计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2022,31(7):143−148 [doi: 10.15888/j.cnki.csa.008583]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于自适应分片的大文件快速上传①

周煜莹, 崔岩松, 王丹志, 陈科良

(北京邮电大学 电子工程学院, 北京 100876)
通信作者: 崔岩松, E-mail: cuiys@bupt.edu.cn

摘　要: 随着互联网技术的不断发展, 通过网络 Web 进行文件的上传拥有越来越多的应用需求. 其中, 在大容量文件的上传中, 常常因资源过大导致带宽资源紧张、浏览器崩溃或加载超时等问题, 大大降低了用户体验. 针对大文件上传的众多限制问题, 本文设计并实现了基于 Node.js 的大文件上传系统, 采用自适应分片结合并发上传的方法, 有效地缩短了大文件上传时间. 同时结合 element-ui 框架, 利用进度条实时展示上传进度, 具备良好的交互性能.

关键词: 自适应分片; 并发; 大文件上传; Node.js

引用格式: 周煜莹,崔岩松,王丹志,陈科良.基于自适应分片的大文件快速上传.计算机系统应用,2022,31(7):143–148. http://www.c-s-a.org.cn/1003-3254/8583.html

Fast Upload of Large Files Based on Adaptive Slicing

ZHOU Yu-Ying, CUI Yan-Song, WANG Dan-Zhi, CHEN Ke-Liang

(School of Electronic Engineering, Beijing University of Posts and Telecommunications, Beijing 100876, China)

Abstract: As Internet technology continues to evolve, the application demand for file uploads via the Web is on the rise. Nevertheless, large file uploads are often faced with bandwidth resource constraints, browser crashes, or loading timeouts due to excessive resources, which greatly reduces user experience. In view of the many limitations on large file uploads, this study designs and implements a large file upload system based on Node.js, combining adaptive slicing with concurrent upload to effectively shorten the upload time of such files. It also integrates the element-ui framework to show the upload progress in real time with a progress bar, and hence comes the favorable interactive performance.

Key words: adaptive slicing; concurrency; large file uploads; Node.js

在计算机网络技术迅猛发展的今天, 文件的上传是一个重要的应用交互场景. 对于普通的图片或 word 文档等几十 KB 或者几 MB 的文件上传, 使用 Web 的组件即可完成流畅的上传功能. 通常的文件上传是一次性获取整个文件上传, 传输过程简单: 第 1 步获取本地文件, 第 2 步将文件转换成字节传输, 第 3 步后端接收按顺序接收字节到内存中, 最后将接收完的字节保存为文件[1]. 但在大文件传输的应用中, 例如在邮箱管理系统中上传大容量的资源压缩包, 在网络视频发布系统中上传视频文件, 在线制作电子相册时需要上传高清图片, 网络硬盘服务系统, 局域网文件交换系统等[2]. 这些业务场景下的大文件传输很容易占据较大的带宽资源, 造成网页访问速度降低, 也可能导致后端服务器响应超时, 前端页面长时间无响应, 甚至卡顿而导致页面崩溃. 即便能够上传成功, 用户需要较长的等待时间, 在此期间不能刷新页面, 只能等待请求完成. 这些问题严重降低了用户体验, 因而大文件上传一直是 Web 应用系统的一大痛点.

针对以上问题, 本文基于 Node.js 边读边写的流模式传输, 采用 HTML5 的 File API 对上传的大文件进行分片处理, 通过上传速率动态调整分片大小, 同时充分利用带宽, 结合多并发上传进一步缩短上传时间, 在服务端

① 收稿时间: 2021-09-05; 修改时间: 2021-09-29, 2021-10-08, 2021-10-19; 采用时间: 2021-11-26; csa 在线出版时间: 2022-05-30

检验所有分片文件上传完整后, 再进行文件的合并, 有效的提高了大文件的上传速率, 减少了用户的等待时长.

1 文件上传的常用方式

目前, 基于 HTTP 协议的文件上传方式有以下几种:

(1) 表单上传

这是 Web 开发中最常见的上传方式, 使用 Form 表单的 input[type="input"] 打开文件选择界面, 通过 POST 方法向指定资源提交表单数据[3]. 上传的文件使用 multipart 格式, 编码类型为"multipart/form-data"[4].

