# 基础类

## Object

Object中包含的方法:

1.protected Object clone() //创建并返回此对象的一个副本。

2.boolean equals(Object obj) //指示某个其他对象是否与此对象“相等”。

实现其实就是用 == 判断的其实是地址。比较的是栈中的值。（原型数据相同或相同引用）

3.protected void finalize() //当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

finalize会在对象被垃圾回收时由垃圾回收器调用，垃圾对象是指没有引用指向的对象。

4.Class<? extends Object> getClass() //返回一个对象的运行时类。

5.int hashCode() //返回该对象的哈希码值。

6.void notify() //唤醒在此对象监视器上等待的单个线程。

7.void notifyAll() //唤醒在此对象监视器上等待的所有线程。

8.String toString() //返回该对象的字符串：类名 @ 内存地址。

getClass().getName() + "@" + Integer.toHexString(hashCode())

9.void wait() //导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。

void wait(long timeout) //导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或notifyAll() 方法，或者超过指定的时间量。

void wait(long timeout, int nanos) //导致当前的线程等待，直到其他线程调用此对象的notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量。

## String

### String对象

1、String 类是不可改变的，所以你一旦创建了 String 对象，那它的值就无法改变了，相关的任何change操作都会生成新的对象。

2、String类是final类，也即意味着String类不能被继承，并且它的成员方法都默认为final方法。

3、String类其实是通过char数组来保存字符串的。

4、每当我们创建字符串常量时，JVM会首先检查字符串常量池，如果该字符串已经存在常量池中，那么就直接返回常量池中的实例引用。如果字符串不存在常量池中，就会实例化该字符串并且将其放到常量池中。

Java中的常量池，实际上分为两种形态：静态常量池和运行时常量池。

所谓静态常量池，即\*.class文件中的常量池，class文件中的常量池不仅仅包含字符串(数字)字面量，还包含类、方法的信息，占用class文件绝大部分空间。

而运行时常量池，则是jvm虚拟机在完成类装载操作后，将class文件中的常量池载入到内存中，并保存在方法区中，我们常说的常量池，就是指**方法区**中的运行时常量池。

实例：

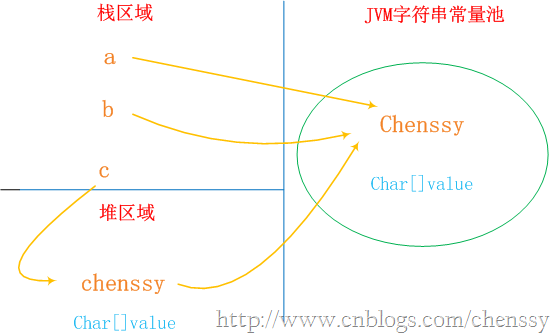
String a = "chenssy";

String b = "chenssy";

a、b和字面上的chenssy都是指向JVM字符串常量池中的"chenssy"对象，他们指向同一个对象。

String c = new String("chenssy");

new关键字一定会产生一个对象chenssy（注意这个chenssy和上面的chenssy不同），同时这个对象是存储在堆中。所以上面应该产生了两个对象：保存在栈中的c和保存堆中chenssy。但是在Java中根本就不存在两个完全一模一样的字符串对象。故堆中的chenssy应该是引用字符串常量池中chenssy。所以c、chenssy、池chenssy的关系应该是：c--->chenssy--->池chenssy。整个关系如下：



### String方法

（1）对于==，如果作用于基本数据类型的变量（byte,short,char,int,long,float,double,boolean ），则直接比较其存储的"值"是否相等；如果作用于引用类型的变量（String），则比较的是所指向的对象的地址（即是否指向同一个对象）。

（2）equals方法是基类Object中的方法，因此对于所有的继承于Object的类都会有该方法。在Object类中，equals方法是用来比较两个对象的引用是否相等，即是否指向同一个对象。

