

## 61 | 策略模式（下）：如何实现一个支持给不同大小文件排序的小程序？

2020-03-23 王争

设计模式之美

[进入课程 >](#)



# 策略模式（下）

讲述：冯永吉

时长 08:59 大小 8.24M



上一节课，我们主要介绍了策略模式的原理和实现，以及如何利用策略模式来移除 if-else 或者 switch-case 分支判断逻辑。今天，我们结合“给文件排序”这样一个具体的例子，来详细讲一讲策略模式的设计意图和应用场景。

除此之外，在今天的讲解中，我还会通过一步一步地分析、重构，给你展示一个设计模式是如何“创造”出来的。通过今天的学习，你会发现，**设计原则和思想其实比设计模式更加普适和重要，掌握了代码的设计原则和思想，我们甚至可以自己创造出来新的设计模式。** ☆

话不多说，让我们正式开始今天的学习吧！

## 问题与解决思路

假设有这样一个需求，希望写一个小程序，实现对一个文件进行排序的功能。文件中只包含整型数，并且，相邻的数字通过逗号来区隔。如果由你来编写这样一个小程序，你会如何实现呢？你可以把它当作面试题，先自己思考一下，再来看我下面的讲解。

你可能会说，这不是很简单嘛，只需要将文件中的内容读取出来，并且通过逗号分割成一个一个的数字，放到内存数组中，然后编写某种排序算法（比如快排），或者直接使用编程语言提供的排序函数，对数组进行排序，最后再将数组中的数据写入文件就可以了。

但是，如果文件很大呢？比如有 10GB 大小，因为内存有限（比如只有 8GB 大小），我们没办法一次性加载文件中的所有数据到内存中，这个时候，我们就要利用外部排序算法（具体怎么做，可以参看我的另一个专栏《数据结构与算法之美》中的“排序”相关章节）了。


如果文件更大，比如有 100GB 大小，我们为了利用 CPU 多核的优势，可以在外部排序的基础之上进行优化，加入多线程并发排序的功能，这就有点类似“单机版”的 MapReduce。

如果文件非常大，比如有 1TB 大小，即便是单机多线程排序，这也算很慢了。这个时候，我们可以使用真正的 MapReduce 框架，利用多机的处理能力，提高排序的效率。

## 代码实现与分析

解决思路讲完了，不难理解。接下来，我们看一下，如何将解决思路翻译成代码实现。

我先用最简单直接的方式实现将它实现出来。具体代码我贴在下面了，你可以先看一下。因为我们是在讲设计模式，不是讲算法，所以，在下面的代码实现中，我只给出了跟设计模式相关的骨架代码，并没有给出每种排序算法的具体代码实现。感兴趣的话，你可以自行实现一下。

 复制代码

```
1 public class Sorter {
2     private static final long GB = 1000 * 1000 * 1000;
3
4     public void sortFile(String filePath) {
5         // 省略校验逻辑
6         File file = new File(filePath);
7         long fileSize = file.length();
```

```

8     if (fileSize < 6 * GB) { // [0, 6GB)
9         quickSort(filePath);
10    } else if (fileSize < 10 * GB) { // [6GB, 10GB)
11        externalSort(filePath);
12    } else if (fileSize < 100 * GB) { // [10GB, 100GB)
13        concurrentExternalSort(filePath);
14    } else { // [100GB, ~)
15        mapreduceSort(filePath);
16    }
17 }
18
19 private void quickSort(String filePath) {
20     // 快速排序
21 }
22
23 private void externalSort(String filePath) {
24     // 外部排序
25 }
26
27 private void concurrentExternalSort(String filePath) {
28     // 多线程外部排序
29 }
30
31 private void mapreduceSort(String filePath) {
32     // 利用MapReduce多机排序
33 }
34 }
35
36 public class SortingTool {
37     public static void main(String[] args) {
38         Sorter sorter = new Sorter();
39         sorter.sortFile(args[0]);
40     }
41 }

```

在“编码规范”那一部分我们讲过，函数的行数不能过多，最好不要超过一屏的大小。所以，为了避免 sortFile() 函数过长，我们把每种排序算法从 sortFile() 函数中抽离出来，拆分成 4 个独立的排序函数。

