面试题

1、浏览器渲染原理

1. HTML被HTML解析器解析成`DOM Tree`
2. CSS则被CSS解析器解析成`CSSOM Tree`
3. `DOM Tree`和`CSSOM Tree`解析完成后，被附加到一起，形成渲染树（`Render Tree`）
4. 布局，根据渲染树计算每个节点的几何信息生成`布局树`（`Layout Tree`）
5. 对布局树进行分层，并生成`分层树`（`Layer Tree`）
6. 为每个图层生成绘制列表
7. 渲染绘制(Paint)。根据计算好的绘制列表信息绘制整个页面，并将其提交到`合成线程`
8. `合成线程`将图层分成图块，并在光栅化线程池中将图块转换成位图，发送绘制图块命令`DrawQuad`给浏览器进程
9. 浏览器进程根据 DrawQuad 消息生成页面，并显示到显示器上

2、什么是重排

定义： 当通过JS或css改变了元素的宽度、高度等，修改了元素的几何位置属性，那么浏览器会触发重新布局，解析之后的一系列子阶段，这个过程就叫`重排`。无疑，**重排需要更新完整的渲染流水线，所以开销也是最大的。**

触发时机和影响范围： DOM节点信息更改,触发重排时，这个DOM更改程度会决定周边DOM更改范围。

全局范围： 就是从根节点html开始对整个渲染树进行重新布局，例如当我们改变了窗口尺寸或方向或者是修改了根元素的尺寸或者字体大小等。

局部范围： 对渲染树的某部分或某一个渲染对象进行重新布局。

3、重绘

定义： 如果修改了元素的背景颜色，并没有引起几何位置的变换，所以就直接进入了绘制阶段，然后执行之后的一系列子阶段，这个过程就叫`重绘`。 **相较于重排操作，重绘省去了布局和分层阶段，所以执行效率会比重排操作要高一些。**

触发时机和影响范围： 每一次的dom更改或者css几何属性更改，都会引起一次浏览器的重排/重绘过程，而如果是css的非几何属性更改，则只会引起重绘过程。

4、合成

定义： 合成是一种将页面的各个部分分离成层（Layer Tree），分别将它们栅格化，然后在称为“合成线程”的中组合为页面的技术。

触发时机和影响范围： 在GUI渲染线程后执行，将GUI渲染线程生成的绘制列表转换为位图,然后发送绘制图块命令 DrawQuad 给浏览器进程，浏览器进程根据 DrawQuad 消息生成页面，将页面显示到显示器上

优点： 我们使用了 CSS 的`transform`来实现动画效果，避开了`重排`和`重绘`阶段，直接在非主线程上执行合成动画操作。这样的效率是最高的，因为是在非主线程上合成，并没有占用主线程的资源，另外也避开了布局和绘制两个子阶段，所以相对于重绘和重排，合成能大大提升绘制效率。

5、常见的触发重排、重绘的属性和方法

引发重排的操作

- 页面首次渲染。
- 浏览器窗口大小发生改变——`resize`事件发生时。
- 元素尺寸或位置发生改变——定位、边距、填充、边框、宽度和高度。
- 元素内容变化（文字数量或图片大小等等）。
- 元素字体大小变化。
- 添加或者删除可见的DOM元素。
- 激活CSS伪类（例如：:hover）。
- 设置style属性
- 查询某些属性或调用某些方法。

引起重排属性和方法

```
width、 display、 clientWidth、 offsetWidth、 scrollWidth、 scrollIntoView()、
getBoundingClientRect()、 height、 border、 clientHeight、 offsetHeight、
scrollHeight、 scrollTo()、 scrollIntoViewIfNeeded()、 margin、 position、
clientTop、 offsetTop、 scrollTop、 getComputedStyle()、 padding、 overflow、
clientLeft、 offsetLeft、 scrollLeft
```

引起重绘的属性

```
color、 text-decoration、 outline-color、 outline-width、
border-style、 background-image、 outline、 box-shadow、
visibility、 background-position、 outline-style、
background-size、 background、 background-repeat、 border-radius
```