（3）对于equals方法，注意：equals方法不能作用于基本数据类型的变量。如果没有对equals方法进行重写，则比较的是引用类型的变量所指向的对象的地址；而String类对equals方法进行了重写，用来比较指向的字符串对象所存储的字符串是否相等。其他的一些类诸如Double，Date，Integer等，都对equals方法进行了重写用来比较指向的对象所存储的内容是否相等。

（4）String中hashCode方法的实现源码。

public int hashCode() {

int h = hash;

if (h == 0 && value.length > 0) {

char val[] = value;

for (int i = 0; i < value.length; i++) {

h = 31 \* h + val[i];

}

hash = h;

}

return h;

}

String类中的hashCode计算方法还是比较简单的，就是以31为权，每一位为字符的ASCII值进行运算，用自然溢出来等效取模。

哈希计算公式可以计为s[0]\*31^(n-1) + s[1]\*31^(n-2) + ... + s[n-1]

关于为什么取31为权，可以参考StackOverflow上的这个问题

主要是因为31是一个奇质数，所以31\*i=32\*i-i=(i<<5)-i，这种位移与减法结合的计算相比一般的运算快很多。

String、StringBuffer、StringBuilder的区别

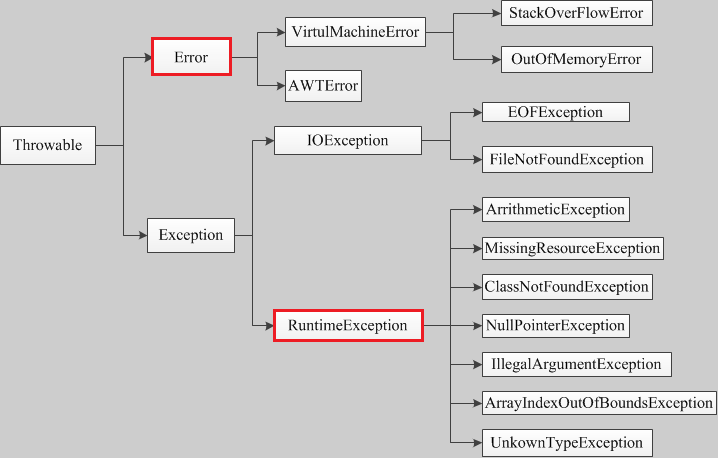
（1）可变与不可变：String是不可变字符串对象，StringBuilder和StringBuffer是可变字符串对象（其内部的字符数组长度可变）。

（2）是否多线程安全：String中的对象是不可变的，也就可以理解为常量，显然线程安全。StringBuffer 与 StringBuilder 中的方法和功能完全是等价的，只是StringBuffer 中的方法大都采用了synchronized 关键字进行修饰，因此是线程安全的，而 StringBuilder 没有这个修饰，可以被认为是非线程安全的。

（3）String、StringBuilder、StringBuffer三者的执行效率：

StringBuilder > StringBuffer > String 当然这个是相对的，不一定在所有情况下都是这样。比如String str = "hello"+ "world"的效率就比 StringBuilder st  = new StringBuilder().append("hello").append("world")要高。因此，这三个类是各有利弊，应当根据不同的情况来进行选择使用：

