1. 即时通信和接口加密
2. 即时通信的业务场景

即时通信（Instant Messaging, IM）是一个允许两人或多人使用网络实时进行文字，文件，语音，视频交流的技术，应用于实时收发消息的场景。如客服系统，语音视频社交软件。

1. 短连接和长连接
2. 短连接

短连接是指客户端和服务器每进行一次通信，就建立一次连接，当通信结束时就中断连接。如客户端首先向服务器发送请求建立连接，服务器返回同意请求后，客户端再次请求，服务器返回页面HTML代码。一次通信结束，服务器和客户端断开连接。通常使用的是HTTP协议，它运行在TCP协议之上，HTTP/1.0使用的TCP默认是短连接。

1. 长连接

长连接是指在建立了连接后可以一直保持着，客户端和服务器之间可以连续多次地发送数据，知道双方断开连接。HTTP从1.1版本起，TCP默认使用的是长连接，使用长连接的HTTP协议，会在响应头添加 Connection:keep-alive

1. 使用场景
2. 短连接

短连接适用于 并发量大，数据交互不频繁的场景。为何？并发量大，说明同时会有很多连接，短连接最适合这种并发大的情况，一个连接用完就释放，不会占用服务器太多资源。数据交互不频繁，说明连接的使用次数很少，那么在进行了一次连接传输数据后，可能就不再传输了，如果这里使用长连接，就会造成资源的浪费，因此最好使用短连接用完就释放。

1. 长连接

对于数据交互频繁的情况，保持长连接是最佳选择，免去了短连接不断创建连接，关闭连接的资源消耗。

对于点对点通信，需要使用长连接。当使用短连接时，连接断开后，服务器如果有数据的改动是无法通知到浏览器的，因为连接已经断开了，HTTP是一个无状态的协议。而长连接就不一样了，连接会持续保持着，服务器能够即时地通知浏览器数据的变动，这就使得点对点的通信称为可能。

1. websocket协议
2. websocket概述

Websocket是HTML5开始提供的一种在单个TCP连接上进行全双工通信的协议。全双工指的是双方在通信时允许在两个方向上同时传输，相当于两个单工通信方式的结合。那么，使用websocket，浏览器和服务器只需要完成一次握手，就可以创建持久性的连接，并进行双向数据传输。相比于Ajax的定时轮询机制能够节省服务器资源和带宽。websocket协议一般用于服务器方客户端发送数据。

1. websocket常用事件方法
2. 创建WebSocket对象

var Socket = new WebSocket(url, [protocol]);

1. 连接建立时触发

Socket.onopen()

1. 客户端接受服务端数据时触发

Socket.onmessage()

1. 通信发生错误时触发

Socket.onerror()

1. 连接关闭时触发

Socket.onclose()

1. 使用连接发送数据

Socket.send()

1. 关闭连接

Socket.close()

1. 环信云IM功能
2. 技术选型

环信im云

前端框架vue

1. 结构设计

前端页面使用十次方用户微服务认证用户身份，使用环信im云进行即时消息通信。

1. 环信im云介绍

环信im云是即时通讯云PaaS平台，开发者可以通过简单的SDK和REST API进行对接。

1. 环信云调试接口

<http://api-docs.easemob.com/#/%E8%8E%B7%E5%8F%96token>

该接口可以调试环信im，也可以使用POSTMAN调试。

1. 项目的即时通讯功能实现
2. 创建tensquare\_user子模块
3. 添加依赖

tensquare\_user子模块依然继承自tensquare\_parent父模块，同时要导入tensquare\_common模块。user模块主要操作mysql数据库，因此需要导入mybatis-plus相关依赖。

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <parent>  
 <artifactId>tensquare\_parent</artifactId>  
 <groupId>com.tensquare</groupId>  
 <version>1.0-SNAPSHOT</version>  
 </parent>  
 <modelVersion>4.0.0</modelVersion>  
  
 <artifactId>tensquare\_user</artifactId>  
  
 <dependencies>  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.tensquare</groupId>  
 <artifactId>tensquare\_common</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
 <!-- mybatis-pllus-begin-->  
 <dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatisplus-spring-boot-starter</artifactId>  
 <version>${mybatisplus-spring-boot-starter.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatis-plus</artifactId>  
 <version>${mybatisplus.version}</version>  
 </dependency>  
 <!-- mybatis-plus-end-->  
  
