添加一个分支

**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 Orange 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

如果存了 Orange，再存储 Apple就会有问题，这两个是没有关系的类。。

//!- 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，其中都是Fruit的超类，存入Fruit总没错，存Fruit的子类就更没问题了，

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**：

放弃追求完全平衡，追求大致平衡，在于与平衡二叉树相差不大的情况下保证每次插入最多只需要三层旋转即达到平衡，

而平衡二叉树，每次插入节点后旋转次数不能预知，

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（红红的情况-换做是插入祖父节点G），直到符合。--



2、父为红，叔为黑，祖父仍然为黑，这种情况---对G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P做一次左旋，化作左子节点，，不会蔓延到根层。

1. 父结点为黑，直接插入，不用调整XXX？--不追求完全平衡了。

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

1、按二叉树删除。

被删节点没有子节点。

被删节点只有一个子节点-替换，

待删节点的后继节点必然只有一个子节点，

2、旋转，着色修正为二叉树。

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

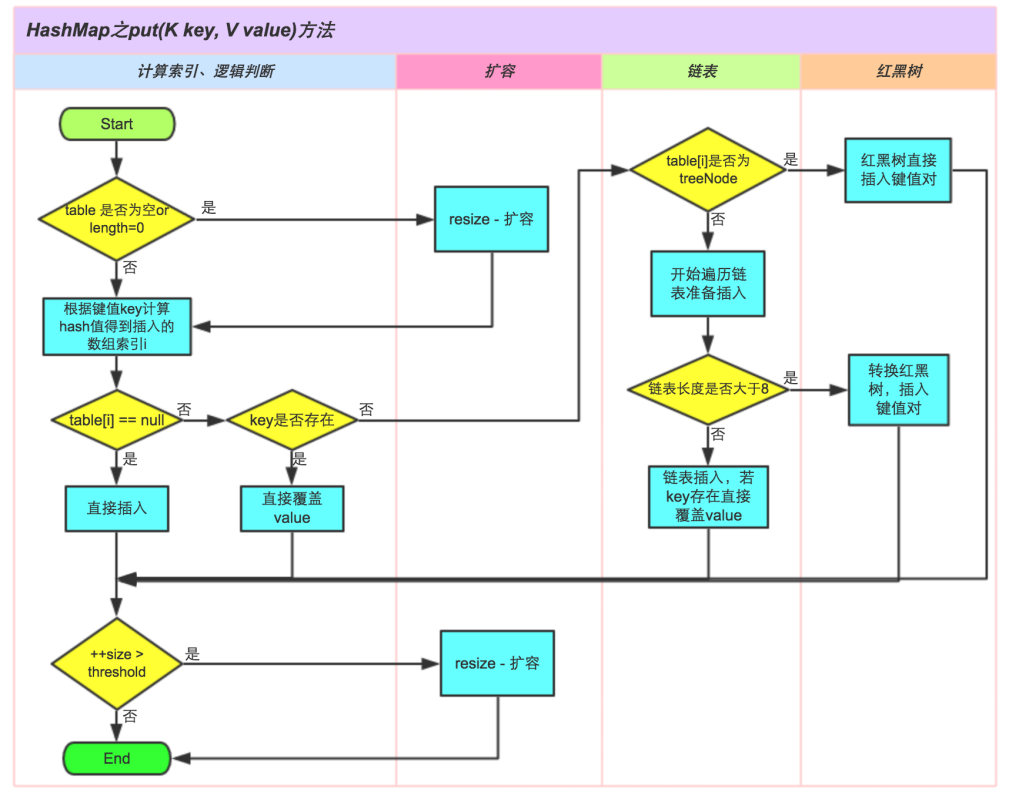
为64.

