说起 HTTP 的那些事，则不得不提 HTTPS ，而说起 HTTPS ，则不得不提数字证书。这篇博客将从 [Java](http://www.2cto.com/kf/ware/Java/) 的角度，学习 HTTPS 和数字证书技术，并分享爬虫开发的过程中针对爬取 HTTPS 站点时可能遇到的一些问题。  
在前面的几篇博客里，其实已经略微提到过 HTTPS 了，譬如使用 HttpsURLConnection类发送 HTTPS 请求，在使用代理时 HTTP 和 HTTPS 的一些差异等等。关于 HTTPS 的概念就不废话了，下面直接进入正题。

**一、访问 HTTPS 站点**

在前面介绍了两种方法来模拟发送 HTTP 请求，访问 HTTP 站点。一种方式是通过 java.net 自带的 HttpURLConnection，另一种方式是通过 Apache 的 HttpClient，这两种方式各有各的优势。这里也使用这两种方式来访问 HTTPS 站点，从下面的代码可以看到，和前面访问 HTTP 站点几乎完全一样。

**1.1 使用 HttpURLConnection**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | @Test **publicvoid** basicHttpsGet() **throws**Exception { String url = "[https://www.baidu.com](https://www.baidu.com/)"; URL obj = **new**URL(url); HttpsURLConnection con = (HttpsURLConnection) obj.openConnection(); con.setRequestProperty("User-Agent","Mozilla/5.0 (Windows NT 6.1; WOW64) ..."); con.setRequestProperty("Accept-Language","en-US,en;q=0.5"); con.setRequestMethod("GET"); String responseBody = readResponseBody(con.getInputStream()); System.out.println(responseBody); } |

**1.2 使用 HttpClient**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | @Test **publicvoid** basicHttpsGet() **throws**Exception { String url = "[https://www.baidu.com](https://www.baidu.com/)"; HttpGet request = **new**HttpGet(url); request.setHeader("User-Agent","Mozilla/5.0 (Windows NT 6.1; WOW64) ..."); CloseableHttpClient httpclient = HttpClients.createDefault(); CloseableHttpResponse response = httpclient.execute(request); String responseBody = readResponseBody(response); System.out.println(responseBody); } |

具体的代码解释参见第一篇博客，这里不再赘述。一般情况下，访问 HTTPS 站点就和访问 HTTP 站点一样简单，无论是 HttpURLConnection 还是 HttpClient ，都将底层的实现细节封装了起来，给我们提供了一致的对外接口，所以我们不用关心 HTTPS 的实现原理。对底层细节的封装，本来是一件好事，也是一种好的设计方式，可以让开发人员使用起来更方便，提高开发效率，但是对于那些不求甚解的人来说，可 能带来的困惑比之带来的方便要更多。

**1.3 遭遇 PKIX path building failed**

使用上面的代码作为爬虫程序爬取成千上万的网页，在大多数情况下，无论是 HTTP 也好，HTTPS 也罢，都可以很好的工作。不过有时候，你可能没那么好的运气，有些站点在墙外，被强大的防火长城拒之门外，这时你可以找一些境外代理，通过《使用代理》这 篇博客中介绍的方法来解决；有些站点需要使用身份认证输入用户名密码才能访问，这可以使用上一篇博客《代理认证》中介绍的方法来解决；另外，在访问有些 HTTPS 站点时，你还可能会遇到下面的异常：

javax.net.ssl.SSLHandshakeException:  
sun.security.validator.ValidatorException: PKIX path building failed:  
sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target

要解决这个异常，这就是我们这篇将要介绍的内容。

**二、证书认证的原理**

大多数人第一次遇到上面的异常时的反应，估计都是一脸茫然，因为这个异常信息提示比较模糊，对于不懂 HTTPS 的人来说，什么 SSLHandshake ，什么 PKIX path ，完全不知所云。所以我们要先弄懂 HTTPS 的工作原理，才好去解决这个问题。我们知道 HTTPS 其实就是 HTTP + SSL/TLS 的合体，它其实还是 HTTP 协议，只是在外面加了一层，SSL 是一种[加密](http://www.2cto.com/article/jiami/)安全协议，引入 SSL 的目的是为了解决 HTTP 协议在不可信网络中使用明文传输数据导致的安全性问题。可以说，整个互联网的通信安全，都是建立在 SSL/TLS 的安全性之上的。

