面试题押题总结

HTML 押题

HTML 5 有哪些新标签?

文章相关：header main footer nav section article figure mark

多媒体相关：video audio svg canvas

表单相关：type=email type=tel

> MDN 把所有标签都列在 这里 了，且有教程

Canvas 和 SVG 的区别是什么?

答题思路为：先说一，再说二，再说相同点，最后说不同点。

1. Canvas 主要是用笔刷来绘制 2D 图形的。
2. SVG 主要是用标签来绘制不规则矢量图的。
3. 相同点：都是主要用来画 2D 图形的。
4. 不同点：Canvas 画的是**位图**，SVG 画的是*矢量图*。
5. 不同点：SVG 节点过多时**渲染慢**，Canvas 性能更好一点，但写起来更复杂。
6. 不同点：SVG 支持分层和事件，Canvas 不支持，但是可以用库实现。

> 得分点：位图 v.s. 矢量图、渲染性能、是否支持分层和事件……

如何理解 HTML 中的语义化标签

答法如下：

- 让人更容易读懂（增加代码可读性）。

- 让搜索引擎更容易读懂，有助于爬虫抓取更多的有效信息，爬虫依赖于标签来确定上下文和各个关键字的权重（SEO）。
- 在没有 CSS 样式下，页面也能呈现出很好地内容结构、代码结构。

1. 是什么：语义化标签是一种写 HTML 标签的**方法论**/方式。
2. 怎么做：实现方法是遇到标题就用 h1 到 h6，遇到段落用 p，遇到文章用 article，主要内容用 main，边栏用 aside，导航用 nav……（就是找到中文对应的英文）
3. 解决了什么问题：明确了 HTML 的书写规范

> 总结：「是什么、怎么做、解决了什么问题、优点是、缺点是、怎么解决缺点」

Script 标签中 defer 和 async 的区别?

- `script` ：会阻碍 HTML 解析，只有下载好并执行完脚本才会继续解析 HTML。
- `async script` ：解析 HTML 过程中进行脚本的异步下载，下载成功立马执行，有可能会阻断 HTML 的解析。
- `defer script`：完全不会阻碍 HTML 的解析，解析完成之后再按照顺序执行脚本。

图解 script 标签中的 async 和 defer 属性

CSS 押题

说一下浮动

从三个方面回答：1、浮动的作用：常用于图片，可以实现文字环绕图片。

2、浮动的特点：脱离文档流，容易造成盒子塌陷，影响其他元素的排列。

3、解决塌陷问题： **流行用法**：

- 父元素中添加overflow:hidden
- 给父元素添加高度、建立空白标签
- 添加clear
- 或者在父级添加伪元素

less 复制代码

```
::after{
    content:'',
```

```
    clear:both,
    display:table
-}
```

如何清除浮动?

实践题，建议写博客，甩链接。

方法一，给父元素加上 .clearfix

```css
.clearfix:after{
     content: '';
     display: block; /*或者 table*/
     clear: both;
 }
 .clearfix{
     zoom: 1; /* IE 兼容*/
 }
```

方法二，给父元素加上 overflow:hidden。

BFC 是什么

答题思路还是「是什么、怎么做、解决了什么问题、优点是、缺点是、怎么解决缺点」

是什么:

避免回答，直接把 BFC 翻译成中文「**块级格式化上下文，独立的渲染区域**」即可，千万别解释。

怎么做:

背诵 BFC 触发条件，虽然 MDN 的这篇文章 列举了所有触发条件，但本押题告诉你只用背这几个就行了

- 浮动元素（元素的 float 不是 none）
- 绝对定位元素（元素的 position 为 absolute 或 fixed）
- 行内块 inline block 元素
- overflow 值不为 visible 的块元素
- 弹性元素（display为 flex 或 inline-flex元素的直接子元素）

解决了什么问题：

1. 清除浮动（为什么不用 .clearfix 呢？）
2. 防止 margin 合并
3. 某些古老的布局方式会用到（已过时）

优点： 无。

缺点： 有副作用。

怎么解决缺点： 使用最新的 `display: flow-root` 来触发 BFC 就没有副作用了，但是很多人不知道。

如何实现垂直居中?

1. 利用绝对定位，设置 `left: 50%` 和 `top: 50%` 现将子元素左上角移到父元素中心位置，然后再通过 `translate` 来调整子元素的中心点到父元素的中心。该方法可以**不定宽高**。

css 复制代码

```
.father {
  position: relative;
}
.son {
  position: absolute;
  left: 50%;
  top: 50%;
  transform: translate(-50%, -50%);
}
```

2. 利用绝对定位，子元素所有方向都为 `0` ，将 `margin` 设置为 `auto` ，由于宽高固定，对应方向实现平分，该方法必须**盒子有宽高**。

css 复制代码

```
.father {
  position: relative;
}
.son {
  position: absolute;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0px;
  margin: auto;
  height: 100px;
```

```
  width: 100px;
}
```

3. 利用绝对定位，设置 `left: 50%` 和 `top: 50%` 现将子元素左上角移到父元素中心位置，然后再通过 `margin-left` 和 `margin-top` 以子元素自己的一半宽高进行负值赋值。该方法**必须定宽高**。

css 复制代码

```
.father {
  position: relative;
}
.son {
  position: absolute;
  left: 50%;
  top: 50%;
  width: 200px;
  height: 200px;
  margin-left: -100px;
  margin-top: -100px;
}
```

4. 利用 `flex` ，最经典最方便的一种了，不用解释，定不定宽高无所谓的。

css 复制代码

```
.father {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

其实还有很多方法，比如 `display: grid` 或 `display: table-cell` 来做，有兴趣点击下面这篇文章可以了解下：

如果 .parent 的 height 写死了，就很难把 .child 居中，以下是垂直居中的方法。

忠告：能不写 height 就千万别写 height。

1. table自带功能
2. 100% 高度的 afrer before 加上 inline block
   这个方法还有一个 优化版本
3. div 装成 table
4. margin-top -50%

5. translate -50%
6. absolute margin auto
7. flex

> 得分点：flex 方案、grid 方案、transform 方案.....

CSS 选择器优先级如何确定?

属性赋值，层叠（Cascading）和继承 (ayqy.net)

样式的优先级一般为 `!important > style(内联样式) > id(选择器） > class（选择器）>标签选择器`

如果记不住，可以记下这三句话：

1. 选择器越具体，其优先级越高
2. 相同优先级，出现在后面的，覆盖前面的
3. 属性后面加 !important 的优先级最高，但是要少用

说一说CSS尺寸设置的单位

1. Px：绝对像素
2. Rem：相对于根元素像素
3. Em：相对于父元素像素 1em 等于当前元素的字号，其准确值取决于作用的元素.（若字号为16px, 则 1em = 16px）
4. Vw：视口宽度
5. Vh：视口高度

两种盒模型（box-sizing）的区别?

答题思路为：先说一，再说二，再说相同点，最后说不同点。

第一种盒模型是 content-box，即 width 指定的是 content 区域宽度，而不是实际宽度，公式为

> 实际宽度 = width + padding + border

第二种盒模型是 border-box，即 width 指定的是左右边框外侧的距离，公式为

> 实际宽度 = width

相同点是都是用来指定宽度的，不同点是 border-box 更好用。

一点有意思的小历史，请看视频。

实现两栏布局（左侧固定 + 右侧自适应布局）

现在有以下 DOM 结构：

ini 复制代码

```ini
<div class="outer">
  <div class="left">左侧</div>
  <div class="right">右侧</div>
</div>
```

1. 利用浮动，左边元素宽度固定，设置向左浮动。将右边元素的 `margin-left` 设为固定宽度。注意，因为右边元素的 `width` 默认为 `auto`，所以会自动撑满父元素。

css 复制代码

```css
.outer {
  height: 100px;
}
.left {
  float: left;
  width: 200px;
  height: 100%;
  background: lightcoral;
}
.right {
  margin-left: 200px;
  height: 100%;
  background: lightseagreen;
}
```

2. 同样利用浮动，左边元素宽度固定，设置向左浮动。右侧元素设置 `overflow: hidden;` 这样右边就触发了 `BFC`，`BFC` 的区域不会与浮动元素发生重叠，所以两侧就不会发生重叠。

```css
.outer {
  height: 100px;
}
.left {
  float: left;
  width: 200px;
  height: 100%;
  background: lightcoral;
}
.right {
  overflow: auto;
  height: 100%;
  background: lightseagreen;
}
```

3. 利用 `flex` 布局，左边元素固定宽度，右边的元素设置 `flex: 1` 。

```css
.outer {
  display: flex;
  height: 100px;
}
.left {
  width: 200px;
  height: 100%;
  background: lightcoral;
}
.right {
  flex: 1;
  height: 100%;
  background: lightseagreen;
}
```

4. 利用绝对定位，父级元素设为相对定位。左边元素 `absolute` 定位，宽度固定。右边元素的 `margin-left` 的值设为左边元素的宽度值。

```css
.outer {
  position: relative;
  height: 100px;
}
.left {
  position: absolute;
  width: 200px;
  height: 100%;
  background: lightcoral;
}
.right {
```

```
  margin-left: 200px;
  height: 100%;
  background: lightseagreen;
}
```

5. 利用绝对定位，父级元素设为相对定位。左边元素宽度固定，右边元素 `absolute` 定位，`left` 为宽度大小，其余方向定位为 `0` 。

css 复制代码

```
.outer {
  position: relative;
  height: 100px;
}
.left {
  width: 200px;
  height: 100%;
  background: lightcoral;
}
.right {
  position: absolute;
  left: 200px;
  top: 0;
  right: 0;
  bottom: 0;
  height: 100%;
  background: lightseagreen;
}
```

实现圣杯布局和双飞翼布局（经典三分栏布局）

圣杯布局和双飞翼布局的目的：

- 三栏布局，中间一栏最先加载和渲染（**内容最重要，这就是为什么还需要了解这种布局的原因**）。
- 两侧内容固定，中间内容随着宽度自适应。
- 一般用于 PC 网页。

圣杯布局和双飞翼布局的技术总结：

- 使用 `float` 布局。
- 两侧使用 `margin` 负值，以便和中间内容横向重叠。
- 防止中间内容被两侧覆盖，圣杯布局用 `padding` ，双飞翼布局用 `margin` 。

圣杯布局： HTML 结构：

ini 复制代码

```ini
<div id="container" class="clearfix">
  <p class="center">我是中间</p>
  <p class="left">我是左边</p>
  <p class="right">我是右边</p>
</div>
```

CSS 样式：

css 复制代码

```css
#container {
  padding-left: 200px;
  padding-right: 150px;
  overflow: auto;
}
#container p {
  float: left;
}
.center {
  width: 100%;
  background-color: lightcoral;
}
.left {
  width: 200px;
  position: relative;
  left: -200px;
  margin-left: -100%;
  background-color: lightcyan;
}
.right {
  width: 150px;
  margin-right: -150px;
  background-color: lightgreen;
}
.clearfix:after {
  content: "";
  display: table;
  clear: both;
}
```

双飞翼布局： HTML 结构：

css 复制代码

