目　录

致谢

当前文档 《Java8 教程》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-05-09。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/junicorn-java8-tutorial

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Java8 教程

Java8 教程

引言

这个 `java8` 教程列表中将分享那些 `java8` 中最重要的那些功能和链接。原文地址 https://howtodoinjava.com/java-8-tutorial/
如果觉得这些内容可以帮助到你，也可以在 github 上点一个?支持一下~

阅读地址

- 在线阅读
- PDF/暂未提供

参与翻译

- 如何参与翻译

感谢以下译者

- @王爵nice
- @huhaifan
- @xingfly
- @Cailei Lu
- @effectivefish
- @rainoflisten
- @jiang85991
- @achilleskwok

License

来源(书栈小编注)

> *https://github.com/junicorn/java8-tutorial*

Java8 - 介绍

Java8 介绍

Java8 在 2014年初 发布，在 Java8 中大家讨论最多的特性是 lambda 表达式。
它还有许多重要的功能，像默认方法、Stream API、新的日期时间 API。让我们通过示例来了解这些新功能。

Lambda 表达式

有许多使用过高级编程语言（比如Scala）的人不知道 lambda 表达式。在编程中，lambda 表达式（或者函数）只是一个匿名函数，即一个没有名称也没有标识符的函数。它们都被写在你需要使用的地方，通常作为其他函数的参数。

lambda 表达式的基本语法是:

```
1. either
2. (parameters) -> expression
```

```
3. or
4. (parameters) -> { statements; }
5. or
6. () -> expression
```

一个典型的 `lambda` 表达式如下所示：

```
1. (x, y) -> x + y  //这个函数接受两个参数并返回它们的和。
```

请注意根据 `x` 和 `y` 的类型，方法可能会在多个地方使用。参数可以匹配到 `int` 类型整数或者字符串类型。
根据上下文，它将两个整数相加或者两个字符串拼接。

编写 lambda 表达式的规则

1. lambda表达式可以有零个、一个或多个参数
2. 可以显式声明参数的类型，也可以从上下文推断参数的类型
3. 多个参数必须包含在括号中，并用逗号分隔，空括号用于表示零个参数
4. 当只有一个参数时，如果推断它的类型，可以不使用括号。如 `a -> return a * a`
5. lambda表达式的函数体可以包含零个，一个或多个语句。
6. 如果lambda表达式的函数体只有一行，则可以不用大括号，匿名函数的返回类型与函数体表达式的返回类型相同。当函数体中大于一行代码则需要用大括号包含。

> 阅读更多：*Java 8 Lambda 表达式*

函数式接口（Functional Interface）

`Functional Interface` 也被称为单一抽象方法的接口（SAM接口）。就像它的名字一样，

它 只能有一个抽象方法 。在 Java8 种引入了一个注解 `@FunctionalInterface` ，
当你在某个接口使用了该注解，接口违反了函数式接口的规定时，会产生编译报错。

一个函数式接口的示例：

```
1.  @FunctionalInterface
2.  public interface MyFirstFunctionalInterface {
3.      public void firstWork();
4.  }
```

请注意，即使省略 `@FunctionalInterface` 注解，函数式接口也是有效的。它仅用于通知编译器在接口内强制定义单个抽象方法。

此外，由于默认方法不是抽象的，您可以随意添加默认方法到你的接口中，只要你喜欢。

另一个要记住的要点是，如果一个接口中定义的函数是重写了 `Object` 类的方法，
那么这个方法不会被计入接口的抽象方法，因为接口的任何实现类都会继承自 `Object` 。
例如，下面是完全有效的函数式接口。

```
1.  @FunctionalInterface
2.  public interface MyFirstFunctionalInterface{
3.
4.      public void firstWork();
5.
6.      @Override
7.      public String toString();                //重写toString
8.
9.      @Override
10.     public boolean equals(Object obj);       //重写equals
```

```
11. }
```

> 阅读更多：函数式接口

默认方法（Default Methods）

Java8 允许你在接口中添加非抽象方法。这些方法必须被声明为 默认方法。java8中引入了默认方法来启用 lambda 表达式的功能。

默认方法可以让你在接口中添加新功能，并确保与旧版本的这些接口编写的代码的二进制兼容。

我们以一个例子来理解：

```
1. public interface Moveable {
2.     default void move(){
3.         System.out.println("I am moving");
4.     }
5. }
```

`Moveable` 接口定义了一个方法 `move` 并且提供了默认实现。如果任何类实现了这个接口，
那么它可以不实现 `move` 方法。子类可以直接调用 `move` 方法，例如：

```
1. public class Animal implements Moveable{
2.     public static void main(String[] args){
3.         Animal tiger = new Animal();
4.         tiger.move();
5.     }
6. }
7.
8. 输出: I am moving
```

如果子类愿意定制 `move` 方法的行为，那么它可以提供它自己的自定

义实现并覆盖该方法。

> 阅读更多：默认方法

Streams

另一个重大变化是引入了 **Streams API**，它提供了一种可以用各种方式处理一组数据的机制，包括过滤、转换或任何其他可能对应用程序有用的方式。

Java8中的 `Streams API` 支持不同类型的迭代，你只需定义要处理的项目集合，对每个项目执行的操作以及要存储操作后的输出。

下面是一个 `Stream API` 的例子。在这个示例中，`items` 是 `String` 值的集合，你需要筛选出一些不是 `prefix` 开头的项目。

```
1.  List<String> items;
2.  String prefix = "base";
3.  List<String> filteredList = items.stream()
4.                      .filter(e -> (!e.startsWith(prefix)))
5.                      .collect(Collectors.toList());
```

