2022 年春招大厂前端面试题集合

1. 页面导入样式时，使用 link 和 @import 有什么区别

1. 从属关系区别。@import 只能导入样式表，link 还可以定义 RSS、rel 连接属性、引入网站图标等；
2. 加载顺序区别；加载页面时，link 标签引入的 CSS 被同时加载；@import 引入的 CSS 将在页面加载完毕后被加载；
3. 兼容性区别；

2. 浏览器的渲染原理

1. 首先解析收到的文档，根据文档定义构建一颗 DOM 树，DOM 树是由 DOM 元素及属性节点组成的；
2. 然后对 CSS 进行解析，生成 CSSOM 规则树；
3. 根据 DOM 树和 CSSOM 规则树构建 Render Tree。渲染树的节点被称为渲染对象，渲染对象是一个包含有颜色和大小等属性的矩形，渲染对象和 DOM 对象相对应，但这种对应关系不是一对一的，不可见的 DOM 元素不会被插入渲染树。
4. 当渲染对象被创建并添加到树中，它们并没有位置和大小，所以当浏览器生成渲染树以后，就会根据渲染树来进行布局（也可以叫做回流）。这一阶段浏览器要做的事情就是要弄清楚各个节点在页面中的确切位置和大小。通常这一行为也被称为“自动重排”。
5. 布局阶段结束后是绘制阶段，比那里渲染树并调用对象的 paint 方法将它们的内容显示在屏幕上，绘制使用 UI 基础组件。

为了更好的用户体验，渲染引擎会尽可能早的将内容呈现到屏幕上，并不会等到所有的 html 解析完成之后再去构建和布局 render tree。它是解析完一部分内容就显示一部分内容，同时可能还在网络下载其余内容。

3. 如何实现浏览器内多个标签页之间的通信？

实现多个标签页之间的通信，本质上都是通过中介者模式来实现的。因为标签页之间没有办法直接通信，因此我们可以找一个中介者来让标签页和中介者进行通信，然后让这个中介者来进行消息的转发。

1. 使用 Websocket，通信的标签页连接同一个服务器，发送消息到服务器后，服务器推送消息给所有连接的客户端；
2. 可以地调用 localStorage，localStorage 在另一个浏览上下文里被添加、修改或删除时，它都会触发一个 storage 事件，我们可以通过监听 storage 事件，控制它的值来进行页面信息通信；
3. 如果我们能够获得对应标签页的引用，通过 postMessage 方法也是可以实现多个标签页通信的；

4. 简述前端性能优化

页面内容方面

1. 通过文件合并、css 雪碧图、使用 base64 等方式来减少 HTTP 请求数，避免过多的请求造成等待的情况；
2. 通过 DNS 缓存等机制来减少 DNS 的查询次数；
3. 通过设置缓存策略，对常用不变的资源进行缓存；
4. 通过延迟加载的方式，来减少页面首屏加载时需要请求的资源，延迟加载的资源当用户需要访问时，再去请求加载；
5. 通过用户行为，对某些资源使用预加载的方式，来提高用户需要访问资源时的响应速度；

服务器方面

1. 使用 CDN 服务，来提高用户对于资源请求时的响应速度；
2. 服务器端自用 Gzip、Deflate 等方式对于传输的资源进行压缩，减少传输文件的体积；
3. 尽可能减小 cookie 的大小，并且通过将静态资源分配到其他域名下，来避免对静态资源请求时携带不必要的 cookie；

5. 什么是 webp？

WebP 是谷歌开发的一种新图片格式，它是支持有损和无损两种压缩方式的使用直接色的点阵图。使用 webp 格式的最大优点是是，在相同质量的文件下，它拥有更小的文件体积。因此它非常适合于网络图片的传输，因为图片体积的减少，意味着请求时间的减少，这样会提高用户的体验。这是谷歌开发的一种新的图片格式。

6.浏览器如何判断是否支持 webp 格式图片？

通过创建 Image 对象，将其 src 属性设置为 webp 格式的图片，然后在 onload 事件中获取图片的宽高，如果能够获取，则说明浏览器支持 webp 格式图片。如果不能获取或者触发了 onerror 函数，那么就说明浏览器不支持 webp 格式的图片。

7. 介绍下 BFC 及其应用

BFC（Block Format Context）块级格式化上下文，是页面盒模型中的一种 CSS 渲染模式，相当于一个独立的容器，里面的元素和外部的元素相互不影响。

创建 BFC 的方式有：

1. html 根元素
2. float 浮动

3. 绝对定位

4. overflow 不为 visible

5. display 为表格布局或者弹性布局；

BFC 主要的作用是：

1. 清除浮动

2. 防止同一 BFC 容器中的相邻元素间的外边距重叠问题

8. 怎么让一个 div 水平垂直居中？

```
<div class="parent">
  <div class="child"></div>
</div>
```

```
<!-- 1 -->
div.parent {
  display: flex;
  justify-content: center;
  align-items: center;
}

<!-- 2 -->
div.parent {
  position: relative;
}
div.child {
  position: absolute;
  left: 50%;
  top: 50%;
  transform: translate(-50%, -50%);
}

<!-- 3 -->
div.parent {
  display: grid;
}
div.child {
  justify-self: center;
  align-self: center;
}

<!-- 4 -->
div.parent {
  font-size: 0;
  text-align: center;
  &::before {
    content: "";
    display: inline-block;
    width: 0;
    height: 100%;
    vertical-align: middle;
  }
}
div.child {
  display: inline-block;
  vertical-align: middle;
}
```