```css
<div id="main" class="float">
  <div id="main-wrap">main</div>
</div>
```

```
<div id="left" class="float">left</div>
<div id="right" class="float">right</div>
```

CSS 样式：

```css
.float {
  float: left;
}
#main {
  width: 100%;
  height: 200px;
  background-color: lightpink;
}
#main-wrap {
  margin: 0 190px 0 190px;
}
#left {
  width: 190px;
  height: 200px;
  background-color: lightsalmon;
  margin-left: -100%;
}
#right {
  width: 190px;
  height: 200px;
  background-color: lightskyblue;
  margin-left: -190px;
}
```

Tips：上述代码中 `margin-left: -100%` 相对的是父元素的 `content` 宽度，即不包含 `paddig` 、 `border` 的宽度。

Flex 布局

这一块内容看 Flex 布局教程 就够了。

当一个容器设置 `display:flex` 变成一个 `flex` 容器后，如果容器没有被占满，换言之有剩余空间，则 `flex-grow` 起作用。
相反，若空间不足，则 `flex-shrink` 起作用。
在计算放大或缩小比例时，要根据 `flex-basis` 的值来计算比例。 这里有个小问题，很多时候我们会用到 `flex: 1` ，它具体包含了以下的意思：

- `flex-grow: 1` ：该属性默认为 `0` ，如果存在剩余空间，元素也不放大。设置为 `1` 代表会放大。
- `flex-shrink: 1` ：该属性默认为 `1` ，如果空间不足，元素缩小。
- `flex-basis: 0%` ：该属性定义在分配多余空间之前，元素占据的主轴空间。浏览器就是根据这个属性来**计算是否有多余空间**的。默认值为 `auto` ，即项目本身大小。设置为 `0%` 之后，因为有 `flex-grow` 和 `flex-shrink` 的设置会自动放大或缩小。在做两栏布局时，如果右边的自适应元素 `flex-basis` 设为 `auto` 的话，其本身大小将会是 `0` 。

Line-height 如何继承?

- 父元素的 `line-height` 写了**具体数值**，比如 `30px` ，则子元素 `line-height` 继承该值。
- 父元素的 `line-height` 写了**比例**，比如 `1.5` 或 `2` ，则子元素 `line-height` 也是继承该比例。
- 父元素的 `line-height` 写了**百分比**，比如 `200%` ，则子元素 `line-height` 继承的是父元素 `font-size * 200%` 计算出来的值。

JS押题

Es6中箭头函数

1. 没有自己的this，即不能作为构造函数，里面的this是执行上下文的this
2. 不能被new
3. 没有arguments对象
4. 不能作为generator函数，不能用yied命令
5. 没有prototype属性

JS 的数据类型有哪些?

纯记忆题，答案有 8 个词，建议背诵 10 次。

字符串、数字、布尔、undefined、null、大整数、符号、对象

String、number、boolean、undefined、null、bigint、symbol、object

> 提了就零分的答案有：数组、函数、日期。这些是类 class，不是类型 type

说一说Null 和 undefined的区别，如何让一个属性变为null。

Undefined 表示一个变量自然的、最原始的状态值，而 null 则表示一个变量被人为的设置为空对象，而不是原始状态；让一个属性变为null，只需要定义一个变量，然后直接赋值为 null 即可

原型链是什么?

- 原型：每一个 JavaScript 对象（null 除外）在创建的时候就会与之关联另一个对象，这个对象就是我们所说的原型，每一个对象都会从原型"继承"属性，其实就是 `prototype` 对象。
  假设我们有一个数组对象 `a=[]` ，这个 `a` 也会有一个隐藏属性，叫做_proto_这个属性会指向 `Array.prototype`

ini 复制代码

```ini
var a = [];
a.__proto__ ===  Array.prototype;
// 用 b表示 Array.prototype
b._proto_ === Object.prototype
```

于是就通过隐藏属性 `__proto__` 形成了一个链条：

javascript 复制代码

```javascript
a ===> Array.prototype ===> Object.prototype
```

这就是原型链。 **怎么做：**

看起来只要改写 b 的隐藏属性 `__proto__` 就可以改变 b的原型（链）

javascript 复制代码

```javascript
const x = Object.create(原型)
// 或
const x = new 构造函数() // 会导致 x.__?????__ === 构造函数.prototype
```

这样一来，a 就既拥有 Array.prototype 里的属性，又拥有 Object.prototype 里的属性。 **解决了什么问题：**

在没有 Class 的情况下实现「继承」。以 `a ===> Array.prototype ===> Object.prototype` 为例，我们说：

1. a 是 Array 的实例，a 拥有 Array.prototype 里的属性
2. Array 继承了 Object（注意专业术语的使用）

3. a 是 Object 的间接实例，a 拥有 Object.prototype 里的属性

优点：

简单、优雅。

缺点：

跟 class 相比，不支持私有属性。

怎么解决缺点：

使用 class 呗。但 class 是 ES6 引入的，不被旧 IE 浏览器支持。

> 建议熟读这篇文章：
>
> JS 中 **proto** 和 prototype 存在的意义是什么?

JS有几种方法判断变量的类型

1. typeof只能判断基本类型 ）,不能判断数据类型：null和object
2. intanceof(根据原型链判断),原生数据类型不能判断
3. constructor.name(根据构造器判断)，不能判断null数据类型
4. Object.prototype.toString.call()（用Object的toString方法判断）所有类型数据都能判断,记住判断结果打印为：'[object Xxx]'

Map 和 forEach 的区别?

相同点：1.都能遍历数组 2.中途不能被break打断 3.函数中都有三个参数，当前遍历的元素，当前元素的索引，原数组。

不同：1.forEach没有返回值，也就是返回undefined，map会开辟新的一个内存空间，返回新的数组，这点也方便链式调用其他数组方法。 2.map的效率比forEach高

这段代码中的 this 是多少?

普通函数执行指向window，箭头函数中的this指向上一级作用域中的this。

ini 复制代码

```ini
var length = 4;
function callback() {
  console.log(this.length); // => 打印出什么？
}

const obj = {
  length: 5,
  method(callback) {
    callback();
  }
};

obj.method(callback, 1, 2);
```

> 建议熟读这篇文章：

This 的值到底是什么？一次说清楚

JS 的 new 做了什么？（数组去重都有哪些方法）

看下new的实现代码就知道了：

1. 创建临时对象/新对象： 创建了一个空对象
2. 绑定原型： 把空对象的原型指向构造函数的prototype
3. 指定 this = 临时对象
4. 执行构造函数： 执行构造函数并返回结果判断返回结果是否是空对象，如果是空对象返回新对象，否则就直接返回。

js 复制代码

```js
function myNew(context) {
    const obj = new Object();
    obj.__proto__ = context.prototype;
    const res = context.apply(obj, [...arguments].slice(1));
    return typeof res === "object" ? res : obj;
}
```

> 建议熟读这篇文章：

JS 的 new 到底是干什么的?

伪数组和数组的区别

1. 伪数组的特点：类型是object、不能使用数组方法、可以获取长度、可以使用for in遍历 2. 伪数组可以装换为数组的方法：
   a. Array.prototype.slice.call()
   b.Array.from()
   c. [...伪数组] \
2. 有哪些是伪数组：函数的参数arguments，Map和Set的keys()、values()和entires()

JS 的立即执行函数是什么?

概念题，「是什么、怎么做、解决了什么问题、优点是、缺点是、怎么解决缺点」

是什么:

声明一个匿名函数，然后立即执行它。这种做法就是立即执行函数。

怎么做:

```js
(function(){alert('我是匿名函数')} ())  // 用括号把整个表达式包起来
(function(){alert('我是匿名函数')}) ()  // 用括号把函数包起来
!function(){alert('我是匿名函数')}()    // 求反，我们不在意值是多少，只想通过语法检查。
+function(){alert('我是匿名函数')}()
-function(){alert('我是匿名函数')}()
~function(){alert('我是匿名函数')}()
void function(){alert('我是匿名函数')}()
new function(){alert('我是匿名函数')}()
var x = function(){return '我是匿名函数'}()
```

上面每一行代码都是一个立即执行函数。（举例法）

解决了什么问题:

在 ES6 之前，只能通过它来「创建局部作用域」。

优点:

兼容性好。

缺点：

丑。为什么这么丑？看视频分析。

怎么解决缺点：

使用 ES6 的 block + let 语法，即

```javascript 复制代码
{
  let a = '我是局部变量啦'
  console.log(a) // 能读取 a
}
console.log(a) // 找不到 a
```

JS 的闭包是什么？怎么用？

概念题，「是什么、怎么做、解决了什么问题、优点是、缺点是、怎么解决缺点」

是什么

闭包是指那些能够访问自由变量的函数，当所有函数被保存到外部时

> 闭包 = 函数 + 自由变量

怎么做

```csharp 复制代码
let count
function add (){ // 访问了外部变量的函数
  count += 1
}
```

把上面代码放在「非全局环境」里，就是闭包。

> 注意，闭包不是 count，闭包也不是 add，闭包是 count + add 组成的整体。

怎么制造一个「非全局环境」呢？答案是立即执行函数：

```php
const x = function (){
        var count
        function add (){ // 访问了外部变量的函数
          count += 1
        }
}()
```

但是这个代码什么用也没有，所以我们需要 `return add` ，即：

```php
const add2 = function (){
        var count
        return function add (){ // 访问了外部变量的函数
          count += 1
        }
}()
```

此时 add2 其实就是 add，我们可以调用 add2

```scss
add2()
// 相当于
add()
// 相当于
count += 1
```

至此，我们就实现了一个完整的「闭包的应用」。

> 注意：闭包 ≠ 闭包的应用，但面试官问你「闭包」的时候，你一定要答「闭包的应用」，这是规矩。

解决了什么问题：

1. 避免污染全局环境。（因为用的是局部变量）
2. 提供对局部变量的间接访问。（因为只能 count += 1 不能 count -= 1）
3. 维持变量，使其不被垃圾回收。

优点：

简单，好用。

缺点：

闭包**使用不当**可能造成内存泄露。

注意，重点是「使用不当」，不是闭包。

「闭包造成内存泄露」这句话以讹传讹很多年了，曾经旧版本 IE 的 bug 导致的问题，居然被传成这样了。

举例说明：

```csharp
function test() {
  var x = {name: 'x'};
  var y = {name: 'y', content: "-----这里很长，有一万三千五百个字符那么长----" }
  return function fn() {
    return x;
  };
}

const myFn = test() // myFn 就是 fn 了
const myX = myFn() // myX 就是 x 了
// 请问，y 会消失吗？
```

对于一个正常的浏览器来说，y 会在**一段时间**后自动消失（被垃圾回收器给回收掉）。

但旧版本的 IE 并不是正常的浏览器，所以是 IE 的问题。

当然，你可以说

> 君子不立于危墙之下，我们应该尽量少用闭包，因为有些浏览器对闭包的支持不够好

但你不可以说「闭包造成内存泄露」。对吗？

怎么解决缺点：

慎用，少用，不用。（我偏要用）

> 建议熟读这篇文章：

「每日一题」JS 中的闭包是什么？

说一说computed和watch的区别?

Computed是计算属性，依赖其他属性值，并且有缓存，只要他依赖的值发生改变了，下一次获取computed的值时才会重新计算computed的值；watch更多的是监听观察作用，支持异步，每当监听的数值发生变化时就会立即回调进行后续操作。

说一说call apply bind的作用和区别?

作用：改变this指向

区别：call立即执行，返回执行结果，第一个参数为this，后面陆续传入执行参数。

Apply 立即执行，返回执行结果，第一个参数为this，第二个参数为执行参数组成的数组。

Bind，返回一个函数，不会立即执行，第一个参数为this，后面陆续传入参数，支持参数柯里化

实现方法

Call

> Call() 方法在使用一个指定的 this 值和若干个指定的参数值的前提下调用某个函数或方法。

举个例子：

javascript 复制代码

