

声明:文档来源于网络, 仅用于学习请勿擅自传播, 如果侵犯了你的权益, 请联系楼主 QQ  
2694449060, pigx 的代码有需要的也可以联系仅用于学习请勿擅自传播

## 目录

声明:文档来源于网络, 如果侵犯了你的权益, 请联系楼主 .....	1
开发契约 .....	4
Lombok 使用及其技巧说明 .....	4
统一工具类使用说明 .....	7
pig lambda 使用及其常用技巧汇总 .....	9
Mybatis Plus 3 的语法糖演示 .....	9
命令式和函数式 .....	9
什么是函数式编程? .....	10
行为参数化 .....	10
lambda 管中窥豹 .....	10
函数描述符 .....	10
函数式接口, 类型推断 .....	11
Java 8 中的常用函数式接口 .....	11
Lambdas 及函数式接口的例子 .....	12
方法引用 .....	12
本节课小结 .....	12
pig stream api 使用及其常用技巧汇总 .....	13
使用流 .....	16
中间操作和收集操作 .....	22
Collector 收集 .....	22
汇总 .....	23
网关功能 .....	28
网关限流使用 .....	28
网关降级处理 .....	33
路由转发配置 .....	35
认证授权 .....	37
资源服务器配置 .....	37
获取当前用户信息 .....	38
API 直接对外暴露 .....	39
现象 .....	40
服务暴露 .....	40
服务间鉴权及其 token 传递 .....	41
客户端带 Token 情况 .....	41
核心代码 .....	42
postman 等多终端接口调用 .....	44
pig 生成 token 详解 .....	45
pig 的 Oauth2.0 认证流程详解 .....	45
pig CheckToken 过程讲解 .....	70
CheckToken 的目的 .....	70
详解 .....	71

pig 扩展支持 oauth2 客户端模式.....	74
功能使用.....	75
代码生成使用 .....	76
EnablePigFeignClients 原理解析.....	79
Pig 中 FeignClient 使用说明.....	81
验证码配置及开关.....	83
Pig 配置加解密 .....	86
jasypt 的解决方案 .....	86
前端报文加密及其解密处理.....	87
登录后置处理 .....	89
日志处理-Spring Event 处理机制.....	91
监控模块二次认证.....	93
监控模块使用之 web 展示实时日志 .....	95
监控模块使用之动态日志级别.....	97
前端开发.....	98
配置 npm 镜像 .....	98
将 Npm 的源替换成淘宝的源 .....	99
登录详解 .....	99
自定义返回码提示 .....	100
按钮权限控制 .....	101
pig 如何控制菜单权限控制 .....	101
图标引入 .....	103
生产部署.....	104
Docker Compose 部署.....	104
安装 Docker (centos7) .....	105
安装 docker-compose .....	105
pig 打包 .....	105
等待 3 分钟 .....	106
总结 .....	107
前端部署 .....	107
打包 .....	107
nginx .....	107
微服务资源.....	108
PPT 整理.....	108
视频整理 .....	110
IDEA 学习视频 .....	110
Spring Boot 学习视频.....	111
Spring Cloud 学习视频 .....	111
Spring Security 学习视频 .....	111
分布式事物学习视频 .....	111
从无到有搭建中小型互联网公司后台服务架构与运维架构 .....	111
亿级流量电商详情页系统的大型高并发与高可用缓存架构实战 .....	112
高可用 RabbitMQ .....	112
高可用 MySQL .....	112

Docker .....	112
Nginx (必看) .....	112
博客整理 .....	112
oAuth2 开发指南 .....	116
介绍 .....	117
OAuth 2.0 提供者 .....	117
OAuth 2.0 提供者实现类 .....	117
授权服务器配置 .....	118
定制用户界面 .....	124
自定义错误处理 .....	126
将用户角色映射到作用域。 .....	127
资源服务器配置 .....	127
OAuth 2.0 客户端 .....	129
为外部 OAuth2 提供者的客户定制。 .....	133
Pig1.0 文档 .....	133
快速开始 .....	133
运行环境 .....	134
初始化环境 .....	135
Pig 项目初始化 .....	139
开发教程 .....	145
jasypt 的解决方案 .....	145
多维度限流 .....	147
代码生成使用 .....	150
暴露 API 网关 .....	151
验证码开关 .....	152
终端接口调用 .....	154
跨域处理 .....	155
SSO 单点登录实现 .....	157
手机号登录实现 .....	163
获取当前用户 .....	172
数据权限 .....	174
功能扩展 .....	176
Redis 集群 .....	176
Docker 搭建 Redis-Cluster .....	183
配置本地化 .....	184
新增业务微服务 (一) .....	187
新增业务微服务 (二) .....	192
新增业务微服务 (三) .....	194
zuul 动态路由实现 .....	195
zuul 动态加载 Filter .....	204
企业级功能 .....	207
分库分表 .....	207
任务调度 .....	209
链路追踪 .....	212

缓存管理 .....	220
文件系统 .....	225
聚合文档 .....	240
灰度发布 .....	241
生产部署 .....	243
Docker 部署 .....	243
高可用 .....	244
配置中心高可用 .....	246
eureka 注册中心高可用 .....	247
consul 注册中心高可用 .....	249
生产调优参考资料 .....	253

# 开发契约

## Lombok 使用及其技巧说明

### 为什么使用 lombok

还在编写无聊枯燥又难以维护的 **POJO** 吗？ 洁癖者的春天在哪里？请看

**Lombok**！在过往的 Java 项目中，充斥着太多不友好的代码：**POJO** 的  
`getter/setter/toString;` 异常处理；I/O 流的关闭操作等等，这些样板代码既没有技  
术含量，又影响着代码的美观，**Lombok** 应运而生。首先说明一下：任何技术的出  
现都是为了解决某一类问题的，如果在此基础上再建立奇技淫巧，不如回归 Java  
本身。应该保持合理使用而不滥用。

## 如何安装

当前你使用的 ide 未安装 lombok. lombok 能够达到的效果就是在源码中不需要写一些通用的方法，但是在编译生成的字节码文件中会帮我们生成这些方法，减少代码冗余。

[IDEA 安装方法 |](#)

[eclipse 安装方法](#)

pig 中的使用例子，不讲常见的、技巧性的

- @AllArgsConstructor 替代@Autowired 构造注入，多个 bean 注入时更加清晰 L

```
@Slf4j
@Configuration
@AllArgsConstructor
public class RouterFunctionConfiguration {
    private final HystrixFallbackHandler hystrixFallbackHandler;
    private final ImageCodeHandler imageCodeHandler;

}
```

```
@Slf4j
@Configuration
public class RouterFunctionConfiguration {
    @Autowired
    private HystrixFallbackHandler hystrixFallbackHandler;
    @Autowired
    private ImageCodeHandler imageCodeHandler;
}
```

- @SneakyThrows

```
@SneakyThrows
private void checkCode(ServerHttpRequest request) {
```

```

        String code = request.getQueryParams().getFirst("code");

        if (StrUtil.isBlank(code)) {
            throw new ValidateCodeException("验证码不能为空");
        }

        redisTemplate.delete(key);
    }

// 不使用就要加这个抛出
private void checkCode(ServerHttpRequest request) throws ValidateCodeException {
    String code = request.getQueryParams().getFirst("code");

    if (StrUtil.isBlank(code)) {
        throw new ValidateCodeException("验证码不能为空");
    }
}

```

- `@UtilityClass` 工具类再也不用定义 static 的方法了，直接就可以 Class.Method 使用

```

@UtilityClass
public class Utility {

    public String getName() {
        return "name";
    }

    public static void main(String[] args) {
        System.out.println(Utility.getName());
    }
}

```

- `@Cleanup`: 清理流对象,不用手动去关闭流，多么优雅

```

@Cleanup
OutputStream outStream = new FileOutputStream(new File("text.txt"));
@Cleanup

```

```
InputStream inStream = new FileInputStream(new File("text2.txt"));
byte[] b = new byte[65536];
while (true) {
    int r = inStream.read(b);
    if (r == -1) break;
    outStream.write(b, 0, r);
}
```

## 总结

Lombok 常用的注解就那么几个，`@Data`、`@Getter/Setter`，`Pig` 使用例子中的几个可以让代码的更加优雅，建议在你的工程中使用

## 统一工具类使用说明

### 统一工具类的意义

Hutool 帮助我们简化每一行代码，减少每一个方法，然代码可读性、容错性更高。完整文档方便使用 [hutool-doc](#), 避免每个开发乱引入造成辣鸡代码。

强制使用 hutool 工具类

### hutool 提供类哪些功能

一个 Java 基础工具类，对文件、流、加密解密、转码、正则、线程、XML 等 JDK 方法进行封装，组成各种 Util 工具类，同时提供以下组件：

- hutool-aop JDK 动态代理封装，提供非 IOC 下的切面支持
- hutool-bloomFilter 布隆过滤，提供一些 Hash 算法的布隆过滤
- hutool-cache 缓存
- hutool-core 核心，包括 Bean 操作、日期、各种 Util 等
- hutool-cron 定时任务模块，提供类 Crontab 表达式的定时任务

- hutool-crypto 加密解密模块
- hutool-db JDBC 封装后的数据操作，基于 ActiveRecord 思想
- hutool-dfa 基于 DFA 模型的多关键字查找
- hutool-extra 扩展模块，对第三方封装（模板引擎、邮件、Servlet、二维码等）
- hutool-http 基于 HttpURLConnection 的 Http 客户端封装
- hutool-log 自动识别日志实现的日志门面
- hutool-script 脚本执行封装，例如 Javascript
- hutool-setting 功能更强大的 Setting 配置文件和 Properties 封装
- hutool-system 系统参数调用封装（JVM 信息等）
- hutool-json JSON 实现
- hutool-captcha 图片验证码实现
- hutool-poi 针对 POI 中 Excel 的封装

可以根据需求对每个模块单独引入，也可以通过引入 hutool-all 方式引入所有模块。

## Pig 中的使用

业务模块使用 pig-common-security 的时候会自动引入 pig-common-core 模块，引入最新的 hutool-all 工具类

```
<!--hutool-->
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
    <version>${hutool.version}</version>
</dependency>
```

## 使用例子

比如记录日志时候，从 request 获取参数的字符串

```
HttpUtil.toParams(request.getParameterMap())
```

获取当前时间

```
//当前时间，格式 yyyy-MM-dd HH:mm:ss  
DateUtil.now()
```

## 个人建议

pig 在 2.0 中尽量避免自己造轮子封装工具类，让代码可读性更高，所以我建议各位工程师在实际开发过程中一定要注意工具类这个问题。

## pig lambda 使用及其常用技巧汇总

## Mybatis Plus 3 的语法糖演示

```
@Override  
@CacheEvict(value = "menu_details", allEntries = true)  
@Transactional(rollbackFor = Exception.class)  
public Boolean removeRoleById(Integer id) {  
    sysRoleMenuMapper.delete(Wrappers  
        .  
        .  
        .eq(SysRoleMenu::getRoleId, id));  
    return this.removeById(id);  
}
```

## 命令式和函数式

**命令式编程：**命令“机器”如何去做事情(how)，这样不管你想要的是什么(what)，它都会按照你的命令实现。

**声明式编程：**告诉“机器”你想要的是什么(what)，让机器想出如何去做(how)。

## 什么是函数式编程？

---

每个人对函数式编程的理解不尽相同。我的理解是：在完成一个编程任务时，通过使用不可变的值或函数，对他们进行处理，然后得到另一个值的过程。

不同的语言社区往往对各自语言中的特性孤芳自赏。现在谈 Java 程序员如何定义函数式编程还为时尚早，但是，这根本不重要！

我们关心的是如何写出好代码，而不是符合函数式编程风格的代码。

## 行为参数化

---

把算法的策略（行为）作为一个参数传递给函数。

## lambda 管中窥豹

---

- 匿名：它不像普通的方法那样有一个明确的名称：写得少而想得多！
- 函数：Lambda 函数不像方法那样属于某个特定的类。但和方法一样，Lambda 有参数列表、函数主体、返回类型，还可能有可以抛出的异常列表。
- 传递：Lambda 表达式可以作为参数传递给方法或存储在变量中。
- 简洁：无需像匿名类那样写很多模板代码。

## 函数描述符

---

函数式接口的抽象方法的签名基本上就是 Lambda 表达式的签名，这种抽象方法叫作函数描述符。

## 函数式接口，类型推断

函数式接口定义且只定义了一个抽象方法，因为抽象方法的签名可以描述 Lambda 表达式的签名。

函数式接口的抽象方法的签名为函数描述符。

所以为了应用不同的 Lambda 表达式，你需要一套能够描述常见函数描述符的函数式接口。

## Java 8 中的常用函数式接口

函数式接口	函数描述符	原始类型特化
Predicate<T>	T->boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T->void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T->R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>
Supplier<T>	()->T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T->T	IntUnaryOperator, DoubleUnaryOperator, LongUnaryOperator
BinaryOperator<T>	(T, T)->T	IntBinaryOperator, DoubleBinaryOperator, LongBinaryOperator
BiPredicate<L, R>	(L, R)->boolean	
BiConsumer<T, U>	(T, U)->void	ObjIntConsumer<T>, ObjDoubleConsumer<T>, ObjLongConsumer<U>
BiFunction<T, U, R>	(T, U)->R	ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>

## Lambdas 及函数式接口的例子

使用案例	Lambda 的例子	对应的函数式接口
布尔表达式	<code>(List&lt;String&gt; list) -&gt; list.isEmpty()</code>	<code>Predicate&lt;List&lt;String&gt;&gt;</code>
创建对象	<code>() -&gt; new Project()</code>	<code>Supplier&lt;Project&gt;</code>
消费一个对象	<code>(Project p) -&gt; System.out.println(p.getStars())</code>	<code>Consumer&lt;Project&gt;</code>
从一个对象中选择/提取	<code>(int a, int b) -&gt; a * b</code>	<code>IntBinaryOperator</code>
比较两个对象	<code>(Project p1, Project p2) -&gt; p1.getStars().compareTo(p2.getStars())</code>	<code>Comparator&lt;Project&gt;</code> <code>BiFunction&lt;Project, Project, Integer&gt;</code> <code>ToIntBiFunction&lt;Project, Project&gt;</code>

## 方法引用

方法引用让你可以重复使用现有的方法定义，并像 Lambda 一样传递它们。

## 本节课小结

- lambda 表达式可以理解为一种匿名函数：它没有名称，但有参数列表、函数主体、返回类型，可能还有一个可以抛出的异常的列表。
- lambda 表达式让你可以简洁地传递代码。

- 函数式接口就是仅仅声明了一个抽象方法的接口。
- 只有在接受函数式接口的地方才可以使用 `lambda` 表达式。
- `lambda` 表达式允许你直接内联，为函数式接口的抽象方法提供实现，并且将整个表达式作为函数式接口的一个实例。
- Java 8 自带一些常用的函数式接口，放在 `java.util.function` 包里，包括 `Predicate<T>`、`Function<T,R>`、`Supplier<T>`、`Consumer<T>` 和 `BinaryOperator<T>`。
- `Lambda` 表达式所需要代表的类型称为目标类型。
- 方法引用让你重复使用现有的方法实现并直接传递它们。
- `Comparator``、`Predicate` 和 `Function`` 等函数式接口都有几个可以用来结合 `lambda` 表达式的默认方法。

## pig stream api 使用及其常用技巧汇总

什么是流？

先来看看 Pig upms 中的使用

```
~~~
@Override
@Transactional(rollbackFor = Exception.class)
public Boolean saveUser(UserDTO userDto) {
    SysUser sysUser = new SysUser();
    BeanUtils.copyProperties(userDto, sysUser);
    sysUser.setDelFlag(CommonConstants.STATUS_NORMAL);
    sysUser.setPassword(ENCODER.encode(userDto.getPassword()));
    baseMapper.insert(sysUser);
    List<SysUserRole> userRoleList = userDto.getRole()
        .stream().map(roleId -> {
            SysUserRole userRole = new SysUserRole();
            userRole.setUserId(sysUser.getUserId());
            userRole.setRoleId(roleId);
            return userRole;
        })
        .collect(Collectors.toList());
    baseMapper.insertBatch(userRoleList);
    return true;
}
```

```
    }).collect(Collectors.toList());
    return sysUserRoleService.saveBatch(userRoleList);
}
~~~
```

流是 Java8 引入的全新概念，它用来处理集合中的数据，暂且可以把它理解为一种高级集合。

众所周知，集合操作非常麻烦，若要对集合进行筛选、投影，需要写大量的代码，而流是以声明的形式操作集合，它就像 SQL 语句，我们只需告诉流需要对集合进行什么操作，它就会自动进行操作，并将执行结果交给你，无需我们自己手写代码。

因此，流的集合操作对我们来说是透明的，我们只需向流下达命令，它就会自动把我们想要的结果给我们。由于操作过程完全由 Java 处理，因此它可以根据当前硬件环境选择最优的方法处理，我们也无需编写复杂又容易出错的多线程代码了。

## 流的特点

### 1. 只能遍历一次

我们可以把流想象成一条流水线，流水线的源头是我们的数据源(一个集合)，数据源中的元素依次被输送到流水线上，我们可以在流水线上对元素进行各种操作。一旦元素走到了流水线的另一头，那么这些元素就被“消费掉了”，我们无法再对这个流进行操作。当然，我们可以从数据源那里再获得一个新的流重新遍历一遍。

### 2. 采用内部迭代方式

若要对集合进行处理，则需我们手写处理代码，这就叫做外部迭代。  
而要对流进行处理，我们只需告诉流我们需要什么结果，处理过程由流自行完成，这就称为内部迭代。

## 流的操作种类

流的操作分为两种，分别为中间操作和终端操作。

### 1. 中间操作

当数据源中的数据上了流水线后，这个过程对数据进行的所有操作都称为“中间操作”。

中间操作仍然会返回一个流对象，因此多个中间操作可以串连起来形成一个流水线。

### 2. 终端操作

当所有的中间操作完成后，若要将数据从流水线上拿下来，则需要执行终端操作。

终端操作将返回一个执行结果，这就是你想要的数据。

## 流的操作过程

使用流一共需要三步：

1. 准备一个数据源
2. 执行中间操作

中间操作可以有多个，它们可以串连起来形成流水线。

3. 执行终端操作

执行终端操作后本次流结束，你将获得一个执行结果。

# 使用流

---

## 创建流

在使用流之前，首先需要拥有一个数据源，并通过 StreamAPI 提供的一些方法获取该数据源的流对象。数据源可以有多种形式：

### 1. 集合

这种数据源较为常用，通过 stream() 方法即可获取流对象：

```
List<Person> list = new ArrayList<Person>();
Stream<Person> stream = list.stream();
```

### 2. 数组

通过 Arrays 类提供的静态函数 stream() 获取数组的流对象：

```
String[] names = {"chaimm", "peter", "john"};
Stream<String> stream = Arrays.stream(names);
```

### 3. 值

直接将几个值变成流对象：

```
Stream<String> stream = Stream.of("chaimm", "peter", "john");
```

### 4. 文件

```
try(Stream lines = Files.lines(Paths.get("文件路径名"), Charset.defaultCharset())){
    //可对 lines 做一些操作
} catch(IOException e){
}
```

### 5. iterator

## 创建无限流

```
Stream.iterate(0, n -> n + 2)
    .limit(10)
    .forEach(System.out::println);
```

PS：Java7 简化了 IO 操作，把打开 IO 操作放在 try 后的括号中即可省略关闭 IO

的代码。

## 筛选 filter

filter 函数接收一个 Lambda 表达式作为参数，该表达式返回 boolean，在执行过

程中，流将元素逐一输送给 filter，并筛选出执行结果为 true 的元素。

如，筛选出所有学生：

```
List<Person> result = list.stream()
    .filter(Person::isStudent)
    .collect(toList());
```

## 去重 distinct

去掉重复的结果：

```
List<Person> result = list.stream()
    .distinct()
    .collect(toList());
```

## 截取

截取流的前 N 个元素：

```
List<Person> result = list.stream()
    .limit(3)
    .collect(toList());
```

## 跳过

跳过流的前 n 个元素：

```
List<Person> result = list.stream()
    .skip(3)
    .collect(toList());
```

## 映射

对流中的每个元素执行一个函数，使得元素转换成另一种类型输出。流会将每一个

元素输送给 map 函数，并执行 map 中的 Lambda 表达式，最后将执行结果存入一

一个新的流中。

如，获取每个人的姓名(实则是将 Person 类型转换成 String 类型):

```
List<Person> result = list.stream()
    .map(Person::getName)
    .collect(toList());
```

## 合并多个流

例：列出 List 中各不相同的单词，List 集合如下：

```
List<String> list = new ArrayList<String>();
list.add("I am a boy");
list.add("I love the girl");
list.add("But the girl loves another girl");
```

思路如下：

首先将 list 变成流：

```
list.stream();
```

按空格分词：

```
list.stream()
    .map(line->line.split(" "));
```

分完词之后，每个元素变成了一个 String[] 数组。

将每个 String[] 变成流：

```
list.stream()
    .map(line->line.split(" "))
    .map(Arrays::stream)
```

此时一个大流里面包含了一个个小流，我们需要将这些小流合并成一个流。

将小流合并成一个大流：用 flatMap 替换刚才的 map

```
list.stream()
    .map(line->line.split(" "))
    .flatMap(Arrays::stream)
```

去重

```
list.stream()
    .map(line->line.split(" "))
    .flatMap(Arrays::stream)
```

```
.distinct()  
.collect(toList());
```

### 是否匹配任一元素: anyMatch

anyMatch 用于判断流中是否存在至少一个元素满足指定的条件，这个判断条件通过 Lambda 表达式传递给 anyMatch，执行结果为 boolean 类型。

如，判断 list 中是否有学生：

```
boolean result = list.stream()  
.anyMatch(Person::isStudent);
```

### 是否匹配所有元素: allMatch

allMatch 用于判断流中的所有元素是否都满足指定条件，这个判断条件通过 Lambda 表达式传递给 anyMatch，执行结果为 boolean 类型。

如，判断是否所有人都是学生：

```
boolean result = list.stream()  
.allMatch(Person::isStudent);
```

### 是否未匹配所有元素: noneMatch

noneMatch 与 allMatch 恰恰相反，它用于判断流中的所有元素是否都不满足指定条件：

```
boolean result = list.stream()  
.noneMatch(Person::isStudent);
```

### 获取任一元素 findAny

findAny 能够从流中随便选一个元素出来，它返回一个 Optional 类型的元素。

```
Optional<Person> person = list.stream().findAny();
```

### 获取第一个元素 findFirst

```
Optional<Person> person = list.stream().findFirst();
```

## 归约

归约是将集合中的所有元素经过指定运算，折叠成一个元素输出，如：求最值、平

均数等，这些操作都是将一个集合的元素折叠成一个元素输出。

在流中，`reduce` 函数能实现归约。

`reduce` 函数接收两个参数：

1. 初始值
2. 进行归约操作的 Lambda 表达式

**元素求和：自定义 Lambda 表达式实现求和**

例：计算所有人的年龄总和

```
int age = list.stream().reduce(0, (person1, person2) -> person1.getAge() + person2.getAge());
```

1. `reduce` 的第一个参数表示初试值为 0；
2. `reduce` 的第二个参数为需要进行的归约操作，它接收一个拥有两个参数的 Lambda 表达式，`reduce` 会把流中的元素两两输给 Lambda 表达式，最后将计算出累加之和。

**元素求和：使用 `Integer.sum` 函数求和**

上面的方法中我们自己定义了 Lambda 表达式实现求和运算，如果当前流的元素为数值类型，那么可以使用 `Integer` 提供了 `sum` 函数代替自定义的 Lambda 表达式，如：

```
int age = list.stream().reduce(0, Integer::sum);
```

`Integer` 类还提供了 `min`、`max` 等一系列数值操作，当流中元素为数值类型时可以直接使用。

## 数值流的使用

采用 `reduce` 进行数值操作会涉及到基本数值类型和引用数值类型之间的装箱、拆箱操作，因此效率较低。

当流操作为纯数值操作时，使用数值流能获得较高的效率。

### 将普通流转换成数值流

`StreamAPI` 提供了三种数值流：`IntStream`、`DoubleStream`、`LongStream`，也提供了将普通流转换成数值流的三种方法：`mapToInt`、`mapToDouble`、`mapToLong`。

如，将 `Person` 中的 `age` 转换成数值流：

```
IntStream stream = list.stream().mapToInt(Person::getAge);
```

### 数值计算

每种数值流都提供了数值计算函数，如 `max`、`min`、`sum` 等。如，找出最大的年龄：

```
OptionalInt maxAge = list.stream()
    .mapToInt(Person::getAge)
    .max();
```

由于数值流可能为空，并且给空的数值流计算最大值是没有意义的，因此 `max` 函数返回 `OptionalInt`，它是 `Optional` 的一个子类，能够判断流是否为空，并对流为空的情况作相应的处理。

此外，`mapToInt`、`mapToDouble`、`mapToLong` 进行数值操作后的返回结果分别为：`OptionalInt`、`OptionalDouble`、`OptionalLong`

## 中间操作和收集操作

操作	类型	返回类型	使用的类型/函数式接口	函数描述符
filter	中间	Stream<T>	Predicate<T>	T -> boolean
distinct	中间	Stream<T>		
skip	中间	Stream<T>	long	
map	中间	Stream<R>	Function<T, R>	T -> R
flatMap	中间	Stream<R>	Function<T, Stream<R>>	T -> Stream<R>
limit	中间	Stream<T>	long	
sorted	中间	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	终端	boolean	Predicate<T>	T -> boolean
noneMatch	终端	boolean	Predicate<T>	T -> boolean
allMatch	终端	boolean	Predicate<T>	T -> boolean
findAny	终端	Optional<T>		
findFirst	终端	Optional<T>		
forEach	终端	void	Consumer<T>	T -> void
collect	终端	R	Collector<T, A, R>	
reduce	终端	Optional<T>	BinaryOperator<T>	(T, T) -> R
count	终端	long		

## Collector 收集

收集器用来将经过筛选、映射的流进行最后的整理，可以使得最后的结果以不同的形式展现。

`collect` 方法即为收集器，它接收 `Collector` 接口的实现作为具体收集器的收集方法。

`Collector` 接口提供了很多默认实现的方法，我们可以直接使用它们格式化流的结果；也可以自定义 `Collector` 接口的实现，从而定制自己的收集器。

## 归约

流由一个个元素组成，归约就是将一个个元素“折叠”成一个值，如求和、求最值、求平均值都是归约操作。

### 一般性归约

若你需要自定义一个归约操作，那么需要使用 `Collectors.reducing` 函数，该函数接收三个参数：

- 第一个参数为归约的初始值
- 第二个参数为归约操作进行的字段
- 第三个参数为归约操作的过程

## 汇总

---

`Collectors` 类专门为汇总提供了一个工厂方法：`Collectors.summingInt`。

它可接受一个把对象映射为求和所需 `int` 的函数，并返回一个收集器；该收集器在传递给普通的 `collect` 方法后即执行我们需要的汇总操作。

## 分组

数据分组是一种更自然的分割数据操作，分组就是将流中的元素按照指定类别进行划分，类似于 `SQL` 语句中的 `GROUPBY`。

## 多级分组

多级分组可以支持在完成一次分组后，分别对每个小组再进行分组。

使用具有两个参数的 `groupingBy` 重载方法即可实现多级分组。

- 第一个参数：一级分组的条件
- 第二个参数：一个新的 `groupingBy` 函数，该函数包含二级分组的条件

## Collectors 类的静态工厂方法

工厂方法	返回类型	用途	示例
<code>toList</code>	<code>List&lt;T&gt;</code>	把流中所有项目收集到一个 List	<code>List&lt;Project&gt; projects = projectStream.collect(Collectors.toList());</code>
<code>toSet</code>	<code>Set&lt;T&gt;</code>	把流中所有项目收集到一个 Set，删除重复项	<code>Set&lt;Project&gt; projects = projectStream.collect(Collectors.toSet());</code>
<code>toCollection</code>	<code>Collection&lt;T&gt;</code>	把流中所有项目收集到给定的供应商创建的集合	<code>Collection&lt;Project&gt; projects = projectStream.collect(Collectors.toCollection(() -&gt; new TreeSet&lt;Project&gt;()));</code>
<code>counting</code>	<code>Long</code>	计算流中元素的个数	<code>long howManyProjects = projectStream.collect(Collectors.counting());</code>
<code>summingInt</code>	<code>Integer</code>	对流中项目的一个整数属性求和	<code>int sumOfProjectIntegers = projectStream.collect(Collectors.summingInt(Project::getInteger));</code>
<code>averagingInt</code>	<code>Double</code>	计算流中项目 Integer 属性的平均值	<code>double averageOfProjectIntegers = projectStream.collect(Collectors.averagingInt(Project::getInteger));</code>
<code>summarizingInt</code>	<code>IntSummaryStatistics</code>	收集关于流中项目 Integer 属性的统计值，例如最大、最小、总和与平均值	<code>IntSummaryStatistics stats = projectStream.collect(Collectors.summarizingInt(Project::getInteger));</code>
<code>joining</code>	<code>String</code>	连接对流中每个项目调用 <code>toString</code> 方法所生成的字符串	<code>String joinedProjects = projectStream.map(Project::toString).collect(Collectors.joining());</code>

工厂方法	返回类型	用途	示例
<code>maxBy</code>	<code>Optional&lt;T&gt;</code>	按照给定比较器选出的最大元素的 <code>Optional</code> , 或如果流为空则为 <code>Optional.empty()</code>	<code>Optional&lt;Project&gt; projectStream.collect(...)</code>
<code>minBy</code>	<code>Optional&lt;T&gt;</code>	按照给定比较器选出的最小元素的 <code>Optional</code> , 或如果流为空则为 <code>Optional.empty()</code>	<code>Optional&lt;Project&gt; projectStream.collect(...)</code>
<code>reducing</code>	归约操作产生的类型	从一个作为累加器的初始值开始, 利用 <code>BinaryOperator</code> 与流中的元素逐个结合, 从而将流归约为单个值	<code>int totalStars = projectStream.collect(...);</code>
<code>collectingAndThen</code>	转换函数返回的类型	包含另一个收集器, 对其结果应用转换函数	<code>int projectStream.collect(List::size));</code>
<code>groupingBy</code>	<code>Map&lt;K, List&lt;T&gt;&gt;</code>	根据项目的一个属性的值对流中的项目作分组, 并将属性值作为结果 Map 的键	<code>Map&lt;String, List&lt;Project&gt; projectStream.collect(...);</code>
<code>partitioningBy</code>	<code>Map&lt;Boolean, List&lt;T&gt;&gt;</code>	根据对流中每个项目应用断言的结果来对项目进行分区	<code>Map&lt;Boolean, List&lt;Project&gt; projectStream.collect(...);</code>

## 转换类型

有一些收集器可以生成其他集合。比如前面已经见过的 `toList`, 生成了 `java.util.List` 类的实例。

还有 `toSet` 和 `toCollection`, 分别生成 `Set` 和 `Collection` 类的实例。

到目前为止，我已经讲了很多流上的链式操作，但总有一些时候，需要最终生成一个集合——比如：

- 已有代码是为集合编写的，因此需要将流转换成集合传入；
- 在集合上进行一系列链式操作后，最终希望生成一个值；
- 写单元测试时，需要对某个具体的集合做断言。

使用 `toCollection`，用定制的集合收集元素

```
stream.collect(toCollection(TreeSet::new));
```

还可以利用收集器让流生成一个值。`maxBy` 和 `minBy` 允许用户按某种特定的顺序生成一个值。

## 数据分区

分区是分组的特殊情况：由一个断言（返回一个布尔值的函数）作为分类函数，它称分区函数。

分区函数返回一个布尔值，这意味着得到的分组 `Map` 的键类型是 `Boolean`，于是它最多可以分为两组：`true` 是一组，`false` 是一组。

分区的好处在于保留了分区函数返回 `true` 或 `false` 的两套流元素列表。

## 并行流

并行流就是一个把内容分成多个数据块，并用不同的线程分别处理每个数据块的流。最后合并每个数据块的计算结果。

将一个顺序执行的流转变成一个并发的流只要调用 `parallel()` 方法

```
public static long parallelSum(long n){  
    return Stream.iterate(1L, i -> i +1).limit(n).parallel().reduc  
e(0L, Long::sum);
```

```
}
```

将一个并发流转成顺序的流只要调用 `sequential()` 方法

```
stream.parallel().filter(...).sequential().map(...).parallel().reduce();
```

这两个方法可以多次调用，只有最后一个调用决定这个流是顺序的还是并发的。

并发流使用的默认线程数等于你机器的处理器核心数。

通过这个方法可以修改这个值，这是全局属性。

```
System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "12");
```

并非使用多线程并行流处理数据的性能一定高于单线程顺序流的性能，因为性能受到多种因素的影响。

如何高效使用并发流的一些建议：

1. 如果不确定，就自己测试。
2. 尽量使用基本类型的流 `IntStream`, `LongStream`, `DoubleStream`
3. 有些操作使用并发流的性能会比顺序流的性能更差，比如 `limit`, `findFirst`, 依赖元素顺序的操作在并发流中是极其消耗性能的。`findAny` 的性能就会好很多，应为不依赖顺序。
4. 考虑流中计算的性能(Q)和操作的性能(N)的对比，Q 表示单个处理所需的时间，N 表示需要处理的数量，如果 Q 的值越大，使用并发流的性能就会越高。
5. 数据量不大时使用并发流，性能得不到提升。
6. 考虑数据结构：并发流需要对数据进行分解，不同的数据结构被分解的性能时不一样的。

流的数据源和可分解性

源	可分解性
<code>ArrayList</code>	非常好
<code>LinkedList</code>	差
<code>IntStream.range</code>	非常好
<code>Stream.iterate</code>	差
<code>HashSet</code>	好
<code>TreeSet</code>	好

流的特性以及中间操作对流的修改都会对数据对分解性能造成影响。比如固定大小的流在任务分解的时候就可以平均分配，但是如果有 `filter` 操作，那么流就不能预先知道在这个操作后还会剩余多少元素。

考虑终端操作的性能：如果终端操作在合并并发流的计算结果时的性能消耗太大，那么使用并发流提升的性能就会得不偿失。

## 网关功能

### 网关限流使用

#### POM 依赖

这里一定要注意，是网关引入的 redis-reactive，背压模式的 redis。

```
<!--基于 reactive stream 的 redis -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
```

#### 配置按照请求 IP 的限流

```
spring:
  cloud:
    gateway:
```

```

routes:
- id: requestratelimiter_route
  uri: lb://pigx-upms
  order: 10000
  predicates:
  - Path=/admin/**
  filters:
  - name: RequestRateLimiter
    args:
      redis-rate-limiter.replenishRate: 1 # 令牌桶的容积
      redis-rate-limiter.burstCapacity: 3 # 流速 每秒
      key-resolver: "#{@remoteAddrKeyResolver}" #SPEL 表达式去
的对应的 bean
  - StripPrefix=1

```

- 配置 bean, 多维度限流量的入口 对应上边 key-resolver 【我自己哦

```

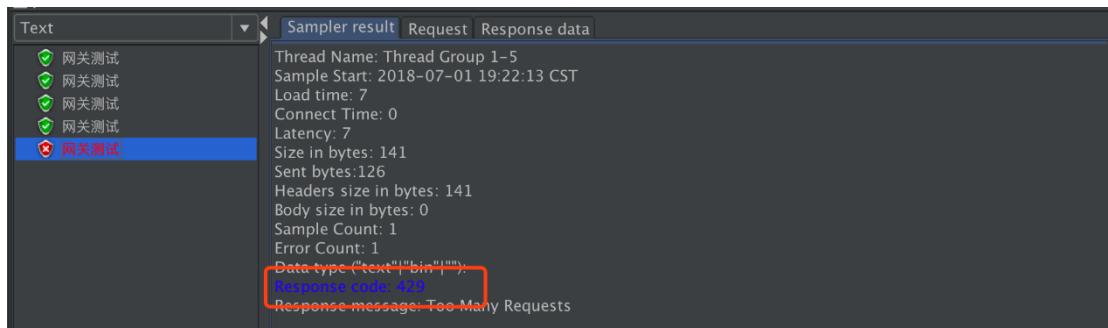
/**
 * 自定义限流标志的 key, 多个维度可以从这里入手
 * exchange 对象中获取服务 ID、请求信息, 用户信息等
 */
@Bean
KeyResolver remoteAddrKeyResolver() {
    return exchange -> Mono.just(exchange.getRequest().getRemoteAd
dress().getHostName());
}

```

OK 完成。

## 压力测试

并发 5 个线程。

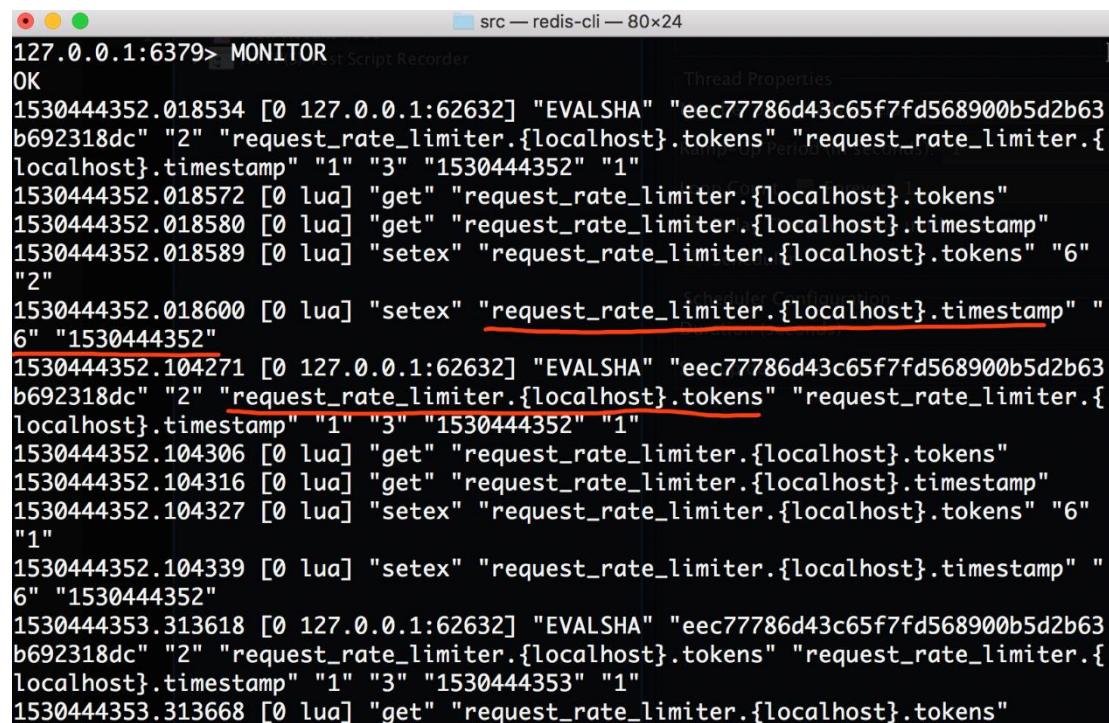


## Redis 数据变化

我们使用 redis 的 **monitor** 命令，实时查看 redis 的操作情况。

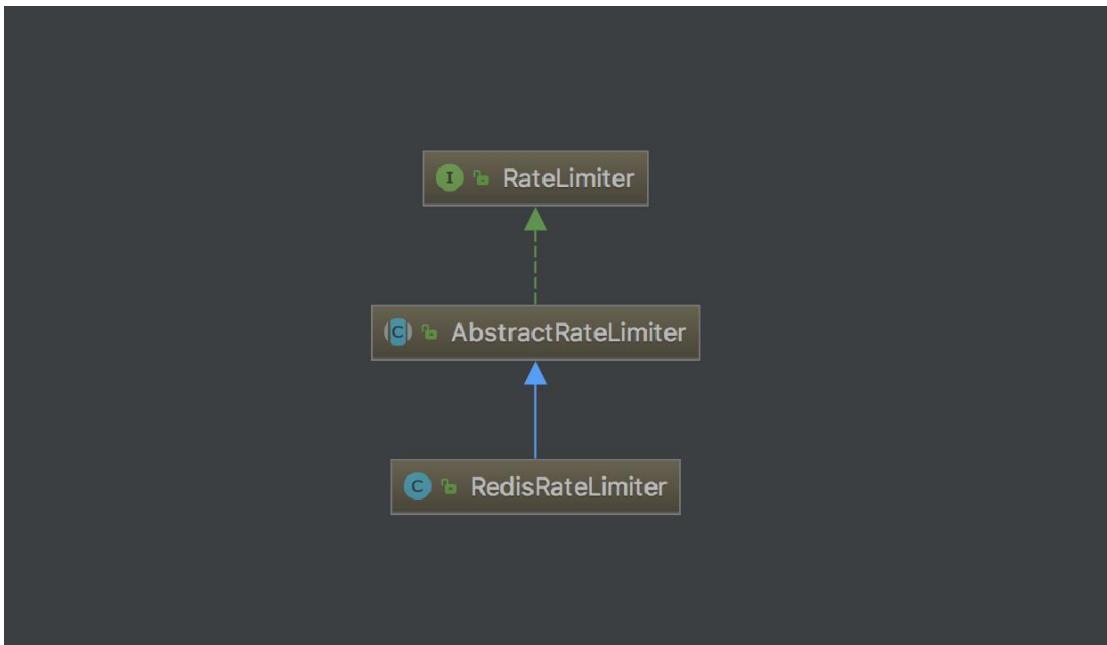
会发现在 redis 中会操作两个 key

- `request_rate_limiter.{xxx}.timestamp`
- `request_rate_limiter.{xxx}.tokens`



```
127.0.0.1:6379> MONITOR
OK
1530444352.018534 [0 127.0.0.1:62632] "EVALSHA" "eec77786d43c65f7fd568900b5d2b63
b692318dc" "2" "request_rate_limiter.{localhost}.tokens" "request_rate_limiter.{localhost}.timestamp" "1" "3" "1530444352" "1"
1530444352.018572 [0 lua] "get" "request_rate_limiter.{localhost}.tokens"
1530444352.018580 [0 lua] "get" "request_rate_limiter.{localhost}.timestamp"
1530444352.018589 [0 lua] "setex" "request_rate_limiter.{localhost}.tokens" "6"
"2"
1530444352.018600 [0 lua] "setex" "request_rate_limiter.{localhost}.timestamp" "6"
"1530444352"
1530444352.104271 [0 127.0.0.1:62632] "EVALSHA" "eec77786d43c65f7fd568900b5d2b63
b692318dc" "2" "request_rate_limiter.{localhost}.tokens" "request_rate_limiter.{localhost}.timestamp" "1" "3" "1530444352" "1"
1530444352.104306 [0 lua] "get" "request_rate_limiter.{localhost}.tokens"
1530444352.104316 [0 lua] "get" "request_rate_limiter.{localhost}.timestamp"
1530444352.104327 [0 lua] "setex" "request_rate_limiter.{localhost}.tokens" "6"
"1"
1530444352.104339 [0 lua] "setex" "request_rate_limiter.{localhost}.timestamp" "6"
"1530444352"
1530444353.313618 [0 127.0.0.1:62632] "EVALSHA" "eec77786d43c65f7fd568900b5d2b63
b692318dc" "2" "request_rate_limiter.{localhost}.tokens" "request_rate_limiter.{localhost}.timestamp" "1" "3" "1530444353" "1"
1530444353.313668 [0 lua] "get" "request_rate_limiter.{localhost}.tokens"
```

## 实现原理



Spring Cloud Gateway 默认实现 Redis 限流，如果扩展只需要实现 `ratelimter` 接口即可。

`RedisRateLimiter` 的核心代码，判断是否取到令牌的实现，通过调用 redis 的 LUA 脚本。

```
public Mono<Response> isAllowed(String routeId, String id) {
    Config routeConfig = getConfig().getOrDefault(routeId, defaultConfig);
    int replenishRate = routeConfig.getReplenishRate();
    int burstCapacity = routeConfig.getBurstCapacity();

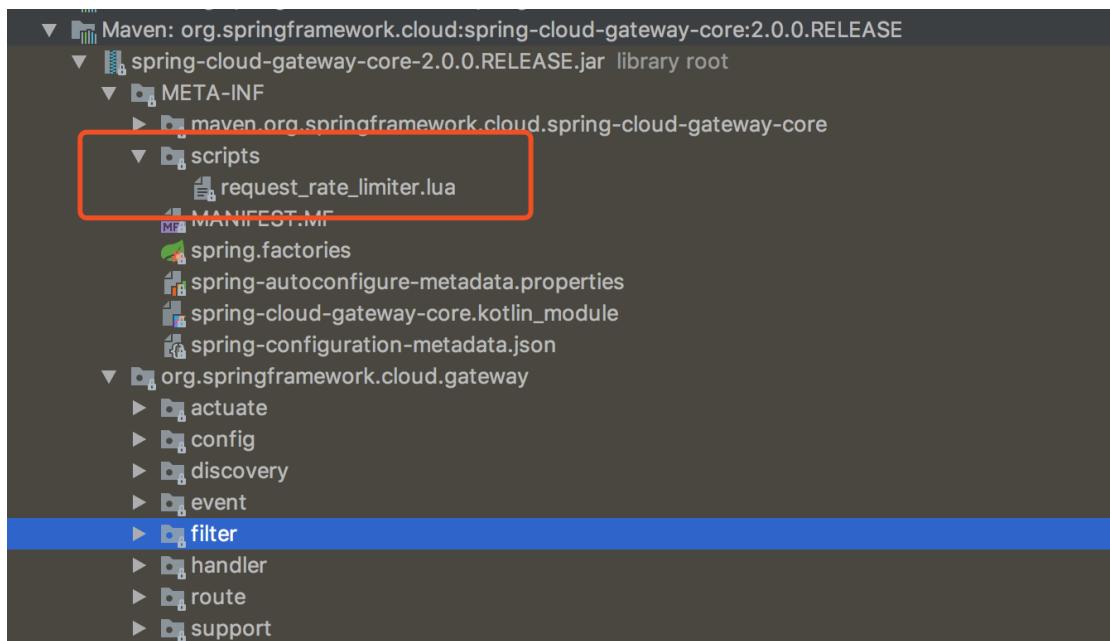
    try {
        List<String> keys = getKeys(id);
        returns unixtime in seconds.
        List<String> scriptArgs = Arrays.asList(replenishRate + "", burstCapacity + "",
                                                Instant.now().getEpochSecond() + "",
                                                "1");
        // 这里是核心，执行 redis 的 LUA 脚本。
        Flux<List<Long>> flux =
```

```
        this.redisTemplate.execute(this.script, keys, scriptArgs);
        return flux.onErrorResume(throwable -> Flux.just(Arrays.asList(1L, -1L)))
                .reduce(new ArrayList<Long>(), (long s, 1) -> {
                    longs.addAll(1);
                    return longs;
                }) .map(results -> {
                    boolean allowed = results.get(0) == 1L;
                    Long tokensLeft = results.get(1);

                    Response response = new Response(allowed, getHeaders(routeConfig, tokensLeft));

                    if (log.isDebugEnabled()) {
                        log.debug("response: " + response);
                    }
                    return response;
                });
    }
    catch (Exception e) {
        log.error("Error determining if user allowed from redis", e);
    }
    return Mono.just(new Response(true, getHeaders(routeConfig, -1L)));
}
```

## LUA 脚本



## 网关降级处理

### 网关降级

当网关流量向后转发的时候，如果微服务模块不可用，则会触发降级事件

### Pig 中的使用讲解，以 UPMS 路由配置为例

```
spring:
  cloud:
    gateway:
      locator:
        enabled: true
      routes:
        #UPMS 模块
        - id: pig-upms      # 唯一的服务 ID
          uri: lb://pig-upms # 注册中心的服务名称，实现负载均衡
          predicates:
            - Path=/admin/**  #所有业务的请求前缀
          filters:
            - name: Hystrix          #断路器降级策略
              args:
                name: default
                fallbackUri: 'forward:/fallback' # 降级接口的地址
```

## 原理

Spring Cloud Gateway 会自动寻找配置 Hystrix 的 Filter，这个功能是内置的，然

后回调我们提供 fallbackUri,Pig 中的降级接口是使用 webflux 实现的

- 降级业务很简单，输出那个接口的异常日志

```
@Slf4j
@Component
public class HystrixFallbackHandler implements HandlerFunction<ServerResponse> {
    @Override
    public Mono<ServerResponse> handle(ServerRequest serverRequest) {
        Optional<Object> originalUris = serverRequest.attribute(GATEWAY_ORIGINAL_REQUEST_URL_ATTR);

        originalUris.ifPresent(originalUri -> log.error("网
关执行请求:{}失败,hystrix 服务降级处理", originalUri));

        return ServerResponse.status(HttpStatus.INTERNAL_SE
RVER_ERROR.value())
            .contentType(MediaType.TEXT_PLAIN).body(Bod
yInserters.fromObject("服务异常"));
    }
}
```

- 降级入口。这里的意思类似于 SpringMVC 定义一个 @GetMapping("/fallback")

接口

```
@Slf4j
@Configuration
@AllArgsConstructor
public class RouterFunctionConfiguration {
    private final HystrixFallbackHandler hystrixFallbackHandle
r;
    private final ImageCodeHandler imageCodeHandler;

    @Bean
```

```
public RouterFunction routerFunction() {
    return RouterFunctions.route(
        RequestPredicates.path("/fallback")
            .and(RequestPredicates.accept(MediaType.TEXT_PLAIN)), hystrixFallbackHandler)
    }
}
```

## 路由转发配置

# Spring Cloud Gateway

Pig 2.0 采用的是 spring 官方的网关组件，通过异步背压的高性能网关。

路由配置是整个微服务中最为核心的功能

## 配置路由

我们以 UPMS 的路由为例子，注意注释

```
spring:
  cloud:
    gateway:
      locator:
        enabled: true
      routes:
        #UPMS 模块
        - id: pig-upms      # 唯一的服务 ID
          uri: lb://pig-upms # 注册中心的服务名称，实现负载均衡
          predicates:
            - Path=/admin/**  #所有业务的请求前缀
          filters:
            # 限流配置
            - name: RequestRateLimiter      #限流策略
              args:
                key-resolver: '#{@remoteAddrKeyResolver}'
                redis-rate-limiter.replenishRate: 10
                redis-rate-limiter.burstCapacity: 20
            # 降级配置
            - name: Hystrix                  #断路器降级策略
              args:
                name: default
```

```
fallbackUri: 'forward:/fallback'
```

## pig 默认提供了全局的路由过滤器原理

PigRequestGlobalFilter,对全部的微服务提供了安全过滤（这个后边会讲）和全局 StripPrefix=1 配置，意味着你在使用 **Pig** 的时候，网关转发到业务模块时候会自动截取前缀，不用再每个微服务路由配置了 **StripPrefixFilter**

```
public class PigRequestGlobalFilter implements GlobalFilter, Ordered {
    private static final String HEADER_NAME = "X-Forwarded-Prefix";

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        // 1. 清洗请求头中 from 参数
        ServerHttpRequest request = exchange.getRequest().mutate()
            .headers(httpHeaders -> httpHeaders.remove(SecurityConstants.FROM))
            .build();

        // 2. 重写 StripPrefix
        addOriginalRequestUrl(exchange, request.getURI());
        String rawPath = request.getURI().getRawPath();
        String newPath = "/" + Arrays.stream(StringUtils.tokenizeToStringArray(rawPath, "/"))
            .skip(1L).collect(Collectors.joining("/"));
        ServerHttpRequest newRequest = request.mutate()
            .path(newPath)
            .build();
        exchange.getAttributes().put(GATEWAY_REQUEST_URL_ATTRIBUTE, newRequest.getURI());

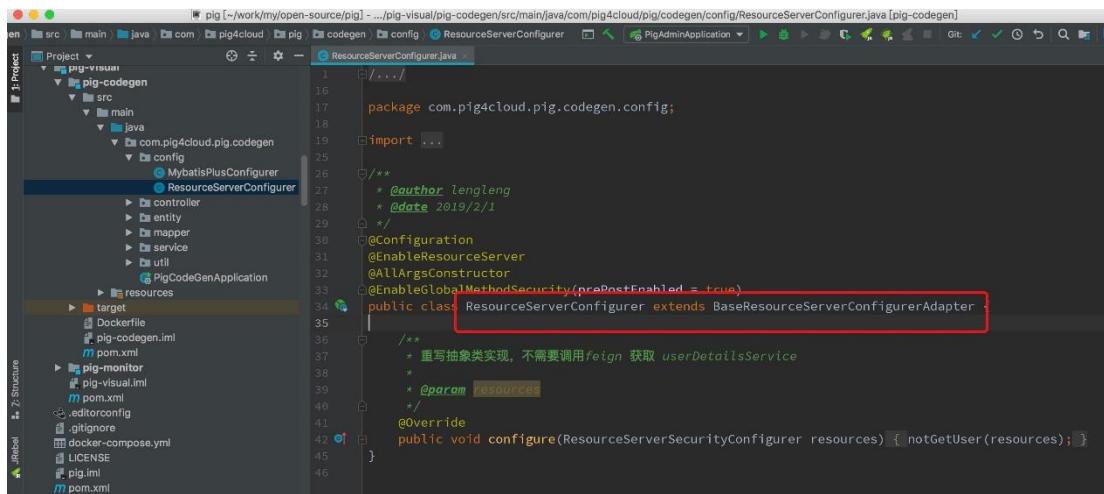
        return chain.filter(exchange.mutate()
            .request(newRequest.mutate()
                .build()).build());
    }

    @Override
    public int getOrder() {
```

```
        return -1000;
    }
}
```

