前后端分离的开发方式，我们以接口为标准来进行推动，定义好接口，各自开发自己的功能，最后进行联调整合。无论是开发原生的 APP 还是 webapp 还是 PC 端的软件,只要是前后端分离的模式，就避免不了调用后端提供的接口来进行业务交互。

网页或者 app,只要抓下包就可以清楚的知道这个请求获取到的数据，这样的接口对爬虫工程师来说是一种福音，要抓你的数据简直轻而易举。

数据的安全性非常重要，特别是用户相关的信息，稍有不慎就会被不法分子盗用，所以我们对这块要非常重视，容不得马虎。

如何保证 API 调用时数据的安全性?

1. 通信使用 https
2. 请求签名，防止参数被篡改
3. 身份确认机制，每次请求都要验证是否合法
4. APP 中使用 ssl pinning 防止抓包操作
5. 对所有请求和响应都进行加解密操作
6. 等等方案…….

对所有请求和响应都进行加解密操作

方案有很多种，当你做的越多，也就意味着安全性更高，今天我跟大家来介绍一下对所有请求和响应都进行加解密操作的方案，即使能抓包，

即使能调用我的接口，但是我返回的数据是加密的，只要加密算法够安全，你得到了我的加密内容也对我没什么影响。

像这种工作最好做成统一处理的，你不能让每个开发都去关注这件事情，如果让每个开发去关注这件事情就很麻烦了，返回数据时还得手动调用下加密的方法，接收数据后还得调用下解密的方法。

为此，我基于 Spring Boot 封装了一个 Starter，内置了 AES 加密算法。GitHub 地址如下：

https://github.com/yinjihuan/spring-boot-starter-encrypt

先来看看怎么使用，可以下载源码，然后引入即可，然后在启动类上增加@EnableEncrypt 注解开启加解密操作：

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

增加加密的 key 配置：

```
spring.encrypt.key=abcdef0123456789
spring.encrypt.debug=false
```

- spring.encrypt.key：加密 key，必须是 16 位

- spring.encrypt.debug：是否开启调试模式,默认为 false,如果为 true 则不启用加解密操作

为了考虑通用性，不会对所有请求都执行加解密，基于注解来做控制

响应数据需要加密的话，就在 Controller 的方法上加@Encrypt 注解即可。

```
@Encrypt
@GetMapping("/list")
public Response queryNews(String city) {
    return Response.ok(city);
}
```

当我们访问/list 接口时，返回的数据就是加密之后 base64 编码的格式。

还有一种操作就是前段提交的数据，分为 2 种情况，一种是 get 请求，这种暂时没处理，后面再考虑，目前只处理的 post 请求，基于 json 格式提交的方式，也就是说后台需要用@RequestBody 接收数据才行，需要解密的操作我们加上@Decrypt 注解即可。

```
@Decrypt
@PostMapping("/save")
public Response savePageLog(@RequestBody PageLogParam logParam,
HttpServletRequest request) {
    pageLogService.save(logParam);
    return Response.ok();
}
```

加了@Decrypt 注解后，前端提交的数据需要按照 AES 加密算法，进行加密，然后提交到后端，后端这边会自动解密，然后再映射到参数对象中。

上面讲解的都是后端的代码，前端使用的话我们以 js 来讲解，当然你也能用别的语言来做，如果是原生的安卓 app 也是用 java 代码来处理。

前端需要做的就 2 件事情：

1. 统一处理数据的响应，在渲染到页面之前进行解密操作
2. 当有 POST 请求的数据发出时，统一加密

js 加密文件请参考我 GitHub 中 encrypt 中的 aes.js,crypto-js.js,pad-zeropadding.js

我们以 axios 来作为请求数据的框架，用 axios 的拦截器来统一处理加密解密操作

首先还是要封装一个 js 加解密的类，需要注意的是加密的 key 需要和后台的对上，不然无法相互解密，代码如下：

```
var key  = CryptoJS.enc.Latin1.parse('abcdef0123456789');
var iv   = CryptoJS.enc.Latin1.parse('abcdef0123456789');
// 加密
function EncryptData(data) {
    var srcs = CryptoJS.enc.Utf8.parse(data);
    var encrypted = CryptoJS.AES.encrypt(srcs, key, {
        mode : CryptoJS.mode.ECB,
        padding : CryptoJS.pad.Pkcs7
    });
```

```
    return encrypted.toString();
}
// 解密
function DecryptData(data) {
    var stime = new Date().getTime();
    var decrypt = CryptoJS.AES.decrypt(data, key, {
        mode : CryptoJS.mode.ECB,
        padding : CryptoJS.pad.Pkcs7
    });
                    var                    result                    =
JSON.parse(CryptoJS.enc.Utf8.stringify(decrypt).toString());
    var etime = new Date().getTime();
    console.log("DecryptData Time:" + (etime - stime));
    return result;
}
```

