

一.宏观视角下的浏览器

1.多进程浏览器

I.进程与线程的概念

进程是操作系统为了执行程序功能所分配的一个实例，一个应用程序可以分配多个进程，**每个进程有独自の内存空间，进程之间的内容相互隔离**。比如现代浏览器的就有多个进程，包裹网络服务进程，页面服务进程，GPU 进程等等。

比进程更小的单位是线程，一个进程可以被分为多个小任务，这个小任务就是线程。在多线程环境中，多个线程可以并行执行，即在同一时间下可以执行多个线程。而单线程环境下线程只可以并行处理。**线程之间共享进程之间的数据**，当一个线程执行出错整个进程就会崩溃。

II.单进程浏览器

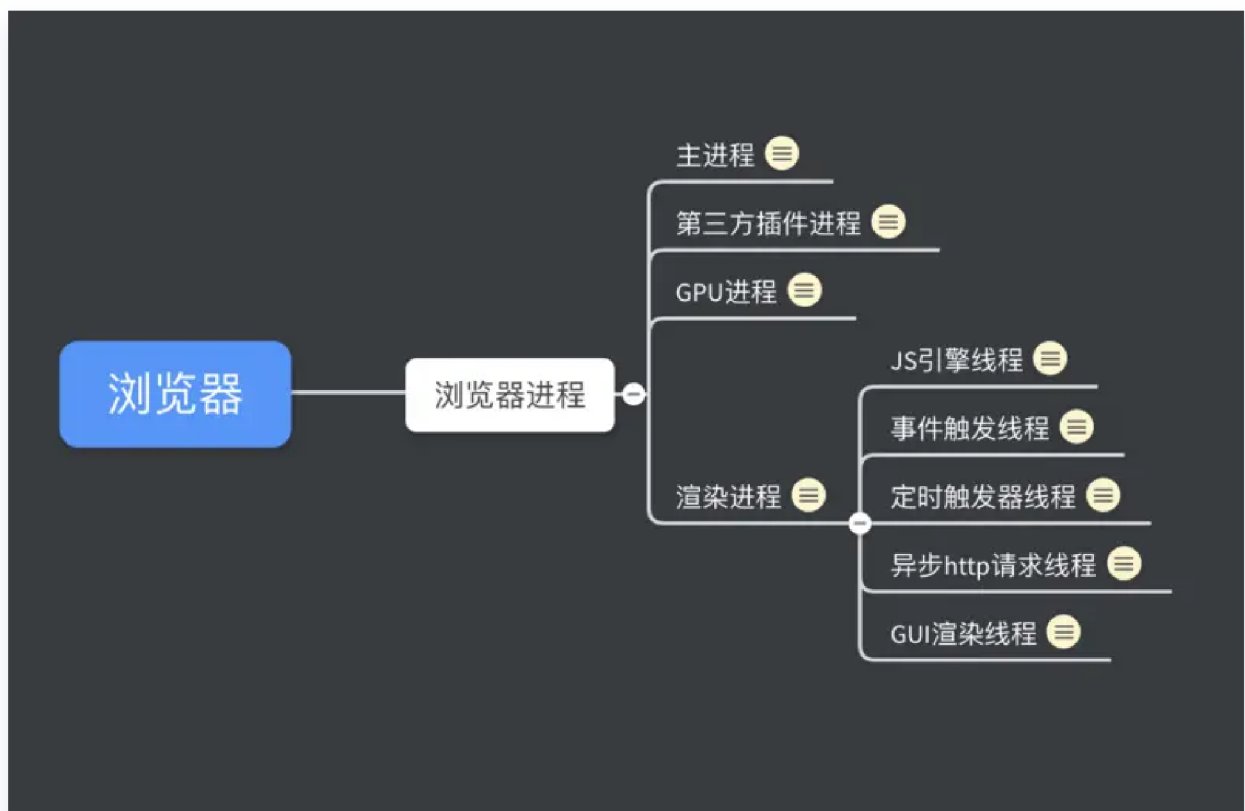
单进程浏览器自然会有很多问题，比如插件也在单进程中，一个插件崩溃就会让整个浏览器崩溃；由于单进程无法并行工作，一个功能占据的时间进程的时间过长其它功能都无法正常使用，同时单进程浏览器也会引发安全问题，进程在运行恶意的插件的时候就可以对整个浏览器进行控制，可以获取密码等等。

III.多进程浏览器

多个进程之间是通过 IPC 机制进行通信。多进程解决了很多问题，比如 JavaScript 运行在渲染进程中，即使 JavaScript 阻塞了渲染进程，影响到的也只是当前的渲染页面，而并不会影响浏览器和其他页面，因为其他页面的脚本是运行在它们自己的渲染进程中的。

也解决了安全问题，采用多进程架构的额外好处是可以使用[安全沙箱](#)，你可以把沙箱看成是操作系统给进程上了一把锁，沙箱里面的程序可以运行，但是不能在你的硬盘上写入任何数据，也不能在敏感位置读取任何数据，例如你的文档和桌面。[Chrome 把插件进程和渲染进程锁在沙箱里面](#)，这样即使在渲染进程或者插件进程里面执行了恶意程序，恶意程序也无法突破沙箱去获取系统权限。

现代Chrome浏览器的进程：



1. GUI 渲染线程

- 负责渲染浏览器界面,解析 HTML,CSS,构建 DOM 树和 RenderObject 树,布局 and 绘制等。
- 当界面需要重绘（Repaint）或由于某种操作引发回流(reflow)时,该线程就会执行。
- 注意,GUI 渲染线程与 JS 引擎线程是互斥的,当 JS 引擎执行时 GUI 线程会被挂起（相当于被冻结了）,GUI 更新会被保存在一个队列中等到 JS 引擎空闲时立即被执行。

2. JS 引擎线程

- Javascript 引擎,也称为 JS 内核,负责处理 Javascript 脚本程序。（例如 V8 引擎）
- JS 引擎线程负责解析 Javascript 脚本,运行代码。
- JS 引擎一直等待着任务队列中任务的到来,然后加以处理,一个 Tab 页（renderer 进程）中无论什么时候都只有一个 JS 线程在运行 JS 程序。
- 注意,GUI 渲染线程与 JS 引擎线程是互斥的,所以如果 JS 执行的时间过长,这样就会造成页面的渲染不连贯,导致页面渲染加载阻塞。

3. 事件触发线程

- 归属于浏览器而不是 JS 引擎,用来控制事件循环（可以理解,JS 引擎自己都忙不过来,需要浏览器另开线程协助）
- 当对应的事件符合触发条件被触发时,该线程会把事件添加到待处理队列的队尾,等待 JS 引擎的处理
- 注意,由于 JS 的单线程关系,所以这些待处理队列中的事件都得排队等待 JS 引擎处理（当 JS 引擎空闲时才会去执行）

4. 定时触发器线程

- 传说中的 setInterval 与 setTimeout 所在线程
- 浏览器定时计数器并不是由 JavaScript 引擎计数的,（因为 JavaScript 引擎是单线程的,如果处于阻塞线程状态就会影响计时的准确）
- 因此通过单独线程来计时并触发定时（计时完毕后,添加到事件队列中,等待 JS 引擎空闲后执行）

5. 异步 http 请求线程

- 在 XMLHttpRequest 在连接后是通过浏览器新开一个线程请求。
- 将检测到状态变更时,如果设置有回调函数,异步线程就产生状态变更事件,将这个回调再放入事件队列中。再由 JavaScript 引擎执行。

一些问题：

即使是如今的多进程架构，我偶尔还会碰到一些由于单个页面卡死最终崩溃导致所有页面崩溃的情况，请问这是为什么呢

通常情况下是一个页面使用一个进程，但是，有一种情况，叫"同一站点(same-site)"，比如下面这三个：<https://time.geekbang.org> <https://www.geekbang.org> <https://www.geekbang.org:8080> 都是属于同一站点，因为它们的协议都是https，而根域名也都是geekbang.org。Chrome的默认策略是，每个标签对应一个渲染进程。但是如果从一个页面打开了新页面，而新页面和当前页面属于同一站点时，那么新页面会复用父页面的渲染进程。官方把这个默认策略叫process-per-site-instance。直白的讲，就是如果几个页面符合同一站点，那么他们将被分配到一个渲染进程里面去。所以，这种情况下，一个页面崩溃了，会导致同一站点的页面同时崩溃，因为他们使用了同一个渲染进程。为什么要让他们跑在一个进程里面呢？因为在一个渲染进程里面，他们就会共享JS的执行环境，也就是说A页面可以直接在B页面中执行脚本。

二.浏览器安全

1.同源策略

一个url由三部分组成:协议，域名，端口。

只有当协议，域名，端口都一致的时候，才被称为同源。

而同源策略规定，只有客户端和服务端处于同源的情况下，浏览器才会接受响应。

客户端的请求会直接发送到服务器 ===》 然后服务器处理完毕响应客户端 ===》 客户端检测是否同源 ===》 发现不同源，报错

2.代理解决跨域

产生跨域问题的前提是浏览器环境，代理服务器不受浏览器的限制，所以可以和远程服务器正常通信。同时代理服务器和html网页在相同的域名、端口下，所以不存在跨域，代理服务器自然可以把数据传给页面。

代理服务器会在请求头中添加一些额外的信息，其中包括目标服务器的主机名（Host）和客户端原始的请求信息等。目标服务器收到请求后，会根据请求头中的主机名以及请求的具体路径等信息来确定客户端需要的是哪个资源，并进行相应的处理和响应。所以也能知道客户端到底想要的是哪个资源。

3.使用node.js实现一个代理服务器

首先，确保你已经安装了 `http-proxy-middleware` 模块，如果没有安装，可以通过以下命令安装：

```
npm install http-proxy-middleware --save
```

然后，创建一个 JavaScript 文件，例如 `proxyServer.js`，并添加以下代码：

```
const express = require('express');
const { createProxyMiddleware } = require('http-proxy-middleware');

const app = express();

// 定义要代理的目标地址
```

```
const target = 'http://example.com'; // 修改为你要代理的目标地址

// 创建代理中间件
const proxyMiddleware = createProxyMiddleware({
  target,
  changeOrigin: true, // 更改请求头中的 host 为目标地址的 host
  pathRewrite: {
    // 重写路径，如果有需要的话
    '^/api': '/', // 将请求路径中的 /api 替换为空，比如
    // /api/users 将会被代理到目标地址的 /users
  },
  // 其他配置项可以根据需要添加，比如 header 配置等
});

// 将代理中间件挂载到 /api 路径下
app.use('/api', proxyMiddleware);

// 启动服务器监听指定端口
const port = 3000; // 修改为你想要监听的端口号
app.listen(port, () => {
  console.log(`Proxy server is running on port ${port}`);
});
```

