2020 前端押题

第一部分：HTML

1、必考：你是如何理解 HTML 语义化的?

- 举例法：HTML 语义化就是使用正确的标签（总结）段落就写 p 标签，标题就写 h1 标签，文章就写 article 标签，视频就写 video 标签，等等
- 阐述法：首先讲以前的后台开发人员使用 table 布局，然后讲美工人员使用 div+css 布局，最后讲专业的前端会使用正确的标签进行页面开发

参考答案：

> 语义化是指根据内容的结构化（内容语义化），选择合适的标签（代码语义化），便于开发者阅读和写出更优雅的代码的同时，让浏览器的爬虫和机器很好的解析。

2、meta viewport 是做什么用的，怎么写?

举例法，然后逐个解释每个单词的意思。淘宝 H5 的 meta 标签，仅供参考：

```
<meta
  name="viewport"
  content="width=device-width,initial-scale=1,minimum-scale=1,maximum-scale=1,user-scalable=no,v
/>
```

3、你用过哪些 HTML 5 标签?

举例法，平时如果只用 div 写页面你就完了，把你平时用到的 html5 标签列举出来即可，但是要注意如果这个标签的用法比较复杂，你要先看一下 MDN 的文档再说这个标签；如果你说出一个标签，却不知道它有哪些 API，那么你就会被扣分

参考答案：

```
<header></header>
<article></article>
<section></section>
<footer></footer>

<video width="320" height="240" controls>
  <source src="movie.mp4" type="video/mp4" />
  <source src="movie.ogg" type="video/ogg" />
  您的浏览器不支持 video 标签。
</video>

<audio controls>
  <source src="horse.ogg" type="audio/ogg" />
  <source src="horse.mp3" type="audio/mpeg" />
  您的浏览器不支持 audio 元素。
</audio>

<canvas id="myCanvas" width="200" height="200"></canvas>
<script type="text/javascript">
  var canvas = document.getElementById("myCanvas");
  var ctx = canvas.getContext("2d");
  ctx.fillStyle = "#FF0000";
  ctx.fillRect(0, 0, 80, 100);
</script>

<svg xmlns="http://www.w3.org/2000/svg" version="1.1" height="190">
  <polygon
    points="100,10 40,180 190,60 10,60 160,180"
    style="fill:lime;stroke:purple;stroke-width:5;fill-rule:evenodd;"
  />
</svg>
```

4、H5 是什么?

阐述法，搜一下知乎就知道了，H5 表示移动端页面，反正不是 HTML5

参考答案:

https://www.zhihu.com/question/30363342

> 我们在谈论 H5 的时候，实际上是一个解决方案，一个看起来酷炫的移动端 onepage 网站的解决方案。而这个解决方案不仅包含了 HTML5 新增的 audio 标签，canvas，拖拽特性，本地存储，websocket 通信，同时也包括了盒模型、绝对定位，包括一切前端的基本知识

第二部分：CSS

1、必考：两种盒模型分别说一下

先说两种盒模型分别怎么写，具体到代码。然后说你平时喜欢用 border box，因为更好用

参考答案：

> box-sizing: conent-box; 将盒子设置为标准模型（盒子默认为标准模型）
> width = content

> box-sizing: border-box; 将盒子设置为 IE 模型（也叫做怪异盒子）
> width = content + padding + border

2、必考：如何垂直居中?

背代码

参考答案：

如果 .parent 的 height 不写，你只需要 padding: 10px 0; 就能将 .child 垂直居中；
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

- flex 布局

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

- 绝对定位 + transform

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

- 绝对定位 + margin

```
.parent {
  position: relative;
}
.child {
  position: absolute;
  margin: auto;
  top: 0;
  bottom: 0;
  left: 0;
  right: 0;
}
```