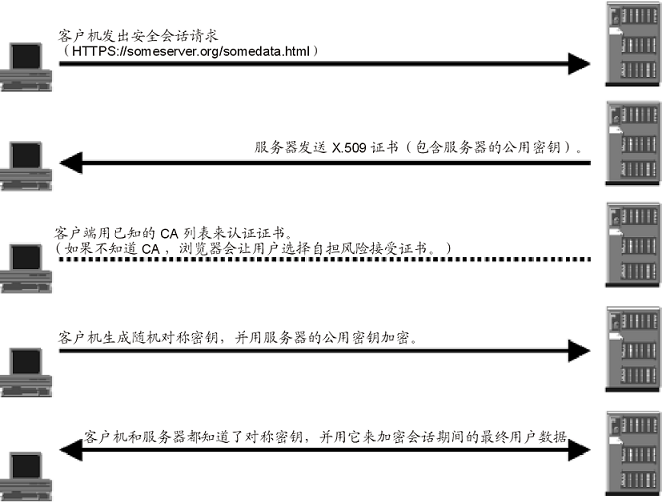
**2.1 SSL/TLS 协议及其握手过程**

学过计算机网络的同学肯定都还记得 TCP 在建立连接时的三次握手，之所以需要 TCP 三次握手，是因为网络中存在延迟的重复分组，可能会导致服务器重复建立连接造成不必要的开销。SSL/TLS 协议在建立连接时与此类似，也需要客户端和服务器之间进行握手，但是其目的却大相径庭，在 SSL/TLS 握手的过程中，客户端和服务器彼此交换并验证证书，并协商出一个 “对话密钥” ，后续的所有通信都使用这个 “对话密钥” 进行加密，保证通信安全。

网上有很多 SSL/TLS 握手的示意图，其中下面这副非常全面，也非常专业，想深入了解 SSL/TLS 的同学可以研究下。

http://www.cheat-sheets.org/saved-copy/Ssl\_handshake\_with\_two\_way\_authentication\_with\_certificates-1.pdf

阮一峰在他的《SSL/TLS协议运行机制的概述》和《图解SSL/TLS协议》两篇博客中详细介绍了 SSL/TLS 的原理，感兴趣的同学可以去看看。我这里使用IBM Tivoli Risk Manager 用户手册里的一张图（因为这张图比较浅显易懂）来大概的说明下我们在平时使用[浏览器](http://www.2cto.com/os/liulanqi/)访问 HTTPS 站点时，中间发生的握手过程。



整个 SSL/TLS 的握手和通信过程，简单来说，其实可以分成下面三个阶段：

1打招呼

2当用户通过浏览器访问 HTTPS 站点时，浏览器会向服务器打个招呼（ClientHello），服务器也会和浏览器打个招呼（ServerHello）。所谓的打招呼，实际上是告诉彼此各自的 SSL/TLS 版本号以及各自支持的加密算法等，让彼此有一个初步了解。

表明身份、验证身份

第二步是整个过程中最复杂的一步，也是 HTTPS 通信中的关键。为了保证通信的安全，首先要保证我正在通信的人确实就是那个我想与之通信的人，服务器会发送一个证书来表明自己的身份，浏览器根据证书里的 信息进行核实（为什么通过证书就可以证明身份呢？怎么通过证书来验证对方的身份呢？这个后面再说）。如果是双向认证的话，浏览器也会向服务器发送客户端证 书。

双方的身份都验证没问题之后，浏览器会和服务器协商出一个 “对话密钥” ，要注意这个 “对话密钥” 不能直接发给对方，而是要用一种只有对方才能懂的方式发给他，这样才能保证密钥不被别人截获（或者就算被截获了也看不懂）。

通信至此，握手就结束了。双方开始聊天，并通过 “对话密钥” 加密通信的数据。

握手的过程大致如此，我们现在已经了解到 HTTPS 通信需要进行一次握手，所以上面看到的 javax.net.ssl.SSLHandshakeException这个异常，我们也不难理解，实际上也就是在 SSL/TLS 握手的过程中出现了问题。当然，这其中还有很多很多细节，下面继续。

**2.2 HTTPS 中的密码学**

HTTPS 协议之所以复杂，是为了保证通信过程中数据的安全性，而要保证通信安全，它在协议中运用了大量的密码学原理，可以说 HTTPS 是集密码学之大成。无论是在 SSL/TLS 握手的过程中，还是在加密通信的过程中，HTTPS 都涉及了大量的密码学概念，譬如，在证书的数字签名中使用了哈希算法和非对称加密算法，在加密通信的过程中使用了对称加密算法，为了防止传输的数据被篡改 和重放还使用了 MAC（消息认证码）等。