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

java的注解:至少要知道

注解的元素在使用时表现为: name-value对的形式，并需要置于@XXX声明之后的括号里面，XXX为定义的名称：如：

**public** **@interface** UseCase {

**public** **int** id();

**public** String description();

}

XXX 即表示 UseCase

javaSE5目前支持三种标准注解：

@Override覆盖超类中的方法

@Desprecated 如使用了注解为它的元素，则编译器会有警告信息(表示已舍弃?不建议使用该类)

@SuppressWarnings，关闭不当的编译器警告信息，屏蔽警告？~

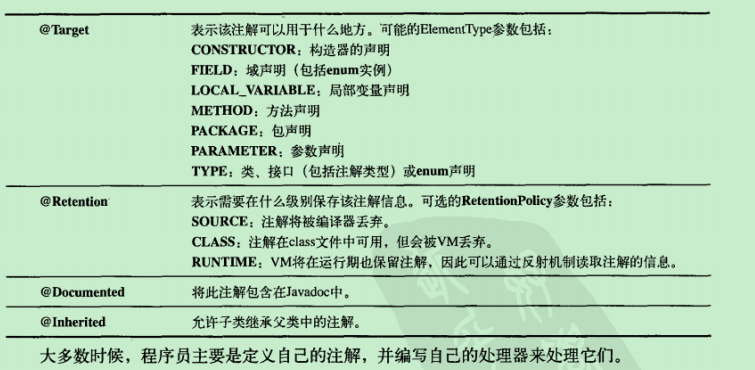
以及四种元注解:负责注解其他的注解。

@Target 表该注解可用于什么地方，可选参数：7个

@Retention 表需要在什么级别保存该注解信息，可选参数：3个

@Documented 将该注解包含在Javadoc中

@Inherited 允许子类继承父类中的注解。



2017-08-23

与别人沟通注意歧义，例如占比你写一个小数，就不如一个百分比格式的数字合适。

2017-08-24

合理使用MongoDB中的语句，来简化查询范围，也可节省一些时间：对于脚本的执行。如查询caseNo以 DCW-2017开头的数据条目：

db.case.find({‘caseNo’:/^DCW-2017}),MongoDB的原生语言要合理使用，用limit()查询前多少条，skip() 跳过前多少条。

2017-08-25

二叉树的操作，记着用递归去实现，更合适，

如**求两个节点的最低公共父结点**，两种思路：

1. 通过中序遍历和后序遍历来实现：两种遍历的特性，交集可得~

左右根、左根右，如果是互为父子，另行考虑，直接返回后序的父节点。

1. 递归，从树根开始，分情况(1)节点A和B都在左子树中，(2) 都在右子树中，(3) 一左一右； 分治法。

TreeNode<T> left=commonNode(node1,node2,root.left);

//若left为空，表示node1，node2都不在 左子树中

TreeNode<T> right=commonNode(node1,node2,root.right);

//若right==null,说明node1，node2 都不在右子树中。

if(left !=null &&right !=null)

Return root;

Return left ==null ? right : left;

又如查找第k层节点的个数，可以使用队列，在每一层末尾设一个标识，

同时也可以使用递归：

1. 如二叉树为空或K<0,返回0；
2. K等于0 则返回一个根
3. K大于0， 返回左子树中第K-1层节点的个数，加上右子树中第K-1层节点的个数。--OK~

binarySearch()方法返回的是下标，如果没有找到，返回负数，-1\*应插入的位置从1开始。

你要会很熟练的写出二分查找，有几种变形，竟然自己糊涂了。

如： 只需稍作修改即可的情况：

查找第一个target（数组有重复数据）

查找第一个大于target的下标

二分查找：

while(st<ed){

int mid

if()

st=mid+1 (mid)

else if()

ed=mid (mid-1)

} //结果是 st==ed

while(st<=ed){

int mid

if()

st=mid+1

else if()

ed=mid-1;

}//结果得 st= ed+1 注意不同点。

合理使用短路原理：

// 矩阵为空~自己去体会。

if(matrix ==null || matrix.length==0 ||matrix[0]==null ||matrix[0].length==0)

return false;

2017-08-28

某些东西，手动输入的话不会很不规则么；

注意NIO、多看看。

2017-08-29

经验之谈，写日志是有多重要：免不了会有错误发生：网络传输、人为操作失误等等。这次你传输数据的事，真是醉了~ 连错了2次传输type的事情，另一次是网络问题，没有记录错误情况导致。摸不到头脑，另外，多线程对文件的读写也是一个问题，注意一点。

