> # Hello,邢文鹏小伙伴!
>
> Chitty的快乐复习礼包1.0先来一波吧哈哈哈哈哈!
>
> 我边复习边整理的一个小文档，希望可以帮你查漏补缺鸭！！
>
> 目录我Check过了，应该都是完整的~
>
> 不过目前设计模式还没有看，数据结构也还没有整理，剑指offer还没有刷完……所以这个文档还木有更新完，欢迎你随时滴滴督促我学习，顺便催我更新嘻嘻

HTML

基础

Doctype有什么作用?

声明文件类型定义（DTD），位于文档中最前面，作用是为了告知浏览器应该用哪种文档类型规范来解析文档。

如何区分严格模式和混杂模式？区分的意义是什么?

- 定义
  - 严格模式（标准模式),浏览器按照W3C标准来解析;
  - 混杂模式，向后兼容的解析方法，浏览器用自己的方式解析代码。
- 如何区分?

  用DTD来判断
  - 严格格式DTD——严格模式;
  - 有URL的过渡DTD——严格模式，没有URL的过渡DTD——混杂模式;
  - DTD不存在/格式不对——混杂模式;
  - HTML5没有严格和混杂之分
- 区分的意义
  严格模式的排版和js运行模式以浏览器支持的最高标准运行。如果只存在严格模式，那么很多旧网站站点无法工作。

为什么HTML5只需要写`<!DOCTYPE HTML>`

HTML5不基于SGML(标准通用标记语言)，因此不需要对DTD进行引用，但是需要doctype来规范浏览器的行为。

meta标签用来做什么?

提供给机器解读的一些元数据。页面搜索引擎优化，定义页面实用语言等等。
属性有两个

1）**http-equiv+content**

- charset(编码格式)
- expires(过期时间)

- refresh(特定时间内自动刷新跳转)
- pragma(禁止浏览器从本地计算机缓存中访问页面内容no-cache)
- widows-target(设定页面在窗口中以独立页面展示，防止被当成frame页调用)
- set-cookie(自定义cooke)、
- ontent-Type(字符集)

2）**name+content**

- keywords(关键字)
- description(主要内容)
- robots(none不被检索)
- author、generator(使用的制作软件)
- copyright
- viewport(缩放比例)

src和href的区别

href 指向网络资源位置，建立当前文档和资源的连接，一般用于超链接

src将资源嵌入到当前文档中，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。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是如此，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。

HTML5

新特点

很多语义化的标签

- canvas js绘图
- draggable属性 可拖动
- geolocationAPI 获取用户地理位置
- audio/video 音频 视频元素
- input类型增多，color、date、datetime、datetime-local、email、month、range、search、tel、time、url、week
- 表单元素增强

  datalist 与input配合使用规定输入域的选项列表；keygen密钥;output定义不同类的输出。
- Web存储，sessionStorge针对一个session进行数据存储，关闭浏览器创港口后清除，localStorage没有事件限制，不过它可能会因为本地时间修改失效。不过大量复杂数据结构一般用indexDB
- Web worker 页面中执行脚本时，页面状态不可响应，直到脚本完成。在后台运行，独立于其他脚本，不会影响页面的性能。（相当于多线程并发）
- SSE server-sent-event 网页自动获取来自服务器的更新。用于接收服务器发送时间通知
- WebSocket 在单个TCP连接上进行全双工通讯的协议。只需要握手一次，形成快速通道，传输数据。客户端和服务器可以直接通过TCP交换数据。获取连接之后，可以用send发送数据，用onmessage接收服务器返回的数据。
- 新API

  History、Command、Application cache ……

什么是shadow DOM

Shadow DOM是浏览器的一种功能，能自动添加子元素，比如radio元素的controls，这些相关元素由浏览器生成。

Canvas有什么用

相当于在页面上新建了一个画布，可以用JS画图。只写一些基本的。

获取DOM,创建画布 getContext('2d');

- 矩形

  fillRect(x,y,width,height) 填充

  strokeRect(x,y,width,height) 边框

  clearRect(x,y,width,height)清除指定区域

- 路径

  创建起始点，画图，闭合路径。路径绘制完成，可以描边或者填充。

  beginPath()新建路径

  closePath()闭合路径

  stroke()描边

  fill()填充

- 移动笔触

  moveTo(x,y) 移动到某点

- 绘制直线

  lineTo(x,y) 从当前位置绘制到(x,y)的一条直线

- 绘制圆

  arc(x,y,radius,startAngle,endAngle,anticlockwise) 以(x,y)为圆心，radius为半径，startAngle和endAngle以X轴为准开始结束的弧度，anticlockwise为true顺时针，false逆时针。

不改变图片原始大小画到canvas上面

第一种，直接在坐标上画图，如果图片大小超出了画布也不缩放

drawImage(image,x,y);

第二种，绘制开始位置，缩放位置，图片会变形

drawImage(image,x1,y1,x2,y2);

第三种，从图片的某个坐标开始截图，截取m*n的大小。然后截的图片从canvas的x1,y1画到x2,y2。

drwaImage(image,xi,yi,m,n,x1,y1,x2,y2)

CSS

基础

页面导入样式时，使用link和@import有什么区别?

- link属于XHTML标签，除了加载CSS外，还能用于定义RSS, 定义rel连接属性等作用；而@import是CSS提供的，只能用于加载CSS;
- 页面被加载的时，link会同时被加载，而@import引用的CSS会等到页面被加载完再加载;
- @import是CSS2.1 提出的，只在IE5以上才能被识别，而link是XHTML标签，无兼容问题;

盒子模型，CSS3的box-sizing值有哪些?

盒子模型是指用来装页面上元素的矩形区域，CSS的盒子模型包括IE盒子模型和标准W3C的模型。区别在于width，IE盒子中width表示Content+padding+border。

box-sizing有三种属性，一种是content-box，一种是border-box,还有一个是从父元素继承的inherit。现在还有一个padding-box。

content-box一 宽高应用到内容框

border-box— 宽高包括了内边距和边框

padding-box—高宽包括了内边距

行内元素有哪些？块级元素有哪些？空元素有哪些

首先：CSS规范规定，每个元素都有display属性，确定该元素的类型，每个元素都有默认的display值，如div的display默认值为“block”，则为“块级”元素；span默认display属性值为“inline”，是“行内”元素。
（1）行内元素有：a b span img input select strong（强调的语气）
（2）块级元素有：div ul ol li dl dt dd h1 h2 h3 h4…p
（3）常见的空元素：
`<br><hr><img><input><link><meta>`
鲜为人知的是：
`<area><base><col><command><embed><keygen><param><source><track><wbr>`

行内元素和块级元素的区别与互换

- 区别
  - 行内元素会在一行水平方向排列，块级元素独占一行，自动填满父级元素
  - 块级元素可以包含行内元素和块级元素，行内只能包含文本和其他行内元素
  - 盒模型属性上，行内元素width height无效，pading和margin垂直方向上无效
- 互换

  display：inline/block
- inline-block元素

  既可以设置高宽，又有inline元素不换行的特性。

inline-block、inline和block的区别?

block是块级元素，能设置宽高，margin/padding都有效前后都有换行符

inline设置宽高无效，margin在竖直方向无效，padding有效，前后无换行符

inline-block可以设置宽高，margin/padding有效，前后无换行符

为什么img是inline还可以设置宽高?

img是**可替换元素**，这类元素的展现效果不是由CSS来控制的。他们是一种外部对象，外观的渲染独立于CSS。内容不受当前文档的样式影响，CSS可以影响可替换元素的位置，但是不会影响到可替换元素自身的内容。（比如iframe，可能有自己的样式表，不会继承父文档的样式）
可替换元素有内置宽高，性质同设置了inline-block一样。

CSS 中 inline 元素可以设置 padding 和 margin 吗?

width、height可以设置，但是没有用。

padding 左右是有用的，上下么有用。

margin 上下左右都有用。

伪类和伪元素?

伪类可以理解为是一种状态，而伪元素则代表一些实实在在存在的元素，但是它们都是抽象刻画的，游离于标准文档之外。

伪类存在的意义是为了通过选择器，格式化dom树以外的信息（:visited,:link），以及**不能被常规CSS选择器获取到的信息**（比如说获取第一个子元素，常规css选择器不行,可以用：first-child）。

伪类常用的有`first-child、last-child、nth-child、first-of-type`(父元素第一个特定的子元素)、`last-of-type、nth-of-type、lang、focus`、lvha（a标签四个）

伪元素可以**创建一些文档语言无法创建的虚拟元素**，比如文档语言没有一种机制可以描述元素内容第一个字母或者第一行，但是伪元素可以`::first-letter,::first-line`。同时伪元素还可以创建文档中不存在的内容比如说`::after,::before`。

伪元素主要有：

`::after,::before,::first-letter,::first-line,::selection`

CSS选择器的权重

- ！important 权重无限大
- 内联样式 写在html标签里的
- 类 伪类 和属性选择器
- 标签选择器和伪元素选择器 div p:after
- 通配符、子选择器、相邻选择器
- 继承的样式没有权值

外边距重叠

多个相邻(兄弟或父子) 普通流的块级元素在垂直方向的margin会重叠

- 两个相邻的外边距都为正数，折叠结果是较大的值
- 两个相邻的外边距为负数，折叠结果是绝对值较大的值
- 两个相邻外边距为一正一负，折叠结果是他们的和

层叠上下文

层叠上线文就是结界，其中的元素如果跟层叠上下文之外的元素发生层叠，就比较他们的层叠水平高低来显示。

创建的方法：postion为relative、absolute、fixed的元素设置z-index

顺序是：底层的border、background，负值z-index，块级盒子，浮动盒子，内联盒子，z-index：auto, 正z-index

说一下什么是BFC

BFC是块级格式化范围，决定了元素如何对其内容进行定位，以及和其他元素的关系和相互作用。可以理解为它就是个**独立的容器，容器里面的布局与外面互不影响。**

触发规则：

- 根元素
- 浮动元素
- postion：absolute 或 fixed
- display：inline-block，table-cell，table-caption，flex，inline-flex
- overflow：不为visible

规则：垂直方向一个一个放；距离由margin决定，同一个bfc里面相邻会重叠；不会和浮动元素重叠；计算高度时浮动子元素也计算；容器内与容器外互不影响。

主要用途：清除浮动 防止margin重叠

元素隐藏方法和区别

- display：none元素不可见，不占据空间，资源会加载，DOM可以访问
- visibility:hidden元素不可见，不能点击，但占据空间，资源会加载，可以使用。
- opacity：0 元素不可见、可以点击，占据空间，可以使用。（不占据的话再加一个position absolute)(不能点击不占据空间 postion absolute+z-index:-1)(不能点击、占据空间 postion relative z-index:-1)

`display: none`和`visibility:hidden`的区别

- display：none元素不占据空间，visibility:hidden空间保留
- display：none会影响opacity过渡效果
- display会产生重绘回流，visibility:hidden只重绘
- display：none节点和子孙节点都不见，visibility:hidden的子孙节点可以设置visibility:visible显示。
- visibility:hidden不会影响计数器计数（ol标签）

Rem,em和px的区别

px是绝对长度单位。

em是相对长度单位，继承父级元素的字体大小，所有字体都是相对于父元素大小的

rem也是相对长度单位，但它是相对于根元素（html),不会像em造成混乱。

如何清除浮动

当元素设置float浮动后，该元素会脱离文档并向左向右浮动，直到碰到父元素或者另一个浮动元素，浮动元素会造成父元素高度塌陷，所以设置完浮动之后需要进行清除浮动

解决方案：

- BFC

  给父容器加上overflow:hidden，加上之后，形成BFC，需要计算超出的大小来隐藏，所以父容器会撑开放入子元素，同时计算浮动的子元素。

  缺点：但是一旦子元素大小超过父容器大小就会显示异常。

- 使用带有clear属性的空元素

  在浮动元素后面添加一个不浮动的空元素，父容器必须考虑浮动子元素的位置，子元素出现在浮动元素后面，所以显示出来就正常了。

  同时要给空元素加上:`style="clear:both"`

  缺点：需要添加额外的html标签，这违背了语义网的原则

- 使用伪元素::after

  它父容器尾部自动创建一个子元素，原理和空元素一样，可以把它设置为height：0不显示，clear：both display:block，保证空白字符不浮动区块。
  (但是：after不支持IE6，只需要添加上zoom：1,这个是激活父元素的haslayout属性，让父元素拥有自己的布局)

```
.clearifx::after{
    content:'';
    height:0;
    clear:both;
    display:block;
}
.clear{
    zoom:1;
}
```

img和background-image的区别

- 解析机制：img属于html标签，background-img属于css。img先解析
- SEO：img标签有一个alt 属性可以指定图像的替代文本，有利于SEO，并且在图片加载失败时有利于阅读
- 语义化角度：img语义更加明确

rgba()和opacity的区别

- opacity作用于元素及元素中所有的内容（包括文字、图片） 有继承性
- rgba()只用于元素的颜色及背景色
- 当opacity属性的值应用于某个元素上时，把这个元素和它内容当作一个整体来看待，即使这个值没有被子元素继承。因此一个元素和它包含的元素都会有与元素背景相同的透明度，哪怕父子元素由不同的opacity的值。

outline和border的区别

- outline轮廓是绘制于元素周围的一条线，位于边框边缘的外围，可以起到突出元素的作用
- outline的效果将随元素的focus而自动出现，相应的由blur自动消失，这些都是浏览器的默认行为，不需要js配合css来控制
- outline不占据空间，不会像border那样影响元素的尺寸或者位置。

CSS动画如何实现

创建动画序列，需要animation属性或其子属性，属性允许配置动画时间、时长和动画细节。

动画的实际表现由@keyframes 规则实现

transtion也可以实现动画，但强调过渡，是元素的一个或多个属性变化时产生的过渡效果，同一个元素通过两个不同的途径获取样式，而第二个途径当某种改变发生时（如：hover)才能获取样式，这样就会产生过渡动画。

transition、animation的区别

animation和transition大部分属性相同，都是随时间改变元素的属性值，区别是transition需要触发一个事件才能改变属性；animation不需要触发任何事件随时间改变属性。transition为2帧，从from.....to ，animation可以一帧一帧的。

布局

水平居中的实现方案

- 利用块级元素撑满父元素的特点，如果宽度已定，左右margin auto就可以平分剩余空间
- 利用行内块居中： 把父级元素设置为text-align=center，之后子元素的display设置为inline-block
- 绝对定位：postion ： absolute，之后left 50%
- flex: 父元素**display:flex justify-content：center**

垂直居中的实现方案

- 元素无高度

  利用内边距，让块级文字包裹在padding中，实现垂直居中。

- 父元素高度确定的单行文本：

  使用行高的特性：height=line-height即可。

- 父元素高度确定的多行文本：

  利用vertical-align（只能内联元素）如果是div，可以设置为table和table-cell。

  ```
  display：table-cell;
  vertical-align:center;
  ```

- 父元素高度未知：

  绝对定位，设置top 50%

  ```
  parent{
      position:relative;
  }
  child{
      position:absolute;
      top:50%;
      transform:translateY(-50%);
  }
  ```

  如果子元素有高度

  ```
  child{
      position:absolute;
      top:50%;
      height=Hpx;
      margin-top:(H/2)px;
  }
  ```

- 父元素高度已知

```
parent{
    height:Hpx;
}
child{
    position:relative;
    top:50%;
    transform:translateY(-50%);
}
```

- flex方法

```
disply:flex;
align-items:center;
```

垂直水平居中的实现方案

- 居中元素的宽高已知
  - 利用绝对定位和margin

```
parent{
    position:relative;
}
child{
    position:absolute;
    top:50%;
    left:50%;
    margin-top:(-H/2)px;
    margin-left:(-W/2)px;
}
```

  - absolute+margin:auto

```
child{
    width:50px;
    height:50px;
    position:absolute;
    top:0;
    left:0;
    right:0;
    bottom:0;
    margin:auto;
}
```

  - 用calu计算

```
child{
    position:absolute;
    top:calc(50%-50px);
    left:calc(50%-50px);
}
```

- 垂直居中的元素宽高未知
  - transform的translate方法

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

- flex布局

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

两栏布局，左边固定，先加载内容区

- float。两个div。左边float:left，width:200 px，右边 margin-left=width。
- 绝对定位。两个div。左边absolute或者fixed 右边margin-left=width
- table布局。三个div，父元素display：table，子元素display table-cell width，右边自适应
- flex布局。三个div,父元素display flex; 子元素flex:1

三栏布局

- 浮动布局 float:left right，中间根据两边的width设置margin(要加两边的border)
- 绝对定位 父元素 relative，左右leftright各为0，absolute，中间元素设置margin
- BFC 左右float，中间overflow：hidden
- Flex 写法左中右，父元素display:flex，中间区域flex:1
- table布局，写法左中右，父元素display：table，三个元素table-cell
- 圣杯布局 写法中左右，中间width100%，左边margin-left -100%，右边margin-left= -width（-100px); 然后防遮住中间，左右相对定位relative。
- 双飞翼 中间再包裹一层，左，右。中间父元素 float left，width：100%。

  中间子元素 margin left right 左右的width。 左右float,左边margin-left -100%，右边margin-left -width

Flex布局

Flex是弹性布局，用来为盒装模型提供最大的灵活性。布局的传统解决方案基于盒装模型，依赖display、position和float属性。**任何一个容器都可以指定为 Flex 布局。**

注意：**设置为 Flex 布局后，子元素的 float 、clear 和 vertical-align 属性将失效。**

属性分为容器属性和元素属性：

- 容器属性包括
  - flex-direaction:决定主轴方向

```
.box{
    flex-direaction:row|row-reverse|column|column-reverse;
}
```

  - flex-wrap:决定了如何换行

```
.box{
    flex-wrap:nowrap|wrap|wrap-reverse;
    /*不换行|换行第一行在上|换行第一行在下*/
}
```

- flex-flow:前面两个的简写

```
.box{
      flex-flow:<flex-direction>||<flex-wrap>;
  }
```

- justify-content：水平轴对齐方式

```
.box{
      justify-content:flex-start|flex-end|center|space-between|space-around;

  }
```

- align-items: 垂直轴对齐方式

```
.box{
    align-items:flex-start|flex-end|center|space-between|space-around;
}
```

- 元素属性align-content
  - order 定义项目的排序顺序，越小越靠前
  - flex-grow 放大比列，默认是0，即使存在空间也不会放大，1是说等分剩余空间
  - flex-shrink 缩小比例，当空间不够的情况下，会等比缩小为0不缩小，为1等比缩小
  - flex-basis 定义分配多余空间时，项目占据的控件
  - flex 上面三个属性的缩写，默认是0 1 auto。后两个属性可选。
  - align-self 允许单个与其他不一样的对齐方式，可以覆盖align-items属性，默认是auto表示继承。

多端适应

静态布局

特点：**不管浏览器尺寸具体是多少**，网页布局始终**按照最初写代码时的布局**来显示。

一般都使用min-width定宽，小于这个宽度就会出现滚动条，大于就内容居中加背景。

设计方法：

PC端居中布局所有样式绝对宽高，设计一个layout，在屏幕宽高有调整时，使用横向和竖向的滚动条来查阅被遮掩的部分。

移动设备另外建立，单独设计一个布局，使用不同的域名如wap或m

流式布局

布局特点：**屏幕分辨率变化时，页面里元素的大小会变化而但布局不变。**【这就导致如果屏幕太大或者太小都会导致元素无法正常显示】

设计方法：使用%百分比定义宽度，高度大都是用px来固定住，可以根据可视区域 (viewport) 和父元素的实时尺寸进行调整，尽可能的适应各种分辨率。往往配合 max-width/min-width 等属性控制尺寸流动范围以免过大或者过小影响阅读。

这种布局方式在Web前端开发的早期历史上，用来**应对不同尺寸的PC屏幕**（那时屏幕尺寸的差异不会太大），在当今的移动端开发也是常用布局方式，但缺点明显：**如果屏幕尺度跨度太大，那么在相对其原始设计而言过小或过大的屏幕上不能正常显示。**因为宽度使用%百分比定义，但是高度和文字大小等大都是用px来固定，所以在大屏幕的手机下显示效果会变成有些页面元素宽度被拉的很长，但是高度、文字大小还是和原来一样，显示非常不协调。

自适应布局

分别为不同的屏幕分辨率定义布局，即**创建多个静态布局，每个静态布局对应一个屏幕分辨率范围。**改变屏幕分辨率可以切换不同的静态局部（页面元素位置发生改变），但在每个静态布局中，页面元素不随窗口大小的调整发生变化。可以把自适应布局看作是静态布局的一个系列。

布局特点：屏幕**分辨率变化**时，页面里面元素的**位置会变化而大小不会变化。**

设计方法：使用 @media 媒体查询给不同尺寸和介质的设备切换不同的样式。在优秀的响应范围设计下可以给适配范围内的设备最好的体验，在同一个设备下实际还是固定的布局。

响应式布局

响应式设计的目标是确保一个页面在所有终端上（各种尺寸的PC、手机、手表、冰箱的Web浏览器等等）都能显示出令人满意的效果，对CSS编写者而言，在实现上不拘泥于具体手法，但通常是糅合了流式布局+弹性布局，再搭配媒体查询技术使用。

分别为不同的屏幕分辨率定义布局，同时，在每个布局中，应用流式布局的理念，即页面元素宽度随着窗口调整而自动适配。即：创建多个流体式布局，分别对应一个屏幕分辨率范围。可以把响应式布局看作是流式布局和自适应布局设计理念的融合。

布局特点：**每个屏幕分辨率下面会有一个布局样式，即元素位置和大小都会变。**

媒体查询+流式布局。通常使用 @media 媒体查询 和网格系统 (Grid System) 配合相对布局单位进行布局，实际上就是综合响应式、流动等上述技术通过 CSS 给单一网页不同设备返回不同样式的技术统称。

优点：适应pc和移动端，如果足够耐心，效果完美

缺点：（1）媒体查询是有限的，也就是可以枚举出来的，只能适应主流的宽高。（2）要匹配足够多的屏幕大小，工作量不小，设计也需要多个版本。

弹性布局

特点是：**包裹文字的各元素的尺寸采用em/rem做单位，而页面的主要划分区域的尺寸仍使用百分数或px做单位（同「流式布局」或「静态/固定布局」）。早期浏览器不支持整个页面按比例缩放**，仅支持网页内文字尺寸的放大，这种情况下。使用em/rem做单位，可以使包裹文字的元素随着文字的缩放而缩放。

JS

基本数据类型与类型判断

基本数据类型

6种，boolean，number，string，undefined，null，symbol

null是Object吗

虽然 typeof null会输出 object，但是这只是 JS 存在的一个悠久 Bug。在 JS 的最初版本中使用的是 32 位系统，为了性能考虑使用低位存储变量的类型信息，`000` 开头代表是对象，然而 `null` 表示为全零，所以将它错误的判断为 `object` 。虽然现在的内部类型判断代码已经改变了，但是对于这个 Bug 却是一直流传下来。

原始类型与对象类型区别

对象类型和原始类型不同的是，原始类型存储的是值，对象类型存储的是地址。

当创建了一个对象类型的时候，计算机会在内存中帮我们开辟一个空间来存放值，但是我们需要找到这个空间，这个空间会拥有一个地址（指针）。

typeof vs instanceof

typeof对于原始类型来说，除null会显示成object，其余都可以显示正确的类型typeof 对于对象来说，除了函数都会显示 object，所以说typeof并不能准确判断变量到底是什么类型。

如果我们想判断一个对象的正确类型，这时候可以考虑使用 instanceof，因为内部机制是通过原型链来判断的。

如何判断一个对象类型是数组

- 根据构造函数来判断 xxx instanceof Array
- 根据class属性判断 Object.prototype.toString.call(obj)==='[object Array]'
- 直接用isArray判断

类型转换

在 JS 中类型转换只有三种情况，分别是：

- 转换为布尔值
- 转换为数字
- 转换为字符串

转Boolean

在条件判断时，除了 `undefined`， `null`， `false`， `NaN`， `''`， `0`， `-0`，其他所有值都转为 `true`，包括所有对象。

对象转原始类型

对象在转换类型的时候，会调用内置的 `[[ToPrimitive]]` 函数，对于该函数来说，算法逻辑一般来说如下：

- 如果已经是原始类型了，那就不需要转换了
- 如果需要转字符串类型就调用 `x.toString()`，转换为基础类型的话就返回转换的值。不是字符串类型的话就先调用 `valueOf`，结果不是基础类型的话再调用 `toString`
- 调用 `x.valueOf()`，如果转换为基础类型，就返回转换的值
- 如果都没有返回原始类型，就会报错

当然可以重写 `Symbol.toPrimitive` ，该方法在转原始类型时调用优先级最高。

```
let a = {
  valueOf() {
    return 0
  },
  toString() {
    return '1'
  },
  [Symbol.toPrimitive]() {
    return 2
  }
}
1 + a // => 3
```

四则运算符

加法运算符不同于其他几个运算符，它有以下几个特点：

- 运算中其中一方为字符串，那么就会把另一方也转换为字符串
- 如果一方不是字符串或者数字，那么会将它转换为数字或者字符串

```
1 + '1' // '11'
true + true // 2
4 + [1,2,3] // "41,2,3"
```

如果你对于答案有疑问的话，请看解析：

- 对于第一行代码来说，触发特点一，所以将数字 `1` 转换为字符串，得到结果 `'11'`
- 对于第二行代码来说，触发特点二，所以将 `true` 转为数字 `1`
- 对于第三行代码来说，触发特点二，所以将数组通过 `toString` 转为字符串 `1,2,3`，得到结果 `41,2,3`

另外对于加法还需要注意这个表达式 `'a' + + 'b'`

```
'a' + + 'b' // -> "aNaN"
```

因为 `+ 'b'` 等于 `NaN`，所以结果为 `"aNaN"`。

对于除了加法的运算符来说，只要其中一方是数字，那么另一方就会被转为数字

```
4 * '3' // 12
4 * [] // 0
4 * [1, 2] // NaN
```

比较运算符

1. 如果是对象，就通过 `toPrimitive` 转换对象
2. 如果是字符串，就通过 `unicode` 字符索引来比较

```
let a = {
  valueOf() {
    return 0
  },
  toString() {
    return '1'
  }
}
a > -1 // true
```

在以上代码中，因为 `a` 是对象，所以会通过 `valueOf` 转换为原始类型再比较值。

==比较

- Boolean，number，string三类比较的时候把值转换成数字，在看转换结果是否相等。证明：('1'==true) 是真 ('abc'==true）是假。
- undefined 参与比较，换成了NaN,所以其他三个类型跟它比较都是false，跟null类型比较的时候是true。 (NaN==NaN)是假
- null参与比较，被当成对象，因为null没有valueof和toString，除了undefined谁跟他比较都是false。
- 值类型与对象比较：先调用对象valueof 如果仍返回对象，调用tostring，如果还是没有就不等。

==与===

对于 `==` 来说，如果对比双方的类型**不一样**的话，就会进行**类型转换。**

假如我们需要对比 `x` 和 `y` 是否相同，就会进行如下判断流程：

1. 首先会判断两者类型是否**相同。**相同的话就是比大小了
2. 类型不相同的话，那么就会进行类型转换
3. 会先判断是否在对比 `null` 和 `undefined`，是的话就会返回 `true`
4. 判断两者类型是否是string和number，是的话就会将字符串转换为number

```
1 == '1'
     ↓
1 ==  1
```

5. 判断其中一方是否为boolean，是的话就会把boolean转为number再进行判断

```
'1' == true
       ↓
'1' ==  1
       ↓
 1  ==  1
```

6. 判断其中一方是否object且另一方为string、number或者symbol，是的话就会把object转为原始类型再进行判断

```
'1' == { name: 'xxx' }
       ↓
'1' == '[object Object]'
```

现在来想想[]==![]输出什么?

