# 集合概述体系、常用API

## 前提引入

- 数组特点

数组定义完成后，类型确定、长度固定。
适合元素的个数和类型确定的业务场景，不适合做需要增删数据操作。

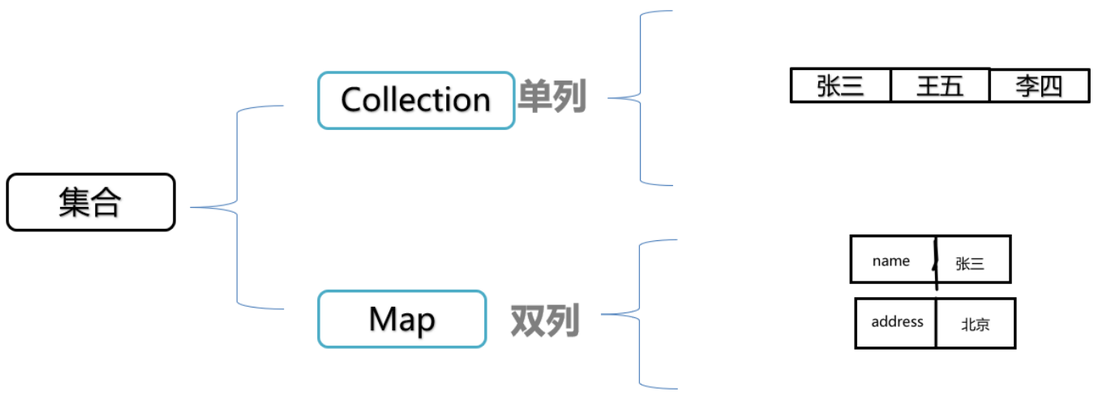
- 集合的特点

集合的大小不固定，启动后可以动态变化，类型也可以选择不固定
集合非常适合做元素的增删操作

- 集合数组对比

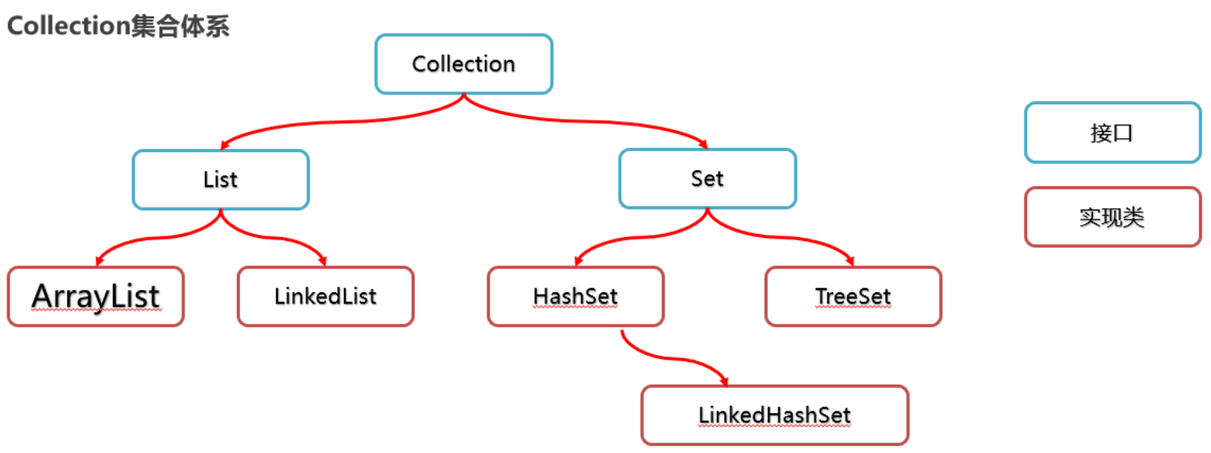
1.数组和集合的元素存储的个数问题。
　数组定义后类型确定，长度固定
　集合类型可以不固定，大小是可变的。
2、数组和集合存储元素的类型问题。
　数组可以存储基本类型和引用类型的数据。
　集合只能存储引用数据类型的数据。（包装类）
3、数组和集合适合的场景
　数组适合做数据个数和类型确定的场景。
　集合适合做数据个数不确定，且要做增删元素的场景。

## 两大种集合




Collection：单列集合，每个元素（数据）只包含一个值。
Map：双列集合，每个元素包含两个值（键值对）

- **collection集合体系**




List系列集合：添加的元素是有序、可重复、有索引。
　ArrayList、LinekdList ： 有序、可重复、有索引。
Set系列集合：添加的元素是无序、不重复、无索引。
　HashSet: 无序、不重复、无索引；

LinkedHashSet: 【有序】、不重复、无索引。
TreeSet：按照大小默认【升序排序】、不重复、无索引。

```
/**
 * ClassName: ListDemo
 * Description:List接口特点:有序，可重复，有索引
 * date:2022/3/11
 *
 * @author fgcy
 * @since JDK 1.8
 */

@Test
public void test1() {
    Collection list = new ArrayList<>();
    list.add('a');
    list.add(97);
    list.add(3.141592653578);
    list.add(true);
    list.add("java");
    list.add("java");
    System.out.println(list);
}
```

```
[a, 97, 3.141592653578, true, java, java]
```

```
/**
 * ClassName: ListDemo
 * Description:set接口特点:无序，不重复，无索引
 * date:2022/3/11
 *
 * @author fgcy
 * @since JDK 1.8
 */

@Test
public void test() {
    Collection set = new HashSet();
    set.add("java");
    set.add("java");
    set.add('a');
    set.add(97);
    set.add(true);
    System.out.println(set);
}
```

```
[a, 97, java, true]
```

## 集合对于泛型的支持

集合都是支持泛型的，可以在编译阶段约束集合只能操作某种数据类型（可以通过反射和通过赋值无泛型的引用越过泛型约束）
集合和泛型都只能支持引用数据类型，不支持基本数据类型，所以集合中存储的元素都认为是对象
集合中要存储基本类型的数据,需要将基本数据类型转换为包装类；

```
    @Test
    public void test2() {
        final Collection list1 = new ArrayList<String>();
        final Collection list2 = new ArrayList<>();// JDK 1.7开始后面的泛型类型申明可以省略不写
        final Collection<String> list3 = new ArrayList();//建议加上尖括号，规范
//        final Collection<int> list = new ArrayList<>();//集合与泛型不支持基本数据类型，可以使用包装类
    }
```

