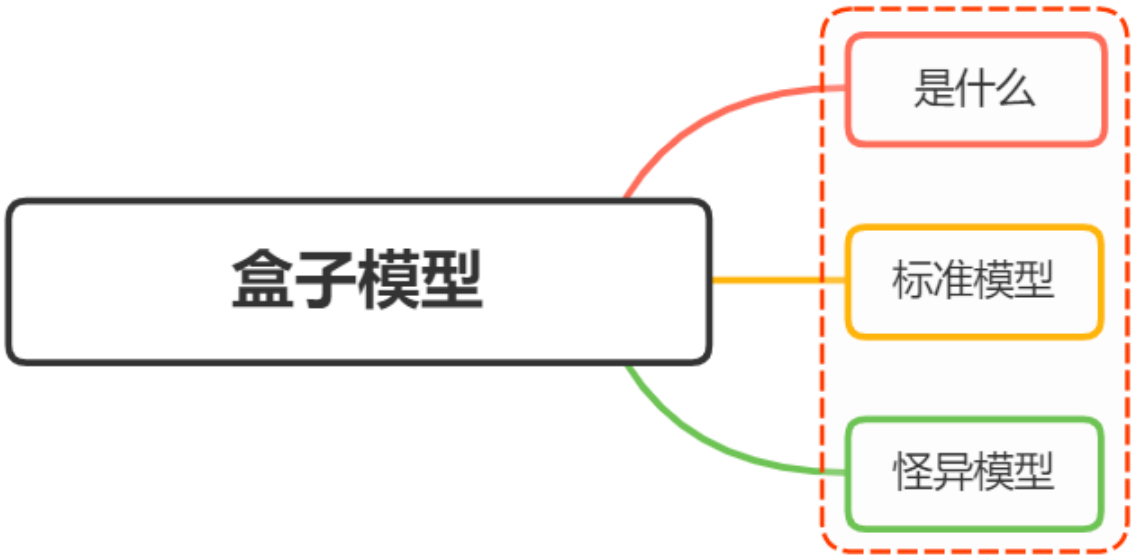
成都前端*重点面试宝典*v1.0

1.HTML5和CSS3

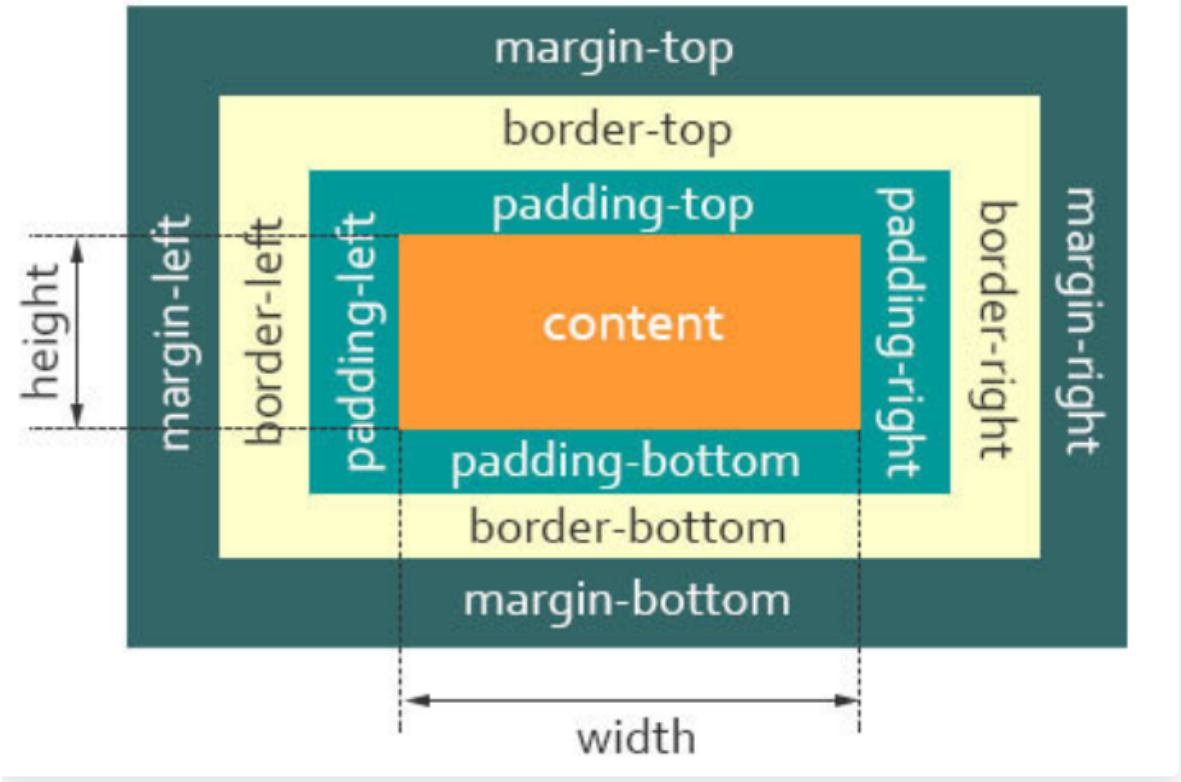
01.说说你对盒子模型的理解



一、是什么

当对一个文档进行布局（layout）的时候，浏览器的渲染引擎会根据标准之一的 CSS 基础框盒模型（CSS basic box model），将所有元素表示为一个个矩形的盒子（box）

一个盒子由四个部分组成：`content`、`padding`、`border`、`margin`



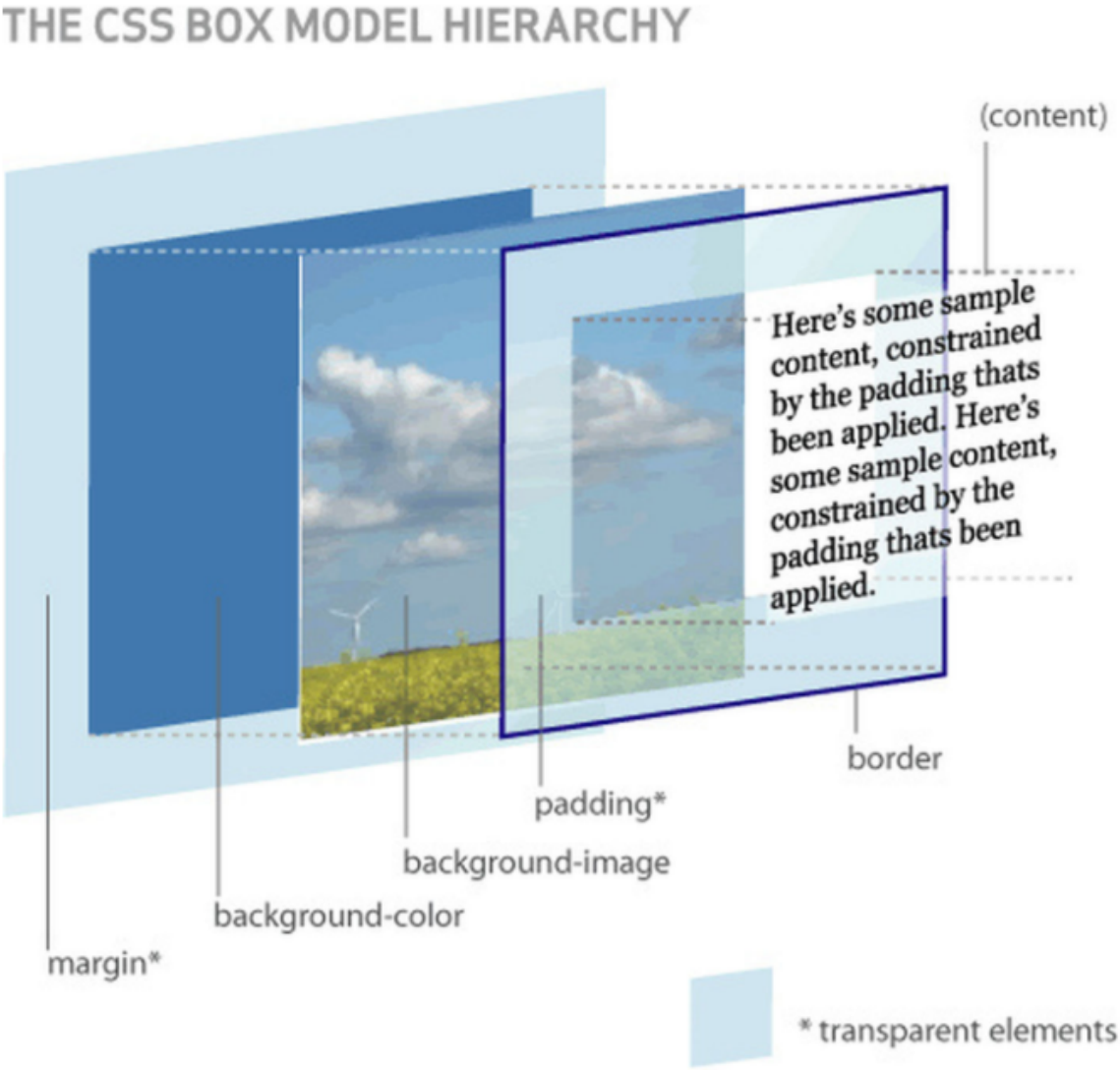
`content`，即实际内容，显示文本和图像

`boreder`，即边框，围绕元素内容的内边距的一条或多条线，由粗细、样式、颜色三部分组成

`padding`，即内边距，清除内容周围的区域，内边距是透明的，取值不能为负，受盒子的`background`属性影响

`margin`，即外边距，在元素外创建额外的空白，空白通常指不能放其他元素的区域

上述是一个从二维的角度观察盒子，下面再看看看三维图：



下面来段代码：

```
<style>
  .box {
    width: 200px;
    height: 100px;
    padding: 20px;
  }
</style>
<div class="box">
  盒子模型
</div>
```

当我们在浏览器查看元素时，却发现元素的大小变成了`240px`

这是因为，在`css`中，盒子模型可以分成：

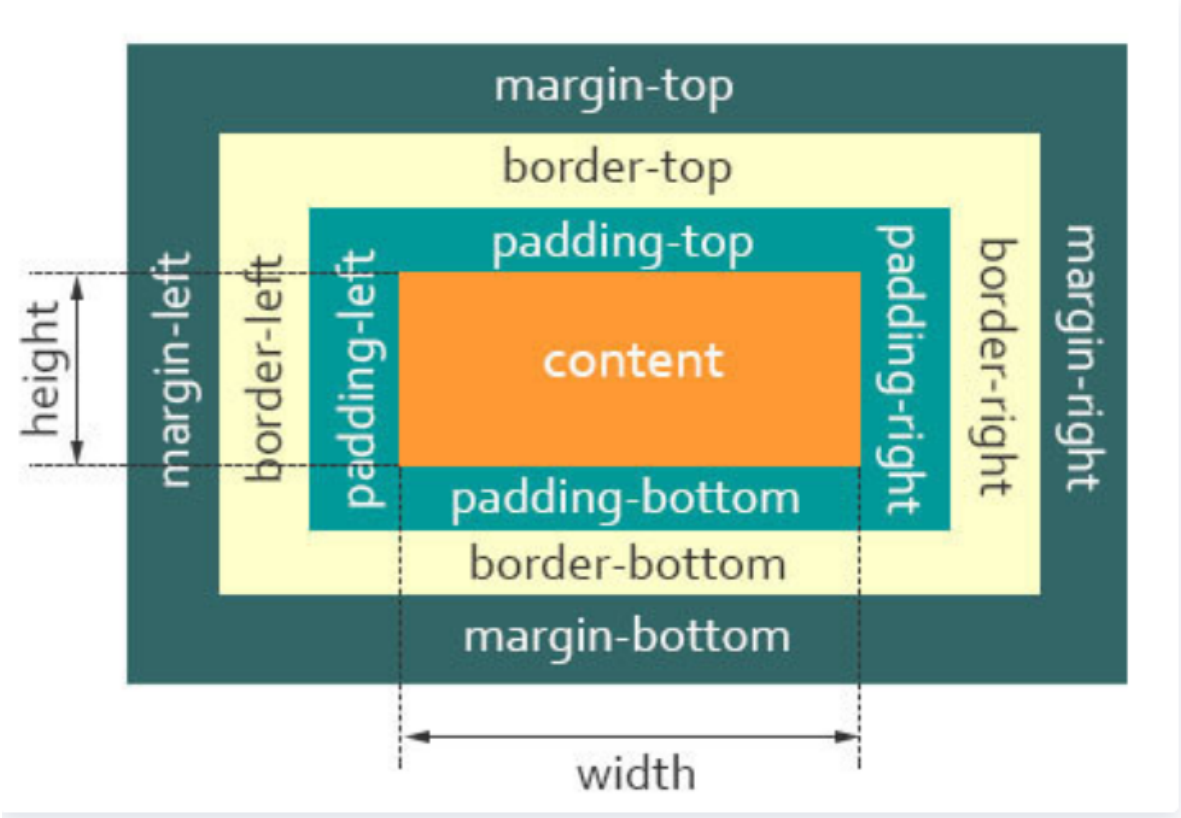
- W3C 标准盒子模型
- IE 怪异盒子模型

默认情况下，盒子模型为`W3C`标准盒子模型

二、标准盒子模型

标准盒子模型，是浏览器默认的盒子模型

下面看看标准盒子模型的模型图：



从上图可以看到：

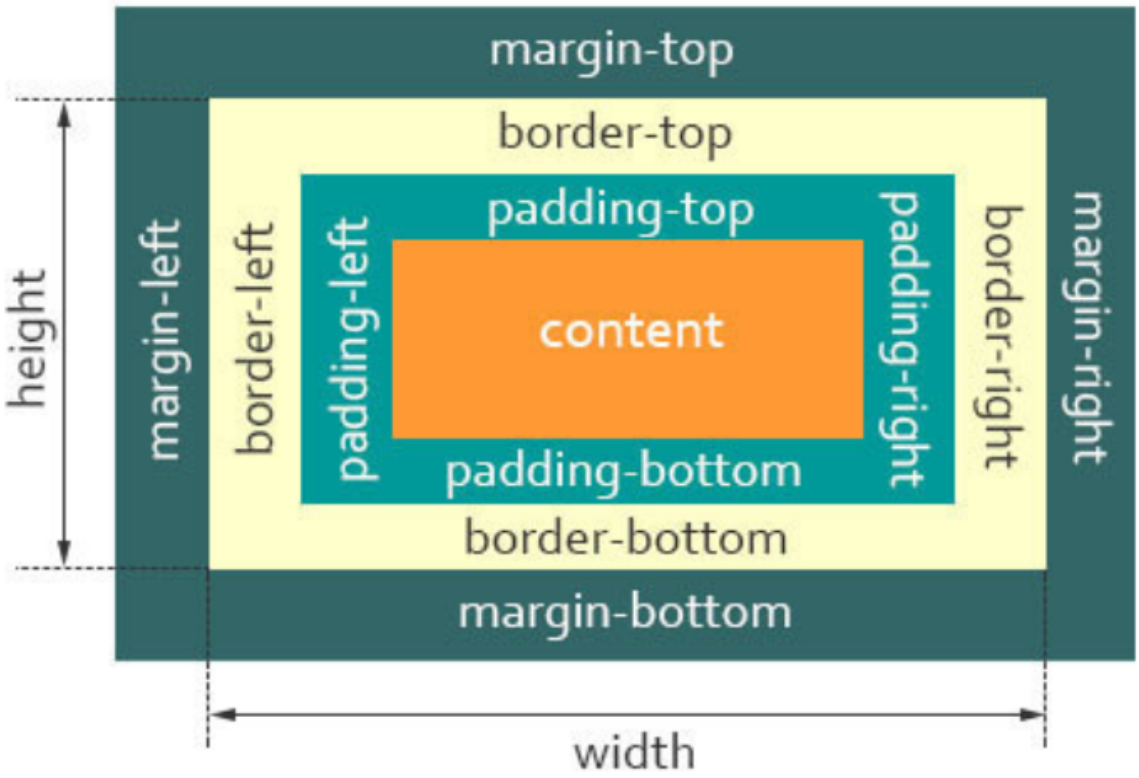
- 盒子总宽度 = width + padding + border + margin;
- 盒子总高度 = height + padding + border + margin

也就是，`width/height` 只是内容高度，不包含 `padding` 和 `border` 值

所以上面问题中，设置 `width` 为200px，但由于存在 `padding`，但实际上盒子的宽度有240px

三、IE 怪异盒子模型

同样看看IE 怪异盒子模型的模型图：



从上图可以看到：

- 盒子总宽度 = width + margin;
- 盒子总高度 = height + margin;

也就是，`width/height` 包含了 `padding` 和 `border` 值

Box-sizing

CSS 中的 box-sizing 属性定义了引擎应该如何计算一个元素的总宽度和总高度

语法：

```
box-sizing: content-box|border-box|inherit:
```

- content-box 默认值，元素的 width/height 不包含padding，border，与标准盒子模型表现一致
- border-box 元素的 width/height 包含 padding，border，与怪异盒子模型表现一致
- inherit 指定 box-sizing 属性的值，应该从父元素继承

回到上面的例子里，设置盒子为 border-box 模型

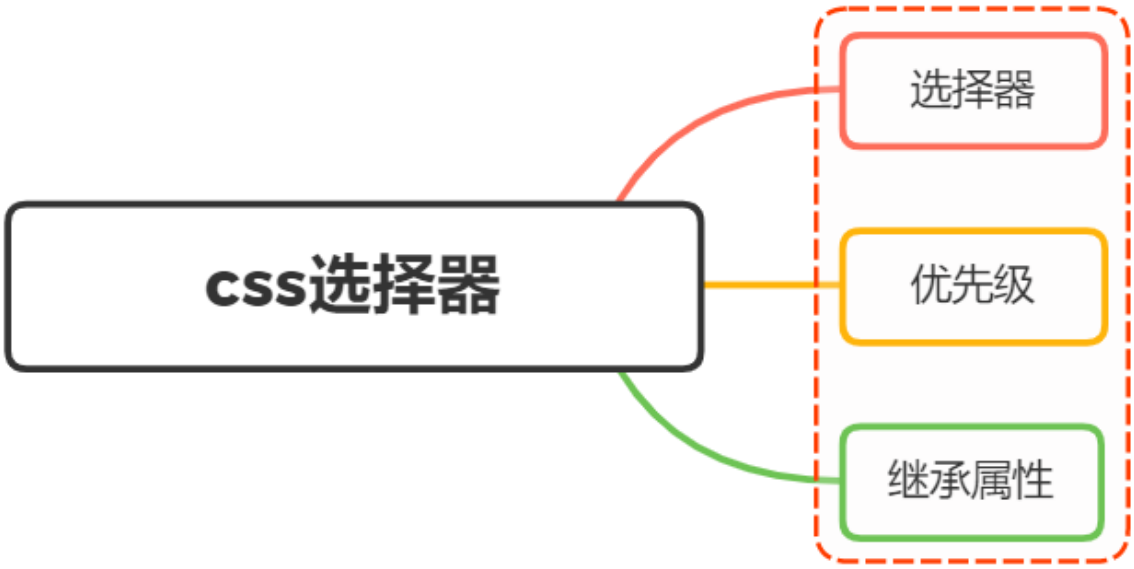
```
<style>
  .box {
    width: 200px;
    height: 100px;
    padding: 20px;
    box-sizing: border-box;
  }
</style>
<div class="box">
  盒子模型
</div>
```

