# 简介

## 为什么需要再次修改Java

多核CPU的兴起成为了不容回避的事实。涉及锁的编程算法不但容易出错，而且耗费时间。人们开发了java.util.concurrent包和很多第三方类库，试图将并发抽象化，帮助程序员写出在多核CPU上运行良好的程序。很可惜，到目前为止，我们的成果还远远不够。

开发类库的程序员使用Java时，发现抽象级别还不够。处理大型数据集合就是个很好的例子，面对大型数据集合，Java还欠缺高效的并行操作。开发者能够使用Java 8编写复杂的集合处理算法，只需要简单修改一个方法，就能让代码在多核CPU上高效运行。为了编写这类处理批量数据的并行类库，需要在语言层面上修改现有的Java：增加Lambda表达式。

面向对象编程是对数据进行抽象，而函数式编程是对行为进行抽象。现实世界中，数据和行为并存，程序也是如此，因为这两种编程方式我们都得学。这种新的抽象方式还有其他好处。不是所有人都在编写性能优先的代码，对于这些人来说，函数式编程带来的好处尤为明显，程序员能编写出更容易阅读的代码——这种代码更多地表达了业务逻辑的意图，而不是它的实现机制。易读的代码也易于维护、更可靠、更不容易出错。

在写回调函数和事件处理程序时工程序员不必再纠缠于匿名内部类的冗繁和可读性，函数式编程让事件处理系统变得更加简单。能将函数方便地传递也让编写惰性代码变得容易，惰性代码在真正需要时才初始化变量的值。

Java 8还让集合类可以拥有一些额外的方法default方法。程序员在维护自己的类库时，可以使用这些方法。

## 什么是函数式编程

每个人对函数式编程的理解不尽相同。但其核心是：在思考问题时，使用不可变值和函数，函数对一个值进行处理，映射成另一个值。不理解什么是函数式编程这根本不 重要，我们关心的是如何写出好代码，而不是符合函数式编程风格的代码。

## 1.3 示例

本书中的示例全部都围绕一个常见的问题领域构造：音乐。具体来说，这些示例代表了在专辑上常常看到的信息，有关术语定义如下。

* Artist：创作音乐的个人或团队
  + name：艺术家的名字 (例如“甲壳虫乐队”)
  + members乐队成员 (例如“约翰·列侬”)，该字段可为空
  + origin乐队来自哪里(例如“利物浦”)
* Track：专辑中的一支曲目
  + name：曲目名称
* Album：专辑，由若干曲目组成
  + name：专辑名
  + tracks：专辑上所有曲目的列表
  + musicians：参与创作本专辑的艺术家列表

本书将使用这个问题讲解如何在正常的业务领域或者Java应用中使用函数式编程技术。

# Lambda表达式

Java8的最大变化是引入了Lambda表达式：这是一种紧凑的、传递行为的方式。

## 2.1 第一个Lambda表达式

button.setOnClickListener(new OnClickListener() {

public void onClick(View v) {

System.out.println("button clicked");

}

});

设计匿名内部类的目的是为了方便Java程序员将代码作为数据传递，不过，匿名内部类不够简洁，为了调用一行重要的逻辑代码，不得不加上4行冗繁的样板代码。而且这些代码相当难读懂，因为它没有清楚地表达程序员的意图。我们不想传入对象，只想传入行为，在Java8中，上述代码可以写成一个Lambda表达式，如下：

button.setOnClickListener(v -> System.out.println("button clicked"));

和传入一个实现接口的对象不同，我们传入了一段代码块——一个没有名字的函数。“v”是参数名，“->”将参数和Lambda表达式的主体分开，而主体是用户点击按钮时会运行的一些代码。

和使用匿名内部类的另一处不同在于声明v参数的方式。使用匿名内部类时需要显式地声明参数类型View v，而在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(setOnClickListener方法的签名)在后台推断出了参数v的类型。这意味着如果参数类型不言而明，则无需显式指定。

注：尽管与之前相比，Lambda表达式中的参数需要的样板代码很少，但是Java 8仍然是一种静态类型语言。为了增加可读性并迁就我们的习惯，声明参数时也可以包括类型信息，而且有时编译器不一定能根据上下文推断出参数的类型！

Lambda表达式还可以这样用：

OnClickListener ocl = v -> System.***out***.println(**"button clicked"**);  
button.setOnClickListener(ocl);

所以可以这样理解：Lambda表达式是用于创建一个接口对象的一种简便方式。

button.setOnClickListener(v -> System.***out***.println(**"button clicked"**));

把一个Lambda表达式传给了setOnClickListener()方法，该方法接收的是一个OnClickListener对象，而我们传进去的Lambda表达式看起来不像是一个对象，而像是一个匿名方法，我们把一个方法像数据一样进行了传递。

