**西西的前端笔记**

1. **计算机网络**
   1. **哪五层模型，知道7层吗**

物理层：用网线把电脑都连接起来通上电

链路层：规定了电信号的传输方式（0101）等等，通过网卡的mac地址以及arp协议将信号发送到别的电脑上，这里组成了局域网

网络层：引入了IP协议，引入了一套新的ip地址，将通过ip区分，哪些电脑在否在某个局域网内

传输层：之前只是电脑到电脑的通信，我们需要有端口到端口的通信（应用之间的通信）。

        这里引入了udp和tcp协议，通过这些协议把数据发往目的地的某个端口号上的应用协议上去

应用层：每个应用程序都有自己的数据规范，所有需要有不同的协议来规定不同的数据包，就有了很多的应用层协议。比如http

简述下计算机五层：

  我感觉计算机五层就像是这个世界上的两个人要联系到对方，和对方沟通的过程的缩影

  物理层：人与人可以通过当面说话来沟通，那如何与远方的人沟通呢，我们可以规定一种规则，来告诉别人你的想法

          三体里面就有一个描述我印象挺深刻的，如何带领士兵打仗听自己的话，就是通过黑白旗子的排列，那放到物理层就是01信号的组合，高电平和低电平，曼彻斯特编码的规则

  链路层：链路层做的是，文字怎么如何给到别人的手里，我们通过信件，和人名来寄信，寄给谁就喊谁是xxx，等那个        人答应，就把信给他。下次再寄给这个人的时候，我就直接找到他给他就不喊了，喊人就是iarp协议，第一次问所有的人，谁是xxx，知道了是谁之后我就把他的名字和位置记下来，以后就直接找他就好了

  网络层：这里加入了ip协议，

物理层：机器与电的链接，电信号的稳定传输，关心01信号的传输

链路层：电信号的发送方式，机器与机器之间的连接关心以太网帧的传输

网络层：跨地域的离开了交换机之间，机器与机器之间的连接，关心全球任意两台机器之间的传输

传输层：机器的应用与机器的应用之间的连接，即为端口到端口的连接

网络层：关心接受到的数据怎么使用

**TCP/IP，他是包含一系列处理数据通信的协议**

* 1. **HTTP**
  2. **tcp。Udp**

86. 请说出TCP协议的通信模型

```js

tcp模型中，有一个服务端，多个客户端，

怎么区分链接对象是谁呢？就是给谁回邮件呢

这里通过数据包的四元组确定，源ip，port，目的ip，port

服务端是A，他在等朋友找他玩，客户端是BCDE等等其他人，准备找A玩。

A家里只能接受电子邮件接受好友的来访

首先：B会给A发邮件，我是你的好友B，我一会儿想来找你玩，可以吗

A收到邮件会马上进行查看，并且回复，好的，但是这里的A并不能走开，因为万一B没收到怎么办

A发出邮件后会等待B的回复，B会回复：那我十分钟后到，勿回

这里本次的tcp友谊聚会才算是成功的搭上线

注意：A这里也可以回复B说好的，但是准备来他家找他玩的人太多了

既然B马上就来那我也可以不用回复了。因为她知道我在家，并且也知道我知道他马上来找我玩了

我先看看别人的邮件吧，为了省时间和精力，就不会再回复b的邮件了

所以可靠的tcp传输是三次握手

并且BCDE好多人每天都会找他玩，他可以通过邮件接受任何来自好友的来访

```

87. 请说出UDP协议的通信模型

```js

每个udp端口是对等的，不存在服务器和客户端的说法

任何一个端口可以像其他的udp端口发送消息

不局限于只能为某个端口大大送

发送的消息也不能保证会送达

送达后也不会有响应

这里没有连接的概念的

udp仅仅是在ip上面加了一个端口

