3.this

```
let a = {}
let fn = function () { console.log(this) }
fn.bind().bind(a)() // => ?
```

上述代码可以转换为

```
// fn.bind().bind(a) 等于
let fn2 = function fn1() {
  return function() {
    return fn.apply()
  }.apply(a)
}
fn2()
```

可以从上述代码中发现，不管我们给函数 bind 几次，fn 中的 this 永远由第一次 bind 决定，所以结果永远是 window

this的优先级

> 首先，new 的方式优先级最高，接下来是 bind 这些函数，然后是 obj.foo() 这种调用方式，最后是 foo 这种调用方式，同时，箭头函数的 this 一旦被绑定，就不会再被任何方式所改变。

4.==和===的区别

- 对于==来说，如果双方的类型不一致，就进行类型和转换

1. 首先会判断两者类型是否相同。相同的话就是比大小了
2. 类型不相同的话，那么就会进行类型转换
3. 会先判断是否在对比 null 和 undefined，是的话就会返回 true
4. 判断两者类型是否为 string 和 number，是的话就会将字符串转换为 number
5. 判断其中一方是否为 boolean，是的话就会把 boolean 转为 number 再进行判断
6. 判断其中一方是否为 object 且另一方为 string、number 或者 symbol，是的话就会把 object 转为原始类型再进行判断

- 对于===来说，判断两者的值是否相同

8.var、let和const

涉及面试题：什么是提升？什么是暂时性死区？var、let 及 const 区别?
函数提升优先于变量提升，函数提升会把整个函数挪到作用域顶部，变量提升只会把声明挪到作用域顶

部
var 存在提升，我们能在声明之前使用。let、const 因为暂时性死区的原因，不能在声明前使用
var 在全局作用域下声明变量会导致变量挂载在 window上，其他两者不会
let 和 const 作用基本一致，但是后者声明的变量不能再次赋值

9.原型继承与class继承

涉及面试题：原型如何实现继承？Class 如何实现继承？Class 本质是什么？

- 原型：组合继承(call)和寄生组合继承(call和Object.create)
- class：核心在于使用extends表明继承自哪个父类，并且在子类构造函数中必须调用super，这句代码可以看成Parent.call(this,arg)

10.模块化

涉及面试题：为什么要使用模块化？都有哪几种方式可以实现模块化，各有什么特点？

- 原因是：解决命名冲突、提供复用性、提高代码可维护性

实现方法：
立即执行函数：在早期，使用立即执行函数实现模块化是常见的手段，通过函数作用域解决了命名冲突、污染全局作用域的问题
CommonJS：CommonJS 最早是 Node 在使用，目前也仍然广泛使用，比如在 Webpack 中你就能见到它，当然目前在 Node 中的模块管理已经和 CommonJS有一些区别了

```
// a.js
module.exports = {
    a: 1
}
// or
exports.a = 1
// b.js
var module = require('./a.js')
module.a // -> log 1
```

- module的基本实现

```
ar module = require('./a.js')
module.a
// 这里其实就是包装了一层立即执行函数，这样就不会污染全局变量了，
// 重要的是 module 这里，module 是 Node 独有的一个变量
module.exports = {
    a: 1
}
// module 基本实现
var module = {
  id: 'xxxx', // 我总得知道怎么去找到他吧
  exports: {} // exports 就是个空对象
}
// 这个是为什么 exports 和 module.exports 用法相似的原因
var exports = module.exports
var load = function (module) {
    // 导出的东西
    var a = 1
    module.exports = a
    return module.exports
};
// 然后当我 require 的时候去找到独特的
// id，然后将要使用的东西用立即执行函数包装下，over
```

ES Module

> ES Module是原生实现的模块化方案，与CommonJS有以下几个不同

- CommonJS 支持动态导入，也就是 require(${path}/xx.js)，后者目前不支持，但是已有提案
- CommonJS 是同步导入，因为用于服务端，文件都在本地，同步导入即使卡住主线程影响也不大。而后者是异步导入，因为用于浏览器，需要下载文件，如果也采用同步导入会对渲染有很大影响
- CommonJS 在导出时都是值拷贝，就算导出的值变了，导入的值也不会改变，所以如果想更新值，必须重新导入一次。但是 ES Module 采用实时绑定的方式，导入导出的值都指向同一个内存地址，所以导入值会跟随导出值变化
  ES Module 会编译成 require/exports来执行的

```
// 引入模块 API
import XXX from './a.js'
import { XXX } from './a.js'
// 导出模块 API
export function a() {}
export default function() {}
```

11.简单实现Promise

```
// 三个常量用于表示状态
const PENDING = 'pending'
const RESOLVED = 'resolved'
const REJECTED = 'rejected'

function MyPromise(fn) {
    const that = this
    this.state = PENDING

    // value 变量用于保存 resolve 或者 reject 中传入的值
    this.value = null

    // 用于保存 then 中的回调，因为当执行完 Promise 时状态可能还是等待中，这时候应该把 then 中的回调
    that.resolvedCallbacks = []
    that.rejectedCallbacks = []


    function resolve(value) {
         // 首先两个函数都得判断当前状态是否为等待中
        if(that.state === PENDING) {
            that.state = RESOLVED
            that.value = value

            // 遍历回调数组并执行
            that.resolvedCallbacks.map(cb=>cb(that.value))
        }
    }
    function reject(value) {
        if(that.state === PENDING) {
            that.state = REJECTED
            that.value = value
            that.rejectedCallbacks.map(cb=>cb(that.value))
        }
    }

    // 完成以上两个函数以后，我们就该实现如何执行 Promise 中传入的函数了
    try {
        fn(resolve,reject)
    }cach(e){
        reject(e)
    }
}

// 最后我们来实现较为复杂的 then 函数
MyPromise.prototype.then = function(onFulfilled,onRejected){
  const that = this

  // 判断两个参数是否为函数类型，因为这两个参数是可选参数
  onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : v=>v
  onRejected = typeof onRejected === 'function' ? onRejected : e=>throw e
```

```
  // 当状态不是等待态时，就去执行相对应的函数。如果状态是等待态的话，就往回调函数中 push 函数
  if(this.state === PENDING) {
      this.resolvedCallbacks.push(onFulfilled)
      this.rejectedCallbacks.push(onRejected)
  }
  if(this.state === RESOLVED) {
      onFulfilled(that.value)
  }
  if(this.state === REJECTED) {
      onRejected(that.value)
  }
}
```

12.Event Loop

12.1 进程与线程

涉及面试题：进程与线程的区别？JS单线程带来的好处?