所以，也可以这样理解，Lambda表达式就是一个匿名方法,将行为像数据一样进行传递。

## 2.2 如何辨别Lambda表达式

Lambda表达式除了基本的形式之外，还有几种变体，如下：

1、Runnable noArguments = () -> System.out.prlintln("Hello World");

2、OnClickListener oneArgument = v -> System.out.println("button clicked");

3、Runnable multiStatement = () -> {

System.out.print("Hello");

System.out.print(" World");

};

4、BinaryOperator<Long> add = (x, y) -> x + y;

5、BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;

一、1中所示的Lambda表达式不包含参数，使用空括号()表示没有参数。该Lambda表达式实现了Runnable接口，该接口只有一个run方法，没有参数，返回类型为void。

二、2中所示的Lambda表达式包含一个参数，可以省略参数的括号

三、3中所示的Lambda表达式的主体是一段代码块，使用{ }括起来，只有一行的代码可以省略花括号

四、4中所示的Lambda表达式包含2个参数，BinaryOperator的函数可以是有返回值的，则x + y的结果是返回值（如果只有一行代码，则不需要写return，否则必须写上return）。

五、5中所示的Lambda表达式显式地声明了参数的类型

Java中初始化数组时，数组的类型就是根据上下文推断出来的，如下：

String[] array = { “hello”, “world” }; // 等号右边并没有声明类型，系统根据上下文推断出类型信息

## 2.3 引用值，而不是变量

在内部类中引用外部的一个局部变量，这个局部变量必须是final的，意味着这个变量不能为其重复赋值，那就不应该算变量了，因为不会变了，就只是一个值而已了，所以内部类引用的是外部的一个值。

String name = **"button clicked"**;  
button.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {  
 System.***out***.println(name);  
 }  
});

这段代码在Java8中不会报错，虽然name没有加上final，但是它已经相当于是final，编译器在编译的时候会自动加上final，这样的final变量称之为“既成实事上的final变量”，所以在Lambda表达式中使用的局部变量也必须是final类型的，如下将会报错：

String name = **"button clicked"**;  
name = **"haha"**;  
button.setOnClickListener(v -> System.***out***.println(name));

这里name进行了两次赋值，编译器没办法把它变成final，这种行为也解释了为什么Lambda表达式被称为闭包。未赋值的变量与周边环境隔离起来，进而被绑定到一个特定的值。

下面则是正确的：

String name = **"button clicked"**;// 其实这里加上final修饰也是可以的，但是既然不用加也可以就不写了，省事。写不写看个人爱好吧！  
button.setOnClickListener(v -> System.***out***.println(name));

## 2.4 函数接口

函数接口是指只拥有一个抽象方法的接口，用作Lambda表达式的类型。如：Runnable noArguments = () -> System.out.prlintln("Hello World");这里的Lambda表达式为“() -> System.out.prlintln("Hello World");”，它的类型为：Runnable。

Java开发工具包（JDK）中提供的一组核心函数接口会频繁出现 ，如下罗列了一些最重要的函数接口：‘

|  |  |  |
| --- | --- | --- |
| 接口 | 参数 | 返回类型 |
| Predicate<T> | T | boolean |
| Consumer<T> | T | void |
| Function<T,R> | T | R |
| Supplier<T> | None | T |
| UnaryOperator<T> | T | T |
| BinaryOperator<T> | (T, T) | T |

## 2.5 类型推断

什么时候需要显式地指明类型？应该出于让代码最便于阅读的目的去考虑，有时省略类型可以减少干扰更容易弄清状况；而有时却需要类型信息帮助理解代码。

Lambda表达式中的类型推断实际上是Java7就引入的目标类型推断的扩展。Map<String, String> map = new Map<String, String>(); 这行代码到了Java 7就可以使用类型推断：Map<String, String> map = new Map<>();编译器根据变量的声明就可以推断出new的Map的泛型类型，这没什么难度。

如果将构造函数直接传给一个方法，也可以根据方法签名来推断类型，如下：

public void useHashMap(Map<String, String> map);

useHashMap(new HashMap<>());

Java7中程序员可以省略构造函数的泛型类型，Java8更进一步，程序员可以省略Lambda表达式中的所有参数类型。需要注意的是，Java8中对类型推断系统的改善值得一提，上面的例子将new HashMap<>()传给useHashMap方法，即使编译器拥有足够的信息，也无法在java7中通过编译。

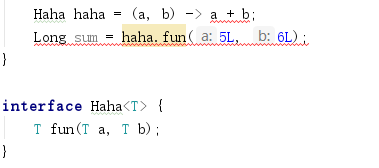
**interface** Haha<T> {  
 **void** fun(T a, T b);  
}