# 认证授权

## 资源服务器配置



The screenshot shows a Java code editor with the file `ResourceServerConfigurer.java` open. The code is part of the `com.pig4cloud.pig.codegen.config` package. The class `ResourceServerConfigurer` extends `BaseResourceServerConfigurerAdapter`. A red box highlights the inheritance line: `public class ResourceServerConfigurer extends BaseResourceServerConfigurerAdapter {`. The code includes annotations like `@Configuration`, `@EnableResourceServer`, and `@EnableGlobalMethodSecurity(prePostEnabled = true)`. The code also contains a comment about overriding the `configure` method.

```
package com.pig4cloud.pig.codegen.config;
import ...;
/**
 * @author lengleng
 * @date 2019/2/1
 */
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ResourceServerConfigurer extends BaseResourceServerConfigurerAdapter {
    /**
     * 重写抽象类实现，不需要调用Feign 获取 userDetailsService
     *
     * @param resources
     */
    @Override
    public void configure(ResourceServerSecurityConfigurer resources) { not GetUser(resources); }
}
```

继承 `BaseResourceServerConfigurerAdapter` 即可实现接入 oauth2

重写 `configure` 方法的意义

`BaseResourceServerConfigurerAdapter` 提供了两种解析用户信息的方法

- 不获取用户详细 只有用户名

```
protected void not GetUser(ResourceServerSecurityConfigurer
resources) {
    DefaultAccessTokenConverter accessTokenConverter =
new DefaultAccessTokenConverter();
    DefaultUserAuthenticationConverter userTokenConvert
er = new DefaultUserAuthenticationConverter();
    accessTokenConverter.setUserTokenConverter(userToke
nConverter);

    remoteTokenServices.setRestTemplate(lbRestTemplate
());
```

```
        remoteTokenServices.setAccessTokenConverter(accessTokenConverter);
        resources.authenticationEntryPoint(resourceAuthExceptionEntryPoint)
            .accessDeniedHandler(pigAccessDeniedHandler)
            .tokenServices(remoteTokenServices);
    }
```

- 上下文中获取用户全部信息，两次调用 userDetailsService，影响性能

```
private void can GetUser(ResourceServerSecurityConfigurer resources) {
    DefaultAccessTokenConverter accessTokenConverter =
new DefaultAccessTokenConverter();
    DefaultUserAuthenticationConverter userTokenConverter =
new DefaultUserAuthenticationConverter();
    userTokenConverter.setUserDetailsService(userDetailsService);
    accessTokenConverter.setUserTokenConverter(userTokenConverter);

    remoteTokenServices.setRestTemplate(lbRestTemplate());
    remoteTokenServices.setAccessTokenConverter(accessTokenConverter);
    resources.authenticationEntryPoint(resourceAuthExceptionEntryPoint)
        .accessDeniedHandler(pigAccessDeniedHandler)
        .tokenServices(remoteTokenServices);
}
```

获取当前用户信息

获取当前用户

SecurityUtils.getUser()

```
public PigUser getUser(Authentication authentication) {
    Object principal = authentication.getPrincipal();
    if (principal instanceof PigUser) {
        return (PigUser) principal;
```

```
    }
    return null;
}
```

## 为什么 CodenGen 获取用户为空

当在 CodeGen 模块，通过 `SecurityUtils.getUser()` 的返回值始终为 `null`,因为 CodeGen 重写了资源服务的配置,不通过 pig 获取用户信息提高性能

### ResourceServerConfigurer

```
@Configuration
@EnableResourceServer
@AllArgsConstructor
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ResourceServerConfigurer extends BaseResourceServerConfigurerAdapter {

    /**
     * 重写抽象类实现，不需要调用 feign 获取 userDetailsService
     *
     * @param resources
     */
    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        not GetUser(resources);
    }
}
```

可以提供一个获取用户名的方法

```
public String getUsername(Authentication authentication) {
    Object principal = authentication.getPrincipal();
    return principal.toString();
}
```

资源服务器配置章节会详细讲重写和不重写两个的区别

API 直接对外暴露

## 现象

---

如果微服务接入了资源服务器，那么全部的资源被 spring security oauth 拦截，如果没有合法 token 直接会被拒绝。

如下图,提示如下错误。



## 服务暴露

---

- 直接在对应微服务模块配置

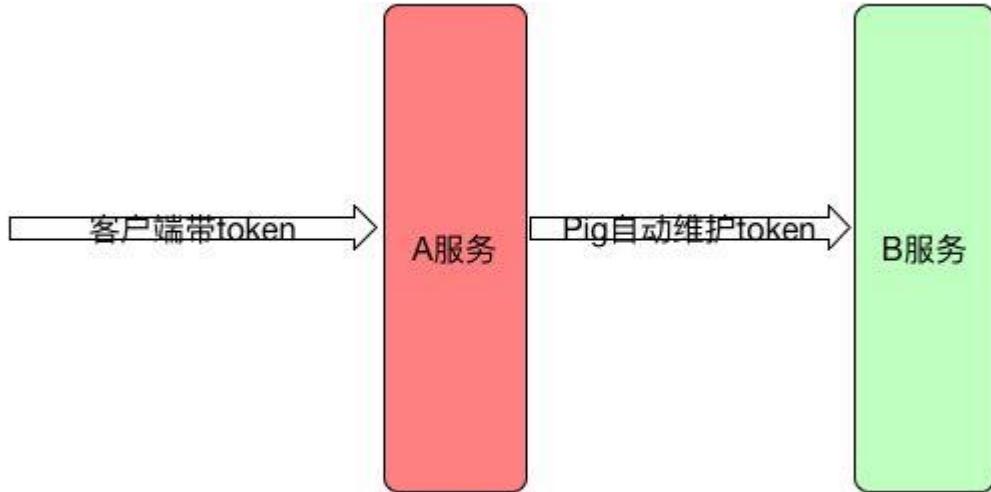
```
ignore:  
  urls:  
    - 目标借口的 Ant 表达式即可
```

```
1 security:
2   oauth2:
3     client:
4       client-id: ENC(imENT07M8bL0
5       client-secret: ENC(i3cDFhs2
6       scope: server
7
8     # 数据源
9     spring:
10    datasource:
11      type: com.zaxxer.hikari.Hikar
12      driver-class-name: com.mysql.
13      username: root
14      password: root
15      url: jdbc:mysql://pig-mysql:3
16
17    # 直接放行URL
18    ignore:
19      urls:
20        - /actuator/**
21        - /user/info/*
22        - /log/**
```

服务间鉴权及其 token 传递

## 客户端带 Token 情况

1. 如下图客户端携带 token 访问 A 服务。
2. A 服务通过 FeignClient 调用 B 服务获取相关依赖数据。
3. 所以只要带 token 访问 A 无论后边链路有多长 ABCD 都可以获取当前用户信息
4. 权限需要有这些整个链路接口的全部权限才能成功



## 核心代码

fein 拦截器将本服务的 token 通过 copyToken 的形式传递给下游服务

```

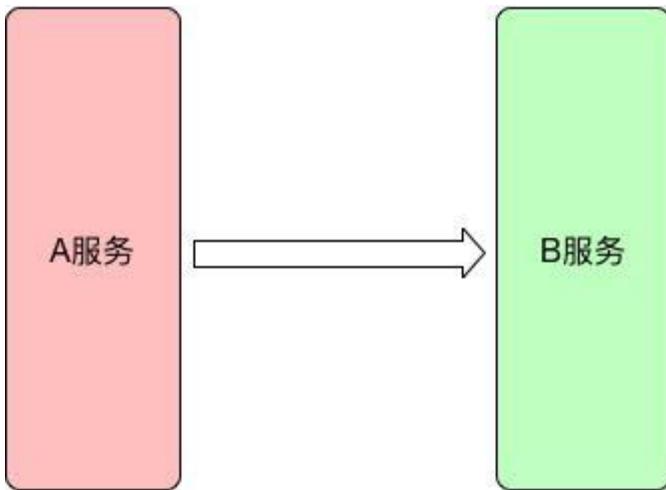
public class PigFeignClientInterceptor extends OAuth2FeignRequestInterceptor {

    @Override
    public void apply(RequestTemplate template) {
        Collection<String> fromHeader = template.headers().get(SecurityConstants.FROM);
        if (CollUtil.isNotEmpty(fromHeader) && fromHeader.contains(SecurityConstants.FROM_IN)) {
            return;
        }

        accessTokenContextRelay.copyToken();
        if (oAuth2ClientContext != null
            && oAuth2ClientContext.getAccessToken() != null) {
            super.apply(template);
        }
    }
}

```

无 token 请求，服务内部发起情况处理。



很多情况下，比如定时任务。A 服务并没有 token 去请求 B 服务，pig 也对这种情况进行了兼容。类似于 A 对外暴露 API，但是又安全限制。参考日志插入情况

- FeignClient 需要带一个请求 token, FROM\_IN 声明是内部调用

```
remoteLogService.saveLog(sysLog, SecurityConstants.FROM_IN);
```

- 参考 API 对外暴露章节对外暴露目标接口

```
ignore:  
  urls:  
    - 目标借口的 Ant 表达式即可
```

- 目标接口对内外调用进行限制

@Inner 注解，这样就避免接口对外暴露的安全问题。只能通过内部调用才能使用，浏览器不能直接访问该接口

```
@Inner  
@PostMapping  
public R save(@Valid @RequestBody SysLog sysLog) {  
    return new R<>(sysLogService.save(sysLog));  
}
```

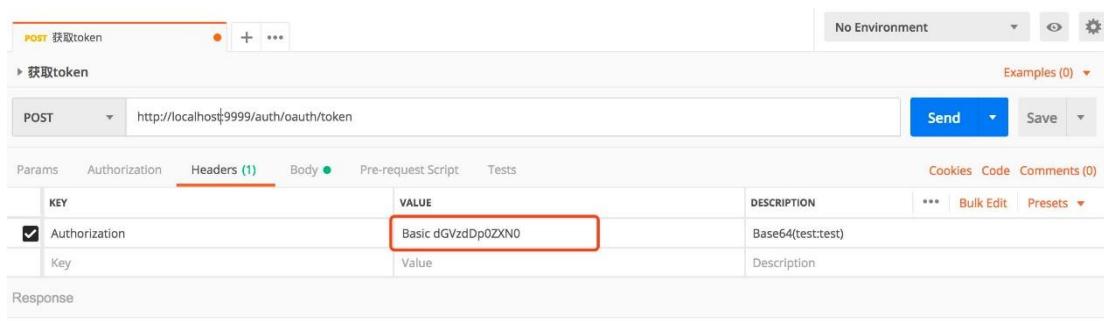
后边会专门讲@Inner 注解

## postman 等多终端接口调用

通过网关访问 auth-server 获取 access-token

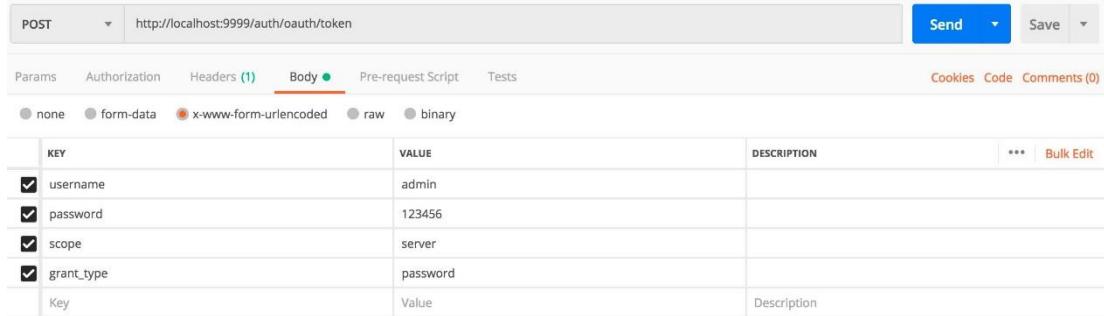
- 请求地址为 <http://pig:9999/auth/oauth/token>
- headr 参数为： Authorization Basic dGVzdDp0ZXN0， 这里为 Base64(test:test),只能使用 test 终端， 其他终端需要输验证码， 可以参考 验证码

处理章节



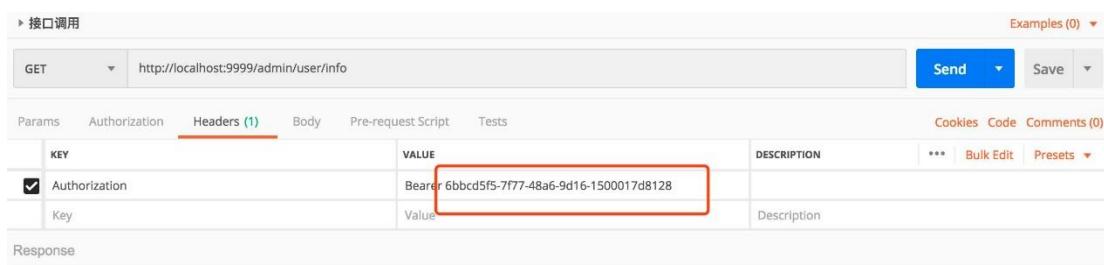
The screenshot shows the Postman interface with a POST request to 'http://localhost:9999/auth/oauth/token'. The 'Headers' tab is selected, showing a single header 'Authorization' with the value 'Basic dGVzdDp0ZXN0'. This value is highlighted with a red box.

- 参数如下



The screenshot shows the Postman interface with a POST request to 'http://localhost:9999/auth/oauth/token'. The 'Body' tab is selected, showing four form-data parameters: 'username' (value: admin), 'password' (value: 123456), 'scope' (value: server), and 'grant\_type' (value: password). The 'x-www-form-urlencoded' option is selected under the body type dropdown.

通过 access-token 访问受保护的资源



The screenshot shows the Postman interface with a GET request to 'http://localhost:9999/admin/user/info'. The 'Headers' tab is selected, showing an Authorization header with the value 'Bearer 6bbcd5f5-7f77-48a6-9d16-1500017d8128'. This value is highlighted with a red box.

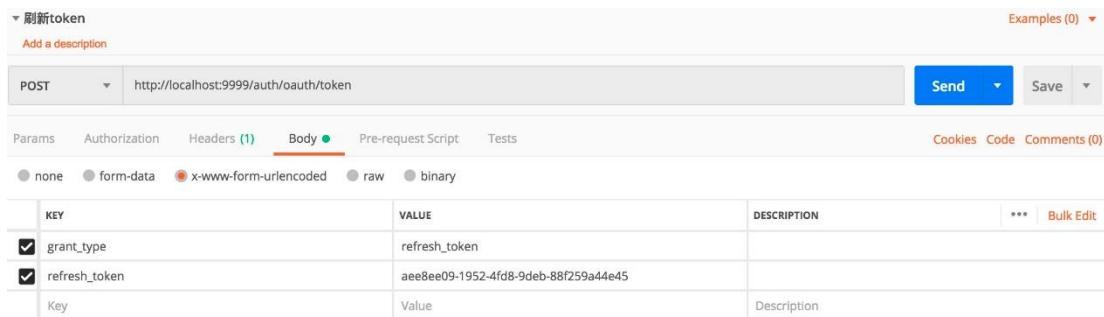
## 刷新 token

- 请求接口通获取 token 接口一致，header 参数一致



The screenshot shows the Postman interface with a request to '刷新token' (Refresh token). The method is POST and the URL is 'http://localhost:9999/auth/oauth/token'. The 'Headers' tab is active, showing one header: 'Authorization' with the value 'Basic dGVzdDp0ZXN0'. The 'Body' tab is also present.

- 注意 grant\_type refresh\_token



The screenshot shows the Postman interface with a request to '刷新token' (Refresh token). The method is POST and the URL is 'http://localhost:9999/auth/oauth/token'. The 'Body' tab is active, showing two parameters: 'grant\_type' with value 'refresh\_token' and 'refresh\_token' with value 'aeef8ee0-1952-4fd8-9deb-88f259a44e45'. The 'Headers' tab is also present.

## pig 生成 token 详解

### pig 的 Oauth2.0 认证流程详解

#### Spring Security OAuth 核心类图解析

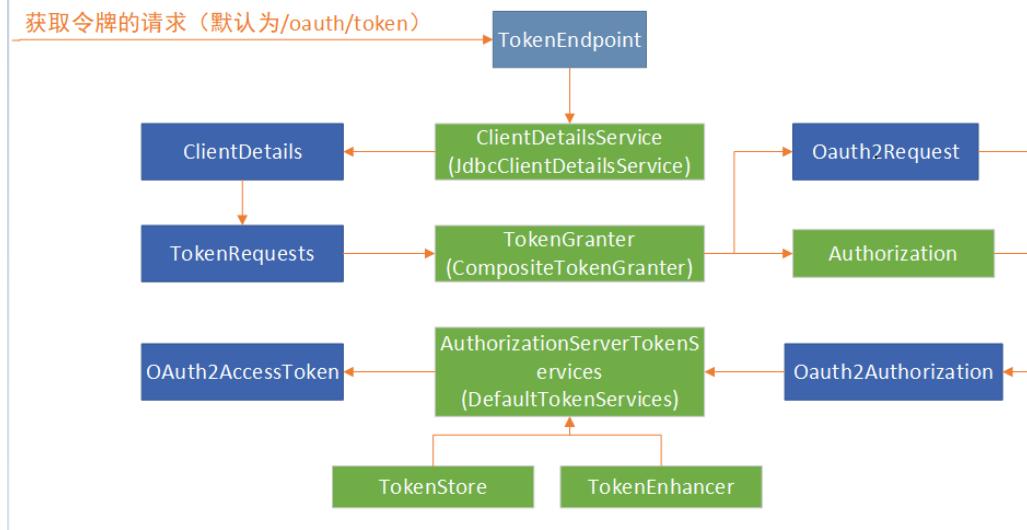
关于 Oauth2 是什么以及 Oauth2 的四种授权模式请移步 [Oauth2 官网](#)。

下面简单介绍一下关于 Spring Security OAuth 基本的原理。这也是理解 pigx 的第一步。

下面这张图涉及到了 Spring OAuth 的一些核心类和接口。

不多说，直接上图。

# Spring Security Oauth核心源码



上图蓝色的方块代表执行过程中调用的具体的类，绿色的方块代表整个执行流程中调用的类，绿色的括号中代表的是该接口调用的具体的实现类。

整个流程的入口点是在 **TokenEndpoint**，由它来处理获取令牌的请求，获取令牌的请求默认是\*\*/oauth/token\*\*这个路径。

- 当 TokenEndpoint 收到请求时，它首先会调用 ClientDetailsService, ClientDetailsService 从名字上看就很可以知道是一个类似于 UserDetailsService 的接口，只不过 UserDetailsService 读取的是用户的信息，而 ClientDetailsService 读取的是第三方应用的信息。
- pigx 会在登录请求头中带上 Client 的信息，而这个类就可以做到根据 ClientId 读取相应的配置信息。而 ClientDetailsService 读取到的信息都会封装到 ClientDetails 这个对象中。
- 同时，TokenEndpoint 还会创建一个 TokenRequests 的对象，这个对象中封装了除了第三方应用以外的其他信息。比如说 grant\_type, scope, username, password(限密码模式) 等等信息，而这些信息都是封装在

`TokenRequests` 里面的。同时，`ClientDetails` 也会被放到 `TokenRequests` 中，因为第三方应用的信息也是令牌请求的一部分。

- 之后利用 `TokenRequests` 去调用一个叫做 `TokenGranter` 的令牌授权者的接口，这个接口其实是对四种不同的授权模式进行的一个封装。在这个接口里，它会根据请求传递过来的 `grant_type` 去挑一个具体的实现来执行令牌生成的逻辑。
- 不论采用哪种方式进行令牌的生成，在这个生成的过程中都会产生两个对象，一个是 `OAuth2Request`，这个对象实际上是之前的 `ClientDetails` 和 `TokenRequests` 这两个对象的一个整合。另一个 `Authorization` 封装的实际上是当前授权用户的一些信息，也就是谁在进行授权行为，`Authorization` 里封装的就是谁的信息。这里的用户信息是通过 `UserDetailsService` 进行读取的。
- `OAuth2Request` 和 `Authorization` 这两个对象组合起来，会形成一个 `OAuth2Authorization` 对象，而这个最终产生的对象它的里面就包含了当前是哪个第三方应用在请求哪个用户以哪种授权模式（包括授权过程中的一些其他参数）进行授权，也就是这个对象会汇总之前的几个对象的信息都会封装到 `OAuth2Authorization` 这个对象中。
- 然后这个对象会传递到一个叫做 `AuthorizationServerTokenServices` 的接口的实现类，它拿到 `OAuth2Authorization` 中所有的信息之后最终会生成一个 `OAuth2` 的令牌 `OAuth2AccessToken`。

**Tips:** 个性化 token 生成

`AuthorizationServerTokenServices` 的接口的默认实现 `DefaultTokenServices` 中包含着其他两个接口的引用，`TokenStore` 是用来定制 token 存储策略的，`pigx`

用它实现了往 redis 里存放 token, TokenEnhancer 是 token 的增强器,pigx 用它实现了返回的 token 的信息的增强。

Spring Security OAuth 的令牌生成过程——以 pigx 的登录过程为例

## 前期准备

光说不练假把式，下面我们结合 pigx 项目代码，来验证一下上面的过程。目前代码采用的是 2018 年 9 月 1 日最新的开发分支上的代码做演示，选择 1.5.0 的稳定版来复现以下的过程问题应该也不大，这部分逻辑最近几个版本也没啥变化。

首先启动核心的五个工程：注册中心，配置中心，认证中心，网关以及统一用户管理中心，同时启动前端工程以**避免跨域问题**。同时，为了避免影响测试结果请先在 redis-cli 上执行 flushall 或者 flushdb 清空 redis。

个人建议萌新可以通过访问 <http://127.0.0.1:9999/swagger-ui.html>,通过 swagger 上面的 Authorization 按钮进行登录，我这里选择和作者视频里相同的 curl 的方式进行登录。

## 开始学习

### 客户端选择

我选择的应用是 test,因为这个应用可以忽略验证码，关于这一块的配置可以参考 pigx-gateway-dev.yml 这个配置文件的 ignore.clients 属性，它可以接收一个想要忽略验证码的客户端的列表。至于为什么配置了这里的属性就可以忽略验证码也很简单。

网关工程有一个 `FilterIgnorePropertiesConfig` 类，这个类当配置文件里的 `ignore` 属性不为空时会生效。而验证码过滤器会对 `FilterIgnorePropertiesConfig` 中配置的客户端进行放行。如下所示：

```
// 终端设置不校验， 直接向下执行(1. 从请求参数中获取 2. 从 header 取)
String clientId = request.getQueryParams().getFirst("client_id");
if (StrUtil.isNotBlank(clientId)) {
    if (filterIgnorePropertiesConfig.getClients().contain(clientId)) {
        return chain.filter(exchange);
    }
}
```

能够获取到 `token` 的姿势有很多，我这里就选择一种类似 `pigx` 前端工程登录的方式。

## 前端密码加解密讲解

`pigx` 大致的请求流程就是前端通过 `vue-router`（模拟 `nginx`）发送请求到后台网关，网关再根据配置的路由规则转发到各个微服务上。

根据 `pigx-gateway-dev.yml` 上的配置，可知经过认证中心的请求都需要经过两个过滤器，一个是验证码的处理，一个是将加密过的密码解密的过滤器。

```
routes:
# 认证中心
- id: pigx-auth
  uri: lb://pigx-auth
predicates:
- Path=/auth/**
filters:
    # 验证码处理
- ImageCodeGatewayFilter
    # 前端密码解密
- PasswordDecoderFilter
- StripPrefix=1
```

验证码的过滤器已经被我们干掉了，密码解密的过滤器我这边不想处理，就直接网上搜个在线加密的链接手动加密了,我用的是这个[在线 AES 加密解密、AES 在线加密解密、AES encryption and decryption](#)。

AES 作为对称加密的方式，前后端的加解密方式肯定是一致的，我们先看看后端的解密逻辑，之后再看看前端的加密逻辑做验证。后端的解密逻辑位于

PasswordEncoderFilter 这个类中，解密的代码不长，如下：

```
/*
 * AES/CBC/NoPadding 要求
 * 密钥必须是 16 字节长度的; Initialization vector (IV) 必须是 16 字节
 * 待加密内容的字节长度必须是 16 的倍数，如果不是 16 的倍数，就会出如下异常：
 * javax.crypto.IllegalBlockSizeException: Input length not multiple of 16 bytes
 *
 * 由于固定了位数，所以对于被加密数据有中文的，加、解密不完整
 *
 * 可以看到，在原始数据长度为 16 的整数 n 倍时，假如原始数据长度等于 16*n，则使用 NoPadding 时加密后数据长度等于 16*n，
 * 其它情况下加密数据长度等于 16*(n+1)。在不足 16 的整数倍的情况下，假如原始数据长度等于 16*n+m[其中 m 小于 16]，除了 NoPadding 填充之外的任何方式，加密数据长度都等于 16*(n+1)。
 */

private static final String PASSWORD = "password";
private static final String KEY_ALGORITHM = "AES";
private static final String DEFAULT_CIPHER_ALGORITHM = "AES/CBC/NO Padding";
@Value("${security.encode.key:1234567812345678}")
private String encodeKey;

private static String decryptAES(String data, String pass) throws Exception {
    Cipher cipher = Cipher.getInstance(DEFAULT_CIPHER_ALGORITHM);
    SecretKeySpec keyspec = new SecretKeySpec(pass.getBytes(), KEY_ALGORITHM);
    IvParameterSpec ivspec = new IvParameterSpec(pass.getBytes());
    cipher.init(Cipher.DECRYPT_MODE, keyspec, ivspec);
```

```
    byte[] result = cipher.doFinal(Base64.decode(data.getBytes(Ch
rsetUtil.UTF_8)));
    return new String(result, CharsetUtil.UTF_8);
}
```

Value 注解是将配置文件中的 `security.encode.key` 的值作为参数注入，当这个参数不存在时，就会使用冒号后面的默认值 `1234567812345678`，当然，在 `pigx` 的配置文件中这个参数值是肯定存在的，正是 `pigxpigxpigxpigx`。这段解密逻辑说的就是采用 AES 的 CBC 加密方式，填充方式为零填充，密码和偏移量都是卸载配置文件中的 `pigxpigxpigxpigx`。

接着我们去找一下前端的加密逻辑验证一下我们的判断，前端的加密逻辑位于 `util/util.js` 中，核心的加密代码如下：

```
/**
 * 加密处理
 */
export const encryption = (params) => {
  let {
    data,
    type,
    param,
    key
  } = params
  const result = JSON.parse(JSON.stringify(data))
  if (type === 'Base64') {
    param.forEach(ele => {
      result[ele] = btoa(result[ele])
    })
  } else {
    param.forEach(ele => {
      var data = result[ele]
      key = CryptoJS.enc.Latin1.parse(key)
      var iv = key
      // 加密
      var encrypted = CryptoJS.AES.encrypt(
        data,
        key, {
          iv: iv,
          mode: CryptoJS.mode.CBC,
```

```

        padding: CryptoJS.pad.ZeroPadding
    })
    result[ele] = encrypted.toString()
})
}

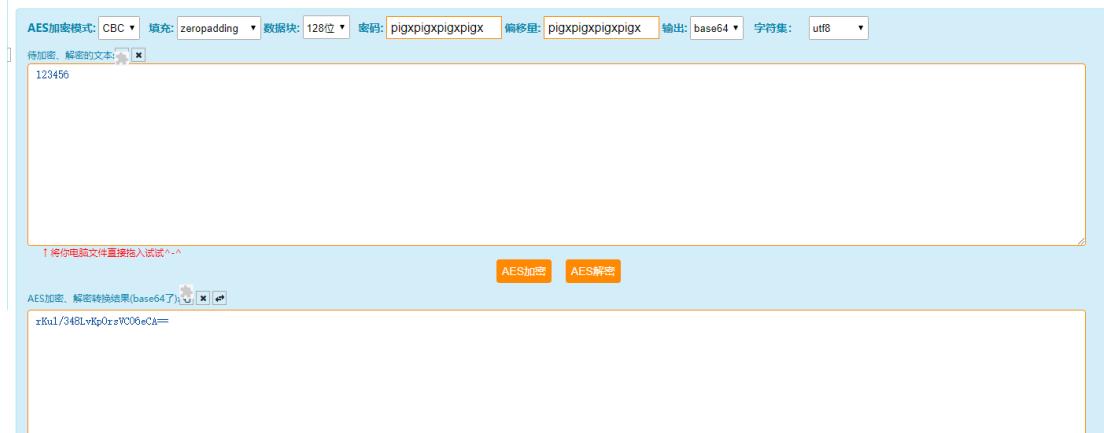
return result
}

```

我们可以很显然地看到，加密采用的正式无填充的 CBC 模式，偏移量就是传入的加密密钥，零填充，这个和后端的代码是一致的，所以我们可以大胆地在上面地网站上填入参数了，填写的参数如下：

#### 在线AES加密解密、AES在线加密解密、AES encryption and decryption

AES,高级加密标准（英语：Advanced Encryption Standard，缩写：AES），在密码学中又称Rijndael加密法，是美国联邦政府采用的一种区块加密标准。这个标准用来替代原先的DES，已经被多方分析且广为全世界所使用。严格地说，AES和Rijndael加密法并不完全一样（虽然在实际应用中二者可以互换），因为Rijndael加密法可以支持更大范围的区块和密钥长度：AES的区块长度固定为128比特，密钥长度则可以是128, 192或256比特；而Rijndael使用的密钥和区块长度可以是32位的整数倍，以128位为下限，256比特为上限。包括AES-ECB,AES-CBC,AES-CTR,AES-OFB,AES-CFB



好了，密文：rKu1/348LvKp0rsVC06eCA==我们拿到了。

#### 构造请求参数

接着我们开始吧。

```
curl -H "Authorization:Basic dGVzdDp0ZXN0" -d "username=admin&password=rKu1/348LvKp0rsVC06eCA==&grant_type=password&scope=server"
http://localhost:8000/auth/oauth/token
```

通过 curl 构造以上的链接，这个链接中包含着请求头的信息，请求头是一个

"Basic"加一个空格加"clientId:clientSecret"base64 化的一个 **Authorization** 字段，

请求的参数里包含了 grant\_type 和 scope 以及在 password 模式下必须的

username 和 password 字段。顺带一提，如果是 windows 中文语言的系统建议执行命令：

```
chcp 65001
```

将你的 shell 临时的更换为 UTF-8 的编码避免中文乱码的问题，虽然生成 token 的过程中不涉及中文化的操作，但如果后期扩展了中文化可以避免问题。

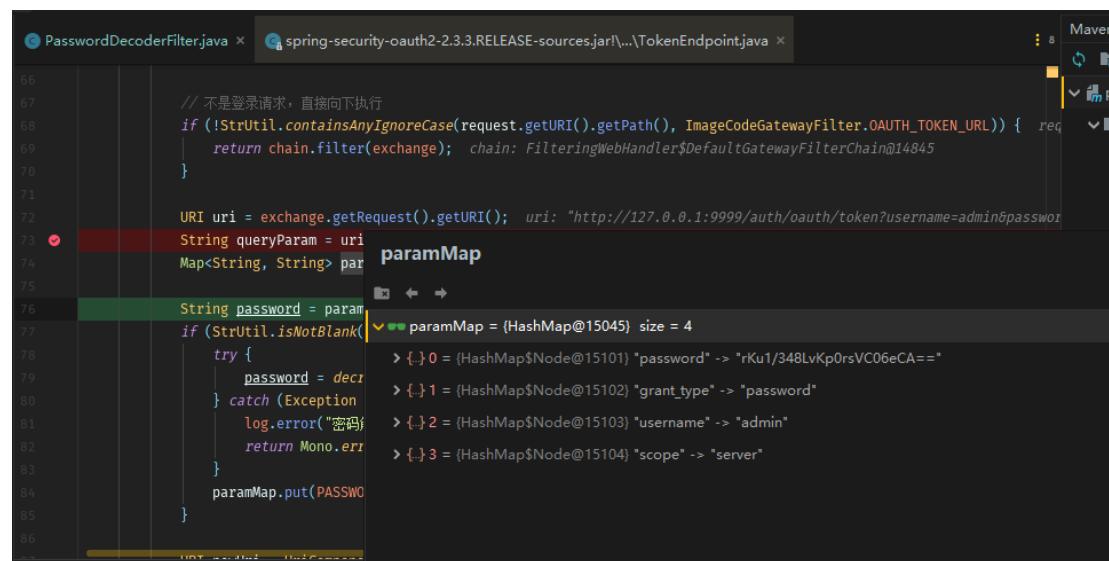
下面的是一个标准的 POST 请求并且在 URL 中携带参数的请求，但是这个请求不符合我们这边测试的要求，原因看下面的注意事项。

```
curl -H "Authorization:Basic dGVzdDp0ZXN0" -X POST http://localhost:8000/auth/oauth/token?username=admin&password=rKu1/348LvKp0rsVC06eCA==&grant_type=password&scope=server
```

所以我们把它改造一下。

```
curl -H "Authorization:Basic dGVzdDp0ZXN0" -X POST http://localhost:8000/auth/oauth/token?username=admin\&password=rKu1/348LvKp0rsVC06eCA%3D%3D\&grant_type=password\&scope=server
```

回车以后我们可以看到首先会经过网关的密码解密过滤器，并且参数经过我们的一通改造之后已经可以获取到正确的值了。



```
>PasswordDecoderFilter.java x spring-security-oauth2-2.3.3.RELEASE-sources.jar!...\\TokenEndpoint.java x
56
57
58     // 不是登录请求，直接向下执行
59     if (!StrUtil.containsIgnoreCase(request.getURI().getPath(), ImageCodeGatewayFilter.OAUTH_TOKEN_URL)) {
60         return chain.filter(exchange);
61     }
62
63     URI uri = exchange.getRequest().getURI();
64     String queryParam = uri
65     Map<String, String> paramMap
66
67     String password = paramMap.get("password");
68     if (StrUtil.isNotBlank(password)) {
69         try {
70             password = decoder.decode(password);
71         } catch (Exception e) {
72             log.error("密码解密失败");
73             return Mono.error(e);
74         }
75         paramMap.put(PASSWO
76     }
77
78     return chain.filter(exchange);
79 }
80
81
82
83
84
85
86
```

注意：

这个 url.getRawQuery() 方法会获取拼接到请求的 URL 后面的参数，这也是为什

么我们之前构造的 curl 格式不是标准的把数据放入 POST 请求的请求体中的原

因，标准的做法如下：

```
curl -H "Authorization:Basic dGVzdDp0ZXN0" -d "username=admin&password=rKu1/348LvKp0rsVC06eCA==&grant_type=password&scope=server"
http://localhost:8000/auth/oauth/token
```

但是这种方式会有问题，在这套密码解密过滤器的机制下将会获取不到任何的参数。而会出现这个问题的原因也正是因为这 73 行的代码。

```
63 public GatewayFilter apply(Object config) {
64     return (exchange, chain) -> {
65         exchange: DefaultServerWebExchange@11647 chain: FilteringWebHandler$DefaultGatewayFilter
66         ServerHttpRequest request = exchange.getRequest(); request: ReactorServerHttpRequest@11649
67
68         // 不是登录请求，直接向下执行
69         if (!StrUtil.containsAnyIgnoreCase(request.getURI().getPath(), ImageCodeGatewayFilter.OAUTH_TOKEN_URL)) {
70             return chain.filter(exchange); chain: FilteringWebHandler$DefaultGatewayFilterChain@11648
71         }
72
73         URI uri = exchange.getRequest().getURI(); uri: "http://127.0.0.1:9999/auth/oauth/token" exchange: DefaultServerWe
74         String queryParam = uri.getRawQuery(); uri: "http://127.0.0.1:9999/auth/oauth/token"
75         Map<String, String> paramMap = HttpUtil.decodeParamMap(queryParam, CharsetUtil.UTF_8);
76
77         String password = paramMap.get(PASSWORD);
78         if (StrUtil.isNotBlank(password)) {
79             try {
80                 password = decryptAES(password, encodeKey);
81             } catch (Exception e) {
82                 log.error("密码解密失败:{}", password);
83             }
84         }
85     }
86 }
```

不过 Spring 的 OAuth2.0 本身是支持把数据放入 POST 请求体中的这种方式的。

### Tips:

URL 中用于拼接多个参数的符号:"&",在 shell 脚本中有特殊的意义（以 daemon 运行）,所以要在"&"前加上反斜杠"\\"转义一下。

### Tips:

URL 中"="这个符号具有特殊的含义,可能在服务器端无法获得正确的参数值, 所以我们也要用"%3D"转义一下。

继续我们刚才的过程, 可以看到在密码解密过滤器接收到参数之后, 就很简单了, 整个密码解密过滤器的作用就是对登录请求中发送过来的加密密码进行解密的操作。我们直接看解密的结果。

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
```

解密的结果对于不足 16 位的密码会填充空格到 16 位，trim 之后我们就可以看到得到我们真正想要的密码"123456"了，这也侧面验证了我们之前生成的密码策略是正确的。

## 认证过程详解

经过上面的一通操作，我们已经拿到了获取 token 的一些必要的请求了。

clientId,clientSecret,grant\_type,username,password,scope,终于可以带着我们的参数深入源码啦！

这里结合上文提到的核心类图来看效果更好

上文提过，OAuth2.0 的认证的入口点位于 TokenEndPoint。我们也可以看到，代码确实已经进来了。

```
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
```

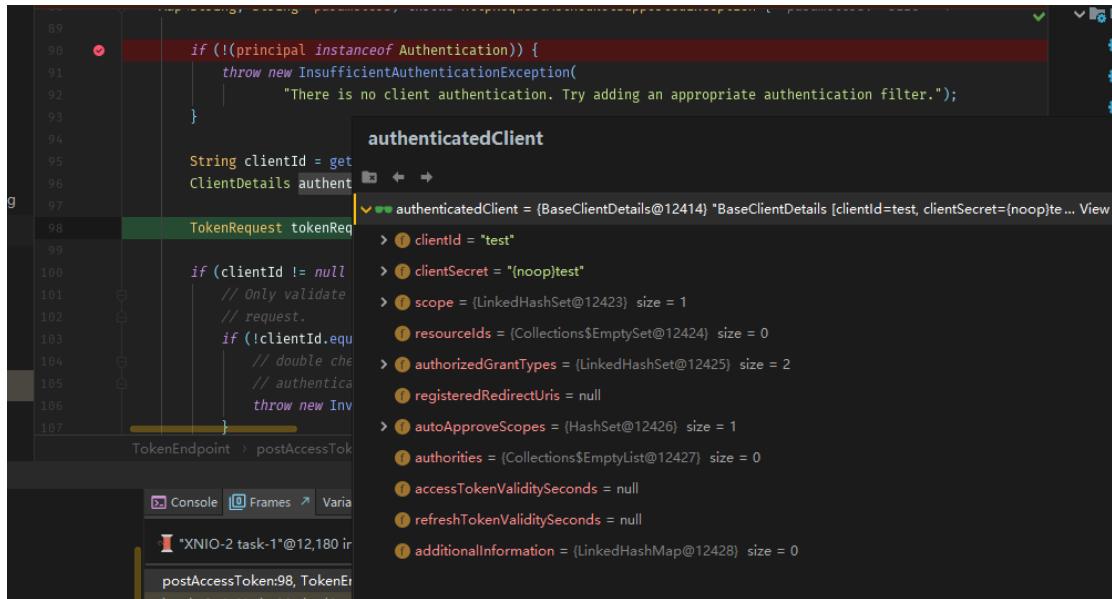
我们可以看到这个类上有一个@RequestMapping注解，它来处理/oauth/token的POST请求。

1. 进来之后的第一步，就是在代码的95行，获取请求头中的clientId。
2. 然后在96行调用

```
getClientDetailsService().loadClientByClientId(clientId)方法获取整个第三方应用的详细配置。
```

扩展：

第三方应用有非常丰富的配置项，如图所示：



具体的参数的意义可以看[spring-oauth-server 数据库表说明](#)

3. 在拿到客户端的信息之后在代码的98行通过传递进来的参数和查询出来的第三方应用信息构建TokenRequest。

创建TokenRequest的代码很简单，如下：

```
public TokenRequest createTokenRequest(Map<String, String> request
Parameters, ClientDetails authenticatedClient) {
```

```
        String clientId = requestParameters.get(OAuth2Utils.CLIENT_ID);
        if (clientId == null) {
            // if the clientId wasn't passed in in the map, we add pull
            // it from the authenticated client object
            clientId = authenticatedClient.getClientId();
        }
        else {
            // otherwise, make sure that they match
            if (!clientId.equals(authenticatedClient.getClientId())) {
                throw new InvalidClientException("Given client ID does
not match authenticated client");
            }
        }
        String grantType = requestParameters.get(OAuth2Utils.GRANT_TYPE);

        Set<String> scopes = extractScopes(requestParameters, clientId);
        TokenRequest tokenRequest = new TokenRequest(requestParameters,
clientId, scopes, grantType);

        return tokenRequest;
    }
```

所以其实它就干了一件事，校验传递进来 clientId 和查询出来的 clientId,如果匹配的话，就根据之前传递进来的 clientId 和和查询出来的第三方应用构建 TokenRequest。

然后我们就拿到 TokenRequest 了，后面的代码很简单了：

```

    if (clientId != null && !clientId.equals("")) {
        // Only validate the client details if a client authenticated during this
        // request.
        if (!clientId.equals(tokenRequest.getClientId())) {
            // double check to make sure that the client ID in the token request is the same as that in the
            // authenticated client
            throw new InvalidClientException("Given client ID does not match authenticated client");
        }
    }
    if (authenticatedClient != null) {
        oAuth2RequestValidator.validateScope(tokenRequest, authenticatedClient);
    }
    if (!StringUtils.hasText(tokenRequest.getGrantType())) {
        throw new InvalidRequestException("Missing grant type");
    }
    if (tokenRequest.getGrantType().equals("implicit")) {
        throw new InvalidGrantException("Implicit grant type not supported from token endpoint");
    }

    if (isAuthCodeRequest(parameters)) {
        // The scope was requested or determined during the authorization step
        if (!tokenRequest.getScope().isEmpty()) {
            logger.debug("Clearing scope of incoming token request");
            tokenRequest.setScope(Collections.<-> emptySet());
        }
    }

    if (isRefreshTokenRequest(parameters)) {
        // A refresh token has its own default scopes, so we should ignore any added by the factory here.
        tokenRequest.setScope(OAuth2Utils.parseParameterList(parameters.get(OAuth2Utils.SCOPE)));
    }

    OAuth2AccessToken token = getTokenGranter().grant(tokenRequest.getGrantType(), tokenRequest);
    if (token == null) {
        throw new UnsupportedGrantTypeException("Unsupported grant type: " + tokenRequest.getGrantType());
    }

    return getResponse(token);
}

```

无非就是对下面这些参数的校验：

- clientId:是否有值，值是否和查询结果匹配
- scope:请求的一些授权内容，所请求的授权必须是第三方应用可以发送的授权集合的子集，否则无法通过校验)
- grant\_type:必须显式指定按照哪种授权模式获取令牌
- 判断传递的授权模式是否是简化模式，如果是简化模式也会抛异常。因为简化模式其实是对授权码模式的一种简化:在用户的第一步的授权行为的时候就直接返回令牌,所以是不会有调用请求令牌服务的机会的

- 判断是不是授权码模式,因为授权码模式包含两个步骤, 在授权码模式中发出的令牌中拥有的权限不是由发令牌的请求决定的, 而是在发令牌之前的授权的请求里就已经决定好了。因此它会对请求过来的 scope 进行置空操作, 然后根据之前发出的授权码里的权限重新设置你的 scope, 因此它根本不会使用请求令牌的这个请求中携带的 scope 参数。
- 之后判断是不是刷新令牌的请求, 应为刷新令牌的请求有自己的 scope, 所以也会进行重新设置 scope 的操作。

经过一系列的校验之后, 最终 TokenRequest 会在 132 行传递给 TokenGranter, 然后由 granter 产生最终的 accessToken。之后直接将 accessToken 写入响应里就可以了。

TokenGranter 中总共封装了四种授权模式加一个刷新令牌的操作, 我们看看其中的一些细节。

```

1 * @author Dave Sayer
2 *
3 */
4 public class CompositeTokenGranter implements TokenGranter {
5     private final List<TokenGranter> tokenGranters;
6
7     public CompositeTokenGranter(List<TokenGranter> tokenGranters) {
8         this.tokenGranters = new ArrayList<TokenGranter>();
9         for (TokenGranter grant : tokenGranters) {
10             if (grant != null) {
11                 addTokenGranter(grant);
12             }
13         }
14     }
15
16     public OAuth2AccessToken grant(String grantType) {
17         for (TokenGranter grant : tokenGranters) {
18             OAuth2AccessToken grant = grant.grant(grantType);
19             if (grant != null) {
20                 return grant;
21             }
22         }
23         return null;
24     }
25
26     public void addTokenGranter(TokenGranter tokenGranter) {
27         if (tokenGranter == null) {
28             throw new IllegalArgumentException("tokenGranter must not be null");
29         }
30         tokenGranters.add(tokenGranter);
31     }
32 }

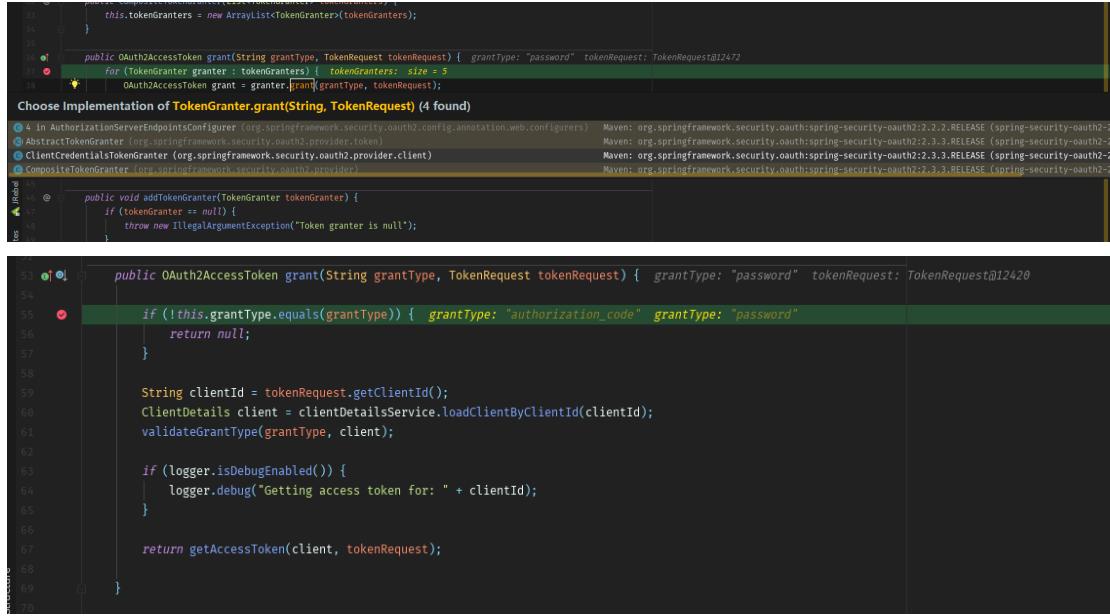
```

CompositeTokenGranter 中有一个集合, 这个集合里封装着的就是五个会产生令牌的操作。

它会对遍历这五种情况，并根据之前请求中携带的 grant\_type 在五种情况中挑一

种进行最终的 accessToken 的生成。

然后我们看这个代码的第 38 行的具体的 grant 方法。



The screenshot shows the Java code for the `grant` method in the `AbstractTokenGranter` class. The code is annotated with Maven dependency information for different versions of Spring Security OAuth 2.2.x.