(2) 无刷新的 Ajax 上传

区别于表单上传, 使用 Ajax 的异步上传, 在提交表单数据不需要刷新和跳转页面. 提交数据时, 可以使用 FormData 对象模拟表单提交, 发送表单的二进制文件内容, 通过 XMLHttpRequest 实例将参数提交至服务端[5].

(3) Flash 上传

在传统表单的上传功能基础上, Flash 上传方式在不刷新网页的条件下, 支持多个文件批量上传以及显示上传进度等功能. 它采用 Flash 作为中间代理层与服务端进行通信, 以此为基础的 SWFUpload、Plupload 及 Uploadify 等文件上传插件被广泛应用[6].

(4) 第三方组件上传/插件上传

插件技术主要包括 ActiveX、Applet 等, 虽然可能受限于浏览器的安全性设置, 但在学校及企业内部网站环境中有一定的使用价值[7]. 例如 ActiveX 组件, 在 VB 6.0 运行环境下, 使用关键的 Winsock 控件来建立与服务端之间的通信, 通过 Socket 连接发送文件数据. 文献 [8] 对 FileUpload, SWFUpload 及 SlickUpload 三种组件的特性进行了分析和评估. FileUpload 控件使用简单, 但默认对上传组件的大小有限制, 因而需要通过修改配置文件中响应时间和大小的限制实现大文件的上传. SWFUpload 作为一个开源的 JavaScript 和 Flash 库, 它结合了二者的功能, 可以实现交互性更好的界面展示. Slickupload 是来自国外的商业组件, 其在局域网的文件上传中具有良好的表现[8].

2 关键技术

2.1 Node.js

Node.js 基于事件驱动的非阻塞 I/O 模型, 旨在支持能够管理大量并发请求的轻量级服务器的简单而快速的开发[9]. 受益于 V8, Node.js 性能优越, 运行速度快, 可以在服务端运行, 匿名函数和闭包的使用使其在语言层面具备了异步、事件编程的特性[10] . 在处理二进制数据流时, 常用的有 stream 合并与 buffer 合并两种方式. Node.js 中使用 buffer 库实现原始数据的存储方法, 数据被保存在 buffer 的实例中. Node.js 中的 stream 流是处理流式数据的抽象接口, 在处理较大数据量的文件时, 采用 stream 合并比 buffer 合并更有优势. Buffer 需要一次性将数据全部放入内存, 如果数据流较大容易导致速度慢, 内存爆满. 流模式合并数据则是一边读取数据一边进行操作, 在空间上只占用当前处理数据区域的内存大小, 有效地降低了内存的开销[11]. 同时, 对于传输过程中的加密及压缩处理, stream 流具有更高的扩展性. 因此, 本文选择流合并, 使用 Node.js 的可读流与可写流, 实现读取和写入同步, 提高合并效率.

2.2 HTML5 file system

在 HTML5 中提供了一种通过 File API 规范与本地文件进行交互的标准方法, 它的主要作用是将本地文件以文件对象的形式提供给 Web 应用程序进行访问, 为浏览器端应用程序的开发提供了无限可能[12]. File API 提供了前端处理本地文件的能力, 让图片预览、分块上传、拖拽上传等操作变为可能. 以下是本文所用到的对象简介.

(1) FileList 是一个由 File 对象组成的类数组对象.

(2) File 是 FileList 中的一个对象, 包含文件名称 (name)、大小 (size)、类别 (type)、修改时间 (lastModifiedDate) 等基本信息.

(3) FileReader 用来读取文件的 API, 将文件读取到内存中, 提供将文件读取为文本、base64 图片编码、Buffer 数据类型、二进制字符串等方法, 可以实现预览图片、计算 MD5 等等操作.

(4) Blob 是一个二进制数据, File 对象就继承自 Blob 对象. 通过 slice 方法, 可以使二进制数据按照字节分块, 返回的对象中包含了源 Blob 对象中指定范围内的数据[13].

2.3 Spark-md5

对分片文件的标识也是整个文件处理过程中必不可少的一部分. 异步提交的数据中必须包含文件的唯一标识来确认文件分片的顺序, 验证是否上传完毕[14]. MD5 生成的 hash 码不可逆, 可以作为文件上传的有效标识, 这也是实现文件秒传的基础. Spark-md5 是基于 Javascript 的前端类库, 它基于文件的内容生成相应

hash 值, 利用 File API 对文件进行分块之后再进行 MD5 计算, 与传统的 MD5 计算相比, 它的传输效率很高, 不容易引起浏览器卡顿、崩溃等问题.

