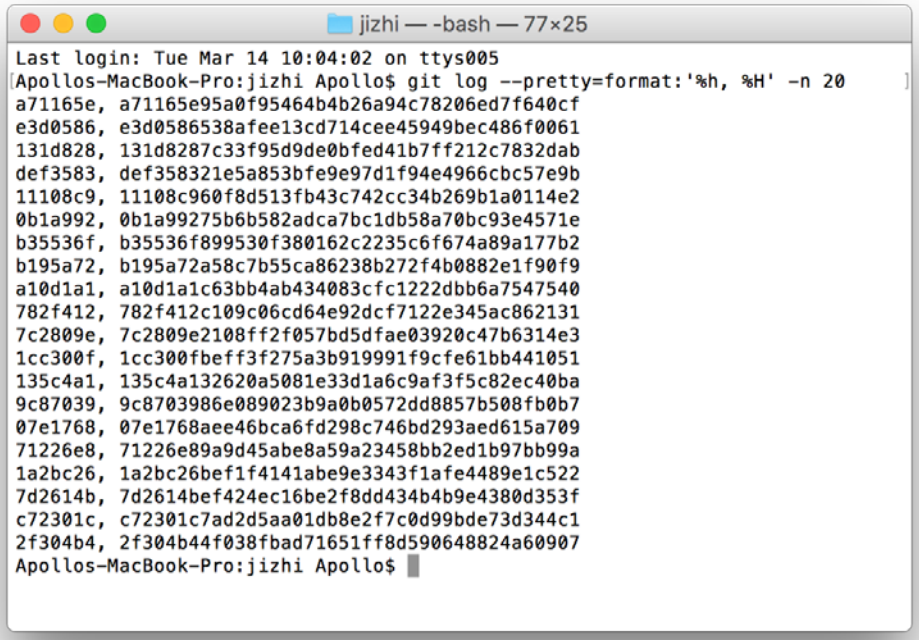
1. Hash 是什么，它的作用

先举个例子。我们每个活在世上的人，为了能够参与各种社会活动，都需要一个用于识别自己的标志。也许你觉得名字或是身份证就足以代表你这个人，但是这种代表性非常脆弱，因为重名的人很多，身份证也可以伪造。最可靠的办法是把一个人的所有基因序列记录下来用来代表这个人，但显然，这样做并不实际。而指纹看上去是一种不错的选择，虽然一些专业组织仍然可以模拟某个人的指纹，但这种代价实在太高了。

而对于在互联网世界里传送的文件来说，如何标志一个文件的身份同样重要。比如说我们下载一个文件，文件的下载过程中会经过很多网络服务器、路由器的中转，如何保证这个文件就是我们所需要的呢？我们不可能去一一检测这个文件的每个字节，也不能简单地利用文件名、文件大小这些极容易伪装的信息，这时候，我们就需要一种指纹一样的标志来检查文件的可靠性，这种指纹就是我们现在所用的 Hash 算法(也叫散列算法)。

散列算法（Hash Algorithm），又称哈希算法，杂凑算法，是一种从任意文件中创造小的数字「指纹」的方法。与指纹一样，散列算法就是一种以较短的信息来保证文件唯一性的标志，这种标志与文件的每一个字节都相关，而且难以找到逆向规律。因此，当原有文件发生改变时，其标志值也会发生改变，从而告诉文件使用者当前的文件已经不是你所需求的文件。

这种标志有何意义呢？之前文件下载过程就是一个很好的例子，事实上，现在大部分的网络部署和版本控制工具都在使用散列算法来保证文件可靠性。而另一方面，我们在进行文件系统同步、备份等工具时，使用散列算法来标志文件唯一性能帮助我们减少系统开销，这一点在很多云存储服务器中都有应用。



当然，作为一种指纹，散列算法最重要的用途在于给证书、文档、密码等高安全系数的内容添加加密保护。这一方面的用途主要是得益于散列算法的**不可逆性**，这种不可逆性体现在，你不仅不可能根据一段通过散列算法得到的指纹来获得原有的文件，也不可能简单地创造一个文件并让它的指纹与一段目标指纹相一致。散列算法的这种不可逆性维持着很多安全框架的运营，而这也将是本文讨论的重点。

