HTML

1. HTML5 有哪些新特性?

（1）HTML4 规定了三种声明方式，分别是：严格模式、过渡模式 和 框架集模式；而HTML5因为不是SGML的子集，只需要<!DOCTYPE>就可以了：

（2）语义化更好的内容标签。header/footer/article等

（3）音频、视频 API(audio,video)

（4）表单控件：HTML5 拥有多个新的表单输入类型。这些新特性提供了更好的输入控制和验证。

- color
- date
- datetime
- datetime-local
- email
- month
- number
- range
- search
- tel
- time
- url
- week

（5）5个API-本地存储，长期存储数据的 localStorage，比较常用，临时存储的 sessionStorage，浏览器关闭后自动删除，实际工作中使用的场景不多。

- **画布/Canvas**，canvas，figure,figcaption.
- **地理/Geolocation**.地理位置 API 允许用户向 Web 应用程序提供他们的位置。出于隐私考虑，报告地理位置前会先请求用户许可。
- **拖拽释放**.HTML拖拽释放 (Drag and drop) 接口使应用程序能够在浏览器中使用拖放功能。例如，通过这些功能，用户可以使用鼠标选择可拖动元素，将元素拖动到可放置元素，并通过释放鼠标按钮来放置这些元素。可拖动元素的一个半透明表示在拖动操作期间跟随鼠标指针。
- **Web Workers.**webworker, websocket, Geolocation,当在 HTML 页面中执行脚本时，页面的状态是不可响应的，直到脚本已完成。web worker 是运行在后台的 JavaScript，独立于其他脚本，不会影响页面的性能。您可以继续做任何愿意做的事情：点击、选取内容等等，而此时 web worker 在后台运行。

2. Doctype作用? 严格模式与混杂模式如何区分？它们有何意义?

- 页面被加载的时， link 会同时被加载，而 @imort 页面被加载的时， link 会同时被加载，而 @import 引用的 CSS 会等到页面被加载完再加载 import 只在 IE5 以上才能识别，而 link 是 XHTML 标签，无兼容问题 link 方式的样式的权重 高于 @import 的权重
- `<!DOCTYPE>` 声明位于文档中的最前面，处于 `<html>` 标签之前。告知浏览器的解析器， 用什么文档类型 规范来解析这个文档
  严格模式的排版和 JS 运作模式是 以该浏览器支持的最高标准运行
  在混杂模式中，页面以宽松的向后兼容的方式显示。模拟老式浏览器的行为以防止站点无法工作。
  DOCTYPE 不存在或格式不正确会导致文档以混杂模式呈现

3. 如何实现浏览器内多个标签页之间的通信?

调用 localstorge、cookies 等本地存储方式;

- 在 B 页面中可以使用 window.opener 获得 A 页面的 window 句柄，使用该句柄即可调用 A 页面中的对象，函数等。
  例如 A 页面定义 js 函数 onClosePageB，在 B 页面可以用 window.opener.onClosePageB 来进行回调。
- 使用 window.showModalDialog(sURL [, vArguments] [,sFeatures])打开新窗口。 其中 vArguments 参数可以用来向对话框传递参数。传递的参数类型不限，包括数组、函数等。对话框通过window.dialogArguments来取得传递进来的参数。
- 如果是支持 HTML5 的话，建议用本地存储 (local storage)，它支持一个事件方法 window.onstorage，只要其中一个窗口修改了本地存储，其他同源窗口会触发这个事件。

总结:

- WebSocket、SharedWorker;
- 也可以调用 localstorge、cookies 等本地存储方式;
- localstorge 另一个浏览上下文里被添加、修改或删除时，它都会触发一个事件;
- 我们通过监听事件，控制它的值来进行页面信息通信;
- 注意 quirks：Safari 在无痕模式下设置 localstorge 值时会抛出 QuotaExceededError 的异常;

方法一：调用 localstorge

标签1:

```
<input id="name">
<input type="button" id="btn" value="提交">
```

```
<script type="text/javascript">
    $(function(){
        $("#btn").click(function(){
           var name=$("#name").val();
            localStorage.setItem("name", name); //存储需要的信息
        });
   });
</script>
```

标签2:

```
$(function(){
    window.addEventListener("storage", function(event){
       console.log(event.key + "=" + event.newValue);
    });     //使用storage事件监听添加、修改、删除的动作
});
```

方法二：使用 cookie+setInterval

将要传递的信息存储在 cookie 中，每隔一定时间读取 cookie 信息，即可随时获取要传递的信息。

标签1：

```
$(function(){
    $("#btn").click(function(){
        var name=$("#name").val();
        document.cookie="name="+name;
    });
});
```

标签2：

```
$(function(){
    function getCookie(key) {
       return JSON.parse("{\"" +
document.cookie.replace(/;\s+/gim,"\",\"").replace(/=/gim, "\":\"") + "\"}")
[key];
    }
    setInterval(function(){
       console.log("name=" + getCookie("name"));
    }, 10000);
});
```

4. 行内元素有哪些？块级元素有哪些？ 空(void)元素有那些？行内元 素和块级元素有什么区别？

- 行内元素有： a b span img input select strong
- 块级元素有： div ul ol li dl dt dd h1 h2 h3 h4... p
- 空元素： br,hr img input link meta
- 行内元素不可以设置宽高，不独占一行
- 块级元素可以设置宽高，独占一行

5. 简述一下src与href的区别？

- src 用于替换当前元素，href用于在当前文档和引用资源之间确立联系。
- src 是 source 的缩写，指向外部资源的位置，指向的内容将会嵌入到文档中当前标签所在位置；在请求 src 资源时会将其指向的资源下载并应用到文档内，例如 js 脚本，img 图片和 frame 等元素
- `<script src ="js.js"></script>` 当浏览器解析到该元素时，会暂停其他资源的下载和处理，直到将该资源加载、编译、执行完毕，图片和框架等元素也如此，类似于将所指向资源嵌入当前标签内。这也是为什么将js脚本放在底
  部而不是头部
- href 是 Hypertext Reference 的缩写，指向网络资源所在位置，建立和当前元素（锚点）或当前文档（链接）之间的链接，如果我们在文档中添加
- `<link href="common.css" rel="stylesheet"/>` 那么浏览器会识别该文档为 css 文件，就会并行下载资源并且不会停止对当前文档的处理。这也是为什么建议使用 link 方式来加载 css ， 而不是使用 @import 方式

6. cookies,sessionStorage,localStorage 的区别?

- cookie 是网站为了标示用户身份而储存在用户本地终端（Client Side）上的数据（通常经过加密）。
- cookie 数据始终在同源的 http 请求中携带（即使不需要），记会在浏览器和服务器间来回传递。
- sessionStorage 和 localStorage 不会自动把数据发给服务器，仅在本地保存。

存储大小

- cookie 数据大小不能超过 4k。
- sessionStorage 和 localStorage 虽然也有存储大小的限制，但比 cookie 大得多，可以达到 5M 或更大。

有期时间

- localStorage 存储持久数据，浏览器关闭后数据不丢失除非主动删除数据；
- sessionStorage 数据在当前浏览器窗口关闭后自动删除。
- cookie 设置的 cookie 过期时间之前一直有效，即使窗口或浏览器关闭

7. HTML5 的离线储存的使用和原理?

相似存储

localStorage 长期存储数据，浏览器关闭后数据不丢失； sessionStorage 数据在浏览器关闭后自动删除。

离线的存储

两种方式

- HTML5 的离线存储.appcache文件【废弃】
- service-worker 的标准

HTML5 的离线存储.appcache文件【废弃】

在用户没有与因特网连接时，可以正常访问站点或应用，在用户与因特网连接时，更新用户机器上的缓存文件。

原理：HTML5 的离线存储是基于一个新建的。appcache 文件的缓存机制（不是存储技术），通过这个文件上的解析清单离线存储资源，这些资源就会像 cookie 一样被存储了下来。

之后当网络在处于离线状态下时，浏览器会通过被离线存储的数据进行页面展示。

如何使用

- 页面头部像下面一样加入一个 manifest 的属性
- 在 cache.manifest 文件的编写离线存储的资源

```
CACHE MANIFEST
#v0.11
CACHE:
js/app.js
css/style.css
NETWORK:
resourse/logo.png
FALLBACK:
/ /offline.html
```

- 在离线状态时，操作 window.applicationCache 进行需求实现。

`service-worker`

8. 怎样处理 移动端 1px 被 渲染成 2px 问题?

- meta 标签中的 viewport 属性 ， initial-scale 设置为 1
- rem 按照设计稿标准走，外加利用 transfrome 的 scale(0.5) 缩小一倍即可； 2 全局处理
- meta 标签中的 viewport 属性 ， initial-scale 设置为 0.5
- rem 按照设计稿标准走即可

解释

UI 设计师设计的时候，画的 1px（真实像素）实际上是 0.5px(css) 的线或者边框。但是他不这么认为，他认为他画的就是 1px 的线，因为他画的稿的尺寸本身就是屏幕尺寸的 2 倍。假设手机视网膜屏的宽度是 320x480 宽，但实际尺寸是 640x960 宽，设计师设计图的时候一定是按照 640x960 设计的。但是前端工程师写代码的时候，所有 css 都是按照 320x480 写的，写 1px(css)，浏览器自动变成 2px（真实像素）。

那么前端工程师为什么不能直接写 0.5px(css) 呢？因为在老版本的系统里写 0.5px(css) 的话，会被浏览器解读为 0px(css)，就没有边框了。所以只能写成 1px(css)，实际在屏幕上显示出来就是设计师画的 1px（真实像素）的 2 倍那么宽，所以设计师会觉得这个线太粗了，和他的设计稿不一样。在新版的系统里，已经开始逐渐支持 0.5px(css) 这种写法。所以如果设计师在大图上设计了一个 1px（真实像素）的线的话，前端工程师直接除以 2，写 0.5px(css) 就好了。

另外一种解释

事实就是它并没有变粗，就是 css 单位中的 1px，对于 dpr 为 2 的设备，它实际能显示的最小值是 0.5px。

设计师口中说的 1px 是针对设备物理像素的，换算成 css 像素就是 0.5px。

一句话总结，background:1px solid black 在任何屏幕上都是一样粗的，但是 retina 屏可以显示比这更细的边框，然后设计师就不乐意了，让你改。

9. 浏览器是如何渲染页面的?

解析 HTML 文件，创建 DOM 树

自上而下，遇到任何样式（link、style）与脚本（script）都会阻塞（外部样式不阻塞后续外部脚本的加载）。

解析 CSS

优先级：浏览器默认设置<用户设置<外部样式<内联样式<HTML中的style样式；

构建渲染树

将 CSS 与 DOM 合并，构建渲染树（Render Tree）

布局和绘制

布局和绘制，重绘（repaint）和重排（reflow）

10. iframe 的优缺点?

- iframe 会阻塞主页面的 Onload 事件;
- iframe 和主页面共享连接池，而浏览器对相同域的连接有限制，所以会影响页面的并行加载。
- 使用 iframe 之前需要考虑这两个缺点。如果需要使用 iframe，最好是通过 javascript 动态给 iframe 添加 src 属性值，这样可以可以绕开以上两个问题。

11. Canvas 和 SVG 图形的区别是什么?

Canvas 和 SVG 都可以在浏览器上绘制图形。

SVG Canvas 绘制后记忆，换句话说任何使用 SVG 绘制的形状都能被记忆和操作，浏览器可以再次显示 Canvas 则是绘制后忘记，一旦绘制完成你就不能访问像素和操作它 SVG 对于创建图形例如 CAD 软件是良好的，一旦东西绘制，用户就想去操作它 Canvas 则用于绘制和遗忘类似动漫和游戏的场画。

为了之后的操作，SVG 需要记录坐标，所以比较缓慢。

因为没有记住以后事情的任务，所以 Canvas 更快。

我们可以用绘制对象的相关事件处理我们不能使用绘制对象的相关事件处理，因为我们没有他们的参考分辨率独立分辨率依赖

- SVG 并不属于 html5 专有内容，在 html5 之前就有 SVG。
- SVG 文件的扩展名是".svg"。
- SVG 绘制的图像在图片质量不下降的情况下被放大。
- SVG 图像经常在网页中制作小图标和一些动态效果图。

12. meta 标签?

核心

提供给页面的一些元信息（名称 / 值对），有助于 SEO。

属性值

- name

  名称 / 值对中的名称。author、description、keywords、generator、revised、others。 把 content 属性关联到一个名称。

- http-equiv

  没有 name 时，会采用这个属性的值。content-type、expires、refresh、set-cookie。把 content 属性关联到 http 头部

- content

  名称 / 值对中的值， 可以是任何有效的字符串。 始终要和 name 属性或 http-equiv 属性一起使用

- scheme

  用于指定要用来翻译属性值的方案。

CSS 基础

1. 请你讲一讲 CSS 的权重和优先级

权重

- 从0开始，一个行内样式+1000，一个id选择器+100，一个属性选择器、class或者伪类+10，一个元素选择器，或者伪元素+1，通配符+0

优先级

- 权重相同，写在后面的覆盖前面的
- 使用 `!important` 达到最大优先级，都使用 `!important` 时，权重大的优先级高

2. 介绍 Flex 布局，flex 是什么属性的缩写：

- 弹性盒布局，CSS3 的新属性，用于方便布局，比如垂直居中
- flex属性是 `flex-grow`、`flex-shrink` 和 `flex-basis` 的简写

3. CSS 怎么画一个大小为父元素宽度一半的正方形?

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .outer {
        width: 400px;
        height: 600px;
        background: red;
      }

      .inner {
        width: 50%;
        padding-bottom: 50%;
        background: blue;
      }
    </style>
  </head>
  <body>
    <div class="outer">
      <div class="inner"></div>
    </div>
  </body>
</html>
```


4. CSS实现自适应正方形、等宽高比矩形

- 双重嵌套，外层 relative，内层 absolute
- padding 撑高
- 如果只是要相对于 body 而言的话，还可以使用 vw 和 vh
- 伪元素设置 `margin-top: 100%` 撑高

双重嵌套，外层 relative，内层 absolute

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .outer {
        padding-top: 50%;
        height: 0;
        background: #ccc;
        width: 50%;
        position: relative;
      }

      .inner {
        position: absolute;
        width: 100%;
        height: 100%;
        top: 0;
        left: 0;
        background: blue;
      }
    </style>
  </head>
  <body>
    <div class="outer">
      <div class="inner">hello</div>
    </div>
  </body>
</html>
```

padding 撑高画正方形

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .outer {
        width: 400px;
        height: 600px;
        background: blue;
      }

      .inner {
        width: 100%;
        height: 0;
        padding-bottom: 100%;
```

```
        background: red;
      }
    </style>
  </head>
  <body>
    <div class="outer">
      <div class="inner"></div>
    </div>
  </body>
</html>
```

相对于视口 VW VH

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .inner {
        width: 1vw;
        height: 1vw;
        background: blue;
      }
    </style>
  </head>
  <body>
    <div class="outer">
      <div class="inner"></div>
    </div>
  </body>
</html>
```

伪元素设置 margin-top

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .inner {
        width: 100px;
        overflow: hidden;
        background: blue;
      }

      .inner::after {
        content: "";
        margin-top: 100%;
        display: block;
```