 </dependencies>  
  
</project>

1. 配置application.yml

修改端口号，添加用户数据库信息，配置mybatis。

spring:  
 application:  
 name: tensquare-article  
 datasource:  
 driver-class-name: com.mysql.jdbc.Driver  
 url: jdbc:mysql://192.168.112.128:3306/tensquare\_user?characterEncoding=utf-8  
 username: root  
 password: root  
  
server:  
 port: 9008  
  
*# Mybatis-Plus 配置*mybatis-plus:  
 *# mapper-locations: classpath:/mapper/\*Mapper.xml  
 #实体扫描，多个package用逗号或者分号分隔* typeAliasesPackage: com.tensquare.user.pojo  
 global-config:  
 id-type: 1 *#0:数据库ID自增 1:用户输入id* db-column-underline: false  
 refresh-mapper: true  
 configuration:  
 map-underscore-to-camel-case: true  
 cache-enabled: true *#配置的缓存的全局开关* lazyLoadingEnabled: true *#延时加载的开关* multipleResultSetsEnabled: true *#开启延时加载，否则按需加载属性* log-impl: org.apache.ibatis.logging.stdout.StdOutImpl *#打印sql语句,调试用*

1. 添加用户子模块的引导类

package com.tensquare.user;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class UserApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(UserApplication.class, args);  
 }  
}

1. 创建User的POJO

package com.tensquare.user.pojo;  
  
import com.baomidou.mybatisplus.annotations.TableId;  
import com.baomidou.mybatisplus.annotations.TableName;  
import com.baomidou.mybatisplus.enums.IdType;  
  
import java.io.Serializable;  
import java.util.Date;  
  
@TableName("tb\_user")  
public class User implements Serializable {  
  
 @TableId(type = IdType.*INPUT*)  
 private String id;  
  
 private String mobile;  
 private String password;  
 private String nickname;  
 private String sex;  
 private Date birthday;  
 private String avatar;  
 private String email;  
 private Date regdate;  
 private Date updatedate;  
 private Date lastdate;  
 private Long online;  
 private String interest;  
 private String personality;  
 private Integer fanscount;  
 private Integer followcount;  
  
 public String getId() {  
 return id;  
 }  
  
 public void setId(String id) {  
 this.id = id;  
 }  
  
 public String getMobile() {  
 return mobile;  
 }  
  
 public void setMobile(String mobile) {  
 this.mobile = mobile;  
 }  
  
 public String getPassword() {  
 return password;  
 }  
  
 public void setPassword(String password) {  
 this.password = password;  
 }  
  
 public String getNickname() {  
 return nickname;  
 }  
  
 public void setNickname(String nickname) {  
 this.nickname = nickname;  
 }  
  
 public String getSex() {  
 return sex;  
 }  
  
 public void setSex(String sex) {  
 this.sex = sex;  
 }  
  
 public Date getBirthday() {  
 return birthday;  
 }  
  
 public void setBirthday(Date birthday) {  
 this.birthday = birthday;  
 }  
  
 public String getAvatar() {  
 return avatar;  
 }  
  
 public void setAvatar(String avatar) {  
 this.avatar = avatar;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 public Date getRegdate() {  
 return regdate;  
 }  
  
 public void setRegdate(Date regdate) {  
 this.regdate = regdate;  
 }  
  
 public Date getUpdatedate() {  
 return updatedate;  
 }  
  
 public void setUpdatedate(Date updatedate) {  
 this.updatedate = updatedate;  
 }  
  
 public Date getLastdate() {  
 return lastdate;  
 }  
  
 public void setLastdate(Date lastdate) {  
 this.lastdate = lastdate;  
 }  
  
 public Long getOnline() {  
 return online;  
 }  
  
 public void setOnline(Long online) {  
 this.online = online;  
 }  
  
 public String getInterest() {  
 return interest;  
 }  
  
 public void setInterest(String interest) {  
 this.interest = interest;  
 }  
  
 public String getPersonality() {  
 return personality;  
 }  
  
 public void setPersonality(String personality) {  
 this.personality = personality;  
 }  
  
 public Integer getFanscount() {  
 return fanscount;  
 }  
  
 public void setFanscount(Integer fanscount) {  
 this.fanscount = fanscount;  
 }  
  
 public Integer getFollowcount() {  
 return followcount;  
 }  
  
 public void setFollowcount(Integer followcount) {  
 this.followcount = followcount;  
 }  
  
