

# 【Java8新特性】冰河带你看尽Java8新特性，你想要的都在这儿了！！

---

【Java8新特性】冰河带你看尽Java8新特性，你想要的都在这儿了！！

[写在前面](#)

[作者简介](#)

[如何持续提升自身能力](#)

[Java8有哪些新特性？](#)

[Java8有哪些优点？](#)

[什么是Lambda表达式？](#)

[匿名内部类](#)

[Lambda表达式](#)

[对比常规方法和Lambda表达式](#)

[1.常规遍历集合](#)

[2.使用设计模式优化代码](#)

[3.匿名内部类](#)

[4.重头戏：Lambda表达式](#)

[5.重头戏：Stream API](#)

[匿名类到Lambda表达式](#)

[Lambda表达式的语法](#)

[函数式接口](#)

[Lambda表达式典型案例](#)

[案例一](#)

[需求](#)

[实现](#)

[案例二](#)

[需求](#)

[实现](#)

[案例三](#)

[需求](#)

[实现](#)

[函数式接口总览](#)

[四大核心函数式接口](#)

[其他函数接口](#)

[四大核心函数式接口](#)

[Consumer接口](#)

[1.接口说明](#)

[2.使用示例](#)

[Supplier接口](#)

[1.接口说明](#)

[2.使用示例](#)

[Function接口](#)

[1.接口说明](#)

[2.使用示例](#)

[Predicate接口](#)

[1.接口说明](#)

[2.使用示例](#)

[Java8知识点总结](#)

[Java7与Java8中的HashMap](#)

[JDK8 HashMap重排序](#)

[筛选与切片](#)

[中间操作](#)

[终止操作](#)

归约

Optional 容器类

方法引用

构造器引用

数组引用

什么是Stream?

Stream操作的三个步骤

如何创建Stream?

- 1.获取Stream
- 2.由数组创建Stream
- 3.由值创建流
- 4.由函数创建流

Stream的中间操作

- 1.筛选与切片
- 2.映射
- 3.排序

Stream 的终止操作

- 1.查找与匹配
- 2.规约
- 3.收集

并行流与串行流

Fork/Join 框架

- 1.简单概述
- 2.Fork/Join 框架与传统线程池的区别

Stream概述

何为Stream?

Stream操作步骤

如何创建Stream流?

Stream的中间操作

筛选与切片

- 1.filter()方法
- 2.limit()方法
- 3.skip()方法
- 4.distinct()方法

映射

- 1.map()方法
- 2.flatMap()

排序

Stream的终止操作

查找与匹配

- 1.allMatch()
- 2.anyMatch()方法
- 3.noneMatch()方法
- 4.findFirst()方法
- 5.findAny()方法
- 6.count()方法
- 7.max()方法
- 8.min()方法
- 9.forEach()方法

规约

收集

如何收集Stream流?

什么是并行流?

Fork/Join 框架

Fork/Join 框架与传统线程池有啥区别?

Fork/Join框架实例

Java8中的并行流实例

什么是Optional类?

Optional类示例

- 1.创建Optional类
- 2.isPresent
- 3.orElse和orElseGet
- 4.orElseThrow
- 5.get
- 6.filter
- 7.map
- 8.flatMap

接口中的默认方法

默认方法的原则

接口中的静态方法

本地时间和时间戳

使用 LocalDate、LocalTime、LocalDateTime

Instant 时间戳

Duration 和 Period

日期的操纵

解析与格式化

时区的处理

与传统日期处理的转换

JDK5中的注解

- 1.注解 (@)
- 2.作用
- 3.如何理解注解?
- 4.关于注解
- 5.注解分为三个阶段
- 6.注解的属性类型
- 7.为注解增加属性

Java8中的注解

- 1.类型注解
- 2.重复注解

Java8对注解的增强

写在最后

## 写在前面

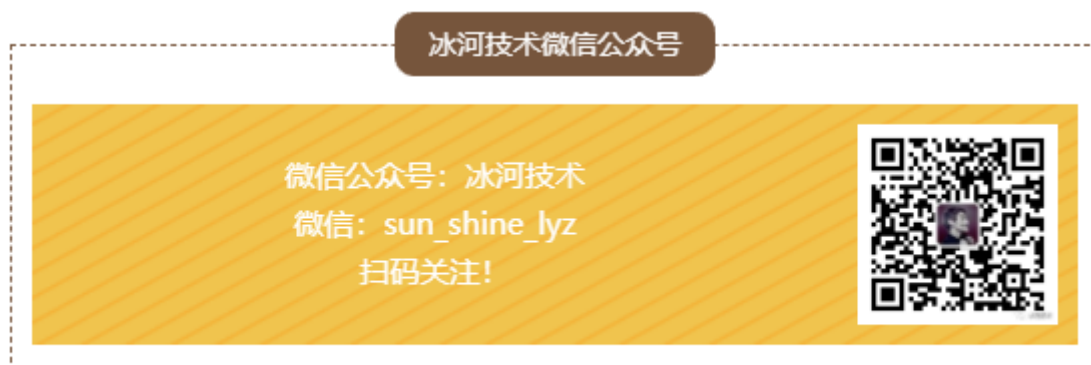
很多小伙伴留言说，冰河你能不能写一些关于Java8的文章呢，看书看不下去，看视频进度太慢。好吧，看到不少读者对Java8还是比较陌生的，那我就写一些关于Java8的文章吧，希望对大家有所帮助。经过一段时间的更新，【Java8新特性】专题基本更新完了，也算是告一段落了，今天，我就将【Java8新特性】专题的文章进行汇总，以PDF的形式开放出来，以便大家能够更好的学习和阅读。

## 作者简介

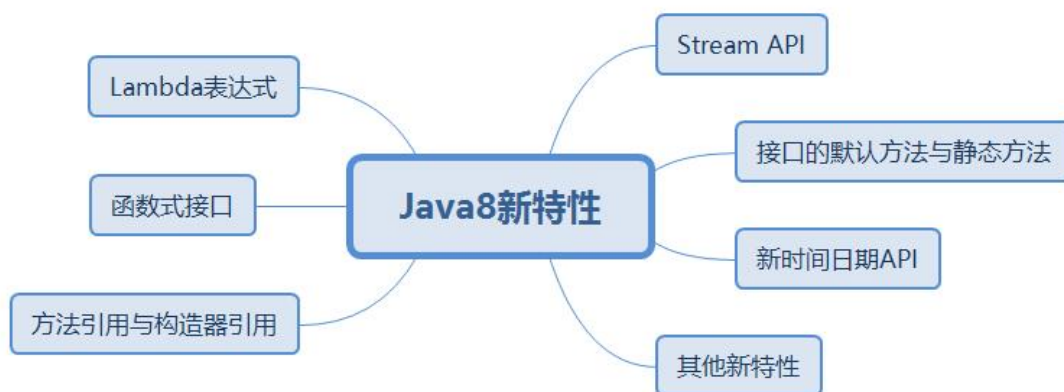
冰河，大数据架构师，编程专家，Mykit系列开源框架作者，多年来致力于分布式系统架构、微服务、分布式数据库、分布式事务与大数据技术的研究，曾主导过众多分布式系统、微服务及大数据项目的架构设计、研发和实施落地。在高并发、高可用、高可扩展性、高可维护性和大数据等领域拥有丰富的架构经验。对Hadoop, Storm, Spark, Flink等大数据框架源码进行过深度分析，并具有丰富的实战经验。目前在研究云原生技术。公众号【冰河技术】作者，《海量数据处理与大数据技术实战》、《MySQL开发、优化与运维实战》作者，基于最终消息可靠性的开源分布式事务框架mykit-transaction-message作者。

## 如何持续提升自身能力

如果你觉得冰河写的还不错，请微信搜索并关注「冰河技术」微信公众号，跟冰河学习高并发、分布式、微服务、大数据、互联网和云原生技术，「冰河技术」微信公众号更新了大量技术专题，每一篇技术文章干货满满！不少读者已经通过阅读「冰河技术」微信公众号文章，吊打面试官，成功跳槽到大厂；也有不少读者实现了技术上的飞跃，成为公司的技术骨干！如果你也想像他们一样提升自己的能力，实现技术能力的飞跃，进大厂，升职加薪，那就关注「冰河技术」微信公众号吧，每天更新超硬核技术干货，让你对如何提升技术能力不再迷茫！



## Java8有哪些新特性？

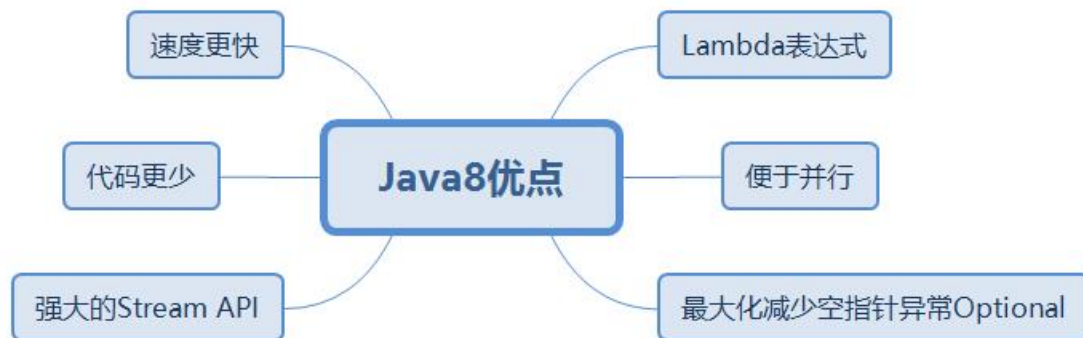


简单来说，Java8新特性如下所示：

- Lambda表达式
- 函数式接口
- 方法引用与构造器引用
- Stream API
- 接口的默认方法与静态方法
- 新时间日期API
- 其他新特性

其中，引用最广泛的新特性是Lambda表达式和Stream API。

## Java8有哪些优点？



简单来说Java8优点如下所示。

- 速度更快
- 代码更少（增加了新的语法Lambda表达式）
- 强大的Stream API
- 便于并行
- 最大化减少空指针异常Optional

## 什么是Lambda表达式？

Lambda表达式是一个匿名函数，我们可以这样理解Lambda表达式：Lambda是一段可以传递的代码（能够做到将代码像数据一样进行传递）。使用Lambda表达式能够写出更加简洁、灵活的代码。并且，使用Lambda表达式能够使Java的语言表达能力得到提升。

## 匿名内部类

在介绍如何使用Lambda表达式之前，我们先来看看匿名内部类，例如，我们使用匿名内部类比较两个Integer类型数据的大小。

```
Comparator<Integer> com = new Comparator<Integer>() {  
    @Override  
    public int compare(Integer o1, Integer o2) {  
        return Integer.compare(o1, o2);  
    }  
};
```

在上述代码中，我们使用匿名内部类实现了比较两个Integer类型数据的大小。

接下来，我们就可以将上述匿名内部类的实例作为参数，传递到其他方法中了，如下所示。

```
TreeSet<Integer> treeSet = new TreeSet<>(com);
```

完整的代码如下所示。

```
@Test
public void test1(){
    Comparator<Integer> com = new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            return Integer.compare(o1, o2);
        }
    };
    TreeSet<Integer> treeSet = new TreeSet<>(com);
}
```

我们分析下上述代码，在整个匿名内部类中，实际上真正有用的就是下面一行代码。

```
return Integer.compare(o1, o2);
```

其他的代码本质上都是“冗余”的。但是为了书写上面的一行代码，我们不得不在匿名内部类中书写更多的代码。

## Lambda表达式

如果使用Lambda表达式完成两个Integer类型数据的比较，我们该如何实现呢？

```
Comparator<Integer> com = (x, y) -> Integer.compare(x, y);
```

看到没，使用Lambda表达式，我们只需要使用一行代码就能够实现两个Integer类型数据的比较。

我们也可以将Lambda表达式传递到TreeSet的构造方法中，如下所示。

```
TreeSet<Integer> treeSet = new TreeSet<>((x, y) -> Integer.compare(x, y));
```

直观的感受就是使用Lambda表达式一行代码就能搞定匿名内部类多行代码的功能。

看到这，不少读者会问：我使用匿名内部类的方式实现比较两个整数类型的数据大小并不复杂啊！我为啥还要学习一种新的语法呢？



其实，我想说的是：上面咱们只是简单的列举了一个示例，接下来，咱们写一个稍微复杂一点的例子，来对比下使用匿名内部类与Lambda表达式哪种方式更加简洁。

# 对比常规方法和Lambda表达式

例如，现在有这么一个需求：**获取当前公司中员工年龄大于30岁的员工信息。**

首先，我们需要创建一个Employee实体类来存储员工的信息。

```
@Data
@Builder
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class Employee implements Serializable {
    private static final long serialVersionUID = -9079722457749166858L;
    private String name;
    private Integer age;
    private Double salary;
}
```

在Employee中，我们简单存储了员工的姓名、年龄和薪资。

接下来，我们创建一个存储多个员工的List集合，如下所示。

```
protected List<Employee> employees = Arrays.asList(
    new Employee("张三", 18, 9999.99),
    new Employee("李四", 38, 5555.55),
    new Employee("王五", 60, 6666.66),
    new Employee("赵六", 16, 7777.77),
    new Employee("田七", 18, 3333.33)
);
```

## 1.常规遍历集合

我们先使用常规遍历集合的方式来查找年龄大于等于30的员工信息。

```
public List<Employee> filterEmployeesByAge(List<Employee> list){
    List<Employee> employees = new ArrayList<>();
    for(Employee e : list){
        if(e.getAge() >= 30){
            employees.add(e);
        }
    }
    return employees;
}
```

接下来，我们测试一下上面的方法。

```
@Test
public void test3(){
    List<Employee> employeeList = filterEmployeesByAge(this.employees);
    for (Employee e : employeeList){
        System.out.println(e);
    }
}
```

运行test3方法，输出信息如下所示。

```
Employee(name=李四, age=38, salary=5555.55)
Employee(name=王五, age=60, salary=6666.66)
```

总体来说，查找年龄大于或者等于30的员工信息，使用常规遍历集合的方式稍显复杂了。

例如，需求发生了变化：获取当前公司中员工工资大于或者等于5000的员工信息。

此时，我们不得不再次创建一个按照工资过滤的方法。

```
public List<Employee> filterEmployeesBySalary(List<Employee> list){
    List<Employee> employees = new ArrayList<>();
    for(Employee e : list){
        if(e.getSalary() >= 5000){
            employees.add(e);
        }
    }
    return employees;
}
```

对比filterEmployeesByAge()方法和filterEmployeesBySalary方法后，我们发现，大部分的方法体是相同的，只是for循环中对于条件的判断不同。

如果此时我们再来一个需求，查找当前公司中年龄小于或者等于20的员工信息，那我们又要创建一个过滤方法了。看来使用常规方法是真的不方便啊！

## 笑容渐渐消失



这里，问大家一个问题：对于这种常规方法最好的优化方式是啥？相信有不少小伙伴会说：将公用的方法抽取出来。没错，将公用的方法抽取出来是一种优化方式，但它不是最好的方式。最好的方式是啥？那就是使用 **设计模式** 啊！设计模式可是无数前辈不断实践而总结出的设计原则和设计模式。大家可以查看 [《设计模式汇总——你需要掌握的23种设计模式都在这儿了！》](#) 一文来学习设计模式专题。

## 2.使用设计模式优化代码

如何使用设计模式来优化上面的方法呢，大家继续往下看，对于设计模式不熟悉的同学可以先根据 [《设计模式汇总——你需要掌握的23种设计模式都在这儿了！》](#) 来学习。

我们先定义一个泛型接口MyPredicate，对传递过来的数据进行过滤，符合规则返回true，不符合规则返回false。



```
public interface MyPredicate<T> {

    /**
     * 对传递过来的T类型的数据进行过滤
     * 符合规则返回true，不符合规则返回false
     */
    boolean filter(T t);

}
```

接下来，我们创建MyPredicate接口的实现类FilterEmployeeByAge来过滤年龄大于或者等于30的员工信息。

```
public class FilterEmployeeByAge implements MyPredicate<Employee> {
    @Override
    public boolean filter(Employee employee) {
        return employee.getAge() >= 30;
    }
}
```

我们定义一个过滤员工信息的方法，此时传递的参数不仅有员工的信息集合，同时还有一个我们定义的接口实例，在遍历员工集合时将符合过滤条件的员工信息返回。

```
//优化方式一
public List<Employee> filterEmployee(List<Employee> list, MyPredicate<Employee> myPredicate){
    List<Employee> employees = new ArrayList<>();
    for(Employee e : list){
        if(myPredicate.filter(e)){
            employees.add(e);
        }
    }
    return employees;
}
```

接下来，我们写一个测试方法来测试优化后的代码。

```
@Test
public void test4(){
    List<Employee> employeeList = this.filterEmployee(this.employees, new FilterEmployeeByAge());
    for (Employee e : employeeList){
        System.out.println(e);
    }
}
```

运行test4()方法，输出的结果信息如下所示。

```
Employee(name=李四, age=38, salary=5555.55)
Employee(name=王五, age=60, salary=6666.66)
```

写到这里，大家是否有一种豁然开朗的感觉呢？



没错，这就是设计模式的魅力，对于设计模式不熟悉的小伙伴，一定要参照《[设计模式汇总——你需要掌握的23种设计模式都在这儿了！](#)》来学习。

我们继续获取当前公司中工资大于或者等于5000的员工信息，此时，我们只需要创建一个 `FilterEmployeeBySalary` 类实现 `MyPredicate` 接口，如下所示。

```
public class FilterEmployeeBySalary implements MyPredicate<Employee>{
    @Override
    public boolean filter(Employee employee) {
        return employee.getSalary() >= 5000;
    }
}
```

