批量文件上传遇到并发问题咋整

并发模式下，如何控制并发数量

并发

先明确一下概念：同一时间段内，两个或多个事件或活动看似同时发生，但实际上这些事件或活动是在不同的时刻依次进行的。

生活中并发的场景并不少见，多人行走的道路、抢票、网站访问等。

为什么需要控制并发？首先明确一个事实，肯定是有某条”通道“上的压力才需要控制流动的数量。当流动的数量足够多时，通道就会拥挤，甚至瘫痪，所以需要控制通道的流动数量，这是我对并发来由的简单理解。

如何控制并发？

控制并发说到底就是要控制流动的数量。这几天我刚好遇到一个并发上传的问题，借此机会分享一下我解决问题的方法。

我遇到的场景是，使用分片上传的方式批量上传文件，这个时候会同时产生很多文件流上传到oss的请求，导致中间其他请求延迟和网站崩溃。

同个域名的请求数量大概是6个左右（不同浏览器不同），超出的就要等待，造成延迟。请求数量太多也会导致浏览器卡顿和性能下降，响应速度变慢，造成其他请求延迟或者网站崩溃。在业务场景里边，一般上传完会做一些像后端同步上传文件地址还有其他信息的操作，这些请求延迟太久可能失败，其次对于用户体验也不友好如果想要在上传完成后做点什么的话。

一开始的做法是减少切片分片数量，但是批量上传的文件多了还会有问题，不仅慢，而且只要其中一个上传有问题，其他没上传完成的都会因为网站崩溃而需要重新上传。

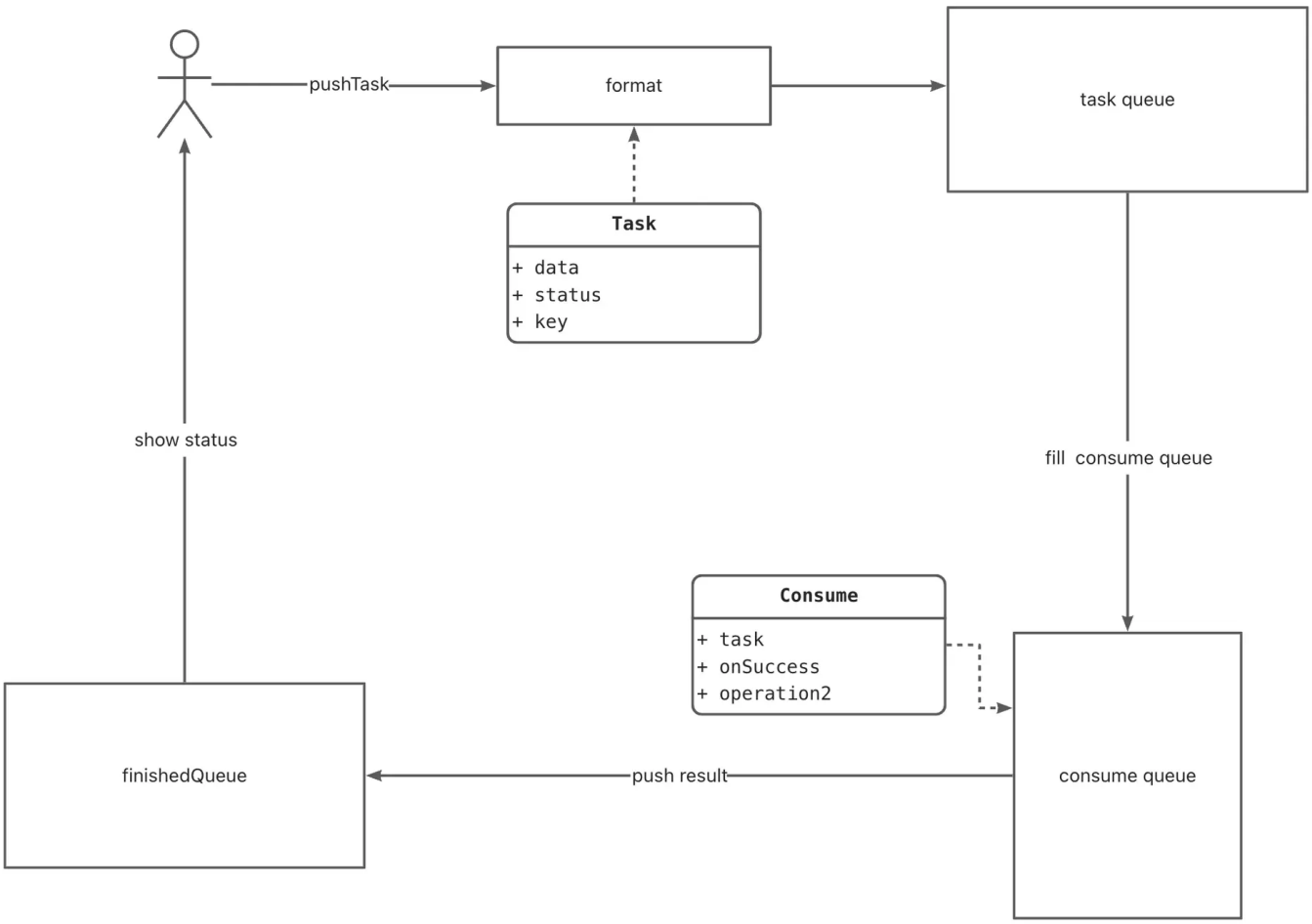
后边又将允许批量上传的数量从10改到5个，没想到还是有问题，总不能让我减少到2个吧，2个确实可以了，不过用户就得上传完2个再上传两个……，多麻烦，能用技术来解决的尽量用技术解决。

