转载：<http://www.cnblogs.com/Stay-Hungry-Stay-Foolish/p/7435414.html>

# Lambda表达式

## 行为参数化

### 引入

在软件工程中，一个众所周知的问题就是：不管你做什么，用户的需求肯定会变。比如说，有一个应用程序是帮助农民了解自己的库存的。这位农民可能想有一个查找库存中所有的绿苹果的功能。但是到了第二天，他可能告诉你，其实还想找出所有重量超过150g的苹果。又过了两天，农民又跑过来补充道，要是我可以找出所有既是绿色，重量也超过150g的苹果，那就太好了。你要如何应对这样不断变化的需求呢？

行为参数化就是可以帮助你处理频繁变更的需求的一种软件开发模式。总之，它意味着拿出一个代码块，把它准备好却不去执行它。这个代码块以后可以被你的程序的其他部分调用，这就表示你可以推迟这个代码块的执行。例如，你可以将代码块作为参数传给一个方法，在这个方法里面再去执行它。这样，这个方法的行为就基于那个代码块被参数化了。

编写能够应对变化的需求的代码并不容易。让我们看看一个例子，我们会逐步地改进这个例子，以展示一些让代码更加灵活的最佳做法。就农场库存程序而言，你必须实现一个从列表中筛选绿苹果的功能。

第一个解决方案可能是下面这样的：

private List<Apple> filterGreenApple(List<Apple> inventory) {

List<Apple> result = new ArrayList<>(); // 累积的苹果列表

for (Apple apple : inventory) {

if ("green".equals(apple.getColor())) { //仅仅筛选出绿苹果

result.add(apple);

}

}

return result;

}

这里筛选条件是筛选绿苹果。但是现在农民改主意了，他还要筛选红苹果，你该怎么做呢？简单的方法就是复制这段代码，将名字改为filterRedApple，然后更改if条件来匹配红苹果。然而，要是农民想要筛选多种颜色：浅绿色、暗红色、黄色等等，这种方法就应付不了。一个良好的原则是在编写类似的代码之后，尝试将其抽象化。

一种做法是给方法加一个参数，把颜色变为参数，这样就能灵活的适应变化了：

private List<Apple> filterAppleByColor(List<Apple> inventory, String color) {

List<Apple> result = new ArrayList<>();

for (Apple apple : inventory) {

if (color.equals(apple.getColor())) {

result.add(apple);

}

}

return result;

}

现在，只要像下面这样调用方法，农民朋友就会满意了：

List<Apple> greenApples = filterAppleByColor(inventory, "green");

List<Apple> redApples = filterAppleByColor(inventory, "red");

太简单了对吧？那我们在把例子弄得更加复杂一点儿。这位农民又跑过来和你说，要是能够区分轻的苹果和重的苹果就太好了，重的苹果一般是重量大于150g。于是你写了下面的方法，用一个参数来应对不同的重量：

private List<Apple> filterAppleByWeight(List<Apple> inventory, int weight) {

List<Apple> result = new ArrayList<>();

for (Apple apple : inventory) {

if (apple.getWeight() > weight) {

result.add(apple);

}

}

return result;

}

解决方法不错，但是复制大部分的代码来实现遍历库存，并对每一个苹果应用筛选条件。这让人有点失望，因为它打破了DRY(Don't Repeat Yourself，不要重复自己)的软件工程原则。如果你想要改变筛选方式来提升性能，那就得修改所有方法的实现，而不是只改一个。从工程的工作量角度来看，这个代价太大了。

### 行为参数化

在上面已经看到了，你需要一种比添加很多参数更好的方法来应对变化的需求。让我们后退一步来看看更高层次的抽象。一种可能的解决方案是对你的选择标准进行建模。让我们定义一个接口来对选择标准进行建模：

public interface ApplePredicate {

boolean test(Apple apple);

}

现在可以使用ApplePredicate的多个实例代表不同的选择标准了，比如：

public class AooleHeavyWeightPericate implements ApplePredicate { //选出重的苹果

@Override

public boolean test(Apple apple) {

return apple.getWeight() > 150;

}

}

public class AppleGreenColorPerdicate implements ApplePredicate { //选择绿苹果

@Override

public boolean test(Apple apple) {

return "green".equals(apple.getColor());

}

}

使用ApplePredicate改过之后，filter方法看起来是这样的：

private List<Apple> filterApple(List<Apple> inventory, ApplePredicate p){

List<Apple> result = new ArrayList<>();

for (Apple apple : inventory) {

if (p.test(apple)) {

result.add(apple);

}

}

return result;

}

我们都知道，人们不愿意使用那些很麻烦的功能或者概念。目前，当要把新的行为传递给filterApples方法的时候，你不得不声明好几个实现ApplePredicate接口的类，然后去实例化好几个只会提到一次的ApplePredicate对象。下面的程序总结了你目前看到的一切。这个真是非常的啰嗦，很费时间！

public class AppleHeavyWeightPericate implements ApplePredicate { //选出重的苹果

@Override

public boolean test(Apple apple) {

return apple.getWeight() > 150;

}

}

public class AppleGreenColorPerdicate implements ApplePredicate {//选择绿苹果

@Override

public boolean test(Apple apple) {

return "green".equals(apple.getColor());

}

}

public class Demo {

public static void main(String[] args) {

List<Apple> inventory = Arrays.asList(

new Apple("green", 80),

new Apple("green", 155),

new Apple("red", 120));

List<Apple> greenApples = filterApple(inventory,

new AppleGreenColorPerdicate());

List<Apple> heavyWeightApples= filterApple(inventory,

new AppleHeavyWeightPericate());

}

private static List<Apple> filterApple(List<Apple> inventory, ApplePredicate p) {

List<Apple> result = new ArrayList<>();

for (Apple apple : inventory) {

if (p.test(apple)) {

result.add(apple);

}

}

return result;