- 进程和线程都是CPU工作时间片的一个描述，进程描述了CPU在运行指令及加载和保存上下文所需的时间，放在应用上来说就代表了一个程序。线程是进程中更小单位，描述了执行一段指令所需的时间

> 把这些概念拿到浏览器中来说，当你打开一个 Tab 页时，其实就是创建了一个进程，一个进程中可以有多个线程，比如渲染线程、JS 引擎线程、HTTP 请求线程等等。当你发起一个请求时，其实就是创建了一个线程，当请求结束后，该线程可能就会被销毁

- 上文说到的 JS 引擎线程和渲染线程，在 JS 运行的时候可能会阻止 UI 渲染，这说明了两个线程是互斥的。这其中的原因是因为 JS 可以修改 DOM，如果在 JS 执行的时候 UI 线程还在工作，就可能导致不能安全的渲染 UI。这其实也是一个单线程的好处，得益于 JS 是单线程运行的，可以达到节省内存，节约上下文切换时间，没有锁的问题的好处

12.2 执行栈

涉及面试题：什么是执行栈

可以把执行栈看做是一个存储函数调用的栈结构，遵循先进后出的原则

12.3 浏览器中的Event Loop

涉及面试题：异步代码执行顺序？什么是Event Loop?

- JS 在执行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 Task（有多种 task） 队列中。一旦执行栈为空，Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为

- 宏任务：script setTimeout setInterval setImmedprocessiate I/O UI rendering
- 微任务：Promise MutationObserver process.nextTick

13.手写bind，apply，call

14.new

涉及面试题：new的原理是什么？通过new方式创建对象和通过字面量创建有什么区别?

- 在调用new的过程中会发生四件事

1. 新生成一个对象
2. 链接到原型
3. 绑定this
4. 返回新对象

- new的简单实现

```
// 第一版代码
function objectFactory() {
    var obj = new Object(),
    Constructor = [].shift.call(arguments);
    obj.__proto__ = Constructor.prototype;
    // 我们还需要判断返回的值是不是一个对象，如果是一个对象，我们就返回这个对象，如果没有，我们该返回什么
    var ret = Constructor.apply(obj, arguments);
    return typeof ret === "object"? ret:obj;
    //用new Object() 的方式新建了一个对象 obj取出第一个参数，
    //就是我们要传入的构造函数。此外因为 shift 会修改原数组，
    //所以 arguments 会被去除第一个参数将 obj 的原型指向构造函数，
    //这样 obj 就可以访问到构造函数原型中的属性使用 apply，改变构造
    //函数 this 的指向到新建的对象，这样 obj 就可以访问到构造函数中的属性返回 obj

};
```

- 对于对象来说，其实都是通过new产生的，无论是function Foo()还是let a = {b：1}
- 对于创建一个对象来说，更推荐使用字面量的方式来创建对象(无论性能上还是可读性上)。因为使用 new Object() 的方式创建对象需要通过作用域链一层层找到 Object，但是你使用字面量的方式就没这个问题

```
function Foo() {}
// function 就是个语法糖
// 内部等同于 new Function()
let a = { b: 1 }
// 这个字面量内部也是使用了 new Object()
```

15.instanceof的原理

涉及面试题：instanceof的原理是什么?

- instanceof 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 prototype
- 实现instanceof

```
function myInstance(instance,target){
    var prototype = target.prototype
    var instance = instance.__proto__
    while(true){
        if(instance === null || instance === undefined){
            return false
        }
        if(prototype === instance){
            return true
        }
        instance = instance.__proto__
    }
}
```

17.事件机制

涉及面试题：事件的触发过程是怎么样的？知道什么是事件代理嘛?

17.1 事件触发三个阶段

> 事件触发一般是按照捕获，触发处，冒泡这个顺序进行，但如果给子节点同时注册捕获和冒泡事件，事件触发会按照注册的顺序进行

17.2 注册事件

- stopPropagation阻止事件进一步传播

17.3 事件代理

- 如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话应该注册在父节点上，这样可以节省内存并且不需要给子节点注销事件

18.跨域

涉及面试题：什么是跨域？为什么浏览器要使用同源策略？你有几种方式可以解决跨域问题？了解预检请求嘛?

- 因为浏览器出于安全考虑，有同源策略。也就是说，如果协议、域名或者端口有一个不同就是跨域，Ajax 请求会失败。
- 那么是出于什么安全考虑才会引入这种机制呢？ 其实主要是用来防止 CSRF 攻击的。简单点说，CSRF 攻击是利用用户的登录态发起恶意请求。
- 也就是说，没有同源策略的情况下，A 网站可以被任意其他来源的 Ajax 访问到内容。如果你当前 A 网站还存在登录态，那么对方就可以通过 Ajax 获得你的任何信息。当然跨域并不能完全阻止 CSRF。

> 跨域并不能完全组织CSRF，因为请求跨域是发出了请求，而浏览器拦截了响应。表单不会获取新的内容，所以可以发情跨域请求，而Ajax可以获取响应，浏览器认为不安全，从而拦截了响应。

19.存储

- cookie,localStorage,sessionStorage,indexDB
- Service Worker：Service Worker是运行在浏览器背后的独立线程，一般可以用来实现缓存功能。使用SW的话，传输协议必须是HTTPS，因为其中涉及到请求拦截，所以必须用HTTPS来保障安全
    - Service Worker 实现缓存功能一般分为三个步骤：首先需要先注册 Service Worker，然后监听到 install 事件以后就可以缓存需要的文件，那么在下次用户访问的时候就可以通过拦截请求的方式查询是否存在缓存，存在缓存的话就可以直接读取缓存文件，否则就去请求数据。以下是这个步骤的实现：

