链滴

30G 上亿数据的超大文件，如何快速导入生产环境?

作者：9526xu

原文链接：https://ld246.com/article/1608770948014

来源网站：链滴

Hello，大家好，我是楼下小黑哥~

如果给你一个包含一亿行数据的超大文件，让你在一周之内将数据转化导入生产数据库，你会如何操？

上面的问题其实是小黑哥前端时间接到一个真实的业务需求，将一个老系统历史数据通过线下文件的式迁移到新的生产系统。

~~由于老板们已经敲定了新系统上线时间，所以只留给小黑哥一周的时间将历史数据导入生产系统。~~

由于时间紧，而数据量又超大，所以小黑哥设计的过程想到一下解决办法：

- 拆分文件
- 多线程导入

拆分文件

首先我们可以写个小程序，或者使用拆分命令 split 将这个超大文件拆分一个个小文件。

```
-- 将一个大文件拆分成若干个小文件，每个文件 100000 行
split -l 100000 largeFile.txt -d -a 4 smallFile_
```

这里之所以选择先将大文件拆分，主要考虑到两个原因：

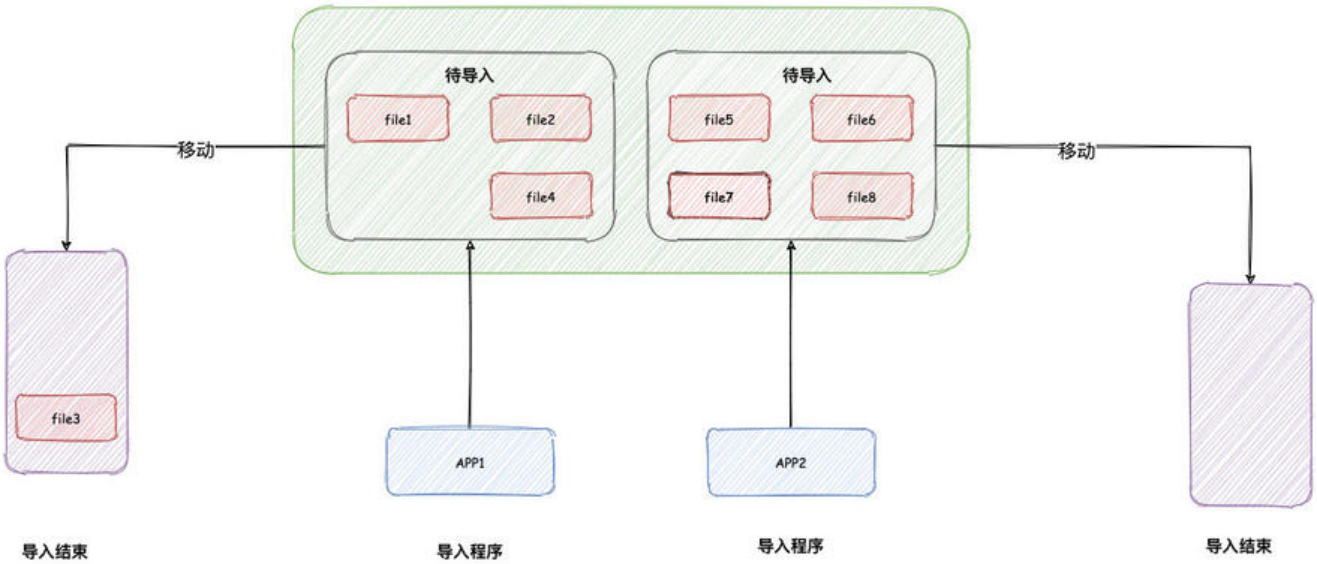
第一如果程序直接读取这个大文件，假设读取一半的时候，程序突然宕机，这样就会直接丢失文件读的进度，又需要重新开头读取。

而文件拆分之后，一旦小文件读取结束，我们可以将小文件移动一个指定文件夹。

这样即使应用程序宕机重启，我们重新读取时，只需要读取剩余的文件。

第二，一个文件，只能被一个应用程序读取，这样就限制了导入的速度。

而文件拆分之后，我们可以采用多节点部署的方式，水平扩展。每个节点读取一部分文件，这样就可成倍的加快导入速度。



原文链接：30G 上亿数据的超大文件，如何快速导入生产环境?

