1. Vue
2. 生命周期
3. v-model数据双向绑定
4. Vue-router
5. Vuex
6. 常用api（nextTick）
7. 注意事项
8. React
9. 生命周期
10. 数据方向绑定需要。。。All in js
11. React-router
12. Redux
13. setStates什么时候执行
14. 常用api
15. 注意事项
16. Jquery

1.事件怎么曝露

1. Javascript
2. 原型链
3. 面向对象（继承）this/call/apply/bind
4. 数据操作。。字符串、数组。。
5. 数据存储（堆栈），怎么优化
6. 变量
7. 垃圾回收机制
8. Es6新的语法
9. dom事件
10. http协议
11. 算法
12. js运行机制
13. 深浅拷贝
14. 防抖节流
15. Reduce
16. Webpack
17. 基本配置
18. 性能优化
19. 常用插件
20. Node
21. Html
22. Css
23. css盒模型
24. 常用布局
25. Ie8
26. Jenkine
27. Nginx
28. 其他
29. 同源策略（域名，协议，端口）
30. 前端安全分两类：CSRF、XSS

浏览器请求的时候你发现一次几个是一组，为什么

那怎么保证返回的跟请求顺序一样

事件为什么要用事件委托而不直接绑定？

Proxy和object. definedefine

Promise,async,await

NextTick

样式？

Js数据存在哪，怎么优化

Js变量

事件循环

Es6数组的操作

样式一般会怎么优化

Jq事件是怎么暴露的

设计模式

Js垃圾回收

Vue-model是如何实现的

原型链

双向数据绑定

vuex

深拷贝浅拷贝自己实现

事件委托

跨域

**A、Http相关问题：协议（浏览器缓存）**

一、GET和POST的区别

1.get请求会被浏览器缓存起来，post请求内容不会被缓存。

2.对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

3.get请求参数跟在url后面，有长度限制(浏览器长度限制，如果是内部的get请求则不限制长度)，post请求参数封装在请求头中，无长度限制

二、浏览器输入url按回车背后经历了哪些？

在PC浏览器的地址栏输入一串URL，然后按Enter键这个页面渲染出来，这个过程中都发生了什么事?

1、首先，在浏览器地址栏中输入url，先解析url，检测url地址是否合法

2、浏览器先查看浏览器缓存-系统缓存-路由器缓存，如果缓存中有，会直接在屏幕中显示页面内容。若没有，则跳到第三步操作。  
浏览器缓存：浏览器会记录DNS一段时间，因此，只是第一个地方解析DNS请求；  
操作系统缓存：如果在浏览器缓存中不包含这个记录，则会使系统调用操作系统，获取操作系统的记录(保存最近的DNS查询缓存)；  
路由器缓存：如果上述两个步骤均不能成功获取DNS记录，继续搜索路由器缓存；  
ISP缓存：若上述均失败，继续向ISP搜索。  
3、在发送http请求前，需要域名解析(DNS解析)，解析获取相应的IP地址。  
4、浏览器向服务器发起tcp连接，与浏览器建立tcp三次握手。  
5、握手成功后，浏览器向服务器发送http请求，请求数据包。  
6、服务器处理收到的请求，将数据返回至浏览器  
7、浏览器收到HTTP响应  
8、浏览器解码响应，如果响应可以缓存，则存入缓存。  
9、 浏览器发送请求获取嵌入在HTML中的资源（html，css，javascript，图片，音乐······），对于未知类型，会弹出对话框。  
10、 浏览器发送异步请求。  
11、页面全部渲染结束。