20.浏览器缓存机制

20.1缓存位置

> 从缓存位置上来说分为四种，并且各自有优先级，当依次查找缓存且都没有命中的时候，才会去请求网络
> servie worker->Memory Cache->Disk Cache->Push Cache->网络请求

1. Service Worker

- service Worker 的缓存与浏览器其他内建的缓存机制不同，它可以让我们自由控制缓存哪些文件、如何匹配缓存、如何读取缓存，并且缓存是持续性的。
  当 Service Worker 没有命中缓存的时候，我们需要去调用 fetch 函数获取数据。也就是说，如果我们没有在 Service Worker 命中缓存的话，会根据缓存查找优先级去查找数据。但是不管我们是从 Memory Cache 中还是从网络请求中获取的数据，浏览器都会显示我们是从 Service Worker 中获取的内容。

2. Memory Cache

- Memory Cache 也就是内存中的缓存，读取内存中的数据肯定比磁盘快。但是内存缓存虽然读取高效，可是缓存持续性很短，会随着进程的释放而释放。 一旦我们关闭 Tab 页面，内存中的缓存也就被释放了。当我们访问过页面以后，再次刷新页面，可以发现很多数据都来自于内存缓存

那么既然内存缓存这么高效，我们是不是能让数据都存放在内存中呢?

- 先说结论，这是不可能的。首先计算机中的内存一定比硬盘容量小得多，操作系统需要精打细算内存的使用，所以能让我们使用的内存必然不多。内存中其实可以存储大部分的文件，比如说 JS、HTML、CSS、图片等等
- 对于大文件来说，大概率是不存储在内存中的，反之优先当前系统内存使用率高的话，文件优先存储进硬盘

3. Disk Cache

- Disk Cache 也就是存储在硬盘中的缓存，读取速度慢点，但是什么都能存储到磁盘中，比之 Memory Cache 胜在容量和存储时效性上。
- 在所有浏览器缓存中，Disk Cache 覆盖面基本是最大的。它会根据 HTTP Herder 中的字段判断哪些资源需要缓存，哪些资源可以不请求直接使用，哪些资源已经过期需要重新请求。并且即使在跨站点的情况下，相同地址的资源一旦被硬盘缓存下来，就不会再次去请求数据

4. Push Cache

- Push Cache 是 HTTP/2 中的内容，当以上三种缓存都没有命中时，它才会被使用。并且缓存时间也很短暂，只在会话（Session）中存在，一旦会话结束就被释放。

5. 网络请求

- 如果所有缓存都没有命中的话，那么只能发起请求来获取资源了。

20.2缓存策略

- 强缓存和协商缓存
- 以上就是缓存策略的所有内容了，看到这里，不知道你是否存在这样一个疑问。如果什么缓存策略都没设置，那么浏览器会怎么处理?
    - 通常会取响应头Date减去Last-Modified值得10%作为缓存时间

20.3实际场景应用缓存策略

频繁变动的资源

> 对于频繁变动的资源，首先需要使用 Cache-Control: no-cache 使浏览器每次都请求服务器，然后配合 ETag 或者 Last-Modified 来验证资源是否有效。这样的做法虽然不能节省请求数量，但是能显著减少响应数据大小。

代码文件

> 这里特指除了HTML外的代码文件，因为HTML文件一般不缓存或者缓存时间很短。一般来说，现在都会使用工具来打包代码，那么我们就可以对文件名进行哈希处理，只有当代码修改后才会生成新的文件名。基于此，我们就可以给代码文件设置缓存有效期一年 Cache-Control: max-age=31536000，这样只有当 HTML 文件中引入的文件名发生了改变才会去下载最新的代码文件，否则就一直使用缓存

21.浏览器渲染原理

21.1 渲染过程

1. 浏览器接收到HTML文件并转换为DOM树
   字节数据=>字符串=>Token=>Node=>DOM
2. 将CSS文件转换为CSSOM树
   字节数据=>字符串=>Token=>Node=>CSSOM

- 在这一过程中，浏览器会确认每一个节点的样式到底是什么，并且这一过程其实是很消耗资源的。因为样式你可以自行设置给某个节点，也可以通过继承获得。在这一过程中，浏览器得递归 CSSOM 树，然后确定具体的元素到底是什么样式。
  所以我们应该尽可能的避免写过于具体的 CSS 选择器，然后对于 HTML 来说也尽量少的添加无意义标签，保证层级扁平

3. 生成渲染树

- 渲染树不是简单的将DOM树和CSSOM树合并。它只会包括需要显示的节点和这些节点的样式信息。
- 生成渲染树后，对渲染树进行布局（回流），调用GPU绘制，合成图层，显示在屏幕上。

21.2 为什么操作DOM慢

- 因为DOM属于渲染引擎中的东西，JS属于JS引擎，当通过JS操作DOM，其实是涉及两个线程间的通信，势必会带来一些性能的损耗。另外还可能会有重绘回流的问题。
  涉及面试题：插入几万个DOM，如何保证页面不卡帧

1. requestAnimationFrame
2. virtualized scroller虚拟滚动：原理就是只渲染可视区域内的内容，非可见区域的那就完全不渲染了，当用户在滚动的时候就实时去替换渲染的内容

21.3 什么情况阻塞渲染

- 渲染的前提是生成渲染树，所以HTML和CSS肯定会阻塞渲染。如果想渲染的越快，应该降低一开始需要渲染的文件大小，并且扁平层级，优化选择器
- 当浏览器解析到script标签的时候，会暂停构建DOM，完成后才会从暂停的地方重新开始。因此，如果想首屏渲染的越快，就不应该在首屏加载JS文件，这就是将script标签放在body标签底部的原因
- 另外也可以给script标签添加async和defer，加上defer后，JS文件会并行下载，但是会放到HTML解析完成后顺序执行，所以对于这种情况你可以把script标签放在任意位置
- 对于没有任何依赖的JS文件可以加上async属性，表示JS文件下载和解析不会阻塞渲染

21.4 回流和重绘

22. 安全防范

22.1 XSS

涉及面试题：什么是XSS攻击？如何防范XSS攻击？什么事CSP?