在将字符串加载为json格式时一个错误：

json.loads(text,encoding='utf8') 报Unexpected UTF-8 BOM (decode using utf-8-sig)错误，将encoding改为'utf-8-sig'仍然报错。

原因分析：

text包含BOM字符

解决方案：

将BOM头去掉，代码如下：

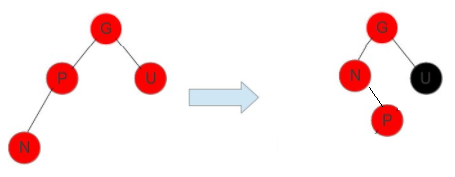
if text.startswith(u'\ufeff'):

            text = text.encode('utf8')[3:].decode('utf8')

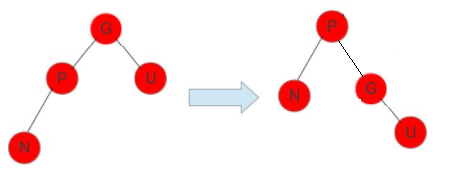
将json数据导入到mongo中，一个语句save(json\_name) 就行了。

注意左旋右旋的含义：注意是对谁旋转，左旋同理，

对P进行右旋：顺时针旋（绕着左子节点） -P点无法左旋，左旋（绕着右子节点）



对G进行右旋：从右向左旋，



红黑树，删除结点考虑的是一般二叉查找树替换后，补充被替换的节点的位置的元素，只把值赋过去，原始颜色不变，替代的表示N（而N不管是红还是黑），或者为待删节点的后继（左右子节点都存在），或者被删除的元素本身（左右子结点至少有一个不为空）

2017-08-30

pymongo.errors.CursorNotFound: Cursor not found：在取得游标时间太长之后，，在cursor上长时间没有进行操作，引发cursor在mongodb服务端超时

Python可以使用线程将一个记录器绑定，在调用记录器时，模块会根据线程ID从主类中找到对应的记录器并返回，如果没有找到，就调用设置方法来初始化一个新的记录器并放入字典中缓存起来，因此日志模块是唯一的，但中间有一个字典，这是重点；因为日志模块要被其他模块调用，是日志的唯一出口，其主类会做成单例（singleton）以避免重复初始化可以设为单例。

Java：可变参数列表不依赖于自动包装机制，而实际使用的是基本类型；

在单一参数列表中，自动包装机制会有选择的将int 提升成为Integer。

2017-08-31

不能因为在类中某个对象的引用是private，就任务其他对象无法拥有该对象的public 引用--别名机制；

final修饰符，可用于匿名内部类，之前有闭包提到过。~回去翻一下；

关于初始化顺序问题：先初始化静态成员，静态方法被调用才会使用到，然后成员变量，接下来是自身的构造器，构造器第一行调用父类构造器（如果有的话）成员变量的初始化必然在构造器调用之前，以防止在构造器中使用到，但会在父类构造器之后调用，因为可能会用到父类的一些成员变量；这三个层次，在每一层中就按代码顺序初始化即可。

因此顺序为：1、进入XXX之后

1. 的总结（每一章的总结都值得好好读）：设计一个系统时，目标应该能找到或创建某些类，其中每个类都有具体的用途，而且既不会太大（包含太多的功能而难以复用），也不会太小(不添加其他功能就无法使用)；如果设计过于复杂，通过将现有类拆分为更小的部分而添加更多的对象，通常会有所帮助。

在设计系统时，需要知道程序开发是增量过程，组合和继承可以执行这种实验的最基本的两种工具。

**注意：**

1. **通用的准则： 用继承表达行为间的差异，并用字段表达状态上的变化**

工厂方法客户拿着需求去工厂中要对象，客户不知道具体实现（不会直接调用构造器）

内部类： 在拥有外部类对象之前是不能创建内部类对象的，除非是嵌套类（静态内部类）；因此要新建一个外部类对象，然后

Outer.Inner oin=.new Inner();获得内部类对象。

2017-09-01

Python .join()方法可以连接元组、list，但其中的元素必须是string类型。

在某一作用域的内部类，假如在if语句中，并不是说，只有满足条件才创建该内部类，其实是与其他类一起被编译的，但是在作用域外仍然不可用。

Jdk1.8,对于匿名内部类，可以不用加final关键字了，当在类中你进行非法操作（修改）时，提示你错误13：从内部类引用的本地变量必须是最终变量或实际上的最终变量，然后你将其增加修饰符final-->接下来再次报错14： 无法为最终变量分配值；

