前端面试题

1、写一个 mySetInterVal(fn, a, b),每次间隔 a,a+b,a+2b 的时间，然后写一个 myClear，停止上面的 mySetInterVal

公司：头条　　分类：JavaScript

```
function mySetInterVal(fn, a, b) {
    this.a = a;
    this.b = b;
    this.time = 0;
    this.handle = -1;
    this.start = () => {
        this.handle = setTimeout(() => {
            fn();
            this.time++;
            this.start();
            console.log( this.a + this.time * this.b);
        }, this.a + this.time * this.b);
    }

    this.stop = () => {
        clearTimeout(this.handle);
        this.time = 0;
    }
}

var a = new mySetInterVal(() => {console.log('123')},1000, 2000 );
a.start();
a.stop();
```

2、合并二维有序数组成一维有序数组，归并排序的思路

公司：头条　　分类：算法

```
/**
 * 解题思路:
 * 双指针 从头到尾比较 两个数组的第一个值，根据值的大小依次插入到新的数组中
 * 空间复杂度: O(m + n)
 * 时间复杂度: O(m + n)
 * @param {Array} arr1
 * @param {Array} arr2
 */
```

```
function  merge(arr1, arr2){
    var  result=[];
    while(arr1.length>0 && arr2.length>0){
        if(arr1[0]<arr2[0]){
              /*shift()方法用于把数组的第一个元素从其中删除，并返回第一个元素的值。*/
            result.push(arr1.shift());
        }else{
            result.push(arr2.shift());
        }
    }
    return  result.concat(arr1).concat(arr2);
}

function mergeSort(arr){
    let lengthArr = arr.length;
    if(lengthArr === 0){
     return [];
    }
    while(arr.length > 1){
     let arrayItem1 = arr.shift();
     let arrayItem2 = arr.shift();
     let mergeArr = merge(arrayItem1, arrayItem2);
     arr.push(mergeArr);
    }
    return arr[0];
}
let arr1 = [[1,2,3],[4,5,6],[7,8,9],[1,2,3],[4,5,6]];
let arr2 = [[1,4,6],[7,8,10],[2,6,9],[3,7,13],[1,5,12]];
mergeSort(arr1);
mergeSort(arr2);
```

3、斐波那契数列

公司：**腾讯、CVTE、微软**　　分类：**算法**

```
//基础版斐波那契数列

function fabonacci(n) {
    let num1 = 1,
        num2 = 1,
        sum;
    let arr = [1, 1];
    for (let i = 3; i <= n; i++) {
        sum = num1 + num2;
        num1 = num2;
        num2 = sum;
        arr.push(sum);
    }
    return arr;
}
```

```
//基于ES6 Generator实现
function* fabonacciG(num) {
    var a = 1,
        b = 1,
        n = 0;
    while (n < num) {
        yield a;
        [a, b] = [b, a + b];
        n++
    }
}
```

4、字符串出现的不重复最长长度

公司：**腾讯**　　分类：**算法**

```
/**
 * 解题思路：
 * 用一个滑动窗口装没有重复的字符，枚举字符记录最大值即可
 * 对于遇到重复字符如何收缩窗口大小?
 * 我们可以用 map 维护字符的索引，遇到相同的字符，把左边界移动过去即可
 * 挪动的过程中记录最大长度
 */
var lengthOfLongestSubstring = function (s) {
    let map = new Map();
    let i = -1
    let res = 0
    let n = s.length
    for (let j = 0; j < n; j++) {
        if (map.has(s[j])) {
            i = Math.max(i, map.get(s[j]))
        }
        res = Math.max(res, j - i)
        map.set(s[j], j)
    }
    return res
};
```

5、React 项目中有哪些细节可以优化？实际开发中都做过哪些性能优化?

公司：**滴滴、掌门一对一、网易、有赞、沪江、喜马拉雅、酷家乐、快手**　　分类：**React**

对于正常的项目优化，一般都涉及到几个方面，开发过程中、上线之后的首屏、运行过程的状态

■ 来聊聊上线之后的首屏及运行状态：

◇ 首屏优化一般涉及到几个指标FP、FCP、FMP；要有一个良好的体验是尽可能的把FCP提前，需要做一些工程化的处理，去优化资源的加载

◇ 方式及分包策略，资源的减少是最有效的加快首屏打开的方式；

◇ 对于CSR的应用，FCP的过程一般是首先加载js与css资源，js在本地执行完成，然后加载数据回来，做内容初始化渲染，这中间就有几次的网络反复请求的过程；所以CSR可以考虑使用骨架屏及预渲染（部分结构预渲染）、suspence与lazy做懒加载动态组件的方式

◇ 当然还有另外一种方式就是SSR的方式，SSR对于首屏的优化有一定的优势，但是这种瓶颈一般在Node服务端的处理，建议使用stream流的方式来处理，对于体验与node端的内存管理等，都有优势；

◇ 不管对于CSR或者SSR，都建议配合使用Service worker，来控制资源的调配及骨架屏秒开的体验

◇ react项目上线之后，首先需要保障的是可用性，所以可以通过React.Profiler分析组件的渲染次数及耗时的一些任务，但是Profile记录的是commit阶段的数据，所以对于react的调和阶段就需要结合performance API一起分析；

◇ 由于React是父级props改变之后，所有与props不相关子组件在没有添加条件控制的情况之下，也会触发render渲染，这是没有必要的，可以结合React的PureComponent以及React.memo等做浅比较处理，这中间有涉及到不可变数据的处理，当然也可以结合使用ShouldComponentUpdate做深比较处理；

◇ 所有的运行状态优化，都是减少不必要的render，React.useMemo与React.useCallback也是可以做很多优化的地方；

◇ 在很多应用中，都会涉及到使用redux以及使用context，这两个都可能造成许多不必要的render，所以在使用的时候，也需要谨慎的处理一些数据；

◇ 最后就是保证整个应用的可用性，为组件创建错误边界，可以使用componentDidCatch来处理；

■ 实际项目中开发过程中还有很多其他的优化点：

◇ 保证数据的不可变性

◇ 使用唯一的键值迭代

◇ 使用web worker做密集型的任务处理

◇ 不在render中处理数据

◇ 不必要的标签，使用React.Fragments

6、react 最新版本解决了什么问题 加了哪些东西?

公司：滴滴　　分类：React

1）React 16.x的三大新特性 Time Slicing, Suspense，hooks

● Time Slicing（解决CPU速度问题）使得在执行任务的期间可以随时暂停，跑去干别的事情，这个特性使得react能在性能极其差的机器跑时，仍然保持有良好的性能

● Suspense（解决网络IO问题）和lazy配合，实现异步加载组件。 能暂停当前组件的渲染，当完成某件事以后再继续渲染，解决从react出生到现在都存在的「异步副作用」的问题，而且解决得非
的优雅，使用的是「异步但是同步的写法」，我个人认为，这是最好的解决异步问题的方式

● 此外，还提供了一个内置函数 componentDidCatch，当有错误发生时，我们可以友好地展示 fallback 组件；可以捕捉到它的子元素（包括嵌套子元素）抛出的异常；可以复用错误组件。

2）React16.8

● 加入hooks，让React函数式组件更加灵活

● hooks之前，React存在很多问题

- □ 在组件间复用状态逻辑很难
- □ 复杂组件变得难以理解，高阶组件和函数组件的嵌套过深。
- □ class组件的this指向问题
- □ 难以记忆的生命周期

● hooks很好的解决了上述问题，hooks提供了很多方法

- □ useState 返回有状态值，以及更新这个状态值的函数
- □ useEffect 接受包含命令式，可能有副作用代码的函数。

□ useContext 接受上下文对象（从React.createContext返回的值）并返回当前上下文值，
□ useReducer useState的替代方案。接受类型为(state, action) => newState的reducer，并返回与dispatch方法配对的当前状态。
□ useCallback 返回一个回忆的memoized版本，该版本仅在其中一个输入发生更改时才会更改。纯函数的输入输出确定性
□ useMemo 纯的一个记忆函数
□ useRef 返回一个可变的ref对象，其.current属性被初始化为传递的参数，返回的 ref 对象在组件的整个生命周期内保持不变。
□ useImperativeMethods 自定义使用ref时公开给父组件的实例值
□ useMutationEffect 更新兄弟组件之前，它在React执行其DOM改变的同一阶段同步触发
□ useLayoutEffect DOM改变后同步触发。使用它来从DOM读取布局并同步重新渲染

3）React16.9

- 重命名 Unsafe 的生命周期方法。新的 UNSAFE_ 前缀将有助于在代码 review 和 debug 期间，使这些有问题的字样更突出
- 废弃 javascript: 形式的 URL。以 javascript: 开头的 URL 非常容易遭受攻击，造成安全漏洞。
- 废弃 “Factory” 组件。 工厂组件会导致 React 变大且变慢。
- act() 也支持异步函数，并且你可以在调用它时使用 await。
- 使用 <React.Profiler> 进行性能评估。 在较大的应用中追踪性能回归可能会很方便

4）React16.13.0

- 支持在渲染期间调用setState，但仅适用于同一组件
- 可检测冲突的样式规则并记录警告
- 废弃unstable_createPortal，使用createPortal
- 将组件堆栈添加到其开发警告中，使开发人员能够隔离bug并调试其程序，这可以清楚地说明问题所在，并更快地定位和修复错误。

7、说一下 Http 缓存策略，有什么区别，分别解决了什么问题?

公司：**滴滴、头条、网易、易车、脉脉、掌门一对一、虎扑、挖财、爱范儿**　　分类：**网路&安全**

1）浏览器缓存策略

浏览器每次发起请求时，先在本地缓存中查找结果以及缓存标识，根据缓存标识来判断是否使用本地缓存。如果缓存有效，则使用本地缓存；否则，则向服务器发起请求并携带缓存标识。
1> 根据是否需向服务器发起HTTP请求，将缓存过程划分为两个部分：强制缓存和协商缓存，强缓优先于协商缓存。
- 强缓存，服务器通知浏览器一个缓存时间，在缓存时间内，下次请求，直接用缓存，不在时间内，执行比较缓存策略。
- 协商缓存，让客户端与服务器之间能实现缓存文件是否更新的验证、提升缓存的复用率，将缓存信息中的Etag和Last-Modified通过请求发送给服务器，由服务器校验，返回304状态码时，浏览器直接使用缓存。

2> HTTP缓存都是从第二次请求开始的：
- 第一次请求资源时，服务器返回资源，并在response header中回传资源的缓存策略；
- 第二次请求时，浏览器判断这些请求参数，击中强缓存就直接200，否则就把请求参数加到request header头中传给服务器，看是否击中协商缓存，击中则返回304，否则服务器会返回新的资源。

2）强缓存

- 强缓存命中则直接读取浏览器本地的资源，在network中显示的是from memory或者from disk
- 控制强制缓存的字段有：Cache-Control（http1.1）和Expires（http1.0）
- Cache-control是一个相对时间，用以表达自上次请求正确的资源之后的多少秒的时间段内缓存有效。
- Expires是一个绝对时间。用以表达在这个时间点之前发起请求可以直接从浏览器中读取数据，而无需发起请求
- Cache-Control的优先级比Expires的优先级高。前者的出现是为了解决Expires在浏览器时间被手动更改导致缓存判断错误的问题。如果同时存在则使用Cache-control。

3）强缓存-expires

- 该字段是服务器响应消息头字段，告诉浏览器在过期时间之前可以直接从浏览器缓存中存取数据。
- Expires 是 HTTP 1.0 的字段，表示缓存到期时间，是一个绝对的时间（当前时间+缓存时间）。在响应消息头中，设置这个字段之后，就可以告诉浏览器，在未过期之前不需要再次请求。
- 由于是绝对时间，用户可能会将客户端本地的时间进行修改，而导致浏览器判断缓存失效，重新请求该资源。此外，即使不考虑修改，时差或者误差等因素也可能造成客户端与服务端的时间不一致，致使缓存失效。

*** 优势特点
1、HTTP 1.0 产物，可以在HTTP 1.0和1.1中使用，简单易用。
2、以时刻标识失效时间。
*** 劣势问题
1、时间是由服务器发送的(UTC)，如果服务器时间和客户端时间存在不一致，可能会出现问题。
2、存在版本问题，到期之前的修改客户端是不可知的。

4）强缓存-cache-control

■ 已知Expires的缺点之后，在HTTP/1.1中，增加了一个字段Cache-control，该字段表示资源缓存的最大有效时间，在该时间内，客户端不需要向服务器发送请求。
■ 这两者的区别就是前者是绝对时间，而后者是相对时间。下面列举一些 Cache-control 字段常用的值：（完整的列表可以查看MDN）
- max-age：即最大有效时间。
- must-revalidate：如果超过了 max-age 的时间，浏览器必须向服务器发送请求，验证资源是否还有效。
- no-cache：不使用强缓存，需要与服务器验证缓存是否新鲜。
- no-store：真正意义上的“不要缓存”。所有内容都不走缓存，包括强制和对比。
- public：所有的内容都可以被缓存（包括客户端和代理服务器， 如 CDN）
- private：所有的内容只有客户端才可以缓存，代理服务器不能缓存。默认值。
- Cache-control 的优先级高于 Expires，为了兼容 HTTP/1.0 和 HTTP/1.1，实际项目中两个字段都可以设置。 ■ 该字段可以在请求头或者响应头设置，可组合使用多种指令：
- 可缓存性：
    - public：浏览器和缓存服务器都可以缓存页面信息
    - private：default，代理服务器不可缓存，只能被单个用户缓存
    - no-cache：浏览器器和服务器都不应该缓存页面信息，但仍可缓存，只是在缓存前需要向服务器确认资源是否被更改。可配合private，
    - 过期时间设置为过去时间。
    - only-if-cache：客户端只接受已缓存的响应
- 到期
    - max-age=：缓存存储的最大周期，超过这个周期被认为过期。
    - s-maxage=：设置共享缓存，比如can。会覆盖max-age和expires。
    - max-stale[=]：客户端愿意接收一个已经过期的资源

        - min-fresh=：客户端希望在指定的时间内获取最新的响应
        - stale-while-revalidate=：客户端愿意接收陈旧的响应，并且在后台一部检查新的响应。时间代表客户端愿意接收陈旧响应的时间长度。
        - stale-if-error=：如新的检测失败，客户端则愿意接收陈旧的响应，时间代表等待时间。
    - 重新验证和重新加载
        - must-revalidate：如页面过期，则去服务器进行获取。
        - proxy-revalidate：用于共享缓存。
        - immutable：响应正文不随时间改变。
    - 其他
        - no-store：绝对禁止缓存
        - no-transform：不得对资源进行转换和转变。例如，不得对图像格式进行转换。
- 优势特点
    1> HTTP 1.1 产物，以时间间隔标识失效时间，解决了Expires服务器和客户端相对时间的问题。
    2> 比Expires多了很多选项设置。
- 劣势问题
    1> 存在版本问题，到期之前的修改客户端是不可知的。

5）协商缓存

- 协商缓存的状态码由服务器决策返回200或者304
- 当浏览器的强缓存失效的时候或者请求头中设置了不走强缓存，并且在请求头中设置了If-Modified-Since 或者 If-None-Match 的时候，会将这两个属性值到服务端去验证是否命中协商缓存，如果命中了协商缓存，会返回 304 状态，加载浏览器缓存，并且响应头会设置 Last-Modified 或者 ETag 属性。
- 对比缓存在请求数上和没有缓存是一致的，但如果是 304 的话，返回的仅仅是一个状态码而已，并没有实际的文件内容，因此 在响应体体积上的节省是它的优化点。
- 协商缓存有 2 组字段(不是两个)，控制协商缓存的字段有：Last-Modified/If-Modified-since（http1.0）和Etag/If-None-match（http1.1）
- Last-Modified/If-Modified-since表示的是服务器的资源最后一次修改的时间；Etag/If-None-match表示的是服务器资源的唯一标识，只要资源变化，Etag就会重新生成。
- Etag/If-None-match的优先级比Last-Modified/If-Modified-since高。

6）协商缓存-协商缓存-Last-Modified/If-Modified-since

■ 服务器通过 Last-Modified 字段告知客户端，资源最后一次被修改的时间，例如 Last-Modified: Mon, 10 Nov 2018 09:10:11 GMT
■ 浏览器将这个值和内容一起记录在缓存数据库中。
■ 下一次请求相同资源时时，浏览器从自己的缓存中找出“不确定是否过期的”缓存。因此在请求头中将上次的 Last-Modified 的值写入到请求头的 If-Modified-Since 字段
■ 服务器会将 If-Modified-Since 的值与 Last-Modified 字段进行对比。如果相等，则表示未修改，响应 304；反之，则表示修改了，响应 200 状态码，并返回数据。
***优势特点
- 不存在版本问题，每次请求都会去服务器进行校验。服务器对比最后修改时间如果相同则返回304，不同返回200以及资源内容。

***劣势问题
- 只要资源修改，无论内容是否发生实质性的变化，都会将该资源返回客户端。例如周期性重写，这种情况下该资源包含的数据实际上一样的。
- 以时刻作为标识，无法识别一秒内进行多次修改的情况。 如果资源更新的速度是秒以下单位，那么该缓存是不能被使用的，因为它的时间单位最低是秒。
- 某些服务器不能精确的得到文件的最后修改时间。
- 如果文件是通过服务器动态生成的，那么该方法的更新时间永远是生成的时间，尽管文件可能没有变化，所以起不到缓存的作用。

7）协商缓存-Etag/If-None-match

■ 为了解决上述问题，出现了一组新的字段 Etag 和 If-None-Match
■ Etag 存储的是文件的特殊标识(一般都是 hash 生成的)，服务器存储着文件的 Etag 字段。之后的流程和 Last-Modified 一致，只是 Last-Modified 字段和它所表示的更新时间改变成了 Etag 字段和它所表示的文件 hash，把 If-Modified-Since 变成了 If-None-Match。服务器同样进行比较，命中返回 304，不命中返回新资源和 200。
■ 浏览器在发起请求时，服务器返回在Response header中返回请求资源的唯一标识。在下一次请求时，会将上一次返回的Etag值赋值给If-No-Matched并添加在Request Header中。服务器将浏览器传来的if-no-matched跟自己的本地的资源的ETag做对比，如果匹配，则返回304通知浏览器读取本地缓存，否则返回200和更新后的资源。
■ Etag 的优先级高于 Last-Modified。
***优势特点
- 可以更加精确的判断资源是否被修改，可以识别一秒内多次修改的情况。
- 不存在版本问题，每次请求都回去服务器进行校验。

