# 加密与算法

1.

# 一 编码算法

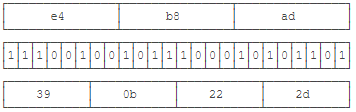
## 1.1 什么是编码？

对字符的另一种映射。比如二进制对应不同的字符

## 1.2 什么是Base64编码？

Base64编码可以把任意长度的二进制数据变为纯文本，且只包含A~Z、a~z、0~9、+、/、=这些字符。它的原理是把3字节的二进制数据按6bit一组，用4个int整数表示，然后查表，把int整数用索引对应到字符，得到编码后的字符串。

举个例子：3个byte数据分别是e4、b8、ad，按6bit分组得到39、0b、22和2d：



因为6位整数的范围总是0~63，所以，能用64个字符表示：字符A~Z对应索引0~25，字符a~z对应索引26~51，字符0~9对应索引52~61，最后两个索引62、63分别用字符+和/表示。

在Java中，二进制数据就是byte[]数组。Java标准库提供了Base64来对byte[]数组进行编解码：

*/\*\*  
 \* Base64编码可以把任意长度的二进制数据变为纯文本，且只包含A~Z、a~z、0~9、+、/、=这些字符。  
 \*  
 \* 原理:  
 \* 把3字节的二进制数据按6bit一组，用4个int整数表示，然后查表，把int整数用索引对应到字符，得到编码后的字符串。  
 \*  
 \* 符中的UTF-8编码是0xe4b8ad，因此，它的URL编码是%E4%B8%AD。URL编码总是大写。  
 \* e4=228=1110 0100  
 \*  
 \** ***@author*** *lipu  
 \** ***@since*** *2020-08-05 14:34:04  
 \*/*public class Demo01Base64 {  
  
 public static void main(String[] args) {  
 byte[] input = new byte[] { (byte) 0xe4, (byte) 0xb8, (byte) 0xad };  
 String b64encoded = Base64.*getEncoder*().encodeToString(input);  
 System.*out*.println(b64encoded);  
 byte[] bytes = Base64.*getDecoder*().decode(b64encoded);  
 System.*out*.println(new String(bytes));  
 }  
}

5Lit

中

如果输入的byte[]数组长度不是3的整数倍怎么办？

这种情况下，需要对输入的末尾补一个或两个0x00，编码后，在结尾加一个=表示补充了1个0x00，加两个=表示补充了2个0x00，解码的时候，去掉末尾补充的一个或两个0x00即可。

实际上，因为编码后的长度加上=总是4的倍数，所以即使不加=也可以计算出原始输入的byte[]。Base64编码的时候可以用withoutPadding()去掉=，解码出来的结果是一样的：

## 1.3 为什么会有Base64？

Base64编码的目的是**把二进制数据变成文本格式**，这样在很多文本中就可以处理二进制数据。例如，电子邮件协议就是文本协议，如果要在电子邮件中添加一个二进制文件，就可以用Base64编码，然后以文本的形式传送。

**缺点：**传输效率会降低，因为它把原始数据的长度增加了1/3。

和URL编码一样，Base64编码是一种**编码算法，不是加密算法**。

如果把Base64的64个字符编码表换成32个、48个或者58个，就可以使用Base32编码，Base48编码和Base58编码。字符越少，编码的效率就会越低。

# 二 哈希算法

## 2.1 什么是哈希算法？

哈希算法（Hash）又称摘要算法（Digest），它的作用是：对任意一组输入数据进行计算，得到一个固定长度的输出摘要。

**特点：**

相同的输入一定得到相同的输出；

不同的输入大概率得到不同的输出。

**目的：** 验证原始数据是否被篡改。

Java字符串的hashCode()就是一个哈希算法，它的输入是任意字符串，输出是固定的4字节int整数：

"hello".hashCode(); // 0x5e918d2

"hello, java".hashCode(); // 0x7a9d88e8

"hello, bob".hashCode(); // 0xa0dbae2f

两个相同的字符串永远会计算出相同的hashCode，否则基于hashCode定位的HashMap就无法正常工作。这也是为什么当我们自定义一个class时，覆写equals()方法时我们必须正确覆写hashCode()方法。

## 2.2 什么哈希碰撞？

两个不同的输入得到了相同的输出：

"AaAaAa".hashCode(); // 0x7460e8c0

"BBAaBB".hashCode(); // 0x7460e8c0

### 碰撞能不能避免？

答案是不能。

碰撞是一定会出现的，因为输出的字节长度是固定的，String的hashCode()输出是4字节整数，最多只有4294967296种输出，但输入的数据长度是不固定的，有无数种输入。所以，哈希算法是把一个无限的输入集合映射到一个有限的输出集合，必然会产生碰撞。

碰撞不可怕，我们担心的不是碰撞，而是碰撞的概率，因为碰撞概率的高低关系到哈希算法的安全性。一个安全的哈希算法必须满足：

碰撞概率低；

不能猜测输出。

不能猜测输出是指，输入的任意一个bit的变化会造成输出完全不同，这样就很难从输出反推输入（只能依靠暴力穷举）。假设一种哈希算法有如下规律：

hashA("java001") = "123456"

hashA("java002") = "123457"

hashA("java003") = "123458"

那么很容易从输出123459反推输入，这种哈希算法就不安全。安全的哈希算法从输出是看不出任何规律的：

hashB("java001") = "123456"

hashB("java002") = "580271"

hashB("java003") = ???