9. 介绍下重绘和回流（Repaint & Reflow），以及如何进行优化？

浏览器渲染机制

- 浏览器采用流式布局模型（Flow Based Layout）；
- 浏览器会把 HTML 解析成 DOM，把 CSS 解析成 CSSOM,DOM 和 CSSOM 合并就产生了渲染树（Render Tree）；
- 有了 RenderTree，我们就知道了所有节点的样式，然后计算他们在页面上的大小和位置，最后把节点绘制到页面上；
- 由于浏览器使用流式布局，对 Render Tree 的计算通常只需要遍历一次就可以完成，但 table 及其内部元素除外，他们可能需要多次计算，通常要花 3 倍于同等元素的时间，这也是为什么要避免使用 table 布局的原因之一；

重绘

由于节点的集合属性发生改变或者由于样式改变而不会影响布局的，成为重绘，例如 outline、visibility、color、background-color 等，重绘的代价是高昂的，因此浏览器必须验证 DOM 树上其他节点元素的可见性。

回流

回流是布局或者几何属性需要改变就称为回流。回流是影响浏览器性能的关键因素，因为其变化涉及到部分页面（或是整个页面）的布局更新。一个元素的回流可能会导致其素有子元素以及 DOM 中紧随其后的节点、祖先节点元素的随后的回流。大部分的回流将导致页面的重新渲染。

回流必定会发生重绘，重绘不一定会引发回流。

浏览器优化

现代浏览器大多是通过队列机制来批量更新布局，浏览器会把修改操作放在队列中，至少一个浏览器刷新（即 16.6ms）才会清空队列，但当你获取布局信息的时候，队列中可能会有影响这些属性或方法返回值的操作，即使没有，浏览器也会强制清空队列，触发回流和重绘来确保返回正确的值。

例如 `offsetTop`、`clientTop`、`scrollTop`、`getComputedStyle()`、`width`、`height`、`getBoundingClientRect()`，应避免频繁使用这些属性，他们都会强制渲染刷新队列。

减少重绘和回流

1. CSS

- 使用 `transform` 代替 `top`；
- 使用 `visibility` 替换 `display: none`，前者引起重绘，后者引发回流；
- 避免使用 `table` 布局；
- 尽可能在 `DOM` 树的最末端改变 `class`；
- 避免设置多层内联样式，CSS 选择符从右往左匹配查找，避免节点层级过多；
- 将动画效果应用到 `position` 属性为 `absolute` 或 `fixed` 的元素上，避免影响其他元素的布局；
- 避免使用 `CSS` 表达式，可能会引发回流；
- CSS 硬件加速；

2.Javascript

- 避免频繁操作样式，修改 class 最好；
- 避免频繁操作 DOM，合并多次修改为一次；
- 避免频繁读取会引发回流/重绘的属性，将结果缓存；
- 对具有复杂动画的元素使用绝对定位，使它脱离文档流；

10. 分析比较 opacity: 0、visibility: hidden、display: none 优劣和适用场景

1. display: none - 不占空间，不能点击，会引起回流，子元素不影响
2. visibility: hidden - 占据空间，不能点击，引起重绘，子元素可设置 visible 进行显示
3. opacity: 0 - 占据空间，可以点击，引起重绘，子元素不影响

11. 简述 Rem 及其转换原理

rem 是 CSS3 新增的相对长度单位，是指相对于根元素 html 的 font-size 计算值的大小。

默认根元素的 font-size 都是 16px 的。如果想要设置 12px 的字体大小也就是 12px/16px = 0.75rem。

- 由于 px 是相对固定单位，字号大小直接被定死，无法随着浏览器进行缩放；
- rem 直接相对于根元素 html，避开层级关系，移动端新型浏览器对其支持较好；

个人用 vw + 百分比布局用的比较多，可以使用 webpack 的 postcss-loader 的一个插件 postcss-px-to-viewport 实现对 px 到 vw 的自动转换，非常适合开发。

12. 简述伪类和伪元素

伪类

伪类用于当已有元素处于某种状态时，为其添加对应的样式，这个状态是根据用户行为变化而变化的。比如说 :hover。它只有处于 dom 树无法描述的状态才能为元素添加样式，所以称为伪类。