2. Hash 算法有什么特点

一个优秀的 hash 算法，将能实现：

- 正向快速：给定明文和 hash 算法，在有限时间和有限资源内能计算出 hash 值。
- 逆向困难：给定（若干） hash 值，在有限时间内很难（基本不可能）逆推出明文。
- 输入敏感：原始输入信息修改一点信息，产生的 hash 值看起来应该都有很大不同。
- 冲突避免：很难找到两段内容不同的明文，使得它们的 hash 值一致（发生冲突）。即对于任意两个不同的数据块，其 hash 值相同的可能性极小；对于一个给定的数据块，找到和它 hash 值相同的数据块极为困难。

但在不同的使用场景中，如数据结构和安全领域里，其中对某一些特点会有所侧重。

2.1 Hash 在管理数据结构中的应用

在用到 hash 进行管理的数据结构中，就对速度比较重视，对抗碰撞不太看中，只要保证 hash 均匀分布就可以。比如 hashmap，hash 值（key）存在的目的是加速键值对的查找，key 的作用是为了将元素适当地放在各个桶里，对于抗碰撞的要求没有那么高。换句话说，hash 出来的 key，只要保证 value 大致均匀的放在不同的桶里就可以了。但整个算法的 set 性能，直接与 hash 值产生的速度有关，所以这时候的 hash 值的产生速度就尤为重要，以 JDK 中的 String.hashCode() 方法为例：

```
public int hashCode() {
    int h = hash;
    //hash default value : 0
    if (h == 0 && value.length > 0) {
        //value : char storage
        char val[] = value;
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

很简洁的一个乘加迭代运算，在不少的 hash 算法中，使用的是异或+加法进行迭代，速度和前者差不多。

2.1 Hash 在在密码学中的应用

在密码学中，hash 算法的作用主要是用于消息摘要和签名，换句话说，它主要用于对整个消息的完整性进行校验。举个例子，我们登陆知乎的时候都需要输入密码，那么知乎如果明文保存这个密码，那么黑客就很容易窃取大家的密码来登陆，特别不安全。那么知乎就想到了一个方法，使用 hash 算法生成一个密码的签名，知乎后台只保存这个签名值。由于 hash 算法是不可逆的，那么黑客即便得到这个签名，也丝毫没有用处；而如果你在网站登陆界面上输入你的密码，那么知乎后台就会重新计算一下这个 hash 值，与网站中储存的原 hash 值进行比对，如果相同，证明你拥有这个账户的密码，那么就会允许你登陆。银行也是如

此，银行是万万不敢保存用户密码的原文的，只会保存密码的 hash 值而而已。在这些应用场景里，对于抗碰撞和抗篡改能力要求极高，对速度的要求在其次。一个设计良好的 hash 算法，其抗碰撞能力是很高的。以 MD5 为例，其输出长度为 128 位，设计预期碰撞概率为，这是一个极小极小的数字——而即便是在 MD5 被王小云教授破解之后，其碰撞概率上限也高达，也就是说，至少需要找次才能有 1/2 的概率来找到一个与目标文件相同的 hash 值。而对于两个相似的字符串，MD5 加密结果如下：

```
MD5("version1") = "966634ebf2fc135707d6753692bf4b1e";
MD5("version2") = "2e0e95285f08a07dea17e7ee111b21c8";
```

可以看到仅仅一个比特位的改变，二者的 MD5 值就天差地别了

ps：其实把 hash 算法当成是一种加密算法，这是不准确的，我们知道加密总是相对于解密而言的，没有解密何谈加密呢，HASH 的设计以无法解为目的的。并且如果我们不附加一个随机的 salt 值，HASH 口令是很容易被字典攻击入侵的。

3. Hash 算法是如何实现的?

密码学和信息安全发展到现在，各种加密算法和散列算法已经不是只言片语所能解释得了的。在这里我们仅提供几个简单的概念供大家参考。

作为散列算法，首要的功能就是要使用一种算法把原有的体积很大的文件信息用若干个字符来记录，还要保证每一个字节都会对最终结果产生影响。那么大家也许已经想到了，求模这种算法就能满足我们的需要。

事实上，求模算法作为一种不可逆的计算方法，已经成为了整个现代密码学的根基。只要是涉及到计算机安全和加密的领域，都会有模计算的身影。散列算法也并不例外，一种最原始的散列算法就是单纯地选择一个数进行模运算，比如以下程序。

