链滴

《Java8 实战》- 第八章笔记（重构、测试和调试）

作者：Not-Found
原文链接：https://ld246.com/article/1541259925404
来源网站：链滴

重构、测试和调试

通过本书的前七章，我们了解了Lambda和Stream API的强大威力。你可能主要在新项目的代码中使这些特性。如果你创建的是全新的Java项目，这是极好的时机，你可以轻装上阵，迅速地将新特性应到项目中。然而不幸的是，大多数情况下你没有机会从头开始一个全新的项目。很多时候，你不得不对的是用老版Java接口编写的遗留代码。

这些就是本章要讨论的内容。我们会介绍几种方法，帮助你重构代码，以适配使用Lambda表达式，你维护的代码具备更好的可读性和灵活性。除此之外，我们还会讨论目前比较流行的几种面向对象的计模式，包括策略模式、模板方法模式、观察者模式、责任链模式，以及工厂模式，在结合Lambda达式之后变得更简洁的情况。最后，我们会介绍如何测试和调试使用Lambda表达式和Stream API的码。

为改善可读性和灵活性重构代码

从本书的开篇我们就一直在强调，利用Lambda表达式，你可以写出更简洁、更灵活的代码。用“更洁”来描述Lambda表达式是因为相较于匿名类，Lambda表达式可以帮助我们用更紧凑的方式描述序的行为。第3章中我们也提到，如果你希望将一个既有的方法作为参数传递给另一个方法，那么方引用无疑是我们推荐的方法，利用这种方式我们能写出非常简洁的代码。

采用Lambda表达式之后，你的代码会变得更加灵活，因为Lambda表达式鼓励大家使用第2章中介绍的行为参数化的方式。在这种方式下，应对需求的变化时，你的代码可以依据传入的参数动态选择和行相应的行为。

这一节，我们会将所有这些综合在一起，通过例子展示如何运用前几章介绍的Lambda表达式、方法用以及Stream接口等特性重构遗留代码，改善程序的可读性和灵活性。

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，比如确保你代码附有良好的文档，并严格遵守编程规范。

跟之前的版本相比较，Java 8的新特性也可以帮助提升代码的可读性：

- 使用Java 8，你可以减少冗长的代码，让代码更易于理解
- 通过方法引用和Stream API，你的代码会变得更直观

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

从匿名类到 Lambda 表达式的转换

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为Lambda表达式。什么呢？前面几章的介绍应该足以说服你，因为匿名类是极其繁琐且容易出错的。采用Lambda表达之后，你的代码会更简洁，可读性更好。还记得第3章的例子就是一个创建Runnable 对象的匿名类

这段代码及其对应的Lambda表达式实现如下：

```
Runnable r1 = new Runnable(){
    public void run() {
        System.out.println("Hello");
    }
};
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，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（它们会导致编错误），譬如下面这段代码：

```
int a = 10;
Runnable r1 = () -> {
    // 编译错误
    int a = 2;
    System.out.println(a);
};

Runnable r2 = new Runnable(){
    public void run(){
        // 一切正常
        int a = 2;
        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，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 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：

```
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：

```
// 麻烦来了： 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：

```
doSomething((Task)() -> System.out.println("Danger danger!!"));
```

但是不要因此而放弃对Lambda的尝试。

从 Lambda 表达式到方法引用的转换

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，也请尽量使用法引用。因为方法名往往能更直观地表达代码的意图。比如，第6章中我们曾经展示过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<Dish.CaloricLevel, List<Dish>> dishesByCaloricLevel = menu.stream().collect(
        groupingBy(dish -> {
            if (dish.getCalories() <= 400) {
                return Dish.CaloricLevel.DIET;
            } else if (dish.getCalories() <= 700) {
                return Dish.CaloricLevel.NORMAL;
            } else {
                return Dish.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。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(Dis
::getCaloricLevel));
```

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

```
public class Dish{
    ...
    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 。这些方法设计之就考虑了会结合方法引用一起使用。通过示例，我们看到相对于第3章中的对应代码，优化过的代码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。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ummingInt （方法的名词很直观地解释了它的功能）：

