

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

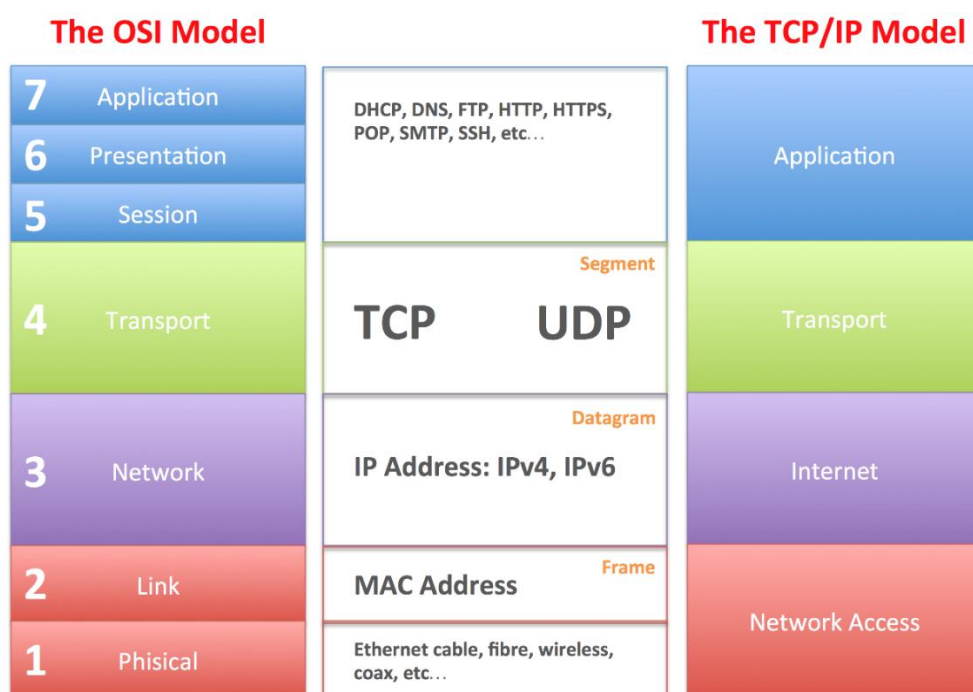
一、 概述.....	3
二、 基础概念.....	3
1. 网络层次.....	3
2. HTTP 的缺陷	3
3. HTTPS 的优势及原理.....	4
4. SSL/TSL 原理.....	4
(1). 加密技术.....	4
(2). 身份验证.....	5
(3). 根证书与证书链	6
(4). CA 证书的使用流程.....	8
(5). SSL/TSL 简介.....	8
三、 实操.....	10
1. 使用 OpenSSL 生成自己的 CA 证书	10
2. 使用 SpringBoot 搭建简单后台.....	12
3. Android 端使用 HTTPS 访问 SpringBoot 后台.....	13
(1). 构建 OkhttpClient.....	13
(2). 请求与结果.....	16
4. 双向认证（拓展）	16
(1). 生成新证书.....	16
(2). 配置 Android 端	17
(3). 转换根证书格式	17
(4). 配置服务端.....	17
四、 其他问题	18
1. 关于双端证书.....	18
2. 关于客户端根证书的更新策略	18
3. HTTPS 为什么不能中间者被解密.....	18

一、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。

二、基础概念

1. 网络层次



This image is part of the Bioinformatics Web Development tutorial at http://www.cellbiol.com/bioinformatics_web_development/ © cellbiol.com, all rights reserved

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)

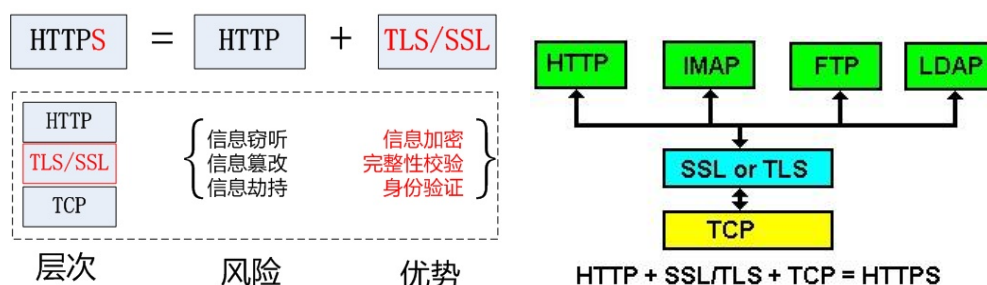
2. HTTP 的缺陷

- 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)。

3. 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端之间的加解密算法协商、密钥交换、通信连接的建立。

4. SSL/TSL 原理

(1).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，如客户端和服务端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。