这时候，就可以发现盒子的所占据的宽度为200px

参考文献

- https://developer.mozilla.org/zh-CN/docs/Web/CSS/CSS_Box_Model/Introduction_to_the_CSS_box_model
- https://developer.mozilla.org/zh-CN/docs/Web/CSS/box-sizing

02.css选择器有哪些？优先级？哪些属性可以继承？



一、选择器

CSS选择器是CSS规则的第一部分

它是元素和其他部分组合起来告诉浏览器哪个HTML元素应当是被选为应用规则中的CSS属性值的方式

选择器所选择的元素，叫做“选择器的对象”

我们从一个`Html`结构开始

```
<div id="box">
    <div class="one">
        <p class="one_1">
        </p >
        <p class="one_1">
        </p >
    </div>
    <div class="two"></div>
    <div class="two"></div>
    <div class="two"></div>
</div>
```

关于`css`属性选择器常用的有：

- id选择器（#box），选择id为box的元素
- 类选择器（.one），选择类名为one的所有元素

- 标签选择器（div），选择标签为div的所有元素
- 后代选择器（#box div），选择id为box元素内部所有的div元素
- 子选择器（.one>one_1），选择父元素为.one的所有.one_1的元素
- 相邻同胞选择器（.one+.two），选择紧接在.one之后的所有.two元素
- 群组选择器（div,p），选择div、p的所有元素

还有一些使用频率相对没那么多的选择器：

- 伪类选择器

```
:link ：选择未被访问的链接
:visited：选取已被访问的链接
:active：选择活动链接
:hover ： 鼠标指针浮动在上面的元素
:focus ： 选择具有焦点的
:first-child：父元素的首个子元素
```

- 伪元素选择器

```
:first-letter ：用于选取指定选择器的首字母
:first-line ：选取指定选择器的首行
:before ： 选择器在被选元素的内容前面插入内容
:after ： 选择器在被选元素的内容后面插入内容
```

- 属性选择器

```
[attribute] 选择带有attribute属性的元素
[attribute=value] 选择所有使用attribute=value的元素
[attribute~=value] 选择attribute属性包含value的元素
[attribute|=value]：选择attribute属性以value开头的元素
```

在`CSS3`中新增的选择器有如下：

- 层次选择器（p~ul），选择前面有p元素的每个ul元素
- 伪类选择器

```
:first-of-type 父元素的首个元素
:last-of-type 父元素的最后一个元素
:only-of-type 父元素的特定类型的唯一子元素
:only-child 父元素中唯一子元素
:nth-child(n) 选择父元素中第N个子元素
:nth-last-of-type(n) 选择父元素中第N个子元素，从后往前
:last-child 父元素的最后一个元素
:root 设置HTML文档
:empty 指定空的元素
:enabled 选择被禁用元素
:disabled 选择被禁用元素
:checked 选择选中的元素
:not(selector) 选择非 <selector> 元素的所有元素
```

- 属性选择器

```
[attribute*=value]：选择attribute属性值包含value的所有元素
[attribute^=value]：选择attribute属性开头为value的所有元素
[attribute$=value]：选择attribute属性结尾为value的所有元素
```

二、优先级

相信大家对`css`选择器的优先级都不陌生：

> 内联 > ID选择器 > 类选择器 > 标签选择器

到具体的计算层面，优先级是由 A 、B、C、D 的值来决定的，其中它们的值计算规则如下：

- 如果存在内联样式，那么 A = 1, 否则 A = 0
- B的值等于 ID选择器出现的次数
- C的值等于 类选择器 和 属性选择器 和 伪类 出现的总次数
- D 的值等于 标签选择器 和 伪元素 出现的总次数

这里举个例子：

```
#nav-global > ul > li > a.nav-link
```

套用上面的算法，依次求出`A` `B` `C` `D`的值：

- 因为没有内联样式，所以 A = 0
- ID选择器总共出现了1次， B = 1
- 类选择器出现了1次， 属性选择器出现了0次，伪类选择器出现0次，所以 C = (1 + 0 + 0) = 1
- 标签选择器出现了3次， 伪元素出现了0次，所以 D = (3 + 0) = 3

上面算出的`A` 、 `B`、`C`、`D`可以简记作：`(0, 1, 1, 3)`

知道了优先级是如何计算之后，就来看看比较规则：

- 从左往右依次进行比较 ，较大者优先级更高
- 如果相等，则继续往右移动一位进行比较
- 如果4位全部相等，则后面的会覆盖前面的

经过上面的优先级计算规则，我们知道内联样式的优先级最高，如果外部样式需要覆盖内联样式，就需要使用`!important`

三、继承属性

在`css`中，继承是指的是给父元素设置一些属性，后代元素会自动拥有这些属性

关于继承属性，可以分成：

- 字体系列属性

```
font:组合字体
font-family:规定元素的字体系列
font-weight:设置字体的粗细
font-size:设置字体的尺寸
font-style:定义字体的风格
font-variant:偏大或偏小的字体
```

- 文本系列属性

```
text-indent：文本缩进
text-align：文本水平对刘
line-height：行高
word-spacing：增加或减少单词间的空白
letter-spacing：增加或减少字符间的空白
text-transform：控制文本大小写
direction：规定文本的书写方向
color：文本颜色
```

- 元素可见性

```
visibility
```

- 表格布局属性

```
caption-side：定位表格标题位置
border-collapse：合并表格边框
border-spacing：设置相邻单元格的边框间的距离
empty-cells：单元格的边框的出现与消失
table-layout：表格的宽度由什么决定
```

- 列表属性

```
list-style-type：文字前面的小点点样式
list-style-position：小点点位置
list-style：以上的属性可通过这属性集合
```

- 引用

```
quotes：设置嵌套引用的引号类型
```

- 光标属性

```
cursor：箭头可以变成需要的形状
```

继承中比较特殊的几点：

- a 标签的字体颜色不能被继承
- h1-h6标签字体的大下也是不能被继承的

无继承的属性

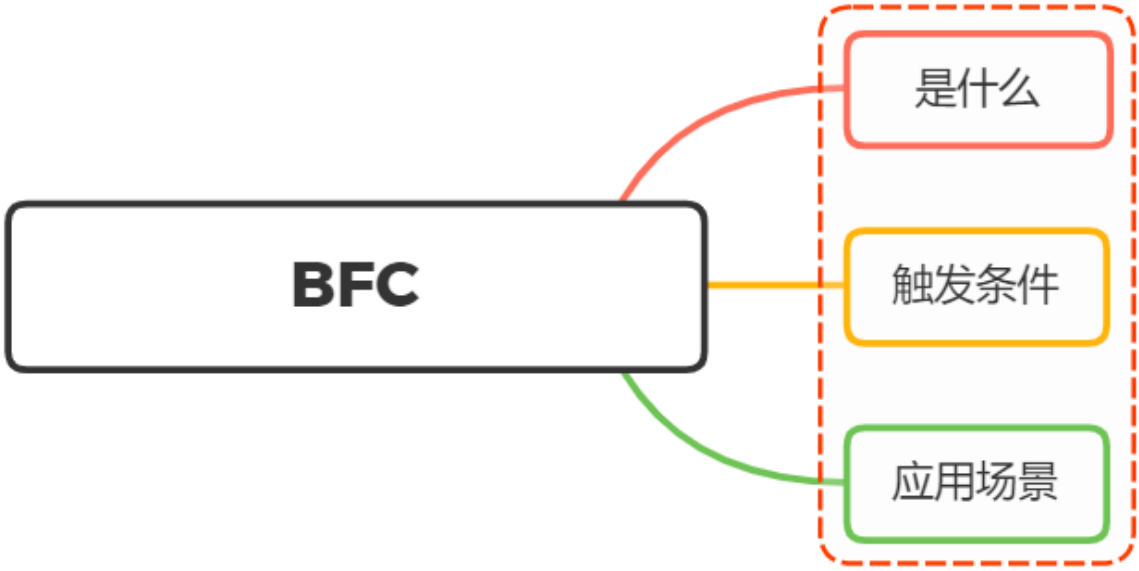
- display
- 文本属性：vertical-align、text-decoration
- 盒子模型的属性：宽度、高度、内外边距、边框等
- 背景属性：背景图片、颜色、位置等
- 定位属性：浮动、清除浮动、定位position等
- 生成内容属性：content、counter-reset、counter-increment
- 轮廓样式属性：outline-style、outline-width、outline-color、outline

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

参考文献

- https://www.html.cn/qa/css3/13444.html
- https://developer.mozilla.org/zh-CN/docs/Learn/CSS/Building_blocks/Selectors

03.谈谈你对BFC的理解



一、是什么

我们在页面布局的时候，经常出现以下情况：

- 这个元素高度怎么没了？
- 这两栏布局怎么没法自适应？
- 这两个元素的间距怎么有点奇怪的样子？
- ……

原因是元素之间相互的影响，导致了意料之外的情况，这里就涉及到`BFC`概念

`BFC`（Block Formatting Context），即块级格式化上下文，它是页面中的一块渲染区域，并且有一套属于自己的渲染规则：

- 内部的盒子会在垂直方向上一个接一个的放置
- 对于同一个BFC的俩个相邻的盒子的margin会发生重叠，与方向无关。
- 每个元素的左外边距与包含块的左边界相接触（从左到右），即使浮动元素也是如此
- BFC的区域不会与float的元素区域重叠
- 计算BFC的高度时，浮动子元素也参与计算
- BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素，反之亦然

`BFC`目的是形成一个相对于外界完全独立的空间，让内部的子元素不会影响到外部的元素

二、触发条件

触发`BFC`的条件包含不限于：

- 根元素，即HTML元素
- 浮动元素：float值为left、right

- overflow值不为 visible，为 auto、scroll、hidden
- display的值为inline-block、inltable-cell、table-caption、table、inline-table、flex、inline-flex、grid、inline-grid
- position的值为absolute或fixed

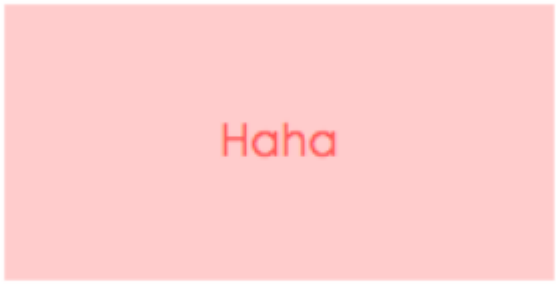
三、应用场景

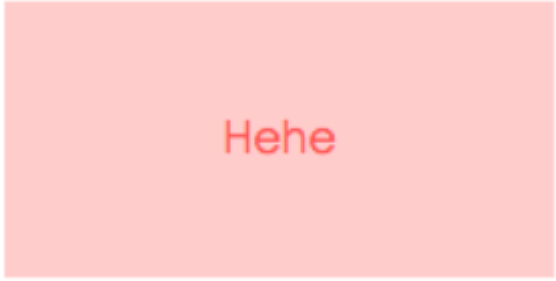
利用`BFC`的特性，我们将`BFC`应用在以下场景：

防止margin重叠（塌陷）

```
<style>
    p {
        color: #f55;
        background: #fcc;
        width: 200px;
        line-height: 100px;
        text-align:center;
        margin: 100px;
    }
</style>
<body>
    <p>Haha</p >
    <p>Hehe</p >
</body>
```

页面显示如下：





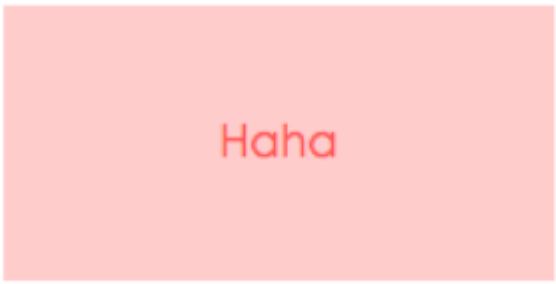
两个`p`元素之间的距离为`100px`，发生了`margin`重叠（塌陷），以最大的为准，如果第一个P的`margin`为80的话，两个P之间的距离还是100，以最大的为准。

前面讲到，同一个`BFC`的俩个相邻的盒子的`margin`会发生重叠

可以在`p`外面包裹一层容器，并触发这个容器生成一个`BFC`，那么两个`p`就不属于同一个`BFC`，则不会出现`margin`重叠

```
<style>
    .wrap {
        overflow: hidden;// 新的BFC
    }
    p {
        color: #f55;
        background: #fcc;
        width: 200px;
        line-height: 100px;
        text-align:center;
        margin: 100px;
    }
</style>
<body>
    <p>Haha</p >
    <div class="wrap">
        <p>Hehe</p >
    </div>
</body>
```

这时候，边距则不会重叠：



清除内部浮动

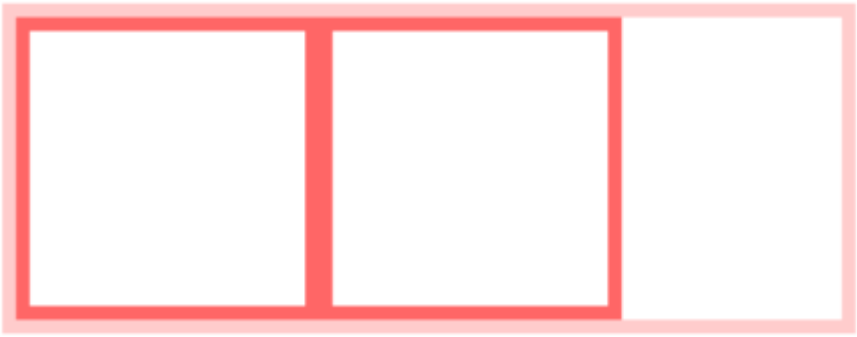
```
<style>
    .par {
        border: 5px solid #fcc;
        width: 300px;
    }

    .child {
        border: 5px solid #f66;
        width:100px;
        height: 100px;
        float: left;
    }
</style>
<body>
    <div class="par">
        <div class="child"></div>
        <div class="child"></div>
    </div>
</body>
```

页面显示如下：

而`BFC`在计算高度时，浮动元素也会参与，所以我们可以触发`.par`元素生活才能`BFC`，则内部浮动元素计算高度时候也会计算

```
.par {
    overflow: hidden;
}
```

实现效果如下：



自适应多栏布局

这里举个两栏的布局

```
<style>
    body {
        width: 300px;
        position: relative;
    }

    .aside {
        width: 100px;
        height: 150px;
        float: left;
        background: #f66;
    }

    .main {
        height: 200px;
        background: #fcc;
    }
</style>
<body>
    <div class="aside"></div>
    <div class="main"></div>
```

```
</body>
```

效果图如下：

前面讲到，每个元素的左外边距与包含块的左边界相接触

因此，虽然`.aslide`为浮动元素，但是`main`的左边依然会与包含块的左边相接触

而`BFC`的区域不会与浮动盒子重叠

所以我们可以通过触发`main`生成`BFC`，以此适应两栏布局

```
.main {
    overflow: hidden;
}
```

这时候，新的`BFC`不会与浮动的`.aside`元素重叠。因此会根据包含块的宽度，和`.aside`的宽度，自动变窄

效果如下：

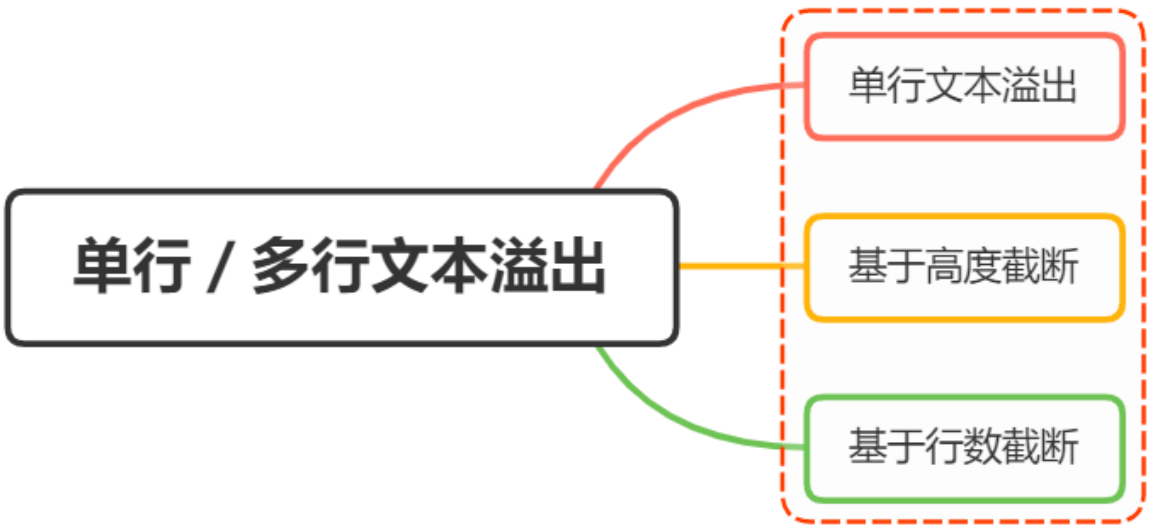
小结

可以看到上面几个案例，都体现了`BFC`实际就是页面一个独立的容器，里面的子元素不影响外面的元素

参考文献

- https://developer.mozilla.org/zh-CN/docs/Web/Guide/CSS/Block_formatting_context
- https://github.com/zuopf769/notebook/blob/master/fe/BFC%E5%8E%9F%E7%90%86%E5%89%96%E6%9E%90/README.md

04.如何实现单行 / 多行文本溢出的省略样式



一、前言

在日常开发展示页面，如果一段文本的数量过长，受制于元素宽度的因素，有可能不能完全显示，为了提高用户的使用体验，这个时候就需要我们把溢出的文本显示成省略号

对于文本的溢出，我们可以分成两种形式：

- 单行文本溢出
- 多行文本溢出

二、实现方式

单行文本溢出省略

理解也很简单，即文本在一行内显示，超出部分以省略号的形式展现

实现方式也很简单，涉及的`css`属性有：

- text-overflow：规定当文本溢出时，显示省略符号来代表被修剪的文本
- white-space：设置文字在一行显示，不能换行
- overflow：文字长度超出限定宽度，则隐藏超出的内容

`overflow`设为`hidden`，普通情况用在块级元素的外层隐藏内部溢出元素，或者配合下面两个属性实现文本溢出省略

`white-space:nowrap`，作用是设置文本不换行，是`overflow:hidden`和`text-overflow: ellipsis`生效的基础

`text-overflow`属性值有如下：

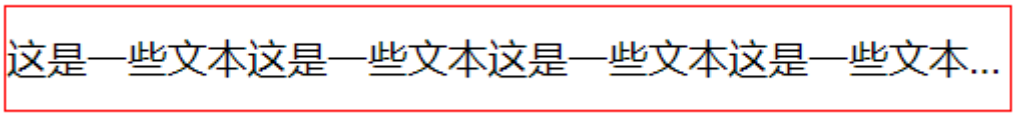
- clip：当对象内文本溢出部分裁切掉
- ellipsis：当对象内文本溢出时显示省略标记（...）

`text-overflow`只有在设置了`overflow:hidden`和`white-space:nowrap`才能够生效的

举个例子

```
<style>
    p{
        overflow: hidden;
        line-height: 40px;
        width:400px;
        height:40px;
        border:1px solid red;
        text-overflow: ellipsis;
        white-space: nowrap;
    }
</style>
<p 这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本</p >
```

