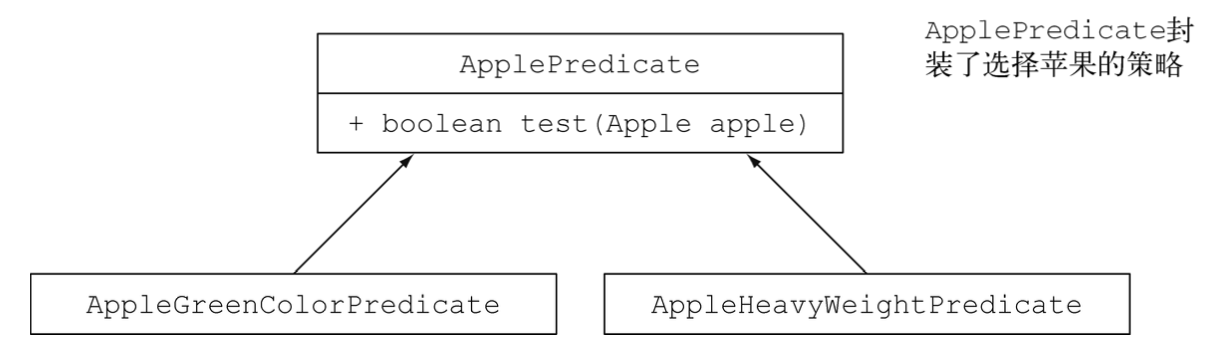
从上一节中看出，需要一种比添加很多参数更好的方法来应对变化的需求。一种可能的解决方案是对你的选择标准建模:你考虑的是苹果，需要根据Apple的某些属性(比如它是绿色的吗?重量超过150克吗?)来返回一个boolean值。我们把它称为谓词(即一个返回boolean值的函数)。让我们定义一个接口来对选择标准建模以及它的实现：



你可以把这些标准看作filter方法的不同行为。你刚做的这些和“策略设计模式”相关，它让你定义一族算法，把它们封装起来（称为“策略”），然后在运行时选择一个算法。在这里，算法族就是ApplePredicate，不同的策略就是AppleHeavyWeightPredicate和AppleGreenColorPredicate。

但是，该怎么利用ApplePredicate的不同实现呢？你需要filterApple方法接受ApplePredicate对象，对Apple做条件测试。这就是行为参数化：让方法接受多种行为（或战略）作为参数，并在内部使用，来完成不同的行为。

要在我们的例子中实现这一点，你要给filterApples方法添加一个参数，让它接受ApplePredicate对象。这在软件工程上有很多好处：现在你把filterApples方法迭代集合的逻辑与你要应用到集合中每个元素的行为区分开了（这里是一个谓词）区分开了。

#### 根据抽象条件筛选

利用ApplePredicate改过之后，filter方法看起来是这样的：

public static List<Apple> filterApples1(List<Apple> inventory, ApplePredicate p) {

List<Apple> result = new ArrayList<>();  
 for (Apple apple : inventory) {  
 if (p.test(apple)) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