2.4 Web worker

Node.js 和 JavaScript 都是单线程编程模型, HTML5 的新特性 Web worker 为浏览器实现多线程操作提供了支持. 在文件上传过程中, 多线程操作显然比单线程更具有优势, 且不容易造成阻塞. Web worker 允许在 Web 程序中并发执行多个 JavaScript 脚本, 每个脚本执行过程都作为一个线程, 各个线程之间彼此独立, 由 JavaScript 引擎负责管理[15]. 线程一旦被创建, 可以在主线程调用 worker 线程, 通过将消息发布到代码指定的事件处理程序.

3 设计与实现

3.1 整体设计

基于对大文件上传常用方法与关键技术的研究, 本文设计并实现了完整的前后端大文件上传系统. 该系统基于 HTTP 协议, 利用 HTML5 的 File API 对需要上传的目标大文件进行分片处理. 同时, 充分发挥 CPU 多核的性能, 创建 Web worker 线程计算和处理分片的文件, 避免主线程阻塞. 通过对分片文件的 MD5 校验及标记, 增加文件传输的安全性. 在此基础上, 通过自适应分片结合多并发上传进行优化, 提高了传输速率. 在服务端, 服务器接收前端传输的分片文件, 按分片顺序依次存储, 当收到前端的合并请求, 服务端使用流模式将收到的所有文件切片进行合并. 此外, 在上传过的切片列表中进行查询比对, 对已经上传过的相同文件无需再传, 避免重复上传. 整个系统的流程示意图如图 1 所示.

3.2 前端实现

3.2.1 Hash 计算

为了使服务端对已上传的内容进行识别, 必须要生成文件和切片的 hash 作为校验. 这里使用 Web worker 为 JavaScript 创造多线程环境, 调用 Worker() 构造函数, 新建一个名为 hash 的 worker 线程. 在主线程调用 worker 线程, 通过 postMessage() 函数传入文件内容切片后得到的数组 fileChunkList, worker 线程利用 FileReader 读取每个切片的 ArrayBuffer 并不断传入 Spark-md5 中, 每计算完一个切片通过 postMessage 向主线程发送一个进度事件. 主线程通过 onMessage 函数监听子线程消息, 待全部文件读取完成后, 子线程将最终的 hash 发送给主线程. 整个流程如图 2 所示.

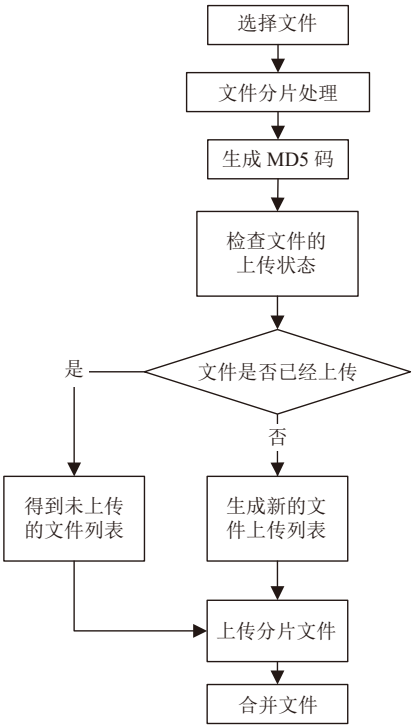

图 1 系统流程图

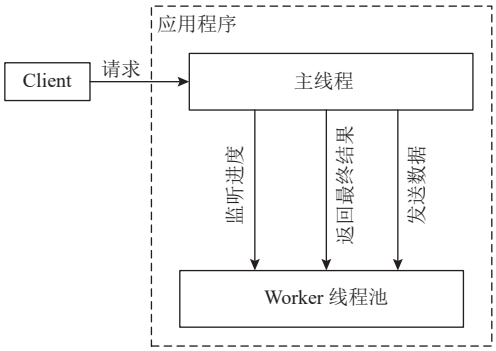

图 2 Web worker 示意图

3.2.2 自适应分片

在实际的应用场景中, 所需要上传的文件大小往往是不固定的, 而分块大小对文件传输有较大影响[16]. 因此, 目前常用的设置固定大小的分片方法不具有灵活性. 自适应分片算法的核心在于, 根据上传文件时的网络状况, 实现切片大小的动态调整. 在当前切片文件上传完成时, 通过获取当前切片文件所用上传时间来调整下一个切片文件的大小, 目的是为了每次上传时切片大小与当前网速相匹配, 具有更好的传输效率[17].

