java ssl https 连接详解 生成证书

我们先来了解一下什么理HTTPS

1. HTTPS概念

HTTPS（全称：Hypertext Transfer Protocol over Secure Socket Layer），是以安全为目标的HTTP通道，简单讲是HTTP的安全版。即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。这个系统的最初研发由网景公司进行，提供了身份验证与加密通讯方法，现在它被广泛用于万维网上安全敏感的通讯，例如交易支付方面。

2）HTTPS和HTTP的区别

a. https协议需要到ca申请证书，一般免费证书很少，需要交费。

b. http是超文本传输协议，信息是明文传输；https 则是具有安全性的ssl加密传输协议。

c. http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

d. http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

3）HTTPS的作用

它的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

a． 一般意义上的https，就是服务器有一个证书。主要目的是保证服务器就是他声称的服务器，这个跟第一点一样；服务端和客户端之间的所有通讯，都是加密的。

b. 具体讲，是客户端产生一个对称的密钥，通过服务器的证书来交换密钥，即一般意义上的握手过程。

c. 接下来所有的信息往来就都是加密的。第三方即使截获，也没有任何意义，因为他没有密钥，当然篡改也就没有什么意义了。

d．少许对客户端有要求的情况下，会要求客户端也必须有一个证书。

这里客户端证书，其实就类似表示个人信息的时候，除了用户名/密码，还有一个CA 认证过的身份。因为个人证书一般来说是别人无法模拟的，所有这样能够更深的确认自己的身份。目前少数个人银行的专业版是这种做法，具体证书可能是拿U盘（即U盾）作为一个备份的载体。

2.SSL简介

SSL (Secure Socket Layer)为Netscape所研发，用以保障在Internet上数据传输之安全，利用数据加密(Encryption)技术，可确保数据在网络上之传输过程中不会被截取及窃听。它已被广泛地用于Web浏览器与服务器之间的身份认证和加密数据传输。SSL协议位于TCP/IP协议与各种应用层协议之间，为数据通讯提供安全支持。

2) SSL提供的服务 www.2cto.com [1]

a.认证用户和服务器，确保数据发送到正确的客户机和服务器

b.加密数据以防止数据中途被窃取

c.维护数据的完整性，确保数据在传输过程中不被改变。

3) SSL协议的握手过程

SSL 协议既用到了公钥加密技术又用到了对称加密技术，对称加密技术虽然比公钥加密技术的速度快，可是公钥加密技术提供了更好的身份认证技术。SSL 的握手协议非常有效的让客户和服务器之间完成相互之间的身份认证，其主要过程如下：

①客户端的浏览器 [2]向服务器传送客户端SSL 协议的版本号，加密算法的种类，产生的随机数，以及其他服务器和客户端之间通讯所需要的各种信息。

②服务器向客户端传送SSL 协议的版本号，加密算法的种类，随机数以及其他相关信息，同时服务器还将向客户端传送自己的证书。

③客户利用服务器传过来的信息验证服务器的合法性，服务器的合法性包括：证书

是否过期，发行服务器证书的CA 是否可靠，发行者证书的公钥能否正确解开服务器证书的“发行者的数字签名”，服务器证书上的域名是否和服务器的实际域名相匹配。如果合法性验证没有通过，通讯将断开；如果合法性验证通过，将继续进行第四步。

④用户端随机产生一个用于后面通讯的“对称密码”，然后用服务器的公钥（服务器的公钥从步骤②中的服务器的证书中获得）对其加密，然后传给服务器。

⑤服务器用私钥解密“对称密码”(此处的公钥和私钥是相互关联的，公钥加密的数据只能用私钥解密，私钥只在服务器端保留。详细请参看：http://zh.wikipedia.org/wiki/RSA%E7%AE%97%E6%B3%95)，然后用其作为服务器和客户端的“通话密码”加解密通讯。同时在SSL 通讯过程中还要完成数据通讯的完整性，防止数据通讯中的任何变化。