- XSS就是攻击者想尽一切办法将可以执行的代码注入到网页中
- XSS可以分别持久型和非持久型
  - 持久型：攻击的代码被服务端写入数据库中
  - 非持久型：一般通过修改URL参数的方式加入攻击代码，诱导用户访问链接从而进行攻击

  防御方式：

1. 转义字符
   对用户的输入永远不信任，对引号尖括号斜杠进行转义；但通常需要白名单过滤

```
const xss = require('xss')
let html = xss('<h1 id="title">XSS Demo</h1><script>alert("xss");</script>')
// -> <h1>XSS Demo</h1>&lt;script&gt;alert("xss");&lt;/script&gt;
console.log(html)
```

2. CSP:Content Security Policy
   CSP 本质上就是建立白名单，开发者明确告诉浏览器哪些外部资源可以加载和执行。我们只需要配置规则，如何拦截是由浏览器自己实现的。我们可以通过这种方式来尽量减少 XSS 攻击。
   通常可以通过两种方式开启CSP

- 设置HTTP Header中的Content-Security-Policy
- 设置meta标签的方式<meta http-equiv="Content-Security-Policy">

22.2 CSRF

涉及面试题：什么CSRF攻击？如何防范CSRF攻击?

> CSRF 中文名为跨站请求伪造。原理就是攻击者构造出一个后端请求地址，诱导用户点击或者通过某些途径自动发起请求。如果用户是在登录状态下的话，后端就以为是用户在操作，从而进行相应的逻辑。

如何防御

- GET请求不对数据进行修改
- 不让第三方网站访问到用户Cookie
- 阻止第三方网站请求接口
- 请求时附带验证信息，比如验证码或者Token

22.3 点击劫持

涉及面试题：什么是点击劫持？如何防范？

> 点击劫持是一种视觉欺骗的攻击手段。攻击者将需要攻击的网站通过 iframe 嵌套的方式嵌入自己的网页中，并将 iframe 设置为透明，在页面中透出一个按钮诱导用户点击

1. X-FRAME-OPTIONS

> 是一个HTTP响应头，为了防御iframe嵌套的点击劫持攻击

2. JS防御

23. 从V8中看JS性能优化

23.1测试性能工具

23.2 JS性能优化

> JS 是编译型还是解释型语言其实并不固定。首先 JS 需要有引擎才能运行起来，无论是浏览器还是在 Node 中，这是解释型语言的特性。但是在 V8 引擎下，又引入了 TurboFan 编译器，他会在特定的情况下进行优化，将代码编译成执行效率更高的 Machine Code，当然这个编译器并不是 JS 必须需要的，只是为了提高代码执行性能，所以总的来说 JS 更偏向于解释型语言。

关于V8

> 在这一过程中，JS 代码首先会解析为抽象语法树（AST），然后会通过解释器或者编译器转化为 Bytecode 或者Machine Code
> **什么情况下会被编译为Machine Code**

- 当传入的参数类型固定，不再需要执行逻辑判断，就编译为MC，如果类型改变，就进行DeOptimized。

24. 性能优化

24.1 图片优化

- 减少像素点
- 减少每个像素点能够显示的颜色

24.2 图片加载优化

- 用CSS代替图片
- 一般图片都用CDN加载，可以计算出适配移动端屏幕的宽度，请求裁剪好的图片
- base64格式
- 图片格式：WebP；小图用PNG，图标可用SVG，照片用JPEG

24.3 DNS预解析

<link rel="dns-prefetch" href="//blog.poetries.top">

24.4 防抖

24.5 节流

24.6 预加载

- 预加载其实是声明式的 fetch ，强制浏览器请求资源，并且不会阻塞 onload 事件，可以使用以下代码开启预加载
  <link rel="preload" href="http://blog.poetries.top">

24.7 预渲染

可以通过预渲染将下载的文件预先在后台渲染，可以使用以下代码开启预渲染
<link rel="prerender" href="http://blog.poetries.top">

24.8 懒执行

> 懒执行就是将某些逻辑延迟到使用时再计算。该技术可以用于首屏优化，对于某些耗时逻辑并不需要在首屏就使用的，就可以使用懒执行。懒执行需要唤醒，一般可以通过定时器或者事件的调用来唤醒。

24.9 懒加载

- 懒加载就是将不关键的资源延后加载。

- 懒加载的原理就是只加载自定义区域（通常是可视区域，但也可以是即将进入可视区域）内需要加载的东西。对于图片来说，先设置图片标签的 src 属性为一张占位图，将真实的图片资源放入一个自定义属性中，当进入自定义区域时，就将自定义属性替换为 src 属性，这样图片就会去下载资源，实现了图片懒加载。
- 懒加载不仅可以用于图片，也可以使用在别的资源上。比如进入可视区域才开始播放视频等等。

24.10 CDN

> 因此，我们可以将静态资源尽量使用 CDN 加载，由于浏览器对于单个域名有并发请求上限，可以考虑使用多个 CDN 域名。并且对于 CDN 加载静态资源需要注意 CDN 域名要与主站不同，否则每次请求都会带上主站的 Cookie，平白消耗流量

27.MVVM/虚拟DOM/前端路由

27.1 MVVM

涉及面试题：什么是MVVM？比之MVC有什么区别?

- 传统的 MVC 架构通常是使用控制器更新模型，视图从模型中获取数据去渲染。当用户有输入时，会通过控制器去更新模型，并且通知视图进行更新
- 在 MVVM 架构中，引入了 ViewModel 的概念。ViewModel 只关心数据和业务的处理，不关心 View 如何处理数据，在这种情况下，View和 Model 都可以独立出来，任何一方改变了也不一定需要改变另一方，并且可以将一些可复用的逻辑放在一个 ViewModel 中，让多个 View 复用这个 ViewModel。
- 以 Vue 框架来举例，ViewModel 就是组件的实例。View 就是模板，Model 的话在引入 Vuex 的情况下是完全可以和组件分离的。
- 同样以 Vue 框架来举例，这个隐式的 Binder 层就是 Vue 通过解析模板中的插值和指令从而实现 View 与 ViewModel 的绑定。
  对于 MVVM来说，其实最重要的并不是通过双向绑定或者其他的方式将 View 与 ViewModel 绑定起来，而是通过 ViewModel 将视图中的状态和用户的行为分离出一个抽象，这才是 MVVM 的精髓

27.2 Virtual DOM

涉及面试题：什么是Virtual DOM？为什么Virtual DOM比原生DOM快?

