WNA+1Z+pQFpyhFBokqOp2jsFtB+P9b3gB601rfas3dSfvS8Bgo3MyP
      1nojJgVp6gCVi+B/XUs0keXPn+pbX/19Hr1UN1LeEweHS/LCRZs1hWJCsIXTwZo1P1pXRv3Vyhf20EzzKLm3mIAYj51CrEaN3w5cMiCE
      S1lwvKUhpAJVz/uXQJ1spLUAMuXCKKrXM/6dSRnWyTtdFRost5cChEU9uRjw5M+8HU3BLemtcK0vM8iYDjEi5zDbZtwxD3hA=)
    type: com.alibaba.druid.pool.DruidDataSource

```

上述配置是使用默认的 `prefix=ENC(`、`suffix=)`，当然我们可以根据自己的要求更改，只需要在配置文件中更改即可，如下：

```

jasypt:
  encryptor:
    ## 指定前缀、后缀
    property:
      prefix: 'PASS('
      suffix: ')'

```

那么此时的配置就必须使用 `PASS()` 包裹才会被解密，如下：

```

spring:
  datasource:
    # 数据源基本配置
    username: PASS(L8I2RqYPptEtQNL4x8VhRVakSUdlsTGzEND/3TOOnVTYPWe0ZnWsW0/5JdUsw9u1m)
    password: PASS(EJYCSbBL8Pmf2HubIH7dHhpfdZcLyJCEGMR9jAV3apJtvFtx9TVdhUPsAxjQ2pnJ)
    driver-class-name: com.mysql.jdbc.Driver
    url:
      PASS(szkFDG56WcAOzG2utv0m2aoAvNFH5g3DXz0o6.joZjT26Y5WNA+1Z+pQFpyhFBokqOp2jsFtB+P9b3gB601rfas3dSfvS8Bgo3My
      P1nojJgVp6gCVi+B/XUs0keXPn+pbX/19Hr1UN1LeEweHS/LCRZs1hWJCsIXTwZo1P1pXRv3Vyhf20EzzKLm3mIAYj51CrEaN3w5cMiC
      ES1lwvKUhpAJVz/uXQJ1spLUAMuXCKKrXM/6dSRnWyTtdFRost5cChEU9uRjw5M+8HU3BLemtcK0vM8iYDjEi5zDbZtwxD3hA=)
    type: com.alibaba.druid.pool.DruidDataSource

```

5. 总结

jasypt 有非常多的用法，比如可以自己配置加密算法，具体的操作可以参考Github上的文档。

三、接口返回数据如何脱敏？

通常接口返回值中的一些敏感数据也是要脱敏的，比如身份证号、手机号码、地址.....通常的手段就是用 * 隐藏一部分数据，当然也可以根据自己需求定制。

言归正传，如何优雅的实现呢？有两种实现方案，如下：

- 整合Mybatis插件，在查询的时候针对特定的字段进行脱敏
- 整合Jackson，在序列化阶段对特定字段进行脱敏
- 基于 Sharding Sphere 实现数据脱敏，查看之前的文章：[基于Sharding Sphere实现数据“一键脱敏”](#)

第一种方案网上很多实现方式，下面演示第二种，整合Jackson。

1. 自定义一个Jackson注解

需要自定义一个脱敏注解，一旦有属性被标注，则进行对应得脱敏，如下：

```
/**  
 * 自定义jackson注解，标注在属性上  
 */  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.FIELD)  
@JacksonAnnotationsInside  
@JsonSerialize(using = SensitiveJsonSerializer.class)  
public @interface Sensitive {  
    //脱敏策略  
    SensitiveStrategy strategy();  
}
```

2. 定制脱敏策略

针对项目需求，定制不同字段的脱敏规则，比如手机号中间几位用 * 替代，如下：

```
/**  
 * 脱敏策略，枚举类，针对不同的数据定制特定的策略  
 */  
public enum SensitiveStrategy {  
    /**  
     * 用户名  
     */  
    USERNAME(s -> s.replaceAll("(\\S)\\S(\\S*)", "$1*$2")),  
    /**  
     * 身份证  
     */  
    ID_CARD(s -> s.replaceAll("(\\d{4})\\d{10}(\\w{4})", "$1****$2")),  
    /**  
     * 手机号  
     */  
    PHONE(s -> s.replaceAll("(\\d{3})\\d{4}(\\d{4})", "$1****$2")),  
    /**  
     * 地址  
     */  
    ADDRESS(s -> s.replaceAll("(\\S{3})\\S{2}(\\S*)\\S{2}", "$1****$2****"));
```

```

    private final Function<String, String> desensitizer;

    SensitiveStrategy(Function<String, String> desensitizer) {
        this.desensitizer = desensitizer;
    }

    public Function<String, String> desensitizer() {
        return desensitizer;
    }
}

```

以上只是提供了部分，具体根据自己项目要求进行配置。

3. 定制JSON序列化实现

下面将是重要实现，对标注注解 `@Sensitive` 的字段进行脱敏，实现如下：

```

/**
 * 序列化注解自定义实现
 * JsonSerializer<String>：指定String 类型， serialize()方法用于将修改后的数据载入
 */
public class SensitiveJsonSerializer extends JsonSerializer<String> implements ContextualSerializer {
    private SensitiveStrategy strategy;

    @Override
    public void serialize(String value, JsonGenerator gen, SerializerProvider serializers) throws
IOException {
        gen.writeString(strategy.desensitizer().apply(value));
    }

    /**
     * 获取属性上的注解属性
     */
    @Override
    public JsonSerializer<?> createContextual(SerializerProvider prov, BeanProperty property) throws
JsonMappingException {

        Sensitive annotation = property.getAnnotation(Sensitive.class);
        if (Objects.nonNull(annotation)&&Objects.equals(String.class,
property.getType().getRawClass())) {
            this.strategy = annotation.strategy();
            return this;
        }
        return prov.findValueSerializer(property.getType(), property);

    }
}

```

4. 定义Person类，对其数据脱敏

使用注解 `@Sensitive` 注解进行数据脱敏，代码如下：

```

@Data
public class Person {
    /**
     * 真实姓名
     */

```

```

@Sensitive(strategy = SensitiveStrategy.USERNAME)
private String realName;
/**
 * 地址
 */
@Sensitive(strategy = SensitiveStrategy.ADDRESS)
private String address;
/**
 * 电话号码
 */
@Sensitive(strategy = SensitiveStrategy.PHONE)
private String phoneNumber;
/**
 * 身份证号码
 */
@Sensitive(strategy = SensitiveStrategy.ID_CARD)
private String idCard;
}

```

5. 模拟接口测试

以上4个步骤完成了数据脱敏的Jackson注解，下面写个controller进行测试，代码如下：

```

@RestController
public class TestController {
    @GetMapping("/test")
    public Person test() {
        Person user = new Person();
        user.setRealName("不才陈某");
        user.setPhoneNumber("19796328206");
        user.setAddress("浙江省杭州市温州市....");
        user.setIdCard("433333333334334333");
        return user;
    }
}

```

调用接口查看数据有没有正常脱敏，结果如下：

```
{
    "realName": "不*陈某",
    "address": "浙江省***市温州市..****",
    "phoneNumber": "197****8206",
    "idCard": "4333****34333"
}
```

6. 总结

数据脱敏有很多种实现方式，关键是哪种更加适合，哪种更加优雅.....

Watermark

四、日志文件如何数据脱敏？

上面讲了配置文件、接口返回值的数据脱敏，现在该轮到日志脱敏了。项目中总避免不了打印日志，肯定会涉及到一些敏感数据被明文打印出来，那么此时就需要过滤掉这些敏感数据（身份证、号码、用户名……）。

关于Spring Boot 日志方面的问题有不理解的可以看我之前的文章：[Spring Boot第三弹，一文带你搞懂日志如何配置？](#)、[Spring Boot第二弹，配置文件怎么造？](#)。

下面以log4j2这款日志为例讲解一下日志如何脱敏，其他日志框架大致思路一样。

1. 添加log4j2日志依赖

Spring Boot 默认日志框架是logback，但是我们可以切换到log4j2，依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <!-- 去掉springboot默认配置 -->
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<!--使用log4j2替换 LogBack-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

2. 在/resource目录下新建log4j2.xml配置

log4j2的日志配置很简单，只需要在 `/resource` 文件夹下新建一个 `log4j2.xml` 配置文件，内容如下图：



```
<?xml version="1.0" encoding="UTF-8"?>
<configuration status="DEBUG">
    <!-- 定义日志存放目录 -->
    <properties>
        <property name="logPath">logs</property>
        <property name="PATTERN">%d{yyyy-MM-dd HH:mm:ss.SSS} [%X{traceId}] [%t-%L] %-5level %logger{36}|%L %M - %msg%xEx%n</property>
        <!--%d{yyyy-MM-dd HH:mm:ss} [%thread] %m%n-->
    </properties>
    <!--先定义所有的appender(输出器) -->
    <Appenders>
        <!--输出到控制台 -->
        <Console name="ConsoleLog" target="SYSTEM_OUT">
            <!--只输出level及以上级别的信息 (onMatch)，其他的直接拒绝 (onMismatch) -->
            <ThresholdFilter level="TRACE" onMatch="ACCEPT" onMismatch="DENY" />
            <!--输出日志的格式，引用自定义模板 PATTERN -->
            <PatternLayout pattern="${PATTERN}" />
        </Console>
        <RollingFile name="APPLog" fileName="${logPath}/log_app.log" filePattern="${logPath}/log_app_%d{yyyy-MM-dd}.log">
            <ThresholdFilter level="DEBUG" onMatch="ACCEPT" onMismatch="DENY" />
            <!--<PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss} [%thread] %m%n"/>-->
            <PatternLayout pattern="${PATTERN}" />
            <Policies>
                <TimeBasedTriggeringPolicy modulate="true" interval="1"/>
            </Policies>
            <CronTriggeringPolicy schedule="0 0 5 * * ? "/>
            <DefaultRolloverStrategy>
                <Delete basePath="logs" maxDepth="1" />
                <IfLastModified age="15d" />
            </DefaultRolloverStrategy>
        </RollingFile>
    </Appenders>
</configuration>
```

```

</RollingFile>
<!-- 把error等级记录到文件 一般不用 -->
<File name="ERRORLog" fileName="${logPath}/error.log">
    <ThresholdFilter level="error" onMatch="ACCEPT" onMismatch="DENY" />
    <PatternLayout pattern="${PATTERN}" />
</File>
</Appenders>
<!-- 然后定义Logger, 只有定义了logger并引入的appender, appender才会生效 -->
<Loggers>
    <!-- 建立一个默认的Root的logger, 记录大于Level高于warn的信息, 如果这里的level高于Appenders中的, 则Appenders中也是以此等级为起点, 比如, 这里level="fatal", 则Appenders中只出现fatal信息 -->
    <!-- 生产环境level=>warn -->
    <Root level="debug">
        <!-- 输出器, 可选上面定义的任何项组合, 或全选, 做到可随意定制 -->
        <appender-ref ref="ConsoleLog" />
        <appender-ref ref="ERRORLog" />
        <appender-ref ref="APPLog" />
    </Root>
    <!-- 第三方日志系统 -->
    <!--过滤掉spring和mybatis的一些无用的DEBUG信息, 也可以在spring boot 的logging.level.org.springframework=FATAL设置-->
    <logger name="org.springframework" level="DEBUG"></logger>
    <logger name="java.sql.Connection" level="DEBUG"></logger>
    <logger name="java.sql.Statement" level="DEBUG"></logger>
    <logger name="Java.sql.PreparedStatement" level="DEBUG"></logger>
</Loggers>
</Configuration>

```

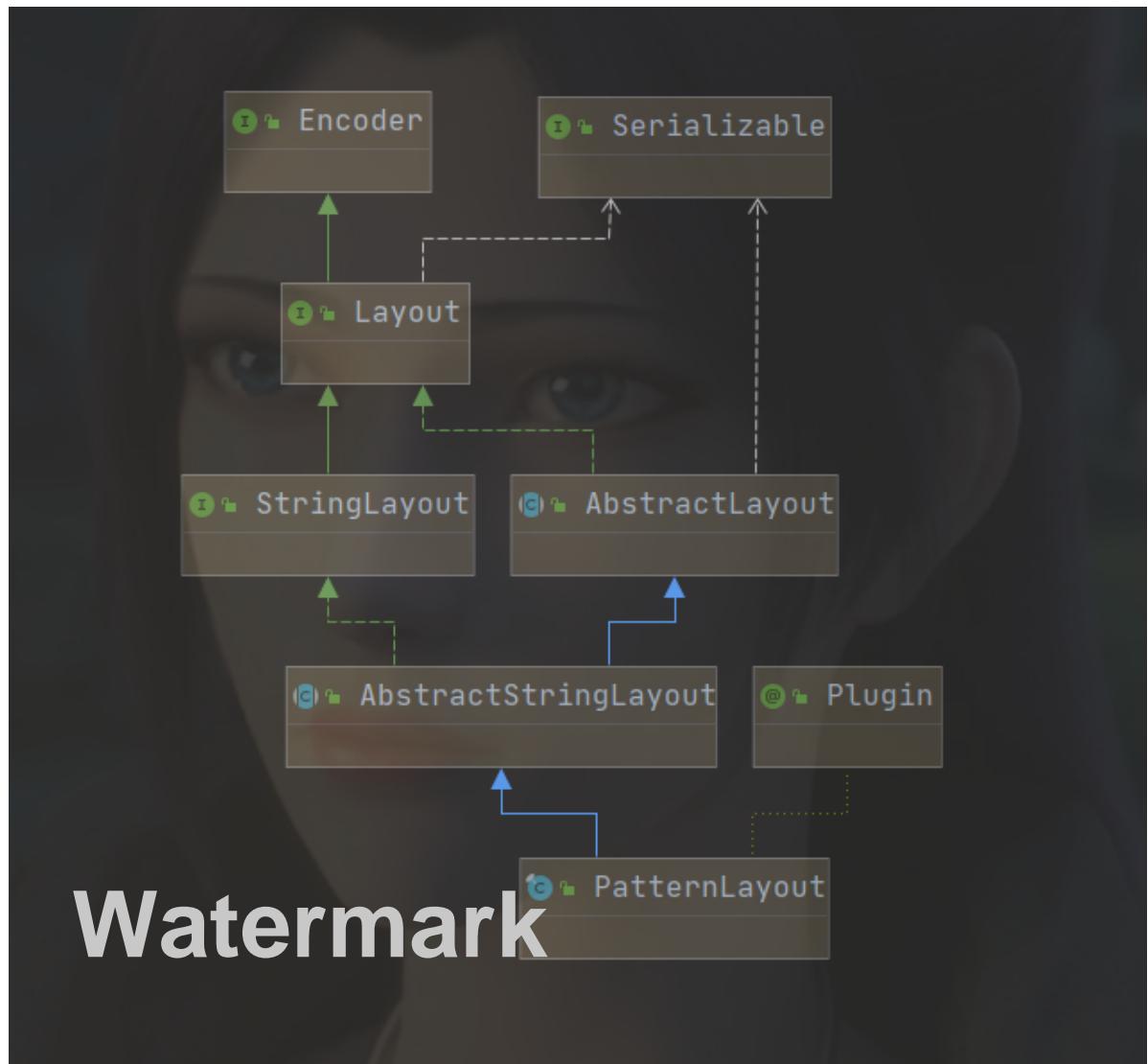
关于每个节点如何配置，含义是什么，在我上面的两篇文章中有详细的介绍。

上图的配置并没有实现数据脱敏，这是普通的配置，使用的是 **PatternLayout**

3. 自定义PatternLayout实现数据脱敏

步骤2中的配置使用的是 **PatternLayout** 实现日志的格式，那么我们也可以自定义一个**PatternLayout**来实现日志的过滤脱敏。

PatternLayout的类图继承关系如下：



从上图中可以清楚的看出来，`PatternLayout`继承了一个抽象类 `AbstractStringLayout`，因此想要自定义只需要继承这个抽象类即可。

1、创建`CustomPatternLayout`, 继承抽象类`AbstractStringLayout`

代码如下：

```
/*
 * log4j2 脱敏插件
 * 继承AbstractStringLayout
 */
@Plugin(name = "CustomPatternLayout", category = Node.CATEGORY, elementType = Layout.ELEMENT_TYPE,
printObject = true)
public class CustomPatternLayout extends AbstractStringLayout {

    public final static Logger logger = LoggerFactory.getLogger(CustomPatternLayout.class);
    private PatternLayout patternLayout;

    protected CustomPatternLayout(Charset charset, String pattern) {
        super(charset);
        patternLayout = PatternLayout.newBuilder().withPattern(pattern).build();
        initRule();
    }

    /**
     * 要匹配的正则表达式map
     */
    private static Map<String, Pattern> REG_PATTERN_MAP = new HashMap<>();
    private static Map<String, String> KEY_REG_MAP = new HashMap<>();

    private void initRule() {
        try {
            if (MapUtils.isEmpty(Log4j2Rule.regularMap)) {
                return;
            }
            Log4j2Rule.regularMap.forEach((a, b) -> {
                if (StringUtils.isNotBlank(a)) {
                    Map<String, String> collect =
                        Arrays.stream(a.split(",")).collect(Collectors.toMap(c -> c, w -> b, (key1, key2) -> key1));
                    KEY_REG_MAP.putAll(collect);
                }
                Pattern compile = Pattern.compile(b);
                REG_PATTERN_MAP.put(b, compile);
            });
        } catch (Exception e) {
            logger.info(">>>> 初始化日志脱敏规则失败 ERROR: {}", e);
        }
    }
}
```

Watermark

```
/*
 * 处理日志信息，进行脱敏
 * 1. 判断配置文件中是否已经配置需要脱敏字段
 * 2. 判断内容是否有需要脱敏的敏感信息
*/
```

```

    * 2.1 没有需要脱敏信息直接返回
    * 2.2 处理：身份证号，姓名，手机号敏感信息
    */
public String hideMarkLog(String logStr) {
    try {
        //1. 判断配置文件中是否已经配置需要脱敏字段
        if (StringUtils.isBlank(logStr) || MapUtils.isEmpty(KEY_REG_MAP) ||
MapUtils.isEmpty(REG_PATTERN_MAP)) {
            return logStr;
        }
        //2. 判断内容是否有需要脱敏的敏感信息
        Set<String> charKeys = KEY_REG_MAP.keySet();
        for (String key : charKeys) {
            if (logStr.contains(key)) {
                String regExp = KEY_REG_MAP.get(key);
                logStr = matchingAndEncrypt(logStr, regExp, key);
            }
        }
        return logStr;
    } catch (Exception e) {
        logger.info(">>>>>> 脱敏处理异常 ERROR: {}", e);
        //如果抛出异常为了不影响流程，直接返回原信息
        return logStr;
    }
}

/**
 * 正则匹配对应的对象。
 *
 * @param msg
 * @param regExp
 * @return
 */
private static String matchingAndEncrypt(String msg, String regExp, String key) {
    Pattern pattern = REG_PATTERN_MAP.get(regExp);
    if (pattern == null) {
        logger.info(">>> logger 没有匹配到对应的正则表达式 ");
        return msg;
    }
    Matcher matcher = pattern.matcher(msg);
    int length = key.length() + 5;
    boolean contains = Log4j2Rule.USER_NAME_STR.contains(key);
    String hiddenStr = "";
    while (matcher.find()) {
        String originStr = matcher.group();
        if (contains) {
            // 计算关键词和需要脱敏词的距离小于5。
            int i = msg.indexOf(originStr);
            if (i < 0) {
                continue;
            }
            int end = i + length;
            int start = i - span > 0 ? span : 0;
            String substring = msg.substring(startIndex, i);
            if (StringUtils.isBlank(substring) || !substring.contains(key)) {
                continue;
            }
            hiddenStr = hideMarkStr(originStr);
        }
    }
}

```

```

        msg = msg.replace(originStr, hiddenStr);
    } else {
        hiddenStr = hideMarkStr(originStr);
        msg = msg.replace(originStr, hiddenStr);
    }

}

return msg;
}

/**
 * 标记敏感文字规则
 *
 * @param needHideMark
 * @return
 */
private static String hideMarkStr(String needHideMark) {
    if (StringUtils.isBlank(needHideMark)) {
        return "";
    }
    int startSize = 0, endSize = 0, mark = 0, length = needHideMark.length();

    StringBuffer hideRegBuffer = new StringBuffer("(\\S+");
    StringBuffer replaceSb = new StringBuffer("$1");

    if (length > 4) {
        int i = length / 3;
        startSize = i;
        endSize = i;
    } else {
        startSize = 1;
        endSize = 0;
    }

    mark = length - startSize - endSize;
    for (int i = 0; i < mark; i++) {
        replaceSb.append("*");
    }
    hideRegBuffer.append(startSize).append(")\\$*(\\S+").append(endSize).append(")");
    replaceSb.append("$2");
    needHideMark = needHideMark.replaceAll(hideRegBuffer.toString(), replaceSb.toString());
    return needHideMark;
}

/**
 * 创建插件
 */
@PluginFactory
public static Layout createLayout(@PluginAttribute(value = "pattern") final String pattern,
                                  @PluginAttribute(value = "charset") final
Character charset) {
    return new CustomPatternLayout(charset, pattern);
}

@Override
public String toSerializable(LogEvent event) {
}

```

```
        return hideMarkLog(patternLayout.toSerializable(event));
    }

}
```

关于其中的一些细节，比如 `@Plugin`、`@PluginFactory` 这两个注解什么意思？`log4j2`如何实现自定义一个插件，这里不再详细介绍，不是本文重点，有兴趣的可以查看 `log4j2` 的官方文档。

2、自定义自己的脱敏规则

上述代码中的 `Log4j2Rule` 则是脱敏规则静态类，我这里是直接放在了静态类中配置，实际项目中可以设置到配置文件中，代码如下：

```
/**
 * 现在拦截加密的日志有三类：
 * 1, 身份证
 * 2, 姓名
 * 3, 身份证号
 * 加密的规则后续可以优化在配置文件中
 **/


public class Log4j2Rule {

    /**
     * 正则匹配 关键词 类别
     */
    public static Map<String, String> regularMap = new HashMap<>();

    /**
     * TODO 可配置
     * 此项可以后期放在配置项中
     */
    public static final String USER_NAME_STR = "Name, name, 联系人, 姓名";
    public static final String USER_IDCARD_STR = "empCard, idCard, 身份证, 证件号";
    public static final String USER_PHONE_STR = "mobile, Phone, phone, 电话, 手机";

    /**
     * 正则匹配，自己根据业务要求自定义
     */
    private static String IDCARD_REGEX = "(\\d{17} [0-9Xx] | \\d{14} [0-9Xx])";
    private static String USERNAME_REGEX = "[\\u4e00-\\u9fa5]{2,4}";
    private static String PHONE_REGEX = "(?<!\\d) (?:(?:1[3456789]\\d{9}) | (?:861[356789]\\d{9})) (?!\d)";

    static {
        regularMap.put(USER_NAME_STR, USERNAME_REGEX);
        regularMap.put(USER_IDCARD_STR, IDCARD_REGEX);
        regularMap.put(USER_PHONE_STR, PHONE_REGEX);
    }

}
```

经过上述两个步骤，自定义的 `PatternLayout` 已经完成，下面将是改写 `log4j2.xml` 这个配置文件了。

Watermark

4. 修改log4j2.xml配置文件

其实这里修改很简单，原配置文件是直接使用 `PatternLayout` 进行日志格式化的，那么只需要将默认的 `<PatternLayout/>` 这个节点替换成 `<CustomPatternLayout/>`，如下图：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="DEBUG">
    <!-- 定义日志存放目录 -->
    <properties>
        <property name="logPath">logs</property>
        <property name="PATTERN">%d{yyyy-MM-dd HH:mm:ss.SSS} [%X{traceId}] [%t-%L] %-5level %logger{36} %L %M - %msg%xEx%n</property>
        <!-- "%d{yyyy-MM-dd HH:mm:ss} [%thread] %m%n"-->
    </properties>
    <!--先定义所有的appender(输出器) -->
    <Appenders>
        <!--输出到控制台 -->
        <Console name="ConsoleLog" target="SYSTEM_OUT">
            <!--只输出level及以上级别的信息 (onMatch) , 其他的直接拒绝 (onMismatch) -->
            <ThresholdFilter level="TRACE" onMatch="ACCEPT" onMismatch="DENY" />
            <!--输出日志的格式, 引用自定义模板 PATTERN -->
            <CustomPatternLayout pattern="${PATTERN}" />
        </Console>
        <RollingFile name="APPLog" fileName="${logPath}/log_app.log" filePattern="${logPath}/log_app_%d{yyyy-MM-dd}.log">
            <ThresholdFilter level="DEBUG" onMatch="ACCEPT" onMismatch="DENY" />
            <!--<CustomPatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss} [%thread] %m%n"/>-->
            <CustomPatternLayout pattern="${PATTERN}" />
        </RollingFile>
        <Policies>
            <TimeBasedTriggeringPolicy modulate="true" interval="1"/>
        </Policies>
        <CronTriggeringPolicy schedule="0 0 5 * * ? "/>
        <DefaultRolloverStrategy>
            <Delete basePath="${logPath}" maxDepth="1">
                <IfFileName glob="Log_app.*.log" />
                <!--删除15天前的文件-->
                <IfLastModified age="15d" />
            </Delete>
        </DefaultRolloverStrategy>
    </Appenders>
</Configuration>
```

直接全局替换掉即可，至此，这个配置文件就修改完成了。

5. 演示效果

在步骤3这边自定义了脱敏规则静态类 `Log4j2Rule`，其中定义了姓名、身份证、号码这三个脱敏规则，如下：

```
public class Log4j2Rule {
    /**
     * 正则匹配 关键词 类别
     */
    public static Map<String, String> regularMap = new HashMap<>();
    /**
     * TODO 可配置
     * 此项可以后期放在配置项中
     */
    public static final String USER_NAME_STR = "Name,name,联系人,姓名";
    public static final String USER_IDCARD_STR = "empCard,idCard,身份证,证件号";
    public static final String USER_PHONE_STR = "mobile,Phone,phone,电话,手机";

    /**
     * 正则匹配, 自己根据业务要求自定义
     */
    private static String IDCARD_REGEX = "(\\d{17}[0-9Xx]|\\d{14}[0-9Xx])";
    private static String USERNAME_REGEX = "[\\u4e00-\\u9fa5]{2,4}";
    private static String PHONE_REGEX = "(?(<!\\d)(?:(:1[3456789]\\d{9})|(:861[356789]\\d{9}))(!\\d)";

    static {
        regularMap.put(USER_NAME_STR, USERNAME_REGEX);
        regularMap.put(USER_IDCARD_STR, IDCARD_REGEX);
        regularMap.put(USER_PHONE_STR, PHONE_REGEX);
    }
}
```

下面就来演示这三个规则能否正确脱敏，直接使用日志打印，代码如下：

```
@Test  
public void test3() {  
    log.debug("身份证: {}, 姓名: {}, 电话: {}", "320829112334566767", "不才陈某", "19896327106");  
}
```

控制台打印的日志如下：

```
身份证: 320829*****566767, 姓名: 不***, 电话: 198*****106
```

哦豁，成功了，so easy! ! !

6. 总结

日志脱敏的方案很多，陈某也只是介绍一种常用的，有兴趣的可以研究一下。

五、总结

本篇文章从三个维度介绍了隐私数据的脱敏实现方案，码字不易，赶紧点赞收藏吧！！！

源码已经上传GitHub，需要的公众号码猿技术专栏，回复关键词 **数据脱敏** 获取。

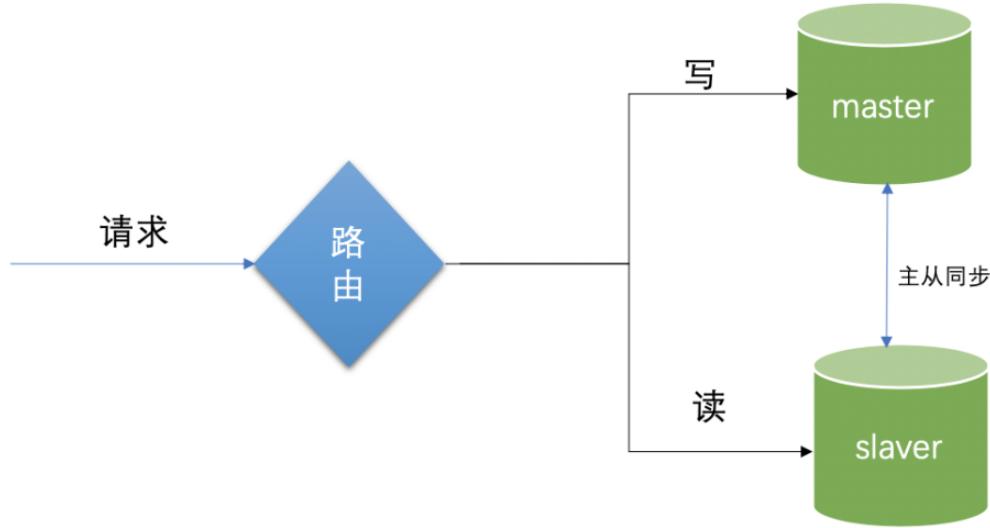
SpringBoot，来实现MySQL读写分离技术

前言

首先思考一个问题：在高并发的场景中，关于数据库都有哪些优化的手段？常用的有以下的实现方法：读写分离、加缓存、主从架构集群、分库分表等，在互联网应用中，大部分都是**读多写少**的场景，设置两个库，主库和读库。

主库的职能是负责写，从库主要是负责读，可以建立读库集群，通过读写职能在数据源上的隔离达到减少读写冲突、释压数据库负载、保护数据库的目的。在实际的使用中，凡是涉及到写的部分直接切换到主库，读的部分直接切换到读库，这就是典型的读写分离技术。本篇博文将聚焦读写分离，探讨如何实现它。

Watermark



主从同步的局限性：这里分为主数据库和从数据库,主数据库和从数据库保持数据库结构的一致,主库负责写,当写入数据的时候,会自动同步数据到从数据库;从数据库负责读,当读请求来的时候,直接从读库读取数据,主数据库会自动进行数据复制到从数据库中。不过本篇博客不介绍这部分配置的知识,因为它更偏运维工作一点。

这里涉及到一个问题:主从复制的延迟问题,当写入到主数据库的过程中,突然来了一个读请求,而此时数据还没有完全同步,就会出现读请求的数据读不到或者读出的数据比原始值少的情况。具体的解决方法最简单的就是将读请求暂时指向主库,但是同时也失去了主从分离的部分意义。也就是说在严格意义上的数据一致性场景中,读写分离并非是完全适合的,注意更新的时效性是读写分离使用的缺点。

好了,这部分只是了解,接下来我们看下具体如何通过 java 代码来实现读写分离:

该项目需要引入如下依赖: springBoot、spring-aop、spring-jdbc、aspectjweaver 等

一: 主从数据源的配置

我们需要配置主从数据库,主从数据库的配置一般都是写在配置文件里面。通过 @ConfigurationProperties 注解,可以将配置文件(一般命名为:application.Properties)里的属性映射到具体的类属性上,从而读取到写入的值注入到具体的代码配置中,按照习惯大于约定的原则,主库我们都是注为 master,从库注为 slave。

本项目采用了阿里的 druid 数据库连接池,使用 build 建造者模式创建 DataSource 对象,DataSource 就是代码层面抽象出来的数据源,接着需要配置 sessionFactory、sqlTemplate、事务管理器等

```

/**
 * 主从配置
 *
 * @author wyq
 */
@Configuration
@MapperScan(basePackages = "com.wyq.mysqlreadwriteseperate.mapper", sqlSessionTemplateRef =
"sqlSession")
public class DataSourceConfig {

    /**
     * 主库
     */
}

```

```

@Bean
@ConfigurationProperties(prefix = "spring.datasource.master")
public DataSource master() {
    return DruidDataSourceBuilder.create().build();
}

/**
 * 从库
 */
@Bean
@ConfigurationProperties(prefix = "spring.datasource.slave")
public DataSource slaver() {
    return DruidDataSourceBuilder.create().build();
}

/**
 * 实例化数据源路由
 */
@Bean
public DataSourceRouter dynamicDB(@Qualifier("master") DataSource masterDataSource,
                                  @Autowired(required = false)
@Qualifier("slaver") DataSource slaveDataSource) {
    DataSourceRouter dynamicDataSource = new DataSourceRouter();
    Map<Object, Object> targetDataSources = new HashMap<>();
    targetDataSources.put(DataSourceEnum.MASTER.getDataSourceName(), masterDataSource);
    if (slaveDataSource != null) {
        targetDataSources.put(DataSourceEnum.SLAVE.getDataSourceName(), slaveDataSource);
    }
    dynamicDataSource.setTargetDataSources(targetDataSources);
    dynamicDataSource.setDefaultTargetDataSource(masterDataSource);
    return dynamicDataSource;
}

/**
 * 配置sessionFactory
 * @param dynamicDataSource
 * @return
 * @throws Exception
 */
@Bean
public SqlSessionFactory sessionFactory(@Qualifier("dynamicDB") DataSource dynamicDataSource)
throws Exception {
    SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
    bean.setMapperLocations(
        new PathMatchingResourcePatternResolver().getResources("classpath*:mapper/*Mapper.xml"));
    bean.setDataSource(dynamicDataSource);
    return bean.getObject();
}

/**
 * 创建sqlTemplate
 * @param sqlSessionFactory
 * @return
 */

```

Watermark

```

    @Bean
    public SqlSessionTemplate sqlTemplate(@Qualifier("sessionFactory") SqlSessionFactory
sqlSessionFactory) {
        return new SqlSessionTemplate(sqlSessionFactory);
    }

    /**
     * 事务配置
     *
     * @param dynamicDataSource
     * @return
     */
    @Bean(name = "dataSourceTx")
    public DataSourceTransactionManager dataSourceTransactionManager(@Qualifier("dynamicDB")
DataSource dynamicDataSource) {
        DataSourceTransactionManager dataSourceTransactionManager = new
DataSourceTransactionManager();
        dataSourceTransactionManager.setDataSource(dynamicDataSource);
        return dataSourceTransactionManager;
    }
}

```

二：数据源路由的配置

路由在主从分离是非常重要的,基本是读写切换的核心。Spring 提供了 AbstractRoutingDataSource 根据用户定义的规则选择当前的数据源,作用就是在执行查询之前,设置使用的数据源,实现动态路由的数据源,在每次数据库查询操作前执行它的抽象方法 determineCurrentLookupKey() 决定使用哪个数据源。

为了能有一个全局的数据源管理器,此时我们需要引入 DataSourceContextHolder 这个数据库上下文管理器,可以理解为全局的变量,随时可取(见下面详细介绍),它的主要作用就是保存当前的数据源;

```

public class DataSourceRouter extends AbstractRoutingDataSource {

    /**
     * 最终的determineCurrentLookupKey返回的是从DataSourceContextHolder中拿到的,因此在动态切换数据源的时候注解
     * 应该给DataSourceContextHolder设值
     *
     * @return
     */
    @Override
    protected Object determineCurrentLookupKey() {
        return DataSourceContextHolder.get();
    }
}

```

三：数据源上下文环境

数据源上下文保存器,便于程序中可以随时取到当前的数据源,它主要利用 ThreadLocal 封装,因为 ThreadLocal 是线程隔离的,天然具有线程安全的优势。这里暴露了 set 和 get、 clear 方法, set 方法用于赋值当前的数据源名,get 方法用于获取当前的数据源名称,clear 方法用于清除 ThreadLocal 中的内容,因为 ThreadLocal 的 key 是 weakReference 是有内存泄漏风险的,通过 remove 方法防止内存泄漏;

```
/*
 * 利用ThreadLocal封装的保存数据源上线的上下文context
 */
public class DataSourceContextHolder {

    private static final ThreadLocal<String> context = new ThreadLocal<>();

    /**
     * 赋值
     *
     * @param datasourceType
     */
    public static void set(String datasourceType) {
        context.set(datasourceType);
    }

    /**
     * 获取值
     * @return
     */
    public static String get() {
        return context.get();
    }

    public static void clear() {
        context.remove();
    }
}
```

四：切换注解和 Aop 配置

首先我们来定义一个@DataSourceSwitcher 注解,拥有两个属性 ① 当前的数据源 ② 是否清除当前的数据源,并且只能放在方法上,(不可以放在类上,也没必要放在类上,因为我们在进行数据源切换的时候肯定是方法操作),该注解的主要作用就是进行数据源的切换,在 dao 层进行操作数据库的时候,可以在方法上注明表示的是当前使用哪个数据源;

@DataSourceSwitcher 注解的定义:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Documented
public @interface DataSourceSwitcher {
    /**
     * 默认数据源
     * @return
     */
    DataSourceEnum value() default DataSourceEnum.MASTER;
    /**
     * 清除
     */
```

```

    * @return
    */
    boolean clear() default true;

}

```

DataSourceAop 配置

为了赋予@DataSourceSwitcher 注解能够切换数据源的能力,我们需要使用 AOP,然后使用@Aroud 注解找到方法上有@DataSourceSwitcher.class 的方法,然后取注解上配置的数据源的值,设置到 DataSourceContextHolder 中,就实现了将当前方法上配置的数据源注入到全局作用域当中;

```

@Slf4j
@Aspect
@Order(value = 1)
@Component
public class DataSourceContextAop {

    @Around("@annotation(com.wyq.mysqlreadwriteseparete.annotation.DataSourceSwitcher)")
    public Object setDynamicDataSource(ProceedingJoinPoint pjp) throws Throwable {
        boolean clear = false;
        try {
            Method method = this.getMethod(pjp);
            DataSourceSwitcher dataSourceSwitcher =
                    method.getAnnotation(DataSourceSwitcher.class);
            clear = dataSourceSwitcher.clear();
            DataSourceContextHolder.set(dataSourceSwitcher.value().getDataSourceName());
            log.info("数据源切换至: {}", dataSourceSwitcher.value().getDataSourceName());
            return pjp.proceed();
        } finally {
            if (clear) {
                DataSourceContextHolder.clear();
            }
        }
    }

    private Method getMethod(JoinPoint pjp) {
        MethodSignature signature = (MethodSignature) pjp.getSignature();
        return signature.getMethod();
    }
}

```

五：用法以及测试

在配置好了读写分离之后,就可以在代码中使用了,一般而言我们使用在 service 层或者 dao 层,在需要查询的方法上添加@DataSourceSwitcher(DataSourceEnum.SLAVE),它表示该方法下所有的操作都走的是读库;在需要 update 或者 insert 的时候使用@DataSourceSwitcher(DataSourceEnum.MASTER)表示接下
Watermark

其实还有一种更为自动的写法,可以根据方法的前缀来配置 AOP 自动切换数据源,比如 update、insert、fresh 等前缀的方法名一律自动设置为写库,select、get、query 等前缀的方法名一律配置为读库,这是一种更为自动的配置写法。缺点就是方法名需要按照 aop 配置的严格来定义,否则就会失效

```

@Service
public class OrderService {

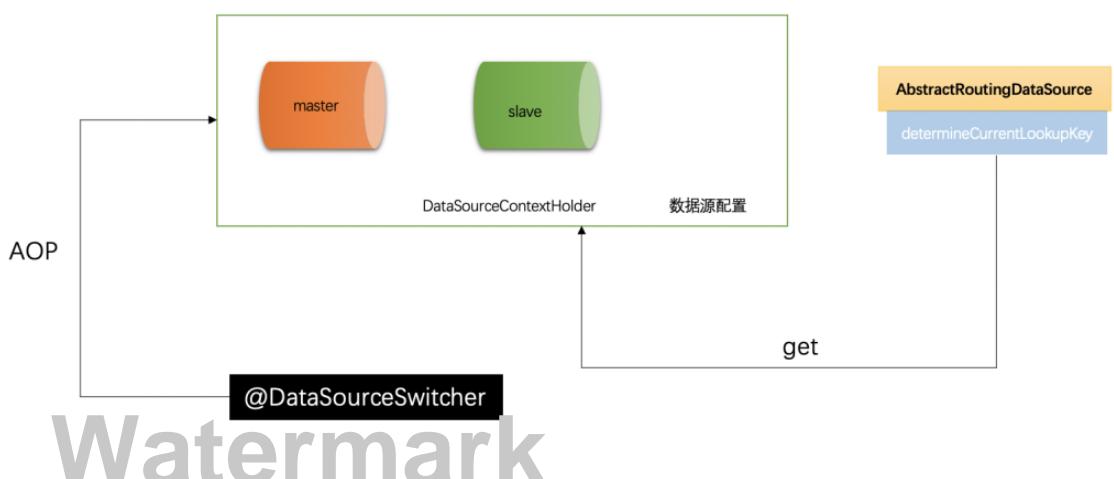
    @Resource
    private OrderMapper orderMapper;

    /**
     * 读操作
     *
     * @param orderId
     * @return
     */
    @DataSourceSwitcher(dataSourceEnum.SLAVE)
    public List<Order> getOrder(String orderId) {
        return orderMapper.listOrders(orderId);
    }

    /**
     * 写操作
     *
     * @param orderId
     * @return
     */
    @DataSourceSwitcher(dataSourceEnum.MASTER)
    public List<Order> insertOrder(Long orderId) {
        Order order = new Order();
        order.setOrderId(orderId);
        return orderMapper.saveOrder(order);
    }
}

```

六：总结



上面是基本流程简图,本篇博客介绍了如何实现数据库读写分离,注意读写分离的核心点就是数据路由,需要继承 `AbstractRoutingDataSource`,复写它的 `determineCurrentLookupKey` 方法,同时需要注意全局的上下文管理器 `DataSourceContextHolder`,它是保存数据源上下文的主要类,也是路由方法中寻找的数据源取值,相当于数据源的中转站.再结合 `jdbc-Template` 的底层去创建和管理数据源、事务等, 我们

的数据库读写分离就完美实现了。

SpringBoot中使用注解来实现 Redis 分布式锁

一、业务背景

有些业务请求，属于耗时操作，需要加锁，防止后续的并发操作，同时对数据库的数据进行操作，需要避免对之前的业务造成影响。

二、分析流程

使用 Redis 作为分布式锁，将锁的状态放到 Redis 统一维护，解决集群中单机 JVM 信息不互通的问题，规定操作顺序，保护用户的数据正确。

梳理设计流程

1. 新建注解 @interface，在注解里设定入参标志
2. 增加 AOP 切点，扫描特定注解
3. 建立 @Aspect 切面任务，注册 bean 和拦截特定方法
4. 特定方法参数 ProceedingJoinPoint，对方法 pjp.proceed() 前后进行拦截
5. 切点前进行加锁，任务执行后进行删除 key

核心步骤：加锁、解锁和续时

加锁

使用了 RedisTemplate 的 opsForValue.setIfAbsent 方法，判断是否有 key，设定一个随机数 UUID.randomUUID().toString，生成一个随机数作为 value。

从 redis 中获取锁之后，对 key 设定 expire 失效时间，到期后自动释放锁。

按照这种设计，只有第一个成功设定 Key 的请求，才能进行后续的数据操作，后续其它请求由于无法获得锁资源，将会失败结束。

超时问题

担心 pjp.proceed() 切点执行的方法太耗时，导致 Redis 中的 key 由于超时提前释放了。

例如，线程 A 先去执行，调用 a 方法耗时超过了锁超时时间，到期释放了锁，这时另一个线程 B 成功获取 Redis 锁，两个线程同时对同一批数据进行操作，导致数据不准确。

解决方案：增加一个「续时」

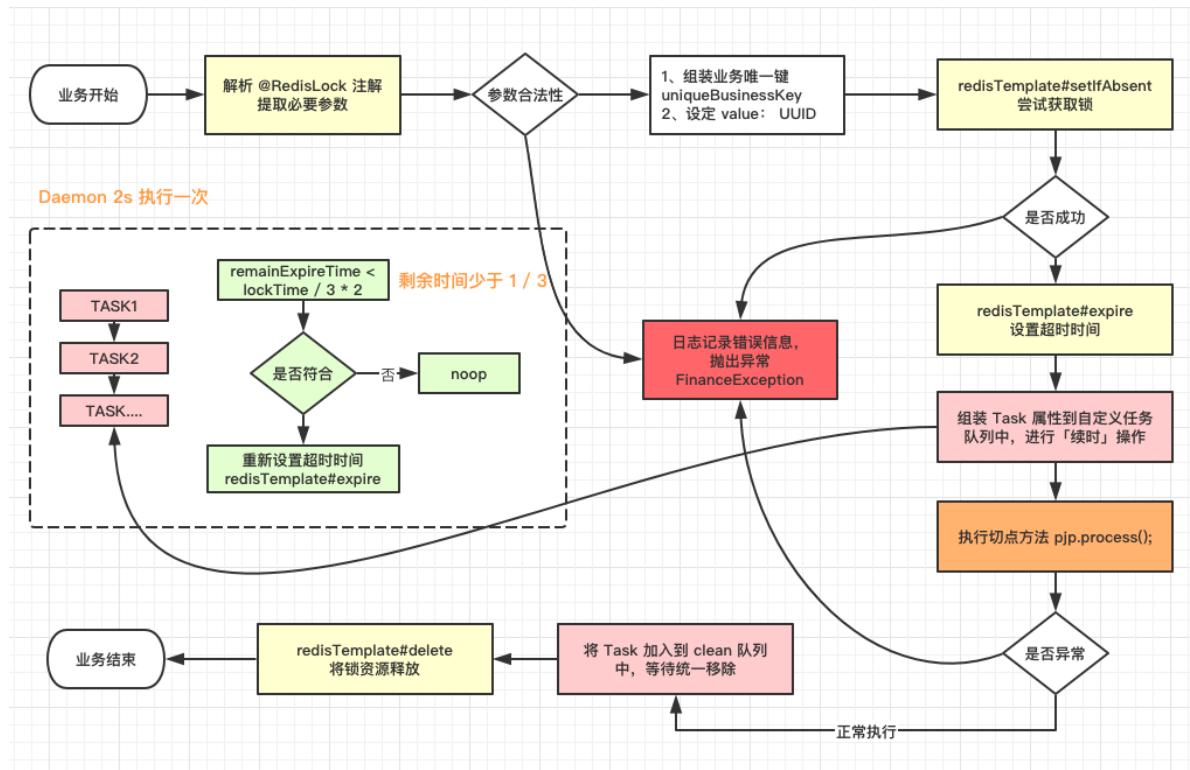
任务不完成，锁不释放：

维护了一个定时线程池 `ScheduledExecutorService`，每隔 2s 去扫描加入队列中的 Task，判断是否失效时间是否快到了，公式为：【失效时间】<= 【当前时间】+ 【失效间隔（三分之一超时）】

```
/**  
 * 线程池，每个 JVM 使用一个线程去维护 keyAliveTime，定时执行 runnable  
 */  
  
private static final ScheduledExecutorService SCHEDULER =  
    new ScheduledThreadPoolExecutor(1,  
        new BasicThreadFactory.Builder().namingPattern("redisLock-schedule-pool").daemon(true).build());  
  
static {  
    SCHEDULER.scheduleAtFixedRate(() -> {  
        // do something to extend time  
    }, 0, 2, TimeUnit.SECONDS);  
}
```

三、设计方案

经过上面的分析，同事小赵设计出了这个方案：



前面已经说了整体流程，这里强调一下几个核心步骤：

- 拦截注解 `@RedisLock`, 获取必要的参数
- 加锁操作
- 续时操作
- 结束业务, 释放锁

四、实操

之前也有整理过 AOP 使用方法，可以参考一下

相关属性类配置

业务属性枚举设定

```
public enum RedisLockTypeEnum {
    /**
     * 自定义 key 前缀
     */
    ONE("Business1", "Test1"),

    TWO("Business2", "Test2");
    private String code;
    private String desc;
    RedisLockTypeEnum(String code, String desc) {
        this.code = code;
        this.desc = desc;
    }
    public String getCode() {
        return code;
    }
    public String getDesc() {
        return desc;
    }
    public String getUniqueKey(String key) {
        return String.format("%s:%s", this.getCode(), key);
    }
}
```

任务队列保存参数

```
public class RedisLockDefinitionHolder {
    /**
     * 业务唯一 key
     */
    private String businessKey;
    /**
     * 加锁时间 (秒 s)
     */
    private Long lockTime;
    /**
     * 上次更新时间 (ms)
     */
    private Long lastModifyTime;
    /**
     * 保存当前线程
     */
    private Thread currentThread;
    /**
     * 总共尝试次数
     */
    private int tryCount;
}
```

```

    * 当前尝试次数
    */
    private int currentCount;
    /**
     * 更新的时间周期（毫秒），公式 = 加锁时间（转成毫秒） / 3
     */
    private Long modifyPeriod;
    public RedisLockDefinitionHolder(String businessKey, Long lockTime, Long lastModifyTime, Thread
currentTread, int tryCount) {
        this.businessKey = businessKey;
        this.lockTime = lockTime;
        this.lastModifyTime = lastModifyTime;
        this.currentTread = currentTread;
        this.tryCount = tryCount;
        this.modifyPeriod = lockTime * 1000 / 3;
    }
}

```

设定被拦截的注解名字

```

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface RedisLockAnnotation {
    /**
     * 特定参数识别，默认取第 0 个下标
     */
    int lockFiled() default 0;
    /**
     * 超时重试次数
     */
    int tryCount() default 3;
    /**
     * 自定义加锁类型
     */
    RedisLockTypeEnum typeEnum();
    /**
     * 释放时间，秒 s 单位
     */
    long lockTime() default 30;
}

```

核心切面拦截的操作

`RedisLockAspect.java` 该类分成三部分来描述具体作用

Pointcut 设定

```

/**
 * @annotation 中的路径表示拦截特定注解
 */
@Pointcut("@annotation(cn.sevenyuan.demo.aop.lock.RedisLockAnnotation)")
public void redisLockPC() {
}

```

Around 前后进行加锁和释放锁

前面步骤定义了我们想要拦截的切点，下一步就是在切点前后做一些自定义操作：

```
@Around(value = "redisLockPC()")
public Object around(ProceedingJoinPoint pjp) throws Throwable {
    // 解析参数
    Method method = resolveMethod(pjp);
    RedisLockAnnotation annotation = method.getAnnotation(RedisLockAnnotation.class);
    RedisLockTypeEnum typeEnum = annotation.typeEnum();
    Object[] params = pjp.getArgs();
    String ukString = params[annotation.lockFiled()].toString();
    // 省略很多参数校验和判空
    String businessKey = typeEnum.getUniqueKey(ukString);
    String uniqueValue = UUID.randomUUID().toString();
    // 加锁
    Object result = null;
    try {
        boolean isSuccess = redisTemplate.opsForValue().setIfAbsent(businessKey, uniqueValue);
        if (!isSuccess) {
            throw new Exception("You can't do it, because another has get the lock ===");
        }
        redisTemplate.expire(businessKey, annotation.lockTime(), TimeUnit.SECONDS);
        Thread currentThread = Thread.currentThread();
        // 将本次 Task 信息加入「延时」队列中
        holderList.add(new RedisLockDefinitionHolder(businessKey, annotation.lockTime(),
            System.currentTimeMillis(),
            currentThread, annotation.tryCount()));
        // 执行业务操作
        result = pjp.proceed();
        // 线程被中断，抛出异常，中断此次请求
        if (currentThread.isInterrupted()) {
            throw new InterruptedException("You had been interrupted ===");
        }
    } catch (InterruptedException e) {
        log.error("Interrupt exception, rollback transaction", e);
        throw new Exception("Interrupt exception, please send request again");
    } catch (Exception e) {
        log.error("has some error, please check again", e);
    } finally {
        // 请求结束后，强制删掉 key，释放锁
        redisTemplate.delete(businessKey);
        log.info("release the lock, businessKey is [" + businessKey + "]");
    }
    return result;
}
```

上述流程简单总结一下：

- **解析注解参数，获取注解值和方法上的参数值**
- **redis 加锁并且设置超时时间**
- **将本次 Task 信息加入「延时」队列中，进行续时，方式提前释放锁**
- **加了一个步是中止点**
- **结束请求，finally 中释放锁**

续时操作

这里用了 `ScheduledExecutorService`，维护了一个线程，不断对任务队列中的任务进行判断和延长超时时间：

```
// 扫描的任务队列
private static ConcurrentLinkedQueue<RedisLockDefinitionHolder> holderList = new
ConcurrentLinkedQueue();
/**
 * 线程池，维护keyAliveTime
 */
private static final ScheduledExecutorService SCHEDULER = new ScheduledThreadPoolExecutor(1,
        new BasicThreadFactory.Builder().namingPattern("redisLock-schedule-
pool").daemon(true).build());
{
    // 两秒执行一次「续时」操作
    SCHEDULER.scheduleAtFixedRate(() -> {
        // 这里记得加 try-catch，否者报错后定时任务将不会再执行==
        Iterator<RedisLockDefinitionHolder> iterator = holderList.iterator();
        while (iterator.hasNext()) {
            RedisLockDefinitionHolder holder = iterator.next();
            // 判空
            if (holder == null) {
                iterator.remove();
                continue;
            }
            // 判断 key 是否还有效，无效的话进行移除
            if (redisTemplate.opsForValue().get(holder.getBusinessKey()) == null) {
                iterator.remove();
                continue;
            }
            // 超时重试次数，超过时给线程设定中断
            if (holder.getCurrentCount() > holder.getTryCount()) {
                holder.getCurrentThread().interrupt();
                iterator.remove();
                continue;
            }
            // 判断是否进入最后三分之一时间
            long curTime = System.currentTimeMillis();
            boolean shouldExtend = (holder.getLastModifyTime() + holder.getModifyPeriod()) <=
curTime;
            if (shouldExtend) {
                holder.setLastModifyTime(curTime);
                redisTemplate.expire(holder.getBusinessKey(), holder.getLockTime(),
TimeUnit.SECONDS);
                log.info("businessKey : [" + holder.getBusinessKey() + "], try count : " +
holder.getCurrentCount());
                holder.setCurrentCount(holder.getCurrentCount() + 1);
            }
        }
    }, 0, 2, TimeUnit.SECONDS);
}
```

这段代码，用来实现设计图中虚线框的思想，避免一个请求十分耗时，导致提前释放了锁。

这里加了「线程中断」`Thread#interrupt`，希望超过重试次数后，能让线程中断（未经严谨测试，仅供参考哈哈哈哈）

不过建议如果遇到这么耗时的请求，还是能够从根源上查找，分析耗时路径，进行业务优化或其它处理，避免这些耗时操作。

所以记得多打点 `Log`，分析问题时可以更快一点。如何使用SpringBoot AOP 记录操作日志、异常日志？

五、开始测试

在一个入口方法中，使用该注解，然后在业务中模拟耗时请求，使用了 `Thread#sleep`

```
@GetMapping("/testRedisLock")
@RedisLockAnnotation(typeEnum = RedisLockTypeEnum.ONE, lockTime = 3)
public Book testRedisLock(@RequestParam("userId") Long userId) {
    try {
        log.info("睡眠执行前");
        Thread.sleep(10000);
        log.info("睡眠执行后");
    } catch (Exception e) {
        // log error
        log.info("has some error", e);
    }
    return null;
}
```

使用时，在方法上添加该注解，然后设定相应参数即可，根据 `typeEnum` 可以区分多种业务，限制该业务被同时操作。

测试结果：

```
2020-04-04 14:55:50.864 INFO 9326 --- [nio-8081-exec-1] c.s.demo.controller.BookController : 睡眠执行前
2020-04-04 14:55:52.855 INFO 9326 --- [k-schedule-pool] c.s.demo.aop.lock.RedisLockAspect : businessKey : [Business1:1024], try count : 0
2020-04-04 14:55:54.851 INFO 9326 --- [k-schedule-pool] c.s.demo.aop.lock.RedisLockAspect : businessKey : [Business1:1024], try count : 1
2020-04-04 14:55:56.851 INFO 9326 --- [k-schedule-pool] c.s.demo.aop.lock.RedisLockAspect : businessKey : [Business1:1024], try count : 2
2020-04-04 14:55:58.852 INFO 9326 --- [k-schedule-pool] c.s.demo.aop.lock.RedisLockAspect : businessKey : [Business1:1024], try count : 3
2020-04-04 14:56:00.857 INFO 9326 --- [nio-8081-exec-1] c.s.demo.controller.BookController : has some error
java.lang.InterruptedIOException: sleep interrupted
at java.lang.Thread.sleep(Native Method) [na:1.8.0_221]
```

我这里测试的是重试次数过多，失败的场景，如果减少睡眠时间，就能让业务正常执行。

如果同时请求，你将会发现以下错误信息：

```
2020-04-04 14:58:36.036 ERROR 9326 --- [nio-8081-exec-9] c.s.demo.aop.lock.RedisLockAspect : has some error, please check again
java.lang.InterruptedIOException: sleep interrupted, and get the lock == 1
```

表示我们的锁确实生效了，避免了重复请求。

六、总结

对于耗时业务和核心数据，不能让重复的请求同时操作数据，避免数据的不正确，所以要使用分布式锁来对它们进行保护。

再来梳理一下设计流程：

1. 新建注解 @interface，在注解里设定入参标志
2. 增加 AOP 切点，扫描特定注解
3. 建立 @Aspect 切面任务，注册 bean 和拦截特定方法
4. 特定方法参数 ProceedingJoinPoint，对方法 pjp.proceed() 前后进行拦截
5. 切点前进行加锁，任务执行后进行删除 key

本次学习是通过 Review 小伙伴的代码设计，从中了解分布式锁的具体实现，仿照他的设计，重新写了一份简化版的业务处理。对于之前没考虑到的「续时」操作，这里使用了守护线程来定时判断和延长超时时间，避免了锁提前释放。

于是乎，同时回顾了三个知识点：

- 1、AOP 的实现和常用方法
- 2、定时线程池 ScheduledExecutorService 的使用和参数含义
- 3、线程 Thread#interrupt 的含义以及用法（这个挺有意思的，可以深入再学习一下）

具体代码放在了之前学习 SpringBoot 的项目中，感兴趣的可以克隆一下，使用这个 Redis 锁



<https://github.com/Vip-Augus/springboot-note/blob/master/src/main/java/cn/sevenyuan/demo/aop/lock/RedisLockAspect.java>

七、参考资料

- 小飞
- <https://blog.csdn.net/XWForever/article/details/103163021>
- <https://www.zhihu.com/question/4104803>

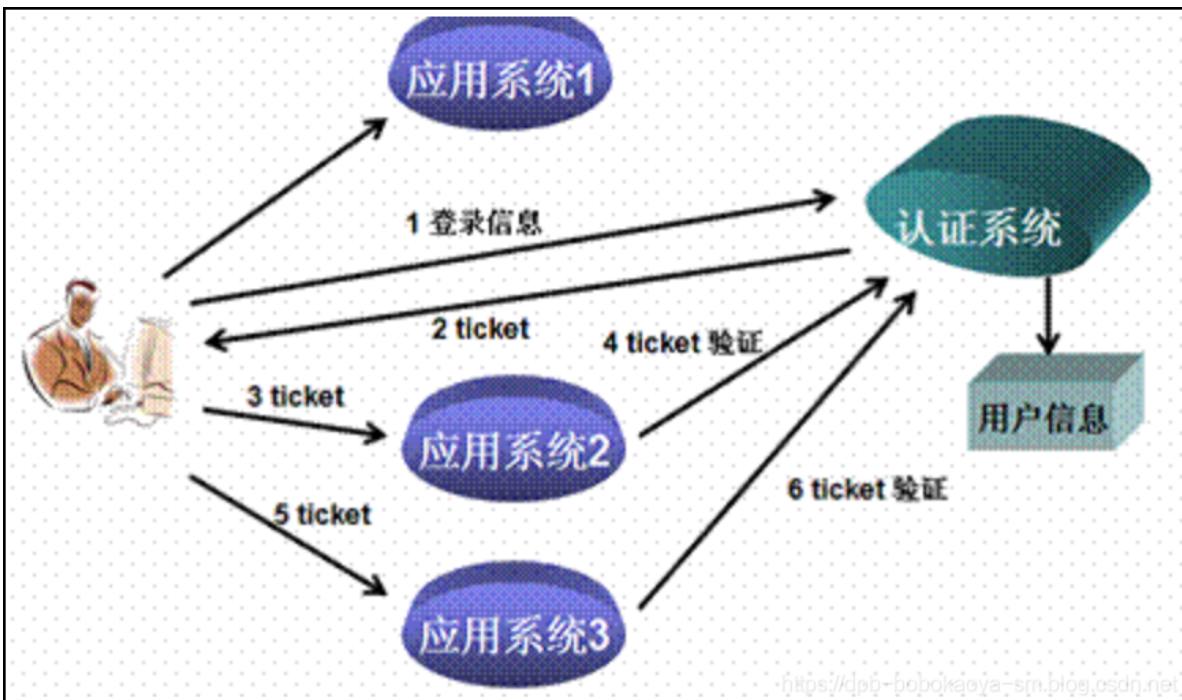
SpringBoot+JWT整合实现单点登录SSO

一、什么是单点登陆

单点登录 (Single Sign On)，简称为 SSO，是目前比较流行的企业业务整合的解决方案之一。SSO 的定义是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统

二、简单的运行机制

单点登录的机制其实是比较简单的，用一个现实中的例子做比较。某公园内部有许多独立的景点，游客可以在各个景点门口单独买票。对于需要游玩所有的景点的游客，这种买票方式很不方便，需要在每个景点门口排队买票，钱包拿进拿出的，容易丢失，很不安全。于是绝大多数游客选择在大门口买一张通票（也叫套票），关注公众号码猿技术专栏获取更多面试资源，就可以玩遍所有的景点而不需要重新再买票。他们只需要在每个景点门口出示一下刚才买的套票就能够被允许进入每个独立的景点。单点登录的机制也一样，如下图所示，



用户认证：这一环节主要是用户向认证服务器发起认证请求，认证服务器给用户返回一个成功的令牌 token，主要在认证服务器中完成，即图中的认证系统，注意认证系统只能有一个。 **身份校验**：这一环节是用户携带token去访问其他服务器时，在其他服务器中要对token的真伪进行检验，主要在资源服务器中完成，即图中的应用系统2 3

三、JWT介绍

概念说明

从分布式认证流程中，我们不难发现，这中间起最关键作用的就是token，token的安全与否，直接关系到系统的健壮性，这里我们选择使用 **JWT** 来实现token的生成和校验。**JWT**，全称 **JSON Web Token**，官网地址 <https://jwt.io>，是一款出色的分布式身份校验方案。可以生成token，也可以解析检验token。

JWT生成的token由三部分组成：

头部：主要设置一些规范信息，签名部分的编码格式就在头部中声明。**载荷**：token中存放有效信息的部分，比如用户名，用户角色，过期时间等。但是不要放密码，会泄露！**签名**：将头部与载荷分别采用base64编码后，用“.”相连，再加入盐，最后使用头部声明的编码类型进行编码，就得到了签名。

JWT生成token的安全性分析

从JWT生成的token组成上来看，要想避免token被伪造，主要就得看签名部分了，而签名部分又有三部分组成，其中头部和载荷的base64编码，几乎是透明的，毫无安全性可言，关注公众号码猿技术专栏获取更多面试资源，那么最终守护token安全的重担就落在了加入的 盐 上面了！试想：如果生成token所用的盐与解析token时加入的盐是一样的。岂不是类似于中国人民银行把人民币防伪技术公开了？大家可以用这个盐来解析token，就能用来伪造token。这时，我们就需要对盐采用 非对称加密 的方式进行加密，以达到生成token与校验token方所用的盐不一致的安全效果！

非对称加密RSA介绍

基本原理：同时生成两把密钥：私钥和公钥，私钥隐秘保存，公钥可以下发给信任客户端 私钥加密，持有私钥或公钥才可以解密 公钥加密，持有私钥才可解密 优点：安全，难以破解 缺点：算法比较耗时，为了安全，可以接受 历史：三位数学家Rivest、Shamir 和 Adleman 设计了一种算法，可以实现非对称加密。这种算法用他们三个人的名字缩写：RSA。

四、SpringSecurity整合JWT

1.认证思路分析

SpringSecurity主要是通过过滤器来实现功能的！我们要找到SpringSecurity实现认证和校验身份的过滤器！关注公众号码猿技术专栏获取更多面试资源

回顾集中式认证流程

用户认证： 使用 `UsernamePasswordAuthenticationFilter` 过滤器中 `attemptAuthentication` 方法实现认证功能，该过滤器父类中 `successfulAuthentication` 方法实现认证成功后的操作。 **身份校验**： 使用 `BasicAuthenticationFilter` 过滤器中 `doFilterInternal` 方法验证是否登录，以决定能否进入后续过滤器。

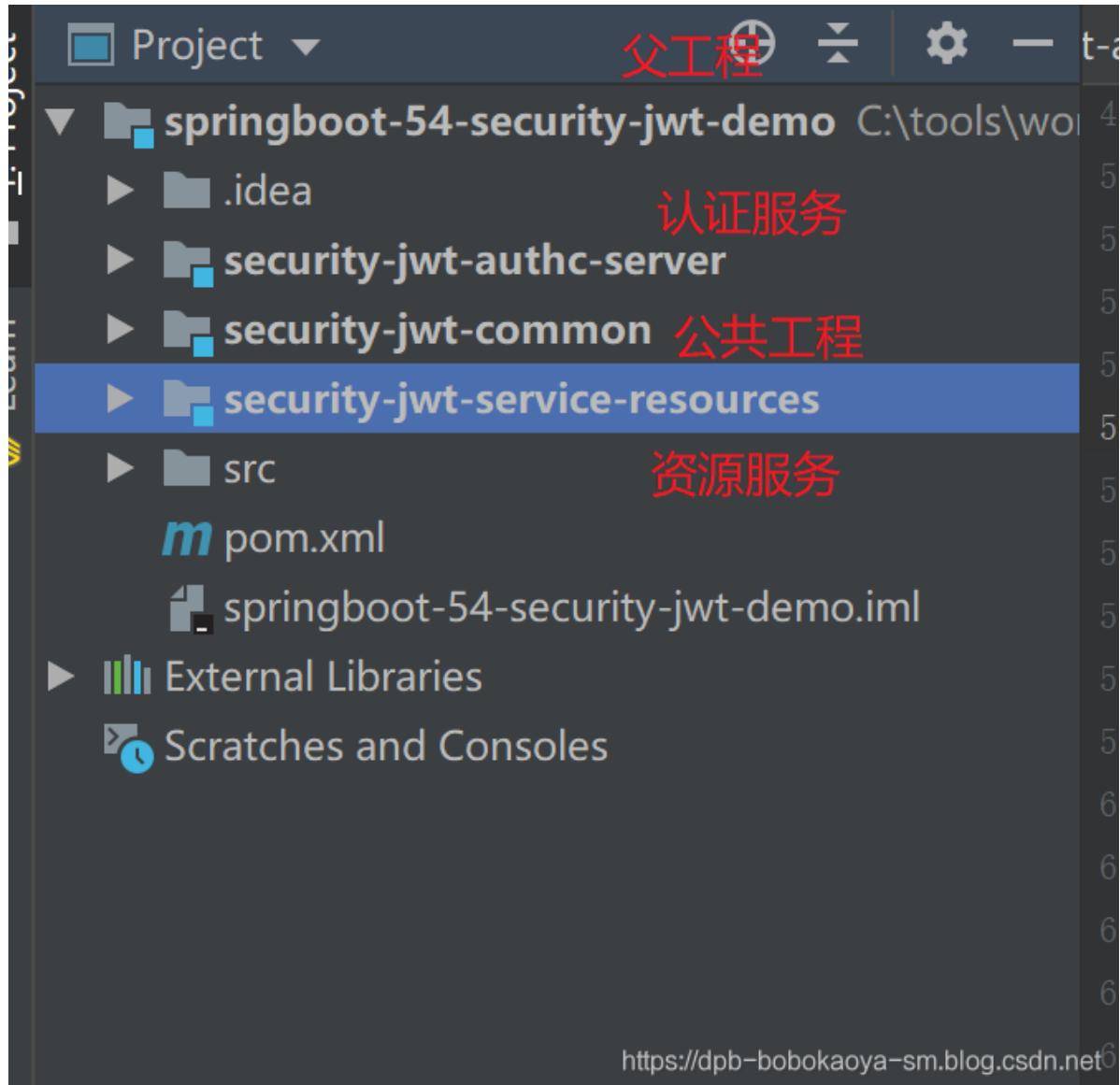
分析分布式认证流程

用户认证： 由于分布式项目，多数是前后端分离的架构设计，我们要满足可以接受异步post的认证请求参数，需要修改`UsernamePasswordAuthenticationFilter`过滤器中`attemptAuthentication`方法，让其能够接收请求体。 另外，默认`successfulAuthentication`方法在认证通过后，是把用户信息直接放入session就完事了，现在我们需要修改这个方法，在认证通过后生成token并返回给用户。 **身份校验**： 原来`BasicAuthenticationFilter`过滤器中`doFilterInternal`方法校验用户是否登录，就是看session中是否有用户信息，我们要修改为，验证用户携带的token是否合法，并解析出用户信息，交给SpringSecurity，以便于后续的授权功能可以正常使用。关注公众号码猿技术专栏获取更多面试资源

2.具体实现

Watermark

为了演示单点登录的效果，我们设计如下项目结构



<https://dpb-bobokaoya-sm.blog.csdn.net>

2.1父工程创建

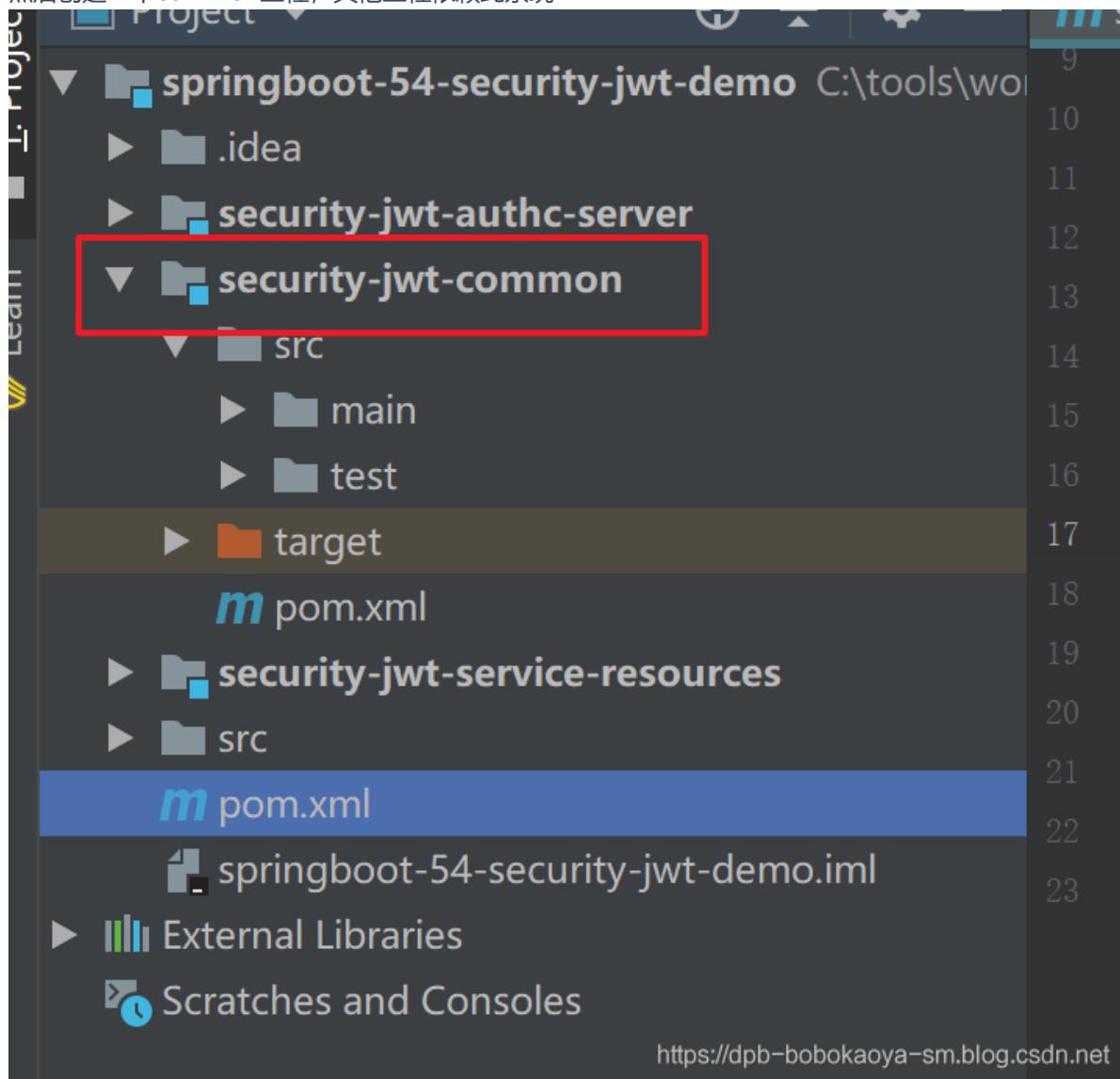
因为本案例需要创建多个系统，所以我们使用maven聚合工程来实现，首先创建一个父工程，导入springboot的父依赖即可；关注公众号码猿技术专栏获取更多面试资源

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.3.RELEASE</version>
  <relativePath/>
</parent>
123456
```

