视频云-前端面试题库（一面）

> 此文档汇总了大家常用的面试题，希望大家有合适的题都向里面扩充

CSS

垂直水平居中（flex、position）

问题：编写样式表，使得 #child 节点在 #parent 节点中垂直水平居中

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

1. flex 追问

实现骰子3点布局

```
示例：
·
   ·
      ·

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

2. 绝对定位 追问

position的取值：static(默认) relative absolute sticky

staic-默认位置；元素会像通常那样流入页面。顶部，底部，左，右，z-index属性不适用。

relative-元素的位置相对于自身进行调整，而不改变布局（从而为未被定位的元素留下一个空白）。

absolute-该元素从页面的流中移除，并相对于其最近位置的祖先定位（非static）在指定位置，如果有的话，或者与初始包含块相对。绝对定位的框可以有边距，并且不会与其他边距折叠。这些元素不影响其他元素的位置。

fixed元素是定位在相对于窗口。

sticky，是相对定位和固定定位的混合。该元素被视为相对位置，直到它越过指定的阈值，此时它被视为固定位置。

实现1：1正方形的方式

```
<div class="box">
    <p>这是一个自适应的正方形</p>
</div>
```

1、vw

```
width: 20vw;
height: 20vw;
```

2、设置盒子的padding-bottom

```
width: 20%;
/* 设置height为0 ，避免盒子被内容撑开多余的高度 */
height: 0px;
/* 把盒子的高撑开，和width设置同样的固定的宽度或者百分比，百分比相对的是父元素盒子的宽度 */
padding-bottom: 20%;

```

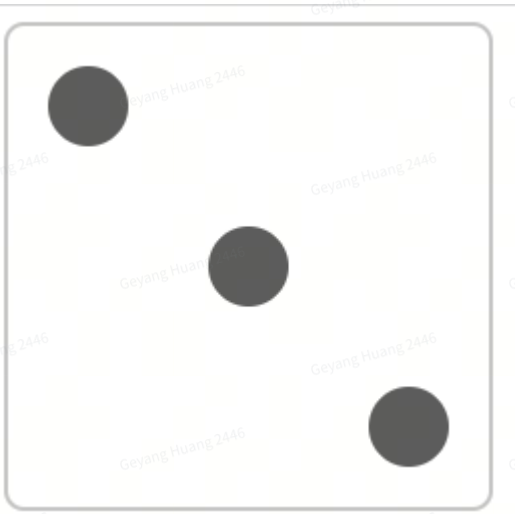
实现一个左右宽度固定，中间宽度自适应的三列布局的方法

```
<style>
.wrap {
    width: 100%;
}
.left,
.right {
    width: 100px;
}
.center {}
</style>
<div class="wrap">
  <div class="left"></div>
```

```
    <div class="center"></div>
    <div class="right"></div>
</div>
```

方法很多，可以让候选人多说几种方法

实现一个如图所示的骰子



```
<div class="box">
    <span class="item"></span>
    <span class="item"></span>
    <span class="item"></span>
</div>

 <style>
     /*BFC实现*/
    .box{
        width: 200px;
        height: 200px;
        border: 1px solid #ccc;
        border-radius: 10px;
        padding: 20px;
        position: relative;
    }
    .item{
        width: 40px;
        height: 40px;
        background-color: #666;
        border-radius: 50%;
        display: block;
    }
    .item:nth-child(2){
```

```
            position: absolute;
            top: 100px;
            left: 100px;
        }

        .item:nth-child(3){
            position: absolute;
            bottom: 20px;
            right: 20px;
        }

    </style>

    <style type="text/css">
        /*flex实现*/
        .box {
            width: 200px;
            height: 200px;
            border: 2px solid #ccc;
            border-radius: 10px;
            padding: 20px;
            display: flex;
            justify-content: space-between;
        }

        .item {
            width: 40px;
            height: 40px;
            border-radius: 50%;
            background-color: #666;
        }

        .item:nth-child(2) {
            align-self: center;
        }

        .item:nth-child(3) {
            align-self: flex-end;
        }
    </style>

     <style>
         /*grid实现*/
        .box {
            width: 200px;
            height: 200px;
            border: 2px solid #ccc;
```

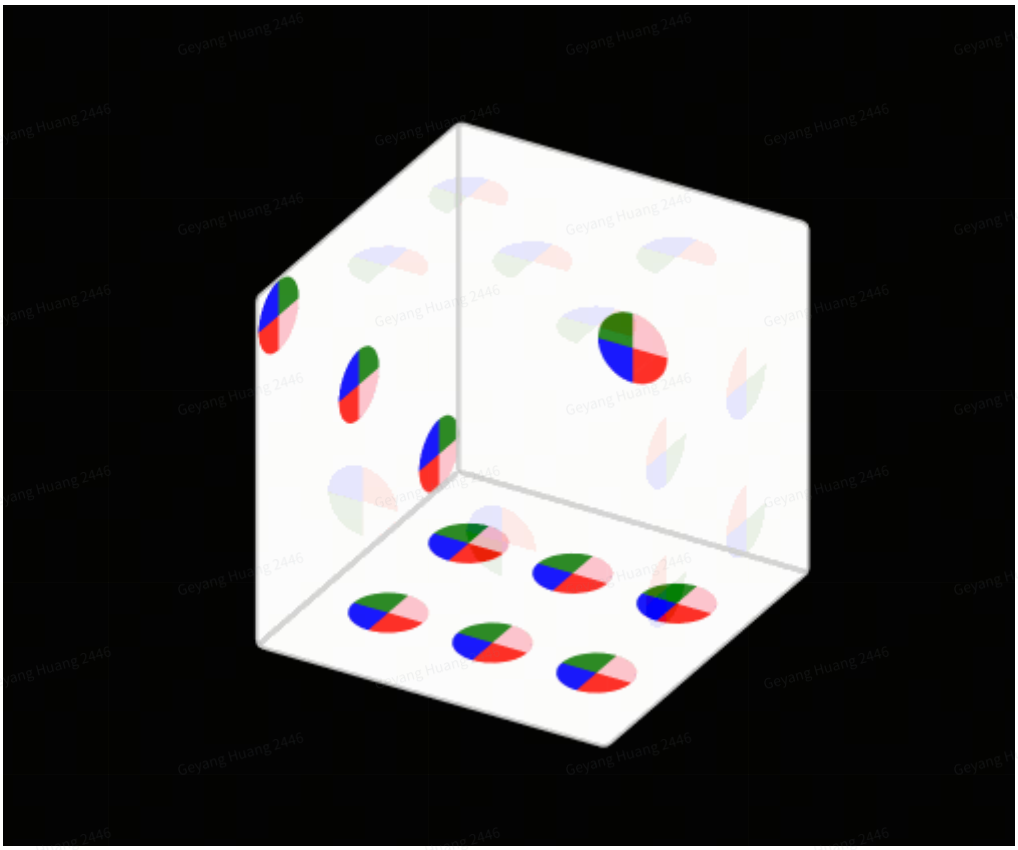
```
            border-radius: 10px;
            padding: 20px;
            display: grid;
            grid-template-rows: 1fr 1fr 1fr;
            grid-template-columns: 1fr 1fr 1fr;
            /* grid-template-rows: 33.33% 33.33% 33.33%;; */
            /* grid-template-columns: 33.33% 33.33% 33.33%;; */
            justify-items: center;
        }

        .item {
            width: 40px;
            height: 40px;
            border-radius: 50%;
            background-color: #666;
            align-self: center;
            grid-row: 1 / 2;
            grid-column: 1 / 2;
        }

