# 集合

Java集合框架是Java编程语言中一个强大的工具箱，它提供了多种数据结构来高效地管理和操作对象集合。集合框架位于java.util​包中，并且它的设计遵循了统一的接口规范，使得开发者可以根据具体需求选择合适的实现类

* **线程安全**：Java集合框架中的大部分实现类默认不是线程安全的。若要在多线程环境中安全使用，可以考虑使用Collections.synchronizedXxx()​方法包装，或者使用java.util.concurrent​包下的并发集合，如ConcurrentHashMap​、CopyOnWriteArrayList​等。

* **迭代器**：所有集合类都可以通过iterator()​方法获取迭代器进行遍历，迭代过程中支持remove()​方法移除当前迭代项。

* **泛型**：Java集合框架全面支持泛型，可以在编译时确保集合内元素的类型一致，提高程序的类型安全性。

* **流(Streams)** ：Java 8引入了Stream API，它可以与集合无缝配合，提供更高级的函数式编程风格的数据处理能力。

集合对比数组的优点：

集合是存储对象数据的一种容器。集合非常适合做元素的增删操作。

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。**

1. 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。**

1. 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。**

‍

## 接口层次

​​

#### - java.util.Collection​

这是所有集合类的根接口，它代表一组对象。Collection有两个直接子接口：

* ​java.util.List​: 有序（可重复）、元素可以通过索引访问的集合，如ArrayList​、LinkedList​等。

* ​java.util.Set​: 不允许重复元素（每个元素都是唯一的），无序集合，如HashSet​、LinkedHashSet​和TreeSet​。

#### - java.util.Map​

不同于集合，Map是一个键值对的集合，它维护的是键到值的映射关系，不继承自Collection，但与之并列。常见的实现类有HashMap​、TreeMap​和LinkedHashMap​等。

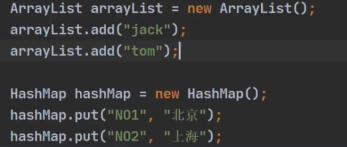
‍

### 集合类体系结构

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）。

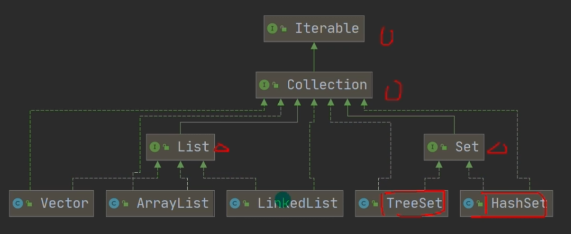
​​

单列与双良

​​

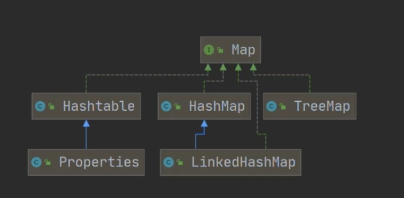
Collection体系

都是基于Iterable接口实现的，所以collection接口下都可以使用**迭代器**

​​

‍

map体系

​​

‍

快捷键。Ctrl+J (很多常用代码的快捷码如：sout) ，输入itit

‍

## 迭代器

在Java中，迭代器（Iterator）是一种设计模式，用于遍历集合对象（如数组、列表、集合等）中的元素，而不需要暴露集合的内部表示。Java的java.util​包中提供了一个Iterator​接口，定义了遍历集合的方法。

### 迭代器的基本方法：

1. ​hasNext()​: 检查序列中是否还有元素。

1. ​next()​: 返回序列中的下一个元素。

1. ​remove()​: 移除迭代器刚返回的元素。

### 使用迭代器

import java.util.ArrayList;  
import java.util.Iterator;  
public class Main {  
 public static void main(String[] args) {  
 List list = new ArrayList<>();  
 list.add(1);  
 list.add(2);  
 list.add(3);  
  
 Iterator iterator=list.iterator();  
 while (iterator.hasNext()) {  
// 编译类型设置为Object类可以接收所有的运行类型的子类。  
 Object next = iterator.next();  
 System.out.println("obj="+next);  
 }  
  
 }  
}

### 迭代器的类型

1. **泛型迭代器**：如Iterator<String>​，可以提供类型安全。

1. **列表迭代器**（ListIterator​）：提供了更多的功能，如向前遍历、设置元素等。

1. **枚举迭代器**（Enumeration​）：用于遍历旧版本的集合类，如Vector​。

### 注意事项

* 使用迭代器遍历时，不应该尝试修改集合结构，这可能会导致ConcurrentModificationException​异常。

* 迭代器允许在遍历过程中移除元素，但通常不建议这么做。  
  通过迭代器，Java提供了灵活且统一的方式来遍历不同的集合类型，提高了代码的可读性和可维护性。

‍

### 增强for

底层仍然是迭代器，但是写法更简化。不仅可以遍历集合，还可以遍历数组。

debug进源码

public Iterator<E> iterator() {  
 return new Itr();  
 }

for (Object o : list) {  
 System.out.println(o);  
 }

‍

‍

## Collection集合

* 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

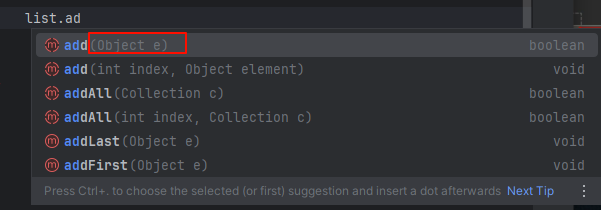
* Set系列集合：添加的元素是无序、不重复、无索引。
* ◆HashSet:无序、不重复、无索引；LinkedHashSet::有序、不重复、无索引。  
  ◆TreeSet:按照大小默认升序排序、不重复、无索引。

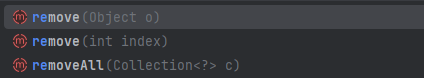
Collection接口

**以ArrayList为例，Set实现类也有。**

实例化对象并使用常见的方法

List list=new ArrayList();

* 添加元素
* //添加单个  
   list.add("beidao");  
   list.add('d');  
   list.add(213);  
   list.add(true);  
    
   //添加多个 可以传入一个实现collection接口的集合  
   List list2=new ArrayList();  
    
   list2.add("dasda");  
   list2.add("大闹三国");  
    
   list.addAll(list2);  
    
   System.out.println(list);
* 添加的内容 Object就是泛型，可以添加任意类型的数据
* 添加多个时，只要是实现了collectinn接口的集合都可以直接添加一整个集合进去。
* ​​

* 删除元素（如果有多个就会先删除且只删除索引值小的）
* // 删除单个元素  
   list.remove(true);  
   list.remove(1);  
    
   System.out.println(list);  
    
   List list2=new ArrayList();  
   list2.add("dasda");  
   list2.add("大闹三国");  
   // 删除多个元素 可以传入一个实现collection接口的集合  
   list.removeAll(list2);
* 一种是通过元素值去移除，一种是通过索引值。
* ​​

* contains 查找元素是否存在。
* 可以传入一个实现collection接口的集合
* // 查找是否包含元素  
   System.out.println(list.contains('d'));  
   System.out.println(list.contains(213));  
    
    
   List list2=new ArrayList();  
   list2.add("dasda");  
   list2.add("大闹三国");  
  // containsAll 查找多个元素是否都存在,   
   System.out.println(list.containsAll(list2));

* ​获取元素个数​
* System.out.println(list.size());

* ​判断元素是否为空​
* System.out.println(list.isEmpty());

* ​清空​
* list.clear();

‍

### List接口

1. List集合元素是有序的，存入和取出的顺序一直，且可重复。**（就是队列）**

1. 支持索引

‍

#### ArrayList

ArrayList是由数组来实现数据存储的

ArrayList基本等同于Vector,除了ArrayList是线程不安全（执行效率高）看源码.，在多线程情况下，不建议使用ArrayList

LIst单独的接口方法：

**功能描述：**

1. **add方法：** 在指定位置插入元素。此例中，我们在列表的第二个位置插入元素"第一个元素"。

1. **indexOf方法：** 查找列表中元素的首次出现的位置，并返回它的索引。此例中，我们查找字符'd'的位置。

1. **get方法：** 获取列表中指定位置的元素。此例中，我们获取位置为2的元素。

1. **lastIndexOf方法：** 查找列表中元素的最后一次出现的位置，并返回它的索引。此例中，我们查找整数213的位置。

1. **remove方法：** 移除列表中指定位置的元素，并返回该元素。此例中，我们移除位置为1的元素。

1. **set方法：** 替换列表中指定位置的元素值。此例中，我们将位置为2的元素值替换为"重新设置"。

1. **subList方法：** 获取列表中指定区间的一个子列表。此例中，我们获取从开始到位置3（不包括3）的子列表。  
   **代码演示：**

import java.util.ArrayList;  
import java.util.List;  
public class ListFeatures {  
 public static void main(String[] args) {  
 List<String> list = new ArrayList<>();  
 list.add("元素1");  
 list.add("元素2");  
 list.add("元素3");  
 list.add("元素4");  
 // 在指定位置插入元素  
 list.add(2, "第一个元素");  
 System.out.println(list); // 输出： [元素1, 元素2, 第一个元素， 元素3, 元素4]  
 // 查找元素位置  
 System.out.println(list.indexOf('d')); // 输出： 输出可能为-1，因为没有字符'd'  
 // 获取元素  
 System.out.println(list.get(2)); // 输出： 第一个元素  
 // 获取位置是最后一个所查找的元素  
 System.out.println(list.lastIndexOf(213)); // 输出： -1，因为没有数字213  
 // 删除固定位置id元素，并返回  
 System.out.println(list.remove(1)); // 输出： 元素2，列表变为： [元素1, 第一个元素， 元素3, 元素4]  
 // 替换某个位置的元素值，需要存在  
 list.set(2, "重新设置");  
 System.out.println(list); // 输出： [元素1, 第一个元素， 重新设置， 元素4]  
 // 截取一个区间的内容，左开右闭区间  
 System.out.println(list.subList(0, 3)); // 输出： [元素1, 第一个元素， 重新设置]  
 }  
}