 @Override  
 public String toString() {  
 return "User{" +  
 "id='" + id + '\'' +  
 ", mobile='" + mobile + '\'' +  
 ", password='" + password + '\'' +  
 ", nickname='" + nickname + '\'' +  
 ", sex='" + sex + '\'' +  
 ", birthday=" + birthday +  
 ", avatar='" + avatar + '\'' +  
 ", email='" + email + '\'' +  
 ", regdate=" + regdate +  
 ", updatedate=" + updatedate +  
 ", lastdate=" + lastdate +  
 ", online=" + online +  
 ", interest='" + interest + '\'' +  
 ", personality='" + personality + '\'' +  
 ", fanscount=" + fanscount +  
 ", followcount=" + followcount +  
 '}';  
 }  
}

1. 创建Dao层接口UserDao

继承BaseMapper<User>

package com.tensquare.user.dao;  
  
import com.baomidou.mybatisplus.mapper.BaseMapper;  
import com.tensquare.user.pojo.User;  
  
public interface UserDao extends BaseMapper<User> {  
}

1. 创建Service层的UserService类的selectById方法

package com.tensquare.user.service;  
  
import com.tensquare.user.dao.UserDao;  
import com.tensquare.user.pojo.User;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
@Service  
public class UserService {  
  
 @Autowired  
 private UserDao userDao;  
  
 //根据id查询用户  
 public User selectById(String userId){  
 return userDao.selectById(userId);  
 }  
}

1. 创建Controller层的selectById方法

package com.tensquare.user.controller;  
  
import com.tensquare.user.pojo.User;  
import com.tensquare.user.service.UserService;  
import entity.Result;  
import entity.StatusCode;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
@RequestMapping("user")  
public class UserController {  
  
 @Autowired  
 private UserService userService;  
  
 //根据id查询用户  
 @RequestMapping(value="{userId}", method= RequestMethod.*GET*)  
 public Result selectById(@PathVariable String userId){  
 User user = userService.selectById(userId);  
 return new Result(true, StatusCode.*OK*, "查询成功", user);  
 }  
}

1. 创建mybatispuls的配置类

就是从引导类中将mybatis的配置抽取出来，独立配置成类。

package com.tensquare.user;  
  
import com.baomidou.mybatisplus.plugins.PaginationInterceptor;  
import javafx.scene.control.Pagination;  
import org.mybatis.spring.annotation.MapperScan;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@MapperScan("com.tensquare.user.dao")  
public class MyBatisPlusConfig {  
  
 @Bean  
 public PaginationInterceptor createPaginationInterceptor(){  
 return new PaginationInterceptor();  
 }  
}

1. 测试user微服务模块

访问controller中定义好的地址，localhost:9008/user/1，返回了id为1的用户信息。

1. 添加环信IM的SDK模块
2. 在tensquare\_user模块中添加环信IM的SimpleDemo
3. 在resources目录下添加static目录，用于存储静态文件，如js，html文件。将环信IM的SimpleDemo 中的静态文件复制过去，并且添加了依赖的SDK。
4. 测试环信IM的SimpleDemo

该Demo的功能是用户的登录与互相发送消息。当登录成功时，登录方发送了websocket到服务器中，建立长连接，发送消息时，可以从浏览器的开发者工具中查看效果

1. 改造聊天室页面

添加注册，登录功能，添加用户到环信IM上，可以即时通讯。

使用spring-websocket项目中的聊天室页面chatroom.jsp，将文件格式改为html。并对页面进行修改

由于是要参照环信IM的SimpleDemo的功能，因此参照SimpleDemo中的demo.html页面，删除jsp开头的声明，添加页面配置信息，添加SDK资源导入，删除所有的script，