2.2公共工程创建

Watermark

然后创建一个common工程，其他工程依赖此系统



<https://dpb-bobokaoya-sm.blog.csdn.net>

导入JWT相关的依赖

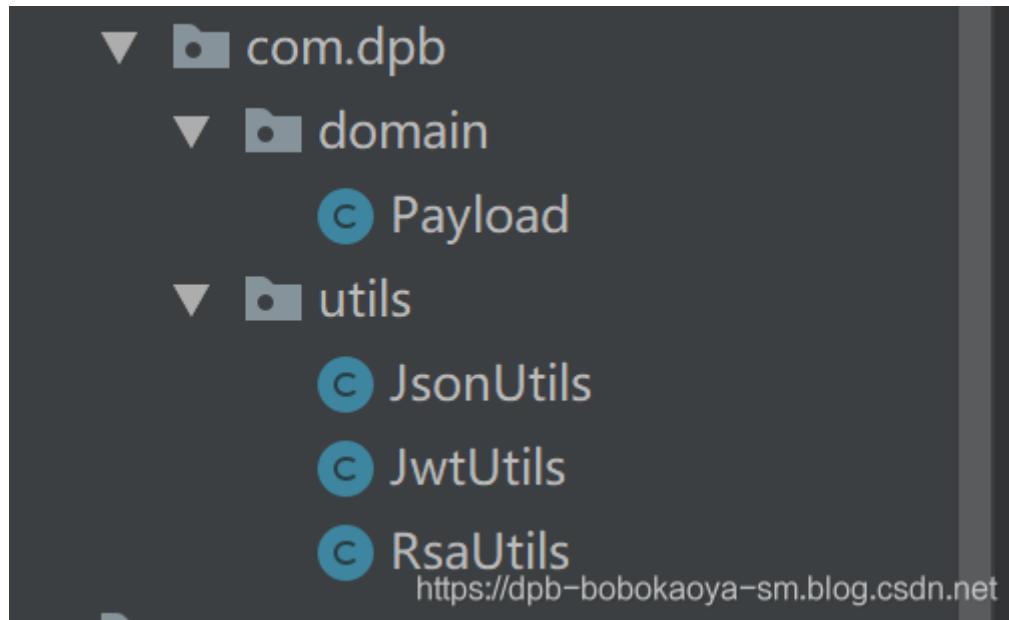
```
<dependencies>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-api</artifactId>
        <version>0.10.7</version>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-impl</artifactId>
        <version>0.10.7</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-jackson</artifactId>
        <version>0.10.7</version>
        <scope>runtime</scope>
    </dependency>
    <!-- jackson包 -->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.9.9</version>
    </dependency>
```

```

    </dependency>
    <!--日志包-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-logging</artifactId>
    </dependency>
    <dependency>
        <groupId>joda-time</groupId>
        <artifactId>joda-time</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
    </dependency>
</dependencies>

```

创建相关的工具类



Payload

```

/**
 * @program: springboot-54-security-jwt-demo
 * @description:
 * @author: 波波烤鸭
 * @create: 2019-12-03 10:28
 */
@Data
public class Payload <T>{
    private String id;
    private T userInfo;
}
123456789101112

```

JsonUtils

```
package com.dpb.utils;
```

```
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.util.List;
import java.util.Map;

/**
 * @author: 波波烤鸭
 */
public class JsonUtils {

    public static final ObjectMapper mapper = new ObjectMapper();

    private static final Logger logger = LoggerFactory.getLogger(JsonUtils.class);

    public static String toString(Object obj) {
        if (obj == null) {
            return null;
        }
        if (obj.getClass() == String.class) {
            return (String) obj;
        }
        try {
            return mapper.writeValueAsString(obj);
        } catch (JsonProcessingException e) {
            logger.error("json序列化出错: " + obj, e);
            return null;
        }
    }

    public static <T> T toBean(String json, Class<T> tClass) {
        try {
            return mapper.readValue(json, tClass);
        } catch (IOException e) {
            logger.error("json解析出错: " + json, e);
            return null;
        }
    }

    public static <E> List<E> toList(String json, Class<E> eClass) {
        try {
            return mapper.readValue(json,
                    mapper.getTypeFactory().constructCollectionType(List.class, eClass));
        } catch (IOException e) {
            logger.error("json解析出错: " + json, e);
            return null;
        }
    }

    public static <K, V> Map<K, V> toMap(String json, Class<K> kClass, Class<V> vClass) {
        try {
            return mapper.readValue(json, mapper.getTypeFactory().constructMapType(Map.class,
                    kClass, vClass));
        }
```

Watermark

```

        } catch (IOException e) {
            logger.error("json解析出错: " + json, e);
            return null;
        }
    }

    public static <T> T nativeRead(String json, TypeReference<T> type) {
        try {
            return mapper.readValue(json, type);
        } catch (IOException e) {
            logger.error("json解析出错: " + json, e);
            return null;
        }
    }
}

```

JwtUtils

```

package com.dpb.utils;

import com.dpb.domain.Payload;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jws;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.joda.time.DateTime;

import java.security.PrivateKey;
import java.security.PublicKey;
import java.util.Base64;
import java.util.UUID;

/**
 * @author: 波波烤鸭
 * 生成token以及校验token相关方法
 */
public class JwtUtils {

    private static final String JWT_PAYLOAD_USER_KEY = "user";

    /**
     * 私钥加密token
     *
     * @param userInfo 载荷中的数据
     * @param privateKey 私钥
     * @param expire 过期时间，单位分钟
     * @return JWT
     */
    public static String generateTokenExpireInMinutes(Object userInfo, PrivateKey privateKey, int expire) {
        return Jwts.builder()
                .claim(JWT_PAYLOAD_USER_KEY, JsonUtils.toString(userInfo))
                .setId(createJTI())
                .setExpiration(DateTime.now().plusMinutes(expire).toDate())
                .signWith(privateKey, SignatureAlgorithm.RS256)
                .compact();
    }
}

```

```

    /**
     * 私钥加密token
     *
     * @param userInfo 载荷中的数据
     * @param privateKey 私钥
     * @param expire 过期时间，单位秒
     * @return JWT
     */
    public static String generateTokenExpireInSeconds(Object userInfo, PrivateKey privateKey, int
expire) {
        return Jwts.builder()
            .claim(JWT_PAYLOAD_USER_KEY, JsonUtils.toString(userInfo))
            .setId(createJTI())
            .setExpiration(DateTime.now().plusSeconds(expire).toDate())
            .signWith(privateKey, SignatureAlgorithm.RS256)
            .compact();
    }

    /**
     * 公钥解析token
     *
     * @param token 用户请求中的token
     * @param publicKey 公钥
     * @return Jws<Claims>
     */
    private static Jws<Claims> parserToken(String token, PublicKey publicKey) {
        return Jwts.parser().setSigningKey(publicKey).parseClaimsJws(token);
    }

    private static String createJTI() {
        return new String(Base64.getEncoder().encode(UUID.randomUUID().toString().getBytes()));
    }

    /**
     * 获取token中的用户信息
     *
     * @param token 用户请求中的令牌
     * @param publicKey 公钥
     * @return 用户信息
     */
    public static <T> Payload<T> getInfoFromToken(String token, PublicKey publicKey, Class<T>
userType) {
        Jws<Claims> claimsJws = parserToken(token, publicKey);
        Claims body = claimsJws.getBody();
        Payload<T> claims = new Payload<T>();
        claims.setId(body.getId());
        claims.setUserInfo(JsonUtils.getBean(body.get(JWT_PAYLOAD_USER_KEY).toString(), userType));
        claims.setExpiration(body.getExpiration());
        return claims;
    }
}

*! 获取token中的载荷信息
*
* @param token 用户请求中的令牌
* @param publicKey 公钥
* @return 用户信息

```

Watermark

*! 获取token中的载荷信息
*
* @param token 用户请求中的令牌
* @param publicKey 公钥
* @return 用户信息

```

/*
public static <T> Payload<T> getInfoFromToken(String token, PublicKey publicKey) {
    Jws<Claims> claimsJws = parserToken(token, publicKey);
    Claims body = claimsJws.getBody();
    Payload<T> claims = new Payload<T>();
    claims.setId(body.getId());
    claims.setExpiration(body.getExpiration());
    return claims;
}
}

```

RsaUtils

```

package com.dpb.utils;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.security.*;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.Base64;

/**
 * @author 波波烤鸭
 */
public class RsaUtils {

    private static final int DEFAULT_KEY_SIZE = 2048;
    /**
     * 从文件中读取公钥
     *
     * @param filename 公钥保存路径，相对于classpath
     * @return 公钥对象
     * @throws Exception
     */
    public static PublicKey getPublicKey(String filename) throws Exception {
        byte[] bytes = readFile(filename);
        return getPublicKey(bytes);
    }

    /**
     * 从文件中读取密钥
     *
     * @param filename 私钥保存路径，相对于classpath
     * @return 私钥对象
     * @throws Exception
     */
    public static PrivateKey getPrivateKey(String filename) throws Exception {
        byte[] bytes = readFile(filename);
        return getPrivateKey(bytes);
    }

    /**
     * 获取公钥
     *
     * @param bytes 公钥的字节形式
     */

```

Watermark

```
* @return
* @throws Exception
*/
private static PublicKey getPublicKey(byte[] bytes) throws Exception {
    bytes = Base64.getDecoder().decode(bytes);
    X509EncodedKeySpec spec = new X509EncodedKeySpec(bytes);
    KeyFactory factory = KeyFactory.getInstance("RSA");
    return factory.generatePublic(spec);
}

/**
 * 获取密钥
 *
 * @param bytes 私钥的字节形式
 * @return
 * @throws Exception
*/
private static PrivateKey getPrivateKey(byte[] bytes) throws NoSuchAlgorithmException,
InvalidKeySpecException {
    bytes = Base64.getDecoder().decode(bytes);
    PKCS8EncodedKeySpec spec = new PKCS8EncodedKeySpec(bytes);
    KeyFactory factory = KeyFactory.getInstance("RSA");
    return factory.generatePrivate(spec);
}

/**
 * 根据密文，生存rsa公钥和私钥，并写入指定文件
 *
 * @param publicKeyFilename 公钥文件路径
 * @param privateKeyFilename 私钥文件路径
 * @param secret           生成密钥的密文
 */
public static void generateKey(String publicKeyFilename, String privateKeyFilename, String secret,
int keySize) throws Exception {
    KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
    SecureRandom secureRandom = new SecureRandom(secret.getBytes());
    keyPairGenerator.initialize(Math.max(keySize, DEFAULT_KEY_SIZE), secureRandom);
    KeyPair keyPair = keyPairGenerator.genKeyPair();
    // 获取公钥并写出
    byte[] publicKeyBytes = keyPair.getPublic().getEncoded();
    publicKeyBytes = Base64.getEncoder().encode(publicKeyBytes);
    writeFile(publicKeyFilename, publicKeyBytes);
    // 获取私钥并写出
    byte[] privateKeyBytes = keyPair.getPrivate().getEncoded();
    privateKeyBytes = Base64.getEncoder().encode(privateKeyBytes);
    writeFile(privateKeyFilename, privateKeyBytes);
}

private static byte[] readFile(String fileName) throws Exception {
    return Files.readAllBytes(new File(fileName).toPath());
}

private static void writeFile(Path destPath, byte[] bytes) throws IOException {
    File dest = new File(destPath);
    if (!dest.exists()) {
        dest.createNewFile();
    }
    Files.write(dest.toPath(), bytes);
}
```

```
    }  
}
```

在通用子模块中编写测试类生成rsa公钥和私钥

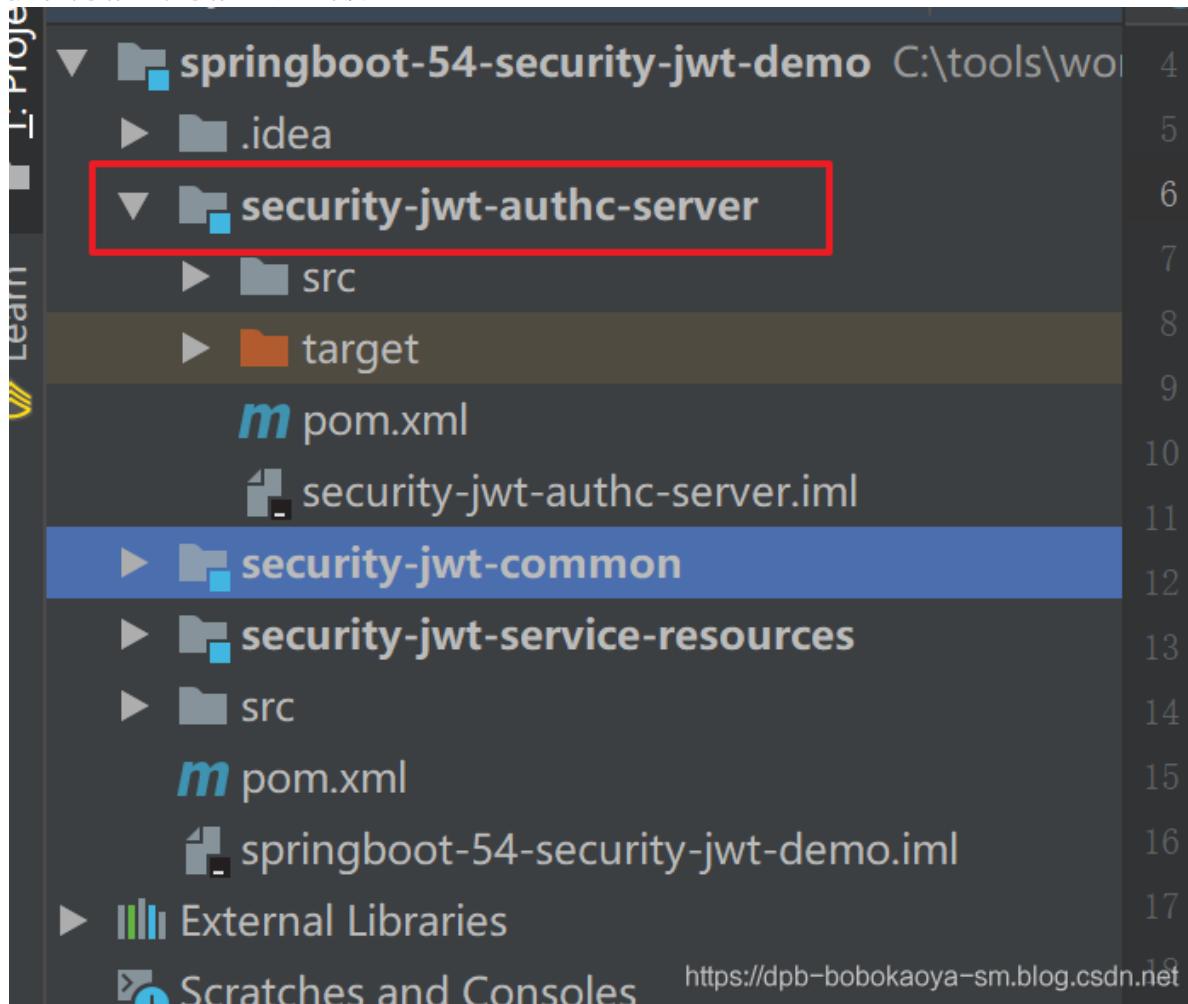
```
/**  
 * @program: springboot-54-security-jwt-demo  
 * @description:  
 * @author: 波波烤鸭  
 * @create: 2019-12-03 11:08  
 */  
public class JwtTest {  
    private String privateKey = "c:/tools/auth_key/id_key_rsa";  
  
    private String publicKey = "c:/tools/auth_key/id_key_rsa.pub";  
  
    @Test  
    public void test1() throws Exception{  
        RsaUtils.generateKey(publicKey, privateKey, "dpb", 1024);  
    }  
}
```

电脑 > BOOTCAMP (C:) > tools > auth_key				
	名称	修改日期	类型	大小
	id_key_rsa	2019/12/3 20:44	文件	2 KB
	id_key_rsa.pub	2019/12/3 20:44	Microsoft Office Pu...	1 KB

2.3认证系统创建

Watermark

接下来我们创建我们的认证服务。



导入相关的依赖

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <artifactId>security-jwt-common</artifactId>
        <groupId>com.dpb</groupId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>2.1.0</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
```

```

<artifactId>druid</artifactId>
<version>1.1.10</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
</dependencies>

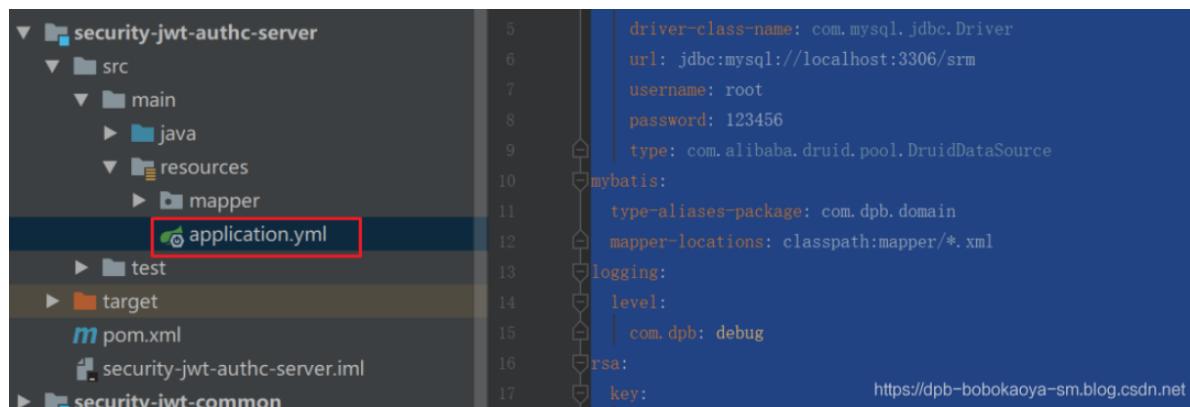
```

创建配置文件

```

spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/srm
    username: root
    password: 123456
    type: com.alibaba.druid.pool.DruidDataSource
  mybatis:
    type-aliases-package: com.dpb.domain
    mapper-locations:classpath:mapper/*.xml
  logging:
    level:
      com.dpb: debug
  rsa:
    key:
      publicKeyFile: c:\tools\auth_key\id_key_rsa.pub
      priKeyFile: c:\tools\auth_key\id_key_rsa

```



提供公钥私钥的配置类

```

package com.dpb.config;

import com.dpb.utils.RsaUtils;
import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;

import javax.validation.constraints.NotNull;
import java.security.PrivateKey;
import java.security.PublicKey;

/**
 * @program: springboot-54-security-jwt-demo
 * @description:

```

```

 * @author: 波波烤鸭
 * @create: 2019-12-03 11:25
 */
@Data
@ConfigurationProperties(prefix = "rsa.key")
public class RsaKeyProperties {

    private String pubKeyFile;
    private String priKeyFile;

    private PublicKey publicKey;
    private PrivateKey privateKey;

    /**
     * 系统启动的时候触发
     * @throws Exception
     */
    @PostConstruct
    public void createRsaKey() throws Exception {
        publicKey = RsaUtils.getPublicKey(pubKeyFile);
        privateKey = RsaUtils.getPrivateKey(priKeyFile);
    }

}

```

创建启动类

```

/**
 * @program: springboot-54-security-jwt-demo
 * @description: 启动类
 * @author: 波波烤鸭
 * @create: 2019-12-03 11:23
 */
@SpringBootApplication
@MapperScan("com.dpb.mapper")
@EnableConfigurationProperties(RsaKeyProperties.class)
public class App {

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

```

完成数据认证的逻辑

pojo

```

package com.dpb.domain;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.security.core.userdetails.UserAuthority;

/**
 * @program: springboot-54-security-jwt-demo
 * @description:
 * @author: 波波烤鸭

```

```
* @create: 2019-12-03 15:21
*/
@Data
public class RolePojo implements GrantedAuthority {

    private Integer id;
    private String roleName;
    private String roleDesc;

    @JsonIgnore
    @Override
    public String getAuthority() {
        return roleName;
    }
}

12345678910111213141516171819202122232425
package com.dpb.domain;

import com.fasterxml.jackson.annotation.JsonIgnore;
import lombok.Data;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

/**
 * @program: springboot-54-security-jwt-demo
 * @description:
 * @author: 波波烤鸭
 * @create: 2019-12-03 11:33
 */
@Data
public class UserPojo implements UserDetails {

    private Integer id;

    private String username;

    private String password;

    private Integer status;

    private List<RolePojo> roles;

    @JsonIgnore
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<SimpleGrantedAuthority> auth = new ArrayList<>();
        auth.add(new SimpleGrantedAuthority("ADMIN"));
        return auth;
    }

    @Override
    public String getPassword() {
        return this.password;
```

```
}

@Override
public String getUsername() {
    return this.username;
}

@JsonIgnore
@Override
public boolean isAccountNonExpired() {
    return true;
}

@JsonIgnore
@Override
public boolean isAccountNonLocked() {
    return true;
}

@JsonIgnore
@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@JsonIgnore
@Override
public boolean isEnabled() {
    return true;
}
}
```

Mapper接口

```
public interface UserMapper {
    public UserPojo queryByUserName(@Param("userName") String userName);
}
```

Mapper映射文件

```
<?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.dpb.mapper.UserMapper">
    <select id="queryByUserName" resultType="UserPojo">
        select * from t_user where username = #{userName}
    </select>
</mapper>
```

Service

```
public interface UserService extends UserDetailsService {

}

123
@Service
@Transactional
public class UserServiceImpl implements UserService {

    @Autowired
    private UserMapper mapper;
```

```

    @Override
    public UserDetails loadUserByUsername(String s) throws UsernameNotFoundException {
        UserPojo user = mapper.queryUserName(s);

        return user;
    }
}

```

自定义认证过滤器

```

package com.dpb.filter;

import com.dpb.config.RsaKeyProperties;
import com.dpb.domain.RolePojo;
import com.dpb.domain.UserPojo;
import com.dpb.utils.JwtUtils;
import com.fasterxml.jackson.databind.ObjectMapper;
import net.bytebuddy.agent.builder.AgentBuilder;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * @program: springboot-54-security-jwt-demo
 * @description:
 * @author: 波波烤鸭
 * @create: 2019-12-03 11:57
 */
public class TokenLoginFilter extends UsernamePasswordAuthenticationFilter {

    private AuthenticationManager authenticationManager;
    private RsaKeyProperties prop;

    public TokenLoginFilter(AuthenticationManager authenticationManager, RsaKeyProperties prop) {
        this.authenticationManager = authenticationManager;
        this.prop = prop;
    }

    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {

```

```

        try {
            UserPojo sysUser = new ObjectMapper().readValue(request.getInputStream(),
UserPojo.class);

            UsernamePasswordAuthenticationToken authRequest = new
UsernamePasswordAuthenticationToken(sysUser.getUsername(), sysUser.getPassword());
            return authenticationManager.authenticate(authRequest);
        } catch (Exception e) {
            try {
                response.setContentType("application/json;charset=utf-8");
                response.setStatus(HttpStatus.SC_UNAUTHORIZED);
                PrintWriter out = response.getWriter();
                Map resultMap = new HashMap();
                resultMap.put("code", HttpStatus.SC_UNAUTHORIZED);
                resultMap.put("msg", "用户名或密码错误！");
                out.write(new ObjectMapper().writeValueAsString(resultMap));
                out.flush();
                out.close();
            } catch (Exception outEx) {
                outEx.printStackTrace();
            }
            throw new RuntimeException(e);
        }
    }

    public void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
FilterChain chain, Authentication authResult) throws IOException, ServletException {
        UserPojo user = new UserPojo();
        user.setUsername(authResult.getName());
        user.setRoles((List<RolePojo>)authResult.getAuthorities());
        String token = JwtUtils.generateTokenExpireInMinutes(user, prop.getPrivateKey(), 24 * 60);
        response.addHeader("Authorization", "Bearer "+token);
        try {
            response.setContentType("application/json;charset=utf-8");
            response.setStatus(HttpStatus.SC_OK);
            PrintWriter out = response.getWriter();
            Map resultMap = new HashMap();
            resultMap.put("code", HttpStatus.SC_OK);
            resultMap.put("msg", "认证通过！");
            out.write(new ObjectMapper().writeValueAsString(resultMap));
            out.flush();
            out.close();
        } catch (Exception outEx) {
            outEx.printStackTrace();
        }
    }
}

```

自定义校验token的过滤器

```

package com.dpb.filter;
import com.dpb.config.JasyptProperties;
import com.dpb.domain.Payload;
import com.dpb.domain.UserPojo;
import com.dpb.utils.JwtUtils;
import com.fasterxml.jackson.databind.ObjectMapper;

```

```
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.authentication.www.BasicAuthenticationFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map;

/**
 * @program: springboot-54-security-jwt-demo
 * @description:
 * @author: 波波烤鸭
 * @create: 2019-12-03 12:39
 */
public class TokenVerifyFilter extends BasicAuthenticationFilter {
    private RsaKeyProperties prop;

    public TokenVerifyFilter(AuthenticationManager authenticationManager, RsaKeyProperties prop) {
        super(authenticationManager);
        this.prop = prop;
    }

    public void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws IOException, ServletException {
        String header = request.getHeader("Authorization");
        if (header == null || !header.startsWith("Bearer ")) {
            //如果携带错误的token，则给用户提示请登录!
            chain.doFilter(request, response);
            response.setContentType("application/json;charset=utf-8");
            response.setStatus(HttpServletRequest.SC_FORBIDDEN);
            PrintWriter out = response.getWriter();
            Map resultMap = new HashMap();
            resultMap.put("code", HttpServletRequest.SC_FORBIDDEN);
            resultMap.put("msg", "请登录!");
            out.write(new ObjectMapper().writeValueAsString(resultMap));
            out.flush();
            out.close();
        } else {
            //如果携带了正确格式的token要先得到token
            String token = header.replace("Bearer ", "");
            //验证token是否正确
            Payload<UserPojo> payload = JwtUtils.getInfoFromToken(token, prop.getPublicKey(),
                    UserPojo.class);
            UserPojo user = payload.getUserInfo();
            if(user!=null){
                UsernamePasswordAuthenticationToken authResult = new UsernamePasswordAuthenticationToken(token, user.getUsername(), null, user.getAuthorities());
                SecurityContextHolder.getContext().setAuthentication(authResult);
                chain.doFilter(request, response);
            }
        }
    }
}
```

```
}
```

编写SpringSecurity的配置类

```
package com.dpb.config;

import com.dpb.filter.TokenLoginFilter;
import com.dpb.filter.TokenVerifyFilter;
import com.dpb.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

/**
 * @program: springboot-54-security-jwt-demo
 * @description:
 * @author: 波波烤鸭
 * @create: 2019-12-03 12:41
 */
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled=true)
public class WebSecurityConfig     extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserService userService;

    @Autowired
    private RsaKeyProperties prop;

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    //指定认证对象的来源
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userService).passwordEncoder(passwordEncoder());
    }

    //SpringSecurity配置信息
    public void configure(HttpSecurity http) throws Exception {
        http.csrf()
            .disable()
            .authorizeRequests()
                .antMatchers("/user/query").hasAnyRole("ADMIN")
                .anyRequest()
                    .authenticated()
                    .and()
    }
}
```

Watermark

```

        .addFilter(new TokenLoginFilter(super.authenticationManager(), prop))
        .addFilter(new TokenVerifyFilter(super.authenticationManager(), prop))
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    }
}

```

启动服务测试

启动服务

```

2019-12-04 18:49:29.331 INFO 9910 --- [           main] com.apollo.App                : No active profile set, falling back to default profiles: default
2019-12-04 18:49:32.497 INFO 9916 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 9001 (http)
2019-12-04 18:49:32.562 INFO 9916 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-12-04 18:49:32.563 INFO 9916 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.16]
2019-12-04 18:49:32.574 INFO 9916 --- [           main] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: [/usr/lib64:/lib64:/usr/lib:/lib]
2019-12-04 18:49:32.990 INFO 9916 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2019-12-04 18:49:32.990 INFO 9916 --- [           main] o.s.web.context.ContextLoader      : Root WebApplicationContext: initialization completed in 3511 ms
2019-12-04 18:49:33.949 INFO 9916 --- [           main] o.s.s.web.DefaultSecurityFilterChain : Creating filter chain: any request, [org.springframework.security.filter]
2019-12-04 18:49:34.123 INFO 9916 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-12-04 18:49:34.427 INFO 9916 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 9001 (http) with context path ''
2019-12-04 18:49:34.431 INFO 9916 --- [           main] com.apollo.App                  : Started App in 6.469 seconds (JVM running for 11.549)
2019-12-04 18:50:28.701 INFO 9916 --- [nio-9001-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2019-12-04 18:50:28.701 INFO 9916 --- [nio-9001-exec-1] o.s.web.servlet.DispatcherServlet   : Initializing Servlet 'dispatcherServlet'
2019-12-04 18:50:28.717 INFO 9916 --- [nio-9001-exec-1] o.s.web.servlet.DispatcherServlet   : Completed initialization in 6 ms

```

通过Postman来访问测试

The screenshot shows a Postman interface with the following details:

- Request Method:** POST
- URL:** http://localhost:9001/login
- Body:** JSON (highlighted with a red box)
- Body Content:**

```
{
  "username": "cs1",
  "password": "123"
}
```
- Response Status:** 200 OK
- Response Body:**

```

1 {
2   "msg": "认证通过!",
3   "code": 200
4 }
```

Watermark

Body Cookies Headers (10) Test Results

Status: 200 OK Time: 1340ms Size: 991 B Save Response

KEY	VALUE
Authorization	Bearer eyJhbGciOiJSUzI1NiJ9eyJ1c2Vyljoie1wiaWRcljpudWxsLFwidXNlcm5hbWVclpclmNzMWwlFwicGFzc3dvcmRcljpudWxsLFwic3RhHVzXC16bnVsbCxclnjvbGVzXC16W3tclmF1dGhvcmloevWliQURNSUScln1dfSlsImp0aSI6k0yTTRNRFPzT1RBdE5qY3dOUzAwTTjVMEXuUZ3lOemN0TxprMU1ETTVNVE5qTVdjdylsImV4cCl6MTU3NTU0NjYyNn0.SkeC9MwnNG-A2jO4-5zqGkfimpyHs0_LkBAPPHWZyOMml_EhtpmGGuBodvnO5xbVmpM3p_4BRAjPaAf6j6qWRN9AcjCrmElpbM4LifwbtDo-1_ZQazUEzkvfRS9gsU77VF5gAluKlFYDdCbY_ExzPFE2hA-6xiqwRQiXTaC-ptOMlms3eQATn5gZUDbsxP8wxz2OTmpVswQy4SjwCL_4pEOG6VetT_wVuyGBKbOULf1ylHo850dp6PNEOVnBR4FCvtxaEF1fjCWnHITXTXWSAuq-4gIyyKGcF_ae0RtRhuYq4uqDMmZRUOkxPzZyrfhf_c9nKhbYmPMNjP3DA
Content-Type	application/json; charset=utf-8
Transfer-Encoding	chunked

<https://dpb-bobokaoya-sm.blog.csdn.net>

根据token信息我们访问其他资源

Untitled Request

GET http://localhost:9001/user/update Send Save

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Headers (1)

KEY	VALUE	DESCRIPTION
Authorization	Bearer eyJhbGciOiJSUzI1NiJ9eyJ1c2Vyljoie1wiaWRclpclmNzMWwlFwicGFzc3dvcmRcljpudWxsLFwic3RhHVzXC16bnVsbCxclnjvbGVzXC16W3tclmF1dGhvcmloevWliQURNSUScln1dfSlsImp0aSI6k0yTTRNRFPzT1RBdE5qY3dOUzAwTTjVMEXuUZ3lOemN0TxprMU1ETTVNVE5qTVdjdylsImV4cCl6MTU3NTU0NjYyNn0.SkeC9MwnNG-A2jO4-5zqGkfimpyHs0_LkBAPPHWZyOMml_EhtpmGGuBodvnO5xbVmpM3p_4BRAjPaAf6j6qWRN9AcjCrmElpbM4LifwbtDo-1_ZQazUEzkvfRS9gsU77VF5gAluKlFYDdCbY_ExzPFE2hA-6xiqwRQiXTaC-ptOMlms3eQATn5gZUDbsxP8wxz2OTmpVswQy4SjwCL_4pEOG6VetT_wVuyGBKbOULf1ylHo850dp6PNEOVnBR4FCvtxaEF1fjCWnHITXTXWSAuq-4gIyyKGcF_ae0RtRhuYq4uqDMmZRUOkxPzZyrfhf_c9nKhbYmPMNjP3DA	Description

Temporary Headers (7)

Body Cookies Headers (9) Test Results

Pretty Raw Preview Visualize BETA Text

1 update

Status: 200 OK Time: 93ms Size: 303 B Save Response

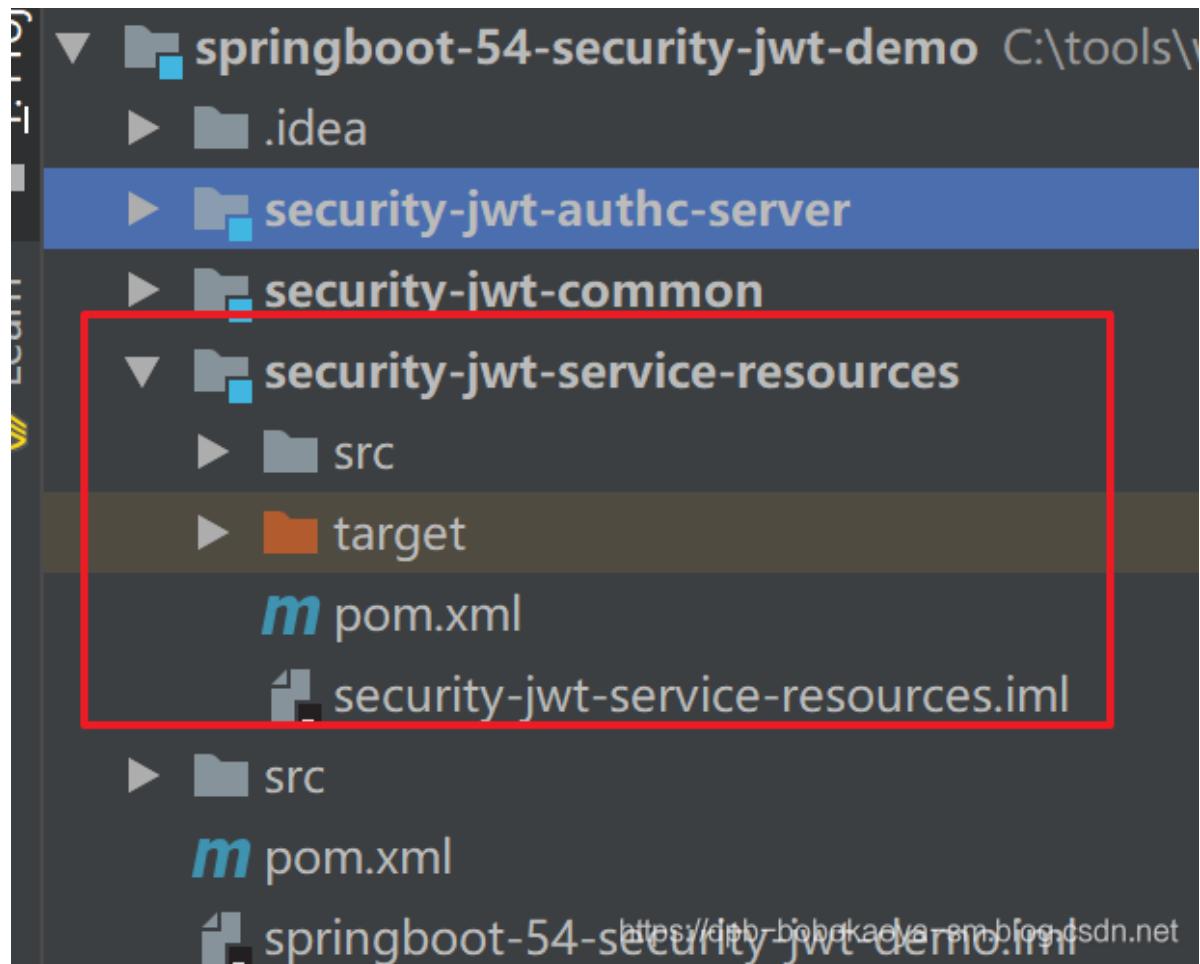
<https://dpb-bobokaoya-sm.blog.csdn.net>

2.4 资源系统创建

说明 资源服务可以有很多个，这里只拿产品服务为例，记住，资源服务中只能通过公钥验证认证。不能签发token！创建产品服务并导入jar包根据实际业务导包即可，咱们就暂时和认证服务一样了。

接下来我们再创建一个资源服务

Watermark



导入相关的依赖

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <artifactId>security-jwt-common</artifactId>
        <groupId>com.dpb</groupId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>2.0.0</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.10</version>
    </dependency>
```

```
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
</dependencies>
1234567891011121314151617181920212223242526272829303132333435
```

编写产品服务配置文件

切记这里只能有公钥地址!

```
server:
  port: 9002
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/srm
    username: root
    password: 123456
    type: com.alibaba.druid.pool.DruidDataSource
  mybatis:
    type-aliases-package: com.dpb.domain
    mapper-locations: classpath:mapper/*.xml
  logging:
    level:
      com.dpb: debug
rsa:
  key:
    pubKeyFile: c:\tools\auth_key\id_key_rsa.pub
```

编写读取公钥的配置类

```
package com.dpb.config;

import com.dpb.utils.RsaUtils;
import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

import javax.annotation.PostConstruct;
import java.security.PrivateKey;
import java.security.PublicKey;

/**
 * @program: springboot-54-security-jwt-demo
 * @description:
 * @author: 波波烤鸭
 * @create: 2019-12-03 11:25
 */
@Data
@ConfigurationProperties(prefix = "rsa")
public class RsaKeyProperties {

    private String pubKeyFile;

    private PublicKey publicKey;
```

```

    /**
     * 系统启动的时候触发
     * @throws Exception
     */
    @PostConstruct
    public void createRsaKey() throws Exception {
        publicKey = RsaUtils.getPublicKey(pubKeyFile);
    }

}

```

编写启动类

```

package com.dpb;

import com.dpb.config.RsaKeyProperties;
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.EnableConfigurationProperties;

/**
 * @program: springboot-54-security-jwt-demo
 * @description:
 * @author: 波波烤鸭
 * @create: 2019-12-03 17:23
 */
@SpringBootApplication
@MapperScan("com.dpb.mapper")
@EnableConfigurationProperties(RsaKeyProperties.class)
public class App {

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

```

复制认证服务中，用户对象，角色对象和校验认证的接口

复制认证服务中的相关内容即可

复制认证服务中SpringSecurity配置类做修改

```

package com.dpb.config;

import com.dpb.filter.TokenVerifyFilter;
import com.dpb.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

```

```

/**
 * @program: springboot-54-security-jwt-demo
 * @description:
 * @author: 波波烤鸭
 * @create: 2019-12-03 12:41
 */
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled=true)
public class WebSecurityConfig     extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserService userService;

    @Autowired
    private RsaKeyProperties prop;

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    //指定认证对象的来源
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userService).passwordEncoder(passwordEncoder());
    }

    //SpringSecurity配置信息
    public void configure(HttpSecurity http) throws Exception {
        http.csrf()
            .disable()
            .authorizeRequests()
                //.antMatchers("/user/query").hasAnyRole("USER")
                .anyRequest()
                .authenticated()
                .and()
                .addFilter(new TokenVerifyFilter(super.authenticationManager(), prop))
                // 禁用掉session
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    }
}

```

去掉“增加自定义认证过滤器”即可！

编写产品处理器

```

package com.dpb.controller;

import org.springframework.security.access.annotation.Secured;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

```

```

@RestController
@RequestMapping("/user")
public class UserController {

    @RequestMapping("/query")
    public String query() {
        return "success";
    }

    @RequestMapping("/update")
    public String update() {
        return "update";
    }
}

```

测试

The screenshot shows a Postman interface with the following details:

- Request Method:** GET
- Request URL:** `http://localhost:9002/user/update`
- Headers Tab:** Selected. It contains one entry: `Authorization: Bearer eyJhbGciOiJSUzI1NiJ9.eyJ1c2VyIjoiZW1vZW...`.
- Body Tab:** Contains the word `update`.
- Status Bar:** Status: 200 OK, Time: 22ms, Size: 303 B.

搞定~

SpringBoot 接口幂等性的实现方案，真服了

~

系统环境：

- Java JDK 版本：1.8
- SpringBoot 版本：2.3.4.RELEASE

示例地址：

Watermark
<https://github.com/my-cq-bio/example/tree/master/springboot/springboot-idempotent-token/>

一、什么是幂等性

幂等是一个数学与计算机学概念，在数学中某一元运算为幂等时，其作用在任一元素两次后会和其作用一次的结果相同。在计算机中编程中，一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同。

幂等函数或幂等方法是指可以使用相同参数重复执行，并能获得相同结果的函数。这些函数不会影响系统状态，也不用担心重复执行会对系统造成改变。

二、什么是接口幂等性

在HTTP/1.1中，对幂等性进行了定义。它描述了一次和多次请求某一个资源对于资源本身应该具有同样的结果（网络超时等问题除外），即第一次请求的时候对资源产生了副作用，但是以后的多次请求都不会再对资源产生副作用。

这里的副作用是不会对结果产生破坏或者产生不可预料的结果。也就是说，其任意多次执行对资源本身所产生的影响均与一次执行的影响相同。

三、为什么需要实现幂等性

在接口调用时一般情况下都能正常返回信息不会重复提交，不过在遇见以下情况时可以就会出现问题，如：

- **前端重复提交表单：**在填写一些表格时候，用户填写完成提交，很多时候会因网络波动没有及时对用户做出提交成功响应，致使用户认为没有成功提交，然后一直点提交按钮，这时就会发生重复提交表单请求。
- **用户恶意进行刷单：**例如在实现用户投票这种功能时，如果用户针对一个用户进行重复提交投票，这样会导致接口接收到用户重复提交的投票信息，这样会使投票结果与事实严重不符。
- **接口超时重复提交：**很多时候 HTTP 客户端工具都默认开启超时重试的机制，尤其是第三方调用接口时候，为了防止网络波动超时等造成的请求失败，都会添加重试机制，导致一个请求提交多次。
- **消息进行重复消费：**当使用 MQ 消息中间件时候，如果发生消息中间件出现错误未及时提交消费信息，导致发生重复消费。

使用幂等性最大的优势在于使接口保证任何幂等性操作，免去因重试等造成系统产生的未知的问题。

四、引入幂等性后对系统的影响

幂等性是为了简化客户端逻辑处理，能放置重复提交等操作，但却增加了服务端的逻辑复杂性和成本，其主要是：

- 把并行执行的功能改为串行执行，降低了执行效率。
- 增加了额外控制幂等的业务逻辑，复杂化了业务功能；

所以在使用时候要考虑是否引入幂等性的必要性，根据实际业务场景具体分析，除了业务上的特殊要求外，一般情况下不需要引入的接口幂等性。

五、Restful API 接口的幂等性

现在流行的 Restful 推荐的几种 HTTP 接口方法中，分别存在幂等行与不能保证幂等的方法，如下：

- ✓ 满足幂等
- ✗ 不满足幂等
- - 可能满足也可能不满足幂等，根据实际业务逻辑有关

方法类型	是否幂等	描述
Get	✓	Get 方法用于获取资源。其一般不会也不应当对系统资源进行改变，所以是幂等的。
Post	✗	Post 方法一般用于创建新的资源。其每次执行都会新增数据，所以不是幂等的。
Put	-	Put 方法一般用于修改资源。该操作则分情况来看是不是满足幂等，更新操作中直接根据某个值进行更新，也能保持幂等。不过执行累加操作的更新是非幂等的。
Delete	-	Delete 方法一般用于删除资源。该操作则分情况来看是不是满足幂等，当根据唯一值进行删除时，删除同一个数据多次执行效果一样。不过需要注意，带查询条件的删除则就不一定满足幂等了。例如在根据条件删除一批数据后，这时候新增加了一条数据也满足条件，然后又执行了一次删除，那么将会导致新增加的这条满足条件的数据也被删除。 

六、如何实现幂等性

方案一：数据库唯一主键

方案描述

数据库唯一主键的实现主要是利用数据库中主键唯一约束的特性，一般来说唯一主键比较适用于“插入”时的幂等性，其能保证一张表中只能存在一条带该唯一主键的记录。

使用数据库唯一主键完成幂等性时需要注意的是，该主键一般来说并不是使用数据库中自增主键，而是使用分布式 ID 充当主键，这样才能保证在分布式环境下 ID 的全局唯一性。

适用操作：

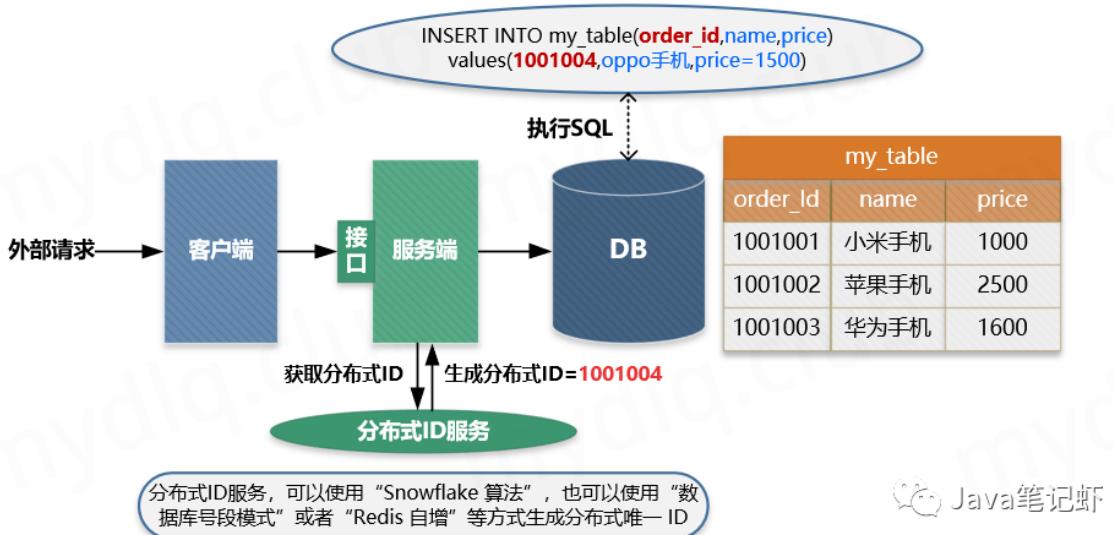
- 插入操作
- 删除操作

使用限制：

- 需要生成全局唯一主键 ID；

主要流程：

Watermark



主要流程:

- ① 客户端执行创建请求, 调用服务端接口。
- ② 服务端执行业务逻辑, 生成一个分布式 ID, 将该 ID 充当待插入数据的主键, 然后执数据插入操作, 运行对应的 SQL 语句。
- ③ 服务端将该条数据插入数据库中, 如果插入成功则表示没有重复调用接口。如果抛出主键重复异常, 则表示数据库中已经存在该条记录, 返回错误信息到客户端。

方案二：数据库乐观锁

方案描述:

数据库乐观锁方案一般只能适用于执行“更新操作”的过程, 我们可以提前在对应的数据表中多添加一个字段, 充当前数据的版本标识。这样每次对该数据库该表的这条数据执行更新时, 都会将该版本标识作为一个条件, 值为上次待更新数据中的版本标识的值。

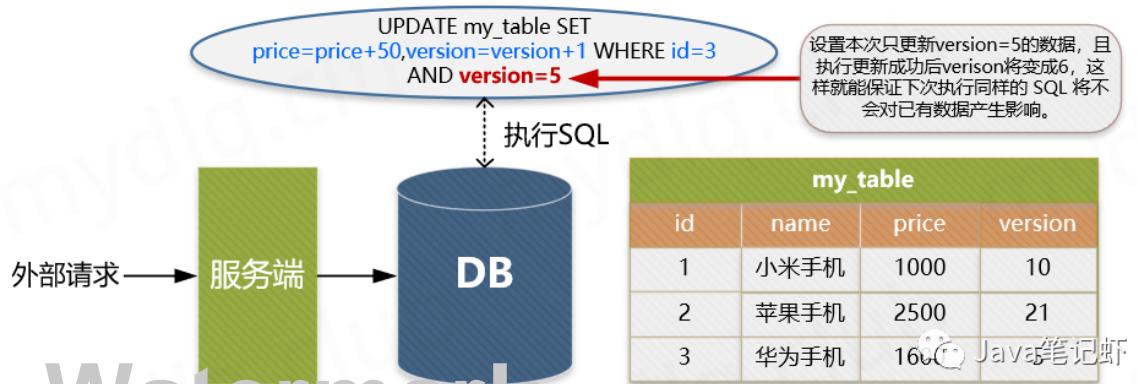
适用操作:

- 更新操作

使用限制:

- 需要数据库对应业务表中添加额外字段;

描述示例:



例如, 存在如下的数据表中:

id	name	price
1	小米手机	1000
2	苹果手机	2500
3	华为手机	1600

 Java笔记虾

为了每次执行更新时防止重复更新，确定更新的一定是要更新的内容，我们通常都会添加一个 version 字段记录当前的记录版本，这样在更新时候将该值带上，那么只要执行更新操作就能确定一定更新的是某个对应版本下的信息。

id	name	price	version
1	小米手机	1000	10
2	苹果手机	2500	21
3	华为手机	1600	5

 Java笔记虾

这样每次执行更新时候，都要指定要更新的版本号，如下操作就能准确更新 version=5 的信息：

```
UPDATE my_table SET price=price+50,version=version+1 WHERE id=1 AND version=5
```

上面 WHERE 后面跟着条件 id=1 AND version=5 被执行后，id=1 的 version 被更新为 6，所以如果重复执行该条 SQL 语句将不生效，因为 id=1 AND version=5 的数据已经不存在，这样就能保住更新的幂等，多次更新对结果不会产生影响。

方案三：防重 Token 令牌

方案描述：

针对客户端连续点击或者调用方的超时重试等情况，例如提交订单，此种操作就可以用 Token 的机制实现防止重复提交。

简单的说就是调用方在调用接口的时候先向后端请求一个全局 ID (Token)，请求的时候携带这个全局 ID 一起请求 (Token 最好将其放到 Headers 中)，后端需要对这个 Token 作为 Key，用户信息作为 Value 到 Redis 中进行键值内容校验，如果 Key 存在且 Value 匹配就执行删除命令，然后正常执行后面的业务逻辑。如果不存在对应的 Key 或 Value 不匹配就返回重复执行的错误信息，这样来保证幂等操作。

适用操作：

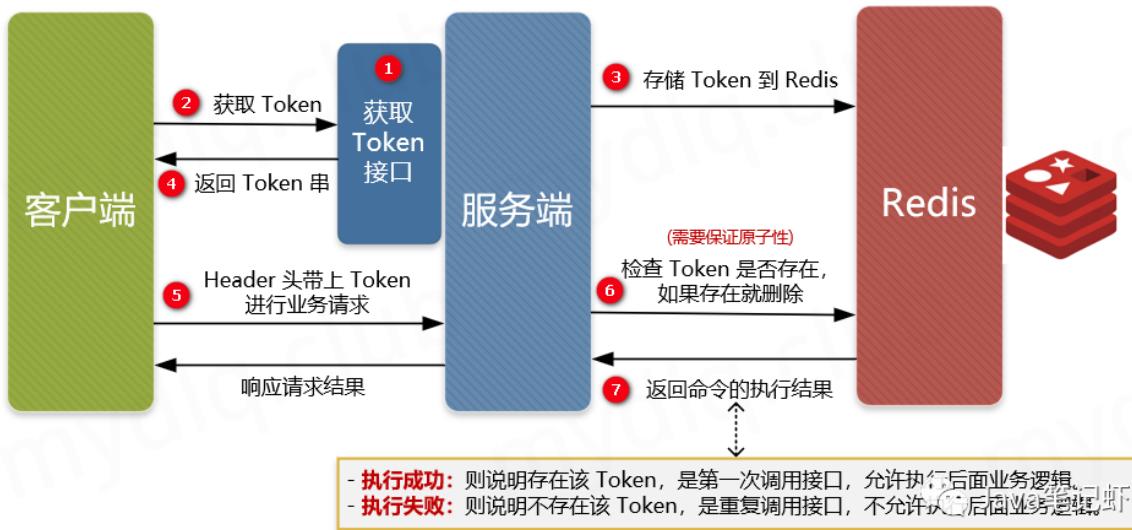
- 插入操作
- 更新操作
- 删除操作

使用限制：

- 需要生成全局唯一 Token 串；
- 需要使用第三方组件 Redis 进行数据效验；

主要流程：

Watermark



- ① 服务端提供获取 Token 的接口，该 Token 可以是一个序列号，也可以是一个分布式 ID 或者 UUID 串。
- ② 客户端调用接口获取 Token，这时候服务端会生成一个 Token 串。
- ③ 然后将该串存入 Redis 数据库中，以该 Token 作为 Redis 的键（注意设置过期时间）。
- ④ 将 Token 返回到客户端，客户端拿到后应存到表单隐藏域中。
- ⑤ 客户端在执行提交表单时，把 Token 存入到 Headers 中，执行业务请求带上该 Headers。
- ⑥ 服务端接收到请求后从 Headers 中拿到 Token，然后根据 Token 到 Redis 中查找该 key 是否存在。
- ⑦ 服务端根据 Redis 中是否存在该 key 进行判断，如果存在就将该 key 删除，然后正常执行业务逻辑。如果不存在就抛异常，返回重复提交的错误信息。

注意，在并发情况下，执行 Redis 查找数据与删除需要保证原子性，否则很可能在并发下无法保证幂等性。其实现方法可以使用分布式锁或者使用 Lua 表达式来注销查询与删除操作。

方案四、下游传递唯一序列号

方案描述:

所谓请求序列号，其实就是每次向服务端请求时候附带一个短时间内唯一不重复的序列号，该序列号可以是一个有序 ID，也可以是一个订单号，一般由下游生成，在调用上游服务端接口时附加该序列号和用于认证的 ID。

当上游服务器收到请求信息后拿取该 序列号 和下游 认证ID 进行组合，形成用于操作 Redis 的 Key，然后到 Redis 中查询是否存在对应的 Key 的键值对，根据其结果：

- 如果存在，就说明已经对该下游的该序列号的请求进行了业务处理，这时可以直接响应重复请求的错误信息，关注公众号猿技术专栏获取更多面试资源。
- 如果不存在，就以该 Key 作为 Redis 的键，以下游关键信息作为存储的值（例如下游商传递的一些业务逻辑信息），将该键值对存储到 Redis 中，然后再正常执行对应的业务逻辑即可。

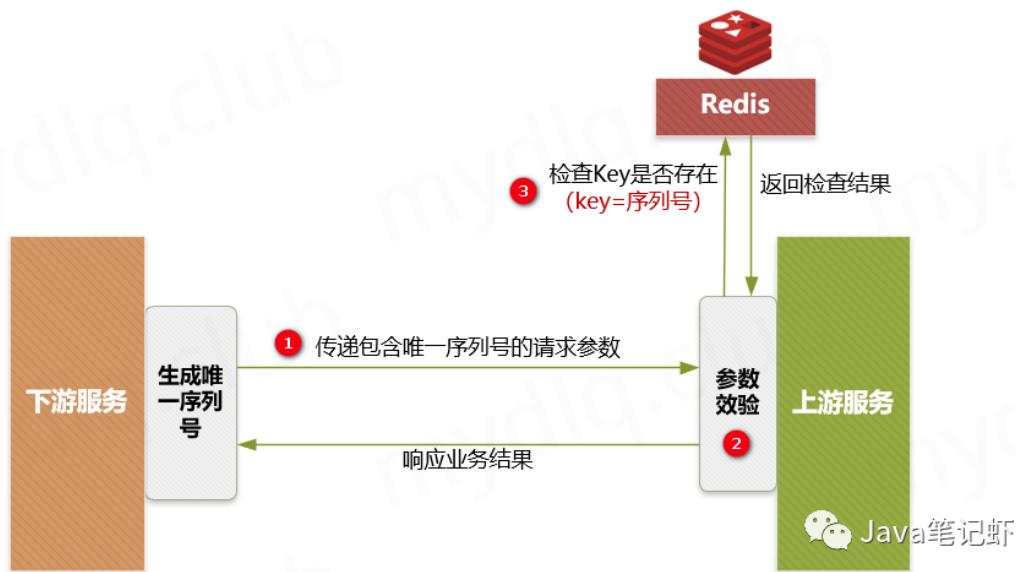
适用操作:

- 插入操作
- 更新操作
- 删除操作

使用限制:

- 要求第三方传递唯一序列号；
- 需要使用第三方组件 Redis 进行数据校验；

主要流程:



主要步骤:

- ① 下游服务生成分布式 ID 作为序列号，然后执行请求调用上游接口，并附带“唯一序列号”与请求的“认证凭据ID”。
- ② 上游服务进行安全效验，检测下游传递的参数中是否存在“序列号”和“凭据ID”。
- ③ 上游服务到 Redis 中检测是否存在对应的“序列号”与“认证ID”组成的 Key，如果存在就抛出重复执行的异常信息，然后响应下游对应的错误信息。如果不存在就以该“序列号”和“认证 ID”组合作为 Key，以下游关键信息作为 Value，进而存储到 Redis 中，然后正常执行接下来的业务逻辑。

上面步骤中插入数据到 Redis 一定要设置过期时间。这样能保证在这个时间范围内，如果重复调用接口，则能够进行判断识别。如果不设置过期时间，很可能导致数据无限量的存入 Redis，致使 Redis 不能正常工作。

七、实现接口幂等示例

这里使用防重 Token 令牌方案，该方案能保证在不同请求动作下的幂等性，实现逻辑可以看上面写的“防重 Token 令牌”方案，接下来写下实现这个逻辑的代码。

1、Maven 引入相关依赖

这里使用 Maven 工具管理依赖，这里在 pom.xml 中引入 SpringBoot、Redis、lombok 相关依赖。

```
<?xml version="1.0" encoding="UTF-8"?>
<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 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.4.RELEASE</version>
  </parent>
```

```

<groupId>mydlq.club</groupId>
<artifactId>springboot-idempotent-token</artifactId>
<version>0.0.1</version>
<name>springboot-idempotent-token</name>
<description>Idempotent Demo</description>

<properties>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <!--springboot web-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!--springboot data redis-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-pool2</artifactId>
    </dependency>
    <!--lombok-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

</project>

```

2、配置连接 Redis 的参数

在 application 配置文件中配置连接 Redis 的参数，如下：

```

spring:
  redis:
    ssl: false
    host: 127.0.0.1
    port: 6379
    database: 0
    timeout: 1000
    password:
    lettuce:
      pool:

```

```
max-active: 100  
max-wait: -1  
min-idle: 0  
max-idle: 20
```

3、创建与验证 Token 工具类

创建用于操作 Token 相关的 Service 类，里面存在 Token 创建与验证方法，其中：

- **Token 创建方法：** 使用 UUID 工具创建 Token 串，设置以 “idempotent_token:”+ “Token 串” 作为 Key，以用户信息当成 Value，将信息存入 Redis 中。
- **Token 验证方法：** 接收 Token 串参数，加上 Key 前缀形成 Key，再传入 value 值，执行 Lua 表达式（Lua 表达式能保证命令执行的原子性）进行查找对应 Key 与删除操作。执行完成后验证命令的返回结果，如果结果不为空且非0，则验证成功，否则失败。

```
import java.util.Arrays;  
import java.util.UUID;  
import java.util.concurrent.TimeUnit;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.redis.core.StringRedisTemplate;  
import org.springframework.data.redis.core.script.DefaultRedisScript;  
import org.springframework.data.redis.core.script.RedisScript;  
import org.springframework.stereotype.Service;  
  
@Slf4j  
@Service  
public class TokenUtilService {  
  
    @Autowired  
    private StringRedisTemplate redisTemplate;  
  
    /**  
     * 存入 Redis 的 Token 键的前缀  
     */  
    private static final String IDEMPOTENT_TOKEN_PREFIX = "idempotent_token:";  
  
    /**  
     * 创建 Token 存入 Redis，并返回该 Token  
     *  
     * @param value 用于辅助验证的 value 值  
     * @return 生成的 Token 串  
     */  
    public String generateToken(String value) {  
        // 实例化生成 ID 工具对象  
        String token = UUID.randomUUID().toString();  
        // 设置存入 Redis 的 Key  
        String key = IDEMPOTENT_TOKEN_PREFIX + token;  
        // 存储 Token 到 Redis，且设置过期时间为5分钟  
        redisTemplate.opsForValue().set(key, value, 5, TimeUnit.MINUTES);  
        // 返回 Token  
        return token;  
    }  
  
    /**  
     * 验证 Token 正确性  
     */
```

```

    * @param token token 字符串
    * @param value value 存储在Redis中的辅助验证信息
    * @return 验证结果
    */
    public boolean validToken(String token, String value) {
        // 设置 Lua 脚本, 其中 KEYS[1] 是 key, KEYS[2] 是 value
        String script = "if redis.call('get', KEYS[1]) == KEYS[2] then return redis.call('del', KEYS[1]) else return 0 end";
        RedisScript<Long> redisScript = new DefaultRedisScript<>(script, Long.class);
        // 根据 Key 前缀拼接 Key
        String key = IDEMPOTENT_TOKEN_PREFIX + token;
        // 执行 Lua 脚本
        Long result = redisTemplate.execute(redisScript, Arrays.asList(key, value));
        // 根据返回结果判断是否成功成功匹配并删除 Redis 键值对, 若果结果不为空和0, 则验证通过
        if (result != null && result != 0L) {
            log.info("验证 token={}，key={},value={} 成功", token, key, value);
            return true;
        }
        log.info("验证 token={}，key={},value={} 失败", token, key, value);
        return false;
    }
}

```

4、创建测试的 Controller 类

创建用于测试的 Controller 类, 里面有获取 Token 与测试接口幂等性的接口, 内容如下:

```

import lombok.extern.slf4j.Slf4j;
import myd1q.club.example.service.TokenUtilService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@Slf4j
@RestController
public class TokenController {

    @Autowired
    private TokenUtilService tokenService;

    /**
     * 获取 Token 接口
     *
     * @return Token 串
     */
    @GetMapping("/token")
    public String getToken() {
        // 获取用户信息 (这里使用模拟数据)
        // 注: 这里存储该内容只是举例, 其作用为辅助验证, 使其验证逻辑更安全, 如这里存储用户信息, 其
        // 目的为:
        // - 1) 使用"token"验证 Redis 中是否存在对应的 Key
        // - 2) 使用用户信息验证 Redis 的 Value 是否匹配。
        String userInfo = "myd1q";
        // 获取 Token 字符串, 并返回
        return tokenService.generateToken(userInfo);
    }
}

```

```

    /**
     * 接口幂等性测试接口
     *
     * @param token 幂等 Token 串
     * @return 执行结果
     */
    @PostMapping("/test")
    public String test(@RequestHeader(value = "token") String token) {
        // 获取用户信息（这里使用模拟数据）
        String userInfo = "mydlq";
        // 根据 Token 和与用户相关的信息到 Redis 验证是否存在对应的信息
        boolean result = tokenService.validToken(token, userInfo);
        // 根据验证结果响应不同信息
        return result ? "正常调用" : "重复调用";
    }
}

```

5、创建 SpringBoot 启动类

创建启动类，用于启动 SpringBoot 应用。

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

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

}

```

6、写测试类进行测试

写个测试类进行测试，多次访问同一个接口，测试是否只有第一次能否执行成功。

```

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcBuilders;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

@Slf4j
@SpringBootTest
@RunWith(SpringRunner.class)
public class IdempotenceTest {

    @Autowired
    
```

Watermark

```

    public class IdempotenceTest {

```

@Autowired

```

private WebApplicationContext webApplicationContext;

@Test
public void interfaceIdempotenceTest() throws Exception {
    // 初始化 MockMvc
    MockMvc mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    // 调用获取 Token 接口
    String token = mockMvc.perform(MockMvcRequestBuilders.get("/token")
        .accept(MediaType.TEXT_HTML))
        .andReturn()
        .getResponseBody().getContentAsString();
    log.info("获取的 Token 串: {}", token);
    // 循环调用 5 次进行测试
    for (int i = 1; i <= 5; i++) {
        log.info("第{}次调用测试接口", i);
        // 调用验证接口并打印结果
        String result = mockMvc.perform(MockMvcRequestBuilders.post("/test")
            .header("token", token)
            .accept(MediaType.TEXT_HTML))
            .andReturn().getResponseBody().getContentAsString();
        log.info(result);
        // 结果断言
        if (i == 0) {
            Assert.assertEquals(result, "正常调用");
        } else {
            Assert.assertEquals(result, "重复调用");
        }
    }
}

```

显示如下：

```

[main] IdempotenceTest: 获取的 Token 串: 980ea707-ce2e-456e-a059-0a03332110b4
[main] IdempotenceTest: 第1次调用测试接口
[main] IdempotenceTest: 正常调用
[main] IdempotenceTest: 第2次调用测试接口
[main] IdempotenceTest: 重复调用
[main] IdempotenceTest: 第3次调用测试接口
[main] IdempotenceTest: 重复调用
[main] IdempotenceTest: 第4次调用测试接口
[main] IdempotenceTest: 重复调用
[main] IdempotenceTest: 第5次调用测试接口
[main] IdempotenceTest: 重复调用

```

八、最后总结

幂等性是开发当中很常见也很重要的一个需求，尤其是支付、订单等与金钱挂钩的服务，保证接口幂等性尤其重要。在现实中，对于不同的业务场景我们需要灵活的选择幂等性的实现方式：

- 对于下单等存在唯一主键的，可以使用“唯一主键方案”的方式实现。
- 对于更新订单状态等相关的更新场景操作，使用“乐观锁方案”实现更为简单。
- 对于上下游这种，下游请求上游，上游服务可以使用“下游传递唯一序列号方案”更为合理。

- 类似于前端重复提交、重复下单、没有唯一ID号的场景，可以通过 Token 与 Redis 配合的“防重 Token 方案”实现更为快捷。

上面只是给出一些建议，再次强调一下，实现幂等性需要先理解自身业务需求，根据业务逻辑来实现这样才合理，处理好其中的每一个结点细节，完善整体的业务流程设计，才能更好的保证系统的正常运行。最后做一个简单总结，然后本博文到此结束，如下：

方案名称	适用方法	实现复杂度	方案缺点
数据库唯一主键	插入操作	简单	- 只能用于插入操作；
	删除操作		- 只能用于存在唯一主键场景；
数据库乐观锁	更新操作	简单	- 只能用于更新操作；
			- 表中需要额外添加字段；
请求序列号	插入操作	简单	- 需要保证下游生成唯一序列号；
	更新操作		- 需要 Redis 第三方存储已经请求的序列号；
	删除操作		
防重 Token 令牌	插入操作	适中	- 需要 Redis 第三方存储生成的 Token 令牌；
	更新操作		
	删除操作		

如何使用 Arthas 定位 Spring Boot 接口超时？

背景

公司有个渠道系统，专门对接三方渠道使用，没有什么业务逻辑，主要是转换报文和参数校验之类的工作，起着一个承上启下的作用。

最近在优化接口的响应时间，优化了代码之后，但是时间还是达不到要求；有一个诡异的100ms左右的耗时问题，在接口中打印了请求处理时间后，和调用方的响应时间还有差了100ms左右。比如程序里记录150ms，但是调用方等待时间却为250ms左右。

下面记录下当时详细的定位&解决流程（其实解决很简单，关键在于怎么定位并找到解决问题的方法）

定位过程

分析代码

渠道系统是一个常见的spring-boot web工程，使用了集成的tomcat。分析了代码之后，发现并没有特殊的地方，没有特殊的过滤器或者拦截器，所以初步排除是业务代码问题

Watermark

分析调用流程

出现这个问题之后，首先确认了下接口的调用流程。由于是内部测试，所以调用流程较少。关注公众号码猿技术专栏获取更多面试资源。

Nginx -> 反向代理 -> 渠道系统

公司是云服务器，网络走的也是云的内网。由于不明确问题的原因，所以用排除法，首先确认服务器网络是否有问题。

先确认发送端到Nginx Host是否有问题：

```
[jboss@VM_0_139_centos ~]$ ping 10.0.0.139
PING 10.0.0.139 (10.0.0.139) 56(84) bytes of data.
64 bytes from 10.0.0.139: icmp_seq=1 ttl=64 time=0.029 ms
64 bytes from 10.0.0.139: icmp_seq=2 ttl=64 time=0.041 ms
64 bytes from 10.0.0.139: icmp_seq=3 ttl=64 time=0.040 ms
64 bytes from 10.0.0.139: icmp_seq=4 ttl=64 time=0.040 ms
```

从ping结果上看，发送端到Nginx主机的延迟是无问题的，接下来查看Nginx到渠道系统的网络。

```
## 由于日志是没问题的，这里直接复制上面日志了
[jboss@VM_0_139_centos ~]$ ping 10.0.0.139
PING 10.0.0.139 (10.0.0.139) 56(84) bytes of data.
64 bytes from 10.0.0.139: icmp_seq=1 ttl=64 time=0.029 ms
64 bytes from 10.0.0.139: icmp_seq=2 ttl=64 time=0.041 ms
64 bytes from 10.0.0.139: icmp_seq=3 ttl=64 time=0.040 ms
64 bytes from 10.0.0.139: icmp_seq=4 ttl=64 time=0.040 ms
```

从ping结果上看，Nginx到渠道系统服务器网络延迟也是没问题的

既然网络看似没问题，那么可以继续排除法，砍掉Nginx，客户端直接再渠道系统的服务器上，通过回环地址（localhost）直连，避免经过网卡/dns，缩小问题范围看看能否复现（这个应用和地址是我后期模拟的，测试的是一个空接口）：

```
[jboss@VM_10_91_centos tmp]$ curl -w "@curl-time.txt" http://127.0.0.1:7744/send
success
http: 200
dns: 0.001s
redirect: 0.000s
time_connect: 0.001s
time_appconnect: 0.000s
time_pretransfer: 0.001s
time_starttransfer: 0.073s
size_download: 7bytes
speed_download: 95.000B/s
-----
time_total: 0.073s 请求总耗时
```

从curl日志上看，通过回环地址调用一个空接口耗时也有73ms。这就奇怪了，跳过了中间所有调用节点（包括过滤器&拦截器之类），直接请求应用一个空接口，都有73ms的耗时，再请求一次看看：

Watermark

```
[jboss@VM_10_91_centos tmp]$ curl -w "@curl-time.txt" http://127.0.0.1:7744/send
success
      http: 200
      dns: 0.001s
      redirect: 0.000s
      time_connect: 0.001s
      time_appconnect: 0.000s
      time_pretransfer: 0.001s
      time_starttransfer: 0.003s
      size_download: 7bytes
      speed_download: 2611.000B/s
      -----
      time_total: 0.003s
```

更奇怪的是，第二次请求耗时就正常了，变成了3ms。经查阅资料，linux curl是默认开启http keep-alive的。就算不开启keep-alive，每次重新handshake，也不至于需要70ms。关注公众号码猿技术专栏获取更多面试资源。

经过不断分析测试发现，连续请求的话时间就会很短，每次请求只需要几毫秒，但是如果隔一段时间再请求，就会花费70ms以上。

从这个现象猜想，可能是某些缓存机制导致的，连续请求因为有缓存，所以速度快，时间长缓存失效后导致时间长。

那么这个问题点到底在哪一层呢？tomcat层还是spring-webmvc呢？

光猜想定位不了问题，还是得实际测试一下，把渠道系统的代码放到本地ide里启动测试能否复现。在ide中启动后并不能复现问题，并没有70+ms的延迟问题。这下头疼了，本地无法复现，不能Debug，由于问题点不在业务代码，也不能通过加日志的方式来Debug。

这时候可以祭出神器Arthas了

Arthas分析问题

Arthas 是Alibaba开源的Java诊断工具，深受开发者喜爱。当你遇到以下类似问题而束手无策时，Arthas可以帮助你解决：

- 这个类从哪个 jar 包加载的？为什么会报各种类相关的 Exception？
- 我改的代码为什么没有执行到？难道是我没 commit？分支搞错了？
- 遇到问题无法在线上 debug，难道只能通过加日志再重新发布吗？
- 线上遇到某个用户的数据处理有问题，但线上同样无法 debug，线下无法重现！
- 是否有一个全局视角来查看系统的运行状况？
- 有什么办法可以监控到JVM的实时运行状态？

上面是Arthas的官方简介，这次我只需要用他的一个小功能trace。动态计算方法调用路径和时间，这样我就可以定位时间在哪个地方被消耗了。

- trace 方法内部调用路径，并输出方法路径上的每个节点上耗时
- trace 命令能主动搜索 class-pattern / method-pattern
- 对应的方法调用路径，渲染和统计整个调用链路上的所有性能开销和追踪调用链路。

有了神器，那么该追踪什么方法呢？由于我对Tomcat源码不是很熟，所以只能从spring mvc下手，先来trace一下spring mvc的入口：

```
[arthas@24851]$ trace org.springframework.web.servlet.DispatcherServlet *
Press Q or Ctrl+C to abort.
```

```
Affect(class=cnt:1 , method=cnt:44) cost in 508 ms.
`---ts=2019-09-14 21:07:44;thread_name=http-nio-7744-exec-
2;id=11;is_daemon=true;priority=5;TCCL=org.springframework.boot.web.embedded.tomcat.TomcatEmbeddedWebapp
ClassLoader@7c136917
    `---[2. 952142ms] org.springframework.web.servlet.DispatcherServlet:buildLocaleContext()

    `---ts=2019-09-14 21:07:44;thread_name=http-nio-7744-exec-
2;id=11;is_daemon=true;priority=5;TCCL=org.springframework.boot.web.embedded.tomcat.TomcatEmbeddedWebapp
ClassLoader@7c136917
        `---[18. 08903ms] org.springframework.web.servlet.DispatcherServlet:doService()
            +---[0. 041346ms] org.apache.commons.logging.Log:isDebugEnabled() ##889
            +---[0. 022398ms] org.springframework.web.util.WebUtils:isIncludeRequest() ##898
            +---[0. 014904ms]

        org.springframework.web.servlet.DispatcherServlet:getWebApplicationContext() ##910
            +---[1. 071879ms] javax.servlet.http.HttpServletRequest:setAttribute() ##910
            +---[0. 020977ms] javax.servlet.http.HttpServletRequest:setAttribute() ##911
            +---[0. 017073ms] javax.servlet.http.HttpServletRequest:setAttribute() ##912
            +---[0. 218277ms] org.springframework.web.servlet.DispatcherServlet:getThemeSource() ##913
                |   `---[0. 137568ms] org.springframework.web.servlet.DispatcherServlet:getThemeSource()
                |       `---[min=0. 00783ms, max=0. 014251ms, total=0. 022081ms, count=2]
            org.springframework.web.servlet.DispatcherServlet:getWebApplicationContext() ##782
                +---[0. 019363ms] javax.servlet.http.HttpServletRequest:setAttribute() ##913
                +---[0. 070694ms] org.springframework.web.servlet.FlashMapManager:retrieveAndUpdate() ##916
                +---[0. 01839ms] org.springframework.web.servlet.FlashMap:<init>() ##920
                +---[0. 016943ms] javax.servlet.http.HttpServletRequest:setAttribute() ##920
                +---[0. 015268ms] javax.servlet.http.HttpServletRequest:setAttribute() ##921
                +---[15. 050124ms] org.springframework.web.servlet.DispatcherServlet:doDispatch() ##925
                    |   `---[14. 943477ms] org.springframework.web.servlet.DispatcherServlet:doDispatch()
                    |       +---[0. 019135ms]

        org.springframework.web.context.request.async.WebAsyncUtils:getAsyncManager() ##953
            |   +---[2. 108373ms]

        org.springframework.web.servlet.DispatcherServlet:checkMultipart() ##960
            |   |   `---[2. 004436ms]

        org.springframework.web.servlet.DispatcherServlet:checkMultipart()
            |   |   `---[1. 890845ms]

        org.springframework.web.multipart.MultipartResolver:isMultipart() ##1117
            |   +---[2. 054361ms] org.springframework.web.servlet.DispatcherServlet:getHandler()
##964
            |   |   `---[1. 961963ms]

        org.springframework.web.servlet.DispatcherServlet:getHandler()
            |   |   +---[0. 02051ms] java.util.List:iterator() ##1183
            |   |   +---[min=0. 003805ms, max=0. 009641ms, total=0. 013446ms, count=2]

        java.util.Iterator:hasNext() ##1183
            |   |   +---[min=0. 003181ms, max=0. 009751ms, total=0. 012932ms, count=2]

        java.util.Iterator:next() ##1183
            |   |   +---[min=0. 005841ms, max=0. 015308ms, total=0. 021149ms, count=2]

        org.apache.commons.logging.Log:isTraceEnabled() ##1184
            |   |   `---[min=0. 474739ms, max=1. 19145ms, total=1. 666189ms, count=2]

        org.springframework.web.servlet.HandlerMapping:getHandler() ##1188
            |   +---[0. 013071ms]

        org.springframework.web.servlet.HandlerExecutionChain:getHandler() ##971
            |   +---[0. 079236ms]

        org.springframework.web.servlet.DispatcherServlet:HandlerAdapter:getHandlerAdapter() ##971
            |   |   `---[0. 280073ms]

        org.springframework.web.servlet.DispatcherServlet:getHandlerAdapter()
            |   |   +---[0. 004804ms] java.util.List:iterator() ##1224
            |   |   +---[0. 003668ms] java.util.Iterator:hasNext() ##1224
            |   |   +---[0. 003038ms] java.util.Iterator:next() ##1224
```

```

| | | +---[0.006451ms] org.apache.commons.logging.Log:isTraceEnabled()
##1225
| | | `---[0.012683ms]
org.springframework.web.servlet.HandlerAdapter:supports() ##1228
| | +---[0.012848ms] javax.servlet.http.HttpServletRequest:getMethod() ##974
| | +---[0.013132ms] java.lang.String>equals() ##975
| | +---[0.003025ms]
org.springframework.web.servlet.HandlerExecutionChain:getHandler() ##977
| | +---[0.008095ms] org.springframework.web.servlet.HandlerAdapter:getLastModified()
##977
| | +---[0.006596ms] org.apache.commons.logging.Log:isDebugEnabled() ##978
| | +---[0.018024ms] org.springframework.web.context.request.ServletWebRequest:<init>
() ##981
| | +---[0.017869ms]
org.springframework.web.context.request.ServletWebRequest:checkNotModified() ##981
| | +---[0.038542ms]
org.springframework.web.servlet.HandlerExecutionChain:applyPreHandle() ##986
| | +---[0.00431ms]
org.springframework.web.servlet.HandlerExecutionChain:getHandler() ##991
| | +---[4.248493ms] org.springframework.web.servlet.HandlerAdapter:handle() ##991
| | +---[0.014805ms]
org.springframework.web.context.request.async.WebAsyncManager:isConcurrentHandlingStarted() ##993
| | +---[1.444994ms]
org.springframework.web.servlet.DispatcherServlet:applyDefaultViewName() ##997
| | | `---[0.067631ms]
org.springframework.web.servlet.DispatcherServlet:applyDefaultViewName()
| | +---[0.012027ms]
org.springframework.web.servlet.HandlerExecutionChain:applyPostHandle() ##998
| | +---[0.373997ms]
org.springframework.web.servlet.DispatcherServlet:processDispatchResult() ##1008
| | | `---[0.197004ms]
org.springframework.web.servlet.DispatcherServlet:processDispatchResult()
| | | +---[0.007074ms] org.apache.commons.logging.Log:isDebugEnabled()
##1075
| | | +---[0.005467ms]
org.springframework.web.context.request.async.WebAsyncUtils:getAsyncManager() ##1081
| | | +---[0.004054ms]
org.springframework.web.context.request.async.WebAsyncManager:isConcurrentHandlingStarted() ##1081
| | | `---[0.011988ms]
org.springframework.web.servlet.HandlerExecutionChain:triggerAfterCompletion() ##1087
| | | `---[0.004015ms]
org.springframework.web.context.request.async.WebAsyncManager:isConcurrentHandlingStarted() ##1018
| | +---[0.005055ms]
org.springframework.web.context.request.async.WebAsyncUtils:getAsyncManager() ##928
`---[0.003422ms]
org.springframework.web.context.request.async.WebAsyncManager:isConcurrentHandlingStarted() ##928

```

~

Watermark

```
[jboss@VM_10_91_centos tmp]$ curl -w "@curl-time.txt" http://127.0.0.1:7744/send
success
      http: 200
      dns: 0.001s
      redirect: 0.000s
      time_connect: 0.001s
      time_appconnect: 0.000s
      time_pretransfer: 0.001s
      time_starttransfer: 0.115s
      size_download: 7bytes
      speed_download: 60.000B/s
      -----
      time_total: 0.115s
```

本次调用，调用端时间花费115ms，但是从arthas trace上看，spring mvc只消耗了18ms，那么剩下的97ms去哪了呢？

本地测试后已经可以排除spring mvc的问题了，最后也是唯一可能出问题的点就是tomcat

可是本人并不熟悉tomcat中的源码，就连请求入口都不清楚，tomcat里需要trace的类都不好找。。。

不过没关系，有神器Arthas，可以通过stack命令来反向查找调用路径，以

`org.springframework.web.servlet.DispatcherServlet`作为参数：

stack 输出当前方法被调用的调用路径

很多时候我们都只知道一个方法被执行，但这个方法被执行的路径非常多，或者你根本就不知道这个方法是从哪里被执行了，此时你需要的是 stack 命令。

```
[arthas@24851]$ stack org.springframework.web.servlet.DispatcherServlet *
Press Q or Ctrl+C to abort.
Affect(class=cnt:1 , method=cnt:44) cost in 495 ms.
ts=2019-09-14 21:15:19;thread_name=http-nio-7744-exec-5;id=14;is_daemon=true;priority=5;TCCL=org.springframework.boot.web.embedded.tomcat.TomcatEmbeddedWebappClassLoader@7c136917
@org.springframework.web.servlet.FrameworkServlet.processRequest()
  at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:866)
  at javax.servlet.http.HttpServlet.service(HttpServlet.java:635)
  at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:851)
  at javax.servlet.http.HttpServlet.service(HttpServlet.java:742)
  at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231)
  at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
  at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
  at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
  at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
  at
org.springframework.web.filter.RequestContextFilter.doFilterInternal(RequestContextFilter.java:99)
  at
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
  at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
  at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
  at
org.springframework.web.filter.HttpPutFormContentFilter.doFilterInternal(HttpPutFormContentFilter.java:109)
```

```
        at
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
        at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at
org.springframework.web.filter.HiddenHttpMethodFilter.doFilterInternal(HiddenHttpMethodFilter.java:81)
        at
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
        at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at
org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal(CharacterEncodingFilter.java:200)
)
        at
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
        at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:198)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:96)
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:496)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:140)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:81)
        at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:87)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:342)
        at org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:803)
        at org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)
        at org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:790)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1468)
        at org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:748)
```

ts=2019-09-14 21:15:19;thread_name=http-nio-7744-exec-
5;id=14;is_daemon=true;priority=5;TCL=org.springframework.boot.web.embedded.tomcat.TomcatEmbeddedWebapp
ClassLoader@7c136917

```
@org.springframework.web.servlet.DispatcherServlet.doService()
        at
org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:974)
        at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:866)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:635)
        at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:851)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:742)
        at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
        at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at
org.springframework.web.filter.RequestContextFilter.doFilterInternal(RequestContextFilter.java:99)
        at
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
```

Watermark

```
        at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at
org.springframework.web.filter.HttpPutFormContentFilter.doFilterInternal(HttpPutFormContentFilter.java:1
09)
        at
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
        at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at
org.springframework.web.filter.HiddenHttpMethodFilter.doFilterInternal(HiddenHttpMethodFilter.java:81)
        at
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
        at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at
org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal(CharacterEncodingFilter.java:200
)
        at
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
        at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at
org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:198)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:96)
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:496)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:140)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:81)
        at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:87)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:342)
        at org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:803)
        at org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)
        at org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:790)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1468)
        at org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:748)
```

从stack日志上可以很直观的看出DispatchServlet的调用栈，那么这么长的路径，该trace哪个类呢（这里跳过spring mvc中的过滤器的trace过程，实际排查的时候也trace了一遍，但这诡异的时间消耗不是由这里过滤器产生的）？

有一定经验的老司机从名字上大概也能猜出来从哪里下手比较好，那就是

`org.apache.coyote.http11.Http11Processor.service`，从名字上看，http1.1处理器，这可能是一个比较好的切入点。下面来trace一下：

```
[art,asg4.51] 14 trace org.apache.coyote.http11.Http11Processor service
Press Esc or Ctrl-C to abort.
Affect(class=cnt:1 , method=cnt:1) cost in 269 ms.
`---ts=2019-09-14 21:22:51;thread_name=http-nio-7744-exec-
8;id=17;is_daemon=true;priority=5;TCCL=org.springframework.boot.loader.LaunchedURLClassLoader@20ad9418
`---[131.650285ms] org.apache.coyote.http11.Http11Processor:service()
+---[0.036851ms] org.apache.coyote.Request:getRequestProcessor() ##667
```

```
+---[0.009986ms] org.apache.coyote.RequestInfo:setStage() ##668
+---[0.008928ms] org.apache.coyote.http11.Http11Processor:setSocketWrapper() ##671
+---[0.013236ms] org.apache.coyote.http11.Http11InputBuffer:init() ##672
+---[0.00981ms] org.apache.coyote.http11.Http11OutputBuffer:init() ##673
+---[min=0.00213ms, max=0.007317ms, total=0.009447ms, count=2]
org.apache.coyote.http11.Http11Processor:getErrorState() ##683
+---[min=0.002098ms, max=0.008888ms, total=0.010986ms, count=2]
org.apache.coyote.ErrorState:isError() ##683
+---[min=0.002448ms, max=0.007149ms, total=0.009597ms, count=2]
org.apache.coyote.http11.Http11Processor:isAsync() ##683
+---[min=0.002399ms, max=0.00852ms, total=0.010919ms, count=2]
org.apache.tomcat.util.net.AbstractEndpoint:isPaused() ##683
+---[min=0.033587ms, max=0.11832ms, total=0.151907ms, count=2]
org.apache.coyote.http11.Http11InputBuffer:parseRequestLine() ##687
+---[0.005384ms] org.apache.tomcat.util.net.AbstractEndpoint:isPaused() ##695
+---[0.007924ms] org.apache.coyote.Request:getMimeHeaders() ##702
+---[0.006744ms] org.apache.tomcat.util.net.AbstractEndpoint:getMaxHeaderCount() ##702
+---[0.012574ms] org.apache.tomcat.util.http.MimeHeaders:setLimit() ##702
+---[0.14319ms] org.apache.coyote.http11.Http11InputBuffer:parseHeaders() ##703
+---[0.003997ms] org.apache.coyote.Request:getMimeHeaders() ##743
+---[0.026561ms] org.apache.tomcat.util.http.MimeHeaders:values() ##743
+---[min=0.002869ms, max=0.01203ms, total=0.014899ms, count=2]
java.util.Enumeration:hasMoreElements() ##745
+---[0.070114ms] java.util.Enumeration:nextElement() ##746
+---[0.010921ms] java.lang.String:toLowerCase() ##746
+---[0.008453ms] java.lang.String:contains() ##746
+---[0.002698ms] org.apache.coyote.http11.Http11Processor:getErrorState() ##775
+---[0.00307ms] org.apache.coyote.ErrorState:isError() ##775
+---[0.002708ms] org.apache.coyote.RequestInfo:setStage() ##777
+---[0.171139ms] org.apache.coyote.http11.Http11Processor:prepareRequest() ##779
+---[0.009349ms] org.apache.tomcat.util.net.SocketWrapperBase:decrementKeepAlive() ##794
+---[0.002574ms] org.apache.coyote.http11.Http11Processor:getErrorState() ##800
+---[0.002696ms] org.apache.coyote.ErrorState:isError() ##800
+---[0.002499ms] org.apache.coyote.RequestInfo:setStage() ##802
+---[0.005641ms] org.apache.coyote.http11.Http11Processor:getAdapter() ##803
+---[129.868916ms] org.apache.coyote.Adapter:service() ##803
+---[0.003859ms] org.apache.coyote.http11.Http11Processor:getErrorState() ##809
+---[0.002365ms] org.apache.coyote.ErrorState:isError() ##809
+---[0.003844ms] org.apache.coyote.http11.Http11Processor:isAsync() ##809
+---[0.002382ms] org.apache.coyote.Response:getStatus() ##809
+---[0.002476ms] org.apache.coyote.http11.Http11Processor:statusDropsConnection() ##809
+---[0.002284ms] org.apache.coyote.RequestInfo:setStage() ##838
+---[0.00222ms] org.apache.coyote.http11.Http11Processor:isAsync() ##839
+---[0.037873ms] org.apache.coyote.http11.Http11Processor:endRequest() ##843
+---[0.002188ms] org.apache.coyote.RequestInfo:setStage() ##845
+---[0.002112ms] org.apache.coyote.http11.Http11Processor:getErrorState() ##849
+---[0.002063ms] org.apache.coyote.ErrorState:isError() ##849
+---[0.002504ms] org.apache.coyote.http11.Http11Processor:isAsync() ##853
+---[0.009808ms] org.apache.coyote.Request:updateCounters() ##854
+---[0.002008ms] org.apache.coyote.http11.Http11Processor:getErrorState() ##855
+---[0.002192ms] org.apache.coyote.ErrorState:isIoAllowed() ##855
+---[0.01008ms] org.apache.coyote.http11.Http11InputBuffer:nextRequest() ##856
+---[0.0065ms] org.apache.coyote.http11.Http11OutputBuffer:nextRequest() ##857
+---[0.002576ms] org.apache.coyote.RequestInfo:setStage() ##870
+---[0.016599ms] org.apache.coyote.http11.Http11Processor:processSendfile() ##872
+---[0.008182ms] org.apache.coyote.http11.Http11InputBuffer:getParsingRequestLinePhase()
##688
```

Watermark

```

+---[0.0075ms] org.apache.coyote.http11.Http11Processor:handleIncompleteRequestLineRead()
##690
+---[0.001979ms] org.apache.coyote.RequestInfo:setStage() ##875
+---[0.001981ms] org.apache.coyote.http11.Http11Processor:getErrorState() ##877
+---[0.001934ms] org.apache.coyote.ErrorState:isError() ##877
+---[0.001995ms] org.apache.tomcat.util.net.AbstractEndpoint:isPaused() ##877
+---[0.002403ms] org.apache.coyote.http11.Http11Processor:isAsync() ##879
`---[0.006176ms] org.apache.coyote.http11.Http11Processor:isUpgrade() ##881

```

日志里有一个129ms的耗时点（时间比没开arthas的时候更长是因为arthas本身带来的性能消耗，所以生产环境小心使用），这个就是要找的问题点。

打问题点找到了，那怎么定位是什么导致的问题呢，又如何解决呢？

继续trace吧，细化到具体的代码块或者内容。trace由于性能考虑，不会展示所有的调用路径，如果调用路径过深，只有手动深入trace，原则就是trace耗时长的那个方法：

```

[arthas@24851]$ trace org.apache.coyote.Adapter service
Press Q or Ctrl+C to abort.
Affect(class=cnt:1 , method=cnt:1) cost in 608 ms.
`---ts=2019-09-14 21:34:33;thread_name=http-nio-7744-exec-
1;id=10;is_daemon=true;priority=5;TCCL=org.springframework.boot.loader.LaunchedURLClassLoader@20ad9418
`---[81.70999ms] org.apache.catalina.connector.CoyoteAdapter:service()
+---[0.032546ms] org.apache.coyote.Request:getNote() ##302
+---[0.007148ms] org.apache.coyote.Response:getNote() ##303
+---[0.007475ms] org.apache.catalina.connector.Connector:getXpoweredBy() ##324
+---[0.00447ms] org.apache.coyote.Request:getRequestProcessor() ##331
+---[0.007902ms] java.lang.ThreadLocal:get() ##331
+---[0.006522ms] org.apache.coyote.RequestInfo:setWorkerThreadName() ##331
+---[73.793798ms] org.apache.catalina.connector.CoyoteAdapter:postParseRequest() ##336
+---[0.001536ms] org.apache.catalina.connector.Connector:getService() ##339
+---[0.004469ms] org.apache.catalina.Service:getContainer() ##339
+---[0.007074ms] org.apache.catalina.Engine:getPipeline() ##339
+---[0.004334ms] org.apache.catalina.Pipeline:isAsyncSupported() ##339
+---[0.002466ms] org.apache.catalina.connector.Request:setAsyncSupported() ##339
+---[6.01E-4ms] org.apache.catalina.connector.Connector:getService() ##342
+---[0.001859ms] org.apache.catalina.Service:getContainer() ##342
+---[9.65E-4ms] org.apache.catalina.Engine:getPipeline() ##342
+---[0.005231ms] org.apache.catalina.Pipeline:getFirst() ##342
+---[7.239154ms] org.apache.catalina.Valve:invoke() ##342
+---[0.006904ms] org.apache.catalina.connector.Request:isAsync() ##345
+---[0.00509ms] org.apache.catalina.connector.Request:finishRequest() ##372
+---[0.051461ms] org.apache.catalina.connector.Response:finishResponse() ##373
+---[0.007244ms] java.util.concurrent.atomic.AtomicBoolean:<init>() ##379
+---[0.007314ms] org.apache.coyote.Response:action() ##380
+---[0.004518ms] org.apache.catalina.connector.Request:isAsyncCompleting() ##382
+---[0.001072ms] org.apache.catalina.connector.Request:getContext() ##394
+---[0.007166ms] java.lang.System:currentTimeMillis() ##401
+---[0.004367ms] org.apache.coyote.Request:getStartTime() ##401
+---[0.011483ms] org.apache.catalina.Context:logAccess() ##401
+---[0.0014ms] org.apache.coyote.Request:getRequestProcessor() ##406
+---[min=8.0E-4ms,max=9.22E-4ms,total=0.001722ms,count=2] java.lang.Integer:<init>() ##406
`---[0.001851ms] java.lang.ThreadMethod:invoke() ##406
+---[0.001851ms] org.apache.coyote.RequestInfo:setWorkerThreadName() ##406
+---[0.035805ms] org.apache.catalina.connector.Request.recycle() ##410
`---[0.007849ms] org.apache.catalina.connector.Response.recycle() ##411

```

一段无聊的手动深入trace之后.....

```
[arthas@24851]$ trace org.apache.catalina.webresources.AbstractArchiveResourceSet getArchiveEntries
Press Q or Ctrl+C to abort.
Affect(class=cnt:4 , method=cnt:2) cost in 150 ms.
`---ts=2019-09-14 21:36:26;thread_name=http-nio-7744-exec-
3:id=12;is_daemon=true;priority=5;TCCL=org.springframework.boot.loader.LaunchedURLClassLoader@20ad9418
`---[75.743681ms] org.apache.catalina.webresources.JarWarResourceSet:getArchiveEntries()
    +---[0.025731ms] java.util.HashMap:<init>() ##106
    +---[0.097729ms] org.apache.catalina.webresources.JarWarResourceSet:openJarFile() ##109
    +---[0.091037ms] java.util.jar.JarFile:getJarEntry() ##110
    +---[0.096325ms] java.util.jar.JarFile:getInputStream() ##111
    +---[0.451916ms] org.apache.catalina.webresources.TomcatJarInputStream:<init>() ##113
    +---[min=0.001175ms, max=0.001176ms, total=0.002351ms, count=2] java.lang.Integer:<init>()
##114
    +---[0.00104ms] java.lang.reflect.Method:invoke() ##114
    +---[0.045105ms] org.apache.catalina.webresources.TomcatJarInputStream:getNextJarEntry()
##114
    +---[min=5.02E-4ms, max=0.008531ms, total=0.028864ms, count=31]
java.util.jar.JarEntry:getName() ##116
    +---[min=5.39E-4ms, max=0.022805ms, total=0.054647ms, count=31] java.util.HashMap:put() ##116
    +---[min=0.004452ms, max=34.479307ms, total=74.206249ms, count=31]
org.apache.catalina.webresources.TomcatJarInputStream:getNextJarEntry() ##117
    +---[0.018358ms] org.apache.catalina.webresources.TomcatJarInputStream:getManifest() ##119
    +---[0.006429ms] org.apache.catalina.webresources.JarWarResourceSet:setManifest() ##120
    +---[0.010904ms] org.apache.tomcat.util.compat.JreCompat:isJre9Available() ##121
    +---[0.003307ms] org.apache.catalina.webresources.TomcatJarInputStream:getMetaInfEntry()
##133
    +---[5.5E-4ms] java.util.jar.JarEntry:getName() ##135
    +---[6.42E-4ms] java.util.HashMap:put() ##135
    +---[0.001981ms] org.apache.catalina.webresources.TomcatJarInputStream:getManifestEntry()
##137
    +---[0.064484ms] org.apache.catalina.webresources.TomcatJarInputStream:close() ##141
    +---[0.007961ms] org.apache.catalina.webresources.JarWarResourceSet:closeJarFile() ##151
`---[0.004643ms] java.io.InputStream:close() ##155
```

发现了一个值得暂停思考的点：

```
+---[min=0.004452ms, max=34.479307ms, total=74.206249ms, count=31]
org.apache.catalina.webresources.TomcatJarInputStream:getNextJarEntry() ##117
```

这行代码加载了31次，一共耗时74ms；从名字上看，应该是tomcat加载jar包时的耗时，那么是加载了31个jar包的耗时，还是加载了jar包内的某些资源31次耗时呢？

TomcatJarInputStream这个类源码的注释写到：

```
The purpose of this sub-class is to obtain references to the JarEntry objects for META-INF/ and META-INF/MANIFEST.MF that are otherwise swallowed by the JarInputStream implementation.
```

大概意思也就是，获取jar包内META-INF/，META-INF/MANIFEST的资源，这是一个子类，更多的功能在父类JarInputStream里。

其实看到这里大概也能猜到问题了，tomcat加载jar包内META-INF/，META-INF/MANIFEST的资源导致的耗时。至于为什么连续请求不会耗时，应该是tomcat的缓存机制（下面介绍源码分析）

不着急定位问题，试着通过Arthas最细粒度问题细节，继续手动深入trace

```
[arthas@24851]$ trace org.apache.catalina.webresources.TomcatJarInputStream *
Press Q or Ctrl+C to abort.
Affect(class=cnt:1 , method=cnt:4) cost in 44 ms.
`---ts=2019-09-14 21:37:47;thread_name=http-nio-7744-exec-
5;id=14;is_daemon=true;priority=5;TCCL=org.springframework.boot.loader.LaunchedURLClassLoader@20ad9418
    `---[0.234952ms] org.apache.catalina.webresources.TomcatJarInputStream:createZipEntry()
        +---[0.039455ms] java.util.jar.JarInputStream:createZipEntry() ##43
            `---[0.007827ms] java.lang.String>equals() ##44

    `---ts=2019-09-14 21:37:47;thread_name=http-nio-7744-exec-
5;id=14;is_daemon=true;priority=5;TCCL=org.springframework.boot.loader.LaunchedURLClassLoader@20ad9418
        `---[0.050222ms] org.apache.catalina.webresources.TomcatJarInputStream:createZipEntry()
            +---[0.001889ms] java.util.jar.JarInputStream:createZipEntry() ##43
                `---[0.001643ms] java.lang.String>equals() ##44
##这里一共31个trace日志，删减了剩下的
```

从方法名上看，还是加载资源之类的意思。都已经到dk源码了，这时候来看一下 `TomcatJarInputStream` 这个类的源码：

```
/**
 * Creates a new <code>JarEntry</code> (<code>ZipEntry</code>) for the
 * specified JAR file entry name. The manifest attributes of
 * the specified JAR file entry name will be copied to the new
 * <CODE>JarEntry</CODE>.
 *
 * @param name the name of the JAR/ZIP file entry
 * @return the <code>JarEntry</code> object just created
 */
protected ZipEntry createZipEntry(String name) {
    JarEntry e = new JarEntry(name);
    if (man != null) {
        e.attr = man.getAttributes(name);
    }
    return e;
}
```

这个 `createZipEntry` 有个name参数，从注释上看，是jar/zip文件名，如果能得到文件名这种关键信息，就可以直接定位问题了；还是通过Arthas，使用watch命令，动态监测方法调用数据

watch方法执行数据观测

让你能方便的观察到指定方法的调用情况。能观察到的范围为：返回值、抛出异常、入参，通过编写 OGNL 表达式进行对应变量的查看。

watch 该方法的入参

```
[arthas@24851]$ watch org.apache.catalina.webresources.TomcatJarInputStream createZipEntry "
{params[0]}"
Press Q or Ctrl+C to abort.
Affect(class=cnt:1 , method=cnt:1) cost in 27 ms.
ts=2019-09-14 21:51:14; [cost=0.14547ms] result=@ArrayList[
    @String[META-INF/F],
]
ts=2019-09-14 21:51:14; [cost=0.048028ms] result=@ArrayList[
    @String[META-INF/MANIFEST.MF],
]
ts=2019-09-14 21:51:14; [cost=0.046071ms] result=@ArrayList[
```

```

    @String[META-INF/resources/],
]
ts=2019-09-14 21:51:14; [cost=0.033855ms] result=@ArrayList[
    @String[META-INF/resources/swagger-ui.html],
]
ts=2019-09-14 21:51:14; [cost=0.039138ms] result=@ArrayList[
    @String[META-INF/resources/webjars/],
]
ts=2019-09-14 21:51:14; [cost=0.033701ms] result=@ArrayList[
    @String[META-INF/resources/webjars/springfox-swagger-ui/],
]
ts=2019-09-14 21:51:14; [cost=0.033644ms] result=@ArrayList[
    @String[META-INF/resources/webjars/springfox-swagger-ui/favicon-16x16.png],
]
ts=2019-09-14 21:51:14; [cost=0.033976ms] result=@ArrayList[
    @String[META-INF/resources/webjars/springfox-swagger-ui/springfox.css],
]
ts=2019-09-14 21:51:14; [cost=0.032818ms] result=@ArrayList[
    @String[META-INF/resources/webjars/springfox-swagger-ui/swagger-ui-standalone-preset.js.map],
]
ts=2019-09-14 21:51:14; [cost=0.04651ms] result=@ArrayList[
    @String[META-INF/resources/webjars/springfox-swagger-ui/swagger-ui.css],
]
ts=2019-09-14 21:51:14; [cost=0.034793ms] result=@ArrayList[
    @String[META-INF/resources/webjars/springfox-swagger-ui/swagger-ui.js.map],
]

```

这下直接看到了具体加载的资源名，这么熟悉的名字：swagger-ui，一个国外的rest接口文档工具，又有国内开发者基于swagger-ui做了一套spring mvc的集成工具，通过注解就可以自动生成swagger-ui需要的接口定义json文件，用起来还比较方便，就是侵入性较强。

删除swagger的jar包后问题，诡异的70+ms就消失了

```

<!--pom 里删除这两个引用，这两个包时国内开发者封装的，swagger-ui并没有提供java spring-mvc的支持包，swagger只是一个浏览器端的ui+editor -->
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>

```

那么为什么swagger会导致请求耗时呢，为什么每次请求偶读会加载swagger内部的静态资源呢？

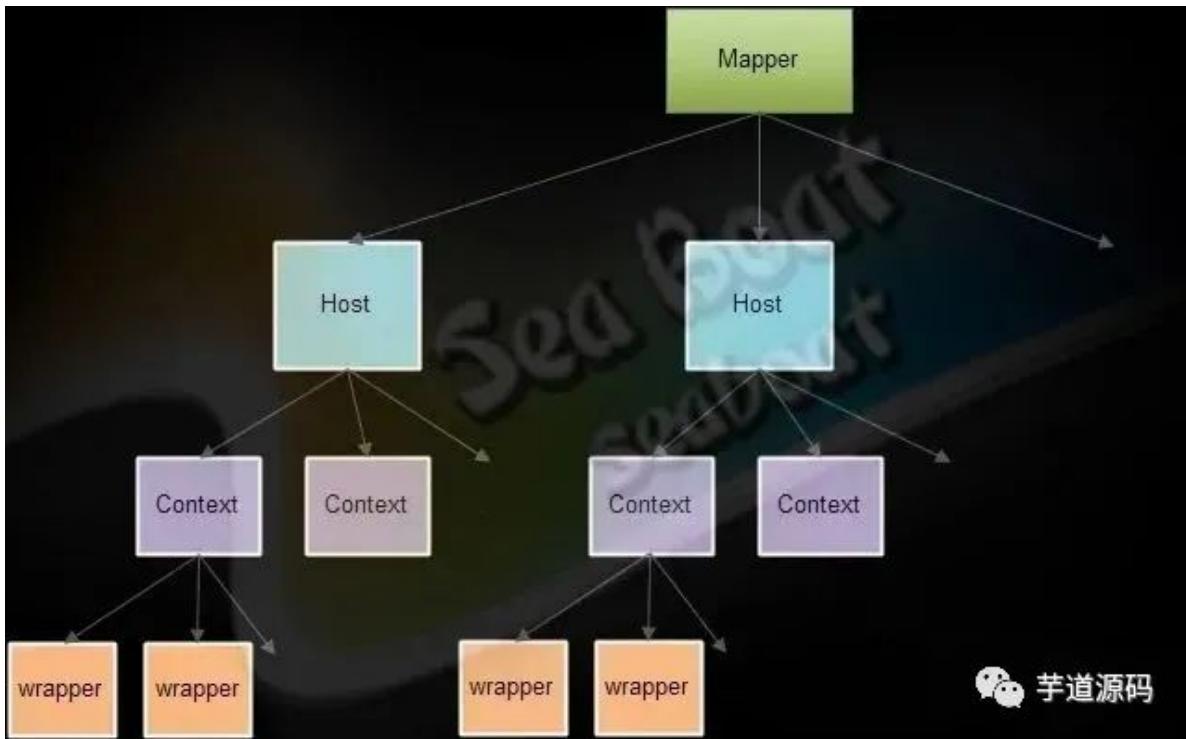
其实这是tomcat-embed的一个bug吧，下面详细介绍一下该Bug

Tomcat embed Bug分析&解决

Watermark

源码分析过程实在太漫长，而且也不是本文的重点，所以就不介绍了，下面直接介绍下分析结果

顺便贴一张tomcat处理请求的核心类图



为什么每次请求会加载Jar包内的静态资源

关键在于 `org.apache.catalina.mapper.Mapper##internalMapWrapper` 这个方法，该版本下处理请求的方式有问题，导致每次都校验静态资源。

为什么连续请求不会出现问题

因为Tomcat对于这种静态资源的解析是有缓存的，优先从缓存查找，缓存过期后再重新解析。具体参考 `org.apache.catalina.webresources.Cache`，默认过期时间ttl是5000ms。

为什么本地不会复现

其实确切的说，是通过spring-boot打包插件后不能复现。由于启动方式的不同，tomcat使用了不同的类去处理静态资源，所以没问题

如何解决

升级tomcat-embed版本即可

当前出现Bug的版本为：

`spring-boot:2.0.2.RELEASE`, 内置的tomcat embed版本为8.5.31

升级tomcat embed版本至8.5.40+即可解决此问题，新版本已经修复了

通过替换springboot pom properties方式

如果项目是maven是继承的springboot，即parent配置为springboot的，或者dependencyManagement中import spring boot包的

```

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.2.RELEASE</version>
  <relativePath/> 
</parent>
  
```

pom中直接覆盖properties即可：

```
<properties>
    <tomcat.version>8.5.40</tomcat.version>
</properties>
```

升级spring boot版本

springboot 2.1.0.RELEASE中的tomcat embed版本已经大于8.5.31了，所以直接将springboot升级至该版本及以上版本就可以解决此问题

在SpringBoot项目中，自定义注解+拦截器优雅的实现敏感数据的加解密！

在实际生产项目中，经常需要对如身份证信息、手机号、真实姓名等的敏感数据进行加密数据库存储，但在业务代码中对敏感信息进行手动加解密则十分不优雅，甚至会存在错加密、漏加密、业务人员需要知道实际的加密规则等的情况。

本文将介绍使用springboot+mybatis拦截器+自定义注解的形式对敏感数据进行存储前拦截加密的详细过程。

一、什么是Mybatis Plugin

在mybatis官方文档中，对于Mybatis plugin的的介绍是这样的：

MyBatis 允许你在已映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：

```
//语句执行拦截
Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)

// 参数获取、设置时进行拦截
ParameterHandler (getParameterObject, setParameters)

// 对返回结果进行拦截
ResultSetHandler (handleResultSets, handleOutputParameters)

//sql语句拦截
StatementHandler (prepare, parameterize, batch, update, query)
```

简而言之，即在执行sql的整个周期中，我们可以任意切入到某一点对sql的参数、sql执行结果集、sql语句本身等进行切面处理。基于这个特性，我们便可以使用其对我们需要进行加密的数据进行切面统一加密处理了（分页插件 pageHelper 就是这样实现数据库分页查询的）。

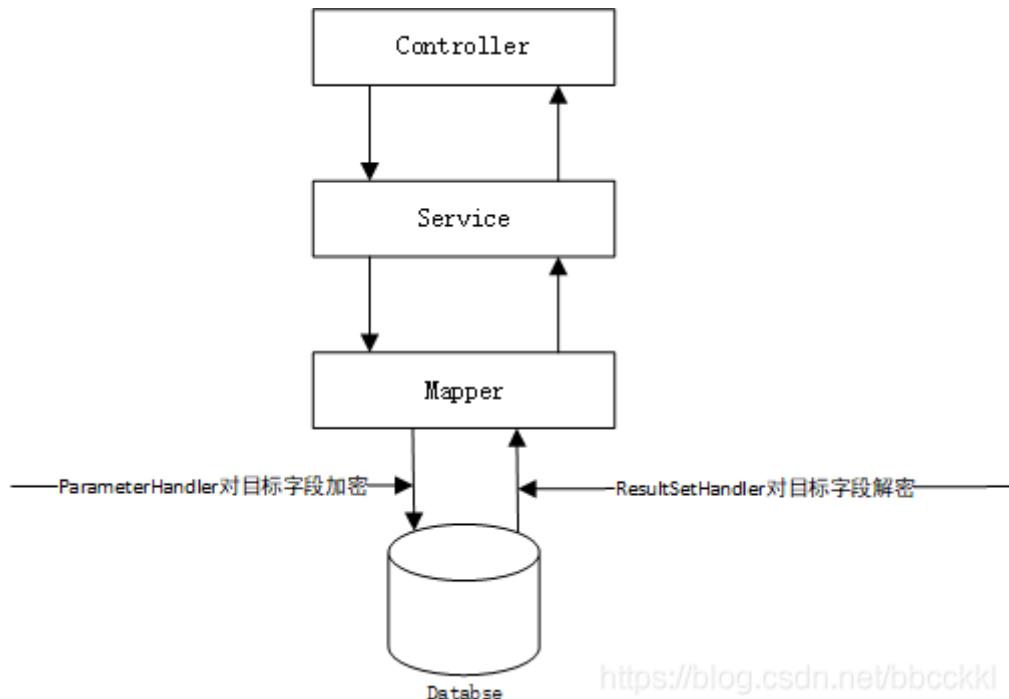
二、实现基于注解的敏感信息加解密拦截器

2.1 实现思路

对于数据的加密与解密，应当存在两个拦截器对数据进行拦截操作

参照官方文档，因此此处我们应当使用ParameterHandler拦截器对入参进行加密

使用ResultSetHandler拦截器对出参进行解密操作。



<https://blog.csdn.net/bbcockkl>

目标需要加密、解密的字段可能需要灵活变更，此时我们定义一个注解，对需要加密的字段进行注解，那么便可以配合拦截器对需要的数据进行加密与解密操作了。

mybatis的interceptor接口有以下方法需要实现。

```
public interface Interceptor {  
  
    //主要参数拦截方法  
    Object intercept(Invocation invocation) throws Throwable;  
  
    //mybatis插件链  
    default Object plugin(Object target) {return Plugin.wrap(target, this);}  
  
    //自定义插件配置文件方法  
    default void setProperties(Properties properties) {}  
  
}
```

2.2 定义需要加密解密的敏感信息注解

定义注解敏感信息类（如实体类POJO\PO）的注解

```
/**  
 * 注解敏感信息类的注解  
 */  
@Inherited  
@Target({ ElementType.TYPE })  
@Retention(RetentionPolicy.RUNTIME)  
public @interface SensitiveData {  
}
```

定义注解敏感信息类中敏感字段的注解

```
/**  
 * 注解敏感信息类中敏感字段的注解  
 */  
@Inherited  
@Target({ ElementType.Field })  
@Retention(RetentionPolicy.RUNTIME)  
public @interface SensitiveField {  
}
```

2.3 定义加密接口及其实现类

定义加密接口，方便以后拓展加密方法（如AES加密算法拓展支持PBE算法，只需要注入时指定一下便可）

```
public interface EncryptUtil {  
  
    /**  
     * 加密  
     *  
     * @param declaredFields paramsObject所声明的字段  
     * @param paramsObject    mapper中paramsType的实例  
     * @return T  
     * @throws IllegalAccessException 字段不可访问异常  
     */  
    <T> T encrypt(Field[] declaredFields, T paramsObject) throws IllegalAccessException;  
}
```

EncryptUtil 的AES加密实现类，此处AESUtil为自封装的AES加密工具，需要的小伙伴可以自行封装，本文不提供。

```
@Component  
public class AESEncrypt implements EncryptUtil {  
  
    @Autowired  
    AESUtil aesUtil;  
  
    /**  
     * 加密  
     *  
     * @param declaredFields paramsObject所声明的字段  
     * @param paramsObject    mapper中paramsType的实例  
     * @return T  
     * @throws IllegalAccessException 字段不可访问异常  
     */  
    @Override  
    public <T> T encrypt(Field[] declaredFields, T paramsObject) throws IllegalAccessException {  
        for (Field field : declaredFields) {  
            //取出所有被EncryptDecryptField注解的字段  
            SensitiveField sensitiveField = field.getAnnotation(SensitiveField.class);  
            if (sensitiveField != null) {  
                field.setAccessible(true);  
                Object object = field.get(paramsObject);  
                //暂时只实现String类型的加密  
                if (object instanceof String) {  
                    String value = (String) object;  
                }  
            }  
        }  
    }  
}
```

```

        //加密 这里我使用自定义的AES加密工具
        field.set(paramsObject, aesUtil.encrypt(value));
    }
}
return paramsObject;
}
}

```

2.4 实现入参加密拦截器

Mybatis包中的org.apache.ibatis.plugin.Interceptor拦截器接口要求我们实现以下三个方法

```

public interface Interceptor {

    //核心拦截逻辑
    Object intercept(Invocation invocation) throws Throwable;

    //拦截器链
    default Object plugin(Object target) {return Plugin.wrap(target, this);}

    //自定义配置文件操作
    default void setProperties(Properties properties) { }

}

```

因此，参考官方文档的示例，我们自定义一个入参加密拦截器。

@Intercepts 注解开启拦截器，@Signature 注解定义拦截器的实际类型。

@Signature中

- type 属性指定当前拦截器使用StatementHandler、ResultSetHandler、ParameterHandler、Executor的一种
- method 属性指定使用以上四种类型的具体方法（可进入class内部查看其方法）。
- args 属性指定预编译语句

此处我们使用了 ParameterHandler.setParamters()方法，拦截mapper.xml中paramsType的实例
(即在每个含有paramsType属性的mapper语句中，都执行该拦截器，对paramsType的实例进行拦截处理)

```

/**
 * 加密拦截器
 * 注意@Component注解一定要加上
 *
 * @author : tanzj
 * @date : 2020/1/19.
 */
@Slf4j
@Component
@Intercepts({
    @Signature(type = ParameterHandler.class, method = "setParameters", args =
    PreparedStatmentWrapper.class,
})
public class EncryptInterceptor implements Interceptor {

    private final EncryptDecryptUtil encryptUtil;

```

```

    @Autowired
    public EncryptInterceptor(EncryptDecryptUtil encryptUtil) {
        this.encryptUtil = encryptUtil;
    }

    @Override

    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        //在Signature 指定了 type= parameterHandler 后，这里的 invocation.getTarget() 便是
        parameterHandler
        //若指定ResultSetHandler，这里则能强转为ResultSetHandler
        ParameterHandler parameterHandler = (ParameterHandler) invocation.getTarget();
        // 获取参数对像，即 mapper 中 paramsType 的实例
        Field parameterField = parameterHandler.getClass().getDeclaredField("parameterObject");
        parameterField.setAccessible(true);
        //取出实例
        Object parameterObject = parameterField.get(parameterHandler);
        if (parameterObject != null) {
            Class<?> parameterObjectClass = parameterObject.getClass();
            //校验该实例的类是否被@SensitiveData所注解
            SensitiveData sensitiveData = AnnotationUtils.findAnnotation(parameterObjectClass,
                SensitiveData.class);
            if (Objects.nonNull(sensitiveData)) {
                //取出当前当前类所有字段，传入加密方法
                Field[] declaredFields = parameterObjectClass.getDeclaredFields();
                encryptUtil.encrypt(declaredFields, parameterObject);
            }
        }
        return invocation.proceed();
    }

    /**
     * 切记配置，否则当前拦截器不会加入拦截器链
     */
    @Override
    public Object plugin(Object o) {
        return Plugin.wrap(o, this);
    }

    //自定义配置写入，没有自定义配置的可以直接置空此方法
    @Override
    public void setProperties(Properties properties) {
    }
}

```

至此完成自定义加密拦截加密。

2.5 定义解密接口及其实现类

解密接口 其中result为mapper.xml中resultType的实例。

Watermark

```

public interface DecryptUtil {

    /**
     * 解密
     *
     * @param result resultType的实例
     * @return T
     * @throws IllegalAccessException 字段不可访问异常
     */
    <T> T decrypt(T result) throws IllegalAccessException;

}

```

解密接口AES工具解密实现类

```

public class AESDecrypt implements DecryptUtil {

    @Autowired
    AESUtil aesUtil;

    /**
     * 解密
     *
     * @param result resultType的实例
     * @return T
     * @throws IllegalAccessException 字段不可访问异常
     */
    @Override
    public <T> T decrypt(T result) throws IllegalAccessException {
        //取出resultType的类
        Class<?> resultClass = result.getClass();
        Field[] declaredFields = resultClass.getDeclaredFields();
        for (Field field : declaredFields) {
            //取出所有被EncryptDecryptField注解的字段
            SensitiveField sensitiveField = field.getAnnotation(SensitiveField.class);
            if (!Objects.isNull(sensitiveField)) {
                field.setAccessible(true);
                Object object = field.get(result);
                //只支持String的解密
                if (object instanceof String) {
                    String value = (String) object;
                    //对注解的字段进行逐一解密
                    field.set(result, aesUtil.decrypt(value));
                }
            }
        }
        return result;
    }
}

```

2.6 定义参数解密拦截器

```

@Slf4j
@Component
@Intercepts({
    @Signature(type = ResultSetHandler.class, method = "handleResultSets", args =
{Statement.class})
})

```

```

    })
}

public class DecryptInterceptor implements Interceptor {

    @Autowired
    DecryptUtil aesDecrypt;

    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        //取出查询的结果
        Object resultObject = invocation.proceed();
        if (Objects.isNull(resultObject)) {
            return null;
        }
        //基于selectList
        if (resultObject instanceof ArrayList) {
            ArrayList resultList = (ArrayList) resultObject;
            if (!CollectionUtils.isEmpty(resultList) && needToDecrypt(resultList.get(0))) {
                for (Object result : resultList) {
                    //逐一解密
                    aesDecrypt.decrypt(result);
                }
            }
        }
        //基于selectOne
    } else {
        if (needToDecrypt(resultObject)) {
            aesDecrypt.decrypt(resultObject);
        }
    }
    return resultObject;
}

private boolean needToDecrypt(Object object) {
    Class<?> objectClass = object.getClass();
    SensitiveData sensitiveData = AnnotationUtils.findAnnotation(objectClass,
SensitiveData.class);
    return Objects.nonNull(sensitiveData);
}

@Override
public Object plugin(Object target) {
    return Plugin.wrap(target, this);
}

@Override
public void setProperties(Properties properties) {
}
}

```

至此完成解密拦截器的配置工作。

Watermark

三、注解实体类中需要加解密的字段

```
 ① @Data  
 ② @SensitiveData  
 ③ @Accessors(chain = true)  
 public class User {  
 ④     /**  
      * 用户名  
 ⑤     */  
 ⑥     private String username;  
 ⑦     /**  
      * 身份证 (aes数据库加密)  
 ⑧     */  
 ⑨     @SensitiveField  
 ⑩     private String identityNo;  
 ⑪     /**  
      * 真实姓名 (aes数据库加密)  
 ⑫     */  
 ⑬     @SensitiveField  
 ⑭     private String realName;  
 ⑮     /**  
      * 手机号 (aes数据库加密)  
 ⑯     */  
 ⑰     @SensitiveField  
 ⑱     private String mobile;  
 }  
 https://blog.csdn.net/bbcckk1
```

此时在mapper中，指定paramType=User resultType=User 便可实现脱离业务层，基于mybatis拦截器的加解密操作。

SpringBoot 中实现跨域的5种方式

一、为什么会出现跨域问题

出于浏览器的同源策略限制。同源策略（Sameoriginpolicy）是一种约定，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，则浏览器的正常功能可能都会受到影响。可以说Web是构建在同源策略基础之上的，浏览器只是针对同源策略的一种实现。

同源策略会阻止一个域的javascript脚本和另外一个域的内容进行交互。所谓同源（即指在同一个域）就是两个页面具有相同的协议（protocol），主机（host）和端口号（port）

二、Watermark

当一个请求url的协议、域名、端口三者之间任意一个与当前页面url不同即为跨域

当前页面url	被请求页面url	是否跨域	原因
http://www.test.com/	http://www.test.com/index.html	否	同源（协议、域名、端口号相同）
http://www.test.com/	https://www.test.com/index.html	跨域	协议不同（http/https）
http://www.test.com/	http://www.baidu.com/	跨域	主域名不同（test/baidu）
http://www.test.com/	http://blog.test.com/	跨域	子域名不同（www/blog）
http://www.test.com:8080/	http://www.test.com:7001/	跨域	端口号不同（8080/7001）

三、非同源限制

【1】无法读取非同源网页的 Cookie、LocalStorage 和 IndexedDB

【2】无法接触非同源网页的 DOM

【3】无法向非同源地址发送 AJAX 请求

四、java 后端 实现 CORS 跨域请求的方式

对于 CORS的跨域请求，主要有以下几种方式可供选择：

1. 返回新的CorsFilter
2. 重写 WebMvcConfigurer
3. 使用注解 @CrossOrigin
4. 手动设置响应头 (HttpServletResponse)
5. 自定web filter 实现跨域

注意:

- CorFilter / WebMvConfigurer / @CrossOrigin 需要 SpringMVC 4.2以上版本才支持，对应 springBoot 1.3版本以上
- 上面前两种方式属于全局 CORS 配置，后两种属于局部 CORS配置。如果使用了局部跨域是会覆盖全局跨域的规则，所以可以通过 @CrossOrigin 注解来进行细粒度更高的跨域资源控制。
- 其实无论哪种方案，最终目的都是修改响应头，向响应头中添加浏览器所要求的数据，进而实现跨域

。

1.返回新的 CorsFilter(全局跨域)

在任意配置类，返回一个 新的 CorsFIltter Bean ， 并添加映射路径和具体的CORS配置路径。

```
@Configuration  
public class GlobalCorsConfig {  
    @Bean  
    public CorsFilter corsFilter() {  
        //添加CORS配置信息  
        CorsConfiguration config = new CorsConfiguration();  
        //放行哪些原始域  
        config.addAllowedOrigin("*");  
        //是否发送 Cookie  
        config.setAllowCredentials(true);  
        //放行哪些请求方式
```

```

        config.addAllowedMethod("*");
        //放行哪些原始请求头部信息
        config.addAllowedHeader("*");
        //暴露哪些头部信息
        config.addExposedHeader("*");
        //2. 添加映射路径
        UrlBasedCorsConfigurationSource corsConfigurationSource = new
        UrlBasedCorsConfigurationSource();
        corsConfigurationSource.registerCorsConfiguration("/**", config);
        //3. 返回新的CorsFilter
        return new CorsFilter(corsConfigurationSource);
    }
}

```

2. 重写 WebMvcConfigurer(全局跨域)