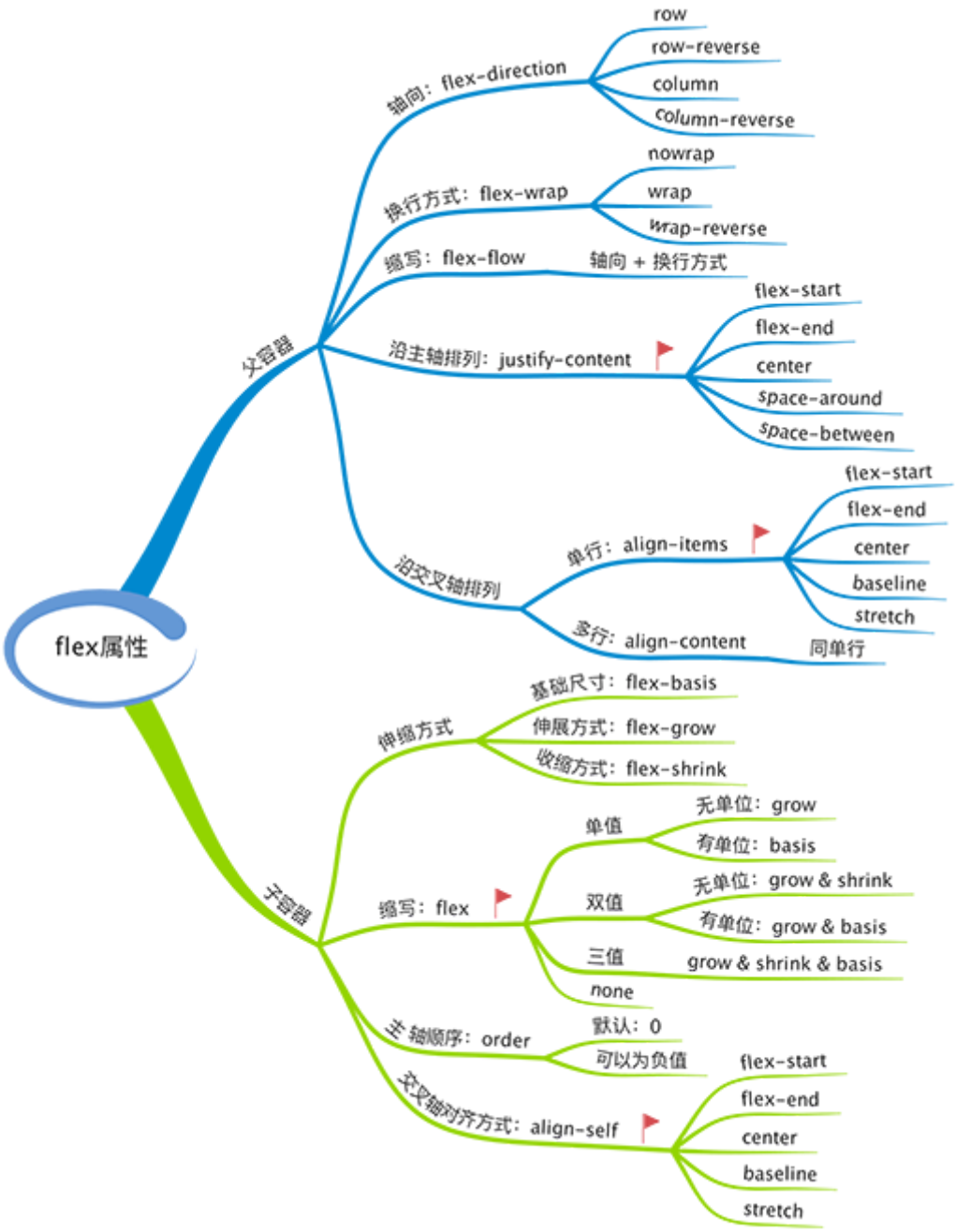
3、必考：flex 怎么用，常用属性有哪些?

看 MDN，背代码

参考答案:

https://blog.tmaize.net/posts/2016/04/18/flex%E5%B8%83%E5%B1%80%E7%AC%94%E8%AE%B0.html

```
.box {
  display: flex;
  /*Webkit 内核的浏览器，必须加上-webkit前缀*/
  display: -webkit-flex;
}
/*也可以设置为行内的flex属性*/
.box {
  display: inline-flex;
}
```



4、必考：BFC 是什么?

背 BFC 触发条件，MDN 写了
但是不用全部背下来，面试官只知道其中几个：

- 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）

参考答案：

> 块格式化上下文（Block Formatting Context，BFC） 是 Web 页面的可视 CSS 渲染的一部分，是块盒子的布局过程发生的区域，也是浮动元素与其他元素交互的区域。

> 事实上，BFC 的目的是形成一个相对于外界完全独立的空间，让内部的子元素不会影响到外部的元素

5、CSS 选择器优先级

- 背人云亦云的答案（错答案、已过时）：https://www.cnblogs.com/xugang/archive/2010/09/24/1833760.html
- 看面试官脸色行事
- 方方给的三句话：
  - 越具体优先级越高
  - 同样优先级写在后面的覆盖写在前面的
  - !important 优先级最高，但是要少用

参考答案：

https://juejin.im/post/5e97045b6fb9a03c31762a2f

> 内联 > id 选择器 > 类、属性、伪类选择器 > 标签元素、伪元素

6、清除浮动说一下

背代码

参考答案：

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

第三部分：原生 JS

1、必考：ES6 语法知道哪些，分别怎么用?

举例法，let const 箭头函数 Promise 展开操作符 默认参数 import export，见方方整理的列表

参考答案：

https://juejin.im/post/5e9d0ad1e51d4547144282cd

2、必考 Promise、Promise.all、Promise.race 分别怎么用?

> Promise 对象用于表示一个异步操作的最终完成 (或失败), 及其结果值.

- 背代码 Promise 用法

```
function fn() {
  return new Promise((resolve, reject) => {
    // 成功时调用 resolve(数据)
    // 失败时调用 reject(错误)
  });
}
fn().then(success, fail).then(success2, fail2);
```

- 背代码 Promise.all 用法

```
// promise1和promise2都成功才会调用success1
Promise.all([promise1, promise2]).then(success1, fail1);
```

- 背代码 Promise.race 用法

```
// promise1和promise2只要有一个成功就会调用success1;
// promise1和promise2只要有一个失败就会调用fail1;
// 总之，谁第一个成功或失败，就认为是race的成功或失败。
Promise.race([promise1, promise2]).then(success1, fail1);
```

3、必考：手写函数防抖和函数节流

背代码

- 节流

```
// 节流（一段时间执行一次之后，就不执行第二次）
function throttle(fn, delay) {
  let timer = null;
  return function () {
    const context = this;
    if (!timer) {
      timer = setTimeout(() => {
        fn.apply(context, arguments);
        timer = null;
      }, delay);
    }
  };
}
window.addEventListener(
  "scroll",
  throttle(() => {
    console.log("节流");
  }, 1000)
);
```