- 因为操作DOM涉及到线程通信，并且可能产生回流，因此操作JS对象会比操作DOM快很多，因此可以通过JS模拟DOM
- 既然DOM可以通过JS对象来模拟，因此也可以通过JS对象来渲染出对应的DOM。这其中的难点在于**如何判断新旧两个JS对象的最小差异并且实现局部更新DOM**

> 完成对比两颗多叉树的时间复杂度是O(n^3)，React团队的算法实现了O(n)，方法是对比同城的节点，不跨层。

1. 首先从上至下从左往右进行深度遍历，这一步中给每个节点增加索引，便于最后渲染差异
2. 一旦节点有子元素，判断子元素是否有不同

> 第一步中，判断新旧节点的tagName是否相同，如果不同就代表节点被替换，如果没有更改，就判断是否有子元素，有子元素就进行第二步算法

> 第二步中，判断原本的列表中是否有节点移除、增加或移动，通过key属性实现

- 判断以上差异时，也同时判断属性是否有变化，全部判断完成后，实现局部更新

Virtual DOM提高性能是优势其一，最大的优势是

- 将Virtual DOM作为一个兼容层，对接非Web端的系统，实现跨段开发
- 可以通过Virtual DOM渲染到其他的平台，实现SSR、同构渲染
- 实现组件的高度抽象化

27.3 路由原理

涉及面试题：前端路由原理？两种实现方式有什么区别？

- 前端路由本质就是监听URL的变化，然后匹配路由规则，显示相应的页面，并且无须刷新页面。共两种实现方式

1. Hash模式：相对简单，兼容性更好

> www.test.com/#/ 就是 Hash URL，当 # 后面的哈希值发生变化时，可以通过 hashchange 事件来监听到 URL 的变化，从而进行跳转页面，并且无论哈希值如何变化，服务端接收到的 URL 请求永远是 www.test.com

2. History模式

> History 模式是 HTML5 新推出的功能，主要使用 history.pushState 和 history.replaceState 改变 URL。通过 History 模式改变 URL 同样不会引起页面的刷新，只会更新浏览器的历史记录。当用户做出浏览器动作时，比如点击后退按钮时会触发 popState 事件

两者对比

- Hash只可以更改#后面的内容，History模式可以通过API设置任意的同源URL
- History 模式可以通过 API 添加任意类型的数据到历史记录中，Hash 模式只能更改哈希值，也就是字符串

- Hash 模式无需后端配置，并且兼容性好。History 模式在用户手动输入地址或者刷新页面的时候会发起 URL 请求，后端需要配置 index.html 页面用于匹配不到静态资源的时候

27.4 Vue和React的区别

- Vue 的表单可以使用 v-model 支持双向绑定，相比于 React 来说开发上更加方便，当然了 v-model 其实就是个语法糖，本质上和 React 写表单的方式没什么区别
- 改变数据方式不同，Vue 修改状态相比来说要简单许多，React 需要使用 setState 来改变状态，并且使用这个 API 也有一些坑点。并且 Vue 的底层使用了依赖追踪，页面更新渲染已经是最优的了，但是 React 还是需要用户手动去优化这方面的问题。
- React 16以后，有些钩子函数会执行多次，这是因为引入 Fiber 的原因
- React 需要使用 JSX，有一定的上手成本，并且需要一整套的工具链支持，但是完全可以通过 JS 来控制页面，更加的灵活。Vue 使用了模板语法，相比于 JSX 来说没有那么灵活，但是完全可以脱离工具链，通过直接编写 render 函数就能在浏览器中运行。
- 在生态上来说，两者其实没多大的差距，当然 React的用户是远远高于Vue 的

28. Vue常考知识点

28.1 生命周期钩子

- 在beforeCreate钩子函数调用的时候，获取不到props和data中的数据，因为这些数据的初始化在initState中
- 然后执行created钩子函数，在这一步可以访问到之前不能访问的数据，但是这时组件还没挂载，所以看不到
- 接下来执行beforeMount钩子函数，创建VDOM，之后执行mouted钩子，将VDOM渲染为真实DOM并且渲染数据。组件中如果有子组件的话，会递归挂载子组件，只有当所有子组件全部挂载完毕，才会执行根组件的挂载钩子。
- 接下来数据更新时会调用beforeUpdate和updated
- keep-alive有独有的生命周期，分别为activated和deactivated。用keep-alive包裹的组件在切换时不会进行销毁，而是保存在内存中并执行deactivated钩子函数，命中缓存渲染后会执行activated钩子函数
- 最后执行销毁组件的beforeDestory和destoryed。前者适合移除事件、定时器等等，否则可能引起内存泄露的问题。然后进行一系列的销毁操作，如果有子组件的话，会递归销毁子组件，所有子组件都销毁完毕后才会执行根组件的destroyed钩子函数。

28.2 组件通信

组件通信一般分为：

1. 父子组件通信
2. 兄弟组件通信

3. 跨多层级组件通信

父子组件通信

- 父组件通过props传递数据给子组件，子组件通过emit发送事件传递数据给父组件。这就是典型的单向数据流
- 也可以通过$parent和$children对象来访问组件实例中的方法和数据
- Vue2.3版本以上，可以适用$listeners和.sync。前者会将父组件中的v-on事件监听传递给子组件，子组件可以通过访问$listeners来自定义监听器。.sync属性是个语法糖，可以简单的实现父子通信

```
<!--父组件中-->
<input :value.sync="value" />
<!--以上写法等同于-->
<input :value="value" @update:value="v => value = v"></comp>
<!--子组件中-->
<script>
  this.$emit('update:value', 1)
</script>
```

兄弟组件通信

对于这种情况可以通过查找父组件中的子组件实现，也就是 this.$parent.$children，在 $children 中可以通过组件 name 查询到需要的组件实例，然后进行通信。