- 总结

1、单列集合的代表是?
Collection接口。
2、Collection集合分了哪2大常用的集合体系?
List系列集合：添加的元素是有序、可重复、有索引。
Set系列集合：添加的元素是无序、不重复、无索引。
3、如何约定集合存储数据的类型，需要注意什么?
集合支持泛型。
集合和泛型不支持基本类型，只支持引用数据类型。

## Collection常用API

Collection是单列集合的祖宗接口，它的功能是全部单列集合都可以继承使用的。

---

### Collection API如下:

| 方法名称 | 说明 |
| --- | --- |
| public boolean add(E e) | 把给定的对象添加到当前集合中 |
| public void clear() | 清空集合中所有的元素 |
| public boolean remove(E e) | 把给定的对象在当前集合中删除 |
| public boolean contains(Object obj) | 判断当前集合中是否包含给定的对象 |
| public boolean isEmpty() | 判断当前集合是否为空 |
| public int size() | 返回集合中元素的个数。 |
| public Object[] toArray() | 把集合中的元素，存储到数组中 |

```
    @Test
    public void test1() {
        Collection<String> list = new ArrayList<>();
        System.out.println("添加元素" + list.add("java"));//返回真假，一般来说除了set有不能重复的限制外，其他一般为
true
        list.add("java");
        list.add("css");
        list.add("phthon");
        list.add("javascript");
        list.add("c#");
        System.out.println("目前集合元素个数" + list.size());//目前集合中的元素个数
        System.out.println("移除元素" + list.remove("css"));//是否删除成功
        System.out.println("包含元素" + list.contains("java"));//是否包含某个元素
        final Object[] array = list.toArray();
        System.out.println(Arrays.toString(array));
        System.out.println("集合是否为空" + list.isEmpty());
        list.clear();//清空集合
        System.out.println("集合是否为空" + list.isEmpty());
        System.out.println("----------------------------");
        Collection<String> set = new HashSet<>();
        Collection<String> set2 = new HashSet<>();
        System.out.println("添加元素" + set.add("java"));
        System.out.println("添加元素" + set.add("java"));
        set.add("phthon");
        set2.add("c++");
        set2.add("c#");

        set.addAll(set2);//把set2中的元素拷贝到set中
        System.out.println(set);
        System.out.println(set2);
    }
```

```
添加元素true
目前集合元素个数6
移除元素true
包含元素true
[java, java, phthon, javascript, c#]
集合是否为空false
集合是否为空true
----------------------------
添加元素true
添加元素false
[c#, c++, java, phthon]
[c#, c
```

## Collection集合存储自定义类型的对象

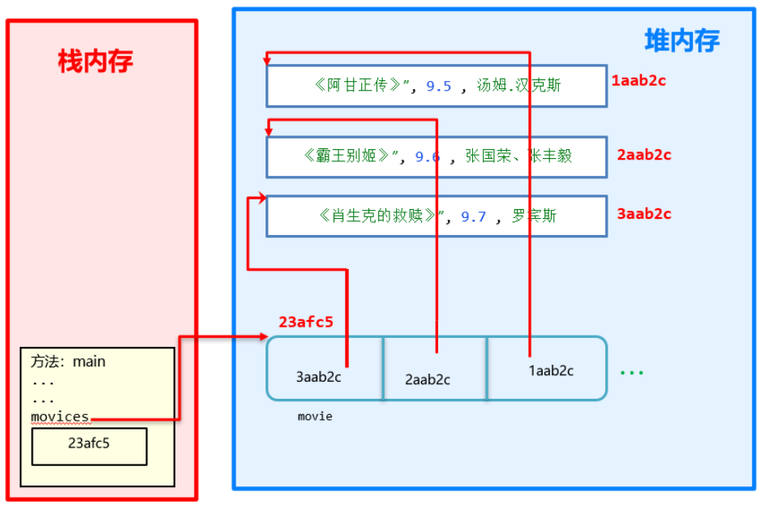
```
public class Movie {
    private String name;
    private double score;
    private String acotr;

    public Movie(String name, double score, String acotr) {
        this.name = name;
        this.score = score;
        this.acotr = acotr;
    }
    // ... getter + setter
}
```

```
public class SystemDemo {
    public static void main(String[] args) {
        Collection <Movie> movies = new ArrayList<>();
        movies.add(new Movie("《肖生克的救赎》", 9.7 , "罗宾斯"));
        movies.add(new Movie("《霸王别姬》", 9.6 , "张国荣、张丰毅"));
        movies.add(new Movie("《阿甘正传》", 9.5 , "汤姆.汉克斯"));
        System.out.println(movies);

        for (Movie movie : movies) {

            System.out.println("片名: " + movie.getName());
            System.out.println("评分: " + movie.getScore());
            System.out.println("主演: " + movie.getAcotr());
        }
    }
}
```




集合中存储的是元素对象的地址

# 集合迭代

## 第一种遍历方式迭代器

迭代器是集合的专用遍历方式
遍历就是一个一个的把容器中的元素访问一遍。
迭代器在Java中的代表是Iterator

### Collection集合获取迭代器

| 方法名称 | 说明 |
| --- | --- |
| `Iterator<E> iterator()` | 返回集合中的迭代器对象，该迭代器对象默认指向当前集合的0索引 |

### Iterator中的常用方法

| 方法名称 | 说明 |
| --- | --- |
| `boolean hasNext()` | 询问当前位置是否有元素存在，存在返回true ,不存在返回false |
| `E next()` | 获取当前位置的元素，并同时将迭代器对象移向下一个位置，注意防止取出越界。 |

```
    @Test
    public void test2() {
        Collection list = new ArrayList<>();
        list.add("a");
        list.add("b");
        list.add("c");
        final Iterator iterator = list.iterator();//获取该集合的迭代器，并指向第一个元素
/*          System.out.println(iterator.next());//取出第一个元素，并移到下一个位置
        System.out.println(iterator.next());
        System.out.println(iterator.next());
        System.out.println(iterator.next());//异常NoSuchElementException*/

        while (iterator.hasNext()) {//看一下当前位置是否有元素
            System.out.println(iterator.next());//取出当前位置的元素值，并把指向移到下一位
        }
    }
```