如果只是开发一个简单的工具，那上面的代码实现就足够了。毕竟，代码不多，后续修改、扩展的需求也不多，怎么写都不会导致代码不可维护。但是，如果我们是在开发一个大型项目，排序文件只是其中的一个功能模块，那我们就要在代码设计、代码质量上下点儿功夫了。只有每个小的功能模块都写好，整个项目的代码才能不差。

在刚刚的代码中，我们并没有给出每种排序算法的代码实现。实际上，如果自己实现一下的话，你会发现，每种排序算法的实现逻辑都比较复杂，代码行数都比较多。所有排序算法的代码实现都堆在 Sorter 一个类中，这就会导致这个类的代码很多。而在“编码规范”那一部分中，我们也讲到，一个类的代码太多也会影响到可读性、可维护性。除此之外，所有的排序算法都设计成 Sorter 的私有函数，也会影响代码的可复用性。

## 代码优化与重构

只要掌握了我们之前讲过的设计原则和思想，针对上面的问题，即便我们想不到该用什么设计模式来重构，也应该能知道该如何解决，那就是将 Sorter 类中的某些代码拆分出来，独立成职责更加单一的小类。实际上，拆分是应对类或者函数代码过多、应对代码复杂性的一个常用手段。按照这个解决思路，我们对代码进行重构。重构之后的代码如下所示：

 复制代码

```
1 public interface ISortAlg {
2     void sort(String filePath);
3 }
4
5 public class QuickSort implements ISortAlg {
6     @Override
7     public void sort(String filePath) {
8         //...
9     }
10 }
11
12 public class ExternalSort implements ISortAlg {
13     @Override
14     public void sort(String filePath) {
15         //...
16     }
17 }
18
19 public class ConcurrentExternalSort implements ISortAlg {
20     @Override
21     public void sort(String filePath) {
22         //...
23     }
24 }
25
26 public class MapReduceSort implements ISortAlg {
27     @Override
28     public void sort(String filePath) {
29         //...
30     }
31 }
```


```

32 public class Sorter {
33     private static final long GB = 1000 * 1000 * 1000;
34
35     public void sortFile(String filePath) {
36         // 省略校验逻辑
37         File file = new File(filePath);
38         long fileSize = file.length();
39         ISortAlg sortAlg;
40         if (fileSize < 6 * GB) { // [0, 6GB)
41             sortAlg = new QuickSort();
42         } else if (fileSize < 10 * GB) { // [6GB, 10GB)
43             sortAlg = new ExternalSort();
44         } else if (fileSize < 100 * GB) { // [10GB, 100GB)
45             sortAlg = new ConcurrentExternalSort();
46         } else { // [100GB, ~)
47             sortAlg = new MapReduceSort();
48         }
49         sortAlg.sort(filePath);
50     }
51 }
52

```

经过拆分之后，每个类的代码都不会太多，每个类的逻辑都不会太复杂，代码的可读性、可维护性提高了。除此之外，我们将排序算法设计成独立的类，跟具体的业务逻辑（代码中的 if-else 那部分逻辑）解耦，也让排序算法能够复用。这一步实际上就是策略模式的第一步，也就是将策略的定义分离出来。

实际上，上面的代码还可以继续优化。每种排序类都是无状态的，我们没必要在每次使用的时候，都重新创建一个新的对象。所以，我们可以使用工厂模式对对象的创建进行封装。按照这个思路，我们对代码进行重构。重构之后的代码如下所示：

 复制代码

```

1 public class SortAlgFactory {
2     private static final Map<String, ISortAlg> algs = new HashMap<>();
3
4     static {
5         algs.put("QuickSort", new QuickSort());
6         algs.put("ExternalSort", new ExternalSort());
7         algs.put("ConcurrentExternalSort", new ConcurrentExternalSort());
8         algs.put("MapReduceSort", new MapReduceSort());
9     }
10
11     public static ISortAlg getSortAlg(String type) {
12         if (type == null || type.isEmpty()) {
13             throw new IllegalArgumentException("type should not be empty.");
14         }
15     }
16 }