多线程导入

当我们拆分完文件，接着我们就需要读取文件内容，进行导入。

之前拆分的时候，设置每个小文件包含 10w 行的数据。由于担心一下子将 10w 数据读取应用中，导堆内存占用过高，引起频繁的 **Full GC**，所以下面采用流式读取的方式，一行一行的读取数据。

当然了，如果拆分之后文件很小，或者说应用的堆内存设置很大，我们可以直接将文件加载到应用内中处理。这样相对来说简单一点。

逐行读取的代码如下：

```
File file = ...
try (LineIterator iterator = IOUtils.lineIterator(new FileInputStream(file), "UTF-8")) {
    while (iterator.hasNext()) {
        String line=iterator.nextLine();
        convertToDB(line);
    }

}
```

上面代码使用 commons-io 中的 LineIterator类，这个类底层使用了 BufferedReader 读取文件内。它将其封装成迭代器模式，这样我们可以很方便的迭代读取。

如果当前使用 JDK1.8 ，那么上述操作更加简单，我们可以直接使用 JDK 原生的类 Files将文件转成 Stream 方式读取，代码如下：

```
Files.lines(Paths.get("文件路径"), Charset.defaultCharset()).forEach(line -> {
    convertToDB(line);
});
```

其实仔细看下 Files#lines底层源码，其实原理跟上面的 LineIterator类似，同样也是封装成迭代器模。

```
public Stream<String> lines() {
    Iterator<String> iter = new Iterator<String>() {
        String nextLine = null;

        @Override
        public boolean hasNext() {
            if (nextLine != null) {
                return true;
            } else {
                try {
                    nextLine = readLine();
                    return (nextLine != null);
                } catch (IOException e) {
                    throw new UncheckedIOException(e);
                }
            }
        }

        @Override
        public String next() {
            if (nextLine != null || hasNext()) {
                String line = nextLine;
                nextLine = null;
                return line;
            } else {
                throw new NoSuchElementException();
            }
        }
    };
    return StreamSupport.stream(Spliterators.spliteratorUnknownSize(
            iter, characteristics: Spliterator.ORDERED | Spliterator.NONNULL), parallel: false);
}
}
```

多线程的引入存在的问题

上述读取的代码写起来不难，但是存在效率问题，主要是因为只有单线程在导入，上一行数据导入完之后，才能继续操作下一行。

为了加快导入速度，那我们就多来几个线程，并发导入。

多线程我们自然将会使用线程池的方式，相关代码改造如下：

```
File file = ...;
ExecutorService executorService = new ThreadPoolExecutor(
        5,
        10,
        60,
        TimeUnit.MINUTES,
            // 文件数量，假设文件包含 10W 行
        new ArrayBlockingQueue<>(10*10000),
             // guava 提供
        new ThreadFactoryBuilder().setNameFormat("test-%d").build());
try (LineIterator iterator = IOUtils.lineIterator(new FileInputStream(file), "UTF-8")) {
    while (iterator.hasNext()) {
        String line = iterator.nextLine();
        executorService.submit(() -> {
            convertToDB(line);
        });
    }
```