效果如下：



可以看到，设置单行文本溢出较为简单，并且省略号显示的位置较好

多行文本溢出省略

多行文本溢出的时候，我们可以分为两种情况：

- 基于高度截断
- 基于行数截断

基于高度截断

伪元素 + 定位

核心的`css`代码结构如下：

- position: relative：为伪元素绝对定位
- overflow: hidden：文本溢出限定的宽度就隐藏内容)
- position: absolute：给省略号绝对定位
- line-height: 20px：结合元素高度,高度固定的情况下,设定行高, 控制显示行数
- height: 40px：设定当前元素高度
- ::after {}：设置省略号样式

代码如下所示：

```
<style>
    .demo {
```

```
        position: relative;
        line-height: 20px;
        height: 40px;
        overflow: hidden;
    }
    .demo::after {
        content: "...";
        position: absolute;
        bottom: 0;
        right: 0;
        padding: 0 20px 0 10px;
    }
</style>

<body>
    <div class='demo'>这是一段很长的文本</div>
</body>
```

实现原理很好理解，就是通过伪元素绝对定位到行尾并遮住文字，再通过 `overflow: hidden` 隐藏多余文字

这种实现具有以下优点：

- 兼容性好，对各大主流浏览器有好的支持
- 响应式截断，根据不同宽度做出调整

一般文本存在英文的时候，可以设置 `word-break: break-all` 使一个单词能够在换行时进行拆分

基于行数截断

纯 `css` 实现也非常简单，核心的 `css` 代码如下：

- -webkit-line-clamp: 2：用来限制在一个块元素显示的文本的行数，为了实现该效果，它需要组合其他的WebKit属性）
- display: -webkit-box：和1结合使用，将对象作为弹性伸缩盒子模型显示
- -webkit-box-orient: vertical：和1结合使用 ，设置或检索伸缩盒对象的子元素的排列方式
- overflow: hidden：文本溢出限定的宽度就隐藏内容
- text-overflow: ellipsis：多行文本的情况下，用省略号“...”隐藏溢出范围的文本

```
<style>
    p {
        width: 400px;
        border-radius: 1px solid red;
        -webkit-line-clamp: 2;
        display: -webkit-box;
        -webkit-box-orient: vertical;
        overflow: hidden;
        text-overflow: ellipsis;
    }
</style>
<p>
    这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本
    这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本
</p >
```

可以看到，上述使用了`webkit`的`CSS`属性扩展，所以兼容浏览器范围是`PC`端的`webkit`内核的浏览器，由于移动端大多数是使用`webkit`，所以移动端常用该形式

需要注意的是，如果文本为一段很长的英文或者数字，则需要添加`word-wrap: break-word`属性

还能通过使用`javascript`实现配合`css`，实现代码如下所示：

css结构如下：

```
p {
    position: relative;
    width: 400px;
    line-height: 20px;
    overflow: hidden;

}
.p-after:after{
    content: "...";
    position: absolute;
    bottom: 0;
    right: 0;
    padding-left: 40px;
    background: -webkit-linear-gradient(left, transparent, #fff 55%);
    background: -moz-linear-gradient(left, transparent, #fff 55%);
    background: -o-linear-gradient(left, transparent, #fff 55%);
    background: linear-gradient(to right, transparent, #fff 55%);
}
```

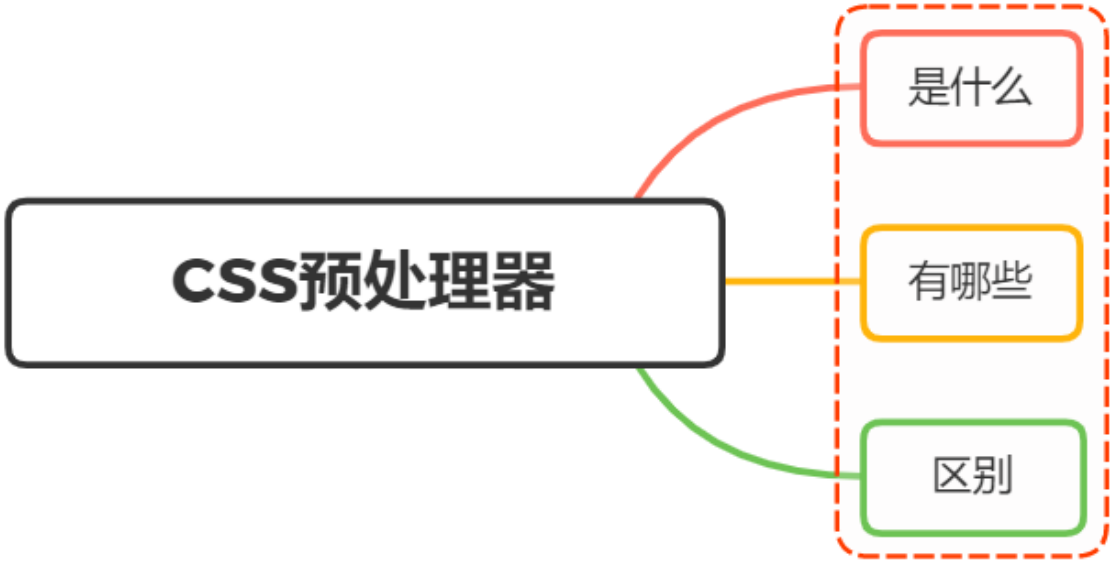
javascript代码如下：

```
$(function(){
 //获取文本的行高，并获取文本的高度，假设我们规定的行数是五行，那么对超过行数的部分进行限制高度，并加上省略号
   $('p').each(function(i, obj){
        var lineHeight = parseInt($(this).css("line-height"));
        var height = parseInt($(this).height());
        if((height / lineHeight) >3 ){
            $(this).addClass("p-after")
            $(this).css("height","60px");
        }else{
            $(this).removeClass("p-after");
        }
    });
})
```

参考文献

- https://www.zoo.team/article/text-overflow
- https://segmentfault.com/a/1190000017078153

05.说说对Css预编语言的理解？有哪些区别?



一、是什么

`Css` 作为一门标记性语言，语法相对简单，对使用者的要求较低，但同时也带来一些问题

需要书写大量看似没有逻辑的代码，不方便维护及扩展，不利于复用，尤其对于非前端开发工程师来讲，往往会因为缺少 `Css` 编写经验而很难写出组织良好且易于维护的 `Css` 代码

`Css` 预处理器便是针对上述问题的解决方案

预处理语言

扩充了 `Css` 语言，增加了诸如变量、混合（mixin）、函数等功能，让 `Css` 更易维护、方便

本质上，预处理是 `Css` 的超集

包含一套自定义的语法及一个解析器，根据这些语法定义自己的样式规则，这些规则最终会通过解析器，编译生成对应的 `Css` 文件

二、有哪些

`Css` 预编译语言在前端里面有三大优秀的预编处理器，分别是：

- sass
- less
- stylus

sass

2007 年诞生，最早也是最成熟的 `Css` 预处理器，拥有 Ruby 社区的支持和 `Compass` 这一最强大的 `Css` 框架，目前受 `LESS` 影响，已经进化到了全面兼容 `Css` 的 `Scss`

文件后缀名为 `.sass` 与 `scss`，可以严格按照 sass 的缩进方式省去大括号和分号

less

2009年出现，受 `SASS` 的影响较大，但又使用 `Css` 的语法，让大部分开发者和设计师更容易上手，在 `Ruby` 社区之外支持者远超过 `SASS`

其缺点是比起 `SASS` 来，可编程功能不够，不过优点是简单和兼容 `Css`，反过来也影响了 `SASS` 演变到了 `Scss` 的时代

stylus

`Stylus` 是一个 `Css` 的预处理框架，2010 年产生，来自 `Node.js` 社区，主要用来给 `Node` 项目进行 `Css` 预处理支持

所以 `Stylus` 是一种新型语言，可以创建健壮的、动态的、富有表现力的 `Css`。比较年轻，其本质上做的事情与 `SASS/LESS` 等类似

三、区别

虽然各种预处理器功能强大，但使用最多的，还是以下特性：

- 变量（variables）
- 作用域（scope）
- 代码混合（ mixins）
- 嵌套（nested rules）
- 代码模块化（Modules）

因此，下面就展开这些方面的区别

基本使用

less和scss

```
.box {
  display: block;
}
```

sass

```
.box
  display: block
```

stylus

```
.box
  display: block
```

嵌套

三者的嵌套语法都是一致的，甚至连引用父级选择器的标记 & 也相同

区别只是 Sass 和 Stylus 可以用没有大括号的方式书写

less

```
.a {
  &.b {
    color: red;
  }
}
```

变量

变量无疑为 Css 增加了一种有效的复用方式，减少了原来在 Css 中无法避免的重复「硬编码」

`less` 声明的变量必须以 `@` 开头，后面紧跟变量名和变量值，而且变量名和变量值需要使用冒号 `:` 分隔开

```
@red: #c00;

strong {
  color: @red;
}
```

`sass` 声明的变量跟 `less` 十分的相似，只是变量名前面使用 `@` 开头

```
$red: #c00;

strong {
  color: $red;
}
```

`stylus` 声明的变量没有任何的限定，可以使用 `$` 开头，结尾的分号 `;` 可有可无，但变量与变量值之间需要使用 `=`

在 `stylus` 中我们不建议使用 `@` 符号开头声明变量

```
red = #c00

strong
  color: red
```

作用域

`Css` 预编译器把变量赋予作用域，也就是存在生命周期。就像 `js` 一样，它会先从局部作用域查找变量，依次向上级作用域查找

`sass` 中不存在全局变量

```
$color: black;
.scoped {
  $bg: blue;
  $color: white;
  color: $color;
  background-color:$bg;
}
.unscoped {
  color:$color;
}
```

编译后

```
.scoped {
  color:white;/*是白色*/
  background-color:blue;
}
.unscoped {
  color:white;/*白色（无全局变量概念）*/
}
```

所以，在`sass`中最好不要定义相同的变量名

`less`与`stylus`的作用域跟`javascript`十分的相似，首先会查找局部定义的变量，如果没有找到，会像冒泡一样，一级一级往下查找，直到根为止

```
@color: black;
.scoped {
  @bg: blue;
  @color: white;
  color: @color;
  background-color:@bg;
}
.unscoped {
  color:@color;
}
```

编译后:

```
.scoped {
  color:white;/*白色（调用了局部变量）*/
  background-color:blue;
}
.unscoped {
  color:black;/*黑色（调用了全局变量）*/
}
```

混入

混入（mixin）应该说是预处理器最精髓的功能之一了，简单点来说，`Mixins`可以将一部分样式抽出，作为单独定义的模块，被很多选择器重复使用

可以在`Mixins`中定义变量或者默认参数

在`less`中，混合的用法是指将定义好的`ClassA`中引入另一个已经定义的`Class`，也能使用够传递参数，参数变量为`@`声明

```
.alert {
  font-weight: 700;
}

.highlight(@color: red) {
  font-size: 1.2em;
  color: @color;
}

.heads-up {
  .alert;
  .highlight(red);
}
```

编译后

```
.alert {
  font-weight: 700;
}
.heads-up {
  font-weight: 700;
  font-size: 1.2em;
  color: red;
}
```

`Sass`声明`mixins`时需要使用`@mixinn`，后面紧跟`mixin`的名，也可以设置参数，参数名为变量`$`声明的形式

```
@mixin large-text {
  font: {
    family: Arial;
    size: 20px;
    weight: bold;
  }
  color: #ff0000;
}

.page-title {
  @include large-text;
  padding: 4px;
  margin-top: 10px;
}
```

`stylus`中的混合和前两款`Css`预处理器语言的混合略有不同，他可以不使用任何符号，就是直接声明`Mixins`名，然后在定义参数和默认值之间用等号（=）来连接

```
error(borderWidth= 2px) {
```

```
  border: borderWidth solid #F00;
  color: #F00;
}
.generic-error {
  padding: 20px;
  margin: 4px;
  error(); /* 调用error mixins */
}
.login-error {
  left: 12px;
  position: absolute;
  top: 20px;
  error(5px); /* 调用error mixins，并将参数$borderWidth的值指定为5px */
}
```

代码模块化

模块化就是将`Css`代码分成一个个模块

`scss`、`less`、`stylus`三者的使用方法都如下所示

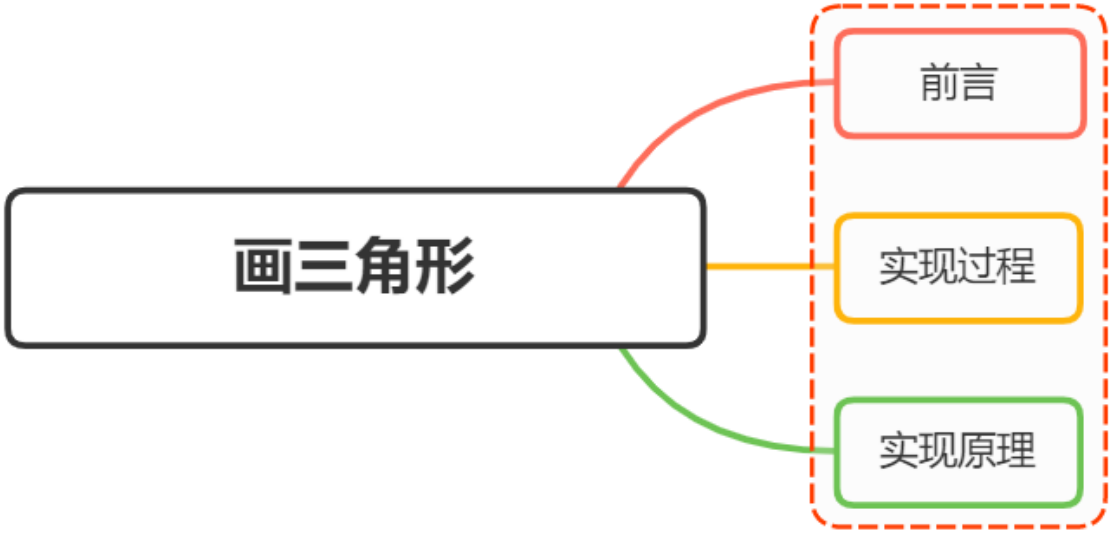
```
@import './common';
@import './github-markdown';
@import './mixin';
@import './variables';
```

参考文献

- https://jelly.jd.com/article/5dcb9c73641a030153732a89
- https://zhuanlan.zhihu.com/p/23382462
- https://baike.baidu.com/item/Less/17570158

2.移动web

01.CSS如何画一个三角形？原理是什么？



一、前言

在前端开发的时候，我们有时候会需要用到一个三角形的形状，比如地址选择或者播放器里面播放按钮

通常情况下，我们会使用图片或者`svg`去完成三角形效果图，但如果单纯使用`css`如何完成一个三角形呢?

实现过程似乎也并不困难，通过边框就可完成

二、实现过程

在以前也讲过盒子模型，默认情况下是一个矩形，实现也很简单

```
<style>
    .border {
        width: 50px;
        height: 50px;
        border: 2px solid;
        border-color: #96ceb4 #ffeead #d9534f #ffad60;
    }
</style>
<div class="border"></div>
```

效果如下图所示:

将`border`设置`50px`，效果图如下所示：

白色区域则为`width`、`height`，这时候只需要你将白色区域部分宽高逐渐变小，最终变为0，则变成如下图所示：

这时候就已经能够看到4个不同颜色的三角形，如果需要下方三角形，只需要将上、左、右边框设置为0就可以得到下方的红色三角形

但这种方式，虽然视觉上是实现了三角形，但实际上，隐藏的部分任然占据部分高度，需要将上方的宽度去掉

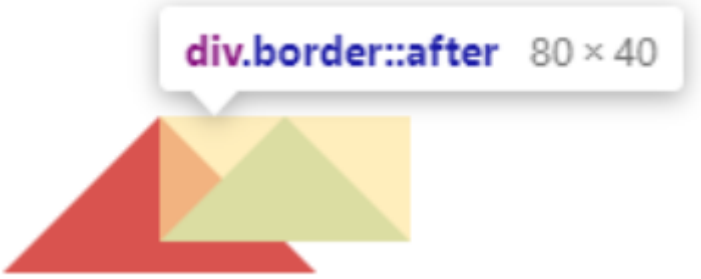
最终实现代码如下：

```
.border {
    width: 0;
    height: 0;
    border-style:solid;
    border-width: 0 50px 50px;
    border-color: transparent transparent #d9534f;
}
```

如果想要实现一个只有边框是空心的三角形，由于这里不能再使用`border`属性，所以最直接的方法是利用伪类新建一个小一点的三角形定位上去

```
.border {
    width: 0;
    height: 0;
    border-style:solid;
    border-width: 0 50px 50px;
    border-color: transparent transparent #d9534f;
    position: relative;
}
.border:after{
    content: '';
    border-style:solid;
    border-width: 0 40px 40px;
    border-color: transparent transparent #96ceb4;
    position: absolute;
    top: 0;
    left: 0;
}
```

效果图如下所示：



伪类元素定位参照对象的内容区域宽高都为0，则内容区域即可以理解成中心一点，所以伪元素相对中心这点定位

将元素定位进行微调以及改变颜色，就能够完成下方效果图：

最终代码如下：

```
.border:after {
    content: '';
    border-style: solid;
    border-width: 0 40px 40px;
    border-color: transparent transparent #96ceb4;
    position: absolute;
    top: 6px;
    left: -40px;
}
```

三、原理分析

可以看到，边框是实现三角形的部分，边框实际上并不是一个直线，如果我们将四条边设置不同的颜色，将边框逐渐放大，可以得到每条边框都是一个梯形



当分别取消边框的时候，发现下面几种情况：

- 取消一条边的时候，与这条边相邻的两条边的接触部分会变成直的
- 当仅有邻边时， 两个边会变成对分的三角
- 当保留边没有其他接触时，极限情况所有东西都会消失

通过上图的变化规则，利用旋转、隐藏，以及设置内容宽高等属性，就能够实现其他类型的三角形

如设置直角三角形，如上图倒数第三行实现过程，我们就能知道整个实现原理

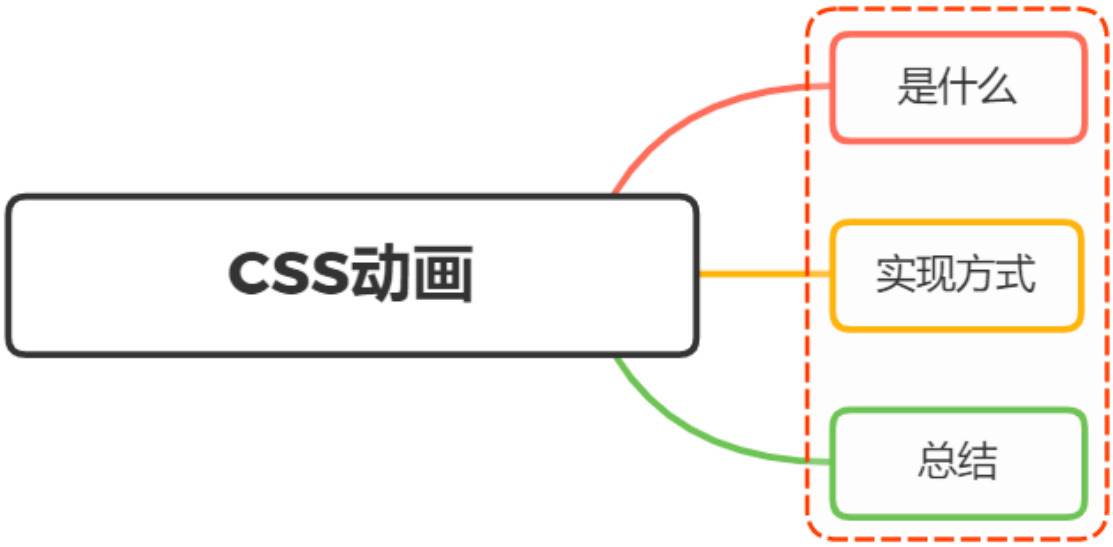
实现代码如下：

```
.box {
    /* 内部大小 */
    width: 0px;
    height: 0px;
    /* 边框大小 只设置两条边*/
    border-top: #4285f4 solid;
    border-right: transparent solid;
    border-width: 85px;
    /* 其他设置 */
    margin: 50px;
}
```