## Exception



总体上我们根据Javac对异常的处理要求，将异常类分为2类。

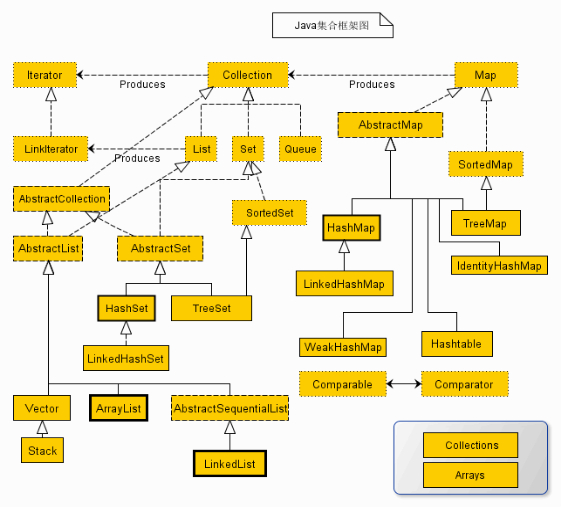
非检查异常（unckecked exception）：Error 和 RuntimeException 以及他们的子类。javac在编译时，不会提示和发现这样的异常，不要求在程序处理这些异常。所以如果愿意，我们可以编写代码处理（使用try…catch…finally）这样的异常，也可以不处理。对于这些异常，我们应该修正代码，而不是去通过异常处理器处理 。这样的异常发生的原因多半是代码写的有问题。如除0错误ArithmeticException，错误的强制类型转换错误ClassCastException，数组索引越界ArrayIndexOutOfBoundsException，使用了空对象NullPointerException等等。

检查异常（checked exception）：除了Error 和 RuntimeException的其它异常。javac强制要求程序员为这样的异常做预备处理工作（使用try…catch…finally或者throws）。在方法中要么用try-catch语句捕获它并处理，要么用throws子句声明抛出它，否则编译不会通过。这样的异常一般是由程序的运行环境导致的。因为程序可能被运行在各种未知的环境下，而程序员无法干预用户如何使用他编写的程序，于是程序员就应该为这样的异常时刻准备着。如SQLException , IOException,ClassNotFoundException 等。

当没有try catch时，发生了异常，这样一直向调用栈的栈底回溯。这种行为叫做异常的冒泡，异常的冒泡是为了在当前发生异常的函数或者这个函数的caller中找到最近的异常处理程序。由于这个例子中没有使用任何异常处理机制，因此异常最终由main函数抛给JRE，导致程序终止。

throws声明：如果一个方法内部的代码会抛出检查异常（checked exception），而方法自己又没有完全处理掉，则javac保证你必须在方法的签名上使用throws关键字声明这些可能抛出的异常，否则编译不通过。throws仅仅是将函数中可能出现的异常向调用者声明，而自己则不具体处理。

# 集合类



Collection接口是集合类的根接口，Java中没有提供这个接口的直接的实现类。但是却让其被继承产生了两个接口，就是Set和List。Set中不能包含重复的元素。List是一个有序的集合，可以包含重复的元素，提供了按索引访问的方式。

Map是Java.util包中的另一个接口，它和Collection接口没有关系，是相互独立的，但是都属于集合类的一部分。Map包含了key-value对。Map不能包含重复的key，但是可以包含相同的value。

Iterator，所有的集合类，都实现了Iterator接口，这是一个用于遍历集合中元素的接口，主要包含以下三种方法：

1.hasNext()是否还有下一个元素。

2.next()返回下一个元素。

3.remove()删除当前元素。

## 1、List（有序、可重复）

List里存放的对象是有序的，同时也是可以重复的，List关注的是索引，拥有一系列和索引相关的方法，查询速度快。因为往list集合里插入或删除数据时，会伴随着后面数据的移动，所有插入删除数据速度慢。

### ArrayList

基于动态数组的数据结构，用get和set方法是花费常数时间的，但是如果插入元素和删除元素，除非插入和删除的位置都在表末尾，否则代码开销会很大，因为里面需要数组的移动。

ArrayList 初始化大小是 10，扩容点规则是，新增的时候发现容量不够用了，就去扩容，扩容大小规则是，扩容后的大小= 原始大小+原始大小/2 + 1。

### LinkedList

基于双向链表的数据结构，get会非常消耗资源，除非位置离头部很近。但是插入和删除元素花费常数时间。

没有初始化大小，也没有扩容的机制，就是一直在前面或者后面新增就好。

## 2、Set（无序、不能重复）

Set里存放的对象是无序，不能重复的，集合中的对象不按特定的方式排序，只是简单地把对象加入集合中。

### HashSet

其底层其实是包装了HashMap去实现。Set集合是不允许重复元素的，HashSet需要同时通过equals和HashCode来判断两个元素是否相等，具体规则是，如果两个元素通过equals为true，并且两个元素的hashCode相等，则这两个元素相等（即重复）。所以如果要重写保存在HashSet中的对象的equals方法，也要重写hashCode方法。

