# 上級JSSE開発者のためのカスタムSSL

## JSSEを使用して、SSL接続のプロパティーをカスタマイズする

Ian Parkinson (ianp@hursley.ibm.com)
Software Engineer, WebSphere MQ
IBM

2002年 9月 01日

JSSEは、データがネットワークを介して伝送される際に、SSLを使用してデータを暗号化し保護することによって、Javaアプリケーションにセキュアな通信を提供します。Javaミドルウェア開発者のIan Parkinsonは、このテクノロジーについて高度な観点から、JSSE APIのあまり知られていない側面について掘り下げ、SSLの制限のいくつかを回避するプログラミング方法を示しています。KeyStoreとTrustStoreの動的な選択やJSSEのパスワード・マッチング要件の緩和、カスタマイズされた独自のKeyManager実装の構築を行う方法を学びましょう。

JSSE (Java Secure Socket Extension) の機能により、Javaアプリケーションは、SSLを使用してインターネット上でセキュアに通信することができます。developerWorksにはすでにJSSEの基本的な使用法について取り上げたチュートリアルがあるので (参考文献を参照)、ここでは、このテクノロジーのさらに高度な使用法を紹介しましょう。この記事では、JSSEインターフェースを使用して、SSL接続のプロパティーをカスタマイズする方法について説明します。

まず、非常に簡単なセキュア・クライアント / サーバーであるチャット・アプリケーションの作成から始めましょう。このアプリケーションのクライアント側を作成するにあたり、クライアントのファイルシステムからKeyStoreおよびTrustStoreファイルをロードできるよう、これらのファイルをカスタマイズする方法を説明します。次に、証明書と識別について取り上げます。KeyStoreから異なる証明書を選択することによって、サーバー毎にクライアントとしてのふるまいを変えることができます。このような高度な機能は、特に、クライアント・アプリケーションが複数のピアに接続する必要がある場合、または異なるユーザーになりすます必要のある場合に役立ちます。

この記事では、より高度なトピックについて取り上げているため、以前にJSSEを使用したことのある読者を対象にしています。例を実行するには、JSSEプロバイダーが正しくインストールおよび設定された、Java SDKが必要です。J2SE 1.4 SDKには、JSSEがあらかじめインストール、構成されています。J2SE 1.2または1.3を使用する場合は、JSSE実装を取得し、インストールする必要があります。参考文献を参照して、JSSE拡張をダウンロードしてください。

JSSE APIは、J2SE 1.4で初めて標準となりました。また以前のJSSE実装とはわずかに差異があります。この記事の例は1.4 APIに基づいています。これらの例をJ2SE 1.2または1.3のSunのJSSE実装で、これらの例を使用するために、必要な変更があれば、そのつど指示することにします。

## はじめに: 設定

JSSEについて詳しく説明する前に、これから使用するクライアント / サーバー・アプリケーションについて知っておきましょう。SimpleSSLServerとSimpleSSLClientは、デモ用のアプリケーションの2つのコンポーネントです。例を実行するには、アプリケーションのそれぞれの終わりでいくつかのKeyStoreおよびTrustStoreファイルを設定する必要があります。具体的には、以下のファイルが必要です。

- clientKeysと呼ばれるクライアント用KeyStoreファイル。架空のユーザーAliceとBobの証明書が含まれます。
- serverKeysと呼ばれるサーバー用KeyStoreファイル。サーバー用の証明書が含まれます。
- clientTrustと呼ばれるクライアント用TrustStoreファイル。サーバーの証明書が含まれます。
- serverTrustと呼ばれるサーバー用TrustStoreファイル。AliceとBobの証明書が含まれます。

お好きなキー管理ユーティリティーを使用して、これらをひとまとめにしてください。`alice`と`bob`、`server`をそれぞれのキーのエイリアスとし、すべてのストアと証明書のパスワードとして`password`を使用します。この方法をご存じない場合は、サイドバーの「Setting up KeyStore and TrustStore files」をご覧ください。

次に、この記事に添付されているjarファイルをダウンロードしてください。これらのファイルには、クライアント / サーバー・アプリケーションのソースコードとコンパイルしたものが含まれているので、ファイルをCLASSPATHの指定に加えるだけで使用できるようになります。

## セキュア接続の確立

SimpleSSLServerを実行するには、以下の (少し長い) コマンドを入力します。

```
java -Djavax.net.ssl.keyStore=serverKeys
 -Djavax.net.ssl.keyStorePassword=password
 -Djavax.net.ssl.trustStore=serverTrust
 -Djavax.net.ssl.trustStorePassword=password SimpleSSLServer
```