接下来，就可以直接写测试方法了，在测试方法中继续调用 `filterEmployee(List<Employee> list, MyPredicate<Employee> myPredicate)` 方法。

```
@Test
public void test5(){
    List<Employee> employeeList = this.filterEmployee(this.employees, new
    FilterEmployeeBySalary());
    for (Employee e : employeeList){
        System.out.println(e);
    }
}
```

运行 `test5` 方法，输出的结果信息如下所示。

```
Employee(name=张三, age=18, salary=9999.99)
Employee(name=李四, age=38, salary=5555.55)
Employee(name=王五, age=60, salary=6666.66)
Employee(name=赵六, age=16, salary=7777.77)
```

可以看到，使用设计模式对代码进行优化后，无论过滤员工信息的需求如何变化，我们只需要创建 `MyPredicate` 接口的实现类来实现具体的过滤逻辑，然后在测试方法中调用 `filterEmployee(List<Employee> list, MyPredicate<Employee> myPredicate)` 方法将员工集合和过滤规则传入即可。



这里，问大家一个问题：上面优化代码使用的设计模式是哪种设计模式呢？如果是你，你会想到使用设计模式来优化自己的代码吗？小伙伴们自己先思考一下到底使用的设计模式是什么？**文末我会给出答案！**

使用设计模式优化代码也有不好的地方：每次定义一个过滤策略的时候，我们都要单独创建一个过滤类！！

### 3.匿名内部类

那使用匿名内部类是不是能够优化我们书写的代码呢，接下来，我们就使用匿名内部类来实现对员工信息的过滤。先来看过滤年龄大于或者等于30的员工信息。

```
@Test
public void test6(){
    List<Employee> employeeList = this.filterEmployee(this.employees, new
    MyPredicate<Employee>() {
        @Override
        public boolean filter(Employee employee) {
            return employee.getAge() >= 30;
        }
    });
    for (Employee e : employeeList){
        System.out.println(e);
    }
}
```

运行test6方法，输出如下结果信息。

```
Employee(name=李四, age=38, salary=5555.55)
Employee(name=王五, age=60, salary=6666.66)
```

再实现过滤工资大于或者等于5000的员工信息，如下所示。

```

@Test
public void test7(){
    List<Employee> employeeList = this.filterEmployee(this.employees, new
    MyPredicate<Employee>() {
        @Override
        public boolean filter(Employee employee) {
            return employee.getSalary() >= 5000;
        }
    });
    for (Employee e : employeeList){
        System.out.println(e);
    }
}

```

运行test7方法，输出如下结果信息。

```

Employee(name=张三, age=18, salary=9999.99)
Employee(name=李四, age=38, salary=5555.55)
Employee(name=王五, age=60, salary=6666.66)
Employee(name=赵六, age=16, salary=7777.77)

```

匿名内部类看起来比常规遍历集合的方法要简单些，并且将使用设计模式优化代码时，每次创建一个类来实现过滤规则写到了匿名内部类中，使得代码进一步简化了。

但是，使用匿名内部类代码的可读性不高，并且冗余代码也比较多！！

**那还有没有更加简化的方式呢？**



## 4.重头戏：Lambda表达式

在使用Lambda表达式时，我们还是要调用之前写的 `filterEmployee(List<Employee> list, MyPredicate<Employee> myPredicate)` 方法。

注意看，获取年龄大于或者等于30的员工信息。

```

@Test
public void test8(){
    filterEmployee(this.employees, (e) -> e.getAge() >=
    30).forEach(System.out::println);
}

```

看到没，使用Lambda表达式只需要一行代码就完成了员工信息的过滤和输出。是不是很6呢。



运行test8方法，输出如下的结果信息。

```
Employee(name=李四, age=38, salary=5555.55)
Employee(name=王五, age=60, salary=6666.66)
```

再来看使用Lambda表达式来获取工资大于或者等于5000的员工信息，如下所示。

```
@Test
public void test9(){
    filterEmployee(this.employees, (e) -> e.getSalary() >=
5000).forEach(System.out::println);
}
```

没错，使用Lambda表达式，又是一行代码就搞定了！！

运行test9方法，输出如下的结果信息。

```
Employee(name=张三, age=18, salary=9999.99)
Employee(name=李四, age=38, salary=5555.55)
Employee(name=王五, age=60, salary=6666.66)
Employee(name=赵六, age=16, salary=7777.77)
```

另外，使用Lambda表达式时，只需要给出需要过滤的集合，我们就能够实现从集合中过滤指定规则的元素，并输出结果信息。

## 5.重头戏：Stream API

使用Lambda表达式结合Stream API，只要给出相应的集合，我们就可以完成对集合的各种过滤并输出结果信息。

例如，此时只要有一个 `employees` 集合，我们使用Lambda表达式来获取工资大于或者等于5000的员工信息。

```
@Test
public void test10(){
    employees.stream().filter((e) -> e.getSalary() >=
5000).forEach(System.out::println);
}
```

没错，只给出一个集合，使用Lambda表达式和Stream API，一行代码就能够过滤出想要的元素并进行输出。

运行test10方法，输出如下的结果信息。

```
Employee(name=张三, age=18, salary=9999.99)
Employee(name=李四, age=38, salary=5555.55)
Employee(name=王五, age=60, salary=6666.66)
Employee(name=赵六, age=16, salary=7777.77)
```

如果我们只想要获取前两个员工的信息呢？其实也很简单，如下所示。

```
@Test
public void test11(){
    employees.stream().filter((e) -> e.getSalary() >=
5000).limit(2).forEach(System.out::println);
}
```

可以看到，我们在代码中添加了 `limit(2)` 来限制只获取两个员工信息。运行 `test11` 方法，输出如下的结果信息。

```
Employee(name=张三, age=18, salary=9999.99)
Employee(name=李四, age=38, salary=5555.55)
```

使用 Lambda 表达式和 Stream API 也可以获取指定的字段信息，例如获取工资大于或者等于 5000 的员工姓名。

```
@Test
public void test12(){
    employees.stream().filter((e) -> e.getSalary() >=
5000).map(Employee::getName).forEach(System.out::println);
}
```

可以看到，使用 `map` 过滤出了工资大于或者等于 5000 的员工姓名。运行 `test12` 方法，输出如下的结果信息。

```
张三
李四
王五
赵六
```

是不是很简单呢？



**就是这么简单**

最后，给出文中使用的设计模式：**策略模式**。

## 匿名类到Lambda表达式

我们先来看看从匿名类如何转换到Lambda表达式呢？

这里，我们可以使用两个示例来说明如何从匿名内部类转换为Lambda表达式。

- 匿名内部类到Lambda表达式

使用匿名内部类如下所示。

```
Runnable r = new Runnable(){
    @Override
    public void run(){
        System.out.println("Hello Lambda");
    }
}
```

转化为Lambda表达式如下所示。

```
Runnable r = () -> System.out.println("Hello Lambda");
```

- 匿名内部类作为参数传递到Lambda表达式作为参数传递

使用匿名内部类作为参数如下所示。

```
TreeSet<Integer> ts = new TreeSet<>(new Comparator<Integer>(){
    @Override
    public int compare(Integer o1, Integer o2){
        return Integer.compare(o1, o2);
    }
});
```

使用Lambda表达式作为参数如下所示。

```
TreeSet<Integer> ts = new TreeSet<>(
    (o1, o2) -> Integer.compare(o1, o2);
);
```

从直观上看，Lambda表达式要比常规的语法简洁的多。

## Lambda表达式的语法

Lambda表达式在Java语言中引入了“->”操作符，“->”操作符被称为Lambda表达式的操作符或者箭头操作符，它将Lambda表达式分为两部分：

- **左侧部分指定了Lambda表达式需要的所有参数。**

Lambda表达式本质上是对接口的实现，Lambda表达式的参数列表本质上对应着接口中方法的参数列表。

- **右侧部分指定了Lambda体，即Lambda表达式要执行的功能。**

Lambda体本质上就是接口方法具体实现的功能。

我们可以将Lambda表达式的语法总结如下。

### 1.语法格式一：无参，无返回值，Lambda体只有一条语句

```
Runnable r = () -> System.out.println("Hello Lambda");
```

具体示例如下所示。

```
@Test
public void test1(){
    Runnable r = () -> System.out.println("Hello Lambda");
    new Thread(r).start();
}
```

### 2.语法格式二：Lambda表达式需要一个参数，并且无返回值

```
Consumer<String> func = (s) -> System.out.println(s);
```

具体示例如下所示。

```
@Test
public void test2(){
    Consumer<String> consumer = (x) -> System.out.println(x);
    consumer.accept("Hello Lambda");
}
```

### 3.语法格式三：Lambda只需要一个参数时，参数的小括号可以省略

```
Consumer<String> func = s -> System.out.println(s);
```

具体示例如下所示。

```
@Test
public void test3(){
    Consumer<String> consumer = x -> System.out.println(x);
    consumer.accept("Hello Lambda");
}
```

### 4.语法格式四：Lambda需要两个参数，并且有返回值

```
BinaryOperator<Integer> bo = (a, b) -> {
    System.out.println("函数式接口");
    return a + b;
};
```

具体示例如下所示。



```
@Test
public void test4(){
    Comparator<Integer> comparator = (x, y) -> {
        System.out.println("函数式接口");
        return Integer.compare(x, y);
    };
}
```

### 5. 语法格式五：当Lambda体只有一条语句时，return和大括号可以省略

```
BinaryOperator<Integer> bo = (a, b) -> a + b;
```

具体示例如下所示。

```
@Test
public void test5(){
    Comparator<Integer> comparator = (x, y) -> Integer.compare(x, y);
}
```

### 6. 语法格式六：Lambda表达式的参数列表的数据类型可以省略不写，因为JVM编译器能够通过上下文推断出数据类型，这就是“类型推断”

```
BinaryOperator<Integer> bo = (Integer a, Integer b) -> {
    return a + b;
};
```

等同于

```
BinaryOperator<Integer> bo = (a, b) -> {
    return a + b;
};
```

上述 Lambda 表达式中的参数类型都是由编译器推断得出的。Lambda 表达式中无需指定类型，程序依然可以编译，这是因为 javac 根据程序的上下文，在后台推断出了参数的类型。Lambda 表达式的类型依赖于上下文环境，是由编译器推断出来的。这就是所谓的“类型推断”。

## 函数式接口

Lambda表达式需要函数式接口的支持，所以，我们有必要来说说什么什么是函数式接口。

只包含一个抽象方法的接口，称为函数式接口。

可以通过 Lambda 表达式来创建该接口的对象。（若 Lambda表达式抛出一个受检异常，那么该异常需要在目标接口的抽象方法上进行声明）。

可以在任意函数式接口上使用 @FunctionalInterface 注解，这样做可以检查它是否是一个函数式接口，同时 javadoc 也会包含一条声明，说明这个接口是一个函数式接口。

我们可以自定义函数式接口，并使用Lambda表达式来实现相应的功能。

例如，使用函数式接口和Lambda表达式实现对字符串的处理功能。

首先，我们定义一个函数式接口MyFunc，如下所示。

```
@FunctionalInterface
public interface MyFunc <T> {
    public T getValue(T t);
}
```

接下来，我们定义一个操作字符串的方法，其中参数为MyFunc接口实例和需要转换的字符串。

```
public String handlerString(MyFunc<String> myFunc, String str){
    return myFunc.getValue(str);
}
```

接下来，我们对自定义的函数式接口进行测试，此时我们传递的函数式接口的参数为Lambda表达式，并且将字符串转化为大写。

```
@Test
public void test6(){
    String str = handlerString((s) -> s.toUpperCase(), "binghe");
    System.out.println(str);
}
```

运行test6方法，得出的结果信息如下所示。

```
BINGHE
```

我们也可以截取字符串的某一部分，如下所示。

```
@Test
public void test7(){
    String str = handlerString((s) -> s.substring(0,4), "binghe");
    System.out.println(str);
}
```

运行test7方法，得出的结果信息如下所示。

```
bing
```

可以看到，我们可以通过 `handlerString(MyFunc<String> myFunc, String str)` 方法结合Lambda表达式对字符串进行任意操作。

**注意：作为参数传递 Lambda 表达式：**为了将 Lambda 表达式作为参数传递，接收Lambda表达式的参数类型必须是与该 Lambda 表达式兼容的函数式接口的类型。

## Lambda表达式典型案例

### 案例一

#### 需求

调用Collections.sort()方法，通过定制排序比较两个Employee（先比较年龄，年龄相同按姓名比较），使用Lambda表达式作为参数传递。

#### 实现

这里，我们先创建一个Employee类，为了满足需求，我们在Employee类中定义了姓名、年龄和工资三个字段，如下所示。

```
@Data
@Builder
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class Employee implements Serializable {
    private static final long serialVersionUID = -9079722457749166858L;
    private String name;
    private Integer age;
    private Double salary;
}
```

接下来，我们在TestLambda类中定义一个成员变量employees，employees变量是一个List集合，存储了Employee的一个列表，如下所示。

```
protected List<Employee> employees = Arrays.asList(
    new Employee("张三", 18, 9999.99),
    new Employee("李四", 38, 5555.55),
    new Employee("王五", 60, 6666.66),
    new Employee("赵六", 8, 7777.77),
    new Employee("田七", 58, 3333.33)
);
```

前期的准备工作完成了，接下来，我们就可以实现具体的业务逻辑了。

```
@Test
public void test1(){
    Collections.sort(employees, (e1, e2) -> {
        if(e1.getAge() == e2.getAge()){
            return e1.getName().compareTo(e2.getName());
        }
        return Integer.compare(e1.getAge(), e2.getAge());
    });
    employees.stream().forEach(System.out::println);
}
```

上述代码比较简单，我就不赘述具体逻辑了。运行test1方法，得出的结果信息如下所示。

```
Employee(name=赵六, age=8, salary=7777.77)
Employee(name=张三, age=18, salary=9999.99)
Employee(name=李四, age=38, salary=5555.55)
Employee(name=田七, age=58, salary=3333.33)
Employee(name=王五, age=60, salary=6666.66)
```

如果想倒叙输出如何处理呢，只需要在将 `return Integer.compare(e1.getAge(), e2.getAge());` 修改成 `-return Integer.compare(e1.getAge(), e2.getAge());` 即可，如下所示。

```

@Test
public void test1(){
    Collections.sort(employees, (e1, e2) -> {
        if(e1.getAge() == e2.getAge()){
            return e1.getName().compareTo(e2.getName());
        }
        return -Integer.compare(e1.getAge(), e2.getAge());
    });
    employees.stream().forEach(System.out::println);
}

```

再次运行test1方法，得出的结果信息如下所示。

```

Employee(name=王五, age=60, salary=6666.66)
Employee(name=田七, age=58, salary=3333.33)
Employee(name=李四, age=38, salary=5555.55)
Employee(name=张三, age=18, salary=9999.99)
Employee(name=赵六, age=8, salary=7777.77)

```

结果符合我们的需求。

## 案例二

### 需求

- 1.声明函数式接口，接口中声明抽象方法 `public String getValue(String str);`
- 2.声明类TestLambda，类中编写方法使用接口作为参数，将一个字符串转换为大写，并作为方法的返回值。
- 3.再将一个字符串的第2个和第4个索引位置进行截取子串。

### 实现

首先，创建一个函数式接口MyFunction，在MyFunction接口上加上注解@FunctionalInterface标识接口是一个函数式接口。如下所示。

```

@FunctionalInterface
public interface MyFunction {
    public String getValue(String str);
}

```

在TestLambda类中声明stringHandler方法，参数分别为待处理的字符串和函数式接口的实例，方法中的逻辑就是调用函数式接口的方法来处理字符串，如下所示。

```

public String stringHandler(String str, MyFunction myFunction){
    return myFunction.getValue(str);
}

```

接下来，我们实现将一个字符串转换为大写的逻辑，如下所示。

```
@Test
public void test2(){
    String value = stringHandler("binghe", (s) -> s.toUpperCase());
    System.out.println(value);
}
```

运行test2方法，得出如下的结果信息。

```
BINGHE
```

我们再来实现字符串截取的操作，如下所示。

```
@Test
public void test3(){
    String value = stringHandler("binghe", (s) -> s.substring(1, 3));
    System.out.println(value);
}
```

**注意：**需求中是按照第2个和第4个索引位置进行截取子串，字符串的下标是从0开始的，所以这里截取字符串时使用的是substring(1, 3)，而不是substring(2, 4)，这也是很多小伙伴容易犯的错误。

**另外，**使用上述Lambda表达式形式，可以实现字符串的任意处理，并返回处理后的新字符串。

运行test3方法，结果如下所示。

```
in
```

## 案例三

### 需求

- 1.声明一个带两个泛型的函数式接口，泛型类型为<T, R>，其中，T作为参数的类型，R作为返回值的类型。
- 2.接口中声明对象的抽象方法。
- 3.在TestLambda类中声明方法。使用接口作为参数计算两个long型参数的和。
- 4.再就按两个long型参数的乘积。

### 实现

首先，我们按照需求定义函数式接口MyFunc，如下所示。

```
@FunctionalInterface
public interface MyFunc<T, R> {

    R getValue(T t1, T t2);
}
```

接下来，我们在TestLambda类中创建一个处理两个long型数据的方法，如下所示。

```
public void operate(Long num1, Long num2, MyFunc<Long, Long> myFunc){
    System.out.println(myFunc.getValue(num1, num2));
}
```

我们可以使用下面的方法来完成两个long型参数的和。

```
@Test
public void test4(){
    operate(100L, 200L, (x, y) -> x + y);
}
```

运行test4方法，结果如下所示。

300

实现两个long型数据的乘积，也很简单。

```
@Test
public void test5(){
    operate(100L, 200L, (x, y) -> x * y);
}
```