伪元素

伪元素用于创建一些原本不在文档树中的元素，并为其添加样式，比如说 ::before。虽然用户可以看到这些内容，但是其实他不在文档树中。

区别

伪类的操作对象是文档树中已存在的元素，而伪元素是创建一个文档树外的元素。

css 规范中用双冒号 :: 表示伪元素，用一个冒号 : 表示伪类。

Javascript

13. 什么是防抖和节流？有什么区别？如何实现？

防抖

触发高频事件后 n 秒内函数只会执行一次，如果 n 秒内高频事件再次被触发，则重新计算时间。

```
function debounce(fn, timing) {
  let timer;
  return function() {
    clearTimeout(timer);
    timer = setTimeout(() => {
      fn();
    }, timing);
  }
}
```

节流

高频事件触发，但在 n 秒内只会执行一次，所以节流会稀释函数的执行效率。

```
function throttle(fn, timing) {
  let trigger;
  return function() {
    if (trigger) return;
    trigger = true;
    fn();
    setTimeout(() => {
      trigger = false;
    }, timing);
  }
}
```

Tips：我记这个很容易把两者弄混，总结了个口诀，就是 DTTV（Debounce Timer Throttle Variable - 防抖靠定时器控制，节流靠变量控制）。

14. setTimeout、Promise、Async/Await 的区别

setTimeout: `setTimeout` 的回调函数放到宏任务队列里，等到执行栈清空以后执行；

Promise: `Promise` 本身是同步的立即执行函数，当在 executor 中执行 resolve 或者 reject 的时候，此时是异步操作，会先执行 then/catch 等，当主栈完成时，才会去调用 resolve/reject 方法中存放的方法。

async: `async` 函数返回一个 `Promise` 对象，当函数执行的时候，一旦遇到 `await` 就会先返回，等到触发的异步操作完成，再执行函数体内后面的语句。可以理解为，是让出了线程，跳出了 `async` 函数体。

15. JS 异步解决方案的发展历程以及优缺点。

回调函数

优点：解决了同步的问题（整体任务执行时长）；

缺点：回调地狱，不能用 `try catch` 捕获错误，不能 `return`;

Promise

优点：解决了回调地狱的问题；

缺点：无法取消 Promise，错误需要通过回调函数来捕获；

Generator

特点：可以控制函数的执行。

Async/Await

优点：代码清晰，不用像 Promise 写一大堆 then 链，处理了回调地狱的问题；

缺点：await 将异步代码改造成同步代码，如果多个异步操作没有依赖性而使用 await 会导致性能上的降低；

16. 简单说说 js 中有哪几种内存泄露的情况

1. 意外的全局变量；
2. 闭包；
3. 未被清空的定时器；
4. 未被销毁的事件监听；
5. DOM 引用；

17. Async/Await 如何通过同步的方式实现异步

Async/Await 是一个自执行的 generate 函数。利用 generate 函数的特性把异步的代码写成“同步”的形式。

```
var fetch = require("node-fetch");

function *gen() { // 这里的 * 可以看成 async
  var url = "https://api.github.com/users/github";
  var result = yield fetch(url); // 这里的 yield 可以看成 await
  console.log(result.bio);
}

var g = gen();
var result = g.next();
result.value.then(data => data.json()).then(data  => g.next(data));
```

18. 介绍下 Set、Map、WeakSet 和 WeakMap 的区别？

Set

1. 成员不能重复；
2. 只有键值，没有键名，有点类似数组；
3. 可以遍历，方法有 add、delete、has

WeakSet

1. 成员都是对象（引用）；
2. 成员都是弱引用，随时可以消失（不计入垃圾回收机制）。可以用来保存 DOM 节点，不容易造成内存泄露；
3. 不能遍历，方法有 add、delete、has；

Map

1. 本质上是键值对的集合，类似集合；
2. 可以遍历，方法很多，可以跟各种数据格式转换；

WeakMap

1. 只接收对象为键名（null 除外），不接受其他类型的值作为键名；
2. 键名指向的对象，不计入垃圾回收机制；
3. 不能遍历，方法同 get、set、has、delete；

19. Vue 的响应式原理中 Object.defineProperty 有什么缺陷？为什么在 Vue3.0 采用了 Proxy，抛弃了 Object.defineProperty？

原因如下：

1. Object.defineProperty 无法低耗费的监听到数组下标的变化，导致通过数组下标添加元素，不能实时响应；
2. Object.defineProperty 只能劫持对象的属性，从而需要对每个对象，每个属性进行遍历。如果属性值是对象，还需要深度遍历。 Proxy 可以劫持整个对象， 并返回一个新的对象。
3. Proxy 不仅可以代理对象，还可以代理数组。还可以代理动态增加的属性。

20. 简述浏览器缓存读取规则

浏览器缓存可以优化性能，比如直接使用缓存而不发起请求，或者发起了请求但后端存储的数据和前端一致，则使用缓存从而减少响应数据。

缓存位置

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 Service Worker 的话，传输协议必须为 HTTPS。Service Worker 的缓存与浏览器其他内建的缓存机制不同，它可以让我们自由缓存哪些文件、如何匹配缓存、如何读取缓存，而缓存是可持续性的。Service Worker 也是 PWA 的核心技术。