***劣势问题
- 计算ETag值需要性能损耗。
- 分布式服务器存储的情况下，计算ETag的算法如果不一样，会导致浏览器从一台服务器上获得页面内容后到另外一台服务器上进行验证时现ETag不匹配的情况。

8、介绍防抖节流原理、区别以及应用，并用JavaScript进行实现

公司：**滴滴、虎扑、挖财、58、头条**　　分类：**JavaScript、编程题**

1）防抖

- 原理：在事件被触发n秒后再执行回调，如果在这n秒内又被触发，则重新计时。
- 适用场景：
  - 按钮提交场景：防止多次提交按钮，只执行最后提交的一次
  - 搜索框联想场景：防止联想发送请求，只发送最后一次输入
- 简易版实现

```
function debounce(func, wait) {
    let timeout;
    return function () {
        const context = this;
        const args = arguments;
        clearTimeout(timeout)
        timeout = setTimeout(function(){
            func.apply(context, args)
        }, wait);
    }
}
```

- 立即执行版实现
    - 有时希望立刻执行函数，然后等到停止触发 n 秒后，才可以重新触发执行。

```
// 有时希望立刻执行函数，然后等到停止触发 n 秒后，才可以重新触发执行。
function debounce(func, wait, immediate) {
  let timeout;
  return function () {
    const context = this;
    const args = arguments;
    if (timeout) clearTimeout(timeout);
    if (immediate) {
      const callNow = !timeout;
      timeout = setTimeout(function () {
        timeout = null;
      }, wait)
      if (callNow) func.apply(context, args)
    } else {
      timeout = setTimeout(function () {
        func.apply(context, args)
      }, wait);
    }
  }
}
```

- 返回值版实现
    - func函数可能会有返回值，所以需要返回函数结果，但是当 immediate 为 false 的时候，因为使用了 setTimeout ，我们将 func.apply(context, args) 的返回值赋给变量，最后再 return 的时候，值将会一直是 undefined，所以只在 immediate 为 true 的时候返回函数的执行结果。

```
function debounce(func, wait, immediate) {
  let timeout, result;
  return function () {
    const context = this;
    const args = arguments;
    if (timeout) clearTimeout(timeout);
    if (immediate) {
      const callNow = !timeout;
      timeout = setTimeout(function () {
        timeout = null;
      }, wait)
      if (callNow) result = func.apply(context, args)
    }
    else {
      timeout = setTimeout(function () {
```

```
        func.apply(context, args)
      }, wait);
    }
    return result;
  }
}
```

2）节流

- 原理：规定在一个单位时间内，只能触发一次函数。如果这个单位时间内触发多次函数，只有一次生效。
- 适用场景
  - 拖拽场景：固定时间内只执行一次，防止超高频次触发位置变动
  - 缩放场景：监控浏览器resize
- 使用时间戳实现
  - 使用时间戳，当触发事件的时候，我们取出当前的时间戳，然后减去之前的时间戳(最一开始值设为 0)，如果大于设置的时间周期，就执行函数，然后更新时间戳为当前的时间戳，如果小于，就不执行。

```
function throttle(func, wait) {
  let context, args;
  let previous = 0;

  return function () {
    let now = +new Date();
    context = this;
    args = arguments;
    if (now - previous > wait) {
      func.apply(context, args);
      previous = now;
    }
  }
}
```

- 使用定时器实现
  - 当触发事件的时候，我们设置一个定时器，再触发事件的时候，如果定时器存在，就不执行，直到定时器执行，然后执行函数，清空定时器，这样就可以设置下个定时器。

```
function throttle(func, wait) {
  let timeout;
  return function () {
    const context = this;
    const args = arguments;
    if (!timeout) {
      timeout = setTimeout(function () {
        timeout = null;
        func.apply(context, args)
      }, wait)
    }

  }
}
```

9、前端安全、中间人攻击

公司：**滴滴**　　分类：**网络&安全**

1）XSS：跨站脚本攻击

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。

1> 存储型（server端）：

- 场景：见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信等。
- 攻击步骤：
  - 攻击者将恶意代码提交到目标网站的数据库中
  - 用户打开目标网站时，服务端将恶意代码从数据库中取出来，拼接在HTML中返回给浏览器
  - 用户浏览器在收到响应后解析执行，混在其中的恶意代码也同时被执行
  - 恶意代码窃取用户数据，并发送到指定攻击者的网站，或者冒充用户行为，调用目标网站的接口，执行恶意操作

2> 反射型（Server端）

与存储型的区别在于，存储型的恶意代码存储在数据库中，反射型的恶意代码在URL上

- 场景：通过 URL 传递参数的功能，如网站搜索、跳转等。
- 攻击步骤：
  - 攻击者构造出特殊的 URL，其中包含恶意代码。
  - 用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器。
  - 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。
  - 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

3> Dom 型(浏览器端)

DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞。

- 场景：通过 URL 传递参数的功能，如网站搜索、跳转等。
- 攻击步骤：
  - 攻击者构造出特殊的 URL，其中包含恶意代码。
  - 用户打开带有恶意代码的 URL。
  - 用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行。
  - 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

4> 预防方案：（防止攻击者提交恶意代码，防止浏览器执行恶意代码）

- 对数据进行严格的输出编码：如HTML元素的编码，JS编码，CSS编码，URL编码等等
  - 避免拼接 HTML；Vue/React 技术栈，避免使用 v-html / dangerouslySetInnerHTML
- CSP HTTP Header，即 Content-Security-Policy、X-XSS-Protection
  - 增加攻击难度，配置CSP(本质是建立白名单，由浏览器进行拦截)
  - Content-Security-Policy: default-src 'self' -所有内容均来自站点的同一个源（不包括其子域名）
  - Content-Security-Policy: default-src 'self' *.trusted.com-允许内容来自信任的域名及其子域名（域名不必须与CSP设置所在的域名相同）
  - Content-Security-Policy: default-src https://yideng.com-该服务器仅允许通过HTTPS方式并仅从yideng.com域名来访问文档
- 输入验证：比如一些常见的数字、URL、电话号码、邮箱地址等等做校验判断

- 开启浏览器XSS防御：Http Only cookie，禁止 JavaScript 读取某些敏感 Cookie，攻击者完成 XSS 注入后也无法窃取此 Cookie。
- 验证码

2）CSRF：跨站请求伪造

攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发送跨站请求。利用受害者在被攻击网站已经获取的注册凭证，绕过后台的用户验证，达到冒充用户对被攻击的网站执行某项操作的目的。

1> 攻击流程举例
- 受害者登录 a.com，并保留了登录凭证（Cookie）
- 攻击者引诱受害者访问了b.com
- b.com 向 a.com 发送了一个请求：a.com/act=xx浏览器会默认携带a.com的Cookie
- a.com接收到请求后，对请求进行验证，并确认是受害者的凭证，误以为是受害者自己发送的请求
- a.com以受害者的名义执行了act=xx
- 攻击完成，攻击者在受害者不知情的情况下，冒充受害者，让a.com执行了自己定义的操作

2> 攻击类型
- GET型：如在页面的某个 img 中发起一个 get 请求
- POST型：通过自动提交表单到恶意网站
- 链接型：需要诱导用户点击链接

3> 预防方案：
CSRF通常从第三方网站发起，被攻击的网站无法防止攻击发生，只能通过增强自己网站针对CSRF的防护能力来提升安全性。）
- 同源检测：通过Header中的Origin Header 、Referer Header 确定，但不同浏览器可能会有不一样的实现，不能完全保证
- CSRF Token 校验：将CSRF Token输出到页面中（通常保存在Session中），页面提交的请求携带这个Token，服务器验证Token是否正确
- 双重cookie验证：
    - 流程：
        - 步骤1：在用户访问网站页面时，向请求域名注入一个Cookie，内容为随机字符串（例如csrfcookie=v8g9e4ksfhw）
        - 步骤2：在前端向后端发起请求时，取出Cookie，并添加到URL的参数中（接上例POST https://www.a.com/comment?csrfcookie=v8g9e4ksfhw）
        - 步骤3：后端接口验证Cookie中的字段与URL参数中的字段是否一致，不一致则拒绝。
    - 优点：
        - 无需使用Session，适用面更广，易于实施。
        - Token储存于客户端中，不会给服务器带来压力。
        - 相对于Token，实施成本更低，可以在前后端统一拦截校验，而不需要一个个接口和页面添加。
    - 缺点：
        - Cookie中增加了额外的字段。
        - 如果有其他漏洞（例如XSS），攻击者可以注入Cookie，那么该防御方式失效。
        - 难以做到子域名的隔离。
        - 为了确保Cookie传输安全，采用这种防御方式的最好确保用整站HTTPS的方式，如果还没切HTTPS的使用这种方式也会有风险。
- Samesite Cookie属性：Google起草了一份草案来改进HTTP协议，那就是为Set-Cookie响应头新增Samesite属性，它用来标明这个 Cookie是个“同站 Cookie”，同站Cookie只能作为第一方Cookie，不能作为第三方Cookie，Samesite 有两个属性值，Strict 为任何情况下都不可以作为第三方 Cookie ，Lax 为可以作为第三方 Cookie ，但必须是Get请求

3）iframe 安全

♣ 说明：
- 嵌入第三方 iframe 会有很多不可控的问题，同时当第三方 iframe 出现问题或是被劫持之后，也会诱发安全性问题
- 点击劫持
  - 攻击者将目标网站通过 iframe 嵌套的方式嵌入自己的网页中，并将 iframe 设置为透明，诱导用户点击。
- 禁止自己的 iframe 中的链接外部网站的JS

♣ 预防方案：
- 为 iframe 设置 sandbox 属性，通过它可以对iframe的行为进行各种限制，充分实现“最小权限“原则
- 服务端设置 X-Frame-Options Header头，拒绝页面被嵌套，X-Frame-Options 是HTTP 响应头中用来告诉浏览器一个页面是否可以嵌入 <iframe> 中
  - eg.X-Frame-Options: SAMEORIGIN
  - SAMEORIGIN: iframe 页面的地址只能为同源域名下的页面
  - ALLOW-FROM: 可以嵌套在指定来源的 iframe 里
  - DENY: 当前页面不能被嵌套在 iframe 里
- 设置 CSP 即 Content-Security-Policy 请求头
- 减少对 iframe 的使用

4）错误的内容推断

♣ 说明：
文件上传类型校验失败后，导致恶意的JS文件上传后，浏览器 Content-Type Header 的默认解析为可执行的 JS 文件
♣ 预防方案：
设置 X-Content-Type-Options 头

5）第三方依赖包

♣ 减少对第三方依赖包的使用，如之前 npm 的包如：event-stream 被爆出恶意攻击数字货币；

6）HTTPS

♣ 描述：
黑客可以利用SSL Stripping这种攻击手段，强制让HTTPS降级回HTTP，从而继续进行中间人攻击。
♣ 预防方案：
使用HSTS（HTTP Strict Transport Security），它通过下面这个HTTP Header以及一个预加载的清单，来告知浏览器和网站进行通信的时候强制性的使用HTTPS，而不是通过明文的HTTP进行通信。这里的“强制性”表现为浏览器无论在何种情况下都直接向务器端发起HTTPS请求，而不再像以往那样从HTTP跳转到HTTPS。另外，当遇到证书或者链接不安全的时候，则首先警告用户，并且不再用户选择是否继续进行不安全的通信。

7）本地存储数据

避免重要的用户信息存在浏览器缓存中

8）静态资源完整性校验

♣ 描述
使用 内容分发网络（CDNs）在多个站点之间共享脚本和样式表等文件可以提高站点性能并节省带宽。然而，使用CDN也存在风险，如果攻击者获得对 CDN 的控制权，则可以将任意恶意内容注入到 CDN 上的文件中 （或完全替换掉文件），因此可能潜在地攻击所有从该 CDN 获取文件的站点。
♣ 预防方案
将使用 base64 编码过后的文件哈希值写入你所引用的 <script> 或 标签的 integrity 属性值中即可启用子资源完整性能。

9）网络劫持

♣ 描述：
- DNS劫持（涉嫌违法）：修改运行商的 DNS 记录，重定向到其他网站。DNS 劫持是违法的行为，目前 DNS 劫持已被监管，现在很少见 DNS 劫持
- HTTP劫持：前提有 HTTP 请求。因 HTTP 是明文传输，运营商便可借机修改 HTTP 响应内容（如加广告）。

♣ 预防方案
全站 HTTPS

10）中间人攻击

中间人攻击（Man-in-the-middle attack，MITM），指攻击者与通讯的两端分别创建独立的联系，并交换其所收到的数据，使通讯的两端认为他们正在通过一个私密的连接与对方直接对话，但事实上整个会话都被攻击者窃听、篡改甚至完全控制。没有进行严格的证书校验是中间人攻击着手点。目前大多数加密协议都提供了一些特殊认证方法以阻止中间人攻击。如 SSL （安全套接字层）协议可以验证参与通讯的用户的证书是否有权威、受信任的数字证书认证机构颁发，并且能执行双向身份认证。攻击场景如用户在一个未加密的 WiFi下访问网站。在中间人攻击中，攻击者可以拦截通讯双方的通话并插入新的内容。

1> 场景
- 在一个未加密的Wi-Fi 无线接入点的接受范围内的中间人攻击者，可以将自己作为一个中间人插入这个网络
- Fiddler / Charles （花瓶）代理工具
- 12306 之前的自己证书

2> 过程
- 客户端发送请求到服务端，请求被中间人截获
- 服务器向客户端发送公钥
- 中间人截获公钥，保留在自己手上。然后自己生成一个【伪造的】公钥，发给客户端
- 客户端收到伪造的公钥后，生成加密hash值发给服务器
- 中间人获得加密hash值，用自己的私钥解密获得真秘钥,同时生成假的加密hash值，发给服务器
- 服务器用私钥解密获得假密钥,然后加密数据传输给客户端

3> 使用抓包工具fiddle来进行举例说明
- 首先通过一些途径在客户端安装证书
- 然后客户端发送连接请求，fiddle在中间截取请求，并返回自己伪造的证书
- 客户端已经安装了攻击者的根证书，所以验证通过
- 客户端就会正常和fiddle进行通信，把fiddle当作正确的服务器

- 同时fiddle会跟原有的服务器进行通信，获取数据以及加密的密钥，去解密密钥

4> 常见攻击方式

- 嗅探是一种用来捕获流进和流出的网络数据包的技术，就好像是监听电话一样。比如：抓包工具
- 数据包注入：在这种，攻击者会将恶意数据包注入到常规数据中的，因为这些恶意数据包是在正常的数据包里面的，用户和系统都很难发现这个内容。
- 会话劫持：当我们进行一个网站的登录的时候到退出登录这个时候，会产生一个会话，这个会话是攻击者用来攻击的首要目标，因为这个会话，包含了用户大量的数据和私密信息。
- SSL剥离：HTTPS是通过SSL/TLS进行加密过的，在SSL剥离攻击中，会使SSL/TLS连接断开，让受保护的HTTPS，变成不受保护的HTTP（这对于网站非常致命）
- DNS欺骗，攻击者往往通过入侵到DNS服务器，或者篡改用户本地hosts文件，然后去劫持用户发送的请求，然后转发到攻击者想要转发到的服务器
- ARP欺骗，ARP(address resolution protocol)地址解析协议，攻击者利用APR的漏洞，用当前局域网之间的一台服务器，来冒充客户端想要请求的服务端，向客户端发送自己的MAC地址，客户端无从得到真正的主机的MAC地址，所以，他会把这个地址当作真正的主机来进行通信，将MAC存入ARP缓存表。
- 代理服务器

预防方案：

- 用可信的第三方CA厂商
- 不下载未知来源的证书，不要去下载一些不安全的文件
- 确认你访问的URL是HTTPS的，确保网站使用了SSL，确保禁用一些不安全的SSL，只开启：TLS1.1，TLS1.2
- 不要使用公用网络发送一些敏感的信息
- 不要去点击一些不安全的连接或者恶意链接或邮件信息

11）sql 注入

1> 描述

就是通过把SQL命令插入到Web表单递交或输入域名或页面请求的查询字符串，最终达到欺骗数据库服务器执行恶意的SQL命令,从而达到和服务器
进行直接的交互

2> 预防方案：

- 后台进行输入验证，对敏感字符过滤。
- 使用参数化查询，能避免拼接SQL，就不要拼接SQL语句。

12）前端数据安全

1> 描述

反爬虫。如猫眼电影、天眼查等等，以数据内容为核心资产的企业

2> 预防方案：

- font-face拼接方式：猫眼电影、天眼查
- background 拼接：美团
- 伪元素隐藏：汽车之家
- 元素定位覆盖式：去哪儿
- iframe 异步加载：网易云音乐

13）其他建议

- 定期请第三方机构做安全性测试，漏洞扫描
- 使用第三方开源库做上线前的安全测试，可以考虑融合到CI中
- code review 保证代码质量
- 默认项目中设置对应的 Header 请求头，如 X-XSS-Protection、 X-Content-Type-Options 、X-Frame-Options Header、Content-Security-Policy 等等
- 对第三方包和库做检测：NSP(Node Security Platform)，Snyk

10、对闭包的看法，为什么要用闭包？说一下闭包原理以及应用场景

公司：**滴滴、携程、喜马拉雅、微医、蘑菇街、酷家乐、腾讯应用宝、安居客**　　分类：**JavaScript**

1）什么是闭包

函数执行后返回结果是一个内部函数，并被外部变量所引用，如果内部函数持有被执行函数作用域的变量，即形成了闭包。

可以在内部函数访问到外部函数作用域。使用闭包，一可以读取函数中的变量，二可以将函数中的变量存储在内存中，保护变量不被污染。而正因闭包会把函数中的变量值存储在内存中，会对内存有消耗，所以不能滥用闭包，否则会影响网页性能，造成内存泄漏。当不需要使用闭包时，要及时释放内存，可将内层函数对象的变量赋值为null。