```
public OAuth2AccessToken grant(String grantType, TokenRequest tokenRequest) { grantType: "password" tokenRequest: TokenRequest@12472
    for (TokenGranter granter : tokenGranters) { tokenGranters: size = 5
        OAuth2AccessToken grant = granter.grant(grantType, tokenRequest);
    }
}

Choose Implementation of TokenGranter.grant(String, TokenRequest) (4 found)
4 in AuthorizationServerEndpointsConfigurer (org.springframework.security.oauth2.config.annotation.web.configurers) Maven: org.springframework.security.oauth:spring-security-oauth2:2.2.2.RELEASE (spring-security-oauth2-2.2.2.RELEASE)
AbstractTokenGranter (org.springframework.security.oauth2.provider.token) Maven: org.springframework.security.oauth:spring-security-oauth2:2.3.3.RELEASE (spring-security-oauth2-2.3.3.RELEASE)
ClientCredentialsTokenGranter (org.springframework.security.oauth2.provider.client) Maven: org.springframework.security.oauth:spring-security-oauth2:2.3.3.RELEASE (spring-security-oauth2-2.3.3.RELEASE)
CompositeTokenGranter (org.springframework.security.oauth2.provider) Maven: org.springframework.security.oauth:spring-security-oauth2:2.3.3.RELEASE (spring-security-oauth2-2.3.3.RELEASE)

public void addTokenGranter(TokenGranter tokenGranter) {
    if (tokenGranter == null) {
        throw new IllegalArgumentException("Token granter is null");
    }
}

public OAuth2AccessToken grant(String grantType, TokenRequest tokenRequest) { grantType: "password" tokenRequest: TokenRequest@12420
    if (!this.grantType.equals(grantType)) { grantType: "authorization_code" grantType: "password"
        return null;
    }

    String clientId = tokenRequest.getClientId();
    ClientDetails client = clientDetailsService.loadClientByClientId(clientId);
    validateGrantType(grantType, client);

    if (logger.isDebugEnabled()) {
        logger.debug("Getting access token for: " + clientId);
    }

    return getAccessToken(client, tokenRequest);
}
```

首先在

`org.springframework.security.oauth2.provider.token.AbstractTokenGranter` 中判

断当前携带的授权类型和这个类所支持的授权类型是否匹配，如果不匹配就返回空

值，如果匹配的话就进行令牌的生成操作。

59 到第 63 行是重新获取一下 `clientId` 和客户端信息跟授权类型再做一个校验,67

行的 `getAccessToken` 方法会产生最终的一个令牌。

这个方法也非常简单：

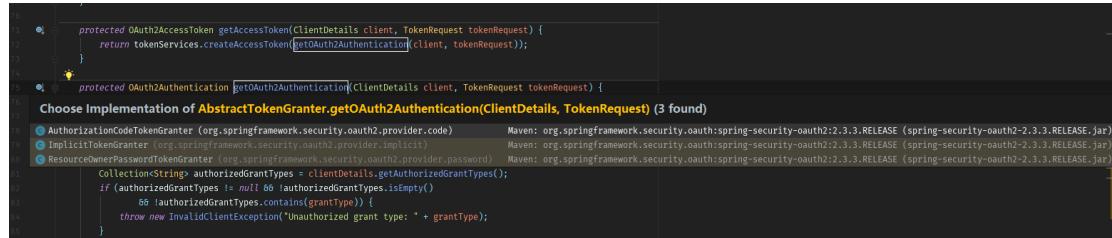
```
protected OAuth2AccessToken getAccessToken(ClientDetails client, TokenRequest tokenRequest) {
    return tokenServices.createAccessToken(getOAuth2Authentication(client, tokenRequest));
}
```

它实际上就是对 `tokenServices` 的一个调用，而 `tokenServices` 其实就是从 37 行我

们可以看到其实质上就是 `AuthorizationServerTokenServices`。这个类要想创建

accessToken 需要一个 OAuth2Authentication 对象，所以 createAccessToken 中包含了一个方法 `getOAuth2Authentication`。

这个方法不同的授权模式会有不同的实现。



在 **Spring Security OAuth 核心类图解析** 中我们已经知道最终产生的 `Oauth2Authorization` 包含两部分信息，一部分是请求中的一些信息，另一部分是根据请求获取的授权用户的信息。而在不同的授权模式下获取授权用户的信息的方式是不同的，比如说 `pigx` 所使用的密码模式就是使用请求中携带的用户名和密码来获取当前授权用户中的授权信息，而在授权码模式的两个步骤中是根据第一步发出授权码的同时会记录相关用户的信息，之后对第二步进行授权的时候根据第三方应用请求过来的授权码再读取该授权码对应的用户信息。所以 `getOAuth2Authentication` 对于不同的授权类型有不同的实现。

我们以 `pigx` 所使用的密码模式继续下面的流程。密码模式对应的是 `org.springframework.security.oauth2.provider.password.ResourceOwnerPasswordTokenGranter`

```
    @Override
    protected OAuth2Authentication getOAuth2Authentication(ClientDetails client, TokenRequest tokenRequest) { client: "BaseClientDetails [clientId=test, clientSecret=[noop]test, scope=[server], resourceIds=[...]
    Map<String, String> parameters = new LinkedHashMap<String, String>();
    String username = parameters.get("username");
    String password = parameters.get("password");
    // Protect from downstream leaks of password
    parameters.remove( key: "password");
    Authentication userAuth = new UsernamePasswordAuthenticationToken(username, password);
    ((AbstractAuthenticationToken) userAuth).setDetails(parameters);
    try {
        userAuth = authenticationManager.authenticate(userAuth);
    }
    catch (AccountStatusException ase) {
        //covers expired, locked, disabled cases (mentioned in section 5.2, draft 31)
        throw new InvalidGrantException(ase.getMessage());
    }
    catch (BadCredentialsException e) {
        // If the username/password are wrong the spec says we should send 400/invalid grant
        throw new InvalidGrantException(e.getMessage());
    }
    if (userAuth == null || !userAuth.isAuthenticated()) {
        throw new InvalidGrantException("Could not authenticate user: " + username);
    }
    OAuth2Request storedOAuth2Request = getRequestFactory().createOAuth2Request(client, tokenRequest);
    return new OAuth2Authentication(storedOAuth2Request, userAuth);
}
```

而这个方法我们可以看到它其实就是根据所请求的用户名和密码去创建

UsernamePasswordAuthenticationToken, 然后传递给 authenticationManager 做认证, 在这个认证过程中它会去调用

com.pig4cloud.pigx.common.security.service.PigxUserDetailsService  
Impl 的 loadUserByUsername 方法，根据用户名和密码去读取用户的信息，之后  
我们其实就已经拿到 Authorization 的信息，而 Oauth2Request 根据第 85 行我们  
可以知道是根据传进来的第三方应用详情和 tokenRequest 产生的，而 86 行的  
OAuth2Authentication 也是由 Oauth2Request 和 Authorization 这两个对象  
拼接起来的。而拼接的方式就是调用

org.springframework.security.oauth2.provider.request.DefaultOAuth2RequestFactory 的 createOAuth2Request 方法。

```
public OAuth2Request createOAuth2Request(ClientDetails client, TokenRequest tokenRequest) {
    return tokenRequest.createOAuth2Request(client);
}
```

这个方法最终会创建一个由 `clientDetails` 和 `tokenRequest` 组合而成的 `OAuth2Request`。

```
DefaultOAuth2RequestFactory.java x TokenRequest.java x OAuth2RequestFactory.java x PigxUserDetailsServiceImpl.java x TokenEndpoint.java
public void setScope(Collection<String> scope) { super.setScope(scope); }

/**
 * Set the Request Parameters on this authorization request, which represent the original request parameters and
 * should never be changed during processing. The map passed in is wrapped in an unmodifiable map instance.
 *
 * @see AuthorizationRequest#setRequestParameters
 */
public void setRequestParameters(Map<String, String> requestParameters) {
    super.setRequestParameters(requestParameters);
}

public OAuth2Request createOAuth2Request(ClientDetails client) {
    Map<String, String> requestParameters = getRequestParameters();
    HashMap<String, String> modifiable = new HashMap<String, String>(requestParameters);
    // Remove password if present to prevent leaks
    modifiable.remove("password");
    modifiable.remove("client_secret");
    // Add grant type so it can be retrieved from OAuth2Request
    modifiable.put("grant_type", grantType);
    return new OAuth2Request(modifiable, client.getClientId(), client.getAuthorities(), approved: true, this.getScope(),
        client.getResourceIds(), redirectUri: null, responseTypes: null, extensionProperties: null);
}
```

拿到 `OAuth2Request` 就可以去生成 `OAuth2Authentication` 了。

而 `OAuth2Authentication` 就是

`org.springframework.security.oauth2.provider.token.AbstractTokenGenerator` 第 71 到 73 行最终传递进去生成 `accessToken` 的对象。

而 `OAuth2Authentications` 生成成功之后进行返回的话就可以执行

`AuthorizationServerTokenServices` 的 `createAccessToken` 方法，而一旦这个 `access token` 生成成功并写入响应进行返回那么整个流程也就结束了，最终我们就拿到了想要的访问令牌。

```
protected OAuth2AccessToken getAccessToken(ClientDetails client, TokenRequest tokenRequest) {
    return tokenServices.createAccessToken(getOAuth2Authentication(client, tokenRequest));
}
```

具体创建 `accessToken` 的代码，我们需要仔细读一读

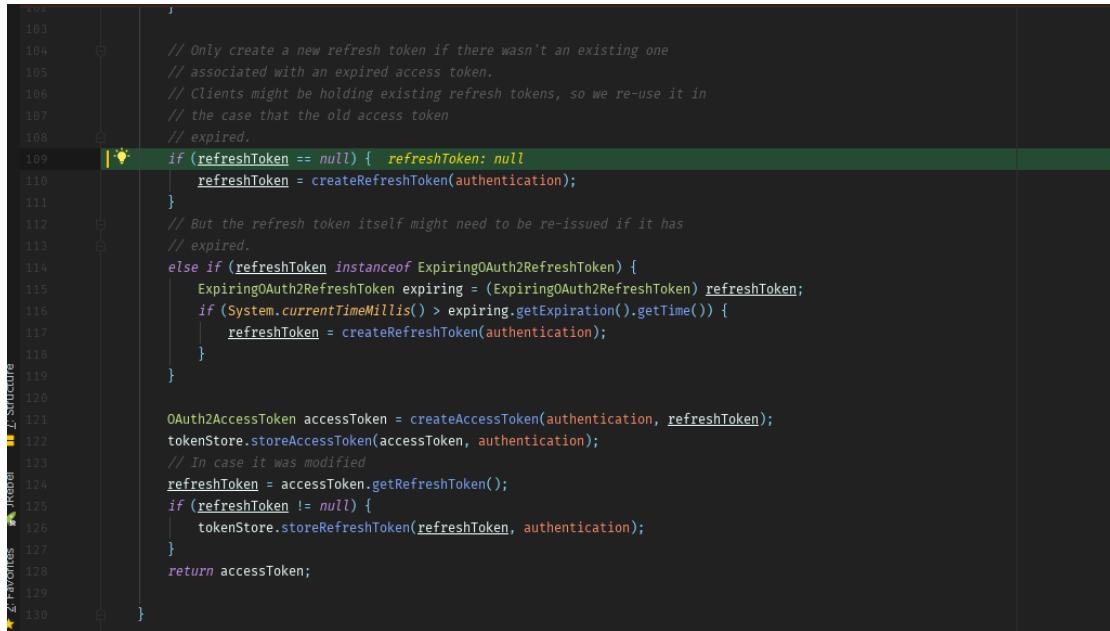
`org.springframework.security.oauth2.provider.token.DefaultTokenServices` 的 `createAccessToken` 方法。

```
81
82     @Transactional
83     public OAuth2AccessToken createAccessToken(OAuth2Authentication authentication) throws AuthenticationException { authentication: "org.springframework.security.oauth2.provider.OAuth2Authentication@0361fa5"
84         | OAuth2AccessToken existingAccessToken = tokenStore.getAccessToken(authentication); tokenStore: RedisTokenStore@13305 authentication: "org.springframework.security.oauth2.provider.OAuth2Authentication@0361fa5"
85         | OAuth2RefreshToken refreshToken = null;
86         | if (existingAccessToken != null) {
87             |     if (existingAccessToken.isExpired()) {
88                 |         if (existingAccessToken.getRefreshToken() != null) {
89                     |             refreshToken = existingAccessToken.getRefreshToken();
90                     // The token store could remove the refresh token when the
91                     // access token is removed, but we want to
92                     // be sure...
93                     tokenStore.removeRefreshToken(refreshToken);
94                 }
95                 tokenStore.removeAccessToken(existingAccessToken);
96             }
97         } else {
98             // Re-store the access token in case the authentication has changed
99             tokenStore.storeAccessToken(existingAccessToken, authentication);
100            return existingAccessToken;
101        }
102    }
103
104    // Only create a new refresh token if there wasn't an existing one
105    // associated with an expired access token.
106    // Clients might be holding existing refresh tokens, so we re-use it in
107    // the case that the old access token
108    // expired.
109    if (refreshToken == null) {
```

首先这个类一进来就会尝试在 `tokenStore` 中获取 `accessToken`, 因为同一个用户只要令牌没过期那么再次请求令牌的时候会把之前发送的令牌再次发还。因此一开始就会找当前用户已经存在的令牌。

如果已经发送的令牌不为空, 那么会在 87 行判断当前的令牌是否已经过期, 如果令牌过期了, 那么就会在 `tokenStore` 里把 `accessToken` 和 `refreshToken` 一起删掉, 如果令牌没过期, 那么就把这个没过期的令牌重新再存一下。因为可能用户是使用另外的方式来访问令牌的, 比如说一开始用授权码模式, 后来用密码模式, 而这两种模式需要存的信息是不一样的, 所以这个令牌要重新 `store` 一次。之后直接返回这个不过期的令牌。

如果令牌已经过期了或者说这个是第一次请求, 令牌压根没生成, 就会走下面的逻辑。



```
104
105     // Only create a new refresh token if there wasn't an existing one
106     // associated with an expired access token.
107     // Clients might be holding existing refresh tokens, so we re-use it in
108     // the case that the old access token
109     // expired.
110     if (refreshToken == null) { refreshToken: null
111         refreshToken = createRefreshToken(authentication);
112     }
113     // But the refresh token itself might need to be re-issued if it has
114     // expired.
115     else if (refreshToken instanceof ExpiringOAuth2RefreshToken) {
116         ExpiringOAuth2RefreshToken expiring = (ExpiringOAuth2RefreshToken) refreshToken;
117         if (System.currentTimeMillis() > expiring.getExpiration().getTime()) {
118             refreshToken = createRefreshToken(authentication);
119         }
120     }
121
122     OAuth2AccessToken accessToken = createAccessToken(authentication, refreshToken);
123     tokenStore.storeAccessToken(accessToken, authentication);
124     // In case it was modified
125     refreshToken = accessToken.getRefreshToken();
126     if (refreshToken != null) {
127         tokenStore.storeRefreshToken(refreshToken, authentication);
128     }
129
130 }
```

首先看看刷新的令牌有没有，如果刷新的令牌没有的话，那么创建一枚刷新的令牌。然后在 121 行根据 authentication, refreshToken 创建 accessToken。而这个创建 accessToken 的方法也非常简单：

```
private OAuth2AccessToken createAccessToken(OAuth2Authentication authentication, OAuth2RefreshToken refreshToken) {
    DefaultOAuth2AccessToken token = new DefaultOAuth2AccessToken(UUID.randomUUID().toString());
    int validitySeconds = getAccessTokenValiditySeconds(authentication.getOAuth2Request());
    if (validitySeconds > 0) {
        token.setExpiration(new Date(System.currentTimeMillis() + (validitySeconds * 1000L)));
    }
    token.setRefreshToken(refreshToken);
    token.setScope(authentication.getOAuth2Request().getScope());

    return accessTokenEnhancer != null ? accessTokenEnhancer增强(token, authentication) : token;
}
```

OAuth2AccessToken 其实就是用 UUID 创建一个 accessToken, 然后把过期时间，刷新令牌和 scope 这些 OAuth 协议规定的必须要存在的参数设置上，设置完了以

后它会判断是否存在 tokenEnhancer，如果存在 tokenEnhancer 它就会按照定制的 **tokenEnhancer** 增强生成出来的 token。

拿到返回的令牌之后，在 122 行 tokenStore 会把拿到的令牌存起来，然后拿 refreshToken 存起来，最后把生成的令牌返回回去。

于是我们就获取到了令牌。

```
c:\Windows\system32 /c Windows\system32
$ curl -H "Authorization:Basic dG9zDpOZKNo" -X POST http://localhost:8000/auth/oauth/token?username=admin&password=rEu1/348LvvKpOrsW06eCa%3D%3D&grant_type=password&scope=server
{"access_token": "94aff8c9-1e33-42e4-adbc-9370e04db0c3", "token_type": "bearer", "refresh_token": "420971a9-ee07-4d87-b750-75f4e955b6fe", "expires_in": 43199, "scope": "server", "license": "made by pigx"}
$ curl -H "Authorization: Bearer 94aff8c9-1e33-42e4-adbc-9370e04db0c3" http://localhost:8000/auth/me
{"username": "admin", "email": "admin@localhost", "name": "Administrator", "roles": ["ROLE_ADMIN"]}
```

扩展：

pigx 对于查询 JdbcClientDetailsService 中的查询语句做了一些增强，为什么要做增强，下面来简单分析一下。

首先我们看用于处理认证的 pigx-auth 工程的 WebSecurityConfigurer 这个配置类中创建的是如下的一个 PasswordEncoder：

```
@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
```

这个类是 Spring Security5 新出的一个类，列出了 SpringSecurity5 支持的所有的密码匹配器。

```
public static PasswordEncoder createDelegatingPasswordEncoder() {
    String encodingId = "bcrypt";
    Map<String, PasswordEncoder> encoders = new HashMap<>();
    encoders.put(encodingId, new BCryptPasswordEncoder());
    encoders.put("ldap", new LdapShaPasswordEncoder());
    encoders.put("MD4", new Md4PasswordEncoder());
    encoders.put("MD5", new MessageDigestPasswordEncoder("MD5"));
    encoders.put("noop", NoOpPasswordEncoder.getInstance());
    encoders.put("pbkdf2", new Pbkdf2PasswordEncoder());
    encoders.put("scrypt", new SCryptPasswordEncoder());
    encoders.put("SHA-1", new MessageDigestPasswordEncoder("SHA-1"));
}
```

```

        encoders.put("SHA-256", new MessageDigestPasswordEncoder("SHA-256"));
        encoders.put("sha256", new StandardPasswordEncoder());

        return new DelegatingPasswordEncoder(encodingId, encoders);
    }
}

```

具体创建密码编码器的过程也展示了要求的新密码的格式:

```

public class DelegatingPasswordEncoder implements PasswordEncoder {
    //密码匹配器 id 的前缀
    private static final String PREFIX = "{";
    //密码匹配器 id 的后缀
    private static final String SUFFIX = "}";
    //密码匹配器的类型
    private final String idForEncode;
    private final PasswordEncoder passwordEncoderForEncode;
    private final Map<String, PasswordEncoder> idToPasswordEncoder;

    /**
     * 密码的格式匹配不上就会报错，相信每个人在升级的时候都经历过 There is no PasswordEncoder mapped for the id “null”的绝望吧！
     */
    private class UnmappedIdPasswordEncoder implements PasswordEncoder {

        @Override
        public String encode(CharSequence rawPassword) {
            throw new UnsupportedOperationException("encode is not supported");
        }

        @Override
        public boolean matches(CharSequence rawPassword,
                              String prefixEncodedPassword) {
            String id = extractId(prefixEncodedPassword);
            throw new IllegalArgumentException("There is no PasswordEncoder mapped for the id \\" + id + "\\\"");
        }
    }
}

```

这个类就要求了密码必须符合带上{"具体的解密器 id"}, 最后根据这个 id 去找密码匹配器匹配,

clientSecret 最终也是要参与解码的, 所以它也需要带上{"id"},clientSecret 我们并不需要做什么艰深的加密, 所以使用原始密码就行, 这个解密器就是 NoOpPasswordEncoder, 它的 id 从上文我们看到是"noop",也就是说数据库里的 clientSecret 要想在 Spring Security5 下正常工作, clientId 应该是 testclientSecret 应该是{noop}test,但是我们可以看到数据库里存储的都是 test/test 那为什么进行解密的时候没有抛出 PasswordEncoder mapped for the id "null"的异常呢?

原因很简单。

在 com.pig4cloud.pigx.common.core.constant.SecurityConstants 查询客户端信息的语句中, 我们可以看到{noop}这个字段在查询出来注入 JdbcClientDetailsService 之前, 作者已经利用 Mysql 的连接函数帮我们拼接好了。

```
String CLIENT_FIELDS = "client_id, CONCAT('{noop}',client_secret)  
as client_secret, resource_ids, scope,  
+ "authorized_grant_types, web_server_redirect_uri,  
authorities, access_token_validity,  
+ "refresh_token_validity, additional_information,  
autoapprove";
```

同样巧妙的设定也体现在了用户密码的加密上。

在 upms 模块的 UserController 模块中, 作者显式指定了密码解密器为 BCryptPasswordEncoder。

```
@Slf4j  
@Service  
@AllArgsConstructor
```

```
public class SysUserServiceImpl extends ServiceImpl<SysUserMapper, SysUser> implements SysUserService {
    private static final PasswordEncoder ENCODER = new BCryptPasswordEncoder();
    private final SysMenuService sysMenuService;
    private final SysUserMapper sysUserMapper;
    private final SysRoleService sysRoleService;
    private final SysUserRoleService sysUserRoleService;
    private final SysDeptRelationService sysDeptRelationService;
    // 其他代码省略
}
```

#### Tips:

为什么都声明成 `final` 级别变量，要结合上面的 `lombok` 的 `@AllArgsConstructor` 注解来看，其实就是为了使用 `Lombok` 的黑科技进行构造器注入，这也是 `Spring 5` 推荐的一种注入方式。

但是很显然，这种密码解密器直接参与进 `Spring Security5` 的执行流程又会报喜闻乐见的 `There is no PasswordEncoder mapped for the id "null"` 错误，那么为什么没报呢？见代码的

`com.pig4cloud.pigx.common.security.service.PigxUserDetailsService` `Impl` 类的 `getUserDetails` 方法：

```
/**
 * 构建 userdetails
 *
 * @param result 用户信息
 * @return
 */
private UserDetails getUserDetails(R<UserInfo> result) {
    if (result == null || result.getData() == null) {
        throw new UsernameNotFoundException("用户不存在");
    }

    UserInfo info = result.getData();
    Set<String> dbAuthsSet = new HashSet<>();
    if (ArrayUtil.isNotEmpty(info.getRoles())) {
```

```
// 获取角色
    Arrays.stream(info.getRoles()).forEach(role -> dbAuthsSet.
add(SecurityConstants.ROLE + role));
    // 获取资源
    dbAuthsSet.addAll(Arrays.asList(info.getPermissions()));

}
Collection<? extends GrantedAuthority> authorities
    = AuthorityUtils.createAuthorityList(dbAuthsSet.toArray(ne
w String[0]));
    SysUser user = info.getSysUser();
    boolean enabled = StrUtil.equals(user.getDelFlag(), CommonCons
tant.STATUS_NORMAL);
    // 构造 security 用户

    return new PigxUser(user.getUserId(), user.getDeptId(), user.g
etUsername(), SecurityConstants.BCRYPT + user.getPassword(), enab
led,
        true, true, true, authorities);
}
```

见构造 security 用户的部分，作者在构造 Security 的 User 对象进行认证之前，进行了和处理 clientSecret 类似操作，手动拼接了"{}bcrypt"的字符。

作者的这两个操作，据我个人推测，应该是为了保证和 Spring Security 4.x 的密码格式的兼容性，隐藏密码变更的细节。

## pig CheckToken 过程讲解

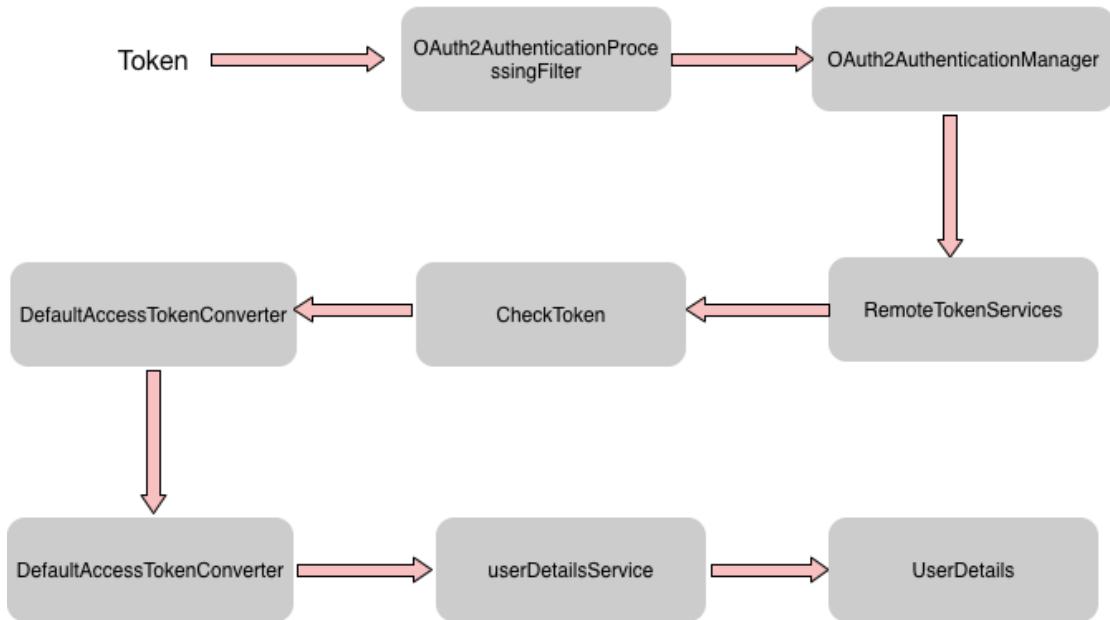
### CheckToken 的目的

---

当用户携带 token 请求资源服务器的资源

时，**OAuth2AuthenticationProcessingFilter** 拦截 token，进行 token 和 userdetails 过程，把无状态的 token 转化成用户信息。

Made by pigX



## 详解

1. OAuth2AuthenticationManager.authenticate(), filter 执行判断的入口

```
public Authentication authenticate(Authentication authentication) throws AuthenticationException {
    if (authentication == null) {
        throw new InvalidTokenException("Invalid token (token not found)");
    }
    String token = (String) authentication.getPrincipal();
    OAuth2Authentication auth = tokenServices.loadAuthentication(token);
    if (auth == null) {
        throw new InvalidTokenException("Invalid token: " + token);
    }

    Collection<String> resourceIds = auth.getOAuth2Request().getResourceIds();
    if (resourceId != null && resourceIds != null && !resourceIds.isEmpty() &&
    !resourceIds.contains(resourceId)) {
        throw new OAuth2AccessDeniedException("Invalid token does not contain resource id (" + resourceId +
    ")");
    }

    return auth;
}
```

2. 当用户携带 token 去请求微服务模块，被资源服务器拦截调用

RemoteTokenServices.loadAuthentication ,执行所谓的 check-token 过程。

源码如下

```
public OAuth2Authentication loadAuthentication(String accessToken) throws
    AuthenticationException, InvalidTokenException {

    MultiValueMap<String, String> formData = new LinkedMultiValueMap<String, String>();
    formData.add(tokenName, accessToken);
    HttpHeaders headers = new HttpHeaders();
    headers.set("Authorization", getAuthorizationHeader(clientId, clientSecret));
    Map<String, Object> map = postForMap(checkTokenEndpointUrl, formData, headers);

    if (map.containsKey("error")) {
        logger.debug("check_token returned error: " + map.get("error"));
        throw new InvalidTokenException(accessToken);
    }

    // gh-838
    if (!Boolean.TRUE.equals(map.get("active"))) {
        logger.debug("check_token returned active attribute: " + map.get("active"));
        throw new InvalidTokenException(accessToken);
    }

    return tokenConverter.extractAuthentication(map);
}
```

3. CheckToken 处理逻辑很简单，就是调用 redisTokenStore 查询 token 的合法性，及其返回用户的部分信息（username）

```
@RequestMapping(value = "/oauth/check_token")
@ResponseBody
public Map<String, ?> checkToken(@RequestParam("token") String value) {

    OAuth2AccessToken token = resourceServerTokenServices.readAccessToken(value);
    if (token == null) {
        throw new InvalidTokenException("Token was not recognised");
    }

    if (token.isExpired()) {
        throw new InvalidTokenException("Token has expired");
    }

    OAuth2Authentication authentication = resourceServerTokenServices.loadAuthentication(token.getValue());

    Map<String, Object> response = (Map<String, Object>) accessTokenConverter.convertAccessToken(token,
        authentication);

    // gh-1070
    response.put("active", true); // Always true if token exists and not expired

    return response;
}
```

4. 继续看 返回给 RemoteTokenServices.loadAuthentication 最后一句  
tokenConverter.extractAuthentication 解析组装服务端返回的信息

```
public OAuth2Authentication extractAuthentication(Map<String, ?> map) {  
    Map<String, String> parameters = new HashMap<String, String>();  
    Set<String> scope = extractScope(map);  
    Authentication user = userTokenConverter.extractAuthentication(map);  
    String clientId = (String) map.get(clientIdAttribute);  
    parameters.put(clientIdAttribute, clientId);  
    if (includeGrantType && map.containsKey(GRANT_TYPE)) {  
        parameters.put(GRANT_TYPE, (String) map.get(GRANT_TYPE));  
    }  
    Set<String> resourceIds = new LinkedHashSet<String>(map.containsKey(AUD) ? getAudience(map)  
        : Collections.<String>emptySet());  
  
    Collection<? extends GrantedAuthority> authorities = null;  
    if (user==null && map.containsKey(AUTHORITIES)) {  
        @SuppressWarnings("unchecked")  
        String[] roles = ((Collection<String>)map.get(AUTHORITIES)).toArray(new String[0]);  
        authorities = AuthorityUtils.createAuthorityList(roles);  
    }  
    OAuth2Request request = new OAuth2Request(parameters, clientId, authorities, true, scope, resourceIds, null,  
        null,null);  
    return new OAuth2Authentication(request, user);  
}
```

最重要的 `userTokenConverter.extractAuthentication(map);`

5. 最重要的一步，是否判断是否有 `userDetailsService` 实现，如果有 的话去查根据返回的

`username` 查询一次全部的用户信息，没有实现直接返回 `username`，这也是很多时候问的为什么只能查询到 `username` 也就是 `EnablePigxResourceServer.details` `true` 和 `false` 的区别。

```
public Authentication extractAuthentication(Map<String, ?> map) {  
    if (map.containsKey(USERNAME)) {  
        Object principal = map.get(USERNAME);  
        Collection<? extends GrantedAuthority> authorities = getAuthorities(map);  
        if (userDetailsService != null) {  
            UserDetails user = userDetailsService.loadUserByUsername((String) map.get(USERNAME));  
            authorities = user.getAuthorities();  
            principal = user;  
        }  
        return new UsernamePasswordAuthenticationToken(principal, "N/A", authorities);  
    }  
    return null;  
}
```

6. 那根据的你问题，继续看 `UserDetailsService.loadUserByUsername` 根据用户名去换取用户全部信息

```
public interface UserDetailsService {
    // ~ Methods
    // =====

    /**
     * Locates the user based on the username. In the actual implementation, the search
     * may possibly be case sensitive, or case insensitive depending on how the
     * implementation instance is configured. In this case, the <code>UserDetails</code>
     * object that comes back may have a username that is of a different case than what
     * was actually requested..
     *
     * @param username the username identifying the user whose data is required.
     *
     * @return a fully populated user record (never <code>null</code>)
     *
     * @throws UsernameNotFoundException if the user could not be found or the user has no
     * GrantedAuthority
     */
    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
}
```

## pig 扩展支持 oauth2 客户端模式

客户端模式，没有具体用户信息，只有客户的 `client-id` 信息，完成不了 `upms` 的 RBAC，现有认证中心不支持发放 token。

怎么支持认证中心发放 token，有了 token，我其他的微服务不用 `upms` 是否可以？当然

`AuthorizationServerConfig.java` 修改 token 增强逻辑，如果为客户端模式就不进行增强，即不维护用户信息。

```
/**
 * token 增强，客户端模式不增强。
 *
 * @return TokenEnhancer
 */
@Bean
public TokenEnhancer tokenEnhancer() {
    return (accessToken, authentication) -> {
        if (SecurityConstants.CLIENT_CREDENTIALS
            .equals(authentication.getOAuth2Request().getGrantType())) {
            return accessToken;
        }
    }
}
```

```

        final Map<String, Object> additionalInfo = new HashMap<>(8);
        PigxUser pigxUser = (PigxUser) authentication.getUserAuthentication().getPrincipal();
        additionalInfo.put("user_id", pigxUser.getId());
        additionalInfo.put("username", pigxUser.getUsername());
        additionalInfo.put("dept_id", pigxUser.getDeptId());
        additionalInfo.put("tenant_id", pigxUser.getTenantId());
        additionalInfo.put("license", SecurityConstants.PIGX_LICENSE);
        ((DefaultOAuth2AccessToken) accessToken).setAdditionalInformation(additionalInfo);
        return accessToken;
    };
}

```

即可方法令牌了，只是这个令牌只有客户端信息。

为何不增强参考 源码 DefaultTokenServices

```

private OAuth2AccessToken createAccessToken(OAuth2Authentication authentication, OAuth2RefreshToken refreshToken) {
    DefaultOAuth2AccessToken token = new DefaultOAuth2AccessToken(UUID.randomUUID().toString());
    int validitySeconds = getAccessTokenValiditySeconds(authentication.getOAuth2Request());
    if (validitySeconds > 0) {
        token.setExpiration(new Date(System.currentTimeMillis() + (validitySeconds * 1000L)));
    }
    token.setRefreshToken(refreshToken);
    token.setScope(authentication.getOAuth2Request().getScope());

    return accessTokenEnhancer != null ? accessTokenEnhancer.enhance(token, authentication) : token;
}

```

## 功能使用

## 代码生成使用

### 功能支持

前端:

- api.js
- crud.js
- index.vue

后端

- entity
- mapper
- service
- Controller

### 开始使用

项目启动

1. 启动 pig-eureka、pig-config、pig-gateway、pig-auth、pig-upms-biz
2. 启动 **PigCodeGenApplication**
3. 启动 **pig-ui**

界面操作

1. 代码生成模块

The screenshot shows the PigX interface for code generation. On the left, a sidebar lists various management modules: 系统管理, 用户管理, 菜单管理, 角色管理, 日志管理, 字典管理, 部门管理, 客户端管理, and 代码生成. The '代码生成' module is highlighted with a red box. The main area displays a table of tables with columns: #, 表名称, 表注释, 索引, 创建时间, and 操作. Each row has a '生成' button in the '操作' column. The table contains 8 rows of data.

#	表名称	表注释	索引	创建时间	操作
1	sys_role_menu	角色菜单表	InnoDB	1532557694000	<button>✓ 生成</button>
2	sys_role_dept	角色与部门对应关系	InnoDB	1532557694000	<button>✓ 生成</button>
3	sys_role		InnoDB	1532557694000	<button>✓ 生成</button>
4	sys_oauth_client_data	lls	InnoDB	1532557694000	<button>✓ 生成</button>
5	sys_menu	菜单权限表	InnoDB	1532557694000	<button>✓ 生成</button>
6	sys_log	日志表	InnoDB	1532557694000	<button>✓ 生成</button>
7	sys_dict	字典表	InnoDB	1532557694000	<button>✓ 生成</button>
8	sys_dept_relation		InnoDB	1532557694000	<button>✓ 生成</button>

## 2. 选择要生成的表

以下为空则从 `pigx-codegen/generator.properties` 获取

The screenshot shows the '生成配置' (Generation Configuration) dialog box overlaid on the main code generation interface. The dialog contains fields for: 表名称 (Table Name: sys\_user), 包名 (Package: com.pig4cloud.pigx), 作者 (Author: 冷冷), 模块 (Module: admin), 表前缀 (Table Prefix: sys\_), and 注释 (Annotation: 代码生成). A '生成' (Generate) button is at the bottom. In the background, the main table of tables is visible with several rows having their '生成' buttons highlighted in blue.

## 3. 解压下载的 `pig_code_gen.zip`

生成代码结构，安装前后端 `maven`、`vue-cli` 目录生成，可以覆盖到指定业务模块

名称	修改日期	大小	种类
└─ pigx	今天 08:44	48 KB	文件夹
└─ src	今天 08:44	42 KB	文件夹
└─ main	今天 08:44	36 KB	文件夹
└─ java	今天 08:44	29 KB	文件夹
└─ com	今天 08:44	23 KB	文件夹
└─ pig4cloud	今天 08:44	16 KB	文件夹
└─ pigx	今天 08:44	10 KB	文件夹
└─ admin	今天 08:42	4 KB	文件夹
└─ controller	今天 08:42	2 KB	文件夹
└─ entity	今天 08:42	1 KB	文件夹
└─ mapper	今天 08:42	324 字节	文件夹
└─ service	今天 08:42	816 字节	文件夹
└─ resources	今天 08:42	911 字节	文件夹
└─ pigx-ui	今天 08:44	33 KB	文件夹
└─ src	今天 08:45	27 KB	文件夹
└─ api	今天 08:42	1 KB	文件夹
└─ const	今天 08:42	2 KB	文件夹
└─ views	今天 08:45	18 KB	文件夹
└─ admin	今天 08:45	11 KB	文件夹
└─ user	今天 08:42	5 KB	文件夹
└─ index.vue	今天 08:42	5 KB	Sublime...pp 文稿
└─ user_menu.sql	今天 08:42	1 KB	SQL

#### 4. 重点讲解生成的 SQL 使用

生成的 SQL 脚本，不要直接执行，完善 菜单、按钮的菜单 ID 的层级

PS: 为什么菜单 ID 不自动生成呢？

- 通过自己录入保证，数据库展示层级。
- 比如 父菜单 ID 为 1,子菜单的 ID 则为 11，按钮 ID 为 12, 13, 14

```

1
├── 11
├── 12
├── 13
└── 14
user_menu.sql
1 -- 该脚本不要执行，请完善 ID 对应关系,注意层级关系 !!!!!
2
3 -- 菜单SQL
4 insert into `pigx`.`sys_menu` ( `parent_id`, `component`, `permission`, `type`, `path`, `icon`, `menu_id`, `del_flag`, `cre
5 | values ('父菜单ID', 'views/admin/user/index', '菜单ID', '0', 'user', 'icon-bangzhushouji', '8', '0', '2018-01-20 13:17:1
6
7 -- 菜单对应按钮SQL
8 insert into `pigx`.`sys_menu` ( `parent_id`, `component`, `permission`, `type`, `path`, `icon`, `menu_id`, `del_flag`, `cre
9 | values ('子按钮ID1', null, 'admin_user_add', '1', null, '1', '菜单ID', '0', '2018-05-15 21:35:18', '0', '2018-07-29 13:1
10 insert into `pigx`.`sys_menu` ( `parent_id`, `component`, `permission`, `type`, `path`, `icon`, `menu_id`, `del_flag`, `cre
11 | values ('子按钮ID2', null, 'admin_user_edit', '1', null, '1', '菜单ID', '0', '2018-05-15 21:35:18', '1', '2018-07-29 13:
12 insert into `pigx`.`sys_menu` ( `parent_id`, `component`, `permission`, `type`, `path`, `icon`, `menu_id`, `del_flag`, `cre
13 | values ('子按钮ID3', null, 'admin_user_del', '1', null, '1', '菜单ID', '0', '2018-05-15 21:35:18', '2', '2018-07-29 13:1
14

```

#### 5. 配置路由代理

前端 vue.config.js

```
'/demo': {
    target: baseUrl,
    changeOrigin: true,
    pathRewrite: {
        '^/demo': '/demo'
    }
},
```

网关新增路由

```
spring:
  cloud:
    gateway:
      locator:
        enabled: true
      routes:
        - id: pig-demo
          uri: lb://pig-demo
          predicates:
            - Path=/demo/**
```

复制

最后给角色分配，你新增的菜单和按钮喔

## EnablePigFeignClients 原理解析

### 源码解析

只是给默认的 EnableFeignClients 增加了一个默认值。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@EnableFeignClients
public @interface EnablePigxFeignClients {

    String[] value() default {};

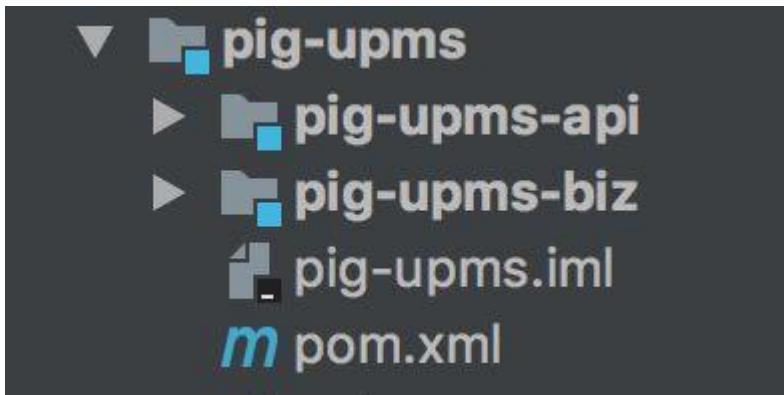
    // 指定默认的扫描范围
    String[] basePackages() default {"com.pig4cloud.pigx"};

    Class<?>[] basePackageClasses() default {};

    Class<?>[] defaultConfiguration() default {};
```

```
    Class<?>[] clients() default {};  
}
```

以 UPMS 为例分析封装的好处



- 如果使用原生的 `@EnableFeignClients` 默认的扫描范围是 `com.pig4cloud.pig.admin` 包的所有 `FeignClient`。
- 而由于微服务拆分所有的 `feignClient` 都在 `com.pig4cloud.pig.api` 包里面，这样默认情况会扫描不到
- 除非明确指定扫描范围 `@EnableFeignClients("com.pig4cloud.pig.模块.api")`
- 使用了`@EnablePigFeignClients` 默认扫描 `com.pig4cloud.pigx` 下边的 `feignClient` 更为简洁

`@EnableFeignClients`

```
@EnableFeignClients  
@SpringCloudApplication  
public class PigAdminApplication {  
}
```

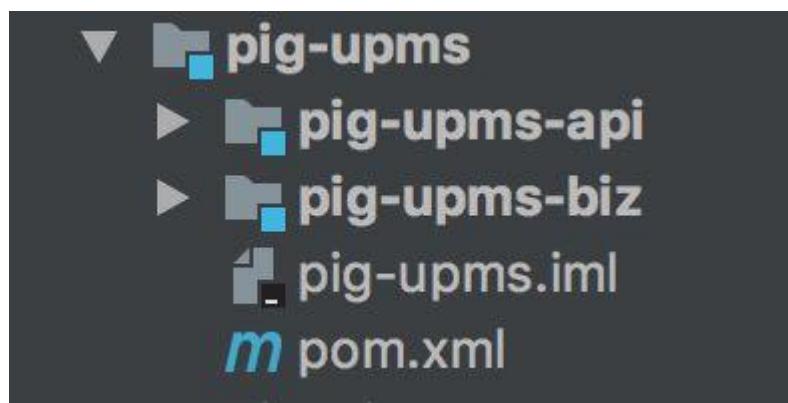
`@EnablePigFeignClients`

```
@EnablePigFeignClients
```

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

## Pig 中 FeignClient 使用说明

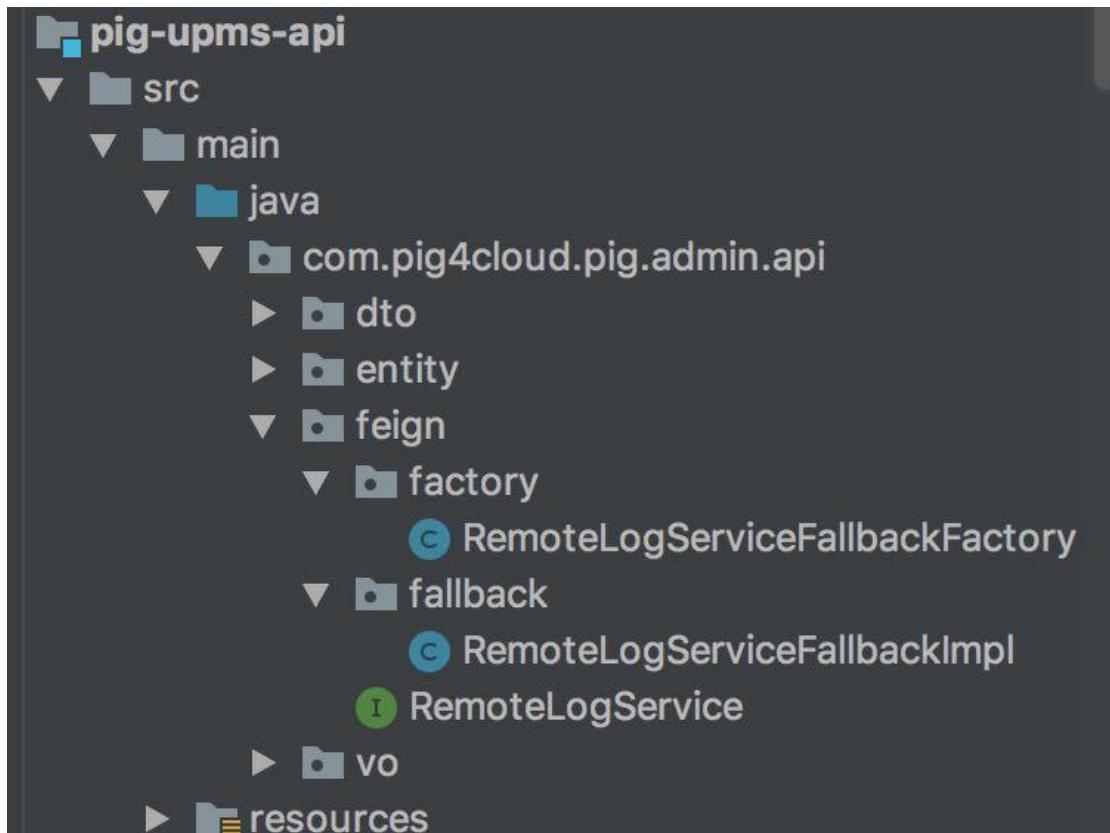
### 分包说明



具体参考上一章节[[EnablePigFeignClients 原理解析](#)]

- main 类 使用了@EnablePigFeignClients 默认扫描
- 出现 FeignClient 调用报错等, 请检查你的分包是否符合 pig 的标准
- 包名 com.pig4cloud.pig.模块.api

新增 feignClient



- RemoteXXXService FeignClient 客户端, 声明具体的接口和降级工程

```
@FeignClient(value = ServiceNameConstants.UMPS_SERVICE, fallbackFactory = RemoteLogServiceFallbackFactory.class)
```

- 降级工厂类, 注意这里的@Component 注入到 Spring

```
@Component
public class RemoteLogServiceFallbackFactory implements FallbackFactory<RemoteLogService> {

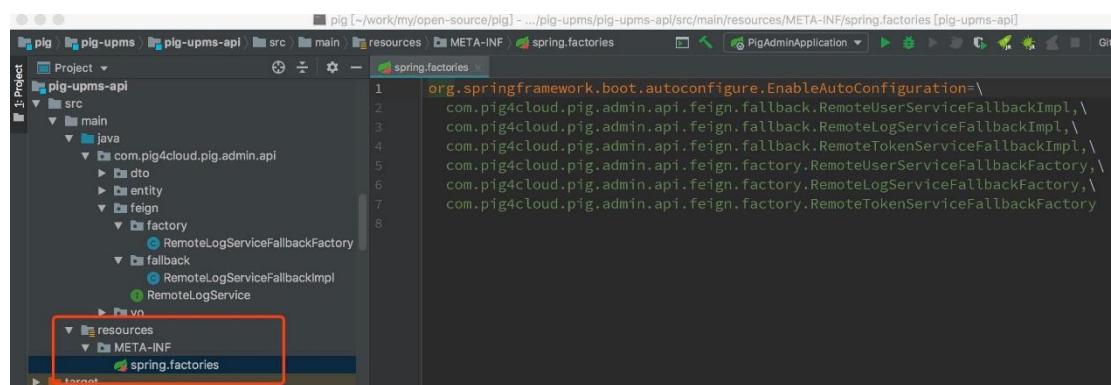
    @Override
    public RemoteLogService create(Throwable throwable) {
        RemoteLogServiceFallbackImpl remoteLogServiceFallback = new RemoteLogServiceFallbackImpl();
        remoteLogServiceFallback.setCause(throwable);
        return remoteLogServiceFallback;
    }
}
```

```
}
```

- 具体的降级业务

```
@Component
public class RemoteLogServiceFallbackImpl implements RemoteLogService {
    @Setter
    private Throwable cause;
    public R<Boolean> saveLog(SysLog sysLog, String from) {
        log.error("feign 插入日志失败", cause);
        return null;
    }
}
```

为了保证引入 api 模块可以直接使用，在 `spring.factories` 配置 bean



```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
    com.pig4cloud.pig.admin.api.feign.fallback.RemoteXXXServiceFall\
backImpl,\

com.pig4cloud.pig.admin.api.feign.factory.RemoteXXXServiceFallbac\
kFactory
```

## 验证码配置及开关

验证码直接网关异步生成，基于 webflux

webflux 生成验证码的 handler

```
public class ImageCodeHandler implements HandlerFunction<ServerRespon\
se> {
```

```

private final Producer producer;
private final RedisTemplate redisTemplate;

@Override
public Mono<ServerResponse> handle(ServerRequest serverRequest) {
    //生成验证码
    String text = producer.createText();
    BufferedImage image = producer.createImage(text);

    //保存验证码信息
    String randomStr = serverRequest.queryParam("randomStr").get();
    redisTemplate.opsForValue().set(CommonConstants.DEFAULT_CODE_KEY + randomStr, text, 60, TimeUnit.SECONDS);

    // 转换流信息写出
    FastByteArrayOutputStream os = new FastByteArrayOutputStream();
    try {
        ImageIO.write(image, "jpeg", os);
    } catch (IOException e) {
        log.error("ImageIO write err", e);
        return Mono.error(e);
    }

    return ServerResponse
        .status(HttpStatus.OK)
        .contentType(MediaType.IMAGE_JPEG)
        .body(BodyInserters.fromResource(new ByteArrayResource(os.toByteArray())));
}
}

```

## webflux 请求处理入口

```

public class RouterFunctionConfiguration {
    private final HystrixFallbackHandler hystrixFallbackHandler;
    private final ImageCodeHandler imageCodeHandler;

    @Bean
    public RouterFunction routerFunction() {

```

```

        return RouterFunctions.route(RequestPredicates.GET
        ("/code"))
            .and(RequestPredicates.accept(MediaType.TEXT_PLAIN)), imageCodeHandler);

    }

}

```

校验逻辑,通过 oauth2 终端的 client-id 来确定是否校验验证码

```

public class ValidateCodeGatewayFilter extends AbstractGatewayFilterFactory {
    @Override
    public GatewayFilter apply(Object config) {
        return (exchange, chain) -> {
            ServerHttpRequest request = exchange.getRequest();

            // 终端设置不校验, 直接向下执行
            String[] clientInfos = WebUtils.getClientId(request);
            if (filterIgnorePropertiesConfig.getClients().contains(clientInfos[0])) {
                return chain.filter(exchange);
            }

            //校验验证码
            checkCode(request);

            return chain.filter(exchange);
        };
    }
}

```

配置终端不校验验证码

```

pig-gateway-dev.yml

# 不校验验证码终端
ignore:
  clients:

```

- 不校验验证码的终端 client-id

## Pig 配置加解密

# jasypt 的解决方案

### 1. Maven 依赖

```
<dependency>
    <groupId>com.github.ulisesbocchio</groupId>
    <artifactId>jasypt-spring-boot-starter</artifactId>
    <version>1.16</version>
</dependency>
```

### 2. 配置

```
jasypt:
  encryptor:
    password: foo #根密码
```

### 3 调用 JAVA API 生成密文

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = PigAdminApplication.class)
public class PigAdminApplicationTest {
    @Autowired
    private StringEncryptor stringEncryptor;

    @Test
    public void testEnvironmentProperties() {
        System.out.println(stringEncryptor.encrypt("lenglen
g"));
    }
}
```

或者直接使用 JAVA 方法调用（不依赖 spring 容器）

```
/**
 * jasypt.encryptor.password 对应 配置中心 application-dev.yml
 中的密码
 */
@Test
```

```
public void testEnvironmentProperties() {
    System.setProperty(JASYPT_ENCRYPTOR_PASSWORD, "lengleng");
    StringEncryptor stringEncryptor = new DefaultLazyEncryptor
(new StandardEnvironment());

    //加密方法
    System.out.println(stringEncryptor.encrypt("123456"));
    //解密方法
    System.out.println(stringEncryptor.decrypt("saRv7ZnXsNAfs1
3AL90pCQ=="));
}
```

#### 4 配置文件中使用密文

```
spring:
  datasource:
    password: ENC(密文)

  xxx: ENC(密文)
```

#### 5 其他非对称等高级配置参考

### 总结

---

1. Spring Cloud Config 提供了统一的加解密方式，方便使用，但是如果应用配置没有走配置中心，那么加解密过滤是无效的；依赖 JCE 对于低版本 spring cloud 的兼容性不好。
2. jasypt 功能更为强大，支持的加密方式更多，但是如果多个微服务，需要每个服务模块引入依赖配置，较为麻烦；但是功能强大、灵活。
3. 个人选择 jasypt

### 前端报文加密及其解密处理

#### 前端报文加密的业务

用户登录时，对登录密码进行加密传输

▼ Query String Parameters [view source](#) [view URL encoded](#)

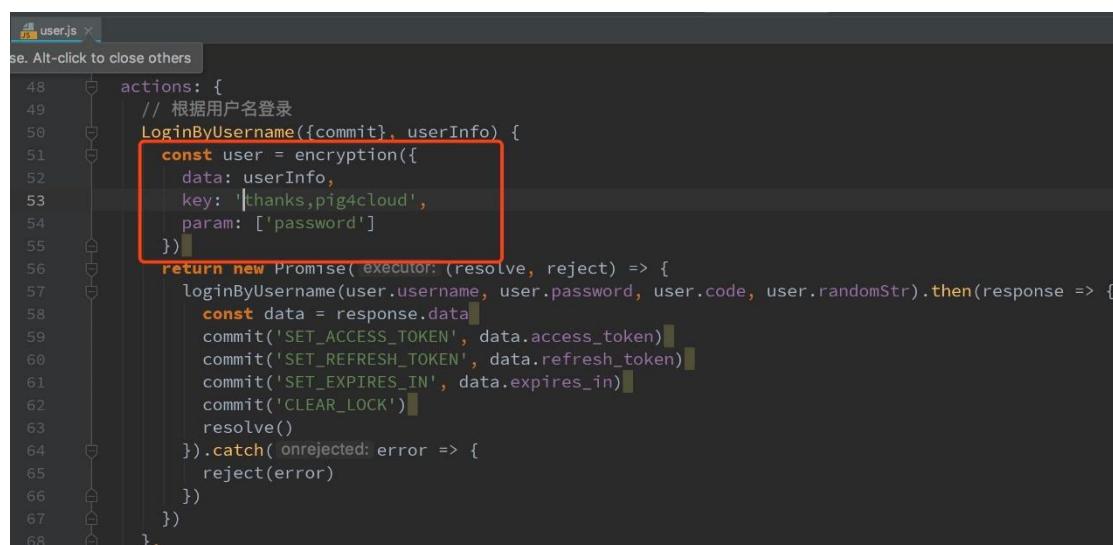
```

username: admin
password: rKu1/348LvKp0rsVC06eCA==
randomStr: 2002155067290905
code: n6xe
grant_type: password
scope: server

```

## 前端加密功能

前端提供简单的 AES 对称加密算法，注意 key 和后端网关配置相同，这里打包混淆后，相对安全。



