DOI:10.16186/j.cnki.1673-9787.2015.03.019

基于异步文件通道的Java Web多任务分块文件上传

陈 冈,夏火松
(武汉纺织大学 管理学院,武汉 430073)

摘要:针对目前多数基于同步阻塞式处理模式、采用Java字节码数据流方式进行读写操作的低效率处理技术,将异步文件通道与线程池结合,通过数据分块和读写处理的异步执行,构造一种基于非阻塞数据传输模式的文件上传处理架构,讨论了关键技术的实现方法。实际测试数据的结果表明:与常规方案相比,所提出的技术方案具有非常明显的效率优势,对于Java Web系统开发具有较好的应用参考价值。

关 键 词:异步文件通道;非阻塞;线程池;文件上传;数据块
中图分类号:TP311 **文献标志码**:A **文章编号**:1673-9787(2015)03-0400-06

Java Web file upload with multi-task division based on asynchronous file channel

CHEN Gang, XIA Huosong
(*School of Management, Wuhan Textile University, Wuhan* 430073, *China*)

Abstract: According to the low efficiency of present processing technologies, an asynchronous file channel was combined with a thread pool, a file upload processing architecture was designed through data block and implementations of asynchronous read and write processing, and key techniques was discussed, based on synchronous block processing mode and Java byte streams to read and write data operation. The actual test results show that this solution has very obvious efficiency advantages, compared with conventional solution. The achievements provide a good application reference for Java Web system developments.

Key words: asynchronous file channel; non-blocking; thread pool; file upload; data block

0 引 言

文件上传是Java Web系统中很常见的一种处理,一般通过客户端Form表单,基于HTTP的RFC1867上传规范实现文件数据的传输和解析[1-2]。RFC1867规范对于上传文件的数据结构形式定义包含四个部分:数据流开始符;文件头信息(包含Content-Disposition、filename、Content-Type等上传附加数据);文件内容;随机生成的数据流结束符。

目前,Java Web文件上传主要有两种方式。

(1)采用第三方上传组件[3-4],例如著名的Apache Commons FileUpload文件上传组件(以下简称方案1)。这种方式的优势:无需额外编写代码,使用方便,运行效率满足日常应用。其不足:代码非常复杂,效率提升困难,如果需要文件上传过程中进行干预处理,例如显示文件上传进度指示、或者在文件中嵌入一些特殊标记数据,则非常困难。

(2)自定义文件上传方式(以下简称方案

收稿日期:2014-12-12
基金项目:国家自然科学基金资助项目(71171153)
作者简介:陈冈(1970—),男,湖北英山人,副教授,主要从事信息管理与信息系统、电子商务教学与研究工作。
E-mail:acoease@126.com

2）。这种方式使用纯 Java 代码进行操作（非 Java 上传方式，例如使用 Flash 组件等，不在本文讨论范围），使用 File Output Stream、File、Random Access File 等相关输入输出流处理类，以字节数据为单位进行读写[5-8]。这种方式的优势在于可以对上传中间过程进行干预处理，应用更为灵活。其不足：需要用户自行编写处理代码，由于采用基于字节码的数据流处理方式，传输效率低下，效率提升困难。另外，该方案上传不同种类的文件时仍存在较多问题，具体情况如表 1 所示。

表 1　上传错误描述

Tab. 1　Upload error description

上传文件类型	错误现象
RAR	不可预料的压缩文件末端
DOC、XLS、PPT	无法读取此文档，文档可能已损坏
PDF	文件大小减少。有些可正常打开，有些则显示“文件已损坏且无法修复”
TXT	偶发性数据丢失
MP4	正常
SQL	偶发性数据丢失
JAVA	丢失若干行数据