添加注册按钮和登录按钮，修改自己的appkey，

在script中添加 回调函数，直接从demo.html中复制建立连接的js代码以及在收到文本时的回调函数。

添加注册，登录功能函数，在注册的同时还要将注册的userId注册到环信IM上，需要使用Ajax，因此需要导入jquery。要设置Ajax为同步的，因为首先要获取用户数据，才能够进行注册，如果是异步的，可能获取不到用户的数据。

通信未实现

1. 接口加密
2. 概述

项目是使用前后端分离进行开发的，前端与后端之间的通信都是使用HTTP协议，调用接口传输数据，这样就可以很容易地使用抓包工具来抓取传输的数据，如fiddler抓包工具，charles抓包工具。如视频网站播放视频前的广告，该请求地址都是通过加密后传输的，不然让人破解了真实视频的地址，就可以绕过广告，直接播放视频了。

1. 加密方式
2. 摘要算法

消息摘要是把任意长度的输入变为固定长度的输出，主要特点是加密过程不需要密匙，经过加密的数据无法被解密，只有输入相同的明文数据且经过相同的消息摘要算法才能够得到相同的密文。消息摘要算法不存在密匙的管理与分发问题，适合于分布式网络上使用。

如： CRC， MD5， SHA

1. 对称加密

对称加密通常在消息发送方需要加密大量数据时使用，因为它的加密速度快，通过一个密钥控制加密和解密的过程，这个密钥的管理就要重中之重，如何将密钥安全地传递到解密者的手中是关键，且有一组规则规定如何进行加密和解密。应用场景是，离线的大量数据加密，用于存储。如DES，AES等

1. 非对称加密

非对称加密算法使用两个不同的密钥，公开密钥（publickey）和私有密钥（privatekey），公钥和私钥是一对的，如果用公钥对数据进行加密，只有用对应的私钥才能够解密。

但是它的算法强度复杂，速度没有对称密钥块，经典的应用场景是数字签名（私钥加密。公钥验证）。如RSA，Elgamal等。

1. 数字签名

前面的加密算法能够解决数据的加密与解密问题，可是如果数据在传输过程中被人截获，并且修改了数据的内容呢？接收方如何辨别数据是否完整地来自于发送方，这就需要数字签名的帮助了。

数字签名通常使用私钥生成签名，使用公钥验证签名。过程如下：

1. 发送方用一个哈希函数（如MD5）从报文文本中生成报文摘要，然后用自己的私钥对这个摘要进行加密。
2. 将加密后的摘要作为报文的数字签名和报文一起发送给接收方。
3. 接收方使用与发送方一样的哈希函数从接收到的原始报文中计算出报文摘要。
4. 接收方再用发送方的公用密钥对报文附加的数字签名进行解密。
5. 如果这两个摘要相同，接收方就能确认该数字签名是发送方的。
6. 使用OpenSSL生成RSA密钥对
7. RSA算法的密钥格式

密钥就是一长串字符，长度介于512 – 65536之间，JDK中的默认长度是1024，但必须保证是64的倍数，密钥的文本格式有pem（文本存储）或者der（二进制存储）。

当使用Java API生成RSA密钥对时，公钥以X.509格式编码，私钥一PKCS#8格式编码。

1. 使用OpenSSL生成RSA私钥

格式如下， 命令中包含了openssl命令， 存储的格式，文件的位置，私钥的长度, out表示输出：

openssl genrsa -out ../mycerts/rsa\_private\_key.pem 2048

1. 根据私钥生成对应的公钥

上面生成的私钥为输入，公钥为输出：

openssl rsa -in ../mycerts/rsa\_private\_key.pem -pubout -out ../mycerts/rsa\_public\_key\_2048.pub

1. 私钥转换为pkcs8格式

私钥要存储的话需要转成pkcs8格式的。

openssl pkcs8 -topk8 -inform PEM -in ../mycerts/rsa\_private\_key.pem -outform PEM -nocrypt > ../mycerts/rsa\_private\_key\_pkcs8.pem