        .item:nth-child(2) {
            grid-row: 2 / 3;
            grid-column: 2 / 3;
        }

        .item:nth-child(3) {
            grid-row: 3 / 4;
            grid-column: 3/ 4;
        }
    </style>
```

> 本题写出bfc/flex/grid任一种即可，可以让候选人都写一下或者说一下实现思路；
> 本题拓展(资深)，实现一个立体的骰子(六面体)，可结合动画进行考察（如下图）



```
<!-- 给出HTML结构 -->
<div class="cube">
    <div class="box box1">
        <span class="item"></span>
    </div>
    <div class="box box2">
        <span class="item"></span>
        <span class="item"></span>
    </div>
    <div class="box box3">
        <span class="item"></span>
        <span class="item"></span>
        <span class="item"></span>
    </div>
    <div class="box box4">
        <div>
            <span class="item"></span>
            <span class="item"></span>
        </div>
        <div>
            <span class="item"></span>
            <span class="item"></span>
        </div>
    </div>
    <div class="box box5">
        <div>
```

```
            <span class="item"></span>
            <span class="item"></span>
        </div>
        <div>
            <span class="item"></span>
        </div>
        <div>
            <span class="item"></span>
            <span class="item"></span>
        </div>
    </div>
    <div class="box box6">
        <div>
            <span class="item"></span>
            <span class="item"></span>
        </div>
        <div>
            <span class="item"></span>
            <span class="item"></span>
        </div>
        <div>
            <span class="item"></span>
            <span class="item"></span>
        </div>
    </div>
</div>

 <style>
 /*以下flex实现，也可以用grid实现*/
  body {
    margin: 0;
    padding: 0;
    background-color: #000;
  }

  .cube {
    width: 200px;
    height: 200px;
    margin: 200px auto;
    transform-style: preserve-3d;
    transform: rotateX(45deg) rotateZ(45deg);
    animation: rotate 5s infinite linear;
  }

  .cube:hover {
    animation-play-state: paused;
  }
```

```
74
75      @keyframes rotate {
76        from {
77          transform: rotateX(0) rotateZ(0);
78        }
79
80        to {
81          transform: rotateX(360deg) rotateZ(360deg);
82        }
83      }
84
85      .box {
86        width: 200px;
87        height: 200px;
88        border: 2px solid #ccc;
89        border-radius: 5px;
90        background-color: #fff;
91        display: flex;
92        opacity: 0.9;
93        position: absolute;
94      }
95
96      .item {
97        width: 0;
98        height: 0;
99        border: 20px solid transparent;
100       border-radius: 20px;
101       border-top-color: green;
102       border-bottom-color: red;
103       border-left-color: blue;
104       border-right-color: pink;
105       transform: rotateZ(45deg);
106     }
107
108     .box1 {
109       justify-content: center;
110       align-items: center;
111       transform: rotateY(90deg) translateZ(100px);
112     }
113
114     .box2 {
115       flex-direction: column;
116       justify-content: space-evenly;
117       align-items: center;
118       transform: rotateY(-90deg) translateZ(100px);
119     }
120
```

```
    .box3 {
      justify-content: space-between;
      transform: rotateX(90deg) translateZ(100px);
    }

    .box3 .item:nth-child(2) {
      align-self: center;
    }

    .box3 .item:nth-child(3) {
      align-self: flex-end;
    }

    .box4 {
      justify-content: space-evenly;
      flex-direction: column;
      transform: rotateX(-90deg) translateZ(100px);
    }

    .box4 div {
      display: flex;
      justify-content: space-evenly;
    }

    .box5 {
      flex-direction: column;
      justify-content: space-evenly;
      transform: rotateY(180deg) translateZ(100px);
    }

    .box5 div {
      display: flex;
      justify-content: space-evenly;
    }

    .box6 {
      flex-direction: column;
      justify-content: space-evenly;
      transform: rotateY(0deg) translateZ(100px);
    }

    .box6 div {
      display: flex;
      justify-content: space-evenly;
    }
  </style>
```

判断蓝色区域的宽度和高度

```
<style>
  .box {
    width: 10px;
    height: 10px;
    border: 1px solid red;
    margin: 2px;
    padding: 2px;
    background: blue;
  }

  #borderBox {
    box-sizing: border-box;
  }

  #contentBox {
    box-sizing: content-box;
  }
</style>

<div id="borderBox" class="box"></div>
<div id="contentBox" class="box"></div>
```

考察盒模型的理解，追问盒模型的区别

> 正确答案：8, 14

了解什么实现动画的方法

transition，animation都什么时候使用

如果想实现一个从右到左展开的抽屉效果怎么实现

JS 基础

js的数据类型有哪些

追问：

写一个判断js引用类型是否相等的方法

写一个js深拷贝的函数

~~dom操作~~

问题：添加事件，使点击li元素时，打印出li的内容

```
<ul class="wrap">
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
```

追问：

如果点击的li是最后一个元素时，将该li元素删除

~~this 问题~~

```
class Student{
    constructor(name) {
        this.name = 'Jarry'
    }
    getInfo() {
        return {
            name: 'Tom',
            getName() {
                return this.name
            }
        }
    }
}
const stu = new Student()
stu.getInfo().getName() // 输出是什么?
```

> 正确答案：'Tom'

追问：如何改造 输出 'Jarry'?

```
// ...
// 修改为箭头函数
```

```
  getName: () => {
    return this.name
  }
  // ...
```

构造函数实例化

问题：构造函数在实例化过程中具体做了什么？

答案：生成一个新的对象，将对象的原型设置为构造函数的原型。将构造函数的上下文指向到这个新的对象，然后执行构造函数。如果构造函数没有返回一个对象类型的返回值，那么构造函数会默认返回这个新对象。

拓展

7. 构造函数和函数上下文，下面代码运行结束后 `a` 和 `b` 的值是什么？marvel#1362

```
function Person(name) {
  this.name = name;
}

Person.prototype.print = function() {
  return this.name;
};

Person('abc');
const a = new Person('abc').print.call({});
console.log(a);

const fn = () => {
  this.x = 'z';
};

const b = {x: 'y'};
fn.call(b);
console.log(b);
```

正确答案：a 是 undefined，b 是 `{x: 'y'}`。

箭头函数的 this 指向在函数声明时确定，之后无法更改。加问，在 nodejs 环境下和在浏览器环境下，执行下面的代码会有什么区别？答案：nodejs 环境下 fn 的 this 会指向 module.exports，而浏览器环境下指向 window

setTimeout

问题：setTimeout 能保证回调函数被执行的精确时间吗？什么情况下setTimeout会变得不准

答案：不能。setTimeout 只能保证回调函数至少在指定的时间之后运行，不能保证完全的精确。实际上，setTimeout 会将回调函数放置在一个队列中，js 会定期扫描该队列，在合适的时间点取出回调函数并执行。

追问：

给定一个截止时间戳unix，如何实现一个倒计时，使其在任何情况下都尽量精准

Promise

写出下面程序的打印内容 marvel#1361

```
setTimeout(function() {
  console.log(1)
}, 0);
new Promise(function(resolve) {
  console.log(2);
  for(var i=0 ; i < 10000 ; i++) {
    if (i == 9999) {
      resolve();
    }
  }
  console.log(3);
}).then(function() {
  console.log(4);
});
console.log(5);
```

正确答案：2 3 5 4 1。重点关注：候选人是否把 2 写在第一位，以及 4 和 1 的顺序。

实现 Function.prototype.bind

问题：Function.prototype.bind

答案