}

}

费这么大劲儿真没必要，能不能做的更好呢？

### 匿名类

 Java有一个机制称为匿名类，它可以让你同时声明和实例化一个类，这可以帮助你进一步改善代码，让代码变得更加的简洁。

匿名类和你熟悉的Java局部类(块中定义的类)差不多，但匿名类没有名字。它允许你同时声明并且实例化一个类。换句话说，它允许你随用随建。

下面的代码展示了如何通过创建一个匿名类实现ApplePredicate的对象，重写筛选的例子：

List<Apple> redApples = filterApple(inventory, new ApplePredicate() {

@Override

public boolean test(Apple apple) {

return "red".equals(apple.getColor());

}

});

但匿名类还是不够好。它往往显得非常的笨重，因为占用了很多空间。比如：

List<Apple> redApples = filterApple(inventory, new ApplePredicate() {

@Override

public boolean test(Apple apple) {

return "red".equals(apple.getColor());

}

});

List<Apple> greenApples = filterApple(inventory, new ApplePredicate() {

@Override

public boolean test(Apple apple) {

return "green".equals(apple.getColor());

}

});

上面代码中红色部分都是重复的。

整体来说，啰嗦就不好，它让人不愿意使用语言的某种功能，因为编写和维护啰嗦的代码需要很多时间，而且代码也不易读。好的代码应该是一目了然的。即使使用匿名类处理在某种程度上盖过了为一个接口声明好几个实体类的啰嗦问题，但是它仍然不能令人满意。

### Lambda表达式

上面的代码在Java8里可以使用Lambda表达式重写为下面的样子：

List<Apple> result = filterApple(inventory,

(Apple apple) ->"red".equals(apple.getColor()));

 不得不承认这代码看上去比先前干净很多。这个很好，因为它看起来更像问题描述本身了。我们现在已经解决了啰嗦的问题了。

 在通向抽象的路上，我们还可以更进一步。目前，filterApples方法还只适用于Apple。你还可以将List类型抽象化，从而超越你眼前要处理的问题：

public interface ApplePredicate<T> {

boolean test(T t);

}

public class Demo {

public static void main(String[] args) {

List<Apple> inventory = Arrays.asList(

new Apple("green", 80),

new Apple("green", 155),

new Apple("red", 120));

List<Apple> result = filterApple(inventory,

(Apple apple) ->"red".equals(apple.getColor()));

for(Apple apple : result){

System.out.println(apple);

}

}

private static <T>List<T> filterApple(List<T> inventory, ApplePredicate<T> p) {

List<T> result = new ArrayList<>();

for (T t : inventory) {

if (p.test(t)) {

result.add(t);

}

}

return result;

}

}

现在你可以把filter方法用在了香蕉、橘子、Integer或是String的列表上了。酷不酷？你现在在灵活性和间接性之间找到了最佳的平衡点，这在Java8之前是不可能的！

### 真实的例子

 你现在已经看到了，行为参数化是一个很有用的模式，它能够轻松的适应不断变化的需求。这种模式可以把一个行为(一段代码)封装起来，并通过传递和使用创建的行为将方法的行为参数化。这种做法类似于策略设计模式。

1、使用Comparator来排序

 对集合进行排序是一个常见的编程任务。比如，你的那位农民朋友想要根据苹果的重量对库存的进行排序，或者他可能改变了主意，希望你根据颜色对苹果进行排序，听起来怎么有点耳熟呢？是的，你需要一种方法来表示和使用不同的排序行为，来轻松的适应变化的需求。

在Java8中，List自带一个sort方法(你也可以使用Collections.sort)。sort的行为可以用Java.util.Comparator对象来进行参数化，它的接口如下：

public interface Comparator<T> {

int compare(T o1, T o2);

}

因此，你可以随时创建Comparator的实现，用sort方法表现出不同的行为。比如，你可以使用匿名类，按照重量升序对库存进行排序：

inventory.sort(new Comparator<Apple>() {

@Override

public int compare(Apple o1, Apple o2) {

return o1.compareTo(o2);

}

});

用Lambda表达式的话，看起来就是这样：

inventory.sort((Apple o1, Apple o2) ->o1.compareTo(o2));

2、使用Runnable执行代码块

线程就像是轻量级的进程:它们自己执行一个代码块。但是怎么才能告诉线程要执行那块代码呢？多个线程可能会运行不同的代码。我们需要一种方式来代表稍后执行的一段代码。在Java里面，你可以使用Runnable接口表示一个要执行的代码块。请注意，代码不会返回任何结果(即void):

你可以像下面这样，使用这个接口创建执行不同行为的线程：

Thread t = new Thread(new Runnable() {

@Override

public void run() {

System.out.println("hello world");

}

});

使用Lambda表达式，看起来是这样的：

Thread t = new Thread(()->System.out.println("hello world"));

## 语法

使用匿名类来表示不同的行为并不令人满意:代码十分的啰嗦，这会影响程序员在实践中使用行为参数化的积极性。在这里，你会认识Java8中解决这个问题的工具Lambda表达式，它可以让你很简洁地表示一个行为或者传递代码。

你可以把Lambda表达式理解为简洁地表示可传递的匿名函数的一种方式：它没有名称，但它有参数列表、函数主体、返回类型，可能还有可以抛出的异常列表。Lambda表达式的基本语法是：

(parameters) -> expression

或者：

(parameters) -> {expression;}

注意，带括号与不带括号有一定的区别：不带括号只能是一行语句，并且不能带return，末尾不能加分号；带括号可以是一行也可以是多行，但是如果函数有返回值的话，在末尾必须return，而且每一句后面必须带分号。例如：() -> "hello world"等价于 () -> { return "hello world";}

## 函数式接口

我们在之前实现filter方法中是这样使用Lambda的：

List<Apple> result = filterApple(inventory, (Apple apple) ->"red".equals(apple.getColor()));

那么，在哪里可以使用Lambda呢？你可以在函数式接口上使用Lambda表达式。在上面的代码中，你可以把Lambda表达式作为第二个参数传递给filter方法，这里需要一个Predicate<T>接口，而这个接口就是一个函数式接口，因为Predicate接口仅仅定义了一个抽象方法：

public interface ApplePredicate<T> {

boolean test(T t);