1. 搭建接口加密微服务
2. 接口加解密请求参数的流程

接口调用者，传输加密后的参数到前端的网关中，网关解密后，将解密后的参数传递到真正处理的微服务中。

1. 修改tensquare\_parent

添加spring-cloud依赖

<dependencyManagement>  
 <dependencies>  
 <!-- spring cloud-->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>Greenwich.SR1</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
</dependencyManagement>

1. 创建Eureka微服务

创建tensquare\_eureka子工程，

1. 添加Eureka依赖

<dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
 </dependency>  
</dependencies>

1. 添加application.yml配置文件

server:  
 port: 6868  
spring:  
 application:  
 name: tensquare\_eureka  
eureka:  
 client:  
 *#是否将自己注册到Eureak中* register-with-eureka: false  
 *#是否从Eureka中获取信息* fetch-registry: false  
 service-url:   
 defaultZone: http://127.0.0.1:/${server.port}/eureka/

1. 添加引导类

package com.tensquare.eureka;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;  
  
@SpringBootApplication  
//开启Eureka服务  
@EnableEurekaServer  
public class EurekaApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(EurekaApplication.class, args);  
 }  
}

1. 修改文章微服务

注册到Eureka微服务中。

1. 添加Eureka-client客户端依赖

<!-- Eureka-client-->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>

1. 添加Eureka client配置信息

eureka:  
 client:  
 service-url:   
 defaultZone: http://127.0.0.1:6868/eureka/  
 instance:  
 prefer-ip-address: true

prefer-in-address 表示将ip地址注册上去，而不是localhost

至此， 文章微服务就可以注册到Eureka注册中心了。

1. 创建tensquare\_encrypt网关服务子工程
2. 添加依赖

该服务需要注册到Eureka中，同时还需要网关依赖zuul。

<dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-zuul</artifactId>  
 </dependency>  
</dependencies>

1. 编写配置文件

首先，配置zuul网关的路由规则，由于这里要将请求路由到 文章微服务中，因此设置文章微服务在Eureka中的名称，以及路由转发规则。

然后，配置Eureka client的地址与规则。

server:  
 port: 9013  
spring:  
 application:  
 name: tensquare\_encrypt  
zuul:  
 routes:  
 tensquare\_article: *#文章* path: /article/\*\* *#配置请求URL的请求规则* serviceId: tensquare\_article *#指定Eureka注册中心的服务id* strip-prefix: true *#所有article前缀的请求都进行转发* sentiviteHaders:  
 customSensitiveHaders: true *#让zuul网关处理cookie和重定向*eureka:  
 client:  
 service-url:   
 defaultZone: http://127.0.0.1:6868/eureka/  
 instance:  
 prefer-ip-address: true

1. 添加引导类

开启EurekaClient, 开启zuul网关

package com.tensquare.encrypt;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;  
  
@SpringBootApplication  
@EnableEurekaClient  
//开启zuul网关  
@EnableZuulProxy  
public class EncryptApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(EncryptApplication.class, args);  
 }  
}

1. 添加接口加密，解密的工具类

使用准备好的加密，解密工具类

1. 测试网关

网关设置路由到文章微服务中，因此使用postman访问网关的ip就可以根据路由规则转发到对应的文章微服务的地址处。

如，文章微服务中的查询方法地址为：

<http://127.0.0.1:9004/article/search/1/1>

通过网关就可以访问网关地址，如下所示，访问到文章的查询服务：

<http://127.0.0.1:9013/article/article/search/1/1>

1. 测试加密类

数据首先要经过加密才能传输给网关，网关再经过解密传输原始数据给文章微服务。这里使用已经准备好的加密测试类EncryptTest，测试公钥加密情况。

1. 实现zuul网关过滤器

参数加密后需要经过网关的解密才能够发送给对应的微服务，因此，我们需要使用网关过滤器在过滤的过程中解密。

创建过滤器RSARequestFilter，该过滤器在Zuul网关下，需要继承Zuul网关的过滤器ZuulFilter。且这里要明确的一点是解密过程是在网关接受到发送过来的加密数据之后，但在网关转发数据之前发生的。

