# Java8



# Lambda表达式

简述



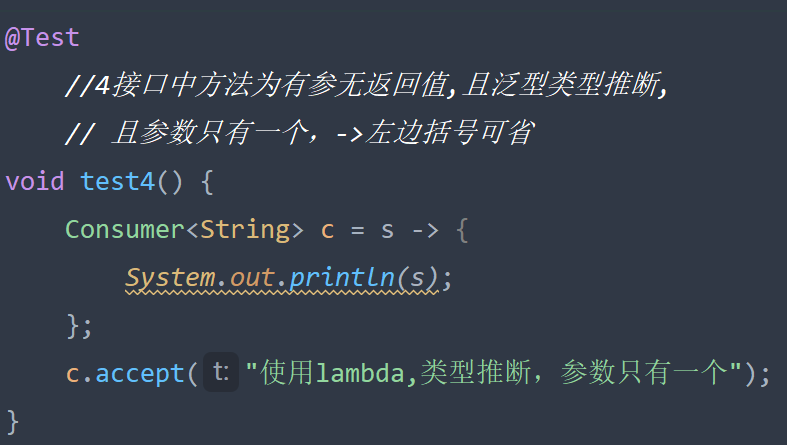
->叫做lambda操作符

Lambda表达式是依赖于接口存在的，在别的语言中，lambda表达式是作为一个函数出现，但是在java中，是作为任意接口的实现类出现。万事万物皆对象。

匿名接口实现类：Runnbale r=new Runnable{重写接口中的方法 }

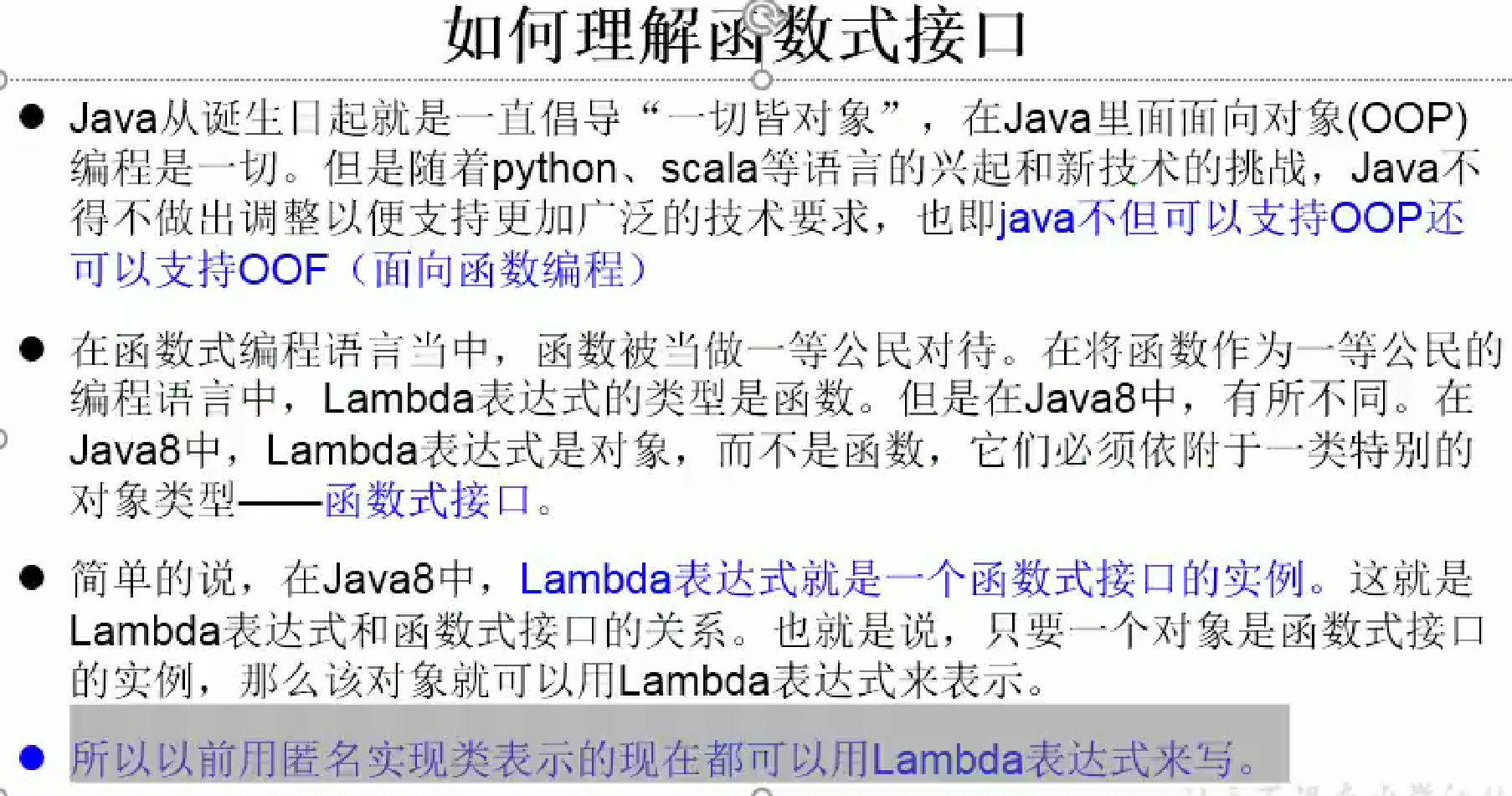
lambda表达式：Runnable r= （重写的方法的形参列表）->{ 重写的方法的方法体 }

Lambda表达式需要依赖函数式接口，只能在接口中只有一个抽象方法时使用。



## 函数式接口Functional





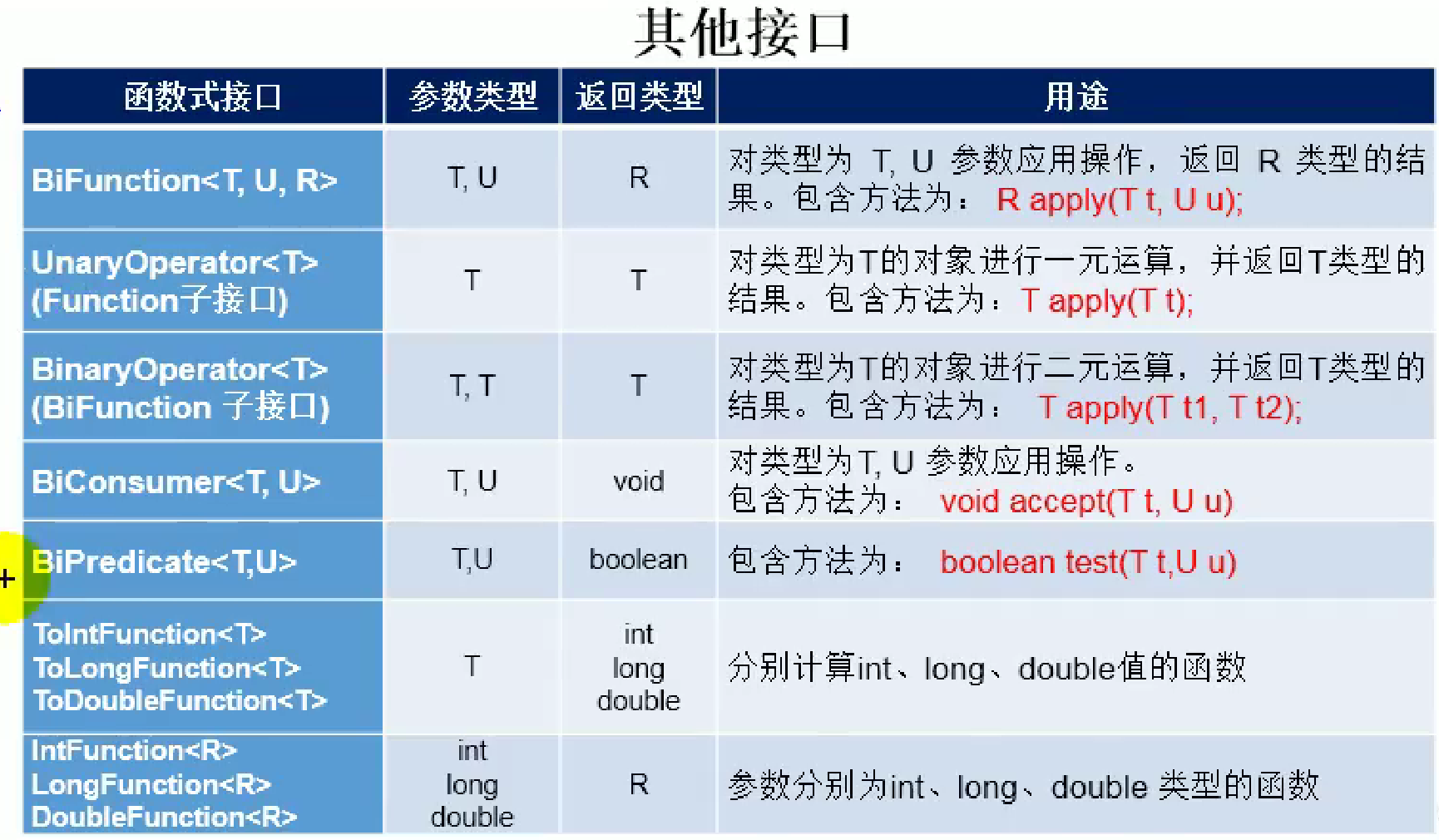
也就是说，在别的以函数为主体的语言中，lambda表达式就是一个函数。但是在java中，我们不可能将一个函数作为具体内容初始化到内存中。所以Lambda表达式实际上作为一个接口对象加载到内存中，但是前提是接口中只能有一个抽象方法。

