

18 函数式编程 *Java8*

本章我们介绍Java的函数式编程。

我们先看看什么是函数。函数是一种最基本的任务，一个大型程序就是一个顶层函数调用若干底层函数，这些被调用的函数又可以调用其他函数，即大任务被一层层拆解并执行。所以函数就是面向过程的程序设计的基本单元。

Java不支持单独定义函数，但可以把静态方法视为独立的函数，把实例方法视为自带 `this` 参数的函数。

而函数式编程（请注意多了一个“式”字）——Functional Programming，虽然也可以归结到面向过程的设计，但其思想更接近数学计算。

我们首先要搞明白计算机（Computer）和计算（Compute）的概念。

在计算机的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。

而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。

对应到编程语言，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Lisp语言。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

函数式编程最早是数学家[阿隆佐·邱奇](#)研究的一套函数变换逻辑，又称Lambda Calculus (λ -Calculus)，所以也经常把函数式编程称为Lambda计算。

Java平台从Java 8开始，支持函数式编程。

Lambda基础

在了解Lambda之前，我们先回顾一下Java的方法。

Java的方法分为实例方法，例如 `Integer` 定义的 `equals()` 方法：

```
public final class Integer {  
    boolean equals(Object o) {  
        ...  
    }  
}
```

以及静态方法，例如 `Integer` 定义的 `parseInt()` 方法：

```
public final class Integer {  
    public static int parseInt(String s) {  
        ...  
    }  
}
```

无论是实例方法，还是静态方法，本质上都相当于过程式语言的函数。例如C函数：

```
char* strcpy(char* dest, char* src)
```

只不过Java的实例方法隐含地传入了一个 `this` 变量，即实例方法总是有一个隐含参数 `this`。

函数式编程（Functional Programming）是把函数作为基本运算单元，函数可以作为变量，可以接收函数，还可以返回函数。历史上研究函数式编程的理论是Lambda演算，所以我们经常把支持函数式编程的编码风格称为Lambda表达式。

Lambda表达式

在Java程序中，我们经常遇到一大堆单方法接口，即一个接口只定义了一个方法：

- Comparator
- Runnable
- Callable

以 `Comparator` 为例，我们想要调用 `Arrays.sort()` 时，可以传入一个 `Comparator` 实例，以匿名类方式编写如下：

```
String[] array = ...
Arrays.sort(array, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s1.compareTo(s2);
    }
});
```

上述写法非常繁琐。从Java 8开始，我们可以用Lambda表达式替换单方法接口。改写上述代码如下：

```
// Lambda
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        String[] array = new String[] { "Apple", "Orange", "Banana", "Lemon" };
        Arrays.sort(array, (s1, s2) -> {
            return s1.compareTo(s2);
        });
        System.out.println(String.join(", ", array));
    }
}
```

观察Lambda表达式的写法，它只需要写出方法定义：

```
(s1, s2) -> {
    return s1.compareTo(s2);
}
```

其中，参数是 `(s1, s2)`，参数类型可以省略，因为编译器可以自动推断出 `String` 类型。`-> { ... }` 表示方法体，所有代码写在内部即可。Lambda表达式没有 `class` 定义，因此写法非常简洁。

如果只有一行 `return xxx` 的代码，完全可以用更简单的写法：

```
Arrays.sort(array, (s1, s2) -> s1.compareTo(s2));
```

返回值的类型也是由编译器自动推断的，这里推断出的返回值是 `int`，因此，只要返回 `int`，编译器就不会报错。

FunctionalInterface

我们把只定义了单方法的接口称之为 `FunctionalInterface`，用注解 `@FunctionalInterface` 标记。例如，`Callable` 接口：

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

再来看 `Comparator` 接口：

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

    int compare(T o1, T o2);

    boolean equals(Object obj);

    default Comparator<T> reversed() {
        return Collections.reverseOrder(this);
    }

    default Comparator<T> thenComparing(Comparator<? super T> other) {
        ...
    }
    ...
}
```

虽然 `Comparator` 接口有很多方法，但只有一个抽象方法 `int compare(T o1, T o2)`，其他的方法都是 `default` 方法或 `static` 方法。另外注意到 `boolean equals(Object obj)` 是 `Object` 定义的方法，不算在接口方法内。因此，`Comparator` 也是一个 `FunctionalInterface`。

练习

下载练习：[使用Lambda表达式实现忽略大小写排序](#)（推荐使用[IDE练习插件](#)快速下载）

小结

- 单方法接口被称为 `FunctionalInterface`。
- 接收 `FunctionalInterface` 作为参数的时候，可以把实例化的匿名类改写为 `Lambda` 表达式，能大大简化代码。
- `Lambda` 表达式的参数和返回值均可由编译器自动推断。

方法引用

使用 `Lambda` 表达式，我们就可以不必编写 `FunctionalInterface` 接口的实现类，从而简化代码：

```
Arrays.sort(array, (s1, s2) -> {
    return s1.compareTo(s2);
});
```

实际上，除了Lambda表达式，我们还可以直接传入方法引用。例如：

```
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        String[] array = new String[] { "Apple", "Orange", "Banana", "Lemon" };
        Arrays.sort(array, Main::cmp);
        System.out.println(String.join(", ", array));
    }

    static int cmp(String s1, String s2) {
        return s1.compareTo(s2);
    }
}
```