}

简而言之，函数式接口就是只定义了一个抽象方法的接口。注意，Java8中的接口还可以默认方法和静态方法，但只要接口只定义了一个抽象方法，该接口就仍然是一个函数式接口。

用函数式接口可以干什么呢？Lamba表达式允许你可以直接以内联的形式为函数式接口的抽象方法提供实现，并把整个表达式作为函数式接口的实例。你用匿名内部类也可以完成同样的事情，只不过比较笨拙：需要提供一个实现，然后再直接内联将其实例化。如下；

// 使用Lambda

Runnable r1 = () -> System.out.println("hello world");

// 使用匿名类

Runnable r2 = new Runnable() {

@Override

public void run() {

System.out.println("Hello World");

}

};

## 函数描述符

函数式接口的抽象方法的签名基本上就是Lambda表达式的签名，我们把这种抽象方法叫做函数描述符，例如：() -> void代表参数列表为空且返回void的函数，(Apple, Apple) -> int代表接收两个Appe作为参数并且返回int的函数。

你可能在想，Lambda表达式时如何做类型检查的，这个之后再细说。现在，你只需要知道Lambda表达式可以被赋值给一个变量，或者传递给一个接收函数式接口作为参数的方法就行，当然，这个Lambda表达式的签名要和函数式接口的抽象方法的签名一致，例如：

public void process(Runnable r) {

r.run();

}

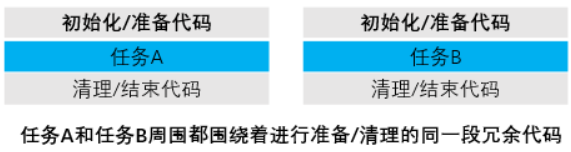
process(() -> System.out.println("hello world"));

上面的代码打印的是"hello world"，Lambda表达式：() -> System.out.println("hello world")不接收参数并且返回void，这个恰恰就是Runnable接口中run()方法的签名。

 如果你去看看新的Java API，会发现函数式接口带有@FunctionalInterface的注解。这个注解用于表示该接口会设计成一个函数式接口，如果使用了@FunctionalInterface定义了一个接口，而它却不是函数式接口的话，编译器将返回一个提示原因的错误，例如错误信息可能是:Multiple non-overriding abstract methods found in interface Foo,表明存在多个抽象方法。需要注意的是：@FunctionalInterface不是必需的，但是对于为此设计的接口而言，使用它是比较好的做法，它就好像是@Override注解表示方法被重写了。

## 环绕执行模式

 让我们通过一个例子，看看在实践中如何利用Lambda和行为参数化来让代码更加灵活，更加简洁。资源处理(例如处理文件或者数据库)时，一个常见的模式就是打开一个资源，做一些处理，然后再关闭资源。这个打开资源和清理资源阶段总是很类似，并且会围绕着执行处理的代码，这就是所谓的环绕执行(execute around)模式。如下图所示：



例如：在以下代码中，红色部分就是从一个从文件中读取一行所需的模板代码(注意：Java7中，带资源的try语句会自动关闭资源，所以你不需要显示地关闭资源了)：

public static String processFile() throws IOException{

try(BufferedReader read = new BufferedReader(new FileReader("D://data.txt"))){

return read.readLine();

}

}

上面那段代码是有局限性的，你只能读文件的第一行，如果你想要读取文件的前两行，甚至是返回使用最频繁的词，该怎么办呢？在理想的情况下，你要重用打开资源和清理资源的代码，并且告诉processFile()方法对文件执行不同的操作。这听起来是不是很耳熟呢？是的，你需要把processFile的行为参数化。你需要一种方法把行为传递给processFile方法，以便它可以利用BufferedReader执行不同的行为。

（1）创建函数式接口

public interface BufferedReaderProcessor {

String process(BufferedReader b) throws IOException;

}

（2）执行一个行为

任何BufferedReader -> String形式的Lambda都可以可以作为参数来传递，因为它们都符合BufferedReaderProcessor接口中定义的process方法的签名。请记住，Lambda表达式允许你直接内联为函数式接口的抽象方法提供实现，并且将整个表达式作为函数式接口的一个实例。因此，你可以在processFile主体内，对传递过来的BufferReaderProcessor对象调用process方法执行处理：

public static String processFile(BufferedReaderProcessor b) throws IOException{

try(BufferedReader br = new BufferedReader(new FileReader("D://data.txt"))){

return b.process(br);

}

}

（3）传递Lambda

现在你就可以传递不同的Lambda来重用precessFile方法，并用不同的方式处理文件了。

处理一行：

String result = processFile((BufferedReader p) -> p.readLine());

处理两行：

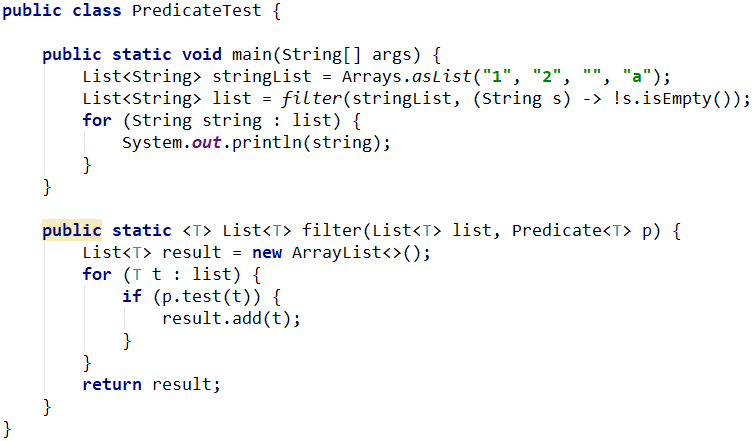
String result = processFile((BufferedReader p) -> p.readLine() + p.readLine());

## 系统中的函数式接口

函数式接口的抽象方法的签名被称为函数描述符。因为函数式接口的抽象方法的签名可以描述Lambda表达式的签名，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描述符的函数式接口。Java API中已经提供了几个函数式接口，例如Runnable、Callalbe、Comparable，此外，Java8中的java.util.function包中还引入了几个新的函数式接口，例如Predicate、Consumer、Function等。