‍

源码解析：

ArrayListl的底层操作机制源码分析（重点，难点.）

1)ArrayListi中维护了一个Object类型的数组elementData.[debug看源码]transient Object[]elementData;

2)当创建对像时，如果使用的是无参构造器，则初始elementData容量为0(Jdk7是10)

3)当添加元素时：先判断是否需要容，如果需要扩容，则调用grow方法，否则直接添加素到合适位置

4)如果使用的是无参构造器，如果第一次添加，需要扩容的话，则扩容elementData为10如果需要再次扩容的话，则扩容elementData为1.5倍。

5)如果使用的是指定容量capacityl的构造器，则初始elementData容量为capacity

6)如果使用的是指定容量capacity的构造器，如果需要扩容，则直接扩容elementData为1.5倍。

‍

‍

小案例：

为了实现一个按价格从小到大排序的 Book​ 类，我们首先需要创建一个 Book​ 类，它应该包含价格作为其中一个属性。然后，我们将使用 ArrayList​ 来存储 Book​ 对象的实例，并使用 Collections.sort()​ 方法对 ArrayList​ 进行排序。

package Collection.collection.List;  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
// Book 类定义  
class Book {  
 private String name;  
 private double price;  
 // Book 类的构造函数  
 public Book(String name, double price) {  
 this.name = name;  
 this.price = price;  
 }  
 // 获取书籍名称  
 public String getName() {  
 return name;  
 }  
 // 设置书籍名称  
 public void setName(String name) {  
 this.name = name;  
 }  
 // 获取书籍价格  
 public double getPrice() {  
 return price;  
 }  
 // 设置书籍价格  
 public void setPrice(double price) {  
 this.price = price;  
 }  
 // toString 方法，用于打印书籍信息  
 @Override  
 public String toString() {  
 return "书名:"+name+",价格:"+price;  
 }  
}  
// 主类  
public class ArrayList02 {  
 public static void main(String[] args) {  
 // 创建一个ArrayList来存储Book对象  
 ArrayList<Book> books = new ArrayList<>();  
 // 添加一些Book对象到ArrayList中  
 books.add(new Book("Book1", 9.99));  
 books.add(new Book("Book2", 223.99));  
 books.add(new Book("Book3", 29.99));  
 books.add(new Book("Book4", 91.99));  
 books.add(new Book("Book5", 123.99));  
 books.add(new Book("Book6", 229.99));  
 // 打印原始的书籍列表  
 System.out.println("Original Books: " + books);  
 // 使用Collections.sort()方法对书籍列表按价格进行排序  
 Collections.sort(books, new Comparator<Book>() {  
 @Override  
 public int compare(Book b1, Book b2) {  
 // 按价格升序排序  
 return Double.compare(b1.getPrice(), b2.getPrice());  
 }  
 });  
 // 打印排序后的书籍列表  
 System.out.println("Sorted Books by Price: " + books);  
 }  
}

‍

‍

‍

#### LinekdList

​LinkedList​ 是 List​ 接口的一个实现，它使用双向链表来存储数据，底层实现了双向链表和双端队列特点。与 ArrayList​ 相比，LinkedList​ 在数据存储方面有一些不同的特点：

1. **动态存储**：LinkedList​ 的大小不是固定的，它可以根据需要动态地增长。当添加新元素时，如果列表没有足够的空间，LinkedList​ 会自动分配新的内存空间。

1. **链表结构**：LinkedList​ 中的每个元素（节点）都包含数据和两个指针，分别指向前一个节点和后一个节点。这种结构使得 LinkedList​ 在插入和删除操作时非常高效，因为这些操作只需要改变指针的指向，而不需要移动其他元素。

1. **插入和删除效率极高**：由于 LinkedList​ 的节点之间是通过指针连接的，所以插入和删除操作的时间复杂度是 O(1)，即常数时间。这意味着无论列表的大小如何，插入和删除操作的效率都保持不变。

1. **顺序访问**：与 ArrayList​ 不同，LinkedList​ 不提供随机访问功能。要访问 LinkedList​ 中的元素，需要从头节点开始遍历，直到达到所需的位置。这使得查找操作在 LinkedList​ 中变得较慢，时间复杂度为 O(n)。

1. **内存占用**：由于 LinkedList​ 需要为每个节点存储额外的指针信息，因此在存储大量元素时，它可能会比 ArrayList​ 占用更多的内存。

1. **线程安全性**：LinkedList​ 不是线程安全的。如果多个线程同时访问 LinkedList​ 并进行操作，可能会导致数据不一致的问题。在这种情况下，需要使用额外的同步措施，例如使用 synchronized​ 关键字或特定的并发库来确保线程安全。  
   下面是一个简单的 LinkedList​ 使用示例：

import java.util.LinkedList;  
public class LinkedListExample {  
 public static void main(String[] args) {  
 // 创建一个空的LinkedList  
 LinkedList<String> linkedList = new LinkedList<>();  
 // 向LinkedList中添加元素  
 linkedList.add("Element1");  
 linkedList.add("Element2");  
 linkedList.add("Element3");  
 // 获取LinkedList的大小  
 System.out.println("LinkedList size: " + linkedList.size());  
 // 遍历LinkedList中的元素  
 for (String element : linkedList) {  
 System.out.println(element);  
 }  
 // 在LinkedList的开始添加元素  
 linkedList.addFirst("NewFirstElement");  
 // 在LinkedList的末尾添加元素  
 linkedList.addLast("NewLastElement");  
 // 删除LinkedList中的第一个元素  
 linkedList.removeFirst();  
 // 删除LinkedList中的最后一个元素  
 linkedList.removeLast();  
 }  
}

‍

‍

#### Vector

Vector 是 List 接口的一个实现，它是一个动态数组，可以确保线程安全。这意味着 Vector 的方法在多线环境中是同步的，可以同时被多个线程安全地访问。Vector 类直接继承自 AbstractList 类，并且实现了 List接口的所有方法。

以下是 Vector 的一些重要特性：

线程安全性：Vector 是线程安全的，它的所有公共方法都是同步的，这保证了在多线程环境中对 Vector 的操作不会出现数据竞争。

‍

‍

#### 总结

如何选择ArrayList和LinkedList:

1)如果我们改查的操作多，选择ArrayList

2)如果我们增删的操作多，选择LinkedList

3)一般来说，在程序中，80%-90%都是查询，因此大部分情况下会选择ArrayList

4)在一个项目中，根据业务灵活选择，也可能这样，一个模块使用的是ArrayList,另外一个模块是LinkedList.

​​

‍

​​

‍

‍

### Set接口

* 无序（添加和取出的顺序不一致）没有索引[后面演示]

* 不允许重复元素，所以最多包含一个u川

* JDK APIE中Set接口的实现类有：

public static void main(String[] args) {  
 Set set1=new HashSet();  
/\*\*  
 \*  
 \* 1. 无序（添加和取出的顺序不一致）没有索引，取出实际上是是固定的，每次取出都是一样的结果  
 \* 2. 不允许重复元素，所以最多包含一个u川  
 \*  
 \* \*\*/  
  
 set1.add("再别康桥");  
 set1.add("雨巷");  
 set1.add("雨巷");  
 set1.add("雨巷");  
 set1.add("雨巷");  
 set1.add("志摩的诗");  
 set1.add(null);  
 set1.add(null);  
  
 System.out.println(set1);

常用方法和Collection接口一样.,因为是Collection接口的子接口。**遍历Set接口对象（指的是实现了Set接口的实例化对象）**

* Set接口的遍历方式
* 1.可以使用迭代器
* 2.增强for
* 3.**不能使用索引的方式来获取.**

/\*\*  
 \*  
 \* 遍历Set接口对象（指的是实现了Set接口的实例化对象）  
 \* 1. 使用迭代器  
 \* 2.使用增强for  
 \*  
 \* \*\*/  
  