运行test5方法，结果如下所示。

20000

看到这里，我相信很多小伙伴已经对Lambda表达式有了更深层次的理解。只要多多练习，就能够更好的掌握Lambda表达式的精髓。

## 函数式接口总览

这里，我使用表格的形式来简单说明下Java8中提供的函数式接口。

### 四大核心函数式接口

首先，我们来看四大核心函数式接口，如下所示。

函数式接口	参数类型	返回类型	使用场景
Consumer消费型接口	T	void	对类型为T的对象应用操作，接口定义的方法：void accept(T t)
Supplier供给型接口	无	T	返回类型为T的对象，接口定义的方法：T get()
Function<T, R>函数式接口	T	R	对类型为T的对象应用操作，并R类型的返回结果。接口定义的方法：R apply(T t)
Predicate断言型接口	T	boolean	确定类型为T的对象是否满足约束条件，并返回boolean类型的数据。接口定义的方法：boolean test(T t)

## 其他函数接口

除了四大核心函数接口外，Java8还提供了一些其他的函数式接口。

函数式接口	参数类型	返回类型	使用场景
BiFunction(T, U, R)	T, U	R	对类型为T,U的参数应用操作，返回R类型的结果。接口定义的方法：R apply(T t, U u)
UnaryOperator (Function 子接口)	T	T	对类型为T的对象进行一元运算，并返回T类型的结果。包含方法为 T apply(T t)
BinaryOperator (BiFunction 子接口)	T, T	T	对类型为T的对象进行二元运算，并返回T类型的结果。包含方法为 T apply(T t1, T t2)
BiConsumer<T, U>	T, U	void	对类型为T, U 参数应用操作。包含方法为 void accept(T t, U u)
ToIntFunction	T	int	计算int值的函数
ToLongFunction	T	long	计算long值的函数
ToDoubleFunction	T	double	计算double值的函数
IntFunction	int	R	参数为int 类型的函数
LongFunction	long	R	参数为 long类型的函数
DoubleFunction	double	R	参数为double类型的函数

## 四大核心函数式接口

### Consumer接口

#### 1.接口说明

Consumer接口是消费性接口，无返回值。Java8中对Consumer的定义如下所示。

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

#### 2.使用示例

```
public void handlerConsumer(Integer number, Consumer<Integer> consumer){
    consumer.accept(number);
}

@Test
public void test1(){
    this.handlerConsumer(10000, (i) -> System.out.println(i));
}
```

## Supplier接口

### 1.接口说明

Supplier接口是供给型接口，有返回值，Java8中对Supplier接口的定义如下所示。

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

### 2.使用示例

```
public List<Integer> getNumberList(int num, Supplier<Integer> supplier){
    List<Integer> list = new ArrayList<>();
    for(int i = 0; i < num; i++){
        list.add(supplier.get())
    }
    return list;
}

@Test
public void test2(){
    List<Integer> numberList = this.getNumberList(10, () -> new
Random().nextInt(100));
    numberList.stream().forEach(System.out::println);
}
```

## Function接口

### 1.接口说明

Function接口是函数型接口，有返回值，Java8中对Function接口的定义如下所示。

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose(Function<? super V, ? extends T> before)
    {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }
}
```



```

    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}

```

## 2.使用示例

```

public String handlerString(String str, Function<String, String> func){
    return func.apply(str);
}

@Test
public void test3(){
    String str = this.handlerString("binghe", (s) -> s.toUpperCase());
    System.out.println(str);
}

```

# Predicate接口

## 1.接口说明

Predicate接口是断言型接口，返回值类型为boolean，Java8中对Predicate接口的定义如下所示。

```

@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> negate() {
        return (t) -> !test(t);
    }

    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }

    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef
            ? Objects::isNull
            : object -> targetRef.equals(object));
    }
}

```

## 2.使用示例

```
public List<String> filterString(List<String> list, Predicate<String> predicate)
{
    List<String> strList = new ArrayList<>();
    for(String str : list){
        if(predicate.test(str)){
            strList.add(str);
        }
    }
    return strList;
}

@Test
public void test4(){
    List<String> list = Arrays.asList("Hello", "Lambda", "binghe", "lyz",
    "world");
    List<String> strList = this.filterString(list, (s) -> s.length() >= 5);
    strList.stream().forEach(System.out::println);
}
```

注意：只要我们学会了Java8中四大核心函数式接口的用法，其他函数式接口我们也知道如何使用了！

## Java8知识点总结

### Java7与Java8中的HashMap

- JDK7 HashMap结构为数组+链表（发生元素碰撞时，会将新元素添加到链表开头）
- JDK8 HashMap结构为数组+链表+红黑树（发生元素碰撞时，会将新元素添加到链表末尾，当HashMap总容量大于等于64，并且某个链表的大小大于等于8，会将链表转化为红黑树（注意：红黑树是二叉树的一种））

### JDK8 HashMap重排序

如果删除了HashMap中红黑树的某个元素导致元素重排序时，不需要计算待重排序的元素的HashCode码，只需要将当前元素放到（HashMap总长度+当前元素在HashMap中的位置）的位置即可。

### 筛选与切片

- filter——接收 Lambda，从流中排除某些元素。
- limit——截断流，使其元素不超过给定数量。
- skip(n) —— 跳过元素，返回一个扔掉了前 n 个元素的流。若流中元素不足 n 个，则返回一个空流。与 limit(n) 互补
- distinct——筛选，通过流所生成元素的 hashCode() 和 equals() 去除重复元素

### 中间操作

- map——接收 Lambda，将元素转换成其他形式或提取信息。接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
- flatMap——接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流

- sorted()——自然排序
- sorted(Comparator com)——定制排序

## 终止操作

---

- allMatch——检查是否匹配所有元素
- anyMatch——检查是否至少匹配一个元素
- noneMatch——检查是否没有匹配的元素
- findFirst——返回第一个元素
- findAny——返回当前流中的任意元素
- count——返回流中元素的总个数
- max——返回流中最大值
- min——返回流中最小值

## 归约

---

- reduce(T identity, BinaryOperator) / reduce(BinaryOperator) ——可以将流中元素反复结合起来，得到一个值。
- collect——将流转换为其他形式。接收一个 Collector接口的实现，用于给Stream中元素做汇总的方法

**注意：流进行了终止操作后，不能再次使用**

## Optional 容器类

---

用于尽量避免空指针异常

- Optional.of(T t) : 创建一个 Optional 实例
- Optional.empty() : 创建一个空的 Optional 实例
- Optional.ofNullable(T t):若 t 不为 null,创建 Optional 实例,否则创建空实例
- isPresent() : 判断是否包含值
- orElse(T t) : 如果调用对象包含值，返回该值，否则返回t
- orElseGet(Supplier s) :如果调用对象包含值，返回该值，否则返回 s 获取的值
- map(Function f): 如果有值对其处理，并返回处理后的Optional，否则返回 Optional.empty()
- flatMap(Function mapper):与 map 类似，要求返回值必须是Optional

## 方法引用

---

当要传递给Lambda体的操作，已经有实现的方法了，可以使用方法引用！这里需要注意的是：实现抽象方法的参数列表，必须与方法引用方法的参数列表保持一致！

那么什么是方法引用呢？方法引用就是操作符“::”将方法名和对象或类的名字分隔开来。

有如下三种使用情况：

- 对象::实例方法
- 类::静态方法
- 类::实例方法

这里，我们可以列举几个示例。

例如：

```
(x) -> System.out.println(x);
```

等同于：

```
System.out::println
```

例如：

```
BinaryOperator<Double> bo = (x, y) -> Math.pow(x, y);
```

等同于

```
BinaryOperator<Double> bo = Math::pow;
```

例如：

```
compare((x, y) -> x.equals(y), "binghe", "binghe")
```

等同于

```
compare(String::equals, "binghe", "binghe")
```

**注意：** 当需要引用方法的第一个参数是调用对象，并且第二个参数是需要引用方法的第二个参数(或无参数)时： `ClassName::methodName` 。

## 构造器引用

格式如下所示：

```
ClassName::new
```

与函数式接口相结合，自动与函数式接口中方法兼容。可以把构造器引用赋值给定义的方法，与构造器参数列表要与接口中抽象方法的参数列表一致！

例如：

```
Function<Integer, MyClass> fun = (n) -> new MyClass(n);
```

等同于

```
Function<Integer, MyClass> fun = MyClass::new;
```

## 数组引用

格式如下所示。

```
type[]::new
```

例如：

```
Function<Integer, Integer[]> fun = (n) -> new Integer[n];
```

等同于

```
Function<Integer, Integer[]> fun = Integer[]::new;
```

## 什么是Stream?

Java8中有两大最为重要的改变。第一个是 Lambda 表达式；另外一个则是 Stream API(java.util.stream.\*).

Stream 是 Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。使用Stream API 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。也可以使用 Stream API 来并行执行操作。简而言之，Stream API 提供了一种高效且易于使用的处理数据的方式

流是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。“集合讲的是数据，流讲的是计算！”

**注意：**

- ① Stream 自己不会存储元素。
- ② Stream 不会改变源对象。相反，他们会返回一个持有结果的新Stream。
- ③ Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。

## Stream操作的三个步骤

- 创建 Stream

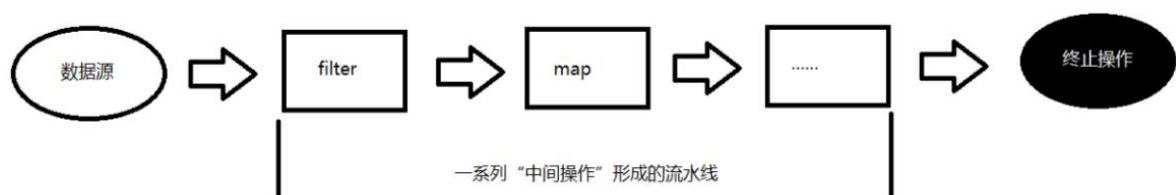
一个数据源（如：集合、数组），获取一个流。

- 中间操作

一个中间操作链，对数据源的数据进行处理。

- 终止操作(终端操作)

一个终止操作，执行中间操作链，并产生结果。



## 如何创建Stream?

Java8 中的 Collection 接口被扩展，提供了两个获取流的方法：

### 1.获取Stream

- default Stream stream(): 返回一个顺序流
- default Stream parallelStream(): 返回一个并行流

### 2.由数组创建Stream

Java8 中的 Arrays 的静态方法 stream() 可以获取数组流：

- static Stream stream(T[] array): 返回一个流

重载形式，能够处理对应基本类型的数组：

- `public static IntStream stream(int[] array)`
- `public static LongStream stream(long[] array)`
- `public static DoubleStream stream(double[] array)`

### 3.由值创建流

可以使用静态方法 `Stream.of()`，通过显示值创建一个流。它可以接收任意数量的参数。

- `public static Stream of(T... values)`：返回一个流

### 4.由函数创建流

由函数创建流可以创建无限流。

可以使用静态方法 `Stream.iterate()` 和 `Stream.generate()`，创建无限流。

- 迭代

`public static Stream iterate(final T seed, final UnaryOperator f)`

- 生成

`public static Stream generate(Supplier s)`

## Stream的中间操作

多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行任何的处理！而在终止操作时一次性全部处理，称为“惰性求值”

### 1.筛选与切片

方 法	描 述
<code>filter(Predicate p)</code>	接收 Lambda，从流中排除某些元素。
<code>distinct()</code>	筛选，通过流所生成元素的 <code>hashCode()</code> 和 <code>equals()</code> 去除重复元素
<code>limit(long maxSize)</code>	截断流，使其元素不超过给定数量。
<code>skip(long n)</code>	跳过元素，返回一个扔掉了前 n 个元素的流。若流中元素不足 n 个，则返回一个空流。与 <code>limit(n)</code> 互补

### 2.映射

方 法	描 述
<code>map(Function f)</code>	接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
<code>mapToDouble(ToDoubleFunction f)</code>	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 <code>DoubleStream</code> 。
<code>mapToInt(ToIntFunction f)</code>	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 <code>IntStream</code> 。
<code>mapToLong(ToLongFunction f)</code>	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 <code>LongStream</code> 。
<code>flatMap(Function f)</code>	接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流

### 3.排序

方 法	描 述
<code>sorted()</code>	产生一个新流，其中按自然顺序排序
<code>sorted(Comparator comp)</code>	产生一个新流，其中按比较器顺序排序

## Stream 的终止操作

终端操作会从流的流水线生成结果。其结果可以是任何不是流的价值，例如：List、Integer，甚至是 void。

### 1.查找与匹配

方 法	描 述
<code>allMatch(Predicate p)</code>	检查是否匹配所有元素
<code>anyMatch(Predicate p)</code>	检查是否至少匹配一个元素
<code>noneMatch(Predicate p)</code>	检查是否没有匹配所有元素
<code>findFirst()</code>	返回第一个元素
<code>findAny()</code>	返回当前流中的任意元素

方 法	描 述
<code>count()</code>	返回流中元素总数
<code>max(Comparator c)</code>	返回流中最大值
<code>min(Comparator c)</code>	返回流中最小值
<code>forEach(Consumer c)</code>	<b>内部迭代</b> (使用 Collection 接口需要用户去做迭代，称为 <b>外部迭代</b> 。相反，Stream API 使用内部迭代——它帮你把迭代做了)



## 2.规约

<code>reduce(T iden, BinaryOperator b)</code>	可以将流中元素反复结合起来，得到一个值。 返回 <code>T</code>
<code>reduce(BinaryOperator b)</code>	可以将流中元素反复结合起来，得到一个值。 返回 <code>Optional&lt;T&gt;</code>

## 3.收集

方 法	描 述
<code>collect(Collector c)</code>	将流转换为其他形式。接收一个 <code>Collector</code> 接口的实现，用于给Stream中元素做汇总的方法

`Collector` 接口中方法的实现决定了如何对流执行收集操作(如收集到 `List`、`Set`、`Map`)。但是 `Collectors` 实用类提供了很多静态方法，可以方便地创建常见收集器实例，具体方法与实例如下表

方法	返回类型	作用
<code>toList</code>	<code>List&lt;T&gt;</code>	把流中元素收集到List
<code>List&lt;Employee&gt; emps= list.stream().collect(Collectors.toList());</code>		
<code>toSet</code>	<code>Set&lt;T&gt;</code>	把流中元素收集到Set
<code>Set&lt;Employee&gt; emps= list.stream().collect(Collectors.toSet());</code>		
<code>toCollection</code>	<code>Collection&lt;T&gt;</code>	把流中元素收集到创建的集合
<code>Collection&lt;Employee&gt;emps=list.stream().collect(Collectors.toCollection(ArrayList::new));</code>		
<code>counting</code>	<code>Long</code>	计算流中元素的个数
<code>long count = list.stream().collect(Collectors.counting());</code>		
<code>summingInt</code>	<code>Integer</code>	对流中元素的整数属性求和
<code>inttotal=list.stream().collect(Collectors.summingInt(Employee::getSalary));</code>		
<code>averagingInt</code>	<code>Double</code>	计算流中元素Integer属性的平均值
<code>doubleavg= list.stream().collect(Collectors.averagingInt(Employee::getSalary));</code>		
<code>summarizingInt</code>	<code>IntSummaryStatistics</code>	收集流中Integer属性的统计值。 如：平均值
<code>IntSummaryStatisticsiss= list.stream().collect(Collectors.summarizingInt(Employee::getSalary));</code>		



<b>joining</b>	String	连接流中每个字符串
<b>String str= list.stream().map(Employee::getName).collect(Collectors.joining());</b>		
<b>maxBy</b>	Optional<T>	根据比较器选择最大值
<b>Optional&lt;Emp&gt; max= list.stream().collect(Collectors.maxBy(comparingInt(Employee::getSalary)));</b>		
<b>minBy</b>	Optional<T>	根据比较器选择最小值
<b>Optional&lt;Emp&gt; min = list.stream().collect(Collectors.minBy(comparingInt(Employee::getSalary)));</b>		
<b>reducing</b>	归约产生的类型	从一个作为累加器的初始值开始，利用BinaryOperator与流中元素逐个结合，从而归约成单个值
<b>int total=list.stream().collect(Collectors.reducing(0, Employee::getSalar, Integer::sum));</b>		
<b>collectingAndThen</b>	转换函数返回的类型	包裹另一个收集器，对其结果转换函数
<b>int how= list.stream().collect(Collectors.collectingAndThen(Collectors.toList(), List::size));</b>		
<b>groupingBy</b>	Map<K, List<T>>	根据某属性值对流分组，属性为K，结果为V
<b>Map&lt;Emp.Status, List&lt;Emp&gt;&gt; map= list.stream().collect(Collectors.groupingBy(Employee::getStatus));</b>		
<b>partitioningBy</b>	Map<Boolean, List<T>>	根据true或false进行分区
<b>Map&lt;Boolean, List&lt;Emp&gt;&gt; vd= list.stream().collect(Collectors.partitioningBy(Employee::getManage));</b>		

## 并行流与串行流

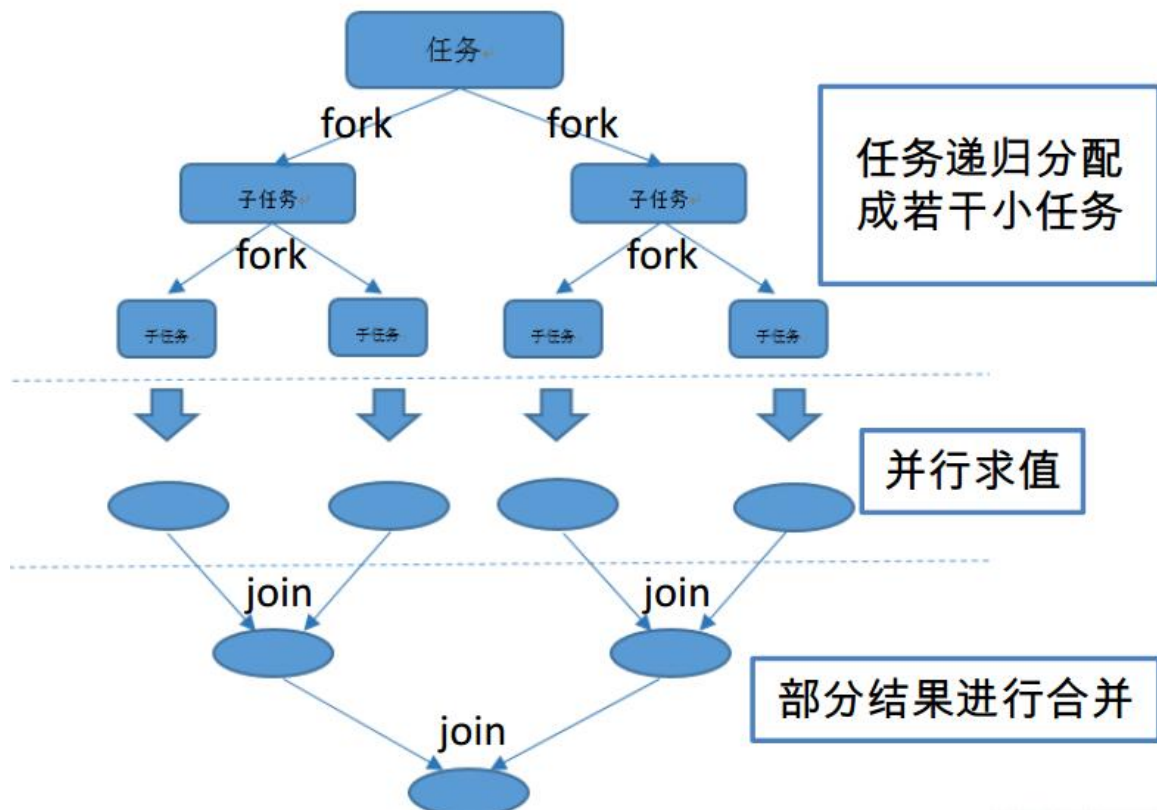
并行流就是把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流。

Java 8 中将并行进行了优化，我们可以很容易的对数据进行并行操作。Stream API 可以声明性地通过 parallel() 与 sequential() 在并行流与顺序流之间进行切换

## Fork/Join 框架

### 1.简单概述

Fork/Join 框架：就是在必要的情况下，将一个大任务，进行拆分(fork)成若干个小任务（拆到不可再拆时），再将一个个的小任务运算的结果进行 join 汇总。



## 2.Fork/Join 框架与传统线程池的区别

采用“工作窃取”模式（work-stealing）：

当执行新的任务时它可以将其拆分成更小的任务执行，并将小任务加到线程队列中，然后再从一个随机线程的队列中偷一个并把它放在自己的队列中。

相对于一般的线程池实现, fork/join 框架的优势体现在对其中包含的任务的处理方式上. 在一般的线程池中, 如果一个线程正在执行的任务由于某些原因无法继续运行, 那么该线程会处于等待状态. 而在 fork/join 框架实现中, 如果某个子问题由于等待另外一个子问题的完成而无法继续运行, 那么处理该子问题的线程会主动寻找其他尚未运行的子问题来执行. 这种方式减少了线程的等待时间, 提高了性能。

## Stream概述

Java8中有两大最为重要的改变。第一个是 Lambda 表达式；另外一个则是 Stream API(`java.util.stream.*`)。

Stream 是 Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。使用 Stream API 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。也可以使用 Stream API 来并行执行操作。简而言之，Stream API 提供了一种高效且易于使用的处理数据的方式。

## 何为Stream?

**流(Stream) 到底是什么呢？**

可以这么理解流：流就是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。

“集合讲的是数据，流讲的是计算！”

**注意：**

①Stream 自己不会存储元素。

②Stream 不会改变源对象。相反，他们会返回一个持有结果的新Stream。

③Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。

## Stream操作步骤

### 1.创建 Stream

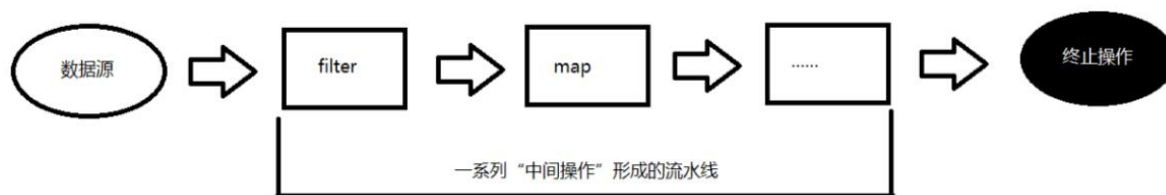
一个数据源（如：集合、数组），获取一个流。

### 2.中间操作

一个中间操作链，对数据源的数据进行处理。

### 3.终止操作(终端操作)

一个终止操作，执行中间操作链，并产生结果。



## 如何创建Stream流？

这里，创建测试类TestStreamAPI1，所有的操作都是在TestStreamAPI1类中完成的。

### (1) 通过Collection系列集合提供的stream()方法或者parallelStream()方法来创建Stream。

在Java8中，Collection 接口被扩展，提供了两个获取流的默认方法，如下所示。