2）闭包原理

函数执行分成两个阶段(预编译阶段和执行阶段)。

- 在预编译阶段，如果发现内部函数使用了外部函数的变量，则会在内存中创建一个“闭包”对象并保存对应变量值，如果已存在“闭包”，则只需要增加对应属性值即可。
- 执行完后，函数执行上下文会被销毁，函数对“闭包”对象的引用也会被销毁，但其内部函数还持用该“闭包”的引用，所以内部函数可以继续使用“外部函数”中的变量

利用了函数作用域链的特性，一个函数内部定义的函数会将包含外部函数的活动对象添加到它的作用域链中，函数执行完毕，其执行作用域链销毁，但因内部函数的作用域链仍然在引用这个活动对象，所以其活动对象不会被销毁，直到内部函数被烧毁后才被销毁。

3）优点

1> 可以从内部函数访问外部函数的作用域中的变量，且访问到的变量长期驻扎在内存中，可供之后使用
2> 避免变量污染全局
3> 把变量存到独立的作用域，作为私有成员存在

4）缺点

1> 对内存消耗有负面影响。因内部函数保存了对外部变量的引用，导致无法被垃圾回收，增大内存使用量，所以使用不当会2导致内存泄漏
2> 对处理速度具有负面影响。闭包的层级决定了引用的外部变量在查找时经过的作用域链长度
3> 可能获取到意外的值(captured value)

5）应用场景

应用场景一： 典型应用是模块封装，在各模块规范出现之前，都是用这样的方式防止变量污染全局。

```
var Yideng = (function () {
    // 这样声明为模块私有变量，外界无法直接访问
    var foo = 0;

    function Yideng() {}
    Yideng.prototype.bar = function bar() {
        return foo;
    };
    return Yideng;
}());
```

应用场景二： "在循环中创建闭包，防止取到意外的值。"

如下代码，无论哪个元素触发事件，都会弹出 3。因为函数执行后引用的 i 是同一个，而 i 在循环结束后就是 3

```
for (var i = 0; i < 3; i++) {
    document.getElementById('id' + i).onfocus = function() {
      alert(i);
    };
}
//可用闭包解决
function makeCallback(num) {
  return function() {
    alert(num);
  };
}
for (var i = 0; i < 3; i++) {
    document.getElementById('id' + i).onfocus = makeCallback(i);
}
```

11、css 伪类与伪元素区别

公司：滴滴　　分类：CSS

1）伪类(pseudo-classes)

- 其核心就是用来选择DOM树之外的信息，不能够被普通选择器选择的文档之外的元素，用来添加一些选择器的特殊效果。
- 比如:hover :active :visited :link :visited :first-child :focus :lang等
- 由于状态的变化是非静态的，所以元素达到一个特定状态时，它可能得到一个伪类的样式；当状态改变时，它又会失去这个样式。
- 由此可以看出，它的功能和class有些类似，但它是基于文档之外的抽象，所以叫 伪类。

2）伪元素(Pseudo-elements)

- DOM树没有定义的虚拟元素
- 核心就是需要创建通常不存在于文档中的元素，
- 比如::before ::after 它选择的是元素指定内容，表示选择元素内容的之前内容或之后内容。
- 伪元素控制的内容和元素是没有差别的，但是它本身只是基于元素的抽象，并不存在于文档中，所以称为伪元素。

用于将特殊的效果添加到某些选择器

3）伪类与伪元素的区别

- 表示方法
  - CSS2 中伪类、伪元素都是以单冒号:表示，
  - CSS2.1 后规定伪类用单冒号表示,伪元素用双冒号::表示，
  - 浏览器同样接受 CSS2 时代已经存在的伪元素(:before，:after，:first line，:first-letter 等)的单冒号写法。
  - CSS2 之后所有新增的伪元素(如::selection)，应该采用双冒号的写法。
  - CSS3中，伪类与伪元素在语法上也有所区别，伪元素修改为以::开头。浏览器对以:开头的伪元素也继续支持，但建议规范书写为::开头
- 定义不同
  - 伪类即假的类，可以添加类来达到效果
  - 伪元素即假元素，需要通过添加元素才能达到效果
- 总结:
  - 伪类和伪元素都是用来表示文档树以外的"元素"。
  - 伪类和伪元素分别用单冒号:和双冒号::来表示。
  - 伪类和伪元素的区别，关键点在于如果没有伪元素(或伪类)，
  - 是否需要添加元素才能达到效果，如果是则是伪元素，反之则是伪类。

4）相同之处

- 伪类和伪元素都不出现在源文件和DOM树中。也就是说在html源文件中是看不到伪类和伪元素的。

不同之处:

- 伪类其实就是基于普通DOM元素而产生的不同状态，他是DOM元素的某一特征。
- 伪元素能够创建在DOM树中不存在的抽象对象，而且这些抽象对象是能够访问到的。

12、有一堆整数，请把他们分成三份，确保每一份和尽量相等（11，42，23，4，5，6 4 5 6 11 23 42 56 78 90）

公司：**滴滴**　　分类：**算法**

```
"方法一"
function f1 (arr,count){
    //数组从大到小排序
    arr.sort((a,b) => b - a);
    //计算平均值
    let avg = arr.reduce((a,b) => a + b) / count;
    //从大到小求和，取最接近平均值的一组，放入二维数组
    let resArr = [];
    let current = 0;
```

```
    for (let i = 0; i < count-1; i++) {
        if(current + arr[arr.length-1]/2 < avg && i){
            arr.pop();
            resArr[i-1].push(arr[arr.length-1]);
        }
        current = 0;
        resArr[i] = [];
        arr.forEach((item,index) => {
            current += item;
            arr.splice(index,1);
            resArr[i].push(item);
            if(current > avg){
                current -= item;
                arr.splice(index,0,item);
                resArr[i].pop();
            }
        })
    }
    resArr[count-1] = arr;
    return resArr
}
//测试，第一个参数为数组，第二个为份数
f1([11,42,23,4,5,6,4,5,6,11,23,42,56,78,90],3)

"方法二"
function foo(arr) {
        let AMOUNT = arr.length
        if (!AMOUNT) return false;
        if (AMOUNT === 3) return arr;
        arr.sort((a, b) => a - b);
        let total = 0;
        let maxNumberTotal = 0;
        for (let i = 0; i < AMOUNT; i++) {
            total += arr[i];
        }
        maxNumberTotal = total / 3;
        let tempTotal = arr[AMOUNT - 1];

        let firstArr = [arr[AMOUNT - 1]];
        let delIndex = [AMOUNT - 1];
        let firstIndex = -1;

        // 获取第一份数组
        for (let i = AMOUNT - 2; i > 0; i--) {
            const el = arr[i];
            tempTotal += el; // 每次拿最大的;
            firstArr.push(el);
            delIndex.push(i);
            if (tempTotal === maxNumberTotal) { // 刚好等于最大值跳出循环
                break;
            } else if (tempTotal > maxNumberTotal) { // 发现超过最大值，减回去
                tempTotal -= el;
```

```
                delIndex.pop();
                firstArr.pop();
            } else if (tempTotal < maxNumberTotal) { // 发现超过最小值，处理边界问题
                let nextTotal = tempTotal + arr[i + 1]
                if (maxNumberTotal - tempTotal < Math.abs(maxNumberTotal - nextTotal)) { // 当前总值比上一个总值大；这里是临界值，说明上一个总值肯定是一个比最大值大，所以这里需要和绝对值比较
                    if (maxNumberTotal - tempTotal > arr[0]) { // 如果下一个平局值和总值相减，比数组第一个数还大，说明还可以继续走下去；
                        continue;
                    } else {
                        break;
                    }
                }
            }
        }
        for (let i = 0; i < delIndex.length; i++) {
            arr.splice(delIndex[i], 1)
        }

        AMOUNT = arr.length; // 注意每次的arr都是不一样的
        let secondArr = [arr[AMOUNT - 1]];
        delIndex = [AMOUNT - 1];
        let secondIndex = -1;
        tempTotal = arr[AMOUNT - 1];
        // 获取第二份数组
        for (let i = AMOUNT - 2; i > 0; i--) {
            const el = arr[i];
            tempTotal += el; // 每次拿最大的;
            secondArr.push(el);
            delIndex.push(i);
            if (tempTotal === maxNumberTotal) { // 刚好等于最大值跳出循环
                break;
            } else if (tempTotal > maxNumberTotal) { // 发现超过最大值，减回去
                tempTotal -= el;
                delIndex.pop();
                secondArr.pop();
            } else if (tempTotal < maxNumberTotal) { // 发现超过最小值，处理边界问题
                let nextTotal = tempTotal + arr[i + 1]
                if (maxNumberTotal - tempTotal < Math.abs(maxNumberTotal - nextTotal)) { // 当前总值恒小于下一个总值；这里是临界值
                    if (maxNumberTotal - tempTotal > arr[0]) {
                        continue;
                    } else {
                        break;
                    }
                }
            }
        }
        for (let i = 0; i < delIndex.length; i++) {
            arr.splice(delIndex[i], 1)
        }
        // 公平处理，当出现极差情况就需要做公平处理了，这里暂时不考虑极差情况
```

```
        return [firstArr, secondArr, arr]
    }

"方法三"
const numSort = (arr) => {
  return arr.sort((a, b) => b - a)
};
const numArr = numSort([11, 42, 23, 4, 5, 6, 4, 5, 6, 11, 23, 42, 56, 78, 90]);
const sumTotal = eval(numArr.join('+'));
let a = 0, b = 0, c = 0;
const sumFN = (x) => {
  let newX = x;
  numArr.map((num, index) => {
    if ((newX + num) <= Math.ceil(sumTotal / 3)) {
      newX = newX + num;
      numArr.splice(index, 1);
    } else {
      return false;
    }
  })
  return newX
};
a = sumFN(a);
b = sumFN(b);
c = eval(numArr.join('+'));
console.log(a, b, c);
136 136 134

"方法四"
let arr = [11,42,23,4,5,6,4,5,6,11,23,42,56,78,90]
    arr.sort((a,b)=>a-b)
    let arr0 = []
    let arr1 = []
    let arr2 = []
    let getsum = function(arr){
        return arr.reduce((a,b)=>{
        return a+b
    },0)
    }
    let sum = getsum(arr)
    let avg = sum/3
    console.log(avg)
    // 先分大数，大数影响分布均衡
    while(arr.length>0){
      // console.log(arr.length)
      let sum0 = getsum(arr0)
      let sum1 = getsum(arr1)
      let sum2 = getsum(arr2)

      if(Math.min(sum0,sum1,sum2)===sum0 && sum0<=avg){
        let popnum = arr.pop()
        arr0.push(popnum||0)
```

```
      }
      if(Math.min(sum0,sum1,sum2)===sum1 && sum1<=avg){
        let popnum = arr.pop()
        arr1.push(popnum||0)
      }
      if(Math.min(sum0,sum1,sum2)===sum2 && sum2<=avg){
        let popnum = arr.pop()
        arr2.push(popnum||0)
      }
    }
  console.log(arr0,arr1,arr2,getsum(arr0),getsum(arr1),getsum(arr2))
// [90, 23, 11, 6, 5]  [78, 42, 11, 4]  [56, 42, 23, 6, 5, 4] 135 135 136
```

13、实现 lodash 的_.get

公司：滴滴　　分类：JavaScript

在 js 中经常会出现嵌套调用这种情况，如 a.b.c.d.e，但是这么写很容易抛出异常。你需要这么写 a && a.b && a.b.c && a.b.c.d && a.b.c.d.e，但是显得有些啰嗦与冗长了。特别是在 graphql 中，这种嵌套调用更是难以避免。
这时就需要一个 get 函数，使用 get(a, 'b.c.d.e') 简单清晰，并且容错性提高了很多。

1) "代码实现"

```
function get(source, path, defaultValue = undefined) {
  // a[3].b -> a.3.b -> [a,3,b]
 // path 中也可能是数组的路径，全部转化成 . 运算符并组成数组
  const paths = path.replace(/\[(\d+)\]/g, ".$1").split(".");
  let result = source;
  for (const p of paths) {
    // 注意 null 与 undefined 取属性会报错，所以使用 Object 包装一下。
    result = Object(result)[p];
    if (result == undefined) {
      return defaultValue;
    }
  }
  return result;
}
// 测试用例
console.log(get({ a: null }, "a.b.c", 3)); // output: 3
console.log(get({ a: undefined }, "a", 3)); // output: 3
console.log(get({ a: null }, "a", 3)); // output: 3
console.log(get({ a: [{ b: 1 }] }, "a[0].b", 3)); // output: 1
```

2) "代码实现不考虑数组的情况"

```
const _get = (object, keys, val) => {
 return keys.split(/\./).reduce(
  (o, j)=>( (o || {})[j] ),
  object
 ) || val
}
console.log(get({ a: null }, "a.b.c", 3)); // output: 3
```

```
console.log(get({ a: undefined }, "a", 3)); // output: 3
console.log(get({ a: null }, "a", 3)); // output: 3
console.log(get({ a: { b: 1 } }, "a.b", 3)); // output: 1
```

14、实现 add(1)(2)(3)

公司：滴滴　　分类：JavaScript

> "考点：函数柯里化"
> 函数柯里化概念： 柯里化（Currying）是把接受多个参数的函数转变为接受一个单一参数的函数，并且返回接受余下的参数且返回结果的新函数的技术。

```
1) "暴力版"
function add (a) {
  return function (b) {
    return function (c) {
        return a + b + c;
    }
  }
}
console.log(add(1)(2)(3)); // 6

2) "柯里化解决方案"
```

- 参数长度固定

```
const curry = (fn) =>
(judge = (...args) =>
    args.length === fn.length
    ? fn(...args)
    : (...arg) => judge(...args, ...arg));
const add = (a, b, c) => a + b + c;
const curryAdd = curry(add);
console.log(curryAdd(1)(2)(3)); // 6
console.log(curryAdd(1, 2)(3)); // 6
console.log(curryAdd(1)(2, 3)); // 6
```

- 参数长度不固定

```
function add (...args) {
    //求和
    return args.reduce((a, b) => a + b)
}

function currying (fn) {
    let args = []
    return function temp (...newArgs) {
        if (newArgs.length) {
            args = [
                ...args,
                ...newArgs
            ]
            return temp
```

```
        } else {
            let val = fn.apply(this, args)
            args = [] //保证再次调用时清空
            return val
        }
    }
}
let addCurry = currying(add)
console.log(addCurry(1)(2)(3)(4, 5))  //15
console.log(addCurry(1)(2)(3, 4, 5))  //15
console.log(addCurry(1)(2, 3, 4, 5))  //15
```

15、实现链式调用

公司：滴滴　　分类：JavaScript

链式调用的核心就在于调用完的方法将自身实例返回

示例一

```
function Class1() {
    console.log('初始化')
}
Class1.prototype.method = function(param) {
    console.log(param)
    return this
}
let cl = new Class1()
//由于new 在实例化的时候this会指向创建的对象，  所以this.method这个方法会在原型链中找到。
cl.method('第一次调用').method('第二次链式调用').method('第三次链式调用')
```

示例二

```
var obj = {
    a: function() {
        console.log("a");
        return this;
    },
    b: function() {
        console.log("b");
        return this;
    },
};
obj.a().b();
```

示例三

```
// 类
class Math {
    constructor(value) {
        this.hasInit = true;
        this.value = value;
        if (!value) {
            this.value = 0;
            this.hasInit = false;
        }
    }
    add() {
        let args = [...arguments]
        let initValue = this.hasInit ? this.value : args.shift()
        const value = args.reduce((prev, curv) => prev + curv, initValue)
        return new Math(value)
    }
    minus() {
        let args = [...arguments]
        let initValue = this.hasInit ? this.value : args.shift()
        const value = args.reduce((prev, curv) => prev - curv, initValue)
        return new Math(value)
    }
    mul() {
        let args = [...arguments]
        let initValue = this.hasInit ? this.value : args.shift()
        const value = args.reduce((prev, curv) => prev * curv, initValue)
        return new Math(value)
    }
    divide() {
        let args = [...arguments]
        let initValue = this.hasInit ? this.value : args.shift()
        const value = args.reduce((prev, curv) => prev / (+curv ? curv : 1), initValue)
        return new Math(value)
    }
}

let test = new Math()
const res = test.add(222, 333, 444).minus(333, 222).mul(3, 3).divide(2, 3)
console.log(res.value)

// 原型链
Number.prototype.add = function() {
    let _that = this
    _that = [...arguments].reduce((prev, curv) => prev + curv, _that)
    return _that
}
Number.prototype.minus = function() {
    let _that = this
    _that = [...arguments].reduce((prev, curv) => prev - curv, _that)
```

```
    return _that
}
Number.prototype.mul = function() {
    let _that = this
    _that = [...arguments].reduce((prev, curv) => prev * curv, _that)
    return _that
}
Number.prototype.divide = function() {
    let _that = this
    _that = [...arguments].reduce((prev, curv) => prev / (+curv ? curv : 1), _that)
    return _that
}
let num = 0;
let newNum = num.add(222, 333, 444).minus(333, 222).mul(3, 3).divide(2, 3)
console.log(newNum)
```

16、React 事件绑定原理

公司：滴滴、沪江　　分类：React

1）事件注册流程

- 组件装载 / 更新。
- 通过lastProps、nextProps判断是否新增、删除事件分别调用事件注册、卸载方法。
- 调用EventPluginHub的enqueuePutListener进行事件存储
- 获取document对象。
- 根据事件名称（如onClick、onCaptureClick）判断是进行冒泡还是捕获。
- 判断是否存在addEventListener方法，否则使用attachEvent（兼容IE）。
- 给document注册原生事件回调为dispatchEvent（统一的事件分发机制）。

2）事件存储

- EventPluginHub负责管理React合成事件的callback，它将callback存储在listenerBank中，另外还存储了负责合成事件的Plugin。
- EventPluginHub的putListener方法是向存储容器中增加一个listener。
- 获取绑定事件的元素的唯一标识key。
- 将callback根据事件类型，元素的唯一标识key存储在listenerBank中。
- listenerBank的结构是：listenerBank[registrationName][key]。

```
{
    onClick:{
        nodeid1:()=>{...}
        nodeid2:()=>{...}
    },
    onChange:{
        nodeid3:()=>{...}
        nodeid4:()=>{...}
    }
}
```

