算法导论初步要学习的内容：
线性表里面的并发队列和阻塞并发队列；
散列表里的冲突解决——链表法 ，开放寻址法，和其他方法；以及散列表里的动态扩容和位图法；树中的平衡二叉树中的——红黑树；多路查找树种的 2-3 树和 2-3-4 树，堆中的优先级队列，斐波那契堆，二项堆树状数组和线段树；掌握图中的基本算法再加上最大流以及二分图；掌握排序算法中的八大排序算法再附加一个桶排序；学习深度优先搜索，广度优先搜索和 A*启发式搜索，掌握查找中的线性表查找，树结构查找和散列表查找；掌握字符串匹配中的 KMP 算法，Robin_Karp 算法，Boyer_Moore 算法，实现 AC 自动机，Tire 算法后缀数组
了解复杂度分析中的平均和均摊时间复杂分析；
掌握基本的贪心算法，分治算法，动态规划，回溯算法和枚举算法；
其他掌握数论，计算几何，概率分析，并查集，拓扑网络，矩阵运算，线性规划等。

最核心的内容：
10 个数据结构：数组、链表、栈、队列、散列表、二叉树、堆、跳表、图、Trie 树；
10 个算法：递归、排序、二分查找、搜索、哈希算法、贪心算法、分治算法、回溯算法、动态规划、字符串匹配算法。

从阅览的前后顺序进行记录
删除操作会导致数组
中的数据不连续。你还记得我们当时是怎么解决的吗？对，用数据搬移！但是，每次进行出
队操作都相当于删除数组下标为 0 的数据，要搬移整个队列中的数据，这样出队操作的时
间复杂度就会从原来的 O(1) 变为 O(n)。能不能优化一下呢？
实际上，我们在出队时可以不用搬移数据。如果没有空闲空间了，我们只需要在入队时，再
集中触发一次数据的搬移操作。借助这个思想，出队函数 dequeue() 保持不变，我们稍加
改造一下入队函数 enqueue() 的实现，就可以轻松解决刚才的问题了。

均摊时间复杂度：
均摊时间复杂度并不经常常见，但见了还是要知道是怎么一回事，当然了最常用的还是最好时间复杂度最坏时间复杂度和平均时间复杂度；均摊时间复杂度用于这种情况下：对于一组连续的树的操作中，大部分情况下时间复杂度都很低，但就是极个别的情况下时间复杂度会很高，而且这一过程还需要保证事件发生的时序关系，这时候我们可以将这一组操作放在一起进行分析，看是佛可以将时间复杂度较高的操作平摊到那些时间复杂度较低的操作上。可以将均摊时间复杂度看成是一种特殊的平均时间复杂度
为什么不用数组或者链表来代替栈？
从功能上来说，数组或者链表确实可以来代替栈但是特定的数据结构是对特定的场景的抽象，同时数组或者链表中暴露了太多的操作接口，使用时比较不可控，也非常容易出

错
动态扩容的顺序栈：首先，在栈中存 next 指针是一种很不明智的方法因为这样内存消耗相对过多，可以采用类似于数组扩容的方式，当空间不够的时候就申请一个更大一点的空间，将原来的数据搬到新的数组中去
删除操作会导致数组中的数据不连续。你还记得我们当时是怎么解决的吗？对，用数据搬移！但是，每次进行出队操作都相当于删除数组下标为 0 的数据，要搬移整个队列中的数据，这样出队操作的时间复杂度就会从原来的 O(1) 变为 O(n)。能不能优化一下呢？实际上，我们在出队时可以不用搬移数据。如果没有空闲空间了，我们只需要在入队时，再集中触发一次数据的搬移操作。借助这个思想，出队函数 dequeue() 保持不变，我们稍加改造一下入队函数 enqueue() 的实现，就可以轻松解决刚才的问题了。
线程池没有空闲线程时，新的任务
请求线程资源时，线程池该如何处理？各种处理策略又是如何实现的呢？
第一种是非阻塞的处理方式，直接拒绝任务请求；另一种是阻塞的处理方式，将请求排队，等到有空闲线程时，取出排队的请求继续处理。先进者先服务，所以队列这种数据结构很适合来存储排队请求。而基于数组实现的有界队列（bounded queue），队列的大小有限，所以线程池中排队的请求超过队列大小时，接下来的请求就会被拒绝，这种方式对响应时间敏感的系统来说，就相对更加合理。
队列的应用非常广泛，特别是一些具有某些额外特性的队列，比如循环队列、阻塞队列、并发队列。它们在很多偏底层的系统、框架、中间件的开发中，起着关键性的作用。比如高性能队列 Disruptor、Linux 环形缓存，都用到了循环并发队列；Java concurrent 并不常见的排序算法：猴子排序、睡眠排序、面条排序等