axios 拦截器中统一处理代码:

```
// 添加请求拦截器
axios.interceptors.request.use(function (config) {
    // 对所有 POST 请加密，必须是 json 数据提交，不支持表单
    if (config.method == "post") {
        config.data = EncryptData(JSON.stringify(config.data));
    }
    return config;
  }, function (error) {
    return Promise.reject(error);
});
// 添加响应拦截器
axios.interceptors.response.use(function (response) {
    // 后端返回字符串表示需要解密操作
    if(typeof(response.data) == "string"){
        response.data = DecryptData(response.data);
    }
    return response;
  }, function (error) {
    return Promise.reject(error);
});
```

到此为止，我们就为整个前后端交互的通信做了一个加密的操作，只要加密的 key 不泄露，别人得到你的数据也没用，问题是如何保证 key 不泄露呢?

服务端的安全性较高，可以存储在数据库中或者配置文件中，毕竟在我们自己的服务器上，最危险的其实就时前端了，app 还好，可以打包，但是要防止反编译等等问题。

如果是 webapp 则可以依赖于 js 加密来实现，下面我给大家介绍一种动态获取加密 key 的方式，只不过实现起来比较复杂，我们不上代码，只讲思路:

加密算法有对称加密和非对称加密，AES 是对称加密，RSA 是非对称加密。之所以用 AES 加密数据是因为效率高，RSA 运行速度慢,可以用于签名操作。

我们可以用这 2 种算法互补，来保证安全性，用 RSA 来加密传输 AES 的秘钥，用 AES 来加密数据，两者相互结合，优势互补。

其实大家理解了 HTTPS 的原理的话对于下面的内容应该是一看就懂的，HTTPS 比 HTTP 慢的原因都是因为需要让客户端与服务器端安全地协商出一个对称加密算法。剩下的就是通信时双方使用这个对称加密算法进行加密解密。

1. 客户端启动，发送请求到服务端，服务端用 RSA 算法生成一对公钥和私钥，我们简称为 pubkey1,prikey1，将公钥 pubkey1 返回给客户端。
2. 客户端拿到服务端返回的公钥 pubkey1 后，自己用 RSA 算法生成一对公钥和私钥，我们简称为 pubkey2,prikey2，并将公钥 pubkey2 通过公钥 pubkey1 加密，加密之后传输给服务端。
3. 此时服务端收到客户端传输的密文，用私钥 prikey1 进行解密，因为数据是用公钥 pubkey1 加密的，通过解密就可以得到客户端生成的公钥 pubkey2
4. 然后自己在生成对称加密，也就是我们的 AES,其实也就是相对于我们配置中的那个 16 的长度的加密 key,生成了这个 key 之后我们就用公钥 pubkey2 进行加密，返回给客户端，因为只有客户端有 pubkey2 对应的私钥 prikey2，只有客户端才能解密，客户端得到数据之后，用 prikey2 进行解密操作，得到 AES 的加密 key，最后就用加密 key 进行数据传输的加密，至此整个流程结束。

spring-boot-starter-encrypt 原理

最后我们来简单的介绍下 spring-boot-starter-encrypt 的原理吧，也让大家能够理解为什么 Spring Boot 这么方便，只需要简单的配置一下就可以实现很多功能。

启动类上的@EnableEncrypt 注解是用来开启功能的,通过@Import 导入自动配置类

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import({EncryptAutoConfiguration.class})
public @interface EnableEncrypt {
}
```

EncryptAutoConfiguration 中配置请求和响应的处理类，用的是 Spring 中的 RequestBodyAdvice 和 ResponseBodyAdvice，在 Spring 中对请求进行统计处理比较方便。如果还要更底层去封装那就要从 servlet 那块去处理了。

```
@Configuration
@Component
@EnableAutoConfiguration
@EnableConfigurationProperties(EncryptProperties.class)
public class EncryptAutoConfiguration {
    /**
     * 配置请求解密
     * @return
     */
    @Bean
    public EncryptResponseBodyAdvice encryptResponseBodyAdvice() {
        return new EncryptResponseBodyAdvice();
    }
    /**
     * 配置请求加密
     * @return
     */
    @Bean
    public EncryptRequestBodyAdvice encryptRequestBodyAdvice() {
        return new EncryptRequestBodyAdvice();
    }
}
```

通过 RequestBodyAdvice 和 ResponseBodyAdvice 就可以对请求响应做处理了，大概的原理就是这么多了