```
var obj = {
  value: "vortesnail",
};

function fn() {
  console.log(this.value);
}

fn.call(obj); // vortesnail
```

复制代码

通过 `call` 方法我们做到了以下两点：

- `call` 改变了 this 的指向，指向到 `obj` 。

- `fn` 函数执行了。

那么如果我们自己写 `call` 方法的话，可以怎么做呢？我们先考虑改造 `obj` 。

javascript 复制代码

```
var obj = {
  value: "vortesnail",
  fn: function () {
    console.log(this.value);
  },
};

obj.fn(); // vortesnail
```

复制代码

这时候 this 就指向了 `obj` ，但是这样做我们手动给 `obj` 增加了一个 `fn` 属性，这显然是不行的，不用担心，我们执行完再使用对象属性的删除方法（delete）不就行了？

ini 复制代码

```
obj.fn = fn;
obj.fn();
delete obj.fn;
```

复制代码

根据这个思路，我们就可以写出来了：

javascript 复制代码

```
Function.prototype.myCall = function (context) {
  // 判断调用对象
  if (typeof this !== "function") {
    throw new Error("Type error");
  }
  // 首先获取参数
  let args = [...arguments].slice(1);
  let result = null;
  // 判断 context 是否传入，如果没有传就设置为 window
  context = context || window;
  // 将被调用的方法设置为 context 的属性
  // this 即为我们要调用的方法
  context.fn = this;
  // 执行要被调用的方法
  result = context.fn(...args);
  // 删除手动增加的属性方法
  delete context.fn;
  // 将执行结果返回
  return result;
```

```
};
```
复制代码

Apply

我们会了 `call` 的实现之后， `apply` 就变得很简单了，他们没有任何区别，除了传参方式。

```ini
Function.prototype.myApply = function (context) {
  if (typeof this !== "function") {
    throw new Error("Type error");
  }
  let result = null;
  context = context || window;
  // 与上面代码相比，我们使用 Symbol 来保证属性唯一
  // 也就是保证不会重写用户自己原来定义在 context 中的同名属性
  const fnSymbol = Symbol();
  context[fnSymbol] = this;
  // 执行要被调用的方法
  if (arguments[1]) {
    result = context[fnSymbol](...arguments[1]);
  } else {
    result = context[fnSymbol]();
  }
  delete context[fnSymbol];
  return result;
};
```
复制代码

Bind

`Bind` 返回的是一个函数，这个地方可以详细阅读这篇文章，讲的非常清楚： 解析 bind 原理，并手写 bind 实现 。

```javascript
Function.prototype.myBind = function (context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new Error("Type error");
  }
  // 获取参数
  const args = [...arguments].slice(1),
  const fn = this;
  return function Fn() {
    return fn.apply(
      this instanceof Fn ? this : context,
      // 当前的这个 arguments 是指 Fn 的参数
```

```
      args.concat(...arguments)
    );
  };
};
```

JS实现异步的方法?

1.promise 2. 定时器 3. 回调函数

重绘和回流（考察频率：中）

重排(回流）：当dom的变化影响元素的几何信息（元素的位置和尺寸大小），浏览器需要重新计算元素的几何属性，将其安放在界面的正确的位置。这个过程叫重排。

避免重排的方法是样式集中改变；使用absolute或fixed脱离文档流；使用gpu加速，transform。

- 重绘：当页面中元素样式的改变并不影响它在文档流中的位置时（例如：color、background-color、visibility等），浏览器会将新样式赋予给元素并重新绘制它，这个过程称为重绘。
- 回流：当Render Tree（DOM）中部分或全部元素的尺寸、结构、或某些属性发生改变时，浏览器重新渲染部分或全部文档的过程称为回流。
- 回流要比重绘消耗性能开支更大。
- 回流必将引起重绘，重绘不一定会引起回流。

如何避免：用定位脱离文档流，统一改变格式减少dom操作

Axios的拦截器原理及应用?

Axios拦截器分为响应和请求拦截器，请求拦截器 在请求发送前进行必要操作处理，例如添加统一cookie、请求体加验证、设置请求头等，相当于是对每个接口里相同操作的一个封装； 响应拦截器 同理，响应拦截器也是如此功能，只是在请求得到响应之后，对响应体的一些处理，通常是数据统一处理等，也常来判断登录失效等。

说一下fetch 请求方式?

Fetch是一种HTTP数据请求的方式。fetch()方法返回一个Promise解析Response来自Request显示状态（成功与否）的方法。

手写 AJAX

AJAX（Asynchronous JavaScript and XML），指的是通过 JavaScript 的异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。

1.创建 `XMLHttpRequest` 对象,创建一个异步调用对象.
2.创建一个新的 `HTTP` 请求,并指定该 `HTTP` 请求的方法、 `URL` 及验证信息.
3.设置响应 `HTTP` 请求状态变化的函数.
4.发送 `HTTP` 请求

记忆题，写博客吧

scss 复制代码

```
const ajax = (method, url, data, success, fail) => {
  var request = new XMLHttpRequest()
  request.open(method, url);
  request.onreadystatechange = function () {
    if(request.readyState === 4) {
      if(request.status >= 200 && request.status < 300 || request.status === 304) {
        success(request)
      }else{
        fail(request)
      }
    }
  };
  request.send();
}
```

Promise是什么与使用方法?

1. 概念：异步编程的一种解决方案，解决了地狱回调的问题
2. 使用方法：new Promise((resolve,reject) => {
   resolve（）；reject（）；
   })
3. 里面有多个resovle或者reject只执行第一个。如果第一个是resolve的话后面可以接.then查看成功消息。如果第一个是reject的话，.catch查看错误消息。

DOM 押题

请简述 DOM 事件模型

先经历从上到下的捕获阶段，再经历从下到上的冒泡阶段。

AddEventListener('click',fn,true/false) 第三个参数可以选择阶段。

可以使用 `event.stopPropagation()` 来阻止捕获或冒泡。

说一说事件循环Event loop，宏任务与微任务？

Js是单线程的，主线程在执行时会不断循环往复的从同步队列中读取任务，执行任务，当同步队列执行完毕后再从异步队列中依次执行。

宏任务与微任务都属于异步任务，再执行上微任务的优先级高于宏任务，因此每一次都会先执行完微任务在执行宏任务。

宏任务有定时器，Dom事件，ajax事件，

微任务有：promise的回调、MutationObserver 的回调 ,process.nextTick

手写事件委托

错误版（但是可能能通过面试）

```scss
ul.addEventListener('click', function(e){
    if(e.target.tagName.toLowerCase() === 'li'){
        fn()// 执行某个函数
    }
})
```

Bug 在于，如果用户点击的是 li 里面的 span，就没法触发 fn，这显然不对。

好处

1. 节省监听器
2. 实现动态监听

坏处

调试比较复杂，不容易确定监听者。

解决坏处

解决不了

高级版（不用背）

思路是点击 span 后，递归遍历 span 的祖先元素看其中有没有 ul 里面的 li。

```js
function delegate(element, eventType, selector, fn) {
        element.addEventListener(eventType, e => {
          let el = e.target
          while (!el.matches(selector)) {
            if (element === el) {
              el = null
              break
            }
            el = el.parentNode
          }
          el && fn.call(el, e, el)
        })
        return element
}