但是如果你在匿名内部类中不修改，则不会有错，即便不添加final修饰。

嵌套类：不需要与外围类对象之间有联系，声明为static：

考虑与普通内部类的一个区别：普通内部类的字段和方法，只能用在类外部层上，故普通的内部类不能有static数据和static字段，当然也不能包含嵌套类，但这对嵌套类确没有限制。

接口中是可以定义类的，1.5就可以，现在1.8可以定义默认方法了。

内部类的主要用处：多重继承；每个内部类都能独立的继承自一个（接口的）实现，所以无论外围类是否已经继承了（接口的）实现，对于内部类都没有影响，如果是继承自类，则不可多重继承。

2017-09-06

内部类的继承，有些奇特的语法，被继承的内部类如果带参构造器，继承它的类构造器需要先获得该类的外围类，以及父类的参数，即两个参数：

ExportClass(Outer out,int param){

Out.super(param);

}

通过外部类继承，内部类可以被覆盖（重写）么？ 并没有关系，两个同名的内部类是完全独立的两个实体，各自在自己的命名空间中。

局部内部类（在方法等域中声明），而不使用匿名内部类的理由就是：

1. 需要一个已命名的构造器，或者需要重载构造器，匿名内部类只能用于实例初始化。
2. 需要不止一个该内部类的对象（后者只能有一个）

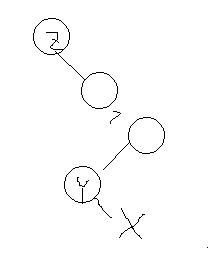
C++类库并没有其容器的任何公共基类--容器中之间所有的共性都是通过迭代器来达成的，Javc也遵循此规则： 即迭代器不是用Collection来表示容器的共性，但是这两种方法绑定到一起，因为实现Collection就意味着必须实现iterator()方法。

虽然forEach语句可以用于数组也可以用于其他任何，但是这并不意味着数组也是一个Iterable,而且自动包装也不会自动发生。

2017-09-07

在红黑树删除操作中，不动颜色，先删节点Z的值，以后继代替之，一般来说分三种情况：

1. 只有左或者只有右，令Y=Z，不用赋值（y的值赋给z），直接删除，X为Y的子节点；
2. 左右都有，且Z的后继为右子树的某一左子节点： X为空，



1. 左右都有，但Y为Z的右子节点，则X为Y的右子节点，为Z的右孙子节点；

确定X后，判断Y的颜色，然后决定是否对树进行调整，入口为X--这是理解其他删除操作的前提。--不管X的颜色是什么，总会修改为符合条件的树~

删除也会向上蔓延：

X的父为黑、兄弟为黑、兄弟的两个儿子也为黑~~包围了。将兄弟设为红，然后继续将X的父结点传入修改入口 X.parent(为黑)，同样要调整。

2017-09-08

在插入mysql数据时，发现一个问题， 2017-08 --它竟然被mysql 擅自改成了2009： 当成是减号而且操作了。你加个引号就行了。0.0

LinkedHashSet使用accessOrder参数（boolean）表示两种存储顺序：

1. 插入顺序，每次插到尾部accessOrder=false（default），
2. 存取顺序，LRU(之前你是有了解过的)，将访问过的节点放到尾部；-因此需要重写父类HashMap的get()方法,get之后要调整。（双链表）

其中有一个removeEldestEntry() 方法，默认返回为false、遇到这种情况看，应该考虑到是要被重写的，也确实是这样：如果使用LinkedHashmap实现LRU算法，就要覆盖该方法（再次印证，默认存储顺序是插入顺序）

一般的重写方法：如果设定的内存已满，则返回true，再次put Entry时，在调用的addEntry方法中就会将近期最少使用的节点删除掉，（head之后的节点）

TreeMap是对key进行排序，排序和定位依赖于比较器或者重写Comparable 接口，不同点一：因此也不需要key复写hashCode()方法和equals()方法，就可以排除掉重复的key，而HashMap的key则需要通过重写hashCode方法和equals方法来确保没有重复的key.