6、优化策略

1.减少DOM操作

最小化DOM访问次数，尽量缓存访问DOM的样式信息，避免过度触发重排。

如果在一个局部方法中需要多次访问同一个dom，可以在第一次获取元素时用变量保存下来，减少遍历时间。

用事件委托来减少事件处理器的数量。

用querySelectorAll()替代getElementByXX()。

querySelectorAll()：获取静态集合，通过函数获取元素之后，元素之后的改变并不会影响之前获取后存储到的变量。也就是获取到元素之后就和html中的这个元素没有关系了

getElementByXX()：获取动态集合，通过函数获取元素之后，元素之后的改变还是会动态添加到已经获取的这个元素中。换句话说，通过这个方法获取到元素存储到变量的时候，以后每一次在Javascript函数中使用这个变量的时候都会再去访问一下这个变量对应的html元素。

2.减少重排

放弃传统操作DOM的时代，基于vue/react开始数据影响视图模式。

避免设置大量的style内联属性，因为通过设置style属性改变结点样式的话，每一次设置都会触发一次reflow，所以最好是使用class属性。

不要使用table布局，因为table中某个元素一旦触发了reflow，那么整个table的元素都会触发reflow。那么在不得已使用table的场合，可以设置table-layout:auto;或者是table-layout:fixed这样可以让table一行一行的渲染，这种做法也是为了限制reflow的影响范围。

尽量少使用display：none可以使用visibility：hidden代替，display：none会造成重排，visibility：hidden只会造成重绘。

使用resize事件时，做防抖和节流处理。

分离读写操作（现代的浏览器都有渲染队列的机制）

分离读写减少重排的原理

```
<style>
  #box{
    width:100px;
    height:100px;
    border:10px solid #ddd;
  }
</style>
<body>
  <div id="box"></div>
  <script>
    // 读写分离，一次重排
    let box = document.getElementById('box')
    box.style.width='200px';//（写）js改变样式，加入渲染队列中，顿一下，查看下一行是否还是修改样式，如果是则再加入到渲染队列，一直到下一行代码不是修改样式为止
    box.style.height='200px';//（写）
    box.style.margin='10px';//（写）
    console.log(box.clientWidth);//（读）
  </script>
  <script>
    // 没做到读写分离，两次重排
    box.style.width='200px';//（写）js改变样式，加入渲染队列中,顿一下，下一行不是修改样式的代码，浏览器就会直接渲染一次(重排)
    console.log(box.clientWidth);//（读）
    box.style.height='200px';//（写）
    box.style.margin='10px';//（写）
  </script>
</body>
```

3.css及优化动画

少用css表达式

样式集中改变（批量处理） 减少通过JavaScript代码修改元素样式，尽量使用修改class名方式操作样式或动画；

可以把动画效果应用到position属性为absolute或fixed的元素上，这样对其他元素影响较小

动画实现的速度的选择:牺牲平滑度换取速度。比如实现一个动画，以1个像素为单位移动这样最平滑，但是reflow就会过于频繁，大量消耗CPU资源，如果以3个像素为单位移动则会好很多。

开启css3动画硬件加速（GPU加速）把渲染计算交给GPU。（能用transform做的就不要用其他的，因为transform可以开启硬件加速，而硬件加速可以规避重排。直接跳过重排、重绘，走合成进程）

