第 6 章

验证数据完整性 —— 消息摘要算法

相信读者对于MySQL已经非常了解了，作为一款开源的数据库，它已经成为无数开源爱好者的数据库首选。

读者朋友一定对图6-1并不陌生，它是MySQL官方提供的数据库发行版（版本5.1.38）的下载页面（详见http://dev.mysql.com/downloads/mysql/5.1.html#win32）。

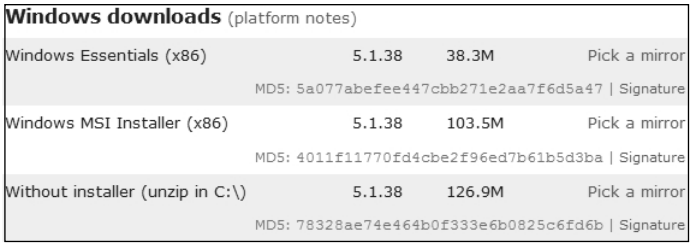


图6-1　MySQL下载页面

在这个下载页面中，有一个不规律的字符串（MD5: 5a077abefee447cbb271e2aa7f6d5a47）引起了作者的注意。这个不规律的字符串长度为32个字符，由英文字母和数字组成，很显然这是一个十六进制编码字符串。它还有一个时髦的名字，叫“数字指纹”。它就是本章的主角，消息摘要算法——MD5。

我们看到Windows Essentials (x86)、Windows MSI Installer (x86)和Without installer (unzip in C:\)三种发行版的大小是有所不同的，三种发行版对应的大小分别为38.3MB、103.5MB和126.8MB。但是，其数字指纹的长度却都是一样的，都是32个字符的十六进制串。

6.1　消息摘要算法简述

消息摘要算法包含MD、SHA和MAC共3大系列，常用于验证数据的完整性，是数字签名算法的核心算法。

6.1.1　消息摘要算法的由来

相信读者朋友都有从网上下载软件的经历，偶尔也有从网上下载到破损文件的经历。情况

严重时，还可能从某软件的官网上下载到被篡改的软件。如何来验证下载到的文件和官方提供的文件是否一致？这就引入了数据完整性验证的问题。

该如何验证其一致性？肉眼比较？大小比较？均无可取之处！我们需要一种方便快捷、安全有效的算法。

先不说如何比较文件是否相同的问题，我们说说如何比较两个对象是否相同。

相信广大读者朋友都有使用equals()方法来比较对象的经历。但很多读者朋友不知道，实际上equals()方法比较的是两个对象的散列值，即比较两个对象hashCode ()方法的值是否相同，这说明hashCode可以作为辨别对象的唯一标识。

什么是hashCode呢？顾名思义，hashCode就是散列值。

我们在第2章中曾经介绍过散列函数，它恰恰能够用于数据完整性的校验。任何消息经过散列函数处理后，都会获得唯一的散列值。这一过程称为“消息摘要”，其散列值称为“数字指纹”，自然其算法就是“消息摘要算法”了。换句话说，如果其数字指纹唯一，就说明其消息是一致的。

由此，消息摘要算法成了校验数据完整性的主要手段。各大软件厂商提供软件下载的同时总要附带上数字签名，这一做法也就不足为奇了。为了能够更加方便、有效地验证数据的完整性，有的软件厂商还提供了不同的消息摘要算法的数字指纹，如MD5和SHA算法，甚至是HMAC算法的数字指纹。此外，用于校验数据完整性的算法还有CRC32算法等。为了方便人们识别和阅读，数字指纹常以十六进制字符串的形式出现。

消息摘要算法最初是用来构建数字签名的。数字签名操作中，签名操作其实是变相地使用消息摘要算法获得的数字指纹，而验证操作则是验证其数字指纹是否相符。这也是为什么当山东大学王小云教授使用碰撞算法破解了MD5和SHA算法后，使得数字签名在理论上被伪造成为可能。

消息摘要算法一直是非对称加密算法中一项举足轻重的关键性算法。

6.1.2 消息摘要算法的家谱

消息摘要算法又称为散列算法，其核心在于散列函数的单向性。即通过散列函数可获得对应的散列值，但不可通过该散列值反推其原始信息。这是消息摘要算法的安全性的根本所在。

消息摘要算法主要分为三大类：MD（Message Digest，消息摘要算法）、SHA（Secure Hash Algorithm，安全散列算法）和MAC（Message Authentication Code，消息认证码算法）。

如前文所述，MD5、SHA和HMAC都属于消息摘要算法，它们是三大消息摘要算法的主要代表。MD系列算法包括MD2、MD4和MD5共3种算法；SHA算法主要包括其代表算法SHA-1和SHA-1算法的变种SHA-2系列算法（包含SHA-224、SHA-256、SHA-384和SHA-512）；MAC算法综合了上述两种算法，主要包括HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和HmacSHA512算法。

我们知道，科学技术的发展是不以人的意志为转移的。尽管上述内容列举了各种消息摘要算法，但仍不能满足应用需要。基于这些消息摘要算法，又衍生出了RipeMD系列（包含

RipeMD128、RipeMD160、RipeMD256、RipeMD320）、Tiger、GOST3411和Whirlpool算法。

6.2 MD算法家族

每当人们一提到消息摘要算法，就很自然地会联想到MD5和SHA算法。MD5算法是典型的消息摘要算法，是计算机广泛使用的杂凑算法之一（又译摘要算法、散列算法），更是消息摘要算法的首要代表。

6.2.1 简述

MD5算法是典型的消息摘要算法，其前身有MD2、MD3和MD4算法，它由MD4、MD3、MD2算法改进而来。不论是哪一种MD算法，它们都需要获得一个随机长度的信息并产生一个128位的信息摘要。如果将这个128位的二进制摘要信息换算成十六进制，可以得到一个32位（每4位二进制数转换为1位十六进制数）的字符串，故我们见到的大部分MD5算法的数字指纹都是32位十六进制的字符串，如本文开篇中，MySQL下载页上的数字指纹（MD5: 5a077abefee447cbb271e2aa7f6d5a47）就是32位的十六进制串。现在，各大主流计算机语言均支持MD5算法。

虽然，MD5算法漏洞越来越多，已不再安全，但至今我们仍没有看到它的下一版本 —— MD6算法的出现。或许，同样基于MD4算法改进而来的SHA算法将会是MD系列算法的主要替代者。

让我们一同回顾一下MD算法家族的发展历史。

❑ MD2算法

1989年，著名的非对称算法RSA发明人之一 —— 麻省理工学院教授罗纳德·李维斯特（Ronald L. Rivest）开发了MD2算法。这个算法首先对信息进行数据补位，使信息的字节长度是16的倍数。再以一个16位的检验和作为补充信息追加到原信息的末尾。最后根据这个新产生的信息计算出一个128位的散列值，MD2算法由此诞生。

有关MD2算法详情请参见RFC 1319（http://www.ietf.org/rfc/rfc1319.txt）。

❑ MD4算法

1990年，罗纳德·李维斯特教授开发出较之MD2算法有着更高安全性的MD4算法。在这个算法中，我们仍需对信息进行数据补位。不同的是，这种补位使其信息的字节长度加上448个字节后能成为512的倍数（信息字节长度mod 512 = 448）。此外，关于MD4算法的处理与MD2又有很大差别。但最终仍旧是会获得一个128位的散列值。MD4算法对后续消息摘要算法起到了推动作用，许多比较有名的消息摘要算法都是在MD4算法的基础上发展而来的，如MD5、SHA-1、RIPE-MD和HAVAL算法等。

有关MD4算法的详情请参见RFC 1320（http://www.ietf.org/rfc/rfc1320.txt）。

著名开源P2P（Peer-To-Peer，点对点）下载软件EMule（http://www.emule.com）所使用的消息摘要算法正是经过改良后的MD4算法。该算法用于对文件分块后做消息摘要，以验证其文

件的完整性。

❑ MD5算法

1991年，继MD4算法后，罗纳德·李维斯特教授开发了MD5算法，将MD算法推向成熟。MD5算法经MD2、MD3和MD4算法发展而来，算法复杂程度和安全强度大大提高。但不管是MD2、MD4还是MD5算法，其算法的最终结果均是产生一个128位的消息摘要，这也是MD系列算法的特点。MD5算法执行效率略次于MD4算法，但在安全性方面，MD5算法更胜一筹。随着计算机技术的发展和计算水平的不断提高，MD5算法暴露出来的漏洞也越来越多。MD5算法已不再适合安全要求较高的场合使用。

有关MD5算法的详情请参见RFC 1321（http://www.ietf.org/rfc/rfc1321.txt），其中包含了MD2、MD4和MD5三种算法的C语言版实现。

6.2.2 模型分析

我们以一般Web系统中的用户注册/登录模型为例，分析MD5算法在其中的作用。当然，SHA算法也完全适用这个模型，并且有着更高的安全性。

架构师在设计Web应用中的用户注册/登录模块时，都会考虑这样几个问题：

1. 密码如何传递

（1）以Get方式明文传递，不安全。任何人都可以通过链接分析得到。

（2）以Post方式编码传递，也不安全。编码算法是公开的，任何人都能解码。

2. 密码如何存储

（1）明文存储，不安全。拥有数据库查询权限的人就很可能泄露用户密码。

（2）密文存储，安全。如果对密码做摘要处理，任何人都难以破解。

3. 密码如何校验

（1）明文校验，不安全。与明文存储有同样的隐患。

（2）密文校验，安全。与密文存储有相同的优势。

很明显，三个问题答案选项中，后者更具优势。那么这样一个系统的用户注册与登录是怎样一个流程呢？我们通过图6-2和图6-3进行分析。

对于上述流程，大部分读者朋友都有所体会，作者在这里就不复述了。

用户注册/登录加入消息摘要机制后，有以下几个特点：

1）用户注册时，需要将其密码做摘要处理，以摘要信息存入数据库。

2）用户登录时，需要将其密码做摘要处理，通过数据做查询。

3）用户登录入口处，密码是明文，而数据库中是密文。

4）明文与密文存在对应关系，但仅限于明文到密文的转换，即便是消息摘要算法泄露也不能反推出密码，增强了系统的安全性。

在这样的用户注册/登录模块中，常能见到MD5和SHA算法。为了提高安全性，往往将一些其他的不可变信息，如用户名、Email地址串入原始密码中，使得密码破译的难度加大。其