```
Apple, Banana, Lemon, Orange
```

上述代码在 `Arrays.sort()` 中直接传入了静态方法 `cmp` 的引用，用 `Main::cmp` 表示。

因此，所谓方法引用，是指如果某个方法签名和接口恰好一致，就可以直接传入方法引用。

因为 `Comparator` 接口定义的方法是 `int compare(String, String)`，和静态方法 `int cmp(String, String)` 相比，除了方法名外，方法参数一致，返回类型相同，因此，我们说两者的方法签名一致，可以直接把方法名作为Lambda表达式传入：

```
Arrays.sort(array, Main::cmp);
```

注意：在这里，方法签名只看参数类型和返回类型，不看方法名称，也不看类的继承关系。

我们再看看如何引用实例方法。如果我们把代码改写如下：

```
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        String[] array = new String[] { "Apple", "Orange", "Banana", "Lemon" };
        Arrays.sort(array, String::compareTo);
        System.out.println(String.join(", ", array));
    }
}
```

```
Apple, Banana, Lemon, Orange
```

不但可以编译通过，而且运行结果也是一样的，这说明 `String.compareTo()` 方法也符合Lambda定义。

观察 `String.compareTo()` 的方法定义：

```
public final class String {
    public int compareTo(String o) {
        // ...
    }
}
```

这个方法的签名只有一个参数，为什么和 `int Comparator.compare(String, String)` 能匹配呢？

因为实例方法有一个隐含的 `this` 参数，`String` 类的 `compareTo()` 方法在实际调用的时候，第一个隐含参数总是传入 `this`，相当于静态方法：

```
public static int compareTo(this, String o);
```

所以，`String.compareTo()` 方法也可作为方法引用传入。

构造方法引用

除了可以引用静态方法和实例方法，我们还可以引用构造方法。

我们来看一个例子：如果要把一个 `List` 转换为 `List`，应该怎么办？

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

List<String> names = List.of("Bob", "Alice", "Tim");
List<Person> persons = ???
```

传统的做法是先定义一个 `ArrayList`，然后用 `for` 循环填充这个 `List`：

```
List<String> names = List.of("Bob", "Alice", "Tim");
List<Person> persons = new ArrayList<>();
for (String name : names) {
    persons.add(new Person(name));
}
```

要更简单地实现 `String` 到 `Person` 的转换，我们可以引用 `Person` 的构造方法：

```
// 引用构造方法
import java.util.*;
import java.util.stream.*;
public class Main {
    public static void main(String[] args) {
        List<String> names = List.of("Bob", "Alice", "Tim");
        List<Person> persons =
names.stream().map(Person::new).collect(Collectors.toList());
        System.out.println(persons);
    }
}

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

```
[Person:Bob, Person:Alice, Person:Tim]
```

后面我们会讲到 `Stream` 的 `map()` 方法。现在我们看到，这里的 `map()` 需要传入的 `FunctionalInterface` 的定义是：

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

把泛型对应上就是方法签名 `Person apply(String)`，即传入参数 `String`，返回类型 `Person`。而 `Person` 类的构造方法恰好满足这个条件，因为构造方法的参数是 `String`，而构造方法虽然没有 `return` 语句，但它会隐式地返回 `this` 实例，类型就是 `Person`，因此，此处可以引用构造方法。构造方法的引用写法是 `类名::new`，因此，此处传入 `Person::new`。

练习

下载练习：[使用方法引用实现忽略大小写排序](#)（推荐使用[IDE练习插件](#)快速下载）

小结

`FunctionalInterface` 允许传入：

- 接口的实现类（传统写法，代码较繁琐）；
- Lambda表达式（只需列出参数名，由编译器推断类型）；
- 符合方法签名的静态方法；
- 符合方法签名的实例方法（实例类型被看做第一个参数类型）；
- 符合方法签名的构造方法（实例类型被看做返回类型）。

`FunctionalInterface` 不强制继承关系，不需要方法名称相同，只要求方法参数（类型和数量）与方法返回类型相同，即认为方法签名相同。

使用Stream

Java从8开始，不但引入了Lambda表达式，还引入了一个全新的流式API：Stream API。它位于 `java.util.stream` 包中。

划重点：这个 `Stream` 不同于 `java.io` 的 `InputStream` 和 `OutputStream`，它代表的是任意Java对象的序列。两者对比如下：

	java.io	java.util.stream
存储	顺序读写的 byte 或 char	顺序输出的任意Java对象实例
用途	序列化至文件或网络	内存计算 / 业务逻辑

有同学会问：一个顺序输出的Java对象序列，不就是一个 `List` 容器吗？

再次划重点：这个 `Stream` 和 `List` 也不一样，`List` 存储的每个元素都是已经存储在内存中的某个Java对象，而 `Stream` 输出的元素可能并没有预先存储在内存中，而是实时计算出来的。

换句话说，`List` 的用途是操作一组已存在的Java对象，而 `Stream` 实现的是惰性计算，两者对比如下：

	java.util.List	java.util.stream
元素	已分配并存储在内存	可能未分配，实时计算
用途	操作一组已存在的Java对象	惰性计算

Stream 看上去有点不好理解，但我们举个例子就明白了。

如果我们要表示一个全体自然数的集合，显然，用 List 是不可能写出来的，因为自然数是无限的，内存再大也没法放到 List 中：

```
List<BigInteger> list = ??? // 全体自然数？
```

但是，用 Stream 可以做到。写法如下：

```
Stream<BigInteger> naturals = createNaturalStream(); // 全体自然数
```

我们先不考虑 createNaturalStream() 这个方法是如何实现的，我们看看如何使用这个 Stream。

首先，我们可以对每个自然数做一个平方，这样我们就把这个 Stream 转换成了另一个 Stream：

