假如有几十个请求，如何去控制并发?

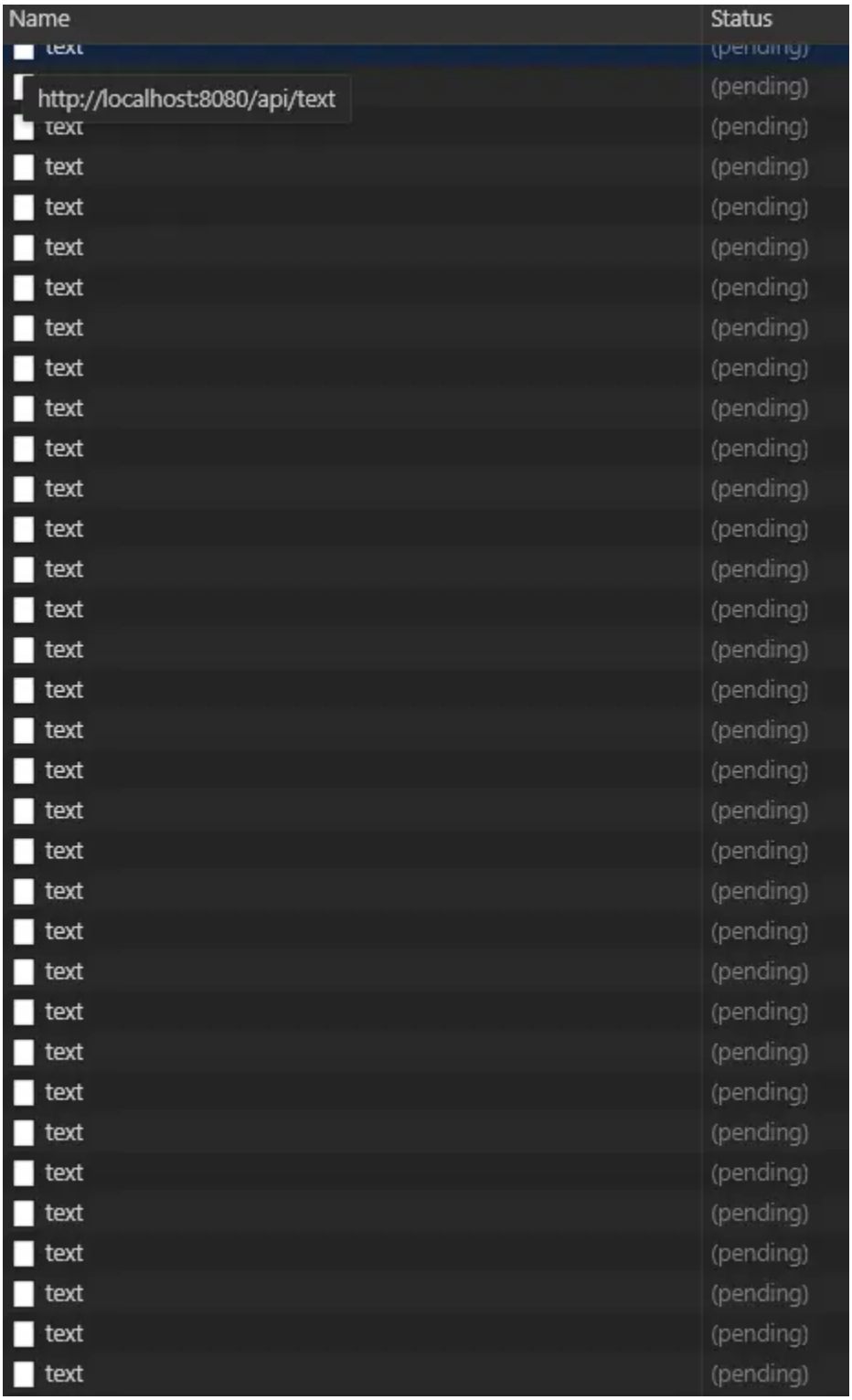
面试官：看你简历上做过**图片或文件批量下载**，那么假如我一次性下载几十个，如何去控制并发请求的?
让我想想，额~， 选中ID，循环请求？，八嘎！肯定不是那么沙雕的做法，这样做服务器直接崩溃啦！突然灵光一现，请求池！！！
我：利用Promise模拟任务队列，从而实现请求池效果。
面试官：大佬!