跨多层次组件通信

- Vue2.2中提供了API provide和inject

```
// 父组件 A
export default {
  provide: {
    data: 1
  }
}
// 子组件 B
export default {
  inject: ['data'],
  mounted() {
    // 无论跨几层都能获得父组件的 data 属性
    console.log(this.data) // => 1
  }
}
```

28.3 extend能做什么

这个API很少用到，作用是扩展组件生成一个构造器，通常会与$mount一起用

28.4 mixin和mixins的区别

mixin：用于全局混入，会影响到每个组件实例，通常插件都是这样做初始化的

```
Vue.mixin({
    beforeCreate() {
        // ...逻辑
        // 这种方式会影响到每个组件的 beforeCreate 钩子函数
    }
})
```

mixins：

- mixins应该是最常使用的扩展组件的方式了。如果多个组件中有相同的业务逻辑，就可以将这些逻辑剥离出来，通过mixins混入代码，比如上拉下拉加载数据等
- 另外需要注意的是 mixins 混入的钩子函数会先于组件内的钩子函数执行，并且在遇到同名选项的时候也会有选择性的进行合并

28.5 watch和computed的区别

- computed是计算属性，依赖其他属性计算值，并且computed的值有缓存，只有当计算值变化才会返回内容
- watch监听到值的变化就会执行回调，在回调中可以进行一些逻辑操作
  **所以一般来说，需要依赖别的属性动态获得值得时候可以适用computed，对于监听值的变化需要做一些复杂业务逻辑的情况可以适用watch，二者都支持对象的写法

28.6 keep-alive组件有什么作用

- 当需要切换组件，保存组件的状态防止多次渲染，就可以使用keep-alive组件包裹需要保存的组件
- keep-alive的两个生命周期钩子函数activated和deactivated

28.7 v-show和v-if的区别

- v-show只是在display:none和display:block之间切换，无论初始条件是什么都会被渲染出来，后面只需要切换CSS，DOM一直保留。因此，v-show在初始渲染时总的开销大，但是切换开销很小，更适合于频繁切换的场景。
- v-if的属性为false时，组件不会被渲染，为true时，被渲染。在切换时，触发销毁和挂载，所以其在切换时开销更高，适合不经常切换的场景。并且基于 v-if 的这种惰性渲染机制，可以在必要的时候才去渲染组件，减少整个页面的初始渲染开销。

28.8 组件中data什么时候可以使用对象

- 组件复用时，所有组件实例都会共享data，如果data是对象的话，就会造成一个组件修改data，影响到其他所有组件，所以需要将data写成函数，每次用到就调用一次函数获得新的数据
- 当使用new Vue()时，无论data是对象还是函数都是可以的，因为new Vue()的方式是生成一个根组件，该组件不会复用，也就不存在data共享的问题了

28.9 响应式原理

- Vue使用Object.defineProperty()来实现响应式原理，通过这个函数可以监听到set和get事件

```
var data= { name: 'jhx }
observe(data)
let name = data.name // -> get value
data.name = 'yyy' // -> change value

function observe(obj){
    if(!obj || typeof obj !== 'object')
        return
    Object.keys.forEach(key => {
        defineReactive(obj, key, obj[key])
    })
}
function defineReactive(obj, key, val){
    // 递归子属性
    observe(val)
    Object.defineProperty(obj,key,{
        enumerable: true,
        configurable: true,
        get: function reactiveGetter() {
            return val
        }
        set: function reactiveSetter(newVal){
            val = newVal
        }
    })
}
```

以上代码简单实现了set和get事件，但自定义的函数一开始是不会执行的，只有先进行了依赖收集，才能在属性更新时派发更新，所以接下来我们需要先触发依赖收集。以下代码，当需要依赖收集时调用addSub，需要派发更新的时候调用notify

```
class Dep(){
    constructor(){
        this.subs = [];
    }
    addSub(sub){
        this.subs.push(sub);
    }
    notify(){
        this.subs.forEach(sub=>{
            sub.update()
        })
    }
}
// 全局属性，通过该属性配置 Watcher
Dep.target = null
```

> 以下是简单的Vue的组件挂载时添加响应式的过程。在组件挂载时，会先对所有需要的属性用Object.defineProperty()，然后实例化Watcher，传入组件更新的回调。在实例化过程中，会对模板中的属性进行求值，触发依赖收集。

以下是简单的触发依赖收集时的操作

```
class Watcher {
  constructor(obj, key, cb) {
    // 将 Dep.target 指向自己
    // 然后触发属性的 getter 添加监听
    // 最后将 Dep.target 置空
    Dep.target = this
    this.cb = cb
    this.obj = obj
    this.key = key
    this.value = obj[key]
    Dep.target = null
  }
  update() {
    // 获得新值
    this.value = this.obj[this.key]
    // 调用 update 方法更新 Dom
    this.cb(this.value)
  }
}
```

以上就是Watcher的简单实现，在执行构造函数的时候将Dep.target指向自身，从而使得收集到了对应的Watcher，在派发更新的时候取出对应的Watcher然后执行update函数

所以，在defineReactive函数中增加依赖收集和派发更新相关的代码

```
function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  let dp = new Dep()
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log('get value')
      // 将 Watcher 添加到订阅
      if (Dep.target) {
        dp.addSub(Dep.target)
      }
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
      // 执行 watcher 的 update 方法
      dp.notify()
    }
  })
}
```

以上所有代码实现了一个简易的数据响应式，核心思路就是手动触发一次属性的getter来实现依赖收集

28.9.1 Object.defineProperty()的缺陷

- 如果通过下标方式修改数组数据或者给对象新增属性并不会触发组件的重新渲染，因为Object.defineProperty()拦截不到大部分的数组操作，Vue内部通过重写函数的方式解决了这个问题

28.9.2 编译过程

模板是怎么在浏览器中运行的?