该过滤器继承了ZuulFilter，需要重写4个函数，filterType方法表示过滤器在什么环境下执行的，本次过滤器在转发之前执行，因此该重写方法如下所示：

@Override  
public String filterType() {  
 return FilterConstants.*PRE\_TYPE*;  
}

filterOrder()方法设置过滤器的执行顺序，

@Override  
public int filterOrder() {  
 //设置过滤器的执行顺序  
 return FilterConstants.*PRE\_DECORATION\_FILTER\_ORDER*+1;  
}

shouldFilter()方法表示是否使用过滤器：

@Override  
public boolean shouldFilter() {  
 //是否使用过滤器  
 return true;  
}

run方法中编写过滤器具体执行的逻辑：

@Override  
public Object run() throws ZuulException {  
 System.*out*.println("过滤器执行");  
 return null;  
}

通过网关测试后，过滤器同样执行了run方法中的操作。

1. 在run()方法中添加解密逻辑

首先要注入rsaService接口，其中包含着加密解密方法。然后，获取当前的requestContext容器，通过该容器可以获取当前的请求request和当前的响应response，：

//获取requestContext容器  
RequestContext ctx = RequestContext.*getCurrentContext*();  
//获取request和response  
HttpServletRequest request = ctx.getRequest();  
HttpServletResponse response = ctx.getResponse();

获取请求中的加密后的数据，这里我们使用了输入流的方式获取请求中的参数，然后使用Springframework中的StreamUtils工具类将流转换为了String。

//声明存放加密后的数据的变量  
String requestData = null;  
try {  
 //通过request获取inputStream  
 ServletInputStream inputStream = request.getInputStream();  
 //从inputStream中得到加密后的数据  
 requestData = StreamUtils.*copyToString*(inputStream, Charsets.*UTF\_8*);  
 System.*out*.println(requestData);  
} catch (Exception e) {  
 e.printStackTrace();  
}

对加密后的数据进行解密操作，同样是调用rsaService接口中的解密方法，可以在控制台中输出解密后的数据：

//对加密后的数据进行解密操作  
if(!Strings.*isNullOrEmpty*(requestData)){  
 decryptData = rsaService.RSADecryptDataPEM(requestData, RsaKeys.*getServerPrvKeyPkcs8*());  
 System.*out*.println(decryptData);  
}

1. 转发解密后的数据

上面在网关过滤器中实现了解密数据，得到了解密后的参数，但是这些参数并没有形成请求转发到Controller中，还需要我们构建一个新的请求。

同样是从requestContext容器中使用setRequest方法，创建一个新的request，使用setRequest方法，重写getInputStream方法, 这就表示着修改了request中的参数为传入的新参数（解密后的byte字节数组）；重写getContentLength()方法和getContentLengthLong方法，如果不重写，那么从request中获取到的还是原来的参数长度。

可以看出，在新建一个request包装器中，重写哪些方法与我们要修改request中哪些参数有很大的关系，要修改传入的参数，就要修改inputStream和参数的长度。

//转发解密后的数据到Controller中  
if(!Strings.*isNullOrEmpty*(decryptData)){  
 //获取到解密后的数据的字节数组  
 byte[] bytes = decryptData.getBytes();  
 //使用RequestContext进行数据的转发  
 ctx.setRequest(new HttpServletRequestWrapper(request){  
 @Override  
 public ServletInputStream getInputStream() throws IOException {  
 return new ServletInputStreamWrapper(bytes);  
 }  
  
 @Override  
 public int getContentLength() {  
 return bytes.length;  
 }  
  
 @Override  
 public long getContentLengthLong() {  
 return bytes.length;  
 }  
 });  
}

设置请求头中Content-Type为JSON格式，还有编码格式为UTF-8，以防止参数中的格式错误以及编码错误：

ctx.addZuulRequestHeader("Content-Type",  
 MediaType.*APPLICATION\_JSON\_VALUE*+";charset=UTF-8");

1. 测试传递加密后的数据能够查询成功

在POSTMAN 中传递加密后的数据，能够解密为参数，并且传递给Controller，最终查询到相应的数据，至此，加密，解密任务完成，且能够通过网关访问到文章微服务。用户从前端请求中传入的参数会经过加密，传递到网关，网关解密后传递到Controller中调用文章微服务。