这是道哥的第012篇原创

关注+星标公众号，不错过最新文章

前言
一个典型的物联网产品
数据加密
    明文传输的缺点
    加密传输
    加密方式
        可逆加密
        不可逆加密
        公钥和私钥
证书
    如何申请证书
    如何确认证书的合法性
        单向认证和双向认证
    认证机构
    证书链
    证书文件的后缀名
    证书文件的格式
        PEM格式(Privacy Enhanced Mail)
        DER格式(Distinguished Encoding Rules)
        X.509标准
        证书格式
    OpenPGP协议/标准
        OpenPGP是什么?
        OpenPGP协议的实现
        OpenPGP的使用流程
SSL/TLS
    协议分层
    握手过程
    HTTPS与SSL的关系
OpenSSL
    OpenSSL是什么?
    密码算法库
    信息摘要算法
    秘钥和证书管理
    SSL协议库
    应用程序
OpenSSH又是什么?
    SSH
    SSH中基于口令的安全验证
    SSH中基于秘钥的安全认证

SSH协议的实现
总结

前言

在一个物联网系统中，终端设备在连接云平台(服务器)的时候，云平台需要对设备的身份进行验证，验证这是一个合法的设备之后才允许接入。这看似很简单的一句话，背后包含了很多相关的概念，例如：加密、证书、证书标准、签名、认证机构、SSL/TLS、OpenSSL、握手等一堆容易混淆的概念。

之前我在做智能家居项目时，每次遇到证书以及加密的问题时，都是满大街的查资料，但是由于每次都是解决问题之后就停止下来，没有进行完整、系统的梳理，因此对这些概念始终感觉自己都理解了，但是又说不出所以然来。

这篇文章我们就把这些概念以及相关的使用步骤进行梳理，就像联想记忆一样，很多分散的东西总是记不住，但是如果把这些东西按照特定的关系组织在一起，那么记忆起来就非常容易了。

做个小游戏：在1分钟内记下这十个东西：茶杯、猴子、玻璃、垃圾桶、鱼竿、鸟窝、和尚、汽车、医院、饮水机。这里可以暂停一下，看看自己的记忆力是不是不如以前了。

我们再换个记忆方法，把这十个东西以任意荒诞的逻辑联系在一起，比如：一只猴子，左手拿着茶杯，右手拿着玻璃，往垃圾桶走去。在垃圾桶旁边，看到一只鱼竿，于是它就用鱼竿去戳树上的鸟窝。鸟窝里掉下来一个鸟蛋，正好砸在了和尚的头上，流血了，赶紧拦下一辆汽车去医院。到了医院，和尚失血太多口渴了，正好看到一台饮水机。

把这个荒诞的故事想几遍，然后再试着把这十个东西说出来，这回是不是感觉很容易？而且连顺序都不会记错！这就是联想记忆的魔力。

那么学习知识也是这个道理：分散的知识是记不住的，只有梳理成体系，把相互之间的联系和脉络掌握了，再去理解这些分散的点就很容易记住了。这里的关键就是把这些知识点相互之间的关系掌握了，就像一张网一样，随便把一个知识点拎出来，都可以根据这张网把其他的知识点联想起来。

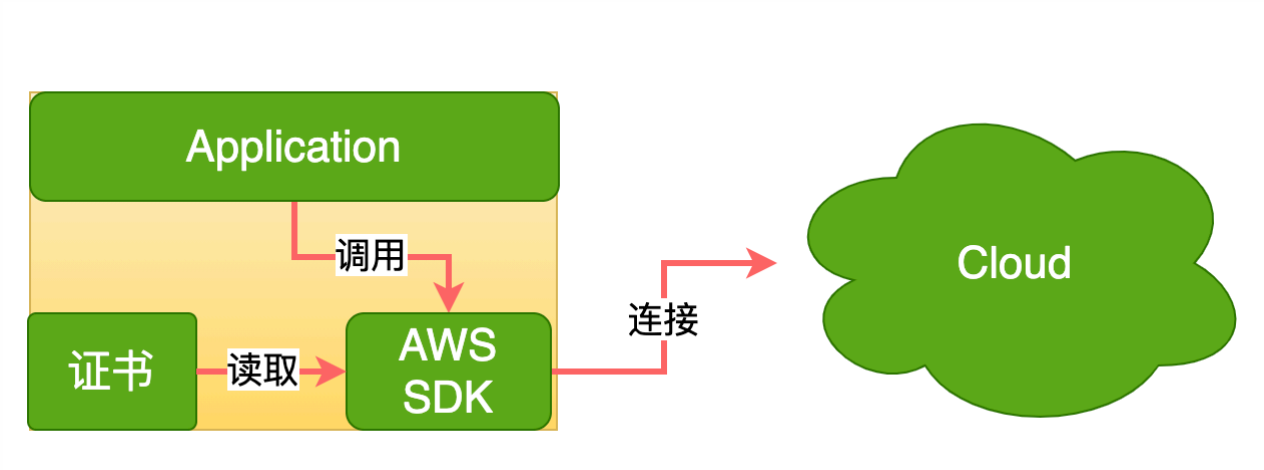
这篇文章的内容包括：

> 物联网云平台，是如何验证设备端的合法性？
> SSL/TLS是什么？有什么作用？在哪些场合下使用？
> OpenSSL是什么？它与SSL是什么关系？
> OpenSSH又是什么？它与OpenSSL又有什么区别？
> HTTPS中是如何利用SSL来交换秘钥的？握手步骤是什么？
> 证书是什么？有什么作用？在哪些场合下使用？
> 证书是如何得到的？它的标准格式是什么？包含哪些内容？
> 认证机构是什么？什么是链式证书？
> 证书与SSL有什么关系？
> 签名是什么意思？与加密是什么关系？
> 什么是单向认证？双向认证？

另外补充一点：这篇文章只描述“是什么”，而不会描述“为什么”。“为什么”的事情留给那些数学家、密码学家来搞定就可以了。

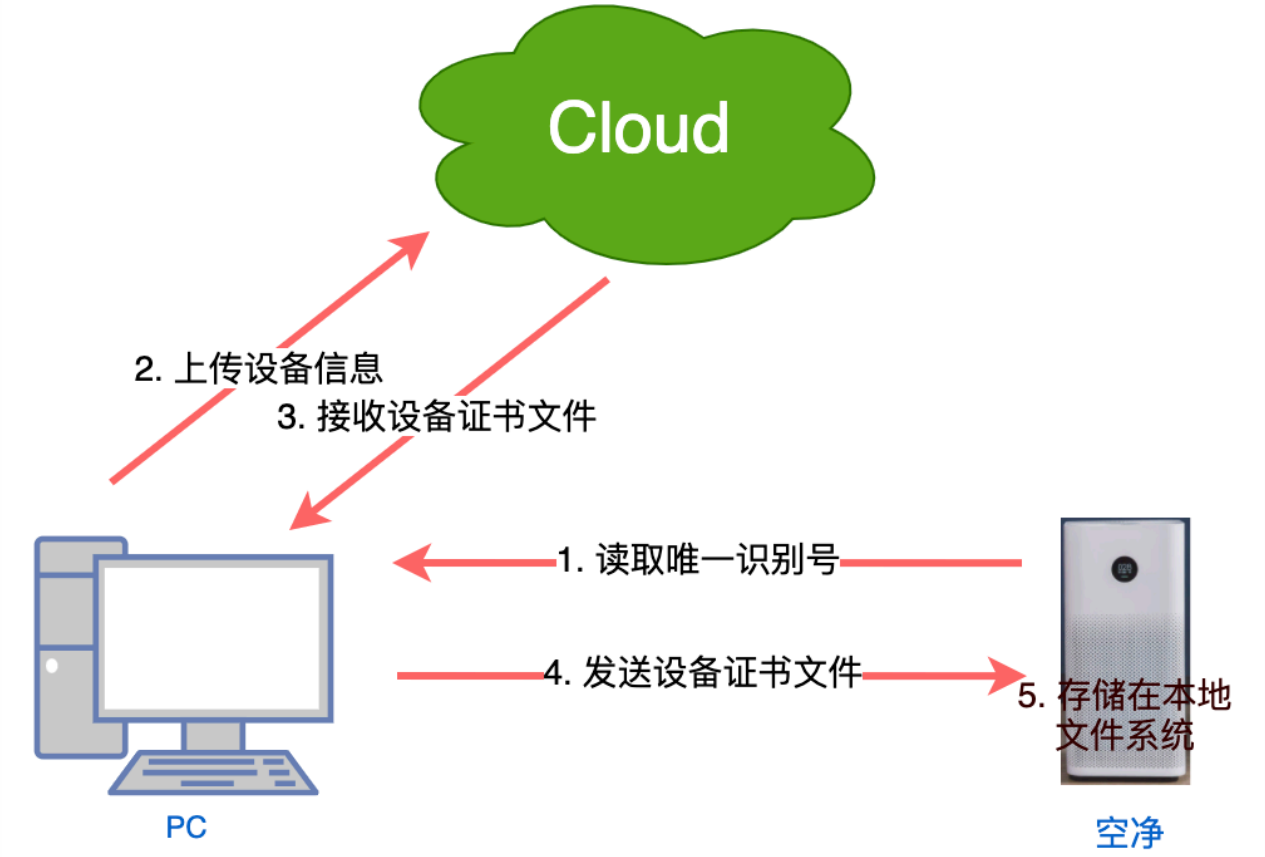
一个典型的物联网产品

在实际的项目中，如果用到云平台，一般来说选择性就那么几个：国外就用亚马逊，国内就用阿里云，最近也碰到一些项目使用华为云。这里就以之前做过的一个空气净化器项目来举例：



首先，亚马逊提供了一套SDK，这个SDK中包含了一组API函数供应用程序调用，向云平台进行安全连接、收发数据。在调用API函数的时候，必须提供一些必要的设备信息，这其中最重要的就是设备证书文件，也就是说，证书必须要预先存储在设备的文件系统中。

那么，证书是在什么时候被放到空气净化器设备中的？当然是生产阶段，看一下这个流程：



1. 生产工具软件运行在产线的PC机上，通过串口连接到空气净化器设备，从设备中读取唯一识别码(例如：MAC地址);
2. 生产工具软件上传必要的信息(厂商基本信息、厂商秘钥、空净设备的唯一识别码)给AWS云平台，申请得到一个证书文件;
3. AWS云平台根据厂商预先在平台上的部署程序，产生一个证书文件，返回给生产工具软件;
4. 生产工具软件把证书文件通过串口发送给空气净化器设备;
5. 空气净化器设备接收到的证书文件之后，存储到本地的文件系统中。