参考文献

- https://www.cnblogs.com/echolun/p/11888612.html
- https://juejin.cn/post/6844903567795421197
- https://vue3js.cn/interview

02.css3动画有哪些



一、是什么

CSS动画（CSS Animations）是为层叠样式表建议的允许可扩展标记语言（XML）元素使用CSS的动画的模块

即指元素从一种样式逐渐过渡为另一种样式的过程

常见的动画效果有很多，如平移、旋转、缩放等等，复杂动画则是多个简单动画的组合

`css`实现动画的方式，有如下几种：

- transition 实现渐变动画
- transform 转变动画
- animation 实现自定义动画

二、实现方式

transition 实现渐变动画

`transition`的属性如下：

- property:填写需要变化的css属性
- duration:完成过渡效果需要的时间单位(s或者ms)
- timing-function:完成效果的速度曲线
- delay: 动画效果的延迟触发时间

其中`timing-function`的值有如下：

值	描述
linear	匀速（等于 cubic-bezier(0,0,1,1)）
ease	从慢到快再到慢（cubic-bezier(0.25,0.1,0.25,1)）
ease-in	慢慢变快（等于 cubic-bezier(0.42,0,1,1)）
ease-out	慢慢变慢（等于 cubic-bezier(0,0,0.58,1)）
ease-in-out	先变快再到慢（等于 cubic-bezier(0.42,0,0.58,1)），渐显渐隐效果
cubic-bezier(*n*,*n*,*n*,*n*)	在 cubic-bezier 函数中定义自己的值。可能的值是 0 至 1 之间的数值

注意：并不是所有的属性都能使用过渡的，如`display:none<->display:block`

举个例子，实现鼠标移动上去发生变化动画效果

```
<style>
        .base {
            width: 100px;
            height: 100px;
            display: inline-block;
            background-color: #0EA9FF;
            border-width: 5px;
            border-style: solid;
            border-color: #5daf34;
            transition-property: width, height, background-color, border-width;
            transition-duration: 2s;
            transition-timing-function: ease-in;
            transition-delay: 500ms;
        }

        /*简写*/
        /*transition: all 2s ease-in 500ms;*/
        .base:hover {
            width: 200px;
            height: 200px;
            background-color: #5daf34;
            border-width: 10px;
            border-color: #3a8ee6;
        }
</style>
<div class="base"></div>
```

transform 转变动画

包含四个常用的功能：

- translate：位移
- scale：缩放
- rotate：旋转
- skew：倾斜

一般配合`transition`过度使用

注意的是，`transform`不支持`inline`元素，使用前把它变成`block`

举个例子

```
<style>
    .base {
        width: 100px;
        height: 100px;
        display: inline-block;
        background-color: #0EA9FF;
        border-width: 5px;
        border-style: solid;
        border-color: #5daf34;
        transition-property: width, height, background-color, border-width;
        transition-duration: 2s;
        transition-timing-function: ease-in;
        transition-delay: 500ms;
    }
    .base2 {
        transform: none;
        transition-property: transform;
        transition-delay: 5ms;
    }

    .base2:hover {
        transform: scale(0.8, 1.5) rotate(35deg) skew(5deg) translate(15px,
25px);
    }
</style>
 <div class="base base2"></div>
```

可以看到盒子发生了旋转，倾斜，平移，放大

animation 实现自定义动画

`animation`是由 8 个属性的简写，分别如下：

属性	描述	属性值
animation-duration	指定动画完成一个周期所需要时间，单位秒（s）或毫秒（ms），默认是 0	
animation-timing-function	指定动画计时函数，即动画的速度曲线，默认是 "ease"	linear、ease、ease-in、ease-out、ease-in-out
animation-delay	指定动画延迟时间，即动画何时开始，默认是 0	
animation-iteration-count	指定动画播放的次数，默认是 1	
animation-direction 指定动画播放的方向	默认是 normal	normal、reverse、alternate、alternate-reverse
animation-fill-mode	指定动画填充模式。默认是 none	forwards、backwards、both
animation-play-state	指定动画播放状态，正在运行或暂停。默认是 running	running、pauser
animation-name	指定 @keyframes 动画的名称	

`css` 动画只需要定义一些关键的帧，而其余的帧，浏览器会根据计时函数插值计算出来，

通过 `@keyframes` 来定义关键帧

因此，如果我们想要让元素旋转一圈，只需要定义开始和结束两帧即可：

```
@keyframes rotate{
    from{
        transform: rotate(0deg);
    }
    to{
        transform: rotate(360deg);
    }
}
```

`from` 表示最开始的那一帧，`to` 表示结束时的那一帧

也可以使用百分比刻画生命周期

```
@keyframes rotate{
    0%{
        transform: rotate(0deg);
    }
    50%{
        transform: rotate(180deg);
    }
    100%{
        transform: rotate(360deg);
    }
}
```

定义好了关键帧后，下来就可以直接用它了：

```
animation: rotate 2s;
```

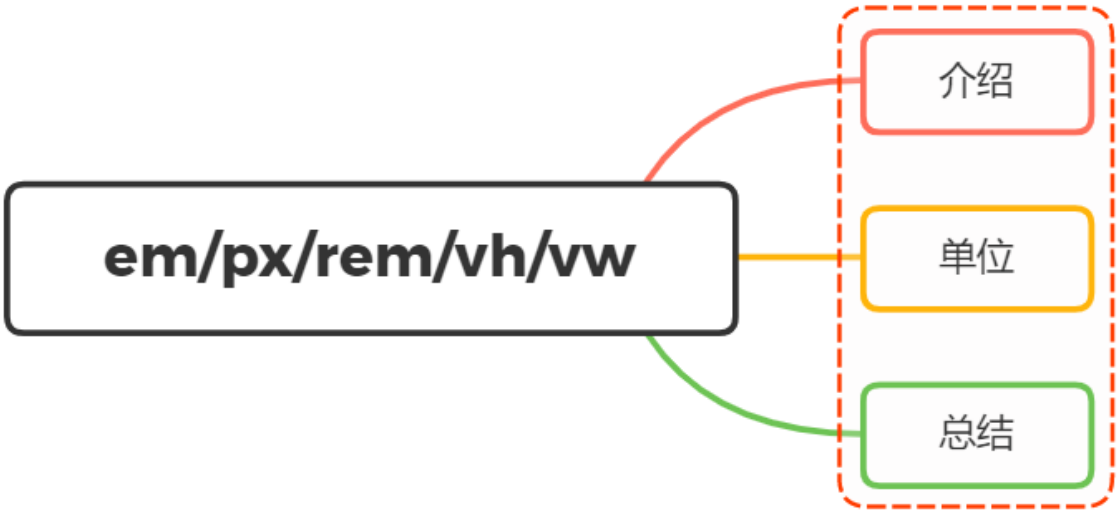
三、总结

属性	含义
transition（过度）	用于设置元素的样式过度，和animation有着类似的效果，但细节上有很大的不同
transform（变形）	用于元素进行旋转、缩放、移动或倾斜，和设置样式的动画并没有什么关系，就相当于color一样用来设置元素的“外表”
translate（移动）	只是transform的一个属性值，即移动
animation（动画）	用于设置动画属性，他是一个简写的属性，包含6个属性

参考文献

- https://segmentfault.com/a/1190000022540857
- https://zh.m.wikipedia.org/wiki/CSS%E5%8A%A8%E7%94%BB
- https://vue3js.cn/interview

03.说说em/px/rem/vh/vw区别



一、介绍

传统的项目开发中，我们只会用到`px`、`%`、`em`这几个单位，它可以适用于大部分的项目开发，且拥有比较良好的兼容性

从`CSS3`开始，浏览器对计量单位的支持又提升到了另外一个境界，新增了`rem`、`vh`、`vw`、`vm`等一些新的计量单位

利用这些新的单位开发出比较良好的响应式页面，适应多种不同分辨率的终端，包括移动设备等

二、单位

在`css`单位中，可以分为长度单位、绝对单位，如下表所指示

CSS单位	
相对长度单位	em、ex、ch、rem、vw、vh、vmin、vmax、%
绝对长度单位	cm、mm、in、px、pt、pc

这里我们主要讲述px、em、rem、vh、vw

px

px，表示像素，所谓像素就是呈现在我们显示器上的一个个小点，每个像素点都是大小等同的，所以像素为计量单位被分在了绝对长度单位中

有些人会把`px`认为是相对长度，原因在于在移动端中存在设备像素比，`px`实际显示的大小是不确定的

这里之所以认为`px`为绝对单位，在于`px`的大小和元素的其他属性无关

em

em是相对长度单位。相对于当前对象内文本的字体尺寸。如当前对行内文本的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸（`1em = 16px`）

为了简化`font-size`的换算，我们需要在`css`中的`body`选择器中声明`font-size`=`62.5%`，这就使em 值变为`16px*62.5% = 10px`

这样`12px = 1.2em`，`10px = 1em`，也就是说只需要将你的原来的`px`数值除以 10，然后换上`em`作为单位就行了

特点：

- em 的值并不是固定的
- em 会继承父级元素的字体大小
- em 是相对长度单位。相对于当前对象内文本的字体尺寸。如当前对行内文本的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸
- 任意浏览器的默认字体高都是 16px

举个例子

```
<div class="big">
    我是14px=1.4rem<div class="small">我是12px=1.2rem</div>
</div>
```

样式为

```
<style>
    html {font-size: 10px;  } /*  公式16px*62.5%=10px  */
    .big{font-size: 1.4rem}
    .small{font-size: 1.2rem}
</style>
```

这时候`.big`元素的`font-size`为14px，而`.small`元素的`font-size`为12px

rem

rem，相对单位，相对的只是HTML根元素`font-size`的值

同理，如果想要简化`font-size`的转化，我们可以在根元素`html`中加入`font-size: 62.5%`

```
html {font-size: 62.5%;  } /*  公式16px*62.5%=10px  */
```

这样页面中1rem=10px、1.2rem=12px、1.4rem=14px、1.6rem=16px;使得视觉、使用、书写都得到了极大的帮助

特点：

- rem单位可谓集相对大小和绝对大小的优点于一身
- 和em不同的是rem总是相对于根元素，而不像em一样使用级联的方式来计算尺寸

vh、vw

vw，就是根据窗口的宽度，分成100等份，100vw就表示满宽，50vw就表示一半宽。（vw 始终是针对窗口的宽），同理，`vh`则为窗口的高度

这里的窗口分成几种情况：

- 在桌面端，指的是浏览器的可视区域
- 移动端指的就是布局视口

像`vw`、`vh`，比较容易混淆的一个单位是`%`，不过百分比宽泛的讲是相对于父元素：

- 对于普通定位元素就是我们理解的父元素
- 对于position: absolute;的元素是相对于已定位的父元素
- 对于position: fixed;的元素是相对于 ViewPort（可视窗口）

三、总结

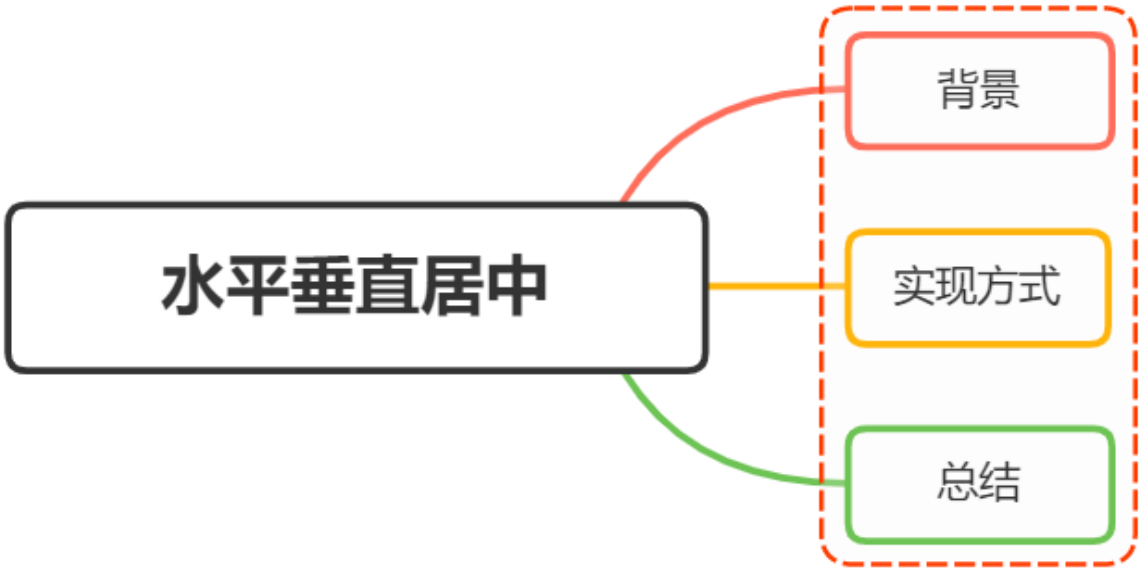
px：绝对单位，页面按精确像素展示

em：相对单位，基准点为父节点字体的大小，如果自身定义了`font-size`按自身来计算，整个页面内`1em`不是一个固定的值

rem：相对单位，可理解为`root em`，相对根节点`html`的字体大小来计算

vh、vw：主要用于页面视口大小布局，在页面布局上更加方便简单

04.元素水平垂直居中的方法有哪些？如果元素不定宽高呢?



一、背景

在开发中经常遇到这个问题，即让某个元素的内容在水平和垂直方向上都居中，内容不仅限于文字，可能是图片或其他元素

居中是一个非常基础但又是非常重要的应用场景，实现居中的方法存在很多，可以将这些方法分成两个大类：

- 居中元素（子元素）的宽高已知
- 居中元素宽高未知

二、实现方式

实现元素水平垂直居中的方式：

- 利用定位+margin:auto
- 利用定位+margin:负值
- 利用定位+transform
- table布局
- flex布局
- grid布局

利用定位+margin:auto

先上代码：

```
<style>
    .father{
        width:500px;
        height:300px;
        border:1px solid #0a3b98;
```

```
        position: relative;
    }
    .son{
        width:100px;
        height:40px;
        background: #f0a238;
        position: absolute;
        top:0;
        left:0;
        right:0;
        bottom:0;
        margin:auto;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

父级设置为相对定位，子级绝对定位 ，并且四个定位属性的值都设置了0，那么这时候如果子级没有设置宽高，则会被拉开到和父级一样宽高

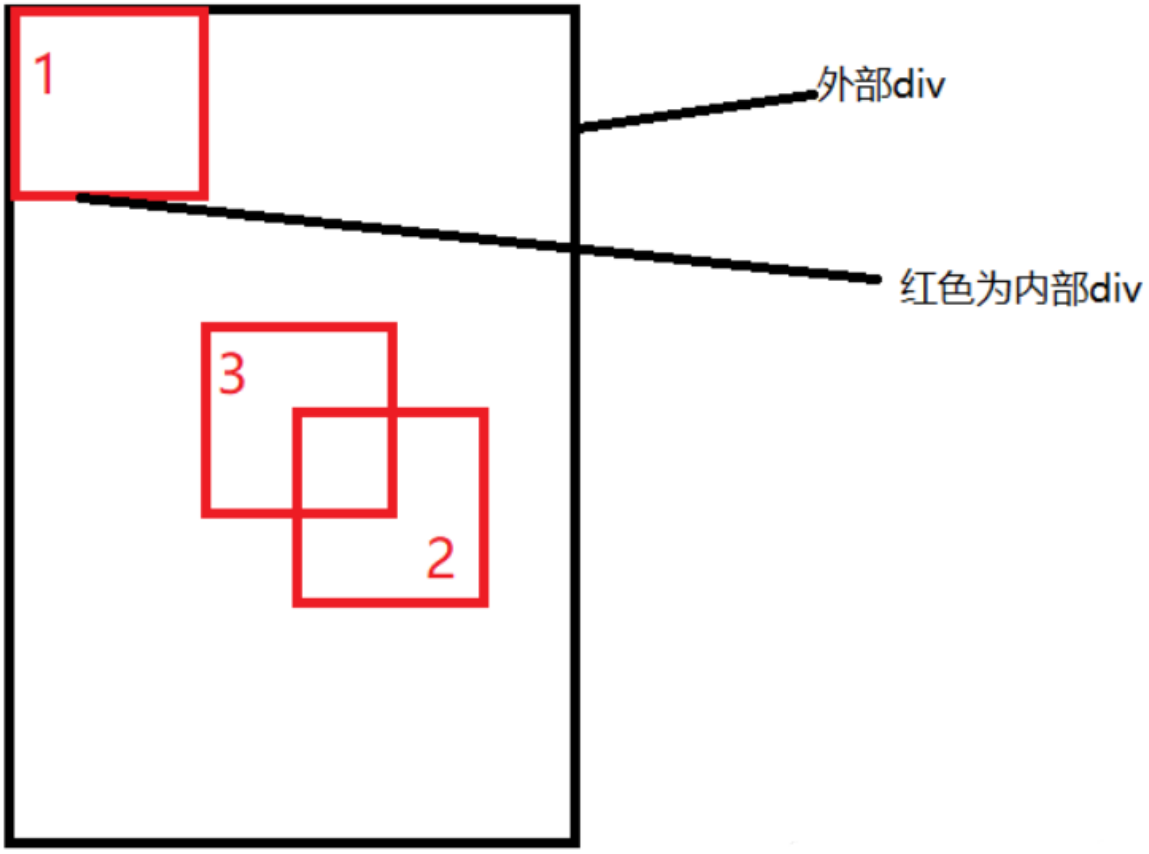
这里子元素设置了宽高，所以宽高会按照我们的设置来显示，但是实际上子级的虚拟占位已经撑满了整个父级，这时候再给它一个`margin: auto`它就可以上下左右都居中了

利用定位+margin:负值

绝大多数情况下，设置父元素为相对定位， 子元素移动自身50%实现水平垂直居中

```
<style>
    .father {
        position: relative;
        width: 200px;
        height: 200px;
        background: skyblue;
    }
    .son {
        position: absolute;
        top: 50%;
        left: 50%;
        margin-left:-50px;
        margin-top:-50px;
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

整个实现思路如下图所示：



- 初始位置为方块1的位置
- 当设置left、top为50%的时候，内部子元素为方块2的位置
- 设置margin为负数时，使内部子元素到方块3的位置，即中间位置

这种方案不要求父元素的高度，也就是即使父元素的高度变化了，仍然可以保持在父元素的垂直居中位置，水平方向上是一样的操作

但是该方案需要知道子元素自身的宽高，但是我们可以通过下面`transform`属性进行移动

利用定位+transform

实现代码如下：

```
<style>
    .father {
        position: relative;
        width: 200px;
        height: 200px;
        background: skyblue;
    }
    .son {
        position: absolute;
        top: 50%;
        left: 50%;
  transform: translate(-50%,-50%);
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
```

```
<div class="father">
    <div class="son"></div>
</div>
```