```

88. 请说出HTTP协议的通信模型

```js

http是请求/响应模型

浏览器会建立一个到服务器的tcp连接，然后发送自己的请求，服务器响应，结束

dns也是请求响应模型

ntp协议也是（时间的协议）

    401：未授权，未登录

    404：未找到，not found

    强制缓存的响应头：cache-control（也能使请求头也能使响应头）

    age

    expires

    这三个都是缓存的响应头

    当用户强制刷新，有清除缓存

    http缓存是面试的重中之重

    请求头和响应头，自己也可以添加，非常容易扩 展，也需要服务器认可

    http协议本身就是文本协议，

    user-agent：浏览器的标识，js通过navigator.useragent读出这个内容，表明自己的身份，

    referer:此次页面正在被哪个页面使用，可以用来做防盗链

    防盗链：你自己的页面上使用了自己的服务器上的图片，那么久可以返回

    但是，如果别人的网页上访问了自己服务器上的图片，就不会给返回，因为referer不是自己旗下的网站

    content-security-policy，限制这个浏览器上的js资源是否能被其他的网站加载

    也可以禁止，简称csp

    跨站请求伪造，CSRF，原因在于就算是跨域请求，但是也会发出去，服务器也会处理

    如何防范呢

    通过referer判断是否是自己的网站的请求

    多加一个传输字段，设置请求数据是随时变化的，就是增加一个token令牌/钥匙，请求数据时必须加上

    CORS:未解决跨域问题出现的,有下面几个头

    access-control-allow-origin//对于某个域的请求同意，大多为\*

    access-control-allow-methods//接受的请求方式

    access-control-allow-headers

    access-control-max-age//多少时间内不需要再次发送预检请求

    注意：什么时候有预检请求，什么时候没有

    注意：知道这四个cors的头的意思

    在很多网站上面，请求响应头是可以自己添加的

    content-length和keep-alive一起使用可以实现pipe line

    还有可以做连接复用

    pipe-alive表示tcp链接不要断开

    同时也需要响应头的长度发出来

    跨域的时候，浏览器会发一个预检请求，查看该请求是否能够请求权限

    预检请求是OPTIONS的请求方式，带着referer和host给服务请发送请求

    那么如果能够跨域请求的话就返回一个access-control-allow-origin：\*

    并且可以带着age，表示多少时间内对于该请求都不需要再次发送预检请求，

    但是并不是所有的跨域请求都会触发预检请求

    预检请求：preflight request

1. **Html+css**
   1. **盒模型**

**## 盒模型**

      一个盒子有四个部分，margin，border，padding，content

      margin可以为负数，盒子就会往负数的方向溢出，margin的部分是不可以交互的

      border，padding，content三个内容的宽度，为border-box的width

      现在浏览器的width为content的宽度

      但是也可以改变为border-box的设置：box-sizing:border-box

* 1. **布局**

**## 浮动 float**

      浮动重点：

        块级元素感受不到浮动元素

        行内元素能够感知浮动元素

      关于浮动的自己理解：

      当设置了浮动之后需要这样理解，浏览器遇到浮动的元素的时候，先不会加上浮动的效果

      等到元素的位置都摆放完成之后，我在按照浮动的效果摆放那些有浮动样式的元素

      那么浮动的时候是按照什么标准摆放呢

        如果原来的元素是行内元素，就会将其变为块级元素，规则同块级元素

        如果本来就是块级元素，就按照包含块的盒子的边界来摆放，left就往最左边放，right就往最右边放，不会保留原来的位置，也不会挤掉最左右的元素的位置，而是浮动在其上方，会挡住下下面的元素

      如果是多个浮动元素，多个浮动元素也会形成一个类似文档流的情况

      但是依旧是那个原理，按照原来的常规流摆放，然后在依次一个一个的摆放浮动元素，摆放一次看一次布局，该根据这种布局继续摆放

      遮挡元素的时候也有讲究，行内元素会跟在浮动元素的后面，不会挡住行内元素

      但是浮动元素却不会跟在块级元素的后面，形成两层布局流，从而挡住块级元素

* 1. **定位**

**## 定位**

      1. position：

        1. fixed

          脱离常规流，表现为会挡住常规流的内容

        2. relative

          这个样式的设置是当出现relative及其相关的时候会先放好元素，再根据设置的上下左右去移动元素的位置

        3. absolute

          脱离常规流，但是定位的位置是相对的，相对于距离自己最近的position值不为static的元素的padding-box

          如果都没有，则会相对于整个窗口不是body，也不是html，而是这个窗口的边界

        4. static

          这个值为默认值，不设置定位，元素按照原本的方式摆放

        5. sticky

          这个元素综合了上面的其他定位元素

          该元素的定位方式会根据自己是否还在自己的包含块里面来决定

          页面滚动的时候，元素会跟着包含块一起移动

          当该元素的边界接触到窗口的边界的时候有两种情况

            1. 该元素因为还在包含块内部，包含块还在页面之中，元素不会再移动，相当于fixed

            2. 该元素即将离开自己的包含块的时候，会跟着包含块一起移动相当于relative和absolute

          所以决定该元素的定位方式的因素有窗口的边界，包含块的边界和自己的边界

        使用好sticky可以做出非常高级的吸顶效果

      2. 定位元素覆盖效果

        定位元素会挡住常规流元素

        但是两个定位元素的遮挡是取决与出现的早晚，后出现的会挡住先出现的元素

        当自己的父元素也是定位元素的时候，无法通过z-index改变遮挡效果

**## 回流（reflow/relayout）与重绘(repaint)**

      回流：页面的样式变化涉及到重新计算布局（计算量大，会导致慢）

      重绘：页面样式的改变不涉及布局的计算，比如只改变颜色背景阴影等不影响布局效果的样式

      应该减少回流，尽量使用重绘实现效果

**## BFC bfc作为一种为盒子布局准备的一种渲染规则，让盒子之间互不影响**

      比如，盒子之间会存在margin合并的问题，会改变开发者本来设置的想好的样式

      那么我们可以通过触发盒子的bfc来防止margin合并的问题

      bfc的用处：

        防止margin合并,可以将盒子包裹在不同的bfc元素中

        防止浮动的元素会跑到父元素的border-box的中间去

        用于让元素不被浮动元素遮挡

      实际理解

        触发了bfc的元素可以形成自己的小窗口，不被外界的浮动，margin影响到元素内部，是一个隔离区

      常规的块级元素会包着子元素的border-box，父子元素之间的margin会合并

      bfc元素会包裹着子元素的margin-box,里面的元素怎么动，都不会跑出去

      即为，里面不会影响外面，外面不会影响里面

      那盒子怎么出发bfc渲染模式呢？

      可以设置overflow不为visible 以外的值

      display：flow-root可以直接设置元素为bfc元素

      当一个盒子为bfc盒子的时候，只会让其盒子的包裹其子元素的最上和最下的margin，子盒子之间的margin合并并不会消失

**## 移动端知识点学习**

       移动端布局有好几种方式，但是每种方式都有优势和劣势

       1. 直接用vm，100vm对应适口全部的宽度

       2. 用meta标签的viewport内容，width为假定的窗口宽度，直接基于这个宽度设置即可

       3. 用font-size和rem

        rem继承的是font-size，可以用这样的比例来动态的适应页面的大小

        视觉稿的宽度为X px

        通过js实时获取的页面宽度的Y px

        对于Y的1rem也要对应X的1px

        所以我们先得出，Y的几个px对应X的1px(假设为z)

        X :1px =Y :zpx

        zpx  = Y/X

        那么这个zpx 可以设置到font-size上去，通过rem为统一单位

        但是font-size又不能小于12，所以zpx\*100 为font-size

        那么放大了一百倍，在使用rem的时候，我们应该将原来的rem除上100再使用

        比如：视觉稿1000px，真实页面大小为400px，

        那么可以得知，font-size可以为4/10\*100=25px

        那么1em为25px，为视觉稿里面的100px

        所以在使用的时候直接除上100，即为对应视觉稿的1px

        视觉稿里面的50px可以表示为，0.5rem

**## flex布局学习**

      display:flex/inline-flex

      flex布局默认情况下，水平排列无间隙，并且margin合并不会发生在flex布局里面

      设置为flex的元素为flex-container，子元素为flex-item

      关于设置flex布局的相关属性有很多

      其中有一些对于容器元素 (控制整体)

      有一些对于子元素 (控制个体)

      1. 对于容器元素

        1. flex-dirction

          - cloumn 从上至下排列

          - cloumn-reverse 从下至上排列

          - row 从左到右排列

          - row-reverse 从右到左排列

        2. flex-wrap 摆放元素的时候区域不够，是否换行展示

          - nowwrap 不会换行，即使内容溢出