```

```

15     return algs.get(type);
16 }
17 }
18
19 public class Sorter {
20     private static final long GB = 1000 * 1000 * 1000;
21
22     public void sortFile(String filePath) {
23         // 省略校验逻辑
24         File file = new File(filePath);
25         long fileSize = file.length();
26         ISortAlg sortAlg;
27         if (fileSize < 6 * GB) { // [0, 6GB)
28             sortAlg = SortAlgFactory.getSortAlg("QuickSort");
29         } else if (fileSize < 10 * GB) { // [6GB, 10GB)
30             sortAlg = SortAlgFactory.getSortAlg("ExternalSort");
31         } else if (fileSize < 100 * GB) { // [10GB, 100GB)
32             sortAlg = SortAlgFactory.getSortAlg("ConcurrentExternalSort");
33         } else { // [100GB, ~)
34             sortAlg = SortAlgFactory.getSortAlg("MapReduceSort");
35         }
36         sortAlg.sort(filePath);
37     }
38 }

```

经过上面两次重构之后，现在的代码实际上已经符合策略模式的代码结构了。我们通过策略模式将策略的定义、创建、使用解耦，让每一部分都不至于太复杂。不过，Sorter 类中的 sortFile() 函数还是有一堆 if-else 逻辑。这里的 if-else 逻辑分支不多、也不复杂，这样写完全没问题。但如果你特别想将 if-else 分支判断移除掉，那也是有办法的。我直接给出代码，你一看就能明白。实际上，这也是基于查表法来解决的，其中的“algs”就是“表”。

 复制代码

```

1 public class Sorter {
2     private static final long GB = 1000 * 1000 * 1000;
3     private static final List<AlgRange> algs = new ArrayList<>();
4     static {
5         algs.add(new AlgRange(0, 6*GB, SortAlgFactory.getSortAlg("QuickSort")));
6         algs.add(new AlgRange(6*GB, 10*GB, SortAlgFactory.getSortAlg("ExternalSort")));
7         algs.add(new AlgRange(10*GB, 100*GB, SortAlgFactory.getSortAlg("ConcurrentExternalSort")));
8         algs.add(new AlgRange(100*GB, Long.MAX_VALUE, SortAlgFactory.getSortAlg("MapReduceSort")));
9     }
10
11     public void sortFile(String filePath) {
12         // 省略校验逻辑
13         File file = new File(filePath);
14         long fileSize = file.length();

```



```

15     ISortAlg sortAlg = null;
16     for (AlgRange algRange : algs) {
17         if (algRange.inRange(fileSize)) {
18             sortAlg = algRange.getAlg();
19             break;
20         }
21     }
22     sortAlg.sort(filePath);
23 }
24
25 private static class AlgRange {
26     private long start;
27     private long end;
28     private ISortAlg alg;
29
30     public AlgRange(long start, long end, ISortAlg alg) {
31         this.start = start;
32         this.end = end;
33         this.alg = alg;
34     }
35
36     public ISortAlg getAlg() {
37         return alg;
38     }
39
40     public boolean inRange(long size) {
41         return size >= start && size < end;
42     }
43 }
44 }

```

现在的代码实现就更加优美了。我们把可变的部分隔离到了策略工厂类和 Sorter 类中的静态代码段中。当要添加一个新的排序算法时，我们只需要修改策略工厂类和 Sort 类中的静态代码段，其他代码都不需要修改，这样就将代码改动最小化、集中化了。

你可能会说，即便这样，当我们添加新的排序算法的时候，还是需要修改代码，并不完全符合开闭原则。有什么办法让我们完全满足开闭原则呢？

对于 Java 语言来说，我们可以通过反射来避免对策略工厂类的修改。具体是这么做的：我们通过一个配置文件或者自定义的 annotation 来标注都有哪些策略类；策略工厂类读取配置文件或者搜索被 annotation 标注的策略类，然后通过反射了动态地加载这些策略类、创建策略对象；当我们新添加一个策略的时候，只需要将这个新添加的策略类添加到配置文件或者用 annotation 标注即可。还记得上一节课的课堂讨论题吗？我们也可以用这种方法来解决。

对于 Sorter 来说，我们可以通过同样的方法来避免修改。我们通过将文件大小区间和算法之间的对应关系放到配置文件中。当添加新的排序算法时，我们只需要改动配置文件即可，不需要改动代码。

## 重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

一提到 if-else 分支判断，有人就觉得它是烂代码。如果 if-else 分支判断不复杂、代码不多，这并没有任何问题，毕竟 if-else 分支判断几乎是所有编程语言都会提供的语法，存在即有理由。遵循 KISS 原则，怎么简单怎么来，就是最好的设计。非得用策略模式，搞出 n 多类，反倒是一种过度设计。