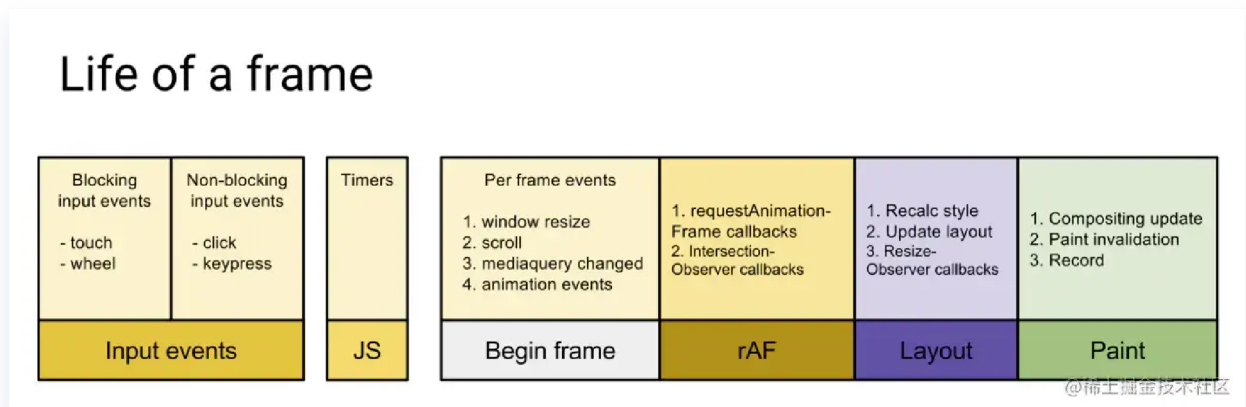
在上面的代码中，你需要修改 `target` 变量的值为你想要代理的目标地址，然后根据需要修改 `pathRewrite` 对象中的路径重写规则。

保存文件后，在命令行中运行 `node proxyServer.js` 启动代理服务器。

现在，你可以通过访问 `http://localhost:3000/api/your-api-endpoint` 来访问目标地址的 API，并且跨域问题应该已经被解决了。

三.浏览器的每一帧都做了什么

通常刷新频率和显示器帧率相同，大部分显示器帧率是60fps， $\approx 16\text{ms}$ 每帧。



- 用户交互输入事件（Input events），能够让用户得到最早的反馈。
- JavaScript 引擎线程解析执行：执行定时器（Timers）事件的回调
- 帧开始（Begin frame）：每一帧事件（Per frame events），例如 window resize、scroll 或 media query change
- 执行请求动画帧 rAF（requestAnimationFrame），即在每次绘制之前，会执行 rAF 回调。

`requestAnimationFrame` 的回调函数会在浏览器下一次重绘之前执行，浏览器会根据当前设备的屏幕刷新率来调整重绘频率，从而保证动画的流畅性，并且在隐藏或不可见的页面中不会执行动画，节省了资源。

- 页面布局（Layout）：计算样式（Recalculate style）和更新布局（Update Layout）。即这个元素的样式是怎样的，它应该在页面如何展示。

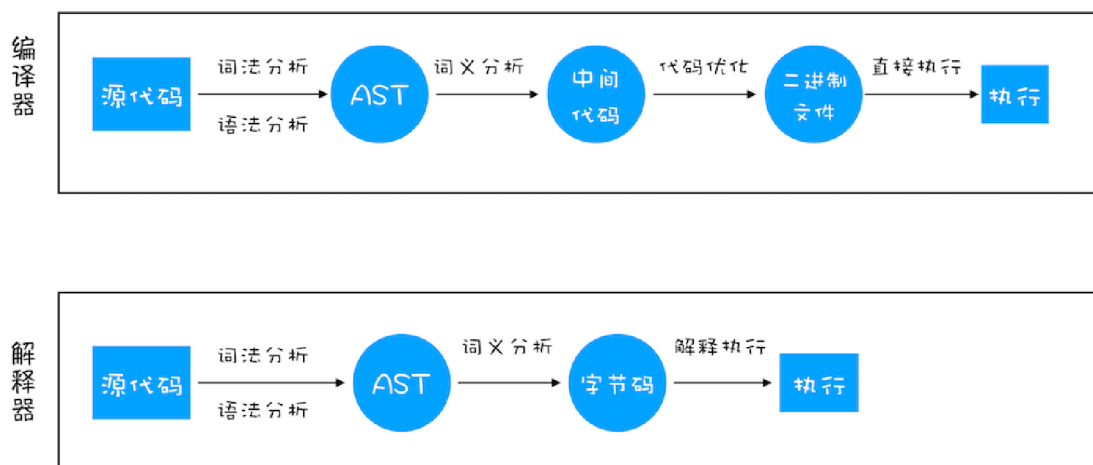
- 绘制渲染（Paint）：合成更新（Compositing update）、重绘部分节点（Paint invalidation）和 Record。得到树中每个节点的尺寸与位置等信息，浏览器针对每个元素进行内容填充。
- 上述6个阶段完成，就处于 空闲阶段（Idle Peroid）会执行 RIC (RequestIdleCallback)注册的任务。

js线程和GUI渲染线程都属于浏览器的渲染进程，但是两者是互斥关系。如果在某个阶段执行任务特别长，时间已经明显超过了16ms，那么就会阻塞页面的渲染，从而出现卡顿现象。

四.V8引擎

在现有的javascript引擎中，V8引擎绝对是其中的佼佼者，chrome和node底层都使用了V8引擎，其中chrome的市场占有率已经达到70%，而node更是前端工程化以及扩展边界的核心支柱

CPU并不认识我们的js代码，而不同的CPU只认识自己对应的指令集，javascript引擎就是负责将js代码编译成CPU认识的指令集，当然除了编译之外还要负责执行以及内存的管理。大家都知道js是解释形语言，由引擎直接读取源码，一边编译一边执行，这样效率相对较低，而编译形语言（如c++）是把源码直接编译成可直接执行的代码执行效率更高。



执行过程

• 生成抽象语法树（AST）和执行上下文

将源代码转换为抽象语法树，并生成执行上下文

什么是抽象语法树？

高级语言是开发者可以理解的语言，但是让编译器或者解释器来理解就非常困难了。对于编译器或者解释器来说，它们可以理解的就是 AST 了。所以无论你使用的是解释型语言还是编译型语言，在编译过程中，它们都会生成一个 AST

怎么形成AST？

第一阶段是**分词（tokenize）**，又称为**词法分析**，其作用是将一行行的源码拆解成一个个 token。所谓 token，指的是**语法上不可能再分的、最小的单个字符或字符串**

第二阶段是**解析（parse）**，又称为**语法分析**，其作用是将上一步**生成的 token 数据，根据语法规则转为 AST**。如果源码符合语法规则，这一步就会顺利完成。但如果源码存在语法错误，这一步就会终止，并抛出一个“语法错误”。

什么是执行上下文？

JavaScript（JS）语言的执行上下文是指在代码执行过程中，JavaScript引擎用来管理执行代码的环境。它包含了代码运行时所需的所有信息，包括变量、函数、作用域链等。

- **全局执行上下文（Global Execution Context）**：当代码在全局作用域中执行时，会创建全局执行上下文。它是默认的执行上下文，所有不在函数内部的代码都在全局上下文中执行。

- 函数执行上下文（Function Execution Context）：每当调用一个函数时，都会创建一个函数执行上下文。每个函数都有自己的执行上下文，用于管理函数内部的变量、函数声明等。
- eval执行上下文（Eval Execution Context）：通过eval()函数动态执行的代码会创建一个eval执行上下文。不推荐使用eval，因为它会导致代码的可读性和性能问题。

• 生成字节码

有了AST和执行上下文后，那接下来的第二步，解释器 Ignition 就登场了，它会根据AST生成字节码，并解释执行字节码。

什么是字节码？为什么要有字节码？

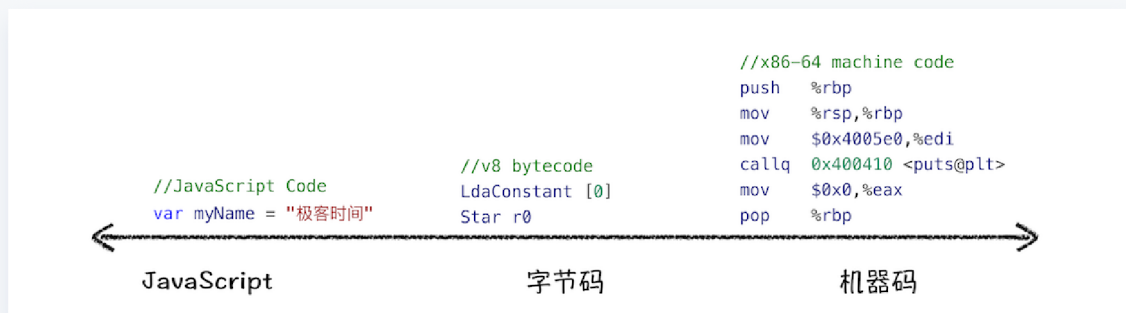
其实一开始V8并没有字节码，而是直接将AST转换为机器码，由于执行机器码的效率是非常高效的，所以这种方式在发布后的一段时间内运行效果是非常好的。

但是随着Chrome在手机上的广泛普及，特别是运行在512M内存的手机上，内存占用问题也暴露出来了，因为**V8需要消耗大量的内存来存放转换后的机器码。**

为了解决内存占用问题，V8团队大幅重构了引擎架构，引入字节码，并且抛弃了之前的编译器，最终花了将近四年的时间，实现了现在的这套架构。

那什么是字节码呢？为什么引入字节码就能解决内存占用问题呢？**字节码就是介于AST和机器码之间的一种代码。但是与特定类型的机器码无关，字节码需要通过解释器将其转换为机器码后才能执行。**

理解了什么是字节码，我们再来对比下高级代码、字节码和机器码，可以参考下图：



从图中可以看出，机器码所占用的空间远远超过了字节码，所以使用字节码可以减少系统的内存使用。

• 执行代码

通常，如果有一段第一次执行的字节码，解释器 Ignition 会逐条解释执行。到了这里，相信你已经发现了，解释器 Ignition 除了负责生成字节码之外，它还有另外一个作用，就是解释执行字节码。在 Ignition 执行字节码的过程中，如果发现有热点代码（HotSpot），比如一段代码被重复执行多次，这种就称为热点代码，那么后台的编译器 TurboFan 就会把该段热点的字节码编译为高效的机器码，然后当再次执行这段被优化的代码时，只需要执行编译后的机器码就可以了，这样就大大提升了代码的执行效率。这就是 JIT 技术的原理。

