

文件上传情景

首先我们先来了解文件上传的整个流程机制

流程

你要做一个页面，用户点按钮 → 选中一个本地文件（比如 PDF、图片、Excel） → 上传到服务器 → 后端存储到磁盘 / 云存储 / 数据库。

1. 用户选择文件

在浏览器中，用户点击一个按钮或输入框，选择本地文件（比如图片、PDF、视频）。

前端使用HTML的 `<input type="file" id="fileInput" />` 元素来实现这个选择界面。它允许用户从文件管理器中挑选文件。

```
<input type="file" id="fileInput" />
```

选择文件后，浏览器会给你一个 **File 对象**（它继承自 **Blob**）。

```
const file = document.getElementById('fileInput').files[0];
console.log(file);
// File { name: "demo.png", size: 12345, type: "image/png", ... }
```

2. 构造请求数据

不能直接把 **File** 发给后端，要用 **FormData** 封装一下，模拟表单提交。

FormData 是专门给表单和文件准备的数据类型。

为什么需要 `FormData` ?

服务器通常需要的不只是文件本身，还可能需要一些伴随的文本信息，比如用户 ID、文件的描述等。

`FormData` 的作用就是把文件（`File` 对象）和这些额外的文本数据封装到一个统一的请求体中。

它会按照 `multipart/form-data` 的标准格式进行编码，以便服务器能够正确解析。

3. 发送请求

通过 `axios` 或 `fetch` 发请求，注意 `Content-Type` 是 `multipart/form-data`。

设置请求头的格式

```
axios.post('/api/upload', formData, {  
  headers: { 'Content-Type': 'multipart/form-data' }  
}).then(res => {  
  console.log('上传成功', res.data);  
}).catch(err => {  
  console.error('上传失败', err);  
});
```

4. 后端处理接收

后端框架（比如 `Node.js/Express`、`Java/Spring`、`Python/Flask`）会解析 `multipart/form-data` 请求，把文件内容和参数拆开。

它通常会：

- 把文件存到磁盘 / 云存储（阿里 OSS、七牛云、AWS S3）
- 把文件路径 / 信息存到数据库

5. 前端处理结果

- 如果上传成功，前端可能需要：
 - 显示“上传成功”的提示
 - 展示上传的文件（比如预览图片）
 - 把后端返回的 URL 存起来，之后可以下载

数据类型

◆ 上传过程涉及的数据类型

类型	用途	示例	📄
File	表示用户选择的文件，继承自 Blob	<code>{ name: "demo.png", size: 12345, type: "image/png" }</code>	
Blob	二进制数据对象 (File 的父类)	<code>new Blob(["Hello"], { type: "text/plain" })</code>	
FormData	发送表单数据和文件的容器	<code>formData.append("file", file)</code>	

简单理解：

- **File** → 用户本地文件（选中的文件）
- **FormData** → 把文件和其它字段打包，发给后端
- **Blob** → 底层的二进制容器（你用得少，但它是 **File** 的基石）

1.Blob对象

Blob 全称是 *Binary Large Object*（二进制大对象）

它表示不可变的原始二进制数据，比如文件内容、图片、音视频等。

在 JavaScript 中，**Blob** 是一个构造函数，能够让你把原始的二进制数据（数组、字符串、文件等）封装成一个 **Blob** 对象。

简单理解：

👉 **Blob** 就像一个文件的数据容器，只存数据本身，不关心这些数据是什么类型。

基本使用

1.创建Blob对象

`new Blob()`手动创建一个**Blob**对象，内容可以是字符串、数组、甚至其他**Blob**。

```
new Blob(array, options);
```

- `array`：一个数组，包含要存入Blob的数据（可以是字符串、ArrayBuffer、其他Blob等）。
- `options`：对象，包含 `type`（MIME类型）和可选的 `endings`（行结束符，少用）。

创建文本类型的Blob

```
const text = "Hello, this is a text file!";  
const blob = new Blob([text], { type: 'text/plain' });  
console.log(blob.size); // 输出: 25 (字节数)  
console.log(blob.type); // 输出: text/plain
```

创建JSON类型的Blob

```
const data = { name: "Alice", age: 25 };  
const jsonString = JSON.stringify(data);  
const blob = new Blob([jsonString], { type: 'application/json' });
```

File文件类型继承Blob

当用户通过选择文件时，`files`属性返回的File对象就是Blob。

```
<input type="file" id="fileInput">  
<script>  
  const input = document.getElementById('fileInput');  
  input.addEventListener('change', () => {  
    const file = input.files[0]; // File对象, 继承自Blob  
    console.log('文件名:', file.name); // 如: image.jpg  
    console.log('大小:', file.size); // 字节数  
    console.log('类型:', file.type); // 如: image/jpeg  
  });  
</script>
```

2.Blob常用的API函数

1.blob.slice()

`blob.slice(start, end, contentType?)`

作用：把一个 Blob 切成小块，**返回新的 Blob**。

参数：

- `start`：起始字节索引
- `end`：结束字节索引（不包含）
- `contentType`：可选，新的 MIME 类型

场景：大文件分片上传、断点续传时使用。

```
const blob = new Blob(["Hello World"], { type: "text/plain" });  
// 取前 5 个字节  
const chunk = blob.slice(0, 5, "text/plain");  
  
chunk.text().then(console.log); // Hello
```

2.blob.stream()

作用：把 Blob 转成 `ReadableStream`（可读流）。

场景：大文件上传、流式处理二进制数据。

```
const blob = new Blob(["Hello World"]);  
const stream = blob.stream();  
  
const reader = stream.getReader();  
reader.read().then(({ value, done }) => {  
  console.log(value); // Uint8Array(5) [72, 101, 108, 108, 111]  
});
```

3.blob.text()

作用：异步把 Blob 转成 **字符串**。

返回：`Promise<string>`

场景：读取文本文件（txt、json、csv）

```
const blob = new Blob(["Hello World"], { type: "text/plain" });

blob.text().then(text => {
  console.log(text); // Hello World
});
```

4.blob.arrayBuffer()

作用：异步把 Blob 转成 ArrayBuffer。

返回：Promise<ArrayBuffer>

场景：需要用到二进制处理（比如音频、图片、PDF 的底层处理）。

```
const blob = new Blob(["Hello World"]);
blob.arrayBuffer().then(buffer => {
  console.log(new Uint8Array(buffer));
  // Uint8Array(11) [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100]
});
```

什么是ArrayBuffer?