3）事件触发执行

- 触发document注册原生事件的回调dispatchEvent
- 获取到触发这个事件最深一级的元素
- 这里的事件执行利用了React的批处理机制

代码示例

```
<div onClick={this.parentClick} ref={ref => this.parent = ref}>
      <div onClick={this.childClick} ref={ref => this.child = ref}>
          test
     </div>
</div>
```

- 首先会获取到this.child
- 遍历这个元素的所有父元素，依次对每一级元素进行处理。
- 构造合成事件。
- 将每一级的合成事件存储在eventQueue事件队列中。
- 遍历eventQueue。
- 通过isPropagationStopped判断当前事件是否执行了阻止冒泡方法。
- 如果阻止了冒泡，停止遍历，否则通过executeDispatch执行合成事件。
- 释放处理完成的事件。

4）事件触发执行

- 调用EventPluginHub的extractEvents方法。
- 循环所有类型的EventPlugin（用来处理不同事件的工具方法）。
- 在每个EventPlugin中根据不同的事件类型，返回不同的事件池。
- 在事件池中取出合成事件，如果事件池是空的，那么创建一个新的。
- 根据元素nodeid(唯一标识key)和事件类型从listenerBink中取出回调函数
- 返回带有合成事件参数的回调函数

17、类数组和数组的区别，dom 的类数组如何转换成数组

公司：**海康威视**　　分类：**JavaScript**

1）定义

- 数组是一个特殊对象,与常规对象的区别：
  - 当由新元素添加到列表中时，自动更新length属性
  - 设置length属性，可以截断数组
  - 从Array.protoype中继承了方法
  - 属性为'Array'
- 类数组是一个拥有length属性，并且他属性为非负整数的普通对象，类数组不能直接调用数组方法。

2）区别

本质：类数组是简单对象，它的原型关系与数组不同。

```
// 原型关系和原始值转换
let arrayLike = {
    length: 10,
```

```
};
console.log(arrayLike instanceof Array); // false
console.log(arrayLike.__proto__.constructor === Array); // false
console.log(arrayLike.toString()); // [object Object]
console.log(arrayLike.valueOf()); // {length: 10}

let array = [];
console.log(array instanceof Array); // true
console.log(array.__proto__.constructor === Array); // true
console.log(array.toString()); // ''
console.log(array.valueOf()); // []
```

3）类数组转换为数组

- 转换方法

  - 使用 `Array.from()`
  - 使用 `Array.prototype.slice.call()`
  - 使用 `Array.prototype.forEach()` 进行属性遍历并组成新的数组

- 转换须知
  - 转换后的数组长度由 length 属性决定。索引不连续时转换结果是连续的，会自动补位。

    代码示例

    ```
    let al1 = {
    length: 4,
    0: 0,
    1: 1,
    3: 3,
    4: 4,
    5: 5,
    };
    console.log(Array.from(al1)) // [0, 1, undefined, 3]
    ```

- 仅考虑 0或正整数 的索引

```
let al2 = {
    length: 4,
    '-1': -1,
    '0': 0,
    a: 'a',
    1: 1
};
console.log(Array.from(al2)); // [0, 1, undefined, undefined]
```

- 使用slice转换产生稀疏数组

```
let al2 = {
    length: 4,
    '-1': -1,
    '0': 0,
    a: 'a',
    1: 1
};
console.log(Array.prototype.slice.call(al2)); //[0, 1, empty × 2]
```

4）使用数组方法操作类数组注意地方

```
 let arrayLike2 = {
    2: 3,
    3: 4,
    length: 2,
    push: Array.prototype.push
  }

  // push 操作的是索引值为 length 的位置
  arrayLike2.push(1);
  console.log(arrayLike2); // {2: 1, 3: 4, length: 3, push: ƒ}
  arrayLike2.push(2);
  console.log(arrayLike2); // {2: 1, 3: 2, length: 4, push: ƒ}
```

18、webpack 做过哪些优化，开发效率方面、打包策略方面等等

公司：**滴滴、快手、掌门一对一、高思教育**　　分类：**工程化**

1）优化 Webpack 的构建速度

- 使用高版本的 Webpack （使用webpack4）
- 多线程/多实例构建：HappyPack(不维护了)、thread-loader
- 缩小打包作用域：
  - exclude/include (确定 loader 规则范围)
  - resolve.modules 指明第三方模块的绝对路径 (减少不必要的查找)
  - resolve.extensions 尽可能减少后缀尝试的可能性
  - noParse 对完全不需要解析的库进行忽略 (不去解析但仍会打包到 bundle 中，注意被忽略掉的文件里不应该包含 import、require、define 等模块化语句)
  - IgnorePlugin (完全排除模块)
  - 合理使用alias
- 充分利用缓存提升二次构建速度：
  - babel-loader 开启缓存
  - terser-webpack-plugin 开启缓存
  - 使用 cache-loader 或者 hard-source-webpack-plugin
    注意：thread-loader 和 cache-loader 兩個要一起使用的話，請先放 cache-loader 接著是 thread-loader 最後才是 heavy-loader
- DLL：

- 使用 DllPlugin 进行分包，使用 DllReferencePlugin(索引链接) 对 manifest.json 引用，让一些基本不会改动的代码先打包成静态资源，避免反复编译浪费时间。

2）使用webpack4-优化原因

- V8带来的优化（for of替代forEach、Map和Set替代Object、includes替代indexOf）
- 默认使用更快的md4 hash算法
- webpacks AST可以直接从loader传递给AST，减少解析时间
- 使用字符串方法替代正则表达式

① noParse

- 不去解析某个库内部的依赖关系
- 比如jquery 这个库是独立的， 则不去解析这个库内部依赖的其他的东西
- 在独立库的时候可以使用

```
module.exports = {
  module: {
    noParse: /jquery/,
    rules:[]
  }
}
```

② IgnorePlugin

- 忽略掉某些内容 不去解析依赖库内部引用的某些内容
- 从moment中引用 ./locol 则忽略掉
- 如果要用local的话 则必须在项目中必须手动引入 **import 'moment/locale/zh-cn'**

```
module.exports = {
plugins: [
new Webpack.IgnorePlugin(/./local/, /moment/),
]
}
```

③ dillPlugin

- 不会多次打包， 优化打包时间
- 先把依赖的不变的库打包
- 生成 manifest.json文件
- 然后在webpack.config中引入
- webpack.DllPlugin Webpack.DllReferencePlugin

④ happypack -> thread-loader

- 大项目的时候开启多线程打包
- 影响前端发布速度的有两个方面，一个是构建，一个就是压缩，把这两个东西优化起来，可以减少很多发布的时间。

⑤ thread-loader

thread-loader 会将您的 loader 放置在一个 worker 池里面运行，以达到多线程构建。
把这个 loader 放置在其他 loader 之前（如下图 example 的位置）， 放置在这个 loader 之后的 loader 就会在一个单独的 worker 池(worker pool)中运行。

```
// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        include: path.resolve("src"),
        use: [
          "thread-loader",
          // 你的高开销的loader放置在此 (e.g babel-loader)
        ]
      }
    ]
  }
}
```

每个 worker 都是一个单独的有 600ms 限制的 node.js 进程。同时跨进程的数据交换也会被限制。请在高开销的loader中使用，否则效果不佳

⑥ 压缩加速——开启多线程压缩

- 不推荐使用 webpack-paralle-uglify-plugin，项目基本处于没人维护的阶段，issue 没人处理，pr没人合并。
  Webpack 4.0以前：uglifyjs-webpack-plugin，parallel参数

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        parallel: true,
      }),
    ],
  },};
```

- 推荐使用 terser-webpack-plugin

```
module.exports = {
  optimization: {
    minimizer: [new TerserPlugin(
      parallel: true   // 多线程
    )],
  },
};
```

3）优化 Webpack 的打包体积

- 压缩代码
  - webpack-paralle-uglify-plugin
  - uglifyjs-webpack-plugin 开启 parallel 参数 (不支持ES6)
  - terser-webpack-plugin 开启 parallel 参数
  - 多进程并行压缩
  - 通过 mini-css-extract-plugin 提取 Chunk 中的 CSS 代码到单独文件，通过optimize-css-assets-webpack-plugin插件 开启 cssnano 压缩 CSS。
- 提取页面公共资源：
  - 使用 html-webpack-externals-plugin，将基础包通过 CDN 引入，不打入 bundle 中
  - 使用 SplitChunksPlugin 进行(公共脚本、基础包、页面公共文件)分离(Webpack4内置)，替代了 CommonsChunkPlugin 插件
  - 基础包分离：将一些基础库放到cdn，比如vue，webpack 配置 external是的vue不打入bundle
- Tree shaking
  - purgecss-webpack-plugin 和 mini-css-extract-plugin配合使用(建议)
  - 打包过程中检测工程中没有引用过的模块并进行标记，在资源压缩时将它们从最终的bundle中去掉(只能对 ES6 Modlue生效) 开发中尽可能使用ES6 Module的模块，提高tree shaking效率
  - 禁用 babel-loader 的模块依赖解析，否则 Webpack 接收到的就都是转换过的 CommonJS 形式的模块，无法进行 tree-shaking
  - 使用 PurifyCSS(不在维护) 或者 uncss 去除无用 CSS 代码
- Scope hoisting
  - 构建后的代码会存在大量闭包，造成体积增大，运行代码时创建的函数作用域变多，内存开销变大。Scope hoisting 将所有模块的代码按照引用顺序放在一个函数作用域里，然后适当的重命名一些变量以防止变量名冲突
  - 必须是ES6的语法，因为有很多第三方库仍采用 CommonJS 语法，为了充分发挥 Scope hoisting 的作用，需要配置 mainFields 对第三方模块优先采用 jsnext:main 中指向的ES6模块化语法
- 图片压缩
  - 使用基于 Node 库的 imagemin (很多定制选项、可以处理多种图片格式)
  - 配置 image-webpack-loader
- 动态Polyfill
  - 建议采用 polyfill-service 只给用户返回需要的polyfill，社区维护。(部分国内奇葩浏览器UA可能无法识别，但可以降级返回所需全部polyfill)
  - @babel-preset-env 中通过useBuiltIns: 'usage参数来动态加载polyfill。

4）speed-measure-webpack-plugin (监控面板)

简称 SMP，分析出 Webpack 打包过程中 Loader 和 Plugin 的耗时，有助于找到构建过程中的性能瓶颈，来精准优化

```
// webpack.config.js文件
const SpeedMeasurePlugin = require('speed-measure-webpack-plugin');
const smp = new SpeedMeasurePlugin();
//.............
// 用smp.warp()包裹一下合并的config
module.exports = smp.wrap(merge(_mergeConfig, webpackConfig));
```

19、说一下事件循环机制(node、浏览器)

公司：滴滴、伴鱼、高德、自如、虎扑、58　　分类：Node、JavaScript

1）为什么会有Event Loop

JavaScript的任务分为两种同步和异步，它们的处理方式也各自不同，**同步任务**是直接放在主线程上排队依次执行，**异步任务**会放在任务队列中，若有多个异步任务则需要在任务队列中排队等待，任务队列类似于缓冲区，任务下一步会被移到**调用栈**然后主线程执行调用栈的任务。

> **调用栈**：调用栈是一个栈结构，函数调用会形成一个栈帧，帧中包含了当前执行函数的参数和局部变量等上下文信息，函数执行完后，它的执行上下文会从栈中弹出。

JavaScript是单线程的，单线程是指 js引擎中解析和执行js代码的线程只有一个（主线程），每次只能做一件事情，然而ajax请求中，主线程在等待响应的过程中回去做其他事情，浏览器先在事件表注册ajax的回调函数，响应回来后回调函数被添加到任务队列中等待执行，不会造成线程阻塞，所以说js处理ajax请求的方式是异步的。

综上所述，检查调用栈是否为空以及讲某个任务添加到调用栈中的个过程就是event loop，这就是JavaScript实现异步的核心。

2）浏览器中的Event Loop

① Micro-Task 与 Macro-Task

浏览器端事件循环中的异步队列有两种：macro（宏任务）队列和 micro（微任务）队列。

