前端求职

疫情期间充电


高效前端
Web高效编程与优化实践
EFFECTIVE FRONT END
奖品: 高效前端实体书（包邮） ×5 份
02月07日 20:00 自动开奖
长按识别小程序，参与抽奖
抽奖说明: 中奖后根据引导填写收货地址

关于我

大圣

https://github.com/shengxinjing

大厂面试章法

简历

2页以内，简历有点类似相亲的介绍，小时候拿过三好学生就别说了，就像别写你会html+css，别写你用vue做过todolist一样

突出自己的技术亮点

别瞎写精通

markdown就好，别用word

突出亮点！

目标公司

天眼查，脉脉，知乎

如何描述做过的项目

1. 做过的明星项目
2. 项目技术栈和细节
3. 源码深度
4. 优化，性能，体验，极客
5. 填坑
6. 成长

面试技巧合计

1. 认识自己（市场）
2. 阐述优势
3. 谈判得来的 都是纯利润
4. hr细节(考勤，补贴，996，五险一金，补贴

一个面试题的精讲

前端文件上传

原理概述

考察全栈思维，http协议，node文件处理

我用vue+element+nodejs来演示

基本答案 （10K）

formData

```
    <input type="file" @change="handleFileChange" />

    <el-button type="primary" @click="handleUpload">上传</el-button>
```

```
  handleFileChange(e) {
    const [file] = e.target.files;
    if (!file) return;

    form.append("filename", this.container.file.name);
    form.append("file", this.container.file);
    request({
        url: '/upload',
        data: form,
    })

  },
```

node

```
const http = require("http")
const path = require('path')
const Controller = require('./controller')
const schedule = require('./schedule')
const server = http.createServer()

const UPLOAD_DIR = path.resolve(__dirname, "..", "target"); // 大文件存储目录
```

```
8
9
10 // schedule.start(UPLOAD_DIR)
11
12 const ctrl = new Controller(UPLOAD_DIR)
13
14
15 server.on("request", async (req, res)
   => {
16   res.setHeader("Access-Control-Allow-
   Origin", "*")
17   res.setHeader("Access-Control-Allow-
   Headers", "*")
18   if (req.method === "OPTIONS") {
19     res.status = 200
20     res.end()
21     return
22   }
23   if (req.method === "POST") {
24     if (req.url == '/upload') {
25       await ctrl.handleUpload(req,res)
26       return
27     }
28   }
29
30 })
31
```

```
server.listen(3000, () =>
console.log("正在监听 3000 端口"))

```

Controller.js

```
  async handleUpload(req, res) {
    const multipart = new
multiparty.Form()
    multipart.parse(req, async (err,
field, file) => {
      if (err) {
        console.log(err)
        return
      }
      const [chunk] = file.file

      const [filename] = field.filename

      const filePath = path.resolve(
        this.UPLOAD_DIR,

 `${fileHash}${extractExt(filename)}`
      )
```

```
16        const chunkDir = path.resolve(this.UPLOAD_DIR, fileHash)
17        // 文件存在直接返回
18        if (fse.existsSync(filePath)) {
19          res.end("file exist")
20          return
21        }
22
23        if (!fse.existsSync(chunkDir)) {
24          await fse.mkdirs(chunkDir)
25        }
26        await fse.move(chunk.path, `${chunkDir}/${hash}`)
27        res.end("received file chunk")
28      })
29    }
```

总结:

1. forData
2. httpserver
3. fs文件处理
4. multiparty解析post数据

加分项-拖拽，粘贴

考点: 拖拽事件drop，clipboardData

```
<div class="drop-box" id="drop-box">
```

```

box.addEventListener("drop", function (e) {
    e.preventDefault(); //取消浏览器默认拖拽效果

    var fileList = e.dataTransfer.files; //获取拖拽中的文件对象
    var len=fileList.length;//用来获取文件的长度（其实是获得文件数量）

    const [file] = e.target.files;
    if (!file) return;

...上传

}, false);
```

