1、List 和 Set 的区别

List：有序的，允许重复的

Set：无序的，不允许重复的

2、Vector 和 ArrayList 的区别

Vector 最早的动态数组，线程安全，初始化长度 10，扩容为 2 倍，Vector 也支持 foreach 和 Iterator，还支持 Enumeration 迭代器

ArrayList 是相对新的动态数组，线程不安全，初始化长度 10，扩容为 1.5 倍，ArrayList 支持 foreach 和 Iterator

3、动态数组与 LinkedList 的区别

动态数组的底层实现数组

LinkedList 的底层实现是链表

展开来说：

动态数组对于按索引来操作（查询）效率比较高，链表如果按照索引操作（查询）效率比较低

链表对于插入和删除效率比较高，因为不用移动很多元素，只有修改前后元素的关系。

4、Hashtable 与 HashMap 的区别

Hashtable 是散列表，哈希表，是旧版的，线程安全的（支持同步的），不允许 key,value 为空

HashMap 也是哈希表，相对新版，线程不安全的（非同步）,允许 key,value 为空

5、HashSet 与 TreeSet 的区别

HashSet：是完全无序的，依据元素的 equals 方法

TreeSet：是按照元素的“大小”顺序排列，依据元素的“大小”，认为大小相同的两个元素就是“相等，重复”的元素。

6、HashSet 与 LinkedHashSet 的区别

HashSet：是完全无序的

LinkedHashSet：LinkedHashSet 遍历时可以保证元素的添加顺序，它是 HashSet 的子类

7、HashMap 与 LinkedHashMap 的区别

LinkedHashMap 是 HashMap 的子类，比 HashMap 多维护了添加顺序

8、Properties 与 Hashtable 的区别

Properties：它的 key 和 value 都是 String 类型，可以从流中加载或者把数据存到流中

Properties 是 Hashtable 的子类

9、HashMap 与 TreeMap 的区别

HashMap：无序的，底层实现是 JDK1.7（数组+链表），JDK1.8（数组+链表/红黑树）

TreeMap：按照 key 的大小排序，底层实现红黑树

10、Collection 与 Map 的区别

Collection：存储一组对象。

Map：存储键值对，映射关系。

1．内存中多个相同数据类型数据的存储“容器”：

数组 、 集合

2．数组存储的特点：

连续的

数组存储的弊端：

（1）长度一旦确定，就不能修改，如果要修改，那么程序员就要创建新的数组，然后复制元素等，比较麻烦

（2）数组无法获取有效元素的个数，需要借助于例如total这样的变量

3．集合存储的优点：

集合的底层存储结构有很多种，程序员在选择时可以多样化。

可以选择有序的，无序的，可重复的，不可重复的。

4．数据结构研究的问题：

集合的物理结构由两种演化：

（1）数组

（2）链式存储结构

依赖于结点类型

单链表：

```
class Node{
    Object data;
    Node next;
}
```

双向链表：

```
class Node{
    Node prev;
    Object data;
    Node next;
}
```

二叉树：

```
class Node{
    Node Parent;
    Node left;
    Object data;
    Node right;
}
```

4.4 常用实现类：

HashSet：完全无序的，不可重复依赖于元素的hashCode和equals方法
TreeSet：按照元素的大小顺序，依赖于元素的自然排序（compareTo()），或者指定定制比较器对象（Comparator接口compare()）
LinkedHashSet：维护添加顺序，不可重复依赖于元素的hashCode和equals方法

4.5 存储的元素的要求：

HashSet和LinkedHashSet：建议重新hashCode和equals方法
TreeSet：建议元素实现java.lang.Comparable接口，或者指定java.util.Comparator接口的对象

java定义的一个迭代器接口，用于遍历Collection.(List或Set)

步骤：

（1）先获取Iterator的对象

Collection系列的集合对象.iterator()

（2）调用Iterator的方法

```
boolean hasNext()
E next()
```

如果遍历时要根据条件删除，也可以调用Iterator的对象.remove()

双列集合框架：Map

1．存储数据特点：存储键值对象

2．常用方法：

```
添加：
    put(K key, V value)
    putAll(Map map)
删除：
    remove(K key)
    clear()
查询：
    containsKey(K key)
    containsValue(V value)
    V get(K key)
    isEmpty()
获取有效映射关系的对数：int size()
遍历相关的方法：
    Set keySet()
    Collection values()
    Set entrySet()
```

