# spring boot 文件上传接口并发性能调优

## 前言

在一个项目现场，文件上传接口(文件500K)QPS只有30，这个并发性能确实堪忧。此文记录出坑过程。

## 问题一、InputStream按字节读取效率低

```
// 读取上传的文件
Part part = request.getPart("data");
InputStream in = part.getInputStream();

ByteArrayOutputStream str=new ByteArrayOutputStream();
int k;
byte[] file = null;
while((k=in.read())!=-1){
    str.write(k);
}
file = str.toByteArray();
str.close();
in.close();
```

```
/**
 * Reads the next byte of data from the input stream. The value byte is
 * returned as an {@code int} in the range {@code 0} to
 * {@code 255}. If no byte is available because the end of the stream
 * has been reached, the value {@code -1} is returned. This method
 * blocks until input data is available, the end of the stream is detected,
 * or an exception is thrown.
 *
 * <p> A subclass must provide an implementation of this method.
 *
```

```
 * @return      the next byte of data, or {@code -1} if the end of the
 *              stream is reached.
 * @throws      IOException  if an I/O error occurs.
 */
public abstract int read() throws IOException;

```

直接调用接口发现接口响应确实比较慢，经过排查是上述代码`in.read()`按字节读取效率特别低。既然定位到问题了，换个方式，每次读取8K数据。

```
byte[] buffer = new byte[8192];
int bytesRead;
while ((bytesRead = in.read(buffer)) != -1) {
    str.write(buffer, 0, bytesRead);
}

```

```
/**
 * Reads some number of bytes from the input stream and stores them into
 * the buffer array <code>b</code>. The number of bytes actually read is
 * returned as an integer.  This method blocks until input data is
 * available, end of file is detected, or an exception is thrown.
 *
 * <p> If the length of <code>b</code> is zero, then no bytes are read and
 * <code>0</code> is returned; otherwise, there is an attempt to read at
 * least one byte. If no byte is available because the stream is at the
 * end of the file, the value <code>-1</code> is returned; otherwise, at
 * least one byte is read and stored into <code>b</code>.
 *
 * <p> The first byte read is stored into element <code>b[0]</code>, the
 * next one into <code>b[1]</code>, and so on. The number of bytes read is,
 * at most, equal to the length of <code>b</code>. Let <i>k</i> be the
 * number of bytes actually read; these bytes will be stored in elements
 * <code>b[0]</code> through <code>b[</code><i>k</i><code>-1]</code>,
 * leaving elements <code>b[</code><i>k</i><code>]</code> through
 * <code>b[b.length-1]</code> unaffected.
```

```
     *
     * <p> The <code>read(b)</code> method for class
    <code>InputStream</code>
     * has the same effect as: <pre><code> read(b, 0, b.length)
    </code></pre>
     *
     * @param      b   the buffer into which the data is read.
     * @return     the total number of bytes read into the buffer, or
     *             <code>-1</code> if there is no more data because the end
    of
     *             the stream has been reached.
     * @exception  IOException  If the first byte cannot be read for any
    reason
     * other than the end of the file, if the input stream has been closed, or
     * if some other I/O error occurs.
     * @exception  NullPointerException  if <code>b</code> is
    <code>null</code>.
     * @see        java.io.InputStream#read(byte[], int, int)
     */
    public int read(byte b[]) throws IOException {
        return read(b, 0, b.length);
    }
```

```

> 如果JDK>=9,可以使用`readAllBytes`方法，更为便捷。内部实现其实也是按照8K进行读取的。

> 文件上传接口通常仅对业务逻辑做处理，文件存储往往会调用专门的存储服务。有2种处理思路：1、接收到完整文件数据，存储至内存中，然后调用存储接口；2、用流的方式，一边read ServletRequest#InputStream，一边write到存储服务的Stream中。个人认为方式2更合理，节约内存。

## 问题二、tomcat暂存性能瓶颈

接口采用`multipart/form-data`方式上传文件，tomcat接收到请求后会将请求内容暂存至本地磁盘，目录通常位于tomcat basedir目录下，比如我本地路径为`{basedir}\work\Tomcat\localhost\ROOT`。受限于磁盘写入速率瓶颈，限制了接口性能上限。
```

> 机械硬盘写入速率预估100MB/s，则在千兆组网场景不存在性能瓶颈，如果是固态硬盘，则写入速率更高。所以此项配置在2G以上组网才需考虑配置。

修改方法为修改sizeThreshold,`默认值为0`。如下所示修改为`1MB`,即内容大于1MB才存入磁盘，小于直接存入内存。

> 关于sizeThreshold，catalina包中处理逻辑为：如果对servlet做了配置，会使用配置的值。如果未配置，默认值为0。util包中DiskFileItemFactory默认值为10k。

```
servlet:
  multipart:
    file-size-threshold: 1MB
```

