JavaScript 中如何实现大文件并发上传?

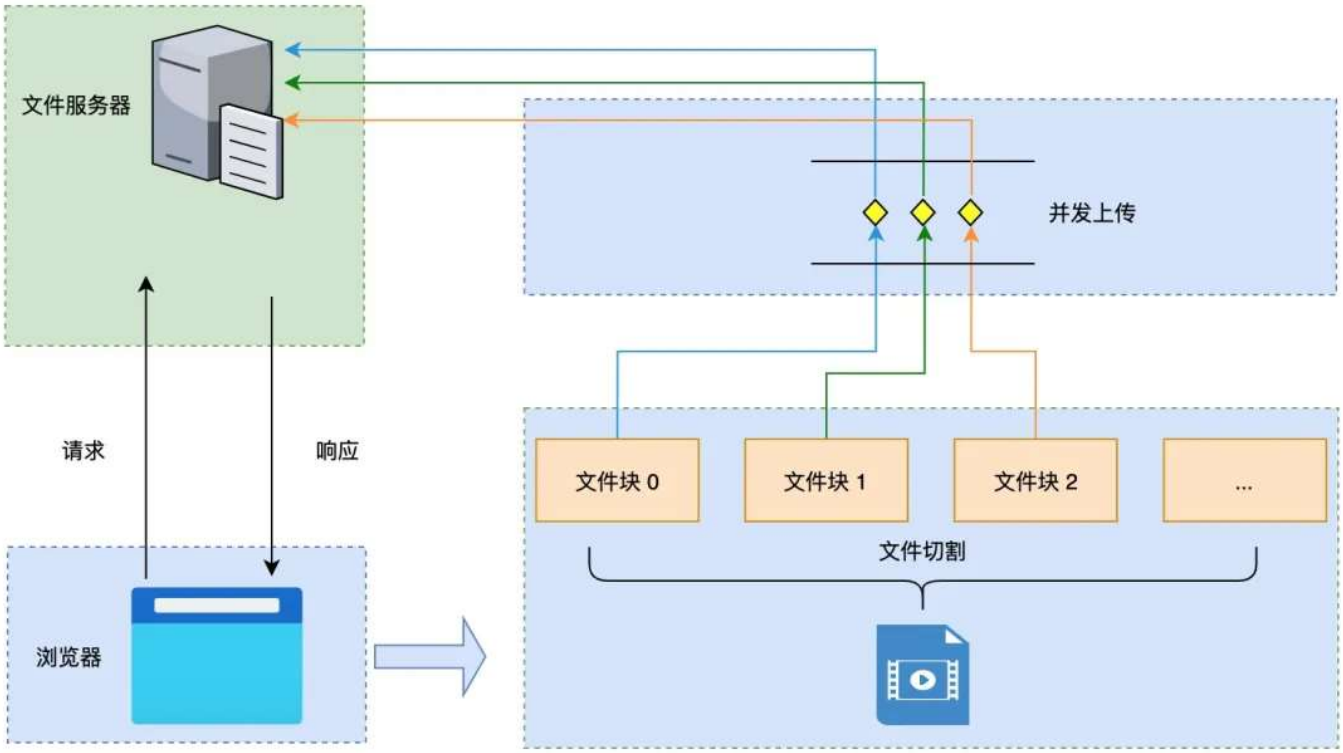
原创 阿宝哥 全栈修仙之路 6月1日

全栈修仙之路
专注分享 TS、Vue3、前端架构和源码解析等技术干货。
125篇原创内容

公众号

在 **JavaScript 中如何实现并发控制？** 这篇文章中，阿宝哥详细分析了 async-pool 这个库如何利用 `Promise.all` 和 `Promise.race` 函数实现异步任务的并发控制。之后，阿宝哥通过 **JavaScript 中如何实现大文件并行下载？** 这篇文章介绍了 async-pool 这个库的实际应用。

本文将介绍如何利用 async-pool 这个库提供的 `asyncPool` 函数来实现大文件的并发上传。相信有些小伙伴已经了解大文件上传的解决方案，在上传大文件时，为了提高上传的效率，我们一般会使用 Blob.slice 方法对大文件按照指定的大小进行切割，然后通过多线程进行分块上传，等所有分块都成功上传后，再通知服务端进行分块合并。



看完上图相信你对大文件上传的方案，已经有了一定的了解。接下来，我们先来介绍 Blob 和 File 对象。

一、Blob 和 File 对象

1.1 Blob 对象

Blob（Binary Large Object）表示二进制类型的大对象。在数据库管理系统中，将二进制数据存储为一个单一个体的集合。Blob 通常是影像、声音或多媒体文件。在 **JavaScript** 中 **Blob** 类型的对象表示不可变的类似文件对象的原始数据。 为了更直观的感受 Blob 对象，我们先来使用 Blob 构造函数，创建一个 myBlob 对象，具体如下图所示：

如你所见，myBlob 对象含有两个属性：size 和 type。其中 `size` 属性用于表示数据的大小（以字节为单位）， `type` 是 MIME 类型的字符串。 `Blob` 由一个可选的字符串 `type` （通常是 MIME 类型）和 `blobParts` 组成：

