# JavaScript 中如何实现大文件并行下载?

原创 阿宝哥 全栈修仙之路 4月19日

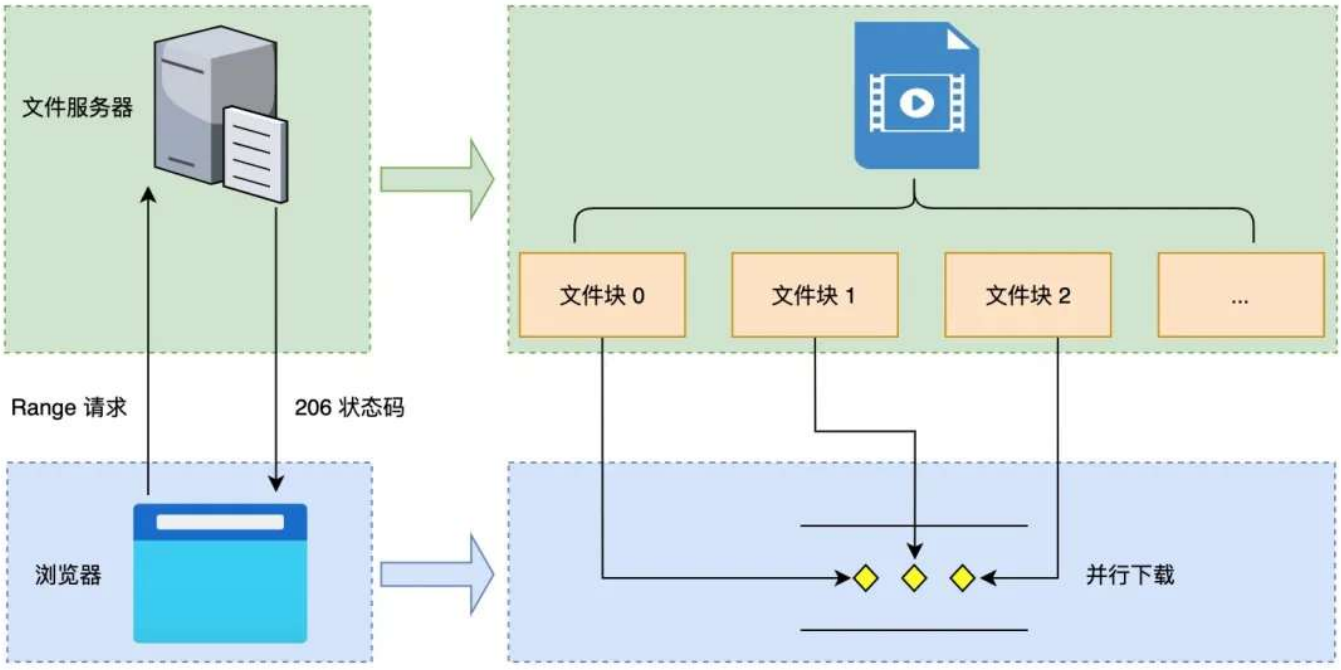
全栈修仙之路
专注分享 TS、Vue3、前端架构和源码解析等技术干货。
125篇原创内容

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。本文阿宝哥将介绍如何利用 async-pool 这个库提供的 `asyncPool` 函数来实现大文件的并行下载。

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，然后在开启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。

那么对大文件下载来说，我们能否采用类似的思想呢？在服务端支持 `Range` 请求首部的条件下，我们也是可以实现多线程分块下载的功能，具体如下图所示：




看完上图相信你对大文件下载的方案，已经有了一定的了解。接下来，我们先来介绍 HTTP 范围请求。

## 一、HTTP 范围请求

HTTP 协议范围请求允许服务器只发送 HTTP 消息的一部分到客户端。范围请求在传送大的媒体文件，或者与文件下载的断点续传功能搭配使用时非常有用。如果在响应中存在 `Accept-Ranges` 首部（并且它的值不为 “none”），那么表示该服务器支持范围请求。