Tomcat中的相关处理逻辑,`parseRequest`方法按照`RFC 1867`规范对request进行处理。

```
// org.apache.tomcat.util.http.fileupload.disk.DiskFileItemFactory.java

/**
 * <p>The default {@link org.apache.tomcat.util.http.fileupload.FileItemFactory}
 * implementation. This implementation creates
 * {@link org.apache.tomcat.util.http.fileupload.FileItem} instances which keep
 * their
 * content either in memory, for smaller items, or in a temporary file on disk,
 * for larger items. The size threshold, above which content will be stored on
 * disk, is configurable, as is the directory in which temporary files will be
```

```
 * created.</p>
 *
 * <p>If not otherwise configured, the default configuration values are as
 * follows:</p>
 * <ul>
 *   <li>Size threshold is 10 KiB.</li>
 *   <li>Repository is the system default temp directory, as returned by
 *       {@code System.getProperty("java.io.tmpdir")}.</li>
 * </ul>
 * <p>
 * <b>NOTE</b>: Files are created in the system default temp directory
with
 * predictable names. This means that a local attacker with write access
to that
 * directory can perform a TOUTOC attack to replace any uploaded file
with a
 * file of the attackers choice. The implications of this will depend on
how the
 * uploaded file is used but could be significant. When using this
 * implementation in an environment with local, untrusted users,
 * {@link #setRepository(File)} MUST be used to configure a repository
location
 * that is not publicly writable. In a Servlet container the location
identified
 * by the ServletContext attribute {@code javax.servlet.context.tempdir}
 * may be used.
 * </p>
 *
 * <p>Temporary files, which are created for file items, will be deleted
when
 * the associated request is recycled.</p>
 *
 * @since FileUpload 1.1
 */
public class DiskFileItemFactory implements FileItemFactory {

    // ----------------------------------------------------------
Manifest constants

    /**
     * The default threshold above which uploads will be stored on disk.
     */
    public static final int DEFAULT_SIZE_THRESHOLD = 10240;
}

```

```
// org.apache.tomcat.util.http.fileupload.disk.DiskFileItem.java

/**
 * The threshold above which uploads will be stored on disk.
 */
private final int sizeThreshold;

/**
 * Returns an {@link java.io.OutputStream OutputStream} that can
 * be used for storing the contents of the file.
 *
 * @return An {@link java.io.OutputStream OutputStream} that can be
used
 *         for storing the contents of the file.
 *
 */
@Override
public OutputStream getOutputStream() {
    if (dfos == null) {
        final File outputFile = getTempFile();
        dfos = new DeferredFileOutputStream(sizeThreshold, outputFile);
    }
    return dfos;
}

```


```
// org.apache.tomcat.util.http.fileupload.DeferredFileOutputStream.java

/**
 * An output stream which will retain data in memory until a specified
 * threshold is reached, and only then commit it to disk. If the stream is
 * closed before the threshold is reached, the data will not be written to
 * disk at all.
 * <p>
 * This class originated in FileUpload processing. In this use case, you
do
 * not know in advance the size of the file being uploaded. If the file is
small
 * you want to store it in memory (for speed), but if the file is large you
want
 * to store it to file (to avoid memory issues).
 */
public class DeferredFileOutputStream
    extends ThresholdingOutputStream
```

```
{
    /**
     * Constructs an instance of this class which will trigger an event at
the
     * specified threshold, and save data to a file beyond that point.
     * The initial buffer size will default to 1024 bytes which is
ByteArrayOutputStream's default buffer size.
     *
     * @param threshold  The number of bytes at which to trigger an
event.
     * @param outputFile The file to which data is saved beyond the
threshold.
     */
    public DeferredFileOutputStream(final int threshold, final File
outputFile)
    {
        this(threshold,  outputFile, null, null, null,
ByteArrayOutputStream.DEFAULT_SIZE);
    }
}
```

问题三、网络带宽瓶颈
==========

对于常规企业内部应用，局域网环境下，至少能提供稳定的千兆带宽，常规业务接口不存在网络带宽瓶颈。但是对于文件上传接口而言，即使是小文件上传，接口并发高的场景带宽消耗依然较大，可能是性能瓶颈。

以千兆带宽为例，理论最大上传速率=1000Mbps÷8=125MB/s理论最大上传速率=1000Mbps÷8=125MB/s理论最大上传速率=1000Mbps÷8=125MB/s,实际场景很难达到理论最大速率，按照100MB/s预估。
`500K:200QPS，1M:100QPS，2M:50QPS`

问题解决思路整理
========

```
sequenceDiagram
client->>nginx: 1
nginx->>+tomcat: 2
tomcat->>+webserver: 3
```