ArrayBuffer是JavaScript中的一种内置对象，表示一块固定长度的二进制数据缓冲区。

在解释 `blob.arrayBuffer()` 的作用之前，我们需要先了解一下 `ArrayBuffer`。

你可以把 `ArrayBuffer` 想象成一个 通用的、固定长度的、原始二进制数据缓冲区。它本身并不能直接操作，你需要用一个“视图”（如 `Int8Array`，`Uint8Array`，`DataView` 等）来访问和操作它里面的数据。

简单来说，`ArrayBuffer` 提供了数据本身，而视图提供了操作这些数据的方式。这就像一个图书馆的仓库（`ArrayBuffer`）和一本本具体的书（视图）。

以下是 `ArrayBuffer` 常见的应用场景，以及它解决的具体问题：

场景	问题	<code>ArrayBuffer</code> 如何解决	
文件解析	需要读取文件头或特定部分（如PNG图片的元数据）。	用 <code>ArrayBuffer</code> 读取文件内容，通过 <code>DataView</code> 解析特定字节。	
音视频处理	处理音频样本或视频帧数据（如Web Audio API）。	<code>ArrayBuffer</code> 存储原始音频/视频数据， <code>TypedArray</code> 操作样本值。	
网络传输	WebSocket或WebRTC需要发送/接收二进制数据。	<code>ArrayBuffer</code> 作为数据载体，高效传输字节流。	
WebGL渲染	向GPU发送顶点数据或纹理数据。	<code>ArrayBuffer</code> 存储顶点坐标或像素数据，供WebGL使用。	
大文件分片	大文件上传需要切片，逐块处理。	用 <code>ArrayBuffer</code> 读取文件片段，精细控制每块内容。	
协议解析	解析自定义二进制协议（如游戏数据包）。	<code>ArrayBuffer</code> 提供字节级访问，结合 <code>DataView</code> 解析协议字段。	

为什么不用字符串或Blob？

- 字符串：字符串适合文本，但二进制数据（如图片像素）转成字符串会变大（Base64编码增加33%体积），且解析慢。
- Blob：Blob适合整体操作（如上传整个文件），但不适合逐字节修改或解析，因为它是“黑盒”。

总结：ArrayBuffer解决了需要低级、高效、逐字节操作二进制数据的问题，特别在性能敏感或需要解析复杂格式的场景。

blob.arrayBuffer()的作用

定义：

- `blob.arrayBuffer()` 是 `Blob` 对象的一个方法，返回一个 `Promise`，解析后得到一个 `ArrayBuffer`，表示 `Blob` 的全部二进制内容。
- 语法：`blob.arrayBuffer().then(buffer => { /* 处理buffer */ });`

例如

- 使用 `new Uint8Array(buffer)` 创建一个视图来访问每个字节

```
blob.arrayBuffer().then(buffer => {
  const byteArray = new Uint8Array(buffer);
  // 逐字节处理
  for (let i = 0; i < byteArray.length; i++) {
    const byte = byteArray[i];
    console.log(`字节 ${i}: ${byte.toString(16)}`);
    // 这里可以对每个字节进行处理
  }
});
```

作用：

- 将 **Blob**（或 **File**，因为 **File** 继承自 **Blob**）的内容转成 **ArrayBuffer**，以便进行低级字节操作。
- 适合需要深入解析或修改 **Blob** 内容的场景，而不是直接上传或预览。

使用场景：

- 解析文件内容：比如读取PNG图片的元数据（宽高、分辨率）。
- 处理音视频数据：提取音频样本或视频帧。
- 自定义协议：解析服务器发来的二进制数据包。
- 分片处理：读取文件的一部分，进行加工后再上传。

示例：读取图片文件的ArrayBuffer

```
javascript 收起 自动换行 运行 复制

const input = document.querySelector('input[type="file"]');
input.addEventListener('change', async () => {
  const file = input.files[0]; // File是Blob
  const buffer = await file.arrayBuffer(); // 转成ArrayBuffer
  const uint8Array = new Uint8Array(buffer); // 用TypedArray查看字节
  console.log('前8个字节:', uint8Array.slice(0, 8)); // 比如PNG文件头
});
```


操作ArrayBuffer

要操作ArrayBuffer，通常需要配合视图（TypedArray或DataView）来读写数据，因为ArrayBuffer本身只存字节，不提供直接访问方法。

TypedArray：如Uint8Array（8位无符号整数）、Int16Array（16位有符号整数），适合结构化数据。

DataView：更灵活，允许读取任意字节位置的任意类型数据（8位、16位、32位等）。

步骤：

1. 获取或创建 **ArrayBuffer**（如通过 `blob.arrayBuffer()`）。
2. 用 **TypedArray** 或 **DataView** 创建视图。
3. 读取或修改字节。

示例1：创建和操作ArrayBuffer

javascript

✕ 收起 自动换行 ▶ 运行 复制

```
// 创建一个10字节的ArrayBuffer
const buffer = new ArrayBuffer(10);
const view = new Uint8Array(buffer); // 8位无符号整数视图
// 写入数据
view[0] = 255; // 第一个字节设为255
view[1] = 128; // 第二个字节设为128
console.log(view); // 输出: Uint8Array [255, 128, 0, 0, ...]
```

示例2：解析PNG文件头

PNG文件的前8个字节是固定签名（89 50 4E 47 0D 0A 1A 0A）。可以用 **ArrayBuffer** 验证：

javascript

✕ 收起 自动换行 ▶ 运行 复制

```
const input = document.querySelector('input[type="file"]');
input.addEventListener('change', async () => {
  const file = input.files[0];
  const buffer = await file.arrayBuffer();
  const view = new Uint8Array(buffer);
  const isPNG = view[0] === 0x89 && view[1] === 0x50 && view[2] === 0x4E && view[3] === 0x47 && view[4] === 0x0D && view[5] === 0x0A && view[6] === 0x1A && view[7] === 0x0A;
  console.log('是PNG文件?', isPNG);
});
```

