

# Asym+Symm+Md 安全网络传输方案使用指南

---

- Asym 是 Asymmetric 非对称加密的简写，默认实现是 RSA
- Symm 是 Symmetric 对称加密的简写，默认实现是 AES
- Md 是 Message Digest 消息摘要的简写，默认实现是 SHA256
- Slf 是 self 自身的简写，对应的就是服务器或者客户端自身
- Oth 是 other 他人的简写，对应的就是相对的对方
- 为了方便介绍，介绍中全部使用默认实现进行描述
- 当然，这些实现，都可以方便的替换

## 预置方案

---

- Asym: RSA/SM2
  - Symm: AES/SM4
  - Md: SHA256/SM3
  - 可以任意搭配
  - 保证得到：Asym + Symm + Md 的完整组合即可
  - 通过修改 SecureProvider 进行替换配置
    - 如果不需要自定义实现
    - 可以使用预置实现直接替换即可
    - 预置实现类 SecureProviderPresets
  - 需要注意的是
  - 替换之后，服务端和前端需要清空历史生成的密钥对
  - 后端对应清空cache实现
  - 前端对应清空sessionStorage和localStorage
- 

## 简介

---

- 方案说明
  - 采用RSA+AES+SHA组合形式完成前后端交互的加解密过程
  - 同时进行nonce防重放攻击防御
  - 同时支持动态刷新RSA密钥
- 优势
  - 采用Filter+Aop+Forward实现无侵入式接入
  - 对于程序员来说是透明的
  - 支持请求体（body）/请求参数（queryString）的加密传输
  - 支持响应体（body）的加密传输
  - 支持请求URL的加密传输
  - 实现请求过程的全参数加密
- 缺点

- 可能某些特殊接口会发生错误
- 可以使用白名单或者注解进行排除处理
- 前端只提供了基于axios拦截器的过滤器实现
- 因为这能够实现程序员无感化
- 其他请求方式，因为不支持拦截器或者无感化
- 总体流程
  - 客户端
    - 登录后获取服务器RSA公钥
    - 获取客户端自己的RSA私钥
      - 如果客户端能够生成RSA密钥对
        - 生成自己的密钥对
        - 则直接和服务器进行交换公钥即可
      - 如果客户端不能够生成RSA密钥对
        - 则可以从服务端生成一个私钥返回（不推荐）
        - 同时公钥将会保留在服务端进行后续的数据交流
  - 服务端
    - 项目启动后生成RSA公钥私钥
    - 公钥发送给连接初始化的客户端
    - 私钥自己保存
    - 客户端的密钥处理
      - 客户端能够生成RSA密钥对
        - 客户端生成自己的RSA密钥对
        - 将自己的公钥发送密钥交换请求
        - 得到服务端的公钥
      - 客户端不能够生成RSA密钥对
        - 客户端请求自己的私钥时，生成随机的客户端密钥对
        - 返回客户端私钥，保留客户端公钥
- 发送数据
  - 客户端
    - 随机生成一个nonce
    - 随机生成一个AES密钥
    - 使用客户端的RSA公钥对AES密钥加密，放入请求头sswh
    - 使用AES密钥对请求体进行加密（也可以对其他部分加密，比如URL参数等）
    - 发送请求
  - 服务端
    - 随机生成一个AES密钥
    - 使用服务端的RSA私钥对AES密钥加密，放入响应头sswh
    - 使用AES密钥对响应体进行加密
    - 如果发现客户端的RSA密钥签名和服务端最新的RSA签名不一致
    - 则表示客户端的RSA密钥应该更新，这时同时返回响应头skey存放最新的RSA公钥
    - 结束响应
- 接受数据
  - 客户端
    - 检查响应头是否包含新的RSA公钥skey
    - 如果存在，则保存新的公钥
    - 从响应头中获取响应头sswh