要想深入了解 HTTPS 的工作原理，下面这些概念还是得好好研究下，网上已经有很多文章介绍这些概念了，我在这里总结一下。

哈希

哈希算法又称散列，它是一种将任意长度的数据转化为固定长度的算法

哈希算法是不可逆的

常见的哈希算法有 MD5 和 SHA1

对称加密

对称加密指的是加密和解密使用相同一个密钥

对称加密的优点是速度快，缺点是密钥管理不方便，必须共享密钥

常见的对称加密算法有 DES、AES、Blowfish 等

非对称加密

非对称加密指的是加密和解密使用不同的密钥，其中一个是公钥，另一个是私钥，公钥是公开的，私钥只有自己知道

使用公钥加密的数据必须使用私钥解密，使用私钥加密的数据必须使用公钥解密

公钥和私钥之间存在着某种联系，但是从公钥不能（或很难）推导出私钥

非对称加密的缺点是速度慢，优点是密钥管理很方便

常见的非对称加密算法有 RSA、ECC 等

数字证书

**2.3 关于证书**

简单来说，数字证书就好比介绍信上的公章，有了它，就可以证明这份介绍信确实是由某个公司发出的，而证书可以用来证明任何一个东西的身份，只要这个 东西能出示一份证明自己身份的证书即可，譬如可以用来验证某个网站的身份，可以验证某个文件是否可信等等。《数字证书及 CA 的扫盲介绍》和《数字证书原理》这篇博客对数字证书进行了很通俗的介绍。

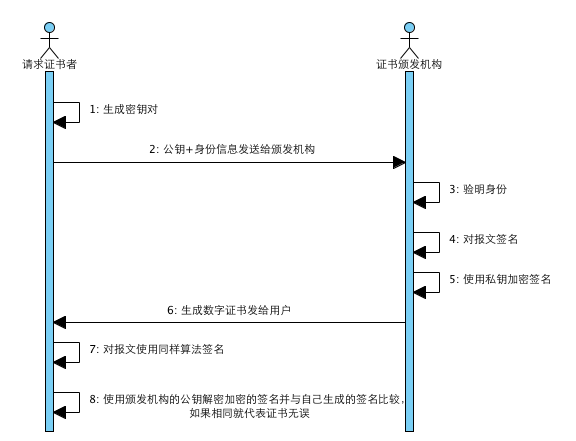
知道了证书是什么之后，我们往往更关心它的原理，在上面介绍 SSL/TLS 握手的时候留了两个问题：为什么通过证书就可以证明身份呢？怎么通过证书来验证对方的身份呢？

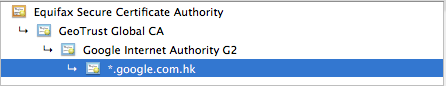
这就要用到上面所说的非对称加密了，非对称加密的一个重要特点是：使用公钥加密的数据必须使用私钥才能解密，同样的，使用私钥加密的数据必须使用公 钥解密。正是因为这个特点，网站就可以在自己的证书中公开自己的公钥，并使用自己的私钥将自己的身份信息进行加密一起公开出来，这段被私钥加密的信息就是 证书的数字签名，浏览器在获取到证书之后，通过证书里的公钥对签名进行解密，如果能成功解密，则说明证书确实是由这个网站发布的，因为只有这个网站知道他 自己的私钥（如果他的私钥没有泄露的话）。

在非对称加密算法中，最出众的莫过于 RSA 算法，关于 RSA 算法的数学细节，可以参考阮一峰的《RSA算法原理（一）》和《RSA算法原理（二）》这两篇博客，强烈推荐。