首先先执行的是![]，它会得到false。

然后[]==false，返回true。

那么[]==[],{}=={}又输出什么呢?

类型一致，它们是引用类型，地址是不一样的，所以为false!

对于 `===` 来说就简单多了，就是判断两者类型和值是否相同。

深拷贝与浅拷贝

浅拷贝

- `Object.assign`

```
let a = {
  age: 1
}
let b = Object.assign({}, a)
a.age = 2
console.log(b.age) // 1
```

- 展开运算符 `...`

```
let a = {
  age: 1
}
let b = { ...a }
a.age = 2
console.log(b.age) // 1
```

深拷贝

- `JSON.parse(JSON.stringify(object))`

```
let a = {
  age: 1,
  jobs: {
    first: 'FE'
  }
}
let b = JSON.parse(JSON.stringify(a))
a.jobs.first = 'native'
console.log(b.jobs.first) // FE
```

局限性：会忽略 `undefined`；会忽略 `symbol`；不能序列化函数；不能解决循环引用的对象

- 写一个：

```
function deepClone(obj) {
  function isObject(o) {
    return (typeof o === 'object' || typeof o === 'function') && o !== null
  }
```

```
  if (!isObject(obj)) {
    throw new Error('非对象')
  }

  let isArray = Array.isArray(obj)
  let newObj = isArray ? [...obj] : { ...obj }
  Reflect.ownKeys(newObj).forEach(key => {
    newObj[key] = isObject(obj[key]) ? deepClone(obj[key]) : obj[key]
  })

  return newObj
}
```

this

四种绑定规则

详情参考这个

默认绑定

- 全局环境，默认绑定到window
- 函数独立调用的时候，this默认绑定到window
- 被嵌套的函数独立调用时，this默认绑定到window
- 立即执行的函数this是window
- 闭包的this默认绑定到window

隐式绑定

- 被直接对象所包含的函数调用时，也称为方法调用，this隐式绑定到该直接对象

显示绑定

- 通过call()、apply()、bind()方法把对象绑定到this上，叫做显式绑定。对于被调用的函数来说，叫做间接调用

new绑定

- 如果函数或者方法调用之前带有关键字new，它就构成构造函数调用。对于this绑定来说，称为new绑定

优先级

首先，`new` 的方式优先级最高，接下来是 `bind` 这些函数，然后是 `obj.foo()` 这种调用方式，最后是 `foo` 这种调用方式，同时，箭头函数的 `this` 一旦被绑定，就不会再被任何方式所改变。

实例与解析

先来看几个函数调用的场景：

```
function foo() {
  console.log(this.a)
}
var a = 1
foo();

const obj = {
  a: 2,
  foo: foo
}
obj.foo()

const c = new foo()
```

接下来一个个分析上面几个场景

- 对于直接调用 `foo` 来说，不管 `foo` 函数被放在了什么地方，`this` 一定是 `window`
- 对于 `obj.foo()` 来说，我们只需要记住，**谁调用了函数，谁就是** `this`，所以在这个场景下 `foo` 函数中的 `this` 就是 `obj` 对象
- 对于 `new` 的方式来说，`this` 被永远绑定在了 `c` 上面，不会被任何方式改变 `this`

说完了以上几种情况，其实很多代码中的 `this` 应该就没什么问题了。

下面让我们看看箭头函数中的 `this`

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

首先箭头函数其实是没有 `this` 的，**箭头函数中的** `this` **只取决包裹箭头函数的第一个普通函数的** `this`。在这个例子中，因为包裹箭头函数的第一个普通函数是 `a`，所以此时的 `this` 是 `window`。另外**对箭头函数使用** `bind` **这类函数是无效的。**

最后种情况也就是 `bind` 这些改变上下文的 API 了，对于这些函数来说，`this` 取决于第一个参数，如果第一个参数为空，那么就是 `window`。

那么说到 `bind`，**如果对一个函数进行多次** `bind`**，那么上下文会是什么**呢?

```
let a = {}
let fn = function () { console.log(this) }
fn.bind().bind(a)() // => ?
```

如果认为输出结果是 `a`，那么就错了。

其实可以把上述代码转换成另一种形式：

```
// fn.bind().bind(a) 等价于
let fn2 = function fn1() {
  return function() {
    return fn.apply()
  }.apply(a)
}
fn2()
```

可以从上述代码中发现，不管我们给函数 `bind` 几次， `fn` 中的 `this` 永远由第一次 `bind` 决定，所以结果永远是 `window`。

闭包与作用域链

闭包

- 定义：闭包就是能够读取其他函数内部变量的函数，其实就是利用了作用域链向上查找的特点。
- 作用：读取函数内部变量，让这些变量的值一直保持在内存中。

作用域链

- 作用域链针对函数作用域来说的，比如创建了一个函数，函数里面又包含了一个函数，那么就会有全局作用域、函数1的作用域、函数2的作用域。
- 查找规范：先在自己的变量范围中找，如果找不到就沿着作用域往上找。

原型链

说一下什么是原型链

在js中，每个函数都有prototype属性，这个属性值是一个对象，同时它带有constructor属性，指向这个构造函数。

通过new可以创建一个函数的实例，每个实例都有`_proto_`属性，指向构造函数的prototype对象。

当我们在一个对象或者方法获取某个值的时候，会先查找实例上是否存在这个值，如果没有的话，就在原型里查找。

怎么判断一个属性是对象上的属性还是其原型对象上的属性

使用hasOwnProperty()返回true，说明是这个对象上的；如果返回false，但是属性in 这个对象返回了true，说明是原型对象上的属性。如果都是false，那么不存在这个属性。

继承

继承的原理

还是跟原型链有关。每个函数都有个原型对象，这个对象用来存储通过这个函数所创建的所有实例的共有属性和方法。在读取某个对象属性的时候，从实例开始，如果实例有就返回，如果没有就找原型对象，找到了就返回。通过实例只能访问原型对象里的值，但是不能修改。这就实现了继承。

组合继承

组合继承是最常用的继承方式，

```
function Parent(value) {
```

```
  this.val = value
}
Parent.prototype.getValue = function() {
  console.log(this.val)
}
function Child(value) {
  Parent.call(this, value)
}
Child.prototype = new Parent()

const child = new Child(1)

child.getValue() // 1
child instanceof Parent // true
```

以上继承的方式核心是在子类的**构造函数中通过 `Parent.call(this)` 继承父类的属性**，然后改变**子类的原型为 `new Parent()`** 来继承父类的函数。

这种继承方式优点在于构造函数可以传参，不会与父类引用属性共享，可以复用父类的函数，但是也存在一个缺点就是在继承父类函数的时候调用了父类构造函数，导致子类的原型上多了不需要的父类属性，存在内存上的浪费。

寄生组合继承

这种继承方式对组合继承进行了优化，组合继承缺点在于继承父类函数时调用了构造函数，我们只需要优化掉这点就行了。

```
function Parent(value) {
  this.val = value
}
Parent.prototype.getValue = function() {
  console.log(this.val)
}

function Child(value) {
  Parent.call(this, value)
}
Child.prototype = Object.create(Parent.prototype, {
  constructor: {
    value: Child,
    enumerable: false,
    writable: true,
    configurable: true
  }
})

const child = new Child(1)

child.getValue() // 1
child instanceof Parent // true
```

以上继承实现的核心就是将父类的原型赋值给了子类，并且将构造函数设置为子类，这样既解决了无用的父类属性问题，还能正确的找到子类的构造函数。

Class 继承

以上两种继承方式都是通过原型去解决的，在 ES6 中，我们可以使用 `class` 去实现继承，并且实现起来很简单

```
class Parent {
  constructor(value) {
    this.val = value
  }
  getValue() {
    console.log(this.val)
  }
}
class Child extends Parent {
  constructor(value) {
    super(value)
  }
}
let child = new Child(1)
child.getValue() // 1
child instanceof Parent // true
```

`class` 实现继承的核心在于使用 `extends` 表明继承自哪个父类，并且在子类构造函数中必须调用 `super`，因为这段代码可以看成 `Parent.call(this, value)`。

Class中的箭头函数和普通函数有何区别

箭头函数相当于绑定上了this，类似构造函数内的一个值。

箭头函数会绑定在实例对象上，普通函数最后会绑定在原型上。

DOM与事件

DOM如何创建元素

- 创建

  createHTML('div');

- 添加

  appendChild(element)

  insertBefore(insertdom.chosendom)

DOM获取元素的方式

通过元素类型获取

1. document.getElementById();//id名，在实际开发中较少使用，选择器中多用class id一般只用在顶级层存在 不能太过依赖id
2. document.getElementsByTagName();//标签名
3. document.getElementsByClassName();//类名
4. document.getElementsByName();//name属性值，一般不用
5. document.querySelector();//css选择符模式，返回与该模式匹配的第一个元素，结果为一个元素；如果没找到匹配的元素，则返回null
6. document.querySelectorAll()//css选择符模式，返回与该模式匹配的所有元素，结果为一个类数组

根据关系树来选择

1. parentNode//获取所选节点的父节点，最顶层的节点为#document
2. childNodes //获取所选节点的子节点们
3. firstChild //获取所选节点的第一个子节点
4. lastChild //获取所选节点的最后一个子节点
5. nextSibling //获取所选节点的后一个兄弟节点 列表中最后一个节点的nextSibling属性值为null
6. previousSibling //获取所选节点的前一兄弟节点 列表中第一个节点的previousSibling属性值为null

根据元素节点树来选择

1. parentElement //返回当前元素的父元素节点（IE9以下不兼容）
2. children // 返回当前元素的元素子节点
3. firstElementChild //返回的是第一个元素子节点（IE9以下不兼容）
4. lastElementChild //返回的是最后一个元素子节点（IE9以下不兼容）
5. nextElementSibling //返回的是后一个兄弟元素节点（IE9以下不兼容）
6. previousElementSibling //返回的是前一个兄弟元素节点（IE9以下不兼容）

如何给元素添加事件

- 在HTML元素中绑定事件 onclick=show()
- 获取dom，dom.onclick
- addEventListener(click,show,1/0)

addEventListener三个参数，取值意思

第一个参数是事件类型，第二个是事件发生的回调函数，第三个是个布尔值，默认是false，false是冒泡阶段执行，true是捕获阶段。

事件冒泡与事件捕获

事件是先捕获，后冒泡

捕获阶段是外部元素先触发然后触发内部元素

冒泡阶段是内部元素先触发然后触发外部元素

如何阻止事件冒泡？如何取消默认事件？如何阻止事件的默认行为？

- 阻止事件冒泡：

  W3C: stopPropagation();

  IE: e.cancelBubble=true;

  写法 :

  window.event ? window.event.cancelBubble=true:e.stop(Propagation)

- 取消默认事件

  W3C：preventDefault()

  IE: e.returnValue:false;

- 阻止默认行为：

  return false

  原生的js会阻止默认行为，但会继续冒泡；

jquery会阻止默认行为，并停止冒泡。

获取DOM节点get系列和query系列哪种性能好?

1.从性能上说get系列的性能都比query系列好，get系列里面各有差异，这些差异可以结合算法如何遍历搜索去理解，都解释得通。

2.`getElementsByTagName`比`querySelectorAll`快的原因在于：`getElementsByTagName`创建的过程不需要做任何操作，只需要返回一个指针即可。而`querySelectorAll`会循环遍历所有的的结果，然后创建一个新的NodeList。

3.实际在用的过程中**取决于要获取的是什么**，再进行选择。

ES6

变量提升与var、let 及 const

首先，我们先了解提升（hoisting）这个概念：

```
console.log(a) // undefined
var a = 1
```

从上述代码中我们可以发现，虽然变量还没有被声明，但是我们却可以使用这个未被声明的变量，这种情况就叫做提升，并且提升的是声明。

对于这种情况，我们可以把代码这样来看

```
var a
console.log(a) // undefined
a = 1
```

接下来，再来看一个例子：

```
var a = 10
var a
console.log(a)
```

对于这个例子，如果你认为打印的值为`undefined`那么就错了，答案应该是`10`，对于这种情况，等价于下面这样：

```
var a
var a
a = 10
console.log(a)
```

其实不仅变量会提升函数也会被提升:

```
console.log(a) // ƒ a() {}
function a() {}
var a = 1
```

对于上述代码，打印结果会是`ƒ a() {}`，即使变量声明在函数之后，这也说明了函数会被提升，并且优先于变量提升。

使用 `var` 声明的变量会被提升到作用域的顶部，接下来我们再来看 `let` 和 `const` 。

还是来看一个例子：

```
var a = 1
let b = 1
const c = 1
console.log(window.b) // undefined
console.log(window. c) // undefined

function test(){
  console.log(a)
  let a
}
test();//报错
```

首先在全局作用域下使用 `let` 和 `const` 声明变量，变量并不会被挂载到 `window` 上，这一点就和 `var` 声明有了区别。

再者当我们在声明 `a` 之前如果使用了 `a`，就会出现报错：

Uncaught ReferenceError: Cannot access 'a' before initialization

报错的原因是因为存在**暂时性死区**，我们不能在声明前就使用变量，这也是 `let` 和 `const` 优于 `var` 的一点。然后这里你认为的提升和 `var` 的提升是有区别的，虽然变量在编译的环节中被告知在这块作用域中可以访问，但是访问是受限制的。

`var`、`let` 及 `const` 区别其实已经不言而喻了。

其实**提升存在的根本原因就是为了解决函数间互相调用的情况**

```
function test1() {
    test2()
}
function test2() {
    test1()
}
test1();
```

假如不存在提升这个情况，那么就实现不了上述的代码，因为不可能存在 `test1` 在 `test2` 前面然后 `test2` 又在 `test1` 前面。

小结:

- 函数提升优先于变量提升，函数提升会把整个函数挪到作用域顶部，变量提升只会把声明挪到作用域顶部
- `var` 存在提升，我们能在声明之前使用。`let`、`const` 因为暂时性死区的原因，不能在声明前使用
- `var` 在全局作用域下声明变量会导致变量挂载在 `window` 上，其他两者不会
- `let` 和 `const` 作用基本一致，但是后者声明的变量不能再次赋值

Promise

Promise是一种用于解决异步问题的思路、方案，简单说是个容器，里面存的是某个未来会结束的结果。是一个对象，可以获取异步操作的消息。有三种状态，pending resolved rejected ,状态变了就不能修改了。

在js里面，经常用异步的是ajax，比如sucess：一个回调，error一个回调。但是如果一次请求需要多个接口的时候就产生了回调地狱，promise可以用then来处理，它可以在一个then里面再写一个promise对象。

promise.all 多任务并行，输出失败的那个，如果成功，返回所有的执行结果。

promise.race 多任务执行，返回最先执行结束的任务结果。

Generator+yield

是ES6里面的新数据类型，像一个函数，可以返回多次。特点就是函数有个*号。

调用的话就是不断调用next() 返回当前的value 值 done的状态

return() 直接忽略所有yield ， 返回最终的结果

可以随心所欲的交出和恢复函数的执行权。

await+async

async函数返回的是一个promise对象，await用于等待一个async函数的返回值。

优势在于处理then链。如果是多个Promise组成的then链，那么优势就比较明显了，可以用Promise来解决多层回调的题，可以进一步优化它。

Proxy

Proxy 是 ES6 中新增的功能，它可以用来自定义对象中的操作。

```
let p = new Proxy(target, handler)
```

`target` 代表需要添加代理的对象，`handler` 用来自定义对象中的操作，比如可以用来自定义 `set` 或者 `get` 函数。

接下来我们通过 `Proxy` 来实现一个数据响应式

```
let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver)
    },
    set(target, property, value, receiver) {
      setBind(value, property)
      return Reflect.set(target, property, value)
    }
  }
  return new Proxy(obj, handler)
}

let obj = { a: 1 }
let p = onWatch(
  obj,
  (v, property) => {
    console.log(`监听到属性${property}改变为${v}`)
  },
  (target, property) => {
    console.log(`'${property}' = ${target[property]}`)
```

```
  }
)
p.a = 2 // 监听到属性a改变
p.a // 'a' = 2
```

在上述代码中，我们通过自定义 `set` 和 `get` 函数的方式，在原本的逻辑中插入了我们的函数逻辑，实现了在对对象任何属性进行读写时发出通知。

EventLoop

浏览器执行是有一个执行栈的，当遇到异步的代码时，会被挂起并在需要执行的时候加入到task队列里面，一旦执行栈为空，eventloop会从Task队列里拿出需要执行的代码放入执行栈中执行。执行完毕后就弹出。

但是不同的任务源会分到微任务和宏任务里面。

顺序：

首先执行同步代码，属于宏任务

执行完所有宏任务之后，查询有没有异步代码需要执行，然后执行微任务

执行微任务之后，如果有必要会重新渲染页面

然后一下轮执行宏任务中异步代码，比如说setTimeout的回调函数。

数组

API有哪些?

直接修改原数组的：push(),unshift(), pop(),shift(), splice() ,reverse(), sort() fill(x,start,end) copyWithin(tochangeindex,changestart,changeend)

返回新数组的： concat(),slice()

返回字符串： join()

位置或是否在数组内： indexOf() lastindexOf() includes()是否包括 find()满足条件的索引;

遍历方法：forEach()所有元素执行一次，返回undefined;map()返回值新数组;filter()返回通过元素的新数组 ;every()所有都满足返回true;some()只要有一个元素满足就返回true；reduce(fn(pre,cur,index,arr),basevalue)累加器;reduceRight()从右边往左边加

迭代器：arr.keys()返回索引迭代器；arr.values()返回迭代器，值；arr.entries()返回键值对。

数组的sort方法底层如何排序?

谷歌浏览器：大于22的数组 快排；小于22的用插入排序

火狐浏览器：归并排序

webkit：用的c++的qsort();

前后端数据交互

交互方式

利用Cookie

Cookie 是一些数据, 存储于电脑上的文本文件中,只要客户端cookie开放且有数据，每一次请求都会自动添加到http报文中，后台可以实时接收观察获取这些Cookie 。

Cookie 的作用就是用于解决 "如何记录客户端的用户信息":

- 当用户访问 web 页面时，他的名字可以记录在 cookie 中。
- 在用户下一次访问该页面时，可以在 cookie 中读取用户访问记录。

利用Session对象

session对象表示特定会话session的用户数据。

客户第一次访问支持session的JSP网页，服务器会创建一个session对象记录客户的信息。当客户访问同一网站的不同网页时，仍处于同一个session中。

```
request.getSession().setAttribute();
request.getSession().getAttribute();
```

只要浏览器不关闭，就能使用。所以用户访问网站整个生命都会用到的数据一般都用session来存储，比如用户名、登录状态之类的。

利用Request参数设置

```
request.setAttribute();
request.getRequestDispatcher("welcome.jsp").forward(request, response);

request.getAttribute();
```

不能用sendRedirect(),因为已经切换到另一个请求了，request参数的有效期为本次请求。

Form表单

form表单的action设置好路径

```
<form id="loginform" name="loginform" action="<%=path %>/login" method="post">
</form>
```

Ajax

前端用ajax发起请求。

```
window.onload=function(){
    var jsondata={
        "name":"Sarrans",
        "password":"123456"
    }
    $.ajax({
        type:"post",
        url:"login",
        data:jsondata,
        success:function(data){
            alert(data.name+"请求成功");
        }
```

```
        error:function(e){
            alert("Error");
    }
    })
}
```

后台servlet接收请求处理

```
@WebServlet("/login")//ajax的url
public class login extends HttpServlet{
    protected void doPost(HttpServletRequest request,HttpServletResponse
response) throws ServletException,IOException{
        //从前端传递的request取值
        String name=request.getParameter("name");
        //构造一个新的json传回去
        String s="{\"name\":\"Gerorge\",\"password\":\"1234567\"}";
        response.setCharacterEncoding("utf-8");
        response.setContentType("application/json;charset=utf-8");
        response.getWriter().write(s);//写入返回结果
        //如果前台结果为success，会输出 Gerorge请求成功
    }
}
```

jsonp

结合跨域方式，因为前端请求到数据需要在回调函数中使用，所以后端得将数据放回到回调函数中。

```
$.ajax({
    url:"",
    dataType:"jsonp",
    jsonp:'callback',
    success(function(res){
      console.log(res)
    })
})
```

性能改进

Comet

Comet的实现主要有两种方式，基于Ajax的长轮询方式和基于 Iframe 及 htmlfile 的流（http streaming）方式。而这些大部分功能在后台完成，前端要做的就是通过ajax请求成功后，在XMLHttpRequest的onreadystatechange函数中持续获取数据。

典型的Ajax通信方式也是http协议的经典使用方式，要想取得数据，必须首先发送请求。在低延迟要求比较高的web应用中，只能增加服务器请求的频率。Comet则不同，**客户端与服务器端保持一个长连接，只有客户端需要的数据更新时，服务器才主动将数据推送给客户端。**

```
var xhr=getXmlHttpRequest();
xhr.onreadystatechange=function(){
    console.log(xhr.readyStae);
    if(xhr.readyState===3&&xhr.status===200){
        //获取成功后执行操作
        //数据在xhr.responseText
        console.log(xhr.responseText);
    }
}
```

SSE

SSE是一种允许服务端向客户端推送新数据的HTML5技术。它是 WebSocket 的一种轻量代替方案，使用 HTTP 协议。

严格地说，HTTP 协议是没有办法做服务器推送的，但是**当服务器向客户端声明接下来要发送流信息时，客户端就会保持连接打开**，SSE 使用的就是这种原理。

与由客户端每隔几秒从服务端轮询拉取新数据相比，这是一种更优的解决方案。

应用场景：例如邮箱服务的新邮件提醒，微博的新消息推送、管理后台的一些操作实时同步等.

```
var source=new EventSource("myevent");
source.onmessage=function(event){
    console.log(event.data);
};
source.onerror=function(){
    console.log("失败，连接状态"+source.readySate)
};
```

