07 集合

本节我们将介绍Java的集合类型。集合类型也是Java标准库中被使用最多的类型。

Java集合简介

什么是集合（Collection）？集合就是“由若干个确定的元素所构成的整体”。例如，5只小兔构成的集合：

```
┌ - - - - - - - - - - - - - - - - - - - - - - - - - ┐

│   (\_(\     (\_/)     (\_/)     (\_/)      (\(\   │
    ( -.-)    (•.•)     (>.<)     (^.^)     (='.')
│  C(")_(")  (")_(")   (")_(")   (")_(")   O(_")")  │

└ - - - - - - - - - - - - - - - - - - - - - - - - - ┘
```

在数学中，我们经常遇到集合的概念。例如：

- 有限集合：
    - 一个班所有的同学构成的集合；
    - 一个网站所有的商品构成的集合；
    - ...
- 无限集合：
    - 全体自然数集合：1，2，3，……
    - 有理数集合；
    - 实数集合；
    - ...

为什么要在计算机中引入集合呢？这是为了便于处理一组类似的数据，例如：

- 计算所有同学的总成绩和平均成绩；
- 列举所有的商品名称和价格；
- ……

在Java中，如果一个Java对象可以在内部持有若干其他Java对象，并对外提供访问接口，我们把这种Java对象称为集合。很显然，Java的数组可以看作是一种集合：

```
String[] ss = new String[10]; // 可以持有10个String对象
ss[0] = "Hello"; // 可以放入String对象
String first = ss[0]; // 可以获取String对象
```

既然Java提供了数组这种数据类型，可以充当集合，那么，我们为什么还需要其他集合类？这是因为数组有如下限制：

- 数组初始化后大小不可变；
- 数组只能按索引顺序存取。

因此，我们需要各种不同类型的集合类来处理不同的数据，例如：

- 可变大小的顺序链表；
- 保证无重复元素的集合；
- ...

Collection

Java标准库自带的`java.util`包提供了集合类：`Collection`，它是除`Map`外所有其他集合类的根接口。Java的`java.util`包主要提供了以下三种类型的集合：

- `List`：一种有序列表的集合，例如，按索引排列的`Student`的`List`；
- `Set`：一种保证没有重复元素的集合，例如，所有无重复名称的`Student`的`Set`；
- `Map`：一种通过键值（key-value）查找的映射表集合，例如，根据`Student`的`name`查找对应`Student`的`Map`。

Java集合的设计有几个特点：一是实现了接口和实现类相分离，例如，有序表的接口是`List`，具体的实现类有`ArrayList`，`LinkedList`等，二是支持泛型，我们可以限制在一个集合中只能放入同一种数据类型的元素，例如：

```
List<String> list = new ArrayList<>(); // 只能放入String类型
```

最后，Java访问集合总是通过统一的方式——迭代器（Iterator）来实现，它最明显的好处在于无需知道集合内部元素是按什么方式存储的。

由于Java的集合设计非常久远，中间经历过大规模改进，我们要注意到有一小部分集合类是遗留类，不应该继续使用：

- `Hashtable`：一种线程安全的`Map`实现；
- `Vector`：一种线程安全的`List`实现；
- `Stack`：基于`Vector`实现的`LIFO`的栈。

还有一小部分接口是遗留接口，也不应该继续使用：

- `Enumeration`：已被`Iterator`取代。

小结

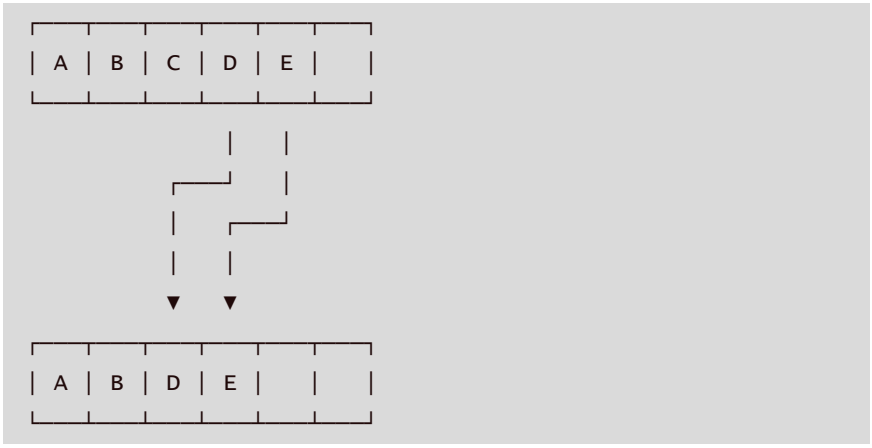
- Java的集合类定义在`java.util`包中，支持泛型，主要提供了3种集合类，包括`List`，`Set`和`Map`。Java集合使用统一的`Iterator`遍历，尽量不要使用遗留接口。

使用List

在集合类中，`List`是最基础的一种集合：它是一种有序链表。

`List`的行为和数组几乎完全相同：`List`内部按照放入元素的先后顺序存放，每个元素都可以通过索引确定自己的位置，`List`的索引和数组一样，从`0`开始。

数组和`List`类似，也是有序结构，如果我们使用数组，在添加和删除元素的时候，会非常不方便。例如，从一个已有的数组`{'A', 'B', 'C', 'D', 'E'}`中删除索引为`2`的元素：



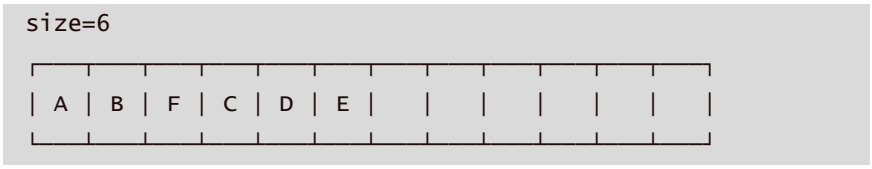
这个“删除”操作实际上是把`'C'`后面的元素依次往前挪一个位置，而“添加”操作实际上是把指定位置以后的元素都依次向后挪一个位置，腾出来的位置给新加的元素。这两种操作，用数组实现非常麻烦。

因此，在实际应用中，需要增删元素的有序列表，我们使用最多的是`ArrayList`。实际上，`ArrayList`在内部使用了数组来存储所有元素。例如，一个ArrayList拥有5个元素，实际数组大小为`6`（即有一个空位）：

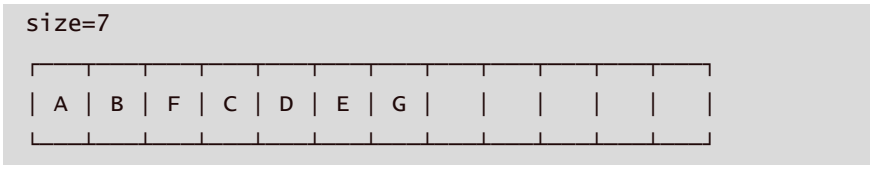
当添加一个元素并指定索引到`ArrayList`时，`ArrayList`自动移动需要移动的元素：

然后，往内部指定索引的数组位置添加一个元素，然后把`size`加`1`：

继续添加元素，但是数组已满，没有空闲位置的时候，`ArrayList`先创建一个更大的新数组，然后把旧数组的所有元素复制到新数组，紧接着用新数组取代旧数组：



现在，新数组就有了空位，可以继续添加一个元素到数组末尾，同时`size`加`1`：

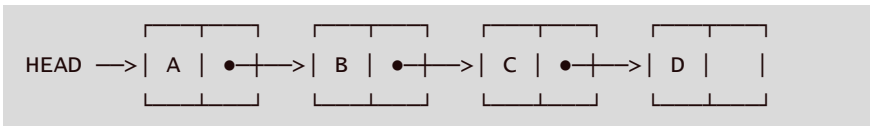


可见，`ArrayList`把添加和删除的操作封装起来，让我们操作`List`类似于操作数组，却不用关心内部元素如何移动。

我们考察`List`接口，可以看到几个主要的接口方法：

- 在末尾添加一个元素：`void add(E e)`
- 在指定索引添加一个元素：`void add(int index, E e)`
- 删除指定索引的元素：`int remove(int index)`
- 删除某个元素：`int remove(Object e)`
- 获取指定索引的元素：`E get(int index)`
- 获取链表大小（包含元素的个数）：`int size()`

但是，实现`List`接口并非只能通过数组（即`ArrayList`的实现方式）来实现，另一种`LinkedList`通过“链表”也实现了List接口。在`LinkedList`中，它的内部每个元素都指向下一个元素：



我们来比较一下`ArrayList`和`LinkedList`：

	ARRAYLIST	LINKEDLIST
获取指定元素	速度很快	需要从头开始查找元素
添加元素到末尾	速度很快	速度很快
在指定位置添加/删除	需要移动元素	不需要移动元素
内存占用	少	较大

通常情况下，我们总是优先使用`ArrayList`。

List的特点

使用`List`时，我们要关注`List`接口的规范。`List`接口允许我们添加重复的元素，即`List`内部的元素可以重复：

```
import java.util.ArrayList;
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("apple"); // size=1
        list.add("pear"); // size=2
        list.add("apple"); // 允许重复添加元素，size=3
        System.out.println(list.size());
    }
}
```

`List`还允许添加`null`：

```
import java.util.ArrayList;
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("apple"); // size=1
        list.add(null); // size=2
        list.add("pear"); // size=3
        String second = list.get(1); // null
        System.out.println(second);
    }
}
```

创建List

除了使用`ArrayList`和`LinkedList`，我们还可以通过`List`接口提供的`of()`方法，根据给定元素快速创建`List`：

```
List<Integer> list = List.of(1, 2, 5);
```

但是`List.of()`方法不接受`null`值，如果传入`null`，会抛出`NullPointerException`异常。

遍历List

和数组类型，我们要遍历一个List，完全可以用for循环根据索引配合`get(int)`方法遍历：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear",
"banana");
        for (int i=0; i<list.size(); i++) {
            String s = list.get(i);
            System.out.println(s);
        }
    }
}
```

但这种方式并不推荐，一是代码复杂，二是因为`get(int)`方法只有`ArrayList`的实现是高效的，换成`LinkedList`后，索引越大，访问速度越慢。

所以我们要始终坚持使用迭代器`Iterator`来访问`List`。`Iterator`本身也是一个对象，但它是由`List`的实例调用`iterator()`方法的时候创建的。`Iterator`对象知道如何遍历一个`List`，并且不同的`List`类型，返回的`Iterator`对象实现也是不同的，但总是具有最高的访问效率。

`Iterator`对象有两个方法：`boolean hasNext()`判断是否有下一个元素，`E next()`返回下一个元素。因此，使用`Iterator`遍历`List`代码如下：

```
import java.util.Iterator;
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear",
"banana");
        for (Iterator<String> it = list.iterator();
it.hasNext(); ) {
            String s = it.next();
            System.out.println(s);
        }
    }
}
```

有童鞋可能觉得使用`Iterator`访问`List`的代码比使用索引更复杂。但是，要记住，通过`Iterator`遍历`List`永远是最高效的方式。并且，由于`Iterator`遍历是如此常用，所以，Java的`for each`循环本身就可以帮我们使用`Iterator`遍历。把上面的代码再改写如下：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear",
"banana");
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

上述代码就是我们编写遍历`List`的常见代码。

实际上，只要实现了`Iterable`接口的集合类都可以直接用`for each`循环来遍历，Java编译器本身并不知道如何遍历集合对象，但它会自动把`for each`循环变成`Iterator`的调用，原因就在于`Iterable`接口定义了一个`Iterator iterator()`方法，强迫集合类必须返回一个`Iterator`实例。

List和Array转换

把`List`变为`Array`有三种方法，第一种是调用`toArray()`方法直接返回一个`Object[]`数组：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear",
"banana");
        Object[] array = list.toArray();
        for (Object s : array) {
            System.out.println(s);
        }
    }
}
```