示例3: Web Audio处理

读取音频文件的 `ArrayBuffer`，交给Web Audio API处理：

javascript

✕ 收起

⇌ 自动换行

▶ 运行

📄 复制

```
const input = document.querySelector('input[type="file"]');
input.addEventListener('change', async () => {
  const file = input.files[0];
  const buffer = await file.arrayBuffer();
  const audioContext = new AudioContext();
  audioContext.decodeAudioData(buffer, (audioBuffer) => {
    console.log('音频时长:', audioBuffer.duration); // 解析音频
  });
});
```

视图

- `TypedArray` (如 `Uint8Array`、`Int32Array`)：
 - `length`：元素个数 (不是字节数，取决于类型)。
 - `set(array, offset)`：写入数据。
 - `subarray(start, end)`：返回子数组视图。
- 示例：

javascript

✕

⇌

▶

📄

```
const buffer = new ArrayBuffer(8);
const uint8 = new Uint8Array(buffer);
uint8.set([1, 2, 3], 0); // 写入1,2,3
console.log(uint8.subarray(0, 2)); // Uint8Array [1, 2]
```

- **DataView** :
 - `getUint8(offset)` : 读取指定字节位置的8位无符号整数。
 - `setUint8(offset, value)` : 写入8位整数。
 - `getInt16(offset, littleEndian?)` : 读取16位整数（可指定字节序）。
- 示例:

javascript

```
const buffer = new ArrayBuffer(4);
const view = new DataView(buffer);
view.setUint8(0, 255); // 第一个字节设为255
console.log(view.getUint8(0)); // 255
```

注意

- **性能**: `ArrayBuffer` 适合低级操作，但操作大文件时注意内存使用，尽量用分片 (`blob.slice`)。
- **视图选择**: 用 `TypedArray` (如 `Uint8Array`) 处理均匀数据，用 `DataView` 处理复杂结构。
- **异步操作**: `blob.arrayBuffer()` 是异步的，记得用 `await` 或 `.then`。
- **兼容性**: `ArrayBuffer` 和 `blob.arrayBuffer()` 在现代浏览器广泛支持，IE10+可用。
- **调试**: 用 `console.log(new Uint8Array(buffer))` 查看字节内容，便于调试。

5.URL.createObjectURL(blob)

作用: 生成一个临时的本地 URL (指向 Blob 的数据)。

场景: 文件预览、下载文件。

```
const blob = new Blob(["Hello World"], { type: "text/plain" });
const url = URL.createObjectURL(blob);

const a = document.createElement("a");
a.href = url;
a.download = "hello.txt";
a.click();

URL.revokeObjectURL(url); // 释放内存
```

为什么需要 `URL.createObjectURL(blob)`?

你可以把 `Blob` 对象想象成一个“只存在于浏览器内存中的数据包”。这个数据包没有网络地址，浏览器无法直接通过 `` 或 `` 这样的常规方式来访问它。

`URL.createObjectURL(blob)` 的作用就是给这个“内存中的数据包”创建一个临时的、可以在浏览器内部使用的“身份证”或“地址”。这个地址的格式通常是 `blob:http://...`

它的核心目的就是：让那些需要通过 `URL` 才能访问资源的 `API` 或 `HTML` 标签，能够临时访问到你本地的 `Blob` 数据。

没有它，会怎么样？

想象一下，如果你想在前端直接预览一张用户选择的图片，你会怎么做？

如果你没有 `URL.createObjectURL`，你可能会想：

HTML

```

```

这个 `src` 属性需要一个 `URL`。你不能直接把 `Blob` 对象扔进去，因为浏览器不认识它。你也不能直接从用户的电脑中获取文件的本地路径（出于安全原因，浏览器禁止这么做）。

有了它，一切都简单了

有了 `URL.createObjectURL`，整个流程就通畅了：

1. 用户选择了一张图片，你得到一个 `File` 对象（`File` 是 `Blob` 的子类）。
2. 你调用 `URL.createObjectURL(file)`，浏览器为你生成一个临时的 `blob:URL`。
3. 你把这个 `blob:URL` 设置给 ``。
4. 浏览器看到这个 `src` 地址，知道它指向的是你内存中的 `Blob` 数据，于是它能正确地加载和显示这张图片。

这个过程不需要向服务器发送任何请求！一切都在前端浏览器中完成，速度非常快，极大地提升了用户体验。

另一个例子：文件下载

同样，在文件下载的场景中：

1. 你用 `Axios` 下载文件，后端返回一个 `Blob` 对象。
2. 你不能直接把这个 `Blob` 对象放在 `` 标签中。
3. 你需要用 `URL.createObjectURL(blob)` 生成一个临时的 `blob:URL`。
4. 把这个 URL 赋值给 `<a>` 的 `href` 属性，并设置 `download` 属性。
5. 模拟点击这个 `<a>` 标签，浏览器看到 `href` 是一个 `blob:URL`，并且 `download` 属性被设置了，就会触发下载。

总而言之，`URL.createObjectURL(blob)` 的作用就是 将本地的二进制数据 `Blob` 对象，转换为一个临时的、可供浏览器直接访问的 URL，从而打通了浏览器内部数据和需要 URL 的外部接口之间的桥梁。

6.URL.revokeObjectURL(url)

作用：释放 `createObjectURL` 生成的内存。

场景：避免内存泄漏。

```
URL.revokeObjectURL(url);
```

7.FileReader 相关方法

FileReader 是浏览器提供的 API，用来读取 Blob/File 数据。和 `blob.text()` / `blob.arrayBuffer()` 类似，但更兼容旧浏览器。

(1) readAsDataURL(blob)

- 作用：读取成 Base64 字符串。
- 场景：图片上传前预览。

```
const blob = new Blob(["Hello World"], { type: "text/plain" });
const reader = new FileReader();

reader.onload = () => {
  console.log(reader.result); // data:text/plain;base64,SGVsbG8gV29ybGQ=
};
reader.readAsDataURL(blob);
```

(2) readAsText(blob)

- 作用：读取成 文本。
- 场景：读取 .txt、.json 文件。

```
const blob = new Blob(["Hello World"], { type: "text/plain" });
const reader = new FileReader();

reader.onload = () => {
  console.log(reader.result); // Hello World
};
reader.readAsText(blob);
```

(3) readAsArrayBuffer(blob)