三、http的请求报文和响应报文？

请求类型：get,post,head,put,delete,options,tarce,connect

http请求报文：

1.请求行：请求方法+url

2.请求头：

User-Agent：产生请求的浏览器类型。

Accept：客户端可识别的内容类型列表。

Host：请求的主机名，允许多个域名同处一个IP地址，即虚拟主机。

content-type：请求数据格式

3.空行：它的作用是通过一个空行，告诉服务器请求头部到此为止

4.请求体：请求参数

http响应报文：

1. 状态行：http版本+状态码
2. 响应头：

Allow：服务器支持哪些请求方法(如GET、POST等)。

Content-Encoding：文档的编码(Encode)方法

Content- Type：表示后面的文档属于什么MIME类型。（json、html）

1. 响应体：返回的数据
2. http常见状态码

1.200：请求成功

2.301：永久重定向，被请求的资源已永久移动到新位置

3.302：临时重定向，请求的资源现在临时从不同的 URI 响应请求

4.400语义有误，当前请求无法被服务器理解。除非进行修改，否则客户端不应该重复提交这个请求或者请求参数有误。

5.401：需要验证，权限问题

6.402：服务器已经理解请求，拒绝执行。权限问题

7.404：找不到资源，地址错误

8.500：服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。一般来说，这个问题都会在服务器的程序码出错时出现。

9.501：服务器无法识别请求的方法

10.503：由于临时的服务器维护或者过载，服务器当前无法处理请求

五、http和https区别

HTTPS和HTTP的区别主要如下：HTTPS = HTTP + SSL

　　1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

　　2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

　　3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

　　4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

六、数据存储（cookie/session/localstorage）

1.cookie：存储在本地浏览器，生命周期为浏览器关闭结束；cookie有自身作用域范围，可设置；cookie大小限制为4kb，一般网站最多存储20个，字符串类型；cookie不是很安全，别人可以分析存放在本地的cookie并进行cookie欺骗,考虑到安全应当使用session；

2.session：存储在服务器，访问量大会占用服务器性能；不能区分路径，同一个用户在访问一个网站期间，所有的session在任何一个地方都可以访问到；session必需借助cookie来使用；存储的是对象；

3.localstorage：存储在本地，永久有效，需手动删除；可存储数据量5M+

七、http缓存机制（强制缓存和协商缓存）

1.**强制缓存**原理：所请求的数据在缓存数据库中尚未过期时，不与服务器进行交互，直接使用缓存数据库中的数据

Cache-Control通用消息头用于在http 请求和响应中通过指定指令来实现缓存机制。其常用的几个取值有：

private：客户端可以缓存

public：客户端和代理服务器都可以缓存

max-age=xxx：缓存的内容将在xxx 秒后失效

s-max-age=xxx：同s-max-age，但仅适用于共享缓存(比如各个代理)，并且私有缓存中忽略。

no-cache：需要使用协商缓存来验证缓存数据

no-store：所有内容都不会缓存，强缓存和协商缓存都不会触发

must-revalidate：缓存必须在使用之前验证旧资源的状态，并且不可使用过期资源。

2.当强缓存过期未命中或者响应报文Cache-Control中有must-revalidate标识必须每次请求验证资源的状态时，便使用协商缓存的方式去处理缓存文件。

**协商缓存**主要原理是从缓存数据库中取出缓存的标识，然后向浏览器发送请求验证请求的数据是否已经更新，如果已更新则返回新的数据，若未更新则使用缓存数据库中的缓存数据。

**B、Call apply bind（改变this指向）**

1.第一个参数都是this的指向对象

2.call参数逗号分隔obj.myFun.call(db,'成都', ... ,function )

3.apply参数放在数组里obj.myFun.apply(db,['成都', ...,{} ])

4.bind除了返回函数外，参数和call一样，bind调用需要()()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**C、this指向问题**

#### 一、默认绑定：独立函数调用时（函数通过函数名（）调用），this 指向全局对象，如果使用严格模式strict，那么全局对象无法使用默认绑定， this绑定至 undefined。

function foo() { var a = 3 console.log( this.a ); boo() } function boo() { var a = 4 console.log( this.a ); }

function coo() { 'use strict' var a = 4 console.log('coo', this.a ); }

var a = 2;

foo();  // 2 2

coo(); //Cannot read property 'a' of undefined

#### 二、隐含绑定：方法调用是指通过对象来调用其方法函数，类似于obj.foo(..)的调用方式。此时，this指向调用该方法的对象，注意，是最终调用该方法的对象。

#### 隐含丢失：this 绑定最常让人沮丧的事情之一，就是当一个 隐含绑定 丢失了它的绑定，这通常意味着它会退回到 默认绑定， 根据 strict mode 的状态，其结果不是全局对象就是 undefined

function foo() { console.log( this.a ); }

var obj = { a: 2, foo: foo };