```
      }
    </style>
  </head>
  <body>
    <div class="outer">
      <div class="inner"></div>
    </div>
  </body>
</html>
```

5. 实现两栏布局的方式

左 float，然后右 margin-left（右边自适应）

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      div {
        height: 500px;
      }

      .aside {
        width: 300px;
        float: left;
        background: yellow;
      }

      .main {
        background: aqua;
        margin-left: 300px;
      }
    </style>
  </head>
  <body>
    <div class="aside"></div>
    <div class="main"></div>
  </body>
</html>
```

右 float，margin-right

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      div {
```

```
        height: 500px;
      }

      .aside {
        width: 300px;
        float: right;
        background: yellow;
      }

      .main {
        background: aqua;
        margin-right: 300px;
      }
    </style>
  </head>
  <body>
    <div class="aside"></div>
    <div class="main"></div>
  </body>
</html>
```

BFC + float

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      div {
        height: 500px;
      }

      .aside {
        width: 300px;
        float: left;
        background: yellow;
      }

      .main {
        overflow: hidden;
        background: aqua;
      }
    </style>
  </head>
  <body>
    <div class="aside"></div>
    <div class="main"></div>
  </body>
</html>
```

float + 负 margin

```
<head>
  <style>
    .left {
      width: 100%;
      float: left;
      background: #f00;
      margin-right: -200px;
    }

    .right {
      float: left;
      width: 200px;
      background: #0f0;
    }
  </style>
</head>

<div class="left"><p>hello</p></div>
<div class="right"><p>world</p></div>
```

圣杯布局实现两栏布局

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      /* div {
        height: 500px;
      } */

      /* .box {
        overflow: hidden;
      } */

      /* .container {
        padding: 0 300px 0 200px;
        border: 1px solid black;
      } */

      html,
      body {
        height: 100%;
      }

      div {
        height: 100%;
      }

      .container {
        display: flex;
      }
```

```
      .content {
        flex: 1 1;
        order: 2;
        background: #f00;
      }

      .left {
        float: left;
        width: 100%;
        background: #0f0;
      }

      .right {
        float: left;
        width: 300px;
        margin-left: -300px;
        background: #00f;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="left">你好</div>
      <div class="right">我好</div>
    </div>
  </body>
</html>
```

flex 实现两栏布局

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      /* div {
        height: 500px;
      } */

      /* .box {
        overflow: hidden;
      } */

      /* .container {
        padding: 0 300px 0 200px;
        border: 1px solid black;
      } */

      html,
      body {
        height: 100%;
      }
```

```
      div {
        height: 100%;
      }

      .container {
        display: flex;
      }

      .content {
        flex: 1 1;
        order: 2;
        background: #f00;
      }

      .left {
        flex: 0 0 200px;
        background: #0f0;
      }

      .right {
        flex: 1 1;
        background: #00f;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="left">你好</div>
      <div class="right">我好</div>
    </div>
  </body>
</html>
```

position + margin

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      /* div {
        height: 500px;
      } */

      /* .box {
        overflow: hidden;
      } */

      /* .container {
        padding: 0 300px 0 200px;
        border: 1px solid black;
      } */
```

```
      html,
      body {
        height: 100%;
      }

      div {
        height: 100%;
      }

      .container {
        display: flex;
        position: relative;
      }

      .content {
        flex: 1 1;
        order: 2;
        background: #f00;
      }

      .left {
        position: absolute;
        width: 300px;
        background: #0f0;
      }

      .right {
        width: 100%;
        margin-left: 300px;
        background: #00f;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="left">你好</div>
      <div class="right">我好</div>
    </div>
  </body>
</html>
```


6. 实现三列布局的方式

position + margin-left + margin-right

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      div {
        height: 500px;
      }
```

```
      .box {
        position: relative;
      }

      .left {
        position: absolute;
        left: 0;
        top: 0;
        width: 200px;
        background: green;
      }

      .right {
        position: absolute;
        right: 0;
        top: 0;
        width: 200px;
        background: red;
      }

      .middle {
        margin-left: 200px;
        margin-right: 200px;
        background: black;
      }
    </style>
  </head>
  <body>
    <div class="box">
      <div class="left"></div>
      <div class="middle"></div>
      <div class="right"></div>
    </div>
  </body>
</html>
```


通过 float + margin

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      div {
        height: 500px;
      }

      .left {
        float: left;
        width: 200px;
        height: 200px;
        background: green;
```

```
      }

      .right {
        float: right;
        width: 200px;
        height: 200px;
        background: red;
      }

      .middle {
        margin-left: 210px;
        margin-right: 210px;
        background: black;
        height: 200px;
      }
    </style>
  </head>
  <body>
    <div class="box">
      <div class="left"></div>
      <div class="right"></div>
      <div class="middle"></div>
    </div>
  </body>
</html>
```

圣杯布局

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .container {
        padding: 0 300px 0 200px;
        border: 1px solid black;
      }

      .content {
        float: left;
        width: 100%;
        background: #f00;
      }

      .left {
        width: 200px;
        background: #0f0;
        float: left;
        margin-left: -100%;
        position: relative;
        left: -200px;
      }
```

```
      .right {
        width: 300px;
        background: #00f;
        float: left;
        margin-left: -300px;
        position: relative;
        right: -300px;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="content">中间内容</div>
      <div class="left">左侧区域</div>
      <div class="right">右侧区域</div>
    </div>
  </body>
</html>
```

双飞翼布局

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      html,
      body {
        height: 100%;
      }

      div {
        height: 100%;
      }

      .main {
        float: left;
        width: 100%;
        background: #f00;
      }

      .main .content {
        margin-left: 200px;
        margin-right: 300px;
      }

      .left {
        width: 200px;
        background: #0f0;
        float: left;
        margin-left: -100%;
      }
```

```
      .right {
        width: 300px;
        background: #00f;
        float: left;
        margin-left: -300px;
      }
    </style>
  </head>
  <body>
    <div class="main">
      <div class="content">hello world</div>
    </div>
    <div class="left">你好</div>
    <div class="right">我好</div>
  </body>
</html>
```

flex 布局

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      html,
      body {
        height: 100%;
      }

      div {
        height: 100%;
      }

      .container {
        display: flex;
      }

      .content {
        flex: 1 1;
        order: 2;
        background: #f00;
      }

      .left {
        flex: 0 0 200px;
        order: 1;
        background: #0f0;
      }

      .right {
        flex: 0 0 300px;
        order: 3;
        background: #00f;
```

```
      }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="content">hello world</div>
      <div class="left">你好</div>
      <div class="right">我好</div>
    </div>
  </body>
</html>
```

7. CSS 动画有哪些?

animation、transition、transform、translate 这几个属性要搞清楚：

- animation：用于设置动画属性，他是一个简写的属性，包含6个属性
- transition：用于设置元素的样式过度，和animation有着类似的效果，但细节上有很大的不同
- transform：用于元素进行旋转、缩放、移动或倾斜，和设置样式的动画并没有什么关系
- translate：translate只是transform的一个属性值，即移动，除此之外还有 scale 等

8. 用css2和css3分别写一下垂直居中和水平居中

CSS2

水平居中：

- div + margin: auto;
- span + text-align

垂直居中

- 使用 position 然后 left/top 和 margin 的方式垂直居中（已知宽高和未知宽高）
- 使用 position + margin
- 使用 display: table-cell;

已知宽高，进行水平垂直居中

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .outer {
        position: relative;
        width: 400px;
        height: 600px;
        background: blue;
      }

      .inner {
```

```
        position: absolute;
        width: 200px;
        height: 300px;
        background: red;
        left: 50%;
        top: 50%;
        margin: -150px 0 0 -100px;
      }
    </style>
  </head>
  <body>
    <div class="outer">
      <div class="inner"></div>
    </div>
  </body>
</html>
```

宽高未知，比如 内联元素，进行水平垂直居中

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .outer {
        width: 400px;
        height: 600px;
        /* background: blue; */
        border: 1px solid red;
        background-color: transparent;
        position: relative;
      }

      .inner {
        position: absolute;
        background: red;
        left: 50%;
        top: 50%;
        transform: translate(-50%, -50%);
      }
    </style>
  </head>
  <body>
    <div class="outer">
      <span class="inner">我想居中显示</span>
    </div>
  </body>
</html>
```

绝对定位的 div 水平垂直居中

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .outer {
        width: 400px;
        height: 600px;
        /* background: blue; */
        border: 1px solid red;
        background-color: transparent;
        position: relative;
      }

      .inner {
        position: absolute;
        background: red;
        left: 0;
        right: 0;
        bottom: 0;
        top: 0;
        width: 200px;
        height: 300px;
        margin: auto;
        border: 1px solid blue;
      }
    </style>
  </head>
  <body>
    <div class="outer">
      <div class="inner">我想居中显示</div>
    </div>
  </body>
</html>
```

图片和其他元素使用 display: table-cell; 进行垂直居中

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .outer {
        width: 400px;
        height: 600px;
        /* background: blue; */
        border: 1px solid red;
        background-color: transparent;
        display: table-cell;
        vertical-align: middle;
      }
```

```
      .inner {
        background: red;
        width: 200px;
        height: 300px;
        border: 1px solid blue;
        margin: 0 auto;
      }
    </style>
  </head>
  <body>
    <div class="outer">
      <div class="inner">我想居中显示</div>
    </div>
  </body>
</html>
```

CSS3

垂直、水平居中

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .outer {
        width: 400px;
        height: 600px;

        display: flex;

        /* 垂直居中 */
        align-items: center;

        /* 水平居中 */
        justify-content: center;
        border: 1px solid red;
        background-color: transparent;
      }

      .inner {
        background: red;
        width: 200px;
        height: 300px;
        border: 1px solid blue;
      }
    </style>
  </head>
  <body>
    <div class="outer">
      <div class="inner">我想居中显示</div>
    </div>
  </body>
```

```
</html>
```

9. visibility 和 display 的差别（还有opacity）

- visibility：设置 hidden 会隐藏元素，但是其位置还存在与页面文档流中，不会被删除，所以会触发浏览器渲染引擎的重绘
- display：设置了 none 属性会隐藏元素，且其位置也不会被保留下来，所以会触发浏览器渲染引擎的回流和重绘。
- opacity：会将元素设置为透明，但是其位置也在页面文档流中，不会被删除，所以会触发浏览器渲染引擎的重绘

10. opacity 可以有过渡效果嘛?

可以设置过渡效果

11. BFC 与 IFC 区别

BFC 是块级格式上下文，IFC 是行内格式上下文：

- 内部的 Box 会水平放置
- 水平的间距由 margin，padding，border 决定

12. BFC会与float元素相互覆盖吗？为什么？举例说明

不会，因为 BFC 是页面中一个独立的隔离容器，其内部的元素不会与外部的元素相互影响，比如两个 div，上面的 div 设置了 float，那么如果下面的元素不是 BFC，也没有设置 float，会形成对上面的元素进行包裹内容的情况，如果设置了下面元素为 overflow：hidden；属性那么就能够实现经典的两列布局，左边内容固定宽度，右边因为是 BFC 所以会进行自适应。

13. 了解box-sizing吗?

box-sizing 属性可以被用来调整这些表现:

- `content-box` 是默认值。如果你设置一个元素的宽为100px，那么这个元素的内容区会有100px宽，并且任何边框和内边距的宽度都会被增加到最后绘制出来的元素宽度中。
- `border-box` 告诉浏览器：你想要设置的边框和内边距的值是包含在width内的。也就是说，如果你将一个元素的width设为100px，那么这100px会包含它的border和padding，内容区的实际宽度是width减去(border + padding)的值。大多数情况下，这使得我们更容易地设定一个元素的宽高。

14. 什么是 BFC

BFC（Block Formatting Context）格式化上下文，是 Web 页面中盒模型布局的 CSS 渲染模式，指一个独立的渲染区域或者说是一个隔离的独立容器。

形成 BFC 的条件

五种：

- 浮动元素，float 除 none 以外的值
- 定位元素，position（absolute，fixed）
- display 为以下其中之一的值 inline-block，table-cell，table-caption
- overflow 除了 visible 以外的值（hidden，auto，scroll）
- HTML 就是一个 BFC

BFC 的特性：

- 内部的 Box 会在垂直方向上一个接一个的放置。
- 垂直方向上的距离由 margin 决定
- bfc 的区域不会与 float 的元素区域重叠。
- 计算 bfc 的高度时，浮动元素也参与计算
- bfc 就是页面上的一个独立容器，容器里面的子元素不会影响外面元素。

15. 了解盒模型吗?

CSS盒模型本质上是一个盒子，封装周围的HTML元素，它包括：`外边距（margin）`、`边框（border）`、`内边距（padding）`、`实际内容（content）`四个属性。 CSS盒模型：**标准模型 + IE模型**

标准盒子模型：宽度=内容的宽度（content）+ border + padding

低版本IE盒子模型：宽度=内容宽度（content+border+padding），如何设置成 IE 盒子模型:

```
box-sizing: border-box;
```

16. 说一下你知道的position属性，都有啥特点?

static：无特殊定位，对象遵循正常文档流。top，right，bottom，left等属性不会被应用。

relative：对象遵循正常文档流，但将依据top，right，bottom，left等属性在正常文档流中偏移位置。而其层叠通过z-index属性定义。

absolute：对象脱离正常文档流，使用top，right，bottom，left等属性进行绝对定位。而其层叠通过z-index属性定义。

fixed：对象脱离正常文档流，使用top，right，bottom，left等属性以窗口为参考点进行定位，当出现滚动条时，对象不会随着滚动。而其层叠通过z-index属性定义。

sticky：具体是类似 relative 和 fixed，在 viewport 视口滚动到阈值之前应用 relative，滚动到阈值之后应用 fixed 布局，由 top 决定。

17. 两个div上下排列，都设margin，有什么现象?

- 都正取大
- 一正一负相加

问：为什么会有这种现象？你能解释一下吗

是由块级格式上下文决定的，BFC，元素在 BFC 中会进行上下排列，然后垂直距离由 margin 决定，并且会发生重叠，具体表现为同正取最大的，同负取绝对值最大的，一正一负，相加

BFC 是页面中一个独立的隔离容器，内部的子元素不会影响到外部的元素。

18. 清除浮动有哪些方法?

不清楚浮动会发生高度塌陷：浮动元素父元素高度自适应（父元素不写高度时，子元素写了浮动后，父元素会发生高度塌陷）

- clear清除浮动（添加空div法）在浮动元素下方添加空div,并给该元素写css样式：{clear:both;height:0;overflow:hidden;}
- 给浮动元素父级设置高度
- 父级同时浮动（需要给父级同级元素添加浮动）
- 父级设置成inline-block，其margin: 0 auto居中方式失效
- 给父级添加overflow:hidden 清除浮动方法
- 万能清除法 after伪类 清浮动（现在主流方法，推荐使用）

