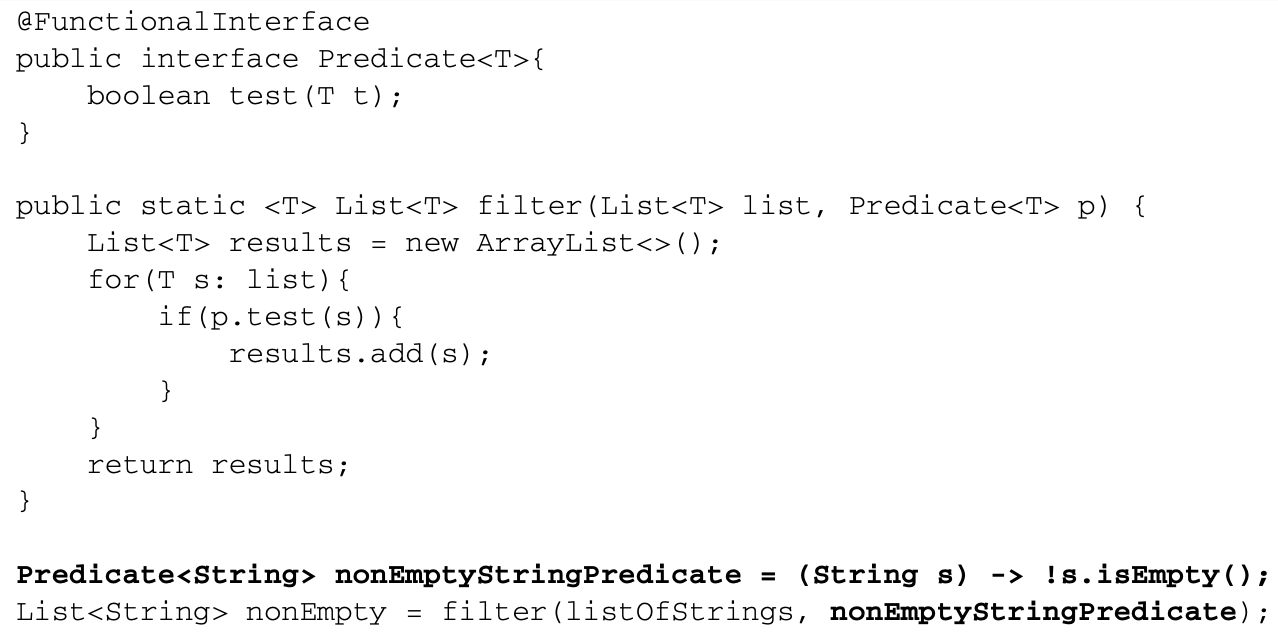
函数式接口定义且只定义了一个抽象方法。函数式接口很有用，因为抽象方法的签名可以描述Lambda表达式的签名。函数式接口的抽象方法的签名称为函数描述符。所以为了应用不同的Lambda表达式，你需要一套能够描述常见函数符的函数式接口。Java API中已经有了几个函数式接口，比如Comparable、Runnable和Callable。

Java 8的库设计师帮你在java.util.function包中引入了几个新的函数式接口。Predicate、Consumer和Function等。

#### 1、Predicate接口

java.util.function.Predicate<T>接口定义了一个名叫test的抽象方法，它接受泛型T对象，并返回一个boolean。在你需要表示一个涉及类型T的布尔表达式时，就可以使用这个接口。比如，你可以定义一个接受String对象的Lambda表达式，如下所示：



#### 2、Consumer接口

java.util.function.Consumer<T>定义了一个名叫accept的抽象方法，它接受泛型T的对象，没有返回（void）。你如果需要访问类型T的对象，并对其执行某些操作，就可以使用这个接口。比如，你可以用它来创建一个forEach方法，接受一个Integers的列表，并对其中每个元素执行操作。在下面的代码中，你就可以使用这个forEach方法，并配合Lambda来打印列表中的所有元素。

