链滴

实战篇：断点续传？文件秒传？手撸大文件上传

作者：jianzh5

原文链接：https://ld246.com/article/1636731734181

来源网站：链滴

各位看官大家好，今天给大家分享的又是一篇实战文章，希望大家能够喜欢。

开味菜

最近接到一个新的需求，需要上传 2G左右的视频文件，用测试环境的 OSS试了一下，上传需要十几钟，再考虑到公司的网络资源问题，果断放弃该方案。

一提到大文件上传，我最先想到的就是各种网盘了，现在大家都喜欢将自己收藏的 **小电影**上传到网盘行保存。网盘一般都支持断点续传和文件秒传功能，减少了网络波动和网络带宽对文件的限制，大大高了用户体验，让人爱不释手。

说到这，大家先来了解一下这几个概念：

- **文件分块**：将大文件拆分成小文件，将小文件上传\下载，最后再将小文件组装成大文件；
- **断点续传**：在文件分块的基础上，将每个小文件采用单独的线程进行上传\下载，如果碰到网络故障可以从已经上传\下载的部分开始继续上传\下载未完成的部分，而没有必要从头开始上传\下载；
- **文件秒传**：资源服务器中已经存在该文件，其他人上传时直接返回该文件的URI。

RandomAccessFile

平时我们都会使用 FileInputStream，FileOutputStream，FileReader以及 FileWriter等 IO流来读文件，今天我们来了解一下 RandomAccessFile。

它是一个直接继承 Object的独立的类，底层实现中它实现的是 DataInput和 DataOutput接口。该支持随机读取文件，随机访问文件类似于文件系统中存储的大字节数组。

它的实现基于**文件指针**（一种游标或者指向隐含数组的索引），文件指针可以通过 getFilePointer方读取，也可以通过 seek方法设置。

输入时从文件指针开始读取字节，并使文件指针超过读取的字节，如果写入超过隐含数组当前结尾的出操作会导致扩展数组。该类有四种模式可供选择：

- r： 以只读方式打开文件，如果执行写入操作会抛出 IOException;
- rw： 以读、写方式打开文件，如果文件不存在，则尝试创建文件；
- rws： 以读、写方式打开文件，要求对文件内容或元数据的每次更新都同步写入底层存储设备；
- rwd： 以读、写方式打开文件，要求对文件内容的每次更新都同步写入底层存储设备；