```
const bind = (fn, ctx, ...args) => (...restArgs) => fn.call(ctx, ...args, ...restArgs)
```

> 候选人如果没有思路，可以提示使用函数的 call 或者 apply 方法实现

实现 Array.prototype.flat

问题：Array.prototype.flat

答案

```
const flat = array => {
  const _flat = (array, ret) => {
    Array.isArray(array)
      ? array.forEach(item => _flat(item, ret))
      : ret.push(array);
    return ret;
  };
  return _flat(array, []);
};
```

如果候选人没有思路，可以给出测试用例，比如

```
// 补全 flat 函数
const flat = (arr) => {
  // todo
};

flat([1, [2, 3, [4]], [5, 6]]) // 返回 [1,2,3,4,5,6]
```

如果候选人在写递归时卡住了，可以提醒使用一个中间变量储存最终结果。

实现 Array.prototype.fliter

问题：Array.prototype.flter

答案

```
Array.prototype.filter = function(cb, context){
  context = context || this;  //确定上下文，默认为this
  var len = this.length;  //数组的长度
  var r = [];  //最终将返回的结果数组
  for(var i = 0; i < len; i++){
    if(cb.call(context, this[i], i, this)){  //filter回调函数的三个参数：元素值，元素索引，原数组
      r.push(this[i]);
    }
  }
  return r;
};
```

节流和防抖的区别

问题：实现防抖debounce（也可问节流）

答案：

```
const debounce = (fn, timeout) => {
  let timer = 0;
  return (...args) => {
      clearTimeout(timer);
      timer = setTimeout(() => {
        fn(...args);
        timer = 0;
      }, timeout);
  };
};
```

数字格式化

问题：marvel#1047 格式化数字 1234567890 为字符串 '1,234,567,890'

答案

```
const format = num => {
  const str = num.toString();
  let ret = [];
  let i = str.length % 3;
  if (i) {
    ret.push(str.slice(0, i));
  }
  for (;i < str.length; i = i + 3) {
    ret.push(str.slice(i, i + 3));
  }
  return ret.join(',');
};
```

求无重复字符的最长子串

示例 1:

输入: s = "abcabcbb"

输出: 3

解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

示例 2:

输入: s = "bbbbb"

输出: 1

解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

示例 3:

输入: s = "pwwkew"

输出: 3

解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。

请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。

示例 4:

输入: s = ""

输出: 0

答案：

```
var lengthOfLongestSubstring = function(s) {
    let startNumber = 0;
    let result = 0;
    // 滑动窗口
    s.split('').forEach((v, i) => {
        const newStr = s.substring(startNumber, i);
        const sameIndex = newStr.indexOf(v);
        if(sameIndex === -1) {
            result = Math.max(result, (i + 1) - startNumber);
        } else {
            startNumber += (sameIndex + 1);
        }
    });
    return result;
};
```

合并有序数组

问题：marvel#1030 编写函数合并两个有序数组

```
const merge = (a, b) => {
  // todo
};

merge([2,5,6,9], [1,2,3,29]) // 返回 [1,2,2,3,5,6,9,29]
```

回答

```
const merge = (a, b) => {
  let i = 0;
  let j = 0;
  const ret = [];
  while (i < a.length || j < b.length) {
    if (i > a.length) {
```

```
      ret.push(b[j]);
      ++j
    }
    if (j > b.length) {
      ret.push(a[i]);
      ++i
    }
    const x = a[i];
    const y = b[j];
    if (x < y) {
      ret.push(x);
      ++i;
    } else {
      ret.push(y);
      ++j
    }
  }
  return ret;
};
```

链表反转

输入: 1->2->3->4->5->NULL

输出: 5->4->3->2->1->NULL

答案:

```
const reverseList = function(head) {
    let node1 = null;
    let node2 = head;
    if(!node2) {
        return head;
    }
    while(node2) {
        const n = node2.next;
        node2.next = node1;
        node1 = node2;
        node2 = n;
    }
    head = node1;
    return head;
};
```

封装异步请求

问题：marvel#807 封装带有超时（重试）机制的异步请求工具函数

答案

```
const defaultOptions = {
  timeout: 5000,
  retry: 3,
};

const timeout = (timeout = defaultOptions.timeout) => new Promise((_, reject) => setTimeout(() => reject(new Error('timeout')), timeout));

const request = async (url, fetchOption, options = defaultOptions) => {
  let times = 0;
  const _request = () => Promise.race([
    fetch(url, fetchOption),
    timeout(options.timeout),
  ]);
  while (times < (options.retry || defaultOptions.retry)) {
    try {
      return await _request();
    } catch(err) {
      ++times;
      continue;
    }
  }
  throw new Error('fetch failed');
}
```

评分：

8. 至少需要实现超时或重试其中一个功能，2.5 分
9. 两个功能全部实现的 3 分
10. 在 3 分的基础上，能设计出比参考答案更细致的 API 的给 3.5 分

浏览器

从浏览器输入URL到页面渲染发生了什么

考察整体理解，无标准回答，可参考 细说浏览器输入URL后发生了什么

> 参考答案：
>
> - 合成 URL
> - DNS域名解析
> - 建立TCP链接、SSL
> - 发送请求，服务端处理并返回 (验证缓存)
> - 关闭 TCP 连接，四次挥手
> - 浏览器渲染
>   - 资源加载 defer/async
>   - 构建 DOM 树
>   - 样式计算
>   - 页面布局
>   - 生成分层树
>   - 栅格化
>   - 显示

拓展

1. 为什么网页中经常把 css 文件添加在 head，而 js 文件添加在 body 尾部？

答案：因为 head 中的资源会在浏览器解析 body 之前加载并执行完毕。样式表需要在 body 解析前生效，否则用户有可能看到一闪而过的样式崩坏的页面。js 逻辑一般在类似 `DOMContentLoaded` 的事件回调中执行，所以在很多情况下，并不需要把 js 文件放置在 head 头部。因为在浏览器加载 head 中引用的资源时，页面会处于完全空白的状态。放在 body 尾部的原因是，浏览器在解析 body 期间，遇到 js 文件会先等待 js 文件加载执行完毕，然后再解析剩余部分。放置在 body 尾部，可以让页面更快呈现出来，带来更好的用户体验。

2. script 标签上的 defer 和 async 有什么用？浏览器下载脚本资源时不阻塞 dom 的后续渲染。defer 会在 body 全部渲染完毕后执行，async 会在资源下载完成时，停止 dom 渲染并立即执行脚本内容，然后继续渲染 dom

Workers

```
1. web有哪些常见的worker?
```

A: web worker、service worker、shared worker

2. worker的优势和作用?
A: 一个独立于JavaScript主线程的独立线程，在里面执行需要消耗大量资源的操作不会堵塞主线程

3. worker和页面怎么通信?
A: 通过 onmessage/postMessage方法

4. shared worker是什么?
A: web worker默认只能被生成它的父级页面所调用，但是shared worker可以被多个页面共享使用。

5. Service worker是什么? 有什么用?
A: **基于web worker**，在基础上增加了**离线缓存能力**，可以充当一个网站和浏览器之间的代理服务器，可以拦截全部的请求并做出相应的动作；创建有效的离线体验；具有声明周期；可以访问cache和indexDB；支持推送。

共享内存

Q: 共享内存是什么?
A: 所谓共享内存就是使得**多个进程可以访问同一块内存空间**，是最快的可用IPC（进程间通信 Inter-Process Communication）形式。

是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，**来达到进程间的同步及互斥**。

所有进程**都能访问共享内存中的地址**。如果一个进程向这段共享内存写了数据，所做的改动会即时被有访问同一段共享内存的其他进程看到。

WASM

WebAssembly或称wasm是一个实验性的低级编程语言，应用于浏览器内的客户端。
它设计的目的不是为了手写代码而是为诸如C、C++和Rust等低级源语言提供一个高效的编译目标。

Q: 为什么会有wasm呢?
A:
- 前端项目业务越来越多，文件依赖关系越来越复杂，启动速度越来越慢等。
- 除了代码量大，还有js自己的历史缺陷，没有类型限定的脚本语言，运行的效率有很大瓶颈。

Q: 什么时候该用wasm呢?
A:
- 对性能有很高要求的App/Module/游戏
- 在Web中使用C/C++/Rust/Go的库

浏览器的渲染原理