(2).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**。

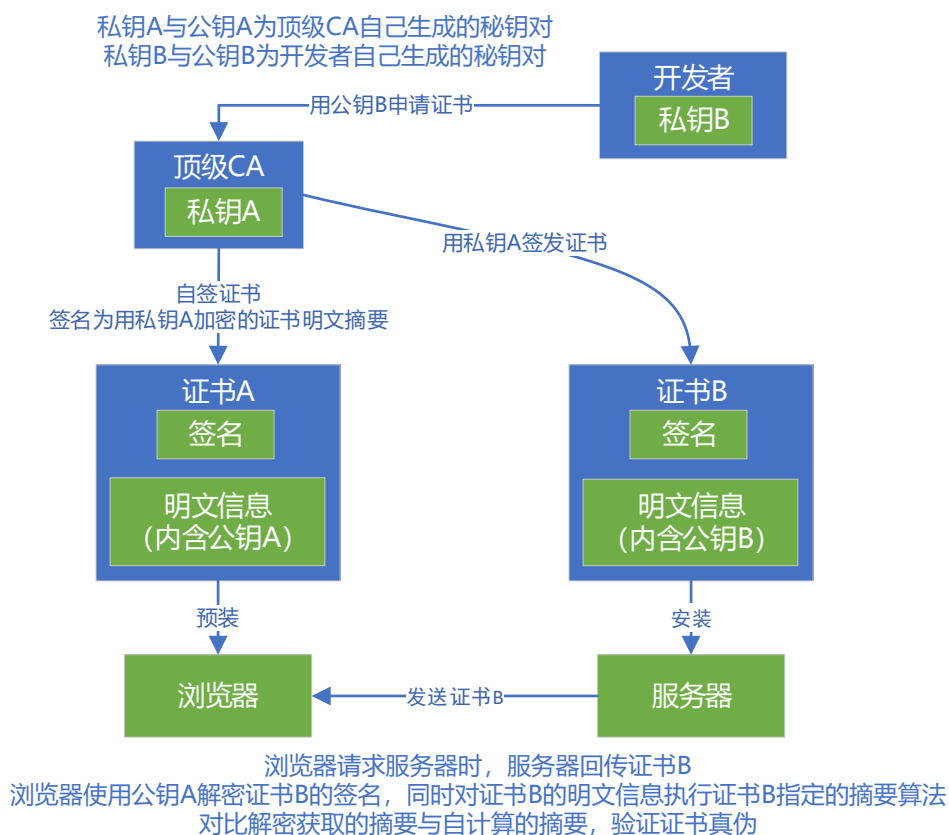
CA 是公钥基础设施 (PKI 即 Public-Key Infrastructure) 之一, PKI 采用证书进行公钥管理,通过第三方的可信任机构 (认证中心,即 CA),把用户的公钥和用户的其他标识信息捆绑在一起,其中包括用户名和电子邮件地址等信息,以在 Internet 网上验证用户的身份。