 System.out.println("-------迭代器--------");  
 Iterator iterator=set1.iterator();  
 while (iterator.hasNext()) {  
// next会返回当前元素并指向下一个元素  
 Object next = iterator.next();  
 System.out.println(next);  
   
 }  
  
 System.out.println("-------for循环--------");  
  
 for (Object o : set1) {  
 System.out.println(o);  
 }

‍

#### HashSet

HashSet它实现了Set接口，用来存储不包含重复元素的集合。在HashSet中，元素的位置是由元素的哈希值决定的，它使用哈希表来存储数据，这使得添加、删除和查找元素的操作都能达到较高的效率。

**底层的本质是：数组+链表+红黑树**

HashSet的特点主要包括：

* **无序性**：HashSet不保证元素的顺序，也就是说，元素在集合中的位置与它们的添加顺序无关。

* **基于哈希表**：HashSet的内部实现依赖于哈希表，这使得通常情况下，对HashSet的操作（如添加、删除和查找）的时间复杂度为O(1)。

* **允许null元素**：HashSet允许存储一个null元素。

* **非同步性**：HashSet不是线程安全的，如果多个线程同时访问一个HashSet实例，并且至少有一个线程对集合进行修改，那么必须外部同步。

* **哈希函数**：HashMap使用键对象的 hashCode()​ 方法计算哈希码，然后通过某种策略（如取模运算）将哈希码转换为数组的索引。

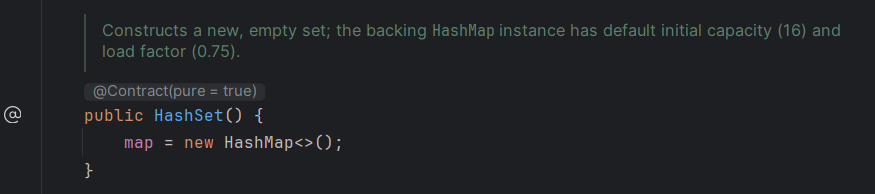
* **键的唯一性**：HashMap要求键的hashCode和equals方法一起保证键的唯一性。也就是说，两个键相等（equals()​ 返回 true​）时，它们的哈希码也应该相等；反之，哈希码相等的键不一定相等，但必须调用 equals()​ 方法来确认。

* **负载因子**：HashMap有一个名为负载因子的参数，它决定了何时自动调整其容量。默认负载因子为0.75，表示当HashMap中的元素个数达到了容量的75%时，会触发自动扩容操作。

* **扩容**：扩容时，HashMap会创建一个新的数组，容量通常是原数组的2倍加1，然后重新计算所有键值对的哈希位置，并将它们移动到新的数组中。

‍

* **HashSet的实现原理是使用哈希表来存储元素。**
* 每个元素都被映射到哈希表中的一个位置，这个位置是由元素的哈希值决定的。如果两个元素的哈希值相同，它们会被存储在哈希表中的同一个位置，这称为哈希冲突。HashSet通过链表来解决哈希冲突，当冲突发生时，新的元素会被添加到链表的末尾。
* ‍

* **在Java中，HashSet的实现实际上是依赖于HashMap的**。
* Map是双列所以需要一个PRESENT来占位
* 每个HashSet实例内部都有一个HashMap实例，用来存储实际的数据。当我们向HashSet添加元素时，**实际上是通过HashMap的put**​**方法来添加的，value**​**始终是PRESENT**​**，这是一个静态的最终字段，**用来标识在HashMap中的值。由于HashMap不允许键重复，这也就确保了HashSet中不会有重复的元素。
* HashSet本质是还是HashMap接口
* ​​
* ‍
* PRESENT本身是没有意义的。
* // Dummy value to associate with an Object in the backing Map  
   private static final Object PRESENT = new Object();
* ‍

‍

* **细节**
* ‍
* add添加元素后会返回一个布尔值，成功是true，错误是false

* 1. 能否添加多个内容相同的对象？
  + 返回结果为：  
    ​-------添加多个内容相同的实例类对象-------- [Dog{Dname='jzc'}, Dog{Dname='jzc'}]​。
  + 原因是对象地址不相同所以不是一样。
  + 只要设置重写**hashCode和equals方法就会被认为是同步一个元素**
  + System.out.println("-------添加多个内容相同的实例类对象--------");  
    // 添加多个内容相同的实例类对象  
      
     Set set02=new HashSet();  
      
     set02.add(new Dog("jzc"));  
     set02.add(new Dog("jzc"));  
      
     System.out.println(set02);  
      
      
     }

1. 相同字符串对象不能存储的原因

* 总结：表象相同就会被视为一个元素
* **由于String类已经重写了hashCode和equals方法，**这两个对象尽管是不同的实例，**但因为它们的内容相同，所以它们在HashSet中也会被视为同一个元素，最终只会在集合中保留一个。**

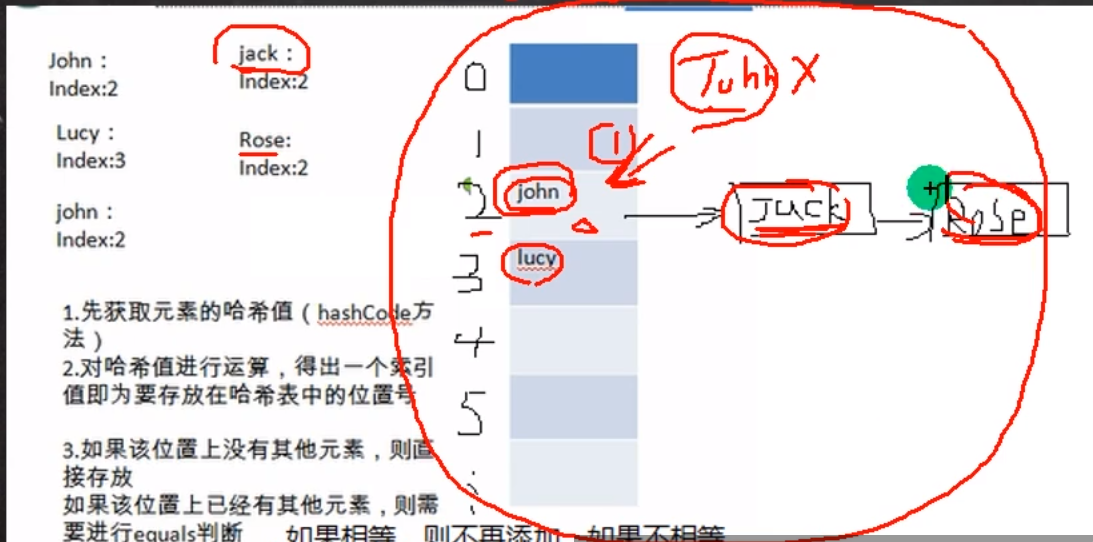
set02.add(new String("abc"));  
 set02.add(new String("abc"));  
  
 System.out.println(set02);

‍

‍

* **底层机制**
* 1.HashSet底层是HashMap
* 2.添加一个元素时，先得到hash值-会转成>索引值
* **根据key,得到hash去计算该key应该存放到table表的哪个索引位置**
* 3.找到存储数据表table,看这个索引位置是否已经存放的有元素
* 4.如果没有，直接加入
* 5.如果有调用equals()比较，如果相同，就放弃添动，双果不相同，则添加到最后
* 从多数组单链表转换到多数组树结构，目的是让存取效率更高。
* 6.在Java8中，如果一条链表的元素个数超过TREEIFY THRESHOLD(默认是8)，并且table的大小>=MIN TREEIFY CAPACITY(默认64)，就会进行树化（红黑树）

‍

​​

‍

‍

* **源码解读**

从add来进行解读

‍

1. 创建HashSet接口对象 Set set02=new HashSet();​实际上创建了HashMap接口对象，并赋值。

* public HashSet() {  
   map = new HashMap<>();  
   }

1. 调用add添加元素内容set02.add(new Dog("jzc"));​

* **是由map**​**中的put**​**方法来添加这个元素**，e​是我们传入的元素，PRESENT​是Map双列的一个占位符，key和value。而我们只传了一个元素
* public boolean add(E e) {  
   return map.put(e, PRESENT)==null;  
   }
* PRESENT本身是没有意义的一个空对象
* // Dummy value to associate with an Object in the backing Map  
   private static final Object PRESENT = new Object();

1. 执行put。该方法会执行hash(key)得到对应key的hash值

* public V put(K key, V value) {  
   return putVal(hash(key), key, value, false, true);  
   }
* hash(key)
* **根据key,得到hash去计算该key应该存放到table表的哪个索引位置**
* static final int hash(Object key) {  
   int h;  
  ///这么些是为了避免hash冲突，就是hash值相同。  
   return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
   }

1. 执行putVal

* 在Java HashMap的实现中，当向HashMap中添加一个新的键值对时，首先会通过散列函数计算出该键的哈希码（hash），然后使用这个哈希码确定元素在数组（称为table）中的索引位置。以下是对putVal​方法中将元素添加到链表部分的详细解释：
* final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {  
   // 获取当前的桶数组（table）  
   Node<K,V>[] tab;   
   // 获取table中对应索引位置的第一个节点  
   Node<K,V> p;   
   // 获取table的长度  
   int n, i;  
    