对这两个方案作进一步研究，发现两者本质类似：采用同步阻塞式处理思想，基于字节码数据流读写。因此，方案 1 可以说是方案 2 的功能性封装。由于采用同步阻塞式数据流读写模式[9]，其上传效率很难得到改善，原因在上传文件触发读数据的操作时，应用程序会阻塞直到数据传输完成（或发生错误），然后数据被移动到用户空间缓冲区中[10]，如图 1 所示。当读操作调用返回时，应用程序解除阻塞，开始写操作。而写操作过程中又会一直阻塞，直到将用户缓冲空间的数据写入服务器指定位置。问题的核心在于阻塞使得读写操作之间存在系统内核时间的浪费，导致操作过程中无法处理其他工作，例如将已经读取的部分数据块写入文件，或者读取其他数据，这会导致整体数据吞吐量的降低。

由于方案 1 设计了复杂的读写数据缓冲调度机制，以牺牲代码的简洁性获得了性能提升，其上传效率、稳定性尚可，因而在实际中仍得到广泛使用。

因此，设计一种结构简单、上传效率良好、能够干预上传数据读写过程的方案，具有较好的现实应用意义。为了解决上述问题，本文研究了一种新的基于异步非阻塞、线程池、文件通道和数据分块等相结合的文件上传方案（以下简称方案 3）。该方案既能实现良好的上传效率，避免各类上传错误，又能够根据实际需要灵活对上传中间过程进行干预。

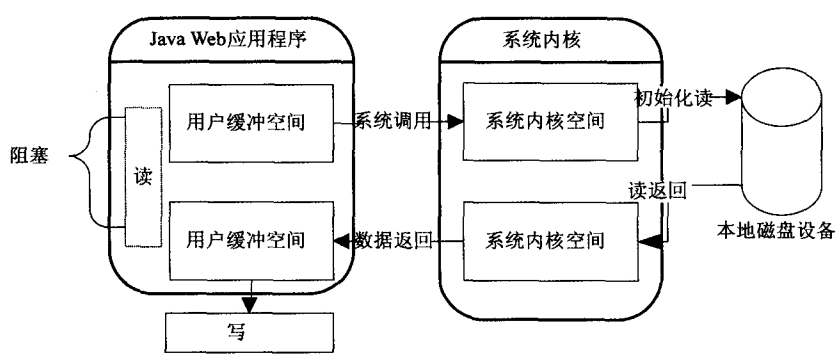

图 1　同步阻塞读写

Fig.1　Synchronous blocking read and write

1　异步通道式上传

1.1　异步非阻塞式读写

耗时较多的 I/O 数据流读写操作通常会阻止 Java Web 应用程序的主线程。而异步非阻塞式读写操作可以在执行占用大量资源的 I/O 数据流操作时，允许应用程序发起多个 I/O 数据流操作[11]，这时本地磁盘设备以非阻塞方式打开，系统不阻塞执行其他处理任务，也就是处理操作和 I/O 数据流操作可并行进行，如图 2 所示。

在同步情况下，任务线程必须一直等到 I/O 请求完成，此时程序被阻塞。而在异步环境中，处理 I/O 操作的线程将读数据请求提交给系统内核后，不要求读数据操作完成即可返回，继续去处理其他事务，例如对其他已经返回的数据进行写操作。当系统内核处理完读数据的任务后，反馈一个信号给线程，线程收到信号后，中断当前工作并

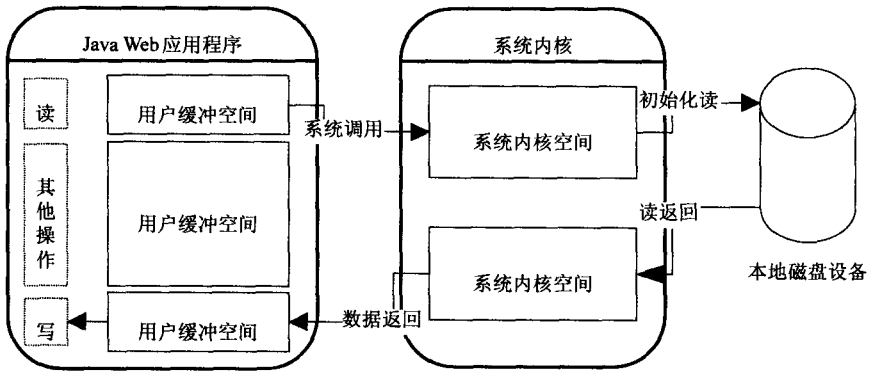

图 2 异步非阻塞读写

Fig.2 Asynchronous non-blocking read and write

处理读操作的结果数据。由于异步非阻塞模式并不需要等待响应结果,只需要在适当的时机去获得(例如操作系统反馈信号后),因此,在效率和吞吐量方面优于同步方式。一些操作系统(例如 Windows、Linux 等)已经在系统内核的底层为异步 I/O 操作提供了支持[11]。