这里的 `items.stream()` 表示我们希望使用 `Streams API` 处理集合中的数据。

> 阅读更多：内部迭代 *VS* 外部迭代

时间日期API的改变

新的日期和时间API（JSR-310）也称为*ThreeTen*，它们简单地改变了在java应用程序中处理日期的方式。

日期相关

`Date` 类甚至已经过时了。用于替换 `Date` 类的新类是 `LocalDate`，`LocalTime` 和 `LocalDateTime`。

1. `LocalDate` 表示没有时区的日期
2. `LocalTime` 表示没有时区的时间
3. `LocalDateTime` 表示没有时区的日期时间

如果要使用带有时区信息的日期功能，Java8为您提供了类似于上述的3个类，
即 `OffsetDate`，`OffsetTime` 和 `OffsetDateTime`。
时区偏移可以用 `05:30` 或 `Europe/Paris` 格式表示。
这是通过使用另一个类 `ZoneId` 来完成的。

时间戳和周期

为了表示任何时刻的具体时间戳，需要使用的类是 `Instant`。
`Instant` 类代表一个即时的时间精度为纳秒。即时操作包括与另一个时间的比较，添加或减去一个周期的时间。

```
1.  Instant instant = Instant.now();
2.  Instant instant1 = instant.plus(Duration.ofMillis(5000));
3.  Instant instant2 = instant.minus(Duration.ofMillis(5000));
4.  Instant instant3 = instant.minusSeconds(10);
```

`Duration` 类是Java语言中首次引入的概念。它用来代表两个时间戳之间的时差。

```
1.  Duration duration = Duration.ofMillis(5000);
2.  duration = Duration.ofSeconds(60);
3.  duration = Duration.ofMinutes(10);
```

`Duration` 一般处理小的时间单位，如毫秒，秒，分钟和小时。它们

在程序代码中的交互更多一些。
要与人交互，你需要操作 更长的时间，这些更长的时间在 `Period` 类中可以体现。

```
1. Period period = Period.ofDays(6);
2. period = Period.ofMonths(6);
3. period = Period.between(LocalDate.now(),
   LocalDate.now().plusDays(60));
```

> 阅读更多：日期时间*API*的改变

原文出处：http://howtodoinjava.com/java-8-tutorial

Java8 - lambda表达式

lambda表达式

Java8 - 方法引用

方法引用

在 Java8 中，你可以使用 `class::methodName` 语法引用类或对象的方法。让我们来学习一下 Java 8 中不同类型的方法引用。

方法引用类型 - 快速预览

Java8 中包含了四种类型的方法引用。

方法引用	描述	例子
静态方法引用	用于引用类的静态方法	`Math::max` 相当于 `Math.max(x,y)`
从对象中引用实例方法	使用对象的引用来调用实例方法	`System.out::println` 相当于 `System.out.println(x)`
从类中引用实例方法	在上下文提供的对象的引用上调用实例方法	`String::length` 相当于 `str.length()`
引用构造函数	引用构造函数	`ArrayList::new``` 相当于 new ArrayList()`

引用静态方法 - Class::staticMethodName

一个使用 `Math.max()` 静态方法的例子。

```
List<Integer> integers = Arrays.asList(1,12,433,5);

Optional<Integer> max = integers.stream().reduce( Math::max );

max.ifPresent(value -> System.out.println(value));
```

输出：

```
433
```

从对象中引用实例方法 - ClassInstance::instanceMethodName

在上面的例子中，我们使用了 `System.out.println(value)` 打印集合中的最大值，我们可以使用 `System.out::println` 打印这个值。

```
List<Integer> integers = Arrays.asList(1,12,433,5);
Optional<Integer> max = integers.stream().reduce( Math::max );
max.ifPresent( System.out::println );
```

输出：

```
433
```

引用特定类型的实例方法 - Class::instanceMethodName

在这个例子中 `s1.compareTo(s2)` 被简写为 `String::compareTo` 。

```
List<String> strings = Arrays
        .asList("how", "to", "do", "in", "java", "dot", "com");

List<String> sorted = strings
```

```
        .stream()
        .sorted((s1, s2) -> s1.compareTo(s2))
        .collect(Collectors.toList());

System.out.println(sorted);

List<String> sortedAlt = strings
        .stream()
        .sorted(String::compareTo)
        .collect(Collectors.toList());

System.out.println(sortedAlt);
```

输出：

```
[com, do, dot, how, in, java, to]
[com, do, dot, how, in, java, to]
```

引用构造函数 - Class::new

使用 `lambda` 表达式修改第一个例子中的方法，可以非常简单的创建一个从1到100的集合(不包含100)。创建一个新的 `ArrayList` 实例，我们可以使用 `ArrayList::new` 。

```
List<Integer> integers = IntStream
                .range(1, 100)
                .boxed()
                .collect(Collectors.toCollection( ArrayList::new ));

Optional<Integer> max = integers.stream().reduce(Math::max);

max.ifPresent(System.out::println);
```

输出：

```
1. 99
```

这是Java8 lambda 增强功能中的四种方法引用。

学习愉快！

原文出处：https://howtodoinjava.com/java-8/lambda-method-references-example

Java8 - 默认方法

默认方法

`java 8` 允许你在接口中添加非抽象方法 ， 这些方法必须定义为 `default` 方法 ， `java 8` 引入默认方法是为了使用 `lambda` 函数表达式 .