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

从命令式的数据处理切换到 Stream

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？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第7章绍的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的困难也多得多：

```
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，比如 break 、 continue 、 return ，并选择使用恰当的流操作。

增加代码的灵活性

第2章和第3章中，我们曾经介绍过Lambda表达式有利于行为参数化。你可以使用不同的Lambda表不同的行为，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。

- 采用函数接口

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。

- 有条件的延迟执行

我们经常看到这样的代码，控制语句被混杂在业务逻辑代码之中。典型的情况包括进行安全性检查以日志输出。比如，下面的这段代码，它使用了Java语言内置的 Logger 类：

```
if (logger.isLoggable(Log.FINER)){
```

```
    logger.finer("Problem: " + generateDiagnostic());
}
```

这段代码有什么问题吗？其实问题不少。

-
  - 日志器的状态（它支持哪些日志等级）通过 isLoggable 方法暴露给了客户端代码。
-
  - 为什么要在每次输出一条日志之前都去查询日志器对象的状态？这只能搞砸你的代码。更好的案是使用 log 方法，该方法在输出日志消息之前，会在内部检查日志对象是否已经设置为恰当的日志级：

```
logger.log(Level.FINER, "Problem: " + generateDiagnostic());
```

这种方式更好的原因是你不再需要在代码中插入那些条件判断，与此同时日志器的状态也不再被暴露去。不过，这段代码依旧存在一个问题。日志消息的输出与否每次都需要判断，即使你已经传递了参，不开启日志。

这就是Lambda表达式可以施展拳脚的地方。你需要做的仅仅是延迟消息构造，如此一来，日志就只在某些特定的情况下才开启（以此为例，当日志器的级别设置为 FINER 时）。显然，Java 8的API设者们已经意识到这个问题，并由此引入了一个对 log 方法的重载版本，这个版本的 log 方法接受一个upplier 作为参数。这个替代版本的 log 方法的函数签名如下：

```
public void log(Level level, Supplier<String> msgSupplier)
```

你可以通过下面的方式对它进行调用：

```
logger.log(Level.FINER, () -> "Problem: " + generateDiagnostic());
```

如果日志器的级别设置恰当， log 方法会在内部执行作为参数传递进来的Lambda表达式。这里介绍的Log 方法的内部实现如下：

```
public void log(Level level, Supplier<String> msgSupplier) {
    if(logger.isLoggable(level)){
        log(level, msgSupplier.get());
    }
}
```

从这个故事里我们学到了什么呢？如果你发现你需要频繁地从客户端代码去查询一个对象的状态（比前文例子中的日志器的状态），只是为了传递参数、调用该对象的一个方法（比如输出一条日志），么可以考虑实现一个新的方法，以Lambda或者方法表达式作为参数，新方法在检查完该对象的状态后才调用原来的方法。你的代码会因此而变得更易读（结构更清晰），封装性更好（对象的状态也不暴露给客户端代码了）。

通过这一节，你已经了解了如何通过不同方式来改善代码的可读性和灵活性。接下来，你会了解Lambda表达式如何避免常规面向对象设计中的僵化的模板代码。

使用 Lambda 重构面向对象的设计模式

新的语言特性常常让现存的编程模式或设计黯然失色。比如， Java 5中引入了 foreach 循环，由于的稳健性和简洁性，已经替代了很多显式使用迭代器的情形。Java 7中推出的菱形操作符（ <> ）让家在创建实例时无需显式使用泛型，一定程度上推动了Java程序员们采用类型接口（type interface

进行程序设计。

对设计经验的归纳总结被称为设计模式。设计软件时，如果你愿意，可以复用这些方式方法来解决一常见问题。这看起来像传统建筑工程师的工作方式，对典型的场景（比如悬挂桥、拱桥等）都定义有重用的解决方案。例如，访问者模式常用于分离程序的算法和它的操作对象。单例模式一般用于限制的实例化，仅生成一份对象

Lambda表达式为程序员的工具箱又新添了一件利器。它们为解决传统设计模式所面对的问题提供了的解决方案，不但如此，采用这些方案往往更高效、更简单。使用Lambda表达式后，很多现存的略臃肿的面向对象设计模式能够用更精简的方式实现了。这一节中，我们会针对五个设计模式展开讨论它们分别是：

- 策略模式
- 模板方法
- 观察者模式
- 责任链模式
- 工厂模式

我们会展示Lambda表达式是如何另辟蹊径解决设计模式原来试图解决的问题的。

策略模式

策略模式代表了解决一类算法的通用解决方案，你可以在运行时选择使用哪种方案。在第2章中你已简略地了解过这种模式了，当时我们介绍了如何使用不同的条件（比如苹果的重量，或者颜色）来筛库存中的苹果。你可以将这一模式应用到更广泛的领域，比如使用不同的标准来验证输入的有效性，用不同的方式来分析或者格式化输入。

策略模式包含三部分内容：

- 一个代表某个算法的接口（它是策略模式的接口）。
- 一个或多个该接口的具体实现，它们代表了算法的多种实现（比如，实体类 ConcreteStrategyA 者 ConcreteStrategyB ）。
- 一个或多个使用策略对象的客户。

我们假设你希望验证输入的内容是否根据标准进行了恰当的格式化（比如只包含小写字母或数字）。可以从定义一个验证文本（以 String 的形式表示）的接口入手：