delete(ul, 'click', 'li', f1)
```

手写可拖曳 div

参考代码： jsbin.com/munuzureya/...

要点：

1. 注意监听范围，不能只监听 div
2. 不要使用 drag 事件，很难用。
3. 使用 transform 会比 top / left 性能更好，因为可以避免 reflow 和 repaint

HTTP 押题

浏览器垃圾回收机制？

有两种机制：
1、标记清除：对所有活动对象进行标记，清除阶段会将没有标记的对象清除；标记整理算法：

标记结束后，算法将活动对象压入内存一端，则需要清理的对象在边界，直接被清理掉就行。（效率低）

2、引用计数：将对象是否不再需要简化定义为有没有其他对象引用它，如果没有引用指向这个对象，则会被垃圾回收机制回收。（内存空间不连续）

GET 和 POST 的区别有哪些?

区别一：幂等性

1. 由于 GET 是读，POST 是写，所以 GET 是幂等的，POST 不是幂等的。
2. 由于 GET 是读，POST 是写，所以用浏览器打开网页会发送 GET 请求，想要 POST 打开网页要用 form 标签。
3. 由于 GET 是读，POST 是写，所以 GET 打开的页面刷新是无害的，POST 打开的页面刷新需要确认。
4. 由于 GET 是读，POST 是写，所以 GET 结果会被缓存，POST 结果不会被缓存。
5. 由于 GET 是读，POST 是写，所以 GET 打开的页面可被书签收藏，POST 打开的不行。

区别二：请求参数

1. 通常，GET 请求参数放在 url 里，POST 请求数据放在 body（消息体）里。（这里注意老师的讲解）
2. GET 比 POST 更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。（xjb 扯）
3. GET 请求参数放在 url 里是有长度限制的，而 POST 放在 body 里没有长度限制。（xjb 扯）

区别三：TCP packet

1. GET 产生一个 TCP 数据包；POST 产生两个或以上 TCP 数据包。

根据技术规格文档，GET 和 POST 最大的区别是语义；但面试官一般问的是实践过程中二者的区别，因此你需要了解服务器和浏览器对 GET 和 POST 的常见实现方法。

HTTP 缓存有哪些方案?

	缓存（强缓存）	内容协商（弱缓存）
HTTP 1.1	Cache-Control: max-age=3600 Etag: ABC	If-None-Match: ABC 响应状态码：304 或 200
HTTP 1.0	Expires: Wed, 21 Oct 2015 02:30:00 GMT Last-Modified: Wed, 21 Oct 2015 01:00:00 GMT	If-Modified-Since: Wed, 21 Oct 2015 01:00:00 GMT 响应状态码：304 或 200

面试官可能还会提到 `Pragma` ，但 MDN 已经明确不推荐使用它。

更详细的内容可以看我的课程《 全面攻克 Web 性能优化 》中的《 缓存与内容协商 》视频。

HTTP 和 HTTPS 的区别有哪些?

⚠

HTTPS = HTTP + SSL/TLS（安全层）

区别列表

1. HTTP 是明文传输的，不安全；HTTPS 是加密传输的，非常安全。
2. HTTP 使用 80 端口，HTTPS 使用 443 端口。
3. HTTP 较快，HTTPS 较慢。
4. HTTPS 的证书一般需要购买（但也有免费的），HTTP 不需要证书。

HTTPS 的细节可以看网上的博客，比较复杂，难以记忆，建议写博客总结一下。

图解SSL/TLS协议 - 阮一峰的网络日志 (ruanyifeng.com)

HTTPS原理以及握手阶段

HTTP/1.1 和 HTTP/2 的区别有哪些?

区别列表

1. HTTP/2 使用了**二进制传输**，而且将 head 和 body 分成**帧**来传输；HTTP/1.1 是字符串传输。

2. HTTP/2 支持**多路复用**，HTTP/1.1 不支持。多路复用简单来说就是一个 TCP 连接从单车道（不是单行道）变成了几百个双向通行的车道。
3. HTTP/2 可以**压缩 head**，但是 HTTP/1.1 不行。
4. HTTP/2 支持**服务器推送**，但 HTTP/1.1 不支持。（实际上没多少人用）

更详细的内容可以看我的课程《 全面攻克 Web 性能优化 》中的《 什么是多路复用 》视频。

从浏览器地址栏输入 url 到请求返回发生了什么

先阅读这篇科普性质的： 从 URL 输入到页面展现到底发生什么？ 先阅读篇文章： 从输入 URL 开始建立前端知识体系 。

- DNS 解析:将域名解析成 IP 地址
- TCP 连接：TCP 三次握手
- 发送 HTTP 请求
- 服务器处理请求并返回 HTTP 报文
- 浏览器解析渲染页面
- 断开连接：TCP 四次挥手

1. 输入 URL 后解析出协议、主机、端口、路径等信息，并构造一个 HTTP 请求。

- 强缓存。
- 协商缓存。

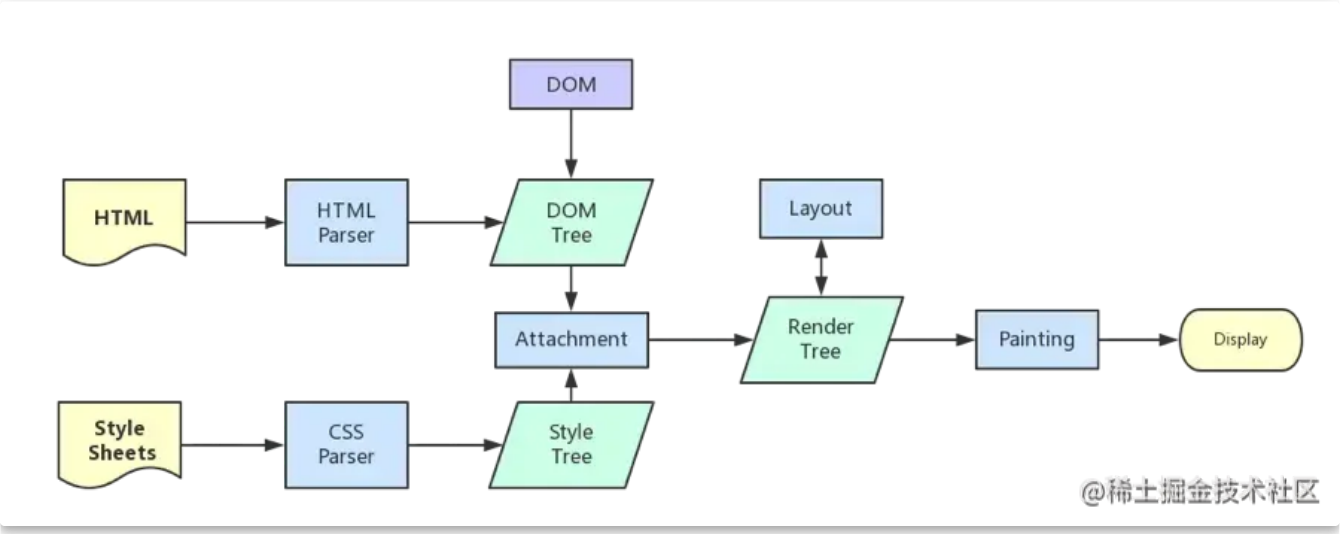
2. DNS 域名解析。DNS 的作用就是将**主机名转换成 IP 地址**。（ 字节面试被虐后，是时候搞懂 DNS 了 ）
3. TCP 连接。

> 总是要问：为什么需要三次握手，两次不行吗？其实这是由 TCP 的自身特点**可靠传输**决定的。客户端和服务端要进行可靠传输，那么就需要**确认双方的接收和发送能力**。第一次握手可以确认客服端的**发送能力**，第二次握手，确认了服务端的**发送能力和接收能力**，所以第三次握手才可以确认客户端的**接收能力**。不然容易出现丢包的现象。

4. Http 请求。

5. 服务器处理请求并返回 HTTP 报文。

6. 浏览器渲染页面。



7. 断开 TCP 连接。

TCP 三次握手和四次挥手是什么?

SYN	**请求号标记位**
ACK	确认号标记位
FIN	要求释放连接
seq	序号，代表请求方将会发送的数据的第一个字节编号
ack	返回的确认号，代表接收方收到收据后（也就是前面说的seq），代表希望对方下一次传输数据的第一个字节编号

建立 TCP 连接时 server 与 client 会经历三次握手

1. *浏览器*向*服务器*发送 TCP 数据：SYN(seq=x)
2. *服务器*向*浏览器*发送 TCP 数据：ACK(seq=x+1) SYN(y)
3. *浏览器*向*服务器*发送 TCP 数据：ACK(seq=y+1)

关闭 TCP 连接时 server 与 client 会经历四次挥手

1. *浏览器*向*服务器*发送 TCP 数据：FIN(seq=x)

2. *服务器*向*浏览器*发送 TCP 数据：ACK(seq=x+1)
3. *服务器*向*浏览器*发送 TCP 数据：FIN(seq=y)
4. *浏览器*向*服务器*发送 TCP 数据：ACK(seq=y+1)

为什么 2、3 步骤不合并起来呢？看起来是脱裤子放屁。

答案：2、3 中间服务器很可能还有数据要发送，不能提前发送 FIN。

说说同源策略和跨域

同源策略是什么?

如果两个 URL 的协议、端口和域名都完全一致的话，则这两个 URL 是同源的。 **跨域**：当前页面中的某个接口请求的地址和当前页面的地址如果协议、域名、端口其中有一项不同，就说该接口跨域了。 跨域限制的原因：浏览器为了保证网页的安全，出的同源协议策略。

复制代码

```
http://www.baidu.com/s
http://www.baidu.com:80/ssdasdsadad
```

同源策略怎么做?

只要在**浏览器**里打开页面，就默认遵守同源策略。

优点

保证用户的隐私安全和数据安全。

缺点

很多时候，前端需要访问另一个域名的后端接口，会被浏览器阻止其获取响应。

比如甲站点通过 AJAX 访问乙站点的 /money 查询余额接口，请求会发出，但是响应会被浏览器屏蔽。

怎么解决缺点

使用跨域手段。

1. JSONP（前端体系课有完整且详细的介绍）
    1. 甲站点利用 script 标签可以跨域的特性，向乙站点发送 get 请求。
    2. 乙站点**后端改造** JS 文件的内容，将数据传进回调函数。
    3. 甲站点通过回调函数拿到乙站点的数据。
2. CORS（前端体系课有完整且详细的介绍）
    1. 对于简单请求，乙站点在响应头里添加 `Access-Control-Allow-Origin: http://甲站点` 即可。
    2. 对于复杂请求，如 PATCH，乙站点需要：
        1. 响应 OPTIONS 请求，在响应中添加如下的响应头

```makefile
Access-Control-Allow-Origin: https://甲站点
Access-Control-Allow-Methods: POST, GET, OPTIONS, PATCH
Access-Control-Allow-Headers: Content-Type
```

        2. 响应 POST 请求，在响应中添加 `Access-Control-Allow-Origin` 头。
    3. 如果需要附带身份信息，JS 中需要在 AJAX 里设置 `xhr.withCredentials = true` 。
3. Nginx 代理 / Node.js 代理
    1. 前端 ⇒ 后端 ⇒ 另一个域名的后端

详情参考 MDN CORS 文档 。

Session、Cookie、LocalStorage、SessionStorage 的区别

1.都是浏览器存储 2.都存储在浏览器本地 **区别**：

1.cookie由服务器写入， sessionStorage以及localStorage都是由前端写入

2.cookie的生命周期由服务器端写入时就设置好的，localStorage是写入就一直存在，除非手动清除，sessionStorage是由页面关闭时自动清除

3.cookie存储空间大小约4kb， sessionStorage及localStorage空间比较大，大约5M

4.3 者的数据共享都遵循同源原则，sessionStorage还限制必须是同一个页面 5.前端给后端发送请求时，自动携带cookie, session 及 local都不携带 6.cookie一般存储登录验证信息或者token，localStorage常用于存储不易变动的数据，减轻服务器压力，sessionStorage可以用来监测用户是否是刷新进入页面，如音乐播放器恢复进度条功能

- Cookie V.S. LocalStorage
    1. 主要区别是 Cookie 会被发送到服务器，而 LocalStorage 不会
    2. Cookie 一般最大 4k，LocalStorage 可以用 5Mb 甚至 10Mb（各浏览器不同）
- LocalStorage V.S. SessionStorage
    1. LocalStorage 一般不会自动过期（除非用户手动清除）
    2. SessionStorage 在回话结束时过期（如关闭浏览器之后，具体由浏览器自行决定）
- Cookie V.S. Session
    1. Cookie 存在浏览器的文件里，Session 存在服务器的文件里
    2. Session 是基于 Cookie 实现的，具体做法就是把 SessionID 存在 Cookie 里

TS押题

TS 和 JS 的区别是什么？有什么优势?

1. 语法层面：TypeScript = JavaScript + Type（TS 是 JS 的超集）
2. 执行环境层面：浏览器、Node.js 可以直接执行 JS，但不能执行 TS（Deno 可以执行 TS）
3. 编译层面：TS 有编译阶段，JS 没有编译阶段（只有转译阶段和 lint 阶段）
4. 编写层面：TS 更难写一点，但是**类型更安全**
5. 文档层面：TS 的代码写出来就是文档，IDE 可以完美**提示。**JS 的提示主要靠 TS

Any、unknown、never 的区别是什么?

Any V.S. unknown

二者都是顶级类型（top type），任何类型的值都可以赋值给顶级类型变量：

ts 复制代码

```ts
let foo: any = 123; // 不报错
let bar: unknown = 123; // 不报错
```

但是 unknown 比 any 的类型检查更严格，any 什么检查都不做，unknown 要求先收窄类型：

ts 复制代码

```ts
const value: unknown = "Hello World";
const someString: string = value;
// 报错：Type 'unknown' is not assignable to type 'string'.(2322)
```

ts 复制代码

```ts
const value: unknown = "Hello World";
const someString: string = value as string; // 不报错
```

如果改成 any，基本在哪都不报错。所以能用 unknown 就优先用 unknown，类型更安全一点。

Never

Never 是底类型，表示不应该出现的类型，这里有一个 尤雨溪给出的例子 ：

go 复制代码