以上这个流程是在设备生产环节完成的，这里的描述还是属于粗线条的，其他一些重要的信息没有列出来，比如：AWS后台如何产生证书、在连接阶段后台是如何通过证书来验证设备的合法性的、厂商的秘钥是如何工作的等等，这些问题等到这边文章的末尾就自然明白了。

下面，就按照这些概念之间的相互关系来一步一步的梳理，每一个概念是按照相互之间的关系来逐步引入的，因此建议按照顺序来理解。

数据加密

明文传输的缺点

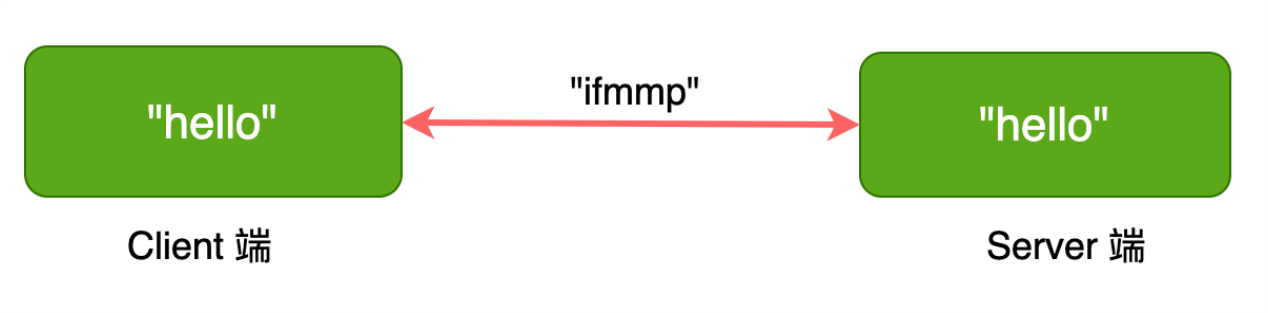
我们知道，client端与server端之间传输数据，要么是明文传输、要么是加密传输。明文传输的缺点显而易见：

> 数据容易比第三方截获；
> 第三方可以篡改数据；
> 第三方可能会冒充server与你进行通信。

总之一句话：明文通信就像裸奔一样，任何东西都被别人看的一清二楚，恶意的第三方很容易利用明文通信来做一些违法的事情。

所以，最好还是穿上衣服，最好还是带密码锁的，这样别人就看不到了！这就是加密传输。

加密传输



client端对传输的信息进行加密，server端接收到密文后再进行解密。例如上图中：

> client想发送字符串"hello"，那么就先加密成"ifmmp"，然后发送出去；
> server接收到"ifmmp"，进行解密，得到"hello"。

但是示例中的加密方式太弱智了，稍微研究下就会搞明白，这里的加密方式就是把明文字符串中的每一个字符变成ascII码表中的下一个字。server在解密时操作相反：把每一个字符变成ascII码表中的前一个字符即可，只要client和server事先商量好这样的加密和解密算法就可以通信了。

但是，这样的加密方式太简单了，恶意的第三方不会吹灰之力就可以破解出来，因此client与server之间需要更加复杂的加密算法，这就是SSL要解决的问题，这部分内容稍后再表。

加密方式

根据是否可以把密文还原成明文，加密方式分为两类：

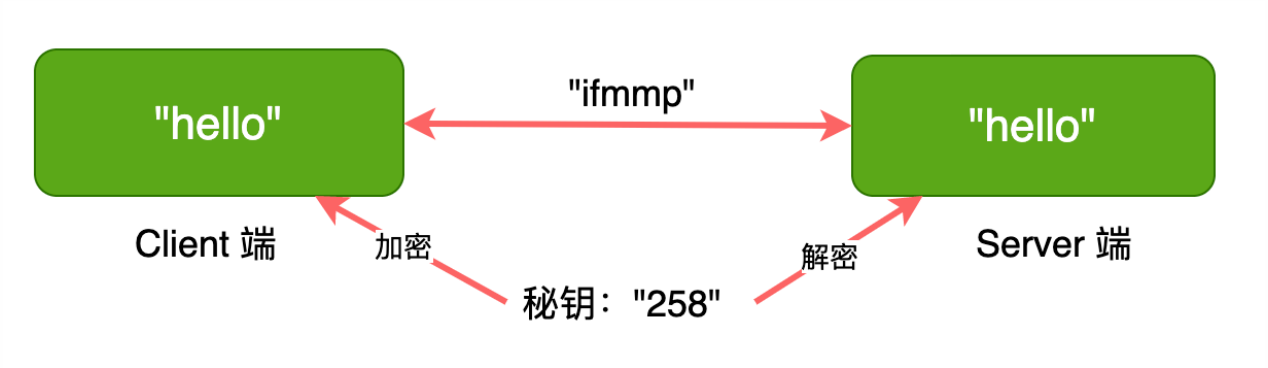
> 可逆加密；
> 不可逆加密。

刚才描述的加密、解密过程("hello"->"ifmmp"->"hello")是属于可逆加密，也就是说可以把密文还原成明文，主要应用在通信场景中。如果一个密文不能还原成明文，就称为不可逆加密，不可逆加密也非常重要。

可逆加密

刚才已经说到，可逆加密就是可以把密文还原成明文，只要client端和server端商量好加密算法(例如刚才所说的利用ascII表的下一个字符)就可以达到目的，也就是说：client端的加密算法和server端的解密算法是一样的，当然了这里的算法太简单。

我们可以稍微复杂一点点，先定义一个固定的字符串“258”，然后把明文"hello"中的每一个字符，用固定的字符串进行计算：先加2，再减5，最后加8，得到加密后的字符串"mjqqt"，server接收到之后再执行相反操作就解密得到明文“hello”。从算法角度看，这两个加密方式是一样的，但是第二种算法利用了一个独立的、固定的字符串“258”，这个字符串就叫做秘钥，当然，实际通信中使用的秘钥更复杂。通信双方是通过算法+秘钥的方式来进行加密和解密。而且，通信双方使用的秘钥是相同的，这就叫做对称加密。



既然存在对称加密，那肯定就存在非对称加密，也就是说，根据通信双方使用的秘钥是否相同，可逆加密分为2种：

> 1. 对称加密；
> 2. 非对称加密。

对称加密常用算法有：DES、AES；非对称加密常用算法有：RSA、DH、ECC。

对称加密的特点：

> 1. 计算速度快；

2. 加密强度高;
3. 能处理的数据量大。

非对称加密的特点:

1. 效率低;
2. 能处理的数据量大小有限制。

既然非对称加密的缺点这么明显，那么它有什么作用呢?

回到刚才的通信示例场景中：client与server需要使用同一个秘钥“258”，那么它们双方应该如何协商得到这个对称秘钥呢？难道是使用固定的秘钥吗？显然这个答案不太可能，需要通信的设备那么多，不可能像网卡的MAC地址那样预先分配，而且秘钥很容易泄漏。因此，这个对称秘钥一般都是在通信的刚开始的握手阶段，由client与server动态的协商得到的。在这个协商的过程中，为了防止协商内容被第三方截获，就需要使用非对称加密来保证握手阶段的数据安全性。

因为握手数据只发生在通信的刚开始阶段，即使效率低一点也没关系，安全比效率更重要。

一句话：非对称加密在通信初始阶段的协商过程中使用，用来得到一个对称秘钥，这个协商过程就叫做握手，在后面的HTTPS通信过程中，我们再详细看一下握手过程。

不可逆加密

顾名思义，不可逆加密就是说把明文加密之后得到密文，但是不能从密文还原得到明文。从术语上来说，一般不把这个加密结果称作密文，而是称作摘要或者指纹。

不可逆加密原理：把一个任意大小的数据，经过一定的算法，转换成规定长度的输出。如果数据的内容发生了一丝丝的变化，再次加密就得到另一个不同的结果，而且是大不相同。从这个角度看，是不是称作指纹更形象一些?

不可逆加密最常用的算法就是：MD5、SHA1。

回想一下：我们在下载一些软件的时候，在服务器上除了看到软件的下载地址，一般还会看到该软件的MD5码。我们把软件下载到本地之后，计算得到MD5，也就是文件的指纹，然后把这个MD5与服务器上公布的MD5进行比较，如果这两个MD5不一致，就说明下载的文件被别人修改过。

这是glib库的下载页面:

glib-2.16.6.changes
glib-2.16.6.md5sum
glib-2.16.6.news
glib-2.16.6.sha256sum
glib-2.16.6.tar.bz2
glib-2.16.6.tar.gz

补充：SHA相关知识

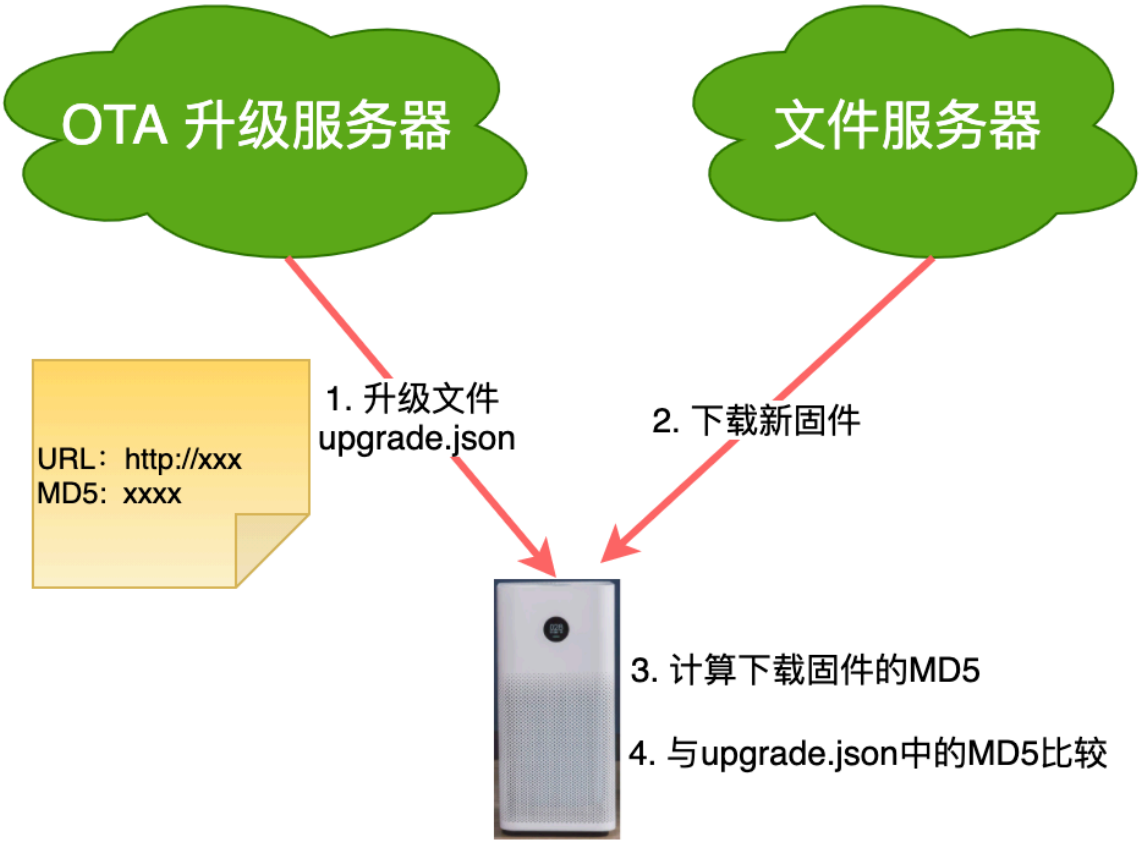
1. SHA

   SHA安全哈希算法，由美国国家标准技术研究院发布的一组加密函数。它是一种常用的摘要算法，就是输入一段数据，输出合法的证书一个摘要信息，包括SHA0、SHA1、SHA2等不同的版本。
2. SHA1