```
常见的 macro-task：setTimeout、setInterval、script（整体代码）、 I/O 操作、UI 渲染等。

常见的 micro-task: new Promise().then(回调)、MutationObserve 等。
```

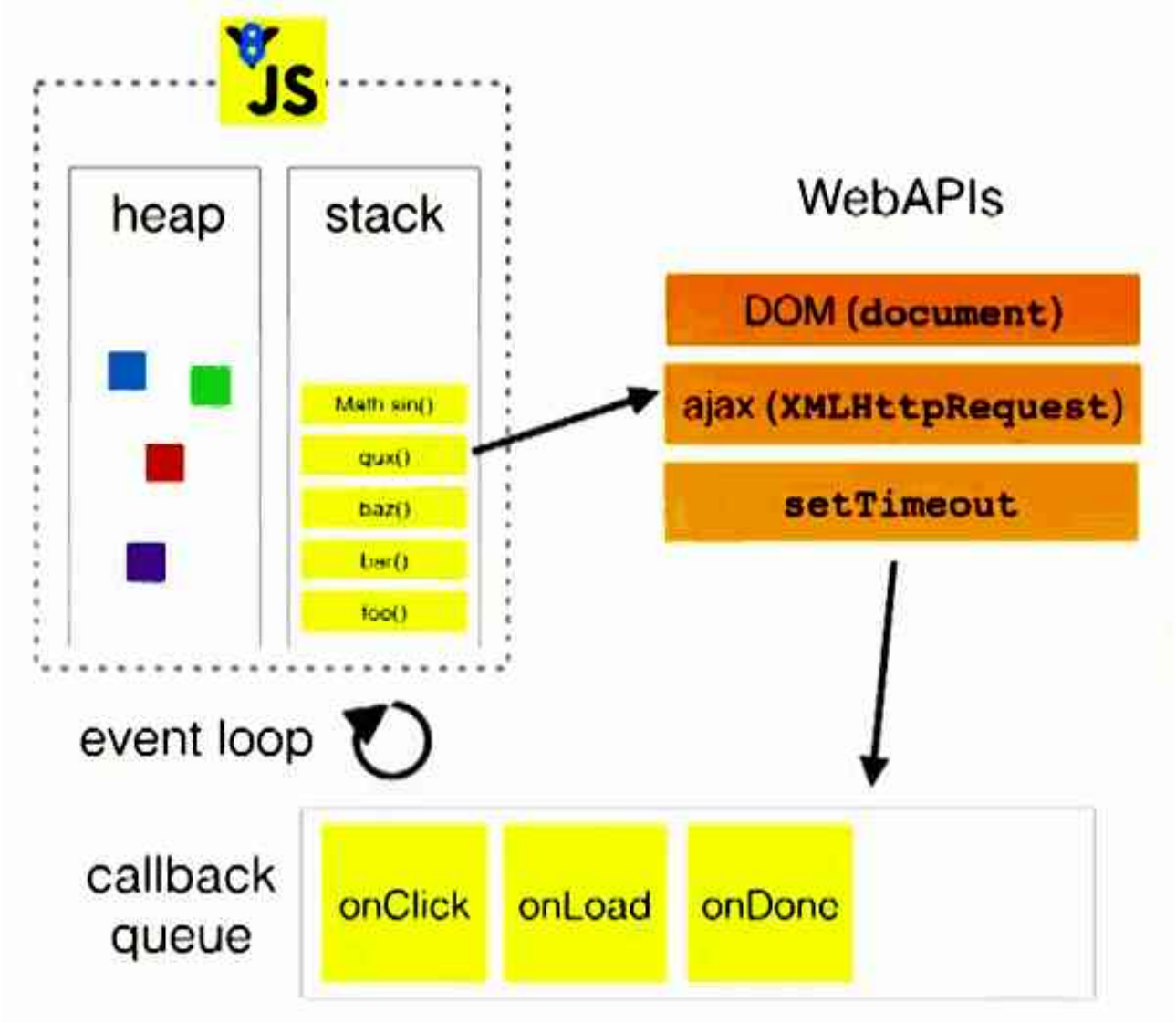
② requestAnimationFrame

requestAnimationFrame也属于异步执行的方法，但该方法既不属于宏任务，也不属于微任务。按照MDN中的定义：

> `window.requestAnimationFrame()` 告诉浏览器——你希望执行一个动画，并且要求浏览器在下次重绘之前调用指定的回调函数更新动画。该方法需要传入一个回调函数作为参数，该回调函数会在浏览器下一次重绘之前执行

requestAnimationFrame是GUI渲染之前执行，但在`Micro-Task`之后，不过requestAnimationFrame不一定会在当前帧必须执行，由浏览器根据当前的策略自行决定在哪一帧执行。

③ event loop过程

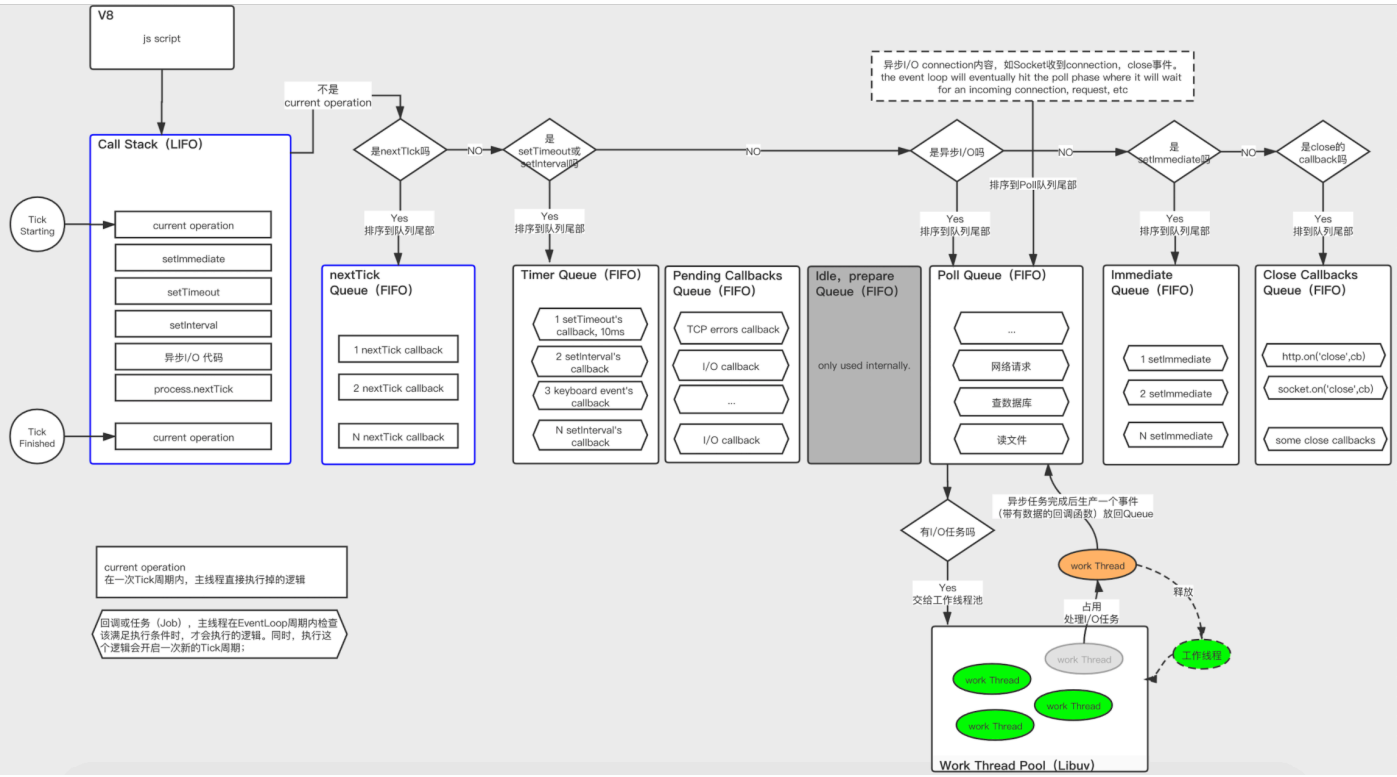


1. 检查macrotask队列是否为空，非空则到2，为空则到3
2. 执行macrotask中的一个任务
3. 继续检查microtask队列是否为空，若有则到4，否则到5
4. 取出microtask中的任务执行，执行完成返回到步骤3
5. 执行视图更新

> 当某个宏任务执行完后,会查看是否有微任务队列。如果有，先执行微任务队列中的所有任务，如果没有，会读取宏任务队列中排在最前的任务，执行宏任务的过程中，遇到微任务，依次加入微任务队列。栈空后，再次读取微任务队列里的任务，依次类推。

3）node中的 Event Loop

Node 中的 Event Loop 和浏览器中的是完全不相同的东西。Node.js采用V8作为js的解析引擎，而I/O处理方面使用了自己设计的libuv，libuv是一个基于事件驱动的跨平台抽象层，封装了不同操作系统一些底层特性，对外提供统一的API，事件循环机制也是它里面的实现

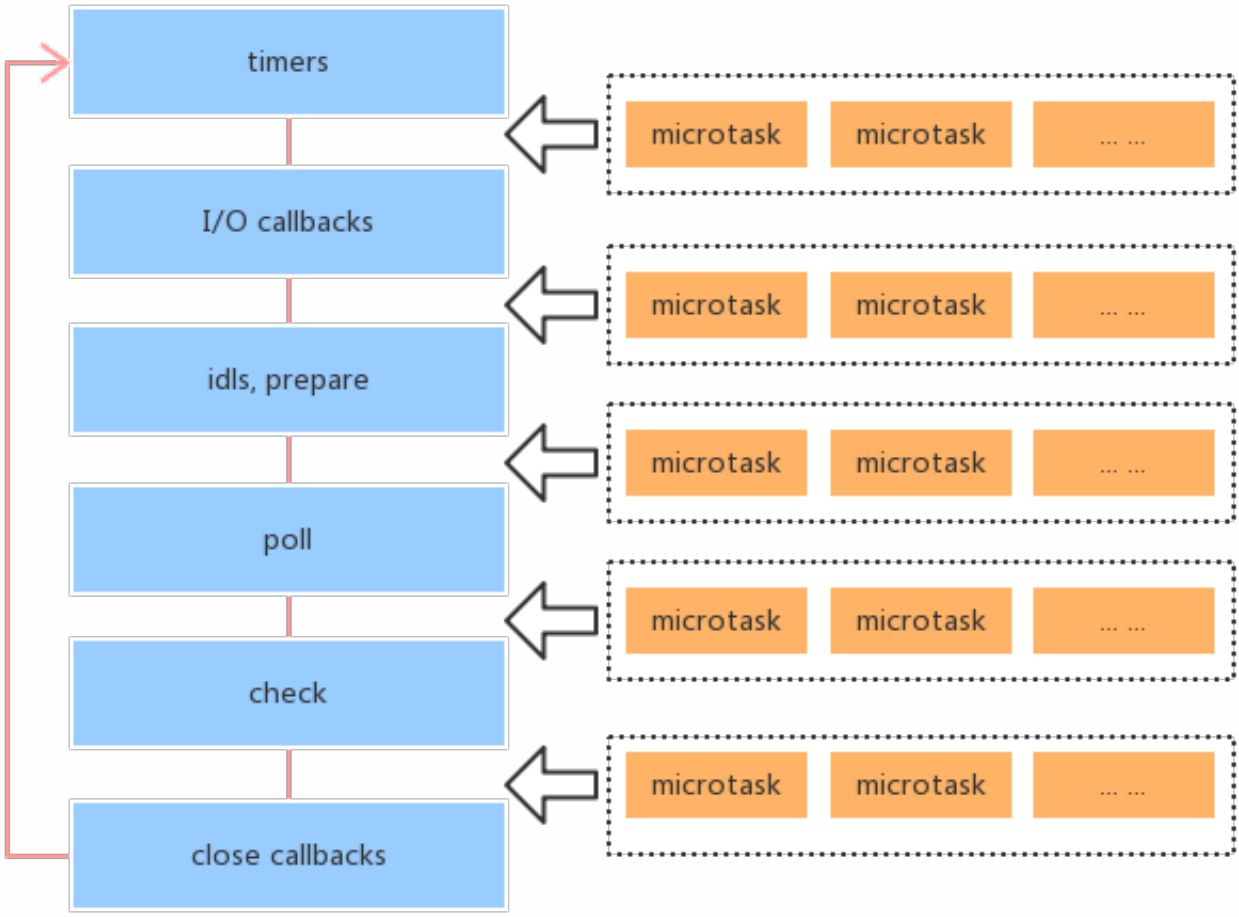


根据上图node的运行机制如下

1. V8引擎解析JavaScript脚本。
2. 解析后的代码，调用Node API。
3. libuv库负责Node API的执行。它将不同的任务分配给不同的线程，形成一个Event Loop（事件循环），以异步的方式将任务的执行结果返回给V8引擎。
4. V8引擎再将结果返回给用户。

1> 六大阶段

其中libuv引擎中的事件循环分为 6 个阶段，它们会按照顺序反复运行。每当进入某一个阶段的时候，都会从对应的回调队列中取出函数去执行。当队列为空或者执行的回调函数数量到达系统设定的阈值，就会进入下一阶段。



1. `timers` 阶段：这个阶段执行timer（setTimeout、setInterval）的回调，并且是由 poll 阶段控制的。
2. `I/O callbacks` 阶段：处理一些上一轮循环中的少数未执行的 I/O 回调
3. `idle, prepare` 阶段：仅node内部使用
4. `poll` 阶段：获取新的I/O事件，适当的条件下node将阻塞在这里
5. `check` 阶段：执行 setImmediate() 的回调
6. `close callbacks` 阶段：执行 socket 的 close 事件回调

2> poll阶段

poll 是一个至关重要的阶段，这一阶段中，系统会做两件事情

1.回到 timer 阶段执行回调
2.执行 I/O 回调

并且在进入该阶段时如果没有设定了 timer 的话，会发生以下两件事情

- 如果 poll 队列不为空，会遍历回调队列并同步执行，直到队列为空或者达到系统限制
- 如果 poll 队列为空时，会有两件事发生
    - 如果有 setImmediate 回调需要执行，poll 阶段会停止并且进入到 check 阶段执行回调
    - 如果没有 setImmediate 回调需要执行，会等待回调被加入到队列中并立即执行回调，这里同样会有个超时时间设置防止一直等待下去

当然设定了 timer 的话且 poll 队列为空，则会判断是否有 timer 超时，如果有的话会回到 timer 阶段执行回调。

3> Micro-Task 与 Macro-Task

Node端事件循环中的异步队列也是这两种：macro（宏任务）队列和 micro（微任务）队列。

- 常见的 macro-task 比如：setTimeout、setInterval、 setImmediate、script（整体代码）、 I/O 操作等。
- 常见的 micro-task 比如：process.nextTick、new Promise().then(回调)等。

4> setTimeout 和 setImmediate

二者非常相似，区别主要在于调用时机不同。

- setImmediate 设计在poll阶段完成时执行，即check阶段；
- setTimeout 设计在poll阶段为空闲时，且设定时间到达后执行，但它在timer阶段执行

```
setTimeout(function timeout () {
console.log('timeout');
},0);
setImmediate(function immediate () {
console.log('immediate');
});
```

1. 对于以上代码来说，setTimeout 可能执行在前，也可能执行在后。
2. 首先 setTimeout(fn, 0) === setTimeout(fn, 1)，这是由源码决定的 进入事件循环也是需要成本的，如果在准备时候花费了大于 1ms 的时间，那么在 timer 阶段就会直接执行 setTimeout 回调
3. 如果准备时间花费小于 1ms，那么就是 setImmediate 回调先执行了

4）Node与浏览器的 Event Loop 差异

- Node端，microtask 在事件循环的各个阶段之间执行
- 浏览器端，microtask 在事件循环的 macrotask 执行完之后执行

20、如何封装 Node 中间件

公司：**滴滴、酷狗**　　分类：**Node**

在NodeJS中，中间件主要是指封装所有Http请求细节处理的方法。一次Http请求通常包含很多工作，如记录日志、ip过滤、查询字符串、请求体解析、Cookie处理、权限验证、参数验证、异常处理等，但对于Web应用而言，并不希望接触到这么多细节性的处理，因此引入中间件来简化和隔离这些基础设施与业务逻辑之间的细节，让开发者能够关注在业务的开发上，以达到提升开发效率的目的。

中间件的行为比较类似Java中过滤器的工作原理，就是在进入具体的业务处理之前，先让过滤器处理。

```
const http = require('http')
function compose(middlewareList) {
  return function (ctx) {
    function dispatch (i) {
```

```
      const fn = middlewareList[i]
      try {
        return Promise.resolve(fn(ctx, dispatch.bind(null, i + 1)))
      } catch (err) {
        Promise.reject(err)
      }
    }
    return dispatch(0)
  }
}
class App {
  constructor(){
    this.middlewares = []
  }
  use(fn){
    this.middlewares.push(fn)
    return this
  }
  handleRequest(ctx, middleware) {
    return middleware(ctx)
  }
  createContext (req, res) {
    const ctx = {
      req,
      res
    }
    return ctx
  }
  callback () {
    const fn = compose(this.middlewares)
    return (req, res) => {
      const ctx = this.createContext(req, res)
      return this.handleRequest(ctx, fn)
    }
  }
  listen(...args) {
    const server = http.createServer(this.callback())
    return server.listen(...args)
  }
}
module.exports = App
```

21、node 中间层怎么样做的请求合并转发

公司：易车　　分类：Node

1） 什么是中间层

- 就是前端---请求---> nodejs ----请求---->后端 ----响应--->nodejs--数据处理---响应---->前端。这么一个流程，这个流程的好处就是当业务逻辑过多，或者业务需求在不断变更的时候，前端不需要过多当去改变业务逻辑，与后端低耦合。前端即显示，渲染。后端获取和存储数据。中间层处理数据结构，返回给前端可用可渲染的数据结构。
- nodejs是起中间层的作用，即根据客户端不同请求来做相应的处理或渲染页面，处理时可以是把获取的数据做简单的处理交由底层java那边做真正的数据持久化或数据更新，也可以是从底层获取数据做简单的处理返回给客户端。
- 通常我们把Web领域分为客户端和服务端，也就是前端和后端，这里的后端就包含了网关，静态资源，接口，缓存，数据库等。而中间层呢，就是在后端这里再抽离一层出来，在业务上处理和客户端衔接更紧密的部分，比如页面渲染（SSR），数据聚合，接口转发等等。
- 以SSR来说，在服务端将页面渲染好，可以加快用户的首屏加载速度，避免请求时白屏，还有利于网站做SEO，他的好处是比较好理解的。

2）中间层可以做的事情

- 代理：在开发环境下，我们可以利用代理来，解决最常见的跨域问题；在线上环境下，我们可以利用代理，转发请求到多个服务端。
- 缓存：缓存其实是更靠近前端的需求，用户的动作触发数据的更新，node中间层可以直接处理一部分缓存需求。
- 限流：node中间层，可以针对接口或者路由做响应的限流。
- 日志：相比其他服务端语言，node中间层的日志记录，能更方便快捷的定位问题（是在浏览器端还是服务端）。
- 监控：擅长高并发的请求处理，做监控也是合适的选项。
- 鉴权：有一个中间层去鉴权，也是一种单一职责的实现。
- 路由：前端更需要掌握页面路由的权限和逻辑。
- 服务端渲染：node中间层的解决方案更灵活，比如SSR、模板直出、利用一些JS库做预渲染等等。

3）node转发API（node中间层）的优势

- 可以在中间层把java|php的数据，处理成对前端更友好的格式
- 可以解决前端的跨域问题，因为服务器端的请求是不涉及跨域的，跨域是浏览器的同源策略导致的
- 可以将多个请求在通过中间层合并，减少前端的请求

4）如何做请求合并转发

- 使用express中间件multifetch可以将请求批量合并
- 使用express+http-proxy-middleware实现接口代理转发

5）不使用用第三方模块手动实现一个nodejs代理服务器，实现请求合并转发

1、实现思路

- ①搭建http服务器，使用Node的http模块的createServer方法
- ②接收客户端发送的请求，就是请求报文，请求报文中包括请求行、请求头、请求体
- ③将请求报文发送到目标服务器，使用http模块的request方法

2、实现步骤

- 第一步：http服务器搭建

```
const http = require("http");
const server = http.createServer();
server.on('request',(req,res)=>{
    res.end("hello world")
})
server.listen(3000,()=>{
    console.log("running");
})
```

- 第二步：接收客户端发送到代理服务器的请求报文

```
const http = require("http");
const server = http.createServer();
server.on('request',(req,res)=>{
    // 通过req的data事件和end事件接收客户端发送的数据
    // 并用Buffer.concat处理一下
    //
    let postbody = [];
    req.on("data", chunk => {
        postbody.push(chunk);
    })
    req.on('end', () => {
        let postbodyBuffer = Buffer.concat(postbody);
        res.end(postbodyBuffer)
    })
})
server.listen(3000,()=>{
    console.log("running");
})
```

这一步主要数据在客户端到服务器端进行传输时在nodejs中需要用到buffer来处理一下。处理过程就是将所有接收的数据片段chunk塞到一个数组中，然后将其合并到一起还原出源数据。合并方法需要用到Buffer.concat，这里不能使用加号，加号会隐式的将buffer转化为字符串，这种转化不安全。

- 第三步：使用http模块的request方法，将请求报文发送到目标服务器

> 第二步已经得到了客户端上传的数据，但是缺少请求头，所以在这一步根据客户端发送的请求需要构造请求头，然后发送

```
const http = require("http");
const server = http.createServer();

server.on("request", (req, res) => {
    var { connection, host, ...originHeaders } = req.headers;
    var options = {
        "method": req.method,
        // 随表找了一个网站做测试，被代理网站修改这里
```

```
        "hostname": "www.nanjingmb.com",
        "port": "80",
        "path": req.url,
        "headers": { originHeaders }
    }
    //接收客户端发送的数据
    var p = new Promise((resolve,reject)=>{
        let postbody = [];
        req.on("data", chunk => {
            postbody.push(chunk);
        })
        req.on('end', () => {
            let postbodyBuffer = Buffer.concat(postbody);
            resolve(postbodyBuffer)
        })
    });
    //将数据转发，并接收目标服务器返回的数据，然后转发给客户端
    p.then((postbodyBuffer)=>{
        let responsebody=[]
        var request = http.request(options, (response) => {
            response.on('data', (chunk) => {
                responsebody.push(chunk)
            })
            response.on("end", () => {
                responsebodyBuffer = Buffer.concat(responsebody)
                res.end(responsebodyBuffer);
            })
        })
        // 使用request的write方法传递请求体
        request.write(postbodyBuffer)
        // 使用end方法将请求发出去
        request.end();
    })
});
server.listen(3000, () => {
    console.log("runnng");
})
```

22、介绍下 promise 的特性、优缺点，内部是如何实现的，动手实现 Promise

公司：滴滴、头条、喜马拉雅、兑吧、寺库、百分点、58、安居客　　分类：JavaScript、编程题

1）Promise基本特性

1、Promise有三种状态：pending(进行中)、fulfilled(已成功)、rejected(已失败)
2、Promise对象接受一个回调函数作为参数，该回调函数接受两个参数，分别是成功时的回调resolve和失败时的回调reject；另外resolve的参数除了正常值以外， 还可能是一个Promise对象的实例；reject的参数通常是一个Error对象的实例。
3、then方法返回一个新的Promise实例，并接收两个参数onResolved(fulfilled状态的回调)；onRejected(rejected状态的回调，该参数可选)
4、catch方法返回一个新的Promise实例
5、finally方法不管Promise状态如何都会执行，该方法的回调函数不接受任何参数
6、Promise.all()方法将多个多个Promise实例，包装成一个新的Promise实例，该方法接受一个由Promise对象组成的数组作为参数(Promise.all()方法的参数可以不是数组，但必须具有Iterator接口，且返回的每个成员都是Promise实例)，注意参数中只要有一个实例触发catch方法，都会触发Promise.all()方法返回的新的实例的catch方法，如果参数中的某个实例本身调用了catch方法，将不会触发Promise.all()方法返回的新实例的catch方法
7、Promise.race()方法的参数与Promise.all方法一样，参数中的实例只要有一个率先改变状态就会将该实例的状态传给Promise.race()方法，并将返回值作为Promise.race()方法产生的Promise实例的返回值
8、Promise.resolve()将现有对象转为Promise对象，如果该方法的参数为一个Promise对象，Promise.resolve()将不做任何处理；如果参数thenable对象(即具有then方法)，Promise.resolve()将该对象转为Promise对象并立即执行then方法；如果参数是一个原始值，或者是一个不具有then方法的对象，则Promise.resolve方法返回一个新的Promise对象，状态为fulfilled，其参数将会作为then方法中onResolved回调函数的参数，如果Promise.resolve方法不带参数，会直接返回一个fulfilled状态的 Promise 对象。需要注意的是，立即resolve()的 Promise 对象，是在本轮“事件循环”（event loop）的结束时执行，而不是在下一轮“事件循环”的开始时。
9、Promise.reject()同样返回一个新的Promise对象，状态为rejected，无论传入任何参数都将作为reject()的参数

2）Promise优点

①统一异步 API

- Promise 的一个重要优点是它将逐渐被用作浏览器的异步 API，统一现在各种各样的 API，以及不兼容的模式和手法。

②Promise 与事件对比

- 和事件相比较， Promise 更适合处理一次性的结果。在结果计算出来之前或之后注册回调函数都是可以的，都可以拿到正确的值。 Promise 的这个优点很自然。但是，不能使用 Promise 处理多次触发的事件。链式处理是 Promise 的又一优点，但是事件却不能这样链式处理。

③Promise 与回调对比

- 解决了回调地狱的问题，将异步操作以同步操作的流程表达出来。

④Promise 带来的额外好处是包含了更好的错误处理方式（包含了异常处理），并且写起来很轻松（因为可以重用一些同步的工具，比如 Array.prototype.map() ）。

3）Promise缺点

1、无法取消Promise，一旦新建它就会立即执行，无法中途取消。
2、如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。
3、当处于Pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。
4、Promise 真正执行回调的时候，定义 Promise 那部分实际上已经走完了，所以 Promise 的报错堆栈上下文不太友好。

4）简单代码实现

最简单的Promise实现有7个主要属性, state(状态), value(成功返回值), reason(错误信息), resolve方法, reject方法, then方法.