```go
interface A {
  type: 'a'
}

interface B {
  type: 'b'
}

type All = A | B

function handleValue(val: All) {
  switch (val.type) {
    case 'a':
      // 这里 val 被收窄为 A
      break
    case 'b':
      // val 在这里是 B
      break
    default:
      // val 在这里是 never
      const exhaustiveCheck: never = val
      break
```

```
  }
}
```

现在你应该理解什么是「不应该出现的类型」了吧。

Type 和 interface 的区别是什么?

官方给出的 文档说明 ：

1. 组合方式：interface 使用 extends 来实现继承，type 使用 & 来实现联合类型。
2. 扩展方式：interface 可以重复声明用来扩展，type 一个类型只能声明一次
3. 范围不同：type 适用于基本类型，interface 一般不行。
4. 命名方式：interface 会创建新的类型名，type 只是创建类型别名，并没有新创建类型。

其他……建议搜一下博客。

TS 工具类型 Partial、Required、Readonly、Exclude、Extract、Omit、ReturnType 的作用和实现?

1. 将英文翻译为中文。
    1. Partial 部分类型
    2. Required 必填类型
    3. Readonly 只读类型
    4. Exclude 排除类型
    5. Extract 提取类型
    6. Pick/Omit 排除 key 类型
    7. ReturnType 返回值类型
2. 举例说明每个工具类型的用法。

Vue2押题

说一说 Vue 列表为什么加 key?

1.key 是为 Vue 中 vnode 的唯一标记，通过这个 key，我们的 diff 操作可以更准确、更快速
2.如果不使用 key，Vue 会使用一种最大限度减少动态元素并且尽可能的尝试就地修改/复用相同类型元素的算法。
3.如果绑定数组的索引index，则起不到优化diff算法的作用，因为一旦数组内元素进行增删，后续节点的绑定的key也会发生变化，导致diff进行多余的更新操作。

Vue 中 $nextTick 作用与原理?

NextTick作用：Vue更新DOM是异步执行的，当数据发生变化时，Vue会开启一个异步更新队列，视图需要等队列中所有数据更新完后再更新视图，所以$nextTick可以解决这样的问题，相当于一种优化策略。 原理：

1. 把回调函数放入callbacks等待执行
2. 将执行函数放到微任务或者宏任务中
3. 事件循环到了微任务或者宏任务，执行函数依次执行callbacks中的回调

说一说 v-if 和 v-show区别?

相同点:都是控制元素隐藏和显示的指令

不同点: v-show控制的无论是true还是false都会在DOM树中显示,相当于通过display:none控制元素隐藏，`v-if` 显示隐藏是将 `dom` 元素整个添加或删除,v-show适合在切换频繁显示/隐藏的元素上,v-if不适合使用在切换频繁的元素上也不适合在元素内容很多上，

- `v-show` 由 `false` 变为 `true` 的时候不会触发组件的生命周期
- `v-if` 由 `false` 变为 `true` 的时候，触发组件的 `beforeCreate` 、 `create` 、 `beforeMount` 、 `mounted` 钩子，由 `true` 变为 `false` 的时候触发组件的 `beforeDestory` 、 `destoryed` 方法

性能消耗： `v-if` 有更高的切换消耗； `v-show` 有更高的初始渲染消耗；

Vue 2 的生命周期钩子有哪些？数据请求放在哪个钩子?

三个阶段 挂载阶段：beforeCreate、created、beforeMounted、mounted

更新阶段：beforeUpdate、updated 销毁阶段：beforeDestroy、destroyed

2. 每个阶段的特性 🤔 beforeCreate：创建实例之前 🤔 created：实例创建完成（执行new Vue(options)），可访问data、computed、watch、methods上的方法和数据，可进行数据请求，未挂载到DOM结构上，不能获取el属性，如果要进行dom操作，那就要用

nextTick函数 🤔 beforeMount：在挂载开始之前被调用，beforeMount之前，会找到对应的template，并编译成render函数 🤔 mounted：实例挂载到DOM上，此时可以通过DOM API获取到DOM节点，可进行数据请求 🤔 beforeupdate：响应式数据更新时调用，发生在虚拟DOM打补丁之前，适合在更新之前访问现有的DOM，比如手动移除已添加的事件监听器 🤔 updated：虚拟 DOM 重新渲染和打补丁之后调用，组件DOM已经更新 🤔 beforeDestroy：实例销毁之前调用，this仍能获取到实例，常用于销毁定时器、解绑全局事件、销毁插件对象等操作 🤔 destroyed: 实例销毁之后

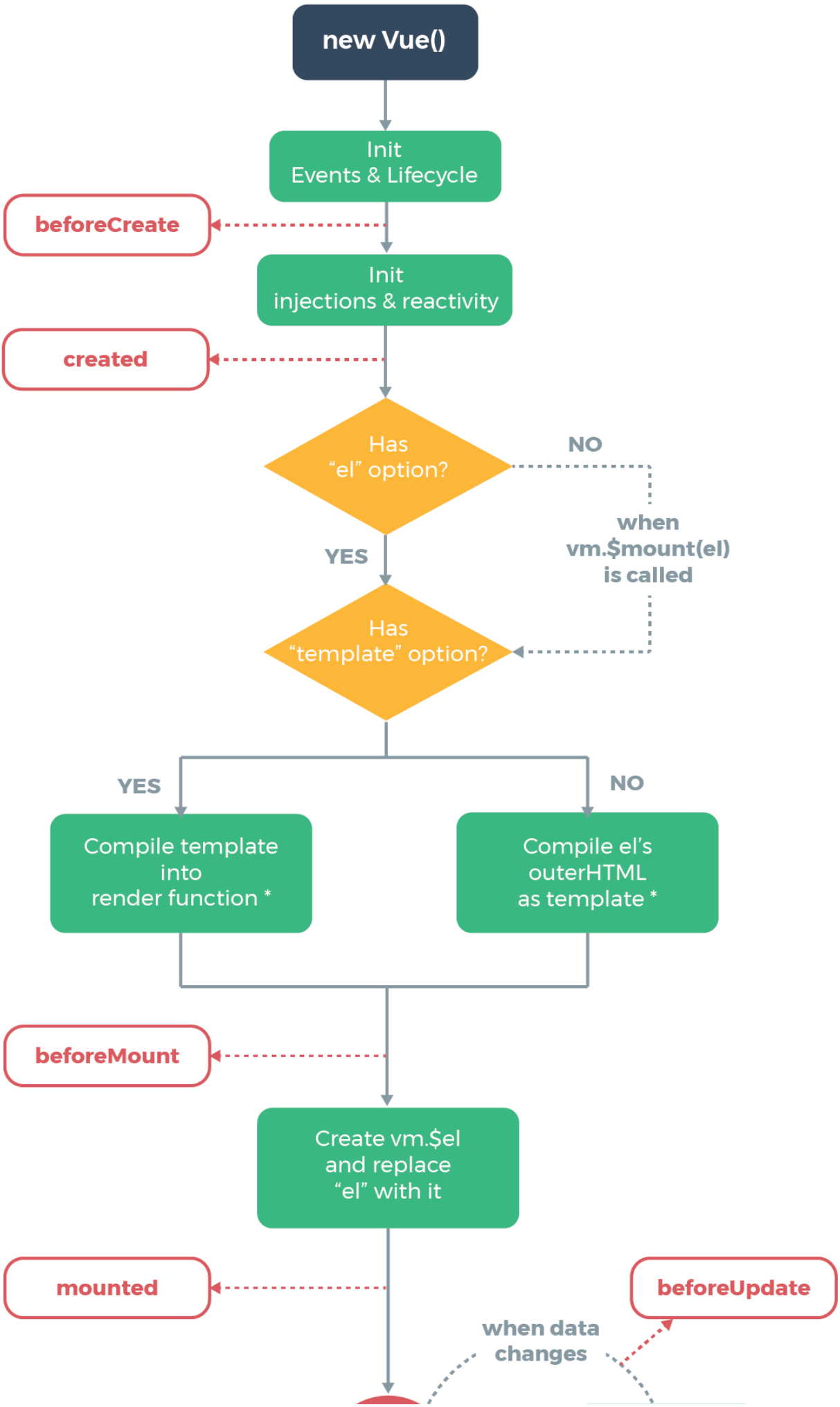
3. 父子组件执行顺序 挂载：父created -> 子created -> 子mounted> 父mounted 更新：父beforeUpdate -> 子beforeUpdated -> 子updated -> 父亲updated 销毁：父beforeDestroy -> 子beforeDestroy -> 子destroyed -> 父destroyed Vue 2 文档写得很清楚，红色空心框中的文字皆为生命周期钩子：

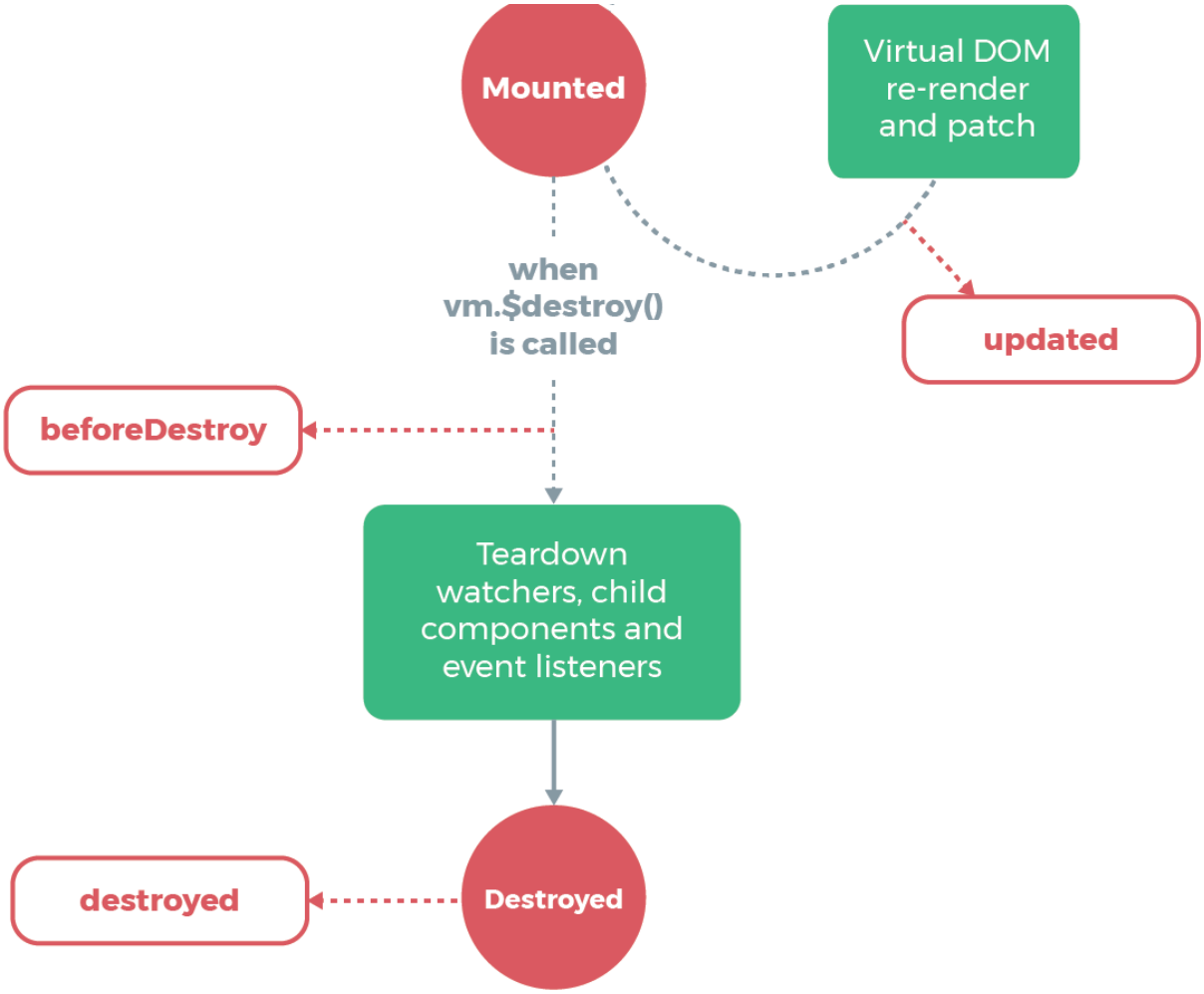
1. create x 2 (before + ed) - SSR
2. mount x 2
3. update x 2
4. destroy x 2

还有三个写在 钩子列表　　里：

1. activated
2. deactivated
3. errorCaptured

请求放在 mounted 里面，因为放在其他地方都不合适（xjb扯）。


new Vue()
Init
Events & Lifecycle
beforeCreate
Init
injections & reactivity
created
Has
“el” option?
NO
when
vm.$mount(el)
is called
YES
Has
“template” option?
YES
NO
Compile template
into
render function *
Compile el’s
outerHTML
as template *
beforeMount
Create vm.$el
and replace
“el” with it
mounted
beforeUpdate
when data
changes



Vue 2 组件间通信方式有哪些?

1. 父子组件：使用「props 和事件」进行通信
2. 爷孙组件：
    1. 使用两次父子组件间通信来实现
    2. 使用「provide + inject」来通信
3. 任意组件：使用 eventBus = new Vue() 来通信
    1. 主要API 是 eventBus.*on*和*eventBus*.emit
    2. 缺点是事件多了就很乱，难以维护

4. 任意组件：使用 Vuex 通信（Vue 3 可用 Pinia 代替 Vuex）

Vuex 用过吗？怎么理解？

1. 背下文档第一句：Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式 + 库 采用**集中式存储管理**应用的

2. 说出核心概念的名字和作用：store/State/Getter/Mutation/Action/Module

    1. store 是个大容器，包含以下所有内容
    2. State 用来读取状态，带有一个 mapState 辅助函数
    3. Getter 用来读取派生状态，附有一个 mapGetters 辅助函数
    4. Mutation 用于同步提交状态变更，附有一个 mapMutations 辅助函数
    5. Action 用于异步变更状态，但它提交的是 mutation，而不是直接变更状态。
    6. Module 用来给 store 划分模块，方便维护代码

常见追问：Mutation 和 Action 为什么要分开?

答案：为了让代码更易于维护。（可是 Pinia 就把 Mutation 和 Action 合并了呀）

完。

VueRouter 用过吗？怎么理解？

1. 背下文档第一句：Vue Router 是 Vue.js 的官方路由。它与 Vue.js 核心深度集成，让用 Vue.js 构建单页应用变得轻而易举。

2. 说出核心概念的名字和作用： `router-link` `router-view` 嵌套路由、Hash 模式和 History 模式、导航守卫、

懒加载： ES6的impot方式: component: () => import(/* webpackChunkName: "about" */ '../views/About.vue'), VUE中的异步组件进行懒加载方式: component: resolve=> (require(['../views/About'],resolve)) 加分回答 vue-router 实现懒加载的作用：性能优化，不用到该路由，不加载该组件。

3. 常见追问：

    1. Hash 模式和 History 模式的区别？\

- History和hash都是利用浏览器的2种特性实现前端路由，history是利用浏览历史记录栈的API实现，hash是监听location hash值变化事件来实现

- 2.history的url没有#号，hash有#号

- 3.相同的url,history会触发添加到浏览器历史记录栈中，hash不会触发，history需要后端配合，如果后端不配合刷新页面会出现404，hash不需要 hashRouter原理：通过window.onhashchange获取url中hash值 historyRouter原理：通过history.pushState,使用它做页面跳转不会触发页面刷新，使用window.onpopstate监听浏览器的前进和后退

2. 导航守卫如何实现登录控制?