`default` 方法使您能够向库的接口添加新功能，并确保与旧版本的这些接口编写的代码的二进制兼容性 。

用一个例子去理解:

```
1.     public interface Moveable {
2.             default void move(){
3.                 System.out.println("I am moving");
4.             }
5.     }
```

`Moveable` 接口定义了一个 `move()` 方法并且提供了默认的实现 . 如果任意一个 `class` 实现这个
接口都没有必要去实现这个 `move()` 方法，能够直接使用 `instance.move()` 进行调用 .

eg:

```
1.  public class Animal implements Moveable{
2.      public static void main(String[] args){
3.          Animal tiger = new Animal();
4.          tiger.move();
5.      }
6.  }
```

```
7.  Output: I am moving
```

如果该 `class` 想要自定义一个 `move()` 也能提供自定义的实现去覆写这个 `move` 方法 .

原文 : https://howtodoinjava.com/java-8-tutorial/

read more : https://howtodoinjava.com/java-8/java-8-tutorial-internal-vs-external-iteration/

Java8 - 函数式接口

函数式接口

`java 8` 提供了函数式接口，该接口同样也被称为 单例抽象方法接口(*SAM* 接口) . 如名字所示，接口中只允许一个抽象方法 . `java 8` 引入了 `@FunctionalInterface` 注解 , `@FunctionalInterface` 会在你注解的接口违反了函数式接口的约定时提示一个编译错误 .

一个典型的 `函数式接口` 例子:

```
1. @FunctionalInterface
2. public interface MyFirstFunctionalInterface {
3.     public void firstWork();
4. }
```

不过请注意,即使没有写 `@FunctionalInterface` 这个注解 , 这个函数式接口依旧是合理的 . 因为它仅仅是用来通知编译器确保接口内部只有一个抽象方法 .

同样,因为默认方法不是抽象方法,因此只要你喜欢,你就能自由的添加默认方法到你的函数式接口当中.
另一个重要的点就是记住如果一个接口申明的抽象方法是重载了一个 `java.lang.Object` 的一个公共方法 , 同样不会计算在接口的抽象方法数目当中因为该接口的任意实现都会实现 `java.lang.Object` 或者在别处实现.

一个合理而有效的函数式接口.

```java
@FunctionalInterface
public interface MyFirstFunctionalInterface{
    public void firstWork();
    @Override
    public String toString();                //重写自 Object 类
    @Override
    public boolean equals(Object obj);        //重写自 Object 类
}
```

原文出处：https://howtodoinjava.com/java-8/functional-interface-tutorial/

Java8 - Optional

Optional

我们在写程序时一定都遇到过 `空指针异常` 。它通常发生在你试图去引用一个没有被初始化，被初始化为空或者没有指向任何实例的对象的情况下。空仅仅只是意味着没有值，最有可能的是，罗马人是唯一没有遇到过空，从 1 开始计数 1,2,3…(没有0) 的人，或许，他们没有模拟市场上没有苹果的情况。[:-)]

“我叫它价值十亿美元的错误” —— Sir C. A. R. Hoare，他在提出空引用这个概念时说。

在本文中，我将讨论 java 8 新特性 中 Optional 的具体用法。为了阐明诸多概念之间的异同，本文将被划分为多个部分一一讲解。

```
1.  讨论的要点
2.
3.  1）什么是null类型？
4.  2）返回一个null值有什么问题？
5.  3）Java8中的Optional类提供了一种怎样的解决办法？
6.   a）创建一个Optional对象
7.   b）若值存在则进行一些操作
8.   c）默认或者缺省的值和行为
9.   d）使用过滤器方法拒绝一些特殊值
10.  4）Optional内部是如何工作的？
11.  5）Optional试图去解决什么问题？
12.  6）Optional不能解决什么问题？
13.  7）Optional应该怎样被使用呢？
14.  8）最后的结论
```

1）什么是Null类型？

在java中，我们通过引用去访问一个对象，当我们的的引用没有指向一个具体的对象时，我们就会把这个引用置为Null，用于标识该引用没有值，对吧？

虽然null值的使用非常普遍，但是我们很少会去仔细地研究它。例如一个类的成员变量会被自动初始化为null，程序员通常会给没有初始值的引用类型置null，总体上说，null值主要用于我们不知道具体的或者没有具体指向的引用类型。

1. 补充，在java中，null是一个具体而又特殊的数据类型，因为它没有名字，所以我们不能声明一个类型为null的变量或者将一个类型
2. 强转为null；实际上，仅仅有一个能和null有关联的值（例如。。。）。请记住，null不像java中的其他类型，一个空引用能
3. 完全正确，安全地赋给其他的任何引用类型，而且不报任何错。

2）返回null值有什么问题？

通常来说，API的设计者会附上详细的文档，并说明在某种情况下，这个API会返回一个null值，现在问题是，因为一些原因，我们在调用API的时候可能不会去读API文档，然后就忘记了处理**null**值的情况，这在未来一定会是一个bug。

相信我，这是经常发生的，而且是空指针异常的主要原因之一，虽然并不是唯一的原因。因此，请牢记，在第一次调用API时看看它的文档说明（。。至少）[:-)]。

1. 现在我们知道了在大多数情况下，null值是一个问题，那么处理null值最好的方式是什么呢？

一个比较好的解决方案是在定义引用类型时总是去将它初始化为某个值，并且不要初始化为null。采用这种方式，你的程序永远不会抛出 `空指针异常` 。但是有些情况下，引用并没有一个默认的初始值，那么