```
.float_div:after{
  content:".";
  clear:both;
  display:block;
  height:0;
  overflow:hidden;
  visibility:hidden;
}
.float_div{
  zoom:1
}
```

JavaScript 基础

1. 问：0.1 + 0.2 === 0.3 嘛？为什么?

JavaScirpt 使用 Number 类型来表示数字（整数或浮点数），遵循 IEEE 754 标准，通过 64 位来表示一个数字（1 + 11 + 52）

- 1 符号位，0 表示正数，1 表示负数 s
- 11 指数位（e）
- 52 尾数，小数部分（即有效数字）

最大安全数字：Number.MAX_SAFE_INTEGER = Math.pow(2, 53) - 1，转换成整数就是 16 位，所以 0.1 === 0.1，是因为通过 toPrecision(16) 去有效位之后，两者是相等的。

在两数相加时，会先转换成二进制，0.1 和 0.2 转换成二进制的时候尾数会发生无限循环，然后进行对阶运算，JS 引擎对二进制进行截断，所以造成精度丢失。

所以总结：**精度丢失可能出现在进制转换和对阶运算中**

2. JS 数据类型

基本类型：Number、Boolean、String、null、undefined、symbol（ES6 新增的），BigInt（ES2020） 引用类型：Object，对象子类型（Array，Function）

3. JS 整数是怎么表示的?

通过 Number 类型来表示，遵循 IEEE754 标准，通过 64 位来表示一个数字，（1 + 11 + 52），最大安全数字是 Math.pow(2, 53) - 1，对于 16 位十进制。（符号位 + 指数位 + 小数部分有效位）

4. Number() 的存储空间是多大？如果后台发送了一个超过最大自己的数字怎么办

Math.pow(2, 53)，53 为有效数字，会发生截断，等于 JS 能支持的最大数字。

5. 写代码：实现函数能够深度克隆基本类型

浅克隆：

```
function shallowClone(obj) {
  let cloneObj = {};

  for (let i in obj) {
    cloneObj[i] = obj[i];
  }

  return cloneObj;
}
```

深克隆：

- 考虑基础类型
- 引用类型
  - RegExp、Date、函数 不是 JSON 安全的
  - 会丢失 constructor，所有的构造函数都指向 Object
  - 破解循环引用

```
function deepCopy(obj) {
  if (typeof obj === 'object') {
    var result = obj.constructor === Array ? [] : {};

    for (var i in obj) {
      result[i] = typeof obj[i] === 'object' ? deepCopy(obj[i]) : obj[i];
    }
  } else {
    var result = obj;
  }

  return result;
}
```

6. 事件流

事件流是网页元素接收事件的顺序，"DOM2级事件"规定的事件流包括三个阶段：事件捕获阶段、处于目标阶段、事件冒泡阶段。 首先发生的事件捕获，为截获事件提供机会。然后是实际的目标接受事件。最后一个阶段是时间冒泡阶段，可以在这个阶段对事件做出响应。 虽然捕获阶段在规范中规定不允许响应事件，但是实际上还是会执行，所以有两次机会获取到目标对象。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>事件冒泡</title>
</head>
<body>
    <div>
        <p id="parEle">我是父元素    <span id="sonEle">我是子元素</span></p>
    </div>
</body>
</html>
<script type="text/javascript">
var sonEle = document.getElementById('sonEle');
var parEle = document.getElementById('parEle');

parEle.addEventListener('click', function () {
    alert('父级 冒泡');
}, false);
parEle.addEventListener('click', function () {
    alert('父级 捕获');
}, true);

sonEle.addEventListener('click', function () {
    alert('子级冒泡');
}, false);
sonEle.addEventListener('click', function () {
    alert('子级捕获');
}, true);

</script>
```

当容器元素及嵌套元素，即在 `捕获阶段` 又在 `冒泡阶段` 调用事件处理程序时：**事件按DOM事件流的顺序**执行事件处理程序：

- 父级捕获
- 子级冒泡
- 子级捕获
- 父级冒泡

且当事件处于目标阶段时，事件调用顺序决定于绑定事件的**书写顺序**，按上面的例子为，先调用冒泡阶段的事件处理程序，再调用捕获阶段的事件处理程序。依次alert出“子集冒泡”，“子集捕获”。

IE 兼容

- attchEvent('on' + type, handler)
- detachEvent('on' + type, handler)

7. 事件是如何实现的?

基于发布订阅模式，就是在浏览器加载的时候会读取事件相关的代码，但是只有实际等到具体的事件触发的时候才会执行。

比如点击按钮，这是个事件（Event），而负责处理事件的代码段通常被称为事件处理程序（Event Handler），也就是「启动对话框的显示」这个动作。

在 Web 端，我们常见的就是 DOM 事件：

- DOM0 级事件，直接在 html 元素上绑定 on-event，比如 onclick，取消的话，dom.onclick = null，同一个事件只能有一个处理程序，后面的会覆盖前面的。
- DOM2 级事件，通过 addEventListener 注册事件，通过 removeEventListener 来删除事件，一个事件可以有多个事件处理程序，按顺序执行，捕获事件和冒泡事件
- DOM3级事件，增加了事件类型，比如 UI 事件，焦点事件，鼠标事件

8. new 一个函数发生了什么

构造调用：

- 创造一个全新的对象
- 这个对象会被执行 [[Prototype]] 连接，将这个新对象的 [[Prototype]] 链接到这个构造函数.prototype 所指向的对象
- 这个新对象会绑定到函数调用的 this
- 如果函数没有返回其他对象，那么 new 表达式中的函数调用会自动返回这个新对象

9. new 一个构造函数，如果函数返回 `return {}` 、 `return null` ，`return 1` ， `return true` 会发生什么情况?

如果函数返回一个对象，那么new 这个函数调用返回这个函数的返回对象，否则返回 new 创建的新对象

10. symbol有什么用处

可以用来表示一个独一无二的变量防止命名冲突。但是面试官问还有吗？我没想出其他的用处就直接答我不知道了，还可以利用 `symbol` 不会被常规的方法（除了 `Object.getOwnPropertySymbols` 外）遍历到，所以可以用来模拟私有变量。

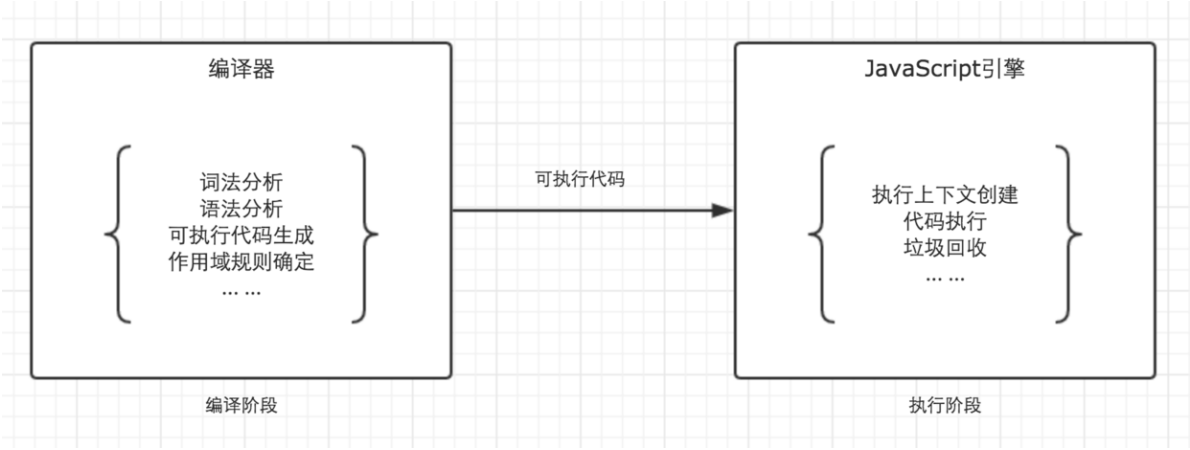
主要用来提供遍历接口，布置了 `symbol.iterator` 的对象才可以使用 `for···of` 循环，可以统一处理数据结构。调用之后回返回一个遍历器对象，包含有一个 next 方法，使用 next 方法后有两个返回值 value 和 done 分别表示函数当前执行位置的值和是否遍历完毕。

Symbol.for() 可以在全局访问 symbol

11. 闭包是什么?

闭包是指有权访问另外一个函数作用域中的变量的函数

JavaScript代码的整个执行过程，分为两个阶段，代码编译阶段与代码执行阶段。编译阶段由编译器完成，将代码翻译成可执行代码，这个阶段作用域规则会确定。执行阶段由引擎完成，主要任务是执行可执行代码，执行上下文在这个阶段创建。



12. 闭包产生的本质

当前环境中存在指向父级作用域的引用

13. 一般如何产生闭包

- 返回函数
- 函数当做参数传递

14. 闭包的应用场景

- 柯里化 bind
- 模块

15. 什么是作用域?

ES5 中只存在两种作用域：全局作用域和函数作用域。在 JavaScript 中，我们将作用域定义为一套规则，这套规则用来管理引擎如何在当前作用域以及嵌套子作用域中根据标识符名称进行变量（变量名或者函数名）查找

16. 什么是作用域链?

首先要了解作用域链，当访问一个变量时，编译器在执行这段代码时，会首先从当前的作用域中查找是否有这个标识符，如果没有找到，就会去父作用域查找，如果父作用域还没找到继续向上查找，直到全局作用域为止，，而作用域链，就是有当前作用域与上层作用域的一系列变量对象组成，它保证了当前执行的作用域对符合访问权限的变量和函数的有序访问。

17. NaN 是什么，用 typeof 会输出什么?

Not a Number，表示非数字，typeof NaN === 'number'

18. JS 隐式转换，显示转换

一般非基础类型进行转换时会先调用 valueOf，如果 valueOf 无法返回基本类型值，就会调用 toString

1）字符串和数字

- "+" 操作符，如果有一个为字符串，那么都转化到字符串然后执行字符串拼接
- "-" 操作符，转换为数字，相减 (-a, a * 1 a/1) 都能进行隐式强制类型转换

```
[] + {} 和 {} + []
```

2）布尔值到数字

- 1 + true = 2
- 1 + false = 1

3）转换为布尔值

- for 中第二个
- while
- if
- 三元表达式
- || （逻辑或） && （逻辑与）左边的操作数

4）符号

- 不能被转换为数字
- 能被转换为布尔值（都是 true）
- 可以被转换成字符串 "Symbol(cool)"

5）宽松相等和严格相等

宽松相等允许进行强制类型转换，而严格相等不允许

（1）字符串与数字

转换为数字然后比较

（2）其他类型与布尔类型

- 先把布尔类型转换为数字，然后继续进行比较

（3）对象与非对象

- 执行对象的 ToPrimitive(对象）然后继续进行比较

（4）假值列表

- undefined
- null
- false
- +0, -0, NaN
- ""

19. 了解 this 嘛，bind，call，apply 具体指什么

它们都是函数的方法

`call: Array.prototype.call(this, args1, args2])` `apply: Array.prototype.apply(this, [args1, args2])`：ES6 之前用来展开数组调用，`foo.appy(null, [])`，ES6 之后使用 ... 操作符

- New 绑定 > 显示绑定 > 隐式绑定 > 默认绑定
- 如果需要使用 bind 的柯里化和 apply 的数组解构，绑定到 null，尽可能使用 Object.create(null) 创建一个 DMZ 对象

四条规则：

- 默认绑定，没有其他修饰（bind、apply、call），在非严格模式下定义指向全局对象，在严格模式下定义指向 undefined

```
function foo() {
    console.log(this.a);
}

var a = 2;
foo();
```

- 隐式绑定：调用位置是否有上下文对象，或者是否被某个对象拥有或者包含，那么隐式绑定规则会把函数调用中的 this 绑定到这个上下文对象。而且，对象属性链只有上一层或者说最后一层在调用位置中起作用

```
function foo() {
  console.log(this.a);
}

var obj = {
  a: 2,
  foo: foo,
}

obj.foo(); // 2
```

- 显示绑定：通过在函数上运行 call 和 apply，来显示的绑定 this

```
function foo() {
  console.log(this.a);
}

var obj = {
  a: 2
};

foo.call(obj);
```

显示绑定之硬绑定

```
function foo(something) {
  console.log(this.a, something);

  return this.a + something;
}

function bind(fn, obj) {
  return function() {
    return fn.apply(obj, arguments);
  };
}

var obj = {
  a: 2
}

var bar = bind(foo, obj);
```

New 绑定，new 调用函数会创建一个全新的对象，并将这个对象绑定到函数调用的 this。

- New 绑定时，如果是 new 一个硬绑定函数，那么会用 new 新建的对象替换这个硬绑定 this，

```
function foo(a) {
  this.a = a;
}

var bar = new foo(2);
console.log(bar.a)
```


20. 手写 bind、apply、call

```
// call

Function.prototype.call = function (context, ...args) {
  context = context || window;

  const fnSymbol = Symbol("fn");
  context[fnSymbol] = this;

  context[fnSymbol](...args);
  delete context[fnSymbol];
}
```

```
// apply

Function.prototype.apply = function (context, argsArr) {
  context = context || window;

  const fnSymbol = Symbol("fn");
  context[fnSymbol] = this;

  context[fnSymbol](...argsArr);
  delete context[fnSymbol];
}
```

```
// bind

Function.prototype.bind = function (context, ...args) {
  context = context || window;
  const fnSymbol = Symbol("fn");
  context[fnSymbol] = this;

  return function (..._args) {
    args = args.concat(_args);

    context[fnSymbol](...args);
    delete context[fnSymbol];
  }
}
```


21. setTimeout(fn, 0)多久才执行，Event Loop

setTimeout 按照顺序放到队列里面，然后等待函数调用栈清空之后才开始执行，而这些操作进入队列的顺序，则由设定的延迟时间来决定

22. 手写题：Promise 原理

```
class MyPromise {
  constructor(fn) {
    this.resolvedCallbacks = [];
    this.rejectedCallbacks = [];

    this.state = 'PENDING';
    this.value = '';

    fn(this.resolve.bind(this), this.reject.bind(this));

  }

  resolve(value) {
    if (this.state === 'PENDING') {
      this.state = 'RESOLVED';
      this.value = value;

      this.resolvedCallbacks.map(cb => cb(value));
    }
  }

  reject(value) {
    if (this.state === 'PENDING') {
      this.state = 'REJECTED';
      this.value = value;

      this.rejectedCallbacks.map(cb => cb(value));
    }
  }

  then(onFulfilled, onRejected) {
    if (this.state === 'PENDING') {
      this.resolvedCallbacks.push(onFulfilled);
      this.rejectedCallbacks.push(onRejected);

    }

    if (this.state === 'RESOLVED') {
      onFulfilled(this.value);
    }

    if (this.state === 'REJECTED') {
      onRejected(this.value);
    }
  }
}
```

23. js脚本加载问题，async、defer问题

- 如果依赖其他脚本和 DOM 结果，使用 defer
- 如果与 DOM 和其他脚本依赖不强时，使用 async

24. 如何判断一个对象是不是空对象?