这种方法会丢失类型信息，所以实际应用很少。

第二种方式是给`toArray(T[])`传入一个类型相同的`Array`，`List`内部自动把元素复制到传入的`Array`中：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<Integer> list = List.of(12, 34, 56);
        Integer[] array = list.toArray(new Integer[3]);
        for (Integer n : array) {
            System.out.println(n);
        }
    }
}
```

注意到这个`toArray(T[])`方法的泛型参数并不是`List`接口定义的泛型参数，所以，我们实际上可以传入其他类型的数组，例如我们传入`Number`类型的数组，返回的仍然是`Number`类型：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<Integer> list = List.of(12, 34, 56);
        Number[] array = list.toArray(new Number[3]);
        for (Number n : array) {
            System.out.println(n);
        }
    }
}
```

但是，如果我们传入类型不匹配的数组，例如，`String[]`类型的数组，由于`List`的元素是`Integer`，所以无法放入`String`数组，这个方法会抛出`ArrayStoreException`。

如果我们传入的数组大小和`List`实际的元素个数不一致怎么办？根据List接口的文档，我们可以知道：

如果传入的数组不够大，那么`List`内部会创建一个新的刚好够大的数组，填充后返回；如果传入的数组比`List`元素还要多，那么填充完元素后，剩下的数组元素一律填充`null`。

实际上，最常用的是传入一个“恰好”大小的数组：

```
Integer[] array = list.toArray(new Integer[list.size()]);
```

最后一种更简洁的写法是通过`List`接口定义的`T[] toArray(IntFunction generator)`方法：

```
Integer[] array = list.toArray(Integer[]::new);
```

这种函数式写法我们会在后续讲到。

反过来，把`Array`变为`List`就简单多了，通过`List.of(T...)`方法最简单：

```
Integer[] array = { 1, 2, 3 };
List<Integer> list = List.of(array);
```

对于JDK 11之前的版本，可以使用`Arrays.asList(T...)`方法把数组转换成`List`。

要注意的是，返回的`List`不一定就是`ArrayList`或者`LinkedList`，因为`List`只是一个接口，如果我们调用`List.of()`，它返回的是一个只读`List`：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<Integer> list = List.of(12, 34, 56);
        list.add(999); // UnsupportedOperationException
    }
}
```

对只读`List`调用`add()`、`remove()`方法会抛出`UnsupportedOperationException`。

练习

给定一组连续的整数，例如：10，11，12，.....，20，但其中缺失一个数字，试找出缺失的数字：

```
import java.util.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // 构造从start到end的序列：
        final int start = 10;
        final int end = 20;
        List<Integer> list = new ArrayList<>();
        for (int i = start; i <= end; i++) {
            list.add(i);
        }
        // 随机删除List中的一个元素：
        int removed = list.remove((int) (Math.random() *
list.size()));
        int found = findMissingNumber(start, end, list);
        System.out.println(list.toString());
        System.out.println("missing number: " + found);
        System.out.println(removed == found ? "测试成功" :
"测试失败");
    }
    static int findMissingNumber(int start, int end,
List<Integer> list) {
        return 0;
    }
}
```

增强版：和上述题目一样，但整数不再有序，试找出缺失的数字：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // 构造从start到end的序列：
```

```
        final int start = 10;
        final int end = 20;
        List<Integer> list = new ArrayList<>();
        for (int i = start; i <= end; i++) {
            list.add(i);
        }
        // 洗牌算法suffle可以随机交换List中的元素位置:
        Collections.shuffle(list);
        // 随机删除List中的一个元素:
        int removed = list.remove((int) (Math.random() *
list.size()));
        int found = findMissingNumber(start, end, list);
        System.out.println(list.toString());
        System.out.println("missing number: " + found);
        System.out.println(removed == found ? "测试成功" :
"测试失败");
    }
    static int findMissingNumber(int start, int end,
List<Integer> list) {
        return 0;
    }
}
```

下载练习：找出缺失的数字 （推荐使用IDE练习插件快速下载）

小结

- `List`是按索引顺序访问的长度可变的有序表，优先使用`ArrayList`而不是`LinkedList`；
- 可以直接使用`for each`遍历`List`；
- `List`可以和`Array`相互转换。

编写equals方法

我们知道`List`是一种有序链表：`List`内部按照放入元素的先后顺序存放，并且每个元素都可以通过索引确定自己的位置。

`List`还提供了`boolean contains(Object o)`方法来判断`List`是否包含某个指定元素。此外，`int indexOf(Object o)`方法可以返回某个元素的索引，如果元素不存在，就返回`-1`。

我们来看一个例子：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("A", "B", "C");
        System.out.println(list.contains("C")); // true
        System.out.println(list.contains("X")); // false
        System.out.println(list.indexOf("C")); // 2
        System.out.println(list.indexOf("X")); // -1
    }
}
```

这里我们注意一个问题，我们往`List`中添加的`"C"`和调用`contains("C")`传入的`"C"`是不是同一个实例？

如果这两个`"C"`不是同一个实例，这段代码是否还能得到正确的结果？我们可以改写一下代码测试一下：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("A", "B", "C");
        System.out.println(list.contains(new
String("C"))); // true or false?
        System.out.println(list.indexOf(new String("C")));
// 2 or -1?
    }
}
```

因为我们传入的是`new String("C")`，所以一定是不同的实例。结果仍然符合预期，这是为什么呢？

因为`List`内部并不是通过`==`判断两个元素是否相等，而是使用`equals()`方法判断两个元素是否相等，例如`contains()`方法可以实现如下：

```
public class ArrayList {
    Object[] elementData;
    public boolean contains(Object o) {
        for (int i = 0; i < size; i++) {
            if (o.equals(elementData[i])) {
                return true;
            }
        }
        return false;
    }
}
```

因此，要正确使用`List`的`contains()`、`indexOf()`这些方法，放入的实例必须正确覆写`equals()`方法，否则，放进去的实例，查找不到。我们之所以能正常放入`String`、`Integer`这些对象，是因为Java标准库定义的这些类已经正确实现了`equals()`方法。

我们以`Person`对象为例，测试一下：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<Person> list = List.of(
            new Person("Xiao Ming"),
            new Person("Xiao Hong"),
            new Person("Bob")
        );
        System.out.println(list.contains(new
Person("Bob"))); // false
    }
}

class Person {
    String name;
    public Person(String name) {
        this.name = name;
    }
}
```

不出意外，虽然放入了`new Person("Bob")`，但是用另一个`new Person("Bob")`查询不到，原因就是`Person`类没有覆写`equals()`方法。

编写equals

如何正确编写`equals()`方法？`equals()`方法要求我们必须满足以下条件：

- 自反性（Reflexive）：对于非`null`的`x`来说，`x.equals(x)`必须返回`true`；
- 对称性（Symmetric）：对于非`null`的`x`和`y`来说，如果`x.equals(y)`为`true`，则`y.equals(x)`也必须为`true`；
- 传递性（Transitive）：对于非`null`的`x`、`y`和`z`来说，如果`x.equals(y)`为`true`，`y.equals(z)`也为`true`，那么`x.equals(z)`也必须为`true`；
- 一致性（Consistent）：对于非`null`的`x`和`y`来说，只要`x`和`y`状态不变，则`x.equals(y)`总是一致地返回`true`或者`false`；
- 对`null`的比较：即`x.equals(null)`永远返回`false`。

上述规则看上去似乎非常复杂，但其实代码实现`equals()`方法是很简单的，我们以`Person`类为例：

```
public class Person {
    public String name;
    public int age;
}
```

首先，我们要定义“相等”的逻辑含义。对于`Person`类，如果`name`相等，并且`age`相等，我们就认为两个`Person`实例相等。

因此，编写`equals()`方法如下：

```
public boolean equals(Object o) {
    if (o instanceof Person) {
        Person p = (Person) o;
        return this.name.equals(p.name) && this.age ==
p.age;
    }
    return false;
}
```

对于引用字段比较，我们使用`equals()`，对于基本类型字段的比较，我们使用`==`。

如果`this.name`为`null`，那么`equals()`方法会报错，因此，需要继续改写如下：

```
public boolean equals(Object o) {
    if (o instanceof Person) {
        Person p = (Person) o;
        boolean nameEquals = false;
        if (this.name == null && p.name == null) {
            nameEquals = true;
        }
        if (this.name != null) {
            nameEquals = this.name.equals(p.name);
        }
        return nameEquals && this.age == p.age;
    }
    return false;
}
```

如果`Person`有好几个引用类型的字段，上面的写法就太复杂了。要简化引用类型的比较，我们使用`Objects.equals()`静态方法：

```
public boolean equals(Object o) {
    if (o instanceof Person) {
        Person p = (Person) o;
        return Objects.equals(this.name, p.name) &&
this.age == p.age;
    }
    return false;
}
```

因此，我们总结一下`equals()`方法的正确编写方法：

1. 先确定实例“相等”的逻辑，即哪些字段相等，就认为实例相等；
2. 用`instanceof`判断传入的待比较的`Object`是不是当前类型，如果是，继续比较，否则，返回`false`；
3. 对引用类型用`Objects.equals()`比较，对基本类型直接用`==`比较。

使用`Objects.equals()`比较两个引用类型是否相等的目的是省去了判断`null`的麻烦。两个引用类型都是`null`时它们也是相等的。

如果不调用`List`的`contains()`、`indexOf()`这些方法，那么放入的元素就不需要实现`equals()`方法。

练习

给Person类增加equals方法，使得调用indexOf()方法返回正常：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<Person> list = List.of(
            new Person("Xiao", "Ming", 18),
            new Person("Xiao", "Hong", 25),
            new Person("Bob", "Smith", 20)
        );
        boolean exist = list.contains(new Person("Bob",
"Smith", 20));
        System.out.println(exist ? "测试成功!" : "测试失
败!");
    }
}

class Person {
    String firstName;
    String lastName;
    int age;
    public Person(String firstName, String lastName, int
age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
}
```

下载练习：覆写equals方法 （推荐使用IDE练习插件快速下载）

小结

- 在`List`中查找元素时，`List`的实现类通过元素的`equals()`方法比较两个元素是否相等，因此，放入的元素必须正确覆写`equals()`方法，Java标准库提供的`String`、`Integer`等已经覆写了`equals()`方法；
- 编写`equals()`方法可借助`Objects.equals()`判断。
- 如果不在`List`中查找元素，就不必覆写`equals()`方法。

使用Map

我们知道，`List`是一种顺序列表，如果有一个存储学生`Student`实例的`List`，要在`List`中根据`name`查找某个指定的`Student`的分数，应该怎么办？

最简单的方法是遍历`List`并判断`name`是否相等，然后返回指定元素：

```
List<Student> list = ...
Student target = null;
for (Student s : list) {
    if ("Xiao Ming".equals(s.name)) {
        target = s;
        break;
    }
}
System.out.println(target.score);
```

这种需求其实非常常见，即通过一个键去查询对应的值。使用`List`来实现存在效率非常低的问题，因为平均需要扫描一半的元素才能确定，而`Map`这种键值（key-value）映射表的数据结构，作用就是能高效通过`key`快速查找`value`（元素）。

用`Map`来实现根据`name`查询某个`Student`的代码如下：

```
import java.util.HashMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {
        Student s = new Student("Xiao Ming", 99);
        Map<String, Student> map = new HashMap<>();
        map.put("Xiao Ming", s); // 将"Xiao Ming"和Student实例映射并关联
        Student target = map.get("Xiao Ming"); // 通过key查找并返回映射的Student实例
        System.out.println(target == s); // true，同一个实例
        System.out.println(target.score); // 99
        Student another = map.get("Bob"); // 通过另一个key查找
        System.out.println(another); // 未找到返回null
    }
}

class Student {
    public String name;
    public int score;
    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
}
```

通过上述代码可知：`Map`是一种键-值映射表，当我们调用`put(K key, V value)`方法时，就把`key`和`value`做了映射并放入`Map`。当我们调用`V get(K key)`时，就可以通过`key`获取到对应的`value`。如果`key`不存在，则返回`null`。和`List`类似，`Map`也是一个接口，最常用的实现类是`HashMap`。