7、总结

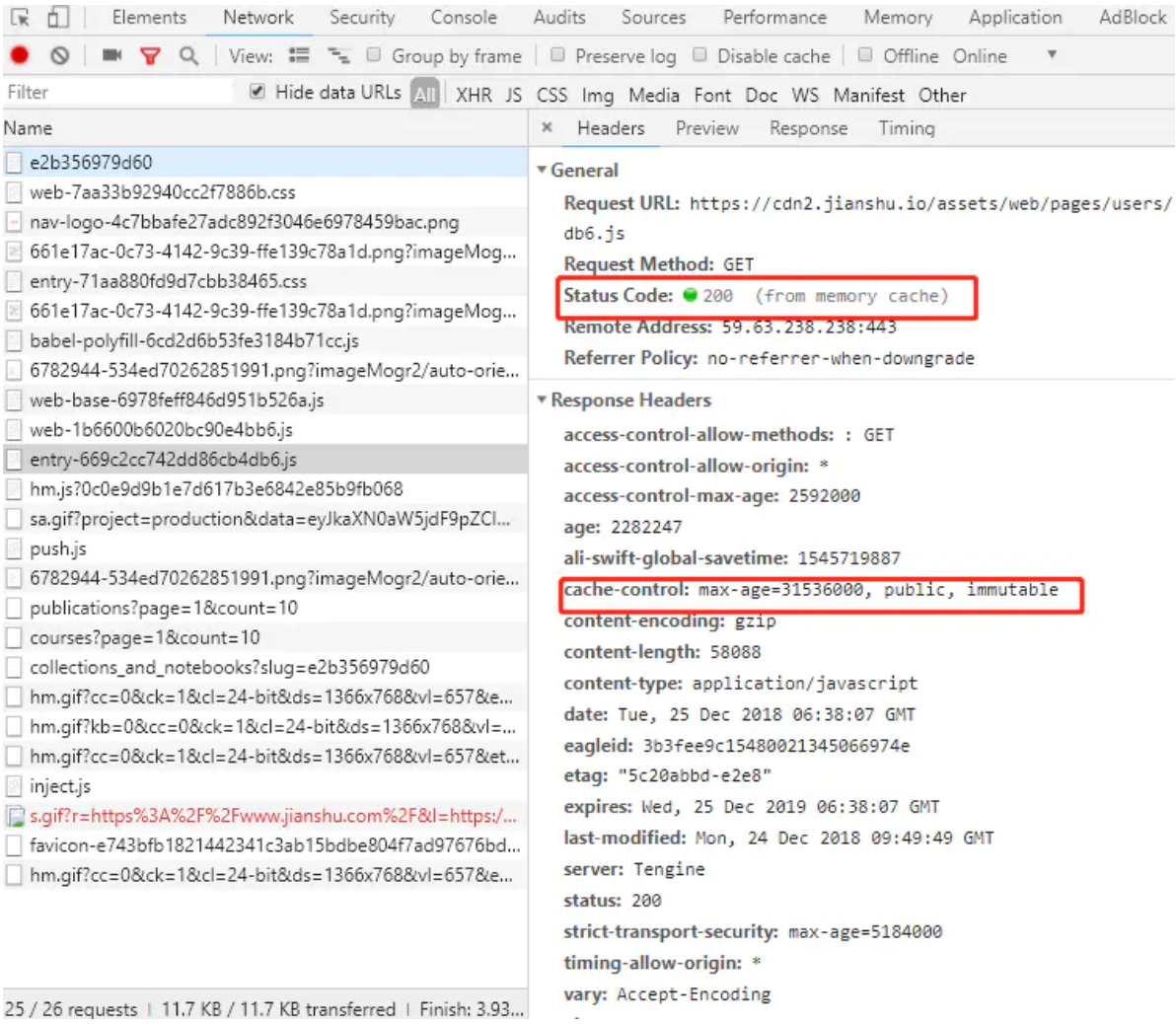
重排一定会引起重绘，而重绘不一定会引起重排，重绘的开销较小，重排的代价较高。在日常开发过程中应该尽量减少重排和重绘操作。

8、强缓存和协商缓存

强缓存

到底什么是强缓存？强在哪？其实强是强制的意思。当浏览器去请求某个文件的时候，服务端就在respone header里面对该文件做了缓存配置。缓存的时间、缓存类型都由服务端控制，具体表现为：respone header 的cache-control，常见的设置是max-age public private no-cache no-store等

如下图，
设置了**cache-control:max-age=31536000,public,immutable**



max-age表示缓存的时间是31536000秒（1年），public表示可以被浏览器和代理服务器缓存，代理服务器一般可用nginx来做。immutable表示该资源永远不变，但是实际上该资源并不是永远不变，它这么设置的意思是为了让用户在刷新页面的时候不要去请求服务器！啥意思？就是说，如果你只设置了cahe-control:max-age=31536000,public 这属于强缓存，每次用户正常打开这个页面，浏览器会判断缓存是否过期，没有过期就从缓存中读取数据；但是有一些 "聪明" 的用户会点击浏览器左上角的刷新按钮去刷新页面，这时候就算资源没有过期（1年没这么快过），浏览器也会直接去请求服务器，这就是额外的请求消耗了，这时候就相当于是走协商缓存的流程了（下面会讲到）。如果cahe-control:max-age=315360000,public再加个immutable的话，就算用户刷新页面，浏览器也不会发起请求去服务，浏览器会直接从本地磁盘或者内存中读取缓存并返回200状态，看上图的红色框（from memory cache）。这是2015年facebook团队向制定 HTTP 标准的 IETF 工作组提到的建议：他们希望 HTTP 协议

