前言

用户的需求不断的在变化，因而代码也要进行重构，如果前期设计的不好，那么代码就会不断改变，大改变，如果设计的足够好，可以少改代码甚至不改变代码

概念

行为参数化，就是可以帮助我们处理频繁变化需求的一种软件开发模式，通俗的说，就是拿出一个代码块，把它准备好，却不去执行它。这个代码块以后可以被程序的其他部分调用，这就意味着我们可以推迟这块代码的执行

需求

农场里有很多苹果，我们要去筛选特定的苹果

片段一

开始的别人给我们提出的需求是，筛选出来所有颜色是绿色的苹果

```
1. public static List<Apple> filterGreenApples(List<Apple>
inventory) {
2.     List<Apple> result = new ArrayList<>();
3.     for (Apple apple : inventory) {
4.         if ("green".equals(apple.getColor())) {
5.             result.add(apple);
6.         }
7.     }
8.     return result;
9. }
```

这里是将绿色的筛选条件写死

片段二

又来了新需求，筛选红色苹果的接口

```
10. public static List<Apple> filterGreenApples(List<Apple>
inventory, String color) {
11.     List<Apple> result = new ArrayList<>();
12.     for (Apple apple : inventory) {
13.         if (color.equals(apple.getColor())) {
14.             result.add(apple);
15.         }
16.     }
17.     return result;
18. }
```

为了防止后面需求又有变化，这里讲筛选条件也传递进来，不管什么颜色都不用修改接口了

片段三

又来了新的需求，筛选重量超过150g的苹果

```
1. public static List<Apple> filterApplesByWeight(List<Apple>
inventory, int weight) {
2.     List<Apple> result = new ArrayList<>();
3.     for (Apple apple : inventory) {
4.         if (apple.getWeight() > weight) {
5.             result.add(apple);
6.         }
7.     }
8.     return result;
9. }
```

那么新写个接口，将weight也传递进来进行筛选

片段四

需求又来了，筛选重量超过150g并且红色的苹果

```
1. public static List<Apple> filterApples(List<Apple> inventory,
String color, int weight) {
2.     List<Apple> result = new ArrayList<>();
3.     for (Apple apple : inventory) {
4.         if (color.equals(apple.getColor()) &&
apple.getWeight() > weight) {
5.             result.add(apple);
6.         }
7.     }
8.     return result;
9. }
```

加上颜色和重量两个联合筛选条件当做参数

片段五

前面已经将颜色和重量联合起来当做筛选条件了，那么需求还会不会变？颜色，重量做筛选条件，会不会有其他？产地、品种、口感.....

那么采用策略模式：

定义一个苹果筛选的策略接口

```
1. public interface ApplePredicate {
2.     boolean test(Apple apple);
3. }
```

分别定义筛选颜色和筛选重量的实现类

```
1. //绿色苹果
2. public class GreenApplePredicate implements ApplePredicate {
3.     @Override
4.     public boolean test(Apple apple) {
5.         return apple.getColor().equals("green");
6.     }
7. }
8.
9. //大苹果
10. public class BigApplePredicate implements ApplePredicate {
11.     @Override
12.     public boolean test(Apple apple) {
13.         return apple.getWeight() > 150;
14.     }
15. }
```

编写筛选接口

```
1. public static List<Apple> filterApples(List<Apple> inventory,
ApplePredicate p) {
2.     List<Apple> result = new ArrayList<>();
3.     for (Apple apple : inventory) {
4.         if (p.test(apple)) {
5.             result.add(apple);
6.         }
7.     }
8.     return result;
9. }
```

调用代码

```
List<Apple> heavyApples = filterApples(inventory, new
```

```
BigApplePredicate ());
```

片段六

前面发现，通过策略模式可以做到筛选接口不变，但是每次不同的筛选方式来了，都要新写一种策略实现类，调用的时候都要**创建策略实现类**

那么能否不去创建策略实现类呢？答案是匿名，在java中有匿名内部类。如

```
1. EasyCache.execute(new Runnable() {
2.     @Override
3.     public void run() {
4.         try {
5.             WarningInstanceVo value =
preliminaryStatisticsService.warningInstance(item, queryType);
6.             EasyCache.put(key, value);
7.         } catch (BizException e) {
8.             e.printStackTrace();
9.         } catch (InterruptedException e) {
10.             e.printStackTrace();
11.         }
12.     }
13. });
```

那么同理：不去编写策略实现类，通过匿名的方式解决，调用代码改造如下：

```
1. List<Apple> greenApples = filterApples(inventory, new
ApplePredicate() {
2.     @Override
3.     public boolean test(Apple apple) {
4.         return apple.getColor().equals("green");
5.     }
6. });
```

代码看起来就简洁很多了，但是调用代码实际上就 一行代码有用

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

那么在java8中 可以这样简写

```
1. List<Apple> greenApples = filterApples(inventory, (Apple
apple) -> apple.getColor().equals("green"));
```

这里的 `(Apple apple) -> apple.getColor().equals("green")` 就是**lambda**表达式了，该表达式可以简写如下：

```
1. List<Apple> greenApples = filterApples(inventory, apple ->
apple.getColor().equals("green"));
```

片段七

前面只能筛选苹果，那么能不能做一个筛选香蕉，菠萝，西瓜通用的呢?

```
1. public interface Predicate<T>{
2.     boolean test(T t);
3. }
4.
5.
6. public static <T> List<T> filter(List<T> list, Predicate<T> p)
{
7.     List<T> result = new ArrayList<>();
8.         for(T e: list){
9.             if(p.test(e)){
10.                 result.add(e);
11.             }
12.     }
13.     return result;
14. }
```

真实实例

排序

实体类

```
1. public class User implements Comparator<User>{
2.     private int id;
3.     private String code;
4.     private String name;
5.     private int sex;
6. }
```

jdk8之前，调用代码

```
1. List<User> list = new LinkedList<>();
2. Collections.sort(list, new Comparator<User>() {
3.     @Override
4.     public int compare(User o1, User o2) {
5.         return 0;
6.     }
7. });
```

jdk8之后，调用代码

```
1. List<User> list = new LinkedList<>();
2. list.sort((User u1 , User u2) ->
u1.getCode().compareTo(u2.getCode()));
```

线程调用

```
1. Thread t = new Thread(new Runnable() {
2.     public void run(){
3.         System.out.println("Hello world");
4.     }
5. });
```

jdk8之后

```
1. Thread t = new Thread(() -> System.out.println("Hello
world"));
```

其他

GUI等