即仅含有一个抽象方法的接口就叫做函数式接口。可以使用lambda表达式来简化调用该函数。

## function包下的函数式接口



有了这四种API接口，我们以后如果要定义一种接口，就是上述四种的一种，那么我们就直接new上面类的lambda对象就可以了。重写里面的方法。而不用再自己去自定义接口。





# 引用

## 方法引用

概述

如果使用lambda表达式生成接口实例时，恰好方法体中我们只写了一句调用某个对象/类的方法b，而恰好接口的抽象方法a的形参和这个方法b的形参列表是同一个/同好几个形参（即传入接口的形参实际被方法使用），恰好a的形参类型又跟b的形参类型一致且个数相同，比如并且，如果方法a有返回值，则方法b必须有返回值且返回值类型跟方法a的一致。

**为什么一直强调是抽象方法a形参是b的形参类型的父类或者类型一致呢？因为从a传进来的实参终究是被b使用的，所以最起码b的形参类型是a的形参类型的父类。即多态。**

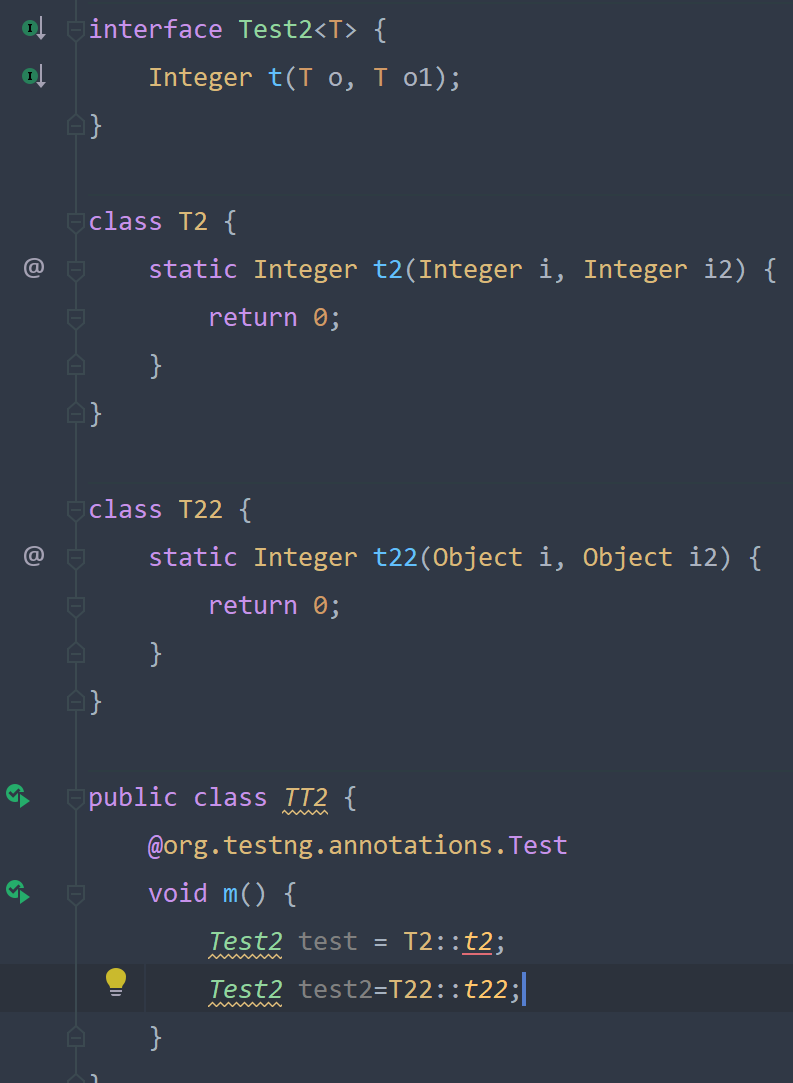
此时，用lambda表达式实例化函数式接口对象时，方法a和b都不需要再去写形参列表，用方法b的形参列表代表方法a的形参列表

使用时必须保证方法a的形参类型起码与方法b的形参的类型一致或者a形参列表类型是b形参列表类型的子类。或者方法a的形参是T，不指明T情况下b方法的形参必须为Object类型的。因为不指明T的话，编译器认为T就是引用的方法中的形参类型。

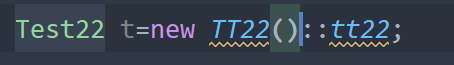
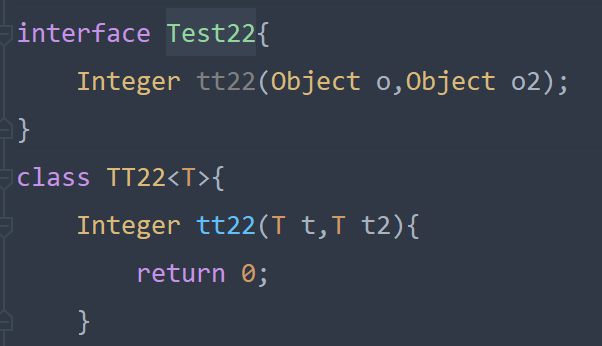
或者干脆方法a的形参类型和个数与方法b的一样。



泛型也是Object的子类，由此说明抽象方法形参类型不得大于b方法中形参类型。

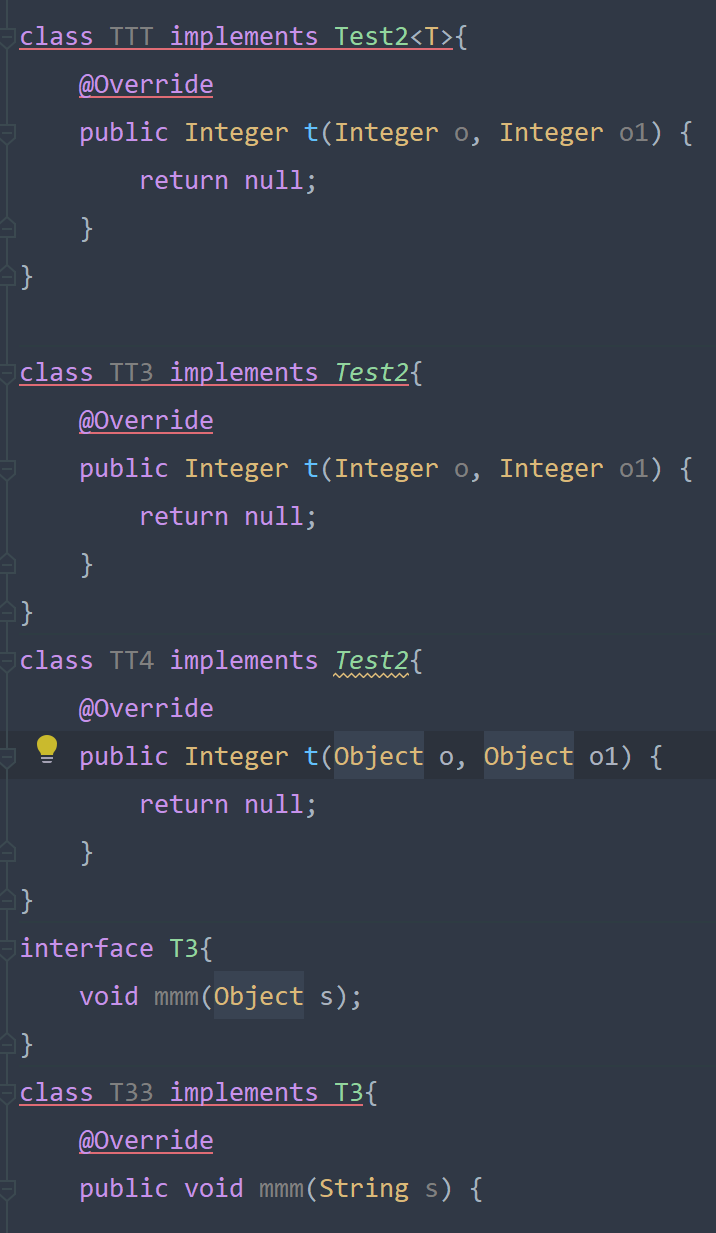


附加知识：实际上泛型也是Object类型的，Object也可以传给泛型，换句话说泛型=Object在大小上，但是泛型不是一个类！如果抽象方法参数为泛型的，引用方法形参为Object的，是可以引用该方法的，因为不管调用接口实例的实现方法传入什么参数，都是Object类型的，都可以作为方法b的实参。