```
关键过程如下：
- 解析HTML，生成DOM树(DOM)
- 解析CSS，生成CSSOM树
- 将DOM和CSSOM合并，生成渲染树（Rendere-Tree）
- 计算渲染树的布局Layout
- 将布局渲染到屏幕上Paint

Q：为什么要构建DOM树？
A：因为浏览器不能直接理解和使用HTML，需要将HTML转换为浏览器能够理解的结构，这就是DOM树。

Q：CSS加载会阻塞页面展示吗？
A：不会阻塞DOM树的解析，但是会阻塞DOM树的渲染，并且会阻塞后面的JS执行。

Q：什么是回流？什么时候触发回流？
A：通过构造渲染树，我们将可见DOM节点以及它对应的样式结合起来，可是我们还需要计算它们在设备视口(viewport)内的确切位置和大小，这个计算的阶段就是回流。

- 页面一开始渲染的时候（这肯定避免不了）
- 浏览器的窗口尺寸变化（因为回流是根据视口的大小来计算元素的位置和大小的）
- 添加或删除可见的DOM元素
- 元素的位置发生变化
- 元素的尺寸发生变化（包括外边距、内边框、边框大小、高度和宽度等）
- 内容发生变化，比如文本变化或图片被另一个不同尺寸的图片所替代。
- 元素字体大小变化
- 激活CSS伪类（例如：:hover）

Q：重绘和回流的关系？
A：回流一定重绘，重绘不一定回流。当修改元素的css属性不影响其位置和大小时，不会触发回流，只有重绘。
```

Cookie

问题：客户端数据存储在 cookie 合适吗

答案：不合适。因为 cookie 数据会被添加到每次 http 请求头部 `Cookie` 上，增加了请求体的大小，占用上行带宽（虽然不大）；一般后端框架会自动解析请求的 cookie 数据，影响后端服务的执行效率（虽然不大）。如果只是纯客户端数据，localStorage 或 sessionStorage 更加合适。

拓展

3. 根据 http 协议，服务端如何给客户端设置 cookie？答案：在响应头部中添加 `Set-Cookie` 头部，这个头部可以有多个。
4. cookie 有哪些属性？回答：domain path http-only max-age。domain path 指定域名的有效范围，http-only 禁止客户端访问或修改 cookie，max-age 设置 cookie 的有效期。
5. 客户端如何设置 cookie？如何删除 cookie？添加 `document.cookie = 'a=1'`，删除 `document.cookie='a=1; max-age=0'`。在设置 cookie 时不指定 max-age 或 expires，表明 cookie 的有效期为当次会话，浏览器关闭后，cookie 会被自动删除
6. 了解其他前端存储都有哪些

requestAnimationFrame

问题：requestAnimationFrame 有什么用？

答案：可以用 requestAnimationFrame 优化网页性能，提升用户体验。浏览器的页面渲染和 js 引擎共用一个线程，导致在执行 js 时，网页的渲染会停止。在 60Hz 刷新率情况下，浏览器保持每秒 60 次的刷新频率，也就是大约 16 毫秒渲染一次；但是如果某段 js 脚本逻辑比较复杂，执行时间超过了 16 毫秒，那么期间如果 dom 发生了改变，浏览器就无法立即重绘，视觉上就会产生页面卡顿的感觉。解决方案是，可以将一段执行时间较长的 js 逻辑拆解成若干个小逻辑，放置在 requestAnimationFrame 的回调中。那么浏览器会在帧刷新前执行对应的回调。

> 相比 setTimeout，requestAnimationFrame 还有一个额外的优势。当页面处于非当前 tab 时，requestAnimationFrame 的回调函数不会被执行。
>
> 候选人一般解释不了那么详细。也可以直接上题目。在网页中插入 10000 个 `<div>hello world</div>` 节点，使用 requestAnimationFrame 保证插入期间页面不卡顿

```
let c = 1e4

const run = () => {
  c -= 100;
  const frag = document.createDocumentFragment();
  for (let i = 0; i < 100; ++i) {
    const div = document.createElement('div');
```

```
      div.textContent = 'hello world';
      frag.appendChild(div);
    }
    document.body.appendChild(frag);
    if (c > 0) {
      requestAnimationFrame(run)
    }
  }