我们又该怎么去处理呢？

上述的解决方案在很多情况下是行得通的。然而，Java 8 `Optionals` 无疑是最好的方案。

3）Java 8的Optionals提供了一种怎样的解决方案？

Optional提供了一种可以替代一个指向非空值但是可以为空T的方法。一个Optional类或许包含了一个指向非空变量的引用（这种情况下我们说这个引用是Present的），或者什么也没包含（这种情况下我们说这个引用是absent）。

> 请记住，永远不要说*optional*包含*null*

```
1.  Optional<Integer> canBeEmpty1 = Optional.of(5);
2.  canBeEmpty1.isPresent();                    // returns true
3.  canBeEmpty1.get();                          // returns 5
4.
5.  Optional<Integer> canBeEmpty2 = Optional.empty();
6.  canBeEmpty2.isPresent();                    // returns false
```

我们可以看到 **Optional**作为保存单变量的容器，既可以保存值，也可以不保存值。

必须要指出Optional类并不是为了简单地去替代每一个null引用。它的出现是为了帮助设计更易于理解的**API**，以便我们只需要读方法的声明，就能分辨是否是一个Optional变量。这迫使你去捕获Optional中的变量，然后处理它，同时也处理optional为空的情形。这才是解决因为空引用导致的 `NullPointerException` 的正确之道。

下面是一些学习在编程中如何使用Optional的例子。

a）创建一个Optional对象
创建一个Optional对象有三种方式。

i)使用Optional.empty()方法创建一个空的Optional。

```
1.  Optional<Integer> possible = Optional.empty();
```

ii）使用Optional.of()方法创建一个带有非空默认值的对象，如果你赋值为null，则会抛出空指针异常。

```
1.  Optional<Integer> possible = Optional.of(5);
```

iii)使用Optional.ofNullable()创建一个可以为null值的对象。如果值是null，则目标Optional对象将会是空(记得值是absent，不是null)。

```
1.  Optional<Integer> possible = Optional.ofNullable(null);
2.  //or
3.  Optional<Integer> possible = Optional.ofNullable(5);
```

b)如果Optional的值是present的，来做一些操作吧
得到Optional对象是第一步。现在我们先来检查一下它的内部是否保存了值，然后来使用它。

```
1.  Optional<Integer> possible = Optional.of(5);
2.  possible.ifPresent(System.out::println);
```

你可以像下面的代码一样重写你的代码。然而这并不是Optional类的最佳实践，因为这样做和手动检查是否为null没有任何区别，没有任何提升。

```
1.  if(possible.isPresent()){
2.  System.out.println(possible.get());
3.  }
```

如果Optional对象是空的，不会输出任何东西。

c)默认或者缺省的值和动作

一个典型的编程模式是如果操作的结果是null，那么最好返回一个默认的值。简要地说，你可以设置缺省操作。但是如果使用Optional，你可以像下面那样写你的代码。

```
1.  //Assume this value has returned from a method
2.  Optional<Company> companyOptional = Optional.empty();
3.
4.  //Now check optional; if value is present then return it,
5.  //else create a new Company object and retur it
6.  Company company = companyOptional.orElse(new Company());
7.
8.  //OR you can throw an exception as well
9.  Company company =
    companyOptional.orElseThrow(IllegalStateException::new);
```

d)使用过滤方法拒绝某些特定的值

我们经常调用某个类中的方法去检查一些属性，例如，在下面的示例代码检查Company类是否有 “Finance”部门；如果有，则输出其值。

```
1.  Optional<Company> companyOptional = Optional.empty();
2.  companyOptional.filter(department ->
    "Finance".equals(department.getName())
3.                  .ifPresent(() -> System.out.println("Finance is
    present"));
```

这个过滤方法起了论断的作用。如果Optional对象中值是Present的，并且符合了这个论断，这个过滤方法返回这个值；否则，它返回一个空的Optional对象，你可能已经看出来了，当我们使用过滤方法时，其实和Stream的模式是相似的。

很好，这些代码看起来更接近我们提出问题的最佳答案，再也没有冗余的检查是否为空的代码。

Wow，我们已经从写令人烦闷的非null检查的路上迈出了一大步，现在我们可以写更加易于理解，可读性更高的代码，而且不会有空指针异常。

4）Optional类内部是如何实现的呢？

如果你打开Optional类的源代码，你将会看到如下定义：

```
1.  /**
2.   * If non-null, the value; if null, indicates no value is present
3.   */
4.  private final T value;
```

如果你定义一个空的Optional，就像下面这样，**static**关键字保证了只存在一个空的实例。

```
1.  /**
2.   * Common instance for {@code empty()}.
3.   */
4.  private static final Optional<?> EMPTY = new Optional<>();
```

默认的没有参数的构造函数被定义成private的，所以除了上面的三种方式，不能通过其他的方式创建一个实例。

当你创建一个Optional类的对象时，下面的函数会被调用，然后将确定的值传递给value属性。

```
1.  this.value = Objects.requireNonNull(value);
```

当你试图从Optional容器中获取值时，值如果存在就会被返回，否则抛出 `NoSuchElementException` 异常：

```
1.  public T get() {
2.  if (value == null) {
```

```
3.      throw new NoSuchElementException("No value present");
4.  }
5.  return value;
6.  }
```