   代表安全哈希算法1，接收输入，输出一个160位的哈希值，称作信息摘要。在2005年之后，SHA1被认为不安全。
3. SHA2

   SHA2指的是具有两个不同块大小的相似哈希函数的族，其中包括：SHA256，SHA512。SHA256可以输出一个256位的哈希值，安全级别更高。

一个实际的使用场景：OTA升级



1. 首先服务器推送一个upgrate.json格式字符串给设备，文件中包括：新固件的下载地址URL，新固件的MD5值；
2. 设备根据URL下载新固件到本地；
3. 设备计算下载的新固件MD5值，与upgrade.json中的MD5值进行比较；
4. 如果这两个MD5值一致，说明下载的固件没有问题，那么就开始升级。

再来了解一下不可逆加密的特点：

1. 不可逆：除非使用穷举等手段，原则上不存在根据密文推断出原文的算法；
2. 雪崩效应：对输入数据敏感，原始数据的极小改动会造成输出指纹的巨大差异；
3. 防碰撞：很难找到两段不同的数据，输出相同的指纹。

公钥和私钥

上面说到了非对称加密，那么就必须再补充一下私钥和公钥。从字面上就可以看出：它俩是一对兄弟，都是秘钥，必须成对使用，称作：秘钥对。我们可以通过一些软件工具(例如：OpenSSL)生成自己的公钥和私钥。

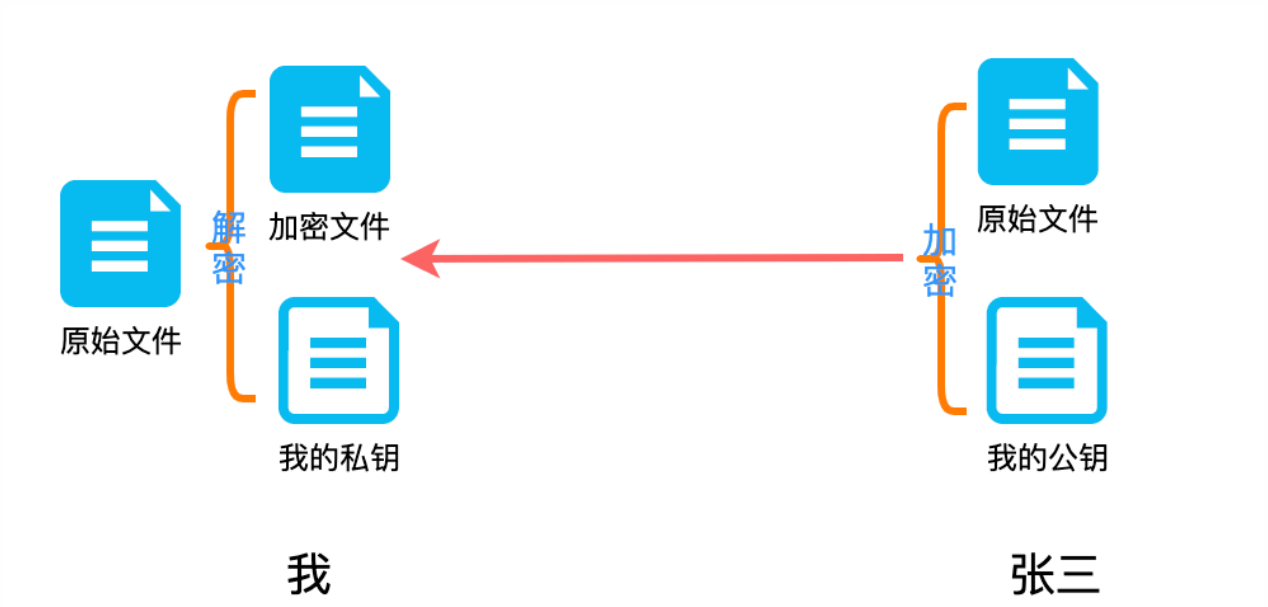
公钥：就是公开告诉别人的；私钥：就是自己的，作为宝贝一样自己私藏起来，千万不要告诉别人。

公钥和私钥的作用有2个：

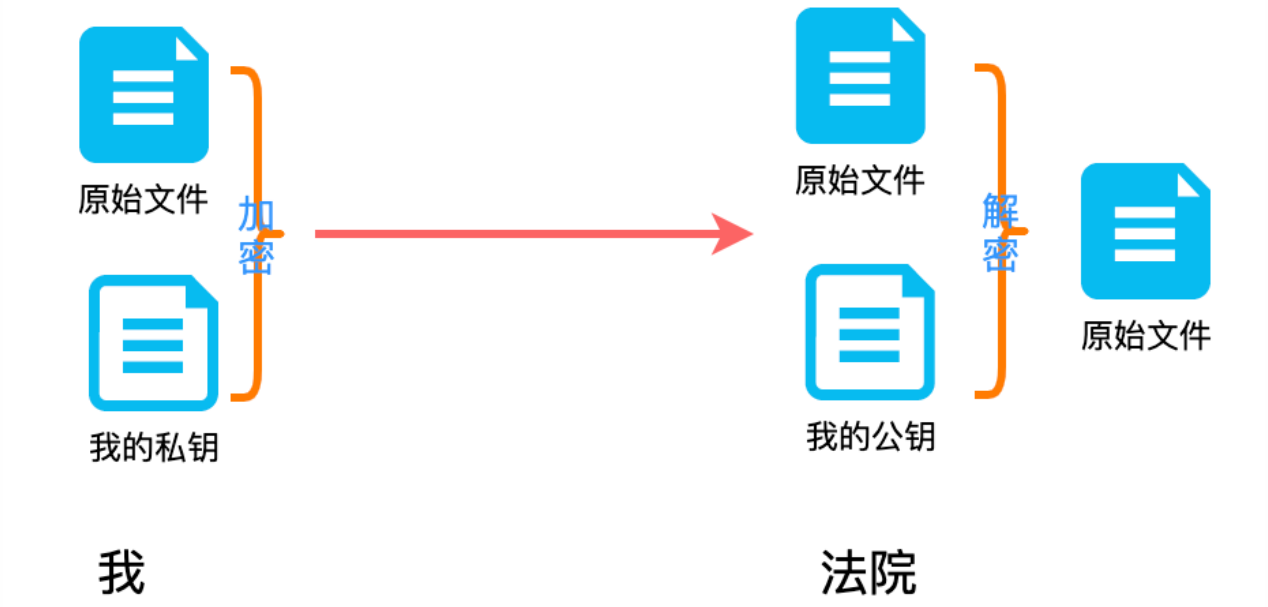
1. 数据加密：公钥加密，私钥解密，用于通信场景；
2. 数字签名：私钥加密，公钥解密，用于不可要赖场景。

数据加密就是上面描述的非对称加密，例如：

张三想发一个文件给我，为了防止文件被其他人看到，于是张三用我的公钥对文件进行加密，然后把加密后的文件发给我。我拿到密文后，用我的私钥就可以把密文还原成原始的文件，其他人即使拿到了密文，但是没有我的私钥，就解不开文件。如下面这张图：



数字签名与我们日常生活中的借条上的签名类似，一旦签名了，就具有法律效力，不能要赖说：这个不是我签名，我不认。具体流程是：我写了一个文件，然后用我的私钥对文件进行加密，那么如果以后我要赖说：这个文件不是我写的，其他人就可以用我的公钥来尝试对加密后的文件进行解密。如果成功解密了，就说明这个文件一定是用我的私钥进行加密的，而私钥只有我才有，那就说明这个文件一定是我写的。如下图：