Note right of webserver: 业务代码处理耗时，可通过controller打印日志监控耗时
webserver-->>-tomcat: 4
tomcat-->>-nginx: 5
nginx-->>client: 6

```

* client

指请求接口的客户端
* nginx

作为反向代理服务器
* tomcat

web容器
* webserver

web服务，比如springboot项目

排查过程可以根据由外向内层层递进的方式进行排查，当然也可采用经验判断法，对最有可能出现性能瓶颈的webserver进行排查。

1. 复现问题，在高负载场景请求接口复现问题或者使用Jmeter等工作做并发压力测试。复现问题是解决问题的基础。
2. 查看接口请求耗时，对耗时结构进行分析，比如Wating(TTFB)、Content Download耗时长，。比如Content Download耗时长，那就会首先怀疑带宽。
3. nginx性能较高，出现瓶颈概率低。可通过查看nginx访问日志，对比接口总耗时，如果耗时差异较大，就需要排查nginx本身性能、nginx与tomcat之间网络。
4. tomcat作为主流的web容器，影响性能的配置主要是maxThreads、maxConnections、堆内存、垃圾回收。对于成熟的应用开发团队，会有相对合理的初始配置。可通过查看tomcat访问日志，对比webserver接口耗时,如果耗时差异较大，就需要排查tomcat自身性能问题。
5. webserver中的业务处理逻辑，通常是接口总耗时占比最高的。优先在controller入口和出口记录日志，计算controller总耗时。如果确定是业务逻辑耗时长，再层层递进排查缩小范围，找到罪魁祸首。

测试性能汇总
======

测试环境
----
```

* 服务器主机、客户机

测试环境所限，服务器主机、客户机使用同一台开发主机。操作系统：windows10,CPU：Intel(R) Xeon(R) Gold 6242R CPU @ 3.10GHz，内存16G
* 磁盘

RND512KQ1T1 Read1219.86Mb/s Write44.88Mb/s
* Jmeter

400线程，60s拉起全部线程
* tomcat

tomcat9，做了如下配置

```
tomcat:
    threads:
        max: 400
    max-connections: 10000
    accept-count: 1000
```
* jar启动参数

配置了初始堆内存
`java -Dfile.encoding=UTF-8 -jar .\xxx.jar -server -Xms4096m -Xmx9000m`

测试结果
---

| 类型 | 平均响应时间 ms | 吞吐量/s |
| --- | --- | --- |
| 原始状态 | 22081 | 0.18 |
| 优化Byte[] | 3966 | 89 |
| 优化file-size-threshold | 1203 | 265 |
| 基准-（form-data） | 1279 | 279 |
| 基准-（优化file-size-threshold） | 109 | 2930 |
| 基准-空接口 | 28 | 12401 |

> **原始状态**：现场报性能问题时的版本，性能太过炸裂，Jmeter线程数调整为4，测试上传文件5KB
>
> **优化Byte[]**：优化了从stream读取存入优化Byte[]方法，测试上传文件5KB。此时网络吞吐量45MB/s，生产环境服务器配置性能至少比当前测试机器高2倍，接口性能至少提高1倍，对于千兆组网场景无须进一步优化，并发瓶颈是网络带宽
>
> **优化file-size-threshold**：优化为>1MB文件才存入磁盘，测试场景文件全部读入内存，测试上传文件5KB。此时网络吞吐量已大于100MB/s
>
> **基准-（form-data）**：form-data配置简单key参数，不上传文件，服务端接口直接返回简单字符串。相当于默认情况下form-data参数类型接口的性能基准，性能瓶颈是磁盘写入速率
>
> **基准-（优化file-size-threshold）**：form-data配置简单key参数，不上传文件，服务端接口直接返回简单字符串，优化为>1MB文件才存入磁盘。可以对比看出磁盘与内存的速率差异
>
> **基准-空接口**：普通的get无参接口，直接返回“hello”，作为当前配置环境下，tomcat接口性能极限

现场问题处理方案
========

经过定位现场性能瓶颈是`网络`。现场采用分布式架构，客户端、服务端部署多个节点，客户端通过本地回环地址调用服务端，降低网络压力。

原架构
---

```
classDiagram
存储服务 <|-- Server0
Server0 <|-- Server1
Server0 <|-- Server2
Server0 <|-- Server3
存储服务:+获取上传地址()
存储服务:+上传文件()
class Server0{
```

```
+服务端
+获取上传地址()
+上传文件()
}
class Server1{
-客户端
}
class Server2{
-客户端
}
class Server3{
-客户端
}

```

新架构
---

```
classDiagram
存储服务 <|-- Server0
存储服务 <|-- Server1
存储服务 <|-- Server2
存储服务 <|-- Server3
存储服务:+获取上传地址()
存储服务:+上传文件()
class Server0{
+服务端
-客户端
+获取上传地址()
+上传文件()
}
class Server1{
+服务端
-客户端
+获取上传地址()
+上传文件()
}
class Server2{
+服务端
-客户端
+获取上传地址()
+上传文件()
}
class Server3{
```

+服务端
-客户端
+获取上传地址()
+上传文件()
}

```

参考资料
====

tomcat 中是怎么处理文件上传的?

[Apache Tomcat 9 Configuration Reference](http://cxyroad.com/”https://tomcat.apache.org/tomcat-9.0-doc/config/http.html”)

[4-高并发运行环境优化-Tomcat](http://cxyroad.com/”https://zhuanlan.zhihu.com/p/672751242”)
```