同样的，定义在Optional类中其他的函数也只操作value属性，点击 这里 查看Optional类的源代码。

5）Optional类的出现是为了解决什么问题？

Optional试图解决在java整个系统中出现空指针异常的次数，通过设计考虑了返回值可能为null的更具表达意义的API接口来实现。如果Optional类在java设计之初就已经存在了，那么时下的一些函数库和程序可能能更好地处理返回值为null的情况，减少出现空指针异常的次数，总体上减少一些bug。

通过使用Optional类，用户不得不去考虑一些异常情况，除了通过给null一个名字而带来的程序可读性的提高，最大的好处就是 如果你想让你的程序通过编译，你就不得不去考虑值为null的情况。

6）Optional不能解决什么问题？

Optional类并不是一种可以避免所有空指针的的机制。例如：一个方法或者构造函数中必须的参数就必须做一下检测。

当使用null时，Optional不会表现出值缺失的意思。所以当你调用一个方法时，为了正确地调用，请查看相关的文档，确保你知道Optional类中值缺失的意思。

请确保在下列情况下不要使用Optional，因为并不能给我们带来任何实质性的帮助：

- in the domain model layer (it’s not serializable)

- in DTOs (it’s not serializable)
- in input parameters of methods
- in constructor parameters

7）我们应该使用Optional吗？

当函数的返回值有可能是null时，我们应该尽可能地去使用它。

下面是从OpenJDK中摘抄的一段文字：
The JSR-335 EG felt fairly strongly that Optional should not be on any more than needed to support the optional-return idiom only.
Someone suggested maybe even renaming it to “OptionalReturn“.

这段文字明确地表示，当从具体的业务逻辑层，数据库访问层或者实体中返回值时应该使用Optional，当然前提是Optional能起到作用。

8)结论

在这篇文章中，我们学习到了如何理解和使用java SE8中java.util包下Optional类。Optional存在的目的不是为了替代我们代码中简单的null引用，而是帮助我们设计更清晰明了的API，通过阅读方法的说明，使用者就能知道有Optional类，可能存在null的情况，从而恰当地去处理它。

这就是这个厉害特性的所有内容，在下面的评论区和我交流你的想法吧。

尽情享受学习的乐趣吧！！

Java8 - 如何使用Predicate

如何使用Predicate

Java8 - 日期时间API的改变

日期时间API的改变

Java8 - Stream示例

Stream 示例

在上一篇文章,我们学习了关于“内部迭代VS外部迭代”

以及内部迭代是如何有助于业务逻辑，而不是如何迭代某些需要被应用逻辑的对象。
它有助于写出更整洁并且更具有可读性的代码。
在本章节，我们将会讨论一个新的抽象概念 —— “Streams”

> 注意：*Streams* 可以被定义为元素集合从源支持聚合操作

这里的源指的是向 Stream 提供数据的集合或数组。Stream 保持原始数据的顺序，
并且聚合操作和批量操作可以使我们容易且清晰地对这些值表达常见的操作。

我将在以下部分讨论 Java 8 中 Streams 的各个方面：

> *Streams vs. Collections*
> 不同的方式构建 *Streams*
> 将 *Streams* 转换成 *Collections*
> *Stream*核心操作
> 中间操作
> 终端操作
> 短路操作
> 并行

在进行之前，了解 Java8 中 Streams 被设计成大多数流操作只返回 Streams 这一点很重要。
这有助于我们创建各种流操作链。这被称为流水线式操作(pipelining)，我将在这篇文章中多次使用这个词，
所以请记住。

Streams vs. Collections" class="reference-link">Streams vs. Collections

我们所有人都在YouTube或者其他此类网站上观看在线视频。当你开始

观看视频时，
文件的一小部分将先被加载到计算机中并开始播放。在开始播放之前，你不需要下载完整的视频。
这被称为流式传输。我将尝试将这个概念与 Collections 相关联，并与 Streams 相区分。

在基础层面上，Collections 和 Streams 之间的区别与被计算的对象有关。
集合是一种内存中的数据结构，它保存数据结构当前具有的所有值 —— 必须先计算集合中的每个元素，
然后才能将其添加到集合中。流是概念上固定的数据结构，其中的元素需要根据需要计算，
这产生了巨大的编程效益。这个想法是用户只能从流中提取所需的值，并且这些元素仅在需要时向用户生成 —— 对用户不可见。这是一种生产者-消费者关系的形式。

在java中，java.util.Stream表示为一个或多个可以执行操作的流。
流操作是中间或终端流程。当终端操作返回某种类型的结果时，中间操作将返回流本身，
以便可以将多个方法调用连接在一行。流是在源对象上创建的，例如像列表或集合（不支持maps）的java.util.Collection。
流操作可以串行执行或者并行执行。

基于上述几点，我们尝试列出Stream的各种特征：

- 不是数据结构
- 专为lambdas设计
- 不支持下标访问
- 可以容易地作为数组或列表输出

- 支持懒访问
- 可并行

不同的方式构建 Streams" class="reference-link">不同的方式构建 Streams

以下是几个较常用的方式将集合构建成流。

1)使用 Stream.of(val1, val2, val3….)

```
1. public class StreamBuilders {
2.      public static void main(String[] args){
3.          Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);
4.          stream.forEach(p -> System.out.println(p));
5.      }
6. }
```

2)使用 Stream.of(arrayOfElements)

```
1. public class StreamBuilders {
2.      public static void main(String[] args){
3.          Stream<Integer> stream = Stream.of( new Integer[]
   {1,2,3,4,5,6,7,8,9} );
4.          stream.forEach(p -> System.out.println(p));
5.      }
6. }
```