为什么要区分三中时间复杂度？1.有些排序算法会区分，为了方便对比我们还是最好都区分一下；2.有序度不同的数据对于排序的执行时间是有影响的；
逆序度，满有序度，和有序度之间的关系：逆序度=满有序度-有序度；排序是一个增加有序度减少逆序度最后达到满有序度的过程。
冒泡排序和插入排序的时间复杂度都是 n2 都是原地排序算法，为什么插入排序要比冒泡排序更受欢迎？因为冒泡排序不管怎么优化元素的交换次数都是一个固定值，是原始数据的逆序度，插入排序不管怎么优化，元素的移动次数也等于原始代码的逆序度。但从代码的实现上来看冒泡排序的数据交换要比插入排序的数据交换更复杂，冒泡排序要三个赋值操作，而插入排序只需要一个，因此在冒泡排序和插入排序中首选还是插入排序，从而可以实现性能优化到极致。
普通的快速排序算法并不是原地排序，但是可以用一种特殊的方式来实现原地排序：可以考录用游标将数组分成两个部分，左边部分都是小于枢纽的称作是已处理区间，右边称作是未处理区间，每次都在未处理区间内取出一个元素个枢纽进行对比，如果是小于枢纽就把它插在已处理区间的尾部，类似于插入排序

题目：现在你有 10 个接口访问日志文件，每个日志文件大小约 300MB，每个文件里的日志都是按照时间戳从小到大排序的。你希望将这 10 个较小的日志文件，合并为 1 个日志文件，合并之后的日志仍然按照时间戳从小到大排列。如果处理上述排序任务的机器内存只有 1GB，你有什么好的解决思路，能“快速”地将这 10 个日志文件合并吗？
优质解答：先构建十条 io 流，分别指向十个文件，每条 io 流读取对应文件的第一条数据，然后比较时间戳，选择出时间戳最小的那条数据，将其写入一个新的文件，然后指向该时间戳的 io 流读取下一行数据，然后继续刚才的操作，比较选出最小的时间戳数

据，写入新文件，io 流读取下一行数据，以此类推，完成文件的合并， 这种处理方式，日志文件有 n 个数据就要比较 n 次，每次比较选出一条数据来写入，时间复杂度是 O(n)，空间复杂度是 O（1）