```vbnet
router.beforeEach((to, from, next) => {
  if (to.path === '/login') return next()
  if (to是受控页面 && 没有登录) return next('/login')
  next()
})
```

推荐阅读：

路由守卫

Vue 2 是如何实现双向绑定的?

Vue通过v-model指令进行双向绑定

原理：通过Object.definedProperty劫持数据发生的改变，如果数据发生了改变（在set中进行赋值的），触发update方法进行更新节点内容，从而实现了数据双向绑定的原理。

Object.defineProperty的缺点

1. 一次性递归到底开销很大，如果数据很大，大量的递归导致调用栈溢出
   2.不能监听对象的新增属性和删除属性
   3.无法正确的监听数组的方法，当监听的下标对应的数据发生改变时

Vue 3 押题

Vue3.0 实现数据双向绑定的方法

在Vue2.0的基础上将Object.definedproperty换成了功能更强大的**proxy**，原理相同。

采用Proxy来劫持整个对象，相比Vue2.0中的Object.defineProperty，能够动态监听添加的属性，可以监听数组的索引和length属性,将其中每个数据进行一遍数据劫持（get实现依赖收集，set实现事件派发（这里的模式为发布订阅模式））。

补充：相对vue2.0解决的问题：解决无法监听新增属性或删除属性的响应式问题、解决无法监听数组长度和index变化问题。

Vue 3 为什么使用 Proxy?

1. 弥补 Object.defineProperty 的两个不足
    1. 动态创建的 data 属性需要用 Vue.set 来赋值，Vue 3 用了 Proxy 就不需要了
    2. 基于性能考虑， Vue 2 篡改了数组的 7 个 API ，Vue 3 用了 Proxy 就不需要了
2. defineProperty 需要提前递归地遍历 data 做到响应式，而 Proxy 可以在真正用到深层数据的时候再做响应式（惰性）

Vue 3 为什么使用 Composition API?

答案参考尤雨溪的博客： Vue Function-based API RFC - 知乎 (zhihu.com)

1. Composition API 比 mixins、高阶组件、extends、Renderless Components 等更好，原因有三：
    1. 模版中的数据来源不清晰。
    2. 命名空间冲突。
    3. 性能。
2. 更适合 TypeScript

Vue 3 对比 Vue 2 做了哪些改动?

官方文档 写了（中文在这 ），这里列出几个容易被考的：

1. createApp() 代替了 new Vue()
2. v-model 代替了以前的 v-model 和 .sync

3. 根元素可以有不止一个元素了
4. 新增 Teleport 传送门
5. destroyed 被改名为 unmounted 了（before 当然也改了）
6. ref 属性支持函数了

其他建议自己看看写写。、

React 押题

React生命周期的各个阶段是什么?

React生命周期分为3个阶段；分别是：1、创建阶段，也被称为初始化阶段，表示组件第一次在DOM树中进行渲染的过程；2、更新阶段，也叫存在阶段，表示组件被重新渲染的过程；3、卸载阶段，也叫销毁阶段，表示组件从DOM中删除的过程。

- 1. 挂载卸载过程
  - 1.1.constructor()
  - 1.2.componentWillMount()
  - 1.3.componentDidMount()
  - 1.4.componentWillUnmount ()
- 2. 更新过程
  - 2.1. componentWillReceiveProps (nextProps)
  - 2.2.shouldComponentUpdate(nextProps,nextState)
  - 2.3.componentWillUpdate (nextProps,nextState)
  - 2.4.componentDidUpdate(prevProps,prevState)
  - 2.5.render()
- 3. React新增的生命周期(个人补充)
  - 3.1. getDerivedStateFromProps(nextProps, prevState)
  - 3.2. getSnapshotBeforeUpdate(prevProps, prevState)

这周开始学习React的生命周期。

React的生命周期从广义上分为三个阶段：挂载、渲染、卸载

SetState是同步还是异步的?

同步代码异步表现，原因是react的批处理。 react18之前在生命周期函数和合成事件中表现为异步，在原生事件中表现为同步。 在react18优化批处理之后，在任何地方调用setState都会批处理，因此都表现为异步。

虚拟 DOM 的原理是什么?

1. 是什么

   虚拟 DOM 就是虚拟节点（这句汉化很重要）。React 用 JS 对象来**模拟** DOM 节点，然后将其渲染成真实的 DOM 节点。

2. 怎么做

   第一步是模拟

   用 JSX 语法写出来的 div 其实就是一个虚拟节点：

   ini 复制代码

   ```ini
   <div id="x">
     <span class="red">hi</span>
   </div>
   ```

   这代码会得到这样一个对象：

   css 复制代码

   ```css
   {
     tag: 'div',
     props: {
       id: 'x'
     },
   ```

```
  children: [
    {
      tag: 'span',
      props: {
        className: 'red'
      },
      children: [
        'hi'
      ]
    }
  ]
}
```

能做到这一点是因为 JSX 语法会被转译为 createElement 函数调用（也叫 h 函数），如下：

php 复制代码

```
React.createElement("div", { id: "x"},
  React.createElement("span", { class: "red" }, "hi")
)
```

第二步是将虚拟节点渲染为真实节点

javascript 复制代码

```
function render(vdom) {
  // 如果是字符串或者数字，创建一个文本节点
  if (typeof vdom === 'string' || typeof vdom === 'number') {
    return document.createTextNode(vdom)
  }
  const { tag, props, children } = vdom
  // 创建真实DOM
  const element = document.createElement(tag)
  // 设置属性
  setProps(element, props)
  // 遍历子节点，并获取创建真实DOM，插入到当前节点
  children
    .map(render)
    .forEach(element.appendChild.bind(element))

