集合框架 - Day01

今日学习内容：

- 常见的数据结构
- 集合框架体系
- List接口和实现类
- 泛型

今日学习目标：

- 了解什么是数据结构
- 了解常见的数据结构和其特点
- 掌握集合框架体系结构
- 掌握List接口存储特点
- 掌握ArrayList类常用方法
- 了解泛型的定义和作用
- 掌握使用集合时使用泛型来约束元素类型

1、数据结构概述 (了解)

Java的集合框架其实就是对数据结构的封装，在学习集合框架之前，有必要先了解下数据结构。

1.1 什么是数据结构（了解）

所谓数据结构，其实就是计算机存储、组织数据的方式。

数据结构是用来分析研究数据存储操作的，其实就是对数据做增删改查操作。

- 增：把某个数据存储到某个容器中
- 删：从容器中把某个数据删除掉
- 改：把容器中某个数据替换成另一个数据
- 查：把容器中的数据查询出来

> 开发意识
>
> 只要提到容器，那核心操作一定是增删改查。

不同的数据结构，底层采用不同的存储方式（算法），在具体操作的时候效率是不一样的，比如有的查询速度很快，有的插入速度很快，有的操作头和尾速度很快等。

1.2 为什么要熟悉数据结构（了解）

常见的数据结构：

- **数组（Array）　　掌握**
- 链表（Linked List）　了解
- 哈希表（Hash）　　了解
- 栈（Stack）　　了解
- 队列（Queue）　　了解
- 树（Tree）　　了解

- 图（Graph）
- 堆（Heap）

在未来具体业务场景时，我们需要分析具体的需求场景，是查询的操作多，还是添加的操作多？

如果是查询操作多，我们就要选择适合查询性能高的集合；

如果添加操作多，我们就需要选择添加性能高的集合。

一句话：根据具体业务场景，选择合适的集合类。

2、数据结构 (了解)

2.1 数组的性能分析（了解）

在计算机科学中，算法的时间复杂度是一个函数，它定性描述了该算法的运行时间，常用大O符号来表述。

时间复杂度是同一问题可用不同算法解决，而一个算法的质量优劣将影响到算法乃至程序的效率。算法分析的目的在于选择合适算法和改进算法。

我们在这里针对**数组**存储数据的增删改查（CRUD）做性能分析：

- 添加操作：

  如果保存在数组的最后一个位置，至少需要操作一次。

  如果保存在数组的第一个位置，如果存在N个元素，此时需要操作N次(后面的元素要整体后移)。

  平均：(N+1) /2 次。 N表示数组中元素的个数。 如果要扩容，更慢，性能更低。

- 删除操作:

  如果删除最后一个元素，操作一次。

  如果删除第一个元素，操作N次。

  平均：(N+1)/2次.

- 修改操作：给定索引时，操作1次.
- 查询操作：根据索引查询元素需要操作1次；根据内容查询需要N次；

结论：基于数组的数据结构做查询是和修改是非常快的，添加和删除操作比较慢了。

那如果想保证保存和删除操作的性能，此时就得提提链表这种数据结构了。

2.2 其他线性数据结构（了解）

1.2.1. 链表（了解）

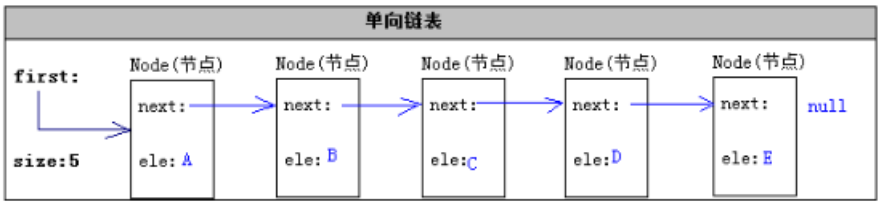
链表结构（火车和火车车厢）：

1)：单向链表，只能从头遍历到尾/只能从尾遍历到头。

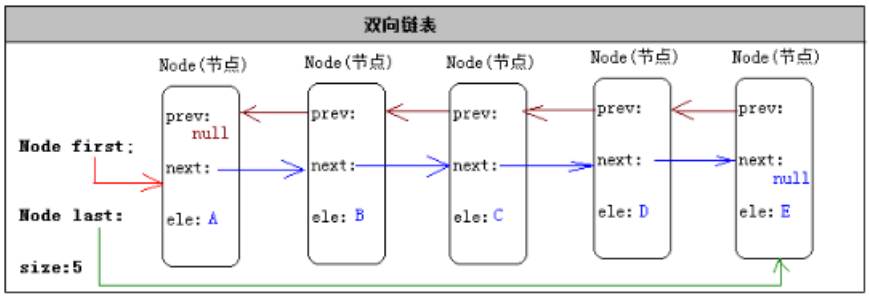
2)：双向链表，既可以从头遍历到尾,又可以从尾遍历到头。

通过引用来表示上一个节点和下一个节点的关系。

单向链表：



双向链表：



对链表操作的性能分析：

双向链表可以直接获取自己的第一个和最后一个节点。

- 添加操作

  如果新增的元素在第一个或最后一个位置，那么操作只有1次。

- 删除操作

  如果删除第一个元素 ： 操作一次

  如果删除最后一个元素：操作一次

  如果删除中间的元素：

  找到元素节点平均操作：(1+N)/2次

  找到节点之后做删除操作: 1次

- 修改操作

  平均：(N+1)/2次

- 查询操作:

  平均:(N+1)/2次

结论：

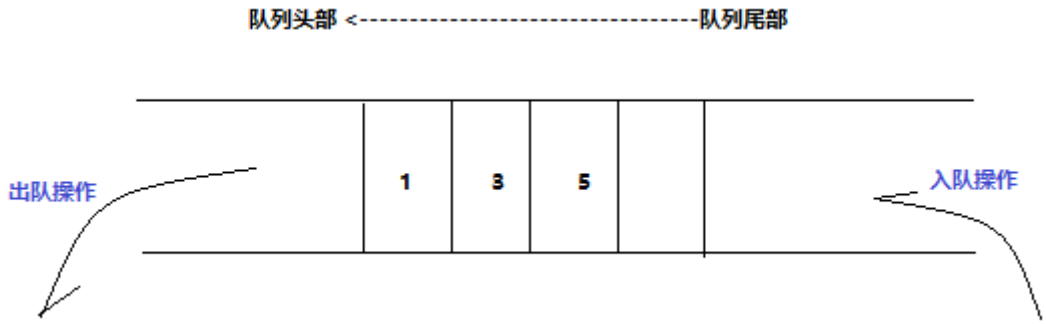
基于链表的数据结构做查询是和修改是比较慢的（因为每次从头开始查找），添加和删除操作比较块了（不需要需用任何元素）。

1.2.2. 队列（了解）

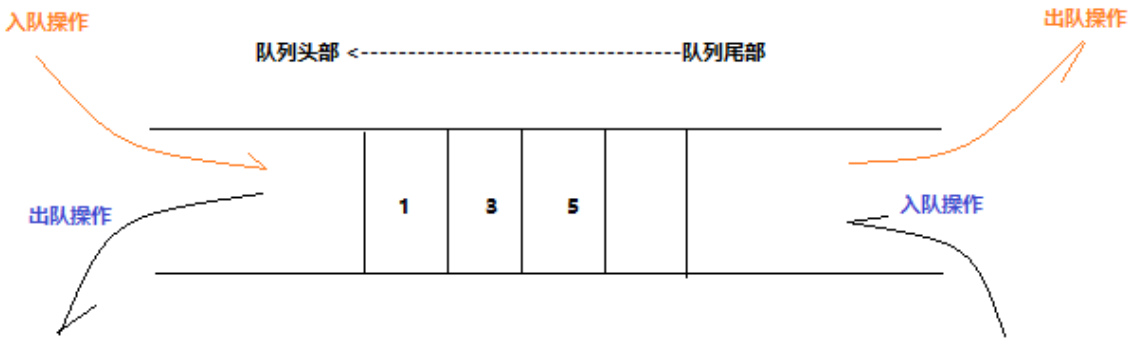
队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，队列是一种操作受限制的线性表。

进行插入操作的端称为队尾，进行删除操作的端称为队头。

单向队列(Queue)：先进先出(FIFO)，只能从队列尾插入数据，只能从队列头删除数据。



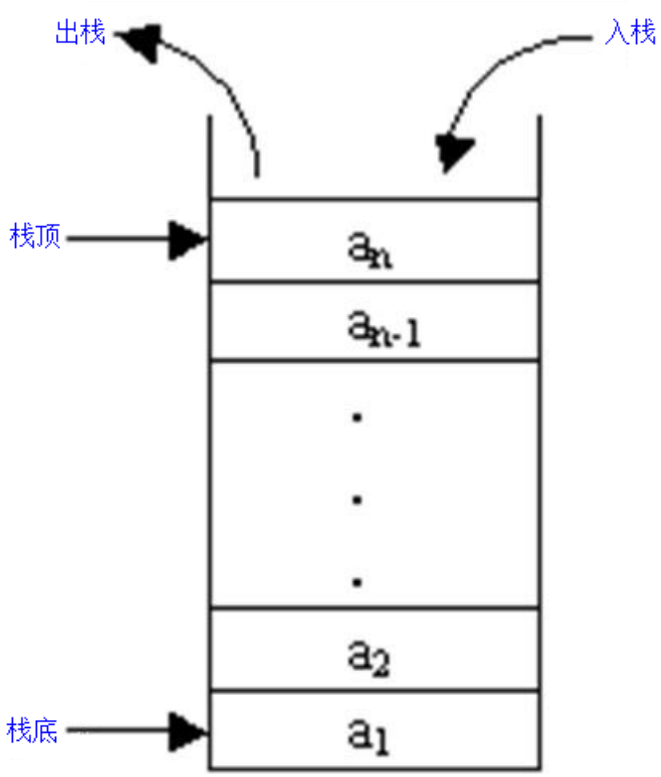
双向队列(Deque)：可以从队列尾/头插入数据，只能从队列头/尾删除数据。



结论：最擅长操作头和尾，元素先进先出(FIFO)

1.2.3. 栈（了解）

栈（stack）又名堆栈，它是一种运算受限的线性表，后进先出(LIFO)，和手枪弹夹类似。



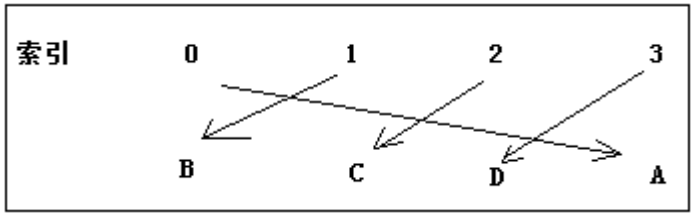
栈结构仅允许在表的一端进行插入和删除运算，这一端被称为栈顶，相对地，把另一端称为栈底。向一个栈中插入新元素又称作入栈，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素。从一个栈中删除元素又称作出栈，表示把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素（LIFO）。