   // 如果table为空或长度为0，则进行初始化或扩容  
   if ((tab = table) == null || (n = tab.length) == 0)  
   n = (tab = resize()).length;  
    
   // 计算元素在table中的索引i  
   i = (n - 1) & hash;  
    
   // 当前索引位置为空时，直接新建节点并放入table  
   if ((p = tab[i]) == null)  
   //将key-value放入到一个node中  
   tab[i] = newNode(hash, key, value, null);  
    
   // 否则，表示当前位置已有节点，需要进一步处理  
   else {  
   // 遍历链表，寻找是否有已存在的键值对  
   Node<K,V> e;   
   K k;  
   // 如果找到了相同的键（根据哈希码和equals判断）  
   if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))  
   e = p;  
   // 如果当前节点是红黑树节点，调用相应的方法插入新节点  
   else if (p instanceof TreeNode)  
   e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  
   // 否则，遍历链表，直到找到空位或者相同键的节点  
   else {  
   for (int binCount = 0; ; ++binCount) {  
   if ((e = p.next) == null) {  
   // 链表末尾没有相同键的节点，就在末尾新建节点  
   p.next = newNode(hash, key, value, null);  
    
   // 当链表长度达到阈值时，转化为红黑树  
   if (binCount >= TREEIFY\_THRESHOLD - 1) // -1是因为第一个节点不计入数量  
   treeifyBin(tab, hash);  
    
   break;  
   }  
   // 如果找到了相同的键，结束循环  
   if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))  
   break;  
   p = e;  
   }  
   }  
    
   // 如果找到了相同的键，更新其值（如果onlyIfAbsent为false或旧值为null）  
   if (e != null) {  
   V oldValue = e.value;  
   if (!onlyIfAbsent || oldValue == null)  
   e.value = value;  
   afterNodeAccess(e);  
   return oldValue;  
   }  
   }  
    
   // 更新修改次数和容量大小，并在必要时进行扩容  
   ++modCount;  
   if (++size > threshold)  
   resize();  
   afterNodeInsertion(evict);  
   return null;  
  }
* 在这段代码中，可以看到当新的元素需要添加到链表时，它会被作为一个新的节点创建，并插入到链表的适当位置（即基于哈希码计算得到的数组索引位置）。如果链表长度过长，甚至会将其转换为红黑树以优化查找和插入性能。同时，如果发现键已经存在，那么会根据条件决定是否更新对应的值。

‍

‍

#### LinkedHashSet

‍

​LinkedHashSet​ 是 HashSet​ 和 LinkedList​ 特性的结合体。具体来说，LinkedHashSet​ 继承自 HashSet​，这意味着它同样遵守集合的数学特性——无序且不包含重复元素（每个元素唯一）。

以下是 LinkedHashSet​ 主要特点的总结：

* **不重复元素**：如同 HashSet​，不允许存储重复的元素，依据 .equals()​ 方法检查元素是否相等。

* **保持插入顺序**：插入元素时会维护一个**双链表**，因此集合的迭代顺序与元素插入顺序一致。

* **性能**：相对于仅基于哈希表的 HashSet​，LinkedHashSet​ 的插入、删除和查找操作可能会稍慢，因为它需要额外维护链表结构。

* **线程安全性**：默认情况下不是线程安全的，如果需要在多线程环境下使用，需考虑同步控制或其他并发集合类（如 Collections.synchronizedSet()​）。

* **允许存储 null 值**：LinkedHashSet​ 允许存储一个 null 值，但仅能存储一个。

总的来说，如果你的应用场景既需要集合的无重复特性，又希望保持元素的插入顺序，那么 LinkedHashSet​ 就是一个理想的选择。

‍

‍

## Map集合

Map集合是Java中一种非常重要的数据结构，它用于存储键值对（key-value）形式的 data。在Map集合中，每个元素都由一个键（key）和一个值（value）组成，键和值之间存在一种映射关系。

Map集合的特点如下：

1. **键唯一：Map中的每个key键都是唯一的，不能有重复的键。但是可以有多个相同的value**

1. 无序集合：Map中的元素没有特定的顺序，访问顺序也不保证与插入顺序相同。

1. 键值对：每个元素都是一个键值对（key-value），键用于查找对应的值。

1. 高效的键查找：通过键来查找值非常高效，时间复杂度接近O(1)。

1. 如果添加相同的key,则会覆盖原来的key-val,等同于修改.(key不会替换，val会替换)

1. 在Java8中，如果一条链表的元素个数超过TREEIFY THRESHOLD(默认是8)，并且table的大小>=MIN TREEIFY CAPACITY(默认64)，就会进行树化（红黑树）

‍

Java提供了多种Map的实现，包括：

* **HashMap**：基于哈希表实现，提供了键值映射，访问速度快，但是不保证顺序，且允许键和值为null。

* TreeMap：基于红黑树实现，能够对键进行排序，但是相比HashMap来说，查找速度慢，因为它需要维护树的结构。

* LinkedHashMap：维护了一个双向链表，它可以在迭代时保持插入的顺序，但是它比HashMap要慢，因为它需要维护链表结构。

* Hashtable：与HashMap类似，但它所有的方法都是同步的，线程安全，但是效率比HashMap低。

‍

在实际应用中，Map集合常用于：

* 存储用户信息，如数据库中的记录。

* 缓存机制中，存储 key-value 对。

* 映射表，如语言翻译字典。

* 各种配置信息，如系统设置或应用程序的配置参数。

‍

Map集合提供了丰富的操作方法，包括put(K key, V value)​来添加键值对，get(Object key)​来通过键获取值，entrySet()​来获取所有的键值对，keySet()​来获取所有的键，以及values()​来获取所有的值等。  
遍历Map集合有多种方式，例如：

1. 使用entrySet()​配合Iterator​遍历。

1. 使用keySet()​遍历键，然后通过get()​方法获取值。

1. 使用增强型for​循环直接遍历entrySet()​。

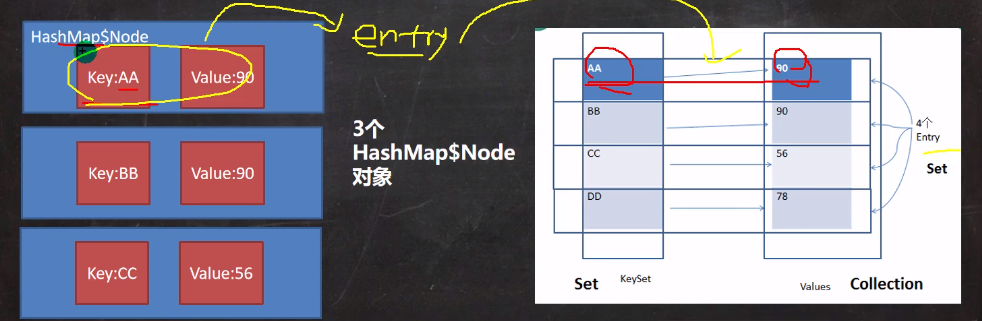
1. 使用Map.Entry​类型的增强型for​循环遍历。  
   选择哪种遍历方式取决于具体需求和性能考量。