桶排序对数据的要求：首先要排序的数据可以很容易的划分成 m 个同，并且桶与桶之间存在着田然的大小顺序，这样每个桶的数据都能在排序完之后桶与桶之间不需要再进行排序，其次数据在同志间的分布是比较均匀的，如果数据经过桶划分后，有些桶的数据非常多，有些桶的数据非常少，那桶内数据排序的时间复杂度就不是一个常量级，极端情况下，数据的划分都在一个桶里，那就退化成 O（nlogn）的排序算法了；桶排序更加适合在外部排序中，数据存放在磁盘上，数据量比较有大，而且内存也非常有限，无法将全部的数据加载到内存中；
几种流行的语言中采用的排序方法：1.Java 中 Arrays.sort 采用的是 TimSort 算法，对元素个数小于 47 的采用是二分查找和插入排序，大于 47 但小于 286 的用的是双轴排序，元素个数大于 32 的采用的是归并排序；golang 标准库中的 sort 用的是快排+希尔排序+插入排序；.net 中三个数以内的直接比较交换实现，大于 3 小于 16 用的是 插入排序，大于 16 并且有深度限制的用的是堆排序，大于 15 且深度不受限制的用的是快速排序；c++标准库中用的 https://liam.page/2018/09/18/std-sort-in-STL/；google 中用的都是快排，对 10 以内的用快排，10,1000 之间的选择中点作为快排节点，1000 以上的每隔 200-215 个数选择一个数，将选出来的数作为枢纽进行快排
二分查找算法中比较容易出错的几个点：1.循环退出条件是 low<=high,而不是 low<high;2.mid 的取值实际上 mid=(low+high)/2 这种写法存在问题，可能会溢出，改进方法是写成 low+(high-low)/2,也可以用性能更优的方式将除二操作转化为 low+((high-low)>>1);3.low 和 high 的更新，必须要写成 low=mid+1,high=mid-1;否则就会进入死循环。
数据量太小或者数据量太大都不适合使用二分查找：数据量太小，完全没有必要使用二分查找，顺序遍历就已经足够，查找的速度差不都且没有那么多的赋值操作；数据量比较大的时候也不适合用二分查找因为二分查找的底层需要数组这种数据结构，而数组的内存空间是连续的，堆内存要求比较苛刻

队列的使用场景：高性能队列 disruptor Linux 环形缓存 java concurrent 并发包的利用 arrayblockingqueue 来实现公平锁；
递归怎样检测环的存在：构造一个 set 集合或者散列表，每次取上层推荐人就去散列表里面查找，没有查到的话就加入，如果查到了就表示存在环；
二分查找所应用的地方：大部分能够用二分查找解决的问题都可以使用散列表或者是二分查找树来解决，但是在下面的几种情况中必须要使用二分查找来解决：就是二查找的近似问题，这些问题使用二分查找的有事更加明显
二分查找比较容易出错的细节，也经常会产生 bug 的点：终止条件，区间上下界更新方法，返回值的选择；
调表通过随机函数来维护平衡性（索引与原始链表之间的大小平衡）
linkedhashmap 是通过双向链表和散列表这两种数据结构的组合来实现的，linkedhashmap 中的 linked 实际上指的是双向链表，并非指的是用链表法来解决散列冲突问题；
hash 算法的原理将任意长度的二进制串映射成固定长度的二进制值串；
hash 算法的第五应用之负载均衡：

1.怎样实现一个会话粘滞的负载均衡：最直接的方法是维护一张映射关系表，这张表的内容是 ip 地址或者会话 ID 与服务器编号的映射关系，但这样如果客户很多时，映射表会很大；我们可以通过哈希算法对客户端的 ip 地址或者是会话 ID 计算 hash 值，将取得的 hash 值与服务器列表的大小进行取模运算，最终的到的值就是路由到服务器的编号；

day01：
一些计算机必回的基础知识：
万维网的三要素：1.http：一个请求在网络上怎么传输；2.html 对应互联网早期的论文格式；3.URL（统一资源定位符）指示一个资源在互联网的位置；
URL 的三大部分和四小部分：
https:// （协议）
detail.tmall.com （域名）
/item.htm?id=15835225798&ali_trackid=17_ec596b5a12c12f34101e38b0c1be41d9&spm=a21bo.21814703.201862-3.1&skuId=4451679141229（服务器内部路径）
http://（域名） 115.29.141.32:8085 （ip+端口） /#/mall/show/index（服务器内部路径）
DNS 解析：dns 称作是域名系统相当于 Internet 的电话簿，可以将域名转换成 ip 地址，以便浏览器能够加载 Internet 资源；
DNS 涉及有四个 DNS 服务器：1.DNS 解析器是一种服务器主要是通过 Web 浏览器等应用程序来接收客户端计算机的查询；2.根域名服务器：将人类可读的主机名转换为 ip 地址的第一步，可以视为指向不同暑假的图书馆索引；TLD 域名服务器可以视为图书馆的特定书架，可以解读 com；权威域名服务器：可以将最终域名服务器视为书架上的字典，将特定的名称转换为其定义。
如何获得一个网页：1.在浏览器中输入一个地址；2.浏览器把域名变成 ip 地址；3.浏览器根据 ip 地址发起网络请求；4.根据 ip 地址，在互联网上找到对应的计算机要分析“服务器内部路径”根据请求的描述获得对应的计算机里面的内容；6.把这个根据“服务器内部路径”找到的内容返回给请求者；7.浏览器获得对应的内容解析显示；