结论：最擅长在线性表的一端添加和删除操作，元素后进先出(LIFO)

2.3 非线性数据结构(后续讲)

1.3.1. 哈希表（了解）

一般的，数组中元素在数组中的索引位置是随机的，元素的取值和元素的位置之间不存在确定的对应关系。因此，在数组中查找特定的值时，需要把查找值和一系列的元素进行比较。



此时的查询效率依赖于查找过程中所进行的比较次数，如果比较次数较多，查询效率还是不高。

如果元素的值（value）和在数组中的索引位置（index）有一个确定的对应关系，我们把这种关系称之为哈希（hash）。则元素值和索引对应的公式为：index = hash(value)。也就是说，通过给定元素值，只要调用hash(value)方法，就能找到数组中取值为value的元素的位置。

元　素　值	11	22	33	44	55	66	77	88	99
元素的位置	0	1	2	3	4	5	6	7	8

比如，图中的hash算法公式为：**index = value / 10 - 1**。

在往哈希表中存储对象时，该hash算法就是对象的hashCode方法。

注：这里仅仅是假设算法公式是这样的，真实的算法公式我们可以不关心。

1.3.4. 树和二叉树（了解）

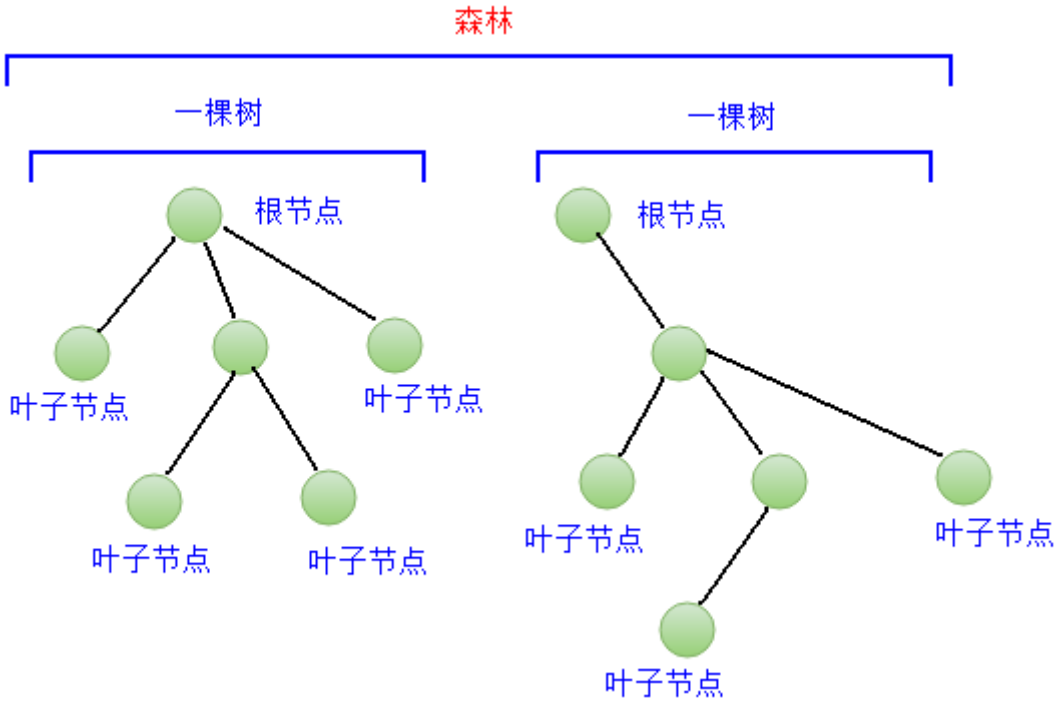
前面我们介绍的数据结构数组、栈、队列，链表都是线性数据结构，除此之外还有一种比较复杂的数据结构——树。

计算机中的树，是根据生活中的树抽象而来的，表示N个有父子关系的节点的集合。

- N为0的时候，该节点集合为空，这棵树就是空树
- 任何非空树中，有且只有一个根节点（root）
- N>1时，一颗树由根和若干棵子树组成，每棵子树由更小的若干子树组成