Blob 表示的不一定是 JavaScript 原生格式的数据。比如 File 接口基于 Blob，继承了 Blob 的功能并将其扩展使其支持用户系统上的文件。

1.2 File 对象

通常情况下， File 对象是来自用户在一个 `<input>` 元素上选择文件后返回的 FileList 对象，也可以是来自由拖放操作生成的 DataTransfer 对象，或者来自 HTMLCanvasElement 上的 mozGetAsFile() API。

File 对象是特殊类型的 Blob，且可以用在任意的 Blob 类型的上下文中。比如说 FileReader、URL.createObjectURL() 及 XMLHttpRequest.send() 都能处理 Blob 和 File。在大文件上传的场景中，

我们将使用 Blob.slice 方法对大文件按照指定的大小进行切割，然后对分块进行并行上传。接下来，我们来看一下具体如何实现大文件上传。

二、如何实现大文件上传

为了让大家能够更好地理解后面的内容，我们先来看一下整体的流程图：

了解完大文件上传的流程之后，我们先来定义上述流程中涉及的一些辅助函数。

2.1 定义辅助函数

2.1.1 定义 calcFileMD5 函数

顾名思义 `calcFileMD5` 函数，用于计算文件的 MD5 值（数字指纹）。在该函数中，我们使用 FileReader API 分块读取文件的内容，然后通过 `spark-md5` 这个库提供的方法来计算文件的 MD5 值。

```
function calcFileMD5(file) {
  return new Promise((resolve, reject) => {
    let chunkSize = 2097152, // 2M
      chunks = Math.ceil(file.size / chunkSize),
      currentChunk = 0,
      spark = new SparkMD5.ArrayBuffer(),
      fileReader = new FileReader();

      fileReader.onload = (e) => {
        spark.append(e.target.result);
        currentChunk++;
        if (currentChunk < chunks) {
          loadNext();
        } else {
          resolve(spark.end());
        }
      };

      fileReader.onerror = (e) => {
        reject(fileReader.error);
        reader.abort();
      };

      function loadNext() {
        let start = currentChunk * chunkSize,
          end = start + chunkSize >= file.size ? file.size : start + chunkSize;
        fileReader.readAsArrayBuffer(file.slice(start, end));
      }
      loadNext();
  });
}
```

2.1.2 定义 asyncPool 函数

在 **JavaScript 中如何实现并发控制？** 这篇文章中，我们介绍了 `asyncPool` 函数，它用于实现异步任务的并发控制。该函数接收 3 个参数：

- `poolLimit` （数字类型）：表示限制的并发数；
- `array` （数组类型）：表示任务数组；
- `iteratorFn` （函数类型）：表示迭代函数，用于实现对每个任务项进行处理，该函数会返回一个 Promise 对象或异步函数。

```
async function asyncPool(poolLimit, array, iteratorFn) {
  const ret = []; // 存储所有的异步任务
  const executing = []; // 存储正在执行的异步任务
```

```
  for (const item of array) {
    // 调用iteratorFn函数创建异步任务
    const p = Promise.resolve().then(() => iteratorFn(item, array));
    ret.push(p); // 保存新的异步任务

    // 当poolLimit值小于或等于总任务个数时，进行并发控制
    if (poolLimit <= array.length) {
      // 当任务完成后，从正在执行的任务数组中移除已完成的任务
      const e = p.then(() => executing.splice(executing.indexOf(e), 1));
      executing.push(e); // 保存正在执行的异步任务
      if (executing.length >= poolLimit) {
        await Promise.race(executing); // 等待较快的任务执行完成
      }
    }
  }
  return Promise.all(ret);
}
```

2.1.3 定义 checkFileExist 函数

`checkFileExist` 函数用于检测文件是否已经上传过了，如果已存在则秒传，否则返回已上传的分块 ID 列表：

```
function checkFileExist(url, name, md5) {
  return request.get(url, {
    params: {
      name,
      md5,
    },
  }).then((response) => response.data);
}
```

在 `checkFileExist` 函数中使用到的 `request` 对象是 `Axios` 实例，通过 `axios.create` 方法来创建：

```
const request = axios.create({
  baseURL: "http://localhost:3000/upload",
  timeout: 10000,
});
```

有了 `request` 对象之后，我们就可以轻易地发送 HTTP 请求。在 `checkFileExist` 函数内部，我们会发起一个 GET 请求，同时携带的查询参数是文件名（name）和文件的 MD5 值。

2.1.4 定义 upload 函数

当调用 `checkFileExist` 函数之后，如果发现文件尚未上传或者只上传完部分分块的话，就会继续调用 `upload` 函数来执行上传任务。在 `upload` 函数内，我们使用了前面介绍的 `asyncPool` 函数来实现异步任务的并发控制，具体如下所示：

```
function upload({
  url, file, fileMd5,
  fileSize, chunkSize, chunkIds,
  poolLimit = 1,
}) {
  const chunks = typeof chunkSize === "number" ? Math.ceil(fileSize / chunkSize) : 1;
  return asyncPool(poolLimit, [...new Array(chunks).keys()], (i) => {
    if (chunkIds.indexOf(i + "") !== -1) { // 已上传的分块直接跳过
      return Promise.resolve();
    }
    let start = i * chunkSize;
    let end = i + 1 == chunks ? fileSize : (i + 1) * chunkSize;
    const chunk = file.slice(start, end); // 对文件进行切割
    return uploadChunk({
      url,
      chunk,
      chunkIndex: i,
      fileMd5,
      fileName: file.name,
    });
  });
}
```