中，不可变信息称为“盐”，而这种处理方式常称为“加盐处理”，其实就是对原始信息的一堆混淆。在基于密码的加密算法PBE中，我们将再次看到这种处理方式。

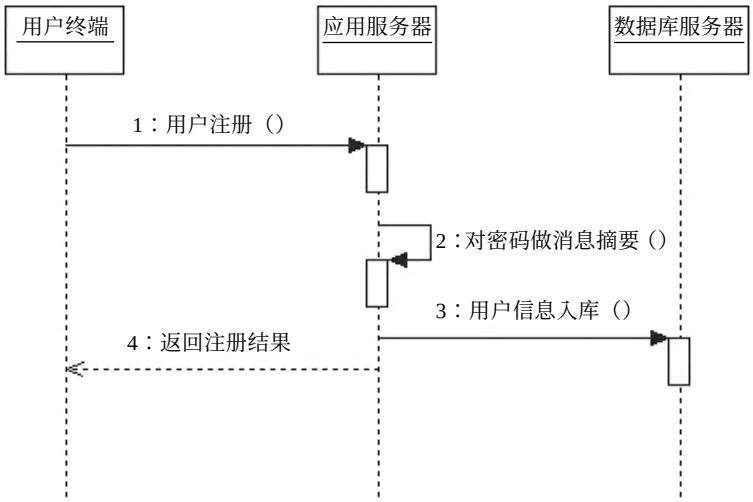


图6-2 用户注册

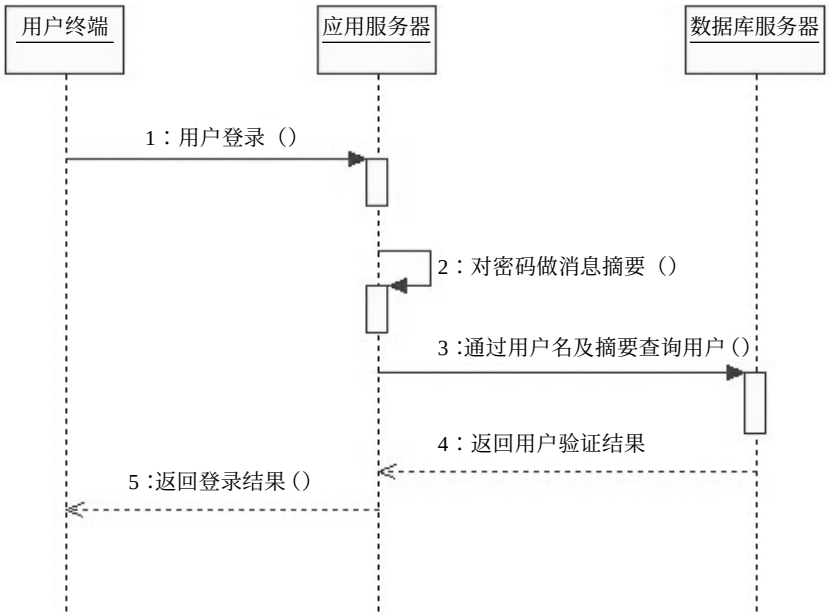


图6-3 用户登录

在一些开源架构中，例如Spring Security（http://static.springsource.org/spring-security/site/），使用了与上述流程相近的设计方案，并提供了MD5和SHA算法支持，以及对加盐处理方式的支持。

6.2.3 实现

MD系列算法的实现是通过MessageDigest类来完成的，如果需要以流的处理方式完成消息摘要，则需要使用DigestInputStream和DigestOutputStream，有关Java API请读者朋友参照第3章内容。Java 6仅支持MD2和MD5两种算法，通过第三方加密组件包Bouncy Castle（详见第4章），可支持MD4算法。

表6-1 MD系列算法

算法	摘要长度	备注
MD2	128	Java 6实现
MD5		
MD4		Bouncy Castle实现

MD系列算法支持如表6-1所示。

1. Sun

在Java 6中使用MD算法是很简单的。例如，要使用MD5算法对数据做消息摘要，可参考如下代码：

```
// 初始化MessageDigest，并指定MD5算法
MessageDigest md = MessageDigest.getInstance("MD5");
// 摘要处理
byte[] b = md.digest(data);
```

在上述代码中，data[]为待做消息摘要处理的数据，b[]是经过消息摘要处理后的摘要信息，也就是数字指纹。

Java 6支持MD2和MD5算法，如要使用MD2算法只需替换算法名即可，相关实现如代码清单6-1所示。

代码清单6-1 MD2和MD5算法实现

```
import java.security.MessageDigest;
/**
 * MD消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class MDCoder {
    /**
     * MD2消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeMD2(byte[] data) throws Exception {
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("MD2");
            // 执行消息摘要
            return md.digest(data);
    }
```

```
    /**
     * MD5消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeMD5(byte[] data) throws Exception {
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("MD5");
            // 执行消息摘要
            return md.digest(data);
    }
}
```

消息摘要的主要特点就是对同一段数据做多次摘要处理后，其摘要值完全一致。因此，我们通过必要两次消息摘要的结果来判别消息摘要是否一致，见代码清单6-2。

代码清单6-2 MD2和MD5算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * MD校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MDCoderTest {
    /**
     * 测试MD2
     * @throws Exception
     */
    @Test
    public final void testEncodeMD2() throws Exception {
            String str = "MD2消息摘要";
            // 获得摘要信息
            byte[] data1 = MDCoder.encodeMD2(str.getBytes());
            byte[] data2 = MDCoder.encodeMD2(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试MD5
     * @throws Exception
     */
    @Test
    public final void testEncodeMD5() throws Exception {
            String str = "MD5消息摘要";
```

```
            // 获得摘要信息
            byte[] data1 = MDCoder.encodeMD5(str.getBytes());
            byte[] data2 = MDCoder.encodeMD5(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
}
```

测试结果自然不用说，一定完全一致！

2. Bouncy Castle

第三方加密组件包Bouncy Castle是对Java 6的友善补充，不仅提供了Base64算法的实现，更是弥补了Sun未能提供MD4算法的空白。

使用Bouncy Castle支持MD4算法比较简单的办法是将Bouncy Castle导入项目中，并在初始化MessageDigest前通过Security类的addProvider()方法加载第三方加密组件包提供者，我们以BouncyCastleProvider为例，提供MD4算法支持。要使用MD4算法对数据做消息摘要，可参考如下代码：

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// 省略
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
// 初始化MessageDigest
MessageDigest md = MessageDigest.getInstance("MD4");
// 执行消息摘要
md.digest(data);
```

有关于第三方加密组件包Bouncy Castle的配置，读者朋友可阅读第4章相关内容。

下面给出MD4算法实现，如代码清单6-3所示。

代码清单6-3 MD4算法实现

```
import java.security.MessageDigest;
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.encoders.Hex;
/**
 * MD4消息摘要组件
 * @author梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class MD4Coder {
    /**
     * MD4消息摘要
     * @param data待做摘要处理的数据
     * @return byte[]消息摘要
```

```
     * @throws Exception
     */
    public static byte[] encodeMD4(byte[] data) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("MD4");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * MD4消息摘要
     * @param data 待做摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeMD4Hex(byte[] data) throws Exception {
            // 执行消息摘要
            byte[] b = encodeMD4(data);
            // 做十六进制编码处理
            return new String(Hex.encode(b));
    }
}
```

上述代码提供了MD4算法实现（encode MD4 ()方法）的同时，加入了十六进制转换方法实现（encodeMD4Hex()方法）。关于十六进制编码转换相关API请阅读第4章内容。

我们完全可以将Bouncy Castle的相关支持融入其他MD算法实现中，提供完整的MD系列算法，并加入十六进制转换实现。

对于上述方法实现给出对应的测试用例，见代码清单6-4。

代码清单6-4 MD4算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * MD4校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MD4CoderTest {
    /**
     * 测试MD4
     * @throws Exception
     */
    @Test
    public final void testEncodeMD4() throws Exception {
```

```
            String str = "MD4消息摘要";
            // 获得摘要信息
            byte[] data1 = MD4Coder.encodeMD4(str.getBytes());
            byte[] data2 = MD4Coder.encodeMD4(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试MD4Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeMD4Hex() throws Exception {
            String str = "MD4Hex消息摘要";
            // 获得摘要信息
            String data1 = MD4Coder.encodeMD4Hex(str.getBytes());
            String data2 = MD4Coder.encodeMD4Hex(str.getBytes());
            System.err.println("原文：\t" + str);
            System.err.println("MD4Hex-1：\t" + data1);
            System.err.println("MD4Hex-2：\t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
}
```

我们来关注一下testEncodeMD4Hex()方法在控制台上的输出结果，如下所示：

```
原文：      MD4Hex消息摘要
MD4Hex-1： 6d46694b818e0bab41eab6783749f963
MD4Hex-2： 6d46694b818e0bab41eab6783749f963
```

我们获得的两个摘要值都是32位的十六进制字符串，并且是一致的。

如果把十六进制转换的实现融入到其他MD算法的实现中，那岂不是相当完美？

3. Commons Codec

对于Commons Codec，想必读者朋友已经不再陌生。它实现了Base64算法，还提供了用于消息摘要的工具类——DigestUtils类（它位于org.apache.commons.codec.digest包中，请读者朋友阅读第4章相关内容。）。DigestUtils类是对Sun提供的MessageDigest类的一次封装，提供了MD5和SHA系列消息摘要算法的实现。

我们通过代码清单6-5来了解如何通过Commons Codec实现MD5算法。

代码清单6-5 MD5算法实现

```
import org.apache.commons.codec.digest.DigestUtils;
/**
 * MD5消息摘要组件
 * @author 梁栋
 * @version 1.0
```

```
 * @since 1.0
 */
public abstract class MD5Coder {
    /**
     * MD5消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeMD5(String data) throws Exception {
            // 执行消息摘要
            return DigestUtils.md5(data);
    }
    /**
     * MD5消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static String encodeMD5Hex(String data) throws Exception {
            // 执行消息摘要
            return DigestUtils.md5Hex(data);
    }
}
```

虽然Commons Codec只是对Sun提供的MD5算法实现做了一次简单的包装，但着实为我们使用该算法提供了不小的便利。相应的测试用例见代码清单6-6。

代码清单6-6 MD5算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * MD5校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MD5CoderTest {
    /**
     * 测试MD5
     * @throws Exception
     */
    @Test
    public final void testEncodeMD5() throws Exception {
            String str = "MD5消息摘要";
            // 获得摘要信息
            byte[] data1 = MD5Coder.encodeMD5(str);
```

```
            byte[] data2 = MD5Coder.encodeMD5(str);
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试MD5Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeMD5Hex() throws Exception {
            String str = "MD5Hex消息摘要";
            // 获得摘要信息
            String data1 = MD5Coder.encodeMD5Hex(str);
            String data2 = MD5Coder.encodeMD5Hex(str);
            System.err.println("原文：\t" + str);
            System.err.println("MD5Hex-1：\t" + data1);
            System.err.println("MD5Hex-2：\t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
}
```

我们来关注一下testEncodeMD5Hex()方法在控制台上的输出结果，如下所示：

```
原文：      MD5Hex消息摘要
MD5Hex-1： 4effa30b56b61156bcd005eb9968cc9b
MD5Hex-2： 4effa30b56b61156bcd005eb9968cc9b
```

对同一个消息做MD5Hex处理后，得到的摘要值都是32位的十六进制字符串，并且是一致的。

4. 三种实现方式的差异

三种实现方式以Sun提供的实现为基础，在算法支持上和方法易用性上提供了更好的扩展与支持。

❑ Sun

Sun提供的算法实现较为底层，支持MD2和MD5两种算法。但缺少了相应的进制转换实现，不能将其字节数组形式的摘要信息转为十六进制字符串，这多少有点不方便。

❑ Bouncy Castle

Bouncy Castle是对Sun的友善补充，提供了对MD4算法的支持。支持多种形式的参数，支持十六进制字符串形式的摘要信息。

❑ Commons Codec

如果仅仅需要实现MD5算法的话，使用Commons Codec完成消息摘要处理是一个不错的选择。它支持多种形式的参数，支持十六进制字符串形式的摘要信息。

综上所述，我们可以根据需要有机地结合两种实现方式，满足系统的需要。如果只要求MD5算法支持，Commons Codec是首选；若要支持MD2或MD4算法，或者要求在此基础上获

得十六进制编码结果，就让Sun和Bouncy Castle联姻。

6.3 SHA算法家族

SHA算法基于MD4算法基础之上，作为MD算法的继任者，成为了新一代的消息摘要算法的代表。SHA与MD算法不同之处主要在于摘要长度，SHA算法的摘要长度更长，安全性更高。

6.3.1 简述

SHA（Secure Hash Algorithm，安全散列算法）是消息摘要算法的一种，被广泛认可为MD5算法的继任者。它是由美国国家安全局（NSA，National Security Agency）设计，经美国国家标准与技术研究院（NIST，National Institute of Standards and Technology）发布的一系列密码散列函数。SHA算法家族目前共有SHA-1、SHA-224、SHA-256、SHA-384和SHA-512五种算法，通常将后四种算法并称为SHA-2算法。除上述五种算法外，还有发布不久就夭折的SHA-0算法。