サーバーを特定するのに使用されるKeyStoreを指定し、KeyStoreに設定されているパスワードも指定しました。サーバーにはクライアント認証が必要なので、TrustStoreも指定しました。TrustStoreを指定することによって、SSLSimpleServerが、SSLSimpleClientによって提示された証明書を信用するようにします。サーバーの初期化後、以下の報告が表示されます。

```
SimpleSSLServer running on port 49152
```

その後、サーバーは、クライアントからの接続を待ちます。異なるポートでサーバーを実行したい場合は、コマンドの終わりに`-port xxx`と指定し、変数`xxx`を選択した番号で置き換えます。

次に、アプリケーションのクライアント・コンポーネントを設定します。別のコンソール・ウィンドウから以下のコマンドを入力します。

```
java -Djavax.net.ssl.keyStore=clientKeys
 -Djavax.net.ssl.keyStorePassword=password
 -Djavax.net.ssl.trustStore=clientTrust
 -Djavax.net.ssl.trustStorePassword=password SimpleSSLClient
```

クライアントは、デフォルトで、ローカルホストのポート49152で稼働しているサーバーへの接続を試みます。コマンド・ラインで`-host`と`-port`引数を使用してホストを変更することができます。クライアントがサーバーに接続されると、以下のメッセージが表示されます。

```
Connected
```

一方で、サーバーは接続要求を報告し、以下のように、クライアントによって提示された識別名を表示します。

```
1: New connection request
1: Request from CN=Bob, OU=developerWorks, O=IBM, L=Winchester, ST=Hampshire, C=UK
```

新しい接続をテストするには、何らかのテキストをクライアントに入力し、Returnを押して、サーバーがそのテキストをエコーするのを確認します。クライアントを切断するには、クライアント・コンソールでCtrl-Cを押します。サーバーは、以下のように切断を記録します。

```
1: Client disconnected
```

サーバーを終了する必要はありません。サーバーを稼働させたまま、さまざまな作業をすることにしましょう。

## SimpleSSLServerの内部

この記事の残りの部分では、主にクライアント・アプリケーションについて取り上げる予定ですが、サーバー・コードについても一通り説明しておいたほうがいいでしょう。サーバー・アプリケーションがどのように機能するかを学びながら、`HandshakeCompletedListener`インターフェースを使用して、SSL接続に関する情報を取り出す方法を学びましょう。

`SimpleSSLServer.java`は、以下のように、3つのインポート・ステートメントから始まります。

```
import javax.net.ssl.*;
import java.security.cert.*;
import java.io.*;
```

- `javax.net.ssl`は、3つのなかで最も重要なステートメントです。このステートメントには、コアのJSSEクラスのほとんどが含まれており、SSLによるどの作業にもこれが必要です。
- `java.security.cert`は、個々の証明書を扱う必要がある場合に役立ちます。これについては、後で説明します。
- `java.io`は、標準のJava I/Oパッケージです。ここでは、これを使用して、セキュア・ソケットを介して送受信されるデータを処理します。

次に、`main()`メソッドが、コマンド・ラインでオプションの`-port`引数を確認します。そして、以下に示されているように、デフォルトの`SSLServerSocketFactory`を取得し、`SimpleSSLServer`オブジェクトを作成し、ファクトリーをコンストラクターに渡し、サーバーを起動します。

```
SSLServerSocketFactory ssf=
  (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
SimpleSSLServer server=new SimpleSSLServer(ssf, port);
server.start();
```

サーバーは別スレッドで稼働するので、それが起動すると`main()` は終了します。新しいスレッドは`run()` メソッドを呼び出し、その中では以下に示されているように、`SSLServerSocket` を作成し、クライアント認証を要求するようにサーバーを設定します。

```
SSLServerSocket serverSocket=
  (SSLServerSocket)serverSocketFactory.createServerSocket(port);
serverSocket.setNeedClientAuth(true);
```

## HandshakeCompletedListener

いったん起動すると、`run()` メソッドは無限ループに入り、クライアントからの要求を待ちます。各ソケットは、クライアントの証明書の識別名 (DN) を表示するのに使用される`HandshakeCompletedListener` の実装１つ１つに対応付けられています。`InputDisplayer` は、ソケットの`InputStream` をラップして別のスレッドとして実行され、ソケットから来たデータを`System.out` そのまま出力します。SimpleSSLServerのメイン・ループをリスト1に示します。

## リスト1. SimpleSSLServerメイン・ループ

```
while (true) {
  String ident=String.valueOf(id++);
  // Wait for a connection request.
  SSLSocket socket=(SSLSocket)serverSocket.accept();
  // We add in a HandshakeCompletedListener, which allows us to
  // peek at the certificate provided by the client.
  HandshakeCompletedListener hcl=new SimpleHandshakeListener(ident);
  socket.addHandshakeCompletedListener(hcl);
  InputStream in=socket.getInputStream();
  new InputDisplayer(ident, in);
}
```