- 小结

1、迭代器的默认位置在哪里。
Iterator<E> iterator()：得到迭代器对象，默认指向当前集合的索引0

2、迭代器如果取元素越界会出现什么问题。
会出现NoSuchElementException异常。

## 第二种遍历方式foreach也叫增强for循环

增强for循环：既可以遍历集合也可以遍历数组

格式：

```
    for(元素数据类型 变量名 : 数组或者Collection集合) {
            //在此处使用变量即可，该变量就是元素
    }
```

修改第三方变量的值不会影响到集合中的元素

## 方式三：lambda表达式

Lambda表达式，提供了一种更简单、更直接的遍历集合的方式。
集合才拥有Lambda表达式的遍历方法，数组没有

| 方法名称 | 说明 |
| --- | --- |
| `default void forEach(Consumer<? super T> action):` | 结合lambda遍历集合 |

- 源码


```
    @Test
    public void test6() {
        final int[] ints = {1, 2, 32, 5, -4, 0, 21};
        Collection<String> list = new ArrayList<>();
        Collections.addAll(list, "spring", "springframe", "mybatis", "hibernate", "mysql");

        list.forEach(new Consumer<String>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        });//匿名内部类
//        list.forEach(s -> System.out.println(s));//Lanbda表达式简化写法
//        list.forEach(System.out::println);//方法引用 Lambda表达式的简化写法，要求将接收到的参数直接输出sout
    }
```

```
spring
springframe
mybatis
hibernate
mysql
```

# 常用数据结构

## 数据结构概述、栈、队列

- 数据机构概述

数据结构是计算机底层存储、组织数据的方式。是指数据相互之间是以什么方式排列在一起的。
通常情况下，精心选择的数据结构可以带来更高的运行或者存储效率

- 常见的数据结构

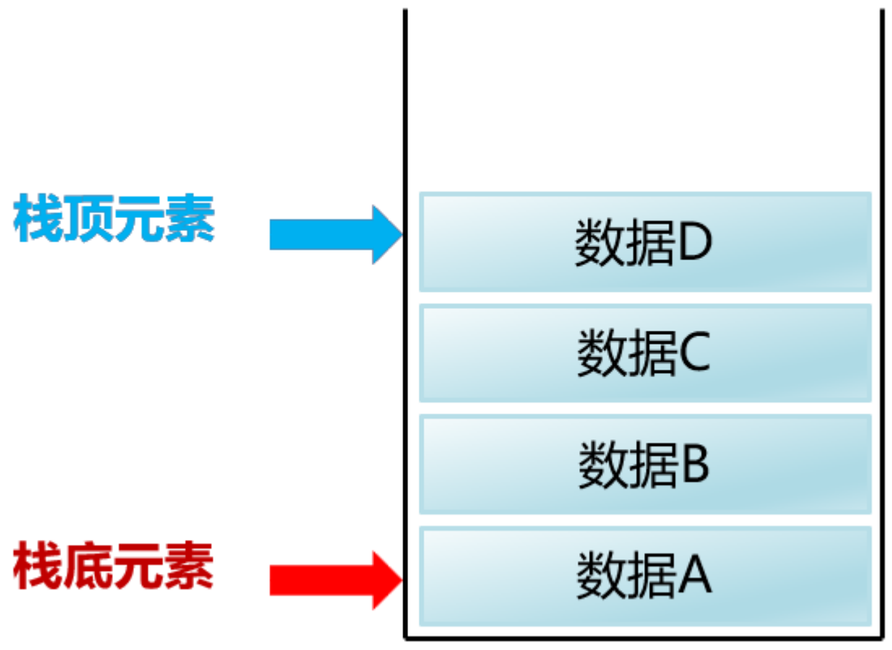
栈
队列
数组
链表
二叉树
二叉查找树
平衡二叉树
红黑树
....

- **栈数据结构**

特点：
　后进先出，先进后出

数据进入栈模型的过程称为：压/进栈
数据离开栈模型的过程称为：弹/出栈

---




---

- **队列**

先进先出，后进后出
数据从后端进入队列模型的过程称为：入队(列)
数据从前端离开队列模型的过程称为：出队(列)

---

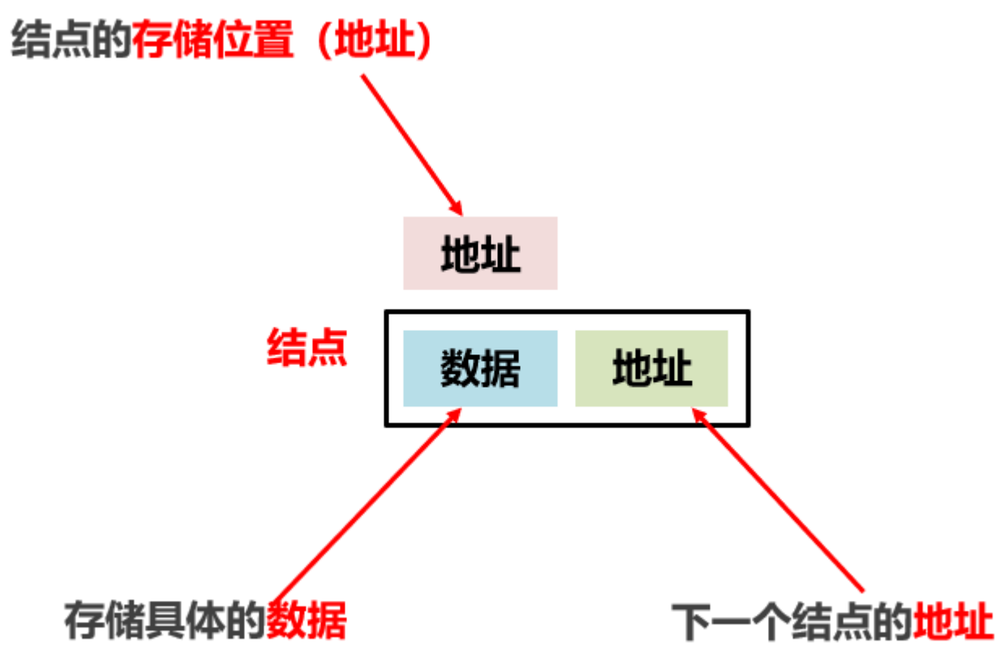

- **数组**

查询速度快：查询数据通过地址值和索引定位，查询任意数据耗时相同。（元素在内存中是连续存储的）随机存储存取;通过索引查询的速度快
删除效率低：要将原始数据删除，同时后面每个数据前移。
添加效率极低：添加位置后的每个数据后移，再添加元素。

