RSA+AES+SHA256 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

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秘钥
- 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秘钥对
        - 生成自己的秘钥对
        - 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秘钥对
        - 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

        - 客户端的秘钥处理
            - 客户端能够生成RSA秘钥对
                - 客户端生成自己的RSA秘钥对
                - 将自己的公钥发送秘钥交换请求
                - 得到服务端的公钥
            - 客户端不能够生成RSA秘钥对
                - 客户端请求自己的私钥时，生成随机的客户端秘钥对
                - 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秘钥
        - 使用客户端的RSA公钥对AES秘钥加密，放入请求头sswh
        - 使用AES秘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秘钥
        - 使用服务端的RSA私钥对AES秘钥加密，放入响应头sswh
        - 使用AES秘钥对响应体进行加密
        - 如果发现客户端的RSA秘钥签名和服务端最新的RSA签名不一致
        - 则表示客户端的RSA秘钥应该更新，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秘钥
        - 使用得到的AES秘钥解密响应体得到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秘钥
        - 使用得到的AES秘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
        - 由于客户端的秘钥对绑定问题
        - 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秘钥对
        - 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秘钥对
```

- 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秘钥对，并返回客户端私钥clientPrivateKey，服务器保留
   客户端公钥，这里实现客户端与客户端秘钥对的绑定
```

- 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秘钥
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
```

```
if(reqSign != sign){
    消息摘要验证失败
}
// 验证是否重放
if(exists(nonce)){
    重放请求验证失败
}
// 解密aes秘钥
let aesKey=RSA.privateKeyDecrypt(encAesKey,serverPrivateKey)
// 解密消息体
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

```
@SecureParams
@PostMapping("safe")
public Object safe(@RequestBody UserDto user){
    return user;
}
```

- 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

```
@SecureParams
@PostMapping("param")
public Object param(@SecureParams String password){
    System.out.println("password:"+password);
    return password;
}
```

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
    - 当使用秘钥交换时，使用 secureTransfer.getWebAsymPublicKeyAndSwap(request,clientKey) 交换获

- 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
  - 交换秘钥时，则使用接口 secure/swapKey

```java
@ConditionalOnExpression("${i2f.springboot.config.secure.api.enable:true}")
@RestController
@RequestMapping("secure")
public class SecureController {

    @Autowired
    private SecureTransfer secureTransfer;

    @Autowired
    private SecureConfig secureConfig;

    @SecureParams(in = false, out = false)
    @PostMapping("key")
    public String key() {
        String pubKey = secureTransfer.getWebAsymPublicKey();
        return pubKey;
    }

    @SecureParams(in = false, out = false)
    @PostMapping("clientKey")
    public String clientKey(HttpServletRequest request) {
        if(secureConfig.isEnableSwapAsymKey()){
            throw new SecureException(SecureErrorCode.BAD_SECURE_REQUEST,"服务端不允许请求秘钥策略");
        }
        String priKey = secureTransfer.getWebClientAsymPrivateKey(request);
        return priKey;
    }

    @SecureParams(in = false, out = false)
    @PostMapping("swapKey")
    public String swapKey(HttpServletRequest request, @RequestBody String clientKey) throws Exception {
        String pubKey = secureTransfer.getWebAsymPublicKeyAndSwap(request,clientKey);
        return pubKey;
    }
}
```

- 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

```
this.$axios({
    url: 'secure/key',
    method: 'post'
  }).then(({data}) => {
    SecureTransfer.saveAsymOthPubKey(data)
  })

this.$axios({
    url: 'secure/clientKey',
    method: 'post'
  }).then(({data}) => {
    SecureTransfer.saveAsymSlfPriKey(data)
  })
```