私たちの`HandshakeCompletedListener` である`SimpleHandshakeListener` は、`handshakeCompleted()` メソッドの実装を提供します。SSLハンドシェーク・フェーズが完了すると、JSSEインフラストラクチャーによってこのメソッドが呼び出され、接続に関する情報を (`HandshakeCompletedEvent` オブジェクトで) 渡します。リスト2に示されているように、これを使用して、クライアントのDNを取得し表示します。

## リスト2. SimpleHandshakeListener

```
class SimpleHandshakeListener implements HandshakeCompletedListener
{
  String ident;
  /**
   * Constructs a SimpleHandshakeListener with the given
   * identifier.
   * @param ident Used to identify output from this Listener.
   */
  public SimpleHandshakeListener(String ident)
  {
    this.ident=ident;
  }
  /** Invoked upon SSL handshake completion. */
  public void handshakeCompleted(HandshakeCompletedEvent event)
```

```
  {
    // Display the peer specified in the certificate.
    try {
X509Certificate cert=(X509Certificate)event.getPeerCertificates()[0];
      String peer=cert.getSubjectDN().getName();
      System.out.println(ident+": Request from "+peer);
    }
    catch (SSLPeerUnverifiedException pue) {
      System.out.println(ident+": Peer unverified");
    }
  }
}
```

### J2SE 1.2または1.3でのサーバー・アプリケーションの実行

J2SE 1.2または1.3でSimpleSSLServerアプリケーションを実行する場合は、わずかに異なる (現在では古い) JSSE APIを使用する必要があります。`java.security.cert` をインポートするのではなく、`javax.security.cert` を使用してください。このパッケージにも、`X509Certificate` と呼ばれるクラスが含まれています。ただし、`HandshakeCompletedEvent` から証明書オブジェクトの配列を取得するには、`getPeerCertificates()` メソッドではなく`getPeerCertificateChain()` メソッドを使用しなければなりません。

赤色でハイライト表示されているのは、重要な2行です。`getPeerCertificates` は、`X509Certificate` オブジェクトの配列として証明書のチェーンを返します。これらの証明書オブジェクトは、ピア (つまり、クライアント) の識別を確立します。この配列の最初は、クライアントそのものの証明書です。最後は、通常、認証局証明書です。ピアの証明書を取得すると、DNを取得し、それを`System.out` に表示します。`X509Certificate` は、`java.security.cert` パッケージで定義されています。

## SimpleSSLClientの内部

最初に説明するクライアント・アプリケーションでは難しいことは何もしていません。ただし、後の例でそれを拡張し、さらに高度な機能について説明していきます。SimpleSSLClientは、サブクラスを簡単に追加できるように作られています。以下の4つのメソッドが上書きされます。

- クラスがコマンド・ラインから実行されると、当然、`main()` が呼び出されます。各サブクラスににについて、`main()` は、適切なクラスのオブジェクトを作成し、そのオブジェクトで`runClient()` メソッドと`close()` メソッドを呼び出すように上書きされなければなりません。これらのメソッドは、`SimpleSSLClient` スーパークラスで提供され、上書きされません。
- `handleCommandLineOption()` と`displayUsage()` によって、各サブクラスは、親クラスを更新せずにコマンド・ラインでオプションを追加できます。これらは両方とも`runClient()` メソッドから呼び出されます。
- `getSSLSocketFactory()` は、興味深いメソッドです。JSSEセキュア・ソケットは、常に、`SSLSocketFactory` オブジェクトから作成されます。カスタマイズされたソケット・ファクトリーを作成することによって、JSSEの振る舞いをカスタマイズできます。今後の実行のために、各SimpleSSLClientサブクラスは、このメソッドを実装し、`SSLSocketFactory` を適切にカスタマイズします。

当面は、SimpleSSLClientは、ユーザーがサーバーでクライアントを提示できる`-host` と`-port` パラメーターのみを理解します。以下に示されているように、この最初の基本的な例