参考 TCP 协议的慢启动策略思想, 从分片的小容量文件传输开始试探网络状况, 根据实际测得结果动态调整下一次分片的大小[18]. 比如, 如果理想的状态下每 20 s 上传一个文件块, 其初始文件大小为 1 MB, 实际计算的上传时间仅为 10 s, 那么可以动态的调整下一个分片的大小为 2 MB. 另一种可能是实际上传所用时间为 40 s, 那么说明当前网络状况不足以传输 1 MB 文件, 下一个文件的分片大小可以改为初始值的一半. 因而, 在自适应分片算法的计算方法中, 设置一个初始切片文件大小为 *fileChunk*, 设置理想的上传单个分片所需时间为 *ts*, 实际上传过程中每个切片所用时间为 *t*, 那么当前切片的上传速率 *rate* 可以表示为 *t*/*ts*. 此时下一切片的文件大小 *newFileChunk* 的计算方式为:

$$newFileChunk = fileChunk/rate \tag{1}$$

本文参照文献 [4] 的参数, 设置初始文件大小设为 1 MB, 理想的参照上传时间 *ts* 为 2 s, 实际上传中所用时间 *t* 通过 new Date().getTime() 获取上传请求前后的时间戳, 得到当前切片上传时间. 利用式 (1) 不断计算得到新的下一切片大小, 达到切片大小动态调整的效果.

切片调整部分关键代码摘录如下:

```
while (cur < fileSize) {
const chunk = file.slice(cur, cur + offset);
cur += offset;
const chunkName = this.container.hash + “-” + count;
const form = new FormData();
form.append(“chunk”, chunk);
form.append(“hash”, chunkName);
form.append(“filename”, this.container.file.name);
form.append(“fileHash”, this.container.hash);
let start = new Date().getTime();
await request({ url: ‘/upload’, data: form })
const now = new Date().getTime();
//获取文件上传过程用时
const time = ((now–start)/1000).toFixed(2);
let rate = time/ts;
//按当前速率调整切片大小
offset = parseInt(offset/rate);
count++;
}
```

3.2.3 多并发上传

为充分利用网络带宽, 采用多并发的方式进行文件上传. 并发上传的并发数受浏览器支持的最大并发数限制, 超过这个值, 执行过程中的并发请求需要等待. 文献 [7] 中采用固定分片大小结合多并发上传, 研究得到在双核处理器条件下, 并发数为 3 时上传文件的耗时出现拐点, 也即上传时间出现明显的减少. 本文设置 *max* 为最大并发数, 通过 while 循环执行并发请求, 设置 *counter* 计数, 当 *max*>0 并且当前计数值小于请求长度时进入循环体. 进入执行循环 *max* 值减少 1, 每次传输完成, 释放并发通道, 以此保证并发数在设定值. 通过对 *max* 取值 3 到 6 进行分别测试, 得到上传耗时在 *max* 取值为 5 时出现明显减少. 以此为基础结合自适应分片, 在代码实现中设置并发数为 5, 使得文件的分片大小每 5 片为一组进行自适应大小的变化, 实际耗时 *t* 通过 5 个切片文件的上传总耗时求平均得到. 通过这样的改进方法, 得到更短的上传耗时.

多并发上传结合自适应分片算法的流程示意图如图 3 所示.