如果只是想查询某个`key`是否存在，可以调用`boolean containsKey(K key)`方法。

如果我们在存储`Map`映射关系的时候，对同一个key调用两次`put()`方法，分别放入不同的`value`，会有什么问题呢？例如：

```
import java.util.HashMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("apple", 123);
        map.put("pear", 456);
        System.out.println(map.get("apple")); // 123
        map.put("apple", 789); // 再次放入apple作为key，但
value变为789
        System.out.println(map.get("apple")); // 789
    }
}
```

重复放入`key-value`并不会有任何问题，但是一个`key`只能关联一个`value`。在上面的代码中，一开始我们把`key`对象`"apple"`映射到`Integer`对象`123`，然后再次调用`put()`方法把`"apple"`映射到`789`，这时，原来关联的`value`对象`123`就被“冲掉”了。实际上，`put()`方法的签名是`V put(K key, V value)`，如果放入的`key`已经存在，`put()`方法会返回被删除的旧的`value`，否则，返回`null`。

始终牢记：Map中不存在重复的key，因为放入相同的key，只会把原有的key-value对应的value给替换掉。

此外，在一个`Map`中，虽然`key`不能重复，但`value`是可以重复的：

```
Map<String, Integer> map = new HashMap<>();
map.put("apple", 123);
map.put("pear", 123); // ok
```

遍历Map

对`Map`来说，要遍历`key`可以使用`for each`循环遍历`Map`实例的`keySet()`方法返回的`Set`集合，它包含不重复的`key`的集合：

```
import java.util.HashMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
```

```
        map.put("apple", 123);
        map.put("pear", 456);
        map.put("banana", 789);
        for (String key : map.keySet()) {
            Integer value = map.get(key);
            System.out.println(key + " = " + value);
        }
    }
}
```

同时遍历`key`和`value`可以使用`for each`循环遍历`Map`对象的`entrySet()`集合，它包含每一个`key-value`映射：

```
import java.util.HashMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("apple", 123);
        map.put("pear", 456);
        map.put("banana", 789);
        for (Map.Entry<String, Integer> entry :
map.entrySet()) {
            String key = entry.getKey();
            Integer value = entry.getValue();
            System.out.println(key + " = " + value);
        }
    }
}
```

`Map`和`List`不同的是，`Map`存储的是`key-value`的映射关系，并且，它*不保证顺序*。在遍历的时候，遍历的顺序既不一定是`put()`时放入的`key`的顺序，也不一定是`key`的排序顺序。使用`Map`时，任何依赖顺序的逻辑都是不可靠的。以`HashMap`为例，假设我们放入`"A"`，`"B"`，`"C"`这3个`key`，遍历的时候，每个`key`会保证被遍历一次且仅遍历一次，但顺序完全没有保证，甚至对于不同的JDK版本，相同的代码遍历的输出顺序都是不同的！

遍历Map时，不可假设输出的key是有序的！

练习

请编写一个根据`name`查找`score`的程序，并利用`Map`充当缓存，以提高查找效率：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<Student> list = List.of(
            new Student("Bob", 78),
            new Student("Alice", 85),
```

```
            new Student("Brush", 66),
            new Student("Newton", 99));
        var holder = new Students(list);
        System.out.println(holder.getScore("Bob") == 78 ?
"测试成功!" : "测试失败!");
        System.out.println(holder.getScore("Alice") == 85
? "测试成功!" : "测试失败!");
        System.out.println(holder.getScore("Tom") == -1 ?
"测试成功!" : "测试失败!");
    }
}

class Students {
    List<Student> list;
    Map<String, Integer> cache;

    Students(List<Student> list) {
        this.list = list;
        cache = new HashMap<>();
    }

    /**
     * 根据name查找score，找到返回score，未找到返回-1
     */
    int getScore(String name) {
        // 先在Map中查找:
        Integer score = this.cache.get(name);
        if (score == null) {
            // TODO:
        }
        return score == null ? -1 : score.intValue();
    }

    Integer findInList(String name) {
        for (var ss : this.list) {
            if (ss.name.equals(name)) {
                return ss.score;
            }
        }
        return null;
    }
}

class Student {
    String name;
    int score;

    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
}
```

下载练习：find-student-score （推荐使用IDE练习插件快速下载）

小结

- `Map`是一种映射表，可以通过`key`快速查找`value`。
- 可以通过`for each`遍历`keySet()`，也可以通过`for each`遍历`entrySet()`，直接获取`key-value`。
- 最常用的一种`Map`实现是`HashMap`。

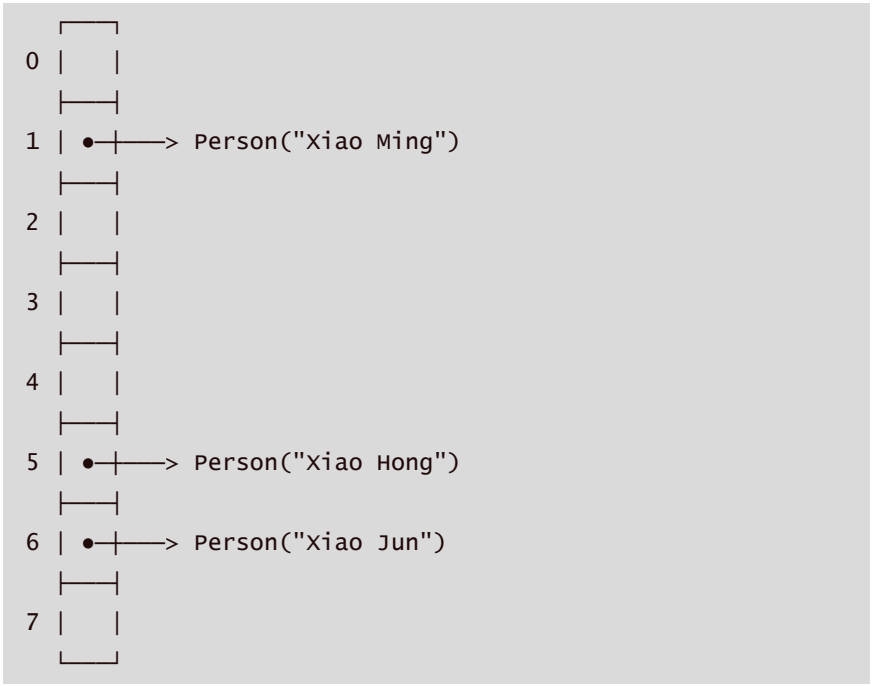
编写equals和hashCode

我们知道Map是一种键-值（key-value）映射表，可以通过key快速查找对应的value。

以HashMap为例，观察下面的代码：

```
Map<String, Person> map = new HashMap<>();
map.put("a", new Person("Xiao Ming"));
map.put("b", new Person("Xiao Hong"));
map.put("c", new Person("Xiao Jun"));

map.get("a"); // Person("Xiao Ming")
map.get("x"); // null
```

`HashMap`之所以能根据`key`直接拿到`value`，原因是它内部通过空间换时间的方法，用一个大数组存储所有`value`，并根据key直接计算出`value`应该存储在哪个索引：



如果`key`的值为`"a"`，计算得到的索引总是`1`，因此返回`value`为`Person("Xiao Ming")`，如果`key`的值为`"b"`，计算得到的索引总是`5`，因此返回`value`为`Person("Xiao Hong")`，这样，就不必遍历整个数组，即可直接读取`key`对应的`value`。

当我们使用key存取value的时候，就会引出一个问题：

我们放入Map的key是字符串"a"，但是，当我们获取Map的value时，传入的变量不一定就是放入的那个key对象。

换句话讲，两个key应该是内容相同，但不一定是同一个对象。测试代码如下：

```
import java.util.HashMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {
        String key1 = "a";
        Map<String, Integer> map = new HashMap<>();
        map.put(key1, 123);

        String key2 = new String("a");
        map.get(key2); // 123

        System.out.println(key1 == key2); // false
        System.out.println(key1.equals(key2)); // true
    }
}
```

因为在Map的内部，对key做比较是通过equals()实现的，这一点和List查找元素需要正确覆写equals()是一样的，即正确使用Map必须保证：作为key的对象必须正确覆写equals()方法。

我们经常使用String作为key，因为String已经正确覆写了equals()方法。但如果我们放入的key是一个自己写的类，就必须保证正确覆写了equals()方法。

我们再思考一下HashMap为什么能通过key直接计算出value存储的索引。相同的key对象（使用equals()判断时返回true）必须要计算出相同的索引，否则，相同的key每次取出的value就不一定对。

通过key计算索引的方式就是调用key对象的hashCode()方法，它返回一个int整数。HashMap正是通过这个方法直接定位key对应的value的索引，继而直接返回value。

因此，正确使用Map必须保证：

1. 作为key的对象必须正确覆写equals()方法，相等的两个key实例调用equals()必须返回true；
2. 作为key的对象还必须正确覆写hashCode()方法，且hashCode()方法要严格遵循以下规范：

- 如果两个对象相等，则两个对象的hashCode()必须相等；
- 如果两个对象不相等，则两个对象的hashCode()尽量不要相等。

即对应两个实例a和b：

- 如果`a`和`b`相等，那么`a.equals(b)`一定为`true`，则`a.hashCode()`必须等于`b.hashCode()`；
- 如果`a`和`b`不相等，那么`a.equals(b)`一定为`false`，则`a.hashCode()`和`b.hashCode()`尽量不要相等。

上述第一条规范是正确性，必须保证实现，否则`HashMap`不能正常工作。

而第二条如果尽量满足，则可以保证查询效率，因为不同的对象，如果返回相同的`hashCode()`，会造成`Map`内部存储冲突，使存取的效率下降。

正确编写`equals()`的方法我们已经在编写equals方法一节中讲过了，以`Person`类为例：

```
public class Person {
    String firstName;
    String lastName;
    int age;
}
```

把需要比较的字段找出来：

- firstName
- lastName
- age

然后，引用类型使用`Objects.equals()`比较，基本类型使用`==`比较。

在正确实现`equals()`的基础上，我们还需要正确实现`hashCode()`，即上述3个字段分别相同的实例，`hashCode()`返回的`int`必须相同：

```
public class Person {
    String firstName;
    String lastName;
    int age;

    @Override
    int hashCode() {
        int h = 0;
        h = 31 * h + firstName.hashCode();
        h = 31 * h + lastName.hashCode();
        h = 31 * h + age;
        return h;
    }
}
```

注意到`String`类已经正确实现了`hashCode()`方法，我们在计算`Person`的`hashCode()`时，反复使用`31*h`，这样做的目的是为了尽量把不同的`Person`实例的`hashCode()`均匀分布到整个`int`范围。

和实现`equals()`方法遇到的问题类似，如果`firstName`或`lastName`为`null`，上述代码工作起来就会抛`NullPointerException`。为了解决这个问题，我们在计算`hashCode()`的时候，经常借助`Objects.hash()`来计算：

```
int hashCode() {
    return Objects.hash(firstName, lastName, age);
}
```

所以，编写`equals()`和`hashCode()`遵循的原则是：

`equals()`用到的用于比较的每一个字段，都必须在`hashCode()`中用于计算；`equals()`中没有使用到的字段，绝不可放在`hashCode()`中计算。

另外注意，对于放入`HashMap`的`value`对象，没有任何要求。

延伸阅读

既然`HashMap`内部使用了数组，通过计算`key`的`hashCode()`直接定位`value`所在的索引，那么第一个问题来了：hashCode()返回的`int`范围高达±21亿，先不考虑负数，`HashMap`内部使用的数组得有多大？

实际上`HashMap`初始化时默认的数组大小只有16，任何`key`，无论它的`hashCode()`有多大，都可以简单地通过：

```
int index = key.hashCode() & 0xf; // 0xf = 15
```

把索引确定在0～15，即永远不会超出数组范围，上述算法只是一种最简单的实现。

第二个问题：如果添加超过16个`key-value`到`HashMap`，数组不够用了怎么办？

添加超过一定数量的`key-value`时，`HashMap`会在内部自动扩容，每次扩容一倍，即长度为16的数组扩展为长度32，相应地，需要重新确定`hashCode()`计算的索引位置。例如，对长度为32的数组计算`hashCode()`对应的索引，计算方式要改为：