```
#  构造散列函数
def hash(a):
    return a % 8

#  测试散列函数功能
print(hash(233))
print(hash(234))
print(hash(235))

# 输出结果
- 1
```

```
- 2
- 3
```

很显然，上述的程序完成了一个散列算法所应当实现的初级目标：用较少的文本量代表很长的内容（求模之后的数字肯定小于 8）。但也许你已经注意到了，单纯使用求模算法计算之后的结果带有明显的规律性，这种规律将导致算法将能难保证不可逆性。所以我们将使用另外一种手段，那就是异或。

再来看下面一段程序，我们在散列函数中加入一个异或过程。

```
#  构造散列函数
def hash(a):
    return (a % 8) ^ 5

#  测试散列函数功能
print(hash(233))
print(hash(234))
print(hash(235))

# 输出结果
- 4
- 7
- 6
```

很明显的，加入一层异或过程之后，计算之后的结果规律性就不是那么明显了。

当然，大家也许会觉得这样的算法依旧很不安全，如果用户使用连续变化的一系列文本与计算结果相比对，就很有可能找到算法所包含的规律。但是我们还有其他的办法。比如在进行计算之前对原始文本进行修改，或是加入额外的运算过程（如移位），比如以下程序。

```
#  构造散列函数
def hash(a):
    return (a + 2 + (a << 1)) % 8 ^ 5

#  测试散列函数功能
print(hash(233))
print(hash(234))
print(hash(235))

# 输出结果
- 0
- 5
```

```
- 6
```

这样处理得到的散列算法就很难发现其内部规律，也就是说，我们并不能很轻易地给出一个数，让它经过上述散列函数运算之后的结果等于 4——除非我们去穷举测试。

上面的算法是不是很简单？事实上，下面我们即将介绍的常用算法 MD5 和 SHA1，其本质算法就是这么简单，只不过会加入更多的循环和计算，来加强散列函数的可靠性。

4. Hash 有哪些流行的算法

目前流行的 Hash 算法包括 MD5、SHA-1 和 SHA-2。

- MD4（RFC 1320）是 MIT 的 Ronald L. Rivest 在 1990 年设计的，MD 是 Message Digest 的缩写。其输出为 128 位。MD4 已证明不够安全。
- MD5（RFC 1321）是 Rivest 于 1991 年对 MD4 的改进版本。它对输入仍以 512 位分组，其输出是 128 位。MD5 比 MD4 复杂，并且计算速度要慢一点，更安全一些。MD5 已被证明不具备"强抗碰撞性"。
- SHA （Secure Hash Algorithm）是一个 Hash 函数族，由 NIST（National Institute of Standards and Technology）于 1993 年发布第一个算法。目前知名的 SHA-1 在 1995 年面世，它的输出为长度 160 位的 hash 值，因此抗穷举性更好。SHA-1 设计时基于和 MD4 相同原理，并且模仿了该算法。SHA-1 已被证明不具"强抗碰撞性"。
- 为了提高安全性，NIST 还设计出了 SHA-224、SHA-256、SHA-384，和 SHA-512 算法（统称为 SHA-2），跟 SHA-1 算法原理类似。SHA-3 相关算法也已被提出。

可以看出，上面这几种流行的算法，它们最重要的一点区别就是"强抗碰撞性"。

5. 那么，何谓 Hash 算法的「碰撞」？

你可能已经发现了，在实现算法章节的第一个例子，我们尝试的散列算法得到的值一定是一个不大于 8 的自然数，因此，如果我们随便拿 9 个数去计算，肯定至少会得到两个相同的值，我们把这种情况就叫做散列算法的「碰撞」（Collision）。

这很容易理解，因为作为一种可用的散列算法，其位数一定是有限的，也就是说它能记录的文件是有限的——而文件数量是无限的，两个文件指纹发生碰撞的概率永远不会是零。

但这并不意味着散列算法就不能用了，因为凡事都要考虑代价，买光所有彩票去中一次头奖是毫无意义的。现代散列算法所存在的理由就是，它的不可逆性能在较大概率上得到实现，也就是说，发现碰撞的概率很小，这种碰撞能被利用的概率更小。

随意找到一组碰撞是有可能的，只要穷举就可以。散列算法得到的指纹位数是有限的，比如 MD5 算法指纹字长为 128 位，意味着只要我们穷举 21282128 次，就肯定能得到一组碰撞——当然，这个时间代价是难以想象的，而更重要的是，仅仅找到一组碰撞并没有什么实际意义。更有意义的是，如果我们已经有了一组指纹，能否找到一个原始文件，让它的散列计算结果等于这组指纹。如果这一点被实现，我们就可以很容易地篡改和伪造网络证书、密码等关键信息。

你也许已经听过 MD5 已经被破解的新闻——但事实上，即便是 MD5 这种已经过时的散列算法，也很难实现逆向运算。我们现在更多的还是依赖于海量字典来进行尝试，也就是通过已经知道的大量的文件——指纹对应关系，搜索某个指纹所对应的文件是否在数据库里存在。

5.1 MD5 的实际碰撞案例

下面让我们来看看一个真实的碰撞案例。我们之所以说 MD5 过时，是因为它在某些时候已经很难表现出散列算法的某些优势——比如在应对文件的微小修改时，散列算法得到的指纹结果应当有显著的不同，而下面的程序说明了 MD5 并不能实现这一点。