EventSource对象参数为入口点，必须与创建对象的页面同源(url模式，域、端口)。连接断开会自动建立,或者使用source.close()强制断开。open事件在连接建立时触发，message事件在接收到新数据时触发，error事件在无法建立连接时触发。推送数据保存在event.data中。

WebSocket

Websocket是一个全新的、独立的协议，基于TCP协议，与http协议兼容、却不会融入http协议。他被设计出来的目的就是要取代轮询和 Comet 技术。

WebSocket通过**单个TCP连接提供全双工（双向通信）通信信道**的计算机通信协议。此WebSocket API可在用户的浏览器和服务器之间进行双向通信。用户可以向服务器发送消息并接收事件驱动的响应，而无需轮询服务器。 它可以让**多个用户连接到同一个实时服务器，并通过API进行通信并立即获得响应。**

它**允许用户和服务器之间的流连接，并允许即时信息交换。**在聊天应用程序的示例中，通过套接字汇集消息，可以实时与一个或多个用户交换，具体取决于谁在服务器上“监听”（连接）。

WebSockets适用于**需要实时更新和即时信息交换**的任何应用程序。一些示例包括但不限于：现场体育更新，股票行情，多人游戏，聊天应用，社交媒体等。

```
var socket=new WebSocket("url");
socket.send("hello world");
socket.onmessage=function(event){
    console.log(event.data);
    console.log(event.readyState);
}
```

请求方式变化

jQuery ajax

传统 Ajax 指的是 XMLHttpRequest（XHR）， 最早出现的发送后端请求技术，隶属于原始js中，核心使用XMLHttpRequest对象，多个请求之间如果有先后关系的话，就会出现**回调地狱。**
JQuery ajax 是对原生XHR的封装，除此以外还增添了对**JSONP**的支持。

```
$.ajax({
   type: 'POST',
   url: url,
   data: data,
   dataType: dataType,
   success: function () {},
   error: function () {}
});
```

缺点：

- 本身是针对MVC的编程,不符合现在前端**MVVM**的浪潮
- 基于原生的XHR开发，XHR本身的架构不清晰。
- JQuery整个项目太大，单纯使用ajax却要引入整个JQuery非常的不合理（采取个性化打包的方案又不能享受CDN服务）
- 不符合关注分离（Separation of Concerns）的原则
- 配置和调用方式非常混乱，而且基于事件的异步模型不友好。

fetch

fetch

```
try {
  let response = await fetch(url);
  let data = response.json();
  console.log(data);
} catch(e) {
  console.log("Oops, error", e);
}
```

fetch号称是AJAX的替代品，是在ES6出现的，使用了ES6中的promise对象。Fetch是基于promise设计的。Fetch的代码结构比起ajax简单多了，参数有点像jQuery ajax。但是，**fetch不是ajax的进一步封装，而是原生js，没有使用XMLHttpRequest对象。**

优点：

- 符合关注分离，没有将输入、输出和用事件来跟踪的状态混杂在一个对象里
- 语法简洁，更加语义化
- 基于标准 Promise 实现，支持 async/await
- 同构方便，使用 isomorphic-fetch
- 脱离了XHR，是ES规范里新的实现方式
- 更加底层，提供的API丰富（request, response）

缺点：

- fetch只对网络请求报错，对400，500都当做成功的请求，服务器返回 400，500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。

- fetch默认不会带cookie，需要添加配置项： fetch(url, {credentials: 'include'})
- fetch不支持abort，不支持超时控制，使用setTimeout及Promise.reject的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费
- fetch没有办法原生监测请求的进度，而XHR可以

axios

```
axios({
    method: 'post',
    url: '/user/12345',
    data: {
        firstName: 'Fred',
        lastName: 'Flintstone'
    }
})
.then(function (response) {
    console.log(response);
})
.catch(function (error) {
    console.log(error);
});
```

axios 是一个基于Promise 用于浏览器和 nodejs 的 HTTP 客户端，本质上也是对原生XHR的封装，只不过它是Promise的实现版本，符合最新的ES规范。

特点：

- 从浏览器中创建 XMLHttpRequest
- 支持 Promise API
- 客户端支持防止CSRF
- 提供了一些并发请求的接口（重要，方便了很多的操作）
- 从 node.js 创建 http 请求
- 拦截请求和响应
- 转换请求和响应数据
- 取消请求
- 自动转换JSON数据

跨域

跨域的原因是什么?

当一个资源从与该资源本身所在服务器中不同域、协议、端口请求一个资源时，出于安全原因，浏览器限制从脚本内发起的跨源HTTP请求，XMLHttpRequest和Fetch API。

引入这个机制主要是用来防止CSRF攻击的（利用用户的登录态发起恶意请求）。

没有同源策略的情况下，A 网站可以被任意其他来源的 Ajax 访问到内容。如果你当前 A 网站还存在登录态，那么对方就可以通过 Ajax 获得你的任何信息。(当然跨域并不能完全阻止 CSRF)

解决跨域方式

JSONP

JSONP 的原理很简单，就是利用 `<script>` 标签没有跨域限制的漏洞。

通过 `<script>` 标签指向一个需要访问的地址并提供一个回调函数来接收数据当需要通讯时。

```
<script src="http://domain/api?param1=a&param2=b&callback=jsonp"></script>
<script>
    function jsonp(data) {
        console.log(data)
    }
</script>
```

JSONP 使用简单且兼容性不错，但是只限于 `get` 请求。

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，以下是简单实现:

```
function jsonp(url, jsonpCallback, success) {
  let script = document.createElement('script')
  script.src = url
  script.async = true
  script.type = 'text/javascript'
  window[jsonpCallback] = function(data) {
    success && success(data)
  }
  document.body.appendChild(script)
}
jsonp('http://xxx', 'callback', function(value) {
  console.log(value)
})
```

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。

虽然设置 CORS 和前端没什么关系，但是通过这种方式解决跨域问题的话，会在发送请求时出现两种情况，分别为**简单请求和预检请求。**

另外，规范要求，对那些可能对服务器数据产生副作用的 HTTP 请求方法（特别是 GET 以外的 HTTP 请求，或者搭配某些 MIME 类型的 POST 请求），**浏览器必须首先使用 OPTIONS 方法发起一个预检请求（preflight request），从而获知服务器是否允许该跨域请求。**

服务器确认允许之后，才发起实际的HTTP请求。 在预检请求的返回中，服务器端也可以通知客户端，是否需要携带身份凭证（包括 cookie 和 HTTP 认证相关数据）。

如果发起请求时设置WithCredentials标志设置为 true，从而向服务器发送cookie，但是如果服务器的**响应中未携带Access-Control-Allow-Credentials: true，浏览器将不会把响应内容返回给请求的发送者。**

对于附带身份凭证的请求，服务器不得设置 Access-Control-Allow-Origin的值为`*`，必须是某个具体的域名。

注意，简单 GET 请求不会被预检；如果此类带有身份凭证请求的响应中不包含该字段，这个响应将被忽略掉，并且浏览器也不会将相应内容返回给网页。

document.domain

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了。

postMessage

通常用于获取嵌入页面中的第三方页面数据。一个页面发送消息，另一个页面判断来源并接收消息。

```
// 发送消息端
window.parent.postMessage('message', 'http://test.com')
// 接收消息端
var mc = new MessageChannel()
mc.addEventListener('message', event => {
  var origin = event.origin || event.originalEvent.origin
  if (origin === 'http://test.com') {
    console.log('验证通过')
  }
})
```

iframe的跨域

1. 用iframe的获取方式： 只能显示，不能控制

   a.location.hash

   b.window.name

2. iframe+postMessage

   window.postMessage(data,origin)

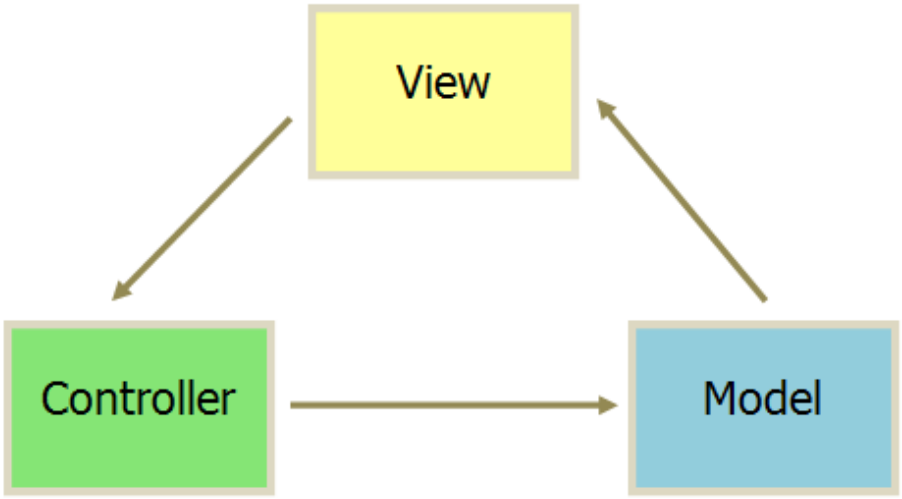
   子页面向父页面传递数据，则在子页面中调用父级window的postMessage

   window.parent.postMessage=function(data,origin)

框架入门

MVC、MVP与MVVM

MVC



M（Model）：数据保存

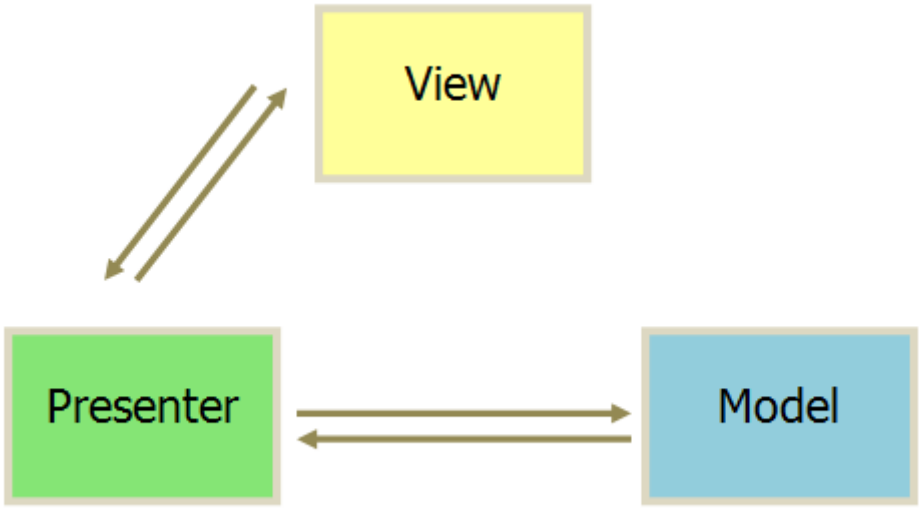
V（View）：用户页面

C（Controller）：业务逻辑

所有通信都是单向的。

1. View传指令到Controller。
2. Controller完成业务逻辑后，要求Model改变状态。
3. Model将新的数据发送到View，用户得到反馈。

MVP



M（Model）是业务逻辑层，主要负责数据，网络请求等操作

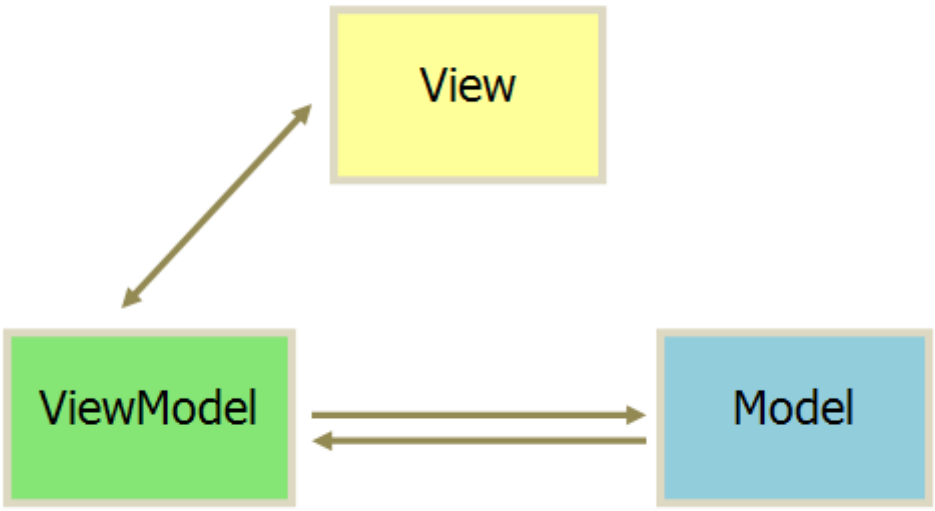
V（View）是视图层，负责绘制UI元素、与用户进行交互

P（Presenter）是View与Model交互的中间纽带，处理与用户交互的逻辑

MVP模式将Controller改名为Presenter，同时改变了通信方向。

1. 各部分之间的通信，都是双向的。
2. View与Model不发生联系，都通过Presenter传递。
3. View非常薄，不部署任何业务逻辑，称为“被动视图”，即没有任何主动性，而Presenter非常厚，所有业务逻辑都部署在那里。

MVVM



Model层代表数据模型，也可以在Model中定义数据修改和操作的业务逻辑；View代表UI组件，它负责将数据模型转换成UI展现出来，ViewModel是一个同步View和Model的对象。

在MVVM架构下，View和Model之间并没有直接的联系，而是通过ViewModel进行交互，Model和ViewModel之间的交互是双向的，因此View数据的变化会同步到Model中，而Model数据的变化也会立即反应到View上。

ViewModel通过**双向数据绑定**把View层和Model层连接了起来，而View和Model之间的同步工作完全是自动的，无需人为干涉，因此开发者只需关注业务逻辑，不需要手动操作DOM，不需要关注数据状态的同步问题，复杂的数据状态维护完全由MVVM来统一管理。

对于 MVVM 来说，其实最重要的并不是通过双向绑定或者其他的方式将 View 与 ViewModel 绑定起来，**而是通过 ViewModel 将视图中的状态和用户的行为分离出一个抽象，这才是 MVVM 的精髓。**

虚拟DOM

虚拟DOM初探

相较于 DOM 来说，操作 JS 对象会快很多，并且我们也可以通过 JS 来模拟 DOM

```
const ul = {
  tag: 'ul',
  props: {
    class: 'list'
  },
  children: {
    tag: 'li',
    children: '1'
  }
}
```

上述代码对应的 DOM 就是

```
<ul class='list'>
  <li>1</li>
</ul>
```

那么既然 DOM 可以通过 JS 对象来模拟，反之也可以通过 JS 对象来渲染出对应的 DOM。

当然了，通过 JS 来模拟 DOM 并且渲染对应的 DOM 只是第一步，难点在于如何判断新旧两个 JS 对象的**最小差异**并且实现**局部更新** DOM。

这就需要Diff算法了。

虚拟DOM真的能提升性能吗?

使用虚拟 DOM，**在DOM 阶段操作少了通讯的确是变高效了，但代价是在 JS 阶段需要完成额外的工作**（diff计算），这项额外的工作是需要耗时的!

虚拟DOM**并不是说比原生DOM API的操作快，而是说不管数据怎么变化，都可以以最小的代价来进行更新 DOM。**在每个点上，其实用手工的原生方法会比diff好很多。比如说仅仅是修改了一个属性，需要整体重绘吗？显然这不是虚拟DOM提出来的意义。框架的意义在于掩盖底层的 DOM 操作，用更声明式的方式来描述，从而让代码更容易维护。

diff算法

首先 DOM 是一个多叉树的结构，如果需要完整的对比两颗树的差异，那么需要的时间复杂度会是 O(n ^ 3)，这个复杂度肯定是不能接受的。于是 React 团队优化了算法，实现了 O(n) 的复杂度来对比差异。实现 O(n) 复杂度的关键就是只对比同层的节点，而不是跨层对比，这也是考虑到在实际业务中很少会去跨层的移动 DOM 元素。 所以判断差异的算法就分为了两步

- 首先从上至下，从左往右遍历对象，也就是树的深度遍历，这一步中会给每个节点添加索引，便于最后渲染差异
- 一旦节点有子元素，就去判断子元素是否有不同

在第一步算法中，需要判断新旧节点的 `tagName` 是否相同，如果不相同的话就代表节点被替换了。如果没有更改 `tagName` 的话，就需要判断是否有子元素，有的话就进行第二步算法。

在第二步算法中，需要判断原本的列表中是否有节点被移除，在新的列表中需要判断是否有新的节点加入，还需要判断节点是否有移动。

举个例子来说，假设页面中只有一个列表，我们对列表中的元素进行了变更

```
// 假设这里模拟一个 ul，其中包含了 5 个 li
[1, 2, 3, 4, 5]
// 这里替换上面的 li
[1, 2, 5, 4]
```

从上述例子中，我们一眼就可以看出先前的 `ul` 中的第三个 `li` 被移除了，四五替换了位置。

那么在实际的算法中，我们如何去识别改动的是哪个节点呢？这就引入了 `key` 这个属性。这个属性是用来给每一个节点打标志的，用于判断是否是同一个节点。

当然在判断以上差异的过程中，我们还需要判断节点的属性是否有变化等等。

当我们判断出以上的差异后，就可以把这些差异记录下来。当对比完两棵树以后，就可以通过差异去局部更新 DOM，实现性能的最优化。

双向绑定

利用`Object.defineProperty()`对数据进行劫持，设置一个监听器`Observer`，用来监听所有属性，如果属性上发生变化了，就需要告诉订阅者`Watcher`去更新数据，最后指令解析器`Compile`解析对应的指令，进而会执行对应的更新函数，从而更新视图，实现双向绑定。

前端路由

前端路由的本质是**监听 URL 的变化**，然后匹配路由规则，显示相应的页面，并且无须刷新页面。目前前端使用的路由就只有两种实现方式

- Hash 模式
- History 模式

Hash 模式

`www.test.com/#/` 就是 Hash URL，当 `#` 后面的哈希值发生变化时，可以通过 `hashchange` 事件来监听到 URL 的变化，从而进行跳转页面，并且无论哈希值如何变化，服务端接收到的 URL 请求永远是 `www.test.com`。

```
window.addEventListener('hashchange', () => {
  // ... 具体逻辑
})
```

Hash 模式相对来说更简单，并且兼容性也更好。

History 模式

History 模式是 HTML5 新推出的功能，主要使用 `history.pushState` 和 `history.replaceState` 改变 URL。

通过 History 模式改变 URL 同样不会引起页面的刷新，只会更新浏览器的历史记录。

```
// 新增历史记录
history.pushState(stateObject, title, URL)
// 替换当前历史记录
history.replaceState(stateObject, title, URL)
```

当用户做出浏览器动作时，比如点击后退按钮时会触发 `popState` 事件

```
window.addEventListener('popstate', e => {
  // e.state 就是 pushState(stateObject) 中的 stateObject
  console.log(e.state)
})
```

两种模式对比

- Hash 模式只可以更改 `#` 后面的内容，History 模式可以通过 API 设置任意的同源 URL
- History 模式可以通过 API 添加任意类型的数据到历史记录中，Hash 模式只能更改哈希值，也就是字符串
- Hash 模式无需后端配置，并且兼容性好。History 模式在用户手动输入地址或者刷新页面的时候会发起 URL 请求，后端需要配置 `index.html` 页面用于匹配不到静态资源的时候.

模块化

CommonJS

一个单独的文件就是一个模块，主要运行与服务器端，同步加载模块。

require输入其他模块提供的功能

module.exports规范模块对外接口，输出一个值的拷贝。

输出之后不能改变，会缓存起来。

```
// moduleA.js
var name = 'weiqinl'
function foo() {}

module.exports = exports = {
name,
foo
}

// moduleB.js
var ma = require('./moduleA') // 可以省略后缀.js
exports.bar = function() {
ma.name === 'weiqinl' // true
ma.foo() // 执行foo方法
}

// moduleC.js
var mb = require('./moduleB')
mb.bar()
```

AMD

异步加载，一个单独文件一个模块，主要运行于浏览器端，模块和模块的依赖可以被异步加载。

define定义模块.

require用于输入其他模块提供的功能.

return规范模块对外接口。

define.amd是一个对象，表明函数遵守AMD规范。AMD的运行逻辑是，提前加载，提前执行，申明依赖模块的时候，会第一时间加载并执行模块内的代码，使后面的回调函数能在所需的环境中运行。

```
// moduleA.js
define(['jQuery','lodash'], function($, _) {
var name = 'weiqinl',
function foo() {}
return {
name,
foo
}
})

// index.js
require(['moduleA'], function(a) {
a.name === 'weiqinl' // true
a.foo() // 执行A模块中的foo函数
// do sth...
})

// index.html
<script src="js/require.js" data-main="js/index"></script>
```

CMD

通用模块。一个文件一个模块。主要在浏览器中运行，define全局函数，定义模块，通过exports向外提供接口，用require获取接口，使用某个组件时用use()调用。通过require引入的模块，只有当程序运行到这里时候才会加载执行。

```
// moduleA.js
// 定义模块
define(function(require, exports, module) {
var func = function() {
var a = require('./a') // 到此才会加载a模块
a.func()
if(false) {
var b = require('./b') // 到此才会加载b模块
b.func()
}
}
// do sth...
exports.func = func;
})

// index.js
// 加载使用模块
seajs.use('moduleA.js', function(ma) {
var ma = math.func()
})

// HTML，需要在页面中引入sea.js文件。
<script src="./js/sea.js"></script>
<script src="./js/index.js"></script>
```

UMD

通用模块。解决commonJS和AMD不能通用的问题。define.amd存在，执行AMD规范；module.exports,执行CommonJS规范；都没有，原始代码规范。

```
// 使用Node, AMD 或 browser globals 模式创建模块
(function (root, factory) {
if (typeof define === 'function' && define.amd) {
// AMD模式. 注册为一个匿名函数
define(['b'], factory);
} else if (typeof module === 'object' && module.exports) {
// Node等类CommonJS的环境
module.exports = factory(require('b'));
} else {
// 浏览器全局变量 (root is window)
root.returnExports = factory(root.b);
}
}(typeof self !== 'undefined' ? self : this, function (b) {
// 以某种方式使用 b

//返回一个值来定义模块导出。(即可以返回对象，也可以返回函数)
return {};
}));
```

ES6 Module

ES6模块功能主要由两个命令构成：`import`和`export`。`import`命令用于输入其他模块提供的功能。`export`命令用于规范模块的对外接口。

```
// 输出变量
export var name = 'weiqinl'
export var year = '2018'

// 输出一个对象（推荐）
var name = 'weiqinl'
var year = '2018'
export { name, year}


// 输出函数或类
export function add(a, b) {
return a + b;
}

// export default 命令
export default function() {
console.log('foo')
}

// 正常命令
import { name, year } from './module.js' //后缀.js不能省略

// 如果遇到export default命令导出的模块
import ed from './export-default.js'
```

设计模式

创建型设计模式

简单工厂模式

又叫静态工厂法，由一个工厂对象决定创建某一种产品对象类的实例。

网络

OSI七层模型

应用层： 为应用程序提供服务，并管理应用程序之间的通信（SMTP、HTTP、FTP）
表示层： 处理数据的标识问题，比如编码、格式转化、加密解密等
会话层： 负责建立管理和断开通信连接，实现数据同步
传输层： 端到端传输数据，同时处理传输错误、控制流量等（TCP UDP）
网络层： 地址管理、路由选择（IP协议）
数据链路层： 数据分割成帧，mac寻址、差错校验、信息纠正等。（以太网）
物理层： 利用传输介质为数据链路层提供物理连接

发送端从应用层 → 物理层 打包发送
接收层从物理层 → 应用层 解析获取

路由器工作在哪一层

网络层。

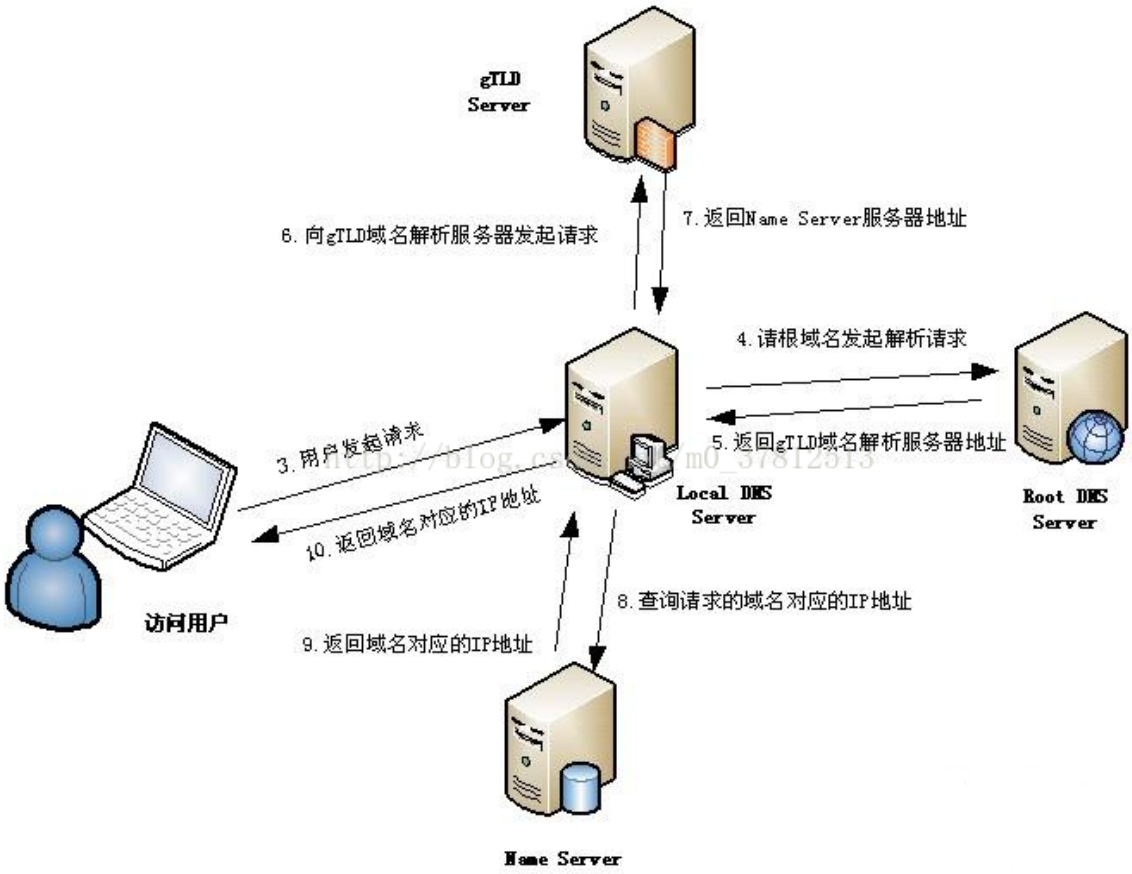
路由器是连接因特网中各局域网、广域网的设备，它会根据信道的情况自动选择和设定路由，以最佳路径，先后顺序发送信号。

路由和交换机最主要的区别就是**交换机发生在OSI参考模型第二层（数据链路层）**，而**路由发生在第三层（网络层）**。

这一区别决定了路由和交换机在移动信息的过程中需要使用不同的控制信息，所以两者实现各自功能的方式是不同的。

DNS解析过程

- **根DNS服务器**：返回顶级域名 DNS 服务器的 IP 地址
- **顶级域DNS服务器**：返回权威域名 DNS 服务器的 IP 地址
- **权威DNS服务器**：返回相应主机的 IP 地址



以输入www.google.com为例，
1.浏览器检查自身缓存，有无解析此域名对应的ip
2.操作系统缓存hosts文件中查询
3.没有的话，请求本地域名服务器（LDNS）解析域名（一般在城市某处，距离不会太远）
4.如果还没有的话，就去根DNS域服务器查询，此时会给出.com的顶级域名服务器
5.然后去.com服务器查询，此时会给出这个域名google.com的地址，这是网站注册的域名服务器
6.去NameServer查询，根据映射关系表找到目标IP,返回给LDNS（LDNS缓存域名及IP）
7.LDNS解析结果返回用户（缓存到系统缓存中），域名解析结束

TCP与UDP

UDP 与 TCP 的区别是什么?