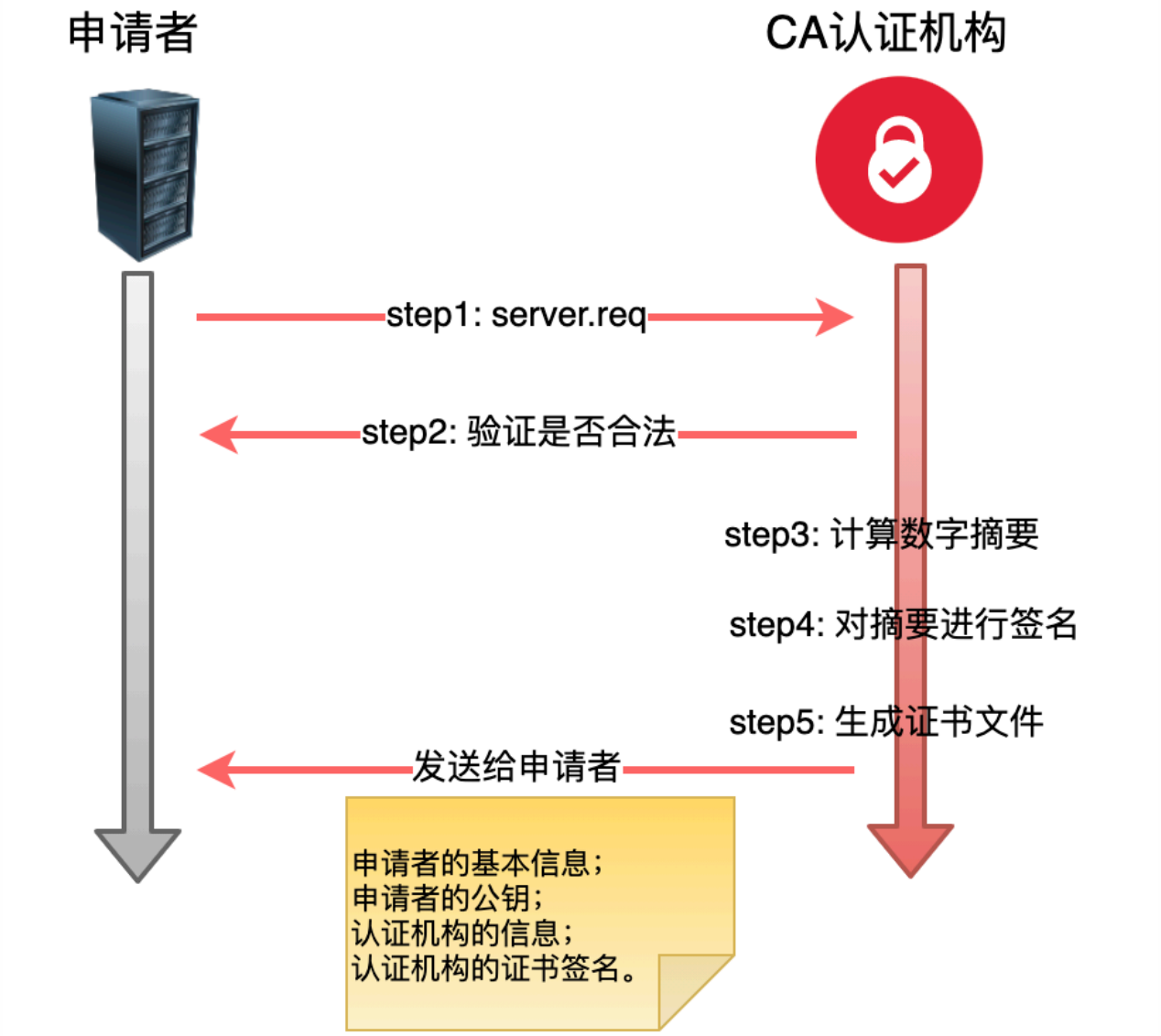


证书

前面谈到了公钥是公开给别人的，本质上就是一段数据，那么这段数据是以什么样的形式或者说以什么样的载体发送给别人的呢？答案就是：证书。

如何申请证书

我们以一个网站为例，浏览器在访问网站的时候，在握手阶段，网站会把自己的证书发送给浏览器。那么这个证书是如何产生的呢？



Step1

在网站上线之初，需要把自己的相关信息放在一个请求文件中(server.req)，把请求文件发送给一个权威的认证机构。请求文件的内容包括：

> 网站的域名
> 申请者信息
> 公钥
> 以及其他一些相关信息

Step2

认证机构通过其他途径来确定申请者是合法的。

Step3

认证机构使用某个算法，对请求文件server.req中的信息进行计算，得到一个数字摘要。

算法包括：

> MD5
> SHA-1
> SHA-256

信息包括：

> 申请者的基本信息：网站使用的加密算法、网站使用的hash算法；
> 申请者的公钥；
> 认证机构的信息：认证机构的名称，证书到期时间。

Step4

认证机构用自己的私钥，对Step3中得到的数字摘要进行加密，得到数字签名(也就证书签名)。

Step5

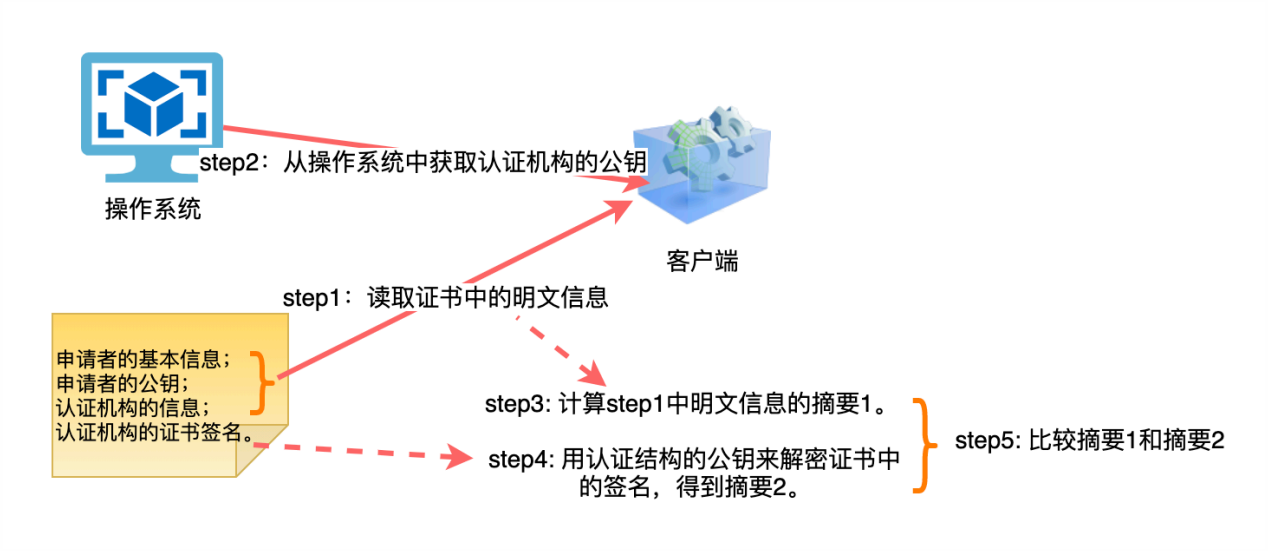
认证机构把以上这些信息进行汇总，得到最终的证书文件server.crt，然后发给申请者。

最终，证书server.crt中的内容包括这几个大类：

> 1. 申请者的基本信息：网站使用的加密算法、网站使用的hash算法；
> 2. 申请者的公钥；
> 3. 认证机构的信息：认证机构的名称，证书到期时间。
> 4. 认证机构的证书签名。

如何确认证书的合法性

现在，客户端拿到了服务器发来的证书文件，应该如何验证这是一个合法的证书呢？



Step1

读取证书中的明文信息，包括：申请者的基本信息，申请者的公钥，认证机构的信息。

Step2

从浏览器或者操作系统中查找这个认证机构的相关信息，得到这个认证机构的公钥。

补充：浏览器或者操作系统中，一般都会预装一些可信任的权威认证机构的证书列表，所以能拿到认证机构的公钥。

Step3

使用认证机构相同的算法，对Step1中的明文信息进行计算，得到摘要1。

Step4

使用认证机构的公钥，对证书中认证机构的数字签名进行解密，得到摘要2。

Step5

比较摘要1与摘要2是否相同，如果相同，说明这个证书是合法的，也就证明当前访问的是一个合法的服务器。

单向认证和双向认证

上面描述的认证过程，是浏览器用来确认所访问的网站是否是一个合法的网站；文章开头所举的例子：一个物联网产品在连接云平台的时候，是云平台来验证这个想连接进来的设备是否为一个合法的设备。

这两个场景中都是单向认证，也就是通信的一方来验证另一方是否合法。那么双向认证就很好理解了：通信的每一方都要认证对方是否合法。

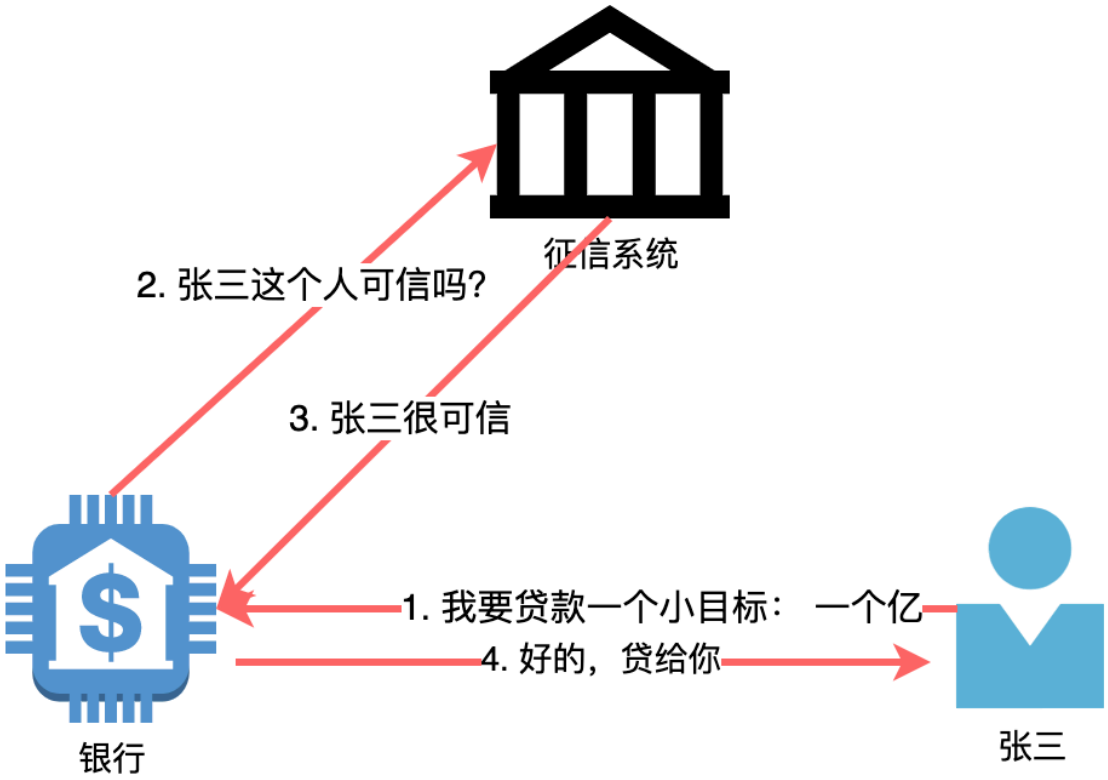
至于如何选择使用单向认证或者双向认证，甚至是不使用证书(只使用用户名和密码来鉴权)，这就需要根据实际的使用场景、安全等级、操作的难易程度来决定了。比如：在物联网产品中，每一个产品都需要在生产阶段把动态生成的证书烧写到设备中，增加了生产环节的流程和成本，为了安全性，万万不可偷懒。如果没有证书来验证，那么黑客就可以模拟无数个设备，频繁的连接到云平台，这就存在极大的安全隐患。