- 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)
})
```

- 此获取RSA公钥的代码
- 如果是使用**Vue**等虚拟DOM主体时
- 建议在**Vue**等主体的初始化时进行调用
- 下面以**Vue**为例
    - 在**Vue**主体实例创建时调用获取RSA公钥
    - 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
    - 否则可能出现请求失败，后端无法解密情况
    - 同时，为 **SecureCallback** 绑定对应的回调函数
    - 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
    - 避免刷新页面来刷新秘钥

```bash
App.vue
```

```
import SecureTransfer from "@/secure/core/secure-transfer";
import SecureCallback from '@/secure/core/secure-callback'
```

```
import SecureConfig from "@/secure/secure-config";

export default {
  name: 'App',
  components: {

  },
  created() {
    if(SecureConfig.enableSwapAsymKey){
      this.swapAsymKey()
      SecureCallback.callSwapKey = this.swapAsymKey
      let _this = this
      window.rsaTimer = setInterval(function () {
        _this.swapAsymKey()
      }, 5 * 60 * 1000)
    }else{
      this.initAsymOthPubKey()
      this.initAsymSlfPriKey()
      SecureCallback.callPubKey = this.initAsymOthPubKey
      SecureCallback.callPriKey = this.initAsymSlfPriKey
      let _this = this
      window.rsaTimer = setInterval(function () {
        _this.initAsymPubKey()
      }, 5 * 60 * 1000)
    }
  },
  destroyed() {
    clearInterval(window.rsaTimer)
  },
  methods: {
    swapAsymKey(){
      this.$axios({
          url: 'secure/swapKey',
          method: 'post',
          data: {
            key: SecureTransfer.loadWebAsymSlfPubKey()
          }
        }).then(({data}) => {
          console.log('SECURE_KEY', data)
          SecureTransfer.saveAsymOthPubKey(data)
        })
    },
    initAsymOthPubKey() {
      this.$axios({
        url: 'secure/key',
        method: 'post'
```

```
      }).then(({data}) => {
        console.log('SECURE_KEY', data)
        SecureTransfer.saveAsymOthPubKey(data)
      })
    },
    initAsymSlfPriKey() {
      this.$axios({
        url: 'secure/clientKey',
        method: 'post'
      }).then(({data}) => {
        console.log('SECURE_KEY', data)
        SecureTransfer.saveAsymSlfPriKey(data)
      })
    }
  }
}
```

如何使用

服务端（springboot环境）

安装

- maven添加依赖

```
<!-- 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 -->
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcprov-jdk15on</artifactId>
    <version>1.64</version>
</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

```
<!-- 增加了对国密SM系列算法的支持 -->
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcpkix-jdk15on</artifactId>
    <version>1.64</version>
</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

```
"dependencies": {
    "axios": "0.21.0",
    "js-base64": "^3.6.1",
    "crypto-js": "^4.1.1",
    "jsrsasign": "^10.8.6",
    "vue": "^2.6.14",
    "vue-router": "^3.0.1",
  },
```

- 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

```
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

```
- web-root
    - src
        - secure
            - secure-vue-main.js
            - secure-config.js
            - secure-axios.js
            - server.js
            - ...
        - 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

```
// request拦截器
request.interceptors.request.use(config => {

  console.log('headers:',config.headers);

  // 核心过滤器
  SecureTransferFilter.requestFilter(config)

  console.log('reqUrl:',config.url);

  return config
}, error => {
  console.log(error)
  Promise.reject(error)
})
```

- 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

```
// 定义请求拦截
BaseRequest.interceptors.request.use(config => {

  SecureTransfer.getSecureHeaderInto(config.headers, true, true)
  SecureTransferFilter.requestFilter(config)

  return config
})
```

- 添加响应拦截器

```
// 响应拦截器
request.interceptors.response.use(res => {
    console.log('res:',res);

    // 核心过滤器
    SecureTransferFilter.responseFilter(res);

    // 未设置状态码则默认成功状态
    let code = res.data.code ;
    if(code==undefined || code==null){
      code=200;
    }
    // 获取错误信息
    const msg =  res.data.msg
```