UDP 协议 是面向无连接的，不需要在正式传递数据之前先连接起双方。 UDP 协议只是数据报文的搬运工，不保证有序且不丢失的传递到对端，并且UDP 协议也没有任何控制流量的算法，总的来说 UDP 相较于 TCP 更加的轻便。一般可以用于直播、即时通讯、即时游戏等。

TCP 无论是建立连接还是断开连接都需要先需要进行握手。在传输数据的过程中，通过各种算法保证数据的可靠性，当然带来的问题就是相比 UDP 来说不那么的高效。

TCP建立连接--三次握手

起初，两端都为 **CLOSED** 状态。在通信开始前，双方都会创建 TCB。 服务器创建完 TCB 后便进入 **LISTEN** 状态，此时开始等待客户端发送数据。

第一次握手

客户端向服务端发送连接请求报文段。该报文段中包含自身的数据通讯初始序号。请求发送后，客户端便进入 **SYN-SENT** 状态。

第二次握手

服务端收到连接请求报文段后，如果同意连接，则会发送一个应答，该应答中也会包含自身的数据通讯初始序号，发送完成后便进入 **SYN-RECEIVED** 状态。

第三次握手
当客户端收到连接同意的应答后，还要向服务端发送一个确认报文。客户端发完这个报文段后便进入 **ESTABLISHED** 状态，服务端收到这个应答后也进入 **ESTABLISHED** 状态，此时连接建立成功。

为什么 TCP 建立连接需要三次握手，明明两次就可以建立起连接

防止出现失效的连接请求报文段被服务端接收的情况，从而产生错误。
客户端发送了一个连接请求 A，但是因为网络原因造成了超时，这时 TCP 会启动超时重传的机制再次发送一个连接请求B。此时请求顺利到达服务端，服务端应答完就建立了请求，然后接收数据后释放了连接。
假设这时候连接请求 A 在两端关闭后终于抵达了服务端，那么此时服务端会认为客户端又需要建立 TCP 连接，从而应答了该请求并进入 **ESTABLISHED** 状态。但是客户端其实是 **CLOSED** 的状态，那么就会导致服务端一直等待，造成资源的浪费。

TCP断开连接--四次握手

TCP 是全双工的，在断开连接时两端都需要发送 **FIN** 和 **ACK**。
第一次握手
若客户端 A 认为数据发送完成，则它需要向服务端 B 发送连接释放请求。
第二次握手
B收到连接释放请求后，会告诉应用层要释放 TCP 链接。然后会发送 ACK 包，并进入 **CLOSE_WAIT** 状态，此时表明 A 到 B 的连接已经释放，不再接收 A 发的数据了。但是因为 TCP 连接是双向的，所以 B 仍旧可以发送数据给 A。
第三次握手
B如果此时还有没发完的数据会继续发送，完毕后会向 A 发送连接释放请求，然后 B 便进入 **LAST-ACK** 状态。
第四次握手
A 收到释放请求后，向 B 发送确认应答，此时 A 进入 **TIME-WAIT** 状态。该状态会持续 2MSL（最大段生存期，指报文段在网络中生存的时间，超时会被抛弃） 时间，若该时间段内没有 B 的重发请求的话，就进入 **CLOSED** 状态。当 B 收到确认应答后，也便进入 **CLOSED** 状态。

PS：通过延迟确认的技术（通常有时间限制，否则对方会误认为需要重传），可以将第二次和第三次握手合并，延迟 ACK 包的发送。

为什么客户端 A 要进入 TIME-WAIT 状态，等待 2MSL 时间后才进入 CLOSED 状态?

为了保证 B 能收到 A 的确认应答。若 A 发完确认应答后直接进入 CLOSED 状态，如果确认应答因为网络问题一直没有到达，那么会造成 B 不能正常关闭。

TCP如何解决数据丢包或报文顺序不对的问题?

TCP有**ARQ**超时重传机制。

- 一种是**停止等待ARQ**。
  比如说A向B发送一个报文，同时启动定时器，如果超时就重新发送。B如果收到相同序号的报文就会丢弃重新应答。A接受相同序号应答也丢弃。
  这种是单个单个传的，效率就比较低，但也不会有那种多个丢包的情况。
- 另一种，高效一点，**连续ARQ**。
  它用的窗口，A持续发这个窗口内的数据，B累积确认，收到多个后统一应答A，ack标志告诉A，这个序号之前的数据已经收到了。但是如果A收到3个重复的ack，那就说明有失序或丢包的情况，就会启用快速重传/快速恢复。
  快速重传TCP taho ，阈值设为当前窗口一半，窗口设为1开始慢开始，重新传送。
  快速恢复TCP Remo，机制是窗口减半，阈值为当前窗口，启用拥堵避免。不过，它是重发接收端

要的包，接受收到一个Ack就退出，如果丢了很多个包就尬住再3ack触发一遍。
因此快恢复进行了优化——**TCP New Reno**。它是区别在于它记下了这个发送段的最大序号，并且每次都比对。
比如说1-10，丢包丢了4，7。最大序号就是10。接受方发的ack包是4，发送方发4，接受方收到4，发7。那么发送方就会对比7和10，知道是丢了多个包，发7。接受方收到，发11，发送方收到11后，对比10，比10大就退出快恢复阶段了。

TCP如何实现流量控制的

通过滑动窗口和拥堵窗口实现的。
滑动窗口主要是用于接收方，保证接收方能够接受数据。接收方通过报文告知发送方当前接收窗口剩余大小，发送窗口根据该值变化大小滑动窗口（待发送区）发送报文。
拥堵窗口，主要用于网络，防止过多的数据拥堵网络，避免负载过大的情况。

Http协议

Post和Get有什么区别?

从用法上说，Post一般用于无副作用、幂等的场景；Post多用于有副作用、不幂等的情况。

> 冥等的定义：发送M和N次请求，服务器上资源状态一致。比如说，注册10个账号和11个账号是不冥等的，对文章进行了10次11次修改是幂等的，因为前者多了一个账号（资源），后者是更新同一个资源。
>
> 副作用的定义：副作用是指对服务器上资源做改变。比如搜索是无副作用的，但更新是有副作用的。

从本质上说，Post和Get都取决于http，使用哪个方法与应用层传输没有必然的联系。HTTP没有要求，如果是POST，数据就要放在BODY中。也没有要求GET，数据（参数）就一定要放在URL中而不能放在BODY中。

细节上有一些区别：

Get能请求缓存，但是Post不可以
Post支持更多编码类型
Get回退无害，Post会再次提交
Get能被保存为书签，Post不可以
由于浏览器Url有限制，所以Get的长度受限，但Post不受限（因为都在Body里）

Http常用首部

- 通用
  - **cache-control**： 控制缓存行为
  - **connection**： 连接的性质，比如keep-alive
  - **user-Agent** ： 用户信息
  - **Date**： 报文创建时间
- 请求
  - **Referrer Policy** : 表示来源的（浏览器所访问的前一个页面）,可以用于辅助检测crsf攻击，一般浏览器的默认值是**no-referrer-when-downgrade**，意思是https降级http的时候不传原地址。
  - **Accept** : 能正确接收的媒体类型
  - **Accept-XX**(Accept-Charset/Accept-Encoding/Accept-Language):能正确接收的xx
  - **Expect** :期待服务端的指定行文
  - **If-Match** :两端资源标记比较

    - **If-Modified-Since** : 比较时间 未修改返回304 Not Modified
    - **If-None-Match** :比较标记 未修改返回304 Not Modified
- 响应
    - **Location** : 重定向到某个location
    - **Server** : 服务器名字
    - **Age** :响应存在时间
    - **Accept-Ranges** ： 可以接受的范围类型

http请求中`connection=keep-alive`的意义在哪里

HTTP 是基于 TCP 的，每一个 HTTP 请求都需要进行三次握手。如果一个页面对某一个域名有多个请求，就会进行频繁的建立连接和断开连接。所以HTTP 1.0 中出现了`Connection: keep-alive`，用于建立长连接。Keep-Alive 模式更加高效，因为避免了连接建立和释放的开销。但是，长时间的TCP连接容易导致系统资源无效占用，配置不当的keep-alive有时比重复利用连接带来的损失还更大。所以，正确设置keep-alive timeout时间非常重要。

http请求中cache-control有哪些参数可以设置

Public :表示任何缓存都可以缓存响应

private :表示响应仅供单个用户使用，不得由共享高速缓存存储。私有缓存可以存储响应。

no-cache : 强制缓存在发布缓存副本之前将请求提交到源服务器以进行验证。

no-store: 缓存不应存储有关客户端请求或服务器响应的任何内容。

http状态码

- **1XX: 通知**
    - **100 Continue** 客户端应重新发请求
    - **101 Switching Protocols** 改用协议 http换到https或者http1.1换到2.0之类
- **2XX：成功**
    - **200 OK** 操作成功
    - **201 Created**按照客户端请求创建了一个新资源
    - **202 Accepted** 请求无法或不被实时处理
    - **204 No Content** 请求成功，但是报文不含实体的主体部分
    - **205 Reset Content** 请求成功，但是报文不含实体主体部分，要求客户端重置内容
    - **206 Partial Content** 进行范围请求
- **3XX：重定向**
    - **301 Moved Permanently**永久性重定向，资源已经被分配到了新的URL
    - **302 Found** 临时重定向，资源临时分配了URL 实际上发部分客户端把它当成303处理
    - **303 See Other** 表示资源存在另一个URL。应用Get获取资源
    - **304 Not Modified** 允许访问资源，但实体主体为空（客户端已经有此数据，不需要再次发送）
    - **307 Temporary Redirect** 临时重定向，资源临时分配了URL，但是希望客户端能够保持方法不变请求新地址（解决302被当成303处理的问题）
- **4XX：客户端错误**
    - **400 Bad Request** 请求报文语法错误
    - **401 Unauthorized** 发送的请求需要通过验证，客户端试图对一个受保护的资源操作但没有认证证书
    - **403 Forbidden** 请求资源存在但被拒绝，常用于一个资源只允许在特定时间段内访问（如果不想透露可以谎报404）
    - **404 Not Found** 找不到请求的资源

    - **405 Method Not Allowed** 不支持的请求方法，比如只支持Get，但是收到了Post请求
- **5XX：服务端错误**
    - **500 Internal Server Error** 执行请求时发生错误（处理异常）
    - **501 Not Implemented** 不支持此请求方法（和405区别在于，405是访问的资源不支持，而501表示服务器不能操作此方法）
    - **502 Bad Gateway** 代理与上行服务器之间出现问题
    - **503 Service Unavailable** 服务器暂时处于超负荷或者维护中

状态码返回204可能的原因

- 服务器拒绝请求返回
- Get资源存在但表示是空的
  服务器通过这个响应代码告诉客户端：客户端的输入已被接受，但客户端不应该改变任何UI元素

状态码204和205的区别

204和205的区别在于**205要求了重置**!
用一个表单为例，如果提交后返回204，那么表单里的各个字段值不变，可以继续修改它们；但假如得到的响应代码205，那么表单里的各个字段将被重置为它们的初始值。

状态码304的可能情况

这个与浏览器的协商缓存有关。用时间的来说明：

当用户第一次请求资源A时，服务器会添加一个名为`Last-Modified`的响应头，这个头说明了A的 **最后修改时间** ，浏览器会把A的内容以及最后的响应时间缓存下来。

当用户第二次请求A时，在请求中包含一个名为`If-Modified-Since`请求头，它的值就是第一次请求时服务器通过`Last-Modified`响应头发送给浏览器的值，即资源A最后的修改时间。

`If-Modified-Since`请求头就是在告诉浏览器，我这里浏览器缓存的A最后修改时间是这个，你看看现在A最后修改时间是不是这个，如果还是，那么就不用响应这个请求了，我会把缓存里的内容直接显示出来。

服务器会获取`If-Modified-Since`值，与A**当前的最后修改时间**作对比，如果相同，则服务器返回304状态码，表示A与浏览器上次缓存的相同，无需再次发送，浏览器可以显示自己的缓存页面，如果比对不同，那么说明A已经改变，服务器会返回200状态码。

Http协议中的长短连接和长短轮询

- 短连接

  所谓短连接，即连接只保持在数据传输过程，请求发起，请求建立，数据返回，连接关闭。它适用于一些实时数据请求，配合轮询来进行新旧数据的更替。

- 长连接

  长连接便是在连接发起后，在请求关闭连接前客户端与服务器都保持连接，实质是保持这个通信管道，之后便可以对其进行复用。

- 短轮询

  短轮询指的是在循环周期内，不断发起请求，每一次请求都立即返回结果，根据新旧数据对比决定是否使用这个结果。

- 长轮询

  长轮询是在请求的过程中，若是服务器端数据并没有更新，那么则将这个连接挂起，直到服务器推送新的数据，再返回，然后再进入循环周期。

长短连接和长短轮询的区别

1. 决定方式。一个TCP连接是否为长连接，是通过设置HTTP的Connection Header来决定的，而且是需要两边都设置才有效。而一种轮询方式是否为长轮询，是根据服务器端的处理方式来决定的，与客户端没有关系。
2. 实现方式。连接的长短是通过协议来规定和实现的。而轮询的长短，是服务器通过编程的方式手动挂起请求来实现的。

Https与Http

Http为什么不安全?

1. 数据以明文传递，有被窃听的风险
2. 接收到的报文无法证明是发送时的报文，不能保证完整性，因此报文有被篡改的风险
3. 不验证通信两端的身份，请求或响应有被伪造的风险

http和https有什么区别?

1. 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
2. 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**
3. HTTPS协议需要CA申请证书，一般免费证书比较少，因而需要一定费用
4. HTTP的连接很简单，是无状态的；HTTPS协议是由**TSL**协议进行了加密，比HTTP协议安全

TLS

https仍用http传输信息，但信息通过TLS进行了加密。

TLS作用于表示层

两种加密技术

- 对称加密

  两边有相同的密钥，都知道如何加密解密

- 非对称加密

  数据公钥加密，私钥解密，私钥只有发出公钥的一方知道

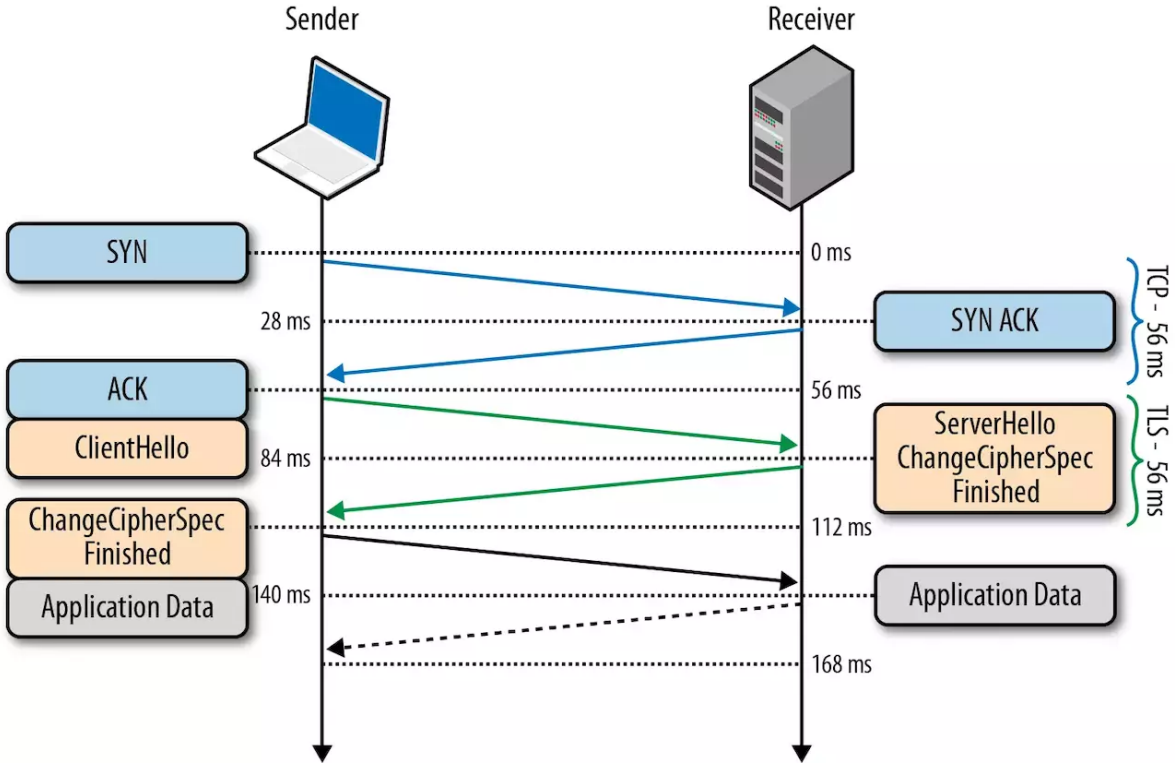
对称加密问题在于如何让双方都知道密码，且不被其他人知道；非对称加密可以完美解决对称加密存在的问题

流程如下：

服务器将公钥发散出去。之后客户端创建一个密钥，用此公钥加密后发送给服务器，服务器用私钥解密，就能得知此密钥。双方都知道密码，之后就可以采用对称加密的方式进行数据传输了。

TSL的握手过程

原理讲完了，那么**TSL的握手过程**就很清晰了



1.客户端发送一个随机值**ClientHello**，并附带上支持的协议和支持的加密方式

2.服务端收到随机值**ClientHello**，产生一个随机值**ServerHello**，根据客户端需求的协议返回，从客户端支持的堆成加密算法中选一个作为最终通讯的算法，并发送自己的CA证书

3.客户端收到服务器CA证书、协商的通讯加密算法、随机值**ServerHello**，验证通过后再生成第三个随机值**预主密钥Pre-Master**，用CA证书公钥加密后发送给服务端（如果需要验证证书，还要附带上证书）。
用三个随机值+协商的加密算法，合成最终通讯的密钥 。
change_cipher_spec Finished 客户端握手结束通知。

4.服务端收到加密过的随机值之后，**私钥** 解密获得第三个随机值**预主密钥Pre-Master 。**
使用第1、2步中协商的对称加密算法+三个随机值合成最终的协商密钥 。
change_cipher_spec Finished 服务器握手结束通知。

5.此时两端都有了最终的协商密钥了，接来下的传输就使用这个密钥加密解密。

所以，在**TLS 握手阶段，两端使用非对称加密的方式来通信** ，但是因为非对称加密损耗的性能比对称加密大，所以**在正式传输数据时，两端传输其实是使用对称加密的方式通信 。**

Https的缺点?

1. 通信两端都需要进行加密和解密，会**消耗** 大量的CPU、内存等**资源** ，**增加了服务器的负载**
2. 加密运算和多次握手**降低了访问速度**
3. 在开发阶段，**加大了页面调试难度 。**由于信息都被加密了，所以用代理工具的话，需要先解密然后才能看到真实信息
4. 用HTTPS访问的页面，页面内的外部资源都得用HTTPS请求，包括脚本中的AJAX请求

HTTPS 的单向认证和双向认证

单向认证

1. 客户端保存着服务器的证书并信任该证书
2. https一般是单向认证，这样可以让绝大部分人都可以访问你的站点

双向认证

1. 先决条件是有2个或者2个以上的证书，一个服务器证书，其它是客户端证书
2. 服务器保存着客户端的证书并信任该证书，客户端保存着服务器的证书并信任该证书。这样，在证书验证成功的情况下即可完成请求响应
3. 双向认证一般用于企业应用对接（比如说堡垒机hh）

Http2.0

http1.0和http2.0的区别是什么?

http2.0是对http1.0的改进，相较于http1.0更快更高效

1http2.0实现了**多路复用**，用**一个TCP进行连接共享，一个请求对应一个id，这样就可以发送多个请求**，接收方通过id来响应不同的请求，解决了http1.0队首阻塞和连接过多的问题。因为http2.0**在同一域名不论访问多少文件都只有一个连接**，所以对服务器而言，提升的并发量是很大的。

2 http2.0引入了**二进制数据帧和流**的概念，**数据拆分成数据帧传输，并进行顺序标识，接收方收到数据后按序组合即可获取正确数据。**这样就可以**并行**传输了，解决了http1.0只能串行传输的问题。

3 http2.0**压缩头部**，使**用序号对头部编码，在两端备份索引表，通过对编码进行比较来判断是否需要传输，减少了需要传输的大小。**解决了http1.0中头部反复传输资源浪费的问题

4 http2.0中，服务器可以在客户端某个请求后，**主动推送一些客户端一定需要的资源** 。这样也能减少请求的数目。

当然http2.0也不是尽善尽美的，比如说在出现丢包的情况时，需要重新传输，后面的数据也就被阻塞了，但是http1.0因为有多个连接，所以不会影响其他连接的传输。这样的话http2.0的性能反倒不如http1.0了。

但这个是TCP的问题了，要说改TCP也不太实际。不过也有解决方案，基于QUIC协议的http3.0就解决了这个问题，需要简单描述一下吗?