> Vue会通过编译器将模板通过几个阶段最终编译为render函数，然后通过执行render函数生成VDOM，最终映射为真实DOM

编译的三个阶段：

1. 将模板解析为AST

- 这一阶段主要是通过各种正则表达式去匹配模板中的内容，然后将内容提取出来做各种逻辑操作，接下来会生成一个基本的AST对象，根据对象中属性进一步扩展AST。这个阶段还会进行一些逻辑判断，比如对比前后开闭标签是否一致，判断根组件是否只存在一个，判断是否符合 HTML5 Content Model规范等等问题。

2. 优化AST

- 优化内容包括对静态内容的提取，实现复用VDOM，跳过对比算法。还可以提取静态属性

3. 将AST转换为render函数

- 这一阶段主要目的是遍历整个AST，根据不同的条件生成不同的代码

28.9.3 NextTick原理

什么是NextTick?

- nextTick 可以让我们在下次 DOM 更新循环结束之后执行延迟回调，用于获得更新后的 DOM
  什么时候需要用到Vue.$nextTick()?
- Vue声明周期的created()钩子函数进行的DOM操作一定要放在nextTick中，因为created时，DOM并未渲染。
- 当需要在改变DOM元素的数据后基于新的DOM做点什么，就需要对新DOM的JS操作放到Vue.$nextTick()回调函数中

29. React常考知识点

30.监控

31. TCP/UDP

涉及面试题：TCP和UDP的区别是什么?

UDP

- UDP协议是面向无连接的，不需要在正式传递数据之前连接起双方。它只是报文的搬运工，不保证有序，也不保证不丢失，协议没有控制流量的算法，总体比TCP更加轻便。

特点一:面向无连接

- UDP不需要在发送数据前进行三次握手，可以直接想发就发
- 只是数据报文的搬运工，不对报文进行拆分和拼接

特点二:不可靠性

- 不可靠性体现在无连接上
- 收到什么数据就传递什么数据，不备份，也不确认是否收到
- 没有阻塞控制导致在网络条件差的时候可能丢包，但在一些实时性要求高的场景（电话会议）就需要UDP

特点三:高效

- 不需要像TCP那样复杂
- UDP只在传输层添加一个UDP头，只有八字节，头部开销小，相比TCP至少二十字节要少得多，在传输数据报文时是很高效的

传输方式

- UDP不止支持单播，还支持多播，广播

适用场景

- 实时性要求高的地方
  - 直播
  - 游戏

TCP

- 建立连接断开连接需要先握手，算法会保证传输过程中数据的可靠性，当然问题就是不想UDP那样高效

1. 头部

- sequence number，这个序号保证了报文是有序的，接收端可以通过序号拼接报文
- Acknowledgement Number，这个序号表示数据接收端期望接收的下一个字节的编号是多少，同时也表示上一个序号的数据已经收到
- Window size，窗口大小，表示还能接收多少字节的数据，用于流量控制
- 标识符
  URG=1：该字段为一表示本数据报的数据部分包含紧急信息，是一个高优先级数据报文，此时紧急指针有效。紧急数据一定位于当前数据包数据部分的最前面，紧急指针标明了紧急数据的尾部。
  ACK=1：该字段为一表示确认号字段有效。此外，TCP 还规定在连接建立后传送的所有报文段都必须把 ACK 置为一。
  PSH=1：该字段为一表示接收端应该立即将数据 push 给应用层，而不是等到缓冲区满后再提交。
  RST=1：该字段为一表示当前 TCP 连接出现严重问题，可能需要重新建立 TCP 连接，也可以用于拒绝非法的报文段和拒绝连接请求。
  SYN=1：当SYN=1，ACK=0时，表示当前报文段是一个连接请求报文。当 SYN=1，ACK=1时，表示当前报文段是一个同意建立连接的应答报文。
  FIN=1：该字段为一表示此报文段是一个释放连接的请求报文。

2. 状态机

建立连接三次握手

- TCP建立完成后，不论客户端还是服务端，都可以发送和接收数据，所以TCP是一个全双工的协议
- 起初两端都是CLOSED状态，在通信开始时双方会创建TCB。服务器创建完TCB后，会进入LISTEN状态，此时开始等待客户端发送数据。
  第一次握手
  客户端向服务端发送连接请求报文段。该报文段中包含自身的数据通讯初始序号。请求发送后，客户端进入SYN-SENT状态
  第二次握手
  服务端如果同意连接，就发送一个应答，包含自身的数据通讯初始序号，发送完便进入SYN-RECEIVED状态
  第三次握手
  客户端收到连接同意的应答后，还要向服务端发送一个确认报文，之后进入ESTABLISHED状态，服务端收到这个应答后，也进入ESTABLISHED状态，此时连接建立成功。

涉及面试题：为什么TCP建立连接需要三次握手，明明两次就可以建立起连接

- 可以防止失效的连接请求报文段被服务端接收的情况，从而产生错误

> 在建立连接中，任意一端掉线，TCP都会重发SYN包。

断开连接四次握手
因为TCP是全双工的，在断开连接时，两端都需要发送FIN和ACK
第一次握手：若客户端A认为数据发送完成，则需要向服务端B发送连接释放请求
第二次握手：B 收到连接释放请求后，会告诉应用层要释放 TCP 链接。然后会发送 ACK 包，并进入CLOSE_WAIT状态，此时表明 A 到 B 的连接已经释放，不再接收 A 发的数据了。但是因为 TCP 连接是双向的，所以 B 仍旧可以发送数据给 A
第三次握手：B 如果此时还有没发完的数据会继续发送，完毕后会向 A 发送连接释放请求，然后 B 便进入 LAST-ACK 状态。
PS通过延迟确认的技术（通常有时间限制，否则对方会误认为需要重传），可以将第二次和第三次握手合并，延迟 ACK 包的发送。
第四次握手：A收到释放请求后，向B发送确认应答，此时A进入TIME-WAIT状态，持续2MSL后，保证确认应答不会因为网络问题没有到达，导致B不能正常关闭。若B没有重发请求，就进入CLOSED状态
滑动窗口
滑动窗口实现了TCP的流量控制功能，接收方通过报文告知发送方还可以发送多少数据，从而保证接收方能够来得及接收数据。

> TCP中维护了两个端口：发送端窗口和接收端窗口。
> 发送端窗口包含已发送但未收到应答的数据和可以发送但是未发送的数据
> 发送端窗口是由接收窗口剩余大小决定的，接收方会把当前接收窗口的剩余大小写入应答报文，发送端收到应答后根据该值和当前网络拥塞情况设置发送窗口的大小，所以发送窗口的大小是不断变化的。