后来我思考了一下，如果能在批量上传多个文件的情况下，两个两个上传就好了。咦，网盘上传文件或者软件商店下载软件不都是这种限流的做法吗，OK，应该可行。

我封装了一个控制并发的服务，接下来讲讲我写的思路。

代码思路实践

我画了张流程图如下：



用户推送任务到我们的消费队列的时候，会先进行格式化为一个带状态和有唯一标识的对象，然后再推送进入我们的任务队列。

任务队列并发进行肯定就会导致我们上面所说的并发问题，所以这里是搭建了一个执行队列进行“限流”，执行队列会按用户限定的最多执行数量去并发消费存储队列推送的任务。

消费队列会在消费完成之后更新任务状态并推送到完成队列，完成队列中可以查看已完成任务的状态。

结果队列会将结果依次推送给用户。

大致的demo如下：

```

class ConsumeQueue {
  queue: IQueueItem[] = [] // 任务存储队列
  consumingQueue: IConsumeItem[] = [] // 执行队列
  perConsumeCount: number // 执行队列单次并发执行的任务数量
  consume: (item: any, onSUccess, onError) => void // 消费函数，需要用户传的
  finishedQueue: IConsumeItem[] = [] // 完成队列
  isAutoStart: boolean // 是否自动开始，如果有控制开始时机可以使用
  private status: 'idle' | 'working' | 'error' // 类的状态，构思的时候写的，这里没用到

```

```
11  constructor(options: ConsumeQueueOptions) {
12    const { perConsumeCount = 2,consume, isAutoStart = true } = options
13    if (!consume) {
14      console.error('请初始化消费函数')
15      return
16    }
17    this.perConsumeCount = perConsumeCount
18    this.consume = consume
19    this.isAutoStart = isAutoStart
20  }
21
22  pushTasks(items: IQueueItem[]) {
23          // 用户推送数据项时，将每个数据项任务格式化
24          const formattedItems = items.map(item => this.formatTask(item))
25    this.queue = [...formattedItems, ...this.queue]
26    if (this.isAutoStart) {
27      this.fillConsumeQueue()
28    }
29  }
30
31  private fillConsumeQueue() {
32          // 将执行队列填满
33    if (this.consumingQueue.length < this.perConsumeCount) {
34      for (let i = this.consumingQueue.length; i < this.perConsumeCount; i++) {
35        if (this.queue.length) {
36          const task = this.queue.shift()
37          this.consumingQueue.push(task)
38        } else break
39      }
40    }
41    this.consumeTasks()
42  }
43
44  private consumeTasks() {
45    this.consumingQueue.forEach((item, index) => {
46      if (item.status === 'wait') {
47        item.status = 'loading'
48        this.consume(
49          item.data,
50          () => this.onSuccess(item.key),
51          () => this.onError(item.key)
52        )
53      }
54    })
55  }
56
57        // 暂停或取消
```

```
58    cancel() {
59            // 本次暂时没用到
60    }
61
62    private formatTask(item: IQueueItem): IConsumeItem {
63            // 格式化用户推送的任务
64      return {
65        data: item,
66        status: 'wait', // wait failed success
67        key: String(Date.now()),
68      }
69    }
70
71    private onSuccess(key: string) {
72      const item = this.consumingQueue.find((item) => item.key === key)!
73      item.status = 'success'
74      this.finishedQueue.push(item)
75      this.consumingQueue = this.consumingQueue.filter((item) => item.key !==
    key)
76      console.log('任务执行成功', key)
77      console.log(`库存剩余${this.queue.length}个`)
78      console.log(`有${this.consumingQueue.length}个任务正在执行中`)
79      if (this.queue.length) {
80        this.fillConsumeQueue()
81      } else if (!this.consumingQueue.length) {
82        console.log('所有任务均已处理完毕，处理结果：', this.finishedQueue)
83      }
84    }
85
86    private onError(key: string) {
87      console.log('任务执行失败', key)
88      const item = this.consumingQueue.find((item) => item.key === key)!
89      this.finishedQueue.push(item)
90      this.consumingQueue = this.consumingQueue.filter((item) => item.key !==
    key)
91      this.fillConsumeQueue()
92    }
93
94  }
95
96  export default ConsumeQueue
```