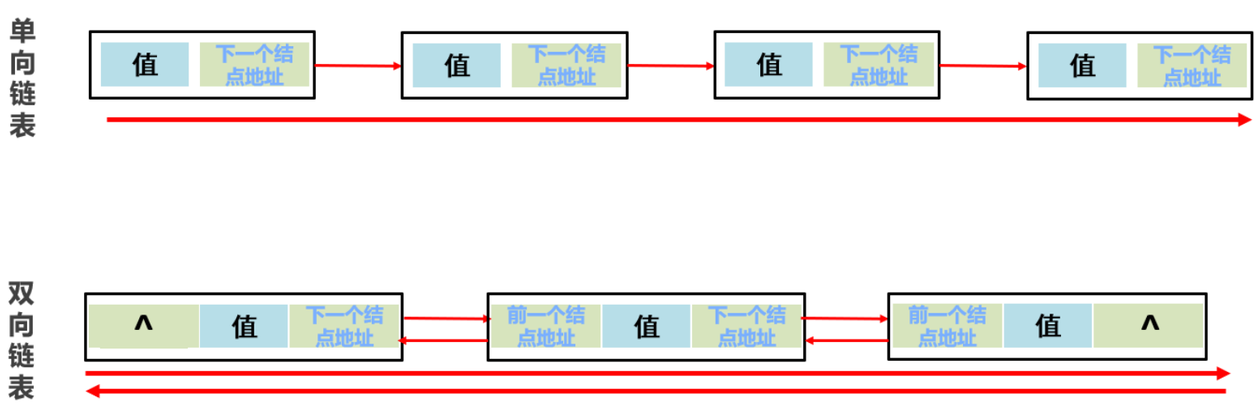
- 链表

链表中的元素是在内存中不连续存储的，每个元素节点包含数据值和下一个元素的地址。
链表查询慢。无论查询哪个数据都要从头开始找(对比数组)
链表增删相对快(对比数组)

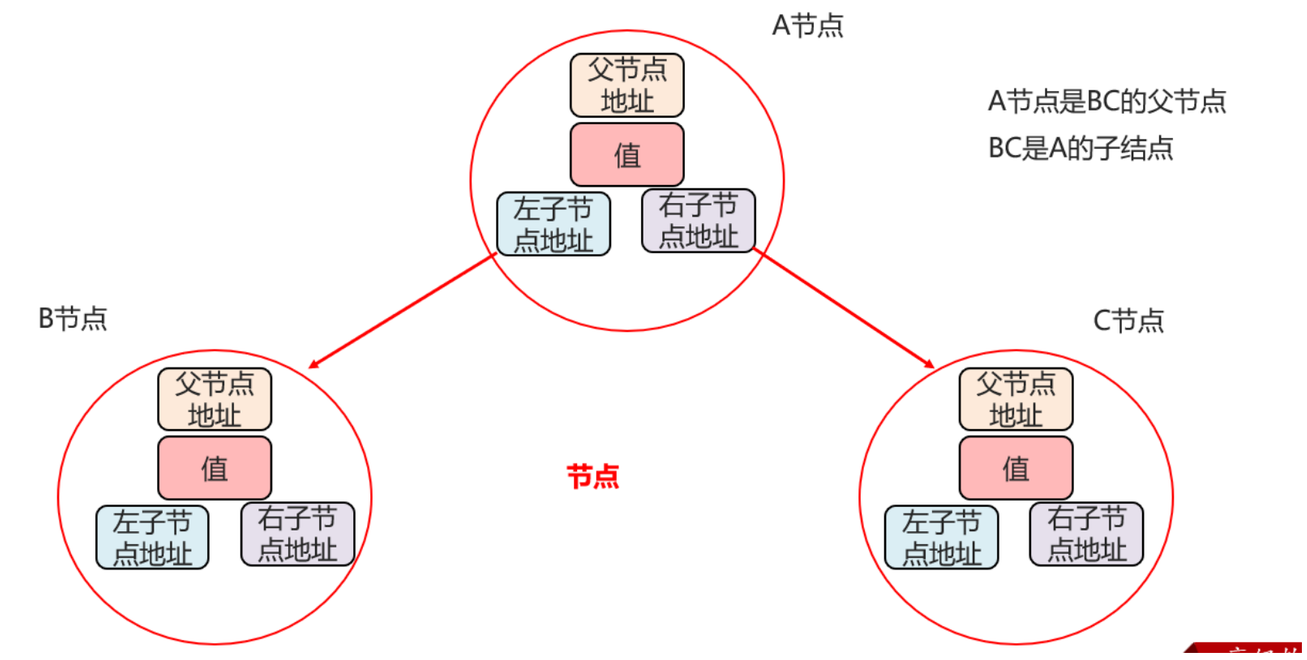
- 链表的结构




- 链表的种类




- 二叉树




- 二叉树的特点

只能有一个根节点，每个节点最多支持2个直接子节点。
节点的度： 节点拥有的子树的个数，二叉树的度不大于2 叶子节点 度为0的节点，也称之为终端结点。
高度：叶子结点的高度为1，叶子结点的父节点高度为2，以此类推，根节点的高度最高。
层：根节点在第一层，以此类推
兄弟节点 ： 拥有共同父节点的节点互称为兄弟节点