集合类与数据结构：
1.为什么要有集合类：可以帮助实现存储更多类型问题，扩容问题，内存浪费问题，数据查找问题，数据删除问题；
数据集合的特点：只能存储引用数据类型，可以自动地调整自己的大小
数组和集合类都是容器他们有何不同：1.数组可以存储基本数据类型的数据，集合不可以；2.数组的长度是固定的，集合可以自动的调整自己的大小；3.数组的效率高，相对来说集合效率比较低；数组没有 api，集合有丰富的 API；
集合类的三个角度：从使用者的角度是数据容器（研究怎么添加删除查找和替换）2.从底层结构来看是一个单链表 3.模拟的数据结构是线性表；
数组和链表：
1.你理解数组是什么样的？它的主要特点是什么？答：数组的本质是一段固定大小的连

续的内存空间，并且这片连续的内存空间被分割为等长的小空间，主要的特点是随机访问；
2.为什么数组的索引一般都是从 0 开始？根据计算机的寻址公式：i address=base address+(i-1)*type length,所以在寻址的过程中从 0 开始可以少计算一次减法操作；
3.为什么数组的效率比链表高？从时间复杂度的角度进行分析，数组添加删除和查找的平均时间复杂度分别是 On、On 和 O1；链表的插入删除和查找的时间复杂度分别表示 O1，O1 和 On 但是链表在进行每次的添加或者是删除的过程中都先要经过查找；从计算机的体系结构来讲 CPU 缓存会把一片连续的内存空间读入，因为数组结构是连续的内存地址，所以数组全部或者部分元素被廉租存在 CPU 缓存里面，平均读取每一个元素的时间只要 3 个 CPU 的时钟周期，而链表的节点分散在堆空间上，不能一次性将所有的节点都添加到缓存里面，只能边计算边读取。增加了节点读入内存所花费的时间；

泛型：
泛型的好处：1.提高了程序运行的安全性 2.将运行体检遇到的问题转到了编译期；3.省去了类型转换的麻烦；
泛型中需要注意的两个点：1.泛型的定义：T：type　E:element　K:key　V:value
2.对于泛型类，可以在泛型类上定义多个泛型，定义了泛型可以不在泛型类中使用，但是创建这个包含泛型对象的是由定义了几个泛型就要传几个泛型；
3.如果创建对象需要传入某种泛型，但是没有传入，这个泛型默认是 object 类型的。e.g.：User user4=new User（）；
泛型擦除：java 定义的泛型，会在编译的时候会被擦除掉，替换成 object 类型，用到了泛型的地方会发生类型转化；从效率上来讲，使用反省和我们直接把某种需要泛型的类型定义成 object 效率是一样的没因为底层最终都会编译成 object 类型，也都不需要类型转换，但是，比我们手动写 object 优点是。一个我们进行手写的代码强转，一个 jvm 在编译的时候会帮助我们做强转
泛型的通配符和泛型协变问题：泛型是不变的，不允许出现 List<Fruit> ...=new arraylist<apple>()这种情况，协变是需要建立某种类型之间的向上或者是向下的类型转变，这时引入了通配符<? extends Fruit>就可以实现向上转型，而<? surper ...>就能实现向下转型；

栈：
栈的应用场景：函数调用栈，反序字符串，实现括号匹配问题，编译器实现表达式的求值，浏览器的前进和后退功能，实现深度优先遍历
队列：
队列的应用：缓存，实现广度优先遍历，阻塞队列通常用在生产者消费者模型当中，

树：
树表示的是一对多的关系；
结点的层级从根开始定义，根为第一层，根的孩子为第二层。若某结点在第 i 层，则其孩子就在 i+1 层。
对任意结点 ni，ni 的深度为从根到 ni 的唯一路径的长。因此，根的深度为 0。ni 的高是从 ni 到一片树叶的最长路径的长。因此，所有树叶的高都为 0。
树的转化：如果一个节点有孩子，那么把它第一个子节点作为这个节点的 left 孩子，2.