var bar = obj.foo; // 函数引用！

var a = "oops, global"; // `a` 也是一个全局对象的属性

bar(); // "oops, global" this指向window

#### 三、显示绑定call,apply,bind(详见B)

四、new绑定

当在函数前面被加入 new 调用时，也就是构造器调用时，下面这些事情会自动完成：

​ 1.一个全新的对象会凭空创建（就是被构建）

​ 2.这个新构建的对象会被接入原形链（[[Prototype]]-linked）

​ 3.这个新构建的对象被设置为函数调用的 this 绑定

​ 4.除非函数返回一个它自己的其他对象(引用类型)，否则这个被 new 调用的函数将 自动 返回这个新构建的对象（如果函数返回一个对象，则this指向返回的对象）。

function foo(a) {this.a = a;}

var bar = new foo( 2 );console.log( bar.a ); // 2

function foo(a) {this.a = a;return {}}

var bar = new foo( 2 );

console.log( bar.a ); // undefined

#### 五、es6的箭头函数:对于箭头函数，它this是词法绑定的。箭头函数体内的 this 对象，就是定义时所在的对象，而不是使用时所在的对象

window.name = 'win';

const obj1 = {

name: 'joy',

getName: () => {

console.log(this); //window 调用前this是什么函数里面的this就是什么

console.log(this.name); //win

}

};

obj1.getName();

**this5种绑定方式的优先级：**

**箭头函数 > new > 显式 > 隐式 > 默认绑定**

**- 函数是否在 `new` 中调用（`new` 绑定）？如果是的话 `this` 绑定的是新创建的对象。**

**- 函数是否通过 `call`、`apply` （显示绑定）或者硬绑定？如果是的话，`this` 绑定的是指定的对象。**

**- 函数是否在某个上下文对象中调用（隐式绑定）？如果是的话，`this` 绑定的是那个上下文对象。**

**- 如果都不是的话，使用默认绑定。如果在严格模式下，就绑定到 `undefined`，否则绑定到全局对象**

1. **深浅拷贝**

Js数据类型：Number（数值型）、String（字符串）、Boolean（布尔型）、Object（对象，object和array都属于Object类型）、null、undefined

这里的深浅拷贝是object引用类型数据的深浅拷贝（堆中存储的数据，复制了指针，指针的指向还是这个对象），而栈中存储的数据，每次都会开辟出新的内存存储

1. 浅拷贝

浅拷贝引用类型数据，修改的实际还是同一个对象

1. 深拷贝

JSON.parse(JSON.stringify())

实现深拷贝的方法

#### function deepClone(obj){

//判断参数是不是一个对象

let objClone = obj instanceof Object?[]:{};

if(obj && typeof obj==="object"){

for(key in obj){

if(obj.hasOwnProperty(key)){

//判断ojb子元素是否为对象，如果是，递归复制

if(obj[key]&&typeof obj[key] ==="object"){

objClone[key] = deepClone(obj[key]);

}else{

//如果不是，简单复制

objClone[key] = obj[key];

}

}

}

}

return objClone;

}

1. **防抖节流**

1.防抖

function debounce(fn, wait) {

var timeout = null;

return function() {

if(timeout !== null)  clearTimeout(timeout);

timeout = setTimeout(fn, wait);

}

}

// 处理函数

function handle() {

console.log(Math.random());

}

// 滚动事件

window.addEventListener('scroll', debounce(handle, 1000));

2.节流

function throttle(fn, wait) {

var valid = true;

if(!valid) return false

Valid = false

setTimeOut(()=>{

fn()

valid = true

},delay)

}

// 处理函数

function handle() {

console.log(Math.random());

}

// 滚动事件

window.addEventListener('scroll', throttle(handle, 1000));

1. **事件序列**

一个 JavaScript 运行时包含了一个待处理消息的消息队列。每一个消息都关联着一个用以处理这个消息的回调函数

在 [事件循环](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/EventLoop" \l "%E4%BA%8B%E4%BB%B6%E5%BE%AA%E7%8E%AF) 期间的某个时刻，运行时会从最先进入队列的消息开始处理队列中的消息。被处理的消息会被移出队列，并作为输入参数来调用与之关联的函数。正如前面所提到的，调用一个函数总是会为其创造一个新的栈帧