```

@Configuration
public class CorsConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
            //是否发送Cookie
            .allowCredentials(true)
            //放行哪些原始域
            .allowedOrigins("*")
            .allowedMethods(new String[]{"GET", "POST", "PUT", "DELETE"})
            .allowedHeaders("*")
            .exposedHeaders("*");
    }
}

```

3. 使用注解 (局部跨域)

在控制器(类上)上使用注解 `@CrossOrigin:`, 表示该类的所有方法允许跨域。

```

@RestController
@CrossOrigin(origins = "*")
public class HelloController {
    @RequestMapping("/hello")
    public String hello() {
        return "hello world";
    }
}

```

在方法上使用注解 `@CrossOrigin:`

```

@RequestMapping("/hello")
@CrossOrigin(origins = "*")
//@CrossOrigin(value = "http://localhost:8081") //指定具体ip允许跨域
public String hello() {
    return "hello world";
}

```

Watermark

4. 手动设置响应头(局部跨域)

使用 `HttpServletResponse` 对象添加响应头(`Access-Control-Allow-Origin`)来授权原始域，这里 `Origin`的值也可以设置为 “`*`” ,表示全部放行。

```
@RequestMapping("/index")
public String index(HttpServletRequest response) {
    response.setHeader("Access-Control-Allow-Origin", "*");
    return "index";
}
```

5. 使用自定义filter实现跨域

首先编写一个过滤器，可以起名字为`MyCorsFilter.java`

```
package com.mesnac.aop;

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.http.HttpServletResponse;
import org.springframework.stereotype.Component;

@Component
public class MyCorsFilter implements Filter {
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOException, ServletException {
        HttpServletResponse response = (HttpServletResponse) res;
        response.setHeader("Access-Control-Allow-Origin", "*");
        response.setHeader("Access-Control-Allow-Methods", "POST, GET, OPTIONS, DELETE");
        response.setHeader("Access-Control-Max-Age", "3600");
        response.setHeader("Access-Control-Allow-Headers", "x-requested-with, content-type");
        chain.doFilter(req, res);
    }
    public void init(FilterConfig filterConfig) {}
    public void destroy() {}
}
```

在`web.xml`中配置这个过滤器，使其生效

```
<!-- 跨域访问 START-->
<filter>
    <filter-name>CorsFilter</filter-name>
    <filter-class>com.mesnac.aop.MyCorsFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>CorsFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<!-- 跨域访问 END -->
```

五、参考连接

1. <https://blog.csdn.net/pjmike233/article/details/82461911>
2. <https://blog.csdn.net/zlbdmm/article/details/105853736>
3. <https://www.cnblogs.com/lenve/p/11724463.html>

ELK 处理 Spring Boot 日志，妙！

在排查线上异常的过程中，查询日志总是必不可缺的一部分。现今大多采用的微服务架构，日志被分散在不同的机器上，使得日志的查询变得异常困难。

工欲善其事，必先利其器。如果此时有一个统一的实时日志分析平台，那可谓是雪中送碳，必定能够提高我们排查线上问题的效率。本文带您了解一下开源的实时日志分析平台 ELK 的搭建及使用。

ELK 简介

ELK 是一个开源的实时日志分析平台，它主要由 Elasticsearch、Logstash 和 Kibana 三部分组成。

Logstash

Logstash 主要用于收集服务器日志，它是一个开源数据收集引擎，具有实时管道功能。Logstash 可以动态地将来自不同数据源的数据统一起来，并将数据标准化到您所选择的目的地。

Logstash 收集数据的过程主要分为以下三个部分：

- 输入：数据（包含但不限于日志）往往都是以不同的形式、格式存储在不同的系统中，而 Logstash 支持从多种数据源中收集数据（File、Syslog、MySQL、消息中间件等等）。
- 过滤器：实时解析和转换数据，识别已命名的字段以构建结构，并将它们转换成通用格式。
- 输出：Elasticsearch 并非存储的唯一选择，Logstash 提供很多输出选择。

Elasticsearch

Elasticsearch (ES) 是一个分布式的 Restful 风格的搜索和数据分析引擎，它具有以下特点：

- 查询：允许执行和合并多种类型的搜索 — 结构化、非结构化、地理位置、度量指标 — 搜索方式随心而变。
- 分析：Elasticsearch 聚合让您能够从大处着眼，探索数据的趋势和模式。
- 速度：很快，可以做到亿万级的数据，毫秒级返回。
- 可扩展性：可以在笔记本电脑上运行，也可以在承载了 PB 级数据的成百上千台服务器上运行。
- 弹性：运行在一个分布式的环境中，从设计之初就考虑到了这一点。
- 灵活性：具备多个案例场景。支持数字、文本、地理位置、结构化、非结构化，所有的数据类型都欢迎。

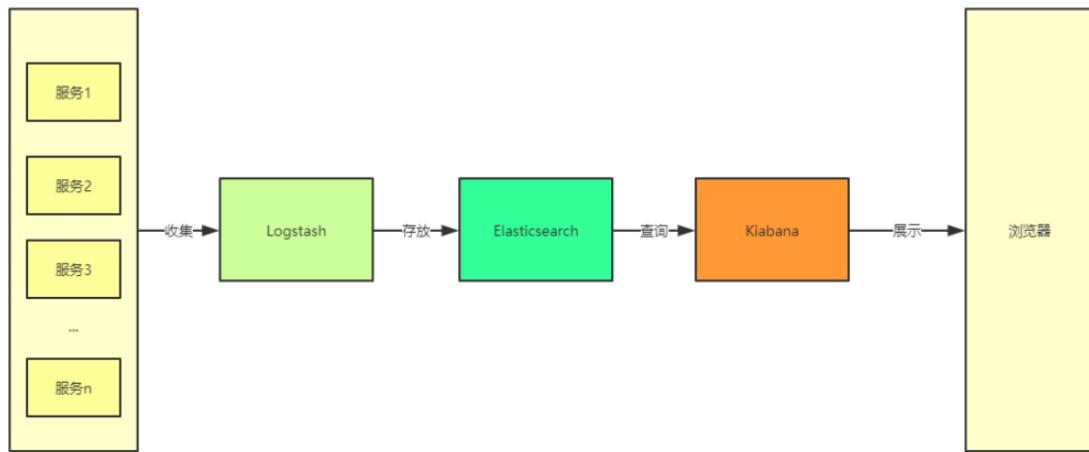
Kibana

Kibana 可以使海量数据通俗易懂。它很简单，基于浏览器的界面便于您快速创建和分享动态数据仪表板来追踪 Elasticsearch 的实时数据变化。其搭建过程也十分简单，您可以分分钟完成 Kibana 的安装并开始探索 Elasticsearch 的索引数据 — 没有代码、不需要额外的基础设施。另外，欢迎关注公众号码猿技术专栏，后台回复“9527”，送你一份Spring Cloud Alibaba实战视频！

对于以上三个组件在《ELK 协议栈介绍及体系结构》一文中有具体介绍，这里不再赘述。

在 ELK 中，三大组件的大概工作流程如下图所示，由 Logstash 从各个服务中采集日志并存放至 Elasticsearch 中，然后再由 Kibana 从 Elasticsearch 中查询日志并展示给终端用户。

图 1. ELK 的大致工作流程

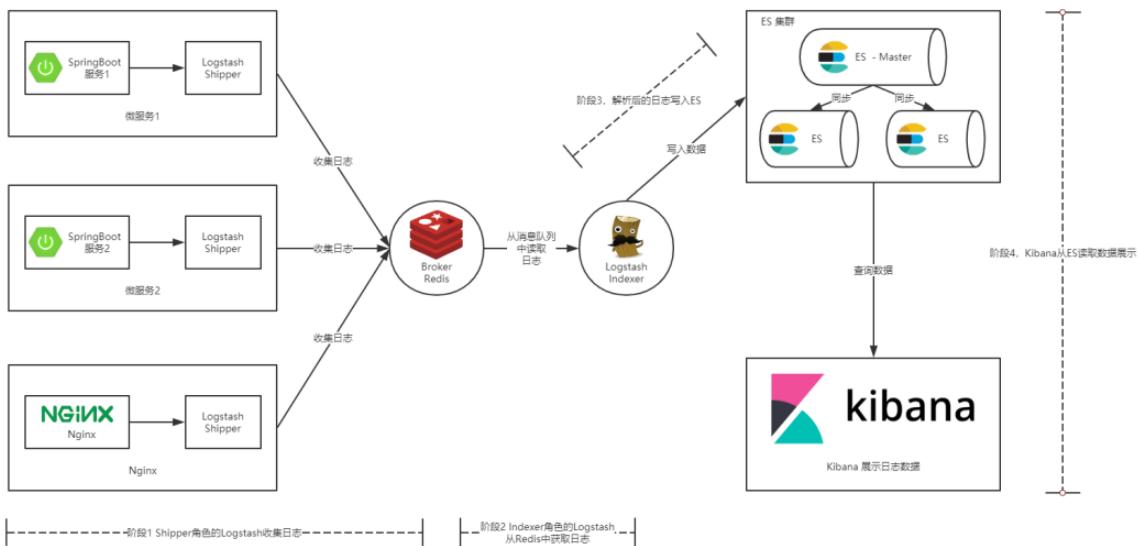


图片

ELK 实现方案

通常情况下我们的服务都部署在不同的服务器上，那么如何从多台服务器上收集日志信息就是一个关键点了。本篇文章中提供的解决方案如下图所示：

图 2. 本文提供的 ELK 实现方案



图片

如上图所示，整个 ELK 的运行流程如下：

1. 在微服务（产生日志的服务）上部署一个 Logstash，作为 Shipper 角色，主要负责对所在机器上的服务产生的日志文件进行数据采集，并将消息推送到 Redis 消息队列。
2. 另用一台服务器部署一个 Indexer 角色的 Logstash，主要负责从 Redis 消息队列中读取数据，并在 Logstash 管道中经过 Filter 的解析和处理后输出到 Elasticsearch 集群中存储。
3. Elasticsearch 主副节点之间数据同步。

4. 单独一台服务器部署 Kibana 读取 Elasticsearch 中的日志数据并展示在 Web 页面。

通过这张图，相信您已经大致清楚了我们将要搭建的 ELK 平台的工作流程，以及所需组件。下面就让我们一起开始搭建起来吧。

ELK 平台搭建

本节主要介绍搭建 ELK 日志平台，包括安装 Indexer 角色的 Logstash，Elasticsearch 以及 Kibana 三个组件。完成本小节，您需要做如下准备：

1. 一台 Ubuntu 机器或虚拟机，作为入门教程，此处省略了 Elasticsearch 集群的搭建，且将 Logstash(Indexer)、Elasticsearch 以及 Kibana 安装在同一机器上。
2. 在 Ubuntu 上安装 JDK，注意 Logstash 要求 JDK 在 1.7 版本以上。
3. Logstash、Elasticsearch、Kibana 安装包，您可以在此页面 [下载](#)。

安装 Logstash

解压压缩包：

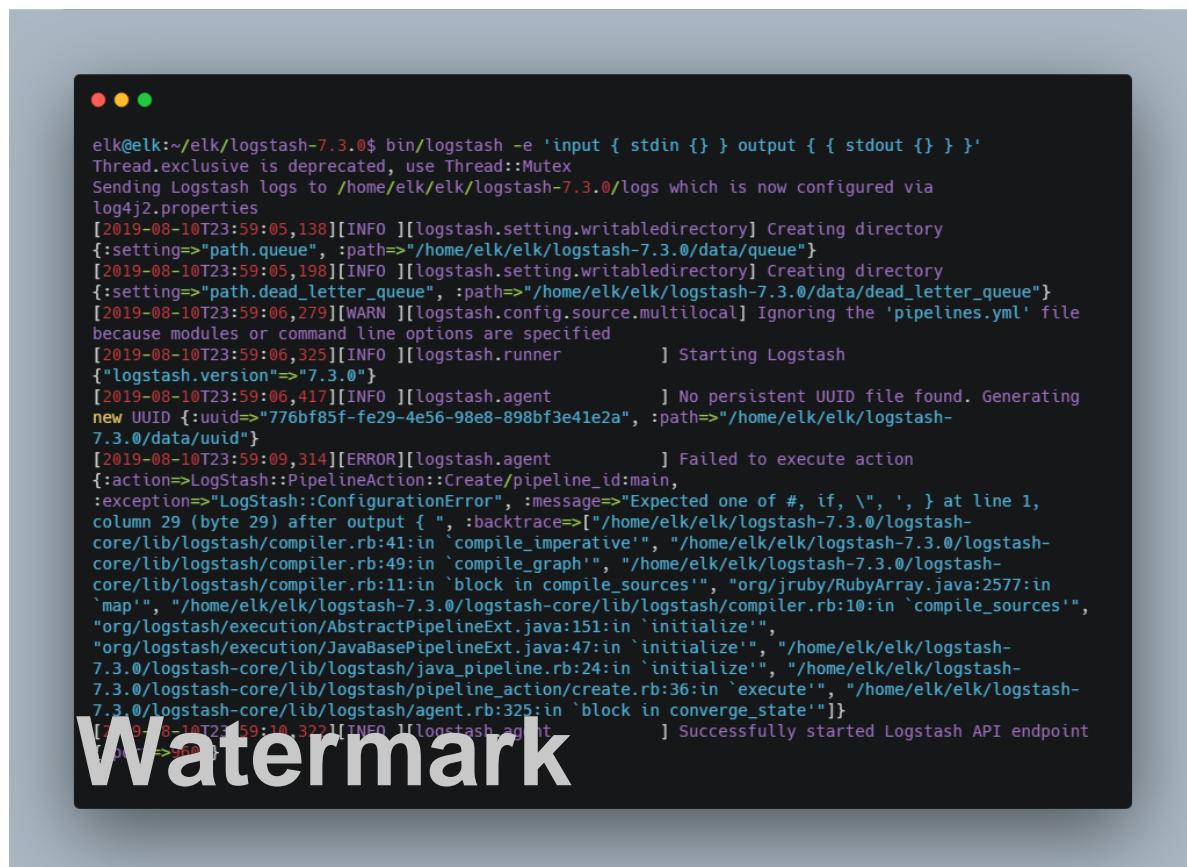
```
tar -xzvf logstash-7.3.0.tar.gz
```

显示更多简单用例测试，进入到解压目录，并启动一个将控制台输入输出到控制台的管道。

```
cd logstash-7.3.0
elk@elk:~/elk/logstash-7.3.0$ bin/logstash -e 'input { stdin {} } output { { stdout {} } }'
```

显示更多看到如下日志就意味着 Logstash 启动成功。

图 3. Logstash 启动成功日志



```
elk@elk:~/elk/logstash-7.3.0$ bin/logstash -e 'input { stdin {} } output { { stdout {} } }'
Thread.exclusive is deprecated, use Thread::Mutex
Sending Logstash logs to /home/elk/elk/logstash-7.3.0/logs which is now configured via
log4j2.properties
[2019-08-10T23:59:05,138][INFO ][logstash.setting.writabledirectory] Creating directory
{:setting=>"path.queue", :path=>"/home/elk/elk/logstash-7.3.0/data/queue"}
[2019-08-10T23:59:05,198][INFO ][logstash.setting.writabledirectory] Creating directory
{:setting=>"path.dead_letter_queue", :path=>"/home/elk/elk/logstash-7.3.0/data/dead_letter_queue"}
[2019-08-10T23:59:06,279][WARN ][logstash.config.source.multilocal] Ignoring the 'pipelines.yml' file
because modules or command line options are specified
[2019-08-10T23:59:06,325][INFO ][logstash.runner] Starting Logstash
{"logstash.version=>"7.3.0"}
[2019-08-10T23:59:06,417][INFO ][logstash.agent] No persistent UUID file found. Generating
new UUID {:uuid=>"776bf85f-fe29-4e56-98e8-898bf3e41e2a", :path=>"/home/elk/elk/logstash-
7.3.0/data/uuid"}
[2019-08-10T23:59:09,314][ERROR][logstash.agent] Failed to execute action
{:action=>LogStash::PipelineAction::Create/pipeline_id:main,
:exception=>"LogStash::ConfigurationError", :message=>"Expected one of #, if, \", ', } at line 1,
column 29 (byte 29) after output { ", :backtrace=>[~/home/elk/elk/logstash-7.3.0/logstash-
core/lib/logstash/compiler.rb:41:in `compile_imperative', ~/home/elk/elk/logstash-7.3.0/logstash-
core/lib/logstash/compiler.rb:49:in `compile_graph', ~/home/elk/elk/logstash-7.3.0/logstash-
core/lib/logstash/compiler.rb:11:in `block in compile_sources', org/jruby/RubyArray.java:2577:in
`map', ~/home/elk/elk/logstash-7.3.0/logstash-core/lib/logstash/compiler.rb:10:in `compile_sources',
org/logstash/execution/AbstractPipelineExt.java:151:in `initialize',
"org/logstash/execution/JavaBasePipelineExt.java:47:in `initialize'", ~/home/elk/elk/logstash-
7.3.0/logstash-core/lib/logstash/java_pipeline.rb:24:in `initialize', ~/home/elk/elk/logstash-
7.3.0/logstash-core/lib/logstash/pipeline_action/create.rb:36:in `execute', ~/home/elk/elk/logstash-
7.3.0/logstash-core/lib/logstash/agent.rb:325:in `block in converge_state"]}
[2019-08-10T23:59:10,322][INFO ][logstash.agent] Successfully started Logstash API endpoint
{:port=>9600}
```

图片

在控制台输入 Hello Logstash，看到如下效果代表 Logstash 安装成功。

清单 1. 验证 Logstash 是否启动成功Hello Logstash

```
{  
    "@timestamp" => "2019-08-10T16:11:10.040Z,  
    "host" => "elk",  
    "@version" => "1",  
    "message" => "Hello Logstash"  
}
```

安装 Elasticsearch

解压安装包：

```
tar -xvzf elasticsearch-7.3.0-linux-x86_64.tar.gz
```

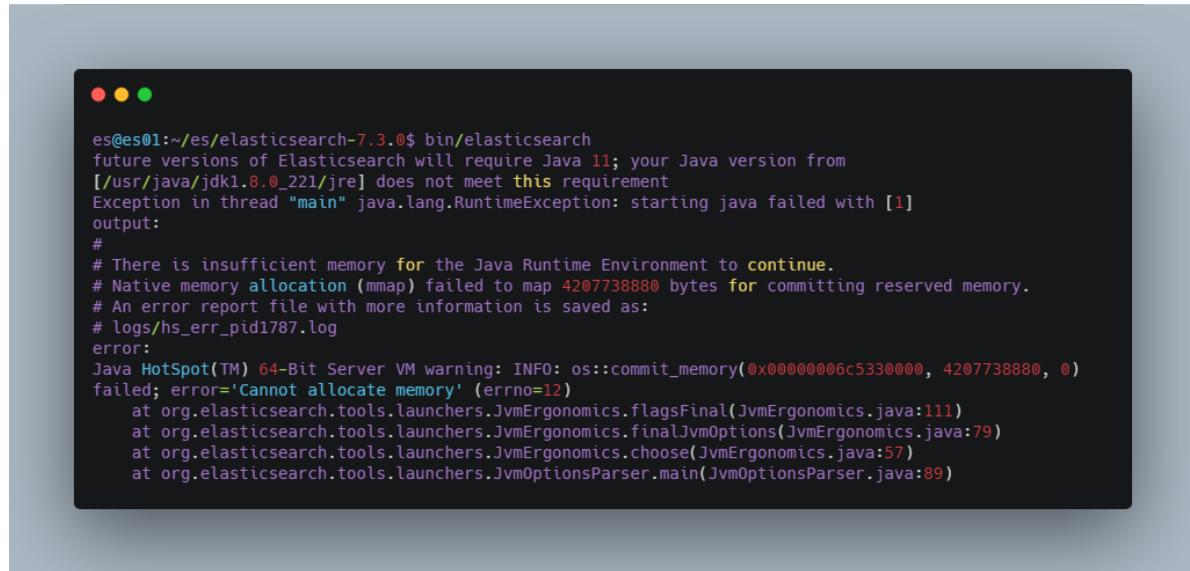
启动 Elasticsearch：

```
cd elasticsearch-7.3.0/  
bin/elasticsearch
```

在启动 Elasticsearch 的过程中我遇到了两个问题在这里列举一下，方便大家排查。

问题一：内存过小，如果您的机器内存小于 Elasticsearch 设置的值，就会报下图所示的错误。解决方案是，修改 elasticsearch-7.3.0/config/jvm.options 文件中的如下配置为适合自己机器的内存大小，若修改后还是报这个错误，可重新连接服务器再试一次。

图 4. 内存过小导致 Elasticsearch 启动报错

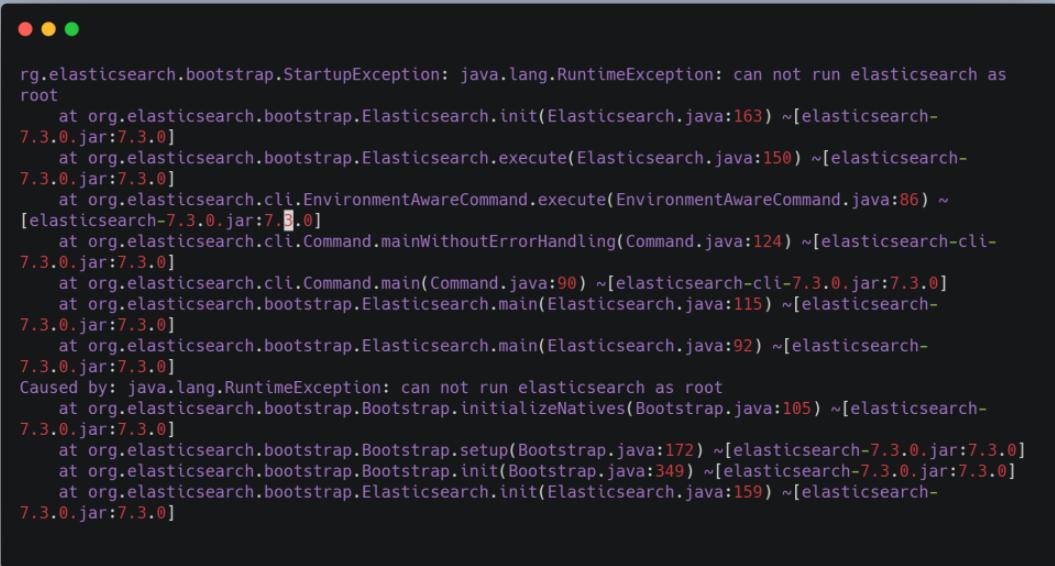


图片

问题二，如果您是以 root 用户启动的话，就会报下图所示的错误。解决方案自然就是添加一个新用户启动 Elasticsearch，至于添加新用户的方法网上有很多，这里就不再赘述。

图 5. Root 用户启动 Elasticsearch 报错

Watermark



```
rg.elasticsearch.bootstrap.StartupException: java.lang.RuntimeException: can not run elasticsearch as root
    at org.elasticsearch.bootstrap.Elasticsearch.init(Elasticsearch.java:163) ~[elasticsearch-7.3.0.jar:7.3.0]
    at org.elasticsearch.bootstrap.Elasticsearch.execute(Elasticsearch.java:150) ~[elasticsearch-7.3.0.jar:7.3.0]
    at org.elasticsearch.cli.EnvironmentAwareCommand.execute(EnvironmentAwareCommand.java:86) ~[elasticsearch-7.3.0.jar:7.3.0]
    at org.elasticsearch.cli.Command.mainWithoutErrorHandling(Command.java:124) ~[elasticsearch-cl-7.3.0.jar:7.3.0]
    at org.elasticsearch.cli.Command.main(Command.java:90) ~[elasticsearch-cl-7.3.0.jar:7.3.0]
    at org.elasticsearch.bootstrap.Elasticsearch.main(Elasticsearch.java:115) ~[elasticsearch-7.3.0.jar:7.3.0]
    at org.elasticsearch.bootstrap.Elasticsearch.main(Elasticsearch.java:92) ~[elasticsearch-7.3.0.jar:7.3.0]
Caused by: java.lang.RuntimeException: can not run elasticsearch as root
    at org.elasticsearch.bootstrap.Bootstrap.initializeNatives(Bootstrap.java:105) ~[elasticsearch-7.3.0.jar:7.3.0]
    at org.elasticsearch.bootstrap.Bootstrap.setup(Bootstrap.java:172) ~[elasticsearch-7.3.0.jar:7.3.0]
    at org.elasticsearch.bootstrap.Bootstrap.init(Bootstrap.java:349) ~[elasticsearch-7.3.0.jar:7.3.0]
    at org.elasticsearch.bootstrap.Elasticsearch.init(Elasticsearch.java:159) ~[elasticsearch-7.3.0.jar:7.3.0]
```

图片

启动成功后，另起一个会话窗口执行 curl <http://localhost:9200> 命令，如果出现如下结果，则代表 Elasticsearch 安装成功。

清单 2. 检查 Elasticsearch 是否启动成功

```
elk@elk:~$ curl http://localhost:9200
{
  "name" : "elk",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "hpq4Aad0T2Gcd4QyiHASmA",
  "version" : {
    "number" : "7.3.0",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "de777fa",
    "build_date" : "2019-07-24T18:30:11.767338Z",
    "build_snapshot" : false,
    "lucene_version" : "8.1.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta"
  },
  "tagline" : "You Know, for Search"
}
```

安装 Kibana

解压安装包：

```
tar -xzvf kibana-7.3.0-linux-x86_64.tar.gz
```

修改配置文件 config/kibana.yml，主要指定 Elasticsearch 的信息。

清单 3. Kibana 配置信息#Elasticsearch主机地址

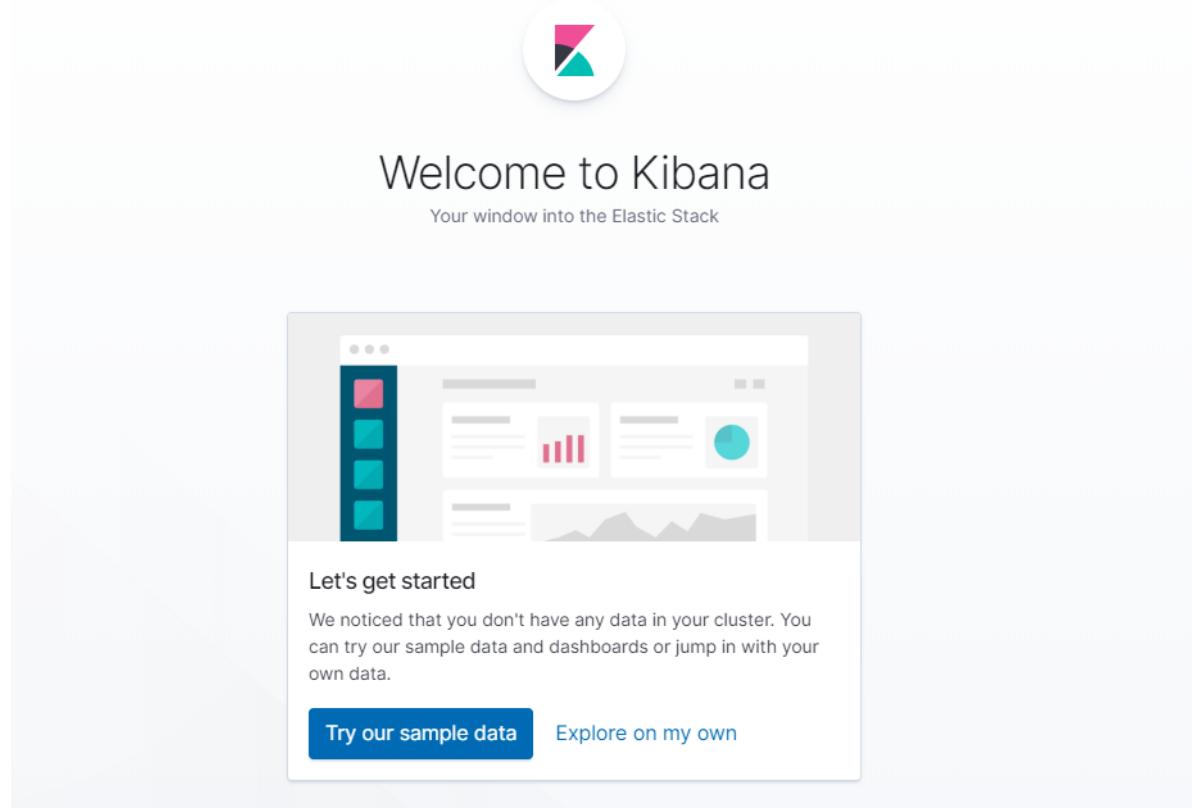
```
elasticsearch.hosts: "http://ip:9200"  
# 允许远程访问  
server.host: "0.0.0.0"  
# Elasticsearch用户名 这里其实就是我在服务器启动Elasticsearch的用户名  
elasticsearch.username: "es"  
# Elasticsearch鉴权密码 这里其实就是我在服务器启动Elasticsearch的密码  
elasticsearch.password: "es"
```

启动 Kibana:

```
cd kibana-7.3.0-linux-x86_64/bin  
./kibana
```

在浏览器中访问 <http://ip:5601>，若出现以下界面，则表示 Kibana 安装成功。

图 6. Kibana 启动成功界面



图片

ELK 日志平台安装完成后，下面我们就将通过具体的例子来看下如何使用 ELK，下文将分别介绍如何将 Spring Boot 日志和 Nginx 日志交由 ELK 分析。

在 Spring Boot 中使用 ELK

首先我们需要创建一个 Spring Boot 的项目，之前我写过一篇文章介绍 如何使用 AOP 来统一处理 Spring Boot 的 Web 日志，本文的 Spring Boot 项目就建立在这篇文章的基础之上。

Watermark

修改并部署 Spring Boot 项目

在项目 resources 目录下创建 spring-logback.xml 配置文件。

清单 4. Spring Boot 项目 Logback 的配置

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">
    <contextName>Logback For demo Mobile</contextName>
    <property name="LOG_HOME" value="/log" />
    <springProperty scope="context" name="appName" source="spring.application.name"
        defaultValue="localhost" />
    ...
    ...
    <appender name="ROLLING_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        ...
        <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{25} ${appName} -
            %msg%n</pattern>
        </encoder>
        ...
    </appender>
    ...
</configuration>
```

以上内容省略了很多内容，您可以在源码中获取。在上面的配置中我们定义了一个名为 ROLLING_FILE 的 Appender 往日志文件中输出指定格式的日志。而上面的 pattern 标签正是具体日志格式的配置，通过上面的配置，我们指定输出了时间、线程、日志级别、logger（通常为日志打印所在类的全路径）以及服务名称等信息。

将项目打包，并部署到一台 Ubuntu 服务器上。

清单 5. 打包并部署 Spring Boot 项目

```
# 打包命令
mvn package -Dmaven.test.skip=true
# 部署命令
java -jar sb-elk-start-0.0.1-SNAPSHOT.jar
```

查看日志文件，logback 配置文件中我将日志存放在 /log/sb-log.log 文件中，执行 more /log/sb-log.log 命令，出现以下结果表示部署成功。

图 7. Spring Boot 日志文件

Watermark



```
root@ubuntu:/log# more /log/sb-log.log
2019-08-11 11:18:56.995 [main] INFO c.i.s.SbElkStartApplication sb-elk -Starting SbElkStartApplication v0.0.1-SNAPSHOT on ubuntu with PID 2560 (/home/elk/sb-elk-start-0.0.1-SNAPSHOT.jar started by root in /home/elk)
2019-08-11 11:18:57.028 [main] INFO c.i.s.SbElkStartApplication sb-elk -No active profile set, falling back to default profiles: default
2019-08-11 11:19:01.453 [main] INFO o.s.d.r.c.RepositoryConfigurationDelegate sb-elk -Multiple Spring Data modules found, entering strict repository configuration mode!
2019-08-11 11:19:01.479 [main] INFO o.s.d.r.c.RepositoryConfigurationDelegate sb-elk -Bootstrapping Spring Data repositories in DEFAULT mode.
2019-08-11 11:19:01.583 [main] INFO o.s.d.r.c.RepositoryConfigurationDelegate sb-elk -Finished Spring Data repository scanning in 46ms. Found 0 repository interfaces.
2019-08-11 11:19:02.838 [main] INFO o.s.c.s.PostProcessorRegistrationDelegate$BeanPostProcessorChecker sb-elk -Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration' of type [org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration$$EnhancerBySpringCGLIB$$66efff60] is not eligible for getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)
2019-08-11 11:19:04.491 [main] INFO o.s.b.w.e.t.TomcatWebServer sb-elk -Tomcat initialized with port(s): 8080 (http)
2019-08-11 11:19:04.575 [main] INFO o.a.c.h.Http11NioProtocol sb-elk -Initializing ProtocolHandler ["http-nio-8080"]
2019-08-11 11:19:04.642 [main] INFO o.a.c.c.StandardService sb-elk -Starting service [Tomcat]
2019-08-11 11:19:04.646 [main] INFO o.a.c.core.StandardEngine sb-elk -Starting Servlet engine: [Apache Tomcat/9.0.22]
2019-08-11 11:19:04.935 [main] INFO o.a.c.c.C.[.[]] sb-elk -Initializing Spring embedded WebApplicationContext
2019-08-11 11:19:04.937 [main] INFO o.s.w.c.ContextLoader sb-elk -Root WebApplicationContext: initialization completed in 7581 ms
2019-08-11 11:19:06.833 [main] INFO c.a.d.p.DruidDataSource sb-elk -{dataSource-1} initited
2019-08-11 11:19:09.786 [main] INFO s.d.s.w.PropertySourcedRequestMappingHandlerMapping sb-elk -Mapped URL path [/v2/api-docs] onto method [public org.springframework.http.ResponseEntity<springfox.documentation.spring.web.json.Json> springfox.documentation.swagger2.web.Swagger2Controller.getDocumentation(java.lang.String, javax.servlet.http.HttpServletRequest)]
2019-08-11 11:19:10.739 [main] INFO o.s.s.c.ThreadPoolTaskExecutor sb-elk -Initializing ExecutorService 'applicationTaskExecutor'
2019-08-11 11:19:12.489 [main] INFO s.d.s.w.p.DocumentationPluginsBootstrapper sb-elk -Context refreshed
2019-08-11 11:19:12.579 [main] INFO s.d.s.w.p.DocumentationPluginsBootstrapper sb-elk -Found 1 custom documentation plugin(s)
2019-08-11 11:19:12.677 [main] INFO s.d.s.w.s.ApiListingReferenceScanner sb-elk -Scanning for api listing references
2019-08-11 11:19:13.109 [main] INFO o.a.c.h.Http11NioProtocol sb-elk -Starting ProtocolHandler ["http-nio-8080"]
2019-08-11 11:19:13.200 [main] INFO o.s.b.w.e.t.TomcatWebServer sb-elk -Tomcat started on port(s): 8080 (http) with context path ''
2019-08-11 11:19:13.215 [main] INFO c.i.s.SbElkStartApplication sb-elk -Started SbElkStartApplication in 18.542 seconds (JVM running for 20.337)
```

图片

配置 Shipper 角色 Logstash

Spring Boot 项目部署成功之后，我们还需要在当前部署的机器上安装并配置 Shipper 角色的 Logstash。Logstash 的安装过程在 ELK 平台搭建小节中已有提到，这里不再赘述。

安装完成后，我们需要编写 Logstash 的配置文件，以支持从日志文件中收集日志并输出到 Redis 消息管道中，Shipper 的配置如下所示。

清单 6. Shipper 角色的 Logstash 的配置

```
input {
    file {
        path => [
            # 这里填写需要监控的文件
            "/log/sb-log.log"
        ]
    }
}

output {
    # 输出到redis
```

Watermark

```

redis {
    host => "10.140.45.190"      # redis主机地址
    port => 6379                  # redis端口号
    db => 8                      # redis数据库编号
    data_type => "channel"        # 使用发布/订阅模式
    key => "logstash_list_0"     # 发布通道名称
}
}

```

其实 Logstash 的配置是与前面提到的 Logstash 管道中的三个部分（输入、过滤器、输出）一一对应的，只不过这里我们不需要过滤器所以就没有写出来。上面配置中 Input 使用的数据源是文件类型的，只需要配置上需要收集的本机日志文件路径即可。Output 描述数据如何输出，这里配置的是输出到 Redis。

Redis 的配置 data_type 可选值有 channel 和 list 两个。channel 是 Redis 的发布/订阅通信模式，而 list 是 Redis 的队列数据结构，两者都可以用来实现系统间有序的消息异步通信。

channel 相比 list 的好处是，解除了发布者和订阅者之间的耦合。举个例子，一个 Indexer 在持续读取 Redis 中的记录，现在想加入第二个 Indexer，如果使用 list，就会出现上一条记录被第一个 Indexer 取走，而下一条记录被第二个 Indexer 取走的情况，两个 Indexer 之间产生了竞争，导致任何一方都没有读到完整的日志。

channel 就可以避免这种情况。这里 Shipper 角色的配置文件和下面将要提到的 Indexer 角色的配置文件中都使用了 channel。

配置 Indexer 角色 Logstash

配置好 Shipper 角色的 Logstash 后，我们还需要配置 Indexer 角色 Logstash 以支持从 Redis 接收日志数据，并通过过滤器解析后存储到 Elasticsearch 中，其配置内容如下所示。

清单 7. Indexer 角色的 Logstash 的配置

```

input {
    redis {
        host      => "192.168.142.131"      # redis主机地址
        port      => 6379                  # redis端口号
        db        => 8                      # redis数据库编号
        data_type => "channel"        # 使用发布/订阅模式
        key       => "sb-logback"    # 发布通道名称
    }
}

filter {
    #定义数据的格式
    grok {
        match => { "message" => "%{TIMESTAMP_ISO8601:time} \[%{NOTSPACE:threadName}\] %{LOGLEVEL:level} %{DATA:logger} %{NOTSPACE:applicationName} -(?:.*=%{NUMBER:timetaken}ms|)%" }
    }
}

output {
    stdout {}
    elasticsearch {
        hosts => "localhost:9200"
        index => "logback"
    }
}

```

与 Shipper 不同的是，Indexer 的管道中我们定义了过滤器，也正是在这里将日志解析成结构化的数据。下面是我截取的一条 logback 的日志内容：

清单 8. Spring Boot 项目输出的一条日志

```
2019-08-11 18:01:31.602 [http-nio-8080-exec-2] INFO c.i.s.aop.WebLogAspect sb-elk -接口日志  
POST请求测试接口结束调用:耗时=11ms, result=BaseResponse {code=10000, message='操作成功'}
```

在 Filter 中我们使用 Grok 插件从上面这条日志中解析出了时间、线程名称、Logger、服务名称以及接口耗时几个字段。Grok 又是如何工作的呢？

1. message 字段是 Logstash 存放收集到的数据的字段，`match = {"message" => ...}` 代表是对日志内容做处理。
2. Grok 实际上也是通过正则表达式来解析数据的，上面出现的 `TIMESTAMP_ISO8601`、`NOTSPACE` 等都是 Grok 内置的 patterns。
3. 我们编写的解析字符串可以使用 Grok Debugger 来测试是否正确，这样避免了重复在真实环境中校验解析规则的正确性。

查看效果

经过上面的步骤，我们已经完成了整个 ELK 平台的搭建以及 Spring Boot 项目的接入。下面我们按照以下步骤执行一些操作来看下效果。

启动 Elasticsearch，启动命令在 ELK 平台搭建 小节中有提到，这里不赘述（Kibana 启动同）。启动 Indexer 角色的 Logstash。

```
# 进入到 Logstash 的解压目录，然后执行下面的命令  
bin/logstash -f indexer-logstash.conf
```

启动 Kibana。

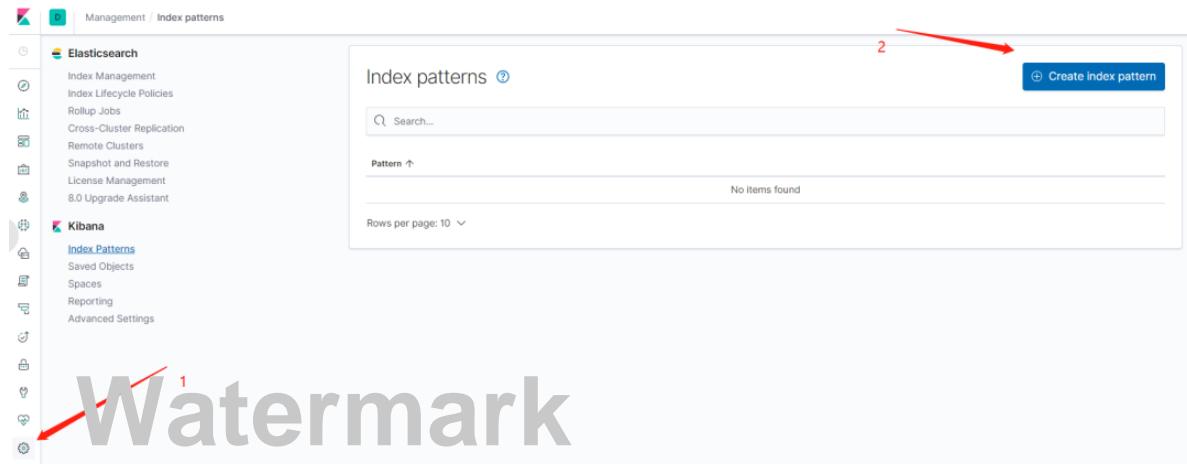
启动 Shipper 角色的 Logstash。

```
# 进入到 Logstash 的解压目录，然后执行下面的命令  
bin/logstash -f shipper-logstash.conf
```

调用 Spring Boot 接口，此时应该已经有数据写入到 ES 中了。

在浏览器中访问 <http://ip:5601>，打开 Kibana 的 Web 界面，并且如下图所示添加 logback 索引。

图 8. 在 Kibana 中添加 Elasticsearch 索引



图片

进入 Discover 界面，选择 logback 索引，就可以看到日志数据了，如下图所示。

图 9. ELK 日志查看

图片

在 Nginx 中使用 ELK

相信通过上面的步骤您已经成功的搭建起了自己的 ELK 实时日志平台，并且接入了 Logback 类型的日志。但是实际场景下，几乎不可能只有一种类型的日志，下面我们就再在上面步骤的基础之上接入 Nginx 的日志。

当然这一步的前提是我们需要在服务器上安装 Nginx，具体的安装过程网上有很多介绍，这里不再赘述。查看 Nginx 的日志如下（Nginx 的访问日志默认在 /var/log/nginx/access.log 文件中）。

清单 9. Nginx 的访问日志

```
192.168.142.1 -- [17/Aug/2019:21:31:43 +0800] "GET / weblog/get-test?name=elk HTTP/1.1"
200 3 "http://192.168.142.131/swagger-ui.html" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/76.0.3809.100 Safari/537.36"
```

同样，我们需要为此日志编写一个 Grok 解析规则，如下所示：

清单 10. 针对 Nginx 访问日志的 Grok 解析规则

```
%{IPV4:ip} \- \- [%{HTTPDATE:time}] "%{NOTSPACE:method} %{DATA:requestUrl}
HTTP/%{NUMBER:httpVersion}" %{NUMBER:httpStatus} %{NUMBER:bytes}
"%{DATA:referer}" "%{DATA:agent}"
```

完成上面这些之后的关键点是 Indexer 类型的 Logstash 需要支持两种类型的输入、过滤器以及输出，如何支持呢？首先需要给输入指定类型，然后再根据不同的输入类型走不同的过滤器和输出，如下所示。

清单 11. 支持两种日志输入的 Indexer 角色的 Logstash 配置

```
input {
    redis {
        type => "logback"
        ...
    }
}

filter {
    type => "nginx"
    ...
}
```

```

filter {
    if [type] == "logback" {
        ...
    }
    if [type] == "nginx" {
        ...
    }
}

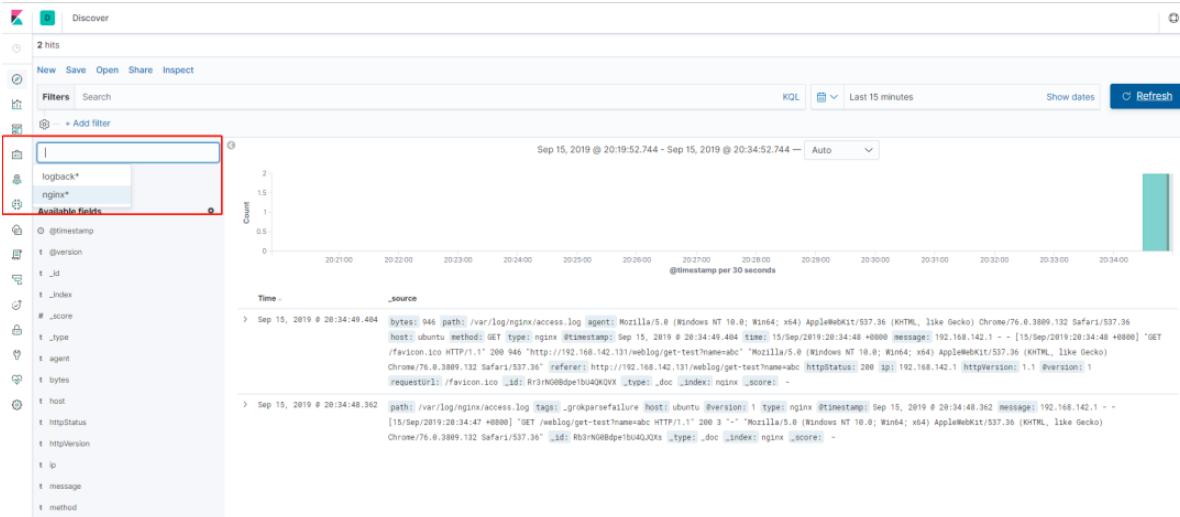
output {
    if [type] == "logback" {
        ...
    }
    if [type] == "nginx" {
        ...
    }
}

```

我的 Nginx 与 Spring Boot 项目部署在同一台机器上，所以还需修改 Shipper 类型的 Logstash 的配置以支持两种类型的日志输入和输出，其配置文件的内容可 [点击这里获取](#)。

以上配置完成后，我们按照 查看效果 章节中的步骤，启动 ELK 平台、Shipper 角色的 Logstash、Nginx 以及 Spring Boot 项目，然后在 Kibana 上添加 Nginx 索引后就可同时查看 Spring Boot 和 Nginx 的日志了，如下图所示。

图 10. ELK 查看 Nginx 日志



图片

ELK 启动

在上面的步骤中，ELK 的启动过程是我们一个一个的去执行三大组件的启动命令的。而且还是在前台启动的，意味着如果我们关闭会话窗口，该组件就会停止导致整个 ELK 平台无法使用，这在实际工作过程中是不现实的，我们剩下的问题就在于如何使 ELK 在后台运行。

根据《Logstash 最佳实践》一书的推荐，我们将使用 Supervisor 来管理 ELK 的启停。首先我们需要安装 Supervisor，在 Ubuntu 上执行 `sudo apt-get install supervisor` 即可。安装成功后，我们还需要在 Supervisor 的配置文件中配置 ELK 三大组件（其配置文件默认为 `/etc/supervisor/supervisord.conf` 文件）。

清单 12. ELK 后台启动

```
[program:elasticsearch]
environment=JAVA_HOME="/usr/java/jdk1.8.0_221/"
directory=/home/elk/elk/elasticsearch
user=elk
command=/home/elk/elk/elasticsearch/bin/elasticsearch

[program:logstash]
environment=JAVA_HOME="/usr/java/jdk1.8.0_221/"
directory=/home/elk/elk/logstash
user=elk
command=/home/elk/elk/logstash/bin/logstash -f /home/elk/elk/logstash/indexer-logstash.conf

[program:kibana]
environment=LS_HEAP_SIZE=5000m
directory=/home/elk/elk/kibana
user=elk
command=/home/elk/elk/kibana/bin/kibana
```

按照以上内容配置完成后，执行 `sudo supervisorctl reload` 即可完成整个 ELK 的启动，而且其默认是开机自启。当然，我们也可以使用 `sudo supervisorctl start/stop [program_name]` 来管理单独的应用。另外，欢迎关注公众号码猿技术专栏，后台回复“9527”，送你一份Spring Cloud Alibaba实战视频！

结束语

在本教程中，我们主要了解了什么是 ELK，然后通过实际操作和大家一起搭建了一个 ELK 日志分析平台，并且接入了 Logback 和 Nginx 两种日志。

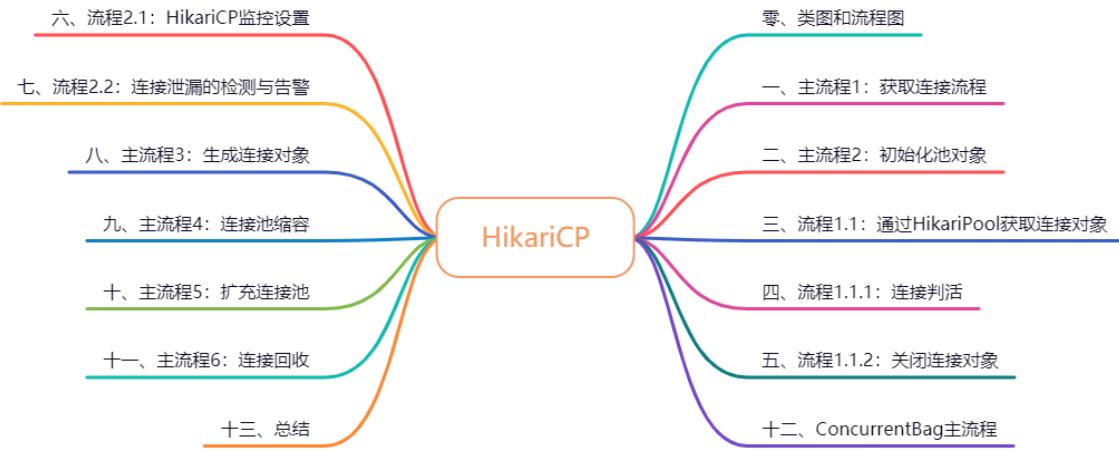
Spring Boot 青睐的数据库连接池HikariCP为什么是史上最快的？

前言

现在已经有很多公司在使用HikariCP了，HikariCP还成为了SpringBoot默认的连接池，伴随着SpringBoot和微服务，HikariCP 必将迎来广泛的普及。

下面陈某带大家从源码角度分析一下HikariCP为什么能够被Spring Boot 青睐，文章目录如下：

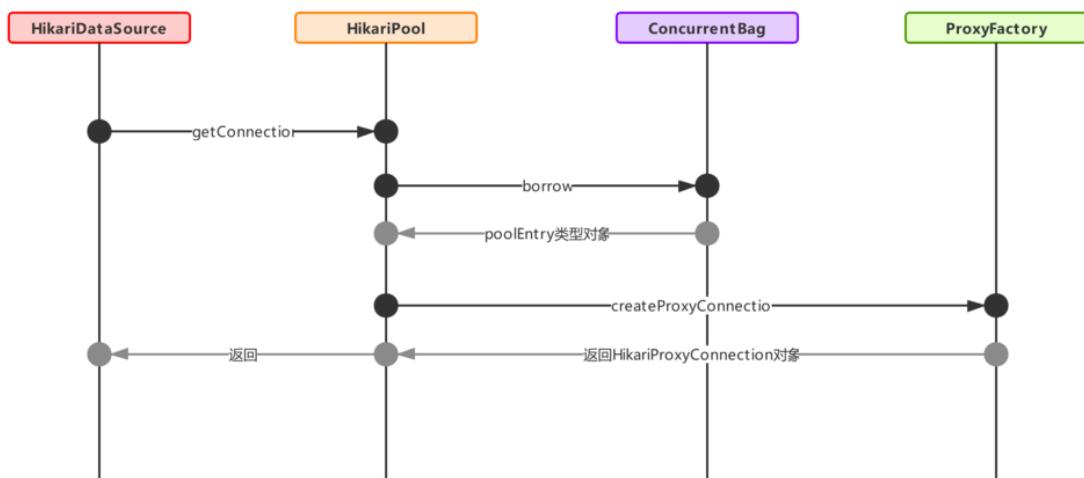
Watermark



零、类图和流程图

开始前先来了解下HikariCP获取一个连接时类间的交互流程，方便下面详细流程的阅读。

获取连接时的类间交互：



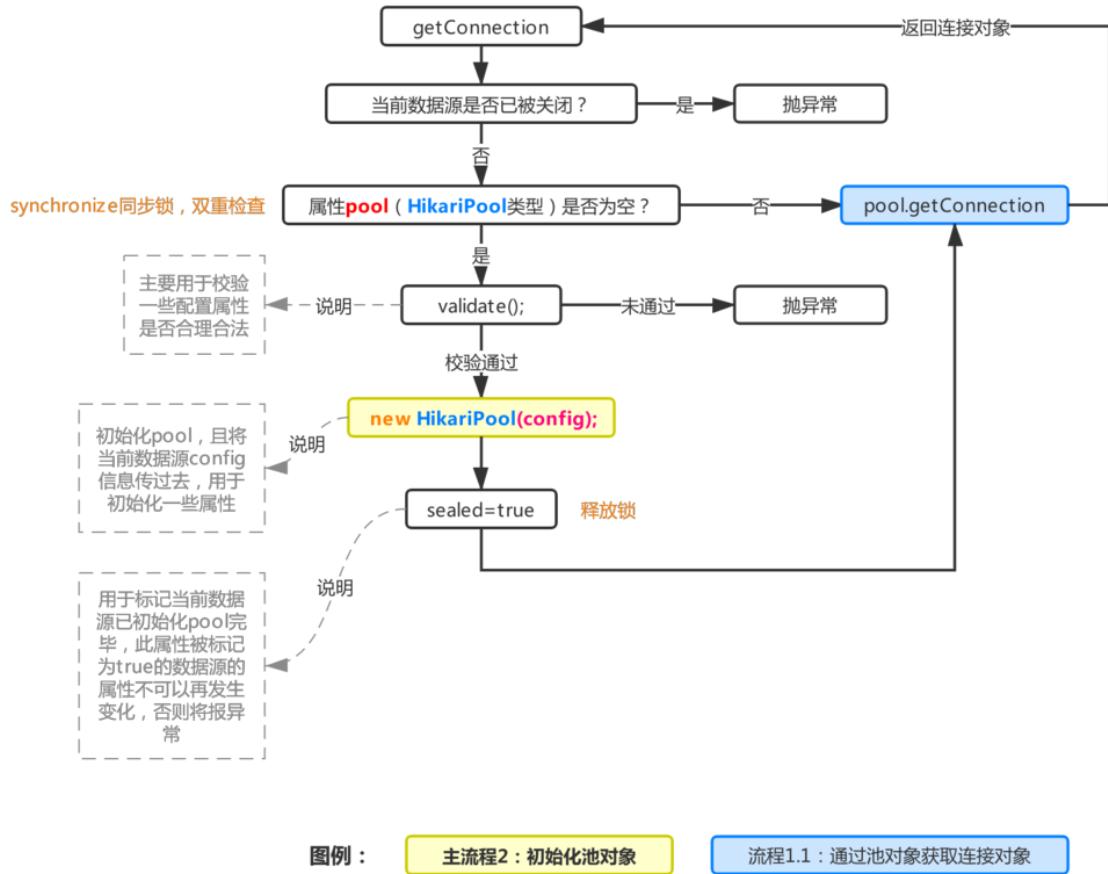
- █ 数据源操作类，直接暴露给用户的类，持有HikariPool对象
- █ 池子管理类，用于取连接、丢弃连接、回收连接等的触发，持有一个ConcurrentBag对象，很多操作都是面向ConcurrentBag的
- █ 真正存放连接对象的地方，内部持有一个CopyOnWriteArrayList对象，保存连接，此外还提供一个ThreadLocal对象，用于缓存当前线程内的连接对象
- █ 生成包装类的地方，对外提供的HikariProxyConnection可以在常规操作的同时做一些额外的工作

图1

一、主流程1：获取连接流程

HikariCP获取连接时的入口是 `HikariDataSource` 里的 `getConnection` 方法，现在来看下该方法的具体流程：

Watermark

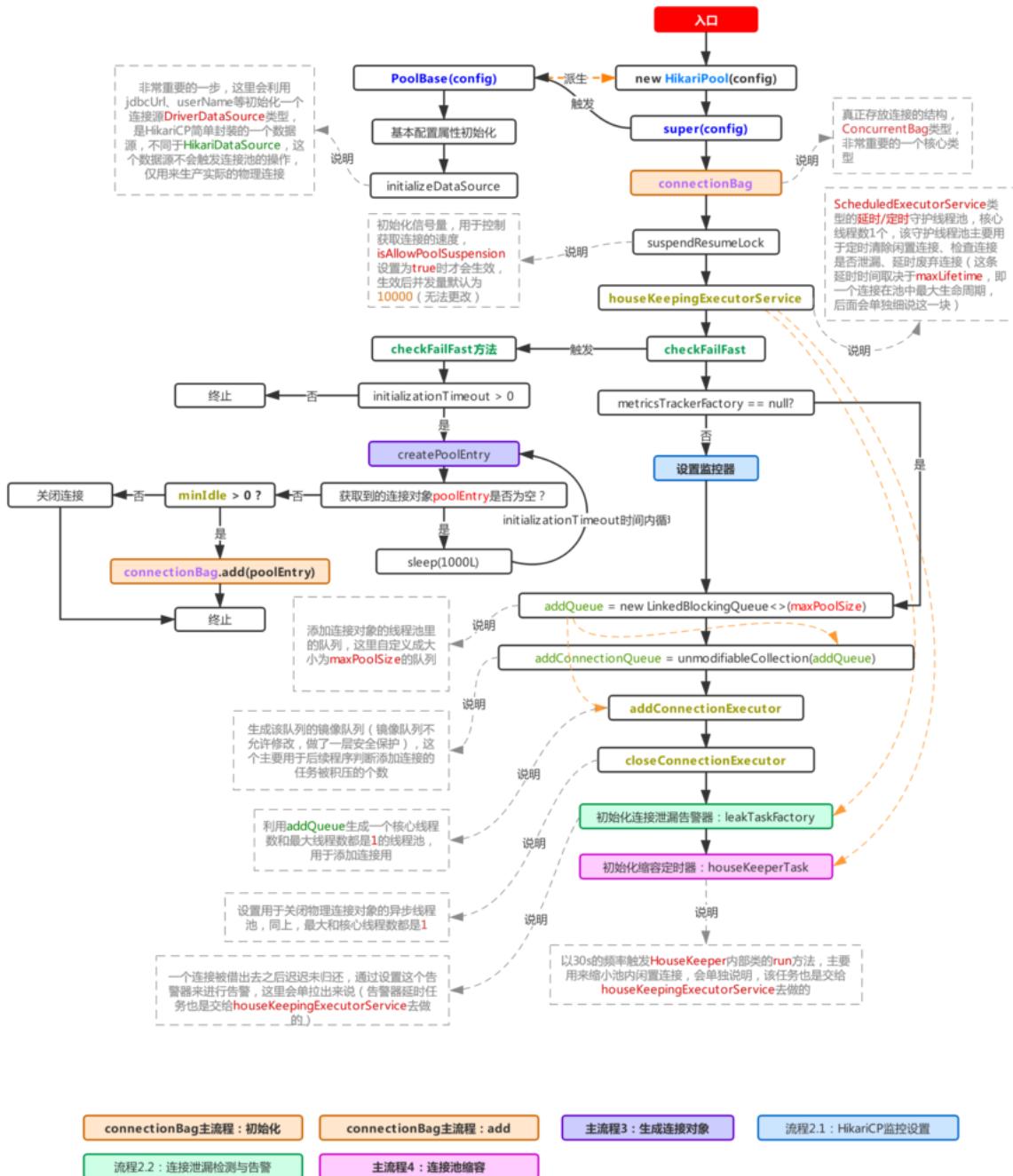


主流程1

上述为HikariCP获取连接时的流程图，由图1可知，每个 `datasource` 对象里都会持有一个 `HikariPool` 对象，记为`pool`，初始化后的`datasource`对象`pool`是空的，所以第一次 `getConnection` 的时候会进行 `实例化pool` 属性（参考 [主流程1](#)），初始化的时候需要将当前`datasource`里的 `config`属性 传过去，用于`pool`的初始化，最终标记 `sealed`，然后根据`pool`对象调用 `getConnection` 方法（参考 [流程1.1](#)），获取成功后返回连接对象。

二、主流程2：初始化池对象

Watermark

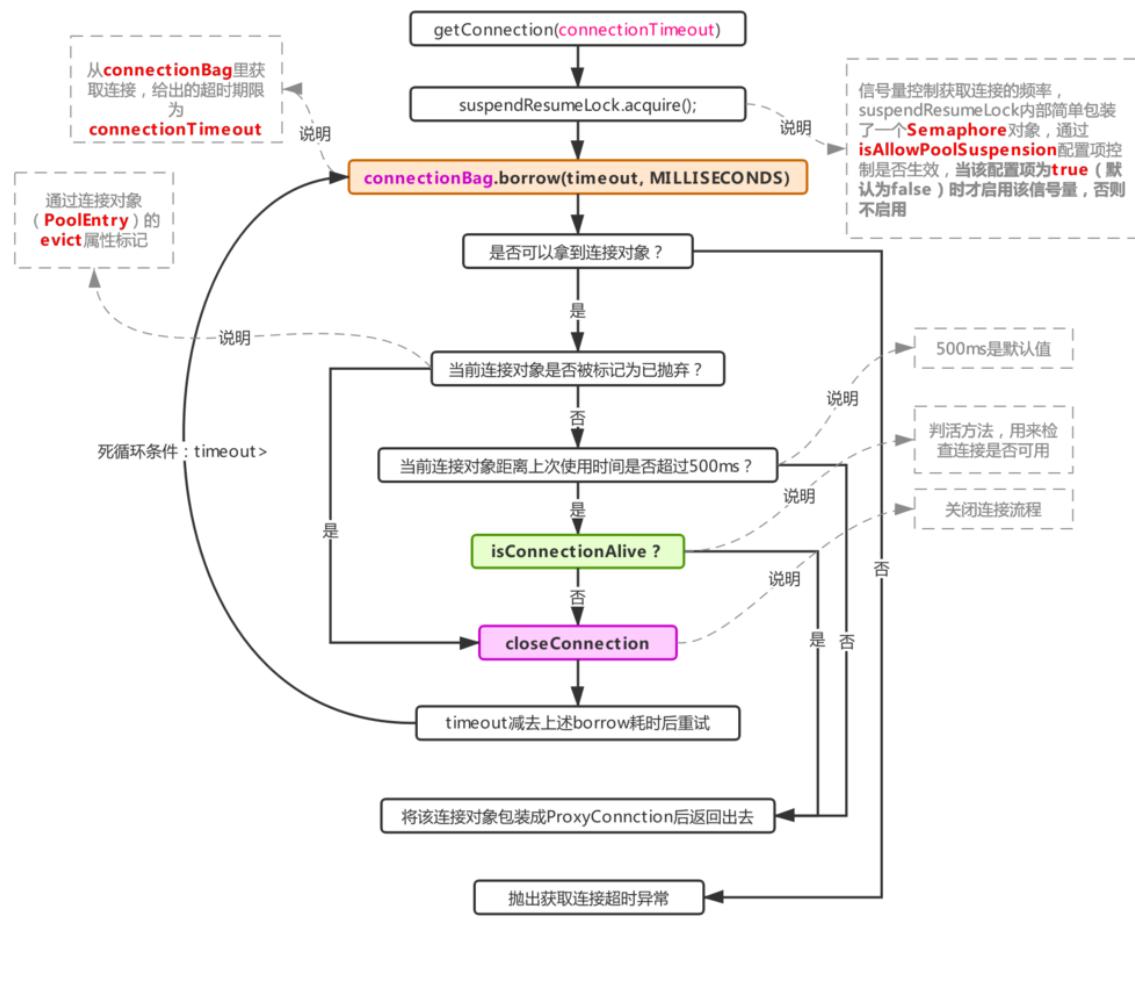


主流程2

该流程用于 **初始化整个连接池**，这个流程会给连接池内所有的属性做初始化的工作，其中比较主要的几个流程上图已经指出，简单概括一下：

- 利用 `config` 初始化各种连接池属性，并且产生一个用于 **生产物理连接** 的数据源 `DriverDataSource`
- 初始化存放连接对象的核心类 `connectionBag`
- 初始化一个延时任务线程池类型的对象 `houseKeepingExecutorService`，用于后续执行一些延时/定时类任务（比如连接泄漏检查延时任务，参考 [流程2.2](#) 以及 [主流程4](#)，除此之外 `maxLifeTime` 后主动回收关闭连接也是交由该对象来执行的，这个过程可以参考 [主流程3](#)）
- 预热连接池，HikariCP会在该流程的 `checkFailFast` 里初始化好一个连接对象放进池子内，当然触发该流程得保证 `initializationTimeout > 0` 时（默认值1），这个配置属性表示留给预热操作的时司（默认值1在预热失败时不会发生重试）。与 Druid 通过 `initialSize` 控制预热连接对象数不一样的是，HikariCP仅预热进池一个连接对象。
- 初始化一个线程池对象 `addConnectionExecutor`，用于后续扩充连接对象
- 初始化一个线程池对象 `closeConnectionExecutor`，用于关闭一些连接对象，怎么触发关闭任务呢？可以参考 [流程1.1.2](#)