```
int index = key.hashCode() & 0x1f; // 0x1f = 31
```

由于扩容会导致重新分布已有的`key-value`，所以，频繁扩容对`HashMap`的性能影响很大。如果我们确定要使用一个容量为`10000`个`key-value`的`HashMap`，更好的方式是创建`HashMap`时就指定容量：

```
Map<String, Integer> map = new HashMap<>(10000);
```

虽然指定容量是`10000`，但`HashMap`内部的数组长度总是2n，因此，实际数组长度被初始化为比`10000`大的`16384`（214）。

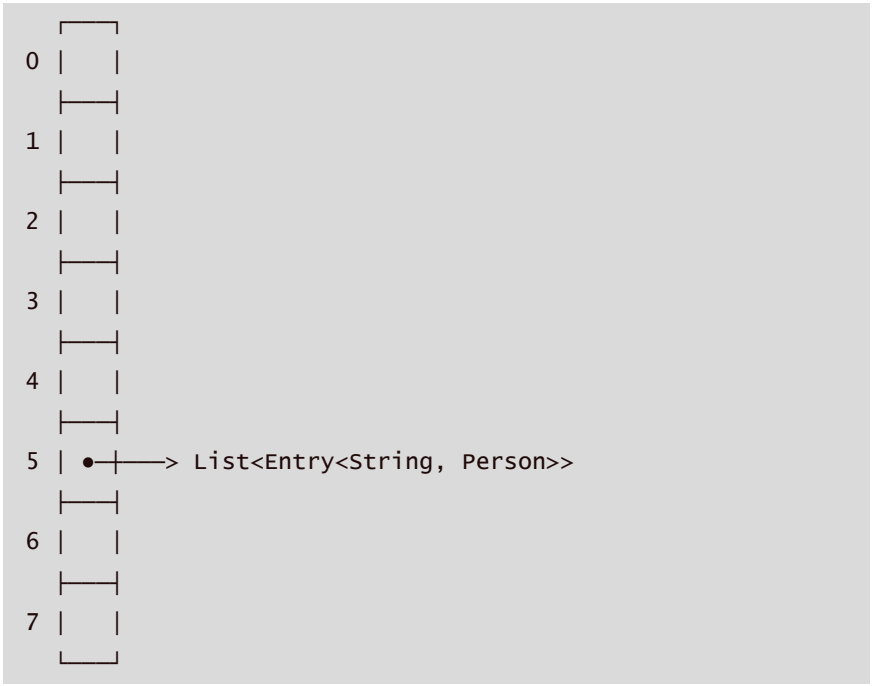
最后一个问题：如果不同的两个`key`，例如`"a"`和`"b"`，它们的`hashCode()`恰好是相同的（这种情况是完全可能的，因为不相等的两个实例，只要求`hashCode()`尽量不相等），那么，当我们放入：

```
map.put("a", new Person("Xiao Ming"));
map.put("b", new Person("Xiao Hong"));
```

时，由于计算出的数组索引相同，后面放入的`"Xiao Hong"`会不会把`"Xiao Ming"`覆盖了？

当然不会！使用`Map`的时候，只要`key`不相同，它们映射的`value`就互不干扰。但是，在`HashMap`内部，确实可能存在不同的`key`，映射到相同的`hashCode()`，即相同的数组索引上，肿么办？

我们就假设`"a"`和`"b"`这两个`key`最终计算出的索引都是5，那么，在`HashMap`的数组中，实际存储的不是一个`Person`实例，而是一个`List`，它包含两个`Entry`，一个是`"a"`的映射，一个是`"b"`的映射：



在查找的时候，例如：

```
Person p = map.get("a");
```

HashMap内部通过`"a"`找到的实际上是`List>`，它还需要遍历这个`List`，并找到一个`Entry`，它的`key`字段是`"a"`，才能返回对应的`Person`实例。

我们把不同的`key`具有相同的`hashCode()`的情况称之为哈希冲突。在冲突的时候，一种最简单的解决办法是用`List`存储`hashCode()`相同的`key-value`。显然，如果冲突的概率越大，这个`List`就越长，`Map`的`get()`方法效率就越低，这就是为什么要尽量满足条件二：

如果两个对象不相等，则两个对象的hashCode()尽量不要相等。

`hashCode()`方法编写得越好，`HashMap`工作的效率就越高。

小结

- 要正确使用`HashMap`，作为`key`的类必须正确覆写`equals()`和`hashCode()`方法；
- 一个类如果覆写了`equals()`，就必须覆写`hashCode()`，并且覆写规则是：
    - 如果`equals()`返回`true`，则`hashCode()`返回值必须相等；
    - 如果`equals()`返回`false`，则`hashCode()`返回值尽量不要相等。
- 实现`hashCode()`方法可以通过`Objects.hashCode()`辅助方法实现。

使用EnumMap

因为`HashMap`是一种通过对key计算`hashCode()`，通过空间换时间的方式，直接定位到value所在的内部数组的索引，因此，查找效率非常高。

如果作为key的对象是`enum`类型，那么，还可以使用Java集合库提供的一种`EnumMap`，它在内部以一个非常紧凑的数组存储value，并且根据`enum`类型的key直接定位到内部数组的索引，并不需要计算`hashCode()`，不但效率最高，而且没有额外的空间浪费。

我们以`DayOfWeek`这个枚举类型为例，为它做一个“翻译”功能：

```
import java.time.DayOfWeek;
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Map<DayOfWeek, String> map = new EnumMap<>
(DayOfWeek.class);
        map.put(DayOfWeek.MONDAY, "星期一");
        map.put(DayOfWeek.TUESDAY, "星期二");
        map.put(DayOfWeek.WEDNESDAY, "星期三");
        map.put(DayOfWeek.THURSDAY, "星期四");
        map.put(DayOfWeek.FRIDAY, "星期五");
        map.put(DayOfWeek.SATURDAY, "星期六");
        map.put(DayOfWeek.SUNDAY, "星期日");
        System.out.println(map);
        System.out.println(map.get(DayOfWeek.MONDAY));
    }
}
```

使用`EnumMap`的时候，我们总是用`Map`接口来引用它，因此，实际上把`HashMap`和`EnumMap`互换，在客户端看来没有任何区别。

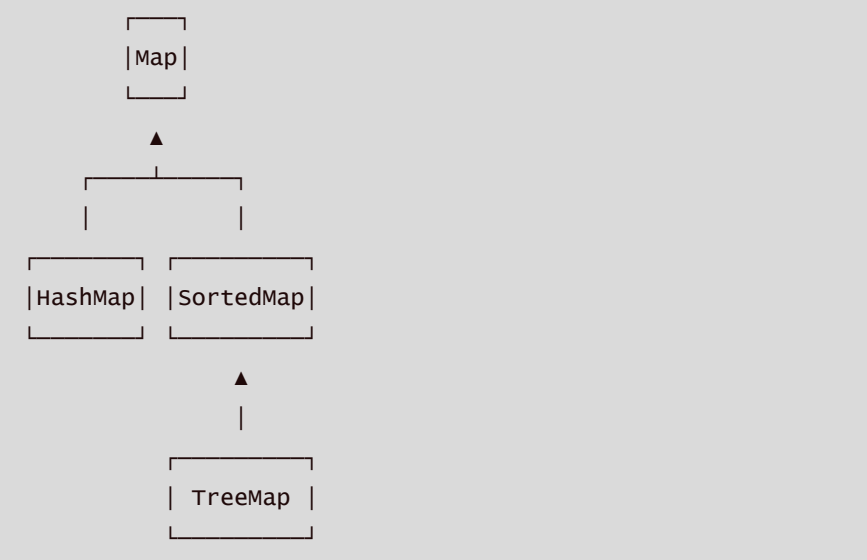
小结

- 如果`Map`的key是`enum`类型，推荐使用`EnumMap`，既保证速度，也不浪费空间。
- 使用`EnumMap`的时候，根据面向抽象编程的原则，应持有`Map`接口。

使用TreeMap

我们已经知道，`HashMap`是一种以空间换时间的映射表，它的实现原理决定了内部的Key是无序的，即遍历`HashMap`的Key时，其顺序是不可预测的（但每个Key都会遍历一次且仅遍历一次）。

还有一种`Map`，它在内部会对Key进行排序，这种`Map`就是`SortedMap`。注意到`SortedMap`是接口，它的实现类是`TreeMap`。



`SortedMap`保证遍历时以Key的顺序来进行排序。例如，放入的Key是`"apple"`、`"pear"`、`"orange"`，遍历的顺序一定是`"apple"`、`"orange"`、`"pear"`，因为`String`默认按字母排序：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new TreeMap<>();
        map.put("orange", 1);
        map.put("apple", 2);
        map.put("pear", 3);
        for (String key : map.keySet()) {
            System.out.println(key);
        }
        // apple, orange, pear
    }
}
```

使用`TreeMap`时，放入的Key必须实现`Comparable`接口。`String`、`Integer`这些类已经实现了`Comparable`接口，因此可以直接作为Key使用。作为Value的对象则没有任何要求。

如果作为Key的class没有实现`Comparable`接口，那么，必须在创建`TreeMap`时同时指定一个自定义排序算法：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Map<Person, Integer> map = new TreeMap<>(new
Comparator<Person>() {
            public int compare(Person p1, Person p2) {
                return p1.name.compareTo(p2.name);
            }
        });
        map.put(new Person("Tom"), 1);
        map.put(new Person("Bob"), 2);
        map.put(new Person("Lily"), 3);
        for (Person key : map.keySet()) {
            System.out.println(key);
        }
        // {Person: Bob}, {Person: Lily}, {Person: Tom}
        System.out.println(map.get(new Person("Bob"))); //
2
    }
}

class Person {
    public String name;
    Person(String name) {
        this.name = name;
    }
    public String toString() {
        return "{Person: " + name + "}";
    }
}
```

注意到`Comparator`接口要求实现一个比较方法，它负责比较传入的两个元素`a`和`b`，如果`a`，则返回负数，通常是`-1`，如果`a==b`，则返回`0`，如果`a>b`，则返回正数，通常是`1`。TreeMap`内部根据比较结果对Key进行排序。

从上述代码执行结果可知，打印的Key确实是按照`Comparator`定义的顺序排序的。如果要根据Key查找Value，我们可以传入一个`new Person("Bob")`作为Key，它会返回对应的`Integer`值`2`。

另外，注意到`Person`类并未覆写`equals()`和`hashCode()`，因为`TreeMap`不使用`equals()`和`hashCode()`。

我们来看一个稍微复杂的例子：这次我们定义了`Student`类，并用分数`score`进行排序，高分在前：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
```

```
        Map<Student, Integer> map = new TreeMap<>(new
Comparator<Student>() {
            public int compare(Student p1, Student p2) {
                return p1.score > p2.score ? -1 : 1;
            }
        });
        map.put(new Student("Tom", 77), 1);
        map.put(new Student("Bob", 66), 2);
        map.put(new Student("Lily", 99), 3);
        for (Student key : map.keySet()) {
            System.out.println(key);
        }
        System.out.println(map.get(new Student("Bob",
66))); // null?
    }
}

class Student {
    public String name;
    public int score;
    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public String toString() {
        return String.format("{%s: score=%d}", name,
score);
    }
}
```

在`for`循环中，我们确实得到了正确的顺序。但是，且慢！根据相同的Key：`new Student("Bob", 66)`进行查找时，结果为`null`！

这是怎么肥四？难道`TreeMap`有问题？遇到`TreeMap`工作不正常时，我们首先回顾Java编程基本规则：出现问题，不要怀疑Java标准库，要从自身代码找原因。

在这个例子中，`TreeMap`出现问题，原因其实出在这个`Comparator`上：

```
public int compare(Student p1, Student p2) {
    return p1.score > p2.score ? -1 : 1;
}
```