```
Stream<BigInteger> naturals = createNaturalStream(); // 全体自然数
Stream<BigInteger> streamNxN = naturals.map(n -> n.multiply(n)); // 全体自然数的平方
```

因为这个 streamNxN 也有无限多个元素，要打印它，必须首先把无限多个元素变成有限个元素，可以用 limit() 方法截取前100个元素，最后用 forEach() 处理每个元素，这样，我们就打印出了前100个自然数的平方：

```
Stream<BigInteger> naturals = createNaturalStream();
naturals.map(n -> n.multiply(n)) // 1, 4, 9, 16, 25...
    .limit(100)
    .forEach(System.out::println);
```

我们总结一下 Stream 的特点：它可以“存储”有限个或无限个元素。这里的存储打了个引号，是因为元素有可能已经全部存储在内存中，也有可能是根据需要实时计算出来的。

Stream 的另一个特点是，一个 Stream 可以轻易地转换为另一个 Stream，而不是修改原 Stream 本身。

最后，真正的计算通常发生在最后结果的获取，也就是惰性计算。

```
Stream<BigInteger> naturals = createNaturalStream(); // 不计算
Stream<BigInteger> s2 = naturals.map(BigInteger::multiply); // 不计算
Stream<BigInteger> s3 = s2.limit(100); // 不计算
s3.forEach(System.out::println); // 计算
```

惰性计算的特点是：一个 Stream 转换为另一个 Stream 时，实际上只存储了转换规则，并没有任何计算发生。

例如，创建一个全体自然数的 Stream，不会进行计算，把它转换为上述 s2 这个 Stream，也不会进行计算。再把 s2 这个无限 Stream 转换为 s3 这个有限的 Stream，也不会进行计算。只有最后，调用 forEach 确实需要 Stream 输出的元素时，才进行计算。我们通常把 Stream 的操作写成链式操作，代码更简洁：

```
createNaturalStream()
    .map(BigInteger::multiply)
    .limit(100)
    .forEach(System.out::println);
```

因此，Stream API的基本用法就是：创建一个Stream，然后做若干次转换，最后调用一个求值方法获取真正计算的结果：

```
int result = createNaturalStream() // 创建Stream
    .filter(n -> n % 2 == 0) // 任意个转换
    .map(n -> n * n) // 任意个转换
    .limit(100) // 任意个转换
    .sum(); // 最终计算结果
```

小结

Stream API的特点是：

- Stream API提供了一套新的流式处理的抽象序列；
- Stream API支持函数式编程和链式操作；
- Stream可以表示无限序列，并且大多数情况下是惰性求值的。

创建Stream

要使用Stream，就必须现创建它。创建Stream有很多种方法，我们来——介绍。

Stream.of()

创建Stream最简单的方式是直接调用Stream.of()静态方法，传入可变参数即创建了一个能输出确定元素的Stream：

```
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("A", "B", "C", "D");
        // forEach()方法相当于内部循环调用，
        // 可传入符合Consumer接口的void accept(T t)的方法引用：
        stream.forEach(System.out::println);
    }
}
```

```
A
B
C
D
```

虽然这种方式基本上没啥实质性用途，但测试的时候很方便。

基于数组或Collection

第二种创建 `Stream` 的方法是基于一个数组或者 `Collection`，这样该 `Stream` 输出的元素就是数组或者 `Collection` 持有的元素：

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

public class Main {
    public static void main(String[] args) {
        Stream<String> stream1 = Arrays.stream(new String[] { "A", "B", "C" });
        Stream<String> stream2 = List.of("X", "Y", "Z").stream();
        stream1.forEach(System.out::println);
        stream2.forEach(System.out::println);
    }
}
```

A
B
C
X
Y
Z

把数组变成 `Stream` 使用 `Arrays.stream()` 方法。对于 `Collection` (`List`、`Set`、`Queue` 等)，直接调用 `stream()` 方法就可以获得 `Stream`。

上述创建 `Stream` 的方法都是把一个现有的序列变为 `Stream`，它的元素是固定的。

基于Supplier

创建 `Stream` 还可以通过 `Stream.generate()` 方法，它需要传入一个 `Supplier` 对象：

```
Stream<String> s = Stream.generate(Supplier<String> sp);
```

基于 `Supplier` 创建的 `Stream` 会不断调用 `Supplier.get()` 方法来不断产生下一个元素，这种 `Stream` 保存的不是元素，而是算法，它可以用来表示无限序列。

例如，我们编写一个能不断生成自然数的 `Supplier`，它的代码非常简单，每次调用 `get()` 方法，就生成下一个自然数：

```
import java.util.function.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Stream<Integer> natual = Stream.generate(new NatualSupplier());
        // 注意：无限序列必须先变成有限序列再打印：
        natual.limit(20).forEach(System.out::println);
    }
}

class NatualSupplier implements Supplier<Integer> {
    int n = 0;
    public Integer get() {
```

```
        n++;  
        return n;  
    }  
}
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20
```

上述代码我们用一个 `Supplier<Integer>` 模拟了一个无限序列（当然受 `int` 范围限制不是真的无限大）。如果用 `List` 表示，即便在 `int` 范围内，也会占用巨大的内存，而 `Stream` 几乎不占用空间，因为每个元素都是实时计算出来的，用的时候再算。