三、流程1.1：通过HikariPool获取连接对象



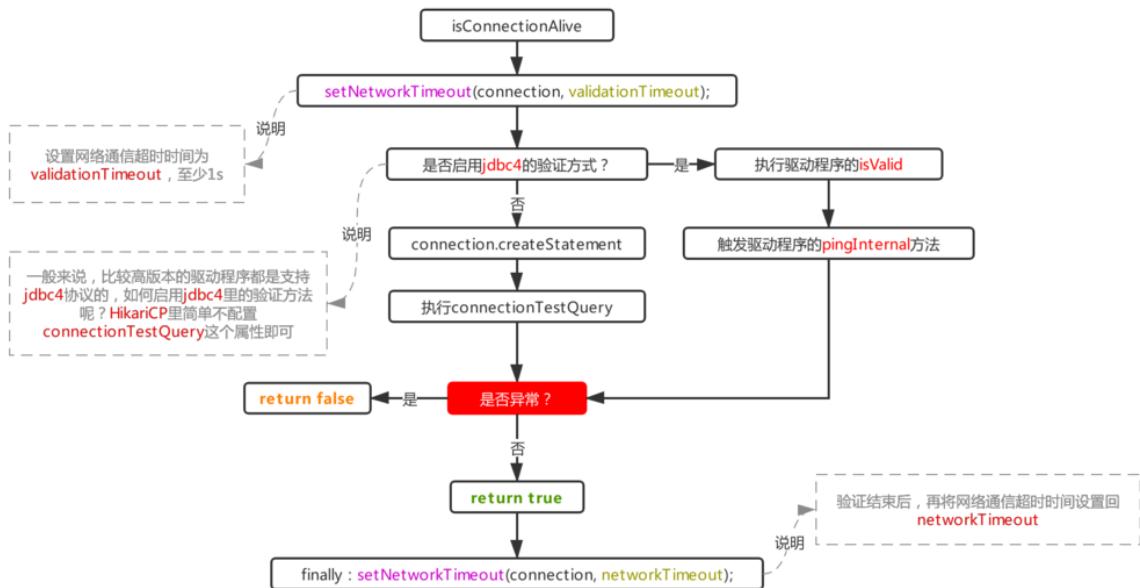
流程1.1

从最开始的结构图可知，每个 HikariPool 里都维护一个 `ConcurrentBag` 对象，用于存放连接对象，由上图可以看到，实际上 HikariPool 的 `getConnection` 就是从 `ConcurrentBag` 里获取连接的（调用其 `borrow` 方法获得，对应 ConnectionBag主流程），在长连接检查这块，与之前说的 Druid 不同，这里的长连接判活检查在连接对象没有被标记为“已丢弃”时，只要距离上次使用超过 500ms 每次取出都会进行检查（500ms是默认值，可通过配置 `com.zaxxer.hikari.aliveBypassWindowMs` 的系统参数来控制），emmmmm，也就是说 HikariCP 对长连接的活性检查很频繁，但是其并发性能依旧优于 Druid，说明频繁的长连接检查并不是导致连接池性能高低的关键所在。

这个其实是由于HikariCP的 无锁 实现，在高并发时对CPU的负载没有其他连接池那么高而产生的并发性能差异，后面会说HikariCP的具体做法，即使是 Druid，在 获取连接、生成连接、归还连接 时都进行了 锁控制，因为通过上篇解析 Druid 的文章可以知道，Druid 里的连接池资源是多线程共享的，不可避免的会有锁竞争，有锁竞争意味着线程状态的变化会很频繁，线程状态变化频繁意味着CPU上下文切换也将会很频繁。

回到 流程1.1，如果拿到的连接为空，直接报告。不为空则进行相应的检查，如果检查通过，则包装成 `ConnectionProxy` 对象返回给业务方，不通过则调用 `closeConnection` 方法关闭连接（对应 流程1.1.2，该流程会触发 `ConcurrentBag` 的 `remove` 方法丢弃该连接，然后把实际的驱动连接交给 `closeConnectionExecutor` 线程池，异步关闭驱动连接）。

四、流程1.1.1：连接判活



流程1.1.1

承接上面的 [流程1.1](#) 里的判活流程，来看下判活是如何做的，首先说验证方法（注意这里该方法接受的这个 `connection` 对象不是 `poolEntry`，而是 `poolEntry` 持有的实际驱动的连接对象），在之前介绍Druid的时候就知道，Druid是根据驱动程序里是否存在 `ping`方法 来判断是否启用ping的方式判断连接是否存活，但是到了HikariCP则更加简单粗暴，仅根据是否配置了 `connectionTestQuery` 觉定是否启用ping：

```
this.isUseJdbc4Validation = config.getConnectionTestQuery() == null;
```

所以一般驱动如果不是特别低的版本，不建议配置该项，否则便会走 `createStatement+execute` 的方式，相比 `ping` 简单发送心跳数据，这种方式显然更低效。

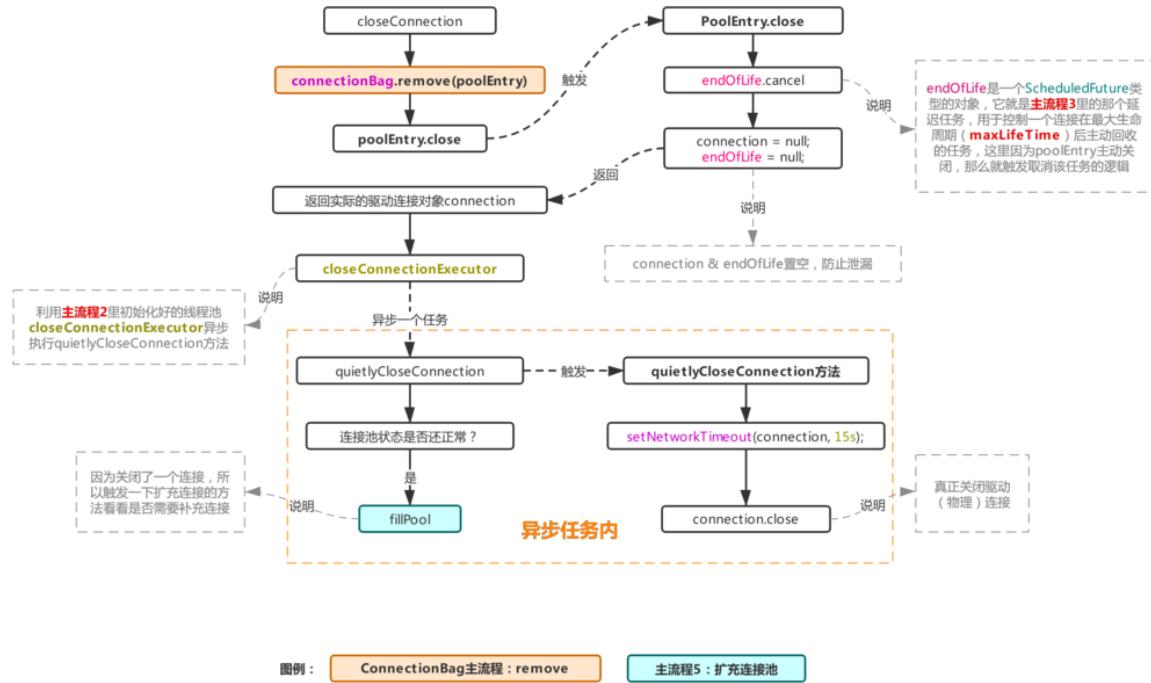
此外，这里在刚进来还会通过驱动的连接对象重新给它设置一遍 `networkTimeout` 的值，使之变成 `validationTimeout`，表示一次验证的超时时间，为啥这里要重新设置这个属性呢？因为在使用ping方法校验时，是没办法通过类似 `statement` 那样可以 `setQueryTimeout` 的，所以只能由网络通信的超时时间来控制，这个时间可以通过 `jdbc` 的连接参数 `socketTimeout` 来控制：

```
jdbc:mysql://127.0.0.1:3306/xxx?socketTimeout=250
```

这个值最终会被赋值给HikariCP的 `networkTimeout` 字段，这就是为什么最后那一步使用这个字段来还原驱动连接超时属性的原因；说到这里，最后那里为啥要再次还原呢？这就很容易理解了，因为验证结束了，连接对象还存活的情况下，它的 `networkTimeout` 的值这时仍然等于 `validationTimeout`（不合预期），显然在拿出去用之前，需要恢复本来的值，也就是HikariCP里的 `networkTimeout` 属性。

五、流程1.1.2：关闭连接对象

Watermark



流程1.1.2

这个流程简单来说就是把 [流程1.1.1](#) 中验证不通过的死连接，主动关闭的一个流程，首先会把这个连接对象从 ConnectionBag 里 移除，然后把实际的物理连接交给一个线程池去异步执行，这个线程池就是在 [主线程2](#) 里初始化池的时候初始化的线程池 closeConnectionExecutor，然后异步任务内开始实际的关连接操作，因为主动关闭了一个连接相当于少了一个连接，所以还会触发一次扩充连接池（参考 [主线程5](#)）操作。

六、流程2.1：HikariCP监控设置

不同于Druid那样监控指标那么多，HikariCP会把我们非常关心的几项指标暴露给我们，比如当前连接池内闲置连接数、总连接数、一个连接被用了多久归还、创建一个物理连接花费多久等，HikariCP的连接池的监控我们这一节专门详细的分解一下，首先找到HikariCP下面的 `metrics` 文件夹，这下面放置了一些规范实现的监控接口等，还有一些现成的实现（比如HikariCP自带对 `prometheus`、`micrometer`、`dropwizard` 的支持，不太了解后面两个，`prometheus` 下文直接称为 普罗米修斯）：

Watermark

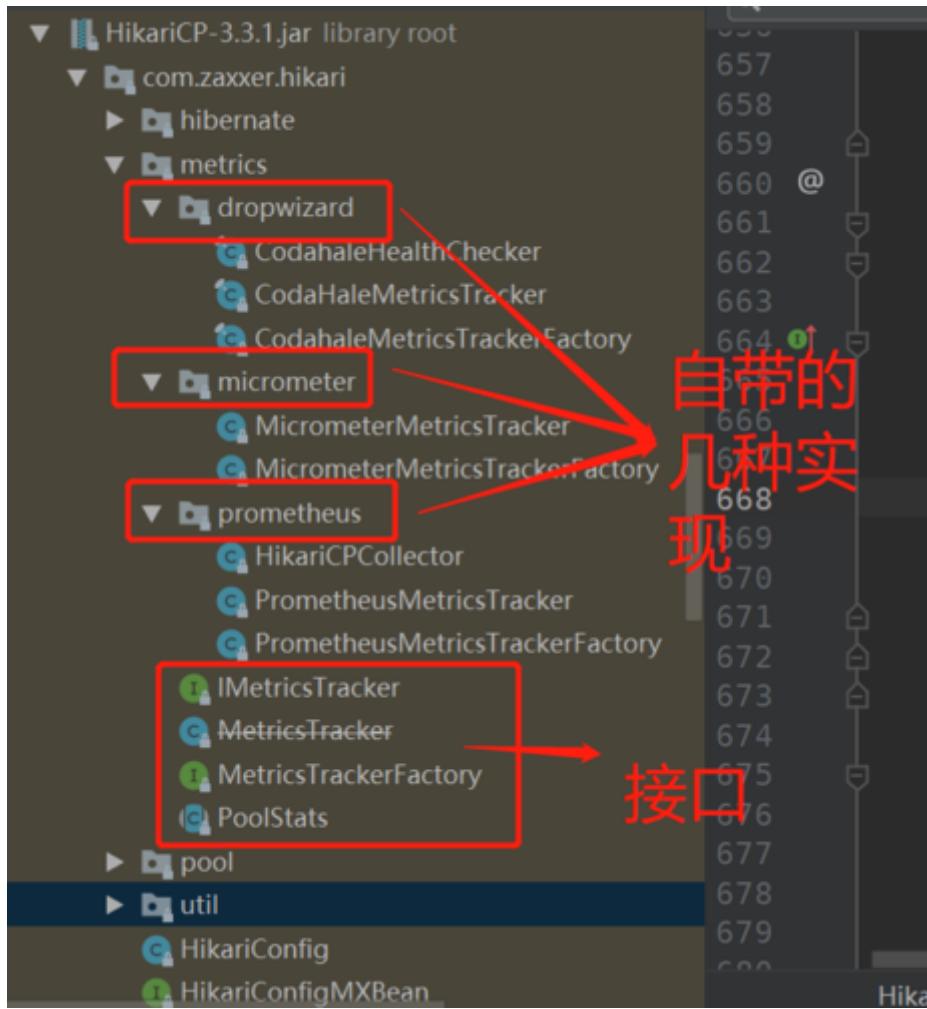


图2

下面，来着重看下接口的定义：

```
//这个接口的实现主要负责收集一些动作的耗时
public interface IMetricsTracker extends AutoCloseable
{
    //这个方法触发点在创建实际的物理连接时（主流程3），用于记录一个实际的物理连接创建所耗费的时间
    default void recordConnectionCreatedMillis(long connectionCreatedMillis) {}

    //这个方法触发点在getConnection时（主流程1），用于记录获取一个连接时实际的耗时
    default void recordConnectionAcquiredNanos(final long elapsedAcquiredNanos) {}

    //这个方法触发点在回收连接时（主流程6），用于记录一个连接从被获取到被回收时所消耗的时间
    default void recordConnectionUsageMillis(final long elapsedBorrowedMillis) {}

    //这个方法触发点也在getConnection时（主流程1），用于记录获取连接超时的次数，每发生一次获取连接超时，就会触发一次该方法的调用
    default void recordConnectionTimeout() {}

    @Override
    default void close() {}
}
```

触发点到了库源码中，来看看 MetricsTrackerFactory 的接口定义：

```
//用于创建IMetricsTracker实例，并且按需记录PoolStats对象里的属性（这个对象里的属性就是类似连接池当前闲置连接数之类的线程池状态类指标）
public interface MetricsTrackerFactory {
    //返回一个IMetricsTracker对象，并且把PoolStats传了过去
    IMetricsTracker create(String poolName, PoolStats poolStats);
}
```

上面的接口用法见注释，针对新出现的 `PoolStats` 类，我们来看看它做了什么：

```
public abstract class PoolStats {
    private final AtomicLong reloadAt; //触发下次刷新的时间（时间戳）
    private final long timeoutMs; //刷新下面的各项属性值的频率，默认1s，无法改变

    // 总连接数
    protected volatile int totalConnections;
    // 闲置连接数
    protected volatile int idleConnections;
    // 活动连接数
    protected volatile int activeConnections;
    // 由于无法获取到可用连接而阻塞的业务线程数
    protected volatile int pendingThreads;
    // 最大连接数
    protected volatile int maxConnections;
    // 最小连接数
    protected volatile int minConnections;

    public PoolStats(final long timeoutMs) {
        this.timeoutMs = timeoutMs;
        this.reloadAt = new AtomicLong();
    }

    //这里以获取最大连接数为例，其他的跟这个差不多
    public int getMaxConnections() {
        if (shouldLoad()) { //是否应该刷新
            update(); //刷新属性值，注意这个update的实现在HikariPool里，因为这些属性值的直接或间接
来源都是HikariPool
        }
        return maxConnections;
    }

    protected abstract void update(); //实现在↑上面已经说了

    private boolean shouldLoad() { //按照更新频率来决定是否刷新属性值
        for (; ; ) {
            final long now = currentTime();
            final long reloadTime = reloadAt.get();
            if (reloadTime > now) {
                return false;
            } else if (reloadAt.compareAndSet(reloadTime, plusMillis(now, timeoutMs))) {
                return true;
            }
        }
    }
}
```

实际上这里就是这些属性获取和触发刷新的地方，那么这个对象是在哪里被生成并且丢给 MetricsTrackerFactory 的 create 方法的呢？这就是本节所需要讲述的要点：[主流程2](#) 里的设置监控器的流程，来看看那里发生了什么事吧：

```
//监控器设置方法（此方法在HikariPool中，metricsTracker属性就是HikariPool用来触发IMetricsTracker里方法调用的）
public void setMetricsTrackerFactory(MetricsTrackerFactory metricsTrackerFactory) {
    if (metricsTrackerFactory != null) {
        //MetricsTrackerDelegate是包装类，是HikariPool的一个静态内部类，是实际持有IMetricsTracker对象的类，也是实际触发IMetricsTracker里方法调用的类
        //这里首先会触发MetricsTrackerFactory类的create方法拿到IMetricsTracker对象，然后利用getPoolStats初始化PoolStat对象，然后也一并传给MetricsTrackerFactory
        this.metricsTracker = new MetricsTrackerDelegate(metricsTrackerFactory.create(config.getPoolName(), getPoolStats()));
    } else {
        //不启用监控，直接等于一个没有实现方法的空类
        this.metricsTracker = new NopMetricsTrackerDelegate();
    }
}

private PoolStats getPoolStats() {
    //初始化PoolStats对象，并且规定1s触发一次属性值刷新的update方法
    return new PoolStats(SECONDS.toMillis(1)) {
        @Override
        protected void update() {
            //实现了PoolStat的update方法，刷新各个属性的值
            this.pendingThreads = HikariPool.this.getThreadsAwaitingConnection();
            this.idleConnections = HikariPool.this.getIdleConnections();
            this.totalConnections = HikariPool.this.getTotalConnections();
            this.activeConnections = HikariPool.this.getActiveConnections();
            this.maxConnections = config.getMaximumPoolSize();
            this.minConnections = config.getMinimumIdle();
        }
    };
}
```

到这里HikariCP的监控器就算是注册进去了，所以要想实现自己的监控器拿到上面的指标，要经过如下步骤：

1. 新建一个类实现 IMetricsTracker 接口，我们这里将该类记为 `IMetricsTrackerImpl`
2. 新建一个类实现 MetricsTrackerFactory 接口，我们这里将该类记为 `MetricsTrackerFactoryImpl`，并且将上面的 `IMetricsTrackerImpl` 在其 `create`方法 内实例化
3. 将 `MetricsTrackerFactoryImpl` 实例化后调用HikariPool的 `setMetricsTrackerFactory` 方法注册到 Hikari连接池。

上面没有提到 `PoolStats` 里的属性怎么监控，这里来说下，由于 `create`方法 是调用一次就没了，`create`方法 只是接收了 `PoolStats` 对象的实例，如果不处理，那么随着 `create`调用的结束，这个实例针对监控模块来说就失去持有了，所以这里如果想要拿到 `PoolStats` 里的属性，就需要开启一个 `守护线程`，让其持有 `PoolStats` 对象实例，并且定时获取其内部属性值，然后 `push` 给监控系统，如果是普罗米修斯等使用 `pull`方式 获取监控数据的监控系统，可以效仿HikariCP原生普罗米修斯监控的实现，自定义一个 `Collector`对象来接收 `PoolStats` 实例，这样普罗米修斯就可以定期拉取了，比如HikariCP根据普罗米修斯监控系统自己定义的 `MetricsTrackerFactory` 实现（对应 [图2](#) 里的 `PrometheusMetricsTrackerFactory` 类）：

```

@Override
public IMetricsTracker create(String poolName, PoolStats poolStats) {
    getCollector().add(poolName, poolStats); //将接收到的PoolStats对象直接交给Collector，这样普罗米修斯服务端每触发一次采集接口的调用，PoolStats都会跟着执行一遍内部属性获取流程
    return new PrometheusMetricsTracker(poolName, this.collectorRegistry); //返回IMetricsTracker接口的实现类
}

//自定义的Collector
private HikariCPCollector getCollector() {
    if (collector == null) {
        //注册到普罗米修斯收集中心
        collector = new HikariCPCollector().register(this.collectorRegistry);
    }
    return collector;
}

```

通过上面的解释可以知道在HikariCP中如何自定义一个自己的监控器，以及相比Druid的监控，有什么区别。工作中很多时候都是需要自定义的，我司虽然也是用的普罗米修斯监控，但是因为HikariCP原生的普罗米修斯收集器里面对监控指标的命名并 不符合我司的规范，所以就 自定义 了一个，有类似问题的不妨也试一试。

✿ 这一节没有画图，纯代码，因为画图不太好解释这部分的东西，这部分内容与连接池整体流程关系也不大，充其量获取了连接池本身的一些属性，在连接池里的触发点也在上面代码段的注释里说清楚了，看代码定义可能更好理解一些。

七、流程2.2：连接泄漏的检测与告警

本节对应 [主流程2](#) 里的 [子流程2.2](#)，在初始化池对象时，初始化了一个叫做 `leakTaskFactory` 的属性，本节来看下它具体是用来做什么的。

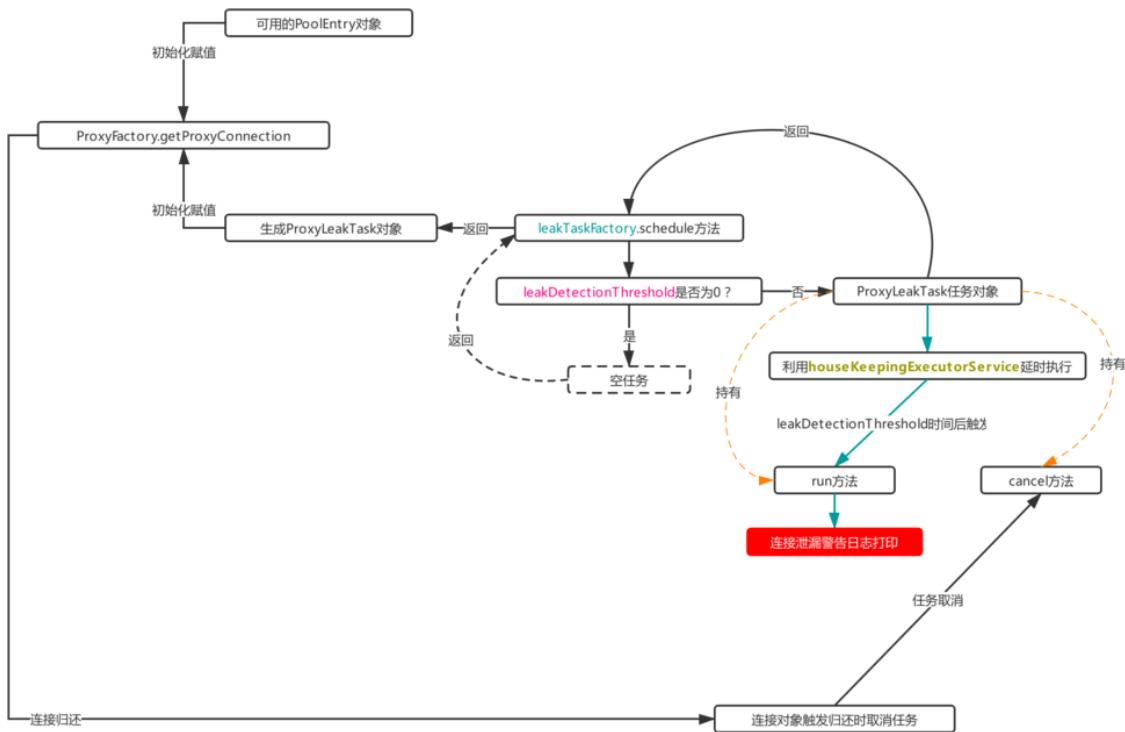
7.1：它是做什么的？

一个连接被拿出去使用时间超过 `leakDetectionThreshold`（可配置，默认0）未归还的，会触发一个连接泄漏警告，通知业务方目前存在连接泄漏的问题。

7.2：过程详解

该属性是 `ProxyLeakTaskFactory` 类型对象，且它还会持有 `houseKeepingExecutorService` 这个线程池对象，用于生产 `ProxyLeakTask` 对象，然后利用上面的 `houseKeepingExecutorService` 延时运行该对象里的 `run` 方法。该流程的触发点在上面的 [流程1.1](#) 最后包装成 `ProxyConnection` 对象的那一步，来看看具体的流程图：

Watermark



流程2.2

每次在 [流程1.1](#) 那里生成 `ProxyConnection` 对象时，都会触发上面的流程，由流程图可以知道，`ProxyConnection` 对象持有 `PoolEntry` 和 `ProxyLeakTask` 的对象，其中初始化 `ProxyLeakTask` 对象时就用到了 `leakTaskFactory` 对象，通过其 `schedule` 方法可以进行 `ProxyLeakTask` 的初始化，并将其实例传递给 `ProxyConnection` 进行初始化赋值（ps：由图知 `ProxyConnection` 在触发回收事件时，会主动取消这个泄漏检查任务，这也是 `ProxyConnection` 需要持有 `ProxyLeakTask` 对象的原因）。

在上面的流程图中可以知道，只有在 `leakDetectionThreshold` 不等于0的时候才会生成一个带有实际延时任务的 `ProxyLeakTask` 对象，否则返回无实际意义的空对象。所以要想启用连接泄漏检查，首先要把 `leakDetectionThreshold` 配置设置上，这个属性表示经过该时间后借出去的连接仍未归还，则触发连接泄漏告警。

`ProxyConnection` 之所以要持有 `ProxyLeakTask` 对象，是因为它可以监听到连接是否触发归还操作，如果触发，则调用 `cancel` 方法取消延时任务，防止误告。

由此流程可以知道，跟Druid一样，HikariCP也有连接对象泄漏检查，与Druid主动回收连接相比，HikariCP实现更加简单，仅仅是在触发时打印警告日志，不会采取具体的强制回收的措施。

与Druid一样，默认也是关闭这个流程的，因为实际开发中一般使用第三方框架，框架本身会保证及时的close连接，防止连接对象泄漏，开启与否还是取决于业务是否需要，如果一定要开启，如何设置 `leakDetectionThreshold` 的大小也是需要考虑的一件事。

八、主流程3：生成连接对象

本节来讲下 [主流程2](#) 里的 `createEntry` 方法，这个方法利用`PoolBase`里的 `DriverDataSource` 对象生成一个实际的连接对象（如果忘记 `DriverDataSource` 是哪里初始化的了，可以看下 [主流程2](#) 里 `PoolBase` 的 `initializeDatasource` 方法的作用），然后用 `PoolEntry` 类包装成 `PoolEntry` 对象，现在来看下这个包装类有哪些主要属性：

```

final class PoolEntry implements IConcurrentBagEntry {
    private static final Logger LOGGER = LoggerFactory.getLogger(PoolEntry.class);
    //通过cas来修改state属性
}

```

```

private static final AtomicIntegerFieldUpdater stateUpdater;

Connection connection; //实际的物理连接对象
long lastAccessed; //触发回收时刷新该时间，表示“最近一次使用时间”
long lastBorrowed; //getConnection里borrow成功后刷新该时间，表示“最近一次借出的时间”

@SuppressWarnings("FieldCanBeLocal")
private volatile int state = 0; //连接状态，枚举值：IN_USE（使用中）、NOT_IN_USE（闲置中）、
REMOVED（已移除）、RESERVED（标记为保留中）

private volatile boolean evict; //是否被标记为废弃，很多地方用到（比如流程1.1靠这个判断连接是否已被废弃，再比如主流程里时钟回拨时触发的直接废弃逻辑）

private volatile ScheduledFuture<?> endOfLife; //用于在超过连接生命周期（maxLifeTime）时废弃连接的延时任务，这里poolEntry要持有该对象，主要是因为在对象主动被关闭时（意味着不需要在超过maxLifeTime时主动失效了），需要cancel掉该任务

private final FastList openStatements; //当前该连接对象上生成的所有statement对象，用于在回收连接时主动关闭这些对象，防止存在漏关的statement
private final HikariPool hikariPool; //持有pool对象

private final boolean isReadOnly; //是否为只读
private final boolean isAutoCommit; //是否存在事务
}

```

上面就是整个 PoolEntry 对象里所有的属性，这里再说下 endOfLife 对象，它是一个利用 houseKeepingExecutorService 这个线程池对象做的延时任务，这个延时任务一般在创建好连接对象后 maxLifeTime 左右的时间触发，具体来看下 createEntry 代码：

```

private PoolEntry createPoolEntry() {

    final PoolEntry poolEntry = newPoolEntry(); //生成实际的连接对象

    final long maxLifetime = config.getMaxLifetime(); //拿到配置好的maxLifetime
    if (maxLifetime > 0) { //<=0的时候不启用主动过期策略
        // 计算需要减去的随机数
        // 源注释：variance up to 2.5% of the maxlifetime
        final long variance = maxLifetime > 10_000 ?
            ThreadLocalRandom.current().nextLong(maxLifetime / 40) : 0;
        final long lifetime = maxLifetime - variance; //生成实际的延时时间
        poolEntry.setFutureEnd(houseKeepingExecutorService.schedule(
            () -> { //实际的延时任务，这里直接触发softEvictConnection，而
                softEvictConnection内则会标记该连接对象为废弃状态，然后尝试修改其状态为STATE_RESERVED，若成功，则触发
                closeConnection（对应流程1.1.2）
                if (softEvictConnection(poolEntry, "(connection has passed
maxLifetime)", false /* not owner *)) {
                    addBagItem(connectionBag.getWaitingThreadCount()); //回收完毕
                    后，连接池内少了一个连接，就会尝试新增一个连接对象
                }
            },
            lifetime, MILLISECONDS)); //给endOfLife赋值，并且提交延时任务，lifetime后
        触发
    }
    return poolEntry;
}

//触发新增连接任务

```

```

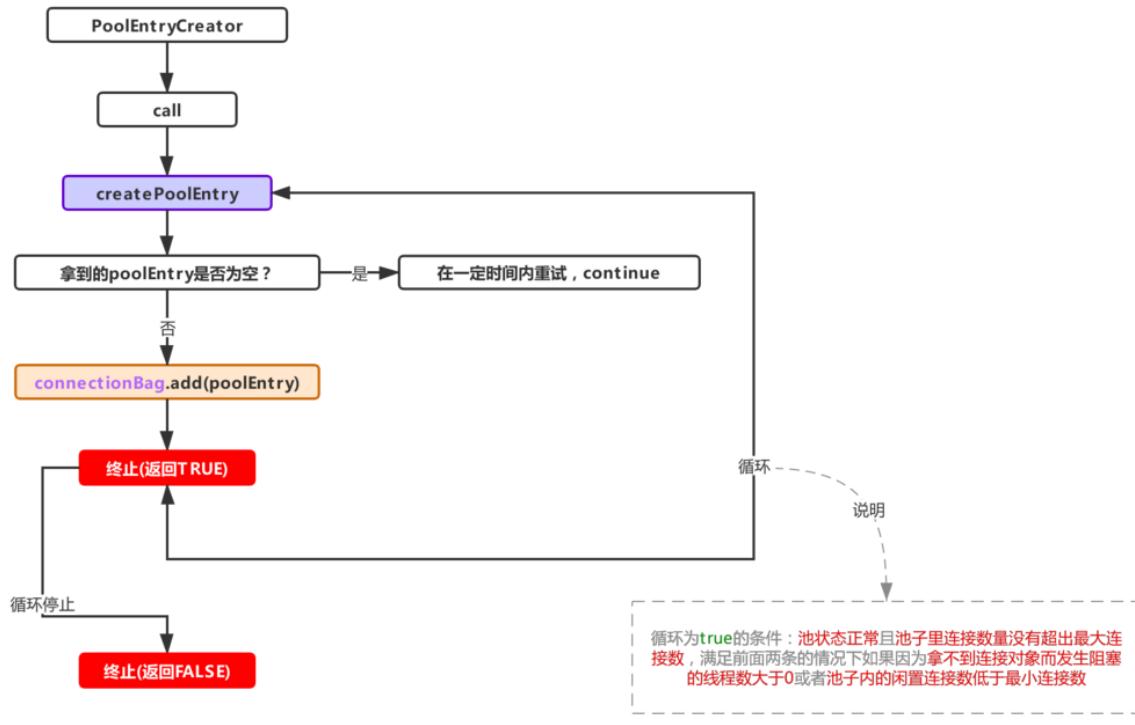
public void addBagItem(final int waiting) {
    //前排提示：addConnectionQueue和addConnectionExecutor的关系和初始化参考主流程2

    //当添加连接的队列里已提交的任务超过那些因为获取不到连接而发生阻塞的线程个数时，就进行提交连接新增连接的任务
    final boolean shouldAdd = waiting - addConnectionQueue.size() >= 0; // Yes, >= is intentional.

    if (shouldAdd) {
        //提交任务给addConnectionExecutor这个线程池，PoolEntryCreator是一个实现了Callable接口的类，下面将通过流程图的方式介绍该类的call方法
        addConnectionExecutor.submit(poolEntryCreator);
    }
}

```

通过上面的流程，可以知道，HikariCP一般通过 `createEntry` 方法来新增一个连接入池，每个连接被包装成`PoolEntry`对象，在创建好对象时，同时会提交一个延时任务来关闭废弃该连接，这个时间就是我们配置的 `maxLifeTime`，为了保证不在同一时间失效，HikariCP还会利用 `maxLifeTime` 减去一个随机数作为最终的延时任务延迟时间，然后在触发废弃任务时，还会触发 `addBagItem`，进行连接添加任务（因为废弃了一个连接，需要往池子里补充一个），该任务则交给由 [主流程2](#) 里定义好的 `addConnectionExecutor` 线程池执行，那么，现在来看下这个异步添加连接对象的任务流程：



图例：
connectionBag主线程：add 主流程3：生成连接对象

`addConnectionExecutor`的call流程

这个流程就是往连接池里加连接用的，跟 `createEntry` 结合起来说是因为这两流程是紧密相关的，除此之外，[主流程5](#) (`fillPool`，扩充连接池) 也会触发该任务。

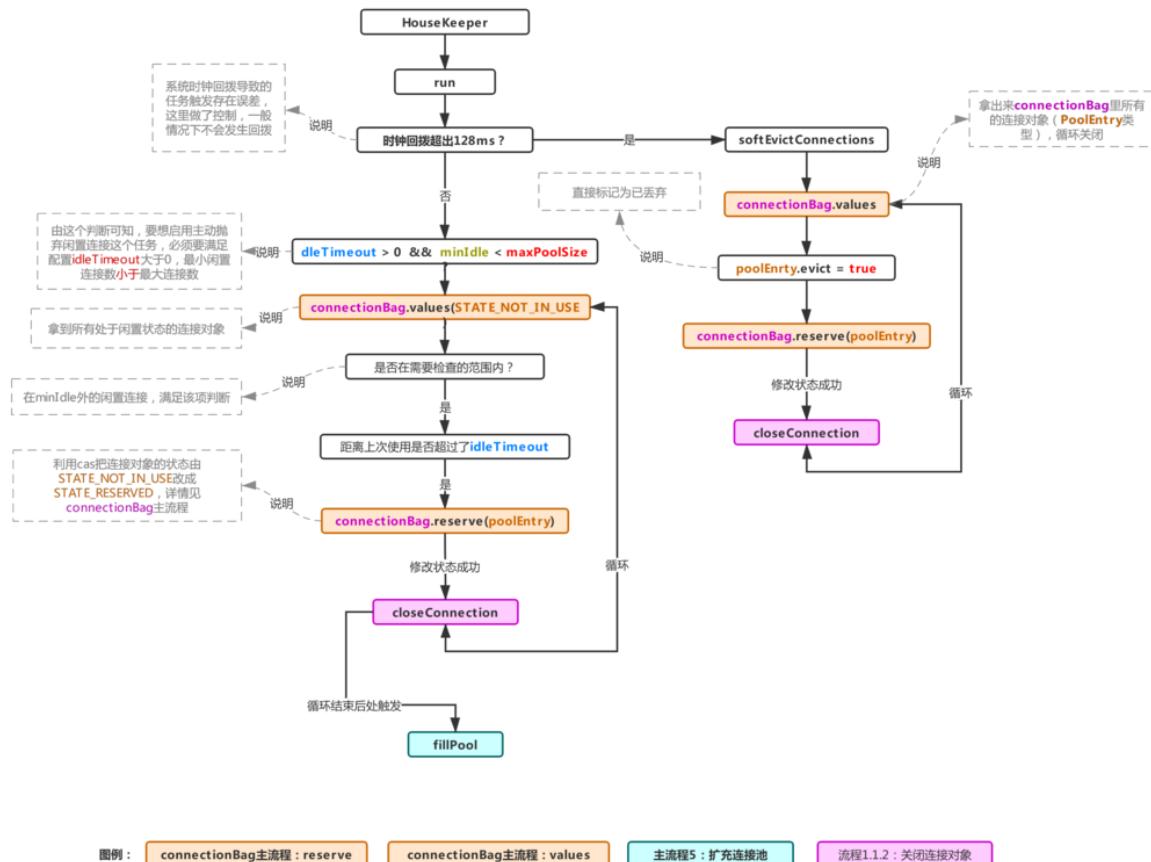
九、主流程4：连接池缩容

HikariCP会按照 `minIdle` 定时清理闲置过久的连接，这个定时任务在 `主流程2` 初始化连接池对象时被启用，跟上面的流程一样，也是利用 `houseKeepingExecutorService` 这个线程池对象做该定时任务的执行器。

来看下 `主流程2` 里是怎么启用该任务的：

```
//housekeepingPeriodMs的默认值是30s，所以定时任务的间隔为30s
this.houseKeeperTask = houseKeepingExecutorService.scheduleWithFixedDelay(new HouseKeeper(), 100L,
housekeepingPeriodMs, MILLISSECONDS);
```

那么本节主要来说下 `HouseKeeper` 这个类，该类实现了 `Runnable` 接口，回收逻辑主要在其 `run` 方法内，来看看 `run` 方法的逻辑流程图：



图例： `connectionBag` 主流程 : `reserve` `connectionBag` 主流程 : `values` `主流程5` : 扩充连接池 `流程1.1.2` : 关闭连接对象

主流程4：连接池缩容

上面的流程就是 `HouseKeeper` 的 `run` 方法里具体做的事情，由于系统时间回拨会导致该定时任务回收一些连接时产生误差，因此存在如下判断：

```
//now就是当前系统时间，previous就是上次触发该任务时的时间，housekeepingPeriodMs就是隔多久触发该任务一次
//也就是说plusMillis(previous, housekeepingPeriodMs)表示当前时间
//如果系统时间没被回拨，那么plusMillis(now, 128)一定是大于当前时间的，如果被系统时间被回拨
//回拨的时间超过128ms，那么下面的判断就成立，否则永远不会成立
if (plusMillis(now, 128) < plusMillis(previous, housekeepingPeriodMs))
```

这是hikariCP在解决系统时钟被回拨时做出的一种措施，通过流程图可以看到，它是直接把池子里所有的连接对象取出来挨个儿的标记成废弃，并且尝试把状态值修改为 `STATE_RESERVED`（后面会说明这些状态，这里不深入）。如果系统时钟没有发生改变（绝大多数情况会命中这一块的逻辑），由图知，会把当前池内所有处于闲置状态（`STATE_NOT_IN_USE`）的连接拿出来，然后计算需要检查的范围，然后循环着修改连接的状态：

```

//拿到所有处于闲置状态的连接
final List<Object> notInUse = connectionBag.values(STATE_NOT_IN_USE);
//计算出需要被检查闲置时间的数量，简单来说，池内需要保证最小minIdle个连接活着，所以需要计算出超出这个范围的闲置对象进行检查
int toRemove = notInUse.size() - config.getMinIdle();
for (PoolEntry entry : notInUse) {
    //在检查范围内，且闲置时间超出idleTimeout，然后尝试将连接对象状态由STATE_NOT_IN_USE变为STATE_RESERVED成功
    if (toRemove > 0 && elapsedMillis(entry.lastAccessed, now) > idleTimeout &&
        connectionBag.reserve(entry)) {
        closeConnection(entry, "(connection has passed idleTimeout)"); //满足上述条件，进行连接关闭
        toRemove--;
    }
}
fillPool(); //因为可能回收了一些连接，所以要再次触发连接池扩充流程检查下是否需要新增连接。

```

上面的代码就是流程图里对应的没有回拨系统时间时的流程逻辑。该流程在 `idleTimeout` 大于0（默认等于0）并且 `minIdle` 小于 `maxPoolSize` 的时候才会启用，默认是不启用的，若需要启用，可以按照条件来配置。

十、主流程5：扩充连接池

这个流程主要依附HikariPool里的 `fillPool` 方法，这个方法已经在上面很多流程里出现过了，它的作用就是在触发连接废弃、连接池连接不够用时，发起扩充连接数的操作，这是个很简单的过程，下面看下源码（为了使代码结构更加清晰，对源码做了细微改动）：

```

// PoolEntryCreator关于call方法的实现流程在主流程3里已经看过了，但是这里却有俩PoolEntryCreator对象，
// 这是个较细节的地方，用于打日志用，不再说这部分，为了便于理解，只需要知道这俩对象执行的是同一块call方法即可
private final PoolEntryCreator poolEntryCreator = new PoolEntryCreator(null);
private final PoolEntryCreator postFillPoolEntryCreator = new PoolEntryCreator("After adding ");

private synchronized void fillPool() {
    // 这个判断就是根据当前池子里相关数据，推算出需要扩充的连接数，
    // 判断方式就是利用最大连接数跟当前连接总数的差值，与最小连接数与当前池内闲置的连接数的差值，取其最小的那个得到
    int needAdd = Math.min(maxPoolSize - connectionBag.size(),
                           minIdle - connectionBag.getCount(STATE_NOT_IN_USE));

    //减去当前排队的任务，就是最终需要新增的连接数
    final int connectionsToAdd = needAdd - addConnectionQueue.size();
    for (int i = 0; i < connectionsToAdd; i++) {
        //一般循环的最后一次会命中postFillPoolEntryCreator任务，其实就是在最后一次会打印一次日志而已（可以忽略该干扰逻辑）
        addConnectionExecutor.submit((i < connectionsToAdd - 1) ? poolEntryCreator :
            postFillPoolEntryCreator);
    }
}

```

由该过程可知，最终这个后叙任务是交由 `addConnectionExecutor` 线程池来处理的，而任务的主题也是 `PoolEntryCreator`，这个流程可以参考 [主流程3](#)。

然后 `needAdd` 的推算：

`Math.min(最大连接数 - 池内当前连接总数, 最小连接数 - 池内闲置的连接数)`

根据这种方式判断，可以保证池内的连接数永远不会超过 `maxPoolSize`，也永远不会低于 `minIdle`。在连接吃紧的时候，可以保证每次触发都以 `minIdle` 的数量扩容。因此如果在 `maxPoolSize` 跟 `minIdle` 配置的值一样的话，在池内连接吃紧的时候，就不会发生任何扩容了。

十一、主流程6：连接回收

最开始说过，最终真实的物理连接对象会被包装成 `PoolEntry` 对象，存放进 `ConcurrentBag`，然后获取时，`PoolEntry` 对象又会被再次包装成 `ProxyConnection` 对象暴露给使用方的，那么触发连接回收，实际上就是触发 `ProxyConnection` 里的 `close` 方法：

```
public final void close() throws SQLException {
    // 原注释: Closing statements can cause connection eviction, so this must run before the conditional below
    closeStatements(); //此连接对象在业务方使用过程中产生的所有statement对象，进行统一close，防止漏close的情况
    if (delegate != ClosedConnection.CLOSED_CONNECTION) {
        leakTask.cancel(); //取消连接泄漏检查任务，参考流程2.2
        try {
            if (isCommitStateDirty && !isAutoCommit) { //在存在执行语句后并且还打开了事务，调用close时需要主动回滚事务
                delegate.rollback(); //回滚
                lastAccess = currentTime(); //刷新“最后一次使用时间”
            }
        } finally {
            delegate = ClosedConnection.CLOSED_CONNECTION;
            poolEntry.recycle(lastAccess); //触发回收
        }
    }
}
```

这个就是 `ProxyConnection` 里的 `close` 方法，可以看到它最终会调用 `PoolEntry` 的 `recycle` 方法进行回收，除此之外，连接对象的最后一次使用时间也是在这个时候刷新的，该时间是个很重要的属性，可以用来判断一个连接对象的闲置时间，来看下 `PoolEntry` 的 `recycle` 方法：

```
void recycle(final long lastAccessed) {
    if (connection != null) {
        this.lastAccessed = lastAccessed; //刷新最后使用时间
        hikariPool.recycle(this); //触发HikariPool的回收方法，把自己传过去
    }
}
```

之前有说过，每个 `PoolEntry` 对象都持有 `HikariPool` 的对象，方便触发连接池的一些操作，由上述代码可以看到，最终还是会触发 `HikariPool` 里的 `recycle` 方法，再来看下 `HikariPool` 的 `recycle` 方法：

```
void recycle(final PoolEntry poolEntry) {
    metricsTracker.recordConnectionUsage(poolEntry); //监控指标相关，忽略
    connectionBag.requite(poolEntry); //最终触发connectionBag的requite方法归还连接，该流程参考ConnectionBag主流程里的requite方法部分
}
```

以上就是连接回收部分的逻辑，相比其他流程，还是比较简洁的。

十二、ConcurrentBag主流程

这个类用来存放最终的PoolEntry类型的连接对象，提供了基本的增删查的功能，被HikariPool持有，上面那么多的操作，几乎都是在HikariPool中完成的，HikariPool用来管理实际的连接生产动作和回收动作，实际操作的却是ConcurrentBag类，梳理下上面所有流程的触发点：

- 主流程2：初始化HikariPool时初始化 ConcurrentBag（构造方法），预热时通过 createEntry 拿到连接对象，调用 ConcurrentBag.add 添加连接到ConcurrentBag。
- 流程1.1：通过HikariPool获取连接时，通过调用 ConcurrentBag.borrow 拿到一个连接对象。
- 主流程6：通过 ConcurrentBag.requite 归还一个连接。
- 流程1.1.2：触发关闭连接时，会通过 ConcurrentBag.remove 移除连接对象，由前面的流程可知关闭连接触发点为：连接超过最大生命周期maxLifeTime主动废弃、健康检查不通过主动废弃、连接池缩容。
- 主流程3：通过异步添加连接时，通过调用 ConcurrentBag.add 添加连接到ConcurrentBag，由前面的流程可知添加连接触发点为：连接超过最大生命周期maxLifeTime主动废弃连接后、连接池扩容。
- 主流程4：连接池缩容任务，通过调用 ConcurrentBag.values 筛选出需要的操作的连接对象，然后再通过 ConcurrentBag.reserve 完成对连接对象状态的修改，然后会通过 流程1.1.2 触发关闭和移除连接操作。

通过触发点整理，可以知道该结构里的主要方法，就是上面触发点里标记为 标签色 的部分，然后来具体看下该类的基本定义和主要方法：

```
public class ConcurrentBag<T extends IConcurrentBagEntry> implements AutoCloseable {

    private final CopyOnWriteArrayList<T> sharedList; //最终存放PoolEntry对象的地方，它是一个
    CopyOnWriteArrayList
    private final boolean weakThreadLocals; //默认false，为true时可以让一个连接对象在下方threadList里的
    list内处于弱引用状态，防止内存泄漏（参见备注1）

    private final ThreadLocal<List<Object>> threadList; //线程级的缓存，从sharedList拿到的连接对象，会
    被缓存进当前线程内，borrow时会先从缓存中拿，从而达到池内无锁实现
    private final IBagStateListener listener; //内部接口，HikariPool实现了该接口，主要用于
    ConcurrentBag主动通知HikariPool触发添加连接对象的异步操作（也就是主流程3里的addConnectionExecutor所触发
    的流程）
    private final AtomicInteger waiters; //当前因为获取不到连接而发生阻塞的业务线程数，这个在之前的流
    程里也出现过，比如主流程3里addBagItem就会根据该指标进行判断是否需要新增连接
    private volatile boolean closed; //标记当前ConcurrentBag是否已被关闭

    private final SynchronousQueue<T> handoffQueue; //这是个即产即销的队列，用于在连接不够用时，及时获
    取到add方法里新创建的连接对象，详情可以参考下面borrow和add的代码

    //内部接口，PoolEntry类实现了该接口
    public interface IConcurrentBagEntry {

        //连接对象的状态，前面的流程很多地方都已经涉及到了，比如主流程4的缩容
        int STATE_NOT_IN_USE = 0; //闲置
        int STATE_IN_USE = 1; //使用中
        int STATE_REMOVED = -1; //已废弃
        int STATE_RESERVED = -2; //标记保留，介于闲置和废弃之间的中间状态，主要由缩容那里触发修改

        boolean compareAndSet(int expectState, int newState); //尝试利用cas修改连接对象的状态值
        void setState(int newState); //设置状态值

        int getState(); //获取状态值
    }
}
```

```

//参考上面listener属性的解释
public interface IBagStateListener {
    void addBagItem(int waiting);
}

//获取连接方法
public T borrow(long timeout, final TimeUnit timeUnit) {
    // 省略...
}

//回收连接方法
public void requite(final T bagEntry) {
    //省略...
}

//添加连接方法
public void add(final T bagEntry) {
    //省略...
}

//移除连接方法
public boolean remove(final T bagEntry) {
    //省略...
}

//根据连接状态值获取当前池子内所有符合条件的连接集合
public List values(final int state) {
    //省略...
}

//获取当前池子内所有的连接
public List values() {
    //省略...
}

//利用cas把传入的连接对象的state从 STATE_NOT_IN_USE 变为 STATE_RESERVED
public boolean reserve(final T bagEntry) {
    //省略...
}

//获取当前池子内符合传入状态值的连接数量
public int getCount(final int state) {
    //省略...
}
}

```

从这个基本结构就可以稍微看出HikariCP是如何优化传统连接池实现的了，相比Druid来说，HikariCP更加偏向无锁实现，尽量避免锁竞争的发生。

12.1: borrow

这个方法是HikariCP一个重要的接口方法，触发上为流程1.1，HikariPool就是利用该方法获取连接的，下面来看下该方法做了什么：

```

public T borrow(long timeout, final TimeUnit timeUnit) throws InterruptedException {
    // 源注释: Try the thread-local list first
}

```

```

    final List<Object> list = threadList.get(); //首先从当前线程的缓存里拿到之前被缓存进来的连接对象集合
    for (int i = list.size() - 1; i >= 0; i--) {
        final Object entry = list.remove(i); //先移除，回收方法那里会再次add进来
        final T bagEntry = weakThreadLocals ? ((WeakReference<T>) entry).get() : (T) entry; //默认不启用弱引用
        // 获取到对象后，通过cas尝试把其状态从STATE_NOT_IN_USE 变为 STATE_IN_USE，注意，这里如果其他线程也在使用这个连接对象，
        // 并且成功修改属性，那么当前线程的cas会失败，那么就会继续循环尝试获取下一个连接对象
        if (bagEntry != null && bagEntry.compareAndSet(STATE_NOT_IN_USE, STATE_IN_USE)) {
            return bagEntry; //cas设置成功后，表示当前线程绕过其他线程干扰，成功获取到该连接对象，直接返回
        }
    }

    // 源注释：Otherwise, scan the shared list ... then poll the handoff queue
    final int waiting = waiters.incrementAndGet(); //如果缓存内找不到一个可用的连接对象，则认为需要“回源”，waiters+1
    try {
        for (T bagEntry : sharedList) {
            //循环sharedList，尝试把连接状态值从STATE_NOT_IN_USE 变为 STATE_IN_USE
            if (bagEntry.compareAndSet(STATE_NOT_IN_USE, STATE_IN_USE)) {
                // 源注释：If we may have stolen another waiter's connection, request another bag add.
                if (waiting > 1) { //阻塞线程数大于1时，需要触发HikariPool的addBagItem方法来进行添加连接入池，这个方法的实现参考主流程
                    listener.addBagItem(waiting - 1);
                }
                return bagEntry; //cas设置成功，跟上面的逻辑一样，表示当前线程绕过其他线程干扰，成功获取到该连接对象，直接返回
            }
        }
    }

    //走到这里说明不光线程缓存里的列表竞争不到连接对象，连sharedList里也找不到可用的连接，这时则认为需要通知HikariPool，该触发添加连接操作了
    listener.addBagItem(waiting);

    timeout = timeUnit.toNanos(timeout); //这时候开始利用timeout控制获取时间
    do {
        final long start = currentTime();
        //尝试从handoffQueue队列里获取最新被加进来的连接对象（一般新入的连接对象除了加进sharedList之外，还会被offer进该队列）
        final T bagEntry = handoffQueue.poll(timeout, NANoseconds);
        //如果超出指定时间后仍然没有获取到可用的连接对象，或者获取到对象后通过cas设置成功，这两种情况都不需要重试，直接返回对象
        if (bagEntry == null || bagEntry.compareAndSet(STATE_NOT_IN_USE, STATE_IN_USE)) {
            return bagEntry;
        }
        //走到这里说明从队列内获取到了连接对象，但是cas设置失败，说明又该对象又被其他线程率先拿去用了，若时间还够，则再次尝试获取
        timeout -= elapsedNanos(start); //timeout减去消耗的时间，表示下次循环可用时间
    } while (timeout > 10_000); //剩余时间大于10s时才继续进行，一般情况下，这个循环只会走一次，因为timeout至少会自己比10s大
    return null; //超时，仍然返回null
} finally {
    waiters.decrementAndGet(); //这一步出去后，HikariPool收到borrow的结果，算是走出阻塞，所以waiters-1
}

```

```
    }  
}
```

仔细看下注释，该过程大致分成三个主要步骤：

1. 从线程缓存获取连接
2. 获取不到再从 `sharedList` 里获取
3. 都获取不到则触发添加连接逻辑，并尝试从队列里获取新生成的连接对象

12.2: add

这个流程会添加一个连接对象进入bag，通常由 `主流程3` 里的 `addBagItem` 方法通过 `addConnectionExecutor` 异步任务触发添加操作，该方法主流程如下：

```
public void add(final T bagEntry) {  
  
    sharedList.add(bagEntry); //直接加到sharedList里去  
  
    // 源注释：spin until a thread takes it or none are waiting  
    // 参考borrow流程，当存在线程等待获取可用连接，并且当前新入的这个连接状态仍然是闲置状态，且队列里  
    // 无消费者等待获取时，发起一次线程调度  
    while (waiters.get() > 0 && bagEntry.getState() == STATE_NOT_IN_USE &&  
        !handoffQueue.offer(bagEntry)) { //注意这里会offer一个连接对象入队列  
        yield();  
    }  
}
```

结合 `borrow` 来理解的话，这里在存在等待线程时会添加一个连接对象入队列，可以让 `borrow` 里发生等待的地方更容易 `poll` 到这个连接对象。

12.3: requite

这个流程会回收一个连接，该方法的触发点在 `主流程6`，具体代码如下：

```
public void requite(final T bagEntry) {  
    bagEntry.setState(STATE_NOT_IN_USE); //回收意味着使用完毕，更改state为STATE_NOT_IN_USE状态  
  
    for (int i = 0; waiters.get() > 0; i++) { //如果存在等待线程的话，尝试传给队列，让borrow获取  
        if (bagEntry.getState() != STATE_NOT_IN_USE || handoffQueue.offer(bagEntry)) {  
            return;  
        }  
        else if ((i & 0xff) == 0xff) {  
            parkNanos(MICROSECONDS.toNanos(10));  
        }  
        else {  
            yield();  
        }  
    }  
  
    final List<Object> threadLocalList = threadList.get();  
    if (threadLocalList.size() < 50) { //线程内连接集合的缓存最多50个，这里回收连接时会再次加进当前线  
        //程的缓存里，下次再获  
        threadLocalList.add(weakThreadLocals ? new WeakReference<T>(bagEntry) : bagEntry); //默认不启  
        //用弱引用，若启用的话，则缓存集合里的连接对象没有内存泄露的风险  
    }  
}
```

12.4: remove

这个负责从池子里移除一个连接对象，触发点在 [流程1.1.2](#)，代码如下：

```
public boolean remove(final T bagEntry) {
    // 下面两个cas操作，都是从其他状态变为移除状态，任意一个成功，都不会走到下面的warn log
    if (!bagEntry.compareAndSet(STATE_IN_USE, STATE_REMOVED) &&
        !bagEntry.compareAndSet(STATE_RESERVED, STATE_REMOVED) && !closed) {
        LOGGER.warn("Attempt to remove an object from the bag that was not borrowed or reserved: {}", bagEntry);
        return false;
    }

    // 直接从sharedList移除掉
    final boolean removed = sharedList.remove(bagEntry);
    if (!removed && !closed) {
        LOGGER.warn("Attempt to remove an object from the bag that does not exist: {}", bagEntry);
    }

    return removed;
}
```

这里需要注意的是，移除时仅仅移除了 `sharedList` 里的对象，各个线程内缓存的那一份集合里对应的对象并没有被移除，这个时候会不会存在该连接再次从缓存里拿到呢？会的，但是不会返回出去，而是直接 `remove` 掉了，仔细看 `borrow` 的代码发现状态不是闲置状态的时候，取出来时就会 `remove` 掉，然后也拿不出去，自然也不会触发回收方法。

12.5: values

该方法存在重载方法，用于返回当前池子内连接对象的集合，触发点在 [主流程4](#)，代码如下：

```
public List values(final int state) {
    //过滤出来符合状态值的对象集合逆序后返回出去
    final List list = sharedList.stream().filter(e -> e.getState() == state).collect(Collectors.toList());
    Collections.reverse(list);
    return list;
}

public List values() {
    //返回全部连接对象（注意下方clone为浅拷贝）
    return (List) sharedList.clone();
}
```

12.6: reserve

该方法单纯将连接对象的状态值由 `STATE_NOT_IN_USE` 修改为 `STATE_RESERVED`，触发点仍然是 [主流程4](#)，缩容时使用，代码如下：

```
public boolean reserve(final T bagEntry) {
    return bagEntry.compareAndSet(STATE_NOT_IN_USE, STATE_RESERVED);
}
```

12.7: getCount

该方法用于返回池内符合某个状态值的连接的总数量，触发点为 [主流程5](#)，扩充连接池时用于获取闲置连接总数，代码如下：

```
public int getCount(final int state){  
    int count = 0;  
    for (IConcurrentBagEntry e : sharedList) {  
        if (e.getState() == state) {  
            count++;  
        }  
    }  
    return count;  
}
```

以上就是 [ConcurrentBag](#) 的主要方法和处理连接对象的主要流程。

十三、总结

到这里基本上一个连接的生产到获取到回收到废弃一整个生命周期在HikariCP内是如何管理的就說完了，相比之前的Druid的实现，有很大的不同，主要是HikariCP的 [无锁](#) 获取连接，本篇没有涉及 [FastList](#) 的说明，因为从连接管理这个角度确实很少用到该结构，用到 [FastList](#) 的地方主要在存储连接对象生成的 [statement对象](#) 以及用于存储线程内缓存起来的连接对象；

除此之外HikariCP还利用 [javassist](#) 技术编译期生成了 [ProxyConnection](#) 的初始化，这里也没有相关说明，网上有关HikariCP的优化有很多文章，大多数都提到了 [字节码优化](#)、[fastList](#)、[concurrentBag](#) 的实现，本篇主要通过深入解析 [HikariPool](#) 和 [ConcurrentBag](#) 的实现，来说明HikariCP相比Druid具体做了哪些不一样的操作。

Spring Boot 这样优化，让你的项目飞起来！

介绍

在SpringBoot的Web项目中，默认采用的是内置Tomcat，当然也可以配置支持内置的jetty，内置有什么好处呢？

1. 方便微服务部署。
2. 方便项目启动，不需要下载Tomcat或者Jetty

针对目前的容器优化，目前来说没有太多地方，需要考虑以下几个点

- 线程数
- 超时时间
- jvm优化

针对上面的优化项，首先 我们以一个重点 [初始线程数和最大线程数](#)，初始线程数保障启动的时候，如果有大量用户访问，能够很稳定的接受请求。

而最大线程数量用来保证系统的稳定性，而超时时间用来保障连接数不容易被压垮，如果大批量的请求过来，延迟比较高，不容易把线程打满。这种情况在生产中是比较常见的，一旦网络不稳定，宁愿丢包也不愿意把机器压垮。

jvm优化一般来说没有太多场景，无非就是加大初始的堆，和最大限制堆,当然也不是无限增大，根据的情况进快速开始

在spring boot配置文件中application.yml，添加以下配置

```
server:  
  tomcat:  
    min-spare-threads: 20  
    max-threads: 100  
    connection-timeout: 5000
```

这块对tomcat进行了一个优化配置，最大线程数是100，初始化线程是20,超时时间是5000ms

Jvm优化

这块主要不是谈如何优化，jvm优化是一个需要场景化的，没有什么太多特定参数，一般来说在server端运行都会指定如下参数

初始内存和最大内存基本会设置成一样的，具体大小根据场景设置，-server是一个必须要用的参数，至于收集器这些使用默认的就可以了，除非有特定需求。

1. 使用-server模式

设置JVM使用server模式。64位JDK默认启动该模式

```
java -server -jar springboot-1.0.jar
```

2. 指定堆参数

这个根据服务器的内存大小，来设置堆参数。

- -Xms :设置Java堆栈的初始化大小
- -Xmx :设置最大的java堆大小

```
java -server -Xms512m -Xmx768m -jar springboot-1.0.jar
```

设置初始化堆内存为512MB，最大为768MB。

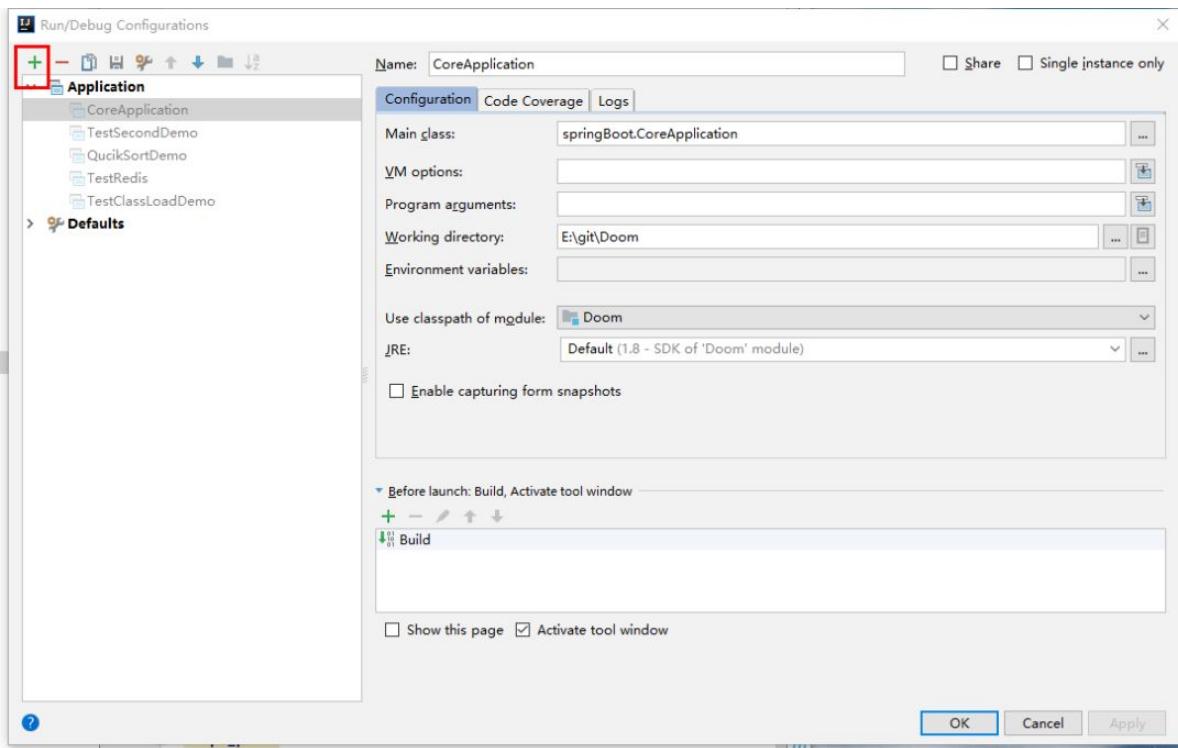
3. 远程Debug

在服务器上将启动参数修改为：

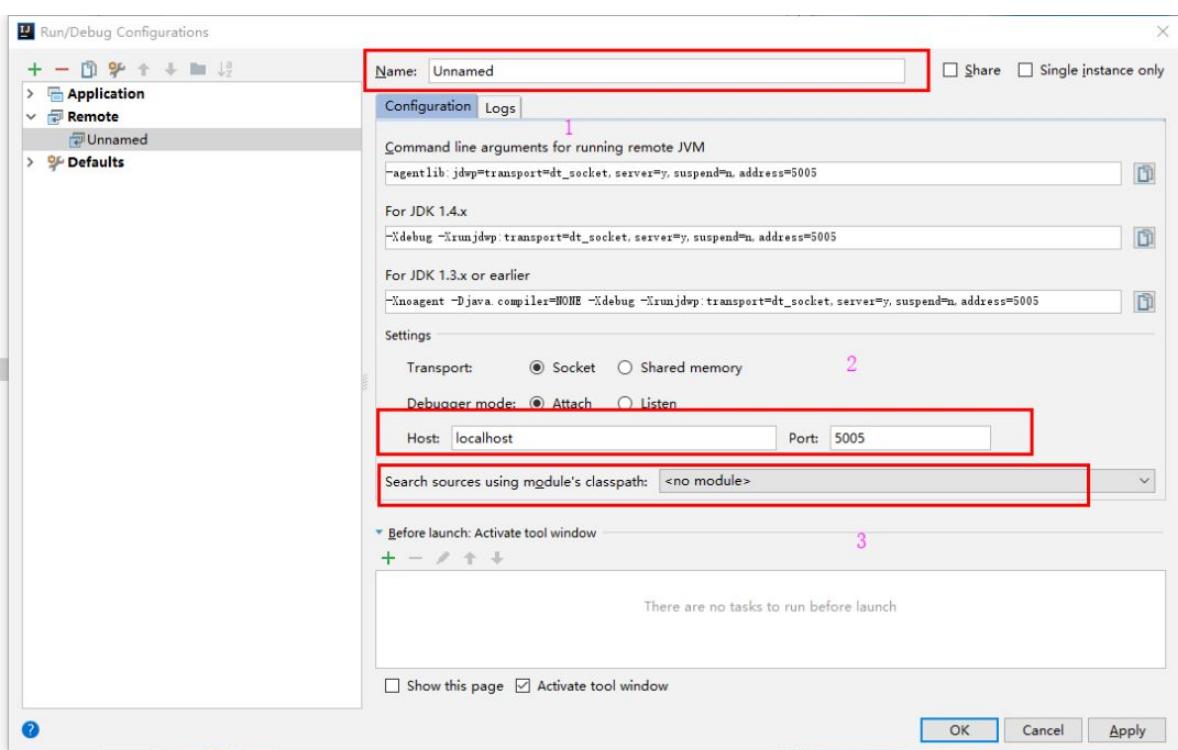
```
java -Djavax.net.debug=  
ssl -Xdebug -Xnoagent -Djava.compiler=  
NONE -Xrunjdwp:transport=  
dt_socket,server=y,suspend=  
n,address=8888 -jar springboot-1.0.jar
```