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**。

(3).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。

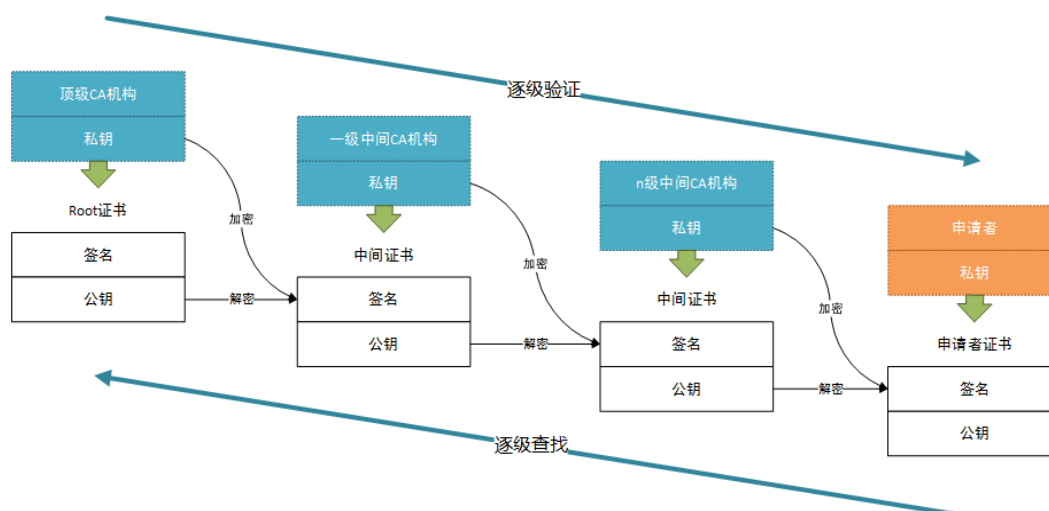


在现实中，有限的几个顶级 CA 机构不可能承担所有组织对与证书的申请，

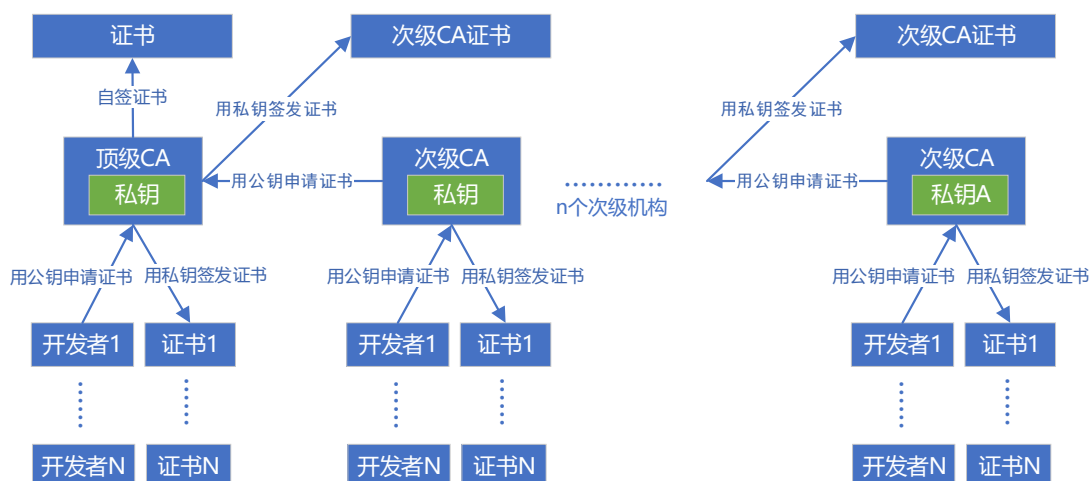
因此顶级机构会授权一些第三方机构作为中间 CA 机构来分担证书颁发的工作 (也可能是其他原因), 这些被授权的中间 CA 机构同样需要向顶级 CA 机构申请一个证书, 用于证明该机构的合法性, 这些机构的证书将与顶级 CA 机构持有的根证书进行绑定, 构成一个证书链的一部分。

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

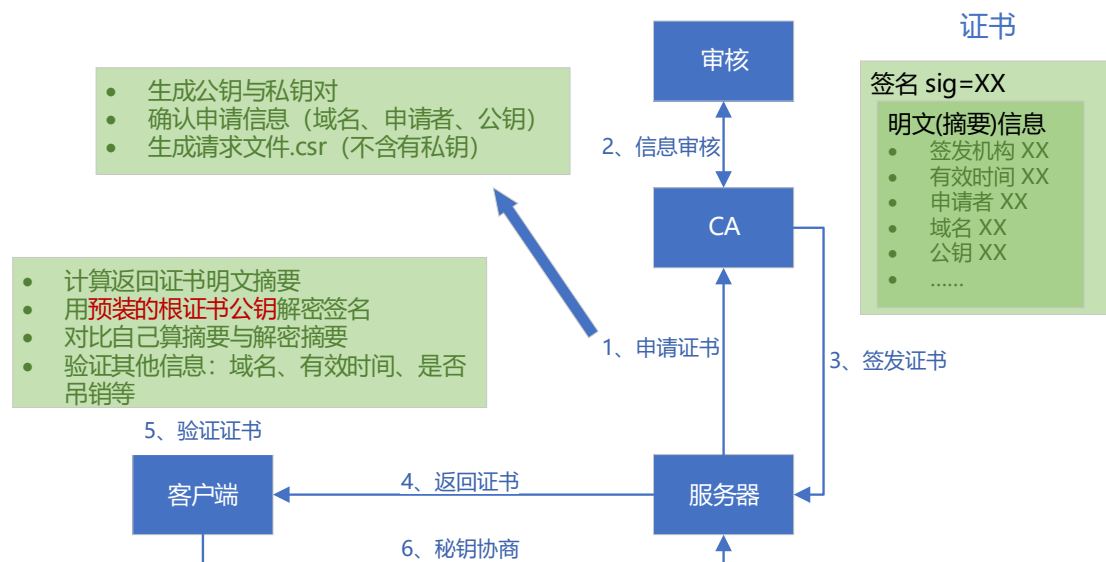
CA 机构证书就是 CA 认证机构持有的证书，而服务证书则是对外发布的证书，仅用于公钥的分发

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

(4).CA 证书的使用流程

CA 证书的申请、使用流程如下：



(5).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>

TCP三次握手				
Source	Destination	Protocol	Length	Info
192.168.0.49	183.3.226.92	TCP	66	9378 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
183.3.226.92	192.168.0.49	TCP	66	443 → 9378 [SYN, ACK] Seq=0 Ack=1 Win=14400 Len=0 MSS=1400 SACK_PERM=1 WS=128
192.168.0.49	183.3.226.92	TCP	54	9378 → 443 [ACK] Seq=1 Ack=1 Win=131584 Len=0
192.168.0.49	183.3.226.92	TLSv1.2	571	Client Hello
183.3.226.92	192.168.0.49	TCP	60	443 → 9378 [ACK] Seq=1 Ack=518 Win=15488 Len=0
183.3.226.92	192.168.0.49	TLSv1.2	1454	Server Hello
183.3.226.92	192.168.0.49	TCP	1454	443 → 9378 [ACK] Seq=1401 Ack=518 Win=15488 Len=1400 [TCP segment of a reassembled PDU]
183.3.226.92	192.168.0.49	TLSv1.2	1454	Certificate [TCP segment of a reassembled PDU]
183.3.226.92	192.168.0.49	TLSv1.2	66	Server Key Exchange, Server Hello Done
192.168.0.49	183.3.226.92	TCP	54	9378 → 443 [ACK] Seq=518 Ack=2801 Win=131584 Len=0
192.168.0.49	183.3.226.92	TCP	54	9378 → 443 [ACK] Seq=518 Ack=4213 Win=131584 Len=0
192.168.0.49	183.3.226.92	TLSv1.2	180	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
192.168.0.49	183.3.226.92	TCP	1454	9378 → 443 [ACK] Seq=644 Ack=4213 Win=131584 Len=1400 [TCP segment of a reassembled PDU]
192.168.0.49	183.3.226.92	TLSv1.2	695	Application Data
183.3.226.92	192.168.0.49	TCP	60	443 → 9378 [ACK] Seq=4213 Ack=2044 Win=18304 Len=0
183.3.226.92	192.168.0.49	TLSv1.2	312	New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
183.3.226.92	192.168.0.49	TLSv1.2	317	Application Data
183.3.226.92	192.168.0.49	TLSv1.2	88	Application Data
192.168.0.49	183.3.226.92	TCP	54	9378 → 443 [ACK] Seq=2685 Ack=4734 Win=131072 Len=0
192.168.0.49	183.3.226.92	TCP	54	9378 → 443 [ACK] Seq=2685 Ack=4768 Win=130816 Len=0

