**客户端与服务器通讯使用HTTPS原理分析与实操**

尉涛

2019/12/20

目录

[一、 概述 3](#_Toc27747842)

[二、 基础概念 3](#_Toc27747843)

[1. 网络层次 3](#_Toc27747844)

[2. HTTP的缺陷 3](#_Toc27747845)

[3. HTTPS的优势及原理 4](#_Toc27747846)

[4. SSL/TSL原理 4](#_Toc27747847)

[(1). 加密技术 4](#_Toc27747848)

[(2). 身份验证 5](#_Toc27747849)

[(3). 根证书与证书链 6](#_Toc27747850)

[(4). CA证书的使用流程 7](#_Toc27747851)

[(5). SSL/TSL简介 8](#_Toc27747852)

[三、 实操 9](#_Toc27747853)

[1. 使用OpenSSL生成自己的CA证书 9](#_Toc27747854)

[2. 使用SpringBoot搭建简单后台 12](#_Toc27747855)

[3. Android端使用HTTPS访问SpringBoot后台 13](#_Toc27747856)

[(1). 构建OkhttpClient 13](#_Toc27747857)

[(2). 请求与结果 15](#_Toc27747858)

[4. 双向认证（拓展） 15](#_Toc27747859)

[(1). 生成新证书 15](#_Toc27747860)

[(2). 配置Android端 16](#_Toc27747861)

[(3). 转换根证书格式 16](#_Toc27747862)

[(4). 配置服务端 16](#_Toc27747863)

[四、 其他问题 17](#_Toc27747864)

[1. 关于双端证书 17](#_Toc27747865)

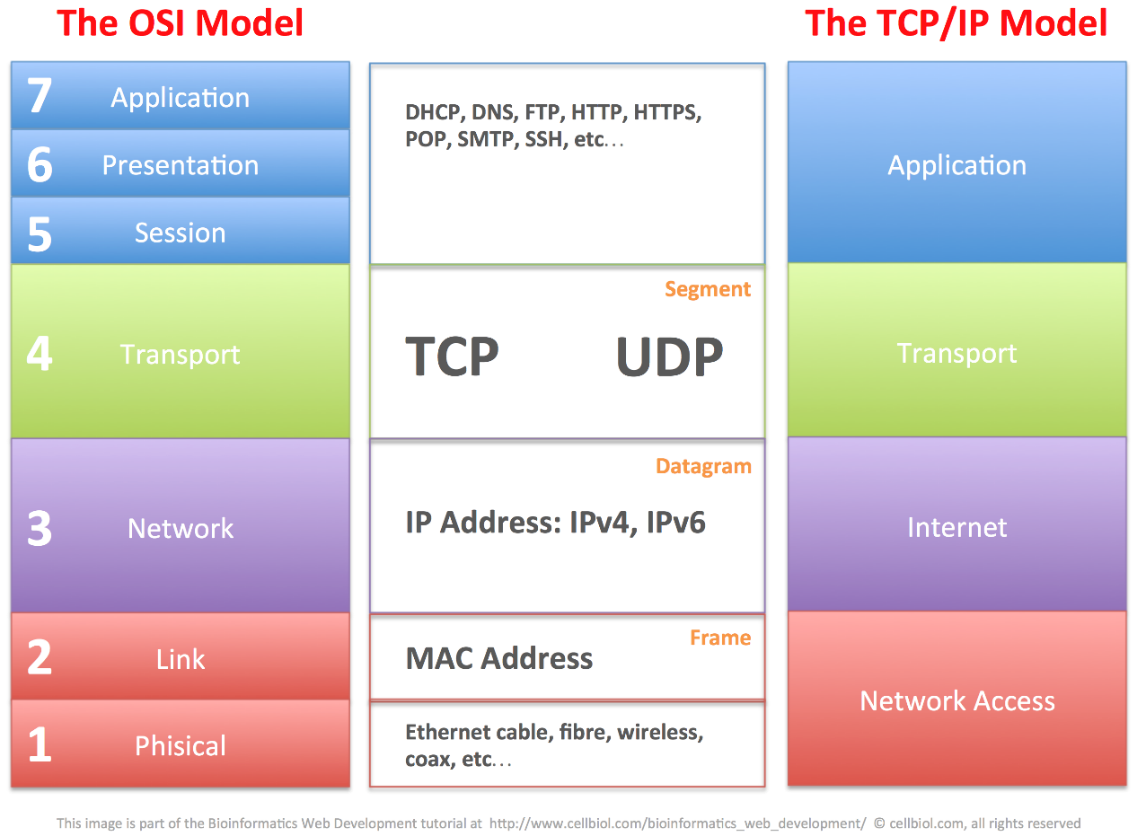
[2. 关于客户端根证书的更新策略 17](#_Toc27747866)

# 概述

为了保护用户的信息安全、保护开发商自己的商业利益，有效防止信息截取、冒充、DoS攻击等问题，建议App与服务器之间的通讯使用HTTPS协议。有些应用也使用了HTTPS协议，但是并没有按要求使用，达不到最好的效果。本文概述了HTTPS的原理及使用方法，参考本文，了解如何正确的使用HTTPS。

# 基础概念

## 网络层次



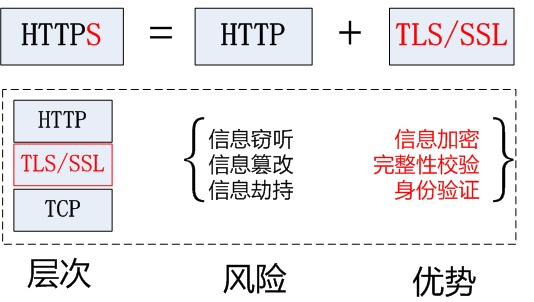
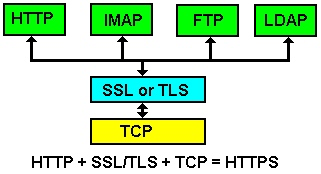
HTTP是基于TCP/IP的技术，属于应用层协议（TCP协议在传输层，IP在网络层）