```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
default Stream<E> parallelStream() {
    return StreamSupport.stream(spliterator(), true);
}
```

其中，stream()方法返回一个顺序流，parallelStream()方法返回一个并行流。

我们可以使用如下代码方式来创建顺序流和并行流。

```
List<String> list = new ArrayList<>();
list.stream();
list.parallelStream();
```

### (2) 通过Arrays中的静态方法stream()获取数组流。

Java8 中的 Arrays类的静态方法 stream() 可以获取数组流，如下所示。

```
public static <T> Stream<T> stream(T[] array) {
    return stream(array, 0, array.length);
}
```

上述代码的作用为：传入一个泛型数组，返回这个泛型的Stream流。

除此之外，在Arrays类中还提供了stream()方法的如下重载形式。

```
public static <T> Stream<T> stream(T[] array) {
    return stream(array, 0, array.length);
}

public static <T> Stream<T> stream(T[] array, int startInclusive, int
endExclusive) {
    return StreamSupport.stream(spliterator(array, startInclusive,
endExclusive), false);
}

public static IntStream stream(int[] array) {
    return stream(array, 0, array.length);
}

public static IntStream stream(int[] array, int startInclusive, int
endExclusive) {
    return StreamSupport.intStream(spliterator(array, startInclusive,
endExclusive), false);
}

public static LongStream stream(long[] array) {
    return stream(array, 0, array.length);
}

public static LongStream stream(long[] array, int startInclusive, int
endExclusive) {
    return StreamSupport.longStream(spliterator(array, startInclusive,
endExclusive), false);
}

public static DoubleStream stream(double[] array) {
    return stream(array, 0, array.length);
}

public static DoubleStream stream(double[] array, int startInclusive, int
endExclusive) {
    return StreamSupport.doubleStream(spliterator(array, startInclusive,
endExclusive), false);
}
```

基本上能够满足基本将基本类型的数组转化为Stream流的操作。

我们可以通过下面的代码示例来使用Arrays类的stream()方法来创建Stream流。

```
Integer[] nums = new Integer[]{1,2,3,4,5,6,7,8,9};
Stream<Integer> numStream = Arrays.stream(nums);
```

### (3) 通过Stream类的静态方法of()获取数组流。

可以使用静态方法 Stream.of(), 通过显示值创建一个流。它可以接收任意数量的参数。

我们先来看看Stream的of()方法，如下所示。

```

public static<T> Stream<T> of(T t) {
    return StreamSupport.stream(new Streams.StreamBuilderImpl<>(t), false);
}
@SafeVarargs
@SuppressWarnings("varargs")
public static<T> Stream<T> of(T... values) {
    return Arrays.stream(values);
}

```

可以看到，在Stream类中，提供了两个of()方法，一个只需要传入一个泛型参数，一个需要传入一个可变泛型参数。

我们可以使用下面的代码示例来使用of方法创建一个Stream流。

```

Stream<String> strStream = Stream.of("a", "b", "c");

```

#### (4) 创建无限流

可以使用静态方法 Stream.iterate() 和Stream.generate(), 创建无限流。

先来看看Stream类中iterate()方法和generate()方法的源码，如下所示。

```

public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f) {
    Objects.requireNonNull(f);
    final Iterator<T> iterator = new Iterator<T>() {
        @SuppressWarnings("unchecked")
        T t = (T) Streams.NONE;

        @Override
        public boolean hasNext() {
            return true;
        }

        @Override
        public T next() {
            return t = (t == Streams.NONE) ? seed : f.apply(t);
        }
    };
    return StreamSupport.stream(Spliterators.spliteratorUnknownSize(
        iterator,
        Spliterator.ORDERED | Spliterator.IMMUTABLE), false);
}

public static<T> Stream<T> generate(Supplier<T> s) {
    Objects.requireNonNull(s);
    return StreamSupport.stream(
        new StreamSpliterators.InfiniteSupplyingSpliterator.OfRef<>
        (Long.MAX_VALUE, s), false);
}

```

通过源码可以看出，iterate()方法主要是使用“迭代”的方式生成无限流，而generate()方法主要是使用“生成”的方式生成无限流。我们可以使用下面的代码示例来使用这两个方法生成Stream流。

- 迭代

```
Stream<Integer> intStream = Stream.iterate(0, (x) -> x + 2);
intStream.forEach(System.out::println);
```

运行上述代码，会在终端一直输出偶数，这种操作会一直持续下去。如果我们只需要输出10个偶数，该如何操作呢？其实也很简单，使用Stream对象的limit方法进行限制就可以了，如下所示。

```
Stream<Integer> intStream = Stream.iterate(0, (x) -> x + 2);
intStream.limit(10).forEach(System.out::println);
```

- 生成

```
Stream.generate(() -> Math.random()).forEach(System.out::println);
```

上述代码同样会一直输出随机数，如果我们只需要输出5个随机数，则只需要使用limit()方法进行限制即可。

```
Stream.generate(() -> Math.random()).limit(5).forEach(System.out::println);
```

## (5) 创建空流

在Stream类中提供了一个empty()方法，如下所示。

```
public static<T> Stream<T> empty() {
    return StreamSupport.stream(Spliterators.<T>emptySplititerator(), false);
}
```

我们可以使用Stream类的empty()方法来创建一个空Stream流，如下所示。

```
Stream<String> empty = Stream.empty();
```

# Stream的中间操作

多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行任何的处理！而在终止操作时一次性全部处理，称为“惰性求值”。Stream的中间操作是不会有结果数据输出的。

Stream的中间操作在整体上可以分为：筛选与切片、映射、排序。接下来，我们就分别对这些中间操作进行简要的说明。

## 筛选与切片

这里，我将与筛选和切片有关的操作整理成如下表格。

方法	描述
filter(Predicate p)	接收Lambda表达式，从流中排除某些元素
distinct()	筛选，通过流所生成元素的 hashCode() 和 equals() 去除重复元素
limit(long maxSize)	截断流，使其元素不超过给定数量
skip(long n)	跳过元素，返回一个扔掉了前 n 个元素的流。若流中元素不足 n 个，则返回一个空流。与 limit(n) 互补

接下来，我们列举几个简单的示例，以便加深理解。

为了更好的测试程序，我先构造了一个对象数组，如下所示。

```
protected List<Employee> list = Arrays.asList(
    new Employee("张三", 18, 9999.99),
    new Employee("李四", 38, 5555.55),
    new Employee("王五", 60, 6666.66),
    new Employee("赵六", 8, 7777.77),
    new Employee("田七", 58, 3333.33)
);
```

其中，Employee类的定义如下所示。

```
@Data
@Builder
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class Employee implements Serializable {
    private static final long serialVersionUID = -9079722457749166858L;
    private String name;
    private Integer age;
    private Double salary;
}
```

Employee类的定义比较简单，这里，我就不赘述了。之后的示例中，我们都是使用的Employee对象的集合进行操作。好了，我们开始具体的操作案例。

## 1.filter()方法

filter()方法主要是用于接收Lambda表达式，从流中排除某些元素，其在Stream接口中的源码如下所示。

```
Stream<T> filter(Predicate<? super T> predicate);
```

可以看到，在filter()方法中，需要传递Predicate接口的对象，Predicate接口又是个什么鬼呢？点进去看下源码。

```
@FunctionalInterface
public interface Predicate<T> {
```



```

boolean test(T t);

default Predicate<T> and(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) && other.test(t);
}

default Predicate<T> negate() {
    return (t) -> !test(t);
}

default Predicate<T> or(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) || other.test(t);
}

static <T> Predicate<T> isEqual(Object targetRef) {
    return (null == targetRef)
        ? Objects::isNull
        : object -> targetRef.equals(object);
}
}

```

可以看到，Predicate是一个函数式接口，其中接口中定义的主要方法为test()方法，test()方法会接收一个泛型对象t，返回一个boolean类型的数据。

看到这里，相信大家明白了：filter()方法是根据Predicate接口的test()方法的返回结果来过滤数据的，如果test()方法的返回结果为true，符合规则；如果test()方法的返回结果为false，则不符合规则。

这里，我们可以使用下面的示例来简单的说明filter()方法的使用方式。

```

//内部迭代：在此过程中没有进行过迭代，由Stream api进行迭代
//中间操作：不会执行任何操作
Stream<Person> stream = list.stream().filter((e) -> {
    System.out.println("Stream API 中间操作");
    return e.getAge() > 30;
});

```

我们，在执行终止语句之后，一边迭代，一边打印，而我们并没有去迭代上面集合，其实这是内部迭代，由Stream API 完成。

下面我们来看看外部迭代，也就是我们人为得迭代。

```

//外部迭代
Iterator<Person> it = list.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}

```

## 2.limit()方法

主要作用为：截断流，使其元素不超过给定数量。

先来看limit方法的定义，如下所示。

```

Stream<T> limit(long maxSize);

```



limit()方法在Stream接口中的定义比较简单，只需要传入一个long类型的数字即可。

我们可以按照如下所示的代码来使用limit()方法。

```
//过滤之后取2个值
list.stream().filter((e) -> e.getAge() >30 ).limit(2).forEach(System.out ::
println);
```

在这里，我们可以配合其他得中间操作，并截断流，使我们可以取得相应个数得元素。而且在上面计算中，只要发现有2条符合条件得元素，则不会继续往下迭代数据，可以提高效率。

### 3.skip()方法

跳过元素，返回一个扔掉了前 n 个元素的流。若流中元素 不足 n 个，则返回一个空流。与 limit(n) 互补。

源码定义如下所示。

```
Stream<T> skip(long n);
```

源码定义比较简单，同样只需要传入一个long类型的数字即可。其含义是跳过n个元素。

简单示例如下所示。

```
//跳过前2个值
list.stream().skip(2).forEach(System.out :: println);
```

### 4.distinct()方法

筛选，通过流所生成元素的 hashCode() 和 equals() 去 除重复元素。

源码定义如下所示。

```
Stream<T> distinct();
```

旨在对流中的元素进行去重。

我们可以如下面的方式来使用disinct()方法。

```
list.stream().distinct().forEach(System.out :: println);
```

这里有一个需要注意的地方：distinct 需要实体中重写hashCode () 和 equals () 方法才可以使用。

## 映射

关于映射相关的方法如下表所示。

方法	描述
map(Function f)	接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
mapToDouble(ToDoubleFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 DoubleStream。
mapToInt(ToIntFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 IntStream。
mapToLong(ToLongFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 LongStream。
flatMap(Function f)	接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。

## 1.map()方法

接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。

先来看Java8中Stream接口对于map()方法的声明，如下所示。

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

我们可以按照如下方式使用map()方法。

```
//将流中每一个元素都映射到map的函数中，每个元素执行这个函数，再返回
List<String> list = Arrays.asList("aaa", "bbb", "ccc", "ddd");
list.stream().map((e) -> e.toUpperCase()).forEach(System.out::printf);

//获取Person中的每一个人得名字name，再返回一个集合
List<String> names = this.list.stream().map(Person ::
getName).collect(Collectors.toList());
```

## 2.flatMap()

接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。

先来看Java8中Stream接口对于flatMap()方法的声明，如下所示。

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>>
mapper);
```

我们可以使用如下方式使用flatMap()方法，为了便于大家理解，这里，我就贴出了测试flatMap()方法的所有代码。

```

/**
 * flatMap — 接收一个函数作为参数，将流中的每个值都换成一个流，然后把所有流连接成一个流
 */
@Test
public void testFlatMap () {
    StreamAPI_Test s = new StreamAPI_Test();
    List<String> list = Arrays.asList("aaa", "bbb", "ccc", "ddd");
    list.stream().flatMap((e) ->
s.filterCharacter(e)).forEach(System.out::println);

    //如果使用map则需要这样写
    list.stream().map((e) -> s.filterCharacter(e)).forEach((e) -> {
        e.forEach(System.out::println);
    });
}

/**
 * 将一个字符串转换为流
 */
public Stream<Character> filterCharacter(String str){
    List<Character> list = new ArrayList<>();
    for (Character ch : str.toCharArray()) {
        list.add(ch);
    }
    return list.stream();
}

```

其实map方法就相当于Collection的add方法，如果add的是个集合的话就会变成二维数组，而flatMap的话就相当于Collection的addAll方法，参数如果是集合的话，只是将2个集合合并，而不是变成二维数组。