粘贴

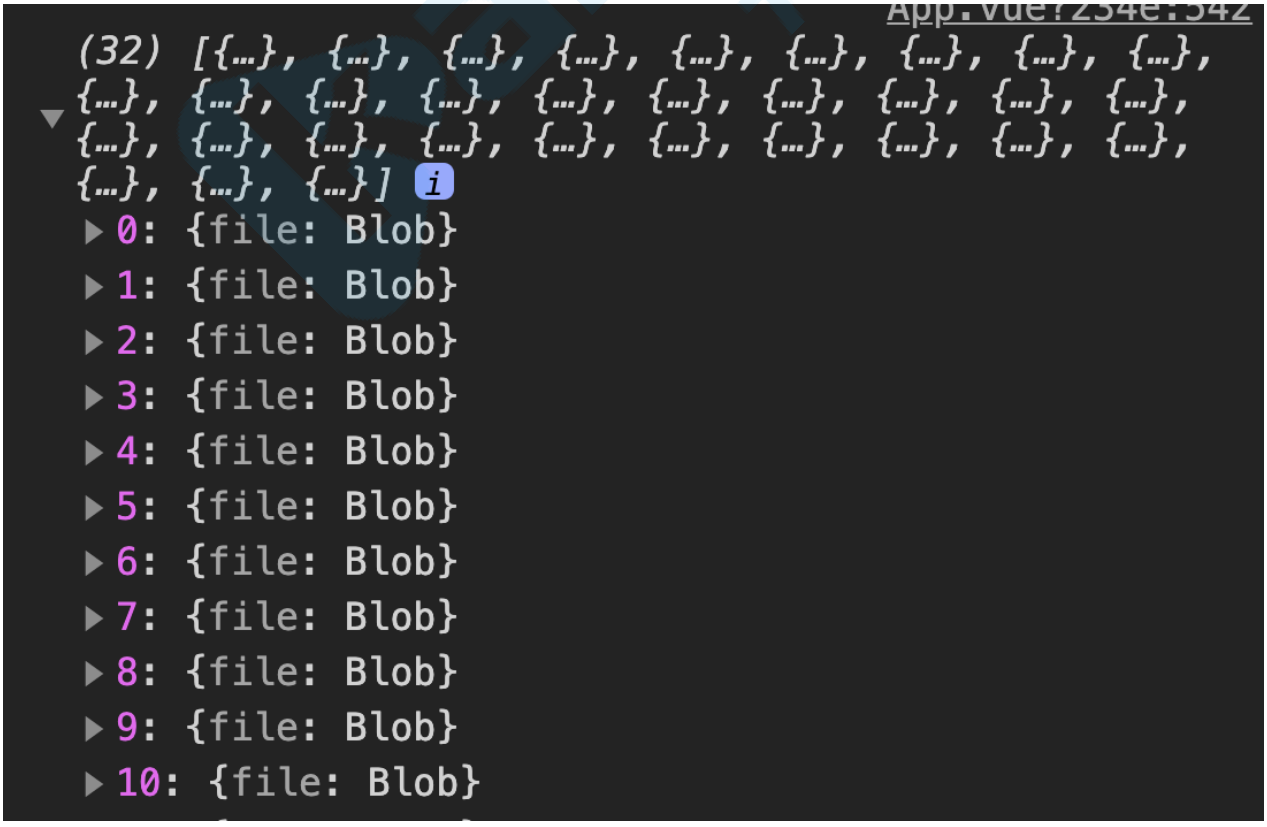
```
box.addEventListener('paste',function (event) {
        var data = (event.clipboardData)

....
    });

```

大文件上传(20K)

blob.slice分片 思想+语法

```
const chunks = this.createFileChunk(this.container.file);
```

```
    createFileChunk(file, size = SIZE)
{
      // 生成文件块
      const chunks = [];
      let cur = 0;
      while (cur < file.size) {
        chunks.push({ file:
file.slice(cur, cur + size) });
        cur += size;
      }
      return chunks;
    },
```



所有切片挨个发请求，然后merge

```
  async handleMerge(req, res) {

    const data = await resolvePost(req)
    const {fileHash, filename, size } = data
    const ext = extractExt(filename)
    const filePath = path.resolve(this.UPLOAD_DIR, `${fileHash}${ext}`)
    await this.mergeFileChunk(filePath, fileHash, size)
    res.end(
      JSON.stringify({
        code: 0,
        message: "file merged success"
      })
    )


  }
```