## HTTP的缺陷

* + HTTP协议是没有加密的明文传输协议，信息容易暴露。
  + HTTP协议无法验证双方身份，任何人都可以请求服务端，导致数据不安全，甚至DoS攻击；并且服务端可以被伪装，App的请求信息可能被恶意利用。
  + HTTP协议无法验证报文的完整性，报文可能被篡改，典型的问题是前端被插入广告，或者遭遇中间人攻击（Man-in-the-Middle attack，MITM）。

## HTTPS的优势及原理

HTTPS在HTTP的基础上加入了这些功能：信息加密、报文完整性校验、身份验证。HTTPS实际上是在HTTP与TCP之间增加了一层TLS/SSL协议，可以说HTTPS是HTTP over SSL/TLS。

SSL/TLS层负责客户端和服务器之间的加解密算法协商、密钥交换、通信连接的建立。

## SSL/TSL原理

### 加密技术

上面说到HTTP协议是没有加密的明文传输协议，信息容易暴露。SSL/TSL中使用了加密技术防止以上问题的发生。

加密算法一般分为两种：

**对称加密**：加密与解密的密钥相同。以DES算法为代表。

优点是强度高、不易破解，缺点是无法安全的保管密钥，如客户端和服务器之间每次会话都使用固定的、相同的密钥加密和解密，肯定存在很大的安全隐患。当将密钥分发给通讯的双方时，一旦密钥被截获，则任何获得密钥的人都可以对信息解密。

**非对称加密**：加密与解密的密钥不相同。以RSA算法为代表。

在实际使用过程中，通常会将公钥发布出去给任何需要接收消息的目标用户，然后发送端利用私钥加密信息，此时只有持有公钥的一端才能对该信息进行解密，而公钥加密的信息则只有私钥一端可以解开，其他同样持有公钥的一方无法解开，从而相对能够保证信息的安全。

非对称密钥交换过程主要就是为了解决对称加密的密钥安全问题，使密钥的生成和使用更加安全。但同时也是HTTPS性能和速度严重降低的“罪魁祸首”。

SSL/TSL采用对称加密和非对称加密两者并用的**混合加密机制**，仅在交换密钥环节使用非对称加密方式，之后的建立通信交换报文阶段则使用对称加密方式，很好地利用了这两种加密方式。

虽然我们使用对称加密解决了HTTP明文的问题，也用非对称加密解决了对称加密的密钥交换问题，但是还有一种安全风险：

就是当分发公钥过程中，公钥会被第三方应用截获，而截获者可以自己再生成一对非法的公、私钥，然后将这个非法公钥发送给目标，代替截获的合法公钥，这样当有信息被截获时，就用获得的合法公钥对信息进行解密并读取或篡改信息后，再用非法的私钥重新加密信息，并发送到目标，目标将会用非法的公钥解密信息，而不会发现信息已被篡改，这就是一种中间人攻击。



### 身份验证

为了避免中间人攻击的发生，能够将合法的公钥分发到目标用户手中，SSL/TSL中使用到了**数字证书认证机构(CA，Certificate Authority) 和其相关机关颁发的公开密钥证书**。

CA是公钥基础设施（PKI即Public-Key Infrastructure）之一，PKI采用证书进行公钥管理，通过第三方的可信任机构（认证中心，即CA），把用户的公钥和用户的其他标识信息捆绑在一起，其中包括用户名和电子邮件地址等信息，以在Internet网上验证用户的身份。

通常的场景是消息发送方首先通过有效组织信息以及一个合法公钥，在CA机构申请一个合法证书，CA机构通过线上及线下的方式对申请者进行审核后，为其颁发有效的CA证书，该证书中包含了申请者合法的公钥，所以，在分发公钥时，只需将合法的证书分发出去即可。那么问题来了，如果证书被截取了，中间人能造一个假证书吗？要了解CA证书，我们还要先了解**证书链**。

### 根证书与证书链

每一个CA机构作为安全机制的一个环节，也必须同时拥有一个CA证书，来证明其是合法的CA证书提供商，全世界仅有几个公认的顶级合法CA提供商，这些提供商都为自己签发了一个用于认证其合法性的证书（这些证书通常会被默认安装在各种主流的浏览器中），而所有由这些机构颁发的证书都绑定到这些合法证书中的某一个上，构成一个证书的链条，链条的开始端就是这些机构自己的证书，也被成为“**根证书**”。根证书是顶级CA机构为自己签名的证书，也称为“自签名证书”。

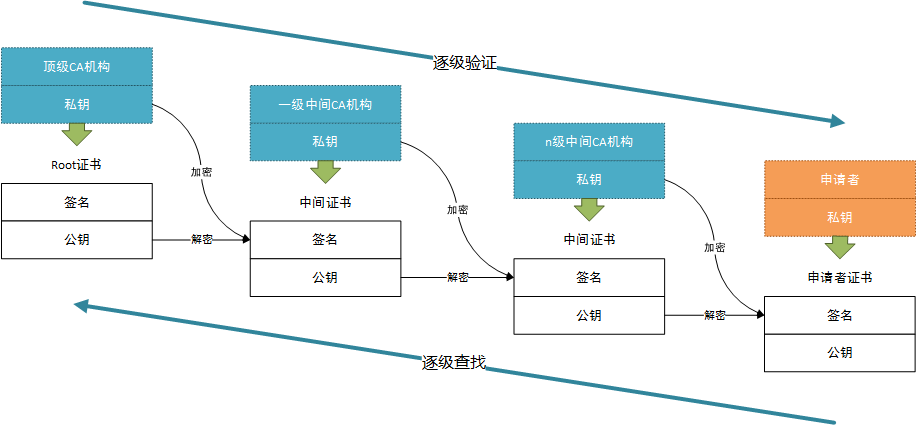
在每一个由顶级CA颁发的证书中，都包含一个**签名**，该签名是**使用顶级CA机构持有的私钥对申请证书者提供的摘要信息进行加密的密文**，这些密文只能通过根证书中包含的公钥进行解密（在根证书中包含顶级机构自己的公钥用于分发，而其保留一个私钥用于为其颁发的证书加密），而验证某证书是否合法时，就是通过**预先获得并安装的根证书中包含的公钥，对发送来的证书的签名进行解密**，然后用解密后的信息与证书的摘要信息进行比较，即可发现该证书是否被篡改或者是否过期等，每一个证书的摘要信息都包含证书拥有者的公钥，若证明证书合法，则证书中的摘要信息中包含的公钥也是合法的。