注意，有些地方认为节流函数不是立刻执行的，而是在冷却时间末尾执行的（相当于施法有吟唱时间），那样说也是对的。

- 防抖

```
// 防抖（一段时间会等，然后带着一起做了）
function debounce(fn, delay) {
  let timer = null;
  return function () {
    const context = this;
    if (timer) {
      window.clearTimeout(timer);
    }
    timer = setTimeout(() => {
      fn.apply(context, arguments);
      timer = null;
    }, delay);
  };
}
window.addEventListener(
  "scroll",
  debounce(() => {
    console.log("防抖");
  }, 1000)
);
```

4、必考：手写 AJAX

背代码

- 完整版

```
var request = new XMLHttpRequest();
request.open("GET", "/a/b/c?name=ff", true);
request.onreadystatechange = function () {
  if (request.readyState === 4 && request.status === 200) {
    console.log(request.responseText);
  }
};
request.send();
```

- 简化版

```
var request = new XMLHttpRequest();
request.open("GET", "/a/b/c?name=ff", true);
request.onload = () => console.log(request.responseText);
request.send();
```

5、必考：这段代码里的 this 是什么?

- 背代码

```
fn(): this => window/global

obj.fn(): this => obj

fn.call(xx): this => xx

fn.apply(xx): this => xx

fn.bind(xx): this => xx

new Fn(): this => 新的对象

fn = ()=> {}: this => 外面的 this
```

- 看调用

《this 的值到底是什么？一次说清楚》

6、必考：闭包/立即执行函数是什么?

https://zhuanlan.zhihu.com/p/22486908

https://zhuanlan.zhihu.com/p/22465092

> 「函数」和「函数内部能访问到的变量」（也叫环境）的总和，就是一个闭包。

> 立即执行函数就是声明一个匿名函数，马上调用这个匿名函数

```
(function () {
  alert("我是匿名函数");
})();
```

7、必考：什么是 JSONP，什么是 CORS，什么是跨域?

- JSONP：跨域请求数据解决方案中的一种，原理是 script 脚本加载不受同源策略的限制
  - JSONP 是通过 script 标签加载数据的方式去获取数据当做 JS 代码来执行
  - 提前在页面上声明一个函数，函数名通过接口传参的方式传给后台，后台解析到函数名后在原始数据上「包裹」这个函数名，发送给前端。换句话说，JSONP 需要对应接口的后端的配合才能实现。
- CORS：跨域资源共享(CORS) 是一种机制，它使用额外的 HTTP 头来告诉浏览器 让运行在一个 origin (domain) 上的 Web 应用被准许访问来自不同源服务器上的指定的资源。当一个资源从与该资源本身所在的服务器不同的域、协议或端口请求一个资源时，资源会发起一个跨域 HTTP 请求
  - CORS （Cross-Origin Resource Sharing，跨域资源共享）是一个系统，它由一系列传输的 HTTP 头组成，这些 HTTP 头决定浏览器是否阻止前端 JavaScript 代码获取跨域请求的响应。
  - 同源安全策略 默认阻止“跨域”获取资源。但是 CORS 给了 web 服务器这样的权限，即服务器可以选择，允许跨域请求访问到它们的资源。
- 跨域：当协议、子域名、主域名、端口号中任意一个不相同时，都算作不同域。不同域之间相互请求资源，就算作“跨域”

8、常考：async/await 怎么用，如何捕获异常?

async 函数是 Generator 函数的语法糖。使用 关键字 async 来表示，在函数内部使用 await 来表示异步。
async 函数返回一个 Promise 对象，可以使用 then 方法添加回调函数。当函数执行的时候，一旦遇到 await 就会先返回，等到异步操作完成，再接着执行函数体内后面的语句

使用 try/catch 捕获异常：

```
async function run() {
  try {
    await Promise.reject(new Error("Oops!"));
  } catch (error) {
    error.message; // "Oops!"
  }
}
```

9、常考：如何实现深拷贝？

背代码，要点：

- 递归
- 判断类型
- 检查环（也叫循环引用）
- 需要忽略原型

参考答案：

https://github.com/mqyqingfeng/Blog/issues/32

```
function deepCopy(obj) {
  if (typeof obj !== "object") return;
  var newObj = obj instanceof Array ? [] : {};
  for (var key in obj) {
    if (obj.hasOwnProperty(key)) {
      newObj[key] =
        typeof obj[key] === "object" ? deepCopy(obj[key]) : obj[key];
    }
  }
  return newObj;
}
```

10、常考：如何用正则实现 trim()?

```
function trim(string) {
  return string.replace(/^\s+|\s+$/g, "");
}
```

11、常考：不用 class 如何实现继承？用 class 又如何实现？

- 不用 class

```
function Animal(color) {
  this.color = color;
}
Animal.prototype.move = function () {}; // 动物可以动
function Dog(color, name) {
  Animal.call(this, color); // 或者 Animal.apply(this, arguments)
  this.name = name;
}
// 下面三行实现 Dog.prototype.__proto__ = Animal.prototype
function temp() {}
temp.prototype = Animal.prototype;
Dog.prototype = new temp();

Dog.prototype.constuctor = Dog; // 这行看不懂就算了，面试官也不问
Dog.prototype.say = function () {
  console.log("汪");
};

var dog = new Dog("黄色", "阿黄");
```

- 用 class

```
class Animal {
  constructor(color) {
    this.color = color;
  }
  move() {}
}
class Dog extends Animal {
  constructor(color, name) {
    super(color);
    this.name = name;
  }
  say() {}
}
```

12、常考：如何实现数组去重?