Memory Cache

Memory Cache 也就是内存中的缓存，主要包含的是当前页面中已经抓取到的资源，例如页面上已经下载的样式、脚本、图片等。读取内存中的数据很高效，但是缓存持续性很短，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。

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 覆盖面基本上是最大的。它会根据 HTTP Header 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。绝大部分的缓存都来自 Disk Cache。

Push Cache

Push Cache（推送缓存）是 HTTP/2 中的内容，当以上三种缓存都没有命中时，它才会被时候用。它只在会话（Session）中存在，一旦会话结束就被释放，并且缓存时间也很短暂（大约 5 分钟）。

缓存过程分析

浏览器与服务器通信的方式为应答模式，即是：浏览器发起 HTTP 请求 - 服务器响应该请求。浏览器第一次向服务器发起该请求后拿到请求结果后，将请求结果和缓存表示存入浏览器缓存，浏览器对于缓存的处理是根据第一次请求资源返回的响应头来确定的。

- 浏览器每次发起请求，都会先在浏览器缓存中查找该请求的结果以及缓存标识；

- 浏览器每次拿到返回的请求结果都会将该结果和缓存表示存入浏览器缓存中；

21. 双向绑定和 vuex 是否冲突

当在严格模式中使用 Vuex 时，在属于 Vuex 的 state 上使用 v-model 会导致出错。

解决方案：

1. 给 <Input> 中绑定 value，然后侦听 input 或者 change 事件，在事件回调中调用一个方法；
2. 使用带有 setter 的双向绑定计算属性；

22. Vue 的父组件和子组件生命周期钩子执行顺序是什么

1. 加载渲染过程：父 beforeCreate -> 父 created -> 父 beforeMount -> 子 beforeCreate -> 子 created -> 子 beforeMount -> 子 mounted -> 父 mounted；
2. 子组件更新过程：父 beforeUpdate -> 子 beforeUpdate -> 子 updated -> 父 updated；
3. 父组件更新过程：父 beforeUpdate -> 父 updated；
4. 销毁过程：父 beforeDestroy -> 子 beforeDestroy -> 子 destroyed -> 父 destroyed；

23. 谈谈对 MVC、MVP、MVVM 模式的理解

在开发图形界面应用程序的时候，会把管理用户界面的层次称为 View，应用程序的数据为 Model，Model 提供数据操作的接口，执行相应的业务逻辑。

MVC

MVC 除了把应用程序分为 View、Model 层，还额外的加了一个 Controller 层，它的职责是进行 Model 和 View 之间的协作（路由、输入预处理等）的应由逻辑（application logic）；Model 进行处理业务逻辑。

用户对 View 操作以后，View 捕获到这个操作，会把处理的权利交移给 Controller（Pass calls）；Controller 会对来自 View 数据进行预处理、决定调用哪个 Model 的接口；然后由 Model 执行相关的业务逻辑；当 Model 变更了以后，会通过观察者模式（Observer Pattern）通知 View；View 通过观察者模式收到 Model 变更的消息以后，会向 Model 请求最新的数据，然后重新更新界面。

MVP

和 MVC 模式一样，用户对 View 的操作都会从 View 交易给 Presenter。Presenter 会执行相应的应用程序逻辑，并且会对 Model 进行相应的操作；而这时候 Model 执行业务逻辑以后，也是通过观察者模式把自己变更的消息传递出去，但是是传给 Presenter 而不是 View。Presenter 获取到 Model 变更的消息以后，通过 View 提供的接口更新界面。

MVVM

MVVM 可以看做是一种特殊的 MVP（Passive View）模式，或者说是对 MVP 模式的一种改良。

MVVM 代表的是 Model-View-ViewModel，可以简单把 ViewModel 理解为页面上所显示内容的数据抽象，和 Domain Model 不一样，ViewModel 更适合用来描述 View。 MVVM 的依赖关系和 MVP 依赖关系一致，只不过是把 P 换成了 VM。

MVVM 的调用关系：

MVVM 的调用关系和 MVP 一样。但是，在 ViewModel 当中会有一个叫 Binder，或者是 Data-binding engine 的东西。以前全部由 Presenter 负责的 View 和 Model 之间数据同步操作交由给 Binder 处理。你只需要在 View 的模板语法当中，指令式声明 View 上的显示的内容是和 Model 的哪一块数据绑定的。当 ViewModel 对进行 Model 更新的时候，Binder 会自动把数据更新到 View 上，当用户对 View 进行操作（例如表单输入），Binder 也会自动把数据更新到 Model 上。这种方式称为：Two-way data-binding，双向数据绑定。可以简单而不恰当地理解为一个模板引擎，但是会根据数据变更实时渲染。

24. react 组件的生命周期

初始化阶段

1. constructor(): 用于绑定事件，初始化 state
2. componentWillMount()：组件将要挂载，在 render 之前调用，可以在服务端调用。
3. render()：用作渲染 dom；
4. componentDidMount()：在 render 之后，而且是所有子组件都 render 之后才调用。