  // 虚拟 DOM 中缓存真实 DOM 节点
  vdom.dom = element

  // 返回 DOM 节点
  return element
}

function setProps // 略
```

```
function setProp // 略

// 作者：Shenfq
// 链接：https://juejin.cn/post/6844903870229905422
```

注意，如果节点发生变化，并不会直接把新虚拟节点渲染到真实节点，而是先经过 diff 算法得到一个 patch 再更新到真实节点上。

3. 解决了什么问题
    1. DOM 操作性能问题。通过虚拟 DOM 和 diff 算法减少不必要的 DOM 操作，保证性能不太差
    2. DOM 操作不方便问题。以前各种 DOM API 要记，现在只有 setState
4. 优点
    1. 为 React 带来了跨平台能力，因为虚拟节点除了渲染为真实节点，还可以渲染为其他东西。
    2. 让 DOM 操作的整体性能更好，能（通过 diff）减少不必要的 DOM 操作。
5. 缺点
    1. 性能要求极高的地方，还是得用真实 DOM 操作（目前没遇到这种需求）
    2. React 为虚拟 DOM 创造了**合成事件**，跟原生 DOM 事件不太一样，工作中要额外注意
        1. 所有 React 事件都绑定到根元素，自动实现事件委托
        2. 如果混用合成事件和原生 DOM 事件，有可能会出 bug
6. 如何解决缺点

    不用 React，用 Vue 3（笑）

React 或 Vue 的 DOM diff 算法是怎样的?

1、Diff算法主要就是在虚拟DOM树发生变化后，生成DOM树更新补丁的方式，对比新旧两株虚拟DOM树的变更差异，将更新补丁作用于真实DOM，以最小成本完成视图更新；

2、框架会将所有的结点先转化为虚拟节点Vnode，在发生更改后将VNode和原本页面的OldNode进行对比，然后以VNode为基准，在oldNode上进行准确的修改。（修改准则：原本没有新版有，则增加；原本有新版没有，则删除；都有则进行比较，都为文本结点则替换值；都为静态资源不处理；都为正常结点则替换）

1. 是什么

   DOM diff 就是对比两棵虚拟 DOM 树的算法（废话很重要）。当组件变化时，会 render 出一个新的虚拟 DOM，diff 算法对比新旧虚拟 DOM 之后，得到一个 patch，然后 React 用 patch 来更新真实 DOM。

2. 怎么做

   1. 首先对比两棵树的根节点

      1. 如果根节点的类型改变了，比如 div 变成了 p，那么直接认为整棵树都变了，不再对比子节点。此时直接删除对应的真实 DOM 树，创建新的真实 DOM 树。

      2. 如果根节点的类型没变，就看看属性变了没有

         1. 如果没变，就保留对应的真实节点

         2. 如果变了，就只更新该节点的属性，不重新创建节点。

            1. 更新 style 时，如果多个 css 属性只有一个改变了，那么 React 只更新改变的。

   2. 然后同时遍历两棵树的子节点，每个节点的对比过程同上。

      1. 情况一

```
<ul>
  <li>A</li>
  <li>B</li>
</ul>

<ul>
  <li>A</li>
  <li>B</li>
  <li>C</li>
</ul>
```

css 复制代码

React 依次对比 A-A、B-B、空-C，发现 C 是新增的，最终会创建真实 C 节点插入页面。

2. 情况二

```
<ul>
  <li>B</li>
  <li>C</li>
</ul>

<ul>
  <li>A</li>
  <li>B</li>
  <li>C</li>
</ul>
```

React 对比 B-A，会删除 B 文本新建 A 文本；对比 C-B，会删除 C 文本，新建 B 文本；（注意，并不是边对比边删除新建，而是把操作汇总到 patch 里再进行 DOM 操作。）对比空-C，会新建 C 文本。

你会发现其实只需要创建 A 文本，保留 B 和 C 即可，为什么 React 做不到呢?

因为 React 需要你加 key 才能做到：

```
<ul>
  <li key="b">B</li>
  <li key="c">C</li>
</ul>

<ul>
  <li key="a">A</li>
  <li key="b">B</li>
  <li key="c">C</li>
</ul>
```

React 先对比 key 发现 key 只新增了一个，于是保留 b 和 c，新建 a。

以上是 React 的 diff 算法（源码分析在下一节补充视频中，时长一小时，有能力者选看）。

但面试官想听的可能是 Vue 的「双端交叉对比」算法：

Diff算法 | Marvin

补充：React DOM diff 和 Vue DOM diff 的区别?

先纠正之前的一个细节错误：

错：我认为数组存储的是整棵树。

对：其实数组存储的是拥有相同爸爸的一群子节点。

React DOM diff 和 Vue DOM diff 的区别：

1. React 是从左向右遍历对比，Vue 是双端交叉对比。
2. React 需要维护三个变量（有点扯），Vue 则需要维护四个变量。
3. Vue 整体效率比 React 更高，举例说明：假设有 N 个子节点，我们只是把最后子节点移到第一个，那么
    1. React 需要进行借助 Map 进行 key 搜索找到匹配项，然后复用节点
    2. Vue 会发现移动，直接复用节点

附 React DOM diff 代码查看流程：

1. 运行 `git clone ht``tps://github.com/facebook/react.git`
2. 运行 `cd react; git switch 17.0.2`
3. 用 VSCode 或 WebStorm 打开 react 目录
4. 打开 `packages/react-reconciler/src/ReactChildFiber.old.js` 第 1274 行查看旧版代码，或打开 `packages/react-reconciler/src/ReactChildFiber.new.js` 第 1267 行查看新代码（实际上一样）
5. 忽略所有警告和报错，因为 React JS 代码中有不是 JS 的代码
6. 折叠所有代码
7. 根据 React 文档中给出的场景反复在大脑中运行代码
    1. 场景0：单个节点，会运行到 reconcileSingleElement。接下来看多个节点的情况。
    2. 场景1：没 key，标签名变了，最终会走到 createFiberFromElement（存疑）

3. 场景2：没 key，标签名没变，但是属性变了，最终走到 updateElement 里的 useFiber
4. 场景3：有 key，key 的顺序没变，最终走到 updateElement
5. 场景4：有 key，key 的顺序变了，updateSlot 返回 null，最终走到 mapRemainingChildren、updateFromMap 和 updateElement(matchedFiber)，整个过程较长，效率较低

8. 代码查看要点：

    1. 声明不看（用到再看）
    2. if 先不看（但 if else 要看）
    3. 函数调用必看

9. 必备快捷键：折叠所有、展开、向前、向后、查看定义

React事件绑定原理

React中事件绑定都不是绑定在对应的DOM上，而是统一绑定在document上，采用事件冒泡的形式，向上传递。之所以这样做是因为React可以统一处理所有的事件绑定，在销毁的时候可以统一移除绑定。

采用合成事件的原因:

1.兼容所有的浏览器和实现跨平台开发，
2.统一挂载在document上，减少内存的消耗，方便在组件挂载/卸载时统一订阅和移除事件，
3.方便事件统一管理

React 有哪些生命周期钩子函数？数据请求放在哪个钩子里？

React 的文档稍微有点乱，需要配合两个地方一起看才能记忆清楚：

React.Component - React

React Lifecycle Methods diagram　　总得来说：

1. 挂载时调用 constructor，更新时不调用
2. 更新时调用 shouldComponentUpdate 和 getSnapshotBeforeUpdate，挂载时不调用
3. should... 在 render 前调用，getSnapshot... 在 render 后调用

4. 请求放在 componentDidMount 里，最好写博客，容易忘。

React 如何实现组件间通信

1. 父子组件通信：props + 函数
2. 爷孙组件通信：两层父子通信或者使用 Context.Provider 和 Context.Consumer
3. 任意组件通信：其实就变成了状态管理了
    1. Redux
    2. Mobx
    3. Recoil

你如何理解 Redux?

1. 文档第一句话背下来：Redux 是一个状态管理库/状态容器。
2. 把 Redux 的核心概念说一下：
    1. State
    2. Action = type + payload 荷载
    3. Reducer
    4. Dispatch 派发
    5. Middleware
3. 把 ReactRedux 的核心概念说一下：
    1. connect()(Component)
    2. mapStateToProps
    3. mapDispatchToProps
4. 说两个常见的中间件 redux-thunk redux-promise

什么是高阶组件 HOC?

参数是组件，返回值也是组件的函数。什么都能做，所以抽象问题就具体回答。

举例说明即可：

1. React.forwardRef
2. ReactRedux 的 connect
3. ReactRouter 的 withRouter

参考阅读： 「react进阶」一文吃透React高阶组件(HOC) - 掘金 (juejin.cn)

React中hooks的优缺点是什么?

Hook的**优点**：1.让函数组件拥有自己的状态和生命周期。2.使用函数组件加Hooks代码更加简洁。3.不需要老是去纠结this指向的问题。4.通过自定义hooks实现逻辑复用。

缺点:class组件的三个生命周期函数合并在一个生命周期函数内。

React Hooks 如何模拟组件生命周期?

1. 模拟 componentDidMount
2. 模拟 componentDidUpdate
3. 模拟 componentWillUnmount

代码示例如下：

javascript 复制代码

```
import { useEffect,useState,useRef } from "react";
import "./styles.css";

export default function App() {
  const [visible, setNextVisible] = useState(true)
  const onClick = ()=>{
    setNextVisible(!visible)
  }
  return (
    <div className="App">
      <h1>Hello CodeSandbox</h1>
      {visible ? <Frank/> : null}
      <div>
        <button onClick={onClick}>toggle</button>
        </div>

    </div>
  );
}
```

```
function Frank(props){
  const [n, setNextN] = useState(0)
  const first = useRef(true)
  useEffect(()=>{
    if(first.current === true ){
      return
    }
    console.log('did update')
  })
  useEffect(()=>{
    console.log('did mount')
    first.current = false
    return ()=>{
      console.log('did unmount')
    }
  }, [])

  const onClick = ()=>{
    setNextN(n+1)
  }
  return (
    <div>Frank
      <button onClick={onClick}>+1</button>
    </div>
  )
}
```

完。

Node.js 押题

Node.js 的 EventLoop 是什么?

背景知识

Event Loop、计时器、nextTick - 掘金

Node.js 将各种函数（也叫任务或回调）分成至少 6 类，按先后顺序调用，因此将时间分为六个阶段：

1. *timers 阶段（setTimeout）*
2. I/O callbacks 该阶段不用管

3. idle, prepare 该阶段不用管
4. *Poll 轮询阶段，停留时间最长，可以随时离开。*
    1. 主要用来处理 I/O 事件，该阶段中 Node 会不停询问操作系统有没有文件数据、网络数据等
    2. 如果 Node 发现有 timer 快到时间了或者有 setImmediate 任务，就会主动离开 poll 阶段
5. *check 阶段，主要处理 setImmediate 任务*
6. close callback 该阶段不用管

Node.js 会不停的从 1 ~ 6 循环处理各种事件，这个过程叫做事件循环（Event Loop）。

NextTick

Process.nextTick(fn) 的 fn 会在什么时候执行呢?

在 Node.js 11 之前，会在每个阶段的*末尾集中*执行（俗称队尾执行）。

在 Node.js 11 之后，会在每个阶段的*任务间隙*执行（俗称插队执行）。

浏览器跟 Node.js 11 之后的情况类似。可以用 window.queueMicrotask 模拟 nextTick。

Promise

Promise.resolve(1).then(fn) 的 fn 会在什么时候执行?

这要看 Promise 源码是如何实现的，一般都是用 process.nextTick(fn) 实现的，所以直接参考 nextTick。

Async / await

这是 Promise 的语法糖，所以直接转为 Promise 写法即可。

面试题1：

javascript 复制代码

```
setTimeout(() => {
  console.log('setTimeout')
```

```
})

setImmediate(() => {
  console.log('setImmediate')
})
// 在 Node.js 运行会输出什么？
// A setT setIm
// B setIm setT
// C 出错
// D A 或 B
// 在浏览器执行会怎样？
```

面试题2：

javascript 复制代码

```
async function async1(){
    console.log('1')                    // 2
    async2().then(()=>{
      console.log('2')
    })