Haha<Long> haha = (a, b) -> System.***out***.println(**"a = "** + a + **", b = "** + b);  
haha.fun(5L, 6L);

如上例子，输出结果：a = 5, b = 6，通过类型推断可知函数a和b的类型是Long，如果我们在声明的时候不写泛型，如下：

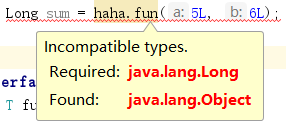
Haha haha = (a, b) -> System.***out***.println(**"a = "** + a + **", b = "** + b);

此时编译运行也是一样的结果，那这时a和b的类型是什么呢？答：Object。再修改例子，如下:



可以看到有两行代码报错了，错误描述分别如下：





因为在声明Haha的时候没有写明泛型，所以参数a和b是被系统推断为Object类型的，函数fun的返回类型也是Object类型。

第一条报错语句原因是两个Object对象是不能使用 + 这种操作符进行相加的。

第二条报错语句原因是fun返回的是Object，不能把一个Object赋值给Long类型的变量。

假设有一个接口A，一个接口B，这两个接口中各有一个方法，方法名不同，但是返回值和参数列表相同，示例如下：

i static interface A {

boolean aa(int param);

}

static interface B {

boolean bb(int param);

}

static class C {

void func(A a) { System.out.println(a.aa(1)); }

void func(B b) { System.out.println(b.bb(3)); }

}

public static void main(String[] args) throws Exception {

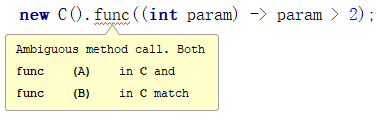
new C().func(param -> param > 2);

}

new C().func(i -> i != 100); 这行代码将报错，有两个，如下：



这个是属于工具问题，Lambada表达式是没问题的，我们给参数加上括号和类型即可解决，然后就剩下一个错误，如下：



大概意思为 ：模棱两可的方法调用，C里的 func(A)和C里的func(B)都匹配。

因为无法推断Lambda表达式是A类型还是B类型。所以在这种路种情况下应该显示的声明Lambda表达式类型，如下：

B b = param -> param > 2;

new C().func(b);

## 练习

ThreadLocal，Java8为该类新加了一个工厂方法，接受一个Lambda表达式并返回一个新的ThreadLocal对象，使用如下：

public class Looper {

public final static ThreadLocal<Looper> sThreadLocal = ThreadLocal.withInitial(() -> new Looper());

}

# 流

## 3.1 从外部迭代到内部迭代

如下代码计算从伦敦来的艺术家有多少个：

int count = 0;

for (Artist art : allArtists) {

if (art.isFrom("London")) count++;

}

尽管这样的操作可行，但存在几个问题，每次迭代集合时，都需要写很多样板代码，将for循环改造成并行方式也很麻烦，需要修改每个for循环才能实现。

此外，上述代码无法流畅传达程序员的意图。for循环的样板代码模糊了代码的本意。程序员必须阅读整个循环体才能理解。若是单一的for循环，倒也问题不大，但面对一个满是循环(尤其是嵌套循环)的庞大代码库时，负担就重了。

就其背后的原理来看，for循环其实是一个封装了迭代的语法糖，我们在这里多花点时间，看看它的工作原理。首先调用iterator方法，产生一个新的Iterator对象，进而控制整个迭代过程，这就是外部迭代。迭代过程通过显式调用Iterator对象的hasNext和next方法完成迭代。展开后的代码如例3-2所示，图3-1展示了迭代过程中的方法调。

例3-2 使用迭代器计算来自伦敦的艺术家人数

int count = 0;

Iterator<Artist> it = allArtists.iterator();

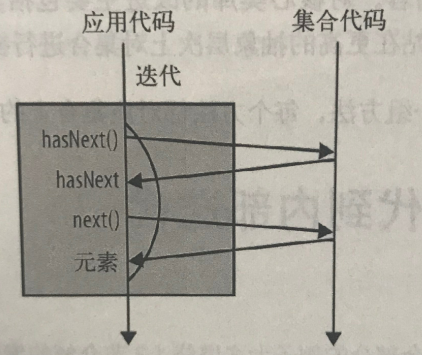
while (it.hasNext()) {

Artist art = iterator.next();

if (art.isFrom("London")) count++;

}

图3-1：外部迭代



然而，外部迭代也有问题。首先，它很难抽象出本章稍后提及的不同操作; 此外，它从本质上来讲是一种串行化操作。总体来看，使用for循环会将行为和方法混为一谈。

另一种方法就是内部迭代，如例3-3所示。首先要注意stream()方法的调用，它和例3-2中调用iterator()的作用一样，但是该方法不是返回一个控制迭代的Iterator对象，而是返