在 rw模式下，默认是使用 buffer的，只有 cache满的或者使用 RandomAccessFile.close()关闭流的候才真正的写到文件。

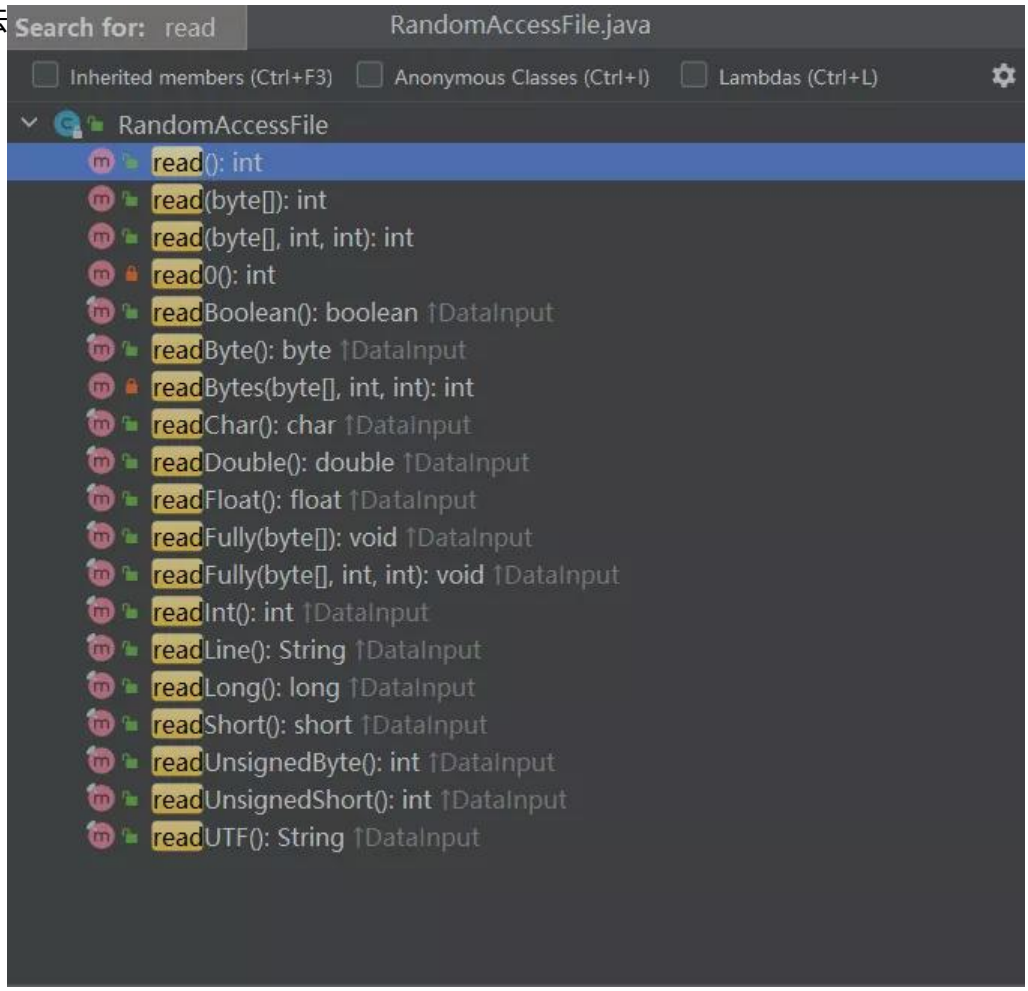
常用API

1. void seek(long pos)：设置下一次读取或写入时的文件指针偏移量，通俗点说就是指定下次读文数据的位置。

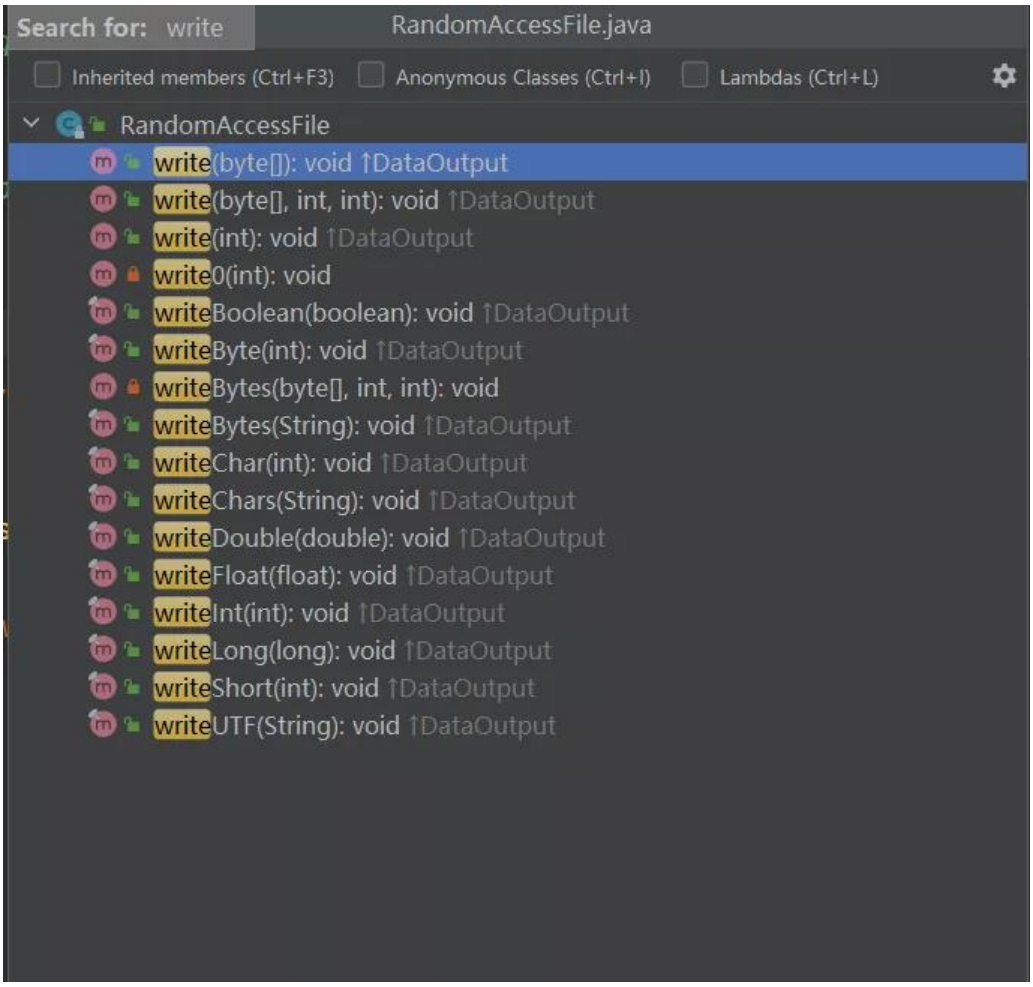
偏移量可以设置在文件末尾之外，只有在偏移量设置超出文件末尾后，才能通过写入更改文件长度；