SHA算法是在MD4算法的基础上演进而来的，通过SHA算法同样能够获得一个固定长度的摘要信息。与MD系列算法不同的是：若输入的消息不同，则与其相对应的摘要信息的差异概率很高。SHA算法是FIPS所认证的五种安全杂凑算法。

这些算法中的“安全”字眼是基于以下两点（根据官方标准的描述）：

1）由消息摘要反推原输入讯息，从计算理论上来说是很困难的。

2）想要找到两组不同的消息对应到相同的消息摘要，从计算理论上来说也是很困难的。任何对输入消息的变动，都有很高的机率导致其产生的消息摘要迥异。

随着时间的推移，安全算法已不再安全。继山东大学王小云教授顺利破解MD5算法后，SHA-1算法也难逃此劫，终被王小云教授破解。两大著名消息摘要算法被破解，预示着数字签名在理论上可被伪造，B2B和B2C系统将存在安全隐患。

让我们简要回顾一下SHA算法的发展历史：

❑ SHA-0算法

1993年，NIST公布了SHA算法家族的第一个版本，FIPS PUB 180。为避免混淆，现在我们称它为SHA-0算法。但SHA-0算法在公布不久后就被NSA撤回，原因是NSA发现SHA-0算法中含有会降低密码安全性的错误。由此，SHA-0算法还未正式推广就已夭折。

❑ SHA-1算法

1995年，继SHA-0算法夭折后，NIST发布了FIPS PUB 180的修订版本 FIPS PUB 180-1，用于取代FIPS PUB 180，称为SHA-1算法，通常我们也把SHA1算法简称为SHA算法。SHA-1算法在许多安全协定中广为使用，包括TLS/SSL、PGP、SSH、S/MIME 和IPsec，曾被视为是MD5算法的后继者。SHA-0和SHA-1算法可对最大长度为264的字节信息做摘要处理，得到一个160位的摘要信息，其设计原理相似于MD4和MD5算法。如果将得到160位的摘要信息换算成十六进制，可以得到一个40位（每4位二进制数转换为1位十六进制数）的字符串。

有关SHA-1算法详情请参见RFC 3174（http://www.ietf.org/rfc/rfc3174.txt）。

我们常使用的数字证书就有SHA-1算法的影子，如图6-4所示。图中指纹值正是一个40位的十六进制字符串（51ee11819f269a961671a5bd77cba3f0815103c8）。

图6-4 数字证书中的SHA-1算法

❑ SHA-2算法

SHA算法家族除了其代表SHA-1算法以外，还有SHA-224、SHA-256、SHA-384和SHA-512四种SHA算法的变体，以其摘要信息字节长度不同而命名，通常将这组算法并称为SHA-2算法。摘要信息字节长度的差异是SHA-2和SHA-1算法的最大差异。

2001年，在FIPS PUB 180-2草稿中包含了SHA-256、SHA-384和SHA-512算法，随即通过了审查和评论，于2002年以官方标准发布。

2004年2月，在FIPS PUB 180-2变更通知中加入了一个额外的变种“SHA-224”，这是为了符合双金钥3DES（三重DES算法）所需的金钥长度而定义的。

6.3.2 模型分析

如果甲乙双方进行明文消息通讯，但要求能够鉴别消息在传输过程中是否被篡改，就可参照如图6-5所示的模型。当然，此模型对MD和SHA算法同样适用。

如图所示，甲乙双方分别作为消息发送者与接收者。我们也可以把甲方看做软件发布厂商，把乙方看做软件使用客户。甲方向乙方发送一则消息，需要经历以下几个步骤：

1）甲方公布消息摘要算法。这个算法可以被其他人获取，包括监听者。

2）甲方对待发送的消息做摘要处理，并获得摘要信息。

3）甲方向乙方发送摘要信息。这个数字指纹可以被其他人获取，包括监听者。

4）甲方向乙方发送消息。该消息可以是明文，有可能被监听者篡改。

5）乙方获得消息后，使用甲方提供的消息摘要算法对其做摘要处理，获得数字指纹并对比甲方传递过来的数字指纹是否匹配。

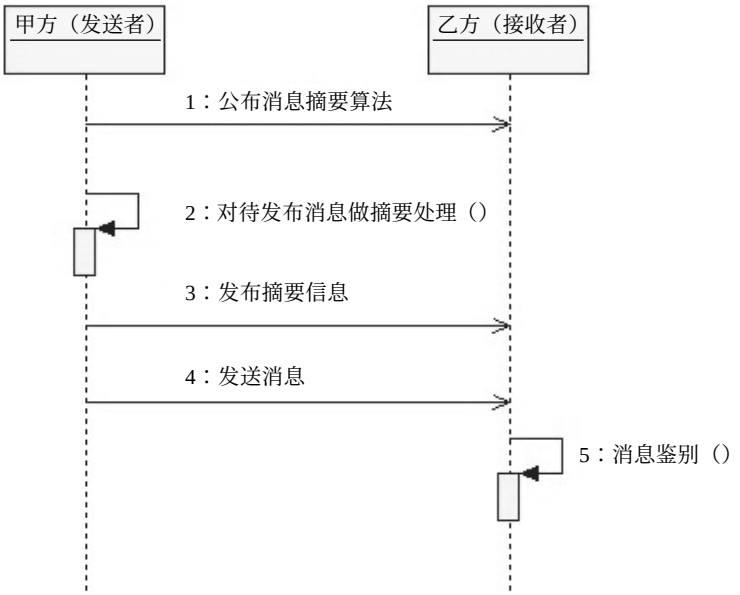


图6-5 基于MD/SHA算法的消息传递

如果两个数字指纹相同则说明消息来源于甲方，否则，甲方传递的消息很可能在传递途中被篡改。

用这个流程图来分析并校验数据的完整性，确定下载到的文件是不是有效的再合适不过了。一般小型的网络交互应用都可参照此模型设计，完成消息的发送与回复。

6.3.3 实现

在Java 6中，MessageDigest类支持MD算法的同时也支持SHA算法，几乎涵盖了我们目前所知的全部SHA系列算法，主要包含SHA-1、SHA-256、SHA-384和SHA-512四种算法。通过第三方加密组件包Bouncy Castle（详见本书附录），可支持SHA-224算法。

SHA系列算法支持如表6-2所示。

表6-2 SHA系列算法

算法	摘要长度	备注
SHA-1	160	Java 6实现
SHA-256	256	
SHA-384	384	
SHA-512	512	
SHA-224	224	Bouncy Castle实现

1. Sun

在Java 6中，“ SHA ”是“ SHA-1 ”的简称，两种算法名称等同。如果要使用SHA-1算法对数据做消息摘要，可参考如下代码：

```
// 初始化MessageDigest，并指定SHA算法
MessageDigest md = MessageDigest.getInstance("SHA");
// 摘要处理
byte[] b = md.digest(data);
```

在上述代码中，data[]为待做消息摘要处理的数据，b[]是经过消息摘要处理后的摘要信息。Java 6支持SHA-1、SHA-256、SHA-384和SHA-512四种算法，实现方式如代码清单6-7所示。

代码清单6-7 SHA算法实现1

```
import java.security.MessageDigest;
/**
 * SHA消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class SHACoder {
    /**
     * SHA-1消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA(byte[] data) throws Exception {
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("SHA");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * SHA-256消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA256(byte[] data) throws Exception {
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("SHA-256");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * SHA-384消息摘要
```

```
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA384(byte[] data) throws Exception {
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("SHA-384");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * SHA-512消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA512(byte[] data) throws Exception {
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("SHA-512");
            // 执行消息摘要
            return md.digest(data);
    }
}
```

对上述代码做测试用例，如代码清单6-8所示。

代码清单6-8 SHA算法实现1测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * SHA校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class SHACoderTest {
    /**
     * 测试SHA-1
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA() throws Exception {
            String str = "SHA1消息摘要";
            // 获得摘要信息
            byte[] data1 = SHACoder.encodeSHA(str.getBytes());
            byte[] data2 = SHACoder.encodeSHA(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
```

```
    }
    /**
     * 测试SHA-256
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA256() throws Exception {
            String str = "SHA256消息摘要";
            // 获得摘要信息
            byte[] data1 = SHACoder.encodeSHA256(str.getBytes());
            byte[] data2 = SHACoder.encodeSHA256(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试SHA-384
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA384() throws Exception {
            String str = "SHA384消息摘要";
            // 获得摘要信息
            byte[] data1 = SHACoder.encodeSHA384(str.getBytes());
            byte[] data2 = SHACoder.encodeSHA384(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试SHA-512
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA512() throws Exception {
            String str = "SHA512消息摘要";
            // 获得摘要信息
            byte[] data1 = SHACoder.encodeSHA512(str.getBytes());
            byte[] data2 = SHACoder.encodeSHA512(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
}
```

作者和读者朋友一定都会这样想：如果能增加十六进制转换的方法，那就更加方便了！

2. Bouncy Castle

Bouncy Castle不仅支持MD4算法，同时也支持SHA-224算法。

比较简单的实现方式如下所示：

```
import java.security.Security;
```

```
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
// 初始化MessageDigest
MessageDigest md = MessageDigest.getInstance("SHA-224");
```

有关第三方加密组件包Bouncy Castle，读者可参考第3章和第4章相关内容。

对于SHA算法，我们仅仅需要补充SHA-244算法实现，如代码清单6-9所示。

代码清单6-9 SHA224算法实现

```
import java.security.MessageDigest;
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.encoders.Hex;
/**
 * SHA-224消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class SHA224Coder {
    /**
     * SHA-224消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA224(byte[] data) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("SHA-224");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * SHA-224消息摘要
     * @param data 待做摘要处理的数据
     * @return String 十六进制消息摘要
     * @throws Exception
     */
    public static String encodeSHA224Hex(byte[] data) throws Exception {
            // 执行消息摘要
            byte[] b = encodeSHA224(data);
            // 做十六进制编码处理
            return new String(Hex.encode(b));
    }
}
```

对于上述实现做测试用例也相当简单，如代码清单6-10所示。

代码清单6-10 SHA224算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * SHA-224校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class SHA224CoderTest {
    /**
     * 测试SHA-224
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA224() throws Exception {
            String str = "SHA224消息摘要";
            // 获得摘要信息
            byte[] data1 = SHACoder.encodeSHA224(str.getBytes());
            byte[] data2 = SHACoder.encodeSHA224(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试SHA-224
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA224Hex() throws Exception {
            String str = "SHA224消息摘要";
            // 获得摘要信息
            String data1 = SHA224Coder.encodeSHA224Hex (str.getBytes());
            String data2 = SHA224Coder.encodeSHA224Hex (str.getBytes());
            // 校验
            assertEquals(data1, data2);
    }
}
```

对于其他SHA算法，我们完全可以对其加工，加入十六进制编码转换实现。

我们来关注控制台输出的信息，如下所示：

```
原文：            SHA224Hex消息摘要
SHA224Hex-1：    fd0c016cc823d094aafc4d64665837df8ffa1d1712f58e5c44fd20d2
SHA224Hex-2：    fd0c016cc823d094aafc4d64665837df8ffa1d1712f58e5c44fd20d2
```

在上述内容中，我们得到了28位的摘要信息，换算成二进制正好是224位。

3. Commons Codec

DigestUtils类除了MD5算法外，还支持多种SHA系列算法，涵盖了Java 6所支持的全部SHA算法。有关DigestUtils类相关SHA算法支持API，读者可阅读第4章内容。

Commons Codec与Sun所提供的SHA算法实现在本质上毫无差别，关键在于Commons Codec提供了更为方便的实现。我们对上述SHA算法实现做了调整，相关实现如代码清单6-11所示。

