|  |  |  |
| --- | --- | --- |
|  | Hashtable | HashMap |
| 继承的父类不同 | Dictionary | AbstractMap |
| 都实现了同时实现了map、Cloneable（可复制）、Serializable（可序列化）这三个接口 | |
| 默认容量 | 11 | 16 |
| Table的初始化时机 | 构造函数中初始化 | 第一次put |
| 并发操作 | 使用同步机制，  实际应用程序中，仅仅是Hashtable本身的同步并不能保证程序在并发操作下的正确性，需要高层次的并发保护。  下面的代码试图在key所对应的value值等于x的情况下修改value为x+1  {   value = hashTable.get(key);     if(value.intValue()== x){  hashTable.put(key,      new Integer(value.intValue()+1));     }  }  如2个线程同时执行以上代码，可能放入不是x+1，而是x+2. | 没有同步机制，需要使用者自己进行并发访问控制 |
| 数据遍历的方式 | Iterator 和 Enumeration | Iterator |
| 是否支持fast-fail | 用Iterator遍历，支持fast-fail  用Enumeration不支持fast-fail. | 支持fast-fail |
| 是否接受值为null的Key 或Value？ | 不接受 | 接受 |
| 根据hash值计算数组下标的算法 | 当数组长度较小，并且Key的hash值低位数值分散不均匀时，不同的hash值计算得到相同下标值的几率较高    hash = key.hashCode();  index=(hash&0x7FFFFFFF) % tab.length;//去掉其符号位置 | 优于hashtable，通过对Key的hash做移位运算和位的与运算，使其能更广泛地分散到数组的不同位置    final int hash(Object k) { int h = hashSeed; if (0 != h && k instanceof String) { return sun.misc.Hashing.stringHash32((String) k); }  h ^= k.hashCode();    hash = hash (k);  index = indexFor(hash, table.length);    static int hash(Object x) {   int h = x.hashCode();  h += ~(h << 9);    h ^= (h >>> 14);     h += (h << 4);    h ^= (h >>> 10);    return h;  }把后面几位移动到前面  static int indexFor(int h, int length) {  return h & (length-1);  } |
| Entry数组的长度 | 缺省初始长度为11，初始化时可以指定initial capacity | 缺省初始长度为16，长度始终保持2的n次方初始化时可以指定initial capacity，若不是2的次方，HashMap将选取第一个大于initial capacity 的2n次方值作为其初始长度 |
| LoadFactor负荷因子 | 0.75 | |
| 负荷超过（loadFactor \* 数组长度）时，内部数据的调整方式 | 扩展数组：2\*原数组长度+1 | 扩展数组： 原数组长度 \* 2 |
| 两者都会重新根据Key的hash值计算其在数组中的新位置，重新放置。算法相似，时间、空间效率相同 | |