这样也是可以的

实现接口中参数为泛型的方法时，若实现类继承了接口的泛型，则在实现方法中必须将参数声明为泛型的，保持一致。

如果实现时没有继承泛型，则jvm默认将泛型替换为Object。那么实现类中实现该方法时必须将参数类型声明为Object的。



最后方法a的方法体改成引用方法b的形式，如对象：：方法b（b是非静态方法时）或类：：方法b（b可以是静态或非静态）

方法引用由于本质还是在写lambda时使用，所以也算作是实例函数式接口的一部分。

在实例化函数式接口时，我们用（形参列表）->，这时/写上形参列表意义是我们写方法体要用的到这些形参，而为啥方法引用不需要再写形参列表?因为此时方法体已经确认了，就是引用的方法，自然不用再画蛇添足。

另有一特殊情况，如果方法a中的形参有多个，且方法a的第一个形参是方法b的调用者，即x.b，并且方法a中的其他形参是方法b中的所有形参，那么也可以使用方法引用。前提就变成了保证方法a形参列表的第一个形参是方法b的调用者，且方法a中的其他形参与方法b中的形参列表中的形参类型个数相同。

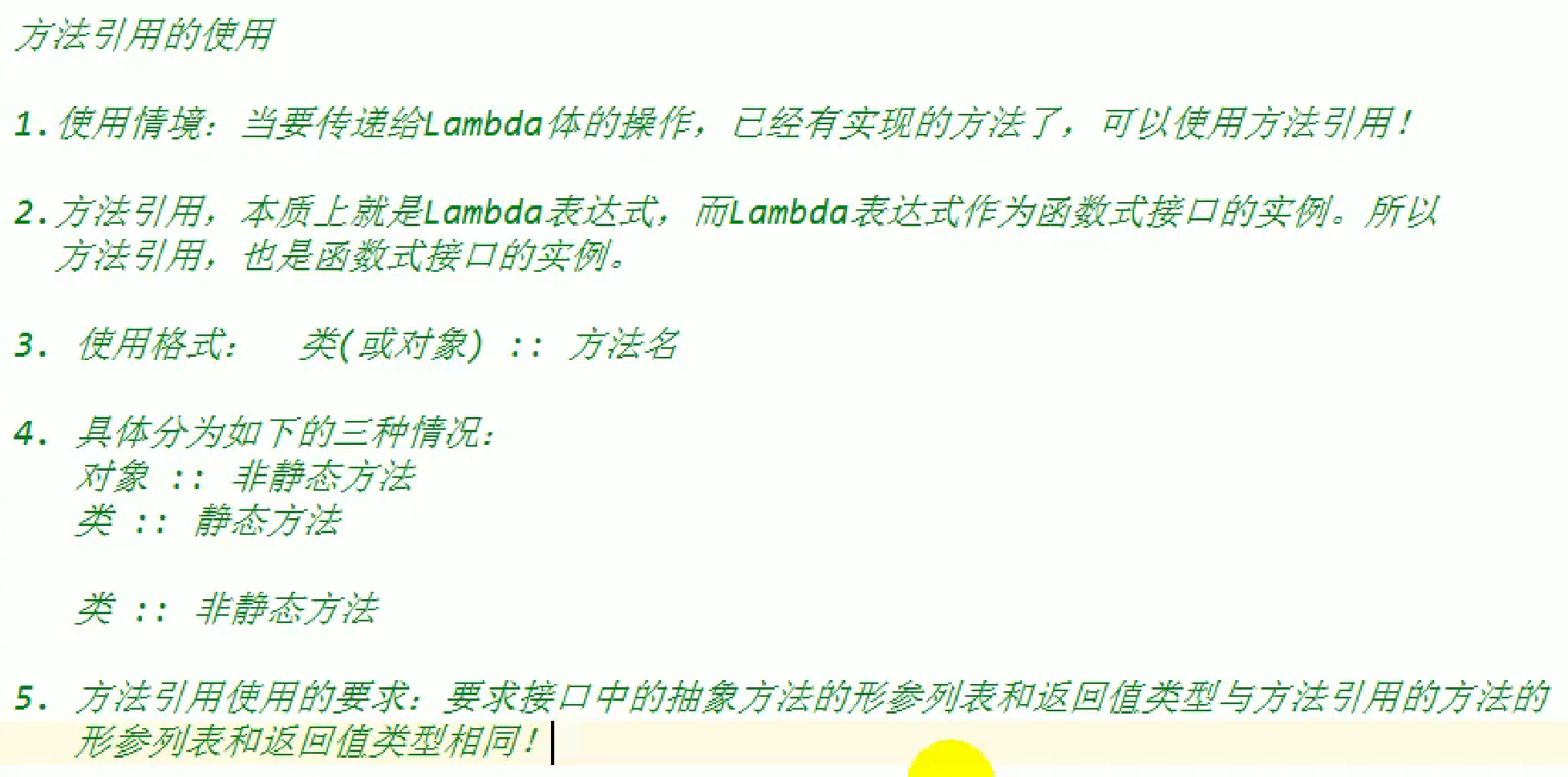
即使用方法引用两种情况：

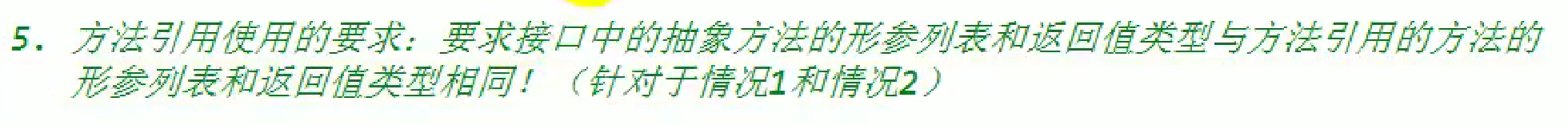
1. 实现函数式接口对象时实现的抽象方法体中只只调用了一个方法，且抽象方法的形参列表与该方法类型个数一样，并且返回值一样。
2. 实现函数式接口对象时实现的抽象方法体中只使用了一个方法，且该方法的调用者就是抽象方法的形参列表的第一个。且抽象方法的从第二个开始的参数的个数类型与该方法的形参列表的个数类型与都相同。且该方法的返回值与抽象方法的返回值类型一样。

这时格式变为方法b的调用者类：：方法b。实际上是方法b的调用者类：：方法a的第一个形参.方法b。

虽然在这种情况下，抽象方法第一个形参作为方法b的调用者，类型我们前提肯定是知道的，但是为了格式，还是要写上XXX：：。

譬如抽象方法a形参只有一个，无返回值，则可以作为方法体中方法b的调用者（b也没有返回值），这时可以直接XXX：：b。XXX为b的类，实际上为抽象方法a中形参的类型，既然形参作为b调用者，那么类型肯定是b的类了。