代码清单6-11 SHA算法实现2

```
import org.apache.commons.codec.digest.DigestUtils;
/**
 * SHA消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class SHACoder {
    /**
     * SHA消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA(String data) throws Exception {
            // 执行消息摘要
            return DigestUtils.sha(data);
    }
    /**
     * SHAHex消息摘要
     * @param data 待做摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeSHAHex(String data) throws Exception {
            // 执行消息摘要
            return DigestUtils.shaHex(data);
    }
    /**
     * SHA256消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA256(String data) throws Exception {
            // 执行消息摘要
```

```
        return DigestUtils.sha256(data);
    }
    /**
     * SHA256Hex消息摘要
     * @param data 待做摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeSHA256Hex(String data) throws Exception {
        // 执行消息摘要
        return DigestUtils.sha256Hex(data);
    }
    /**
     * SHA384消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA384(String data) throws Exception {
        // 执行消息摘要
        return DigestUtils.sha384(data);
    }
    /**
     * SHA384Hex消息摘要
     * @param data 待做摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeSHA384Hex(String data) throws Exception {
        // 执行消息摘要
        return DigestUtils.sha384Hex(data);
    }
    /**
     * SHA512Hex消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA512(String data) throws Exception {
        // 执行消息摘要
        return DigestUtils.sha512(data);
    }
    /**
     * SHA512Hex消息摘要
     * @param data 待做摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
```

```
    public static String encodeSHA512Hex(String data) throws Exception {
            // 执行消息摘要
            return DigestUtils.sha512Hex(data);
    }
}
```

对于上述实现做测试用例，如代码清单6-12所示。

代码清单6-12 SHA算法实现2测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * SHAHex校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class SHACoderTest {
    /**
     * 测试SHA-1
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA() throws Exception {
            String str = "SHA1消息摘要";
            // 获得摘要信息
            byte[] data1 = SHACoder.encodeSHA(str);
            byte[] data2 = SHACoder.encodeSHA(str);
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试SHA-1Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeSHAHex() throws Exception {
            String str = "SHA-1Hex消息摘要";
            // 获得摘要信息
            String data1 = SHACoder.encodeSHAHex(str);
            String data2 = SHACoder.encodeSHAHex(str);
            System.err.println("原文：\t" + str);
            System.err.println("SHA1Hex-1：\t" + data1);
            System.err.println("SHA1Hex-2：\t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
```

```
/**
 * 测试SHA-256
 * @throws Exception
 */
@Test
public final void testEncodeSHA256() throws Exception {
        String str = "SHA256消息摘要";
        // 获得摘要信息
        byte[] data1 = SHACoder.encodeSHA256(str);
        byte[] data2 = SHACoder.encodeSHA256(str);
        // 校验
        assertArrayEquals(data1, data2);
}
/**
 * 测试SHA-256Hex
 * @throws Exception
 */
@Test
public final void testEncodeSHA256Hex() throws Exception {
        String str = "SHA256Hex消息摘要";
        // 获得摘要信息
        String data1 = SHACoder.encodeSHA256Hex(str);
        String data2 = SHACoder.encodeSHA256Hex(str);
        System.err.println("原文：\t" + str);
        System.err.println("SHA256Hex-1：\t" + data1);
        System.err.println("SHA256Hex-2：\t" + data2);
        // 校验
        assertEquals(data1, data2);
}
/**
 * 测试SHA-384
 * @throws Exception
 */
@Test
public final void testEncodeSHA384() throws Exception {
        String str = "SHA384消息摘要";
        // 获得摘要信息
        byte[] data1 = SHACoder.encodeSHA384(str);
        byte[] data2 = SHACoder.encodeSHA384(str);
        // 校验
        assertArrayEquals(data1, data2);
}
/**
 * 测试SHA-384Hex
 * @throws Exception
 */
@Test
public final void testEncodeSHA384Hex() throws Exception {
```

```
        String str = "SHA384Hex消息摘要";
        // 获得摘要信息
        String data1 = SHACoder.encodeSHA384Hex(str);
        String data2 = SHACoder.encodeSHA384Hex(str);
        System.err.println("原文：\t" + str);
        System.err.println("SHA384Hex-1：\t" + data1);
        System.err.println("SHA384Hex-2：\t" + data2);
        // 校验
        assertEquals(data1, data2);
    }
    /**
     * 测试SHA-512
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA512() throws Exception {
        String str = "SHA512消息摘要";
        // 获得摘要信息
        byte[] data1 = SHACoder.encodeSHA512(str);
        byte[] data2 = SHACoder.encodeSHA512(str);
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试SHA-512Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA512Hex() throws Exception {
        String str = "SHA512Hex消息摘要";
        // 获得摘要信息
        String data1 = SHACoder.encodeSHA512Hex(str);
        String data2 = SHACoder.encodeSHA512Hex(str);
        System.err.println("原文：\t" + str);
        System.err.println("SHA512Hex-1：\t" + data1);
        System.err.println("SHA512Hex-2：\t" + data2);
        // 校验
        assertEquals(data1, data2);
    }
}
```

在控制台中，我们可以清晰地观察到4种SHA算法的摘要信息长度有所不同。

❑ SHA-1

```
原文：          SHA-1Hex消息摘要
SHA1Hex-1：    9a4135598d12e61f54d5d790887cf47721cbdd2c
SHA1Hex-2：    9a4135598d12e61f54d5d790887cf47721cbdd2c
```

SHA-1算法的摘要信息是一个40位的十六进制字符串，换算成二进制正好是160位。

❑ SHA-256

```
原文：          SHA256Hex消息摘要
SHA256Hex-1：e001852cd945459a14ecd83ec3f9c73b94e02ec12ec8afbf764a98719acce777
SHA256Hex-2：e001852cd945459a14ecd83ec3f9c73b94e02ec12ec8afbf764a98719acce777
```

SHA-256算法的摘要信息是一个64位的十六进制字符串，换算成二进制正好是256位。

❑ SHA-384和SHA-512

```
原文：          SHA384Hex消息摘要
SHA384Hex-1：
13d87b76ee7a14fcbb7d96c90ed430b8793cdd75afa10700a402ebdce463a29302ce36b19db4f30f
e170399897f191ed
SHA384Hex-2：
13d87b76ee7a14fcbb7d96c90ed430b8793cdd75afa10700a402ebdce463a29302ce36b19db4f30f
e170399897f191ed
原文：          SHA512Hex消息摘要
SHA512Hex-1：
0ecc3742f5ca400c78f84d4bf37a7e34aff371f879d773cab40fe1cd4cb19b6cf2bc598e4bc5a384
808b9b15ef19ff59db4793f0f6b3815bdfd6152208e26756
SHA512Hex-2：
0ecc3742f5ca400c78f84d4bf37a7e34aff371f879d773cab40fe1cd4cb19b6cf2bc598e4bc5a384
808b9b15ef19ff59db4793f0f6b3815bdfd6152208e26756
```

很显然，SHA-384和SHA-512算法的摘要信息没有任何悬念，分别对应384位和512位的二进制数。

消息摘要长度与安全强度成正比。从这一点来说，SHA系列算法比MD系列算法更具优势。MD系列算法仅有128位，而SHA算法则可以从160位扩充到512位，更具安全性。

4. 三种实现方式的差异

三种实现方式以Sun提供的实现为基础，在算法支持上和方法易用性上提供了更好的扩展与支持。

❑ Sun

由于Sun提供了较为底层的SHA算法实现，如SHA-1、SHA-256、SHA-384和SHA-512四种算法，但缺少了对应的进制转换实现，多少有些遗憾。

❑ Bouncy Castle

Bouncy Castle是对Sun的友善补充，提供了对SHA-224算法的支持，支持十六进制字符串形式的摘要信息，相当方便。

❑ Commons Codec

Commons Codec对Sun提供的SHA算法做了包装，支持多种形式的参数，支持十六进制字符串形式的摘要信息，相当方便。

综上所述，若在应用中使用SHA系列算法，且不要求对SHA-224算法提供支持，Commons Codec是更为合适的选择。如果只是需要在Sun原有SHA系列算法支持的基础上加入十六进制编

码转换，Bouncy Castle和Commons Codec都是不错的选择。

6.4 MAC算法家族

MAC算法结合了MD5和SHA算法的优势，并加入密钥的支持，是一种更为安全的消息摘要算法。

6.4.1 简述

MAC（Message Authentication Code，消息认证码算法）是含有密钥散列函数算法，兼容了MD和SHA算法的特性，并在此基础上加入了密钥。因此，我们也常把MAC称为HMAC（keyed-Hash Message Authentication Code）。

MAC算法主要集合了MD和SHA两大系列消息摘要算法。MD系列算法有HmacMD2、HmacMD4和HmacMD5三种算法；SHA系列算法有HmacSHA1、HmacSHA224、HmacSHA256、HmacSHA384和HmacSHA512五种算法。

经MAC算法得到的摘要值也可以使用十六进制编码表示，其摘要值长度与参与实现的算法摘要值长度相同。例如，HmacSHA1算法得到的摘要长度就是SHA1算法得到的摘要长度，都是160位二进制数，换算成十六进制编码为40位。

有关HMac算法详情请参见RFC 2104（http://www.ietf.org/rfc/rfc2104.txt），其中包含了HmacMD5算法的C语言版实现。

基于SSH（Secure Shell，安全外壳）协议的一些软件，也使用了AES算法。图6-6中展示了SecureCRT软件中如何配置加密算法。

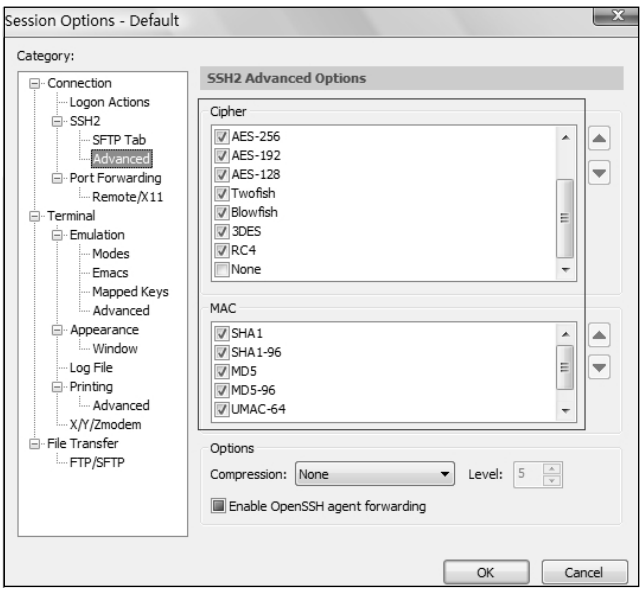


图6-6 SecureCRT配置界面中的加密算法

6.4.2 模型分析

在本章6.4.2节中，我们分析了基于MD/SHA算法单向消息传递模型中消息摘要算法的作用。如果需要更为安全的消息传递，就需要使用到MAC算法了。基于MAC算法的消息传递模型如图6-7所示。

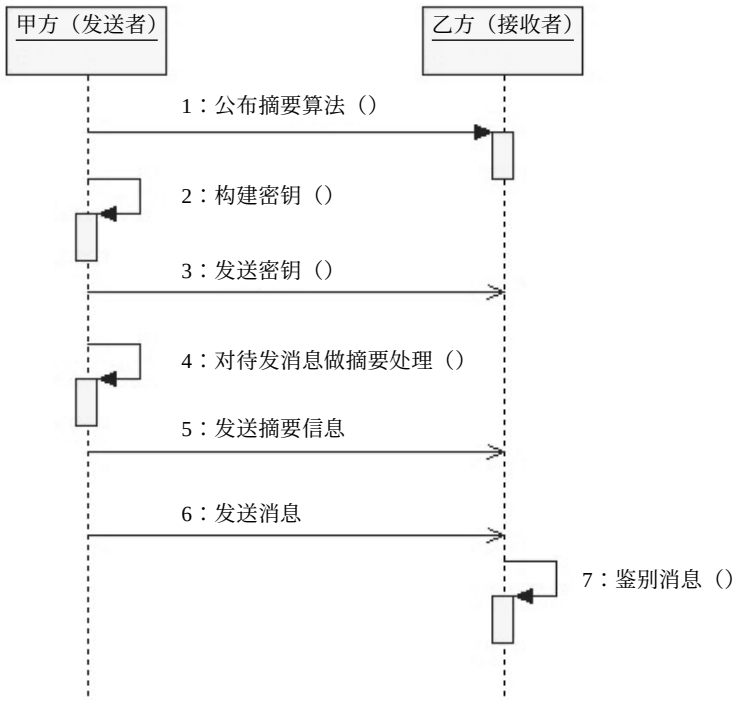


图6-7 基于MAC算法的消息传递模型

甲乙双方可按如下流程完成交互：

1）甲方向乙方公布摘要算法。

2）甲乙双方按照约定构建密钥，并由一方公布给对方（这里是甲方公布密钥给乙方）。

3）甲方使用密钥对消息做摘要处理后，将摘要信息发送给乙方。

4）甲方将消息发送给乙方。

5）乙方收到消息后，使用密钥对消息做摘要处理，并对比甲方发送的摘要信息是否一致。

在上述流程中，只要双方商榷了密钥就可以进行消息交互了。

6.4.3 实现

在Java 6中，MAC系列算法需要通过Mac类提供支持。有MAC算法相关的API请读者朋友参照第3章内容。

Java 6中仅仅提供了HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和HmacSHA512四种算法，而第三方加密组件包Bouncy Castle补充了HmacMD2、HmacMD4和

HmacSHA224三种算法支持。有关Bouncy Castle的内容请见附录。

MAC系列算法支持如表6-3所示。

表6-3 MAC系列算法

算法	摘要长度	备注
HmacMD5	128	Java 6实现
HmacSHA1	160	
HmacSHA256	256	
HmacSHA384	384	
HmacSHA512	512	
HmacMD2	128	Bouncy Castle实现
HmacMD4	128	
HmacSHA224	224	

1. Sun

Mac算法是带有密钥的消息摘要算法，所以实现起来要分两步：

1）构建密钥。

2）执行消息摘要。

对应上述步骤，以HmacMD5算法为例，构建密钥代码如下所示：

