# 使用Java8重构代码

## 改善代码的可读性

改善代码的可读性到底意味着什么？我们很难定义什么是好的可读性，因为这可能非常主观。通常的理解是，“别人理解这段代码的难易程度”。改善可读性意味着你要确保你的代码能非常容易地被包括自己在内的所有人理解和维护。为了确保你的代码能被其他人理解，有几个步骤可以尝试，比如确保你的代码附有良好的文档，并严格遵守编程规范。这里我们会介绍三种简单的重构，利用Lambda表达式、方法引用以及Stream改善程序代码的可读性：

* 重构代码，用Lambda表达式取代匿名类
* 用方法引用重构Lambda表达式
* 用Stream API重构命令式的数据处理

### 用Lambda取代匿名类

你值得尝试的第一种重构，也是简单的方式，是将实现单一抽象方法的匿名类转换为Lambda表达式。为什么呢？因为匿名类是极其繁琐且容易出错的。采用Lambda表达式之后，你的代码会更简洁，可读性更好。比如，创建Runnable 对象的匿名类，这段代码及其对应的Lambda表达式实现如下：

*// 传统方式的匿名类*Runnable **r1** = **new** Runnable(){  
 **public void** run(){  
 System.***out***.println(**"Hello"**);  
 }  
};  
*// Lambda表达式*Runnable **r2** = () -> System.***out***.println(**"Hello"**);

但是某些情况下，将匿名类转换为Lambda表达式可能是一个比较复杂的过程。 首先，匿名类和Lambda表达式中的 this 和 super 的含义是不同的。在匿名类中， this 代表的是类自身，但是在Lambda中，它代表的是包含类。其次，匿名类可以屏蔽包含类的变量，而Lambda表达式不能（它们会导致编译错误），譬如下面这段代码：

**int** a = 10;  
  
Runnable r1 = () -> {  
 **int** a = 2;*// 编译出错，提示变量a已经定义* System.***out***.println(a);  
};  
  
Runnable r2 = **new** Runnable(){  
 **public void** run(){  
 **int** a = 2;*// 编译通过* System.***out***.println(a);  
 }  
};

最后，在涉及重载的上下文里，将匿名类转换为Lambda表达式可能导致最终的代码更加晦涩。实际上，匿名类的类型是在初始化时确定的，而Lambda的类型取决于它的上下文。通过下面这个例子，我们可以了解问题是如何发生的。我们假设你用与 Runnable 同样的签名声明了一个函数接口，我们称之为 Task （你希望采用与你的业务模型更贴切的接口名时，就可能做这样的变更）：

**interface** Task{  
 **public void** execute();  
}  
**public static void** doSomething(Runnable r){ r.run(); }  
**public static void** doSomething(Task a){ a.execute(); }

现在，你再传递一个匿名类实现的 Task ，不会碰到任何问题：

*// 使用匿名类，编译正常  
doSomething*(**new** Task() {  
 **public void** execute() {  
 System.***out***.println(**"Danger danger!!"**);  
 }  
});

但是将这种匿名类转换为Lambda表达式时，就导致了一种晦涩的方法调用，因为 Runnable和 Task 都是合法的目标类型：

*// 编译出错：doSomething(Runnable) 和 doSomething(Task) 都匹配该类型*doSomething(() -> System.***out***.println(**"Danger danger!!"**));

你可以对 Task 尝试使用显式的类型转换来解决这种模棱两可的情况：

*// 正确写法  
doSomething*((Task)() -> System.***out***.println(**"Danger danger!!"**));

但是不要因此而放弃对Lambda的尝试。好消息是，目前大多数的集成开发环境，比如NetBeans和IntelliJ都支持这种重构，它们能自动地帮你检查，避免发生这些问题。

### **用方法引用重构Lambda**

Lambda表达式非常适用于需要传递代码片段的场景。不过，为了改善代码的可读性，也请尽量使用方法引用。因为方法名往往能更直观地表达代码的意图。比如下面这段代码，它的功能是按照食物的热量级别对菜肴进行分类：