能给 Cache-Control 响应头增加一个属性字段表明该资源永不过期，浏览器就没必要再为这些资源发送条件请求了。

强缓存总结

1. cache-control: max-age=xxxx，public
   客户端和代理服务器都可以缓存该资源；
   客户端在xxx秒的有效期内，如果有请求该资源的需求的话就直接读取缓存,statu code:200 ，如果用户做了刷新操作，就向服务器发起http请求
2. cache-control: max-age=xxxx，private
   只让客户端可以缓存该资源；代理服务器不缓存
   客户端在xxx秒内直接读取缓存,statu code:200
3. cache-control: max-age=xxxx，immutable
   客户端在xxx秒的有效期内，如果有请求该资源的需求的话就直接读取缓存,statu code:200 ，即使用户做了刷新操作，也不向服务器发起http请求
4. cache-control: no-cache
   跳过设置强缓存，但是不妨碍设置协商缓存；一般如果你做了强缓存，只有在强缓存失效了才走协商缓存的，设置了no-cache就不会走强缓存了，每次请求都回询问服务端。
5. cache-control: no-store
   不缓存，这个会让客户端、服务器都不缓存，也就没有所谓的强缓存、协商缓存了。

协商缓存

上面说到的强缓存就是给资源设置个过期时间，客户端每次请求资源时都会看是否过期；只有在过期才会去询问服务器。所以，强缓存就是为了给客户端自给自足用的。而当某天，客户端请求该资源时发现其过期了，这是就会去请求服务器了，而这时候去请求服务器的这过程就可以设置协商缓存。这时候，协商缓存就是需要客户端和服务器两端进行交互的。

怎么设置协商缓存?

response header里面的设置

```
etag: '5c20abbd-e2e8'
last-modified: Mon, 24 Dec 2018 09:49:49 GMT
```

etag：每个文件有一个，改动文件了就变了，就是个文件hash，每个文件唯一，就像用webpack打包的时候，每个资源都会有这个东西，如： app.js打包后变为 app.c20abbde.js，加个唯一hash，也是为了解决缓存问题。

last-modified：文件的修改时间，精确到秒

也就是说，每次请求返回来 response header 中的 etag和 last-modified，在下次请求时在 request header 就把这两个带上，服务端把你带过来的标识进行对比，然后判断资源是否更改了，如果更改就直接返回新的资源，和更新对应的response header的标识etag、last-modified。如果资源没有变，那就不变etag、last-modified，这时候对客户端来说，每次请求都是要进行协商缓存了，即：

发请求-->看资源是否过期-->过期-->请求服务器-->服务器对比资源是否真的过期-->没过期-->返回304状态码-->客户端用缓存的老资源。

这就是一条完整的协商缓存的过程。

当然，当服务端发现资源真的过期的时候，会走如下流程：

发请求-->看资源是否过期-->过期-->请求服务器-->服务器对比资源是否真的过期-->过期-->返回200状态码-->客户端如第一次接收该资源一样，记下它的cache-control中的max-age、etag、last-modified等。

所以协商缓存步骤总结：

请求资源时，把用户本地该资源的 etag 同时带到服务端，服务端和最新资源做对比。
如果资源没更改，返回304，浏览器读取本地缓存。
如果资源有更改，返回200，返回最新的资源。