Object.keys(obj).length === 0

手写题：在线编程，getUrlParams(url,key); 就是很简单的获取url的某个参数的问题，但要考虑边界情况，多个返回值等等

25. <script src='xxx' 'xxx'/>外部js文件先加载还是onload先执行，为什么?

onload 是所以加载完成之后执行的

26. 怎么加事件监听

onclick 和 addEventListener

27. 事件传播机制（事件流）

冒泡和捕获

28. 说一下原型链和原型链的继承吧

- 所有普通的 [[Prototype]] 链最终都会指向内置的 Object.prototype，其包含了 JavaScript 中许多通用的功能
- 为什么能创建“类”，借助一种特殊的属性：所有的函数默认都会拥有一个名为 prototype 的共有且不可枚举的属性，它会指向另外一个对象，这个对象通常被称为函数的原型

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

Person.prototype.constructor = Person
```

- 在发生 new 构造函数调用时，会将创建的新对象的 [[Prototype]] 链接到 Person.prototype 指向的对象，这个机制就被称为原型链继承
- 方法定义在原型上，属性定义在构造函数上
- 首先要说一下 JS 原型和实例的关系：每个构造函数 （constructor）都有一个原型对象（prototype），这个原型对象包含一个指向此构造函数的指针属性，通过 new 进行构造函数调用生成的实例，此实例包含一个指向原型对象的指针，也就是通过 [[Prototype]] 链接到了这个原型对象
- 然后说一下 JS 中属性的查找：当我们试图引用实例对象的某个属性时，是按照这样的方式去查找的，首先查找实例对象上是否有这个属性，如果没有找到，就去构造这个实例对象的构造函数的 prototype 所指向的对象上去查找，如果还找不到，就从这个 prototype 对象所指向的构造函数的 prototype 原型对象上去查找

- 什么是原型链：这样逐级查找形似一个链条，且通过 [[Prototype]] 属性链接，所以被称为原型链
- 什么是原型链继承，类比类的继承：当有两个构造函数 A 和 B，将一个构造函数 A 的原型对象的，通过其 [[Prototype]] 属性链接到另外一个 B 构造函数的原型对象时，这个过程被称之为原型继承。

29. 说下对 JS 的了解吧

是基于原型的动态语言，主要独特特性有 this、原型和原型链。

JS 严格意义上来说分为：语言标准部分（ECMAScript）+ 宿主环境部分

语言标准部分

2015 年发布 ES6，引入诸多新特性使得能够编写大型项目变成可能，标准自 2015 之后以年号代号，每年一更

宿主环境部分

- 在浏览器宿主环境包括 DOM + BOM 等
- 在 Node，宿主环境包括一些文件、数据库、网络、与操作系统的交互等

30. 数组能够调用的函数有那些?

- push
- pop
- splice
- slice
- shift
- unshift
- sort
- find
- findIndex
- map/filter/reduce 等函数式编程方法
- 还有一些原型链上的方法：toString/valudOf

31. 如何判断数组类型

Array.isArray

32. 函数中的arguments是数组吗？类数组转数组的方法了解一下?

是类数组，是属于鸭子类型的范畴，长得像数组，

- ... 运算符
- Array.from
- Array.prototype.slice.apply(arguments)

33. 用过 TypeScript 吗？它的作用是什么？

为 JS 添加类型支持，以及提供最新版的 ES 语法的支持，是的利于团队协作和排错，开发大型项目

34. PWA使用过吗？serviceWorker的使用原理是啥？

`渐进式网络应用（PWA）`是谷歌在2015年底提出的概念。基本上算是web应用程序，但在外观和感觉上与`原生app`类似。支持`PWA`的网站可以提供脱机工作、推送通知和设备硬件访问等功能。

`Service Worker`是浏览器在后台独立于网页运行的脚本，它打开了通向不需要网页或用户交互的功能的大门。 现在，它们已包括如推送通知和后台同步等功能。 将来，`Service Worker`将会支持如定期同步或地理围栏等其他功能。 本教程讨论的核心功能是拦截和处理网络请求，包括通过程序来管理缓存中的响应。

35. ES6 之前使用 prototype 实现继承

Object.create() 会创建一个 “新” 对象，然后将此对象内部的 [[Prototype]] 关联到你指定的对象（Foo.prototype）。Object.create(null) 创建一个空 [[Prototype]] 链接的对象，这个对象无法进行委托。

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

Foo.prototype.myName = function () {
  return this.name;
}

// 继承属性，通过借用构造函数调用
function Bar(name, label) {
  Foo.call(this, name);
  this.label = label;
}

// 继承方法，创建备份
Bar.prototype = Object.create(Foo.prototype);

// 必须设置回正确的构造函数，要不然在会发生判断类型出错
Bar.prototype.constructor = Bar;

 // 必须在上一步之后
Bar.prototype.myLabel = function () {
  return this.label;
}

var a = new Bar("a", "obj a");

a.myName(); // "a"
a.myLabel(); // "obj a"
```

36. 如果一个构造函数，bind了一个对象，用这个构造函数创建出的实例会继承这个对象的属性吗？为什么？

不会继承，因为根据 this 绑定四大规则，new 绑定的优先级高于 bind 显示绑定，通过 new 进行构造函数调用时，会创建一个新对象，这个新对象会代替 bind 的对象绑定，作为此函数的 this，并且在此函数没有返回对象的情况下，返回这个新建的对象

37. 箭头函数和普通函数有啥区别？箭头函数能当构造函数吗？

普通函数通过 function 关键字定义， this 无法结合词法作用域使用，在运行时绑定，只取决于函数的调用方式，在哪里被调用，调用位置。（取决于调用者，和是否独立运行）

箭头函数使用被称为 “胖箭头” 的操作 `=>` 定义，箭头函数不应用普通函数 this 绑定的四种规则，而是根据外层（函数或全局）的作用域来决定 this，且箭头函数的绑定无法被修改（new 也不行）。

- 箭头函数常用于回调函数中，包括事件处理器或定时器
- 箭头函数和 var self = this，都试图取代传统的 this 运行机制，将 this 的绑定拉回到词法作用域
- 没有原型、没有 this、没有 super，没有 arguments，没有 new.target
- 不能通过 new 关键字调用
    - 一个函数内部有两个方法：[[Call]] 和 [[Construct]]，在通过 new 进行函数调用时，会执行 [[construct]] 方法，创建一个实例对象，然后再执行这个函数体，将函数的 this 绑定在这个实例对象上
    - 当直接调用时，执行 [[Call]] 方法，直接执行函数体
    - 箭头函数没有 [[Construct]] 方法，不能被用作构造函数调用，当使用 new 进行函数调用时会报错。

```
function foo() {
  return (a) => {
    console.log(this.a);
  }
}

var obj1 = {
  a: 2
}

var obj2 = {
  a: 3
}

var bar = foo.call(obj1);
bar.call(obj2);
```

38. 知道 ES6 的 Class 嘛？Static 关键字有了解嘛

为这个类的函数对象直接添加方法，而不是加在这个函数对象的原型对象上

39. 事件循环机制 （Event Loop）

事件循环机制从整体上告诉了我们 JavaScript 代码的执行顺序 `Event Loop` 即事件循环，是指浏览器或 `Node` 的一种解决 `javaScript` 单线程运行时不会阻塞的一种机制，也就是我们经常使用**异步**的原理。

先执行宏任务队列，然后执行微任务队列，然后开始下一轮事件循环，继续先执行宏任务队列，再执行微任务队列。

- 宏任务：script/setTimeout/setInterval/setImmediate/ I/O / UI Rendering
- 微任务：process.nextTick()/Promise

上诉的 setTimeout 和 setInterval 等都是任务源，真正进入任务队列的是他们分发的任务。

优先级

- setTimeout = setInterval 一个队列
- setTimeout > setImmediate
- process.nextTick > Promise

```
for (const macroTask of macroTaskQueue) {
  handleMacroTask();
  for (const microTask of microTaskQueue) {
    handleMicroTask(microTask);
  }
}
```

40. 手写题：数组扁平化

```
function flatten(arr) {
  let result = [];

  for (let i = 0; i < arr.length; i++) {
    if (Array.isArray(arr[i])) {
      result = result.concat(flatten(arr[i]));
    } else {
      result = result.concat(arr[i]);
    }
  }

  return result;
}

const a = [1, [2, [3, 4]]];
console.log(flatten(a));
```

41. 手写题：实现柯里化

预先设置一些参数

柯里化是什么：是指这样一个函数，它接收函数 A，并且能返回一个新的函数，这个新的函数能够处理函数 A 的剩余参数

```
function createCurry(func, args) {
```

```
  var argity = func.length;
  var args = args || [];

  return function () {
    var _args = [].slice.apply(arguments);
    args.push(..._args);

    if (args.length < argity) {
      return createCurry.call(this, func, args);
    }

    return func.apply(this, args);
  }
}
```

42. 手写题：数组去重

```
Array.from(new Set([1, 1, 2, 2]))
```

43. let 闭包

let 会产生临时性死区，在当前的执行上下文中，会进行变量提升，但是未被初始化，所以在执行上下文执行阶段，执行代码如果还没有执行到变量赋值，就引用此变量就会报错，此变量未初始化。

44. 变量提升

函数在运行的时候，会首先创建执行上下文，然后将执行上下文入栈，然后当此执行上下文处于栈顶时，开始运行执行上下文。

在创建执行上下文的过程中会做三件事：创建变量对象，创建作用域链，确定 this 指向，其中创建变量对象的过程中，首先会为 arguments 创建一个属性，值为 arguments，然后会扫码 function 函数声明，创建一个同名属性，值为函数的引用，接着会扫码 var 变量声明，创建一个同名属性，值为 undefined，这就是变量提升 。

45. instance 如何使用

左边可以是任意值，右边只能是函数

```
'hello tuture' instanceof String // false
```

Vue框架

1. active-class是哪个组件的属性？嵌套路由怎么定义？

vue-router模块的router-link组件。

2. 怎么定义vue-router的动态路由？怎么获取传过来的动态参数？

在router目录下的index.js文件中，对path属性加上/:id。 使用router对象的params.id

3. vue-router有哪几种导航钩子？

三种，一种是全局导航钩子：router.beforeEach(to,from,next)，作用：跳转前进行判断拦截。第二种：组件内的钩子；第三种：单独路由独享组件

4. scss是什么？在vue.cli中的安装使用步骤是？有哪几大特性？

css的预编译。

使用步骤：

第一步：用npm 下三个loader（sass-loader、css-loader、node-sass）

第二步：在build目录找到webpack.base.config.js，在那个extends属性中加一个拓展.scss

第三步：还是在同一个文件，配置一个module属性

第四步：然后在组件的style标签加上lang属性，例如：lang="scss"

有哪几大特性:

1、可以用变量，例如（$变量名称=值）；

2、可以用混合器，例如（）

3、可以嵌套

5. mint-ui是什么？怎么使用？说出至少三个组件使用方法？

基于vue的前端组件库。npm安装，然后import样式和js，vue.use（mintUi）全局引入。在单个组件局部引入：import {Toast} from 'mint-ui'。组件一：Toast('登录成功')；组件二：mint-header；组件三：mint-swiper

6. v-model是什么？怎么使用？ vue中标签怎么绑定事件？

可以实现双向绑定，指令（v-class、v-for、v-if、v-show、v-on）。vue的model层的data属性。绑定事件：<input @click=doLog() />

7. axios是什么？怎么使用？描述使用它实现登录功能的流程？

请求后台资源的模块。npm install axios -S装好，然后发送的是跨域，需在配置文件中config/index.js进行设置。后台如果是Tp5则定义一个资源路由。js中使用import进来，然后.get或.post。返回在.then函数中如果成功，失败则是在.catch函数中

8. axios+tp5进阶中，调用axios.post('api/user')是进行的什么操作？axios.put('api/user/8')呢？

跨域，添加用户操作，更新操作。

9. 什么是RESTful API？怎么使用？

是一个api的标准，无状态请求。请求的路由地址是固定的，如果是tp5则先路由配置中把资源路由配置好。标准有：.post .put .delete

10. vuex是什么？怎么使用？哪种功能场景使用它？

vue框架中状态管理。在main.js引入store，注入。新建了一个目录store，..... export 。场景有：单页应用中，组件之间的状态。音乐播放、登录状态、加入购物车

11. mvvm框架是什么？它和其它框架（jquery）的区别是什么？哪些场景适合？

一个model+view+viewModel框架，数据模型model，viewModel连接两个

区别：vue数据驱动，通过数据来显示视图层而不是节点操作。

场景：数据操作比较多的场景，更加便捷

12. 自定义指令（v-check、v-focus）的方法有哪些？它有哪些钩子函数？还有哪些钩子函数参数？

全局定义指令：在vue对象的directive方法里面有两个参数，一个是指令名称，另外一个是函数。组件内定义指令：directives

钩子函数：bind（绑定事件触发）、inserted(节点插入的时候触发)、update（组件内相关更新）

钩子函数参数：el、binding

13. 说出至少4种vue当中的指令和它的用法？

v-if：判断是否隐藏；v-for：数据循环出来；v-bind:class：绑定一个属性；v-model：实现双向绑定

14. vue-router是什么？它有哪些组件？

vue用来写路由一个插件。router-link、router-view

15. 导航钩子有哪些？它们有哪些参数？

导航钩子有：a/全局钩子和组件内独享的钩子。b/beforeRouteEnter、afterEnter、beforeRouterUpdate、beforeRouteLeave

参数：有to（去的那个路由）、from（离开的路由）、next（一定要用这个函数才能去到下一个路由，如果不用就拦截）最常用就这几种

16. Vue的双向数据绑定原理是什么？

vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过`Object.defineProperty()`来劫持各个属性的`setter`，`getter`，在数据变动时发布消息给订阅者，触发相应的监听回调。

具体步骤：

第一步：需要observe的数据对象进行递归遍历，包括子属性对象的属性，都加上`setter`和`getter`这样的话，给这个对象的某个值赋值，就会触发`setter`，那么就能监听到了数据变化

第二步：compile解析模板指令，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图

第三步：Watcher订阅者是Observer和Compile之间通信的桥梁，主要做的事情是:
1、在自身实例化时往属性订阅器(dep)里面添加自己
2、自身必须有一个update()方法
3、待属性变动dep.notice()通知时，能调用自身的update()方法，并触发Compile中绑定的回调，则功成身退。

第四步：MVVM作为数据绑定的入口，整合Observer、Compile和Watcher三者，通过Observer来监听自己的model数据变化，通过Compile来解析编译模板指令，最终利用Watcher搭起Observer和Compile之间的通信桥梁，达到数据变化 -> 视图更新；视图交互变化(input) -> 数据model变更的双向绑定效果。

ps：16题答案同样适合”vue data是怎么实现的？”此面试题。

17. 请详细说下你对vue生命周期的理解？

答：总共分为8个阶段创建前/后，载入前/后，更新前/后，销毁前/后。

创建前/后： 在beforeCreated阶段，vue实例的挂载元素$el和数据对象data都为undefined，还未初始化。在created阶段，vue实例的数据对象data有了，$el还没有。

载入前/后：在beforeMount阶段，vue实例的$el和data都初始化了，但还是挂载之前为虚拟的dom节点，data.message还未替换。在mounted阶段，vue实例挂载完成，data.message成功渲染。

更新前/后：当data变化时，会触发beforeUpdate和updated方法。

销毁前/后：在执行destroy方法后，对data的改变不会再触发周期函数，说明此时vue实例已经解除了事件监听以及和dom的绑定，但是dom结构依然存在

18. 请说下封装 vue 组件的过程?

答：首先，组件可以提升整个项目的开发效率。能够把页面抽象成多个相对独立的模块，解决了我们传统项目开发：效率低、难维护、复用性等问题。

然后，使用Vue.extend方法创建一个组件，然后使用Vue.component方法注册组件。子组件需要数据，可以在props中接受定义。而子组件修改好数据后，想把数据传递给父组件。可以采用emit方法。

19. 你是怎么认识vuex的?