⑥客户端向服务器端发出信息，指明后面的数据通讯将使用的步骤⑤中的主密码为对称密钥，同时通知服务器客户端的握手过程结束。

⑦服务器向客户端发出信息，指明后面的数据通讯将使用的步骤⑤中的主密码为对称密钥，同时通知客户端服务器端的握手过程结束。

⑧SSL 的握手部分结束，SSL 安全通道的数据通讯开始，客户和服务器开始使用相同的对称密钥进行数据通讯，同时进行通讯完整性的检验。

3.配置服务器端证书

为了能实施SSL，一个web服务器对每个接受安全连接的外部接口(IP 地址)必须要有相应的证书(Certificate)。关于这个设计的理论是一个服务器必须提供某种合理的保证以证明这个服务器的主人就是你所认为的那个人。这个证书要陈述与这个网站相关联的公司，以及这个网站的所有者或系统管理员的一些基本联系信息。

这个证书由所有人以密码方式签字，其他人非常难伪造。对于进行电子商务(e-commerce)的网站，或其他身份认证至关重要的任何商业交易，认证书要向大家所熟知的认证权威(Certificate Authority (CA))如VeriSign或Thawte来购买。这样的证书可用电子技术证明属实。实际上，认证权威单位会担保它发出的认证书的真实性，如果你信任发出认证书的认证权威单位的话，你就可以相信这个认证书是有效的。

关于权威证书的申请，请参考：http://www.cnblogs.com/mikespook/archive/2004/12/22/80591.aspx

在许多情况下，认证并不是真正使人担忧的事。系统管理员或许只想要保证被服务器传送和接收的数据是秘密的，不会被连接线上的偷窃者盗窃到。庆幸的是，Java提供相对简单的被称为keytool的命令行工具，可以简单地产生“自己签名”的证书。自己签名的证书只是用户产生的证书，没有正式在大家所熟知的认证权威那里注册过，因此不能确保它的真实性。但却能保证数据传输的安全性。

用Tomcat来配置SSL主要有下面这么两大步骤：

1)生成证书

a. 在命令行下执行：

%Java_home%\bin\keytool -genkey -alias tomcat -keyalg RSA

在此命令中，keytool是JDK自带的产生证书的工具。把RSA运算法则作为主要安全运算法则，这保证了与其它服务器和组件的兼容性。

这个命令会在用户的home directory产生一个叫做" .keystore " 的新文件。在执行后，你首先被要求出示keystore密码。Tomcat使用的默认密码是" changeit "(全都是小写字母)，如果你愿意，你可以指定你自己的密码。你还需要在server.xml配置文件里指定自己的密码，这在以后会有描述。

b .你会被要求出示关于这个认证书的一般性信息，如公司，联系人名称，等等。这些信息会显示给那些试图访问你程序里安全网页的用户，以确保这里提供的信息与他们期望的相对应。

c.你会被要求出示密钥(key)密码，也就是这个认证书所特有的密码(与其它的储存在同一个keystore文件里的认证书不同)。你必须在这里使用与keystore密码相同的密码。(目前，keytool会提示你按ENTER键会自动帮你做这些)。

如果一切顺利，你现在就拥有了一个可以被你的服务器使用的有认证书的keystore文件

首先我们要用KEYTOOL生成证书

Java 中的 keytool.exe （位于 JDK\Bin 目录下）可以用来创建数字证书，所有的数字证书是以一条一条(采用别名区别)的形式存入证书库的中，证书库中的一条证书包含该条证书的私钥，公钥和对应的数字证书的信息。证书库中的一条证书可以导出数字证书文件，数字证书文件只包括主体信息和对应的公钥。

Keytool是一个Java数据证书的管理工具。

keystore
Keytool将密钥（key）和证书（certificates）存在一个称为keystore的文件中在keystore里，包含两种数据：
密钥实体（Key entity）——密钥（secret key）又或者是私钥和配对公钥（采用非对称加密）
可信任的证书实体（trusted certificate entries）——只包含公钥

Alias（别名）
每个keystore都关联这一个独一无二的alias，这个alias通常不区分大小写

keystore的存储位置
在没有制定生成位置的情况下，keystore会存在与用户的系统默认目录，
如：对于window xp系统，会生成在系统的C:\Documents and Settings\UserName\
文件名为“.keystore”

keystore的生成

引用keytool -genkey -alias tomcat -keyalg RSA -keystore d:\mykeystore -dname "CN=localhost, OU=localhost, O=localhost, L=SH, ST=SH, C=CN" -keypass changeit -storepass -validity 180