```
import hashlib



#  两段 HEX 字节串，注意它们有细微差别
a =
bytearray.fromhex("0e306561559aa787d00bc6f70bbdfe3404cf03659e704f8534c0
0ffb659c4c8740cc942feb2da115a3f4155cbb8607497386656d7d1f34a42059d78f5a8
dd1ef")
```

```
b =
bytearray.fromhex("0e306561559aa787d00bc6f70bbdfe3404cf03659e744f8534c0
0ffb659c4c8740cc942feb2da115a3f415dcbb8607497386656d7d1f34a42059d78f5a8
dd1ef")

#  输出 MD5，它们的结果一致
print(hashlib.md5(a).hexdigest())
print(hashlib.md5(b).hexdigest())

### a 和 b 输出结果都为：
cee9a457e790cf20d4bdaa6d69f01e41
cee9a457e790cf20d4bdaa6d69f01e41
```

而诸如此类的碰撞案例还有很多，上面只是原始文件相对较小的一个例子。事实上现在我们用智能手机只要数秒就能找到 MD5 的一个碰撞案例，因此，MD5 在数年前就已经不被推荐作为应用中的散列算法方案，取代它的是 SHA 家族算法，也就是安全散列算法（Secure Hash Algorithm，缩写为 SHA）。

5.2 SHA 家族算法以及 SHA1 碰撞

安全散列算法与 MD5 算法本质上的算法是类似的，但安全性要领先很多——这种领先型更多的表现在碰撞攻击的时间开销更大，当然相对应的计算时间也会慢一点。

SHA 家族算法的种类很多，有 SHA0、SHA1、SHA256、SHA384 等等，它们的计算方式和计算速度都有差别。其中 SHA1 是现在用途最广泛的一种算法。包括 GitHub 在内的众多版本控制工具以及各种云同步服务都是用 SHA1 来区别文件，很多安全证书或是签名也使用 SHA1 来保证唯一性。长期以来，人们都认为 SHA1 是十分安全的，至少大家还没有找到一次碰撞案例。

但这一事实在 2017 年 2 月破灭了。CWI 和 Google 的研究人员们成功找到了一例 SHA1 碰撞，而且很厉害的是，发生碰撞的是两个真实的、可阅读的 PDF 文件。这两个 PDF 文件内容不相同，但 SHA1 值完全一样。(对于这件事的影响范围及讨论，可参考知乎上的讨论:如何评价 2 月 23 日谷歌宣布实现了 SHA-1 碰撞？)
所以，对于一些大的商业机构来说， MD5 和 SHA1 已经不够安全，推荐至少使用 SHA2-256 算法。

6. Hash 在 Java 中的应用

6.1 HashMap 的复杂度

在介绍 HashMap 的实现之前，先考虑一下，HashMap 与 ArrayList 和 LinkedList 在数据复杂度上有什么区别。下图是他们的性能对比图：

	获取	查找	添加/删除	空间
ArrayList	O(1)	O(1)	O(N）	O(N)
LinkedList	O(N)	O(N)	O(1)	O(N)
HashMap	O(N/Bucket_size)	O(N/Bucket_size)	O(N/Bucket_size)	O(N)

可以看出 HashMap 整体上性能都非常不错，但是不稳定，为 O(N/Buckets)，N 就是以数组中没有发生碰撞的元素，Buckets 是因碰撞产生的链表。

注：发生碰撞实际上是非常稀少的，所以 N/Bucket_size 约等于 1

HashMap 是对 Array 与 Link 的折衷处理，Array 与 Link 可以说是两个速度方向的极端，Array 注重于数据的获取，而处理修改（添加/删除）的效率非常低；Link 由于是每个对象都保持着下一个对象的指针，查找某个数据需要遍历之前所有的数据，所以效率比较低，而在修改操作中比较快。