废话不多说，正文开始：

众所周知，浏览器发起的请求最大并发数量一般都是 `6~8` 个，这是因为浏览器会限制同一域名下的并发请求数量，以避免对服务器造成过大的压力。

首先让我们来模拟大量请求的场景

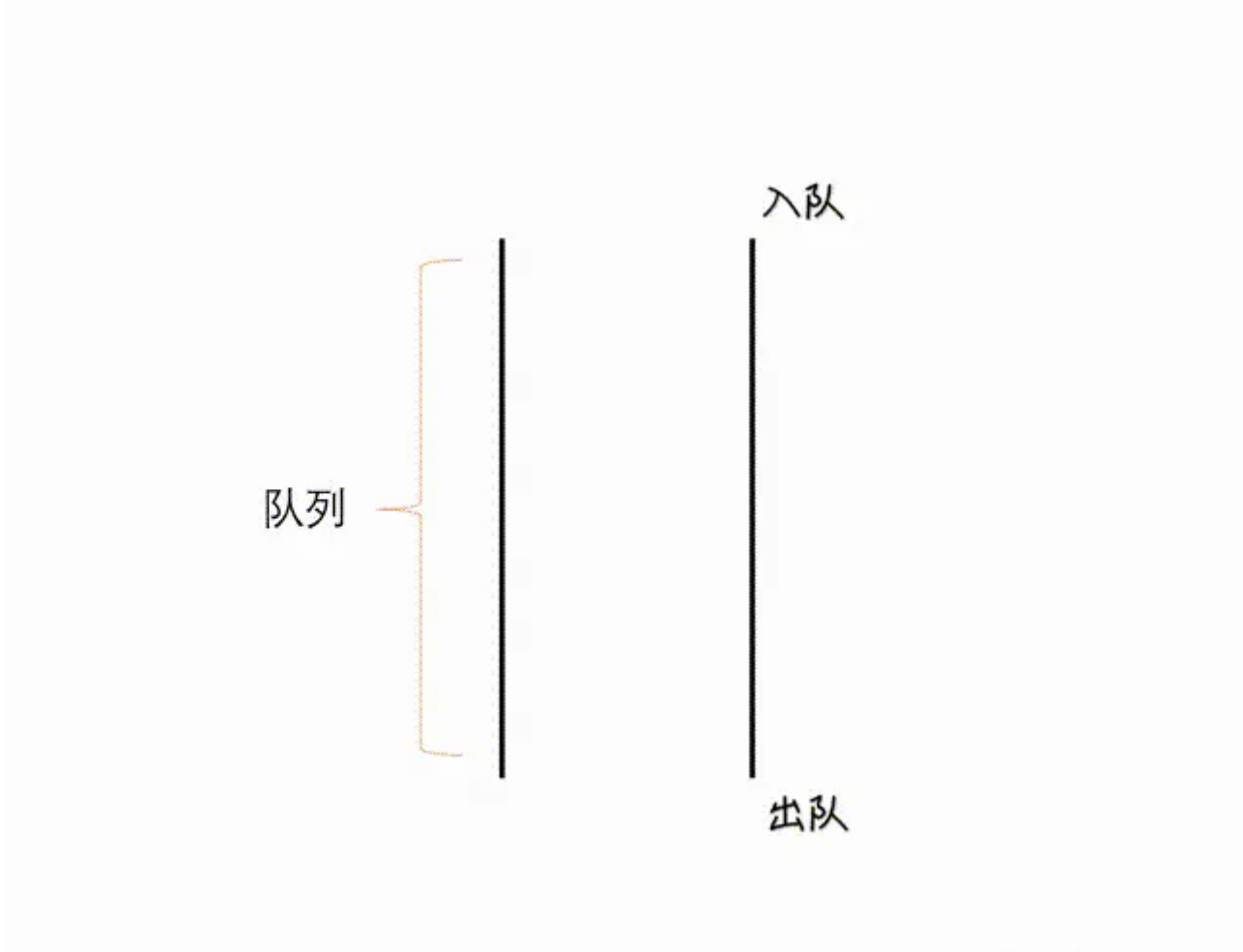
```
const ids = new Array(100).fill('')

console.time()
for (let i = 0; i < ids.length; i++) {
  console.log(i)
}
console.timeEnd()
```