参数说明：
-genkey表示要创建一个新的密钥
-dname表示密钥的Distinguished Names，
CN=commonName
OU=organizationUnit
O=organizationName
L=localityName
S=stateName
C=country
Distinguished Names表明了密钥的发行者身份
-keyalg使用加密的算法，这里是RSA
-alias密钥的别名
-keypass私有密钥的密码，这里设置为changeit
-keystore 密钥保存在D:盘目录下的mykeystore文件中
-storepass 存取密码，这里设置为changeit，这个密码提供系统从mykeystore文件中将信息取出

-validity该密钥的有效期为 180天 (默认为90天)

1，产生一个密钥对

keytool -genkey -alias mykeypair -keypass mykeypairpwd

过程如下：

liqingfeng@liqingfeng:~/WORK_APP/keytooltest$ keytool -genkey -alias mykeypair -keypass mykeypairpwd

输入keystore密码： 123456

您的名字与姓氏是什么？

[Unknown]： fingki

您的组织单位名称是什么？

[Unknown]： server

您的组织名称是什么？

[Unknown]： server

您所在的城市或区域名称是什么？

[Unknown]： bj

您所在的州或省份名称是什么？

[Unknown]： bj

该单位的两字母国家代码是什么

[Unknown]： CN

CN=fingki, OU=server, O=server, L=bj, ST=bj, C=CN 正确吗？

[否]： y

liqingfeng@liqingfeng:~/WORK_APP/keytooltest$

这样将产生一个keypair,同时产生一个keystore.默认名是.keystore,存放到user-home目录

假如你想修改密码,可以用:keytool -keypasswd -alias mykeypair -keypass mykeypairpwd -new newpass

2，产生一个密钥对，存放在指定的keystore中（加上-keystore 参数）

keytool -genkey -alias mykeypair -keypass mykeypairpwd -keystore mykeystore

过程与上面的相同。

执行完后，在当前目录下产生一个名为mykeystore的keystore，里面有一个别名为mykeypair的keypair。

3，检查一个keystore中的内容

keytool -list -v -alias mykeypair -keystore mykeystore

参数 -v指明要列出详细信息

-alias指明列出指定的别名为mykeypair的keypair信息（不指定则列出所

有）

-keystore指明要列出名字为mykeystore的keystore中的信息

过程如下：

liqingfeng@liqingfeng:~/WORK_APP/keytooltest$ keytool -list -v -keystore mykeystore

输入keystore密码： 123456

Keystore 类型： jks
Keystore 提供者： SUN

您的 keystore 包含 1 输入

别名名称： mykeypair
创建日期： 2008-4-16
输入类型：KeyEntry
认证链长度： 1
认证 [1]:
Owner: CN=fingki, OU=server, O=server, L=bj, ST=bj, C=CN
发照者： CN=fingki, OU=server, O=server, L=bj, ST=bj, C=CN
序号： 48058c3c
有效期间： Wed Apr 16 13:18:52 GMT+08:00 2008 至： Tue Jul 15 13:18:52 GMT+08:00 2008
认证指纹：
MD5 [3]： FD:C3:97:DC:84:A0:D8:B2:08:6F:26:7F:31:33:C3:05
SHA1： A3:21:6F:C6:FB:5F:F5:2D:03:DA:71:8C:D3:67:9D:1C:E1:27:A5:11

****
****

liqingfeng@liqingfeng:~/WORK_APP/keytooltest$]

4，Keystore的产生:

当使用-genkey 或-import或-identitydb命令添加数据到一个keystore,而当这个keystore不存在时,产生一个keystore.默认名是.keystore,存放到user-home目录.
当用-keystore指定时,将产生指定的keystore.

5，Keystore的实现:
Keytool 类位于java.security包下,提供一个非常好的接口去取得和修改一个keystore中的信息. 目前有两个命令行:keytool和jarsinger,一个GUI工具Policy 可以

实现keystore.由于keystore是公开的,用户可以用它写一些额外的安全应用程序.