HashSet hs = new HashSet();

        hs.add(new Person("jack", 20));

        hs.add(new Person("rose", 20));

### TreeSet

TreeSet实现了SortedSet接口，顾名思义这是一种排序的Set集合，本质上是一个红黑树原理。红黑树算法的规则: 左小右大。正因为它是排序了的，所以相对HashSet来说，TreeSet提供了一些额外的按排序位置访问元素的方法，例如first(), last(), lower(), higher(), subSet(), headSet(), tailSet()。

TreeSet集合排序的两种方式：1、让元素自身具备比较性，需要元素实现Comparable接口，重写compareTo方法，也就是让元素自身具备比较性，这种方式叫做元素的自然排序也叫做默认排序。2、让容器自身具备比较性，定义一个类实现Comparator 接口，覆盖compare方法，当Comparable比较方式，及Comparator比较方式同时存在，以Comparator比较方式为主。通过compareTo或者compare方法中的来保证元素的唯一性。添加的元素必须要实现Comparable或Comparator接口。当compareTo()或compare()函数返回值为0时，说明两个对象相等，此时该对象不会添加进来。

TreeSet ts = new TreeSet();

ts.add("ccc");

ts.add("aaa");

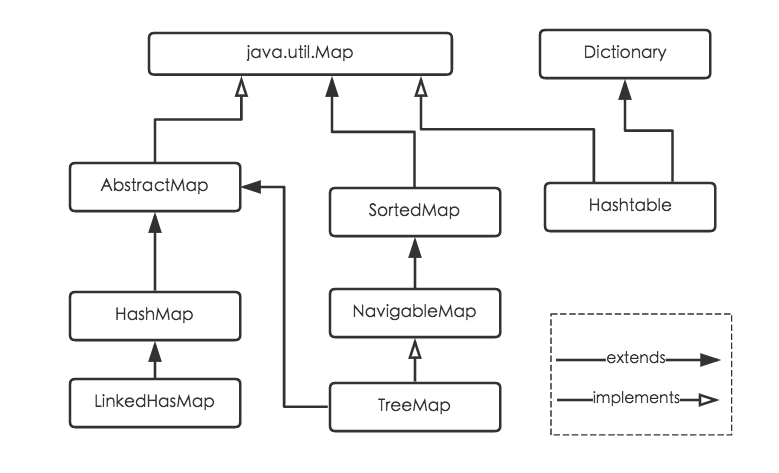
### LinkedHashSet

LinkedHashSet是HashSet的一个子类，LinkedHashSet也根据HashCode的值来决定元素的存储位置，但同时它还用一个链表来维护元素的插入顺序，插入的时候即要计算hashCode又要维护链表，而遍历的时候只需要按链表来访问元素。

LinkedHashSet本质上也是从LinkedHashMap而来，LinkedHashSet的所有方法都继承自HashSet, 而它能维持元素的插入顺序的性质则继承自LinkedHashMap。