```
class Promise{
  constructor(executor) {
    this.state = 'pending';
    this.value = undefined;
    this.reason = undefined;
    let resolve = value => {
      if (this.state === 'pending') {
        this.state = 'fulfilled';
        this.value = value;
      }
    };
    let reject = reason => {
      if (this.state === 'pending') {
        this.state = 'rejected';
        this.reason = reason;
      }
    };
    try {
      // 立即执行函数
      executor(resolve, reject);
    } catch (err) {
      reject(err);
    }
  }
  then(onFulfilled, onRejected) {
    if (this.state === 'fulfilled') {
      let x = onFulfilled(this.value);
    };
    if (this.state === 'rejected') {
      let x = onRejected(this.reason);
    };
  }
}
```

5）面试够用版

```
function myPromise(constructor){ let self=this;
  self.status="pending" //定义状态改变前的初始状态
  self.value=undefined;//定义状态为resolved的时候的状态
  self.reason=undefined;//定义状态为rejected的时候的状态
  function resolve(value){
    //两个==="pending"，保证了了状态的改变是不不可逆的
    if(self.status==="pending"){
      self.value=value;
      self.status="resolved";
```

```
    }
  }
  function reject(reason){
     //两个==="pending"，保证了了状态的改变是不不可逆的
     if(self.status==="pending"){
        self.reason=reason;
        self.status="rejected";
      }
  }
  //捕获构造异常
  try{
      constructor(resolve,reject);
  }catch(e){
    reject(e);
    }
}
myPromise.prototype.then=function(onFullfilled,onRejected){
  let self=this;
  switch(self.status){
    case "resolved": onFullfilled(self.value); break;
    case "rejected": onRejected(self.reason); break;
    default:
  }
}

// 测试
var p=new myPromise(function(resolve,reject){resolve(1)});
p.then(function(x){console.log(x)})
//输出1
```

6）大厂专供版

```
const PENDING = "pending";
const FULFILLED = "fulfilled";
const REJECTED = "rejected";
function Promise(excutor) {
  let that = this; // 缓存当前promise实例例对象
  that.status = PENDING; // 初始状态
  that.value = undefined; // fulfilled状态时 返回的信息
  that.reason = undefined; // rejected状态时 拒绝的原因
  that.onFulfilledCallbacks = []; // 存储fulfilled状态对应的onFulfilled函数
  that.onRejectedCallbacks = []; // 存储rejected状态对应的onRejected函数
  function resolve(value) { // value成功态时接收的终值
    if(value instanceof Promise) {
      return value.then(resolve, reject);
    }
    // 实践中要确保 onFulfilled 和 onRejected 方方法异步执行行行，且应该在 then 方方法被调用用的那
一一轮事件循环之后的新执行行行栈中执行行行。
    setTimeout(() => {
      // 调用用resolve 回调对应onFulfilled函数
      if (that.status === PENDING) {
```

```
        // 只能由pending状态 => fulfilled状态 (避免调用用多次resolve reject)
        that.status = FULFILLED;
        that.value = value;
        that.onFulfilledCallbacks.forEach(cb => cb(that.value));
      }
    });
  }
  function reject(reason) { // reason失败态时接收的拒因
    setTimeout(() => {
      // 调用用reject 回调对应onRejected函数
      if (that.status === PENDING) {
        // 只能由pending状态 => rejected状态 (避免调用用多次resolve reject)
        that.status = REJECTED;
        that.reason = reason;
        that.onRejectedCallbacks.forEach(cb => cb(that.reason));
      }
    });
  }

  // 捕获在excutor执行行行器器中抛出的异常
  // new Promise((resolve, reject) => {
  //     throw new Error('error in excutor')
  // })
  try {
    excutor(resolve, reject);
  } catch (e) {
    reject(e);
  }
}
Promise.prototype.then = function(onFulfilled, onRejected) {
  const that = this;
  let newPromise;
  // 处理理参数默认值 保证参数后续能够继续执行行行
  onFulfilled = typeof onFulfilled === "function" ? onFulfilled : value => value;
  onRejected = typeof onRejected === "function" ? onRejected : reason => {
    throw reason;
  };
  if (that.status === FULFILLED) { // 成功态
    return newPromise = new Promise((resolve, reject) => {
      setTimeout(() => {
        try{
          let x = onFulfilled(that.value);
          resolvePromise(newPromise, x, resolve, reject); //新的promise resolve 上一一个
onFulfilled的返回值
        } catch(e) {
          reject(e); // 捕获前面面onFulfilled中抛出的异常then(onFulfilled, onRejected);
        }
      });
    })
  }
  if (that.status === REJECTED) { // 失败态
    return newPromise = new Promise((resolve, reject) => {
```

```
      setTimeout(() => {
        try {
          let x = onRejected(that.reason);
          resolvePromise(newPromise, x, resolve, reject);
        } catch(e) {
          reject(e);
        }
      });
    });
  }
  if (that.status === PENDING) { // 等待态
// 当异步调用用resolve/rejected时 将onFulfilled/onRejected收集暂存到集合中
    return newPromise = new Promise((resolve, reject) => {
      that.onFulfilledCallbacks.push((value) => {
        try {
          let x = onFulfilled(value);
          resolvePromise(newPromise, x, resolve, reject);
        } catch(e) {
          reject(e);
        }
      });
      that.onRejectedCallbacks.push((reason) => {
        try {
          let x = onRejected(reason);
          resolvePromise(newPromise, x, resolve, reject);
        } catch(e) {
          reject(e);
        }
      });
    });
  }
};
```

23、如何实现 Promise.all

```
Promise.all = function (arr) {
  // 实现代码
};
```

公司：**滴滴、头条、有赞、微医**　　分类：**JavaScript、编程题**

1）核心思路

- 接收一个 Promise 实例的数组或具有 Iterator 接口的对象作为参数
- 这个方法返回一个新的 promise 对象
- 遍历传入的参数，用Promise.resolve()将参数"包一层"，使其变成一个promise对象
- 参数所有回调成功才是成功，返回值数组与参数顺序一致
- 参数数组其中一个失败，则触发失败状态，第一个触发失败的 Promise 错误信息作为 Promise.all 的错误信息。

2）实现代码

> 一般来说，Promise.all 用来处理多个并发请求，也是为了页面数据构造的方便，将一个页面所用到的在不同接口的数据一起请求过来，不过，如果其中一个接口失败了，多个请求也就失败了，页面可能啥也出不来，这就看当前页面的耦合程度了~

```
  /**
   * Promise.all
   * @description 当这个数组里的所有promise对象全部变为resolve状态的时候，才会resolve，当有一个promise对象变为reject状态时，就不再执行直接 reject
   * @param {*} values promise对象组成的数组作为参数
   */

Promise.prototype.all = (values)=>{
  return new Promise((resolve,reject)=>{
      let resultArr = []
      let count = 0
      let resultByKey = (value,index)=>{
           resultArr[index] = value
           if(++count === values.length){
                resolve(resultArr)
           }
      }
      values.forEach((promise,index)=>{
         promise.then((value)=>{
           resultByKey(value,index)
         },reject)
      })
  })

}
```

24、React 组件通信方式

公司：**滴滴、掌门一对一、喜马拉雅、蘑菇街**　　分类：**React**

react组件间通信常见的几种情况:

- 父组件向子组件通信
- 子组件向父组件通信
- 跨级组件通信
- 非嵌套关系的组件通信

1）父组件向子组件通信：父组件通过 props 向子组件传递需要的信息

```
// 子组件: Child
const Child = props =>{
  return <p>{props.name}</p>
}

// 父组件 Parent
const Parent = ()=>{
    return <Child name="京程一灯"></Child>
}
```

2）子组件向父组件通信：：props+回调的方式

```
// 子组件: Child
const Child = props =>{
  const cb = msg =>{
      return ()=>{
          props.callback(msg)
      }
  }
  return (
      <button onClick={cb("京程一灯欢迎你!")}>京程一灯欢迎你</button>
  )
}

// 父组件 Parent
class Parent extends Component {
    callback(msg){
        console.log(msg)
    }
    render(){
        return <Child callback={this.callback.bind(this)}></Child>
    }
}
```

3）跨级组件通信：即父组件向子组件的子组件通信，向更深层子组件通信

- 使用props，利用中间组件层层传递,但是如果父组件结构较深，那么中间每一层组件都要去传递props，增加了复杂度，并且这些props并不是中间组件自己需要的。
- 使用context，context相当于一个大容器，我们可以把要通信的内容放在这个容器中，这样不管嵌套多深，都可以随意取用，对于跨越多层的全局数据可以使用context实现。

```
// context方式实现跨级组件通信
// Context 设计目的是为了共享那些对于一个组件树而言是“全局”的数据

const BatteryContext = createContext();

//  子组件的子组件
class GrandChild extends Component {
    render(){
```

```
        return (
            <BatteryContext.Consumer>
                {
                    color => <h1 style={{"color":color}}>我是红色的:{color}</h1>
                }
            </BatteryContext.Consumer>
        )
    }
}

//  子组件
const Child = () =>{
    return (
        <GrandChild/>
    )
}
// 父组件
class Parent extends Component {
      state = {
          color:"red"
      }
      render(){
          const {color} = this.state
          return (
          <BatteryContext.Provider value={color}>
              <Child></Child>
          </BatteryContext.Provider>
          )
      }
}
```

4）非嵌套关系的组件通信：即没有任何包含关系的组件，包括兄弟组件以及不在同一个父级中的非兄弟组件

- 可以使用自定义事件通信（发布订阅模式）
- 可以通过redux等进行全局状态管理
- 如果是兄弟组件通信，可以找到这两个兄弟节点共同的父节点，结合父子间通信方式进行通信。

25、redux-saga 和 mobx 的比较

公司：掌门一对一　　分类：React

1）状态管理

- redux-sage 是 redux 的一个异步处理的中间件。
- mobx 是数据管理库，和 redux 一样。

2）设计思想

- redux-sage 属于 flux 体系， 函数式编程思想。
- mobx 不属于 flux 体系，面向对象编程和响应式编程。

3）主要特点

- redux-sage 因为是中间件，更关注异步处理的，通过 Generator 函数来将异步变为同步，使代码可读性高，结构清晰。action 也不是 action creator 而是 pure action，
- 在 Generator 函数中通过 call 或者 put 方法直接声明式调用，并自带一些方法，如 takeEvery，takeLast，race 等，控制多个异步操作，让多个异步更简单。
- mobx 是更简单更方便更灵活的处理数据。 Store 是包含了 state 和 action。state 包装成一个可被观察的对象，action 可以直接修改 state，之后通过 Computed values 将依赖 state 的计算属性更新 ，之后触发 Reactions 响应依赖 state 的变更，输出相应的副作用 ，但不生成新的 state。

4）数据可变性

- redux-sage 强调 state 不可变，不能直接操作 state，通过 action 和 reducer 在原来的 state 的基础上返回一个新的 state 达到改变 state 的目的。
- mobx 直接在方法中更改 state，同时所有使用的 state 都发生变化，不生成新的 state。

5）写法难易度

- redux-sage 比 redux 在 action 和 reducer 上要简单一些。需要用 dispatch 触发 state 的改变，需要 mapStateToProps 订阅 state。
- mobx 在非严格模式下不用 action 和 reducer，在严格模式下需要在 action 中修改 state，并且自动触发相关依赖的更新。

6）使用场景

- redux-sage 很好的解决了 redux 关于异步处理时的复杂度和代码冗余的问题，数据流向比较好追踪。但是 redux 的学习成本比 较高，代码比较冗余，不是特别需要状态管理，最好用别
  的方式代替。
- mobx 学习成本低，能快速上手，代码比较简洁。但是可能因为代码编写的原因和数据更新时相对黑盒，导致数据流向不利于追踪。

26、简单说一下 react-fiber

公司：**头条、滴滴、菜鸟网络、挖财、喜马拉雅**　　分类：**React**

1）背景

- react在进行组件渲染时，从setState开始到渲染完成整个过程是同步的（“一气呵成”）。如果需要渲染的组件比较庞大，js执行会占据主线程时间较长，会导致页面响应度变差，使得react在动画、手势等应用中效果比较差。
- 页面卡顿：Stack reconciler的工作流程很像函数的调用过程。父组件里调子组件，可以类比为函数的递归；对于特别庞大的vDOM树来说，reconciliation过程会很长(x00ms)，超过16ms,在这期间，主线程是被js占用的，因此任何交互、布局、渲染都会停止，给用户的感觉就是页面被卡住了。

2）实现原理

> 旧版 React 通过递归的方式进行渲染，使用的是 JS 引擎自身的函数调用栈，它会一直执行到栈空为止。而Fiber实现了自己的组件调用栈，它以链表的形式遍历组件树，可以灵活的暂停、继续和丢弃执行的任务。实现方式是使用了浏览器的requestIdleCallback这一 API。
> Fiber 其实指的是一种数据结构，它可以用一个纯 JS 对象来表示：

```
const fiber = {
 stateNode,    // 节点实例
 child,        // 子节点
 sibling,      // 兄弟节点
 return,       // 父节点
}
```

- react内部运转分三层：
  - Virtual DOM 层，描述页面长什么样。
  - Reconciler 层，负责调用组件生命周期方法，进行 Diff 运算等。
  - Renderer 层，根据不同的平台，渲染出相应的页面，比较常见的是 ReactDOM 和 ReactNative。
- 为了实现不卡顿，就需要有一个调度器 (Scheduler) 来进行任务分配。优先级高的任务（如键盘输入）可以打断优先级低的任务（如Diff）的执行，从而更快的生效。任务的优先级有六种：
  - synchronous，与之前的Stack Reconciler操作一样，同步执行
  - task，在next tick之前执行
  - animation，下一帧之前执行
  - high，在不久的将来立即执行
  - low，稍微延迟执行也没关系
  - offscreen，下一次render时或scroll时才执行
- Fiber Reconciler（react ）执行阶段：
  - 阶段一，生成 Fiber 树，得出需要更新的节点信息。这一步是一个渐进的过程，可以被打断。
  - 阶段二，将需要更新的节点一次过批量更新，这个过程不能被打断。
- Fiber树：React 在 render 第一次渲染时，会通过 React.createElement 创建一颗 Element 树，可以称之为 Virtual DOM Tree，由于要记录上下文信息，加入了 Fiber，每一个 Element 会对应一个 Fiber Node，将 Fiber Node 链接起来的结构成为 Fiber Tree。Fiber Tree 一个重要的特点是链表结构，将递归遍历编程循环遍历，然后配合 requestIdleCallback API, 实现任务拆分、中断与恢复。
- 从Stack Reconciler到Fiber Reconciler，源码层面其实就是干了一件递归改循环的事情

27、手写发布订阅

公司：**头条、滴滴**　　分类：**JavaScript**

```
// 发布订阅中心，on-订阅，off取消订阅，emit发布，内部需要一个单独事件中心caches进行存储；

interface CacheProps {
  [key: string]: Array<((data?: unknown) => void)>;
}

class Observer {

  private caches: CacheProps = {}; // 事件中心
```

```
  on (eventName: string, fn: (data?: unknown) => void){ // eventName事件名-独一无二，fn订阅后执行的自定义行为
    this.caches[eventName] = this.caches[eventName] || [];
    this.caches[eventName].push(fn);
  }

  emit (eventName: string, data?: unknown) { // 发布 => 将订阅的事件进行统一执行
    if (this.caches[eventName]) {
      this.caches[eventName].forEach((fn: (data?: unknown) => void) => fn(data));
    }
  }

  off (eventName: string, fn?: (data?: unknown) => void) { // 取消订阅 => 若fn不传，直接取消该事件所有订阅信息
    if (this.caches[eventName]) {
      const newCaches = fn ? this.caches[eventName].filter(e => e !== fn) : [];
      this.caches[eventName] = newCaches;
    }
  }