131	4.020799	192.168.0.49	183.3.226.92	TCP	54	9378 → 443 [ACK] Seq=1 Ack=1
132	4.021167	192.168.0.49	183.3.226.92	TLSv1.2	571	Client Hello
133	4.037669	183.3.226.92	192.168.0.49	TCP	60	443 → 9378 [ACK] Seq=1 Ack=51

> Frame 132: 571 bytes on wire (4568 bits), 571 bytes captured (4568 bits) on interface 0
> Ethernet II, Src: RivetNet_f2:8f:d7 (9c:b6:d0:f2:8f:d7), Dst: Tp-LinkT_bb:06:f7 (54:75:95:bb:06:f7)
> Internet Protocol Version 4, Src: 192.168.0.49, Dst: 183.3.226.92
> Transmission Control Protocol, Src Port: 9378, Dst Port: 443, Seq: 1, Ack: 1, Len: 517
▼ Transport Layer Security

▼ TLSv1.2 Record Layer: Handshake Protocol: Client Hello 握手协议
Content Type: Handshake (22)
Version: TLS 1.0 (0x0301)
Length: 512

▼ Handshake Protocol: Client Hello
Handshake Type: Client Hello (1)
Length: 508
Version: TLS 1.2 (0x0303)
> Random: f8cf407115a8d66553f1ddf2f8204df9b17e89ba79096bf2...
Session ID Length: 32
Session ID: a2e370045daa9f76473c4a1ce473109374e5dbdc44dee49f...
Cipher Suites Length: 34
> Cipher Suites (17 suites)
Compression Methods Length: 1
> Compression Methods (1 method)
Extensions Length: 401
> Extension: Reserved (GREASE) (len=0)

协议具体内容

三、实操

1. 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)

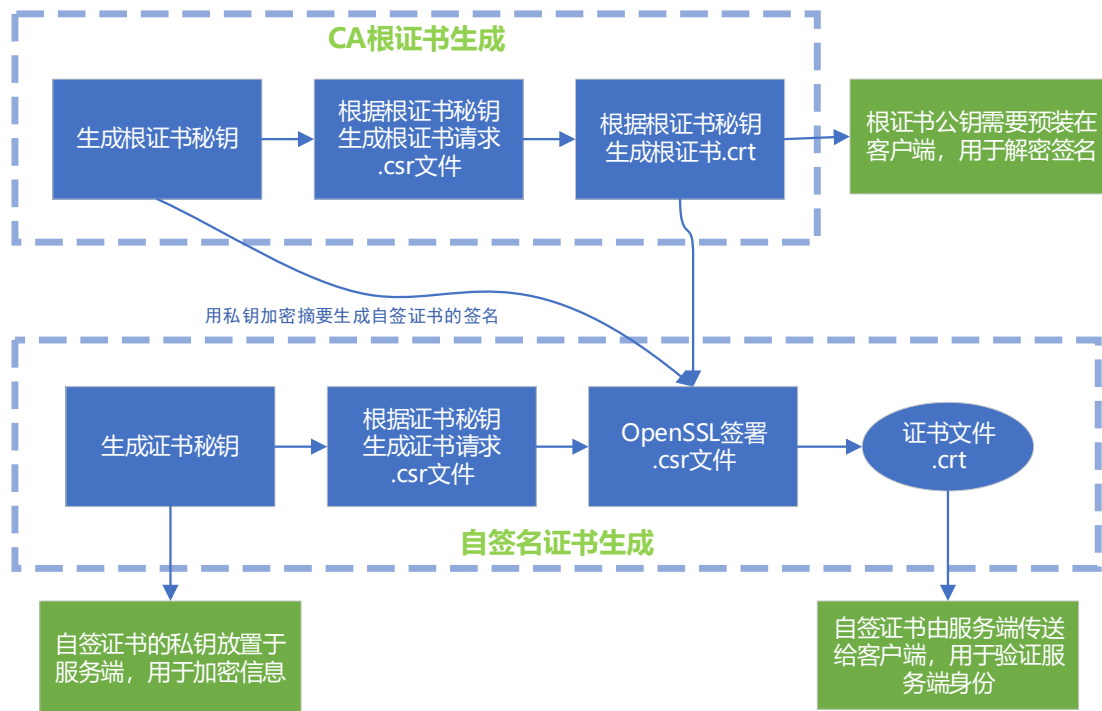
<https://www.jianshu.com/p/37ded4da1095>