在一个 Range 首部中，可以一次性请求多个部分，服务器会以 multipart 文件的形式将其返回。如果服务器返回的是范围响应，需要使用 **206 Partial Content** 状态码。假如所请求的范围不合法，那么服务器会返回 **416 Range Not Satisfiable** 状态码，表示客户端错误。服务器允许忽略 `Range` 首部，从而返回整个文件，状态码用 200 。

### 1.1 Range 语法

```
Range: <unit>=<range-start>-
Range: <unit>=<range-start>-<range-end>
Range: <unit>=<range-start>-<range-end>, <range-start>-<range-end>
Range: <unit>=<range-start>-<range-end>, <range-start>-<range-end>, <range-start>-<range-end>
```

- `unit` ：范围请求所采用的单位，通常是字节（bytes）。
- `<range-start>` ：一个整数，表示在特定单位下，范围的起始值。
- `<range-end>` ：一个整数，表示在特定单位下，范围的结束值。这个值是可选的，如果不存在，表示此范围一直延伸到文档结束。

了解完 `Range` 语法之后，我们来看一下实际的使用示例：

#### 1.1.1 单一范围

```
$ curl http://i.imgur.com/z4d4kWk.jpg -i -H "Range: bytes=0-1023"
```

#### 1.1.2 多重范围

```
$ curl http://www.example.com -i -H "Range: bytes=0-50, 100-150"
```

好了，HTTP 范围请求的相关知识就先介绍到这里，下面我们步入正题开始介绍如何实现大文件下载。

## 二、如何实现大文件下载

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：



了解完大文件下载的流程之后，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 getContentLength 函数

顾名思义 `getContentLength` 函数，用于获取文件的长度。在该函数中，我们通过发送 `HEAD` 请求，然后从响应头中读取 `Content-Length` 的信息，进而获取当前 `url` 对应文件的内容长度。

```
function getContentLength(url) {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest();
    xhr.open("HEAD", url);
```

```
    xhr.send();
    xhr.onload = function () {
      resolve(
        ~~xhr.getResponseHeader("Content-Length")
      );
    };
    xhr.onerror = reject;
  });
}
```

### 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 getBinaryContent 函数

`getBinaryContent` 函数用于根据传入的参数发起范围请求，从而下载指定范围内的文件数据块：

```
function getBinaryContent(url, start, end, i) {
  return new Promise((resolve, reject) => {
    try {
      let xhr = new XMLHttpRequest();
      xhr.open("GET", url, true);
      xhr.setRequestHeader("range", `bytes=${start}-${end}`); // 请求头上设置范围请求信息
      xhr.responseType = "arraybuffer"; // 设置返回的类型为arraybuffer
      xhr.onload = function () {
        resolve({
          index: i, // 文件块的索引
          buffer: xhr.response, // 范围请求对应的数据
        });
      };
      xhr.send();
    } catch (err) {
      reject(new Error(err));
    }
  });
}
```

需要注意的是 **ArrayBuffer** 对象用来表示通用的、固定长度的原始二进制数据缓冲区。我们不能直接操作 **ArrayBuffer** 的内容，而是要通过类型数组对象或 **DataView** 对象来操作，它们会将缓冲区中的数据表示为特定的格式，并通过这些格式来读写缓冲区的内容。

### 2.1.4 定义 concatenate 函数

由于不能直接操作 **ArrayBuffer** 对象，所以我们需要先把 `ArrayBuffer` 对象转换为 `Uint8Array` 对象，然后在执行合并操作。以下定义的 `concatenate` 函数就是为了合并已下载的文件数据块，具体代码如下所示：

```
function concatenate(arrays) {
  if (!arrays.length) return null;
  let totalLength = arrays.reduce((acc, value) => acc + value.length, 0);
  let result = new Uint8Array(totalLength);
  let length = 0;
  for (let array of arrays) {
    result.set(array, length);
```

```
    length += array.length;
  }
  return result;
}
```