这样一来,就可以将上传的文件数据流进行异步的分块读、分块写,提高处理效率。

1.2 处理架构

基于异步非阻塞读写基本原理,本文设计了如图 3 所示的处理架构。

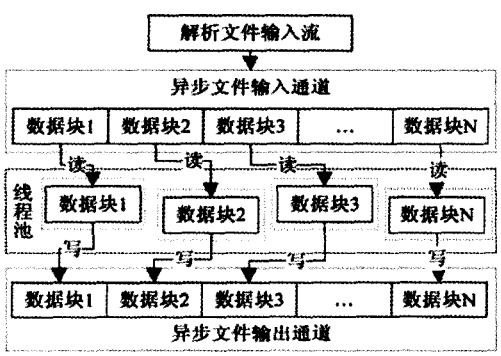

图 3 处理架构

Fig.3 Processing architecture

文件通道是 Java NIO 为大容量数据传输而设计,代表一种与磁盘设备或 I/O 读取操作的开放连接,提供了一个基于 Java I/O 类实现的底层数据传输机制,该机制将一些耗时操作转移给操作系统,有效地利用了操作系统管理文件和内存的模式[12-14]。

异步文件通道采用全双工数据传输模式操作,读写处理异步执行。具体处理步骤:(1)文件输入流载入文件通道;(2)将源文件数据分成若干数据块;(3)从线程池中获取某个空闲线程,读取数据块;(4)从线程池中获取某个空闲线程,写数据块到文件输出通道;(5)依此读写其他数据块。由于是非阻塞式处理,系统读写操作处理与磁盘设备的 I/O 操作重叠进行,因此,下一个数据块的读写并不需要前面数据块读写操作的完成。

1.3 效率比较

为了检验方案 3 的实际效果,需要对上述三种文件上传方案进行实际上传的对比测试。选取的测试环境如下。

中央处理器:AMD Athlon 64 X2 2.7 GHz。

内存:金士顿 DDR2 800 4G。

硬盘:西部数据 WDC WD32 320G 7200 r/min。

操作系统:32 位 Windows7 旗舰版。

软件平台:JDK7、Tomcat7、Spring Tool Suite 3.6、commons-fileupload-1.2.2。

待上传文件大小分别为 1.01,5.45,49.2,100.1,371,525 MB。各方案的每种文件各测试 10 次,记录其所用时间,取其平均值作为结果。测试时未考虑方案 3 相关参数的调整情况,将数据块大小固定为 64 KB,核心线程数为 2,最大线程数为 4,任务等待队列为非阻塞无锁队列。上传测试结果如表 2 所示。

表 2 文件上传测试结果

Tab.2 Test results of file upload /ms

方案名	文件大小/MB					
	1.01	5.45	49.2	100.1	371	525
1	80.6	422.2	4 294.2	8 435.5	73 338.6	107 621.3
2	86.8	522.2	5 615.8	11 963.7	88 178.6	110 708.6
3	87.6	388.6	3 370.1	6 839.9	53 723.5	84 230.3