当然，如果只是简单的对数字签名进行校验的话，还不能完全保证这个证书确实就是网站所有，[黑客](http://www.2cto.com/kf/201609/548236.html)完 全可以在中间进行劫持，使用自己的私钥对网站身份信息进行加密，并将证书中的公钥替换成自己的公钥，这样浏览器同样可以解密数字签名，签名中身份信息也是 完全合法的。这就好比那些地摊上伪造公章的小贩，他们可以伪造出和真正的公章完全一样的出来以假乱真。为了解决这个问题，信息安全的专家们引入了 CA 这个东西，所谓 CA ，全称为 Certificate Authority ，翻译成中文就是证书授权中心，它是专门负责管理和签发证书的第三方机构。因为证书颁发机构关系到所有互联网通信的身份安全，因此一定要是一个非常权威的 机构，像 GeoTrust、GlobalSign 等等，这里有一份常见的 CA 清单。如果一个网站需要支持 HTTPS ，它就要一份证书来证明自己的身份，而这个证书必须从 CA 机构申请，大多数情况下申请数字证书的价格都不菲，不过也有一些免费的证书供个人使用，像最近比较火的Let's Encrypt。从安全性的角度来说，免费的和收费的证书没有任何区别，都可以为你的网站提供足够高的安全性，唯一的区别在于如果你从权威机构购买了付费 的证书，一旦由于证书安全问题导致经济损失，可以获得一笔巨额的赔偿。

如果用户想得到一份属于自己的证书，他应先向 CA 提出申请。在 CA 判明申请者的身份后，便为他分配一个公钥，并且 CA 将该公钥与申请者的身份信息绑在一起，并为之签字后，便形成证书发给申请者。如果一个用户想鉴别另一个证书的真伪，他就用 CA 的公钥对那个证书上的签字进行验证，一旦验证通过，该证书就被认为是有效的。通过这种方式，黑客就不能简单的修改证书中的公钥了，因为现在公钥有了 CA 的数字签名，由 CA 来证明公钥的有效性，不能轻易被篡改，而黑客自己的公钥是很难被 CA 认可的，所以我们无需担心证书被篡改的问题了。

下图显示了证书的申请流程（图片来自刘坤的技术博客）：

CA 证书可以具有层级结构，它建立了自上而下的信任链，下级 CA 信任上级 CA ，下级 CA 由上级 CA 颁发证书并认证。 譬如 Google 的证书链如下图所示：

可以看出：google.com.hk 的 SSL 证书由 Google Internet Authority G2 这个 CA 来验证，而 Google Internet Authority G2 由 GeoTrust Global CA 来验证，GeoTrust Global CA 由 Equifax Secure Certificate Authority 来验证。这个最顶部的证书，我们称之为根证书（root certificate），那么谁来验证根证书呢？答案是它自己，根证书自己证明自己，换句话来说也就是根证书是不需要证明的。浏览器在验证证书时，从根 证书开始，沿着证书链的路径依次向下验证，根证书是整个证书链的安全之本，如果根证书被篡改，整个证书体系的安全将受到威胁。所以不要轻易的相信根证书， 当下次你访问某个网站遇到提示说，请安装我们的根证书，它可以让你访问我们网站的体验更流畅通信更安全时，最好留个心眼。在安装之前，不妨看看这几篇博 客：《12306的证书问题》、《在线买火车票为什么要安装根证书？》。

最后总结一下，其实上面说的这些，什么非对称加密，数字签名，CA 机构，根证书等等，其实都是 PKI 的核心概念。PKI（Public Key Infrastructure）中文称作公钥基础设施，它提供公钥加密和数字签名服务的[系统](http://www.2cto.com/os/)或平台，方便管理密钥和证书，从而建立起一个安全的网络环境。而数字证书最常见的格式是 X.509 ，所以这种公钥基础设施又称之为 PKIX 。

至此，我们大致弄懂了上面的异常信息， sun.security.validator.ValidatorException: PKIX path building failed，也就是在沿着证书链的路径验证证书时出现异常，验证失败了。

讲了这么多，全都是些理论的东西，下面开始实践吧，看看怎么解决这个异常。

### 2.4 关于 Java 里的证书

上面所介绍的是浏览器对证书进行验证的过程，浏览器保存了一个常用的 CA 证书列表，在验证证书链的有效性时，直接使用保存的证书里的公钥进行校验，如果在证书列表中没有找到或者找到了但是校验不通过，那么浏览器会警告用户，由 用户决定是否继续。与此类似的，操作系统也一样保存有一份可信的证书列表，譬如在 Windows 系统下，你可以运行 certmgr.msc打开证书管理器查看，这些证书实际上是存储在 Windows 的注册表中，一般情况下位于： \SOFTWARE\Microsoft\SystemCertificates\路径下。那么在 Java 程序中是如何验证证书的呢？