- 二叉查找树

二叉查找树又称二叉排序树或者二叉搜索树

特点：
1，每一个节点上最多有两个子节点
2，左子树上所有节点的值都小于根节点的值
3，右子树上所有节点的值都大于根节点的值

作用：
提高检索数据的性能

- 平衡二叉树

二叉查找树出现的问题：
出现瘸子现象(数据全存在树的一边)，导致查询的性能与单链表一样，查询速度变慢

平衡二叉树是在满足查找二叉树的大小规则下，让树尽可能矮小，以此提高查数据的性能。

平衡二叉树的要求：
任意节点的左右两个子树的高度差不超过1，任意节点的左右两个子树都是一颗平衡二叉树

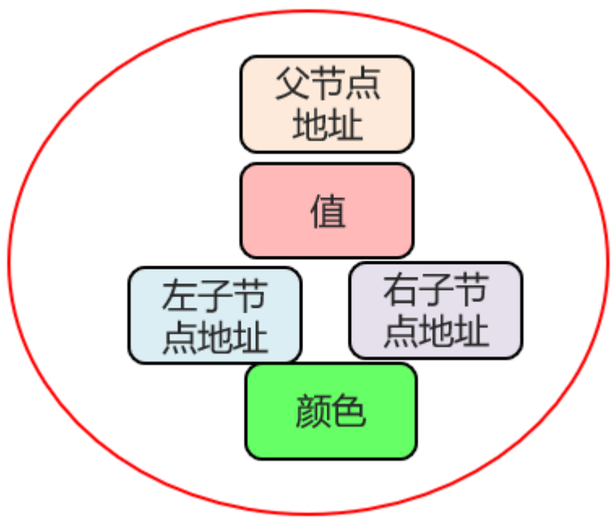
- 红黑树

红黑树是一种自平衡的二叉查找树，是计算机科学中用到的一种数据结构。
1972年出现，当时被称之为平衡二叉B树。1978年被修改为如今的"红黑树"。
每一个节点可以是红或者黑；红黑树不是通过高度平衡的，它的平衡是通过“红黑规则”进行实现的

- 红黑规则

每一个节点或是红色的，或者是黑色的，根节点必须是黑色。
如果某一个节点是红色，那么它的子节点必须是黑色(不能出现两个红色节点相连的情况)。
对每一个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点
添加的节点的颜色.默认用红色效率高

- 红黑树结构图




- 红黑树小结

红黑树增删改查的性能都很好
红黑树不是高度平衡的，它的平衡是通过"红黑规则"进行实现的

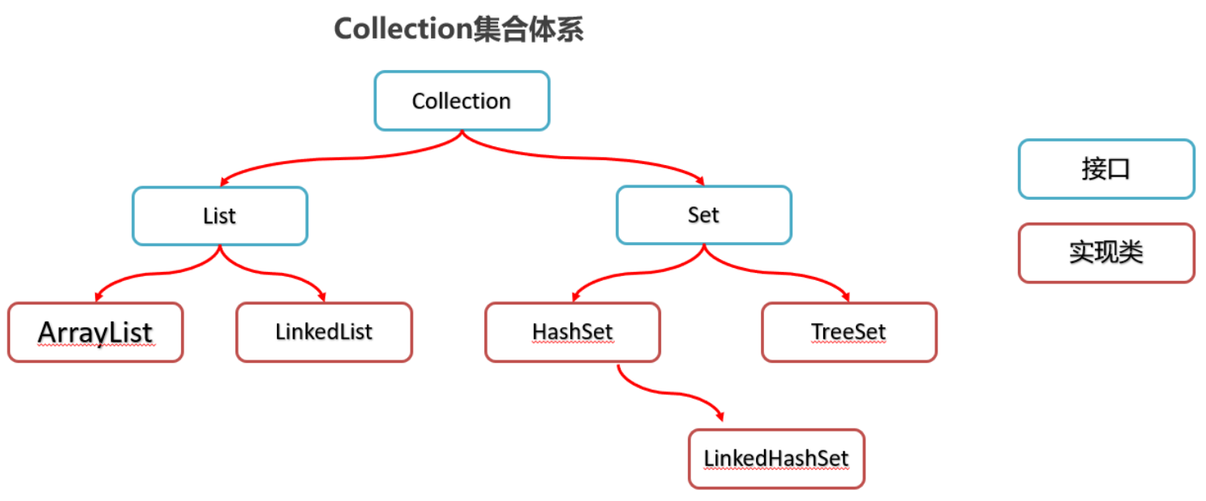
- **各种数据结构的特点和作用**

队列：先进先出，后进后出。
栈：后进先出，先进后出。
数组：内存连续区域，查询快，增删慢。
链表：元素是游离的，查询慢，首尾操作极快。
二叉树：永远只有一个根节点, 每个结点不超过2个子节点的树。

查找二叉树：小的左边，大的右边，但是可能树很高，查询性能变差。
平衡查找二叉树：让树的高度差不大于1，增删改查都提高了。
红黑树（就是基于红黑规则实现了自平衡的排序二叉树）

# List系列集合

## List集合特点、特有API




List系列集合特点:
ArrayList、LinekdList：有序，可重复，有索引。
有序：存储和取出的元素顺序一致
有索引：可以通过索引操作元素
可重复：存储的元素可以重复

List的实现类的底层原理
ArrayList底层是基于数组实现的，根据查询元素快，增删相对慢。(随机存储存取)
LinkedList底层基于双链表实现的，查询元素慢，增删首尾元素是非常快的。

List集合因为支持索引，所以多了很多索引操作的独特api，其他Collection的功能List也都继承了

- **List集合特有API**

| 方法名称 | 说明 |
| --- | --- |
| void add(int index,E element) | 在此集合中的指定位置插入指定的元素 |
| E remove(int index) | 删除指定索引处的元素，返回被删除的元素 |
| E set(int index,E element) | 修改指定索引处的元素，返回被修改的元素 |
| E get(int index) | 返回指定索引处的元素 |

**Collection API如下:**

| 方法名称 | 说明 |
| --- | --- |
| public boolean add(E e) | 把给定的对象添加到当前集合中 |
| public void clear() | 清空集合中所有的元素 |
| public boolean remove(E e) | 把给定的对象在当前集合中删除 |
| public boolean contains(Object obj) | 判断当前集合中是否包含给定的对象 |
| public boolean isEmpty() | 判断当前集合是否为空 |
| public int size() | 返回集合中元素的个数。 |
| public Object[] toArray() | 把集合中的元素，存储到数组中 |

```
@Test
public void test9() {
    List<String> list = new ArrayList<>();
    list.add("java");//Collection
    list.add("java");//Collection
    list.add("springmvc");//Collection
    list.add("spring");//Collection
    list.add("mongodb");//Collection
    System.out.println(list);
    System.out.println(list.get(2));//List
    System.out.println(list.remove(0));//List
    System.out.println(list);
    list.set(1, "SpringMVC");//List
    System.out.println(list.contains("SpringMVC"));//Collection
    System.out.println(list.size());//Collection
```

```
        System.out.println(list.remove("spring"));//Collection
        System.out.println(list);
        final Object[] array = list.toArray();//Collection
        System.out.println(Arrays.toString(array));//Collection
        list.clear();//Collection
        System.out.println(list.isEmpty());//Collection
        System.out.println(list.size());//Collection
    }
```

```
[java, java, springmvc, spring, mongodb]
springmvc
java
[java, springmvc, spring, mongodb]
true
4
true
[java, SpringMVC, mongodb]
[java, SpringMVC, mongodb]
true
0
```