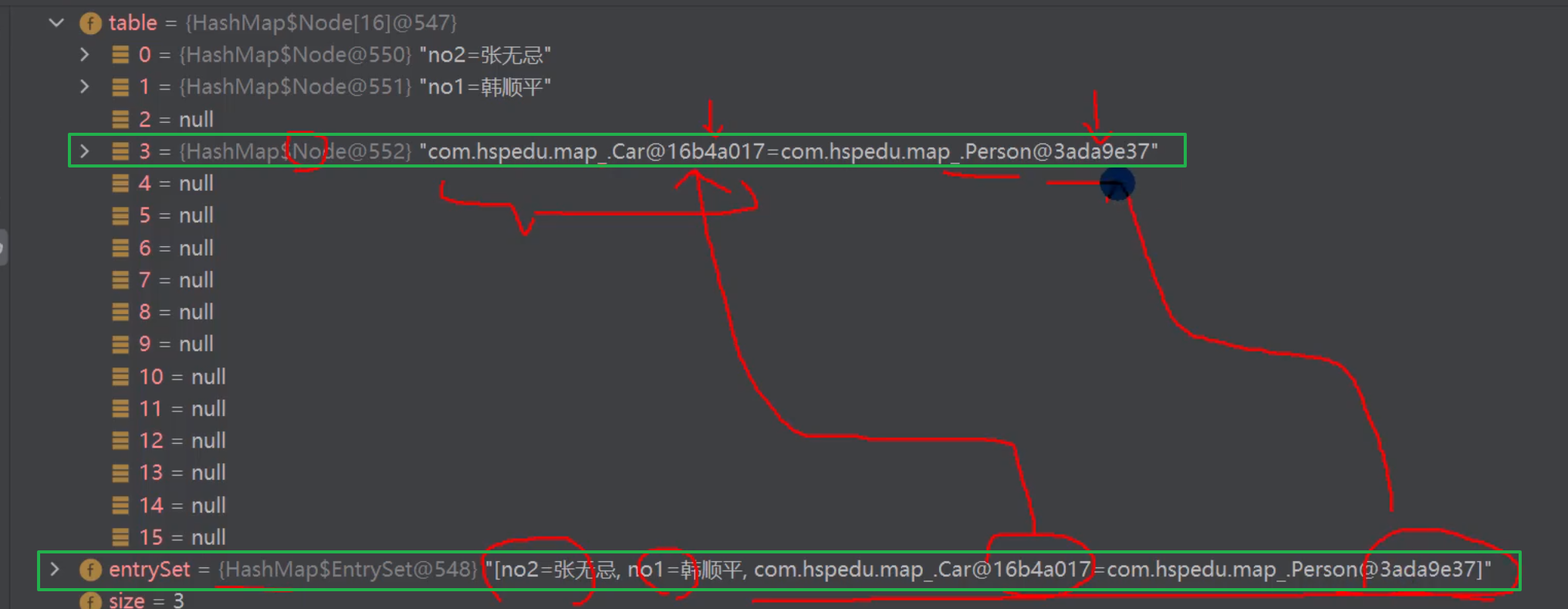
‍

#### EntrySet

目的是为了方便我们存取和管理node节点中的数据

key-value的存储实际上是在一个node节点中。

​​

​​

//将key-value放入到一个node中  
 tab[i] = newNode(hash, key, value, null);

1. k-V为了方便程序员的遍历，还会创建EntrySet集合，

1. entrySet中，定义的类型是Map.Entry,但是实际上存放的还是HashMap$Node

Map接口实现键值对的方式是通过内部的一个叫做Entry​的类，这个类同时实现了Map.Entry接口。

* 每个Entry​对象实际上就是一个键值对，它有getKey()​和getValue()​方法，可以分别用来获取键和值。

import java.util.HashMap;  
import java.util.Map;  
public class MapExample {  
 public static void main(String[] args) {  
 // 创建一个Map对象，用来存储水果和对应的数量  
 Map<String, Integer> fruitStock = new HashMap<>();  
 // 向Map中添加键值对  
 fruitStock.put("苹果", 10);  
 fruitStock.put("香蕉", 20);  
 fruitStock.put("橙子", 30);  
 // 通过键来获取值  
 int appleQuantity = fruitStock.get("苹果");  
 System.out.println("苹果有：" + appleQuantity + "个");  
 // 遍历Map中的所有键值对  
 for (Map.Entry<String, Integer> entry : fruitStock.entrySet()) {  
 String fruit = entry.getKey();  
 int quantity = entry.getValue();  
 System.out.println(fruit + "有：" + quantity + "个");  
 }  
 }  
}

‍

‍

### HashMap

‍

除了基本操作方法不同，其他与HashSet相同，所以不再介绍

#### 基本方法

* ​put(K key, V value)​: 添加键值对到Map中。

* ​size()​: 返回Map中的元素数量。

* ​containsKey(Object key)​: 检查Map是否包含指定的键。

* ​get(Object key)​: 根据键获取对应的值。

* ​containsValue(Object value)​: 检查Map是否包含指定的值。

* ​remove(Object key)​: 删除指定键所对应的键值对。

* ​clear()​: 删除Map中的所有键值对。

* ​entrySet()​: 返回一个包含Map中所有映射关系（键值对）的Set视图。

* ​keySet()​: 返回一个包含Map中所有键的Set视图。

* ​values()​: 返回一个包含Map中所有值的Collection视图。

import java.util.\*;  
  
public class MapMethodsDemo {  
 public static void main(String[] args) {  
 // 创建一个HashMap实例  
 Map<String, Integer> map = new HashMap<>();  
  
 // 添加元素  
 map.put("Apple", 1);  
 map.put("Banana", 2);  
 map.put("Cherry", 3);  
  
 // 显示集合大小  
 System.out.println("Size of the map: " + map.size());  
  
 // 判断是否包含某个键  
 System.out.println("Does map contain 'Apple'? " + map.containsKey("Apple"));  
  
 // 根据键获取值  
 System.out.println("Value associated with 'Apple': " + map.get("Apple"));  
  
 // 判断是否包含某个值  
 System.out.println("Does map contain the value 2? " + map.containsValue(2));  
  
 // 删除元素  
 map.remove("Banana");  
 System.out.println("After removing 'Banana': " + map);  
  
 // 清空整个Map  
 map.clear();  
 System.out.println("After clearing the map: " + map);  
  
 // 遍历Map  
 map.put("Apple", 1);  
 map.put("Banana", 2);  
 for (Map.Entry<String, Integer> entry : map.entrySet()) {  
 System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());  
 }  
  
 // 使用键集合遍历  
 //map.keySet()取出所有的key  
 for (String key : map.keySet()) {  
 System.out.println("Key: " + key);  
 }  
  
 // 使用值集合遍历  
 for (Integer value : map.values()) {  
 System.out.println("Value: " + value);  
 }  
  
 //使用遍历器  
// 这是entey里面的方法keySet()  
// 通过key获取value  
 Set keyset =map.keySet();  
 Iterator iterator=keyset.iterator();  
 while (iterator.hasNext()) {  
 Object key = iterator.next();  
 System.out.println(key+"-"+map.get(key));  
   
 }  
 }  
}

‍

‍

案例：

public class HashMap02 {  
 public static void main(String[] args) {  
  
 /\*\*  
 \*  
 \* 实现案例：  
 \* 使用HashMap添加3个员工对象，要求  
 \* 键：员工id  
 \* 值：员工对象  
 \* 并遍历显示工资>4500的员工（遍历方式最少两种）  
 \* 员工类：姓名、工资、员工d  
 \*  
 \*  
 \*  
 \* \*\*/  
  
 Map hashmap02=new HashMap();  
  
 hashmap02.put("niuma001",new Emp("牛马打工人1","niuma001",4000));  
 hashmap02.put("tianxuan001",new Emp("天选打工人1","tianxuan001",14000));  
 hashmap02.put("niuma002",new Emp("牛马打工人2","niuma002",4500));  
 hashmap02.put("niuma003",new Emp("牛马打工人3","niuma003",4700));  
 hashmap02.put("niuma004",new Emp("牛马打工人4","niuma004",4070));  
  
 Set keyset=hashmap02.keySet();  
 Iterator iterator=keyset.iterator();  
 while (iterator.hasNext()) {  
 Object key = iterator.next();  
// 将获取到的的value转化为emp对象方便获取对象数据。  
 Emp emp =(Emp)hashmap02.get(key);  
 if(emp.getEmpSalary()>=4500){  
 System.out.println(emp);  
 }  
  
 }  
  
  
 }  
}  
  
class Emp{  
 public String EmpName;  
  
 public String EmpID;  
  
 public String getEmpName() {  
 return EmpName;  
 }  
  
 @Override  
 public String toString() {  
 return "Emp{" +  
 "EmpName='" + EmpName + '\'' +  
 ", EmpID='" + EmpID + '\'' +  
 ", EmpSalary=" + EmpSalary +  
 '}';  
 }  
  
 public void setEmpName(String empName) {  
 EmpName = empName;  
 }  
  
 public String getEmpID() {  
 return EmpID;  
 }  
  
 public void setEmpID(String empID) {  
 EmpID = empID;  
 }  
  
 public int getEmpSalary() {  
 return EmpSalary;  
 }  
  
 public void setEmpSalary(int empSalary) {  
 EmpSalary = empSalary;  
 }  
  
 public Emp(String empName, String empID, int empSalary) {  
 EmpName = empName;  
 EmpID = empID;  
 EmpSalary = empSalary;  
 }  
  
 public int EmpSalary;  
  
}

‍

‍

‍

‍

### HashTable

**Java HashTabl**继承自 Dictionary​ 类，并实现了 Map​ 接口。Collections​ 工具类提供的线程安全包装器以及 ConcurrentHashMap​ 的引入，HashTable逐渐不再作为首选的线程安全映射容器。

#### 主要特性

1. **线程安全**：HashTable的所有操作都是同步的，这意味着在同一时间只有一个线程可以修改它。为了保证线程安全，HashTable中的每个方法都被synchronized关键字修饰，这确保了在多线程环境下不会出现数据不一致的情况，但同时也增加了锁竞争和降低了并发性能。

1. **键值对存储**：像其他的Map实现一样，HashTable存储的是键值对（key-value pairs），键和值都可以是任意对象类型。不过，HashTable并不允许键或值为null，尝试将null作为键或值插入会抛出NullPointerException。

1. **哈希策略**：HashTable通过调用键对象的 hashCode()​ 方法生成哈希码，然后使用内置的哈希算法将哈希码转换为数组索引。当发生哈希冲突时，采用开放地址法（链地址法）解决冲突，即将具有相同哈希码的对象放在同一桶（bucket）中的链表上。