`translate(-50%, -50%)`将会将元素位移自己宽度和高度的-50%

这种方法其实和最上面被否定掉的margin负值用法一样，可以说是`margin`负值的替代方案，并不需要知道自身元素的宽高

table布局

设置父元素为`display:table-cell`，子元素设置`display: inline-block`。利用`vertical`和`text-align`可以让所有的行内块级元素水平垂直居中

```
<style>
    .father {
        display: table-cell;
        width: 200px;
        height: 200px;
        background: skyblue;
        vertical-align: middle;
        text-align: center;
    }
    .son {
        display: inline-block;
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

flex弹性布局

还是看看实现的整体代码：

```
<style>
    .father {
        display: flex;
        justify-content: center;
        align-items: center;
        width: 200px;
        height: 200px;
        background: skyblue;
    }
    .son {
        width: 100px;
        height: 100px;
        background: red;
```

```
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

`css3`中了`flex`布局，可以非常简单实现垂直水平居中

这里可以简单看看`flex`布局的关键属性作用：

- display: flex时，表示该容器内部的元素将按照flex进行布局
- align-items: center表示这些元素将相对于本容器水平居中
- justify-content: center也是同样的道理垂直居中

grid网格布局

```
<style>
    .father {
            display: grid;
            align-items:center;
            justify-content: center;
            width: 200px;
            height: 200px;
            background: skyblue;

        }
        .son {
            width: 10px;
            height: 10px;
            border: 1px solid red
        }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

这里看到，`gird`网格布局和`flex`弹性布局都简单粗暴

小结

上述方法中，不知道元素宽高大小仍能实现水平垂直居中的方法有：

- 利用定位+margin:auto
- 利用定位+transform
- 利用定位+margin:负值
- flex布局
- grid布局

三、总结

根据元素标签的性质，可以分为：

- 内联元素居中布局
- 块级元素居中布局

内联元素居中布局

水平居中

- 行内元素可设置：text-align: center
- flex布局设置父元素：display: flex; justify-content: center

垂直居中

- 单行文本父元素确认高度：height === line-height
- 多行文本父元素确认高度：disaply: table-cell; vertical-align: middle

块级元素居中布局

水平居中

- 定宽: margin: 0 auto
- 绝对定位+left:50%+margin:负自身一半

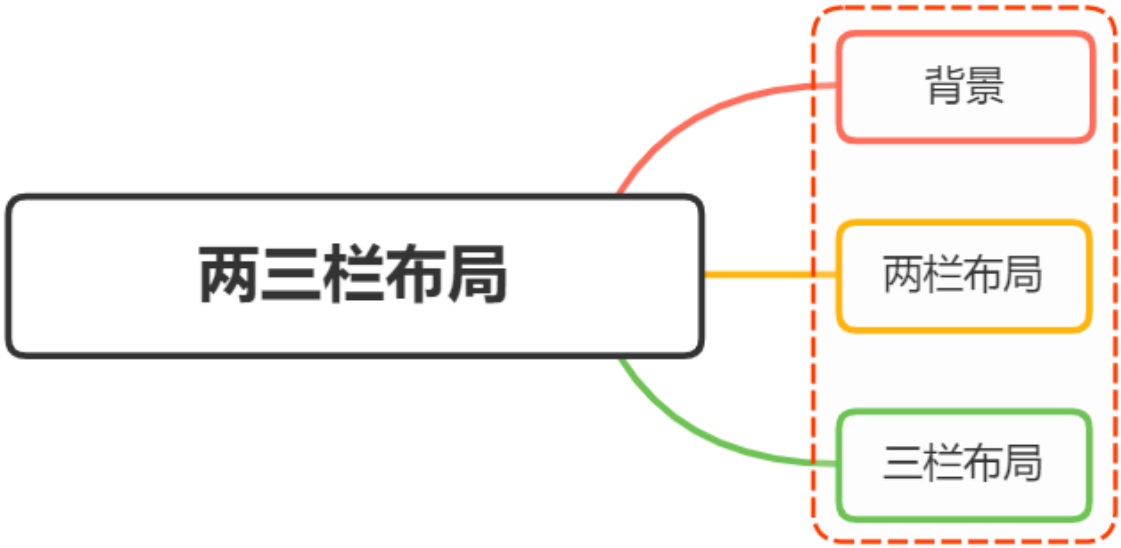
垂直居中

- position: absolute设置left、top、margin-left、margin-top(定高)
- display: table-cell
- transform: translate(x, y)
- flex(不定高，不定宽)
- grid(不定高，不定宽)，兼容性相对比较差

参考文献

- https://juejin.cn/post/6844903982960214029#heading-10

05.如何实现两栏布局，右侧自适应？三栏布局中间自适应呢？



一、背景

在日常布局中，无论是两栏布局还是三栏布局，使用的频率都非常高

两栏布局

两栏布局实现效果就是将页面分割成左右宽度不等的两列，宽度较小的列设置为固定宽度，剩余宽度由另一列撑满，

比如 `Ant Design` 文档，蓝色区域为主要内容布局容器，侧边栏为次要内容布局容器

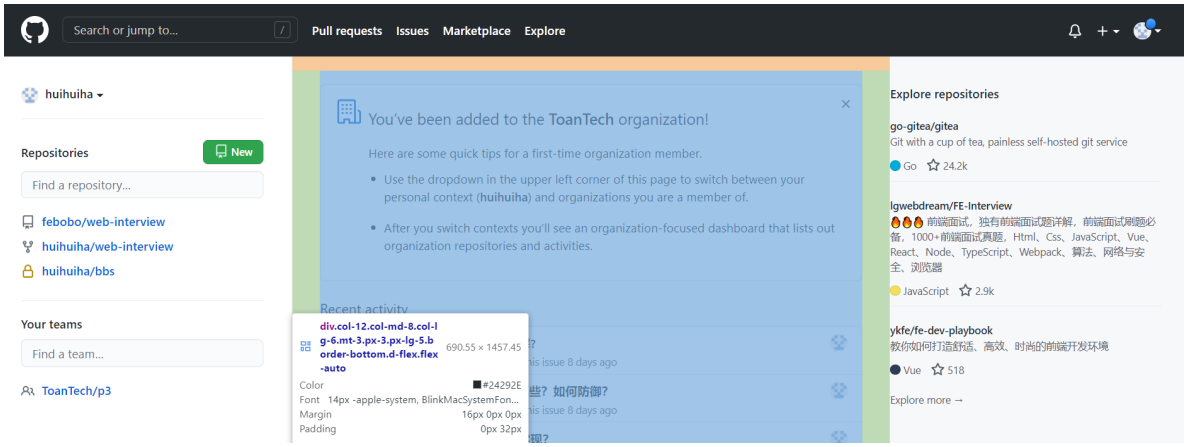
> 这里称宽度较小的列父元素为次要布局容器，宽度较大的列父元素为主要布局容器

这种布局适用于内容上具有明显主次关系的网页

三栏布局

三栏布局按照左中右的顺序进行排列，通常中间列最宽，左右两列次之

大家最常见的就是`github`：



二、双栏布局

双栏布局非常常见，往往是以一个定宽栏和一个自适应的栏并排展示存在

实现思路也非常的简单：

- 使用 float 左浮左边栏
- 右边模块使用 margin-left 撑出内容块做内容展示
- 为父级元素添加BFC，防止下方元素飞到上方内容

代码如下：

```
<style>
    .box{
        overflow: hidden; 添加BFC
    }
    .left {
        float: left;
        width: 200px;
        background-color: gray;
        height: 400px;
    }
    .right {
        margin-left: 210px;
        background-color: lightgray;
        height: 200px;
    }
</style>
<div class="box">
    <div class="left">左边</div>
    <div class="right">右边</div>
</div>
```

还有一种更为简单的使用则是采取：flex弹性布局

flex弹性布局

```
<style>
```

```
    .box{
        display: flex;
    }
    .left {
        width: 100px;
    }
    .right {
        flex: 1;
    }
</style>
<div class="box">
    <div class="left">左边</div>
    <div class="right">右边</div>
</div>
```

`flex`可以说是最好的方案了，代码少，使用简单

注意的是，`flex`容器的一个默认属性值:`align-items: stretch;`

这个属性导致了列等高的效果。 为了让两个盒子高度自动，需要设置:`align-items: flex-start`

三、三栏布局

实现三栏布局中间自适应的布局方式有：

- 两边使用 float，中间使用 margin
- 两边使用 absolute，中间使用 margin
- 两边使用 float 和负 margin
- display: table 实现
- flex实现
- grid网格布局

两边使用 float，中间使用 margin

需要将中间的内容放在`html`结构最后，否则右侧会臣在中间内容的下方

实现代码如下：

```
<style>
    .wrap {
        background: #eee;
        overflow: hidden; <!-- 生成BFC，计算高度时考虑浮动的元素 -->
        padding: 20px;
        height: 200px;
    }
    .left {
        width: 200px;
        height: 200px;
        float: left;
        background: coral;
    }
    .right {
        width: 120px;
        height: 200px;
        float: right;
        background: lightblue;
```

```
    }
    .middle {
        margin-left: 220px;
        height: 200px;
        background: lightpink;
        margin-right: 140px;
    }
</style>
<div class="wrap">
    <div class="left">左侧</div>
    <div class="right">右侧</div>
    <div class="middle">中间</div>
</div>
```

原理如下：

- 两边固定宽度，中间宽度自适应。
- 利用中间元素的margin值控制两边的间距
- 宽度小于左右部分宽度之和时，右侧部分会被挤下去

这种实现方式存在缺陷：

- 主体内容是最后加载的。
- 右边在主体内容之前，如果是响应式设计，不能简单的换行展示

两边使用 absolute，中间使用 margin

基于绝对定位的三栏布局：注意绝对定位的元素脱离文档流，相对于最近的已经定位的祖先元素进行定位。无需考虑HTML中结构的顺序

```
<style>
  .container {
    position: relative;
  }

  .left,
  .right,
  .main {
    height: 200px;
    line-height: 200px;
    text-align: center;
  }

  .left {
    position: absolute;
    top: 0;
    left: 0;
    width: 100px;
    background: green;
  }

  .right {
    position: absolute;
    top: 0;
    right: 0;
    width: 100px;
```

```
    background: green;
  }

  .main {
    margin: 0 110px;
    background: black;
    color: white;
  }
</style>

<div class="container">
  <div class="left">左边固定宽度</div>
  <div class="right">右边固定宽度</div>
  <div class="main">中间自适应</div>
</div>
```

实现流程：

- 左右两边使用绝对定位，固定在两侧。
- 中间占满一行，但通过 margin和左右两边留出10px的间隔

两边使用 float 和负 margin

```
<style>
  .left,
  .right,
  .main {
    height: 200px;
    line-height: 200px;
    text-align: center;
  }

  .main-wrapper {
    float: left;
    width: 100%;
  }

  .main {
    margin: 0 110px;
    background: black;
    color: white;
  }

  .left,
  .right {
    float: left;
    width: 100px;
    margin-left: -100%;
    background: green;
  }

  .right {
    margin-left: -100px; /* 同自身宽度 */
  }
```

```
</style>

<div class="main-wrapper">
  <div class="main">中间自适应</div>
</div>
<div class="left">左边固定宽度</div>
<div class="right">右边固定宽度</div>
```

实现过程：

- 中间使用了双层标签，外层是浮动的，以便左中右能在同一行展示
- 左边通过使用负 margin-left:-100%，相当于中间的宽度，所以向上偏移到左侧
- 右边通过使用负 margin-left:-100px，相当于自身宽度，所以向上偏移到最右侧

缺点：

- 增加了 .main-wrapper 一层，结构变复杂
- 使用负 margin，调试也相对麻烦

使用 display: table 实现

`<table>` 标签用于展示行列数据，不适合用于布局。但是可以使用 `display: table` 来实现布局的效果

```
<style>
  .container {
    height: 200px;
    line-height: 200px;
    text-align: center;
    display: table;
    table-layout: fixed;
    width: 100%;
  }

  .left,
  .right,
  .main {
    display: table-cell;
  }

  .left,
  .right {
    width: 100px;
    background: green;
  }

  .main {
    background: black;
    color: white;
    width: 100%;
  }
</style>

<div class="container">
```

```
  <div class="left">左边固定宽度</div>
  <div class="main">中间自适应</div>
  <div class="right">右边固定宽度</div>
</div>
```

实现原理：

- 层通过 display: table设置为表格，设置 table-layout: fixed`表示列宽自身宽度决定，而不是自动计算。
- 内层的左中右通过 display: table-cell设置为表格单元。
- 左右设置固定宽度，中间设置 width: 100% 填充剩下的宽度

使用flex实现

利用 `flex` 弹性布局，可以简单实现中间自适应

代码如下：

```
<style type="text/css">
    .wrap {
        display: flex;
        justify-content: space-between;
    }

    .left,
    .right,
    .middle {
        height: 100px;
    }

    .left {
        width: 200px;
        background: coral;
    }

    .right {
        width: 120px;
        background: lightblue;
    }

    .middle {
        background: #555;
        width: 100%;
        margin: 0 20px;
    }
</style>
<div class="wrap">
    <div class="left">左侧</div>
    <div class="middle">中间</div>
    <div class="right">右侧</div>
</div>
```

实现过程：

- 仅需将容器设置为`display:flex;`，
- 盒内元素两端对其，将中间元素设置为`100%`宽度，或者设为`flex:1`，即可填充空白
- 盒内元素的高度撑开容器的高度

优点：

- 结构简单直观
- 可以结合 flex的其他功能实现更多效果，例如使用 order属性调整显示顺序，让主体内容优先加载，但展示在中间

grid网格布局

代码如下：

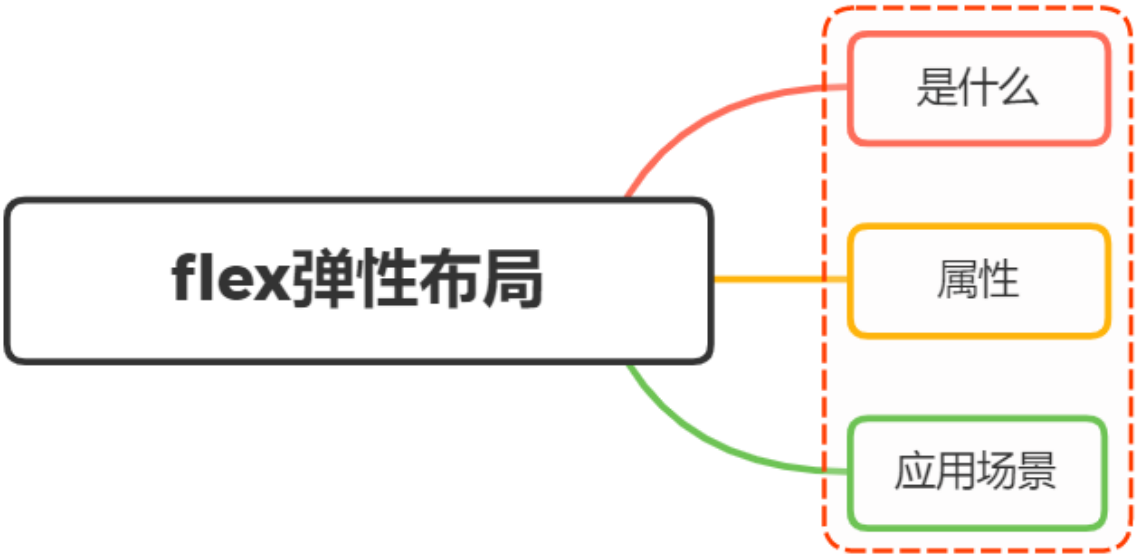
```
<style>
    .wrap {
        display: grid;
        width: 100%;
        grid-template-columns: 300px auto 300px;
    }

    .left,
    .right,
    .middle {
        height: 100px;
    }

    .left {
        background: coral;
    }

    .right {
        width: 300px;
        background: lightblue;
    }

    .middle {
        background: #555;
    }
</style>
<div class="wrap">
    <div class="left">左侧</div>
    <div class="middle">中间</div>
    <div class="right">右侧</div>
</div>
```

跟`flex`弹性布局一样的简单

参考文献

- https://zhuqingguang.github.io/2017/08/16/adapting-two-layout/
- https://segmentfault.com/a/1190000008705541

06.说说flexbox（弹性盒布局模型）,以及适用场景

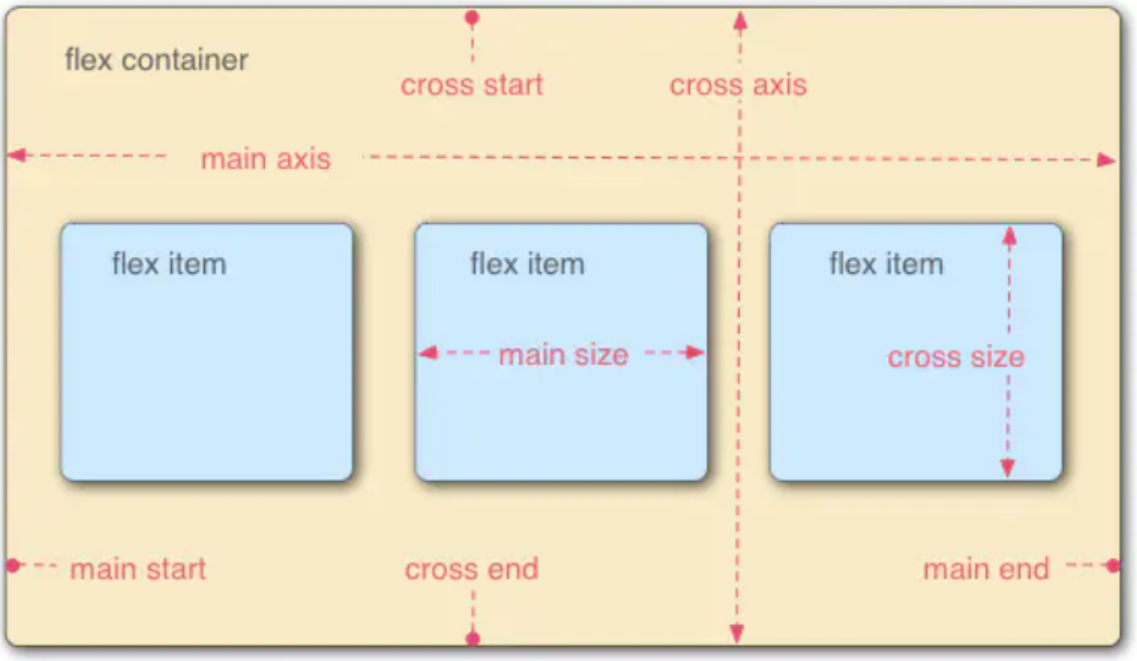


一、是什么

`Flexible Box` 简称 `flex`，意为"弹性布局"，可以简便、完整、响应式地实现各种页面布局

采用Flex布局的元素，称为 `flex` 容器 `container`

它的所有子元素自动成为容器成员，称为 `flex` 项目 `item`



容器中默认存在两条轴，主轴和交叉轴，呈90度关系。项目默认沿主轴排列，通过 `flex-direction` 来决定主轴的方向

每根轴都有起点和终点，这对于元素的对齐非常重要

二、属性

关于 `flex` 常用的属性，我们可以划分为容器属性和容器成员属性

容器属性有：

- flex-direction
- flex-wrap
- flex-flow
- justify-content
- align-items
- align-content

flex-direction

决定主轴的方向(即项目的排列方向)