```
interface ValidationStrategy {
    boolean execute(String s);
}
```

其次，你定义了该接口的一个或多个具体实现：

```
static class IsAllLowerCase implements ValidationStrategy {

    @Override
    public boolean execute(String s) {
        return s.matches("[a-z]+");
    }
}
```

```
static class IsNumeric implements ValidationStrategy {

    @Override
    public boolean execute(String s) {
        return s.matches("\\d+");
    }
}
```

之后，你就可以在你的程序中使用这些略有差异的验证策略了：

```
private static class Validator {
    private final ValidationStrategy validationStrategy;

    public Validator(ValidationStrategy validationStrategy) {
        this.validationStrategy = validationStrategy;
    }

    public boolean validate(String s) {
        return validationStrategy.execute(s);
    }
}

Validator v1 = new Validator(new IsNumeric());
// false
System.out.println(v1.validate("aaaa"));
Validator v2 = new Validator(new IsAllLowerCase());
// true
System.out.println(v2.validate("bbbb"));
```

使用Lambda表达式

到现在为止，你应该已经意识到 ValidationStrategy 是一个函数接口了（除此之外，它还与 Predicate String> 具有同样的函数描述）。这意味着我们不需要声明新的类来实现不同的策略，通过直接传递Lmbda表达式就能达到同样的目的，并且还更简洁：

```
Validator v3 = new Validator((String s) -> s.matches("\\d+"));
System.out.println(v3.validate("aaaa"));
Validator v4 = new Validator((String s) -> s.matches("[a-z]+"));
System.out.println(v4.validate("bbbb"));
```

正如你看到的，Lambda表达式避免了采用策略设计模式时僵化的模板代码。如果你仔细分析一下个缘由，可能会发现，Lambda表达式实际已经对部分代码（或策略）进行了封装，而这就是创建策略计模式的初衷。因此，我们强烈建议对类似的问题，你应该尽量使用Lambda表达式来解决。

模板方法

如果你需要采用某个算法的框架，同时又希望有一定的灵活度，能对它的某些部分进行改进，那么采模板方法设计模式是比较通用的方案。好吧，这样讲听起来有些抽象。换句话说，模板方法模式在你希望使用这个算法，但是需要对其中的某些行进行改进，才能达到希望的效果”时是非常有用的。

让我们从一个例子着手，看看这个模式是如何工作的。假设你需要编写一个简单的在线银行应用。通，用户需要输入一个用户账户，之后应用才能从银行的数据库中得到用户的详细信息，最终完成一些用户满意的操作。不同分行的在线银行应用让客户满意的方式可能还略有不同，比如给客户的账户发红利，或者仅仅是少发送一些推广文件。你可能通过下面的抽象类方式来实现在线银行应用：

```
public abstract class AbstractOnlineBanking {
    public void processCustomer(int id) {
        Customer customer = Database.getCustomerWithId(id);
        makeCustomerHappy(customer);
    }