- 作用：读取成 ArrayBuffer。
- 场景：二进制文件（音频、视频、PDF）。

```
const blob = new Blob(["Hello World"]);
const reader = new FileReader();

reader.onload = () => {
  console.log(new Uint8Array(reader.result));
  // Uint8Array(11) [72, 101, 108, ...]
};
reader.readAsArrayBuffer(blob);
```

本身FileReader会有一堆的监听事件

```
const reader = new FileReader();
const file = document.querySelector("input[type=file]").files[0];

reader.onloadstart = () => console.log("开始读取...");
reader.onprogress = (e) => {
  if (e.lengthComputable) {
    console.log(`进度: ${e.loaded / e.total * 100}.toFixed(2)}%`);
  }
};
reader.onload = () => console.log("读取成功:", reader.result);
reader.onerror = () => console.error("读取失败:", reader.error);
reader.onabort = () => console.warn("读取被取消");
reader.onloadend = () => console.log("读取结束");

// 开始读取为文本
reader.readAsText(file);
```

onloadstart: 当读取操作开始时触发。

onprogress: 读取数据过程中持续触发, 可用于显示进度条。

onload: 当读取成功完成时触发。

onabort: 当读取操作被中断 (调用 `reader.abort()`) 时触发。

onerror: 当读取失败时触发 (例如文件损坏或权限不足)。

onloadend: 当读取操作结束时触发 (无论成功或失败, 都会触发)。

2.File对象

File 对象其实就是 Blob 的子类。它继承了 Blob 的所有特性（比如 `slice()`、`stream()`、`text()`、`arrayBuffer()` 等），只是在此基础上 **额外包含了文件相关的属性**。

◆ File 对象简介

- **定义：** `File` 表示来自用户系统的一个文件对象，比如你用 `<input type="file">` 选择的文件，或者拖拽文件到浏览器得到的对象。
- **本质：** `File` 继承自 `Blob`，只是额外提供了文件的 **元数据 (metadata)**。

主要属性

◆ File 的主要属性

1. `name`
 - **文件名 (不包含路径)**
 - 例如: `"photo.png"`
2. `lastModified`
 - **最后修改时间 (Unix 时间戳, 毫秒)**
 - 可以转成 `new Date(file.lastModified)`
3. `size`
 - **文件大小, 字节数 (继承自 Blob)**
4. `type`
 - **MIME 类型 (继承自 Blob)**
 - 例如: `"image/png"`、`"text/plain"`

创建获取方式

1. 通过 `<input type="file">`

html

```
<input type="file" id="upload">
<script>
  const input = document.getElementById("upload");
  input.addEventListener("change", () => {
    const file = input.files[0];
    console.log(file.name, file.size, file.type);
  });
</script>
```

2. 拖拽上传 (Drag & Drop)

js

```
dropArea.addEventListener("drop", e => {
  e.preventDefault();
  const file = e.dataTransfer.files[0];
  console.log(file.name, file.size);
});
```

3. 通过构造函数 `new File()` 创建

js

第一个参数传递blob数据

复制代码

```
const content = new Blob(["Hello World"], { type: "text/plain" });
const file = new File([content], "hello.txt", { type: "text/plain", lastModified: Date.now() });
console.log(file.name); // hello.txt
```

基础的文件上传实现

```

<template>
  <div class="p-4">
    <h2>文件上传示例</h2>

    <!-- 选择文件 -->
    <input type="file" @change="handleFileChange" />

    <!-- 上传按钮 -->
    <button @click="uploadFile" :disabled="!selectedFile">
      上传文件
    </button>

    <!-- 上传进度条 -->
    <div v-if="progress > 0">
      <p>上传进度: {{ progress }}%</p>
      <progress :value="progress" max="100"></progress>
    </div>

    <!-- 上传结果 -->
    <div v-if="uploadResult">
      <p>{{ uploadResult }}</p>
    </div>
  </div>
</template>

```

```

<script setup>
import { ref } from "vue";
import axios from "axios";    "axios": Unknown word.

```

// 状态

```

const selectedFile = ref(null);
const progress = ref(0);
const uploadResult = ref("");

```

响应式数据

// 选择文件

```

const handleFileChange = (event) => {
  selectedFile.value = event.target.files[0];
  progress.value = 0;
  uploadResult.value = "";
};

```

拿到选中文件数据

// 上传文件

```

const uploadFile = async () => {
  if (!selectedFile.value) return;

```

try {

构造FormData数据 传递文件

// 构造 FormData

```

const formData = new FormData();
formData.append("file", selectedFile.value); // 后端接收的字段名是 "file"

```

// 发送请求

```

const response = await axios.post("http://localhost:3000/upload", formData, { headers: {

```

```
const res = await axios.post("http://localhost:5000/upload", formData, {
  headers: {
    "Content-Type": "multipart/form-data",
  },
  onUploadProgress: (progressEvent) => {
    if (progressEvent.total) {
      progress.value = Math.round(
        (progressEvent.loaded * 100) / progressEvent.total
      );
    }
  },
});
```

大文件的分片上传

整体流程（从前端视角）：

1. 用户选择大文件。
2. 前端将文件（Blob/File）切分成小块（chunks）。
3. 逐块发送到服务器（每个块带索引，如第 1 块、第 2 块）。
4. 服务器接收每个块，临时存储。
5. 所有块上传完，前端通知服务器合并成完整文件。
6. 服务器返回成功响应（如文件 URL）。

数据类型回顾（基于你之前的学习）：

- **File**：用户选择的文件对象。
- **Blob**：用 `file.slice()` 切片得到的小块。
- **FormData**：打包每个块和元信息（如块索引、总块数）。
- **ArrayBuffer** 或 **Blob**：可选，用于读取块内容（但上传时直接用 Blob 即可）。

基础流程

1.选择文件

```
<input type="file" id="fileInput" />
<button id="uploadBtn">上传</button>
```

```
const fileInput = document.getElementById('fileInput')
const uploadBtn = document.getElementById('uploadBtn')

uploadBtn.addEventListener('click', () => {
  const file = fileInput.files[0]
  if (!file) return alert('请选择文件')
  uploadFile(file)
})
```

2.切片

```
const chunkSize = 5 * 1024 * 1024; // 5MB
function createChunks(file) {
  const chunks = [];
  let start = 0;
  while (start < file.size) {
    const end = Math.min(start + chunkSize, file.size);
    chunks.push(file.slice(start, end));
    start = end;
  }
  return chunks;
}
```

解释:

- `file.slice(start, end)` 返回文件的一部分。
- `chunkSize` 可以根据实际情况调整。

3.生成唯一Hash

这里生成文件 Hash 用于：

- 唯一标识文件。
- 支持断点续传（服务器知道哪些 chunk 已上传）。

```
async function getFileHash(file) {  
  const chunkSize = 10 * 1024 * 1024; // 每10MB一块用于哈希  
  const spark = new SparkMD5.ArrayBuffer(); // 使用SparkMD5库  
  for (let start = 0; start < file.size; start += chunkSize) {  
    const chunk = file.slice(start, start + chunkSize);  
    const buffer = await chunk.arrayBuffer();  
    spark.append(buffer);  
  }  
  return spark.end(); // 返回文件唯一 hash  
}
```

提示： SparkMD5 是一个轻量的 JS 库，可以快速生成文件 hash，生产环境常用。

4.分片放入 FormData

每个分片上传时，需要发送：

- 分片本身 (chunk)
- 分片索引 (index)
- 文件 Hash (fileHash)

```

async function uploadChunk(chunk, index, fileHash) {
  const formData = new FormData();
  formData.append('chunk', chunk);
  formData.append('index', index);
  formData.append('fileHash', fileHash);

  const res = await fetch('/upload_chunk', {
    method: 'POST',
    body: formData
  });
  return res.json();
}

```

5.API调用

```

async function uploadFile(file) {
  const chunks = createChunks(file);
  const fileHash = await getFileHash(file);

  for (let i = 0; i < chunks.length; i++) {
    let success = false;
    while (!success) {
      try {
        await uploadChunk(chunks[i], i, fileHash);
        success = true;
        console.log(`第 ${i + 1} 块上传成功`);
      } catch (err) {
        console.log(`第 ${i + 1} 块上传失败，重试中...`);
      }
    }
  }

  // 通知服务器合并
  await fetch('/merge_chunks', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ fileHash, totalChunks: chunks.length, fileName: file.name })
  });
  console.log('上传完成');
}

```

6.优化

分片读取和 Hash 计算本质上都是在处理同一块文件数据，所以完全可以合并成一个流程，这样可以提高性能，减少重复读取文件。

```
async function uploadFile(file) {
  const chunkSize = 5 * 1024 * 1024; // 5MB
  const spark = new SparkMD5.ArrayBuffer(); // Hash 计算器

  const totalChunks = Math.ceil(file.size / chunkSize); // 计算分片总

  for (let i = 0; i < totalChunks; i++) {
    const start = i * chunkSize;
    const end = Math.min(start + chunkSize, file.size);
    const chunk = file.slice(start, end);

    // 计算 Hash // 计算当次分片的大小
    const buffer = await chunk.arrayBuffer();
    spark.append(buffer);

    // 上传分片 // 追加到FormData数据
    const formData = new FormData();
    formData.append('chunk', new Blob([buffer]));
    formData.append('index', i);
    // 这里先不加 fileHash，因为最终 hash 要等所有分片计算完
    await fetch('/upload_chunk', { method: 'POST', body: formData });
  }

  // 分片全部读取完成，计算最终 Hash // 计算最终Hash
  const fileHash = spark.end();

  // 通知服务器合并
  await fetch('/merge_chunks', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ fileHash, totalChunks, fileName: file.name })
  });

  console.log('上传完成');
}
```


对的，你理解得很准确，如果要实现断点续传/重传，就不能在完全没算出文件 Hash 的情况下就直接上传

补充

问题 2: 为什么 SparkMD5 需要 ArrayBuffer? 为什么要把 Blob 转成 ArrayBuffer?

这是因为 SparkMD5 底层是基于字节 (byte) 计算的。

- MD5 算法本质上是对一段 二进制数据 (字节流) 做运算;
- Blob (包括 File) 只是一个“文件片段”的抽象，浏览器里不能直接逐字节读取它;
- 要想逐字节读取 Blob，必须用 FileReader 把它转换成 ArrayBuffer (字节数组)，这样 SparkMD5 才能一边读取字节、一边更新哈希计算。


```

<template>
  <div class="p-4">
    <h2>文件上传示例</h2>

    <!-- 选择文件 -->
    <input type="file" @change="handleFileChange" />

    <!-- 上传按钮 -->
    <button @click="uploadFile" :disabled="!selectedFile">
      上传文件
    </button>

    <!-- 上传进度条 -->
    <div v-if="progress > 0">
      <p>上传进度: {{ progress }}%</p>
      <progress :value="progress" max="100"></progress>
    </div>

    <!-- 上传结果 -->
    <div v-if="uploadResult">
      <p>{{ uploadResult }}</p>
    </div>
  </div>
</template>

<script setup>
import { ref } from "vue";
import sparkMD5 from "spark-md5";
// 状态
const selectedFile = ref(null);
const progress = ref(0);
const uploadResult = ref("");

//分片大小
const chunkSize = 1024 * 1024 * 2;

// 选择文件
const handleFileChange = (event) => {
  selectedFile.value = event.target.files[0];
  progress.value = 0;
  uploadResult.value = "";
};

//切片与生成Hash
const chunksAndHash= async (file)=>{

  const spark=new sparkMD5.ArrayBuffer();

  const totalChunks=Math.ceil(file.size/chunkSize); //计算分片 向上取整
  const FormDataList=[];

  for(let i=0;i<totalChunks;i++){

```

```

const start=i*chunkSize;
const end=Math.min(file.size,start+chunkSize); //需要做一个去处理，防止最后一个分片越界
const chunk=file.slice(start,end);

const formData=new FormData();

const buffer=await chunk.arrayBuffer();
spark.append(buffer); //计算Hash值

formData.append("chunk",chunk);
formData.append("chunkIndex",i);
formData.append("totalChunks",totalChunks);
FormDataList.push(formData);
}

const fileHash=spark.end()

//循环给每一个分片都加上唯一标识符hash
FormDataList.forEach((item)=>{
  item.append("fileHash",fileHash);
})
return FormDataList;
}

const uploadFile=async ()=>{
  //如果没有选择就返回
  if (!selectedFile.value) return;
  const FormDataList=await chunksAndHash(selectedFile.value);

  let uploadedSize = 0; // 已上传总字节

  //for循环进行分片上传
  for(let i=0;i<FormDataList.length;i++){
    const item=FormDataList[i];

    await API.post("http://localhost:3000/upload",item,{
      headers: {
        "Content-Type": "multipart/form-data",
      },
      //注册上传进度事件
      onUploadProgress: (progressEvent) => {
        //已经上传的大小 progressEvent.loaded 只表示当前分片已上传的字节
        const loaded=progressEvent.loaded;

        //总大小
        const chunkAllSize=selectedFile.value.size;
        //计算已经上传的进度
        progress.value=Math.round(((loaded+uploadedSize)/chunkAllSize)*100);
      },
    })

    //更新已上传总字节
    uploadedSize += item.get("chunk").size; // 用实际分片大小累加
  }
}