如果一个节点有 right 兄弟节点，那么久把 right 兄弟节点看成是树的右孩子；

2-3-4 树的添加操作：在 2-3-4 树的添加的过程中，为了避免最后添加的时候需要向上进行分裂，当父元素没有位置可以分裂的会后，为了避免这种情况的发生，我们一般保证再添加的过程中遇到四节点先向上进行分裂；
2-3-4 树的删除操作：1.对于非单 key 的叶子节点可以直接删除，如果删除的是中间节点需要先替换后删除；如果删除的叶子结点是单 key 的需要借结点，为了保证父节点不是单 key 的，在查找的过程中，保证查找遍历的节点是非单 key，如果父节点也是单 key 的需要借结点；

红黑树是一个特殊的二叉搜索树，红黑树上一部分是黑色，一部分是红色，叶节点是黑色的，没有连续的红色节点，红色节点的孩子和父亲都是黑色，对每个节点
2-3-4 树变成红黑树：单 key 节点变成黑色节点，把双 key 节点变成一黑一红，3key 节点变成中间红两边黑，或者中间黑两边红（常用）
llrb_tree 左倾红黑树：拆除 2-3-4 树的时候红色在左边，要求一个节点所有节点在红黑树中，如果只有几个孩子是红色，那么这个红色的孩子应该是这个节点的做孩子；

红黑树的查找：同二叉搜索树，对于红黑树的查找来说和二叉搜索树没有任何的区别
红黑树的添加：添加分为两个步骤：查找：遇到四节点分裂-->添加-->修复
红黑树新加的节点一定是红色，红黑树再添加的过程中，要保证遍历的节点是非四节点，想上分裂，中间节点由黑变红，左右节点由红变黑->可能导致红黑树不在左倾-->怎么办，添加完成后自底向上逐层修复
红黑树的修复：
红黑树的删除：如果删除的是非最底层结点　先替换　再删除；如果删除的确实是最底层结点（非单 key-->直接删除；如果是单 key-->借结点再进行删除，但是结点不一定可借）
如果删除的是最大值，则要保证路径上结点的 right 不是红色同时 right.left 不是红色；如果删除的是最小值，则要保证 left 孩子不是非单 key，即 mid.left 不是红色同时 mid.left.left 也不是红色；

使用红黑树不使用二叉平衡树？红黑树是局部变换，修复比二叉平衡树简单

B 树：节点最多有 m 棵子树　2.m/2

手写集合类的作用？java 的集合类有什么作用，用来存储数据；
集合类的三个角度：实用角度：数据容器（希望比较快速的进行增删改查）底层引入了组织方式：数据结构，容器内部的数据按照哪种结构组织（本质目的是为了高效和快速总是落实到代码上怎么表示数据结构，用什么对象关系来描述数据结构，就回到数组和链表上）数据结构是抽象的一种概念，用什么对象关系具体在内存上是实际存在的
要求：1. 至少要知道什么是线性表，栈队列树，二叉树，二叉搜索树，自平衡二叉搜索树，红黑树；2，多写代码