```
// 初始化KeyGenerator
KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD5");
// 产生密钥
SecretKey secretKey = keyGenerator.generateKey();
// 获得密钥
byte[] key = secretKey.getEncoded();
```

上述代码中的字节数组key[]就是我们构造的密钥。

我们需要对其还原，得到密钥，参考如下代码：

```
// 还原密钥
SecretKey secretKey = new SecretKeySpec(key, "HmacMD5");
```

获得密钥后，我们就可以按如下代码做消息摘要了，参考如下代码：

```
// 实例化Mac
Mac mac = Mac.getInstance(secretKey.getAlgorithm());
// 初始化Mac
mac.init(secretKey);
// 执行消息摘要
byte[] data = mac.doFinal(data);
```

上述代码中的字节数组data[]就是我们获得的摘要结果了。

Sun在Java 6中提供了HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和HmacSHA512四种算法支持，算法实现如代码清单6-13所示。

代码清单6-13 MAC算法实现1

```
import javax.crypto.KeyGenerator;
import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
/**
 * MAC消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class MACCoder {
    /**
     * 初始化HmacMD5密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] initHmacMD5Key() throws Exception {
            // 初始化KeyGenerator
            KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD5");
            // 产生密钥
            SecretKey secretKey = keyGenerator.generateKey();
            // 获得密钥
            return secretKey.getEncoded();
    }
    /**
     * HmacMD5消息摘要
     * @param data 待做摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacMD5(byte[] data, byte[] key)
                    throws Exception {
            // 还原密钥
            SecretKey secretKey = new SecretKeySpec(key, "HmacMD5");
            // 实例化Mac
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            // 初始化Mac
            mac.init(secretKey);
            // 执行消息摘要
            return mac.doFinal(data);
    }
    /**
     * 初始化HmacSHA1密钥
     * @return byte[] 密钥
     * @throws Exception
```

```
 */
public static byte[] initHmacSHAKey() throws Exception {
        // 初始化KeyGenerator
        KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA1");
        // 产生密钥
        SecretKey secretKey = keyGenerator.generateKey();
        // 获得密钥
        return secretKey.getEncoded();
}
/**
 * HmacSHA1消息摘要
 * @param data 待做摘要处理的数据
 * @param key 密钥
 * @return byte[] 消息摘要
 * @throws Exception
 */
public static byte[] encodeHmacSHA(byte[] data, byte[] key)
              throws Exception {
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacSHA1");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
}
/**
 * 初始化HmacSHA256密钥
 * @return byte[] 密钥
 * @throws Exception
 */
public static byte[] initHmacSHA256Key() throws Exception {
        // 初始化KeyGenerator
        KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA256");
        // 产生密钥
        SecretKey secretKey = keyGenerator.generateKey();
        // 获得密钥
        return secretKey.getEncoded();
}
/**
 * HmacSHA256消息摘要
 * @param data 待做摘要处理的数据
 * @param key 密钥
 * @return byte[] 消息摘要
 * @throws Exception
 */
public static byte[] encodeHmacSHA256(byte[] data, byte[] key)
```

```
            throws Exception {
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacSHA256");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
    }
    /**
     * 初始化HmacSHA384密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] initHmacSHA384Key() throws Exception {
        // 初始化KeyGenerator
        KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA384");
        // 产生密钥
        SecretKey secretKey = keyGenerator.generateKey();
        // 获得密钥
        return secretKey.getEncoded();
    }
    /**
     * HmacSHA384消息摘要
     * @param data 待做摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacSHA384(byte[] data, byte[] key) throws Exception {
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacSHA384");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
    }
    /**
     * 初始化HmacSHA512密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] initHmacSHA512Key() throws Exception {
        // 初始化KeyGenerator
```

```
            KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA512");
            // 产生密钥
            SecretKey secretKey = keyGenerator.generateKey();
            // 获得密钥
            return secretKey.getEncoded();
    }
    /**
     * HmacSHA512消息摘要
     * @param data 待做摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacSHA512(byte[] data, byte[] key) throws Exception {
            // 还原密钥
            SecretKey secretKey = new SecretKeySpec(key, "HmacSHA512");
            // 实例化Mac
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            // 初始化Mac
            mac.init(secretKey);
            // 执行消息摘要
            return mac.doFinal(data);
    }
}
```

对于上述代码的测试较为简单，如代码清单6-14所示。

代码清单6-14 MAC算法实现1测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * MAC校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MACCoderTest {
    /**
     * 测试HmacMD5
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacMD5() throws Exception {
            String str = "HmacMD5消息摘要";
            // 初始化密钥
            byte[] key = MACCoder.initHmacMD5Key();
            // 获得摘要信息
            byte[] data1 = MACCoder.encodeHmacMD5(str.getBytes(), key);
```

```
        byte[] data2 = MACCoder.encodeHmacMD5(str.getBytes(), key);
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试HmacSHA1
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacSHA() throws Exception {
        String str = "HmacSHA1消息摘要";
        // 初始化密钥
        byte[] key = MACCoder.initHmacSHAKey();
        // 获得摘要信息
        byte[] data1 = MACCoder.encodeHmacSHA(str.getBytes(), key);
        byte[] data2 = MACCoder.encodeHmacSHA(str.getBytes(), key);
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试HmacSHA256
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacSHA256() throws Exception {
        String str = "HmacSHA256消息摘要";
        // 初始化密钥
        byte[] key = MACCoder.initHmacSHA256Key();
        // 获得摘要信息
        byte[] data1 = MACCoder.encodeHmacSHA256(str.getBytes(), key);
        byte[] data2 = MACCoder.encodeHmacSHA256(str.getBytes(), key);
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试HmacSHA384
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacSHA384() throws Exception {
        String str = "HmacSHA384消息摘要";
        // 初始化密钥
        byte[] key = MACCoder.initHmacSHA384Key();
        // 获得摘要信息
        byte[] data1 = MACCoder.encodeHmacSHA384(str.getBytes(), key);
        byte[] data2 = MACCoder.encodeHmacSHA384(str.getBytes(), key);
        // 校验
```

```
            assertArrayEquals(data1, data2);
        }
        /**
         * 测试HmacSHA512
         * @throws Exception
         */
        @Test
        public final void testEncodeHmacSHA512() throws Exception {
            String str = "HmacSHA512消息摘要";
            // 初始化密钥
            byte[] key = MACCoder.initHmacSHA512Key();
            // 获得摘要信息
            byte[] data1 = MACCoder.encodeHmacSHA512(str.getBytes(), key);
            byte[] data2 = MACCoder.encodeHmacSHA512(str.getBytes(), key);
            // 校验
            assertArrayEquals(data1, data2);
        }
    }
```

上述代码实现唯一不足之处在于缺少了十六进制编码转换实现，可以使用Bouncy Castle或Commons Codec的十六进制编码转换实现来做补充。

2. Bouncy Castle

第三方加密组件包Bouncy Castle作为补充，提供了HmacMD2、HmacMD4和HmacSHA224三种算法支持，弥补了Sun在Java 6中未能提供相关算法实现的缺憾。

比较简单的使用方式是将其jar包导入项目中，在做初始化密钥和消息摘要前，执行如下代码：

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// 省略
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
```

对于上述算法的实现，如代码清单6-15所示。

代码清单6-15 MAC算法实现2

```
import java.security.Security;
import javax.crypto.KeyGenerator;
import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.encoders.Hex;
/**
 * MAC消息摘要组件
 * @author 梁栋
 * @version 1.0
```

```
 * @since 1.0
 */
public abstract class MACCoder {
    /**
     * 初始化HmacMD2密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] initHmacMD2Key() throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化KeyGenerator
            KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD2");
            // 产生秘密密钥
            SecretKey secretKey = keyGenerator.generateKey();
            // 获得密钥
            return secretKey.getEncoded();
    }
    /**
     * HmacMD2消息摘要
     * @param data 待做消息摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacMD2(byte[] data, byte[] key) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 还原密钥
            SecretKey secretKey = new SecretKeySpec(key, "HmacMD2");
            // 实例化Mac
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            // 初始化Mac
            mac.init(secretKey);
            // 执行消息摘要
            return mac.doFinal(data);
    }
    /**
     * HmacMD2Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @param String 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static String encodeHmacMD2Hex(byte[] data, byte[] key) throws Exception {
            // 执行消息摘要
            byte[] b = encodeHmacMD2(data, key);
            // 做十六进制转换
```

```
        return new String(Hex.encode(b));
}
/**
 * 初始化HmacMD4密钥
 * @return byte[] 密钥
 * @throws Exception
 */
public static byte[] initHmacMD4Key() throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 初始化KeyGenerator
        KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD4");
        // 产生秘密密钥
        SecretKey secretKey = keyGenerator.generateKey();
        // 获得密钥
        return secretKey.getEncoded();
}
/**
 * HmacMD4消息摘要
 * @param data 待做消息摘要处理的数据
 * @param key 密钥
 * @return byte[] 消息摘要
 * @throws Exception
 */
public static byte[] encodeHmacMD4(byte[] data, byte[] key) throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacMD4");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
}
/**
 * HmacMD4Hex消息摘要
 * @param data 待做消息摘要处理的数据
 * @param key 密钥
 * @return String 消息摘要
 * @throws Exception
 */
public static String encodeHmacMD4Hex(byte[] data, byte[] key) throws Exception {
        // 执行消息摘要
        byte[] b = encodeHmacMD4(data, key);
```