3．常用实现类：

Hashtable：旧版散列表，哈希表

HashMap：新版散列表，哈希表

LinkeHashMap：比HashMap多维护了添加的顺序，是HashMap的子类

Properties：是Hashtable的子类，key,value的类型都是String

TreeMap：按照key的大小顺序排列

1. 单列集合框架结构：

2. Collection接口常用方法：（记住）

添加：

```
add(E obj)
addAll(Collection other)
```

删除：

```
remove(E obj)
removeAll(Collection other)
clear()
```

查询：

```
contains(E e)
```

查询：

```
contains(E e)

containsAll(Collection all)

isEmpty()
```

获取有效元素的个数：

```
int size()
```

保留交集：

```
retainsAll(Collection other)
```

遍历相关：

```
Object[] toArray()

Iterator iterator()
```

3. List接口

3.1 存储的数据特点：有序的，可重复的

3.2 常用方法：（记住）

比Collection多了一些方法，和index相关的

添加：

```
add(int index,E e)
addAll(int index,Collection other)
```

删除：

```
remove(int index)
```

查询：

```
E get(int index)
int indexOf(E e)
```

```
set(int index,Object value)
```

和遍历相关的：

```
ListIterator listIterator()
ListIterator listIterator(int index)
```

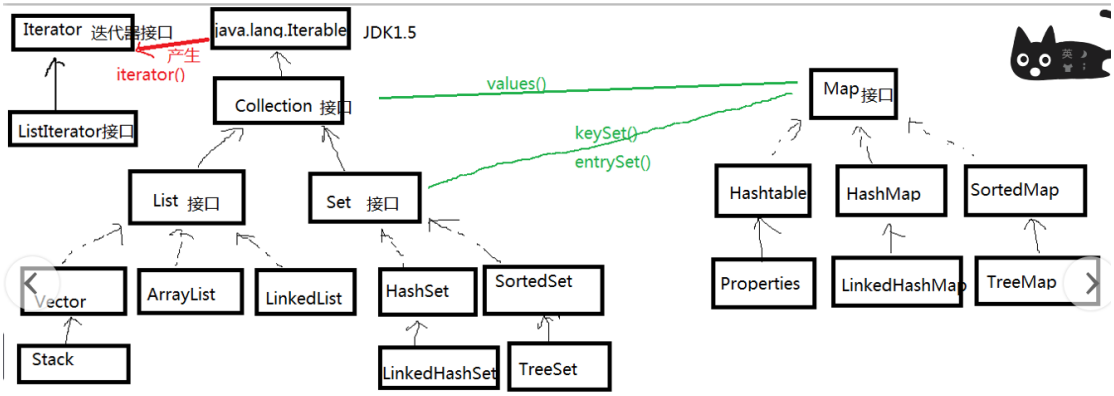
3.3 常用实现类：

Vector：旧版的动态数组

ArrayList：新版的动态数组

LinkedList：双向链表，实现了List接口，又实现了Queue，Deque接口，即可以作为双端队列，队列等来使用。

Stack：栈结构，继承Vector。




```
/*
 * Set的底层实现，内部实现：
 *
 * HashSet：内部实现是HashMap
 *       添加(add)到HashSet的元素是作为HashMap的key，所有的value共享同一个Object类型的常量对象PRESENT
 * LinkedHashSet：内部实现是LinkedHashMap
 *       添加(add)到LinkedHashSet的元素是作为LinkedHashMap的key，所有的value共享同一个Object类型的常量对象PRESE
 * TreeSet：内部实现是TreeMap
 *       添加(add)到TreeSet的元素是作为TreeMap的key，所有的value共享同一个Object类型的常量对象PRESENT
 */
```

```
/*
 * java.util.Iterator接口。
 * （1）boolean hasNext()
 * （2）E next()
 * （3）void remove()
 *
 * 所有的Collection系列的集合都包含一个：
 *  Iterator iterator()
 *
```