在`p1.score`和`p2.score`不相等的时候，它的返回值是正确的，但是，在`p1.score`和`p2.score`相等的时候，它并没有返回`0`！这就是为什么`TreeMap`工作不正常的原因：`TreeMap`在比较两个Key是否相等时，依赖Key的`compareTo()`方法或者`Comparator.compare()`方法。在两个Key相等时，必须返回`0`。因此，修改代码如下：

```
public int compare(Student p1, Student p2) {
    if (p1.score == p2.score) {
        return 0;
    }
    return p1.score > p2.score ? -1 : 1;
}
```

或者直接借助`Integer.compare(int, int)`也可以返回正确的比较结果。

小结

- `SortedMap`在遍历时严格按照Key的顺序遍历，最常用的实现类是`TreeMap`；
- 作为`SortedMap`的Key必须实现`Comparable`接口，或者传入`Comparator`；
- 要严格按照`compare()`规范实现比较逻辑，否则，`TreeMap`将不能正常工作。

使用Properties

在编写应用程序的时候，经常需要读写配置文件。例如，用户的设置：

```
# 上次最后打开的文件:
last_open_file=/data/hello.txt
# 自动保存文件的时间间隔:
auto_save_interval=60
```

配置文件的特点是，它的Key-Value一般都是`String`-`String`类型的，因此我们完全可以用`Map`来表示它。

因为配置文件非常常用，所以Java集合库提供了一个`Properties`来表示一组“配置”。由于历史遗留原因，`Properties`内部本质上是一个`Hashtable`，但我们只需要用到`Properties`自身关于读写配置的接口。

读取配置文件

用`Properties`读取配置文件非常简单。Java默认配置文件以`.properties`为扩展名，每行以`key=value`表示，以`#`课开头的是注释。以下是一个典型的配置文件：

```
# setting.properties

last_open_file=/data/hello.txt
auto_save_interval=60
```

可以从文件系统读取这个`.properties`文件：

```
String f = "setting.properties";
Properties props = new Properties();
props.load(new java.io.FileInputStream(f));

String filepath = props.getProperty("last_open_file");
String interval = props.getProperty("auto_save_interval",
"120");
```

可见，用`Properties`读取配置文件，一共有三步：

1. 创建`Properties`实例；
2. 调用`load()`读取文件；
3. 调用`getProperty()`获取配置。

调用`getProperty()`获取配置时，如果key不存在，将返回`null`。我们还可以提供一个默认值，这样，当key不存在的时候，就返回默认值。

也可以从classpath读取`.properties`文件，因为`load(InputStream)`方法接收一个`InputStream`实例，表示一个字节流，它不一定是文件流，也可以是从jar包中读取的资源流：

```
Properties props = new Properties();
props.load(getClass().getResourceAsStream("/common/setting
.properties"));
```

试试从内存读取一个字节流：

```
// properties
import java.io.*;
import java.util.Properties;

public class Main {
    public static void main(String[] args) throws
IOException {
        String settings = "# test" + "\n" + "course=Java"
+ "\n" + "last_open_date=2019-08-07T12:35:01";
        ByteArrayInputStream input = new
ByteArrayInputStream(settings.getBytes("UTF-8"));
        Properties props = new Properties();
        props.load(input);

        System.out.println("course: " +
props.getProperty("course"));
        System.out.println("last_open_date: " +
props.getProperty("last_open_date"));
        System.out.println("last_open_file: " +
props.getProperty("last_open_file"));
        System.out.println("auto_save: " +
props.getProperty("auto_save", "60"));
    }
}
```

如果有多个`.properties`文件，可以反复调用`load()`读取，后读取的key-value会覆盖已读取的key-value：

```
Properties props = new Properties();
props.load(getClass().getResourceAsString("/common/setting
.properties"));
props.load(new
FileInputStream("C:\\conf\\setting.properties"));
```

上面的代码演示了`Properties`的一个常用用法：可以把默认配置文件放到classpath中，然后，根据机器的环境编写另一个配置文件，覆盖某些默认的配置。

`Properties`设计的目的是存储`String`类型的key－value，但`Properties`实际上是从`Hashtable`派生的，它的设计实际上是有问题的，但是为了保持兼容性，现在已经没法修改了。除了`getProperty()`和`setProperty()`方法外，还有从`Hashtable`继承下来的`get()`和`put()`方法，这些方法的参数签名是`Object`，我们在使用`Properties`的时候，不要去调用这些从`Hashtable`继承下来的方法。

写入配置文件

如果通过`setProperty()`修改了`Properties`实例，可以把配置写入文件，以便下次启动时获得最新配置。写入配置文件使用`store()`方法：

```
Properties props = new Properties();
props.setProperty("url", "http://www.liaoxuefeng.com");
props.setProperty("language", "Java");
props.store(new
FileOutputStream("C:\\conf\\setting.properties"), "这是写入
的properties注释");
```

编码

早期版本的Java规定`.properties`文件编码是ASCII编码（ISO8859-1），如果涉及到中文就必须用`name=\u4e2d\u6587`来表示，非常别扭。从JDK9开始，Java的`.properties`文件可以使用UTF-8编码了。

不过，需要注意的是，由于`load(InputStream)`默认总是以ASCII编码读取字节流，所以会导致读到乱码。我们需要用另一个重载方法`load(Reader)`读取：

```
Properties props = new Properties();
props.load(new FileReader("settings.properties",
StandardCharsets.UTF_8));
```

就可以正常读取中文。`InputStream`和`Reader`的区别是一个是字节流，一个是字符流。字符流在内存中已经以`char`类型表示了，不涉及编码问题。

小结

- Java集合库提供的`Properties`用于读写配置文件`.properties`。`.properties`文件可以使用UTF-8编码。
- 可以从文件系统、classpath或其他任何地方读取`.properties`文件。
- 读写`Properties`时，注意仅使用`getProperty()`和`setProperty()`方法，不要调用继承而来的`get()`和`put()`等方法。

使用Set

我们知道，`Map`用于存储key-value的映射，对于充当key的对象，是不能重复的，并且，不但需要正确覆写`equals()`方法，还要正确覆写`hashCode()`方法。

如果我们只需要存储不重复的key，并不需要存储映射的value，那么就可以使用`Set`。

`Set`用于存储不重复的元素集合，它主要提供以下几个方法：

- 将元素添加进`Set`：`boolean add(E e)`
- 将元素从`Set`删除：`boolean remove(Object e)`
- 判断是否包含元素：`boolean contains(Object e)`

我们来看几个简单的例子：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        System.out.println(set.add("abc")); // true
        System.out.println(set.add("xyz")); // true
        System.out.println(set.add("xyz")); // false，添加失败，因为元素已存在
        System.out.println(set.contains("xyz")); // true，元素存在
        System.out.println(set.contains("XYZ")); // false，元素不存在
        System.out.println(set.remove("hello")); // false，删除失败，因为元素不存在
        System.out.println(set.size()); // 2，一共两个元素
    }
}
```

`Set`实际上相当于只存储key、不存储value的`Map`。我们经常用`Set`用于去除重复元素。

因为放入`Set`的元素和`Map`的key类似，都要正确实现`equals()`和`hashCode()`方法，否则该元素无法正确地放入`Set`。

最常用的`Set`实现类是`HashSet`，实际上，`HashSet`仅仅是对`HashMap`的一个简单封装，它的核心代码如下：

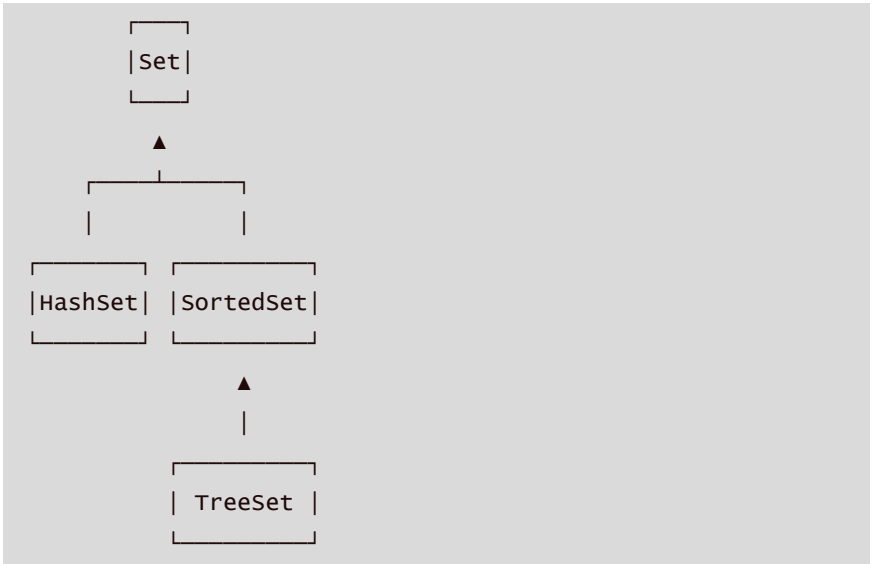
```
public class HashSet<E> implements Set<E> {
```

```
    // 持有一个HashMap:
    private HashMap<E, Object> map = new HashMap<>();

    // 放入HashMap的value:
    private static final Object PRESENT = new Object();

    public boolean add(E e) {
        return map.put(e, PRESENT) == null;
    }

    public boolean contains(Object o) {
        return map.containsKey(o);
    }

    public boolean remove(Object o) {
        return map.remove(o) == PRESENT;
    }
}
```

`Set`接口并不保证有序，而`SortedSet`接口则保证元素是有序的：

- `HashSet`是无序的，因为它实现了`Set`接口，并没有实现`SortedSet`接口；
- `TreeSet`是有序的，因为它实现了`SortedSet`接口。

用一张图表示：



我们来看`HashSet`的输出：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("apple");
        set.add("banana");
        set.add("pear");
        set.add("orange");
        for (String s : set) {
            System.out.println(s);
        }
    }
}
```

注意输出的顺序既不是添加的顺序，也不是`String`排序的顺序，在不同版本的JDK中，这个顺序也可能是不同的。

把`HashSet`换成`TreeSet`，在遍历`TreeSet`时，输出就是有序的，这个顺序是元素的排序顺序：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Set<String> set = new TreeSet<>();
        set.add("apple");
        set.add("banana");
        set.add("pear");
        set.add("orange");
        for (String s : set) {
            System.out.println(s);
        }
    }
}
```

使用`TreeSet`和使用`TreeMap`的要求一样，添加的元素必须正确实现`Comparable`接口，如果没有实现`Comparable`接口，那么创建`TreeSet`时必须传入一个`Comparator`对象。

练习

在聊天软件中，发送方发送消息时，遇到网络超时后就会自动重发，因此，接收方可能会收到重复的消息，在显示给用户看的时候，需要首先去重。请练习使用`Set`去除重复的消息：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<Message> received = List.of(
            new Message(1, "Hello!"),
            new Message(2, "发工资了吗？"),
```

```
            new Message(2, "发工资了吗？"),
            new Message(3, "去哪吃饭？"),
            new Message(3, "去哪吃饭？"),
            new Message(4, "Bye")
        );
        List<Message> displayMessages = process(received);
        for (Message message : displayMessages) {
            System.out.println(message.text);
        }
    }

    static List<Message> process(List<Message> received) {
        // TODO: 按sequence去除重复消息
        return received;
    }
}

class Message {
    public final int sequence;
    public final String text;
    public Message(int sequence, String text) {
        this.sequence = sequence;
        this.text = text;
    }
}
```

小结

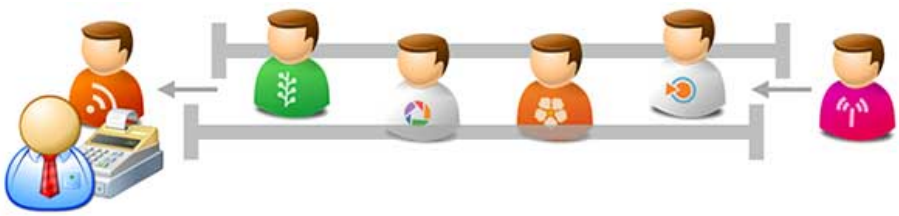
- `Set`用于存储不重复的元素集合：
    - 放入`HashSet`的元素与作为`HashMap`的key要求相同；
    - 放入`TreeSet`的元素与作为`TreeMap`的Key要求相同；
- 利用`Set`可以去除重复元素；
- 遍历`SortedSet`按照元素的排序顺序遍历，也可以自定义排序算法。