          - wrap 内容不够的时候向下换行

          - wrap-reverse 内容不够的时候向上换行，即为视觉上是从下到上摆放

        3. flex-flow ：a , b

          a表示的是flex-dirction的值

          b表示的flex-wrap的值

**\*\*以下几个样式都是在不改变元素的本来的大小的情况下发生\*\***

        4. justify-content 水平方向子元素的**\*\*对齐方式和分布方式\*\***

          这种布局可以看成是文字的对其方式，容易理解

          - flex-start 默认方式，左对齐

          - flex-end 右对齐

          - center 居中对齐

          - space-around 每个元素的周围都保持一样的距离（可以看成一样的margin），保证元素有序摆放

          - space-between 每个元素之间都是一样的距离（仅仅是元素和元素之间，元素和容器的边界不会考虑）

          - space-evenly 子元素摆放的时候让所有产生的空袭都是一样的大小（宽度）

        6. align-content 和justify-content相对对应，决定了垂直方向上元素的**\*\*分布方式\*\***

          - stretch 默认值，每一行子元素都默认拉伸，至一样的高度均等铺满容器

          - flex-start 顶部对齐 同上justify-content

          - flex-end 底部对齐 同上justify-content

          - center 居中对齐 同上justify-content

          - space-around 同上justify-content

          - space-between 同上justify-content

          - space-evenly 同上justify-content

        5. align-items 垂直方向上面元素的**\*\*对齐方式\*\***，是一起对齐顶部对齐还是对齐底部对齐呢

          假设上面的例子，竖起来

          - stretch 默认值，没有高度的时候会拉升至同样的高度，有高度的时候，不做任何修改

          - center 居中对齐

          - baseline 基线对齐方向（不会直接吸底显示）

          - flex-start 顶部对齐

          - flex-end 底部对齐

      2. 对于子元素

        1. order

          - 属性值为数字，数字越小，元素越向前摆放，但是不会改变html文档中的位置

        2. flex-grow 属性值为数字，当有多的剩余空间的时候，怎么分配空间给每个元素

          数字也有区别，多个元素的改值和大于等于1和小于1，但是都是等比例分割

          这里为了理解，可以想象成

          > 当元素遇到这个属性的时候并不会马上生效，浏览器会记住，当该层元素都走完了，

          > 会按照每个元素的改值直接变成比例,比如2334，把空间分为12份，按比例分出来

          > 再比如是 0.2 0.4 0.1 因为和不超过1，所以和为0.7，只会将剩余空间的70%分出来

          > 再按照比例分割

          - 大于等于1

            直接按照属性值之和分割

          - 小于1

            将比例之和的空间占比拿出来再分割

          也不知道说明白没有，我怕我以后再就看不懂了呜呜呜

        3. flex-shrink 属性值为数字，当空间不够又不会换行的时候，进行等比例缩放

          属性值之和大于等于1 和 小于1 等同于flex-grow

        4. flex-basis 不是很明白

        5. flex:a , b , c

          - a: flex-grow

          - b:flex-shrink

          - c:flex-basis

        6. align-self 其实是align-items的同样意思，只是这个只针对于单个添加这个属性的元素

          - auto 继承容器的align-items属性

          - stretch 同

          - center 同

          - baseline 同

          - flex-start 同

          - flex-end 同

**## 块级元素和行内元素**

    可以这样理解，块级元素它的兄弟元素在其上下显示，行内元素其兄弟元素在其左右显示

    css的display属性也能够改变元素的这种性质

**### 伪类选择器（超级多，只列出比较常用的）**

    可以用伪类实现多个元素的关联样式

    以:开头

    a标签的伪类：

      1. link：选择没有被访问过的a标签

      2. visited：选择访问过的a标签

      3. focus， hover，active，应该按照这个顺序写样式，才会真的达到用户想要的效果

    位置相关的伪类，本身匹配作为子节点的顺序，但是在使用的时候加不加空格，效果就不一样了

      4. :first-child

      5. :last:child

      6. :nth-child(n)

      7. :nth-child(odd)奇数个

      8. :nth-child(even)偶数个

      9. :nth-last-child(n)

          n为任意值，并且条件可以叠加

          p:nth-child(n+3):nth-child(odd)

      10. :not(:first-child),取反，里面只有一个单一的条件

**## 选择器优先级**

    首先要搞懂几个概念，内联样式，id，标签选择器，类，属性，伪类选择器，标签选择器和伪元素

    其中伪类选择器和伪元素比较容易搞混

    其区别在于：

      伪类是用于选择，第几个，是否点击过等等

      伪元素一定是用于在该元素的某个位置添加某个元素

    那么再来说一下优先级的比较方式

    我自己的理解：有一个数组，有abcd四个位置（对应0123位置）

    当一个选择器出现了某个选择器的时候，对应的位置加一

    比较的时候，从前往后看，即为从0位置开始比较大小，数值相同往后比较

    数值不同，大的数的选择器的优先级高

    若是数组数字全相同，只能比较出现的先后顺序，后面的覆盖前面的

    那么哪些选择器对应哪个位置的数据呢？

    a:内联样式

    b:id选择器

    c:类，属性，伪类

    d:标签选择器，伪元素

    出现一个加一，

    important一定是最大的，有important不要比较优先级，一定不会被覆盖

    0014

    0022

    这里就算是上面的数比下面的数据大，但是，高位置的数据，2比1大，所以还是以1为主

**## vw/vh**

    显示网页的区域叫做视口（不包括浏览器区域）

    1vw表示视口宽度的百分之一

    1vh表示视口高度的百分之一

    vmax,vmin是表示vw和vh中较大者的值和较小者的值

**## calc**

    仅仅表示用来计算

    比如怎么计算动态的计算不同的电脑网页上面的滚动条宽度

    calc(100vw-100%),前者是视口宽度包含滚动条，后者是页面的宽度不包含滚动条

    这样就能计算出滚动条的宽度了

    比如当一个元素需要居中显示的时候但是同时滚动条也是动态的显示

    这样为了让元素不会左右摇摆，可以这样计算，让窗口左边的paddingleft动态的设置为滚动条宽度

    运算符之间记得一定要加空格

**## 角度（旋转）单位**

    有三种角度单位:

      1. degree 直接在前面加上角度即可  45deg

      2. radian 不常用，弧度制，3.14rad为180deg

      3. turn   1turn为一圈，360度，为360deg，6.28redian

    如何使用：

      transform: ratote(39deg)

      transform: ratote(3.14rad)

      transform: ratote(1 turn)

      transform: ratote(calc(3.14rad/2))也可以加上计算

**## font-family**

      注意这个可以接受多个字体名称，当尝试第一个，发现有文字没有匹配到相应的字体的时候，

      会挨个尝试后面的字体样式，可以自动设置中英文字体不一样

**## 字体自重**

      有100-900 但是取决于字体，有的字体有9钟，有的只有三种

      比如说微软雅黑，他只有三种，在一定的范围内改变数值是不会改变字体的宽度的

      这种粗细不是浏览器计算而来，而是字体本身的设计

**## 字体的百分比是会继承的**

      使用百分比的时候，每次嵌套都会乘上百分比，所以可能会越来越小或者越来越大

**## text-indent**

      可以用于文字的前面的空白，不可显示文字，

      2em 差不多是缩进两个汉字大小的空间

      也可以用px，当设置为负数的时候，可以用于隐藏文字，但是至少会留住一行

**## text-align**

      文字的水平对齐方式

      有一个justfiy是可以设置为一行文字两端对齐的，但是不会对齐最后一行

      并且，这个参数不会改变文字的行序，本来是哪一行，就不会改变，只会改变这一行的每个单词的间距为一定值

**## line-height**

      表示文字的行高，可以用于高度居中，写法有四种方式

      line-height:12px /1.2 /2.5em/ 250%，后三种的意思都差不多，但是在样式的继承上面有差异

      第一种就是直接设置

      第二种是按照父元素的倍数，该值1.2会直接继承给其子元素

      第三种是按照当前文字的倍数计算

      第四种也是计算父元素的值的倍数，但是区别在于继承给其子元素的时候会计算出具体的值

      第四种一直使用百分比会导致，高分比计算出的值越来越小，

**## vertical-align 适用于内联元素**

      该样式只是用来设置在垂直方向上的位置，使该元素相对于父元素的对齐方式

      1. baseline 这个属性表示文字符号的一条线，即为英文四线谱上面的第三条线，作为中间行的下线