1. 栈序列（先进后出）

function foo(b) {

var a = 10;

return a + b + 11;

}

function bar(x) {

var y = 3;

return foo(x \* y);

}

console.log(bar(7)); // 返回 42

当调用bar时，第一个帧被创建并压入栈中，帧中包含了bar的参数和局部变量。 当bar调用foo时，第二个帧被创建并被压入栈中，放在第一个帧之上，帧中包含foo的参数和局部变量。当foo执行完毕然后返回时，第二个帧就被弹出栈（剩下bar函数的调用帧 ）。当bar也执行完毕然后返回时，第一个帧也被弹出，栈就被清空了。

1. 堆序列（先进先出，有特例）

console.log(1);

setTimeout(function(){

console.log(2);

Promise.resolve(1).then(function(){

console.log('promise')

})

})

setTimeout(function(){ console.log(3); }) //1 2 promise 3

任务也是有区别的，总有任务会有一些特权—微任务(micro-task)，也有些任务没有特权，只能乖乖排队—宏任务(masro-task)。

通常任务宏任务包括以下任务：

setTimeout

setinterval

setImmediate

I/O

微任务包含以下任务源

process.nextTick

Promise

MutationObserver

1. nodeJs事件循环机制

Node.js采用事件驱动和异步I/O的方式，实现了一个单线程、高并发的JavaScript运行时环境，而单线程就意味着同一时间只能做一件事；

nodejs的单线程是执行JavaScript语言智能是单线程，而nodejs解决高并发可以建立多线程模型，服务器为每个客户端请求分配一个线程，使用同步I/O，系统通过线程切换来弥补同步I/O调用的事件开销。

而事实上，大多数网站的服务器端都不会做太多的计算，它们接收到请求以后，把请求交给其它服务来处理（比如读取数据库），然后等着结果返回，最后再把结果发给客户端。因此，Node.js 针对这一事实采用了单线程模型来处理，它不会为每个接入请求分配一个线程，而是用一个主线程处理所有的请求，然后对 I/O 操作进行异步处理，避开了创建、销毁线程以及在线程间切换所需的开销和复杂性。

### 事件循环

Node.js 在主线程里维护了一个事件队列，当接到请求后，就将该请求作为一个事件放入这个队列中，然后继续接收其他请求。当主线程空闲时(没有请求接入时)，就开始循环事件队列，检查队列中是否有要处理的事件，这时要分两种情况：如果是非 I/O 任务，就亲自处理，并通过回调函数返回到上层调用；如果是 I/O 任务，就从 线程池 中拿出一个线程来处理这个事件，并指定回调函数，然后继续循环队列中的其他事件。

当线程中的 I/O 任务完成以后，就执行指定的回调函数，并把这个完成的事件放到事件队列的尾部，等待事件循环，当主线程再次循环到该事件时，就直接处理并返回给上层调用。 这个过程就叫 事件循环 (Event Loop)

**F.原型链**

Es6新特性

Webpack性能优化

Redux有什么好处

React setState是啥时候更新

Js运算符优先级（）

闭包

要想延迟指定时间实现( )

v-model

Mixins

页面如何优化？

Vue diff算法

浏览器的重绘和重排

ddd

- [ ] 定时器 this window

- [ ] promise原理，all,

- [ ] 执行顺序

- [ ] promise.then settimeout 宏任务 微任务

- [ ] new发生什么

- [ ] socket

- [ ] ts

- [ ] 跨域

- [ ] tcp

- [ ] http状态码

- [ ] css position

- [ ] zindex 大反而在下

- [ ] css 长宽比固定

- [ ] react 算法

- [ ] react 生命周期 数据请求

- [ ] didMount 父子组件 执行顺序

- [ ] 函数组件和类组件区别

- [ ] react hooks 条件判断执行

- [ ] vue watch 箭头函数

- [ ] node

块级格式上下文

Vue和react区别

React fiber算法

字符串原型链

浏览器缓存内容更新怎么办

Webpack

事件循环

事件冒泡捕捉

Css盒子模型

Vite

React同一个页面重复请求如何去优化

Vue ssr

vue 组件间通讯

Vue单页面性能优化

Vue双向数据绑定原理

V-model如何实现

Vue router路由模式，优缺点

单页面应用优缺点