答：vuex可以理解为一种开发模式或框架。比如PHP有thinkphp，java有spring等。
通过状态（数据源）集中管理驱动组件的变化（好比spring的IOC容器对bean进行集中管理）。

应用级的状态集中放在store中； 改变状态的方式是提交mutations，这是个同步的事物； 异步逻辑应该封装在action中。

20. vue-loader是什么？使用它的用途有哪些?

答：解析.vue文件的一个加载器，跟template/js/style转换成js模块。

用途：js可以写es6、style样式可以scss或less、template可以加jade等

21. 请说出vue.cli项目中src目录每个文件夹和文件的用法?

答：assets文件夹是放静态资源；components是放组件；router是定义路由相关的配置;view视图；app.vue是一个应用主组件；main.js是入口文件

22. vue.cli中怎样使用自定义的组件？有遇到过哪些问题吗?

答：第一步：在components目录新建你的组件文件（smithButton.vue），script一定要export default {

第二步：在需要用的页面（组件）中导入：import smithButton from '../components/smithButton.vue'

第三步：注入到vue的子组件的components属性上面,components:{smithButton}

第四步：在template视图view中使用，
问题有：smithButton命名，使用的时候则smith-button。

23. 聊聊你对Vue.js的template编译的理解?

答：简而言之，就是先转化成AST树，再得到的render函数返回VNode（Vue的虚拟DOM节点）

详情步骤：

首先，通过compile编译器把template编译成AST语法树（abstract syntax tree 即 源代码的抽象语法结构的树状表现形式），compile是createCompiler的返回值，createCompiler是用以创建编译器的。另外compile还负责合并option。

然后，AST会经过generate（将AST语法树转化成render funtion字符串的过程）得到render函数，render的返回值是VNode，VNode是Vue的虚拟DOM节点，里面有（标签名、子节点、文本等等）

24. Vuex是什么？为什么使用Vuex?

答：Vuex 类似 Redux 的状态管理器，用来管理Vue的所有组件状态。

当你打算开发大型单页应用（SPA），会出现多个视图组件依赖同一个状态，来自不同视图的行为需要变更同一个状态。

25. vuejs与angularjs的区别?

答：
一、定位：
虽然Vue.js被定义为MVC framework，但其实Vue本身还是一个library，加了一些其他的工具，可以被当成一个framework，而Angular 2虽然还是一个framework，但其实在设计之初，Angular 2的团队站在了更高的角度，希望做一个platform。

二、文档：
vue.js的更加亲切

三、性能：
angular所有的数据和方法都是挂载在$scope上。而vue的数据和方法都是挂载在vue上，只是数据挂载在vue的data,方法挂载在vue的methods上，vue的代码风格更加优雅，json格式书写代码。Vue.js 有更好的性能，并且非常非常容易优化，因为它不使用脏检查。Angular，当 watcher 越来越多时会变得越来越慢，因为作用域内的每一次变化，所有 watcher 都要重新计算。

其它区别：
渲染性能：Vue> react >angular。
使用场景：Vue React 覆盖中小型，大型项目。angular 一般用于大型（因为比较厚重）。

26. vue为什么不直接操作dom?

答：因为操作dom对象后，会触发一些浏览器行为，比如布局（layout）和绘制（paint）。 paint是一个耗时的过程，然而layout是一个更耗时的过程，我们无法确定layout一定是自上而下或是自下而上进行的，甚至一次layout会牵涉到整个文档布局的重新计算。浏览器的layout是lazy的，也就是说：在js脚本执行时，是不会去更新DOM的，任何对DOM的修改都会被暂存在一个队列中，在当前js的执行上下文完成执行后，会根据这个队列中的修改，进行一次layout。

27. 你怎么理解vue是一个渐进式的框架?

答：我觉得渐进式就是不必一开始就用Vue所有的全家桶，可以根据场景，按需使用想要的插件。也可以说就使用vue不需要太多的要求。

28、Vue声明组件的state是用data方法，那为什么data是通过一个function来返回一个对象，而不是直接写一个对象呢?

答：从语法上说，如果不用function返回就会出现语法错误导致编译不通过。从原理上的话，大概就是组件可以被多次创建，如果不使用function就会使所有调用该组件的页面公用同一个数据域，这样就失去了组件的概念了

29. 说下vue组件之间的通信?

答:
非父子组件间通信，Vue 有提供 Vuex，以状态共享方式来实现同信，对于这一点，应该注意考虑平衡，从整体设计角度去考量，确保引入她的必要。

父传子: this.$refs.xxx 子传父: this.$parent.xxx

还可以通过$emit方法出发一个消息，然后$on接收这个消息

30. vue中mixin与extend区别?

答:
全局注册混合对象，会影响到所有之后创建的vue实例，而Vue.extend是对单个实例进行扩展。

mixin 混合对象（组件复用）

同名钩子函数（bind，inserted，update，componentUpdate，unbind）将混合为一个数组，因此都将被调用，混合对象的钩子将在组件自身钩子之前调用

methods，components，directives将被混为同一个对象。两个对象的键名（方法名，属性名）冲突时，取组件（而非mixin）对象的键值对。

计算机网络基础

1. HTTP 缓存

HTTP 缓存又分为强缓存和协商缓存:

- 首先通过 Cache-Control 验证强缓存是否可用，如果强缓存可用，那么直接读取缓存
- 如果不可以，那么进入协商缓存阶段，发起 HTTP 请求，服务器通过请求头中是否带上 If-Modified-Since 和 If-None-Match 这些条件请求字段检查资源是否更新:
  - 若资源更新，那么返回资源和 200 状态码
  - 如果资源未更新，那么告诉浏览器直接使用缓存获取资源

2. HTTP 常用的状态码及使用场景?

- 1xx：表示目前是协议的中间状态，还需要后续请求
- 2xx：表示请求成功
- 3xx：表示重定向状态，需要重新请求
- 4xx：表示请求报文错误
- 5xx：服务器端错误

常用状态码:

- 101 切换请求协议，从 HTTP 切换到 WebSocket
- 200 请求成功，有响应体
- 301 永久重定向：会缓存
- 302 临时重定向：不会缓存
- 304 协商缓存命中
- 403 服务器禁止访问
- 404 资源未找到

- 400 请求错误
- 500 服务器端错误
- 503 服务器繁忙

3. 你知道 302 状态码是什么嘛？你平时浏览网页的过程中遇到过哪些 302 的场景?

而 302 表示临时重定向，这个资源只是暂时不能被访问了，但是之后过一段时间还是可以继续访问，一般是访问某个网站的资源需要权限时，会需要用户去登录，跳转到登录页面之后登录之后，还可以继续访问。

301 类似，都会跳转到一个新的网站，但是 301 代表访问的地址的资源被永久移除了，以后都不应该访问这个地址，搜索引擎抓取的时候也会用新的地址替换这个老的。可以在返回的响应的 location 首部去获取到返回的地址。301 的场景如下：

- 比如从 baidu.com，跳转到 baidu.com
- 域名换了

4. HTTP 常用的请求方式，区别和用途?

http/1.1 规定如下请求方法：

- GET：通用获取数据
- HEAD：获取资源的元信息
- POST：提交数据
- PUT：修改数据
- DELETE：删除数据
- CONNECT：建立连接隧道，用于代理服务器
- OPTIONS：列出可对资源实行的请求方法，常用于跨域
- TRACE：追踪请求-响应的传输路径

5. 你对计算机网络的认识怎么样

应用层、表示层、会话层、传输层、网络层、数据链路层、物理层

6. HTTPS 是什么？具体流程

HTTPS 是在 HTTP 和 TCP 之间建立了一个安全层，HTTP 与 TCP 通信的时候，必须先进过一个安全层，对数据包进行加密，然后将加密后的数据包传送给 TCP，相应的 TCP 必须将数据包解密，才能传给上面的 HTTP。

浏览器传输一个 client_random 和加密方法列表，服务器收到后，传给浏览器一个 server_random、加密方法列表和数字证书（包含了公钥），然后浏览器对数字证书进行合法验证，如果验证通过，则生成一个 pre_random，然后用公钥加密传给服务器，服务器用 client_random、server_random 和 pre_random，使用公钥加密生成 secret，然后之后的传输使用这个 secret 作为秘钥来进行数据的加解密。

7. 三次握手和四次挥手

为什么要进行三次握手：为了确认对方的发送和接收能力。

三次握手

三次握手主要流程：

- 一开始双方处于 CLOSED 状态，然后服务端开始监听某个端口进入 LISTEN 状态
- 然后客户端主动发起连接，发送 SYN，然后自己变为 SYN-SENT，seq = x
- 服务端收到之后，返回 SYN seq = y 和 ACK ack = x + 1（对于客户端发来的 SYN），自己变成 SYN-REVD
- 之后客户端再次发送 ACK seq = x + 1, ack = y + 1给服务端，自己变成 EASTABLISHED 状态，服务端收到 ACK，也进入 ESTABLISHED

SYN 需要对端确认，所以 ACK 的序列化要加一，凡是需要对端确认的，一点要消耗 TCP 报文的序列化

为什么不是两次?

无法确认客户端的接收能力。

如果首先客户端发送了 SYN 报文，但是滞留在网络中，TCP 以为丢包了，然后重传，两次握手建立了连接。

等到客户端关闭连接了。但是之后这个包如果到达了服务端，那么服务端接收到了，然后发送相应的数据表，就建立了链接，但是此时客户端已经关闭连接了，所以带来了链接资源的浪费。

为什么不是四次?

四次以上都可以，只不过 三次就够了

四次挥手

- 一开始都处于 ESTABLISH 状态，然后客户端发送 FIN 报文，带上 seq = p，状态变为 FIN-WAIT-1
- 服务端收到之后，发送 ACK 确认，ack = p + 1，然后进入 CLOSE-WAIT 状态
- 客户端收到之后进入 FIN-WAIT-2 状态
- 过了一会等数据处理完，再次发送 FIN、ACK，seq = q，ack = p + 1，进入 LAST-ACK 阶段
- 客户端收到 FIN 之后，客户端收到之后进入 TIME_WAIT（等待 2MSL），然后发送 ACK 给服务端 ack = 1 + 1
- 服务端收到之后进入 CLOSED 状态

客户端这个时候还需要等待两次 MSL 之后，如果没有收到服务端的重发请求，就表明 ACK 成功到达，挥手结束，客户端变为 CLOSED 状态，否则进行 ACK 重发

为什么需要等待 2MSL（Maximum Segement Lifetime）：

因为如果不等待的话，如果服务端还有很多数据包要给客户端发，且此时客户端端口被新应用占据，那么就会接收到无用的数据包，造成数据包混乱，所以说最保险的方法就是等服务器发来的数据包都死翘翘了再启动新应用。

- 1个 MSL 保证四次挥手中主动关闭方最后的 ACK 报文能最终到达对端
- 1个 MSL 保证对端没有收到 ACK 那么进行重传的 FIN 报文能够到达

为什么是四次而不是三次?

如果是三次的话，那么服务端的 ACK 和 FIN 合成一个挥手，那么长时间的延迟可能让 TCP 一位 FIN 没有达到服务器端，然后让客户的不断的重发 FIN

8. 在交互过程中如果数据传送完了，还不想断开连接怎么办，怎么维持?

在 HTTP 中响应体的 Connection 字段指定为 keep-alive

9. 你对 TCP 滑动窗口有了解嘛?

在 TCP 链接中，对于发送端和接收端而言，TCP 需要把发送的数据放到**发送缓存区**, 将接收的数据放到**接收缓存区**。而经常会存在发送端发送过多，而接收端无法消化的情况，所以就需要流量控制，就是在通过接收缓存区的大小，控制发送端的发送。如果对方的接收缓存区满了，就不能再继续发送了。而这种流量控制的过程就需要在发送端维护一个发送窗口，在接收端维持一个接收窗口。

TCP 滑动窗口分为两种: **发送窗口**和**接收窗口**。

10. WebSocket与Ajax的区别

本质不同

Ajax 即异步 JavaScript 和 XML，是一种创建交互式网页的应用的网页开发技术

websocket 是 HTML5 的一种新协议，实现了浏览器和服务器的实时通信

生命周期不同:

- websocket 是长连接，会话一直保持
- ajax 发送接收之后就会断开

适用范围:

- websocket 用于前后端实时交互数据
- ajax 非实时

发起人:

- AJAX 客户端发起
- WebSocket 服务器端和客户端相互推送

11. 了解 WebSocket 嘛?

长轮询和短轮询，WebSocket 是长轮询。

具体比如在一个电商场景，商品的库存可能会变化，所以需要及时反映给用户，所以客户端会不停的发请求，然后服务器端会不停的去查变化，不管变不变，都返回，这个是短轮询。

而长轮询则表现为如果没有变，就不返回，而是等待变或者超时（一般是十几秒）才返回，如果没有返回，客户端也不需要一直发请求，所以减少了双方的压力。

12. HTTP 如何实现长连接？在什么时候会超时？

通过在头部（请求和响应头）设置 Connection: keep-alive，HTTP1.0协议支持，但是默认关闭，从HTTP1.1协议以后，连接默认都是长连接

- HTTP 一般会有 httpd 守护进程，里面可以设置 keep-alive timeout，当 tcp 链接闲置超过这个时间就会关闭，也可以在 HTTP 的 header 里面设置超时时间
- TCP 的 keep-alive 包含三个参数，支持在系统内核的 net.ipv4 里面设置：当 TCP 链接之后，闲置了 tcp_keepalive_time，则会发生侦测包，如果没有收到对方的 ACK，那么会每隔 tcp_keepalive_intvl 再发一次，直到发送了 tcp_keepalive_probes，就会丢弃该链接。
    - tcp_keepalive_intvl = 15
    - tcp_keepalive_probes = 5
    - tcp_keepalive_time = 1800

实际上 HTTP 没有长短链接，只有 TCP 有，TCP 长连接可以复用一个 TCP 链接来发起多次 HTTP 请求，这样可以减少资源消耗，比如一次请求 HTML，可能还需要请求后续的 JS/CSS/图片等

13. Fetch API与传统Request的区别

- fetch 符合关注点分离，使用 Promise，API 更加丰富，支持 Async/Await
- 语意简单，更加语意化
- 可以使用 isomorphic-fetch ，同构方便

14. POST一般可以发送什么类型的文件，数据处理的问题

- 文本、图片、视频、音频等都可以
- text/image/audio/ 或 application/json 等

15. TCP 如何保证有效传输及拥塞控制原理。

- tcp 是面向连接的、可靠的、传输层通信协议

可靠体现在：有状态、可控制

- 有状态是指 TCP 会确认发送了哪些报文，接收方受到了哪些报文，哪些没有收到，保证数据包按序到达，不允许有差错
- 可控制的是指，如果出现丢包或者网络状况不佳，则会跳转自己的行为，减少发送的速度或者重发

所以上面能保证数据包的有效传输。

拥塞控制原理

原因是有可能整个网络环境特别差，容易丢包，那么发送端就应该注意了。

主要用三种方法：

- 慢启动阈值 + 拥塞避免
- 快速重传
- 快速回复

慢启动阈值 + 拥塞避免

对于拥塞控制来说，TCP 主要维护两个核心状态：

- 拥塞窗口（cwnd）
- 慢启动阈值（ssthresh）

在发送端使用拥塞窗口来控制发送窗口的大小。

然后采用一种比较保守的慢启动算法来慢慢适应这个网络，在开始传输的一段时间，发送端和接收端会首先通过三次握手建立连接，确定各自接收窗口大小，然后初始化双方的拥塞窗口，接着每经过一轮 RTT（收发时延），拥塞窗口大小翻倍，直到达到慢启动阈值。