```

```

    }
}

</script>

```

这个版本你可以看到，分片和计算Hash是在一个循环里面完成的

然后把每个分片创建一个FormData存入

然后把每一个FormData塞入到List里面

然后等待Hash计算完成之后，List.for循环，给每一个FormData都塞入Hash唯一文件标识

最终你可以看到，我这里是使用For循环遍历List,给一个个分片发送数据

这样有一点不好就是，线性循环发请求很慢，其实可以设置一个并发池，设置并发请求

并发控制上传

我们进行改造，不再使用for循环单线程上传，uploadFile只负责上传分片

```

//单片上传的函数
const uploadFile=async (item,totalSize,uploadedSize)=>{

  await API.post("http://localhost:3000/upload",item,{
    headers: {
      "Content-Type": "multipart/form-data",
    },
    //注册上传进度事件
    onUploadProgress: (progressEvent) => {
      //已经上传的大小 progressEvent.loaded 只表示当前分片已上传的字节
      const loaded=progressEvent.loaded

      //总大小
      const chunkAllSize=totalSize
      //计算已经上传的进度
      progress.value=Math.round(((loaded+uploadedSize.value)/chunkAllSize)*100);
    },
  })

  //更新已上传总字节
  uploadedSize.value += item.get("chunk").size; // 用实际分片大小累加
}

```

只负责上传分片

写一个并发控制函数

```

//并发控制
const chunksConcurrency=async (FormDataList)=>{

  //直接从选中的文件，得到整个文件的大小
  const totalSize=selectedFile.value.size;

  //ref响应式数据动态设置已经上传的文件大小
  const uploadedSize=ref(0);

  //设置一个索引，遍历FormdataList
  let index=0;
  //并发控制池      Array(5)生成一个长度为并发数量的数组
  //                fill填充null 防止map函数跳过稀疏数组
  const pool=Array(concurrencyPool).fill(null).map(async()=>{
    //遍历FormdataList
    while(index<FormDataList.length){
      const item=FormDataList[index];
      index++;
      await uploadFile(item,totalSize,uploadedSize);
    }
  })

  await Promise.all(pool).then(()=>{
    uploadResult.value="上传成功";
    progress.value=100;
    //清空选中的文件
    selectedFile.value=null;
  })
}

```

这里写一个并发池



首选所有的并发槽，都共享同一个index，他负责记录遍历List请求任务。

直接Array(concurrency).fill(null)生成一个数组，数据里面使用map塞入并发函数

```

//设置一个索引，遍历FormDataList
let index=0;
//并发控制池    Array(5)生成一个长度为并发数量的数组
//            fill填充null 防止map函数跳过稀疏数组
const pool=Array(concurrencyPool).fill(null).map(async()=>{
    //遍历FormDataList
    while(index<FormDataList.length){
        const item=FormDataList[index];
        index++;
        await uploadFile(item,totalSize,uploadedSize);
    }
})

```

这里设置成争抢任务，首先一开始生成concurrency数量的并发任务

然后每个并发槽等待自己的任务完成，然后去领取下一个任务

直到index=List.length任务队列为空

然后使用Promise.all去等待并发池里面全部任务结束

```

//并发控制
const chunksConcurrency=async (FormDataList)=>{

    //直接从选中的文件，得到整个文件的大小
    const totalSize=selectedFile.value.size;

    //ref响应式数据动态设置已经上传的文件大小
    const uploadedSize=ref(0);

    //设置一个索引，遍历FormdataList
    let index=0;
    //并发控制池      Array(5)生成一个长度为并发数量的数组
    //                fill填充null 防止map函数跳过稀疏数组
    const pool=Array(concurrencyPool).fill(null).map(async()=>{
        //遍历FormdataList
        while(index<FormDataList.length){
            const item=FormDataList[index];
            index++;
            await uploadFile(item,totalSize,uploadedSize);
        }
    })

    await Promise.all(pool).then(()=>{
        uploadResult.value="上传成功";
        progress.value=100;
        //清空选中的文件
        selectedFile.value=null;
    })
}

```

准确来说，Promise.all 并不直接“监听”池子里的任务，而是等待一组 Promise（你的 pool 数组中的 Promise）全部解析（resolve）或任意一个拒绝（reject）。

它接收一个 Promise 数组，返回一个新的 Promise，当所有输入的 Promise 都完成后，返回它们的解析结果。

并发池，是负责不断地把全部请求promise塞入Promise.all


```

<template>
  <div class="p-4">
    <h2>文件上传示例</h2>

    <!-- 选择文件 -->
    <input type="file" @change="handleFileChange" />

    <!-- 上传按钮 -->
    <button @click="startUpload" :disabled="!selectedFile">
      上传文件
    </button>

    <!-- 上传进度条 -->
    <div v-if="progress > 0">
      <p>上传进度: {{ progress }}%</p>
      <progress :value="progress" max="100"></progress>
    </div>

    <!-- 上传结果 -->
    <div v-if="uploadResult">
      <p>{{ uploadResult }}</p>
    </div>
  </div>
</template>

```

```

<script setup>
import { ref } from "vue";
import sparkMD5 from "spark-md5";
// 状态
const selectedFile = ref(null);
const progress = ref(0);
const uploadResult = ref("");

// 分片大小
const chunkSize = 1024 * 1024 * 2;

// 并发控制池数量
const concurrencyPool=5;

// 选择文件
const handleFileChange = (event) => {
  selectedFile.value = event.target.files[0];
  progress.value = 0;
  uploadResult.value = "";
};

// 切片与生成Hash
const chunksAndHash= async (file)=>{