面试会问哪些内容：数据结构的理论，红黑树几个问题

collection 集合类：记忆（每一个集合类的特点，记忆 api，hashmap 重点记忆）
面试重点：ArrayList 和 hashmap
Se 面试点：string，多线程，垃圾回收的几种算法机制；集合类；网络编程，流

java 的集合类分成两个体系：一个是 collection：作为数据容器存在，存储的是单个元素；2.map 作为数据容器存储的是键值

怎么记忆：集合类的特点。我们写你们记，（记忆顺序）
collection 的特点：1.collection 是 collection 集合体系的顶级接口；2.collection 的一些子实现是有序的，另一些子实现是无序的；3.一些子实现允许存储重复元素。。。4.一些允许存储 null 另一些不允许存储 null；
将入一个接口是另一个接口的子类，它想对父接口进行功能增强
collection 的主要方法：
集合类的添加方法：add（E e），addall（collection<? extends E >C）//可以将 collection 里面的东西往 cllection2 里面 copy 一份；3.clear 清空操作；toArray[]
toarray 是一个泛型方法，根据参数可以返回一个和参数累次能够相同的数组；当我们给定一个和元几个类型不相同类型的数组，有可能会报错，如果我们给定一个数组长度不够（数组长度小于 collection 中的元素）穿进去的数组和返回的数组长度不相同时，返回的长度是原集合类中的长度；如果长度不够。和集合类中存储数据量相同，返回的数组和做参数的数组；
如果给的数组过长，那么返回的数组和参数的数组是同一个数组，并且这个数组的 collection 有 x 个元素，这个数组的前 x 个位置存的是 collection 里面的元素，x+1 存储的是 null
<E>iterator:获得一个迭代器，可以实现遍历：原理是：collection 类继承了 iterable 接口，拥有了一个定义的方法 iterator 以为着所有的 collection 的子类中都要实现 iterator 方法，collection 的所有子类底层结构未必一样，遍历方式必定存在差异，每一个 collection 的子类都应该拥有符合自己数据存储方式的 iterator 方法，这个 iterator 方法返回的对象是一个内部类对象
iterator 是一个接口：有三个方法：hasnext()，next()和 remove()（删除刚刚遍历过的元素）
面试：迭代器开始指向第一个元素之前，iterator.next()指向的是第一个元素和第二个元素之间；
iterator 是一个内部类，保存一些标记，标记元数据，并没有从原数据中复制出新的标记，因为这样遍历效率更高，相比较 toarray，toarray 的遍历是赋值原数据放到数组里面，复制效率不高
删除操作不能在未遍历之前删除，也不能连续的删除，因为删除操作是删除刚刚遍历的结果，所以不希望在遍历的过程中用遍历删除，它希望你在做 iterator 遍历的目的更偏重遍历而不是删除）

为什么 key-value 会更加常用：因为 key-value 有自我描述性
hashSet 的底层是 hashmap
取余用&？？？？

并发修改异常
加锁使效率降低；
modcount：记录集合类的修改次数
在多线程情况下为了避免一个线程在遍历而另外一个线程在修改会导致遍历结果不对，为了避免这种情况出现又不想用加锁的方式处理-->加锁会导致代码的运行效率降级；所以选择用标记的方式来实现数据同步（expectModCount=modCount）,我们会在遍历的 iterator 对象为胡一个标记，一番元集合的数据被别的线程修改，那么立马可以检测出这个修改，由于设计不够完美，会导致单线程的情况下，也会产生并发修改议程（当）

foreach：
foreach/增强 for 循环/加强 for 循环
如果使用 foreash 循环，不要再 foreash 循环中修改集合数据，因为 foreash 循环底层是 iterator 迭代，会抛出并发修改异常
意味着使用 foreash 樽还不可以删除元素---->没有这个需求
foreash 底层（除了数组）都是使用 iterator 遍历、编译，意味着 foreash 循环中，不要通过元集合的修改方法去修改元集合类的修改方法来修改元集合类的数据（添加或删除改变元集合类的结构性修改）
foreash 循环一般用来遍历集合类，但是也可以用来遍历我们自己实现的一个类型，要求自己实现的类型中包含 iterator 方法；数组的 foreash 和集合类的 foreash 循环具有不可比性，数组的 foreash 循环底层是 fori 循环；