断点续传+秒传

md5计算，缓存思想 文件用md5计算一个指纹，上传之前，先问后端，这个文件的hash在不在，在的话就不用传了，就是所谓的断点续传，如果整个文件都存在了 就是秒传

```
  async handleVerify(req, res) {
    const data = await resolvePost(req)
    const { filename, hash } = data
    const ext = extractExt(filename)
    const filePath = path.resolve(this.UPLOAD_DIR, `${hash}${ext}`)

    // 文件是否存在
    let uploaded = false
    let uploadedList = []
    if (fse.existsSync(filePath)) {
      uploaded = true
    }else{
      // 文件没有完全上传完毕，但是可能存在部分切片上传完毕了
```

```
      uploadedList = await
getUploadedList(path.resolve(this.UPLOA
D_DIR, hash))
    }
    res.end(
      JSON.stringify({
        uploaded,
        uploadedList // 过滤诡异的隐藏文件
      })
    )

  }
```

计算hash优化(25+)

web-worker

大文件的md5太慢了,启用webworker计算

```


// web-worker
self.importScripts('spark-md5.min.js')

self.onmessage = e=>{
```

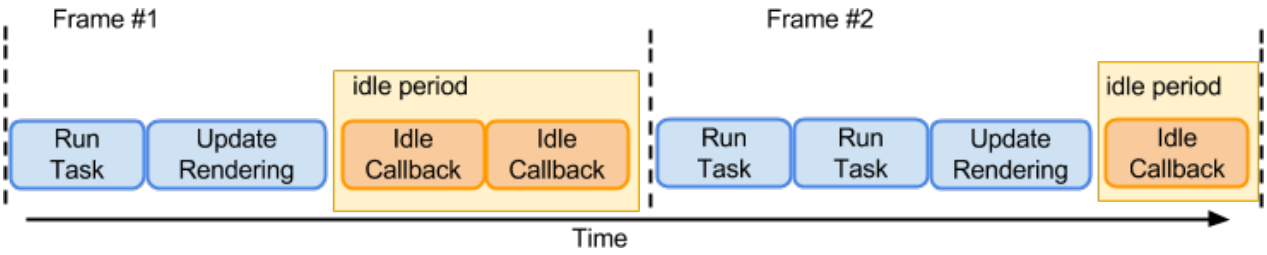
```
7       // 接受主线程的通知
8       const {chunks} = e.data
9       const spark = new
self.SparkMD5.ArrayBuffer()
10      let progress = 0
11      let count = 0
12
13
14
15
16      const loadNext = index=>{
17          const reader = new FileReader()
18
 reader.readAsArrayBuffer(chunks[index]
.file)
19          reader.onload = e=>{
20              // 累加器 不能依赖index,
21              count++
22              // 增量计算md5
23
 spark.append(e.target.result)
24              if(count===chunks.length){
25                  // 通知主线程，计算结束
26                  self.postMessage({
27                      progress:100,
28                      hash:spark.end()
29                  })
```

```
            }else{
                // 每个区块计算结束，通知进度即可
                progress += 100/chunks.length
                self.postMessage({
                    progress
                })
                // 计算下一个
                loadNext(count)
            }
        }
    }
    // 启动
    loadNext(0)

}
```

time-slice(30+)

react fiber架构学习，利用浏览器空闲时间

requestIdleCallback




```
requestIdelCallback(myNonEssentialWork);


    function myNonEssentialWork(deadline) {

        // deadline.timeRemaining()可以获取到当前帧剩余时间
        // 当前帧还有时间 并且任务队列不为空
        while (deadline.timeRemaining() > 0 && tasks.length > 0) {
          doWorkIfNeeded();
        }
        if (tasks.length > 0){

requestIdleCallback(myNonEssentialWork);
        }
    }
```

```
15
16
```