为了更直观地分析测试结果,表 2 数据对应的折线图如图 4 所示。

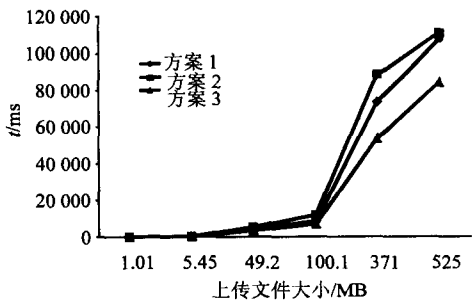

图 4　文件大小与所耗时间的关系

Fig.4　Relationship between file size and time

从表 2 和图 4 可以得出结论：(1)在上传文件较小的情况下，三种方案差别不大；(2)随着上传文件的增大，方案 3 的优势越来越明显；(3)当进行大文件上传时，如上传 100.1，371，525 MB 时，方案 3 比方案 1 上传速度分别提升约18.9%，26.75%，21.73%，比方案 2 分别提升约42.82%，39.07%，23.92%，效率优势非常明显；(4)在上传大文件时，方案 1 相对方案 2 不再具有明显优势，两者差距微小，这印证了前面两者本质相同的结论，也说明在同步阻塞式架构下，方案 1 单纯依靠读写数据缓冲难以得到有效改善；(5)根据实际应用需求情况，调整方案 3 的数据块大小、核心线程数、最大线程数、任务等待队列类型等，效率仍有进一步优化的空间。

上述测试并未考虑方案 3 各参数对上传效率的影响，为了更全面考察方案 3 处理架构的效率，需要进一步测试方案 3 在不同的数据块大小、核心线程数、任务等待队列类型等情况下，其上传效率的差异。首先测试数据块大小对上传效率的影响。从图 4 可以分析出，方案 3 在上传大文件时效率优势明显，所以将待测试文件大小确定为 371 MB。然后将核心线程数固定为 2，数据块大小依次定义：4，16，32，64，128，256，512，1024 KB，分别测试 LinkedTransferQueue，SynchronousQueue，LinkedBlockingQueue，ArrayBlockingQueue 这四种任务等待队列在不同大小数据块下的上传效率。测试结果如图 5 所示。

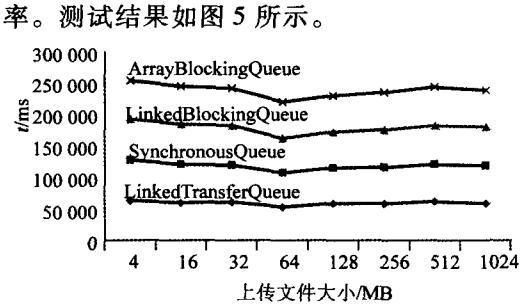

图 5　不同大小数据块下的测试结果

Fig.5　Test results under different data blocks size

对上面的测试结果进行分析，可以得出：(1)过大或过小的数据块，都对上传效率造成不良影响(过小的数据块，将导致任务过多，影响效率；过大的数据块，则容易造成队列中的任务等待，造成时间上的浪费)；(2)四种队列均在 64 KB 大小的数据块时，上传效率最优。

根据上面的测试结果，接下来测试四种队列在 64 KB 大小数据块时，核心线程数对上传效率的影响。测试时，队列容量大小设置原则：核心线程数 +1。测试结果如图 6 所示。

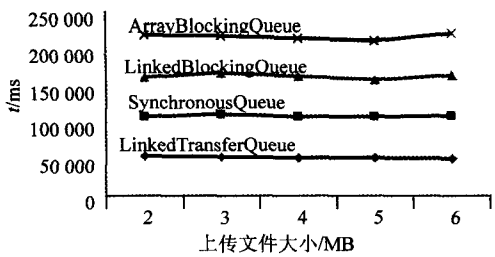

图 6　不同核心线程数下的测试结果

Fig.6　Test results under different core thread numbers

对图 6 的测试结果进行分析，可以得出如下结论。

(1) ArrayBlockingQueue 队列在核心线程数为 3、4 时效率较好，过小或过大的核心线程数都会降低效率。ArrayBlockingQueue 是一种基于数组的并发阻塞有界队列，按 FIFO(先进先出)原则对元素进行排序。ArrayBlockingQueue 有助于防止资源耗尽，过大或过小的队列容量，都容易导致系统吞吐量的降低。

