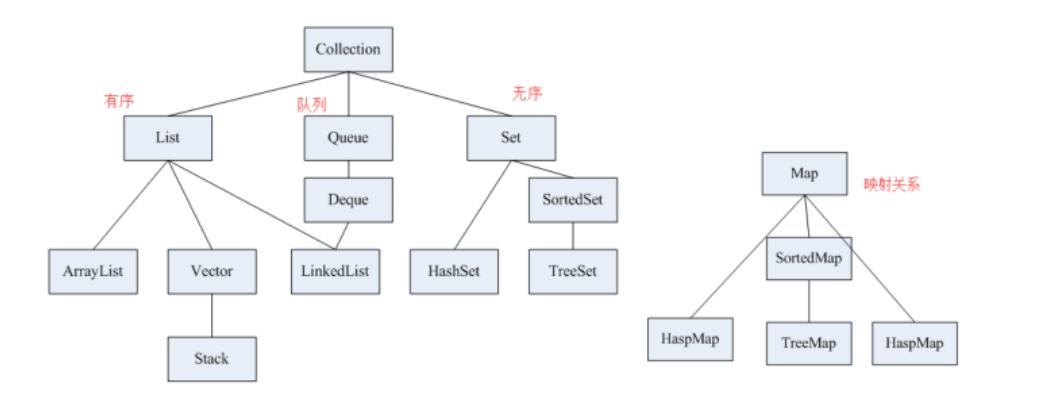
Collection和Map的继承体系图：



# Collection接口：

Collection的定义如下：

public interface Collection<E> extends Iterable<E> {}

**基础API接口**：

abstract boolean add(E object)

abstract boolean addAll(Collection<? extends E> collection)

abstract void clear()

abstract boolean contains(Object object)

abstract boolean isEmpty()

abstract boolean remove(Object object)

abstract boolean removeAll(Collection<?> collection)

abstract int size()

**abstract <T> T[] toArray(T[] array)**

abstract Object[] toArray()

**1.1 List接口**

List的定义如下：

public interface List<E> extends Collection<E> {}

// 相比与Collection，List**新增**的API：location 就是索引

abstract void                add(int location, E object)

abstract boolean             addAll(int location, Collection<? extends E> collection)

abstract E                   get(int location)

abstract int                 indexOf(Object object) //获取对象的第一个索引

abstract int                 lastIndexOf(Object object) //获取对象的最后一个索引

abstract E                   remove(int location)

abstract E                   set(int location, E object)

abstract List<E>             subList(int start, int end)

### 1.1.1 ArrayList

ArrayList定义如下：

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, java.io.Serializable

ArrayList是一个动态数组，它允许任何符合规则的元素插入甚至包括null。每一个ArrayList都有一个初始容量：

private static final int DEFAULT\_CAPACITY = 10;

**用到的基本上就是List 里的接口**

### 1.1.2 LinkedList

LinkedList定义如下：（LinkedList 是 Deque(Queue)的子类！！ 不要搞反）

public class LinkedList<E> extends AbstractSequentialList<E>  
implements List<E>, Deque<E>, Cloneable, java.io.Serializable

同样实现List接口的LinkedList与ArrayList不同，ArrayList是一个动态数组，而LinkedList是一个双向链表。所以它除了有LinkedList适合频繁地对列表进行增加或删除元素操作，因此LinkedList类可用于实现堆栈和队列

public void addFirst(Object o) //在链表头增添元素

public void addLast(Object o) //在链表尾增添元素

public E getFirst() //获取链表首元素

public E getLast() //获取链表尾元素

public Object removeFirst() //删除链表头元素，并返回该元素

public Object removeLast() //删除链表尾元素，并返回该元素

### 1.1.3 Vector（Stack）

Stack 是Vector的子类， java中Stack只有一个**无参构造函数。**

Stack<Character> chStack = **new** Stack<Character>();

Char作为泛型 应该用字符对象 不能用Stack<char>

**属于stack自己的方法包括**

push( num) //入栈

pop() //栈顶元素出栈 返回

**empty() //判定栈是否为空 注意这里和Collection的区别**

peek() //获取栈顶元素

search(num) //栈顶到该元素首次出现的位置的距离

## 1.2 Queue:

**Queue 和List 属于同一级，都继承了Collection**

**注意：Queue和Deque都是抽象接口，用LinkedList实例化！！**

Queue<Integer> que = **new** LinkedList<>();

Deque<Integer> deq = **new** LinkedList<>();

//add()和remove()，element()方法在失败的时候会抛出异常(不推荐)

Queue<String> queue = new LinkedList<String>();