在现实中，有限的几个顶级CA机构不可能承担所有组织对与证书的申请，因此顶级机构会授权一些第三方机构作为中间CA机构来分担证书颁发的工作(也可能是其他原因)，这些被授权的中间CA机构同样需要向顶级CA机构申请一个证书，用于证明该机构的合法性，这些机构的证书将与顶级CA机构持有的根证书进行绑定，构成一个证书链的一部分。

这些中间机构持有的证书中也同样存在一个签名，是由顶级CA机构(或者上级)的密钥进行加密的，可以验证其证书的合法性，这些中间机构也同样会生成一对密钥，其持有私钥，而将公钥用于加密从这些机构申请的CA证书的签名部分。

每一个中间CA机构都可以像顶级CA机构一样为其他的第三方CA机构授权颁发证书，并加密其持有证书的签名部分，从而构成任意长度的证书链。



每个机构都要有自证证书（下图顶部的证书，自证证书也用于预装），由上级CA签发。每个机构也会签发多个证书。**对于CA，公钥用于申请，私钥用于签发。对于开发者，公钥用于申请并用于解密服务端发来的被加密过的对称加密的密钥，私钥用于加密对称加密的公钥。**（有点拗口，但是理解这句话很重要）



CA机构分发的证书分为两种类型：

* **CA机构证书**
* **服务证书**

CA机构证书就是CA认证机构持有的证书，而服务证书则是对外发布的证书，仅用于公钥的分发

认证服务证书是否有效时：

* 首先从服务证书顺证书链向上回溯寻找绑定的上级证书，并逐级找到顶级根证书（必须持有或安装了顶级根证书）
* 由于根证书是合法的，因此可以从根证书开始逐级向下验证证书链中每一级证书的合法性方法是：

1. 用上级证书的公钥解密当前证书的签名

2. 验证签名是否与摘要信息相同

3. 证明合法后，取出摘要中的公钥，用于验证下一级证书

* 最终验证到服务证书（也就是证书链的最末端证书）合法有效

### CA证书的使用流程

CA证书的申请、使用流程如下：



### SSL/TSL简介

SSL：（Secure Socket Layer，安全套接字层），位于可靠的面向连接的网络层协议和应用层协议之间的一种协议层。SSL通过互相认证、使用数字签名确保完整性、使用加密确保私密性，以实现客户端和服务器之间的安全通讯。该协议由两层组成：SSL记录协议和SSL握手协议。

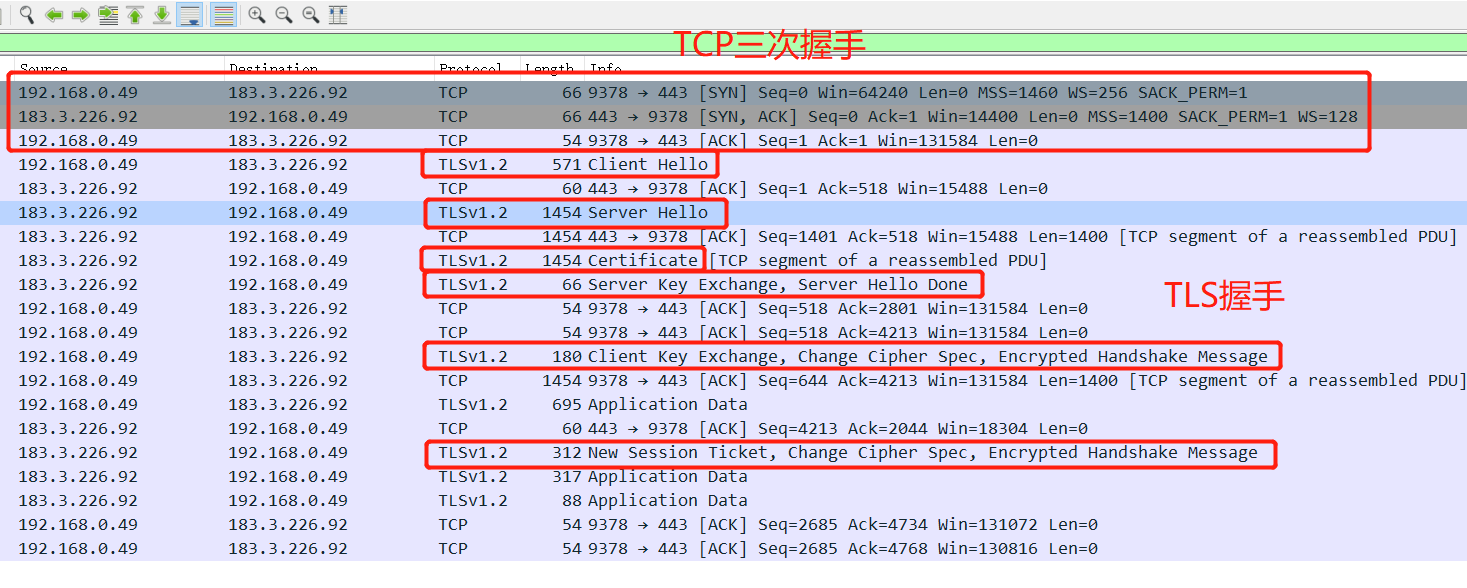
TLS：(Transport Layer Security，传输层安全协议)，用于两个应用程序之间提供保密性和数据完整性。该协议由两层组成：TLS记录协议和TLS握手协议。**TLS继承了SSL**。