(2) LinkedBlockingQueue 队列在核心线程数为 5 时效率最佳，其他情况则较为均衡。LinkedBlockingQueue 是一种基于链接节点的无界队列，按 FIFO 排序元素。由于需要动态地创建链接节点，因此，若频繁地插入队列元素将对性能有些许影响。另外，队列过度扩展可能会导致系统资源耗尽。

(3) SynchronousQueue 队列在核心线程数为 2 时效率最佳。SynchronousQueue 内部并没有数据缓存空间，而是将任务直接提交给线程。队列元素的插入操作需要等待另一个线程的移除操作，队列元素快速地在插入线程移与移除线程之间切换进行。因此，核心线程数对效率的影响很小。

(4) LinkedTransferQueue 队列随着线程数的增加，效率不断得到提升，在核心线程数为 6 时效率最佳。LinkedTransferQueue 由于使用非阻塞操

作,可有效避免序列化瓶颈,有效地实现了元素在线程之间的传递。LinkedTransferQueue 比较适合多线程时的任务处理,大多数参数设置得当的处理情况下,LinkedTransferQueue 性能要强于其它三种队列[15]。

测试表明,处理架构较好地吻合了对文件进行异步多任务分块上传的设计要求。本文后续的关键技术实现,以 LinkedTransferQueue 队列为例进行说明。

2 关键技术实现

2.1 文件解析

文件上传的处理过程就是获取上传数据流 ServletInput Stream,解析该输入数据流,去掉上传时附加的数据流开始符、文件头信息和数据流结束符,然后写入服务器指定文件的过程。解析的关键是首先要获得上传文件在数据流中的起始、结束偏移量。

随机访问通道 SeekableByteChannel 用于通过字节通道随机访问文件数据。既可顺序访问,也可回溯处理,非常适合用来获得文件数据的起始、结束偏移量。

```
// 获取正文起始偏移量
final Callable < Long > worker = new Callable
< Long > ( ) {
@ Override
public Long call( ) throws Exception {
long beginPointer = 0;
int i = 0;
reader. position(0);// 定位到文件头
ByteBuffer buffer = ByteBuffer. allocate(1);
while (i < 2 && reader. read(buffer) > 0)
{// 忽略前 3 行的附加信息
beginPointer = reader. position( );
if (buffer. get(0) = = '\n') {
i + +;
}
buffer. clear( );
}
return beginPointer;
}
};
```

只需要通过换行符"\n"判断并跳过前三行数据,就很容易定位到文件数据的起始偏移量。对于文件结束偏移量,仍然利用 SeekableByteChannel 先定位到文件末尾,然后回溯到倒数第二行,即跳过尾部数据流结束符即可。

2.2 缓冲读写

根据前文设计,数据是分块读写的,因此,需要根据当前偏移量和需要读取的数据块大小进行操作。Callable 用于在线程中产生结果,常用来生成大量生命周期较短的线程来执行单一任务,而 Future 则用来获得具体的结果数据,例如此处读取的数据块数据。Callable 和 Future 是异步执行的,这有利于效率的提升。

读数据块操作提交到线程池后,通过异步文件通道的 read( )、write( )方法读写数据,由线程池中的工作线程将读写操作结果通过 Future 进行反馈。通过判断 Future 的状态,用户还可以跟踪读写进度状态并处理其他任务(例如推进上传任务进度量的变化)。与前面的文件解析不同,没有使用 ByteBuffer. allocate( )方法分配内存区,这是因为其产生的内存开销是基于 JVM 的,大数据量时可能导致内存溢出,关键是因为频繁的读数据块操作,由系统内存拷贝到 JVM 内存的时间开销累计数会比较大,特别是在大文件处理情况下。而 ByteBuffer. allocateDirect ( )产生的内存开销基于操作系统,省去内存拷贝这一操作环节,效率有所提高。

缓冲写时,只需要重写线程任务 Callable 的 call( )方法。

2.3 多线程异步上传

异步文件通道 AsynchronousFileChannel 是线程安全的抽象类,可以同时进行读写操作。JSR-166 规范提供了一个 ExecutorService 接口,将线程启动工厂建模为一个能够集中控制的服务,应用于并发多线程场景,以便根据处理需要生成线程并负责调度任务处理。当调用 shutdown( )方法后,ExecutorService 将阻塞其他任务,执行已经提交(submit)的全部任务。当任务完成后,ExecutorService 终止。