```
.container {
    flex-direction: row | row-reverse | column | column-reverse;
}
```

属性对应如下：

- row（默认值）：主轴为水平方向，起点在左端
- row-reverse：主轴为水平方向，起点在右端
- column：主轴为垂直方向，起点在上沿。
- column-reverse：主轴为垂直方向，起点在下沿

如下图所示：

flex-wrap

弹性元素永远沿主轴排列，那么如果主轴排不下，通过`flex-wrap`决定容器内项目是否可换行

```
.container {
    flex-wrap: nowrap | wrap | wrap-reverse;
}
```

属性对应如下：

- nowrap（默认值）：不换行
- wrap：换行，第一行在上方
- wrap-reverse：换行，第一行在下方

默认情况是不换行，但这里也不会任由元素直接溢出容器，会涉及到元素的弹性伸缩

flex-flow

是`flex-direction`属性和`flex-wrap`属性的简写形式，默认值为`row nowrap`

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

justify-content

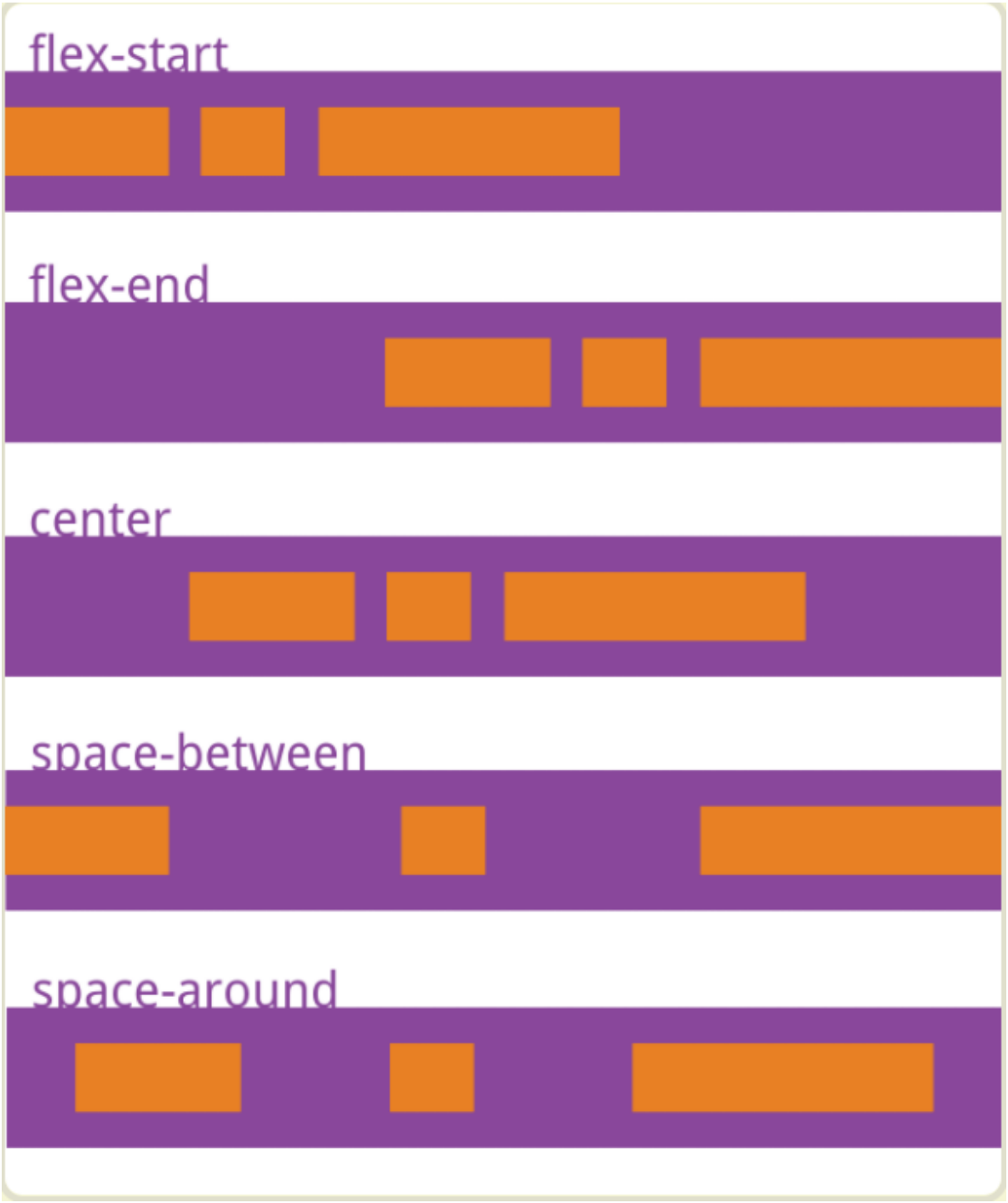
定义了项目在主轴上的对齐方式

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

属性对应如下：

- flex-start（默认值）：左对齐
- flex-end：右对齐
- center：居中
- space-between：两端对齐，项目之间的间隔都相等
- space-around：两个项目两侧间隔相等

效果图如下：



align-items

定义项目在交叉轴上如何对齐

```
.box {
  align-items: flex-start | flex-end | center | baseline | stretch;
}
```

属性对应如下：

- flex-start：交叉轴的起点对齐
- flex-end：交叉轴的终点对齐
- center：交叉轴的中点对齐
- baseline: 项目的第一行文字的基线对齐

- stretch（默认值）：如果项目未设置高度或设为auto，将占满整个容器的高度

align-content

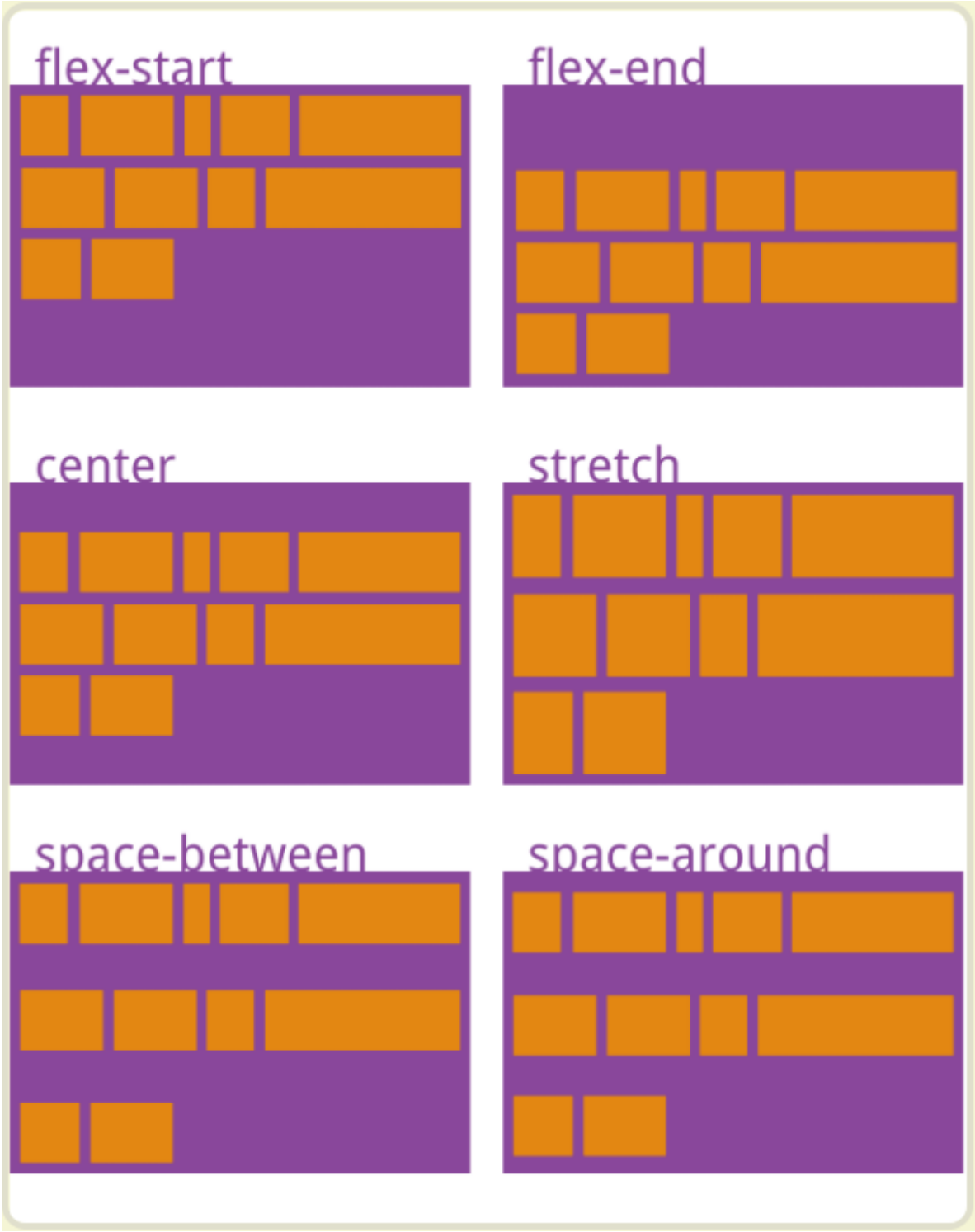
定义了多根轴线的对齐方式。如果项目只有一根轴线，该属性不起作用

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

属性对应如吓：

- flex-start：与交叉轴的起点对齐
- flex-end：与交叉轴的终点对齐
- center：与交叉轴的中点对齐
- space-between：与交叉轴两端对齐，轴线之间的间隔平均分布
- space-around：每根轴线两侧的间隔都相等。所以，轴线之间的间隔比轴线与边框的间隔大一倍
- stretch（默认值）：轴线占满整个交叉轴

效果图如下：



容器成员属性如下：

- `order`
- `flex-grow`
- `flex-shrink`
- `flex-basis`
- `flex`
- `align-self`

order

定义项目的排列顺序。数值越小，排列越靠前，默认为0

```
.item {
    order: <integer>;
}
```

flex-grow

上面讲到当容器设为`flex-wrap: nowrap;`不换行的时候，容器宽度有不够分的情况，弹性元素会根据`flex-grow`来决定

定义项目的放大比例（容器宽度>元素总宽度时如何伸展）

默认为`0`，即如果存在剩余空间，也不放大

```
.item {
    flex-grow: <number>;
}
```

如果所有项目的`flex-grow`属性都为1，则它们将等分剩余空间（如果有的话）

如果一个项目的`flex-grow`属性为2，其他项目都为1，则前者占据的剩余空间将比其他项多一倍

弹性容器的宽度正好等于元素宽度总和，无多余宽度，此时无论`flex-grow`是什么值都不会生效

flex-shrink

定义了项目的缩小比例（容器宽度<元素总宽度时如何收缩），默认为1，即如果空间不足，该项目将缩小

```
.item {
    flex-shrink: <number>; /* default 1 */
}
```

如果所有项目的`flex-shrink`属性都为1，当空间不足时，都将等比例缩小

如果一个项目的`flex-shrink`属性为0，其他项目都为1，则空间不足时，前者不缩小

在容器宽度有剩余时，`flex-shrink`也是不会生效的

flex-basis

设置的是元素在主轴上的初始尺寸，所谓的初始尺寸就是元素在`flex-grow`和`flex-shrink`生效前的尺寸

浏览器根据这个属性，计算主轴是否有多余空间，默认值为`auto`，即项目的本来大小，如设置了`width`则元素尺寸由`width/height`决定（主轴方向），没有设置则由内容决定

```
.item {
    flex-basis: <length> | auto; /* default auto */
}
```

当设置为0的是，会根据内容撑开

它可以设为跟`width`或`height`属性一样的值（比如350px），则项目将占据固定空间

flex

`flex`属性是`flex-grow`，`flex-shrink` 和 `flex-basis`的简写，默认值为`0 1 auto`，也是比较难懂的一个复合属性

```
.item {
  flex: none | [ <'flex-grow'> <'flex-shrink'>? || <'flex-basis'> ]
}
```

一些属性有：

- flex: 1 = flex: 1 1 0%
- flex: 2 = flex: 2 1 0%
- flex: auto = flex: 1 1 auto
- flex: none = flex: 0 0 auto，常用于固定尺寸不伸缩

`flex:1` 和 `flex:auto` 的区别，可以归结于`flex-basis:0`和`flex-basis:auto`的区别

当设置为0时（绝对弹性元素），此时相当于告诉`flex-grow`和`flex-shrink`在伸缩的时候不需要考虑我的尺寸

当设置为`auto`时（相对弹性元素），此时则需要在伸缩时将元素尺寸纳入考虑

注意：建议优先使用这个属性，而不是单独写三个分离的属性，因为浏览器会推算相关值

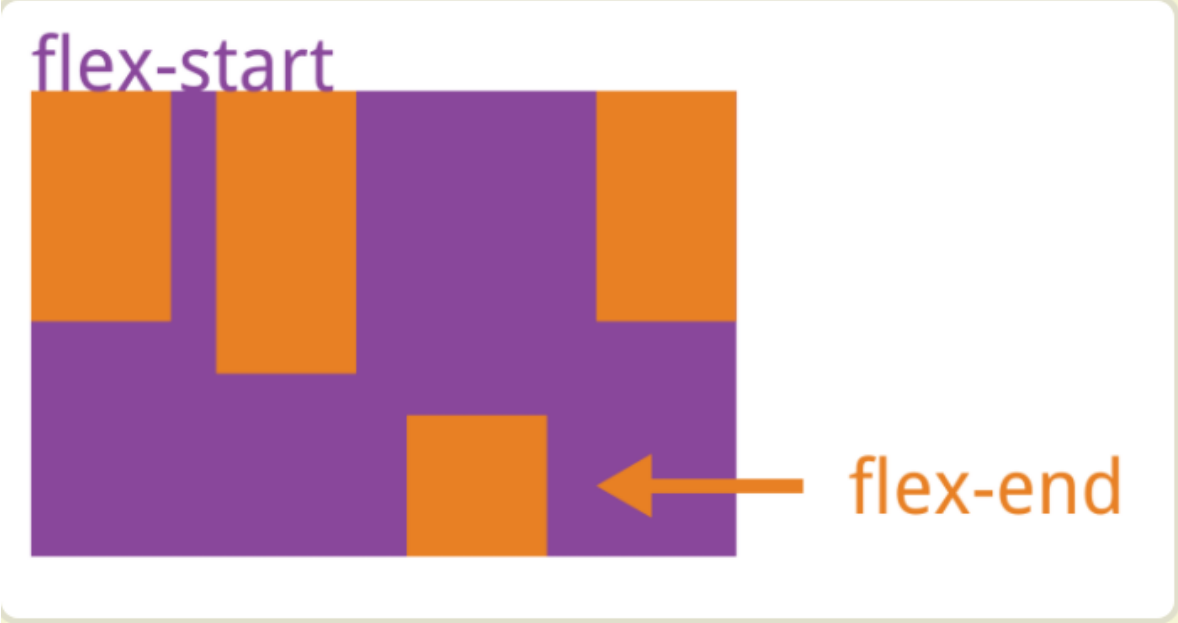
align-self

允许单个项目有与其他项目不一样的对齐方式，可覆盖`align-items`属性

默认值为`auto`，表示继承父元素的`align-items`属性，如果没有父元素，则等同于`stretch`

```
.item {
    align-self: auto | flex-start | flex-end | center | baseline | stretch;
}
```

效果图如下：



三、应用场景

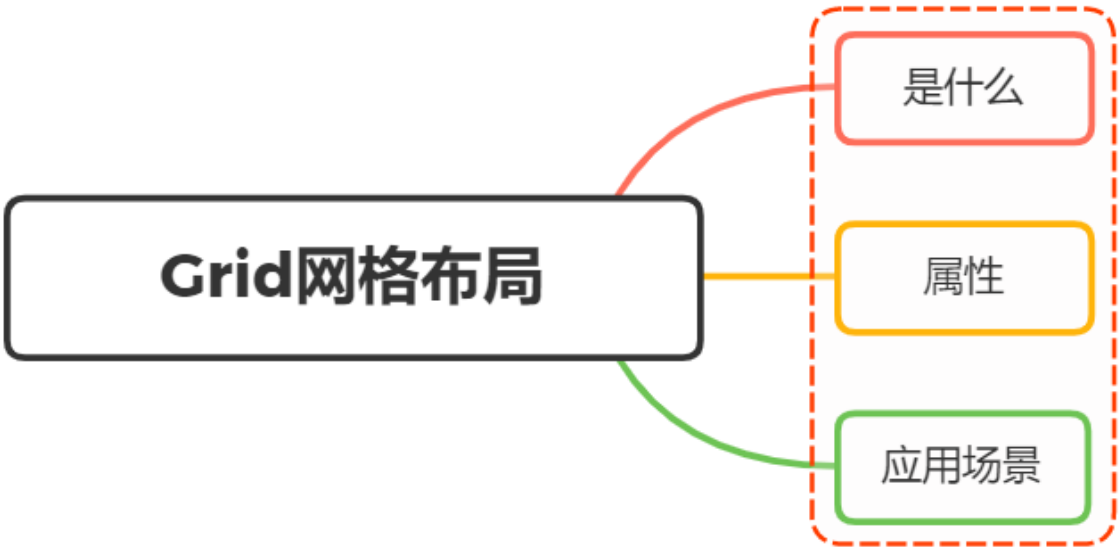
在以前的文章中，我们能够通过`flex`简单粗暴的实现元素水平垂直方向的居中，以及在两栏三栏自适应布局中通过`flex`完成，这里就不再展开代码的演示

包括现在在移动端、小程序这边的开发，都建议使用`flex`进行布局

参考文献

- https://developer.mozilla.org/zh-CN/docs/Web/CSS/flex
- http://www.ruanyifeng.com/blog/2015/07/flex-grammar.html

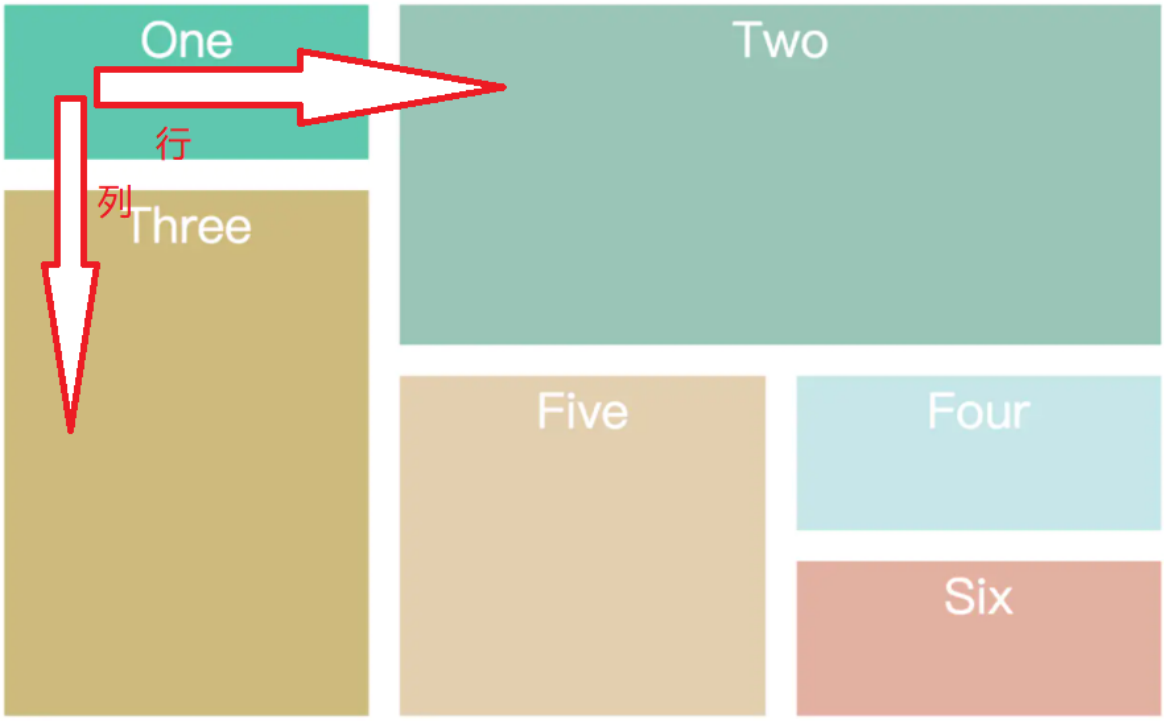
07.介绍一下grid网格布局



一、是什么

`Grid` 布局即网格布局，是一个二维的布局方式，由纵横相交的两组网格线形成的框架性布局结构，能够同时处理行与列

擅长将一个页面划分为几个主要区域，以及定义这些区域的大小、位置、层次等关系



这与之前讲到的 `flex` 一维布局不相同

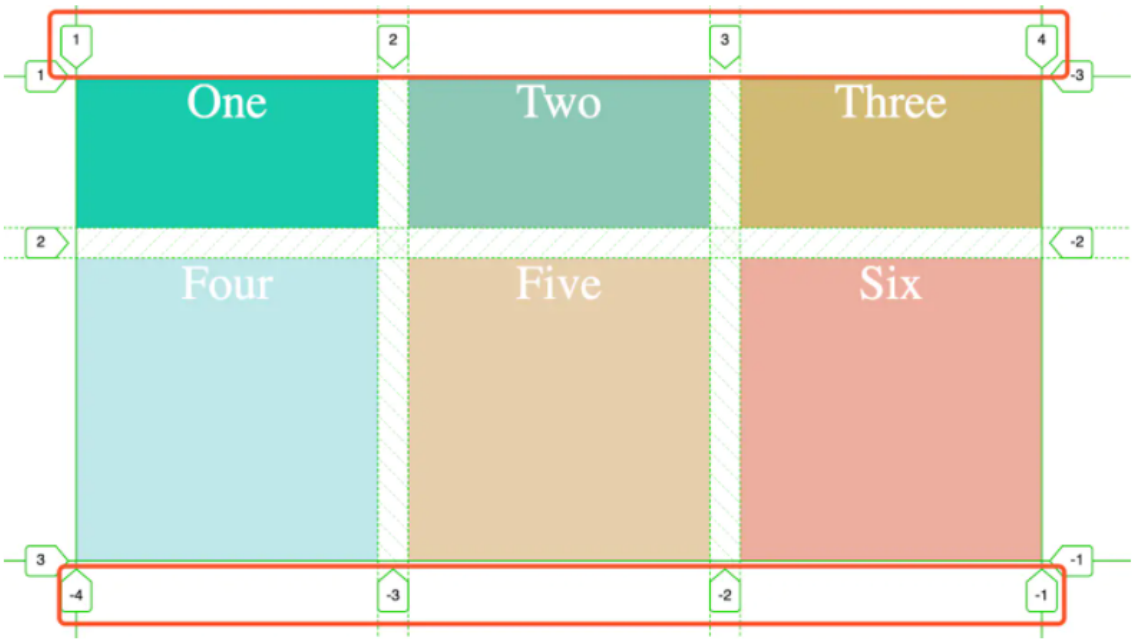
设置 `display:grid/inline-grid` 的元素就是网格布局容器，这样就能出发浏览器渲染引擎的网格布局算法