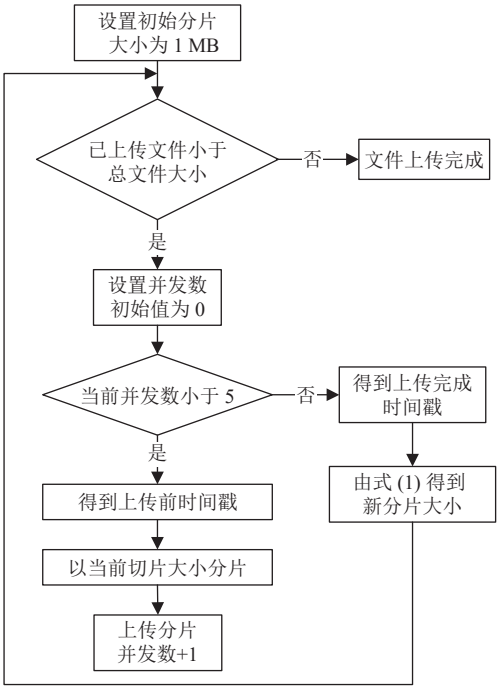

图 3 流程示意图

3.3 服务端实现

3.3.1 接收切片文件

对前端传递的 FormData, 服务端使用 multiparty 包进行处理, 创建 target 文件夹作为文件上传的存储目

录. 前端在发送每个切片时都携带了唯一标识 hash, 服务端将处理后的分片对象从临时路径移动到切片文件夹中.

3.3.2 合并切片

服务端接收到来自前端的合并请求后, 对切片所在文件夹下的所有切片进行合并. 首先采用 sort() 方法根据切片的下标进行排序, 避免从目录读取的文件顺序发生错乱[19]. 使用 fs.createWriteStream 生成可写流, 通过 fs.createReadStream 生成可读流, 将切片文件夹内的切片传输到目标文件夹中并合并. createWriteStream 方法的两个参数控制可读流传输到可写流指定的位置. 这样做能保证在并发合并多个可读流时, 不必按照流的顺序一个接一个传输也能使切片传输到正确的位置[20]. 与确定上一个写入完成再读取下一个流的方式相比, 多并发上传大大提高了传输效率.

3.3.3 文件秒传

文件 hash 值与文件后缀作为目录, 使用 fse.existsSync 检测文件目录是否存在, 如果存在, 则将标志位置为 false, 不需要再次上传. 如果不存在, 则将标志位置为 true. 在此基础上, 文件秒传的实现只需要在用户选择上传已存在的相同资源时, 直接提示上传成功. 在前文服务端验证 hash 的基础上, 如果发现 hash 相同的文件, 说明该文件资源已经上传, 可以直接返回上传成功.

4 实验分析

本文使用文献 [7] 中的设计方法作为对照, 将固定分片上传与自适应分片上传的方法进行对比. 选取了 3 个 100 MB 以上不同大小的文件进行测试, 文献 [7] 所用方法测得的时间记做原始方法用时, 本文提出的方法记做改进方法. 原始方法采用固定分片大小 2 MB, 同时选择并发数为 5 进行多并发上传; 改进方法选择相同的并发数, 采用改进的自适应分片算法, 以 2 MB 大小为起始分片大小进行上传. 浏览器选择 Chrome, 通过控制台的网络 network 面板查看分片的请求状态以及实验结果. 在同样的网络环境下, 每个文件采用两种上传方式分别进行 3 次测试, 统计其平均值作为对照, 测试结果如表 1 所示.

在文件上传的测试过程中, 针对同一文件, 再次上传时, 经过对 MD5 码的校验, 可以直接实现秒传, 系统弹窗提示上传成功. 表 1 中传输时间的计算是从开始上传到服务端接口合并完成文件的整个过程. 由表 1 可以看出, 本系统可以支持 500 MB 以上大文件的上传, 不同大小的文件上传所用时间改进方法均少于原始方法耗时, 并且随着源文件大小的增大更大时比固定分片上传具有更明显的上传时间优势. 通过后端流合并的方式对分片文件进行合并, 得到的上传文件与源文件一致, MD5 值的唯一标识也保证了文件秒传的实现, 避免对同一文件的重复上传, 节约了时间成本.

表 1 实验结果 (s)

文件大小	原始方法用时	平均时间	改进方法用时	平均时间
194 MB	10.668	9.90	7.127	7.46
	10.011		7.703	
	9.021		7.539	
576 MB	28.273	28.07	23.828	22.10
	26.293		21.163	
	29.640		21.302	
1.19 GB	53.663	56.35	47.379	47.95
	57.858		48.341	
	57.531		48.131	

5 结论

本文研究并介绍了常用的大文件上传方法以及存在的问题, 对本系统所用到的关键技术 Node.js 及 File API 进行了阐述, 通过对前后端上传过程的具体研究, 实现了基于 Node.js 的大文件上传系统, 通过对多并发上传与自适应切片相结合的算法, 实现了更具有灵活性和更高传输效率的大文件上传. 同时针对大文件的 MD5 标识计算, 利用 Web worker 多线程计算的方式有效地避免主线程的卡顿. 该系统灵活度高, 适用性强, 能够在文件上传过程中提高上传效率, 提升用户体验.