```

48     actions: {
49       // 根据用户名登录
50       LoginByUsername({commit}, userInfo) {
51         const user = encryption({
52           data: userInfo,
53           key: 'thanks,pig4cloud',
54           param: ['password']
55         })
56         return new Promise((executor, reject) => {
57           loginByUsername(user.username, user.password, user.code, user.randomStr).then(response => {
58             const data = response.data
59             commit('SET_ACCESS_TOKEN', data.access_token)
60             commit('SET_REFRESH_TOKEN', data.refresh_token)
61             commit('SET_EXPIRES_IN', data.expires_in)
62             commit('CLEAR_LOCK')
63             resolve()
64           }).catch(rejected => {
65             reject(error)
66           })
67         })
68       },

```

## 后端解密功能

使用 hutool 提供的工具类进行解密

```

public class PasswordDecoderFilter extends AbstractGatewayFilterFactory {
    @Override
    public GatewayFilter apply(Object config) {
        return (exchange, chain) -> {
            ServerHttpRequest request = exchange.getRequest();
            URI uri = exchange.getRequest().getURI();
            String queryParam = uri.getRawQuery();
        }
    }
}

```

```
        Map<String, String> paramMap = HttpUtil.deco
deParamMap(queryParam, CharsetUtil.UTF_8);

        String password = paramMap.get(PASSWORD);
        if (StrUtil.isNotBlank(password)) {
            try {
                password = decryptAES(password, encodeKey);
            } catch (Exception e) {
                log.error("密码解密失败:{}" , password);
                return Mono.error(e);
            }
            paramMap.put(PASSWORD, password.trim());
        }

        URI newUri = UriComponentsBuilder.fromUri(ur
i)
            .replaceQuery(HttpUtil.toParams(param
Map))
            .build(true)
            .toUri();

        ServerHttpRequest newRequest = exchange.getR
equest().mutate().uri(newUri).build();
        return chain.filter(exchange.mutate().reques
t(newRequest).build());
    };
}
}
```

复制

## 总结

杠精不要抬杠 AES 是对称加密，不安全。

## 登录后置处理

用户登录成功、失败后，pig 捕获了 spring security 发出的对应事件。

- 用户登录成功时，发布 AuthenticationSuccessEvent 事件

```
public class PigAuthenticationSuccessEventHandler extends AuthenticationSuccessEventHandler {

    /**
     * 处理登录成功方法
     * <p>
     * 获取到登录的 authentication 对象
     *
     * @param authentication 登录对象
     */
    @Override
    public void handle(Authentication authentication) {
        log.info("用户: {} 登录成功", authentication.getPrincipal());
    }
}
```

- 用户登录失败时

AuthenticationException 是登录异常信息，包括常见的用户密码不正确，用户信息不正确，用户状态不正确等

```
@Slf4j
@Component
public class PigAuthenticationFailureEvenHandler extends AuthenticationFailureEvenHandler {

    /**
     * 处理登录失败方法
     * <p>
     * 
     * @param authenticationException 登录的 authentication 对象
     * @param authentication          登录的 authenticationException 对象
     */
    @Override
    public void handle(AuthenticationException authenticationException, Authentication authentication) {
        log.info("用户: {} 登录失败, 异常: {}", authentication.getPrincipal(), authenticationException.getLocalizedMessage());
```

```
    }  
}
```

- Authentication 用户身份认证信息

```
public interface Authentication extends Principal, Serializable {  
  
    // 用户角色 + 权限信息（会包含用户的权限标志）  
    Collection<? extends GrantedAuthority> getAuthorities();  
  
    // 用户密码加密串  
    Object getCredentials();  
  
    // 用户名或者用户全部信息（参考资源服务配置章节说明）  
    Object getPrincipal();  
  
    // 是否认证  
    boolean isAuthenticated();  
  
    ...  
}
```

## 日志处理-Spring Event 处理机制

### POM 依赖

```
<!-- 日志处理 -->  
<dependency>  
    <groupId>com.pig4cloud</groupId>  
    <artifactId>pig-common-log</artifactId>  
    <version>${log.version}</version>  
</dependency>
```

### @SysLog 注解

- 接口上使用@SysLog 注释当前接口的作用即可

```
@SysLog("添加终端")  
@PostMapping  
@PreAuthorize("@pms.hasPermission('sys_client_add')")
```

```
public R add(@Valid @RequestBody SysOAuthClientDetails sysOAuthClientDetails) {
    return new R<>(sysOAuthClientDetailsService.save(sysOAuthClientDetails));
}
```

## 原理讲解

- AOP 切面获取当前请求的注解值，并 异步 发送时间，减少日志操作的性能损耗

```
@Aspect
@Slf4j
public class SysLogAspect {

    @Around("@annotation(sysLog)")
    public Object around(ProceedingJoinPoint point, SysLog sysLog) throws Throwable {
        String strClassName = point.getTarget().getClass().getName();
        String strMethodName = point.getSignature().getName();
        log.debug("[类名]:{},[方法]:{}", strClassName, strMethodName);
        SpringContextHolder.publishEvent(new SysLogEvent(logVo));
        return obj;
    }
}
```

- 监听器在接收到日志事件后进行调用 feign 入口处理

```
@Slf4j
@AllArgsConstructor
public class SysLogListener {
    private final RemoteLogService remoteLogService;

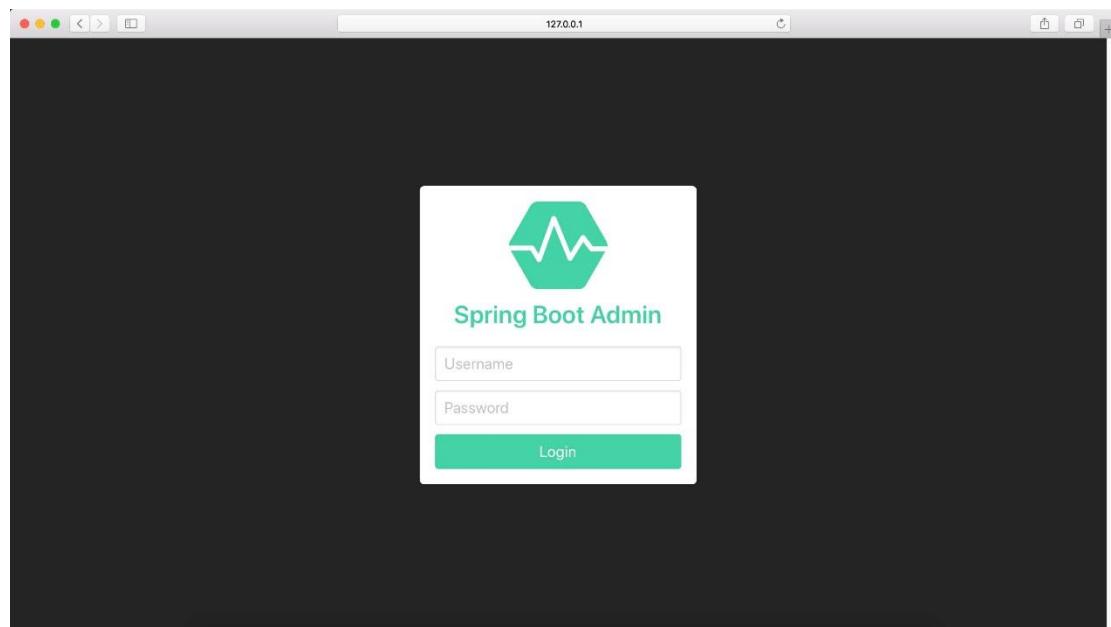
    @Async
    @Order
    @EventListener(SysLogEvent.class)
    public void saveSysLog(SysLogEvent event) {
        SysLog sysLog = (SysLog) event.getSource();
```

```
        remoteLogService.saveLog(sysLog, SecurityConstants.  
FROM_IN);  
    }  
}  
复制
```

## 异步操作说明 `@EnableAsync`

`@EnableAsync` 注解启用了 Spring 异步方法执行功能，在 [Spring Framework API](#) 中有详细介绍。

## 监控模块二次认证



## 为什么要做二次认证

spring boot admin 默认没有开启认证，也是就是别人知道了监控模块的 IP:PORT 即可访问。监控功能在生产上又是必要的功能，所以需要有二次认证

## 实现原理

- 引入 spring security

```
<!--security-->
```

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

- 配置 spring security 即可

```
@Configuration
public class WebSecurityConfigurer extends WebSecurityConfigurerAdapter {
    private final String adminContextPath;

    public WebSecurityConfigurer(AdminServerProperties adminServerProperties) {
        this.adminContextPath = adminServerProperties.getAdminContextPath();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // @formatter:off
        SavedRequestAwareAuthenticationSuccessHandler successHandler = new SavedRequestAwareAuthenticationSuccessHandler();
        successHandler.setTargetUrlParameter("redirectTo");
        successHandler.setDefaultTargetUrl(adminContextPath + "/");
        http
            .headers().frameOptions().disable()
            .and().authorizeRequests()
            .antMatchers(adminContextPath + "/assets/**",
                         adminContextPath + "/login",
                         adminContextPath + "/actuator/**")
            .permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin().loginPage(adminContextPath + "/login")
            .successHandler(successHandler).and()
            .logout().logoutUrl(adminContextPath + "/logout")
            .and()
            .httpBasic().and()
        // @formatter:on
    }
}
```

```
        .csrf()
        .disable();
    // @formatter:on
}
}
```

- 在对应的 pig-monitor-dev.yml 配置用户

pig 默认的登录用户 pig/pig，可以参考配置文件加解密章节

```
spring:
  security:
    user:
      name: ENC(8Hk2ILNJM8UTOuW/Xi75qg==)      # pig
      password: ENC(o6cuPFFUevmTbkmBnE670w====) # pig
```

监控模块使用之 web 展示实时日志

开启后可以在浏览器实时查看当前日志输出

pig 默认没有开启 Logfile viewer

以 UPMS 模块为例

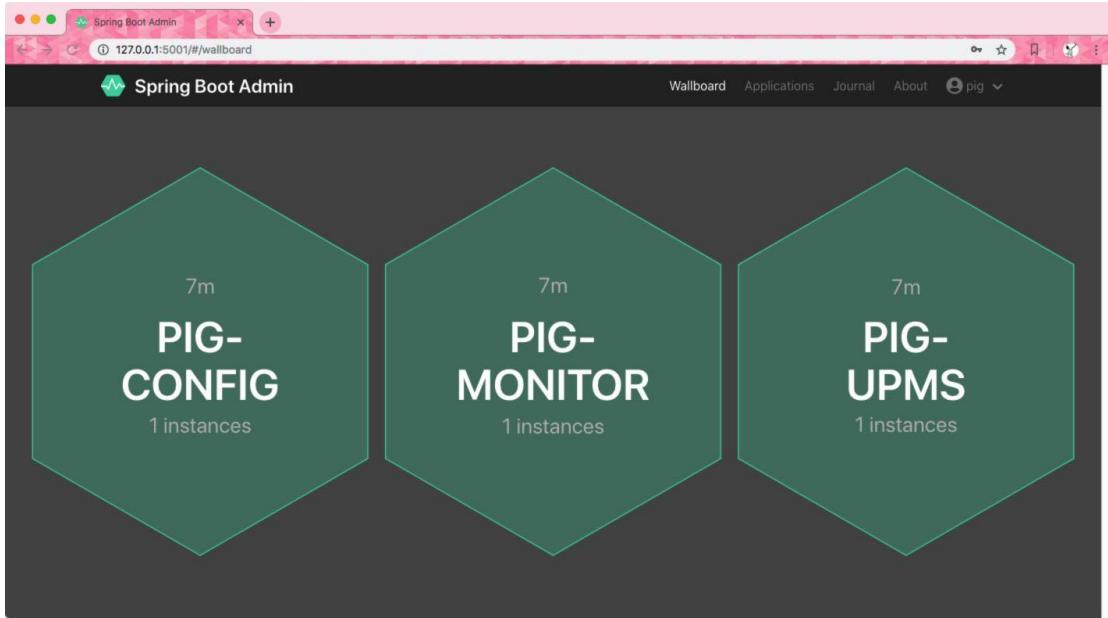
在 bootstrap.yml 配置 logging.file 的配置即可开启 Logfile viewer

```
# 配置 Logfile Viewer

logging:
  file: logs/${spring.application.name}/debug.log
```

## 启动 pig-monitor

- 选中 PIG-UPMS 服务



The top screenshot shows the Spring Boot Admin interface for the PIG-UPMS application (id: f6ad71712115). The left sidebar has tabs: Insights, Details (selected), Metrics, Environment, Configuration Properties, Scheduled Tasks, Logging (selected), JVM, Web, and Audit Log. The main area has tabs: Info, Health, Metadata, Process, and Threads. The Info tab shows 'No info provided.' The Health tab shows 'Instance' status as 'UP'. The Metadata tab shows 'management.port' as 4000. The Process tab shows PID 11178, UPTIME 0d 0h 8m 11s, PROCESS CPU USAGE 0.01, SYSTEM CPU USAGE 0.23, and CPUS 4. The Threads tab shows 67 LIVE threads, 29 DAEMON threads, and a PEAK LIVE of 70.

The bottom screenshot shows the log file for the PIG-UPMS application. The left sidebar has tabs: Insights, Logging (selected), Logfile (selected), Loggers, JVM, Web, and Audit Log. The main area displays a long list of log entries from February 21, 2019, at 12:53:25. The logs include INFO messages from DiscoveryClient, EurekaAutoServiceRegistration, PigAdminApplication, DispatcherServlet, FrameworkServlet, EpollProvider, KqueueProvider, and ConfigClusterResolver, along with RMI and TCP connection logs.

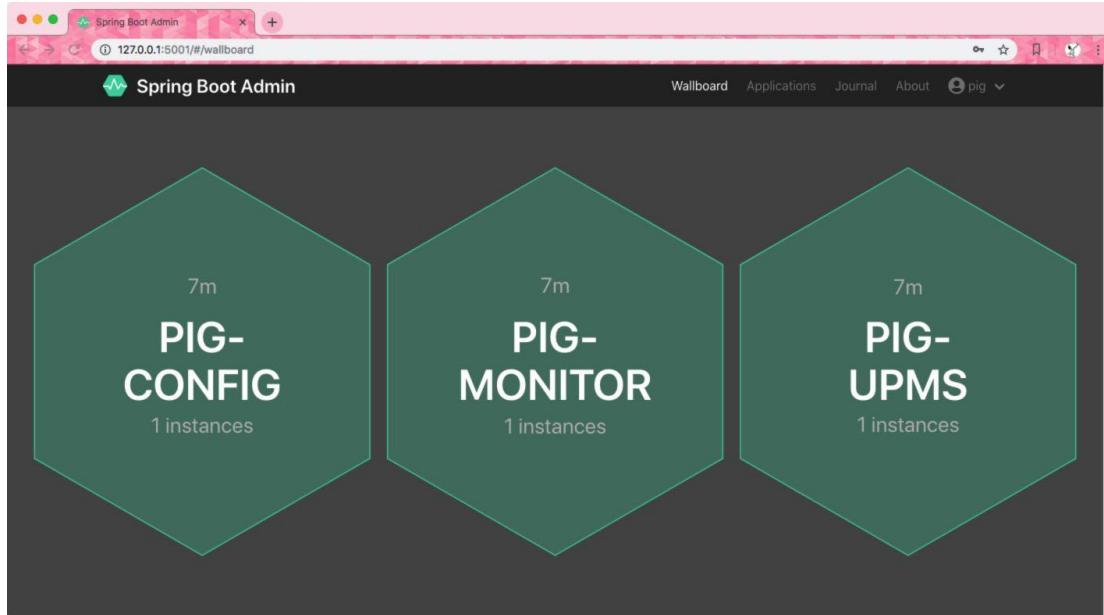
## 监控模块使用之动态日志级别

pig 默认的输出的日志级别是 **INFO**, 生产中建议使用 **ERROR** 级别。

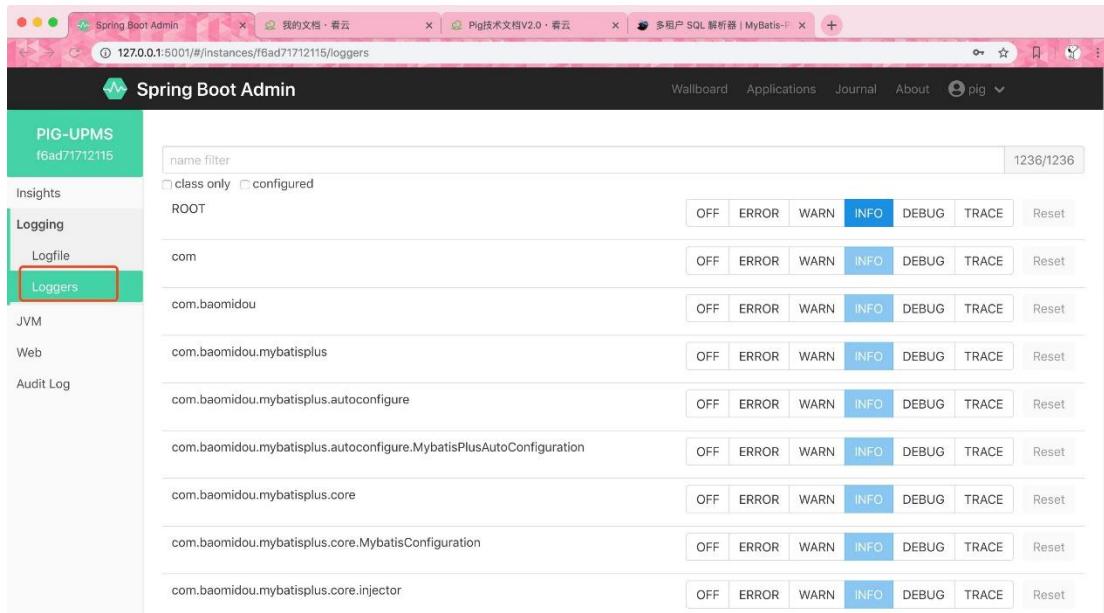
当我们遇到某些 **bug** 时, 动态调整 **log** 输出日志非常有效, **spring boot admin** 提供了这样的功能。

## 启动 pig-monitor

- 选中 PIG-UPMS 服务



- 选择 Loggers, 即可实现动态切换日志级别的需求



## 前端开发

配置 npm 镜像

## 将 Npm 的源替换成淘宝的源

---

在墙内久了，难免会碰到撞墙的时候，所幸国内也有众多 NPM 镜像可供选择，在大多数情况下我们可以使用国内的源（比如 淘宝 NPM 镜像）去替换官方的源以加快下载包的速度。

### 开始

你可以使用我们定制的 cnpm (gzip 压缩支持) 命令行工具代替默认的 npm:

```
$ npm install -g cnpm --registry=https://registry.npm.taobao.org
```

或者你直接通过添加 npm 参数 alias 一个新命令:

```
alias cnpm="npm --registry=https://registry.npm.taobao.org \
--cache=$HOME/.npm/.cache/cnpm \
--disturl=https://npm.taobao.org/dist \
--userconfig=$HOME/.cnpmrc"
```

### 登录详解

首先你要理解 OAuth 2.0(自行百度把)

1. 用户登录成功时调用——loginByUsername 方法会返回 token

vue 中没有与服务器的交互，所以请求的是/api/user 下的数据

2. 将返回的 token 存到 cookie 里面，在 src/store/modules/user.js——

LoginByUsername 方法

token 为服务端返回的

```
...
LoginByUsername({ commit, state, dispatch }, userInfo) {
...
    commit('SET_TOKEN', token);
...
}
```

3. 在调用同 js 文件里的 SET\_TOKEN 方法，将 token 存到 score 里

```
...
mutations: {
    SET_TOKEN: (state, token) => {
        state.token = token;
        setStore({ name: 'token', content: state.token, type: 'session' })
    },
...
}
```

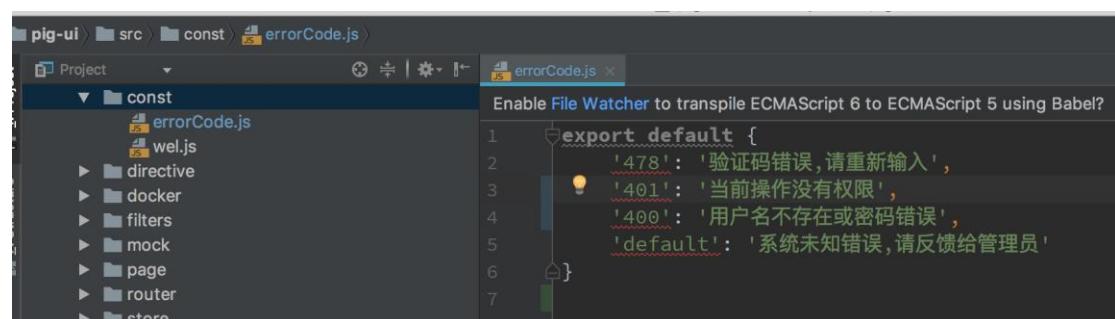
4. 在 ajax 拦截器里，将 token 携带到 header 里，服务器就会接受 token，来区分

那个用户，有那些权限

```
src/router axios.js
...
//HTTPRequest 拦截
axios.interceptors.request.use(config => {
    loadinginstance = Loading.service({
        fullscreen: true
    });
    if (store.getters.token) {
        config.headers['X-Token'] = getToken() // 让每个请求
        携带 token-- ['X-Token'] 为自定义 key 请根据实际情况自行修改
    }
    return config
}, error => {
    console.log('err' + error)// for debug
    return Promise.reject(error)
})
...
```

### 自定义返回码提示

errorCode 直接根据 code，定义如下图。



```
export default {
    '478': '验证码错误,请重新输入',
    '401': '当前操作没有权限',
    '400': '用户名不存在或密码错误',
    'default': '系统未知错误,请反馈给管理员'
}
```

## 服务端定义返回码

### 1. 借助 response 对象

```
response.setCharacterEncoding(CommonConstant.UTF8);
response.setContentType(CommonConstant.CONTENT_TYPE);
R<String> result = new R<>(e);
response.setStatus(478);
printWriter = response.getWriter();
printWriter.append(objectMapper.writeValueAsString(result));
```

### 2. 借助 springMVC

```
@RequestMapping(value="/response/entity/headers", method=RequestMethod.GET)
public ResponseEntity<String> responseEntityCustomHeaders() {

    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.TEXT_PLAIN);
    return new ResponseEntity<String>("The String ResponseBody
with custom header Content-Type=text/plain",
        headers, HttpStatus.OK);
}
```

## 按钮权限控制

# pig 如何控制菜单权限控制

后台声明一个按钮

---

在后台菜单管理中给指定菜单添加 按钮 节点 需要指定 权限标志

例如： sys\_user\_add

## 前端控制

---

前端主要使用 **vuex** 保存用户的权限信息，然后通过 **v-if** 判断是否有权限，如果有权限就渲染这个 **dom** 元素。

我们以 用户管理页面来讲解

```
//按钮 v-if 使用
<el-table-column align="center" label="操作">
    <template slot-scope="scope">
        <el-button v-if="sys_user_upd" size="small" type="success" @click="handleUpdate(scope.row)">编辑
        </el-button>
        <el-button v-if="sys_user_del" size="small" type="danger" @click="Deletes(scope.row)">删除
        </el-button>
    </template>
</el-table-column>

// 变量初始化
created() {
    this.getList();
    this.sys_user_add = this.permissions["sys_user_add"];
    this.sys_user_upd = this.permissions["sys_user_upd"];
    this.sys_user_del = this.permissions["sys_user_del"];
},
// 从 vuex 获取保存的 permissions
computed: {
    ...mapGetters(["permissions"])
}

//permissions 获取
getUserInfo(state.token).then(response => {
    commit('SET_PERMISSIONS', data.permissions)
})
```

## 后端权限控制

---

通过获取用户菜单列表，和请求的地址和请求方法对比判断有没有权限

```
public boolean hasPermission(HttpServletRequest request, Authentication authentication) {
    Object principal = authentication.getPrincipal();
    List<SimpleGrantedAuthority> grantedAuthorityList = (List<SimpleGrantedAuthority>) authentication.getAuthorities();
    boolean hasPermission = false;

    if (principal != null) {
        if (CollectionUtil.isEmpty(grantedAuthorityList)) {
            return hasPermission;
        }

        Set<MenuVo> urls = new HashSet<>();
        for (SimpleGrantedAuthority authority : grantedAuthorityList) {
            urls.addAll(menuService.findMenuByRole(authority.getAuthority()));
        }

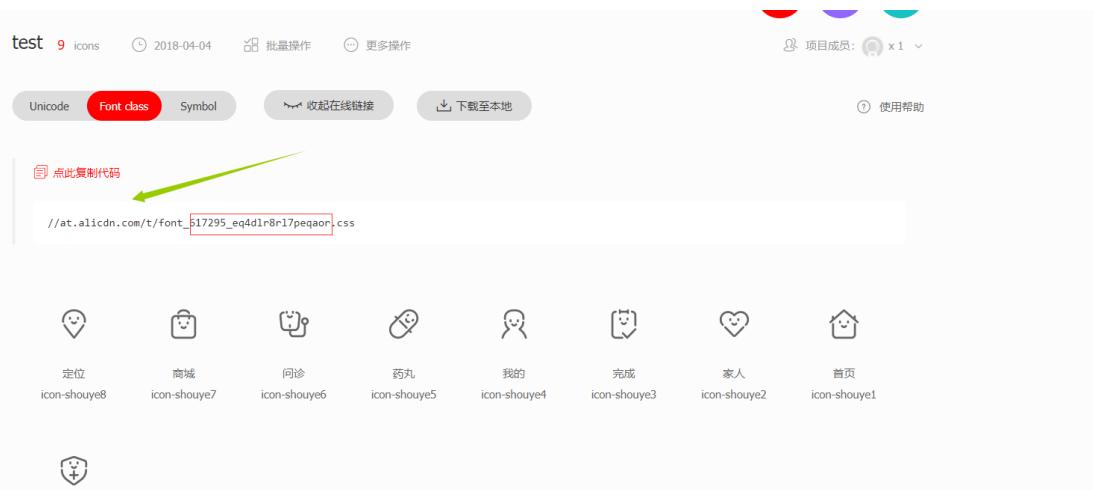
        for (MenuVo menu : urls) {
            if (StringUtils.isNotEmpty(menu.getUrl()) && antPathMatcher.match(menu.getUrl(), request.getRequestURI())
                && request.getMethod().equalsIgnoreCase(menu.getMethod())) {
                hasPermission = true;
                break;
            }
        }
    }
    return hasPermission;
}
```

## 图标引入

步骤 1 :先去阿里巴巴图标库注册一个账号

[阿里巴巴图标库](#)

步骤 2 :完后选择自己喜欢的图标加入到项目中，点击生成在线链接



步骤 3 :图标的加载

将红色框中的部分复制项目中，也就是‘617295\_eq4dlr8rl7peqaor’在  
/src/config/env.js

的 iconfontVersion 的数组中

```
let iconfontVersion = [ '567566_sch40o867ogk3xr', '617295_eq4dlr8rl7peqaor' ];
```

第一个数组的图标不能删除，那是支持 avue 框架的全局图标，如果多个图标库依次添加到数据中即可

步骤 4 :图标的调用

输入在图标库中图标的名称即可

```
<i class="icon-bofangqi-suoping"></i>
```

ps，如果点击更新 URL，更新 env.js

## 生产部署

### Docker Compose 部署

Docker Compose 是 Docker 官方编排项目之一，负责快速的部署分布式应用。

## 安装 Docker (centos7)

---

```
# 更新 yum 源  
yum update  
  
#安装 Docker  
yum -y install docker  
  
#启动 Docker 后台服务  
service docker start  
  
#测试运行 hello-world,由于本地没有 hello-world 这个镜像, 所以会下载一个 hello-world 的镜像, 并在容器内运行。  
docker run hello-world
```

## 安装 docker-compose

---

```
$ sudo curl -L https://github.com/docker/compose/releases/download/1.20.1/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose  
$ sudo chmod +x /usr/local/bin/docker-compose
```

## pig 打包

---

- pig 根目录

```
mvn clean install -Dmaven.test.skip=true
```

```
1. fish /Users/engleng/work/my/open-source/pig (fish)
tor/2.0.2/pig-monitor-2.0.2.jar
[INFO] Installing /Users/engleng/work/my/open-source/pig/pig-visual/pig-monitor
/pom.xml to /Users/engleng/env/repository/com/pig4cloud/pig-monitor/2.0.2/pig-m
onitor-2.0.2.pom
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] pig ..... SUCCESS [ 0.273 s]
[INFO] pig-eureka ..... SUCCESS [ 4.536 s]
[INFO] pig-config ..... SUCCESS [ 0.801 s]
[INFO] pig-common ..... SUCCESS [ 0.007 s]
[INFO] pig-common-core ..... SUCCESS [ 1.336 s]
[INFO] pig-gateway ..... SUCCESS [ 1.564 s]
[INFO] pig-upms ..... SUCCESS [ 0.008 s]
[INFO] pig-upms-api ..... SUCCESS [ 1.297 s]
[INFO] pig-common-security ..... SUCCESS [ 0.905 s]
[INFO] pig-auth ..... SUCCESS [ 0.925 s]
[INFO] pig-common-log ..... SUCCESS [ 0.447 s]
[INFO] pig-upms-biz ..... SUCCESS [ 1.311 s]
[INFO] pig-visual ..... SUCCESS [ 0.005 s]
[INFO] pig-codegen ..... SUCCESS [ 1.075 s]
[INFO] pig-monitor ..... SUCCESS [ 0.778 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 16.038 s
[INFO] Finished at: 2019-02-21T16:35:16+08:00
[INFO] Final Memory: 90M/696M
[INFO] -----
[~] ~w/m/o/pig on dev x 16:35:16
```

- 压缩 pig 整个工程上传到 docker 宿主机
- 执行 docker-compose 命令

```
# 构建镜像
docker-compose build

# 启动容器（-d 后台启动，建议第一次不要加，方便看错误）
docker-compose up -d
```

等待 3 分钟

---

访问 Centos7 IP:8761 查看 eureka 状态，确定所有服务全部启动。

## 总结

---

1. 服务端已启动完毕，前端请参考下一章节《前端部署》
2. 不要和开发环境一样，修改容器 hosts,docker-compose 会根据容器名称自动处理

## 前端部署

### 打包

#### npm 打包

```
# 执行打包命令  
npm run build
```

#### 打包过程

webpack 会生成相应的目录结构(压缩和混淆的代码)

#### 打包产物



```
1. fish /Users/lengleng/work/my/open-source/pig-ui (fish)  
xd ~ /w/m/o/pig-ui on dev x ls  
README.md node_modules public yarn.lock  
babel_config.js package-lock.json src  
dist package.json vue.config.js  
xd ~ /w/m/o/pig-ui on dev x [ ] 13:21:14  
13:21:16
```

### nginx

#### nginx 代理，前端请求，解决跨域

---

在生产中，我们建议使用 nginx 代理前端请求，解决跨域。

并且把上步打包 dist 目录的文件部署到 nginx

#### nginx.conf

---

```
location ^~/admin/ {
```

```
proxy_pass  http://127.0.0.1:9999/admin/;
}

location ^~/auth/ {
proxy_pass  http://127.0.0.1:9999/auth/;
}

location ^~/gen/ {
proxy_pass  http://127.0.0.1:9999/gen/;
}
```

以上配置类似于，vue-cli 的 proxy 配置

## pig 演示环境的配置

```
server {
    listen 80;
    server_name xxxxx.com;

    # 讲打包好的 dist 目录文件，放置到这个目录下
    root /data/pig-ui/;

    location ~* ^/(code|auth|admin|gen) {
        proxy_pass http://127.0.0.1:9999;
        #proxy_set_header Host $http_host;
        proxy_connect_timeout 15s;
        proxy_send_timeout 15s;
        proxy_read_timeout 15s;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

# 微服务资源

PPT 整理

Spring Cloud 中国社区文档

- 2017-0312-Spring Cloud 中国社区成都站 PPT

- 2017-0409-Spring Cloud 中国社区上海站 PPT
- 2017-0506-Spring Cloud 中国社区北京站 PPT
- 2017-0610-Spring Cloud 中国社区深圳站 PPT
- 2017-0820-Spring Cloud 中国社区网关会议 PPT
- Spring Cloud 与摩拜单车 Mobike-范文通.pdf

[Spring Cloud 中国社区线下沙龙文档](#)

## Arcm PPT 分享

- 01.keynote 主题演讲
- 02.数据库架构
- 03.Fintech 技术突围之道（解决方案专场）
- 04.大前端技术与管理
- 05.深入机器学习
- 06.架构升级与优化
- 07.金融应用架构
- 08.互联网产品与创业
- 09.互联网视频技术架构：优化和创新
- 10.云化架构的创新实践（阿里技术专场）
- 11.架构创新与演进（解决方案专场）
- 12.业务系统架构的蜕变与进化
- 13.大数据平台架构
- 14.人工智能与业务应用

- 15.新一代 DevOps
- 16.个性化智能广告系统
- 17.国际化架构设计
- 18.大数据与云原生（解决方案专场）
- 19.容器云平台运维（解决方案专场）
- 20.微服务架构
- 21.内容分发与精准推荐
- 22.移动开发工程化实践
- 23.工程师文化与团队建设
- 24.深度培训

链接:<https://pan.baidu.com/s/1iTUMCGc7kJ3kLYH7vWEEFA> 密码:iddl

链接: [https://pan.baidu.com/s/1fr0S\\_vFYvrTOzR9sOG1hTA](https://pan.baidu.com/s/1fr0S_vFYvrTOzR9sOG1hTA) 提取码: ixnt

## 视频整理

部分视频链接由于百度和谐，分享了里面失效，有需要问我要~

## IDEA 学习视频

---

代码生成技术

批量编辑技术

代码调试技术

代码智能修复技术

智能上下文关联技术

代码重构技术

高效的版本控制管理技术

<https://www.imooc.com/comment/924>

## Spring Boot 学习视频

---

springboot 某智

龙果 2017 年 spring boot 入门实战视频教程

链接:<https://pan.baidu.com/s/1cgQiWnDBbgeE7GJqJuGjqg> 密码:45ak

## Spring Cloud 学习视频

---

Spring Cloud (中文)

链接:<https://pan.baidu.com/s/1uAghMrE1femsLNIVoFFVbQ> 密码:3ctz

SpringCloud(慕课)

链接:<https://pan.baidu.com/s/17h-veRySqqVltqusyFj7IA> 密码:kjw9

## Spring Security 学习视频

---

链接:<https://pan.baidu.com/share/init?surl=dE67zFz> 密码:xkbs

## 分布式事物学习视频

---

链接:<https://pan.baidu.com/s/1g6LX5nNNOeGQ1Mn2NS3AfA> 密码:z0za

## 从无到有搭建中小型互联网公司后台服务架构与运维架构

---

链接:<https://pan.baidu.com/s/1v5dMnR23PrhrPZ6pqrhDaw> 密码:pt47

# 亿级流量电商详情页系统的大型高并发与高可用缓存架构实战

---

链接:<https://pan.baidu.com/s/19HMLvlwgQEIldeGFvgin5g> 密码:32sz

## 高可用 RabbitMQ

---

链接:<https://pan.baidu.com/s/15M4H4mTSmrEupuIH3e97YA> 密码:eyvu

## 高可用 MySQL

---

链接:<https://pan.baidu.com/s/1f9B78W4suQJQgycktnLgtw> 密码:0gfd

## Docker

---

链接:<https://pan.baidu.com/s/1D0DQfrZsCDUOHL423Bdvhw> 密码:ds3y

链接:<https://pan.baidu.com/s/1tczTjmX9N1hLDyApMQDg0A> 密码:2ack

链接: <https://pan.baidu.com/s/1MaPWwlKVvkS6CTIj-EvmTQ> 密码: 7l4t

## Nginx (必看)

---

链接: <https://pan.baidu.com/s/1-TZv7T1Kn1AECEXeRhh8jw> 提取码: cweP

博客整理

## 收藏栏

## 工具

[json](#)

[文本](#)

[Smallpdf.com – 您所有 PDF 问题的免费解决方案](#)

[ProcessOn](#)

[全网 VIP 视频在线解析](#)

[Redis 命令参考 — Redis 命令参考](#)

[在线工具](#)

[爱莫助手](#)

[XMind Cloud](#)

[Eureka Server](#)

[HTML 在线运行](#)

[字帖](#)

[Mac 软件破解库 · 看云](#)

[ASCIIFlow Infinity](#)

[XDOC 云服务 - 云文档、云报表、云表单](#)

[临时邮箱-邮箱大全](#)

[可以自己定义的邮箱](#)

[Bootstrap 可视化布局系统](#)

[ShowDoc](#)

[知笔墨](#)

[冷冷的书籍 - 北半球](#)

[Readhub](#)

收藏

其他

[微信公众平台开发者文档](#)

[MySQL 索引原理及慢查询优化 -](#)

[ifeve.com](#)

[Pixabay](#)

[七牛云](#)

[ztree](#)

[JavaFX](#)

[看云](#)

[Insdep Theme - EasyUI 美化主题 JQuery EasyUI Of Insdep theme](#)

[sonar 常见错误以及处理方案](#)

[Redis 如何分布式，来看京东金融的设计与实践](#)

[分布式环境下限流方案的实现 redis RateLimiter Guava,Token Bucket, Leaky](#)

[Bucket - 沧海一滴 - 博客园](#)

[使用 Redis 实现分布式锁及其优化 | Mz's Blog](#)

限流

[Activiti 工作流](#)

[CentOS6.5 下 RabbitMQ 安装](#)

[程序员你为什么这么累？](#)

[IntelliJ IDEA For Mac 快捷键 - IntelliJ IDEA 使用教程 - 极客学院 Wiki](#)

## java

[图解集合](#)

[Java 线程](#)

[聊聊高并发](#)

[Java Concurrency](#)

[无信不立 - 博客园](#)

[Java 面试题全集](#)

## VUE

[VUE2](#)

[Vue.js](#)

[Vue 2.0 Admin 后台管理模板对比](#)

[Juicy](#)

[fetch](#)

[javascript - vuejs2.0 的 element-ui 组件 select 选择器无法显示选中的内容 - SegmentFault](#)

[Hystrix/BasicCollapserTest.java at master · Netflix/Hystrix](#)

微服务

## spring cloud

[Spring Cloud 中文网-官方文档中文版](#)

[Spring Cloud Dalston 中文文档 参考手册 中文版](#)

[周立/spring-cloud-docker-microservice-book-code - 码云](#)

[Category: Eureka | 芸道源码 —— 纯源码解析 BLOG](#)

[Spring Cloud](#)

[Spring Cloud git](#)

[spring cloud-江南](#)

[史上最简单的 SpringCloud 教程 | 终章 - 方志朋的专栏 - CSDN 博客](#)

[JWT 在 Spring Cloud 中的使用](#)

[如何架构一个合适的企业 API 网关](#)

[Spring Cloud](#)

[Spring Cloud](#)

[spring cloud sleuth | 网易乐得技术团队](#)

[微服务实践 - MSA 的关键架构概念](#)

[微服务-铁汤博客](#)

[JHipster](#)

[极客时间 | 微服务架构核心 20 讲](#)

## spring boot

[Spring Boot](#)

[springboot · GitBook](#)

[Spring Boot2.0](#)

[spring-boot/spring-boot-samples at v2.0.0.RELEASE · spring-projects/spring-boot](#)

[Spring Security 参考手册](#)

## docker

[Docker](#)

[src/page/index.vue · 云集汇通开发团队/vue-xinjiang - 码云 - 开源中国](#)

[CentOS Docker 安装 | 菜鸟教程](#)

[写在最前面 | shipyard 中文文档](#)

[Docker 可视化管理工具 shipyard 安装 - Yang - CSDN 博客](#)

[shipyard](#)

[CentOS 7 配置 Docker 远程 API 访问 - 老胡的笔记 - CSDN 博客](#)

[Portainer](#)

[搭建私有仓库 - Humpback](#)

## spring

[jetty 配置 https](#)

[Spring Projects](#)

[跟我学 Shiro](#)

## 其他

[碧桂园分布式事务](#)

[程立谈大规模 SOA 系统](#)

[芋道源码](#)

[CentOS 7 安装配置分布式文件系统 FastDFS 5.0.5 服务器应用 Linux 公社-Linux 系统门户网站](#)

[凤凰牌老熊的博客 | Shamphone Blog](#)

[elastic-job 与 spring boot 集成 demo - CSDN 博客](#)

[使用 FastDFS 搭建图片服务器单实例篇-老猫 1981-51CTO 博客](#)

[Redis Cluster](#)

[spring security 的一些文档](#)

[JHipster - Generate your Spring Boot + Angular apps!](#)

[安装 · jhipster 开发笔记](#)

## service mesh

[Introduction · Istio 官方文档中文版](#)

pinpoint

[分布式性能管理监控工具 Pinpoint 入门视频课程 共 3 课时-51CTO 学院](#)

[PinPoint 分布式全链路监控搭建 清屏网 在线知识学习平台](#)

[Pinpoint 安装部署 · Issue #166 · ameizi/DevArticles](#)

[pinpoint 安装部署 - 猴子请来的救兵 - 博客园](#)

spring security

[Spring Security | 徐靖峰|个人博客](#)

[配置表单登录\\_Spring Security 4 官方文档中文翻译与源码解读教程\\_田守枝 Java 技术博客](#)

[spring security](#)

[spring cloud 微服务整合 oauth2 | 小明啊喂](#)

[Categories — Blog 龙飞](#)

[Spring Security OAuth2](#)

[通用密码加密 · Spring Security Tutorial](#)

[新标签页](#)

[spring security oauth2 password 授权模式 - code-craft - SegmentFault 思否](#)

Python

[Scrapy 入门教程 \(Scrapy Tutorial\) ·](#)

[Python 教程 - 廖雪峰的官方网站](#)

Golang

[Introduction | go 语言入门](#)

pay

[如何做一个对账系统](#)

[高吞吐消息网关的探索与思考](#)

[最全支付系统设计](#)

[以太坊](#)

[区块链](#)

[\[中文\] 以太坊白皮书](#)

oAuth2 开发指南

## 介绍

---

这是 [OAuth 2.0](#) 支持的用户指南。对于 OAuth 1.0 来说，一切都是不同的，所

以 [see its user guide.](#)

这个用户指南分为两个部分，第一个是 OAuth 2.0 提供者，第二个是 OAuth 2.0 客户端。对于提供者和客户端，示例代码的最佳来源是 [integration tests](#) 和 [sample apps](#).

## OAuth 2.0 提供者

---

OAuth 2.0 提供者机制负责公开 OAuth 2.0 受保护的资源。配置包括建立 OAuth 2.0 客户端，可以独立地或代表用户访问其受保护的资源。提供者通过管理和验证用于访问受保护资源的 OAuth 2.0 令牌来实现这一点。在适用的情况下，提供者还必须为用户提供一个接口，以确认客户端可以访问受保护的资源(即确认页面)。

## OAuth 2.0 提供者实现类

---

OAuth 2.0 中的提供者角色实际上是在授权服务和资源服务之间进行划分的，虽然它们有时是在同一个应用程序中，但在 [Spring Security OAuth](#) 中，您可以选择在两个应用程序之间进行拆分，并且拥有多个共享授权服务的资源服务。令牌的请求由 [Spring MVC](#) 控制器端点来处理，而对受保护资源的访问由标准 [Spring](#) 安全请求过滤器处理。为了实现 OAuth 2.0 授权服务器，Spring 安全过滤器链中需要以下端点：

- `[AuthorizationEndpoint][AuthorizationEndpoint]` 用于服务请求的授权。默认 URL: /oauth/authorize.

- [TokenEndpoint][TokenEndpoint] 用于服务访问令牌的请求。默认

URL: /oauth/token.

下面的过滤器需要实现 OAuth 2.0 资源服务器:

- [OAuth2AuthenticationProcessingFilter][OAuth2AuthenticationProcessingFilter] 用于为请求提供一个经过身份验证的访问令牌进行身份验证。

对于所有 OAuth 2.0 提供者特性，可以使用特殊的 Spring

OAuth @Configuration 适配器配置简化配置。还有一个用于 OAuth 配置的 XML

命名空间，这个模式在 [\[http://www.springframework.org/schema/security/spring-](http://www.springframework.org/schema/security/spring-security-oauth2.xsd)

[\[oauth2.xsd\]](http://www.springframework.org/schema/security/oauth2.xsd). 命名空间

是 <http://www.springframework.org/schema/security/oauth2>.

## 授权服务器配置

---

在配置授权服务器时，您必须考虑客户端用于从最终用户(例如授权代码、用户凭证、刷新令牌)中获得访问令牌的授权类型。服务器的配置用于提供客户端详细信息和服务和令牌服务的实现，并在全局范围内启用或禁用该机制的某些切面。但是，请注意，每个客户端都可以配置特定的权限，以便能够使用某些授权机制和访问授权。也就是说，仅仅因为您的提供者被配置为支持“客户端凭证”授予类型，并不意味着特定的客户端被授权使用该授予类型。

@EnableAuthorizationServer 注释用于配置 OAuth 2.0 授权服务器机制，以及任何实现 AuthorizationServerConfigurer 的@ bean(有一个方便的适配器实现提供了

空方法的实现)。下面的特性被委托给由 Spring 创建的配置器，并传递给

`AuthorizationServerConfigurer`:`

- `ClientDetailsServiceConfigurer`: 定义客户端详细信息服务的配置程序。客户端细节可以被初始化，也可以直接引用现有的存储。
- `AuthorizationServerSecurityConfigurer`: 定义令牌端点上的安全约束。
- `AuthorizationServerEndpointsConfigurer`: 定义授权和令牌端点和令牌服务。

提供者配置的一个重要方面是授权代码被提供给 OAuth 客户端（在授权代码授予中）。授权代码由 OAuth 客户端获得，它将终端用户引导到一个授权页面，用户可以在其中输入她的凭证，从而导致从提供者授权服务器重新定向到带有授权代码的 OAuth 客户端。在 OAuth 2 规范中详细说明了这一点。

在 XML 中，有一个`<authorizationserver />`元素，它以类似的方式用于配置 OAuth 2.0 授权服务器。

## 配置客户端详细信息

`ClientDetailsServiceConfigurer`(来自您的 `AuthorizationServerConfigurer` 的回调)

可以用于定义客户端详细信息服务的内存或 JDBC 实现。客户端的重要属性是：

- `clientId`:(必需)客户 id。
- `secret`: (需要信任的客户)客户的密钥，如果有的话。
- `scope`: 客户受限制的范围。如果作用域是未定义的或空的(默认的)，客户端不受范围限制。

- `authorizedGrantTypes`: 授权给客户端使用的授权类型。默认值是空的。
- `authorities`: 授权给客户的部门。(通常是 Spring Security authorities).

客户端详细信息可以在运行的应用程序中更新，通过直接访问底层存储（例如 `JdbcClientDetailsService` 案例中的数据库表）或通过 `ClientDetailsManager` 接口（`ClientDetailsService` 的两个实现都实现了）。

注意：JDBC 服务的模式并没有与库一起打包（因为在实践中可能会使用太多的变体），但是有一个例子可以从 [\[test code in github\]](#) 开始。

## 管理令牌

`[AuthorizationServerTokenServices][AuthorizationServerTokenServices]` 接口定义了管理 OAuth 2.0 令牌所必需的操作。请注意以下几点：

- 当创建访问令牌时，必须存储身份验证，以便稍后接受访问令牌的资源可以引用它。
- 访问令牌被用来加载用于授权其创建的身份验证。

在创建您的 `AuthorizationServerTokenServices` 实现时，您可能需要考虑使用具有许多策略的 `DefaultTokenServices` 来更改访问令牌的格式和存储。默认情况下，它通过随机值创建令牌，并处理所有的东西，除了它委托给 `TokenStore` 的令牌的持久性。默认存储是[在内存中实现的]`[InMemoryTokenStore]`，但还有一些其他实现可用。下面是对每种实现方式的一些讨论。

- 默认的 `InMemoryTokenStore` 对于单个服务器来说是完美的(例如，在失败的情况下，低流量和没有热交换到备份服务器)。大多数项目都可以从这里开始，并可能在开发模式中使用这种方式，从而能很容易的启动一个没有依赖关系的服务器。
- `JdbcTokenStore` 和 `JDBC` 版本是同一种东西，它使用关系数据库来存储令牌数据。如果您可以在服务器之间共享一个数据库，那么可以使用 `JDBC` 版本，如果只有一个服务器，则可以扩展相同服务器的实例，如果有多个组件，则可以使用授权和资源服务器。为了使用 `JdbcTokenStore`，您需要将 "spring-jdbc" 配置到 classpath 中。
- [JSON Web Token \(JWT\) version](#) 将所有关于 `grant` 的数据编码到令牌本身(因此没有任何后端存储，这是一个重要的优势)。一个缺点是，您不能很容易地撤销访问令牌，因此它们通常在短时间内被授予，而撤销则在刷新令牌中处理。另一个缺点是，如果您在其中存储了大量用户凭证信息，则令牌可以变得相当大。`JwtTokenStore` 并不是真正的“存储”，因为它没有保存任何数据，但是它在 `DefaultTokenServices` 中扮演了转换 between 令牌值和身份验证信息的角色。  
注意：`JDBC` 服务的模式并没有与库一起打包（因为在实践中可能会用到太多的变体），但是有一个示例可以从 `github` 的 [\[test code in github\]](#) 开始。确保 `@EnableTransactionManagement` 能够防止客户端应用程序在创建令牌时争用相同的行。还要注意，示例模式有显式的主键声明——在并发环境中这些声明也是必需的。

## JWT 令牌

要使用 JWT 令牌，您需要在授权服务器中使用 `JwtTokenStore`。资源服务器还需要能够解码令牌，这样 `JwtTokenStore` 就依赖于 `JwtAccessTokenConverter`，并且授权服务器和资源服务器都需要相同的实现。该令牌是默认签名的，并且资源服务器还必须能够验证签名，因此需要与授权服务器(共享私钥或对称密钥)相同的对称(签名)密钥，或者它需要与授权服务器(公私或非对称密钥)中的私钥(签名密钥)相匹配的公钥(验证器密钥)。公钥(如果可用)由 `/oauth/token_key` 端点上的授权服务器公开，该端点在默认情况下是安全的，具有访问规则“`denyAll()`”。您可以通过向 `AuthorizationServerSecurityConfigurer` 中注入标准的 SpEL 表达式来打开它。“`permitAll()`”可能已经足够了，因为它是一个公钥。

要使用 `JwtTokenStore`，您需要在 `classpath` 上使用“`Spring -security-jwt`”(您可以在相同的 `github` 存储库中找到它，它与 `Spring OAuth` 相同，但有不同的发布周期)。

## 授权类型

`AuthorizationEndpoint` 支持的授权类型可以通过 `AuthorizationServerEndpointsConfigurer` 配置。默认情况下，除密码外，所有的授权类型都是受支持的(请参阅下面关于如何切换的详细信息)。以下属性影响授权类型：

- `authenticationManager`: 通过注入一个 `AuthenticationManager` 来打开密码授权。

- `userDetailsService`: 如果您注入了一个 `UserDetailsService`, 或者在全局上配置了一个(例如, 在 `GlobalAuthenticationConfigurer` 中), 那么刷新令牌授权将包含对用户详细信息的检查, 以确保帐户仍然处于活动状态。
- `authorizationCodeServices`: 为身份验证代码授予定义授权代码服务(`AuthorizationCodeServices`)的实例)。
- `implicitGrantService`: 在 `implicit` 授权期间管理状态。
- `tokenGranter`: the `TokenGranter` (完全控制授予和忽略上面的其他属性)

在 XML `grant` 类型中, 包括 `authorization-server` 元素。

配置的端点 `url`

`AuthorizationServerEndpointsConfigurer` 有一个 `pathMapping()` 方法。它需要两个参数:

- 端点的默认(框架实现)URL 路径。
- 需要的自定义路径(以“/”开头)

框架提供的路径是 `/oauth/authorize` (授权端点), `/oauth/token` (令牌端点), `/oauth/confirm_access` (用户在这里获得批准), `/oauth/error` (用于呈现授权服务器错误), `/oauth/check_token` (用于资源服务器解码访问令牌。), and `/oauth/token_key` (如果使用 JWT 令牌, 公开公钥进行令牌验证).

