面试官：你如何实现大文件上传

提到大文件上传，在脑海里最先想到的应该就是将图片保存在自己的服务器（如七牛云服务器），保存在数据库，不仅可以当做地址使用，还可以当做资源使用；或者将图片转换成base64，转换成buffer流，但是在javascript这门语言中不存在，但是这些只适用于一些小图片，对于大文件还是束手无策。

一、问题分析

如果将大文件一次性上传，会发生什么？想必都遇到过在一个大文件上传、转发等操作时，由于要上传大量的数据，导致整个上传过程耗时漫长，更有甚者，上传失败，让你重新上传！这个时候，我已经咬牙切齿了。先不说上传时间长久，毕竟上传大文件也没那么容易，要传输更多的报文，丢包也是常有的事，而且在这个时间段万不可以做什么其他会中断上传的操作；其次，前后端交互肯定是有时间限制的，肯定不允许无限制时间上传，大文件又更容易超时而失败....

一、解决方案

既然大文件上传不适合一次性上传，那么将文件分片散上传是不是就能减少性能消耗了。

没错，就是分片上传。分片上传就是将大文件分成一个个小文件（切片），将切片进行上传，等到后端接收到所有切片，再将切片合并成大文件。通过将大文件拆分成多个小文件进行上传，确实就是解决了大文件上传的问题。因为请求时可以并发执行的，这样的话每个请求时间就会缩短，如果某个请求发送失败，也不需要全部重新发送。

二、具体实现

1、前端

（1）读取文件

准备HTML结构，包括：读取本地文件（`input`类型为`file`）、上传文件按钮、上传进度。

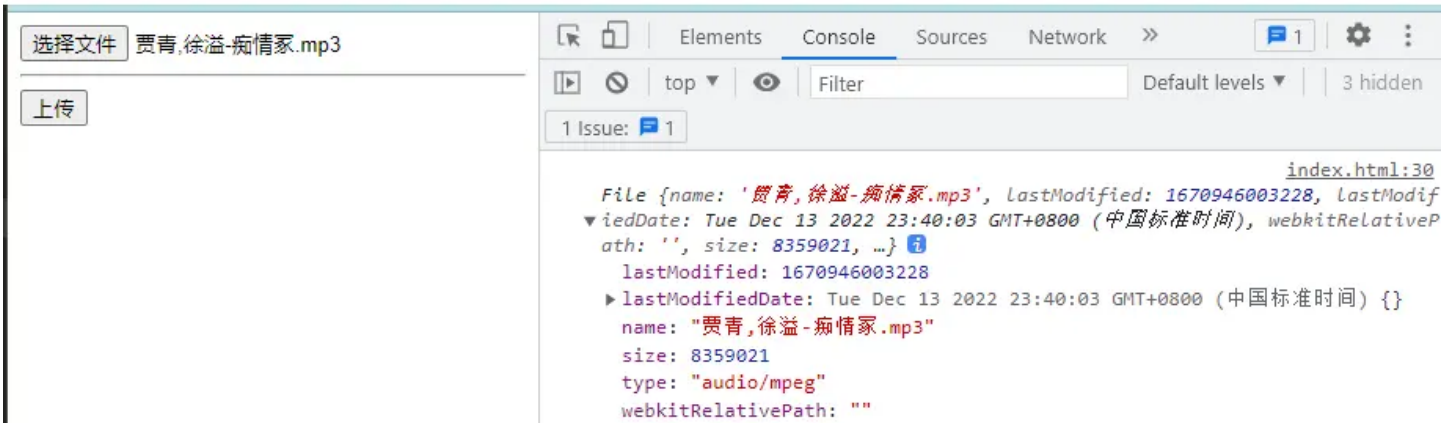
```
1 xml
2 复制代码
3 <input type="file" id="input"<button id="upload"上传</button<!-- 上传进度 --><div
   style="width: 300px" id="progress"</div
```