认证机构

证书本质上就是一个文件，只不过这个文件具有特殊的一个性质：可以被证明是合法的。那么应该如何来证明呢？这就要来说一下认证机构。

认证机构(CA: Certificate Authority)是一个权威的组织，是被国家、行业认可的权威结构，不是随便一个机构都有资格颁发证书，不然也不叫做权威机构。只要能证明一个证书是由CA机构颁发的，我们就认为这个证书是合法的，也就是说：证书的可信任性基于信任机制。

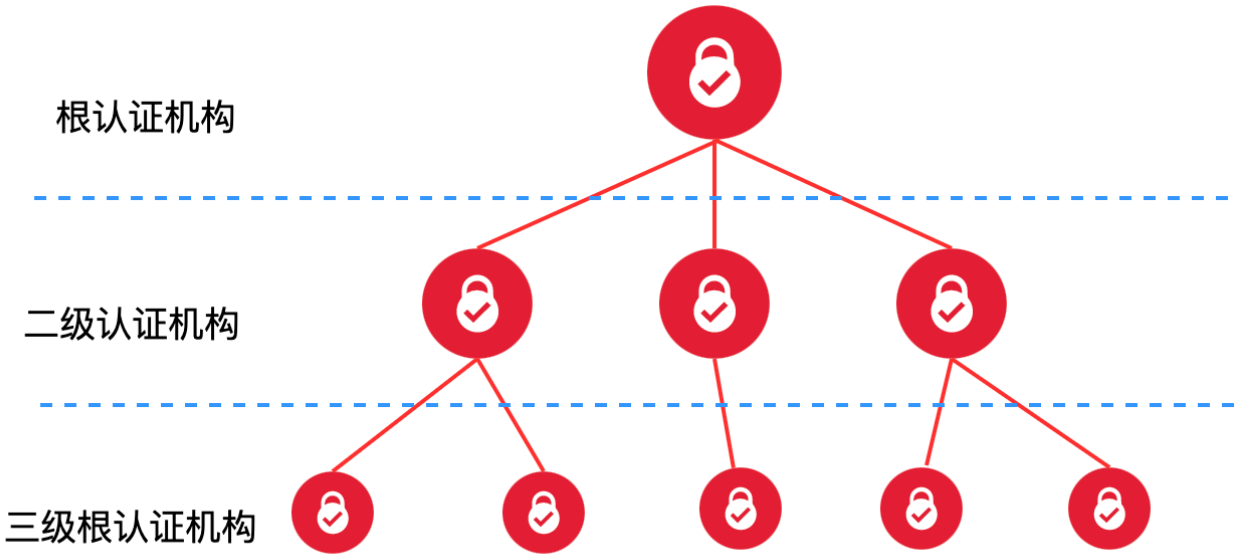
就像银行贷款给个人一样，银行在放款之前，会到征信系统中查询这个人的信用报告，如果征信系统中表明这个人的信用没有问题，银行相信征信系统，所以银行就相信这个人，可以贷款给他，这是一个信任链的传递。



CA认证机构就类似于征信系统，相当于CA结构给证书进行了背书，它保证从它手里颁发的证书都是合法有效的，那么我们只要能证明证书是从CA认证机构颁发的，就可以认为证书是有效的。

证书链

CA认证机构是一个树状的结构，最顶部的称为根认证机构。往下层是：二级认证机构、三级认证机构...。



根认证机构给二级认证机构颁发证书，二级认证机构给三级认证结构颁发证书...。不同等级的认证机构对审核的要求也不一样，于是证书也分为免费的、便宜的和贵的。

你可能会问：那么根认证机构的证书是由谁签名的？答案是：根认证机构自己签名的，这也叫做自签名。因为根认证机构是由国家或者行业组织认可的，已经是一个可以信赖的权威机构，所以可以为自己签名。

另外，我们在测试的过程中，也常常利用OpenSSL中提供的程序来产生自签名的证书，当然，这个测试的自签名证书只能你自己玩，因为别人不信任你。

证书文件的后缀名

刚接触到证书概念的小伙伴，常常被眼花缭乱的后缀名所迷惑。

首先要明确一点：证书文件的后缀名只是为了见名识意，实际上可以取任意的名字。常见的后缀名包括：

> .crt：per格式的证书
> .der: der格式的证书
> .key：pem格式的私钥
> .pub：pem格式的公钥
> .req：申请证书时发送给CA认证机构的请求文件
> .csr：也表示请求文件

证书文件的格式

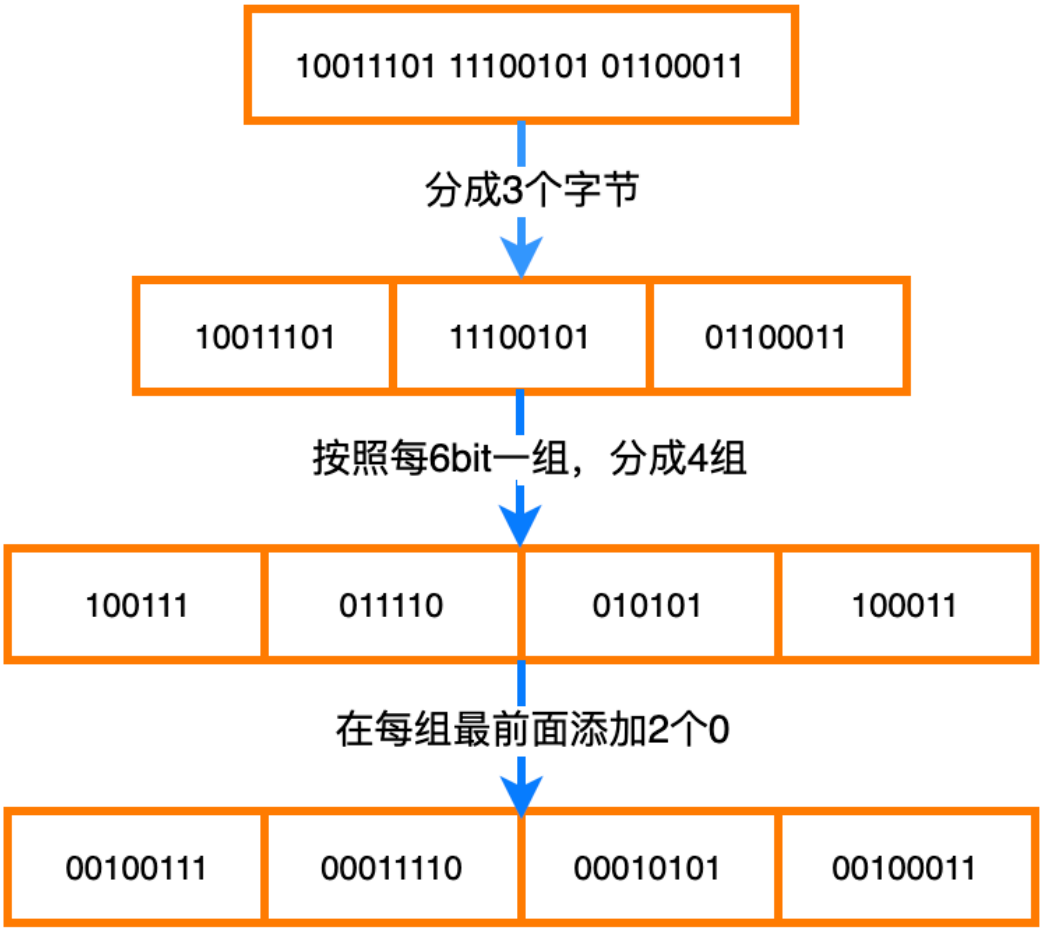
所有证书内容格式有两种：pem格式和der格式，这两种格式的证书文件可以相互转换，利用OpenSSL中的程序就可以完成。

PEM格式(Privacy Enhanced Mail)

pem格式的证书内容是经过加密的文本文件，一般是base64格式，可以用记事本来打开一个base64格式的证书，例如下面这个证书文件的内容：

```
-----BEGIN CERTIFICATE-----
MIIGbzCCBFegAwIBAgIICZftEJ0fB/wwDQYJKoZIhvcNApQELBQAwfDELMAkGA1UE
BhMCVVMxDjAMBgNVBAgMBVRleGFzMRAwDgYDVQQHDAdIb3VzdG9uMRgwFgYDVQQK
DA9TU0wgQ29ycG9yYXRpb24xMTAvBgNVBAMMKFNTTC5jb20gUm9vdCBDZXJ0aWZp
...
Nztr2Isaaz4LpMEo4mGCiGxec5mKr1w8AE9n6D91CvxR5/zL1VU1JCVC7sAtkdki
vnN1/6jEKFJvlUr5/FX04JXeomIjXTI8ciruZ6HIkbtJup1n9Zxvmr9JQcFTsP2c
bRbjaT7JD6MBidAWRCJWClR/5etTZwWwWrRCrzvIHC7WO6rCzwu69a+l7ofCKlWs
y702dmPTKEdEfwhgLx0LxJr/Aw==
-----END CERTIFICATE-----
```

补充：base64算法就是把原始数据中按照每3个字节进行拆分，3个字节是24bit，然后把24bit分成4组，每组6bit，最后在每个6bit的签名添加2个0，这样得到的4组字节码就可以用ascII码来表示了。