で、getSSLSocketFactory は (JVM全体にわたって使われる) デフォルトのファクトリーを返します。

```
protected SSLSocketFactory getSSLSocketFactory()
  throws IOException, GeneralSecurityException
{
  return (SSLSocketFactory)SSLSocketFactory.getDefault();
}
```

runClient() メソッドが、そのサブクラスのmain() メソッドから呼び出され、コマンド・ライン引数を処理し、サブクラスでオーバーライドしたメソッド経由でSSLSocketFactory を取得して使用します。そして、connect() メソッドを使用してサーバーに接続し、transmit() メソッドを使用してセキュア・チャネルを介してデータの送信を開始します。

connect() メソッドは非常に簡単です。SSLSocketFactory を使用してサーバーに接続した後に、セキュア・ソケットでstartHandshake を呼び出します。これによって、JSSEは、SSLハンドシェーク・フェーズを完了するよう強制され、サーバー側のHandshakeCompletedListener が起動されます。JSSEは、自動的にハンドシェークを開始しますが、データがソケットを介して最初に送信されるときにのみ実行します。私たちは、ユーザーがキーボードでメッセージを入力するまでデータは送らないものの、サーバーにはすぐに接続を報告させたいので、startHandshake を使用してハンドシェークを強制する必要があります。

transmit() メソッドも非常に簡単です。その最初のタスクは、以下に示されているように、適切なReader で入力ソースをラップすることです。

```
BufferedReader reader=new BufferedReader(
  new InputStreamReader(in));
```

BufferedReader は、入力を個々の行に分割するのでこれを使用します。

次に、transmit() メソッドが、適切なWriter で出力ストリームをラップします (この場合、セキュア・ソケットによって提供されたOutputStream)。サーバーは、テキストがUTF-8でエンコードされていることを想定しているので、このエンコードを使用するようOutputStreamWriter に指示します。

```
 writer=new OutputStreamWriter(socket.getOutputStream(), "UTF-8");
```

メイン・ループは簡単です。リスト3でおわかりになるように、それは、SimpleSSLServer のInputDisplayer のメイン・ループに非常に類似しています。

## リスト3. SimpleSSLClientのメイン・ループ

```
boolean done=false;
while (!done) {
  String line=reader.readLine();
  if (line!=null) {
    writer.write(line);
    writer.write\n');
    writer.flush();
  }
  else done=true;
}
```

基本的なJSSEサーバーとクライアントのコードについてはこれで終わります。では次に、SimpleSSLClientを拡張して、`getSSLSocketFactory` の他の実装を見てみましょう。

## 自前のKeyStoreの使用

SimpleSSLClientをどのように実行したか覚えていますか。コマンドは以下のようなものでした。

```
java -Djavax.net.ssl.keyStore=clientKeys
   -Djavax.net.ssl.keyStorePassword=password -Djavax.net.ssl.trustStore=clientTrust
   -Djavax.net.ssl.trustStorePassword=password SimpleSSLClient
```

なんと長いコマンドでしょうか。幸いにも、この例と次の例は、KeyStoreとTrustStoreへのハードコーディングされたパスによって`SSLSocketFactory` を設定する方法を示しています。これから学ぶ技術によって、上述のコマンドの長さが短縮されるだけでなく、それぞれ異なるKeyStoreとTrustStoreの設定を持つ、複数の`SSLSocketFactory` オブジェクトを設定できます。こうした設定を行わないと、JVM内の各セキュア接続は、同じKeyStoreとTrustStoreを使用しなければなりません。これは小規模のアプリケーションには受け入れられるかもしれませんが、より大規模のアプリケーションでは、数多くのさまざまなユーザーが複数のピアに接続する必要があります。

### CustomKeyStoreClientの導入

最初の例では、アプリケーション例 CustomKeyStoreClient (記事のソースにあります) を使用して、KeyStoreを動的に定義します。ソース・コードを見る前に、CustomKeyStoreClientの動作を見てみましょう。この実行では、TrustStoreは指定しますがKeyStore は指定しません。CustomKeyStoreClientのコマンド・ラインで以下の引数を入力してどうなるか見てみましょう。

```
java -Djavax.net.ssl.trustStore=clientTrust
   -Djavax.net.ssl.trustStorePassword=password CustomKeyStoreClient
```

クライアントが正常に接続していれば、サーバーは、有効な証明書が提示されたことを報告します。 `CustomKeyStoreClient.java` が、KeyStoreの名前 (`clientKeys`) とパスワード (`password`) でハードコーディングされているので、接続は成功します。クライアントKeyStoreに異なるファイル名またはパスワードを選択した場合は、新しいコマンド・ライン・オプションの`-ks` と`-kspass` を使用して、それらを指定できます。

`CustomKeystoreClient.java` 用のソース・コードを見ると、`getSSLSocketFactory` が最初に行うのは、`getKeyManagers()` ヘルパー・メソッドの呼び出しです。これがどのように機能するかは後で説明します。当面は、それが、必要なKeyStoreファイルとパスワードによって設定された`KeyManager` オブジェクトの配列を返すことだけみておいてください。