QUIC的简单描述：多路复用（ID识别）、纠错机制。

新特性

1. **多路复用**

   通过一个TCP连接传输所有数据。一个请求对应一个id，这样一个链接上可以有多个请求，每个连接的请求可以随机的混杂在一起，接收方可以根据请求的id将请求再归属到各自不同的服务器端请求里面

2. **二进制分帧层**

   HTTP/2.0性能增强的关键，它改变了通信两端交互数据的方式，原先是以文本传输，现在要先对数据进行二进制编码，再把数据分成一个一个的帧，接着把帧送到数据流中，最后对方接受帧并拼接成一条消息，再处理请求

3. **首部压缩**

   前面提到过的HTTP1.x的header带有大量信息，而且每次都要重复发送，HTTP2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小

4. **服务器推送**

   HTTP2.0支持服务器主动推送，简单地说就是一次请求返回多个响应，这也是一减少HTTP请求的方法。服务器除了处理最初的请求外，还会额外push客户端一定会请求的资源，无需客户端发出明确的请求。

存在问题

http2.0使用了多路复用，一般来说同一域名下只需要使用一个TCP连接。
但是当连接中出现丢包时，整个TCP都要开始等待重传，后面的数据也都被阻塞了。而http1.0可以开启多个连接，只会影响一个，不会影响其他的。
所以**在丢包情况下，http2.0的情况反而不如http1.0。**

Http3.0

为了解决2.0丢包性能的问题，Google基于UDP提出了QUIC协议。

HTTP3.0中的底层支撑协议就是QUIC。所以http3.0也叫HTTP-over-QUIC。

QUIC协议

UDP协议高效，但不可靠。QUIC基于UDP，在原来的基础上结合了tcp和http的精华使它可靠。

特点：

多路复用

HTTP2虽然是多路复用，但是TCP协议是没有这个功能的。QUIC 原始就包含此功能，并且传输的单个数据流可以保证有序交付且不会影响其他数据流

其在移动端也会比TCP好，因为TCP基于IP+端口识别连接，不适合多变的网络环境，但是**QUIC是通过ID识别连接，不论网络如何变化，只要ID不变，就能迅速连上**（实时手游就是这样实现的）

纠错机制

假如说这次我要发送三个包，协议会算出这三个包的异或值并单独发出一个校验包，也就是总共发出了四个包。

当出现其中的非校验包丢包的情况时，可以通过另外三个包计算出丢失的数据包的内容。

当然这种技术只能使用在丢失一个包的情况下，如果出现丢失多个包就不能使用纠错机制了，只能使用重传的方式了。

0-RTT

通过使用类似 TCP 快速打开的技术，缓存当前会话的上下文，在下次恢复会话的时候，只需要将之前的缓存传递给服务端验证通过就可以进行传输了。

输入 URL 到页面渲染的整个流程

首先是**DNS解析**：

如果这一步做了智能 DNS 解析的话，会提供访问速度最快的 IP 地址回来，如果没有的话，会进行DNS迭代查询。

之后就是**TCP握手**：

TCP 的握手情况以及 TCP 的一些特性。

如果是https的话，还会 TCP 握手结束后就会进行 TLS 握手，然后就开始正式的传输数据。

应用层会下发数据给传输层，这里 TCP 协议会指明两端的端口号，然后下发给网络层。网络层中的 IP 协议会确定 IP 地址，并且指示了数据传输中如何跳转路由器。然后包会再被封装到数据链路层的数据帧结构中，最后就是物理层面的传输了。

数据在进入服务端之前，可能还会先经过负责负载均衡的服务器，它的作用就是将请求合理的分发到多台服务器上，这时假设服务端会响应一个 HTML 文件。

接收到响应的文件之后，**首先浏览器会判断状态码是什么**，如果是 200 那就继续解析，如果 400 或 500 的话就会报错，如果 300 的话会进行重定向

浏览器开始解析文件，如果是 gzip 格式的话会先解压一下，然后通过文件的编码格式知道该如何去解码文件。

文件解码成功后会正式开始**渲染流程**，先会根据 HTML 构建 DOM 树，有 CSS 的话会去构建 CSSOM 树。如果遇到 script 标签的话，会判断是否存在 async 或者 defer ，前者会并行进行下载并执行 JS，后者会先下载文件，然后等待 HTML 解析完成后顺序执行。如果以上都没有，就会阻塞住渲染流程直到 JS 执行完毕。

遇到文件下载的会去下载文件，这里如果使用 HTTP/2 协议的话会极大的提高多图的下载效率。

CSSOM 树和 DOM 树构建完成后会开始生成 Render 树，这一步就是确定页面元素的布局、样式等等诸多方面的东西

在生成 Render 树的过程中，浏览器就开始调用 GPU 绘制，合成图层，将内容显示在屏幕上了。

> 另外就是：
>
> `DOMContentLoaded`事件触发代表初始的HTML被完全加载和解析，不需要等待CSS，JS，图片加载。
>
> `Load`事件触发代表页面中的DOM，CSS，JS，图片已经全部加载完毕。

最后就是TCP断开连接，四次挥手。

> 客户端：我没有数据要发送了，准备挂了
>
> 服务器：收到，但我还有一些数据没发送完，稍等一下 ……
>
> 服务器：好了，发送完了，可以断开连接了
>
> 客户端：OK，你断开连接吧（内心独白：我将会在2倍的最大报文段生存时间后关闭连接，如果再收到服务器的消息，那么服务器就是没听到我最后这句话，我就再发送一遍）

浏览器

垃圾回收机制

垃圾收集机制的原理其实非常简单：
找出那些不再使用的变量，然后释放其占用的内存。为此，垃圾收集器会按照固定的时间间隔（或代码执行中预定的收集时间），周期性的执行这一操作。

标记清除

当变量进入环境（例如，在函数中声明一个变量）时，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，因为我们在这个环境中可能随时会用到它们。**当变量离开环境时，则将其标记为“离开环境”。**

引用计数

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，这个值的引用次数就是 1。如果同一个值又被赋值给另一个变量，则引用次数加 1。相反，如果包含对这个值的引用的变量有取了另一个值，则引用次数减 1。当这个值的引用次数变为 0 时，说明已经没法再访问这个值了，因此可以将其占用的内存回收了。

引用计数策略有一个很严重的问题：循环引用。所以不常用。
如果对象 A 中包含一个指针指向对象 B，而对象 B 中也包含一个指针指向对象 A。那么这两个对象引用次数都是 2，但实际上已经可以回收了。若这种函数被反复多次调用，会导致大量内存得不到回收。

新生代与老生代

新生代对象一般存活时间短，其内存空间分为两个部分，From空间和To空间，在这两个空间中，有一个是使用的，一个是空闲的。

新分配的对象放入From空间中，当Form占满时，新生代GC启动，检查Form对象存活对象复制到To空间，失活就销毁。复制完成后Form空间和To空间**互换。**

老生代对象一般存活时间长，数量多，用了两个算法：标记清楚算法和标记压缩法。**当新生代对象已经经历过一次GC算法了，将对象放入老生代空间**；或者**To空间对象占比超过25%，为不影响内存分配，会将对象放入老生代空间。**

当某个空间没有分块、空间中被对象超过一定限制、空间中不能保证新生代对象移动到老生代中，会启动标记清除的算法。

遍历所有对象，标记活的对象，销毁未标记的。**清除对象后会造成内存出现碎片的情况，超过一定限制启动压缩算法。**将活的对象一端移动，直到所有对象都移动完成，清理掉不需要的内存。

浏览器存储

浏览器的存储方式

cookie，localStorage，sessionStorage，indexDB，

关于他们的区别如下：

特性	cookie	localStorage	sessionStorage	indexDB
数据生命周期	一般由服务器生成，可以设置过期时间	除非被清理，否则一直存在	页面关闭就清理	除非被清理，否则一直存在
数据存储大小	4K	5M	5M	无限
与服务端通信	每次都会携带在 header 中，对于请求性能影响	不参与	不参与	不参与

所以，处于性能考虑，如果没有大量数据存储需求的话，可以使用`localStorage`和`sessionStorage`。对于不怎么改变的数据尽量使用`localStorage`存储，否则可以用`sessionStorage`存储。

Cookie的属性

属性	作用
value	如果用于保存用户登录态，应该将该值加密，不能使用明文的用户标识
http-only	不能通过 JS 访问 Cookie，减少 XSS 攻击
secure	只能在协议为 HTTPS 的请求中携带
same-site	规定浏览器不能在跨域请求中携带 Cookie，减少 CSRF 攻击

如何设置Cookie

cookie的各种参数用字符串拼接，最后保存到一个变量里，用document.cookie设置：

`document.cookie=cookie`

Service Worker

Service Worker 是运行在浏览器背后的**独立线程**，一般可以用来实现缓存功能。使用 Service Worker 的话，传输协议必须为 **HTTPS**。因为 Service Worker 中涉及到请求拦截，所以必须使用 HTTPS 协议来保障安全。

Service Worker 实现缓存功能一般分为三个步骤：

- 先注册 Service Worker
- 监听到 `install` 事件以后就可以缓存需要的文件
- 下次用户访问的时候就可以通过拦截请求的方式查询是否存在缓存，存在缓存的话就可以直接读取缓存文件，否则就去请求数据。

```
//比如在index.js里注册一个Service Worker
if (navigator.serviceWorker){
    navigator.serviceWorker.register('xx.js').then(
    function (registration){
        console.log ('注册成功')
    }).catch(function(e){
        console.log('注册失败')
    })
}

//xx.js
//监听install事件，缓存所需要的文件
self.addEventListener('install',e=>{
    e.wiatUntil(
        caches.open('my-cache').then(function(cache){
            return cache.addAll(['./index.html','./index.js'])
        })
    )
})

//拦截请求
//如果缓存中已经有数据就直接用缓存，否则去请求数据
self.addEventListener('fetch',e=>{
    e.respondWith(
        cache.match(e.request).then(function(response){
            if(response){
                return response;
            }
```

```
            console.log('fetch source');
        })
    )
})
```

缓存机制

缓存可以说是性能优化中**简单高效**的一种优化方式了，它可以**显著减少网络传输所带来的损耗。**

缓存位置

从缓存位置上来说分为四种，并且各自有**优先级**，当依次查找缓存且都没有命中的时候，才会去请求网络。顺序是：

1. Service Worker
2. Memory Cache
3. Disk Cache
4. Push Cache
5. 网络请求

- ### Service Worker

  它的缓存与浏览器其他内建的缓存机制不同，它可以让我们**自由控制**缓存哪些文件、如何匹配缓存、如何读取缓存，并且**缓存是持续性的。**

  当 Service Worker 没有命中缓存的时候，我们需要去调用 `fetch` 函数获取数据。也就是说，如果我们没有在 Service Worker 命中缓存的话，会根据缓存查找优先级去查找数据。**但是不管我们是从 Memory Cache 中还是从网络请求中获取的数据，浏览器都会显示我们是从 Service Worker 中获取的内容。**

- **Memory Cache**

  Memory Cache 也就是内存中的缓存，读取内存中的数据肯定比磁盘快。**内存缓存虽然读取高效，可是缓存持续性很短，会随着进程的释放而释放。** 一旦我们关闭 Tab 页面，内存中的缓存也就被释放了。

- **Disk Cache**

  Disk Cache是存储在硬盘中的缓存，读取速度慢点，但是什么都能存储到磁盘中，比之 Memory Cache **胜在容量和存储时效性上。**

  在所有浏览器缓存中，Disk Cache 覆盖面基本是最大的。它会根据 HTTP Herder 中的字段判断哪些资源需要缓存，哪些资源可以不请求直接使用，哪些资源已经过期需要重新请求。**并且即使在跨站点的情况下，相同地址的资源一旦被硬盘缓存下来，就不会再次去请求数据。**

- ### Push Cache

  Push Cache 是 HTTP/2 中的内容，当以上三种缓存都没有命中时，它才会被使用。**并且缓存时间也很短暂，只在会话（Session）中存在，一旦会话结束就被释放。**

- ### 网络请求

  如果所有缓存都没有命中的话，那么只能发起请求来获取资源了。

缓存策略

- 强缓存
  - **Expires**

    是http1.0内容，**Expires受限于本地时间**，如果修改了本地时间，可能会造成缓存失效

- **Cache-control**

  http1.1的内容，**优先级比Expires高**，可以**在请求头或者响应头中设置**，并且可以组合使用多种指令。

- 协商缓存
  - **Last-Modified 和 If-Modified-Since**

    `Last-Modified` 表示文件最后修改日期，`If-Modified-Since` 会将 `Last-Modified` 的值发送给服务器，询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来，否则返回 304 状态码。

    缺点：1.如果**本地打开缓存文件，即使没有对文件进行修改，但还是会造成** `Last-Modified` **被修改**；2. `Last-Modified` 只能以秒计时，**如果在不可感知的时间内修改完成文件，那么服务端会认为资源还是命中了**，不会返回正确的资源。

  - **ETag 和 If-None-Match**

    `ETag` 类似于文件指纹，`If-None-Match` 会将当前 `ETag` 发送给服务器，询问该资源 `ETag` 是否变动，有变动的话就将新的资源发送回来。并且 `ETag` **优先级比** `Last-Modified` 高。

浏览器渲染

渲染机制

- 接收到HTML文件，转化为DOM树

  当然，在解析 HTML 文件的时候，浏览器还会遇到 CSS 和 JS 文件，这时候浏览器也会去下载并解析这些文件。

- 将CSS文件转换为CSSOM树

  在这一过程中，浏览器会确定下每一个节点的**样式**到底是什么，并且这一过程其实是**很消耗资源**的。

- 生成渲染树

  渲染树只会包括**需要显示的节点**和这些节点的样式信息。

  比如说，如果某个节点是 `display: none` 的，那么就不会在渲染树中显示。

- 会根据渲染树来进行布局（也可以叫做回流），然后调用 GPU 绘制，合成图层，显示在屏幕上。

什么情况下会阻塞渲染

1.首先渲染的前提是生成渲染树，所以 HTML 和 CSS 肯定会阻塞渲染。

想渲染的越快，越应该**降低一开始需要渲染的文件大小**，并且做到**HTML扁平层级，优化CSS选择器。**

2.然后当浏览器在解析到 `script` 标签时，会暂停构建 DOM，完成后才会从暂停的地方重新开始。

所以，如果想首屏渲染的越快，就越**不应该在首屏就加载 JS 文件**，这也是都建议将 `script` 标签放在 `body` 标签底部的原因。

重绘和回流

- 重绘是当节点需要更改外观而不会影响布局的，比如改变 `color` 就叫称为重绘
- 回流是布局或者几何属性需要改变就称为回流。

回流**必定**会发生重绘，重绘**不一定**会引发回流。回流所需的成本比重绘高的多，改变父节点里的子节点很可能会导致父节点的一系列回流。

哪些问题可能会导致重绘和回流

- 改变 window大小
- 改变字体
- 添加或删除样式
- 文字改变
- 定位或者浮动
- 盒模型

性能优化

加载相关

为什么要强调CSS要放在header里，js放在尾部?

> #### DOMContentLoaded 和 load
>
> - DOMContentLoaded 事件触发时，仅当DOM加载完成，不包括样式表，图片...
> - load 事件触发时，页面上所有的DOM，样式表，脚本，图片都已加载完成

构建Render树需要DOM和CSSOM，所以**HTML和CSS都会阻塞渲染。**所以需要让CSS**尽早加载**（如：放在头部），以**缩短首次渲染的时间。**

阻塞浏览器的解析，也就是说发现一个外链脚本时，**需等待脚本下载完成并执行后才会继续解析HTML。**

普通的脚本会阻塞浏览器解析，**加上defer或async属性，脚本就变成异步，可等到解析完毕再执行**

- async异步执行，异步下载完毕后就会执行，不确保执行顺序，一定在onload前，但不确定在DOMContentLoaded事件的前后
- defer延迟执行，相当于放在body最后（理论上在DOMContentLoaded事件前）

Onload&DOMContentLoaded&domready

- DOMContentLoaded 事件触发时，仅当DOM加载完成，不包括样式表，图片
- load 事件触发时，页面上所有的DOM，样式表，脚本，图片都已加载完成
- domready事件在DOM加载后、资源加载之前被触发，在本地浏览器的DOMContentLoaded事件的形式被调用。

白屏、首屏

白屏

白屏时间指的是浏览器开始显示内容的时间，一般认为浏览器开始渲染body或者解析完head标签的时候就是页面白屏结束的时间。

计算方法：IE8-： title后输出一个时间pagestartime;

head结束前 输出一个时间firstpaint。

白屏时间=firstpaint-pagestarttime/performance.timing.navigationStart;

优化

1.加快js的执行速度，比如无限滚动的页面，可以用js先渲染一个屏幕范围内的东西

2.减少文件体积

3.首屏同步渲染html,后续的滚屏再异步加载和渲染。

首屏

首屏时间是指用户打开网站开始，到浏览器首屏内容渲染完成的时间。

计算方法：

1.模块标签标记。适用于内容不需要拉取数据才能生存以及页面不考虑图片等资源的加载情况。结束位置加时间戳输出时间。

2.统计首屏内图片加载最慢事件。 通常图片加载最慢，所以会把首屏内加载事件最慢的图片时间。

3.自定义计算

优化

首屏数据拉取逻辑放在顶部（数据最快返回）

首屏渲染css及js逻辑优先内联html，返回时能立即执行

次屏逻辑延后执行

DOM构建时间

浏览器开始对基础页文件内容进行解析，构建出一个DOM树的时间。domready事件在DOM加载后、资源加载之前被触发，在本地浏览器的DOMContentLoaded事件的形式被调用。

整页时间

整个页面加载完成时间

loadEvntEnd-navigationStart/onload记录时间戳。

渲染相关

如何减少重绘和回流?

- 使用 `transform` 替代 `top`
- 使用 `visibility` 替换 `display: none` ，因为前者只会引起重绘，后者会引发回流（改变了布局）
- 不使用 `table` 布局，可能很小的一个小改动会造成整个 `table` 的重新布局
- CSS选择符从右往左匹配查找，避免节点层级过多
- 将频繁重绘或者回流的节点设置为图层，图层能够阻止该节点的渲染行为影响别的节点。（will-change属性或者video,iframe标签等）
- 节点属性不要放在一个循环里当循环变量

```
//每次都要去取正确的值才行
for(let i=0;i<100;i++){
    console.log(document.querySeletor('.test').style.offsetTop);
}
```

为什么操作DOM的性能很差?

因为 DOM 是属于渲染引擎中的东西，而 JS 又是 JS 引擎中的东西。当我们通过 JS 操作 DOM 的时候，其实这个操作涉及到了两个线程之间的通信，那么势必会带来一些性能上的损耗。

操作 DOM 次数一多，也就等同于一直在进行线程之间的通信，并且操作 DOM 可能还会带来重绘回流的情况，所以也就导致了性能上的问题。

插入几万个 DOM，如何实现页面不卡顿?

解决问题的重点应该是如何分批次部分渲染 DOM。

- 通过 `requestAnimationFrame` 的方式去循环的插入 DOM；
- 通过虚拟滚动

  这种技术的原理就是只渲染可视区域内的内容，非可见区域的那就完全不渲染了，当用户在滚动的时候就实时去替换渲染的内容。

  即使列表很长，但是渲染的 DOM 元素永远只有那么几个，当我们滚动页面的时候就会实时去更新 DOM。

不考虑缓存和优化网络协议的前提下，可以通过哪些方式来最快的渲染页面?

这个问题的，其实在了解渲染的过程之后，解决方案已经不言而喻了，无非就是减少生成渲染树的时间了。那么回顾下渲染树是怎么生成的呢：DOM+CSSOM。

所以答案如下：

1. 从文件大小考虑
2. 从 `script` 标签使用上来考虑 async和differ
3. 从需要下载的内容是否需要在首屏使用上来考虑
4. 最后就是从 CSS、HTML 的代码书写上来考虑了

图片优化

图片大小如何优化

- **减少像素点**
- **减少每个像素点能够显示的颜色**

图片加载如何优化

1. 用 CSS 去代替。
2. 用 CDN 加载，计算出适配屏幕的宽度，然后去请求相应裁剪好的图片。
3. 小图使用 base64 格式
4. 将多个图标文件整合到一张图片中（雪碧图）
5. 选择正确的图片格式：
   - 尽量使用 WebP 格式。因为 WebP 格式具有更好的图像数据压缩算法，能带来更小的图片体积，而且拥有肉眼识别无差异的图像质量。
   - 小图使用 PNG，其实对于大部分图标这类图片，完全可以使用 SVG 代替
   - 照片使用 JPEG

防抖节流

防抖

如果在频繁的事件回调中做复杂计算，很有可能导致页面卡顿，不如将多次计算合并为一次计算，只在一个精确点做操作。频繁触发，有足够空闲时间才执行

```
const debounce = (func, wait = 50) => {
  let timer = 0
  return function(...args) {
    if (timer) clearTimeout(timer);
    timer = setTimeout(() => {
      func.apply(this, args)
    }, wait)
  }
}
```

节流

防抖动是将多次执行变为最后一次执行，节流是将多次执行变成每隔一段时间执行。

```
function throttle(func,wait){
    let last;
    return function(...args){
        let now=+new Date();
        if(!last||now>last+wait){
            last=now;
            func.apply(this,args);
        }
    }
}
```

预渲染

通过预渲染**将下载的文件预先在后台渲染**，可以使用以下代码开启预渲染

```
<link rel="prerender" href="http://example.com">
```

预渲染虽然可以提高页面的加载速度，但是要**确保该页面大概率会被用户在之后打开**，否则就是白白浪费资源去渲染。

懒执行

懒执行就是**将某些逻辑延迟到使用时再计算。**

该技术可以用于首屏优化，对于某些耗时逻辑并不需要在首屏就使用的，就可以使用懒执行。

懒执行需要唤醒，一般可以通过定时器或者事件的调用来唤醒。

懒加载

懒加载就是**将不关键的资源延后加载。**

懒加载的原理就是只加载自定义区域（通常是可视区域，但也可以是即将进入可视区域）内需要加载的东西。

对于图片来说，先设置图片标签的 `src` 属性为一张占位图，将真实的图片资源放入一个自定义属性中，当进入自定义区域时，就将自定义属性替换为 `src` 属性，这样图片就会去下载资源，实现了图片懒加载。

懒加载不仅可以用于图片，也可以使用在别的资源上。比如进入可视区域才开始播放视频等等。

CDN

CDN 的原理是尽可能的在各个地方分布机房缓存数据，这样即使我们的根服务器远在国外，在国内的用户也可以通过国内的机房迅速加载资源。

因此，我们可以将**静态资源尽量使用 CDN 加载**，由于浏览器对于单个域名有并发请求上限，可以考虑使用多个 CDN 域名。并且对于 CDN 加载静态资源需要注意**CDN 域名要与主站不同**，否则每次请求都会带上主站的 Cookie，平白消耗流量。

Web安全

XSS

XSS概念及攻击手段

XSS 简单点来说， 中文名为跨站脚本, 是发生在**目标用户的浏览器**层面上的，简单理解就是：攻击者想尽一切办法将可以执行的代码注入到网页中。重点在**脚本**上。

可以分为两种类型：**持久型和非持久型**。

- 持久型就是攻击的代码被服务端写入进**数据库**中。

  比如在一些论坛的评论中，写script标签加alert语句，如果前后端没有做好防御的话，这段评论就会被存储到数据库中，这样每个打开该页面的用户都会被攻击到。

- 非持久型一般通过**修改 URL 参数**的方式加入攻击代码，诱导用户访问链接从而进行攻击。

如何防范XSS攻击

转义字符

转义输入输出的内容，对于引号、尖括号、斜杠进行转义。

CSP白名单

明确告诉浏览器哪些外部资源可以加载和执行。

通过两种方式来开启 CSP：

1. 设置 HTTP Header 中的 `Content-Security-Policy`
2. 设置 `meta` 标签的方式 `<meta http-equiv="Content-Security-Policy">`

CSRF

概念及攻击手段

CSRF 中文名为**跨站请求伪造。**

CSRF攻击的本质在于**利用用户的身份，执行非本意的操作。**重点在于：**CSRF的请求是跨域且伪造的。**

跨站请求伪造的攻击是攻击者通过一些技术手段欺骗用户的浏览器去访问用户曾经认证过的网站并执行一些操作（如发送邮件、发消息、甚至财产操作如转账和购买商品等）。由于浏览器曾经认证过，所以被访问的网站会认为是真正的用户操作而去执行。这利用了web登录身份认证的一个漏洞：**简单的身份认证只能保证请求来自用户的浏览器，但不能识别请求是用户自愿发出的。**

如何防范

可以遵循以下几种规则：

1. Get 请求不对数据进行修改
2. 不让第三方网站访问到用户 Cookie
3. 阻止第三方网站请求接口
4. 请求时附带验证信息，比如验证码或者 Token

对Cookie设置SameSite属性

该属性表示 Cookie 不随着跨域请求发送，可以很大程度减少 CSRF 的攻击，但是该属性目前并不是所有浏览器都兼容。

验证Referer

可以通过验证 Referer 来判断该请求是否为第三方网站发起的。

利用Token

服务器下发一个随机 Token，每次发起请求时将 Token 携带上，服务器验证 Token 是否有效。

点击劫持

概念

点击劫持是一种**视觉欺骗**的攻击手段。攻击者将需要攻击的网站通过 `iframe` 嵌套的方式嵌入自己的网页中，并将 `iframe` 设置为透明，在页面中透出一个按钮诱导用户点击。

防范措施

设置HTTP头：X-FRAME-OPTIONS

这个 HTTP 响应头 就是为了防御用 `iframe` 嵌套的点击劫持攻击。

该响应头有三个值可选，分别是

- `DENY`，表示页面不允许通过 `iframe` 的方式展示
- `SAMEORIGIN`，表示页面可以在相同域名下通过 `iframe` 的方式展示
- `ALLOW-FROM`，表示页面可以在指定来源的 `iframe` 中展示

JS 防御

对于某些远古浏览器来说，并不能支持上面的这种方式，那我们只有通过 JS 的方式来防御点击劫持了。

当通过 `iframe` 的方式加载页面时，攻击者的网页直接不显示所有内容了。

```
<head>
```

```
  <style id="click-jack">
    html {
      display: none !important;
    }
  </style>
</head>
<body>
  <script>
    if (self == top) {
      var style = document.getElementById('click-jack')
      document.body.removeChild(style)
    } else {
      top.location = self.location
    }
  </script>
</body>
```

中间人攻击

概念

中间人攻击是**攻击方同时与服务端和客户端建立起了连接，并让对方认为连接是安全的，但是实际上整个通信过程都被攻击者控制了。**攻击者不仅能获得双方的通信信息，还能修改通信信息。

如何防范

防御中间人攻击其实并不难，只需要增加一个安全通道来传输信息。**HTTPS 就可以用来防御中间人攻击**，但是并不是说使用了 HTTPS 就可以高枕无忧了，因为如果你没有完全关闭 HTTP 访问的话，攻击方可以通过某些方式将 HTTPS 降级为 HTTP 从而实现中间人攻击。

情景问题解决方案

关于缓存

对于频繁变动的资源，缓存应该怎么设置?