    /**
     * 让客户满意
     *
     * @param customer
     */
    abstract void makeCustomerHappy(Customer customer);

    private static class Customer {}

    private static class Database {
        static Customer getCustomerWithId(int id) {
            return new Customer();
        }
    }
}
```

processCustomer 方法搭建了在线银行算法的框架：获取客户提供的ID，然后提供服务让用户满意不同的支行可以通过继承 AbstractOnlineBanking 类，对该方法提供差异化的实现。

使用Lambda表达式

使用你偏爱的Lambda表达式同样也可以解决这些问题（创建算法框架，让具体的实现插入某些部分。你想要插入的不同算法组件可以通过Lambda表达式或者方法引用的方式实现。

这里我们向 processCustomer 方法引入了第二个参数，它是一个 Consumer<Customer> 类型的数，与前文定义的 makeCustomerHappy 的特征保持一致：

```
public void processCustomer(int id, Consumer<Customer> makeCustomerHappy) {
    Customer customer = Database.getCustomerWithId(id);
    makeCustomerHappy.accept(customer);
}
```

现在，你可以很方便地通过传递Lambda表达式，直接插入不同的行为，不再需要继承AbstractOnlin Banking 类了：

```
public static void main(String[] args) {
        new AbstractOnlineBankingLambda().processCustomer(1337, (
            AbstractOnlineBankingLambda.Customer c) -> System.out.println("Hello!"));
    }
```

这是又一个例子，佐证了Lamba表达式能帮助你解决设计模式与生俱来的设计僵化问题。

观察者模式

观察者模式是一种比较常见的方案，某些事件发生时（比如状态转变），如果一个对象（通常我们称为主题）需要自动地通知其他多个对象（称为观察者），就会采用该方案。创建图形用户界面（GUI程序时，你经常会使用该设计模式。这种情况下，你会在图形用户界面组件（比如按钮）上注册一系的观察者。如果点击按钮，观察者就会收到通知，并随即执行某个特定的行为。 但是观察者模式并不限于图形用户界面。比如，观察者设计模式也适用于股票交易的情形，多个券商可能都希望对某一支

票价格（主题）的变动做出响应。

让我们写点儿代码来看看观察者模式在实际中多么有用。你需要为Twitter这样的应用设计并实现一个制化的通知系统。想法很简单：好几家报纸机构，比如《纽约时报》《卫报》以及《世界报》都订阅新闻，他们希望当接收的新闻中包含他们感兴趣的关键字时，能得到特别通知。

首先，你需要一个观察者接口，它将不同的观察者聚合在一起。它仅有一个名为 notify 的方法，一旦收到一条新的新闻，该方法就会被调用：

```
interface Observer{
    void inform(String tweet);
}
```

现在，你可以声明不同的观察者（比如，这里是三家不同的报纸机构），依据新闻中不同的关键字分定义不同的行为：

```
private static class NYTimes implements Observer {

    @Override
    public void inform(String tweet) {
        if (tweet != null && tweet.contains("money")) {
            System.out.println("Breaking news in NY!" + tweet);
        }
    }
}

private static class Guardian implements Observer {

    @Override
    public void inform(String tweet) {
        if (tweet != null && tweet.contains("queen")) {
            System.out.println("Yet another news in London... " + tweet);
        }
    }
}

private static class LeMonde implements Observer {

    @Override
    public void inform(String tweet) {
        if(tweet != null && tweet.contains("wine")){
            System.out.println("Today cheese, wine and news! " + tweet);
        }
    }
}
```

你还遗漏了最重要的部分： Subject！让我们为它定义一个接口：

```
interface Subject {
    void registerObserver(Observer o);

    void notifyObserver(String tweet);
}
```

Subject 使用 registerObserver 方法可以注册一个新的观察者，使用 notifyObservers方法通知它的

察者一个新闻的到来。让我们更进一步，实现 Feed 类：

```
private static class Feed implements Subject {
    private final List<Observer> observers = new ArrayList<>();