## リスト4. CustomKeyStoreClient.getSSLSocketFactory

```
protected SSLSocketFactory getSSLSocketFactory()
  throws IOException, GeneralSecurityException
{
  // Call getKeyManagers to get suitable key managers
  KeyManager[] kms=getKeyManagers();
  // Now construct a SSLContext using these KeyManagers. We
  // specify a null TrustManager and SecureRandom, indicating that the
  // defaults should be used.
  SSLContext context=SSLContext.getInstance("SSL");
  context.init(kms, null, null);
  // Finally, we get a SocketFactory, and pass it to SimpleSSLClient.
  SSLSocketFactory ssf=context.getSocketFactory();
  return ssf;
}
```

KeyManager 配列の取得後、getSSLSocketFactory は、JSSEのすべてのカスタマイズに共通する基本的な設定作業を実行します。SSLSocketFactory を作成するために、アプリケーションは、SSLContext のインスタンスを取得し、それを初期化し、SSLContext を使用してSSLSocketFactory を生成します。

SSLContext を取得するときは、"SSL" のプロトコルを指定します。また、SSL (またはTLS) プロトコルの特定のバージョンを置いて、通信がその特定のレベルで行われるよう強制することもできます。"SSL" を指定することによって、JSSEは、サポートできる最高のレベルをデフォルトにすることができます。

SSLContext.init の最初の引数は、使用するKeyManager 配列です。2番目の引数は、ここではヌル値のままですが、TrustManager オブジェクトの同様の配列で、後で説明します。2番目の引数をヌル値にしておくことによって、JSSEにデフォルトのTrustStoreを使用するよう伝えます。これは、javax.net.ssl.trustStore とjavax.net.ssl.trustStorePassword システム・プロパティーから設定を取り出します。3番目のパラメーターによって、JSSEの乱数発生ルーチン (RNG) を上書きできます。RNGはSSL注意の必要な部分であり、このパラメーターを誤用すると、接続がセキュアでなくなる可能性があります。このパラメーターをヌル値のままにすることによって、JSSEはデフォルトでセキュアなRNGであるSecureRandom オブジェクトを使用できます。

## KeyStoreのロード

次に、getKeyManagers がどのようにKeyManagers の配列をロードし初期化するかについて考えましょう。リスト5のコードから始めて、議論を深めていきます。

## リスト5. KeyManagersのロードと初期化

```
protected KeyManager[] getKeyManagers()
  throws IOException, GeneralSecurityException
{
  // First, get the default KeyManagerFactory.
  String alg=KeyManagerFactory.getDefaultAlgorithm();
  KeyManagerFactory kmFact=KeyManagerFactory.getInstance(alg);
    // Next, set up the KeyStore to use. We need to load the file into
  // a KeyStore instance.
  FileInputStream fis=new FileInputStream(keyStore);
  KeyStore ks=KeyStore.getInstance("jks");
  ks.load(fis, keyStorePassword.toCharArray());
  fis.close();
  // Now we initialize the TrustManagerFactory with this KeyStore
  kmFact.init(ks, keyStorePassword.toCharArray());
  // And now get the TrustManagers
  KeyManager[] kms=kmFact.getKeyManagers();
  return kms;
}
```

最初の作業は、KeyManagerFactory の取得ですが、これを行うには、どのアルゴリズムを使用するかを知っている必要があります。幸いにも、JSSEでは、デフォルトのKeyManagerFactory アルゴリズムが利用できます。このデフォルトは、ssl.KeyManagerFactory.algorithm セキュリティー・プロパティーを使用して設定できます。

getKeyManagers() メソッドでは次に、KeyStoreファイルをロードします。これには、このファイルのInputStream の設定、KeyStore のインスタンスの取得、そしてInputStream からのKeyStore のロードが伴います。InputStream と同様に、KeyStore は、ストリームのフォーマット (ここではデフォルトの"jks" を指定) とストア・パスワードを知っておく必要があります。ストア・パスワードは、文字の配列として提供されなければなりません。

### CustomKeyStoreClientパッケージのインポート

KeyStore クラスにアクセスするには、javax.net.ssl とjava.security.cert の両方をインポートしなければなりません。SSLContext やKeyManagerFactory などの他のクラスは、それ以降のJ2SE 1.4のjavax.net.ssl のメンバーです。J2SE 1.2または1.3では、これらのクラスは標準の場所にはありません。たとえば、Sun JSSE実装ではcom.sun.net.ssl にあります。

役に立ちそうなテクニックとして注目したいのは、KeyStore.load に任意のInputStream を使用できることです。アプリケーションは、どこからでもこれを作成することができます。つまり、ファイルだけでなく、ネットワークを介して、またはモバイル装置から作成したり、あるいは、稼働中にストリームを生成することができます。