使用Queue

队列（`Queue`）是一种经常使用的集合。`Queue`实际上是实现了一个先进先出（FIFO：First In First Out）的有序表。它和`List`的区别在于，`List`可以在任意位置添加和删除元素，而`Queue`只有两个操作：

- 把元素添加到队列末尾；
- 从队列头部取出元素。

超市的收银台就是一个队列：



在Java的标准库中，队列接口`Queue`定义了以下几个方法：

- `int size()`：获取队列长度；
- `boolean add(E)`/`boolean offer(E)`：添加元素到队尾；
- `E remove()`/`E poll()`：获取队首元素并从队列中删除；
- `E element()`/`E peek()`：获取队首元素但并不从队列中删除。

对于具体的实现类，有的Queue有最大队列长度限制，有的Queue没有。注意到添加、删除和获取队列元素总是有两个方法，这是因为在添加或获取元素失败时，这两个方法的行为是不同的。我们用一个表格总结如下：

	THROW EXCEPTION	返回FALSE或NULL
添加元素到队尾	add(E e)	boolean offer(E e)
取队首元素并删除	E remove()	E poll()
取队首元素但不删除	E element()	E peek()

举个栗子，假设我们有一个队列，对它做一个添加操作，如果调用`add()`方法，当添加失败时（可能超过了队列的容量），它会抛出异常：

```
Queue<String> q = ...
try {
    q.add("Apple");
    System.out.println("添加成功");
} catch(IllegalStateException e) {
    System.out.println("添加失败");
}
```

如果我们调用`offer()`方法来添加元素，当添加失败时，它不会抛异常，而是返回`false`：

```
Queue<String> q = ...
if (q.offer("Apple")) {
    System.out.println("添加成功");
} else {
    System.out.println("添加失败");
}
```

当我们需要从`Queue`中取出队首元素时，如果当前`Queue`是一个空队列，调用`remove()`方法，它会抛出异常：

```
Queue<String> q = ...
try {
    String s = q.remove();
    System.out.println("获取成功");
} catch(IllegalStateException e) {
    System.out.println("获取失败");
}
```

如果我们调用`poll()`方法来取出队首元素，当获取失败时，它不会抛异常，而是返回`null`：

```
Queue<String> q = ...
String s = q.poll();
if (s != null) {
    System.out.println("获取成功");
} else {
    System.out.println("获取失败");
}
```

因此，两套方法可以根据需要来选择使用。

注意：不要把`null`添加到队列中，否则`poll()`方法返回`null`时，很难确定是取到了`null`元素还是队列为空。

接下来我们以`poll()`和`peek()`为例来说说“获取并删除”与“获取但不删除”的区别。对于`Queue`来说，每次调用`poll()`，都会获取队首元素，并且获取到的元素已经从队列中被删除了：

```
import java.util.LinkedList;
import java.util.Queue;
public class Main {
    public static void main(String[] args) {
        Queue<String> q = new LinkedList<>();
        // 添加3个元素到队列:
        q.offer("apple");
        q.offer("pear");
        q.offer("banana");
        // 从队列取出元素:
        System.out.println(q.poll()); // apple
        System.out.println(q.poll()); // pear
        System.out.println(q.poll()); // banana
        System.out.println(q.poll()); // null,因为队列是空的
    }
}
```

如果用`peek()`，因为获取队首元素时，并不会从队列中删除这个元素，所以可以反复获取：

```
import java.util.LinkedList;
import java.util.Queue;
public class Main {
    public static void main(String[] args) {
        Queue<String> q = new LinkedList<>();
        // 添加3个元素到队列:
        q.offer("apple");
        q.offer("pear");
        q.offer("banana");
        // 队首永远都是apple，因为peek()不会删除它:
        System.out.println(q.peek()); // apple
```

```
        System.out.println(q.peek()); // apple
        System.out.println(q.peek()); // apple
    }
}
```

从上面的代码中，我们还可以发现，`LinkedList`即实现了`List`接口，又实现了`Queue`接口，但是，在使用的时候，如果我们把它当作List，就获取List的引用，如果我们把它当作Queue，就获取Queue的引用：

```
// 这是一个List:
List<String> list = new LinkedList<>();
// 这是一个Queue:
Queue<String> queue = new LinkedList<>();
```

始终按照面向抽象编程的原则编写代码，可以大大提高代码的质量。

小结

队列`Queue`实现了一个先进先出（FIFO）的数据结构：

- 通过`add()`/`offer()`方法将元素添加到队尾；
- 通过`remove()`/`poll()`从队首获取元素并删除；
- 通过`element()`/`peek()`从队首获取元素但不删除。

要避免把`null`添加到队列。

使用PriorityQueue

我们知道，`Queue`是一个先进先出（FIFO）的队列。

在银行柜台办业务时，我们假设只有一个柜台在办理业务，但是办理业务的人很多，怎么办？

可以每个人先取一个号，例如：`A1`、`A2`、`A3`.....然后，按照号码顺序依次办理，实际上这就是一个`Queue`。

如果这时来了一个VIP客户，他的号码是`V1`，虽然当前排队的是`A10`、`A11`、`A12`.....但是柜台下一个呼叫的客户号码却是`V1`。

这个时候，我们发现，要实现“VIP插队”的业务，用`Queue`就不行了，因为`Queue`会严格按FIFO的原则取出队首元素。我们需要的是优先队列：`PriorityQueue`。

`PriorityQueue`和`Queue`的区别在于，它的出队顺序与元素的优先级有关，对`PriorityQueue`调用`remove()`或`poll()`方法，返回的总是优先级最高的元素。

要使用`PriorityQueue`，我们就必须给每个元素定义“优先级”。我们以实际代码为例，先看看`PriorityQueue`的行为：

```
import java.util.PriorityQueue;
import java.util.Queue;
public class Main {
    public static void main(String[] args) {
        Queue<String> q = new PriorityQueue<>();
        // 添加3个元素到队列:
        q.offer("apple");
        q.offer("pear");
        q.offer("banana");
        System.out.println(q.poll()); // apple
        System.out.println(q.poll()); // banana
        System.out.println(q.poll()); // pear
        System.out.println(q.poll()); // null,因为队列为空
    }
}
```

我们放入的顺序是`"apple"`、`"pear"`、`"banana"`，但是取出的顺序却是`"apple"`、`"banana"`、`"pear"`，这是因为从字符串的排序看，`"apple"`排在最前面，`"pear"`排在最后面。

因此，放入`PriorityQueue`的元素，必须实现`Comparable`接口，`PriorityQueue`会根据元素的排序顺序决定出队的优先级。

如果我们要放入的元素并没有实现`Comparable`接口怎么办？`PriorityQueue`允许我们提供一个`Comparator`对象来判断两个元素的顺序。我们以银行排队业务为例，实现一个`PriorityQueue`：

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;
public class Main {
    public static void main(String[] args) {
        Queue<User> q = new PriorityQueue<>(new
UserComparator());
        // 添加3个元素到队列:
        q.offer(new User("Bob", "A1"));
        q.offer(new User("Alice", "A2"));
        q.offer(new User("Boss", "V1"));
        System.out.println(q.poll()); // Boss/V1
        System.out.println(q.poll()); // Bob/A1
        System.out.println(q.poll()); // Alice/A2
        System.out.println(q.poll()); // null,因为队列为空
    }
}

class UserComparator implements Comparator<User> {
    public int compare(User u1, User u2) {
        if (u1.number.charAt(0) == u2.number.charAt(0)) {
            // 如果两人的号都是A开头或者都是V开头,比较号的大小:
            return u1.number.compareTo(u2.number);
        }
```

```
            if (u1.number.charAt(0) == 'V') {
                // u1的号码是V开头,优先级高:
                return -1;
            } else {
                return 1;
            }
        }
    }

    class User {
        public final String name;
        public final String number;

        public User(String name, String number) {
            this.name = name;
            this.number = number;
        }

        public String toString() {
            return name + "/" + number;
        }
    }
```

实现`PriorityQueue`的关键在于提供的`UserComparator`对象，它负责比较两个元素的大小（较小的在前）。`UserComparator`总是把`V`开头的号码优先返回，只有在开头相同的时候，才比较号码大小。

上面的`UserComparator`的比较逻辑其实还是有问题的，它会把`A10`排在`A2`的前面，请尝试修复该错误。

小结

- `PriorityQueue`实现了一个优先队列：从队首获取元素时，总是获取优先级最高的元素。
- `PriorityQueue`默认按元素比较的顺序排序（必须实现`Comparable`接口），也可以通过`Comparator`自定义排序算法（元素就不必实现`Comparable`接口）。

使用Deque

我们知道，`Queue`是队列，只能一头进，另一头出。

如果把条件放松一下，允许两头都进，两头都出，这种队列叫双端队列（Double Ended Queue），学名`Deque`。

Java集合提供了接口`Deque`来实现一个双端队列，它的功能是：

- 既可以添加到队尾，也可以添加到队首；
- 既可以从队首获取，又可以从队尾获取。

我们来比较一下`Queue`和`Deque`出队和入队的方法：

	QUEUE	DEQUE
添加元素到队尾	add(E e) / offer(E e)	addLast(E e) / offerLast(E e)
取队首元素并删除	E remove() / E poll()	E removeFirst() / E pollFirst()
取队首元素但不删除	E element() / E peek()	E getFirst() / E peekFirst()
添加元素到队首	无	addFirst(E e) / offerFirst(E e)
取队尾元素并删除	无	E removeLast() / E pollLast()
取队尾元素但不删除	无	E getLast() / E peekLast()

对于添加元素到队尾的操作，`Queue`提供了`add()`/`offer()`方法，而`Deque`提供了`addLast()`/`offerLast()`方法。添加元素到对首、取队尾元素的操作在`Queue`中不存在，在`Deque`中由`addFirst()`/`removeLast()`等方法提供。

注意到`Deque`接口实际上扩展自`Queue`：

```
public interface Deque<E> extends Queue<E> {
    ...
}
```

因此，`Queue`提供的`add()`/`offer()`方法在`Deque`中也可以使用，但是，使用`Deque`，最好不要调用`offer()`，而是调用`offerLast()`：

```
import java.util.Deque;
import java.util.LinkedList;
public class Main {
    public static void main(String[] args) {
        Deque<String> deque = new LinkedList<>();
        deque.offerLast("A"); // A
        deque.offerLast("B"); // B -> A
        deque.offerFirst("C"); // B -> A -> C
        System.out.println(deque.pollFirst()); // C, 剩下B
-> A
        System.out.println(deque.pollLast()); // B
        System.out.println(deque.pollFirst()); // A
        System.out.println(deque.pollFirst()); // null
    }
}
```

如果直接写`deque.offer()`，我们就需要思考，`offer()`实际上是`offerLast()`，我们明确地写上`offerLast()`，不需要思考就能一眼看出这是添加到队尾。

因此，使用`Deque`，推荐总是明确调用`offerLast()`/`offerFirst()`或者`pollFirst()`/`pollLast()`方法。

`Deque`是一个接口，它的实现类有`ArrayDeque`和`LinkedList`。

我们发现`LinkedList`真是一个全能选手，它即是`List`，又是`Queue`，还是`Deque`。但是我们在使用的时候，总是用特定的接口来引用它，这是因为持有接口说明代码的抽象层次更高，而且接口本身定义的方法代表了特定的用途。