**offer(object)**

**poll() // 出队并返回**

**peek()**

### 1.2.1 Deque 可以用LinkedList 代替使用



### 1.2.2 PriorityQueue

是Queue接口的实现，可以对其中元素进行排序，

可以放基本数据类型的**包装类**（如：Integer，Long等）或自定义的类

对于**基本数据类型的包装器类**，优先队列中元素默认排列顺序是**升序**排列

但对于自己定义的类来说，需要自己定义比较器

常用方法 和上面Queue一样

//定义类cmp,制定排序规则 降序 a-b > 0 升序 a-b<0 降序

class cmp implements Comparator<a>{

public int compare(a A, a B) {

if(A.n > B.n)return -1;

if(A.n == B.n)return 0;

if(A.n < B.n)return 1;

return 0;

}

}

PriorityQueue<**int**[]> minHeap = **new** PriorityQueue<>(cmp);

或者用表达式（上下这两个不是一个排序规则，只是表示这个形式）

PriorityQueue<**int**[]> minHeap = **new** PriorityQueue<>((a, b) -> (a[0] - b[0]));

## 1.3 Set

**跟List 平级，继承自Collection**

public void add(Object element)

public boolean remove(object element)

public boolean contains(Object o)

遍历方式一样可以用for 迭代器

### 1.3.1 HashSet TreeSet

**这两个接口用Set的 构造函数可以传一个Collection类型的进行初始化。**

HashSet(Collection<? extends E> collection)// 创建的HashSet包含collection

TreeSet(Collection<? extends E> collection) // 创建的TreeSet包含collection

对于TreeSet 可以定义 比较器

TreeSet()// 默认构造函数。使用该构造函数，TreeSet中的元素按照自然排序进行排列。

TreeSet(Comparator<? super E> comparator) // 指定TreeSet的比较器

# Map

public Object put(Object key,Object value) //增添元素

public Object remove(Object key) //删除元素,并返回键对应的值

public Object get(Object key) //获取键对应的值

public boolean containsKey(Object key) //判断指定键是否存在

public boolean containsValue(Object value) //判断指定值是否存在

put(n, m.getOrDefault(n, 0) + 1);

//获取键、值、元素集合

public Collection values() //获取值集合

public Set KeySet() //获取键集合

public Set entrySet() //获取元素集合

Map 接口没有提供 iterator() 方法，其子接口 Entry 提供了 iterator() 方法，并且提供了获取键、值的方法

遍历： Map<String, String> map = …

1. Entry<K,T> 遍历 可以取到每一个key和value

for(Map.Entry<String, String> entry : map.entrySet()){

String mapKey = entry.getKey();

String mapValue = entry.getValue();

System.out.println(mapKey+":"+mapValue);

}

1. 只用key 或者只用 value

//key

for(String key : map.keySet()){

System.out.println(key);

}

//value

for(String value : map.values()){

System.out.println(value);

}

**Pair 不要用 用 int[2]**

Pair<Integer, String> pair = new Pair<>(1, "One");

Integer key = pair.getKey();

String value = pair.getValue();

TreeMap: **比Map多的接口**

1. Object firstKey（）：它返回树映射中当前的第一个（最少）键。

2. Object lastKey（）：它返回树映射中当前的最后一个（最大）键。

3. Object ceilingKey（Object key）：返回大于或等于给定键的最小键，如果没有这样的键返回null。

4. Object higherKey（Object key）：返回严格大于指定键的最小键。

# 3.工具类（提供静态方法，宏观操作上面容器）：

## 3.1 Arrays （注意下面方法都是静态的）

Arrays.asList(Object[] a) 数组a 转换成 List

public static int binarySearch(Object[] a, Object key) //二分查找（a已排序）

public static boolean equals(Object[] a, Object[] a2) //判断两数组是否完全一致

public static void fill(Object[] a, Object val) //在a中所有位置填充val

//在[fromIndex,toIndex)中填充val

public static void fill(Object[] a, int fromIndex, int toIndex, Object val)

public static String toString(Object[] a) //将数组a转换为字符串，如"[1, 2, 3]"

public static void sort(Object[] a) //改进的快速排序（升序）

public static void sort(Object[] a, int fromIndex, int toIndex) //对[fromIndex,toIndex)升序排序

public static <T> void sort(T[] a, Comparator<? super T> c) //自定义比较器排序

### 自定义sort比较器

lambda表达式判断返回正负值： 对于（a,b）如果想让b排在a前面，那么用b去比较a，返回正值。

比如 对于数组想要升序：(a,b) -> a – b; 这样记：传进去 （1，2）-> 想让1在前，就返回1-2

同样，降序就是想让2在前：（a,b）-> b-a

二维排序如406题：

**int**[][] people

*// 按照一维降序，二维升序 排序*

Arrays.sort(people, (a, b)->(a[0] == b[0] ? a[1] - b[1] : b[0] - a[0]))