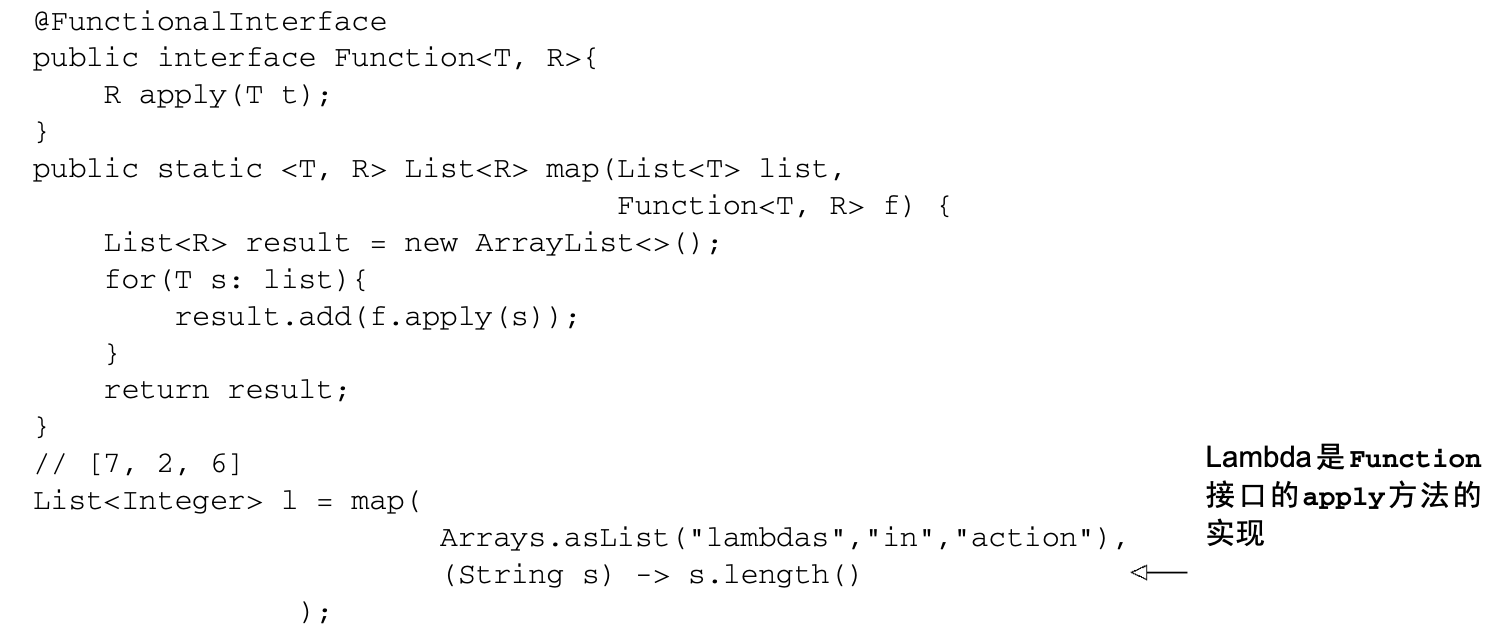
ArrayList里的forEach方法：

public void forEach(Consumer<? super E> action) {

Objects.*requireNonNull*(action);  
 final int expectedModCount = modCount;  
 @SuppressWarnings("unchecked")  
 final E[] elementData = (E[]) this.elementData;  
 final int size = this.size;  
 for (int i=0; modCount == expectedModCount && i < size; i++) {  
 action.accept(elementData[i]);  
 }  
 if (modCount != expectedModCount) {  
 throw new ConcurrentModificationException();  
 }  
}

#### 3、Function接口

java.util.function.Function<T, R>接口定义了一个叫作apply的方法，它接受一个泛型T的对象，并返回一个泛型R的对象。如果你需要定义一个Lambda，将输入对象的信息映射到输出，就可以使用这个接口(比如提取苹果的重量，或把字符串映射为它的长度)。在下面的代码中，我们向你展示如何利用它来创建一个map方法，以将一个String列表映射到包含每个String长度的Integer列表。



#### 原始类型特化

介绍了三个泛型函数式接口：Predicate<T>、Consumer<T>和Function<T,R>。还有些函数式接口专为某些类型而设计。

Java类型要么是引用类型，要么是原始类型。但是泛型（比如Consumer<T>中的T）只能绑定到引用类型。这是由泛型内部的实现方式造成的。因此，在Java里有一个将原始类型转换为对应的引用类型的机制。这个机制叫作装箱（boxing）。相反的操作，也就是将引用类型转换为对应的原始类型，叫作拆箱（unboxing）。Java还有一个自动装箱机制来帮助程序员执行这一块任务。比如，下面代码中一个int被装箱成为Integer：

List<Integer> list = new ArrayList<>();

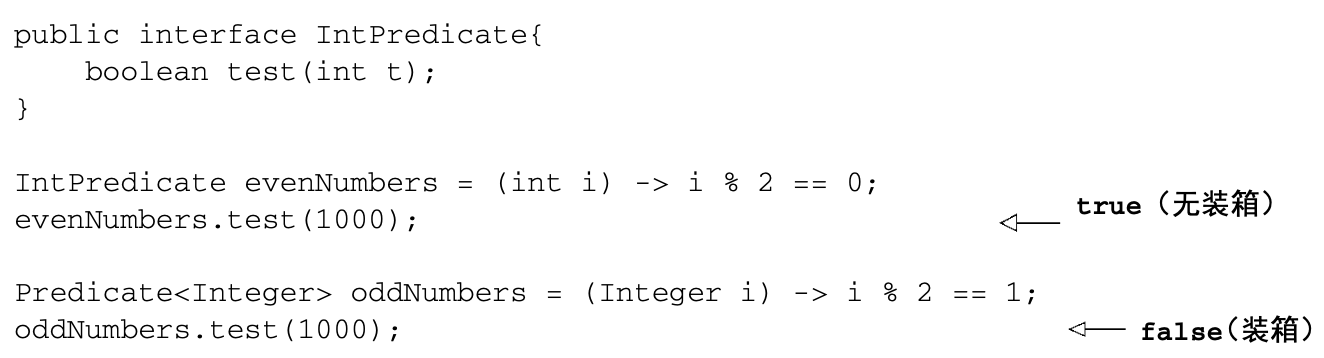
for (int i = 300; i < 400; i++){

list.add(i);