<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:

```

管理员: Windows PowerShell
PS C:\Users\YuChiTao\Desktop> openssl genrsa -out rootCA.key 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
PS C:\Users\YuChiTao\Desktop> openssl req -new -key rootCA.key -out rootCA.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CN
State or Province Name (full name) [Some-State]:Guangdong
Locality Name (eg, city) []:Guangzhou
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Creditft
Organizational Unit Name (eg, section) []:IT
Common Name (e.g. server FQDN or YOUR name) []:WT
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:123456
An optional company name []:
PS C:\Users\YuChiTao\Desktop> openssl x509 -req -in rootCA.csr -signkey rootCA.key -out rootCA.crt
Signature ok
subject=C = CN, ST = Guangdong, L = Guangzhou, O = Creditft, OU = IT, CN = WT
Getting Private key
  
```

生成秘钥 (Generate Private Key)

生成请求文件.csr (Generate Request File .csr)

生成证书.crt (Generate Certificate .crt)

自签名证书（服务端证书）的生成:

```
管理员: Windows PowerShell
PS C:\Users\YuChiTao\Desktop> openssl genrsa -out serverCA.key 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
PS C:\Users\YuChiTao\Desktop> openssl req -new -key serverCA.key -out serverCA.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CN
State or Province Name (full name) [Some-State]:GuangDong
Locality Name (eg, city) []:Guangzhou
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Creditft
Organizational Unit Name (eg, section) []:IT
Common Name (e.g. server FQDN or YOUR name) []:192.168.0.50
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:123456
An optional company name []:
PS C:\Users\YuChiTao\Desktop> openssl x509 -req -CA rootCA.crt -CAkey rootCA.key -CAcreateserial -in serverCA.csr -out s
erverCA.crt
Signature ok
subject=C = CN, ST = Guangdong, L = Guangzhou, O = Creditft, OU = IT, CN = 192.168.0.50
Getting CA Private Key
PS C:\Users\YuChiTao\Desktop>
```

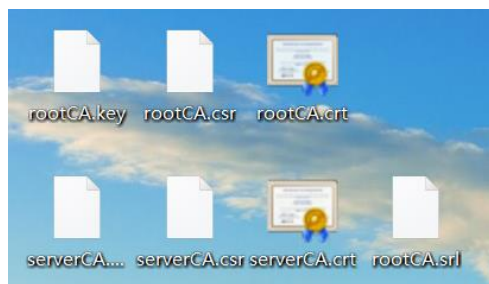
生成秘钥

生成证书请求文件.
csr

此处必须与网站域名一致，以便
之后进行Host校验

生成证书.crt

最后一共生成了七个文件：



2. 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

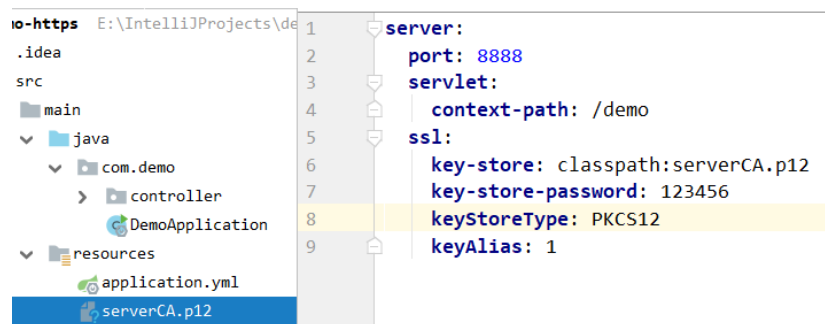
在 SpringBoot 中使用 SSL，可以参考这篇文章

<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。

3. Android 端使用 HTTPS 访问 SpringBoot 后台

(1).构建 OkHttpClient

初始化 OkHttpClient:

```
OkHttpClient.Builder builder = new OkHttpClient().newBuilder()
    .retryOnConnectionFailure(false)
    .connectTimeout(timeout: 60, TimeUnit.SECONDS)
    .readTimeout(timeout: 60, TimeUnit.SECONDS)
    .writeTimeout(timeout: 60, TimeUnit.SECONDS);
```

配置 TrustManager 与 SSLContext:

```

TrustManagerFactory trustManagerFactory =
    TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
trustManagerFactory.init(getKeyStore());

TrustManager[] trustManagers = trustManagerFactory.getTrustManagers();
if (trustManagers.length != 1 || !(trustManagers[0] instanceof X509TrustManager)) {
    throw new IllegalStateException("Unexpected default trust managers:"
        + Arrays.toString(trustManagers));
}
X509TrustManager trustManager = (X509TrustManager) trustManagers[0];

SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init( km: null, trustManagers, new SecureRandom());
builder.sslSocketFactory(sslContext.getSocketFactory(), trustManager);