      剩下有非常多的其他属性值，不一一列举

**##  常规流**

      常规流就是大家常说的文档流，实际应该叫做常规流，常规流不包括浮动或者定位元素

      常规流是指：页面的内容，从上到下，从左到右依次排序，当插入一个元素时，会自动把后面的元素往后挪

1. **JavaScript**
   1. **原生js**
      1. **闭包**
      2. **原型和原型链**

**# 原型与原型链**

**## 原型**

说起原型就要先说起工厂模式

在js中没有类的概念，但是我们可以通过函数返回一个对象来实现相似的一个类的定义

从工厂模式->构造函数模式->原型模式->构造函数和原型组合模式都是对特定类型的对象的封装

1. 工厂模式：

即为用函数来封装以特定接口创建对象的细节，但是他有缺点，没有办法通过对象识别该类型

构造函数模式：把属性和方法都挂载构造函数返回的this上面，只能通过new调用，构造函数首字母大写，可以通过instanceof来识别对象与原型之间是否有关系

2. 原型模式：

是将原本的构造函数清空，将属性和方法都在外部通过该构造函数的prototype来挂载，所有的属性和方法都可以通过prototype这个属性来访问。所以prototype是一个指针，指向一个对象，而这个对象就是原型对象，里面有所有共享给实例的属性和方法，即为prototype就是实例的原型对象，一般来说，这个原型对象还有一个constructor属性指向自己的构造函数

3. 构造函数和原型的组合模式：

因为如果所有属性和方法都在prototype上面的话，任何一个实例也可以通过原型链来更来原型对象上面的属性，其他所有的实例也会因此更改该属性，所以构造函数里面定义实例的属性，prototype属性来指定共享的所有属性和方法，这样属性在每次new调用将属性继承给实例对象的时候都是独立并且唯一的，都不会在实例对象之间有影响，而方法也可以共享出去，所有实例都调用的一个方法

**## 原型链**

js是通过原型链来实现继承的，原型链的构建通过将一个类型的实例赋值给另一个构造函数的原型而实现的

原型链是作为实现继承的主要方法，其基本的思想就是利用原型让一个引用类型继承另一个引用类型的属性和方法

原型链构成的是实例与原型之间的链条

当原型对象变为另一个对象的实例的时候，就是继承了来自另一个对象的全部属性和方法，

当构造函数创建了一个新对象之后就会在实例的内部产生一个[[Prototype]]指针（即为\_proto\_），指向该实例的构造函数的原型对象

**### 原型链的继承方式**

  还没有写到

**## 比如**

比如工厂，工厂要生产一批碗，一开始工人自己一个一个用泥巴捏碗，又慢又需要大批人手，而且很多碗都打不到使用标准，怎么办呢

找一个最好的师傅，捏一个碗，用这个碗做成模具，大家都用这个模具一起生产，这样就快了很多，把你泥巴放进去碗就成型了。但是供不应求，碗有了更多的标准，大家做出来之后还得加工画画写字上去，更加好看，卖的好，而且还有了不同的分类

通过最开始的那个模具，产生了大一号的碗小一号的碗的各种模具，我还可以通过模具再做一个模具，我也可以自己照着自己的想法做模具。每个碗就是这样来的

但是工厂要正规啊，你的碗不合格怎么办，你的碗卖得好我要表扬怎么办，我怎么哪个碗是哪个模具出品的呢

所以碗和模具被创造出来的时候就增加了一个条形码，通过条形码得到是哪个模具加工雕刻的，又通过那个模具知道那个可以雕刻的磨具是从哪个模具生产过来的，还可以知道是哪个师傅做的，最后都会追溯到最开始那个师傅的第一个碗的模具，大家的碗其实都是从那儿来的，那个才是根本

最开始的碗，是object，那个模具是prototype原型对象，那个条形码是[[Prototype]]指向原型对象的指针，constructor就可以知道是哪个师傅做的碗

重新写prototype,就是悄咪咪换了模具

**## 总结**

原型和原型链一定要知道prototype(是共享)和[[Prototype]](是继承)

每一个函数都会有自己prototype原型对象，存放共享出去的东西，但是原型对象也可以继承，每个原型对象都可以有一个[[Prototype]]指针，指向别的构造函数的原型对象，实际上是别的原型对象的实例

但是都是通过[[Prototype]]去查找每一个原型对象和每一个原型对象继承自哪个原型

原型只是实现了构造函数和实例的两两之间的共享关系

但是原型链才是真的实现了继承，即为让一个原型对象的原型链属性指向另一个原型的对象

相关方法：instanceof，isprototypeOf，hasOwnproperty，getpropertyOf,Object.defineProperty(),

* + 1. **异步**
    2. **事件循环**

  ##  事件执行机制

      eventLoop是计算机的运行机制

      JavaScript就是采用了这种机制，为了解决单线程运行带来的一些问题

      怎么解释调用栈，同步异步，和事件轮询

      一个老板，每天要处理很多事情来运作自己的公司

      他正在打电话处理自己的公示的时候，突然秘书进来，有很重要的事情要当面告诉他，

      老板会和电话那边的人说，你等会我，我这边有点事情。老板会先暂停，解决完秘书的事情再继续电话。

      老板解决秘书的当面问题就是调用栈

      这个就是JavaScript同步执行的效果

      那么JavaScript异步执行的效果是什么

      决定异步处理，是不是也表示，当前情况下不是很紧急，用户不需要马上得到反馈？

      那秘书会给老板第一个小纸条，或者递上一份合同，表示，老板你打完电话处理一下这份合同

      这个就是异步

      那么事件轮询是什么呢

      事件轮询就是，老板在处理事情的时候，秘书一直在递交合同。

      老板是不是会把合同分先后顺序放在自己的办公桌上，等自己处理完当前的事情，再去挨个处理自己的合同

      桌上的所有合同就是任务列表

      这里面的任务，电话是主线程，合同是异步函数，口头事件是同步

      闭包是什么呢？

      ？？？？？

      整个过程代表了事件轮询也就是eventloop

      事件是怎么转起来的

      事件轮询就是主线程不停的从任务列表读取数据的过程

      事件的执行机制就是Event Loop，事件轮询，也是一种js实现异步的方式

      事件分为同步和异步，更加细致的话分为宏任务和微任务

      宏任务按顺序执行，且浏览器在每个宏任务之间渲染页面

      所有微任务也按顺序执行，且在以下场景会立即执行所有微任务

      每个回调之后且js执行栈中为空。

      每个宏任务结束后。

   1. macro-task(宏任务)：包括整体代码script，setTimeout，setInterval

      2. micro-task(微任务)：Promise，process.nextTick

调用栈：函数之间额调用形成的栈结构

在一个函数里面调用另一个函数，就形成了调用栈，只有下一个函数执行结束，才会回来继续执行当前函数

就好比：

  就像是去某a单位办理a证件，去办的时候，工作人员告诉你，需要去地b单位办理b证明才行，

  于是你跑到b单位去办理b证件，但是b的工作人员告诉你，需要去c单位办理c证明才行，

  于是你又跑到c单位去办理c证件，终于你办理好了c证明，拿着c证明去办理b证明，

  拿到了b证明才能去办理a证明，最后才能得到结果即a证件。

  这就是调用栈

事件循环：应该是事件轮询，就是事件的执行环，听的课程很有用，记得画图

JavaScript的事件是在一个环里面运行的，没有事件的时候环里面的小人不会动

当事件发生小人就开始跑

跑动一圈就是一个函数的执行过程

执行的过程中，会有一些异步事件或者浏览器渲染事件，他们会告诉小人，你来帮我处理下事情把

但是小人并不会马上停止自己的代码，而是回复，等我跑完这一圈，当我当前的调用栈结束的时候，我来找你们

那么那些等着他处理的事情就变成了一个任务队列

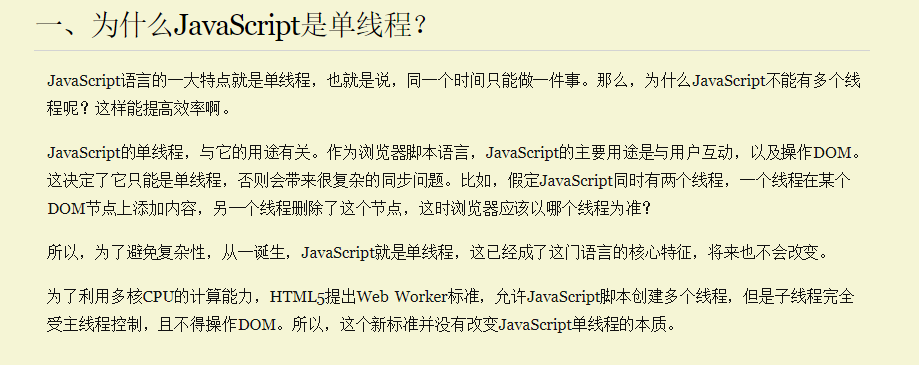
当自己跑完了，就会跑到左边的岔路口去执行异步的函数，执行完之后再接着继续接下来的任务

闭包：闭包就是一个函数A被调用的时候返回了一个函数B，而函数B中使用到了函数A中的变量c，导致在外部可以使用到变量c，这就是一个闭包

同步：你去排队买小空包，只有一个阿姨卖，你就得排队，一个个买，知道轮到你，你得告诉阿姨你买几个，等阿姨一个个帮你装起来，付好钱，离开，才算是买好了小笼包。阿姨一个时间只能做一件事，你必须排队，也必须等待

异步：你去吃海底捞，人特别多，你得排队，服务员会告诉你先留个电话号码取号排队，你可以先去逛街买玩自己的，等排到你了，海底捞的服务员会给你打电话让你快去吃火锅，这样你就吃上火锅了。异步就是，你要做什么，告诉我，我帮你记着，你先做你接下来要做的事情，轮到你了我再叫你。

ps：留电话号码取号叫做申明回调函数，海底捞的服务员给你打电话叫你吃火锅就是执行了回调函数

* + 1. **This**
    2. **Js为什么是单线程**
  1. **ES6**
     1. **Generator**
     2. **Promise**

 ## Promise

    个人理解：

    在看张鑫旭的文章的时候，说过promise一开始本来想命名为futures，意味未来

    其实，翻译为承诺有点误区，个人喜好理解为‘如果’

    你喜欢上一个人，和他表白，有两种可能，拒绝你，答应你

    如果答应你，你会开始计划与她要做的事情，我们去逛街，看电影

    如果拒绝你，你就马上找个酒吧，去买醉

    这个如果，就是提前计划好自己未来的行为

    但是未来有很多可能，一种有结果，一种没有结果

    看见一句话，promise也还是有很多的回调，但是，他让回调更加可控

    promise.all更加像是人对于第二天的所有计划，之后时间到了我才能继续下一个计划

    明日计划：这就是一个promise未来链

    1. 早上九点去爬山

      1.1 爬山完成

      1.2 要是下雨，不能爬山我就起床看个电影

        1.2.1 看电影完成

        1.2.2 直接睡过头，没看成电影，over

    2. 中午睡个午觉

      2.1 睡午觉完成

      2.2 要是隔壁在装修，没睡成那我就直接找老王去

    3. 下午约隔壁老王去打篮球

      3.1 打篮球完成

      3.2 还在下雨，没打成球，那就和老王去逛超市

        3.2.1 和老王去逛超市完成

        3.2.1 老王不在家，over

    4. 晚上约小李吃个饭，约会

      4.1 约会完成

      4.2 小李放我鸽子了，over，没心情干别的了

      也有点像以前看的古装电视剧，里面的大佬死之前会给主角三个锦囊，告诉他

      你其实是皇帝的儿子。但是被抛弃了，你现在要去夺回你太子的位置

      1. 你先去隔壁村找王员外，告诉他你的身份