对于无限序列，如果直接调用 `forEach()` 或者 `count()` 这些最终求值操作，会进入死循环，因为永远无法计算完这个序列，所以正确的方法是先把无限序列变成有限序列，例如，用 `limit()` 方法可以截取前面若干个元素，这样就变成了一个有限序列，对这个有限序列调用 `forEach()` 或者 `count()` 操作就没有问题。

其他方法

创建 `Stream` 的第三种方法是通过一些API提供的接口，直接获得 `Stream`。

例如，`Files` 类的 `lines()` 方法可以把一个文件变成一个 `Stream`，每个元素代表文件的一行内容：

```
try (Stream<String> lines = Files.lines(Paths.get("/path/to/file.txt"))) {  
    ...  
}
```

此方法对于按行遍历文本文件十分有用。

另外，正则表达式的 `Pattern` 对象有一个 `splitAsStream()` 方法，可以直接把一个长字符串分割成 `Stream` 序列而不是数组：

```
Pattern p = Pattern.compile("\\s+");  
Stream<String> s = p.splitAsStream("The quick brown fox jumps over the lazy dog");  
s.forEach(System.out::println);
```

基本类型


因为Java的范型不支持基本类型，所以我们无法用 `Stream<int>` 这样的类型，会发生编译错误。为了保存 `int`，只能使用 `Stream<Integer>`，但这样会产生频繁的装箱、拆箱操作。为了提高效率，Java标准库提供了 `IntStream`、`LongStream` 和 `DoubleStream` 这三种使用基本类型的 `Stream`，它们的使用方法和范型 `Stream` 没有大的区别，设计这三个 `Stream` 的目的是提高运行效率：

```
// 将int[]数组变为IntStream：
IntStream is = Arrays.stream(new int[] { 1, 2, 3 });
// 将Stream<String>转换为LongStream：
LongStream ls = List.of("1", "2", "3").stream().mapToLong(Long::parseLong);
```

练习

编写一个能输出斐波拉契数列（Fibonacci）的 `LongStream`：

```
1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

从  **gitee** 下载练习：[FibonacciStream练习](#)（推荐使用[IDE练习插件](#)快速下载）

小结

创建 `Stream` 的方法有：

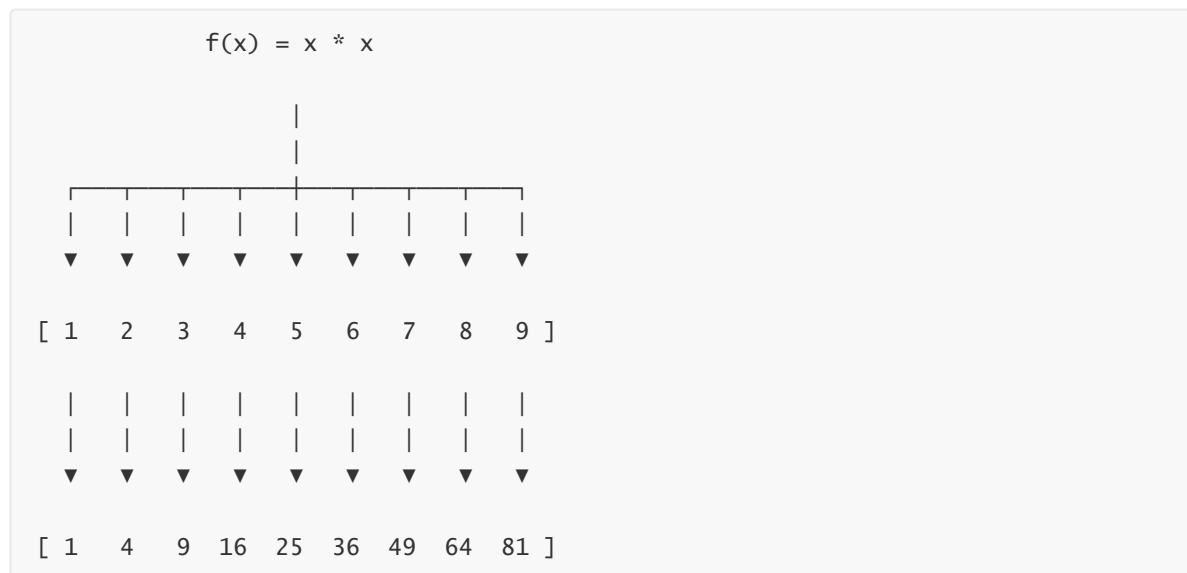
- 通过指定元素、指定数组、指定 `Collection` 创建 `Stream`；
- 通过 `Supplier` 创建 `Stream`，可以是无限序列；
- 通过其他类的相关方法创建。

基本类型的 `Stream` 有 `IntStream`、`LongStream` 和 `DoubleStream`。

使用map

`Stream.map()` 是 `Stream` 最常用的一个转换方法，它把一个 `Stream` 转换为另一个 `Stream`。

所谓 `map` 操作，就是把一种操作运算，映射到一个序列的每一个元素上。例如，对 `x` 计算它的平方，可以使用函数 `f(x) = x * x`。我们把这个函数映射到一个序列 1, 2, 3, 4, 5 上，就得到了另一个序列 1, 4, 9, 16, 25：



可见，`map` 操作，把一个 `Stream` 的每个元素——对应到应用了目标函数的结果上。

```
Stream<Integer> s = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> s2 = s.map(n -> n * n);
```

如果我们查看 `Stream` 的源码，会发现 `map()` 方法接收的对象是 `Function` 接口对象，它定义了一个 `apply()` 方法，负责把一个 `T` 类型转换成 `R` 类型：

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

其中，`Function` 的定义是：