1. **增长策略**：当HashTable的容量不足以容纳更多元素时，它会自动扩容，扩容后的新容量通常是原始容量的两倍加1，并重新计算和分配所有的键值对。

1. **性能**：由于线程安全的实现方式，HashTable在多线程环境下的读写性能相较于非线程安全的HashMap更低。尤其是在高并发场景下，由于频繁的锁竞争可能导致性能瓶颈。

#### 示例代码

import java.util.Hashtable;  
  
public class HashTableExample {  
 public static void main(String[] args) {  
 // 创建一个HashTable实例  
 Hashtable<String, Integer> hashtable = new Hashtable<>();  
  
 // 添加元素  
 hashtable.put("Apple", 1);  
 hashtable.put("Banana", 2);  
 hashtable.put("Cherry", 3);  
  
 // 根据键获取值  
 int appleCount = hashtable.get("Apple");  
  
 // 判断是否包含某个键  
 boolean containsKey = hashtable.containsKey("Banana");  
  
 // 删除键值对  
 hashtable.remove("Cherry");  
  
 // 遍历HashTable  
 for (Map.Entry<String, Integer> entry : hashtable.entrySet()) {  
 System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());  
 }  
 }  
}

#### 注意事项

* **线程安全成本**：虽然HashTable是线程安全的，但因其全局锁定策略，在多线程环境下可能会导致性能下降，尤其是当多个线程需要同时读写不同键的时候。

* **同步粒度**：HashTable的所有操作都在一个全局锁上进行，这意味着即使是在读操作期间，也可能阻止其他线程的写操作。现代Java集合框架提供了更细粒度的线程安全控制，如ConcurrentHashMap，它可以实现分区锁从而提高并发性能。

* **不推荐使用**：鉴于以上原因，对于新的项目开发，一般建议使用HashMap（非线程安全，但性能更高）或ConcurrentHashMap（线程安全且性能更好），而非HashTable。在需要兼容旧代码或特别重视线程安全且性能要求不高的场景下，才可能考虑使用HashTable。

‍

### Properties

​Properties​ 类是 Map​ 接口的一个实现，**也是HashTable的一个子类，**它特别用于处理属性文件，这些文件通常以键值对的形式存储信息。Properties​ 类提供了加载和保存属性文件的方法，以及其他一些管理属性集合的实用方法。

#### 1. Properties 的特点

* **键和值的类型**：在 Properties​ 中的键和值都是 String​ 类型的。

* **无序集合**：Properties​ 中的元素没有特定的顺序。

* **线程安全**：Properties​ 是线程安全的，它可以被多个线程共享和修改，而不会导致数据竞争。

* **兼容性**：Properties​ 类经常用于读取和写入配置文件，这些配置文件在 Java 应用中广泛使用。

#### 2. Properties 的主要方法

* **load(InputStream in)**：从指定的 InputStream​ 中加载属性文件。

* **store(OutputStream out, String comments)**：将属性列表保存到指定的 OutputStream​ 中。

* **setProperty(String key, String value)**：设置属性文件中的键值对。

* **getProperty(String key)**：获取属性文件中与指定键关联的值。

* **containsKey(String key)**：检查属性文件中是否包含指定的键。

* **size()**：返回属性文件中键值对的数量。

* **isEmpty()**：检查属性文件是否为空。

* **keys()**：获取属性文件中所有键的集合。

* **clear()**：清空属性文件中的所有键值对。

#### 3. 使用示例

以下是一个简单的 Properties​ 使用示例：

import java.io.FileInputStream;  
import java.io.IOException;  
import java.util.Properties;  
public class PropertiesExample {  
 public static void main(String[] args) {  
 Properties properties = new Properties();  
 try {  
 // 从文件加载属性  
 properties.load(new FileInputStream("config.properties"));  
   
 // 获取属性值  
 String propertyValue = properties.getProperty("username");  
 System.out.println("Username: " + propertyValue);  
   
 // 设置属性值  
 properties.setProperty("password", "newPassword123");  
   
 // 存储属性到文件  
 properties.store(new FileOutputStream("config.properties"), "Updated configuration");  
   
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

在这个示例中，我们首先创建了一个 Properties​ 对象，然后通过 load​ 方法从名为 config.properties​ 的文件中加载属性。接着，我们使用 getProperty​ 方法获取键为 username​ 的值，并使用 setProperty​ 方法设置一个新的键值对。最后，我们使用 store​ 方法将更改后的属性保存回文件。

#### 4. 注意事项

* 在使用 Properties​ 类时，通常会通过 FileInputStream​ 或 FileOutputStream​ 来读写文件，这些文件通常是 .properties​ 格式的，这种格式使用 =​ 来分隔键和值，例如：username=admin​。

* ​Properties​ 类中的方法大多数都是同步的，这意味着它们可以安全地被多个线程同时调用。

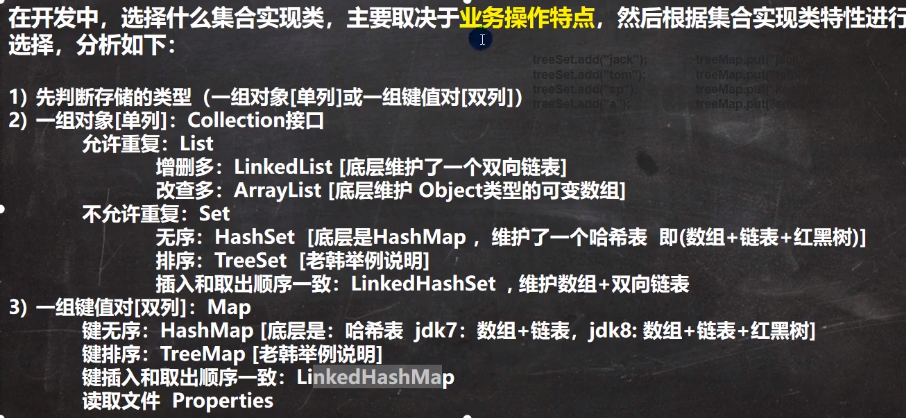
* 在保存属性文件时，如果文件不存在，store​ 方法会创建一个新的文件。如果文件已经存在，store​ 方法会覆盖现有的文件。  
  ​Properties​ 类是 Java 中处理属性文件的标准方式，它在 Java 应用程序中提供了方便的键值对存储和管理功能。

‍

‍

## 集合的选择

‍

​​

‍

在 Java 标准库中，有多种 List​ 和 Map​ 实现类，它们适用于不同的使用场景。以下是一些广泛应用于常见场景的 List​ 和 Map​ 实现类：

### List 实现类

1. **ArrayList**

* + **使用场景**：需要一个可变的、大小可变的数组。适用于频繁的查找、更新操作。

* + **主要功能**：提供所有可选的列表操作，允许包含重复的元素和 null​ 值。

1. **LinkedList**

* + **使用场景**：需要在列表中间进行频繁的插入和删除操作。

* + **主要功能**：双向链表实现，适用于在列表中间进行快速的插入和删除操作。

1. **Vector**

* + **使用场景**：线程安全的 List​ 实现，适用于多线程环境。

* + **主要功能**：与 ArrayList​ 类似，但所有方法都是同步的。

1. **Stack**

* + **使用场景**：实现栈的数据结构，主要用于后进先出（LIFO）操作。

* + **主要功能**：内部使用 Vector​ 实现，所有操作都是同步的。

1. **HashSet**

* + **使用场景**：需要一个不包含重复元素的 Set​。

* + **主要功能**：基于哈希表实现，提供快速的查找、插入和删除操作。

### Map 实现类

1. **HashMap**

* + **使用场景**：需要一个键值对映射，不需要顺序。

* + **主要功能**：基于哈希表实现，提供快速的查找、插入和删除操作。

1. **TreeMap**

* + **使用场景**：需要一个有序的键值对映射，通常用于排序键或需要按顺序访问键。

* + **主要功能**：基于红黑树实现，维护键的顺序，支持排序和范围查找。

1. **LinkedHashMap**

* + **使用场景**：需要一个保持插入顺序的 Map​。

* + **主要功能**：内部使用链表维护插入顺序，适合需要顺序访问和频繁更新的场景。

1. **Hashtable**

* + **使用场景**：线程安全的 Map​ 实现，适用于多线程环境。

* + **主要功能**：与 HashMap​ 类似，但所有方法都是同步的。

1. **Properties**

* + **使用场景**：处理属性文件，通常用于配置信息。

* + **主要功能**：继承自 Hashtable​，专用于处理属性键值对。  
    在选择具体的实现类时，应该根据实际需求和场景来决定。例如，如果需要频繁的插入和删除操作，应该选择 LinkedList​ 而不是 ArrayList​。如果需要在多线程环境中使用，应该选择 Vector​ 或 Hashtable​。如果需要保持元素插入顺序，应该选择 LinkedHashMap​。如果需要键值对有序，应该选择 TreeMap​。在性能和资源利用方面，也需要考虑这些因素。

‍

## Collection工具类

​java.util.Collections​ 是Java集合框架中的一个工具类，它提供了大量的静态方法来帮助我们操作和管理各种集合（包括List、Set和Map）。以下是一些Collections​工具类中常用的方法及其功能简介和演示：