JS实现文件读取：

监听`input`的`change`事件，当选取了本地文件后，打印事件源可得到文件的一些信息：

```
javascript
复制代码
let input = document.getElementById('input')let upload =
document.getElementById('upload')let files = {}//创建一个文件对象let chunkList =
[]//存放切片的数组// 读取文件input.addEventListener('change', (e) => {    files =
e.target.files[0]    console.log(files);        //创建切片    //上传切片})
```

观察控制台，打印读取的文件信息如下：



（2）创建切片

文件的信息包括文件的名字，文件的大小，文件的类型等信息，接下来可以根据文件的大小来进行切片，例如将文件按照1MB或者2MB等大小进行切片操作：

- arduino
  复制代码
  // 创建切片function createChunk(file, size = 210241024) {//两个形参：file是大文件，size是切片的大小 const chunkList = [] let cur = 0 while (cur < file.size) {   chunkList.push({
  file: file.slice(cur, cur + size)//使用slice()进行切片   })   cur += size } return chunkList}

切片的核心思想是：创建一个空的切片列表数组 `chunkList` ，将大文件按照每个切片2MB进行切片操作，因为File 接口没有定义任何方法，但是它从 `Blob` 接口继承了以下方法：`Blob.slice([start[, end[, contentType]]])` ，这里使用的是Blob接口的 `Blob.slice()` 方法，那么每个切片都应该在2MB大小左右，如上文件的大小是 `8359021` ，那么可得到4个切片，分别是[0，2MB]、[2MB，4MB]、[4MB,6MB]、[6MB,8MB]。调用 `createChunk函数` ，会返回一个切片列表数组，实际上，有几个切片就相当于有几个请求。

调用创建切片函数：

```
scss
复制代码
//注意调用位置，不是在全局，而是在读取文件的回调里调用chunkList =
createChunk(files)console.log(chunkList);
```

观察控制台打印的结果：

（3）上传切片

上传切片的个关键的操作：

第一、数据处理。需要将切片的数据进行维护成一个包括该文件，文件名，切片名的对象，所以采用 `FormData` 对象来进行整理数据。 `FormData 对象` 用以将数据编译成键值对,可用于发送带键数据，通过调用它的 `append()` 方法来添加字段，FormData.append() 方法会将字段类型为数字类型的转换成字符串（字段类型可以是 `Blob、File` 或者字符串：**如果它的字段类型不是 Blob 也不是 File，则会被转换成字符串类**。

第二、并发请求。每一个切片都分别作为一个请求，只有当这4个切片都传输给后端了，即四个请求都成功发起，才上传成功，使用 `Promise.all()` 保证所有的切片都已经传输给后端。

```
1 javascript
2 复制代码
3 //数据处理async function uploadFile(list) {    const requestList =
  list.map(({file,fileName,index,chunkName}) => {        const formData = new
  FormData() // 创建表单类型数据        formData.append('file', file)//该文件
   formData.append('fileName', fileName)//文件名
  formData.append('chunkName', chunkName)//切片名        return {formData,index}
    })        .map(({formData,index}) =>axiosRequest({            method:
  'post',            url: 'http://localhost:3000/upload',//请求接口，要与后端一一一对
  应            data: formData        })            .then(res => {
  console.log(res);                //显示每个切片上传进度                let p =
  document.createElement('p')                p.innerHTML =
  `${list[index].chunkName}--${res.data.message}`
  document.getElementById('progress').appendChild(p)            })        )
    await Promise.all(requestList)//保证所有的切片都已经传输完毕}//请求函数function
  axiosRequest({method = "post",url,data}) {    return new Promise((resolve,
  reject) => {        const config = {//设置请求头            headers: 'Content-
  Type:application/x-www-form-urlencoded',        }        //默认是post请求，可更改
        axios[method](url,data,config).then((res) => {            resolve(res)
      })    })}// 文件上传upload.addEventListener('click', () => {    const
  uploadList = chunkList.map(({file}, index) => ({        file,        size:
  file.size,        percent: 0,        chunkName: `${files.name}-${index}`,
    fileName: files.name,        index    }))    //发请求，调用函数
  uploadFile(uploadList)})
```