这个时通过远程JDWP模式启动，端口号为8888。

在IDEA中，点击Edit Configuration按钮。



出现弹窗，点击+按钮，找到Remote选项。

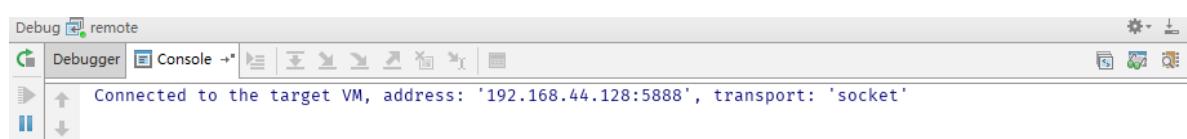


在【1】中填入Remote项目名称，在【2】中填IP地址和端口号，在【3】选择远程调试的项目module，配置完成后点击OK即可

如果碰到连接超时的情况，很有可能服务器的防火墙的问题，举例CentOs7,关闭防火墙

```
systemctl stop firewalld.service #停止firewall
systemctl disable firewalld.service #禁止firewall开机启动
```

Watermark
点击debug 安装IDEA远程调试 打印言



说明远程调试成功。

JVM工具远程连接

jconsole与Jvisualvm远程连接

通常我们的web服务都部署在服务器上的，在window使用jconsole是很方便的，相对于Linux就有一些麻烦了，需要进行一些设置。

1.查看hostname,首先使用

```
hostname -i
```

查看，服务器的hostname为127.0.0.1，这个是不对的，需要进行修改

2.修改hostname

修改/etc/hosts文件，将其第一行的“127.0.0.1 localhost.localdomain localhost”，修改为：“192.168.44.128 localhost.localdomain localhost”。“192.168.44.128”为实际的服务器的IP地

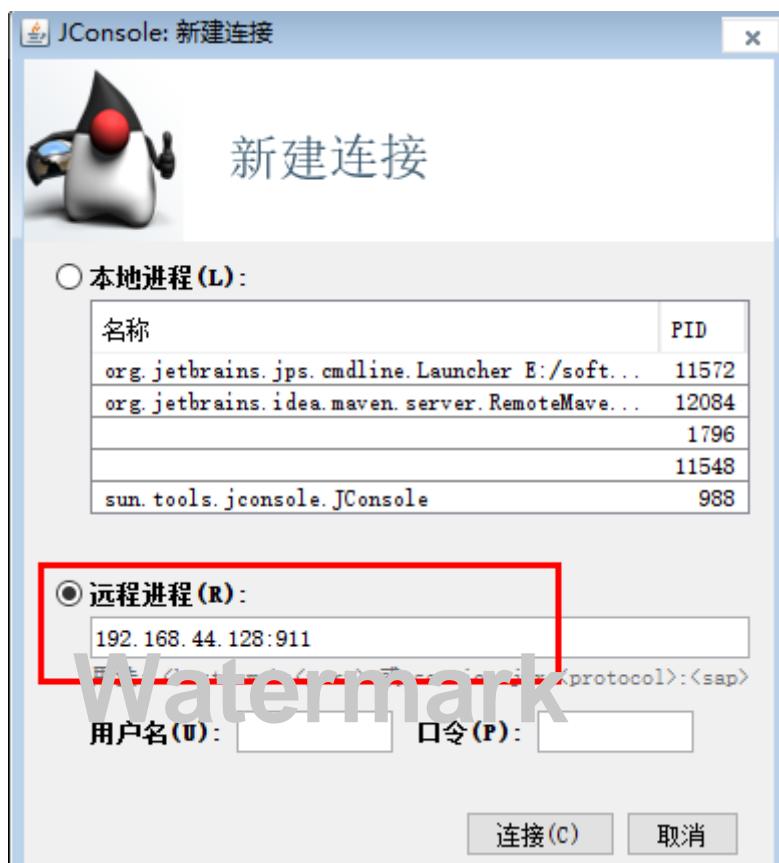
3.重启Linux，在服务器上输入hostname -i，查看实际设置的IP地址是否为你设置的

4.启动服务，参数为：

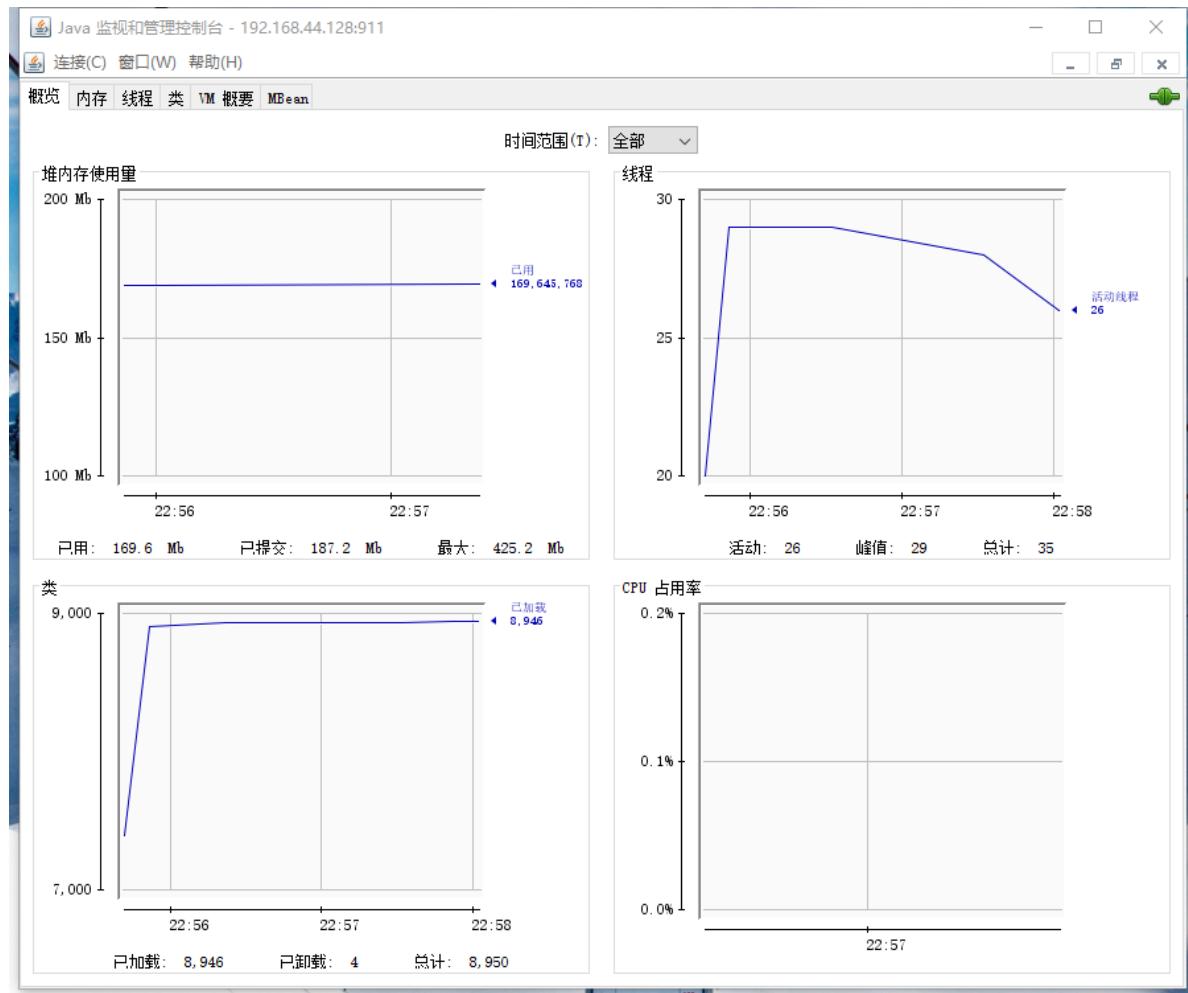
```
java -jar -Djava.rmi.server.hostname=192.168.44.128 -  
Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=911 -  
Dcom.sun.management.jmxremote.ssl=false -  
Dcom.sun.management.jmxremote.authenticate=false jantent-1.0-SNAPSHOT.jar
```

ip为192.168.44.128，端口为911。

5.打开Jconsole，进行远程连接，输入IP和端口即可

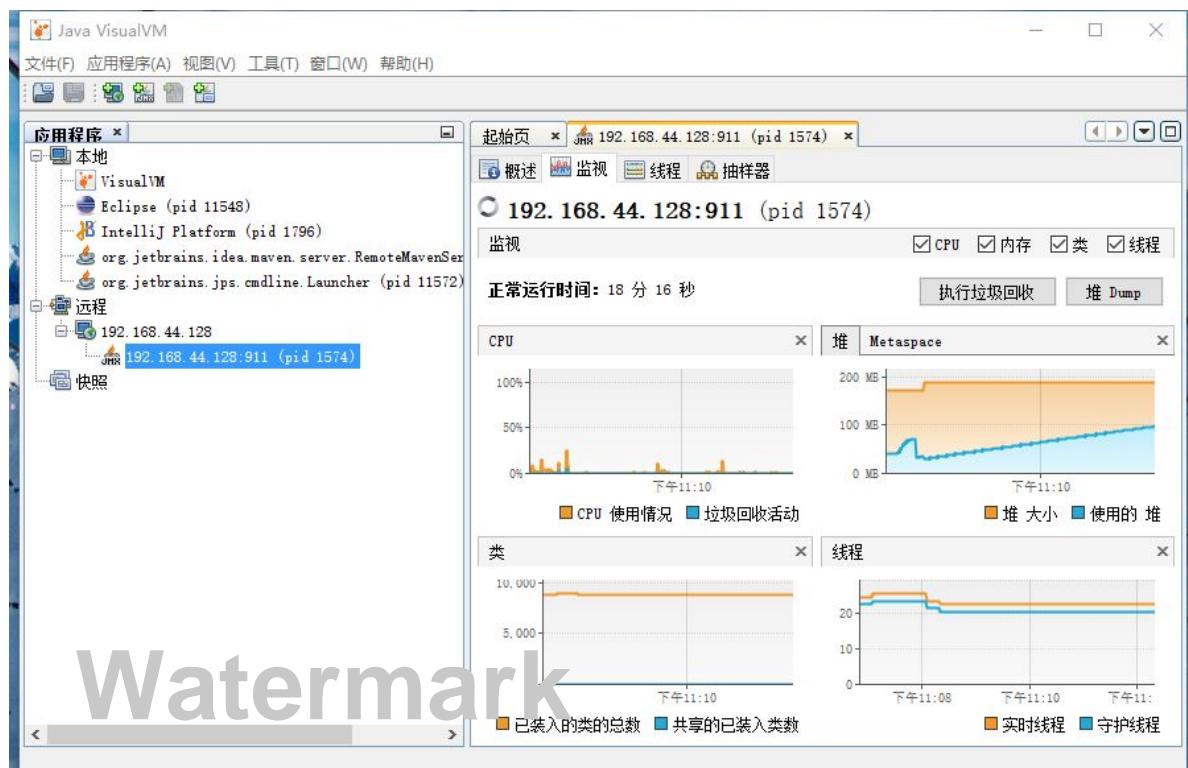


点击连接，经过稍稍等待之后，即可完成连接，如下图所示：



同理，JvisualVm的远程连接是同样的，启动参数也是一样。

然后在本机JvisualVm输入IP: PORT，即可进行远程连接：如下图所示：



相比较Jvisualvm功能更加强大一下，界面也更美观。

Spring Boot 这样做可视化监控，一目了然！

在本文中，我们来学习使用Spring Actuator, Micrometer, Prometheus和Grafana监控Spring Boot应用程序。你可能觉得这需要大量工作，但是其实很容易！

1、简介

当某个应用程序在生产环境中运行时，监控其运行状况是必要的。通过实时了解应用程序的运行状况，你能在问题出现之前得到警告，也可以在客户注意到问题之前解决问题。在本文中，我们将创建一个Spring Boot应用程序，在Spring Actuator, Micrometer, Prometheus和Grafana的帮助下监控系统。其中，Spring Actuator和Micrometer是Spring Boot App的一部分。



简要说明了不同组件的目的：

- **Spring Actuator**：在应用程序里提供众多 Web 接口，通过它们了解应用程序运行时的内部状况。有关更多信息，请参见Spring Boot 2.0中的Spring Boot Actuator。
- **Micrometer**：为 Java 平台上的性能数据收集提供了一个通用的 API，它提供了多种度量指标类型（Timers、Gauges、Counters等），同时支持接入不同的监控系统，例如 Influxdb、Graphite、Prometheus 等。Spring Boot Actuator对此提供了支持。
- **Prometheus**：一个时间序列数据库，用于收集指标。
- **Grafana**：用于显示指标的仪表板。

下面，我们将分别介绍每个组件。本文中使用的代码存档在GitHub上。

2、创建示例应用

首先要做的是创建一个可以监控的应用程序。通过 **Spring Initializr**，并添加Spring Boot Actuator, Prometheus和Spring Web依赖项，我们创建了一个如下所示的Spring MVC应用程序。

```
@RestController
public class MetricsController {

    @GetMapping("/endPoint1")
    public String endPoint1() {
        return "Metrics for endPoint1";
    }

    @GetMapping("/endPoint2")
    public String endPoint2() {
        return "Metrics for endPoint2";
    }
}
```

启动应用程序：

```
$ mvn spring-boot:run
```

验证接口是否正常：

```
$ curl http://localhost:8080/endPoint1Metrics for endPoint1$ curl http://localhost:8080/endPoint2Metrics for endPoint2
```

验证Spring Actuator接口。为了使响应信息方便可读，我们通过python -mjson.tool来格式化信息。

```
$ curl http://localhost:8080/actuator | python -mjson.tool
...
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "health-path": {
      "href": "http://localhost:8080/actuator/health/{*path}",
      "templated": true
    },
    "info": {
      "href": "http://localhost:8080/actuator/info",
      "templated": false
    }
  }
}
```

默认情况下，会显示以上信息。除此之外，Spring Actuator可以提供更多信息，但是你需要启用它。为了启用Prometheus，你需要将以下信息添加到application.properties文件中。

```
management.endpoints.web.exposure.include=health, info, prometheus
```

重启应用程序，访问<http://localhost:8080/actuator/prometheus>从Prometheus拉取数据，返回了大量可用的指标信息。我们这里只显示输出的一小部分，因为它是一个很长的列表。

```
$ curl http://localhost:8080/actuator/prometheus
# HELP jvm_gc_pause_seconds Time spent in GC pause
# TYPE jvm_gc_pause_seconds summary
jvm_gc_pause_seconds_count{action="end of minor GC",cause="G1 Evacuation Pause",} 2.0
jvm_gc_pause_seconds_sum{action="end of minor GC",cause="G1 Evacuation Pause",} 0.009
...
```

如前所述，还需要Micrometer。[Micrometer](#)为最流行的监控系统提供了一个简单的仪表板，允许仪表化JVM应用，而无需关心是哪个供应商提供的指标。它的作用和[SLF4J](#)类似，只不过它关注的不是Logging（日志），而是application metrics（应用指标）。简而言之，它就是应用监控界的[SLF4J](#)。

Spring Boot Actuator为Micrometer提供了自动配置。Spring Boot 2在spring-boot-actuator中引入了micrometer对Metrics进行了重构，另外支持对接的监控系统也更加丰富([Atlas](#)、[Datadog](#)、[Ganglia](#)、[Graphite](#)、[Influx](#)、[JMX](#)、[NewRelic](#)、[Prometheus](#)、[SignalFx](#)、[StatsD](#)、[Wavefront](#))。

更新后的[application.properties](#)文件如下所示：

```
management.endpoints.web.exposure.include=health,info,metrics,prometheus
```

重启应用程序，并从 <http://localhost:8080/actuator/metrics> 中检索数据。

```
$ curl http://localhost:8080/actuator/metrics | python -mjson.tool
...
{
  "names": [
    "http.server.requests",
    "jvm.buffer.count",
    "jvm.buffer.memory.used",
    ...
}
```

可以直接通过指标名来检索具体信息。例如，如果查询 `http.server.requests` 指标，可以按以下方式检索：

```
$ curl http://localhost:8080/actuator/metrics/http.server.requests | python -mjson.tool
...
{
  "name": "http.server.requests",
  "description": null,
  "baseUnit": "seconds",
  "measurements": [
    {
      "statistic": "COUNT",
      "value": 3.0
    },
    {
      "statistic": "TOTAL_TIME",
      "value": 0.08918682
    },
    ...
  ]
##
```

3、添加Prometheus

Prometheus是Cloud Native Computing Foundation的一个开源监控系统。由于我们的应用程序中有一个 </actuator/Prometheus> 端点来供 Prometheus 抓取数据，因此你现在可以配置Prometheus来监控你的Spring Boot应用程序。

Prometheus有几种安装方法，在本文中，我们将在Docker容器中运行Prometheus。

你需要创建一个 `prometheus.yml` 文件，以添加到Docker容器中。

```
global:
  scrape_interval:15s

  scrape_configs:
  - job_name: 'mvspingmetricsplanet'
    metrics_path: '/actuator/prometheus'
    static_configs:
    - targets: ['HOST:8080']
```

- `scrape_interval`：Prometheus多久轮询一次应用程序的指标
- `job_name`：轮询任务名称

- `metrics_path`：指标的URL的路径
- `targets`：主机名和端口号。使用时，替换HOST为主机的IP地址

如果在Linux上查找IP地址有困难，则可以使用以下命令：

```
$ ip -f inet -o addr show docker0 | awk '{print $4}' | cut -d '/' -f 1
```

启动Docker容器并将本地 `prometheus.yml` 文件，映射到Docker容器中的文件。

```
$ docker run \
-p 9090:9090 \
-v /path/to/prometheus.yml:/etc/prometheus/prometheus.yml \
prom/prometheus
```

成功启动Docker容器后，首先验证Prometheus是否能够通过 <http://localhost:9090/targets> 收集数据。

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://[REDACTED]:8080/actuator/prometheus	UP	instance="myspringmetricsplanet"	10m6s	10.0s	

如上图所示，我们遇到 `context deadline exceeded` 错误，造成Prometheus无法访问主机上运行的 Spring Boot应用程序。如何解决呢？

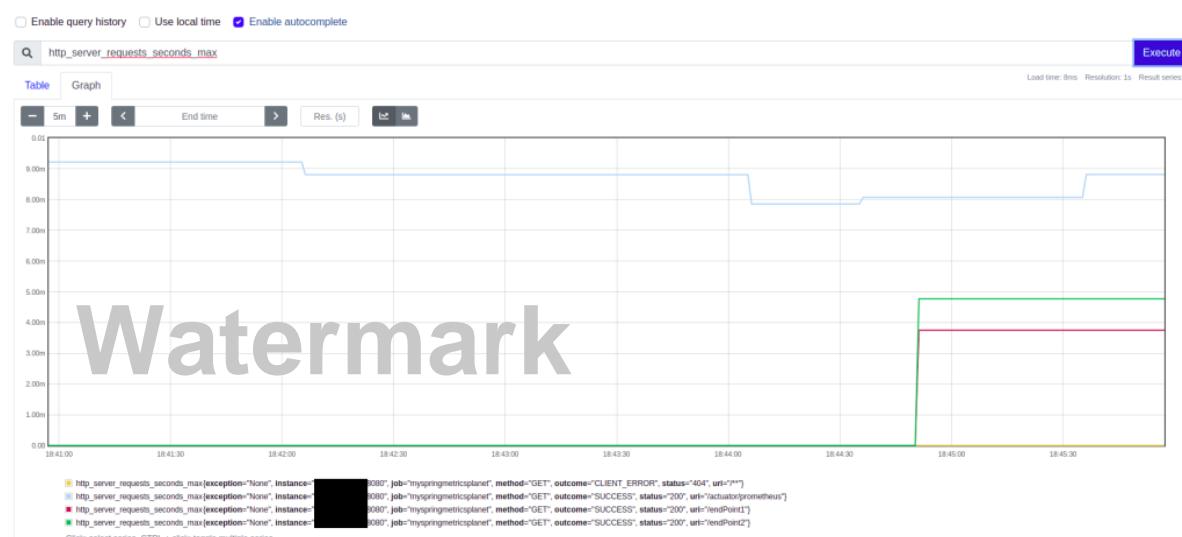
可以通过将Docker容器添加到你的主机网络来解决此错误，这将使Prometheus能够访问Spring Boot 应用程序。

```
$ docker run \
--name prometheus \
--network host \
-v /path/to/prometheus.yml:/etc/prometheus/prometheus.yml \
-d \
prom/prometheus
```

再次验证，状态指示为UP。

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://[REDACTED]:8080/actuator/prometheus	UP	instance="myspringmetricsplanet"	13.651s	16.531ms	

现在可以显示Prometheus指标。通过访问 <http://localhost:9090/graph>，在搜索框中输入 `http_server_requests_seconds_max` 并单击“执行”按钮，将为你提供请求期间的最长执行时间。

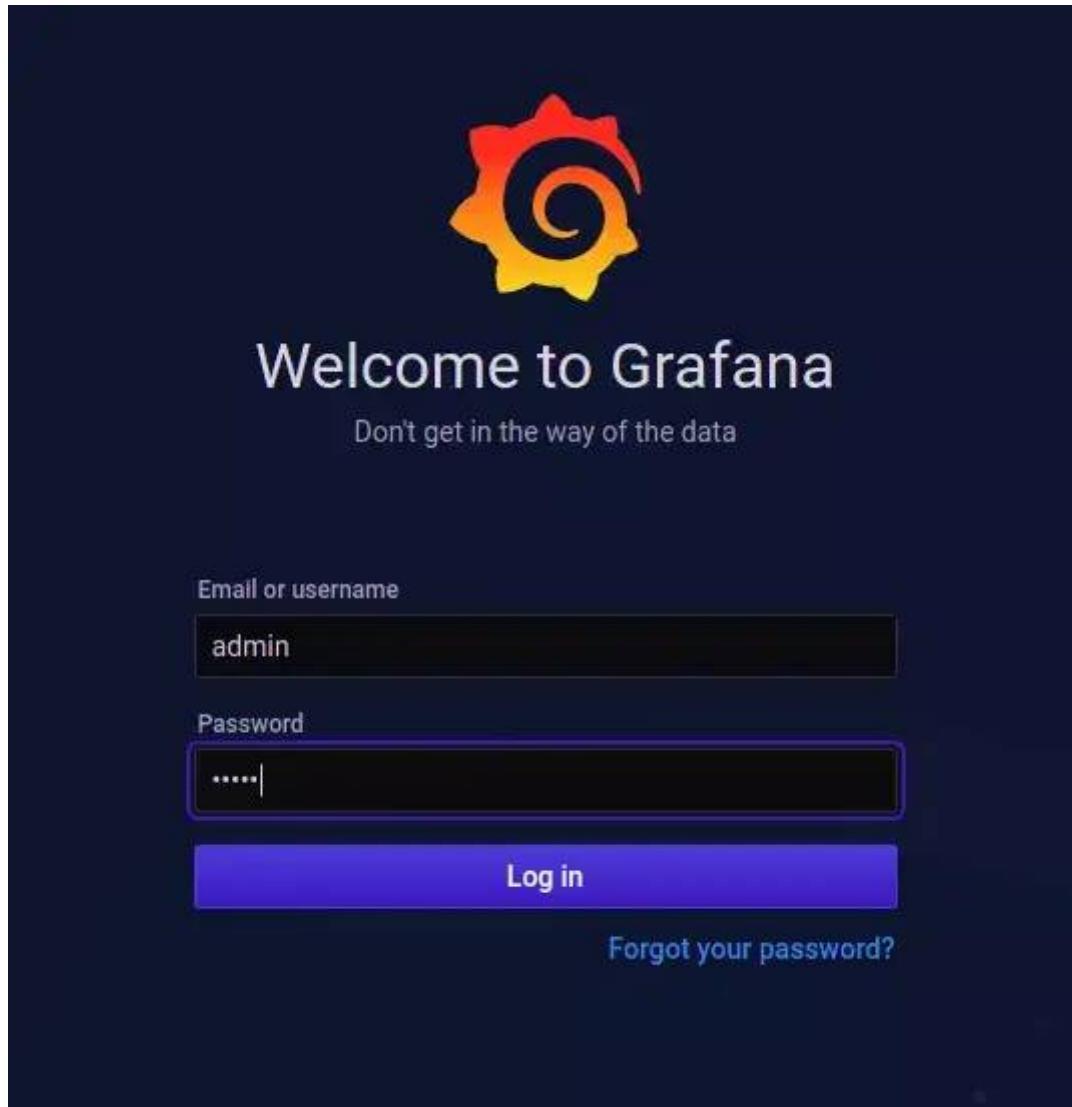


4、添加Grafana

最后添加的组件是Grafana。尽管Prometheus可以显示指标，但Grafana可以帮助你在更精美的仪表板中显示指标。Grafana也支持几种安装方式，在本文中，我们也将在Docker容器中运行它。

```
$ docker run --name grafana -d -p 3000:3000 grafana/grafana
```

点击 <http://localhost:3000/>，就可以访问Grafana。



默认的用户名/密码为 [admin/admin](#)。单击“登录”按钮后，你需要更改默认密码。

Watermark



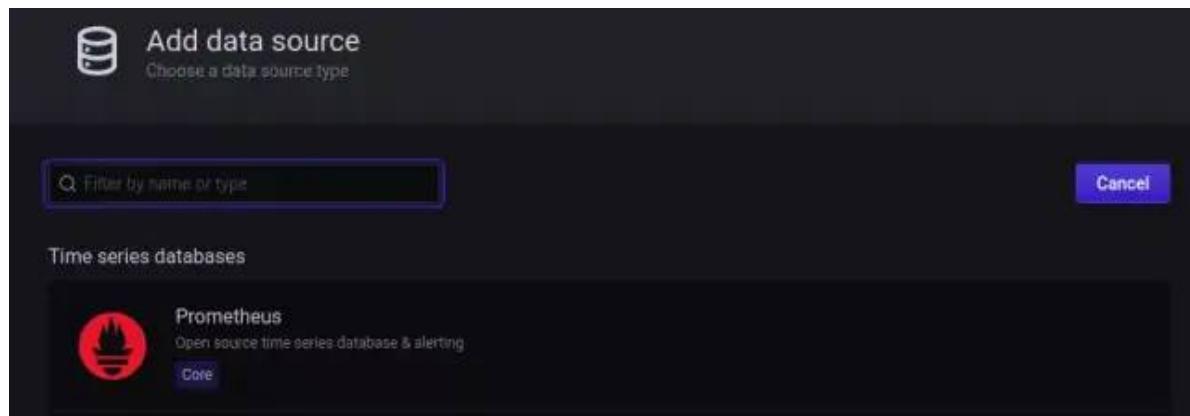
接下来要做的是添加一个数据源。单击左侧边栏中的“配置”图标，然后选择“Data Sources(数据源)”。



单击 Add data source (添加数据源) 按钮。

Prometheus在列表的顶部，选择Prometheus。

Watermark



填写可访问Prometheus的URL，将 `HTTP Access` 设置为 `Browser`，然后单击页面底部的 `Save&Test` 按钮。

Data Sources / Prometheus
Type: Prometheus

Configure your Prometheus data source below

Or skip the effort and get Prometheus (and Loki) as fully managed, scalable and hosted data sources from Grafana Labs with the [free-for-ever Grafana Cloud plan](#).

HTTP

Name: Prometheus Default:

URL: Access: Help

Auth

Basic auth With Credentials

Scrape interval: 1s
Query timeout: 60s
HTTP Method: Choose

Misc

Disable metrics lookup
Custom query parameters: Example: max_source/_resolution=5m&timeout=10s

Exemplars

+ Add

Watermark

Save & Test **Delete** **Back**

一切正常后，将显示绿色的通知标语，指示数据源正在工作。

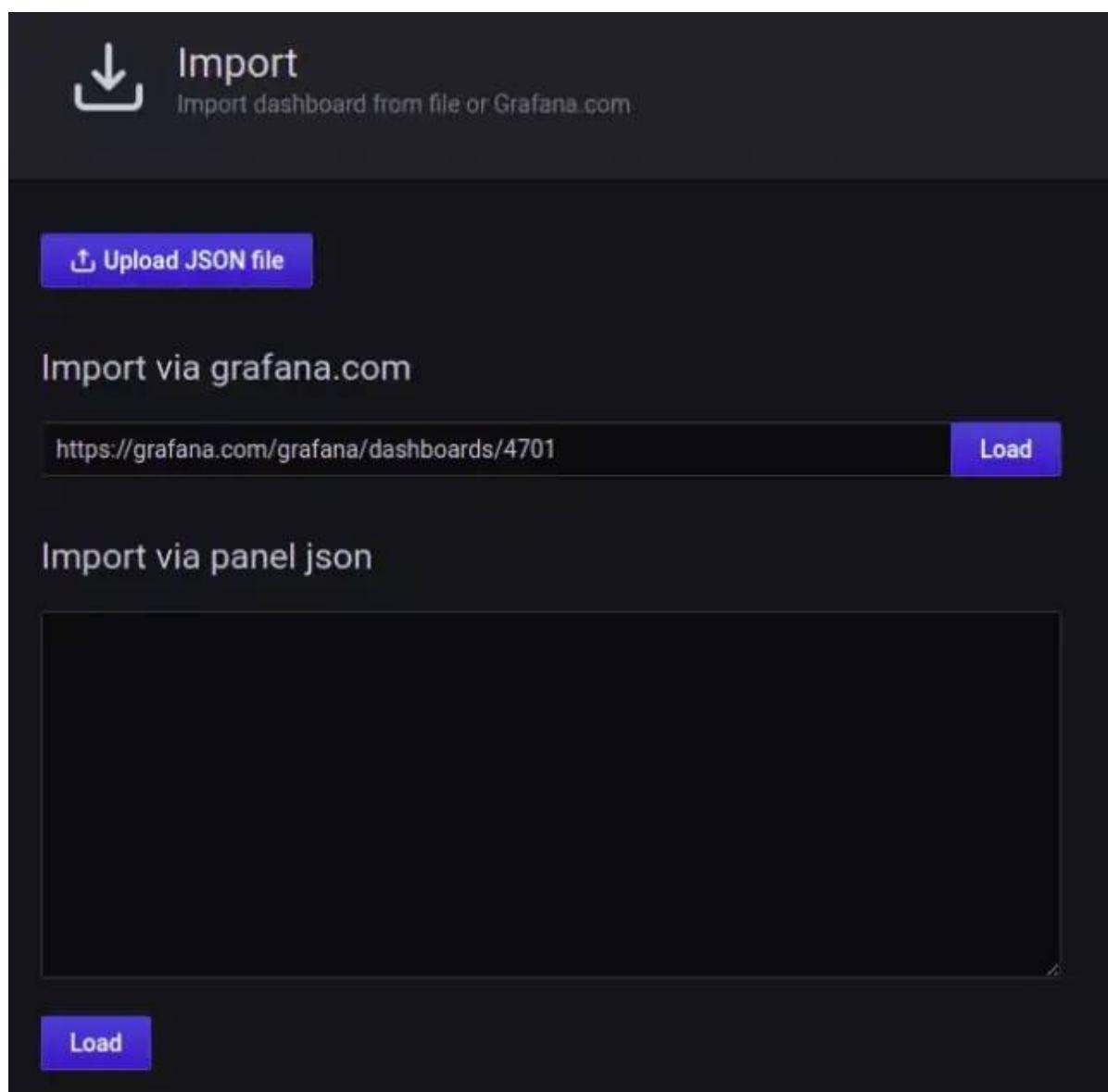


现在该创建仪表板了。你可以自定义一个，但也可以使用开源的仪表板。用于显示Spring Boot指标的一种常用仪表板是JVM仪表板。

在左侧边栏中，点击+号，然后选择导入。



输入JVM仪表板的URL <https://grafana.com/grafana/dashboards/4701>，然后单击“Load(加载)”按钮。



为仪表板输入一个有意义的名称（例如MySpringMonitoringPlanet），选择Prometheus作为数据源，然后单击Import按钮。



Import

Import dashboard from file or Grafana.com

Importing Dashboard from Grafana.com

Published by

mweirauch

Updated on

2019-11-03 18:00:25

Options

Name:

MySpringMonitoringPlanet

Folder:

General

Unique Identifier (uid)

The unique identifier (uid) of a dashboard can be used for uniquely identify a dashboard between multiple Grafana installs. The uid allows having consistent URLs for accessing dashboards so changing the title of a dashboard will not break any bookmarked links to that dashboard.

[Change uid](#)

Prometheus

Prometheus

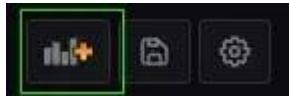
[Import](#)

[Cancel](#)

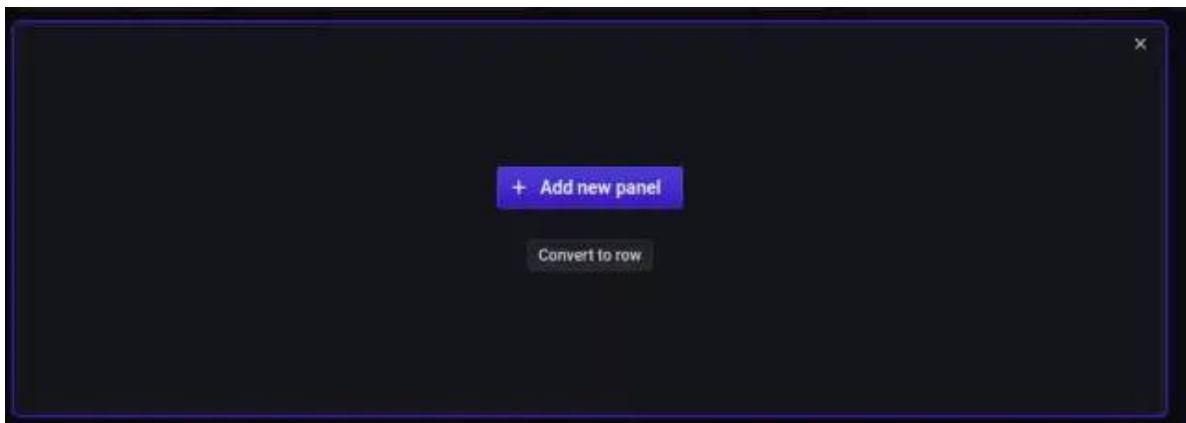
到目前为止，你就可以使用一个很酷的Grafana仪表板。



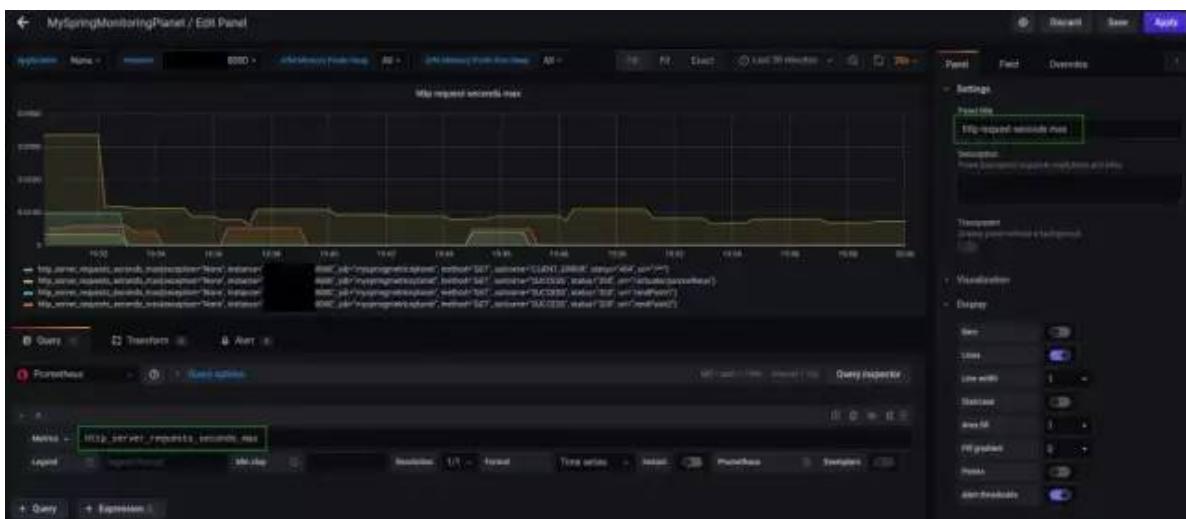
也可以将自定义面板添加到仪表板。在仪表板顶部，单击Add panel (添加面板) 图标。



单击Add new panel (添加新面板)。



在Metrics 字段中，输入http_server_requests_seconds_max，在右侧栏中的Panel title字段中，可以输入面板的名称。



最后，单击右上角的Apply 按钮，你的面板将添加到仪表板。不要忘记保存仪表板。

为应用程序设置一些负载，并查看仪表板上的http_server_requests_seconds_max指标发生了什么。

```
$ watch -n 5 curl http://localhost:8080/endPoint1$ watch -n 10 curl http://localhost:8080/endPoint2
```



5、结论

在本文中，我们学习了如何为Spring Boot应用程序添加一些基本监控。这非常容易，只需要通过将Spring Actuator, Micrometer, Prometheus和Grafana组合使用。当然，这只是一个起点，但是从这里开始，你可以为Spring Boot应用程序扩展和配置更多、更具体的指标。

Springboot 日志、配置文件、接口数据脱敏！

一、前言

核心隐私数据无论对于企业还是用户来说尤其重要，因此要想办法杜绝各种隐私数据的泄漏。下面陈某带大家从以下三个方面讲解一下隐私数据如何脱敏，也是日常开发中需要注意的：

1. 配置文件数据脱敏
2. 接口返回数据脱敏
3. 日志文件数据脱敏

文章目录如下：



二、配置文件如何脱敏？

经常会遇到这样一种情况：项目的配置文件中总有一些敏感信息，比如数据源的url、用户名、密码....这些信息一旦被暴露那么整个数据库都将会被泄漏，那么如何将这些配置隐藏呢？

以前都是手动将加密之后的配置写入到配置文件中，提取的时候再手动解密，当然这是一种思路，也能解决问题，但是每次都要手动加密、解密不觉得麻烦吗？

Watermark



今天介绍一种方案，让你在无感知的情况下实现配置文件的加密、解密。利用一款开源插件：[jasypt-spring-boot](https://github.com/ulisesbocchio/jasypt-spring-boot)。项目地址如下：

```
https://github.com/ulisesbocchio/jasypt-spring-boot
```

使用方法很简单，整合Spring Boot 只需要添加一个 `starter`。

1. 添加依赖

```
<dependency>
    <groupId>com.github.ulisesbocchio</groupId>
    <artifactId>jasypt-spring-boot-starter</artifactId>
    <version>3.0.3</version>
</dependency>
```

2. 配置秘钥

在配置文件中添加一个加密的秘钥（任意），如下：

```
jasypt:
  encryptor:
    password: Y6M9fAJQdU7jNp5MW
```

当然将秘钥直接放在配置文件中也是不安全的。我们可以在项目启动的时候配置秘钥，命令如下：

```
java -jar xxx.jar -Djasypt.encryptor.password=Y6M9fAJQdU7jNp5MW
```

3. 生成加密后的数据

这一步骤是将配置明文进行加密，代码如下：

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

    /**
     * 注入加密方法
     */
    @Autowired
    private StringEncryptor encryptor;

    /**
     * 手动生成密文，此处演示了url, user, password
     */
    @Test
    public void encrypt() {
        String url = encryptor.encrypt("jdbc:mysql://127.0.0.1:3306/test?
useUnicode=true&characterEncoding=UTF-
8&zeroDateTimeBehavior=convertToNull&useSSL=false&allowMultiQueries=true&serverTimezone=Asia/Shang
hai");
        String name = encryptor.encrypt("root");
        String password = encryptor.encrypt("123456");
        System.out.println("database url: " + url);
        System.out.println("database name: " + name);
        System.out.println("database password: " + password);
        Assert.assertTrue(url.length() > 0);
        Assert.assertTrue(name.length() > 0);
        Assert.assertTrue(password.length() > 0);
    }
}
```

上述代码对数据源的url、user、password进行了明文加密，输出的结果如下：

```
database url:
szkFDG56WcAOzG2utv0m2aoAvNFH5g3DXz0o6joZjT26Y5WNA+1Z+pQFpyhFBokq0p2jsFtB+P9b3gB601rfas3dSfvS8Bgo3MyP1noj
JgVp6gCVi+B/XUs0keXPn+pbX/19Hr1UN1LeEweHS/LCRZs1hWJCIXTwZo1P1pXRv3Vyhf20EzzKLm3mIAYj51CrEaN3w5cMiCES1lwv
KUhpAJVz/uXQJ1spLUAMuXCKKrXM/6dSRnWyTtdFRost5cChEU9uRjw5M+8HU3BLemtck0vM8iYDjEi5zDbZtwxD3hA=


database name: L8I2RqYPptEtQNL4x8hRVakSUdlsTGzEND/3TOnVTYPWe0ZnWsW0/5JdUsw9u1m


database password: EJYCSbBL8Pmf2HubIH7dHhpfdDZcLyJCEGMR9jAV3apJtvFtx9TVdhUPsAxjQ2pnJ
```

4. 将加密后的密文写入配置

jasypt 默认使用 ENC() 包裹，此时的数据源配置如下：

Watermark

```

spring:
  datasource:
    # 数据源基本配置
    username: ENC(L8I2RqYPptEtQNL4x8VhRVakSUD1sTGzEND/3TOnVTYPWe0ZnWsW0/5JdUsw9u1m)
    password: ENC(EJYCSbBL8Pmf2HubIH7dHhpfdZcLyJCEGMR9jAV3apJtvFtx9TVdhUPsAxjQ2pnJ)
    driver-class-name: com.mysql.jdbc.Driver
    url:
      ENC(szkFDG56WcAOzG2utv0m2aoAvNFH5g3DXz0o6joZjT26Y5WNA+1Z+pQFpyhFBokqOp2jsFtB+P9b3gB601rfas3dSfvS8Bgo3MyP
      1nojJgVp6gCVi+B/XUs0keXPn+pbX/19Hr1UN1LeEweHS/LCRzs1hWJCsIXTwZo1P1pXRv3Vyhf20EzzKLm3mIAYj51CrEaN3w5cMiCE
      S1wvKUhpAJVz/uXQJ1spLUAMuXCKKrXM/6dSRnWyTtdFRost5cChEU9uRjw5M+8HU3BLemtcK0vM8iYDjEi5zDbZtwxD3hA=)
    type: com.alibaba.druid.pool.DruidDataSource

```

上述配置是使用默认的 `prefix=ENC(`、`suffix=)`，当然我们可以根据自己的要求更改，只需要在配置文件中更改即可，如下：

```

jasypt:
  encryptor:
    ## 指定前缀、后缀
  property:
    prefix: 'PASS('
    suffix: ')'

```

那么此时的配置就必须使用 `PASS()` 包裹才会被解密，如下：

```

spring:
  datasource:
    # 数据源基本配置
    username: PASS(L8I2RqYPptEtQNL4x8VhRVakSUD1sTGzEND/3TOnVTYPWe0ZnWsW0/5JdUsw9u1m)
    password: PASS(EJYCSbBL8Pmf2HubIH7dHhpfdZcLyJCEGMR9jAV3apJtvFtx9TVdhUPsAxjQ2pnJ)
    driver-class-name: com.mysql.jdbc.Driver
    url:
      PASS(szkFDG56WcAOzG2utv0m2aoAvNFH5g3DXz0o6joZjT26Y5WNA+1Z+pQFpyhFBokqOp2jsFtB+P9b3gB601rfas3dSfvS8Bgo3My
      P1nojJgVp6gCVi+B/XUs0keXPn+pbX/19Hr1UN1LeEweHS/LCRzs1hWJCsIXTwZo1P1pXRv3Vyhf20EzzKLm3mIAYj51CrEaN3w5cMiC
      S1wvKUhpAJVz/uXQJ1spLUAMuXCKKrXM/6dSRnWyTtdFRost5cChEU9uRjw5M+8HU3BLemtcK0vM8iYDjEi5zDbZtwxD3hA=)
    type: com.alibaba.druid.pool.DruidDataSource

```

5. 总结

Jasypt还有许多高级用法，比如可以自己配置加密算法，具体的操作可以参考Github上的文档。

三、接口返回数据如何脱敏？

通常接口返回值中的一些敏感数据也是要脱敏的，比如身份证号、手机号码、地址.....通常的手段就是用 `*` 隐藏一部分数据，当然也可以根据自己需求定制。

言归正传，如何优雅的实现呢？有两种实现方案，如下：

- 整合 Mybatis 拦截，在查询的时候针对特定的字段进行脱敏
- 整合 Jackson，在序列化阶段对指定字段进行脱敏
- 基于 Sharding Sphere 实现数据脱敏，查看之前的文章：[基于Sharding Sphere实现数据“一键脱敏”](#)

第一种方案网上很多实现方式，下面演示第二种，整合Jackson。

1. 自定义一个Jackson注解

需要自定义一个脱敏注解，一旦有属性被标注，则进行对应得脱敏，如下：

```
/**  
 * 自定义jackson注解，标注在属性上  
 */  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.FIELD)  
@JacksonAnnotationsInside  
@JsonSerialize(using = SensitiveJsonSerializer.class)  
public @interface Sensitive {  
    //脱敏策略  
    SensitiveStrategy strategy();  
}
```

2. 定制脱敏策略

针对项目需求，定制不同字段的脱敏规则，比如手机号中间几位用 * 替代，如下：

```
/**  
 * 脱敏策略，枚举类，针对不同的数据定制特定的策略  
 */  
public enum SensitiveStrategy {  
    /**  
     * 用户名  
     */  
    USERNAME(s -> s.replaceAll("(\\$)\\$((\\$*))", "$1*$2")),  
    /**  
     * 身份证  
     */  
    ID_CARD(s -> s.replaceAll("(\\d{4})\\d{10}(\\w{4})", "$1****$2")),  
    /**  
     * 手机号  
     */  
    PHONE(s -> s.replaceAll("(\\d{3})\\d{4}(\\d{4})", "$1****$2")),  
    /**  
     * 地址  
     */  
    ADDRESS(s -> s.replaceAll("(\\S{3})\\S{2}((\\$*))\\S{2}", "$1****$2****"));  
  
    private final Function<String, String> desensitizer;  
  
    SensitiveStrategy(Function<String, String> desensitizer) {  
        this.desensitizer = desensitizer;  
    }  
  
    public Function<String, String> desensitizer() {  
        return desensitizer;  
    }  
}
```

以上只是提供了部分，具体根据自己项目要求进行配置。

3. 定制JSON序列化实现

下面将是重要实现，对标注注解 `@Sensitive` 的字段进行脱敏，实现如下：

```
/*
 * 序列化注解自定义实现
 * JsonSerializer<String>：指定String 类型， serialize()方法用于将修改后的数据载入
 */
public class SensitiveJsonSerializer extends JsonSerializer<String> implements ContextualSerializer {
    private SensitiveStrategy strategy;

    @Override
    public void serialize(String value, JsonGenerator gen, SerializerProvider serializers) throws
IOException {
        gen.writeString(strategy.desensitizer().apply(value));
    }

    /**
     * 获取属性上的注解属性
     */
    @Override
    public JsonSerializer<?> createContextual(SerializerProvider prov, BeanProperty property) throws
JsonMappingException {
        Sensitive annotation = property.getAnnotation(Sensitive.class);
        if (Objects.nonNull(annotation)&&Objects.equals(String.class,
property.getType().getRawClass())) {
            this.strategy = annotation.strategy();
            return this;
        }
        return prov.findValueSerializer(property.getType(), property);
    }
}
```

4. 定义Person类，对其数据脱敏

使用注解 `@Sensitive` 注解进行数据脱敏，代码如下：

```
@Data
public class Person {
    /**
     * 真实姓名
     */
    @Sensitive(strategy = SensitiveStrategy.USERNAME)
    private String realName;
    /**
     * 地址
     */
    @Sensitive(strategy = SensitiveStrategy.ADDRESS)
    private String address;
    /**
     * 电话号码
     */
    @Sensitive(strategy = SensitiveStrategy.PHONE)
    private String phoneNumber;
    /**
```

```
* 身份证号码  
*/  
@Sensitive(strategy = SensitiveStrategy.ID_CARD)  
private String idCard;  
}
```

5. 模拟接口测试

以上4个步骤完成了数据脱敏的Jackson注解，下面写个controller进行测试，代码如下：

```
@RestController  
public class TestController {  
    @GetMapping("/test")  
    public Person test() {  
        Person user = new Person();  
        user.setRealName("不才陈某");  
        user.setPhoneNumber("19796328206");  
        user.setAddress("浙江省杭州市温州市....");  
        user.setIdCard("43333333334334333");  
        return user;  
    }  
}
```

调用接口查看数据有没有正常脱敏，结果如下：

```
{  
    "realName": "不*陈某",  
    "address": "浙江省****市温州市..****",  
    "phoneNumber": "197****8206",  
    "idCard": "4333****34333"  
}
```

6. 总结

数据脱敏有很多种实现方式，关键是哪种更加适合，哪种更加优雅.....

四、日志文件如何数据脱敏？

上面讲了配置文件、接口返回值的数据脱敏，现在总该轮到日志脱敏了。项目中总避免不了打印日志，肯定会涉及到一些敏感数据被明文打印出来，那么此时就需要过滤掉这些敏感数据（身份证、号码、用户名.....）。

关于Spring Boot 日志方面的问题有不理解的可以看我之前的文章：[Spring Boot第三弹，一文带你搞懂日志如何配置？](#)、[Spring Boot第二弹，配置文件怎么造？](#)。

下面以Log4j2这款日志为例讲解一下日志如何脱敏，其他日志框架大致思路一样。

Watermark

1. 添加log4j2日志依赖

Spring Boot 默认日志框架是logback，但是我们可以切换到log4j2，依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <!-- 去掉springboot默认配置 -->
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<!--使用log4j2替换 LogBack-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

2. 在/resource目录下新建log4j2.xml配置

log4j2的日志配置很简单，只需要在 `/resource` 文件夹下新建一个 `log4j2.xml` 配置文件，内容如下图：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="DEBUG">
    <!-- 定义日志存放目录 -->
    <properties>
        <property name="logPath">logs</property>
        <property name="PATTERN">%d{yyyy-MM-dd HH:mm:ss.SSS} [%X{traceId}] [%t-%L] %-5level %logger{36}|%L %M - %msg%xEx%n</property>
        <!--"%d{yyyy-MM-dd HH:mm:ss} [%thread] %m%n"-->
    </properties>
    <!--先定义所有的appender(输出器) -->
    <Appenders>
        <!--输出到控制台 -->
        <Console name="ConsoleLog" target="SYSTEM_OUT">
            <!--只输出level及以上级别的信息 (onMatch) , 其他的直接拒绝 (onMismatch) -->
            <ThresholdFilter level="TRACE" onMatch="ACCEPT" onMismatch="DENY" />
            <!--输出日志的格式, 引用自定义模板 PATTERN -->
            <PatternLayout pattern="${PATTERN}" />
        </Console>
        <RollingFile name="APPLog" fileName="${logPath}/log_app.log" filePattern="${logPath}/log_app_%d{yyyy-MM-dd}.log">
            <ThresholdFilter level="DEBUG" onMatch="ACCEPT" onMismatch="DENY" />
            <!--<PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss} [%thread] %m%n"/>-->
            <PatternLayout pattern="${PATTERN}" />
            <Policies>
                <TimeBasedTriggeringPolicy modulate="true" interval="1"/>
            </Policies>
            <!--CronTriggeringPolicy schedule="0 0 5 * * ? "/>
            <DefaultRolloverStrategy>
                <Delete basePath="${logPath}" maxDepth="1">
                    <IfFileName glob="log_app_*.log" />
                    <!--删除15天前的文件-->
                    <IfLastModified age="15d" />
                </Delete>
            </DefaultRolloverStrategy>
        </RollingFile>
    </Appenders>
</Configuration>
```

Watermark

```

</RollingFile>
<!-- 把error等级记录到文件 一般不用 -->
<File name="ERRORLog" fileName="${logPath}/error.log">
    <ThresholdFilter level="error" onMatch="ACCEPT" onMismatch="DENY" />
    <PatternLayout pattern="${PATTERN}" />
</File>
</Appenders>
<!-- 然后定义Logger, 只有定义了logger并引入的appender, appender才会生效 -->
<Loggers>
    <!-- 建立一个默认的Root的logger, 记录大于Level高于warn的信息, 如果这里的level高于Appenders中的, 则Appenders中也是以此等级为起点, 比如, 这里level="fatal", 则Appenders中只出现fatal信息 -->
    <!-- 生产环境level=>warn -->
    <Root level="debug">
        <!-- 输出器, 可选上面定义的任何项组合, 或全选, 做到可随意定制 -->
        <appender-ref ref="ConsoleLog" />
        <appender-ref ref="ERRORLog" />
        <appender-ref ref="APPLog" />
    </Root>
    <!-- 第三方日志系统 -->
    <!--过滤掉spring和mybatis的一些无用的DEBUG信息, 也可以在spring boot 的logging.level.org.springframework=FATAL设置-->
    <logger name="org.springframework" level="DEBUG"></logger>
    <logger name="java.sql.Connection" level="DEBUG"></logger>
    <logger name="java.sql.Statement" level="DEBUG"></logger>
    <logger name="Java.sql.PreparedStatement" level="DEBUG"></logger>
</Loggers>
</Configuration>

```

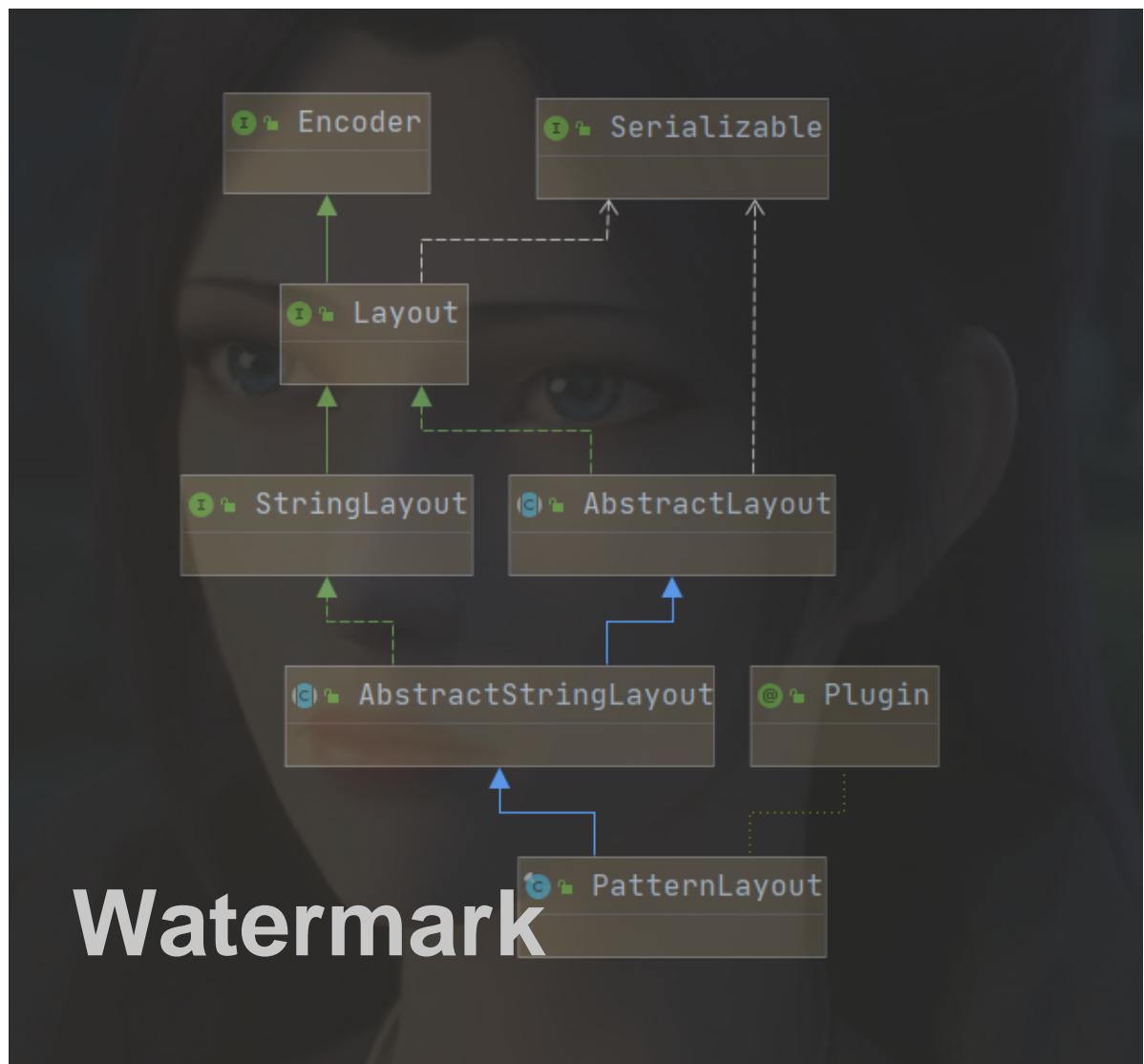
关于每个节点如何配置，含义是什么，在我上面的两篇文章中有详细的介绍。

上图的配置并没有实现数据脱敏，这是普通的配置，使用的是 **PatternLayout**

3. 自定义PatternLayout实现数据脱敏

步骤2中的配置使用的是 **PatternLayout** 实现日志的格式，那么我们也可以自定义一个**PatternLayout**来实现日志的过滤脱敏。

PatternLayout的类图继承关系如下：



从上图中可以清楚的看出来，`PatternLayout`继承了一个抽象类 `AbstractStringLayout`，因此想要自定义只需要继承这个抽象类即可。

1、创建`CustomPatternLayout`, 继承抽象类`AbstractStringLayout`

代码如下：

```
/*
 * log4j2 脱敏插件
 * 继承AbstractStringLayout
 */
@Plugin(name = "CustomPatternLayout", category = Node.CATEGORY, elementType = Layout.ELEMENT_TYPE,
printObject = true)
public class CustomPatternLayout extends AbstractStringLayout {

    public final static Logger logger = LoggerFactory.getLogger(CustomPatternLayout.class);
    private PatternLayout patternLayout;

    protected CustomPatternLayout(Charset charset, String pattern) {
        super(charset);
        patternLayout = PatternLayout.newBuilder().withPattern(pattern).build();
        initRule();
    }

    /**
     * 要匹配的正则表达式map
     */
    private static Map<String, Pattern> REG_PATTERN_MAP = new HashMap<>();
    private static Map<String, String> KEY_REG_MAP = new HashMap<>();

    private void initRule() {
        try {
            if (MapUtils.isEmpty(Log4j2Rule.regularMap)) {
                return;
            }
            Log4j2Rule.regularMap.forEach((a, b) -> {
                if (StringUtils.isNotBlank(a)) {
                    Map<String, String> collect =
                        Arrays.stream(a.split(",")).collect(Collectors.toMap(c -> c, w -> b, (key1, key2) -> key1));
                    KEY_REG_MAP.putAll(collect);
                }
                Pattern compile = Pattern.compile(b);
                REG_PATTERN_MAP.put(b, compile);
            });
        } catch (Exception e) {
            logger.info(">>>> 初始化日志脱敏规则失败 ERROR: {}", e);
        }
    }
}
```

Watermark

```
/*
 * 处理日志信息，进行脱敏
 * 1. 判断配置文件中是否已经配置需要脱敏字段
 * 2. 判断内容是否有需要脱敏的敏感信息
*/
```

```

    * 2.1 没有需要脱敏信息直接返回
    * 2.2 处理：身份证号，姓名，手机号敏感信息
    */
public String hideMarkLog(String logStr) {
    try {
        //1. 判断配置文件中是否已经配置需要脱敏字段
        if (StringUtils.isBlank(logStr) || MapUtils.isEmpty(KEY_REG_MAP) ||
MapUtils.isEmpty(REG_PATTERN_MAP)) {
            return logStr;
        }
        //2. 判断内容是否有需要脱敏的敏感信息
        Set<String> charKeys = KEY_REG_MAP.keySet();
        for (String key : charKeys) {
            if (logStr.contains(key)) {
                String regExp = KEY_REG_MAP.get(key);
                logStr = matchingAndEncrypt(logStr, regExp, key);
            }
        }
        return logStr;
    } catch (Exception e) {
        logger.info(">>>>>> 脱敏处理异常 ERROR: {}", e);
        //如果抛出异常为了不影响流程，直接返回原信息
        return logStr;
    }
}

/**
 * 正则匹配对应的对象。
 *
 * @param msg
 * @param regExp
 * @return
 */
private static String matchingAndEncrypt(String msg, String regExp, String key) {
    Pattern pattern = REG_PATTERN_MAP.get(regExp);
    if (pattern == null) {
        logger.info(">>> logger 没有匹配到对应的正则表达式 ");
        return msg;
    }
    Matcher matcher = pattern.matcher(msg);
    int length = key.length() + 5;
    boolean contains = Log4j2Rule.USER_NAME_STR.contains(key);
    String hiddenStr = "";
    while (matcher.find()) {
        String originStr = matcher.group();
        if (contains) {
            // 计算关键词和需要脱敏词的距离小于5。
            int i = msg.indexOf(originStr);
            if (i < 0) {
                continue;
            }
            int end = i + length;
            int start = i - span > 0 ? span : 0;
            String substring = msg.substring(startIndex, i);
            if (StringUtils.isBlank(substring) || !substring.contains(key)) {
                continue;
            }
            hiddenStr = hideMarkStr(originStr);
        }
    }
}

```

```

        msg = msg.replace(originStr, hiddenStr);
    } else {
        hiddenStr = hideMarkStr(originStr);
        msg = msg.replace(originStr, hiddenStr);
    }

}

return msg;
}

/**
 * 标记敏感文字规则
 *
 * @param needHideMark
 * @return
 */
private static String hideMarkStr(String needHideMark) {
    if (StringUtils.isBlank(needHideMark)) {
        return "";
    }
    int startSize = 0, endSize = 0, mark = 0, length = needHideMark.length();

    StringBuffer hideRegBuffer = new StringBuffer("(\\S+");
    StringBuffer replaceSb = new StringBuffer("$1");

    if (length > 4) {
        int i = length / 3;
        startSize = i;
        endSize = i;
    } else {
        startSize = 1;
        endSize = 0;
    }

    mark = length - startSize - endSize;
    for (int i = 0; i < mark; i++) {
        replaceSb.append("*");
    }
    hideRegBuffer.append(startSize).append(")\\$*(\\S+").append(endSize).append(")");
    replaceSb.append("$2");
    needHideMark = needHideMark.replaceAll(hideRegBuffer.toString(), replaceSb.toString());
    return needHideMark;
}

/**
 * 创建插件
 */
@PluginFactory
public static Layout createLayout(@PluginAttribute(value = "pattern") final String pattern,
                                  @PluginAttribute(value = "charset") final
Character charset) {
    return new CustomPatternLayout(charset, pattern);
}

@Override
public String toSerializable(LogEvent event) {
}

```

```
        return hideMarkLog(patternLayout.toSerializable(event));
    }

}
```

关于其中的一些细节，比如 `@Plugin`、`@PluginFactory` 这两个注解什么意思？`log4j2`如何实现自定义一个插件，这里不再详细介绍，不是本文重点，有兴趣的可以查看 `log4j2` 的官方文档。

2、自定义自己的脱敏规则

上述代码中的 `Log4j2Rule` 则是脱敏规则静态类，我这里是直接放在了静态类中配置，实际项目中可以设置到配置文件中，代码如下：

```
/**
 * 现在拦截加密的日志有三类：
 * 1, 身份证
 * 2, 姓名
 * 3, 身份证号
 * 加密的规则后续可以优化在配置文件中
 **/


public class Log4j2Rule {

    /**
     * 正则匹配 关键词 类别
     */
    public static Map<String, String> regularMap = new HashMap<>();

    /**
     * TODO 可配置
     * 此项可以后期放在配置项中
     */
    public static final String USER_NAME_STR = "Name, name, 联系人, 姓名";
    public static final String USER_IDCARD_STR = "empCard, idCard, 身份证, 证件号";
    public static final String USER_PHONE_STR = "mobile, Phone, phone, 电话, 手机";

    /**
     * 正则匹配，自己根据业务要求自定义
     */
    private static String IDCARD_REGEX = "(\\d{17} [0-9Xx] | \\d{14} [0-9Xx])";
    private static String USERNAME_REGEX = "[\\u4e00-\\u9fa5]{2,4}";
    private static String PHONE_REGEX = "(?:\\d{11}|\\d{14})|(?:1[3456789]\\d{9})|(?:861[356789]\\d{9})";
    (?!\d);

    static {
        regularMap.put(USER_NAME_STR, USERNAME_REGEX);
        regularMap.put(USER_IDCARD_STR, IDCARD_REGEX);
        regularMap.put(USER_PHONE_STR, PHONE_REGEX);
    }

}
```