https://segmentfault.com/a/1190000016418021

```
function uniqueArray(arr) {
  var newArray = [];
  for (var i = 0; i < arr.length; i++) {
    if (newArray.indexOf(arr[i]) < 0) {
      newArray.push(arr[i]);
    }
  }
  return newArray;
}

function uniqueArray2(arr) {
  return arr.filter(function (item, index, arr) {
    return arr.indexOf(item) === index;
  });
}

[...new Set(arr)];
```

13、放弃：== 相关题目（反着答）

日常工作中只使用===，抛弃使用==，因为坑太多

14、送命题：手写一个 Promise

放弃，可以去看看别人的实现，然后说说 promise 的大概原理

第四部分：DOM

1、必考：事件委托

- 错误版（但是可能能过）

```
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

- 高级版

```
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;
}
```

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

2、用 mouse 事件写一个可拖曳的 div

```
<div id="element"></div>


var dragging = false;
var position = null;

element.addEventListener("mousedown", function (e) {
  dragging = true;
  position = [e.clientX, e.clientY];
});

document.addEventListener("mousemove", function (e) {
  if (dragging) {
    var x = e.clientX;
    var y = e.clientY;
    var deltaX = x - position[0];
    var deltaY = y - position[1];
    var left = parseInt(element.style.left || 0);
    var top = parseInt(element.style.top || 0);
    element.style.left = left + deltaX + "px";
    element.style.top = top + deltaY + "px";
    position = [x, y];
  }
});
document.addEventListener("mouseup", function (e) {
  dragging = false;
});
```

第五部分：HTTP

1、必考：HTTP 状态码知道哪些？分别什么意思？

- 2xx 表示成功
- 3xx 表示需要进一步操作
- 4xx 表示浏览器方面出错
- 5xx 表示服务器方面出错

常见的 HTTP 状态码：

- 200 - 请求成功
- 301 - 资源（网页等）被永久转移到其它 URL
- 404 - 请求的资源（网页等）不存在
- 500 - 内部服务器错误

2、大公司必考：HTTP 缓存有哪几种？

- 需要详细的了解 ETag、CacheControl、Expires 的异同
- 参考 https://imweb.io/topic/5795dcb6fb312541492eda8c
- 答题要点：
  - ETag 是通过对比浏览器和服务器资源的特征值（如 MD5）来决定是否要发送文件内容，如果一样就只发送 304（not modified）
  - Expires 是设置过期时间（绝对时间），但是如果用户的本地时间错乱了，可能会有问题
  - CacheControl: max-age=3600 是设置过期时长（相对时间），跟本地时间无关。

3、必考：GET 和 POST 的区别

- 错解，但是能过面试
  - GET 在浏览器回退时是无害的，而 POST 会再次提交请求。
  - GET 产生的 URL 地址可以被加入收藏栏，而 POST 不可以。
  - GET 请求会被浏览器主动 cache，而 POST 不会，除非手动设置。
  - GET 请求只能进行 url 编码，而 POST 支持多种编码方式。
  - GET 请求参数会被完整保留在浏览器历史记录里，而 POST 中的参数不会被保留。
  - GET 请求在 URL 中传送的参数是有长度限制的，而 POST 么有。
  - 对参数的数据类型，GET 只接受 ASCII 字符，而 POST 没有限制。
  - 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。
  - GET 参数通过 URL 传递，POST 放在 Request body 中。
- 正解：就一个区别，语义——GET 用于获取资源，POST 用于提交资源

4、Cookie V.S. LocalStorage V.S. SessionStorage V.S. Session

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

第六部分：Vue

1、必考：watch 和 computed 和 methods 区别是什么?

- 思路：先翻译单词，再阐述作用，最后强行找不同。
- 要点：
  - computed 和 methods 相比，最大区别是 computed 有缓存：如果 computed 属性依赖的属性没有变化，那么 computed 属性就不会重新计算。methods 则是看到一次计算一次。
  - watch 和 computed 相比，computed 是计算出一个属性（废话），而 watch 则可能是做别的事情（如上报数据）

2、必考：Vue 有哪些生命周期钩子函数？分别有什么用?

beforeCreate created beforeMount mounted beforeUpdate updated beforeDestroy destroyed

- 钩子在文档全都有，看红色的字
- 把名字翻译一遍就是满分
- 要特别说明哪个钩子里请求数据，答案是 mounted

3、必考：Vue 如何实现组件间通信?

- 父子组件：使用 v-on 通过事件通信
- 爷孙组件：使用两次 v-on 通过爷爷爸爸通信，爸爸儿子通信实现爷孙通信
- 任意组件：使用 `eventBus = new Vue()` 来通信， `eventBus.$on` 和 `eventBus.$emit` 是主要 API
- 任意组件：使用 Vuex 通信

4、必考：Vue 数据响应式怎么做到的?

- 答案在文档深入响应式原理
- 要点
  - 使用 Object.defineProperty 把这些属性全部转为 getter/setter

- Vue 不能检测到对象属性的添加或删除，解决方法是手动调用 Vue.set 或者 this.$set

5、必考：Vue.set 是做什么用的?

见上一题

6、Vuex 你怎么用的?

文档

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

7、VueRouter 你怎么用的?

文档

- 背下文档第一句：Vue Router 是 Vue.js 官方的路由管理器。
- 说出核心概念的名字和作用：History 模式/导航守卫/路由懒加载
- 说出常用 API：
  - router-link
  - router-view
  - this.$router.push
  - this.$router.replace
  - this.$route.params