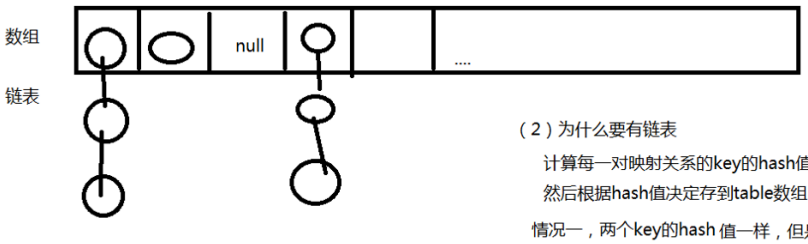
```
 * 所有的Collection系列的集合都包含一个：
 *  Iterator iterator()
 *
 * 问？Iterator接口的实现类在哪里？
 *   或Collection系列的集合中的iterator()是返回谁（哪个类）的对象？
 *
 * 例如：
 * ArrayList：内部有一个内部类,Itr  implements Iterator接口
 * Vector：内部也有一个内部类,Itr  implements Iterator接口
 * LinkedList：内部有一个Itr implements Iterator接口,还有ListItr implements ListIterator接口
 *
 * 结论：每一种Collection系列集合的实现类中都有一个内部类还实现Iterator接口。
 *
 * 为什么要这么做？
 * 这个Iterator接口对象的作用就是遍历，迭代集合的元素用，设计为内部类的好处，就是可以方便的，直接的访问集合的内部的元素。
 * 比喻：公交车上售票员，火车上乘务员，飞机上空姐
 *    即每一个集合的迭代器只为当前的集合服务。
 */
```

```
/*
 * Iterator迭代器和foreach遍历，多线程并发的问题。
 *
 * 用迭代器或foreach遍历时，再用集合对象的remove方法时会报ConcurrentModificationException异常，
 * 因为迭代器和集合两条线路同时操作元素。
 */
```

```
* Iterator迭代器和foreach遍历，多线程并发的问题。
*
* 用迭代器或foreach遍历时，再用集合对象的remove方法时会报ConcurrentModificationException异常，
* 因为迭代器和集合两条线路同时操作元素。
*
* 在foreach或获取Iterator迭代器对象时，就会用expectedModCount记录当前集合被修改的次数modCount的值。
* expectedModCount = modCount;
* 如果在迭代器或foreach遍历的过程中，发现expectedModCount != modCount，说明你用集合的add或remove等方法，
* 了当前集合的元素，就会报ConcurrentModificationException异常
*
* 如果你用Iterator迭代器自己的删除方法的话，那么它会重新修改expectedModCount变量的值，包证与与modCount的值一样
*
* 为了避免将来因为其他线程而修改了集合的元素，导致当前这个操作的数据不正确的风险，干脆快速失败，只有发现
* expectedModCount != modCount，说明数据已经不是原来的而数据，就快速失败。
```

```
* ArrayList：动态数组
*   内部实现：数组
*
* 1、初始化大小：10
* 如果JDK1.8时new ArrayList()，发现数组初始化为一个DEFAULTCAPACITY_EMPTY_ELEMENTDATA，长度为0的空数组。
* 如果JDK1.6时new ArrayList()，发现数组直接初始化为一个长度为10的Object[]
* 如果JDK1.7时new ArrayList()，发现数组初始化为一个 EMPTY_ELEMENTDATA，长度为0的空数组;
*
* 2、添加元素时，如果数组满了，如何扩容
* 扩容为1.5倍
*
* JDK1.7和JDK1.8时，因为一开始是空数组，那么第一次扩展为长度为10的数组。
* 然后不够了，再扩为原来的1.5倍
*
* 3、删除元素时，数组会不会缩小
*   不会
*   但是像ArrayList有一个trimToSize()可以调整大小。
```

```
* LinkedList：
* 1、内部实现：链表
* 记录Node first;
*     Node last;
*
* 2、add(xx)
* 默认添加到链表的尾部 linkLast(xx)
*
* 3、add(int index,xx)
*
* 4、remove(xx)
*/
```



（1）数组的元素的类型是什么

Map.Entry接口的类型 (key,value)

HashMap.Entry内部类类型，实现了Map.Entry (key,value,next)

（2）为什么要有链表

计算每一对映射关系的key的hash值，

然后根据hash值决定存到table数组[index]

情况一，两个key的hash 值一样，但是equals不一样

-->index相同

情况二：两个key的hash值不一样，equals也不一样，但是通过公式运算后-->index相同

那么table[index]中无法存放两个对象，所以只能设计为链表的结构，把它们串起来

```
* JDK1.7以及之前：HashMap的底层实现是数组+链表
*
* 1、数组的类型
* 2、为什么有链表
*
* 3、数组的初始化的长度是多少
* 初始化长度默认为16，
* 如果手动指定，那么也必须是2的n次方，如果不是会自动纠正为2的n次方
*
* 4、数组是否会扩容
* 会
*
* 为什么要扩容？因为如果不扩容，会导致链表会变的很长，那么它的查询效率，添加的效率整个会降低
*/
```

```
* 什么情况会扩容？
* 有一个变量threshold阈值来判断是否需要扩容，
*   当这个threshold达到临界值时，就会考虑扩容，还要看当前添加(key,value)时，是否table[index]==null，
* 如果table[index]!=null，那么就会扩容，如果table[index]==null，那么本次先不扩容。
*
* DEFAULT_LOAD_FACTOR：默认加载因子0.75
* threshold = table.length * 0.75
* 第一次：16 * 0.75 = 12，当我们size达到12个，就会考虑扩容。
*/
```

```
* 5、index如何计算
* 拿到一个key的hash值之后，如何计算[index]
* (1)key是null，固定位置[index]=[0]
* (2)第一步，先用hashCode值通过hash(key)函数得到一个比较分散的“hash值”
*     第二步，再根据“hash值”与table.length做运算得到index
*           hash & table.length-1 按位与   确保index在[0,length-1]范围内
*
```

