**redis**

查看集合大小：

Scard key 如果set是空或者key不存在，返回0

创建**Git**版本库：

git init

Git 相关命令：git status 查看结果

执行两步提交到仓库：

1. git add XXX(文件名)
2. git commit -m “explain”

版本回退： git log 查看历史修改

git checkout -- file可以丢弃工作区的修改

要设置用户名以及邮箱，否则公钥私钥不匹配，

git remote add origin [git@github.com:](mailto:git@github.com:) 路径.git

然后，本地库的所有内容远程推送到Github:（若更改，需要先commit）

git push -u origin master //远程库初始是空的就需要加-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令以后就不用了

Eclipse下使用PyDev搭建Python 开发环境：<http://www.cnblogs.com/wangcp-2014/p/5531130.html>

HashMap中threshold=capacity\*loadFactor ---故一般情况下threshold 会更小一些。当HashMap的size大于threshold时会执行resize()操作;

常规情况来讲就是: size<threshold<capacity

讨论容器的类型，而不是容器持有的类型。

**class** Food{}

**class** Fruit **extends** Food{}

**class** Apple **extends** Fruit{}

//定了上界，但不能维持list中是同样的类型，如 Apple、与orange 虽然都继承于Fruit 但是是不同的类，写入会失败

如 orange 是继承于 Fruit ，你填入一个 Apple()，是错误的，即使是可以的，但编译器是没有那么智能

List<? extends Fruit> flist=new ArrayList<>();

//!- Flist.add(new Apple()); Apple is Fruit or it’s childern (orange)?Xerror

//!- Flist.add(new Fruit()); Fruit is Fruit or it’s children (orange) ?X error

//!- Flist.add(new Object()); error

flist.add(**null**);// 可以但无意义~

List<? **extends** Fruit>flist=Arrays.*asList*(**new** Apple());

//!-List<? **extends** Fruit>flist=Arrays.*asList*(**new** Food());

List<? **super** Fruit>flist=Arrays.*asList*(**new** Apple()); 也可以

这样却可以

//定了下界，中都是Fruit的超类，存入Fruit总没错。

List<? **super** Fruit>flist=**new** ArrayList<>();

flist.add(**new** Apple());//ok-- Apple is Fruit or it’s parents

flist.add(**new** Fruit());//ok --Fruit is Fruit

//!-flist.add(**new** Food());//可能会有其他类转型为Food() 作为参数传入，-- Food is Fruit -error

--20170802

hashMap的modCount: 记录结构的变化次数，value更改不作数,添加则会++；

1.8中扩容（resize()）的一个改进(?)是扩容之后不用重新再计算key在hash桶中的位置了，（扩容为原来的2倍 capacity\*2），要么是原始位置index，要么是index= oldCap + index,

这两个不要弄混了：

---计算桶中的位置是index&(len-1)

---计算hashCode的值是：(h=k.hashCode())^(h>>>16) 保证高低位都参与到Hash的计算中，

<http://blog.csdn.net/u011240877/article/category/6447444>讲的挺详细的。

HashMap允许key，Value为null，同时他们都保存在第一个桶中,可查看源码得:

Put()方法： return putVal(hash(key),key,value,false,true)

而hash()方法：

return (key==null) ? 0: (h=key.hashCode())^(h>>>16)

**为什么哈希表的容量一定是2的整数次幂？**--很有道理啊

1、capacity为2的整数次幂的话， h& (len-1) 就相当于对len去模，提升了计算效率，

1. capacity为2的整数次幂，为偶数，因此capacity-1为奇数，可以保证h&(len-1)最后一位可能为1也可能为0， 可以保证散列的均匀性，

而如果 capacity 为奇数的话，很明显 capacity-1 为偶数，它的最后一位是 0，这样 h&(capacity-1) 的最后一位肯定为 0，即只能为偶数，这样任何 hash 值都只会被散列到数组的偶数下标位置上，这便浪费了近一半的空间。

---20170803

**红黑树：**

**特性**：

1. 节点非红即黑
2. 根为黑色
3. 所有叶子节点均为黑（java在原本子节点追加了NUll作为子节点）
4. 每个红色节点的两个子节点均为黑（红色节点不连续）
5. \*从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点（利用该性质作为平衡方法，以提高性能）