  run();
```

跨域是怎么出现的，有什么解决方法

由于浏览器同源策略，凡是发送请求url的协议、域名、端口三者之间任意一与当前页面地址不同即为跨域。

（1）JSONP 动态创建script标签

但缺点是只支持get请求，并且很难判断请求是否失败（一般通过判断请求是否超时）。

（2）Proxy代理

这种方式首先将请求发送给后台服务器，通过服务器来发送请求，然后将请求的结果传递给前端。

（3）CORS跨域

是现代浏览器提供的一种跨域请求资源的方法，需要客户端和服务器端的同时支持。整个CORS通信过程，都是浏览器自动完成，不需要用户参与。对于开发者来说，CORS通信与同源的AJAX通信没有差别，代码完全一样。浏览器一旦发现AJAX请求跨源，就会自动添加一些附加的头信息，有时还会多出一次附加的请求，但用户不会有感觉。因此，实现CORS通信的关键是服务器。只要服务器实现了CORS接口，就可以跨源通信

服务响应头返回，Access-Control-Allow-Origin: *

追问：

跨域时如何带上cookie

答案：

通过设置 `withCredentials: true` ，发送Ajax时，Request header中便会带上 Cookie 信息

服务器端 `Access-Control-Allow-Credentials = true` 时，参数 `Access-Control-Allow-Origin` 的值不能为 `'*'`

浏览器的缓存策略

追问请求头

强缓存：expires、cache-control

public：所有内容都将被缓存（客户端和代理服务器都可缓存）

private：所有内容只有客户端可以缓存，Cache-Control的默认取值

no-cache：客户端缓存内容，但是是否使用缓存则需要经过协商缓存来验证决定

no-store：所有内容都不会被缓存，即不使用强制缓存，也不使用协商缓存

max-age=xxx (xxx is numeric)：缓存内容将在xxx秒后失效

协商缓存：last-modified、etag（区别）

HTTP

HTTP 协议

问题：POST 和 PUT 方法在语义上有什么区别?

答案：在 HTTP 语义上，POST 方法指创建资源，PUT 方法指更新资源。多次 POST 相同的数据，服务器需要创建多个新资源；多次 PUT 相同的数据，服务器只会更新同一个资源。简单说，PUT 请求应当和 GET 请求一样，具有幂等性。

评分

11. 如果候选人对这个问题没有任何概念，可以退一步询问前端常用的 HTTP 请求方法有哪些？如果能回答出基本的 GET POST PUT DELETE，可以给 2.5 分
12. 回答正确，3 分
13. 根据拓展问题的回答情况，酌情给到 3.5 分

拓展

14. 一个 HTTP 请求由哪几个部分组成?

- 第一行：请求方法 + 空格 + 请求路径 + 空格 + 字符串 HTTP + 字符串 / + http 版本号 + 回车换行
- 第二行：头部，头部名称 + 冒号 + 值 + 回车换行（可以顺便问问 HTTP 的头部是大小写敏感的吗？答案：不敏感。Content-Type content-type content-Type 都是一样的）
- 随后是可选的多个头部
- 头部结束后添加一个空行
- 最后加上请求体

15. GET 或 DELETE 请求可以有请求体吗？可以有，但一般前端业务中发起的 GET 请求没有请求体。
16. 前端业务中常用的请求头部 `Content-Type` 的值有哪些？分别用在哪些业务场景中？

- application/json 用于提交 JSON 格式的数据，常见在各种异步请求中
- application/x-www-form-urlencoded 用于提交表单数据，html form 表单的默认提交格式（这个 ContentType 太长，候选人不能完全记下来很正常，不需要苛求，但至少要知道有这个格式存在）
- multipart/form-data 用于提交二进制文件。
- 可以顺便问问，使用 JSON 格式和使用 form-urlencoded 格式提交数据的差异在哪里？答案：JSON 格式可以很方便地表示数组、布尔值、以及嵌套对象这种数据结构。以数组为例，form-urlencoded 格式按照不同的服务端实现，可能有多种情况，比如 `a[]=1&a[]=2&a[]=3`，或 `a[0]=1&a[1]=2&a[2]=3`，或 `a=1,2,3&b=4,5,6` 等。使用这种格式会增加前后端沟通数据结构的隐性成本。

用户登录功能

问题：在你参与的项目中，用户登录登出功能是如何实现的？

> 问这个问题的目的是：观察候选人日常工作中是否会做主动的技术探索，一般登录登出功能都是后端实现好，前端调个接口。但实际上，登录登出功能的设计和实现是一个需要跨前后端的业务；理解整个登录登出的实现细节有助于实现更加稳健安全的网页应用。

答案：

常见的登录登出设计方案有下面两种

17. 传统表单提交方式。后端接收到请求后先根据提交上来的信息做鉴权。成功后，响应一个 302 重定向，同时添加一个响应头部 `Set-Cookie`，将该次登录状态写入客户端 cookie。
18. 单页应用的 token 方案。客户端携带鉴权信息，向某个公开 API 请求 token。得到 token 后，客户端存储在 localStorage 或 sessionStorage 中。之后的异步请求，将这个 token 设置在一个约定好的请求头部中。服务端在执行业务逻辑前，会统一校验 token 是否合法。

评分

19. 回答请求后端接口就可以了，无法说出任何技术细节的，2 分
20. 至少提到 token 或 cookie 两种方案一种的，或提出其他合理方案的，2.5 分
21. 能准确描述任意一种方案的 3 分
22. 根据拓展问题的回答情况，酌情给到 3.5 分

拓展

23. 如果候选人提到的 token 方案。可以追问，现在常用的 token 生成方案有哪些？比如 jwt 之类的。使用 token 的缺点有哪些？答案：以 jwt 为例，token 一旦签发是无法撤回的，在时效内如果 token 泄露了，服务端有可能收到恶意请求。如何处理 token 过期问题？答案：在约定的过期时间之前，向服务器交换一个新的 token；并且在网页应用中维护一个统一的逻辑，比如当有异步请求响应类似 401 之类的状态码时，网页自动重定向到登录页之类的。
24. 如果候选人提到 cookie 方案。可以追问。服务端在设置客户端 cookie 时，添加 http-only 属性，能否避免 CSRF 攻击？回答：不能。http-only 只能防止浏览器脚本读取这个 cookie 值。CSRF 攻击并不需要获取 cookie 信息。采用 token 方案才能避免 CSRF。

应用层

vue双向数据绑定原理？（二选一）

vue是通过基于发布订阅模式的数据劫持来实现双向数据绑定的

- **Observer 观察者函数**：监听所有数据的变化，当数据变动时获取最新的值并通知给订阅者（数据劫持）
- **Watcher 订阅者函数**：当接受到观察者的通知和提供的数据后同步更新视图
- **Compile 解析器函数**：解析 DOM 元素上的 `v-model` 指令和 `{{}}` 语法

vue.2 是基于Object.defineProperty，vue.3是基于Proxy

>> 进一步问：不同劫持方式的优缺点

1. Object.defineProperty的缺点
    - 不能监听数组：因为数组没有getter和setter，因为数组长度不确定，如果太长性能负担太大
    - 只能监听属性，而不是整个对象，需要遍历循环属性
    - 只能监听属性变化，不能监听属性的删减
2. proxy的好处
    - 可以监听数组
    - 监听整个对象不是属性

- 13种拦截方法，强大很多
- 返回新对象而不是直接修改原对象，更符合immutable；

3. proxy的缺点
   - 兼容性不好，而且无法用polyfill磨平

>> 进一步问： 用Object.defineProperty 实现在对象上定义一个新属性

```
const obj = { name: 'xxx' };

// 添加具有数据描述符的属性，相当于 obj.age = 18
Object.defineProperty(obj, 'age', {
  configurable: true, // 可删除
  enumerable: true, // 可枚举
  value: 18,
  writable: true // 可重写
})