DER格式(Distinguished Encoding Rules)

der格式的证书文件内容是经过加密的二进制数据，也就是说文件内容打开后是乱码。

X.509标准

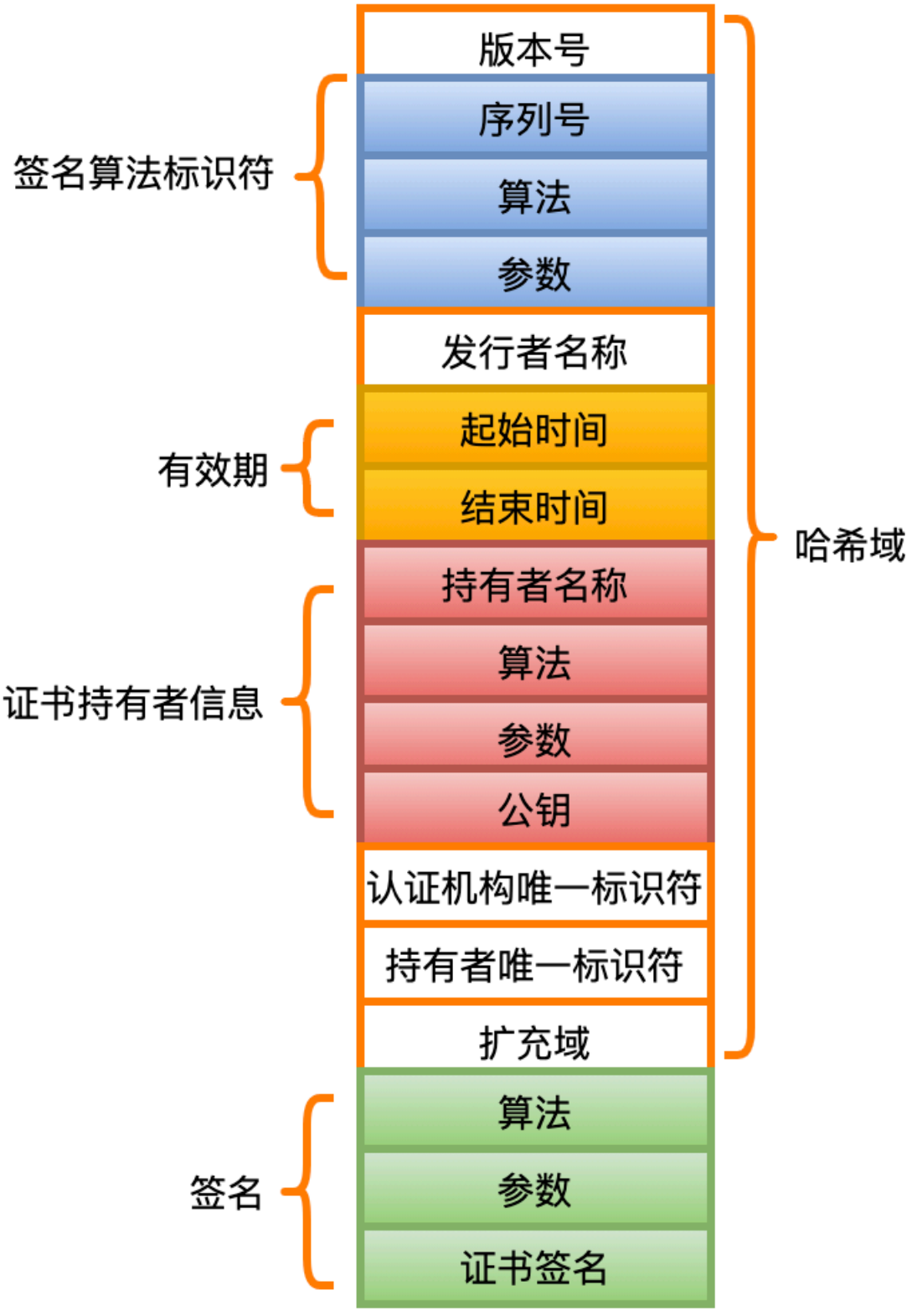
上面说到证书中包含了必要的信息，那么这些信息在文件中并不是随意摆放的，而是要根据固定的格式来存储，只有这样才能通过软件生成或解析。
那么这个固定的格式是由谁来规定的呢？这就是X.509标准与公共秘钥证书。

X.509是一个体系、标准，用来规定一个证书的格式标准，CA认证结构在生成证书的时候，就根据这个标准把每部分信息写入到证书文件中。

X.509包括3个版本：V1, V2和V3。每一个版本中颁发证书时，必须包含下列信息：

> 版本号：用来区分版本；
> 系列号：由CA认证机构给每一个证书分配一个唯一的数字编号；
> 算法签名标识符：用来指定CA认证机构在颁发证书时使用的签名算法；
> 认证机构：颁发证书的机构唯一名称；
> 有效期限：证书有效期(开始时间和结束时间)；
> 主题信息：证书持有人的基本信息；
> 公钥信息：证书持有者的公钥；
> 认证结构签名：以确保这个证书在颁发之后没有被篡改过；

证书格式



总之：证书的核心功能就是安全的传递公钥！

OpenPGP协议/标准

加密和证书的概念介绍完了，再来了解一个行业标准：OpenPGP。

OpenPGP是什么?

OpenPGP是一种非专有协议，为加密消息、签名、私钥和用于交换公钥的证书定义了统一标准。

OpenPGP协议的实现

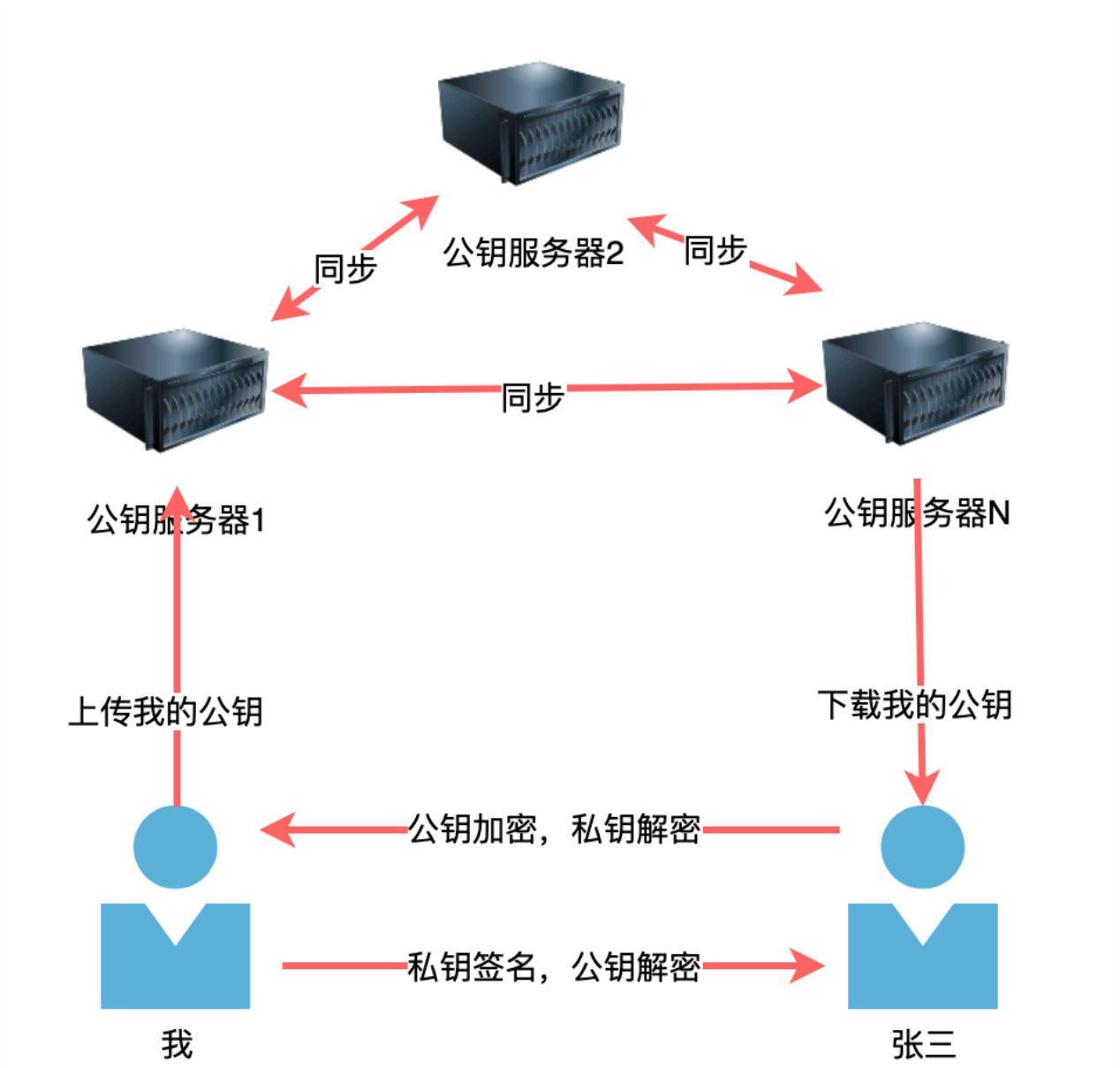
OpenPGP协议有2个实现:

> 1. PGP(Pretty Good Privacy)
> 2. GPG(GNU Privacy Guard)

PGP是一个加密程序，为数据通信提供了加密和验证功能，通常用于签名、加密和解密文本、电子邮件和文件。

GPG是PGP的开源实现。

OpenPGP的使用流程



Step1: 公布自己的公钥

每个人把自己的公钥上传到公钥服务器上(subkeys.pgp.net)，然后通过交换机制，所有的公钥服务器最终都会包含你的公钥，就类似域名服务器同步域名解析信息一样。

由于公钥服务器没有检查机制，任何人都可以用我的名义上传公钥，所以没有办法来保证服务器上的公钥一定是可靠性。通常，我可以在网站上公布一个公钥指纹，让其他人下载我的公钥之后，计算一下公钥指纹，然后与我公布的指纹进行比对，以此来确认证书的有效性。

Step2: 获取别人的公钥

为了获得别人的公钥，可以让对方直接发给我，也可以从公钥服务器上下载。为了安全起见，需要对下载的公钥使用其他机制进行安全认证，例如刚才说的指纹。

Step3: 用于加密

用对方的公钥加密文件，发送给对方，对方用他自己的私钥进行解密。

Step4: 用于签名

用我的私钥进行加密，把加密后文件发送给对方，对方用我的公钥进行解密，只要能正确解密，就证明这个文件的确是我加密的。

SSL/TLS

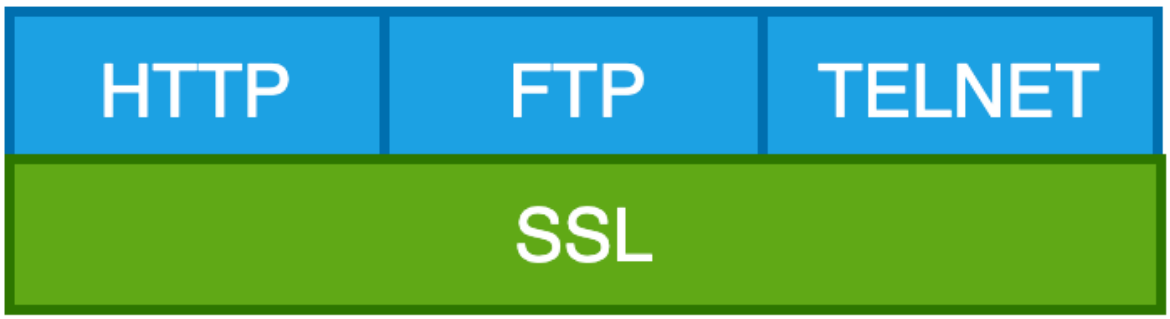
SSL全称是Secure Socket Layer(安全套阶层协议)，它是一个安全协议，目的是用来为互联网提供安全的数据传输。