不同点二：TreeMap的查询、插入和删除效率没有HashMap高，一般只有对key排序才会用到； 不同点三：TreeMap的key不能为空null，而HashMap确可以为null。TreeMap表现为源码中：

**if** (key == **null**)

**throw** **new** NullPointerException();

另外，由于有序性，也会有许多其他的方法，如 floorKey(K key)、ceilKey(K key)等等。

2017-09-11

计数排序，应用并不是特别广泛；但效率还是比较高的，思想是：对于x，找到已有的数组中确定小于x的个数，那么排序后x的位置也就明确了。

2017-09-13

好多问题记着化简、减小规模、递归或者动态规划~~ 求字符的所有组合（用二进制表示也可以0、1取即可。）但也可以用递归、分解成两部分：

重要的是思考过程：

N个字符组合成长度为m的组合：求n-1个字符串中长度为m-1的组合（包含第一个字符），与求n-1个字符中长度为m的组合（不包含第一个字符）；都可用递归解，

String 的**public** **native** String intern();方法：

会将字符串放入常量池中，如果常量池中已经有该字符，会将其引用返回，即不会新建对象；若常量池中没有，将该对象放入常量池(aa、bb赋值交换顺序输出不变)： 原本以上两种情况下都会 == 都会为false。

String bb="asdf";

String aa=**new** String("asdf").intern();

System.***out***.println(aa.equals(bb));//true

System.***out***.println(aa==bb);//true.

数组中只出现一次的数字：若其他都出现三次，这个出现1次，对每一位分别做统计，出现次数不为3，则保留--必然属于出现一次的数字，取出即可。

负数的补码啊：正数二进制取反加1啊，想什么啊，这都忘了？

20170914

异常一般来讲有两种模型：

终止模型、与 恢复模型：更多的转向了第一种，并忽略了恢复行为；

主要是因为，恢复所导致的耦合，恢复性从处理程序必然要了解异常抛出的地点，从而要依赖于抛出位置的非通用性代码，进而增加了代码编写和维护的困难。

System.out 输出可以通过System.setOut()方法被重定向到文件中；

但System.err 却不可以重定向。

Throwable是所有异常类的父类。

RuntimeException代表的是编程错误：

1. 无法预料的错误，如从控制范围外传递进来null的引用
2. 作为程序员，应该在代码中进行检查的错误。

finally块的内容总会被执行，即使try/catch块中有return 也会在调用return的最后一步执行---

\*\*\*甚至在异常没有被当前的异常处理程序捕获的情况下，异常处理机制也会在跳到更高一层的异常处理程序之前执行finally子句！

同样的，由于finally的机制，使得有可能丢失异常、如在finally中抛出另一种异常、或在finally中返回等。

异常的限制，当覆盖方法时，只能抛出在基类方法的异常说明里列出的异常：当基类使用的代码应用到其派生类对象的时候，一样能够工作，异常也不例外。（覆盖方法可以缩小范围，但不可引入未知异常）；归纳为：在继承和覆盖的过程中，某特定方法的“异常说明的接口”不是变大了而是变小了--这与类接口在继承时的情况完全相反。

若派生类在扩展基类的同时又实现了某个接口，而基类和接口中都有同一个方法f() 这样，这两个方法所抛出的异常就必然要相同；否则的话，在使用基类时就不能判断是否捕获了正确的异常。当然如果接口中定义的方法不是来自于基类，则无此限制；

异常限制对构造器不起作用。派生类的构造器可以抛出任何异常，而不必理会基类构造器抛出的异常，然而因为基类构造器需要被调用，派生类构造器的异常说明必须包含激烈构造器的异常说明。