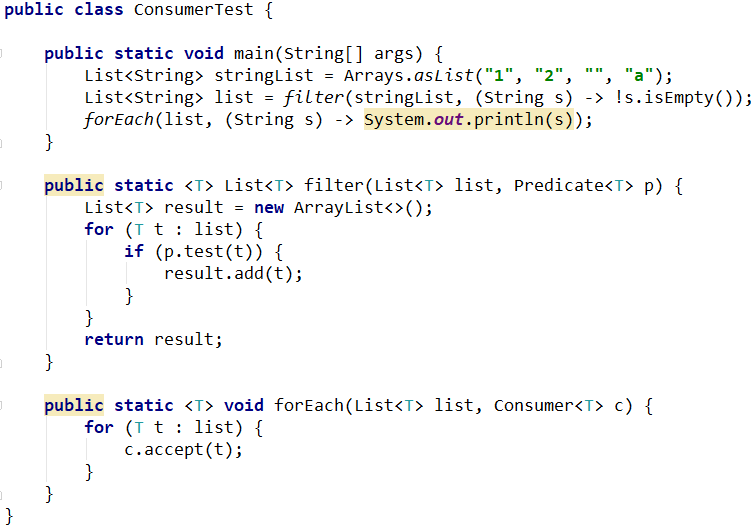
1. Predicate

java.util.function.Predicate<T>接口定义了一个名为test的抽象方法，它接收泛型T对象，并返回一个boolean类型。



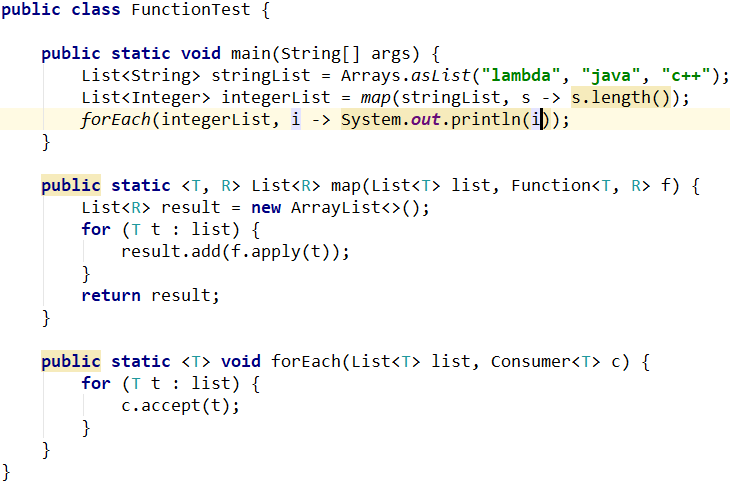
2、Consumer

Java.util.function.Consumer<T>定义了一个名为accept的抽象方法，它接收泛型T对象，并返回void。



3、Function

Java.util.function.Function<T, R>接口定义了一个名为apply的方法，它接收一个泛型T对象，并返回一个泛型R对象。如果你需要定义一个Lambda，将输入对象映射到输出对象，就可以使用这个接口。



4、原始类型特化

我们已经介绍了三个泛型函数式接口：Predicate<T>、Consumer<T>和Function<T, R>，还有一些函数式接口是专为原始类型设计的。

回顾一下：Java类型要么是引用类型，比如Byte、Integer、Object、List等，要么是原始类型，比如int、double、byte、char等。但是泛型只能绑定到引用类型，这是由于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，这个机制叫做装箱，而相反的操作，即将引用类型转换为对应的原始类型，就叫做拆箱。另外，Java还有一个自动装箱拆箱机制来帮助程序员执行这个任务，即装箱和拆箱是自动完成的，但是，自动装箱拆箱在性能方面要付出代价：装箱的本质就是把原始类型包裹起来，并保存到堆里，因此装箱后的值需要更多的内存，且需要额外的内存搜索来获取被包裹的原始类型值。

Java8为原始类型提供了专门的函数式接口，以便在输入和输出都是原始类型时避免自动装箱操作。比如下面的代码中，使用intPredicate就避免对值1000进行装箱操作，但如果使用Predicate<Integer>，则会把参数1000装箱到一个Integer对象中：

IntPredicate evenNumbers = (int i) -> i % 2 == 0;

evenNumbers.test(1000);

Predicate<Integer> oddNumbers = (Integer i) -> i % 2== 0;

oddNumbers.test(1000);

一般来说，针对输入参数类型为原始类型的函数式接口的名称都会加上对应的原始类型前缀，比如DoublePredicate、IntConsumer、LongBinaryOperator、IntFunction<R>等。另外，Function接口还有针对输出参数类型为原始类型的函数式接口，比如ToIntFunction<T>、IntToDoubleFucntion等。

下表中总结了Java API中提供的最常用的函数式接口及其函数描述符。

| 函数式接口 | 函数描述符 | 原始类型转换 |
| --- | --- | --- |
| Predicate<T> | T –> boolean | IntPredicate、LongPredicate、DoublePredicate |
| Consumer<T> | T -> void | IntConsumer、LongConsumer、DoubleConsumer |
| Supplier<T> | () -> T | BooleanSupplier、IntSupplier、LongSupplier、DoubleSupplier |
| Function<T, R> | T -> R | IntFunction<R>、LongFunction<R>、DoubleFunction<R>、ToIntFunction<T>、  ToLongFunction<T>、ToDoubleFunction<T>、  IntToLongFunction、IntToDoubleFunction、  LongToDoubleFunction、LongToIntFucntion、  DoubleToIntFunction、DoubleToLongFucntion |
| 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> |
| UnaryOperator<T> | T -> T | IntUnaryOperator、LongUnaryOperator、DoubleUnaryOperator |
| BinaryOperator<T> | (T, T) -> T | IntBinaryOperator、LongBinaryOperator、DoubleBinaryOperator |