}

但这在性能方面是要付出代价的。装箱后的值本质上就是把原始类型包裹起来，并保存在堆里。因此，装箱后的值需要更多的内存，并需要额外的内存搜索来获取被包裹的原始值。

Java 8为我们前面所说的函数式接口带来了一个专门的版本，以便在输入和输出都是原始类型时避免自动装箱的操作。比如，在下面的代码中，使用IntPredicate就避免了对值1000进行装箱操作，但要是用Predicate<Integer>就会把参数1000装箱到一个Integer对象中：



一般来说，针对专门的输入参数类型的函数式接口的名称都要加上对应的原始类型前缀，比如DoublePredicate、IntConsumer、LongBinaryOperator、IntFunction等。Function接口还有针对输出参数类型的变种：ToIntFunction<T>、IntToDoubleFunction等。

|  |  |  |
| --- | --- | --- |
| 函数式接口 | 函数描述符 | 原始类型特化 |
| Predicate<T> | T->boolean | IntPredicate,  LongPredicate,  DoublePredicate |
| Consumer<T> | T->void | IntConsumer,  LongConsumer,  DoubleConsumer |
| Function<T,R> | T->R | IntFunction<R>  IntToDoubleFunction  IntToLongFunction  LongFunction<R>  LongToDoubleFunction  LongToIntFunction  DoubleFunction<R>  ToIntFunction<T>  ToDoubleFunction<T>  ToLongFunction<T> |
| Supplier<T> | ()->T | BooleanSupplier,IntSupplier, LongSupplier,  DoubleSupplier |
| UnaryOperator<T> | T->T | IntUnaryOperator,  LongUnaryOperator,  DoubleUnaryOperator |
| BinaryOperator<T> | (T,T)->T | IntBinaryOperator,  LongBinaryOperator,  DoubleBinaryOperator |
| BiPredicate<L,R> | (L,R)->boolean |  |
| BiConsumer<T,U> | (T,U)->void | ObjIntConsumer<T>,  ObjLongConsumer<T>,  ObjDoubleConsumer<T> |
| BiFunction<T,U,R> | (T,U)->R | ToIntBiFunction<T,U>,  ToLongBiFunction<T,U>,  ToDoubleBiFunction<T,U> |

很多函数式接口可以用于描述各种Lambda表达式的签名。

#### 总结关于Lambdas及函数接口的例子

|  |  |  |
| --- | --- | --- |
| 使用案例 | Lambda的例子 | 对应的函数式接口 |
| 布尔表达式 | (List<String> list) ->  list.isEmpty() | Predicate<List<String>> |
| 创建对象 | ()->new Apple(10) | Supplier<Apple> |
| 消费一个对象 | (Apple a)->  System.out,println(a.getWeight) | Consumer<Apple> |
| 从一个对象中  选择/提取 | (String s)->  s.length() | Function<String,Integer>或ToIntFunction<String> |
| 合并两个值 | (int a,int b)  ->a\*b | IntBinaryOperator |
| 比较两个对象 | (Apple a1,Apple a2)->  a1.getWeight().compareTo(a2.getWeight) | Comparator<Apple>或 BiFunction<Apple, Apple, Integer> 或 ToIntBiFunction<Apple, Apple> |

#### 异常、Lambda，还有函数式接口又是怎么回事呢？

请注意，任何函数式接口都不允许抛出受检异常(checked exception)。如果你需要Lambda表达式来抛出异常，有两种办法:定义一个自己的函数式接口，并声明受检异常，或者把Lambda 包在一个try/catch块中。

如下函数式接口BufferedReaderProcessor，它显式声明了一个IOException:

@FunctionalInterface

public interface BufferedReaderProcessor {

String process(BufferedReader b) throws IOException;

}

BufferedReaderProcessor p = (BufferedReader br) -> br.readLine();

但是你可能是在使用一个接受函数式接口的API，比如Function<T, R>，没有办法自己创建一个。这种情况下，

你可以显式捕捉受检异常:

Function<BufferedReader, String> f = (BufferedReader b) -> {

try {

return b.readLine();

}

catch(IOException e) {

throw new RuntimeException(e);

} };

接下来我们会介绍一些高级的细节：编译器如何对Lambda做类型检查，以及你应该了解的规则，诸如Lambda都在自身内部引用局部变量，还有和void兼容的Lambda等。