## List集合的遍历方式小结

1.迭代器
2.增强for循环
3.Lambda表达式
4.for循环（因为List集合存在索引）

1. 迭代器

```
    @Test
    public void test12() {
        List<String> list = new ArrayList<>();
        Collections.addAll(list, "Mybatis", "hibernate", "ObjectOrientedProgram", "redis", "tomcat");
        final Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
```

```
Mybatis
hibernate
ObjectOrientedProgram
redis
tomcat
```

2. 增强for（foreach）

```
    @Test
    public void test12() {
        List<String> list = new ArrayList<>();
        Collections.addAll(list, "Mybatis", "hibernate", "ObjectOrientedProgram", "redis", "tomcat");
        for (String s : list) {
            System.out.println(s);
        }
    }
```

```
Mybatis
hibernate
ObjectOrientedProgram
redis
tomcat
```

3. Lambda表达式

```
    @Test
    public void test12() {
        List<String> list = new ArrayList<>();
        Collections.addAll(list, "Mybatis", "hibernate", "ObjectOrientedProgram", "redis", "tomcat");
//        list.forEach(System.out::println);//方法引用
//        list.forEach(s -> System.out.println(s));//Lambda表达式
        list.forEach(new Consumer<String>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        });
    }
```

```
Mybatis
hibernate
ObjectOrientedProgram
redis
tomcat
```

4. for循环

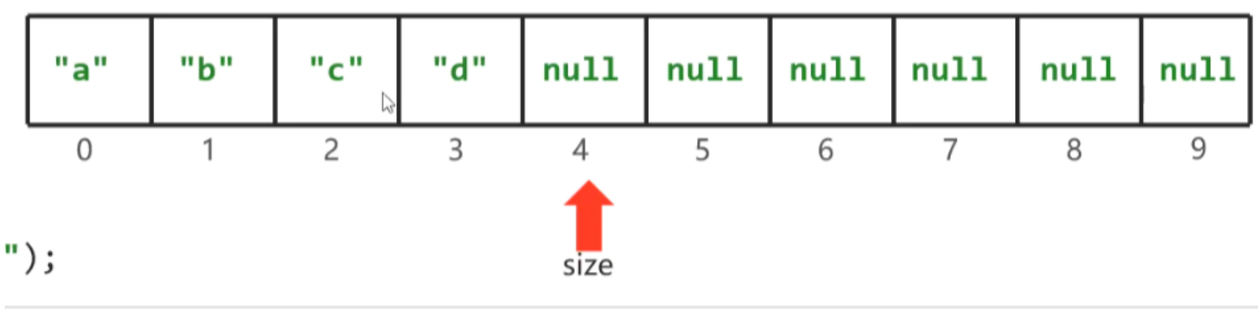
```
    @Test
    public void test12() {
        List<String> list = new ArrayList<>();
        Collections.addAll(list, "Mybatis", "hibernate", "ObjectOrientedProgram", "redis", "tomcat");
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
    }
```

```
Mybatis
hibernate
ObjectOrientedProgram
redis
tomcat
```

## ArrayList集合的底层原理

ArrayList底层是基于数组实现的：根据索引定位元素快，增删需要做元素的移位操作。
第一次创建集合并添加第一个元素的时候，在底层创建一个默认长度为10的数组。
List集合的size和下一个元素插入的位置----------》当前元素的下一个位置（当前元素索引加一）
当插入元素的位置与当前集合大小相等，就会扩容 >>1;

---




");

---

## LinkedList集合的底层原理

底层数据结构是双链表，查询慢，首尾操作的速度是极快的，所以多了很多首尾操作的特有API

---

## LinkedList集合的特有功能

| 方法名称 | 说明 |
| --- | --- |
| public void addFirst(E e) | 在该列表开头插入指定的元素 |
| public void addLast(E e) | 将指定的元素追加到此列表的末尾 |
| public E getFirst() | 返回此列表中的第一个元素 |
| public E getLast() | 返回此列表中的最后一个元素 |
| public E removeFirst() | 从此列表中删除并返回第一个元素 |
| public E removeLast() | 从此列表中删除并返回最后一个元素 |

---

```
/**
 * ClassName: ListDemo
 * Description:LinkedList实现栈
 * date:2022/3/12
 *
 * @author fgcy
 * @since JDK 1.8
 */

@Test
public void test13() {
    LinkedList<String> linkedList = new LinkedList<>();
    linkedList.push("第1");//入栈 (内部封装了addFirst)
    linkedList.addFirst("第2");//在首位加入元素
    linkedList.addFirst("第3");//在首位加入元素
    linkedList.addFirst("第4");//在首位加入元素
    linkedList.addFirst("第5");//在首位加入元素
    linkedList.addFirst("第6");//在首位加入元素
    System.out.println(linkedList);
    System.out.println(linkedList.getFirst());
    System.out.println(linkedList.pop());//出栈(内部封装了removeFirst)
    System.out.println(linkedList.removeFirst());//取出并获取第二个元素
    System.out.println(linkedList.removeFirst());//取出并获取第三个元素
    System.out.println(linkedList.removeFirst());//取出并获取第四个元素
    System.out.println(linkedList.removeFirst());//取出并获取第五个元素
    System.out.println(linkedList.removeFirst());//取出并获取第六个元素
}
```

```
[第6, 第5, 第4, 第3, 第2, 第1]
第6
第6
第5
第4
第3
第2
第1
```

```
/**
 * ClassName: ListDemo
```

```
 * Description:LinkedList实现队列
 * date:2022/3/12
 *
 * @author fgcy
 * @since JDK 1.8
 */

@Test
public void test14() {
    LinkedList linkedList = new LinkedList();
    linkedList.addLast("第一个");//入队
    linkedList.addLast("第二个");//入队
    linkedList.addLast("第三个");//入队
    linkedList.addLast("第四个");//入队
    linkedList.addLast("第五个");//入队
    System.out.println(linkedList);
    System.out.println(linkedList.getFirst());//获取队头第一个元素
    System.out.println(linkedList.removeFirst());//出队
    System.out.println(linkedList.removeFirst());//出队
    System.out.println(linkedList.removeFirst());//出队
    System.out.println(linkedList.removeFirst());//出队
    System.out.println(linkedList.removeFirst());//出队
    System.out.println(linkedList);
}
```

```
[第一个, 第二个, 第三个, 第四个, 第五个]
第一个
第二个
第三个
第四个
第五个
[]
```