    @Override
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void notifyObserver(String tweet) {
        observers.forEach(o -> o.inform(tweet));
    }
}
```

这是一个非常直观的实现： Feed 类在内部维护了一个观察者列表，一条新闻到达时，它就进行通知。

毫不意外，《卫报》会特别关注这条新闻！

使用Lambda表达式

你可能会疑惑Lambda表达式在观察者设计模式中如何发挥它的作用。不知道你有没有注意到， Observer 接口的所有实现类都提供了一个方法： inform 。新闻到达时，它们都只是对同一段代码封装执。Lambda表达式的设计初衷就是要消除这样的僵化代码。使用Lambda表达式后，你无需显式地实化三个观察者对象，直接传递Lambda表达式表示需要执行的行为即可：

```
Feed feedLambda = new Feed();
feedLambda.registerObserver((String tweet) -> {
    if (tweet != null && tweet.contains("money")) {
        System.out.println("Breaking news in NY!" + tweet);
    }
});

feedLambda.registerObserver((String tweet) -> {
    if (tweet != null && tweet.contains("queen")) {
        System.out.println("Yet another news in London... " + tweet);
    }
});

feedLambda.notifyObserver("Money money money, give me money!");
```

那么，是否我们随时随地都可以使用Lambda表达式呢？答案是否定的！我们前文介绍的例子中，Labda适配得很好，那是因为需要执行的动作都很简单，因此才能很方便地消除僵化代码。但是，观察的逻辑有可能十分复杂，它们可能还持有状态，抑或定义了多个方法，诸如此类。在这些情形下，你是应该继续使用类的方式。

责任链模式

责任链模式是一种创建处理对象序列（比如操作序列）的通用方案。一个处理对象可能需要在完成一工作之后，将结果传递给另一个对象，这个对象接着做一些工作，再转交给下一个处理对象，以此类。

通常，这种模式是通过定义一个代表处理对象的抽象类来实现的，在抽象类中会定义一个字段来记录续对象。一旦对象完成它的工作，处理对象就会将它的工作转交给它的后继。代码中，这段逻辑看起

是下面这样：

```
private static abstract class AbstractProcessingObject<T> {
    protected AbstractProcessingObject<T> successor;

    public void setSuccessor(AbstractProcessingObject<T> successor) {
        this.successor = successor;
    }

    public T handle(T input) {
        T r = handleWork(input);
        if (successor != null) {
            return successor.handle(r);
        }
        return r;
    }

    protected abstract T handleWork(T input);
}
```

下面让我们看看如何使用该设计模式。你可以创建两个处理对象，它们的功能是进行一些文本处理工
。

```
private static class HeaderTextProcessing extends AbstractProcessingObject<String> {

    @Override
    protected String handleWork(String text) {
        return "From Raoul, Mario and Alan: " + text;
    }
}

private static class SpellCheckerProcessing extends AbstractProcessingObject<String> {

    @Override
    protected String handleWork(String text) {
        return text.replaceAll("labda", "lambda");
    }
}
```

现在你就可以将这两个处理对象结合起来，构造一个操作序列！

```
AbstractProcessingObject<String> p1 = new HeaderTextProcessing();
AbstractProcessingObject<String> p2 = new SpellCheckerProcessing();
p1.setSuccessor(p2);
String result = p1.handle("Aren't labdas really sexy?!!");
System.out.println(result);
```

使用Lambda表达式

稍等！这个模式看起来像是在链接（也即是构造） 函数。第3章中我们探讨过如何构造Lambda表达
。你可以将处理对象作为函数的一个实例，或者更确切地说作为 UnaryOperator<String> 的一个实
。为了链接这些函数，你需要使用 andThen 方法对其进行构造。

```
UnaryOperator<String> headerProcessing =
        (String text) -> "From Raoul, Mario and Alan: " + text;
```

```
UnaryOperator<String> spellCheckerProcessing =
        (String text) -> text.replaceAll("labda", "lambda");

Function<String, String> pipeline = headerProcessing.andThen(spellCheckerProcessing);

String result2 = pipeline.apply("Aren't labdas really sexy?!!");
System.out.println(result2);
```

工厂模式

使用工厂模式，你无需向客户暴露实例化的逻辑就能完成对象的创建。比如，我们假定你为一家银行作，他们需要一种方式创建不同的金融产品：贷款、期权、股票，等等。

通常，你会创建一个工厂类，它包含一个负责实现不同对象的方法，如下所示：