如果异常发生了，所有东西都能被正确清理吗？构造器-有些问题；

基本规则：在创建需要清理的对象之后，立即进入一个try-finally语句块；

在捕获异常的时候，如果发现，有的语句永远不会执行，则编译器会报错。

Sql模糊查询：

%代表任意多个或0个字符；

\_代表任意单个字符；

[] 表示括号内所列字符中的一个：[A-Z] :表示匹配一个字符，该字符在A至Z之间；

[^ ]表示不在括号所列之内的单个字符；

查询内容包含通配符时：（如查找 有 %、\_、[）把特殊字符用[]括起来就可以正常查询了：

function sqlencode(str)  
str=replace(str,"[","[[]") '此句一定要在最前  
str=replace(str,"\_","[\_]")  
str=replace(str,"%","[%]")  
sqlencode=str  
end function

20170915

**Jdk8 取消永久代**的几个原因：

字符串存在永久代中，容易出现性能问题和内存溢出。

类及方法的信息比较难以确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大容易出现老年代溢出，

永久代会为GC带来不必要的复杂度，并且回收效率偏低。

Oracle可能会将HotSpot与JRockit合二为一。

应聘者是否考虑了边界条件、特殊输入（如NULL指针，空字符串等）及错误处理。

面试时遇到难题，有三种方法分析、解决复杂问题：

> 画图使抽象问题形象化，

> 举例使抽象问题具体化

> 分解使复杂问题简单化。

Mysql实现行号自增：设初值为0、然后自增即可：

Select ID,(@rowNum: =@rowNum +1) as rowNo from a,

(Select (@rowNum :=0)) b

合并两个数组（包括字符串）时，如果从前往后复制每个数字，需要重复移动数字（字符多次），可以考虑从后向前复制，从而减少移动次数提高效率。--逆向。

20170918

快排序：

第一种解法（一般方法）注意，不要找到两个再交换，前面找到一个就换到后面、后面找到一个换到前面： 一般方法，设两个指针

**if**(first>=last) **return**;

**int** pivot=data[first];

**int** st=first,ed=last;

**while**(st<ed){

**while**(st<ed && data[ed]>=pivot) ed--;

data[st]=data[ed]; //以第一个作为枢值，可以考虑随机选定

**while**(st<ed && data[st]<=pivot) st++;

data[ed]=data[st];

}

data[st]=pivot;

第二种：只设一个指针，先随机选一个下标与尾部交换，然后和尾部比较，index< last 不会遍历到 last，small记录小于 data[last]的元素个数，这样好像是不稳当排序了（本来就不稳定）

Random rand=**new** Random();

**int** index= rand.nextInt(last-first+1)+first;

System.***out***.println(index);

*swap*(data,index,last);

**int** small=first-1;

**for**(index=first;index<last;index++){

**if**(data[index]<=data[last]){

++small;

**if**(small !=index)

*swap*(data,index,small);

}

}

small++;

*swap*(data,small,last);

顺序数组的旋转，求最小数字，循环右移n位（可能为0）；用二分查找、然后有漏洞，解决、剑指offer；

养**成代码规范化的习**惯：

清晰的书写、

清晰的布局、 ====> 才能算写出规范的代码。

合理的命名、

考虑**代码的完整性**：

功能测试、

边界测试、 ===> 完整的代码，设计测试用例，非法输入等；

负面测试、

训练： 实现 double Power(double base,int exponent), 即base的 exponent次方；

计算机表示的小数（double和float）都有误差--

打印所有的n位数值竟然可以用数字排列（递归）来解，还是有差别的；

在O(1)时间内删除指定节点（给定头结点）：

1. 删除结点是头、
2. 删除结点是尾、
3. 删除结点为null
4. 链表本身为空
5. 链表中这只有一个节点，头结点要置空

给定节点是否存在于链表中、也要考虑，要有这个意识去考虑。

20170919

代**码的鲁棒性**： -如果一段代码写出来后有多处崩溃的潜在风险，那凭什么通过面试。

防御性编程：平时在写代码时养成防御式的好习惯，在方法的入口判断输入是否有效并对各种无效输入做好相应处理；

如果输入是字符串->为空怎么处理

输入为数组 -> 数组为空

输入为整数 -> 负数同样处理？

数组中查找第k个元素，k> array.length 怎么处理、k为0 怎么处理？

...