```
this.$router.push("/user-admin");
this.$route.params;
```

8、路由守卫（导航守卫）是什么?

“导航”表示路由正在发生改变，vue-router 提供的导航守卫主要用来通过跳转或取消的方式守卫导航。有多种机会植入路由导航过程中：全局的, 单个路由独享的, 或者组件级的

> 简单的说，导航守卫就是路由跳转过程中的一些钩子函数。路由跳转是一个大的过程，这个大的过程分为跳转前中后等等细小的过程，在每一个过程中都有一函数，这个函数能让你操作一些其他的事儿，这就是导航守卫。类似于组件生命周期钩子函数

第七部分：React

1、必考：受控组件 V.S. 非受控组件

```
<Input value={x} onChange={fn}/> // 受控组件
<Input defaultValue={x} ref={input}/> // 非受控组件
```

区别：受控组件的状态由开发者维护，非受控组件的状态由组件自身维护（不受开发者控制）

2、必考：React 有哪些生命周期函数？分别有什么用？（Ajax 请求放在哪个阶段？）

答题思路跟 Vue 的一样：

- 钩子在文档里，蓝色框框里面的都是生命周期钩子
- 把名字翻译一遍就是满分
- 要特别说明哪个钩子里请求数据，答案是 componentDidMount

参考答案：

- constructor：组件构造函数，第一个被执行
- render：React 中最核心的方法，一个组件中必须要有这个方法
- componentDidMount：组件装载之后调用，此时我们可以获取到 DOM 节点并操作，比如对 canvas，svg 的操作，服务器请求，订阅都可以写在这个里面，但是记得在 componentWillUnmount 中取消订阅
- componentDidUpdate：在这个函数里我们可以操作 DOM，和发起服务器请求，还可以 setState，但是注意一定要用 if 语句控制，否则会导致无限循环
- componentWillUnmount：当我们的组件被卸载或者销毁了就会调用，我们可以在这个函数里去清除一些定时器，取消网络请求，清理无效的 DOM 元素等垃圾清理工作

3、必考：React 如何实现组件间通信?

- 父子靠 props 传函数
- 爷孙可以穿两次 props
- 任意组件用 Redux（也可以自己写一个 eventBus）

4、必考：shouldComponentUpdate 有什么用?

- 要点：用于在没有必要更新 UI 的时候返回 false，以提高渲染性能
- 参考：http://taobaofed.org/blog/2016/08/12/optimized-react-components/

5、必考：虚拟 DOM 是什么?

- 要点：虚拟 DOM 就是用来模拟 DOM 的一个对象，这个对象拥有一些重要属性，并且更新 UI 主要就是通过对比（DIFF）旧的虚拟 DOM 树 和新的虚拟 DOM 树的区别完成的。
- 参考：http://www.alloyteam.com/2015/10/react-virtual-analysis-of-the-dom/

6、必考：什么是高阶组件?

- 要点：文档原话——高阶组件就是一个函数，且该函数接受一个组件作为参数，并返回一个新的组件。
- 举例：React-Redux 里 connect 就是一个高阶组件，比如 connect(mapState)(MyComponent) 接受组件 MyComponent，返回一个具有状态的新 MyComponent 组件。

7、React diff 的原理是什么?

虚拟 Dom 算法的实现是以下三步：

- 通过 JS 来模拟生成虚拟 Dom 树
- 判断两个树的差异
- 渲染差异

8、必考 Redux 是什么?

- 背下文档第一句：Redux 是 JavaScript 状态容器，提供可预测化的状态管理。重点是『状态管理』。
- 说出核心概念的名字和作用：Action/Reducer/Store/单向数据流
- 说出常用 API：store.dispatch(action)/store.getState()

9、connect 的原理是什么?

react-redux 库提供的一个 API，connect 的作用是让你把组件和 store 连接起来，产生一个新的组件（connect 是高阶组件）
参考：https://segmentfault.com/a/1190000017064759

第八部分：TypeScript

1、never 类型是什么?

不应该出现的类型

2、TypeScript 比起 JavaScript 有什么优点?

提供了类型约束，因此更可控、更容易重构、更适合大型项目、更容易维护

第九部分：Webpack

1、必考：有哪些常见 loader 和 plugin，你用过哪些?

- loader
  - file-loader：把文件输出到一个文件夹中，在代码中通过相对 URL 去引用输出的文件

    - url-loader：和 file-loader 类似，但是能在文件很小的情况下以 base64 的方式把文件内容注入到代码中去
    - source-map-loader：加载额外的 Source Map 文件，以方便断点调试
    - image-loader：加载并且压缩图片文件
    - babel-loader：把 ES6 转换成 ES5
    - css-loader：加载 CSS，支持模块化、压缩、文件导入等特性
    - style-loader：把 CSS 代码注入到 JavaScript 中，通过 DOM 操作去加载 CSS。
    - eslint-loader：通过 ESLint 检查 JavaScript 代码
- plugin
    - html-webpack-plugin：创建一个 html 文件，并把 webpack 打包后的静态文件自动插入到这个 html 文件当中
    - define-plugin：定义环境变量