// 添加具有存取描述符的属性
let age = 18;
Object.defineProperty(obj, 'age', {
  configurable: true,
  enumerable: true,
  get(){ return age },
  set(newVal){ age = newVal }
})
```

评分标准：

一、至少说出数据拦截和发布订阅模式。2.5 分

二、能说出优缺点并完成Object.defineProperty 在一个对象上定义一个新属性 3分

React SSR 背后发生了什么

> 参考答案：
>
> node server 接收客户端请求，得到当前的请求路径，服务端和客户端使用同一套路由规则，在路由配置表根据 path 匹配具体的组件，请求对应的数据，将数据通过 props/context/store 形式传入组件，利用 react 提供的服务端渲染api renderToString/renderToNodeStream 将组件渲染为生成带有标记的 html 字符串或者数据流，在输出最终的html之前，将数据注入（注水），服务端输出后，

> 客户端就能得到组件数据（脱水），客户端利用 ReactDOM.hydrate 渲染，根据服务端携带的标记更新 React 组件树，并附加事件响应。

介绍下常用的 react hooks（非 react 官方也算）

- useState()，状态钩子。为函数组建提供内部状态
- useContext()，共享钩子。该钩子的作用是，在组件之间共享状态。 可以解决react逐层通过Porps传递数据，它接受React.createContext()的返回结果作为参数，使用useContext将不再需要Provider 和 Consumer。
- useReducer()，Action 钩子。useReducer() 提供了状态管理，其基本原理是通过用户在页面中发起action, 从而通过reducer方法来改变state, 从而实现页面和状态的通信。使用很像redux
- useEffect()，副作用钩子。它接收两个参数， 第一个是进行的异步操作， 第二个是数组，用来给出Effect的依赖项
- useRef()，获取组件的实例；渲染周期之间共享数据的存储(state不能存储跨渲染周期的数据，因为state的保存会触发组件重渲染）
- useRef传入一个参数initValue，并创建一个对象{ current: initValue }给函数组件使用，在整个生命周期中该对象保持不变。
- useMemo和useCallback：可缓存函数的引用或值，useMemo用在计算值的缓存，注意不用滥用。经常用在下面两种场景（要保持引用相等；对于组件内部用到的 object、array、函数等，如果用在了其他 Hook 的依赖数组中，或者作为 props 传递给了下游组件，应该使用 useMemo/useCallback）
- useLayoutEffect：会在所有的 DOM 变更之后同步调用 effect，可以使用它来读取 DOM 布局并同步触发重渲染

评分标准：

答出 2 个以上给 2.5 分

答出 4 个以上给 3 分

答出 6 个以上，给 3.5 分

做过什么前端优化

操作系统相关

I/O模型

Q：一次I/O操作可以分成哪几个阶段？
A：两个阶段，**等待资源**和**使用资源**。

Q：简单讲讲「阻塞与非阻塞I/O」？
A：相对于操作系统内核而言，**阻塞发生在等待资源阶段**，根据发起I/O请求是否阻塞来判断。
- 阻塞I/O：这种模式下一个用户进程在发起一个 I/O 操作之后，只有收到响应或者超时才可进行处理其它事情，否则 I/O 将会一直阻塞。以读取磁盘上的一段文件为例，系统内核在完成磁盘寻道、读取数据、复制数据到内存中之后，这个调用才算完成。阻塞的这段时间对 CPU 资源是浪费的。
- 非阻塞I/O：这种模式下一个用户进程发起一个 I/O 操作之后，如果数据没有就绪，会立刻返回（标志数据资源不可用），此时 CPU 时间片可以用来做一些其它事情。

Q：简单讲讲「同步与异步I/O」？
A：发生在**使用资源阶段**，根据实际I/O操作来判断。
- 同步I/O：应用发送或接收数据后，若不返回，则继续等待（发生阻塞），直到数据成功或失败返回。
- 异步I/O：应用发送或接收数据后立刻返回，数据写入OS缓存，由OS完成数据发送或接收，并返回成功或失败的信息给应用。Node.js就是典型的异步编程例子。

系统内存

Q：内存溢出（OOM）是什么？
A：系统已经不能再分配出你所需要的空间。
内存泄漏的堆积是内存溢出的一种原因。

Q：内存泄漏是什么？
A：是指程序中已动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

Q：JS会有内存泄漏的问题吗？
A：**JS的内存回收机制采用的是「引用计数」**，垃圾回收器会定期扫描内存，当某个内存中的值被引用为零时就会将其回收。当前变量已经使用完毕但依然被引用，导致垃圾回收器无法回收这就造成了内存泄漏。
容易造成内存泄漏的用法是：闭包。

Q：内存溢出的原因和解决办法？
A：
原因：
1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据；
2.集合类中有对对象的引用，使用完后未清空，使得GC不能回收；
3.代码中存在死循环或循环产生过多重复的对象实体；
4.使用的第三方软件中的BUG；

5.启动参数内存值设定的过小。
解决办法：
1．增加内存；
2．检查错误日志，查看“OutOfMemory”错误前是否有其它异常或错误；
3．对代码进行走查和分析，找出可能发生内存溢出的位置；
4．使用内存查看工具动态查看内存使用情况。

并发和并行有什么区别?

并发就是在一段时间内，多个任务都会被处理；但在某一时刻，只有一个任务在执行。

并行就是在同一时刻，有多个任务在执行。这个需要多核处理器才能完成，在微观上就能同时执行多条指令，不同的程序被放到不同的处理器上运行，这个是物理上的多个进程同时进行。

什么是死锁？死锁产生的条件?

在两个或者多个并发进程中，如果每个进程持有某种资源而又等待其它进程释放它或它们现在保持着的资源，在未改变这种状态之前都不能向前推进，称这一组进程产生了死锁。通俗的讲就是两个或多个进程无限期的阻塞、相互等待的一种状态。

死锁产生的四个必要条件：（有一个条件不成立，则不会产生死锁）
- 互斥条件：一个资源一次只能被一个进程使用
- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得资源保持不放
- 不剥夺条件：进程获得的资源，在未完全使用完之前，不能强行剥夺
- 循环等待条件：若干进程之间形成一种头尾相接的环形等待资源关系

如何处理死锁问题：
- 检测死锁并且恢复。
- 仔细地对资源进行动态分配，以避免死锁。
- 通过破除死锁四个必要条件之一，来防止死锁产生。

进程调度策略?

- 先来先服务：非抢占式的调度算法，按照请求的顺序进行调度。
- 短作业优先：非抢占式的调度算法，按估计运行时间最短的顺序进行调度。
- 最短剩余时间优先：最短作业优先的抢占式版本，按剩余运行时间的顺序进行调度。 当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。如果新的进程需要的时间更少，则挂起当前进程，运行新的进程。否则新的进程等待。
- 时间片轮转：将所有就绪进程按 FCFS 的原则排成一个队列，每次调度时，把 CPU 时间分配给队首进程，该进程可以执行一个时间片。当时间片用完时，由计时器发出时钟中断，调度程序便停止该

进程的执行，并将它送往就绪队列的末尾，同时继续把 CPU 时间分配给队首的进程。
- 优先级调度：为每个进程分配一个优先级，按优先级进行调度。

React

简要介绍Virtual DOM及工作原理?

Virtual DOM 是一个轻量级的 JavaScript 对象，它最初只是 real DOM 的副本。它是一个节点树，它将元素、它们的属性和内容作为对象及其属性。 React 的渲染函数从 React 组件中创建一个节点树。然后它响应数据模型中的变化来更新该树，该变化是由用户或系统完成的各种动作引起的。

Virtual DOM 工作过程有三个简单的步骤。
一、每当底层数据发生改变时，整个 UI 都将在 Virtual DOM 描述中重新渲染。
二、然后计算之前 DOM 表示与新表示的之间的差异。
三、完成计算后，将只用实际更改的内容更新 real DOM。

React 合成事件是什么？请举一个合成事件?

合成事件是围绕浏览器原生事件充当跨浏览器包装器的对象。它们将不同浏览器的行为合并为一个 API。这样做是为了确保事件在不同浏览器中显示一致的属性。

复合事件:
onCompositionEnd onCompositionStart onCompositionUpdate
键盘事件:
onKeyDown onKeyPress onKeyUp
表单事件:
onChange onInput onInvalid onReset onSubmit
（其他见https://zh-hans.reactjs.org/docs/events.html）

React类式组件的生命周期?

componentWillMount() - 在渲染之前执行，在客户端和服务器端都会执行。
componentDidMount() - 仅在第一次渲染后在客户端执行。
componentWillReceiveProps() - 当从父类接收到 props 并且在调用另一个渲染器之前调用。
shouldComponentUpdate() - 根据特定条件返回 true 或 false。如果你希望更新组件，请返回 true 否则返回 false。默认情况下，它返回 false。
componentWillUpdate() - 在 DOM 中进行渲染之前调用。

componentDidUpdate() - 在渲染发生后立即调用。
componentWillUnmount() - 从 DOM 卸载组件后调用。用于清理内存空间。

React常用的hooks？什么时候用useReducer？

useState、useEffect、useLayoutEffect、useRef、
useContext、useCallback、useMemo、useReducer、
useImperativeHandle、useDebugValue

当一个state有比较复杂的数据结构，并且有很多子项想单独修改，就推荐使用useReducer
useReducer可以像Redux一样使用action来更新state。

简述React-Fiber解决了什么问题？原理？

React V15 在渲染时，会递归比对 VirtualDOM 树，找出需要变动的节点，然后同步更新它们，一气呵成。这个过程期间， React 会占据浏览器资源，这会导致用户触发的事件得不到响应，并且会导致掉帧，导致用户感觉到卡顿。

所以 React 通过Fiber 架构，让这个执行过程变成可被中断。
“适时”地让出 CPU 执行权，除了可以让浏览器及时地响应用户的交互，还有其他好处：
一、分批延时对DOM进行操作，避免一次性操作大量 DOM 节点，可以得到更好的用户体验；
二、给浏览器一点喘息的机会，它会对代码进行编译优化（JIT）及进行热代码优化，或者对 reflow 进行修正。

Fiber也称协程或者纤程。它和线程并不一样，协程本身是没有并发或者并行能力的（需要配合线程），它只是一种控制流程的让出机制。让出 CPU 的执行权，让 CPU 能在这段时间执行其他的操作。渲染的过程可以被中断，可以将控制权交回浏览器，让位给高优先级的任务，浏览器空闲后再恢复渲染。

React如何减少不必要的re-Render

一、shouldComponentUpdate 和 PureComponent
二、函数式组件没有shouldComponentUpdate，所以可以封装一个高阶组件来判断是否要render
三、React.memo
四、immutable

forwardRef有什么作用？该什么时候使用？

React.forwardRef 会创建一个React组件，这个组件能够将其接受的 ref 属性转发到其组件树下的另一个组件中。这种技术并不常见，但在以下两种场景中特别有用：

一、转发 refs 到 DOM 组件
二、在高阶组件中转发 refs

网络

队头阻塞

Q：什么是队头阻塞？
A：队头阻塞分以下两种：
- TCP队头阻塞：TCP数据包是有序传输，中间一个数据包丢失，会等待该数据包重传，造成后面的数据包的阻塞。
- HTTP队头阻塞：HTTP遵守“请求-响应”的模式，也就是客户端每次发送一个请求到服务端，服务端返回响应。但有一个致命缺陷那就是页面中有多个请求，**每个请求必须等到前一个请求响应之后才能发送**，并且当前请求的响应返回之后，当前请求的下一个请求才能发送。**此时如果有一个请求响应慢了，会造成后面的响应都延迟。**
  为了提高速度和效率，在持久连接的基础上，HTTP1.1进一步地支持在持久连接上使用管道化（pipelining）特性。**管道化允许客户端在已发送的请求收到服务端的响应之前发送下一个请求**，借此来减少等待时间提高吞吐，如果多个请求能在同一个TCP分节发送的话，还能提高网络利用率。
  同一个tcp连接中可以同时发送多个http请求，也就是并发，**但是在响应的时候，必须排队响应，谁先到达的谁先响应**，相比不支持管道化的http请求确实提高了效率，但是还是有局限性，**加入其中某个响应因为某种原因延迟了几秒，后面的响应都会被阻塞。**

HTTP/2与HTTP/3

Q：HTTP/1.x 的缺陷？
A：
- 连接无法复用。HTTP/1.0 传输数据时，每次都需要重新建立连接，增加延迟。HTTP/1.1 虽然加入 keep-alive 可以复用一部分连接，但域名分片等情况下仍然需要建立多个 connection，耗费资源，给服务器带来性能压力。
- 队头阻塞。一系列package请求都因为第一个包被阻塞。
- 协议开销大。header携带内容过大。
- 安全因素。传输都是明文。

Q：HTTP/2的新特性？
A：
- 二进制传输。将请求和响应数据分割成更小的帧，并且采用二进制编码。
- 多路复用。很好的解决了浏览器限制同一个域名下请求数量的问题。
- Header压缩。对于与上个请求相同的header，不再发送，只发送差异的键值对

- Server Push。服务侧可以将客户端可能需要的资源，主动推送到客户端，不需要等客户端解析HTML时再发送这些请求。

Q: HTTP/3简介?

A: HTTP/1.x和HTTP/2都是基于TCP协议的，**HTTP/3直接放弃了TCP，Google搞了个基于UDP协议的QUIC（Quick UDP Internet Connections）协议**，使用在了HTTP/3上。
将修改协议使用UDP，**主要还是为了解决队头阻塞问题**。

CSRF是什么？怎么解决?

描述
CSRF，跨站请求伪造（英文全称是Cross-site request forgery），是一种挟制用户在当前已登录的Web应用程序上执行非本意的操作的攻击方法。

怎么解决?
检查Referer字段。
添加校验token。

TCP和UDP的区别?

TCP：收发数据前必须和对方建立可靠的连接，建立连接的3次握手、断开连接的4次挥手，为数据传输打下可靠基础。
面向连接，是可靠服务，传输比较慢，报文格式面向字节流，常见场景网页、邮件。

UDP：是一个面向无连接的协议，数据传输前，源端和终端不建立连接，发送端尽可能快的将数据扔到网络上，接收端从消息队列中读取消息段。
面向无连接，可靠性不保证，传输快，保温格式面向报文，常见场景语音广播。

WebSocket连接过程？和Socket有什么区别?

WebSocket是一个持久化协议，WebSocket是基于HTTP协议的，借用HTTP协议来完成一部分握手。
建连过程：
1．发送一个GET请求，标记自己是特殊的http请求
关键：**Upgrade：websocket**，Connection：Upgrade
2．服务端返回，**状态码101**:Switching Protocol， 这样就连接成功了
3．接下来就是通过Websocket来通讯

WebSocket和Socket的区别?
- WebSocket是一个持久化的协议，它是伴随H5而出的协议，用来解决http不支持持久化连接的问题。