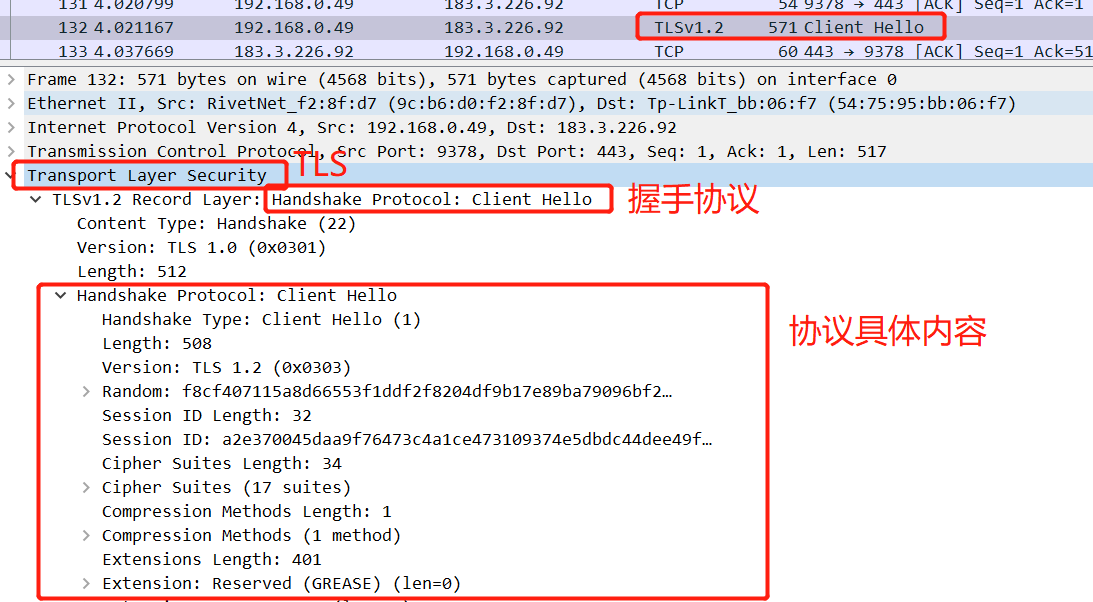
TLS有四个核心协议：

* 握手协议(handshake protocol)；
* 密钥规格变更协议(change cipher spec protocol)；
* 应用数据协议(application data protocol)；
* 警报协议(alert protocol)；

其中握手协议最为复杂，下图为在登陆QQ邮箱过程中，使用Wireshark抓包的结果，具体抓包分析可以参考

<https://zhuanlan.zhihu.com/p/27040041>





# 实操

## 使用OpenSSL生成自己的CA证书

OpenSSL下载（注意32位和64位的安装包是不同的）

<http://slproweb.com/products/Win32OpenSSL.html>

OpenSSL安装配置

<https://blog.csdn.net/qq_28938933/article/details/88832306>

OpenSSL使用指引（查了很多，每个都有一些缺漏，具体操作可以看下面的图片中的操作）

<https://www.jianshu.com/p/37ded4da1095>

<https://zhuanlan.zhihu.com/p/22816331>

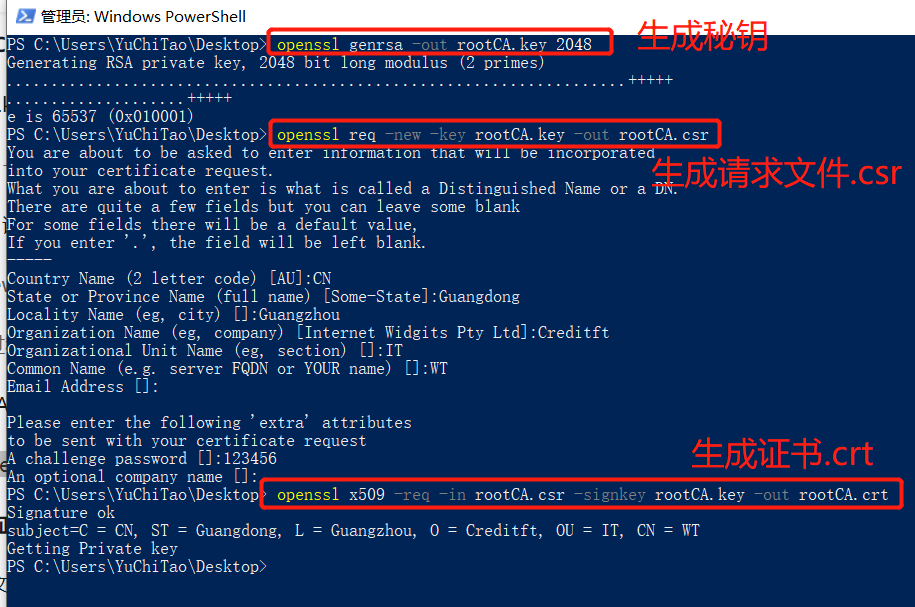
文件格式说明：

|  |  |
| --- | --- |
| **格式** | **说明** |
| .key | 密钥文件（也是.pem文件） |
| .csr | 证书签名请求（证书请求文件），含有公钥信息，certificate signing request的缩写 |
| .crt | 证书文件，certificate的缩写 |
| .crl | 证书吊销列表，Certificate Revocation List的缩写 |
| .pem | 用于导出，导入证书时候的证书的格式，Privacy Enhanced Mail，一般为文本格式，以 -----BEGIN... 开头，以 -----END... 结尾，中间的内容是 BASE64 编码。 |