Map<CaloricLevel, List<Dish>> dishesByCaloricLevel = menu.stream()  
 .collect(groupingBy(dish -> {  
 **if** (dish.getCalories() <= 400) **return** CaloricLevel.DIET;  
 **else if** (dish.getCalories() <= 700) **return** CaloricLevel.NORMAL;  
 **else return** CaloricLevel.FAT;  
 }));

你可以将Lambda表达式的内容抽取到一个单独的方法中，将其作为参数传递给 groupingBy方法。变换之后，代码变得更加简洁，程序的意图也更加清晰了：

Map<CaloricLevel, List<Dish>> dishesByCaloricLevel = menu.stream()  
 .collect(groupingBy(Dish::getCaloricLevel));

为了实现这个方案，你还需要在 Dish 类中添加 getCaloricLevel 方法：

**public** CaloricLevel getCaloricLevel(){  
 **if** (**this**.getCalories() <= 400) **return** CaloricLevel.DIET;  
 **else if** (**this**.getCalories() <= 700) **return** CaloricLevel.NORMAL;  
 **else return** CaloricLevel.FAT;  
}

除此之外，我们还应该尽量考虑使用静态辅助方法，比如 comparing 、maxBy 。这些方法设计之初就考虑了会结合方法引用一起使用。通过示例，我们看到优化过的代码更清晰地表达了它的设计意图：

inventory.sort(  
 (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())  
);  
inventory.sort(comparing(Apple::getWeight));

此外，很多通用的归约操作，比如 sum 、 maximum ，都有内建的辅助方法可以和方法引用结合使用。比如，在我们的示例代码中，使用 Collectors 接口可以轻松得到和或者最大值，与采用Lambada表达式和底层的归约操作比起来，这种方式要直观得多。与其编写：

**int** totalCalories = menu.stream()  
 .map(Dish::getCalories)  
 .reduce(0, (c1, c2) -> c1 + c2);

不如尝试使用内置的集合类，它能更清晰地表达问题陈述是什么。下面的代码中，我们使用了集合类 summingInt （方法的名词很直观地解释了它的功能）：

**int** totalCalories = menu.stream()  
 .collect(summingInt(Dish::getCalories));

### **使用Stream 重构数据处理**

我们建议你将所有使用迭代器这种数据处理模式处理集合的代码都转换成Stream API的方式。为什么呢？Stream API能更清晰地表达数据处理管道的意图。除此之外，通过短路和延迟载入以及利用现代计算机的多核架构，我们可以对Stream进行优化。比如，下面的命令式代码使用了两种模式：筛选和抽取，这两种模式被混在了一起，这样的代码结构迫使程序员必须彻底搞清楚程序的每个细节才能理解代码的功能。此外，实现需要并行运行的程序所面对的困难也多得多（具体细节可以参考7.2节的分支/合并框架）：

List<String> dishNames = **new** ArrayList<>();  
**for**(Dish dish: menu){  
 **if**(dish.getCalories() > 300){  
 dishNames.add(dish.getName());  
 }  
}

替代方案使用Stream API，采用这种方式编写的代码读起来更像是问题陈述，并行化也非常容易：

menu.parallelStream()  
 .filter(d -> d.getCalories() > 300)  
 .map(Dish::getName)  
 .collect(toList());

不幸的是，将命令式的代码结构转换为Stream API的形式是个困难的任务，因为你需要考虑控制流语句，比如 break 、 continue 、 return ，并选择使用恰当的流操作。好消息是已经有一些工具可以帮助我们完成这个任务。

## 增加代码的灵活性

Lambda表达式有利于行为参数化。你可以使用不同的Lambda表示不同的行为，并将它们作为参数传递给函数去处理执行。这种方式可以帮助我们淡

定从容地面对需求的变化。比如，我们可以用多种方式为 Predicate 创建筛选条件，或者使用Comparator 对多种对象进行比较。现在，我们来看看哪些模式可以马上应用到你的代码中，让你享受Lambda表达式带来的便利。