- 将sswh内容使用客户端RSA公钥解密得到随机的AES密钥
- 使用得到的AES密钥解密响应体得到JSON串
- 对JSON串解析得到JSON对象
- 使用JSON对象即可
- 服务端
  - 从请求头中获取请求头sswh
  - 将sswh内容使用服务端RSA私钥解密得到随机的AES密钥
  - 使用得到的AES密钥解密请求体得到解密内容
  - 将解密内容重新包装为请求交给spring处理，自动完成请求参数注入
  - 接口中直接使用即可
  - 特别的，如果这个接口的参数不再请求体中
  - 则使用@SecureParams注解作用在对应的参数上，AOP完成解密直接使用即可
  - 由于客户端的密钥对绑定问题
  - 客户端还必须携带sswcas的客户端私钥签名
  - 这样服务器才能确定客户端使用的密钥对
  - 才能正确的进行解密
- 注意
  - 请求和响应中，不包含sswh则认为是不加密的
  - 如果实际数据时加密的，那将会失败，无法使用数据
  - 对于后端而言，定义了@SecureParams的接口，是一定需要加密的
  - 如果没有sswh,那么将会认为是非法的请求
  - 对于后端没有定义必须安全的接口
  - 收到带有sswh的请求之后，会进行解密，也就是说，这种情况下时可选的

## 伪代码流程

- 服务器初始化过程

```
1. let serverKeyPair=生成服务器的RSA密钥对
```

- 客户端初始化过程

```
1. let serverPublicKey=从服务器获取服务器的公钥serverPublicKey
2. let clientPrivateKey=从服务器生成客户端密钥对，并返回客户端私钥clientPrivateKey，服务器
   客户端公钥，这里实现客户端与客户端密钥对的绑定
```

- 发送过程

```
1. let body=消息正文
2. let aesKey=随机产生16字节的随机值
3. let nonce=使用UUID生成随机值
4. // AES加密消息体
5. let encText=AES.encrypt(body,aesKey)
6. // RSA加密aes密钥
7. let encAesKey=RSA.publicKeyEncrypt(aesKey,serverPublicKey)
```

```

8. // 计算消息摘要
9. let sign=SHA256.make(encText+encAesKey+nonce)
10. // RSA计算数字签名
11. let digital=RSA.privateKeyEncrypt(sign,clientPrivateKey)
12. // 发送请求
13. send(encText,encAesKey,nonce,sign,digital)

```

- 接受过程

```

1. // RSA解密数字签名
2. let digSign=RSA.publicKeyDecrypt(sign,clientPublicKey)
3. // 验证数字签名
4. if(digSign != sign){
5.     数字签名验证失败
6. }
7. // 计算消息摘要
8. let reqSign=SHA256.make(encText+encAesKey+nonce)
9. // 验证消息摘要
10. if(reqSign != sign){
11.     消息摘要验证失败
12. }
13. // 验证是否重放
14. if(exists(nonce)){
15.     重放请求验证失败
16. }
17. // 解密aes密钥
18. let aesKey=RSA.privateKeyDecrypt(encAesKey,serverPrivateKey)
19. // 解密消息体
20. let body=AES.decrypt(encText,aesKey)

```

## 使用示例

- 服务端
- 直接是请求体中的，则只需要请求头中存在sswh即可
- 另外这里在方法上加了@SecureParams注解，其中in/out默认为true
- 则代表对返回值加密响应给前端，同时前端发送过来的也需要加密

```

1. @SecureParams
2. @PostMapping("safe")
3. public Object safe(@RequestBody UserDto user){
4.     return user;
5. }

```

- 这是另一种，加密参数在URL中的形式
- 因为这里的password在URL参数中，因此无法被正常的请求体解密处理
- 因此在参数上添加@SecureParams注解，其中in默认为true
- 则会自动进行解密
- 方法上也有该注解，上面已经说了，不再重复

```
1. @SecureParams
2. @PostMapping("param")
3. public Object param(@SecureParams String password){
4.     System.out.println("password:"+password);
5.     return password;
6. }
```