## 排序

关于排序相关的方法如下表所示。

方法	描述
sorted()	产生一个新流，其中按自然顺序排序
sorted(Comparator comp)	产生一个新流，其中按比较器顺序排序

从上述表格可以看出：sorted有两种方法，一种是不传任何参数，叫自然排序，还有一种需要传Comparator 接口参数，叫做定制排序。

先来看Java8中Stream接口对于sorted()方法的声明，如下所示。

```

Stream<T> sorted();
Stream<T> sorted(Comparator<? super T> comparator);

```

sorted()方法的定义比较简单，我就不再赘述了。

我们也可以按照如下方式来使用Stream的sorted()方法。

```
// 自然排序
List<Employee> persons = list.stream().sorted().collect(Collectors.toList());

//定制排序
List<Employee> persons1 = list.stream().sorted((e1, e2) -> {
    if (e1.getAge() == e2.getAge()) {
        return 0;
    } else if (e1.getAge() > e2.getAge()) {
        return 1;
    } else {
        return -1;
    }
}).collect(Collectors.toList());
```

## Stream的终止操作

终端操作会从流的流水线生成结果。其结果可以是任何不是流的值，例如：List、Integer、Double、String等等，甚至是void。

在Java8中，Stream的终止操作可以分为：查找与匹配、规约和收集。接下来，我们就分别简单说明下这些终止操作。

## 查找与匹配

Stream API中有关查找与匹配的方法如下表所示。

方法	描述
allMatch(Predicate p)	检查是否匹配所有元素
anyMatch(Predicate p)	检查是否至少匹配一个元素
noneMatch(Predicate p)	检查是否没有匹配所有元素
findFirst()	返回第一个元素
findAny()	返回当前流中的任意元素
count()	返回流中元素总数
max(Comparator c)	返回流中最大值
min(Comparator c)	返回流中最小值
forEach(Consumer c)	内部迭代(使用 Collection 接口需要用户去做迭代，称为外部迭代。相反，Stream API 使用内部迭代)

同样的，我们对每个重要的方法进行简单的示例说明，这里，我们首先建立一个Employee类，Employee类的定义如下所示。

```
@Data
@Builder
@ToString
@NoArgsConstructor
```

```
@AllArgsConstructor
public class Employee implements Serializable {
    private static final long serialVersionUID = -9079722457749166858L;
    private String name;
    private Integer age;
    private Double salary;
    private Stauts stauts;
    public enum Stauts{
        WORKING,
        SLEEPING,
        VOCATION
    }
}
```

接下来，我们在测试类中定义一个用于测试的集合employees，如下所示。

```
protected List<Employee> employees = Arrays.asList(
    new Employee("张三", 18, 9999.99, Employee.Stauts.SLEEPING),
    new Employee("李四", 38, 5555.55, Employee.Stauts.WORKING),
    new Employee("王五", 60, 6666.66, Employee.Stauts.WORKING),
    new Employee("赵六", 8, 7777.77, Employee.Stauts.SLEEPING),
    new Employee("田七", 58, 3333.33, Employee.Stauts.VOCATION)
);
```

好了，准备工作就绪了。接下来，我们就开始测试Stream的每个终止方法。

## 1.allMatch()

allMatch()方法表示检查是否匹配所有元素。其在Stream接口中的定义如下所示。

```
boolean allMatch(Predicate<? super T> predicate);
```

我们可以通过类似如下示例来使用allMatch()方法。

```
boolean match = employees.stream().allMatch((e) ->
    Employee.Stauts.SLEEPING.equals(e.getStauts()));
System.out.println(match);
```

**注意：**使用allMatch()方法时，只有所有的元素都匹配条件时，allMatch()方法才会返回true。

## 2.anyMatch()方法

anyMatch方法表示检查是否至少匹配一个元素。其在Stream接口中的定义如下所示。

```
boolean anyMatch(Predicate<? super T> predicate);
```

我们可以通过类似如下示例来使用anyMatch()方法。

```
boolean match = employees.stream().anyMatch((e) ->
    Employee.Stauts.SLEEPING.equals(e.getStauts()));
System.out.println(match);
```

**注意：**使用anyMatch()方法时，只要有任意一个元素符合条件，anyMatch()方法就会返回true。

### 3.noneMatch()方法

noneMatch()方法表示检查是否没有匹配所有元素。其在Stream接口中的定义如下所示。

```
boolean noneMatch(Predicate<? super T> predicate);
```

我们可以通过类似如下示例来使用noneMatch()方法。

```
boolean match = employees.stream().noneMatch((e) ->
Employee.Stauts.SLEEPING.equals(e.getStauts()));
System.out.println(match);
```

**注意：**使用noneMatch()方法时，只有所有的元素都不符合条件时，noneMatch()方法才会返回true。

### 4.findFirst()方法

findFirst()方法表示返回第一个元素。其在Stream接口中的定义如下所示。

```
Optional<T> findFirst();
```

我们可以通过类似如下示例来使用findFirst()方法。

```
Optional<Employee> op = employees.stream().sorted((e1, e2) ->
Double.compare(e1.getSalary(), e2.getSalary())).findFirst();
System.out.println(op.get());
```

### 5.findAny()方法

findAny()方法表示返回当前流中的任意元素。其在Stream接口中的定义如下所示。

```
Optional<T> findAny();
```

我们可以通过类似如下示例来使用findAny()方法。

```
Optional<Employee> op = employees.stream().filter((e) ->
Employee.Stauts.WORKING.equals(e.getStauts())).findFirst();
System.out.println(op.get());
```

### 6.count()方法

count()方法表示返回流中元素总数。其在Stream接口中的定义如下所示。

```
long count();
```

我们可以通过类似如下示例来使用count()方法。

```
long count = employees.stream().count();
System.out.println(count);
```

### 7.max()方法

max()方法表示返回流中最大值。其在Stream接口中的定义如下所示。

```
Optional<T> max(Comparator<? super T> comparator);
```

我们可以通过类似如下示例来使用max()方法。

```
Optional<Employee> op = employees.stream().max((e1, e2) ->  
Double.compare(e1.getSalary(), e2.getSalary()));  
System.out.println(op.get());
```

## 8.min()方法

min()方法表示返回流中最小值。其在Stream接口中的定义如下所示。

```
Optional<T> min(Comparator<? super T> comparator);
```

我们可以通过类似如下示例来使用min()方法。

```
Optional<Double> op =  
employees.stream().map(Employee::getSalary).min(Double::compare);  
System.out.println(op.get());
```

## 9.forEach()方法

forEach()方法表示内部迭代(使用 Collection 接口需要用户去做迭代, 称为外部迭代。相反, Stream API 使用内部迭代)。其在Stream接口内部的定义如下所示。

```
void forEach(Consumer<? super T> action);
```

我们可以通过类似如下示例来使用forEach()方法。

```
employees.stream().forEach(System.out::println);
```

## 规约

Stream API中有关规约的方法如下表所示。

方法	描述
reduce(T identity, BinaryOperator b)	可以将流中元素反复结合起来, 得到一个值。返回 T
reduce(BinaryOperator b)	可以将流中元素反复结合起来, 得到一个值。返回 Optional

reduce()方法在Stream接口中的定义如下所示。

```
T reduce(T identity, BinaryOperator<T> accumulator);  
Optional<T> reduce(BinaryOperator<T> accumulator);  
<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator,  
BinaryOperator<U> combiner);
```

我们可以通过类似如下示例来使用reduce方法。

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
Integer sum = list.stream().reduce(0, (x, y) -> x + y);
System.out.println(sum);
System.out.println("-----");
Optional<Double> op =
employees.stream().map(Employee::getSalary).reduce(Double::sum);
System.out.println(op.get());
```

我们也可以搜索employees列表中“张”出现的次数。

```
Optional<Integer> sum = employees.stream()
    .map(Employee::getName)
    .flatMap(TestStreamAPI1::filterCharacter)
    .map((ch) -> {
        if(ch.equals('六'))
            return 1;
        else
            return 0;
    }).reduce(Integer::sum);
System.out.println(sum.get());
```

**注意：**上述例子使用了硬编码的方式来累加某个具体值，大家在实际工作中再优化代码。

## 收集

方法	描述
collect(Collector c)	将流转换为其他形式。接收一个 Collector接口的实现，用于给Stream中元素做汇总的方法

collect()方法在Stream接口中的定义如下所示。

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);

<R, A> R collect(Collector<? super T, A, R> collector);
```

我们可以通过类似如下示例来使用collect方法。

```
Optional<Double> max = employees.stream()
    .map(Employee::getSalary)
    .collect(Collectors.maxBy(Double::compare));
System.out.println(max.get());
Optional<Employee> op = employees.stream()
    .collect(Collectors.minBy((e1, e2) -> Double.compare(e1.getSalary(),
e2.getSalary())));
System.out.println(op.get());
Double sum =
employees.stream().collect(Collectors.summingDouble(Employee::getSalary));
System.out.println(sum);
```



```
Double avg =
employees.stream().collect(Collectors.averagingDouble(Employee::getSalary));
System.out.println(avg);
Long count = employees.stream().collect(Collectors.counting());
System.out.println(count);
System.out.println("-----");
DoubleSummaryStatistics dss = employees.stream()
    .collect(Collectors.summarizingDouble(Employee::getSalary));
System.out.println(dss.getMax());
```

## 如何收集Stream流？

Collector接口中方法的实现决定了如何对流执行收集操作(如收集到 List、Set、Map)。Collectors实用类提供了很多静态方法，可以方便地创建常见收集器实例，具体方法与实例如下表：

方法	返回类型	作用
toList	List	把流中元素收集到List
toSet	Set	把流中元素收集到Set
toCollection	Collection	把流中元素收集到创建的集合
counting	Long	计算流中元素的个数
summingInt	Integer	对流中元素的整数属性求和
averagingInt	Double	计算流中元素Integer属性的平均 值
summarizingInt	IntSummaryStatistics	收集流中Integer属性的统计值。 如：平均值
joining	String	连接流中每个字符串
maxBy	Optional	根据比较器选择最大值

方法	返回类型	作用
minBy	Optional	根据比较器选择最小值
reducing	归约产生的类型	从一个作为累加器的初始值 开始，利用 BinaryOperator与 流中元素逐个结合，从而归约成单个值
collectingAndThen	转换函数返回的类型	包裹另一个收集器，对其结果转换函数
groupingBy	Map<K, List>	根据某属性值对流分组，属性为K，结果为V
partitioningBy	Map<Boolean, List>	根据true或false进行分区

每个方法对应的使用示例如下表所示。

方法	使用示例
toList	List employees= list.stream().collect(Collectors.toList());
toSet	Set employees= list.stream().collect(Collectors.toSet());
toCollection	Collection employees=list.stream().collect(Collectors.toCollection(ArrayList::new));
counting	long count = list.stream().collect(Collectors.counting());
summingInt	int total=list.stream().collect(Collectors.summingInt(Employee::getSalary));
averagingInt	double avg= list.stream().collect(Collectors.averagingInt(Employee::getSalary))
summarizingInt	IntSummaryStatistics iss= list.stream().collect(Collectors.summarizingInt(Employee::getSalary));
Collectors	String str= list.stream().map(Employee::getName).collect(Collectors.joining());
maxBy	Optionalmax= list.stream().collect(Collectors.maxBy(comparingInt(Employee::getSalary)));
minBy	Optional min = list.stream().collect(Collectors.minBy(comparingInt(Employee::getSalary)));
reducing	int total=list.stream().collect(Collectors.reducing(0, Employee::getSalar, Integer::sum));
collectingAndThen	int how= list.stream().collect(Collectors.collectingAndThen(Collectors.toList(), List::size));
groupingBy	Map<Emp.Status, List> map= list.stream() .collect(Collectors.groupingBy(Employee::getStatus));
partitioningBy	Map<Boolean,List>vd= list.stream().collect(Collectors.partitioningBy(Employee::getManage));

```
public void test4(){
    Optional<Double> max = emps.stream()
        .map(Employee::getSalary)
        .collect(Collectors.maxBy(Double::compare));
    System.out.println(max.get());

    Optional<Employee> op = emps.stream()
        .collect(Collectors.minBy((e1, e2) -> Double.compare(e1.getSalary(),
e2.getSalary())));

    System.out.println(op.get());
}
```

```

Double sum = emps.stream()
    .collect(Collectors.summingDouble(Employee::getSalary));

System.out.println(sum);

Double avg = emps.stream()
    .collect(Collectors.averagingDouble(Employee::getSalary));
System.out.println(avg);

Long count = emps.stream()
    .collect(Collectors.counting());

DoubleSummaryStatistics dss = emps.stream()
    .collect(Collectors.summarizingDouble(Employee::getSalary));
System.out.println(dss.getMax());

```

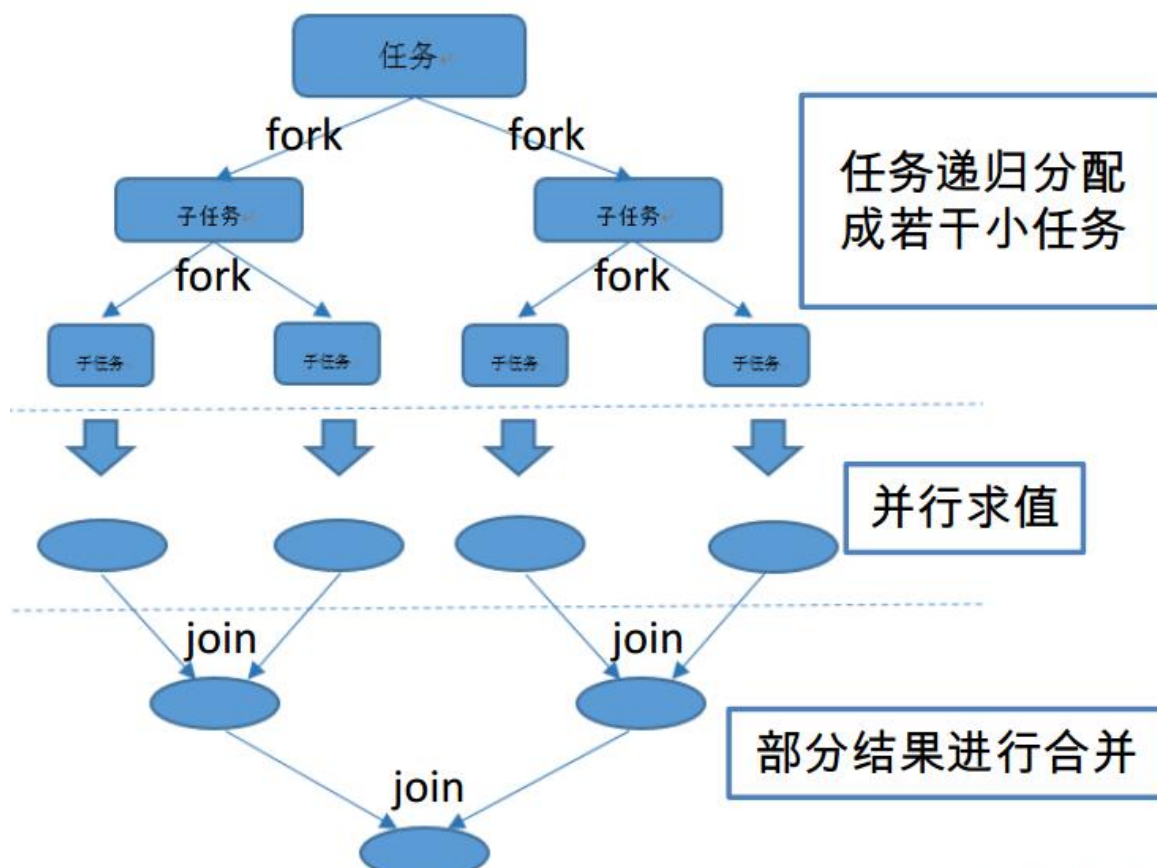
## 什么是并行流？

简单来说，并行流就是把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流。

Java 8 中将并行进行了优化，我们可以很容易的对数据进行并行操作。Stream API 可以声明性地通过 `parallel()` 与 `sequential()` 在并行流与顺序流之间进行切换。

## Fork/Join 框架

Fork/Join 框架：就是在必要的情况下，将一个大任务，进行拆分(fork)成若干个小任务（拆到不可再拆时），再将一个个的小任务运算的结果进行 join 汇总。



# Fork/Join 框架与传统线程池有啥区别？

采用“工作窃取”模式（work-stealing）：

当执行新的任务时它可以将其拆分成更小的任务执行，并将小任务加到线程队列中，然后再从一个随机线程的队列中偷一个并把它放在自己的队列中。

相对于一般的线程池实现，fork/join框架的优势体现在对其中包含的任务的处理方式上。在一般的线程池中，如果一个线程正在执行的任务由于某些原因无法继续运行，那么该线程会处于等待状态。而在fork/join框架的实现中，如果某个子任务由于等待另外一个子任务的完成而无法继续运行。那么处理该子问题的线程会主动寻找其他尚未运行的子任务来执行。这种方式减少了线程的等待时间，提高了程序的性能。

## Fork/Join框架实例

了解了ForkJoin框架的原理之后，我们就来手动写一个使用Fork/Join框架实现累加和的示例程序，以帮助读者更好的理解Fork/Join框架。好了，不废话了，上代码，大家通过下面的代码好好体会下Fork/Join框架的强大。