其实字节码配合解释器和编译器是最近一段时间很火的技术，比如 Java 和 Python 的虚拟机也都是基于这种技术实现的，我们把这种技术称为即时编译（JIT）。具体到 V8，就是指解释器 Ignition 在解释执行字节码的同时，收集代码信息，当它发现某一部分代码变热了之后，TurboFan 编译器便闪亮登场，把热点的字节码转换为机器码，并把转换后的机器码保存起来，以备下次使用。对于 JavaScript 工作引擎，除了 V8 使用了“字节码 +JIT”技术之外，苹果的 SquirrelFish Extreme 和 Mozilla 的 SpiderMonkey 也都使用了该技术。

四.浏览器中页面之间的通信

1.同源情况下

localStorage

一个窗口更新localStorage，另一个窗口监听window对象的"storage"事件，来实现通信。

注：两个页面要同源（URL的协议、域名和端口相同）

```
// 本窗口的设值代码
localStorage.setItem('aaa', (Math.random()*10).toString())

// 其他窗口监听storage事件
window.addEventListener("storage", function (e) {
    console.log(e)
    console.log(e.newValue)
})
```

- 适用于页面之间需要共享少量数据，并且不需要实时更新的情况。

- 限制：存储容量有限，通常在 5MB 左右，且只能存储字符串类型的数据，不适合存储大量或敏感数据。

iframe

利用iframe，同源情况下可以实现互相的dom节点的访问

cookie

同源情况下 cookie 也是共享的，可以实现简单的通信，通过 document.cookie 就可以获得了。

SharedWorker:

场景：适用于多个页面需要共享数据或协同工作的情况，例如多个标签页需要访问同一份数据。

限制：SharedWorker 只能在同一站点的不同上下文之间共享，且在一些老版本浏览器中可能不支持。

SharedWorker 是 HTML5 提供的一种特殊的 **Web Worker**，它可以被多个浏览上下文（例如同一域名下的多个页面或不同域名下的页面）共享。Web Workers 是一种在后台运行的 JavaScript 线程，用于在不阻塞主页面的情况下执行耗时的任务。

与普通的 **Worker** 不同，**SharedWorker** 可以被多个浏览上下文共享，包括同源或不同源的页面。这使得 **SharedWorker** 成为了实现多个浏览上下文之间共享数据和通信的一种有效方式。

下面是一个简单的示例，演示了如何使用 **SharedWorker** 实现两个页面之间的通信。

首先，我们创建一个名为 **shared-worker.js** 的共享工作线程文件，其中包含共享的代码逻辑：

```
// shared-worker.js

// 定义一个变量来保存共享的状态
let sharedState = 0;

// 监听来自页面的消息
onconnect = function(event) {
  // 获取与页面建立连接的端口
  let port = event.ports[0];

  // 监听页面发送的消息
  port.onmessage = function(event) {
    // 从页面接收到消息后，更新共享状态并发送回复消息
    sharedState += event.data;
    port.postMessage(sharedState);
  };
};
```

然后，我们在两个不同的 HTML 页面中分别使用 `SharedWorker` 来共享相同的工作线程：

```
<!-- page1.html -->

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Page 1</title>
</head>
<body>
  <h1>Page 1</h1>
```

```

<button onclick="sendMessageToWorker()">Send Message to
Worker</button>

<script>
  // 创建一个 SharedWorker, 并与之建立连接
  const worker = new SharedWorker('shared-worker.js');
  const workerPort = worker.port;

  // 当接收到来自共享工作线程的消息时, 更新页面上的状态
  workerPort.onmessage = function(event) {
    document.querySelector('h1').textContent = 'Page 1 -
Shared State: ' + event.data;
  };

  // 页面向共享工作线程发送消息
  function sendMessageToWorker() {
    // 向共享工作线程发送消息
    workerPort.postMessage(1);
  }
</script>
</body>
</html>

```

```

<!-- page2.html -->

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Page 2</title>
</head>
<body>
  <h1>Page 2</h1>

```



```
<button onclick="sendMessageToWorker()">Send Message to  
Worker</button>  
  
<script>  
  // 创建一个 SharedWorker, 并与之建立连接  
  const worker = new SharedWorker('shared-worker.js');  
  const workerPort = worker.port;  
  
  // 当接收到来自共享工作线程的消息时, 更新页面上的状态  
  workerPort.onmessage = function(event) {  
    document.querySelector('h1').textContent = 'Page 2 -  
Shared State: ' + event.data;  
  };  
  
  // 页面向共享工作线程发送消息  
  function sendMessageToWorker() {  
    // 向共享工作线程发送消息  
    workerPort.postMessage(-1);  
  }  
</script>  
</body>  
</html>
```

在这个例子中, 我们创建了两个 HTML 页面: `page1.html` 和 `page2.html`, 它们分别创建了一个 `SharedWorker` 实例, 并与同一个 `shared-worker.js` 文件建立连接。当页面中的按钮被点击时, 页面将向共享工作线程发送消息, 并接收并更新共享状态的变化。这样, 两个页面就能够共享同一个工作线程, 实现了跨页面通信。

直接引用

其实就是直接获取对方DOM，**适用于两个页面在同一域**；可以传递对象数据（对象数据使用 instanceof 做类型判断时有坑）；参考 window.open；例：

```
// 父页面获取子iframe
document.getElementById('iframe的id').contentWindow.document

// 子iframe获取父页面
window.parent.document
```

不同源

WebSocket

WebSocket 可以用于**不同源**之间的通信，**需要服务器端进行相应的配置**以允许跨源访问。

- 场景：适用于实时的双向通信，例如在线聊天、实时游戏等应用场景。
- 限制：需要在服务器端实现 WebSocket 服务器，且与传统的 HTTP 服务器不同，需要单独维护 WebSocket 连接。

```
// 在客户端建立 WebSocket 连接
const socket = new WebSocket('ws://example.com/chat');

// 监听消息
socket.onmessage = function(event) {
  console.log('Received message: ' + event.data);
};

// 发送消息
socket.send('Hello, WebSocket server!');
```

为啥不用http服务器或者https服务器呢？

ws全双工通信，也就是说客户端可以一边向服务器发送数据（发消息），一边向服务器拿数据（接收消息），而http服务器的情况下，客户端需要发送一次网络请求拿到数据，还需要再发送一次网络请求接收数据，并且只能是串行的。这样实现两个页面之间的通信效率就太差了。并且 HTTP 请求与响应可能会包含较长的头部，其中真正有效的数据可能只是很小的一部分，所以这样会消耗很多带宽资源。

PostMessage API:

场景：适用不同源的页面、**跨域 iframe 之间的通信**，可以传递任意类型的数据。

限制：需要知道目标窗口的引用，存在安全性考虑，需要谨慎处理来自其他窗口的消息以防止 XSS 攻击。

```
// 在页面 A 中发送消息给 iframe B
const iframe = document.getElementById('iframeB');
iframe.contentWindow.postMessage('Hello from Page A!',
  'http://example.com');
```

window.name

浏览器窗口有window.name属性。这个属性的最大特点是，无论是否同源，只要在同一个窗口里，前一个网页设置了这个属性，后一个网页可以读取它。

父窗口先打开一个子窗口，载入一个不同源的网页，该网页将信息写入window.name属性。

```
window.name = data;
```

接着，子窗口跳回一个与主窗口同域的网址。

```
window.location.href = 'http://parent.url.com/xxx.html';
```

然后，主窗口就可以读取子窗口的window.name了。

```
var data = document.getElementById('iframe的id').contentWindow.name;
```

这种方法的优点是，window.name容量很大，可以放置非常长的字符串；缺点是必须监听子窗口window.name属性的变化，影响网页性能。

片段识别符

片段标识符（fragment identifier）指的是，URL的 # 号后面的部分，比如 `http://example.com/x.html#fragment` 的 `#fragment`。如果只是改变片段标识符，页面不会重新刷新。

父窗口可以把信息，写入子窗口的片段标识符。

```
var src = originURL + '#' + data;  
document.getElementById('myIFrame').src = src;
```

子窗口通过监听 `hashchange` 事件得到通知。

```
window.onhashchange = checkMessage;

function checkMessage() {
  var message = window.location.hash;
  // ...
}
```

同样的，子窗口也可以改变父窗口的片段标识符。

```
parent.location.href= target + "#" + hash;
```

五.axios二次封装以及axios的原理

1.一般是先创建一个axios实例：

```
const instance = axios.create({
  baseURL: 'https://some-domain.com/api/',
  timeout: 1000,
  headers: {'X-Custom-Header': 'foobar'}
});
```

- **baseURL**: 用于设置请求的基础URL，会被添加到相对URL之前。
- **headers**: 设置请求的自定义headers。
- **timeout**: 设置请求超时时间，单位是毫秒。

- **withCredentials**: 表示跨域请求时是否需要使用凭据（如cookies、授权头或TLS客户端证书）。
- **transformRequest**: 在发送请求之前对请求数据进行处理函数。
- **transformResponse**: 在接收到响应数据之后对响应数据进行处理函数。
- **paramsSerializer**: 序列化请求参数的函数。
- **responseType**: 表示服务器响应的数据类型，可以是'arraybuffer', 'blob', 'document', 'json', 'text', 'stream'等。
- **xsrCookieName**: 用作 XSRF 令牌的值的cookie的名称。
- **xsrHeaderName**: 包含 XSRF 令牌的值的 HTTP 头的名称。
- **maxContentLength**: 限制响应内容的最大长度。
- **maxRedirects**: 设置在Node.js中要遵循的重定向的最大数量。
- **httpAgent**: 自定义 HTTP 代理。
- **httpsAgent**: 自定义 HTTPS 代理。
- **proxy**: 设置代理服务器的主机和端口号。//跨域问题

2.根据创建的实例添加请求拦截器以及响应拦截器

拦截器的use方法的参数有两个，都是回调函数。

请求拦截器的第一个回调函数指的是在发送请求之前做些什么，第二个参数是请求错误之后做什么

响应拦截器的第一个回调函数指的是在响应成功后对响应数据做什么，第二个回调指的是响应错误之后做些什么

```
// 请求拦截器
instance.interceptors.request.use(
  config => {
    if (config.showLoading) showLoading()
    const token = storage.get('token')
    if (token) {
      config.headers.Authorization = 'Bearer ' + token
    }
  },
  error => {
    // 这里可以处理请求错误
  }
)
```

```

    }
    if (env.mock) {
      config.baseUrl = env.mockApi
    } else {
      config.baseUrl = env.baseApi
    }
    return {
      ...config
    }
  },
  (error: AxiosError) => {
    return Promise.reject(error)
  }
)

// 响应拦截器
instance.interceptors.response.use(
  response => {
    const data: Result = response.data
    hideLoading()
    if (response.config.responseType === 'blob') return
    response
    if (data.code === 500001) {
      message.error(data.msg)
      storage.remove('token')
      location.href = '/login?callback=' +
encodeURIComponent(location.href)
    } else if (data.code !== 0) {
      if (response.config.showError === false) {
        return Promise.resolve(data)
      } else {
        message.error(data.msg)
        return Promise.reject(data)
      }
    }
  }
)
return data.data