- **LinkedList是一种双向列表**


## 集合的并发修改异常问题

问题引出：
　当我们从集合中找出某个元素并删除的时候可能出现一种并发修改异常问题
如：
　迭代器遍历集合且直接用集合删除元素的时候可能出现。
　增强for循环遍历集合且直接用集合删除元素的时候可能出现（底层是foreach）

哪种遍历且删除元素不出问题:
　迭代器遍历集合但是用迭代器自己的删除方法操作可以解决。
　使用for循环遍历并删除元素不会存在这个问题

1. 迭代器

```
 * ClassName: ListDemo
 * Description:迭代器遍历删除元素，会出现并发修改的问题；可以解决
 * date:2022/3/12
 *
 * @author fgcy
 * @since JDK 1.8
 */

@Test
public void test15() {
    List<String> list = new ArrayList<>();
    Collections.addAll(list, "spring", "spring", "jpa", "oracle", "mongodb");
    System.out.println(list);
    final Iterator<String> iterator = list.iterator();
    while (iterator.hasNext()) {
        if (Objects.equals("spring", iterator.next())) {
            //会出现并发修改异常（漏删元素）
//              list.remove("spring");//java.util.ConcurrentModificationException
            iterator.remove();//删除元素后会把索引减一，保证不会越过元素
        }
    }
    System.out.println(list);
}
```

```
[spring, spring, jpa, oracle, mongodb]
[jpa, oracle, mongodb]
```

2. foreach

```
/**
 * ClassName: ListDemo
 * Description:foreach遍历删除元素，会出现并发修改的问题；不可以解决
 * date:2022/3/12
 *
 * @author fgcy
 * @since JDK 1.8
 */
@Test
public void test16() {
    List<String> list = new ArrayList<>();
    Collections.addAll(list, "spring", "spring", "jpa", "oracle", "mongodb");
    System.out.println(list);
    for (String s : list) {
```

```
            if (Objects.equals(s, "spring")) {
                //会出现并发修改异常（漏删元素）不能解决
                list.remove("spring");//java.util.ConcurrentModificationException
            }
        }
        System.out.println(list);
    }
```

3. Lambda

```
    /**
     * ClassName: ListDemo
     * Description:Lambda表达式foreach遍历删除元素，会出现并发修改的问题；不可以解决
     * date:2022/3/12
     *
     * @author fgcy
     * @since JDK 1.8
     */
    @Test
    public void test16() {
        List<String> list = new ArrayList<>();
        Collections.addAll(list, "spring", "spring", "jpa", "oracle", "mongodb");
        System.out.println(list);
        list.forEach(s -> {
            if (Objects.equals(s, "spring")) {
                list.remove("spring");
            }
        });
        System.out.println(list);
    }
```

4. for循环

```
    /**
     * ClassName: ListDemo
     * Description:for遍历删除元素，会出现漏删的问题，不会报错；可以解决
     * date:2022/3/12
     *
     * @author fgcy
     * @since JDK 1.8
     */
    @Test
    public void test17() {
        List<String> list = new ArrayList<>();
        Collections.addAll(list, "spring", "spring", "jpa", "oracle", "mongodb");
        System.out.println(list);
/*          //第一种方法：从后往前删
        for (int i = list.size() - 1; i >= 0; i--) {
            if (Objects.equals("spring", list.get(i))) {
                list.remove(i);
            }
        }*/
        //第二种方法，成功删除元素后，将索引值减一
        for (int i = 0; i < list.size(); i++) {
            if (Objects.equals("spring", list.get(i))) {
                list.remove(i);
                i--;
            }
        }
        System.out.println(list);
    }
```

```
[spring, spring, jpa, oracle, mongodb]
[jpa, oracle, mongodb]
```

# 泛型深入

## 泛型的概述和优势

泛型概述：
泛型：是JDK5中引入的特性，可以在编译阶段约束操作的数据类型，并进行检查。
泛型的格式：<数据类型>; 注意：泛型只能支持引用数据类型。
集合体系的全部接口和实现类都是支持泛型的使用的。

泛型的好处：
统一数据类型。
把运行时期的问题提前到了编译期间，避免了强制类型转换可能出现的异常，因为编译阶段类型就能确定下来

泛型可以在很多地方进行定义：
类后面----------》泛型类
方法申明上----------》泛型方法
接口后面----------》泛型接口

## 自定义泛型类

- 泛型类概述

定义类时同时定义了泛型的类就是泛型类。
泛型类的格式：修饰符 class 类名<泛型变量>{}
此处泛型变量可以随便写为任意标识，常见的如E、T、K、V等

- 作用

编译阶段可以指定数据类型，类似于集合的作用

- **泛型类的原理**

把出现泛型变量的地方全部替换成传输的真实数据类型

```
public class MyArrayList<E> {
    private List list = new ArrayList();
```

```
    //这个不是泛型方法，这里只是使用类定义的泛型变量
    public boolean add(E e) {
        return list.add(e);
    }

    public boolean remove(E e) {
        return list.remove(e);
    }

    @Override
    public String toString() {
        return list.toString();
    }
}
```

```
class Test {
    public static void main(String[] args) {
        final MyArrayList<String> my = new MyArrayList<>();
        System.out.println(my.add("123"));
        System.out.println(my.add("456"));
        System.out.println(my.add("java"));
        System.out.println(my.add("spring"));
        System.out.println(my.add("mybatis"));
        System.out.println(my);
        System.out.println(my.remove("java"));
        System.out.println(my);
    }
}
```

```
true
true
true
true
true
[123, 456, java, spring, mybatis]
true
[123, 456, spring, mybatis]
```