```

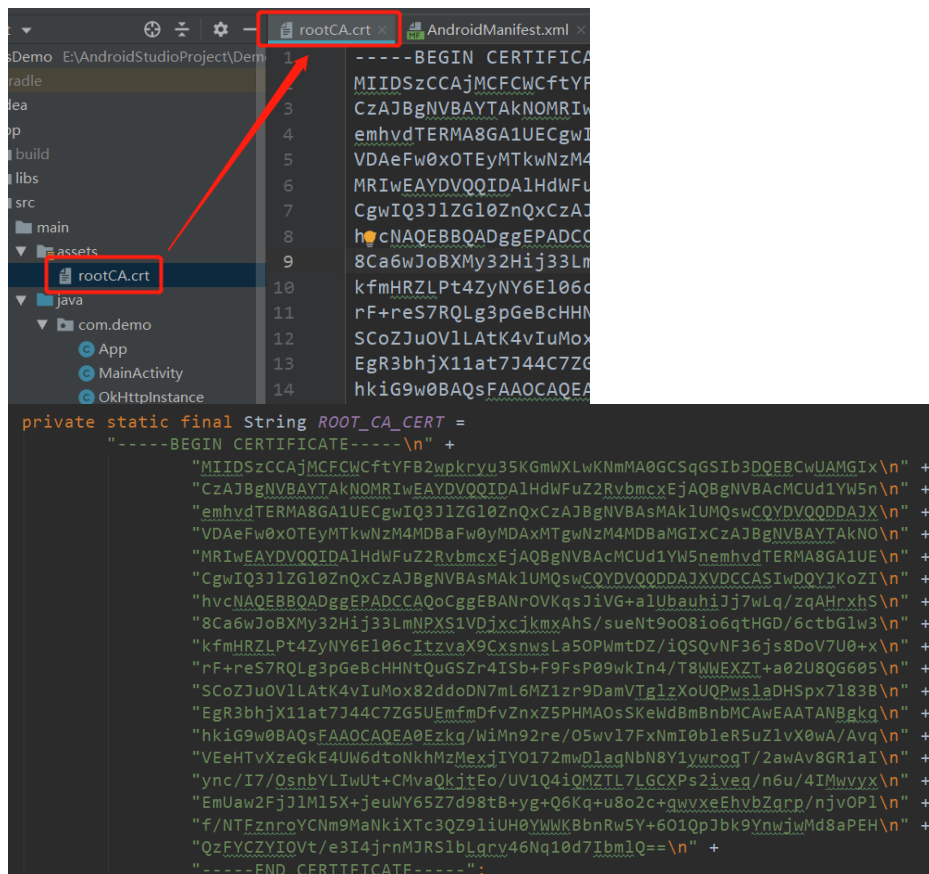
证书导入方法（根证书放在 assets 目录下，或者也可以写入 java 文件中）：

```

private KeyStore getKeyStore() {
    // 添加https证书
    try {
        CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
        KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
        keyStore.load( param: null);
        InputStream is = App.getApp().getAssets().open( fileName: "rootCA.crt");
        // 签名文件设置证书
        keyStore.setCertificateEntry( alias: "0", certificateFactory.generateCertificate(is));
        is.close();
        return keyStore;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}

```

写入 java 文件的话，打开 crt 文件，将证书内容复制到代码中：



```

private static final String ROOT_CA_CERT =
    "-----BEGIN CERTIFICATE-----\n" +
    "MIIDSzCCAjMCFCWCftYFB2wpryu35KGmWLWKNMMA0GCSqGSIb3DQEBCwUAMGIX\n" +
    "CzAIBgNVBAYTAkNOMRIwEAYDVQQIDAlHdWZ2R2bmcxEjAQBgNVBACMCud1YW5n\n" +
    "emhvdTERMA8GA1UECgwIQ3JlZG10ZnQxCzAIBgNVBAsMAklUMQswCQYDVQDDAJX\n" +
    "VDAeFw0xOTEyMTkwNzQ0MDBAFw0yMDA0MTkwNzQ0MDBAIGIxMAk1UEBAYTAkNO\n" +
    "MRIwEAYDVQQIDAlHdWZ2R2bmcxEjAQBgNVBACMCud1YW5nemhvdTERMA8GA1UE\n" +
    "CgwIQ3JlZG10ZnQxCzAIBgNVBAsMAklUMQswCQYDVQDDAJXVDCCAStwDQYJKoZI\n" +
    "hvcNAQEBBQADggEPADCCAQoCggEBANrOVKqsJiVG+alUBauhi7j7wLq/zqAhrxhS\n" +
    "8Ca6wJoBXM32Hij33LmNPXS1VDjxcjkmXAhS/sueNt9o08io6qtHGd/6ctbG1w3\n" +
    "kfmHRZLPt4ZyNY6E106cItzvaX9CXsnwSLa50PWmtDZ/iQSQvNF36js8DoV7U0+x\n" +
    "rF+reS7RQLg3pGeBcHHNTQuGSZr4ISb+F9FsP09wkIn4/T8WwEXZI+a02U8QG605\n" +
    "SCoZJuOV1LAtK4vIuMox82ddoDN7mL6MZ1zr9DamVTglzXoUQPwsladHSpx7183B\n" +
    "EgR3bhjX11at7J44C7ZG5UEfmDfvZnxZ5PHMA0sSKewdBmBnbMCawEAATANBgkq\n" +
    "hkiG9w0BAQsFAAQCAQEAEZkg/WiMn92re/O5wv17FxNmI0bleR5uZ1vX0wA/Avq\n" +
    "VEeHTVxzeGkE4UW6dtoNkhMzMexjIYO172mwDlagNbN8Y1ywrqT/2awAv8GR1aI\n" +
    "ync/I7/QsnbYLIwUt+CMvaQKjtEo/UV1Q4iQMZTL7LGCXPs2iveq/n6u/4IMwvYx\n" +
    "EmUaw2Fj1M15X+jeuWY65Z7d98tB+yg+Q6Kq+u8o2c+qwwxeEhvbZgrp/njvOP1\n" +
    "f/NTFznroYCNm9MaNkiXTc3QZ91iUH0YWWKBbnRw5Y+601QpJbk9YnwjwMd8aPEH\n" +
    "QzFYCZYIQvt/e3I4jrnM3RS1bLgrv46Nq10d7Ibm1Qe=\n" +
    "-----END CERTIFICATE-----";