### 类：：静态方法

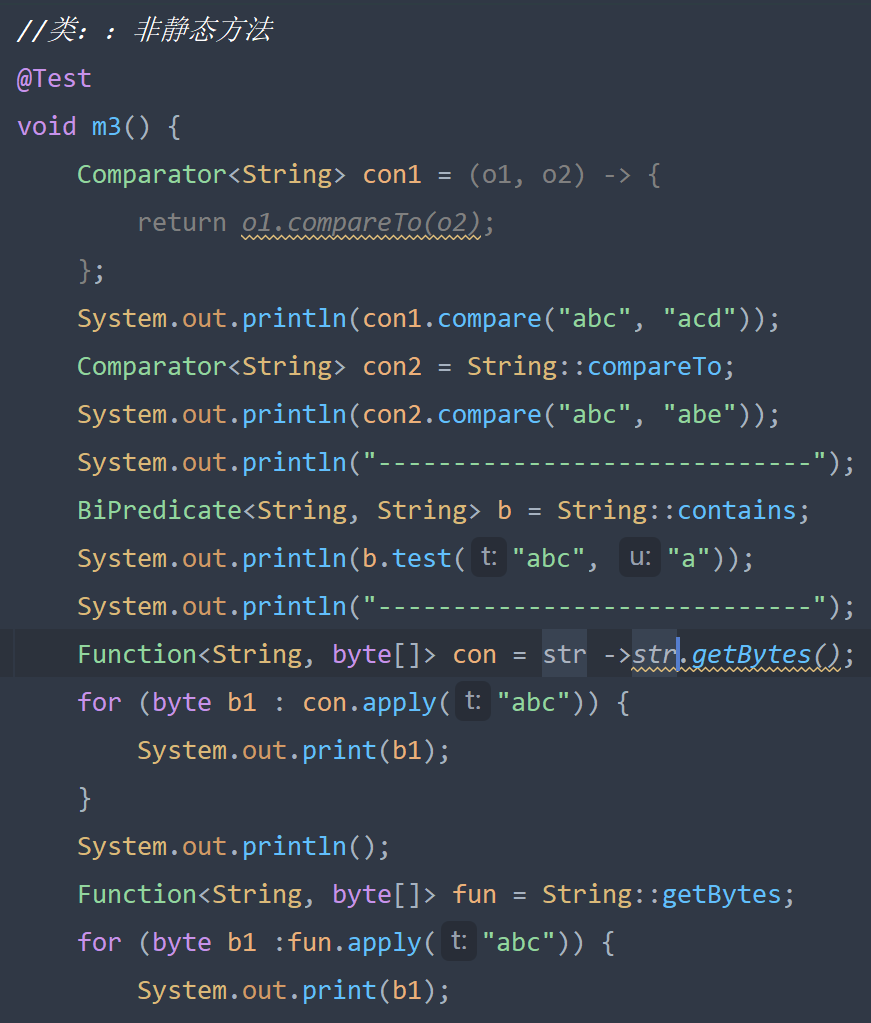
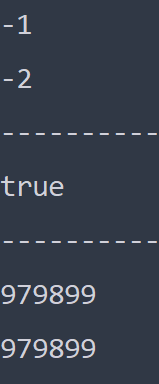


### 对象：：非静态方法

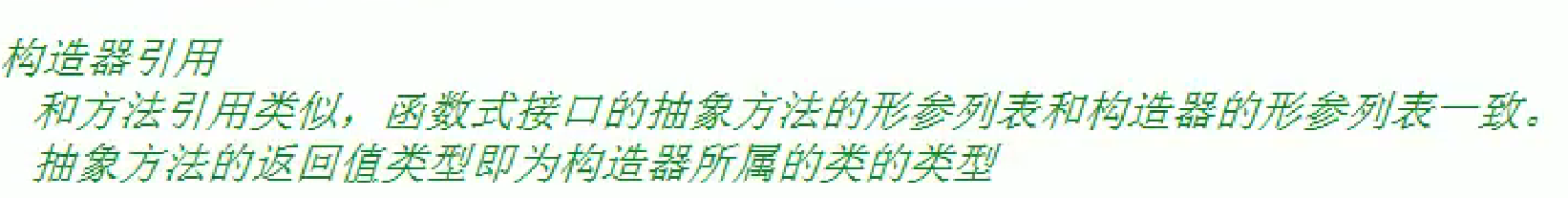


### 类：：非静态方法

这里的类：：只起到声明方法的调用者的类型作用，而不是上面两种情况一样是作为方法的调用者。这里的调用者是抽象方法形参列表的第一个形参。

## 构造器引用



构造器引用类似于方法引用。

类似于前面提到的方法引用的三种情况的类：：方法。

但是格式是：类：：new。实际上是类：：new（xxxxx）

抽象方法a中的形参列表可以作为new构造器的形参列表的，这就使得我们保证方法b中的形参列表的类型个数都必须与抽象方法的形参列表一致。这样才可以省略形参列表。这个方面与方法引用一致。

有一点但是构造器是new关键字调的，不是类或者对象中调的，所以类：：这个类也就只是告诉jvm我们使用的是这个类的构造器。

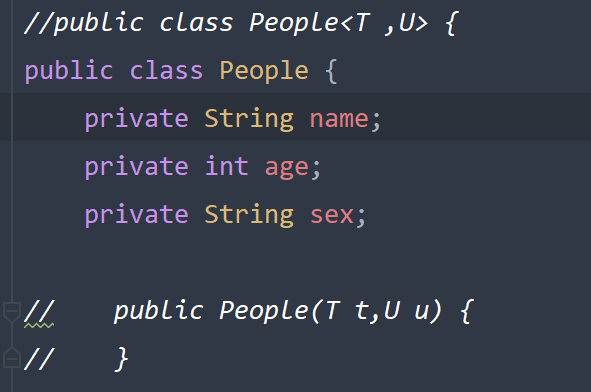
同时这个类也代表了抽象方法的返回值类型。

当然构造器是哪个类的得跟抽象方法的返回值类型的那个类一致才行。换句话说，使用构造器引用，编译器就会自动检测抽象方是否有返回值，并且返回值类型是否与构造器引用的类一样，因为new是个特殊方法，实例化出来并返回一个对象。所以接口的抽象方法也必须声明有返回值且类型为构造器引用返回的对象的类。

当然实际上还是构造器引用配合抽象方法。抽象方法才是老大。

//People类中有三个属性String name；int age；String sex；且有空参构造器，一个参数构造器，两个参数构造器，三个参数构造器。

同时还有这个



为了辩证抽象方法形参类型必须是构造器中形参类型的子类或者同类的。不仅构造器引用，方法引用也是一样。



## 数组引用

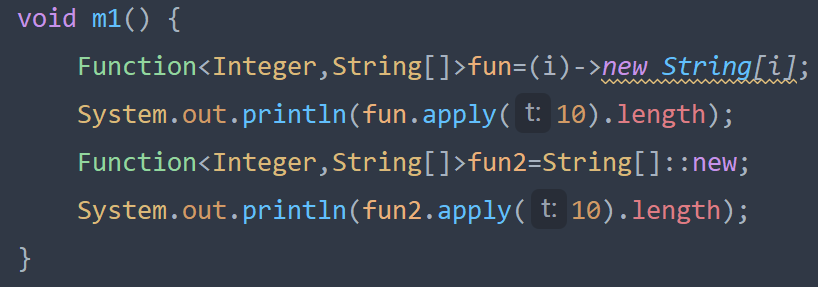
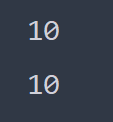


数组引用实际上就是new一个数组返回，跟构造器引用时一个性质。

即抽象方法中的形参要保证个数是一个，且形参类型为Integer的，代表数组的个数（new的形参）。

然后抽象方法的返回值类型就是new出来的数组的类型。

格式：…=String[] ：：new;