```
@FunctionalInterface
public interface Function<T, R> {
    // 将T类型转换为R:
    R apply(T t);
}
```

利用 `map()`，不但能完成数学计算，对于字符串操作，以及任何Java对象都是非常有用的。例如：

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

public class Main {
    public static void main(String[] args) {
        List.of(" Apple ", " pear ", " ORANGE", " BaNaNa ")
            .stream()
            .map(String::trim) // 去空格
            .map(String::toLowerCase) // 变小写
            .forEach(System.out::println); // 打印
    }
}
```

```
apple
pear
orange
banana
```

通过若干步 `map` 转换，可以写出逻辑简单、清晰的代码。

练习

使用 `map()` 把一组 `String` 转换为 `LocalDate` 并打印。

从  **gitee** 下载练习：[map练习](#)（推荐使用[IDE练习插件](#)快速下载）

小结

`map()` 方法用于将一个 `Stream` 的每个元素映射成另一个元素并转换成一个新的 `Stream`；

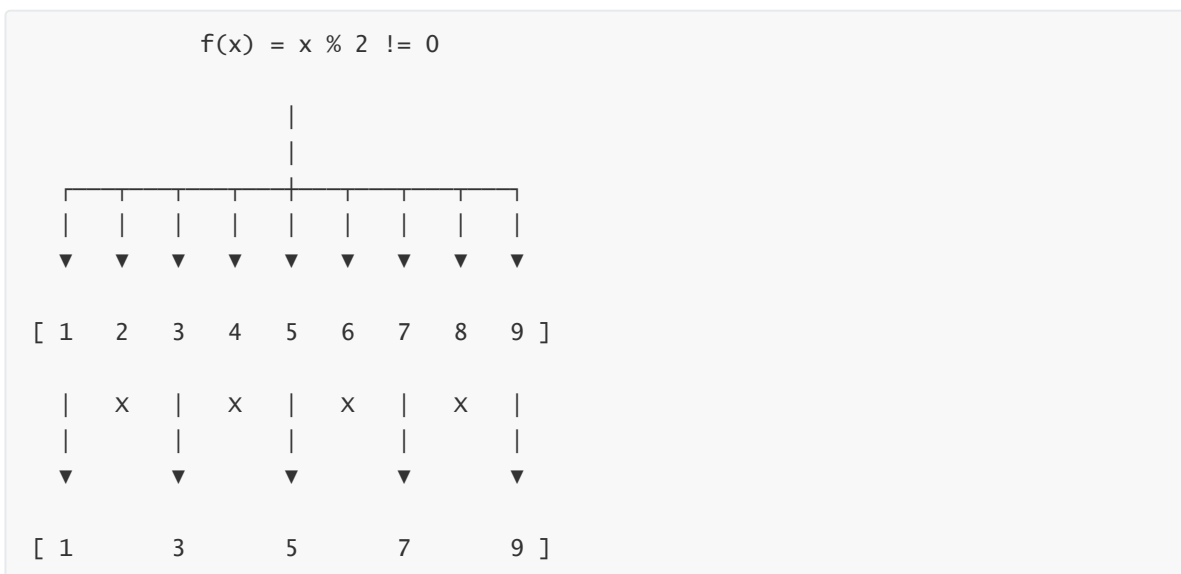
可以将一种元素类型转换成另一种元素类型。

使用filter

`Stream.filter()` 是 `Stream` 的另一个常用转换方法。

所谓 `filter()` 操作，就是对一个 `Stream` 的所有元素——进行测试，不满足条件的就被“滤掉”了，剩下的满足条件的元素就构成了一个新的 `Stream`。

例如，我们对 1, 2, 3, 4, 5 这个 `Stream` 调用 `filter()`，传入的测试函数 `f(x) = x % 2 != 0` 用来判断元素是否是奇数，这样就过滤掉偶数，只剩下奇数，因此我们得到了另一个序列 1, 3, 5:



用 `IntStream` 写出上述逻辑，代码如下：

```
import java.util.stream.IntStream;

public class Main {
    public static void main(String[] args) {
        IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
            .filter(n -> n % 2 != 0)
            .forEach(System.out::println);
    }
}
```

```
1
3
5
7
9
```

从结果可知，经过 `filter()` 后生成的 `Stream` 元素可能变少。

`filter()` 方法接收的对象是 `Predicate` 接口对象，它定义了一个 `test()` 方法，负责判断元素是否符合条件：

```
@FunctionalInterface
public interface Predicate<T> {
    // 判断元素t是否符合条件：
    boolean test(T t);
}
```

`filter()` 除了常用于数值外，也可应用于任何Java对象。例如，从一组给定的 `LocalDate` 中过滤掉工作日，以便得到休息日：

```
import java.time.*;
import java.util.function.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Stream.generate(new LocalDateSupplier())
            .limit(31)
            .filter(ldt -> ldt.getDayOfWeek() == DayOfWeek.SATURDAY ||
                ldt.getDayOfWeek() == DayOfWeek.SUNDAY)
            .forEach(System.out::println);
    }
}

class LocalDateSupplier implements Supplier<LocalDate> {
    LocalDate start = LocalDate.of(2020, 1, 1);
    int n = -1;
    public LocalDate get() {
        n++;
        return start.plusDays(n);
    }
}
```

```
2020-01-04
2020-01-05
2020-01-11
2020-01-12
2020-01-18
2020-01-19
2020-01-25
2020-01-26
```