## 3、Map（键值对、键唯一、值不唯一）

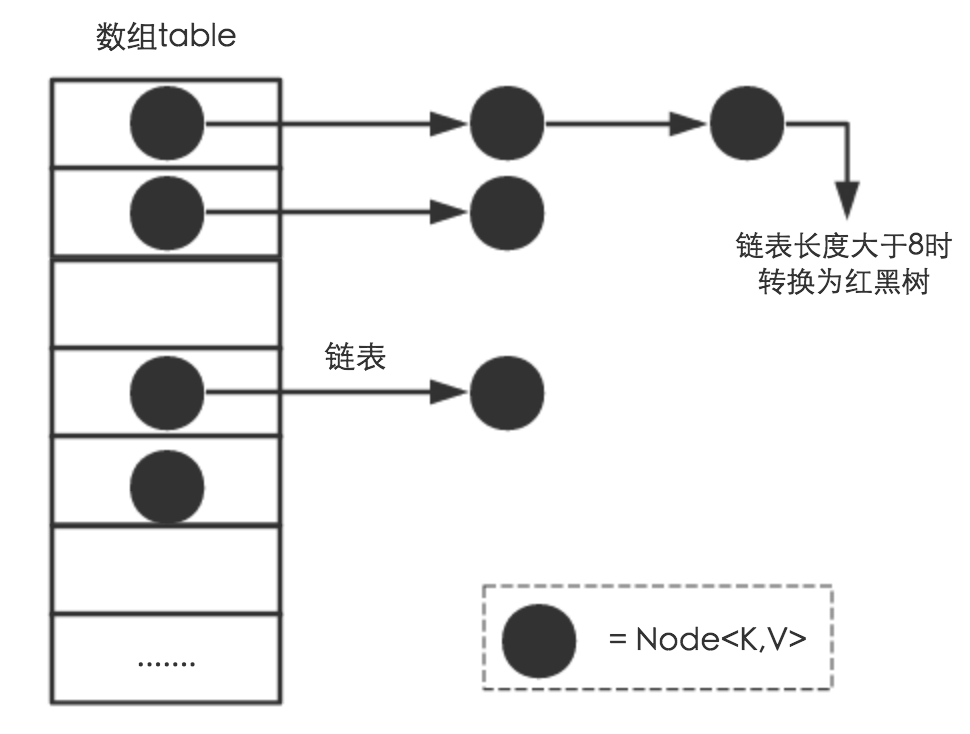
Map集合中存储的是键值对，键不能重复，值可以重复。根据键得到值，对map集合遍历时先得到键的set集合，对set集合进行遍历，得到相应的值。



### HashMap

它根据键的hashCode值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。 HashMap最多只允许一条记录的键为null，允许多条记录的值为null。HashMap非线程安全，即任一时刻可以有多个线程同时写HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections的synchronizedMap方法使HashMap具有线程安全的能力，或者使用ConcurrentHashMap。

从结构实现来讲，HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的



(1) HashMap类中有一个非常重要的字段，就是 Node[] table，即哈希桶数组，明显它是一个Node的数组。我们来看Node[JDK1.8]是何物。

static class Node<K,V> implements Map.Entry<K,V> {

final int hash; //用来定位数组索引位置

final K key;

V value;

Node<K,V> next; //链表的下一个node

Node(int hash, K key, V value, Node<K,V> next) { ... }

public final K getKey(){ ... }

public final V getValue() { ... }

public final String toString() { ... }

public final int hashCode() { ... }

public final V setValue(V newValue) { ... }

public final boolean equals(Object o) { ... }

}

Node是HashMap的一个内部类，实现了Map.Entry接口，本质是就是一个映射(键值对)。上图中的每个黑色圆点就是一个Node对象。

Entry是HashMap的内部类 包含四个值（next，key，value，hash），其中next是一个指向 Entry的指针，key相当于上面节点的值 value对应要保存的值，hash值由key产生，hashmap中要找到某个元素，需要根据hash值来求得对应数组中的位置，然后在由key来在链表中找Entry的位置。

(2) HashMap就是使用哈希表来存储的。哈希表为解决冲突，可以采用开放地址法和链地址法等来解决问题，Java中HashMap采用了链地址法。链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被Hash后，得到数组下标，把数据放在对应下标元素的链表上。

在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。

#### 构造函数

HashMap(int initialCapacity, float loadFactor)