9、常见的http请求头

Request Header:

```
Host: www.test.com/  //请求的目标域名和端口号
Origin: http://localhost:8081/  //请求的来源域名和端口号 （跨域请求时，浏览器会自动带上这个头信息）
Referer: https:/localhost:8081/link?query=xxxxx  //请求资源的完整URI
User-Agent //浏览器信息
Cookie: //当前域名下的Cookie
Accept: text/html,image/apng  //代表客户端希望接受的数据类型是html或者是png图片类型
Accept-Encoding: gzip, deflate  //代表客户端能支持gzip和deflate格式的压缩
Accept-Language: zh-CN,zh;q=0.9  //代表客户端可以支持语言zh-CN或者zh(值得一提的是q(0~1)是优先级权重的意思，不写默认为1，这里zh-CN是1，zh是0.9)
Connection: keep-alive  //告诉服务器，客户端需要的tcp连接是一个长连接
If-None-Match  //如果内容未改变返回304代码，对应Etag
If-Modified-Since //对应last-midified，未被修改则返回304代码
```

Response Header:

```
Date: //服务端发送资源时的服务器时间
Expires: //缓存过期时间
Cache-Control:  no-cache  // 缓存方式
Etag  // 文件内容hash
Last-Modified  //最近一次文件修改时间
Content-Type: text/html; charset=utf-8  //编码格式
Content-Encoding: gzip  //采用gzip对资源进行解码
Connection: keep-alive  //tcp是长连接
Set-Cookie //设置Http Cookie
```

10、cookie的属性

Cookie属性：
name字段：一个cookie的名称
value字段：一个cookie的值
domain字段：可以访问此cookie的域名
path字段：可以访问此cookie的页面路径
Size字段：此cookie大小
http字段：cookie的httponly属性，若此属性为True，则只有在http请求头中会有此cookie信息，而不能通过document.cookie来访问此cookie。
secure字段：设置是否只能通过https来传递此条cookie。
expires/Max-Age字段：设置cookie超时时间。如果设置的值为一个时间，则当到达该时间时此cookie失效。不设置的话默认是session，意思是cookie会和session一起失效，当浏览器关闭（并不是浏览器标签关闭，而是整个浏览器关闭）后，cookie失效。

11、观察者模式和发布订阅模式的区别

有些人认为观察者模式就是发布订阅模式，实际上观察者模式是包含了订阅发布模式，发布订阅模式只是观察者模式中的一种。*观察者模式是观察者和被观察者之间的通信，而发布订阅模式中间增加了一个中转层，通过第三方来分发信息。*

观察者模式

```
// Subject为被观察者，Subject中的状态（state）改变，就通知 Observer更新
class Subject {
  constructor() {
    this.observes = []
    this.state = false
  }
  // this.observes存储观察者
  attach(observe){
    this.observes.push(observe)
  }
  // 状态改变，通知 Observer 触发更新
  setState(newState){
    this.state = newState
    this.observes.forEach( observer => observer.update(newState))
  }
}
// Observer为观察者，观察Subject的状态是否改变
class Observer {
  constructor(name) {
    this.name = name
  }
  // 更新
  update(state){
    console.log(this.name + "，接收到了通知，被观察者的属性变为 " + state)
  }
}
var sub = new Subject()
var obs1 = new Observer('观察者1')
var obs2 = new Observer('观察者2')
sub.attach(obs1)
sub.attach(obs2)
// 被观察者的状态改变，触发观察者更新
sub.setState(true)

```

vue中数据劫持中就用到了观察者模式，data中的属性一发生变化，就通知view界面更新，从而实现数据驱动，想要进一步了解vue底层原理，可以参考可以参考github上的一篇文章 MVVM实现

发布订阅模式

```
// 发布订阅
  class Events {
    constructor() {
      this.sub = {} // 容器
    }
    // 根绝不同 name，订阅对应函数
    $on(name, fn) {
      const wrap = this.sub[name] || (this.sub[name] = [])
      wrap.push(fn)
    }
    // 遍历所有相同name的订阅函数，并发布
```

```
      $emit(name, ...args) {
        const fns = this.sub[name] || []
        fns.forEach(fn => {
          fn.apply(this, args)
        })
      }
      // 销毁，避免内存泄漏
      $of(name){
        this.sub[name] = null
      }
    }
    // event 相当于中转器
    const event = new Events()
    // 订阅
    event.$on('eventname', function (a, b) {
      console.log(a, b)
    })
    event.$on('eventname', function (a, b) {
      console.log(a, b)
    })
    // 发布
    event.$emit('eventname', 'a', 'b')
```

vue中的兄弟组件通信bus的原理就是发布订阅模式，该模式有个缺点，当你订阅一个消息后，也许此消息最后都未发生，但这个订阅者会始终存在于内存中。所以该消息不使用的时候，调用$of销毁，以避免内存泄漏。