## 如何获取与存储RSA公钥

- 服务端提供一个接口提供给客户端调用
- 接口返回内容从 SecureTransfer.getWebAsymPublicKey() 获取
  - 当使用密钥交换时，使用 secureTransfer.getWebAsymPublicKeyAndSwap(request,clientKey) 交换
- 可以如下定义：
- 也可以通过配置i2f.springboot.config.secure.api.enable=true直接启用
  - 内置的SecureController提供接口secure/key
  - 交换密钥时，则使用接口 secure/swapKey

```
1. @ConditionalOnExpression("${i2f.springboot.config.secure.api.enable:true}")
2. @RestController
3. @RequestMapping("secure")
4. public class SecureController {
5.
6.     @Autowired
7.     private SecureTransfer secureTransfer;
8.
9.     @Autowired
10.    private SecureConfig secureConfig;
11.
12.    @SecureParams(in = false, out = false)
13.    @PostMapping("key")
14.    public String key() {
15.        String pubKey = secureTransfer.getWebAsymPublicKey();
16.        return pubKey;
17.    }
18.
19.    @SecureParams(in = false, out = false)
20.    @PostMapping("clientKey")
21.    public String clientKey(HttpServletRequest request) {
22.        if(secureConfig.isEnableSwapAsymKey()){
```

```

23.         throw new SecureException(SecureErrorCode.BAD_SECURE_REQUEST, "服务端不允许请求秘钥策略");
24.     }
25.     String priKey = secureTransfer.getWebClientAsymPrivateKey(request);
26.     return priKey;
27. }
28.
29. @SecureParams(in = false, out = false)
30. @PostMapping("swapKey")
31. public String swapKey(HttpServletRequest request, @RequestBody String clientKey
    throws Exception {
32.     String pubKey = secureTransfer.getWebAsymPublicKeyAndSwap(request, clientKey
33.     return pubKey;
34. }
35. }

```

- 客户端收到之后进行保存
- 默认是存储在session中，如有其他需要，请修改secure-transfer.js

```

1. this.$axios({
2.     url: 'secure/key',
3.     method: 'post'
4. }).then(({data}) => {
5.     SecureTransfer.saveAsymOthPubKey(data)
6. })
7.
8. this.$axios({
9.     url: 'secure/clientKey',
10.    method: 'post'
11. }).then(({data}) => {
12.     SecureTransfer.saveAsymSlfPriKey(data)
13. })

```

- 交换秘钥的方式

```shell script

```

this.$axios({
url: 'secure/swapKey' ,
method: 'post' ,
data: {
key: SecureTransfer.loadWebAsymSlfPubKey()
}
}).then(({data}) => {
console.log( 'SECURE_KEY' , data)
SecureTransfer.saveAsymOthPubKey(data)
})

```

- 1.
2. - 此获取RSA公钥的代码
3. - 如果是使用Vue等虚拟DOM主体时
4. - 建议在Vue等主体的初始化时进行调用
5. - 下面以Vue为例
6.     - 在Vue主体实例创建时调用获取RSA公钥
7.     - 如果后端配置了动态刷新RSA，则建议使用定时器进行定时刷新
8.     - 否则可能出现请求失败，后端无法解密情况
9.     - 同时，为 SecureCallback 绑定对应的回调函数
10.    - 这样在请求响应错误时，能够自动进行对应的秘钥交换或者秘钥更新
11.    - 避免刷新页面来刷新秘钥
- 12.
13. ```bash
14. App.vue

```

1. import SecureTransfer from "@/secure/core/SecureTransfer";
2. import SecureCallback from '@/secure/core/SecureCallback'
3. import SecureConfig from "@/secure/SecureConfig";
4.
5. export default {
6.   name: 'App',
7.   components: {
8.
9.   },
10.  created() {
11.    if (SecureConfig.enableSwapAsymKey) {
12.      this.swapAsymKey()
13.      SecureCallback.callSwapKey = this.swapAsymKey
14.      let _this = this
15.      window.rsaTimer = setInterval(function () {
16.        _this.swapAsymKey()
17.      }, 5 * 60 * 1000)
18.    } else {
19.      this.initAsymOthPubKey()
20.      this.initAsymSlfPriKey()
21.      SecureCallback.callPubKey = this.initAsymOthPubKey
22.      SecureCallback.callPriKey = this.initAsymSlfPriKey
23.      let _this = this
24.      window.rsaTimer = setInterval(function () {
25.        _this.initAsymPubKey()
26.      }, 5 * 60 * 1000)
27.    }
28.  },
29.  destroyed() {
30.    clearInterval(window.rsaTimer)

```