```
<div class="container">
    <div class="item item-1">
        <p class="sub-item"></p >
 </div>
    <div class="item item-2"></div>
    <div class="item item-3"></div>
</div>
```

上述代码实例中，`.container`元素就是网格布局容器，`.item`元素就是网格的项目，由于网格元素只能是容器的顶层子元素，所以`p`元素并不是网格元素

这里提一下，网格线概念，有助于下面对`grid-column`系列属性的理解

网格线，即划分网格的线，如下图所示：



上图是一个 2 x 3 的网格，共有3根水平网格线和4根垂直网格线

二、属性

同样，`Grid`布局属性可以分为两大类：

- 容器属性，
- 项目属性

关于容器属性有如下：

display 属性

文章开头讲到，在元素上设置`display: grid`或`display: inline-grid`来创建一个网格容器

- display：grid 则该容器是一个块级元素
- display: inline-grid 则容器元素为行内元素

grid-template-columns 属性，grid-template-rows 属性

`grid-template-columns` 属性设置列宽，`grid-template-rows` 属性设置行高

```
.wrapper {
  display: grid;
  /*  声明了三列，宽度分别为 200px 200px 200px */
  grid-template-columns: 200px 200px 200px;
  grid-gap: 5px;
  /*  声明了两行，行高分别为 50px 50px  */
  grid-template-rows: 50px 50px;
}
```

以上表示固定列宽为 200px 200px 200px，行高为 50px 50px

上述代码可以看到重复写单元格宽高，通过使用`repeat()`函数，可以简写重复的值

- 第一个参数是重复的次数
- 第二个参数是重复的值

所以上述代码可以简写成

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(3,200px);
  grid-gap: 5px;
  grid-template-rows:repeat(2,50px);
}
```

除了上述的`repeact`关键字，还有：

- auto-fill：示自动填充，让一行（或者一列）中尽可能的容纳更多的单元格

> `grid-template-columns: repeat(auto-fill, 200px)` 表示列宽是 200 px，但列的数量是不固定的，只要浏览器能够容纳得下，就可以放置元素

- fr：片段，为了方便表示比例关系

> `grid-template-columns: 200px 1fr 2fr` 表示第一个列宽设置为 200px，后面剩余的宽度分为两部分，宽度分别为剩余宽度的 1/3 和 2/3

- minmax：产生一个长度范围，表示长度就在这个范围之中都可以应用到网格项目中。第一个参数就是最小值，第二个参数就是最大值

> `minmax(100px, 1fr)`表示列宽不小于`100px`，不大于`1fr`

- auto：由浏览器自己决定长度

> `grid-template-columns: 100px auto 100px` 表示第一第三列为 100px，中间由浏览器决定长度

grid-row-gap 属性， grid-column-gap 属性， grid-gap 属性

`grid-row-gap` 属性、`grid-column-gap` 属性分别设置行间距和列间距。`grid-gap` 属性是两者的简写形式

`grid-row-gap: 10px` 表示行间距是 10px

`grid-column-gap: 20px` 表示列间距是 20px

`grid-gap: 10px 20px` 等同上述两个属性

grid-template-areas 属性

用于定义区域，一个区域由一个或者多个单元格组成

```
.container {
  display: grid;
  grid-template-columns: 100px 100px 100px;
  grid-template-rows: 100px 100px 100px;
  grid-template-areas: 'a b c'
                       'd e f'
                       'g h i';
}
```

上面代码先划分出9个单元格，然后将其定名为`a`到`i`的九个区域，分别对应这九个单元格。

多个单元格合并成一个区域的写法如下

```
 grid-template-areas: 'a a a'
                      'b b b'
                      'c c c';
```

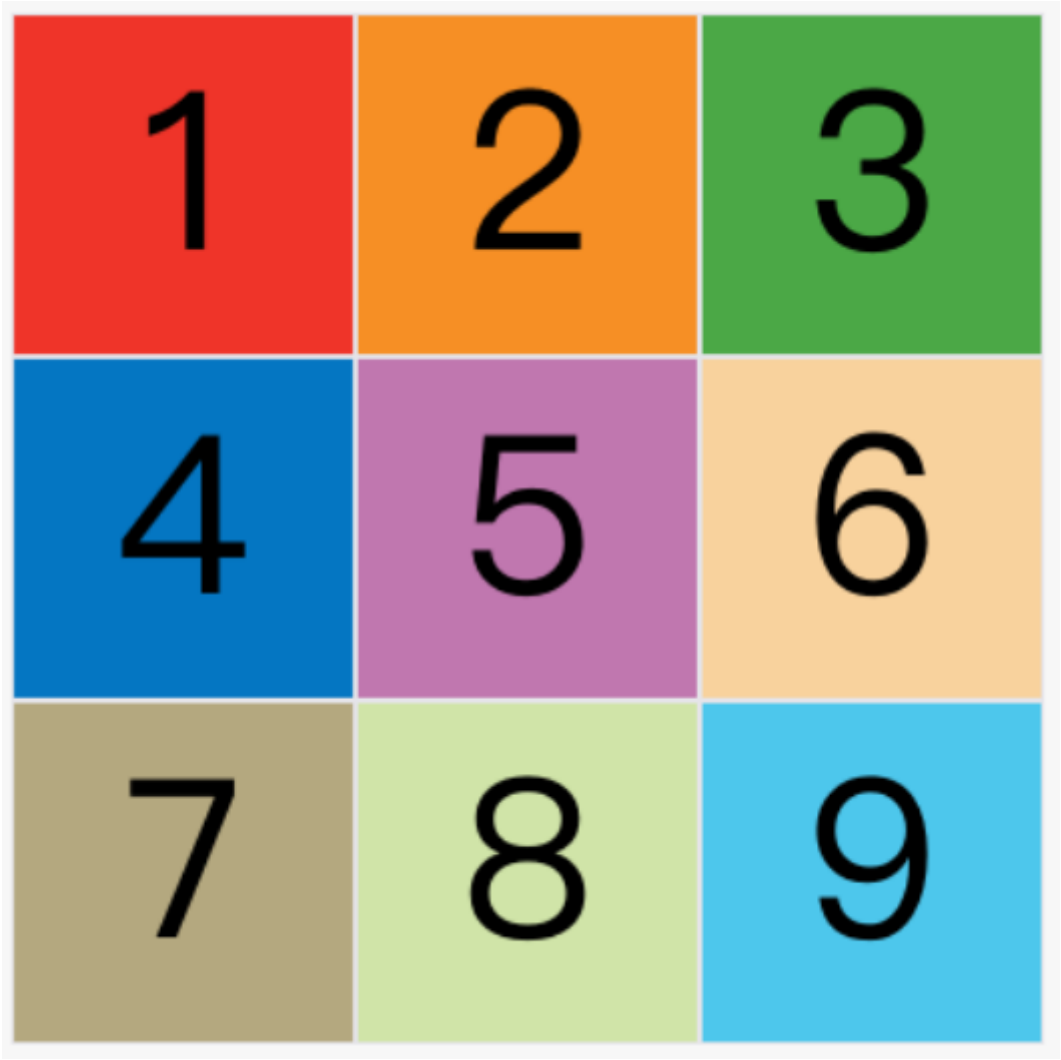
上面代码将9个单元格分成`a`、`b`、`c`三个区域

如果某些区域不需要利用，则使用"点"（`.`）表示

grid-auto-flow 属性

划分网格以后，容器的子元素会按照顺序，自动放置在每一个网格。

顺序就是由`grid-auto-flow`决定，默认为行，代表"先行后列"，即先填满第一行，再开始放入第二行



当修改成`column`后，放置变为如下：

justify-items 属性， align-items 属性， place-items 属性

`justify-items` 属性设置单元格内容的水平位置（左中右），`align-items` 属性设置单元格的垂直位置（上中下）

两者属性的值完成相同

```
.container {
  justify-items: start | end | center | stretch;
  align-items: start | end | center | stretch;
}
```

属性对应如下：

- start：对齐单元格的起始边缘
- end：对齐单元格的结束边缘
- center：单元格内部居中
- stretch：拉伸，占满单元格的整个宽度（默认值）

`place-items` 属性是 `align-items` 属性和 `justify-items` 属性的合并简写形式

justify-content 属性， align-content 属性， place-content 属性

`justify-content`属性是整个内容区域在容器里面的水平位置（左中右），`align-content`属性是整个内容区域的垂直位置（上中下）

```
.container {
  justify-content: start | end | center | stretch | space-around | space-between
| space-evenly;
  align-content: start | end | center | stretch | space-around | space-between |
space-evenly;
}
```

两个属性的写法完全相同，都可以取下面这些值：

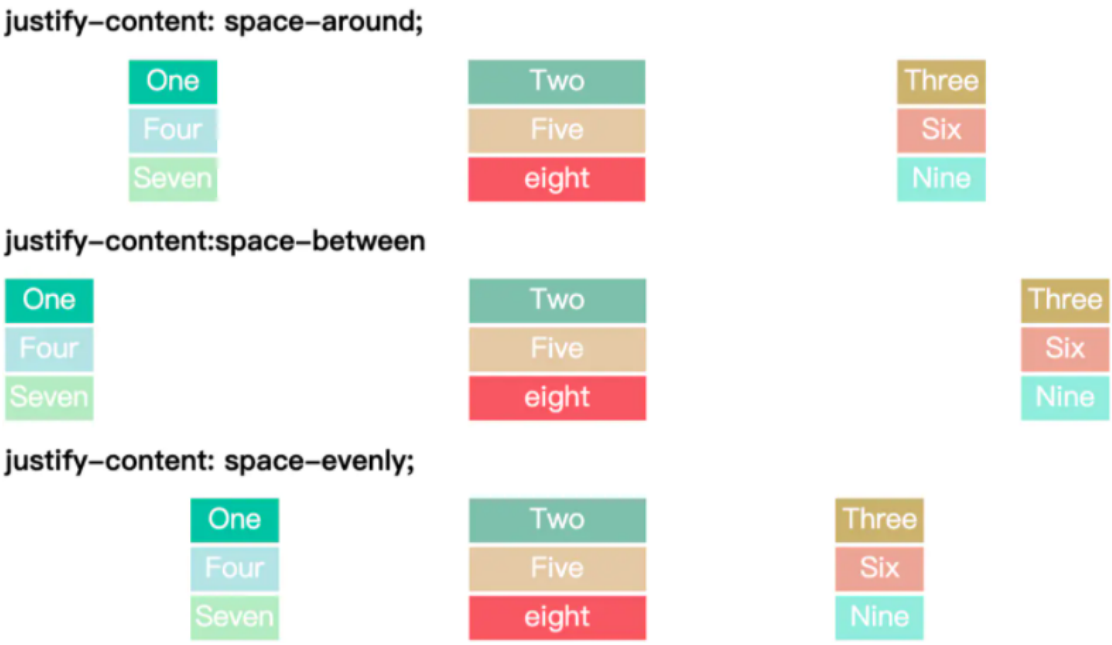
- start - 对齐容器的起始边框
- end - 对齐容器的结束边框
- center - 容器内部居中

justify-content: start;

justify-content: end;

justify-content: center;

- space-around - 每个项目两侧的间隔相等。所以，项目之间的间隔比项目与容器边框的间隔大一倍
- space-between - 项目与项目的间隔相等，项目与容器边框之间没有间隔
- space-evenly - 项目与项目的间隔相等，项目与容器边框之间也是同样长度的间隔
- stretch - 项目大小没有指定时，拉伸占据整个网格容器



grid-auto-columns 属性和 grid-auto-rows 属性

有时候，一些项目的指定位置，在现有网格的外部，就会产生显示网格和隐式网格

比如网格只有3列，但是某一个项目指定在第5行。这时，浏览器会自动生成多余的网格，以便放置项目。超出的部分就是隐式网格

而`grid-auto-rows`与`grid-auto-columns`就是专门用于指定隐式网格的宽高

关于项目属性，有如下：

grid-column-start 属性、grid-column-end 属性、grid-row-start 属性以及 grid-row-end 属性

指定网格项目所在的四个边框，分别定位在哪根网格线，从而指定项目的位置

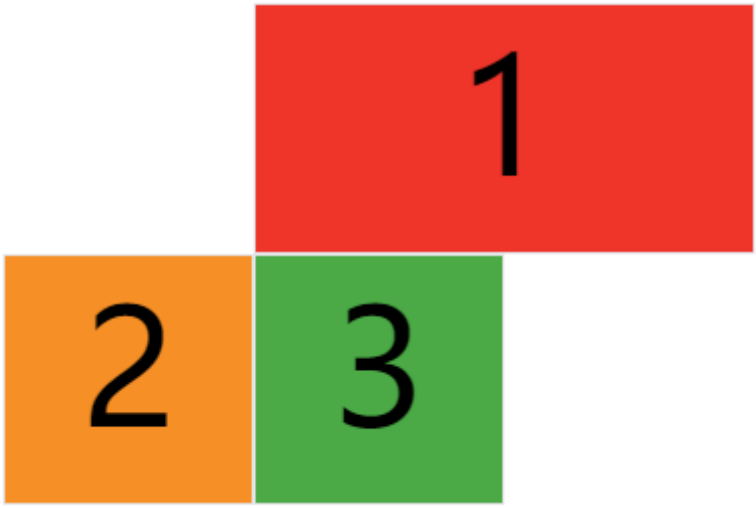
- grid-column-start 属性：左边框所在的垂直网格线
- grid-column-end 属性：右边框所在的垂直网格线
- grid-row-start 属性：上边框所在的水平网格线
- grid-row-end 属性：下边框所在的水平网格线

举个例子：

```
<style>
    #container{
        display: grid;
        grid-template-columns: 100px 100px 100px;
        grid-template-rows: 100px 100px 100px;
    }
    .item-1 {
        grid-column-start: 2;
        grid-column-end: 4;
```

```
    }
</style>

<div id="container">
    <div class="item item-1">1</div>
    <div class="item item-2">2</div>
    <div class="item item-3">3</div>
</div>
```

通过设置`grid-column`属性，指定1号项目的左边框是第二根垂直网格线，右边框是第四根垂直网格线



grid-area 属性

`grid-area`属性指定项目放在哪一个区域

```
.item-1 {
  grid-area: e;
}
```

意思为将1号项目位于`e`区域

与上述讲到的`grid-template-areas`搭配使用

justify-self 属性、align-self 属性以及 place-self 属性

`justify-self`属性设置单元格内容的水平位置（左中右），跟`justify-items`属性的用法完全一致，但只作用于单个项目。

`align-self`属性设置单元格内容的垂直位置（上中下），跟`align-items`属性的用法完全一致，也是只作用于单个项目

```
.item {
  justify-self: start | end | center | stretch;
  align-self: start | end | center | stretch;
}
```

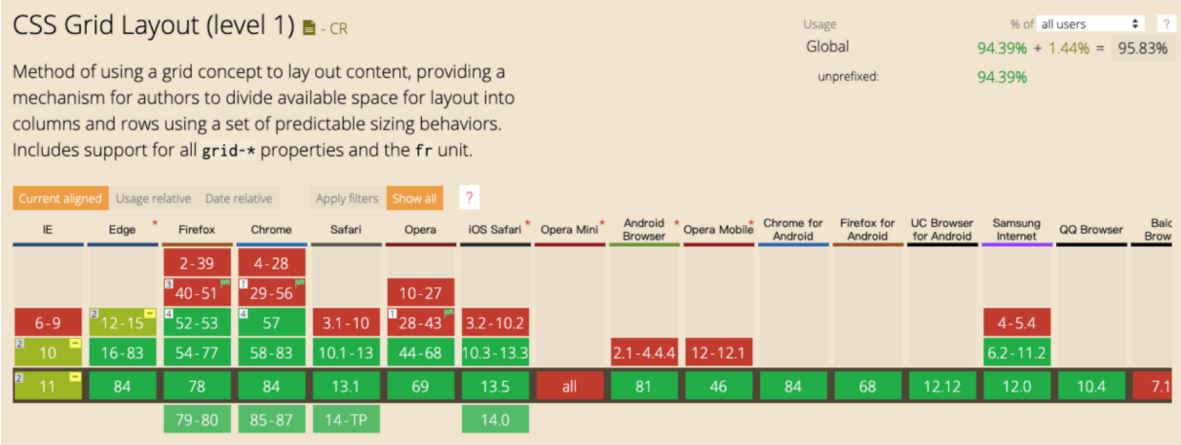
这两个属性都可以取下面四个值。

- start：对齐单元格的起始边缘。
- end：对齐单元格的结束边缘。
- center：单元格内部居中。
- stretch：拉伸，占满单元格的整个宽度（默认值）