回内部迭代中的相应接口：Stream。

例3-3 使用内部迭代计算来自伦敦的艺术家人数

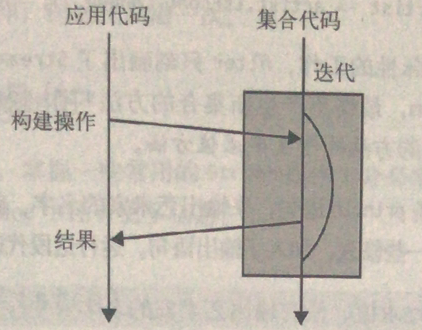
long count = allArtists.stream()

.filter(art -> art.isFrom("London"))

.count();

图3-2 展示了使用类库后的方法调用流程，与图3-1形成对比。

图3-2：内部迭代



通过两张图片的对比，一个是在“应用代码”的地方进行的迭代，一个是在“集合代码”的内部进行的迭代。

Stream是用函数式编程方式在集合类上进行复杂操作的工作。

例3-3可被分解为两步更简单的操作

·找出所有来自伦敦的艺术家，

·计算他们的人数。

每种操作都对应stream接口的一个方法。为了找出来自伦敦的艺术家，需要对stream对象进行过滤filter。过滤在这里是指“只保留通过某项测试的对象”。测试由一个函数完成，根据艺术家是否来自伦敦，该函数返回true或者false。由于Stream API的函数式编程风格，我们并没有改变集合的内容，而是描述出stream里的内容。count()方法计算给定Stream里包含多少个对象。

## 3.2实现机制

例3-3中，整个过程被分解为两种更简单的操作：过滤和计数，看似有化简为繁之嫌——

例3-1中只含一个for循环，两种操作是否意味着需要两次循环?事实上，类库设计精妙，只需对艺术家列表迭代一次。

通常，在Java中调用一个方法，计算机会随即执行操作比如，System.out.println("Hello world")会在终端上输出一条信息。stream里的一些方法却略有不同，它们虽是普通的Java方法，但返回的stream对象却不是一个新集合，而是创建新集合的配方。现在尝试思考一下例3-4中代码的作用，一时毫无头绪也没关系，稍后会详细解释。

例 3-4 只过滤，不计数

allArtists.stream()

.filter(art -> art.isFrom("London"));

这行代码并未做什么实际性的工作，filter只刻画出了Stream，但没有产生新的集合。像filter这样只描述Strean，最终不产生新集合的方法叫作惰性求值方法，而像count这样最终会从Strean产生值的方法叫作及早求值方法。

如果在过滤器中加入一条println语句，来输出艺术家的名字，就能轻而易举地看出其中的不同，如下：

allArtists.stream()

.filter(art -> {

System.out.println(art.getName());

return art.isFrom("London")

});

如上代码，运行后将看不到任何的输出，如果再调用一个拥有终止操作的流，如计数操作count()，就会看到艺术家的名字会被输出。可以这么理解：filter方法只是往Stream对象上保存了一个过虑的条件（Lambda表达式），但是并没有使用，当调用count函数时和进行集合的迭代，而且在迭代时使用之前保存的过虑条件。

判断一个操作是惰性求值还是及早求值很简单，只需看它的返回值。如果返回值是Stream，那么是惰性求值，如果返回值是另一个值或为空，那么就是及早求值。使用这些操作的理想方式就是形成一个惰性求值的链，最后用一个及早求值的操作返回想要的结果，这正是它的合理之处。

整个过程和建造者模式有共通之处。建造者模式使用一系列操作设置属性和配置，最后调用一个build方法，这时对象才被真正创建。

读者一定会问：“为什么要区分惰性求值和及早求值?”只有在对需要什么样的结果和操作有了更多了解之后，才能更有效率地进行计算。例如，如果要找出大于10的第一个数字，那么并不需要和所有元素去做比较，只要找出第一个匹配的元素就够了（不然所有元素都遍历一下就有点浪费了），这也意味着可以在集合类上级联多种操作，但迭代只需一次（也就是说在一次迭代中完成你需要做的多件事）。

## 3.3 常用的流操作

### 3.3.1 collect(toList())

collect(Collectors.toList())方法由Stream里的值生成一个列表，是一个及早求值操作。

Stream的of方法使用一组初始值生成新的Stream对象，其内部原理是使用Arrays.stream(T[] array)方法将数组变成Stream对象的：

List<String> list1 = Stream.*of*(**"a"**, **"b"**, **"c"**).collect(Collectors.*toList*());  
List<String> list2 = Arrays.*asList*(**"a"**, **"b"**, **"c"**);  
System.***out***.println(list1.equals(list2)); *// 结果为true*