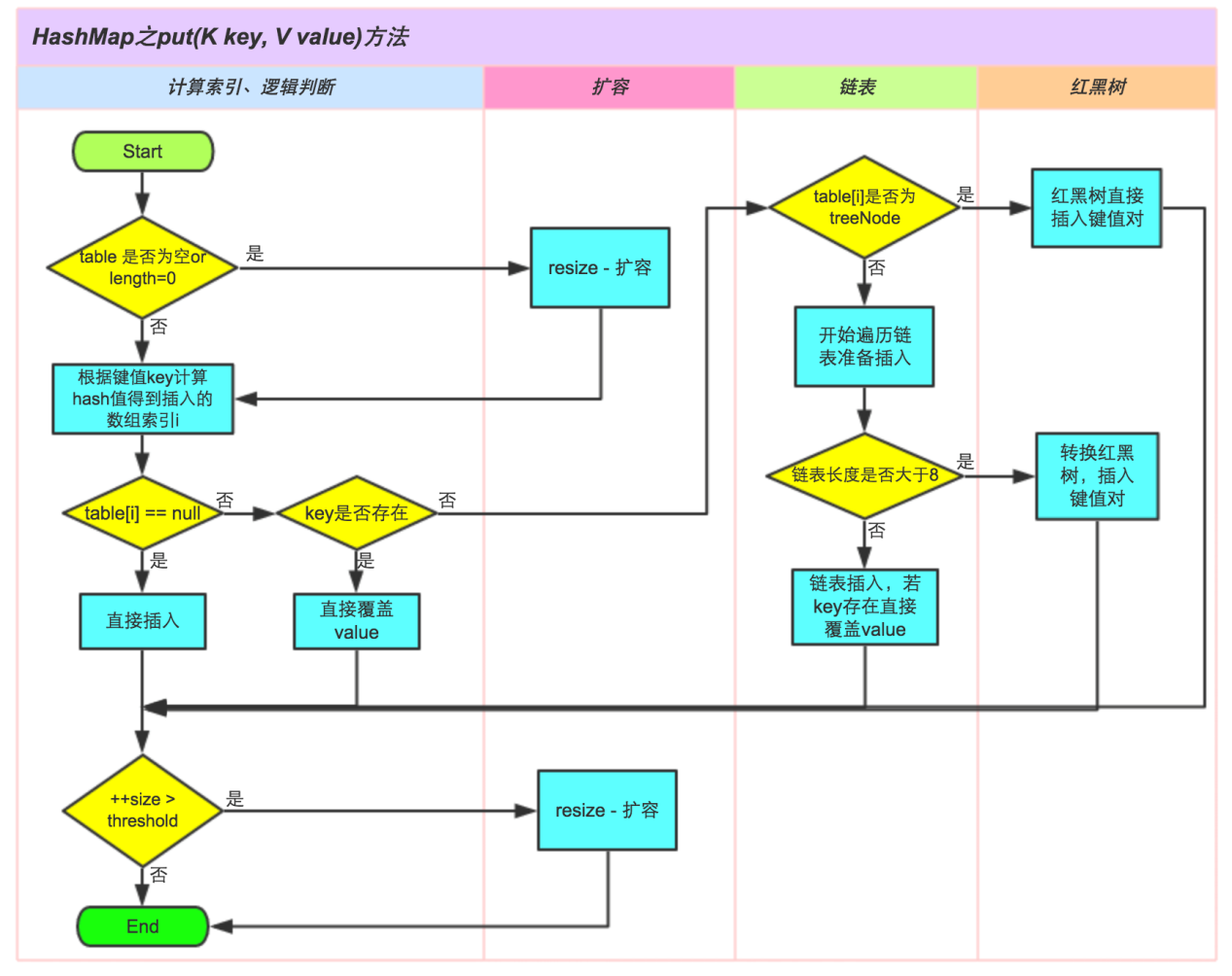
①填装因子:loadFactor 表示填装因子的大小，简单的介绍一下填装因子：假设数组大小为20，每个放到数组中的元素mod 17，所有元素取模后放的位置是（0–16） 此时填装因子的大小为 17/20 ,装填因子就为0.85啦，你装填因子越小，说明你备用的内存空间越多，装填因子的选定，可以影响冲突的产生，装填因子越小，冲突越小。