然后开始进行拥塞避免，拥塞避免具体的做法就是之前每一轮 RTT，拥塞窗口翻倍，现在每一轮就加一个。

快速重传

在 TCP 传输过程中，如果发生了丢包，接收端就会发送之前重复 ACK，比如 第 5 个包丢了，6、7 达到，然后接收端会为 5，6，7 都发送第四个包的 ACK，这个时候发送端受到了 3 个重复的 ACK，意识到丢包了，就会马上进行重传，而不用等到 RTO （超时重传的时间）

选择性重传：报文首部可选性中加入 SACK 属性，通过 left edge 和 right edge 标志那些包到了，然后重传没到的包

快速恢复

如果发送端收到了 3 个重复的 ACK，发现了丢包，觉得现在的网络状况已经进入拥塞状态了，那么就会进入快速恢复阶段：

- 会将拥塞阈值降低为 拥塞窗口的一半
- 然后拥塞窗口大小变为拥塞阈值
- 接着 拥塞窗口再进行线性增加，以适应网络状况

16. http知道嘛？哪一层的协议？（应用层）

- 灵活可扩展，除了规定空格分隔单词，换行分隔字段以外，其他都没有限制，不仅仅可以传输文本，还可以传输图片、视频等任意资源
- 可靠传输，基于 TCP/IP 所以继承了这一特性
- 请求-应答，有来有回
- 无状态，每次 HTTP 请求都是独立的，无关的、默认不需要保存上下文信息

缺点：

- 明文传输不安全
- 复用一个 TCP 链接，会发生对头拥塞
- 无状态在长连接场景中，需要保存大量上下文，以避免传输大量重复的信息

17. OSI七层模型和TCP/IP四层模型

- 应用层
- 表示层
- 会话层
- 传输层
- 网络层
- 数据链路层
- 物理层

TCP/IP 四层概念：

- 应用层：应用层、表示层、会话层：HTTP
- 传输层：传输层：TCP/UDP
- 网络层：网络层：IP
- 数据链路层：数据链路层、物理层

18. TCP 协议怎么保证可靠的，UDP 为什么不可靠?

- TCP 是面向连接的、可靠的、传输层通信协议
- UDP 是无连接的传输层通信协议，继承 IP 特性,基于数据报

为什么 TCP 可靠？TCP 的可靠性体现在有状态和控制

- 会精准记录那些数据发送了，那些数据被对方接收了，那些没有被接收，而且保证数据包按序到达，不允许半点差错，这就是有状态
- 当意识到丢包了或者网络环境不佳，TCP 会根据具体情况调整自己的行为，控制自己的发送速度或者重发，这是可控制的

反之 UDP 就是无状态的和不可控制的

19. HTTP 2 改进

改进性能：

- 头部压缩
- 多路信道复用
- Server Push

20. DDOS 攻击

1）什么是 DDOS 攻击

分布式拒绝服务攻击(Distributed denial of service attack)

向目标系统同时提出数量庞大的服务请求。

2）DDOS 攻击方式

- 通过使网络过载来干扰甚至阻断正常的网络通讯；
- 通过向服务器提交大量请求，使服务器超负荷；
- 阻断某一用户访问服务器；
- 阻断某服务与特定系统或个人的通讯。

3）如何应对 DDOS 攻击

- 黑名单
- DDOS 清洗：对用户请求数据进行实时监控，及时发现 DOS 攻击等异常流量，在不影响正常业务开展的情况下清洗掉这些异常流量。
- CDN 加速
- 高防服务器：高防服务器主要是指能独立硬防御 50Gbps 以上的服务器，能够帮助网站拒绝服务攻击，定期扫描网络主节点

算法

链表

1. 前序遍历判断回文链表

利用链表的后续遍历，使用函数调用栈作为后序遍历栈，来判断是否回文

```
/**
  *
  */
var isPalindrome = function(head) {
    let left = head;
    function traverse(right) {
        if (right == null) return true;
        let res = traverse(right.next);
        res = res && (right.val === left.val);
        left = left.next;
        return res;
    }
    return traverse(head);
};
```

通过 快、慢指针找链表中点，然后反转链表，比较两个链表两侧是否相等，来判断是否是回文链表

```
/**
  *
  */
var isPalindrome = function(head) {
    // 反转 slower 链表
    let right = reverse(findCenter(head));
    let left = head;
    // 开始比较
    while (right != null) {
        if (left.val !== right.val) {
            return false;
        }
        left = left.next;
        right = right.next;
    }
    return true;
}
function findCenter(head) {
    let slower = head, faster = head;
```

```
    while (faster && faster.next != null) {
        slower = slower.next;
        faster = faster.next.next;
    }
    // 如果 faster 不等于 null，说明是奇数个，slower 再移动一格
    if (faster != null) {
        slower = slower.next;
    }
    return slower;
}
function reverse(head) {
    let prev = null, cur = head, nxt = head;
    while (cur != null) {
        nxt = cur.next;
        cur.next = prev;
        prev = cur;
        cur = nxt;
    }
    return prev;
}
```

2. 反转链表

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var reverseList = function(head) {
    if (head == null || head.next == null) return head;
    let last = reverseList(head.next);
    head.next.next = head;
    head.next = null;
    return last;
};
```

3. 合并K个升序链表

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
```

```
 * @param {ListNode[]} lists
 * @return {ListNode}
 */
var mergeKLists = function(lists) {
    if (lists.length === 0) return null;
    return mergeArr(lists);
};
function mergeArr(lists) {
    if (lists.length <= 1) return lists[0];
    let index = Math.floor(lists.length / 2);
    const left = mergeArr(lists.slice(0, index))
    const right = mergeArr(lists.slice(index));
    return merge(left, right);
}
function merge(l1, l2) {
    if (l1 == null && l2 == null) return null;
    if (l1 != null && l2 == null) return l1;
    if (l1 == null && l2 != null) return l2;
    let newHead = null, head = null;
    while (l1 != null && l2 != null) {
        if (l1.val < l2.val) {
            if (!head) {
                newHead = l1;
                head = l1;
            } else {
                newHead.next = l1;
                newHead = newHead.next;
            }
            l1 = l1.next;
        } else {
            if (!head) {
                newHead = l2;
                head = l2;
            } else {
                newHead.next = l2;
                newHead = newHead.next;
            }
            l2 = l2.next;
        }
    }
    newHead.next = l1 ? l1 : l2;
    return head;
}
```

4. K 个一组翻转链表

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
```

```
 * @param {number} k
 * @return {ListNode}
 */
var reverseKGroup = function(head, k) {
    let a = head, b = head;
    for (let i = 0; i < k; i++) {
        if (b == null) return head;
        b = b.next;
    }
    const newHead = reverse(a, b);
    a.next = reverseKGroup(b, k);
    return newHead;
};
function reverse(a, b) {
    let prev = null, cur = a, nxt = a;
    while (cur != b) {
        nxt = cur.next;
        cur.next = prev;
        prev = cur;
        cur = nxt;
    }
    return prev;
}
```

5. 环形链表

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @return {boolean}
 */
var hasCycle = function(head) {
    if (head == null || head.next == null) return false;
    let slower = head, faster = head;
    while (faster != null && faster.next != null) {
        slower = slower.next;
        faster = faster.next.next;
        if (slower === faster) return true;
    }
    return false;
};
```

6. 排序链表

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var sortList = function(head) {
    if (head == null) return null;
    let newHead = head;
    return mergeSort(head);
};
function mergeSort(head) {
    if (head.next != null) {
        let slower = getCenter(head);
        let nxt = slower.next;
        slower.next = null;
        console.log(head, slower, nxt);
        const left = mergeSort(head);
        const right = mergeSort(nxt);
        head = merge(left, right);
    }
    return head;
}
function merge(left, right) {
    let newHead = null, head = null;
    while (left != null && right != null) {
        if (left.val < right.val) {
            if (!head) {
                newHead = left;
                head = left;
            } else {
                newHead.next = left;
                newHead = newHead.next;
            }
            left = left.next;
        } else {
            if (!head) {
                newHead = right;
                head = right;
            } else {
                newHead.next = right;
                newHead = newHead.next;
            }
            right = right.next;
        }
    }
    newHead.next = left ? left : right;
    return head;
}
function getCenter(head) {
```

```
    let slower = head, faster = head.next;
    while (faster != null && faster.next != null) {
        slower = slower.next;
        faster = faster.next.next;
    }
    return slower;
}
```

7. 相交链表

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} headA
 * @param {ListNode} headB
 * @return {ListNode}
 */
var getIntersectionNode = function(headA, headB) {
    let lastHeadA = null;
    let lastHeadB = null;
    let originHeadA = headA;
    let originHeadB = headB;
    if (!headA || !headB) {
        return null;
    }
    while (true) {
        if (headB == headA) {
            return headB;
        }
        if (headA && headA.next == null) {
            lastHeadA = headA;
            headA = originHeadB;
        } else {
            headA = headA.next;
        }
        if (headB && headB.next == null) {
            lastHeadB = headB
            headB = originHeadA;
        } else {
            headB = headB.next;
        }
        if (lastHeadA && lastHeadB && lastHeadA != lastHeadB) {
            return null;
        }
    }
    return null;
};
```

字符串

1. 【面试真题】最长回文子串【双指针】

```
/**
 * @param {string} s
 * @return {string}
 */
var longestPalindrome = function(s) {
    if (s.length === 1) return s;
    let maxRes = 0, maxStr = '';
    for (let i = 0; i < s.length; i++) {
        let str1 = palindrome(s, i, i);
        let str2 = palindrome(s, i, i + 1);
        if (str1.length > maxRes) {
            maxStr = str1;
            maxRes = str1.length;
        }
        if (str2.length > maxRes) {
            maxStr = str2;
            maxRes = str2.length;
        }
    }
    return maxStr;
};
function palindrome(s, l, r) {
    while (l >= 0 && r < s.length && s[l] === s[r]) {
        l--;
        r++;
    }
    return s.slice(l + 1, r);
}
```

2. 最长公共前缀【双指针】

```
/**
 * @param {string[]} strs
 * @return {string}
 */
var longestCommonPrefix = function(strs) {
    if (strs.length === 0) return "";
    let first = strs[0];
    if (first === "") return "";
    let minLen = Number.MAX_SAFE_INTEGER;
    for (let i = 1; i < strs.length; i++) {
        const len = twoStrLongestCommonPrefix(first, strs[i]);
        minLen = Math.min(len, minLen);
    }
    return first.slice(0, minLen);
};
function twoStrLongestCommonPrefix (s, t) {
    let i = 0, j = 0;
    let cnt = 0;
    while (i < s.length && j < t.length) {
```

```
        console.log(s[i], t[j], cnt)
        if (s[i] === t[j])  {
            cnt++;
        } else {
            return cnt;
        }
        i++;
        j++;
    }
    return cnt;
}
```

3. 无重复字符的最长子串【双指针】

```
/**
 * @param {string} s
 * @return {number}
 */
var lengthOfLongestSubstring = function(s) {
  let window = {};
  let left = 0, right = 0;
  let maxLen = 0, maxStr = '';
  while (right < s.length) {
    let c = s[right];
    right++;
    if (window[c]) window[c]++;
    else window[c] = 1
    while (window[c] > 1) {
      let d = s[left];
      left++;
      window[d]--;
    }
    if (maxLen < right - left) {
      maxLen = right - left;
    }
  }
  return maxLen;
};
```

4. 【面试真题】 最小覆盖子串【滑动窗口】

```
/**
 * @param {string} s
 * @param {string} t
 * @return {string}
 */
var minWindow = function(s, t) {
    let need = {}, window = {};
    for (let c of t) {
        if (!need[c]) need[c] = 1;
        else need[c]++;
    }
```

```
    let left = 0, right = 0;
    let valid = 0, len = Object.keys(need).length;
    let minLen = s.length + 1, minStr = '';
    while (right < s.length) {
        const d = s[right];
        right++;
        if (!window[d]) window[d] = 1;
        else window[d]++;
        if (need[d] && need[d] === window[d]) {
            valid++;
        }
        console.log('left - right', left, right);
        while (valid === len) {
            if (right - left < minLen) {
                minLen = right - left;
                minStr = s.slice(left, right);
            }
            console.lo
            let c = s[left];
            left++;
            window[c]--;
            if (need[c] && window[c] < need[c]) {
                valid--;
            }
        }
    }
    return minStr;
};
```

数组问题

1. 【面试真题】俄罗斯套娃信封问题【排序+最长上升子序列】

```
/**
 * @param {number[][]} envelopes
 * @return {number}
 */
var maxEnvelopes = function(envelopes) {
  if (envelopes.length === 1) return 1;
    envelopes.sort((a, b) => {
        if (a[0] !== b[0]) return a[0] - b[0];
        else return b[1] - a[1];
    });
    let LISArr = [];
    for (let [key, value] of envelopes) {
      LISArr.push(value);
    }
    console.log( LISArr);
    return LIS(LISArr);
};
function LIS(arr) {
  let dp = [];
  let maxAns = 0;
  for (let i = 0; i < arr.length; i++) {
    dp[i] = 1;
```

```
  }
  for (let i = 1; i < arr.length; i++) {
    for (let j = i; j >= 0; j--) {
      if (arr[i] > arr[j]) {
        dp[i] = Math.max(dp[i], dp[j] + 1)
      }
      maxAns = Math.max(maxAns, dp[i]);
    }
  }
  return maxAns;
}
```

2. 最长连续递增序列【快慢指针】

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var findLengthOfLCIS = function(nums) {
    if (nums.length === 0) return 0;
    const n = nums.length;
    let left = 0, right = 1;
    let globalMaxLen = 1, maxLen = 1;
    while (right < n) {
        if (nums[right] > nums[left]) maxLen++;
        else {
            maxLen = 1;
        }
        left++;
        right++;
        globalMaxLen = Math.max(globalMaxLen, maxLen);
    }
    return globalMaxLen;
};
```

3. 最长连续序列 【哈希表】

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var longestConsecutive = function(nums) {
    if (nums.length === 0) return 0;
    const set = new Set(nums);
    const n = nums.length;
    let globalLongest = 1;
    for (let i = 0; i < n; i++) {
        if (!set.has(nums[i] - 1)) {
            let longest = 1;
            let currentNum = nums[i];
            while (set.has(currentNum + 1)) {
                currentNum += 1;
```

```
                longest++;
            }
            globalLongest = Math.max(globalLongest, longest);
        }
    }
    return globalLongest;
};
```

4. 【面试真题】盛最多水的容器【哈希表】

```
/**
 * @param {number[]} height
 * @return {number}
 */
var maxArea = function(height) {
    let n = height.length;
    let left = 0, right = n - 1;
    let maxOpacity = 0;
    while (left < right) {
        let res = Math.min(height[left], height[right]) * (right - left);
        maxOpacity = Math.max(maxOpacity, res);
        if (height[left] < height[right]) left++
        else right--;
    }
    return maxOpacity;
};
```