SSL在工作过程中，就利用了前面描述的概念：对称加密、非对称加密、证书等。如果前面的概念都梳理清楚了，那么理解SSL也就不成问题了。

SSL协议有1，2，3这个三个版本，TLS是SSL V3标准化之后的产物。事实上现在用的都是TLS，但是大家都习惯了SSL这个称呼。

协议分层

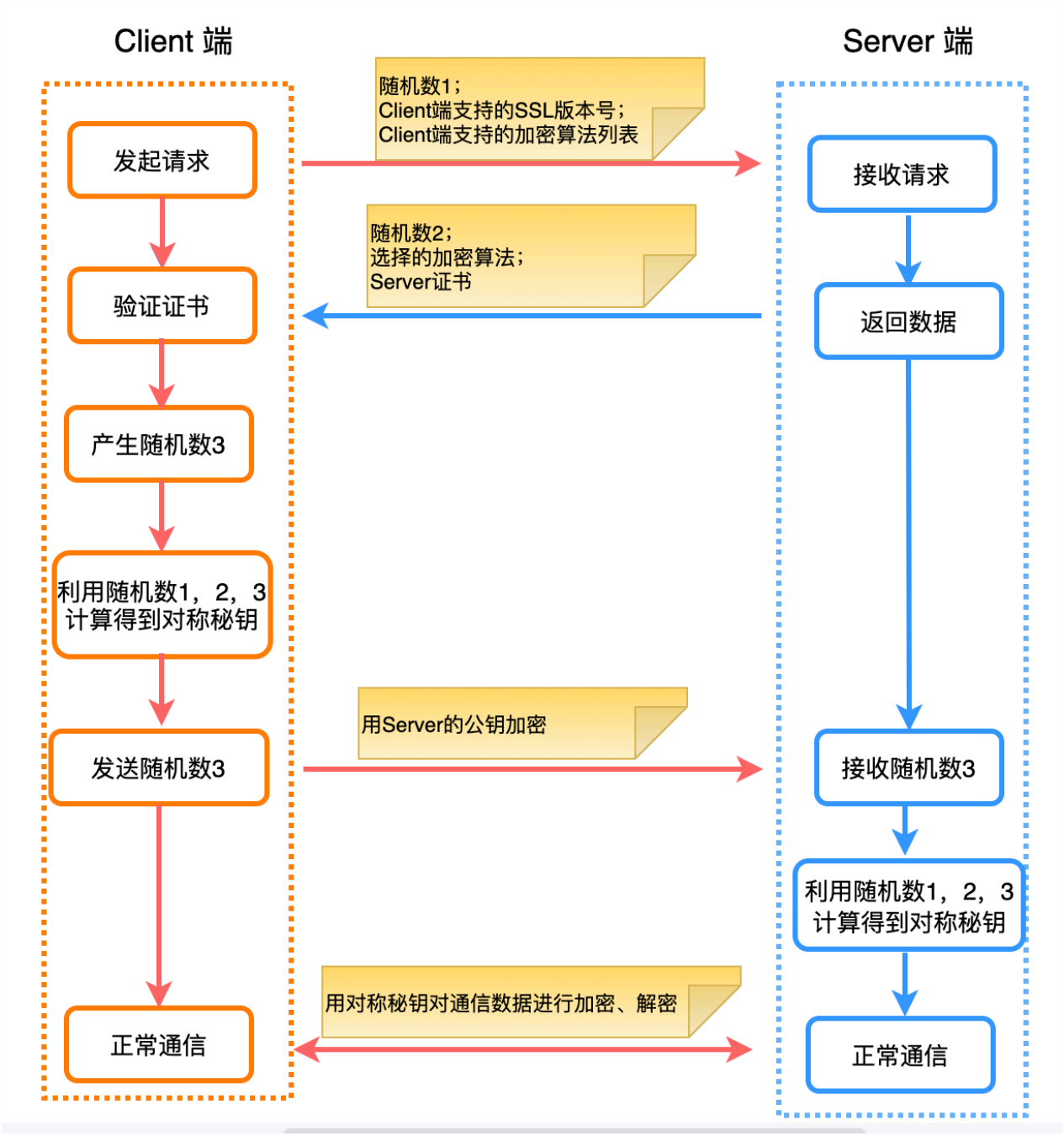
SSL协议最大的优点就是与应用层无关，在SSL协议的上层，可以运行一些高层应用协议，例如：HTTP, FTP, TELNET...，也就是说这些高层协议可以透明的建立在SSL协议层之上。



握手过程

SSL使用X.509标准，握手就是指客户端与服务端在通信的开始阶段进行鉴权和协商，最终目的是：

1. 确认对方是合法的通信对象;
2. 与对方协商得到对称加密秘钥。



我们来一步一步梳理握手过程:

Step1

Client向Server发送如下信息:

> 随机数1;
> Client端支持的SSL版本号;
> Client端支持的加密算法列表。

Step2

Server分析接收到的信息，返回如下信息给Client:

> 随机数2；
> 选择的加密算法；
> Server证书

Step3

Client端验证Server发来的证书是否有效，具体过程上面已经描述过。

如果验证失败，通信结束；
如果验证通过，就产生随机数3，并使用刚才的随机数1、随机数2，然后用选择的算法生成一个对称加密秘钥，这个秘钥就用于后面正常的数据通信中。

然后发送如下信息给Server:

> 随机数3，并且用server证书中的公钥进行加密；

此时，Cliend端的握手流程结束，因为已经达到了握手的最终目的：确认Server合法，得到对称加密秘钥。

Step4

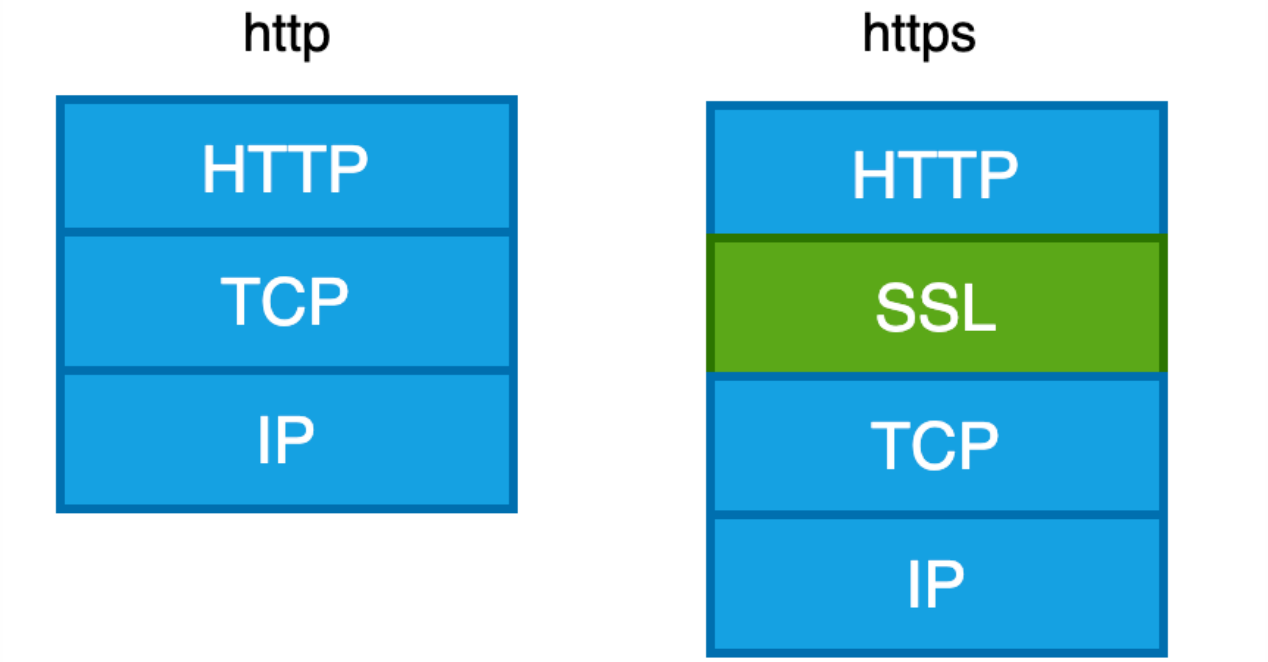
Server端在接收到加密后的随机数3时，用自己的私钥进行解密，然后和之前的随机数1、随机数2一起，使用相同的算法生成对称加密秘钥。

至此，Server端的握手过程也就结束，下面就可以用对称加密秘钥来对数据进行加密了。

注意：上面描述的握手过程中是单向认证，也就是Client端验证Server是否合法的。如果需要双向认证，那么客户端也应该把自己的证书发送给Server，然后Server来验证这个证书是否合法，确认证书合法之后才继续执行后面面的握手流程。

HTTPS与SSL的关系

HTTPS拆开来就是：HTTP+SSL，就是在HTTP的下面增加了SSL安全传输协议层，在浏览器连接到服务器之后，就执行上面描述的SSL握手过程。握手结束之后，双方得到对称加密秘钥，在HTTP协议看来是明文传输数据，下面的SSL层对数据进行加密和解密。



OpenSSL

OpenSSL是什么?

上面描述的SSL中这么多的东西都是协议(或者称为标准)，协议只是规定了应该怎么做，但是具体的代码实现应该由谁来做呢？我们在写相关的SSL程序时好像从来没有实现过这个协议，都是直接调用第三方提供的库就达到了加密传输的目的。当然了，如果你实现过SSL协议，请允许我对你表示佩服，给你一万个赞！

在编程领域，永远都存在热心肠的人！OpenSSL就是一个免费的SSL/TLS实现，就是说：OpenSSL实现了SSL/TLS协议中定义的所有功能，包括：

> SSL2
> SSL3
> TLSv1
> TLSv1.1
> TLSv1.2

而且，OpenSSL是用C语言开发的，具有优秀的跨平台特性，在Linux、Windows、BSD、MAC等平台上可以执行。

具体来说，OpenSSL实现中，包括下面几个功能模块：

密码算法库

密码算法库中包括：

> 对称加密算法: AES、DES等。
> 非对称加密算法: DH、RSA、DSA、EC等。

信息摘要算法

信息摘要算法包括：MD5、SHA等。

秘钥和证书管理

OpenSSL提供的CA应用程序就是一个小型的证书管理中心，实现了证书签发的整个流程和证书管理的大部分机制，我们在学习的时候一般都会用OpenSSL中提供的CA程序来生成秘钥对、自签名等等。具体的内容包括：

> 证书秘钥产生、请求产生、证书签发、吊销和验证功能；
> 对证书的X.509标准解码、PKCS#12/PKCS#7格式的编解码；
> 提供了产生各种秘钥对的函数；

SSL协议库

实现了SSLv2、SSLv3、TLSv1.0协议。

应用程序

OpenSSL的应用程序是基于密码算法库和SSL协议库实现的，是非常好的OpenSSL的API函数使用范例，主要包括：秘钥生成、证书管理、格式转换、数据加密和签名、SSL测试以及其他辅助配置功能。

OpenSSH又是什么?