```



```
    },  
    error => {  
      hideLoading()  
      message.error(error.message)  
      return Promise.reject(error.message)  
    }  
  )  
}
```

3.然后默认导出实例的get方法， post方法（写成对象的形式）， 我们通常用闭包的形式对get方法和post方法做增强

```
export default {  
  get(url, params, options) {  
    return instance.get(url, { params, ...options })  
  },  
  post(url, params, options): Promise<T> {  
    return instance.post(url, params, options)  
  }  
}
```

4.对于项目中各种形形色色的网络请求， 我们可以导入默认导出的封装好的axios对象， 然后调用对象的get或者post方法传入ulr, params参数等等从而发送网络请求

源码解读

1.axios.create()原理:

把axios本身默认的配置与我们传入的配置参数进行合并（merge）， 然后调用底层创建实例的方法createInstance， 参数就是我们合并好的参数。createInstance方法核心就是把Axios原型对象上的request方法绑定到创建的实例上然后返回这个实例

axios的常使用的api 请求方法在这个request方法上。

2.Axios.prototype.request:

1.将各种网络请求的api（get, post, push等等）都挂载到prototype上，并且均调用request方法。类似于提供用户使用的接口，但是本质只是给request方法起了别名。

2.请求拦截器和响应拦截器：核心实现是**链式调用**，创建一个叫做chain的数组，把设置的请求拦截器的成功处理函数、失败处理函数放到数组最前面（unshift方法），把设置的响应拦截器的成功处理函数、失败处理函数放到数组最后面（push方法），只要数组不为空，进入循环，每次取两个出来组成promise.then执行，并且将then方法的结果传递给一个叫promise的变量。参数分别是then方法的第一个参数和then方法的第二个参数。（这也是为什么我们写拦截器的两个参数这么神似then方法的原因）最后返回整个request返回promise对象。

问题：但是拦截器也用同步和异步的区别，如果拦截器是同步的，不会出现问题，但是**异步**可能会出现问题：

- **异步操作：**

请求拦截器中可能包含异步操作，例如发送网络请求获取认证 token、读取本地文件等。如果这些异步操作在请求发送之前没有完成，那么请求就会因为等待异步操作而延迟发送。

- **长时间的宏任务执行：**

在某些情况下，请求拦截器中的某些代码可能会导致长时间的宏任务执行，例如复杂的计算、大量的数据处理等。如果这些操作耗时较长，会阻塞 JavaScript 主线程，从而延迟了请求的发送时机。

在这两种情况下，原始的代码可能没有正确处理异步操作或长时间的宏任务执行，导致**请求不能及时发送**。

解决的办法：

本质其实没有解决问题，但是对同步的流程进行了优化。因为异步的流程和之前的一致，只要你拦截器异步了，并且耗时过长，那么只能等待很长的耗时才可以发送网络请求。

如果请求拦截器中一个是异步的，那么就执行异步的流程，异步的流程与之前的一致，都是把设置的请求拦截器的成功处理函数、失败处理函数放到数组最前面（unshift方法），把设置的响应拦截器的成功处理函数、失败处理函数放到数组最后面（push方法），只要数组不为空，进入循环，每次取两个出来组成promise.then执行，分别是then方法的第一个参数和then方法的第二个参数。

如果请求拦截器均是同步的，执行同步的流程。核心思想是不用promise.then来执行请求拦截器的处理函数，（避免过早创建微任务，导致当前宏任务过长使得网络发送有延迟），而是先通过请求拦截器的成功处理函数得到新的配置队先后，然后带上这个新的配置对象直接用分发请求的方式发起真正的ajax请求。

3.什么是分发请求？

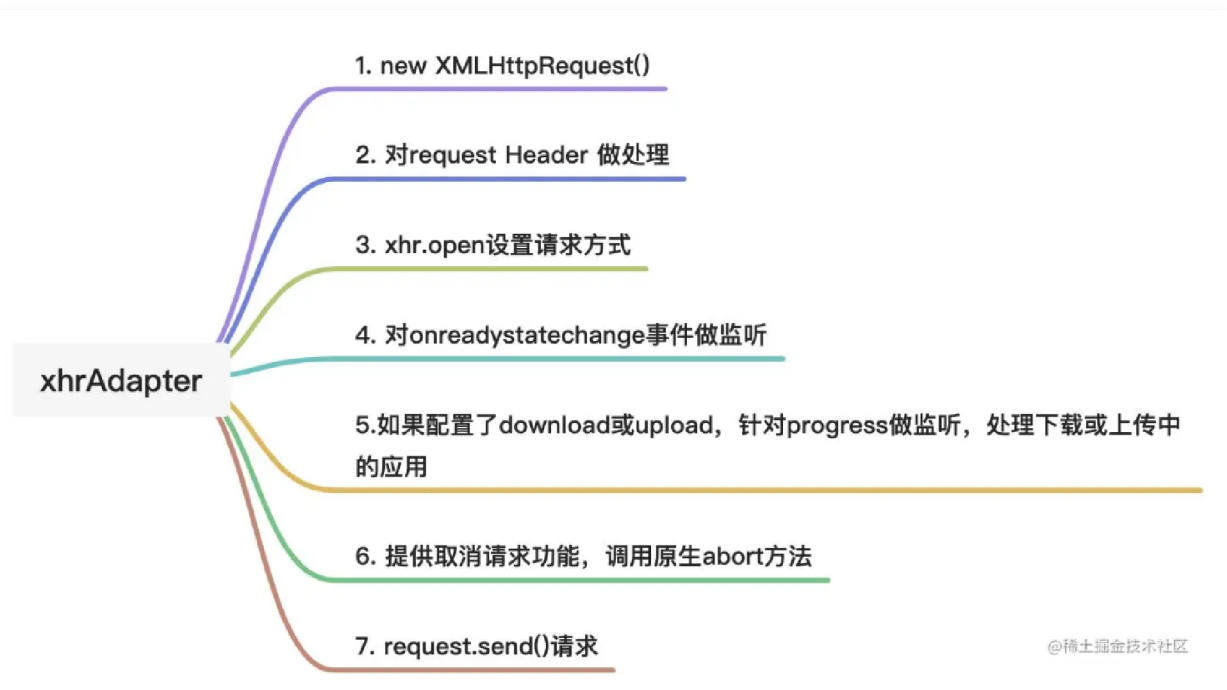
`dispatchRequest`，调用 `adapter` 适配器发起真正的网络请求，针对浏览器环境发起ajax请求，node环境发起http请求。

4.什么是适配器adaptor

根据不同的环境使用不同的封装过的函数发起网络请求，针对浏览器环境发起ajax请求（已经封装过了），node环境发起http请求（也已经封装过了）。

5.如何封装ajax请求

把请求的结果封装为一个promise对象



下面是一个简易的用 Promise 封装 Ajax 请求的函数示例:

```
function ajax(url, method = 'GET', data = null) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open(method, url, true);
    xhr.setRequestHeader('Content-Type',
      'application/json');

    xhr.onload = function() {
      if (xhr.status ≥ 200 && xhr.status < 300) {
        resolve(xhr.responseText);
      } else {
        reject(new Error(`请求失败: ${xhr.status} -
          ${xhr.statusText}`));
      }
    };

    xhr.onerror = function() {
      reject(new Error('网络错误'));
    };

    xhr.send(data);
  });
}
```

```
xhr.send(data ? JSON.stringify(data) : null);
});
}

// 示例用法
ajax('https://api.example.com/data')
.then(response => {
  console.log('成功获取数据: ', response);
})
.catch(error => {
  console.error('请求失败: ', error);
});
```

这个函数接受三个参数：`url` 表示请求的地址，`method` 表示请求的方法，默认为 `'GET'`，`data` 表示请求的数据，默认为 `null`。函数返回一个 Promise 对象，可以通过 `.then()` 方法处理成功的回调，通过 `.catch()` 方法处理失败的回调。

在函数内部，首先创建了一个 XMLHttpRequest 对象，并使用 `xhr.open()` 方法打开了一个异步请求。然后设置了请求头部信息，并监听了 `xhr.onload` 和 `xhr.onerror` 事件。当请求成功返回时，调用 `resolve()` 方法，并传入服务器返回的响应数据；当请求失败时，调用 `reject()` 方法，并传入一个错误对象。

在调用该函数时，可以链式调用 `.then()` 和 `.catch()` 方法来处理成功和失败的回调。

你说得对但是你为什么不用其它的

六.nginx

Nginx (engine x) 是一个高性能的[HTTP](#)和[反向代理](#)web服务器。

优点是：

1.高并发、高性能

高并发 (High Concurrency)

它通常是指，通过设计保证系统能够同时并行处理很多请求。

高性能

是指服务响应时间快，（CPU/处理器/内存）特别是在高并发下响应时间不会急剧增加。

2.可以进行反向代理

反向代理（Reverse Proxy）方式是指以代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器。

反向代理是作用在服务器端的，是一个虚拟ip(VIP)。对于用户的一个请求，会转发到多个后端处理器中的一台来处理该具体请求。

正向代理，是在客户端的。比如vpn是在我们的用户浏览器端设置的

七.什么是websocket

Websocket 使用 ws 或 wss 的统一资源标志符（URI），其中 wss 表示使用了 TLS 的 Websocket。

WebSocket 与 HTTP 和 HTTPS 使用相同的 TCP 端口，可以绕过大多数防火墙的限制。

默认情况下：

- 1) WebSocket 协议使用 **80 端口**；
- 2) 若运行在 TLS 之上时，默认使用 **443 端口**。

WebSocket 使得客户端和服务端之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。在 WebSocket API 中，浏览器和服务器只需要完成一次握手，两者之间就可以创建持久性的连接，并进行双向数据传输。

特点：

- 长连接：建立一个tcp连接就可以实现客户端和服务器的多次通信，而无需在每次通信前重新建立一个tcp连接。
- 全双工通信
- 通信的报文格式简单：遵循了一种基于帧（Frame）的格式，这些帧用于在 WebSocket 连接上进行数据传输

WebSocket 报文的基本结构包括：

- **帧头（Frame Header）**：帧头通常由一个或多个字节组成，用于指示帧的类型、是否有掩码（Masking）、载荷数据的长度等信息。
- **掩码（Masking）**：如果客户端向服务器发送的帧需要进行掩码操作，会在帧头中设置相应的标志位，并包含 4 字节的掩码密钥。掩码用于对载荷数据进行异或操作，以保护数据的安全性。

- **载荷数据（Payload Data）**：载荷数据即实际需要传输的数据，可以是文本、二进制数据或者其他类型的数据。载荷数据的长度可以根据帧头中的信息进行确定。