一提到策略模式，有人就觉得，它的作用是避免 if-else 分支判断逻辑。实际上，这种认识是很片面的。策略模式主要的作用还是解耦策略的定义、创建和使用，控制代码的复杂度，让每个部分都不至于过于复杂、代码量过多。除此之外，对于复杂代码来说，策略模式还能让其满足开闭原则，添加新策略的时候，最小化、集中化代码改动，减少引入 bug 的风险。

实际上，设计原则和思想比设计模式更加普适和重要。掌握了代码的设计原则和思想，我们能更清楚的了解，为什么要用某种设计模式，就能更恰到好处地应用设计模式。

## 课堂讨论

1. 在过去的项目开发中，你有没有用过策略模式，都是为了解决什么问题才使用的？
2. 你可以说一说，在什么情况下，我们才有必要去掉代码中的 if-else 或者 switch-case 分支逻辑呢？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。



# 学习 6 小时， 「免费」领课程！



🕒 3月23日-3月29日

【点击】图片, 查看详情, 参与学习

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 60 | 策略模式（上）：如何避免冗长的if-else/switch分支判断代码？

下一篇 加餐二 | 设计模式、重构、编程规范等相关书籍推荐

## 精选留言 (28)

💬 写留言



唔多志

2020-03-23

"设计原则和思想比设计模式更加普适和重要"，被这句话一下子点醒了。可以这样说，设计原则和思想是更高层次的理论和指导原则，设计模式只是这些理论和指导原则下，根据经验和场景，总结出来的编程范式。

💬 1

👍 20



业余爱好者

2020-03-23

策略集合的信息也可以定义成枚举，可以放在数据库，可以放kv配置中心，等等。都是一样的道理。

策略模式是对策略的定义，创建和使用解藕。定义和创建的过程与业务逻辑关系不大，写在一起会影响可读性。...

展开 ▾



3



韵呀

2020-03-23

最近在做公司的告警模块，对接收方式用了策略模式，以前也不知道这就是策略模式，只是觉得模块和思想解藕了。

还有接收方式的选择策略和升级策略都用了策略模式。

非要去掉if-else，我觉得如果分支不多，简单，以后可扩展性不强可以不去掉。否则就换...

展开 ▾



2



Jxin

2020-03-23

1.为了让调度的代码更优雅时使用。（就调度策略的代码而言，可读性高。理解结构后，阅读的心智负担低，因为调度的原理已经抽象成了共同的type查表，无需逐行检阅分支判断。像一些与持久数据相关的策略，有时为了兼容老数据或则平滑过度，无法全采用type查表，这时就需要结合if来实现。所以采用if会让我误以为是这种场景，进而逐行检阅）。

...

展开 ▾



1



Jackey

2020-03-23

想想以前都没用过策略模式，一会到公司看看我的if-else有没有必要优化



1



攻城拔寨

2020-03-23

1. 奖励系统，奖励用户不同奖励是通过后台配置，从多个奖励中选择一种或者多种。后端代码根据不同的奖励类型找到对应的奖励策略类，执行对应的代码。

2. 我觉得在 if-else 每个分支代码比较多是时候就应该拆分了。但是拆分不一定就能通过设计模式拆分，很多时候把分支逻辑抽成另一个方法就足够了，代码简洁，方便维护。