经过上述两个步骤，自定义的 `PatternLayout` 已经完成，下面将是改写 `log4j2.xml` 这个配置文件了。

Watermark

4. 修改log4j2.xml配置文件

其实这里修改很简单，原配置文件是直接使用 `PatternLayout` 进行日志格式化的，那么只需要将默认的 `<PatternLayout/>` 这个节点替换成 `<CustomPatternLayout/>`，如下图：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="DEBUG">
    <!-- 定义日志存放目录 -->
    <properties>
        <property name="logPath">logs</property>
        <property name="PATTERN">%d{yyyy-MM-dd HH:mm:ss.SSS} [%X{traceId}] [%t-%L] %-5level %logger{36} %L %M - %msg%xEx%n</property>
        <!-- "%d{yyyy-MM-dd HH:mm:ss} [%thread] %m%n"-->
    </properties>
    <!--先定义所有的appender(输出器) -->
    <Appenders>
        <!--输出到控制台 -->
        <Console name="ConsoleLog" target="SYSTEM_OUT">
            <!--只输出level及以上级别的信息 (onMatch) , 其他的直接拒绝 (onMismatch) -->
            <ThresholdFilter level="TRACE" onMatch="ACCEPT" onMismatch="DENY" />
            <!--输出日志的格式, 引用自定义模板 PATTERN -->
            <CustomPatternLayout pattern="${PATTERN}" />
        </Console>
        <RollingFile name="APPLog" fileName="${logPath}/log_app.log" filePattern="${logPath}/log_app_%d{yyyy-MM-dd}.log">
            <ThresholdFilter level="DEBUG" onMatch="ACCEPT" onMismatch="DENY" />
            <!--<CustomPatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss} [%thread] %m%n"/-->
            <CustomPatternLayout pattern="${PATTERN}" />
        </RollingFile>
        <Policies>
            <TimeBasedTriggeringPolicy modulate="true" interval="1"/>
        </Policies>
        <CronTriggeringPolicy schedule="0 0 5 * * ? "/>
        <DefaultRolloverStrategy>
            <Delete basePath="${logPath}" maxDepth="1">
                <IfFileName glob="Log_app.*.log" />
                <!--删除15天前的文件-->
                <IfLastModified age="15d" />
            </Delete>
        </DefaultRolloverStrategy>
    </Appenders>
</Configuration>
```

直接全局替换掉即可，至此，这个配置文件就修改完成了。

5. 演示效果

在步骤3这边自定义了脱敏规则静态类 `Log4j2Rule`，其中定义了姓名、身份证、号码这三个脱敏规则，如下：

```
public class Log4j2Rule {
    /**
     * 正则匹配 关键词 类别
     */
    public static Map<String, String> regularMap = new HashMap<>();
    /**
     * TODO 可配置
     * 此项可以后期放在配置项中
     */
    public static final String USER_NAME_STR = "Name,name,联系人,姓名";
    public static final String USER_IDCARD_STR = "empCard,idCard,身份证,证件号";
    public static final String USER_PHONE_STR = "mobile,Phone,phone,电话,手机";

    /**
     * 正则匹配, 自己根据业务要求自定义
     */
    private static String IDCARD_REGEXP = "(\\d{17}[0-9Xx]|\\d{14}[0-9Xx])";
    private static String USERNAME_REGEXP = "[\\u4e00-\\u9fa5]{2,4}";
    private static String PHONE_REGEXP = "(?<!\\d)(?:(:1[3456789]\\d{9})|(:861[356789]\\d{9}))(!\\d)";

    static {
        regularMap.put(USER_NAME_STR, USERNAME_REGEXP);
        regularMap.put(USER_IDCARD_STR, IDCARD_REGEXP);
        regularMap.put(USER_PHONE_STR, PHONE_REGEXP);
    }
}
```

下面就来演示这三个规则能否正确脱敏，直接使用日志打印，代码如下：

```
@Test  
public void test3() {  
    log.debug("身份证: {}, 姓名: {}, 电话: {}","320829112334566767", "不才陈某", "19896327106");  
}
```

控制台打印的日志如下：

```
身份证: 320829*****566767, 姓名: 不***, 电话: 198*****106
```

哦豁，成功了，so easy! ! !

6. 总结

日志脱敏的方案很多，陈某也只是介绍一种常用的，有兴趣的可以研究一下。

五、总结

本篇文章从三个维度介绍了隐私数据的脱敏实现方案，码字不易，赶紧点赞收藏吧！！！

源码已经上传GitHub，需要的公众号码猿技术专栏，回复关键词 **数据脱敏** 获取。

40 个 SpringBoot 常用注解：让生产力爆表！

一、Spring Web MVC 与 Spring Bean 注解

@RequestMapping

@RequestMapping注解的主要用途是将Web请求与请求处理类中的方法进行映射。Spring MVC和Spring WebFlux都通过 `RequestMappingHandlerMapping` 和 `RequestMappingHandlerAdapter` 两个类来提供对@RequestMapping注解的支持。

`@RequestMapping` 注解对请求处理类中的请求处理方法进行标注；`@RequestMapping` 注解拥有以下的六个配置属性：

- `value` :映射的请求URL或者其别名
- `method` :兼容HTTP的方法名
- `params` :根据HTTP参数的存在、缺省或值对请求进行过滤
- `headers` :根据HTTP Headers 的存在、缺省或值对请求进行过滤
- `consumes` :指定在HTTP请求正文不允许使用的媒体类型
- `produces` :在HTTP响应体中允许使用的媒体类型

提示：在使用@RequestMapping之前，请求处理类还需要使用@Controller或@RestController进行标记

下面是使用@RequestMapping的两个示例:

```
1 @Controller
2 public class DemoController{
3
4     @RequestMapping(value="/demo/home",method=RequestMethod.GET)
5     public String home(){
6         return "/home";
7     }
8 }
```

@RequestMapping还可以对类进行标记，这样类中的处理方法在映射请求路径时，会自动将类上@RequestMapping设置的value拼接到方法中映射路径之前，如下：

```
1 @Controller
2 @RequestMapping(value="/demo")
3 public class DemoController{
4
5     @RequestMapping(value="/home",method=RequestMethod.GET)
6     public String home(){
7         return "/home";
8     }
9 }
```

@RequestBody

@RequestBody在处理请求方法的参数列表中使用，它可以将请求主体中的参数绑定到一个对象中，请求主体参数是通过 `HttpMessageConverter` 传递的，根据请求主体中的参数名与对象的属性名进行匹配并绑定值。此外，还可以通过@Valid注解对请求主体中的参数进行校验。

下面是一个使用 `@RequestBody` 的示例：

Watermark

```
1 @RequestController
2 @RequestMapping("/api/v1")
3 public class UserController{
4     @Autowired
5     private UserService userService;
6
7     @PostMapping("/users")
8     public User createUser(@Valid @RequestBody User user){
9         return userService.save(user);
10    }
11 }
```

@GetMapping

`@GetMapping` 注解用于处理HTTP GET请求，并将请求映射到具体的处理方法中。具体来说，`@GetMapping`是一个组合注解，它相当于是 `@RequestMapping(method=RequestMethod.GET)` 的快捷方式。

下面是 `@GetMapping` 的一个使用示例：

```
1 @RequestController
2 @RequestMapping("/api/v1")
3 public class UserController{
4     @Autowired
5     private UserService userService;
6
7     @GetMapping("/users")
8     public List<User> findAllUser(){
9         List<User> users = userService.findAll();
10        return users;
11    }
12
13    @GetMapping("/users/{id}")
14    public User findOneById(@PathVariable(name="id") long id) throws
15        UserNotFoundException{
16        return userService.findOne(id);
17    }
18 }
```

@PostMapping

`@PostMapping` 注解用于处理HTTP POST请求，并将请求映射到具体的处理方法中。`@PostMapping`与`@GetMapping`一样，也是一个组合注解，它相当于是 `@RequestMapping(method=RequestMethod.POST)` 的快捷方式。

下面是使用 `@PostMapping` 的一个示例：

```
1 @RequestController
2 @RequestMapping("/api/v1")
3 public class UserController{
4     @Autowired
5     private UserService userService;
6
7     @PostMapping("/users")
8     public User createUser(@Valid @RequestBody User user){
9         return userService.save(user);
10    }
11 }
```

@PutMapping

`@PutMapping` 注解用于处理HTTP PUT请求，并将请求映射到具体的处理方法中，`@PutMapping`是一个组合注解，相当于是 `@RequestMapping(method=HttpMethod.PUT)` 的快捷方式。

下面是使用 `@PutMapping` 的一个示例：

```
1 @RequestController
2 @RequestMapping("/api/v1")
3 public class UserController{
4     @Autowired
5     private UserRepository userRepository;
6
7     @PutMapping("/users/{id}")
8     public ResponseEntity<User> updateUser(@PathVariable(name="id")long id,@Value
9     @ResponseBody User detail) throws UserNotFoundException{
10
11     User user = userRepository.findById(id)
12     .orElseThrow(()→new UserNotFoundException("User not found with this id:"+id));
13
14     user.setLastName(detail.getLastName());
15     user.setEmail(detail.getEmail());
16     user.setAddress(detail.getAddress());
17     final User origin = userRepository.save(user);
18
19 }
```

@DeleteMapping

`@DeleteMapping` 注解用于处理HTTP DELETE请求，并将请求映射到删除方法中。`@DeleteMapping`是一个组合注解，它相当于是 `@RequestMapping(method=HttpMethod.DELETE)` 的快捷方式。

下面是使用 `@DeleteMapping` 的一个示例：

```
1 @RequestController
2 @RequestMapping("/api/v1")
3 public class UserController{
4     @Autowired
5     private UserRepository userRepository;
6
7     @DeleteMapping("/users/{id}")
8     public Map<String, Boolean> deleteById(@PathVariable(name="id")long id) throws
UserNotFoundException{
9         User user = userRepository.findById(id)
10        .orElseThrow(()→new UserNotFoundException("User not found for this id :" + id));
userRepository.delete(user);
11        Map<String, Boolean> response = new HashMap<>();
12        response.put("deleted", Boolean.TRUE);
13        return response;
14    }
15 }
```

@PatchMapping

`@PatchMapping` 注解用于处理HTTP PATCH请求，并将请求映射到对应的处理方法中。

`@PatchMapping` 相当于 `@RequestMapping(method=HttpMethod.PATCH)` 的快捷方式。

下面是一个简单的示例：

```
1 @RequestController
2 @RequestMapping("/api/v1")
3 public class UserController{
4     @PatchMapping("/users/patch")
5     public ResponseEntity<Object> patch(){
6         return new ResponseEntity<>("Patch method response message", HttpStatus.OK);
7     }
8 }
```

@ControllerAdvice

`@ControllerAdvice` 是`@Component`注解的一个延伸注解，Spring会自动扫描并检测被`@ControllerAdvice`所标注的类。`@ControllerAdvice` 需要和`@ExceptionHandler`、`@InitBinder`以及`@ModelAttribute`注解搭配使用，主要是用来处理控制器所抛出的异常信息。

首先，我们需要定义一个被`@ControllerAdvice` 所标注的类，在该类中，定义一个用于处理具体异常的方法，并使用`@ExceptionHandler`注解进行标记。

此外，在必要的时候，可以使用`@InitBinder` 在类中进行全局的配置，还可以使用`@ModelAttribute`配置与视图相关的参数。使用`@ControllerAdvice`注解，就可以快速的创建统一的，自定义的异常处理类。

下面是一个使用`@ControllerAdvice` 的示例代码：

```
1 @ControllerAdvice(basePackages={"com.ramostear.controller.user"})
2 public class UserControllerAdvice{
3
4     @InitBinder
5     public void binder(WebDataBinder binder){
6         SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
7         format.setLenient(false);
8         binder.registerCustomEditor(Date.class,"user",new CustomDateFormat(format,true));
9     }
10
11     @ModelAttribute
12     public void modelAttribute(Model model){
13         model.addAttribute("msg","User not found exception.");
14     }
15
16
17     @ExceptionHandler(UserNotFoundException.class)
18     public ModelAndView userNotFoundExceptionHandler(UserNotFoundException ex){
19
20         ModelAndView modelAndView = new ModelAndView();
21         modelAndView.addObject("exception",ex);
22         modelAndView.setViewName("error");
23         return modelAndView;
24     }
25 }
```

@ResponseBody

`@ResponseBody` 会自动将控制器中方法的返回值写入到HTTP响应中。特别的，`@ResponseBody` 注解只能用在被`@Controller`注解标记的类中。如果在被`@RestController`标记的类中，则方法不需要使用`@ResponseBody`注解进行标注。`@RestController`相当于是`@Controller`和`@ResponseBody`的组合注解。

下面是使用该注解的一个示例

```
1 @ResponseBody
2 @GetMapping("/users/{id}")
3 public User findById(@PathVariable long id) throws UserNotFoundException{
4     User user = userService.findOne(id);
5     return user;
6 }
```

@ExceptionHandler

`@ExceptionHandler`注解用于标注处理特定类型异常类所抛出异常的方法。当控制器中的方法抛出异常时，Spring会自动捕获异常，并将捕获的异常信息传递给被`@ExceptionHandler`标注的方法。

下面是使用该注解的一个示例。

```
1 @ExceptionHandler(UserNotFoundException.class)
2 public ResponseEntity<Object> userNotFoundExceptionHandler(UserNotFoundException
   ex,WebRequest request){
3
4   UserErrorDetail detail = new UserErrorDetail(new
   Date(),ex.getMessage(),request.getDescription(false));
5   return new ResponseEntity<>(detail,HttpStatus.NOT_FOUND);
6
7 }
```

@ResponseStatus

`@ResponseStatus` 注解可以标注请求处理方法。使用此注解，可以指定响应所需要的HTTP STATUS。特别地，我们可以使用`HttpStauts`类对该注解的`value`属性进行赋值。

下面是使用 `@ResponseStatus` 注解的一个示例：

```
1 @ResponseStatus(HttpStatus.BAD_REQUEST)
2 @ExceptionHandler(UserNotFoundException.class)
3 public ResponseEntity<Object> userNotFoundExceptionHandler(UserNotFoundException
   ex,WebRequest request){
4
5   UserErrorDetail detail = new UserErrorDetail(new
   Date(),ex.getMessage(),request.getDescription(false));
6   return new ResponseEntity<>(detail,HttpStatus.NOT_FOUND);
7
8 }
```

@PathVariable

`@PathVariable` 注解是将方法中的参数绑定到请求URI中的模板变量上。可以通过 `@RequestMapping` 注解来指定URI的模板变量，然后使用 `@PathVariable` 注解将方法中的参数绑定到模板变量上。

特别地，`@PathVariable` 注解允许我们使用`value`或`name`属性来给参数取一个别名。下面是使用此注解的一个示例：

```
1 @GetMapping("/users/{id}/roles/{roleId}")
2 public Role getUserRole(@PathVariable(name="id")long
   id,@PathVariable(value="roleId")long roleId) throws ResourceNotFoundException{
3
4   return userRoleService.findByIdAndRoleId(id,roleId);
5
6 }
```

模板变量名需要使用`{ }`进行包裹，如果方法的参数名与URI模板变量名一致，则在 `@PathVariable` 中就可以省去别名的步骤。

下面是一个简写的示例：

```
1 @GetMapping("/users/{id}/roles/{roleId}")
2 public Role getUserRole(@PathVariable long id,@PathVariable long roleId) throws
3     ResourceNotFoundException{
4
5     return userRoleService.findByIdAndRoleId(id,roleId);
6 }
```

提示：如果参数是一个非必须的，可选的项，则可以在 `@PathVariable` 中设置 `require = false`

@RequestParam

`@RequestParam` 注解用于将方法的参数与Web请求的传递的参数进行绑定。使用 `@RequestParam` 可以轻松的访问HTTP请求参数的值。

下面是使用该注解的代码示例：

```
1 @GetMapping
2 public Role getUserRole(@RequestParam(name="id") long id,@RequestParam(name="roleId")
3     long roleId) throws ResourceNotFoundException{
4
5     return userRoleService.findByIdAndRoleId(id,roleId);
6 }
```

该注解的其他属性配置与 `@PathVariable` 的配置相同，特别的，如果传递的参数为空，还可以通过 `defaultValue` 设置一个默认值。示例代码如下：

```
1 @GetMapping
2 public Role getUserRole(@RequestParam(name="id",defaultValue="0") long
3     id,@RequestParam(name="roleId",defaultValue="0") long roleId) throws
4     ResourceNotFoundException{
5     if(id==0 || roleId==0){
6         return new Role();
7     }
8     return userRoleService.findByIdAndRoleId(id,roleId);
9 }
```

@Controller

`@Controller` 是 `@Component` 注解的一个延伸，Spring会自动扫描并配置被该注解标注的类。此注解用于标注Spring MVC的控制器。下面是使用此注解的示例代码：

Watermark

```
1 @Controller
2 @RequestMapping("/api/v1")
3 public class UserApiController{
4
5     @Autowired
6     private UserService userService;
7
8     @GetMapping("/users/{id}")
9     @ResponseBody
10    public User getUserById(@PathVariable long id) throws UserNotFoundException{
11        return userService.findOne(id);
12    }
13
14 }
```

@RestController

`@RestController` 是在Spring 4.0开始引入的，这是一个特定的控制器注解。此注解相当于 `@Controller` 和 `@ResponseBody` 的快捷方式。当使用此注解时，不需要再在方法上使用 `@ResponseBody` 注解。

下面是使用此注解的示例代码：

```
1 @RestController
2 @RequestMapping("/api/v1")
3 public class UserApiController{
4
5     @Autowired
6     private UserService userService;
7
8     @GetMapping("/users/{id}")
9     public User getUserById(@PathVariable long id) throws UserNotFoundException{
10        return userService.findOne(id);
11    }
12
13 }
```

@ModelAttribute

通过此注解，可以通过模型索引名称来访问已经存在于控制器中的model。下面是使用此注解的一个简单示例：

```
1 @PostMapping("user")
2 public void createUser(@ModelAttribute("user") User user){
3     userService.save(user);
4 }
```

与 `@PathVariable` 和 `@RequestParam` 注解一样，如果参数名与模型具有相同的名字，则不必指定索引名称，简写示例如下：

```
1 @PostMapping("/users")
2 public void createUser(@ModelAttribute User user){
3     userService.save(user);
4 }
```

特别地，如果使用 `@ModelAttribute` 对方法进行标注，Spring会将方法的返回值绑定到具体的Model上。示例如下：

```
1 @ModelAttribute("ramostear")
2 User getUser(){
3     User user = new User();
4     user.setId(1);
5     user.setFirstName("ramostear");
6     user.setEmail("ramostear@163.com")
7     //...
8     return user;
9 }
```

在Spring调用具体的处理方法之前，被 `@ModelAttribute` 注解标注的所有方法都将被执行。

@CrossOrigin

`@CrossOrigin` 注解将为请求处理类或请求处理方法提供跨域调用支持。如果我们将此注解标注类，那么类中的所有方法都将获得支持跨域的能力。使用此注解的好处是可以微调跨域行为。使用此注解的示例如下：

```
1 @CrossOrigin
2 @GetMapping("/users/home")
3 public String userDetails(@RequestParam(name="id",defaultValue="0")long id) throws
    UserNotFoundException{
4     if(id == 0){
5         return new User();
6     }
7     return userService.findOne(id).toString();
8 }
```

Watermark

@InitBinder

`@InitBinder` 注解用于标注初始化 `WebDataBinder` 的方法，该方法用于对Http请求传递的表单数据进行处理，如时间格式化、字符串处理等。下面是使用此注解的示例：

```
1 @InitBinder
2 public void initBinder(WebDataBinder dataBinder){
3
4     StringTrimmerEditor editor = new StringTrimmerEditor(true);
5     dataBinder.registerCustomEditor(String.class,editor);
6 }
```

二、Spring Bean 注解

在本小节中，主要列举与Spring Bean相关的4个注解以及它们的使用方式。

@ComponentScan

`@ComponentScan` 注解用于配置 Spring 需要扫描的被组件注解注释的类所在的包。可以通过配置其 `basePackages` 属性或者 `value` 属性来配置需要扫描的包路径。`value` 属性是 `basePackages` 的别名。

@Component

`@Component` 注解用于标注一个普通的组件类，它没有明确的业务范围，只是通知 Spring 被此注解的类需要被纳入到 Spring Bean 容器中并进行管理。此注解的使用示例如下：

```
1 @Component
2 public class EncryptUserPasswordComponent{
3
4
5     public String encrypt(String password, String salt){
6         // ...
7     }
8 }
```

@Service

`@Service` 注解是 `@Component` 的一个延伸（特例），它用于标注业务逻辑类。与 `@Component` 注解一样，被此注解标注的类，会自动被 Spring 所管理。下面是使用 `@Service` 注解的示例：

Watermark

```
1 public interface UserService{
2
3     User createUser(User user);
4 }
5
6 @Service("userService")
7 public class UserServiceImpl implements UserService{
8
9     @Autowired
10    private UserRepository userRepository;
11
12    @Override
13    public User createUser(User user){
14        return userRepository.save(user);
15    }
16 }
17
18 @RestController
19 @RequestMapping("/users")
20 public class UserController{
21
22    @Autowired
23    private UserService userService;
24
25    @PostMapping
26    public User createUser(@RequestBody User user){
27        return userService.createUser(user);
28    }
29 }
```

@Repository

`@Repository` 注解也是 `@Component` 注解的延伸，与 `@Component` 注解一样，被此注解标注的类会被Spring自动管理起来，`@Repository` 注解用于标注DAO层的数据持久化类。此注解的用法如下：

```
1 @Entity
2 @Table(name="t_user")
3 public class User{
4     @Id
5     @Column(name="USER_ID")
6     private Long id;
7     // ...
8 }
9
10 @Repository
11 public interface UserRepository extends JpaRepository<User, Long>{
12     // ...
13 }
```

@DependsOn

`@DependsOn` 注解可以配置Spring IoC容器在初始化一个Bean之前，先初始化其他的Bean对象。下面是此注解使用示例代码：

Watermark

```
1 public class FirstBean {  
2  
3     @Autowired  
4     private SecondBean secondBean;  
5  
6     @Autowired  
7     private ThirdBean thirdBean;  
8  
9     public FirstBean(){  
10        // ...  
11    }  
12    // ...  
13 }  
14  
15 public class SecondBean {  
16  
17     public SecondBean(){  
18         // ...  
19     }  
20 }  
21  
22 public class ThirdBean {  
23  
24     public ThirdBean(){  
25         // ...  
26     }  
27 }  
28  
29 @Configuration  
30 public class CustomBeanConfig{  
31  
32     @Bean("firstBean")  
33     @DependsOn(value={"secondBean", "thirdBean"})  
34     public FirstBean firstBean(){  
35         return new FirstBean();  
36     }  
37  
38     @Bean("secondBean")  
39     public SecondBean secondBean(){  
40  
41         return new SecondBean();  
42     }  
43 }  
44  
45  
46     @Bean("thirdBean")  
47     public ThirdBean thirdBean(){  
48  
49         return new ThirdBean();  
50     }  
51 }
```

Watermark

@Bean

@Bean注解主要的作用是告知Spring，被此注解所标注的类将需要纳入到Bean管理工厂中。@Bean注解的用法很简单，在这里，着重介绍@Bean注解中 `initMethod` 和 `destroyMethod` 的用法。示例如下：

```
1 @Component
2 public class DataBaseInitializer {
3
4     public void init(){
5
6         System.out.println("This is init method.");
7         // ...
8     }
9
10
11    public void destroy(){
12        System.out.println("This is destroy method.");
13        // ...
14    }
15
16 }
17
18 @Configuration
19 public class SpringBootApplicatioinConfig{
20
21
22     @Bean(initMethod="init",destroyMethod="destroy")
23     public DataBaseInitializer databaseInitializer(){
24
25         return new DataBaseInitializer();
26     }
27
28 }
```

@Scope

@Scope注解可以用来定义@Component标注的类的作用范围以及@Bean所标记的类的作用范围。
@Scope所限定的作用范围有：`singleton`、`prototype`、`request`、`session`、`globalSession`或者其他的自定义范围。这里以`prototype`为例子进行讲解。

当一个Spring Bean被声明为`prototype`（原型模式）时，在每次需要使用到该类的时候，Spring IoC容器都会初始化一个新的改类的实例。在定义一个Bean时，可以设置Bean的scope属性为`prototype`:
`scope=“prototype”`，也可以使用@Scope注解设置，如下：

```
@Scope(value=ConfigurableBeanFactory.SCOPE_PROTOTYPE)
```

下面将给出两种不同的方式来使用@Scope注解，示例代码如下：

Watermark

```
1 public interface UserService {  
2     // ...  
3 }  
4  
5  
6 @Component  
7 @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)  
8 public class UserServiceImpl implements UserService {  
9  
10    // ...  
11  
12 }  
13  
14  
15 @Configuration  
16 @ComponentScan(basePackages = "com.ramostear.service")  
17 public class ServiceConfig {  
18     // ...  
19 }  
20  
21 //-----  
22  
23 public class StudentService implements UserService {  
24     // ...  
25 }  
26  
27 @Configuration  
28 public class StudentServiceConfig {  
29  
30     @Bean  
31     @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)  
32     public UserService userService() {  
33  
34         return new StudentServiceImpl();  
35     }  
36 }  
37  
38 }  
39  
40
```

@Scope 单例模式

当@Scope的作用范围设置成Singleton时，被此注解所标注的类只会被Spring IoC容器初始化一次。在默认情况下，Spring IoC容器所初始化的类实例都为singleton。同样的原理，此情形也有两种配置方式，示例代码如下：

Watermark

```
1 public interface UserService {
2     // ...
3 }
4
5
6 @Component
7 @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
8 public class UserServiceImpl implements UserService {
9
10    // ...
11
12 }
13
14
15 @Configuration
16 @ComponentScan(basePackages = "com.ramostear.service")
17 public class ServiceConfig {
18     // ...
19 }
20
21 //-----
22
23 public class StudentService implements UserService {
24     // ...
25 }
26
27 @Configuration
28 public class StudentServiceConfig {
29
30     @Bean
31     @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
32     public UserService userService() {
33
34         return new StudentServiceImpl();
35
36     }
37
38 }
39
40
```

四、容器配置注解

@Autowired

@Autowired注解用于标记Spring将要解析和注入的依赖项。此注解可以作用在构造函数、字段和setter方法上。

作用于构造函数

下面是@.Autowired注解标注构造函数的使用示例：

Watermark

```
1 @RestController
2 public class UserController {
3
4     private UserService userService;
5
6     @Autowired
7     UserController(UserService userService){
8         this.userService = userService;
9     }
10
11    // ...
12
13 }
```

作用于setter方法

下面是@Autowire注解标注setter方法的示例代码：

```
1 @RestController
2 public class UserController {
3
4     private UserService userService;
5
6     @Autowired
7     public void setUserService(UserService userService){
8         this.userService = userService;
9     }
10
11    // ...
12
13 }
```

作用于字段

@Autowire注解标注字段是最简单的，只需要在对应的字段上加入此注解即可，示例代码如下：

```
1 @RestController
2 public class UserController {
3
4     @Autowired
5     private UserService userService;
6
7     // ...
8
9 }
```

Watermark

@Primary

当系统中需要配置多个具有相同类型的bean时，@Primary可以定义这些Bean的优先级。下面将给出一个实例代码来说明这一特性：

```
 1 public interface MessageService {  
 2     String sendMessage();  
 3 }  
 4  
 5 @Component  
 6 public class EmailMessageServiceImpl implements MessageService {  
 7  
 8     @Override  
 9     public String sendMessage(){  
10         return "this is send email method message.";  
11     }  
12 }  
13  
14 @Component  
15 public class WechatMessageImpl implements MessageService {  
16  
17     @Override  
18     public String sendMessage(){  
19  
20         return "this is send wechat method message.";  
21     }  
22 }  
23  
24  
25 @Primary  
26 @Component  
27 public class DingDingMessageImpl implements MessageService {  
28  
29     @Override  
30     public String sendMessage(){  
31         return "this is send DingDing method message.";  
32     }  
33 }  
34  
35  
36  
37 @RestController  
38 public class MessageController {  
39  
40     @Autowired  
41     private MessageService messageService;  
42  
43  
44     @GetMapping("/info")  
45     public String info(){  
46         return messageService.sendMessage();  
47     }  
48 }  
49
```

输出结果：
Watermark

this is send DingDing method message.

@PostConstruct与@PreDestroy

值得注意的是，这两个注解不属于Spring，它们是源于JSR-250中的两个注解，位于 `common-annotations.jar` 中。@PostConstruct注解用于标注在Bean被Spring初始化之前需要执行的方法。@PreDestroy注解用于标注Bean被销毁前需要执行的方法。下面是具体的示例代码：

```
1 @Component
2 public class DemoComponent {
3
4     private List<String> list = new ArrayList();
5
6     @PostConstruct
7     public void init(){
8         list.add("hello");
9         list.add("world");
10    }
11
12
13     @PreDestroy
14     public void destroy(){
15
16         list.clear();
17
18    }
19
20 }
```

@Qualifier

当系统中存在同一类型的多个Bean时，@Autowired在进行依赖注入的时候就不知道该选择哪一个实现类进行注入。此时，我们可以使用@Qualifier注解来微调，帮助@Autowired选择正确的依赖项。下面是一个关于此注解的代码示例：

Watermark

```
1 public interface MessageService {  
2  
3     public String sendMessage(String message);  
4  
5 }  
6  
7 @Service("emailService")  
8 public class EmailServiceImpl implements MessageService {  
9  
10    @Override  
11    public String sendMessage(String message){  
12        return "send email,content:"+message;  
13    }  
14  
15 }  
16  
17 @Service("smsService")  
18 public class SMSServiceImpl implements MessageService{  
19  
20    @Override  
21    public String sendMessage(String message){  
22        return "send SMS,content:"+message;  
23    }  
24 }  
25  
26  
27 public interface MessageProcessor {  
28     public String processMessage(String message);  
29 }  
30  
31  
32 public class MessageProcessorImpl implements MessageProcessor{  
33  
34     private MessageService messageService;  
35  
36     @Autowired  
37     @Qualifier("emailService")  
38     public void setMessageService(MessageService messageService){  
39         this.messageService = messageService;  
40     }  
41  
42     @Override  
43     public String processMessage(String message){  
44         return messageService.sendMessage(message);  
45     }  
46  
47 }
```

五、Spring Boot注解

@SpringBootApplication

`@SpringBootApplication` 主要是对一个快现的配置注解，在被它标注的类中，可以定义一个或多个Bean，并自动触发自动配置Bean和自动扫描组件。此注解相当于 `@Configuration`、`@EnableAutoConfiguration` 和 `@ComponentScan` 的组合。

在Spring Boot应用程序的主类中，就使用了此注解。示例代码如下：

```
@SpringBootApplication
public class Application{
    public static void main(String [] args){
        SpringApplication.run(Application.class, args);
    }
}
```

@EnableAutoConfiguration

@EnableAutoConfiguration注解用于通知Spring，根据当前类路径下引入的依赖包，自动配置与这些依赖包相关的配置项。

@ConditionalOnClass与@ConditionalOnMissingClass

这两个注解属于类条件注解，它们根据是否存在某个类作为判断依据来决定是否要执行某些配置。下面是一个简单的示例代码：

```
@Configuration
@ConditionalOnClass(DataSource.class)
class MySQLAutoConfiguration {
    //...
}
```

@ConditionalOnBean与@ConditionalOnMissingBean

这两个注解属于对象条件注解，根据是否存在某个对象作为依据来决定是否要执行某些配置方法。示例代码如下：

```
@Bean
@ConditionalOnBean(name="dataSource")
LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    //...
}

@Bean
@ConditionalOnMissingBean
public MyBean myBean() {
    //...
}
```

@ConditionalOnProperty

@ConditionalOnProperty注解会根据Spring配置文件中的配置项是否满足配置要求，从而决定是否要执行被其标注的方法。示例代码如下：

```
@Bean
@ConditionalOnProperty(name="alipay", havingValue="on")
Alipay alipay() {
    return new Alipay();
}
```

@ConditionalOnResource

此注解用于检测当某个配置文件存在时，则触发被其标注的方法，下面是使用此注解的代码示例：

```
@ConditionalOnResource(resources = "classpath:website.properties")
Properties addWebsiteProperties() {
    //...
}
```

@ConditionalOnWebApplication与 @ConditionalOnNotWebApplication

这两个注解用于判断当前的应用程序是否是Web应用程序。如果当前应用是Web应用程序，则使用Spring WebApplicationContext，并定义其会话的生命周期。下面是一个简单的示例：

```
@ConditionalOnWebApplication
HealthCheckController healthCheckController() {
    //...
}
```

@ConditionalExpression

此注解可以让我们控制更细粒度的基于表达式的配置条件限制。当表达式满足某个条件或者表达式为真的时候，将会执行被此注解标注的方法。

```
@Bean
@ConditionalException("${localstore} && ${local == 'true'}")
LocalFileStore store() {
    //...
}
```

@Conditional

@Conditional注解可以控制更为复杂的配置条件。在Spring内置的条件控制注解不满足应用需求的时候，可以使用此注解定义自定义的控制条件，以达到自定义的要求。下面是使用该注解的简单示例：

```
@Conditional(CustomCondition.class)
CustomProperties addCustomProperties() {
    //...
}
```

六、总结

本次课程总结了Spring Boot中常见的各类型注解的使用方式，让大家能够统一的对Spring Boot常用注解有一个全面的了解。

SpringBoot巧用 @Async 提升API接口并发能力

异步调用几乎是处理高并发Web应用性能问题的万金油，那么什么是“异步调用”？

“异步调用”对应的是“同步调用”，同步调用指程序按照定义顺序依次执行，每一行程序都必须等待上一行程序执行完成之后才能执行；异步调用指程序在顺序执行时，不等待异步调用的语句返回结果就执行后面的程序。

同步调用

下面通过一个简单示例来直观的理解什么是同步调用：

定义Task类，创建三个处理函数分别模拟三个执行任务的操作，操作消耗时间随机取（10秒内）

```
@Component
public class Task {

    public static Random random = new Random();

    public void doTaskOne() throws Exception {
        System.out.println("开始做任务一");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务一，耗时：" + (end - start) + "毫秒");
    }

    public void doTaskTwo() throws Exception {
        System.out.println("开始做任务二");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务二，耗时：" + (end - start) + "毫秒");
    }

    public void doTaskThree() throws Exception {
        System.out.println("开始做任务三");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务三，耗时：" + (end - start) + "毫秒");
    }
}
```

在单元测试用例中，注入Task对象，并在测试用例中执行doTaskOne、doTaskTwo、doTaskThree三个函数。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = Application.class)
public class ApplicationTests {
```

```
@Autowired  
private Task task;  
  
@Test  
public void test() throws Exception {  
    task.doTaskOne();  
    task.doTaskTwo();  
    task.doTaskThree();  
}  
}
```

执行单元测试，可以看到类似如下输出：

```
开始做任务一  
完成任务一，耗时：4256毫秒  
开始做任务二  
完成任务二，耗时：4957毫秒  
开始做任务三  
完成任务三，耗时：7173毫秒
```

任务一、任务二、任务三顺序的执行完了，换言之doTaskOne、doTaskTwo、doTaskThree三个函数顺序的执行完成。

异步调用

上述的同步调用虽然顺利的执行完了三个任务，但是可以看到执行时间比较长，若这三个任务本身之间不存在依赖关系，可以并发执行的话，同步调用在执行效率方面就比较差，可以考虑通过异步调用的方式来并发执行。另外关注公众号：码猿技术专栏，回复关键词“面试宝典”送你一份阿里内部面试资料！

在Spring Boot中，我们只需要通过使用@Async注解就能简单的将原来的同步函数变为异步函数，Task类改在为如下模式：

```
@Component  
public class Task {  
  
    @Async  
    public void doTaskOne() throws Exception {  
        // 同上内容，省略  
    }  
  
    @Async  
    public void doTaskTwo() throws Exception {  
        // 同上内容，省略  
    }  
  
    @Async  
    public void doTaskThree() throws Exception {  
        // 同上内容，省略  
    }  
}
```

为了让@Async注解能够生效，还需要在Spring Boot的主程序中配置@EnableAsync，如下所示：

```
@SpringBootApplication
@EnableAsync
public class Application {

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

}
```

此时可以反复执行单元测试，您可能会遇到各种不同的结果，比如：

- 没有任何任务相关的输出
- 有部分任务相关的输出
- 乱序的任务相关的输出

原因是目前doTaskOne、doTaskTwo、doTaskThree三个函数的时候已经是异步执行了。主程序在异步调用之后，主程序并不会理会这三个函数是否执行完成了，由于没有其他需要执行的内容，所以程序就自动结束了，导致了不完整或是没有输出任务相关内容的情况。

注：@Async所修饰的函数不要定义为static类型，这样异步调用不会生效

异步回调

为了让doTaskOne、doTaskTwo、doTaskThree能正常结束，假设我们需要统计一下三个任务并发执行共耗时多少，这就需要等到上述三个函数都完成调动之后记录时间，并计算结果。

那么我们如何判断上述三个异步调用是否已经执行完成呢？我们需要使用Future来返回异步调用的结果，就像如下方式改造doTaskOne函数：

```
@Async
public Future<String> doTaskOne() throws Exception {
    System.out.println("开始做任务一");
    long start = System.currentTimeMillis();
    Thread.sleep(random.nextInt(10000));
    long end = System.currentTimeMillis();
    System.out.println("完成任务一，耗时：" + (end - start) + "毫秒");
    return new AsyncResult<>("任务一完成");
}
```

按照如上方式改造一下其他两个异步函数之后，下面我们改造一下测试用例，让测试在等待完成三个异步调用之后来做一些其他事情。

```
@Test
public void test() throws Exception {

    long start = System.currentTimeMillis();

    Future<String> task1 = task.doTaskOne();
    Future<String> task2 = task.doTaskTwo();
    Future<String> task3 = task.doTaskThree();

    while(true) {
        if(task1.isDone() && task2.isDone() && task3.isDone()) {
            // 三个任务都调用完成，退出循环等待
    }
}
```

```
        break;
    }
    Thread.sleep(1000);
}

long end = System.currentTimeMillis();

System.out.println("任务全部完成，总耗时：" + (end - start) + "毫秒");

}
```

看看我们做了哪些改变：

- 在测试用例一开始记录开始时间
- 在调用三个异步函数的时候，返回Future类型的结果对象
- 在调用完三个异步函数之后，开启一个循环，根据返回的Future对象来判断三个异步函数是否都结束了。若都结束，就结束循环；若没有都结束，就等1秒后再判断。

跳出循环之后，根据 **结束时间 - 开始时间**，计算出三个任务并发执行的总耗时。

执行一下上述的单元测试，可以看到如下结果：

```
开始做任务一
开始做任务二
开始做任务三
完成任务三，耗时：37毫秒
完成任务二，耗时：3661毫秒
完成任务一，耗时：7149毫秒
任务全部完成，总耗时：8025毫秒
```

可以看到，通过异步调用，让任务一、二、三并发执行，有效的减少了程序的总运行时间。

SpringBoot+WebSocket 实时监控异常

写在前面

此异常非彼异常，标题所说的异常是业务上的异常。

最近做了一个需求，消防的设备巡检，如果巡检发现异常，通过手机端提交，后台的实时监控页面实时获取到该设备的信息及位置，然后安排员工去处理。

因为需要服务端主动向客户端发送消息，所以很容易的就想到了用WebSocket来实现这一功能。

WebSocket就不做介绍了，上链接：

<https://developer.mozilla.org/zh-CN/docs/Web/API/WebSocket>

前端略微复杂，需要在一张位置分布图上进行鼠标描点定位各个设备和根据不同屏幕大小渲染，本文不做介绍，只是简单地用页面样式进行效果呈现。

绿色代表正常，红色代表异常

预期效果，未接收到请求前----->id为3的提交了异常，id为3的王五变成了红色

- | | |
|-------|---|
| 1.张三 | ● |
| 2.李四 | ● |
| 3.王五 | ● |
| 4.韩梅梅 | ● |
| 5.李磊 | ● |

实现

前端：

直接贴代码

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>实时监控</title>
    </head>
    <style>
        .item {
            display: flex;
            border-bottom: 1px solid #000000;
            justify-content: space-between;
            width: 30%;
            line-height: 50px;
            height: 50px;
        }

        .item span:nth-child(2) {
            margin-right: 10px;
            margin-top: 15px;
            width: 20px;
            height: 20px;
            border-radius: 50%;
            background: #55ff00;
        }

        .nowI{
            background: #ff0000 !important;
        }
    </style>
    <body>
        <div v-for="item in list" class="item">
            <span>{{item.id}}.{{item.name}}</span>
            <span :class='item.state==1?"nowI":""'></span>
        </div>
    </body>

```

```

        </div>
    </body>
    <script src=". ./js/vue.min.js"></script>
    <script type="text/javascript">
        var vm = new Vue({
            el: "#app",
            data: {
                list: [
                    {
                        id: 1,
                        name: '张三',
                        state: 1
                    },
                    {
                        id: 2,
                        name: '李四',
                        state: 1
                    },
                    {
                        id: 3,
                        name: '王五',
                        state: 1
                    },
                    {
                        id: 4,
                        name: '韩梅梅',
                        state: 1
                    },
                    {
                        id: 5,
                        name: '李磊',
                        state: 1
                    }
                ]
            }
        })
    
```

```

        var webSocket = null;
        if ('WebSocket' in window) {
            //创建WebSocket对象
            webSocket = new WebSocket("ws://localhost:18801/webSocket/" + getUUID());

            //连接成功
            webSocket.onopen = function() {
                console.log("已连接");
                webSocket.send("消息发送测试")
            }
            //接收到消息
            webSocket.onmessage = function(msg) {
                //处理消息
                var serverMsg = msg.data;
                var t_id = parseInt(serverMsg)      //服务端发过来的消息，ID，string需转化为int
                for (var i = 0; i < vm.list.length; i++) {
                    var item = vm.list[i];
                    if(item.id == t_id){
                        item.state = -1;
                        vm.list.splice(i, 1, item)
                        break;
                    }
                }
            }
        }
    
```

```

        }
    }
};

//关闭事件
webSocket.onclose = function() {
    console.log("websocket已关闭");
};

//发生了错误事件
webSocket.onerror = function() {
    console.log("websocket发生了错误");
}

} else {
    alert("很遗憾，您的浏览器不支持WebSocket!")
}

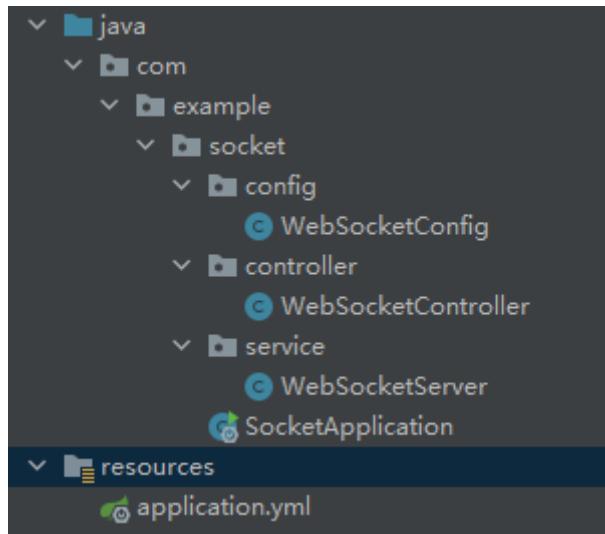
function getUUID() { //获取唯一的UUID
    return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxx'.replace(/[xy]/g, function(c) {

        var r = Math.random() * 16 | 0,
            v = c == 'x' ? r : (r & 0x3 | 0x8);
        return v.toString(16);
    });
}
</script>
</html>

```

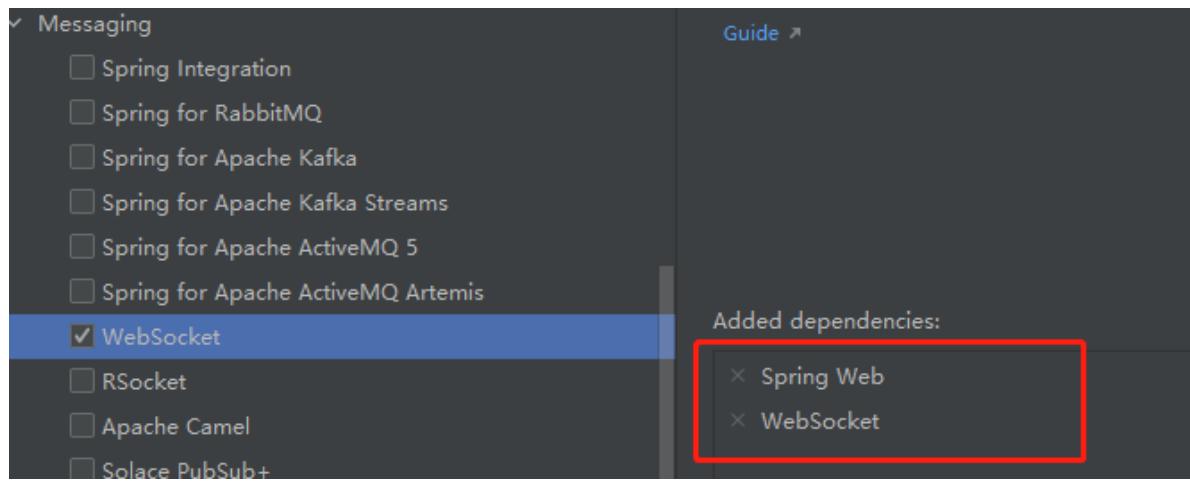
后端：

项目结构是这样子的，后面的代码关键注释都有，就不重复描述了



1、新建SpringBoot工程，选择web和WebSocket依赖

Watermark



2、配置application.yml

```
#端口
server:
  port: 18801

#密码，因为接口不需要权限，所以加了个密码做校验
mySocket:
  myPwd: jae_123
```

3、WebSocketConfig配置类

```
@Configuration
public class WebSocketConfig {

    /**
     * 注入一个ServerEndpointExporter, 该Bean会自动注册使用@ServerEndpoint注解申明的websocket endpoint
     */
    @Bean
    public ServerEndpointExporter serverEndpointExporter() {
        return new ServerEndpointExporter();
    }
}
```

4、WebSocketServer类，用来进行服务端和客户端之间的交互

```
/**
 * @author jae
 * @ServerEndpoint("/webSocket/{uid}") 前端通过此URI与后端建立链接
 */

@Component
public class WebSocketServer {

    private static Logger log = LoggerFactory.getLogger(WebSocketServer.class);

    //静态变量，用来记录当前在线连接数。应该把它设计成线程安全的。
    private static AtomicInteger allNum = new AtomicInteger(0);

    //concurrent包的线程安全Set，用来存放每个客户端对应的WebSocketServer对象。
    private static CopyOnWriteArraySet<Session> sessionPools = new CopyOnWriteArraySet<Session>();

    /**

```

```

    * 有客户端连接成功
    */
@OnOpen
public void onOpen(Session session, @PathParam(value = "uid") String uid) {
    sessionPools.add(session);
    onlineNum.incrementAndGet();
    log.info(uid + "加入WebSocket! 当前人数为" + onlineNum);
}

/**
 * 连接关闭调用的方法
 */
@OnClose
public void onClose(Session session) {
    sessionPools.remove(session);
    int cnt = onlineNum.decrementAndGet();
    log.info("有连接关闭, 当前连接数为: {}", cnt);
}

/**
 * 发送消息
 */
public void sendMessage(Session session, String message) throws IOException {
    if(session != null) {
        synchronized (session) {
            session.getBasicRemote().sendText(message);
        }
    }
}

/**
 * 群发消息
 */
public void broadCastInfo(String message) throws IOException {
    for (Session session : sessionPools) {
        if(session.isOpen()) {
            sendMessage(session, message);
        }
    }
}

/**
 * 发生错误
 */
@OnError
public void onError(Session session, Throwable throwable) {
    log.error("发生错误");
    throwable.printStackTrace();
}
}

```

5、WebSocketController类，进行接口测试

```

@RestController
@RequestMapping("/open/socket")
public class WebSocketController {

```

```

@Value("${mySocket.myPwd}")
public String myPwd;

@.Autowired
private WebSocketServer webSocketServer;

/**
 * 手机客户端请求接口
 * @param id    发生异常的设备ID
 * @param pwd    密码（实际开发记得加密）
 * @throws IOException
 */
@PostMapping(value = "/onReceive")
public void onReceive(String id, String pwd) throws IOException {
    if(pwd.equals(myPwd)){ //密码校验一致（这里举例，实际开发还要有个密码加密的校验的），则进行群发
        webSocketServer.broadcastInfo(id);
    }
}

```

测试

1、打开前端页面，进行WebSocket连接

控制台输出，连接成功

```
: 18216d86-3f66-4caa-9440-3e5409481fe8加入WebSocket! 当前人数为1
```

2、因为是模拟数据，所以全部显示正常，没有异常提交时的页面呈现

1.张三	
2.李四	
3.王五	
4.韩梅梅	
5.李磊	

3、接下来，我们用接口测试工具Postman提交一个异常

Watermark

POST localhost:18801/open/socket/onReceive?id=3&pwd=jae_123

Params ● Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESC
<input checked="" type="checkbox"/> id	3	
<input checked="" type="checkbox"/> pwd	jae_123	
Key	Value	Desc

注意id为3的数据的状态变化

1.张三	
2.李四	
3.王五	
4.韩梅梅	
5.李磊	

我们可以看到，id为3的王五状态已经变成异常的了，实时通讯成功。

参考：

<https://developer.mozilla.org/zh-CN/docs/Web/API/WebSocket>

Spring Boot Security+JWT前后端分离架构 登录认证！

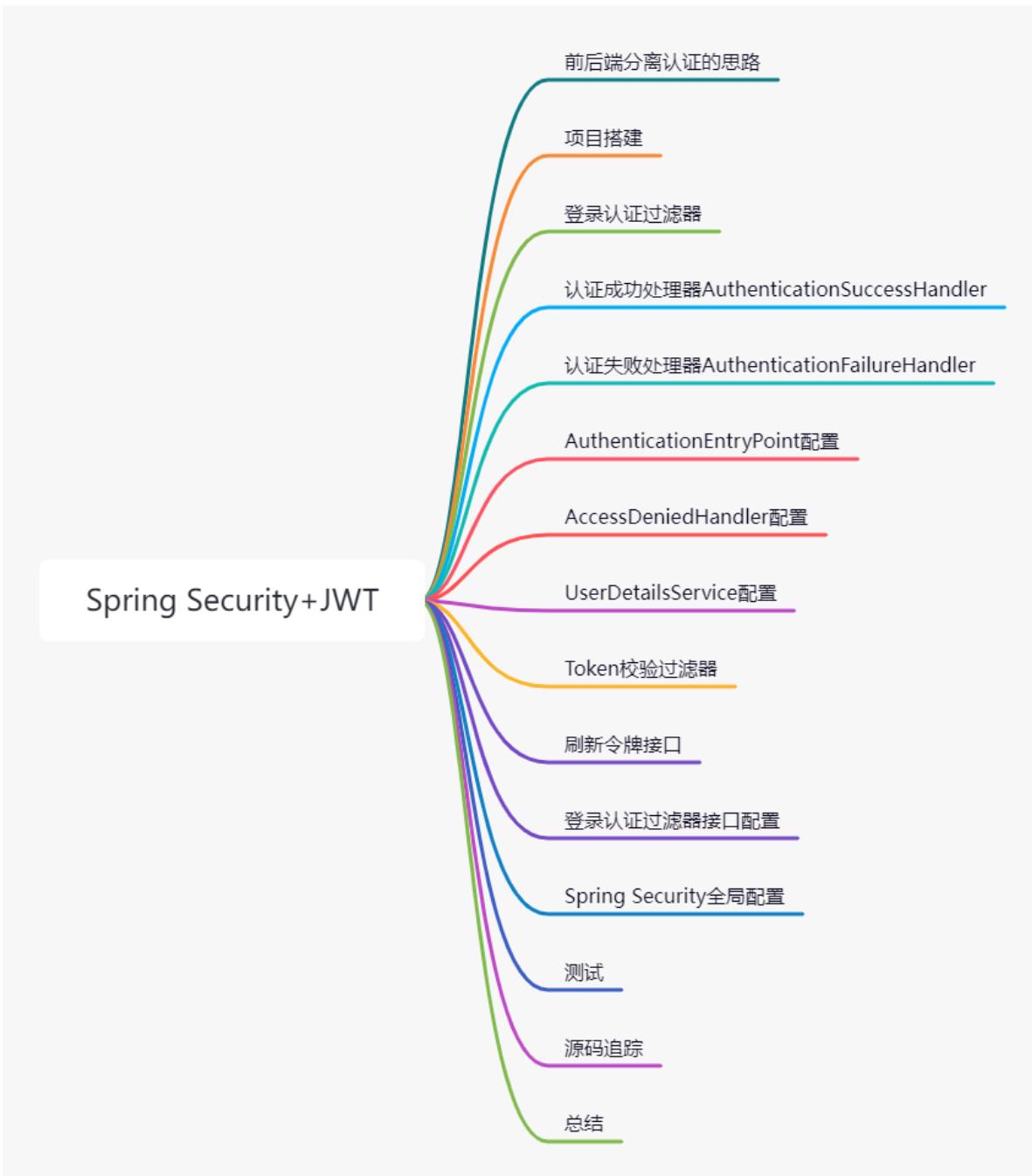
大家好，我是不才陈某~

认证、授权是实战项目中必不可少的部分，而Spring Security则将作为首选安全组件，因此陈某新开了《Spring Security 进阶》这个专栏，写一写从单体架构到OAuth2分布式架构的认证授权。

Spring security这里就不再过多介绍了，相信大家都用过，也都恐惧过，相比Shiro而言，Spring Security更加重量级，之前的SSM项目更多企业都是用的Shiro，但是Spring Boot出来之后，整合Spring Security更加方便了，用的企业也就多了。

今天陈某就来介绍一下在前后端分离的项目中如何使用Spring Security进行登录认证。文章的目录如下：

Watermark



前后端分离认证的思路

前后端分离不同于传统的web服务，无法使用session，因此我们采用JWT这种无状态机制来生成token，大致的思路如下：

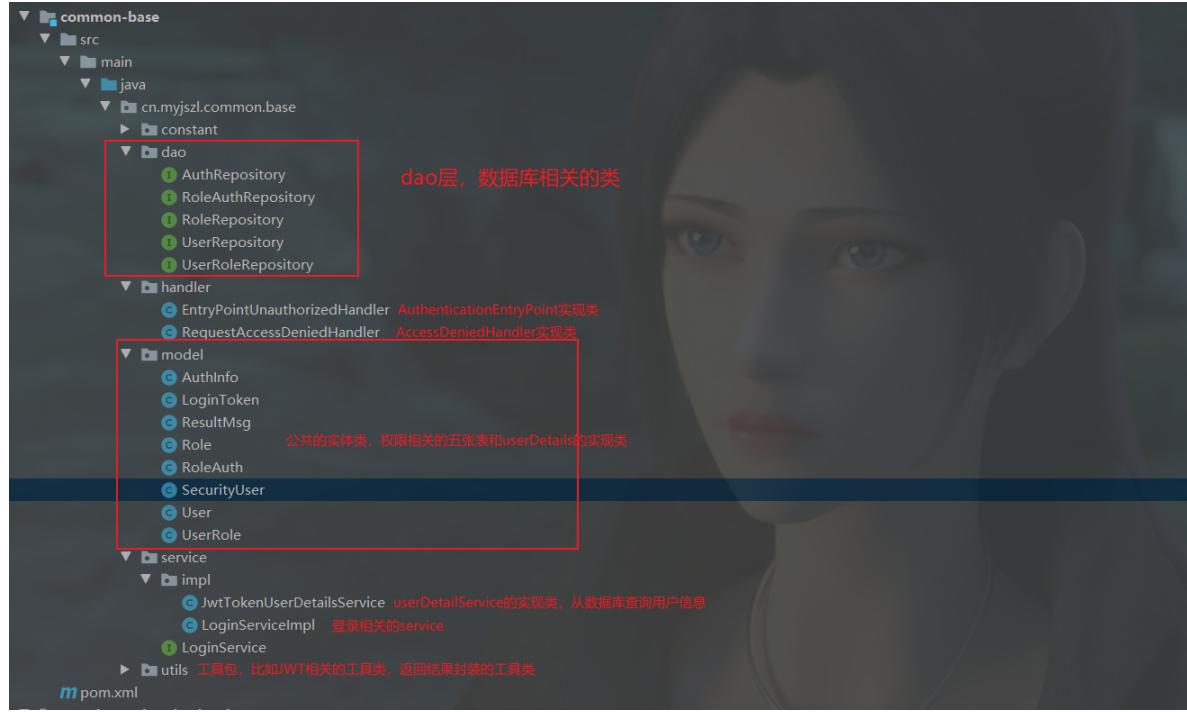
1. 客户端调用服务端登录接口，输入用户名、密码登录，登录成功返回两个**token**，如下：
 1. **accessToken**：客户端携带这个token访问服务端的资源
 2. **refreshToken**：刷新令牌，一旦accessToken过期了，客户端需要使用refreshToken重新获取一个accessToken。因此refreshToken的过期时间一般大于accessToken。
2. 客户请求头中携带**accessToken**访问服务端的资源，服务端对**accessToken**进行鉴定（验签、是否有效...），如果这个**accessToken**没有问题则放行。
3. **accessToken**一旦过期需要客户端携带**refreshToken**调用刷新令牌的接口重新获取一个新的**accessToken**。

项目搭建

陈某使用的是Spring Boot 框架，演示项目新建了两个模块，分别是 common-base、 security-authentication-jwt。

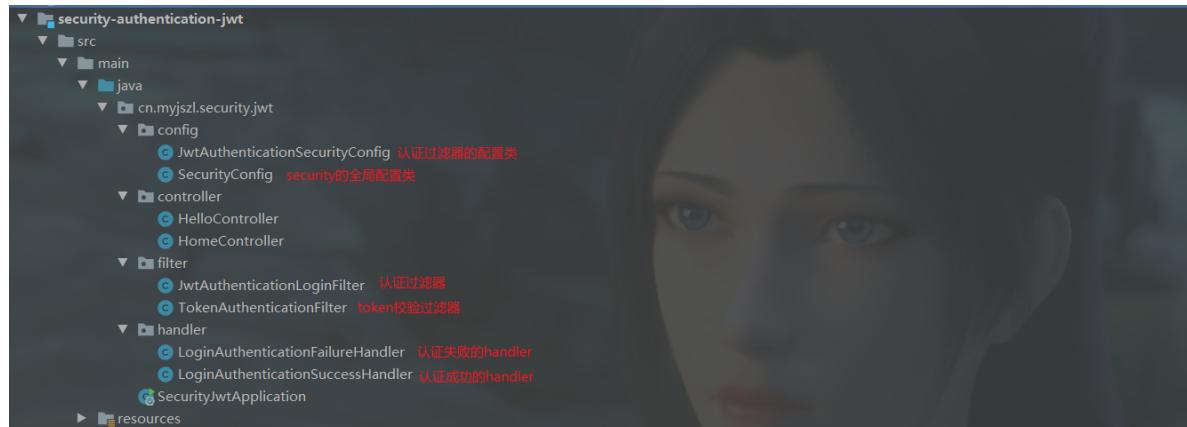
1、common-base模块

这是一个抽象出来的公共模块，这个模块主要放一些公用的类，目录如下：



2、security-authentication-jwt模块

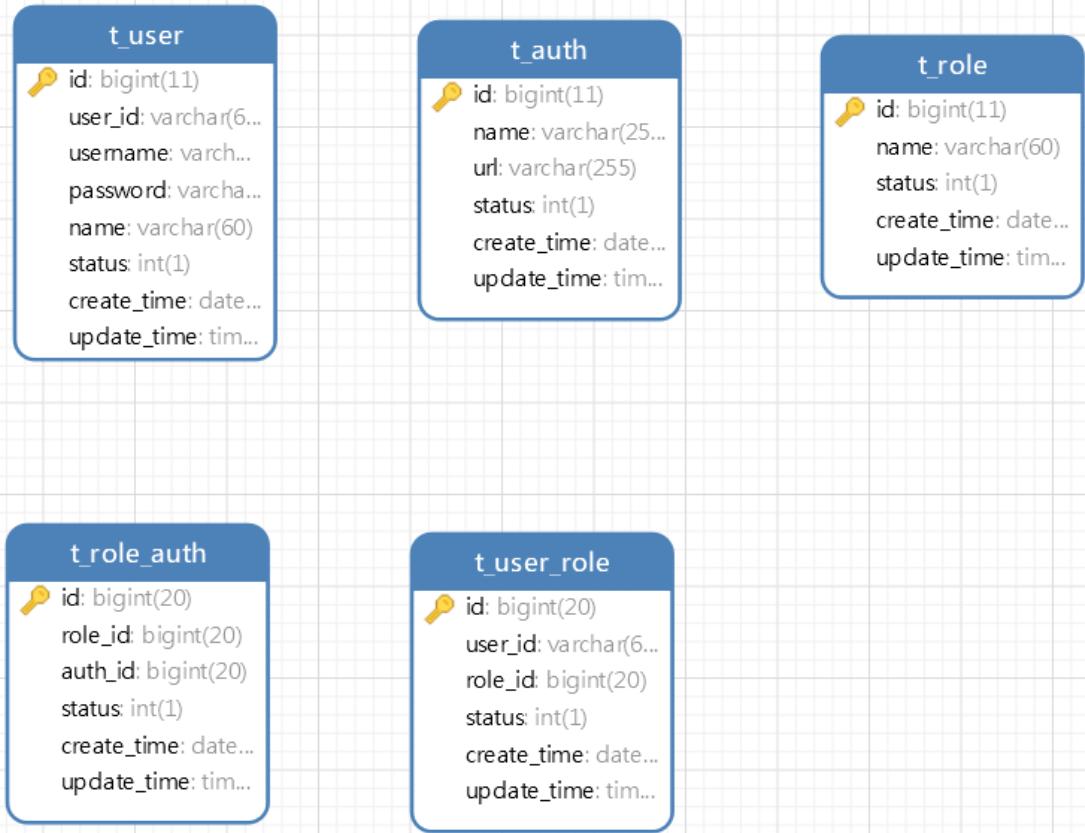
一些需要定制的类，比如security的全局配置类、Jwt登录过滤器的配置类，目录如下：



3、五张表

权限设计根据业务的需求往往有不同的设计，陈某用的**RBAC**规范，主要涉及到五张表，分别是**用户表、角色表、权限表、用户<->角色表、角色<->权限表**，如下图：

Watermark



上述几张表的SQL会放在案例源码中（这几张表字段为了省事，设计的并不全，自己根据业务逐步拓展即可）

登录认证过滤器

登录接口的逻辑写法有很多种，今天陈某介绍一种使用过滤器的定义的登录接口。

Spring Security默认的表单登录认证的过滤器是 `UsernamePasswordAuthenticationFilter`，这个过滤器并不适用于前后端分离的架构，因此我们需要自定义一个过滤器。

逻辑很简单，参照 `UsernamePasswordAuthenticationFilter` 这个过滤器改造一下，代码如下：

```

/**
 * @author 公众号：码猿技术专栏
 * 登录认证的filter，参照UsernamePasswordAuthenticationFilter，添加到这之前的过滤器
 */
public class JwtAuthenticationLoginFilter extends AbstractAuthenticationProcessingFilter {

    /**
     * 构造方法，调用父类的，设置登录地址/login，请求方式POST
     */
    public JwtAuthenticationLoginFilter() { super(new AntPathRequestMatcher( pattern: "/login", httpMethod: "POST")); }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) {
        //获取表单提交数据
        String username = request.getParameter(name: "username");
        String password = request.getParameter(name: "password");
        //封装到TOKEN中提交
        UsernamePasswordAuthenticationToken authRequest = new UsernamePasswordAuthenticationToken(
            username, password);
        return getAuthenticationManager().authenticate(authRequest);
    }
}

```

认证成功处理器AuthenticationSuccessHandler

上述的过滤器接口一旦认证成功，则会调用**AuthenticationSuccessHandler**进行处理，因此我们可以自定义一个认证成功处理器进行自己的业务处理，代码如下：

```
/*
 * 登录成功操作
 * @author 公众号: 码猿技术专栏
 */
@Component
@Slf4j
public class LoginAuthenticationSuccessHandler implements AuthenticationSuccessHandler {

    @Autowired
    private JwtUtils jwtTokenUtil;

    @Override
    public void onAuthenticationSuccess(HttpServletRequest httpServletRequest,
                                       HttpServletResponse httpServletResponse,
                                       Authentication authentication) throws IOException {
        UserDetails userDetails = (UserDetails) authentication.getPrincipal();
        SecurityContextHolder.getContext().setAuthentication(authentication);
        //TODO 根据业务需要进行处理, 陈某这里只返回两个token
        //生成令牌
        String accessToken = jwtTokenUtil.createToken(userDetails.getUsername());
        //生成刷新令牌, 如果accessToken令牌失效, 则使用refreshToken重新获取令牌 (refreshToken过期时间必须大于accessToken)
        String refreshToken = jwtTokenUtil.refreshToken(accessToken);
        renderToken(httpServletResponse, LoginToken.builder().accessToken(accessToken).refreshToken(refreshToken).build());
    }

    /**
     * 渲染返回 token 数据, 因为前端页面接收的都是Result对象, 故使用application/json返回
     */
    public void renderToken(HttpServletResponse response, LoginToken token) throws IOException {
        ResponseUtils.result(response,new ResultMsg(code: 200, msg: "登录成功!", token));
    }
}
```

陈某仅仅返回了**accessToken**、**refreshToken**，其他的业务逻辑处理自己完善。

认证失败处理器AuthenticationFailureHandler

同样的，一旦登录失败，比如用户名或者密码错误等等，则会调用**AuthenticationFailureHandler**进行处理，因此我们需要自定义一个认证失败的处理器，其中根据异常信息返回特定的**JSON**数据给客户端，代码如下：

Watermark

```
/**  
 * 登录失败操作  
 * @author 公众号: 码猿技术专栏  
 */  
  
@Component  
public class LoginAuthenticationFailureHandler implements AuthenticationFailureHandler {  
    /**  
     * 一旦登录失败则会被调用  
     * @param httpServletRequest  
     * @param response  
     * @param exception 这个参数是异常信息, 可以根据不同的异常类返回不同的提示信息  
     * @throws IOException  
     * @throws ServletException  
     */  
  
    @Override  
    public void onAuthenticationFailure(HttpServletRequest httpServletRequest,  
                                         HttpServletResponse response,  
                                         AuthenticationException exception) throws IOException {  
  
        //TODO 根据项目需要返回指定异常提示, 陈某这里演示了一个用户名密码错误的异常  
        //BadCredentialsException 这个异常一般是用户名或者密码错误  
        if (exception instanceof BadCredentialsException){  
            ResponseUtils.result(response,new ResultMsg( code: 200, msg: "用户名或密码不正确!", data: null));  
        }  
        ResponseUtils.result(response,new ResultMsg( code: 200, msg: "登录失败", data: null));  
    }  
}
```

逻辑很简单，**AuthenticationException**有不同的实现类，根据异常的类型返回特定的提示信息即可。

AuthenticationEntryPoint配置

AuthenticationEntryPoint这个接口当用户未通过认证访问受保护的资源时，将会调用其中的**commence()**方法进行处理，比如客户端携带的token被篡改，因此我们需要自定义一个**AuthenticationEntryPoint**返回特定的提示信息，代码如下：