与HTTP长连接的区别：

HTTP长连接就是在请求头部中Connection默认为Keep-alive参数，Keep-alive的确可以实现长连接，但是这个长连接是有问题的，本质上依然是**客户端主动发起-服务端应答**的模式，是没法做到服务端主动发送通知给客户端的。也就是说，在一个HTTP连接中，可以发送多个Request，接收多个Response。但是一个request只能有一个response。而且**这个response也是被动的，不能主动发起**。相较于没有keep-alive优化的只是不用客户端每次请求应答都要重新建立一个tcp连接了

WebSocket 是类似 TCP 长连接的通讯模式，一旦 WebSocket 连接建立后，**后续数据都以帧序列的形式传输**。在客户端断开 WebSocket 连接或 Server 端断掉连接前，不需要客户端和服务端重新发起连接请求。**在海量并发及客户端与服务器交互负载流量大的情况下**，极大的节省了网络带宽资源的消耗，有明显的性能优势。

为啥常见的还是https？

1. **复杂性**：相对于传统的 HTTP 请求，WebSocket 的实现和维护可能更加复杂。WebSocket 需要在服务器和客户端之间维持长连接，这需要额外的处理和资源。
2. **兼容性问题**：尽管大多数现代浏览器和服务器都支持 WebSocket，但仍存在一些旧版本或者特定环境下不支持 WebSocket 的情况。这可能会导致在某些情况下选择不使用 WebSocket。
3. **性能考虑**：尽管 WebSocket 在某些情况下可以提供更低的延迟和更高的吞吐量，但并不是所有的应用都需要这种性能。**对于一些简单的应用或者低流量的场景，传统的 HTTP 请求可能已经足够满足需求**，而且可能更容易缓存和调试。

4. **安全性考虑：** WebSocket 的长连接特性可能会增加一些安全风险，例如长时间保持连接可能使服务器容易受到拒绝服务（DDoS）攻击。因此，在一些安全性要求较高的场景下，可能会选择避免使用 WebSocket。

什么是DDoS攻击？

DDoS攻击是指利用大量的合法或非法手段，同时向目标服务器发送大量的请求或连接，以使目标服务器过载，**无法正常处理合法用户的请求**。攻击者可能使用各种手段，包括僵尸网络（botnet）、分布式拒绝服务（DDoS）工具等，来发动这种攻击。DDoS攻击可能导致目标服务器崩溃或网络服务中断，给服务器运维者带来严重的影响。

长时间保持连接可能会使服务器容易受到拒绝服务（DDoS）攻击，这主要是因为**在长时间保持连接的情况下，服务器资源（如内存、带宽等）会被占用，导致服务器无法响应其他合法用户的请求**。而DDoS攻击正是利用大量的恶意请求或连接来占用服务器资源，从而使合法用户无法正常访问服务。

八.什么是iframe以及什么是postMessage

什么是iframe

iframe是嵌入式框架, 是html标签, 还是一个内联元素, iframe 元素会创建包含另外一个文档的内联框架（即行内框架）。说白了, iframe用来在页面嵌入其他页面

通常我们使用iframe直接直接在页面嵌套iframe标签指定src就可以了。

```
<iframe src="demo_iframe_sandbox.htm"></iframe>
```

前提: **同城**情况

iframe 元素作为页面的嵌入式框架, 在**同城情况**下可以通过 JavaScript 自由地操作 iframe 和父框架的内容, 包括对 DOM 的访问、修改以及相互之间的通信。这种情况下, 因为同源策略的限制被解除, 所以可以实现更丰富的交互和功能。

然而, 当涉及跨域操作时, 同源策略会限制对 iframe 内部内容的访问。不可以实现对 DOM 的访问, 但是可以**通过 postMessage API**实现通信。如果两个窗口**一级域名相同, 只是二级域名不同**, 那么 `document.domain` 属性均为一级域名, 就可以规避同源政策, 拿到 DOM。

iframe的优缺点

优点:

1. 可以通过iframe嵌套**通用的页面**, 提高代码的**重用率**, 比如页面的头部样式和底部版权信息
2. 重新加载页面时, **不需要重载iframe框架页的内容**, 增加页面重载速度.

缺点:

1. 会产生很多页面, 不容易管理
2. iframe框架的内容无法被搜索引擎捕获, 不利于SEO
3. iframe**兼容性较差**
4. iframe有一定的安全风险
5. iframe会阻塞主页面的Onload事件

`onload` 和 `DOMContentLoaded` (或简称为 `contentLoaded`) 都是用于在页面加载完成后执行 JavaScript 代码的事件。它们之间的主要区别在于触发的时机和适用的对象。

1. onload:

- `onload` 事件在整个页面（包括所有资源如图片、样式表等）加载完成后触发。
- 这意味着当整个文档及其相关资源都加载完毕后，浏览器会触发 `onload` 事件。因此，`onload` 通常用于执行那些依赖于整个页面及其资源完全加载后才能进行的操作，例如操作 DOM 元素、初始化页面内容等。

```
<script>
    window.onload = function() {
        // 页面及其所有资源加载完成后执行的代码
    };
</script>
```

2. DOMContentLoaded:

- `DOMContentLoaded` 事件在 DOM 树构建完成后触发，但不等待样式表、图片和其他资源的加载完成。
- 这意味着当 HTML 文档被完全解析和加载后，就会触发 `DOMContentLoaded` 事件。相对于 `onload`，`DOMContentLoaded` 更早地触发，因此适合用于执行那些不依赖于页面所有资源加载完成的操作，例如修改 DOM 结构、绑定事件处理程序等。

```
<script>
    document.addEventListener('DOMContentLoaded',
function() {
        // DOM 结构构建完成后执行的代码
    });
</script>
```

总的来说，`onload` 适用于需要等待所有资源加载完成后才执行的操作，而 `DOMContentLoaded` 则适用于在 DOM 构建完成后即可执行的操作，这样可以提高页面的响应速度。

有哪些常用的api

```
var iframe = document.getElementById("iframe1");
var iwindow = iframe.contentWindow;
var idoc = iwindow.document;

    console.log("window", iwindow); // 获取iframe的window对象
    console.log("document", idoc);   // 获取iframe的document
对象

    console.log("html", idoc.documentElement); // 获取iframe
的html

    console.log("head", idoc.head);   // 获取head
    console.log("body", idoc.body);   // 获取body
```

`document` 对象和 `window` 对象是 JavaScript 中两个常用的对象，它们在 Web 开发中扮演着不同的角色，但也有一些交集。

1. window 对象：

- `window` 对象代表浏览器中的窗口或标签页，它是 JavaScript 中的全局对象，可以通过 `window` 关键字直接访问。
- `window` 对象包含了浏览器窗口的所有内容，比如浏览器的视口、地址栏、导航栏等。
- 它还包含了浏览器窗口的各种属性和方法，比如 `window.location` 用于获取或设置当前页面的 URL，`window.alert()` 用于显示警告框等。

2. document 对象：

- `document` 对象代表当前 HTML 文档，它是 `window` 对象的一个属性，可以通过 `window.document` 或者简写为 `document` 来访问。
- `document` 对象包含了整个 HTML 文档的内容，包括 HTML 元素、文本内容、CSS 样式等。
- 它提供了访问和操作 HTML 文档内容的方法和属性，比如 `document.getElementById()` 用于通过元素 ID 获取元素对象，`document.quer`

`ySelector()` 用于通过 CSS 选择器获取元素对象等。

区别：

- `window` 对象是全局对象，代表浏览器窗口或标签页，而 `document` 对象是 `window` 对象的一个属性，代表当前 HTML 文档。
- `window` 对象包含了浏览器窗口的所有内容和功能，而 `document` 对象包含了当前 HTML 文档的内容和结构。
- `window` 对象提供了管理浏览器窗口和导航的方法和属性，而 `document` 对象提供了访问和操作 HTML 文档内容的方法和属性。

总的来说，`window` 对象用于管理浏览器窗口和提供浏览器相关的功能，而 `document` 对象用于操作 HTML 文档的内容和结构。

当然`iframe`也可以获取到使用它的父`document`对象的一些信息

- `window.parent` 获取上一级的`window`对象，如果还是`iframe`则是该`iframe`的`window`对象
- `window.top` 获取最顶级容器的`window`对象，即，就是你打开页面的文档

`window.parent` 和 `window.top` 的区别：

`window.parent` 和 `window.top` 都是 JavaScript 中的 `Window` 对象的属性，用于访问嵌套页面之间的关系。它们之间的主要区别在于它们指向的对象不同：

1. **`window.parent`**：这个属性返回当前窗口的父级窗口。如果当前窗口是顶级窗口（没有嵌套），那么 `window.parent` 返回自身。如果当前窗口是嵌套在另一个窗口（例如 `iframe`）中的，则 `window.parent` 返回其父级窗口的 `Window` 对象。

```
// 在 iframe 中，获取父级窗口的 document 对象
var parentDocument = window.parent.document;
```

2. **window.top**: 这个属性返回当前窗口的顶级窗口，即整个页面的最顶级容器的 Window 对象。无论当前窗口是否嵌套在其他窗口中，**window.top** 总是指向整个页面的顶级窗口。

```
// 在任何窗口中，获取顶级窗口的 document 对象
var topDocument = window.top.document;
```

总之，**window.parent** 和 **window.top** 都用于访问嵌套页面中不同级别的窗口对象，但是 **window.parent** 返回的是直接父级窗口，而 **window.top** 返回的是整个页面的顶级窗口。

什么是postMessage

postMessage() 是 HTML5 提供的一种跨窗口通信的 API。它允许在不同的窗口或 iframe 之间安全地传递信息，即使这些窗口来自不同的源（跨域）。**postMessage()** 允许发送一个消息到目标窗口，同时可以指定接收消息的窗口的源（origin），从而确保消息只会被预期的接收方处理。

基本语法如下：

```
otherWindow.postMessage(message, targetOrigin, [transfer]);
```

- **otherWindow**：目标窗口的引用，可以是 iframe 的 **contentWindow**，或者是通过 **window.open()** 打开的窗口的引用。
- **message**：要发送的消息，可以是字符串、数字、对象等。
- **targetOrigin**：接收消息窗口的源（origin），可以是具体的源（如 "<http://example.com>"），也可以是通配符 "*"，表示可以接收来自任意源的消息。
- **transfer** (可选)：一个传输对象数组，其中包含要在消息中传输的 **Transferable** 对象。这些对象会被转移给接收方窗口的控制。