    - commons-chunk-plugin：提取公共代码
    - uglifyjs-webpack-plugin：通过 UglifyES 压缩 ES6 代码

2、英语题：loader 和 plugin 的区别是什么?

- Loader 直译为"加载器"。Webpack 将一切文件视为模块，但是 webpack 原生是只能解析 js 文件，如果想将其他文件也打包的话，就会用到 loader。 所以 Loader 的作用是让 webpack 拥有了加载和解析非 JavaScript 文件的能力。
- Plugin 直译为"插件"。Plugin 可以扩展 webpack 的功能，让 webpack 具有更多的灵活性。 在 Webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。

3、必考：如何按需加载代码?

通过 import()语句来控制加载时机，webpack 内置了对于 import()的解析，会将 import()中引入的模块作为一个新的入口在生成一个 chunk。 当代码执行到 import()语句时，会去加载 Chunk 对应生成的文件。import()会返回一个 Promise 对象，所以为了让浏览器支持，需要事先注入 Promise polyfill

4、必考：如何提高构建速度?

- 多入口情况下，使用 CommonsChunkPlugin 来提取公共代码
- 通过 externals 配置来提取常用库
- 利用 DllPlugin 和 DllReferencePlugin 预编译资源模块 通过 DllPlugin 来对那些我们引用但是绝对不会修改的 npm 包来进行预编译，再通过 DllReferencePlugin 将预编译的模块加载进来。
- 使用 Happypack 实现多线程加速编译
- 使用 webpack-uglify-parallel 来提升 uglifyPlugin 的压缩速度。 原理上 webpack-uglify-parallel 采用了多核并行压缩来提升压缩速度
- 使用 Tree-shaking 和 Scope Hoisting 来剔除多余代码

5、转义出的文件过大怎么办？如何利用 webpack 来优化前端性能？（提高性能和体验）

- 压缩代码。删除多余的代码、注释、简化代码的写法等等方式。可以利用 webpack 的 UglifyJsPlugin 和 ParallelUglifyPlugin 来压缩 JS 文件， 利用 cssnano（css-loader?minimize）来压缩 css
- 利用 CDN 加速。在构建过程中，将引用的静态资源路径修改为 CDN 上对应的路径。可以利用 webpack 对于 output 参数和各 loader 的 publicPath 参数来修改资源路径
- 删除死代码（Tree Shaking）。将代码中永远不会走到的片段删除掉。可以通过在启动 webpack 时追加参数--optimize-minimize 来实现
- 提取公共代码

第十部分：安全

1、必考：什么是 XSS？如何预防？

- 正常用户 A 提交正常内容，显示在另一个用户 B 的网页上，没有问题。
- 恶意用户 H 提交恶意内容，显示在另一个用户 B 的网页上，对 B 的网页随意篡改。

造成 XSS 有几个要点：

- 恶意用户可以提交内容
- 提交的内容可以显示在另一个用户的页面上
- 这些内容未经过滤，直接运行在另一个用户的页面上

XSS (Cross Site Scripting) 即跨站脚本。比如在有表单输入的地方，用户输入了类似 `<script>alert("ok")</script>` 的东西，而服务器没有经过过滤就保存下来了，别的用户看到网页就会弹出提示框。当然，xss 攻击还可以做其他更加复杂的事情。比如，获取 cookie 什么的

把"<"和">"转换为 html 字符实体可以防范大多数 xss 攻击。可以使用 js 过滤，也可以在服务器端过滤

2、必考：什么是 CSRF？如何预防？

CSRF (Cross Site Request Forgery)攻击，中文名：跨站请求伪造。其原理是攻击者构造网站后台某个功能接口的请求地址，诱导用户去点击或者用特殊方法让该请求地址自动加载。用户在登录状态下这个请求被服务端接收后会被误以为是用户合法的操作。对于 GET 形式的接口地址可轻易被攻击，对于 POST 形式的接口地址也不是百分百安全，攻击者可诱导用户进入带 Form 表单可用 POST 方式提交参数的页面

客户端防范：对于数据库的修改请求，全部使用 POST 提交，禁止使用 GET 请求
服务器端防范：一般的做法是在表单里面添加一段隐藏的唯一的 token(请求令牌)

- 服务端在收到路由请求时，生成一个随机数，在渲染请求页面时把随机数埋入页面（一般埋入 form 表单内，`<input type="hidden" name="_csrf_token" value="xxxx">`）
- 服务端设置 setCookie，把该随机数作为 cookie 或者 session 种入用户浏览器
- 当用户发送 GET 或者 POST 请求时带上_csrf_token 参数（对于 Form 表单直接提交即可，因为会自动把当前表单内所有的 input 提交给后台，包括_csrf_token）
- 后台在接受到请求后解析请求的 cookie 获取_csrf_token 的值，然后和用户请求提交的_csrf_token 做个比较，如果相等表示请求是合法的

第十一部分：算法

1、排序算法

参考：https://github.com/hustcc/JS-Sorting-Algorithm

- 冒泡排序

算法步骤：

1. 比较相邻的元素。如果第一个比第二个大，就交换它们两个
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数
3. 针对所有的元素重复以上的步骤，除了最后一个
4. 重复步骤 1~3，直到排序完成

JavaScript 代码实现：