KeyStore のロード後、それを使用して、以前に作成したKeyManagerFactory を初期化します。再び、パスワード、ただし今回は個々の証明書のパスワード、を指定する必要があります。通常JSSEでは、KeyStoreの各証明書は、KeyStoreそのものと同じパスワードを持つ必要があります。KeyManagerFactory を自分で作成することによって、この制限を解除できます。

KeyManagerFactory の初期化後、getKeyManagers() メソッドを使用して、適切なKeyManager オブジェクトの配列を取得するのは容易な操作なので説明はいらないでしょう。

CustomKeyStoreClientについて、どのロケーションからでも (ここではローカル・ファイルシステムを使用) KeyStoreをロードする方法と、異なるパスワードを証明書とKeyStoreそのものに使用できるようにする方法がわかりました。KeyStoreの各証明書が異なるパスワードを持つことができるようにする方法については、あとで説明します。この例では、クライアント側について取り上げましたが、サーバー側でも同じ手法を使用して、適切な`SSLServerSocketFactory`オブジェクトを作成できるでしょう。

## 自前のTrustStoreの使用

### CustomTrustStoreClientパッケージのインポート

ここでもう一度、この例で使用されるクラスは、さまざまなJSSEプロバイダー用のさまざまなパッケージにあります。J2SE 1.4では、`TrustManagerFactory`は`javax.net.ssl`にあり、J2SE 1.2または1.3では、通常`com.sun.net.ssl`にあります。

予想されたことですが、JSSEのデフォルトのTrustStoreを置き換える方法は、KeyStoreでの方法と非常に類似しています。CustomTrustStoreClient (記事のソースにあります) は、ハードコーディングされたKeyStoreとハードコーディングされたTrustStoreの両方を使用するように設定されています。これを実行するために必要なのは、`java CustomTrustStoreClient`コマンドのみです。

CustomTrustStoreClientは、KeyStoreが、`password`というパスワードを持つ`clientKeys`と呼ばれるファイルであり、TrustStoreが、`password`というパスワードを持つ`clientTrust`と呼ばれるファイルであることを想定しています。CustomKeyStoreClientと同様に、これらのデフォルトは、`-ks`、`-kspass`、`-ts`、`-tspass`引数を使用して置き換えることができます。

`getSSLSocketFactory()`の大部分は、CustomKeyStoreClientの同じメソッドと同一です。前述の例のように、CustomKeyStoreClientから`getKeyManagers()`メソッドを呼び出して、カスタマイズされた`KeyManager`オブジェクトの同じ配列を取得します。ただし今回は、`getSSLSocketFactory`は、カスタマイズされた`TrustManager`オブジェクトの配列も取得しなければなりません。リスト6では、`getSSLSocketFactory`が`getTrustManagers()`ヘルパー・メソッドを使用して、カスタマイズされた`TrustManager`オブジェクトを取得する方法がわかります。

### リスト6. getSSLSocketFactoryがTrustManagersを使用する方法

```
protected SSLSocketFactory getSSLSocketFactory()
  throws IOException, GeneralSecurityException
{
  // Call getTrustManagers to get suitable trust managers
  TrustManager[] tms=getTrustManagers();
    // Call getKeyManagers (from CustomKeyStoreClient) to get suitable
  // key managers
  KeyManager[] kms=getKeyManagers();
  // Next construct and initialize a SSLContext with the KeyStore and
  // the TrustStore. We use the default SecureRandom.
  SSLContext context=SSLContext.getInstance("SSL");
  context.init(kms, tms, null);
  // Finally, we get a SocketFactory, and pass it to SimpleSSLClient.
  SSLSocketFactory ssf=context.getSocketFactory();
  return ssf;
}
```

今回は、コンテキストを初期化するときにKeyStoreとTrustStoreの両方を置き換えます。ただし、3番目のパラメーターとして`null` を渡すことで、JSSEは、依然としてデフォルトの`SecureRandom` を使用します。

予想通り、リスト7に示されているように、`getTrustManagers` は、CustomKeyStoreClientの`getKeyManagers` によく似ています。

### リスト7. TrustManagersのロードと初期化

```
protected TrustManager[] getTrustManagers()
throws IOException, GeneralSecurityException
{
  // First, get the default TrustManagerFactory.
  String alg=TrustManagerFactory.getDefaultAlgorithm();
  TrustManagerFactory tmFact=TrustManagerFactory.getInstance(alg);
    // Next, set up the TrustStore to use. We need to load the file into
  // a KeyStore instance.
  FileInputStream fis=new FileInputStream(trustStore);
  KeyStore ks=KeyStore.getInstance("jks");
  ks.load(fis, trustStorePassword.toCharArray());
  fis.close();
  // Now we initialize the TrustManagerFactory with this KeyStore
  tmFact.init(ks);
  // And now get the TrustManagers
  TrustManager[] tms=tmFact.getTrustManagers();
  return tms;
}
```