遇到问题**先分析与设计**，不要急于动手写代码，这段时间不应该比编代码的时间短

多用递归：如合并两个有序链表，递归真的太简单了，养成下意识的用递归解。

解决问题前，先谈思路：否则面试官会感觉你很差。

**单元测试JUnit**

一个测试方法主要包括了三部分：

1. Setup
2. 执行操作
3. 验证结果

@Before：

如果一个方法被@Before修饰过了，那么在每个测试方法调用前，这个方法都会得到调用，

@After:

每个方法运行结束后，会得到运行方法，如在before处打开文件、在after处关闭文件，这些都是自动执行的。

类似的@BeforeClass 和@AfterClass，有一点注意：被这两个annotation修饰的方法必须是静态的；

第三步的验证结果方法是通过一下assert方法来完成的，如

assertEquals(expected,actual)

assertEquals(expected,actual,tolerance) --有误差容忍

assertTrue(boolean condition)

assertNull(Object obj)

...

fail() -使测试方法失败

@Ignore 忽略一些方法（可能正式代码还没有实现等原因）等等

JUnit使用最佳实践：~~what

1. 测试方法必须使用@Test进行修饰
2. 测试方法必须使用public void 进行修饰，不能带任何参数

测试失败有两种情况：

1. Failure：由单元测试使用的断言方法判断失败所引起的，即程序运行结果与我们预期的结果不一致，
2. Error：代码异常，可以产生于测试代码本身的错误，也可以是被测试代码中的一个隐藏的bug。

20170920

自己想的测试用例一定要全面：

功能性用例测试

特殊输入测试

可能考虑一些“大数”测试

解决复杂问题的三种方法：画图、举例子和分解（dp、分治等）

在递归效率可能很差的情况下，可以用递归分析问题，但写代码的时候可以用数组（一维或多维数组）来保存中间结果基于循环实现；

当对数组操作时，询问是否可以改变数组或链表中的元素值、值的顺序等

Mongo的认证模式与mysql的**认证模式不**同，前者是基于库的认证，一个账号绑定一个库，如需访问其他库时需要认证之后再切换；后者是全局认证，认证时只需用户名密码即可登录所有的路径下的库。

20170921

使用堆排序解决topK的问题时，若有重复（使用的是TreeSet）,通过判断add()的返回值来对k进行数量调整（+1），反正k只在建立堆时有用到，之后不用再考虑它了：(min --pollLast() max--pollFirst()) 如此简单；

**if**(!treeSet.add(arr[i]))

k++;

归并排序：一种递归求法，每次取一半，然后 Merge()整体; 不是对Merge递归，对外层递归：

void Merge\_sort(int l,int r){

if(l < r){

int m = (l + r) >> 1;

Merge\_sort(l,m);

Merge\_sort(m+1,r);

Merge(l,m,r);

}

}

另一种非递归，按排序前的顺序先是两个两个的排序，然后4个4个的排序，余下的同样排列；

求逆序数对数的方法：1、树状数组；；2、归并排序；-没看

20170922

骰子概率问题： n个骰子，求总数是sum出现的概率；

这也是动态规划，缩小规模：m个骰子-去找m+1个骰子的情况，且无后效性。

保留一个old数组、一个new数组；old数组是当前m个骰子掷出后达到的总和的情况，然后new数组是 m+1个骰子掷出总和的情况，newSum[n]=oldSum[n-1]+oldSum[n-2]...+oldSum[n-6];