}
```

28、手写数组转树

公司：滴滴　　分类：JavaScript

```
var list = [
  { id: 1, name: '部门A', parentId: 0 },
  { id: 3, name: '部门C', parentId: 1 },
  { id: 4, name: '部门D', parentId: 1 },
  { id: 5, name: '部门E', parentId: 2 },
  { id: 6, name: '部门F', parentId: 3 },
  { id: 7, name: '部门G', parentId: 2 },
  { id: 8, name: '部门H', parentId: 4 }
];
function convert(list) {
  const map = list.reduce((acc, item) => {
    acc[item.id] = item
    return acc
  }, {})
  const result = []
  for (const key in map) {
    const item = map[key]
    if (item.parentId === 0) {
      result.push(item)
    } else {
      const parent = map[item.parentId]
      if (parent) {
        parent.children = parent.children || []
        parent.children.push(item)
```

```
      }
    }
  }
  return result
}
var result = convert(list)
```

29、使用ES6 的Proxy实现数组负索引。（负索引：例如，可以简单地使用arr[-1]替代arr[arr.length-1]访问最后一个元素，[-2]访问倒数第二个元素，以此类推）

公司：滴滴　　分类：JavaScript

```
const proxyArray = arr => {
    const length = arr.length;
    return new Proxy(arr, {
        get(target, key) {
            key = +key;
            while (key < 0) {
                key += length;
            }
            return target[key];
        }
    })
};
var a = proxyArray([1, 2, 3, 4, 5, 6, 7, 8, 9]);
console.log(a[1]);  // 2
console.log(a[-10]);  // 9
console.log(a[-20]);  // 8
```

30、请写出以下代码的执行结果

```
console.log(1);
setTimeout(() => {
  console.log(2);
  process.nextTick(() => {
    console.log(3);
  });
  new Promise((resolve) => {
    console.log(4);
    resolve();
  }).then(() => {
    console.log(5);
  });
});
new Promise((resolve) => {
  console.log(7);
  resolve();
}).then(() => {
  console.log(8);
```

```
});
process.nextTick(() => {
  console.log(6);
});
setTimeout(() => {
  console.log(9);
  process.nextTick(() => {
    console.log(10);
  });
  new Promise((resolve) => {
    console.log(11);
    resolve();
  }).then(() => {
    console.log(12);
  });
});
```

公司：滴滴　　分类：JavaScript

答案：

```
node <11:1 7 6 8 2 4 9 11 3 10 5 12
node>=11:1 7 6 8 2 4 3 5 9 11 10 12
```

思路：

- 宏任务和微任务
  - 宏任务：macrotask,包括setTimeout、setInerVal、setImmediate(node独有)、requestAnimationFrame(浏览器独有)、I/O、UI rendering(浏览器独有)
  - 微任务：microtask,包括process.nextTick(Node独有)、Promise.then()、Object.observe、MutationObserver
- Promise构造函数中的代码是同步执行的，new Promise()构造函数中的代码是同步代码，并不是微任务
- Node.js中的EventLoop执行宏队列的回调任务有

  6个阶段
  - 1.timers阶段：这个阶段执行setTimeout和setInterval预定的callback
  - 2.I/O callback阶段：执行除了close事件的callbacks、被timers设定的callbacks、setImmediate()设定的callbacks这些之外的callbacks
  - 3.idle, prepare阶段：仅node内部使用
  - 4.poll阶段：获取新的I/O事件，适当的条件下node将阻塞在这里
  - 5.check阶段：执行setImmediate()设定的callbacks
  - 6.close callbacks阶段：执行socket.on('close', ...)这些callbacks
- NodeJs中宏队列主要有4个
  - 1.Timers Queue
  - 2.IO Callbacks Queue
  - 3.Check Queue
  - 4.Close Callbacks Queue
  - 这4个都属于宏队列，但是在浏览器中，可以认为只有一个宏队列，所有的macrotask都会被加到这一个宏队列中，但是在NodeJS中，不同的macrotask会被放置在不同的宏队列中。

- NodeJS中微队列主要有2个
  - 1.Next Tick Queue：是放置process.nextTick(callback)的回调任务的
  - 2.Other Micro Queue：放置其他microtask，比如Promise等
  - 在浏览器中，也可以认为只有一个微队列，所有的microtask都会被加到这一个微队列中，但是在NodeJS中，不同的microtask会被放置在不同的微队列中。
- Node.js中的EventLoop过程
  - 1.执行全局Script的同步代码
  - 2.执行microtask微任务，先执行所有Next Tick Queue中的所有任务，再执行Other Microtask Queue中的所有任务
  - 3.开始执行macrotask宏任务，共6个阶段，从第1个阶段开始执行相应每一个阶段macrotask中的所有任务，注意，这里是所有每个阶段宏任务队列的所有任务，在浏览器的Event Loop中是只取宏队列的第一个任务出来执行，每一个阶段的macrotask任务执行完毕后，开始执行微任务，也就是步骤2
  - 4.Timers Queue -> 步骤2 -> I/O Queue -> 步骤2 -> Check Queue -> 步骤2 -> Close Callback Queue -> 步骤2 -> Timers Queue ......
  - 5.这就是Node的Event Loop
- Node 11.x新变化
  - 现在node11在timer阶段的setTimeout,setInterval...和在check阶段的immediate都在node11里面都修改为一旦执行一个阶段里的一个任务就立刻执行微任务队列。为了和浏览器更加趋同.

31、请写出以下代码的执行结果，并说明原因

```
function side(arr) {
  arr[0] = arr[2];
}
function a(a, b, c = 3) {
  c = 10;
  side(arguments);
  return a + b + c;
}
a(1, 1, 1);
```

分类：JavaScript

答案：12

思路：

arguments 中 c 的值还是 1 不会变成 10，因为 a 函数加了默认值，就按 ES 的方式解析，ES6 是有块级作用域的，所以 c 的值是不会改变的

是arguments 中 c 的值，function a(a, b, c = 3) 这里的c，因为 a 函数加了默认值，所以就按 ES 的方式解析，函数中的参数就不会变了

```
function side(arr) {
arr[0] = arr[2];
}
function a(a, b, c = 3) {
c = 10;
console.log(arguments);
side(arguments);  // 这里 a，c的值不管怎么改变都是不会改变的
return a + b + c;
}
a(1, 1, 1);  //12
```

但是，如果是

```
function side(arr) {
arr[0] = arr[2];
}
function a(a, b, c) {
c = 10;
console.log(arguments);
side(arguments);  // 这里 a，c的值不管怎么改变都是不会改变的
return a + b + c;
}
a(1, 1, 1);  // 21
```

32、写出执行结果，并说明原因

```
var min = Math.min();
max = Math.max();
console.log(min < max);
```

分类：JavaScript

答案：false

思路：

- 按常规的思路，这段代码应该输出 true，毕竟最小值小于最大值。但是却输出 false
- MDN 相关文档是这样解释的
  - Math.min 的参数是 0 个或者多个，如果多个参数很容易理解，返回参数中最小的。如果没有参数，则返回 Infinity，无穷大。
  - 而 Math.max 没有传递参数时返回的是-Infinity.所以输出 false

33、写出执行结果，并说明原因

```
var a = 1;
(function a () {
    a = 2;
    console.log(a);
})();
```

分类：JavaScript

答案：

```
f  a () {
      a = 2;
      console.log(a);
 }
```

思路：

> 立即执行的函数表达式(IIFE)的函数名称跟内部变量名称重名后，函数名称优先，因为函数名称是不可改变的，内部会静默失败，在严格模式下会报错

34、写出执行结果，并说明原因

```
var a = [0];
if (a) {
  console.log(a == true);
} else {
  console.log(a);
}
```

分类：JavaScript

*答案：**false***

思路：

1）当 a 出现在 if 的条件中时，被转成布尔值，而 Boolean([0])为 true,所以就进行下一步判断 a == true,在进行比较时，[0]被转换成了 0，所以 0==true 为 false
数组从非 primitive 转为 primitive 的时候会先隐式调用 join 变成“0”，string 和 boolean 比较的时候，两个都先转为 number 类型再比较，最后就是 0==1 的比较了

```
var a = [1];
if (a) {
  console.log(a == true);
} else {
  console.log(a);
}
// true

!![] //true 空数组转换为布尔值是 true,
!![0]//true 数组转换为布尔值是 true
```

```
[0] == true;//false 数组与布尔值比较时却变成了 false
Number([])//0
Number(false)//0
Number(['1'])//1
```

2）所以当 a 出现在 if 的条件中时，被转成布尔值，而 Boolean([0])为 true,所以就进行下一步判断 a == true,在进行比较时，js 的规则是：
① 如果比较的是原始类型的值，原始类型的值会转成数值再进行比较

```
1 == true  //true   1 === Number(true)
'true' == true //false Number('true')->NaN  Number(true)->1
'' = 0//true
'1' == true//true  Number('1')->1
```

② 对象与原始类型值比较，对象会转换成原始类型的值再进行比较。
③ undefined和null与其它类型进行比较时，结果都为false，他们相互比较时结果为true。

> 数组从非 primitive 转为 primitive 的时候会先隐式调用 join 变成“0”
> 这一句解释应该有点问题，应该是隐式调用数组的toString方法。[0].toString() / / '0', 只不过这里和[0].join()产生一样的结果。
> [0].toString() 和 [0].join(',') 和 [0].join() 都返回 '0'
> 顺便说一下，非 primitive 都是通过调用自身的 valueOf、和toString 来进行隐式转换的。

35、写出执行结果，并说明原因

```
(function () {
  var a = (b = 5);
})();
console.log(b);
console.log(a);
```

分类：JavaScript

答案：5 Error，a is not defined

思路：

> 在这个立即执行函数表达式（IIFE）中包括两个赋值操作，其中a使用var关键字进行声明，因此其属于函数内部的局部变量（仅存在于函数中），相反，b被分配到全局命名空间。
> 另一个需要注意的是，这里没有在函数内部使用严格模式(use strict;)。如果启用了严格模式，代码会在输出 b 时报错Uncaught ReferenceError: b is not defined,需要记住的是，严格模式要求你显式的引用全局作用域。因此，你需要写成：
>
> ```
> (function () {
> "use strict";
> var a = (window.b = 5);
> })();
> console.log(b);
> ```

在看一个

```
(function() {
'use strict';
var a = b = 5;
})();

console.log(b);  //Uncaught ReferenceError: b is not defined

/*--------------------------*/

(function() {
'use strict';
var a = window.b = 5;
})();

console.log(b);  // 5
```

36、写出执行结果，并说明原因

```
var fullname = 'a';
var obj = {
   fullname: 'b',
   prop: {
      fullname: 'c',
      getFullname: function() {
         return this.fullname;
      }
   }
};

console.log(obj.prop.getFullname()); // c
var test = obj.prop.getFullname;
console.log(test());  // a
```

分类：JavaScript

答案：c a

思路：

- 原因在于this指向的是函数的执行环境，this取决于其被谁调用了，而不是被谁定义了。
- 对第一个console.log()语句而言，getFullName()是作为obj.prop对象的一个方法被调用的，因此此时的执行环境应该是这个对象。另一方面，但getFullName()被分配给test变量时，此时的执行环境变成全局对象（window），原因是test是在全局作用域中定义的。因此，此时的this指向的是全局作用域的fullname变量，即a。

37、写出执行结果，并说明原因

```
var company = {
    address: 'beijing'
}
var yideng = Object.create(company);
delete yideng.address
console.log(yideng.address);
```

分类：JavaScript

答案：beijing

思路：

> 这里的 yideng 通过 prototype 继承了 company的 address。yideng自己并没有address属性。所以delete操作符的作用是无效的。

知识点：

> 1、delete使用原则：delete 操作符用来删除一个对象的属性。
> 2、delete在删除一个不可配置的属性时在严格模式和非严格模式下的区别:
> （1）在严格模式中，如果属性是一个不可配置（non-configurable）属性，删除时会抛出异常;
> （2）非严格模式下返回 false。
> 3、delete能删除隐式声明的全局变量：这个全局变量其实是global对象(window)的属性
> 4、delete能删除的：
> （1）可配置对象的属性（2）隐式声明的全局变量 （3）用户定义的属性 （4）在ECMAScript 6中，通过 const 或 let 声明指定的 "temporal dead zone" (TDZ) 对 delete 操作符也会起作用
> delete不能删除的：
> （1）显式声明的全局变量 （2）内置对象的内置属性 （3）一个对象从原型继承而来的属性
> 5、delete删除数组元素：
> （1）当你删除一个数组元素时，数组的 length 属性并不会变小，数组元素变成undefined
> （2）当用 delete 操作符删除一个数组元素时，被删除的元素已经完全不属于该数组。
> （3）如果你想让一个数组元素的值变为 undefined 而不是删除它，可以使用 undefined 给其赋值而不是使用 delete 操作符。此时数组元素是在数组中的
> 6、delete 操作符与直接释放内存（只能通过解除引用来间接释放）没有关系。
>
> 7.其它例子
> **(1) 下面代码输出什么?**
>
> ```
> var output = (function(x){
>  delete x;
>  return x;
> })(0);
> console.log(output);
> ```
>
> 答案：0，**delete** 操作符是将object的属性删去的操作。但是这里的 **x** 是并不是对象的属性， **delete** 操作符并不能作用。
>
> **(2) 下面代码输出什么?**

```
var x = 1;
var output = (function(){
 delete x;
 return x;
})();
console.log(output);
```

答案：输出是 1。delete 操作符是将object的属性删去的操作。但是这里的 x 是并不是对象的属性， delete 操作符并不能作用。

（3）下面代码输出什么?

```
x = 1;
var output = (function(){
 delete x;
 return x;
})();
console.log(output);
```

答案：报错 VM548:1 Uncaught ReferenceError: x is not defined,

（4）下面代码输出什么?

```
var x = { foo : 1};
var output = (function(){
 delete x.foo;
 return x.foo;
})();
console.log(output);
```

答案：输出是 undefined。x虽然是全局变量，但是它是一个object。delete作用在x.foo上，成功的将x.foo删去。所以返回undefined

38、写出执行结果，并说明原因

```
var foo = function bar(){ return 12; };
console.log(typeof bar());
```

分类：JavaScript

答案：输出是抛出异常，bar is not defined。

思路：

这种命名函数表达式函数只能在函数体内有效

```
var foo = function bar(){
 // foo is visible here
 // bar is visible here
 console.log(typeof bar()); // Work here :)
};
// foo is visible here
// bar is undefined here
```

39、写出执行结果，并说明原因

```
var x=1;
if(function f(){}){
    x += typeof f;
}
console.log(x)
```

分类：JavaScript

答案：1 undefined

思路：

> 条件判断为假的情况有：0，false，''，null，undefined，未定义对象。函数声明写在运算符中，其为true，但放在运算符中的函数声明在执行阶段是找不到的。另外，对未声明的变量执行typeOf不会报错，会返回undefined

40、写出执行结果，并说明原因

```
function f(){
      return f;
 }
console.log(new f() instanceof f);
```

分类：JavaScript

答案：false

思路：

> a instanceof b 用于检测a是不是b的实例。如果题目f中没有return f,则答案明显为true;而在本题中new f()其返回的结果为f的函数对象，其并不是f的一个实例。

```
function f(){}
console.log(new f() instanceof f);
// 答案：true
```

41、写出执行结果，并说明原因

```
var foo = {
        bar: function(){
            return this.baz;
        },
         baz:1
    }
console.log(typeof (f=foo.bar)());
```

分类：JavaScript

答案：undefined

思路：

这里并不是因为赋值给 f，相当于f()，所以this指向window的。
可以试试下面代码也会打印 undefined

```
var foo = {
     bar: function(){
         return this.baz;
     },
      baz:1
}
console.log(typeof (foo.bar=foo.bar)());
```

接下来从规范的角度帕努单这个this指向，可以分为以下几个步骤来进行：

1、计算 MemberExpression 的结果赋值给 ref

> MemberExpression 我们可以简单理解为括号前的部分，针对这题就是 (f=foo.bar) 这部分。

2、判断 ref 是不是一个 Reference 类型

> Reference 是规范类型的一种。如果通过 GetValue 方法从 Reference 类型中获取值，则该 MemberExpression 返回结果不再是 Reference 类型。
> 这里的关键就是判断 MemberExpression 的返回结果是不是 Reference 类型。
> 由于 f=foo.bar 存在赋值操作符，根据规范 11.13.1 Simple Assignment (=) 规定，其第三步使用了 GetValue(ref)，故返回值不是 Reference 类型。
> 对照上述 2.3 的规范，如果表达式返回值不是 Reference 类型，那么 this 的值为 undefined，在非严格模式下，被隐式转换为全局对象 window。

```
2.1 如果 ref 是 Reference，并且 IsPropertyReference(ref) 是 true，那么 this 的值为 GetBase(ref)

2.2 如果 ref 是 Reference，并且 base value 值是 Environment Record，那么this的值为 ImplicitThisValue(ref)

2.3 如果 ref 不是 Reference，那么 this 的值为 undefined
```

42、说一下React Hooks在平时开发中需要注意的问题和原因

分类：React

答案：

1）不要在循环，条件或嵌套函数中调用Hook，必须始终在React函数的顶层使用Hook

> 这是因为React需要利用调用顺序来正确更新相应的状态，以及调用相应的钩子函数。一旦在循环或条件分支语句中调用Hook，就容易导致调用顺序的不一致性，从而产生难以预料到的后果。

2）使用`useState`时候，使用push，pop，splice等直接更改数组对象的坑

> 使用push直接更改数组无法获取到新值，应该采用析构方式，但是在class里面不会有这个问题

```
function Indicatorfilter() {
let [num,setNums] = useState([0,1,2,3])
const test = () => {
 // 这里坑是直接采用push去更新num
 // setNums(num)是无法更新num的
 // 必须使用num = [...num ,1]
 num.push(1)
 // num = [...num ,1]
 setNums(num)
}
return (
 <div className='filter'>
   <div onClick={test}>测试</div>
     <div>
       {num.map((item,index) => (
           <div key={index}>{item}</div>
       ))}
   </div>
 </div>
)
}

class Indicatorfilter extends React.Component<any,any>{
constructor(props:any){
   super(props)
   this.state = {
       nums:[1,2,3]
   }
   this.test = this.test.bind(this)
}

test(){
   // class采用同样的方式是没有问题的
   this.state.nums.push(1)
   this.setState({
       nums: this.state.nums
   })
```