下表中总结了使用函数式接口的一些使用案例。

| 案例 | Lambda | 函数式接口 |
| --- | --- | --- |
| 布尔表达式 | (List<String> list) -> list.isEmpty() | Predicate<List<String>> |
| 创建对象 | () -> new Apple() | Supplier<Apple> |
| 消费对象 | (Apple a) -> System.out.println(a.geWeight()) | Consumer<Apple> |
| 过滤对象 | (String s) -> s.length() | Function<String, Integer>、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.compareTo(a2) | Compartor<Apple>、BiFunction<Apple, Apple, Integer>、ToIntBiFunction<Apple, Apple> |

## Lambda表达式异常

如果你需要Lambda表达式抛出异常，有两种做法：

1. 在定义对应的函数式接口时，让其抛出一个检查性异常
2. 把Lambda表达式放在try-catch语句块中

比如，之前我们定义了一个函数式接口BufferedReaderProcessor，它就显式声明了一个IOException：

public interface BufferedReaderProcessor {

String process(BufferedReader br) throws IOException;

}

如果你使用的是Java8中的Function<T, R>，这个函数式接口没有显式声明一个检查新异常，因此，你需要捕获异常：

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

try {

return b.readLine();

} catch (IOException e) {

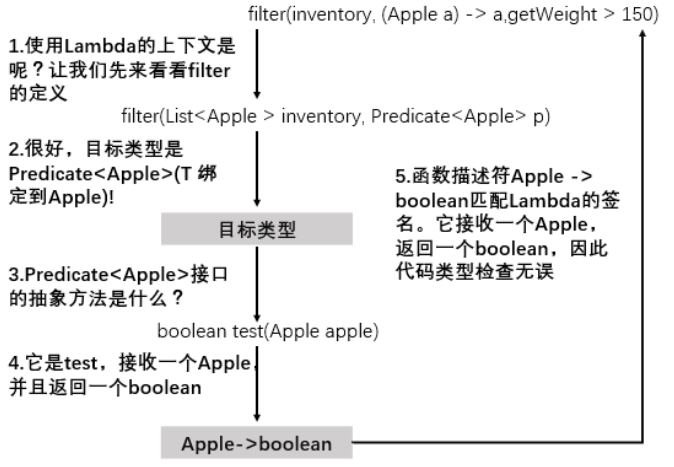
throw new RuntimeException();

}

};

## 类型推导

Lambda的类型是从使用Lambda的上下文推导出来的，上下文中Lambda表达式所需要的类型被称为目标类型。让我们通过一个例子，看看当使用Lmabda表达式时，背后发生了什么。



有了目标类型的概念，同一个Lambda表达式就可以与不同的函数式接口联系起来，只要它们的抽象方法签名能够兼容。比如，前面提到的Callable和PrivilegedAction，这两个接口都代表着什么也不接收并且返回一个泛型T的函数。因此，下面两个赋值是有效的：

Callable<Integer> c = () -> 42;

PrivilegedAction<Integer> p = () -> 42;

这里，第一个赋值的目标类型是Callable<Integer>, 第二个赋值的目标类型是PrivilegedAction<Integer>。

同理，下面三个赋值也是有效的：

Comparator<Apple> c1 = (Apple a1, Apple a2) -> a1.compareTo(a2);

ToIntBiFunction<Apple, Apple> c2 = (Apple a1, Apple a2) -> a1.compareTo(a2);

BiFunction<Apple, Apple, Integer> c3 = (Apple a1, Apple a2) -> a1.compareTo(a2);

注意，如果一个Lambda表达式的函数体是一个语句表达式，它就和一个返回void的函数描述符兼容，例如，以下两行都是合法的，尽管List的add方法返回一个boolean，而不是Consumer所需的void：

//Predicate返回了一个boolean

Predicate<String> p = (String s) -> stringList.add(s);

//Consumer返回了一个void

Consumer<String> b = s -> stringList.add(s);

另外，编译器会从上下文(目标类型)推断出用什么函数式接口来配合Lambda表达式，这就意味着它也可以推断出Lambda的签名，所以我们可以在Lambda语法中就不用显式的给出参数类型，从而提高代码的可读性。也就是说，Java编译器会像下面这样推断Lambda的参数类型

//apple 没有显式的给出类型

List<Apple> result = filterApple(inventory, apple ->"red".equals(apple.getColor()));

Lambda表达式有多个参数，代码可读性的好处就显得更加明显，例如创建一个Comparator对象：

Comparator<Apple> c1 = (Apple a1, Apple a2) -> a1.compareTo(a2);

Comparator<Apple> c2 = (a1, a2) -> a1.compareTo(a2);

## 变量捕获

我们迄今为止所介绍的所有Lambda表达式都只用在方法的参数上，但是Lambda表达式也允许捕获变量，例如下面的Lambda捕获了portNumber变量：

int portNumber = 1337;

Runnable r = () -> System.out.println(portNumber);

对于变量捕获，Lambda有一些限制：Lambda可以没有限制的捕获实例变量和静态变量，但是局部变量必须显式的声明为final，或者本质上是final变量，即如果一个局部变量在初始化后从未被修改过，那么它就可以被捕获，换句话说，加上final后也不会导致编译错误的局部变量就是可以被Lambda捕获的变量。例如，下面的变量无法编译，因为portNumer变量被赋值了两次：

int portNumber = 1337;

Runnable r = () -> System.out.println(portNumber);

portNumber = 10;

## 方法引用

 方法的引用让你可以重复使用现有的方法定义，并像Lambda一样传递它们。在一些情况下，比起使用Lambda表达式，方法引用似乎更加容易读，感觉也更加自然。下面就是我们借助更新的Java8 API，用方法的引用写的一个排序例子：

// 先前

Inventory.sort((Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

// 之后

Inventory.sort(Comparator.comparing(Apple::getWeight));