```
- Socket一个是网编编程的标准接口，而WebSocket则是应用层通信协议。
```

从输入URL到打开页面经历了什么?

```
输入地址
浏览器查找域名的 IP 地址
浏览器向 web 服务器发送一个 HTTP 请求
服务器的永久重定向响应
服务器处理请求
服务器返回一个 HTTP 响应200，并包含HTML内容
浏览器显示 HTML
浏览器发送请求获取嵌入在 HTML 中的资源（如图片、音频、视频、CSS、JS等等）
```

重定向 301 vs. 302

```
301（永久性转移）
请求的网页已被永久移动到新位置。服务器返回此响应时，会自动将请求者转到新位置。

302（暂时性转移）
服务器目前正从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。此代码与响应GET和HEAD请求的301代码类似，会自动将请求者转到不同的位置。
```

对称加密？和非对称加密的区别?

```
对称加密：指加密和解密使用同一密钥，优点是运算速度较快，缺点是如何安全将密钥传输给另一方。常见的对称加密算法有：DES、AES等。

非对称加密：指的是加密和解密使用不同的密钥（即公钥和私钥）。公钥与私钥是成对存在的，如果用公钥对数据进行加密，只有对应的私钥才能解密。常见的非对称加密算法有RSA。
```

Coding能力

实现Promise.all

```
Promise.prototype.all = function(promises) {
  let results = [];
  let promiseCount = 0;
  let promisesLength = promises.length;
```

```
  return new Promise(function(resolve, reject) {
    for (let val of promises) {
      Promise.resolve(val).then(function(res) {
        promiseCount++;
        // results.push(res);
        results[i] = res;
        // 当所有函数都正确执行了，resolve输出所有返回结果。
        if (promiseCount === promisesLength) {
          return resolve(results);
        }
      }, function(err) {
        return reject(err);
      });
    }
  });
};
```

实现Array.prototype.flat

```
// 对ES6比较熟悉的会写下面这种
const flat = arr => {
  return arr.reduce((pre, cur) => {
    return pre.concat(Array.isArray(cur) ? flat(cur) : cur);
  }, []);
};
```

字符串相乘

