**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

Git 创建分支new\_master，然后切换到new\_master分支：

Git checkout -b new\_master

//-b表示创建并切换相当于：

Git branch new\_master //创建

Git checkout new\_master //切换

切换为master后:

合并：git merge new\_master.

然后删除分支：git branch –d new\_master. ---一个完整的流程。

git branch –D new\_master. 强行删除，用于分支还没有合并的情况（删除将丢失修改，使用-d 友情提示错误）。

**多人协助的git流程**：

1、可以试图用git push origin branch-name 推送自己的修改；

2、如果推送失败，则因为远程分支比你的分支更新，需要先使用git pull 试图合并；

3、如果合并有冲突，则解决冲突，并在本地提交（commit）；

4、没有冲突或者解决掉冲突后，再用git push origin brance-name推送就能成功

如果git pull提示“no tracking information”,则说明本地分支和远程分支的链接关系没有创建，用命令：

git branch –set-upstream branch-name origin/branch-name.

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。

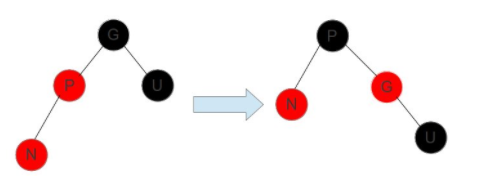
插入一个红色节点时，如在红色节点插入，则违背第4条。插入叶子节点

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

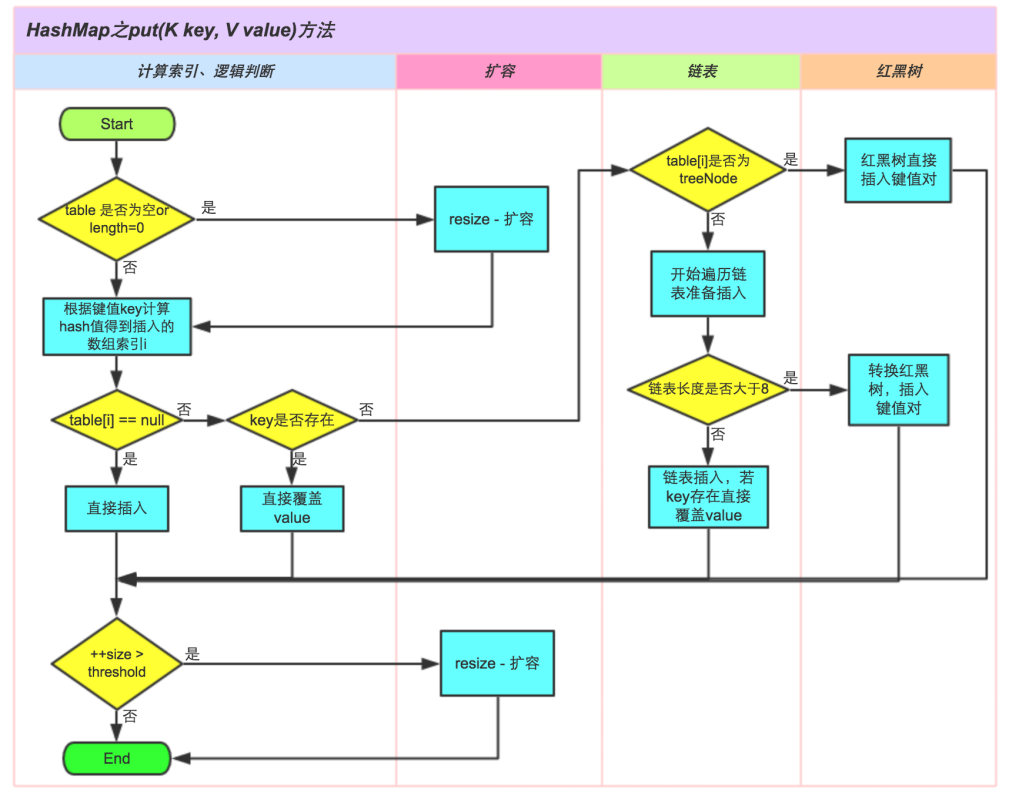
为64.

