如何在Java中读取超过内存大小的文件

读取文件内容，然后进行处理，在Java中我们通常利用 Files 类中的方法，将可以文件内容加载到内存，并流顺利地进行处理。但是，在一些场景下，我们需要处理的文件可能比我们机器所拥有的内存要大。此时，我们则需要采用另一种策略：部分读取它，并具有其他结构来仅编译所需的数据。

接下来，我们就来说说这一场景：当遇到大文件，无法一次载入内存时候要如何处理。

模拟场景

假设，当前我们需要开发一个程序来分析来自服务器的日志文件，并生成一份报告，列出前 10 个最常用的应用程序。

每天，都会生成一个新的日志文件，其中包含时间戳、主机信息、持续时间、服务调用等信息，以及可能与我们的特定方案无关的其他数据。

```
2024-02-25T00:00:00.000+GMT host7 492 products 0.0.3 PUT
73.182.150.152 eff0fac5-b997-40a3-87d8-02ff2f397b44
2024-02-25T00:00:00.016+GMT host6 123 logout 2.0.3 GET
34.235.76.94 8b97acae-dd36-4e83-b423-12905a4ab38d
2024-02-25T00:00:00.033+GMT host6 50 payments/:id 0.4.6 PUT
148.241.146.59 ac3c9064-4782-46d9-a0b6-69e4d55a5b38
2024-02-25T00:00:00.050+GMT host2 547 orders 1.5.0 PUT
6.232.116.248 2285a81e-c511-41b9-b0ea-a475a0a45805
2024-02-25T00:00:00.067+GMT host4 400 suggestions 0.8.6 DELETE
149.138.227.154 8031b639-700e-4a7c-b257-fcbed0d029ce
2024-02-25T00:00:00.084+GMT host2 644 login 6.90 GET
208.158.145.204 3906a28c-56e4-4e5f-b548-591eab737aa7
2024-02-25T00:00:00.101+GMT host5 339 suggestions 0.8.9 PUT
173.109.21.97 c7dfec8a-5ca8-4d0d-b903-aaf65629fdd0
2024-02-25T00:00:00.118+GMT host9 87 products 2.6.3 POST
220.252.90.140 e5ceef67-2f0f-4c2d-a6d2-c698598aaef2
2024-02-25T00:00:00.134+GMT host0 845 products 9.4.6 GET
136.79.178.188 f28578c1-c37c-47a3-a473-4e65371e0245
```

2024-02-25T00:00:00.151+GMT host4 675 login 0.89 DELETE 32.159.65.239 d27ff353-e501-43e6-bdce-680d79a07c36

```

我们的代码将收到日志文件列表，我们的目标是编制一份报告，列出最常用的 10 个服务。但是，要包含在报告中，服务必须在提供的每个日志文件中至少有一个条目。简而言之，一项服务必须每天使用才有资格包含在报告中。

## 基础实现

解决这个问题的最初方法是考虑业务需求并创建以下代码：

```
public void processFiles(final List<File> fileList) {
  final Map<LocalDate, List<LogLine>> fileContent = getFileContent(fileList);
  final List<String> serviceList = getServiceList(fileContent);
  final List<Statistics> statisticsList = getStatistics(fileContent, serviceList);
  final List<Statistics> topCalls = getTop10(statisticsList);

  print(topCalls);
}
```

该方法接收文件列表作为参数，核心流程如下：

* 创建一个包含每个文件条目的映射，其中Key是 LocalDate，Value是文件行列表。
* 使用所有文件中的唯一服务名称创建字符串列表。
* 生成所有服务的统计信息列表，将文件中的数据组织到结构化地图中。
* 筛选统计信息，获取排名前 10 的服务调用。
* 打印结果。

可以注意到，这种方法将太多数据加载到内存中，不可避免地会导致 `OutOfMemoryError`

## 改进实现
```