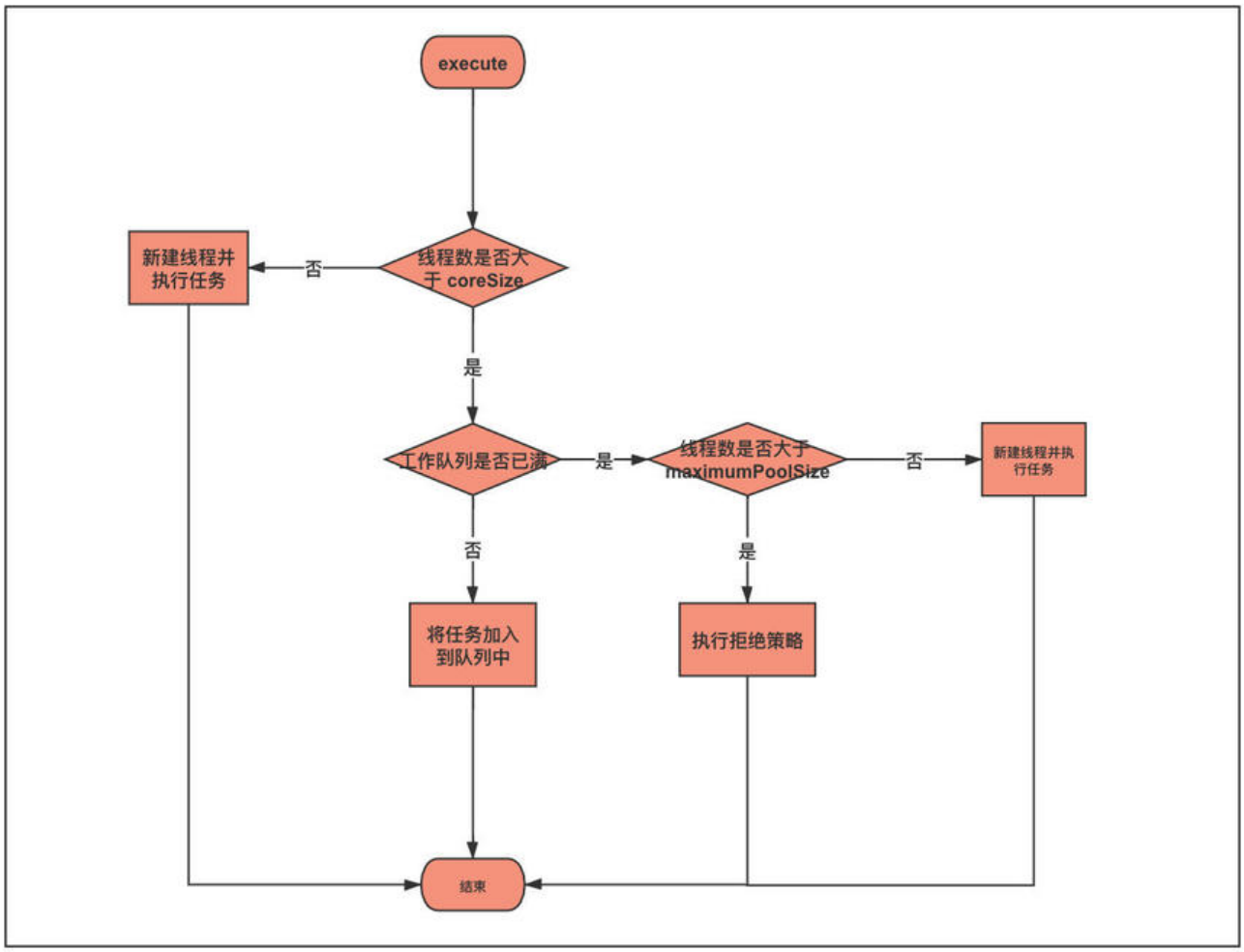
原文链接：30G 上亿数据的超大文件，如何快速导入生产环境?

```
}
```

上述代码中，每读取到一行内容，就会直接交给线程池来执行。

我们知道线程池原理如下：

1. 如果核心线程数未满，将会直接创建线程执行任务。
2. 如果核心线程数已满，将会把任务放入到队列中。
3. 如果队列已满，将会再创建线程执行任务。
4. 如果最大线程数已满，队列也已满，那么将会执行拒绝策略。



由于我们上述线程池设置的核心线程数为 5，很快就到达了最大核心线程数，后续任务只能被加入队
。

为了后续任务不被线程池拒绝，我们可以采用如下方案：

- 将队列容量设置成很大，包含整个文件所有行数
- 将最大线程数设置成很大，数量大于件所有行数

以上两种方案都存在同样的问题，第一种是相当于将文件所有内容加载到内存，将会占用过多内存。

而第二种创建过多的线程，同样也会占用过多内存。

一旦内存占用过多，GC 无法清理，就可能会引起频繁的 **Full GC**，甚至导致 **OOM**，导致程序导入度过慢。

解决这个问题，我们可以如下两种解决方案：

- CountDownLatch 批量执行
- 扩展线程池

CountDownLatch 批量执行

JDK 提供的 CountDownLatch，可以让主线程等待子线程都执行完成之后，再继续往下执行。

利用这个特性，我们可以改造多线程导入的代码,主体逻辑如下：

```
try (LineIterator iterator = IOUtils.lineIterator(new FileInputStream(file), "UTF-8")) {
    // 存储每个任务执行的行数
    List<String> lines = Lists.newArrayList();
    // 存储异步任务
    List<ConvertTask> tasks = Lists.newArrayList();
    while (iterator.hasNext()) {
        String line = iterator.nextLine();
        lines.add(line);
        // 设置每个线程执行的行数
        if (lines.size() == 1000) {
            // 新建异步任务，注意这里需要创建一个 List
            tasks.add(new ConvertTask(Lists.newArrayList(lines)));
            lines.clear();
        }
        if (tasks.size() == 10) {
            asyncBatchExecuteTask(tasks);
        }

    }
    // 文件读取结束，但是可能还存在未被内容
    tasks.add(new ConvertTask(Lists.newArrayList(lines)));
    // 最后再执行一次
    asyncBatchExecuteTask(tasks);
}
```