### 3.3.2 map

如果有一个函数可以将一种类型的值转换成另外一种类型，map操作就可以使用该函数，将一个流中的值转换成一个新的流。

例3-8 使用for循环将字符串转换为大写

List<String> uppercaseList = **new** ArrayList<>();  
List<String> lowercaseList = Arrays.*asList*(**"a"**, **"b"**, **"hello"**);  
**for** (String s: lowercaseList) {  
 String uppercaseString = s.toUpperCase();  
 uppercaseList.add(uppercaseString);  
}

例3-9 使用map操作将字符串转换为大写

List<String> uppercaseList = Stream.*of*(**"a"**, **"b"**, **"hello"**)  
 .map(s -> s.toUpperCase())  
 .collect(Collectors.*toList*());

map()方法是返回了一个新的Stream对象，这个Stream对象中包含了大写的"A"、"B"、"HELLO"。再来理解开头的那段话，String的toUpperCase()函数可以将一种小写字母转换成大写字母，则map操作就可以使用toUpperCase()函数将一个流中的值转换成一个新的流。

注：map函数接收一个Function参数，这个接口里面有一个apply方法，声明为：R apply(T t); ，Lambda表达式的格式还是看得懂的，不明白的是系统是怎么推断出T和R都是String的。模拟的简化代码如下：

**public interface** MyFunction<T, R> {  
 R apply(T t);  
}

**public static class** MyStream<T> {  
 **public static** <T> MyStream<T> of(T... args) {  
 **return new** MyStream<>();  
 }  
  
 <R> MyStream<R> map(MyFunction<? **super** T, ? **extends** R> mapper) {  
 **return new** MyStream<R>();  
 }  
}

MyStream.*of*(**"haha"**)  
 .map(s -> s.toUpperCase());

MyStream.of("haha")，这里方法传了String，所以静态方法中的泛型T被实例化为String，MyStream类上的泛型也被实例化为String，如下：

**public static class** MyStream<T> {  
 **public static** <String> MyStream<String> of(String... args) {  
 **return new** MyStream<>();  
 }  
  
 <R> MyStream<R> map(MyFunction<? **super** T, ? **extends** R> mapper) {  
 **return new** MyStream<R>();  
 }  
}

of方法中的泛型T并不是类上面的泛型T，通过上面代码可知，of方法的返回值类型为：MyStream<String>，所以return new MyStream<>()中虽然没有显示地声明类的泛型，但是系统肯定知道是要创建MyStream<String>类型的对象，所以这里MyStream类上的泛型T也被实例化了，如下：

**public static class** MyStream<String> {  
 <R> MyStream<R> map(MyFunction<? **super** String, ? **extends** R> mapper) {  
 **return new** MyStream<R>();  
 }  
}

Fuction接口如下：

**public interface** MyFunction<? **super** String, ? **extends** R> {  
 ? **extends** R apply(? **super** String t);  
}

MyStream.of("haha")已经看完，接下来看.map(s -> s.toUpperCase()); 怎么理解这句代码呢？这是一句Lambda表达式，这个表达式代表了MyFunction中的apply方法，所以s代表了参数。不明白的是编译是怎么把参数s推断成String类型的？

MyStream.of("String").map(s -> s.toUpperCase()); 使用Lambda表达式，这个代码是OK的，使用我们以前的方式，如下：

MyStream.*of*(**"String"**).map(**new** Function<String, Object>() {  
 @Override  
 **public** Object apply(String s) {  
 **return** s.toUpperCase();  
 }  
});

上面的方法实现是通过开发工具自动生成的，根据泛型Function中的第一个泛型可以是String或者String的父类，所以可以把String替换成CharSequence，如下：

MyStream.*of*(**"String"**).map(**new** MyFunction<CharSequence, Object>() {  
 @Override  
 **public** Object apply(CharSequence s) {  
 **return** s.toUpperCase(); // 但是CharSequence并没有这个toUpperCase()方法，所以可以肯定的是编译器把这个泛型实例化成了String  
 }  
});

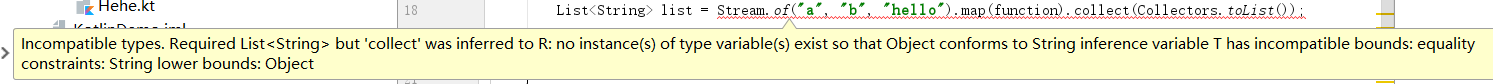
但是如果让开发工具自己去生成一个Function，他怎么能知道你第一个泛型是用String还是用String的哪个父类呢？可以 肯定的是编译器通过“? super String”这个泛型自动选择了String这个类型来当成第一个泛型的类型，所以map(s -> s.toUpperCase())中的Lambda表达式其实就是创建一个Function对象，由编译器帮我们创建，则编译器也只能从“? super String”这里选择String做为泛型的实参。我们把Function参数用到真实的代码，如下：