接收消息的窗口可以监听 `message` 事件来处理收到的消息，事件对象中包含了发送消息的窗口的引用、消息内容等信息。

```
window.addEventListener('message', function(event) {  
    // event.source 为发送消息的窗口的引用  
    // event.origin 为发送消息窗口的源  
    // 验证消息来源  
    if (event.origin === 'https://example.com') {  
        // 处理接收到的消息  
        console.log('Message from parent page:', event.data);  
    }  
});  
// event.data 为接收到的消息内容  
// 进行相应处理  
});
```

通过 `postMessage()`，页面可以实现跨域安全通信，例如在父窗口和嵌套的 `iframe` 之间，或者在弹出的窗口之间传递数据。这种跨域通信方式能够有效地应对同源策略的限制，提供了一种安全且灵活的方式进行跨窗口通信。并且也可以实现 `localStorage` 只能在同源的页面中共享的难题，可以将 `event.data` 存入 `localStorage` 中。

九.什么是webWorker

1.概念以及分类

Web Worker 是 HTML5 提供的一种机制，用于在后台运行脚本，从而使得在主线程中运行的 JavaScript 代码不会被阻塞。Web Worker 允许开发者创建一个独立的后台线程，用于执行一些耗时的任务，比如大量计算、网络请求等，而不会影响到主线程的执行和页面的响应性能。

Web Worker 分为两种类型：Dedicated Worker（专用 Worker）和 Shared Worker（共享 Worker）。

- **Dedicated Worker**：每个 Dedicated Worker 都与创建它的脚本有一对一的关系，它只能被创建它的脚本访问，用于执行与其相关的任务。
- **Shared Worker**：Shared Worker 可以被多个脚本共享，它们可以在不同的浏览器窗口或标签页中共享数据和通信，用于多个脚本之间的协作和数据共享。

Web Worker 的**主要用途**包括：

- 执行**大量计算或数据处理任务**，以提高页面的响应性能。
- 在后台进行网络请求、文件读写等 **I/O** 操作，以避免阻塞主线程。
- 实现复杂的数据处理、算法运算等功能。

简单代码示例：

```
// 在主线程创建并启动一个 Web Worker
const worker = new Worker('worker.js');
// 默认为dedicatedWorker

// 主线程向 Web Worker 发送消息，并监听消息回调
worker.postMessage({ type: 'start', data: someData });
worker.onmessage = function(event) {
    console.log('Web Worker 返回结果:', event.data);
};
```

```
// worker.js 文件中的代码，用于在后台执行耗时任务
self.onmessage = function(event) {
    const data = event.data;
    // 执行耗时任务...
    const result = processData(data);
    // 将处理结果发送回主线程
    self.postMessage(result);
};

function processData(data) {
    // 处理数据的逻辑...
    return processedData;
}
```

2.原理

JS 多线程，是有独立于主线程的 JS 运行环境，Worker 线程有独立的内存空间，Message Queue，Event Loop，Call Stack 等，线程间通过 postMessage 通信。

如果在 new Worker(test.js) 中 test.js 文件存在且可访问，浏览器会生成一个线程来异步下载文件。当下载完成的时候，Web Worker 会创建 **操作系统级别的线程**。主线程和这个创建的线程**并行运行**js 代码，需要通信时通过 postMessage 进行通信即可。

Worker 线程运行在一个受限的环境中，与主线程相比，它没有对 DOM 的访问权限，也无法直接操作页面的 UI。这种限制是为了确保 Worker 线程的安全性，防止其影响到页面的正常运行。

3.缺点

兼容性比较差

这是浏览器的js实现多线程的方案，那么node怎么实现多线程的？

在 Node.js 中，与浏览器不同，JavaScript 本身仍然是单线程执行的。这是因为 Node.js 的设计目标之一是简化并发操作，通过单线程和事件驱动模型来处理 I/O 操作。然而，Node.js 提供了一些方式来利用多核 CPU，实现类似多线程的效果：

1. Child Processes: Node.js 提供了 `child_process` 模块，允许创建子进程来执行耗时的操作。每个子进程都是一个独立的进程，可以并行执行任务，从而利用多核 CPU。可以使用 `fork()` 方法来创建子进程，并通过进程间通信来传递消息。

```
javascriptCopy Codeconst { fork } =
require('child_process');

// 创建子进程
const child = fork('child.js');

// 主进程向子进程发送消息
child.send({ message: 'Hello from parent process!'
});

// 子进程接收消息并处理
process.on('message', (message) => {
  console.log('Message from parent process:',
message);
});
```

2. Worker Threads: Node.js 10.5.0 版本引入了实验性的 `worker_threads` 模块，允许创建真正的线程来执行 JavaScript 代码。Worker Threads 提供了类似于 Web Workers 的功能，可以在多个线程中并行执行任务，从而利用多核 CPU。

```
javascriptCopy Codeconst { Worker, isMainThread,
parentPort } = require('worker_threads');

if (isMainThread) {
    // 在主线程中创建 Worker
    const worker = new Worker('worker.js');

    // 主线程向 Worker 发送消息
    worker.postMessage('Hello from main thread!');

    // Worker 接收消息并处理
    worker.on('message', (message) => {
        console.log('Message from worker:',
message);
    });
} else {
    // Worker 线程接收消息并处理
    parentPort.on('message', (message) => {
        console.log('Message from main thread:',
message);

        // Worker 向主线程发送消息
        parentPort.postMessage('Hello from worker
thread!');
    });
}
```

十.JWT

JWT 是一种特定格式的 token，具有固定的结构和一些特定的特性，适用于需要在不同系统之间传递和验证身份信息的场景。

JWT 由三部分组成，用点号 (.) 分隔开来：

- 1. Header（头部）：** 包含了 Token 的元数据和签名算法，通常包含两个部分：token 的类型（即 JWT）和加密所使用的算法，例如签名（HMAC）或公钥/私钥（RSA 或 ECDSA）
- 2. Payload（负载）：** 包含了要传输的信息，比如用户的 ID、角色等。也可以包含自定义的键值对信息，但是敏感信息应该避免放在 Payload 中，因为 Payload 默认是 Base64 编码的，不是加密的。
- 3. Signature（签名）：** Signature 部分是对前两部分的签名，防止数据篡改。

首先，需要指定一个密钥（secret）。这个密钥只有服务器才知道，不能泄露给用户。然后，使用 Header 里面指定的签名算法（默认是 HMAC SHA256），按照下面的公式产生签名。