```
31. },
32. methods: {
33.     swapAsymKey() {
34.         this.$axios({
35.             url: 'secure/swapKey',
36.             method: 'post',
37.             data: {
38.                 key: SecureTransfer.loadWebAsymSlfPubKey()
39.             }
40.         }).then(({data}) => {
41.             console.log('SECURE_KEY', data)
42.             SecureTransfer.saveAsymOthPubKey(data)
43.         })
44.     },
45.     initAsymOthPubKey() {
46.         this.$axios({
47.             url: 'secure/key',
48.             method: 'post'
49.         }).then(({data}) => {
50.             console.log('SECURE_KEY', data)
51.             SecureTransfer.saveAsymOthPubKey(data)
52.         })
53.     },
54.     initAsymSlfPriKey() {
55.         this.$axios({
56.             url: 'secure/clientKey',
57.             method: 'post'
58.         }).then(({data}) => {
59.             console.log('SECURE_KEY', data)
60.             SecureTransfer.saveAsymSlfPriKey(data)
61.         })
62.     }
63. }
64. }
```

---

## 如何使用

### 服务端（springboot环境）

#### 安装

- maven添加依赖

---

1. <!-- 加密算法的BC实现，没有出口政策限制，可以使用更强的加密强度 -->



```

2. <dependency>
3.   <groupId>org.bouncycastle</groupId>
4.   <artifactId>bcprov-jdk15on</artifactId>
5.   <version>1.64</version>
6. </dependency>

```

- 如果你需要替换其中的算法为国密SM系列算法
- 还需要引入此依赖

```

1. <!-- 增加了对国密SM系列算法的支持 -->
2. <dependency>
3.   <groupId>org.bouncycastle</groupId>
4.   <artifactId>bcpkix-jdk15on</artifactId>
5.   <version>1.64</version>
6. </dependency>

```

- 如果使用预置的国密 sm-crypto 实现
- 则引入此包

```

1. <dependency>
2.   <groupId>com.antherd</groupId>
3.   <artifactId>sm-crypto</artifactId>
4.   <version>0.3.2</version>
5. </dependency>

```

- 引入本包secure
- 如果本包在项目的扫描路径下，则不需要配置
- 如果不再扫描路径下，则在启动类上添加注解 `@EnableSecureConfig` 注解，以自动引入此功能
- 剩下就是使用了，在上面的示例中已经演示了，如何使用

## 使用

- 查看上面的使用示例

## 客户端（vue环境）

### 安装

- 引入本包secure
- 添加package.json依赖
- 当然你也可以单独npm install这些依赖，这里使用另一种方式
- 先添加前三个依赖到对应的dependencies节点中，直接复制进去即可
- 这里保留了vue的两个依赖，方便做参考
- 其中 sm-crypto 是国密算法的预置实现

```

1. "dependencies": {

```

```
2.   "vue": "^2.6.14",
3.   "vue-router": "^3.0.1",
4.   "js-base64": "^3.6.1",
5.   "crypto-js": "^4.1.1",
6.   "jsrsasign": "^10.8.6",
7.   "sm-crypto": "^0.3.12"
8. }
```

- 保存package.json之后，进入自己的项目路径
- 进行npm install,这就会自动把新加的依赖进行下载

```
1. npm install
```

- 【注意】，你可能知道jsencrypt有现成的npm依赖可以用
- 但是不要那么做，npm中的jsencrypt不能使用，这是别人从jsencrypt分支出来的一个修复版本
- 所以，不要替换成npm依赖，否则将不会正常工作
- 下面是文件夹结构

```
1. - web-root
2.   - src
3.     - secure
4.       - secure-vue-main.js
5.       - SecureConfig.js
6.       - secure-axios.js
7.       - server.js
8.       - ...
9.     - App.vue
10.    - main.js
```

- 在main.js中引入本包

```
1. import './secure/secure-vue-main'
```

- web端是基于过滤器实现的自动加解密
- 因此，需要对请求响应拦截器进行配置
- 以axios中使用请求响应拦截器为例
- 简单的封装，可以以此文件作为参考