Keystore还有一个sun公司提供的内在实现.它把keystore作为一个文件来实现.利用了一个keystore类型(格式)"JKS".它用单独的密码保护每一个私有钥匙.也用可能不同的密码保护整个keystore的完整性.

支持的算法和钥匙大小:

keytool允许用户指定钥匙对和注册密码服务供应者所提供的签名算法.缺省的钥匙对产生算法是"DSA".假如私有钥匙是"DSA"类型,缺省签名算法是"SHA1withDSA",假如私有钥匙是"RSA"类型,缺省算法是"MD5withRSA".

当产生一个DSA钥匙对,钥匙必须在512-1024位之间.对任何算法的缺省钥匙大小是1024位.

6，关于证书

一个证书是一个实体的数字签名,还包含这个实体的公共钥匙值.

公共钥匙 :是一个详细的实体的数字关联,并有意让所有想同这个实体发生信任关系的其他实体知道.公共钥匙用来检验签名;

数字签名:是实体信息用实体的私有钥匙签名（加密）后的数据.这条数据可以用这个实体的公共钥匙来检验签名（解密）出实体信息以鉴别实体的身份;

签名:用实体私有钥匙加密某些消息，从而得到加密数据;

私有钥匙:是一些数字,私有和公共钥匙存在所有用公共钥匙加密的系统的钥匙对中.公共钥匙用来加密数据，私有钥匙用来计算签名.公钥加密的消息只能用私钥解密，私钥签名的消息只能用公钥检验签名。

实体:一个实体可以是一个人,一个组织,一个程序,一台计算机,一个商业,一个银行,或其他你想信任的东西.

实际上，我们用 [1] 中的命令已经生成了一个自签名的证书，没有指定的参数都使用的是默认值。

我们也可以用如下命令生成一个自签名的证书：

keytool -genkey -dname "CN=fingki,OU=server,O=server,L=bj,ST=bj,C=CN" -alias myCA -keyalg RSA -keysize 1024 -keystore myCALib -keypass 654321 -storepass 123456 -validity 3650

这条命令将生成一个别名为myCA的自签名证书，证书的keypair的密码为654321，证书中实体信息为 "CN=fingki,OU=server,O=server,L=bj,ST=bj,C=CN"，存储在名为myCALib的keystore中（如果 没有将自动生成一个），这个keystore的密码为123456，密钥对产生的算法指定为RSA，有效期为10年。

7，将证书导出到证书文件

keytool -export -alias myCA -file myCA.cer -keystore myCALib -storepass 123456 -rfc

使用该命令从名为myCALib的keystore中，把别名为myCA的证书导出到证书文件myCA.cer中。(其中-storepass指定keystore的密码，-rfc指定以可查看编码的方

式输出，可省略)。

8，通过证书文件查看证书信息

keytool -printcert -file myCA.cer

9，密钥库中证书条目口令的修改

Keytool -keypasswd -alias myCA -keypass 654321 -new newpass -storepass 123456 -keystore myCALib

10，删除密钥库中的证书条目

keytool -delete -alias myCA -keystore myCALib

11，把一个证书文件导入到指定的密钥库

keytool -import -alias myCA -file myCA.cer -keystore truststore

(如果没有名为truststore的keystore，将自动创建,将会提示输入keystore的密码)

12，更改密钥库的密码

keytool -storepasswd -new 123456 -storepass 789012 -keystore truststore

其中-storepass指定原密码，-new指定新密码。

13，将[keystore]导入java信任证书库

keytool -import -trustcacerts -alias tomcat_pso -file [keystore] -keypass changeit -keystore "%JAVA_HOME%/jre/lib/security/cacerts"

注：%JAVA_HOME%/jre/lib/security/cacerts为java自带的证书库，默认密码为changeit

keytool -list -v -keystore c:/jdk15/jre/lib/security/cacerts (列出信任库中已经存在的证书)

keytool -delete -trustcacerts -alias tomcat -keystore c:/jdk15/jre/lib/security/cacerts -storepass changeit(删除某一个证书)