3) 使用 someList.stream()

```
1. public class StreamBuilders {
2.      public static void main(String[] args){
3.          List<Integer> list = new ArrayList<Integer>();
```

```
4.          for(int i = 1; i< 10; i++){
5.              list.add(i);
6.          }
7.          Stream<Integer> stream = list.stream();
8.          stream.forEach(p -> System.out.println(p));
9.      }
10. }
```

4) 使用 Stream.generate() 或者 Stream.iterate() functions

```
1. public class StreamBuilders {
2.      public static void main(String[] args){
3.          Stream<Date> stream = Stream.generate(() -> { return new
    Date();});
4.          stream.forEach(p -> System.out.println(p));
5.      }
6. }
```

5) 使用 String chars 或者 String tokens

```
1. public class StreamBuilders {
2.      public static void main(String[] args){
3.         IntStream stream = "12345_abcdefg".chars();
4.         stream.forEach(p -> System.out.println(p));
5.
6.         //OR
7.
8.         Stream<String> stream = Stream.of("A$B$C".split("\\$"));
9.         stream.forEach(p -> System.out.println(p));
10.     }
11. }
```

除了以上方法外还有其他方法使用 Stream ,构建者模式或者使用中间操作。

我们将会在不同的章节中了解它们。

将 Streams 转换成 Collections" class="reference-link">将 Streams 转换成 Collections

我本该说转换 Streams 到其他数据结构

> 注意：请不要说这是转变。这只是将 *Stream* 中的数据收集到 *Collection* 或数组中

1) 使用 stream.collect(Collectors.toList()) 将 Stream 转换成 List

```
1.  public class StreamBuilders {
2.       public static void main(String[] args){
3.           List<Integer> list = new ArrayList<Integer>();
4.           for(int i = 1; i< 10; i++){
5.               list.add(i);
6.           }
7.           Stream<Integer> stream = list.stream();
8.           List<Integer> evenNumbersList = stream.filter(i -> i%2 ==
    0).collect(Collectors.toList());
9.           System.out.print(evenNumbersList);
10.      }
11.  }
```

2) 使用 stream.toArray(EntryType[]::new) 将 Stream 转换成数组

```
1.  public class StreamBuilders {
2.       public static void main(String[] args){
3.           List<Integer> list = new ArrayList<Integer>();
4.           for(int i = 1; i< 10; i++){
```

```
            list.add(i);
        }
        Stream<Integer> stream = list.stream();
        Integer[] evenNumbersArr = stream.filter(i -> i%2 ==
0).toArray(Integer[]::new);
        System.out.print(evenNumbersArr);
    }
}
```

还有很多其他方式可以收集 Stream 进 Set、Map等结构。去熟悉下 Collectors 类并尽量记住他们。

Stream核心操作" class="reference-link">Stream核心操作

Stream 有一个很长且有用的功能列表。
我不会提到他们的全部，但我计划在这里列出最重要、你必须先了解的一些功能。

在前进之前，我们可以预先构建一个 String 集合。我们将在这个列表中建立一个用例，
这有助于我们更简单地去接触和理解。

```
List<String> memberNames = new ArrayList<>();
memberNames.add("Amitabh");
memberNames.add("Shekhar");
memberNames.add("Aman");
memberNames.add("Rahul");
memberNames.add("Shahrukh");
memberNames.add("Salman");
memberNames.add("Yana");
memberNames.add("Lokesh");
```

这些核心方法分为以下两部分：

中间操作" class="reference-link">中间操作

A) filter()

Filter 接受一个断言条件去过滤流的所有元素。
这个流的操作是中间的，
这使得我们能够对结果调用另外一个流操作（例如forEach）

```
1.  memberNames.stream().filter((s) -> s.startsWith("A"))
2.                      .forEach(System.out::println);
3.
4.  Output:
5.
6.  Amitabh
7.  Aman
```

B) map()

中间操作 map 通过给定的功能将每个元素转换成另一个对象。
以下示例将每个字符串转换成大写字符串。当然你也可以使用 map 将每个对象转换成另一种类型。

```
1.  memberNames.stream().filter((s) -> s.startsWith("A"))
2.                       .map(String::toUpperCase)
3.                       .forEach(System.out::println);
4.
5.  Output:
6.
7.  AMITABH
8.  AMAN
```

C) sorted()

Sorted 是一个中间操作，返回流的排序视图。元素按照自然顺序进行

排序，除非传递了一个自定义的比较器。

```
memberNames.stream().sorted()
                    .map(String::toUpperCase)
                    .forEach(System.out::println);
Output:

AMAN
AMITABH
LOKESH
RAHUL
SALMAN
SHAHRUKH
SHEKHAR
YANA
```

请记住排序只会创建流的排序视图，而无需操纵后续集合的排序。
成员名称的顺序是不变的。

终端操作" class="reference-link">终端操作

终端操作返回某个类型的结构，而不是再次返回一个流。

A) forEach()

该方法有助于迭代流的所有元素，并对他们中的每一个执行一些操作。
改操作作为 lambda 表达式参数传递

```
memberNames.forEach(System.out::println);
```

B) collect()

collect() 方法用于从一个 stream 中接收元素，并将他们存储在一个集合中，并通过参数分类。

```
List<String> memNamesInUppercase = memberNames.stream().sorted()
                                .map(String::toUpperCase)
                                .collect(Collectors.toList());

System.out.print(memNamesInUppercase);

Outpout: [AMAN, AMITABH, LOKESH, RAHUL, SALMAN, SHAHRUKH, SHEKHAR,
YANA]
```