一次性并发上百个请求，要是配置低一点，又或者带宽不够的服务器，直接宕机都有可能，所以我们前端这边是需要控制的并发数量去为服务器排忧解难。

什么是队列?

先进先出就是队列，`push`一个的同时就会有一个被`shift`。我们看下面的动图可能就会更加的理解：



我们接下来的操作就是要模拟上图的队列行为。

定义请求池主函数函数

```
export const handQueue = (
  reqs // 请求数量
) => {}
```

接受一个参数 `reqs` ，它是一个数组，包含需要发送的请求。函数的主要目的是对这些请求进行队列管理，确保并发请求的数量不会超过设定的上限。

定义dequeue函数

```
const dequeue = () => {
  while (current < concurrency && queue.length) {
    current++;
    const requestPromiseFactory = queue.shift() // 出列
    requestPromiseFactory()
      .then(() => { // 成功的请求逻辑
      })
      .catch(error => { // 失败
```

```
      console.log(error)
    })
    .finally(() => {
      current--
      dequeue()
    });
  }
}
```

这个函数用于从请求池中取出请求并发送。它在一个循环中运行，直到当前并发请求数 `current` 达到最大并发数 `concurrency` 或请求池 `queue` 为空。对于每个出队的请求，它首先增加 `current` 的值，然后调用请求函数 `requestPromiseFactory` 来发送请求。当请求完成（无论成功还是失败）后，它会减少 `current` 的值并再次调用 `dequeue` ，以便处理下一个请求。