```
        // 做十六进制转换
        return new String(Hex.encode(b));
    }
    /**
     * 初始化HmacSHA224密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] initHmacSHA224Key() throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 初始化KeyGenerator
        KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA224");
        // 产生秘密密钥
        SecretKey secretKey = keyGenerator.generateKey();
        // 获得密钥
        return secretKey.getEncoded();
    }
    /**
     * HmacSHA224消息摘要
     * @param data 待做消息摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacSHA224(byte[] data, byte[] key) throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacSHA224");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
    }
    /**
     * HmacSHA224Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @param key 密钥
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeHmacSHA224Hex(byte[] data, byte[] key) throws Exception {
        // 执行消息摘要
```

```
            byte[] b = encodeHmacSHA224(data, key);
            // 做十六进制转换
            return new String(Hex.encode(b));
    }
}
```

对于上述实现做相关测试，如代码清单6-16所示。

代码清单6-16 MAC算法实现2测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * MAC校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MACCoderTest {
    /**
     * 测试HmacMD2
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacMD2() throws Exception {
            String str = "HmacMD2消息摘要";
            // 初始化密钥
            byte[] key = MACCoder.initHmacMD2Key();
            // 获得摘要信息
            byte[] data1 = MACCoder.encodeHmacMD2(str.getBytes(), key);
            byte[] data2 = MACCoder.encodeHmacMD2(str.getBytes(), key);
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试HmacMD4
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacMD4() throws Exception {
            String str = "HmacMD4消息摘要";
            // 初始化密钥
            byte[] key = MACCoder.initHmacMD4Key();
            // 获得摘要信息
            byte[] data1 = MACCoder.encodeHmacMD4(str.getBytes(), key);
            byte[] data2 = MACCoder.encodeHmacMD4(str.getBytes(), key);
            // 校验
            assertArrayEquals(data1, data2);
```

```
    }
    /**
     * 测试HmacSHA224
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacSHA224() throws Exception {
            String str = "HmacSHA224消息摘要";
            // 初始化密钥
            byte[] key = MACCoder.initHmacSHA224Key();
            // 获得摘要信息
            byte[] data1 = MACCoder.encodeHmacSHA224(str.getBytes(), key);
            byte[] data2 = MACCoder.encodeHmacSHA224(str.getBytes(), key);
            // 校验
            assertArrayEquals(data1, data2);
    }
}
```

观察控制台输出的信息，如下所示：

```
原文：              HmacMD2Hex消息摘要
HmacMD2Hex-1：      bf5fa06c2c4855825a23ee08206c892f
HmacMD2Hex-2：      bf5fa06c2c4855825a23ee08206c892f
原文：              HmacMD4Hex消息摘要
HmacMD4Hex-1：      c6ac3ec24690011bdfad9ce2e7aed1e7
HmacMD4Hex-2：      c6ac3ec24690011bdfad9ce2e7aed1e7
原文：              HmacSHA224Hex消息摘要
HmacSHA224Hex-1：   54786b4722f72a2599ebb1bba3732bca9f0353392e51729bfca91c78
HmacSHA224Hex-2：   54786b4722f72a2599ebb1bba3732bca9f0353392e51729bfca91c78
```

我们可以清楚地看到，经HmacMD2Hex 和HmacMD4Hex处理后得到的字符串是一个32位的十六进制字符串，换算成二进制正好是128位，也就是MD系列算法做消息摘要处理后得到的摘要值长度。同理，HmacSHA224Hex处理得到的消息摘要值与SHA-224算法做消息摘要处理后得到的摘要值长度相同。

3. 两种实现方式的差异

两种实现方式以Sun提供的实现为基础，在算法支持上提供了更好的扩展。

❑ Sun

提供了基本的算法支持，如HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和Hmac512五种算法支持。

❑ Bouncy Castle

Bouncy Castle在Sun的基础上添加了对HmacMD2、HmacMD4和HmacSHA224三种算法的支持，同时支持十六进制编码。

综上所述，根据需求恰当使用上述实现是很有必要的。如果需要HmacMD2、HmacMD4或HmacSHA224算法支持，可以使用Bouncy Castle做相关实现。如果还需要对摘要结果做十六进

制编码，则使用Bouncy Castle更为恰当。

6.5 其他消息摘要算法

除了MD、SHA和MAC这三大主流消息摘要算法外，还有许多我们不了解的消息摘要算法，包括RipeMD系列（包含RipeMD128、RipeMD160、RipeMD256和RipeMD320）、Tiger、Whirlpool和GOST3411算法。

RipeMD系列算法与MAC系列算法相结合，又产生了HmacRipeMD128和HmacRipeMD160两种算法。

作为了解，我们在这里做简要介绍。也许有一天，我们会需要这些非主流消息摘要算法。

6.5.1 简述

作者未能获得上述4种算法的相关定义，在此简要介绍。

❑ RipeMD

RipeMD（RACE Integrity Primitives Evaluation Message Digest），是由Hans Dobbertin等3人在对MD4，MD5缺陷分析的基础上，于1996年提出。目前，RipeMD算法共有4个标准，主要是对摘要值长度的区分，类似于SHA系列算法，包含RipeMD128、RipeMD160、RipeMD256和RipeMD320共4种算法。HmacRipeMD128和HmacRipeMD160算法是RipeMD与MAC算法融合的产物。

❑ Tiger

Tiger由Ross于1995年提出。Tiger号称是最快的Hash算法，专门为64位机器做了优化，其消息摘要长度为192位。

❑ Whirlpool

Whirlpool已被列入ISO标准，由于它使用了与AES加密标准相同的转化技术，极大地提高了安全性，被称为最安全的摘要算法。Whirlpool在历史上共有3个版本，目前最新的版本是2003年颁布的，通常将其称为Whirlpool 3.0，其消息摘要长度为512位。

❑ GOST3411

对于GOST3411算法，作者未能获得更多的相关信息，只能得知该算法得到的摘要信息长度为256位。

6.5.2 实现

Java 6内并没有提供上述算法的实现，这里仅仅是Bouncy Castle的专场。为避免内容冗长，作者在这里以RipeMD系列算法和HmacRipeMD系列算法为例，做简要介绍。

RipeMD系列算法和HmacRipeMD系列算法如表6-4所示。

表6-4 RipeMD系列算法和HmacRipeMD系列算法

算法	摘要长度	备注
RipeMD128	128	Bouncy Castle实现
RipeMD160	160	
RipeMD256	256	
RipeMD320	320	
HmacRipeMD128	128	
HmacRipeMD160	160	

1. RipeMD系列算法

RipeMD算法的实现具有代表性，Tiger、Whirlpool和GOST3411算法实现与其别无二致。

目前，Bouncy Castle提供了RipeMD128、RipeMD160、RipeMD256和RipeMD320共4种算法实现，代码实现如代码清单6-17所示。

代码清单6-17 RipeMD消息摘要算法实现

```
import java.security.MessageDigest;
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.encoders.Hex;
/**
 * RipeMD系列消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class RipeMDCoder {
    /**
     * RipeMD128消息摘要
     * @param data 待做消息摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeRipeMD128(byte[] data) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("RipeMD128");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * RipeMD128Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
```

```
 */
public static String encodeRipeMD128Hex(byte[] data) throws Exception {
        // 执行消息摘要
        byte[] b = encodeRipeMD128(data);
        // 做十六进制编码处理
        return new String(Hex.encode(b));
}
/**
 * RipeMD160消息摘要
 * @param data 待做消息摘要处理的数据
 * @return byte[] 消息摘要
 * @throws Exception
 */
public static byte[] encodeRipeMD160(byte[] data) throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 初始化MessageDigest
        MessageDigest md = MessageDigest.getInstance("RipeMD160");
        // 执行消息摘要
        return md.digest(data);
}
/**
 * RipeMD160Hex消息摘要
 * @param data 待做消息摘要处理的数据
 * @return String 消息摘要
 * @throws Exception
 */
public static String encodeRipeMD160Hex(byte[] data) throws Exception {
        // 执行消息摘要
        byte[] b = encodeRipeMD160(data);
        // 做十六进制编码处理
        return new String(Hex.encode(b));
}
/**
 * RipeMD256消息摘要
 * @param data 待做消息摘要处理的数据
 * @return byte[] 消息摘要
 * @throws Exception
 */
public static byte[] encodeRipeMD256(byte[] data) throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 初始化MessageDigest
        MessageDigest md = MessageDigest.getInstance("RipeMD256");
        // 执行消息摘要
        return md.digest(data);
}
/**
 * RipeMD256Hex消息摘要
```

```
     * @param data 待做消息摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeRipeMD256Hex(byte[] data) throws Exception {
            // 执行消息摘要
            byte[] b = encodeRipeMD256(data);
            // 做十六进制编码处理
            return new String(Hex.encode(b));
    }
    /**
     * RipeMD320消息摘要
     * @param data 待做消息摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeRipeMD320(byte[] data) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("RipeMD320");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * RipeMD320Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeRipeMD320Hex(byte[] data) throws Exception {
            // 执行消息摘要
            byte[] b = encodeRipeMD320(data);
            // 做十六进制编码处理
            return new String(Hex.encode(b));
    }
}
```

上述代码对应的测试用例如代码清单6-18所示。

代码清单6-18 RipeMD消息摘要算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * RipeMD校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
```

```
 */
public class RipeMDCoderTest {
    /**
     * 测试RipeMD128
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD128() throws Exception {
            String str = "RipeMD128消息摘要";
            // 获得摘要信息
            byte[] data1 = RipeMDCoder.encodeRipeMD128(str.getBytes());
            byte[] data2 = RipeMDCoder.encodeRipeMD128(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试RipeMD128Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD128Hex() throws Exception {
            String str = "RipeMD128Hex消息摘要";
            // 获得摘要信息
            String data1 = RipeMDCoder.encodeRipeMD128Hex(str.getBytes());
            String data2 = RipeMDCoder.encodeRipeMD128Hex(str.getBytes());
            System.err.println("原文：\t" + str);
            System.err.println("RipeMD128Hex-1：\t" + data1);
            System.err.println("RipeMD128Hex-2：\t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
    /**
     * 测试RipeMD160
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD160() throws Exception {
            String str = "RipeMD160消息摘要";
            // 获得摘要信息
            byte[] data1 = RipeMDCoder.encodeRipeMD160(str.getBytes());
            byte[] data2 = RipeMDCoder.encodeRipeMD160(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试RipeMD160Hex
     * @throws Exception
     */
    @Test
```

```
    public final void testEncodeRipeMD160Hex() throws Exception {
            String str = "RipeMD160Hex消息摘要";
            // 获得摘要信息
            String data1 = RipeMDCoder.encodeRipeMD160Hex(str.getBytes());
            String data2 = RipeMDCoder.encodeRipeMD160Hex(str.getBytes());
            System.err.println("原文：\t" + str);
            System.err.println("RipeMD160Hex-1：\t" + data1);
            System.err.println("RipeMD160Hex-2：\t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
    /**
     * 测试RipeMD256
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD256() throws Exception {
            String str = "RipeMD256消息摘要";
            // 获得摘要信息
            byte[] data1 = RipeMDCoder.encodeRipeMD256(str.getBytes());
            byte[] data2 = RipeMDCoder.encodeRipeMD256(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试RipeMD256Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD256Hex() throws Exception {
            String str = "RipeMD256Hex消息摘要";
            // 获得摘要信息
            String data1 = RipeMDCoder.encodeRipeMD256Hex(str.getBytes());
            String data2 = RipeMDCoder.encodeRipeMD256Hex(str.getBytes());
            System.err.println("原文：\t" + str);
            System.err.println("RipeMD256Hex-1：\t" + data1);
            System.err.println("RipeMD256Hex-2：\t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
    /**
     * 测试RipeMD320
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD320() throws Exception {
            String str = "RipeMD320消息摘要";
            // 获得摘要信息
            byte[] data1 = RipeMDCoder.encodeRipeMD320(str.getBytes());
```