## 自定义泛型方法

- 泛型方法概述

定义方法时同时定义了泛型的方法就是泛型方法。

泛型方法的格式：修饰符 <泛型变量> 方法返回值 方法名称(形参列表){}
  泛型变量定义在返回值之前

范例：
  public  void show(T t) {}

作用：
  方法中可以使用泛型接收一切实际类型的参数，方法更具备通用性

泛型方法的原理：
  把出现泛型变量的地方全部替换成传输的真实数据类型

```
public class GenericMethodDemo {

    public static void main(String[] args) {
        final String[] strings = {"阿黄", "阿雄", "阿刘"};
        printArrays(strings);
        printArrays(shuffleArray(strings));

        final Integer[] ints = {11, 52, 1, 47, -47, 0, 3};
        printArrays(ints);
        printArrays(shuffleArray(ints));
    }


    public static <T> void printArrays(T[] arr) {
        if (arr == null) {
            System.out.println("null");
            return;
        }
        final StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < arr.length; i++) {
            sb.append(arr[i]).append(i == arr.length - 1 ? "" : ",");
        }
        sb.append("]");
        System.out.println(sb);
    }
```

```
    /**
     * ClassName: GenericMethodDemo
     * Description:返回值类型就是泛型
     * date:2022/3/12
     *
     * @author fgcy
     * @since JDK 1.8
     */

    public static <T> T[] shuffleArray(T[] arr) {
        if (arr == null) return null;
        final Random r = new Random();
        for (int i = 0; i < arr.length - 1; i++) {
            final int anInt = r.nextInt(arr.length);
            T temp = arr[i];
            arr[i] = arr[anInt];
            arr[anInt] = temp;
        }
        return arr;
    }
}
```

```
[阿黄,阿雄,阿刘]
[阿雄,阿黄,阿刘]
[11,52,1,47,-47,0,3]
[0,1,47,52,-47,11,3]
```

## 自定义泛型接口

- **泛型接口**的概述

使用了泛型定义的接口就是泛型接口。

泛型接口的格式：修饰符 interface 接口名称<泛型变量>{}

作用：
　泛型接口可以让实现类选择当前功能需要操作的数据类型（约束实现类操作的数据类型）

泛型接口的原理：
　实现类可以在实现接口的时候传入自己操作的数据类型，这样重写的方法都将是针对于该类型的操作
　因为操作的类型在接口中定义了

```
public class GenericDemo2 {
    public static void main(String[] args) {
        final TeacherData teacherData = new TeacherData();
        System.out.println(teacherData.findAllData());
    }
}
```

```
interface Data<E> {
    boolean removeData(int id);

    boolean addData(E e);

    List<E> findAllData();
}
```

```
class TeacherData implements Data<Teacher> {
    @Override
    public boolean removeData(int id) {
        return false;
    }

    @Override
    public boolean addData(Teacher teacher) {
        return false;
    }

    @Override
    public List<Teacher> findAllData() {
        final List<Teacher> list = new ArrayList<>();
        Collections.addAll(list, new Teacher("aaa", 12), new Teacher("bbb", 13), new Teacher("ccc", 22));
        return list;
    }
}
```

```
class Teacher {
    private String name;
    private int age;

    @Override
    public String toString() {
        return "Teacher{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    public Teacher(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```
[Teacher{name='aaa', age=12}, Teacher{name='bbb', age=13}, Teacher{name='ccc', age=22}]
```

## 泛型通配符、上下限

通配符: ?
　? 可以在“使用泛型”的时候代表一切类型。 使用
　E T K V 是在定义泛型的时候使用的。 定义

泛型的上下限：
　? extends Car: ?必须是Car或者其子类　泛型上限
　? super Car ：　?必须是Car或者其父类　泛型下限

- 问题1：ArrayList<Car>类型与ArrayList<BMW>的类型没有关系

```
public class GenericDemo3 {


    public static void go(List<Car> cars) {

    }

    public static void main(String[] args) {
        final ArrayList<BENZ> list1 = new ArrayList<>();
        Collections.addAll(list1, new BENZ(), new BENZ(), new BENZ());
//        go(list1);//报错

        final ArrayList<BMW> list2 = new ArrayList<>();
        Collections.addAll(list2, new BMW(), new BMW(), new BMW());
```

```
//        go(list2);//报错
        //虽然BMW和BENZ都继承了Car但是ArrayList<BMW>和ArrayList<BENZ>与ArrayList<Car>没有关系的！！
        //Car类确实可以通过多态的方式接收子类的参数，但ArrayList<Car>是作为一个整体，一个类型，不存在之前的继承关系
    }
}
```

```
class Car {
}

class BENZ extends Car {
}

class BMW extends Car {
}
```

- 问题2：使用了？通配符 出现了新的问题

```
package generic_demo;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;


public class GenericDemo3 {

    //使用了 ？通配符，代表可以使用任何类型
    public static void go(List<?> cars) {

    }

    public static void main(String[] args) {
        final ArrayList<BENZ> list1 = new ArrayList<>();
        Collections.addAll(list1, new BENZ(), new BENZ(), new BENZ());
        go(list1);//不报错

        final ArrayList<BMW> list2 = new ArrayList<>();
        Collections.addAll(list2, new BMW(), new BMW(), new BMW());
        go(list2);//不报错

        final ArrayList<Dog> list3 = new ArrayList<>();
        Collections.addAll(list3, new Dog(), new Dog(), new Dog());
        go(list3);//不报错,有问题：因为狗类与汽车类没有关系，不符合逻辑
    }
}
```

```
class Car {
}


class BENZ extends Car {
}

class BMW extends Car {
}

class Dog {
}
```

- 使用泛型上限

```
package generic_demo;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;


public class GenericDemo3 {

    //使用了 ？通配符，代表可以使用任何类型;但是这里做了限制: ? extends 父类 表示----> ? 只能代表夫类型或是父类型的子类型
    public static void go(List<? extends Car> cars) {

    }

    public static void main(String[] args) {
        final ArrayList<BENZ> list1 = new ArrayList<>();
        Collections.addAll(list1, new BENZ(), new BENZ(), new BENZ());
        go(list1);//不报错

        final ArrayList<BMW> list2 = new ArrayList<>();
        Collections.addAll(list2, new BMW(), new BMW(), new BMW());
        go(list2);//不报错

        final ArrayList<Dog> list3 = new ArrayList<>();
        Collections.addAll(list3, new Dog(), new Dog(), new Dog());
//        go(list3);//报错,没有问题：因为狗类与汽车类没有关系，符合逻辑


        final ArrayList<Car> list4 = new ArrayList<>();
        //集合里里面的类型,子类和父类还是有关系的,可以使用多态接收参数
        Collections.addAll(list4, new Car(), new BMW(), new BMW());
        go(list4);//不报错
    }
}
```

```
class Car {

}


class BENZ extends Car {

}

class BMW extends Car {
```

```
}

class Dog {

}
```