练习

请使用 `filter()` 过滤出成绩及格的同学，并打印出名字。

从  **gitee** 下载练习：[filter练习](#)（推荐使用[IDE练习插件](#)快速下载）

小结

使用 `filter()` 方法可以对一个 `Stream` 的每个元素进行测试，通过测试的元素被过滤后生成一个新的 `Stream`。

使用reduce

`map()` 和 `filter()` 都是 `Stream` 的转换方法，而 `Stream.reduce()` 则是 `Stream` 的一个聚合方法，它可以把一个 `Stream` 的所有元素按照聚合函数聚合成一个结果。

我们来看一个简单的聚合方法：

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

public class Main {
    public static void main(String[] args) {
        int sum = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).reduce(0, (acc, n) -> acc
+ n);
        System.out.println(sum); // 45
    }
}
```

45

`reduce()` 方法传入的对象是 `BinaryOperator` 接口，它定义了一个 `apply()` 方法，负责把上次累加的结果和本次的元素 进行运算，并返回累加的结果：

```
@FunctionalInterface
public interface BinaryOperator<T> {
    // Bi操作：两个输入，一个输出
    T apply(T t, T u);
}
```

上述代码看上去不好理解，但我们用 `for` 循环改写一下，就容易理解了：

```
Stream<Integer> stream = ...
int sum = 0;
for (n : stream) {
    sum = (sum, n) -> sum + n;
}
```

可见，`reduce()` 操作首先初始化结果为指定值（这里是0），紧接着，`reduce()` 对每个元素依次调用 `(acc, n) -> acc + n`，其中，`acc` 是上次计算的结果：

```
// 计算过程：
acc = 0 // 初始化为指定值
acc = acc + n = 0 + 1 = 1 // n = 1
acc = acc + n = 1 + 2 = 3 // n = 2
acc = acc + n = 3 + 3 = 6 // n = 3
acc = acc + n = 6 + 4 = 10 // n = 4
acc = acc + n = 10 + 5 = 15 // n = 5
acc = acc + n = 15 + 6 = 21 // n = 6
acc = acc + n = 21 + 7 = 28 // n = 7
acc = acc + n = 28 + 8 = 36 // n = 8
acc = acc + n = 36 + 9 = 45 // n = 9
```

因此，实际上这个 `reduce()` 操作是一个求和。

如果去掉初始值，我们会得到一个 `Optional<Integer>`：

```
Optional<Integer> opt = stream.reduce((acc, n) -> acc + n);
if (opt.isPresent()) {
    System.out.println(opt.get());
}
```

这是因为 `Stream` 的元素有可能是0个，这样就无法调用 `reduce()` 的聚合函数了，因此返回 `Optional` 对象，需要进一步判断结果是否存在。

利用 `reduce()`，我们可以把求和改成求积，代码也十分简单：

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

public class Main {
    public static void main(String[] args) {
        int s = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).reduce(1, (acc, n) -> acc * n);
        System.out.println(s); // 362880
    }
}
```

362880

注意：计算求积时，初始值必须设置为 1。

除了可以对数值进行累积计算外，灵活运用 `reduce()` 也可以对Java对象进行操作。下面的代码演示了如何将配置文件的每一行配置通过 `map()` 和 `reduce()` 操作聚合成一个 `Map<String, String>`：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // 按行读取配置文件：
        List<String> props = List.of("profile=native", "debug=true",
            "logging=warn", "interval=500");
        Map<String, String> map = props.stream()
            // 把k=v转换为Map[k]=v：
            .map(kv -> {
                String[] ss = kv.split("\\=", 2);
                return Map.of(ss[0], ss[1]);
            })
            // 把所有Map聚合到一个Map：
            .reduce(new HashMap<String, String>(), (m, kv) -> {
                m.putAll(kv);
                return m;
            });
        // 打印结果：
        map.forEach((k, v) -> {
            System.out.println(k + " = " + v);
        });
    }
}
```

```
logging = warn
interval = 500
debug = true
profile = native
```


小结

`reduce()` 方法将一个 `Stream` 的每个元素依次作用于 `BinaryOperator`，并将结果合并。

`reduce()` 是聚合方法，聚合方法会立刻对 `Stream` 进行计算。

输出集合

我们介绍了 `Stream` 的几个常见操作：`map()`、`filter()`、`reduce()`。这些操作对 `Stream` 来说可以分为两类，一类是转换操作，即把一个 `Stream` 转换为另一个 `Stream`，例如 `map()` 和 `filter()`，另一类是聚合操作，即对 `Stream` 的每个元素进行计算，得到一个确定的结果，例如 `reduce()`。

区分这两种操作是非常重要的，因为对于 `Stream` 来说，对其进行转换操作并不会触发任何计算！我们可以做个实验：

```
import java.util.function.Supplier;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        Stream<Long> s1 = Stream.generate(new NatualSupplier());
        Stream<Long> s2 = s1.map(n -> n * n);
        Stream<Long> s3 = s2.map(n -> n - 1);
        System.out.println(s3); // java.util.stream.ReferencePipeline$3@49476842
    }
}

class NatualSupplier implements Supplier<Long> {
    long n = 0;
    public Long get() {
        n++;
        return n;
    }
}
```

```
java.util.stream.ReferencePipeline$3@358ee631
```

因为 `s1` 是一个 `Long` 类型的序列，它的元素高达922亿个，但执行上述代码，既不会有任何内存增长，也不会有任何计算，因为转换操作只是保存了转换规则，无论我们对一个 `Stream` 转换多少次，都不会有任何实际计算发生。

而聚合操作则不一样，聚合操作会立刻促使 `Stream` 输出它的每一个元素，并依次纳入计算，以获得最终结果。所以，对一个 `Stream` 进行聚合操作，会触发一系列连锁反应：