参考文献

1 张立伟, 李涪帆. 基于 java 语言的大文件分片传输. 通讯世界, 2020, 27(6): 51, 53.
2 周明俊. 基于 PHP 大文件上传的研究和设计. 福建电脑, 2009, 25(4): 147–148. [doi: 10.3969/j.issn.1673-2782.2009.04.095]
3 路石坚. 一种基于 HTTP 的断点续传客户端. 电脑编程技巧与维护, 2017, (9). 71–72. [doi: 10.3969/j.issn.1006-4052.2017.09.027]
4 王建斌, 赵靓. Web 上传文件的三种解决方案. 计算机与信息技术, 2011, 19(S1): 65–68.
5 刘耀钦. 利用 HTML5 拖放技术实现多文件异步上传. 四川理工学院学报 (自然科学版), 2015, 28(1): 17–20, 30. [doi: 10.11863/j.suse.2015.01.05]

6 陈涛, 黄艳峰. Java Web 开发中文件上传方法研究与实现. 电脑知识与技术, 2016, 12(11): 48–49, 52. [doi: 10.14004/j.cnki.ckt.2016.1265]
7 阮晓龙, 李朋楠. 基于 Web 的大文件高效上传方法. 计算机系统应用, 2020, 29(3): 234–239. [doi: 10.15888/j.cnki.csa.007352]
8 刘苡, 吴刚. 基于. Net 的 Web 应用系统中大文件传输方案的研究. 微型电脑应用, 2012, 28(7): 27–30. [doi: 10.3969/j.issn.1007-757X.2012.07.008]
9 任强, 车鹏飞. 基于 Node. js 平台物联网 Web 服务的设计与实现. 现代科学仪器, 2020, (1): 29–33.
10 Tilkov S, Vinoski S. Node.js: Using JavaScript to build high-performance network programs. IEEE Internet Computing, 2010, 14(6): 80–83.
11 张煜. 一种使用 Node.js 构建的分布式数据流日志服务系统. 计算机系统应用, 2013, 22(2): 68–71.
12 胡渝苹. 文件秒传系统在云存储环境下的设计与实现. 计算机应用与软件, 2016, 33(4): 329–333. [doi: 10.3969/j.issn.1000-386x.2016.04.076]
13 王莉敏, 梁正和, 段全锋. 基于 HTML5 大文件断点续传的实现方案. 计算机与现代化, 2016, (3): 91–95. [doi: 10.3969/j.issn.1006-2475.2016.03.018]
14 叶文全. 基于 HTML5、AJAX 的文件分割上传与加密存储研究. 三明学院学报, 2018, 35(4): 60–66.
15 任双君, 周旭, 任勇毛, 等. 基于 HTML5 的浏览器端多线程下载技术. 计算机系统应用, 2017, 26(11): 11–18. [doi: 10.15888/j.cnki.csa.006091]
16 黎苑文, 程明智, 徐秀花, 等. 断点续传及多线程机制在远程传版中的应用研究. 北京印刷学院学报, 2012, 20(6): 53–56. [doi: 10.3969/j.issn.1004-8626.2012.06.019]
17 邓彬, 成卫青. 基于改进慢启动算法的大文件快速传输. 计算机应用研究, 2020, 37(3): 860–863. [doi: 10.19734/j.issn.1001-3695.2018.09.0645]
18 黎国华. 基于历史连接参数的网络拥塞改进算法研究. 网络安全技术与应用, 2020, (5): 57–58. [doi: 10.3969/j.issn.1009-6833.2020.05.032]
19 邹鹤敏, 黄海于. 大文件分块上传和下载软件的设计与实现. 电子技术应用, 2013, 39(8): 137–139. [doi: 10.3969/j.issn.0258-7998.2013.08.040]
20 刘昊, 杨世平. 基于新型分割技术的文件云存储. 贵州大学学报 (自然科学版), 2017, 34(2): 76–79. [doi: 10.15958/j.cnki.gdxbzrb.2017.02.16]

(校对责编: 孙君艳)