```
// 不推荐的写法:
LinkedList<String> d1 = new LinkedList<>();
d1.offerLast("z");
// 推荐的写法:
Deque<String> d2 = new LinkedList<>();
d2.offerLast("z");
```

可见面向抽象编程的一个原则就是：尽量持有接口，而不是具体的实现类。

小结

`Deque`实现了一个双端队列（Double Ended Queue），它可以：

- 将元素添加到队尾或队首：`addLast()`/`offerLast()`/`addFirst()`/`offerFirst()`；
- 从队首 / 队尾获取元素并删除：`removeFirst()`/`pollFirst()`/`removeLast()`/`pollLast()`；
- 从队首 / 队尾获取元素但不删除：`getFirst()`/`peekFirst()`/`getLast()`/`peekLast()`；
- 总是调用`xxxFirst()`/`xxxLast()`以便与`Queue`的方法区分开；
- 避免把`null`添加到队列。

使用Stack

栈（Stack）是一种后进先出（LIFO：Last In First Out）的数据结构。

什么是LIFO呢？我们先回顾一下`Queue`的特点FIFO：

```
        ┌─────────────────────────┐
  (\(\      (\(\    (\(\    (\(\      (\(\
 (='.') -> (='.')  (='.')  (='.') -> (='.')
o(_")")   o(_")") o(_")") o(_")")   o(_")")
        └─────────────────────────┘
```

所谓FIFO，是最先进队列的元素一定最早出队列，而LIFO是最后进`Stack`的元素一定最早出`Stack`。如何做到这一点呢？只需要把队列的一端封死：

```
        ┌─────────────────────────────┐
  (\(\      (\(\    (\(\    (\(\    (\(\ |
 (='.') <-> (='.')  (='.')  (='.')  (='.')|
o(_")")   o(_")") o(_")") o(_")") o(_")")|
        └─────────────────────────────┘
```

因此，`Stack`是这样一种数据结构：只能不断地往`Stack`中压入（push）元素，最后进去的必须最早弹出（pop）来：

`Stack`只有入栈和出栈的操作：

- 把元素压栈：`push(E)`；
- 把栈顶的元素“弹出”：`pop(E)`；
- 取栈顶元素但不弹出：`peek(E)`。

在Java中，我们用`Deque`可以实现`Stack`的功能：

- 把元素压栈：`push(E)`/`addFirst(E)`；
- 把栈顶的元素“弹出”：`pop(E)`/`removeFirst()`；
- 取栈顶元素但不弹出：`peek(E)`/`peekFirst()`。

为什么Java的集合类没有单独的`Stack`接口呢？因为有个遗留类名字就叫`Stack`，出于兼容性考虑，所以没办法创建`Stack`接口，只能用`Deque`接口来“模拟”一个`Stack`了。

当我们把`Deque`作为`Stack`使用时，注意只调用`push()`/`pop()`/`peek()`方法，不要调用`addFirst()`/`removeFirst()`/`peekFirst()`方法，这样代码更加清晰。

Stack的作用

Stack在计算机中使用非常广泛，JVM在处理Java方法调用的时候就会通过栈这种数据结构维护方法调用的层次。例如：

```
static void main(String[] args) {
    foo(123);
}

static String foo(x) {
    return "F-" + bar(x + 1);
}

static int bar(int x) {
    return x << 2;
}
```

JVM会创建方法调用栈，每调用一个方法时，先将参数压栈，然后执行对应的方法；当方法返回时，返回值压栈，调用方法通过出栈操作获得方法返回值。

因为方法调用栈有容量限制，嵌套调用过多会造成栈溢出，即引发`StackOverflowError`：

```
public class Main {
    public static void main(String[] args) {
        increase(1);
    }

    static int increase(int x) {
        return increase(x) + 1;
    }
}
```

我们再来看一个`Stack`的用途：对整数进行进制的转换就可以利用栈。

例如，我们要把一个`int`整数`12500`转换为十六进制表示的字符串，如何实现这个功能？

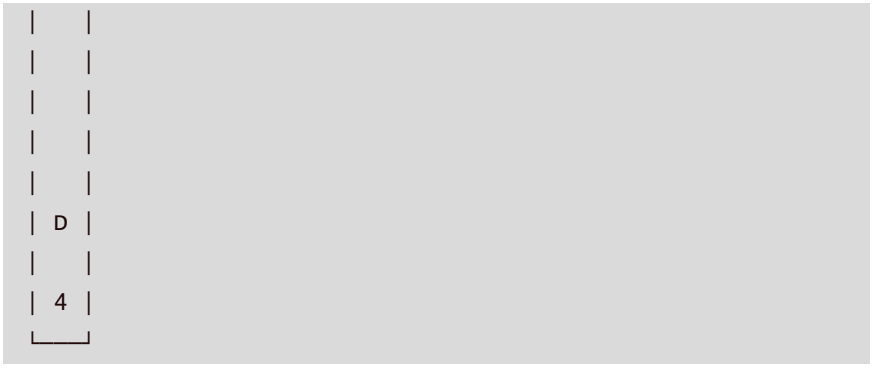
首先我们准备一个空栈：

```
|    |
|    |
|    |
|    |
|    |
|    |
|    |
|    |
└────┘
```

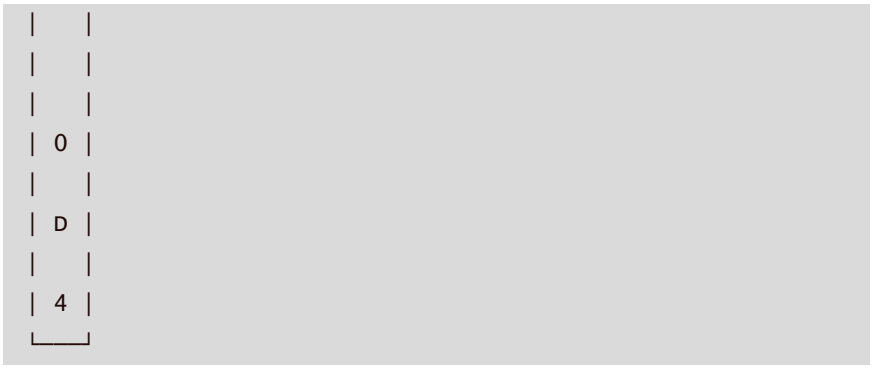
然后计算12500÷16=781...4，余数是`4`，把余数`4`压栈：

```
|    |
|    |
|    |
|    |
|    |
|    |
|    |
|    |
| 4  |
└────┘
```

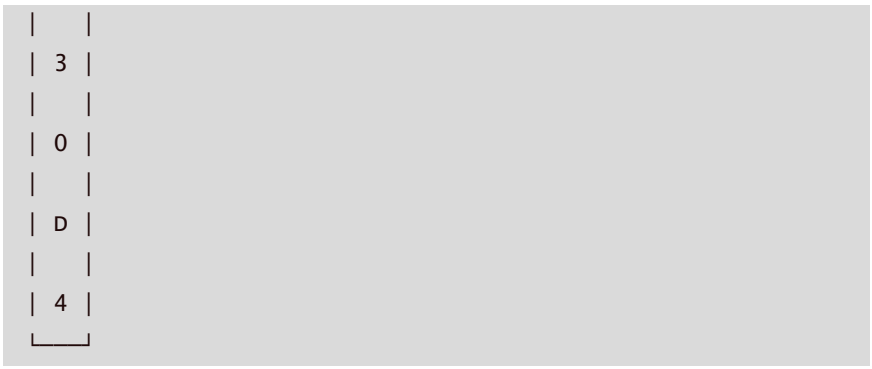
然后计算781÷16=48...13，余数是`13`，`13`的十六进制用字母`D`表示，把余数`D`压栈：



然后计算48÷16=3...0，余数是`0`，把余数`0`压栈：



最后计算3÷16=0...3，余数是`3`，把余数`3`压栈：



当商是`0`的时候，计算结束，我们把栈的所有元素依次弹出，组成字符串`30D4`，这就是十进制整数`12500`的十六进制表示的字符串。

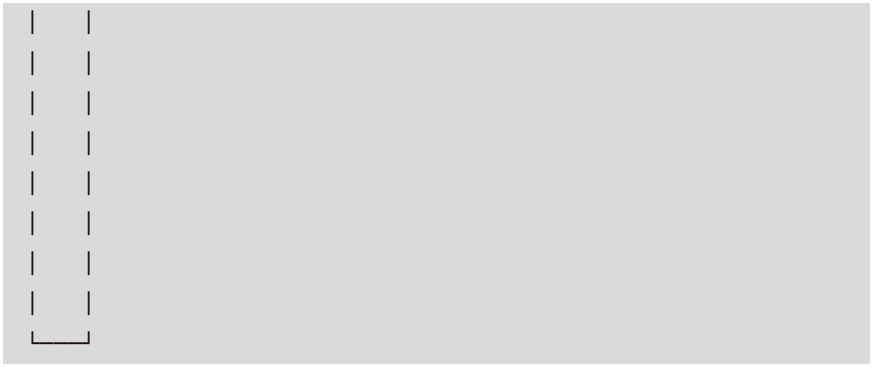
计算中缀表达式

在编写程序的时候，我们使用的带括号的数学表达式实际上是中缀表达式，即运算符在中间，例如：`1 + 2 * (9 - 5)`。

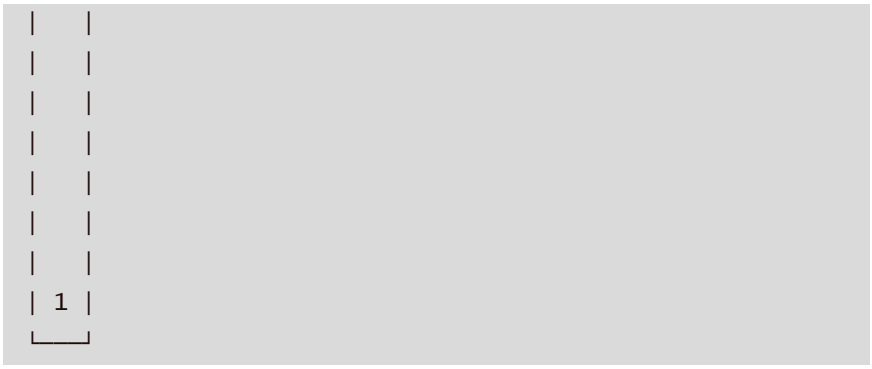
但是计算机执行表达式的时候，它并不能直接计算中缀表达式，而是通过编译器把中缀表达式转换为后缀表达式，例如：`1 2 9 5 - * +`。

这个编译过程就会用到栈。我们先跳过编译这一步（涉及运算优先级，代码比较复杂），看看如何通过栈计算后缀表达式。

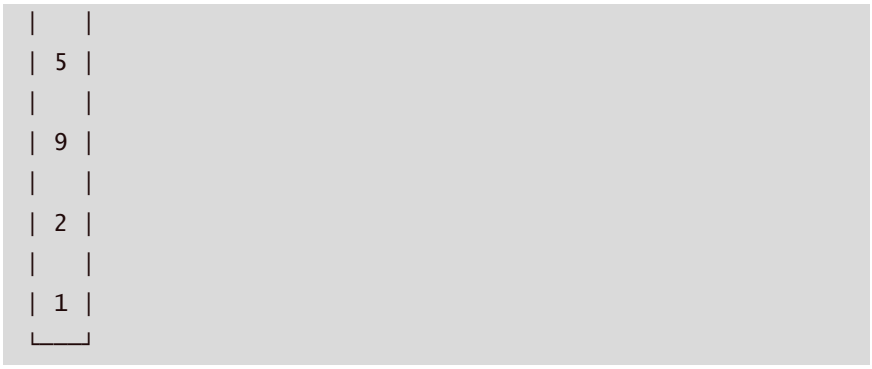
计算后缀表达式不考虑优先级，直接从左到右依次计算，因此计算起来简单。首先准备一个空的栈：



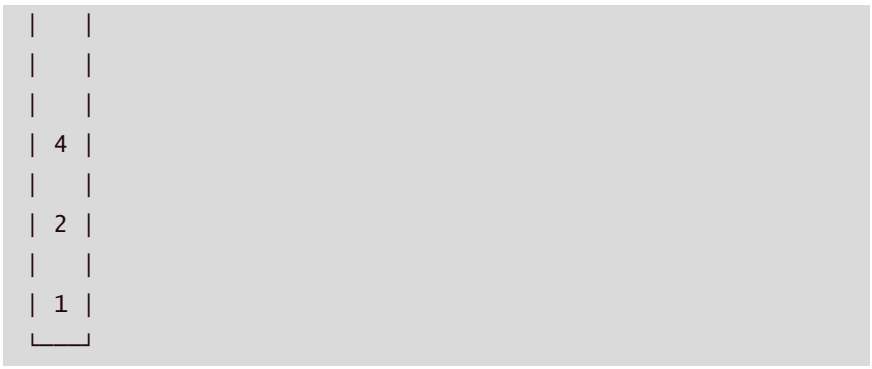
然后我们依次扫描后缀表达式`1 2 9 5 - * +`，遇到数字`1`，就直接扔到栈里：



紧接着，遇到数字`2`，`9`，`5`，也扔到栈里：



接下来遇到减号时，弹出栈顶的两个元素，并计算`9-5=4`，把结果`4`压栈：



接下来遇到`*`号时，弹出栈顶的两个元素，并计算`2*4=8`，把结果`8`压栈：