```
Stream<Long> s1 = Stream.generate(new NatualSupplier());
Stream<Long> s2 = s1.map(n -> n * n);
Stream<Long> s3 = s2.map(n -> n - 1);
Stream<Long> s4 = s3.limit(10);
s4.reduce(0, (acc, n) -> acc + n);
```

我们对 `s4` 进行 `reduce()` 聚合计算，会不断请求 `s4` 输出它的每一个元素。因为 `s4` 的上游是 `s3`，它又会向 `s3` 请求元素，导致 `s3` 向 `s2` 请求元素，`s2` 向 `s1` 请求元素，最终，`s1` 从 `Supplier` 实例中请求到真正的元素，并经过一系列转换，最终被 `reduce()` 聚合出结果。

可见，聚合操作是真正需要从 `Stream` 请求数据的，对一个 `Stream` 做聚合计算后，结果就不是一个 `Stream`，而是一个其他的Java对象。

输出为List

`reduce()` 只是一种聚合操作，如果我们希望把 `Stream` 的元素保存到集合，例如 `List`，因为 `List` 的元素是确定的Java对象，因此，把 `Stream` 变为 `List` 不是一个转换操作，而是一个聚合操作，它会强制 `Stream` 输出每个元素。

下面的代码演示了如何将一组 `String` 先过滤掉空字符串，然后把非空字符串保存到 `List` 中：

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

public class Main {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("Apple", "", null, "Pear", " ",
"Orange");
        List<String> list = stream.filter(s -> s != null &&
!s.isBlank()).collect(Collectors.toList());
        System.out.println(list);
    }
}
```

```
[Apple, Pear, Orange]
```

把 `Stream` 的每个元素收集到 `List` 的方法是调用 `collect()` 并传入 `Collectors.toList()` 对象，它实际上是一个 `Collector` 实例，通过类似 `reduce()` 的操作，把每个元素添加到一个收集器中（实际上是 `ArrayList`）。

类似的，`collect(Collectors.toSet())` 可以把 `Stream` 的每个元素收集到 `Set` 中。

输出为数组

把 `Stream` 的元素输出为数组和输出为 `List` 类似，我们只需要调用 `toArray()` 方法，并传入数组的“构造方法”：

```
List<String> list = List.of("Apple", "Banana", "Orange");
String[] array = list.stream().toArray(String[]::new);
```

注意到传入的“构造方法”是 `String[]::new`，它的签名实际上是 `IntFunction<String[]>` 定义的 `String[] apply(int)`，即传入 `int` 参数，获得 `String[]` 数组的返回值。

输出为Map

如果我们要把 `Stream` 的元素收集到 `Map` 中，就稍微麻烦一点。因为对于每个元素，添加到 `Map` 时需要 `key` 和 `value`，因此，我们要指定两个映射函数，分别把元素映射为 `key` 和 `value`：

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

public class Main {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("APPL:Apple", "MSFT:Microsoft");
        Map<String, String> map = stream
```

```

        .collect(Collectors.toMap(
            // 把元素s映射为key:
            s -> s.substring(0, s.indexOf(':')),
            // 把元素s映射为value:
            s -> s.substring(s.indexOf(':') + 1)));
    System.out.println(map);
}
}

```

```
{MSFT=Microsoft, APPL=Apple}
```

分组输出

`Stream` 还有一个强大的分组功能，可以按组输出。我们看下面的例子：

```

import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("Apple", "Banana", "Blackberry", "Coconut",
            "Avocado", "Cherry", "Apricots");
        Map<String, List<String>> groups = list.stream()
            .collect(Collectors.groupingBy(s -> s.substring(0, 1),
            Collectors.toList()));
        System.out.println(groups);
    }
}

```

```
{A=[Apple, Avocado, Apricots], B=[Banana, Blackberry], C=[Coconut, Cherry]}
```

分组输出使用 `Collectors.groupingBy()`，它需要提供两个函数：一个是分组的key，这里使用 `s -> s.substring(0, 1)`，表示只要首字母相同的 `String` 分到一组，第二个是分组的value，这里直接使用 `Collectors.toList()`，表示输出为 `List`，上述代码运行结果如下：

```

{
    A=[Apple, Avocado, Apricots],
    B=[Banana, Blackberry],
    C=[Coconut, Cherry]
}

```

可见，结果一共有3组，按 "A"，"B"，"C" 分组，每一组都是一个 `List`。

假设有这样一个 `Student` 类，包含学生姓名、班级和成绩：

```

class Student {
    int gradeId; // 年级
    int classId; // 班级
    String name; // 名字
    int score; // 分数
}

```

如果我们有一个 `Stream<Student>`，利用分组输出，可以非常简单地按年级或班级把 `Student` 归类。

小结

`Stream` 可以输出为集合：

`Stream` 通过 `collect()` 方法可以方便地输出为 `List`、`Set`、`Map`，还可以分组输出。

其他操作

我们把 `Stream` 提供的操作分为两类：转换操作和聚合操作。除了前面介绍的常用操作外，`Stream` 还提供了一系列非常有用的方法。

排序

对 `Stream` 的元素进行排序十分简单，只需调用 `sorted()` 方法：

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

public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("Orange", "apple", "Banana")
            .stream()
            .sorted()
            .collect(Collectors.toList());
        System.out.println(list);
    }
}
```

```
[Banana, Orange, apple]
```