从特征来看，**插入**节点，可能会违背4、5两条性质；

若插入一个黑色节点，违背了第5条性质

插入节点的父亲节点为左子树或右子树的操作时相互对称的，下面只考虑插入位置的父亲节点为左子树。

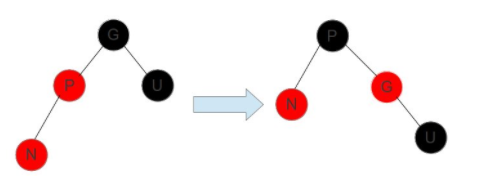
插入一个红色节点时，如在红色节点插入，则违背第4条。插入叶子节点

考虑直接**将节点染红**，再插入，去调整（父结点是否为红）

1. 父结点与叔节点均为红，祖父必为黑，--调整：祖父为红，父与叔为黑，如祖父节点处不能满足4，继续向上调整，直到符合（至根）。



2、父为红，叔为黑，祖父仍然为黑，这种情况---右旋一次，然后 P换色为黑，G换色为红；若插入节点为P的右子节点，先对N、P做一次左旋，不会蔓延到根层。



1. 父结点为黑，直接插入，不用调整。

根据红黑树的特性**删除**红色，不会破坏特性；若删除为黑-则需要调整来进行满足条件

平衡二叉出，若左右都有子树，取右子树最小的结点替换删除的节点，然后调整

2017-08-04

HashMap三个视图返回的迭代器都是fail-fast的：如果在迭代时使用非迭代方法修改了map的内容、结果，迭代器会报ConcurrentModificationExceptiond的错误

HashMap中关于红黑树的三个关键参数：

当冲突节点个数大于8时，现尝试table扩容，当table大于64是进行树化，树化时，不是每个冲突桶都树化，同样要达到 树化阈值 才可以

TREEIFY\_THRESHOLD -\*-一个桶的树化阈值，大于这个值进行树化 8

UNTREEIFY\_THRESHOLD -\*- 一个树的链表还原阈值 要比上面的值小，避免频繁转换，至少为6

MIN\_TREEIFY\_CAPACITY -\*-哈希表的最小树形化容量，表中的容量大于这个值时，表中的桶才进行树形化；否则桶内元素太多时会扩容，而不是树形化；

源码treeifyBin()方法中：

if(tab ==null || (n=tab.length)<MIN\_TREEIFY\_CAPACITY )

resize();

为避免进行扩容、树形化选择的冲突，这个值不能小于4\*TREEIFY\_THRESHOLD

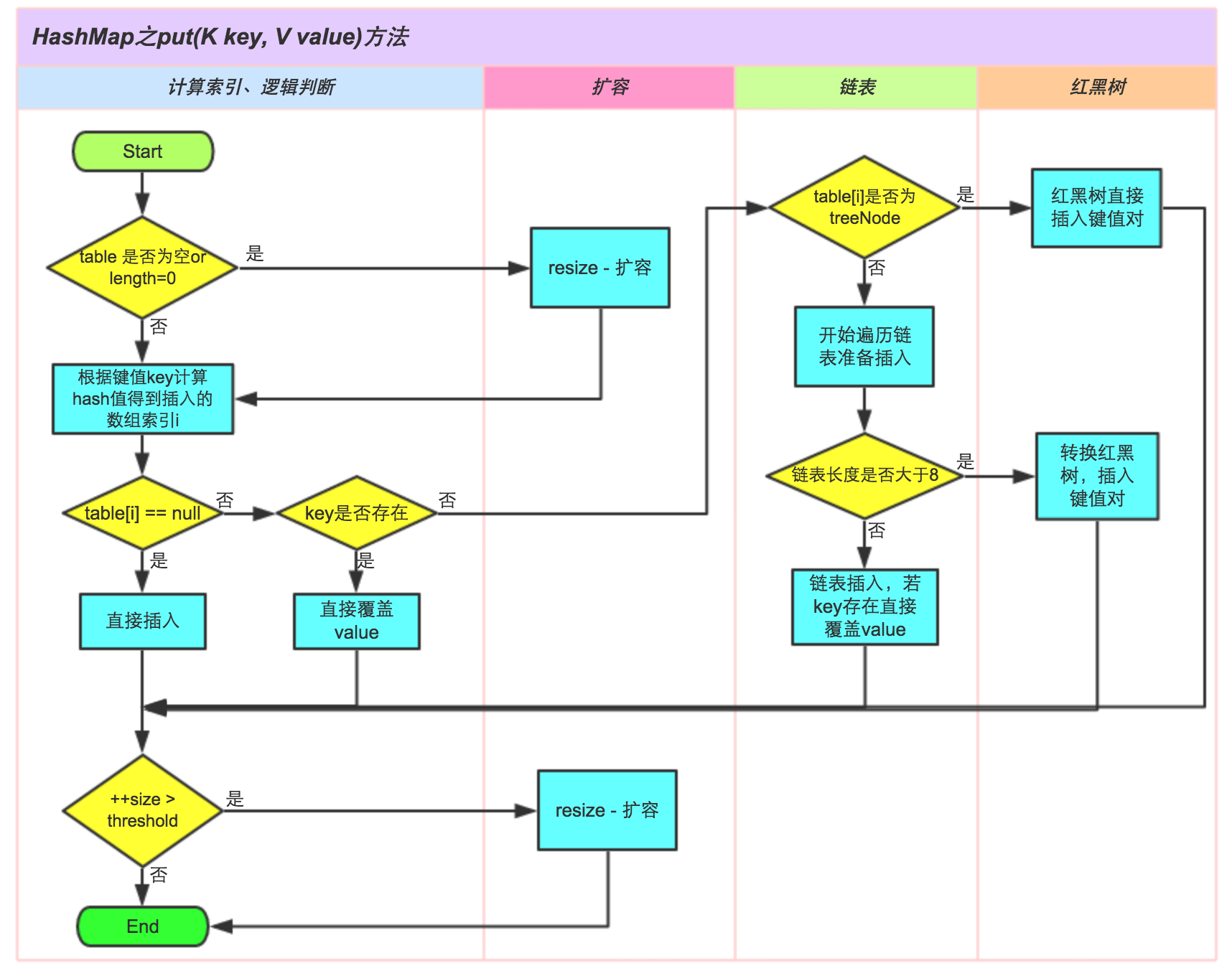
为64.