方法引用可以被看做是仅仅调用特定方法的Lambda的一种快捷写法，它的基本思想是：如果一个Lambda表达式只是“直接调用这个方法”，那最好还是用名称来调用它，而不是去描述如何调用。事实上，方法引用就是让你根据已有的方法实现来创建Lambda表达式，其工作方式是：当你需要使用方法引用时，将目标引用放在分隔符::前，方法名称放在分隔符后。例如Apple::getWeight就是引用了Apple类中定义的getWeight方法，这是Lambda表达式(Apple a) -> a.getWeight()的快捷写法。注意，不需要括号，因为你没有实际调用这个方法。下表给出了Java8中方法引用的其他例子：

| Lambda表达式 | 等效的方法引用 |
| --- | --- |
| (Apple a) -> a.getWeight() | Apple::getWeight |
| () -> Thread.currentThread.dumpstack() | Thread.currentThread::dumpstack |
| (String str, int i) -> str.substring(i) | String::substring |
| (String s) -> System.out.println(s) | System.out::println |

你可以把方法引用看做是一个针对仅仅涉及单一方法的Lambda语法糖，因为你表达同样的事情时要写的代码更加少了。方法引用和lambda表达式拥有相同的特性，比如它们都需要一个目标类型，并需要被转换为函数式接口的实例，不过我们并不需要为方法引用提供方法体，而是可以直接通过方法名称引用已有方法。

方法引用有很多种，它们的语法如下：

静态方法引用：ClassName:: methName

对象的实例方法引用：instanceRef::methName

对象的超类方法引用：super::methName

类的任意对象上的实例方法引用：ClassName::methName

构造方法引用：Class::new

数组构造方法引用：TypeName[]::new

举几个例子。

1、对一个字符串的List排序，忽略大小写。

List<String> str = Arrays.asList("a", "b", "c", "d");

// str.sort((s1, s2) -> s1.compareToIgnoreCase(s2));

str.sort(String::compareToIgnoreCase);

2、将一个整型字符串转换成整型。

// Function<String, Integer> strToInt = s -> Integer.parseInt(s);

Function<String, Integer> strToInt = Integer::parseInt;

3、判断一个List是否包含某个元素。

// BiPredicate<List<String>, String> contains = (list, element) -> list.contains(element);

BiPredicate<List<String>, String> contains = List::contains;

4、构造无参对象

// Supplier<Apple> supplier = () -> new Apple();

// Supplier<Apple> supplier = new Supplier<Apple>() {

// @Override

// public Apple get() {

// return new Apple();

// }

// }

Supplier<Apple> supplier = Apple::new; // 指向Apple()的构造函数引用

Apple apple = supplier.get();

5、构造带有一个参数的对象

// IntFunction<Apple> f = (weight) -> new Apple(weight);

IntFunction<Apple> f = Apple::new;; // 指向Apple(String color)的构造函数引用

Apple anotherApple = f.apply(70);

6、构造带有两个参数的对象

// BiFunction<Integer, String, Apple> bf = (weight, color) -> new Apple(weight, color);

// 指向Apple(String color, Integer weight)的构造函数引用

BiFunction<Integer, String, Apple> bf = Apple::new;

Apple yetAnotherApple = bf.apply(70, "red");

7、构造带有三个参数的对象

你已经看到了如何将0个、1个、2个参数的构造参数转变为构造函数引用，那要怎么样才能对三个参数的构造函数，比如Color(int r, int g, int b)，使用构造函数引用呢？

 构造函数引用的语法是ClassName::new，那么在这个例子里面就是Color::new，但是你需要与构造函数引用的签名匹配的函数式接口。因为语言本身没有并没有提供这样的函数式接口，所以你可以自己创建一个：

@FunctionalInterface

interface TriFunction<T, U, V, R> {

R apply(T T, U U, V v);

}

现在你就可以像下面这样使用构造函数引用了：

TriFunction<Integer, Integer, Integer, Color> f = Color::new;

## 复合Lambda表达式

Java8的好几个函数式接口都有为方便使用而设计的方法，比如Comparator、Function和Predicate都提供了允许你进行复合的方法。这是什么意思呢？在实践中，这意味着你可以把多个简单的Lambda复合成为复杂的表达式，比如，你可以让两个谓词之间做一个or操作从而组合成一个更大的谓词。而且，你还可以让一个函数的结果成为另一个函数的输入。你可能会在想，函数式接口中怎么可能会有更多的方法呢？毕竟，这违背了函数式接口的定义。关键在于，我们即将介绍的方法都是默认方法，也就是说它们不是抽象方法。

### 比较器复合

  我们前面看到了，你可以使用Comparator.comparing来返回一个Comparator对象，如下所示：

inventory.sort(Comparator.comparing(Apple::getWeight));

（1）逆序

如果你想要对苹果按质量递减排序怎么办？用不着去建立另一个Comparator的实例。接口有一个默认方法reversed可以使给定的比较器逆序排序：

inventory.sort(Comparator.comparing(Apple::getWeight).reversed());  
 （2）比较器链

如果发现有两个苹果一样重，哪个苹果应该排在前面呢？你可能需要再提供一个Comparator来进一步定义这一个比较。比如，在按重量比较两个苹果之后，你可能想要按颜色排序。thenComparing方法就是用来做来这个的，它接收一个函数作为参数(就像comparing方法一样)，如果两个对象用第一个Comparator比较之后是一样的，就提供第二个Comparator。你又可以优雅的解决这个问题：

inventory.sort(Comparator.*comparing*(Apple::getWeight).reversed());

### 谓词复合

 谓词接口包括三个方法：negate、and和or，让你可以重用已有的Predicate来创建更加复杂的谓词。比如你可以使用negate方法来返回一个Predicate的非：

Predicate<Apple> redApple = apple -> apple.getColor().equals("red");

Predicate<Apple> notRedApple = redApple.negate();

你可能想要把两个Lambda用and方法组合起来，比如一个苹果既是红色又比较重：

Predicate<Apple> redAndHeavyApple = redApple.and(apple -> apple.getWeight() > 150);

你可以进一步组合谓词，比如要么是重(150g以上)的红苹果，要么是绿苹果：

Predicate<Apple> redAndHeavyOrGreenApple = redApple

.and(apple -> apple.getWeight() > 150)