        1.1 他信了，你就听他的话

        1.2 他要是没信，你就把我的黄色的锦囊，让他带你去找

      promise.race有点像，你去好几家公司面试了，每个hr都告诉你这周末会给你答复

      但是在周末来临之前，你都不知道哪个hr会最先给你答复

      那race就是，你告诉我，周末是哪位hr最先给你答复的，不管是否被录取

      promise.try是可以用于如果一个异步的请求本来就可能会报错

      使用try执行，promise会用catch接住错误

      比如下面的database.users根本就不存在怎么办

      Promise.try(() => database.users.get({id: userId}))

      .then(...)

      .catch(...)

      promise有一个主要的then方法，会为promise注册两个回调函数，

      用来接收这个promise成功返回的值或者是失败的原因

      promise还有一个finally方法的回调函数不接受任何参数，

      和状态无关，是最终一定会执行的代码

      promise有三种状态，

      其中一种状态是可以变为其他两种状态其中之一的，

      而其中两种状态是不可更改的

      而其中的不可变也仅仅指的是

      一个promise最终只能有一种状态，成功，或者失败，

      每一个promise都会返回一个东西，可能是一个值，也可能是一个promise，也可能抛错

      那最后的结果取决于x的值，或者返回的promise的最终状态，也可能是一个中间的错误

      一个promise的最终状态是reslove还是reject是看这个promise内部的代码是否成功运行，成功运行，是resolve，如果发生错误或者抛出错误就是reject

      如果要自己实现一个promise，就应该知道一个promise做了什么，如何使用

      所以我们写一个用例：

      用promise异步的读取

      如果你的Promise的then的回调函数没有return ，那么不能实现then的链式调用。

      因为then会返回一个promise，如果不返回就是undefined，

      then可以接受两个参数，但是建议then，catch这样写，因为更加全面的接受错误

* + 1. **Async/await**

1. **React**
   1. **生命周期函数**

**##### 生命周期；**

1.1：beforeCreate Vue实例初始化前期；

此时根节点以及数据对象都没有初始化；

1.2：created Vue实例初始化完成；

数据对象已经初始化成功，可以访问其中的数据，但DOM对还未生成

1.3 在created和beforeMount之间这个阶段，判断以何种方式来编译模板

2.1：beforeMount 挂载初期

此时的虚拟DOM已经生成，但真实DOM还没有，

2.2：mounted 挂载完成阶段

虚拟DOM经过经过编译生成了HTML，并且已经插入到真实的DOM中

3.1 beforeUpdete 数据变化出发更新之前，此时的视图还没有更新

3.2 update 视图更新完毕

4.1 beforeDestory 实例销毁初期，所有的数据绑定和事件，都即将解绑，都可以调用

4.4 destory 实例销毁后，只是一个DOM没有任何数据绑定，且依旧存在DOM中

* 1. **hooks**

**# Hook笔记**

**## 函数组件**

    没有生命周期函数，没有内部state

**## useState**

    通过useState来定义函数组件内部的state

    var [state1,setState] = useState(0)这个0就是

    var [count,setCount] = useState(0)

    state就是组件的state，setState就是一个函数，通过这个函数来改变前面的state的值

    对于函数组件来说，更新state，就是重新刷新整个函数组件，重新运行

    但是userstate的时候传入的参数只会在第一次运行的时候取传入值，

    后期set之后的状态都是有内存存着的

    所以就算组件重新渲染也不会拿到初始值而是改变之后的值，因为返回的数据改变了

    所以当有多个状态的时候可以有多个usestate。但是也可以在传递初始值的时候给一个所有state的对象

**## useEffect相当于是组件挂载完成和更新完成之后会执行的函数**

    useEffect(()=>{这个函数不接受参数

      这个接受一个函数来处理相应的事物

      这个时候数据什么的都已经可以读取了

      return ()=>{}这里是可以返回一个函数来处理当组件被卸载的时候调用的函数

      这里返回的函数用来清除一些副作用

    })

**##**

* 1. **redux**

**# redux 笔记**

  1. 什么是redux,是干什么的:

    redux是JavaScript的状态容器，提供了可预测化的状态管理

    当你发现你的项目的state已经多到不可控制的时候就会需要redux来帮你有序的管理state

    因为当一个项目有很多的组件很多的state很多的组件依赖的时候，当你更新了某个state，

    你会发现好像引起了其他组件的重新渲染，

    那么那个组件是什么时候，由于什么原因，如何变化都已经不再受到控制

    在项目里面你甚至不会再敢随随便便更新数据，数据剪不断理还乱

    因为state只能通过setstate去改变数据，但是，当出现成千上百个setstate的时候

    你会发现，你可能不会知道为什么要在这里更新一个state,更新了之后会有什么影响，有什么好处

    这就是为什么我们需要redux，也是redux要来帮我们解决的问题，就是明确过程，规范变化

    这个时候你就需要一个redux来帮助管理你的state

    就是把每一个组件的state的数据改变都抽离出来，

    在组件里面我们只会负责改变state，得到新的state，再去更新state

  2. 实现原理是什么:

    当我们需要把一个事情理清楚，就需要把每件事情细化，一桩桩一件件，有条理的记录下来

    redux就是这样的，状态管理被放在一个store对象上面

    她规定了当你需要更新state的时候需要通过store.dispach触发一个action，

    通过action调用一个reducer函数，去处理对应的state，并且返回一个新的state

    这样你就可以通过store.getstate来得到新的state，通过setstae去更新state，更新组件

    action是把数据从应用传到store的荷载，是store的state的唯一来源，action只是用来描述数据的更新

  3. 有什么用:

  4. 类似的东西有吗,以及区别在哪里:

  5. 如何使用

  6. 如何自己实现

  7. react优化

    通过setstate着手去优化react组件更新

    在setstate的时候可能有某些数据没有改变，就可以通知组件不需要更新

    通过shouldComponentUpdate(props,state)来对比前后两次props和state是否有改变，按照需求去判断是否需要更新组件

    但是如果仅仅通过前后的props的对象是否相等来进行判断，就不太合理

    因为有的时候仅仅是对象某个舒心的值发生了改变，但是对象的指向并没有改变

    所以仅仅判断对象是否相等是不够的

    我们需要当对象的属性值一旦发生改变了，就返回一个新的对象

    但是返回一个新的对象又不合理，如果数据很大，就会造成时间开销，希望没有改变的数据还是保留原来的指向，不需要开辟新的空间重新赋值，只是返回了一个看起来是新对象的对象，改变的数据变了，没有改变的数据还是和原来的一样

* 1. **虚拟dom**

**## 虚拟dom的例子**

    好比如真实的房子和施工的图纸

    你根据图纸修好了一栋房子，但是房主不是很满意，让设计师又修改了一部分结构，

    设计师会给你一张新的图纸，说，你按照这个图纸来修吧

    但是你真的会把房子拆了重修吗，当然不会

    你会把前后两个图纸进行对比，到底需要改变哪里的结构

    你就只改变那里的结构就好了

**关于key的必考题**

**## 关于key的必考题，为什么要有key，key的好处是什么**

    在对比图纸的时候，图纸太大了，你要一个细节一个细节的对比就很乱，而且容易出错

    你会发现，对比的时候我们可以不需要专业人士，随便一个人都可以找不同

    那么这个时候，我们可以通过图纸上的每个区域的名字来进行对比，

    我们先对比下大客厅，两个大客厅之间有什么不同？

    没有不同，那我们接着对比下主卧室，通过对比我们发现，主卧室的有一面墙被打通了

    接着对比次卧，书房，每一个房间，然后厕所，

    我们从上往下一个一个对比图纸上每一个能说得上名字的内容

    那么key就是给每一个地方加一个名字，名字必须不同，一样的key会导致你对比失误

    我们把每一个区域变成不同的名字分给不同的人，最后让他们反馈每一个区域都改变了哪些地方

    比如说，在渲染5个li的时候

    我们只是在这5个li中间最前面增加一个li，

    如果每一个li没有唯一的key值

    那么他的diff算法会从头开始对比每一个li，如果在最前面增加一个li，会导致看起来，每一个li都改变了，

    都不一样了，那么我需要重新渲染所有的li

    如果给每一个li都唯一的key值，会发生什么？

    diff在对比的时候会对比所有key值相同的

    我们会发现，key值相同的都没有改变

    只是最前面增加一个key值不同的li

    那么我们只渲染那个新增的li就好了

    这样就大大的减少了浏览器的渲染效率

    这就是key的用处

**关于虚拟dom**

**## 关于vue的dom以及渲染,diff算法**

    浏览器为了更好的用户体验，不会等整个html文档解析完成才去构建render树和计算布局

    而是一边加载一边解析一边渲染

    一个网页展示的步骤：

      html解析构建dom树-> css解析构建StyleRules->构建render树->布局Layout->绘制painting

      这里还要知道重绘和回流的差别

      每一次对dom的操作都会引起整个dom的流程走一遍，代价是巨大的

      所以需要虚拟dom来简化dom的操作，在一定的时间内去更新dom，重新布局，

      那么在那些时间内的所有操作都会放在队列里面等待

      当需要更新的时候，通过前后两次虚拟dom更新，比较不同的地方