2、后端

（1）接收切片

主要工作：

第一：需要引入 `multiparty` 中间件，来解析前端传来的 `FormData` 对象数据；

第二：通过`path.resolve()`在根目录创建一个文件夹--`qiepian`，该文件夹将存放另一个文件夹（存放所有的切片）和合并后的文件；

第三：处理跨域问题。通过`setHeader()`方法设置所有的请求头和所有的请求源都允许；

第四：解析数据成功后，拿到文件相关信息，并且在`qiepian`文件夹创建一个新的文件夹`${fileName}-chunks`，用来存放接收到的所有切片；

第五：通过`fse.move(filePath,fileName)`将切片移入`${fileName}-chunks`文件夹，最后向前端返回上传成功的信息。

```
1 javascript
2 复制代码
3 //app.jsconst http = require('http')const multiparty = require('multiparty')//
  中间件，处理FormData对象的中间件const path = require('path')const fse =
  require('fs-extra')//文件处理模块const server = http.createServer()const
  UPLOAD_DIR = path.resolve(__dirname, '.', 'qiepian')// 读取根目录，创建一个文件夹
  qiepian存放切片server.on('request', async (req, res) => {    // 处理跨域问题，允许
  所有的请求头和请求源    res.setHeader('Access-Control-Allow-Origin', '*')
  res.setHeader('Access-Control-Allow-Headers', '*')    if (req.url ===
  '/upload') { //前端访问的地址正确        const multipart = new multiparty.Form()
  // 解析FormData对象        multipart.parse(req, async (err, fields, files) => {
            if (err) { //解析失败                return            }
  console.log('fields=', fields);            console.log('files=', files);
                const [file] = files.file            const [fileName] =
  fields.fileName            const [chunkName] = fields.chunkName
        const chunkDir = path.resolve(UPLOAD_DIR, `${fileName}-chunks`)//在
  qiepian文件夹创建一个新的文件夹，存放接收到的所有切片            if
  (!fse.existsSync(chunkDir)) { //文件夹不存在，新建该文件夹                await
  fse.mkdirs(chunkDir)            }            // 把切片移动进chunkDir
  await fse.move(file.path, `${chunkDir}/${chunkName}`)
  res.end(JSON.stringify({ //向前端输出                code: 0,
  message: '切片上传成功'            }))        })    }})server.listen(3000, () =>
  {    console.log('服务已启动');})
```

通过`node app.js`启动后端服务，可在控制台打印`fields和files`：

（2）合并切片

第一：前端得到后端返回的上传成功信息后，通知后端合并切片：

```
1 javascript
2 复制代码
3 // 通知后端去做切片合并function merge(size, fileName) {    axiosRequest({
  method: 'post',        url: 'http://localhost:3000/merge',//后端合并请求
  data: JSON.stringify({            size,            fileName        }),    })}//
```

```
调用函数，当所有切片上传成功之后，通知后端合并await
Promise.all(requestList)merge(files.size, files.name)
```