此方法要求 `Stream` 的每个元素必须实现 `Comparable` 接口。如果要自定义排序，传入指定的 `Comparator` 即可：

```
List<String> list = List.of("Orange", "apple", "Banana")
    .stream()
    .sorted(String::compareToIgnoreCase)
    .collect(Collectors.toList());
```

注意 `sorted()` 只是一个转换操作，它会返回一个新的 `Stream`。

去重

对一个 `Stream` 的元素进行去重，没必要先转换为 `Set`，可以直接用 `distinct()`：

```
List.of("A", "B", "A", "C", "B", "D")
    .stream()
    .distinct()
    .collect(Collectors.toList()); // [A, B, C, D]
```

截取

截取操作常用于把一个无限的 `Stream` 转换成有限的 `Stream`，`skip()` 用于跳过当前 `Stream` 的前N个元素，`limit()` 用于截取当前 `Stream` 最多前N个元素：

```
List.of("A", "B", "C", "D", "E", "F")
    .stream()
    .skip(2) // 跳过A, B
    .limit(3) // 截取C, D, E
    .collect(Collectors.toList()); // [C, D, E]
```

截取操作也是一个转换操作，将返回新的 `Stream`。

合并

将两个 `Stream` 合并为一个 `Stream` 可以使用 `Stream` 的静态方法 `concat()`：

```
Stream<String> s1 = List.of("A", "B", "C").stream();
Stream<String> s2 = List.of("D", "E").stream();
// 合并：
Stream<String> s = Stream.concat(s1, s2);
System.out.println(s.collect(Collectors.toList())); // [A, B, C, D, E]
```

flatMap

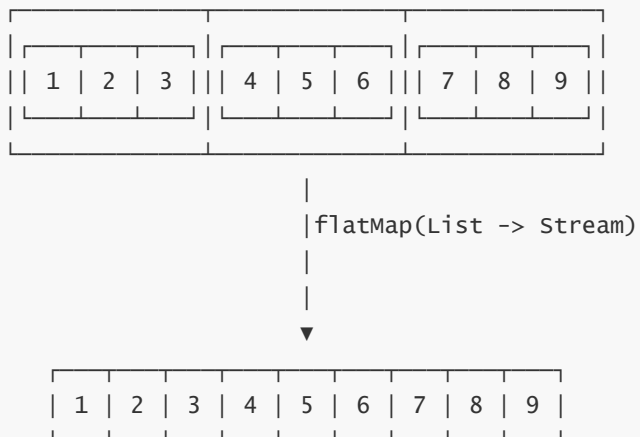
如果 `Stream` 的元素是集合：

```
Stream<List<Integer>> s = Stream.of(
    Arrays.asList(1, 2, 3),
    Arrays.asList(4, 5, 6),
    Arrays.asList(7, 8, 9));
```

而我们希望把上述 `Stream` 转换为 `Stream<Integer>`，就可以使用 `flatMap()`：

```
Stream<Integer> i = s.flatMap(list -> list.stream());
```

因此，所谓 `flatMap()`，是指把 `Stream` 的每个元素（这里是 `List`）映射为 `Stream`，然后合并成一个新的 `Stream`：



并行

通常情况下，对 `Stream` 的元素进行处理是单线程的，即一个一个元素进行处理。但是很多时候，我们希望可以并行处理 `Stream` 的元素，因为在元素数量非常大的情况，并行处理可以大大加快处理速度。

把一个普通 `Stream` 转换为可以并行处理的 `Stream` 非常简单，只需要用 `parallel()` 进行转换：

```
Stream<String> s = ...
String[] result = s.parallel() // 变成一个可以并行处理的Stream
                    .sorted() // 可以进行并行排序
                    .toArray(String[]::new);
```

经过 `parallel()` 转换后的 `Stream` 只要可能，就会对后续操作进行并行处理。我们不需要编写任何多线程代码就可以享受到并行处理带来的执行效率的提升。

其他聚合方法

除了 `reduce()` 和 `collect()` 外，`Stream` 还有一些常用的聚合方法：

- `count()`：用于返回元素个数；
- `max(Comparator<? super T> cp)`：找出最大元素；
- `min(Comparator<? super T> cp)`：找出最小元素。

针对 `IntStream`、`LongStream` 和 `DoubleStream`，还额外提供了以下聚合方法：

- `sum()`：对所有元素求和；
- `average()`：对所有元素求平均数。

还有一些方法，用来测试 `Stream` 的元素是否满足以下条件：

- `boolean allMatch(Predicate<? super T>)`：测试是否所有元素均满足测试条件；
- `boolean anyMatch(Predicate<? super T>)`：测试是否至少有一个元素满足测试条件。

最后一个常用的方法是 `forEach()`，它可以循环处理 `Stream` 的每个元素，我们经常传入 `System.out::println` 来打印 `Stream` 的元素：

```
Stream<String> s = ...
s.forEach(str -> {
    System.out.println("Hello, " + str);
});
```

小结

`Stream` 提供的常用操作有：

转换操作： `map()`，`filter()`，`sorted()`，`distinct()`；

合并操作： `concat()`，`flatMap()`；

并行处理： `parallel()`；

聚合操作： `reduce()`，`collect()`，`count()`，`max()`，`min()`，`sum()`，`average()`；

其他操作： `allMatch()`，`anyMatch()`，`forEach()`。