```
function bubbleSort(arr) {
  var len = arr.length;
  for (var i = 0; i < len - 1; i++) {
    for (var j = 0; j < len - 1 - i; j++) {
      if (arr[j] > arr[j + 1]) {
        // 相邻元素两两对比
        var temp = arr[j + 1]; // 元素交换
        arr[j + 1] = arr[j];
        arr[j] = temp;
      }
    }
  }
  return arr;
}
```

- 选择排序

算法步骤：

1. 首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置
2. 再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾
3. 重复第二步，直到所有元素均排序完毕

JavaScript 代码实现：

```
function selectionSort(arr) {
  var len = arr.length;
  var minIndex, temp;
  for (var i = 0; i < len - 1; i++) {
    minIndex = i;
    for (var j = i + 1; j < len; j++) {
      if (arr[j] < arr[minIndex]) {
        // 寻找最小的数
        minIndex = j; // 将最小数的索引保存
      }
    }
    temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
  return arr;
}
```

2、二分查找

二分法查找又称折半查找，需要数据排好顺序

1. 首先，从有序数组的中间的元素开始搜索，如果该元素正好是目标元素（即要查找的元素），则搜索过程结束，否则进行下一步
2. 如果目标元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半区域查找，然后重复第一步的操作
3. 如果某一步数组为空，则表示找不到目标元素

```
// arr为已按"升序排列"的数组，key为要查询的元素
// 返回目标元素的下标

// 非递归算法
function binarySearch(arr, key) {
  let low = 0;
  let high = arr.length - 1;
  while (low <= high) {
    let mid = parseInt((high + low) / 2);
    if (key === arr[mid]) {
      return mid;
    } else if (key > arr[mid]) {
      low = mid + 1;
    } else if (key < arr[mid]) {
      high = mid - 1;
    } else {
      return -1;
    }
  }
}
// 递归算法
function binarySearch(arr, low, high, key) {
  if (low > high) {
    return -1;
  }
  let mid = parseInt((high + low) / 2);
  if (arr[mid] === key) {
    return mid;
  } else if (arr[mid] > key) {
    high = mid - 1;
    return binarySearch(arr, low, high, key);
  } else if (arr[mid] < key) {
    low = mid + 1;
    return binarySearch(arr, low, high, key);
  }
}
```

第十二部分：手写一个 xxx

都是一些常考的手写代码，这里只列出简单版实现

1、实现一个双向数据绑定

参考：
https://juejin.im/post/5d26e368e51d4577407b1dd7
https://juejin.im/post/5acd0c8a6fb9a028da7cdfaf

```
let obj = {};
let input = document.getElementById("input");
let span = document.getElementById("span");
// 数据劫持
Object.defineProperty(obj, "text", {
  configurable: true,
  enumerable: true,
  get() {
    console.log("获取数据了");
  },
  set(newVal) {
    console.log("数据更新了");
    input.value = newVal;
    span.innerHTML = newVal;
  },
});
// 输入监听
input.addEventListener("keyup", function (e) {
  obj.text = e.target.value;
});
```

2、实现一个简单路由

参考：https://juejin.im/post/5ac61da66fb9a028c71eae1b

```
// hash路由
class Route {
  constructor() {
    // 路由存储对象
    this.routes = {};
    // 当前hash
    this.currentHash = "";
    // 绑定this，避免监听时this指向改变
    this.freshRoute = this.freshRoute.bind(this);
    // 监听
    window.addEventListener("load", this.freshRoute, false);
    window.addEventListener("hashchange", this.freshRoute, false);
  }
  // 存储
  storeRoute(path, cb) {
    this.routes[path] = cb || function () {};
  }
  // 更新
  freshRoute() {
    this.currentHash = location.hash.slice(1) || "/";
    this.routes[this.currentHash]();
  }
}
```

3、实现懒加载

```
<ul>
  <li><img src="./imgs/default.png" data="./imgs/1.png" alt="" /></li>
  <li><img src="./imgs/default.png" data="./imgs/2.png" alt="" /></li>
  <li><img src="./imgs/default.png" data="./imgs/3.png" alt="" /></li>
  <li><img src="./imgs/default.png" data="./imgs/4.png" alt="" /></li>
  <li><img src="./imgs/default.png" data="./imgs/5.png" alt="" /></li>
  <li><img src="./imgs/default.png" data="./imgs/6.png" alt="" /></li>
  <li><img src="./imgs/default.png" data="./imgs/7.png" alt="" /></li>
  <li><img src="./imgs/default.png" data="./imgs/8.png" alt="" /></li>
  <li><img src="./imgs/default.png" data="./imgs/9.png" alt="" /></li>
  <li><img src="./imgs/default.png" data="./imgs/10.png" alt="" /></li>
</ul>
```

```
let imgs = document.querySelectorAll("img");
// 可视区高度
let clientHeight =
  window.innerHeight ||
  document.documentElement.clientHeight ||
  document.body.clientHeight;
function lazyLoad() {
  // 滚动卷去的高度
  let scrollTop =
    window.pageYOffset ||
    document.documentElement.scrollTop ||
    document.body.scrollTop;
  for (let i = 0; i < imgs.length; i++) {
    // 图片在可视区冒出的高度
    let x = clientHeight + scrollTop - imgs[i].offsetTop;
    // 图片在可视区内
    if (x > 0 && x < clientHeight + imgs[i].height) {
      imgs[i].src = imgs[i].getAttribute("data");
    }
  }
}
// addEventListener('scroll', lazyLoad) or setInterval(lazyLoad, 1000)
```

4、rem 基本设置

```
// 提前执行，初始化 resize 事件不会执行
setRem();
// 原始配置
function setRem() {
  let doc = document.documentElement;
  let width = doc.getBoundingClientRect().width;
  // 假设设计稿为宽750，则rem为10px
  let rem = width / 75;
  doc.style.fontSize = rem + "px";
}
// 监听窗口变化
addEventListener("resize", setRem);
```

5、手写实现 ajax

```
// 实例化
let xhr = new XMLHttpRequest();
// 初始化
xhr.open(method, url, async);
// 发送请求
xhr.send(data);
// 设置状态变化回调处理请求结果
xhr.onreadystatechange = () => {
  if (xhr.readyStatus === 4 && xhr.status === 200) {
    console.log(xhr.responseText);
  }
};
```

6、Object.create 的基本实现原理