Function<String, Object> function = **new** Function<String, Object>() {  
 @Override  
 **public** Object apply(String s) {  
 **return** s.toUpperCase();  
 }  
};  
  
Stream.*of*(**"a"**, **"b"**, **"hello"**).map(function);

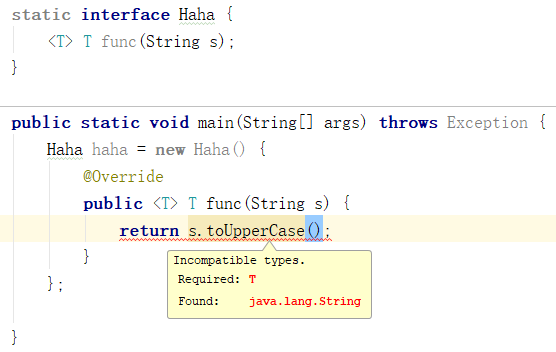
把最后一行代加上转变为集合，如下：

List<Object> list = Stream.*of*(**"a"**, **"b"**, **"hello"**).map(function).collect(Collectors.*toList*());

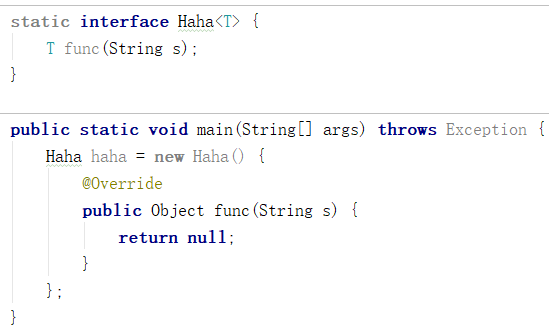
编译通过，修改List中的泛型为String，如下：



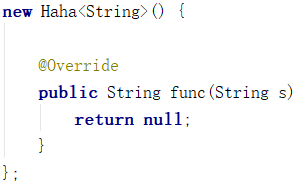
报错了，说明我们传的Function函数不符合要求，把Function函数中的第二个函数参数也改成String，这时编译就OK了，运行也正常了。这说明对于map(s -> s.toUpperCase())，编译器给我们生成的Function对象中的两个泛型都是String，对于第一个泛型“? super String”可以是String或String的父类，编译器选择了推断成String，那第二个泛型参数编译器是怎么推断成String的呢？因为apply方法有返回值，而s.toUpperCase()就是方法的返回值，所以编译器把s.toUpperCase()的类型String当成了返回类型。我们做实验如下：



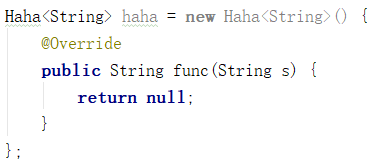
报错信息告诉我们“不兼容的类型，需要T，找到的是java.lang.String”，其实我们看Function类它的泛型是定义在类上面的，所以修改如下：



这就没报错了，这是因为泛型声明在类上面，我们在new Haha的时候没有指定泛型，则系统默认为Object，所以开发工具自动把返回类似创建为Object类型。因为类上有泛型，一般使用这个类的话要实例化类上的泛型，如下：



或者我们在声明Haha的时候指定泛型，则系统也会推断出返回类型是什么，如下：



接着试验：

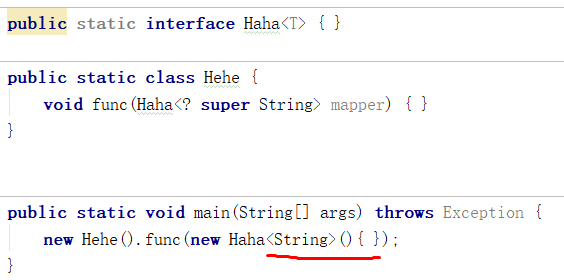
Haha<String> haha = s -> **"haha"**.toUpperCase();

这行代码编译通过，我们在声明Haha的时候使用了String类型，这充分说明了系统通过Lambda表达式给我们创建Haha对象时就是把泛型实例化为String了，不然这行代码没法编译通过的。我们知道Lambda表达式其实就是一种创建接口对象的方便方式，如果接口上有泛型，则在new的时候肯定就需要实例化泛型，所以在Lambda“s -> **"haha"**.toUpperCase();”这句表达式中，编译器通过我们的返回语句haha.toUpperCase()的类型，在创建 Haha时就把泛型实例化为这个类型。

#### 类型推断总结：

1. 在Lambda表达式中，编译器会把”? super String”这样类型的泛型直接实例化为String
2. 在Lambda表达式中，如果返回类型是泛型，则编译器会把泛型实例化为返回值一样的类型