C) match()

可以使用各种匹配操作检查某个断言条件是否与流匹配。所有这些操作都是终端并返回一个 Boolean 结果。

```
boolean matchedResult = memberNames.stream()
                    .anyMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);

matchedResult = memberNames.stream()
                    .allMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);

matchedResult = memberNames.stream()
                    .noneMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);

Output:

true
false
false
```

D) count()

Count 是一个终端操作，将流中的元素的数量返回一个 long 类型的值

```
long totalMatched = memberNames.stream()
                    .filter((s) -> s.startsWith("A"))
                    .count();

System.out.println(totalMatched);

Output: 2
```

E) reduce()

该终端操作使用给定的函数来对流的元素进行操作。
返回值类型是 Optional。

```
Optional<String> reduced = memberNames.stream()
                    .reduce((s1,s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);

Output: Amitabh#Shekhar#Aman#Rahul#Shahrukh#Salman#Yana#Lokesh
```

短路操作" class="reference-link">短路操作

虽然可以对满足断言条件的集合中的所有元素执行流操作，但每次在迭代期间遇到
匹配元素时，通常都希望中断操作。在外部迭代中，你可以使用if-else块。
在内部迭代中，有一些方法可以用于实现这样的目的。让我们看看两个这样方法的例子：

A) anyMatch()

一旦满足断言条件，该方法将返回 true。它不会处理任何更多的元素。

```
1. boolean matched = memberNames.stream()
2.                     .anyMatch((s) -> s.startsWith("A"));
3.
4. System.out.println(matched);
5.
6. Output: true
```

B) findFirst()

该方法将从流返回第一个元素，然后不再处理任何元素。

```
1. String firstMatchedName = memberNames.stream()
2.                 .filter((s) -> s.startsWith("L"))
3.                 .findFirst().get();
4.
5. System.out.println(firstMatchedName);
6.
7. Output: Lokesh
```

并行" class="reference-link">并行

随着Java SE 7中添加了 Fork / Join 框架，我们可以在应用程序中实施并行操作的高效机制。
但实施这个框架本身就是一个复杂的任务，如果没有做好，这将是复杂的多线程错误的源头，有可能使应用程序崩溃。
随着内部迭代的引入，我们有了另一个并行完成操作的可能。

要启用并行，你只需要创建一个并行流，而不是串行流。
并给你惊喜的是，这真的非常简单。

在任何上面列出的流示例中，任何时候你想要在并行核心中使用多个线程的特点作业，
你只需调用 parallelStream() 方法而不是 stream()方法。

```
1.  public class StreamBuilders {
2.       public static void main(String[] args){
3.          List<Integer> list = new ArrayList<Integer>();
4.           for(int i = 1; i< 10; i++){
5.               list.add(i);
6.           }
7.           //Here creating a parallel stream
8.           Stream<Integer> stream = list.parallelStream();
9.           Integer[] evenNumbersArr = stream.filter(i -> i%2 ==
    0).toArray(Integer[]::new);
10.          System.out.print(evenNumbersArr);
11.      }
12.  }
```

这项工作的关键驱动因素是让开发人员更易于使用并行性。
虽然 Java 平台已经提供了对并发和并行的强大支持，但开发人员在根据需要
将代码从串行迁移到并行时面临着不必要的障碍。
因此，重要的是鼓励形成串行与友好的并行相结合的风格。
这促进了将焦点转移到描述应该执行什么计算，而不是如何实现执行的步骤。
同样重要的是，要使得并行更容易但不会使其失去可视化之间的平衡。
使并行透明化将引入不确定性以及用户可能不期望的数据竞争的可能性。

这就是我想要分享的关于 Java 8中引入的 Stream 概念的基础知识。我将在后面的章节中讨论与 Streams 有关的其它事项。

学习快乐 ！！

Java8 - Stream Distinct

Stream Distinct

Java8 - Stream一行一行读

Stream一行一行读

Java8 - 遍历目录

遍历目录

Java8 - 写入文件

写入文件

Java8 - WatchService

WatchService API

Java8 - 字符串转日期

字符串转日期

Java8 - 连接字符串数组

连接字符串数组

Java8 - Base64编解码

Base64编解码

Java8 - Math的精确运算支持

Math的精确运算支持

Java8 - 比较器使用lambda

比较器使用lambda

Java8 - 内部迭代 VS 外部迭代

内部迭代 vs. 外部迭代

外部迭代

直到 Java 7 为止，集合框架都一直依赖于外部迭代的概念。一个集合通过实现 `Iterable` 接口提供一个遍历自身所有元素的方法，即 `Iterator`，使用者则通过它来顺序地遍历集合中的元素。举例来说，如果我们希望将一组字符串全部变成大写，那么我们可能会这样实现：

```
1. public class IterationExamples {
2.     public static void main(String[] args){
3.         List<String> alphabets = Arrays.asList(new String[]
   {"a","b","b","d"});
4.
5.         for(String letter: alphabets){
6.             System.out.println(letter.toUpperCase());
7.         }
8.     }
9. }
```

或者我们也可以这样写：

```
1. public class IterationExamples {
2.     public static void main(String[] args){
3.         List<String> alphabets = Arrays.asList(new String[]
   {"a","b","b","d"});
4.
```

```
5.          Iterator<String> iterator = alphabets.listIterator();
6.          while(iterator.hasNext()){
7.              System.out.println(iterator.next().toUpperCase());
8.          }
9.      }
10. }
```