```
    if (code !== 200) {
      console.warn(msg);
      return Promise.reject(new Error(msg))
    } else {
      return res
    }
  },
  error => {
    SecureTransferFilter.responseFilter(error.response)
    console.log('err' , error)
    return Promise.reject(error)
  }
)
```

- 下面为了方便使用，将其绑定到Vue原型上

```
import Vue from 'vue'

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

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

```
/**
 * RSA工具
 */
// 注释掉webpack引入方式
// import JSEncrypt from '../static/jsencrypt'

const RsaUtil = {
    ...
```

```
index.html
```

```
<html>
<head>
```

```
    ...
    <!-- 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 -->
    <script src="./jsencrypt.js"></script>
    ...
</head>
</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

```
// 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
// 可能是大多数情况下使用的
// 方法参数：是否开始URL参数加密，是否开启编码URL转发
// 返回值：一个headers对象
secureTransfer.getSecureHeader(openSecureParams,openSecureUrl)
// 使用场景，已经有了一些headers值，需要添加加密标记时
// 可能少部分场景使用
// 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
// 返回值，入参的headers对象
secureTransfer.getSecureHeaderInto(headers,openSecureParams,openSecureUrl)
```

```
this.$axios({
    url: 'test/safe',
    method: 'POST',
    data:{
      userId:'1001',
      userName: '张',
      tel: '13122223333',
      password: 'pass'
    },
    headers:this.$secureTransfer.getSecureHeader(false,false)
  }).then(({data})=>{
    this.form.output=data;
  })
```

- 使用URL参数params

```
this.$axios({
    url:'test/param',
```

```
    method:'POST',
    params:{
      password: this.form.input
    },
    headers:this.$secureTransfer.getSecureHeader(true,false)
  }).then(({data})=>{
    this.form.output=data
  })
```

- 使用编码后的URL转发

```
this.$axios({
    url:'test/enc',
    method:'POST',
    params:{
      password: this.form.input
    },
    headers:this.$secureTransfer.getSecureHeader(false,true)
  }).then(({data})=>{
    this.form.output=data
  })
```

- 全功能开启

```
this.$axios({
    url:'test/all',
    method:'POST',
    params:{
      password: this.form.input
    },
    headers:this.$secureTransfer.getSecureHeaderInto({
            token: sessionStorage.getItem('token')
        },true,true)
  }).then(({data})=>{
    this.form.output=data
  })
```

后端配置详解

```
# secure 配置
i2f:
  springboot:
    config:
      secure:
```

```yaml
        # 是否开启
        enable: true
        # asym秘钥的存储路径，默认../
        asym-store-path: ../
        # 响应字符集，默认UTF-8
        responseCharset: 'UTF-8'
        # Asym秘钥长度，默认1024，可选1024,2048
        asymKeySize: 1024
        # Symm秘钥长度，默认128，可选128,192,256
        symmKeySize: 128
        # 随机秘钥生成的随机数的最大值，默认8192
        randomKeyBound: 8192
        # 一次性消息的保持时间秒数，默认6*60
        # 这段时间内重复出现的nonce将会被认为是重放请求被拦截
        nonceTimeoutSeconds: 360
        # 是否启动动态Asym更新秘钥，默认true
        enableDynamicAsymKey: true
        # 每次更新秘钥的时长秒数，默认6*60
        dynamicRefreshDelaySeconds: 360
        # 最多保留多少历史秘钥，默认5
        dynamicMaxHistoriesCount: 5
        # 客户端秘钥对的获取策略，是否是本地生成交换策略，默认true
        enableSwapAsymKey: true
        # 用于存储安全头的请求头名称，默认sswh
        headerName: sswh
        # 安全头格式的分隔符，默认;
        headerSeparator: ;
        # 动态刷新Asym秘钥的响应头，默认skey
        dynamicKeyHeaderName: skey
        # URL加密的后端forward路径
        encUrlPath: /enc/
        # 请求URL参数加密的加密参数名
        parameterName: sswp
        # 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
        defaultControl:
          # 入站是否安全
          in: true
          # 出站是否安全
          out: true
        # 白名单配置列表项，符合ant-match模式
        whiteList:
          # 进出站都忽略的列表清单
          bothPattens:
            - /file/**
            - /secure/key
          # 进站忽略的列表清单
```