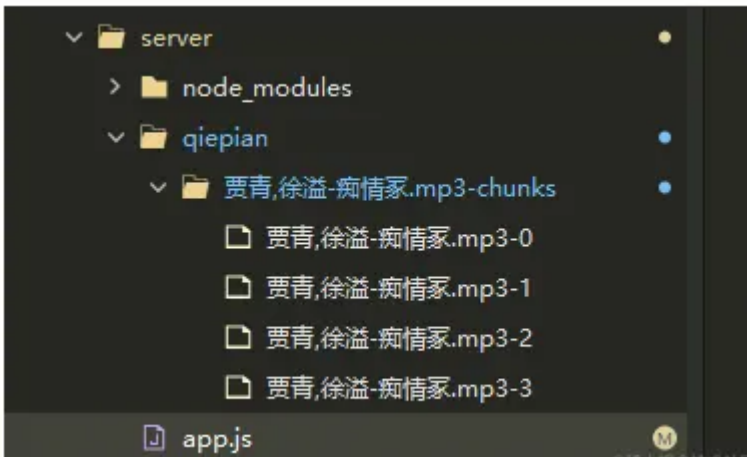
第二：后端接收到合并的数据，创建新的路由进行合并，合并的关键在于：前端通过 `POST` 请求向后端传递的合并数据是通过 `JSON.stringify()` 将数据转换成字符串，所以后端合并之前，需要进行以下操作：

- 解析POST请求传递的参数，自定义函数 `resolvePost` ，目的是将每个切片请求传递的数据进行拼接，拼接后的数据仍然是字符串，然后通过 `JSON.parse()` 将字符串格式的数据转换为JSON对象；
- 接下来该去合并了，拿到上个步骤解析成功后的数据进行解构，通过 `path.resolve` 获取每个切片所在的路径；
- 自定义合并函数 `mergeFileChunk` ，只要传入切片路径，切片名字和切片大小，就真的将所有的切片进行合并。在此之前需要将每个切片转换成流 `stream` 对象的形式进行合并，自定义函数 `pipeStream` ，目的是将切片转换成流对象，在这个函数里面创建可读流，读取所有的切片，监听 `end` 事件，所有的切片读取完毕后，销毁其对应的路径，保证每个切片只被读取一次，不重复读取，最后将汇聚所有切片的可读流汇入可写流；
- 最后，切片被读取成流对象，可读流被汇入可写流，那么在指定的位置通过 `createWriteStream` 创建可写流，同样使用 `Promise.all()` 的方法，保证所有切片都被读取，最后调用合并函数进行合并。

```
1 javascript
2 复制代码
3 if (req.url === '/merge') { // 该去合并切片了        const data = await
  resolvePost(req)        const {            fileName,            size        }
  = data        const filePath = path.resolve(UPLOAD_DIR, fileName)//获取切片路径
        await mergeFileChunk(filePath, fileName, size)
  res.end(JSON.stringify({            code: 0,            message: '文件合并成功'
        }))}// 合并async function mergeFileChunk(filePath, fileName, size) {
  const chunkDir = path.resolve(UPLOAD_DIR, `${fileName}-chunks`)    let
  chunkPaths = await fse.readdir(chunkDir)    chunkPaths.sort((a, b) =>
  a.split('-')[1] - b.split('-')[1])    const arr = chunkPaths.map((chunkPath,
  index) => {        return pipeStream(            path.resolve(chunkDir,
  chunkPath),            // 在指定的位置创建可写流
  fse.createWriteStream(filePath, {                start: index * size,
        end: (index + 1) * size            })        )    })    await
  Promise.all(arr)//保证所有的切片都被读取}// 将切片转换成流进行合并function
  pipeStream(path, writeStream) {    return new Promise(resolve => {        // 创
  建可读流，读取所有切片        const readStream = fse.createReadStream(path)
   readStream.on('end', () => {            fse.unlinkSync(path)// 读取完毕后，删除已
  经读取过的切片路径            resolve()        })
  readStream.pipe(writeStream)//将可读流流入可写流    })}// 解析POST请求传递的参数
```

```
function resolvePost(req) {    // 解析参数    return new Promise(resolve => {
    let chunk = ''        req.on('data', data => { //req接收到了前端的数据
    chunk += data //将接收到的所有参数进行拼接        })        req.on('end', ()
=> {            resolve(JSON.parse(chunk))//将字符串转为JSON对象        })    })}
```

还未合并前，文件夹如下图所示：



合并后，文件夹新增了合并后的文件：