### 1. 排序操作

* **sort(List<T> list)**：对List进行自然排序（要求元素实现Comparable接口）。

List<Integer> numbers = Arrays.asList(5, 3, 8, 1, 9);  
Collections.sort(numbers);  
System.out.println(numbers); // 输出 [1, 3, 5, 8, 9]

* **sort(List<T> list, Comparator<? super T> c)**：对List进行自定义排序。

List<String> names = Arrays.asList("Tom", "Jerry", "Alice", "Bob");  
Collections.sort(names, Comparator.comparing(String::length));  
System.out.println(names); // 输出 ["Bob", "Tom", "Jerry", "Alice"]

### 2. 反转操作

* **reverse(List<?> list)**：反转List中的元素顺序。

List<String> words = Arrays.asList("Hello", "World", "Java");  
Collections.reverse(words);  
System.out.println(words); // 输出 ["Java", "World", "Hello"]

### 3. 查找和替换

* **indexOfSubList(List<?> source, List<?> target)**：查找子列表在源列表中的起始索引。

* **lastIndexOfSubList(List<?> source, List<?> target)**：查找子列表在源列表中最后一个匹配项的起始索引。

* **replaceAll(List<T> list, T oldVal, T newVal)**：替换List中所有oldVal为newVal。

List<Integer> nums = Arrays.asList(1, 2, 3, 2, 4, 2);  
Collections.replaceAll(nums, 2, 5);  
System.out.println(nums); // 输出 [1, 5, 3, 5, 4, 5]

### 4. 随机化操作

* **shuffle(List<?> list)**：随机打乱List中的元素顺序。

List<Integer> shuffled = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));  
Collections.shuffle(shuffled);  
System.out.println(shuffled); // 输出每次运行可能不同，如：[3, 1, 5, 2, 4]

### 5. 其他操作

* **fill(List<? super T> list, T obj)**：使用给定的对象填充整个List。

* **copy(List<? super T> dest, List<? extends T> src)**：将src列表中的所有元素复制到dest列表中。

* **max(Collection<? extends T> coll, Comparator<? super T> comp)**：返回给定集合的最大元素（根据Comparator）。

* **min(Collection<? extends T> coll, Comparator<? super T> comp)**：返回给定集合的最小元素（根据Comparator）。

* **binarySearch(List<? extends Comparable<? super T>> list, T key)**：在已排序的List中进行二分查找。

* **unmodifiable…(Collection<? extends T> c)**：返回不可变的集合视图，防止集合被修改。

请注意，以上代码仅为简化的示例，实际应用中应根据具体需求选择合适的方法。在处理集合时，确保了解方法对集合的影响以及线程安全性等因素。

# 泛型

泛型可以约束存入数据的类型，简化存入和去除的类型转火的过程。

泛型在没指定之前可以是任意的类型，由程序员指定为具体类型。

### 泛型的基本概念

泛型本质上是类型参数化的概念。在Java中，你可以定义一个类、接口或方法时使用一个或多个类型参数，这些类型参数在实例化时会被具体的类型所替换。

class Book<E>{  
 E BookName;  
  
 Book<String> book1 = new Book<String>("哈利波特");  
  
 \* 指定为String类型后就会变成这样。编译时类型就确定为所指定的String  
 \*  
 \*class Book<String>{  
 \* String BookName;

## 泛型的使用

### 泛型类

定义一个泛型类时，在类名后面添加一对尖括号，里面写上类型参数：

传入Integer后，T就替换成Integer类型

public class Box<T> {  
 private T t;  
 public void set(T t) {  
 this.t = t;  
 }  
 public T get() {  
 return t;  
 }  
}

使用时，你需要指定具体的类型：

Box<Integer> integerBox = new Box<Integer>();  
integerBox.set(10); // 自动装箱

### 泛型接口

接口中，静态变量也不能使用泛型，而方法可以

泛型接口的类型，在继承接口或者实现接口时确定

没有指定类型，默认为Object

泛型接口与泛型类的定义方式类似：

public interface Comparable<T> {  
 public int compareTo(T o);  
}

使用：

public class gen03 implements shape<String> {  
  
 @Override  
 public void A(String o) {  
  
 }

### 泛型方法

泛型方法，可以定义在普通类中，也可以定义在泛型类中

当泛型方法被调用时，类型会确定

泛型方法 public static <T> void printArrays

使用了泛型的方法：public static void printArray(T[] array)

你还可以定义一个带有类型参数的方法，而不需要将整个类泛型化：

public static <T> void printArray(T[] array) {  
 for (T element : array) {  
 System.out.print(element + " ");  
 }  
 System.out.println();  
}

当调用方法时传入参数，编译器就会确定类型。使用：

Integer[] array1={123,3,123,5,13};  
  
 Animal.printArray(array1);

### 泛型的继承

泛型不具有继承性

以下是错误的：

List<Object>list new ArrayList<String>()

### 类型通配符

类型通配符通常用?表示，用于灵活地指定泛型类型。它可以分为无界通配符和有界通配符。

### 无界通配符

无界通配符?表示任何类型：

List<?> list = new ArrayList<String>();  
list = new ArrayList<Integer>();

### 有界通配符

有界通配符通过? extends或? super来指定上界或下界：

1. ？extends AA表示上限，可以接受AA或者AA子类

1. super 子类类名AA: 支持AA类以及AA类的父类，不限于直接父类，

// 上界通配符  
List<? extends Number> list = new ArrayList<Integer>();  
// 下界通配符  
List<? super Integer> list = new ArrayList<Number>();

### 泛型的擦除

Java的泛型是伪泛型，因为泛型信息只存在于编译阶段，在运行时泛型信息会被擦除。这意味着泛型类型在运行时会被转换为它们的原生类型（Raw Type），即List<Integer>和List<String>在运行时都是List类型。

## 细节

* 不能使用基本数据类型作为泛型类型的参数，需要使用它们的包装类。

* 不能创建泛型数组。

* 不能直接实例化类型参数，例如new T()。

* 泛型类不能继承Exception或Throwable。

* 不能在静态字段或方法中引用类型变量。

1. 泛型的数据类型要求是引用类型不能是基本类型（所以需要做成包装类）

1. 指定泛型的类型后可以传入该类型的以及其子类型

1. 泛型的使用形式，接口和实现类的泛型是一样的可以。

1. 不指定集合的泛型默认是Object类型。

推荐使用的写法，有左边的编译类型即可

Book<String> book1 = new Book<>("哈利波特");

### 案例

package Egenericity;  
  
import java.util.\*;  
  
public class genericity02 {  
 public static void main(String[] args) {  
 /\*  
 \*  
 \*  
 \* 定义Employee类  
 1)该类包含：privatel成员变量name,sal,birthday,其中birthday为MyDate类的对象：  
 2)为每一个属性定义getter,,setter方法；  
 3)重写toString方法输出name,sal,birthday  
 4)MyDate类包含：private成员变量nonth,day,year;并为每一个属性定义getter,setter方法：  
 5)创建该类的3个对像，并把这些对象放入ArrayList集合中(ArrayList需使用泛型来定义)，对集合中的元素进行排序，并遍历输出：  
  
 排序方式：调用ArrayList的sort方法，传入Comparator对像[使用泛型，先按照name排序，如果name相同，则按生日日期的先后排序。【即：定制排序】  
 \*  
 \*  
 \* \*/  
  
 List<Employee> list=new ArrayList<>();  
 list.add(new Employee("张三",4120,new MyDate(23,2,2007)));  
 list.add(new Employee("李四",4000,new MyDate(22,5,2010)));  
 list.add(new Employee("张三",4520,new MyDate(22,5,1992)));  
 list.add(new Employee("王五",4120,new MyDate(23,2,2003)));  
 list.add(new Employee("来福",4120,new MyDate(23,2,2002)));  
  
// 自定义排序规则  
 list.sort(new Comparator<Employee>(){  
 @Override  
 public int compare(Employee o1, Employee o2) {  
// 使用姓名来排序，  
 /\*\*  
 \*  
 \*  
 \* CompareTo方法已经对中文字符串进行了很好的支持，因为它使用了字符串中字符的Unicode编码来进行比较。汉字在Unicode中是按照拼音顺序排列的，  
 \*  
 \* \*\*/  
 return Comparator1 ( o1, o2);  
 }  
 private int Comparator1(Employee o1, Employee o2) {  
 int a= o1.getEmpName().compareTo(o2.getEmpName());  
 if (a==0){  
 return o1.getBirthday().Comparator(o2);  
 }else return a;  
 }  
 });  
 Iterator<Employee> iterator = list.iterator();  
 while (iterator.hasNext()) {  
 Employee next = iterator.next();  
 System.out.println(next);  
 }  
 }  
  