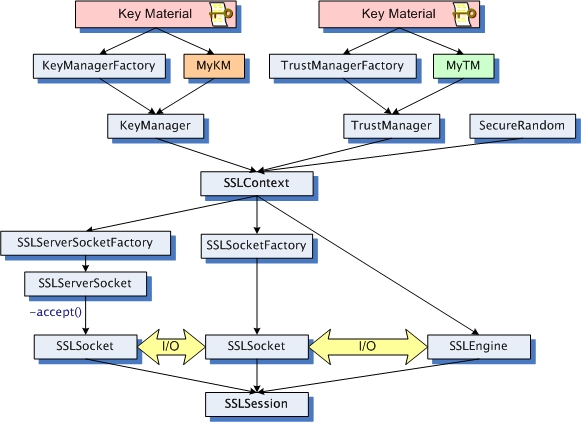
和浏览器和操作系统类似，Java 在 JRE 的安装目录下也保存了一份默认可信的证书列表，这个列表一般是保存在 $JRE/lib/security/cacerts文件中。要查看这个文件，可以使用类似KeyStore Explorer这样的软件，当然也可以使用 JRE 自带的 keytool 工具（后面再介绍），cacerts 文件的默认密码为 changeit（但是我保证，大多数人都不会 change it）。

我们知道，证书有很多种不同的存储格式，譬如 CA 在发布证书时，常常使用 PEM 格式，这种格式的好处是纯文本，内容是 BASE64 编码的，证书中使用 "-----BEGIN CERTIFICATE-----" 和 "-----END CERTIFICATE-----" 来标识。另外还有比较常用的二进制 DER 格式，在 Windows 平台上较常使用的 PKCS#12 格式等等。当然，不同格式的证书之间是可以相互转换的，我们可以使用 openssl这个命令行工具来转换，参考SSL Converter，另外，想了解更多证书格式的，可以参考这里：Various SSL/TLS Certificate File Types/Extensions。

在 Java 平台下，证书常常被存储在 KeyStore 文件中，上面说的 cacerts 文件就是一个 KeyStore 文件，KeyStore 不仅可以存储数字证书，还可以存储密钥，存储在 KeyStore 文件中的对象有三种类型：Certificate、PrivateKey 和 SecretKey 。Certificate 就是证书，PrivateKey 是非对称加密中的私钥，SecretKey 用于对称加密，是对称加密中的密钥。KeyStore 文件根据用途，也有很多种不同的格式：JKS、JCEKS、PKCS12、DKS 等等，PixelsTech 上有一系列文章对 KeyStore 有深入的介绍，可以学习下：Different types of keystore in Java。

到目前为止，我们所说的 KeyStore 其实只是一种文件格式而已，实际上在 Java 的世界里 KeyStore 文件分成两种：KeyStore 和 TrustStore，这是两个比较容易混淆的概念，不过这两个东西从文件格式来看其实是一样的。KeyStore 保存私钥，用来加解密或者为别人做签名；TrustStore 保存一些可信任的证书，访问 HTTPS 时对被访问者进行认证，以确保它是可信任的。所以准确来说，上面的 cacerts 文件应该叫做 TrustStore 而不是 KeyStore，只是它的文件格式是 KeyStore 文件格式罢了。

除了 KeyStore 和 TrustStore ，Java 里还有两个类 KeyManager 和 TrustManager 与此息息相关。JSSE 的参考手册中有一张示意图，说明了各个类之间的关系：



可以看出如果要进行 SSL 会话，必须得新建一个 SSLSocket对象，而 SSLSocket 对象是通过 SSLSocketFactory来管理的，SSLSocketFactory 对象则依赖于 SSLContext，SSLContext 对象又依赖于 keyManager、 TrustManager和 SecureRandom。我们这里最关心的是 TrustManager 对象，另外两个暂且忽略，因为正是 TrustManager 负责证书的校验，对网站进行认证，要想在访问 HTTPS 时通过认证，不报 sun.security.validator.ValidatorException异常，必须从这里开刀。

## 三、自定义 TrustManager 绕过证书检查

|  |  |
| --- | --- |
|  |  |

我们新建了一个匿名类，继承自 X509TrustManager 接口，这个接口提供了三个方法用于验证证书的有效性： getAcceptedIssuers、 checkClientTrusted、 checkServerTrusted，我们在验证的函数中直接返回，不做任何校验，这样在访问 HTTPS 站点时，就算是证书不可信，也不会抛出异常，可以继续执行下去。