```
1. ./secure/secure-axios.js
```

- 如果你使用默认的axios
- 则在main.js中引入

```
1. import './secure/secure-axios'
```

- 然后根据自己项目修改一下两个文件内容

1. `./secure/server.js`
2. `./secure/secure-axios.js`

- 下面介绍，自己封装的过程
- 在axios包装中，引入过滤器（当然还有必不可少的axios）
- 引入axios

1. `import axios from 'axios'`

- 引入过滤器

1. `import SecureTransferFilter from "../secure/core/secure-transfer-filter";`

- 添加一个请求实例

1. `const request = axios.create({`
2. `// axios 中请求配置有baseUrl选项，表示请求URL公共部分`
3. `baseUrl: 'http://localhost:9090',`
4. `// 超时`
5. `timeout: 60000`
6. `})`

- 为这个实例，添加请求拦截器

1. `// request拦截器`
2. `request.interceptors.request.use(config => {`
3.
4. `console.log('headers:', config.headers);`
5.
6. `// 核心过滤器`
7. `SecureTransferFilter.requestFilter(config)`
8.
9. `console.log('reqUrl:', config.url);`
10.
11. `return config`
12. `}, error => {`
13. `console.log(error)`
14. `Promise.reject(error)`
15. `})`

- 如果想要针对全局的请求都进行加密处理
- 则可以在拦截器中配置

- 这样配置之后，在 SecureConfig 中通过白名单配置的方式去除白名单即可

```
1. // 定义请求拦截
2. BaseRequest.interceptors.request.use(config => {
3.
4.   SecureTransfer.getSecureHeaderInto(config.headers, true, true)
5.   SecureTransferFilter.requestFilter(config)
6.
7.   return config
8. })
```

- 添加响应拦截器

```
1. // 响应拦截器
2. request.interceptors.response.use(res => {
3.   console.log('res:', res);
4.
5.   // 核心过滤器
6.   SecureTransferFilter.responseFilter(res);
7.
8.   // 未设置状态码则默认成功状态
9.   let code = res.data.code ;
10.  if(code==undefined || code==null){
11.    code=200;
12.  }
13.  // 获取错误信息
14.  const msg = res.data.msg
15.  if (code !== 200) {
16.    console.warn(msg);
17.    return Promise.reject(new Error(msg))
18.  } else {
19.    return res
20.  }
21. },
22. error => {
23.   SecureTransferFilter.responseFilter(error.response)
24.   console.log('err' , error)
25.   return Promise.reject(error)
26. }
27. )
```

- 下面为了方便使用，将其绑定到Vue原型上

```
1. import Vue from 'vue'
2.
```

```
3. Vue.prototype.$axios=request;
```

## 注意事项

- 关于 secure/static/jsencrypt.js
- 如果直接引入编译报错，也就是webpack方式引入报错
- 请注释 secure/util/rsa.js 中关于这个依赖的引入
- 改为直接在html中通过script方式引入
- 如下

```
1. secure/util/rsa.js
```

```
1. /**
2.  * RSA工具
3.  */
4. // 注释掉webpack引入方式
5. // import JSEncrypt from '../static/jsencrypt'
6.
7. const RsaUtil = {
8.     ...
```

```
1. index.html
```

```
1. <html>
2. <head>
3.     ...
4.     <!-- 通过静态引入方式引入，注意这个路径，放到自己的静态资源目录中对应引入 -->
5.     <script src="./jsencrypt.js"></script>
6.     ...
7. </head>
8. </html>
```

- 下面开始使用

## 使用

- 使用post请求
- 主要的就是添加一个secure的请求头
- 过滤器，将会检测这个请求头，如果包含这个请求头，将会进行自动的data加密
- 通过这个方法，进行给headers附加加密标记

```
1. // 使用场景，需要获取纯粹的secure请求标记头或者直接只有设置标记头时
2. // 可能是大多数情况下使用的
3. // 方法参数：是否开始URL参数加密，是否开启编码URL转发
```

```

4. // 返回值：一个headers对象
5. secureTransfer.getSecureHeader(openSecureParams,openSecureUrl)
6. // 使用场景，已经有了一些headers值，需要添加加密标记时
7. // 可能少部分场景使用
8. // 方法参数：已有的headers对象，是否开始URL参数加密，是否开启编码URL转发
9. // 返回值，入参的headers对象
10. secureTransfer.getSecureHeaderInto(headers,openSecureParams,openSecureUrl)

```

```

1. this.$axios({
2.   url: 'test/safe',
3.   method: 'POST',
4.   data:{
5.     userId:'1001',
6.     userName: '张',
7.     tel: '13122223333',
8.     password: 'pass'
9.   },
10.   headers:this.$secureTransfer.getSecureHeader(false,false)
11. }).then(({data})=>{
12.   this.form.output=data;
13. })

```

- 使用URL参数params