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secret)
```

算出签名以后，把 Header、Payload、Signature 三个部分拼成一个字符串，每个部分之间用"点" (.) 分隔，就可以返回给用户。

服务器就不保存任何 session 数据了，也就是说，服务器变成无状态了，从而比较容易实现扩展

服务器在拿到客户端传来的token以后：

- **解析 Token：**服务器首先需要解析 JWT，以提取其中的有效载荷信息。JWT 由三部分组成：头部 (Header)、载荷 (Payload) 和签名 (Signature)。服务器需要解码 Base64 编码的头部和载荷，以获取其中的信息。
- **验证签名：**JWT 的签名部分用于验证 token 的完整性和真实性。服务器需要使用密钥解密，然后来验证签名的有效性，**以确保 token 没有被篡改。**
- **检查有效期：**JWT 通常包含有关 token 有效期的信息，如过期时间 (exp)。服务器需要检查当前时间是否在 token 指定的有效期内，以确保 token 没有过期。
- **处理权限和身份验证：**一旦服务器验证了 token 的**有效性和真实性**，就可以根据其中的载荷信息来处理权限和身份验证。载荷通常包含有关用户身份、角色、权限等信息，服务器可以根据这些信息来判断用户是否有权执行请求的操作。
- **缓存或存储信息：**在处理完 JWT 后，服务器可能会根据需要**缓存或存储**其中的信息，以便后续的请求可以**更快地验证用户身份和权限**，或者用于其他需要的业务逻辑。
- **响应处理结果：**最后，服务器根据验证的结果和业务逻辑，生成响应并返回给客户端。如果 token 有效并且用户有权执行请求的操作，则服务器可以继续处理请求；否则，服务器可能会返回相应的错误信息或状态码。

优点

- JWT使用签名（HMAC）或公钥/私钥（RSA 或 ECDSA）进行加密，确保令牌的完整性和可信度。通过对令牌进行签名或加密，可以防止令牌**被篡改或伪造**。
- JWT 适用于分布式系统和无状态服务架构，每个服务都可以独立验证 JWT 的有效性，而无需依赖中心化的会话存储，**性能很好**

缺点

- 默认情况下，JWT 中的信息是以 Base64 编码的形式进行传输，虽然可以使用加密算法对整个令牌进行签名，但令牌中的信息本身并不加密。这意味着任何人都可以解码 JWT 并读取其中的信息，因此**不适合传输敏感信息**。
- **增加 CSRF（Cross-Site Request Forgery，跨站请求伪造）攻击的风险**。这是因为 JWT 通常存储在客户端的本地存储或会话存储中，而不像 Cookie 那样可以设置 HttpOnly 属性来防止 JavaScript 访问。
- 一旦签发了 JWT，在到期之前就会始终有效，可能会导致令牌被**滥用**，恶意用户可能利用令牌来执行一些恶意操作

解决的办法：

- **使用 HTTPS 协议**来加密数据传输，防止敏感信息在传输过程中被窃取或篡改。通过 HTTPS，可以确保登录过程中的用户名、密码和令牌等信息在传输过程中的安全性。
- 对于访问令牌和刷新令牌，**应该及时更新并且合理设置有效期**，以减少令牌被盗用的风险。同时，在令牌过期后，及时重新进行身份验证并获取新的令牌。
- **对于敏感操作，例如修改用户密码或执行付款等，应该要求用户输入密码或进行其他形式的额外验证，以确保操作的合法性。**

十一.cookie

1.什么是cookie

如果这个网站我们曾经登录过，那么当我们再次打开网站时，发现就不需要再次登录了，而是直接进入了首页。例如bilibili，csdn等网站。

这是怎么做到的呢？其实就是浏览器保存了我们的cookie，里面记录了一些信息，当然，这些cookie是服务器创建后返回给浏览器的。浏览器只进行了保存。

2.cookie的存储形式

一般情况下，cookie是以键值对进行表示的(key-value)，例如name=jack，这个就表示cookie的名字是name，cookie携带的值是jack。

以下是cookie中常用属性的解释。

- Name：这个是cookie的名字
- Value：这个是cookie的值
- Path：这个定义了Web站点上可以访问该Cookie的目录
- Expires：这个值表示cookie的过期时间，也就是有效值，cookie在这个值之前都有效。
- Size：这个表示cookie的大小

在 HTTP 请求头部中，Cookie 通过 `Cookie` 字段来表示。`Cookie` 字段包含了多个键值对，每个键值对代表一个 Cookie。每个键值对之间使用分号和空格进行分隔。

在HTTP响应头部，服务器可以通过 `Set-Cookie` 字段来设置 Cookie。其中，`name=value` 是 Cookie 的名称和值，多个键值对也用分号隔开。可以设置的键值对：

- **Path（路径）**：指定 Cookie 的作用路径。它限制了哪些路径下的页面可以访问该 Cookie。默认情况下，Cookie 的作用路径是设置它的页面所在的路径。
- **Expires（过期时间）**：指定 Cookie 的过期时间（绝对时间，一个具体的日期和时间）。它告诉客户端在何时应该删除该 Cookie。**如果未设置 Expires 或 Max-Age，Cookie 将成为会话 Cookie，仅在当前会话中有效。**
- **Max-Age（最大生存时间）**：指定 Cookie 的最大生存时间，以秒为单位。与 Expires 不同，Max-Age 指定了**从创建 Cookie 开始经过的时间长度，而不是具体的日期和时间。**

3.缺点

- Cookie存储于浏览器，可以被篡改，服务器接收后必须先验证数据的合法性。
- 浏览器限制Cookie的数量和大小（通常限制为50个，每个不超过4KB），对于复杂的存储需求来说是不够用的。
- 跨域没法访问了

4.优点

- 实现简单
- 兼容性好

5.请求头里有关cookie的设置

1.Secure属性指定浏览器只有在加密协议 HTTPS 下，才能将这个 Cookie 发送到服务器。另一方面，如果当前协议是 HTTP，浏览器会自动忽略服务器发来的Secure属性。该属性只是一个开关，不需要指定值。如果通信是 HTTPS 协议，该开关自动打开。

2.HttpOnly属性指定该 Cookie 无法通过 JavaScript 脚本拿到，主要是 Document.cookie属性、XMLHttpRequest对象和 Request API 都拿不到该属性。这样就防止了该 Cookie 被脚本读到，只有浏览器发出 HTTP 请求时，才会带上该 Cookie。

3.设置过期时间Expires / Max-Age

十二.session

1.是什么

服务器要知道当前发请求给自己的是谁。为了做这种区分，服务器就要给每个客户端分配不同的“身份标识”，然后客户端每次向服务器发请求的时候，都带上这个“身份标识”，服务器就知道这个请求来自于谁了。

在 Web 开发中，会话通常是通过使用会话标识符（Session ID）来实现的。当用户第一次访问网站时，服务器会为其生成一个唯一的会话 ID，并将其发送给客户端。客户端在后续的请求中将会话 ID 发送给服务器，服务器使用该 ID 来识别用户的会话，并**存储相应的会话数据**（比如账号信息，用户设置偏好，购物车信息等等）。

2.怎么实现

session 可以**基于 cookie 实现**，客户端访问服务器，如果没有携带 SESSIONID，那么服务器就会创建一个 session，并且把这个 session 的 SESSIONID 返回给浏览器，返回的方式就是通过 SESSIONID 来实现的。

下次在发送请求时请求头部中可以用 cookie 字段带上这个 SESSIONID 发送给服务器

3.优点：

- 查询速度快，因为是个会话，相当于是在内存中操作。
- 结合 cookie 后很容易实现鉴权。
- 安全，因为存储在服务端。

4.缺点

耗费服务器资源，不适宜分布式服务器

存储在 cookie 中，那 cookie 的缺点你都有

为啥？（重要）

分布式系统是由多个独立的计算机节点组成的系统，这些节点通过网络进行通信和协作，共同完成一个或多个共同的任务。

在分布式系统中，对于用户的每个请求可能会被不同的服务器处理，这就引入了一些挑战，特别是在处理用户会话（session）时：

1. **状态管理**：在传统的单体应用中，可以将用户会话信息直接存储在服务器的内存中。但在分布式系统中，由于用户请求可能会被**不同的服务器处理**，因此**无法简单地依赖服务器内存来管理会话状态**。这就需要一种能够在**不同服务器之间共享会话信息**的机制。
2. **负载均衡**：分布式系统通常会使用**负载均衡器**来分发用户请求到不同的服务器节点，以提高系统的性能和可用性。但这样做可能会导致**用户的不同请求被发送到不同的服务器上**，从而造成用户会话信息的不一致性。

因此，在分布式系统中，需要对会话管理进行额外的处理，以确保用户在不同服务器之间的会话状态是一致的。这可能涉及到将会话信息存储在共享的数据库、缓存或其他持久化存储中，并通过某种方式来标识和检索用户的会话信息，以确保不同服务器之间可以共享和同步会话状态。

十三. 到底该如何选取保持用户的登录状态？（说高端点就是鉴权和认证）

保持用户的登录状态，保存的究竟是什么呢？

其实就是一堆用户的信息

1.cookie只是一个基石

它并不是一个独立的手段，当然你要说非得把服务器拿来的用户状态直接放到cookie里，缺点太多：

- 无法跨域，对跨域需要处理

- 空间太小，保持用户的登陆状态需要的数据过多就没办法胜任
- 可以随意篡改

2.session可以吗？ 如果不可以，为什么？

也可以，一般是依靠cookie进行存储，把session ID放到cookie里，但是性能差，不适宜分布式系统

3.token可以吗？ 如果不可以，为什么？

可以，最常用，因为JWT有很多**优点**：

- JWT使用签名（HMAC）或公钥/私钥（RSA 或 ECDSA）进行加密，确保令牌的完整性和可信度。通过对令牌进行签名或加密，可以防止令牌**被篡改或伪造**。
- JWT 适用于分布式系统和无状态服务架构，每个服务都可以独立验证JWT 的有效性，而无需依赖中心化的会话存储，**性能很好**

但是也有**缺点**：

- 默认情况下，JWT 中的信息是以 Base64 编码的形式进行传输，虽然可以使用加密算法对整个令牌进行签名，但令牌中的信息本身并不加密。这意味着任何人都可以解码 JWT 并读取其中的信息，因此**不适合传输敏感信息**。
- **增加 CSRF（Cross-Site Request Forgery，跨站请求伪造）攻击的风险**。这是因为 JWT 通常存储在客户端的cookie或会话存储中，而不像 Cookie 那样可以设置 HttpOnly 属性来防止 JavaScript 访问。
- 一旦签发了 JWT，在到期之前就会始终有效，可能会导致令牌被**滥用**，恶意用户可能利用令牌来执行一些恶意操作

解决的办法：

- **使用 HTTPS 协议**来加密数据传输，防止敏感信息在传输过程中被窃取或篡改。通过 HTTPS，可以确保登录过程中的用户名、密码和令牌等信息在传输过程中的安全性。

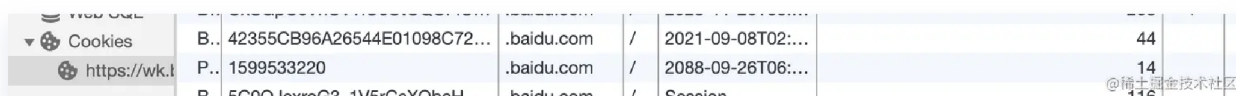
- 对于访问令牌和刷新令牌，**应该及时更新并且合理设置有效期**，以减少令牌被盗用的风险。同时，在令牌过期后，及时重新进行身份验证并获取新的令牌。
- 对于敏感操作，例如修改用户密码或执行付款等，**应该要求用户输入密码或进行其他形式的额外验证**，以确保操作的合法性。

补充：什么是单点登录

英文全称 Single Sign On，简称 SSO。它的定义是：在多个应用系统中，用户只需要登录一次，即可访问所有相互信任的应用系统，就像健康保一样，为你的身份做担保。简单来说，用户只需登录一次，就可以访问多个系统，而无需为每个系统单独提供凭据。

“虚假”的单点登录（主域名相同）

简单的，如果业务系统都在同一主域名下，比如 `wenku.baidu.com` `tieba.baidu.com`，就好办了。可以直接把 cookie domain 设置为主域名 `baidu.com`，百度也就是这么干的。



▼ Cookies	B..	42355CB96A26544E01098C72...	.baidu.com	/	2021-09-08T02:...	44
https://wk.t	P..	1599533220	.baidu.com	/	2088-09-26T06:...	14
	R	500QJeynG3_1V5rCeXQhcH	baidu.com	/	Session	116

@稀土掘金技术社区

“真实”的单点登录（主域名不同）

1.CAS（中央认证服务，核心是CAS Server）方案：

- 用户访问 APP1，需要登录时被重定向到 **CAS Server**。这意味着用户首次访问需要进行身份认证。
- 在 CAS Server 进行账号密码认证，验证成功后，CAS Server **会创建一个会话（session）**，并生成一个 **sessionId** 返回给 SSO Client。
- SSO Client 将 sessionId 写入**当前域的 Cookie**，并根据 TGT（Ticket Granting Ticket）签发一个 ST（Service Ticket）传给 APP1。

- APP1 携带 ST 向 CAS Server 请求校验。CAS Server 校验成功后，确认用户已经登录，并返回登录态给 APP1 服务端。
- APP1 服务端将登录态写入 session 并生成 sessionId 返回给 APP1 Client。
- 用户首次访问 APP2 时，重定向到SSO，发现已经在 SSO 中登录了。SSO 生成一个 ST，并传给 APP2。
- APP2 携带 ST 向 CAS Server 请求校验。CAS Server 校验成功后，确认用户已经登录，并返回登录态给 APP2 服务端。
- APP2 服务端将登录态写入 session 并生成 sessionId 返回给 APP2 Client。

十四.跨域解决策略

为什么要跨域？什么又是同源策略？

所谓"同源"指的是"三个相同"。

- 协议相同
- 域名相同
- 端口相同

为什么要有同源？

- **防止跨站点请求伪造（CSRF）攻击**：同源政策可以防止恶意网站利用用户的身份进行伪造请求，从而执行未经授权的操作，比如转账、更改密码等。
- **保护用户隐私**：同源政策限制了一个网站对其他网站的信息访问，防止恶意站点窃取用户在其他站点的敏感信息，如Cookie、本地存储等。
- **防止跨站脚本（XSS）攻击**：同源政策阻止恶意脚本在一个域内执行，从而减少XSS攻击的可能性。

随着互联网的发展，"同源政策"越来越严格。目前，如果非同源，共有三种行为受到限制。

- (1) Cookie、LocalStorage 和 IndexDB 无法读取。
- (2) DOM 无法获得。
- (3) AJAX 请求不能发送。

如何获取不同源的cookie

允许不同源的cookie之间可以互相访问的行为一般不会实现，因为安全风险太高。但是我们希望两个网页一级域名相同，只是二级域名不同，此时实现两个cookie共享则是一种比较合理的请求。只要设置只要设置相同的 `document.domain`，两个网页就可以共享Cookie。

另外，服务器也可以在设置Cookie的时候，指定Cookie的所属域名为一级域名，比如 `.example.com`。