树中的节点根据有没有子节点，分成两种：

- 普通节点：拥有子节点的节点。
- 叶子节点：没有字节点的节点。



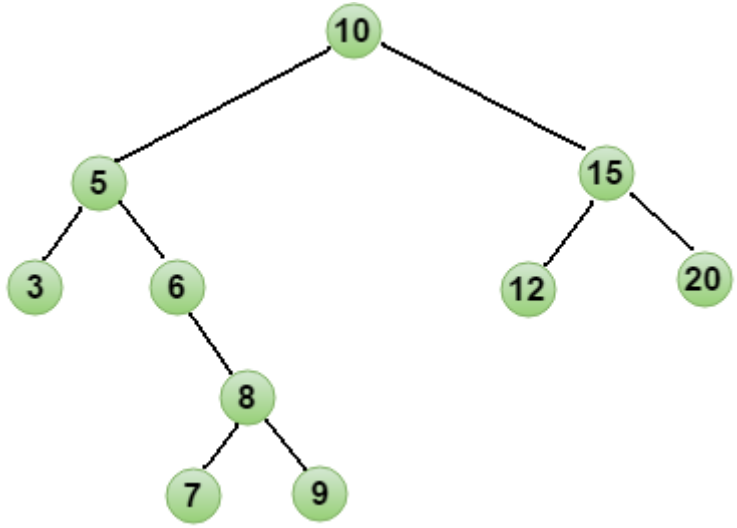
二叉树：一种特殊的，遵循某种规则的树。

树的结构因为存在多种子节点情况，真的太复杂了，如果我们对普通的树加上一些约束，比如让每一棵树的节点最多只能包含两个子节点，而且严格区分左子节点和右子节点（左右位置不能交换），此时就形成了二叉树。

排序二叉树，有顺序的树：

- 若左子树不为空，则左子树所有节点的值小于根节点的值。

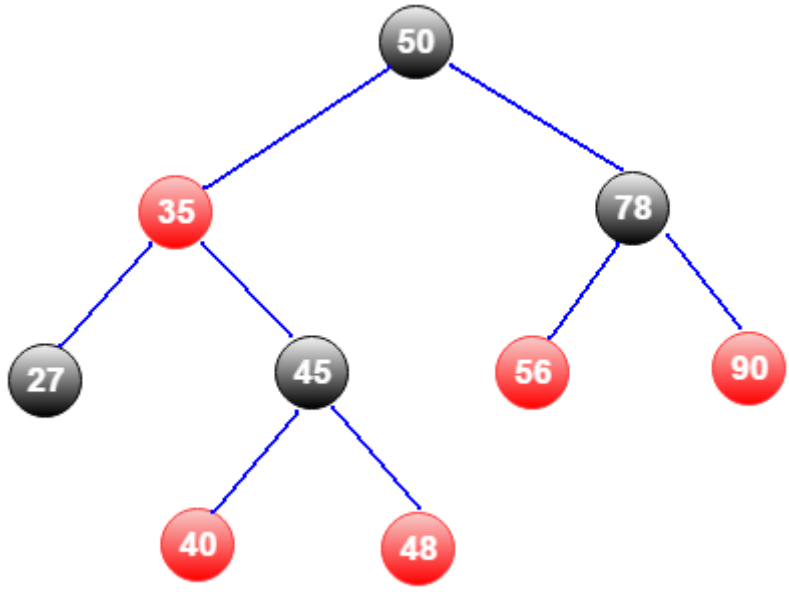
- 若右子树不为空，则右子树所有节点的值大于根节点的值。
- 左右子树也分别是排序二叉树。



红黑树：更高查询效率的的排序二叉树。

排序二叉树可以快速查找，但是如果只有左节点或者左右右节点的时候，此时二叉树就变成了普通的链表结构，查询效率比较低。为此一种更高效的二叉树出现了——红黑树。

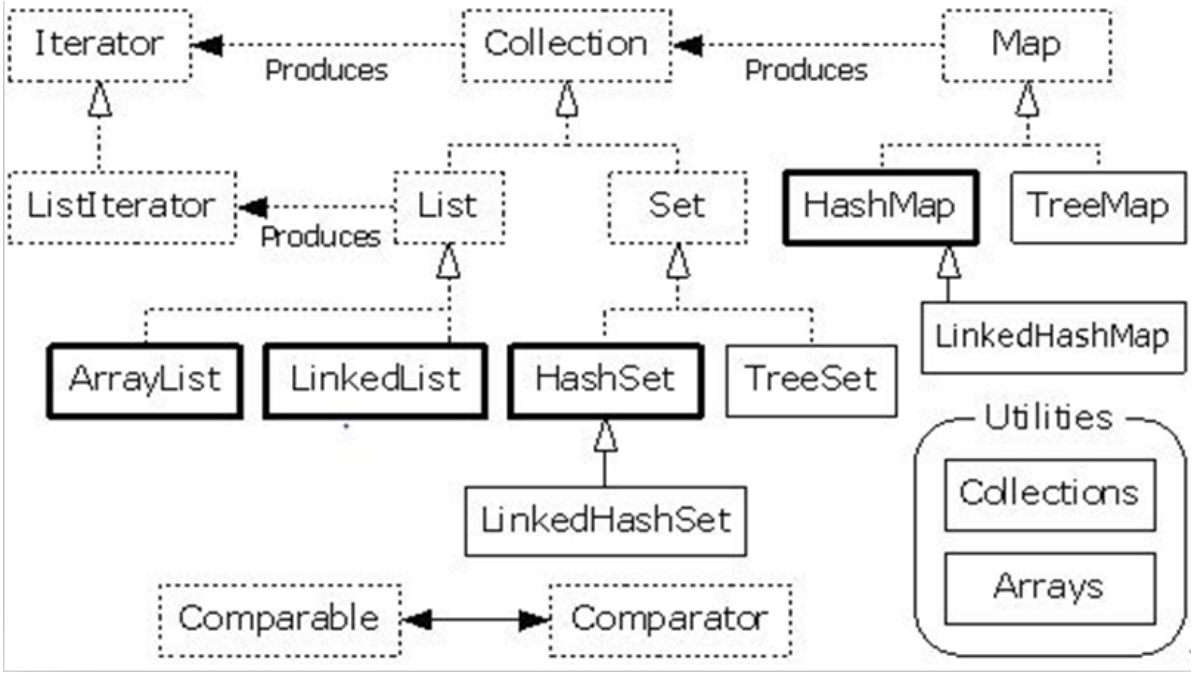
- 每个节点要么是红色的，要么是黑色的。
- 根节点永远是黑色的。
- 所有叶子节点都是空节点（null），是黑色的。
- 每个红色节点的两个子节点都是黑色的。
- 从任何一个节点到其子树每个叶子节点的路径都包含相同数量的黑色节点。