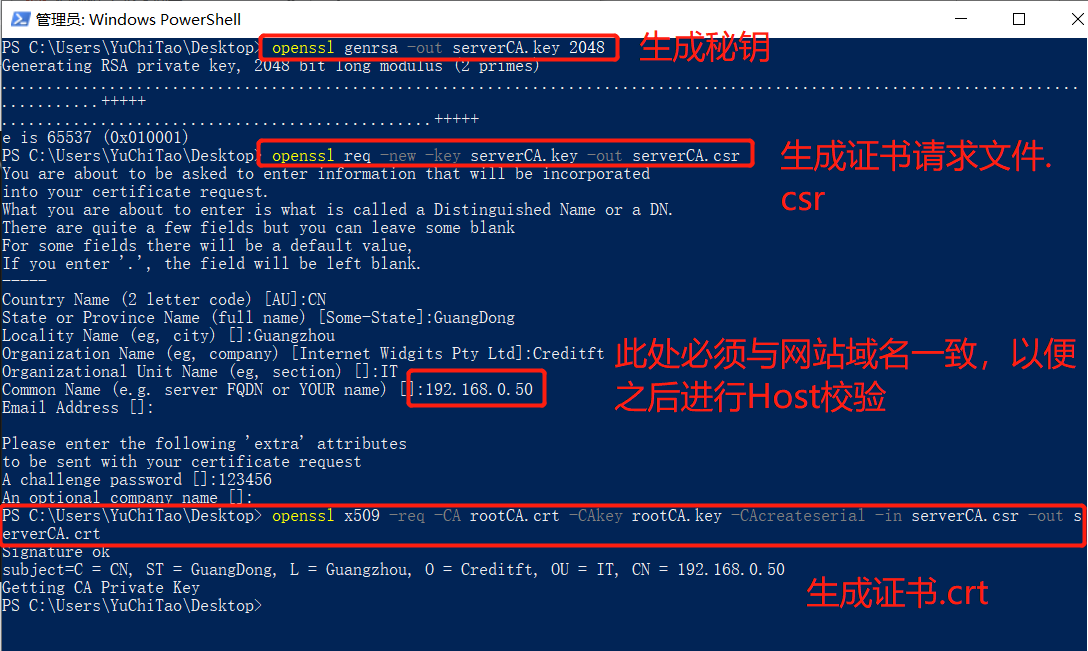
使用OpenSSL生成自签证书流程如下：



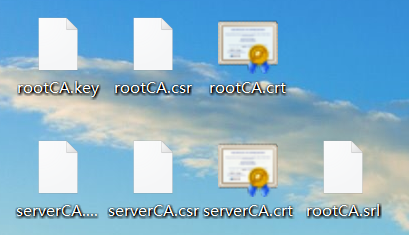
CA根证书的生成：



自签名证书（服务端证书）的生成：



最后一共生成了七个文件：



## 使用SpringBoot搭建简单后台

在实际应用中，证书格式还有许多种，详细描述请参考：

<https://blog.csdn.net/weixin_41863685/article/details/86216586>

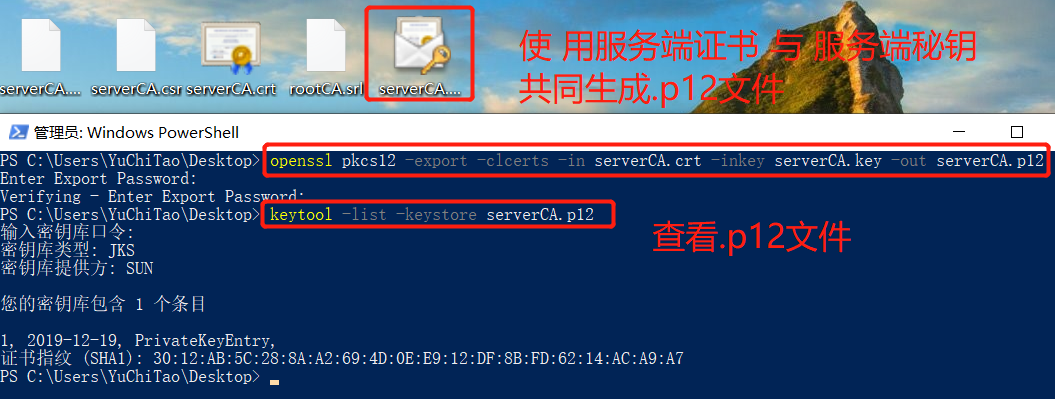
证书格式转换可以使用OpenSSL，也可以在这个网站进行转换：

<https://myssl.com/cert_convert.html>

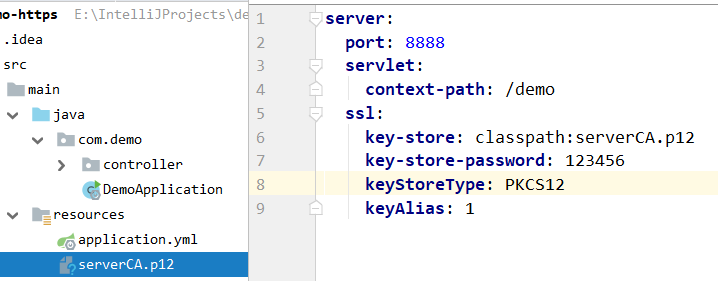
在SpringBoot中使用SSL，可以参考这篇文章

<https://www.jianshu.com/p/eb52e0f5ee85>