```
Set-Cookie: key=value; domain=.example.com; path=/
```

这样的话，二级域名和三级域名不用做任何设置，都可以读取这个Cookie。

CORS（跨源资源共享）：

CORS 是一种机制，允许Web服务器声明哪些源站可以通过浏览器访问哪些资源。在服务端设置响应头中的 `Access-Control-Allow-Origin` 字段来允许特定源站的访问。例如，可以设置为 `Access-Control-Allow-Origin: *` 表示允许所有源站的访问，或者设置为具体的源站地址。

原理：

实现CORS通信的关键是服务器。只要服务器实现了CORS接口，就可以跨源通信

对于前端开发者来说，无需进行配置，而是浏览器一旦发现AJAX请求跨源，就会自动添加一些附加的头信息。而浏览器会将请求分为两种请求，简单请求（simple request）和非简单请求（not-so-simple request），对应两种不同的请求，浏览器会有不同的处理方案。

简单请求：

只要同时满足以下两大条件，就属于简单请求。

(1) 请求方法是以下三种方法之一：

- HEAD
- GET
- POST

(2) HTTP的头信息不超出以下几种字段：

- Accept
- Accept-Language
- Content-Language
- Last-Event-ID
- Content-Type : 只限于三个值 `application/x-www-form-urlencoded`、`multipart/form-data`、`text/plain`

凡是不同时满足上面两个条件，就属于非简单请求。

对于简单请求，浏览器直接发出CORS请求。具体来说，就是在头信息之中，增加一个 `Origin` 字段。`Origin` 字段用来说明，本次请求来自哪个源（协议 + 域名 + 端口）。服务器根据这个值，决定是否同意这次请求。如果 `Origin` 指定的源，不在许可范围内，服务器会返回一个正常的HTTP回应。浏览器发现，这个回应的头信息没有包含 `Access-Control-Allow-Or`

igin 字段，就知道出错了，从而抛出一个错误，被 XMLHttpRequest 的 onerror 回调函数捕获。注意，这种错误无法通过状态码识别，因为HTTP回应的状态码有可能是200。

如果 Origin 指定的域名在许可范围内，服务器返回的响应，会多出几个头信息字段。

```
Access-Control-Allow-Origin: http://api.bob.com
```

```
Access-Control-Allow-Credentials: true
```

```
Access-Control-Expose-Headers: FooBar
```

(1) Access-Control-Allow-Origin

该字段是必须的。它的值要么是请求时 Origin 字段的值，要么是一个 *

(2) Access-Control-Allow-Credentials

该字段可选。它的值是一个布尔值，表示是否允许发送Cookie。默认情况下，Cookie不包括在CORS请求之中。设为 true，即表示服务器明确许可，Cookie可以包含在请求中，一起发给服务器。这个值也只能设为 true，如果服务器不要浏览器发送Cookie，删除该字段即可。

(3) Access-Control-Expose-Headers

该字段可选。CORS请求时，XMLHttpRequest 对象的 getResponseHeader() 方法只能拿到6个基本字段：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma。如果想拿到其他字段，就必须在 Access-Control-Expose-Headers 里面指定。上面的例子指定，getResponseHeader('FooBar') 可以返回 FooBar 字段的值。

CORS请求默认不发送Cookie。如果要把Cookie发到服务器，一方面要服务器同意，指定 `Access-Control-Allow-Credentials` 字段。

```
Access-Control-Allow-Credentials: true
```

另一方面，开发者必须在AJAX请求中打开 `withCredentials` 属性。

```
var xhr = new XMLHttpRequest();  
xhr.withCredentials = true;
```

否则，即使服务器同意发送Cookie，浏览器也不会发送。

需要注意的是，如果要发送Cookie，`Access-Control-Allow-Origin` 就不能设为星号，必须指定明确的、与请求网页一致的域名。同时，Cookie依然遵循同源政策，只有用服务器域名设置的Cookie才会上传，其他域名的Cookie并不会上传

非简单请求：

非简单请求是那种对服务器有特殊要求的请求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。

非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为"预检"请求（`preflight`）。

浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些HTTP动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的 `XMLHttpRequest` 请求，否则就报错。

"预检"请求用的请求方法是 `OPTIONS`，表示这个请求是用来询问的。头信息里面，关键字段是 `Origin`，表示请求来自哪个源。

除了 `Origin` 字段，"预检"请求的头信息包括两个特殊字段。

(1) `Access-Control-Request-Method`

该字段是必须的，用来列出浏览器的CORS请求会用到哪些HTTP方法

(2) `Access-Control-Request-Headers`

该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段

服务器收到"预检"请求以后，检查了 `Origin`、`Access-Control-Request-Method` 和 `Access-Control-Request-Headers` 字段以后，**确认允许跨源请求，就可以做出回应。**

如果服务器否定了"预检"请求，会返回一个正常的HTTP回应，但是没有任何CORS相关的头信息字段。这时，浏览器就会认定，服务器不同意预检请求，因此触发一个错误，被 `XMLHttpRequest` 对象的 `onerror` 回调函数捕获。控制台会打印出如下的报错信息。

一旦服务器通过了"预检"请求，以后每次浏览器正常的CORS请求，就都跟简单请求一样，会有一个 `Origin` 头信息字段。服务器的回应，也都会有一个 `Access-Control-Allow-Origin` 头信息字段。

与JSONP的比较

CORS与JSONP的使用目的相同，但是比JSONP更强大。

JSONP只是相当于一个和GET请求，CORS支持所有类型的HTTP请求。JSONP的优势在于支持老式浏览器，以及可以向不支持CORS的服务器请求数据。

JSONP

JSON with Padding，JSONP是服务器与客户端跨源通信的常用方法。最大特点就是简单适用，老式浏览器全部支持，服务器改造非常小。

它的基本思想是，网页通过添加一个 `<script>` 元素，向服务器请求JSON数据，这种做法不受同源政策限制；服务器收到请求后，将数据放在一个指定名字的回调函数里传回来。

```
function addScriptTag(src) {
  var script = document.createElement('script');
  script.setAttribute("type", "text/javascript");
  script.src = src;
  document.body.appendChild(script);
}

window.onload = function () {
  addScriptTag('http://example.com/ip?callback=foo');
}

function foo(data) {
  console.log('Your public IP address is: ' + data.ip);
};
```

上面代码通过动态添加 `<script>` 元素，向服务器 `example.com` 发出请求。注意，该请求的查询字符串有一个 `callback` 参数，用来指定回调函数的名字，这对于JSONP是必需的。

服务器收到这个请求以后，将数据放在回调函数的参数位置返回。

```
aCausalName({  
  "ip": "8.8.8.8"  
});
```

由于 `<script>` 元素请求的脚本，直接作为代码运行。这时，只要浏览器定义了 `foo` 函数，该函数就会立即调用。作为参数的JSON数据被视为JavaScript对象，而不是字符串，因此避免了使用 `JSON.parse` 的步骤。

WebSocket

WebSocket是一种通信协议，使用 `ws://`（非加密）和 `wss://`（加密）作为协议前缀。该协议不实行同源政策，只要服务器支持，就可以通过它进行跨源通信。

下面是一个例子，浏览器发出的WebSocket请求的头信息（摘自[维基百科](#)）。

```
GET /chat HTTP/1.1  
Host: server.example.com  
Upgrade: websocket  
Connection: Upgrade  
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==  
Sec-WebSocket-Protocol: chat, superchat  
Sec-WebSocket-Version: 13  
Origin: http://example.com
```

上面代码中，有一个字段是 `Origin`，表示该请求的请求源（origin），即发自哪个域名。

正是因为有了 `Origin` 这个字段，所以WebSocket才没有实行同源政策。因为服务器可以根据这个字段，判断是否许可本次通信。如果该域名在白名单内，服务器就会做出如下回应。

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm50PpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

十五.xss攻击和CSRF攻击

1.跨站脚本攻击（XSS）：

- XSS攻击是指黑客通过在网页上注入恶意的脚本代码，然后让用户的浏览器执行这些恶意代码的过程。这些恶意脚本可以窃取用户的cookie信息、会话信息，甚至篡改网页内容，导致用户受到欺骗或者遭受损失。
- XSS攻击可以分为存储型XSS、反射型XSS和DOM-based XSS。存储型XSS是指恶意代码被存储在服务器上，然后被多个用户访问到；反射型XSS是指恶意代码通过URL参数等方式被用户访问到；DOM-based XSS是指恶意代码通过DOM操作被执行。

2.怎么防范

- 输入过滤和验证：

- 对用户输入进行严格的过滤和验证，确保只接受合法的输入。这包括对用户输入的内容进行转义，以防止恶意代码被执行。常见的转义包括HTML转义、JavaScript转义等。
- **内容安全策略（CSP）：**
 - 使用内容安全策略（CSP）来限制网页中允许加载的资源 and 执行的脚本，可以有效地防范XSS攻击。CSP通过白名单机制来控制网页的加载行为，可以限制只允许加载来自指定域名的资源，禁止内联脚本执行等。
- **设置HTTP Only标志：**
 - 在设置Cookie时，使用HTTP Only标志来禁止JavaScript访问该Cookie，这样可以防止恶意脚本窃取用户的Cookie信息，减少XSS攻击的威胁。

3.什么是CSRF攻击

- CSRF攻击是指黑客通过伪造合法用户的请求，然后以合法用户的身份发送这些请求到目标网站，从而执行某些操作或者获取用户的敏感信息。与XSS攻击不同，CSRF攻击不需要注入恶意脚本，而是利用用户已经登录的身份来进行攻击。
- 攻击者通常会通过欺骗用户点击恶意链接或者访问恶意网站的方式触发CSRF攻击。一旦用户点击了恶意链接或者访问了恶意网站，攻击者就可以利用用户当前的登录状态发送伪造的请求。

4.怎么防范CSRF攻击

- **使用CSRF令牌：**
 - 在处理用户提交的敏感操作（如修改用户信息、转账等）时，生成一个随机的CSRF令牌，并将该令牌嵌入到表单中或者作为请求参数发送到服务器端。服务器端在接收到请求时验证该令牌的合法性，如果令牌不匹配则拒绝该请求。这样可以防止攻击者伪造请求。
- **同源检测：**

- 在服务器端对请求的来源进行检测，只允许来自合法域名的请求通过。可以通过检查请求的Referer头部来验证请求的来源是否合法，但要注意Referer头部可能会被一些浏览器禁用或者篡改，因此不应该仅依赖Referer头部来防范CSRF攻击。

- **使用验证码：**

- 对于敏感操作，可以要求用户输入验证码来确认身份，以防止攻击者利用CSRF攻击执行恶意操作。

- **限制敏感操作的请求方法：**

- 将敏感操作限制为使用POST请求，因为CSRF攻击通常利用的是GET请求的特性，GET请求可以通过图片加载、链接点击等方式触发，而POST请求通常需要用户主动提交表单才能发送。