```

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

```
InputStream is = new Buffer().writeUtf8(ROOT_CA_CERT).inputStream();  
// 签名文件设置证书  
keyStore.setCertificateEntry(alias: "0", certificateFactory.generateCertificate(is));
```

配置 hostnameVerifier, 最后构建 OkHttpClient:

```
// If unset, a default hostname verifier will be used.  
// 默认检测只会检测SubjectAltNames, 也就是证书中可替换的域名  
// 这里实现验证请求host与证书内host是否相同  
builder.hostnameVerifier((String hostname, SSLSession session) -> {  
    try {  
        final Certificate[] certs = session.getPeerCertificates();  
        final X509Certificate x509 = (X509Certificate) certs[0];  
        return verifyCn(hostname, x509);  
    } catch (final SSLException ex) {  
        ex.printStackTrace();  
        return false;  
    }  
});  
mOkHttpClient = builder.build();
```

验证证书 host 与请求 host 是否相同方法:

```
/**  
 * 这里手动验证CN与host是否相同  
 *  
 * @param hostname 请求的host  
 * @param x509 服务器返回的证书  
 * @return host是否合法  
 */  
private boolean verifyCn(String hostname, X509Certificate x509) {  
    if (hostname == null) {  
        return false;  
    }  
    String name = x509.getSubjectDN().getName();  
    if (name != null) {  
        String temp = name.substring(name.indexOf("CN="));  
        // "CN=XX"不是最后一个, 所以后面一定有","  
        String cn = temp.substring(3, temp.indexOf(","));  
        if (cn.length() > 0) {  
            return cn.equals(hostname);  
        }  
    }  
    return false;  
}
```

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

(2).请求与结果

```
private void request() {
    String requestUrl = "https://192.168.1.9:8888/demo/test";
    Request request = new Request.Builder().url(requestUrl).get().build();

    OkHttpClient.getInstance().newCall(request).enqueue(new Callback() {
        @Override
        public void onFailure(@NotNull Call call, @NotNull IOException e) {
            e.printStackTrace();
        }

        @Override
        public void onResponse(@NotNull Call call, @NotNull Response response) throws IOException {
            ResponseBody responseBody = response.body();
            if (responseBody != null) {
                Log.e(TAG, "responseBody: " + responseBody.string());
            }
        }
    });
}
```

2019-12-20 00:51:14.307 27453-28868/com.demo E/MainActivity: responseBody: test

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。

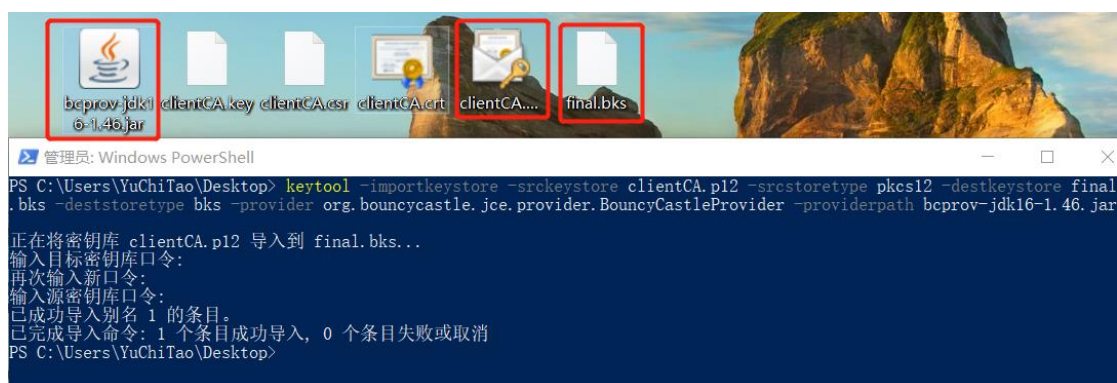
4. 双向认证（拓展）

(1).生成新证书

上面实现的是单向认证，根证书（rootCA.crt）在客户端，签发证书的密钥（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 端有专用的密钥格式.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>