以上两种写法采用了外部迭代的方式，这种方式足够直截了当，但也有下列问题：

1. Java 的 for-each 循环/迭代器天生就是顺序的，因此在处理集合元素时必须按照集合制定的顺序。
2. 它减弱了对控制流的优化，这些优化本能够能通过数据的乱序、并行、短路和懒惰等方法去实现。

内部迭代

有时我们需要 for-each 循环（顺序、依次）的特性，但是通常情况下这些特性有碍于性能的提升。通过将外部迭代替换为内部迭代，用户能够让库来控制遍历，从而只需要编写操作数据的代码。

上一个例子的内部迭代的实现为：

```
1. public class IterationExamples {
2.     public static void main(String[] args){
3.         List<String> alphabets = Arrays.asList(new String[]
   {"a","b","b","d"});
4.
5.         alphabets.forEach(l -> l.toUpperCase());
6.     }
7. }
```

> 外部迭代将“干什么”（将每个字符串变成大写）和“怎么做”（使用 *for* 循环/迭代器）混杂在一起，内部迭代则通过库来控制“怎么做”，而将“干什么”留给用户去实现，这样的方式能够提

> 供很多潜在的好处：用户的代码能够变得更加简洁并聚焦于如何处理问题而不是通过什么方式去处理问题，同时也能够将复杂的性能优化代码移动到库的内部从而使所有用户受益。

Java8 - Regex as Predicate

Regex as Predicate

学习如何将正则表达式编译到`java.util.function.Predicate`中。当您想要对匹配的符号执行某些操作时，这可能很有用。

将正则表达式转换为断言

我有不同域名的电子邮件列表，我只想对域名为`example.com`的电子邮件`ID`进行一些操作。

现在使用`Pattern.compile().asPredicate()`方法从编译的正则表达式获取断言。该断言可以与`lambda`表达式一起使用，以将其应用到每个符号上。

```
1.  import java.util.Arrays;
2.  import java.util.List;
3.  import java.util.function.Predicate;
4.  import java.util.regex.Pattern;
5.  import java.util.stream.Collectors;
6.
7.  public class RegexPredicateExample {
8.      public static void main(String[] args) {
9.          // Compile regex as predicate
10.         Predicate<String> emailFilter = Pattern
11.
    .compile("^(.+)@example.com$")
12.                                         .asPredicate();
```

```
13.
14.         // Input list
15.         List<String> emails = Arrays.asList("alex@example.com",
    "bob@yahoo.com",
16.                                 "cat@google.com", "david@example.com");
17.
18.         // Apply predicate filter
19.         List<String> desiredEmails = emails
20.                                         .stream()
21.                                         .filter(emailFilter)
22.                                         .collect(Collectors.
    <String>toList());
23.
24.         // Now perform desired operation
25.         desiredEmails.forEach(System.out::println);
26.     }
27. }
```

输出：

```
1. alex@example.com
2. david@example.com
```

使用`Pattern.matcher()`使用正则表达式

如果要使用旧的`Pattern.matcher()`，那么可以使用下面的代码模板。

```
1. public static void main(String[] args)
2. {
3.
4.     Pattern pattern = Pattern.compile("^(.+)@example.com$");
5.
6.     // Input list
7.     List<String> emails = Arrays.asList("alex@example.com",
    "bob@yahoo.com",
8.                         "cat@google.com", "david@example.com");
```

```
 9.
10.     for(String email : emails)
11.     {
12.         Matcher matcher = pattern.matcher(email);
13.
14.         if(matcher.matches())
15.         {
16.             System.out.println(email);
17.         }
18.     }
19. }
```

输出：

```
1. alex@example.com
2. david@example.com
```

欢迎在评论部分告诉我你的疑问。

学习愉快！

Java8 - 字符串拼接

字符串拼接

从Java7到目前为止，我们可以通过向 `String.split()` 方法中传递参数的方式来分割一个字符串，然后把分割后的字符串列表以数组的形式返回。
但是，如果需要连接字符串或者通过分隔符连接的字符串来创建一个CSV文件，则必须遍历字符串列表或数组，
然后使用 `StringBuilder` 或 `StringBuffer` 对象拼接这些字符串，最后得到 `CSV` 。

使用 `join()` 进行字符串连接（CSV创建）

在 Java8 中使得字符串拼接任务变得容易。现在你可以使用 `String.join()` 方法，
其中第一个参数是分隔符，然后可以传递多个字符串或实现了 `Iterable` 接口的实例作为第二个参数，下面的示例将返回 `CSV` ：

```
1.  package java8features;
2.
3.  import java.time.ZoneId;
4.
5.  public class StringJoinDemo {
6.      public static void main(String[] args){
7.          String joined = String.join("/","usr","local","bin");
8.          System.out.println(joined);
9.
10.         String ids = String.join(", ",
```

```
    ZoneId.getAvailableZoneIds());
11.         System.out.println(ids);
12.     }
13. }
```

输出：

```
1. usr/local/bin
2. Asia/Aden, America/Cuiaba, Etc/GMT+9, Etc/GMT+8.....
```

所以下次当你在使用 java8 并且想拼接字符串时，记得你的工具箱中已经有一个方便的方法，请尽情使用它吧！

学习愉快！

原文出处：https://howtodoinjava.com/java-8/java-8-string-join-csv-example

Java8 - 比较日期差异

比较日期差异