N.B. 应该使用 Spring Security 保护授权端点`/oauth/authorize`(或其映射的替代), 以便只有经过身份验证的用户才能访问它。例如: 使用标准的 Spring Security `WebSecurityConfigurer`:

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests().antMatchers("/login").permitAll()  
        .and()  
            // default protection for all resources (including /oauth/authorize)  
            .authorizeRequests()  
            .anyRequest().hasRole("USER")  
            // ... more configuration, e.g. for form login  
}
```

注意:如果您的授权服务器也是一个资源服务器, 那么还有另一个安全过滤器链, 它的优先级较低, 控制了 API 资源。 对于那些需要通过访问令牌来保护的请求, 您需要它们的路径不能与主用户所面对的过滤器链中的那些相匹配, 所以一定要包含一个请求 matcher, 它只挑选出上面的 `WebSecurityConfigurer` 中的非 Api 资源。

在`@Configuration` 支持中, 使用客户端机密的 HTTP 基本身份验证, Spring OAuth 默认为您保护令牌端点。这不是 XML 中的情况(因此应该明确地保护它)。在 XML 中, `<authorization-server/>` 元素有一些属性可用于以类似的方式更改缺省端点 url。必须显式地启用 `/check_token` 端点(使用 `check-token-enabled` 的属性)。

## 定制用户界面

---

大多数授权服务器端点主要是由机器使用的, 但是有一些资源需要一个 UI, 而这些资源是 GET for `/oauth/confirm_access` 和 `/oauth/error` 的 HTML 响应。它们在框架中使用了 `whitelabel` 实现, 因此授权服务器的大多数真实实例都希望提供自己的实现, 这样它们就可以控制样式和内容。 您所需要做的就是为这些端点提供一个带有`@RequestMappings` 的 Spring MVC 控制器, 而框架默认值将在调度

程序中降低优先级。 在`/oauth/confirm_access` 端点中，您可以预期一个 `AuthorizationRequest` 绑定到会话，该请求将携带需要获得用户批准的所有数据(默认的实现是 `AuthorizationRequest`，因此要查看那里的起始点以复制)。您可以从该请求中获取所有的数据，并按您喜欢的方式呈现它，然后所有用户需要做的是将批准或拒绝授予的信息发布回 `/oauth/authorize`。请求参数直接传递给 `AuthorizationEndpoint` 中的 `UserApprovalHandler`，这样您就可以随意地解释数据了。 默认的 `UserApprovalHandler` 取决于您是否在 `AuthorizationServerEndpointsConfigurer` 中提供了 `ApprovalStore` (在这种情况下，它是 `ApprovalStoreUserApprovalHandler`) 或 `not` (在这种情况下，它是一个 `TokenStoreUserApprovalHandler`)。标准审批处理程序接受以下内容:

- `TokenStoreUserApprovalHandler`: 通过 `user_oauth_approval` 的一个简单的 yes/no 决策等于“true”或“false”。
- `ApprovalStoreUserApprovalHandler`: 一组 `scope.*` 参数键与所请求的作用域相等。 参数的值可以是“true”或“approved”（如果用户批准了授权），则用户被认为拒绝了该范围。 如果至少有一个范围被批准，那么授权是成功的。

注意:不要忘记将 `CSRF` 保护包含在您为用户呈现的表单中。 `Spring Security` 在默认情况下期望一个名为“\_csrf”的请求参数(它提供了请求属性中的值)。 请参阅 `Spring 安全用户指南`以获得更多信息，或者查看 `whitelabel` 实现以获得指导。

## 执行 SSL

普通 HTTP 可以用于测试，但是授权服务器只能在生产中使用 SSL。 您可以在一个安全的容器或代理的后面运行该应用程序，如果您正确地设置了代理和容器(这与 OAuth2 无关)，那么它应该可以正常工作。 您还可能希望使用 Spring Security `requiresChannel()`约束来保护端点。 对于 /authorize 端点，你要做的是作为你正常的应用程序安全的一部分。对于 /token 端点，在 `AuthorizationServerSecurityConfigurer` 中有一个标记，您可以使用 `sslOnly()`方法进行设置。在这两种情况下，安全通道设置都是可选的，但如果它在不安全的通道上检测到请求，则会导致 Spring Security 重定向到它认为是安全通道的安全通道。

## 自定义错误处理

---

授权服务器中的错误处理使用标准的 Spring MVC 特性，即端点中的 `@ExceptionHandler` 方法。 用户还可以为端点本身提供一个 `WebResponseExceptionTranslator`，这是改变响应内容的最佳方式，而不是改变响应的方式。在授权端点的情况下，在令牌端点和 OAuth 错误视图 (/oauth/error)的情况下，将异常委托给 `HttpMessageConverters`(可以添加到 MVC 配置)。为 HTML 响应提供了 `whitelabel` 错误端点，但是用户可能需要提供一个自定义实现(例如，只需添加一个`@RequestMapping("/oauth/error")` 的 `@Controller`)。

## 将用户角色映射到作用域。

---

有时，限制令牌的范围不仅限于分配给客户端的作用域，还可以根据用户自己的权限来限制令牌的范围。 如果您在 `AuthorizationEndpoint` 中使用 `DefaultOAuth2RequestFactory`，那么您可以设置一个 `flagCheckUserScopes=true`，从而将允许范围限制为与用户角色相匹配的范围。 您还可以将 `OAuth2RequestFactory` 注入到 `TokenEndpoint` 中，但是如果您还安装了一个 `TokenEndpointAuthenticationFilter`(也就是使用密码授权)，那么您只需要在 `HTTP BasicAuthenticationFilter` 之后添加那个过滤器。当然，您也可以实现自己的规则，将范围映射到角色，并安装您自己的 `OAuth2RequestFactory` 版本。如果您使用`@EnableAuthorizationServer`,则 `AuthorizationServerEndpointsConfigurer` 允许您注入自定义的 `OAuth2RequestFactory`，这样您就可以使用该特性来设置一个工厂。

## 资源服务器配置

---

资源服务器(可以与授权服务器或单独的应用程序相同)提供由 `OAuth2` 令牌保护的资源。 Spring OAuth 提供了一个实现此保护的 Spring 安全身份验证过滤器。您可以在 `@Configuration` 类上使用`@EnableResourceServer` 来切换它，并使用 `ResourceServerConfigurer` 配置它（必要时）。他可以配置以下功能：

- `tokenServices`: 定义令牌服务的 bean (`ResourceServerTokenServices` 实例)。
- `resourceId`: 资源的 id(可选，如果存在推荐并将由 auth 服务器验证)。
- 资源服务器的其他扩展点(例如从传入请求中提取令牌的 `tokenExtractor`)。

- 请求受保护资源的请求者(默认为所有)
- 受保护资源的访问规则(默认为“已验证”)
- Spring Security 中 `HttpSecurity` 配置程序允许的受保护资源的其他定制。

`@EnableResourceServer` 注解自动将

`OAuth2AuthenticationProcessingFilter` 类型的过滤器添加到 Spring Security 过滤器链中。

在 XML 中有一个带有 `id` 属性的`<resource-server/>`元素——这是 `servlet` 过滤器的 `bean id`，然后可以手工添加到标准 Spring Security 链中。

您的 `ResourceServerTokenServices` 是与授权服务器的契约的另一半。如果资源服务器和授权服务器都在同一个应用程序中，并且使用

`DefaultTokenServices`，那么您就不必对此进行过多的思考，因为它实现了所有必要的接口，因此它是自动一致的。如果您的资源服务器是一个单独的应用程序，那么您必须确保与授权服务器的功能相匹配，并提供一个知道如何正确解码

`ResourceServerTokenServices`。与授权服务器一样，您可以经常使用 `DefaultTokenServices`，而选择主要通过 `TokenStore`(后端存储或本地编码) 来表示。另一种选择是 `RemoteTokenServices`，它是一个 Spring OAuth 特性(不是规范的一部分)，允许资源服务器通过授权服务器上的 HTTP 资源

`(/oauth/check_token)` 来解码令牌。如果资源服务器中没有大量的流量(每个请求都必须与授权服务器进行验证)，或者如果您有能力缓存结果，那么

`RemoteTokenServices` 是很方便的。要使用 `/oauth/check_token` 端点，您需要在 `AuthorizationServerConfigurer` 中通过更改它的访问规则来公开它(默认为“`denyAll()`”)。例如

```
@Override  
public void configure(AuthorizationServerSecurityConfigurer oauth  
Server) throws Exception {  
    oauthServer.tokenKeyAccess("isAnonymous() || hasAuthority  
('ROLE_TRUSTED_CLIENT')")  
        .checkTokenAccess("hasAuthority('ROLE_TRUSTED_CLIENT')");  
}
```

在这个例子中，我们配置了 /oauth/check\_token 端点和 /oauth/token\_key 端点（因此可信资源可以获得 JWT 验证的公钥）。这两个端点通过使用客户端凭证的 HTTP 基本身份验证保护。

配置一个 **oauthaware** 表达式处理器。

您可能想要利用 Spring Security 的基于表达式的访问控制。表达式处理器将默认在 @enableresourceserver 设置中注册。表达式包括 `#oauth2.clientHasRole`, `#oauth2.clientHasAnyRole`, 和 `_#oath2.denyClient_` 可以根据 oauth 客户端的角色来提供访问(请参阅完整列表的 OAuth2SecurityExpressionMethods )。在 XML 中，您可以使用常规的 `<http/>` 安全配置的 expression-handler 程序元素注册一个 oauth-aware 表达式处理器。

## OAuth 2.0 客户端

---

OAuth 2.0 客户端机制负责访问其他服务器的 OAuth 2.0 保护资源。该配置涉及到建立用户可能访问的相关保护资源。客户端还可能需要为用户提供存储授权代码和访问令牌的机制。

## 受保护的资源配置

可以使用`OAuth2ProtectedResourceDetails` 的 bean 定义来定义受保护的资源(或

“远程资源”)。受保护的资源具有以下属性:

- `id`: 资源的 id。该 id 仅供客户端用于查找资源;在 OAuth 协议中从未使用过。它还被用作 bean 的 id。
- `clientId`: OAuth 客户端 id。这是 OAuth 提供者识别您的客户端的 id。
- `clientSecret`: 与资源有关的 secret。默认情况下，没有 secret 是空的。
- `accessTokenUri`: 提供访问令牌的提供者 OAuth 端点的 URI。
- `scope`: 逗号分隔的字符串列表，指定访问资源的范围。默认情况下，没有指定范围。
- `clientAuthenticationScheme`: 客户端用于验证访问令牌端点的方案。建议值:“http\_basic”和“form”。默认值:“http\_basic”。见 OAuth 2 规范第 2.1 节。

不同的 grant 类型有不同的 `OAuth2ProtectedResourceDetails` 的具体实现(例如，

`ClientCredentialsResource` 用于“client\_credentials”授权类型)。对于需要

用户授权的授权类型，还有一个属性:

- `userAuthorizationUri`: 如果用户需要授权访问该资源，则将重定向用户的 uri。注意，这并不总是必需的，这取决于所支持的 OAuth 2 配置文件。

在 XML 中，有一个`<resource/>`元素，可以用来创建一个

`OAuth2ProtectedResourceDetails` 的 bean。它具有匹配上述所有属性的属

性。

## 客户端配置

对于 OAuth 2.0 客户端，配置是使用`@EnableOAuth2Client` 简化的。它做了两件事：

- 创建一个过滤器 bean(带有 ID `oauth2ClientContextFilter`)来存储当前请求和上下文。在需要进行身份验证的情况下，它管理对 OAuth 身份验证 uri 的重定向。
- 在请求范围内创建一个类型 `AccessTokenRequest` 的 bean。这可以通过授权代码(或隐式)授予客户端来保持与单个用户之间的冲突。

过滤器必须连接到应用程序中(例如，使用 `Servlet` 初始化器或使用相同名称的 `DelegatingFilterProxy` 的 `web.xml` 配置)。

`AccessTokenRequest` 可用于以下的 `OAuth2RestTemplate`:

```
@Autowired  
private OAuth2ClientContext oauth2Context;  
  
@Bean  
public OAuth2RestTemplate sparklrRestTemplate() {  
    return new OAuth2RestTemplate(sparklr(), oauth2Context);  
}
```

`OAuth2ClientContext` 在会话范围内(为您)放置，以保持状态不同的用户分开。如果不这样做，您将不得不在服务器上管理等效的数据结构，将传入的请求映射到用户，并将每个用户与 `OAuth2ClientContext` 的单独实例关联起来。

在 XML 中，有一个带有 `id` 属性的`<client/>`元素—这是一个 `Servlet` 过滤器的 bean `id`，必须在`@Configuration` 案例中映射到 `DelegatingFilterProxy`(具有相同的名称)。

## 访问受保护的资源

一旦您提供了资源的所有配置，现在就可以访问这些资源了。访问这些资源的建议方法是使用 Spring 3 中引入的[`RestTemplate`][`restTemplate`]。Spring Security 的 OAuth 提供了一个扩展的 `RestTemplate`，它只需要提供 [`OAuth2ProtectedResourceDetails`][`OAuth2ProtectedResourceDetails`] 的实例。要使用用户令牌(授权代码授权)，您应该考虑使用`@EnableOAuth2Client` 配置(或 XML 等效的`<oauth:rest-template/>`)，它创建一些请求和会话范围的上下文对象，以便不同用户的请求在运行时不会发生冲突。

一般来说，web 应用程序不应该使用密码授予，因此，如果您能够支持 `AuthorizationCodeResourceDetails`，请避免使用 `ResourceOwnerPasswordResourceDetails`。如果您需要从 Java 客户端获得工作密码，那么使用相同的机制来配置 `OAuth2RestTemplate` 并将凭证添加到 `AccessTokenRequest`(这是一个映射，并且是临时的)，而不是 `ResourceOwnerPasswordResourceDetails` (它在所有访问令牌之间共享)。

## 在客户端持久化令牌

客户端不需要存留令牌，但对于用户来说，在每次重启客户端应用程序时都不需要批准新的令牌授权，这对用户来说是件好事。`ClientTokenServices` 接口定义了为特定用户保存 OAuth 2.0 令牌所需的操作。这里提供了 JDBC 实现，但是如果您喜欢实现您自己的服务，以便在持久性数据库中存储取令牌和相关的认证实例，您可以这样做。

如果您想使用这个特性，您需要为 `OAuth2RestTemplate` 提供一个特殊配置的 `AccessTokenProvider`。

```
@Bean  
{@Scope(value = "session", proxyMode = ScopedProxyMode.INTERFACES)  
public OAuth2RestOperations restTemplate() {  
    OAuth2RestTemplate template = new OAuth2RestTemplate(resource(), new DefaultOAuth2ClientContext(accessTokenRequest));  
    AccessTokenProviderChain provider = new AccessTokenProviderChain(Arrays.asList(new AuthorizationCodeAccessTokenProvider()));  
    provider.setClientTokenServices(clientTokenServices());  
    template.setAccessTokenProvider(provider);  
    return template;  
}}
```

为外部 OAuth2 提供者的客户定制。

---

一些外部的 OAuth2 提供者（例如 Facebook）并没有正确地实现该规范，或者他们只是被困在一个较旧版本的规范中，而不是 Spring Security OAuth。要在客户端应用程序中使用这些提供者，您可能需要调整客户端基础结构的各个部分。

以 Facebook 为例，在 tonr2 应用程序中有一个 Facebook 特性（您需要更改配置以添加您自己的、有效的、客户 id 和 secret——它们在 Facebook 的网站上很容易生成）。

Facebook 令牌响应也包含一个不兼容的 JSON 条目，用于令牌的失效时间（它们使用 `expires` 而不是 `expires_in`），因此，如果您想在应用程序中使用过期时间，您将不得不使用定制的 `OAuth2SerializationService` 来手动解码它。

# Pig1.0 文档

## 快速开始

# 运行环境

## 基础环境

---

JDK: 1.8+

MAVEN: 3.3.9+

MYSQL: 5.7+

Redis: 3.0+

Node: v8.9.3+

Npm: 5.5.1+

RabbitMQ: 3.7.2 (Erlang 20.1.7)

## 企业级功能环境

---

ELK: 6.1.0

CacheCloud: 1.2

Pinpoint: 1.7.1

Zookeeper: 3.3.6

FastDFS: 5.0.5

## IDE 插件

---

Lombok 插件

## 建议

---

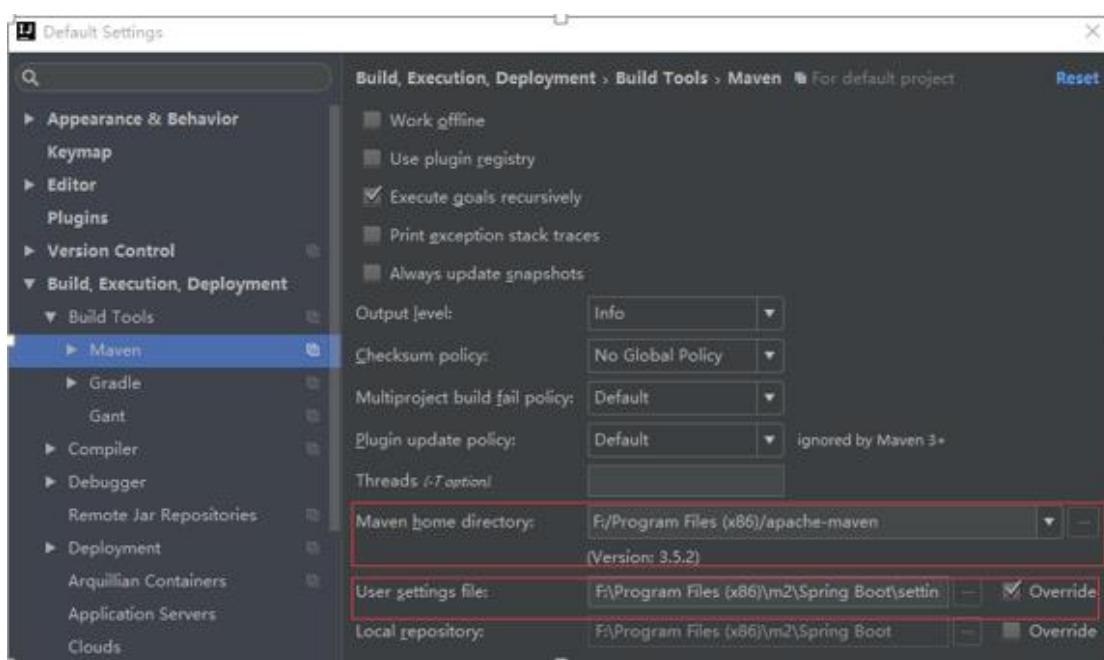
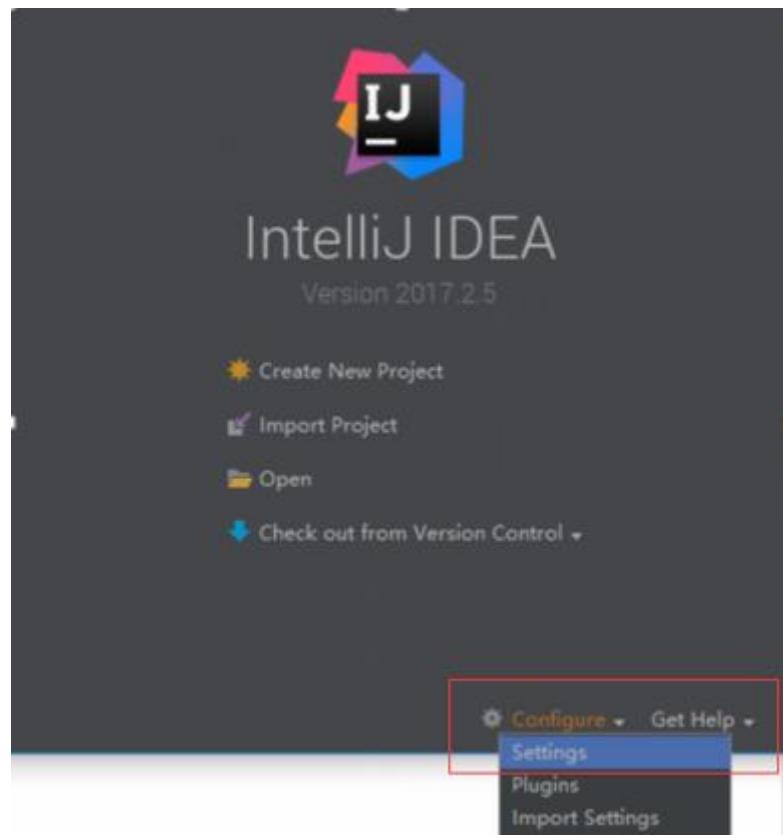
后端 IDE: IDEA 2017.3.2

前端 IDE: WebStorm 2017.3.1

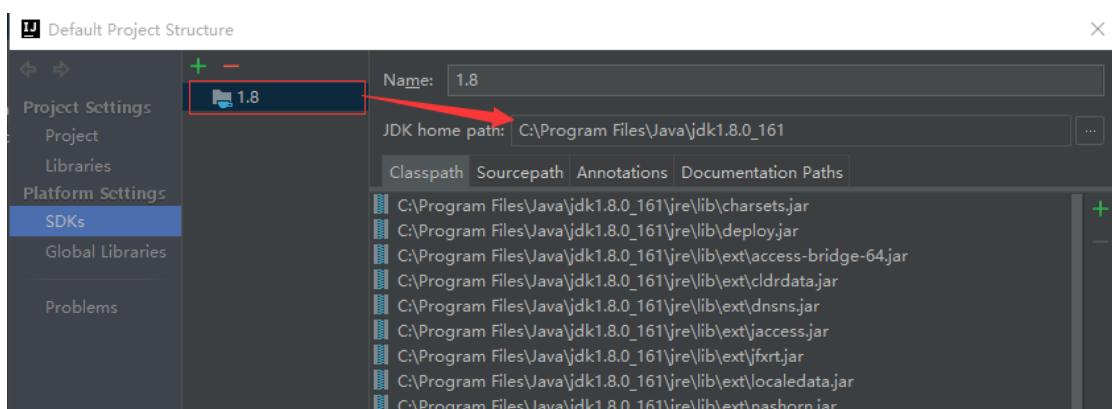
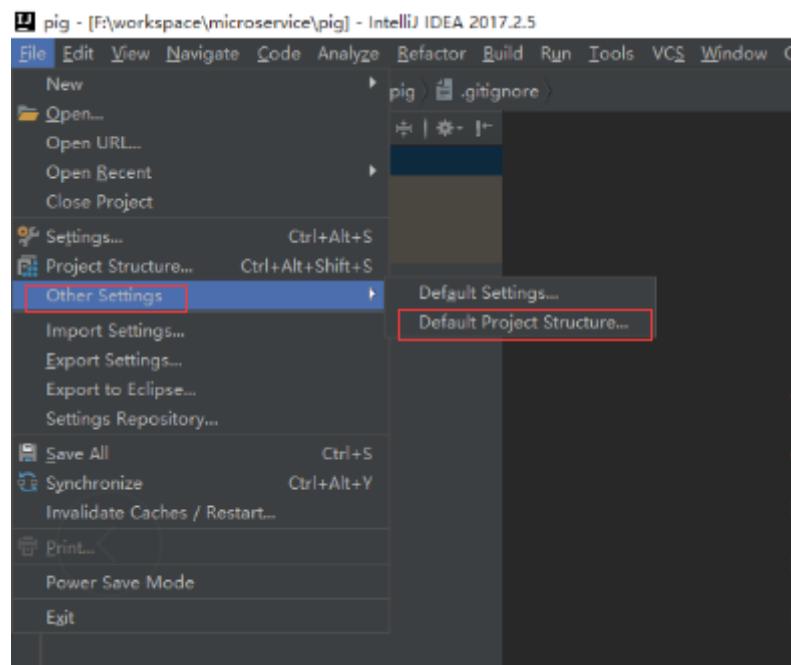
## 初始化环境

### 推荐使用 IDEA

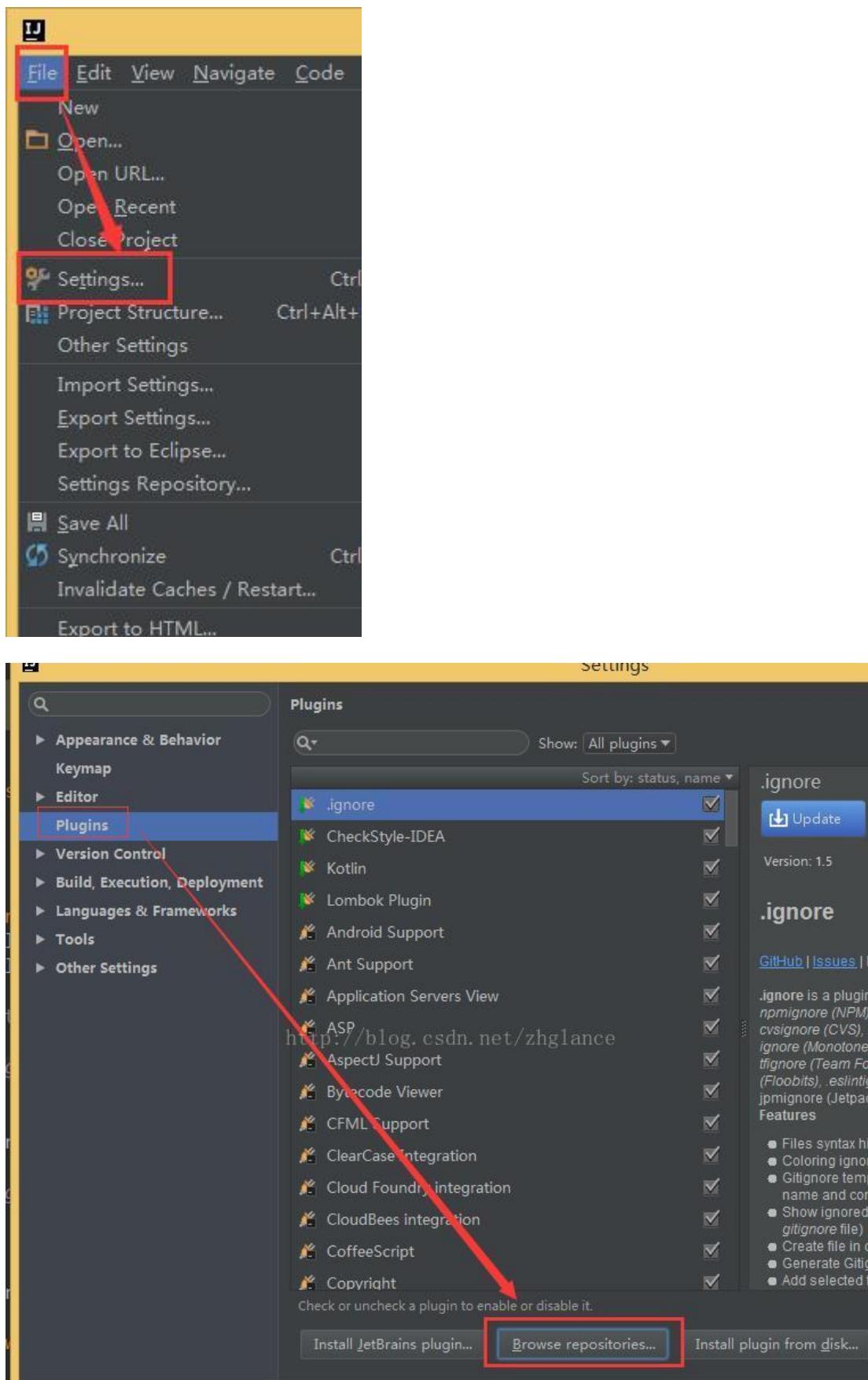
(1) 先配置 Maven 本地仓库，可以使用本地安装的 Maven，也可以使用 IDEA 自带的 Maven。然后就是通过 User setting file 中指定 XML 配置文件，可以设置本地仓库的地址。

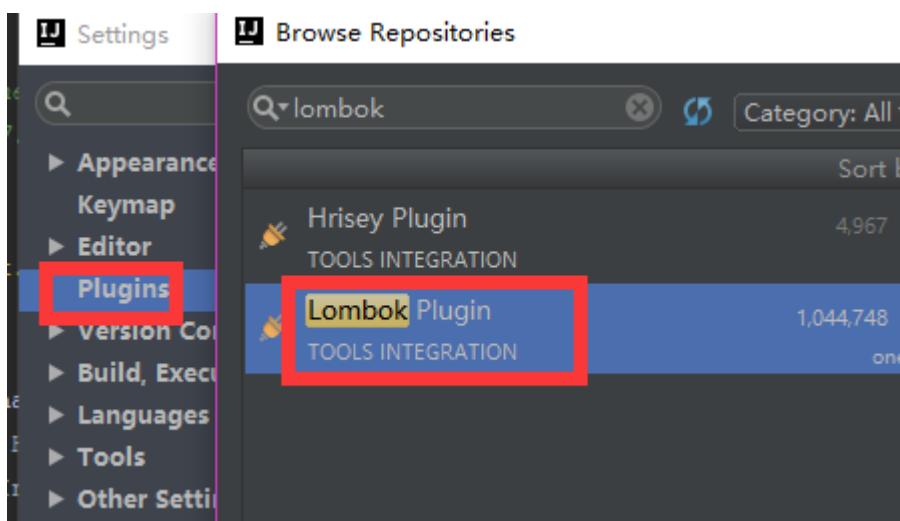
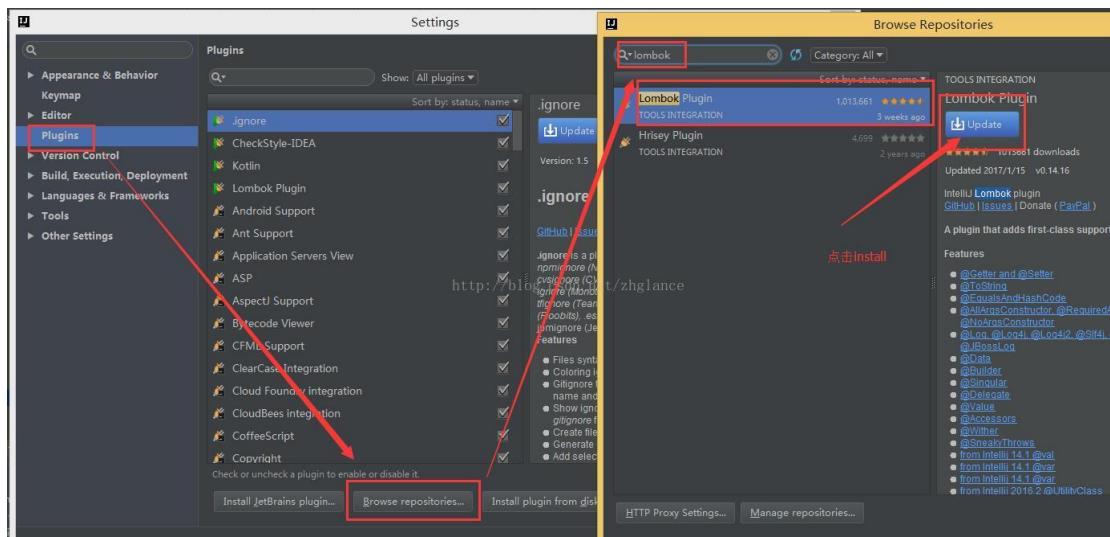


(2) IDEA 的 JDK 配置如下



### (3) IDEA 安装 lombok 插件





## Pig 项目初始化

写在最前

如果想快速部署 pig，请完全参考本篇文档，如果有个性化的修改

（例如：oauth2 配置、token 个性化需求），请参考本篇运行起来以后，自行修改。循序渐进

## 一、项目下载

码云项目地址：<https://gitee.com/log4j>

获取 pig、pig-ui 和 pig-config 项目，建议 pig、pig-ui 和 pig-config 项目在码云上

Forked 到自己的代码项目中，方便自己开发功能维护。切换到自己的代码项目，

使用 git 的 clone 到本地。

注意 pig-config 请 fork 图中，李寻欢名下的项目

冷冷 / pig

基于Spring Cloud、OAuth2.0开发基于Vue前后分离的开发平台，支持账号、短信、SSO等多种登录，提供配套视频开发教程

Java 932 ⭐ 2181 89 1199

冷冷 / pig-ui

基于Spring Cloud、OAuth2.0开发基于Vue前后分离的开发平台，支持账号、短信、SSO等多种登录，提供配套视频开发教程

JavaScript 138 ⭐ 323 89 538

李寻欢 / pig-config

这是pig的指定配置文件，请点击右上角fork到自己的库

Java 56 ⭐ 91 89 800

smallchill / SpringBlade

SpringBlade是一个基于Spring+SpringMVC+Beetl+Beetsql+Shiro的开发框架。具有权限管理，多角色，父子角色，权限...

Java 735 ⭐ 1515 89 679

冷冷 / java\_wiki

程序猿提高篇。Linux、redis、docker、maven、Git、mongodb、剑指offer（JAVA）等学习资料

Java 87 ⭐ 185 89 143

## 1. Forked 项目：

如图点击右上角的 fork 仓库，会在你的仓库生成一份镜像版本,推荐使用 fork 不用自己手动去新建仓库上传

李寻欢 / pig-config

代码 Issues 1 Pull Requests 2 附件 0 Wiki 0 统计 服务 ▾

113 次提交 3 个分支 0 个标签 0 个发行版 2 位贡献者

+ Pull Request + Issue 文件 挂件 克隆/下载

lengleng 最后提交于 6天前 更新 pig-mc-service-dev.yml

application-dev.yml lengleng 关闭ribbon懒加载 13天前

## 2. 获取 fork 到自己名下的项目仓库地址

以 pig-config 为例，下图红色部分

git项目的配置文件 -- 编辑

forked from 李寻欢 / pig-config

183 次提交 2 个分支 0 个标签 0 个发行版 3 位贡献者

dev + Pull Request + Issue 文件 挂件 克隆/下载

HTTPS SSH https://gitee.com/hsLeng/pig-config 复制

application-dev.yml  
pig-auth-dev.yml  
pig-daemon-service-dev.yml  
pig-demo-service-dev.yml  
pig-gateway-dev.yml  
pig-mc-service-dev.yml  
pig-monitor-dev.yml

冷冷 最后提交于 1 天前 更新 pig-gateway-dev.yml  
冷冷 更新 app  
冷冷 更新 pig  
冷冷 开发环境  
冷冷 demo  
冷冷 更新 pig  
冷冷 更新 pig  
冷冷 更新 pig

11 天前 9 天前 1 月前 10 天前 1 天前 8 天前 26 天前

## 3. 使用 git 工具 clone 项目:

```
lin@DESKTOP-BI2QO11 MINGW64 /f/Workspaces/pig
$ git clone https://gitee.com/log4j/pig.git
Cloning into 'pig'...
remote: Counting objects: 8144, done.
remote: Compressing objects: 100% (5138/5138), done.
remote: Total 8144 (delta 3055), reused 2893 (delta 1163)
Receiving objects: 100% (8144/8144), 5.30 MiB | 835.00 KiB/s, done.
Resolving deltas: 100% (3055/3055), done.

lin@DESKTOP-BI2QO11 MINGW64 /f/Workspaces/pig
$ git clone https://gitee.com/log4j/pig-ui.git
Cloning into 'pig-ui'...
remote: Counting objects: 1461, done.
remote: Compressing objects: 100% (1094/1094), done.
remote: Total 1461 (delta 670), reused 760 (delta 239)
Receiving objects: 100% (1461/1461), 6.42 MiB | 1.43 MiB/s, done.
Resolving deltas: 100% (670/670), done.

lin@DESKTOP-BI2QO11 MINGW64 /f/Workspaces/pig
$ git clone https://gitee.com/denglinyi/pig-config.git
Cloning into 'pig-config'...
remote: Counting objects: 374, done.
remote: Compressing objects: 100% (182/182), done.
remote: Total 374 (delta 222), reused 328 (delta 188)
Receiving objects: 100% (374/374), 40.97 KiB | 1.71 MiB/s, done.
Resolving deltas: 100% (222/222), done.
```

项目地址：

git clone <https://gitee.com/用户名/pig.git> (自己的码云项目地址)

git clone <https://gitee.com/用户名/pig-ui.git> (自己的码云项目地址)

git clone <https://gitee.com/用户名/pig-config.git> (自己的码云项目地址)

## 用户名为码云的个性化域

The screenshot shows the Gitee homepage. At the top, there's a navigation bar with links like '应用', '内容管理', '仓库', '工具', 'Readhub', '收藏', '微服务', 'Python', 'Golang', 'pay', and 'emoji'. Below the navigation is a search bar with placeholder text '搜索项目、代码片段...'. The main area features a user profile for '冷冷' (log4j) with a profile picture, follower count (253), star count (120), following count (2), and watch count (68). Below the profile, there's a brief bio: '冷冷 很懒, 懒得写', a link to 'https://my.oschina.net/glegle/blog', and a note '加入于 3年前'. To the right, there's a section titled '精选项目' (Selected Projects) listing repositories: '冷冷 / pig' (GVP), '李寻欢 / pig-config' (OVP), 'smallwei / Avue' (OVP), and '冷冷 / java\_wiki'. Each project entry includes a thumbnail, name, language, star count, fork count, and commit count.

名部分。

## 二、pig-config 修改配置

git status 查看当前分支是否在 dev 分支

The terminal window shows the following output:

```
MINGW64:/f/workspace/pig/pig-config
Lin@DESKTOP-BI2QOI1 MINGW64 /f/workspace/pig/pig-config (dev)
$ git status
On branch dev
Your branch is up-to-date with 'origin/dev'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   pig-zipkin-db-dev.yml

no changes added to commit (use "git add" and/or "git commit -a")

Lin@DESKTOP-BI2QOI1 MINGW64 /f/workspace/pig/pig-config (dev)
$ git checkout dev
```

相应文件中 MySQL、RabbitMQ、Redis 的设置如下：

MySQL: 127.0.0.1:3306/pig (自己的 MySQL 地址和端口)

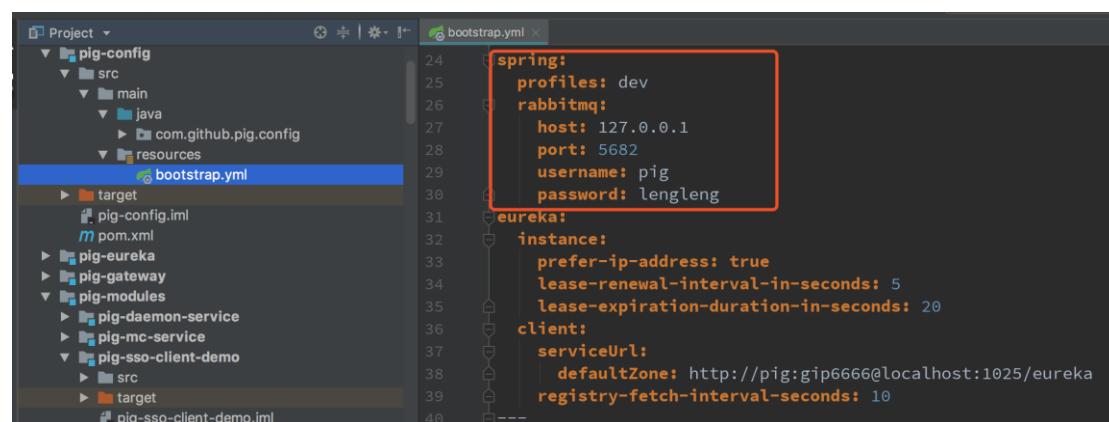
RabbitMQ: 127.0.0.1:5672 (自己的 RabbitMQ 地址和端口)

Redis: 127.0.0.1:6379 (自己的 Redis 地址和端口)

配置完成之后，使用 git 提交到自己 pig-config 项目中

### 三、 pig 修改

继续修改 pig-config bootstrap.yml 中的配置，如下



```
spring:
  profiles: dev
  rabbitmq:
    host: 127.0.0.1
    port: 5682
    username: pig
    password: lengleng
eureka:
  instance:
    prefer-ip-address: true
    lease-renewal-interval-in-seconds: 5
    lease-expiration-duration-in-seconds: 20
  client:
    serviceUrl:
      defaultZone: http://pig:gip6666@localhost:1025/eureka
      registry-fetch-interval-seconds: 10
```

特别强调 数据库 5.7+



### 四、 pig-ui 项目配置：

1、使用 npm install 安装依赖库，待下载完成之后（使用 npm 需要 Node 环境，

相关配置见 Node 配置）

2、使用 npm run dev 启动项目，当编译完成之后，项目启动

I Your application is running here: <http://localhost:8000>

此时访问 8000 页面，界面如下：



## 五、pig 项目启动顺序：

请确保启动顺序（要先启动认证中心，再启动网关）

1.eureka

2.config

3.auth

4.gateway

5.upms

## 开发教程

# jasypt 的解决方案

### 1. Maven 依赖

```
<dependency>
    <groupId>com.github.ulisesbocchio</groupId>
    <artifactId>jasypt-spring-boot-starter</artifactId>
    <version>1.16</version>
</dependency>
```

### 2. 配置

```
jasypt:
  encryptor:
    password: foo #根密码
```

### 3 调用 JAVA API 生成密文

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = PigAdminApplication.class)
public class PigAdminApplicationTest {
    @Autowired
    private StringEncryptor stringEncryptor;

    @Test
    public void testEnvironmentProperties() {
        System.out.println(stringEncryptor.encrypt("lenglen
g"));
    }
}
```

或者直接使用 JAVA 方法调用（不依赖 spring 容器）

```
/**  
 * jasypt.encryptor.password 对应 配置中心 application-dev.yml  
中的密码  
 */  
@Test  
public void testEnvironmentProperties() {  
    System.setProperty(JASYPT_ENCRYPTOR_PASSWORD, "lengleng");  
    StringEncryptor stringEncryptor = new DefaultLazyEncryptor  
(new StandardEnvironment());  
  
    //加密方法  
    System.out.println(stringEncryptor.encrypt("123456"));  
    //解密方法  
    System.out.println(stringEncryptor.decrypt("saRv7ZnXsNAfs1  
3AL90pCQ=="));  
}
```

#### 4 配置文件中使用密文

```
spring:  
  datasource:  
    password: ENC(密文)  
  
xxx: ENC(密文)
```

#### 5 其他非对称等高级配置参考

## 总结

---

1. Spring Cloud Config 提供了统一的加解密方式，方便使用，但是如果应用配置没有走配置中心，那么加解密过滤是无效的；依赖 JCE 对于低版本 spring cloud 的兼容性不好。
2. jasypt 功能更为强大，支持的加密方式更多，但是如果多个微服务，需要每个服务模块引入依赖配置，较为麻烦；但是功能强大、灵活。
3. 个人选择 jasypt

## 多维度限流

- 对请求的目标 URL 进行限流（例如：某个 URL 每分钟只允许调用多少次）
- 对客户端的访问 IP 进行限流（例如：某个 IP 每分钟只允许请求多少次）
- 对某些特定用户或者用户组进行限流（例如：非 VIP 用户限制每分钟只允许调用 100 次某个 API 等）
- 多维度混合的限流。此时，就需要实现一些限流规则的编排机制。与、或、非等关系。

### 介绍

[spring-cloud-zuul-ratelimit](#) 是和 zuul 整合提供分布式限流策略的扩展，只需在 yaml 中配置几行配置，就可使应用支持限流

```
<dependency>
    <groupId>com.marcosbarbero.cloud</groupId>
    <artifactId>spring-cloud-zuul-ratelimit</artifactId>
    <version>1.3.4.RELEASE</version>
</dependency>
```

### 支持的限流粒度

- 服务粒度（默认配置，当前服务模块的限流控制）
- 用户粒度（详细说明，见文末总结）
- ORIGIN 粒度（用户请求的 origin 作为粒度控制）
- 接口粒度（请求接口的地址作为粒度控制）
- 以上粒度自由组合，又可以支持多种情况。
- 如果还不够，自定义 RateLimitKeyGenerator 实现。

```
//默认实现
```

```
public String key(final HttpServletRequest request, final Route route, final RateLimitProperties.Policy policy) {
    final List<Type> types = policy.getType();
    final StringJoiner joiner = new StringJoiner(":");
    joiner.add(properties.getKeyPrefix());
    if (route != null) {
        joiner.add(route.getId());
    }
    if (!types.isEmpty()) {
        if (types.contains(Type.URL) && route != null) {
            joiner.add(route.getPath());
        }
        if (types.contains(Type.ORIGIN)) {
            joiner.add(getRemoteAddr(request));
        }
    }
    // 这个结合文末总结。
    if (types.contains(Type.USER)) {
        joiner.add(request.getUserPrincipal() != null ? request.getUserPrincipal().getName() : ANONYMOUS_USER);
    }
}
return joiner.toString();
}
```

## 支持的存储方式

---

- InMemoryRateLimiter - 使用 ConcurrentHashMap 作为数据存储
- ConsulRateLimiter - 使用 Consul 作为数据存储
- RedisRateLimiter - 使用 Redis 作为数据存储
- SpringDataRateLimiter - 使用 数据库 作为数据存储

## 限流配置

---

- limit 单位时间内允许访问的个数

- quota 单位时间内允许访问的总时间（统计每次请求的时间综合）
- refresh-interval 单位时间设置

```
zuul:  
  ratelimit:  
    key-prefix: your-prefix  
    enabled: true  
    repository: REDIS  
    behind-proxy: true  
    policies:  
      myServiceId:  
        limit: 10  
        quota: 20  
        refresh-interval: 30  
        type:  
          - user
```

以上配置意思是：30 秒内允许 10 个访问，并且要求总请求时间小于 20 秒

## 效果展示

---

yaml 配置：

```
zuul:  
  ratelimit:  
    key-prefix: pig-ratelimite  
    enabled: true  
    repository: REDIS  
    behind-proxy: true  
    policies:  
      pig-admin-service:  
        limit: 2  
        quota: 1  
        refresh-interval: 3
```

动态图 ↓↓↓↓

The screenshot shows a system management interface with a sidebar containing '用户管理', '菜单管理', '角色管理', '字典管理', and '日志管理'. The main area displays a table with one row of data: 序号 (1), 用户名 (admin), 密码 (123456), 角色 (admin), 创建时间 (2017-10-29 15:45), 状态 (有效), and 操作 (button). Below the table are pagination controls (共1条, 20条/页) and a search bar. At the bottom, a network traffic analysis tool is overlaid, showing a single request for 'pig-rate limite'.

## Redis 中数据结构 注意红色字体

The screenshot shows the Redis Desktop Manager interface. A key named 'pig-rate limite:...g-admin-service' is selected, showing its type as STRING and its value as '1'. The TTL is listed as 2. A red box highlights the value '1', which is annotated with the text '单位时间内已通过次数' (Number of times passed through within the unit time).