      通过高效的diff对比，来判断是否需要删除节点，新增节点还是移动节点还是只是更改text

      而diff的对比采用的是深度优先对比，事实上就像是html代码一样，一行一行对比下来，

      先后对比标签，属性，内容是否一样

      不同的地方会记录到一个对象里面，变成一个数据表

      通过这个表，表示删除哪个节点，移动哪个节点，更改哪个text，再去真正的挂载真实的dom节点

      这个时候key值的出现更加高效的改变了对比的效率

      因为插入一个节点之后，从前往后对比会发现插入的节点之后都是乱掉的，

      后面的节点就会不停的卸载，挂载，那么如果有key值

      key值尽量不要取数组的下标，会导致错误

      对比的时候，key值相同的会进行对比，就会发现我不需要不停的卸载挂载，后面的节点都只是需要移动而已

      而对于需要移动的节点，采用的是最短距离算法来决定如何高效的移动

      为最小编辑距离的动态规划求解

**### js操作真实dom的代价**

    当我们需要更新10个dom节点，当浏览器收到第一个dom节点的更新的信息的时候并不会提前知道后面还有9个dom节点的更新，所以会立即更新第一个dom节点的变化

**## vuex**

      这个是状态管理的模式，

      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。

      简单的理解就是打破了单项数据流，不用再使用emit来绑定父子组件之间的数据传递

      而是把整个需要交换数据的组件引入vuex的概念来统一管理，更改，处理需要的数据

      把数据变成大家共用的

**### 正常的vuex的概念有**

      1. 状态state

      2. 模块module，用来同步改变state中的数据

      3. 访问属性getter，用来计算state的各种属性值

      4. 更改状态mutation

      5. 异步操作逻辑actions

      有个问题就是为什么需要异步的antions之后再去同步的直接更改数据mutations，

      因为vue需要知道数据的状态

      只有antions调用了commit，vue才会知道，数据变化了，不管是异步更新还是演示调用，都没有办法直接知道数据是否更新完成

      但是commit就代表，数据更新了，因为mutation是同步的，是一定是会马上执行的

* 1. **优化**

1. **Vue**
   1. **Api**
2. **Node**
   1. **Node是干嘛用的**
   2. **有哪些模型**
   3. **Express讲一下**
3. **算法**
   1. **排序**
   2. **BST，DST**
4. **Java，sql语言**
5. ***正则***
6. **# 正则笔记**
7. **## 元字符**
8. 元字符是构造正则表达式的基本元素
9. 常用元字符表达式
10. \d   ：0到9的所有数字（一位）
11. \b   ：匹配一个单词的开头或者结束位置
12. \w    : [ 0-9 a-z A-Z \\_ ] ,0到9，a到z，A到Z，还有一个下划线
13. \s     : 匹配空格，tab，回车
14. \D    : \d 取反，即出来0到9的所有字符
15. \W   : \w 取反，
16. \S    : \S取反，除了空格符
17. \i     : 表示匹配的内容，大小写不敏感
18. ^     : 在表达式开头出现，表示从字符串开头开始匹配，在 [ ] 中的开头位置出现
19. 表示后面的匹配内容取反（非）比如：[^01]表示非0或者非1，非开头位置只表示该符号
20. &    : 和^一起出现，表示表达式匹配整个字符串，精确匹配，是否完全整个匹配
21. .     : 除了回车外的所有符号，包括中文
22. **## 重复限定符**
23. +    : 表示前面的匹配内容重复1到多次
24. \*     : 和+类似，但是表示前面的匹配内容重复0到多次
25. ？   : 表示出现0次或者出现1次
26. { 4 }: 中间是任意数字，表示精确重复出现多少次，比如重复出现4次
27. {4,6}: 表示重复出现的次数在4到6之间
28. {5，}: 表示重复出现的次数在5次及以上
29. **## 分组符**
30. ( )   ：括号中的内容将作为一个整体
31. **## 转义符**
32. \      : 后面紧跟的符号将作为一个简单符号匹配
33. 条件或
34. |      ： 表示一种分支条件，匹配|左右的任意一个条件，
35. **## 区间符**
36. [ ]    : 表示匹配括号中任意一个字符 ，和|类似，但是 [ ] 中只能匹配一个符号，比如[ 0-9 a-z A-Z \\_ ]，匹配其中其一而已，
37. 注意g：在正则 / / 的后面加一个g即为//g，就是全局匹配，匹配到字符串中所有符合的子集
38. 如果要匹配任意字符怎么办？可以使用[\d\D]、[\w\W]、[\s\S]和[^]中任何的一个
39. **## 其他笔记**
40. \d：表示一位数字
41. 重复的次数
42. \d+：+表示前面的字符可以出现多次，即多个数字
43. \d\*：\*表示出现0次，1次，多次都可以
44. ？：表示出现0次或者1次
45. {4}：表示精确的出现括号中的次数
46. {1,4}：表示精确的出现1到4次中的任意次数
47. {5,}：表示至少5次或者以上
48. {,5}这个写方法是错误的，在正则里面就仅代表四个字符而已
49. 正则表达式可以用构造函数表示，也可以用 '/' 包裹起来
50. 用构造函数的时候需要注意
51. 在任何符号前面，都可以加一个\，表示后面的符号本身
52. 正则表达式的test是表示，test后面的字符串中是否符合前面表达式中所表达的模式，返回真假
53. 正则表达式中的[]，中括号：在中括号里面的符号只有符号的本身的意思，而没有正则表达式赋予的意义
54. 正则表达式中的 '-' ,中划线：表示划线两边所有的字符
55. 取反的^只能出现在中括号的第一个位置表示取反，其他位置没有这个意思
56. \i 表示大小写不敏感
57. exec也是一个正则测试方法，是正则匹配字符串
58. 没有匹配到会返回一个null
59. 匹配到会返回匹配到的内容
60. 而且返回的内容有个index属性，表示在哪里匹配到的
61. 当括号确定了，返回的数组的长度一定是确定的，
62. 会返回数组，数组是，第一个内容是整个正则表达式，后面的内容是正则表达式中出现的括号的匹配到的内容
63. matches,是字符串匹配正则，也有index属性
64. 返回数组
65. 只会返回匹配到的第一个内容
66. Date类型：表达日期和时间
67. ^ 这个符号在正则表达式中出现在第一个位置，表示匹配从字符串的开头开始
68. 若是正则表达式由^开始，由$结束，那么匹配后面的字符串从开始到结束，即为整个字符串
69. \b是表示单词边界，没有特殊的符号，只是表示一种位置
70. 零宽断言：匹配两个符号之间的一个位置，而不是一个符号
71. 匹配发生的时候，光标不会移动，所以可以对同一个位置执行多个零宽断言,只有当每个断言都成功时才可以
72. 有四种零宽断言
73. (?=expr) 正预测零宽断言，这个位置的右边满足expr
74. (?!expr) 负预测零宽断言，这个位置的右边不满足expr
75. (?<=expr) 正回顾零宽断言，这个位置的左边满足expr（旧版JS不支持）
76. (?<!expr) 负回顾零宽断言，这个位置的左边不满足expr（旧版JS不支持）
77. ^ === /(?<![^])/
78. $ === /(?![^])/
79. \b === /(?<=\w)(?=\W)|(?<=\W)(?=\w)|(?<![^])|(?![^])/
80. 正向先行断言（正前瞻）：
81. 语法：（?=pattern）
82. 作用：匹配pattern表达式的前面内容，不返回本身。
83. 正向后行断言（正后顾）:
84. 语法：（?<=pattern）
85. 作用：匹配pattern表达式的后面的内容，不返回本身。
86. 负向先行断言（负前瞻）
87. 语法：(?!pattern)
88. 作用：匹配非pattern表达式的前面内容，不返回本身。
89. 负向后行断言（负后顾）
90. 语法：(?<!pattern)
91. 作用：匹配非pattern表达式的后面内容，不返回本身。
92. 你那个正则的意思是，匹配 [ 后面所有的内容并且不包括 ] 的内容，你应该再加一个[就不会多一个中括号了
93. **## 深入**
94. 后向引用
95. 贪婪匹配
96. 总是先尝试最多的部分，如果匹配不到再以此减少
97. 非贪婪匹配：总是先尝试少的部分，在表达式前加一个？
98. 作业：
99. 使用从通配符构造一个正则表达式的办法来实现上面的函数,比如：
100. match('fooobarbaz', '\*bar?az')
101. match('fooobarbaz', '???\*baz')
102. match('fooo++++barbaz', '???\*baz')
103. \*代表0个或多个任意字符
104. ?代表1个任意字符
105. 正则的lastIndex属性，会对test，exec，match这几个函数都有影响
106. match,
107. 不考虑正则表达式的lastIndex属性。
108. 当正则表达式有g标志的时候，匹配出所有能够满足整条正则表达式的内容
109. 当没g标志的时候，匹配出第一条能满足的内容，同时把分组捕获到的内容也放入结果数组
110. replace(RegExp|String, String|Function)
111. 两个参数都是字符串：匹配第一次出现的内容为目标内容
112. 首参为正则
113. 次参为字符串，里面的$&,$1,$2是特殊内容，表示匹配到匹配到的内容以及各个分组捕获到的内容
114. 次参为函数，把整个匹配到的内容以及各分组捕获到的内容传给函数做为参数，把函数返回值插入被替换位置
115. 有多少次匹配，函数就会调用多少次
116. re.exec(str)方法
117. 如果re不带g标志，则完全等同于str.match(re)
118. 如果re带有g标志，则从str的re.lastIndex位置开始查找，查找成功后把re.lastIndex置为匹配位置的后一个位置
119. 查找不成功的时候，返回null，把lastIndex置为0
120. str.split(String|RegExp)
121. 当参数是字符串时，按字符串把原字符串拆成数组
122. 当参数为正则时，按正则把原字符串拆成各部分的数组
123. 但是当正则里有捕获分组时，分组捕获到的内容也会出现在结果数组的相应位置
124. onkeyup,当键盘松手的时候，在input中的时候
125. 实现正则匹配的实时高亮效果（难）
126. 作业：输入的字符串没有任何多余的空格
127. 解析json的字符串的函数ParseJSON
128. 就是，实现json.parse的函数或者json.string函数
129. y和g是一样的用法，但是y的意思是严格匹配，只会从当前的位置开始匹配，不会跳过
