## 如何实现一个支持给不同大小文件排序的小程序？

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。

### 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。

public class Sorter {

private static final long GB = 1000 \* 1000 \* 1000;

public void sortFile(String filePath) {

// 省略校验逻辑

File file = new File(filePath);

long fileSize = file.length();

if (fileSize < 6 \* GB) { // [0, 6GB)

quickSort(filePath);

} else if (fileSize < 10 \* GB) { // [6GB, 10GB)

externalSort(filePath);

} else if (fileSize < 100 \* GB) { // [10GB, 100GB)

concurrentExternalSort(filePath);

} else { // [100GB, ~)

mapreduceSort(filePath);

}

}

private void quickSort(String filePath) {

// 快速排序

}

private void externalSort(String filePath) {

// 外部排序

}

private void concurrentExternalSort(String filePath) {

// 多线程外部排序

}

private void mapreduceSort(String filePath) {

// 利用MapReduce多机排序

}

}

public class SortingTool {

public static void main(String[] args) {

Sorter sorter = new Sorter();

sorter.sortFile(args[0]);

}

}

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：

public interface ISortAlg {

void sort(String filePath);

}

public class QuickSort implements ISortAlg {

@Override

public void sort(String filePath) {

//...

}

}

public class ExternalSort implements ISortAlg {

@Override

public void sort(String filePath) {

//...

}

}

public class ConcurrentExternalSort implements ISortAlg {

@Override

public void sort(String filePath) {

//...

}

}

public class MapReduceSort implements ISortAlg {

@Override

public void sort(String filePath) {

//...

}

}

public class Sorter {

private static final long GB = 1000 \* 1000 \* 1000;

public void sortFile(String filePath) {

// 省略校验逻辑

File file = new File(filePath);

long fileSize = file.length();

ISortAlg sortAlg;

if (fileSize < 6 \* GB) { // [0, 6GB)

sortAlg = new QuickSort();

} else if (fileSize < 10 \* GB) { // [6GB, 10GB)

sortAlg = new ExternalSort();

} else if (fileSize < 100 \* GB) { // [10GB, 100GB)

sortAlg = new ConcurrentExternalSort();

} else { // [100GB, ~)

sortAlg = new MapReduceSort();

}

sortAlg.sort(filePath);

}

}

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：

public class SortAlgFactory {

private static final Map<String, ISortAlg> algs = new HashMap<>();

static {

algs.put("QuickSort", new QuickSort());

algs.put("ExternalSort", new ExternalSort());

algs.put("ConcurrentExternalSort", new ConcurrentExternalSort());

algs.put("MapReduceSort", new MapReduceSort());

}

public static ISortAlg getSortAlg(String type) {

if (type == null || type.isEmpty()) {

throw new IllegalArgumentException("type should not be empty.");

}

return algs.get(type);

}

}

public class Sorter {

private static final long GB = 1000 \* 1000 \* 1000;

public void sortFile(String filePath) {

// 省略校验逻辑

File file = new File(filePath);

long fileSize = file.length();

ISortAlg sortAlg;

if (fileSize < 6 \* GB) { // [0, 6GB)

sortAlg = SortAlgFactory.getSortAlg("QuickSort");

} else if (fileSize < 10 \* GB) { // [6GB, 10GB)

sortAlg = SortAlgFactory.getSortAlg("ExternalSort");

} else if (fileSize < 100 \* GB) { // [10GB, 100GB)

sortAlg = SortAlgFactory.getSortAlg("ConcurrentExternalSort");

} else { // [100GB, ~)

sortAlg = SortAlgFactory.getSortAlg("MapReduceSort");

}

sortAlg.sort(filePath);

}

}

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”。

public class Sorter {

private static final long GB = 1000 \* 1000 \* 1000;

private static final List<AlgRange> algs = new ArrayList<>();

static {

algs.add(new AlgRange(0, 6\*GB, SortAlgFactory.getSortAlg("QuickSort")));

algs.add(new AlgRange(6\*GB, 10\*GB, SortAlgFactory.getSortAlg("ExternalSort")));

algs.add(new AlgRange(10\*GB, 100\*GB, SortAlgFactory.getSortAlg("ConcurrentExternalSort")));

algs.add(new AlgRange(100\*GB, Long.MAX\_VALUE, SortAlgFactory.getSortAlg("MapReduceSort")));

}

public void sortFile(String filePath) {

// 省略校验逻辑

File file = new File(filePath);

long fileSize = file.length();

ISortAlg sortAlg = null;

for (AlgRange algRange : algs) {

if (algRange.inRange(fileSize)) {

sortAlg = algRange.getAlg();

break;

}

}

sortAlg.sort(filePath);

}

private static class AlgRange {

private long start;

private long end;

private ISortAlg alg;

public AlgRange(long start, long end, ISortAlg alg) {

this.start = start;

this.end = end;

this.alg = alg;

}

public ISortAlg getAlg() {

return alg;

}

public boolean inRange(long size) {

return size >= start && size < end;

}

}

}

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。