```
// 思路：将传入的对象作为原型
function create(obj) {
  function F() {}
  F.prototype = obj;
  return new F();
}
```

7、instanceof 的原理

```
// 思路：右边变量的原型存在于左边变量的原型链上
function instanceOf(left, right) {
  let leftValue = left.__proto__;
  let rightValue = right.prototype;
  while (true) {
    if (leftValue === null) {
      return false;
    }
    if (leftValue === rightValue) {
      return true;
    }
    leftValue = leftValue.__proto__;
  }
}
```

8、实现一个 call 函数

```
// 思路：将要改变this指向的方法挂到目标this上执行并返回
Function.prototype.mycall = function (context) {
  if (typeof this !== "function") {
    throw new TypeError("not funciton");
  }
  context = context || window;
  context.fn = this;
  let arg = [...arguments].slice(1);
  let result = context.fn(...arg);
  delete context.fn;
  return result;
};
```

9、实现一个 apply 函数

```
// 思路：将要改变this指向的方法挂到目标this上执行并返回
Function.prototype.myapply = function (context) {
  if (typeof this !== "function") {
    throw new TypeError("not funciton");
  }
  context = context || window;
  context.fn = this;
  let result;
  if (arguments[1]) {
    result = context.fn(...arguments[1]);
  } else {
    result = context.fn();
  }
  delete context.fn;
  return result;
};
```

10、实现一个 bind 函数

```
// 思路：类似call，但返回的是函数
Function.prototype.mybind = function (context) {
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }
  let _this = this;
  let arg = [...arguments].slice(1);
  return function F() {
    // 处理函数使用new的情况
    if (this instanceof F) {
      return new _this(...arg, ...arguments);
    } else {
      return _this.apply(context, arg.concat(...arguments));
    }
  };
};
```

11、实现 Promise

参考：https://juejin.im/post/5b2f02cd5188252b937548ab

```
// 未添加异步处理等其他边界情况
// ①自动执行函数，②三个状态，③then
class Promise {
  constructor(fn) {
    // 三个状态
    this.state = "pending";
    this.value = undefined;
    this.reason = undefined;
    let resolve = (value) => {
      if (this.state === "pending") {
        this.state = "fulfilled";
        this.value = value;
      }
    };
    let reject = (value) => {
      if (this.state === "pending") {
        this.state = "rejected";
        this.reason = value;
      }
    };
    // 自动执行函数
    try {
      fn(resolve, reject);
    } catch (e) {
      reject(e);
    }
  }
  // then
  then(onFulfilled, onRejected) {
    switch (this.state) {
      case "fulfilled":
        onFulfilled(this.value);
        break;
      case "rejected":
        onRejected(this.value);
        break;
      default:
    }
  }
}
```

12、实现 new 方法

```
/**
 * 1. 创建一个新的对象
 * 2. 链接到原型
 * 3. 绑定this
 * 4. 返回一个新的对象
 */
function create() {
  let obj = {};
  let Constructor = [].shift.call(arguments);
  obj.__proto__ = Constructor.prototype;
  let result = Constructor.apply(obj, arguments);
  return typeof result === "object" ? result : obj;
}
```

13、使用 setTimeout 模拟 setInterval

```
// 可避免setInterval因执行时间导致的间隔执行时间不一致
setTimeout(function () {
  // do something
  setTimeout(arguments.callee, 500);
}, 500);
```

14、实现 LazyMan 类

```
class LazyManClass {
  constructor(name) {
    this.name = name;
    this.queue = [];
    console.log(`Hi I am ${name}`);
    setTimeout(() => {
      this.next();
    }, 0);
  }
  sleepFirst(time) {
    const fn = () => {
      setTimeout(() => {
        console.log(`等待了${time}秒...`);
        this.next();
      }, time * 1000);
    };
    this.queue.unshift(fn);
    return this;
  }
  sleep(time) {
    const fn = () => {
      setTimeout(() => {
        console.log(`等待了${time}秒...`);
        this.next();
      }, time * 1000);
    };
    this.queue.push(fn);
    return this;
  }
  eat(food) {
    const fn = () => {
      console.log(`I am eating ${food}`);
      this.next();
    };
    this.queue.push(fn);
    return this;
  }
  next() {
    const fn = this.queue.shift();
    fn && fn();
  }
}
function LazyMan(name) {
  return new LazyManClass(name);
}

LazyMan("Tony")
  .eat("lunch")
  .eat("dinner")
  .sleepFirst(5)
```

```
  .sleep(4)
  .eat("junk food");
```

15、实现一个 EventEmitter

```
class EventEmitter {
  constructor() {
    this.events = {};
  }
  on(name, fn) {
    if (!this.events[name]) {
      this.events[name] = [];
    }
    this.events[name].push(fn);
  }
  off(name, fn) {
    if (this.events[name]) {
      this.events[name] = this.events[name].filter((item) => item !== fn);
    }
  }
  emit(name, ...args) {
    if (this.events[name]) {
      this.events[name].forEach((item) => item.apply(this, args));
    }
  }
  once(name, fn) {
    const only = () => {
      fn.apply(this, arguments);
      this.off(name, only);
    };
    this.on(name, only);
  }
}
```

16、实现一个 sleep 函数

sleep 函数的作用就是延迟指定时间后再执行接下来的函数，用 promise 很好实现

参考：https://segmentfault.com/a/1190000020848472

```
function sleep(time) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(true);
    }, time * 1000);
  });
}
```