首先，将服务端秘钥.key文件（即pem文件）转换为.p12文件（即.pfx文件），p12文件由.crt与.key共同生成：



配置.yml文件：



配置好后我们写一个简单的接口返回字符串“test”，然后用浏览器访问，此时浏览器会提示“不安全”，因为我们的浏览器还没有安装证书。

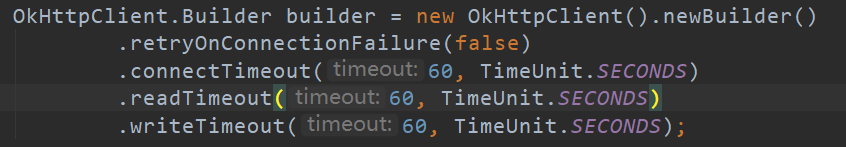


具体浏览器怎么安装证书这里不做研究，接下来我们编写安卓端。

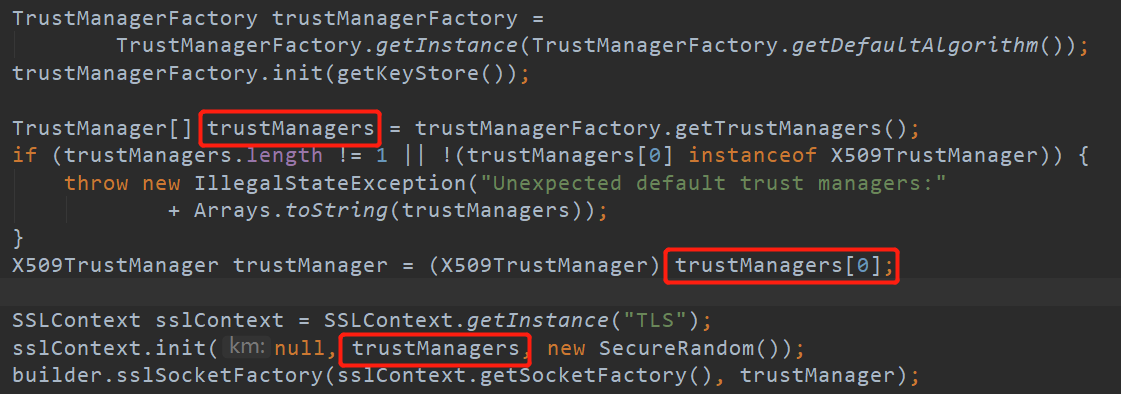
## Android端使用HTTPS访问SpringBoot后台

### 构建OkhttpClient

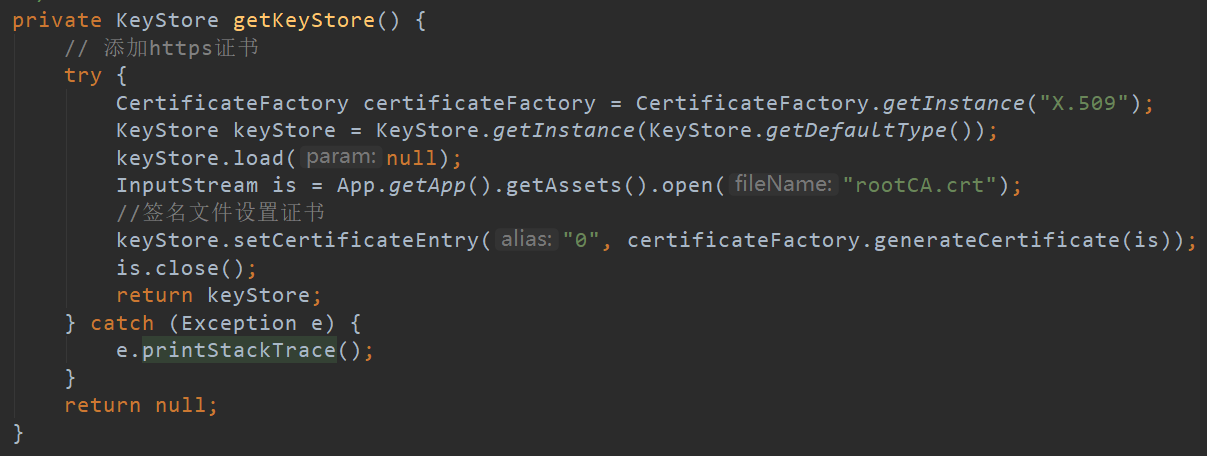
初始化OkhttpClient：



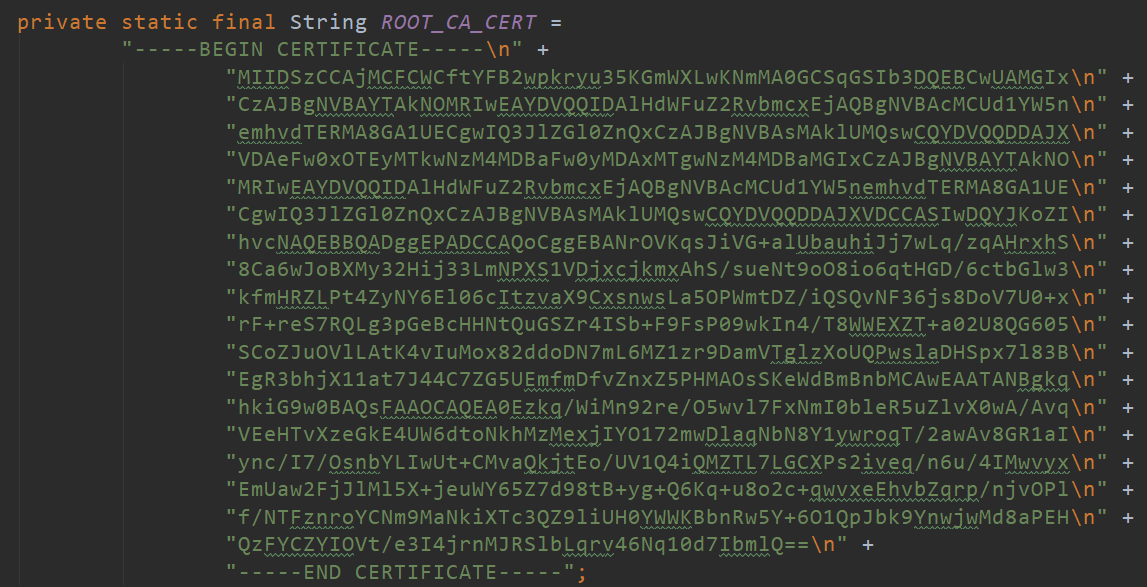
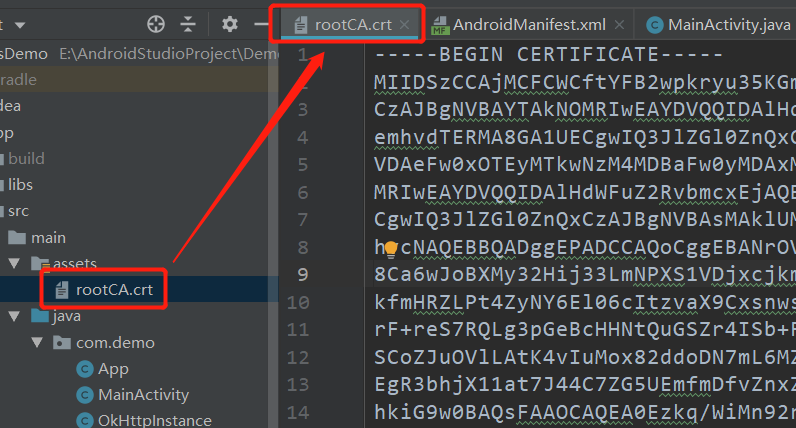
配置TrustManager与SSLContext：