对于频繁变动的资源，首先需要使用 `Cache-Control: no-cache` 使浏览器每次都请求服务器，然后配合 `ETag` 或者 `Last-Modified` 来验证资源是否有效。

这样的做法虽然不能节省请求数量，但是能显著减少响应数据大小。

代码文件的变动，应该怎么缓存?

一般来说，现在都会使用工具来打包代码，那么我们就可以对文件名进行哈希处理，只有当代码修改后才会生成新的文件名。基于此，我们就可以给代码文件设置缓存有效期一年 `Cache-Control: max-age=31536000`，这样只有当 HTML 文件中引入的文件名发生了改变才会去下载最新的代码文件，否则就一直使用缓存。

代码发布错了，如何强制更新?

和上面的问题一样，修改代码文件的hash文件名，浏览器可以自动的去下载

功能实现

如果要实现瀑布流，有什么需要注意的地方

节流、dom元素优化、两边高度统一

判断瀑布流滑动到的位置是否需要加载，循环遍历图片，获取他们的高度，在对他们进行排序，然后懒加载获取图片，进行批量加载。

兼容性

如果要兼容IE8有哪些需要注意的地方?

- 不能使用CSS3伪类 伪类选择器
- 布局要用relative absolute，不可以用flex和grid
- js不能用addEventListener,用DOM0级事件绑定
- 动画使用jq动画实现

手撕代码

数据结构

排序算法

比较

对于评述算法优劣术语的说明

稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；
不稳定：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面；

内排序：所有排序操作都在内存中完成；
外排序：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；

时间复杂度: 一个算法执行所耗费的时间。
空间复杂度: 运行完一个程序所需内存的大小。

算法比较：

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

二分查找

假设一个数组已经排好序了，现在要在数组里找一个数flag的位置。

首先先找到长度中间位置，通过与中间位置的数比较，比中间值大在右边找，比中间值小在左边找。然后再在两边各自寻找中间值，持续进行，直到找到全部位置。

```
//arr排好序
function binarySearch(arr,flag,Start,End){
    let end =End||arr.length-1;
    let start=Start||0;
    let m=Math.floor((start+end)/2);
    if(arr[m]==flag){
        return m;
    }
    if(flag<arr[m]){
        return binarySearch(arr,flag,0,m-1);
    }else{
        return binarySearch(arr,flag,m+1,end);
    }
    return false;
}
```

非递归的方法也写一个吧

```
function binarySearch(arr,flag){
    let r=arr.length-1,//数组下标长度减一
        l=0;
    while(l<=r){
        let m=Math.floor((h+1)/2);
        if(data[m]==flag){
            return m;
        }
        if(flag>data[m]){
            l=m+1;
        }else{
```

```
            r=m-1;
        }
    }
    return false;
}
```

冒泡排序

比较相邻两个元素的，如果前一个比后面的大，就交换位置，第一轮之后最后一个元素是最大的一个，按照这种方法依次比较。

```
function bubbleSort(arr){
    for(let i=arr.length-1;i>=0;i--){
        for(j=0;j<i;j++){
            if(arr[j]>arr[j+1]){
                let tmp=arr[j+1];
                arr[j+1]=arr[j];
                arr[j]=tmp;
            }
        }
    }
    return arr;
}
```

快速排序

是利用二分查找对冒泡排序的改进，选一个元素作为基准，把数字分为两部分，一部分全部比它小，一部分全部比它大，然后递归调用，在两部分都进行快排。

```
function quickSort(arr){
    if(arr.length<=1){
        return arr;
    }
    //中间位置
    let middle=Math.floor(arr.length/2);
    let flag=arr.splice(middle,1)[0];//取出中间元素
    let left=[],right=[];
    for(let i=0;i<arr.length;i++){
        if(arr[i]<flag){
            left.push(arr[i]);
        }else{
            right.push(arr[i]);
        }
    }
    return quickSort(left).cocat([flag],quickSort(right));
}
```

插入排序

从第一个元素开始，该元素认为已经被排序了，取出下一个元素。在已经排序的元素序列中从后向前扫描，如果大于新元素，那么就把这个元素移动到下一个位置。直到找到已排序的元素小于或者等于新元素的位置，将新元素插入下一个位置。依次进行。（其实就是最开头的元素当作是有序数列，后面的元素是无序的，然后从第一个开始往前面插入）

```

```

```
function insertSort(arr){
    //从第一个开始（遍历无序数列）
    for(let i=1;i<arr.length;i++){
        if(arr[i]<arr[i-1]){
            let guard=arr[i];//无序数列中的第i个元素
            let j=i-1;//有序数列的最后一个位置
            arr[i]=arr[j];
            while(j>=0&&guard<arr[j]){
                arr[j+1]=arr[j];
                j--;
            }
            arr[j+1]=guard;
        }
    }
}
```

选择排序

首先在未排序的队里找到最小的元素，存放到排序序列中的起始位置，然后从剩下没有排序的元素中找最小的元素，放到已排序的队尾

```
function selectionSort(arr){
    let len=arr.lengrh;
    let index,temp;
    for(let i=0;i<len-1;i++){
        index=i;
        for(let j=i+1;j<len;j++){
            if(arr[j]<arr[index]){
                index=j;//存最小索引
            }
        }
        tmp=arr[j];
        arr[i]=arr[index];
        arr[index]=temp;
    }
    return arr;
}
```

希尔排序

用于较大规模无序数据。

先将整个待排序的数据集分割为若干组，然后对每一个组分别进行直接插入排序。此时每个组内插入排序所作用的数据量小，效率比较高。

排序完后基本上数组，小元素大体在前，大元素大体在后，然后缩小增量，继续分组，此时虽然每个分组元素个数变多了，但是数组变有序了，效率也是比较高的。

用的比较少，篇幅稍微长一些。

这个算法的图解可以看这篇博客，用一问一答的方式把它描述得非常有趣了。专业一点的描述可以看这篇博客，动图把过程说的非常清楚。

也可以直接看下面的动图（摘自上面的博客）

```
```

```
function shellSort(arr){
    var len=arr.length;
    var tmp,gap=1;

    while(gap<len/5){
        gap=gap*5+1;
    }
    for(gap;gap>0;gap=Math.floor(gap/5)){
        for(var i=gap;i<len;i++){
            tmp=arr[i];
            for(var j=i-gap;j>=0&&arr[j]>tmp;j-=gap){
                arr[j+gap]=arr[j];
            }
            arr[j+gap]=tmp;
        }
    }
    return arr;
}
```

归并排序

一种稳定排序方法，将已有序的子序列合并，得到完全有序的序列，即先使每个子序列有序，再使序列段间有序。

其实也是二分的思想，只不过是在二分的基础上，先分段，段内再排，然后把每一段拼接起来。

这篇博客里有非常仔细的图片分析。这个需要新申请一个数组来做，所以自然是O(n)的空间复杂度啦！

从上面这篇博客偷了个动图帮助理解：

```
function mergeSort(arr){
    //自上而下递归
    var len =arr.length;
    if(len<2){return arr}
    var middle=Math.floor(len/2),
        left=arr.slice(0,middle),
        right=arr.slice(middle);
    return merge(mergeSort(left),mergeSort(right));
}

function merge(left,right){
    var result=[];
    while(left.length&&right.length){
        if(left[0]<=right[0]){
            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;
```

```
}
```

堆排序

堆排序利用堆的数据结果设计的排序算法。堆是一个近似完全二叉树的结构，同时满足：子节点的键值或索引总是小于/大于父节点。

```
function heapSort(arr){
    if(Object.prototype.toString.call(arr).slice(8,-1)==='Array'){
        var heapSize=arr.length,temp;//建堆
        for(var i=Math.floor(heapSize/2)-1;i>=0;i--){
            heapify(arr,i,heapSize);
        }
        for(var j=heapSize-1;j>=1;j--){
            //堆排序
            temp=arr[0];
            arr[0]=arr[j];
            arr[j]=temp;
            heapify(arr,0,--heapSize);
        }
        return arr;
    }else{
        throw TypeError('Input not Array');
    }
}

function heapify(arr,x,len){
    if(Object.prototype.toString.call(arr).slice(8,-1)==='Array'&&typeof
x==='number'){
        var l=x*2+1,
            r=x*2+2,
            largest=x,
            temp;
        if(l<len&&arr[l]>arr[largest]){
            largest=l;
        }
        if(r<len&&arr[r]>arr[largest]){
            largest=r;
        }
        if(largest!=x){
            temp=arr[x];
            arr[x]=arr[largest];
            arr[largest]=temp;
            heapify(arr,largest,len);
        }
    }else{
        throw TypeError('Input Illegal');
    }
}
```

计数排序

使用一个额外的数组C，其中第i个元素是待排序数组A中值等于i的元素的个数，然后根据数组C来将A中的元素排到正确位置。（只能对整数排序）

```
function countingSort(arr){
    var len=arr.length,B=[],C=[],min=max=arr[0];
    for(var i=0;i<len;i++){
        min=min<arr[i]? min:arr[i];
        max=max>arr[i]? max:arr[i];
        C[arr[i]]=C[arr[i]]?C[arr[i]]+1:1;
    }
    for(var j=min;j<max;j++){
        C[j+1]=(C[j+1]||0)+(C[j]||0);
    }
    for(var k=len-1;k>=0;k--){
        B[C[arr[k]]-1]=arr[k];
        C[arr[k]]--;
    }
    return B;
}
```

桶排序

假设输入的数据服从均匀分布，将数据分到有限数量的桶里。每个桶再分别排序。（可以再使用别的排序算法，或者递归继续用桶排序）

```
function bucketSort(arr,num){
    if(arr.length<=1) return arr;
    let len=arr.length;
    let buckets=[],result=[],space,n=0;
    min=max=arr[0];
    let regex='/^[1-9]+[0-9]*$/';

    for(let i=1;i<len;i++){
        min=min<=arr[i]?min:arr[i];
        max=max>=arr[i]?max:arr[i];
    }
    space=(max-min+1)/num;//索引减法+1
    for(let j=0;j<len;j++){
        let index=Math.floor((arr[j]-min)/space);
        if(buckets[index]){
            //不是空桶的话，就插入排序
            var k=buckets[index].length-1;
            while(k>=0&&buckets[index][k]>arr[j]){
                buckets[index][k+1]=buckets[index][k];
                k--;
            }
            buckets[index][k+1]=arr[j];
        }else{
            //空桶，初始化
            buckets[index]=[]
            buckets[index].push(arr[j]);
        }
    }
    while(n<num){
        result=result.concat(buckets[n]);
        n++;
    }
    return result;
}
```

基数排序

基数排序是按照地位先排序，然后收集；再按照高位排序，然后收集；以此类推，直到最高位。有时候有些属性有优先级顺序，先按低优先级排序，在按高优先级排序。

最后次序就是高优先级在前，高优先级相同的低优先级高的在前。

大意就是说：比如[53 3 542 748 14]

先按个位排序就是[542 053 003 014 748]

然后按照十位排序，[003 014 542 748 053]

然后按照百位排序，[003 014 053 542 748]

因为它是分别排序分别收集的，所以是稳定的。

```
function radixSort(arr,maxDigit){
    let mod=10,dev=1,counter=[];
    for(let i=0;i<maxDigit;i++,dev*=10,mod *=10){
        for(let j=0;j<arr.length;j++){
            let bucket=parseInt((arr[j]%mod)/dev);
            if(counter[bucket]==null){
                counter[bucket]=[];
            }
            counter[bucket].push(arr[j]);
        }
        var pos=0;
        for(let j=0;j<counter.length;j++){
            let value=null;
            if(counter[j]!=null){
                while((value=counter[j].shift())!=null){
                    arr[pos++]=value;
                }
            }
        }
    }
    return arr;
}
```

注：这三种算法用到的比较少，但上面这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：

- 基数排序：根据键值的每位数字来分配桶
- 计数排序：每个桶只存储单一键值
- 桶排序：每个桶存储一定范围的数值

JS功能实现

数组去重的方法

双重for循环

外层遍历，内层比较。

```
function distinct(arr){
    for(let i=0,len=arr.length;i<len;i++){
        for(let j=i+1;j<len;j++){
            if(arr[i]==arr[j]){
                arr.splice(j,1);
                //数组长度变了
                len --;
                j--;
            }
        }
    }
}
```

Array.filter()+indexOf()

合并为一个数组，用fliter遍历数组,结合indexOf排除重复项。

```
function distinct(arr){
    return arr.filter((item,index)=>arr.indexOf(item)===index)
}
```

for...of+includes()

双重for的升级版，外层用for...of替换，内层循环改用includes。

创建一个新数组，乳沟includes返回false,将该元素push进去。

```
function distinct(arr){
    let result=[];
    for(let i of arr){
        !result.includes(i)&&result.push(i);
    }
    return result;
}
```

Array.sort()+相邻元素比较

先排序，然后比较相邻元素，从而排除重复项。

```
function distinct(arr){
    arr.sort();
    let result=[arr[0]];

    for(let i=1,len=arr.length;i<len;i++){
        arr[i] !==arr[i-1]&&result.push(arr[i]);
    }
    return result;
}
```

new Set()

```
function distinct(arr){
    return Array.from(new Set([...arr]));
}
```

for...of+Object对象属性不重复

```
function distinct(arr){
    let result=[];
    let obj={};

    for(let i of arr){
        if(!obj[i]){
           result.push(i);
           obj[i]=1;
        }
    }
    return result;
}
```

异步操作对比--实现获取用户信息

先写一个获取用户的方法

```
function fetchUser() {
    return new Promise((resolve, reject) => {
        fetch('https://api.github.com/users/yiichitty')
        .then((data) => {
            resolve(data.json());
        }, (error) => {
            reject(error);
        });
});
```

使用Promise

```
function getUserByPromise() {
    fetchUser()
        .then((data) => {
            console.log(data);
        }, (error) => {
            console.log(error);
        })
}
```

Promise 的方式虽然解决了回调地狱的问题，但是如果处理流程复杂的话，整段代码将充满 then()。

它的**语义化不明显，代码流程不能很好的表示执行流程。**

使用Generator

```
function* fetchUserByGenerator() {
    const user = yield fetchUser();
    return user;
}

const g = fetchUserByGenerator();
const result = g.next().value;
result.then((v) => {
    console.log(v);
}, (error) => {
    console.log(error);
})
```

Generator 的方式解决了 Promise 的一些问题，流程更加直观、语义化。但是 Generator 的问题在于，**函数的执行需要依靠执行器，每次都需要通过next()的方式去执行。**

使用async+await

```
async function getUserByAsync(){
     let user = await fetchUser();
     return user;
 }
getUserByAsync()
.then(v => console.log(v));
```

async函数完美的解决了上面两种方式的问题。流程清晰，直观、语义明显。

操作异步流程就如同操作同步流程。同时async函数自带执行器，执行的时候无需手动加载。

数组去重通用API

要求就是写一个数组去重的API，第一个参数是一个数组，第二个参数是一个函数（对象数组去重的规则）。

思路就是当它是数组的时候，直接用这个Set数据结构去重，如果是个对象数据的话，就新建一个Map，按照传入的函数返回的值，存入Map。这里用到了filter，它是用传入的函数测试所有的元素，并且返回所有通过测试的元素。

```
function fn(arr,rule){
    if(!rule){
        return Array.from(new Set([...arr]));
    }else{
        const res=new Map();
        return arr.filter((a)=>!res.has(rule(a))&&res.set(rule(a),1));
    }
}
```

对象数组按规则排序

有一个数组，里面都是对象，现在要针对对象中的某一个key进行排序，顺序是已给定的数组。
比如原数组为[{a:'ww'},{a:'ff'},{a:'pe'}],
顺序是[{ww:1},{pe:3},{hf:2},{oo:4},{ff:5}]
那么输出是 [{a:'ww'},{a:'pe'},{a:'ff'}]

```
var objarr=[{a:'ww'},{a:'ff'},{a:'pe'}];
```

```
var rulearr=[{ww:1},{pe:3},{hf:2},{oo:4},{ff:5}];

//暴力解决
function sortByRule(objarr,key,rulearr){
    let ResultArr=[];
    rulearr.forEach(item=>{
        //获取当前的value和规则的位次
        let value=Object.getOwnPropertyNames(item)[0];
        let order=item[value];
        //找到对应的obj放入对应位次的位置
        ResultArr[order]=objarr.find(item=>item[key]===value);
    });
    //去掉那些为空的
    return ResultArr.filter(item=>item);
}

//用sort方法
function sortByRule(objarr,key,rulearr){
    //把rulearr处理成一个对象{ww:1,pe:3……}
    let rule={}
    rulearr.forEach(item=>rule={...rule,...item});
    //对objarr排序
    return objarr.sort((a,b)=>{
        //取到“a”
        const akey=Object.keys(a)[0];
        const bkey=Object.keys(b)[0];
        //按升序排
        return rule[a[akey]]-rule[b[bkey]];
        }
        return 0;
    })
}
```

实现防抖

```
//实现按钮防2次点击操作
//在规定时间内再次触发就清除定时后重新设置，直到不触发了
function debounce(fn,delay){
    let timer=0;
    return function(...args){
            if (timer) clearTimeout(timer)
            timer=setTimeout(()=>{func.apply(this,args)},delay);
    }
}

function fn(){
    console.log('防抖')
}
addEventListener('scroll',debounce(fn,1000))
```

实现节流

```
//节流就是说在一定时间内只会被触发一次
//比如滚动事件啥的
function throttle(fn,delay){
    let last;//上次被触发的时间
```

```
    return function(...args){
        let now=+new Date();
        if(!last||now>last+delay){
            last=now;
            func.apply(this,args);
        }
    }
}


//使用
function fn(){
    console.log('节流');
}
addEventListener('scroll',throttle(fn,1000));
```

实现懒加载

```
//首先html的img标签设置一个无关的标签比如说data，加载到的时候再替换成src
//思路就是到视口区了再替换过去加载

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

hash路由

```
//hash路由
class Route{
    constructor(){
        //存储对象
        this.routes=[];
        //当前hash
        this.currentHash=''
        //绑定this.避免监听时this指向改变
        this.freshRoute=this.freshRoute.bind(this);
        //监听
        window.addEventListener('load',this.freshRoute,false);
        window.addEventListener('hashmessage',this.freshRoute,false);
```

```
    }
    //存储
    storeRoute(path,cb){
        this.routes[path]=cb||function(){};
    }
    //更新
    freshRoute(){
        this.currentHash=location.hash.slice(1)||'/'
        this.routes[this.currentHash]();
    }
}
```

实现元素拖拽

```
window.onload=function(){
    //drag目标是绝对定位状态
    var drag=document.getElementById('box');
    drag.onmousedown=function(e){
        e=e||window.event;
        //鼠标与拖拽元素的距离=鼠标与可视区边界的距离-拖拽元素与可视区的距离
        let diffX=e.clientX-drag.offsetLeft;
        let diffY=e.clientY-drag.offsetTop;
        drag.onmousemove=function(e){
            e=e||window.event;
            //拖拽元素的移动距离=鼠标当前与可视区边界的距离-鼠标与拖拽元素的距离
            let left=e.clientX-diffX;
            let top=e.clientY-diffY;
            //避免拖出可视区外
            if(left<0){ left=0;}
            else if(left>window.innerWidth-drag.offsetWidth){
                //超出了就放在innerWidth的位置
                left=window.innerWidth-drag.offsetWidth;
            }
            if(top<0) top=0;
            else if (top>window.innerHeight-drag.offsetHeight){
                top=window,innerHeight-drag.offsetHeight;
            }
            drag.style.left=left+'px';
            drag.style.top=top+'px';
        }
        drag.onmouseup=function(e){
            e=e||window.event;
            this.onmousemove=null;
            this.onmousedown=null;
        }
    }
}
```

构建一个Service Worker

```
//比如在index.js里注册一个Service Worker
if (navigator.serviceWorker){
    navigator.serviceWorker.register('xx.js').then(
    function (registration){
        console.log ('注册成功')
    }).catch(function(e){
```

```
        console.log('注册失败')
    })
}

//xx.js
//监听install事件，缓存所需要的文件
self.addEventListener('install',e=>{
    e.wiatUntil(
        caches.open('my-cache').then(function(cache){
            return cache.addAll(['./index.html','./index.js'])
        })
    )
})

//拦截请求
//如果缓存中已经有数据就直接用缓存，否则去请求数据
self.addEventListener('fetch',e=>{
    e.respondWith(
        cache.match(e.request).then(function(response){
            if(response){
                return response;
            }
            console.log('fetch source');
        })
    )
})
```

JS原理

使用setTimeout模拟setInterval

```
//使用setTimeout模拟setInterval
//避免因执行时间导致间隔执行时间不一致
setTimeout(function(){
    //do something
    //arguments.callee引用该函数体内当前正在执行的函数
    setTimeout(arguments.callee,500);
},500)
```

实现call

```
////传入一个this 绑定上所有的属性
Function.prototype.myCall=function(context){
    if(typeof this !=='function'){
        throw new TypeError('error');
    }
    context=context||window;
    context.fn=this;
    //除去要绑定的对象，剩下参数应该绑定进去
    const args=[...arguments].slice(1);
    const result=context.fn(...args);
    delete context.fn;
    return result;
}
```

实现Apply

```
//与call的区别是，第二个第二个参数传入的是数组
Function.prototype.apply()=function(context){
    if(typeof this !=='function'){
        throw new TypeError('error');
    }
    context=context||window;
    context.fn=this;
    let result;
     //判断是否存在数组参数,毕竟是可选参数
    if(arguments[1]){
        result=context.fn(...arguments[1]);
    }else{
        result=context.fn();
    }
    delete context.fn;
    return result;
}
```

实现Bind

```
Function.prototype.myBind=function(context){
    if (typeof this !== 'function') {
        throw new TypeError('error');
    }
    const _this=this;
    const args=[...arguments].slice(1);
    return function F(){
        // new ,不动this
        if (this instanceof F){
            //链式调用要加上新旧参数
            return new _this(...args,...arguments);
        }
        return _this.apply(context,args.concat(...arguments));
    }
}
```

实现Object.create()

```
//Object.create()方法创建一个新对象，使用现有的对象来提供新创建的对象的__proto__。
Object.prototype.mycreate=function(obj){
    function F(){}
    F.prototype=obj;
    return new F();
}

//其实也可以这么写
Object.prototype.mycreate=function(obj){
    return {'_protp_':obj};
}
```

实现new