6.2 HashMap 的实现

本文以 JDK8 的 API 实现进行分析

6.2.1 对 key 进行 Hash 计算

在 JDK8 中，由于使用了红黑树来处理大的链表开销，所以 hash 这边可以更加省力了，只用计算 hashCode 并移动到低位就可以了。

```
static final int hash(Object key) {
    int h;
    //计算 hashCode，并无符号移动到低位
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

举个例子: 363771819^(363771819 >>> 16)

```
0001 0101 1010 1110 1011 0111 1010 1011(363771819)
```

```
0000 0000 0000 0000 0001 0101 1010 1110(5550) XOR


------------------------------------ =
0001 0101 1010 1110 1010 0010 0000 0101(363766277)
```

这样做可以实现了高地位更加均匀地混到一起。

下面给出在 Java 中几个常用的哈希码(hashCode)的算法。

1. Object 类的 hashCode. 返回对象的经过处理后的内存地址，由于每个对象的内存地址都不一样，所以哈希码也不一样。这个是 native 方法，取决于 JVM 的内部设计，一般是某种 C 地址的偏移。

2. String 类的 hashCode. 根据 String 类包含的字符串的内容，根据一种特殊算法返回哈希码，只要字符串的内容相同，返回的哈希码也相同。

3. Integer 等包装类，返回的哈希码就是 Integer 对象里所包含的那个整数的数值，例如 Integer i1=new Integer(100), i1.hashCode 的值就是 100 。由此可见，2 个一样大小的 Integer 对象，返回的哈希码也一样。

4. int，char 这样的基础类，它们不需要 hashCode，如果需要存储时，将进行自动装箱操作，计算方法同上。

6.2.2 获取到数组的 index 的位置

计算了 Hash，我们现在要把它插入数组中了

```
i = (tab.length - 1) & hash;
```

通过位运算，确定了当前的位置，因为 HashMap 数组的大小总是 2^n，所以实际的运算就是 (0xfff…ff) & hash ，这里的 tab.length-1 相当于一个 mask，滤掉了大于当前长度位的 hash，使每个 i 都能插入到数组中。

6.2.3 生成包装类

这个对象是一个包装类，Node

```
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;
```

```
        //getter and setter .etc.
}
```

6.2.4 插入包装类到数组

(1). 如果输入当前的位置是空的，就插进去，如图，左为插入前，右为插入后

```
   0          0

   |          |

   1 -> null   1 - > null

   |          |

   2 -> null   2 - > null

   |          |

   ..-> null   ..- > null

   |          |

   i -> null   i - > new node

   |          |

   n -> null   n - > null
```