```
private interface Product {
}

private static class ProductFactory {
    public static Product createProduct(String name) {
        switch (name) {
            case "loan":
                return new Loan();
            case "stock":
                return new Stock();
            case "bond":
                return new Bond();
            default:
                throw new RuntimeException("No such product " + name);
        }
    }
}

static private class Loan implements Product {
}

static private class Stock implements Product {
}

static private class Bond implements Product {
}
```

这里贷款（Loan）、股票（Stock）和债券（Bond）都是产品（Product）的子类。createProduct 方法可以通过附加的逻辑来设置每个创建的产品。但是带来的好处也显而易见，你在创建对象时用再担心会将构造函数或者配置暴露给客户，这使得客户创建产品时更加简单：

```
Product p1 = ProductFactory.createProduct("loan");
```

使用Lambda表达式

第3章中，我们已经知道可以像引用方法一样引用构造函数。比如，下面就是一个引用贷款（Loan 构造函数的示例：

```
Supplier<Product> loanSupplier = Loan::new;
Product p2 = loanSupplier.get();
```

通过这种方式，你可以重构之前的代码，创建一个 Map ，将产品名映射到对应的构造函数：

```
final static private Map<String, Supplier<Product>> map = new HashMap<>();

static {
    map.put("loan", Loan::new);
    map.put("stock", Stock::new);
    map.put("bond", Bond::new);
}
```

现在，你可以像之前使用工厂设计模式那样，利用这个 Map 来实例化不同的产品。

```
public static Product createProductLambda(String name) {
    Supplier<Product> p = map.get(name);
    if (p != null) {
        return p.get();
    }
    throw new RuntimeException("No such product " + name);
}
```

这是个全新的尝试，它使用Java 8中的新特性达到了传统工厂模式同样的效果。但是，如果工厂方法 c eateProduct 需要接收多个传递给产品构造方法的参数，这种方式的扩展性不是很好。你不得不提供同的函数接口，无法采用之前统一使用一个简单接口的方式。

比如，我们假设你希望保存具有三个参数（两个参数为 Integer 类型，一个参数为 String类型）的构函数；为了完成这个任务，你需要创建一个特殊的函数接口 TriFunction 。最终的结果是 Map 变得加复杂。

```
public interface TriFunction<T, U, V, R>{
    R apply(T t, U u, V v);
}
Map<String, TriFunction<Integer, Integer, String, Product>> map = new HashMap<>();
```

你已经了解了如何使用Lambda表达式编写和重构代码。接下来，我们会介绍如何确保新编写代码的正确性。

测试 Lambda 表达式

现在你的代码中已经充溢着Lambda表达式，看起来不错，也很简洁。但是，大多数时候，我们受雇行的程序开发工作的要求并不是编写优美的代码，而是编写正确的代码。

通常而言，好的软件工程实践一定少不了单元测试，借此保证程序的行为与预期一致。你编写测试用，通过这些测试用例确保你代码中的每个组成部分都实现预期的结果。比如，图形应用的一个简单的oint 类，可以定义如下：

```
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
```

```
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public Point moveRightBy(int x) {
        return new Point(this.x + x, this.y);
    }
}
```

下面的单元测试会检查 moveRightBy 方法的行为是否与预期一致：

```
public class PointTest {

    @Test
    public void testMoveRightBy() {
        Point p1 = new Point(5, 5);
        Point p2 = p1.moveRightBy(10);

        Assert.assertEquals(15, p2.getX());
        Assert.assertEquals(5, p2.getY());
    }
}
```

测试可见 Lambda 函数的行为

由于 moveRightBy 方法声明为public，测试工作变得相对容易。你可以在用例内部完成测试。但是Lambda并无函数名（毕竟它们都是匿名函数），因此要对你代码中的Lambda函数进行测试实际上比困难，因为你无法通过函数名的方式调用它们。

有些时候，你可以借助某个字段访问Lambda函数，这种情况，你可以利用这些字段，通过它们对封在Lambda函数内的逻辑进行测试。比如，我们假设你在 Point 类中添加了静态字段compareByXAnThenY，通过该字段，使用方法引用你可以访问 Comparator 对象：