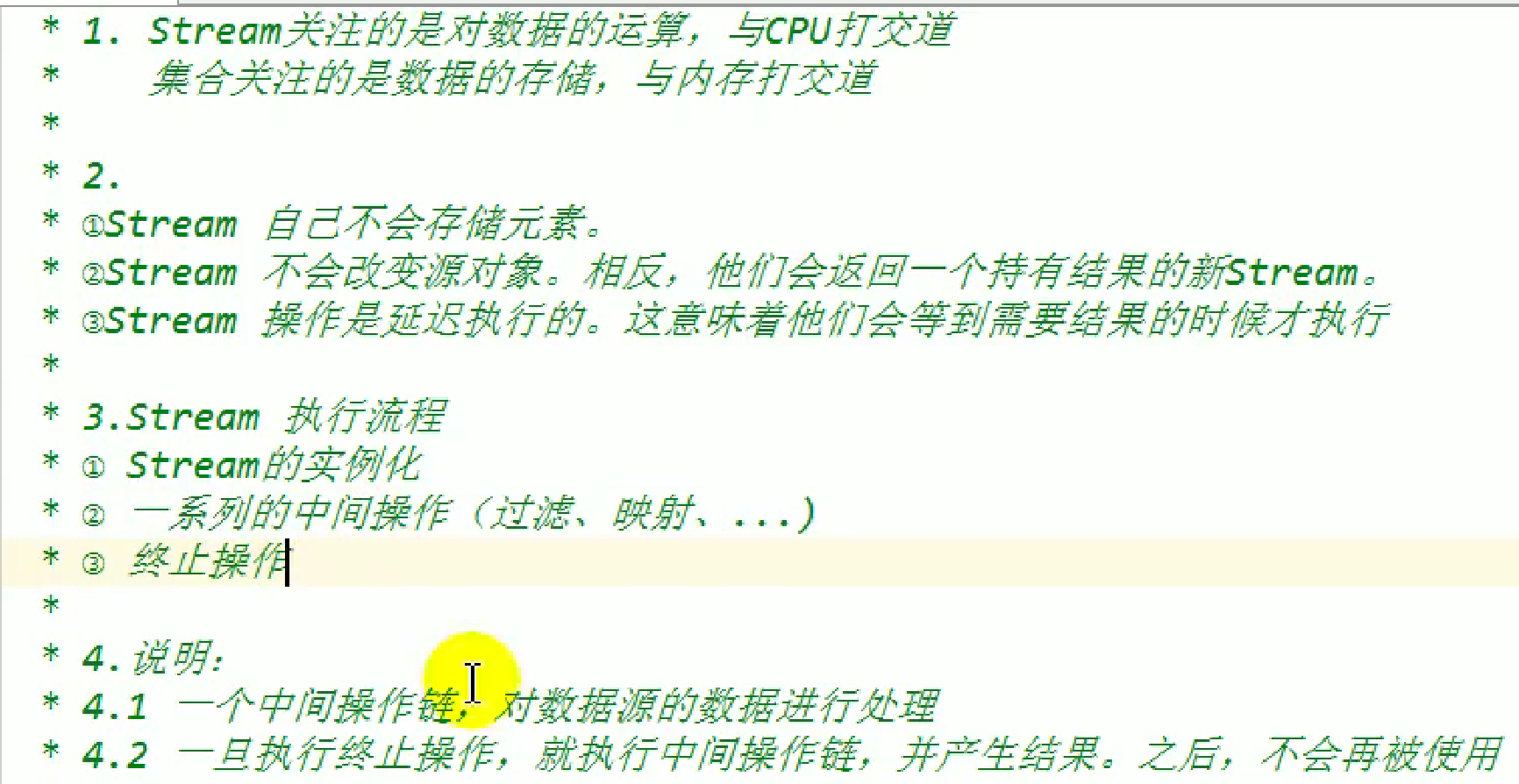
 

# Stream

## 概述

经过一系列流中间操作之后，容器还是最开始的那个容器，只不过容器中存储的数据发生了变化，注意！终结操作永远是对最开始的容器进行操作，或者可以这么理解，对最开始的容器得到的stream这个最开始的流进行forEach等终结操作。

终结操作的永远是最开始的那个容器！！！！假设容器list1中原本只有1，2，3。直接调用forEach传入的是这三个Integer，调用的也是Integer的toString。然而list1.stream经过中间操作map（），1，2，3分别被add到了一个新的list2，list3，list4而这三个list的stream（）被返回作为容器中的新对象取代原来的1，2，3，那么最后forEach时传入的形参是三个stream！输出的就是这三个stream地址。除非forEach嵌套，让这三个stream再调用他们自己的forEach。或者返回这三个的list时用的是flatmap映射。此时虽然映射回的是stream但随后会将生成stream的三个list返回，这样forEach传入的形参就是三个list了。调用的是三个list的toString输出的还是123.



Stream也是一种流，类似于以前的输入输出流，是作用在数据之上建立起来的对数据操作的流。如果流执行到了终止操作关掉了，那么这个流也就作废了。

与IO不同的是IO可以直接操作并覆盖数据，StreamAPI则会在原数据基础上做一些运算，并将运算产生的新数据重新封装返回，不改变原数据。

## 实例化

### 方式一数组工具类生成

既然是对数据计算，那么就需要在数据的容器之上操作，常见的容器之一：数组。

先调用Arrays工具类中的静态方法Stream（xxx），然后将具体的数组对象放入形参列表。

这样才会生成操作在某数组之上的Stream。

### 方式二集合生成

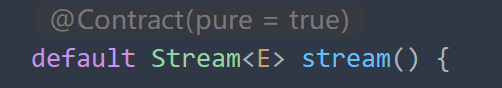
常见的容器之二：集合

即通过集合来生成一个Stream。然后对此集合进行计算。

在Collection接口中有默认方法来生成Stream，但不是静态的，所以仍需Collection接口的具体集合实现类对象来调用。集合对象.Stream()或集合对象.paralleStream()



第一种为生成一个顺序计算的Stream，比如一个集合中有10条数据，那么这个集合用



方法生成的StreamAPI如果对集合进行计算只会从上往下挨个计算。

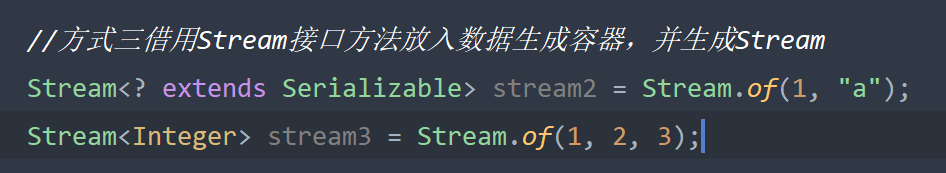
第二种为类似多线程的计算。用



方法生成的Stream对集合进行计算时，会同时对多块数据进行计算。类似多线程。



### 方式三Stream接口默认方法生成

如果我们手头没有装数据的容器对象，Stream接口为我们提供了生成容器的默认方法of（xxxxx）。我们只需要将数据全部放入形参列表即可，且该方法是带泛型的，生成的Stream会自动识别本容器中数据类型，比如都是Integer的，of生成的就是对Integer操作的Stream。

### 方式四直接使用无限流方法对数据操作

1. Stream接口中提供了静态方法iterate（T t，Uxxx<T> u）用来生成迭代类型的Stream流。



形参列表为第一个是种子，即操作对象，我们可以将数据封装在种子中，第二个形参为Funcion函数接口的子接口UnaryOperator，Function接口是传入T返回U，这个子接口是传入T返回T，即传入与返回的类型是一样的。我们可以使用lambda或方法引用实例一个。iterate方法体就是调用我们用lambda/方法引用实例的UnaryOperator匿名对象的apply方法，并将第一个参数种子作为形参传入。

所以实际上就是调用我们的lambda/方法引用对种子操作。我们知道Stream流操作数据需要生成Stream，操作并约束操作数据，终止操作这三部，用无限流方法生成的Stream相当于生成并指定了操作过程但是没有加约束。

然后调用forEach（Customer xx），传入任意一个消费者函数接口的实例。



forEach（），传入消费型函数接口实例，Stream会将数据传入forEach方法，并且调用实例的accept方法并将数据作为形参。

消费型接口实例可以使用lambad/方法引用，forEach即为终结方法。一般我们都会写入System.out::println作为实参。并且参数还是Stream的T泛型的消费者函数接口，可以放入容器的数据。可以用sout语句输出数据作为终结操作。

由于Stream流的延迟调用，检测到有了终结方法之后，无限流开始执行，但是我们没加约束，那么无限流就会一直循环执行方法对种子操作。

那么如何才能终止？

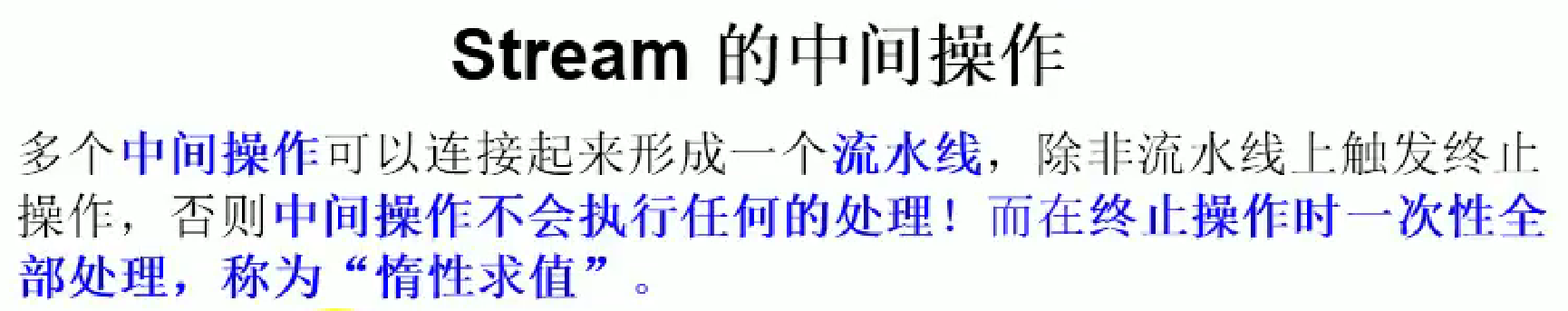
我们可以在返回的Stream流对象之后，forEach终结之前，调用limit（Long xx），传入一个Long类型的形参，也就是循环次数，用来约束Stream流的计算次数。那么这个limit也返回一个Stream流，我们需要终结的就是这个limit返回的流而不是最开始的Stream流。于是iterate返回的Stream流不需要forEach，只需要继续调用limit，调用limit返回的Stream才需要继续调用forEach终结流。