更新阶段

1. getDerivedStateFromProps ： getDerivedStateFromProps 会在调用 render 方法之前调用，并且在初始挂载及后续更新时都会被调用。它应返回一个对象来更新 state，如果返回 null 则不更新任何内容；
2. componentWillReceiveProps(nextProps): 在这里可以拿到即将改变的状态，可以在这里通过 setState 方法设置 state

3. shouldComponentUpdate(nextProps, nextState): 他的返回值决定了接下来的声明周期是否会被调用，默认返回 true
4. componentWillUpdate(): 不能在这里改变 state，否则会陷入死循环
5. componentDidUpdate(): 和 componentDidMount()类似，在这里执行 Dom 操作以及发起网络请求

析构阶段

1. componentWillUnmount()：主要执行清除工作，比如取消网络请求，清除事件监听。

25. 简述浏览器与 Node 的事件循环

浏览器

- 宏任务：script 中的代码、setTimeout、setInterval、I/O、UI render；
- 微任务：promise（async/await）、Object.observe、MutationObserver；

Node

- 宏任务：setTimeout、setInterval、setImmediate、script（整体代码）、I/O 操作等；
- 微任务：process.nextTick（与普通微任务有区别，在微任务队列执行之前执行）、new Promise().then(回调) 等

区别

- node 环境下的 setTimeout 定时器会依次一起执行，浏览器是一个一个分开的；

- 浏览器环境下微任务的执行是每个宏任务执行之后，而 node 中微任务会在各个阶段执行，一个阶段结束立刻执行 microTask；

浏览器环境下：

```
while(true){
    宏任务队列.shift()
    微任务队列全部任务()
}
```

Node 环境下：

```
while(true){
    loop.forEach((阶段)=>{
        阶段全部任务()
        nextTick全部任务()
        microTask全部任务()
    })
}
```

26. 什么是 CSRF 攻击？如何防范 CSRF 攻击？

CSRF 攻击指的是跨站请求伪造攻击，攻击者诱导用户进入一个第三方网站，然后该网站向被攻击网站发送跨站请求。如果用户在被攻击网站中保存了登录状态，那么攻击者就可以利用这个登录状态（cookie），绕过后台的用户验证，冒充用户向服务器执行一些操作。

CSRF 攻击的本质是利用了 cookie 会在同源请求中携带发送给服务器的特点，以此来实现用户的冒充。

防护方法：

1. 同源检测，服务器检测请求来源；
2. 使用 token 来进行验证；
3. 设置 cookie 时设置 Samesite，限制 cookie 不能作为被第三方使用；

27. Vue 中 computed 和 watch 的差异？

1. computed 是计算一个新的属性，并将该属性挂载到 Vue 实例上，而 watch 是监听已经存在且已挂载到 Vue 实例上的数据，所以用 watch 同样可以监听 computed 计算属性的变化；

2. computed 本质是一个惰性求值的观察者，具有缓存性，只有当依赖变化后，第一次访问 computed 值，才会计算新的值。而 watch 则是当数据发送变化便会调用执行函数；

3. 从使用场景上来说，computed 适用一个数据被多个数据影响，而 watch 使用一个数据影响多个数据。

28. webpack 中 loader 和 plugin 的区别是什么？

loader：loader 是一个转换器，将 A 文件进行编译成 B 文件，属于单纯的文件转换过程；

plugin：plugin 是一个扩展器，它丰富了 webpack 本身，针对是 loader 结束后，webpack 打包的整个过程，它并不直接操作文件，而是基于事件机制工作，会监听 webpack 打包过程中的某些节点，执行广泛的任务。

29. 简述一下 React 的源码实现

1. React 的实现主要分为 Component 和 Element；

2. Component 属于 React 实例，在创建实例的过程中会在实例中注册 state 和 props 属性，还会依次调用内置的生命周期函数；

3. Component 中有一个 render 函数，render 函数要求返回一个 Element 对象（或 null）；

4. Element 对象分为原生 Element 对象和组件式对象，原生 Element + 组件式对象会被一起解析成虚拟 DOM 树，并且内部使用的 state 和 props 也以 AST 的形式注入到这棵虚拟 DOM 树之中；

5. 在渲染虚拟 DOM 树的前后，会触发 React Component 的一些生命周期钩子函数，比如 `componentWillMount` 和 `componentDidMount`，在虚拟 DOM 树解析完成后将被渲染成真实 DOM 树；

6. 调用 `setState` 时，会调用更新函数更新 `Component` 的 `state`，并且触发内部的一个 `updater`，调用 `render` 生成新的虚拟 DOM 树，利用 diff 算法与旧的虚拟 DOM 树进行比对，比对以后利用最优的方案进行 DOM 节点的更新，这也是 React 单向数据流的原理（与 Vue 的 MVVM 不同之处）。