```
public class Point {
    public final static Comparator<Point> COMPARE_BY_X_AND_THEN_Y =
            comparing(Point::getX).thenComparing(Point::getY);
    ...
}
```

还记得吗，Lambda表达式会生成函数接口的一个实例。由此，你可以测试该实例的行为。这个例子，我们可以使用不同的参数，对 Comparator 对象类型实例 compareByXAndThenY的 compare 法进行调用，验证它们的行为是否符合预期：

```
@Test
public void testComparingTwoPoints() {
    Point p1 = new Point(10, 15);
    Point p2 = new Point(10, 20);
```

```
    int result = Point.COMPARE_BY_X_AND_THEN_Y.compare(p1 , p2);
    Assert.assertEquals(-1, result);
}
```

测试使用 Lambda 的方法的行为

但是Lambda的初衷是将一部分逻辑封装起来给另一个方法使用。从这个角度出发，你不应该将Lamba表达式声明为public，它们仅是具体的实现细节。相反，我们需要对使用Lambda表达式的方法进行试。比如下面这个方法 moveAllPointsRightBy：

```
public static List<Point> moveAllPointsRightBy(List<Point> points, int x) {
    return points.stream()
            .map(p -> new Point(p.getX() + x, p.getY()))
            .collect(toList());
}
```

我们没必要对Lambda表达式 p -> new Point(p.getX() + x,p.getY()) 进行测试，它只是 moveAllPoitsRightBy 内部的实现细节。我们更应该关注的是方法 moveAllPointsRightBy 的行为：

```
@Test
public void testMoveAllPointsRightBy() {
    List<Point> points =
            Arrays.asList(new Point(5, 5), new Point(10, 5));
    List<Point> expectedPoints =
            Arrays.asList(new Point(15, 5), new Point(20, 5));
    List<Point> newPoints = Point.moveAllPointsRightBy(points, 10);
    Assert.assertEquals(expectedPoints, newPoints);
}
```

注意，上面的单元测试中， Point 类恰当地实现 equals 方法非常重要，否则该测试的结果就取决于bject 类的默认实现。

调试

调试有问题的代码时，程序员的兵器库里有两大老式武器，分别是：

- 查看栈跟踪
- 输出日志

查看栈跟踪

你的程序突然停止运行（比如突然抛出一个异常），这时你首先要调查程序在什么地方发生了异常以为什么会发生该异常。这时栈帧就非常有用。程序的每次方法调用都会产生相应的调用信息，包括程中方法调用的位置、该方法调用使用的参数、被调用方法的本地变量。这些信息被保存在栈帧上。

程序失败时，你会得到它的栈跟踪，通过一个又一个栈帧，你可以了解程序失败时的概略信息。换句说，通过这些你能得到程序失败时的方法调用列表。这些方法调用列表最终会帮助你发现问题出现的因。

Lambda表达式和栈跟踪

不幸的是，由于Lambda表达式没有名字，它的栈跟踪可能很难分析。在下面这段简单的代码中，我刻意地引入了一些错误：

```
public class Debugging {
    public static void main(String[] args) {
        List<Point> points = Arrays.asList(new Point(12, 2), null);
        points.stream().map(p -> p.getX()).forEach(System.out::println);
    }
}
```

运行这段代码会产生下面的栈跟踪：

```
12
Exception in thread "main" java.lang.NullPointerException
    // 这行中的 $0 是什么意思?
    at xin.codedream.java8.chap8.Debugging.lambda$main$0(Debugging.java:15)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline.java:193)
    at java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators.java:948)
    ...
```

这个程序出现了NPE（空指针异常）异常，因为 Points 列表的第二个元素是空（ null ）。

这时你的程序实际是在试图处理一个空引用。由于Stream流水线发生了错误，构成Stream流水线的个方法调用序列都暴露在你面前了。不过，你留意到了吗？栈跟踪中还包含下面这样类似加密的内容：