2. native long getFilePointer()：返回当前文件的光标位置；
3. native long length()：返回当前文件的长度；

4. **读**方法


Search for: read
RandomAccessFile.java
Inherited members (Ctrl+F3)
Anonymous Classes (Ctrl+I)
Lambdas (Ctrl+L)
RandomAccessFile
read(): int
read(byte[]): int
read(byte[], int, int): int
read0(): int
readBoolean(): boolean ↑DataInput
readByte(): byte ↑DataInput
readBytes(byte[], int, int): int
readChar(): char ↑DataInput
readDouble(): double ↑DataInput
readFloat(): float ↑DataInput
readFully(byte[]): void ↑DataInput
readFully(byte[], int, int): void ↑DataInput
readInt(): int ↑DataInput
readLine(): String ↑DataInput
readLong(): long ↑DataInput
readShort(): short ↑DataInput
readUnsignedByte(): int ↑DataInput
readUnsignedShort(): int ↑DataInput
readUTF(): String ↑DataInput

5. **写**方法



6. readFully(byte[] b)：这个方法的作用就是将文本中的内容填满这个缓冲区b。如果缓冲b不能被填，那么读取流的过程将被阻塞，如果发现是流的结尾，那么会抛出异常；

7. FileChannel getChannel()：返回与此文件关联的唯一FileChannel对象；

8. int skipBytes(int n)：试图跳过n个字节的输入，丢弃跳过的字节；

RandomAccessFile的绝大多数功能，已经被 JDK1.4的NIO的**「内存映射」**文件取代了，即把文映射到内存后再操作，省去了频繁磁盘 io。

主菜

总结经验，砥砺前行：之前的实战文章中过多的粘贴了源码，影响了各位小伙伴的阅读感受。经过大的点拨，以后将展示部分关键代码，供各位赏析。

文件分块

文件分块需要在前端进行处理，可以利用强大的 js库或者现成的组件进行分块处理。需要确定分块的小和分块的数量，然后为每一个分块指定一个索引值。

为了防止上传文件的分块与其它文件混淆，采用文件的 md5值来进行区分，该值也可以用来校验服器上是否存在该文件以及文件的上传状态。

- 如果文件存在，直接返回文件地址；

- 如果文件不存在，但是有上传状态，即部分分块上传成功，则返回未上传的分块索引数组；
- 如果文件不存在，且上传状态为空，则所有分块均需要上传。

```
fileRederInstance.readAsBinaryString(file);
fileRederInstance.addEventListener("load", (e) => {
    let fileBolb = e.target.result;
    fileMD5 = md5(fileBolb);
    const formData = new FormData();
    formData.append("md5", fileMD5);
    axios
        .post(http + "/fileUpload/checkFileMd5", formData)
        .then((res) => {
            if (res.data.message == "文件已存在") {
                //文件已存在不走后面分片了，直接返回文件地址到前台页面
                success && success(res);
            } else {
                //文件不存在存在两种情况，一种是返回data：null代表未上传过 一种是data:[xx，xx] 还
哪几片未上传
                if (!res.data.data) {
                    //还有几片未上传情况，断点续传
                    chunkArr = res.data.data;
                }
                readChunkMD5();
            }
        })
        .catch((e) => {});
});
```

在调用上传接口前，通过 slice方法来取出索引在文件中对应位置的分块。

```
const getChunkInfo = (file, currentChunk, chunkSize) => {
        //获取对应下标下的文件片段
        let start = currentChunk * chunkSize;
        let end = Math.min(file.size, start + chunkSize);
        //对文件分块
        let chunk = file.slice(start, end);
        return { start, end, chunk };
    };
```