下面是字符串按字母序列倒排（179题），排位之后可以是：9，89，70，61，6，60，（注意6在60前面）

排位的效果就是， 把字符串组合起来是最大的数值（大的字符在前面高位）

 Arrays.sort(strList, **new** Comparator<String>() { // 重写排序的比较方法

            @Override

**public** **int** compare(String a, String b) {

**return** (b + a).compareTo(a + b);// *注意！！返回值必须是一对相反数，否则无效。*

            } // 比如1 -1

        });

或者用下面Lambda表达式更简洁的重写排序方法

Arrays.sort(strArray, (a, b) -> (b + a).compareTo(a + b));

3.2 Collections 类对**List对象**提供的方法

public static int binarySearch(List list, Object key) //查找元素

public static void copy(List dest, List src) //将src复制给dest

public static void fill(List list, Object obj) //在list中填充obj

public static void reverse(List list) //列表元素倒置

public static void sort(List list) //升序排序

public static <T> void sort(List<T> list, Comparator<? super T> c) //自定义比较器排序

# 4.String 类：

1. 因为String对象是不可变的，所以它们可以被共享。

String s1 = "abc";

String s2 = "abc";

// 内存中只有一个"abc"对象被创建，同时被s1和s2共享

1. "abc" 等效于 char[] data={ 'a' , 'b' , 'c' } 。

例如：

String str = "abc";

相当于：

char data[] = {'a', 'b', 'c'};

String str = new String(data);

// String底层是靠字符数组实现的。

方法：

**静态方法xx转换成String：String.valueOf(xx) xx 可以是基本数据类型，char[], object(相当于toString)**

boolean equals (Object anObject) ：将此字符串与指定对象进行比较。

boolean equalsIgnoreCase (String anotherString) ：将此字符串与指定对象进行比较，忽略大小写

int length () ：返回此字符串的长度。

String concat (String str) ：将指定的字符串连接到该字符串的末尾。

char charAt (int index) ：返回指定索引处的 char值。

int indexOf (String str) ：返回指定子字符串第一次出现在该字符串内的索引。

String substring (int beginIndex) ：从beginIndex开始截取字符串到字符串结尾。

String substring (int beginIndex, int endIndex) ：从[beginIndex,endIndex)截取字符串。

boolean contains(CharSequence s) 检测指定的子字符串是否存在。

 s = s.toLowerCase(); // 大写转换成小写,其他符号不受影响

public char[] toCharArray () ：将此字符串转换为新的字符数组。

//将与target匹配的字符串使 用replacement字符串替换。

public String replace (CharSequence target, CharSequence replacement)

public String[] split(String regex) ：将此字符串按照给定的regex（规则）拆分为字符串数组。

String[] v1 = version1.split("\\.");// 没有加转义符的话，返回长度为0

反转字符串

利用 StringBuffer 或 StringBuilder 的 reverse 成员方法:

// StringBuffer

public static String reverse1(String str) {

return new StringBuilder(str).reverse().toString();

}

# 4.1Character

下面方法都是Character的静态方法： static boolean

Charactter.isLetterOrDigit(char ch) ：判断字符ch是否为字母或数字。

Character.isLetter() 判断，这个字符是否为字母

Character.isDigit() 这个方法判断，这个字符是否为数字字符

Character.isWhitespace() 这个方法判断是否为空格

Character.isUpperCase() 这个方法判断是否为大写字母

Character.isLowerCase() 这个方法判断是否为小写字母

Character.toUpperCase() 这个方法是强转成大写字母

Character.toLowerCase() 这个方法是强转成小写字母

Character.toString() 这个方法是返回字符的字符串形式

# 5. StringBuilder

StringBuilder strB = new StringBuilder();

**往StringBuilder append的时候 可以直接插入 int 类型 自动转换成char**

**1、append(String str)/append(Char c)：字符串连接**  
**2、toString()：返回一个与构建起或缓冲器内容相同的字符串**  
**4、setCharAt(int i, char c)：将第 i 个代码单元设置为 c**

**5、insert(int offset, String str)/insert(int offset, Char c)：在指定位置之前插入字符(串)**  
**6、delete(int startIndex,int endIndex)：删除起始位置（含）到结尾位置（不含）之间的字符串**  
7. **删除最后一个字符 builder.deleteCharAt(builder.length() - 1)**