②initialCapacity初始化hashmap中table表的大小。哈希桶数组table的长度length大小必须为2的n次方

③如果用户直接使用HashMap（）构造函数来new一个HashMap 会生成一个大小为16，填装因子为0.75的 HashMap。

#### put方法：

调用key的hashCode()方法得到其hashCode 值，hash值高16位异或低16位(h = k.hashCode()) ^ (h >>> 16)，再对length取模h& (length-1)，相当于h%length，得到数组索引(key可以为 null ，此时hash=0)



①.判断键值对数组table[i]是否为空或为null，否则执行resize()进行扩容；

②. 调用key的hashCode()方法得到其hashCode 值，hash值高16位异或低16位(h = k.hashCode()) ^ (h >>> 16)，再对length取模h& (length-1)，相当于h%length，得到数组索引(key可以为 null ，此时hash=0)，得到插入的数组索引i，如果table[i]==null，直接新建节点添加，转向⑥，如果table[i]不为空，转向③；

③.判断table[i]的首个元素是否和key一样，如果相同直接覆盖value，否则转向④，这里的相同指的是hashCode以及equals；

④.判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；

⑤.遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

⑥.插入成功后，判断实际存在的键值对数量size是否超多了最大容量threshold，如果超过，进行扩容。

#### get方法

① 通过key来链表中查找元素包括两个过程，先由hash找到链（hash由key产生，相同的hash值放在同一条链上，与put方法相同取法，）再用key在链上找。

② 如果key为null则只在table[0]和其链上查找，因为key为null都放在table[0]及其链上了。

③在链上查找key相同的(hashCode以及equals相同)如果没找到返回null，因为在HashMap中查找到的是Entry对象，返回的值是Entry对象的value值。

#### 遍历方法

1、keySet

for (String key : map. keySet) {

System.out.println("foreach循环进行遍历");

for (Integer key : map.keySet()) {

System.out.println("key:"+key+"对应的值为："+map.get(key));

}

}

2、iterator()

Iterator<Entry<Integer, String>> inte = map.entrySet().iterator();

while(inte.hasNext()){

Map.Entry<Integer, String> m = inte.next();

System.out.println("key:"+m.getKey()+"对应的值为："+m.getValue());

}

3、entrySet()

Set<Map.Entry<String, String>> entrySet = map.entrySet();

for (Entry<Integer, String> str : entrySet()) {

System.out.println("key:"+str.getKey()+"对应的值为："+str.getValue());

}

4、map.values()

Collection< String> collection = map.values();

for (String string : collection) {

System.out.println("map的值为：" + string);

}

Iterator，所有的集合类，都实现了Iterator接口，这是一个用于遍历集合中元素的接口，主要包含以下三种方法：

1.hasNext()是否还有下一个元素。

2.next()返回下一个元素。

3.remove()删除当前元素。

### Hashtable

Hashtable是遗留类，很多映射的常用功能与HashMap类似，不同的是它承自Dictionary类，并且是线程安全的，任一时间只有一个线程能写Hashtable，并发性不如ConcurrentHashMap，因为ConcurrentHashMap引入了分段锁。Hashtable不建议在新代码中使用，不需要线程安全的场合可以用HashMap替换，需要线程安全的场合可以用ConcurrentHashMap替换。

### LinkedHashMap

LinkedHashMap：LinkedHashMap是HashMap的一个子类，保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。