首先，你必须意识到，没有函数接口，你就无法使用Lambda表达式。因此，你需要在代码中引入函数接口。听起来很合理，但是在什么情况下使用它们呢？这里我们介绍两种通用的模式，你可以依照这两种模式重构代码，利用Lambda表达式带来的灵活性，它们分别是：有条件的延迟执行和环绕执行。除此之外，在下一节，我们还将介绍一些基于面向对象的设计模式，比如策略模式或者模板方法，这些在使用Lambda表达式重写后会更简洁。

### **行为参数化：延迟执行**

使用lambda表达式除了可以改善代码的可读性外，还有另一个很重要的原因是，将代码的执行延迟到一个合适的时间点。在使用lambda表达式的时候务必记住一点就是说lambda表达式都是延迟执行的。延迟执行代码的原因有很多，比如：  
1，在另外一个线程中运行代码  
2，多次运行代码  
3，在某个算法的正确时间点上运行代码，比如排序中的比较操作  
4，当某些情况发生时才运行代码  
5，只有在需要的时候才运行代码。  
  
 这里来举一个日志的例子吧。我们在使用logger.info输出日志，代码如下：log.debug("x的长度是：" + x.length());代码是没问题，但是这里会有一个性能上的问题来，不管上面的代码是否被真的执行，这个x的字符串都会被计算然后传递给info方法，那么为什么我们不能再确定需要打印的时候，再去计算这个字符串呢？很多时候我们经常看到这样的代码：

**if** (**log**.isDebugEnabled()) {  
 **log**.debug(**"x的长度是："** + x.length());  
}

这段代码有什么问题吗？

* 首先，日志器的状态通过 isDebugEnabled方法暴露给了客户端代码。
* 其次，每次输出一条日志之前都要在添加一行代码去查询日志器对象的状态。

那怎么改进一下好呢？我们可以将上面的这段代码包装成一个无参数的lambda的表达式，只有在需要的时候才能运行代码，这也是使用lambda表达式的一种情况。以下是包装过后的代码：

**protected final** Log **log** = LogFactory.getLog(getClass());  
**public static void** delayDebug(Supplier<String> message) {  
 **if** (**log**.isDebugEnabled()) {  
 **log**.debug(message.get());  
 }  
}  
  
**public static void** main(String[] args) {  
 *delayDebug*(() -> **"x的长度是："**+x.length());  
}

这段编码实际的运行效果分如下步骤：  
1，接受lambda表达式  
2，检查是否需要打印日志  
3，在需要的时候再去执行x.length()

### **行为参数化：环绕执行**

如果你发现虽然你的业务代码千差万别，如下面示例的红色代码，但是它们拥有同样的准备和清理阶段的逻辑代码，我们将这部分称为模板代码，如加粗的代码，这时，你完全可以用Lambda实现。这种方式的好处是可以重用准备和清理阶段的逻辑代码，减少重复冗余的代码。它在打开和关闭文件时使用了同样的逻辑，但在处理文件时可以使用不同的Lambda进行参数化。

*// 使用Lambda表达式的函数接口，该接口能够抛出一个 IOException***public interface** BufferedReaderProcessor{  
 String process(BufferedReader b) **throws** IOException;  
}  
  
**public static** String processFile(BufferedReaderProcessor p) **throws** IOException {  
 **try**(**BufferedReader br = new BufferedReader(new FileReader("test/data.txt"))**){  
 **return** p.process(br);*//将 BufferedReaderProcessor 作为执行参数传入* }  
}  
  
*// 读取一行*String **oneLine** = *processFile*((BufferedReader b) -> b.readLine());  
*// 读取两行*String **twoLines** = *processFile*((BufferedReader b) -> b.readLine() + b.readLine());

这一优化是凭借函数式接口 BufferedReaderProcessor 达成的，通过这个接口，你可以传递各种Lamba表达式对 BufferedReader 对象进行处理。

## 使用Lambda重构传统的面向对象模式设计