`getTrustManagers()` メソッドは、前述と同様に、デフォルトのアルゴリズムに基づいた`TrustManagerFactory` をインスタンス化することによって起動します。そして、TrustStoreファイルを`KeyStore` オブジェクト（お気づきのとおり、名前付けは悲劇的なことになっています）にロードし、`TrustManagerFactory` を初期化します。

KeyStoreに相当するものと異なり、`TrustManagerFactory` を初期化するときはパスワードを提供する必要がないことに注意してください。秘密鍵と異なり、承認された証明書は、個々のパスワードによって保全される必要がありません。

これまでに、KeyStoreとTrustStoreを動的に上書きする方法について検討してきました。これらの2つの例を完了したことによって、`KeyManagerFactory` と`TrustManagerFactory` を設定し、これらを使用して`SSLContext` を設定する方法についておわかりいただけたと思います。最後の例は、もう少し複雑ですが、`KeyManager` の独自の実装を作成してみましょう。

## カスタムKeyManagerの設定: エイリアスの選択

クライアント・アプリケーションの以前のバージョンを実行したときに、どの証明書DNがサーバーによって表示されたかお気付きになりましたか。私たちは故意に、2つの受け入れ可能な証明書 (1つはAlice用、1つはBob用) が含まれるようにクライアントKeyStoreを設定しました。この場合、JSSEは、どちらでも適合すると思われる証明書を選択します。私のインストールでは、いつもBobの証明書が選択されているようですが、皆さんのJSSEでは違うかもしれません。

例のSelectAliasClientアプリケーションでは、提示する証明書を選択できます。Keystoreの各証明書を`alice` または`bob` のエイリアスで名前付けしたので、Aliceの証明書を選択するには、`java SelectAliasClient -alias alice` のコマンドを入力するだけです。

クライアントが接続し、SSLハンドシェークが完了すると、サーバーは以下のように応答します。

```
1: New connection request
1: Request from CN=Alice, OU=developerWorks, O=IBM, L=Winchester, ST=Hampshire, C=UK
```

(あるいは、Aliceの証明書を作成したときに選択した値)。同様に、Bobの証明書を選択する場合は、`java SelectAliasClient -alias bob` を入力すると、サーバーが以下を報告します。

```
2: New connection request
2: Request from CN=Bob, OU=developerWorks, O=IBM, L=Winchester, ST=Hampshire, C=UK
```

## カスタムKeyManager実装

特定のエイリアスの選択を強制するには、通常JSSEによって使用されるSSL通信用の`KeyManager` である、`X509KeyManager` の実装を作成します。私たちの実装には、本物の`X509KeyManager` が含まれ、単に呼び出しのほとんどを渡します。インターセプトする唯一のメソッドは、`chooseClientAlias()` です。実装は、必要なエイリアスが有効であるか否かを確認し、有効な場合はそれを返します。

`X509KeyManager` インターフェースは、SSLハンドシェーク・フェーズの間に数多くのメソッドを使用して、鍵を検索します。この鍵はピアの特定に使用されます。参考文献 のセクションにすべてのメソッドへの参照があります。この実行には、以下のメソッドが重要です。

- `getClientAliases()` と`getServerAliases()` は、それぞれ`SSLSocket` と`SSLServerSocket` で使用するために有効なエイリアスの配列を提供します。
- `chooseClientAlias()` と`chooseServerAlias()` は、単一の有効なエイリアスを返します。
- `getCertificateChain()` と`getPrivateKey()` は、それぞれ、パラメーターとしてエイリアスをとり、識別された証明書に関する情報を戻します。

### カスタムKeyManagerの制御フロー

制御フローは、以下のように機能します。

1. JSSEは、`chooseClientAlias` を呼び出し、使用するエイリアスを検索します。
2. `chooseClientAlias` は、本物の`X509KeyManager` 上で`getClientAliases` を呼び出し、必要なエイリアスが有効であるかどうかを確認できるよう、有効なエイリアスのリストを検索します。
3. JSSEは、正しいエイリアスを指定して、我々の`X509KeyManager` に対して`getCertificateChain` と`getPrivateKey` を呼び出し、それによって、呼び出しは、ラップされたKeyManagerに渡されます。

私たちのKeyManagerである`AliasForcingKeyManager` の`chooseClientAlias()` メソッドは、実は、リスト8で示されているように、JSSEインストールによってサポートされるキー型ごとに、`getClientAliases()` を複数回呼び出す必要があります。