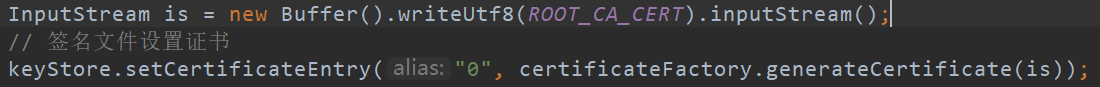
证书导入方法（根证书放在assets目录下，或者也可以写入java文件中）：



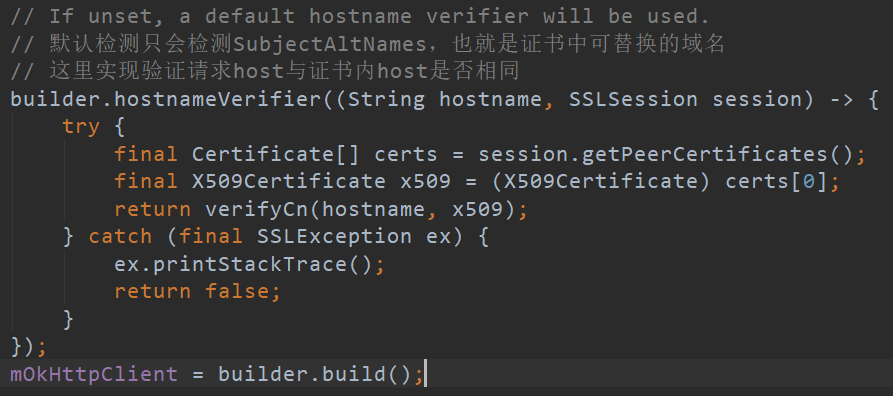
写入java文件的话，打开crt文件，将证书内容复制到代码中：



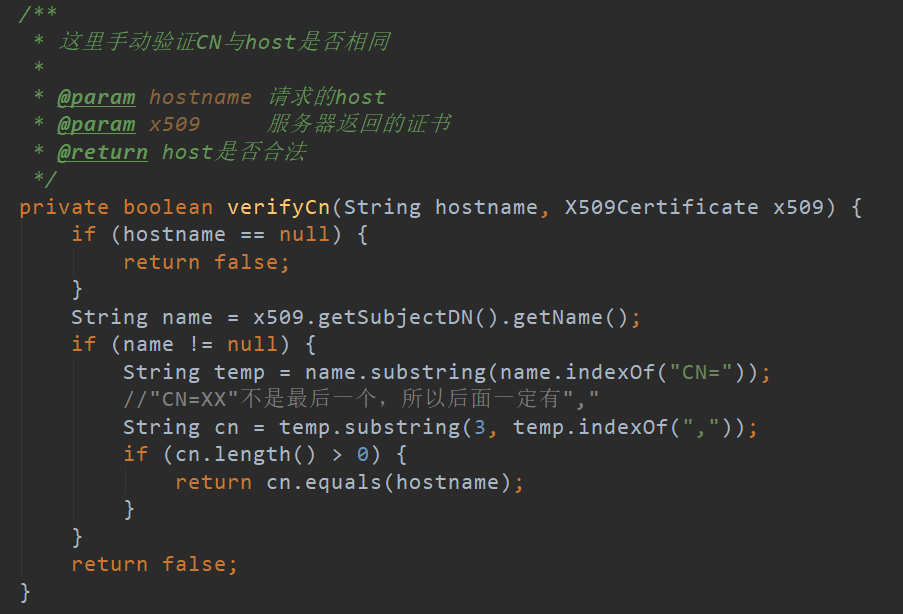
使用okio的Buffer读取字符串为InputString，再设置到KeyStore中：