8.配置tomcat

第二个大步骤是把secure socket配置在$CATALINA_HOME/conf/server.xml文件里。$CATALINA_HOME代表安装Tomcat的目录。一个例子是SSL连接器的元素被包括在和Tomcat一起安装的缺省server.xml文件里。它看起来象是这样：$CATALINA_HOME/conf/server.xml

[html]

< -- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->

< !--

< Connector

port="8443" minProcessors="5" maxProcessors="75"

enableLookups="true" disableUploadTimeout="true"

acceptCount="100" debug="0" scheme="https" secure="true";

clientAuth="false" sslProtocol="TLS"/>

-->

Connector元素本身，其默认形式是被注释掉的(commented out)，所以需要把它周围的注释标志删除掉。然后，可以根据需要客户化(自己设置)特定的属性。一般需要增加一下keystoreFile和keystorePass两个属性，指定你存放证书的路径（如：keystoreFile="C:/.keystore"）和刚才设置的密码（如：keystorePass="123456"）。关于其它各种选项的详细信息，可查阅Server Configuration Reference。

在完成这些配置更改后，必须象重新启动Tomcat，然后你就可以通过SSL访问Tomcat支持的任何web应用程序。只不过指令需要像下面这样：https://localhost:8443

客户端代码实现

在Java中要访问Https链接时，会用到一个关键类HttpsURLConnection；参见如下实现代码：

[java]
// 创建URL对象
URL myURL = new URL("https://www.sun.com");

// 创建HttpsURLConnection对象，并设置其SSLSocketFactory对象
HttpsURLConnection httpsConn = (HttpsURLConnection) myURL.openConnection();

// 取得该连接的输入流，以读取响应内容
InputStreamReader insr = new InputStreamReader(httpsConn.getInputStream());

// 读取服务器的响应内容并显示

```
int respInt = insr.read();
while (respInt != -1) {
   System.out.print((char) respInt);
   respInt = insr.read();
}
```

在取得connection的时候和正常浏览器访问一样，仍然会验证服务端的证书是否被信任（权威机构发行或者被权威机构签名）；如果服务端证书不被信任，则默认的实现就会有问题，一般来说，用SunJSSE会抛如下异常信息：

```
[java]
javax.net.ssl.SSLHandshakeException:
sun.security.validator.ValidatorException: PKIX path building
```

```
[java]
failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to
find valid certification path to requested target
```

上面提到SunJSSE，JSSE（Java Secure Socket Extension）是实现Internet安全通信的一系列包的集合。它是一个SSL和TLS的纯Java实现，可以透明地提供数据加密、服务器认证、信息完整性等功能，可以使我们像使用普通的套接字一样使用JSSE建立的安全套接字。JSSE是一个开放的标准，不只是Sun公司才能实现一个SunJSSE，事实上其他公司有自己实现的JSSE，然后通过JCA就可以在JVM中使用。

关于JSSE的详细信息参考官网Reference：
http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html；
以及Java Security Guide：http://java.sun.com/j2se/1.5.0/docs/guide/security/；

在深入了解JSSE之前，需要了解一个有关Java安全的概念：客户端的TrustStore文件。客户端的TrustStore文件中保存着被客户端所信任的服务器的证书信息。客户端在进行SSL连接时，JSSE将根据这个文件中的证书决定是否信任服务器端的证书。在SunJSSE中，有一个信任管理器类负责决定是否信任远端的证书，这个类有如下的处理规则：
1)若系统属性javax.net.sll.trustStore指定了TrustStore文件，那么信任管理器就去jre安装路径下的lib/security/目录中寻找并使用这个文件来检查证书。
2)若该系统属性没有指定TrustStore文件，它就会去jre安装路径下寻找默认的TrustStore文件，这个文件的相对路径为：lib/security/jssecacerts。
3)若jssecacerts不存在，但是cacerts存在（它随J2SDK一起发行，含有数量有限的可信任的基本证书），那么这个默认的TrustStore文件就是lib/security/cacerts。

那遇到这种情况，怎么处理呢？有以下两种方案：

1)按照以上信任管理器的规则，将服务端的公钥导入到jssecacerts，或者是在系统属性中设置要加载的trustStore文件的路径；证书导入可以用如下命令：keytool -import -file src_cer_file –keystore dest_cer_store；至于证书可以通过浏览器导出获得；