* 6、如何避免key不可重复的
* 换句话说，如果key重复了，会怎么办？
*
* 如果key相同，那么我们会替换旧的value
* key相同，先判断hash值，如果hash值相同，判断key的地址或equals是否相等。
*

* 7、新的（key,value）添加到table[index]后，发现table[index]不为空，怎么连接的
* (key,value)是作为table[index]的头，原来下面的元素作为我的next。
*

* JDK1.8的HashMap：底层实现（数组+链表/红黑树）
*
* 1、为什么要从JDK1.8之前的链表设计，修改为链表或红黑树的设计？
* 当某个链表比较长的时候，查找效率还是会降低。
* 为了提高查询效率，那么把table[index]下面的链表做调整。
* 如果table[index]的链表的节点的个数比较少，（8个或以内），就保持链表。如果超过8个，那么就要考虑把链表转为一棵红黑树。
* TREEIFY_THRESHOLD：树化阈值，从链表转为红黑树的临界值。
*
* 2、什么时候树化？
* table[index]下的结点数一达到8个就树化吗？
* 如果table[index]的节点数量已经达到8个了，还要判断table.length是否达到64，如果没有达到64，先扩容。
*
* 演示：8个->9个 length从16->32
*   9个->10个 length从32->64
*   10个->11个 length已经达到64，table[index]就从Node类型转为TreeNode类型，说明树化
*
* MIN_TREEIFY_CAPACITY：最小树化容量64

* 3、什么时候从树-->链表
* 当你删除结点时，这棵树的结点个数少于6个，会反树化，变为链表
* UNTREEIFY_THRESHOLD：6个
*
* 树的结构太复杂，当结点少了之后，就用链表更好。
*