3、集合框架体系（掌握）

3.1 集合框架概述（了解）

集合是Java中提供的一种容器，可以用来存储多个数据，根据不同存储方式形成的体系结构，就叫做集合框架体系。集合也时常被称为容器。



每一种容器类底层拥有不同的底层算法(数据结构)。

既然数组可以存储多个数据，为什么要出现集合？-

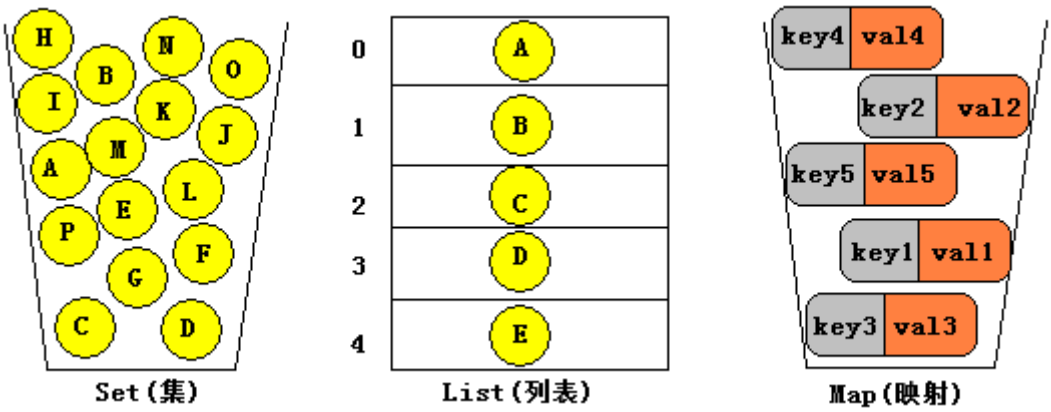
- 数组的长度是固定的，集合的长度是可变的（自动拓展容量）。
- 使用Java类封装出一个个容器类，开发者只需要直接调用即可，不用程序员再手写容器类。

集合中存储的数据，叫做元素，元素只能是对象（引用类型）。

在Java中，集合操作都被设计成接口，被不同底层数据结构的实现类所实现。

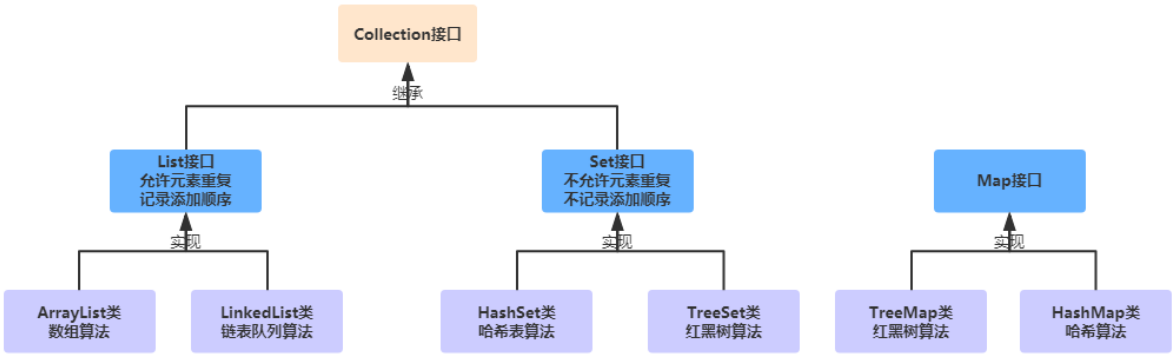
3.2 集合的分类（掌握）

根据容器的存储特点的不同，可以分成三种情况：



- List(列表)：允许记录添加顺序，允许元素重复。
- Set(数据集)：不记录添加顺序，不允许元素重复。
- Map(映射)：容器中每一个元素都包含一对key和value，key不允许重复，value可以重复。严格上说，并不是容器（集合），是两个容器中元素映射关系。

注意：List和Set接口继承于Collection接口，Map接口不继承Collection接口。



- Collection接口：泛指广义上集合，主要表示List和Set两种存储方式。
- **List接口：表示列表，规定了允许记录添加顺序，允许元素重复的规范。**
- **Set接口：表示数据集，规定了不记录添加顺序，不允许元素重复的规范。**
- Map接口：表示映射关系，规定了两个集合映射关系的规范。

注意：我们使用的容器接口或类都处于java.util包中。

4、List接口（重点）

List接口是Collection接口子接口，List接口定义了一种规范，要求该容器允许**记录元素的添加顺序，也允许元素重复。**那么List接口的实现类都会遵循这一种规范。

List集合存储特点：

- 允许记录元素的添加先后顺序
- 允许元素重复

该接口常用的实现类有：

- **ArrayList类：数组列表，表示数组结构，底层采用数组实现，开发中使用最多的实现类，重点。**
- LinkedList类：链表，表示双向列表和双向队列结构，采用链表实现，使用不多。
- Stack类：栈，表示栈结构，采用数组实现，使用不多。
- Vector类：向量，其实就是古老的ArrayList，采用数组实现，使用不多。