32. HTTP/TLS

32.1 HTTP

HTTP请求由三部分构成，分别为：**请求行、首部、实体**

请求行：

如\GET /images/logo.gif HTTP/1.1，由请求方法、URL和协议版本组成

- 请求方法常用的是GET和POST。请求方法虽然多，但更多的是传达一个语义，而不是说POST能做的事情GET就不能做了

涉及面试题：GET和POST的区别

GET多用于无副作用、幂等的场景，如搜索关键字

POST多用于副作用，不幂等的场景，例如注册

- GET请求能缓存，POST不能
- POST 相对 GET 安全一点点，因为GET请求都包含在 URL 里，且会被浏览器保存历史纪录。POST不会，但是在抓包的情况下都是一样的。
- URL有长度限制，会影响GET请求，但这个长度是浏览器规定的
- POST支持更多编码类型且不对数据类型限制

首部：

首部分为请求首部和响应首部，部分首部两种通用

常见状态码

2XX 成功

3XX 重定向

4XX 客户端错误

5XX 服务器错误

32.2 TLS

- HTTPS是通过HTTP传输信息，但是信息通过TLS协议进行了加密
- TLS位于传输层之上，应用层之下。首次进行TLS协议传输需要两个RTT，接下来可以通过Session Resumption减少到一个RTT
- 在TLS中使用了两种加密技术：对称加密和非对称加密

  对称加密
- 两边拥有相同的密钥，双方都知道如何加密解密
- 问题在于，数据传输时，密钥容易被捕获，加密就变得没有意义了

  非对称加密
- 公钥用来加密，私钥用来解密，且私钥只有分发公钥的一方才知道

- 这种加密方式完美解决了对称加密存在的问题。假设两端需要使用对称加密，在这之前，可以通过非对称加密交换密钥

> 流程如下：首先服务器将公钥公布出来，那么客户端就知道了公钥。接下来客户端创建一个密钥，然后通过公钥加密并且发送给服务器，服务器收到密文后通过私钥解密出正确的密钥，这个时候双方就都知道密钥是什么了

TSL握手过程：

1. 客户端发送一个随机值以及需要的协议和加密方式。
2. 服务端收到客户端的随机值，自己也产生一个随机值，并根据客户端需求的协议和加密方式来使用对应的方式，并且发送自己的证书（如果需要验证客户端证书需要说明）
3. 客户端收到服务端的证书并验证是否有效，验证通过会再生成一个随机值，通过服务端证书的公钥去加密这个随机值并发送给服务端，如果服务端需要验证客户端证书的话会附带证书
4. 服务端收到加密过的随机值并使用私钥解密获得第三个随机值，这时候两端都拥有了三个随机值，可以通过这三个随机值按照之前约定的加密方式生成密钥，接下来的通信就可以通过该密钥来加密解密
   通过以上步骤可知，在 TLS握手阶段，两端使用非对称加密的方式来通信，但是因为非对称加密损耗的性能比对称加密大，所以在正式传输数据时，两端使用对称加密的方式通信

33. HTTP2.0

33.1 HTTP/2

- http/1存在队头阻塞，当页面请求的资源达到最大请求数量时，剩余的资源需要等待其他资源请求完成后才能发起请求
- http/2 引入了多路复用的技术，该技术通过一个TCP连接就可以传输所有的请求数据
- 在 HTTP/2 中，有两个非常重要的概念，分别是帧（frame）和流（stream）。帧代表着最小的数据单位，每个帧会标识出该帧属于哪个流，流也就是多个帧组成的数据流。
  多路复用，就是在一个 TCP 连接中可以存在多条流。换句话说，也就是可以发送多个请求，对端可以通过帧中的标识知道属于哪个请求。通过这个技术，可以避免 HTTP 旧版本中的队头阻塞问题，极大的提高传输性能。

33.2 二进制传输

HTTP/2 中所有加强性能的核心点在于此。在之前的 HTTP 版本中，我们是通过文本的方式传输数据。在 HTTP/2 中引入了新的编码机制，所有传输的数据都会被分割，并采用二进制格式编码。

33.3 Header压缩

33.4 服务端Push

34.设计模式

1. 工厂模式

- 工厂起到的作用就是隐藏了创建实例的复杂度，只需要提供一个接口，简单清晰

2. 单例模式

- 单例模式的核心就是保证全局只有一个对象可以访问。比如全局缓存，全局状态管理等等这些只需要一个对象。

```
//在 Vuex 源码中，你也可以看到单例模式的使用，
//虽然它的实现方式不大一样，通过一个外部变量来控制只安装一次 Vuex
let Vue // bind on install=
export function install (_Vue) {
  if (Vue && _Vue === Vue) {
    // 如果发现 Vue 有值，就不重新创建实例了
    return
  }
  Vue = _Vue
  applyMixin(Vue)
}
```

3. 适配器模式

- 适配器用来解决两个接口不兼容的情况，不需要改变现有的接口，通过包装一层的方式实现两个接口正常协作。
- 在Vue中，我们其实经常使用到适配器模式。比如父组件传递给子组件一个时间戳属性，组件内部需要将时间戳转为正常的日期显示，一般会使用 computed 来做转换这件事情，这个过程就使用到了适配器模式

```
class Plug {
  getName() {
    return '港版插头'
  }
}
class Target {
  constructor() {
    this.plug = new Plug()
  }
  getName() {
    return this.plug.getName() + ' 适配器转二脚插头'
  }
}
let target = new Target()
target.getName() // 港版插头 适配器转二脚插头
```

4. 装饰模式

- 装饰模式不需要改变现有的接口，作用是给对象添加功能

5. 代理模式

- 代理就是为了控制对象的访问，不让外部直接访问到对象。比如事件代理

6. 发布-订阅模式

- 也叫做观察者模式。Vue中的响应式就使用了该模式。对于需要实现响应式的对象来说，在get的时候会进行依赖收集，当改变了对象的属性时，就会触发派发更新