## 代码生成使用

结合 mybatis-plus 进行增强，可以生成 Vue 前端代码。

## 生成代码包含

---

```
↳ controller.java.vm  
↳ listvue.vue.vm  
↳ menu.sql.vm
```

Controller、service、mapper、entity 代码

vue 前端代码

菜单配置 SQL

## 使用方法

---

**PigResourcesGenerator.java** main 执行

- 代码生成是在 mybatis-plus 代码生成的扩展，生成 VUE 页面，直接扔到 pig-ui 的模块下，注意检查修改。

## 暴露 API 网关

很多情况，很多服务接口不需要权限判断，或者用户信息，直接提供第三方数据接口，针对这种业务场景，pig 对其进行了支持。

## 修改 pig-gateway 文件

application-dev.yml	lengleng	关闭ribbon懒加载	2天前
pig-auth-dev.yml	lengleng	fixed pig-auth 监控状态不准确问题。监控端点被拦截，如需其他监控...	6天前
pig-daemon-service-dev.yml	冷冷	更新 pig-daemon-service-dev.yml	1月前
pig-gateway-dev.yml	lengleng	更新 pig-gateway-dev.yml	2天前
pig-mc-service-dev.yml	lengleng	更新 pig-mc-service-dev.yml	14天前
pig-monitor-dev.yml	冷冷	更新 pig-monitor-dev.yml	2月前
pig-upms-service-dev.yml	lengleng	fixed 遗漏端口	10天前
pig-zipkin-db-dev.yml	冷冷	更新 pig-zipkin-db-dev.yml	2月前
pig-zipkin-elk-dev.yml	冷冷	更新 pig-zipkin-elk-dev.yml	2月前

## 配置 urls:anon

```
63  urls:
64    anon:
65      - /mobile/**
66      - /auth/**
67      - /admin/code/*
68      - /admin/smsCode/*
69      - /admin/user/info
70      - /admin/menu/userTree
71      - /swagger-resources/**
72      - /swagger-ui.html
73      - /*/v2/api-docs
74      - /swagger/api-docs
75      - /webjars/**
76
```

在最后追加暴露的接口地址。

## 验证码开关

对于开发过程中，验证码（含有短信验证码）可以设置关闭

## pig-gateway 配置

操作	提交者	描述	时间
lengleng 最后提交于 2天前 更新 pig-gateway-dev.yml	lengleng		2天前
application-dev.yml	lengleng	关闭ribbon懒加载	2天前
pig-auth-dev.yml	lengleng	fixed pig-auth 监控状态不准确问题。监控端点被拦截，如需其他监控...	6天前
pig-daemon-service-dev.yml	冷冷	更新 pig-daemon-service-dev.yml	1月前
pig-gateway-dev.yml	lengleng	更新 pig-gateway-dev.yml	2天前
pig-mc-service-dev.yml	lengleng	更新 pig-mc-service-dev.yml	14天前
pig-monitor-dev.yml	冷冷	更新 pig-monitor-dev.yml	2月前
pig-upms-service-dev.yml	lengleng	fixed 遗漏端口	10天前
pig-zipkin-db-dev.yml	冷冷	更新 pig-zipkin-db-dev.yml	2月前
pig-zipkin-elk-dev.yml	冷冷	更新 pig-zipkin-elk-dev.yml	2月前

## security.validate.code

```
48 security:
49   validate:
50     code: false
51   sessions: stateless
52   oauth2:
53     client:
54       client-id: pig
55       client-secret: pig
56     resource:
57       loadBalanced: true
58       token-info-uri: ${security.auth.server}/oauth/check-token
59       service-id: pig-gateway
60       jwt:
61         key-uri: ${security.auth.server}/oauth/token_key #解析jwt令牌所需要密钥的地址
```

false 关闭验证码校验

特殊终端不校验

比如说 app，那么配置忽略的 clientId 就可以

```
ignore:
  clients:
    - app
```

## 校验逻辑参考

```
1 package com.github.pig.gateway.component.filter;
2
3 import ...
4
5 /**
6  * @author lengleng
7  * @date 2017-12-18
8  * 验证码校验, true开启, false关闭校验
9  * 更细化可以 clientId 进行区分
10 */
11 @Component("validateCodeFilter")
12 public class ValidateCodeFilter extends OncePerRequestFilter {
```

## 终端接口调用

通过网关访问 auth-server 获取 access-token

网关校验验证码关闭:

```
  security:  
    validate:  
      code: false
```

注意

**password** 需要加密。, 可以通过前端登录, **network** 获取一个请求的 password, 或者使用 **AES** 加密一个 原文。

```
// cGlnOnBpZw== 是 Base64(clientId:secret) 默认在 auth 模块的配置里面 pig:pig
curl -H "Authorization:Basic cGlnOnBpZw==" -d "grant_type=password&scope=server&username=admin&password=XXX" http://localhost:9999/auth/oauth/token
```

```
RpIjoizWmZmJhMjYtMGJkZS00YjY2LTThhZTQtZGRmYTNiMzkxZGM5IiwiY2xpZW50X2lkIjoicGlnIiwic2NvcGUiOlsic2VydmVyIl19.ZoSU_4NhdoInV6ZsNaSXITC_pewUDiaqZPLoESu9f9s", "expires_in": 3600, "scope": "server"}
```

通过网关访问 auth-server 刷新 token

```
curl -H "Authorization:Basic cGlnOnBpZw==" -d 'grant_type=refresh_token&refresh_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MDk1NzA0NjMsInVzZXJfbmFtZSI6ImFkbWluIiwiYXV0aG9yaXRpZXMiOlsiYWRtaW4iXSwianRpIjoizWmZmJhMjYtMGJkZS00YjY2LTThhZTQtZGRmYTNiMzkxZGM5IiwiY2xpZW50X2lkIjoicGlnIiwic2NvcGUiOlsic2VydmVyIl19.ZoSU_4NhdoInV6ZsNaSXITC_pewUDiaqZPLoESu9f9s' http://localhost:9999/auth/oauth/token
```

通过 access-token 访问受保护的资源

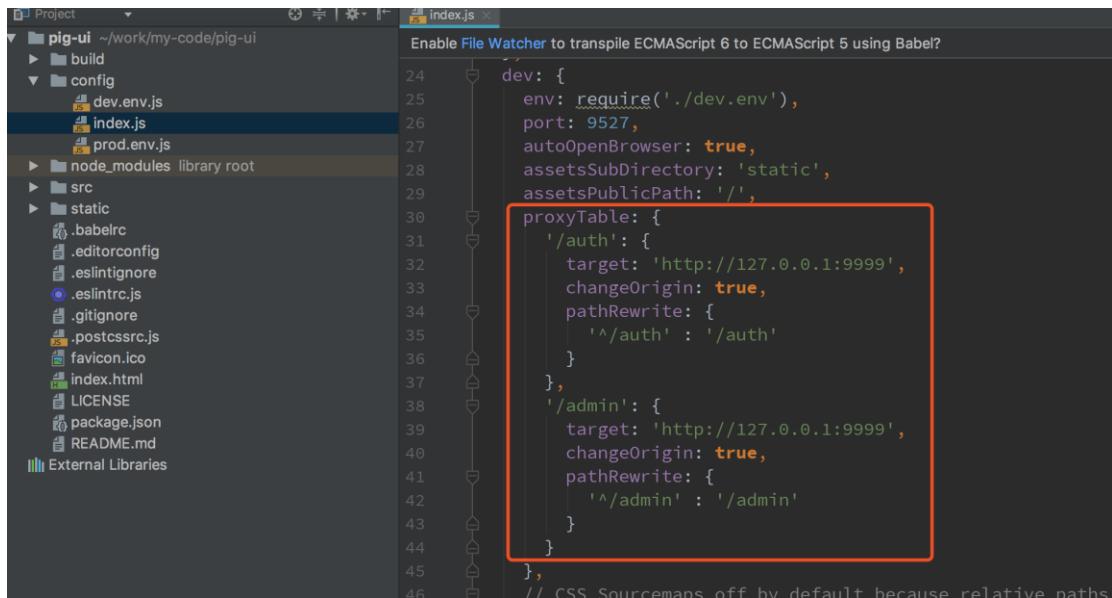
```
curl -H "Authorization:Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MDk1NzA0NjMsInVzZXJfbmFtZSI6ImFkbWluIiwiYXV0aG9yaXRpZXMiOlsiYWRtaW4iXSwianRpIjoizWmZmJhMjYtMGJkZS00YjY2LTThhZTQtZGRmYTNiMzkxZGM5IiwiY2xpZW50X2lkIjoicGlnIiwic2NvcGUiOlsic2VydmVyIl19.ZoSU_4NhdoInV6ZsNaSXITC_pewUDiaqZPLoESu9f9s" http://localhost:9999/admin/user/info
```

跨域处理

跨域实现可以通过两种方式实现，二者取其一

## 前端跨域

vue-cli 支持代理工具已经能满足开发需求，类似于 nginx 的代理配置规则，只需修改成对应的 pig-gateway 的地址即可



```
24 dev: {
25   env: require('./dev.env'),
26   port: 9527,
27   autoOpenBrowser: true,
28   assetsSubDirectory: 'static',
29   assetsPublicPath: '/',
30   proxyTable: {
31     '/auth': {
32       target: 'http://127.0.0.1:9999',
33       changeOrigin: true,
34       pathRewrite: {
35         '^/auth' : '/auth'
36       }
37     },
38     '/admin': {
39       target: 'http://127.0.0.1:9999',
40       changeOrigin: true,
41       pathRewrite: {
42         '/admin' : '/admin'
43       }
44     }
45   },
46   // CSS Sourcemaps off by default because relative paths
47 }
```

## 服务端跨域

如果使用服务端跨域打开注释部分代码即可，生产配置建议使用 nginx 代理，详见生产部署。

```

    package com.github.pig.gateway;
    ...
    @ComponentScan(basePackages = {"com.github.pig.gateway", "com.github.pig.common.bean"})
    public class PigGatewayApplication {
        ...
        public static void main(String[] args) { SpringApplication.run(PigGatewayApplication.class, args); }
        ...
    }

```

```

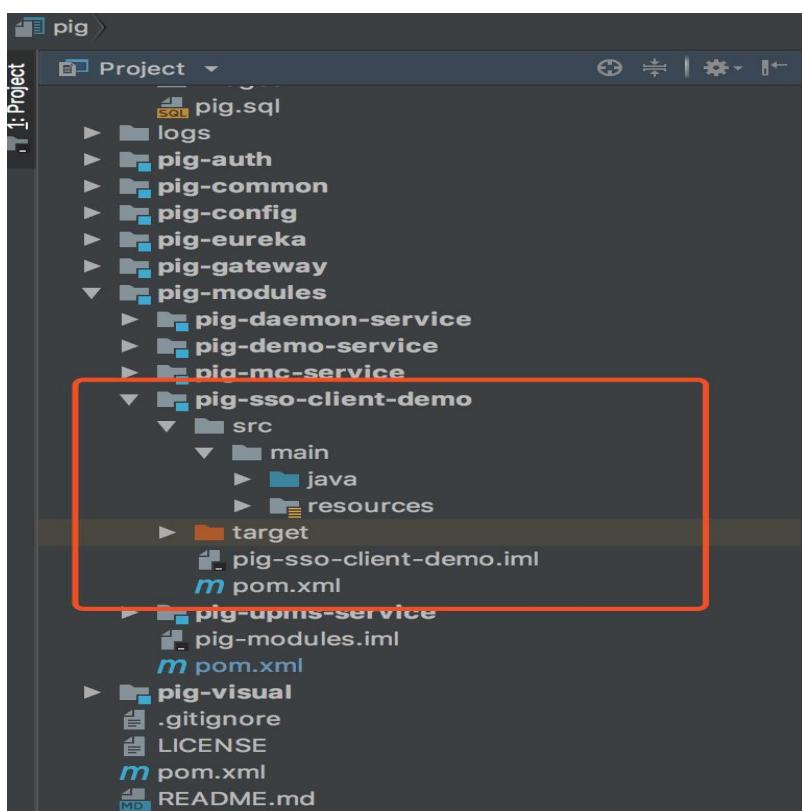
    @Override
    public boolean hasPermission(HttpServletRequest request, Authentication authentication) {
        //ele-admin options 跨域配置，现在处理是通过前端配置代理，不使用这种方式，存在风险
        if (HttpMethod.OPTIONS.name().equalsIgnoreCase(request.getMethod())) {
            return true;
        }
        ...
        Object principal = authentication.getPrincipal();
        List<SimpleGrantedAuthority> grantedAuthorityList = (List<SimpleGrantedAuthority>) authentication.getAuthorities();
        boolean hasPermission = false;
        ...
    }

```

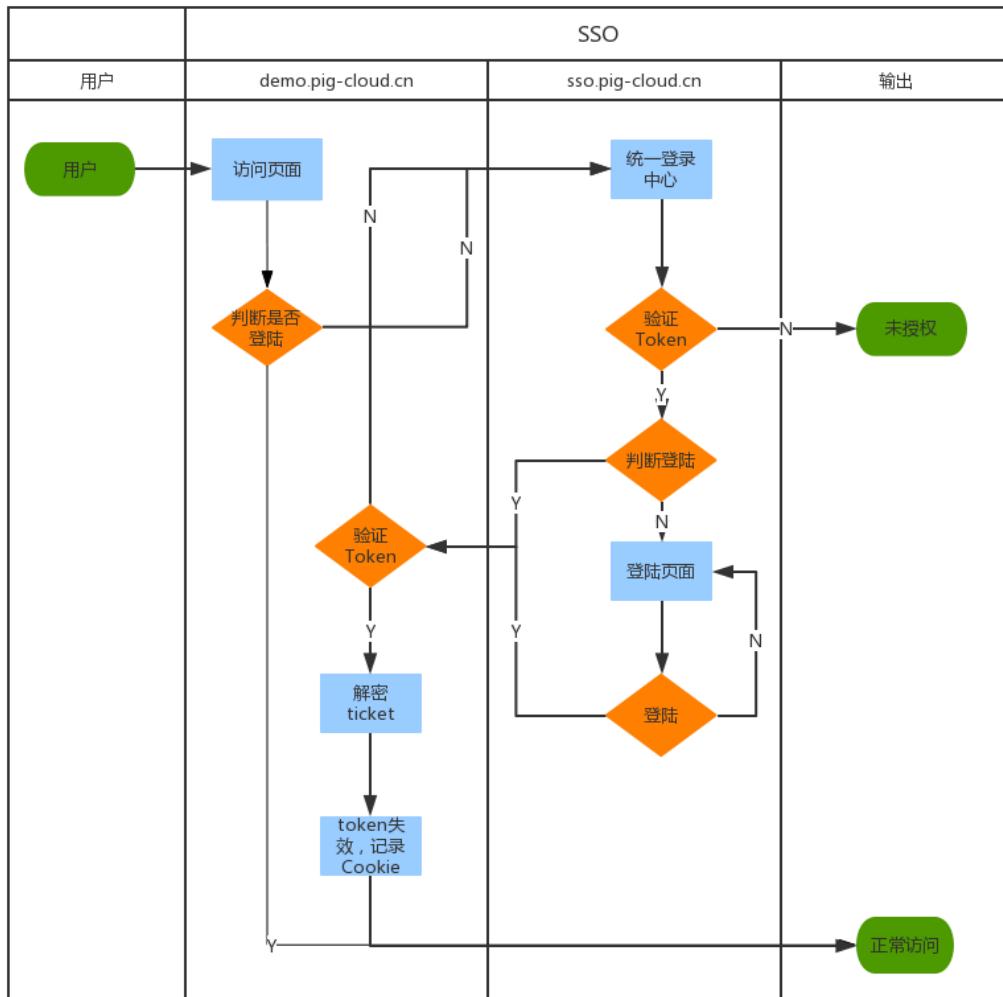
## SSO 单点登录实现

尝试使用

pig 提供了一个 SSO 的客户端, **pig-sso-demo**



1. 访问单点登录: <http://localhost:4040/sso1/>
2. 跳转至统一认证界面（3000 统一认证）
3. 重定向回 <http://localhost:4040/sso1/>（4040 携带用户信息）



## 单点登录概念

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。登录逻辑如上图

# 基于 Spring 全家桶的实现

---

技术选型:

Spring Boot

Spring Cloud

Spring Security oAuth2

客户端:

maven 依赖

```
<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>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
```

```
</dependency>
```

## EnableOAuth2Sso 注解

入口类配置@@EnableOAuth2Sso

```
@SpringBootApplication

public class PigSsoClientDemoApplication {

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

## 配置文件

```
security:

oauth2:

    client:

        client-id: pig

        client-secret: pig

        user-authorization-uri: http://localhost:3000/oauth/authorize

        access-token-uri: http://localhost:3000/oauth/token

        scope: serve

    resource:
```

```
jwt:  
  
  key-uri: http://localhost:3000/oauth/token\_key  
  
sessions: neve
```

## SSO 认证服务器

---

### 认证服务器配置

```
@Configuration  
  
@Order(Integer.MIN\_VALUE)  
  
@EnableAuthorizationServer  
  
public class PigAuthorizationConfig extends AuthorizationServerConfigurerAdapter {  
  
  
  
  @Override  
  
    public void configure(ClientDetailsServiceConfigurer clients)  
throws Exception {  
  
    clients.inMemory()  
  
      .withClient(authServerConfig.getClientId())  
  
      .secret(authServerConfig.getClientSecret())  
  
      .authorizedGrantTypes(SecurityConstants.REFRESH\_TOKEN,  
SecurityConstants.PASSWORD,SecurityConstants.AUTHORIZATION\_CODE)  
  
      .scopes(authServerConfig.getScope());  
  
  }  
  
}
```

```
    @Override

        public void configure(AuthorizationServerEndpointsConfigurer endpoints) {

            endpoints

                .tokenStore(new RedisTokenStore(redisConnectionFactory))

                .accessTokenConverter(jwtAccessTokenConverter())

                .authenticationManager(authenticationManager)

                .exceptionTranslator(pigWebResponseExceptionTranslator)

                .reuseRefreshTokens(false)

                .userDetailsService(userDetailsService);

        }

    }

    @Override

        public void configure(AuthorizationServerSecurityConfigurer security) throws Exception {

            security

                .allowFormAuthenticationForClients()

                .tokenKeyAccess("isAuthenticated()")

                .checkTokenAccess("permitAll()");

        }

    }

    @Bean
```

```
public PasswordEncoder passwordEncoder() {  
    return new BCryptPasswordEncoder();  
}  
  
@Bean  
public JwtAccessTokenConverter jwtAccessTokenConverter() {  
    JwtAccessTokenConverter jwtAccessTokenConverter = new JwtAccessTokenConverter();  
  
    jwtAccessTokenConverter.setSigningKey(CommonConstant.SIGN_KEY);  
  
    return jwtAccessTokenConverter;  
}  
  
}
```

配置完成体验

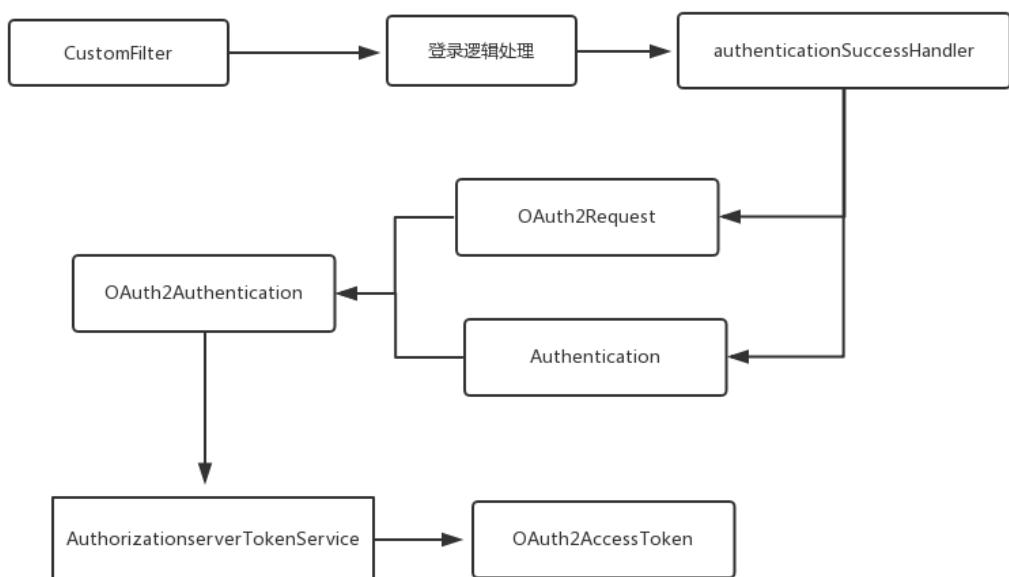
---

1. 访问 SSO 客户端的 index.html
2. 重定向到 SSO 服务端的 Basic 认证
3. 输入账号密码又重定向到原请求的 客户端 index 资源

手机号登录实现

spring security oauth2 登录过程详解

---



## 定义手机号登录令牌

```

/**
 * @author lengleng
 * @date 2018/1/9
 * 手机号登录令牌
 */
public class MobileAuthenticationToken extends AbstractAuthenticationToken {

    private static final long serialVersionUID = SpringSecurityCoreVersion.SERIAL_VERSION_UID;

    private final Object principal;

    public MobileAuthenticationToken(String mobile) {
        super(null);
        this.principal = mobile;
        setAuthenticated(false);
    }

    public MobileAuthenticationToken(Object principal,
                                    Collection<? extends GrantedAuthority> authorities) {
  
```

```

        super(authorities);
        this.principal = principal;
        super.setAuthenticated(true);
    }

    public Object getPrincipal() {
        return this.principal;
    }

    @Override
    public Object getCredentials() {
        return null;
    }

    public void setAuthenticated(boolean isAuthenticated) throws I
llegalArgumentException {
        if (isAuthenticated) {
            throw new IllegalArgumentException(
                "Cannot set this token to trusted - use construc
tor which takes a GrantedAuthority list instead");
        }

        super.setAuthenticated(false);
    }

    @Override
    public void eraseCredentials() {
        super.eraseCredentials();
    }
}

```

## 手机号登录校验逻辑

```

/**
 * @author lengleng
 * @date 2018/1/9
 * 手机号登录校验逻辑
 */
public class MobileAuthenticationProvider implements Authentificatio
nProvider {
    private UserService userService;

    @Override

```

```

    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        MobileAuthenticationToken mobileAuthenticationToken = (MobileAuthenticationToken) authentication;
        UserVo userVo = userService.findUserByMobile((String) mobileAuthenticationToken.getPrincipal());

        UserDetailsImpl userDetails = buildUserDeatils(userVo);
        if (userDetails == null) {
            throw new InternalAuthenticationServiceException("手机号不存在：" + mobileAuthenticationToken.getPrincipal());
        }

        MobileAuthenticationToken authenticationToken = new MobileAuthenticationToken(userDetails, userDetails.getAuthorities());
        authenticationToken.setDetails(mobileAuthenticationToken.getDetails());
        return authenticationToken;
    }

    private UserDetailsImpl buildUserDeatils(UserVo userVo) {
        return new UserDetailsImpl(userVo);
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return MobileAuthenticationToken.class.isAssignableFrom(authentication);
    }

    public UserService getUserService() {
        return userService;
    }

    public void setUserService(UserService userService) {
        this.userService = userService;
    }
}

```

## 登录过程 filter 处理

```

/**
 * @author lengleng
 * @date 2018/1/9

```

```
* 手机号登录验证 filter
*/
public class MobileAuthenticationFilter extends AbstractAuthentica
tionProcessingFilter {
    public static final String SPRING_SECURITY_FORM_MOBILE_KEY = "
mobile";

    private String mobileParameter = SPRING_SECURITY_FORM_MOBILE_K
EY;
    private boolean postOnly = true;

    public MobileAuthenticationFilter() {
        super(new AntPathRequestMatcher(SecurityConstants.MOBILE_T
OKEN_URL, "POST"));
    }

    public Authentication attemptAuthentication(HttpServletRequest re
quest,
                                                HttpServletResponse re
sponse) throws AuthenticationException {
        if (postOnly && !request.getMethod().equals(HttpMethod.POS
T.name())) {
            throw new AuthenticationServiceException(
                "Authentication method not supported: " + reque
st.getMethod());
        }

        String mobile = obtainMobile(request);

        if (mobile == null) {
            mobile = "";
        }

        mobile = mobile.trim();

        MobileAuthenticationToken mobileAuthenticationToken = new
MobileAuthenticationToken(mobile);

        setDetails(request, mobileAuthenticationToken);

        return this.getAuthenticationManager().authenticate(mobile
AuthenticationToken);
    }
}
```

```

protected String obtainMobile(HttpServletRequest request) {
    return request.getParameter(mobileParameter);
}

protected void setDetails(HttpServletRequest request,
                         MobileAuthenticationToken authRequest)
{
    authRequest.setDetails(authenticationDetailsSource.buildDe
tails(request));
}

public void setPostOnly(boolean postOnly) {
    this.postOnly = postOnly;
}

public String getMobileParameter() {
    return mobileParameter;
}

public void setMobileParameter(String mobileParameter) {
    this.mobileParameter = mobileParameter;
}

public boolean isPostOnly() {
    return postOnly;
}
}

```

## 生产 token 位置

---

```

/**
 * @author lengleng
 * @date 2018/1/8
 * 手机号登录成功，返回 oauth token
 */
@Component
public class MobileLoginSuccessHandler implements org.springframework
security.web.authentication.AuthenticationSuccessHandler {
    private Logger logger = LoggerFactory.getLogger(getClass());
    @Autowired
    private ObjectMapper objectMapper;
    @Autowired
    private ClientDetailsService clientDetailsService;
    @Autowired

```

```
    private AuthorizationServerTokenServices authorizationServerTo
kenServices;

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) {
        String header = request.getHeader("Authorization");

        if (header == null || !header.startsWith("Basic ")) {
            throw new UnapprovedClientAuthenticationException("请求头中 client 信息为空");
        }

        try {
            String[] tokens = extractAndDecodeHeader(header);
            assert tokens.length == 2;
            String clientId = tokens[0];
            String clientSecret = tokens[1];

            JSONObject params = new JSONObject();
            params.put("clientId", clientId);
            params.put("clientSecret", clientSecret);
            params.put("authentication", authentication);

            ClientDetails clientDetails = clientDetailsService.load
ClientByClientId(clientId);
            TokenRequest tokenRequest = new TokenRequest(MapUtil.ne
wHashMap(), clientId, clientDetails.getScope(), "mobile");
            OAuth2Request oAuth2Request = tokenRequest.createOAuth2
Request(clientDetails);

            OAuth2Authentication oAuth2Authentication = new OAuth2A
uthentication(oAuth2Request, authentication);
            OAuth2AccessToken oAuth2AccessToken = authorizationServ
erTokenServices.createAccessToken(oAuth2Authentication);
            logger.info("获取 token 成功: {}", oAuth2AccessToken.get
Value());

            response.setCharacterEncoding(CommonConstant.UTF8);
            response.setContentType(CommonConstant.CONTENT_TYPE);
            PrintWriter printWriter = response.getWriter();
            printWriter.append(objectMapper.writeValueAsString(oAu
th2AccessToken));
        } catch (IOException e) {
```

```

        throw new BadCredentialsException(
            "Failed to decode basic authentication token");
    }

}

/**
 * Decodes the header into a username and password.
 *
 * @throws BadCredentialsException if the Basic header is not
 * present or is not valid
 *                               Base64
 */
private String[] extractAndDecodeHeader(String header)
    throws IOException {

    byte[] base64Token = header.substring(6).getBytes("UTF-8");
    byte[] decoded;
    try {
        decoded = Base64.decode(base64Token);
    } catch (IllegalArgumentException e) {
        throw new BadCredentialsException(
            "Failed to decode basic authentication token");
    }

    String token = new String(decoded, CommonConstant.UTF8);

    int delim = token.indexOf ":";

    if (delim == -1) {
        throw new BadCredentialsException("Invalid basic authentication token");
    }
    return new String[]{token.substring(0, delim), token.substring(delim + 1)};
}

```

配置以上自定义

---

```

/**
 * @author lengleng
 * @date 2018/1/9
 * 手机号登录配置入口

```

```

*/
@Component
public class MobileSecurityConfigurer extends SecurityConfigurerAdapter<DefaultSecurityFilterChain, HttpSecurity> {
    @Autowired
    private MobileLoginSuccessHandler mobileLoginSuccessHandler;
    @Autowired
    private UserService userService;

    @Override
    public void configure(HttpSecurity http) throws Exception {
        MobileAuthenticationFilter mobileAuthenticationFilter = new MobileAuthenticationFilter();
        mobileAuthenticationFilter.setAuthenticationManager(http.getSharedObject(AuthenticationManager.class));
        mobileAuthenticationFilter.setAuthenticationSuccessHandler(mobileLoginSuccessHandler);

        MobileAuthenticationProvider mobileAuthenticationProvider
= new MobileAuthenticationProvider();
        mobileAuthenticationProvider.setUserService(userService);
        http.authenticationProvider(mobileAuthenticationProvider)
            .addFilterAfter(mobileAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);
    }
}

```

在 spring security 配置 上边定一个的那个聚合配置

```

/**
 * @author lengleng
 * @date 2018年01月09日14:01:25
 * 认证服务器开放接口配置
 */
@Configuration
@EnableResourceServer
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {
    @Autowired
    private FilterUrlsPropertiesConifg filterUrlsPropertiesConifg;
    @Autowired
    private MobileSecurityConfigurer mobileSecurityConfigurer;

    @Override

```

```
public void configure(HttpSecurity http) throws Exception {
    registry
        .antMatchers("/mobile/token").permissionAll()
        .anyRequest().authenticated()
        .and()
        .csrf().disable();
    http.apply(mobileSecurityConfigurer);
}
```

## 使用

```
curl -H "Authorization:Basic cGlnOnBpZw==" -d "grant_type=mobile&scope=server&mobile=17034642119&code=" http://localhost:9999/auth/mobile/token
```

## 源码

1. 整个逻辑是参考 spring security 自身的 usernamepassword 登录模式实现，可以参考其源码。
2. 验证码的发放、校验逻辑比较简单，方法后通过全局 filter 判断请求中 code 是否和 手机号匹配集合，重点逻辑是令牌的参数

## 获取当前用户

在 **pig** 的微服务模块中获取到用户信息

### 第一种方法 controller 注入一个 UserVo 对象

如下代码是 UserController 获取用户信息的方法

```
@GetMapping("/info")
public R<UserInfo> user(UserVO userVo) {
    UserInfo userInfo = userService.findUserInfo(userVo);
    return new R<>(userInfo);
}
```

实现原理请参考，借助 springmvc 的 web 增强实现

```
public class TokenArgumentResolver implements HandlerMethodArgumentResolver {

    /**
     * 1. 入参筛选
     *
     * @param methodParameter 参数集合
     * @return 格式化后的参数
     */
    @Override
    public boolean supportsParameter(MethodParameter methodParameter) {
        return methodParameter.getParameterType().equals(UserVO.class);
    }

    /**
     * @param methodParameter      入参集合
     * @param modelAndViewContainer model 和 view
     * @param nativeWebRequest      web 相关
     * @param webDataBinderFactory 入参解析
     * @return 包装对象
     * @throws Exception exception
     */
    @Override
    public Object resolveArgument(MethodParameter methodParameter,
                                  ModelAndViewContainer modelAndViewContainer,
                                  NativeWebRequest nativeWebRequest,
                                  WebDataBinderFactory webDataBinderFactory) {
        HttpServletRequest request = nativeWebRequest.getNativeRequest(HttpServletRequest.class);
        String username = request.getHeader(SecurityConstants.USER_HEADER);
        String roles = request.getHeader(SecurityConstants.ROLE_HEADER);
        if (StringUtil.isBlank(username) || StringUtil.isBlank(roles)) {
            log.warn("resolveArgument error username or role is empty");
            return null;
        } else {

```

```
        log.info("resolveArgument username:{} roles:{}", username, roles);
    }
    UserVO userVO = new UserVO();
    userVO.setUsername(username);
    List<SysRole> sysRoleList = new ArrayList<>();
    Arrays.stream(roles.split(",")).forEach(role -> {
        SysRole sysRole = new SysRole();
        sysRole.setRoleName(role);
        sysRoleList.add(sysRole);
    });
    userVO.setRoleList(sysRoleList);
    return userVO;
}

}
```

## 第二种方法 使用 UserUtils

采用 TTL , 保存上下文对象

```
UserUtils.getUser();  
复制
```

这种要求你的接口返回值 是 R<>,Page<> 才能被 AOP 切到

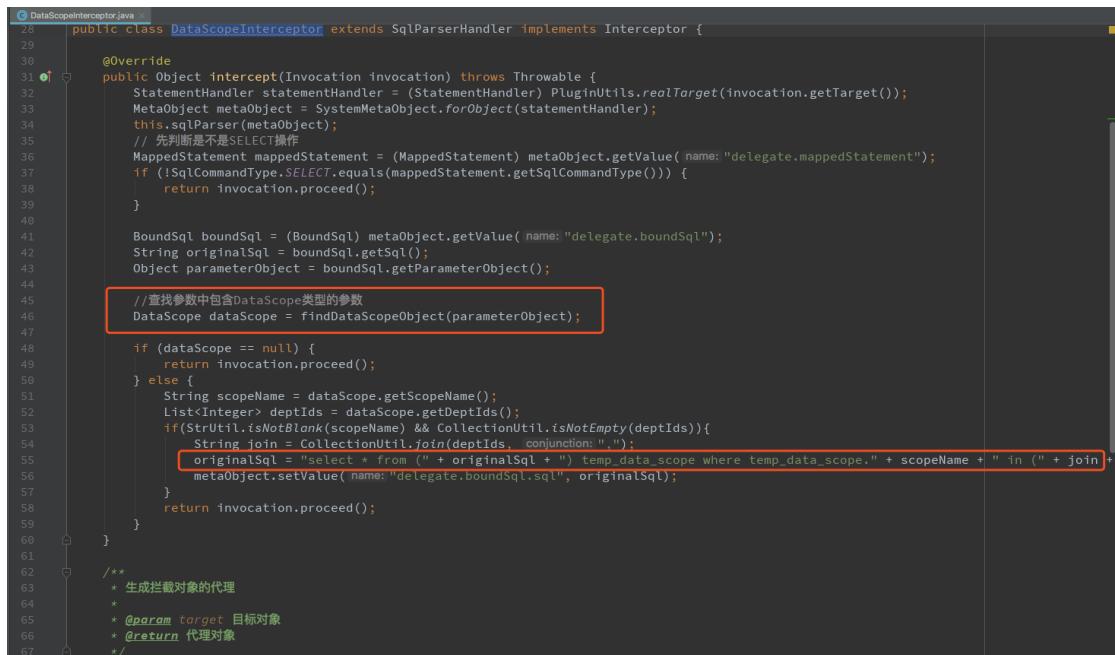
具体参考： ControllerAop.java

## 数据权限

---

所谓数据权限，就是根据部分的上下级层级，来确定数据的展示范围。

pig 中基于 **Mybatis** 拦截器实现。



```
28  public class DataScopeInterceptor extends SqlParserHandler implements Interceptor {  
29  
30  
31     @Override  
32     public Object intercept(Invocation invocation) throws Throwable {  
33         StatementHandler statementHandler = (StatementHandler) PluginUtils.realTarget(invocation.getTarget());  
34         MetaObject metaObject = SystemMetaObject.forObject(statementHandler);  
35         this.sqlParser(metaObject);  
36         // 先判断是不是SELECT操作  
37         MappedStatement mappedStatement = (MappedStatement) metaObject.getValue(name: "delegate.mappedStatement");  
38         if (!SqlCommandType.SELECT.equals(mappedStatement.getSqlCommandType())) {  
39             return invocation.proceed();  
40         }  
41         BoundSql boundSql = (BoundSql) metaObject.getValue(name: "delegate.boundSql");  
42         String originalSql = boundSql.getSql();  
43         Object parameterObject = boundSql.getParameterObject();  
44         //查找参数中包含DataScope类型的参数  
45         DataScope dataScope = findDataScopeObject(parameterObject);  
46         if (dataScope == null) {  
47             return invocation.proceed();  
48         } else {  
49             String scopeName = dataScope.getScopeName();  
50             List<Integer> deptIds = dataScope.getDeptIds();  
51             if (StrUtil.isNotBlank(scopeName) && CollectionUtil.isNotEmpty(deptIds)){  
52                 String join = CollectionUtil.join(deptIds, conjunction:",");  
53                 originalSql = "select * from (" + originalSql + ") temp_data_scope where temp_data_scope." + scopeName + " in (" + join +  
54                         metaObject.setValue(name: "delegate.boundSql.sql", originalSql);  
55             }  
56             return invocation.proceed();  
57         }  
58     }  
59     /**  
60      * 生成拦截对象的代理  
61      *  
62      * @param target 目标对象  
63      * @return 代理对象  
64      */  
65 }  
66  
67
```

## 参数配置

```
@Override  
public Page selectWithRolePage(Query query) {  
    DataScope dataScope = new DataScope();  
    dataScope.setScopeName("deptId");  
    dataScope.setIsOnly(true);  
    dataScope.setDeptIds(getChildDepts());  
    dataScope.putAll(query.getCondition());  
    query.setRecords(sysUserMapper.selectUserVoPageDataScope(query, dataScope));  
    return query;  
}
```

可以在 **SysUserServiceImpl** 实现。

## 详解参数

```
@Data  
public class DataScope extends HashMap {  
    /**  
     * 限制范围的字段名称  
     */  
    private String scopeName = "dept_id";  
  
    /**  
     * 具体的数据范围  
     */
```

```
 */
private List<Integer> deptIds;

/**
 * 是否只查询本部门
 */
private Boolean isOnly = false;
}
```

## 功能扩展

### Redis 集群

application.yml 配置集群模式

```
spring:
  redis:
    cluster:
      nodes:
        - 139.224.200.249:7000
        - 139.224.200.249:7001
        - 139.224.200.249:7002
        - 139.224.200.249:7003
        - 139.224.200.249:7004
        - 139.224.200.249:7005
```

Auth 模块修改原有的 RedisTokenStore

为什么要修改？因为 Spring Security OAuth 自带的 `redistokenstore` 含有大量的管道操作，在 `cluster` 模式不支持会报错

`Pipeline is currently not supported for JedisClusterConnection.`

所以我们要自定义 `redistokenstore` 替代

```
public TokenStore redisTokenStore() {
  PigRedisTokenStore tokenStore = new PigRedisTokenStore();
  tokenStore.setRedisTemplate(redisTemplate);
  return tokenStore;
}
```

## PigRedisTokenStore

```
public class PigRedisTokenStore implements TokenStore {

    private static final String ACCESS = "access:";
    private static final String AUTH_TO_ACCESS = "auth_to_access:";
    ;
    private static final String AUTH = "auth:";
    private static final String REFRESH_AUTH = "refresh_auth:";
    private static final String ACCESS_TO_REFRESH = "access_to_refresh:";
    private static final String REFRESH = "refresh:";
    private static final String REFRESH_TO_ACCESS = "refresh_to_access:";
    private static final String CLIENT_ID_TO_ACCESS = "client_id_to_access:";
    private static final String UNAME_TO_ACCESS = "uname_to_access:";

    private RedisTemplate<String, Object> redisTemplate;

    public RedisTemplate<String, Object> getRedisTemplate() {
        return redisTemplate;
    }

    public void setRedisTemplate(RedisTemplate<String, Object> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    private AuthenticationKeyGenerator authenticationKeyGenerator
= new DefaultAuthenticationKeyGenerator();

    public void setAuthenticationKeyGenerator(AuthenticationKeyGenerator authenticationKeyGenerator) {
        this.authenticationKeyGenerator = authenticationKeyGenerator;
    }

    @Override
    public OAuth2AccessToken getAccessToken(OAuth2Authentication authentication) {
```

```
        String key = authenticationKeyGenerator.extractKey(authentication);
        OAuth2AccessToken accessToken = (OAuth2AccessToken) redisTemplate.opsForValue().get(AUTH_TO_ACCESS + key);
        if (accessToken != null
            && !key.equals(authenticationKeyGenerator.extractKey(readAuthentication(accessToken.getValue())))) {
            storeAccessToken(accessToken, authentication);
        }
        return accessToken;
    }

    @Override
    public OAuth2Authentication readAuthentication(OAuth2AccessToken token) {
        return readAuthentication(token.getValue());
    }

    @Override
    public OAuth2Authentication readAuthentication(String token) {
        return (OAuth2Authentication) this.redisTemplate.opsForValue().get(AUTH + token);
    }

    @Override
    public OAuth2Authentication readAuthenticationForRefreshToken(OAuth2RefreshToken token) {
        return readAuthenticationForRefreshToken(token.getValue());
    }

    public OAuth2Authentication readAuthenticationForRefreshToken(String token) {
        return (OAuth2Authentication) this.redisTemplate.opsForValue().get(REFRESH_AUTH + token);
    }

    @Override
    public void storeAccessToken(OAuth2AccessToken token, OAuth2Authentication authentication) {

        this.redisTemplate.opsForValue().set(ACCESS + token.getValue(), token);
    }
}
```

```
        this.redisTemplate.opsForValue().set(AUTH + token.getValue(), authentication);
        this.redisTemplate.opsForValue().set(AUTH_TO_ACCESS + authenticationKeyGenerator.extractKey(authentication), token);
        if (!authentication.isClientOnly()) {
            redisTemplate.opsForList().rightPush(UNAME_TO_ACCESS + getApprovalKey(authentication), token);
        }

        redisTemplate.opsForList().rightPush(CLIENT_ID_TO_ACCESS + authentication.getOAuth2Request().getClientId(), token);

        if (token.getExpiration() != null) {

            int seconds = token.getExpiresIn();
            redisTemplate.expire(ACCESS + token.getValue(), seconds, TimeUnit.SECONDS);
            redisTemplate.expire(AUTH + token.getValue(), seconds, TimeUnit.SECONDS);

            redisTemplate.expire(AUTH_TO_ACCESS + authenticationKeyGenerator.extractKey(authentication), seconds, TimeUnit.SECONDS);
            redisTemplate.expire(CLIENT_ID_TO_ACCESS + authentication.getOAuth2Request().getClientId(), seconds, TimeUnit.SECONDS);
            redisTemplate.expire(UNAME_TO_ACCESS + getApprovalKey(authentication), seconds, TimeUnit.SECONDS);
        }

        if (token.getRefreshToken() != null && token.getRefreshToken().getValue() != null) {
            this.redisTemplate.opsForValue().set(REFRESH_TO_ACCESS + token.getRefreshToken().getValue(), token.getValue());
            this.redisTemplate.opsForValue().set(ACCESS_TO_REFRESH + token.getValue(), token.getRefreshToken().getValue());
        }
    }

    private String getApprovalKey(OAuth2Authentication authentication) {
        String userName = authentication.getUserAuthentication() == null ? "" : authentication.getUserAuthentication()
            .getName();
        return getApprovalKey(authentication.getOAuth2Request().getClientId(), userName);
    }
}
```

```
    }

    private String getApprovalKey(String clientId, String userNa
e) {
    return clientId + (userName == null ? "" : ":" + userName);
}

@Override
public void removeAccessToken(OAuth2AccessToken accessToken) {
    removeAccessToken(accessToken.getValue());
}

@Override
public OAuth2AccessToken readAccessToken(String tokenValue) {
    return (OAuth2AccessToken) this.redisTemplate.opsForValue()
().get(ACCESS + tokenValue);
}

public void removeAccessToken(String tokenValue) {
    OAuth2AccessToken removed = (OAuth2AccessToken) redisTempl
ate.opsForValue().get(ACCESS + tokenValue);
    // caller to do that
    OAuth2Authentication authentication = (OAuth2Authenticatio
n) this.redisTemplate.opsForValue().get(AUTH + tokenValue);

    this.redisTemplate.delete(AUTH + tokenValue);
    redisTemplate.delete(ACCESS + tokenValue);
    this.redisTemplate.delete(ACCESS_TO_REFRESH + tokenValue);

    if (authentication != null) {
        this.redisTemplate.delete(AUTH_TO_ACCESS + authenti
cationKeyGenerator.extractKey(authentication));

        String clientId = authentication.getOAuth2Request().get
ClientId();
        redisTemplate.opsForList().leftPop(UNAME_TO_ACCESS + g
etApprovalKey(clientId, authentication.getName()));

        redisTemplate.opsForList().leftPop(CLIENT_ID_TO_ACCESS
+ clientId);

        this.redisTemplate.delete(AUTH_TO_ACCESS + authenti
cationKeyGenerator.extractKey(authentication));
    }
}
```

```
}

@Override
public void storeRefreshToken(OAuth2RefreshToken refreshToken,
OAuth2Authentication authentication) {
    this.redisTemplate.opsForValue().set(REFRESH + refreshToken.getValue(),
    refreshToken);
    this.redisTemplate.opsForValue().set(REFRESH_AUTH + refreshToken.getValue(),
    authentication);
}

@Override
public OAuth2RefreshToken readRefreshToken(String tokenValue) {
    return (OAuth2RefreshToken) this.redisTemplate.opsForValue()
    .get(REFRESH + tokenValue);
}

@Override
public void removeRefreshToken(OAuth2RefreshToken refreshToken) {
    removeRefreshToken(refreshToken.getValue());
}

public void removeRefreshToken(String tokenValue) {
    this.redisTemplate.delete(REFRESH + tokenValue);
    this.redisTemplate.delete(REFRESH_AUTH + tokenValue);
    this.redisTemplate.delete(REFRESH_TO_ACCESS + tokenValue);
}

@Override
public void removeAccessTokenUsingRefreshToken(OAuth2RefreshToken refreshToken) {
    removeAccessTokenUsingRefreshToken(refreshToken.getValue());
}

private void removeAccessTokenUsingRefreshToken(String refreshToken) {

    String token = (String) this.redisTemplate.opsForValue().get(
    REFRESH_TO_ACCESS + refreshToken);

    if (token != null) {
```

```
        redisTemplate.delete(ACCESS + token);
    }
}

@Override
public Collection<OAuth2AccessToken> findTokensByClientIdAndUs
erName(String clientId, String userName) {
    List<Object> result = redisTemplate.opsForList().range(UNA
ME_TO_ACCESS + getApprovalKey(clientId, userName), 0, -1);

    if (result == null || result.size() == 0) {
        return Collections.emptySet();
    }
    List<OAuth2AccessToken> accessTokens = new ArrayList<>(res
ult.size());

    for (Iterator<Object> it = result.iterator(); it.hasNext
();) {
        OAuth2AccessToken accessToken = (OAuth2AccessToken) it.
next();
        accessTokens.add(accessToken);
    }

    return Collections.unmodifiableCollection(accessTokens);
}

@Override
public Collection<OAuth2AccessToken> findTokensByClientId(Stri
ng clientId) {
    List<Object> result = redisTemplate.opsForList().range((CL
IENT_ID_TO_ACCESS + clientId), 0, -1);

    if (result == null || result.size() == 0) {
        return Collections.emptySet();
    }
    List<OAuth2AccessToken> accessTokens = new ArrayList<>(res
ult.size());
    for (Iterator<Object> it = result.iterator(); it.hasNext
();) {
        OAuth2AccessToken accessToken = (OAuth2AccessToken) it.
next();
        accessTokens.add(accessToken);
    }
}
```

```
        return Collections.unmodifiableCollection(accessTokens);
    }
}
```

## Docker 搭建 Redis-Cluster

### 镜像

---

```
git clone https://github.com/Grokzen/docker-redis-cluster
```

### 构建简单的集群

---

7000 - 7005 生产 3 + 3 的集群

```
cd docker-redis-cluster
docker-compose up
```

本地查看状态

```
redis -c -h 192.168.0.14 -p 7000 cluster nodes
```

会发现以上集群节点默认绑定 127.0.0.1，所以无法通过应用链接，需要对脚本进行改造

### 改造 docker-redis-cluster 脚本

---

#### docker-compose.yml

```
version: '2'
services:
  redis-cluster:
    build:
      context: .
    args:
      redis_version: '3.2.9'
    hostname: server
    privileged: true
    network_mode: "host"
    ports:
      - '7000-7007:7000-7007'
```

## [docker-entrypoint.sh](#)

修改 26 行的 IP 为公开 IP

```
IP=139.224.200.249
echo "yes" | ruby /redis/src/redis-trib.rb create --replicas 1 ${IP}:7000 ${IP}:7001 ${IP}:7002 ${IP}:7003 ${IP}:7004 ${IP}:7005
```

## redis-cluster.tmpl

```
bind 139.224.200.249
port ${PORT}
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
dir /redis-data/${PORT}
```

复制

## redis.tmpl

```
bind 139.224.200.249
port ${PORT}
appendonly yes
```

配置本地化

配置中心

---

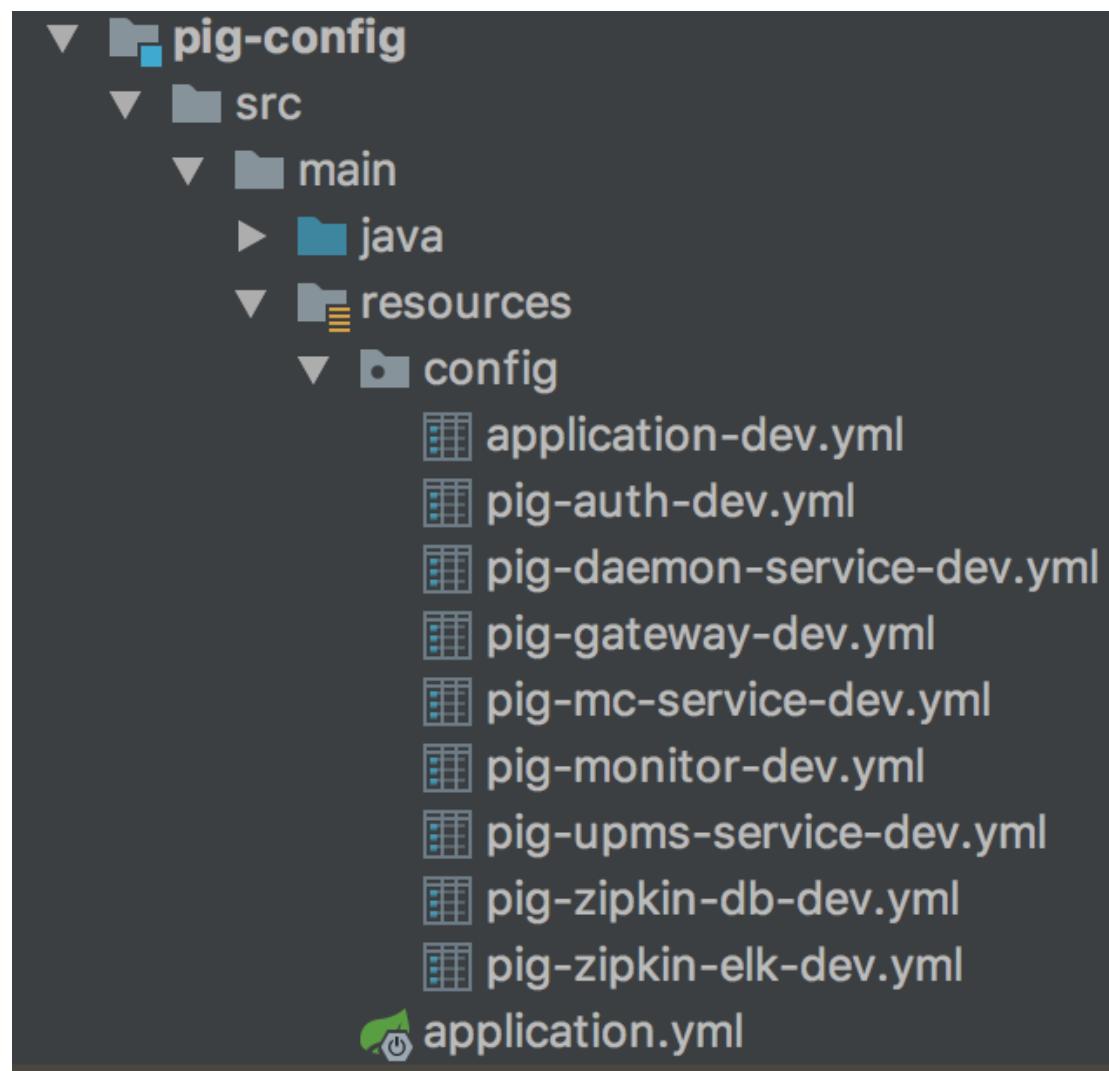
Spring Cloud Config 实现的配置中心默认采用 Git 来存储配置信息，所以使用 Spring Cloud Config 构建的配置服务器，天然就支持对微服务应用配置信息的版本管理，并且可以通过 Git 客户端工具来方便的管理和访问配置内容。当然它也提供了对其他存储方式的支持，比如：SVN 仓库、本地化文件系统。

**PS:** 虽然支持本地化的文件系统，但是不建议使用，因为配置中心的作用是集中配置和版本控制。

修改 pig-config:

```
spring:
  application:
    name: pig-config-server
  profiles:
    active: native #必须为 native
  cloud:
    config:
      server:
        native:
          search-locations: classpath:/config/
```

然后见 pig-config 的文件，放到 config 目录下即可实现



## pig 的 local 分支

---

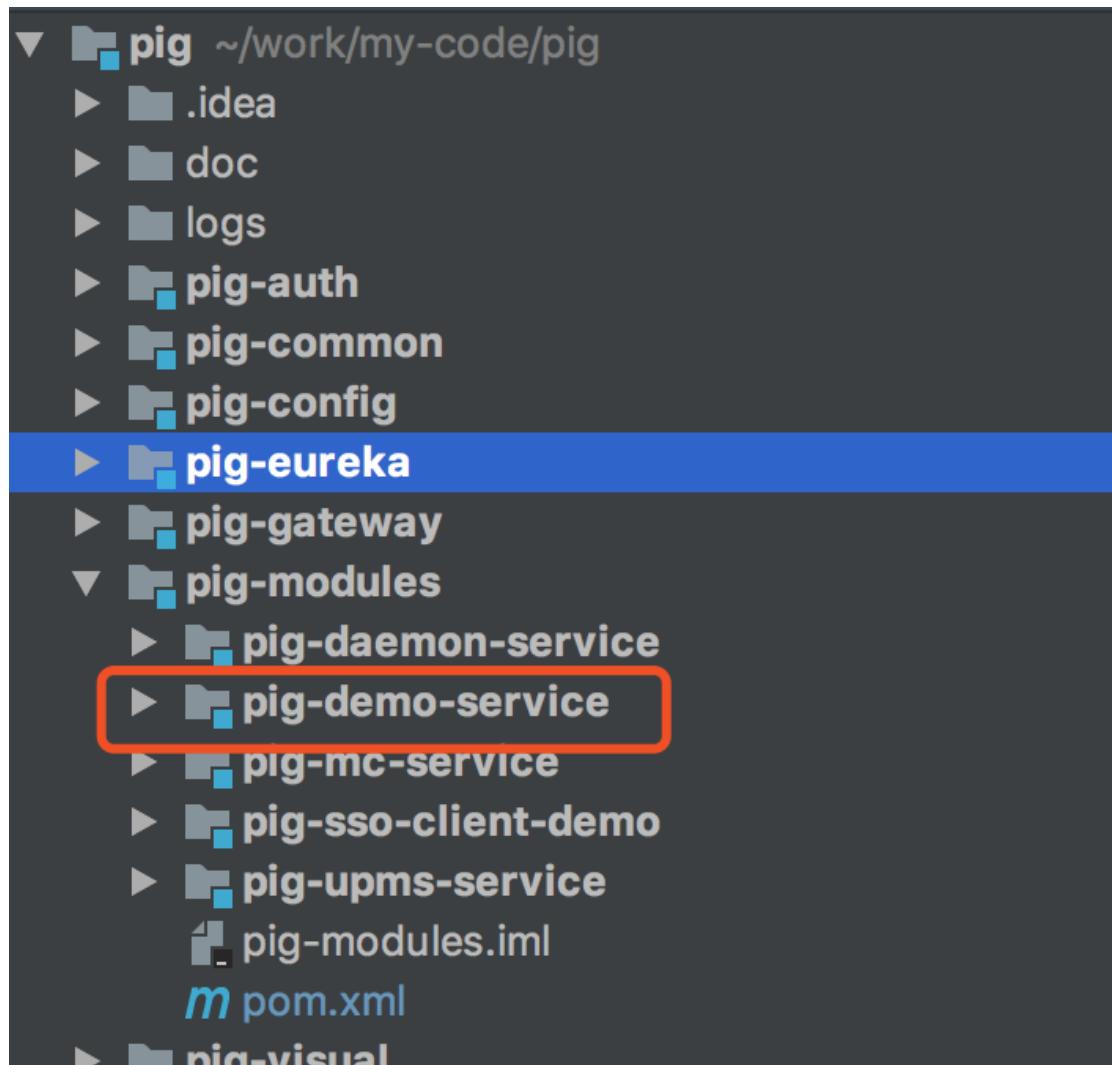
针对各位同学的建议，我在 `pig` 的 `local` 分支对此功能提供了支持。但是代码可能会和 `master` 存在断层次，建议按照上边步骤修改

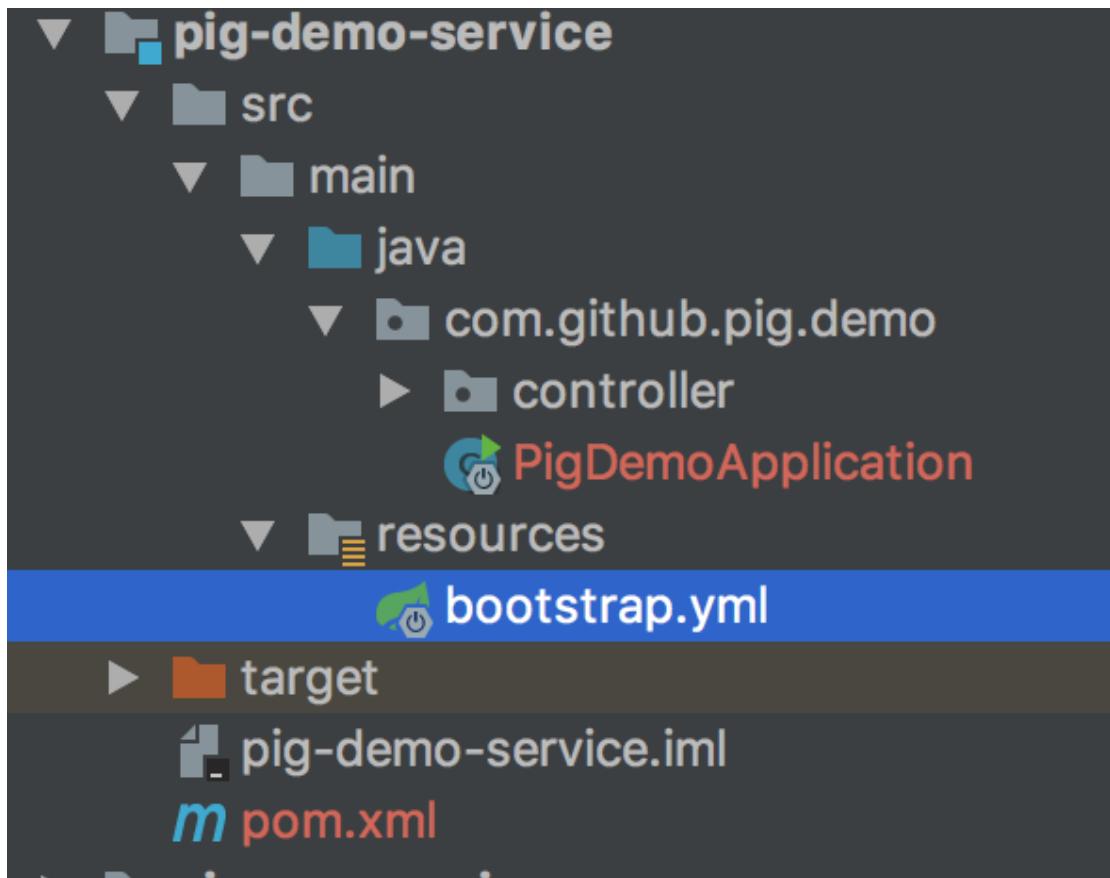
## 新增业务微服务（一）

### 开发业务微服务

---

新增一个 demo 模块，我建议业务模块放在 **pig-moduls** 下边维护。





## demo-pom 最小依赖

提供服务注册、配置中心、链路监控、redis 通用配置