```

1. this.$axios({
2.   url:'test/param',
3.   method:'POST',
4.   params:{
5.     password: this.form.input
6.   },
7.   headers:this.$secureTransfer.getSecureHeader(true,false)
8. }).then(({data})=>{
9.   this.form.output=data
10. })

```

- 使用编码后的URL转发

```

1. this.$axios({
2.   url:'test/enc',
3.   method:'POST',
4.   params:{
5.     password: this.form.input
6.   },
7.   headers:this.$secureTransfer.getSecureHeader(false,true)

```

```
8.   }).then(({data})=>{
9.     this.form.output=data
10.  })
```

- 全功能开启

```
1.  this.$axios({
2.    url:'test/all',
3.    method:'POST',
4.    params:{
5.      password: this.form.input
6.    },
7.    headers:this.$secureTransfer.getSecureHeaderInto({
8.      token: sessionStorage.getItem('token')
9.    },true,true)
10.  }).then(({data})=>{
11.    this.form.output=data
12.  })
```

## 后端配置详解

```
1. # secure 配置
2. i2f:
3.   springboot:
4.     config:
5.       secure:
6.         # 是否开启
7.         enable: true
8.         # asym密钥的存储路径, 默认../
9.         asym-store-path: ../
10.        # 响应字符集, 默认UTF-8
11.        responseCharset: 'UTF-8'
12.        # Asym密钥长度, 默认1024, 可选1024,2048
13.        asymKeySize: 1024
14.        # Symm密钥长度, 默认128, 可选128,192,256
15.        symmKeySize: 128
16.        # 随机密钥生成的随机数的最大值, 默认8192
17.        randomKeyBound: 8192
18.        # 一次性消息的保持时间秒数, 默认6*60
19.        # 这段时间内重复出现的nonce将会被认为是重放请求被拦截
20.        nonceTimeoutSeconds: 360
21.        # 是否启动动态Asym更新密钥, 默认true
22.        enableDynamicAsymKey: true
23.        # 每次更新密钥的时长秒数, 默认6*60
```

```
24.    dynamicRefreshDelaySeconds: 360
25.    # 最多保留多少历史密钥，默认5
26.    dynamicMaxHistoriesCount: 5
27.    # 客户端密钥对的获取策略，是否是本地生成交换策略，默认true
28.    enableSwapAsymKey: true
29.    # 用于存储安全头的请求头名称，默认sswh
30.    headerName: sswh
31.    # 安全头格式的分隔符，默认;
32.    headerSeparator: ;
33.    # 动态刷新Asym密钥的响应头，默认skey
34.    dynamicKeyHeaderName: skey
35.    # URL加密的后端forward路径
36.    encUrlPath: /enc/
37.    # 请求URL参数加密的加密参数名
38.    parameterName: sswp
39.    # 默认的安全控制策略，也就是当注解和白名单都未配置时的策略模式，默认关闭
40.    defaultControl:
41.        # 进站是否安全
42.        in: true
43.        # 出站是否安全
44.        out: true
45.    # 白名单配置列表项，符合ant-match模式
46.    whitelist:
47.        # 进出站都忽略的列表清单
48.        bothPattens:
49.            - /file/**
50.            - /secure/key
51.        # 进站忽略的列表清单
52.        inPattens:
53.            - /common/upload/**
54.        # 出站忽略的列表清单
55.        outPattens:
56.            - /common/download/**
57.    # AOP功能
58.    aop:
59.        # 是否启用AOP功能，默认true
60.        # 改功能包含抛出核心filter的异常，使得能够通过ExceptionHandler进行捕获异常
61.        # 包含支持解密String类型的RequestParam请求参数
62.        # 包含controller为String类型返回值时的特殊处理
63.        # 因此不建议关闭此功能，关闭之后也需要自己进行覆盖实现
64.        enable: true
65.    # 内置的API接口
66.    api:
67.        # 是否开启默认的API响应Asym密钥获取请求，默认true
68.        # 请求路径 : /secure/key
69.        enable: true
```