130. string.raw`shwuw`这里是什么返回什么，用于正则的构造函数，可以自己实现这个函数
131. argument里面有一个raw属性，是得到的字符串不转义的结果
132. **## 扩展**
133. exec是正则表达式的正则对象RegExp的一个方法、
134. 这个方法接受一个字符串，但是返回数据有多重情况，可以从exec入手，自己实现其他的正则的函数
135. /.../.exec(str)
136. 返回的数据里面有几个非常重要的属性
137. 匹配不成功只会返回一个null
138. 若是匹配成功，则有两种情况，正则表达式带g和不带g
139. 有一下几个属性，比较重要
140. 1.index：表示找到的匹配到的字符串的第一个下标
141. 2.input：表示传入的str
142. 3.lastIndex：表示匹配成功一个之后，下次开始匹配的位置（只在有g的情况下有效）
143. 在使用正则的时候也要注意（）的用法，会影响函数返回的数据

***http***

**# http 记录**

**## HTTP是什么**

  HTTP是不保存状态的协议，既无状态协议

  协议本身对于请求或响应之间的通信状态不进行保存，因此连接双方不能知晓对方当前的身份和状态。

  这也是Cookie技术产生的重要原因之一：客户端的状态管理。

  浏览器会根据从服务器端发送的响应报文内 Set-Cookie 首部字段信息自动保持 Cookie。

  而每次客户端发送 HTTP 请求，都会在请求报文中携带 Cookie，作为服务端识别客户端身份状态的标识。

**## 重要概念解读**

  是关于串行连接、持久连接、管道化持久连接、http/2.0多路复用简介

**### 串行连接**

   http是建立在tcp上面的一次请求一次响应模型，每一个响应后都会断开tcp的链接

   故而，每次发起http请求都会重新进行一次tcp连接，这样也太麻烦了吧

   这就是http/1.0的版本，也叫作串行连接或者短连接或者短轮询

**### 持久连接，也叫作长连接长轮询**

   为了解决上面的问题，HTTP/1.1实现并默认了所有的链接都采用持久连接

   即为，客户端和服务端双方都没有提出断开连接的话就会在一定时间内保持tcp的连接

   以保证一段时间内同一个域名下的http请求的复用，这样就减少了多次http请求造成的tcp三次握手和四次挥手的次数

   但是这个连接采用的是拥塞模式，就是只有上次的请求得到了响应才会发起下一次的请求

   如果上次请求的内容还没有返回就只能等着，常说的线头阻塞

   也就是http-alive

**### 管道化持久连接**

   就是解决了线头阻塞，在持久连接的基础上实现了不用等待上次响应返回可以直接发送请求，并安札顺序返回响应

   但是现代浏览器并没有默认开启管道化

   在持久连接的基础上，可以并发请求，而且多次发送不需要上次的等待

**### http/2.0多路复用**

   就是在客户端增加一个序列标识符，客户端可以并发发送请求

   服务端会根据标识符自己排序为不同的请求报文，再并发返回响应

   客户端接收到响应也会根据序列标识符重新排序归入各自的请求报文

   并且同一个域名下的http连接都复用同一个tcp连接

   增加了服务器并发处理的上限

**### webSocket**

   是html5提出的一种客户端和服务端之间的全双工协议

   通过客户端建立链接之后不仅客户端可以像服务端发送消息，

   服务端也可以像客户端发送消息

**## Cookie**

  比如用来记住状态，保存状态

  服务端发送头部set-cookie

  客户端，即为浏览器就会在发送请求的时候带上cookie的值

  这样服务器再接受到cookie的值的时候就去去对比信息，这次是谁和我说话

  cookie像是两个人沟通的暗号

  如果你没有告诉我暗号，我就告诉你，下次找我的时候说暗号，

  这样我就知道你是你，我就知道上次和你聊到啥时候了，咱俩介绍聊

**## HTTP版本相关**

**### HTTP/1.0**

   就是最简单的一次请求响应就断开tcp连接的协议版本

**### HTTP/1.1**

    1. 引入了很多的缓存控制策略，比如：

       cache-control:

                      no-cache（强制向服务器再次验证）

                      no-store（不做任何缓存）

                      max-age=111111（资源可缓存最大时间 秒）

                      public（客户端、代理服务器都可利用缓存）

                      private（代理服务器不可用缓存）

    2. 增加了允许请求范围，即为在请求头中加入Range头部

      这个在于，客户端已经有了一部分内容了，再次响应的时候为了节省带宽就希望请求内部的其他部分

      就可以使用这个range来标明内容对象的偏移和长度，并且响应状态为206

    3. 请求消息和响应消息都支持host头部

      引入host是因为以前默认每台服务器都只有一个ip地址，但是后来服务器可能有多个虚拟主机，

      共享一个ip，这个时候就需要host来标明是哪个主机了

    4. 默认开启持久连接，用keepAlive

      在1.0中如果要保持长连接就需要增加请求头：content-type:keep-alive

      但是在1.1中就默认为长连接，不需要加请求头，但是可以通过Connection:close来关闭长连接

    5. chunked编码传输

      该编码将实体分块传输，并标明长度，知道长度为0的时候结束，这个在于数据量未知的时候比较好用

    6. 新增了 OPTIONS,PUT, DELETE, TRACE, CONNECT

**### HTTP/2.0**

    1. 新的二进制格式，将原本的文本协议解析改为为二进制格式

    2. 支持多路复用

    3. 头部压缩

    4. 服务端推送

**### HTTP/3.0**

    1. 避免包阻塞

    2. 快速重启会话

**## 状态码**

   1. 2开头的是请求成功

      1. 200 请求成功

      2. 204 请求成功，但是响应没有主体内容

      3. 206 请求响应内容的范围成功

   2. 3开头的重定向

      1. 301 永久重定向

      2. 302 临时重定向，应该是用get方法，重新像新的页面请求

      3. 307 也是临时重定向，但是保持请求方法不变

   3. 4开头的客户端错误

      1. 400 bad request,请求存在语法错误

      2. 401 unauthorized，需要通过http认证的信息才能发送请求，没有权限

      3. 403 forbidden，请求被服务器拒绝

      4. 404  not found，没有在服务器上找到该资源

   4. 5开头的是服务端错误

      1. 500  internal sever error ，服务器在执行请求的时候发生了错误

      2. 501  Not Implemented，服务器不支持当前请求所需要的某改能

      3. 503  service unavailable，服务器错误，无法处理请求，可能服务器没有启动或者超负载

**## HTTP报文**

**## Web服务器**

**### 虚拟主机**

    1.1规范允许一台物理服务器（一个ip地址）搭建多个web站点

    所以就有了一个服务器有多个虚拟主机

    当一个用户访问了一个网页的时候就会通过dns域名系统解析为ip地址

    通过ip地址找到对应的物理服务器在通过请求首部的host字段决定你应该访问哪个虚拟主机

**### 代理服务器**

    代理服务器是客户端和服务器之间的中间服务器，http请求由代理服务器发送给真实的服务器，在通过代理服务器返回数据。

    又分为正向代理和反向代理

    1. 正向代理（科学上网用）

       当客户端告诉代理服务器我要访问谁，中间商就去帮我访问谁。

       但是真正的源服务器不会知道我是谁，向原服务器隐藏你过来真实客户端的身份

       （你告诉代理商你要去哪儿就好了）

    2. 反向代理（跨域使用）

       反向代理是客户端向反向代理服务器发送消息，反向代理服务器自己判断应该走向何处，

       再讲得到的数据返回给客户端，我们可以甚至可以把反向代理当做是真正的服务器，正常发送请求就好了

       当我们本地的浏览器localhost页面请求外部的接口的时候肯定是跨域了，

       所以我们可以直接向自己localhost的服务器发送请求，由这个服务器去访问我们配置的接口，就不存在跨域的问题了（你需要配置你自己的反向代理，接收到请求应该去哪儿）

**### 缓存服务器**

    缓存服务器是一种提高客户端访问数据的速度的一种方法

    将用户经常访问的网络内容放在距离用户更近的中间服务器里面

    每次发起http请求的时候就向中间服务器进行鉴权验证啊，缓存匹配等

    如果都没有再向源服务器发起请求

    缓存服务器就是中间服务器，用来做缓存提高效率用的

**## HTTPS**

     https添加了加密措施，除了客户端和目标服务器，别人不知道我们在做什么数据交换

     有对称加密，非对称加密，混合加密和SSL

     1. 对称加密

     2. 非对称加密

     3. 混合加密

     4. SSL协议

     5. https的优点

        1. 内容加密：采用混合加密技术，中间者无法直接查看明文内容

        2. 验证身份：通过证书认证客户端访问的是自己的服务器

        3. 保护数据完整性：防止传输的内容被中间人冒充或者篡改

     6. http和https的区别

       1. HTTPS协议需要到CA（证书颁发机构）申请证书，一般免费证书很少，需要交费。

       2. HTTP协议运行在TCP之上，所有传输的内容都是明文，HTTPS运行在SSL/TLS之上，SSL/TLS运行在TCP之上     所有传输的内容都经过加密的。

       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HTTP+SSL协议构建的可进行加密传输、身份认证的网络协议，可以有效的防止运营商劫持

          解决了防劫持的一个大问题，比http协议安全。

**## Web安全防范**

**### XSS**

      跨站脚本攻击

**### CSRF**

      跨站请求伪造

**### 点击劫持**

      iframe欺骗

**### 中间人攻击**

      https可以防止中间人攻击

**## 个人记录**