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
import java.util.concurrent.RecursiveTask;
@Slf4j
public class ForkJoinTaskExample extends RecursiveTask<Integer> {
    public static final int threshold = 2;
    private int start;
    private int end;
    public ForkJoinTaskExample(int start, int end) {
        this.start = start;
        this.end = end;
    }
    @Override
    protected Integer compute() {
        int sum = 0;
        //如果任务足够小就计算任务
        boolean canCompute = (end - start) <= threshold;
        if (canCompute) {
            for (int i = start; i <= end; i++) {
                sum += i;
            }
        } else {
            // 如果任务大于阈值，就分裂成两个子任务计算
            int middle = (start + end) / 2;
            ForkJoinTaskExample leftTask = new ForkJoinTaskExample(start,
middle);
            ForkJoinTaskExample rightTask = new ForkJoinTaskExample(middle + 1,
end);

            // 执行子任务
            leftTask.fork();
            rightTask.fork();

            // 等待任务执行结束合并其结果
```

```

        int leftResult = leftTask.join();
        int rightResult = rightTask.join();

        // 合并子任务
        sum = leftResult + rightResult;
    }
    return sum;
}

public static void main(String[] args) {
    ForkJoinPool forkJoinPool = new ForkJoinPool();

    //生成一个计算任务，计算1+2+3+4
    ForkJoinTaskExample task = new ForkJoinTaskExample(1, 100);

    //执行一个任务
    Future<Integer> result = forkJoinPool.submit(task);

    try {
        log.info("result:{}", result.get());
    } catch (Exception e) {
        log.error("exception", e);
    }
}
}

```

## Java8中的并行流实例

Java8对并行流进行了大量的优化，并且在开发上也极大的简化了程序员的工作量，我们只需要使用类似如下的代码就可以使用Java8中的并行流来处理我们的数据。

```
LongStream.rangeClosed(0, 10000000L).parallel().reduce(0, Long::sum);
```

### 在Java8中如何优雅的切换并行流和串行流呢？

Stream API 可以声明性地通过 parallel() 与sequential() 在并行流与串行流之间进行切换。

## 什么是Optional类？

Optional 类(java.util.Optional) 是一个容器类，代表一个值存在或不存在，原来用 null 表示一个值不存在，现在 Optional 可以更好的表达这个概念。并且可以避免空指针异常。

### Optional类常用方法：

- Optional.of(T t) : 创建一个 Optional 实例。
- Optional.empty() : 创建一个空的 Optional 实例。
- Optional.ofNullable(T t):若 t 不为 null,创建 Optional 实例,否则创建空实例。
- isPresent() : 判断是否包含值。
- orElse(T t) : 如果调用对象包含值，返回该值，否则返回t。
- orElseGet(Supplier s) :如果调用对象包含值，返回该值，否则返回 s 获取的值。
- map(Function f): 如果有值对其处理，并返回处理后的Optional，否则返回 Optional.empty()。
- flatMap(Function mapper):与 map 类似，要求返回值必须是Optional。

## Optional类示例

### 1.创建Optional类

(1) 使用empty()方法创建一个空的Optional对象:

```
Optional<String> empty = Optional.empty();
```

(2) 使用of()方法创建Optional对象:

```
String name = "binghe";
Optional<String> opt = Optional.of(name);
assertEquals("Optional[binghe]", opt.toString());
```

传递给of()的值不可以为空, 否则会抛出空指针异常。例如, 下面的程序会抛出空指针异常。

```
String name = null;
Optional<String> opt = Optional.of(name);
```

如果我们需要传递一些空值, 那我们可以使用下面的示例所示。

```
String name = null;
Optional<String> opt = Optional.ofNullable(name);
```

使用ofNullable()方法, 则当传递进去一个空值时, 不会抛出异常, 而只是返回一个空的Optional对象, 如同我们用Optional.empty()方法一样。

## 2.isPresent

我们可以使用这个isPresent()方法检查一个Optional对象中是否有值, 只有值非空才返回true。

```
Optional<String> opt = Optional.of("binghe");
assertTrue(opt.isPresent());

opt = Optional.ofNullable(null);
assertFalse(opt.isPresent());
```

在Java8之前, 我们一般使用如下方式来检查空值。

```
if(name != null){
    System.out.println(name.length);
}
```

在Java8中, 我们就可以使用如下方式来检查空值了。

```
Optional<String> opt = Optional.of("binghe");
opt.ifPresent(name -> System.out.println(name.length()));
```

## 3.orElse和orElseGet

(1) orElse

orElse()方法用来返回Optional对象中的默认值, 它被传入一个“默认参数”。如果对象中存在一个值, 则返回它, 否则返回传入的“默认参数”。

```
String nullName = null;
String name = Optional.ofNullable(nullName).orElse("binghe");
assertEquals("binghe", name);
```

## (2) orElseGet

与orElse()方法类似，但是这个函数不接收一个“默认参数”，而是一个函数接口。

```
String nullName = null;
String name = Optional.ofNullable(nullName).orElseGet(() -> "binghe");
assertEquals("binghe", name);
```

## (3) 二者有什么区别？

要想理解二者的区别，首先让我们创建一个无参且返回定值的方法。

```
public String getDefaultName() {
    System.out.println("Getting Default Name");
    return "binghe";
}
```

接下来，进行两个测试看看两个方法到底有什么区别。

```
String text;
System.out.println("Using orElseGet:");
String defaultText = Optional.ofNullable(text).orElseGet(this::getDefaultName);
assertEquals("binghe", defaultText);

System.out.println("Using orElse:");
defaultText = Optional.ofNullable(text).orElse(getDefaultName());
assertEquals("binghe", defaultText);
```

在这里示例中，我们的Optional对象中包含的都是一个空值，让我们看看程序执行结果：

```
Using orElseGet:
Getting default name...
Using orElse:
Getting default name...
```

两个Optional对象中都不存在value，因此执行结果相同。

那么，当Optional对象中存在数据会发生什么呢？我们一起来验证下。

```
String name = "binghe001";

System.out.println("Using orElseGet:");
String defaultName = Optional.ofNullable(name).orElseGet(this::getDefaultName);
assertEquals("binghe001", defaultName);

System.out.println("Using orElse:");
defaultName = Optional.ofNullable(name).orElse(getDefaultName());
assertEquals("binghe001", defaultName);
```

运行结果如下所示。

```
Using orElseGet:  
Using orElse:  
Getting default name...
```

可以看到，当使用`orElseGet()`方法时，`getDefaultName()`方法并不执行，因为`Optional`中含有值，而使用`orElse`时则照常执行。所以可以看到，当值存在时，`orElse`相比于`orElseGet`，多创建了一个对象。如果创建对象时，存在网络交互，那系统资源的开销就比较大了，这是需要我们注意的一个地方。

## 4.orElseThrow

`orElseThrow()`方法当遇到一个不存在的值的时候，并不返回一个默认值，而是抛出异常。

```
String nullName = null;  
String name = Optional.ofNullable(nullName).orElseThrow(  
    IllegalArgumentException::new);
```

## 5.get

`get()`方法表示是`Optional`对象中获取值。

```
Optional<String> opt = Optional.of("binghe");  
String name = opt.get();  
assertEquals("binghe", name);
```

使用`get()`方法也可以返回被包裹着的值。但是值必须存在。当值不存在时，会抛出一个`NoSuchElementException`异常。

```
Optional<String> opt = Optional.ofNullable(null);  
String name = opt.get();
```

## 6.filter

接收一个函数式接口，当符合接口时，则返回一个`Optional`对象，否则返回一个空的`Optional`对象。

```
String name = "binghe";  
Optional<String> nameOptional = Optional.of(name);  
boolean isBinghe = nameOptional.filter(n -> "binghe".equals(name)).isPresent();  
assertTrue(isBinghe);  
boolean isBinghe001 = nameOptional.filter(n ->  
    "binghe001".equals(name)).isPresent();  
assertFalse(isBinghe001);
```

使用`filter()`方法会过滤掉我们不需要的元素。

接下来，我们再来看一例示例，例如目前有一个`Person`类，如下所示。

```
public class Person{  
    private int age;  
    public Person(int age){  
        this.age = age;  
    }  
    //省略get set方法  
}
```



例如，我们需要过滤出年龄在25岁到35岁之前的人群，那在Java8之前我们需要创建一个如下的方法来检测每个人的年龄范围是否在25岁到35岁之前。

```
public boolean filterPerson(Peron person){
    boolean isInRange = false;
    if(person != null && person.getAge() >= 25 && person.getAge() <= 35){
        isInRange = true;
    }
    return isInRange;
}
```

看上去就挺麻烦的，我们可以使用如下的方式进行测试。

```
assertTrue(filterPerson(new Peron(18)));
assertFalse(filterPerson(new Peron(29)));
assertFalse(filterPerson(new Peron(16)));
assertFalse(filterPerson(new Peron(34)));
assertFalse(filterPerson(null));
```

如果使用Optional，效果如何呢？

```
public boolean filterPersonByOptional(Peron person){
    return Optional.ofNullable(person)
        .map(Peron::getAge)
        .filter(p -> p >= 25)
        .filter(p -> p <= 35)
        .isPresent();
}
```

使用Optional看上去就清爽多了，这里，map()仅仅是将一个值转换为另一个值，并且这个操作并不会改变原来的值。

## 7.map

如果有值对其处理，并返回处理后的Optional，否则返回 Optional.empty()。

```
List<String> names = Arrays.asList("binghe001", "binghe002", "", "binghe003",
    "", "binghe004");
Optional<List<String>> listOptional = Optional.of(names);

int size = listOptional
    .map(List::size)
    .orElse(0);
assertEquals(6, size);
```

在这个例子中，我们使用一个List集合封装了一些字符串，然后再把这个List使用Optional封装起来，对其map()，获取List集合的长度。map()返回的结果也被封装在一个Optional对象中，这里当值不存在的时候，我们会默认返回0。如下我们获取一个字符串的长度。

```
String name = "binghe";
Optional<String> nameOptional = Optional.of(name);

int len = nameOptional
    .map(String::length)
    .orElse(0);
assertEquals(6, len);
```

我们也可以将map()方法与filter()方法结合使用，如下所示。

```
String password = " password ";
Optional<String> passOpt = Optional.of(password);
boolean correctPassword = passOpt.filter(
    pass -> pass.equals("password")).isPresent();
assertFalse(correctPassword);

correctPassword = passOpt
    .map(String::trim)
    .filter(pass -> pass.equals("password"))
    .isPresent();
assertTrue(correctPassword);
```

上述代码的含义就是对密码进行验证，查看密码是否为指定的值。

## 8.flatMap

与 map 类似，要求返回值必须是Optional。

假设我们现在有一个Person类。

```
public class Person {
    private String name;
    private int age;
    private String password;

    public Optional<String> getName() {
        return Optional.ofNullable(name);
    }

    public Optional<Integer> getAge() {
        return Optional.ofNullable(age);
    }

    public Optional<String> getPassword() {
        return Optional.ofNullable(password);
    }
    // 忽略get set方法
}
```

接下来，我们可以将Person封装到Optional中，并进行测试，如下所示。

```

Person person = new Person("binghe", 18);
Optional<Person> personOptional = Optional.of(person);

Optional<Optional<String>> nameOptionalWrapper =
    personOptional.map(Person::getName);
Optional<String> nameOptional =
    nameOptionalWrapper.orElseThrow(IllegalArgumentException::new);
String name1 = nameOptional.orElse("");
assertEquals("binghe", name1);

String name = personOptional
    .flatMap(Person::getName)
    .orElse("");
assertEquals("binghe", name);

```

注意：方法getName返回的是一个Optional对象，如果使用map，我们还需要再调用一次get()方法，而使用flatMap()就不需要了。

## 接口中的默认方法

Java 8中允许接口中包含具有具体实现的方法，该方法称为“默认方法”，默认方法使用 default 关键字修饰。

例如，我们可以定义一个接口MyFunction，其中，包含有一个默认方法getName，如下所示。

```

public interface MyFunction<T>{
    T get(Long id);
    default String getName(){
        return "binghe";
    }
}

```

## 默认方法的原则

在Java8中，默认方法具有“类优先”的原则。

若一个接口中定义了一个默认方法，而另外一个父类或接口中又定义了一个同名的方法时，遵循如下的原则。

**1.选择父类中的方法。**如果一个父类提供了具体的实现，那么接口中具有相同名称和参数的默认方法会被忽略。

例如，现在有一个接口为MyFunction，和一个类MyClass，如下所示。

- MyFunction接口

```

public interface MyFunction{
    default String getName(){
        return "MyFunction";
    }
}

```

- MyClass类

```
public class MyClass{
    public String getName(){
        return "MyClass";
    }
}
```

此时，创建SubClass类继承MyClass类，并实现MyFunction接口，如下所示。

```
public class SubClass extends MyClass implements MyFunction{

}
```

接下来，我们创建一个SubClassTest类，对SubClass类进行测试，如下所示。

```
public class SubClassTest{
    @Test
    public void testDefaultFunction(){
        SubClass subClass = new SubClass();
        System.out.println(subClass.getName());
    }
}
```

运行上述程序，会输出字符串：MyClass。

**2.接口冲突。如果一个父接口提供一个默认方法，而另一个接口也提供了一个具有相同名称和参数列表的方法（不管方法是否是默认方法），那么必须覆盖该方法来解决冲突。**

例如，现在有两个接口，分别为MyFunction和MyInterface，各自都有一个默认方法getName()，如下所示。

- MyFunction接口

```
public interface MyFunction{
    default String getName(){
        return "function";
    }
}
```

- MyInterface接口

```
public interface MyInterface{
    default String getName(){
        return "interface";
    }
}
```

实现类MyClass同时实现了MyFunction接口和MyInterface接口，由于MyFunction接口和MyInterface接口中都存在getName()默认方法，所以，MyClass必须覆盖getName()方法来解决冲突，如下所示。

```
public class MyClass{
    @Override
    public String getName(){
        return MyInterface.super.getName();
    }
}
```

此时，MyClass类中的getName方法返回的是：interface。

如果MyClass中的getName()方法覆盖的是MyFunction接口的getName()方法，如下所示。

```
public class MyClass{
    @Override
    public String getName(){
        return MyFunction.super.getName();
    }
}
```

此时，MyClass类中的getName方法返回的是：function。

## 接口中的静态方法

在Java8中，接口中允许添加静态方法，使用方式接口名.方法名。例如MyFunction接口中定义了静态方法send()。

```
public interface MyFunction{
    default String getName(){
        return "binghe";
    }
    static void send(){
        System.out.println("Send Message...");
    }
}
```

我们可以直接使用如下方式调用MyFunction接口的send静态方法。

```
MyFunction.send();
```

## 本地时间和时间戳

**主要方法：**

- now：静态方法，根据当前时间创建对象
- of：静态方法，根据指定日期/时间创建对象
- plusDays,plusWeeks,plusMonths,plusYears：向当前LocalDate 对象添加几天、几周、几个月、几年
- minusDays,minuWeeks,minuMonths,minuYears：从当前LocalDate 对象减去几天、几周、几个月、几年
- plus,minus：添加或减少一个Duration 或Period
- withDayOfMonth,withDayOfYear,withMonth,withYear：将月份天数、年份天数、月份、年份修改为指定的值并返回新的LocalDate 对象
- getDayOfYear：获得年份天数(1~366)
- getDayOfWeek：获得星期几(返回一个DayOfWeek枚举值)

- `getMonth`: 获得月份, 返回一个 `Month` 枚举值
- `getMonthValue`: 获得月份(1~12)
- `getYear`: 获得年份
- `until`: 获得两个日期之间的 `Period` 对象, 或者指定 `ChronoUnits` 的数字
- `isBefore`, `isAfter`: 比较两个 `LocalDate`
- `isLeapYear`: 判断是否是闰年

## 使用 `LocalDate`、`LocalTime`、`LocalDateTime`

`LocalDate`、`LocalTime`、`LocalDateTime` 类的实例是不可变的对象, 分别表示使用 ISO-8601 日历系统的日期、时间、日期和时间。它们提供了简单的日期或时间, 并不包含当前的时间信息。也不包含与时区相关的信息。

注: ISO-8601 日历系统是国际标准化组织制定的现代公民的日期和时间的表示法

方法	描述
<code>now()</code>	静态方法, 根据当前时间创建对象
<code>of()</code>	静态方法, 根据指定日期/时间创建 对象
<code>plusDays</code> , <code>plusWeeks</code> , <code>plusMonths</code> , <code>plusYears</code>	向当前 <code>LocalDate</code> 对象添加几天、几周、几个月、几年
<code>minusDays</code> , <code>minusWeeks</code> , <code>minusMonths</code> , <code>minusYears</code>	从当前 <code>LocalDate</code> 对象减去几天、几周、几个月、几年
<code>plus</code> , <code>minus</code>	添加或减少一个 <code>Duration</code> 或 <code>Period</code>
<code>withDayOfMonth</code> , <code>withDayOfYear</code> , <code>withMonth</code> , <code>withYear</code>	将月份天数、年份天数、月份、年份 修改为 指定的值 并 返回 新的 <code>LocalDate</code> 对象
<code>getDayOfMonth</code>	获得月份天数(1-31)
<code>getDayOfYear</code>	获得年份天数(1-366)
<code>getDayOfWeek</code>	获得星期几(返回一个 <code>DayOfWeek</code> 枚举值)
<code>getMonth</code>	获得月份, 返回一个 <code>Month</code> 枚举值
<code>getMonthValue</code>	获得月份(1-12)
<code>getYear</code>	获得年份
<code>until</code>	获得两个日期之间的 <code>Period</code> 对象, 或者指定 <code>ChronoUnits</code> 的数字
<code>isBefore</code> , <code>isAfter</code>	比较两个 <code>LocalDate</code>
<code>isLeapYear</code>	判断是否是闰年

示例代码如下所示。

```
// 获取当前系统时间
LocalDateTime localDateTime1 = LocalDateTime.now();
System.out.println(localDateTime1);
// 运行结果: 2019-10-27T13:49:09.483
```