5. 寻找两个正序数组的中位数【双指针】

```
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number}
 */
var findMedianSortedArrays = function(nums1, nums2) {
    let m = nums1.length, n = nums2.length;
    let i = 0, j = 0;
    let newArr = [];
    while (i < m && j < n) {
        if (nums1[i] < nums2[j]) {
            newArr.push(nums1[i++]);
        } else {
            newArr.push(nums2[j++]);
        }
    }
    newArr = newArr.concat(i < m ? nums1.slice(i) : nums2.slice(j));
    const len = newArr.length;
    console.log(newArr)
    if (len % 2 === 0) {
        return (newArr[len / 2] + newArr[len / 2 - 1]) / 2;
    } else {
        return newArr[Math.floor(len / 2)];
```

```
    }
};
```

6. 删除有序数组中的重复项【快慢指针】

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var removeDuplicates = function(nums) {
  if (nums.length <= 1) return nums.length;
  let lo = 0, hi = 0;
  while (hi < nums.length) {
    while (nums[lo] === nums[hi] && hi < nums.length) hi++;
    if (nums[lo] !== nums[hi] && hi < nums.length) {
      lo++;
      nums[lo] = nums[hi];
    }
    hi++;
  }
  return lo + 1;
};
```

7. 和为K的子数组【哈希表】

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */
var subarraySum = function(nums, k) {
    let cnt = 0;
    let sum0_i = 0, sum0_j = 0;
    let map = new Map();
    map.set(0, 1);
    for (let i = 0; i <= nums.length; i++) {
        sum0_i += nums[i];
        sum0_j = sum0_i - k;
        console.log('map', sum0_j, map.get(sum0_j))
        if (map.has(sum0_j)) {
            cnt += map.get(sum0_j);
        }
        let sumCnt = map.get(sum0_i) || 0;
        map.set(sum0_i, sumCnt + 1);
    }
    return cnt;
};
```

8. nSum问题【哈希表】

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function(nums, target) {
  let map2 = new Map();
  for (let i = 0; i < nums.length; i++) {
    map2.set(nums[i], i);
  }
  for (let i = 0; i < nums.length; i++) {
    if (map2.has(target - nums[i]) && map2.get(target - nums[i]) !== i) return
[i, map2.get(target - nums[i])]
  }
};
```

9. 【面试真题】接雨水【暴力+备忘录优化】

```
/**
 * @param {number[]} height
 * @return {number}
 */
var trap = function(height) {
    let l_max = [], r_max = [];
    let len = height.length;
    let maxCapacity = 0;
    for (let i = 0; i < len; i++) {
        l_max[i] = height[i];
        r_max[i] = height[i];
    }
    for (let i = 1; i < len; i++) {
        l_max[i] = Math.max(l_max[i - 1], height[i]);
    }
    for (let j = len - 2; j >= 0; j--) {
        r_max[j] = Math.max(r_max[j + 1], height[j]);
    }
    for (let i = 0; i < len; i++) {
        maxCapacity += Math.min(l_max[i], r_max[i]) - height[i];
    }
    return maxCapacity;
};
```

10. 跳跃游戏【贪心算法】

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
var canJump = function(nums) {
    let faster = 0;
    for (let i = 0; i < nums.length - 1; i++) {
        faster = Math.max(faster, i + nums[i]);
        if (faster <= i) return false;
    }
    return faster >= nums.length - 1;
};
```

二叉树

1. 二叉树的最近公共祖先

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {TreeNode}
 */
let visited;let parent;
var lowestCommonAncestor = function(root, p, q) {
    visited = new Set();
    parent = new Map();
    dfs(root);
    while (p != null) {
        visited.add(p.val);
        p = parent.get(p.val);
    }
    while (q != null) {
        if (visited.has(q.val)) {
            return q;
        }
        q = parent.get(q.val);
    }
    return null;
};
function dfs(root) {
    if (root.left != null) {
        parent.set(root.left.val, root);
        dfs(root.left);
    }
    if (root.right != null) {
        parent.set(root.right.val, root);
```

```
        dfs(root.right);
    }
}
```

2. 二叉搜索树中的搜索

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @param {number} val
 * @return {TreeNode}
 */
var searchBST = function(root, val) {
    if (root == null) return null;
    if (root.val === val) return root;
    if (root.val > val) {
        return searchBST(root.left, val);
    } else if (root.val < val) {
        return searchBST(root.right, val);
    }
};
```

3. 删除二叉搜索树中的节点

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @param {number} key
 * @return {TreeNode}
 */
var deleteNode = function(root, key) {
    if (root == null) return null;
    if (root.val === key) {
        if (root.left == null && root.right == null) return null;
        if (root.left == null) return root.right;
        if (root.right == null) return root.left;
        if (root.left != null && root.right != null)  {
            let target = getMinTreeMaxNode(root.left);
            root.val = target.val;
            root.left = deleteNode(root.left, target.val);
```

```
        }
    }
    if (root.val < key) {
        root.right = deleteNode(root.right, key);
    } else if (root.val > key) {
        root.left = deleteNode(root.left, key);
    }
    return root;
};
function getMinTreeMaxNode(root) {
    if (root.right == null) return root;
    return getMinTreeMaxNode(root.right);
}
```

4. 完全二叉树的节点个数

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var countNodes = function(root) {
    if (root == null) return 0;
    let l = root, r = root;
    let lh = 0, rh = 0;
    while (l != null) {
      l = l.left;
      lh++;
    }
    while (r != null) {
      r = r.right;
      rh++;
    }
    if (lh === rh) {
      return Math.pow(2, lh) - 1;
    }
    return 1 + countNodes(root.left) + countNodes(root.right);
};
```

5. 二叉树的锯齿形层序遍历

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
```

```
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
let res;
var zigzagLevelOrder = function(root) {
    if (root == null) return [];
    res = [];
    BFS(root, true);
    return res;
};
function BFS(root, inOrder) {
    let arr = [];
    let resItem = [];
    let node;
    let stack1 = new Stack();
    let stack2 = new Stack();
    // 判断交换时机
    let flag;
    stack1.push(root);
    res.push([root.val]);
    inOrder = !inOrder;
    while (!stack1.isEmpty() || !stack2.isEmpty()) {
        if (stack1.isEmpty()) {
            flag = 'stack1';
        } else if (stack2.isEmpty()) {
            flag = 'stack2';
        }
        // 决定取那个栈里面的元素
        if (flag === 'stack2' && !stack1.isEmpty()) node = stack1.pop();
        else if (flag === 'stack1' && !stack2.isEmpty()) node = stack2.pop();
        if (inOrder) {
            if (node.left) {
                if (flag === 'stack1') {
                    stack1.push(node.left);
                } else {
                    stack2.push(node.left);
                }
                resItem.push(node.left.val);
            }
            if (node.right) {
                if (flag === 'stack1') {
                    stack1.push(node.right);
                } else {
                    stack2.push(node.right);
                }
                resItem.push(node.right.val);
            }
        } else {
            if (node.right) {
                if (flag === 'stack1') {
                    stack1.push(node.right);
                } else {
                    stack2.push(node.right);
                }
                resItem.push(node.right.val);
```

```
            }
            if (node.left) {
                if (flag === 'stack1') {
                    stack1.push(node.left);
                } else {
                    stack2.push(node.left);
                }
                resItem.push(node.left.val);
            }
        }
        // 判断下次翻转的时机
        if ((flag === 'stack2' && stack1.isEmpty()) || (flag === 'stack1' &&
stack2.isEmpty())) {
            inOrder = !inOrder;
            // 需要翻转了，就加一轮值
            if (resItem.length > 0) {
                res.push(resItem);
            }
            resItem = [];
        }
    }
}
class Stack {
    constructor() {
        this.count = 0;
        this.items = [];
    }
    push(element) {
        this.items[this.count] = element;
        this.count++;
    }
    pop() {
        if (this.isEmpty()) return undefined;
        const element = this.items[this.count - 1];
        delete this.items[this.count - 1];
        this.count--;
        return element;
    }
    size() {
        return this.count;
    }
    isEmpty() {
        return this.size() === 0;
    }
}
```

排序算法

1. 用最少数量的箭引爆气球

```
/**
 * @param {number[][]} points
 * @return {number}
 */
var findMinArrowShots = function(points) {
```

```
    if (points.length === 0) return 0;
    points.sort((a, b) => a[1] - b[1]);
    let cnt = 1;
    let resArr = [points[0]];
    let curr, last;
    for (let i = 1; i < points.length; i++) {
        curr = points[i];
        last = resArr[resArr.length - 1];
        if (curr[0] > last[1]) {
            resArr.push(curr);
            cnt++;
        }
    }
    return cnt;
};
```

2. 合并区间【排序算法+区间问题】

```
/**
 * @param {number[][]} intervals
 * @return {number[][]}
 */
var merge = function(intervals) {
    if (intervals.length === 0) return [];
    intervals.sort((a, b) => a[0] - b[0]);
    let mergeArr = [intervals[0]];
    let last, curr;
    for (let j = 1; j < intervals.length; j++) {
        last = mergeArr[mergeArr.length - 1];
        curr = intervals[j];
        if (last[1] >= curr[0]) {
            last[1] = Math.max(curr[1], last[1]);
        } else {
            mergeArr.push(curr);
        }
    }
    return mergeArr;
};
```

二分查找

```
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number}
 */
var findMedianSortedArrays = function(nums1, nums2) {
    let m = nums1.length, n = nums2.length;
    let i = 0, j = 0;
    let newArr = [];
    while (i < m && j < n) {
        if (nums1[i] < nums2[j]) {
```

```
            newArr.push(nums1[i++]);
        } else {
            newArr.push(nums2[j++]);
        }
    }
    newArr = newArr.concat(i < m ? nums1.slice(i) : nums2.slice(j));
    const len = newArr.length;
    console.log(newArr)
    if (len % 2 === 0) {
        return (newArr[len / 2] + newArr[len / 2 - 1]) / 2;
    } else {
        return newArr[Math.floor(len / 2)];
    }
};
```

1. 判断子序列【二分查找】

```
/**
 * @param {string} s
 * @param {string} t
 * @return {boolean}
 */
var isSubsequence = function(s, t) {
    let hash = {};
    for (let i = 0; i < t.length; i++) {
        if (!hash[t[i]]) hash[t[i]] = [];
        hash[t[i]].push(i);
    }
    let lastMaxIndex = 0;
    for (let i = 0; i < s.length; i++) {
        if (hash[s[i]]) {
            const index = binarySearch(hash[s[i]], lastMaxIndex);
            console.log('index', index, hash[s[i]]);
            if (index === -1) return false;
            lastMaxIndex = hash[s[i]][index] + 1;
        } else return false;
    }
    return true;
};
function binarySearch(array, targetIndex) {
    let left = 0, right = array.length;
    while (left < right) {
        let mid = left + Math.floor((right - left) / 2);
        if (array[mid] >= targetIndex) {
            right = mid;
        } else if (array[mid] < targetIndex) {
            left = mid + 1;
        }
    }
    if (left >= array.length || array[left] < targetIndex) return -1;
    return left;
}
```

2. 在排序数组中查找元素的第一个和最后一个位置【二分搜索】

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var searchRange = function(nums, target) {
    const left = leftBound(nums, target);
    const right = rightBound(nums, target);
    return [left, right];
};
function leftBound(nums, target) {
    let left = 0;
    let right = nums.length - 1;
    while (left <= right) {
        let mid = Math.floor(left + (right - left) / 2);
        if (nums[mid] === target) {
            right = mid - 1;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        }
    }
    if (left >= nums.length || nums[left] !== target) {
        return -1;
    }
    return left;
}
function rightBound(nums, target) {
    let left = 0;
    let right = nums.length - 1;
    while (left <= right) {
        let mid = Math.floor(left + (right - left) / 2);
        if (nums[mid] === target) {
            left = mid + 1;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        }
    }
    if (right < 0 || nums[right] !== target) {
        return -1;
    }
    return right;
}
```

动态规划

1. 最长递增子序列

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var lengthOfLIS = function(nums) {
    let maxLen = 0, n = nums.length;
    let dp = [];
    for (let i = 0; i < n; i++) {
        dp[i] = 1;
    }
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < i; j++) {
            if (nums[i] > nums[j]) {
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
        }
        maxLen = Math.max(maxLen, dp[i]);
    }
    return maxLen;
};
```

2. 【面试真题】 零钱兑换

```
/**
 * @param {number[]} coins
 * @param {number} amount
 * @return {number}
 */
var coinChange = function(coins, amount) {
  if (amount === 0) return 0;
  let dp = [];
  for (let i = 0; i <= amount; i++) {
    dp[i] = amount + 1;
  }
  dp[0] = 0;
  for (let i = 0; i <= amount; i++) {
    for (let j = 0; j < coins.length; j++) {
      if (i >= coins[j]) {
        dp[i] = Math.min(dp[i - coins[j]] + 1, dp[i])
      }
    }
  }
  return dp[amount] === amount + 1 ? -1 : dp[amount];
};
```

3. 【面试真题】 最长公共子序列

```
/**
 * @param {string} text1
 * @param {string} text2
 * @return {number}
 */
var longestCommonSubsequence = function(text1, text2) {
    let n1 = text1.length, n2 = text2.length;
    let dp = [];
    for (let i = -1; i < n1; i++) {
        dp[i] = [];
        for (let j = -1; j < n2;j++) {
            dp[i][j] = 0;
        }
    }
    for (let i = 0; i < n1; i++) {
        for (let j = 0; j < n2; j++) {
            if (text1[i] === text2[j]) {
                dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - 1] + 1);
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1])
            }
        }
    }
    return dp[n1 - 1][n2 - 1];
};
```

4. 编辑距离

```
/**
 * @param {string} word1
 * @param {string} word2
 * @return {number}
 */
var minDistance = function(word1, word2) {
  let len1 = word1.length, len2 = word2.length;
  let dp = [];
  for (let i = 0; i <= len1; i++) {
    dp[i] = [];
    for (let j = 0; j <= len2; j++) {
      dp[i][j] = 0;
      if (i === 0) {
        dp[i][j] = j;
      }
      if (j === 0) {
        dp[i][j] = i;
      }
    }
  }
  for (let i = 1; i <= len1; i++) {
    for (let j = 1; j <= len2; j++) {
      if (word1[i - 1] === word2[j - 1]) {
        dp[i][j] = dp[i - 1][j - 1];
```

```
      } else {
        dp[i][j] = Math.min(dp[i - 1][j] + 1, dp[i][j - 1] + 1, dp[i - 1][j - 1]
+ 1);
      }
    }
  }
  return dp[len1][len2];
};
```

5. 【面试真题】最长回文子序列

```
/**
 * @param {string} s
 * @return {number}
 */
var longestPalindromeSubseq = function(s) {
    let dp = [];
    for (let i = 0; i < s.length; i++) {
        dp[i] = [];
        for (let j = 0; j < s.length; j++) {
            dp[i][j] = 0;
        }
        dp[i][i] = 1;
    }
    for (let i = s.length - 1; i >= 0; i--) {
        for (let j = i + 1; j < s.length; j++) {
            if (s[i] === s[j]) {
                dp[i][j] = dp[i + 1][j - 1] + 2;
            } else {
                dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[0][s.length - 1];
};
```

6. 【面试真题】最大子序和

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var maxSubArray = function(nums) {
    let maxSum = -Infinity;
    let dp = [], n = nums.length;
    for (let i = -1; i < n; i++) {
        dp[i] = 0;
    }
    for (let i = 0; i < n; i++) {
        dp[i] = Math.max(nums[i], dp[i - 1] + nums[i]);
        maxSum = Math.max(maxSum, dp[i]);
    }
```

```
    return maxSum;
};
```