在转换成树时，两个哈希值相同，1.8是直接根据两个引用地址进行比较，

首先类相同，然后 System.identtityHashCode(XX),函数名：tieBreakOrder(a,b)--决胜局。

当然，在扩容时，还有种情况，如果当前桶中元素结构是红黑树，由于扩容后要重新洗牌，之前的节点会落在其他位置导致桶中的某些位置的元素小于UNTREEIFY\_THRESHOLD(6),需要把桶中的树形结构缩小或者直接还原为链表：方法:final void split()：

将符合要求（位置不变）的节点保存在一个链表中，不合要求的保存在另一个链表中，然后满足树化或去树化的条件进行 treefiy()或untreeify()的调用。 附一张图片比较详细的说明put方法的流程：



**compareable与comparator 的区别**:

Comparable-**自然排序**

>> Comparable在java.lang包下，是一个接口，内部只有一个方法：public int compareTo(); return >0,=0,<0

Comparable可以让实现它的类的对象进行比较，具体的比较规则是按照compareTo方法中的规则进行。这种顺序称为自然顺序.

1. Null不是一个类，也不是对象，因此在重写compareTo方法时应注意e.compareTo(null)的情况，即使e.equal(null)返回false，compareTo()方法也应该主动抛出一个空指针异常：NullPointerException.
2. Comparable实现类重写compareTo方法时一般要求e1.compareTo(e2) ==0 的结果要和e1.equals(e2)一致。这样将来使用SortedSet等根据类的自然排序进行排序的集合容器时可以保证保存数据的顺序和想像中一致。

实现了Compareable接口的List或数组可以使用Collections.sort()或Arrays.sort()方法进行排序；

只有实现了Comparable接口的对象才能够被直接用作SortedMap()的key,否则需要在外部实现Comparator排序规则。

Comparator **定制排序**

Comparator是在外部制定排序规则（1.8之前 匿名内部类等），然后作为排序策略参数传递给某些类，或者一些内部有序的集合（SortedSet，SortedMap等）

使用方式主要分三步：

1、创建一个 Comparator 接口的实现类，并赋值给一个对象

在 compare 方法中针对自定义类写排序规则

2、将 Comparator 对象作为参数传递给 排序类的某个方法

3、向排序类中添加 compare 方法中使用的自定义类

第二者定制排序使用的是一种策略模式，排序类中持有一个Comparator接口的引用，Comparator <? super K> Comparator;

从而我们可以传入各种自定义排序规则的Comparator实现类，对同样的类指定不同的排序策略。

2017-08-07