  ```js

    401：未授权，未登录

    404：未找到，not found

    强制缓存的响应头：cache-control（也能使请求头也能使响应头）

    age

    expires

    这三个都是缓存的响应头

    当用户强制刷新，有清除缓存

    http缓存是面试的重中之重

    请求头和响应头，自己也可以添加，非常容易扩 展，也需要服务器认可

    http协议本身就是文本协议，

    user-agent：浏览器的标识，js通过navigator.useragent读出这个内容，表明自己的身份，

    referer:此次页面正在被哪个页面使用，可以用来做防盗链

    防盗链：你自己的页面上使用了自己的服务器上的图片，那么久可以返回

    但是，如果别人的网页上访问了自己服务器上的图片，就不会给返回，因为referer不是自己旗下的网站

    content-security-policy，限制这个浏览器上的js资源是否能被其他的网站加载

    也可以禁止，简称csp

    跨站请求伪造，CSRF，原因在于就算是跨域请求，但是也会发出去，服务器也会处理

    如何防范呢

    通过referer判断是否是自己的网站的请求

    多加一个传输字段，设置请求数据是随时变化的，就是增加一个token令牌/钥匙，请求数据时必须加上

    CORS:未解决跨域问题出现的,有下面几个头

    access-control-allow-origin//对于某个域的请求同意，大多为\*

    access-control-allow-methods//接受的请求方式

    access-control-allow-headers

    access-control-max-age//多少时间内不需要再次发送预检请求

    注意：什么时候有预检请求，什么时候没有

    注意：知道这四个cors的头的意思

    在很多网站上面，请求响应头是可以自己添加的

    content-length和keep-alive一起使用可以实现pipe line

    还有可以做连接复用

    pipe-alive表示tcp链接不要断开

    同时也需要响应头的长度发出来

    跨域的时候，浏览器会发一个预检请求，查看该请求是否能够请求权限

    预检请求是OPTIONS的请求方式，带着referer和host给服务请发送请求

    那么如果能够跨域请求的话就返回一个access-control-allow-origin：\*

    并且可以带着age，表示多少时间内对于该请求都不需要再次发送预检请求，

    但是并不是所有的跨域请求都会触发预检请求

    预检请求：preflight request

    ```