```
            byte[] data2 = RipeMDCoder.encodeRipeMD320(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试RipeMD320Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD320Hex() throws Exception {
            String str = "RipeMD320Hex消息摘要";
            // 获得摘要信息
            String data1 = RipeMDCoder.encodeRipeMD320Hex(str.getBytes());
            String data2 = RipeMDCoder.encodeRipeMD320Hex(str.getBytes());
            System.err.println("原文：\t" + str);
            System.err.println("RipeMD320Hex-1：\t" + data1);
            System.err.println("RipeMD320Hex-2：\t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
}
```

我们一起来观察控制台输出的信息。

以下是RipeMD128处理后的十六进制摘要结果：

```
原文：              RipeMD128Hex消息摘要
RipeMD128Hex-1：    3a7d75b69093be33f7f5442b97e09d69
RipeMD128Hex-2：    3a7d75b69093be33f7f5442b97e09d69
```

我们得到了一个32位的十六进制字符串，换算成二进制正好是128位。

对于RipeMD160、RipeMD256和RipeMD320算法的摘要值的十六进制自然是40位、64位和80位。

以下是RipeMD160算法的控制台输出信息：

```
原文：            RipeMD160Hex消息摘要
RipeMD160Hex-1： 5baec3e74e72e719fc702bc0057d6bb80c037ee3
RipeMD160Hex-2： 5baec3e74e72e719fc702bc0057d6bb80c037ee3
```

以下是RipeMD256算法的控制台输出信息：

```
原文：            RipeMD256Hex消息摘要
RipeMD256Hex-1：466776542909d4e57215deaaad7f7a50351c6250a0fd3bdddf20cfa28a5f6678
RipeMD256Hex-2：466776542909d4e57215deaaad7f7a50351c6250a0fd3bdddf20cfa28a5f6678
```

以下是RipeMD320算法的控制台输出信息：

```
原文：          RipeMD320Hex消息摘要
RipeMD320Hex-1：e8e9c116bf1b8762199c1b411f6c4d0bd3ac34fc448736e9ccbb91144f8ec279e55da92c7d3c847c
RipeMD320Hex-2：e8e9c116bf1b8762199c1b411f6c4d0bd3ac34fc448736e9ccbb91144f8ec279e55da92c7d3c847c
```

对于Tiger、Whirlpool和GOST3411算法的实现与测试用例，与RipeMD无任何差别，只需

替换算法名称即可，作者在这里就不再复述了。

2. HmacRipeMD系列算法

HmacRipeMD算法是RipeMD与MAC两种算法的融合，实现起来较为简单。目前，Bouncy Castle提供了HmacRipeMD128和HmacRipeMD160两种算法实现，代码实现如代码清单6-19所示。

代码清单6-19 HmacRipeMD系列算法实现

```
import java.security.Security;
import javax.crypto.KeyGenerator;
import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.encoders.Hex;
/**
 * HmacRipeMD系列消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class HmacRipeMDCoder {
    /**
     * 初始化HmacRipeMD128密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] initHmacRipeMD128Key() throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化KeyGenerator
            KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacRipeMD128");
            // 产生秘密密钥
            SecretKey secretKey = keyGenerator.generateKey();
            // 获得密钥
            return secretKey.getEncoded();
    }
    /**
     * HmacRipeMD128消息摘要
     * @param data 待做消息摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacRipeMD128(byte[] data, byte[] key)
                throws Exception {
            // 加入BouncyCastleProvider支持
```

```
        Security.addProvider(new BouncyCastleProvider());
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacRipeMD128");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
}
/**
 * HmacRipeMD128Hex消息摘要
 * @param data 待做消息摘要处理的数据
 * @param key 密钥
 * @return String 消息摘要
 * @throws Exception
 */
public static String encodeHmacRipeMD128Hex(byte[] data, byte[] key)
              throws Exception {
        // 执行消息摘要
        byte[] b = encodeHmacRipeMD128(data, key);
        // 做十六进制转换
        return new String(Hex.encode(b));
}
/**
 * 初始化HmacRipeMD160密钥
 * @return byte[] 密钥
 * @throws Exception
 */
public static byte[] initHmacRipeMD160Key() throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 初始化KeyGenerator
        KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacRipeMD160");
        // 产生秘密密钥
        SecretKey secretKey = keyGenerator.generateKey();
        // 获得密钥
        return secretKey.getEncoded();
}
/**
 * HmacRipeMD160消息摘要
 * @param data 待做消息摘要处理的数据
 * @param key 密钥
 * @return byte[] 消息摘要
 * @throws Exception
 */
public static byte[] encodeHmacRipeMD160(byte[] data, byte[] key)
              throws Exception {
```

```
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 还原密钥
            SecretKey secretKey = new SecretKeySpec(key, "HmacRipeMD160");
            // 实例化Mac
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            // 初始化Mac
            mac.init(secretKey);
            // 执行消息摘要
            return mac.doFinal(data);
    }
    /**
     * HmacRipeMD160Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @param key 密钥
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeHmacRipeMD160Hex(byte[] data, byte[] key)
                throws Exception {
            // 执行消息摘要
            byte[] b = encodeHmacRipeMD160(data, key);
            // 做十六进制转换
            return new String(Hex.encode(b));
    }
```

对上述代码做相关测试，如代码清单6-20所示。

代码清单6-20 HmacRipeMD系列算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * HmacRipeMD校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class HmacRipeMDCoderTest {
    /**
     * 测试HmacRipeMD128
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacRipeMD128() throws Exception {
            String str = "HmacRipeMD128消息摘要";
            // 初始化密钥
            byte[] key = HmacRipeMDCoder.initHmacRipeMD128Key();
            // 获得摘要信息
```

```
        byte[] data1 = HmacRipeMDCoder.encodeHmacRipeMD128(str.getBytes(), key);
        byte[] data2 = HmacRipeMDCoder.encodeHmacRipeMD128(str.getBytes(), key);
        // 校验
        assertArrayEquals(data1, data2);
}
/**
 * 测试HmacRipeMD128Hex
 * @throws Exception
 */
@Test
public final void testEncodeHmacRipeMD128Hex() throws Exception {
        String str = "HmacRipeMD128Hex消息摘要";
        // 初始化密钥
        byte[] key = HmacRipeMDCoder.initHmacRipeMD128Key();
        // 获得摘要信息
        String data1 = HmacRipeMDCoder.encodeHmacRipeMD128Hex(str.getBytes(), key);
        String data2 = HmacRipeMDCoder.encodeHmacRipeMD128Hex(str.getBytes(), key);
        System.err.println("原文：\t" + str);
        System.err.println("HmacRipeMD128Hex-1：\t" + data1);
        System.err.println("HmacRipeMD128Hex-2：\t" + data2);
        // 校验
        assertEquals(data1, data2);
}
/**
 * 测试HmacRipeMD160
 * @throws Exception
 */
@Test
public final void testEncodeHmacRipeMD160() throws Exception {
        String str = "HmacRipeMD160消息摘要";
        // 初始化密钥
        byte[] key = HmacRipeMDCoder.initHmacRipeMD160Key();
        // 获得摘要信息
        byte[] data1 = HmacRipeMDCoder.encodeHmacRipeMD160(str.getBytes(), key);
        byte[] data2 = HmacRipeMDCoder.encodeHmacRipeMD160(str.getBytes(), key);
        // 校验
        assertArrayEquals(data1, data2);
}
/**
 * 测试HmacRipeMD160Hex
 * @throws Exception
 */
@Test
public final void testEncodeHmacMD4Hex() throws Exception {
        String str = "HmacRipeMD160Hex消息摘要";
        // 初始化密钥
        byte[] key = HmacRipeMDCoder.initHmacRipeMD160Key();
        // 获得摘要信息
```

```
            String data1 = HmacRipeMDCoder.encodeHmacRipeMD160Hex(str.getBytes(), key);
            String data2 = HmacRipeMDCoder.encodeHmacRipeMD160Hex(str.getBytes(), key);
            System.err.println("原文：\t" + str);
            System.err.println("HmacRipeMD160Hex-1：\t" + data1);
            System.err.println("HmacRipeMD160Hex-2：\t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
}
```

以下是控制台中对应的输出信息：

```
原文：                  HmacRipeMD128Hex消息摘要
HmacRipeMD128Hex-1：   f08ccebbafe66f852ade9c73dcfdf14f
HmacRipeMD128Hex-2：   f08ccebbafe66f852ade9c73dcfdf14f
原文：                  HmacRipeMD160Hex消息摘要
HmacRipeMD160Hex-1：   0843ab8d7f3c3d0aca920fa7b8429268bacf5615
HmacRipeMD160Hex-2：   0843ab8d7f3c3d0aca920fa7b8429268bacf5615
```

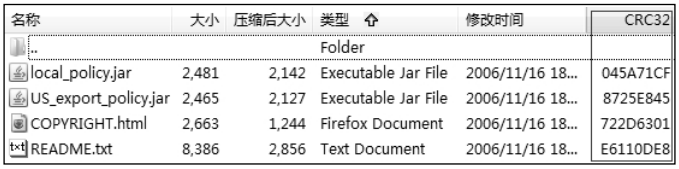
消息摘要长度与相应的摘要算法的摘要长度相同：HmacRipeMD128与RipeMD128相对应，消息摘要都是32个字符的十六进制串；HmacRipeMD160与RipeMD160相对应，消息摘要值都是40个字符的十六进制串。

目前，Bouncy Castle不仅提供了HmacRipeMD系列算法实现，同时还提供了HMacTiger算法实现。实现方式与代码清单6-19相类似。

作者在整理Bouncy Castle加密组件时，才发现原来还有这么多不为人知的消息摘要算法。可见，各种消息摘要算法正以其独到之处快速地发展着，经过一番优胜劣汰，逐步成为标准。

6.6 循环冗余校验算法——CRC算法

大家对于“奇偶校验码”和“循环冗余校验码”这样的名词应该不会感到陌生。即便不是计算机专业出身，也一定使用过压缩软件WinRAR，在压缩文件列表中能看到一个标有“CRC32”字样的内容，如图6-8所示。



名称	大小	压缩后大小	类型	修改时间	CRC32
..			Folder		
local_policy.jar	2,481	2,142	Executable Jar File	2006/11/16 18...	045A71CF
US_export_policy.jar	2,465	2,127	Executable Jar File	2006/11/16 18...	8725E845
COPYRIGHT.html	2,663	1,244	Firefox Document	2006/11/16 18...	722D6301
README.txt	8,386	2,856	Text Document	2006/11/16 18...	E6110DE8

图6-8 压缩包中的CRC32算法

“奇偶校验码”、“循环冗余校验码”和“CRC32”都是同一套东西，它们和CRC有着紧密的联系。

CRC算法并不属于加密算法范畴，但与本章内容较为接近，故在这里为读者朋友做简要介绍。

6.6.1 简述

CRC（Cyclic Redundancy Check，循环冗余校验）是可以根据数据产生简短固定位数的一种散列函数，主要用来检测或校验数据传输/保存后出现的错误。生成的散列值在传输或储存之前计算出来并且附加到数据后面。在使用数据之前，对数据的完整性做校验。一般来说，循环冗余校验的值都是32位的二进制数，以8位十六进制字符串形式表示。它是一类重要的线性分组码，编码和解码方法简单，检错和纠错能力强，在通信领域广泛地用于实现差错控制。

由上述内容分析，消息摘要算法与CRC算法同属散列函数，并且CRC算法很可能就是消息摘要算法的前身。

CRC算法历经了多个版本的演进，从最初的CRC-1算法至最终的CRC-160算法，为消息摘要算法奠定了基础。我们简单回顾一下CRC算法的发展历程：

- CRC-1：主要用于硬件，就是我们常说的奇偶校验码。
- CRC-32-IEEE 802.3：主要用于通信领域实现差错控制，也就是我们今天常说的CRC-32，IEEE 802.3只是一种标准。
- CRC-32-Adler：CRC-32的一个变种，可以称为“ Adler-32 ”，与CRC-32一样可靠，但是速度更快。
- CRC-128：演变为今天的MD算法。MD算法消息摘要值为128位二进制数。
- CRC-160：演变为今天的SHA算法。SHA-1算法消息摘要值为160位二进制数。