常用的哈希算法有：

| **算法** | **输出长度（位）** | **输出长度（字节）** |
| --- | --- | --- |
| MD5 | 128 bits | 16 bytes |
| SHA-1 | 160 bits | 20 bytes |
| RipeMD-160 | 160 bits | 20 bytes |
| SHA-256 | 256 bits | 32 bytes |
| SHA-512 | 512 bits | 64 bytes |

根据碰撞概率，哈希算法的输出长度越长，就越难产生碰撞，也就越安全。

Java标准库提供了常用的哈希算法，并且有一套统一的接口。我们以MD5算法为例，看看如何对输入计算哈希：

public class Demo01Hash {  
  
 public static void main(String[] args) throws Exception {  
 // 创建一个MessageDigest实例:MD5\SHA-1\SHA-256\SHA-512\  
 MessageDigest md = MessageDigest.*getInstance*("MD5");  
// MessageDigest md = MessageDigest.getInstance("SHA-1");  
// MessageDigest md = MessageDigest.getInstance("SHA-256");  
// MessageDigest md = MessageDigest.getInstance("SHA-512");  
 // 反复调用update(byte[])输入数据。当输入结束后，调用digest()方法获得byte[]数组表示的摘要，最后，把它转换为十六进制的字符串。  
 md.update("Hello".getBytes("UTF-8"));  
 md.update("World".getBytes("UTF-8"));  
 byte[] result = md.digest(); // 16 bytes: 68e109f0f40ca72a15e05cc22786f8e6  
 System.*out*.println(new BigInteger(1, result).toString(16));  
 System.*out*.println(new BigInteger(1, result).toString(10));  
 System.*out*.println(new BigInteger(1, result).toString(2));  
 }  
}

### 小结

哈希算法可用于验证数据完整性，具有防篡改检测的功能；

常用的哈希算法有MD5、SHA-1等；

用哈希存储口令时要考虑彩虹表攻击。

# 三 Hmac算法

因此，HmacMD5可以看作带有一个安全的key的MD5。使用HmacMD5而不是用MD5加salt，有如下好处：

HmacMD5使用的key长度是64字节，更安全；

Hmac是标准算法，同样适用于SHA-1等其他哈希算法；

Hmac输出和原有的哈希算法长度一致。

可见，Hmac本质上就是把key混入摘要的算法。验证此哈希时，除了原始的输入数据，还要提供key。

public class Demo01Hmac {  
  
 public static void main(String[] args) throws Exception{  
  
 // KeyGenerator:HmacMD5\HmacSHA-1\HmacSHA-256\HmacSHA-512  
 KeyGenerator keyGen = KeyGenerator.*getInstance*("HmacMD5");  
 //生成秘钥  
 SecretKey key = keyGen.generateKey();  
 // 打印随机生成的key:  
 byte[] skey = key.getEncoded();  
 System.*out*.println(new BigInteger(1, skey).toString(16));  
 //通过名称HmacMD5获取Mac实例；  
 Mac mac = Mac.*getInstance*("HmacMD5");  
 //用SecretKey初始化Mac实例；  
 mac.init(key);  
 //对Mac实例反复调用update(byte[])输入数据；  
 mac.update("HelloWorld".getBytes("UTF-8"));  
 //调用Mac实例的doFinal()获取最终的哈希值。  
 byte[] result = mac.doFinal();  
 System.*out*.println(new BigInteger(1, result).toString(16));  
 }  
}

# 四 对称加密

对称加密算法就是传统的用一个密码进行加密和解密。例如，我们常用的WinZIP和WinRAR对压缩包的加密和解密，就是使用对称加密算法：

从程序的角度看，所谓加密，就是这样一个函数，它接收密码和明文，然后输出密文：

解密则相反，它接收密码和密文，然后输出明文：

在软件开发中，常用的对称加密算法有：

| **算法** | **密钥长度** | **工作模式** | **填充模式** |
| --- | --- | --- | --- |
| DES | 56/64 | ECB/CBC/PCBC/CTR/... | NoPadding/PKCS5Padding/... |
| AES | 128/192/256 | ECB/CBC/PCBC/CTR/... | NoPadding/PKCS5Padding/PKCS7Padding/... |
| IDEA | 128 | ECB | PKCS5Padding/PKCS7Padding/... |

密钥长度直接决定加密强度，而工作模式和填充模式可以看成是对称加密算法的参数和格式选择。Java标准库提供的算法实现并不包括所有的工作模式和所有填充模式，但是通常我们只需要挑选常用的使用就可以了。

最后注意，DES算法由于密钥过短，可以在短时间内被暴力破解，所以现在已经不安全了。

## 4.1使用AES加密

AES算法是目前应用最广泛的加密算法。我们先用ECB模式加密并解密：

Java标准库提供的对称加密接口非常简单，使用时按以下步骤编写代码：

根据算法名称/工作模式/填充模式获取Cipher实例；

根据算法名称初始化一个SecretKey实例，密钥必须是指定长度；

使用SerectKey初始化Cipher实例，并设置加密或解密模式；

传入明文或密文，获得密文或明文。

ECB模式是最简单的AES加密模式，它只需要一个固定长度的密钥，固定的明文会生成固定的密文，这种一对一的加密方式会导致安全性降低，更好的方式是通过CBC模式，它需要一个随机数作为IV参数，这样对于同一份明文，每次生成的密文都不同：