```
输入两个string型数字，计算他们的乘积，结果也用string型
输入: num1 = "123", num2 = "456"
输出: "56088"

var multiply = function(num1, num2) {
        if (isNaN(num1) || isNaN(num2)) return '';
    if (num1 === '0' || num2 === '0') return '0';

    let l1 = num1.length,
        l2 = num2.length;

    let result = [];

    for (let i = l1 -1; i >= 0; i--) {
        for (let j = l2 - 1; j >= 0; j--) {
```

```
            let index1 = i + j;
            let index2 = i + j + 1;

            let product = num1[i] * num2[j] + (result[index2] || 0);
            result[index2] = product % 10;
            result[index1] = Math.floor(product / 10) + (result[index1] || 0);
        }
    }
    return result.join('').replace(/^0+/, '');
};
```

字符串旋转

- 加分点：边界处理、运行安全、要求旋转次数应该是个无限长的列表而不是一个数

```
给定两个字符串，设定一次“旋转”操作是最左边的字符移动到最右边。
example: 'abcde'，在移动一次之后结果就是'bcdea'
请实现一个方法`func(A:string, B:string): number`，获取到A到B的旋转次数
```

手动实现正则

```
给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式完全匹配。

'.' 匹配任意单个字符
'*' 匹配零个或多个前面的那一个元素

eg1:
输入: s = "aab" p = "c*a*b"
输出: true

eg2:
输入: s = "aab" p = "c*a*bc"
输出: false


const isMatch = (s, p) => {
  if (s == null || p == null) return false;

  const sLen = s.length, pLen = p.length;

  const dp = new Array(sLen + 1);
  for (let i = 0; i < dp.length; i++) {
    dp[i] = new Array(pLen + 1).fill(false); // 将项默认为false
```

```
  }
  // base case
  dp[0][0] = true;
  for (let j = 1; j < pLen + 1; j++) {
    if (p[j - 1] == "*") dp[0][j] = dp[0][j - 2];
  }
  // 迭代
  for (let i = 1; i < sLen + 1; i++) {
    for (let j = 1; j < pLen + 1; j++) {

      if (s[i - 1] == p[j - 1] || p[j - 1] == ".") {
        dp[i][j] = dp[i - 1][j - 1];
      } else if (p[j - 1] == "*") {
        if (s[i - 1] == p[j - 2] || p[j - 2] == ".") {
          dp[i][j] = dp[i][j - 2] || dp[i - 1][j - 2] || dp[i - 1][j];
        } else {
          dp[i][j] = dp[i][j - 2];
        }
      }
    }
  }
  return dp[sLen][pLen]; // 长sLen的s串 是否匹配 长pLen的p串
};
```

多媒体

推流和拉流?

推流:
指的是把采集阶段封包好的内容传输到服务器的过程，然后通过CDN分发。
常用的流传输协议有RTSP、RTMP、HLS等，使用RTMP传输的延时通常在1-3秒。

拉流:
拉流是指服务器已有直播内容，根据协议类型（如RTMP、RTP、RTSP、HTTP等），与服务器建立连接并接收数据，进行拉取的过程。拉流端的核心处理在播放器端的解码和渲染。

YUV是什么？与RGB的关系?

YUV:
一种颜色编码方法。【Y】表示明亮度，也就是灰阶值，【U】表示色度，【V】表示明度。

人眼的视觉特点是对亮度更敏感，对位置、色彩相对来说不敏感，所以可以充分压缩UV分量，来减少带宽。

关系?
YUV和RGB可以互相转换。
YUV相比RGB的优点是和黑白兼容，容易压缩带宽。

GOP的组成？每个帧的含义？怎么还原成图像？

GOP (Group of Pictures)是一组连续的画面，由一张I帧和数张B/P帧组成，是视频图像编码器和解码器存取的基本单位。

也就是说GOP组是指一个关键帧I帧所在的组的长度，每个GOP组只有1个I帧。

I 帧：自身可以通过视频解压算法解压成一张单独的完整的图片。
P 帧：需要参考其前面的一个I帧或者P帧来生成一张完整的图片。
B 帧：则要参考其前一个I帧或者P帧及其后面的一个P帧，来生成一张完整的图片。

常见的直播协议？他们的区别和优缺点？

1、**RTMP**：real time messaging protocol，实时传输协议，RTMP协议比较全能，既可以用来推送又可以用来直播，其核心理念是将大块的视频帧和音频帧“剁碎”，然后以小数据包的形式在互联网上进行传输，而且支持加密，因此隐私性相对比较理想，但拆包组包的过程比较复杂，所以在海量并发时也容易出现一些不可预期的稳定性问题。

2、**FLV**：FLV协议由Adobe公司主推，格式极其简单，只是在大块的视频帧和音视频头部加入一些标记头信息，由于这种极致的简洁，在延迟表现和大规模并发方面都很成熟。唯一的不足就是在手机浏览器上的支持非常有限，但是用作手机端APP直播协议却异常合适。

3、**HLS**：苹果原生：HTTP Live Streaming，遵循的是 HTTP 超文本传输协议，端口号8080，将视频分成5-10秒的视频小分片，然后用m3u8索引表进行管理，由于客户端下载到的视频都是5-10秒的完整数据，故视频的流畅性很好，但也同样引入了很大的延迟（HLS的一般延迟在10-30s左右）。

dts、pts是什么？

DTS（Decoding Time Stamp）：
    即解码时间戳，这个时间戳的意义在于告诉播放器该在什么时候解码这一帧的数据。

```
PTS（Presentation Time Stamp）：
    即显示时间戳，这个时间戳用来告诉播放器该在什么时候显示这一帧的数据。

这2个概念经常出现在音频视频编码和播放中，其实际意义是，PTS是真正录制和播放的时间戳，而DTS是解码的时间戳。

对于普通的无B帧视频(H264 Baseline或者VP8)，PTS/DTS应该是相等的，因为没有延迟编码。

对于有B帧的视频，I帧的PTS依然等于DTS，P帧的PTS>DTS，B帧的PTS<DTS。
```

常见的音视频格式？编码格式？封装格式？

```
视频编码格式：H264/AVC，MPEG2，MPEG4，WMV，DivX/XviD等
音频编码格式：AAC，MP3，APE，FLAC等
常见封装格式：MP4，MOV，TS，FLV，MKV等

编码是为了将原始很大的音频或视频按照一定的算法解压缩，方便存储和传播。

封装格式也称多媒体容器，它只是为多媒体编码提供了一个“外壳”，也就是将所有的编码后的视频、音频或字幕都包装到一个文件容器内呈现给观众，这个包装的过程就叫封装。
```

软解和硬解是什么？

```
硬解码：是将原来全部交由CPU来处理的视频数据的一部分交由GPU来做，而GPU的并行运算能力要远远高于CPU，这样可以大大的降低对CPU的负载，CPU的占用率较低了之后就可以同时运行一些其他的程序了。

软解码：即通过软件让CPU来对视频进行解码处理。
```

简述1080p的表达含义？1080p和1080i有什么区别？逐行和隔行的优劣？

```
分辨率：1920*1080
1080表示：垂直方向有1080条水平扫描线
p：逐行扫描 Progressive
i：隔行扫描 Interlaced

隔行扫描：每一帧被分割为两场，每一场包含了一帧中所有的奇数扫描行或者偶数扫描行。
通常是先扫描奇数行得到第一场，然后扫描偶数行得到第二场。

逐行扫描：每帧的所有像素同时显示。
```

```
10
11   优劣：逐行扫描闪烁少，对人眼较好
```