### 2.1.5 定义 saveAs 函数

`saveAs` 函数用于实现客户端文件保存的功能，这里只是一个简单的实现。在实际项目中，你可以考虑直接使用 FileSaver.js 。如果你对 FileSaver.js 的工作原理感兴趣的话，可以阅读 聊一聊 **15.5K** 的 **FileSaver**，是如何工作的？ 这篇文章。

```
function saveAs({ name, buffers, mime = "application/octet-stream" }) {
  const blob = new Blob([buffers], { type: mime });
  const blobUrl = URL.createObjectURL(blob);
  const a = document.createElement("a");
  a.download = name || Math.random();
  a.href = blobUrl;
  a.click();
  URL.revokeObjectURL(blob);
}
```

在 `saveAs` 函数中，我们使用了 Blob 和 Object URL。其中 Object URL 是一种伪协议，允许 Blob 和 File 对象用作图像，下载二进制数据链接等的 URL 源。在浏览器中，我们使用 `URL.createObjectURL` 方法来创建 Object URL，该方法接收一个 `Blob` 对象，并为其创建一个唯一的 URL，其形式为 `blob:<origin>/<uuid>` ，对应的示例如下：

```
blob:https://example.org/40a5fb5a-d56d-4a33-b4e2-0acf6a8e5f641
```

浏览器内部为每个通过 `URL.createObjectURL` 生成的 URL 存储了一个 URL → Blob 映射。因此，此类 URL 较短，但可以访问 `Blob` 。生成的 URL 仅在当前文档打开的状态下才有效。

好了，Object URL 的相关内容就先介绍到这里，如果你进一步了解 Blob 和 Object URL 的话，可以阅读 你不知道的 **Blob** 这篇文章。

### 2.1.6 定义 download 函数

`download` 函数用于实现下载操作，它支持 3 个参数：

- `url` （字符串类型）：预下载资源的地址；
- `chunkSize` （数字类型）：分块的大小，单位为字节；
- `poolLimit` （数字类型）：表示限制的并发数。

```
async function download({ url, chunkSize, poolLimit = 1 }) {
  const contentLength = await getContentLength(url);
  const chunks = typeof chunkSize === "number" ? Math.ceil(contentLength / chunkSize) : 1;
  const results = await asyncPool(
    poolLimit,
    [...new Array(chunks).keys()],
    (i) => {
      let start = i * chunkSize;
      let end = i + 1 == chunks ? contentLength - 1 : (i + 1) * chunkSize - 1;
      return getBinaryContent(url, start, end, i);
    }
  );
  const sortedBuffers = results
    .map((item) => new Uint8Array(item.buffer));
  return concatenate(sortedBuffers);
}
```

## 2.2 大文件下载使用示例

基于前面定义的辅助函数，我们就可以轻松地实现大文件并行下载，具体代码如下所示：

```
function multiThreadedDownload() {
  const url = document.querySelector("#fileUrl").value;
  if (!url || !/https?/.test(url)) return;
  console.log("多线程下载开始: " + +new Date());
  download({
    url,
    chunkSize: 0.1 * 1024 * 1024,
    poolLimit: 6,
  }).then((buffers) => {
    console.log("多线程下载结束: " + +new Date());
    saveAs({ buffers, name: "我的压缩包", mime: "application/zip" });
  });
}
```

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示例代码。

> 完整的示例代码：https://gist.github.com/semlinker/837211c039e6311e1e7629e5ee5f0a42

这里我们来看一下大文件下载示例的运行结果：



## 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，来实现大文件的并行下载。除了介绍 `asyncPool` 函数之外，阿宝哥还介绍了如何通过 HEAD 请求获取文件大小、如何发起 HTTP 范围请求及在客户端如何保存文件等相关知识。其实利用 `asyncPool` 函数不仅可以实现大文件的并行下载，而且还可以实现大文件的并行上传，感兴趣的小伙伴可以自行尝试一下。

## 四、参考资源

- 你不知道的 **Blob**
- MDN - ArrayBuffer
- MDN - HTTP请求范围
- **JavaScript** 中如何实现并发控制？

喜欢此内容的人还喜欢

“JS 葵花宝典”来了！与 JSON 之父一起参悟 JS 之道

全栈修仙之路

是黑人害了南非吗

求实处

聊一款售价亲民的小尺寸全能本

笔吧评测室