```
|   |
|   |
|   |
|   |
|   |
| 8 |
|   |
| 1 |
└───┘
```

接下来遇到`+`号时，弹出栈顶的两个元素，并计算`1+8=9`，把结果`9`压栈：

```
|   |
|   |
|   |
|   |
|   |
|   |
|   |
| 9 |
└───┘
```

扫描结束后，没有更多的计算了，弹出栈的唯一一个元素，得到计算结果`9`。

练习

请利用Stack把一个给定的整数转换为十六进制：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        String hex = toHex(12500);
        if (hex.equalsIgnoreCase("30D4")) {
            System.out.println("测试通过");
        } else {
            System.out.println("测试失败");
        }
    }

    static String toHex(int n) {
        return "";
    }
}
```

进阶练习：

请利用Stack把字符串中缀表达式编译为后缀表达式，然后再利用栈执行后缀表达式获得计算结果：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        String exp = "1 + 2 * (9 - 5)";
        SuffixExpression se = compile(exp);
        int result = se.execute();
        System.out.println(exp + " = " + result + " " +
(result == 1 + 2 * (9 - 5) ? "√" : "X"));
    }

    static SuffixExpression compile(String exp) {
        // TODO:
        return new SuffixExpression();
    }
}

class SuffixExpression {
    int execute() {
        // TODO:
        return 0;
    }
}
```

进阶练习2：

请把带变量的中缀表达式编译为后缀表达式，执行后缀表达式时，传入变量的值并获得计算结果：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        String exp = "x + 2 * (y - 5)";
        SuffixExpression se = compile(exp);
        Map<String, Integer> env = Map.of("x", 1, "y", 9);
        int result = se.execute(env);
        System.out.println(exp + " = " + result + " " +
(result == 1 + 2 * (9 - 5) ? "√" : "X"));
    }

    static SuffixExpression compile(String exp) {
        // TODO:
        return new SuffixExpression();
    }
}

class SuffixExpression {
```

```
    int execute(Map<String, Integer> env) {
        // TODO:
        return 0;
    }
}
```

下载练习：Stack练习 （推荐使用IDE练习插件快速下载）

小结

- 栈（Stack）是一种后进先出（LIFO）的数据结构，操作栈的元素的方法有：
    - 把元素压栈：`push(E)`；
    - 把栈顶的元素“弹出”：`pop(E)`；
    - 取栈顶元素但不弹出：`peek(E)`。
- 在Java中，我们用`Deque`可以实现`Stack`的功能，注意只调用`push()`/`pop()`/`peek()`方法，避免调用`Deque`的其他方法。
- 最后，不要使用遗留类`Stack`。

使用Iterator

Java的集合类都可以使用`for each`循环，`List`、`Set`和`Queue`会迭代每个元素，`Map`会迭代每个key。以`List`为例：

```
List<String> list = List.of("Apple", "Orange", "Pear");
for (String s : list) {
    System.out.println(s);
}
```

实际上，Java编译器并不知道如何遍历`List`。上述代码能够编译通过，只是因为编译器把`for each`循环通过`Iterator`改写为了普通的`for`循环：

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

我们把这种通过`Iterator`对象遍历集合的模式称为迭代器。

使用迭代器的好处在于，调用方总是以统一的方式遍历各种集合类型，而不必关系它们内部的存储结构。

例如，我们虽然知道`ArrayList`在内部是以数组形式存储元素，并且，它还提供了`get(int)`方法。虽然我们可以用`for`循环遍历：

```
for (int i=0; i<list.size(); i++) {
    Object value = list.get(i);
}
```

但是这样一来，调用方就必须知道集合的内部存储结构。并且，如果把`ArrayList`换成`LinkedList`，`get(int)`方法耗时会随着index的增加而增加。如果把`ArrayList`换成`Set`，上述代码就无法编译，因为`Set`内部没有索引。

用`Iterator`遍历就没有上述问题，因为`Iterator`对象是集合对象自己在内部创建的，它自己知道如何高效遍历内部的数据集合，调用方则获得了统一的代码，编译器才能把标准的`for each`循环自动转换为`Iterator`遍历。

如果我们自己编写了一个集合类，想要使用`for each`循环，只需满足以下条件：

- 集合类实现`Iterable`接口，该接口要求返回一个`Iterator`对象；
- 用`Iterator`对象迭代集合内部数据。

这里的关键在于，集合类通过调用`iterator()`方法，返回一个`Iterator`对象，这个对象必须自己知道如何遍历该集合。

一个简单的`Iterator`示例如下，它总是以倒序遍历集合：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        ReverseList<String> rlist = new ReverseList<>();
        rlist.add("Apple");
        rlist.add("Orange");
        rlist.add("Pear");
        for (String s : rlist) {
            System.out.println(s);
        }
    }
}

class ReverseList<T> implements Iterable<T> {

    private List<T> list = new ArrayList<>();

    public void add(T t) {
        list.add(t);
    }

    @Override
    public Iterator<T> iterator() {
        return new ReverseIterator(list.size());
    }

    class ReverseIterator implements Iterator<T> {
```

```
        int index;

        ReverseIterator(int index) {
            this.index = index;
        }

        @Override
        public boolean hasNext() {
            return index > 0;
        }

        @Override
        public T next() {
            index--;
            return ReverseList.this.list.get(index);
        }
    }
}
```

虽然`ReverseList`和`ReverseIterator`的实现类稍微比较复杂，但是，注意到这是底层集合库，只需编写一次。而调用方则完全按`for each`循环编写代码，根本不需要知道集合内部的存储逻辑和遍历逻辑。

在编写`Iterator`的时候，我们通常可以用一个内部类来实现`Iterator`接口，这个内部类可以直接访问对应的外部类的所有字段和方法。例如，上述代码中，内部类`ReverseIterator`可以用`ReverseList.this`获得当前外部类的`this`引用，然后，通过这个`this`引用就可以访问`ReverseList`的所有字段和方法。

小结

`Iterator`是一种抽象的数据访问模型。使用`Iterator`模式进行迭代的好处有：

- 对任何集合都采用同一种访问模型；
- 调用者对集合内部结构一无所知；
- 集合类返回的`Iterator`对象知道如何迭代。

Java提供了标准的迭代器模型，即集合类实现`java.util.Iterable`接口，返回`java.util.Iterator`实例。

使用Collections

`Collections`是JDK提供的工具类，同样位于`java.util`包中。它提供了一系列静态方法，能更方便地操作各种集合。

注意Collections结尾多了一个s，不是Collection！

我们一般看方法名和参数就可以确认`Collections`提供的该方法的功能。例如，对于以下静态方法：

```
public static boolean addAll(Collection<? super T> c, T...
elements) { ... }
```

`addAll()`方法可以给一个`Collection`类型的集合添加若干元素。因为方法签名是`Collection`，所以我们可以传入`List`，`Set`等各种集合类型。

创建空集合

`Collections`提供了一系列方法来创建空集合：

- 创建空List：`List emptyList()`
- 创建空Map：`Map emptyMap()`
- 创建空Set：`Set emptySet()`

要注意到返回的空集合是不可变集合，无法向其中添加或删除元素。

此外，也可以用各个集合接口提供的`of(T...)`方法创建空集合。例如，以下创建空`List`的两个方法是等价的：

```
List<String> list1 = List.of();
List<String> list2 = Collections.emptyList();
```

创建单元素集合

`Collections`提供了一系列方法来创建一个单元素集合：

- 创建一个元素的List：`List singletonList(T o)`
- 创建一个元素的Map：`Map singletonMap(K key, V value)`
- 创建一个元素的Set：`Set singleton(T o)`

要注意到返回的单元素集合也是不可变集合，无法向其中添加或删除元素。

此外，也可以用各个集合接口提供的`of(T...)`方法创建单元素集合。例如，以下创建单元素`List`的两个方法是等价的：

```
List<String> list1 = List.of("apple");
List<String> list2 = Collections.singleton("apple");
```

实际上，使用`List.of(T...)`更方便，因为它既可以创建空集合，也可以创建单元素集合，还可以创建任意个元素的集合：

```
List<String> list1 = List.of(); // empty list
List<String> list2 = List.of("apple"); // 1 element
List<String> list3 = List.of("apple", "pear"); // 2
elements
List<String> list4 = List.of("apple", "pear", "orange");
// 3 elements
```

排序

`Collections`可以对`List`进行排序。因为排序会直接修改`List`元素的位置，因此必须传入可变`List`：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("apple");
        list.add("pear");
        list.add("orange");
        // 排序前:
        System.out.println(list);
        Collections.sort(list);
        // 排序后:
        System.out.println(list);
    }
}
```

洗牌

`Collections`提供了洗牌算法，即传入一个有序的`List`，可以随机打乱`List`内部元素的顺序，效果相当于让计算机洗牌：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        for (int i=0; i<10; i++) {
            list.add(i);
        }
        // 洗牌前:
        System.out.println(list);
        Collections.shuffle(list);
        // 洗牌后:
        System.out.println(list);
    }
}
```

不可变集合

`Collections`还提供了一组方法把可变集合封装成不可变集合：

- 封装成不可变List：`List unmodifiableList(List list)`
- 封装成不可变Set：`Set unmodifiableSet(Set set)`
- 封装成不可变Map：`Map unmodifiableMap(Map m)`

这种封装实际上是通过创建一个代理对象，拦截掉所有修改方法实现的。我们来看看效果：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<String> mutable = new ArrayList<>();
        mutable.add("apple");
        mutable.add("pear");
        // 变为不可变集合:
        List<String> immutable =
Collections.unmodifiableList(mutable);
        immutable.add("orange"); //
UnsupportedOperationException!
    }
}
```

然而，继续对原始的可变`List`进行增删是可以的，并且，会直接影响到封装后的“不可变”`List`：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<String> mutable = new ArrayList<>();
        mutable.add("apple");
        mutable.add("pear");
        // 变为不可变集合:
        List<String> immutable =
Collections.unmodifiableList(mutable);
        mutable.add("orange");
        System.out.println(immutable);
    }
}
```

因此，如果我们希望把一个可变`List`封装成不可变`List`，那么，返回不可变`List`后，最好立刻扔掉可变`List`的引用，这样可以保证后续操作不会意外改变原始对象，从而造成“不可变”`List`变化了：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<String> mutable = new ArrayList<>();
        mutable.add("apple");
        mutable.add("pear");
        // 变为不可变集合:
        List<String> immutable =
Collections.unmodifiableList(mutable);
        // 立刻扔掉mutable的引用:
        mutable = null;
        System.out.println(immutable);
    }
}
```

线程安全集合

`Collections`还提供了一组方法，可以把线程不安全的集合变为线程安全的集合：

- 变为线程安全的List：`List synchronizedList(List list)`
- 变为线程安全的Set：`Set synchronizedSet(Set s)`
- 变为线程安全的Map：`Map synchronizedMap(Map m)`

多线程的概念我们会在后面讲。因为从Java 5开始，引入了更高效的并发集合类，所以上述这几个同步方法已经没有什么用了。

小结

`Collections`类提供了一组工具方法来方便使用集合类：

- 创建空集合；
- 创建单元素集合；
- 创建不可变集合；
- 排序 / 洗牌等操作。