30. 手写 Promise

```
const PENDING = "pending";
const RESOLVED = "resolved";
const REJECTED = "rejected";

class MyPromise {
  constructor(fn) {
    this.state = PENDING;
    this.resolvedHandlers = [];
    this.rejectedHandlers = [];
    fn(this.resolve.bind(this), this.reject.bind(this));
    return this;
  }

  resolve(props) {
    setTimeout(() => {
      this.state = RESOLVED;
      const resolveHandler = this.resolvedHandlers.shift();
      if (!resolveHandler) return;

      const result = resolveHandler(props);
      if (result && result instanceof MyPromise) {
        result.then(...this.resolvedHandlers);
      }
    });
  }
```

```
  reject(error) {
    setTimeout(() => {
      this.state = REJECTED;
      const rejectHandler = this.rejectedHandlers.shift();
      if (!rejectHandler) return;

      const result = rejectHandler(error);
      if (result && result instanceof MyPromise) {
        result.catch(...this.rejectedHandlers);
      }
    });
  }

  then(...handlers) {
    this.resolvedHandlers = [...this.resolvedHandlers, ...handlers];
    return this;
  }

  catch(...handlers) {
    this.rejectedHandlers = [...this.rejectedHandlers, ...handlers];
    return this;
  }
}
```

```
MyPromise.all = function (promises) {
  return new MyPromise((resolve, reject) => {
    const results = [];
    for (let i = 0; i < promises.length; i++) {
      const promise = promises[i];
      promise.then(res => {
        results.push(res);
        if (results.length === promises.length) resolve(results);
      }).catch(reject);
    }
  });
}

MyPromise.race = function (promises) {
  return new MyPromise((resolve, reject) => {
    for (let i = 0; i < promises.length; i++) {
      const promise = promises[i];
      promise.then(resolve).catch(reject);
    }
  });
}
```

31. 简单讲解一下 http2 的多路复用

在 HTTP/1 中，每次请求都会建立一次 HTTP 连接，也就是我们常说的 3 次握手和 4 次挥手，这个过程在一次请求过程中占用了相当长的时间，即使开启了 Keep-Alive，解决了多次连接的问题，但是依然有两个效率上的问题，一是串行的文件传输，二是连接数过多导致的性能问题。

HTTP/2 的多路复用就是为了解决上述的两个性能问题。

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。

32. 什么是 CDN 服务？

CDN 是一个内容分发网络，通过对源网站资源的缓存，利用本身多台位于不同地域、不同运营商的服务器，向用户提供资源就近访问的功能。也就是说，用户的请求并不是直接发送给源网站，而是发送给 CDN 服务器，由 CDN 服务器

将请求定位到最近的含有该资源的服务器上去请求。这样有利于提高网站的访问速度，同时通过这种方式也减轻了源服务器的访问压力。

CDN 访问过程

1. 用户输入访问的域名,操作系统向 LocalDns 查询域名的 ip 地址.
2. LocalDns 向 ROOT DNS 查询域名的授权服务器(这里假设 LocalDns 缓存过期)
3. ROOT DNS 将域名授权 dns 记录回应给 LocalDns
4. LocalDns 得到域名的授权 dns 记录后,继续向域名授权 dns 查询域名的 ip 地址
5. 域名授权 dns 查询域名记录后(一般是 CNAME)，回应给 LocalDns
6. LocalDns 得到域名记录后,向智能调度 DNS 查询域名的 ip 地址
7. 智能调度 DNS 根据一定的算法和策略(比如静态拓扑，容量等),将最适合的 CDN 节点 ip 地址回应给 LocalDns
8. LocalDns 将得到的域名 ip 地址，回应给 用户端
9. 用户得到域名 ip 地址后，访问站点服务器
10. CDN 节点服务器应答请求，将内容返回给客户端.(缓存服务器一方面在本地进行保存，以备以后使用，二方面把获取的数据返回给客户端，完成数据服务过程)

33. SSL 连接断开后如何恢复？

Session ID

每一次的会话都有一个编号，当对话中断后，下一次重新连接时，只要客户端给出这个编号，服务器如果有这个编号的记录，那么双方就可以继续使用以前的密钥，而不用重新生成一把。

Session Ticket

session ticket 是服务器在上一次对话中发送给客户的，这个 ticket 是加密的，只有服务器可能够解密，里面包含了本次会话的信息，比如对话密钥和加密方法等。这样不管我们的请求是否转移到其他的服务器上，当服务器将 ticket 解密以后，就能够获取上次对话的信息，就不用重新生成对话秘钥了。

34. 简述 HTTP2.0 与 HTTP1.1 相较于之前版本的改进

HTTP2.0

1. HTTP2.0 基本单位为二进制，以往是采用文本形式，健壮性不是很好，现在采用二进制格式，更方便更健壮。
2. HTTP2.0 的多路复用，把多个请求当做多个流，请求响应数据分成多个帧，不同流中的帧交错发送，解决了 TCP 连接数量多，TCP 连接慢，所以对于同一个域名只用创建一个连接就可以了。
3. HTTP2.0 压缩消息头，避免了重复请求头的传输，又减少了传输的大小；
4. HTTP2.0 服务端推送，浏览器发送请求后，服务端会主动发送与这个请求相关的资源，之后浏览器就不用再次发送后续的请求了；
5. HTTP2.0 可以设置请求优先级，可以按照优先级来解决阻塞的问题；