一般来说，集合接口的实现类命名规则：（底层数据结构 + 接口名）例如：ArrayList

4.1 List常用API方法（必须记住）

添加操作

- boolean add(Object e)：将元素添加到列表的末尾
- void add(int index, Object element)：在列表的指定位置插入指定的元素
- boolean addAll(Collection c)：把c列表中的所有元素添加到当前列表中

删除操作

- Object remove(int index)：从列表中删除指定索引位置的元素,并返回被删除的元素
- boolean removeAll(Collection c)：从此列表中移除c列表中的所有元素

修改操作

- Object set(int index, Object ele)：修改列表中指定索引位置的元素，返回被替换的旧元素

查询操作

- int size()：返回当前列表中元素个数

- boolean isEmpty()：判断当前列表中元素个数是否为0
- Object get(int index)：查询列表中指定索引位置对应的元素
- Object[] toArray()：把列表对象转换为Object数组
- boolean contains(Object o):判断列表是否存在指定对象

注意，标红的是经常使用的方法。

4.2 ArrayList类（重点）

ArrayList类，基于数组算法的列表，通过查看源代码会发现底层其实就是一个Object数组。

需求1：操作List接口常用方法

```
public class ArrayListDemo1 {
    public static void main(String[] args) {
        // 创建一个默认长度的列表对象
        List list = new ArrayList();
        // 打印集合中元素的个数
        System.out.println("元素数量："+list.size()); // 0

        // 添加操作：向列表中添加4个元素
        list.add("Will");
        list.add("alex");
        list.add("ben");
        list.add("Lucy");

        // 查询操作：
        System.out.println("列表中所有元素：" + list);//输出:[Will, alex, ben, Lucy]
        System.out.println("元素数量："+ list.size());//4
        System.out.println("第一个元素：" + list.get(0));//Will

        // 修改操作：把索引为2的元素，替换为wolfcode
        list.set(2, "wolfcode");
        System.out.println("修改后："+ list);//输出:[Will, alex, wolfcode, Lucy]

        // 删除操作：删除索引为1的元素
        list.remove(1);
        System.out.println("删除后："+ list);//输出:[Will, wolfcode, Lucy]
    }
}
```

需求2：创建四个User对象，存储在ArrayList中，分析内存图。

```
public class User {
    private String name;
    private int age;
     //省略两个参数构造器、getter/setter方法、toString方法
}

public class ArrayListDemo2 {
    public static void main(String[] args) {
        List girls = new ArrayList();

        User u1 = new User("西施", 18);
        girls.add(u1);
```

```
        girls.add(new User("王昭君",19));
        girls.add(new User("貂蝉",20));
        girls.add(new User("杨玉环",21));
        System.out.println(girls);

        //修改u1对象的名字和年龄
        u1.setName("小施");
        u1.setAge(17);
        System.out.println(girls);
    }
}
```

运行结果（观察变化）：

```
[User [name=西施, age=18], User [name=王昭君, age=19], User [name=貂蝉, age=20],
User [name=杨玉环, age=21]]
[User [name=小施, age=17], User [name=王昭君, age=19], User [name=貂蝉, age=20],
User [name=杨玉环, age=21]]
```

内存分析，解释原因：

结论：集合类中存储的对象,都存储的是对象的引用,而不是对象本身。

4.3 LinkedList类（了解）

ArrayList类，基于数组算法的列表，通过查看源代码会发现底层其实就是一个Object数组。

LinkedList类，底层采用链表算法，实现了链表，队列，栈的数据结构。无论是链表还是队列主要操作的都是头和尾的元素，因此在LinkedList类中除了List接口的方法，还有很多操作头尾的方法。

- void addFirst(Object e) 将指定元素插入此列表的开头 （头部入队）。
- void addLast(Object e) 将指定元素添加到此列表的结尾 （尾部入队）。
- Object getFirst() 返回此列表的第一个元素 （获取队列头部元素）。
- Object getLast() 返回此列表的最后一个元素 （获取队列尾部元素）。
- Object removeFirst() 移除并返回此列表的第一个元素 （头部出队）。
- Object removeLast() 移除并返回此列表的最后一个元素 （尾部出队）。

注意： 如果队列为空（无元素），出队操作会出异常。

- boolean offerFirst(Object e) 在此列表的开头插入指定的元素。
- boolean offerLast(Object e) 在此列表末尾插入指定的元素。
- Object peekFirst() 获取但不移除此列表的第一个元素；如果此列表为空，则返回 null。
- Object peekLast() 获取但不移除此列表的最后一个元素；如果此列表为空，则返回 null。
- Object pollFirst() 获取并移除此列表的第一个元素；如果此列表为空，则返回 null。
- Object pollLast() 获取并移除此列表的最后一个元素；如果此列表为空，则返回 null。

- void push(Object e) 将元素推入此列表所表示的栈 （入栈）。
- Object pop() 从此列表所表示的栈处弹出一个元素 （出栈）。
- Object peek() 获取但不移除此列表的头部元素（获取栈顶元素）。