```
//使用new时:1 内部生成一个obj 2 链接到原型 3 obj绑定this(使用构造函数的this) 4 返回新对象（原始值的话忽略，如果是对象的话就返回这个对象）
Function.prototype.myNew()=function(func,...args){
    let obj={};
    obj._proto_=func.prototype;
    let result=func.apply(obj,args);
    return result instanceof Object? result:obj;
}

//可以用Object.create
//以构造函数的原型对象为原型，创建一个空对象；等价于 创建一个新的对象，把他的_proto_指向prototype
Function.prototype.myNew()=function(func,...args){
    let obj=Object.create(func.prototype);
    let result=func.apply(obj,args);
    return result instanceof Object? result:obj;
}
```

Instanceof

```
//本质是看左边变量的原型链是否含有右边的原型
 function myInstanceof(left,right){
     //要找的原型
    let prototype=right.prototype;
    left=left._proto_;
    while(true){
        //全部遍历完都没有 false
        if(left===null||left==="undefined")  return false;
        //匹配上 true
        if(left===prototype) return true;
        //找下一个原型链
        left=left._proto_;
    }
}
```

深拷贝

```
//JSON.parse 解决不了循环引用的问题，会忽略undefined\symbol\函数
let a={}
let b=JSON.parse(JSON.stringify(a));
//循环引用是说a.b.d=a.b这样的

//MessageChannel含有内置类型，不包含函数
function structuralClone(obj){
    return new Promise(resolve=>{
        const{port1,port2}=new MessageChannel();
        port2.onmessage=ev=>{resolve(ev.data)}
        port1.postMessage(obj);
    })
}

//可以处理循环引用对象、也可以处理undefined
//不过它是异步的
const test =async()=>{
    const clone=await structuralClone(obj);
    console.log(clone);
```

```
}
test();


//手写一个简单的deepclone
function deepClone(obj){
    function isObject(o){
        return (typeof o==='object'||typeof o==='function')&&o !==null;
    }
    if(!isObject(obj)){
        throw new Error('Not Object')
    }

    let isArray=Array.isArray(obj);
    let newObj=isArray?[...obj]:{...obj};
    Reflect.ownKeys(newObj).forEach(item=>{
        newObj[item]=isObject(obj[item])?deepClone(obj[item]):obj[item];
    })
    return newObj;
}

//还可以这样写
function deepClone(){
    let copy=obj instanceof Array? []:{};
    for(let i in obj){
        if(obj.hasOwnProperty(i)){
            copy[i]=typeof obj[i] ==='object'?deepClone(obj[i]):obj[i];
        }
    }
    return copy;
}
```

实现Array.map()

```
Array.prototype.newMap = function(fn) {
    var newArr = [];
    for(var i = 0; i<this.length; i++){
        newArr.push(fn(this[i],i,this))
    }
    return newArr;
}
```

实现Array.reduce()

```
Array.prototype.MyReduce = function(fn , prev) {
    for(let i = 0; i<this.length; i++) {
        if (typeof prev === 'undefined') {
            // prev不存在
            prev = fn(this[i], this[i+1], i+1, this);
            i++;
        } else {
            prev = fn(prev, this[i], i, this);
        }
    }
    return prev;
```

原生实现Ajax

```
//简单流程

//实例化
let xhr=new XMLHttpRequest();
//初始化
xhr.open(methond,url,async);
//发送请求
xhr.onreadystatechange=()=>{
    if(xhr.readyState===4&&xhr.status===200){
        console.log(xhr.responseText);
    }
}

//有Promise的实现

function ajax(options){
    //地址
    const url=options.url;
    //请求方法
    const method=options.methond.toLocalLowerCase()||'get';
    //默认为异步true
    const async=options.async;
    //请求参数
    const data=options.data;
    //实例化
    let xhr=new XMLHttpRequest();
    //超时时间
    if(options.timeout&&options.timeout>0){
        xhr.timeout=options.timeout;
    }


    return new Promise((resolve,reject)=>{
        xhr.ontimeout=()=>reject&&reject('请求超时');

        //状态变化回调
        xhr.onreadystatechange=()=>{
            if(xhr.readyState==4){
                if(xhr.status>=200&&xhr.status<300||xhr.status==304){
                    resolve && resolve(xhr.responseText)
                }else{
```

```
                    reject&&reject();
                }
            }
        }

        //错误回调
        xhr.onerr=err=>reject&&reject();

        let paramArr=[];
        let encodeData;
        //处理请求参数
        if(data instanceof Object){
            for(let key in data){

 paramArr.push(encodeURIComponent(key)+"="+encodeURIComponent(data[key]));
            }
            encodeData=paramArr.join('&');
        }

        //get请求拼接参数
        if(method==='get'){
            //检查url中有没有?以及它的位置
            const index=url.indexOf('?');
            if(index===-1) url+='?';
            else if (index !==url.length -1)  url+='&';
            url += encodeData;
        }

        //初始化
        xhr.open(method,url,async);

        if(method ==='get') xhr.send(encodeData);
        else{  //post设置请求头
            xhr.setRequestHeader('Content-Type','application/x-www-form-
unlencoded;charset=UTF-8');
            xjr.send(encodeData);
        }
    })

}
```

手写实现一个Promise

```
//n个promise的数组，按要执行的顺序排好
var promiseArr=[new Promise(()=>{console.log('1')}),
                            new Promise(()=>{console.log('2')}),
                            new Promise(()=>{console.log('3')}),
                            new Promise(()=>{console.log('4')})
                        ];


function* donebyOrder(arr){
    let len=arr.length;
    for(let i=0;i<len;i++){
         yield arr[i];
    }
}
```

```
let gen=donebyOrder(promiseArr)
for(let i=1;i<promiseArr.length;i++){
   gen.next();
}
```

实现Promise.all

```
//Promise.all
//多个Promise任务并行执行,输入是一个数组，最后输出一个新的Promise对象
//1.如果全部成功执行，则以数组的方式返回所有Promise任务的执行结果;
//2.如果有一个Promise任务rejected，则只返回 rejected 任务的结果。
function promiseAll(promises){
    return new Promise(resolve,reject)=>{
        if(!Array.isArray(promises)){
            return reject(new Error("arguments must be an array"))
        }
        let promisecounter=0,
            promiseNum=promises.length,
            //保存结果
            resolvedValues=new Array(promisNum);
        for(let i =0;i<promiseNum;i++){
            (function(i){
                Promise.resolve(promises[i]).then((value)=>{
                    promiscounter++;
                    resolvedValues[i]=value;
                    if(promiseCounter==promiseNum) {
                        return resolve(resolvedValues);
                    }
                }).catch(err=>{reject(err)})
            })(i)
        }
}
```

rem的实现原理

```
function setRem(){
    let doc=document.documentElement;
    let width=doc.getBoundingClientRect().width;
    let rem=width/75
    doc.style.fontsize=rem+'px';
}
addEventListener("resize",setRem);
```

双向数据绑定实现原理

```
//双向数据绑定
let obj={};
let input=document.getElementById('input');
let span=document.getElementById('span');

//数据劫持
Object.defineProperty(obj,'text',{
```

```
    configurable:true,
    enumerable:true,
    get(){
        console.log('获取数据');
    },
    set(newVal){
        console.log('数据更新');
        input.value=newVal;
        span.innerHTML=newVal;
    }
})

//监听
input.addEventListener('keyup',function(e){
    obj.text=e.target.value;
})
```

EventBus组件通信原理

```
//组件通信
//一个触发  监听的过程

class EventEmitter{
    constructor(){
        //存储事件
        this.events=this.events||new Map();
    }

    addListener(type,fn){
        if(!this.events.get(type)){
            this.events.set(type,fn);
        }
    }

    emit(type){
        let handler=this.events.get(type);
        handler.apply(this,[...arguments].slice(1))
    }
}

//测试
let emitter=new EventEmitter();
emitter.addListener('ages',age=>{console.log(age)})
emitter.emit('ages',24);
```

常见算法题

反转字符串

给定一个字符串，你需要反转字符串中每个单词的字符顺序，同时仍保留空格和单词的初始顺序。

示例 1:

输入: "Let's take LeetCode contest"

输出: "s'teL ekat edoCteeL tsetnoc"

注意：在字符串中，每个单词由单个空格分隔，并且字符串中不会有任何额外的空格。

思路：先用利用空格分割每个单词，单词变数组，再在每个数组内进行反转

```
let reverseWord=(str) => {
     return str.split(' ').map(item => {
         return item.split('').reverse().join('')
     }).join(' ')
 }
```

eetcode最快的思路:

先把每个字符都颠倒过来,tsetnoc edoCteeL ekat s'teL

之后按照空格分隔开， 重新排序

```
let reverseWord=(str) => {

    return str.split('').reverse().join('').split(' ').reverse().join(' ')

}
```

最长连续子串

给定一个字符串 s，计算具有相同数量0和1的非空(连续)子字符串的数量，并且这些子字符串中的所有0和所有1都是组合在一起的。

重复出现的子串要计算它们出现的次数。

示例 1 :

输入: "00110011"

输出: 6

解释: 有6个子串具有相同数量的连续1和0：“0011”，“01”，“1100”，“10”，“0011” 和 “01”。

请注意，一些重复出现的子串要计算它们出现的次数。

另外，“00110011”不是有效的子串，因为所有的0（和1）没有组合在一起。

```
/**
 * 思路1:利用正则表达式来写
 * 0011001 每次都从第一个字符开始，直到遇到非它的。比如说第一次，00截止，长度为2，那么就从后检查有没有满足0011的，输出0011。
 *                                                    第二次，从第二个0开始，后面遇到1就截止，长度为1.那么就从后面match有没有满足01的，输出01
 *                                                    第三次，从1开始，11截止，长度为2，那么就从后检测有没有1100的，输出1100
 * 只需要把上述过程实现即可，需要每次减1位
 */
export default (str) => {
    // 保存数据
    let r = []
        // 给定任意子输入都返回第一个符合条件的子串
    let match = (str) => {
        let j = str.match(/^(0+|1+)/)[0]
        let o = (j[0] ^ 1).toString().repeat(j.length)
        let reg = new RegExp(`^(${j}${o})`)
```

```
        if (reg.test(str)) {
            return RegExp.$1
        } else {
            return ''
        }
    }
    for (let i = 0, len = str.length - 1; i < len; i++) {
        let sub = match(str.slice(i))
        if (sub) {
            r.push(sub)
        }
    }
    return r
}
/**
 *   思路2
 *   000111必定有三个子串
 *   00011必定有两个子串
 *   0111必定有1个子串
 *   以此类推，每两组数据之间长度最短的值为子串的数量
 */
export default (str) => {
    let n = 0,
        arr = s.match(/([1]+)|([0]+)/g)
    for (let i = 0; i < arr.length - 1; i++) {
        n += Math.min(arr[i].length, arr[i + 1].length)
    }
    return n
}


/**
 * 常规思路：判断个数，迭代输出
 */
export default (str) => {
    var prev = 0,
        curr = 1,
        res = 0
    for (var i = 1; i < s.length; i++) {
        if (s[i] === s[i - 1]) {
            curr++
        } else {
            prev = curr
            curr = 1
        }
        if (prev >= curr) {
            res++
        }
    }
    return res
}
```

重复的子串

给定一个非空的字符串，判断它是否可以由它的一个子串重复多次构成。给定的字符串只含有小写英文字母，并且长度不超过10000。

示例 1:

输入: "abab"

输出: True

解释: 可由子字符串 "ab" 重复两次构成。

```
export default (s) => {
    //正则表达式
    // \w字符 + 至少一个 ()\1 模式匹配()内的
    var reg = new RegExp(/^(\w+)\1+$/);
    return reg.test(s)
}
```

判断数组是否连续的问题

[0,10] [8,10] [10,30] true

[0,10] [25 30] [8 20] false

写一个方法来实现这个功能，判断数组是不是连续的?

思路一:

先排序，每个子数组中的第一个元素跟前一个数组最大值做比较，比它小就说明是连续的。

思路二:

用一个新的数组来判断，直接根据每个数组的首尾，给一个新的数组全部赋值，比如都是1。最后只需要判断数组includes(undefined)有没有，就可以判断是不是连续的了。

思路三:

就是找子数组起始坐标的最大值和结束坐标的最小值

```
//思路2:
function check(lines){
    let result=[];
    for(let i=0;i<lines.length;i++){
        let start=lines[i][0];
        let end=lines[i][1];
        for(let j=start;j<=end;j++){
            result[j]=1;
        }
    }
    //此时获取数组中有值的起始坐标,因为不一定数组是从0开始的
    let begin =result.indexOf(1);
    //把数组截取出来，然后判断是不是连续的
    return !result.slice(begin).includes(undefined);
}

//思路3:
function check(lines){
    //取起始点最大值、结束点最小值
    let maxstart=0,minend=0;
    for(let i=0;i<lines.length;i++){
```

```
        if(lines[i][0]> maxstart){
            maxstart=lines[i][0];
        }if(lines[i][1]<minend){
            minend=lines[i][0];
        }
    }
    //起始点最大比结束点最小大，那么一定是重合的
    return maxstart>minend ? true:false
}
```

九宫格手机键盘字母排列组合

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

给出数字到字母的映射(九宫格)。注意 1 不对应任何字母。

示例:

输入："23"

输出：["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

```
let phonenum= (str) => {
    //为空返回空
    if (!str) { return [] }

    //map映射关系
    let map = ['', 1, 'abc', 'def', 'ghi', 'jkl', 'mno', 'pqrs', 'tuv', 'wxyz'];
    //输入字符按字符分隔变成数组
    let num = str.split('');
    //保存键盘映射后的字母内容 23=>['abc','def']
    let code = []
    num.forEach(item => {
        if (map[item]) {
            code.push(map[item])
        }
    })

    //只有一个数字的情况
    if (code.length <= 1) {
        return code[0].split('')
    }

    //字符组合
    let comb = (arr) => {
        //保存前两个组合的结果
        let tmp = []
        for (let i = 0, il = arr[0].length; i < il; i++) {
            for (let j = 0, jl = arr[1].length; j < jl; j++) {
                tmp.push(`${arr[0][i]}${arr[1][j]}`)
            }
        }
        //去掉组合后前两个，插入新组合
        arr.splice(0, 2, tmp)
        if (arr.length > 1) {
            comb(arr)
        } else {
            return tmp
```

```
        }
        return arr[0]
    }

    return comb(code)
}
```

格雷编码

格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有一个位数的差异。

给定一个代表编码总位数的非负整数 n，打印其格雷编码序列。格雷编码序列必须以 0 开头。

示例:

输入: 2

输出: [0,1,3,2]

解释:

00 - 0

01 - 1

11 - 3

10 - 2

```
let GrayCode= (num) => {
    if (num === 0) return ['0']
        //递归计算为n的格雷编码
    let getGrayCode = (n) => {
        if (n === 1) {
            return ['0', '1']
        } else {
            let prev = getGrayCode(n - 1)
            let TempResult = []
            let maxlength = Math.pow(2, n) - 1
            for (let i = 0; i < prev.length; i++) {
                TempResult[i] = `0${prev[i]}`
                TempResult[maxlength - i] = `1${prev[i]}`
            }
            return TempResult
        }
    }
    return getGrayCode(num).map((item) => {
        return parseInt(item, 2)
    });
}
```

种花问题

假设你有一个很长的花坛，一部分地块种植了花，另一部分却没有。可是，花卉不能种植在相邻的地块上，它们会争夺水源，两者都会死去。

给定一个花坛（表示为一个数组包含0和1，其中0表示没种植花，1表示种植了花），和一个数 n 。能否在不打破种植规则的情况下种入 n 朵花？能则返回True，不能则返回False。

示例 1:

输入: flowerbed = [1,0,0,0,1], n = 1

输出: True

示例 2:

输入: flowerbed = [1,0,0,0,1], n = 2

输出: False

```
let palceFlower= (arr, n) => {
    let max = 0
        // 右边界补充[0,0,0],最后一块地能不能种只取决于前面的是不是1，所以默认最后一块地的右侧是0（无须考虑右侧边界有阻碍）
    arr.push(0)
    for (let i = 0; i < arr.length - 1; i++) {
        if (arr[i] === 0) {
            if (i === 0 && arr[1] === 0) {
                max++
                i = i + 1
            } else if (arr[i - 1] === 0 && arr[i + 1] === 0) {
                max++
                i = i + 1
            }
        }
    }
    return max >= n
}
```

卡牌游戏

给定一副牌，每张牌上都写着一个整数。 此时，你需要选定一个数字 X，使我们可以将整副牌按下述规则分成 1 组或更多组：

每组都有 X 张牌。组内所有的牌上都写着相同的整数。仅当你可选的 X >= 2 时返回 true。

示例 1：

输入：[1,2,3,4,4,3,2,1]

输出：true

解释：可行的分组是 [1,1]，[2,2]，[3,3]，[4,4]

示例 2：

输入：[1,1,1,2,2,2,3,3]

输出：false

解释：没有满足要求的分组。

```
let deckCard= (arr) => {
    //其实这道题就是在求相同数字数目的最大公约数的问题
    //先写求最大公约数的方法
```

```
    let gcd = (a, b) => {
            if (b === 0) {
                return a
            } else {
                return gcd(b, a % b)
            }
        }
        //记录每张牌的数目
    let group = []
    let tmp = {}
    arr.forEach(element => {
        tmp[element] = tmp[element] ? tmp[element] + 1 : 1
    })
    for (let v of Object.values(tmp)) {
        group.push(v)
    }

    while (group.length > 1) {
        let a = group.shift()
        let b = group.shift()
        let v = gcd(a, b)
        if (v === 1) {
            //最小公约数是1，false
            return false
        } else {
            //放回前两个数的最大公约数继续计算
            group.unshift(v)
        }
    }
    //判断最后得出的最小公约数是否是1
    return group.length ? group[0] > 1 : false
}
```

子串匹配

给定一个字符串 s 和一些长度相同的单词 words。

找出 s 中恰好可以由 words 中所有单词串联形成的子串的起始位置。

注意子串要与 words 中的单词完全匹配，中间不能有其他字符，但不需要考虑 words 中单词串联的顺序。

示例 1：

输入：

s = "barfoothefoobarman",words = ["foo","bar"]

输出：[0,9]

解释：

从索引 0 和 9 开始的子串分别是 "barfoor" 和 "foobar" 。

输出的顺序不重要, [9,0] 也是有效答案。

示例 2：

输入：

s = "wordgoodgoodgoodbestword",words = ["word","good","best","word"]

输出：[]

```
//思路：words[a,b,c]  递归排列组合[a]  第一位 b c中选一位放在第二位
//                                [b] 第一位 a c中选一位放在第二位
//                                [c] 第一位 a b中选一位放在第二位

//先排列组合生成子串组合，然后看看对应字符串有无对应的即可
export default (str, words) => {
    //数组长度
    let num = words.length
        //结果
    let result = []
        //生成递归的函数
    let range = (r, _arr) => {
        //边界，计算完毕
        if (r.length === num) {
            result.push(r)
        } else {
            _arr.forEach((item, idx) => {
                let tmp = [].concat(_arr);
                tmp.splice(idx, 1); //丢出第一位，生成第二位待选数组
                range(r.concat(item), tmp);
            })
        }
    }
    range([], words);
    //子串获取完成，检查有没有匹配的子串
    return result.map(item => {
        return str.indexOf(item.join(''))
    }).filter(item => item !== -1).sort()
}
```

```
// 利用查找每个单词在字符串的位置，然后通过计算这些位置是不是连续的。
// 比如 abforbarcd,[for,bar]，那么for的起始位置是2,bar的起始位置是5;
// 说明这两个单词是连续的2+3(for的长度)=5
//       for:[{start:2,end:5}]
//       bar:[{start:5,end:8}]
// 判断上一个单词的end和下一个单词的start是不是相同来计算两个单词是不是挨着
export default (str, words) => {
    // 计算字符串的总长度
    let strLen = str.length
        // 计算所有的单词数量
    let wordsLen = words.length
        // 计算所有单词出现的起始位置和截止位置
    let pos = {}
        // 如果字符串的长度小于所有单词的总长度直接返回
    if (strLen < words.join('').length) {
        return []
    }
    // 遍历所有单词查找在字符串中的起始位置和截止位置
    words.every(word => {
            if (pos[word]) {
                return true
```

```
            }
            let wl = word.length
            let tmp = []
            for (let i = 0, len = strLen - wl, idx; i <= len; i++) {
                idx = str.slice(i).indexOf(word)
                if (idx > -1) {
                    if (idx === 0) {
                        tmp.push({
                            start: i,
                            end: i + wl
                        })
                    } else if (str[i + 1] !== word[0]) {
                        i += idx - 1
                    }
                } else {
                    break
                }
            }
            // 如果没有匹配到单词终止遍历
            if (tmp[0] === undefined) {
                return false
            } else {
                // 保存当前单词的位置，遍历下一个单词
                pos[word] = tmp.sort((a, b) => a.start - b.start)
                return true
            }
        })
        // 只要有一个单词没找到说明不能匹配到连续的字符串
    if (words.find(item => !pos[item])) {
        return []
    }
    let result = []
        // 计算所有单词的位置
    let match = (poses) => {
        // 记录是不是所有单词都被匹配到了，每一次都应该把所有单词都包括进来并且是相邻的
        let record = []
        let len = Object.keys(poses).length
            // 如果没有单词的位置说明处理结束了
        if (len < 1) {
            return -1
        }
        while (1) {
            // 每次循环应该把记录清空
            record.length = 0
                // 按照起始位置进行升序排序
            let minV = Number.MAX_SAFE_INTEGER
            let minK = ''
                // 优先找到所有单词其实位置最小的单词开始匹配
            for (let [k, v] of Object.entries(poses)) {
                if (!v.length) {
                    return false
                } else {
                    if (v[0].start < minV) {
                        minK = k
                        minV = v[0].start
                    }
                }
            }
```

```
            if (!minK) {
                return false
            }
            // 起始位置最小的单词
            let first = poses[minK].shift()
            if (!first) {
                return false
            }
            // 记录下这个起始位置
            let start = first.start
                // 记录words列表中的单词
            record.push(words.findIndex(item => item === minK))
                // 每次循环要匹配到所有单词
            for (let i = 1; i < wordsLen; i++) {
                for (let j = 0, next; j < wordsLen; j++) {
                    if (record.includes(j)) {
                        continue
                    }
                    if (poses[words[j]][0] === undefined) {
                        return false
                    }
                    next = poses[words[j]].find(item => item.start ===
first.end)
                    if (next) {
                        record.push(j)
                        first = next
                        break
                    }
                }
            }
            // 如果所有单词的顺序是挨着的，记录下当前的起始位置
            if (record.length === wordsLen && !record.find(item => item ===
undefined)) {
                result.push(start)
            }
        }
    }
    match(pos)
        // 对 result 去重，如 result=[1,1,2,3] [...new Set(result)]===[1,2,3]
    return [...new Set(result)]
}
```

复原IP地址

给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式。

示例:

输入: "25525511135"

输出: ["255.255.11.135", "255.255.111.35"]

```
//思路:
// IP三个点分成四部分 每部分0-255 每部分最多不超过3位
// 示例 25525511135 第一部分可能是 2 25 255
//                          如果第一部分是2 那么 第二部分 5 55
```

```
//                                                如果第二部分 5 那么 第三部分 剩下的超过6位 不满
足

export default (str) => {
    //保存所有可能的结果
    let r = []
        //递归函数  上次处理结果，待处理字符串
    let search = (cur, sub) => {
        //已经分为四部分且四部分长度相等
        if (cur.length === 4 && cur.join('') === str) {
            r.push(cur.join('.'));
        } else {
            //每部分长度最多是3位
            for (let i = 0, tmp, len = Math.min(3, sub.length); i < len; i++) {
                //待处理子串，取 1位 2位 3位
                tmp = sub.substr(0, i + 1);
                if (tmp < 256) {
                    //之前的合并进去  子串位置+1
                    search(cur.concat([tmp]), sub.substr(i + 1))
                }
            }
        }
    }
    search([], str);
    return r;
}
```

完全二叉搜索树

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

节点的左子树只包含小于当前节点的数。

节点的右子树只包含大于当前节点的数。

所有左子树和右子树自身必须也是二叉搜索树。

示例 1:

输入:

```
  2
 / \
1   3
```

输出: true

示例 2:

输入:

```
  5
 / \
```

1 4

/ \

3 6

输出: false

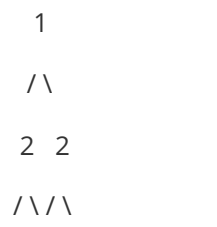
解释: 输入为: [5,1,4,null,null,3,6]。