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

Sql中Find\_in\_set(str,strList)方法,假如字符串str在N个子链组成的字符串链表strList中，则返回值的范围在1到N之间。字符串列表是指一个被一些“,”符号分开的子链组成的字符串，含有str就符合要求

List: 'daodao,xiaohu,xiaoqin'

‘daodao’ in (list) --false

‘daodao’ find\_in\_set(list) --true.

select replace(translate('as,./;''[ldfhsi,,;，。wuqe？、ry',',./;''，。？、[',' '),' ') from dual; 替换标点符号。

310 最小高度树：

可以使用BFS遍历解决，考虑清楚情况，从叶子到叶子，对每个叶子都要尽量取到均匀，才能得到最小的高度。--最长路径的中间点。树中（连通图）若有多个最长路径，中间节点必相交（考虑为什么）

另外一种使用tree dp ~~

2017-08-09

有些hard题目并不是很难，不要被吓到了，只是你的思路有些问题罢了，知道了就挺简单的。

2017-08-11

磁盘读写原理：

磁盘上的数据用一个三维地址唯一标识（柱面号，盘面号，块号(磁道上的盘块)）

读/写数据需要三个步骤：

1. 移动臂根据柱面号使磁头移动到所需要的柱面--定位或查找
2. 根据盘面号来获取指定盘面上的磁道。
3. 盘面确定后，盘片开始旋转，将指定块号的磁道号移动至磁头下。

磁盘读取是以盘块（block）为基本单位的。位于同一盘块中所有数据都能被一次性全部读取出来，而磁盘IO代价主要花费在了查找时间上-因此应尽量将相关信息存放在同一盘块，同一磁道上；或至少放在同一柱面或者相邻柱面上，从而在读/写信息时尽量减少磁头来回移动的次数，避免过多的查找时间。

所以，在大规模数据存储方面，大量数据存储在外存磁盘中，而在外存磁盘中读取/写入块(block)中的数据时，首先需要定位到磁盘中的块，为有效的查找磁盘中的数据，需要一种合理高效的外存数据结构： B-tree 及相关变种结构B+ tree 和B\* tree。

B-树：（m阶-按阶定义）

1. 每个节点最多有m个子节点（m-1个关键字）
2. 内（中间）节点至少有ceil(m/2)个子节点
3. 关键字个数： ceil(m/2)-1 <=n<=m-1, 递增排序

插入可能使关键字> m-1 分裂，将中间节点给父结点，如父结点满了，再分裂一直向上

删除使 关键字 < ceil(m/2)-1 合并，找兄弟（单纯的数目上的不变），不行了再找父亲，父亲下来一个key并与两个子结点合并（节点个数为 2\*min），而父亲减少一个节点，如父亲不和要求，继续向上走-最坏情况会走到根部，然后高度减少1。

B+ tree： 将叶子节点连起来，中间节点只存储索引，无关键字。因此，查询时间完全均衡-更加稳定，内部节点相对B-tree要更小，从而可以更快的找到盘块（减少IO读写次数）一次IO可以查找更多的关键字。

B\*-tree 是B+-tree的变体，在B+-tree的基础上，非根和非叶子节点再增加指向兄弟的指针~~~

Mysql中底层存储使用B+ 树实现，内存中B+树没有优势，但是到磁盘中优势就显露出来了。

2017-08-14

通常，数组与泛型不能很好的结合，不能实例化具有参数化类型的数组：

Peel<Banana>[] peels =new Peel<Banana>[10];//Illegal.