HTTP1.1

1. 缓存处理新增 E-Tag、If-None-Match 之类的缓存来来控制缓存；
2. 长连接，可以在一个 TCP 连接上发送多个请求和响应；

35. 从输入 URL 到页面加载的全过程

1. 浏览器获取用户输入，等待 url 输入完毕，触发 enter 事件；
2. 解析 URL，分析协议头，再分析主机名是域名还是 IP 地址；
3. 如果主机名是域名的话，则发送一个 DNS 查询请求到 DNS 服务器，获得主机 IP 地址；
4. 使用 DNS 获取到主机 IP 地址后，向目的地址发送一个（http/https/protocol）请求，并且在网络套接字上自动添加端口信息（http 80 https 443）；
5. 等待服务器响应结果；
6. 将响应结果(html)经浏览器引擎解析后得到 Render tree，浏览器将 Render tree 进行渲染后显示在显示器中，用户此时可以看到页面被渲染。

36. 简述面向对象的设计原则

1. 依赖倒置原则
    1. 高层模块（稳定）不应该依赖低层模块（变化），两者都应该依赖于抽象（稳定）；
    2. 抽象（稳定）不应该依赖于实现细节（变化），实现细节（变化）应该依赖于抽象（稳定）；
2. 开放封闭原则
    1. 对扩展开放，对更改封闭；
    2. 类模块应该是可扩展的，但是不可修改；
3. 单一职责原则

    1. 一个类应该只有一个引起它变化的原因；
    2. 变化的方向隐含了类的责任；

4. Liskov 替换原则
    1. 子类必须能够替换他们的基类（IS-A）；
    2. 继承表达类型抽象；

5. 接口隔离原则
    1. 不应该强迫客户端使用他们不用的方法；
    2. 接口应该小而完备；

6. 优先使用对象组合，而不是类继承
    1. 类继承通常为“白箱复用”，对象组合通常为“黑箱复用”；
    2. 继承在某种程度上破坏了封装性，子类父类耦合度高；
    3. 而对象组合则只要求被组合的对象具有良好定义的接口，耦合度低；

7. 封装变化点
    1. 使用封装来创建对象之间的分界层，让设计者可以在分界层的一侧进行修改，而不会对另一侧产生不良的影响，从而实现层次间的松耦合；

8. 针对接口编程，而不是针对实现编程
    1. 不将变量类型声明为具体的某个类，而是声明为某个接口；
    2. 客户程序无需获取对象的具体类型，只需要知道对象所具有的接口；
    3. 减少系统中的各部分依赖关系，从而实现“高内聚、低耦合”的类型设计方案；

37. 什么是设计模式？设计模式如何解决复杂问题？

设计模式描述了一个在我们周围不断发生的问题，以及解决该问题方案的核心。有了设计模式，我们就可以一次又一次的使用该方案而不用重复劳动。

设计模式主要通过两个方面来解决复杂问题：

1. 分解：将复杂问题分解成多个简单问题。
2. 抽象：忽略问题的本质细节，去处理泛化和理想化了的对象模型。

38. 什么是白箱复用和黑箱复用？

白箱复用就是 B 类继承 A 类的功能，同时需要了解 A 类的内部细节，从而达到复用的效果，耦合性较强。

在黑箱复用中，B 类只需要关注 A 类所暴露的一些外部方法即可达到复用的效果，达到了解耦的效果。

39. 请介绍一下 Node 中的内存泄露问题和解决方案

内存泄露原因

1. 全局变量：全局变量挂在 root 对象上，不会被清除掉；
2. 闭包：如果闭包未释放，就会导致内存泄露；
3. 事件监听：对同一个事件重复监听，忘记移除（removeListener），将造成内存泄露。

解决方案

最容易出现也是最难排查的就是事件监听造成的内存泄露，所以事件监听这块需要格外注意小心使用。

如果出现了内存泄露问题，需要检测内存使用情况，对内存泄露的位置进行定位，然后对对应的内存泄露代码进行修复。

40. 请介绍一下 require 的模块加载机制

1. 计算模块绝对路径；
2. 如果缓存中有该模块，则从缓存中取出该模块；
3. 按优先级依次寻找并编译执行模块，将模块推入缓存（require.cache）中；
4. 输出模块的 exports 属性；

41. Node 如何实现热更新？

Node 中有一个 api 是 require.cache，如果这个对象中的引用被清楚后，下次再调用就会重新加载，这个机制可以用来热加载更新的模块。

```
function clearCache(modulePath) {
  const path = require.resolve(modulePath);
  if (require.cache[path]) {
    require.cache[path] = null;
  }
}
```

42. Node 更适合处理 I/O 密集型任务还是 CPU 密集型任务？为什么？

Node 更适合处理 I/O 密集型的任务。因为 Node 的 I/O 密集型任务可以异步调用，利用事件循环的处理能力，资源占用极少，并且事件循环能力避开了多线程的调用，在调用方面是单线程，内部处理其实是多线程的。

并且由于 Javascript 是单线程的原因，Node 不适合处理 CPU 密集型的任务，CPU 密集型的任务会导致 CPU 时间片不能释放，使得后续 I/O 无法发起，从而造成阻塞。但是可以利用到多进程的特点完成对一些 CPU 密集型任务的处理，

不过由于 Javascript 并不支持多线程，所以在这方面的处理能力会弱于其他多线程语言（例如 Java、Go）。

43. 为什么 Node. js 不给每一个. js 文件以独立的上下文来避免作用域被污染?