1. 还提供了静态方法generate（Supplier s）用来生产一个生成类型的Stream流。



需要我们传入一个Suppiler函数接口的对象，可以使用lambda/方法引用。同样generate方法体中就是一直调用lambda/方法引用的方法，如果想要终止，同样需要在generate方法返回的Stream对象之后接着调用limit约束这个流之后返回新的Stream流，然后这个新的流调用forEach终结流。

## 中间操作



实例化Stream流之后，需要对数据进行操作，操作又分为筛选与切片，映射，排序。

其中排序是会生成新的Stream流的。

### 切片与筛选

**筛选**

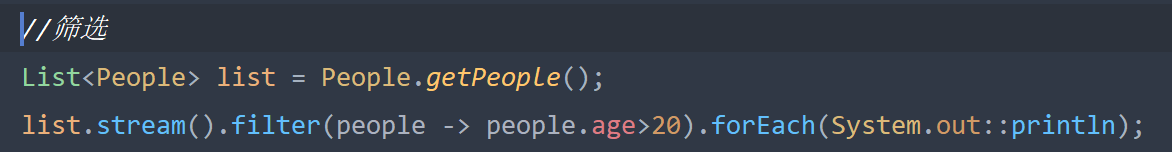
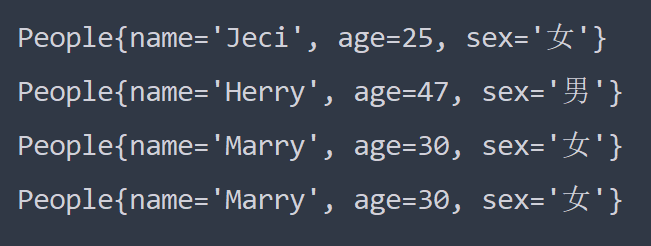
筛选操作即为对数据按照我们手动加入的方法进行操作。如找出工资大于7000的员工。

切片操作为跳过前几条数据，输出前几条数据等。或者去重。

1. 筛选People中年龄大于20的人

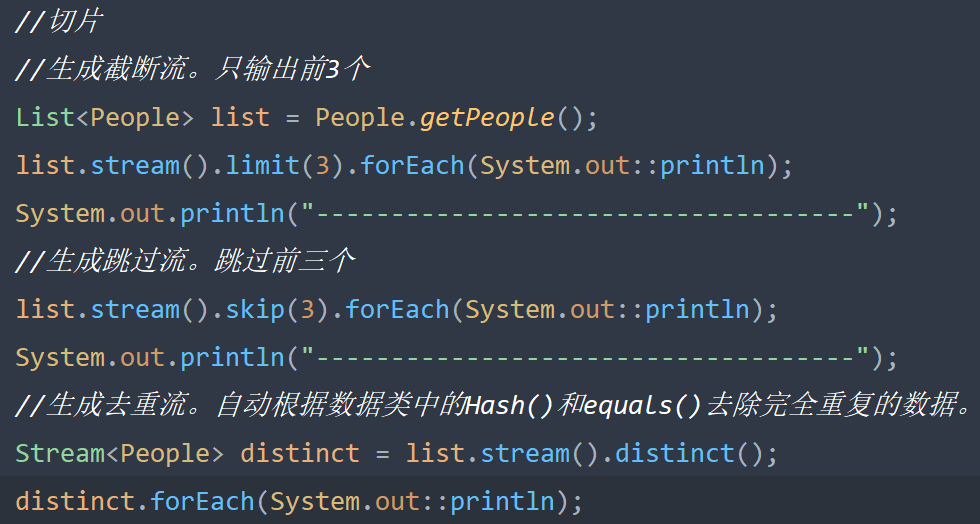
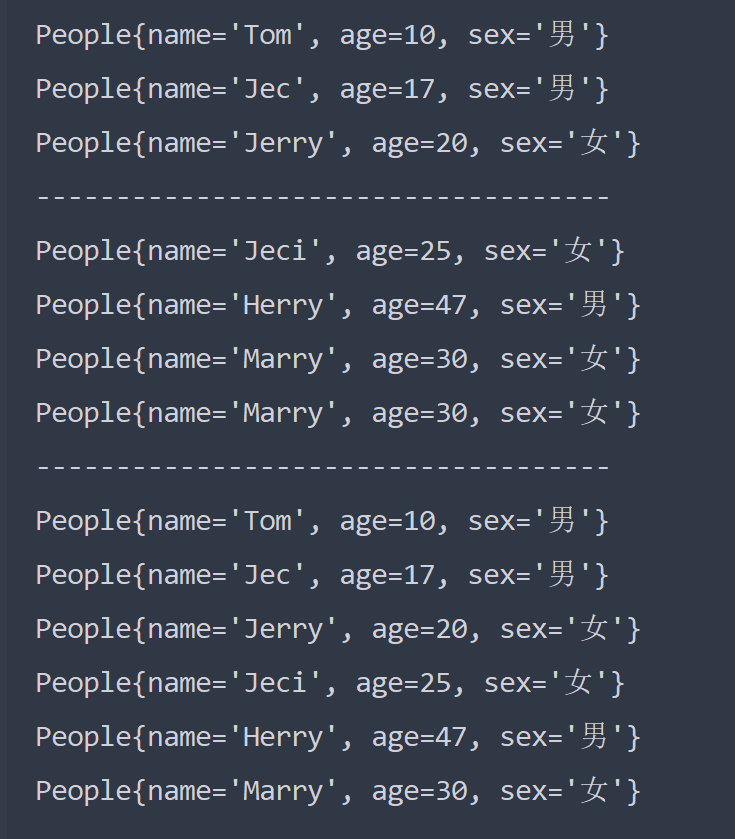
用到方法filter（Predicate<T> xx），参数为断定型函数接口的实例返回值时bool类型，其中泛型T如果容器为ArrayList<People>那么生成的Stream泛型也是People，那么fliter中的断定型函数接口的泛型也是People，所以我们写lambda时，只需要返回形参的年龄>20即可。filter方法会根据bool值筛选数据。

最后别忘了forEach（），传入消费型函数接口实例，Stream会将数据传入forEach方法，并且调用实例的accept方法并将数据作为形参。

**切片**

1. list.Stream().limit（Long n）生成新的截断流，用来限制操作的数据条数前x条。别忘了在截断流之后forEach（……）
2. list.Stream().skip(Long n)生成新的跳过流，用来跳过前x条数据的计算，别忘了在跳过流之后forEach（……）。
3. list.Stream().distinct()生成新的去重流，根据数据类中的hash()和equlas()去除完全重复的两个数据。别忘了在去重流之后forEach(……)

### 映射

终结操作的永远是最开始的那个容器！！！！假设容器list1中原本只有1，2，3。直接调用forEach传入的是这三个Integer，调用的也是Integer的toString。list1.stream经过中间操作map（），1，2，3分别被add到了一个新的list2，list3，list4而这三个list的stream（）被返回作为容器中的新对象取代原来的1，2，3，那么最后forEach时传入的形参是三个stream！输出的就是这三个stream地址。除非forEach嵌套，让这三个stream再调用他们自己的forEach。或者返回这三个的list时用的是flatmap映射。此时虽然映射回的是stream但随后会将生成stream的三个list返回，这样forEach传入的形参就是三个list了。调用的是三个list的toString输出的还是123.