#### 1、传递代码/行为

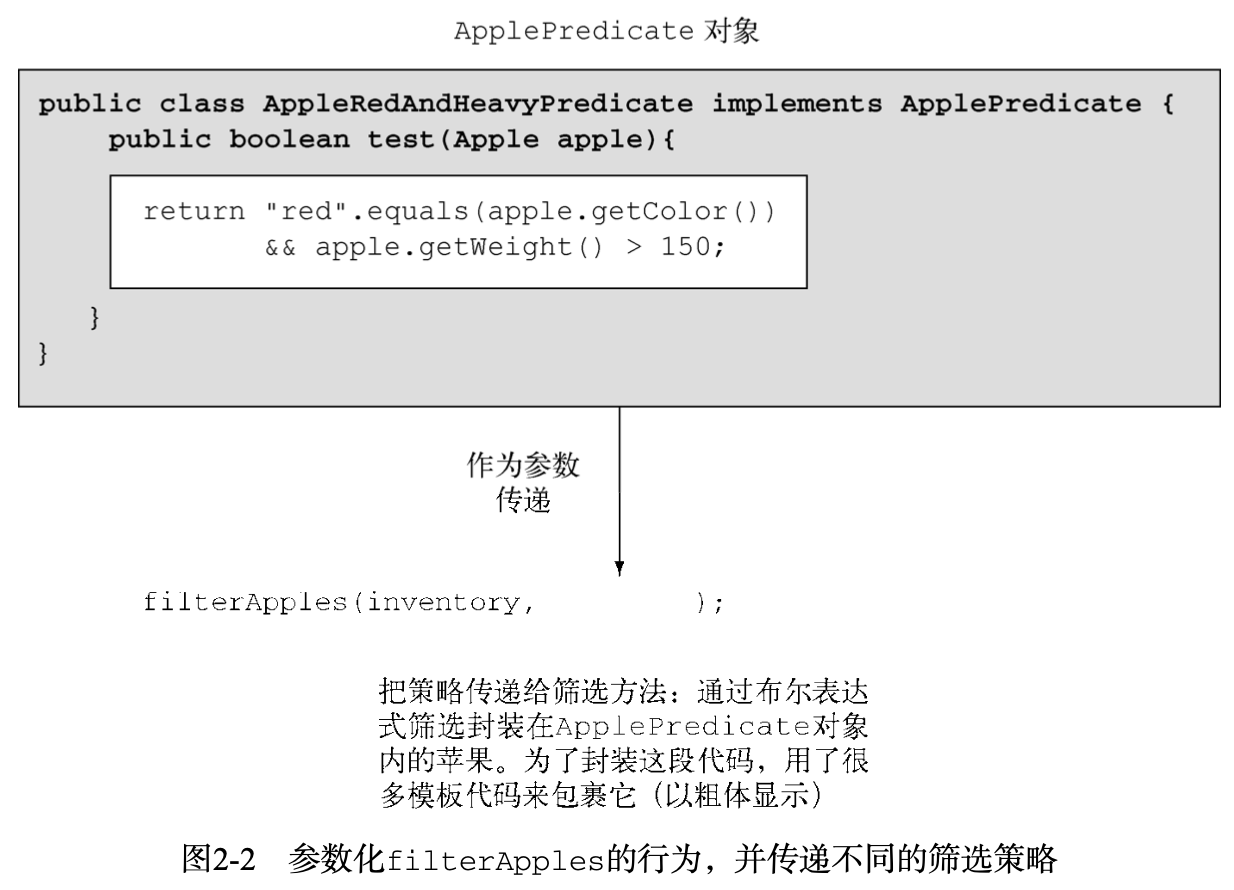
现在你可以创建不同的ApplePredicate对象，并将它们传递给filterApple方法。比如，如果让你找出所有重量超过150克的红苹果，你只需要创建一个类来实现ApplePredicate就行了。你的代码现在足够灵活，可以应对任何涉及苹果属性的需求变更了：

public class AppleRedAndHeavyPredicate implements ApplePredicate {

@Override  
 public boolean test(FilteringApples.Apple apple) {  
 return "red".equals(apple.getColor())  
 && apple.getWeight() > 150;  
 }  
}