关于第1点也可以实验，如下：



如上图，在new Haha的时候，后面红线的代码是由编译工具自动生成的，说明编译工具对于泛型“? Super String”直接取String做为泛型类型。

### 3.3.3 filter

找出集合中以数字打头的元素：

List<String> list = Arrays.*asList*(**"aaa"**, **"acb"**, **"1ff"**, **"e"**, **"2ccc"**);  
List<String> numberStartList = **new** ArrayList<>();  
**for** (String s : list) {  
 **if** (s.charAt(0) >= **'0'** && s.charAt(0) <= **'9'**) {  
 numberStartList.add(s);  
 }  
}  
numberStartList.forEach(e -> System.***out***.println(e));

使用Stream中的filter解决：

List<String> numberStartList = Stream.*of*(**"aaa"**, **"acb"**, **"1ff"**, **"e"**, **"2ccc"**)  
 .filter(s -> s.charAt(0) >= **'0'** && s.charAt(0) <= **'9'**)  
 .collect(Collectors.*toList*());

### 3.3.4 flatMap

List<Integer> list1 = Arrays.*asList*(1, 2, 3);  
List<Integer> list2 = Arrays.*asList*(4, 5, 6);  
List<List<Integer>> list3 = Stream.*of*(list1, list2).collect(Collectors.*toList*());  
list3.forEach(e -> System.***out***.println(e));

打印如果如下：

[1, 2, 3]

[4, 5, 6]

如果想把两个List合并，可使用flatMap方法，如下：

List<Integer> list1 = Arrays.*asList*(1, 2, 3);  
List<Integer> list2 = Arrays.*asList*(4, 5, 6);  
List<Integer> list3 = Stream.*of*(list1, list2).flatMap(list -> list.stream()).collect(Collectors.*toList*());  
System.***out***.println(list3);

打印结果如下：

[1, 2, 3, 4, 5, 6]

Stream.of(list1, list2) ：Stream中有两个列表

flatMap(list -> list.stream()) ：把Stream中的两个列表转换为两个Stream，然后将这两个Stream链接成一个Stream

### 3.3.5 max和min

Stream上的常用操作之一是求最大值和最小值。

求名字最短的人：

List<Person> list = Arrays.*asList*(**new** Person(**"黄蓉"**), **new** Person(**"西门吹雪"**), **new** Person(**"欧阳峰"**));

Person p = list.stream()

.min(Comparator.*comparing*(person -> person.**name**.length()))

.get();

System.***out***.println(p.**name**);

Java 8 给Comparator提供了一个新的静态方法comparing，用于方便地实现一个比较器，只需要告诉编译器拿什么作为比较即可。这个例子中是拿名字的长度作为比较。

Stream的min方法返回的是optional对象，Optional对象有点陌生，它代表一个可能存在也可能不存在的值。如果stream为空，那么该值不存在，如果不为空，则该值存在。先不必细究，4.10节将详细讲述optional对象，现在唯一需要记住的是，通过调用get方法可以取出optional对象中的值。max方法的使用与min一样。

花点时间研究一下comparing方法是值得的。实际上这个方法接受一个函数并返回另一个函数。我知道，这听起来像句废话，但是却很有用。这个方法本该早已加入Java标准库，但由于匿名内部类可读性差且书写冗长，一直未能实现。现在有了Lambda表达式，代码变得简洁易懂。

min方法的源码如下：

**public interface** Stream<T> {

**public final** Optional<T> min(Comparator<? **super T**> comparator) {  
 **return** reduce(BinaryOperator.*minBy*(comparator));  
}

}

Stream在创建的时候已经把T实例化为Person，所以：

**public interface** Stream<Person> {

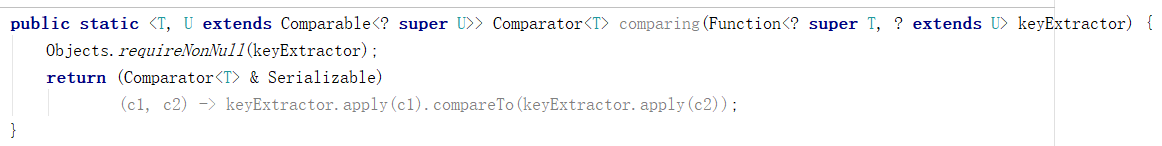
**public final** Optional<Person> min(Comparator<? **super** Person> comparator) {  
 **return** reduce(BinaryOperator.*minBy*(comparator));  
}