--数组必须知道它们所持有的类型，以强制保证类型安全。

2017-08-15

若要在指定位置添加某一列Mysql中没有before ,在第一列添加，使用first，其余指定位置用after,插入、修改列名称都要指定数据类型，

alter table difu\_member\_detail add column `member\_name` char(32) after `member\_id`;

#修改列名称

alter table difu\_member\_detail change COLUMN `hosiptal` `hospital\_level` char(32);

java字符串去除空格:

1、string.trim():去掉首尾空格

2、string.replaceAll(“ ”,””)去所有空格，包括首尾

3、string.replaceAll([\\s+](file:///\\s+),””);去掉大部分空白字符，不限于空格.

**Map的1.8几个方法**：

getOrDefault(Object key,V defaultValue) :若 get(key) 为空，返回defaultValue,否则返回 get(key)

putIfAbsent(K key,V value):如果包含该键值对，不插入，否则put(key,value).

函数式编程： V computeIfAbsent(K key,Function<? super K, ?extends V> mappingFunction---如key对应的value为null，计算获得新value.

map.merge(key, msg, String::concat)

linkedHashMap：存储顺序使用LRU（最近最少使用），访问过的就放在最尾部，所以没访问过的就在队头（可看作是需要删除的）。

2017-08-17

<script charset="Shift\_JIS" src="http://chabudai.sakura.ne.jp/blogparts/honehoneclock/honehone\_clock\_wh.js"></script>

<script charset="Shift\_JIS" src="http://chabudai.sakura.ne.jp/blogparts/honehoneclock/honehone\_clock\_tr.js"></script>

不知名姓的脚本

java容器有一种保护机制,防止多个进程同时修改同一个容器的内容;容器类库采用快速报错(fail-fast)机制,会探查容器上的任何除了你的进程所进行的操作以外的所有变化,一旦发现其他进程修改了容器,会抛出ConcurrentModificationException异常. fail-fast:不用复杂的算法在事后来检查问题。--一个经典的错误：使用迭代器，在迭代过程中，添加/删除一个元素。

mongoDB查询非空数组：

假如medical是个数组：

1、db.cases.find({'medical':{$gt:[]}})

2、db.cases.find({'medical.0':{$exists:true}})

3、db.cases.find({'medical': {$elemMatch:{$ne:null}}}, {medical:1})　　重点： $elemMatch:{$ne:null}

2017-08-18

XAMPP软件(Apache+Mysql+PHP + PERL)--建站集成软件包，知道有这么一类东西。

coalesce –合并，

装饰模式用涂涂料来理解更合适，不同颜色具备不同的功能。

对于Serializable对象，对象完全以它存储的二进制为基础来构造，而不是调用构造器；

而对于Externalizable对象，所有普通的默认构造器都会被调用（包括在字段定义时的初始化，）然后调用readExternal()；所有的默认构造器都会被调用，才能使Externalizable对象产生正确的行为。

从而Externalizable接口对序列化进行控制，如，考虑安全问题、对象被还原后，子对象需要重新创建，从而不必将该子对象进行序列化~功能。

然后，Externalizable的替代方法：使用Serializable接口，然后添加 readObject()和writeObject()方法，从而一旦被序列化或反序列化，会自动调用这两个方法，而不是默认的序列化机制。

transient关键字：在对序列化进行控制时，有些子对象不想让Java的序列化机制自动保存和恢复，即使设置为private ，一经序列化处理，通过读取文件或者拦截网络传输的方式也可以访问到它们。

transient(瞬时)，关闭序列化；另外，由于Exernalizable对象在默认情况下不保存它们的任何字段，需要显式调用；所以transient关键字只和Serializable对象一起使用。

假如想序列化static 值，必须自己手动去实现。-使用serializeStaticState()和deserializeStaticState() 两个static方法的。

XML操作，在TIJ4中用到的是XOM.挺简单的。

使用编辑器，文件的位置并不会那么明确的显示，最好写绝对路径.