```
70.     # 内置的URL请求路径转发接口
71.     enc-url-forward:
72.         # 是否开启enc的url解密请求转发，默认true
73.         # 请求路径: /enc/**
74.         enable: true
75.     # MVC替换converter为spring注册converter实现自定义
76.     # 当出现如果自定义的converter不生效时，需要开启
77.     # 当Long类型需要转换为string类型给前端时，必须开启
78.     mvc:
79.         # 是否开启自定义替换converter
80.         enable: true
81.     # 针对jackson的拓展自定义配置
82.     jackson:
83.         # 是否开启自定义配置
84.         enable: true
85.         # 是否开启Long类型转string类型给前端
86.         enableLongToString: true
87.         # 注意，LocalDateTime的格式化模式和spring.jackson.date-format配置一致
88.         # 因此，不用特殊配置
89.         # 定义LocalDate的格式化模式
90.         LocalDateFormat: yyyy-MM-dd
91.         # 定义LocalTime的格式化模式
92.         LocalTimeFormat: HH:mm:ss
```

## 前端配置详解

```
1. /**
2.  * 主配置
3.  */
4. import SecureConsts from './consts/secure-consts'
5.
6. const SecureConfig = {
7.     // Asymm密钥长度，默认1024，可选1024, 2048
8.     asymKeySize: SecureConsts.RSA_KEY_SIZE_1024(),
9.     // Symm密钥长度，默认128，可选128, 192, 256
10.    symmKeySize: SecureConsts.AES_KEY_SIZE_128(),
11.    // 用于存储安全头的请求头名称，默认sswh
12.    headerName: SecureConsts.DEFAULT_SECURE_HEADER_NAME(),
13.    // 动态刷新Asymm密钥的响应头，默认skey
14.    dynamicKeyHeaderName: SecureConsts.SECURE_DYNAMIC_KEY_HEADER(),
15.    clientKeyHeaderName: SecureConsts.SECURE_CLIENT_KEY_HEADER(),
16.    clientAsymSignName: SecureConsts.DEFAULT_SECURE_CLIENT_ASYNC_SIGN_NAME(),
17.    // 安全头格式的分隔符，默认;
18.    headerSeparator: SecureConsts.DEFAULT_HEADER_SEPARATOR(),
19.    // 指定在使用编码URL转发时的转发路径
```

```
20.   encUrlPath: SecureConsts.ENC_URL_PATH(),
21.   // 安全URL参数的参数名称
22.   parameterName: SecureConsts.DEFAULT_SECURE_PARAMETER_NAME(),
23.   // 客户端密钥对的获取策略, 是否是本地生成交换策略
24.   enableSwapAsymKey: SecureConsts.DEFAULT_SECURE_SWAP_ASYM_KEY(),
25.   // 是否开启详细日志
26.   // 在正式环境中, 请禁用
27.   enableDebugLog: process.env.NODE_ENV !== 'prod',
28.   // 加密配置的白名单url
29.   whileList: ['/secure/key', '/secure/clientKey', '/secure/swapKey'],
30.   // 加密URL的URL白名单
31.   encWhiteList: ['/login', '/logout']
32. }
33.
34. export default SecureConfig
```

## 拓展与变更

- 默认情况下, 使用RSA+AES+StringSignature实现安全传输
- 同时, 提供了快捷的可变方案
- 在SecureProvider中, 定义了这些方法的替代入口
  - asymmetricEncryptor 指定非对称加密算法的实现
  - symmetricEncryptor 指定对称加密算法的实现
  - messageDigester 指定摘要签名算法的实现
  - 同时, 在后端配置中, 都是以Supplier形式提供
    - 并且需要提供响应算法的密钥或密钥对生成器
    - symmetricKeySupplier 对称加密密钥生成器
    - asymmetricKeyPairSupplier 非对称密钥对生成器
  - 在前端配种中, 只需要实现即可, 实现方法可以参考默认实现
- 例如
- 使用其他非对称加密算法替代RSA, 例如ElGamal
  - 特别注意, 算法需要支持签名和验签, 也就是私钥加密公钥解密模式
  - 一般的非对称加密算法, 都是公钥加密私钥解密的
- 使用其他对称加密算法替代AES, 例如DES, 3Des
- 使用其他签名摘要算法替代StringSignature, 例如MD5, SHA1, SHA256, Hmac
- 如果不使用内置实现, 则可以根据内置的SM国密算法实现, 进行实现自己的方案