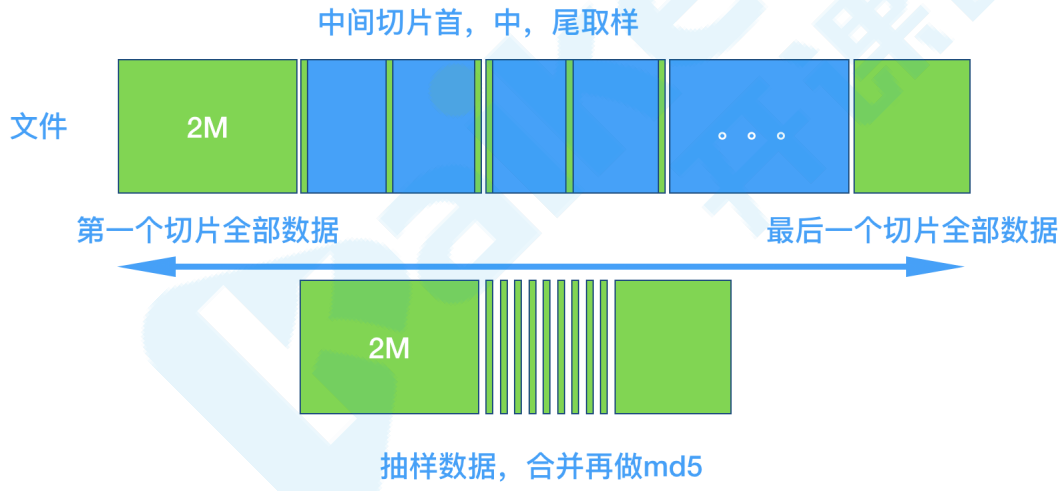
```
1
2       async calculateHashIdle(chunks) {
3         return new Promise(resolve => {
4           const spark = new
SparkMD5.ArrayBuffer();
5           let count = 0;
6           // 根据文件内容追加计算
7           const appendToSpark = async
file => {
8             return new Promise(resolve =>
{
9               const reader = new
FileReader();
10
 reader.readAsArrayBuffer(file);
11               reader.onload = e => {
12
 spark.append(e.target.result);
13                 resolve();
14               };
15             });
16           };
17           const workLoop = async deadline
=> {
```

```
18            // 有任务，并且当前帧还没结束
19            while (count < chunks.length
    && deadline.timeRemaining() > 1) {
20              await
    appendToSpark(chunks[count].file);
21              count++;
22              // 没有了 计算完毕
23              if (count < chunks.length)
    {
24                // 计算中
25                this.hashProgress =
    Number(
26                  ((100 * count) /
    chunks.length).toFixed(2)
27                );
28                //
    console.log(this.hashProgress)
29              } else {
30                // 计算完毕
31                this.hashProgress = 100;
32                resolve(spark.end());
33              }
34            }
35
     window.requestIdleCallback(workLoop);
36          };
```

```
37
    window.requestIdleCallback(workLoop);
38      });
39    },
40
```

抽样hash

布隆过滤器思想




```
1 抽样md5: 1028.006103515625ms
2
3 全量md5: 21745.13916015625ms
```

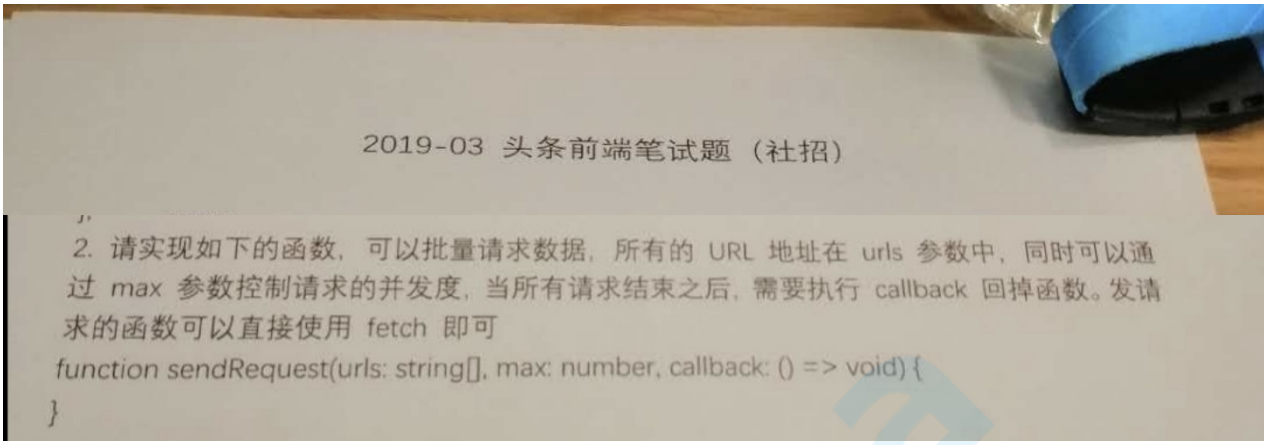
```
1     async calculateHashSample() {
2       return new Promise(resolve => {
```

```
3         const spark = new
SparkMD5.ArrayBuffer();
4         const reader = new
FileReader();
5         const file =
this.container.file;
6         // 文件大小
7         const size =
this.container.file.size;
8         let offset = 2 * 1024 * 1024;
9
10        let chunks = [file.slice(0,
offset)];
11
12        // 前面100K
13
14        let cur = offset;
15        while (cur < size) {
16          // 最后一块全部加进来
17          if (cur + offset >= size) {
18            chunks.push(file.slice(cur,
cur + offset));
19          } else {
20            // 中间的 前中后去两个字节
21            const mid = cur + offset /
2;
22            const end = cur + offset;
```

```
23              chunks.push(file.slice(cur, cur + 2));
24              chunks.push(file.slice(mid, mid + 2));
25              chunks.push(file.slice(end - 2, end));
26            }
27            // 前取两个字节
28            cur += offset;
29          }
30          // 拼接
31          reader.readAsArrayBuffer(new Blob(chunks));
32          reader.onload = e => {
33
 spark.append(e.target.result);
34
35            resolve(spark.end());
36          };
37        });
38      }
39
```

