1. 权限控制
2. 用户角色权限控制

①RBAC基本概念：用户、角色、权限。用户赋予角色，角色赋予权限，从数据库设计角度来说主要是5张表：用户表、角色表、权限表、用户角色中间表、角色权限中间表

②变种：多角色赋予权限交叉时的交集、并集、差集等处理；用户直接赋予权限时和角色赋予权限时的权限交叉时的交集、并集、差集等处理

1. 数据权限控制

一般对返回的实体类控制，如实体类返回userName和password，那么根据用户或角色对数据的权限控制来决定userName或paswword的数据输出，当然这里也涉及到用户数据权限和角色数据权限交叉时的交集、并集、差集等处理

3、token令牌

登录后需要校验token，包括接口和数据的访问权限及token是否失效等

1. 入参、出参加解密

1、建议采用DES+RSA的组合方式进行加解密，具体方法见下面的表格：

|  |  |  |  |
| --- | --- | --- | --- |
| 请求方式 | GET | 不加密请求示例 | [https://www.abc.com/getUserInfoById?id=123&pid=11](https://www.cpic.com/getUserInfoById?id=123&pid=11)  [https://www.abc.com/getUserInfoById/{id}/{pid}](https://www.cpic.com/getUserInfoById?id=123&pid=11) |
| 说明 | header组成： 1、isAuth：0（不加密）；1（加密） 2、token（基本除登录外都需要） |
| 加密请求示例 | <https://www.abc.com/getUserInfoById> |
| 说明 | header组成： 1、isAuth：0（不加密）；1（加密） 2、token（基本除登录外都需要） 3、currentTime：格式为yyyyMMddHHmmss 4、nonce：长度为24位的随机小写字母和数字的组合 5、code：RSA后端公钥加密(参数的KEY和VALUE的JSON字符串)，比如原来是https://www.abc.com/getUserInfoById/{id}/{pid}那么参数的KEY和VALUE的JSON字符串就是形如：{"id":"a","pid":"b"}，对于GET封装的JSON数据一般具有数据少但字段名多变的特点，所以建议使用JSONObject去接收，然后逐个get获取 6、sign：RSA前端私钥签名(MD5(currentTime+","+nonce+","+参数的KEY和VALUE的JSON字符串)) 注： 1、一般GET和DELETE请求是不需要加密的，加密的基本都是POST和PUT请求； 2、若请求中带有中文需要URLEncode下或采用POST传参； 3、所有header参数的key和value建议都采用字符串类型 4、实践中针对header组成的第5点参考POST加密请求的第4点 5、实践中针对header组成的第6点参考POST加密请求的第5点 |
| DELETE | 参考GET（在有些环境下无法使用DELETE） | |
| POST | 不加密请求示例 | <https://www.abc.com/saveUser> |
| 说明 | body参数为： {  'username':'aaaaaa',  'password':Base64('abc123'),  'address':'cccccc' } header组成： 1、isAuth：0（不加密）；1（加密） 2、token（基本除登录外都需要） |
| 加密请求示例 | <https://www.abc.com/saveUser> |
| 说明 | 请求参数记为A： {  'username':'zhangsan',  'password':Base64('abc123'),  'address':'地址',  'phone':'12345685462' } 实际body传参： {  'code':RSA后端公钥加密(A的字符串),  'sign':RSA前端私钥签名(MD5(currentTime+","+nonce+","+A的字符串)) } header组成： 1、isAuth：0（不加密）；1（加密） 2、token（基本除登录外都需要） 3、currentTime：格式为yyyyMMddHHmmss 4、nonce：长度为24位的随机小写字母和数字的组合 注： 1、前端请求参数封装的参数应该是后端请求参数封装的个数的子集，即前端请求的所有参数必须都包含在后端的请求参数封装里 2、请求参数A里的部分字段还可以再特殊处理（如再加密），但是一般不建议，没这个必要 3、所有header参数的key和value建议都采用字符串类型  4、实践中A的字符串太长，又由于前端JS和后端JAVA的加解密方式可能不一致，导致超过128位后分段加解密出错，从官方的讨论里（https://github.com/travist/jsencrypt/issues/137）可以看到，当加密的字符串太长时，也是建议采用如RSA+DES结合的方式。因此，最终修改为： {  'code':DES加密(A的字符串,nonce),  'sign':RSA前端私钥签名(MD5(currentTime+","+nonce+","+A的字符串)) } header组成： 其它不变，nonce：RSA后端公钥加密(nonce)，nonce长度为24位的随机小写字母和数字的组合 5、后端就是反向校验操作：先是DES解密code，然后用前端公钥验签 |
| PUT | 参考POST（在有些环境下无法使用PUT） | |

1. 采用DES+RSA的组合方式进行加解密的一些问题解答

①是如何做到防止信息泄露？答：通过对关键数据的加密；

②是如何防止信息篡改的？答：通过签名，签名需要私钥，如果私钥泄露，那么就会被信息篡改，所以私钥的保密性很重要，如有必要需定期更换。这里的推荐做法是同一个用户每次登录都会获得一套不同的公钥和私钥；

③黑客窃取body数据伪造请求怎么办？答：currentTime有时间戳保证，如果服务端时间与客户端请求时间不在0-1分钟的时间段内（具体视不同的项目要求，但是一般不超过3分钟），则提示请求超时；

④黑客修改时间戳怎么办？答：单单修改currentTime的时间戳是没用的，除非也改变code或sign的值，那么这个又回到第②点私钥泄露问题上来了；

⑤客户端与服务端时间不一致怎么办？答：一般都是北京时间且计算机会自动校准时间，所以客户端与服务端时间差最多在1分钟之内，但是也不排除客户故意修改本地时间影响客户端读取到正确时间，那么可以前端用户一打开应用先同步下后端时间。

1. 总结

①在加解密上采用上述这套较为通用的方案（在实际生产上也得到过验证）已经足够，当然加密手段千变万化，尤其银行、证券、保险、公安、军事等都有自己的一套加解密算法规则；

②在RSA中，加密是为了防止信息泄露，签名是为了防止信息篡改；

③在RSA中四个概念运用就是：公钥加密、私钥解密、私钥签名、公钥验签，从理论上来说私钥加密公钥解密等也是可以的；

④本套方案是对参数进行全加密，当然也可以部分加密，如总结中第①点所言，加解密手段千变万化。另外，除非对数据安全要求特别严苛，不然的话只需请求参数加密，返回参数可以不加密；

⑤加解密手段越复杂，必然对性能造成影响，所以在实际项目中需要把握好度。

1. 网络控制
2. IP白名单
3. IP（或接口）请求（单位时间内）限流
4. HTTPS（证书）请求（或HTTP2.0）

4、防火墙

四、代码控制

1、前端如javascript的文件的源码加密或混淆

2、后端如java的class文件加密

五、数据安全

1、敏感数据（手机、身份证\*替换或加解密）

2、数据库主键

非高并发、非大数据量、非经常用到主键索引、非主键排序等情况下使用字符串型且字符串采用UUID或使用数字型且数字自增即可；在较高安全性同时兼顾高并发、大数据量、主键索引、主键排序等情况下推荐做法：

循环自增序号（1-9） + 时间（new Date().getTime()）+随机4位数字+分布式系统编号

注：当生成的主键ID为数字型且长度超过16位时传给前端可能会造成精度损失，此时后端需要将Long类型转成String类型，有个偷懒办法就是直接将实体类Long改为String即可，但是个人不推荐

3、日志记录新增、修改、删除等操作的数据记录

4、容灾与备份（主从等）

六、接口对接

七、缓存（缓存穿透、缓存击穿、缓存雪崩）