

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？

王争 · 设计模式之美



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，你会如何来实现

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         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("ConcurrentExt

```

```

8     algs.add(new AlgRange(100*GB, Long.MAX_VALUE, SortAlgFactory.getSortAlg("MapR
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，我们只需要修改变策略工厂类和 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。

AI智能总结

本文详细介绍了如何利用策略模式实现一个支持不同大小文件排序的小程序。作者通过逐步分析和重构的方式，展示了从简单的排序算法到外部排序、多线程并发排序，再到MapReduce框架的应用，逐步展示了不同文件大小下的排序实现思路。在代码实现与分析部分，作者给出了一个Sorter类，并提出了代码设计上的优化建议，强调了设计原则和思想的重要性。通过多次重构，最终实现了符合策略模式的代码结构，让每一部分都不至于过于复杂、代码量过多。文章强调了设计原则和思想比设计模式更加普适和重要，指出掌握了代码的设计原则和思想，能更恰到好处地应用设计模式。整体而言，本文通过一个具体的例子，详细讲解了策略模式的设计意图和应用场景，对读者进行了技术上的启发和指导。

© 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。

全部留言 (102)

最新 精选



永旭

2020-04-30

哪位了解无状态的类是什么意思？

作者回复: 你可以简单理解为：只有函数，没有属性。

共 3 条评论 >

👍 6



一万小时

2020-11-23

策略模式看着像是个工厂模式

作者回复: 目的不同



👍 3



Laughing

2020-11-24

在之前项目中，有个缓存的实现开始直接使用的redis来实现的，后来使用内存缓存替换，这样一来我使用了策略模式根据缓存key的类型，自动选择策略进行缓存。

作者回复: 嗯嗯     



Fitch Kuma

2020-07-05

对查表法来避免if-else很有感触，之前设计Jenkins pipeline就是用Map来存放不同的component部署到不同server的映射关系的。感觉策略模式的本质就是多态？

作者回复: 嗯，用多态来实现的



木又寸

2020-05-27

策略类膨胀的情况，小争哥有没一些最佳实践可以分享？

业务实现上，可能一个父级策略下有多个子策略（譬如促销活动有满减/返现/立减等，满减规则下又有几种复杂的优惠策略），这时候把所有满减规则塞进一个策略类，有些偏离我们的初衷；将子策略拆分为独立的策略，策略类的数目又会急剧膨胀。

如何取舍？或者能否混合其他模式来适应这种场景？

作者回复: 只保留满减、返现、立减等子策略，子策略放到子策略中，不单独抽离出来实现，这样可以吗？

共 3 条评论 >



吴小智

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。

共 6 条评论 >



岁月

2020-04-02

看到这里我感觉好多设计模式都很类似, 都是先针对接口编程, 然后再加个查表法😂

共 10 条评论 >

👍 77



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。

创建型模式是对创建和使用的解藕。

做什么和怎么做是应该解藕的, 使用者并不关心具体的细节。

在业务逻辑中, 与业务逻辑不大的代码应该放在外部。就像mvc的三层架构是业界的最佳实践。在service层调用dao层, 而不是直接jdbc, 因为如何操作数据库是个复杂的过程, 却又与业务逻辑无关, 所以单独抽出一层, 代码结构变得更加清晰。

声明式编程很火。它就是把使用和内部实现原理解藕。java, spring的各种注解, 声明式事务等等。使用者一个注解解决所有问题, 无需关心底层。

业务逻辑无关的放在一起影响可读性, 即使自己过一段时间看自己的代码也会迷惑。写代码要有用户思维, 不光是提供的api满足kiss原则, 内部的实现也是一样。能放在外面单位代码尽量怎么放在外面。只要能表达逻辑即可。

共 1 条评论 >

👍 43



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)。

2. 感觉问反了, 应该是什么场景下采用ifelse和witch。毕竟这两个的场景少些。答案是, 在逻辑不复杂又不好用卫语句时采用。能用卫语句就不要ifelse。因为个人而言, 看到return或continue能很明确跳出逻辑, else脑袋得转一下, 当然这可能是个人习惯影响。但ifelse还是只在逻辑简单, 没啥嵌套(一个函数内部不宜嵌套过多), 且语义符合的场景用好些。

3.原则和思想毕竟是指导核心，是地图和尺子，自然是最重要的。但是不等于设计模式就比其不重要了。硬要说的话，我认为两个其实一样重要。设计模式是场景的积累，拉近了普通人与天才的差距，对我这种菜鸡来说可能比设计原则还重要。毕竟从设计原则到设计模式的出现，需要丰富经验，扎实知识和妙手偶得。如果没有设计模式的铺垫，普通人拿着设计模式的知识点要一路走上高质量代码的层级，实属不易。

共 3 条评论 >

👍 25



空白昵称

2020-03-24

很早前使用了类似策略模式的代码做了一个从各种渠道上传下载文件。（当时大学毕业，没看过策略模式，但是学过C++的多态）其实也就是基于接口而非实现编程思想。

if-else使用，如果判断较少，且未来几乎不会有新需求要再加一个else时，保持if-else即可。其他的都可以使用策略模式。不过本节demo的实例，使用range来匹配，目前看是没问题的。但是如果需求变化，需要增加一个根据数据特征的动态分配排序算法，那么这里实现就麻烦了。所以需要根据具体需求来看使用什么模式...



👍 12