请求并发数控制和重试

这个单独也是一个面试题




```
+async sendRequest(forms, max=4) {
+  return new Promise(resolve => {
+    const len = forms.length;
+    let idx = 0;
+    let counter = 0;
+    const start = async ()=> {
+      // 有请求，有通道
+      while (idx < len && max > 0) {
+        max--; // 占用通道
+        console.log(idx, "start");
+        const form = forms[idx].form;
+        const index = forms[idx].index;
+        idx++
+        request({
```

```
16 +            url: '/upload',
17 +            data: form,
18 +            onProgress:
   this.createProgresshandler(this.chunks[
   index]),
19 +            requestList:
   this.requestList
20 +          }).then(() => {
21 +            max++; // 释放通道
22 +            counter++;
23 +            if (counter === len) {
24 +              resolve();
25 +            } else {
26 +              start();
27 +            }
28 +          });
29 +        }
30 +      }
31 +      start();
32 +    });
33 +}
34
35 async uploadChunks(uploadedList = []) {
36   // 这里一起上传，碰见大文件就是灾难
37   // 没被hash计算打到，被一次性的tcp链接把浏
   览器稿挂了
```

```
38    // 异步并发控制策略，我记得这个也是头条一个面试题
39    // 比如并发量控制成4
40    const list = this.chunks
41      .filter(chunk => uploadedList.indexOf(chunk.hash) == -1)
42      .map(({ chunk, hash, index }, i) => {
43        const form = new FormData();
44        form.append("chunk", chunk);
45        form.append("hash", hash);
46        form.append("filename", this.container.file.name);
47        form.append("fileHash", this.container.hash);
48        return { form, index };
49      })
50  -     .map(({ form, index }) =>
51  -       request({
52  -           url: "/upload",
53  -         data: form,
54  -         onProgress: this.createProgresshandler(this.chunks[index]),
55  -         requestList: this.requestList
56  -       })
57  -     );
```

```
58 -    // 直接全量并发
59 -    await Promise.all(list);
60       // 控制并发
61 +    const ret =  await
   this.sendRequest(list,4)
62
63     if (uploadedList.length + list.length
   === this.chunks.length) {
64       // 上传和已经存在之和 等于全部的再合并
65       await this.mergeRequest();
66     }
67 },
68
```

慢启动策略

TCP拥塞控制的问题 其实就是根据当前网络情况，动态调整切片的大小

1. `chunk`中带上`size`值，不过进度条数量不确定了，修改`createFileChunk`， 请求加上时间统计)
2. 比如我们理想是30秒传递一个
3. 初始大小定为1M，如果上传花了10秒，那下一个区块大小变成3M
4. 如果上传花了60秒，那下一个区块大小变成500KB