}
async function async2(){
    console.log('3')                    // 3
}
console.log('4')                         // 1
setTimeout(function(){
    console.log('5')
},0)
async1();
new Promise(function(resolve){
    console.log('6')                     // 4
    resolve();
}).then(function(){
    console.log('7')
})
console.log('8')                            // 5
//4 1 3 6 8 2 7 5
```

浏览器里的微任务和任务是什么?

浏览器中并不存在宏任务，宏任务（Macrotask）是 Node.js 发明的术语。

浏览器中只有任务（Task）和微任务（Microtask）。

1. 使用 script 标签、setTimeout 可以创建任务。

2. 使用 Promise#then、window.queueMicrotask、MutationObserver、Proxy 可以创建微任务。

执行顺序是怎样的呢?

微任务会在任务间隙执行（俗称插队执行）。

⚠

注意，微任务不能插微任务的队，微任务只能插任务的队。

面试题：

为什么以下代码的返回结果是 0 1 2 3 4 5 6 而不是 0 1 2 4 3 5 6?

Express.js 和 koa.js 的区别是什么?

1. 中间件模型不同：express 的中间件模型为线型，而 koa 的为U型（洋葱模型）。
2. 对异步的处理不同：express 通过回调函数处理异步，而 koa 通过generator 和 async/await 使用同步的写法来处理异步，后者更易维护，但彼时 Node.js 对 async 的兼容性和优化并不够好，所以没有流行起来。
3. 功能不同：express 包含路由、渲染等特性，而 koa 只有 http 模块。

总得来说，express 功能多一点，写法烂一点，兼容性好一点，所以当时更流行。虽然现在 Node.js 已经对 await 支持得很好了，但是 koa 已经错过了风口。

不过 express 和 koa 的作者都是 TJ 大神。

工程化押题

说一说前端性能优化手段?

代码层面：

- 防抖和节流（resize，scroll，input）。
- 减少回流（重排）和重绘。
- 事件委托。
- css 放 ， js 脚本放 最底部。

- 减少 DOM 操作。
- 按需加载，比如 React 中使用 `React.lazy` 和 `React.Suspense` ，通常需要与 webpack 中的 `splitChunks` 配合。**懒加载**

构建方面：

- **压缩代码文件**，在 webpack 中使用 `terser-webpack-plugin` 压缩 Javascript 代码；使用 `css-minimizer-webpack-plugin` 压缩 CSS 代码；使用 `html-webpack-plugin` 压缩 html 代码。
- **开启 gzip 压缩**，webpack 中使用 `compression-webpack-plugin` ，node 作为服务器也要开启，使用 `compression` 。
- **常用的第三方库使用 CDN 服务**，在 webpack 中我们要配置 externals，将比如 React，Vue 这种包不打倒最终生成的文件中。而是采用 CDN 服务。

其它：

- 使用 http2。因为解析速度快，头部压缩，多路复用，服务器推送静态资源。
- 使用服务端渲染。
- 图片压缩。
- 使用 http 缓存，比如服务端的响应中添加 `Cache-Control / Expires` 。

常见 loader 和 plugin 有哪些？二者的区别是什么？

常见 loader

在 webpack 文档里写了：

Loaders | webpack　　你可以记住：

1. `babel-loader` 把 JS/TS 变成 JS
2. `ts-loader` 把 TS 变成 JS，**并提示类型错误**
3. `markdown-loader` 把 markdown 变成 html
4. `html-loader` 把 html 变成 JS 字符串
5. `sass-loader` 把 SASS/SCSS 变成 CSS
6. `css-loader` 把 CSS 变成 JS 字符串

7. `style-loader` 把 JS 字符串变成 style 标签
8. `postcss-loader` 把 CSS 变成更优化的 CSS
9. `vue-loader` 把单文件组件（SFC）变成 JS 模块
10. `thread-loader` 用于多进程打包

常见 plugin

也在 webpack 文档里写了：

Plugins | webpack　　你可以记住这些：

1. `html-webpack-plugin` 用于创建 HTML 页面并自动引入 JS 和 CSS
2. `clean-webpack-plugin` 用于清理之前打包的残余文件
3. `mini-css-extract-plugin` 用于将 JS 中的 CSS 抽离成单独的 CSS 文件
4. `SplitChunksPlugin` 用于代码分包（Code Split）
5. `DllPlugin` + `DllReferencePlugin` 用于避免大依赖被频繁重新打包，大幅降低打包时间

   Webpack使用-详解DllPlugin　　3. `eslint-webpack-plugin` 用于检查代码中的错误
4. `DefinePlugin` 用于在 webpack config 里添加全局变量
5. `copy-webpack-plugin` 用于拷贝静态文件到 dist

二者的区别

- Loader 是文件加载器（这句废话很重要）
  - 功能：能够对文件进行编译、优化、混淆（压缩）等，比如 babel-loader / vue-loader
  - 运行时机：在创建最终产物之前运行
- Plugin 是 webpack 插件（这句废话也很重要）
  - 功能：能实现更多功能，比如定义全局变量、Code Split、加速编译等
  - 运行时机：在整个打包过程（以及前后）都能运行

Webpack 如何解决开发时的跨域问题?

在开发时，我们的页面在 `localhost:8080` ，JS 直接访问后端接口（如 `https://xiedaimala.com` 或 `http://localhost:3000` ）会报跨域错误。

为了解决这个问题，可以在 webpack.config.js 中添加如下配置：

java 复制代码

```
module.exports = {
  //...
  devServer: {
    proxy: {
      '/api': {
        target: 'http://xiedaimala.com',
        changeOrigin: true,
      },
    },
  },
};
```

此时，在 JS 中请求 `/api/users` 就会自动被代理到 `http://xiedaimala.com/api/users` 。

如果希望请求中的 Origin 从 8080 修改为 xiedaimala.com，可以添加 `changeOrigin: true` 。

如果要访问的是 HTTPS API，那么就需要配置 HTTPS 证书，否则会报错。

不过，如果在 target 下面添加 `secure: false` ，就可以不配置证书且忽略 HTTPS 报错。

总之，记住常用选项就行了。

如何实现 tree-shaking?

这题属于拿着文档问面试者，欺负那些背不下文档的人。

Tree Shaking | webpack

Tree Shaking | webpack 中文文档　　tree-shaking 就是让没有用到的 JS 代码不打包，以减小包的体积。

怎么做

背下文档说的这几点：

1. 怎么删
    1. 使用 ES Modules 语法（即 ES6 的 import 和 export 关键字）
    2. CommonJS 语法无法 tree-shaking（即 require 和 exports 语法）
    3. 引入的时候只引用需要的模块
        1. 要写 `import {cloneDeep} from 'lodash-es'` 因为方便 tree-shaking
        2. 不要写 `import _ from 'lodash'` 因为会导致无法 tree-shaking 无用模块
2. 怎么不删：在 package.json 中配置 sideEffects，防止某些文件被删掉
    1. 比如我 import 了 x.js，而 x.js 只是添加了 window.x 属性，那么 x.js 就要放到 sideEffects 里
    2. 比如所有被 import 的 CSS 都要放在 sideEffects 里
3. 怎么开启：在 webpack config 中将 mode 设置为 production（开发环境没必要 tree-shaking）
    1. `mode: production` 给 webpack 加了非常多 优化 。

如何提高 webpack 构建速度?

Webpack 文档写着呢：

构建性能 | webpack 中文文档

1. 使用 DllPlugin 将不常变化的代码提前打包，并复用，如 vue、react
2. 使用 thread-loader 或 HappyPack（过时）进行多线程打包
3. 处于开发环境时，在 webpack config 中将 cache 设为 true，也可用 cache-loader（过时）
4. 处于生产环境时，关闭不必要的环节，比如可以关闭 source map
5. 网传的 HardSourceWebpackPlugin 已经一年多没更新了，谨慎使用

Webpack 与 vite 的区别是什么?

1. 开发环境区别
    1. Vite 自己实现 server，不对代码打包，充分利用浏览器对 `<script type=module>` 的支持
        1. 假设 main.js 引入了 vue
        2. 该 server 会把 `import { createApp } from 'vue'` 改为 `import { createApp } from "/node_modules/.vite/vue.js"` 这样浏览器就知道去哪里找 vue.js 了
    2. Webpack-dev-server 常使用 babel-loader 基于内存打包，比 vite 慢很多很多很多
        1. 该 server 会把 vue.js 的代码（递归地）打包进 main.js
2. 生产环境区别
    1. vite 使用 *rollup *+ *esbuild *来打包 JS 代码
    2. *Webpack *使用 *babel *来打包 JS 代码，比 esbuild 慢很多很多很多
        1. webpack 能使用 esbuild 吗？可以，你要自己配置（很麻烦）。
3. 文件处理时机
    1. vite 只会在你请求某个文件的时候处理该文件
    2. webpack 会提前打包好 main.js，等你请求的时候直接输出打包好的 JS 给你

目前已知 vite 的缺点有：

1. 热更新常常失败，原因不清楚
2. 有些功能 rollup 不支持，需要自己写 rollup 插件
3. 不支持非现代浏览器

Webpack 怎么配置多页应用?

这是对应的 webpack config：

```
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
```

```
  entry: {
    app: './src/app.js',
    admin: './src/admin.js',
  },
  plugins: [
    new HtmlWebpackPlugin({
      filename: 'index.html',
      chunks: ['app']
    }),
    new HtmlWebpackPlugin({
      filename: 'admin.html',
      chunks: ['admin']
    })
  ],
};
```

但是，这样配置会有一个「重复打包」的问题：假设 app.js 和 admin.js 都引入了 vue.js，那么 vue.js 的代码既会打包进 app.js，也会打包进 admin.js。我们需要使用 `optimization.splitChunks` 将共同依赖单独打包成 common.js（HtmlWebpackPlugin 会自动引入 common.js）。

如何支持无限多页面呢?

写点 Node.js 代码不就实现了么?

ini 复制代码

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
const fs = require('fs')
const path = require('path')

const filenames = fs.readdirSync('./src/pages')
  .filter(file => file.endsWith('.js'))
  .map(file => path.basename(file, '.js'))

const entries = filenames.reduce((result, name) => (
  { ...result, [name]: `./src/pages/${name}.js` }
), {})
const plugins = filenames.map((name) =>
  new HtmlWebpackPlugin({
    filename: name + '.html',
    chunks: [name]
  })
)

module.exports = {
  entry: {
    ...entries
```

```
  },
  plugins: [
    ...plugins
  ],
};
```

Swc、esbuild 是什么?

Swc

实现语言：Rust

功能：编译 JS/TS、打包 JS/TS

优势：比 babel 快很多很多很多（20倍以上）

能否集成进 webpack：能

使用者：Next.js、Parcel、Deno、Vercel、ByteDance、Tencent、Shopify……

做不到：

1. 对 TS 代码进行类型检查（用 tsc 可以）
2. 打包 CSS、SVG

Esbuild

实现语言：Go

功能：同上

优势：比 babel 快很多很多很多很多很多很多很多（10~100倍）

能否集成进 webpack：能

使用者：vite、vuepress、snowpack、umijs、blitz.js 等

做不到：

1. 对 TS 代码进行类型检查

2. 打包 CSS、SVG