```
// 指定日期时间
LocalDateTime localDateTime2 = LocalDateTime.of(2019, 10, 27, 13, 45, 10);
System.out.println(localDateTime2);
// 运行结果: 2019-10-27T13:45:10

LocalDateTime localDateTime3 = localDateTime1
    // 加三年
    .plusYears(3)
    // 减三个月
    .minusMonths(3);
System.out.println(localDateTime3);
// 运行结果: 2022-07-27T13:49:09.483

System.out.println(localDateTime1.getYear());           // 运行结果: 2019
System.out.println(localDateTime1.getMonthValue());    // 运行结果: 10
System.out.println(localDateTime1.getDayOfMonth());   // 运行结果: 27
System.out.println(localDateTime1.getHour());          // 运行结果: 13
System.out.println(localDateTime1.getMinute());        // 运行结果: 52
System.out.println(localDateTime1.getSecond());        // 运行结果: 6

LocalDateTime localDateTime4 = LocalDateTime.now();
System.out.println(localDateTime4);                   // 2019-10-27T14:19:56.884
LocalDateTime localDateTime5 = localDateTime4.withDayOfMonth(10);
System.out.println(localDateTime5);                   // 2019-10-10T14:19:56.884
```

## Instant 时间戳

用于“时间戳”的运算。它是以Unix元年(传统的设定为UTC时区1970年1月1日午夜时分)开始所经历的描述进行运算。

示例代码如下所示。

```
Instant instant1 = Instant.now();    // 默认获取UTC时区
System.out.println(instant1);
// 运行结果: 2019-10-27T05:59:58.221Z

// 偏移量运算
OffsetDateTime offsetDateTime = instant1.atOffset(ZoneOffset.ofHours(8));
System.out.println(offsetDateTime);
// 运行结果: 2019-10-27T13:59:58.221+08:00

// 获取时间戳
System.out.println(instant1.toEpochMilli());
// 运行结果: 1572156145000

// 以Unix元年为起点, 进行偏移量运算
Instant instant2 = Instant.ofEpochSecond(60);
System.out.println(instant2);
// 运行结果: 1970-01-01T00:01:00Z
```

## Duration 和 Period

Duration:用于计算两个“时间”间隔。

Period:用于计算两个“日期”间隔。

```

Instant instant_1 = Instant.now();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
Instant instant_2 = Instant.now();

Duration duration = Duration.between(instant_1, instant_2);
System.out.println(duration.toMillis());
// 运行结果: 1000

LocalTime localTime_1 = LocalTime.now();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
LocalTime localTime_2 = LocalTime.now();

System.out.println(Duration.between(localTime_1, localTime_2).toMillis());
// 运行结果: 1000

```

```

LocalDate localDate_1 = LocalDate.of(2018, 9, 9);
LocalDate localDate_2 = LocalDate.now();

Period period = Period.between(localDate_1, localDate_2);
System.out.println(period.getYears());      // 运行结果: 1
System.out.println(period.getMonths());     // 运行结果: 1
System.out.println(period.getDays());       // 运行结果: 18

```

## 日期的操纵

TemporalAdjuster : 时间校正器。有时我们可能需要获取例如: 将日期调整到“下个周日”等操作。

TemporalAdjusters : 该类通过静态方法提供了大量的常用 TemporalAdjuster 的实现。

例如获取下个周日, 如下所示:

```

LocalDate nextSunday =
    LocalDate.now().with(TemporalAdjusters.next(DayOfWeek.SUNDAY));

```

完整的示例代码如下所示。

```

LocalDateTime localDateTime1 = LocalDateTime.now();
System.out.println(localDateTime1);      // 2019-10-27T14:19:56.884

// 获取这个第一天的日期
System.out.println(localDateTime1.with(TemporalAdjusters.firstDayOfMonth()));
// 2019-10-01T14:22:58.574

// 获取下个周末的日期
System.out.println(localDateTime1.with(TemporalAdjusters.next(DayOfWeek.SUNDAY))
);      // 2019-11-03T14:22:58.574

// 自定义: 下一个工作日
LocalDateTime localDateTime2 = localDateTime1.with(1 -> {

```



```

LocalDateTime localDateTime = (LocalDateTime) 1;
DayOfWeek dayOfWeek = localDateTime.getDayOfWeek();

if (dayOfWeek.equals(DayOfWeek.FRIDAY)) {
    return localDateTime.plusDays(3);
} else if (dayOfWeek.equals(DayOfWeek.SATURDAY)) {
    return localDateTime.plusDays(2);
} else {
    return localDateTime.plusDays(1);
}
});
System.out.println(localDateTime2);
// 运行结果: 2019-10-28T14:30:17.400

```

## 解析与格式化

java.time.format.DateTimeFormatter 类：该类提供了三种格式化方法：

- 预定义的标准格式
- 语言环境相关的格式
- 自定义的格式

示例代码如下所示。

```

DateTimeFormatter dateTimeFormatter1 = DateTimeFormatter.ISO_DATE;
LocalDateTime localDateTime = LocalDateTime.now();
String strDate1 = localDateTime.format(dateTimeFormatter1);
System.out.println(strDate1);
// 运行结果: 2019-10-27

// Date -> String
DateTimeFormatter dateTimeFormatter2 = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
String strDate2 = dateTimeFormatter2.format(localDateTime);
System.out.println(strDate2);
// 运行结果: 2019-10-27 14:36:11

// String -> Date
LocalDateTime localDateTime1 = LocalDateTime.parse(strDate2,
dateTimeFormatter2);
System.out.println(localDateTime1);
// 运行结果: 2019-10-27T14:37:39

```

## 时区的处理

Java8 中加入了对时区的支持，带时区的时间分别为：ZonedDateTime、ZonedTime、ZonedDate。

其中每个时区都对应着 ID，地区ID都为“{区域}/{城市}”的格式，例如：Asia/Shanghai 等。

- ZonedDateTime：该类中包含了所有的时区信息
- getAvailableZoneIds()：可以获取所有时区时区信息
- of(id)：用指定的时区信息获取 ZonedDateTime 对象

示例代码如下所示。

```
// 获取所有的时区
Set<String> set = ZoneId.getAvailableZoneIds();
System.out.println(set);
```

// [Asia/Aden, America/Cuiaba, Etc/GMT+9, Etc/GMT+8, Africa/Nairobi, America/Marigot, Asia/Aqtau, Pacific/Kwajalein, America/El\_Salvador, Asia/Pontianak, Africa/Cairo, Pacific/Pago\_Pago, Africa/Mbabane, Asia/Kuching, Pacific/Honolulu, Pacific/Rarotonga, America/Guatemala, Australia/Hobart, Europe/London, America/Belize, America/Panama, Asia/Chungking, America/Managua, America/Indiana/Petersburg, Asia/Yerevan, Europe/Brussels, GMT, Europe/Warsaw, America/Chicago, Asia/Kashgar, Chile/Continental, Pacific/Yap, CET, Etc/GMT-1, Etc/GMT-0, Europe/Jersey, America/Tegucigalpa, Etc/GMT-5, Europe/Istanbul, America/Eirunepe, Etc/GMT-4, America/Miquelon, Etc/GMT-3, Europe/Luxembourg, Etc/GMT-2, Etc/GMT-9, America/Argentina/Catamarca, Etc/GMT-8, Etc/GMT-7, Etc/GMT-6, Europe/Zaporozhye, Canada/Yukon, Canada/Atlantic, Atlantic/St\_Helena, Australia/Tasmania, Libya, Europe/Guernsey, America/Grand\_Turk, US/Pacific-New, Asia/Samarkand, America/Argentina/Cordoba, Asia/Phnom\_Penh, Africa/Kigali, Asia/Almaty, US/Alaska, Asia/Dubai, Europe/Isle\_of\_Man, America/Araguaina, Cuba, Asia/Novosibirsk, America/Argentina/Salta, Etc/GMT+3, Africa/Tunis, Etc/GMT+2, Etc/GMT+1, Pacific/Fakaofo, Africa/Tripoli, Etc/GMT+0, Israel, Africa/Banjul, Etc/GMT+7, Indian/Comoro, Etc/GMT+6, Etc/GMT+5, Etc/GMT+4, Pacific/Port\_Moresby, US/Arizona, Antarctica/Syowa, Indian/Reunion, Pacific/Palau, Europe/Kaliningrad, America/Montevideo, Africa/Windhoek, Asia/Karachi, Africa/Mogadishu, Australia/Perth, Brazil/East, Etc/GMT, Asia/Chita, Pacific/Easter, Antarctica/Davis, Antarctica/McMurdo, Asia/Macao, America/Manaus, Africa/Freetown, Europe/Bucharest, Asia/Tomsk, America/Argentina/Mendoza, Asia/Macau, Europe/Malta, Mexico/BajaSur, Pacific/Tahiti, Africa/Asmara, Europe/Busingen, America/Argentina/Rio\_Gallegos, Africa/Malabo, Europe/Skopje, America/Catamarca, America/Godthab, Europe/Sarajevo, Australia/ACT, GB-Eire, Africa/Lagos, America/Cordoba, Europe/Rome, Asia/Dacca, Indian/Mauritius, Pacific/Samoa, America/Regina, America/Fort\_Wayne, America/Dawson\_Creek, Africa/Algiers, Europe/Mariehamn, America/St\_Johns, America/St\_Thomas, Europe/Zurich, America/Anguilla, Asia/Dili, America/Denver, Africa/Bamako, Europe/Saratov, GB, Mexico/General, Pacific/Wallis, Europe/Gibraltar, Africa/Conakry, Africa/Lubumbashi, Asia/Istanbul, America/Havana, NZ-CHAT, Asia/Choibalsan, America/Porto\_Acre, Asia/Omsk, Europe/Vaduz, US/Michigan, Asia/Dhaka, America/Barbados, Europe/Tiraspol, Atlantic/Cape\_Verde, Asia/Yekaterinburg, America/Louisville, Pacific/Johnston, Pacific/Chatham, Europe/Ljubljana, America/Sao\_Paulo, Asia/Jayapura, America/Curacao, Asia/Dushanbe, America/Guyana, America/Guayaquil, America/Martinique, Portugal, Europe/Berlin, Europe/Moscow, Europe/Chisinau, America/Puerto\_Rico, America/Rankin\_Inlet, Pacific/Ponape, Europe/Stockholm, Europe/Budapest, America/Argentina/Jujuy, Australia/Eucla, Asia/Shanghai, Universal, Europe/Zagreb, America/Port\_of\_Spain, Europe/Helsinki, Asia/Beirut, Asia/Tel\_Aviv, Pacific/Bougainville, US/Central, Africa/Sao\_Tome, Indian/Chagos, America/Cayenne, Asia/Yakutsk, Pacific/Galapagos, Australia/North, Europe/Paris, Africa/Ndjamena, Pacific/Fiji, America/Rainy\_River, Indian/Maldives, Australia/Yancowinna, SystemV/AST4, Asia/Oral, America/Yellowknife, Pacific/Enderbury, America/Juneau, Australia/Victoria, America/Indiana/Vevay, Asia/Tashkent, Asia/Jakarta, Africa/Ceuta, Asia/Barnaul, America/Recife, America/Buenos\_Aires, America/Noronha, America/Swift\_Current, Australia/Adelaide, America/Metlakatla, Africa/Djibouti, America/Paramaribo, Europe/Simferopol, Europe/Sofia, Africa/Nouakchott, Europe/Prague, America/Indiana/Vincennes, Antarctica/Mawson, America/Kralendijk, Antarctica/Troll, Europe/Samara, Indian/Christmas, America/Antigua, Pacific/Gambier, America/Indianapolis, America/Inuvik, America/Iqaluit, Pacific/Funafuti, UTC, Antarctica/Macquarie, Canada/Pacific, America/Moncton, Africa/Gaborone, Pacific/Chuuk, Asia/Pyongyang, America/St\_Vincent, Asia/Gaza, Etc/Universal, PST8PDT, Atlantic/Faeroe, Asia/Qyzylorda, Canada/Newfoundland, America/Kentucky/Louisville, America/Yakutat, Asia/Ho\_Chi\_Minh, Antarctica/Casey, Europe/Copenhagen, Africa/Asmara, Atlantic/Azores, Europe/Vienna, ROK, Pacific/Pitcairn, America/Mazatlan, Australia/Queensland,

Pacific/Nauru, Europe/Tirane, Asia/Kolkata, SystemV/MST7, Australia/Canberra, MET, Australia/Broken\_Hill, Europe/Riga, America/Dominica, Africa/Abidjan, America/Mendoza, America/Santarem, Kwajalein, America/Asuncion, Asia/Ulan\_Bator, NZ, America/Boise, Australia/Currie, EST5EDT, Pacific/Guam, Pacific/Wake, Atlantic/Bermuda, America/Costa\_Rica, America/Dawson, Asia/Chongqing, Eire, Europe/Amsterdam, America/Indiana/Knox, America/North\_Dakota/Beulah, Africa/Accra, Atlantic/Faroe, Mexico/BajaNorte, America/Maceio, Etc/UCT, Pacific/Apia, GMT0, America/Atka, Pacific/Niue, Australia/Lord\_Howe, Europe/Dublin, Pacific/Truk, MST7MDT, America/Monterrey, America/Nassau, America/Jamaica, Asia/Bishkek, America/Atikokan, Atlantic/Stanley, Australia/NSW, US/Hawaii, SystemV/CST6, Indian/Mahe, Asia/Aqtobe, America/Sitka, Asia/Vladivostok, Africa/Libreville, Africa/Maputo, Zulu, America/Kentucky/Monticello, Africa/El\_Aaiun, Africa/Ouagadougou, America/Coral\_Harbour, Pacific/Marquesas, Brazil/West, America/Aruba, America/North\_Dakota/Center, America/Cayman, Asia/Ulaanbaatar, Asia/Baghdad, Europe/San\_Marino, America/Indiana/Tell\_City, America/Tijuana, Pacific/Saipan, SystemV/YST9, Africa/Douala, America/Chihuahua, America/Ojinaga, Asia/Hovd, America/Anchorage, Chile/EasterIsland, America/Halifax, Antarctica/Rothera, America/Indiana/Indianapolis, US/Mountain, Asia/Damascus, America/Argentina/San\_Luis, America/Santiago, Asia/Baku, America/Argentina/Ushuaia, Atlantic/Reykjavik, Africa/Brazzaville, Africa/Porto-Novo, America/La\_Paz, Antarctica/DumontDurville, Asia/Taipei, Antarctica/South\_Pole, Asia/Manila, Asia/Bangkok, Africa/Dar\_es\_Salaam, Poland, Atlantic/Madeira, Antarctica/Palmer, America/Thunder\_Bay, Africa/Addis\_Ababa, Asia/Yangon, Europe/Uzhgorod, Brazil/DeNoronha, Asia/Ashkhabad, Etc/Zulu, America/Indiana/Marengo, America/Creston, America/Punta\_Arenas, America/Mexico\_City, Antarctica/Vostok, Asia/Jerusalem, Europe/Andorra, US/Samoa, PRC, Asia/Vientiane, Pacific/Kiritimati, America/Matamoros, America/Blanc-Sablon, Asia/Riyadh, Iceland, Pacific/Pohnpei, Asia/Ujung\_Pandang, Atlantic/South\_Georgia, Europe/Lisbon, Asia/Harbin, Europe/Oslo, Asia/Novokuznetsk, CST6CDT, Atlantic/Canary, America/Knox\_IN, Asia/Kuwait, SystemV/HST10, Pacific/Efate, Africa/Lome, America/Bogota, America/Menominee, America/Adak, Pacific/Norfolk, Europe/Kirov, America/Resolute, Pacific/Tarawa, Africa/Kampala, Asia/Krasnoyarsk, Greenwich, SystemV/EST5, America/Edmonton, Europe/Podgorica, Australia/South, Canada/Central, Africa/Bujumbura, America/Santo\_Domingo, US/Eastern, Europe/Minsk, Pacific/Auckland, Africa/Casablanca, America/Glace\_Bay, Canada/Eastern, Asia/Qatar, Europe/Kiev, Singapore, Asia/Magadan, SystemV/PST8, America/Port-au-Prince, Europe/Belfast, America/St\_Barthelemy, Asia/Ashgabat, Africa/Luanda, America/Nipigon, Atlantic/Jan\_Mayen, Brazil/Acre, Asia/Muscat, Asia/Bahrain, Europe/Vilnius, America/Fortaleza, Etc/GMT0, US/East-Indiana, America/Hermosillo, America/Cancun, Africa/Maseru, Pacific/Kosrae, Africa/Kinshasa, Asia/Kathmandu, Asia/Seoul, Australia/Sydney, America/Lima, Australia/LHI, America/St\_Lucia, Europe/Madrid, America/Bahia\_Banderas, America/Montserrat, Asia/Brunei, America/Santa\_Isabel, Canada/Mountain, America/Cambridge\_Bay, Asia/Colombo, Australia/West, Indian/Antananarivo, Australia/Brisbane, Indian/Mayotte, US/Indiana-Starke, Asia/Urumqi, US/Aleutian, Europe/Volgograd, America/Lower\_Princes, America/Vancouver, Africa/Blantyre, America/Rio\_Branco, America/Danmarkshavn, America/Detroit, America/Thule, Africa/Lusaka, Asia/Hong\_Kong, Iran, America/Argentina/La\_Rioja, Africa/Dakar, SystemV/CST6CDT, America/Tortola, America/Porto\_Velho, Asia/Sakhalin, Etc/GMT+10, America/Scoresbysund, Asia/Kamchatka, Asia/Thimbu, Africa/Harare, Etc/GMT+12, Etc/GMT+11, Navajo, America/Nome, Europe/Tallinn, Turkey, Africa/Khartoum, Africa/Johannesburg, Africa/Bangui, Europe/Belgrade, Jamaica, Africa/Bissau, Asia/Tehran, WET, Europe/Astrakhan, Africa/Juba, America/Campo\_Grande, America/Belem, Etc/Greenwich, Asia/Saigon, America/Ensenada, Pacific/Midway, America/Jujuy, Africa/Timbuktu, America/Bahia, America/Goose\_Bay, America/Virgin, America/Pangnirtung, Asia/Katmandu, America/Phoenix,

Africa/Niamey, America/Whitehorse, Pacific/Noumea, Asia/Tbilisi, America/Montreal, Asia/Makassar, America/Argentina/San\_Juan, Hongkong, UCT, Asia/Nicosia, America/Indiana/winamac, SystemV/MST7MDT, America/Argentina/ComodRivadavia, America/Boa\_Vista, America/Grenada, Asia/Atyrau, Australia/Darwin, Asia/Khandyga, Asia/Kuala\_Lumpur, Asia/Famagusta, Asia/Thimphu, Asia/Rangoon, Europe/Bratislava, Asia/Calcutta, America/Argentina/Tucuman, Asia/Kabul, Indian/Cocos, Japan, Pacific/Tongatapu, America/New\_York, Etc/GMT-12, Etc/GMT-11, Etc/GMT-10, SystemV/YST9YDT, Europe/Ulyanovsk, Etc/GMT-14, Etc/GMT-13, W-SU, America/Merida, EET, America/Rosario, Canada/Saskatchewan, America/St\_Kitts, Arctic/Longyearbyen, America/Fort\_Nelson, America/Caracas, America/Guadeloupe, Asia/Hebron, Indian/Kerguelen, SystemV/PST8PDT, Africa/Monrovia, Asia/Ust-Nera, Egypt, Asia/Srednekolymysk, America/North\_Dakota/New\_Salem, Asia/Anadyr, Australia/Melbourne, Asia/Irkutsk, America/Shiprock, America/Winnipeg, Europe/Vatican, Asia/Amman, Etc/UTC, SystemV/AST4ADT, Asia/Tokyo, America/Toronto, Asia/Singapore, Australia/Lindeman, America/Los\_Angeles, SystemV/EST5EDT, Pacific/Majuro, America/Argentina/Buenos\_Aires, Europe/Nicosia, Pacific/Guadacanal, Europe/Athens, US/Pacific, Europe/Monaco]