```

```

const spark=new sparkMD5.ArrayBuffer();

const totalChunks=Math.ceil(file.size/chunkSize); //计算分片 向上取整
const FormDataList=[];

for(let i=0;i<totalChunks;i++){
  const start=i*chunkSize;
  const end=Math.min(file.size,start+chunkSize); //需要做一个去处理，防止最后一个分片越界
  const chunk=file.slice(start,end);

  const formData=new FormData();

  const buffer=await chunk.arrayBuffer();
  spark.append(buffer); //计算Hash值

  formData.append("chunk",chunk);
  formData.append("chunkIndex",i);
  formData.append("totalChunks",totalChunks);
  FormDataList.push(formData);
}

const fileHash=spark.end()

//循环给每一个分片都加上唯一标识符hash
FormDataList.forEach((item)=>{
  item.append("fileHash",fileHash);
})
return FormDataList;
}

//单片上传的函数
const uploadFile=async (item,totalSize,uploadedSize)=>{

  await API.post("http://localhost:3000/upload",item,{
    headers: {
      "Content-Type": "multipart/form-data",
    },
    //注册上传进度事件
    onUploadProgress: (progressEvent) => {
      //已经上传的大小 progressEvent.loaded 只表示当前分片已上传的字节
      const loaded=progressEvent.loaded

      //总大小
      const chunkAllSize=totalSize
      //计算已经上传的进度
      progress.value=Math.round(((loaded+uploadedSize.value)/chunkAllSize)*100);
    },
  })
}

```

```

    },
  })
  //更新已上传总字节
  uploadedSize.value += item.get("chunk").size; // 用实际分片大小累加
}

//并发控制
const chunksConcurrency=async (FormDataList)=>{

  //直接从选中的文件，得到整个文件的大小
  const totalSize=selectedFile.value.size;

  //ref响应式数据动态设置已经上传的文件大小
  const uploadedSize=ref(0);

  //设置一个索引，遍历FormdataList
  let index=0;
  //并发控制池    Array(5)生成一个长度为并发数量的数组
  //              fill填充null 防止map函数跳过稀疏数组
  const pool=Array(concurrencyPool).fill(null).map(async()=>{
    //遍历FormdataList
    while(index<FormDataList.length){
      const item=FormDataList[index];
      index++;
      await uploadFile(item,totalSize,uploadedSize);
    }
  })

  await Promise.all(pool).then(()=>{
    uploadResult.value="上传成功";
    progress.value=100;
    //清空选中的文件
    selectedFile.value=null;
  })
}

//点击处理事件
const startUpload=async ()=>{
  if(!selectedFile.value){
    uploadResult.value="请选择文件";
    return;
  }
  const FormDataList=await chunksAndHash(selectedFile.value);
  await chunksConcurrency(FormDataList);
}

```

</script>

存在一个问题是，**竞态条件** (Race Condition)

多个并发任务 (pool 中的异步函数) 同时访问和修改共享的 `index` (`index++`) 。

JavaScript 的异步执行可能导致 `index++` 发生竞态条件：多个任务可能同时读取相同的 `index`，导致任务重复或跳过。

例如：两个任务同时读取 `index = 0`，都取 `FormDataList[0]`，然后 `index` 变为 2，跳过了 `FormDataList[1]`。

- **解决方法：**

- 使用原子操作（如锁）或同步访问 `index`。最简单的方法是用一个函数同步获取 `index`：

javascript

```
const getNextIndex = () => {  
  if (index < FormDataList.length) {  
    return index++;  
  }  
  return null;  
};
```

断点续传

我们已经完成分片上传/并发控制，现在我们来完成断点续传

1. 核心思路

断点续传的关键在于：

- **文件唯一标识**：用文件的 hash（你已经计算了 `fileHash`）作为文件的唯一 ID。
- **服务端记录已上传的分片**：服务端需要保存已经上传过的分片（比如保存到临时目录，并记录 `chunkIndex`）。
- **客户端查询已上传分片**：上传前，客户端先请求服务器，获取该 `fileHash` 已经上传过的分片索引列表。
- **跳过已上传分片**：客户端只上传未上传的分片。

客户端要做的支持


(1) 上传前，查询服务端已上传分片

js

```
// 查询服务端已上传分片
const getUploadedChunks = async (fileHash) => {
  const res = await API.get("http://localhost:3000/uploaded", {
    params: { fileHash }
  });
  // 服务器返回已上传的分片索引数组，例如 [0, 1, 2]
  return res.data.uploadedChunks || [];
};
```

(2) 在 `startUpload` 中调用

js

 复制代码

```
const startUpload = async () => {
  if (!selectedFile.value) {
    uploadResult.value = "请选择文件";
    return;
  }

  // 生成分片和 hash
  const FormDataList = await chunksAndHash(selectedFile.value);

  // 获取文件 hash（取第一个分片里的）
  const fileHash = FormDataList[0].get("fileHash");

  // 查询服务端已上传分片
  const uploadedChunks = await getUploadedChunks(fileHash);

  // 过滤掉已经上传的分片
  const needUploadList = FormDataList.filter(item => {
    const index = parseInt(item.get("chunkIndex"));
    return !uploadedChunks.includes(index);
  });

  if (needUploadList.length === 0) {
    uploadResult.value = "已上传完成";
    progress.value = 100;
    return;
  }

  // 并发上传未上传的分片
  await chunksConcurrency(needUploadList);
};
```

服务端需要的做的支持

3. 服务端要做的事

(假设你 Node.js + Express)

(1) 记录已上传分片

- 接收 `fileHash + chunkIndex` , 把分片存到 `uploads/${fileHash}/chunkIndex` 。
- 在服务端保存上传进度 (例如用数据库/文件系统保存) 。

(2) 提供查询接口

js


```
app.get("/uploaded", (req, res) => {
  const { fileHash } = req.query;
  const dir = path.join(__dirname, "uploads", fileHash);

  if (!fs.existsSync(dir)) {
    return res.json({ uploadedChunks: [] });
  }

  const files = fs.readdirSync(dir);
  // 文件名就是分片索引, 例如 0,1,2
  const uploadedChunks = files.map(name => parseInt(name));
  res.json({ uploadedChunks });
});
```