(2). 如果当前位置已经有了 node，且它们发生了碰撞，则新的放到前面，旧的放到后面，这叫做链地址法处理冲突。

```
   0          0

   |          |

   1 -> null   1 - > null

   |          |
```

```
   2 -> null   2 - > null

   |          |

   ..-> null   ..- > null

   |          |

   i -> old    i - > new - > old

   |          |

   n -> null   n - > null
```

我们可以发现，失败的 hashCode 算法会导致 HashMap 的性能由数组下降为链表，所以想要避免发生碰撞，就要提高 hashCode 结果的均匀性。

6.3 扩容

如果当表中的 75%已经被占用，即视为需要扩容了

```
(threshold = capacity * load factor ) < size
```

它主要有两个步骤：

6.3.1 容量加倍

左移 1 位，就是扩大到两倍，用位运算取代了乘法运算

```
newCap = oldCap << 1;
newThr = oldThr << 1;
```

6.3.2 遍历计算 Hash

```
for (int j = 0; j < oldCap; ++j) {
        Node<K,V> e;
        //如果发现当前有 Bucket
        if ((e = oldTab[j]) != null) {
            oldTab[j] = null;
            //如果这里没有碰撞
            if (e.next == null)
                //重新计算 Hash，分配位置
```

```
                    newTab[e.hash & (newCap - 1)] = e;
                //这个见下面的新特性介绍，如果是树，就填入树
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                //如果是链表，就保留顺序....目前就看懂这点
                else { // preserve order
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        else {
                            if (hiTail == null)
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
                    }
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
```

由此可以看出扩容需要遍历并重新赋值，成本非常高，所以选择一个好的初始容量非常重要。

6.4 扩容如何提升性能?

- 解决扩容损失：如果知道大致需要的容量，把初始容量设置好以解决扩容损失；
  比如我现在有 1000 个数据，需要 1000/0.75 = 1333 个坑位，又 1024 < 1333 < 2048，所以最好使用 2048 作为初始容量。

- 解决碰撞损失：使用高效的 HashCode 与 loadFactor，这个…由于 JDK8 的高性能出现，这儿问题也不大了。

6.5 HashMap 与 HashTable 的主要区别

在很多的 Java 基础书上都已经说过了，他们的主要区别其实就是 Table 全局加了线程同步保护

- HashTable 线程更加安全，代价就是因为它粗暴的添加了同步锁，所以会有性能损失。
- 其实有更好的 concurrentHashMap 可以替代 HashTable，一个是方法级，一个是 Class 级。

6.6 在 Android 中使用 SparseArray 代替 HashMap

官方推荐使用 SparseArray([spɑ:s][ə’reɪ],稀疏的数组)或者 LongSparseArray 代替 HashMap。官方总结有一下几点好处：

- SparseArray 使用基本类型(Primitive)中的 int 作为 Key，不需要 Pair

总结

「The Algorithm Design Manual」一书中提到，雅虎的 Chief Scientist ，Udi Manber 曾说过，在 yahoo 所应用的算法中，最重要的三个是：Hash，Hash 和 Hash。其实从上文中所举的 git 用 sha1 判断文件更改，密码用 MD5 生成摘要后加盐等等对 Hash 的应用可看出，Hash 的在计算机世界扮演着多么重要的角色。另书中还举了一个很有趣的显示中例子：

一场拍卖会中，物品是价高者得，如果每个人只有一次出价机会，同时提交自己的价格后，最后一起公布，出价最高则胜出。这种形式存在作弊的可能，如果有出价者能 hack 进后台，然后将自己的价格改为最高价 +1，则能以最低的代价获得胜利。如何杜绝这种作弊呢？

答案很简单，参与者都提交自身出价的 hash 值就可以了，即使有人能黑进后台也无法得知明文价格，等到公布之时，再对比原出价与 hash 值是否对应即可。是不是很巧妙？

是的，上面的做法，与上文提到的网站上储存密码用 MD5 值而非明文，是同一种思想，殊途同归。

可以看到无论是密码学、数据结构、现实生活中的应用，到处可以看到 Hash 的影子，通过这篇文章的介绍，相信你不仅知其名，也能懂其意。