**8. 反转 StringBuilder reverse()**

**9.清空** sb.delete( 0, sb.length() );

10 sb的长度和通过索引值访问 : sb.length() ; sb.charAt(index)

今天在项目上遇到一个问题需要使用StringBuilder的比较，但是使用equals()总是发现错误。

经查询发现，原来是StringBuilder类的equals()没有重写内部其实还是使用了==的比较方法所以会出现问题。

解决办法，使用toString()将StringBuilder转换为String再进行相应比较。

StringBuilder数组使用

StringBuilder[] strs = new StringBuilder[numRows];

// 注意这里要new每一个sb,因为sb是容器，不是像string一样简单的数据类型

for (int i = 0 ; i < numRows; ++i) strs[i] = new StringBuilder();

# 6.Math: 静态方法

public static double abs(double a) ： 返回 double 值的绝对值。

public static double ceil(double a) 返回大于等于参数的最小的整数。

# 7.System:

浅拷贝: 主要用于对一个已有的数组int[] nums进行截取复制，复制出一个**左闭右开**区间的数组

return Arrays.copyOfRange(nums, 0, j);

深拷贝：将数组中指定的数据拷贝到另一个数组中。

static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)：

# 8.类型转换：

包装后的Integer.. 不要用 == 判断 用equals

## **8.1基本数据类型和字符串互相转换：**

**基本数据类型转换字符串：**

int num 转换string :

1. num + ""

2. String.valueOf(nums)

3. Integer.toString(num)

**字符串转基本数据类型:**

除了Character类之外，其他所有包装类都具有parseXxx静态方法可以将字符串参数转换为对应的基本类型（转换不了会抛出异常）：

public static int parseInt(String s)：将字符串参数转换为对应的int基本类型。

public static double parseDouble(String s)：将字符串参数转换为对应的double基本类型。

public static boolean parseBoolean(String s)：将字符串参数转换为对应的boolean基本类型。

如：

int num = Integer.parseInt("100");

## 8.2数组/List 转换

**数组转换成 动态list**  (**最好不要用！！）**

ans.add(**new** ArrayList<>(Arrays.asList(nums))); 报错

Line 10: error: no suitable constructor found for ArrayList(List<int[]>)

ans.add(new ArrayList<Integer>(Arrays.asList(nums)));

因为ArrayList里没有list类型的构造参数 但是Arrays.asList 返回的是list类型

        int[] data = {4, 5, 3, 6, 2, 5, 1};

### int[] 转 List<Integer>

List<Integer> list1 = Arrays.stream(data).boxed().collect(Collectors.toList());

        // Arrays.stream(arr) 可以替换成IntStream.of(arr)。

        // 1.使用Arrays.stream将int[]转换成IntStream。

        // 2.使用IntStream中的boxed()装箱。将IntStream转换成Stream<Integer>。

        // 3.使用Stream的collect()，将Stream<T>转换成List<T>，因此正是List<Integer>。

### int[] 转 Integer[]

        Integer[] integers1 = Arrays.stream(data).boxed().toArray(Integer[]::new);

        // 前两步同上，此时是Stream<Integer>。

        // 然后使用Stream的toArray，传入IntFunction<A[]> generator。

        // 这样就可以返回Integer数组。

        // 不然默认是Object[]。

### List<Integer> 转 Integer[]

        Integer[] integers2 = list1.toArray(new Integer[0]);

        //  调用toArray。传入参数T[] a。这种用法是目前推荐的。

        // List<String>转String[]也同理。

### List<Integer> 转 int[]

        int[] arr1 = list1.stream().mapToInt(Integer::valueOf).toArray();

        // 想要转换成int[]类型，就得先转成IntStream。

        // 这里就通过mapToInt()把Stream<Integer>调用Integer::valueOf来转成IntStream

        // 而IntStream中默认toArray()转成int[]。

### Integer[] 转 int[]

        int[] arr2 = Arrays.stream(integers1).mapToInt(Integer::valueOf).toArray();

        // 思路同上。先将Integer[]转成Stream<Integer>，再转成IntStream。

        // Integer[] 转 List<Integer>

        List<Integer> list2 = Arrays.asList(integers1);

### String[] 转 List<String>

        List<String> list3 = Arrays.asList(strings1);

### List<对象> 转 对象[ ]

如**String**：

String[] strings2 = list3.toArray(new String[0]);

如**int[][]**:

List<int[]> ans = new LinkedList<>();

int[][] tmp = ans.toArray(**new** **int**[0][]);*// list 转成 数组*