}  
  
class Employee{  
 private String EmpName;  
 private double salary;  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Employee employee = (Employee) o;  
 return Double.compare(salary, employee.salary) == 0 && Objects.equals(EmpName, employee.EmpName) && Objects.equals(birthday, employee.birthday);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.hash(EmpName, salary, birthday);  
 }  
  
 private MyDate birthday;  
  
 public Employee(String empName, double salary, MyDate birthday) {  
 EmpName = empName;  
 this.salary = salary;  
 this.birthday = birthday;  
 }  
  
 @Override  
 public String toString() {  
 return "Employee{" +  
 "EmpName='" + EmpName + '\\'' +  
 ", salary=" + salary +  
 ", birthday=" + birthday +  
 '}';  
 }  
  
 public String getEmpName() {  
 return EmpName;  
 }  
  
 public void setEmpName(String empName) {  
 EmpName = empName;  
 }  
  
 public double getSalary() {  
 return salary;  
 }  
  
 public void setSalary(double salary) {  
 this.salary = salary;  
 }  
  
 public MyDate getBirthday() {  
 return birthday;  
 }  
  
 public void setBirthday(MyDate birthday) {  
 this.birthday = birthday;  
 }  
}  
class MyDate {  
 private int day;  
 private int month;  
  
 private int year;  
  
 @Override  
 public String toString() {  
 return year+"-"+ month+"-"+day;  
 }  
  
 public MyDate(int day, int month, int year) {  
 this.day = day;  
 this.month = month;  
 this.year = year;  
 }  
  
 public int getDay() {  
 return day;  
 }  
  
 public void setDay(int day) {  
 this.day = day;  
 }  
  
 public int getMonth() {  
 return month;  
 }  
  
 public void setMonth(int month) {  
 this.month = month;  
 }  
  
 public int getYear() {  
 return year;  
 }  
  
 public void setYear(int year) {  
 this.year = year;  
 }  
  
 public int Comparator(Employee o2){  
 if (year<o2.getBirthday().getYear()){  
 return -1;  
 } else if (month<o2.getBirthday().getMonth()) {  
 return -1;  
 }else if (day<o2.getBirthday().getDay()) {  
 return -1;  
 }  
 return 0;  
 }  
  
}

在Java中，Comparator接口的compare方法的返回值决定了两个比较对象在排序时的顺序。compare方法的返回值是一个整数，它有三种可能的值：

* 负整数：如果第一个参数应该排在第二个参数之前，则返回一个负整数。

* 零：如果两个参数相等，即它们的顺序相同，则返回零。

* 正整数：如果第一个参数应该排在第二个参数之后，则返回一个正整数。

例如，如果您正在比较两个整数，并且您希望按升序排序，那么您的compare方法可能会这样实现：

@Override  
public int compare(Integer o1, Integer o2) {  
 return o1 - o2; // 升序排序  
}

## 自定义泛型

自定义泛型,Animal类后面有泛型就是自定义泛型

* 2.E T Q 都是泛型的标识符，一般是单个的大写字母

* 3.泛型的标识符可以有多个

* 4.普通成员可以使用泛型（属性，方法）

* 5.使用泛型的数组不能初始化，因为没有确定的类型

* 6.静态方法和属性不能使用泛型，因为静态是加载类的时候就加载了，而具体的类型是创建对象才确定。

class Animal<E,T,Q>{  
  
 String AnimalName;  
  
 E e;  
 T t;  
 Q q;  
  
 public Animal(String animalName, E e, T t, Q q) {  
 AnimalName = animalName;  
 this.e = e;  
 this.t = t;  
 this.q = q;  
 }

# JUnit

一个类有很多功能代码需要测试，为了测试，就需要写入到main方法中如果有多个功能代码测试，就需要来回注销，切换很麻烦

使用方法：

1. 在方法上添加注解：@Test

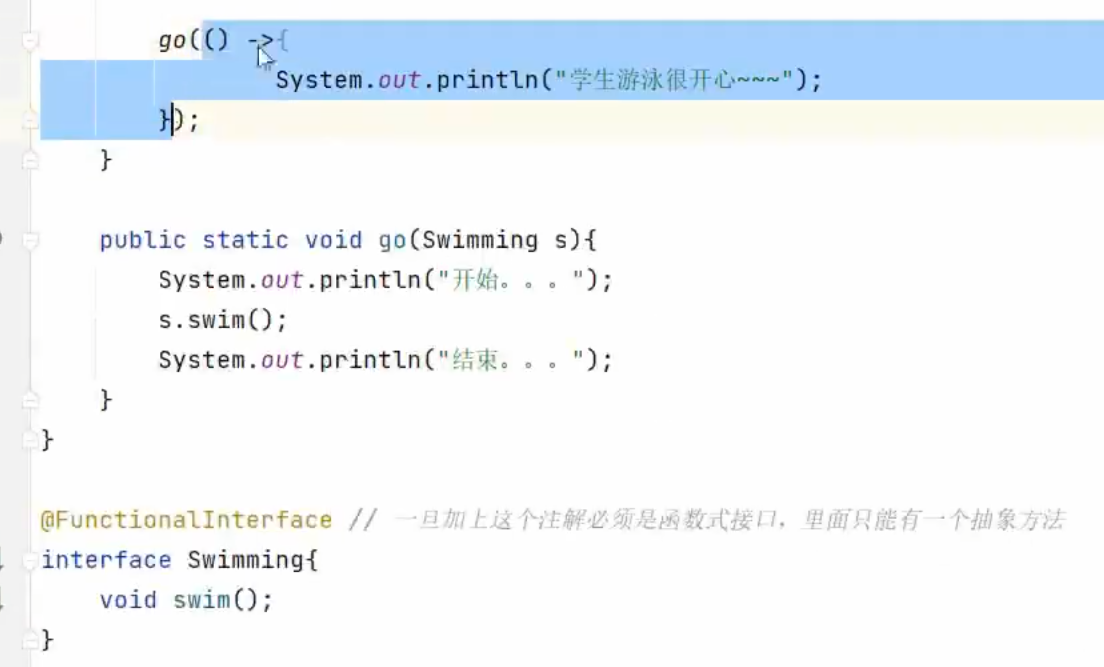
# JDK8

‍

## Lambd表达式

jdk8之后的新语法，作用是简化匿名内部类代码的写法只能简化接口中只有一个抽象方法的医名内部类形式，后续较为常用

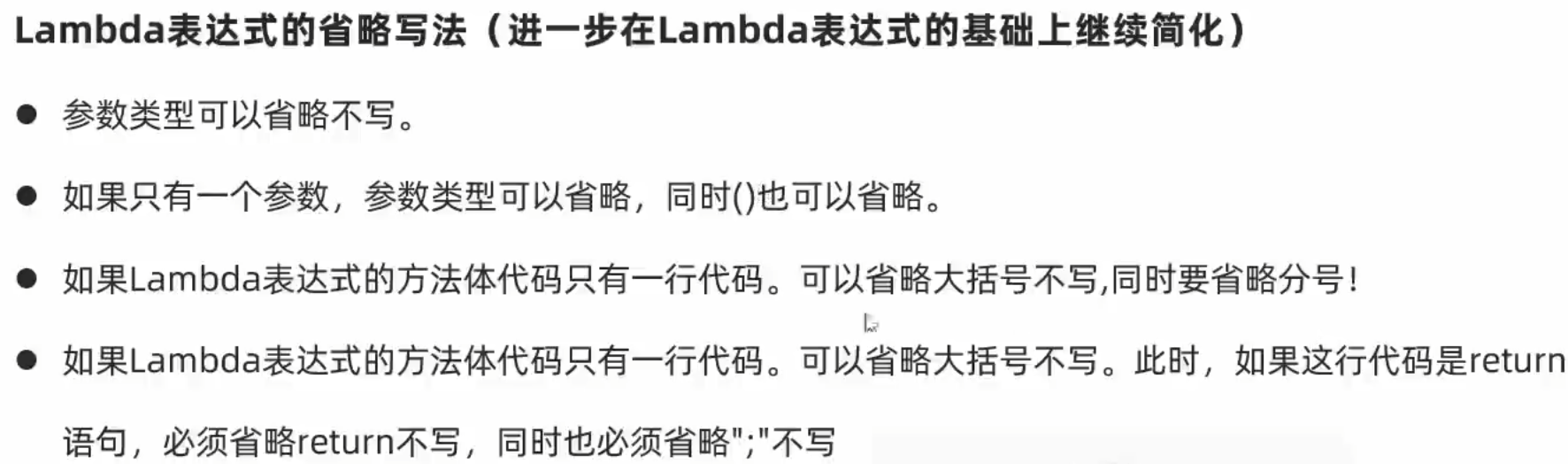
什么是函数式接▣？  
首先必须是接口、其次接口中有且仅有一个抽象方法的形式  
●通常我们会在接口上加上一个@Functionallnterface:注解，标记该接口必须是满足函数式接口。

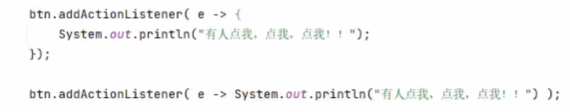
​​

简化的是new新对象的过程和重写方法的过程，直接调用内部的方法即可，

​​

#### 省略规则

​​

​​

‍