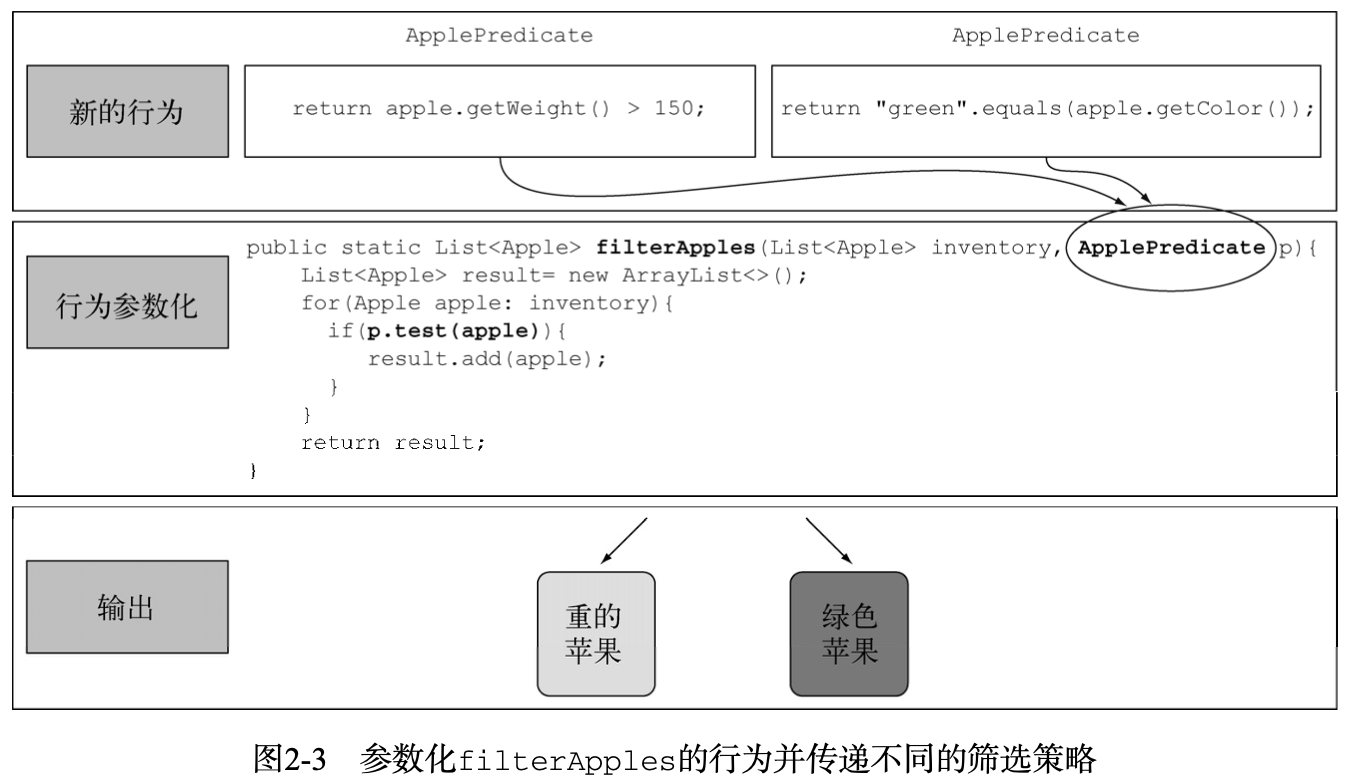
filterApples方法的行为取决于你通过ApplePredicate对象传递的代码，换句话说，你把filterApples方法的行为参数化了！

例子中，唯一重要的代码时test方法的实现；正是它定义了filterApples方法的新行为。但令人遗憾的是，由于该filterApples方法只能接受对象，所以你必须把代码包裹在ApplePredicate对象里。你的做法就类似于在内联“传递代码”，因为你是通过一个实现了test方法的对象来传递布尔表达式“red”.equals(apple.getColor()) && apple.getWeight() > 150传递给filterApples方法，而无需定义多个ApplePredicate类，从而去掉不必要的代码。



#### 2、多种行为，一个参数

行为参数的好处在于你可以把迭代要筛选的集合的逻辑对集合中每个元素应用的行为区分开来。这样你可以重复使用同一个方法，给它不同的行为来达到不同的目的：



行为参数化是一个有用的概念。你应该把它放进你的工具箱，用来编写灵活的API。