至今，CRC-32算法仍是各种压缩算法中最为常用的数据完整性校验算法。而它的变种Adler-32普遍用于zlib压缩算法中的数据完整性校验。

6.6.2 模型分析

我们以甲乙双方传递压缩数据模型为例，演示如何使用CRC算法。

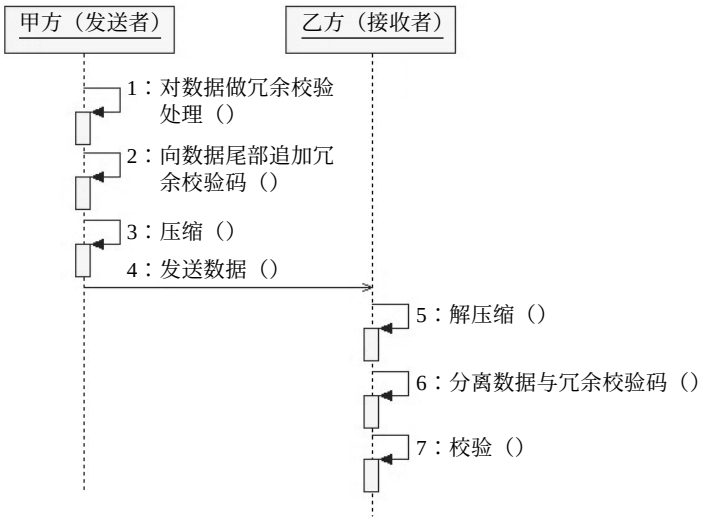


图6-9 压缩/解压缩数据传递模型

6.6.3 实现

在Java 6中，CRC-32算法是由CRC32类来实现的。

```
// 可用于计算数据流的CRC-32的类。
public class CRC32
extends Object
implements Checksum
```

CRC32类使用起来极为简单，通过其无参构造方法完成实例化对象后，需要先执行以下update()方法：

```
// 使用指定的字节数组更新校验和。
public void update(byte[] b)
// 使用指定的字节数组更新 CRC-32。
public void update(byte[] b, int off, int len)
// 使用指定字节更新 CRC-32。
public void update(int b)
```

update()方法可多次执行，此后执行getValue()方法：

```
// 返回 CRC-32 值。
public long getValue()
```

执行完上述方法后，我们可以获得一个长整型的冗余校验码，可通过Long类的toHexString()方法将其转换为十六进制字符串形式。

如需重置，需执行如下方法：

```
// 将 CRC-32 重置为初始值。
public void reset()
```

除此之外，还有CheckedInputStream和CheckedOutputStream两个类，可用于输入输出流的冗余校验处理。对于Adler-32算法，请使用Adler32类。Adler32类同样实现了Checksum接口，其方法与CRC32相同。

CRC32算法实现较为简单，如代码清单6-21所示。

代码清单6-21 CRC32算法实现

```
import java.util.zip.CRC32;
import org.junit.Test;
/**
 * 测试CRC-32
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class CRCTest {
    /**
     * 测试CRC-32
     * @throws Exception
```

```
     */
    @Test
    public void testCRC32() throws Exception {
            String str = "测试CRC-32";
            CRC32 crc32 = new CRC32();
            crc32.update(str.getBytes());
            String hex = Long.toHexString(crc32.getValue());
            System.err.println("原文：\t" + str);
            System.err.println("CRC-32：\t" + hex);
    }
}
```

执行上述代码后，在控制台中得到如下内容：

```
原文：      测试CRC-32
CRC-32：    eda03da1
```

原文经CRC-32算法处理后得到了一个8位十六进制字符串。

对上述代码稍作调整，就可用于文件校验。

6.7 实例：文件校验

秉承程序员的执著，发现问题就要解决问题！既然我们已经看到了MySQL下载页面上的MD5信息，就要对其文件做MD5校验！通过下载页面下载MySQL Essential 5.1.38的Windows安装版（mysql-essential-5.1.38-win32.msi），并将其放置在D盘根目录下。接下来，我们要对这个文件做一次MD5校验。如果校验结果与下载页面上给出的MD5信息不一致，则说明文件在下载的过程中被篡改了！

这个校验并不复杂，我们分别通过testByMessageDigest()和testByDigestUtils()方法做校验处理，如代码清单6-22所示。

代码清单6-22 校验下载文件一致性

```
import static org.junit.Assert.*;
import java.io.File;
import java.io.FileInputStream;
import java.security.DigestInputStream;
import java.security.MessageDigest;
import org.apache.commons.codec.binary.Hex;
import org.apache.commons.codec.digest.DigestUtils;
import org.junit.Test;
/**
 * 消息摘要编码测试<br>
 * 用于校验文件的MD5值
 * <pre>
 * 文件为 mysql-essential-5.1.38-win32.msi
 * 存放于D盘根目录
```

```
 * MD5值为5a077abefee447cbb271e2aa7f6d5a47
 * </pre>
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MD5Test {
    /**
     * 验证文件的MD5值
     * @throws Exception
     */
    @Test
    public void testByMessageDigest() throws Exception {
            // 文件路径
            String path = "D:\\mysql-essential-5.1.38-win32.msi";
            // 构建文件输入流
            FileInputStream fis = new FileInputStream(new File(path));
            // 初始化MessageDigest，并指定MD5算法
            DigestInputStream dis = new DigestInputStream(fis, MessageDigest.getInstance("MD5"));
            // 流缓冲大小
            int buf = 1024;
            // 缓冲字节数组
            byte[] buffer = new byte[buf];
            // 当读到值大于-1就继续读
            int read = dis.read(buffer, 0, buf);
            while (read > -1) {
                read = dis.read(buffer, 0, buf);
            }
            // 关闭流
            dis.close();
            // 获得MessageDigest
            MessageDigest md = dis.getMessageDigest();
            // 摘要处理
            byte[] b = md.digest();
            // 十六进制转换
            String md5hex = Hex.encodeHexString(b);
            // 验证
            assertEquals(md5hex, "5a077abefee447cbb271e2aa7f6d5a47");
    }
    /**
     * 验证文件的MD5值
     * @throws Exception
     */
    @Test
    public void testByDigestUtils() throws Exception {
            // 文件路径
```

```
            String path = "D:\\mysql-essential-5.1.38-win32.msi";
            // 构建文件输入流
            FileInputStream fis = new FileInputStream(new File(path));
            // 使用DigestUtils做MD5Hex处理
            String md5hex = DigestUtils.md5Hex(fis);
            // 关闭流
            fis.close();
            // 验证
            assertEquals(md5hex, "5a077abefee447cbb271e2aa7f6d5a47");
    }
}
```

既然两种方法都是MD5文件校验，那么会有什么差异呢？

从测试结果来说，一定都是通过，那么在效率上呢？如图6-10所示。

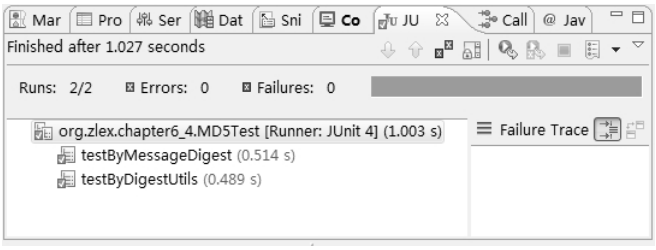


图6-10 MD5文件校验处理测试结果

作者做了反复测试，其测试结果与上图基本相符，两种方法实现效率较为接近。

如果只是对文件做MD5校验，用Commons Codec来实现再合适不过了。对文件做MD5处理/校验，已经是一件很平常的事情了。相信上述简要的代码实现，对读者朋友一定会很有益处。

6.8 小结

对于数据完整性的验证，可能是每一个计算机用户所关心的事情。这好比是自己在菜市场上买了菜要验分量。但如何来验证数据是否完整，而且是快速准确地验证数据的完整性又是一个麻烦的问题，这需要一种算法。

消息摘要算法就是这样一种专门用于验证数据完整性的算法。它源于CRC冗余校验算法，派生出MD和SHA两大系列消息算法，并在此基础上衍生出MAC算法。消息摘要算法是数字签名算法的基础。

几乎每一种消息摘要算法都有大概三种实现方式：Sun、Bouncy Castle和Commons Codec。Sun提供了最基本的算法实现；Bouncy Castle在Sun的基础上做了扩展，实现了Sun未能提供的算法；Commons Codec在Sun的基础上，对其方法做了封装，使其易用性提高，方便使用。

MD算法家族拥有MD2、MD4和MD5三种常用算法。在Java 6中，通过MessageDigest类可提供MD2和MD5两种算法支持；通过Bouncy Castle扩展，可提供MD4算法支持；Commons Codec则直接强化了MD5算法使用的便利性。如果考虑MD4算法的支持，则使用Bouncy

Castle；如果考虑方法的易用性，则使用Commons Codec更为恰当。

虽然MD5算法的破解使其安全性大为降低，但在用户注册/登录模块中仍然是架构师首选的方案。各大软件厂商在其软件下载页面上仍然使用MD5算法作为数据完整性验证的首选方法。MD5算法常作为安全性要求不高的环境中的常用算法。MD4和MD5算法为后续消息摘要算法（如SHA）的设计提供了参考。

SHA算法家族枝繁叶茂，拥有SHA-1、SHA-224、SHA-256、SHA-384和SHA-512五种常用算法。其中，后四种算法是在原有SHA-1基础上扩展了消息摘要长度，通常也称为SHA-2算法。在Java 6中，通过MessageDigest类可提供SHA-1、SHA-256、SHA-384和SHA-512四种算法支持；通过Bouncy Castle扩展，可提供SHA-224算法支持；Commons Codec则强化了SHA-1、SHA-256、SHA-384和SHA-512四种算法相应方法的易用性。如果不要求使用SHA-224算法，使用Commons Codec更方便。

SHA算法较之MD算法更为安全，常常出现在一些安全系数要求较高的环境中。一般的用户注册/登录模块，各大软件厂商用于校验数据完整性的页面中都常常用到SHA算法。这些领域既是MD5算法出没的地方，也是SHA算法盘踞之处。除此之外，MD5和SHA算法还常常作为数字证书的签名算法，而SHA算法则更为常见一些。

MAC是一种基于密钥的散列函数算法，它吸收了MD算法和SHA算法的精髓，并将其发扬光大，包含HmacMD2、HmacMD4和HmacMD5三种MD系列算法，HmacSHA1、HmacSHA224、HmacSHA256、HmacSHA384和HmacSHA512五种SHA系列算法。在Java 6中，通过Mac类可以提供HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和HmacSHA512五种算法支持；通过Bouncy Castle扩展，可提供HmacMD2、HmacMD4和HmacSHA224三种算法支持。如果你想让你的系统更强劲，使用Bouncy Castle就是最好的选择。

虽然已经有了主流的消息摘要算法实现，但仍可能不能满足我们的需要。非主流消息摘要算法RipeMD系列（括含RipeMD128、RipeMD160、RipeMD256和RipeMD320）、Tiger、Whirlpool和GOST3411算法，以及HmacRipeMD系列（包括HmacRipeMD128和HmacRipeMD160）算法，可通过Bouncy Castle完成实现。

MD、SHA和MAC都是加密算法领域的消息摘要算法，与之功能相近的CRC-32算法则是最为古老的数据完整性验证算法。目前，CRC-32算法仍广泛用于通信领域，实现差错控制，其变种Adler-32算法则广泛适用于zlib压缩算法中。