之后调用上传接口完成上传。

断点续传、文件秒传

后端基于 spring boot开发，使用 redis来存储上传文件的状态和上传文件的地址。

如果文件完整上传，返回文件路径；部分上传则返回未上传的分块数组；如果未上传过返回提示信息。

在上传分块时会产生两个文件，一个是文件主体，一个是临时文件。临时文件可以看做是一个数组文，为每一个分块分配一个值为127的字节。

校验MD5值时会用到两个值：

- 文件上传状态：只要该文件上传过就不为空，如果完整上传则为 true，部分上传返回false；

- 文件上传地址：如果文件完整上传，返回文件路径；部分上传返回临时文件路径。

```
/**
 * 校验文件的MD5
 **/
public Result checkFileMd5(String md5) throws IOException {
    //文件是否上传状态：只要该文件上传过该值一定存在
    Object processingObj = stringRedisTemplate.opsForHash().get(UploadConstants.FILE_UPL
AD_STATUS, md5);
    if (processingObj == null) {
        return Result.ok("该文件没有上传过");
    }
    boolean processing = Boolean.parseBoolean(processingObj.toString());
    //完整文件上传完成时为文件的路径，如果未完成返回临时文件路径（临时文件相当于数组，为每
分块分配一个值为127的字节）
    String value = stringRedisTemplate.opsForValue().get(UploadConstants.FILE_MD5_KEY + m
5);
    //完整文件上传完成是true，未完成返回false
    if (processing) {
        return Result.ok(value,"文件已存在");
    } else {
        File confFile = new File(value);
        byte[] completeList = FileUtils.readFileToByteArray(confFile);
        List<Integer> missChunkList = new LinkedList<>();
        for (int i = 0; i < completeList.length; i++) {
            if (completeList[i] != Byte.MAX_VALUE) {
                //用空格补齐
                missChunkList.add(i);
            }
        }
        return Result.ok(missChunkList,"该文件上传了一部分");
    }
}
```

说到这，你肯定会问：当这个文件的所有分块上传完成之后，该怎么得到完整的文件呢？接下来我们说一下分块合并的问题。

分块上传、文件合并

上边我们提到了利用文件的 md5值来维护分块和文件的关系，因此我们会将具有相同 md5值的分块行合并，由于每个分块都有自己的索引值，所以我们会将分块按索引像插入数组一样分别插入文件中形成完整的文件。

分块上传时，要和前端的分块大小、分块数量、当前分块索引等对应好，以备文件合并时使用，此处们采用的是**磁盘映射** 的方式来合并文件。

```
 //读操作和写操作都是允许的
RandomAccessFile tempRaf = new RandomAccessFile(tmpFile, "rw");
//它返回的就是nio通信中的file的唯一channel
FileChannel fileChannel = tempRaf.getChannel();

//写入该分片数据  分片大小 * 第几块分片获取偏移量
long offset = CHUNK_SIZE * multipartFileDTO.getChunk();
```

```
//分片文件大小
byte[] fileData = multipartFileDTO.getFile().getBytes();
//将文件的区域直接映射到内存
MappedByteBuffer mappedByteBuffer = fileChannel.map(FileChannel.MapMode.READ_WRITE,
offset, fileData.length);
mappedByteBuffer.put(fileData);
// 释放
FileMD5Util.freedMappedByteBuffer(mappedByteBuffer);
fileChannel.close();
```

每当完成一次分块的上传，还需要去检查文件的上传进度，看文件是否上传完成。

```
RandomAccessFile accessConfFile = new RandomAccessFile(confFile, "rw");
//把该分段标记为 true 表示完成
accessConfFile.setLength(multipartFileDTO.getChunks());
accessConfFile.seek(multipartFileDTO.getChunk());
accessConfFile.write(Byte.MAX_VALUE);

//completeList 检查是否全部完成,如果数组里是否全部都是(全部分片都成功上传)
byte[] completeList = FileUtils.readFileToByteArray(confFile);
byte isComplete = Byte.MAX_VALUE;
for (int i = 0; i < completeList.length && isComplete == Byte.MAX_VALUE; i++) {
    //与运算, 如果有部分没有完成则 isComplete 不是 Byte.MAX_VALUE
    isComplete = (byte) (isComplete & completeList[i]);
}
accessConfFile.close();
```

然后更新文件的上传进度到 Redis中。

```
//更新redis中的状态：如果是true的话证明是已经该大文件全部上传完成
if (isComplete == Byte.MAX_VALUE) {
    stringRedisTemplate.opsForHash().put(UploadConstants.FILE_UPLOAD_STATUS, multipartFi
eDTO.getMd5(), "true");
    stringRedisTemplate.opsForValue().set(UploadConstants.FILE_MD5_KEY + multipartFileDTO
getMd5(), uploadDirPath + "/" + fileName);
} else {
    if (!stringRedisTemplate.opsForHash().hasKey(UploadConstants.FILE_UPLOAD_STATUS, mult
partFileDTO.getMd5())) {
        stringRedisTemplate.opsForHash().put(UploadConstants.FILE_UPLOAD_STATUS, multipar
FileDTO.getMd5(), "false");
    }
    if (!stringRedisTemplate.hasKey(UploadConstants.FILE_MD5_KEY + multipartFileDTO.getMd
())) {
        stringRedisTemplate.opsForValue().set(UploadConstants.FILE_MD5_KEY + multipartFileD
O.getMd5(), uploadDirPath + "/" + fileName + ".conf");
    }
}
```

以上就是今天的全部内容了，希望对你有所帮助！