### TreeMap

TreeMap实现SortedMap接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator遍历TreeMap时，得到的记录是排过序的。如果使用排序的映射，建议使用TreeMap。在使用TreeMap时，key必须实现Comparable接口或者在构造TreeMap传入自定义的Comparator，否则会在运行时抛出java.lang.ClassCastException类型的异常。

# IO类

Java平台允许我们在内存中创建可复用的Java对象，但一般情况下，只有当JVM处于运行时，这些对象才可能存在，即，这些对象的生命周期不会比JVM的生命周期更长。但在现实应用中，就可能要求在JVM停止运行之后能够保存(持久化)指定的对象，并在将来重新读取被保存的对象。Java对象序列化就能够帮助我们实现该功能。

使用Java对象序列化，在保存对象时，会把其状态保存为一组字节，在未来，再将这些字节组装成对象。必须注意地是，对象序列化保存的是对象的”状态”，即它的成员变量。由此可知，对象序列化不会关注类中的静态变量。

### Serializable

1、在Java中，只要一个类实现了java.io.Serializable接口，那么它就可以被序列化。

2、通过ObjectOutputStream和ObjectInputStream对对象进行序列化及反序列化

3、虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致（就是 private static final long serialVersionUID）

4、序列化并不保存静态变量。

5、要想将父类对象也序列化，就需要让父类也实现Serializable 接口。

6、Transient 关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。

7、服务器端给客户端发送序列化对象数据，对象中有一些数据是敏感的，比如密码字符串等，希望对该密码字段在序列化时，进行加密，而客户端如果拥有解密的密钥，只有在客户端进行反序列化时，才可以对密码进行读取，这样可以一定程度保证序列化对象的数据安全。

### OutputStream、InputStream

#### 1、ByteArrayOutputStream序列化，ByteArrayInputStream反序列化：

ObjectOutputStream oos = new ObjectOutputStream(new ByteArrayOutputStream());

oos.writeObject(object);

byte[] resultByte = bos.toByteArray();

ObjectInputStream ois = new ObjectInputStream (new ByteArrayInputStream ());

ois.writeObject(object);

Object resultObject = ois.readObject();

#### 2、FileOutputStream序列化，FileInputStream反序列化：

//Write Obj to File

ObjectOutputStream oos = null;

try {

oos = new ObjectOutputStream(new FileOutputStream("tempFile"));

oos.writeObject(user);

} catch (IOException e) {

e.printStackTrace();

} finally {

IOUtils.closeQuietly(oos);

}

//Read Obj from File

File file = new File("tempFile");

ObjectInputStream ois = null;

try {

ois = new ObjectInputStream(new FileInputStream(file));

User newUser = (User) ois.readObject();

System.out.println(newUser);

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} finally {

IOUtils.closeQuietly(ois);

try {

FileUtils.forceDelete(file);

} catch (IOException e) {

e.printStackTrace();

}

}

# Java关键字

instanceof运算符 只被用于对象引用变量，检查左边的被测试对象 是不是 右边类或接口的 实例化。如果被测对象是null值，则测试结果总是false。

形象地：自身实例或子类实例 instanceof 自身类 返回true

例： String s=new String("javaisland");

System.out.println(s instanceof String); //true

Class类的isInstance(Object obj)方法，obj是被测试的对象，如果obj是调用这个方法的class或接口 的实例，则返回true。这个方法是instanceof运算符的动态等价。

形象地：自身类.class.isInstance(自身实例或子类实例) 返回true

例：String s=new String("javaisland");

System.out.println(String.class.isInstance(s)); //true

Class类的isAssignableFrom(Class cls)方法，如果调用这个方法的class或接口 与 参数cls表示的类或接口相同，或者是参数cls表示的类或接口的父类，则返回true。

形象地：自身类.class.isAssignableFrom(自身类或子类.class) 返回true

例：System.out.println(ArrayList.class.isAssignableFrom(Object.class)); //false

System.out.println(Object.class.isAssignableFrom(ArrayList.class)); //true