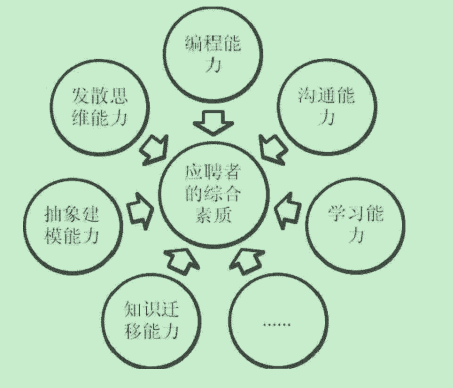
这里注意两点：

1. New与old相互交替，从而可以创建一个二维数组[2][n\*6],然后设一个flag=0；每次交替时：flag=1-flag; Sum[flag][] --真是人才!这个方法可以借鉴；
2. 由于是统计每一个骰子，骰子不可能点数为0，故到达第m个骰子的时候，Sum[flag][m-1] 及之前的都应该设为0，可能会对后来有影响；

加法你是怎么算的？ 一位一进位---

计算机是怎么执行的？ 其实可以第一步不进位a^b，第二步计算进位的数a&b <<1 ，第三步对不进位的和与单独进位的和求和，循环至第一步；

什么是综合能力：



善于提问的人才更有可能给人留下较好的沟通能力和学习能力。

Java的Integer.valueOf(String str) 实现：需要满足atoi；

源码中的实现方法是调用 parseInt(String s ,int radix),各种抛异常，result 存储的是负数的结果，每次与 limit（下界）比较，若为+ ： limit=-2147483647,若为负： limit=-2147483648；

If(result < limit/radix) throw Exception --越界

result \*= radix;

result-= digit;

20170925

全排列实现算法字典序法的描述如下：（使用字典序的一个优点在于，有重复的字符，不会重复排序？）

先将 数组按升序排列，然后去求下一个排列：

设P是1~n的一个全排列 p=p1p2..pj...pk...pn

1. 从当前排列的最右边开始，从右向左找出第一个满足pj<p(j+1)的数
2. 从j+1开始，向右找所有比pj大的数中的最小的数pk（此时，j右边都是降序的，从1的查找条件可以得出）
3. 交换pj，pk
4. 将pj之后的序数进行倒转，得到下一个排列；

静态分派：

虚拟机（准确来讲是编译器）在重载时是通过参数的静态类型来而不是实际类型来确定调用方法的；从深层次理解， 静态类型、实际类型：

Human man=new Man(); //Man 继承于 Human;

Human是静态类型(外观类型)，Man是实际类型,两者在运行时都可以发生一些变化，变量本身的静态类型不会改变，且最终的静态类型是编译期可知的；

20170927

Shadowsocks for Windows github vpn翻墙：

<https://github.com/shadowsocks/shadowsocks-windows>

在java中，一个类用其完全匹配类名（fully qualify class name）作为标识，这里指的是完全匹配类名包括包名和类名；但JVM中一个类用其全称和一个加载类ClassLoader的实例作为唯一标识，不同类加载的类将被置于不同的命名空间；

真正完成类的加载工作是通过调用defineClass来实现的，但启动类的加载过程是通过调用loadClass来实现的； 前一个称作类的定义加载器、后一个是初始化加载器::注意、在Java虚拟机判断两个类是否相同的时候，使用的是类的定义加载器。即那个类加载器启动类的加载过程并不重要，重要的是最终定义这个类的加载器： 两者的关联：一个类的定义加载器是它引用的其他类的初始化加载器。

20170929

**设计模式的设计原则**：

**第一**设计原则：找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起；

**第二**：针对接口编程，而不是针对实现编程；将实现接口的具体动作也封装成类，然后使用组合，别的继承体系下的多态；

0930

效率问题，由于mongo的原因直接查所有的数据会使得查询效率方面较差，考虑查询结果，使用mongo语句查询缩小查询范围；如 模糊查询等、mongo数据库中的语句与python也不太相同，可以使用re模块，传入正则表达式进行解析；

查询看起来应该是这个样子：

db.customer.find({'name': /^s.\*/}) #以s开头的字段；

pymongo的写法如下：

rexExp = re.compile('^s.\*')

db.customer.find({'name':rexExp })

查询Date类型的条件的数据：

.find({"operateDate":{"$gte": ISODate("2017-09-01T00:00:00Z"),"$lt" : ISODate("2017-09-09T00:00:00Z") } })

2017-1013