读写操作时将 AsynchronousFileChannel 与 ExecutorService 关联起来。与目前常规文件上传处理不同,创建 ExecutorService 对象时使用了 LinkedTransferQueue 队列。该队列按照 FIFO(先进先出)顺序处理元素。LinkedTransferQueue 基于利用底层 CPU 命令实现的乐观锁机制,使用 CAS(Compare and Swap)无锁算法,减少了并发时冲突的概率,可有效避免数据序列化瓶颈。LinkedTransferQueue 的非阻塞处理模式,可有效

提升数据块的读写处理效率。

读数据块、写数据块被设计成两个任务，各自推送到线程池中，由 ExecutorService 自动调度任务的完成。

3 结　语

综上论述，可以得出结论：(1)非阻塞异步文件通道的文件上传方式，较常规方法具有更好的数据传输效率；(2)非阻塞异步文件通道的四种任务等待队列中，SynchronousQueue 队列传输效率较为均衡，而 LinkedTransferQueue 队列效率最优；(3)过大或过小的线程数、数据块，都会对传输效率产生不良影响。

因此，实际应用时，应结合服务器资源和应用中的用户文件上传量的分布情况，通过调整异步处理中的核心线程、最大线程数和数据块大小，合理选择排序队列类型，以获取与应用场景相匹配的最佳方案。

参考文献：

[1] 陈晓华，田刚. 关于 JSP 无组件文件上传的研究与实现[J]. 计算机与现代化，2011(3)：132-133.

[2] 白鹤，吕红亮，王劲林. 进度显示的大文件上传组件的设计与实现[J]. 计算机工程与应用，2009(5)：91-94.

[3] 欧阳谦，蒋泽军，王丽芳. 基于 Struts 框架的多文件上传组件设计和研究[J]. 计算机工程与科学，2009，31(1)：11-13.

[4] 曹渠江，陈真. Struts2 框架整合 Spring 框架在文件上传下载中的应用[J]. 上海理工大学学报，2009，31(2)：169-172.

[5] 廖福保，张文梅. 基于 Java 技术的多文件上传和存取的研究及实现[J]. 计算机工程与设计，2008，29(11)：5665-5667.

[6] 张书锋. 基于 Java 语言的文件上传组件研究[J]. 电脑知识与技术，2013(9)：7475-7478.

[7] 王文龙，王武魁. 利用 Java 语言实现文件上传功能[J]. 微计算机信息，2007，23(11)：169-171.

[8] 陈冈. 基于 Comet 属性同步的 JavaWeb 实时进度条研究[J]. 现代计算机(专业版)，2014(4)：66-69.

[9] 袁劲松，马旭东. 基于阻塞与非阻塞 I/O 网络模型的 Java 语言实现[J]. 计算机系统应用，2008(9)：98-101.

[10] TIMJONES M. Boost application performance using asynchronousI/O [EB/OL]. http://www.ibm.com/developerworks/library/l-async，2006-09-28.

[11] AnghelLeonard. Java7NIO. 2 [M]. [S. l.]: Apress，2011.

[12] 丛凤侠，杨玉强. 基于 MINA 框架的高性能短信猫服务平台设计[J]. 计算机技术与发展，2013(4)：213-216.

[13] 叶柏龙，刘蓬. 基于 NIO 框架的 TeeTime 信息平台的设计与实现[J]. 计算机光盘软件与应用，2013(3)：203-205.

[14] 余晓彬，韩国强. 基于 NIO 和设计模式的应用服务器的设计与实现[J]. 微计算机信息，2009(21)：91-93.

[15] ALEXMILIER. Java7TransferQueue[EB/OL]. http://tech.puredanger.com/2009/02/28/java-7-transfer-queue，2009-02-28.

(责任编辑　胡圣武)