这种方法虽然简单，但是却有一个最严重的问题，就是不安全。因为不对证书做任何合法性校验，而且这种处理是全局性的，不管青红皂白，所有的证书都不 会做验证，所以就算遇到不信任的证书，代码依然会继续与之通信，至于通信的数据安全不安全就不能保证了。所以如果你只是想在测试环境做个实验，那没问题， 但是如果你要将代码发布到生产环境，请慎重。

## 四、使用证书

## 对于有些证书，我们基本上确定是可以信任的，但是这些证书又不在 Java 的 cacerts 文件中，譬如 12306 网站，或者使用了 Let's Encrypt 证书的一些网站，对于这些网站，我们可以将其添加到信任列表中，而不是使用上面的方法统统都相信，这样程序的安全性仍然可以得到保障。

## 4.1 使用 keytool 导入证书

简单的做法是将这些网站的证书导入到 cacerts 文件中，这样 Java 程序在校验证书的时候就可以从 cacerts 文件中找到并成功校验这个证书了。上面我们介绍过 JRE 自带的 keytool 这个工具，这个工具小巧而强悍，拥有很多功能。首先我们可以使用它查看 KeyStore 文件，使用下面的命令可以列出 KeyStore 文件中的所有内容（包括证书、私钥等）：

[?](http://www.2cto.com/kf/201609/548236.html)

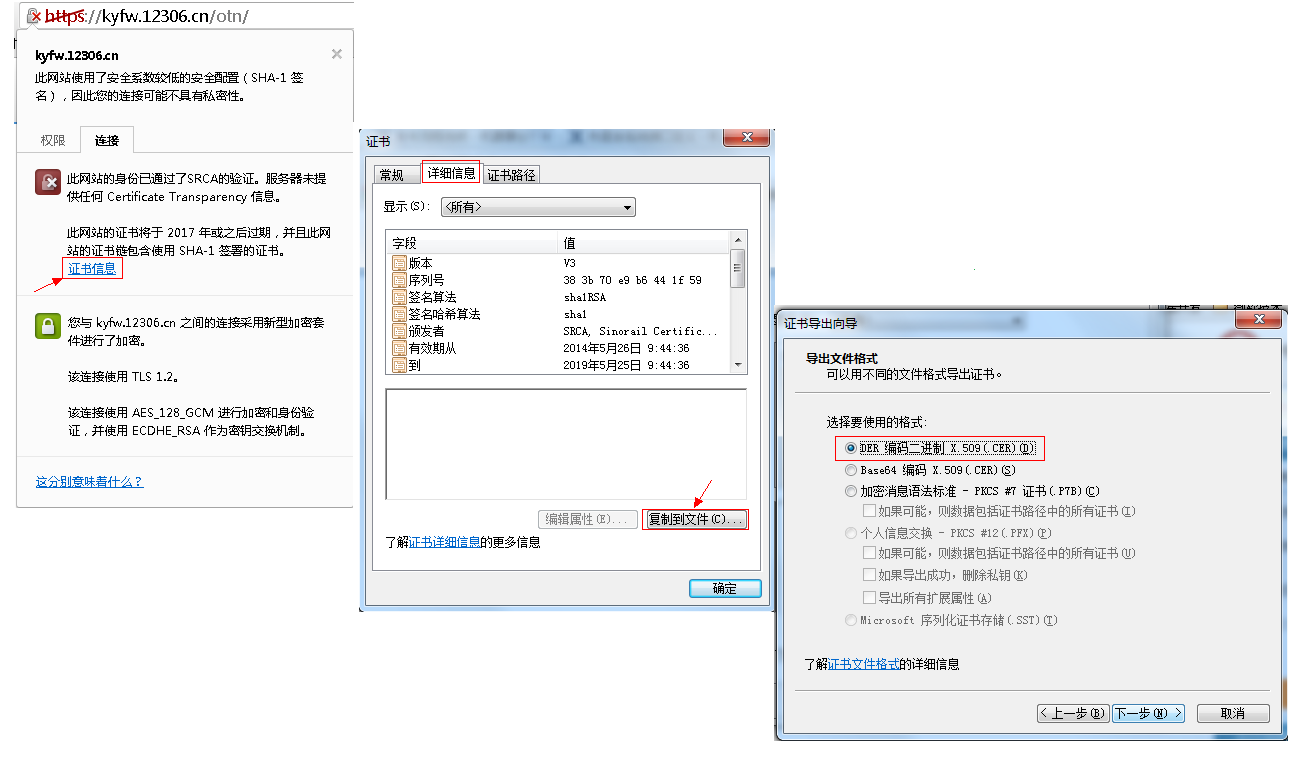
|  |  |
| --- | --- |
| 1 | $ keytool -list -keystore cacerts |