List 以及 list 的子类：
如何记忆：谁的子类接口-->描述数据结构 底层结构 初始和扩容 有序/允许重复/null 线程安全

1.特点：1.list 是 collection 接口的子接口；2.描述的是线性表这种数据结构（线性表：有序序列-->有下标/有下标 api）3.必定有序；4.允许存储重复元素；5.允许存储 null
从语言使用的角度来讲 ArrayList ...=newArrayList();不太好
原因是接口定义的是规范

boolean	add(Ee) 向列表的尾部添加指定的元素（可选操作）。
void	add(intindex,Eelement) 在列表的指定位置插入指定元素（可选操作）。
boolean	addAll(Collection<?extendsE>c) 添加指定 collection 中的所有元素到此列表的结尾，顺序是指定 collection 的迭代器返回这些元素的顺序（可选操作）。
boolean	addAll(intindex,Collection<?extendsE>c) 将指定 collection 中的所有元素都插入到列表中的指定位置

	（可选操作）。
void	clear() 从列表中移除所有元素（可选操作）。
boolean	contains(Objecto) 如果列表包含指定的元素，则返回 true。
boolean	containsAll(Collection<?>c) 如果列表包含指定 collection 的所有元素，则返回 true。
boolean	equals(Objecto) 比较指定的对象与列表是否相等。
E	get(intindex) 返回列表中指定位置的元素。
int	hashCode() 返回列表的哈希码值。
int	indexOf(Objecto) 返回此列表中第一次出现的指定元素的索引；如果此列表不包含该元素，则返回-1。
boolean	isEmpty() 如果列表不包含元素，则返回 true。
Iterator<E>	iterator() 返回按适当顺序在列表的元素上进行迭代的迭代器。
int	lastIndexOf(Objecto) 返回此列表中最后出现的指定元素的索引；如果列表不包含此元素，则返回-1。
ListIterator<E>	listIterator() 返回此列表元素的列表迭代器（按适当顺序）。
ListIterator<E>	listIterator(intindex) 返回列表中元素的列表迭代器（按适当顺序），从列表的指定位置开始。
E	remove(intindex) 移除列表中指定位置的元素（可选操作）。
boolean	remove(Objecto) 从此列表中移除第一次出现的指定元素（如果存在）（可选操作）。
boolean	removeAll(Collection<?>c) 从列表中移除指定 collection 中包含的其所有元素（可选操作）。
boolean	retainAll(Collection<?>c) 仅在列表中保留指定 collection 中所包含的元素(可选操作)。
E	set(intindex,Eelement) 用指定元素替换列表中指定位置的元素（可选操作）。

int	size() 返回列表中的元素数。
List<E>	subList(intfromIndex,inttoIndex) 返回列表中指定的 fromIndex（包括）和 toIndex（不包括）之间的部分视图。
Object[]	toArray() 返回按适当顺序包含列表中的所有元素的数组（从第一个元素到最后一个元素）。
<T>T[]	toArray(T[]a) 返回按适当顺序（从第一个元素到最后一个元素）包含列表中所有元素的数组；返回数组的运行时类型是指定数组的运行时类型。

listiterator ： 是 iterator 的一个子接口，接口是另外一个接口的子接口，增强功能；

方法摘要

void	add(Ee) 将指定的元素插入列表（可选操作）。
boolean	hasNext() 以正向遍历列表时，如果列表迭代器有多个元素，则返回 true（换句话说，如果 next 返回一个元素而不是抛出异常，则返回 true）。
boolean	hasPrevious() 如果以逆向遍历列表，列表迭代器有多个元素，则返回 true。
E	next() 返回列表中的下一个元素。
int	nextIndex() 返回对 next 的后续调用所返回元素的索引。
E	previous() 返回列表中的前一个元素。
int	previousIndex() 返回对 previous 的后续调用所返回元素的索引。
void	remove() 从列表中移除由 next 或 previous 返回的最后一个元素（可选操作）。
void	set(Ee) 用指定元素替换 next 或 previous 返回的最后一个元素（可选操作）。