Nodejs 模块正常情况下对作用域不会造成污染（模块函数内执行），意外创建全局变量是一种例外，可以采用严格模式来避免。

```
function fn() {
  a = 1;
}

fn();

var b = 10;

console.log(a); // 1
console.log(this.a); // undefined
console.log(global.a); // 1 意外的全局上下文污染

console.log(b); // 10
console.log(this.b); // undefined
console.log(global.b); // undefined
```

44. 聊一聊 Node 的垃圾回收机制

Node 的 Javascript 脚本引擎是 Chrome 的 V8 引擎，所以垃圾回收机制也属于 V8 的内部垃圾回收机制。

V8 的垃圾回收机制根据对象的存活时间采用了不同的算法，使得垃圾回收变得更高效。

在 V8 中，内存分为新生代与老生代。

对于新生代的内存采取的是将内存区一分为二，将存活的对象从一个区复制到另一个区，然后对原有的区进行内存释放，反复如此。当一个对象经过多次复制依然存活时，这个较长生命周期的对象会被移动到老生代中。

对于老生代的垃圾回收采用的是标记清除算法，遍历所有对象并标记仍然存在的对象，然后在清除阶段将没有标记的对象进行清除，最后将清除后的空间进行内存释放。

45. 父进程或子进程的死亡是否会影响对方？什么是孤儿进程?

子进程死亡不会影响父进程，不过子进程死亡时，会向它的父进程发送死亡信号。反之父进程死亡，一般情况下子进程也会随之死亡，但如果此时子进程处于可运行状态、僵死状态等等的话，子进程将被 `init` 进程手痒，从而成为孤儿进程。

另外，子进程死亡的时候（处于"终止状态"），父进程没有及时调用 `wait()` 或 `waitpid()` 来返回死亡进程的相关信息，此时子进程还有一个 `PCB` 残留在进程表中，被成为僵尸进程。

46. console.log 是同步还是异步？如何实现一个 console.log?

`console.log` 内部实现是 `process.stdout`，将输入的内容打印到 `stdout`，异步同步取决于 `stdout` 连接的数据流的类型（需要写入的位置）以及不同的操作系统。

- 文件：在 Windows 和 POSIX 上是同步的；
- TTY（终端）：在 Windows 上是异步的，在 POSIX 上是同步；
- 管道（和 socket）：在 Windows 上是同步的，在 POSIX 上是异步的；

造成这种差异的原因是因为一些历史遗留问题，不过这个问题并不会影响正常的输出结果。

47. 什么是守护进程？Node 如何实现守护进程？

守护进程是不依赖终端（tty）的进程，不会因为用户退出终端而停止运行的进程。

Node 实现守护进程的思路：

1. 创建一个进程 A；
2. 在进程 A 中创建进程 B，可以使用 `child_process.fork` 或者其他方法；
3. 启动子进程时，设置 `detached` 属性为 true，保证子进程在父进程退出后继续运行；
4. 进程 A 退出，进程 B 由 init 进程接管。此时进程 B 为守护进程。

48. 什么是粘包问题，如何解决？

默认情况下，TCP 连接会采用延迟传送算法（Nagle 算法），在数据发送之前缓存他们。如果短时间有多个数据发送，会缓冲到一起作一次发送（缓冲大小是 `socket.bufferSize`），这样可以减少 IO 消耗提高性能。（TCP 会出现这个问题，HTTP 协议解决了这个问题）

解决方法

1. 多次发送之前间隔一个等待时间：处理简单，但是影响传输效率；
2. 关闭 Nagle 算法：消耗资源高，整体性能下降；
3. 封包/拆包：使用一些有标识来进行封包拆包（类似 HTTP 协议头尾）；

49. 消息队列的应用场景有哪些？

消息队列的应用场景主要有四个：异步处理、应用解耦、流量削锋和消息通讯。

异步处理：引入消息队列，将不是必须的业务逻辑，推入消息队列做异步处理，从而提高系统并发量与吞吐量。

应用解耦：当两个系统出现强耦合时，可以引入消息队列将两个系统进行解耦，比如订单系统与库存系统。

流量削锋：流量削锋也是消息队列中的常用场景，一般在秒杀和团抢活动中使用广泛！用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面。

日志处理：日志采集客户端，负责日志数据采集，定时写入 Kafka 队列；Kafka 消息队列：负责日志数据的接收、存储和转发；日志处理应用：订阅并消费 kafka 队列中的日志数据。