Perferences-API 用于存储和读取用户的偏好，以及程序配置项的设置。

2017-08-21

枚举类型:enum都是继承于java.lang.Enum类,对于clone方法，直接抛出异常CloneNotSupportException，即Enum类型不可clone;- 但 EnumSet有该方法。

没有finalize() 方法{设为protected属性}

同时实现了deserialization()反序列化，使用readObject();

java.lang.Object 下的finalize() 方法：java不保证任何线程会调用该方法，但是保证，在调用该方法的时候不会持有任何用户可见的同步锁；

EnumSet中，尝试假如插入null元素的话会抛异常；但是去检查是否存在或者移除null元素，确实允许的；是非同步~

查看源码： EnumSet<E extends Enum<E>> ~WTF~:

详细解析见 ：<http://madbean.com/2004/mb2004-3/>

首先在集合中，List<T>表示 T类型的容器，即该容器中放置的变量类型是 T.但一般类呢：

解析： public class Foo<T>{ //(A)

}

Foo<String> f1=…; //(B)

Foo<Integer> f2=…; //(C)

Foo<subClassOfFoo extends Foo<SubClassOfFoo>> means:

abstract class Foo<SubClassOfFoo extends Foo<SubClassOfFoo>>

{

/\*\* subclasses are forced to return themselves from this method \*/

public abstract SubClassOfFoo subclassAwareDeepCopy();

}

class Bar extends Foo<Bar> {

public Bar subclassAwareDeepCopy() {

Bar b = new Bar();

// ...

return b;

}

}

Bar b = new Bar();

Foo<Bar> f = b;

Bar b2 = b.subclassAwareDeepCopy();

Bar b3 = f.subclassAwareDeepCopy(); // no need to cast, return type is Bar

a. Any subclass of Foo must supply a type argument to Foo(inner <SubClassOfFoo>);

b. That type argument must actually be a subclass of Foo;

c. Subclasses of Foo follow the idiom that the type argument they supply to Foo is themselves;

d. Foo has a method that returns SubClassOfFoo. Combined with the above idiom,this allow Foo to formulate a contract that says “any subclass of me must implements subclassAwareDeeoCopy() and they must declare that it returns **that** actual subclass”.

类型参数允许类的使用者在调用该类的方法时避免手动转型：

Class<Date> c1 = Date.class;

Class c2 = Date.class;

Date d1 = c1.newInstance(); // new style, no potential ClassCastException

Date d2 = (Date) c2.newInstance(); // old style

解释为： the Class instance for the class T

—java.lang.Class<T> 也在使用， Class却并不是Collection.

Enum<E extends Enum<E>> :

you can write code like the following that:

a. dose not need to case ;

b. you can use method in Enum in terms of the concrete enum subclass;

Rank r = Rank.ACE;

Suit s = Suit.HEART;

r.compareTo(s); // syntax error, argument must be of type Rank

Rank z = Enum.valueOf(Rank.class, "TWO");

EnumSet 的of方法一共有6个，TIJ4上说是5个，现在应是改了， 1-5 分别重载，添加n个元素； 5个以上，使用可变长度参数列表，

**public** **static** <E **extends** Enum<E>> EnumSet<E> of(E first, E... rest){

} –为提高一点儿效率，已达到了丧心病狂的地步了。

有多种类型的enum，而且他们之间存在互操作的情况下，我们可以用EnumMap实现多路并发（multiple dispatching）,

在面向对象的程序设计中，不同的行为与不同的类关联。而通过常量相关的方法，每个enum实例可以具备自己独特的行为，像一个独特的类， 实例还可以被当做类来使用， 以体现出多态的行为，

虽然 enum实例与类有些相似之处，但是并不是类，每一个enum元素都是XXX类型的static final 实例。--又说回来，虽然enum有些限制，但一般而言，还是可以将其看作是类。

枚举类型十分适合创建**状态机**，

取绝对值的时候，溢出了就不准了，怎么办，但那时已超出了Integer的表示范围~

2017-08-22