- 观察者和被观察者，是松耦合的关系
- 发布者和订阅者，则完全不存在耦合

12、ajax、axios、fetch的区别

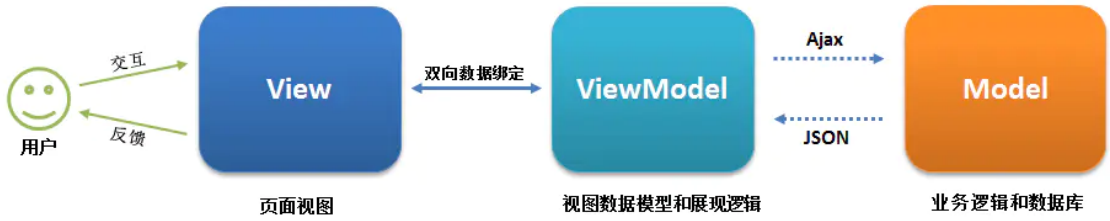
1.jQuery ajax

```
$.ajax({
   type: 'POST',
   url: url,
   data: data,
   dataType: dataType,
   success: function () {},
   error: function () {}
});
```

传统 Ajax 指的是 XMLHttpRequest（XHR）， 最早出现的发送后端请求技术，隶属于原始js中，核心使用XMLHttpRequest对象，多个请求之间如果有先后关系的话，就会出现**回调地狱**。
JQuery ajax 是对原生XHR的封装，除此以外还增添了对**JSONP**的支持。经过多年的更新维护，真的已经是非常的方便了，优点无需多言；如果是硬要举出几个缺点，那可能只有：
1.本身是针对MVC的编程,不符合现在前端**MVVM**的浪潮
2.基于原生的XHR开发，XHR本身的架构不清晰。
3.JQuery整个项目太大，单纯使用ajax却要引入整个JQuery非常的不合理（采取个性化打包的方案又不能享受CDN服务）
4.不符合关注分离（Separation of Concerns）的原则
5.配置和调用方式非常混乱，而且基于事件的异步模型不友好。
PS:MVVM(Model-View-ViewModel), 源自于经典的 Model–View–Controller（MVC）模式。MVVM 的出现促进了 GUI 前端开发与后端业务逻辑的分离，极大地提高了前端开发效率。MVVM 的核

心是 ViewModel 层，它就像是一个中转站（value converter），负责转换 Model 中的数据对象来让数据变得更容易管理和使用，该层向上与视图层进行双向数据绑定，向下与 Model 层通过接口请求进行数据交互，起呈上启下作用。View 层展现的不是 Model 层的数据，而是 ViewModel 的数据，由 ViewModel 负责与 Model 层交互，这就完全解耦了 View 层和 Model 层，这个解耦是至关重要的，它是前后端分离方案实施的最重要一环。

如下图所示：



2.axios

```
axios({
    method: 'post',
    url: '/user/12345',
    data: {
        firstName: 'Fred',
        lastName: 'Flintstone'
    }
})
.then(function (response) {
    console.log(response);
})
.catch(function (error) {
    console.log(error);
});
```

Vue2.0之后，尤雨溪推荐大家用axios替换JQuery ajax，想必让axios进入了很多人的目光中。

axios 是一个基于Promise 用于浏览器和 nodejs 的 HTTP 客户端，本质上也是对原生XHR的封装，只不过它是Promise的实现版本，符合最新的ES规范，它本身具有以下特征：

1.从浏览器中创建 XMLHttpRequest

2.支持 Promise API

3.客户端支持防止CSRF

4.提供了一些并发请求的接口（重要，方便了很多的操作）

5.从 node.js 创建 http 请求

6.拦截请求和响应

7.转换请求和响应数据

8.取消请求

9.自动转换JSON数据

PS:防止CSRF:就是让你的每个请求都带一个从cookie中拿到的key, 根据浏览器同源策略，假冒的网站是拿不到你cookie中得key的，这样，后台就可以轻松辨别出这个请求是否是用户在假冒网站上的误导输入，从而采取正确的策略。