然后重新生成一个密钥，用根证书密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”。

(2).配置 Android 端

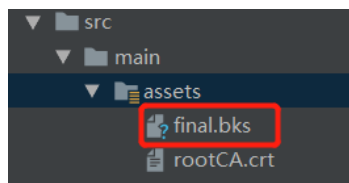
在 SSLContext 初始化的时候我们将原来的 null 改成服务端证书，加入以下代码：

```
//双向认证服务端证书
KeyStore clientKeyStore = KeyStore.getInstance("BKS");
clientKeyStore.load(App.getApp().getAssets().open(fileName: "final.bks"), "123456".toCharArray());
KeyManagerFactory keyManagerFactory =
    KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
keyManagerFactory.init(clientKeyStore, "123456".toCharArray());

//SSLContext
SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(keyManagerFactory.getKeyManagers(), trustManagers, new SecureRandom());
builder.sslSocketFactory(sslContext.getSocketFactory(), trustManager);
```

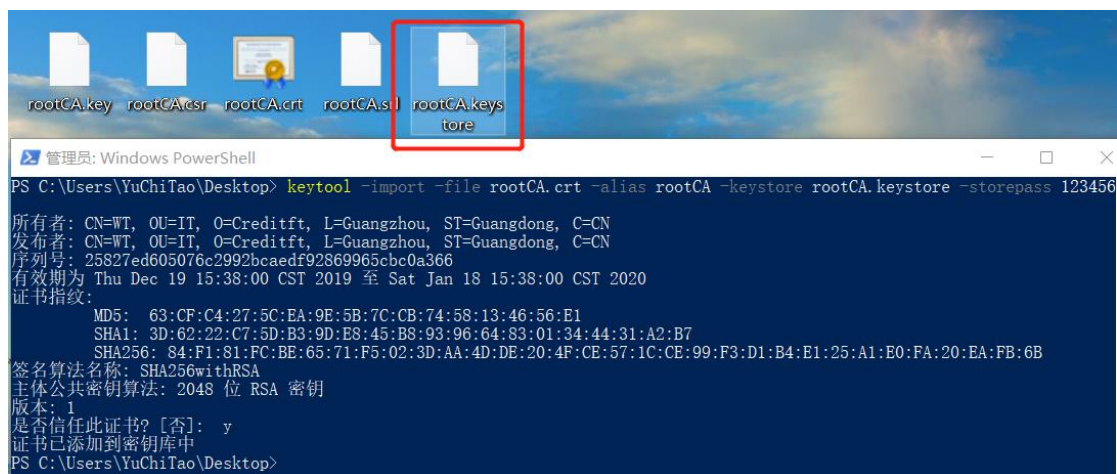
这里原来是null

final.bks 放在 assets 文件夹中：



(3).转换根证书格式

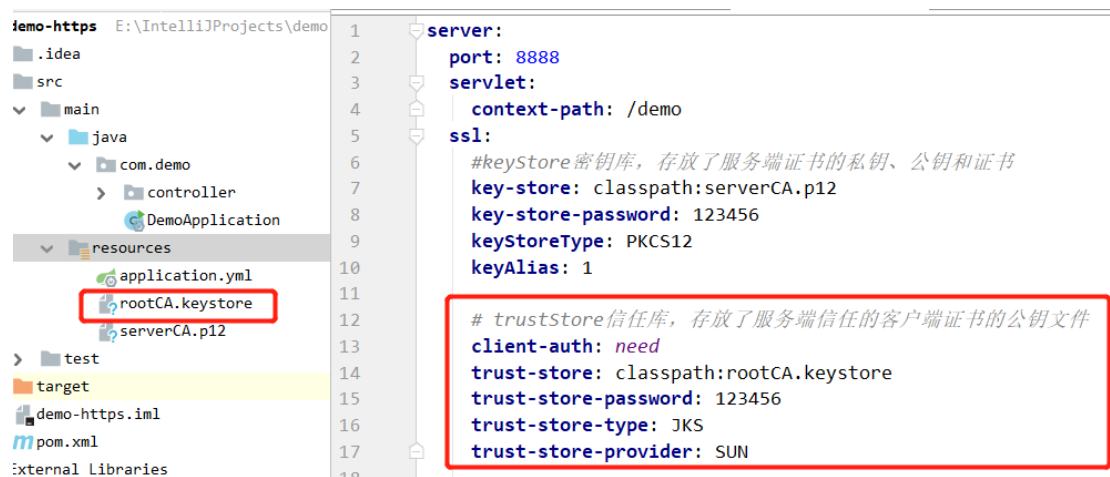
服务端的配置方式一直出问题，直到我把证书形式转换成.keystore 才通过，具体转换方法如下：



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

(4).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。

四、其他问题

1. 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。

2. 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。

3. HTTPS 为什么不能中间者被解密

这是一个补充，写完这个文档后，我在想，如果有一个中间者，通过某些方法获得客户端公钥（比如反编译.apk 包），然后作为中间者进行攻击，是否可行。

下面这个图就很好地解答了我的问题，这是一个双向验证，其中，对于第三个随机数，只有服务端有私钥，其他人都获取不了，则客户端发送的这个信息，中间者无法获得；那么，如果中间者自己制造第三个参数呢？那就会导致在最后

测试加密的时候，客户端和中间者的秘文对不上。