```
class Node {
    constructor(val) {
        this.val = val;
        this.left = this.right = undefined;
    }
}
class Tree {
    constructor(data) {
        let root = new Node(data.shift());
        //遍历数据，插入树
        data.forEach(item => {
            this.insert(root, item);
        });
        return root;
    }

    insert(node, data) {
        if (node.val > data) {
            if (node.left === undefined) {
                node.left = data;
            } else {
                this.insert(node.left, data);
            }
        } else {
            if (node.right === undefined) {
                node.right = data;
            } else {
                this.insert(node.right, data);
            }
        }
    }

    static walk(root) {
        if (!root.left && !root.right) {
            return true;
        } else if ((root.left && root.left.val > root.val) || (root.right &&
root.right.val < root.val)) {
            return false;
        } else {
            return Tree.walk(root.left) && Tree.walk(root.right);
        }
    }
}
```

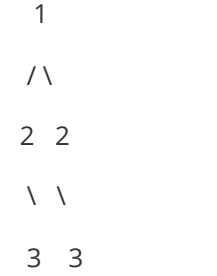
镜像对称树

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。



但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的:



```
class Node {
    constructor(val) {
        this.value = val;
        this.left = this.right = undefined;
    }
}
//构造二叉树
class Tree {
    constructor(data) {
        //临时存储所有节点，方便寻找父子节点
        let nodeList = [];
        //顶点节点
        let root;
        for (let i = 0; i < data.length; i++) {
            let node = new Node(data[i]);
            nodeList.push(node);
            if (i > 0) {
                //计算属于哪一层
                let n = Math.floor(Math.sqrt(i + 1));
                //当前层起始点的索引
                let q = Math.pow(2, n) - 1;
                //上一层起始点的索引
                let p = Math.pow(2, n - 1) - 1;
                //当前节点的父节点
                let parent = nodeList[p + Math.floor((i - q) / 2)];
                if (parent.left) {
                    parent.right = node;
                } else {
                    parent.left = node;
                }
            }
        }
        root = nodeList.shift();
        nodeList = null;
        return root;
```

```
    }
    static isSymmetry(root) {
        if (!root) { return false }
        let walk = (left, right) => {
            if (!left && !right) { return true }
            if ((left && !right) || (!left && right) || left.value !==
right.value) {
                return false;
            }
            return walk(left.left, right.right) && walk(left.right, right.left);
        }
        return walk(root.left, root.right)
    }
}
```

常见应用题

Promise超时处理

```
//Promise 超时处理
//if < 1s return as usual
//if  >1s return timeout

//直接通过一个promise里面抢resolve和reject的执行权
const myFetch=()=>{
    return new Promise((resolve,reject)=>{
        let timer=setTimeout(()=>{
            reject('timeout')
        },1000);
        fetch(url,{headers:{"content-type":"application/text"}})
        .then(res=>res.json()).then(
            res=>{
                clearTimeout(timer);
                resolve(res);
                timer=null;
            }
        )
    })
}


//利用Promise.race()先返回最快的那个resolve
const myFetch=()=>{
    Promise.race([
        new Promise((resolve,reject)=>{
            setTimeout(resolve('timeout'),1000);
        }),
    fetch(url,{headers:{"content-type":"application/text"}})
    .then(res.res.json()).then(
        res=>{
            resolve(res);
        })
    ] )
}
```

```
//用async 真呀么真快乐
const myFetch=async ()=>{
    let start = new Date().getTime()
    let result = await (await fetch(url)).json()
    let time = new Date().getTime() -start
    if (time > 1000) return time;
    else return result
}
```

有N个Promise,要求按顺序执行

```
//n个promise的数组，按要执行的顺序排好
var promiseArr=[new Promise(()=>{console.log('1')}),
                new Promise(()=>{console.log('2')}),
                new Promise(()=>{console.log('3')}),
                new Promise(()=>{console.log('4')})
               ];
function* donebyOrder(arr){
    let len=arr.length;
    for(let i=0;i<len;i++){
         yield arr[i];
    }
}

let gen=donebyOrder(promiseArr)
for(let i=1;i<promiseArr.length;i++){
   gen.next();
}
```

剑指Offer

正则表达式匹配

请实现一个函数用来匹配包括'.'和'*'*的正则表达式。模式中的字符'.'表示任意一个字符，而*'*表示它前面的字符可以出现任意次（包含0次）。 在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"和"ab*ac*a"匹配，但是与"aa.a"和"ab*a"均不匹配

```
function match(s, pattern)
{
    //正常模式:完全匹配
    //.模式：跳过一位匹配即可
    //* 比较复杂，可能0个，可能很多个
    //考虑每次匹配完，就把前面的丢掉，继续递归看后面的满不满足条件
    let isMatch=(s,p)=>{
        //边界是字符串和正则都为空
        if(p.length<=0){
            return !s.length;
        }
        //开始判断当前第一个字符是不是匹配的
        let match=false;
        if(s.length>0&&(s[0]===p[0]||p[0]==='.')){
            match=true;
```

```
        }
        //处理有*的情况
        if(p.length>1&&p[1]==='*'){
            //匹配0个，或者丢弃一个继续匹配   sssa s*a 丢掉一个s 匹配ssa s*a
            return isMatch(s,p.slice(2))||(match&&isMatch(s.slice(1),p));
        }else{
            //正常匹配
           return  match&&isMatch(s.slice(1),p.slice(1));
        }
    }
    return isMatch(s,pattern);
}
```

替换空格

请实现一个函数，将一个字符串中的每个空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

```
//正则表达
function replaceSpace(str){
    return str.replace(new RegExp(/\s/g),'%20');
}
```

数组中重复的数字

在一个长度为n的数组里的所有数字都在0到n-1的范围内。 数组中某些数字是重复的，但不知道有几个数字是重复的。也不知道每个数字重复几次。请找出数组中任意一个重复的数字。 例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是第一个重复的数字2。

```
function duplicate(numbers, duplication)
{
    // write code here
    //这里要特别注意~找到任意重复的一个值并赋值到duplication[0]
    //函数返回True/False
    if(numbers===null&&numbers<0){return false;}
    for(let i=0,tmp;i<numbers.length;i++){
        if(numbers[i]<0&&numbers[i]>length-1) return false;
        //不等于就交换
        let temp=undefined;//临时变量保存交换结果
        while(numbers[i]!==i){
            //找到第一个就返回
            if(numbers[i]===numbers[numbers[i]]){
                duplication[0]=numbers[i];
                return true;
            }
            temp=numbers[i];
            numbers[i]=numbers[temp];
            numbers[temp]=temp;
        }
    }
    return false;
}
```

不修改数组找出重复的数字

在一个长度为n+1的数组里，,所有数字都在1-n的范围中，所以数组中至少有一个数是重复的。 请找出数组中任意一个重复的数字，但不可以修改这个数组。 例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是2或者3。

思路一就是申请一个新的数组，先复制，然后和上面那道题一样。

但如果要避免使用O(n)的辅助空间，那就可以借鉴二分查找的思路。从1-n的数字从中间数字m分为两个部分，如果1-m部分数字数目超过m，那么这一半的区间里一半为m+1~n。如果1~m的数字的数目超过m，那么这一半的区间里一定包含重复数字。

{2,3,5,4,3,2,6,7}，数字在1-7的范围内，从中间3开始，1-3 和4-7；统计1-3这3个数字在在数组中出现的次数。一共出现了4次，那么一定有重复的。接着再1-3分为1-2和3-3，统计1-2在数组中出现的次数，有2次；统计3-3在数组中出现的次数，有2次，那么3重复了。

```
//输入一个数组，数字的范围
function duplicate(arr,nums){
    if(arr==null&&nums<=0)return -1;
    let start=1,end=nums;
    while(end>=start){
        let middle=Math.ceil((end-start)/2)+start;
        let count=countnum(arr,nums,start,middle);
        if(end ===start){
           if(count>1) return start;
        }
        if(count>middle-start+1){
            end=middle;
        }else{
            start=middle+1;
        }
    }
    return -1;
}
//计算区间内数字出现的个数
function countnum(arr,nums,start,end){
    if(nums==null) return 0;
    let counts=0;
    for(let i=0;i<nums;i++){
        if(nums[i]>=start&&nums[i]<=end){
            counts++;
        }
    }
    return counts;
}
```

二维数组中的查找

在一个二维数组中（每个一维数组的长度相同），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。

```
//暴力
function Find(target, array)
{
    let result=false;
    let len=array.length;
    for(let i=0;i<len;i++){
```

```
        for(let j=0;j<array[i].length;j++){
            if (array[i][j]===target){
                result=true;
            }
        }
    }
    return result;
}
```

这道题说了，每一行都是从左到右递增的顺序，每一列都是从上到下的递增的顺序，所以完全可以利用这一点进行。比如从二维矩阵的右上角开始，目标比它小，只需要在它右边查找，目标比它大，那直接向下查找即可。

当target小于元素`a[row][col]`时，那么target必定在元素a所在行的左边,即col--；当target大于元素`a[row][col]`时，那么target必定在元素a所在列的下边,即row++；

```
function Find(target,array){
    //右上角a[0][col]
    let row=0,col=array[0].length-1;
    while(row<array.length&&col>=0){
        if (target==array[row][col]){
            return true;
        }
        if(target<array[row][col]){
            col--;
        }else{
            row++;
        }
    }
    return false;
}
```

从尾到头打印链表

输入一个链表，按链表值从尾到头的顺序返回一个ArrayList。

剑指Offer书上的题解是：从尾到头输出可以想象是一个递归的过程，每次都先输出后面一个节点的值。

不过用js不用这么麻烦，正常遍历，每次都从数组头部插入，这样结果就是逆序的了。

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function printListFromTailToHead(head)
{
    let result=[];
    while(head){
        result.unshift(head.val);
        head=head.next;
    }
    return result;
}
```

重建二叉树

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：

先序遍历的第一个节点就是它的根节点，通过这个根节点，可以在中序遍历中找到它的位置，它之前的就是左子树，后面就是右子树。通过左子树的数目可以在先序遍历结果中找到左子树，后面就是右子树。

可以通过递归的办法得出最后的结果。

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function reConstructBinaryTree(pre, vin)
{
    let node=null;
    if(pre.length>1){
        let head=pre[0];
        let vinheadindex=vin.indexOf(head);
        let vinleft=vin.slice(0,vinheadindex);
        let vinright=vin.slice(vinheadindex+1);
        pre.shift();//头节点丢掉
        let preleft=pre.slice(0,vinleft.length);
        let preright=pre.slice(vinleft.length);
        node={
            val:head,
            left:reConstructBinaryTree(preleft,vinleft),
            right:reConstructBinaryTree(preright,vinright)
        }
    }else if (pre.length===1){
        node={
            val:pre[0],
            left:null,
            right:null
            }
    }
    return node;
}
```

二叉树的下一个节点

给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。

思路：

如果这个节点有右子树，那么下一个节点就是右子树的最左子节点。

如果节点没有右子树：

节点是父节点的左子节点，那下一个节点是父节点。

节点是父节点的右子节点，那就向上找到一个是它父节点的左子节点的节点，如果节点存在，这个节点的父节点就是下一个节点。

```
/*function TreeLinkNode(x){
    this.val = x;
    this.left = null;
    this.right = null;
    this.next = null;
}*/
function GetNext(pNode)
{
    // write code here
    if(!pNode) return null;
    let p;
    //存在右子树
    if(pNode.right){
        p=pNode.right;
        while(p.left){
            p=p.left;
        }
    }else{
        //父节点
        p=pNode.next;
        if(p&&p.right==pNode){
            //是父节点的右节点，向上查一个父亲节点的左节点
             while(p.next && p.next.right == p){
                p = p.next;
            }
            if(p.next == null){
                p =  null;
            }else{
                p = p.next;
            }
        }
    }
    return p;
}
```

用两个栈实现队列

用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。

思路：

先依次把 a,b,c 插stack1中，如果要弹出a,把c,b压入stack2中，弹出a，此时stack1为空，再从stack2弹出b……依此类推。如果在中途插入新的元素，那么必须等stack2全部为空之后再进行操作。

```
let stack1=[],
    stack2=[];
function push(node)
{
    stack1.push(node);
}
function pop()
{
    if(stack2.length==0){
        //stack2为空，说明全部在stack1中，需要从那边压过来
```

```
        if(stack1.length==0){
            return null;
        }else{
            //依次压入
            let len =stack1.length;
            for(let i=0;i<len;i++){
                stack2.push(stack1.pop());
            }
            return stack2.pop();
        }
    }else{
        //stack2不为空直接弹
        return stack2.pop();
    }
}
```

斐波拉契数列

大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项（从0开始，第0项为0）。

n<=39

```
//递归方法
function Fibonacci(n){
    if(n<=0) return 0;
    if(n===1) return 1;
    return Fibonacci(n-1)+Fibonacci(n-2);
}
```

这样做的话效率很低，其实做的过程中的中间数都是重复的，把中间值保存一下，需要计算的时候查找，就能解决这个问题。

```
function Fibonacci(n){
    let res=[0,1];
    if(n<=1){return res[n];}
    let n1=0,n2=1;
    let tmp;
    for(let i=2;i<=n;i++){
        tmp=n1+n2;
        n1=n2;
        n2=tmp;
    }
    return tmp;
}
```

变种题 青蛙跳台阶

一只青蛙一次可以跳1级台阶或者2级台阶，求问该青蛙跳上一个n级台阶有多少种跳法。

其实就是斐波拉契数列，因为n级台阶的跳法看成是n的函数，f(n)。那么有两种跳法，一种只跳1级，即f(n-1);一种跳两级，即f(n-2)。那么f(n)=f(n-1)+f(n-2)。

```
function jumpFloor(number)
{
    let result=[0,1,2];
    if(number<=2){return result[number];}
    let res;
    let n1=1,n2=2;
    for(let i=3;i<=number;i++){
        res=n1+n2;
        n1=n2;
        n2=res;
    }
    return res;
}
```

拓展

一只青蛙可以跳1级、2级……n级。此时青蛙跳上一个n级的台阶有多少种跳法。

数学归纳法是f(n)=2^(n-1)

```
function jumpFloorII(number)
{
    return Math.pow(2,number-1);
}
```

不过,用位运算更快

```
function jumpFloorII(number)
{
    return 1<<(--number);
}
```

变种题2 覆盖矩形

可以用 `2*1` 的小矩形横着或者竖着去覆盖大矩形。请问用8个 `2*1` 的小矩形无重叠覆盖一个 `2*8` 的大矩阵总共有多少种方法?

先把 `2*8` 的覆盖方法记为f(8)，用第一个覆盖时，可以横着或者竖着。竖着放的时候，还剩下 `2*7` 的矩形，记为f(7)。横着放的时候，另一个也只能横着放，还剩下 `2*6` 的矩形，记为f(6)。所以还是斐波那契数列。

```
function rectCover(number)
{
    let res=[0,1,2];
    if(number<=2){return res[number];}
    let tmp,n1=1,n2=2;
    for(let i=3;i<=number;i++){
        tmp=n1+n2;
        n1=n2;
        n2=tmp;
    }
    return tmp;
}
```

旋转数组的最小数字

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。 输入一个非减排序的数组的一个旋转，输出旋转数组的最小元素。 例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。 NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。

思路：

暴力破解其实也挺快的，但是算法复杂度为O(n)。这样肯定不够优雅。

优雅的解题思路是，把旋转之后的数组看做是两个排序的子数组，而且前面子数组的元素都大于或者等于后面子数组的元素。最小的元素就是这两个子数组的分界线。

可以用两个指针首尾二分法查找，以上面的数组为例，开始指针指向3,2。二分法中间为5,大于3,它一定位于递增数列第一个子数组。把头部指针指向5。此时位于中间的1，小于2，位于递增数组第二个子数组，尾指针指向1。此时指针差为1，输出尾指针的元素即可。

特殊情况是当中间三个数的数值都一样的时候，那只能遍历输出了。

```
function minNumberInRotateArray(rotateArray)
{
    let start=0,end=rotateArray.length-1;//记录下标
    let middle=0;
    while(rotateArray[start]>=rotateArray[end]){
        if(end-start==1){
            middle=end;
            break;
        }
        middle=Math.floor((end+start)/2);
        if(rotateArray[middle]>=rotateArray[start]){
            start=middle;
        }else if(rotateArray[middle]<=rotateArray[end]){
            end=middle;
        }
        //如果下标start end middle的值一样，那只能顺序查找了

 if(rotateArray[start]===rotateArray[middle]&&rotateArray[start]===rotateArray[e
nd]){
            let res=rotateArray[start];
            for(let i=start+1;i<=end;i++){
                if(res>rotateArray[i]){
                    res=rotateArray[i];
                }
            }
            return res;
        }
    }
    return rotateArray[middle];
}
```

常规算法思路是上面这样，但在js中可以利用数组的API来计算……但这样就没啥意思了不是么。

```
function minNumberInRotateArray(rotateArray)
{
   return Math.min(...rotateArray);
}
```

矩阵中的路径

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则之后不能再次进入这个格子。 例如 a b c e s f c s a d e e 这样的3 X 4 矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

思路：

首先，在矩阵中任选一个格子作为路径的起点。如果路径上的第i个字符不是ch，那么这个格子不可能处在路径上的第i个位置。如果路径上的第i个字符正好是ch，那么往相邻的格子寻找路径上的第i+1个字符。除在矩阵边界上的格子之外，其他格子都有4个相邻的格子。

重复这个过程直到路径上的所有字符都在矩阵中找到相应的位置。由于回朔法的递归特性，路径可以被开成一个栈。当在矩阵中定位了路径中前n个字符的位置之后，在与第n个字符对应的格子的周围都没有找到第n+1个字符，这个时候只要在路径上回到第n-1个字符，重新定位第n个字符。

由于路径不能重复进入矩阵的格子，还需要定义和字符矩阵大小一样的布尔值矩阵，用来标识路径是否已经进入每个格子。 当矩阵中坐标为（row,col）的
格子和路径字符串中相应的字符一样时，从4个相邻的格子(row,col-1),(row-1,col),(row,col+1)以及(row+1,col)中去定位路径字符串中下一个字符。如果4个相邻的格子都没有匹配字符串中下一个的字符，表明当前路径字符串中字符在矩阵中的定位不正确，我们需要回到前一个，然后重新定位。
一直重复这个过程，直到路径字符串上所有字符都在矩阵中找到合适的位置。

```
function hasPath(matrix, rows, cols, path)
{
    if(path.length===0) return true;
    if(rows*cols< path.length) return false;

    let status=[];
    for(let i=0;i<rows;i++){
        status.push([]);
        for(let j=0;j<cols;j++){
            status[i][j]=false;
        }
    }
    //找到第一个符合的path
    for(let i=0;i<rows;i++){
        for(let j=0;j<cols;j++){
            if(matrix[i*cols+j]===path[0]){
                if(path.length===1){
                    return true;
                }
                status[i][j]=true;
                if(find(matrix,rows,cols,i,j,path.slice(1),status)){
                    return true;
                }
                status[i][j]=false;
            }
        }
    }
    return false;
}

function find(matrix,rows,cols,row,col,path,status){
    if(row>0 && matrix[(row-1)*cols+col]===path[0]&&status[row-1][col]===false){
        if(path.length===1){
            return true;
```

```
        }
        status[row-1][col]=true;
        if(find(matrix,rows,cols,row-1,col,path.slice(1),status)){
            return true;
        }
        status[row-1][col]=false;
    }
    if(row<rows-1&&matrix[(row+1)*cols+col]===path[0]&&status[row+1]
[col]===false){
        if(path.length===1){
            return true;
        }
        status[row+1][col]=true;
        if(find(matrix,rows,cols,row+1,col,path.slice(1),status)){
            return true;
        }
        status[row+1][col]=false;
    }
    if(col>0&&matrix[row*cols+col-1]===path[0]&&status[row][col-1]===false){
        if(path.length==1){
            return true;
        }
        status[row][col-1]=true;
        if(find(matrix,rows,cols,row,col-1,path.slice(1),status)){
            return true;
        }
        status[row][col-1]=false;
    }
    if(col<cols-1&&matrix[row*cols+col+1]===path[0]&&status[row][col+1]===false)
{
        if(path.length===1){
            return true;
        }
        status[row][col+1]=true;
        if(find(matrix,rows,cols,row,col+1,path.slice(1),status)){
            return true;
        }
        status[row][col+1]=false;
    }
    return false;
}
```

机器人的运动范围

地上有一个m行和n列的方格。一个机器人从坐标0,0的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是不能进入行坐标和列坐标的数位之和大于k的格子。 例如，当k为18时，机器人能够进入方格（35,37），因为3+5+3+7 = 18。但是，它不能进入方格（35,38），因为3+5+3+8 = 19。请问该机器人能够达到多少个格子?

从0,0开始移动，当它准备进入坐标为(i,j)的格子时，通过检查坐标的数位和来判断机器人能否进入，如果可以进入，再判断它是否能进入上下左右的四个格子。

```
function movingCount(threshold, rows, cols)
{
    // write code here
    var flag = [];
    for(var i=0;i<rows;i++){
```

```
        flag.push([]);
        for(var j =0;j<cols;j++){
            flag[i][j] = 0;
        }
    }
    return count(0,0,rows,cols,threshold,flag);
}
function count(i,j,rows,cols,threshold,flag){
    if(i<0||j<0||i>=rows||j>=cols||flag[i][j]||sumNum(i)+sumNum(j)>threshold){
        return 0;
    }
    flag[i][j] = 1;
    return count(i+1,j,rows,cols,threshold,flag)
    +count(i-1,j,rows,cols,threshold,flag)
    +count(i,j+1,rows,cols,threshold,flag)
    +count(i,j-1,rows,cols,threshold,flag)+1;
}
function sumNum(num){
    var sum =0;
    do{
        sum+=num%10;
    }while((num = Math.floor(num/10))>0);
    return sum;
}
```

剪绳子

有一根长度为n的绳子，把绳子剪成m段（m、n都是整数，n>1且m>1），每段绳子的长度记为k[0],k[1].....k[m]。请问他们长度的乘积最大是多少？例如，绳子长度是8的时候，把它剪成长度分别为2、3、3的三段，此时最大乘积是18。

动态规划思路：在剪第一刀的时候，有n-1种可能的选择(1,2,3,...n-1)，因而f(n)=max(f(i)*f(n-i)) (0<i<n)。

先得到f(2) f(3) 再从它们得到f(4).....f(n)

```
function maxProductAfterCutting(length){
    if(length<2) return 0;
    if(length==2)return 1;
    if(length==3)return 2;

    let product=[0,1,2,3];

    for(let i=4,max;i<=length;i++){
        max=0;
        for(let j=1;j<=i/2;j++){
            let temp=product[j]*product[i-j];
            if(max<temp){
                max=temp;
            }
            product[i]=max;
        }
    }
    return product[length];
```

```
}
```

贪心思路：

如果按照这个策略来剪绳子：

当n大于等于5时，尽可能多的剪长度为3的绳子；剩下的长度为4时，把绳子剪成两段长度为2的绳子。

```
function maxProductAfterCutting(length){
    if(length<2) return 0;
    if(length==2)return 1;
    if(length==3)return 2;

    //尽可能多剪3
    let timesofthree=length/3;

    //绳子剩下的长度为4时，剪成2*2
    if(length-timesofthree*3==1){
        timesofthree-=1;
    }

    let timesoftwo=(length-timesofthree*3)/2;
    return Math.pow(3,timesofthree)*Math.pow(2,timesoftwo);
}
```

二进制中1的个数

输入一个整数，输出该数二进制表示中1的个数。其中负数用补码表示。

思路：如果最后一位是1，那么减去1，最后一位变成0，其他不变；如果最后一位是0，减去1之后，最右边的1会变成0，后面一位变成1。

把一个整数减去1然后与它做与运算，就能把整数最右边的1变成0，这样的话，有多少个1就可以进行多少次操作。

```
function NumberOf1(n){
    let count=0;
    while (n){
        ++count;
        n=(n-1)&n
    }
    return count;
}
```

数值的n次方

给定一个double类型的浮点数base和int类型的整数exponent。求base的exponent次方。

思路：假设指数为32，如果有16次方，只需要再平方一次就好；16次方是8次方的平方；8次方是4次方的平方……所以对偶数而言就是 a的二分之n方相乘；对奇数而言就是 a的二分之n-1方相乘再乘以a，于是可以用递归。

不过好像没有考虑到幂次是负数的情况，不过也不要紧：分开处理一下就好了。

```
function Power(base, exponent){
    let result=power(base,Math.abs(exponent));
```

```
    function power(base,exponent){
        if(exponent===0){return 1;}
        if(exponent===1){return base;}

        let result=Power(base,exponent>>1);
        result *=result;
        if(exponent& 0x1===1){
            //奇数再乘以a
            result *=base;
        }
        return result
    }
    return exponent>=0? result:1/result;
}
```