```
      inPattens:
        - /common/upload/**
      # 出站忽略的列表清单
      outPattens:
        - /common/download/**
    # AOP功能
    aop:
      # 是否启用AOP功能，默认true
      # 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
      # 包含支持解密String类型的RequestParam请求参数
      # 包含controller为String类型返回值时的特殊处理
      # 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
      enable: true
    # 内置的API接口
    api:
      # 是否开启默认的API响应Asym秘钥获取请求，默认true
      # 请求路径：/secure/key
      enable: true
    # 内置的URL请求路径转发接口
    enc-url-forward:
      # 是否开启enc的url解密请求转发，默认true
      # 请求路径: /enc/**
      enable: true
    # MVC替换converter为spring注册converter实现自定义
    # 当出现如果自定义的converter不生效时，需要开启
    # 当long类型需要转换为string类型给前端时，必须开启
    mvc:
      # 是否开启自定义替换converter
      enable: true
    # 针对jackson的拓展自定义配置
    jackson:
      # 是否开启自定义配置
      enable: true
      # 是否开启long类型转string类型给前端
      enableLongToString: true
      # 注意，LocalDateTime的格式化模式和spring.jackson.date-format配置一致
      # 因此，不用特殊配置
      # 定义LocalDate的格式化模式
      localDateFormat: yyyy-MM-dd
      # 定义LocalTime的格式化模式
      localTimeFormat: HH:mm:ss
```

前端配置详解

```
/**
```

```
 * 主配置
 */
import SecureConsts from './consts/secure-consts'

const SecureConfig = {
  // Asymm秘钥长度，默认1024，可选1024,2048
  asymKeySize: SecureConsts.RSA_KEY_SIZE_1024(),
  // Symm秘钥长度，默认128，可选128,192,256
  symmKeySize: SecureConsts.AES_KEY_SIZE_128(),
  // 用于存储安全头的请求头名称，默认sswh
  headerName: SecureConsts.DEFAULT_SECURE_HEADER_NAME(),
  // 动态刷新Asym秘钥的响应头，默认skey
  dynamicKeyHeaderName: SecureConsts.SECURE_DYNAMIC_KEY_HEADER(),
  clientKeyHeaderName: SecureConsts.SECURE_CLIENT_KEY_HEADER(),
  clientAsymSignName: SecureConsts.DEFAULT_SECURE_CLIENT_ASYM_SIGN_NAME(),
  // 安全头格式的分隔符，默认;
  headerSeparator: SecureConsts.DEFAULT_HEADER_SEPARATOR(),
  // 指定在使用编码URL转发时的转发路径
  encUrlPath: SecureConsts.ENC_URL_PATH(),
  // 安全URL参数的参数名称
  parameterName: SecureConsts.DEFAULT_SECURE_PARAMETER_NAME(),
  // 客户端秘钥对的获取策略，是否是本地生成交换策略
  enableSwapAsymKey: SecureConsts.DEFAULT_SECURE_SWAP_ASYM_KEY(),
  // 是否开启详细日志
  // 在正式环境中，请禁用
  enableDebugLog: process.env.NODE_ENV != 'prod',
  // 加密配置的白名单url
  whileList: ['/secure/key', '/secure/clientKey', '/secure/swapKey'],
  // 加密URL的URL白名单
  encWhiteList: ['/login', '/logout']
}

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秘钥或密钥对生成器
    - symmetricKeySupplier 对称加密秘钥生成器

        - asymmetricKeyPairSupplier 非对称秘钥对生成器
    - 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