SSH

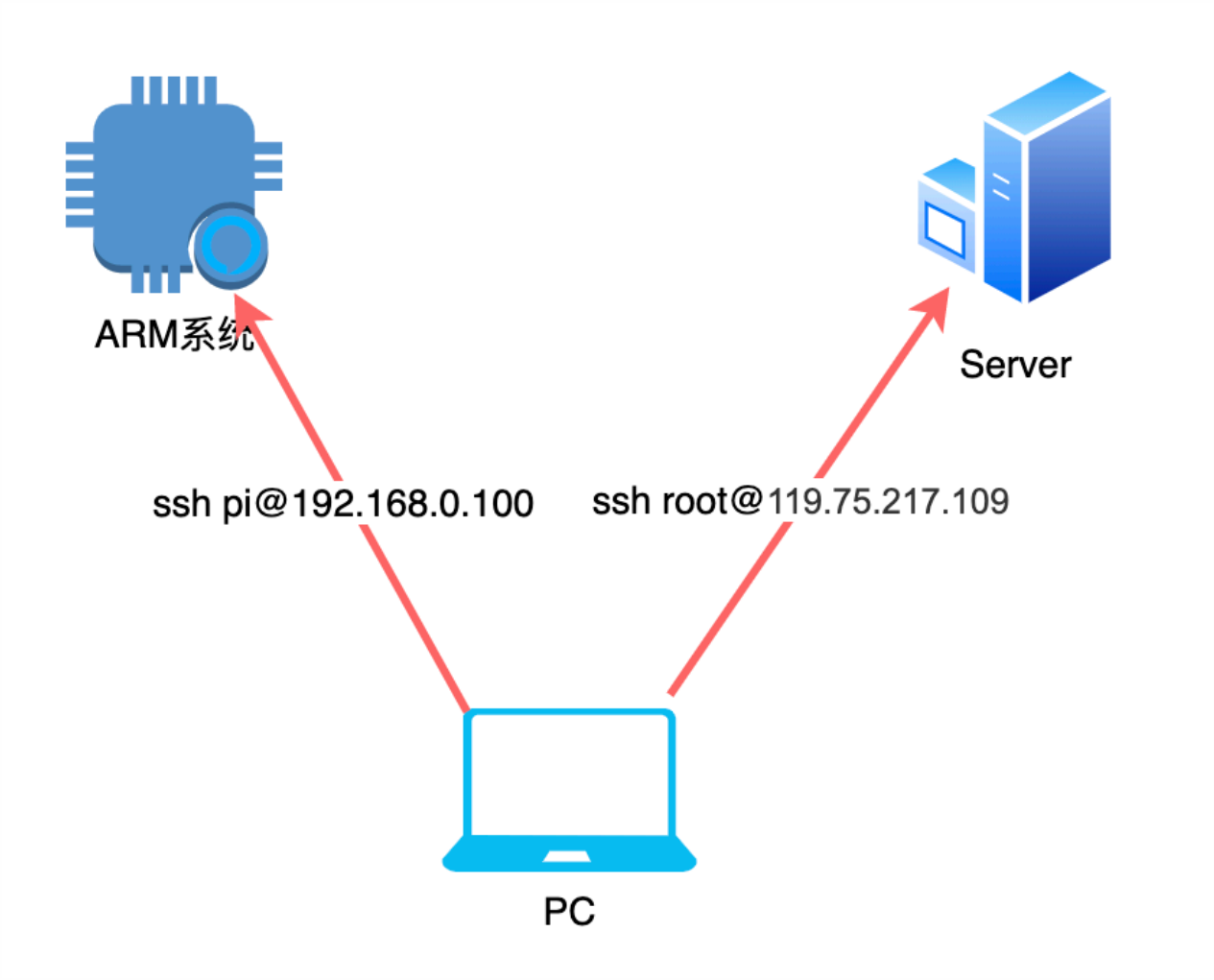
首先说一下SSH：Secure Shell(安全外壳协议)，又是一个协议，用来实现远程登录系统，我们通常利用SSH来传输命令行界面和远程执行命令。

比如：在调试一个ARM系统时，可以通过串口助手连接到ARM板子上；但是更常用的调试场景是在PC机上远程登录到ARM系统中，执行ARM中的任何指令，这就是利用SSH来实现的。

SSH提供2种级别的安全验证：

> 基于密码；
> 基于秘钥。

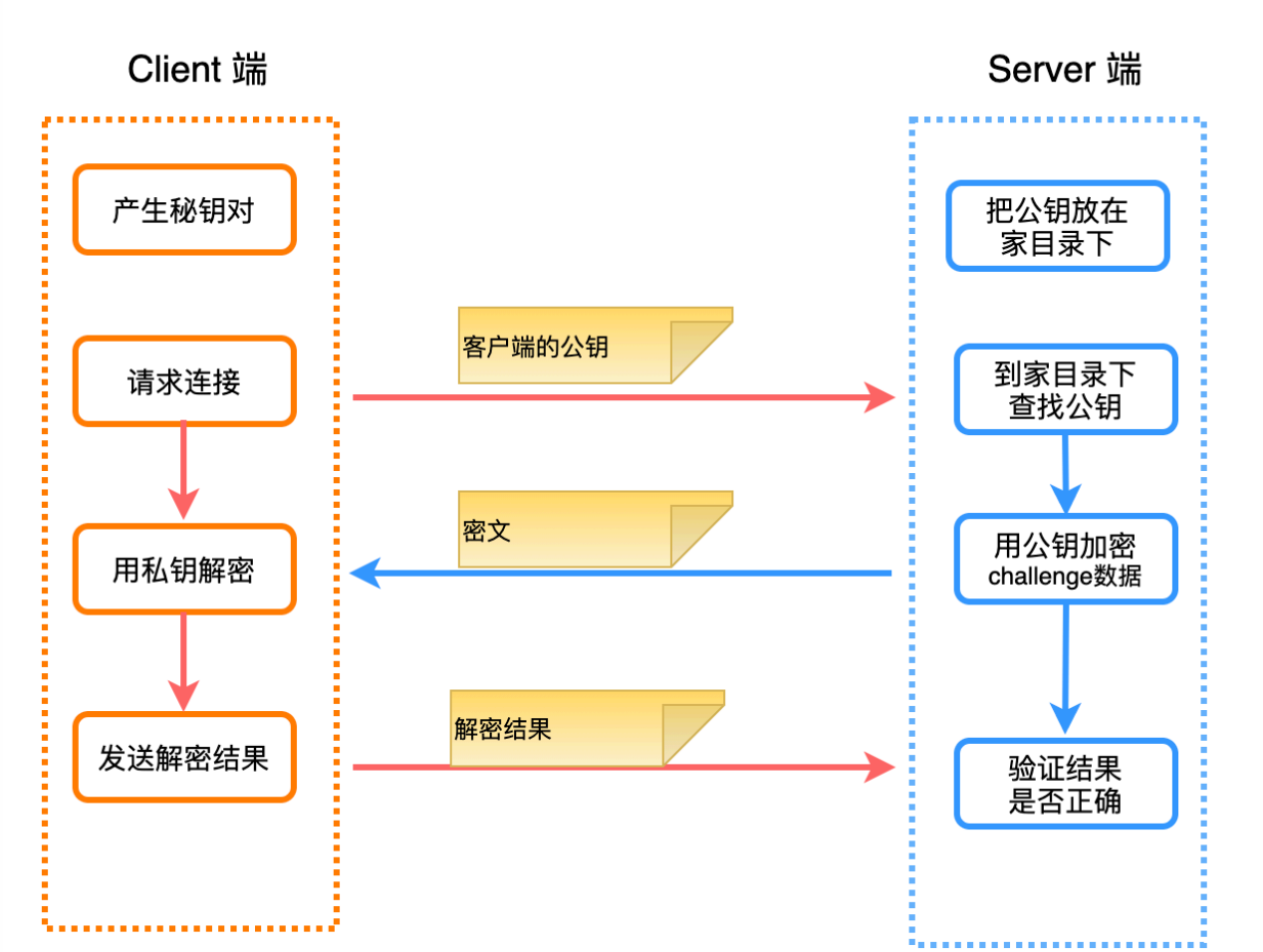
SSH中基于口令的安全验证



只需要知道账号和密码，就可以远程登录到系统，我们一般常用的就是这用方式。但是不能保证我正在连接的设备就是我想连接的那台设备，可能会有别的服务器冒充真正的服务器，也就是受到“中间人”这种方式的攻击。

当然还经常遇到另一种错误：在局域网中有多台设备，本想远程连接到设备A中，由于IP地址记错了，结果远程登录到另外一台设备B上了，如果你的同事也正在调试设备B，接下来就是悲剧发生的时刻！

SSH中基于秘钥的安全认证



Step1

首先为自己创建一个秘钥对，并提前把公钥放在需要访问的服务器上，例如：放在账号的家目录中。

Step2

通过客户端远程登录到服务器，把自己的公钥发给服务器，并请服务器进行安全验证。

Step3

服务器接收到请求后，在登录账号的家目录下查找公钥，然后与接收到的公钥进行比对。

Step4

如果比对不一致，通信结束；如果比对一致，服务器就用公钥加密一段数据(challenge)，并发送给客户端。

Step5

客户端接收到challenge，用私钥进行解密，然后把结果发给服务器。

Step6

服务器把接收到的结果与Step4中的数据进行比对，如果一致则验证通过。

SSH协议的实现

既然SSH是一个协议，那么就一定存在对应的实现，这就是OpenSSH，它是一个免费开源的SSH实现。

OpenSSH实现中，利用了OpenSSL中的加密和算法库函数，这就是它俩之间的关系。

总结

到这里，与加密、证书相关的基础概念都介绍完毕了，不知道你是否有所收获。

如果你是初次接触到这些东西，敬请放心，即使现在明白了，一个星期之后肯定忘记一多半了。只有经历过几个项目的历练之后，才会有更深刻的理解和记忆，最后，祝您好运！

长按下图二维码关注，每篇文章都有干货。

推荐阅读

[1] 原来gdb的底层调试原理这么简单
[2] 生产者和消费者模式中的双缓冲技术
[3] 深入LUA脚本语言，让你彻底明白调试原理
[4] 一步步分析-如何用C实现面向对象编程