2)、实现自己的证书信任管理器类，比如MyX509TrustManager，该类必须实现X509TrustManager接口中的三个method；然后在HttpsURLConnection中加载自定义的类，可以参见如下两个代码片段，其一为自定义证书信任管理器，其二为connect时的代码：

```
[java]
package test;
import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;
import javax.net.ssl.TrustManager;
import javax.net.ssl.TrustManagerFactory;
import javax.net.ssl.X509TrustManager;
public class MyX509TrustManager implements X509TrustManager {
  /*
   * The default X509TrustManager returned by SunX509.  We'll delegate
   * decisions to it, and fall back to the logic in this class if the
   * default X509TrustManager doesn't trust it.
   */
  X509TrustManager sunJSSEX509TrustManager;
  MyX509TrustManager() throws Exception {
    // create a "default" JSSE X509TrustManager.
    KeyStore ks = KeyStore.getInstance("JKS");
    ks.load(new FileInputStream("trustedCerts"),
       "passphrase".toCharArray());
    TrustManagerFactory tmf =
    TrustManagerFactory.getInstance("SunX509", "SunJSSE");
    tmf.init(ks);
    TrustManager tms [] = tmf.getTrustManagers();
    /*
     * Iterate over the returned trustmanagers, look
     * for an instance of X509TrustManager.  If found,
     * use that as our "default" trust manager.
     */
```

```
        for (int i = 0; i < tms.length; i++) {
            if (tms[i] instanceof X509TrustManager) {
                sunJSSEX509TrustManager = (X509TrustManager) tms[i];
                return;
            }
        }
        /*
         * Find some other way to initialize, or else we have to fail the
         * constructor.
         */
        throw new Exception("Couldn't initialize");
    }
    /*
     * Delegate to the default trust manager.
     */
    public void checkClientTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        try {
            sunJSSEX509TrustManager.checkClientTrusted(chain, authType);
        } catch (CertificateException excep) {
            // do any special handling here, or rethrow exception.
        }
    }
    /*
     * Delegate to the default trust manager.
     */
    public void checkServerTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        try {
            sunJSSEX509TrustManager.checkServerTrusted(chain, authType);
        } catch (CertificateException excep) {
            /*
             * Possibly pop up a dialog box asking whether to trust the
             * cert chain.
             */
        }
    }
    /*
     * Merely pass this through.
     */
```

```
    public X509Certificate[] getAcceptedIssuers() {
      return sunJSSEX509TrustManager.getAcceptedIssuers();
    }
  }
      // 创建SSLContext对象，并使用我们指定的信任管理器初始化
      TrustManager[] tm = { new MyX509TrustManager() };
      SSLContext sslContext = SSLContext.getInstance("SSL", "SunJSSE");
      sslContext.init(null, tm, new java.security.SecureRandom());
      // 从上述SSLContext对象中得到SSLSocketFactory对象
      SSLSocketFactory ssf = sslContext.getSocketFactory();
      // 创建URL对象
      URL myURL = new
URL("https://ebanks.gdb.com.cn/sperbank/perbankLogin.jsp [4]");
      // 创建HttpsURLConnection对象，并设置其SSLSocketFactory对象
      HttpsURLConnection httpsConn = (HttpsURLConnection)
myURL.openConnection();
      httpsConn.setSSLSocketFactory(ssf);
      // 取得该连接的输入流，以读取响应内容
      InputStreamReader insr = new
InputStreamReader(httpsConn.getInputStream());
      // 读取服务器的响应内容并显示
      int respInt = insr.read();
      while (respInt != -1) {
        System.out.print((char) respInt);
        respInt = insr.read();
      }
```

对于以上两种实现方式，各有各的优点，第一种方式不会破坏JSSE的安全性，但是要手工导入证书，如果服务器很多，那每台服务器的JRE都必须做相同的操作；第二种方式灵活性更高，但是要小心实现，否则可能会留下安全隐患；

1. http://www.2cto.com/
2. http://www.2cto.com/os/liulanqi/
3. http://www.tmd5.com/
4. http://www.2cto.com/kf/web/jsp/