我相当同意老师 “设计原则和思想比设计模式更加普适和重要。” 这句话。

展开 ▾



1



王飞洋

2020-03-23

GB = 1024 \* 1024 \* 1024吧

展开 ∨



1



黄林晴

2020-03-23

设计原则和思想比设计模式更加普适和重要，就像掌握算法思想比死扣每种算法的实现更重要！

虽说实践是检验真理的唯一标准，但没有理论何来实践呢



1



忆水寒

2020-03-23

1、用过策略模式。我们在自研的rpc框架中为了灵活的设置线程数量（类似于netty里面的策略模式，默认情况下线程数量为CPU两倍，也可以指定）。

2、在每个if else里面的代码都有相似功能，可以抽取出来。

展开 ∨



1



南山

2020-03-23

1:随处可见，比如根据不同的配置选择不同的分配或者负载均衡策略。

2:策略可预期的经常变更或者选择策略的条件判定方式比较复杂时可以考虑不使用if else方式实现，反过来则if else 没什么毛病

展开 ∨



1



iamjohnnyzhuang

2020-03-24

策略模式平时开发用的比较多，主要目的还是解耦策略的定义和使用。在Java中，比较喜欢用枚举策略模式。可以定义一个枚举类，提供静态方法根据传入的Type动态返回对应的枚举类。然后在枚举类中定义执行策略的抽象方法，这样迫使每个枚举都得去实现。对于策略不是非常复杂的情况下，这样可以集中管理这一批策略，新增策略的时候也只要在这个枚举类中添加。但是如果策略很复杂会导致这个类非常庞大，还是用传统的方法不同...

展开 ∨



朱晋君

2020-03-24

- 1.策略模式的使用还是挺多的，比如贷款系统中，业务系统需要查询不同渠道的征信报告，但是各渠道查询方式是不一样的，就用策略模式做了处理，客户端传入渠道编号来获取查询的渠道策略实现征信查询
- 2.if-else或switch中每个分支的实现业务上是由共性的，可以抽象出来，这样可以用策略模式来取代分支。但是没有必要所有的都去掉...

展开 ▾



**辣么大**

2020-03-24

思考题1：很久没写工程代码，但接触的科研算法对比性能时一般采用策略模式。通过输入不同的参数调用不同的算法。

思考题2：通过策略模式的学习，可以看出实现简单的话一开始是不需要使用策略模式的。if else中实现的逻辑都差不多，就是差不多都是干一件事（上节课例子中的购物，本节的文件排序）短的 if else相对易读，代码好理解。使用了策略模式要小心过度设计和影响了...

展开 ▾



**jaryoung**

2020-03-23

在过去的项目开发中，你有没有用过策略模式，都是为了解决什么问题才使用的？

答：业务类型过多，未来还可能继续扩展

你可以说一说，在什么情况下，我们才有必要去掉代码中的 if-else 或者 switch-case 分支逻辑呢？

答：业务逻辑过于复杂（大部分都是条件判断），后期需求不断来。

展开 ▾



**刘大明**

2020-03-23

之前用策略模式做过通过解析统一入参里面的某一个类型来决定实现一个什么样的业务逻辑。当业务类型越来越多的时候，if条件分支也越来越多的时候，如果不做调整，实际上就违反了开闭原则，代码结构越来越乱。



**潇潇雨歇**

2020-03-23

设计原则和思想是道

展开 ▾



Frank

2020-03-23

打卡 今日学习使用策略模式实现支持不同大小的文件排序，收获如下：

从文章中给出简单的需求，需求分析，写出最小原型，基于设计原则和利用策略模式经过三个版本的迭代，最终写出一个比较好的代码模型。通过该案例，对之前学习过的设计原则、编码规范以及策略模式有了一个很好的理解。完成案例后，算是对之前课程中有关“非业务系统需求分析与设计”作了一个简单的复习。同时深深体会到“设计原则和...  
展开



Frank

2020-03-23

1，最近在做规则引擎，前端用户通过页面设置出业务执行的流程图，而流程图中包含数据源节点、条件判断节点、业务分发节点、业务事件执行节点。每一种节点执行的业务校验规则都不同，这个时候适合策略模式。使用策略模式的好处是：以后随着业务的发展很有可能出现其他类型的节点，所以这个时候采用策略模式非常合适，易扩展易维护。另外在整个流程流转的规则上采用了模板方法。...

展开



Heaven

2020-03-23

1.在之前,有一个用户拥有不同的用户组,对不同的用户组拥有这不同的处理逻辑,这就是一种策略模式,通过用户中的不同用户组名,来获取不同的用户组实例,执行其中的处理方式  
2.如果对于判断逻辑中,需要执行的逻辑较为复杂,可以抽取出相同的接口的话,就使用策略模式,将其抽取出接口,并利用策略工厂进行获取策略实例,从而执行对应处理

展开



守拙

2020-03-23

在什么情况下，我们才有必要去掉代码中的 if-else 或者 switch-case 分支逻辑呢？

根据KISS原则，默认我们不会主动去优化if-else/switch-case.

只有当if-else严重影响代码可读性与可维护性，才考虑使用策略模式+工厂方法模式优化分支逻辑.

展开