```
/**  
 * @author 公众号: 码猿技术专栏  
 * 用户访问受保护的资源, 但是用户没有通过认证则会进入这个处理器  
 */  
  
@Component  
@Slf4j  
public class EntryPointUnauthorizedHandler implements AuthenticationEntryPoint {  
  
    @Override  
    public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException) throws IOException {  
        ResponseUtils.result(response,new ResultMsg( code: 403, msg: "无权限访问, 请先登录!", data: null));  
    }  
}
```

AccessDeniedHandler配置

AccessDeniedHandler这处理器当认证成功的用户访问受保护的资源，但是**权限不够**，则会进入这个处理器进行处理，我们可以实现这个处理器返回特定的提示信息给客户端，代码如下：

Watermark

```
/**  
 * @author 公众号: 码猿技术专栏  
 *  
 * 当认证后的用户访问受保护的资源时, 权限不够, 则会进入这个处理器  
 */  
  
@Component  
public class RequestAccessDeniedHandler implements AccessDeniedHandler {  
    @Override  
    public void handle(HttpServletRequest request,  
                        HttpServletResponse response,  
                        AccessDeniedException accessDeniedException) throws IOException {  
        ResponseUtils.result(response, new ResultMsg( code: 403, msg: "权限不足!", data: null));  
    }  
}
```

UserDetailsService配置

UserDetailsService这个类是用来加载用户信息，包括**用户名、密码、权限、角色**集合....其中有一个方法如下：

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
```

在认证逻辑中Spring Security会调用这个方法根据客户端传入的username加载该用户的详细信息，这个方法需要完成的逻辑如下：

- 密码匹配
- 加载权限、角色集合

我们需要实现这个接口，从**数据库**加载用户信息，代码如下：

```
/**  
 * @author 公众号: 码猿技术专栏  
 * 从数据库中根据用户名查询用户的详细信息, 包括权限  
 *  
 * 数据库设计: 角色、用户、权限、角色<->权限、用户<->角色 总共五张表, 遵循RBAC设计  
 */  
  
@Service  
public class JwtTokenUserDetailsService implements UserDetailsService {  
  
    /**  
     * 查询用户详情的服务  
     */  
    @Autowired  
    private LoginService loginService;  
  
    @Override  
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
        //从数据库中查询  
        SecurityUser securityUser = loginService.loadByUsername(username);  
        //用户不存在直接抛出UsernameNotFoundException, security会捕获抛出BadCredentialsException  
        if (Objects.isNull(securityUser))  
            throw new UsernameNotFoundException("用户不存在!");  
        return securityUser;  
    }  
}
```

其中的**LoginService**是根据用户名从数据库中查询出密码、角色、权限，代码如下：

```

    @Nullable
    @Override
    public SecurityUser loadByUsername(String username) {
        //获取用户信息
        User user = userRepository.findByUsernameAndStatus(username, LoginConstant.USER_USED);
        if (Objects.nonNull(user)){
            SecurityUser securityUser = new SecurityUser();
            securityUser.setUsername(username);
            //todo 此处为了方便，直接在数据库存储的明文，实际生产中应该存储密文，则这里不用再次加密
            securityUser.setPassword(passwordEncoder.encode(user.getPassword()));
            //查询该用户的角色
            List<UserRole> userRoles = userRoleRepository.findByIdAndStatus(user.getUserId(), LoginConstant.USER_ROLE_USED);
            //获取权限集合
            Collection<? extends GrantedAuthority> authorities = merge(userRoles);
            securityUser.setAuthorities(authorities);
            return securityUser;
        }
        return null;
    }
}

```

UserDetails这个也是个接口，其中定义了几种方法，都是围绕着**用户名、密码、权限+角色集合**这三个属性，因此我们可以实现这个类拓展这些字段，**SecurityUser**代码如下：

```

/*
 * @author 公众号：码猿技术专栏
 * 存储用户的详细信息，实现UserDetails，后续有定制的字段可以自己拓展
 */
@Data
public class SecurityUser implements UserDetails {
    //用户名
    private String username;

    //密码
    private String password;

    //权限+角色集合
    private Collection<? extends GrantedAuthority> authorities;

    public SecurityUser(String username, String password,Collection<? extends GrantedAuthority> authorities) {
        this.username = username;
        this.password = password;
        this.authorities = authorities;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() { return authorities; }

    @Override
    public String getPassword() { return password; }

    @Override
    public String getUsername() { return username; }

    // 账户是否未过期
    @Override
    public boolean isAccountNonExpired() { return true; }

    // 账户是否未被锁
    @Override
    public boolean isAccountNonLocked() { return true; }

    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @Override
    public boolean isEnabled() { return true; }
}

```

拓展：**UserDetailsService**这个类的实现一般涉及到5张表，分别是**用户表、角色表、权限表、用户<->角色对应关系表、角色<->权限对应关系表**，企业中的实现必须遵循**RBAC**设计规则。这个规则陈某后面会详细介绍。

Token校验过滤器

客户端请求头携带了token，服务端肯定是需要针对每次请求解析、校验token，因此必须定义一个Token过滤器，这个过滤器的主要逻辑如下：

- 从请求头中获取**accessToken**
- 对**accessToken**解析、验签、校验过期时间

- 校验成功，将authentication存入ThreadLocal中，这样方便后续直接获取用户详细信息。

上面只是最基础的一些逻辑，实际开发中还有特定的处理，比如将用户的详细信息放入Request属性中、Redis缓存中，这样能够实现feign的令牌中继效果。

校验过滤器的代码如下：

```
public class TokenAuthenticationFilter extends OncePerRequestFilter {
    /**
     * JWT的工具类
     */
    @Autowired
    private JwtUtils jwtUtils;

    /**
     * UserDetailsService的实现类，从数据库中加载用户详细信息
     */
    @Qualifier("jwtTokenUserDetailsService")
    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws ServletException, IOException {
        String token = request.getHeader(SecurityConstant.TOKEN_HEADER);
        /**
         * token存在则校验token
         * 1. token是否存在
         * 2. token存在：
         *   2.1 校验token中的用户名是否失效
         */
        if (!StringUtils.isEmpty(token)){
            String username = jwtUtils.getUsernameFromToken(token);
            //SecurityContextHolder.getContext().getAuthentication()==null 未认证则为true
            if (!StringUtils.isEmpty(username) && SecurityContextHolder.getContext().getAuthentication()==null){
                UserDetails userDetails = userDetailsService.loadUserByUsername(username);
                //如果token有效
                if (jwtUtils.validateToken(token,userDetails)){
                    // 将用户信息存入 authentication，方便后续校验
                    UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(userDetails, null,
                        userDetails.getAuthorities());
                    authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                    // 将 authentication 存入 ThreadLocal，方便后续获取用户信息
                    SecurityContextHolder.getContext().setAuthentication(authentication);
                }
            }
        }
        //继续执行下一个过滤器
        chain.doFilter(request,response);
    }
}
```

刷新令牌接口

accessToken一旦过期，客户端必须携带着**refreshToken**重新获取令牌，传统web服务是放在cookie中，只需要服务端完成刷新，完全做到无感知令牌续期，但是前后端分离架构中必须由客户端拿着**refreshToken**调接口手动刷新。

代码如下：

Watermark

```

@RestController
@RequestMapping
public class HomeController {

    @Autowired
    private JwtUtils jwtUtils;

    /**
     * 刷新令牌
     * @return
     */
    @PostMapping("/refreshToken")
    public ResultMsg refreshToken(HttpServletRequest request){
        //从请求头中获取refreshToken
        String oldRefreshToken = request.getHeader(SecurityConstant.REFRESH_TOKEN_HEADER);
        //校验refreshToken，如果令牌没有过期
        if (jwtUtils.isTokenExpired(oldRefreshToken)){
            return ResultMsg.builder().code(200).msg("刷新令牌已过期，请重新登录！").build();
        }

        //解析refreshToken
        String username = jwtUtils.getUsernameFromToken(oldRefreshToken);
        //生成新的accessToken
        String newAccessToken = jwtUtils.createToken(username);
        String newRefreshToken = jwtUtils.refreshToken(newAccessToken);
        return ResultMsg.builder().code(200).msg("令牌请求成功！").data(LoginToken.builder().accessToken(newAccessToken).refreshToken(newRefreshToken).build()).build();
    }
}

```

主要逻辑很简单，如下：

- 校验refreshToken
- 重新生成accessToken、refreshToken返回给客户端。

注意：实际生产中refreshToken令牌的生成方式、加密算法可以和accessToken不同。

登录认证过滤器接口配置

上述定义了一个认证过滤器JwtAuthenticationLoginFilter，这个是用来登录的过滤器，但是并没有注入加入Spring Security的过滤器链中，需要定义配置，代码如下：

```

/**
 * @author 公众号：码猿技术专栏
 * 登录过滤器的配置类
 */
@Configuration
public class JwtAuthenticationSecurityConfig extends
SecurityConfigurerAdapter<DefaultSecurityFilterChain, HttpSecurity> {

    /**
     * userDetailsService
     */
    @Qualifier("jwtTokenUserDetailsService")
    @Autowired
    private UserDetailsService userDetailsService;

    /**
     * 登录成功处理器
     */
    @Autowired
    private LoginAuthenticationSuccessHandler loginAuthenticationSuccessHandler;

    /**
     * 登录失败处理器
     */
    @Autowired
    private LoginAuthenticationFailureHandler loginAuthenticationFailureHandler;
}

```

```

/**
 * 加密
 */
@.Autowired
private PasswordEncoder passwordEncoder;

/**
 * 将登录接口的过滤器配置到过滤器链中
 * 1. 配置登录成功、失败处理器
 * 2. 配置自定义的userDetailsService（从数据库中获取用户数据）
 * 3. 将自定义的过滤器配置到spring security的过滤器链中，配置在
UsernamePasswordAuthenticationFilter之前
 * @param http
 */
@Override
public void configure(HttpSecurity http) {
    JwtAuthenticationLoginFilter filter = new JwtAuthenticationLoginFilter();
    filter.setAuthenticationManager(http.getSharedObject(AuthenticationManager.class));
    //认证成功处理器
    filter.setAuthenticationSuccessHandler(loginAuthenticationSuccessHandler);
    //认证失败处理器
    filter.setAuthenticationFailureHandler(loginAuthenticationFailureHandler);
    //直接使用DaoAuthenticationProvider
    DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
    //设置userDetailsService
    provider.setUserDetailsService(userDetailsService);
    //设置加密算法
    provider.setPasswordEncoder(passwordEncoder);
    http.authenticationProvider(provider);
    //将这个过滤器添加到UsernamePasswordAuthenticationFilter之前执行
    http.addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class);
}
}

```

所有的逻辑都在 `public void configure(HttpSecurity http)` 这个方法中，如下：

- 设置认证成功处理器**loginAuthenticationSuccessHandler**
- 设置认证失败处理器**loginAuthenticationFailureHandler**
- 设置userDetailsService的实现类**JwtTokenUserDetailsService**
- 设置加密算法**passwordEncoder**
- 将**JwtAuthenticationLoginFilter**这个过滤器加入到过滤器链中，直接加入到**UsernamePasswordAuthenticationFilter**这个过滤器之前。

Spring Security全局配置

上述仅仅配置了登录过滤器，还需要在全局配置类做一些配置，如下：

- 应用登录过滤器，配置
- 将登录接口、令牌刷新接口放行，不需要拦截
- 配置**AuthenticationEntryPoint**、**AccessDeniedHandler**
- 禁用session，前端分离+JWT方式不需要session

- 将token校验过滤器**TokenAuthenticationFilter**添加到过滤器链中，放在**UsernamePasswordAuthenticationFilter**之前。

完整配置如下：

```
/*
 * @author 公众号：码猿技术专栏
 * @EnableGlobalMethodSecurity 开启权限校验的注解
 */
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private JwtAuthenticationSecurityConfig jwtAuthenticationSecurityConfig;
    @Autowired
    private EntryPointUnauthorizedHandler entryPointUnauthorizedHandler;
    @Autowired
    private RequestAccessDeniedHandler requestAccessDeniedHandler;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.formLogin()
            //禁用表单登录，前后端分离用不上
            .disable()
            //应用登录过滤器的配置，配置分离
            .apply(jwtAuthenticationSecurityConfig)

            .and()
            // 设置URL的授权
            .authorizeRequests()
            // 这里需要将登录页面放行, permitAll() 表示不再拦截, /login 登录的
            url, /refreshToken刷新token的url
            //TODO 此处正常项目中放行的url还有很多，比如swagger相关的url, druid的后台url，一
            些静态资源
            .antMatchers("/login", "/refreshToken")
            .permitAll()
            //hasRole() 表示需要指定的角色才能访问资源
            .antMatchers("/hello").hasRole("ADMIN")
            // anyRequest() 所有请求    authenticated() 必须被认证
            .anyRequest()
            .authenticated()

            //处理异常情况：认证失败和权限不足
            .and()
            .exceptionHandling()
            //认证未通过，不允许访问异常处理器
            .authenticationEntryPoint(entryPointUnauthorizedHandler)
            //认证通过，但是没权限处理器
            .accessDeniedHandler(requestAccessDeniedHandler)

            .and()
            //禁用 session，JWT校验不需要session
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)

            .and()
            //将TOKEN校验过滤器配置到过滤器链中，否则不生效，放到
            UsernamePasswordAuthenticationFilter之前
    }
}
```

```

        .addFilterBefore(authenticationTokenFilterBean(),
UsernamePasswordAuthenticationFilter.class)
        // 关闭csrf
        .csrf().disable();
    }

// 自定义的Jwt Token校验过滤器
@Bean
public TokenAuthenticationFilter authenticationTokenFilterBean() {
    return new TokenAuthenticationFilter();
}

/**
 * 加密算法
 * @return
 */
@Bean
public PasswordEncoder getPasswordEncoder() {
    return new BCryptPasswordEncoder();
}
}

```

注释的很详细了，有不理解的认真看一下。

案例源码已经上传GitHub，关注公众号：码猿技术专栏，回复关键词：9529 获取！

测试

1、首先测试登录接口，postman访问<http://localhost:2001/security-jwt/login>，如下：

KEY	VALUE	DESCRIPTION	...	Bulk Edit
username	zhangsan			x
password	123456			

```

1
2     "code": 200,
3     "msg": "登录成功！",
4     "data": {
5         "accessToken": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ6aGFuZ3NhbiIsImlhCI6MTYzMzk4NzE2NiwiZXhwIjoxNjM4MDQ3MTY2fQ.4NiVsDd-7g-AdVQsQnzW1Q7h0UpVK-wzLgLDcNzPhrAUPL86dKNQ9WRjFEKEu2i9-B6LEq40bxz_5Jl1K1FZA",
6         "refreshToken": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ6aGFuZ3NhbiIsImlhCI6MTYzMzk4NzE2Nm3MiwiZXhwIjoxNjM4MTA3MTY3fQ.A0sg1Dk7XrsyHqZM2eK001PrHzbdAc8hJ0zwoCch0lbjy8_DTHFJGayaJdZDfyvGkMtdUNzGtAbbvFrnr1dAuQ"
7     }
8 }
```

可以看到，成功返回了两个token。

2、请求头不携带token，直接请求<http://localhost:2001/security-jwt/hello>，如下：

Watermark

可以看到，直接进入了**EntryPointUnauthorizedHandler**这个处理器。

3、携带token访问<http://localhost:2001/security-jwt/hello>，如下：

成功访问，token是有效的。

4、刷新令牌接口测试，携带一个过期的令牌访问如下：

5、刷新令牌接口测试，携带未过期的令牌测试，如下：

Watermark

```

1 {
2     "code": 200,
3     "msg": "令牌请求成功!",
4     "data": {
5         "accessToken": "eyJhbGciOiJIUzUxMiJ9eyJzdWIoJ1c2ViIwiaWF0IjoxNjM3ODA3NjM2LCJleHAiOiE2Mzc4Njc2MzZ9._Uh3tMhZB46Q1t68p7Cxw3cldwJCb6l19a80EF1321xkgJF0R8t9P5p52Fyceo0ggx4C1SmA4zv1706L9Q",
6         "refreshToken": "eyJhbGciOiJIUzUxMiJ9eyJzdWIoJ1c2ViIwiaWF0IjoxNjM3ODA3NjM2NDEyLC1leHAiOiE2Mzc8Mjc2MzZ9.vzW82VtveeGNjoZ4nR6dxBy-hCkm_4HISuc1jxSx79dInjM_dw@47QCMo4k1DyhNeHQ9XQ-R43616PhFw"
7     }
8 }

```

可以看到，成功返回了两个新的令牌。

源码追踪

以上一系列的配置完全是参照**UsernamePasswordAuthenticationFilter**这个过滤器，这个是web服务表单登录的方式。

Spring Security的原理就是一系列的过滤器组成，登录流程也是一样，起初在

`org.springframework.security.web.authentication.AbstractAuthenticationProcessingFilter#doFilter()`方法，进行认证匹配，如下：

```

    return;
}

if (logger.isDebugEnabled()) {
    logger.debug("Request is to process authentication");
}

Authentication authResult;

try { // 调用userNamePasswordAuthenticationFilter的认证方法
    authResult = attemptAuthentication(request, response);
    if (authResult == null) {
        // return immediately as subclass has indicated that it hasn't completed
        // authentication
        return;
    }
    sessionStrategy.onAuthentication(authResult, request, response);
}
catch (InternalAuthenticationServiceException failed) {
    logger.error("An internal error occurred while trying to authenticate the user.", failed);
    unsuccessfulAuthentication(request, response, failed); // 认证失败处理
}

return;
}
catch (AuthenticationException failed) {
    // Authentication failed
    unsuccessfulAuthentication(request, response, failed); // 认证失败处理
}

// Authentication success
if (continueChainBeforeSuccessfulAuthentication) {
    chain.doFilter(request, response);
}

successfulAuthentication(request, response, chain, authResult); // 认证成功处理
}

```

attemptAuthentication() 这个方法主要负责从客户端获取username、password，封装成 UsernamePasswordAuthenticationToken 交给 ProviderManager 的进行认证，源码如下：

```

public Authentication attemptAuthentication(HttpServletRequest request,
    HttpServletResponse response) throws AuthenticationException {
    if (!postOnly && !request.getMethod().equals("POST")) {
        throw new AuthenticationServiceException(
            "Authentication method not supported: " + request.getMethod());
    }

    String username = obtainUsername(request);
    String password = obtainPassword(request);          获取username、password并封装

    if (username == null) {
        username = "";
    }

    if (password == null) {
        password = "";
    }

    username = username.trim();

    UsernamePasswordAuthenticationToken authRequest = new UsernamePasswordAuthenticationToken(
        username, password);

    // Allow subclasses to set the "details" property
    setDetails(request, authRequest);

    return this.getAuthenticationManager().authenticate(authRequest); 交给ProviderManager进行认证
}

```

ProviderManager主要流程是调用抽象类 `AbstractUserDetailsAuthenticationProvider#authenticate()` 方法，如下图：

```

public Authentication authenticate(Authentication authentication) authentication: "org.springframework.security.authentication.UsernamePas-
    throws AuthenticationException {
    Assert.isInstanceOf(UsernamePasswordAuthenticationToken.class, authentication,
        () -> messages.getMessage( messages: MessageSourceAccessor@6401
            code: "AbstractUserDetailsAuthenticationProvider.onlySupports",
            defaultMessage: "Only UsernamePasswordAuthenticationToken is supported"));

    // Determine username
    String username = (authentication.getPrincipal() == null) ? "NONE_PROVIDED" username: "user"
        : authentication.getName(); authentication: "org.springframework.security.authentication.UsernamePasswordAuthenticationToken@2_.

    boolean cacheWasUsed = true; cacheWasUsed: false
    UserDetails user = this.userCache.getUserFromCache(username); user: null userCache: NullUserCache@6402

    if (user == null) {
        cacheWasUsed = false; cacheWasUsed: false
        try {
            user = retrieveUser(username, user: null username: "user"          调用userDetailsService获取用户信息
                (UsernamePasswordAuthenticationToken) authentication);
        }
        catch (UsernameNotFoundException notFound) {
            logger.debug( o: "User " + username + " not found");
            if (hideUserNotFoundExceptions) {
                throw new BadCredentialsException(messages.getMessage(
                    code: "AbstractUserDetailsAuthenticationProvider.badCredentials",
                    defaultMessage: "Bad credentials"));
            }
            else {
                throw notFound;
            }
        }
        Assert.notNull(user,
            message: "retrieveUser returned null - a violation of the interface contract");
    }
}

```

`retrieveUser()` 方法就是调用userDetailsService查询用户信息。然后认证，一旦认证成功或者失败，则会调用对应的失败、成功处理器进行处理。

总结 Watermark

Spring Security虽然比较重，但是真的好用，尤其是实现Oauth2.0规范，非常简单方便。

案例源码已经上传GitHub，关注公众号：码猿技术专栏，回复关键词：9529 获取！

SpringBoot+MDC实现全链路调用日志跟踪~

大家好，我是不才陈某~

前面有一篇文章简单的介绍过MDC，这次结合具体的案例、生产中的具体问题深入了解一下MDC。

MDC介绍

1、简介：

MDC (Mapped Diagnostic Context, 映射调试上下文) 是 log4j、logback 及 log4j2 提供的一种方便在多线程条件下记录日志的功能。MDC 可以看成是一个与当前线程绑定的哈希表，可以往其中添加键值对。MDC 中包含的内容可以被同一线程中执行的代码所访问。

当前线程的子线程会继承其父线程中的 MDC 的内容。当需要记录日志时，只需要从 MDC 中获取所需的信息即可。MDC 的内容则由程序在适当的时候保存进去。对于一个 Web 应用来说，通常是在请求被处理的最开始保存这些数据

2、API说明：

- `clear()` : 移除所有MDC
- `get(String key)` : 获取当前线程MDC中指定key的值
- `getContext()` : 获取当前线程MDC的MDC
- `put(String key, Object o)` : 往当前线程的MDC中存入指定的键值对
- `remove(String key)` : 删除当前线程MDC中指定的键值对

3、优点：

代码简洁，日志风格统一，不需要在 log 打印中手动拼写 `traceId`，即 `LOGGER.info("traceId: {} ", traceId)`

MDC使用

1、添加拦截器

```
public class LogInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {
        //如果有上层调用就用上层的ID
        String traceId = request.getHeader(Constants.TRACE_ID);
        if (traceId == null) {
            traceId = TraceIdUtil.getTraceId();
        }
        MDC.put(Constants.TRACE_ID, traceId);
        return true;
    }

    @Override
```

```

    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
    ModelAndView modelAndView)
        throws Exception {
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object
    handler, Exception ex)
        throws Exception {
        //调用结束后删除
        MDC.remove(Constants.TRACE_ID);
    }
}

```

2、修改日志格式

```
<property name="pattern">[TRACEID:%X{traceId}] %d{HH:mm:ss.SSS} %-5level %class{-1}.%M()/%L -
%msg%xEx%n</property>
```

重点是 `%X{traceId}`，`traceId`和MDC中的键名称一致

简单使用就这么容易，但是在有些情况下`traceId`将获取不到

MDC 存在的问题

- 子线程中打印日志丢失`traceId`
- HTTP调用丢失`traceId`

丢失`traceId`的情况，来一个再解决一个，绝不提前优化

解决MDC存在的问题

子线程日志打印丢失`traceId`

子线程在打印日志的过程中`traceId`将丢失，解决方式为重写线程池，对于直接new创建线程的情况不考虑【实际应用中应该避免这种用法】，重写线程池无非是对任务进行一次封装

线程池封装类：`ThreadPoolExecutorMdcWrapper.java`

```

public class ThreadPoolExecutorMdcWrapper extends ThreadPoolExecutor {
    public ThreadPoolExecutorMdcWrapper(int corePoolSize, int maximumPoolSize, long keepAliveTime,
    TimeUnit unit,
                                         BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
    }

    public ThreadPoolExecutorMdcWrapper(int corePoolSize, int maximumPoolSize, long keepAliveTime,
    TimeUnit unit,
                                         ThreadFactory threadFactory) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue, threadFactory);
    }
}

```

```

        public ThreadPoolExecutorMdcWrapper(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit,
                                         BlockingQueue<Runnable> workQueue,
RejectedExecutionHandler handler) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue, handler);
}

    public ThreadPoolExecutorMdcWrapper(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit,
                                         BlockingQueue<Runnable> workQueue,
ThreadFactory threadFactory,
                                         RejectedExecutionHandler handler) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue, threadFactory,
handler);
}

@Override
public void execute(Runnable task) {
    super.execute(ThreadMdcUtil.wrap(task, MDC.getCopyOfContextMap()));
}

@Override
public <T> Future<T> submit(Runnable task, T result) {
    return super.submit(ThreadMdcUtil.wrap(task, MDC.getCopyOfContextMap()), result);
}

@Override
public <T> Future<T> submit(Callable<T> task) {
    return super.submit(ThreadMdcUtil.wrap(task, MDC.getCopyOfContextMap()));
}

@Override
public Future<?> submit(Runnable task) {
    return super.submit(ThreadMdcUtil.wrap(task, MDC.getCopyOfContextMap()));
}
}

```

说明：

- 继承**ThreadPoolExecutor**类，重新执行任务的方法
- 通过**ThreadMdcUtil**对任务进行一次包装

线程traceId封装工具类： ThreadMdcUtil.java

```

public class ThreadMdcUtil {
    public static void setTraceIdIfAbsent() {
        if (MDC.get(Constants.TRACE_ID) == null) {
            MDC.put(Constants.TRACE_ID, TraceIdUtil.getTraceId());
        }
    }

    public static <T> Callable<T> wrap(final Callable<T> callable, final Map<String, String> context)
    {
        return () ->
        {
            if (context == null) {
                MDC.clear();
            } else {
                MDC.setContextMap(context);
            }
        };
    }
}

```

```

        }
        setTraceIdIfAbsent();
        try {
            return callable.call();
        } finally {
            MDC.clear();
        }
    };
}

public static Runnable wrap(final Runnable runnable, final Map<String, String> context) {
    return () -> {
        if (context == null) {
            MDC.clear();
        } else {
            MDC.setContextMap(context);
        }
        setTraceIdIfAbsent();
        try {
            runnable.run();
        } finally {
            MDC.clear();
        }
    };
}
}

```

说明【以封装Runnable为例】：

- 判断当前线程对应MDC的Map是否存在，存在则设置
- 设置MDC中的traceId值，不存在则新生成，针对不是子线程的情况，如果是子线程，MDC中traceId不为null
- 执行run方法

代码等同于以下写法，会更直观

```

public static Runnable wrap(final Runnable runnable, final Map<String, String> context) {
    return new Runnable() {
        @Override
        public void run() {
            if (context == null) {
                MDC.clear();
            } else {
                MDC.setContextMap(context);
            }
            setTraceIdIfAbsent();
            try {
                runnable.run();
            } finally {
                MDC.clear();
            }
        }
    };
}

```

Watermark

重新返回的是包装后的Runnable，在该任务执行之前【`runnable.run()`】先将主线程的Map设置到当前线程中【即`MDC.setContextMap(context)`】，这样子线程和主线程MDC对应的Map就是一样的了

- 判断当前线程对应MDC的Map是否存在，存在则设置
- 设置MDC中的traceId值，不存在则新生成，针对不是子线程的情况，如果是子线程， MDC中traceId不为null
- 执行run方法

HTTP调用丢失traceId

在使用HTTP调用第三方服务接口时traceId将丢失，需要对HTTP调用工具进行改造，在发送时在request header中添加traceId，在下层被调用方添加拦截器获取header中的traceId添加到MDC中

HTTP调用有多种方式，比较常见的有HttpClient、OKHttp、RestTemplate，所以只给出这几种HTTP调用的解决方式

1、HttpClient：

实现HttpClient拦截器：

```
public class HttpClientTraceIdInterceptor implements HttpRequestInterceptor {
    @Override
    public void process(HttpRequest httpRequest, HttpContext httpContext) throws HttpException,
    IOException {
        String traceId = MDC.get(Constants.TRACE_ID);
        //当前线程调用中有traceId，则将该traceId进行透传
        if (traceId != null) {
            //添加请求体
            httpRequest.addHeader(Constants.TRACE_ID, traceId);
        }
    }
}
```

实现HttpRequestInterceptor接口并重写process方法

如果调用线程中含有traceId，则需要将获取到的traceId通过request中的header向下透传下去

为HttpClient添加拦截器：

```
private static CloseableHttpClient httpClient = HttpClientBuilder.create()
    .addInterceptorFirst(new HttpClientTraceIdInterceptor())
    .build();
```

通过addInterceptorFirst方法为HttpClient添加拦截器

2、OKHttp：

实现OKHttp拦截器：

```
public class OkHttpTraceIdInterceptor implements Interceptor {
    @Override
    public Response intercept(Chain chain) throws IOException {
        String traceId = MDC.get(Constants.TRACE_ID);
        Request request = null;
        if (traceId != null) {
            //添加请求体
            Request newRequest = chain.request().newBuilder().addHeader(Constants.TRACE_ID, traceId).build();
            Response originResponse = chain.proceed(newRequest);
            return originResponse;
        }
    }
}
```

```
}
```

实现**Interceptor**拦截器，重写interceptor方法，实现逻辑和HttpClient差不多，如果能够获取到当前线程的traceId则向下透传

为OkHttp添加拦截器：

```
private static OkHttpClient client = new OkHttpClient.Builder()
    .addNetworkInterceptor(new OkHttpTraceIdInterceptor())
    .build();
```

调用**addNetworkInterceptor**方法添加拦截器

3、RestTemplate：

实现RestTemplate拦截器：

```
public class RestTemplateTraceIdInterceptor implements ClientHttpRequestInterceptor {
    @Override
    public ClientHttpResponse intercept(HttpRequest httpRequest, byte[] bytes,
    ClientHttpRequestExecution clientHttpRequestExecution) throws IOException {
        String traceId = MDC.get(Constants.TRACE_ID);
        if (traceId != null) {
            httpRequest.getHeaders().add(Constants.TRACE_ID, traceId);
        }

        return clientHttpRequestExecution.execute(httpRequest, bytes);
    }
}
```

实现**ClientHttpRequestInterceptor**接口，并重写intercept方法，其余逻辑都是一样的不重复说明

为RestTemplate添加拦截器：

```
restTemplate.setInterceptors(Arrays.asList(new RestTemplateTraceIdInterceptor()));
```

调用**setInterceptors**方法添加拦截器

4、第三方服务拦截器：

HTTP调用第三方服务接口全流程traceId需要第三方服务配合，第三方服务需要添加拦截器拿到request header中的traceId并添加到MDC中

```
public class LogInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
    throws Exception {
        //如果有上层调用就用上层的ID
        String traceId = request.getHeader(Constants.TRACE_ID);
        if (traceId == null) {
            traceId = TraceIdUtils.getTraceId();
        }

        MDC.put("traceId", traceId);
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
    ModelAndView modelAndView)
```

```

        throws Exception {
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object
handler, Exception ex)
        throws Exception {
        MDC.remove(Constants.TRACE_ID);
    }
}

```

说明：

- 先从request header中获取traceId
- 从request header中获取不到traceId则说明不是第三方调用，直接生成一个新的traceId
- 将生成的traceId存入MDC中

除了需要添加拦截器之外，还需要在日志格式中添加traceId的打印，如下：

```

<property name="pattern">[TRACEID:%X{traceId}] %d{HH:mm:ss.SSS} %-5level %class{-1}.%M()/%L -
%msg%xEx%n</property>

```

注意：需要添加 `%X{traceId}`

Spring Boot 2.6.0 发布！一大波新特性，禁止了循环依赖，还有哪些更新？

大家好，我是不才陈某~

本来预计12月份中旬发布的**Spring Boot 2.6.0** 版本硬是提前了一个月，当然陈某也是时刻关注 Spring Boot 的每一个的版本迭代，今天给大家分享一下**Spring Boot 2.6.0** 更新了什么？

1、默认禁止了循环依赖

循环依赖大家都知道，也被折磨过，这下**2.6.0**的版本默认禁止了循环依赖，如果程序中出现循环依赖就会报错。

```

*****
APPLICATION FAILED TO START
*****

Description:

The dependencies of some of the beans in the application context form a cycle:

    [ ] logService (field cn.javastack.springboot.features.core.UserService cn.javastack.springboot.features.core.LogService.userService)
    |
    +-- userService (field cn.javastack.springboot.features.core.LogService cn.javastack.springboot.features.core.UserService.logService)

Action:

Relying upon circular references is discouraged and they are prohibited by default. Update your application to remove the dependency
last resort, it may be possible to break the cycle automatically by setting spring.main.allow-circular-references to true.

```

当然并没有一锤子打死，也提供了开启允许循环依赖的配置，只需要在配置文件中开启即可：

```
spring:  
  main:  
    allow-circular-references: true
```

2、支持自定义脱敏规则

Spring Boot 现在可以清理 `/env` 和 `/configprops` 端点中存在的敏感值。

自定义 `SanitizingFunction` 类型的 Bean 即可实现。

```
@Bean  
public SanitizingFunction mobileSanitizingFunction() {  
    return data -> {  
        PropertySource<?> propertySource = data.getPropertySource();  
        if (propertySource.getName().contains("redis.properties")) {  
            if (data.getKey().equals("redis.mobile")) {  
                return data.withValue(SANITIZED_VALUE);  
            }  
        }  
        return data;  
    };  
}
```

关于脱敏陈某之前写过一篇文章，已经非常详细了：[Springboot 日志、配置文件、接口数据如何脱敏？老鸟们都是这样玩的！](#)

3、Redis自动开启连接池

这个版本之前Redis连接池需要开发主动开启，但是这个版本默认是开启的。

如果需要关闭一样是提供了配置，如下：

1、jedis连接池关闭：

```
spring.redis.jedis.pool.enabled = false
```

2、lettuce连接池关闭：

```
spring.redis.lettuce.pool.enabled = false
```

4、响应式应用服务器会话属性

响应式应用服务器支持的会话属性已在此版本中扩展。

以前是在 `spring.webflux.session` 下，现在在 `server.reactive.session` 下，并且提供与 servlet 版本相同的属性。

Watermark

5、Maven构建信息属性排除

现在可以从 Spring Boot Maven 或 Gradle 插件生成的 build-info.properties 文件中排除特定属性。

比如，排除 Maven 的 version 属性：

```
<configuration>
    <excludeInfoProperties>
        <excludeInfoProperty>version</excludeInfoProperty>
    </excludeInfoProperties>
</configuration>
```

6、支持使用WebTestClient来测试Spring MVC

开发人员可以使用 **WebTestClient** 在模拟环境中测试程序，只需要在Mock环境中使用 **@AutoConfigureMockMvc**注释，就可以轻松注入 **WebTestClient**。省去编写测试程序。

7、支持 Log4j2 复合配置

现在支持 Log4j2 的复合配置，可以通过 **logging.log4j2.config.override** 参数来指定覆盖主日志配置文件的其他日志配置文件。

总结

以上陈某只是总结了比较重要的几点，这个版本变动还是有些大的，具体细节可以看官方文档：<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.6-Release-Notes>

Spring Boot 整合 阿里开源中间件 Canal 实现数据增量同步！

大家好，我是不才陈某~

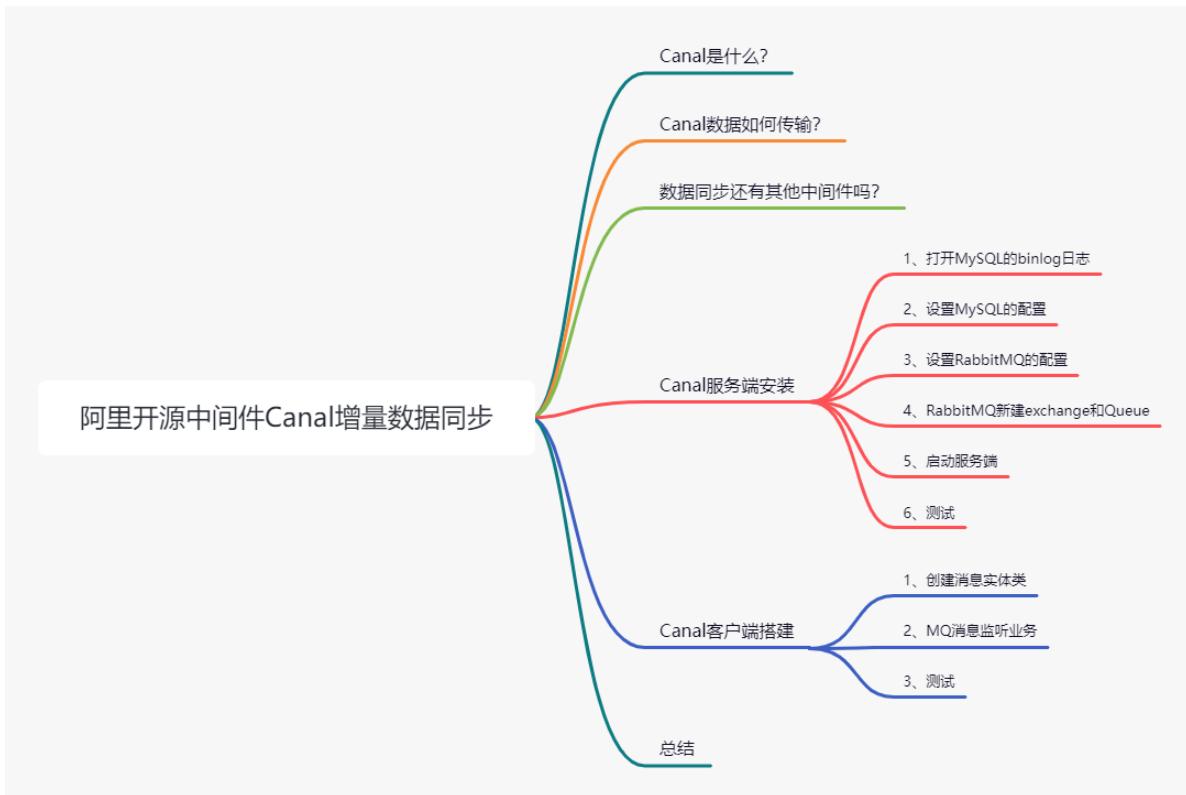
数据同步一直是一个令人头疼的问题。在业务量小，场景不多，数据量不大的情况下我们可能会选择在项目中直接写一些定时任务手动处理数据，例如从多个表将数据查出来，再汇总处理，再插入到相应的地方。

但是随着业务量增大，数据量变多以及各种复杂场景下的分库分表的实现，使数据同步变得越来越困难。

今天这篇文章使用**阿里**开源的中间件**Canal**解决数据增量同步的痛点。

文章目录如下：

Watermark



Canal是什么？

canal译意为水道/管道/沟渠，主要用途是基于 **MySQL** 数据库**增量日志**解析，提供增量数据订阅和消费。

从这句话理解到了什么？

基于MySQL，并且通过MySQL日志进行的增量解析，这也就意味着对原有的业务代码完全是无侵入性的。

工作原理：解析MySQL的binlog日志，提供增量数据。

基于日志增量订阅和消费的业务包括

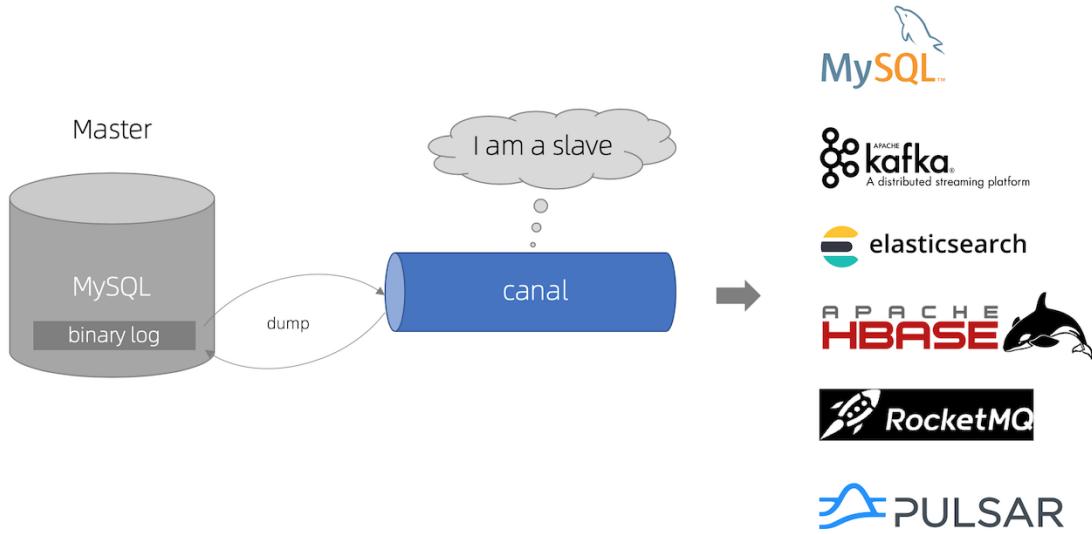
- 数据库镜像
- 数据库实时备份
- 索引构建和实时维护(拆分异构索引、倒排索引等)
- 业务 cache 刷新
- 带业务逻辑的增量数据处理

当前的 canal 支持源端 MySQL 版本包括 5.1.x , 5.5.x , 5.6.x , 5.7.x , 8.0.x。

官方文档：<https://github.com/alibaba/canal>

Canal数据如何传输？

先来一张官方图：



Canal分为服务端和客户端，这也是阿里常用的套路，比如前面讲到的注册中心**Nacos**：

- **服务端**：负责解析MySQL的binlog日志，传递增量数据给客户端或者消息中间件
- **客户端**：负责解析服务端传过来的数据，然后定制自己的业务处理。

目前为止支持的消息中间件很全面了，比如**Kafka**、**RocketMQ**、**RabbitMQ**。

数据同步还有其他中间件吗？

有，当然有，还有一些开源的中间件也是相当不错的，比如**Bifrost**。

常见的几款中间件的区别如下：

	Canal	Debezium	DataX	Databus	Flinkx	Bifrost
实时同步	支持	支持	不支持	支持	支持	支持
增量同步	支持	支持	不支持	支持	支持	支持
写业务逻辑	自己写保存 变更数据的 代码	自己写保存变更 数据的代码	不用写	自己写保存变 更数据的代码	自己写保存变 更数据的代码	不用写
支持 MySQL	支持	支持	支持	支持	支持	支持
活跃度	高	高	高	不高	一般	可以

当然要我选择的话，首选阿里的中间件Canal。

Canal服务端安装

服务端需要下载压缩包，下载地址：<https://github.com/alibaba/canal/releases>

目前最新的是**v1.1.5**，点击下载：

Assets 6			
canal.adapter-1.1.5.tar.gz		188 MB	
canal.admin-1.1.5.tar.gz		36.6 MB	
canal.deployer-1.1.5.tar.gz		57.4 MB	
canal.example-1.1.5.tar.gz		22.1 MB	
Source code (zip)			
Source code (tar.gz)			

下载完成解压，目录如下：

名称	修改日期	类型	大小
bin 启动脚本	2021/12/24 14:29	文件夹	
conf 配置文件	2021/12/24 14:29	文件夹	
lib 依赖包	2021/12/24 14:29	文件夹	
logs 日志	2021/12/24 14:39	文件夹	
plugin 插件	2021/12/24 14:29	文件夹	

本文使用**Canal + RabbitMQ**进行数据的同步，因此下面步骤完全按照这个base进行。

1、打开MySQL的binlog日志

修改MySQL的日志文件，my.cnf 配置如下：

```
[mysqld]
log-bin=mysql-bin # 开启 binlog
binlog-format=ROW # 选择 ROW 模式
server_id=1 # 配置 MySQL replaction 需要定义，不要和 canal 的 slaveId 重复
```

2、设置MySQL的配置

需要设置服务端配置文件中的MySQL配置，这样Canal才能知道需要监听哪个库、哪个表的日志文件。

一个 Server 可以配置多个实例监听，Canal 功能默认自带的有个 example 实例，本篇就用 example 实例。如果增加实例，复制 example 文件夹内容到同级目录下，然后在 `canal.properties` 指定添加实例的名称。

修改canal.deployer-1.1.5\conf\example\instance.properties配置文件

```
# url
canal.instance.master.address=127.0.0.1:3306
# username/password
canal.instance.dbUsername=root
canal.instance.dbPassword=root
# 监听的数据库
canal.instance.defaultDatabaseName=test

# 监听的表，可以指定，多个用逗号分割，这里正则是监听所有
canal.instance.filter.regex=.*\..*
```

Watermark

3、设置RabbitMQ的配置

服务端默认的传输方式是tcp，需要在配置文件中设置MQ的相关信息。

这里需要修改两处配置文件，如下：

1、canal.deployer-1.1.5\conf\canal.properties

这个配置文件主要是设置MQ相关的配置，比如URL，用户名、密码...

```
# 传输方式: tcp, kafka, rocketMQ, rabbitMQ
canal.serverMode = rabbitMQ
#####
# RabbitMQ #####
#####

rabbitmq.host = 127.0.0.1
rabbitmq.virtual.host =
# exchange
rabbitmq.exchange =canal.exchange
# 用户名、密码
rabbitmq.username =guest
rabbitmq.password =guest
## 是否持久化
rabbitmq.deliveryMode = 2
```

2、canal.deployer-1.1.5\conf\example\instance.properties

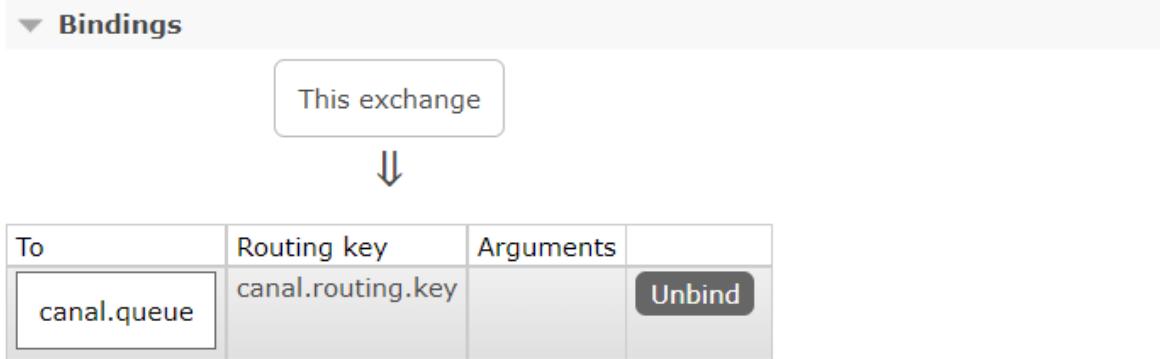
这个文件设置MQ的路由KEY，这样才能路由到指定的队列中，如下：

```
canal.mq.topic=canal.routing.key
```

4、RabbitMQ新建exchange和Queue

在RabbitMQ中需要新建一个**canal.exchange**（必须和配置中的相同）的exchange和一个名称为**canal.queue**（名称随意）的队列。

其中绑定的路由KEY为：**canal.routing.key**（必须和配置中的相同），如下图：



5、启动服务端

点击bin目录下的脚本，windows直接双击**startup.bat**，启动成功如下：

```

C:\WINDOWS\system32\cmd.exe
start cmd : java -Xms128m -Xmx512m -XX:PermSize=128m -Djava.awt.headless=true -Djava.net.preferIPv4Stack=true -Dapplication.codeset=UTF-8 -Dfile.encoding=UTF-8 -server -Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:transport=dt_socket,address=9099,server=y,suspend=n -DappName=otter-canal -Dlogback.configurationFile="E:\canal.deployer-1.1.5\bin\\..\conf\\logback.xml" -Dcanal.conf="E:\canal.deployer-1.1.5\bin\\..\conf\\canal.properties" -classpath "E:\canal.deployer-1.1.5\bin\\..\conf\\*.jar" java -Xms128m -Xmx512m -XX:PermSize=128m -Djava.awt.headless=true -Djava.net.preferIPv4Stack=true -Dapplication.codeset=UTF-8 -Dfile.encoding=UTF-8 -server -Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:transport=dt_socket,address=9099,server=y,suspend=n -DappName=otter-canal -Dlogback.configurationFile="E:\canal.deployer-1.1.5\bin\\..\conf\\logback.xml" -Dcanal.conf="E:\canal.deployer-1.1.5\bin\\..\conf\\canal.properties" -classpath "E:\canal.deployer-1.1.5\bin\\..\conf\\*.jar" com.alibaba.otter.canal.deployer.CanalLauncher
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=128m; support was removed in 8.0
Listening for transport dt_socket at address: 9099

```

6、测试

在本地数据库**test**中的**oauth_client_details**插入一条数据，如下：

```

INSERT INTO `oauth_client_details` VALUES ('myjsz1', 'res1',
'$2a$10$F1tQdeb0SEMdtj108X/0w06Gqybu6vPC/Xg80mP9/TL1i4beXdK9W', 'all',
'password,refresh_token,authorization_code,client_credentials,implicit', 'http://www.baidu.com', NULL,
1000, 1000, NULL, 'false');

```

此时查看MQ中的**canal.queue**已经有了数据，如下：

The server reported 0 messages remaining.	
Exchange	canal.exchange
Routing Key	canal.routing.key
Redelivered	0
Properties	
Payload	{"data": [{"client_id": "myjsz1", "resource_ids": "res1", "client_secret": "\$2a\$10\$F1tQdeb0SEMdtj108X/0w06Gqybu6vPC/Xg80mP9/TL1i4beXdK9W", "scope": "all", "authorized_grant_types": "password,refresh_token,authorization_code,client_credentials,implicit", "web_server_redirect_uri": "http://www.baidu.com", "authorities": null, "access_token_validity": "1000", "refresh_token_validity": "1000", "additional_information": null, "autoapprove": "false"}]}
Encoding:	string

其实是一串JSON数据，这个JSON如下：

```

{
  "data": [
    {
      "client_id": "myjsz1",
      "resource_ids": "res1",
      "client_secret": "$2a$10$F1tQdeb0SEMdtj108X/0w06Gqybu6vPC/Xg80mP9/TL1i4beXdK9W",
      "scope": "all",
      "authorized_grant_types": "password,refresh_token,authorization_code,client_credentials,implicit",
      "web_server_redirect_uri": "http://www.baidu.com",
      "authorities": null,
      "access_token_validity": "1000",
      "refresh_token_validity": "1000",
      "additional_information": null,
      "autoapprove": "false"
    }
  ],
  "database": "test",
  "es": 1640337532000,
  "id": 7,
  "isDdl": false,
  "mysqlType": {
    "client_id": "varchar(48)",
    "resource_ids": "varchar(256)",
    "client_secret": "varchar(256)",
    "scope": "varchar(256)",
    "authorized_grant_types": "varchar(256)",
    "web_server_redirect_uri": "varchar(256)",
    "authorities": "varchar(256)"
  }
}

```

```

    "access_token_validity": "int(11)",
    "refresh_token_validity": "int(11)",
    "additional_information": "varchar(4096)",
    "autoapprove": "varchar(256)"
},
"old": null,
"pkNames": ["client_id"],
"sql": "",
"sqlType": {
    "client_id": 12,
    "resource_ids": 12,
    "client_secret": 12,
    "scope": 12,
    "authorized_grant_types": 12,
    "web_server_redirect_uri": 12,
    "authorities": 12,
    "access_token_validity": 4,
    "refresh_token_validity": 4,
    "additional_information": 12,
    "autoapprove": 12
},
"table": "oauth_client_details",
"ts": 1640337532520,
"type": "INSERT"
}

```

每个字段的意思已经很清楚了，有表名称、方法、参数、参数类型、参数值.....

客户端要做的就是监听MQ获取JSON数据，然后将其解析出来，处理自己的业务逻辑。

Canal客户端搭建

客户端很简单实现，要做的就是消费Canal服务端传递过来的消息，监听**canal.queue**这个队列。

1、创建消息实体类

MQ传递过来的是JSON数据，当然要创建个实体类接收数据，如下：

```

/**
 * @author 公众号 码猿技术专栏
 * Canal消息接收实体类
 */
@Data
@NoArgsConstructor
public class CanalMessage<T> {
    @JsonProperty("type")
    private String type;

    @JsonProperty("table")
    private String table;
    @JsonProperty("data")
    private List<T> data;

    @JsonProperty("database")
    private String database;
}

```

```

    @JsonProperty("es")
    private Long es;

    @JsonProperty("id")
    private Integer id;

    @JsonProperty("isDdl")
    private Boolean isDdl;

    @JsonProperty("old")
    private List<T> old;

    @JsonProperty("pkNames")
    private List<String> pkNames;

    @JsonProperty("sql")
    private String sql;

    @JsonProperty("ts")
    private Long ts;
}

```

2、MQ消息监听业务

接下来就是监听队列，一旦有Canal服务端有数据推送能够及时的消费。

代码很简单，只是给出个接收的案例，具体的业务逻辑可以根据业务实现，如下：

```

import cn.hutool.json.JSONUtil;
import cn.myjszl.middle.ware.canal.mq.rabbit.model.CanalMessage;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.rabbit.annotation.Exchange;
import org.springframework.amqp.rabbit.annotation.Queue;
import org.springframework.amqp.rabbit.annotation.QueueBinding;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

/**
 * 监听MQ获取Canal增量的数据消息
 */
@Component
@Slf4j
@RequiredArgsConstructor
public class CanalRabbitMQListener {

    @RabbitListener(bindings = {
        @QueueBinding(
            value = @Queue(value = "canal.queue", durable = "true"),
            exchange = @Exchange(value = "canal.exchange"),
            key = "canal.routing.key"
        )
    })
    public void handleDataChange(String message) {
        //将message转换为CanalMessage
        CanalMessage canalMessage = JSONUtil.toBean(message, CanalMessage.class);
        String tableName = canalMessage.getTable();
    }
}

```

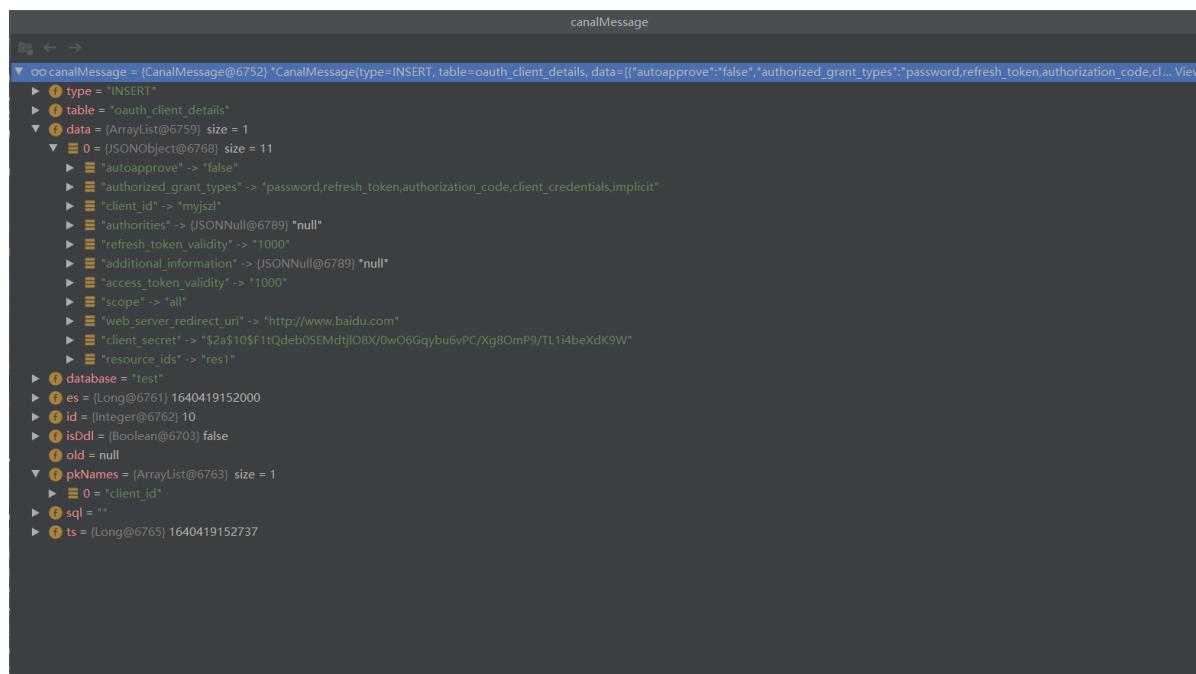
```
        log.info("Canal 监听 {} 发生变化; 明细: {}", tableName, message);
        //TODO 业务逻辑自己完善.....
    }
}
```

3、测试

下面向表中插入数据，看下接收的消息是什么样的，SQL如下：

```
INSERT INTO `oauth_client_details`
VALUES
( 'myjsz1', 'res1', '$2a$10$F1tQdeb0SEMdtj108X/0wO6Gqybu6vPC/Xg80mP9/TL1i4beXdK9W', 'all',
'password,refresh_token,authorization_code,client_credentials,implicit', 'http://www.baidu.com', NULL,
1000, 1000, NULL, 'false' );
```

客户端转换后的消息如下图：



上图可以看出所有的数据都已经成功接收到，只需要根据数据完善自己的业务逻辑即可。

客户端案例源码已经上传GitHub，关注公众号：**码猿技术专栏**，回复关键词：**9530** 获取！

总结

数据增量同步的开源工具并不只有Canal一种，根据自己的业务需要选择合适的组件。

Spring Boot 如何解决多个定时任务阻塞问题?

大家好，我是不才陈某~

最近公司面试新人，无意中听到这样一道面试题：Spring Boot 定时任务如何开启多线程？

本来一道简单的面试题，却有不少人不了解。

今天这篇文章就来介绍一下Spring Boot 中 是如何开启多线程定时任务。

为什么Spring Boot 定时任务是单线程的？

想要解释为什么，一定要从源码入手，直接从 `@EnableScheduling` 这个注解入手，找到了这个 `ScheduledTaskRegistrar` 类，其中有一段代码如下：

```
protected void scheduleTasks() {
    if (this.taskScheduler == null) {
        this.localExecutor = Executors.newSingleThreadScheduledExecutor();
        this.taskScheduler = new ConcurrentTaskScheduler(this.localExecutor);
    }
}
```

如果 `taskScheduler` 为 `null`，则创建单线程的线程池：
`Executors.newSingleThreadScheduledExecutor()`。

多线程定时任务如何配置？

下面介绍三种方案配置多线程下的定时任务。

1、重写 `SchedulingConfigurer#configureTasks()`

直接实现 `SchedulingConfigurer` 这个接口，设置 `taskScheduler`，代码如下：

```
@Configuration
public class ScheduleConfig implements SchedulingConfigurer {
    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        //设定一个长度10的定时任务线程池
        taskRegistrar.setScheduler(Executors.newScheduledThreadPool(10));
    }
}
```

2、通过配置开启

Spring Boot quartz 已经提供了一个配置用来配置线程池的大小，如下；

```
spring.task.scheduling.pool.size=10
```

只需要在配置文件中添加如上的配置即可生效！

Watermark

3、结合@Async

@Async这个注解都用过，用来开启异步任务的，使用@Async这个注解之前一定是要先配置线程池的，配置如下：

```
@Bean
public ThreadPoolTaskExecutor taskExecutor() {
    ThreadPoolTaskExecutor poolExecutor = new ThreadPoolTaskExecutor();
    poolExecutor.setCorePoolSize(4);
    poolExecutor.setMaxPoolSize(6);
    // 设置线程活跃时间（秒）
    poolExecutor.setKeepAliveSeconds(120);
    // 设置队列容量
    poolExecutor.setQueueCapacity(40);
    poolExecutor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
    // 等待所有任务结束后再关闭线程池
    poolExecutor.setWaitForTasksToCompleteOnShutdown(true);
    return poolExecutor;
}
```

然后在@Scheduled方法上标注@Async这个注解即可实现多线程定时任务，代码如下：

```
@Async
@Scheduled(cron = "0/2 * * * * ?")
public void test2() {
    System.out.println(".....执行test2.....");
}
```

总结

本篇文章介绍了Spring Boot 中 实现多线程定时任务的三种方案，你喜欢哪一种？

Spring Boot启动时如何对配置文件进行校验？

大家好，我是不才陈某~

在项目开发过程中，某个功能需要依赖在配置文件中配置的参数。这时候就可能出现下面这种现象：

有时候经常出现项目启动了，等到使用某个功能组件的时候出现异常，提示参数未配置或者bean注入失败。

那有没有一种方法在项目启动时就对参数进行校验而不是在实际使用的时候再抛出提示呢？

当然可以 答案就是使用Spring提供的Java Validation功能，简单实用。

Validation不是做参数校验的吗？居然还可以干这种事！

开启启动时参数校验

只需要在我们创建的配置Properties类增加Validation相关配置即可

```
@Validated  
@Data  
@ConfigurationProperties(prefix = "app")  
@Component  
public class AppConfigProperties {  
    @NotEmpty(message = "配置文件配置必须要配置[app.id]属性")  
    private String id;  
}
```

上面的配置就会校验我们在 `application.yml` 中有没有配置 `app.id` 参数。如果在配置文件中没有该配置，项目启动就会失败，并抛出校验异常。

在使用配置文件校验时，必须使用 `@configurationproperties` 注解，`@value` 注解不支持该功能。

在需要使用app.id的时候注入配置类即可：

```
@Autowired  
private AppConfigProperties appConfigProperties;
```

这样就可以实现我们想要的效果，如下图：

```
*****  
APPLICATION FAILED TO START  
*****  
  
Description:  
  
Binding to target org.springframework.boot.context.properties.bind.BindException: Failed to bind properties under 'app' to org.jeecg.validate.AppConfigProperties failed:  
  
    Property: app.id  
    Value: null  
    Reason: [app.id]属性必须要在配置文件配置  
  
Action:  
  
Update your application's configuration  
  
Disconnected from the target VM, address: 'javadebug', transport: 'shared memory'  
  
Process finished with exit code 1
```

支持校验类型

Watermark

校验规则	规则说明
@Null	限制只能为null
@NotNull	限制必须不为null
@AssertFalse	限制必须为false
@AssertTrue	限制必须为true
@DecimalMax(value)	限制必须为一个不大于指定值的数字
@DecimalMin(value)	限制必须为一个不小于指定值的数字
@Digits(integer,fraction)	限制必须为一个小数，且整数部分的位数不能超过integer，小数部分的位数不能超过fraction
@Future	限制必须是一个将来的日期
@Max(value)	限制必须为一个不大于指定值的数字
@Min(value)	限制必须为一个不小于指定值的数字
@Past	验证注解的元素值（日期类型）比当前时间早
@Pattern(value)	限制必须符合指定的正则表达式
@Size(max,min)	限制字符串长度必须在min到max之间
@NotEmpty	验证注解的元素值不为null且不为空（字符串长度不为0、集合大小不为0）
@NotBlank	验证注解的元素值不为空（不为null、去除首位空格后长度为0），不同于@NotEmpty，@NotBlank只应用于字符串且在比较时会去除字符串的空格
@Email	验证注解的元素值是Email，也可以通过正则表达式和flag指定自定义的email格式

Validation 支持如下几种校验，可以满足基本的业务逻辑，当然如果还是满足不了你的业务逻辑，可以选择定制校验规则。

定制校验逻辑

1. 定义校验逻辑规则，实现 `org.springframework.validation.Validator`

```
public class ConfigPropertiesValidator implements Validator {
    @Override
    public boolean supports(Class<?> aClass) {
        return AppConfigProperties.class.isAssignableFrom(aClass);
    }

    public void validate(Object object, Errors errors) {
        AppConfigProperties config = (AppConfigProperties) object;
        if(StringUtils.isEmpty(config.getId())){
            errors.rejectValue("id", "app.id.empty", "[app.id] 属性必须要在配置文件配置");
        }else if (config.getId().length() < 5) {
            errors.rejectValue("id", "app.id.short", "[app.id] 属性的长度必须不能小于5");
        }
    }
}
```

```
        }
    }
}
```

1. 使用自定义校验规则就不需要再使用原生的@NotEmpty了，将其删除

```
@Validated
@Data
@ConfigurationProperties(prefix = "app")
@Component
public class AppConfigProperties {
    // @NotEmpty(message = "配置文件配置必须要配置[app.id]属性")
    private String id;
}
```

1. 注入自定义校验规则

```
@Bean
public ConfigPropertiesValidator configurationPropertiesValidator() {
    return new ConfigPropertiesValidator();
}
```

注意：这里bean的方法名必须要使用 configurationPropertiesValidator，否则启动的时候不会执行该校验

1. 修改app.id配置，观察启动情况

```
*****
APPLICATION FAILED TO START
*****  
  
Description:  
  
Binding to target org.springframework.boot.context.properties.bind.BindException: Failed to bind properties under 'app'  
to org.jeecg.validate.AppConfigProperties failed:  
  
    Property: app.id  
    Value:  
    Origin: class path resource [application-dev.yml]:2:6  
    Reason: [app.id] 属性必须要在配置文件配置  
  
*****
APPLICATION FAILED TO START
*****  
  
Description:  
  
Binding to target org.springframework.boot.context.properties.bind.BindException: Failed to bind properties under 'app'  
to org.jeecg.validate.AppConfigProperties failed:  
  
    Property: app.id  
    Value: abcd  
    Origin: "app.id" from property source "systemProperties"  
    Reason: [app.id] 属性的长度必须不能小于5
```

错误信息说明了我们自定义校验的错误。

小结

通过配置Spring Boot启动校验功能，可以快速的识别参数配置的错误，避免在使用组件的时候才发现问题，可以减少排查问题的工作量，并且在我们封装自定义starter时可以有更好的体验。试想一下，你们很多项目共用部门一个Redis集群，那不同项目之间肯定要进行key隔离，这时候你提供一个公共的redis starter，强制必须在使用时指定 `app.id` 作为redis的自定义key前缀，岂不是会受到领导大大的赞美？

Watermark