3.fetch

```
try {
  let response = await fetch(url);
  let data = response.json();
  console.log(data);
} catch(e) {
  console.log("Oops, error", e);
}
```

fetch号称是AJAX的替代品，是在ES6出现的，使用了ES6中的promise对象。Fetch是基于promise设计的。Fetch的代码结构比起ajax简单多了，参数有点像jQuery ajax。但是，一定记住**fetch不是ajax的进一步封装，而是原生js，没有使用XMLHttpRequest对象。**

fetch的优点：

1.符合关注分离，没有将输入、输出和用事件来跟踪的状态混杂在一个对象里

2.更好更方便的写法

坦白说，上面的理由对我来说完全没有什么说服力，因为不管是Jquery还是Axios都已经帮我们把xhr封装的足够好，使用起来也足够方便，为什么我们还要花费大力气去学习fetch?

我认为fetch的优势主要优势就是：

```
1.  语法简洁，更加语义化
2.  基于标准 Promise 实现，支持 async/await
3.  同构方便，使用 [isomorphic-fetch](https://github.com/matthew-andrews/isomorphic-fetch)
4.更加底层，提供的API丰富（request, response）
5.脱离了XHR，是ES规范里新的实现方式
```

最近在使用fetch的时候，也遇到了不少的问题：

fetch是一个低层次的API，你可以把它考虑成原生的XHR，所以使用起来并不是那么舒服，需要进行封装。

例如：

```
1）fetch只对网络请求报错，对400，500都当做成功的请求，服务器返回 400，500 错误码时并不会reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。
2）fetch默认不会带cookie，需要添加配置项： fetch(url, {credentials: 'include'})
3）fetch不支持abort，不支持超时控制，使用setTimeout及Promise.reject的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费
4）fetch没有办法原生监测请求的进度，而XHR可以
```

总结：axios既提供了并发的封装，也没有fetch的各种问题，而且体积也较小，当之无愧现在最应该选用的请求的方式。

13、跨域

1.跨域问题的由来

何谓同源:URL由协议、域名、端口和路径组成，如果两个URL的协议、域名和端口相同，则表示它们同源。浏览器的同源策略，从一个域上加载的脚本不允许访问另外一个域的文档属性 ，是浏览器上为安全性考虑实施的非常重要的安全策略。举个例子：比如一个恶意网站的页面通过iframe嵌入了银行的登录页面（二者不同源），如果没有同源限制，恶意网页上的javascript脚本就可以在用户登录银行的时候获取用户名和密码。

克服跨域限制的方法有(实践中后两种最常用，所以重点介绍):

(1)通过jsonp跨域

(2)通过修改document.domain来跨子域

(3)使用window.name来进行跨域

(4)使用HTML5中新引进的window.postMessage方法来跨域传送数据
(5)使用代理服务器,使用代理方式跨域更加直接，因为同源限制是浏览器实现的。如果请求不是从浏览器发起的，就不存在跨域问题了。
使用这个方法跨域步骤如下：

1. 把访问其它域的请求替换为本域的请求
2. 服务器端的动态脚本负责将本域的请求转发成实际的请求
   为了通过Ajax从http://localhost:8080访问http://localhost:8081/api，可以将请求发往http://localhost:8080/api。
   然后利用Apache Web/Nginx 服务器的Reverse Proxy功能做如下配置：ProxyPass /api http://localhost:8081/api

(6)cors

简单请求

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-Origin`字段（详见下文），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
Content-Type: text/html; charset=utf-8

上面的头信息之中，有三个与CORS请求相关的字段，都以`Access-Control-`开头。

（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`*`，表示接受任意域名的请求。

（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非简单请求-预检请求

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，上例是`PUT`。

（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，上例是`X-Custom-Header`。

14、DOCTYPE的作用

DOCTYPE是document type (文档类型) 的缩写。声明位于文档的最前面，处于标签之前，它不是html标签。主要作用是告诉浏览器的解析器使用哪种HTML规范或者XHTML规范来解析页面。