对于切割完的文件块，会通过 `uploadChunk` 函数，来执行实际的上传操作：

```
function uploadChunk({ url, chunk, chunkIndex, fileMd5, fileName }) {
  let formData = new FormData();
  formData.set("file", chunk, fileMd5 + "-" + chunkIndex);
  formData.set("name", fileName);
  formData.set("timestamp", Date.now());
```

```
  return request.post(url, formData);
}
```

2.1.5 定义 concatFiles 函数

当所有分块都上传完成之后，我们需要通知服务端执行分块合并操作，这里我们定义了 `concatFiles` 函数来实现该功能：

```
function concatFiles(url, name, md5) {
  return request.get(url, {
    params: {
      name,
      md5,
    },
  });
}
```

2.1.6 定义 uploadFile 函数

在前面已定义辅助函数的基础上，我们就可以根据大文件上传的整体流程图来实现一个 `uploadFile` 函数：

```
async function uploadFile() {
  if (!uploadFileEle.files.length) return;
  const file = uploadFileEle.files[0]; // 获取待上传的文件
  const fileMd5 = await calcFileMD5(file); // 计算文件的MD5
  const fileStatus = await checkFileExist(  // 判断文件是否已存在
    "/exists",
    file.name, fileMd5
  );
  if (fileStatus.data && fileStatus.data.isExists) {
    alert("文件已上传[秒传]");
    return;
  } else {
    await upload({
      url: "/single",
      file, // 文件对象
      fileMd5, // 文件MD5值
      fileSize: file.size, // 文件大小
      chunkSize: 1 * 1024 * 1024, // 分块大小
```

```
      chunkIds: fileStatus.data.chunkIds, // 已上传的分块列表
      poolLimit: 3, // 限制的并发数
     });
  }
  await concatFiles("/concatFiles", file.name, fileMd5);
}
```

2.2 大文件并发上传示例

定义完 `uploadFile` 函数，要实现大文件并发上传的功能就很简单了，具体代码如下所示：

```
<!DOCTYPE html>
<html lang="zh-CN">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>大文件并发上传示例（阿宝哥）</title>
    <script src="https://cdn.bootcdn.net/ajax/libs/axios/0.21.1/axios.min.js"></script>
    <script src="https://cdn.bootcdn.net/ajax/libs/spark-md5/3.0.0/spark-md5.min.js"></script>
  </head>
  <body>
    <input type="file" id="uploadFile" />
    <button id="submit" onclick="uploadFile()">上传文件</button>
    <script>
      const uploadFileEle = document.querySelector("#uploadFile");

      const request = axios.create({
        baseURL: "http://localhost:3000/upload",
        timeout: 10000,
      });

      async function uploadFile() {
        if (!uploadFileEle.files.length) return;
     const file = uploadFileEle.files[0]; // 获取待上传的文件
     const fileMd5 = await calcFileMD5(file); // 计算文件的MD5
        // ...
      }
      // 省略其他函数
    </script>
```

```
  </body>
</html>
```

由于完整的示例代码内容比较多，阿宝哥就不放具体的代码了。感兴趣的小伙伴，可以访问以下地址浏览客户端和服务器端代码。

> 完整的示例代码（代码仅供参考，可根据实际情况进行调整）：
> https://gist.github.com/semlinker/b211c0b148ac9be0ac286b387757e692

最后我们来看一下大文件并发上传示例的运行结果：

三、总结

本文介绍了在 JavaScript 中如何利用 async-pool 这个库提供的 `asyncPool` 函数，来实现大文件的并发上传。此外，文中我们也使用了 spark-md5 这个库来计算文件的数字指纹，如果你数字指纹感兴趣的话，可以阅读 数字指纹有什么用？赶紧来了解一下 这篇文章。

由于篇幅有限，阿宝哥并未介绍服务端的具体代码。其实在做文件分块合并时，阿宝哥是以流的形式进行合并，感兴趣的小伙伴可以自行阅读一下相关代码。如果有遇到不清楚的地方，欢迎随时跟阿宝哥交流哟。

四、参考资源

- 你不知道的 **Blob**
- MDN - File

- MDN - ArrayBuffer
- MDN - HTTP请求范围
- **JavaScript 中如何实现并发控制？**

期待你加入“前端面试交流与内推群”

全栈修仙之路

专注分享 TS、Vue3、前端架构和源码解析等技术干货。

125篇原创内容

公众号

喜欢此内容的人还喜欢

8 个漂亮的 vue.js 进度条组件

前端先锋

鲜为人知的Python 5种高级特征

Python那些事

TypeScript在项目开发中的实践心得

前端工匠