```
at xin.codedream.java8.chap8.Debugging.lambda$main$0(Debugging.java:15)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline.java:193)
```

这些表示错误发生在Lambda表达式内部。由于Lambda表达式没有名字，所以编译器只能为它们指一个名字。这个例子中，它的名字是 lambda$main$0 ，看起来非常不直观。如果你使用了大量的类其中又包含多个Lambda表达式，这就成了一个非常头痛的问题。

即使你使用了方法引用，还是有可能出现栈无法显示你使用的方法名的情况。将之前的Lambda表达式 p-> p.getX() 替换为方法引用 reference Point::getX 也会产生难于分析的栈跟踪：

```
at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline.java:193)
```

注意，如果方法引用指向的是同一个类中声明的方法，那么它的名称是可以在栈跟踪中显示的。比如下面这个例子：

```
public class Debugging {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3);
        numbers.stream().map(Debugging::divideByZero).forEach(System
                .out::println);
    }

    public static int divideByZero(int n) {
        return n / 0;
    }
}
```

方法 divideByZero 在栈跟踪中就正确地显示了：

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    // divideByZero正确地输出到栈跟踪中
    at xin.codedream.java8.chap8.Debugging.divideByZero(Debugging.java:20)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline.java:193)
```

```
...
```

总的来说，我们需要特别注意，涉及Lambda表达式的栈跟踪可能非常难理解。这是Java编译器未来本可以改进的一个方面。

使用日志调试

假设你试图对流操作中的流水线进行调试，该从何入手呢？你可以像下面的例子那样，使用forEach流操作的结果日志输出到屏幕上或者记录到日志文件中：

```
List<Integer> numbers = Arrays.asList(2, 3, 4, 5);
numbers.stream()
        .map(x -> x + 17)
        .filter(x -> x % 2 == 0)
        .limit(3)
        .forEach(System.out::println);
```

这段代码的输出如下：

```
20
22
```

不幸的是，一旦调用 forEach ，整个流就会恢复运行。到底哪种方式能更有效地帮助我们理解Strea流水线中的每个操作（比如 map 、 filter 、 limit ）产生的输出？

这就是流操作方法 peek 大显身手的时候。 peek 的设计初衷就是在流的每个元素恢复运行之前，插执行一个动作。但是它不像 forEach 那样恢复整个流的运行，而是在一个元素上完操作之后，它只会操作顺承到流水线中的下一个操作。下面的这段代码中，我们使用 peek 输出了Stream流水线操作之和操作之后的中间值：

```
List<Integer> result = Stream.of(2, 3, 4, 5)
            .peek(x -> System.out.println("taking from stream: " + x)).map(x -> x + 17)
            .peek(x -> System.out.println("after map: " + x)).filter(x -> x % 2 == 0)
            .peek(x -> System.out.println("after filter: " + x)).limit(3)
            .peek(x -> System.out.println("after limit: " + x)).collect(toList());
```

通过 peek 操作我们能清楚地了解流水线操作中每一步的输出结果：

```
taking from stream: 2
after map: 19
taking from stream: 3
after map: 20
after filter: 20
after limit: 20
taking from stream: 4
after map: 21
taking from stream: 5
after map: 22
after filter: 22
after limit: 22
```

小结

- Lambda表达式能提升代码的可读性和灵活性。

- 如果你的代码中使用了匿名类，尽量用Lambda表达式替换它们，但是要注意二者间语义的微妙差，比如关键字 this ，以及变量隐藏。
- 跟Lambda表达式比起来，方法引用的可读性更好 。
- 尽量使用Stream API替换迭代式的集合处理。
- Lambda表达式有助于避免使用面向对象设计模式时容易出现的僵化的模板代码，典型的比如策略式、模板方法、观察者模式、责任链模式，以及工厂模式。
- 即使采用了Lambda表达式，也同样可以进行单元测试，但是通常你应该关注使用了Lambda表达的方法的行为。
- 尽量将复杂的Lambda表达式抽象到普通方法中。
- Lambda表达式会让栈跟踪的分析变得更为复杂。
- 流提供的 peek 方法在分析Stream流水线时，能将中间变量的值输出到日志中，是非常有用的工具。

代码

Github:chap8

Gitee:chap8