配置hostnameVerifier，最后构建OkhttpClient：

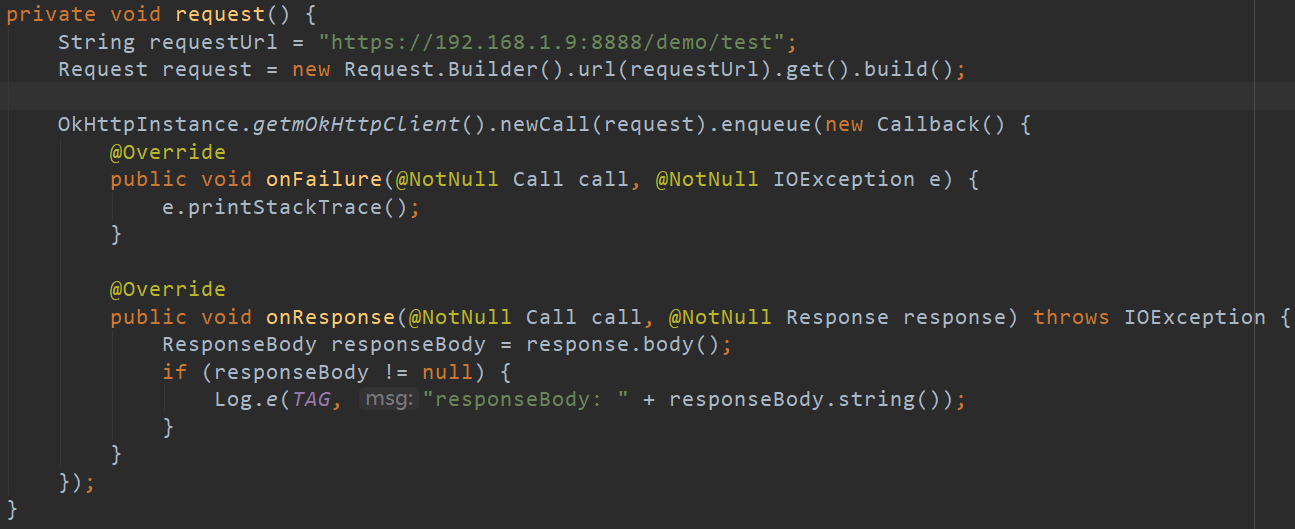


验证证书host与请求host是否相同方法：



这里验证host是验证了证书中的host，也可以在代码中写入固定的host进行验证

### 请求与结果





访问成功，以上便成功完成了客户端请求服务端使用HTTPS的实现过程。

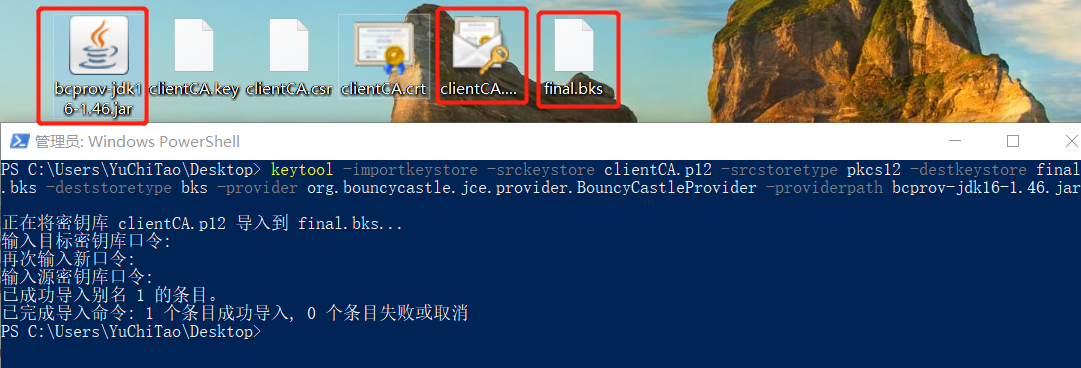
## 双向认证（拓展）

### 生成新证书

上面实现的是单向认证，根证书（rootCA.crt）在客户端，签发证书的秘钥（serverCA.p12）在服务端，是单向认证。实现双向认证，同样的步骤反过来实现就好。但是**Android端有专用的秘钥格式.bks**，需要将.p12转换为.bks。首先从MvnRepository下载转换工具：

<https://mvnrepository.com/artifact/org.bouncycastle/bcprov-jdk16>

然后重新生成一个秘钥，用根证书秘钥签发一个客户端证书，生成.p12文件，再转换为.bks，签发过程参考前面的图片，转换过程如下：

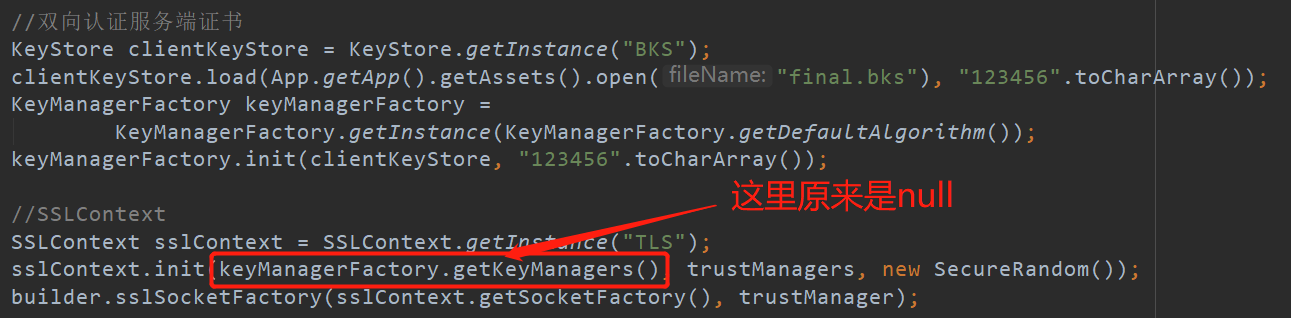


keytool -importkeystore -srckeystore clientCA.p12 -srcstoretype pkcs12 -destkeystore final.bks -deststoretype bks -provider org.bouncycastle.jce.provider.BouncyCastleProvider -providerpath bcprov-jdk16-1.46.jar

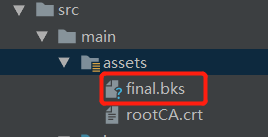
这里clientCA.crt的CN不用写域名，我写成了“Android”。

### 配置Android端

在SSLContext初始化的时候我们将原来的null改成服务端证书，加入以下代码：

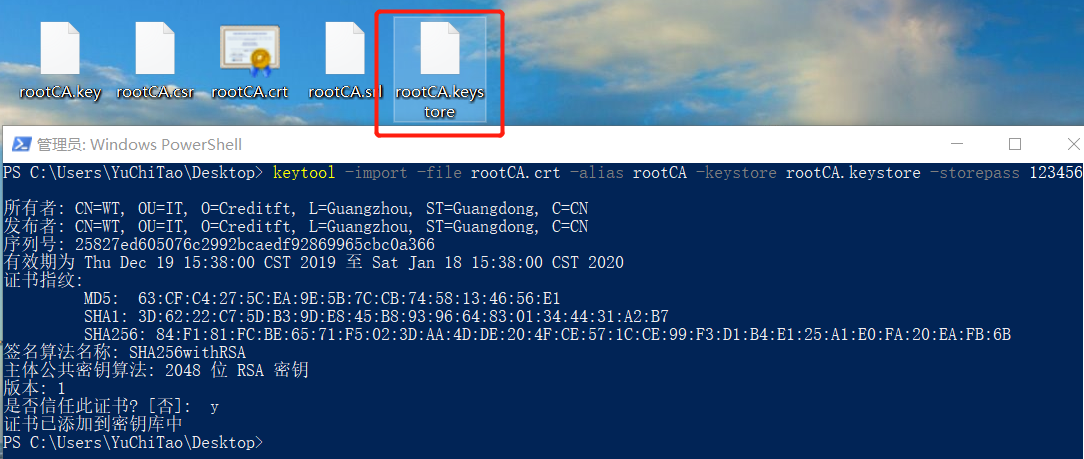


final.bks放在assets文件夹中：



### 转换根证书格式

服务端的配置方式一直出问题，直到我把证书形式转换成.keystore才通过，具体转换方法如下：



keytool -import -file rootCA.crt -alias rootCA -keystore rootCA.keystore -storepass 123456

### 配置服务端

服务端添加以下配置：



至此，我们就可以成功地使用HTTPS的双端认证了。

# 其他问题

## 关于双端证书

双端证书较为复杂，安全系数高，但是要考虑客户端放置的根证书和请求服务端使用的证书的更新。但是一般移动端不会用到双端验证，因为要保护根证书的安全性。

## 关于客户端根证书的更新策略

因为考虑到根证书会过期，所以希望在过期前对证书进行更新，暂时有以下策略：

* 将根证书设置为10年有效期，并每5年更换一次；这算是一种冷处理。
* 在每次APP启动或者用户登录时，获取后台配置的一个状态，该状态说明是否需要更新证书，如果需要，这APP端自动请求某个固定地址进行根证书的下载；这种更新可配置，比较灵活。
* 还有一种就是使用热修复，但是考虑到各个厂家安卓系统的底层代码可能被修改过，维护起来人力成本比较高，所以暂不推荐。