```
<dependency>
    <groupId>com.github.pig</groupId>
    <artifactId>pig-common</artifactId>
    <version>${pig.version}</version>
</dependency>
<!--zipkin-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

## demo-bootstrap.yml

```
spring:
  application:
    name: pig-demo-service
  profiles:
```

```

active: dev
cloud:
  config:
    fail-fast: true
  discovery:
    service-id: pig-config-server
    enabled: true
  profile: ${spring.profiles.active}
  label: ${spring.profiles.active}

---
spring:
  profiles: dev
eureka:
  instance:
    prefer-ip-address: true
    lease-renewal-interval-in-seconds: 5
    lease-expiration-duration-in-seconds: 20
  client:
    serviceUrl:
      defaultZone: http://pig:gip6666@localhost:1025/eureka
    registry-fetch-interval-seconds: 10

---
spring:
  profiles: prd
eureka:
  instance:
    prefer-ip-address: true
  client:
    serviceUrl:
      defaultZone: http://pig:gip6666@eureka:1025/eureka

```

## 特别注意 Main 启动类

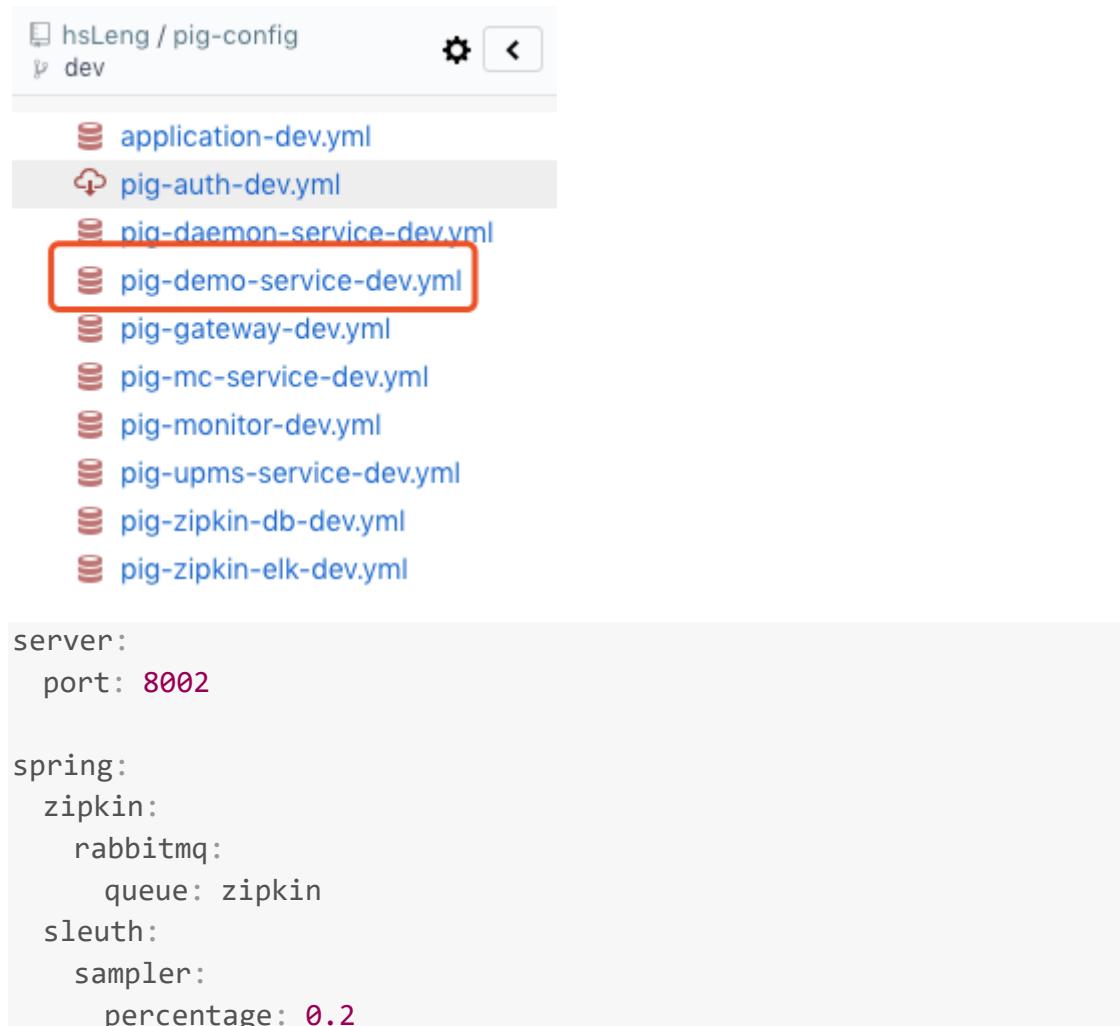
```

/**
 * @author lengleng
 * @date 2018年05月13日
 */
@SpringBootApplication
@EnableDiscoveryClient
@ComponentScan(basePackages = {"com.github.pig.demo", "com.github.pig.common.bean"})
public class PigDemoApplication {
  public static void main(String[] args) { SpringApplication.run(PigDemoApplication.class, args); }
}

```

一定要能扫描到 公共配置，才可以使用通用配置。

## 配置中心新增配置文件



```
hsLeng / pig-config
  dev

application-dev.yml
pig-auth-dev.yml
pig-daemon-service-dev.yml
pig-demo-service-dev.yml
pig-gateway-dev.yml
pig-mc-service-dev.yml
pig-monitor-dev.yml
pig-upms-service-dev.yml
pig-zipkin-db-dev.yml
pig-zipkin-elk-dev.yml

server:
  port: 8002

spring:
  zipkin:
    rabbitmq:
      queue: zipkin
  sleuth:
    sampler:
      percentage: 0.2
```

## 新增网关路由

新增

\* 服务名称: pig-demo

匹配路径: /demo/\*\*

转发地址: 请输入转发地址

\* 去掉前缀  否  是      \* 是否重试  否  是

\* 是否启用  否  是      敏感头: 请输入敏感头

新增 取消

## 新增业务微服务 (二)

### 配置微服务加入权限系统

根据微服务模块的接口进行权限切分

```
13
14  @RestController
15  public class DemoController {
16      @GetMapping("/user")
17      public UserVO demo(UserVO userVO) { return userVO; }
18  }
```

如上图请求路径为:

GET : http://localhost:9999/demo/user

Router Path: demo

## 配置菜单

添加 | 编辑 | 刪除

> 系统管理  
> 系统监控

父级节点	-1
节点ID	123
标题	测试菜单
权限标识	这个地方菜单不用填
图标	请输入图标
资源路径	请输入资源路径
请求方法	请输入资源请求类型
类型	菜单
排序	3
前端组件	Layout
前端地址	iframe嵌套地址

**保存** **取消**

## 配置按钮

资源路径 + 请求方法

满足 SpringMVC 的通配表达式

父级节点 123

节点ID 124

标题 测试接口

权限标识 和前端的 v-if 控制显示隐藏有关系，建议看前端开发章节

图标 请输入图标

资源路径 /demo/\* **SpringMVC的通配表达式**

请求方法 GET

类型 按钮

排序 3

前端组件 Layout

前端地址 iframe嵌套地址

**保存** **取消**

## 新增业务微服务（三）

### 基础知识

pig 前端使用 **Avue** 框架完成，同时支持原生的 **Element-ui**

- Avue
  - Avue, 一套为后端程序员准备的基于 element-ui 的快速开发框架 它的核心是数据驱动 UI 的思想，让我们从繁琐的 crud 开发中解脱出来，它的写法类似 easyUI。
  - [Avue 开发文档](#)
- ElementUI
  - Element, 一套为开发者、设计师和产品经理准备的基于 Vue 2.0 的桌面端组件库
  - [element-ui 开发文档](#)

### 开发前端（pig-ui）

- 模板工程
  - pig-ui/src/api/client.js (后端接口对接 js)
  - pig-ui/src/const/crud/client.js (前端 CURD 展示表格 JS)
  - pig-ui/src/views/admin/client/index.vue (vue 标签承载页面)
- 复制以上文件
  - pig-ui/src/api/test.js
  - pig-ui/src/const/crud/test.js
  - pig-ui/src/views/admin/test/index.vue
- 修改内容即可
- 后端菜单配置

注意红色标识字段，不然 404

1. 以上配置不要操作直接操作数据库 配置菜单，由于缓存缘故。
2. 前端添加页面 例如 `test/index.vue`， 请重启前端服务 重新加载。
3. 请求服务 403 被拦截等， 请检查 权限分配是否正确（URL 表达式是否匹配， 请求方法是否匹配等）

不要直接操作数据库，如果缓存和数据库不匹配 Redis flushdb，重启 Pig 服务吧

## zuul 动态路由实现

Zuul 是 Netflix 提供的一个开源组件,致力于在云平台上提供动态路由，监控，弹性，安全等边缘服务的框架。

配置文件配死（1.0.0 之前的版本）

---

```
zuul:  
  ignoredServices: '*'  
  host:  
    connect-timeout-millis: 30000  
    socket-timeout-millis: 30000  
  routes:  
    pig-auth:  
      path: /auth/**  
      serviceId: pig-auth  
      stripPrefix: true  
      sensitiveHeaders:  
    pig-upms-service:  
      path: /admin/**  
      serviceId: pig-upms-service  
      stripPrefix: true
```

```
sensitiveHeaders:
```

以上是 zuul 官方推荐的配置文件，pig 1.0.0 之前的版本也是这么实现。

## 动态路由

动态路由需要达到可持久化配置，动态刷新的效果。不仅要能满足从 spring 的配置文件 properties 加载路由信息，还需要从数据库加载我们的配置。另外一点是，路由信息在容器启动时就已经加载进入了内存，我们希望配置完成后，实施发布，动态刷新内存中的路由信息，达到不停机维护路由信息的效果。

看几行源码

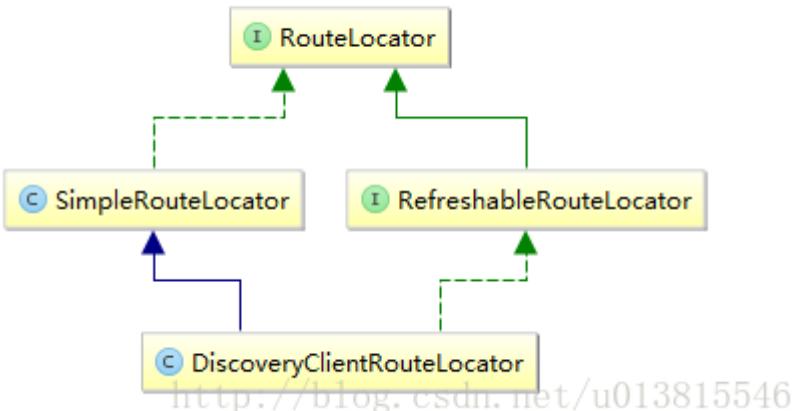
```
public class ZuulServerAutoConfiguration {  
    //核心类，默认的路由定位类  
    @Bean  
    @ConditionalOnMissingBean(SimpleRouteLocator.class)  
    public SimpleRouteLocator simpleRouteLocator() {  
        return new SimpleRouteLocator(this.server.getServerPrefix(),  
                                      this.zuulProperties);  
    }  
    //注册了一个路由刷新监听器，默认实现是 ZuulRefreshListener.class，这个是我们动态路由的关键  
    @Bean  
    public ApplicationListener<ApplicationEvent> zuulRefreshRoutesListener() {  
        return new ZuulRefreshListener();  
    }  
  
    //刷新路由的事件  
    private static class ZuulRefreshListener  
        implements ApplicationListener<ApplicationEvent> {  
  
        @Autowired  
        private ZuulHandlerMapping zuulHandlerMapping;  
  
        private HeartbeatMonitor heartbeatMonitor = new HeartbeatMonitor();  
    }  
}
```

```

    @Override
    public void onApplicationEvent(ApplicationEvent event) {
        if (event instanceof ContextRefreshedEvent
            || event instanceof RefreshScopeRefreshedEvent
            || event instanceof RoutesRefreshedEvent) {
            this.zuulHandlerMapping.setDirty(true);
        }
        else if (event instanceof HeartbeatEvent) {
            if (this.heartbeatMonitor.update(((HeartbeatEvent) event).getValue())) {
                this.zuulHandlerMapping.setDirty(true);
            }
        }
    }
}

```

我们要解决动态路由的难题，第一步就得理解路由定位器的作用。



DiscoveryClientRouteLocator 比 SimpleRouteLocator 多了两个功能，第一是从 DiscoveryClient（如 Eureka）发现路由信息，第二是实现了 RefreshableRouteLocator 接口，能够实现动态刷新。周期性刷新一次重写过后的自定义路由定位器如下：

```
/**
```

```
* @author lengleng
* @date 2018/5/15
* 动态路由实现
*/
@Slf4j
public class DynamicRouteLocator extends DiscoveryClientRouteLocator {
    private ZuulProperties properties;
    private RedisTemplate redisTemplate;

    public DynamicRouteLocator(String servletPath, DiscoveryClient discovery, ZuulProperties properties,
                               ServiceInstance localServiceInstance, RedisTemplate redisTemplate) {
        super(servletPath, discovery, properties, localServiceInstance);
        this.properties = properties;
        this.redisTemplate = redisTemplate;
    }

    /**
     * 重写路由配置
     * <p>
     * 1. properties 配置。
     * 2. eureka 默认配置。
     * 3. DB 数据库配置。
     *
     * @return 路由表
     */
    @Override
    protected LinkedHashMap<String, ZuulProperties.ZuulRoute> locateRoutes() {
        LinkedHashMap<String, ZuulProperties.ZuulRoute> routesMap
= new LinkedHashMap<>();
        //读取 properties 配置、eureka 默认配置
        routesMap.putAll(super.locateRoutes());
        log.debug("初始默认的路由配置完成");
        routesMap.putAll(locateRoutesFromDb());
        LinkedHashMap<String, ZuulProperties.ZuulRoute> values =
new LinkedHashMap<>();
        for (Map.Entry<String, ZuulProperties.ZuulRoute> entry : routesMap.entrySet()) {
            String path = entry.getKey();
            if (!path.startsWith("/")) {
```

```
        path = "/" + path;
    }
    if (StrUtil.isNotBlank(this.properties.getPrefix())) {
        path = this.properties.getPrefix() + path;
        if (!path.startsWith("/")) {
            path = "/" + path;
        }
    }
    values.put(path, entry.getValue());
}
return routesMap;
}

/**
 * Redis 中保存的，没有从 upms 拉去，避免启动链路依赖问题（取舍），  

网关依赖业务模块的问题
*
* @return
*/
private Map<String, ZuulProperties.ZuulRoute> locateRoutesFromDb() {
    Map<String, ZuulProperties.ZuulRoute> routes = new LinkedHashMap<>();

    Object obj = redisTemplate.opsForValue().get(CommonConstant.ROUTE_KEY);
    if (obj == null) {
        return routes;
    }

    List<SysZuulRoute> results = (List<SysZuulRoute>) obj;
    for (SysZuulRoute result : results) {
        if (StrUtil.isBlank(result.getPath()) && StrUtil.isBlank(result.getUrl())) {
            continue;
        }

        ZuulProperties.ZuulRoute zuulRoute = new ZuulProperties.ZuulRoute();
        try {
            zuulRoute.setId(result.getServiceId());
            zuulRoute.setPath(result.getPath());
            zuulRoute.setServiceId(result.getServiceId());
        }
    }
}
```

```

        zuulRoute.setRetryable(StrUtil.equals(result.getRetryable(), "0") ? Boolean.FALSE : Boolean.TRUE);
        zuulRoute.setStripPrefix(StrUtil.equals(result.getStripPrefix(), "0") ? Boolean.FALSE : Boolean.TRUE);
        zuulRoute.setUrl(result.getUrl());
        List<String> sensitiveHeadersList = StrUtil.splitTrim(result.getSensitiveheadersList(), ",");
        if (sensitiveHeadersList != null) {
            Set<String> sensitiveHeaderSet = CollUtil.newHashSet();
            sensitiveHeadersList.forEach(sensitiveHeader ->
                sensitiveHeaderSet.add(sensitiveHeader));
            zuulRoute.setSensitiveHeaders(sensitiveHeaderSet);
        }
        zuulRoute.setCustomSensitiveHeaders(true);
    }
} catch (Exception e) {
    log.error("从数据库加载路由配置异常", e);
}
log.debug("添加数据库自定义的路由配置, path: {}, serviceId:{}", zuulRoute.getPath(), zuulRoute.getServiceId());
routes.put(zuulRoute.getPath(), zuulRoute);
}
return routes;
}
}

```

OK, 对原有的 **DiscoveryClientRouteLocator** 增强完毕, 支持从 Db 中序列化的配置。

## 1. 说说这里为啥没有使用事件机制手动去刷新?

```

ContextRefreshedEvent
RefreshScopeRefreshedEvent
RoutesRefreshedEvent

```

因为继承了 **DiscoveryClientRouteLocator**, 可以自定的周期性刷新一次。pig 中调成 debug 即可发现。

## 2. 支持 zuul 的配置文件 , 还要默认机制 (通过服务名默认路由) , Db 配置。

## DB 中数据是如何加载到 Redis 中的

当 upms 启动完毕后，会触发

```
public class RouteConfigInitListener{
    @Autowired
    private RedisTemplate redisTemplate;
    @Autowired
    private SysZuulRouteService sysZuulRouteService;

    /**
     * Callback used to run the bean.
     * 初始化路由配置的数据，避免 gateway 依赖业务模块
     *
     */
    @EventListener(value = {EmbeddedServletContainerInitializedEvent.class})
    public void init() {
        log.info("开始初始化路由配置数据");
        EntityWrapper wrapper = new EntityWrapper();
        wrapper.eq(CommonConstant.DEL_FLAG, CommonConstant.STATUS_NORMAL);
        List<SysZuulRoute> routeList = sysZuulRouteService.selectList(wrapper);
        if (CollUtil.isNotEmpty(routeList)) {
            redisTemplate.opsForValue().set(CommonConstant.ROUTE_KEY, routeList);
            log.info("更新 Redis 中路由配置数据: {}条", routeList.size());
        }
        log.info("初始化路由配置数据完毕");
    }
}
```

建议使用 **EmbeddedServletContainerInitializedEvent** 事件，在启动完成后回调。

## 图形化配置

新增 ×

* 服务名称	请输入服务名称
匹配路径	请输入匹配路径
转发地址	请输入转发地址
* 去掉前缀 <input type="radio"/> 否 <input checked="" type="radio"/> 是	* 是否重试 <input type="radio"/> 否 <input checked="" type="radio"/> 是
* 是否启用 <input type="radio"/> 否 <input checked="" type="radio"/> 是	敏感头 <input type="text" value="请输入敏感头"/>

新增 取消

其实就是维护和之前配置文件相同的参数。

```
public static class ZuulRoute {  
  
    /**  
     * The ID of the route (the same as its map key by default).  
     */  
    private String id;  
  
    /**  
     * The path (pattern) for the route, e.g. /foo/**.  
     */  
    private String path;  
  
    /**  
     * The service ID (if any) to map to this route. You can specify a physical URL or  
     * a service, but not both.  
     */  
    private String serviceId;
```

```
 /**
     * A full physical URL to map to the route. An alternative is to use a service ID
     * and service discovery to find the physical address.
     */
    private String url;

    /**
     * Flag to determine whether the prefix for this route (the path, minus pattern
     * patcher) should be stripped before forwarding.
     */
    private boolean stripPrefix = true;

    /**
     * Flag to indicate that this route should be retryable (if supported). Generally
     * retry requires a service ID and ribbon.
     */
    private Boolean retryable;

    /**
     * List of sensitive headers that are not passed to downstream requests. Defaults
     * to a "safe" set of headers that commonly contain user credentials. It's OK to
     * remove those from the list if the downstream service is part of the same system
     * as the proxy, so they are sharing authentication data. If using a physical URL
     * outside your own domain, then generally it would be a bad idea to leak user
     * credentials.
     */
    private Set<String> sensitiveHeaders = new LinkedHashSet<>();
}
```

你可以选择重启 upms，或者直接点击同步直接生效



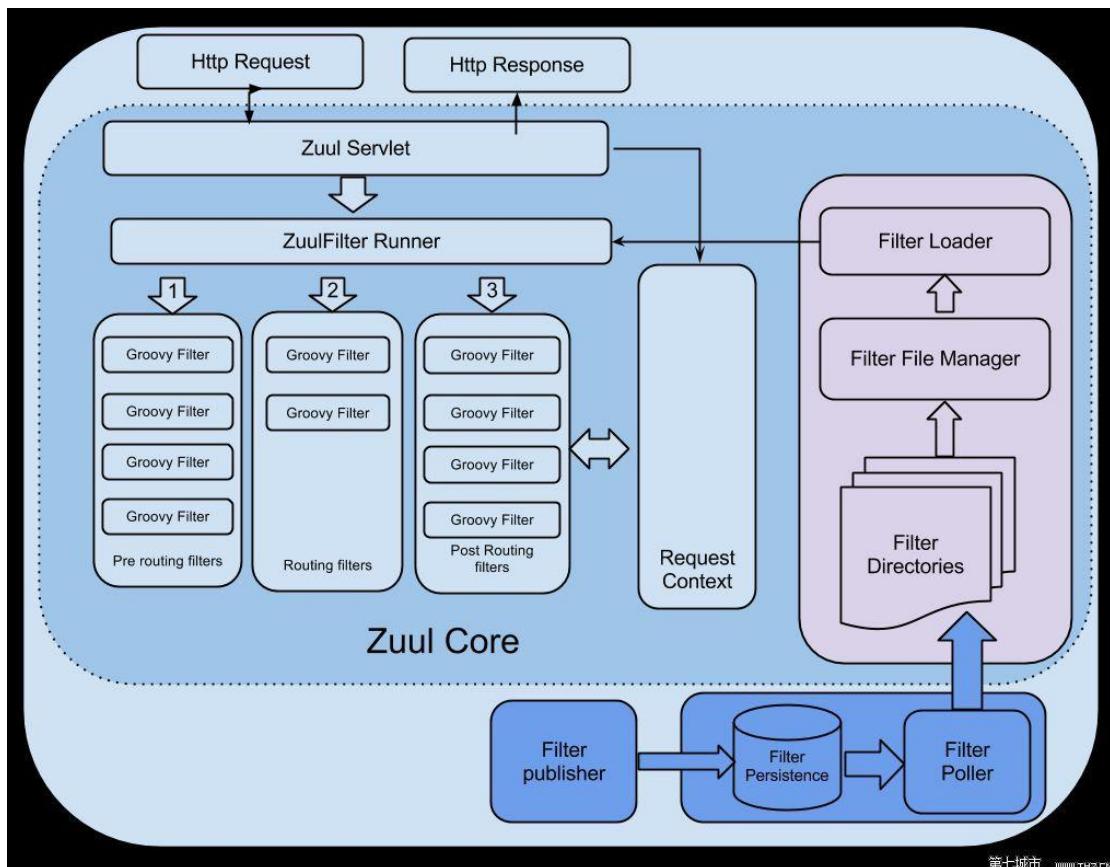
#	服务名称	匹配路径	转发地址	去掉前缀	是否重试	是否忽略失败	操作
1	pig-demo-service	/demo/**		是	是	否	<span>✓ 编辑</span> <span>删除</span>
2	pig-upms-service	/admin/**		是	是	否	<span>✓ 编辑</span> <span>删除</span>
3	pig-auth	/auth/**		是	是	否	<span>✓ 编辑</span> <span>删除</span>

共 3 条 20条/页 < 1 > 前往 1 页

## zuul 动态加载 Filter

### 什么是热加载 Filter

过滤器是由 Groovy 写成。这些过滤器文件被放在 Zuul Server 上的特定目录下面。Zuul 会定期轮询这些目录。修改过的过滤器会动态的加载到 Zuul Server 中以便于 request 使用。如下图右下角部分所示



## pig 扩展实现

网关配置 zuul.groovy.path 指定 groovy 脚本的上传目录。不配置则不启动热加载功能

核心逻辑是初始化 Zuul FilterFileManager 的加载路径

```
@Slf4j
@Component
@ConditionalOnProperty("zuul.groovy.path")
public class GroovyLoadInitListener {
    @Value("${zuul.groovy.path}")
    private String groovyPath;

    @EventListener(value = {EmbeddedServletContainerInitializedEvent.class})
    public void init() {
        MonitoringHelper.initMocks();
        FilterLoader.getInstance().setCompiler(new GroovyCompiler());
        FilterFileManager.setFilenameFilter(new GroovyFileFilter());
        try {
            FilterFileManager.init(10, groovyPath);
        } catch (Exception e) {
            log.error("初始化网关 Groovy 文件失败 {}", e);
        }
        log.warn("初始化网关 Groovy 文件成功");
    }
}
```

提供测试脚本，加载成功会输出 support by pig groovy 。脚本可在群共享获取

```
/**
 * @author lengleng
 * @date 2018/11/15
 *
 * 测试 groovy 脚本支持 filter 动态加载
 */
```

```
@Slf4j
class TestGroovyFilter extends ZuulFilter {
    /**
     * to classify a filter by type. Standard types in Zuul are "pre" for pre-routing filtering,
     * "route" for routing to an origin, "post" for post-routing filters, "error" for error handling.
     * We also support a "static" type for static responses see StaticResponseFilter.
     * Any filterType made be created or added and run by calling FilterProcessor.runFilters(type)
     *
     * @return A String representing that type
     */
    @Override
    String filterType() {
        return "pre"
    }

    /**
     * filterOrder() must also be defined for a filter. Filters may have the same filterOrder if precedence is not important for a filter. filterOrders do not need to be sequential.
     *
     * @return the int order of a filter
     */
    @Override
    int filterOrder() {
        return 0
    }

    /**
     * a "true" return from this method means that the run() method should be invoked
     *
     * @return true if the run() method should be invoked. false will not invoke the run() method
     */
    @Override
    boolean shouldFilter() {
        return true
    }
}
```

```
 /**
 * if shouldFilter() is true, this method will be invoked. this
method is the core method of a ZuulFilter
 *
 * @return Some arbitrary artifact may be returned. Current im-
plementation ignores it.
 * @throws ZuulException if an error occurs during execution.
 */
@Override
Object run() throws ZuulException {
    log.info("support by pig groovy")
    return null
}
}
```

## 企业级功能

### 分库分表

#### 关于 pig 分库分表实现

pig 的采用现在主流的当当网 Sharding-JDBC 实现分库分表。

[详细介绍](#)

#### 关于 Sharding-JDBC

Sharding-JDBC 是一个开源的分布式数据库中间件解决方案。它在 JDBC 层以对业务应用零侵入的方式额外提供数据分片，读写分离，柔性事务和分布式治理能力。并在其基础上提供封装了 MySQL 协议的服务端版本，用于完成对异构语言的支持。

基于 JDBC 的客户端版本定位为轻量级 Java 框架，使用客户端直连数据库，以 jar 包形式提供服务，无需額外部署和依赖，可理解为增强版的 JDBC 驱动，完全兼容 JDBC 和各种 ORM 框架

## pom 依赖

```
<dependency>
    <groupId>io.shardingjdbc</groupId>
    <artifactId>sharding-jdbc-core-spring-boot-starter</artifactId>
    <version>${sharding-jdbc-core-spring-boot-starter.version}</version>
</dependency>
```

## 配置分库分表策略

pig 框架对日志表进行分表进行示例.参看 `pig-upms` 配置

```
sharding:
  jdbc:
    datasource:
      names: ds
      ds:
        type: com.zaxxer.hikari.HikariDataSource
        driver-class-name: com.mysql.jdbc.Driver
        username: root
        password: lengleng
        jdbc-url: jdbc:mysql://106.14.69.75:3309/pig?characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&useSSL=false
    config:
      sharding:
        tables:
          sys_log:
            actual-data-nodes: ds.sys_log_${0..1}
            table-strategy:
              inline:
                sharding-column: id
                algorithm-expression: sys_log_${id % 2}
            key-generator-column-name: id
```

以上配置策略即可，实现日志根据主键的奇偶实现插入到 `sys_log_0`, `sys_log_1`

的表中。特别强调，实体的 ID 的生成策略，使用的 mybatis-plus 内置的雪花算法实现，`IdType.ID_WORKER`。

```
@Data
public class SysLog implements Serializable {
```

```
private static final long serialVersionUID = 1L;

/**
 * 编号
 */
@TableId(type = IdType.ID_WORKER)
private Long id;
}
```

## 任务调度

### 关于 pig 分布式任务调度

pig 的采用现在主流的当当网 Elastic-Job 实现分库分表。

[详细介绍](#)

### 关于 Elastic-Job

Elastic-Job 是一个分布式调度解决方案，由两个相互独立的子项目 Elastic-Job-Lite 和 Elastic-Job-Cloud 组成。

Elastic-Job-Lite 定位为轻量级无中心化解决方案，使用 jar 包的形式提供分布式任务的协调服务。

pig 采用的是轻量级的 Elastic-Job-Lite

### pom 依赖

```
<dependency>
    <groupId>com.github.xjzrc.spring.boot</groupId>
    <artifactId>elastic-job-lite-spring-boot-starter</artifactId>
    <version>1.0.1</version>
</dependency>
```

daemon 模块来负责整体的定时任务

---

提供以下实例

### 传统的 simple 任务

```
@Slf4j
@ElasticJobConfig(cron = "0/2 * * * * ?", shardingTotalCount = 3,
    shardingItemParameters = "0=pig1,1=pig2,2=pig3",
    startedTimeoutMilliseconds = 5000L,
    completedTimeoutMilliseconds = 10000L,
    eventTraceRdbDataSource = "dataSource")
public class DemoSimpleJob implements SimpleJob {
    /**
     * 业务执行逻辑
     *
     * @param shardingContext 分片信息
     */
    @Override
    public void execute(ShardingContext shardingContext) {
        log.info("-----");
    }
}
```

### 流任务（依赖任务）

```
/**
 * @author lengleng
 * @date 2018/2/8
 */
@ElasticJobConfig(cron = "0/2 * * * * ?", shardingTotalCount = 3,
shardingItemParameters = "0=Beijing,1=Shanghai,2=Guangzhou")
public class PigDataflowJob implements DataflowJob<Integer> {

    @Override
    public List<Integer> fetchData(ShardingContext shardingContext) {
        return null;
    }

    @Override
```

```
    public void processData(ShardingContext shardingContext, List<Integer> list) {  
        ...  
    }  
}
```

## 脚本类任务

```
@Slf4j  
@ElasticJobConfig(cron = "0/2 * * * * ?", scriptCommandLine = "demo.sh")  
public class DemoSimpleJob implements ScriptJob {}
```

## 关于管理台

---

elastic-job 提供运维平台

- 登录安全控制
- 注册中心、事件追踪数据源管理
- 快捷修改作业设置
- 作业和服务器维度状态查看
- 操作作业禁用\启用、停止和删除等生命周期
- 事件追踪查询

部署参考: <http://elasticjob.io/docs/elastic-job-lite/02-guide/web-console/>

## 链路追踪

### zipkin

#### zipkin 介绍

---

Zipkin 是 Twitter 的一个开源项目，它基于 Google Dapper 实现。我们可以使用它来收集各个服务器上请求链路的跟踪数据，并通过它提供的 REST API 接口来辅助我们查询跟踪数据以实现对分布式系统的监控程序，从而及时地发现系统中出现的延迟升高问题并找出系统性能瓶颈的根源。除了面向开发的 API 接口之外，它也提供了方便的 UI 组件来帮助我们直观的搜索跟踪信息和分析请求链路明细，比如：可以查询某段时间内各用户请求的处理时间等。

#### 学习资源

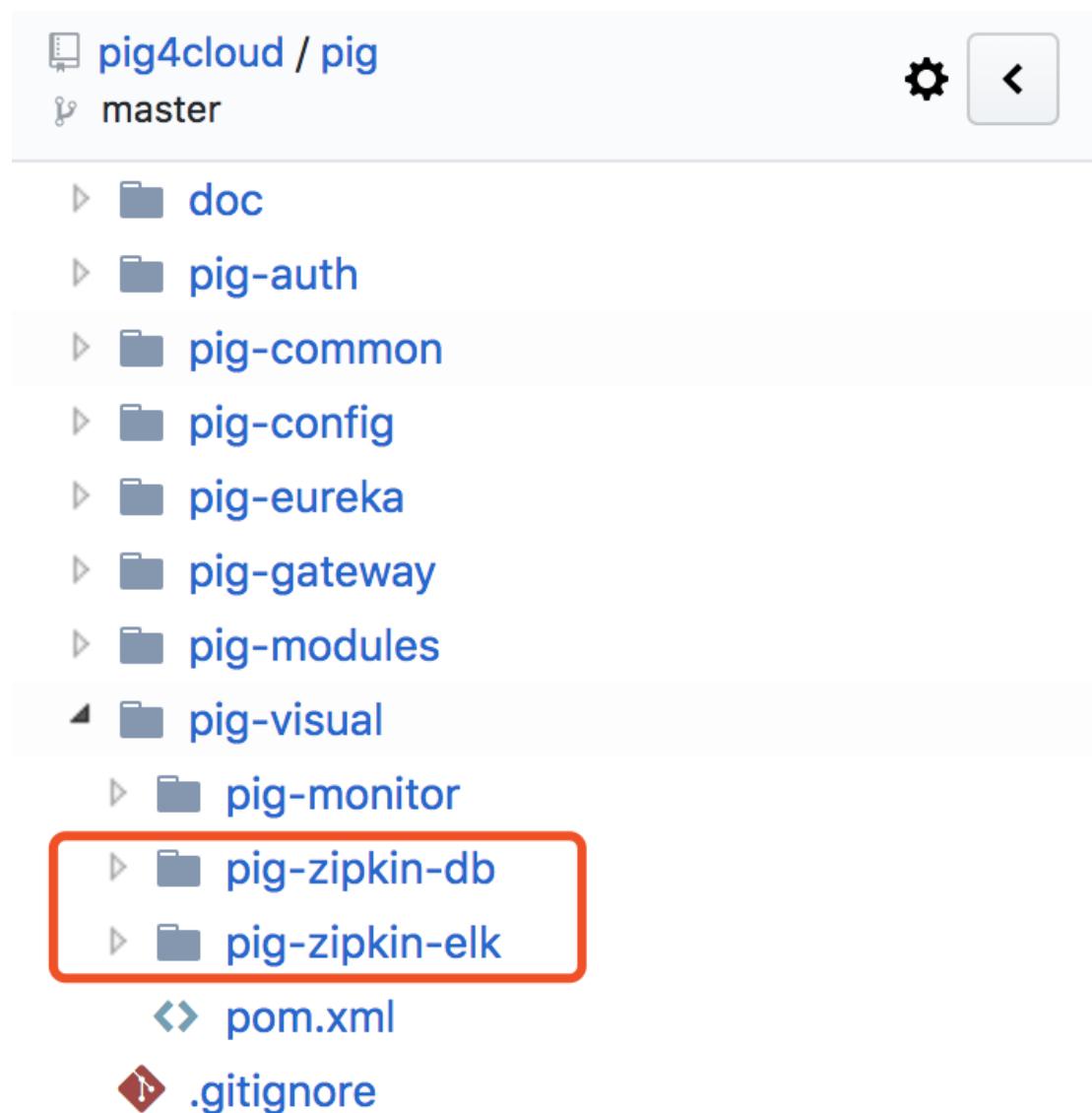
---

推荐一篇比较详细的入门整合教程：

<http://blog.didispace.com/spring-cloud-starter-dalston-8-4/>

## pig 中支持的两种

pig 同时支持了 mysql 和 elasticsearch 的整合方案



## zipkin-mysql

整合了 MySQL 以后，链路调用信息会保存在以下表中。

	<b>zipkin_annotations</b>
	<b>zipkin_dependencies</b>
	<b>zipkin_spans</b>

## zipkin-elasticsearch

重点！新增加的微服务如何接入 pig 的链路追踪

pom 中依赖 zipkin 的即可。

```
<!--zipkin-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

## pinpoint

安装视频请参考李寻欢录制的在 pig 群。李寻欢使用的最新版本 pinpoint，这里有部分不一样。

环境说明：操作系统为 macOS ， JDK 为 1.8

## 所需软件

<b>hbase-1.2.6-bin.tar.gz</b>	
<b>pinpoint-1.6.2.zip</b>	pinpoint 源码，使用 hbase-create.
<b>hbase 脚本初始化 hbase</b>	
<b>pinpoint-collector-1.6.2.war</b>	collector 模块，使用 tomcat 进行部署

pinpoint-web-1.6.2.war	web 管控台，使用 tomcat 进行部署
pinpoint-agent-1.6.2.tar.gz	不需要部署，在被监控应用一端
apache-tomcat-8.5.15.zip	

准备两个 tomcat 环境，一个部署 pinpoint-collector-1.6.2.war，另一个部署

pinpoint-web-1.6.2.war

## 安装 hbase

### 环境配置

解压 hbase-1.2.6-bin.tar.gz，修改 hbase/conf/hbase-site.xml 文件内容如下

```
<configuration>
    <property>
        <name>hbase.rootdir</name>
        <value>file:///pinpoint/data/hbase</value>
    </property>
    <property>
        <name>hbase.zookeeper.property.dataDir</name>
        <value>/pinpoint/data/zookeeper</value>
    </property>
    <property>
        <name>hbase.master.port</name>
        <value>60000</value>
    </property>
    <property>
        <name>hbase.regionserver.port</name>
        <value>60020</value>
    </property>
</configuration>
```

修改 hbase/conf/hbase-env.sh 文件，设置 JAVA\_HOME 环境变量

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home
```

## 启动 hbase

```
hbase/bin/start-hbase.sh
```

执行 jps 查看是否有 HMaster 进程

访问 <http://127.0.0.1:16010/master-status> 查看 hbase web 管控台

导入 hbase 初始化脚步

```
hbase/bin/hbase shell /pinpoint-1.6.2/hbase/scripts/hbase-create.hbase
.
.
.
0 row(s) in 1.4160 seconds

0 row(s) in 1.2420 seconds

0 row(s) in 2.2410 seconds

0 row(s) in 1.2400 seconds

0 row(s) in 1.2450 seconds

0 row(s) in 1.2420 seconds

0 row(s) in 1.2360 seconds

0 row(s) in 1.2400 seconds

0 row(s) in 1.2370 seconds

0 row(s) in 2.2450 seconds

0 row(s) in 4.2480 seconds

0 row(s) in 1.2310 seconds

0 row(s) in 1.2430 seconds

0 row(s) in 1.2390 seconds

0 row(s) in 1.2440 seconds

0 row(s) in 1.2440 seconds
```

```
TABLE
AgentEvent
AgentInfo
AgentLifeCycle
AgentStat
AgentStatV2
ApiMetaData
ApplicationIndex
ApplicationMapStatisticsCallee_Ver2
ApplicationMapStatisticsCaller_Ver2
ApplicationMapStatisticsSelf_Ver2
ApplicationTraceIndex
HostApplicationMap_Ver2
SqlMetaData_Ver2
StringMetaData
TraceV2
Traces
16 row(s) in 0.0190 seconds
```

进入 hbase shell

```
hbase/bin/hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.2.6, rUnknown, Mon May 29 02:25:32 CDT 2017
hbase(main):001:0>
```

输入 status 'detailed'查看刚才初始化的表，是否存在

```
hbase(main):001:0> status 'detailed'
```

访问 <http://127.0.0.1:16010/master-status> 查看 hbase web 管控台

**Backup Masters**

ServerName	Port	Start Time
Total:0		

**Tables**

User Tables		System Tables	Snapshots				
16 table(s) in set. [Details]							
Namespace	Table Name	Online Regions	Offline Regions	Failed Regions	Split Regions	Other Regions	Description
default	AgentEvent	1	0	0	0	0	'AgentEvent', {NAME => 'E', DATA_BLOCK_ENCODING => 'PREFIX', TTL => '5184000 SECONDS (60 DAYS)'}
default	AgentInfo	1	0	0	0	0	'AgentInfo', {NAME => 'Info', DATA_BLOCK_ENCODING => 'PREFIX', TTL => '31536000 SECONDS (365 DAYS)'}
default	AgentLifeCycle	1	0	0	0	0	'AgentLifeCycle', {NAME => 'S', DATA_BLOCK_ENCODING => 'PREFIX', TTL => '5184000 SECONDS (60 DAYS)'}
default	AgentStat	16	0	0	0	0	'AgentStat', {NAME => 'S', DATA_BLOCK_ENCODING => 'PREFIX', TTL => '5184000 SECONDS (60 DAYS)'}
default	AgentStatV2	64	0	0	0	0	'AgentStatV2', {NAME => 'S', DATA_BLOCK_ENCODING => 'PREFIX', TTL => '5184000 SECONDS (60 DAYS)'}

## 部署 collector

解压 apache-tomcat-8.5.15.zip 并重命名为 collector

拷贝 pinpoint-collector-1.6.2.war 到 collector\webapps\目录下并重命名为

ROOT.war

涉及到两个配置文件

- **hbase.properties**

由于使用的为 hbase 自带的 zookeeper 即 hbase 和 zookeeper 在同一台机器上，

则不需要修改 collector\webapps\ROOT\WEB-INF\classes\hbase.properties 文件

- **pinpoint-collector.properties**

注意观察该配置文件中中的三个端口 9994、9995、9996 默认不需要修改，其中

9994 为 collector 监听的 tcp 端口，9995 为 collector 监听的 udp 端口

执行 **collector/bin/startup.sh** 启动 collector

部署 web 管控台

解压 apache-tomcat-8.5.15.zip 并重命名为 web

拷贝 `pinpoint-web-1.6.2.war` 到 `web\webapps\` 目录下并重命名为 `ROOT.war`

同样留意 `web\webapps\ROOT\WEB-INF\classes\hbase.properties` 中的配置，若与 `hbase` 在同一台机器则不需要修改

执行 `web/bin/startup.sh` 启动 web 管控台(注意修改 tomcat 端口号，防止冲突，这里修改端口为 8088)

浏览器访问 <http://127.0.0.1:8088/> 查看 pinpoint 管控台

## 监控 spring boot 应用

这里有一个 `spring-boot-example.jar` 的应用，现在要使用 pinpoint 来对其监控跟踪。操作很简单，分两步

解压 `pinpoint-agent-1.6.2.tar.gz` 并对其进行配置

```
tar zxvf pinpoint-agent-1.6.2.tar.gz -C pinpoint-agent
```

修改 `pinpoint-agent/pinpoint.config` 中的配置与 `collector` 服务一致。此处因为 `pinpoint-agent` 与 `collector` 在同一台机器，因此默认配置即可不需要修改。

为 `spring-boot` 应用配置 `pinpoint-agent`

启动 `spring-boot` 应用时添加如下参数

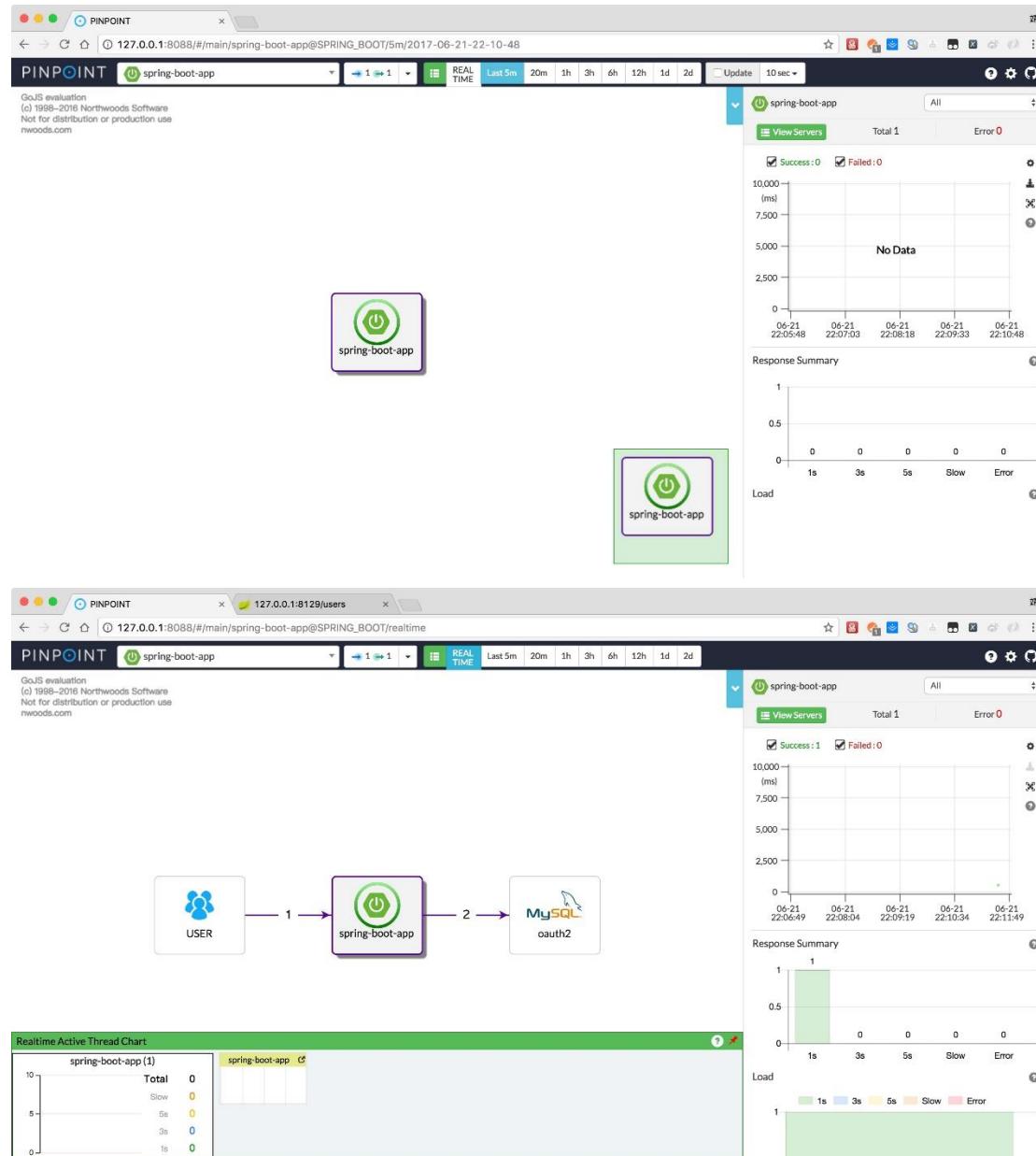
```
-javaagent:$AGENT_PATH/pinpoint-bootstrap-$VERSION.jar  
-Dpinpoint.agentId  
-Dpinpoint.applicationName
```

本例中的启动参数为

```
java -javaagent:/pinpoint-agent/pinpoint-bootstrap-1.6.2.jar -  
Dpinpoint.agentId=spring-boot-app -  
Dpinpoint.applicationName=spring-boot-app -jar spring-boot-  
docker-example-1.0.jar
```

访问 <http://127.0.0.1:8088/>，第一次访问可能没有数据，可以先访问下应用，然后

在刷新 pinpoint 管控台即可



## 缓存管理

# CacheCloud

CacheCloud 提供一个 Redis 云管理平台：实现多种类型(Redis Standalone、Redis Sentinel、Redis Cluster)自动部署、解决 Redis 实例碎片化现象、提供完善

统计、监控、运维功能、减少运维成本和误操作，提高机器的利用率，提供灵活的伸缩性，提供方便的接入客户端。

项目主页：<https://github.com/sohutv/cachecloud>

The screenshot displays the CacheCloud monitoring dashboard. At the top, there's a navigation bar with tabs like '应用统计信息', '实例列表', '应用详情', etc., and search/filter options for dates (2018-01-01 to 2018-01-02) and a '查询' button.

**全局信息** (Global Information):

内存使用率	0.00G Used/0.50G Total	当前连接数	5
应用主节点数	1	应用从节点数	0
应用命中率	0.0%	当前对象数	08
应用当前状态	运行中	应用分布机器节点数	1

**各命令峰值信息** (Peak Command Information):

命令	峰值QPM	峰值产生时间
auth	372	2017-12-31 16:11:02
exec	15	2017-12-31 16:56:01
exists	12	2017-12-31 16:56:02
incrby	08	2017-12-31 16:56:04
get	09	2017-12-31 16:56:02

**命令统计** (Command Statistics):

A pie chart titled '命令分布' (Command Distribution) showing the distribution of various Redis commands. The largest segment is 'auth:730' (blue), followed by 'get:9' (green), 'exists:12' (red), 'incrby:12' (yellow), and 'exec:15' (black).

**全局统计** (Global Statistics):

On the left, a sidebar menu includes '全局统计' (selected), '流程审批', '用户管理', 'Quartz管理', '机器管理', '客户端异常统计', '客户端版本统计', '系统通知', 'Redis配置模板管理', 'Redis报警阈值', and '系统配置管理'.

The main panel shows '全局统计' (Global Statistics) with tables for machine memory and application memory usage.