```
// 通过时区构建LocalDateTime
LocalDateTime localDateTime1 =
LocalDateTime.now(ZoneId.of("America/El_Salvador"));
System.out.println(localDateTime1);
// 2019-10-27T00:46:21.268

// 以时区格式显示时间
LocalDateTime localDateTime2 = LocalDateTime.now();
ZonedDateTime zonedDateTime1 =
localDateTime2.atZone(ZoneId.of("Africa/Nairobi"));
System.out.println(zonedDateTime1);
// 2019-10-27T14:46:21.273+03:00[Africa/Nairobi]
```

## 与传统日期处理的转换

类	To遗留类	From遗留类
java.time.Instant java.util.Date	Date.from(instant)	date.toInstant()
java.time.Instant java.sql.Timestamp	Timestamp.from(instant)	timestamp.toInstant()
java.time.ZonedDateTime java.util.GregorianCalendar	GregorianCalendar.from(zonedDateTime)	cal.toZonedDateTime()
java.time.LocalDate java.sql.Date	Date.valueOf(localDate)	date.toLocalDate()
java.time.LocalTime java.sql.Time	Date.valueOf(localDate)	date.toLocalTime()
java.time.LocalDateTime java.sql.Timestamp	Timestamp.valueOf(localDateTime)	timestamp.toLocalDateTime()
java.time.ZoneId java.util.TimeZone	Timezone.getTimeZone(id)	timeZone.toZoneId()
ava.time.format.DateTimeFormatter java.text.DateFormat	formatter.toFormat()	无

## JDK5中的注解

### 1.注解 (@)

注解就相当于一种标记，在程序中加了注解就等于为程序加了某种标记。（JDK1.5新特性）。

## 2.作用

告诉javac编译器或者java开发工具.....向其传递某种信息，作为一个标记。

## 3.如何理解注解？

一个注解就是一个类。

标记可以加在包、类、字段、方法，方法参数以及局部变量上。可以同时存在多个注解。

每一个注解结尾都没有“；”或者其他特别符号。

定义注解需要的基础注解信息如下所示。

```
@SuppressWarnings("deprecation") //编译器警告过时（source阶段）
@Deprecated //过时（Runtime阶段）
@Override //重写（source阶段）
@Retention(RetentionPolicy.RUNTIME)
//保留注解到程序运行时。（Runtime阶段）
@Target({ElementType.METHOD,ElementType.TYPE})
//标记既能定义在方法上，又能定义在类、接口、枚举上等。
```

**注意：**

- 1) 添加注解需要有注解类。RetentionPolicy是一个枚举类（有三个成员）。
- 2) Target中可以存放数组。它的默认值为任何元素。
  - ElementType.METHOD：表示只能标记在方法上。
  - ElementType.TYPE：表示只能标记定义在类上、接口上、枚举上等
- 3) ElementType也是枚举类。成员包括：ANNOTATION\_TYPE（注解）、CONSTRUCTOR（构造方法）、FIELD（成员变量）、LOCAL\_VARIABLE（变量）、METHOD（方法）、PACKAGE（包）、PARAMETER（参数）、TYPE。

## 4.关于注解

- 元注解：注解的注解（理解：给一个注解类再加注解）
- 元数据：数据的数据
- 元信息：信息的信息

## 5.注解分为三个阶段

java源文件--> class文件 --> 内存中的字节码。

Retention的注解有三种取值：（分别对应注解的三个阶段）

- RetentionPolicy.SOURCE
- RetentionPolicy.CLASS
- RetentionPolicy.RUNTIME

注意：注解的默认阶段是Class。

## 6.注解的属性类型

原始类型（就是八个基本数据类型）、String类型、Class类型、数组类型、枚举类型、注解类型。

## 7.为注解增加属性

value：是一个特殊的属性，若在设置值时只有一个value属性需要设置或者其他属性都采用默认值时，那么value=可以省略，直接写所设置的值即可。

例如：@SuppressWarnings("deprecation")

为属性指定缺省值（默认值）：

例如：String value() default "blue"; //定义在注解类中

数组类型的属性：

例如：int[] arrayArr() default {3,4,5,5}; //定义在注解类中

SunAnnotation(arrayArr={3,9,8}) //设置数组值

注意：如果数组属性中只有一个元素时，属性值部分可以省略大括号。

例如：SunAnnotation(arrayArr=9)

枚举类型的属性：

例如：EnumDemo.TrafficLamp lamp()

////枚举类型属性，定义在注解类中，这里使用了自定义的枚举类EnumDemo.java并没有给出相关代码，这里只是举个例子

default EnumDemo.TrafficLamp.RED;

注解类型的属性：

例如：MetaAnnotation annotationAttr()

//定义在一个注解类中,并指定缺省值,

//此属性关联到注解类：MetaAnnotation.java,

default @MetaAnnotation("lhm");

//设置注解属性值

@SunAnnotation(annotationAttr=@MetaAnnotation("flx"))

## Java8中的注解

对于注解（也被称做元数据），Java 8 主要有两点改进：类型注解和重复注解。

### 1.类型注解

1) Java 8 的类型注解扩展了注解使用的范围。

在java 8之前，注解只能是在声明的地方所使用，java8开始，注解可以应用在任何地方。

例如：

创建类实例

```
new @Interned MyObject();
```

类型映射

```
myString = (@NonNull String) str;
```

implements 语句中

```
class UnmodifiableList<T> implements @ReadOnly List<@ReadOnly T> { ... }
```

throw exception声明



```
void monitorTemperature() throws@Critical TemperatureException { ... }
```

注意：

在Java 8里面，当类型转化甚至分配新对象的时候，都可以在声明变量或者参数的时候使用注解。Java注解可以支持任意类型。

类型注解只是语法而不是语义，并不会影响java的编译时间，加载时间，以及运行时间，也就是说，编译成class文件的时候并不包含类型注解。

## 2) 新增ElementType.TYPE\_USE 和ElementType.PARAMETER (在Target上)

新增的两个注释的程序元素类型 ElementType.TYPE\_USE 和 ElementType.PARAMETER用来描述注解的新场合。

- ElementType.PARAMETER 表示该注解能写在类型变量的声明语句中。
- ElementType.TYPE\_USE 表示该注解能写在使用类型的任何语句中（例如：声明语句、泛型和强制转换语句中的类型）。

例如，下面的示例。

```
@Target({ElementType.PARAMETER, ElementType.TYPE_USE})  
@interface MyAnnotation {}
```

## 3)类型注解的作用

类型注解被用来支持在Java的程序中做强类型检查。配合第三方插件工具Checker Framework（注：此插件so easy,这里不介绍了），可以在编译的时候检测出runtime error（例如：UnsupportedOperationException；NumberFormatException；NullPointerException异常等都是runtime error），以提高代码质量。这就是类型注解的作用。

注意：使用Checker Framework可以找到类型注解出现的地方并检查。

例如下面的代码。

```
import checkers.nullness.quals.*;  
public class TestDemo{  
    void sample() {  
        @NonNull Object my = new Object();  
    }  
}
```

使用javac编译上面的类：（当然若下载了Checker Framework插件就不需要这么麻烦了）

```
javac -processor checkers.nullness.NullnessChecker TestDemo.java
```

上面编译是通过的，但若修改代码：

```
@NonNull Object my = null;
```

但若不想使用类型注解检测出来错误，则不需要processor，正常javac TestDemo.java是可以编译的，但是运行时会报 NullPointerException 异常。

为了能在编译期间就自动检查出这类异常，可以通过类型注解结合 Checker Framework 提前排查出来错误异常。



注意java 5,6,7版本是不支持注解@NonNull，但checker framework 有个向下兼容的解决方案，就是将类型注解@NonNull 用/\*\*\*/注释起来。

```
import checkers.nullness.quals.*;
public class TestDemo{
    void sample() {
        /*@NonNull*/ Object my = null;
    }
}
```

这样javac编译器就会忽略掉注释块，但用checker framework里面的javac编译器同样能够检测出@NonNull错误。

通过 类型注解 + checker framework 可以在编译时就找到runtime error。

## 2.重复注解

允许在同一声明类型（类，属性，或方法）上多次使用同一个注解。

Java8以前的版本使用注解有一个限制是相同的注解在同一位置只能使用一次，不能使用多次。

Java 8 引入了重复注解机制，这样相同的注解可以在同一地方使用多次。重复注解机制本身必须用@Repeatable 注解。

实际上，重复注解不是一个语言上的改变，只是编译器层面的改动，技术层面仍然是一样的。

例如，我们可以使用如下示例来具体对比Java8之前的版本和Java8中的注解。

### 1) 自定义一个包装类Hints注解用来放置一组具体的Hint注解

```
@interface MyHints {
    Hint[] value();
}

@Repeatable(MyHints.class)
@interface Hint {
    String value();
}
```

使用包装类当容器来存多个注解（旧版本方法）

```
@MyHints({@Hint("hint1"), @Hint("hint2")})
class Person {}
```

使用多重注解（新方法）

```
@Hint("hint1")
@Hint("hint2")
class Person {}
```

### 2)完整类测试如下所示。

```
public class RepeatingAnnotations {
    @Target(ElementType.TYPE)
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Filters {
        Filter[] value();
    }
}
```

```

    }

    @Target(ElementType.TYPE)
    @Retention(RetentionPolicy.RUNTIME)
    @Repeatable(Filters.class)
    public @interface Filter {
        String value();
    }
    @Filter("filter1")
    @Filter("filter2")
    public interface Filterable {
    }
    public static void main(String[] args) {
        for (Filter filter :
Filterable.class.getAnnotationsByType(Filter.class)) {
            System.out.println(filter.value());
        }
    }
}

```

输出结果:

```

filter1
filter2

```

### 分析:

注释Filter被@Repeatable( Filters.class )注释。Filters 只是一个容器，它持有Filter，编译器尽力向程序员隐藏它的存在。通过这样的方式，Filterable接口可以被Filter注释两次。

另外，反射的API提供一个新方法getAnnotationsByType() 来返回重复注释的类型（注意Filterable.class.getAnnotation( Filters.class )将会返回编译器注入的Filters实例。

3) java 8之前也有重复使用注解的解决方案，但可读性不好。

```

public @interface MyAnnotation {
    String role();
}

public @interface Annotations {
    MyAnnotation[] value();
}

public class RepeatAnnotationUseOldVersion {
    @Annotations({@MyAnnotation(role="Admin"),@MyAnnotation(role="Manager")})
    public void doSomething(){
    }
}

```

**Java8的实现方式**（由另一个注解来存储重复注解，在使用时候，用存储注解Authorities来扩展重复注解），**可读性更强**。

```

@Repeatable(Annotations.class)
public @interface MyAnnotation {
    String role();
}

```

```

public @interface Annotations {
    MyAnnotation[] value();
}

public class RepeatAnnotationUseOldVersion {
    @MyAnnotation(role="Admin")
    @MyAnnotation(role="Manager")
    public void doSomething(){
    }
}

```

什么？没看懂？那就再来一波！！

## Java8对注解的增强

Java 8对注解处理提供了两点改进：可重复的注解及可用于类型的注解。总体来说，比较简单，下面，我们就以实例的形式来说明Java8中的重复注解和类型注解。

首先，我们来定义一个注解类BingheAnnotation，如下所示。

```

package io.mykit.binghe.java8.annotition;

import java.lang.annotation.*;

/**
 * @author binghe
 * @version 1.0.0
 * @description 定义注解
 */
@Repeatable(BingheAnnotations.class)
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD,
        ElementType.PARAMETER, ElementType.CONSTRUCTOR,
        ElementType.LOCAL_VARIABLE, ElementType.TYPE_PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface BingheAnnotation {
    String value();
}

```

注意：在BingheAnnotation注解类上比普通的注解多了一个@Repeatable(BingheAnnotations.class)注解，有小伙伴会问：这个是个啥啊？这个就是Java8中定义可重复注解的关键，至于BingheAnnotations.class，大家别急，继续往下看就明白了。

接下来，咱们定义一个BingheAnnotations注解类，如下所示。

```

package io.mykit.binghe.java8.annotation;

import java.lang.annotation.*;

/**
 * @author binghe
 * @version 1.0.0
 * @description 定义注解
 */

```

```

@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD,
        ElementType.PARAMETER, ElementType.CONSTRUCTOR,
        ElementType.LOCAL_VARIABLE, ElementType.TYPE_PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface BingheAnnotations {
    BingheAnnotation[] value();
}

```

看到这里，大家明白了吧！！没错，BingheAnnotations也是一个注解类，它相比于BingheAnnotation注解类来说，少了一个@Repeatable(BingheAnnotations.class)注解，也就是说，BingheAnnotations注解类的定义与普通的注解几乎没啥区别。值得注意的是，我们在BingheAnnotations注解类中，定义了一个BingheAnnotation注解类的数组，也就是说，在BingheAnnotations注解类中，包含有多个BingheAnnotation注解。所以，在BingheAnnotation注解类上指定@Repeatable(BingheAnnotations.class)来说明可以在类、字段、方法、参数、构造方法、参数上重复使用BingheAnnotation注解。

接下来，我们创建一个Binghe类，在Binghe类中定义一个init()方法，在init方法上，重复使用@BingheAnnotation注解指定相应的数据，如下所示。

```

package io.mykit.binghe.java8.annotation;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试注解
 */
@BingheAnnotation("binghe")
@BingheAnnotation("class")
public class Binghe {

    @BingheAnnotation("init")
    @BingheAnnotation("method")
    public void init(){

    }

}

```

到此，我们就可以测试重复注解了，创建类BingheAnnotationTest，对重复注解进行测试，如下所示。

```

package io.mykit.binghe.java8.annotation;

import java.lang.reflect.Method;
import java.util.Arrays;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试注解
 */
public class BingheAnnotationTest {

    public static void main(String[] args) throws NoSuchMethodException {
        Class<Binghe> clazz = Binghe.class;
        BingheAnnotation[] annotations =
            clazz.getAnnotationsByType(BingheAnnotation.class);
    }
}

```

```

        System.out.println("类上的重复注解如下: ");
        Arrays.stream(annotations).forEach((a) -> System.out.print(a.value() + "
"));

        System.out.println();
        System.out.println("=====");

        Method method = clazz.getMethod("init");
        annotations = method.getAnnotationsByType(BingheAnnotation.class);
        System.out.println("方法上的重复注解如下: ");
        Arrays.stream(annotations).forEach((a) -> System.out.print(a.value() + "
"));
    }
}

```

运行main()方法，输出如下的结果信息。

```

类上的重复注解如下:
binghe class
=====
方法上的重复注解如下:
init method

```

## 写在最后

如果你觉得冰河写的还不错，请微信搜索并关注「**冰河技术**」微信公众号，跟冰河学习高并发、分布式、微服务、大数据、互联网和云原生技术，「**冰河技术**」微信公众号更新了大量技术专题，每一篇技术文章干货满满！不少读者已经通过阅读「**冰河技术**」微信公众号文章，吊打面试官，成功跳槽到大厂；也有不少读者实现了技术上的飞跃，成为公司的技术骨干！如果你也想像他们一样提升自己的能力，实现技术能力的飞跃，进大厂，升职加薪，那就关注「**冰河技术**」微信公众号吧，每天更新超硬核技术干货，让你对如何提升技术能力不再迷茫！

冰河技术微信公众号

微信公众号：冰河技术  
 微信：sun\_shine\_lyz  
 扫码关注！