}

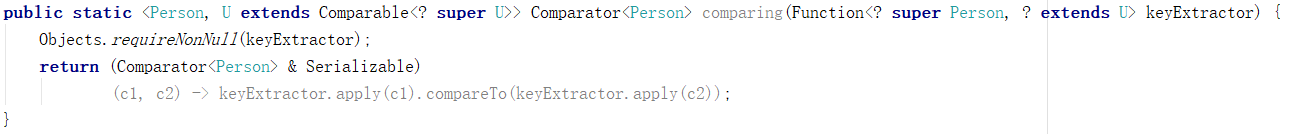
所以min方法的传参相当于：

Comparator<? super Person> comparator = Comparator.comparing(person -> person.name.length());

再来看comparing方法的源码如下：



这个方法返回的Comarator是带泛型的，而我们左边声明的Comparator表明了接收的活泛型为：？super Person，根据以前的经验，编译器会推断为Person，所以可以这样看源吗：



不太理解这里的Function的第一个泛型为什么要限定为可以是Person的父类。

再看comparing方法中参数keyExtractor的传叁：

Function<? super Person, U extends Comparable<? super U>> keyExtractor = person -> person.name.length();

可以这样理解：Function的第一个泛型被实例化为Person，第二个泛型被实例化为Comparable，这样再理解Function源码，如下：

**public interface** Function<Person, Comparable> {  
 Comparable apply(Person t);  
}

根据我们的Lambda表达式（person -> person.name.length()）,编译器为我们创建的Function参数为 ：

Function<? **super** Person, ? **extends** Comparable> keyExtractor = **new** Function<Person, Comparable>() {  
 @Override  
 **public** Comparable apply(Person person) {  
 **return** person.**name**.length();  
 }  
};

所以Function的apply方法返回的对象是Integer，而Integer是实现了 Comparable接口的。



这是一个Lambda表达式，用来创建Comparator<Person>对象的，这个Lambda表达式就是实现了Comparator中的compare方法，所以c1，c2会被推断为Person类型，方法返回一个int值，而Integer的compareTo方法也是返回int。所以可以简单理解成如下：

Comparator<Person> comparator = **new** Comparator<Person>() {  
 @Override  
 **public int** compare(Person o1, Person o2) {  
 **return i**nteger1.*compareTo*(integer2);  
 }  
};

再来看comaring方法的返回值过程：

Comparator<Person> comparator = (Comparator<Person> & Serializable) (c1, c2) -> keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));

我们先理解(Comparator<> & Serializable)，这是类型强转，表示强转为Comparator 和Serializable，且必须同时能强转为这两种类型，否则会抛类转换异常，实验如下：

static class A<T> {}

static class B<T> extends A<T> { }

public static void main(String[] args) throws Exception {

B<String> b = new B<>();

Object c = b;

A<String> a = (A<String> & Serializable) b;

Serializable s = (A<String> & Serializable) b;

}

运行时将会抛出类转换异常，在第一次强转时就会抛出。让B实现Serializable即可解决问题。

不明白的是，通过Lambda表达式（(c1, c2) -> keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2))）由编译创建的Comparator对象是怎么会实现Seriailizable的 ？，难道是因为有Serializable的强转操作符，所以编译器在创建Comparator对象时就自动实现Serializable接口了？试验如下：

**public static interface** Haha<T> {  
 **void** func();  
}  
  
**public static void** main(String[] args) **throws** Exception {  
 Serializable s = (Haha<String> & Serializable) () -> System.***out***.println(**"hello!"**);  
 Haha<String> h = (Haha<String>) s;  
 h.func();  
}

运行通过，实验证明编译器看到有Serializable的强转操作，在创建Haha对象的时候自动实现了Serializable接口。

接下来看comparing方法中的泛型，先看Function的源码，如下：

**public interface** Function<T, R> {  
 R apply(T t);  
}

在comparing函数中声明了 Function变量，在声明Function变量时限定Function的第一个泛型可以是T或者T的父类，限定了第二个泛型可以是U或者U的子类，而这个U本身被限定为必须是Comparable或者继承自Comparable，而Comparable中的泛型为可以是Comparable或者Comparable的父类，这一点很奇葩，Comparable的父类只有Object，这样理解是错的，因为U可能是Comparable的子类，则这个子类它的父类应当是Comparable，所以可以是自己或者父类。

### 3.3.6 通用模式

不知道这节讲的有什么用

### 3.3.7 reduce

### 3.3.8 整合操作

## 3.4 重构遗留代码

## 3.5 多次调用流操作

## 3.6 高阶函数

## 3.7 正确使用Lambda表达式

# 第4章 类库

## 4.1 在代码中使用Lambda表达式

## 4.2 基本类型

## 4.3 重载解析

## 4.4 @FunctionalInterface

## 4.5 二进制接口的兼容性

## 4.6 默认方法

## 4.7 多重继承

## 4.8 权衡

## 4.9 接口的静态方法

## 4.10 Optional