集群当前可对外提供空间: 0.00G

应用ID: 10000 应用名: ?? 应用类型: redis-standalone 内存详情: 0.00G Used/0.50G Total 命中率: 无 已运行时间(天): 0 申请状态: 运行中 操作: 应用下线, 应用运维

## 改造 RedisConnectionFactory

```
/**
 * 根据缓存策略的不同，RedisConnectionFactory 不同
 * 示例是单机模式。
 *
 * @return
 */
@Bean
public RedisConnectionFactory redisConnectionFactory() {
```

```
while (true) {
    try {
        LOCK.tryLock(100, TimeUnit.MILLISECONDS);
        /**
         * 心跳返回的请求为空;
         */
        String response = HttpUtils.doGet("http://localhost:5005/cache/client/redis/standalone/10000.json?clientVersion=1.0-SNAPSHOT");
        if (response == null || response.isEmpty()) {
            continue;
        }
        JSONObject jsonObject = null;
        try {
            jsonObject = JSONObject.parseObject(response);
        } catch (Exception e) {
        }
        if (jsonObject == null) {
            continue;
        }
        /**
         * 从心跳中提取 HostAndPort，构造 JedisPool 实例;
         */
        String instance = jsonObject.getString("standalone");
        String[] instanceArr = instance.split(":");
        if (instanceArr.length != 2) {
            continue;
        }

        //收集上报数据
        ClientDataCollectReportExecutor.getInstance("http://localhost:5005/cachecloud/client/reportData.json");

        String password = jsonObject.getString("password");
        String host = instanceArr[0];
        String port = instanceArr[1];

        JedisConnectionFactory jedisConnectionFactory = new JedisConnectionFactory();
        jedisConnectionFactory.setPassword(password);
        jedisConnectionFactory.setHostName(host);
        jedisConnectionFactory.setPort(Integer.parseInt(port));
    }
    return jedisConnectionFactory;
}
```

```
        } catch (InterruptedException e) {
            logger.error("error in build()", e);
        }

    }
}
```

改造 jedis-2.9.0

---

## Connection.java

```
/***
 * 命令捕获，异常保存
 * @param cmd
 * @param args
 */
public void sendCommand(final ProtocolCommand cmd, final byte[]...
args) {
    try {
        //统计开始
        UsefulDataModel costModel = UsefulDataModel.getCostModel(t
hreadLocal);
        costModel.setCommand(cmd.toString().toLowerCase());
        costModel.setStartTime(System.currentTimeMillis());
        connect();
        Protocol.sendCommand(outputStream, cmd, args);
    } catch (JedisConnectionException ex) {
        UsefulDataCollector.collectException(ex, getHostPort(), Sy
stem.currentTimeMillis());
        broken = true;
        throw ex;
    }
}
```

## JedisClusterCommand.java

```
private T runWithRetries(byte[] key, int attempts, boolean tryRand
omNode, boolean asking) {
    if (attempts <= 0) {
        JedisClusterMaxRedirectsException exception = new Jedis
ClusterMaxRedirectsException("Too many Cluster redirections? ke
y=" + SafeEncoder.encode(key));
        //收集
    }
}
```

```
        UsefulDataCollector.collectException(exception, "", System.currentTimeMillis(), ClientExceptionType.REDIS_CLUSTER);
        throw exception;
    }
}
```

## 更新 spring-boot-starter-data-redis 依赖

```
<!--Redis-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <exclusions>
        <exclusion>
            <artifactId>jedis</artifactId>
            <groupId>redis.clients</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.sohu.tv</groupId>
    <artifactId>cachecloud-open-client-redis</artifactId>
    <version>1.0-SNAPSHOT</version>
    <exclusions>
        <exclusion>
            <artifactId>jedis</artifactId>
            <groupId>redis.clients</groupId>
        </exclusion>
    </exclusions>
</dependency>
<!--上步改造后编译的 jar-->
<dependency>
    <groupId>com.github.pig</groupId>
    <artifactId>pig-cache-cloud-jedis</artifactId>
    <version>2.9.1</version>
</dependency>
```

复制

## 部署服务 war

这一步直接参考 cachecloud 的文档即可

# 文件系统

七牛

## yml 参数配置

### 七牛基本参数

The screenshot shows the 'AccessKey/SecretKey' management section of the Qiniu console. It displays two sets of keys: one set for '2015-11-17' and another set for '2015-11-17'. The first set has an AK (Access Key) value of 'hM2cBDEMOFTYzpXbigRW90kV12NhhzhfM3jCzurJ' and a SK (Secret Key) value of '.....'. The second set has an AK value of 'hM2cBDEMOFTYzpXbigRW90kV12NhhzhfM3jCzurJ' and a SK value of '.....'. Both sets are marked as '使用中' (In Use). A red box highlights the AK and SK fields for the first key.

### bucket

The screenshot shows the 'Storage Space Management' interface of the Qiniu console. On the left, there is a sidebar with icons for 'New Storage Space', 'Search Storage Space', and 'Storage Space List'. The 'Storage Space List' section shows two buckets: 'data' and 'pigcloud', both highlighted with red boxes. The main area is titled 'data' and contains tabs for 'Space Overview', 'Data Statistics', 'Content Management', and 'Mirror Storage'. Under 'Space Overview', there is a chart titled 'File Storage' showing data usage over time. The chart shows a peak at 40 MB on March 26th. Below the chart, there are buttons for 'Standard Storage' and 'Low Frequency Storage', with 'Standard Storage' currently selected. A red box highlights the 'data' bucket in the list.

qiniu:

```
accessKey: accessKey
secretKey: secretKey
bucket: bucket
qiniuHost: http://p0hpm86wj.bkt.clouddn.com/
    @Autowired
    private QiniuPropertiesConfig qiniuPropertiesConfig;
    /**
     * 上传用户头像
     * (多机部署有问题，建议使用独立的文件服务器)
     *
     * @param file 资源
     * @param request request
     * @return filename map
     */
    @PostMapping("/upload")
    public Map<String, String> upload(@RequestParam("file") MultipartFile file, HttpServletRequest request) {
        String fileExt = FileUtil.extName(file.getOriginalFilename());
        Configuration cfg = new Configuration(Zone.zone0());
        UploadManager uploadManager = new UploadManager(cfg);
        String key = RandomUtil.randomUUID() + "." + fileExt;
        Auth auth = Auth.create(qiniuPropertiesConfig.getAccessKey(),
                qiniuPropertiesConfig.getSecretKey());
        String upToken = auth.uploadToken(qiniuPropertiesConfig.getBucket());
        try {
            uploadManager.put(file.getInputStream(), key, upToken,
                    null, null);
        } catch (Exception e) {
            logger.error("文件上传异常", e);
            throw new RuntimeException(e);
        }
    }
}
```

## fastdfs

### 什么是 FastDFS

---

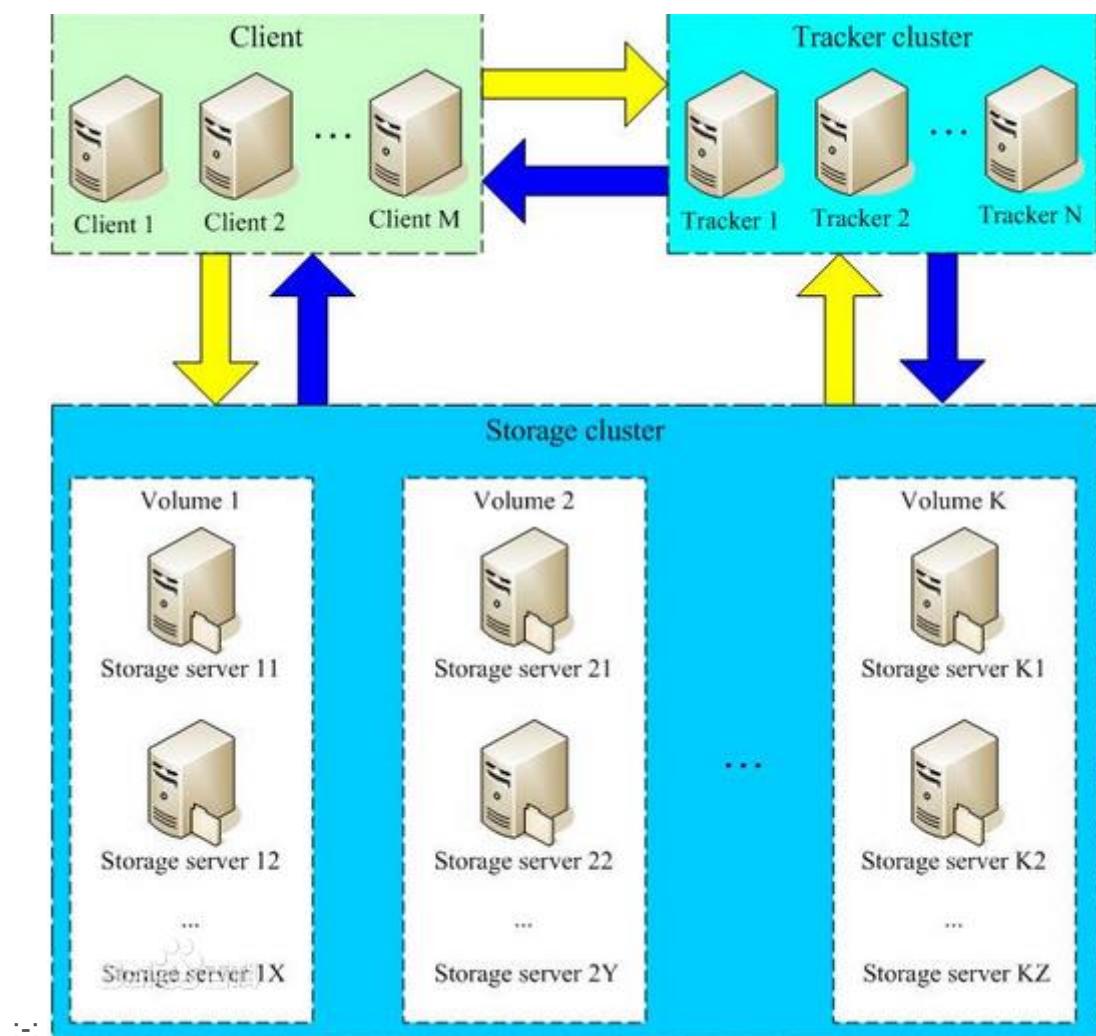
百度百科：FastDFS 是一个开源的轻量级分布式文件系统，它对文件进行管理，功能包括：文件存储、文件同步、文件访问（文件上传、文件下载）等，解决了大

容量存储和负载均衡的问题。特别适合以文件为载体的在线服务，如相册网站、视频网站等等。

FastDFS 为互联网量身定制，充分考虑了冗余备份、负载均衡、线性扩容等机制，并注重高可用、高性能等指标，使用 FastDFS 很容易搭建一套高性能的文件服务器集群提供文件上传、下载等服务。

FastDFS 是由余庆所开发的开源、免费的分布式文件系统，GitHub 项目地址:<https://github.com/happyfish100/fastdfs/>.

其基本架构图如下：



## 手动安装

### FastDFS 安装

软件包	版本
FastDFS	v5.05
libfastcommon	v1.0.7

下载安装 libfastcommon

- 下载

```
wget https://github.com/happyfish100/libfastcommon/archive/V1.0.7.tar.gz
```

- 解压

```
tar -xvf V1.0.7.tar.gz  
cd libfastcommon-1.0.7
```

- 编译、安装

```
./make.sh  
./make.sh install
```

- 创建软链接

```
ln -s /usr/lib64/libfastcommon.so /usr/local/lib/libfastcommon.so  
ln -s /usr/lib64/libfastcommon.so /usr/lib/libfastcommon.so  
ln -s /usr/lib64/libfdfclient.so /usr/local/lib/libfdfclient.so  
ln -s /usr/lib64/libfdfclient.so /usr/lib/libfdfclient.so
```

## 下载安装 FastDFS

- 下载 FastDFS

```
 wget https://github.com/happyfish100/fastdfs/archive/V5.05.tar.gz
```

- 解压

```
 tar -xvf V5.05.tar.gz  
 cd fastdfs-5.05
```

- 编译、安装

```
 ./make.sh  
 ./make.sh install
```

## 配置 Tracker 服务

上述安装成功后，在/etc/目录下会有一个 fdfs 的目录，进入它。会看到三个.sample 后缀的文件，这是作者给我们的示例文件，我们需要把其中的 tracker.conf.sample 文件改为 tracker.conf 配置文件并修改它：

```
 cp tracker.conf.sample tracker.conf  
 vi tracker.conf
```

编辑 tracker.conf

```
# 配置文件是否不生效, false 为生效  
disabled=false  
  
# 提供服务的端口  
port=22122  
  
# Tracker 数据和日志目录地址  
base_path=/home/data/fastdfs  
  
# HTTP 服务端口  
http.server_port=80
```

创建 tracker 基础数据目录，即 `base_path` 对应的目录

```
mkdir -p /home/data/fastdfs
```

使用 `ln -s` 建立软链接

```
ln -s /usr/bin/fdfs_trackerd /usr/local/bin  
ln -s /usr/bin/stop.sh /usr/local/bin  
ln -s /usr/bin/restart.sh /usr/local/bin
```

启动服务

```
service fdfs_trackerd start
```

查看监听

```
netstat -unltp | grep fdfs
```

如果看到 22122 端口正常被监听后，这时候说明 Tracker 服务启动成功啦！

tracker server 目录及文件结构

Tracker 服务启动成功后，会在 `base_path` 下创建 `data`、`logs` 两个目录。目录结构

如下：

```
 ${base_path}  
   |__data  
   |   |__storage_groups.dat: 存储分组信息  
   |   |__storage_servers.dat: 存储服务器列表  
   |__logs  
   |   |__trackerd.log: tracker server 日志文件
```

配置 Storage 服务

进入 `/etc/fdfs` 目录，复制 FastDFS 存储器样例配置文件 `storage.conf.sample`，

并重命名为 `storage.conf`

```
# cd /etc/fdfs  
# cp storage.conf.sample storage.conf  
# vi storage.conf
```

编辑 `storage.conf`

```
# 配置文件是否不生效, false 为生效  
disabled=false  
  
# 指定此 storage server 所在 组(卷)
```

```
group_name=group1

# storage server 服务端口
port=23000

# 心跳间隔时间，单位为秒（这里是指主动向 tracker server 发送心跳）
heart_beat_interval=30

# Storage 数据和日志目录地址(根目录必须存在，子目录会自动生成)
base_path=/home/data/fastdfs/storage

# 存放文件时 storage server 支持多个路径。这里配置存放文件的基路径数目，通常只配一个目录。
store_path_count=1

# 逐一配置 store_path_count 个路径，索引号基于 0。
# 如果不配置 store_path0，那它就和 base_path 对应的路径一样。
store_path0=/home/data/fastdfs/storage

# FastDFS 存储文件时，采用了两级目录。这里配置存放文件的目录个数。
# 如果本参数只为 N（如： 256），那么 storage server 在初次运行时，会在
# store_path 下自动创建 N * N 个存放文件的子目录。
subdir_count_per_path=256

# tracker_server 的列表，会主动连接 tracker_server
# 有多个 tracker server 时，每个 tracker server 写一行
tracker_server=192.168.1.190:22122

# 允许系统同步的时间段（默认是全天）。一般用于避免高峰同步产生一些问题而设定。
sync_start_time=00:00
sync_end_time=23:59
```

使用 ln -s 建立软链接

```
ln -s /usr/bin/fdfs_storaged /usr/local/bin
```

启动服务

```
service fdfs_storaged start
```

查看监听

```
netstat -unltp | grep fdfs
```

启动 Storage 前确保 Tracker 是启动的。初次启动成功，会在 /home/data/fastdfs/storage 目录下创建 data、 logs 两个目录。如果看到 23000 端口正常被监听后，这时候说明 Storage 服务启动成功啦！

查看 Storage 和 Tracker 是否在通信

```
/usr/bin/fdfs_monitor /etc/fdfs/storage.conf
```

## FastDFS 配置 Nginx 模块

软件包	版本
openresty	v1.13.6.1
fastdfs-nginx-module	v1.1.6

FastDFS 通过 Tracker 服务器，将文件放在 Storage 服务器存储，但是同组存储服务器之间需要进行文件复制，有同步延迟的问题。

假设 Tracker 服务器将文件上传到了 192.168.1.190，上传成功后文件 ID 已经返回给客户端。此时 FastDFS 存储集群机制会将这个文件同步到同组存 192.168.1.190，在文件还没有复制完成的情况下，客户端如果用这个文件 ID 在 192.168.1.190 上取文件，就会出现文件无法访问的错误。而 fastdfs-nginx-module 可以重定向文件链接到源服务器取文件，避免客户端由于复制延迟导致的文件无法访问错误。

下载 安装 Nginx 和 fastdfs-nginx-module:

推荐您使用 yum 安装以下的开发库:

```
yum install readline-devel pcre-devel openssl-devel -y
```

下载最新版本并解压:

```
wget https://openresty.org/download/openresty-1.13.6.1.tar.gz
```

```
tar -xvf openresty-1.13.6.1.tar.gz  
  
wget https://github.com/happyfish100/fastdfs-nginx-module/archive/master.zip  
  
unzip master.zip
```

配置 nginx 安装，加入 fastdfs-nginx-module 模块：

```
./configure --add-module=../fastdfs-nginx-module-master/src/
```

编译、安装：

```
make && make install
```

查看 Nginx 的模块：

```
/usr/local/openresty/nginx/sbin/nginx -v
```

有下面这个就说明添加模块成功

```
[root@iz2ze7tgu9zb2gr6av1tysz home]# nginx -V  
nginx version: openresty/1.13.6.1  
built by gcc 4.8.5 20160623 (Red Hat 4.8.5-16) (GCC)  
built with OpenSSL 1.0.2k-fips 26 Jan 2017  
TLS SNI support enabled  
configure arguments: --prefix=/usr/local/openresty/nginx --with-cc-opt=-O2 --add-module=../echo-nginx-module-0.61 --add-module=../xss-nginx-module-0.02 --add-module=../set-mime-type-nginx-module-0.3.1 --add-module=../headers-more-nginx-module-0.33 --add-module=../http-filesystem-nginx-module-0.3.1 --add-module=../http-lua-nginx-module-0.10.11 --add-module=../rpx-lua-upstream-0.07 --add-module=../headers-more-nginx-module-0.33 --add-module=../redis-nginx-module-0.14 --add-module=../redis-nginx-module-0.3.1 --add-module=../rds-nginx-module-0.05 --add-module=../rds-csv-nginx-module-0.08 --add-module=../ngx_stream_lua-0.0.3 --with-ld-opt=-Wl,-rpath,/usr/local/openresty/luajit/lib --add-module=../home-fastdfs-nginx-module/src --with-stream --with-stream_ssl_module --with-http_ssl_module
```

复制 fastdfs-nginx-module 源码中的配置文件到/etc/fdfs 目录，并修改：

```
cp /fastdfs-nginx-module/src/mod_fastdfs.conf /etc/fdfs/
```

```
# 连接超时时间  
connect_timeout=10  
  
# Tracker Server  
tracker_server=192.168.1.190:22122  
  
# StorageServer 默认端口  
storage_server_port=23000  
  
# 如果文件 ID 的 uri 中包含/group**, 则要设置为 true  
url_have_group_name = true  
  
# Storage 配置的 store_path0 路径，必须和 storage.conf 中的一致  
store_path0=/home/data/fastdfs/storage
```

复制 FastDFS 的部分配置文件到/etc/fdfs 目录：

```
cp /fastdfs-nginx-module/src/http.conf /etc/fdfs/  
cp /fastdfs-nginx-module/src/mime.types /etc/fdfs/
```

配置 nginx，修改 nginx.conf：

```
location ~/group([0-9])/M00 {  
    ngx_fastdfs_module;  
}
```

启动 Nginx:

```
[root@iz2ze7tgu9zb2gr6av1tysz sbin]# ./nginx  
ngx_http_fastdfs_set pid=9236
```

测试上传:

```
[root@iz2ze7tgu9zb2gr6av1tysz fdfs]# /usr/bin/fdfs_upload_file /etc/fdfs/client.conf /etc/fdfs/4.jpg  
group1/M00/00/00/rBD8EFqVACuAI9mcAAC_ornlYSU088.jpg
```

## docker 安装 FastDFS

镜像选择

alpine:3.6

### Dockerfile

```
# 使用超小的 Linux 镜像 alpine  
FROM alpine:3.6  
  
ENV HOME /root  
  
# 安装准备  
RUN apk update \  
    && apk add --no-cache --virtual .build-deps bash gcc libc-dev make openssl-dev pcre-dev zlib-dev linux-headers curl gnupg libxslt-dev gd-dev geoip-dev  
  
# 下载 fastdfs、libfastcommon、nginx 插件的源码  
RUN cd /root \  
    && curl -fSL https://github.com/happyfish100/libfastcommon/archive/V1.0.36.tar.gz -o fastcommon.tar.gz \  
    && curl -fSL https://codeload.github.com/happyfish100/fastdfs/tar.gz/V5.11 -o fastfs.tar.gz \  
    && curl -fSL https://github.com/happyfish100/fastdfs-nginx-module/archive/master.tar.gz -o nginx-module.tar.gz \  
    && tar zxf fastcommon.tar.gz \  
    && tar zxf fastfs.tar.gz \  
    && tar zxf nginx-module.tar.gz
```

```
&& tar zxf fastfs.tar.gz \
&& tar zxf nginx-module.tar.gz

# 安装 libfastcommon
RUN cd ${HOME}/libfastcommon-1.0.36/ \
    && ./make.sh \
    && ./make.sh install

# 安装 fastdfs v5.11
RUN cd ${HOME}/fastdfs-5.11/ \
    && ./make.sh \
    && ./make.sh install

# 配置 fastdfs: base_dir
RUN cd /etc/fdfs/ \
    && cp storage.conf.sample storage.conf \
    && cp tracker.conf.sample tracker.conf \
    && cp client.conf.sample client.conf \
    && sed -i "s|/home/pig/fastdfs|/var/local/fdfs/tracker|g" /etc \
/fdfs/tracker.conf \
    && sed -i "s|/home/pig/fastdfs|/var/local/fdfs/storage|g" /etc \
/fdfs/storage.conf \
    && sed -i "s|/home/pig/fastdfs|/var/local/fdfs/storage|g" /etc \
/fdfs/client.conf

# 获取 nginx 源码，与 fastdfs 插件一起编译
RUN cd ${HOME} \
    && curl -fSL http://nginx.org/download/nginx-1.12.2.tar.gz -o \
nginx-1.12.2.tar.gz \
    && tar zxf nginx-1.12.2.tar.gz \
    && chmod u+x ${HOME}/fastdfs-nginx-module-master/src/config \
    && cd nginx-1.12.2 \
    && ./configure --add-module=${HOME}/fastdfs-nginx-module-maste \
r/src \
    && make && make install

# 设置 nginx 和 fastdfs 联合环境，并配置 nginx
RUN cp ${HOME}/fastdfs-nginx-module-master/src/mod_fastdfs.conf / \
etc/fdfs/ \
    && sed -i "s|^store_path0.*$|store_path0=/var/local/fdfs/stora \
ge|g" /etc/fdfs/mod_fastdfs.conf \
    && sed -i "s|^url_have_group_name =.*$|url_have_group_name = t \
rue|g" /etc/fdfs/mod_fastdfs.conf \
    && cd ${HOME}/fastdfs-5.11/conf/ \
```

```
&& cp http.conf mime.types anti-steal.jpg /etc/fdfs/ \
&& echo -e "\
events {\n\
    worker_connections 1024;\n\
}\n\
http {\n\
    include mime.types;\n\
    default_type application/octet-stream;\n\
    server {\n\
        listen 9999;\n\
        server_name localhost;\n\
\n\
        location ~ /group[0-9]/M00 {\n\
            ngx_fastdfs_module;\n\
        }\n\
    }\n\
}">/usr/local/nginx/conf/nginx.conf

# 清理文件
RUN rm -rf ${HOME}/*
RUN apk del .build-deps gcc libc-dev make openssl-dev linux-headers curl gnupg libxslt-dev gd-dev geoip-dev
RUN apk add bash pcre-dev zlib-dev

# 配置启动脚本，在启动时中根据环境变量替换 nginx 端口、fastdfs 端口
# 默认 nginx 端口
ENV WEB_PORT 9999
# 默认 fastdfs 端口
ENV FDFS_PORT 22122

# 创建启动脚本
RUN echo -e "\
mkdir -p /var/local/fdfs/storage/data /var/local/fdfs/tracker; \
ln -s /var/local/fdfs/storage/data/ /var/local/fdfs/storage/data/M00; \
sed -i \"s/listen\ .*\$/listen\ \$WEB_PORT;/g\" /usr/local/nginx/conf/nginx.conf; \
sed -i \"s/http.server_port=.*$/http.server_port=\$WEB_PORT/g\" /etc/fdfs/storage.conf; \
if [ \"\$IP\" = \"\" ]; then \
    IP=\`ifconfig eth0 | grep inet | awk '{print \$2}' | awk -F: '{print \$2}'\``; \
fi \
"
```

```

sed -i \"s/^tracker_server=.*/$tracker_server=\$IP:\$FDFS_PORT/g\
" /etc/fdfs/client.conf; \
sed -i \"s/^tracker_server=.*/$tracker_server=\$IP:\$FDFS_PORT/g\
" /etc/fdfs/storage.conf; \
sed -i \"s/^tracker_server=.*/$tracker_server=\$IP:\$FDFS_PORT/g\
" /etc/fdfs/mod_fastdfs.conf; \
/etc/init.d/fdfs_trackerd start; \
/etc/init.d/fdfs_storaged start; \
/usr/local/nginx/sbin/nginx; \
tail -f /usr/local/nginx/logs/access.log \
">/start.sh \
&& chmod u+x /start.sh

# 暴露端口。改为采用 host 网络，不需要单独暴露端口
# EXPOSE 9999 22122

ENTRYPOINT ["/bin/bash","/start.sh"]

```

## docker-compose.yaml

```

version: '3.0'
services:
  fastdfs:
    build: .
    image: docker.sxpcsj.cn/fastdfs:5.11
    # 该容器是否需要开机启动+自动重启。若需要，则取消注释。
    restart: always
    container_name: fastdfs
    environment:
      # nginx 服务端口
      - WEB_PORT=9999
      # docker 所在主机的 IP 地址
      - IP=123.206.94.20
    volumes:
      # 将本地目录映射到 docker 容器内的 fastdfs 数据存储目录
      - ./fdfs:/var/local/fdfs
    # 使 docker 具有 root 权限以读写主机上的目录
    privileged: true
    # 网络模式为 host，即直接使用主机的网络接口
    network_mode: "host"

    # 此处采用 host 方式网络，即容器共用主机的网络设置。
    # 若采用 bridge 方式网络，容器内部设置为

```

```
  ports:  
    - "9999:9999"  
    - "22122:22122"
```

## 启动

```
docker-compose up -d  
# 因为要下载软件包和源码，并编译，所以过程比较漫长，期间可能会出现红字的  
warning，不必理会。若报错，请根据提示排查。  
Creating fastdfs ...  
Creating fastdfs ... done
```

## 测试

```
# 进入 docker 容器内终端  
$ docker exec -it fastdfs /bin/bash  
  
bash-4.3# echo "Hello FastDFS!">index.html  
bash-4.3# fdfs_test /etc/fdfs/client.conf upload index.html  
This is FastDFS client test program v5.11  
...  
[2018-04-05 16:12:12] DEBUG - base_path=/var/local/fdfs/storage, c  
onnect_timeout=30, network_timeout=60, tracker_server_count=1, an  
ti_steal_token=0, anti_steal_secret_key length=0, use_connection_  
pool=0, g_connection_pool_max_idle_time=3600s, use_storage_id=0,  
storage server id count: 0  
  
tracker_query_storage_store_list_without_group:  
    server 1. group_name=, ip_addr=123.206.94.20, port=23000  
  
group_name=group1, ip_addr=123.206.94.20, port=23000  
storage_upload_by_filename  
group_name=group1, remote_filename=M00/00/00/CmkgdVrGStyAfoXSAAAA  
D4GB4k029.html  
source ip address: 10.105.32.117  
file timestamp=2018-04-05 16:12:12  
file size=15  
file crc32=2172772941  
example file url: http://123.206.94.20/group1/M00/00/00/CmkgdVrGS  
tyAfoXSAAAAD4GB4k029.html  
storage_upload_slave_by_filename  
group_name=group1, remote_filename=M00/00/00/CmkgdVrGStyAfoXSAAAA  
D4GB4k029_big.html  
source ip address: 10.105.32.117
```

```
file timestamp=2018-04-05 16:12:12
file size=15
file crc32=2172772941
example file url:
http://123.206.94.20/group1/M00/00/00/CmkgdVrGStyAfoXSAAAAD4GB4k0
29_big.ht
```

## 系统使用

### yml 参数配置

```
fdfs:
  file-host: http://123.206.94.20:9999/      #nginx 暴露请求地址
  tracker-list:
    - 123.206.94.20:22122                      #fastdfs 服务配置
```

### 代码调用

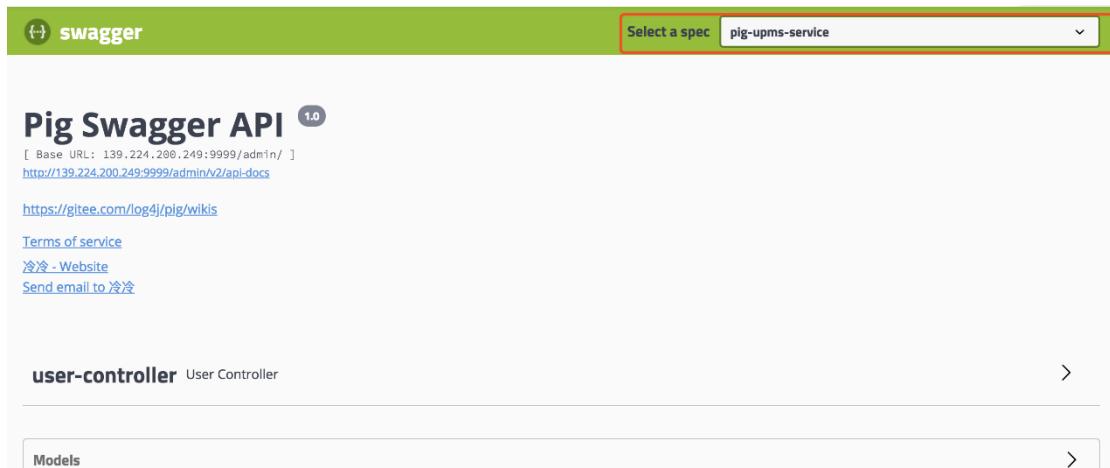
```
//注入 fastdfs 的客户端
@Autowired
private FastFileStorageClient fastFileStorageClient;

// 使用客户端上传
fastFileStorageClient.uploadFile(file.getBytes(), fileExt);
```

## 聚合文档

为什么称为聚合文档?

pig 通过在网关配置 swagger，通过和 zuul 整合，实现一次配置，其他微服务模块复用在一个界面，效果如下：



The screenshot shows the Pig Swagger API interface. At the top, there is a green header bar with the text "swagger" and a "Select a spec" dropdown menu. The dropdown menu is currently set to "pig-upms-service". Below the header, the main title is "Pig Swagger API 1.0". It includes a base URL: "[ Base URL: 139.224.200.249:9999/admin/ ] http://139.224.200.249:9999/admin/v2/api-docs". There are also links to "https://gitee.com/log4j/pig/wikis", "Terms of service", "冷冷 - Website", and "Send email to 冷冷". The main content area shows a list of services: "user-controller User Controller" and "Models". Each item has a right-pointing arrow icon next to it.

通过红色区域的下拉框，选择不同模块，就可以实现文档查看。

什么原理?

```
@Component
@Primary
public class RegistrySwaggerResourcesProvider implements SwaggerResourcesProvider {
    private final RouteLocator routeLocator;

    public RegistrySwaggerResourcesProvider(RouteLocator routeLocator) {
        this.routeLocator = routeLocator;
    }

    @Override
    public List<SwaggerResource> get() {
        List<SwaggerResource> resources = new ArrayList<>();

        List<Route> routes = routeLocator.getRoutes();
        routes.forEach(route -> {
            //授权不维护到 swagger
        });
    }
}
```

```
        if (!StringUtils.contains(route.getId(), ServiceNameConstant.AUTH_SERVICE)){
            resources.add(swaggerResource(route.getId(), route.getFullPath().replace("**", "v2/api-docs")));
        }
    });

    return resources;
}

private SwaggerResource swaggerResource(String name, String location) {
    SwaggerResource swaggerResource = new SwaggerResource();
    swaggerResource.setName(name);
    swaggerResource.setLocation(location);
    swaggerResource.setSwaggerVersion("2.0");
    return swaggerResource;
}
}
```

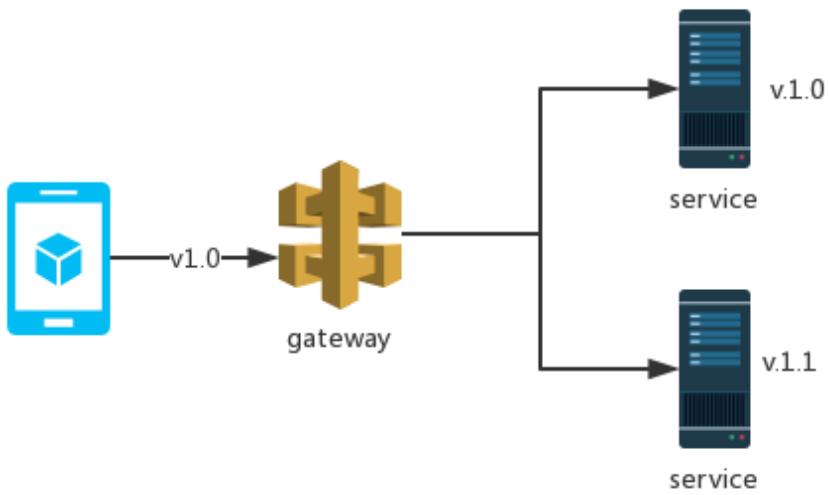
核心配置如上，扩展 **SwaggerResourcesProvider**，根据 **RouteLocator** 获取路由信息，实现聚合文档

## 灰度发布

### 实现原理

---

根据 eureka 客户端 metadata 进行自定义版本数据，然后使用 ribbon 根据请求 header 中版本信息对该版本数据进行过滤和匹配，选择匹配的微服务节点。



## 使用

---

1. 微服务中定义版本信息：

```
eureka:  
  instance:  
    metadata-map:  
      version: v1.0  
eureka:  
  instance:  
    metadata-map:  
      version: v1.1
```

2. 网关配置中开启灰度路由

```
zuul.ribbon.metadata.enabled=true
```

### 3. 客户端指定目标版本

```
// HTTPRequest拦截
axios.interceptors.request.use(config => {
  NProgress.start() // start progress bar
  if (store.getters.access_token) {
    config.headers['Authorization'] = 'Bearer ' + getToken() // 让
    config.headers['version'] = 'v1.1'
  }
  return config
}, error => {
  return Promise.reject(error)
})
```

### 4. 结果

访问的都是 1.1 版本的微服务

## 生产部署

### Docker 部署

#### docker-compose

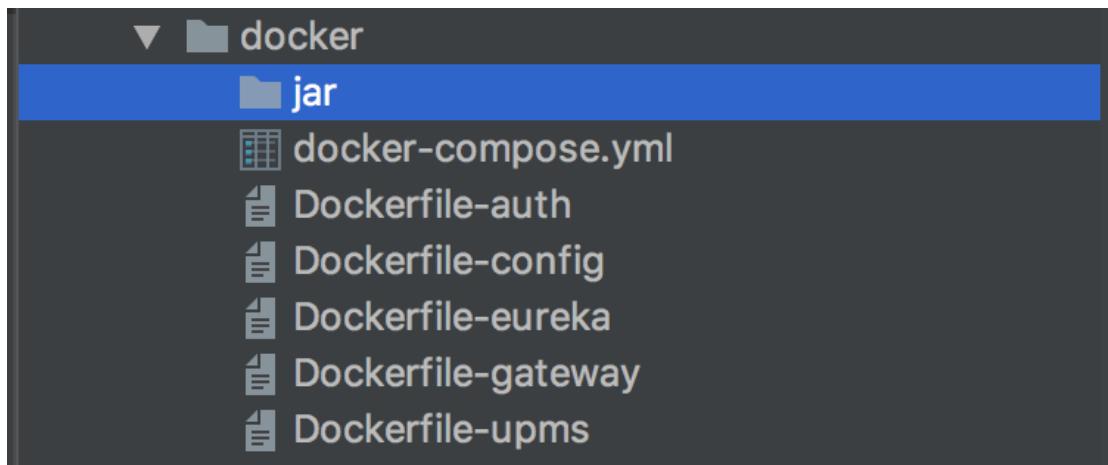
Docker Compose 是 Docker 官方编排（Orchestration）项目之一，负责快速的部署分布式应用。

安装 docker-compose

```
$ sudo curl -L https://github.com/docker/compose/releases/download/1.21.1/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

将基础模块的 jar 放入到当前目录

---



执行

---

cd 到 docker-compose 所在目录执行

```
docker-compose up
```

使用 docker-compose 上线比较慢，建议等待 5-10 分钟

生产集群推荐

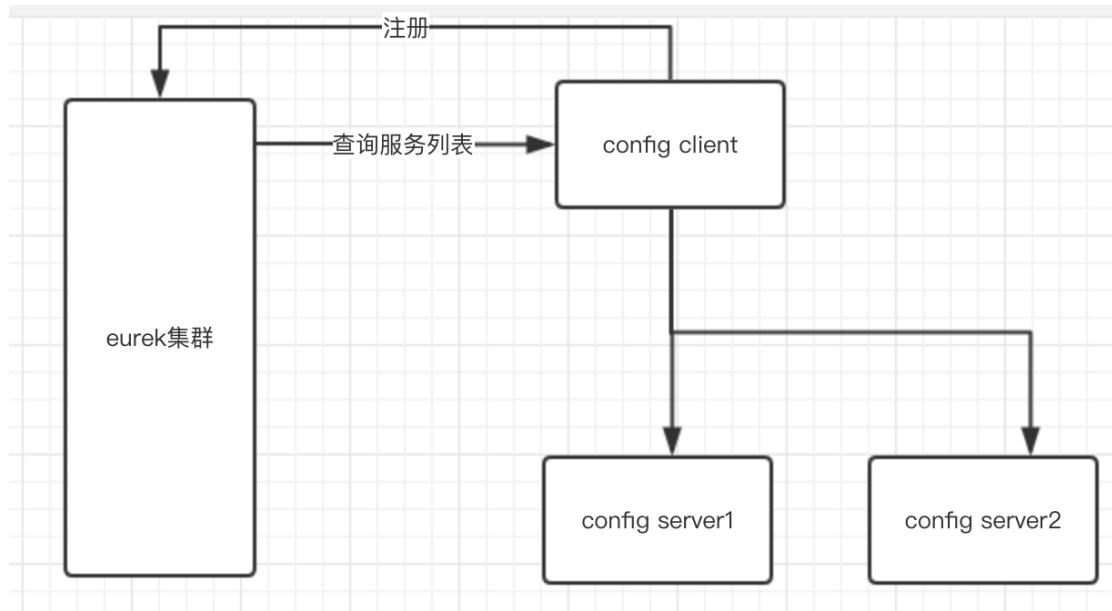
---

Rancher，官方文档：<https://www.cnrancher.com/rancher/>

高可用

在 pig 生产部署中着重考虑一下几个节点的高可用部署：

## 配置中心高可用



我们建议生产配置存储在 Git 仓库中，实现版本控制和高可用，所以实现 Config Server 的高可用，必须有一个高可用的 Git 仓库。

网络拓扑如下图类似。通过 spring cloud 的 ribbon 来实现对 config server 的负载均衡，保证高可用。

1. git 仓库可以使用多个配置中心 来实现高可用。例如： config-server1 加载码云的配置文件，而 config-server2 加载 github 的配置文件
2. 当然也可以使用 Git 来实现高可用，比如自建版本控制 GitLab，而可以实现高可用，那么此时 config-server1,config-server2 加载配置文件的地址一样，而是通过 GitLab 自己的 HA 策略来实现高可用，如果使用 GitLab 可以查看它的官方文档高可用章节来实现。

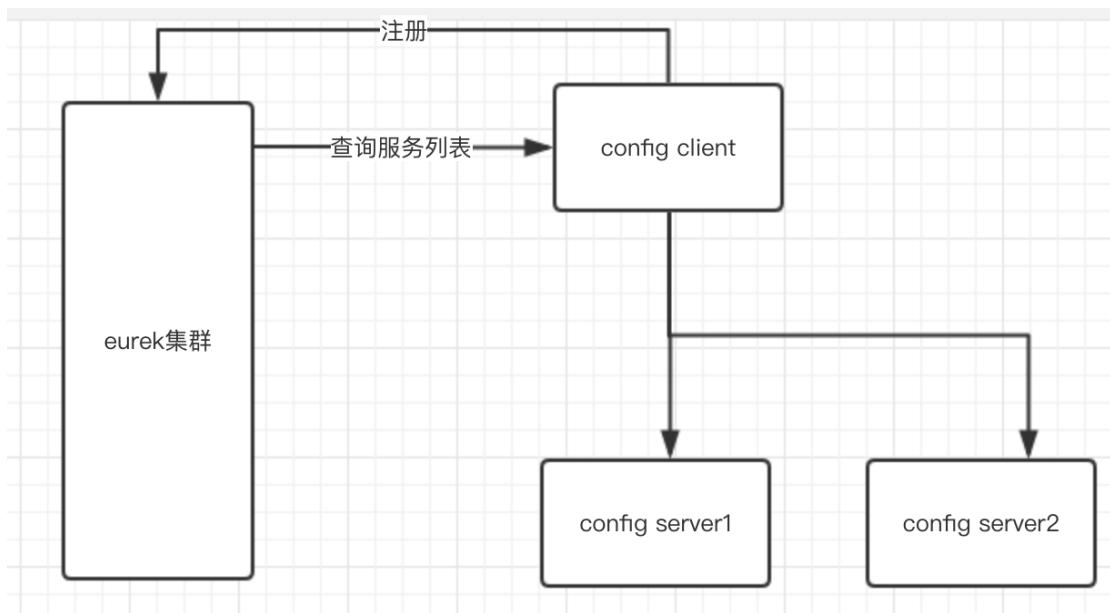
注册中心高可用

配置中心高可用

rabbitmq 高可用

配置中心高可用

配置中心高可用



我们建议生产配置存储在 Git 仓库中，实现版本控制和高可用，所以实现 Config

Server 的高可用，必须有一个高可用的 Git 仓库。

网络拓扑如下图类似。通过 spring cloud 的 ribbon 来实现对 config server 的负载均衡，保证高可用。

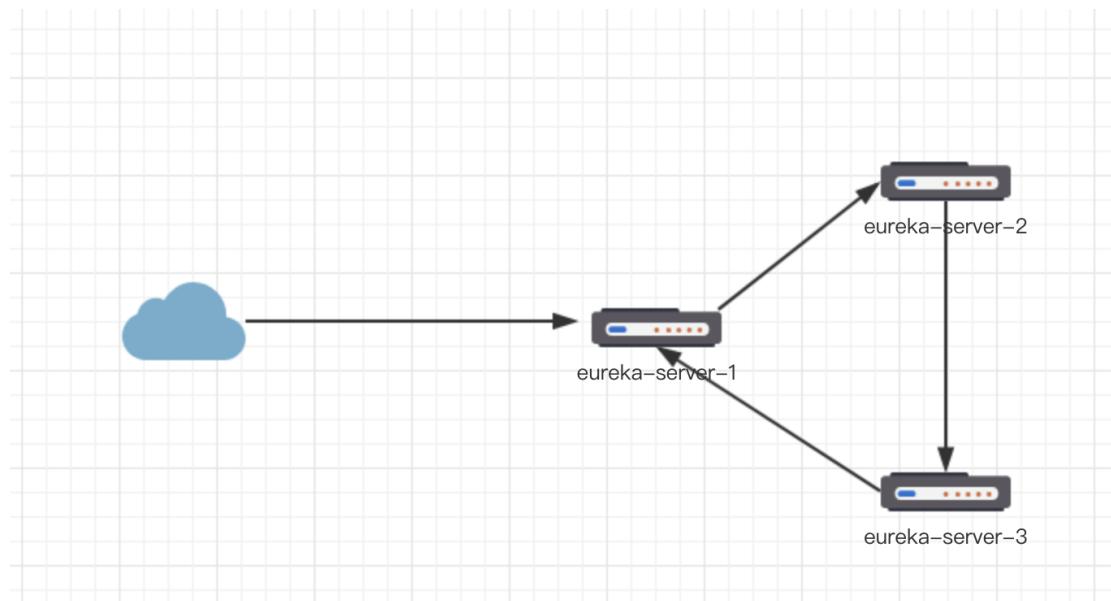
1. git 仓库可以使用多个配置中心 来实现高可用。例如： config-server1 加载码云的配置文件，而 config-server2 加载 github 的配置文件

2. 当然也可以使用 Git 来实现高可用，比如自建版本控制 GitLab，而可以实现高可用，那么此时 config-server1,config-server2 加载配置文件的地址一样，而是通

过 GitLab 自己的 HA 策略来实现高可用，如果使用 GitLab 可以查看它的官方文档高可用章节来实现。

## eureka 注册中心高可用

这里主要针对的 eureka 的高可用



- 添加主机名：

```
127.0.0.1 peer1 peer2
```

- 修改 application.yml

```
---
spring:
  profiles: peer1          # 指定 profile=peer
  1
server:
  port: 8761
eureka:
  instance:
    hostname: peer1        # 指定当 profile=pee
    r1 时， 主机名
    client:
      serviceUrl:
```

```
    defaultZone: http://peer2:8762/eureka/      # 将自己注册到 peer2 这个 Eureka 上面去

---
spring:
  profiles: peer2
server:
  port: 8762
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/
```

- 分别启动两个 Eureka 应用：

```
java -jar microservice-discovery-eureka-0.0.1-SNAPSHOT.jar --spring.profiles.active=peer1
java -jar microservice-discovery-eureka-0.0.1-SNAPSHOT.jar --spring.profiles.active=peer2
```

- 访问 <http://peer1:8761>，我们会发现 registered-replicas 中已经有 peer2 节点了，同样地，访问 <http://peer2:8762>，也能发现其中的 registered-replicas 有 peer1 节点，如下图：

DS Replicas			
<b>peer2</b>			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
UNKNOWN	n/a (2)	(2)	UP (2) - QH-20160301NAVT:8761 , QH-20160301NAVT:8762
General Info			
Name	Value		
total-avail-memory	450mb		
environment	test		
num-of-cpus	4		
current-memory-usage	124mb (27%)		
server-upptime	00:01		
<b>registered-replicas</b>	<a href="http://peer2:8762/eureka/">http://peer2:8762/eureka/</a>		
<b>unavailable-replicas</b>			
<b>available-replicas</b>	<a href="http://peer2:8762/eureka/">http://peer2:8762/eureka/</a> ,		
Instance Info			
Name	Value		
ipAddr	192.168.0.59		

我们尝试将 peer2 节点关闭，然后访问 <http://peer1:8761>，会发现此时 peer2 会被添加到 unavailable-replicas 一栏中。

## 将服务注册到高可用的 Eureka

如果注册中心是高可用的，那么各个微服务配置只需要将 defaultZone 改为如下即可：

```
eureka:
  client:
    serviceUrl:
      defaultZone:
        http://peer1:8761/eureka/,http://peer2:8762/eureka
```

consul 注册中心高可用

## Spring Cloud Config 原理

我们通过 git 把配置文件推送到远程仓库做版本控制，当版本发生变化的时候，远程仓库通过 webhook 机制推送消息给 Config Server，Config Server 将修改通知发送到消息总线，然后所有的 Config Client 进行配置刷新。

非常巧妙的借助了 Git 来做配置文件修改的版本控制。

## Consul Config 的 FILES 机制

```
public enum Format {
    /**
     * Indicates that the configuration specified in consul is
     * of type native key values.
     */
    KEY_VALUE,

    /**
     * Indicates that the configuration specified in consul is
     * of property style i.e.,
     *   * value of the consul key would be a list of key=value pairs
     * separated by new lines.
     */
    PROPERTIES,

    /**
     * Indicates that the configuration specified in consul is
     * of YAML style i.e., value
     *   * of the consul key would be YAML format
     */
    YAML,

    /**
     * Indicates that the configuration specified in consul uses keys as files.
     *   * This is useful for tools like git2consul.
     */
    FILES,
}
```

Consul 提供以上的策略，key/value、yaml、properties,可以很简单的通过 Consule Config 的管理台进行配置，我们主要来看 FILES，就是我们也是 Cloud Config 一样，通过 Git 来做版本控制，只是用 **Consul** 做配置的分发和修改的通知。

原生的 Consul 不支持 Git 来做，需要借助 Consul 社区提供的另外一个工具 [git2consul](#)

非常简单就下载就安装好了。

主要来讲一下初始化脚本的 git2consul.json

```
{  
    "version": "1.0",  
    "local_store": "本地仓库备份地址",  
    "logger": {  
        "name": "git2consul",  
        "streams": [  
            {  
                "level": "trace",  
                "type": "rotating-file",  
                "path": "生成日志路径/git2consul.log"  
            }  
        ]  
    },  
    "repos": [  
        {  
            "name": "pig-config",  
            "url": "远程仓库地址",  
            "include_branch_name": true, //分支信息是否包含到请求  
中，建议使用  
            "branches": [  
                "dev"  
            ],  
            "hooks": [  
                {  
                    "type": "polling", //更新策略定时刷新的  
                    "interval": "1" //1分钟  
                }  
            ]  
        }  
    ]  
}
```

```
        }
    ]
}
```

启动时候指定上边的脚本

```
./git2consul --config-file git2consul.json
```

## bootstarp.yml 配置

```
spring:
  application:
    name: pig-auth
  cloud:
    consul:
      host: localhost
      port: 8500
      config:
        enabled: true
        format: FILES
        watch:
          enabled: true
        prefix: ${spring.application}/${spring.profiles.active}
  profiles:
    active: dev
```

OK 已经可以使用了 git2consul 来同步你的配置文件啦。

## 配置细节

如上图，我配置文件的例子。

FILES 机制和 Spring Cloud Config 加载类似，application.yml 会被所有微服务模块加载公用，对应的 application-name.yml 会被对应的微服务加载。

# 生产调优参考资料

## JVM

<https://docs.oracle.com/javase/specs/jvms/se8/html/index.html>

Metaspace

<http://ifeve.com/jvm-troubleshooting-guide-4/>

压缩类空间

<https://blog.csdn.net/jijijjwwi111/article/details/51564271>

CodeCache

<https://blog.csdn.net/yandaonan/article/details/50844806>

<http://engineering.indeedblog.com/blog/2016/09/job-search-web-app-java-8-migration/>

GC 调优指南：

<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html>

如何选择垃圾收集器

<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html>

G1 最佳实践

[https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1\\_gc\\_tuning.html#recommendations](https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc_tuning.html#recommendations)

G1 GC 的一些关键技术

<https://zhuanlan.zhihu.com/p/22591838>

CMS 日志格式

<https://blogs.oracle.com/poonam/understanding-cms-gc-logs>

G1 日志格式

<https://blogs.oracle.com/poonam/understanding-g1-gc-logs>

GC 日志分析工具

<http://gceeasy.io/>

GCViewer

<https://github.com/chewiebug/GCViewer>

ZGC:

<http://openjdk.java.net/jeps/333>

jdk8 工具集

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/index.html>

Troubleshooting

<https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/>

jps

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jps.html>

jinfo

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jinfo.html>

jstat

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html>

jmap:

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jmap.html>

mat:

<http://www.eclipse.org/mat/downloads.php>

jstack:

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstack.html>

java 线程的状态

<https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr034.html>

java 线程状态转化:

<https://mp.weixin.qq.com/s/GsxeFM7QWuR--Kpb7At2w>

死循环导致 CPU 负载高

<https://blog.csdn.net/goldenfish1919/article/details/8755378>

正则表达式导致死循环:

<https://blog.csdn.net/goldenfish1919/article/details/49123787>

jvisualVM:

<https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/index.html>

<https://visualvm.github.io/documentation.html>

jvisualVM 如何添加插件

<https://visualvm.github.io/index.html>

btrace

<https://github.com/btraceio/btrace>

<https://github.com/btraceio/btrace/releases/tag/v1.3.11>

jdwp 协议

<https://www.ibm.com/developerworks/cn/java/j-lo-jpda3/>

psi-probe:

<https://github.com/psi-probe/psi-probe>

tomcat 优化相关参数:

tomcat-manager: \${tomcat}/webapps/docs/manager-howto.html

\${tomcat}/webapps/docs/config/http.html

\${tomcat}/webapps/docs/config/host.html

\${tomcat}/webapps/docs/config/context.html

\${tomcat}/webapps/docs/connectors.html

apr 连接器:

<http://apr.apache.org/>

nginx 调优

<http://nginx.org/en/docs/>

[http://nginx.org/en/linux\\_packages.html](http://nginx.org/en/linux_packages.html)

ngx\_http\_stub\_status:

[http://nginx.org/en/docs/http/ngx\\_http\\_stub\\_status\\_module.html](http://nginx.org/en/docs/http/ngx_http_stub_status_module.html)

ngxtop:

<https://github.com/lebinh/ngxtop>

nginx-rdd

<http://www.linuxde.net/2012/04/9537.html>

## JAVA

java 虚拟机规范

<https://docs.oracle.com/javase/specs/jvms/se8/html/index.html>

java 语言规范

<https://docs.oracle.com/javase/specs/jls/se8/html/index.html>

javap:

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/javap.html>

字段描述符

<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.3.2>

方法描述符

<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.3.3>

字节码指令:

<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html>

常量池:

<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.4>

本地变量表:

<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.6.1>

<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.13>

操作数栈:

<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.6.2>

Code 属性:

<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.3>

LineNumberTable:

<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.12>

constant variable:

<https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.12.4>

常量表达式

<https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.28>

String.intern

<https://blog.csdn.net/goldenfish1919/article/details/80410349>

String 去重

<https://blog.csdn.net/goldenfish1919/article/details/20233263>