映射即为用容器得到Stream流之后，对数据做map/flatMap操作，map/flatMap形参列表为Function（T t，R r），即用map流将容器中每个数据转换成另一种数据，然后map流一次性forEach。

如将实际想要操作的每个数据的一部分取出来，这部分就构成一个新的数据组。

然后直接对map流forEach输出这部分数据组。

或者在map流上再用filter过滤掉这部分数据组中不符合条件的数据。

map和flatMap区别：

将自定义Function对象传入map之后，map会在Stream流之上根据对象方法将容器中每个数据按照方法映射出新的数据，所有映射出来的数据作为一整个数据组，跟原来容器的模式一样。在map之上再forEach就是对数据组的终止操作。

如果map返回的数据组中每个数据都是一个Stream，以前我们都是对每个对象数据进行forEach如sout。现在一个个对象数据成了一个个Stream。，现在终止操作时就相当于大的map流之内是对每个Stream进行sout。显然有时候不符合我们想要的结果。

那就需要在map执行forEach对每个Stream sout时，再嵌套一层对每个Stream的forEach。这样就达到我们想要的效果了。

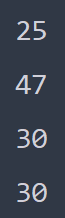
所以flatMap的作用就是检测map流得到的数据组，如果数据组中有任意的Stream类型的数据a，即数据a本身也是一个Stream流，那么map执行forEach时，到了这个Stream类型的数据，就会自动调用a的forEach，将a操作的数据组遍历sout。如果a本身这个Stream操作的数据组仍然是一个Stream组，那么就继续调用每个Stream的forEach。

如我们使用map流，传入Function实例为一个方法，该方法会将Stream数据组每个数据得到对应的Stream流返回，这样flatMap流拿到的就是一个每个数据都是Stream流的数据组了，flatMap做整体的forEach的时候就会对每个Stream流数据再forEach，相当于嵌套forEach。（双重for循环）

所以flatMap的操作类似于树，或者文件目录，第一层目录a下面发现有子目录2个，就去子目录1，然后将子目录1的元素都sout之后，去子目录2，完成后返回第一层的b目录，继续检测……

所以如果我们担心Stream流得到的数据组执行自定义的map操作后映射出来的数据有一些成了Stream类型的数据，直接上flatMap方法，将Function实例（返回什么样的Stream流）传入即可。



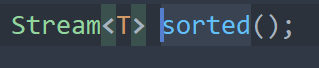
### 排序

排序只有两种自然排序和定制排序。

**自然**

在容器返回的Stream流之上再使用sorted流对数据组进行自然排序，如果数据组不是基本数据类型而是自定义的类对象，那么类如果继承了comparable接口重写了方法也是可以自然排序的。

list.Stream().sorted().forEach(s.out::p)



如果数据是自定义的sorted流会调用对象的重写方法将另一对象传入比较。

将数据组自然排序之后返回sorted流，将这个流forEach输出的就是新的排序后的数据组。

**定制**

如果数据类型不是基本数据类型，而且没有继承接口Comparable，那么就必须往sorted形参列表中传入Comparator实例，因为是函数式接口所以可以使用lambda。



此时sorted流会调用实例的Comparator中的compare方法将数据组中每两个数据放入比较。

最后在sorted流之上forEach（s.out::p），输出的就是sorted流对原数据组排序之后又生成的新数据组。

## 终止操作

终结操作的永远是最开始的那个容器！！！！假设容器list1中原本只有1，2，3。直接调用forEach传入的是这三个Integer，调用的也是Integer的toString。list1.stream经过中间操作map（），1，2，3分别被add到了一个新的list2，list3，list4而这三个list的stream（）被返回作为容器中的新对象取代原来的1，2，3，那么最后forEach时传入的形参是三个stream！输出的就是这三个stream地址。除非forEach嵌套，让这三个stream再调用他们自己的forEach。或者返回这三个的list时用的是flatmap映射。此时虽然映射回的是stream但随后会将生成stream的三个list返回，这样forEach传入的形参就是三个list了。调用的是三个list的toString输出的还是123.

中间操作的方法返回值都是另一个Stream。所以才不会终止，如果要终止，那么方法返回值就不是Stream，而是其他的类型。

### 匹配与查找

**allMatch**



allMatch（），形参为断定型函数接口，T为容器中数据类型，allMatch方法会根据lambda的条件将当前流中数据挨个比较，全都符合才返回true，有一个不符合返回false。