.or(apple -> apple.getColor().equals("green"));

注意，and和or是按照在表达式链中的位置，从左往右确定优先级的。因此，a.or(b).and(c)可以看做(a || b) && c。

### 函数复合

最后，你可以把Function接口代表的Lambda表达式复合起来。Function接口为此配了andThen和compose两个默认方法，它们都会返回Function的一个实例。

andThen方法会返回一个函数，它将输入应用到一个给定函数，再将输出应用到另一个函数。比如，假设有一个函数f数字加1(x -> x + 1)，另一个函数给g给数字乘以2，你可以将它们组合成一个函数h，先给数字加1，再给结果乘以2，即g(f(x))。

Function<Integer, Integer> f = x -> x + 1;

Function<Integer, Integer> g = x -> x \* 2;

Function<Integer, Integer> h = f.andThen(g);

int result = h.apply(1);

compose方法与andThen相反，即f(g(x))。

# 流

## 引入

集合对于很多编程任务来说都是非常基本的，几乎对于任何一个Java程序都是不可或缺的，但是集合操作却远远算法上完美。考虑下面两种情况：

1、很多任务逻辑都涉及类似于数据库的操作，比如对想象一下你有一系列菜组成的菜单，你想对菜按照类别进行分类(比如全素菜肴)，或者查找出最贵的菜。大部分数据库都允许你声明式地指定这些操作，比如，以下sql查询语句就可以选出热量较低的菜肴名称：

select name from dishes where calorie < 400

你看，你不需要实现如何根据菜肴的属性进行筛选，你只需要表达你想要什么。这个基本的思路意味着，你用不着担心怎么去显式地实现这些查询语句。

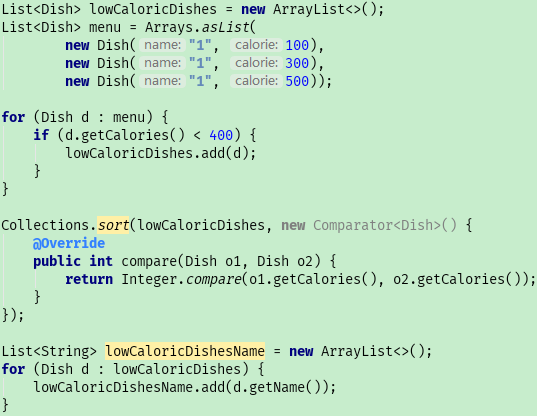
2、要是要处理大量元素又该怎么办呢？为了提供性能，你需要并行处理，并且利用多核架构。但是写并行代码比用迭代器还要复杂，而且调试起来也十分困难。

那Java语言的设计者能做些什么，来帮助程序员节约宝贵的时间呢？你可能已经猜到了，答案就是流。

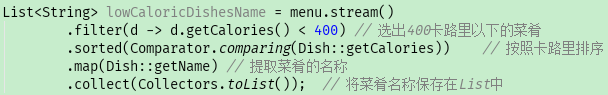
## 流是什么

 流是Java API的新成员，它允许你以声明的方式处理数据集合（通过查询语句来表达，而不是编写一个实现）。此外，流还可以透明地进行并行处理，你无需编写任何多线程代码。

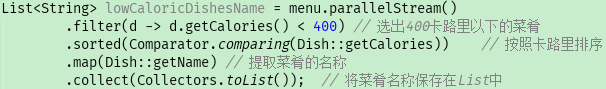
下面我们来简单的看看流的好处。以下两段代码的功能都是返回低热量的菜肴名称，并按照卡路里排序。一个是用Java7写的，另一个是用Java8的流写的。



在这段代码中，你使用了一个“垃圾变量”lowCaloricDishes，它唯一的作用就是作为一次性中间容器。



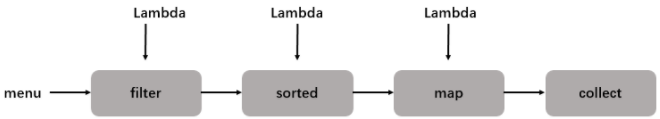
为了利用多核架构并行执行这段代码，你只需要把stream换成parallelStream()：



从软件工程师的角度来看，新的方法有几个显而易见的好处：

1、代码是以声明式方法来写的，说明了想要完成什么(筛选热量低的菜肴)，而不是说明如何实现一个操作(利用循环和if条件等控制流语句)。你在之前也看到了，这种方法加上行为参数化让你可以轻松应对变化的需求：你很容易再创建一个代码版本，利用Lamba表达式来筛选高卡路里的菜肴，而不是去复制粘贴代码。

2、你可以把几个基础操作筛选链接起来，来表达复杂的数据处理流水线(在filter后面接上sorted、map和collect操作，如下图所示)，同时保持代码清晰可读。filter的结果被传给了sorted方法，再传给map方法，最后传给collect方法。



因为filter、sorted、map和collect等操作是与具体线程模型无关的高层次构件，所以它们的内部实现可以是单线程的，也可能透明地充分利用你的多级程架构。在实践中，这意味着你用不着为了让某些数据处理任务并行而去操心线程和锁，Stream API都替你想好了。

## 流简介

 要讨论流，我们先来谈谈集合，这是最容易上手的方法。Java8中的集合支持一个新的stream方法，它会返回一个流(接口定义在java.util.stream.Stream里)。你在后面会看到，还有很多的方式来可以得到流，比如利用数值范围或从I/O资源生成的流元素。

  那么，流到底是什么呢？简短的定义就是"从支持数据处理操作的源生成的元素序列"。让我们一步步剖析这个定义。

  1、元素序列：就像集合一样，流也提供了一个接口，可以访问特定元素类型的一组有序值。因为集合是数据结构，所以它的主要目的是以特定的时间/空间复杂度存储和访问元素，但是流的目的在于表达计算，比如你前面简单的filter、sorted和map。集合讲的是数据，流讲的是计算。