## リスト8. エイリアスの選択の強制

```
public String chooseClientAlias(String[] keyType, Principal[] issuers,
                                Socket socket)
{
  // For each keyType, call getClientAliases on the base KeyManager
  // to find valid aliases. If our requested alias is found, select it
  // for return.
  boolean aliasFound=false;
  for (int i=0; i<keyType.length && !aliasFound; i++) {
    String[] validAliases=baseKM.getClientAliases(keyType[i], issuers);
    if (validAliases!=null) {
      for (int j=0; j<validAliases.length && !aliasFound; j++) {
        if (validAliases[j].equals(alias)) aliasFound=true;
      }
    }
  }
  if (aliasFound) return alias;
  else return null;
}
```

AliasForcingKeyManager には、X509KeyManager のその他の5つのメソッドの実装が必要です。これらは単にbaseKM に対して相当するものを呼び出します。

ここではまだ、通常のKeyManager ではなくAliasForcingKeyManager を使用しています。これはgetSSLSocketFactory で起こります。これは、リスト9で示されているように、まず他の例で示したgetKeyManagers とgetTrustManagers を呼び出しますが、次に、AliasForcingKeyManager のインスタンスでgetKeyManagers から返された各KeyManager をラップします。

## リスト9. X509KeyManagersのラップ

```
protected SSLSocketFactory getSSLSocketFactory() throws IOException, GeneralSecurityException
{
  // Call the superclasses to get suitable trust and key managers
  KeyManager[] kms=getKeyManagers();
  TrustManager[] tms=getTrustManagers();
  // If the alias has been specified, wrap recognized KeyManagers
  // in AliasForcingKeyManager instances.
  if (alias!=null) {
    for (int i=0; i<kms.length; i++) {
      // We can only deal with instances of X509KeyManager
      if (kms[i] instanceof X509KeyManager)
        kms[i]=new AliasForcingKeyManager((X509KeyManager)kms[i], alias);
    }
  }
  // Now construct a SSLContext using these (possibly wrapped)
  // KeyManagers, and the TrustManagers. We still use a null
  // SecureRandom, indicating that the defaults should be used.
  SSLContext context=SSLContext.getInstance("SSL");
  context.init(kms, tms, null);
  // Finally, we get a SocketFactory, and pass it to SimpleSSLClient.
  SSLSocketFactory ssf=context.getSocketFactory();
  return ssf;
}
```

### KeyManagerのリパッケージ

KeyManager とX509KeyManager は両方とも、J2SE 1.2と1.3ではベンダー固有のパッケージにありましたが、J2SE 1.4ではjavax.net.ssl に移動しました。X509KeyManager 用のメソッド・シグニチャーは、インターフェースが移行したときにわずかに変更されました。

これまでに説明した手法を使用すれば、`KeyManager` のどのような側面でも置き換えることができます。同様に、それらを使用して`TrustManager` を置き換え、リモート・ピアからきた証明書を承認するかどうかを決定するためのJSSEのメカニズムを変更できます。

## まとめ

この記事では、かなり多くのテクニックや技術について説明しました。最後に、簡単に復習しておきましょう。皆さんは、以下の方法について、基本的なレベルの理解が得られたと思います。

- `HandshakeCompletedListener` を使用して、接続に関する情報を収集する方法
- `SSLContext` から`SSLSocketFactory` を取得する方法
- カスタムで動的なTrustStoreまたはKeyStoreを使用する方法
- KeyStoreパスワードと個々の証明書パスワードを一致させなければならないJSSEの制限を緩和する方法
- 独自のKeyManagerを使用して、証明書を識別する選択を指定する方法

また、さまざまなアプリケーション・シナリオのために、適宜、これらの技術を拡張する方法も提示しました。独自の実装で`X509KeyManager` をラップするテクニックは、JSSEの他の多くのクラスに対しても使用することができ、単にハードコーディングされたファイルをロードするよりも、`TrustStore` と`KeyStore` を使用するほうが、より面白いのは確かです。

この記事で説明した高度なJSSEカスタマイズの実装をどのように選択するかに関係なく、これらはどれも不用意に使用してはなりません。SSLの内部をいじくるときは、1つの間違いで、接続がセキュアでなくなる可能性を常に忘れないことが大切です。

---

# 著者について

Ian Parkinson

Ian Parkinsonは、オックスフォード大学を卒業後、1998年にIBMに入社しました。メインフレームz/OSシステムのプログラミングに携わった後、現在は、IBMのWebSphere MQメッセージング製品用のJavaインターフェースに携わっており、イギリスのHursley Laboratoryで働いています。氏は、Javaは遅すぎるという意見に賛成していないことで知られています。氏の連絡先は、ianp@hursley.ibm.com です。