```
}

render(){
   let {nums} = this.state
   return(
       <div>
           <div onClick={this.test}>测试</div>
               <div>
                   {nums.map((item:any,index:number) => (
                       <div key={index}>{item}</div>
                   ))}
               </div>
       </div>

   )
}
}
```

3）useState设置状态的时候，只有第一次生效，后期需要更新状态，必须通过useEffect

> TableDeail是一个公共组件，在调用它的父组件里面，我们通过set改变columns的值，以为传递给TableDeail的columns是最新的值，所以tabColumn每次也是最新的值，但是实际tabColumn是最开始的值，不会随着columns的更新而更新

```
const TableDeail = ({
 columns,
}:TableData) => {
 const [tabColumn, setTabColumn] = useState(columns)
}

// 正确的做法是通过useEffect改变这个值
const TableDeail = ({
 columns,
}:TableData) => {
 const [tabColumn, setTabColumn] = useState(columns)
 useEffect(() =>{setTabColumn(columns)},[columns])
}
```

4）善用useCallback

> 父组件传递给子组件事件句柄时，如果我们没有任何参数变动可能会选用useMemo。但是每一次父组件渲染子组件即使没变化也会跟着渲染一次。

5）不要滥用useContext

> 可以使用基于useContext封装的状态管理工具。

43、Vue组件中写name选项有除了搭配keep-alive还有其他作用么？你能谈谈你对keep-alive了解么？（平时使用和源码实现方面）

分类：Vue

答案：

1）组件中写 name 选项有什么作用？

```
1、项目使用 keep-alive 时，可搭配组件 name 进行缓存过滤
2、 DOM 做递归组件时需要调用自身 name
3、vue-devtools 调试工具里显示的组见名称是由vue中组件name决定的
```

2）二、keep-alive使用

```
1、keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，避免重新渲染
2、一般结合路由和动态组件一起使用，用于缓存组件；
3、提供 include 和 exclude 属性，两者都支持字符串或正则表达式， include 表示只有名称匹配的组件会被缓存，exclude 表示任何名称匹配的组件都不会被缓存 ，其中 exclude 的优先级比 include 高；
4、对应两个钩子函数 activated 和 deactivated ，当组件被激活时，触发钩子函数 activated，当组件被移除时，触发钩子函数 deactivated。
```

3）keep-alive实现原理

① 先了解下源码：

```
// 源码位置：src/core/components/keep-alive.js
export default {
  name: 'keep-alive',
  abstract: true, // 判断当前组件虚拟dom是否渲染成真是dom的关键

  props: {
    include: patternTypes, // 缓存白名单
    exclude: patternTypes, // 缓存黑名单
    max: [String, Number] // 缓存的组件实例数量上限
  },

  created () {
    this.cache = Object.create(null) // 缓存虚拟dom
    this.keys = [] // 缓存的虚拟dom的健集合
  },

  destroyed () {
    for (const key in this.cache) { // 删除所有的缓存
      pruneCacheEntry(this.cache, key, this.keys)
    }
  },

  mounted () {
    // 实时监听黑白名单的变动
```

```
    this.$watch('include', val => {
      pruneCache(this, name => matches(val, name))
    })
    this.$watch('exclude', val => {
      pruneCache(this, name => !matches(val, name))
    })
  },

  render () {
    // .....
  }
}
```

大概的分析源码，我们发现与我们定义组件的过程一样，先是设置组件名为keep-alive，其次定义了一个abstract属性，值为true。这个属性在vue的官方教程并未提及，其实是一个虚组件，后面渲染过程会利用这个属性。props属性定义了keep-alive组件支持的全部参数。

② 接下来重点就是keep-alive在它生命周期内定义了三个钩子函数了

created

初始化两个对象分别缓存VNode（虚拟DOM）和VNode对应的键集合

destroyed

删除缓存VNode还要对应执行组件实例的destory钩子函数。

删除this.cache中缓存的VNode实例。不是简单地将this.cache置为null，而是遍历调用pruneCacheEntry函数删除。

```
// src/core/components/keep-alive.js
function pruneCacheEntry (
  cache: VNodeCache,
  key: string,
  keys: Array<string>,
  current?: VNode
) {
  const cached = cache[key]
  if (cached && (!current || cached.tag !== current.tag)) {
    cached.componentInstance.$destroy() // 执行组件的destory钩子函数
  }
  cache[key] = null
  remove(keys, key)
}
```

mounted

在mounted这个钩子中对include和exclude参数进行监听，然后实时地更新（删除）this.cache对象数据。pruneCache函数的核心也是去调用pruneCacheEntry。

③ render

```
// src/core/components/keep-alive.js
render () {
const slot = this.$slots.default
const vnode: VNode = getFirstComponentChild(slot) // 找到第一个子组件对象
const componentOptions: ?VNodeComponentOptions = vnode && vnode.componentOptions
if (componentOptions) { // 存在组件参数
    // check pattern
    const name: ?string = getComponentName(componentOptions) // 组件名
    const { include, exclude } = this
    if ( // 条件匹配
    // not included
    (include && (!name || !matches(include, name))) ||
    // excluded
    (exclude && name && matches(exclude, name))
    ) {
    return vnode
    }

    const { cache, keys } = this
    const key: ?string = vnode.key == null // 定义组件的缓存key
    // same constructor may get registered as different local components
    // so cid alone is not enough (#3269)
    ? componentOptions.Ctor.cid + (componentOptions.tag ? `::${componentOptions.tag}` : '')
    : vnode.key
    if (cache[key]) { // 已经缓存过该组件
    vnode.componentInstance = cache[key].componentInstance
    // make current key freshest
    remove(keys, key)
    keys.push(key) // 调整key排序
    } else {
    cache[key] = vnode // 缓存组件对象
    keys.push(key)
    // prune oldest entry
    if (this.max && keys.length > parseInt(this.max)) {
        // 超过缓存数限制，将第一个删除（LRU缓存算法）
        pruneCacheEntry(cache, keys[0], keys, this._vnode)
    }
    }

    vnode.data.keepAlive = true // 渲染和执行被包裹组件的钩子函数需要用到
}
return vnode || (slot && slot[0])
}
```

- 第一步：获取keep-alive包裹着的第一个子组件对象及其组件名；
- 第二步：根据设定的黑白名单（如果有）进行条件匹配，决定是否缓存。不匹配，直接返回组件实例（VNode），否则执行第三步；
- 第三步：根据组件ID和tag生成缓存Key，并在缓存对象中查找是否已缓存过该组件实例。如果存在，直接取出缓存值并更新该key在this.keys中的位置（更新key的位置是实现LRU置换策略的关键），否则执行第四步；

- 第四步：在this.cache对象中存储该组件实例并保存key值，之后检查缓存的实例数量是否超过max的设置值，超过则根据LRU置换策略删除最近最久未使用的实例（即是下标为0的那个key）。
- 第五步：最后并且很重要，将该组件实例的keepAlive属性值设置为true。

最后就是再次渲染执行缓存和对应钩子函数了

44、Vue 为什么要用 vm.$set() 解决对象新增属性不能响应的问题？ 你能说说如下代码的实现原理么

```
Vue.set (object, propertyName, value)
vm.$set (object, propertyName, value)
```

分类：Vue

答案：

1）Vue为什么要用vm.$set() 解决对象新增属性不能响应的问题

```
1. Vue使用了Object.defineProperty实现双向数据绑定
2. 在初始化实例时对属性执行 getter/setter 转化
3. 属性必须在data对象上存在才能让Vue将它转换为响应式的（这也就造成了Vue无法检测到对象属性的添加或删除）
```

所以Vue提供了Vue.set (object, propertyName, value) / vm.$set (object, propertyName, value)

2）接下来我们看看框架本身是如何实现的呢?

> Vue 源码位置：vue/src/core/instance/index.js

```
export function set (target: Array<any> | Object, key: any, val: any): any {
  // target 为数组
  if (Array.isArray(target) && isValidArrayIndex(key)) {
    // 修改数组的长度，避免索引>数组长度导致splcie()执行有误
    target.length = Math.max(target.length, key)
    // 利用数组的splice变异方法触发响应式
    target.splice(key, 1, val)
    return val
  }
  // key 已经存在，直接修改属性值
  if (key in target && !(key in Object.prototype)) {
    target[key] = val
    return val
  }
  const ob = (target: any).__ob__
  // target 本身就不是响应式数据，直接赋值
  if (!ob) {
    target[key] = val
    return val
  }
  // 对属性进行响应式处理
```

```
  defineReactive(ob.value, key, val)
  ob.dep.notify()
  return val
}
```

看到源码后可知，vm.$set的实现原理是：

```
1、如果目标是数组，直接使用数组的 splice 方法触发相应式；
2、如果目标是对象，会先判读属性是否存在、对象是否是响应式，
3、最终如果要对属性进行响应式处理，则是通过调用 defineReactive 方法进行响应式处理
```

> defineReactive 方法就是 Vue 在初始化对象时，给对象属性采用 Object.defineProperty 动态添加 getter 和 setter 的功能所调用的方法

45、既然 Vue 通过数据劫持可以精准探测数据在具体dom上的变化,为什么还需要虚拟 DOM diff 呢

分类：Vue

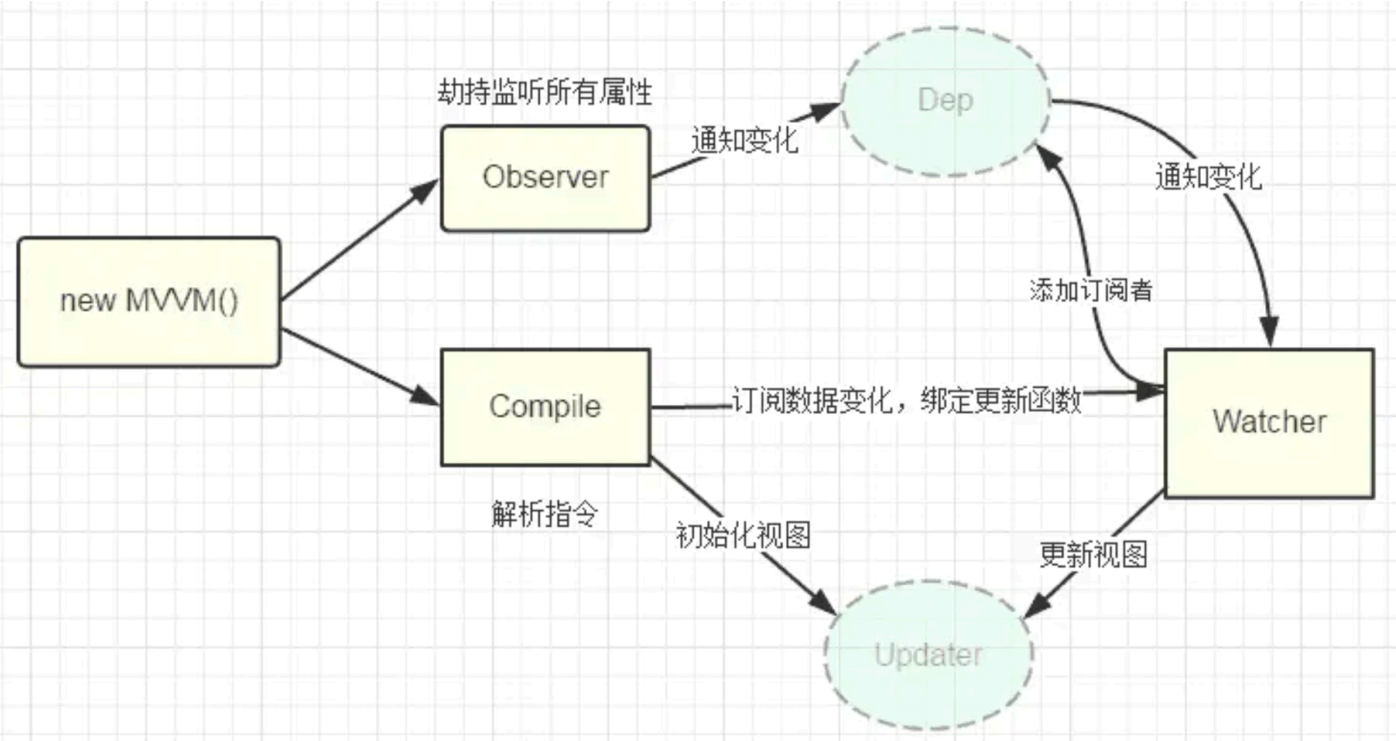
答案：

前置知识: 依赖收集、虚拟 DOM、响应式系统

现代前端框架有两种方式侦测变化，一种是 **pull** ，一种是 **push**

pull: 其代表为React，我们可以回忆一下React是如何侦测到变化的,我们通常会用setStateAPI显式更新，然后React会进行一层层的Virtual Dom Diff操作找出差异，然后Patch到DOM上，React从一开始就不知道到底是哪发生了变化，只是知道「有变化了」，然后再进行比较暴力的Diff操作查找「哪发生变化了」，另外一个代表就是Angular的脏检查操作。

push: Vue的响应式系统则是push的代表，当Vue程序初始化的时候就会对数据data进行依赖的收集，一但数据发生变化,响应式系统就会立刻得知。因此Vue是一开始就知道是「在哪发生变化了」，但是这又会产生一个问题，如果你熟悉Vue的响应式系统就知道，通常一个绑定一个数据就需要一个Watcher



> 也就是说一但我们的绑定细粒度过高就会产生大量的Watcher，这会带来内存以及依赖追踪的开销，而细粒度过低会无法精准侦测变化,因此Vue的设计是选择中等细粒度的方案,在组件级别进行push侦测的方式,也就是那套响应式系统,通常我们会第一时间侦测到发生变化的组件,然后在组件内部进行Virtual Dom Diff获取更加具体的差异，而Virtual Dom Diff则是pull操作，Vue是push+pull结合的方式进行变化侦测的。

46、关于Vue.js虚拟DOM的优缺点说法正确的是？（多选）

> - A.可以保证性能下限，比起粗暴的 DOM 操作性能要好很多，因此框架的虚拟 DOM 至少可以保证在你不需要手动优化的情况下，依然可以提供还不错的性能，即保证性能的下限；
>
> - B.无需手动操作DOM，不再需要手动去操作 DOM，只需要写好 View-Model 的代码逻辑，框架会根据虚拟 DOM 和数据双向绑定，帮我们以可预期的方式更新视图，极大提高我们的开发效率；
>
> - C.可以进行极致优化： 虚拟 DOM + 合理的优化，可以使性能达到极致
>
> - D.可以跨平台，虚拟 DOM 本质上是 JavaScript 对象,而 DOM 与平台强相关，相比之下虚拟 DOM 可以进行更方便地跨平台操作，例如服务器渲染、weex 开发等等。

分类：Vue

答案：A B D

思路：

> 1）优点
>
> - 保证性能下限： 框架的虚拟 DOM 需要适配任何上层 API 可能产生的操作，它的一些 DOM 操作的实现必须是普适的，所以它的性能并不是最优的；但是比起粗暴的 DOM 操作性能要好很多，因此框架的虚拟 DOM 至少可以保证在你不需要手动优化的情况下，依然可以提供还不错的性能，即保证性能的下限；
> - 无需手动操作 DOM： 我们不再需要手动去操作 DOM，只需要写好 View-Model 的代码逻辑，框架会根据虚拟 DOM 和 数据双向绑定，帮我们以可预期的方式更新视图，极大提高我们的开发效率；

- 跨平台： 虚拟 DOM 本质上是 JavaScript 对象,而 DOM 与平台强相关，相比之下虚拟 DOM 可以进行更方便地跨平台操作，例如服务器渲染、weex 开发等等。

2）缺点

无法进行极致优化： 虽然虚拟 DOM + 合理的优化，足以应对绝大部分应用的性能需求，但在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化。比如VScode采用直接手动操作DOM的方式进行极端的性能优化

47、写出执行结果，并说明原因

```
for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1);
}
```

分类：JavaScript

答案：0 1 2

思路：

使用let关键字声明变量i：使用let（和const）关键字声明的变量是具有块作用域的（块是{}之间的任何东西）。在每次迭代期间，i将被创建为一个新值，并且每个值都会存在于循环内的块级作用域。

```
// 下面代码输出什么
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1);
}
```

答案：3 3 3，由于JavaScript中的事件执行机制，setTimeout函数真正被执行时，循环已经走完。 由于第一个循环中的变量i是使用var关键字声明的，因此该值是全局的。 在循环期间，我们每次使用一元运算符++都会将i的值增加1。 因此在第一个例子中，当调用setTimeout函数时，i已经被赋值为3。

48、写出执行结果，并说明原因

```
const num = {
  a: 10,
  add() {
    return this.a + 2;
  },
  reduce: () => this.a -2;
};
console.log(num.add());
console.log(num.reduce());
```

分类：JavaScript

答案：12 NaN

思路：

> 注意，add是普通函数，而reduce是箭头函数。对于箭头函数，this关键字指向是它所在上下文（定义时的位置）的环境，与普通函数不同！这意味着当我们调用reduce时，它不是指向num对象，而是指其定义时的环境（window）。没有值a属性，返回undefined。

49、写出执行结果，并说明原因

```
const person = { name: "yideng" };

function sayHi(age) {
  return `${this.name} is ${age}`;
}
console.log(sayHi.call(person, 5));
console.log(sayHi.bind(person, 5));
```

分类：JavaScript

*答案：yideng is 5 f sayHi(age){return **${this.name} is ${age};**}*

思路：

> 使用两者，我们可以传递我们想要this关键字引用的对象。但是，.call方法会立即执行！
> .bind方法会返回函数的拷贝值，但带有绑定的上下文！它不会立即执行。

50、手写实现 Array.flat()

公司：滴滴、快手、携程　　分类：JavaScript

```
function flat(arr) {
  let res = []
  arr.map(item=>
    Array.isArray(item) ? res = res.concat(flat(item)) : res.push(item)
  )
 return res
}
```