就如文章开头说的，我们需要采用另一种策略：逐行处理文件的模式。

```
private void processFiles(final List<File> fileList) {
  final Map<String, Counter> compiledMap = new HashMap<>();

  for (int i = 0; i < fileList.size(); i++) {
    processFile(fileList, compiledMap, i);
  }

  final List<Counter> topCalls =
      compiledMap.values().stream()
          .filter(Counter::allDaysSet)
.sorted(Comparator.comparing(Counter::getNumberOfCalls).reversed())
       .limit(10)
          .toList();

  print(topCalls);
}
```

* 首先，它声明一个Map（compiledMap），其中一个String作为键，代表服务名称，以及一个Counter对象（稍后解释），它将存储统计信息。
* 接下来，它逐一处理这些文件并相应地更新compileMap。
* 然后，它利用流功能来： 仅过滤具有全天数据的计数器；按调用次数排序；最后，检索前 10 名。

在看整个处理的核心`processFile`方法之前，我们先来分析一下`Counter`类，它在这个过程中也起到了至关重要的作用：

```
public class Counter {
  @Getter private String serviceName;
  @Getter private long numberOfCalls;
  private final BitSet daysWithCalls;

  public Counter(final String serviceName, final int numberOfDays) {
    this.serviceName = serviceName;
    this.numberOfCalls = 0L;
```

```
    daysWithCalls = new BitSet(numberOfDays);
  }

  public void add() {
    numberOfCalls++;
  }

  public void setDay(final int dayNumber) {
    daysWithCalls.set(dayNumber);
  }

  public boolean allDaysSet() {
    return daysWithCalls.stream()
        .mapToObj(index -> daysWithCalls.get(index))
        .reduce(Boolean.TRUE, Boolean::logicalAnd);
  }
}
```

* 它包含三个属性：serviceName、numberOfCalls 和 daysWithCalls
* numberOfCalls 属性通过 add 方法递增，该方法为 serviceName 的每个处理行调用。
* daysWithCalls 属性是一个 Java BitSet，一种用于存储布尔属性的内存高效结构。它使用要处理的天数进行初始化，每个位代表一天，初始化为 false。
* setDay 方法将 BitSet 中与给定日期位置相对应的位设置为 true。

allDaysSet 方法负责检查 BitSet 中的所有日期是否都设置为 true。它通过将 BitSet 转换为布尔流，然后使用逻辑 AND 运算符减少它来实现此目的。

```
private void processFile(final List<File> fileList,
                         final Map<String, Counter> compiledMap,
                         final int dayNumber) {
  try (Stream<String> lineStream =
Files.lines(fileList.get(dayNumber).toPath())) {
    lineStream
        .map(this::toLogLine)
        .forEach(
            logLine -> {
              Counter counter = compiledMap.get(logLine.serviceName());
              if (counter == null) {
                counter = new Counter(logLine.serviceName(), fileList.size());
                compiledMap.put(logLine.serviceName(), counter);
              }
```

```
                counter.add();
                counter.setDay(dayNumber);
              });

    } catch (final IOException e) {
      throw new RuntimeException(e);
    }
}

```

* 该过程使用Files类的lines方法逐行读取文件，并将其转换为流。这里的关键特征是lines方法是惰性的，这意味着它不会立即读取整个文件；相反，它会在流被消耗时读取文件。
* toLogLine 方法将每个字符串文件行转换为具有用于访问日志行信息的属性的对象。
* 处理文件行的主要过程比预期的要简单。它从与serviceName关联的compileMap中检索（或创建）Counter，然后调用Counter的add和setDay方法。

正如我们所看到的，在 Java 中处理大文件而不将整个文件加载到内存中并不是什么复杂的事情。 Files类提供了逐行处理文件的方法，我们还可以在文件处理过程中利用哈希来存储数据，这有助于节省内存。

> 欢迎我的公众号：程序猿DD。第一时间了解前沿行业消息、分享深度技术干货、获取优质学习资源

原文链接: https://juejin.cn/post/7351454390148284428