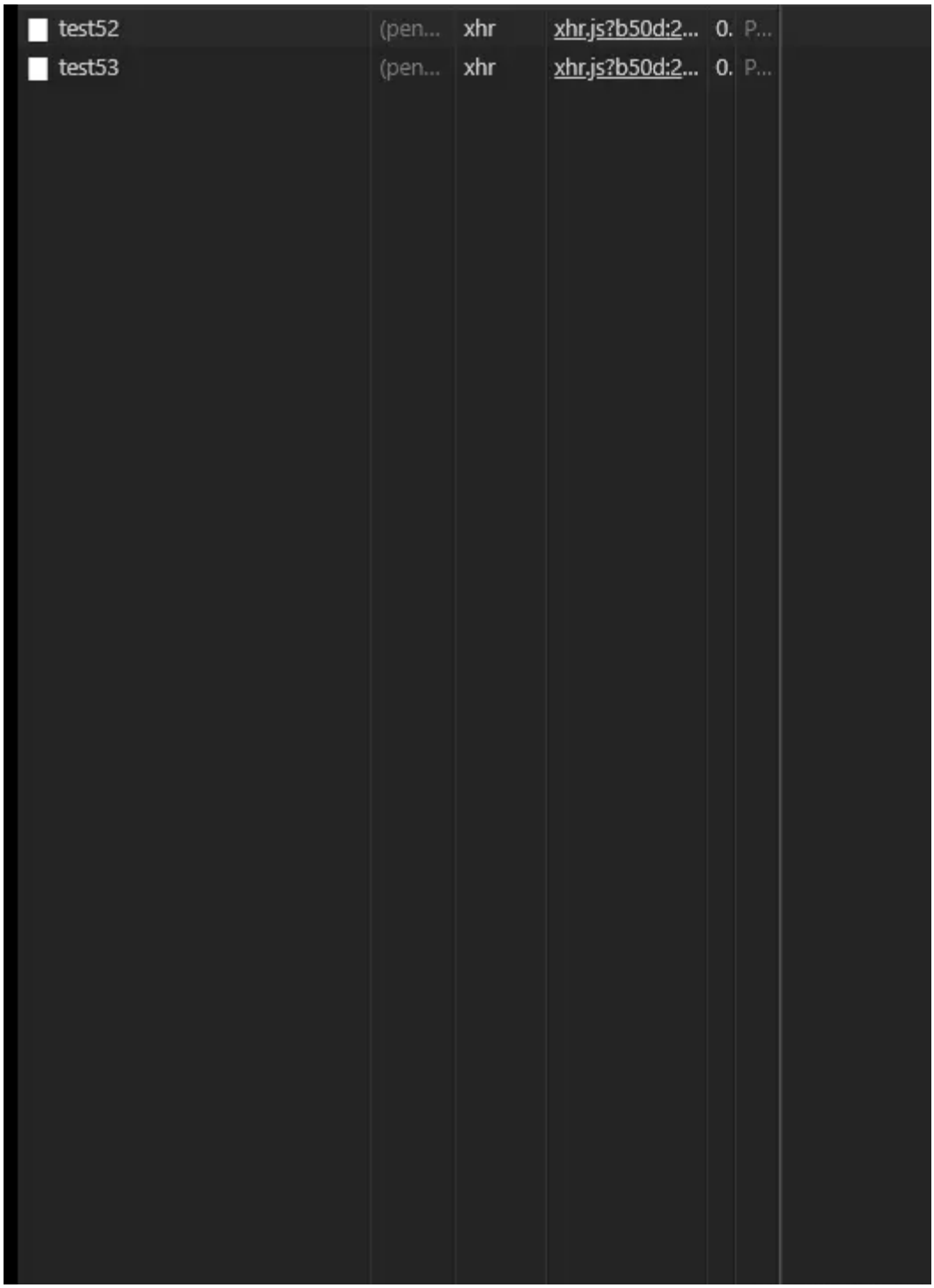
定义返回请求入队函数

```
return (requestPromiseFactory) => {
  queue.push(requestPromiseFactory) // 入队
  dequeue()
}
```

函数返回一个函数，这个函数接受一个参数 `requestPromiseFactory` ，表示一个返回Promise的请求工厂函数。这个返回的函数将请求工厂函数加入请求池 `queue` ，并调用 `dequeue` 来尝试发送新的请求，当然也可以自定义axios，利用 `Promise.all` 统一处理返回后的结果。

实验

```
const enqueue = requestQueue(6) // 设置最大并发数
for (let i = 0; i < reqs.length; i++) {  // 请求
  enqueue(() => axios.get('/api/test' + i))
}
```



我们可以看到如上图所示,请求数确实被控制了，只有有请求响应成功的同时才会有新的请求进来，极大的降低里服务器的压力，后端的同学都只能喊**6**。

整合代码

```
import axios from 'axios'

export const handQueue = (
  reqs // 请求总数
) => {
  reqs = reqs || []

```

```
8
9   const requestQueue = (concurrency) => {
10    concurrency = concurrency || 6 // 最大并发数
11    const queue = [] // 请求池
12    let current = 0
13
14    const dequeue = () => {
15      while (current < concurrency && queue.length) {
16        current++;
17        const requestPromiseFactory = queue.shift() // 出列
18        requestPromiseFactory()
19          .then(() => { // 成功的请求逻辑
20          })
21          .catch(error => { // 失败
22            console.log(error)
23          })
24          .finally(() => {
25            current--
26            dequeue()
27          });
28      }
29
30    }
31
32    return (requestPromiseFactory) => {
33      queue.push(requestPromiseFactory) // 入队
34      dequeue()
35    }
36
37  }
38
39  const enqueue = requestQueue(6)
40
41  for (let i = 0; i < reqs.length; i++) {
42
43    enqueue(() => axios.get('/api/test' + i))
44  }
45 }
```