# 场景题

**统计前端请求耗时【热度: 609】**

在前端业务中，可以通过以下几种方法统计请求耗时：

**初级手段**

**一、使用fetch结合时间戳**

* 在发送请求前记录当前时间戳：
* 使用fetch发送请求：
* 在请求的.then()或.catch()中记录结束时间戳并计算耗时：

**二、使用XMLHttpRequest结合时间戳**

* 创建XMLHttpRequest对象并记录开始时间：
* 配置请求并发送：
* 在请求的onload、onerror等事件处理函数中记录结束时间并计算耗时：

**三、利用拦截器（axios）**

* 如果使用axios或类似的库，可以设置请求拦截器和响应拦截器：
* 在请求拦截器中记录开始时间，在响应拦截器中记录结束时间并计算耗时。

**总结**

上面都属于一些初级手段，因为还是在浏览器进程里面， 一旦出现长任务阻塞了浏览器， 这个统计就不太准确了。

**进阶手段 - Performance API**

**Performance API 可以用来统计请求耗时。**

Performance API 提供了一系列的性能测量工具，可以测量网页加载和运行过程中的各种性能指标。其中，可以通过以下方式来统计网络请求的耗时：

* 使用performance.timing：
  + performance.timing对象包含了网页加载过程中的各个时间点信息。可以通过计算不同时间点之间的差值来得到特定阶段的耗时。
  + 例如，可以计算responseEnd（服务器响应结束的时间）和requestStart（开始请求的时间）之间的差值来得到请求的耗时。
* 使用performance.getEntriesByType('resource')：
* 这个方法可以获取所有资源加载的性能条目。对于每个资源条目，可以获取其startTime（开始时间）和responseEnd（响应结束时间）等属性，从而计算出资源加载的耗时。
* 可以遍历这些条目，找到特定的网络请求资源，并计算其耗时。

示例:

// 计算页面加载过程中第一个请求的耗时

const timing = performance.timing;

const requestDuration = timing.responseEnd - timing.requestStart;

console.log(`First request took ${requestDuration} milliseconds.`);

// 遍历所有资源加载条目，找到特定请求并计算耗时

const resources = performance.getEntriesByType("resource");

for (const resource of resources) {

if (resource.name === "https://example.com/specific-resource") {

const resourceDuration = resource.responseEnd - resource.startTime;

console.log(`Specific resource request took ${resourceDuration} milliseconds.`);

break;

}

}

**高级手段 - Web Worker**

**Web Worker 可以用于统计请求耗时。**

以下是一种使用 Web Worker 统计请求耗时的方法：

1.创建一个 Web Worker 文件，例如worker.js：

self.onmessage = function (event) {

const url = event.data.url;

const startTime = performance.now();

fetch(url)

.then((response) => {

const endTime = performance.now();

const duration = endTime - startTime;

self.postMessage({ duration });

})

.catch((error) => {

self.postMessage({ error: `Error fetching ${url}: ${error}` });

});

};

2.在主页面中使用 Web Worker：

const worker = new Worker("worker.js");

const url = "your-api-url";

worker.postMessage({ url });

worker.onmessage = function (event) {

if (event.data.duration) {

console.log(`Request to ${url} took ${event.data.duration} milliseconds.`);

} else {

console.error(event.data.error);

}

};

在这个例子中，Web Worker 负责发送请求并计算耗时，然后将结果发送回主页面。这样可以在不阻塞主页面 UI 线程的情况下进行请求耗时统计。

## js超过Number最大值的数怎么处理

Number.MAX\_VALUE

### 场景

* 大数据计算
  + 金融计算
  + 科学计算
  + 数据分析
* 格式展示
* 用户输入(输入直接限制大小)

### 方法

* BigInt
  + const bigNum = new BitInt(100000000000000000000000000)
* dicemal.js
* big.js

### 总结

* BigInt处理大数据
* dicemal处理
* 格式化 格式化成用户好读的格式(100000000 ~ 1亿)
* 表单校验 不允许用户输入超过多少位数字

## 如何解决页面请求接口大规模并发问题

**滑动窗口** 算法,专门来控制流量的

### 方案

* 请求队列
* 防抖节流
* 分页加载(滚动加载,分页加载,可视区绘制,按需加载)

## 大文件上传

背景: 在做算法平台时经常需要上传大文件(1G以上)

经常遇到一些问题

-网络断开后,之前上传的没了

-网络波动,传着传着没了

-关机了,想接着传,做不到

* 断点续传
* 断点重连重传
* 切片上传

### 方案

* 前端切片 chunk 1024M(1048576K), 500K const size = 1048576 / 500
* 将切片传递给后端,切的片要取名: hash, index顺序
* 后端组合切片

### 进阶

* 前端切片: 主线程做卡顿,web-worker 多线程切片,处理完后交给主线程发送
* 切完后,将切片存储到 IndexedDB内,下次用户进来之后,检测下是否存在未上传的切片,有就继续上传
* websocket 实时通知,和请求序列的控制 wss
* 整体说一说我主导的大文件上传器整体逻辑
  + 组件设计
  + props,事件,状态
  + 拖拽上传,多文件选择
  + 通用化不同文件的上传,上传统一协议

## 在前端怎么实现页面截图

* 预览
* 截图
* 导出文档/图片

### 方案

* 截图
  + canvas
  + puppeteer(无头浏览器,无头表格,无头UI,headless)
  + html2canvas
* 上传图片
* 全页面截图
* 局部截图
* 特定区域截图

### 实现

* 通用性 selector body,head,dom
* 设计具体协议
  + 函数式
  + 组件式
  + 隐藏canvas
* 代码编写