这段代码中，每个异步任务将会导入 1000 行数据，等积累了 10 个异步任务，然后将会调用 asyncBatchExecuteTask 使用线程池异步执行。

```
/**
 * 批量执行任务
 *
 * @param tasks
 */
private static void asyncBatchExecuteTask(List<ConvertTask> tasks) throws InterruptedException {
    CountDownLatch countDownLatch = new CountDownLatch(tasks.size());
    for (ConvertTask task : tasks) {
```

```
            task.setCountDownLatch(countDownLatch);
            executorService.submit(task);
        }
        // 主线程等待异步线程 countDownLatch 执行结束
        countDownLatch.await();
        // 清空，重新添加任务
        tasks.clear();
    }
```

asyncBatchExecuteTask 方法内将会创建 CountDownLatch，然后主线程内调用 await方法等待有异步线程执行结束。

ConvertTask 异步任务逻辑如下：

```
    /**
     * 异步任务
     * 等数据导入完成之后，一定要调用 countDownLatch.countDown()
     * 不然，这个主线程将会被阻塞，
     */
    private static class ConvertTask implements Runnable {

        private CountDownLatch countDownLatch;

        private List<String> lines;

        public ConvertTask(List<String> lines) {
            this.lines = lines;
        }

        public void setCountDownLatch(CountDownLatch countDownLatch) {
            this.countDownLatch = countDownLatch;
        }

        @Override
        public void run() {
            try {
                for (String line : lines) {
                    convertToDB(line);
                }
            } finally {
                countDownLatch.countDown();
            }
        }
    }
```

ConvertTask任务类逻辑就非常简单，遍历所有行，将其导入到数据库中。所有数据导入结束，调用 countDownLatch#countDown。

一旦所有异步线程执行结束，调用 countDownLatch#countDown，主线程将会被唤醒，继续执行件读取。

虽然这种方式解决上述问题，但是这种方式，每次都需要积累一定任务数才能开始异步执行所有任务。

另外每次都需要等待所有任务执行结束之后，才能开始下一批任务，批量执行消耗的时间等于最慢的步任务消耗的时间。

这种方式线程池中线程存在一定的闲置时间，那有没有办法一直压榨线程池，让它一直在干活呢？

扩展线程池

回到最开始的问题，文件读取导入，其实就是一个**生产者-消费者**消费模型。

主线程作为生产者不断读取文件，然后将其放置到队列中。

异步线程作为消费者不断从队列中读取内容，导入到数据库中。

一旦队列满载，生产者应该阻塞，直到消费者消费任务。

其实我们使用线程池的也是一个**生产者-消费者**消费模型，其也使用阻塞队列。

那为什么线程池在队列满载的时候，不发生阻塞？

这是因为线程池内部使用 offer 方法，这个方法在队列满载的时候**不会发生阻塞**，而是直接返回 。

那我们有没有办法在线程池队列满载的时候，阻塞主线程添加任务？

其实是可以的，我们自定义线程池拒绝策略，当队列满时改为调用 BlockingQueue.put 来实现生产的阻塞。

```
RejectedExecutionHandler rejectedExecutionHandler = new RejectedExecutionHandler() {
    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        if (!executor.isShutdown()) {
            try {
                executor.getQueue().put(r);
            } catch (InterruptedException e) {
                // should not be interrupted
            }
        }

    }
};
```

这样一旦线程池满载，主线程将会被阻塞。

使用这种方式之后，我们可以直接使用上面提到的多线程导入的代码。

```
ExecutorService executorService = new ThreadPoolExecutor(
        5,
        10,
        60,
        TimeUnit.MINUTES,
        new ArrayBlockingQueue<>(100),
        new ThreadFactoryBuilder().setNameFormat("test-%d").build(),
        (r, executor) -> {
            if (!executor.isShutdown()) {
                try {
                     // 主线程将会被阻塞
                    executor.getQueue().put(r);
                } catch (InterruptedException e) {
                    // should not be interrupted
                }
            }

        });
File file = new File("文件路径");

try (LineIterator iterator = IOUtils.lineIterator(new FileInputStream(file), "UTF-8")) {
    while (iterator.hasNext()) {
        String line = iterator.nextLine();
        executorService.submit(() -> convertToDB(line));
    }
}
```

小结

一个超大的文件，我们可以采用拆分文件的方式，将其拆分成多份文件，然后部署多个应用程序提高取速度。

另外读取过程我们还可以使用多线程的方式并发导入，不过我们需要注意线程池满载之后，将会拒绝续任务。

我们可以通过扩展线程池，自定义拒绝策略，使读取主线程阻塞。

好了，今天文章内容就到这里，不知道各位有没有其他更好的解决办法，欢迎留言讨论。