7. 【面试真题】 买卖股票的最佳时机

```
/**
 * @param {number[]} prices
 * @return {number}
 */
var maxProfit = function(prices) {
  let dp = [];
  for (let i = -1; i < prices.length; i++) {
    dp[i] = []
    for (let j = 0; j <= 1; j++) {
      dp[i][j] = [];
      dp[i][j][0] = 0;
      dp[i][j][1] = 0;
      if (i === -1) {
        dp[i][j][1] = -Infinity;
      }
      if (j === 0) {
        dp[i][j][1] = -Infinity;
      }
      if (j === -1) {
        dp[i][j][1] = -Infinity;
      }
    }
  }
  for (let i = 0; i < prices.length; i++) {
    for (let j = 1; j <= 1; j++) {
      dp[i][j][0] = Math.max(dp[i - 1][j][0], dp[i - 1][j][1] + prices[i]);
      dp[i][j][1] = Math.max(dp[i - 1][j][1], dp[i - 1][j - 1][0] - prices[i]);
    }
  }
  return dp[prices.length - 1][1][0];
};
```

BFS

1. 打开转盘锁

```
/**
 * @param {string[]} deadends
 * @param {string} target
 * @return {number}
 */
var openLock = function(deadends, target) {
  let queue = new Queue();
  let visited = new Set();
  let step = 0;
  queue.push('0000');
  visited.add('0000');
```

```
  while (!queue.isEmpty()) {
    let size = queue.size();
    for (let i = 0; i < size; i++) {
      let str = queue.pop();
      if (deadends.includes(str)) continue;
      if (target === str) {
        return step;
      }
      for (let j = 0; j < 4; j++) {
        let plusStr = plusOne(str, j);
        let minusStr = minusOne(str, j);
        if (!visited.has(plusStr)) {
          queue.push(plusStr);
          visited.add(plusStr)
        }
        if (!visited.has(minusStr)) {
          queue.push(minusStr);
          visited.add(minusStr)
        }
      }
    }
    step++;
  }
  return -1;
};
function plusOne(str, index) {
  let strArr = str.split('');
  if (strArr[index] === '9') {
    strArr[index] = '0'
  } else {
    strArr[index] = (Number(strArr[index]) + 1).toString()
  }
  return strArr.join('');
}
function minusOne(str, index) {
  let strArr = str.split('');
  if (strArr[index] === '0') {
    strArr[index] = '9'
  } else {
    strArr[index] = (Number(strArr[index]) - 1).toString()
  }
  return strArr.join('');
}
class Queue {
  constructor() {
    this.items = [];
    this.count = 0;
    this.lowerCount = 0;
  }
  push(elem) {
    this.items[this.count++] = elem;
  }
  pop() {
    if (this.isEmpty()) {
      return;
    }
    const elem = this.items[this.lowerCount];
    delete this.items[this.lowerCount];
```

```
    this.lowerCount++;
    return elem;
  }
  isEmpty() {
    if (this.size() === 0) return true;
    return false;
  }
  size() {
    return this.count - this.lowerCount;
  }
}
```

2. 二叉树的最小深度

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var minDepth = function(root) {
  if (root == null) return 0;
  let depth = 1;
  let queue = new Queue();
  queue.push(root);
  while (!queue.isEmpty()) {
    let size = queue.size();
    for (let i = 0; i < size; i++) {
      const node = queue.pop();
      if (node.left == null && node.right == null) return depth;
      if (node.left) {
        queue.push(node.left);
      }
      if (node.right) {
        queue.push(node.right);
      }
    }
    depth++;
  }
  return depth;
};
class Queue {
  constructor() {
    this.items = [];
    this.count = 0;
    this.lowerCount = 0;
  }
  push(elem) {
    this.items[this.count++] = elem;
  }
```

```
  pop() {
    if (this.isEmpty()) {
      return;
    }
    const elem = this.items[this.lowerCount];
    delete this.items[this.lowerCount];
    this.lowerCount++;
    return elem;
  }
  isEmpty() {
    if (this.size() === 0) return true;
    return false;
  }
  size() {
    return this.count - this.lowerCount;
  }
}
```

栈

1. 最小栈【栈】

```
/**
 * initialize your data structure here.
 */
var MinStack = function() {
    this.stack = [];
    this.minArr = [];
    this.count = 0;
    this.min = Number.MAX_SAFE_INTEGER;
};
/**
 * @param {number} x
 * @return {void}
 */
MinStack.prototype.push = function(x) {
    this.min = Math.min(this.min, x);
    this.minArr[this.count] = this.min;
    this.stack[this.count] = x;
    this.count++;
};
/**
 * @return {void}
 */
MinStack.prototype.pop = function() {
    const element = this.stack[this.count - 1];
    if (this.count - 2 >= 0) this.min = this.minArr[this.count - 2];
    else  this.min = Number.MAX_SAFE_INTEGER;
    delete this.stack[this.count - 1];
    delete this.minArr[this.count - 1];
    this.count--;
    return element;
};
/**
 * @return {number}
```

```
 */
MinStack.prototype.top = function() {
    if (this.count >= 1) {
        return this.stack[this.count - 1];
    }
    return null;
};
/**
 * @return {number}
 */
MinStack.prototype.getMin = function() {
    const element = this.minArr[this.count - 1];
    return element;
};
/**
 * Your MinStack object will be instantiated and called as such:
 * var obj = new MinStack()
 * obj.push(x)
 * obj.pop()
 * var param_3 = obj.top()
 * var param_4 = obj.getMin()
 */
```

2. 下一个更大元素

```
/**
 * @param {number[]} nums
 * @return {number[]}
 */
var nextGreaterElements = function(nums) {
    let ans = [];
    let stack = new Stack();
    const n = nums.length;
    for (let i = 2 * n - 1; i >= 0; i--) {
        while (!stack.isEmpty() && stack.top() <= nums[i % n]) {
            stack.pop();
        }
        ans[i % n] = stack.isEmpty() ? -1 : stack.top();
        stack.push(nums[i % n]);
    }
    return ans;
};
class Stack {
    constructor() {
        this.count = 0;
        this.items = [];
    }
    top() {
        if (this.isEmpty()) return undefined;
        return this.items[this.count - 1];
    }
    push(element) {
        this.items[this.count] = element;
        this.count++;
    }
```

```
    pop() {
        if (this.isEmpty()) return undefined;
        const element = this.items[this.count - 1];
        delete this.items[this.count - 1];
        this.count--;
        return element;
    }
    isEmpty() {
        return this.size() === 0;
    }
    size() {
        return this.count;
    }
}
```

3. 【面试真题】有效的括号

```
/**
 * @param {string} s
 * @return {boolean}
 */
var isValid = function(s) {
    if (s.length === 0) {
        return true;
    }
    if (s.length % 2 !== 0) {
        return false;
    }
    let map = {
        ')': '(',
        ']': '[',
        '}': '{',
    };
    let left = ['(', '[', '{'];
    let right = [')', ']', '}'];
    let stack = new Stack();
    for (let i = 0; i < s.length; i++) {
        if (!right.includes(s[i])) {
            stack.push(s[i]);
        } else {
            const matchStr = map[s[i]];
            while (!stack.isEmpty()) {
                const element = stack.pop();
                if (left.includes(element) && matchStr !== element)  return
false;
                if (element === matchStr) break;
            }
        }
    }
    return stack.isEmpty();
};
class Stack {
    constructor() {
        this.count = 0;
        this.items = [];
```

```
    }
    push(element) {
        this.items[this.count] = element;
        this.count++;
    }
    pop() {
        if (this.isEmpty()) return undefined;
        const element = this.items[this.count - 1];
        delete this.items[this.count - 1];
        this.count--;
        return element;
    }
    isEmpty() {
        return this.size() === 0;
    }
    size() {
        return this.count;
    }
}
```

4. 简化路径

```
/**
 * @param {string} path
 * @return {string}
 */
var simplifyPath = function(path) {
    let newPath = path.split('/');
    newPath = newPath.filter(item => item !== "");
    const stack = new Stack();
    for (let s of newPath) {
        if (s === '..') stack.pop();
        else if (s !== '.') stack.push(s);
    }
    if (stack.isEmpty()) return '/';
    let str = '';
    while (!stack.isEmpty()) {
        const element = stack.pop();
        str = '/' + element + str;
    }
    return str;
};
function handleBack(stack, tag, num) {
    if (!stack.isEmpty()) return num;
    const element = stack.pop();
    if (element === '..') return handleBack(stack, tag, num + 1);
    else {
        stack.push(element);
        return num;
    }
}
class Stack {
    constructor() {
        this.count = 0;
        this.items = [];
```

```
    }
    push(element) {
        this.items[this.count] = element;
        this.count++;
    }
    pop() {
        if (this.isEmpty()) return undefined;
        const element = this.items[this.count - 1];
        delete this.items[this.count - 1];
        this.count--;
        return element;
    }
    size() {
        return this.count;
    }
    isEmpty() {
        return this.size() === 0;
    }
}
```

DFS

1. 岛屿的最大面积

```
/**
 * @param {number[][]} grid
 * @return {number}
 */
let maxX, maxY;let visited;let globalMaxArea;
var maxAreaOfIsland = function(grid) {
    visited = new Set();
    maxX = grid.length;
    maxY = grid[0].length;
    globalMaxArea = 0;
    for (let i = 0; i < maxX; i++) {
        for (let j = 0; j < maxY; j++) {
            if (grid[i][j] === 1) {
                visited.add(`(${i}, ${j})`);
                globalMaxArea = Math.max(globalMaxArea, dfs(grid, i, j));
            }
            visited.clear();
        }
    }
    return globalMaxArea;
};
function dfs(grid, x, y) {
    let res = 1;
    for (let i = -1; i <= 1; i++) {
        for (let j = -1; j <= 1; j++) {
            if (Math.abs(i) === Math.abs(j)) continue;
            const newX = x + i;
            const newY = y + j;
            if (newX >= maxX || newX < 0 || newY >= maxY || newY < 0) continue;
            if (visited.has(`(${newX}, ${newY})`)) continue;
            visited.add(`(${newX}, ${newY})`);
```

```
            const areaCnt = grid[newX][newY]
            if (areaCnt === 1) {
                const cnt = dfs(grid, newX, newY);
                res += cnt;
            }
        }
    }
    return res;
}
```

2. 相同的树

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {boolean}
 */
var isSameTree = function(p, q) {
    if (p == null && q == null) return true;
    if (p == null || q == null) return false;
    if (p.val !== q.val) return false;
    return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
};js
```

回溯算法

1. N皇后

```
/**
 * @param {number} n
 * @return {string[][]}
 */
let result = [];
var solveNQueens = function(n) {
    result = [];
    let board = [];
    for (let i = 0; i < n; i++) {
      board[i] = [];
      for (let j = 0; j < n; j++) {
        board[i][j] = '.'
      }
    }
    backtrack(0, board, n);
    return result;
};
```

```
function deepClone(board) {
  let res = [];
  for (let i = 0; i < board.length; i++) {
    res.push(board[i].join(''));
  }
  return res;
}
function backtrack(row, board, n) {
    if (row === n) {
      result.push(deepClone(board));
      return;
    }
    for (let j = 0; j < n; j++) {
        if (checkInValid(board, row, j, n)) continue;
        board[row][j] = 'Q';
        backtrack(row + 1, board, n);
        board[row][j] = '.';
      }
}
function checkInValid(board, row, column, n) {
  // 行
  for (let i = 0; i < n; i++) {
    if (board[i][column] === 'Q') return true;
  }
  for (let i = row - 1, j = column + 1; i >= 0 && j < n; i--, j++) {
    if (board[i][j] === 'Q') return true;
  }
  for (let i = row - 1, j = column - 1; i >= 0 && j >= 0; i--, j--) {
    if (board[i][j] === 'Q') return true;
  }
  return false;
}
```

2. 全排列

```
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
let results = [];var permute = function(nums) {
    results = [];
    backtrack(nums, []);
    return results;
};
function backtrack(nums, track) {
    if (nums.length === track.length) {
        results.push(track.slice());
        return;
    }
    for (let i = 0; i < nums.length; i++) {
        if (track.includes(nums[i])) continue;
        track.push(nums[i]);
        backtrack(nums, track);
        track.pop();
    }
```

```
}
```

3. 括号生成

```
/**
 * @param {number} n
 * @return {string[]}
 */
var generateParenthesis = function(n) {
    let validRes = [];
    backtrack(n * 2, validRes, '');
    return validRes;
};
function backtrack(len, validRes, bracket) {
    if (bracket.length === len) {
        if (isValidCombination(bracket)) {
            validRes.push(bracket);
        }
        return;
    }
    for (let str of ['(', ')']) {
        bracket += str;
        backtrack(len, validRes, bracket);
        bracket = bracket.slice(0, bracket.length - 1);
    }
}
function isValidCombination(bracket) {
    let stack = new Stack();
    for (let i = 0; i < bracket.length; i++) {
        const str = bracket[i];
        if (str === '(') {
            stack.push(str);
        } else if (str === ')') {
            const top = stack.pop();
            if (top !== '(') return false;
        }
    }
    return stack.isEmpty();
}
class Stack {
    constructor() {
        this.count = 0;
        this.items = [];
    }
    push(element) {
        this.items[this.count] = element;
        this.count++;
    }
    pop() {
        if (this.isEmpty()) return;
        const element = this.items[this.count - 1];
        delete this.items[this.count - 1];
        this.count--;
        return element;
    }
```

```
    size() {
        return this.count;
    }
    isEmpty() {
        return this.size() === 0;
    }
}
```

4. 复原 IP 地址

```
/**
 * @param {string} s
 * @return {string[]}
 */
var restoreIpAddresses = function(s) {
    if (s.length > 12) return [];
    let res = [];
    const track = [];
    backtrack(s, track, res);
    return res;
};
function backtrack(s, track, res) {
    if (track.length === 4 && s.length === 0) {
        res.push(track.join('.'));
        return;
    }
    let len = s.length >= 3 ? 3 : s.length;
    for (let i = 0; i < len; i++) {
        const c = s.slice(0, i + 1);
        if (parseInt(c) > 255) continue;
        if (i >= 1 &&  parseInt(c) < parseInt((1 + '0'.repeat(i)))) continue;
        track.push(c);
        backtrack(s.slice(i + 1), track, res);
        track.pop();
    }
}
```

5. 子集

```
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var subsets = function(nums) {
    if (nums.length === 0) return [[]];
    let resArr = [];
    backtrack(nums, 0, [], resArr);
    return resArr;
};
function backtrack(nums, index, subArr, resArr) {
    if (Array.isArray(subArr)) {
        resArr.push(subArr.slice());
```

```
    }
    if (index === nums.length) {
        return;
    }
    for (let i = index; i < nums.length; i++) {
        subArr.push(nums[i]);
        backtrack(nums, i + 1, subArr, resArr);
        subArr.pop(nums[i]);
    }
}
```