然后通过下面的命令，将证书导入到 cacerts 文件中：

[?](http://www.2cto.com/kf/201609/548236.html)

|  |  |
| --- | --- |
| 1 | $ keytool -import -alias 12306 -keystore cacerts -file 12306.cer |

要想将网站的证书导入 cacerts 文件中，首先要获取网站的证书，譬如上面命令中的 12306.cer 文件，它是使用浏览器的证书导出向导保存的。如下图所示：



关于 keytool 的更多用法，可以参考keytool 的官网手册，SSLShopper 上也有一篇文章列出了常用的 keytool 命令。**4.2 使用 KeyStore 动态加载证书**

使用 keytool 导入证书，这种方法不仅简单，而且保证了代码的安全性，最关键的是代码不用做任何修改。所以我比较推荐这种方法。但是这种方法有一个致命的缺陷，那就是你 需要修改 JRE 目录下的文件，如果你的程序只是在自己的电脑上运行，那倒没什么，可如果你的程序要部署在其他人的电脑上或者公司的服务器上，而你没有权限修改 JRE 目录下的文件，这该怎么办？如果你的程序是一个分布式的程序要部署在成百上千台机器上，难道还得修改每台机器的 JRE 文件吗？好在我们还有另一种通过[编程](http://www.2cto.com/kf)的 手段来实现的思路，在代码中动态的加载 KeyStore 文件来完成证书的校验，抱着知其然知其所以然的态度，我们在最后也实践下这种方法。通过编写代码可以更深刻的了解 KeyStore、 TrustManagerFactory、 SSLContext以及 SSLSocketFactory这几个类之间的关系。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 | @Test **publicvoid** basicHttpsGetUsingSslSocketFactory() **throws**Exception { String keyStoreFile = "D:\\code\\ttt.ks"; String password = "poiuyt"; KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType()); FileInputStream in = **new**FileInputStream(keyStoreFile); ks.load(in, password.toCharArray()); System.out.println(KeyStore.getDefaultType().toString()); System.out.println(TrustManagerFactory.getDefaultAlgorithm().toString()); TrustManagerFactory tmf = TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm()); tmf.init(ks); SSLContext ctx = SSLContext.getInstance("TLS"); ctx.init(**null**, tmf.getTrustManagers(), **null**); LayeredConnectionSocketFactory sslSocketFactory = **new**SSLConnectionSocketFactory(ctx); String url = "https://ttt.aneasystone.com"; /\*\* \* Return the page with content: \* 401 Authorization Required \*/ CloseableHttpClient httpclient = HttpClients.custom() .setSSLSocketFactory(sslSocketFactory) .build(); HttpGet request = **new**HttpGet(url); request.setHeader("User-Agent","Mozilla/5.0 (Windows NT 6.1; WOW64) ..."); CloseableHttpResponse response = httpclient.execute(request); String responseBody = readResponseBody(response); System.out.println(responseBody); } |

上面的代码使用了 HttpClient ，如果是使用 HttpsURLConnection 只需要改动下面两行即可：

|  |  |
| --- | --- |
| 1 2 | HttpsURLConnection con = (HttpsURLConnection) obj.openConnection(); con.setSSLSocketFactory(ctx.getSocketFactory()); |

最后的最后，我们还可以通过下面的属性来指定 trustStore ，这样也不需要编写像上面那样大量繁琐的代码，另外，参考我前面的博客，这些属性还可以通过 JVM 的参数来设置。

|  |  |
| --- | --- |
| 1 2 | System.setProperty("javax.net.ssl.trustStore","D:\\code\\ttt.ks"); System.setProperty("javax.net.ssl.trustStorePassword","poiuyt"); |

**小结**

至此，我们的 HTTPS 之旅就要告一段落了。在学习 HTTPS 的过程中，我时时不经意的会被 HTTPS 中的一些技术或技巧感触到，特别是证书的认证过程以及非对称加密算法的原理，真的是积累了人类无穷的智慧，让人不得不感叹数学的美妙。而在学习过程中越是 刨根问底，越是一发不可收拾，中间牵扯到的细节太多，太深入反而让人不自觉的迷失了方向。