**anyMatch**

****

形参还是断定型函数接口，接口形参为容器中的数据类型，方法根据lambda中条件比较当前流中只要任意数据满足条件返回true。

**noneMatch**

****

形参为断定型函数接口，如果任意数据满足lambda条件，则返回false。

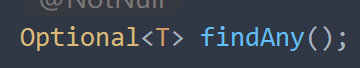
**findFirst**



从当前流中找到第一个数据并返回，但是会将数据封装到Optional的属性value中返回一个Optional的对象。

但是Optional类重写了toString，实际输出的还是数据，只不过格式标注了是Optional类型的。

**findAny**

****

从当前流中随机找一个数据并返回，但是会将数据封装到Optional的属性value中返回一个Optional的对象。

一般使用parallelStream并行流调用此方法，普通的Stream流老是返回第一个数据。

**count**

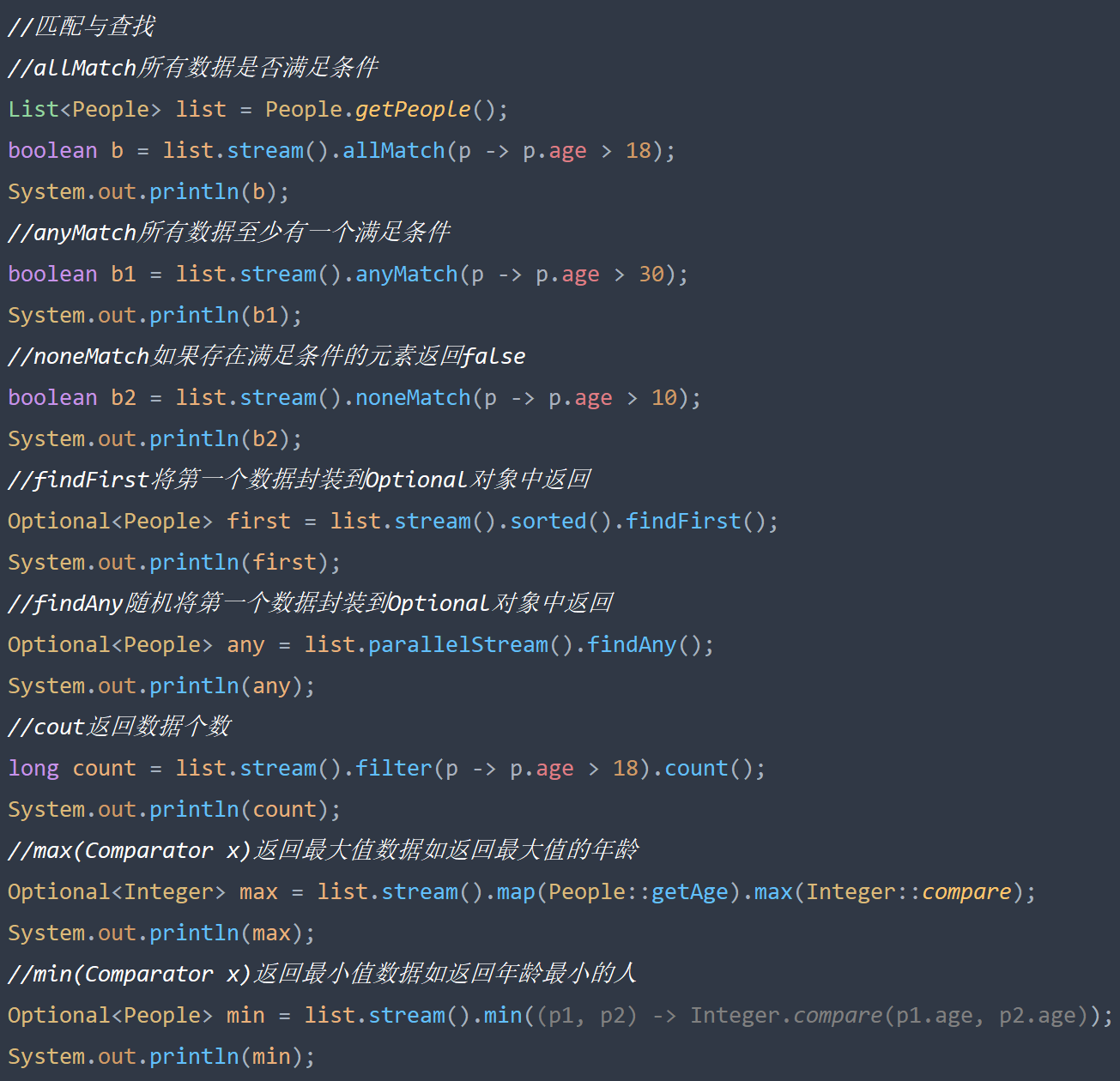


返回当前流中数据的总个数

**max**



根据Comparator的lambda作比较，将最大值取出并返回一个Optional对象。



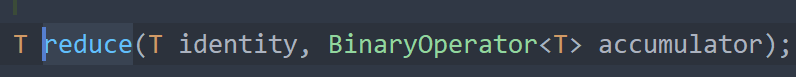
**forEach**

一般就用作输出

### 归约

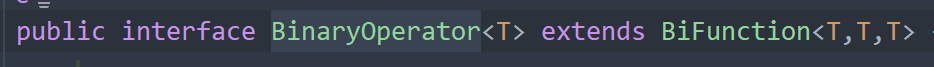
**reduce**

归约，类似于递归，就是用reduce方法将容器中的两个数据做某种计算，然后结果再与第三个数据计算，一直到最后一个数据。





BinaryOperator是BiFunction的子接口



底层核心调用的就是apply方法。但是BinaryOperator的apply方法为两个Stream泛型的形参，返回值也是Stream泛型的。

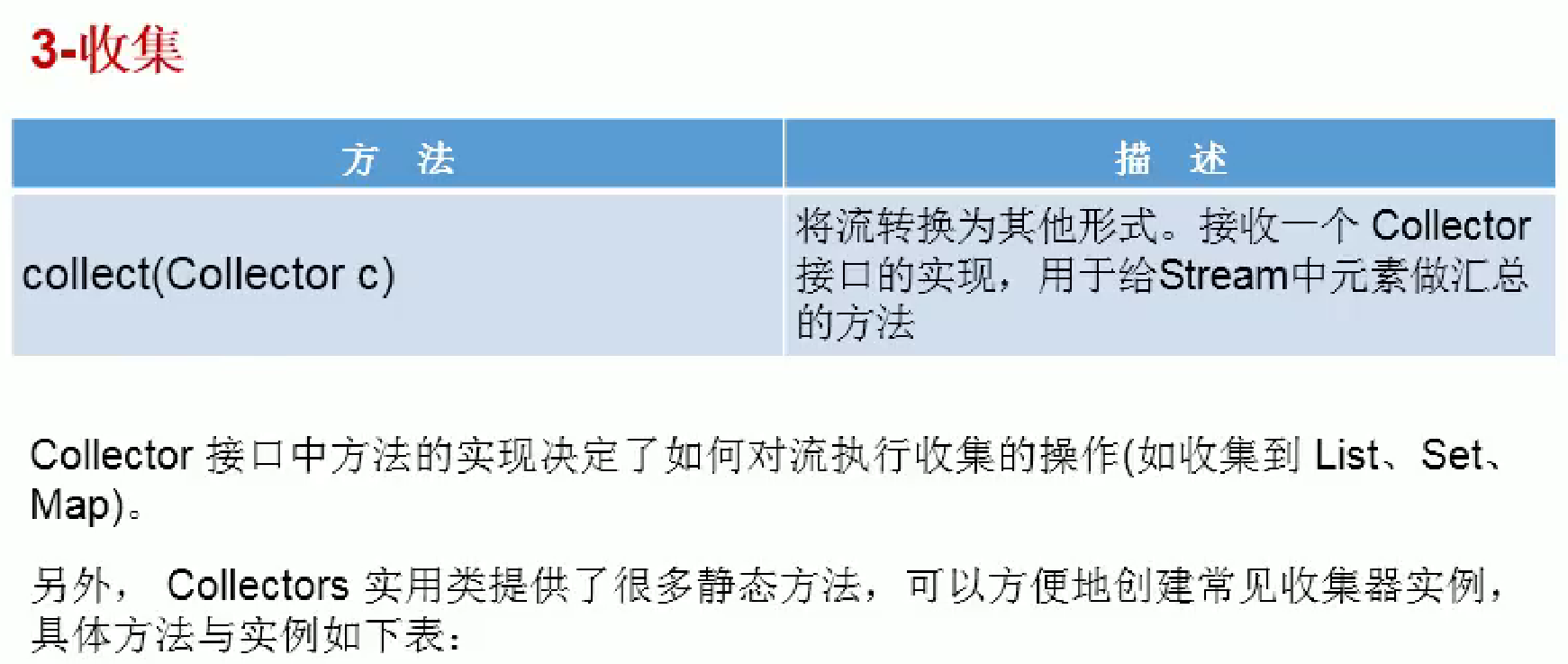
Stream泛型的identity形参就是一个初始值，类型跟容器中数据类型一样。

实现原理：reduce方法调用BinaryOperator函数接口的lambda，将两个容器中的数据传入做计算，完成后自动将结果作为下次使用lambda的第一个形参，第二个形参就再取一个新的数据从容器中。循环至最后一个数据。



### 收集

我们使用中间操作得到的都是流，如果我们此时就想要中间操作之后的新数据，那就可以用收集终止操作并将新数据装到List/Set中。



**collect（Collector c）**

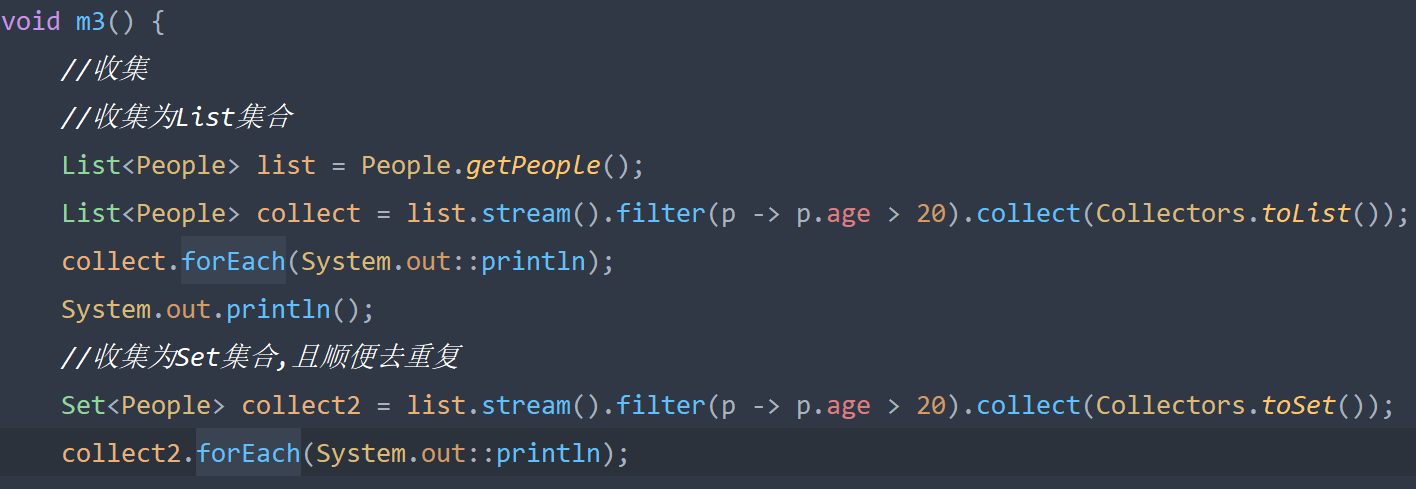
collect需要我们传入Collector接口的实例，这个实例可以用Collectors工具类调用toList/toSet返回一个将List.Set集合封装起来的Collector对象。

这个Collector对象经过collect方法体之后就会产生装流操作的新数据的容器List/Set。

过程就不详细展开了，大概就是在collect方法中将当前流操作的数据组拿出来放到Collector对象中重新封装，将Collector中的List/Set属性赋值给new的List/Set，并返回。

并且返回的集合泛型肯定也是当前Stream泛型的。

需要注意的是，如果收集为Set，那么就是顺便去重复了相当于

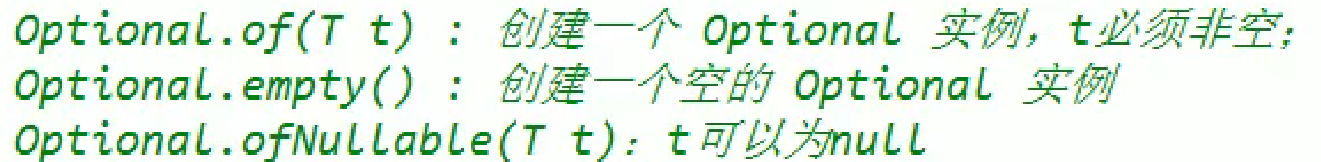


## 补充

其实集合中也有forEach方法，但是性质不与Stream的一样，Stream的是作为终止操作，集合的就是传一个消费型Function，然后将集合中每个数据放入消费型函数的accept方法中去执行。如果我们传入sout，就是遍历集合。

虽然说执行过程差不多，但是本质作用不一样。

# Optional类



Option类是最初是用来解决空指针问题的。

Optional类类似于Integer类，只封装了一个T类型的对象。也算做是一个容器，内部又属性value用来封装数据。

常用的组合形式为Optional.of(T t)封装对象,此时对象一定不能为null否则get出来就是报异常没有值。+Optional对象.get（）得到其封装的value值。

或者Optional.ofNullable（T t）,这个t可以为null +orElse（new备用对象）。

意思就是你可以封装一个空指针的数据，但是调用的时候必须使用orElse方法，然后传入一个new的备用对象。

这样就算真的封装的为空指针的对象，检测到后就会用我们new的备用对象替代这个空指针对象，从而避免出现空指针异常的问题。