LinkedList之所以有这么多方法，是因为自身实现了多种数据结构，而不同的数据结构的操作方法名称不同，在开发中LinkedList使用不是很多，知道存储特点就可以了。

```
public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList queue = new LinkedList();
        // 入队
        queue.addLast("A");
        queue.addLast("B");
        queue.addFirst("C");
        System.out.println(queue.toString());  // [C,A,B]

        // 获取队列的头部和尾部元素
        System.out.println("获取队列的头部元素:" + queue.getFirst());  // C
        System.out.println("获取队列的尾部元素:" + queue.getLast());   // B

        // 出队
        Object item;
        // 如果队列为空，出队操作会出现异常
        item = queue.removeFirst();
        System.out.println("出队元素:" + item);    // C
        System.out.println(queue.toString());   // [A,B]
    }
}
```

程序运行结果：

```
[C, A, B]
获取队列的头部元素:C
获取队列的尾部元素:B
出队元素:C
[A, B]
```

4.4 Stack和Vector类（了解）

Vector类：基于数组算法实现的列表，其实就是ArrayList类的前身。和ArrayList的区别在于方法使用**synchronized**修饰，所以相对于ArrayList来说，线程安全，但是效率就低了点。

Stack类：表示栈，是Vector类的子类，具有后进先出(LIFO)的特点，拥有push（入栈），pop（出栈）方法。

5、泛型（会用即可）

5.1 什么是泛型（了解）

其实就是一种**类型参数，主要用于某个类或接口中数据类型不确定时，可以使用一个标识符来表示未知的数据类型**，然后在使用该类或接口时指定该未知类型的真实类型。

泛型可用到的接口、类、方法中，将数据类型作为参数传递，其实更像是一个数据类型模板。

如果不使用泛型，从容器中获取出元素，需要做类型强转，也不能限制容器只能存储相同类型的元素。

```
List list = new ArrayList();
list.add("A");
list.add("B");
String ele = (String) list.get(0);
```

5.2 自定义和使用泛型（了解）

定义泛型：使用一个标识符，比如T在类中表示一种未知的数据类型。

使用泛型：一般在创建对象时，给未知的类型设置一个具体的类型，当没有指定泛型时，默认类型为Object类型。

需求：定义一个类Point，x和y表示横纵坐标，分别使用String、Integer、Double表示坐标类型。

如果没有泛型需要设计三个类，如下：

```
class Point{
  private String  x;
  private String  y;

  //TODO
}
```

```
class Point{
  private Integer  x;
  private Integer  y;

  //TODO
}
```

```
class Point{
  private Double  x;
  private Double  y;

  //TODO
}
```

定义泛型：

//在类上声明使用符号T,表示未知的类型

```
//在类上声明使用符号T,表示未知的类型
public class Point<T> {
    private T x;
    private T y;
     //省略getter/setter
}
```

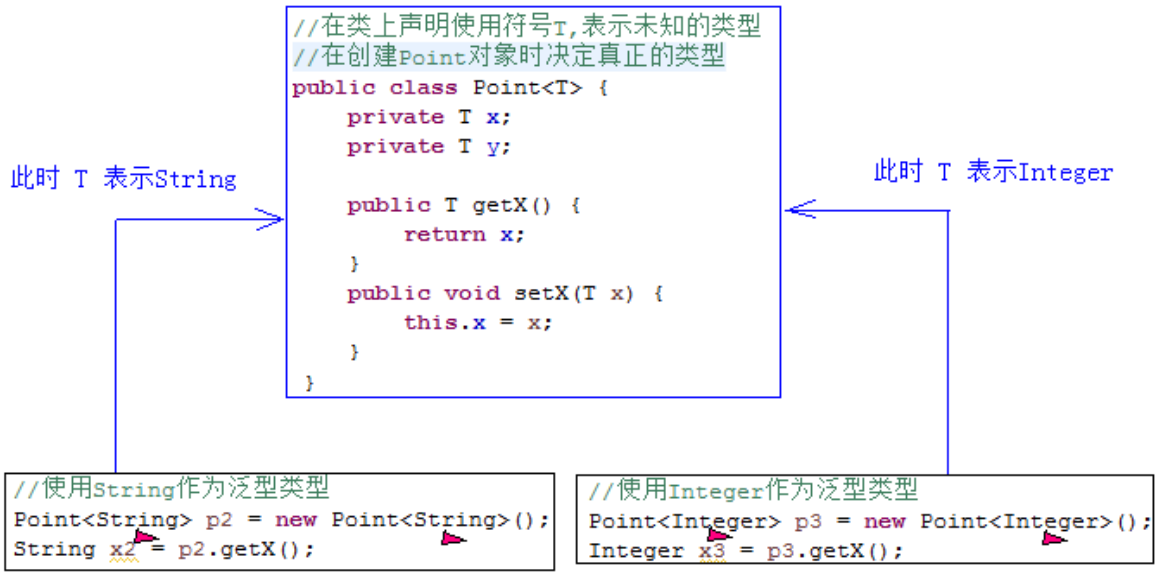
使用泛型：

```
//没有使用泛型，默认类型是Object
Point p1 = new Point();
Object x1 = p1.getX();

//使用String作为泛型类型
Point<String> p2 = new Point<String>();
String x2 = p2.getX();

//使用Integer作为泛型类型
Point<Integer> p3 = new Point<Integer>();
Integer x3 = p3.getX();
```