* 4、put的过程
* （1）[index]的计算问题
* 第一步用key的hashCode值调用hash()函数，干扰hash值，使得(key,value)更加均匀的分布table数组中。
*   JDK1.8中hash()算法更优化。
*
* 第二步： hash值与table.length-1做&运算，保证index在[0,length-1]范围内
*
* （2）扩容问题
* 第一种：当某个table[index]的链表的个数达到8个，并且table.length<64，那么会扩容
* 第二种：size >= threshold，并且table[index]!=null
*   threshold = table.length * loadFator（它的默认值DEFAULT_LOAD_FACTOR：0.75）
*
* （3）当把(key,value)添加到链表中，JDK1.8是在链表的尾部
* 成语：七上八下

```
* 泛型：JDk1.5
*   泛化的类型，参数化的类型。
*
* 1、问？
* 最早设计比较器、集合等的时候，是没有泛型的，因为集合是个容器，用来装对象的，那么为了能够装任意类型的对象，
* 所以集合中全部用Object处理。
*
```

```
* 这么处理（用Object处理）的问题是什么？
* （1）拿到的值都是Object，如果要再用，还得类型转换-->麻烦
* （2）无法阻止逻辑意义上不符合类型的对象-->不安全
*
```

```
* 泛型：JDk1.5
*   泛化的类型，参数化的类型。
*
* 1、问？为什么要泛型
* 最早设计比较器、集合等的时候，是没有泛型的，因为集合是个容器，用来装对象的，那么为了能够装任意类型的对象，
* 所以集合中全部用Object处理。
*
* 这么处理（用Object处理）的问题是什么？
* （1）拿到的值都是Object，如果要再用，还得类型转换-->麻烦
* （2）无法阻止逻辑意义上不符合类型的对象-->不安全
*
* JDK1.5之后，有了泛型？
* （1）不需要类型转换                -->简洁
* （2）不符合类型的对象，无法接受      -->安全
```

```
* 2、如何设计的泛型
* 类比：设计方法时，在实现方法功能有未知的数据，用形参表示
*       设计类或接口等时，某个属性的类型或者方法形参的类型等类型不知道时，把这个类型设计为泛型，我们称为类型形参。
*
```

```
* 2、如何设计的泛型
* 类比：设计方法时，在实现方法功能有未知的数据，用形参表示
*       设计类或接口等时，某个属性的类型或者方法形参的类型等类型不知道时，把这个类型设计为泛型，我们称为类型形参。
*
* 形参：
* public int getMax(int a,int b){}  ->a,b称为数据形参
*      getMax(5,6)                    ->5,6称为数据实参。
* MyArrayList<E>:                   --><E>称为类型形参
*      MyArrayList<String> list;    --><String>称为类型实参
*
```

```
* 泛型有两种形式：
* 1、泛型类、泛型接口
* 2、泛型方法
*
* 一、泛型类、泛型接口
* 语法格式：
* 【修饰符】class/interface 类型名<泛型形参列表>{
* }
*
* 泛型形参列表：
* （1）泛型形参可能有多个
* 例如：Map<K,V>
*      ArrayList<E>
*      BiFunction<T,U,R>
*      Comparator<T>
* （2）泛型形参一般都是一个大写的字母
*  T：Type
*  K：key
*  V：value
*  E：Element元素的类型
*  R：返回值类型
*  U
```

```
* （2）泛型形参一般都是一个大写的字母
*  T：Type
*  K：key
*  V：value
*  E：Element元素的类型
*  R：返回值类型
*  U：类型，因为T用过了，那么换个字母
*  ...
* （3）泛型实参必须是引用数据类型，不能是基本数据类型
*
*  不支持基本数据类型的问题：
*  （1）集合
*  （2）泛型
*  等不支持基本数据类型。
```

```
*   （3）泛型实参必须是引用数据类型，不能是基本数据类型
*   如果是基本数据类型，请使用包装类。
*
*   不支持基本数据类型的问题：
*   （1）集合
*   （2）泛型
*   等不支持基本数据类型。
*
*   （4）泛型类或泛型接口上的泛型形参，可以在该类或该接口中当做
*   属性的类型、方法的形参类型、方法的返回值类型、局部变量的类型都可以。
*   但是不能用作“静态”成员的相关类型。
```

```
*   （4）泛型类或泛型接口上的泛型形参，可以在该类或该接口中当做
*   属性的类型、方法的形参类型、方法的返回值类型、局部变量的类型都可以。
*   但是不能用作“静态”成员的相关类型。
*
```

```
* （5）泛型形参什么时候具体化，即什么时候指定泛型实参？
* 第一种：创建对象，实例化
*  ArrayList<String> list = new ArrayList<String>();
*
* 第二种：在实现接口，或继承类的时候，可以把泛型实际化。
* class Student implements Comparable<Student>{
*    public int compareTo(Student stu){
*   }
* }
```

```
* （6）泛型形参还可以设置上限
* 【修饰符】class/interface 类型名<T extends 上限>{
* }
* 说明这个T只能指定为上限类型本身或它的子类
* 例如：class XueYuan<T extends Number>{
        private T score;//成绩
   }
* 这个T只能指定为Number或Number的子类（Integer,Double,Short...）
*/
```

```
* 二、泛型方法
* 1、为什么要声明泛型方法？
* （1）当该方法是一个静态方法，而该方法的形参类型或返回值类型不确定，那么可以单独为这个方法设计（声明）一个泛型形参。
* （2）当某个类不是泛型类，而它的某个非静态方法想要用泛型，也可以单独为这个方法设计（声明）泛型形参
* （3）当某个类是泛型类，但是某个非静态方法不想用类上的泛型形参，而是想要单独设计一个自己的泛型，那么也可以。
*
* 2、泛型方法的语法格式
* 【修饰符】<泛型形参类型列表> 返回值 方法名(【形参列表】)【throws 异常列表】{}
*
* 3、示例
* （1）java.util.Arrays
* public static <T> List<T> asList(T... a)
```

```
* 类型通配符：
* 当声明一个方法时，这个方法的形参是一个泛型类或泛型接口，但是此时又不确定如何指定这个泛型类的泛型实参，
* 例如：public void test(List list),这里的List是一个泛型接口，即这里无法确定List的泛型实参
*/
```

```
* 解决方法一：
*  该方法声明为一个泛型方法
*      public <T> void test(List<T> list){}
*
* 解决方法二：
*      public void test(List<?> list){}
*
* 方法一和方法二的区别：
* （1）元素的类型一个按照T，一个按照Object
* （2）List<T>不是只读
*    List<?>只读
*
* 2、可以设置?的上限
*  ? extends 上限
*
* ?的类型必须是上限本身或者上限的子类
*
* 3、可以设置?的下限
*  ? super 下限
```