(3) 上传接口存分片

js

 复制代码

```
app.post("/upload", upload.single("chunk"), (req, res) => {
  const { fileHash, chunkIndex } = req.body;
  const chunkDir = path.join(__dirname, "uploads", fileHash);

  if (!fs.existsSync(chunkDir)) {
    fs.mkdirSync(chunkDir, { recursive: true });
  }

  const chunkPath = path.join(chunkDir, chunkIndex);
  fs.renameSync(req.file.path, chunkPath);

  res.json({ success: true });
});
```

4. 合并文件

所有分片上传完成后，前端请求服务端的 **合并接口**：

js

```
await API.post("http://localhost:3000/merge", {
  fileHash,
  filename: selectedFile.value.name,
  totalChunks: FormDataList.length
});
```

服务端在 `uploads/${fileHash}` 下，按序合并所有分片。

小结

1. 选中文件 → 生成 `fileHash + 分片`。
2. 前端调用 `/uploaded?fileHash=xxx` → 服务端返回已上传的分片。
3. 过滤掉已上传分片 → 并发上传剩余分片。
4. 全部分片上传完成后 → 调用 `/merge` 合并。

其实就是新增一个函数/接口

```
async function uploadFileWithResume(file, fileHash) {
  const chunkSize = 1 * 1024 * 1024; // 1MB
  const totalChunks = Math.ceil(file.size / chunkSize);

  // 获取已上传的分片
  const uploadedIndices = await checkUploadStatus(fileHash, totalChunks);

  // 分片并过滤掉已上传的
  const chunks = [];
  for (let i = 0; i < totalChunks; i++) {
    // 跳过已上传的分片
    if (uploadedIndices.includes(i)) {
      continue;
    }

    const start = i * chunkSize;
    const end = Math.min(start + chunkSize, file.size);
    const chunk = file.slice(start, end);
    chunks.push({
      index: i,
      blob: chunk
    });
  }
}
```

获取已经上传了的分片数据

如果传回到数组索引中有, 那么就跳过上传

重新计算分片位置

得到新的分片, 然后塞入

从后端获取已经上传的分片, 然后把分片List数组进行筛选, 继续上传未上传的

重传机制

这里说的是单片失败立即重试的逻辑，而不是整个文件的断点续传。这个可以在你现有的分片 + 并发逻辑上做一些改造。核心是给每个分片上传加 重试机制。

1 核心思路

- 每个分片上传时，如果失败（`catch` 到错误），重试 N 次。
- 并发上传时，失败的分片会继续被放回队列或递归重试。
- 成功上传的分片就不再上传。

2 修改 `uploadFile` 支持重试

重试次数

[复制代码](#)

```
js

const uploadFile = async (item, totalSize, uploadedSize, retryCount = 3) => {
  try {
    await API.post("http://localhost:3000/upload", item, {
      headers: { "Content-Type": "multipart/form-data" },
      onUploadProgress: (progressEvent) => {
        const loaded = progressEvent.loaded;
        const chunkAllSize = totalSize;
        progress.value = Math.round(((loaded + uploadedSize.value) / chunkAllSize) * 100);
      },
    });
  } catch (error) {
    if (retryCount > 0) {
      console.warn(`分片 ${item.get("chunkIndex")} 上传失败，重试剩余 ${retryCount} 次`);
      await uploadFile(item, totalSize, uploadedSize, retryCount - 1); // 递归重试
    } else {
      console.error(`分片 ${item.get("chunkIndex")} 上传失败，超过最大重试次数`);
      throw error; // 可以选择抛出错误，让并发池知道这个分片失败
    }
  }
}
```

自己手写的实现

```

<template>
  <input @change="handleFile" type="file">
</template>

<script setup>
import { ref } from 'vue'
import SparkMD5 from 'spark-md5'

const refFile = ref(null)
const maxSize=1024*1204*5
const concurrency=5

//选中文件
const handleFile=async (e)=>{
  refFile.value=e.target.files[0]

  //生成分片和Hash
  const {chunkList,hash}=await chunksAndHash(refFile.value)

  //并发上传
  await controlConcurrency(chunkList,concurrency)
}

//分片和生成Hash
const chunksAndHash=async (file)=>{

  //分片总数
  const chunkTotal=Math.ceil(file.size/maxSize)

  //分片之后的数组
  const chunkList=[]

  //文件依靠分片生成Hash是逐字节操作，需要转成ArrayBuffer
  const spark=new SparkMD5.ArrayBuffer()

  for(let i=0;i<chunkTotal;i++){

    //防止最后一块越界
    const start=i*maxSize
    const end=Math.min((i+1)*maxSize,file.size)

    //分片
    const chunk =file.slice(start,end)

    //准备好分片生成的formData数据
    const formData=new FormData()
    formData.append('chunks', chunk)
  }
}

```

```

formData.append('chunks', chunk)
formData.append('filename', file.name)
formData.append('chunkTotal', chunkTotal)
formData.append('chunkIndex', i)

//加入上传数据
chunkList.push(formData)

//因为Spark是逐字节操作需要编程二进制缓冲区arrayBuffer
const chunkArrayBuffer=await chunk.arrayBuffer()
spark.append(chunkArrayBuffer)

}

//循环结束计算Hash 停止然后计算得出Hash
const hash=spark.end()

//给上传数组中的每一个都增加上Hash属性
for(let i=0;i<chunkList.length;i++){
  chunkList[i].append('hash', hash)
}

return{
  chunkList,
  hash
}
}

//单纯的分片上传函数,这里只负责上传单个分片
const uploadChunks=async(fomDataChunk,retryCount=3)=>{
  return await fetch('/upload',{
    method:'POST',
    body:fomDataChunk
  }).then(res=>{
    if(res.status===200){
      window.alert("上传成功")
    }
  }).catch(async ()=>{
    if(retryCount>0){
      window.alert("上传失败,重试中")

      await uploadChunks(fomDataChunk,retryCount-1)
    }else{
      window.alert("超过最大重试次数,上传失败")
    }
  })
}
}

```

```
//控制并发池
const controlConcurrency=async(chunkList,concurrency)=>{
  //任务争抢编号
  let index=0
  //并发池
  const pool=Array(concurrency).fill(null).map(async ()=>{

    while(index<chunkList.length){
      const chunk=chunkList[index++]
      await uploadChunks(chunk)
    }

  })

  await Promise.all(pool).then(()=>{
    window.alert("所有分片上传完成")
  })
}

</script>
```