画图分析：



注意：这里仅仅是演示泛型类是怎么回事，并不是要求定义类都要使用泛型。

5.3 在集合框架中使用泛型（掌握）

拿List接口和ArrayList类举例。

```
// Element => E
class ArrayList<E>{
    public boolean add(E e){ }
    public E get(int index){  }
}
```

此时的E也仅仅是一个占位符，表示元素（Element）的类型，那么当使用容器时给出泛型就表示该容器只能存储某种类型的数据。

```
//只能存储String类型的集合
List<String> list1 = new ArrayList<String>();
list1.add("A");
list1.add("B");

//只能存储Integer类型的集合
List<Integer> list2 = new ArrayList<Integer>();
list2.add(11);
list2.add(22);
```

因为前后两个泛型类型相同（也必须相同），泛型类型推断：

```
List<String> list1 = new ArrayList<String>();
可以简写为
List<String> list1 = new ArrayList<>();
```

通过反编译工具，会发现泛型其实是语法糖，也就是说编译之后，泛型就不存在了。

注意：泛型必须是引用类型，不能是基本数据类型（错误如下）：

```
List<int> list = new ArrayList<int>();//编译错误
```

泛型不存在继承的关系（错误如下）：

```
List<Object> list = new ArrayList<String>();    //错误的
```

6、集合元素迭代（掌握）

1.1 集合元素遍历（掌握）

集合的遍历：把集合中的每一个元素获取出来

```
List<String> list = new ArrayList<>();
list.add("西施");
list.add("王昭君");
list.add("貂蝉");
list.add("杨玉环");
```

使用for遍历

```
for (int index = 0; index < list.size(); index++) {
    String ele = list.get(index);
    System.out.println(ele);
}
```

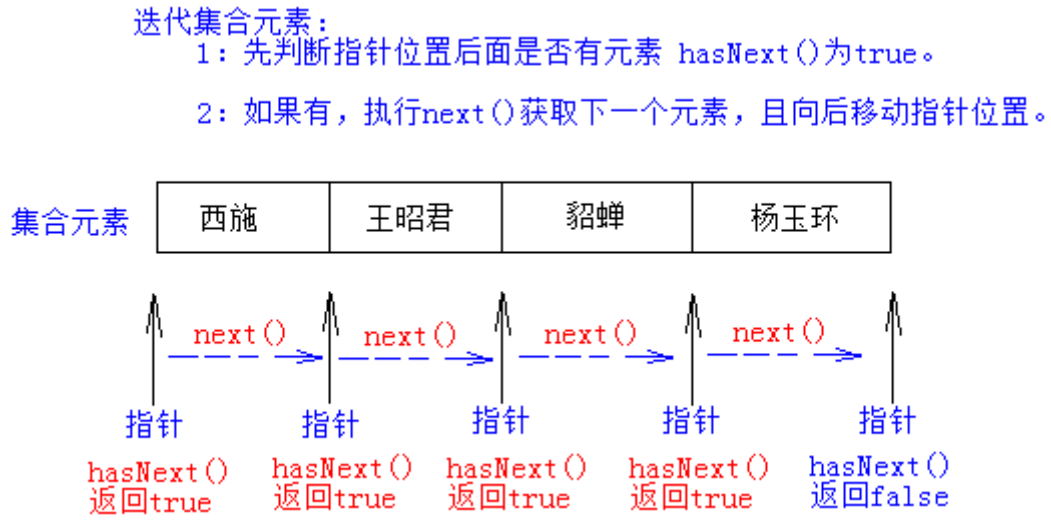
使用迭代器遍历

Iterator表示迭代器对象，迭代器中拥有一个指针，默认指向第一个元素之前，

- boolean hasNext()：判断指针后是否存在下一个元素
- Object next()：获取指针位置下一个元素，获取后指针向后移动一位

```
Iterator<String> it = list.iterator();
while(it.hasNext()) {
    String ele = it.next();
    System.out.println(ele);
}
```

操作原理如下图：



使用for-each遍历（推荐使用）

语法：

```
for(数据类型 迭代变量: 数组/Iterable实例对象){
    //TODO
}
```

这里Iterable实例对象表示Iterable接口实现类对象，其实就是Collection，不包括Map。

```
for (String ele : list) {
        System.out.println(ele);
}
```

通过反编译工具会发现，for-each操作集合时，其实底层依然是Iterator，我们直接使用for-each即可。

1.2 并发修改异常（了解）

需求：在迭代集合时删除集合元素，比如删除王昭君。

```
List<String> list = new ArrayList<>();
list.add("西施");
list.add("王昭君");
list.add("貂蝉");
list.add("杨玉环");
System.out.println(list);

// 在迭代过程中进行删除操作
for (String ele : list) {
    if("王昭君".equals(ele)) {
        list.remove(ele);
    }
}
System.out.println(list);
```

此时报错java.util.ConcurrentModificationException，并发修改异常。

造成该错误的原因是，**不允许在迭代过程中改变集合的长度（不能删除和增加）**。如果要在迭代过程中删除元素，就不能使用集合的remove方法，只能使用迭代器的remove方法，此时只能使用迭代器来操作，不能使用foreach。

```
List<String> list = new ArrayList<>();
list.add("西施");
list.add("王昭君");
list.add("貂蝉");
list.add("杨玉环");
System.out.println(list);

//获取迭代器对象
Iterator<String> it = list.iterator();
while(it.hasNext()) {
    String ele = it.next();
    if("王昭君".equals(ele)) {
        it.remove();
    }
}
System.out.println(list);
```