 2、源：流会使用一个提供数据的源，如集合、数组或者输入/输出资源。请注意，从有序集合生成流时会保留原有的顺序。

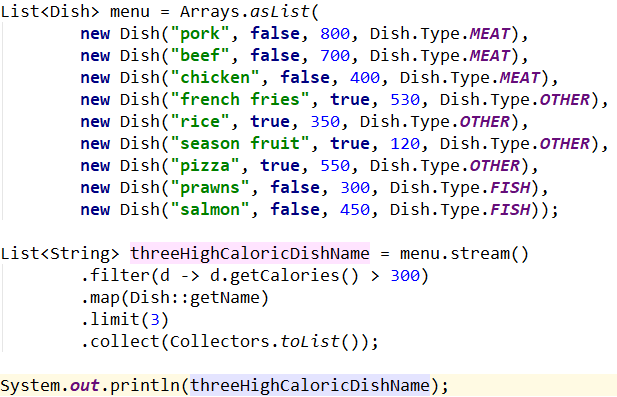
3、数据处理操作：流所支持的数据处理功能类似于数据库的操作，以及函数式编程语言中的常用操作，如filter、map、reduce、find、match、sort等等。流操作可以顺序执行，也可以并行执行。

此外，流操作有两个重要的特点。

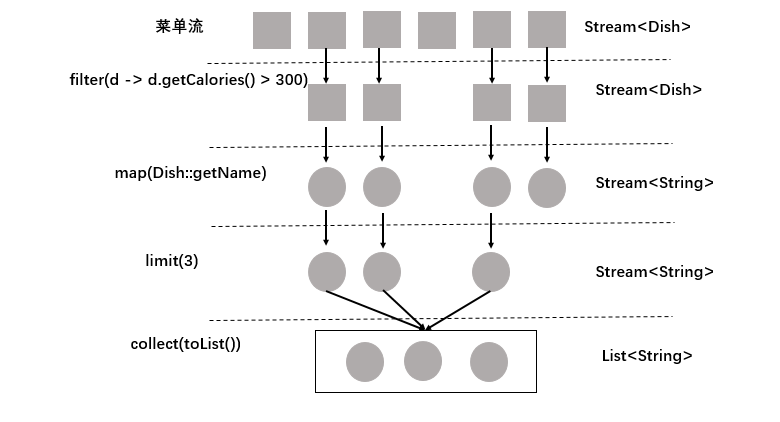
1、流水线：很多流操作本身会返回一个流，这样多个操作就可以链接起来，形成一个大的很大的流水线。这让我们以后的一些优化成为了可能，如延迟和短路。流水线的操作可以可以看做对数据源进行数据库式查询。

2、内部迭代：与使用迭代器进行显式迭代的集合不同，流的迭代操作是在背后进行的。

让我们来看一段能够体现所有这些概念的代码：



在本例中，我们先是对menu调用stream方法，由菜单得到一个流。数据源是菜肴列表(菜单)，它给流提供一个元素序列。接下来，对流应用一系列数据处理操作：filter、map、limit和collect。除了collect之外，所有其他操作都会返回另一个流，从而构成一条流水线。最后，collect操作开始处理流水线，并且返回结果(它和别的操作不一样，因为它返回的不是流，在这里是一个list)。在调用collect之前，没有任何结果产生，实际上根本就没有从menu里选择元素。你可以这么理解：链中的方法调用都在排队等待，直到调用collect。下图显示了流操作的顺序：filter、map、limit、collect。



filter：接收Lambda表达式，从流中排除某些元素。在本例中，通过传递Lambda表达式d -> d.getCalories() > 300，选择出热量超过300卡路里的菜肴。

map：接收Lambda表达式，将元素转换成其他形式或者提取信息。在本例中，通过传递方法引用Dish::getName提取到了每道菜的菜名。

limit：截断流，使其元素个数不超过给定的数量。

collect：将流转换为其他形式。在本例中，流被转换成为一个列表。

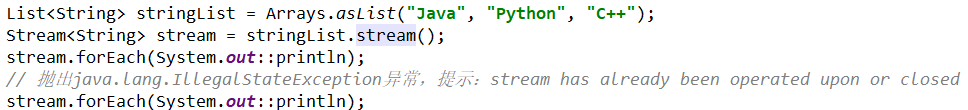
注意看，上面的这段代码与逐项处理菜单列表的代码有很大不同。首先，我们使用了声明的方式来处理菜单数据，即说明对这些数据要做什么，而并没有去实现筛选(filter)、提取(map)或者截断(limit)功能，而是交由Stream API来处理，由Stream API来决定如何优化这条流水线，例如筛选、提取和截断操作可以一次进行，并在找到三道菜后立即停止。

## 流与集合

粗略的说，集合与流之间的差异就在于什么时候进行计算。集合是内存中的数据结构，它是急切创建的，你可以往集合中添加或删除元素，但每个元素必须先计算出来才能添加到集合中或从集合中删除。流则是在概念上的数据结构，你不能往流中添加或删除元素，其元素是按照需要进行计算的。

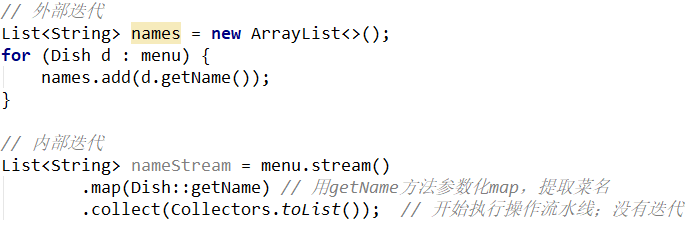
1、流只能遍历一次

注意，和迭代器类似，流只能遍历一次，遍历完成之后，这个流就被消费掉了。



2、流是内部迭代

使用Collection接口时，需要用户去迭代（比如for-each），这被称为外部迭代。而Stream使用内部迭代，即Stream帮你把迭代做了，并且把得到的流值存在了某个地方，你只需要说明要做什么就可以了。



内部迭代时，代码可以透明的并行处理，或者用更优化的顺序进行处理。如果用Java过去的外部迭代，则这些优化都是很困难的。Stream库的内部迭代可以自动选择一种适合你硬件的数据表示和并行实现。