以此类推

5. 并发+慢启动的逻辑有些复杂，我自己还没绕明白，囧所以先一次只传一个切片，来演示这个逻辑，新建一个`handleUpload1`函数

```
async handleUpload1(){
      // @todo数据缩放的比率 可以更平缓
      // @todo 并发+慢启动

      // 慢启动上传逻辑
      const file = this.container.file
      if (!file) return;
      this.status = Status.uploading;
      const fileSize = file.size
      let offset = 1024*1024
      let cur = 0
      let count =0
      this.container.hash = await this.calculateHashSample();

      while(cur<fileSize){
        // 切割offfset大小
        const chunk = file.slice(cur, cur+offset)
        cur+=offset
```

```
19          const chunkName =
this.container.hash + "-" + count;
20          const form = new FormData();
21          form.append("chunk", chunk);
22          form.append("hash", chunkName);
23          form.append("filename",
file.name);
24          form.append("fileHash",
this.container.hash);
25          form.append("size",
chunk.size);
26
27          let start = new
Date().getTime()
28          await request({ url:
'/upload',data: form })
29          const now = new
Date().getTime()
30
31          const time = ((now -
start)/1000).toFixed(4)
32          let rate = time/30
33          // 速率有最大2和最小0.5
34          if(rate<0.5) rate=0.5
35          if(rate>2) rate=2
36          // 新的切片大小等比变化
```

```
37          console.log(`切片${count}大小是${this.format(offset)},耗时${time}秒，是30秒的${rate}倍，修正大小为${this.format(offset/rate)}`)
38          // 动态调整offset
39          offset = parseInt(offset/rate)
40          // if(time)
41          count++
42        }
43      }
44
45
```

```
1 切片0大小是1024.00KB,耗时13.2770秒，是30秒的0.5倍，修正大小为2.00MB
2 切片1大小是2.00MB,耗时25.4130秒，是30秒的0.8471倍，修正大小为2.36MB
3 切片2大小是2.36MB,耗时14.1260秒，是30秒的0.5倍，修正大小为4.72MB
4
```

碎片清理

```
1 // 为了方便测试，我改成每5秒扫一次， 过期1钟的删除做演示
2 const fse = require('fs-extra')
3 const path = require('path')
```

```
4  const schedule = require('node-
   schedule')
5
6
7  // 空目录删除
8  function remove(file,stats){
9      const now = new Date().getTime()
10     const offset = now - stats.ctimeMs
11     if(offset>1000*60){
12         // 大于60秒的碎片
13         console.log(file,'过期了，浪费空间
   的玩意，删除')
14         fse.unlinkSync(file)
15     }
16 }
17
18 async function scan(dir,callback){
19     const files = fse.readdirSync(dir)
20     files.forEach(filename=>{
21         const fileDir =
   path.resolve(dir,filename)
22         const stats =
   fse.statSync(fileDir)
23         if(stats.isDirectory()){
24             return scan(fileDir,remove)
25         }
26         if(callback){
```

```
            callback(fileDir,stats)
        }
    })
}
// *    *    *    *    *    *
// ┬    ┬    ┬    ┬    ┬    ┬
// |    |    |    |    |    |
// |    |    |    |    |    └ day of week (0 - 7) (0 or 7 is Sun)
// |    |    |    |    └──── month (1 - 12)
// |    |    |    └─────────── day of month (1 - 31)
// |    |    └──────────────── hour (0 - 23)
// |    └───────────────────── minute (0 - 59)
// └────────────────────────── second (0 - 59, OPTIONAL)
let start = function(UPLOAD_DIR){
    // 每5秒
    schedule.scheduleJob("*/5 * * * * *",function(){
        console.log('开始扫描')
        scan(UPLOAD_DIR)
    })
}
```

```
47 exports.start = start
48
49
```

```
开始扫描
/upload/target/625c.../625c...-0 过期了，删除
/upload/target/625c.../625c...-1 过期了，删除
/upload/target/625c.../625c...-10 过期了，删除
/upload/target/625c.../625c...-11 过期了，删除
/upload/target/625c.../625c...-12 过期了，删除

```

后续进阶思考

留几个思考题，下次写文章再实现 方便继续蹭热度

1. requestIdleCallback兼容性，如何自己实现一个

    1. react也是自己写的调度逻辑，以后有机会写个文章介绍
    2. React自己实现的requestIdleCallback
2. 并发+慢启动配合
3. 抽样hash+全量哈希+时间切片配合
4. 大文件切片下载
    1. 一样的切片逻辑，通过axios.head请求获取content-Length
    2. 使用http的Range这个header就可以切片下载了，其他逻辑和上传差不多
5. 小的体验优化
    1. 比如离开页面的提醒 等等小tips
6. 慢启动的变化应该更平滑，比如使用三角函数，把变化率平滑的限制在0.5~1.5之间
7. websocket推送进度
8. 文件碎片分机器存储
9. 文件碎片备份
10. cdn

疫情期间充电

学习源码