三、应用场景

文章开头就讲到，`Grid`是一个强大的布局，如一些常见的 CSS 布局，如居中，两列布局，三列布局等等是很容易实现的，在以前的文章中，也有使用`Grid`布局完成对应的功能

关于兼容性问题，结果如下：



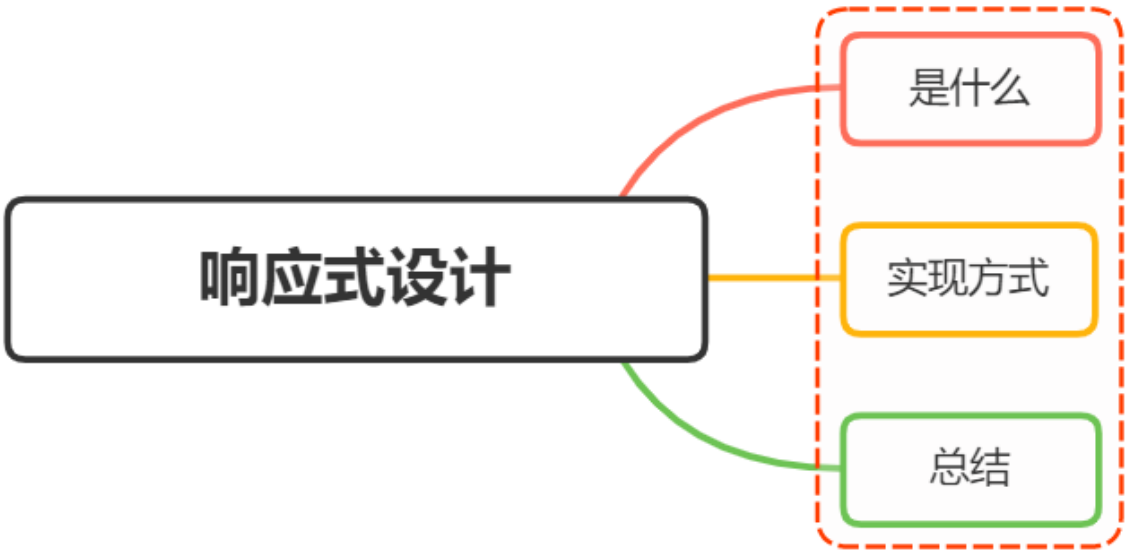
总体兼容性还不错，但在 IE 10 以下不支持

目前，`Grid`布局在手机端支持还不算太友好

参考文献

- https://developer.mozilla.org/zh-CN/docs/Web/CSS/CSS_Grid_Layout
- https://www.ruanyifeng.com/blog/2019/03/grid-layout-tutorial.html
- https://juejin.cn/post/6854573220306255880#heading-2

08.什么是响应式设计？响应式设计的基本原理是什么？如何做？



一、是什么

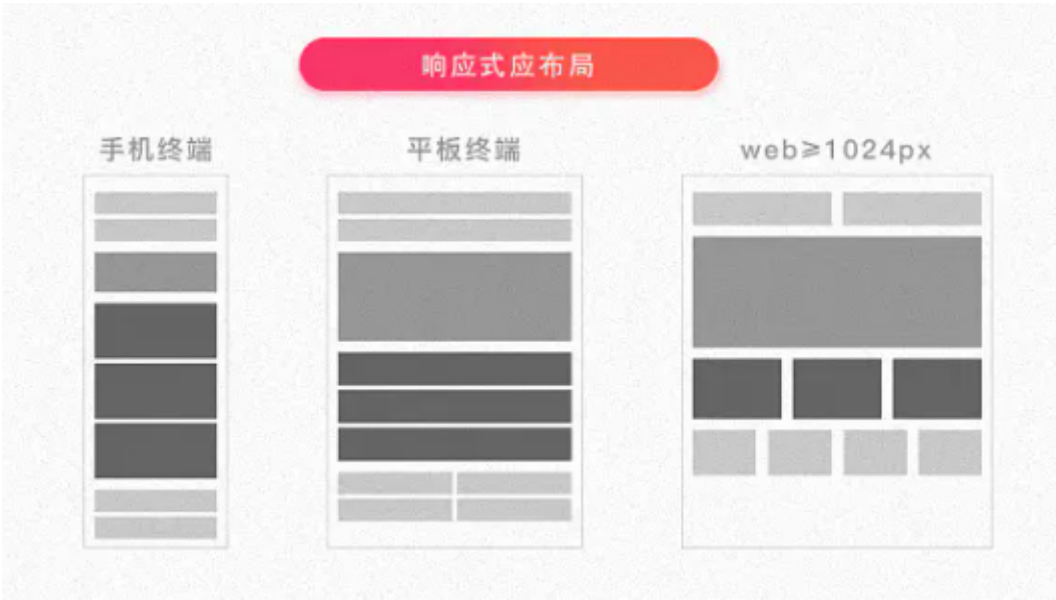
响应式网站设计（Responsive Web design）是一种网络页面设计布局，页面的设计与开发应当根据用户行为以及设备环境(系统平台、屏幕尺寸、屏幕定向等)进行相应的响应和调整

描述响应式界面最著名的一句话就是“Content is like water”

大白话便是“如果将屏幕看作容器，那么内容就像水一样”

响应式网站常见特点：

- 同时适配PC + 平板 + 手机等
- 标签导航在接近手持终端设备时改变为经典的抽屉式导航
- 网站的布局会根据视口来调整模块的大小和位置



二、实现方式

响应式设计的基本原理是通过媒体查询检测不同的设备屏幕尺寸做处理，为了处理移动端，页面头部必须有`meta`声明`viewport`

```
<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no”>
```

属性对应如下：

- width=device-width: 是自适应手机屏幕的尺寸宽度
- maximum-scale:是缩放比例的最大值
- inital-scale:是缩放的初始化
- user-scalable:是用户的可以缩放的操作

实现响应式布局的方式有如下：

- 媒体查询
- 百分比
- vw/vh
- rem

媒体查询

`CSS3`中的增加了更多的媒体查询，就像`if`条件表达式一样，我们可以设置不同类型的媒体条件，并根据对应的条件，给相应符合条件的媒体调用相对应的样式表

使用`@Media`查询，可以针对不同的媒体类型定义不同的样式，如：

```
@media screen and (max-width: 1920px) { ... }
```

当视口在375px - 600px之间，设置特定字体大小18px

```
@media screen (min-width: 375px) and (max-width: 600px) {
  body {
    font-size: 18px;
  }
}
```

通过媒体查询，可以通过给不同分辨率的设备编写不同的样式来实现响应式的布局，比如我们为不同分辨率的屏幕，设置不同的背景图片

比如给小屏幕手机设置@2x图，为大屏幕手机设置@3x图，通过媒体查询就能很方便的实现

百分比

通过百分比单位 " % " 来实现响应式的效果

比如当浏览器的宽度或者高度发生变化时，通过百分比单位，可以使得浏览器中的组件的宽和高随着浏览器的变化而变化，从而实现响应式的效果

`height`、`width`属性的百分比依托于父标签的宽高，但是其他盒子属性则不完全依赖父元素：

- 子元素的top/left和bottom/right如果设置百分比，则相对于直接非static定位(默认定位)的父元素的高度/宽度
- 子元素的padding如果设置百分比，不论是垂直方向或者是水平方向，都相对于直接父亲元素的width，而与父元素的height无关。

- 子元素的margin如果设置成百分比，不论是垂直方向还是水平方向，都相对于直接父元素的width
- border-radius不一样，如果设置border-radius为百分比，则是相对于自身的宽度

可以看到每个属性都使用百分比，会照成布局的复杂度，所以不建议使用百分比来实现响应式

vw/vh

`vw`表示相对于视图窗口的宽度，`vh`表示相对于视图窗口高度。 任意层级元素，在使用`vw`单位的情况下，`1vw`都等于视图宽度的百分之一

与百分比布局很相似，在以前文章提过与`%`的区别，这里就不再展开述说

rem

在以前也讲到，`rem`是相对于根元素`html`的`font-size`属性，默认情况下浏览器字体大小为`16px`，此时`1rem = 16px`

可以利用前面提到的媒体查询，针对不同设备分辨率改变`font-size`的值，如下：

```
@media screen and (max-width: 414px) {
  html {
    font-size: 18px
  }
}

@media screen and (max-width: 375px) {
  html {
    font-size: 16px
  }
}

@media screen and (max-width: 320px) {
  html {
    font-size: 12px
  }
}
```

为了更准确监听设备可视窗口变化，我们可以在`css`之前插入`script`标签，内容如下：

```
//动态为根元素设置字体大小
function init () {
    // 获取屏幕宽度
    var width = document.documentElement.clientWidth
    // 设置根元素字体大小。此时为宽的10等分
    document.documentElement.style.fontSize = width / 10 + 'px'
}

//首次加载应用，设置一次
init()
// 监听手机旋转的事件的时机，重新设置
window.addEventListener('orientationchange', init)
// 监听手机窗口变化，重新设置
```

```
window.addEventListener('resize', init)
```

无论设备可视窗口如何变化，始终设置`rem`为`width`的1/10，实现了百分比布局

除此之外，我们还可以利用主流`UI`框架，如：`element ui`、`antd`提供的栅格布局实现响应式

小结

响应式设计实现通常会从以下几方面思考：

- 弹性盒子（包括图片、表格、视频）和媒体查询等技术
- 使用百分比布局创建流式布局的弹性UI，同时使用媒体查询限制元素的尺寸和内容变更范围
- 使用相对单位使得内容自适应调节
- 选择断点，针对不同断点实现不同布局和内容展示

三、总结

响应式布局优点可以看到：

- 面对不同分辨率设备灵活性强
- 能够快捷解决多设备显示适应问题

缺点：

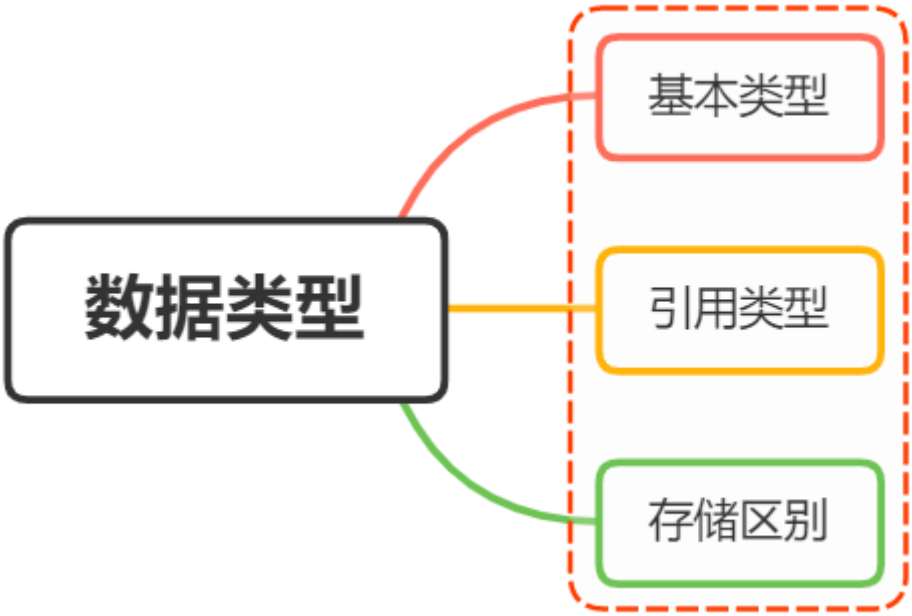
- 仅适用布局、信息、框架并不复杂的部门类型网站
- 兼容各种设备工作量大，效率低下
- 代码累赘，会出现隐藏无用的元素，加载时间加长
- 其实这是一种折中性质的设计解决方案，多方面因素影响而达不到最佳效果
- 一定程度上改变了网站原有的布局结构，会出现用户混淆的情况

参考文献

- https://baike.baidu.com/item/%E5%93%8D%E5%BA%94%E5%BC%8F%E7%BD%91%E9%A1%B5%E8%AE%BE%E8%AE%A1
- https://juejin.cn/post/6844904082751111176
- https://vue3js.cn/interview

3.JS基础

01.说说JavaScript中的数据类型？存储上的差别？



前言

在`JavaScript`中，我们可以分成两种类型：

- 基本类型
- 复杂类型

两种类型的区别是：存储位置不同

一、基本类型

基本类型主要为以下6种：

- Number
- String
- Boolean
- Undefined
- null
- symbol

Number

数值最常见的整数类型格式则为十进制，还可以设置八进制（零开头）、十六进制（0x开头）

```
let intNum = 55 // 10进制的55
let num1 = 070 // 8进制的56
let hexNum1 = 0xA //16进制的10
```

浮点类型则在数值汇总必须包含小数点，还可通过科学计数法表示

```
let floatNum1 = 1.1;
let floatNum2 = 0.1;
let floatNum3 = .1; // 有效，但不推荐
let floatNum = 3.125e7; // 等于 31250000
```

在数值类型中，存在一个特殊数值`NaN`，意为“不是数值”，用于表示本来要返回数值的操作失败了（而不是抛出错误）

```
console.log(0/0); // NaN
console.log(-0/+0); // NaN
```

Undefined

`Undefined`类型只有一个值，就是特殊值`undefined`。当使用`var`或`let`声明了变量但没有初始化时，就相当于给变量赋予了`undefined`值

```
let message;
console.log(message == undefined); // true
```

包含`undefined`值的变量跟未定义变量是有区别的

```
let message; // 这个变量被声明了，只是值为 undefined

console.log(message); // "undefined"
console.log(age); // 没有声明过这个变量，报错
```

String

字符串可以使用双引号（"）、单引号（'）或反引号（`）标示

```
let firstName = "John";
let lastName = 'Jacob';
let lastName = `Jingleheimerschmidt`
```

字符串是不可变的，意思是一旦创建，它们的值就不能变了

```
let lang = "Java";
lang = lang + "Script";  // 先销毁再创建
```

Null

`Null`类型同样只有一个值，即特殊值`null`

逻辑上讲， null 值表示一个空对象指针，这也是给`typeof`传一个`null`会返回`"object"`的原因

```
let car = null;
console.log(typeof car); // "object"
```

`undefined`值是由`null`值派生而来

```
console.log(null == undefined); // true
```

只要变量要保存对象，而当时又没有那个对象可保存，就可用 `null` 来填充该变量

Boolean

`Boolean`（布尔值）类型有两个字面值： `true` 和 `false`

通过 `Boolean` 可以将其他类型的数据转化成布尔值

规则如下：

数据类型	转换为 true 的值	转换为 false 的值
String	非空字符串	""
Number	非零数值（包括无穷值）	0 、 NaN
Object	任意对象	null
Undefined	N/A （不存在）	undefined

Symbol

Symbol （符号）是原始值，且符号实例是唯一、不可变的。符号的用途是确保对象属性使用唯一标识符，不会发生属性冲突的危险

```
let genericSymbol = Symbol();
let otherGenericSymbol = Symbol();
console.log(genericSymbol == otherGenericSymbol); // false

let fooSymbol = Symbol('foo');
let otherFooSymbol = Symbol('foo');
console.log(fooSymbol == otherFooSymbol); // false
```

二、引用类型

复杂类型统称为 `Object`，我们这里主要讲述下面三种：

- Object
- Array
- Function

Object

创建 `object` 常用方式为对象字面量表示法，属性名可以是字符串或数值

```
let person = {
    name: "Nicholas",
    "age": 29,
    5: true
};
```

Array

`JavaScript`数组是一组有序的数据，但跟其他语言不同的是，数组中每个槽位可以存储任意类型的数据。并且，数组也是动态大小的，会随着数据添加而自动增长

```
let colors = ["red", 2, {age: 20 }]
colors.push(2)
```

Function

函数实际上是对象，每个函数都是`Function`类型的实例，而`Function`也有属性和方法，跟其他引用类型一样

函数存在三种常见的表达方式：

- 函数声明

```
// 函数声明
function sum (num1, num2) {
    return num1 + num2;
}
```

- 函数表达式

```
let sum = function(num1, num2) {
    return num1 + num2;
};
```

- 箭头函数

函数声明和函数表达式两种方式

```
let sum = (num1, num2) => {
    return num1 + num2;
};
```

其他引用类型

除了上述说的三种之外，还包括`Date`、`RegExp`、`Map`、`Set`等.....

三、存储区别

基本数据类型和引用数据类型存储在内存中的位置不同：

- 基本数据类型存储在栈中
- 引用类型的对象存储于堆中

当我们把变量赋值给一个变量时，解析器首先要确认的就是这个值是基本类型值还是引用类型值

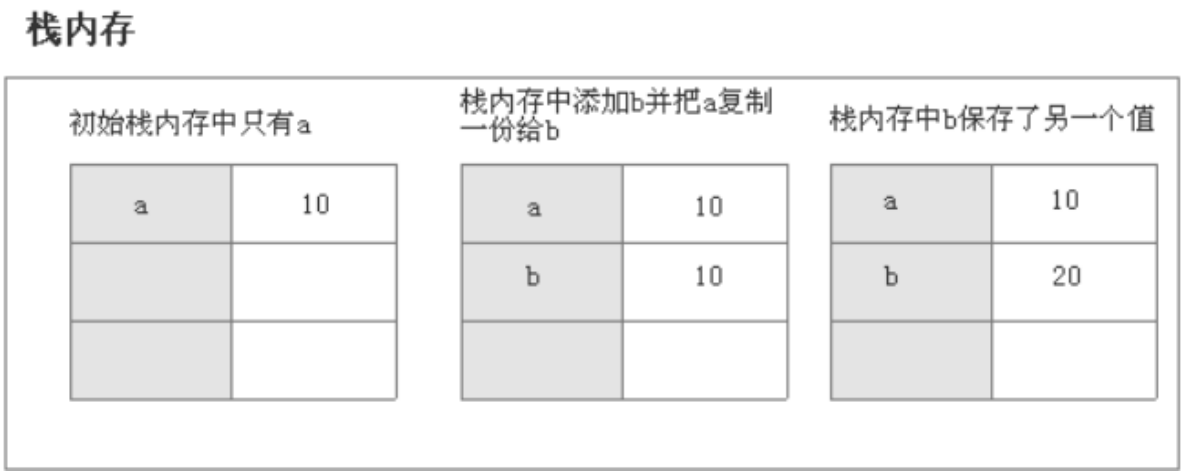
下面来举个例子

基本类型

```
let a = 10;
let b = a; // 赋值操作
b = 20;
console.log(a); // 10值
```

`a`的值为一个基本类型，是存储在栈中，将`a`的值赋给`b`，虽然两个变量的值相等，但是两个变量保存了两个不同的内存地址

下图演示了基本类型赋值的过程：



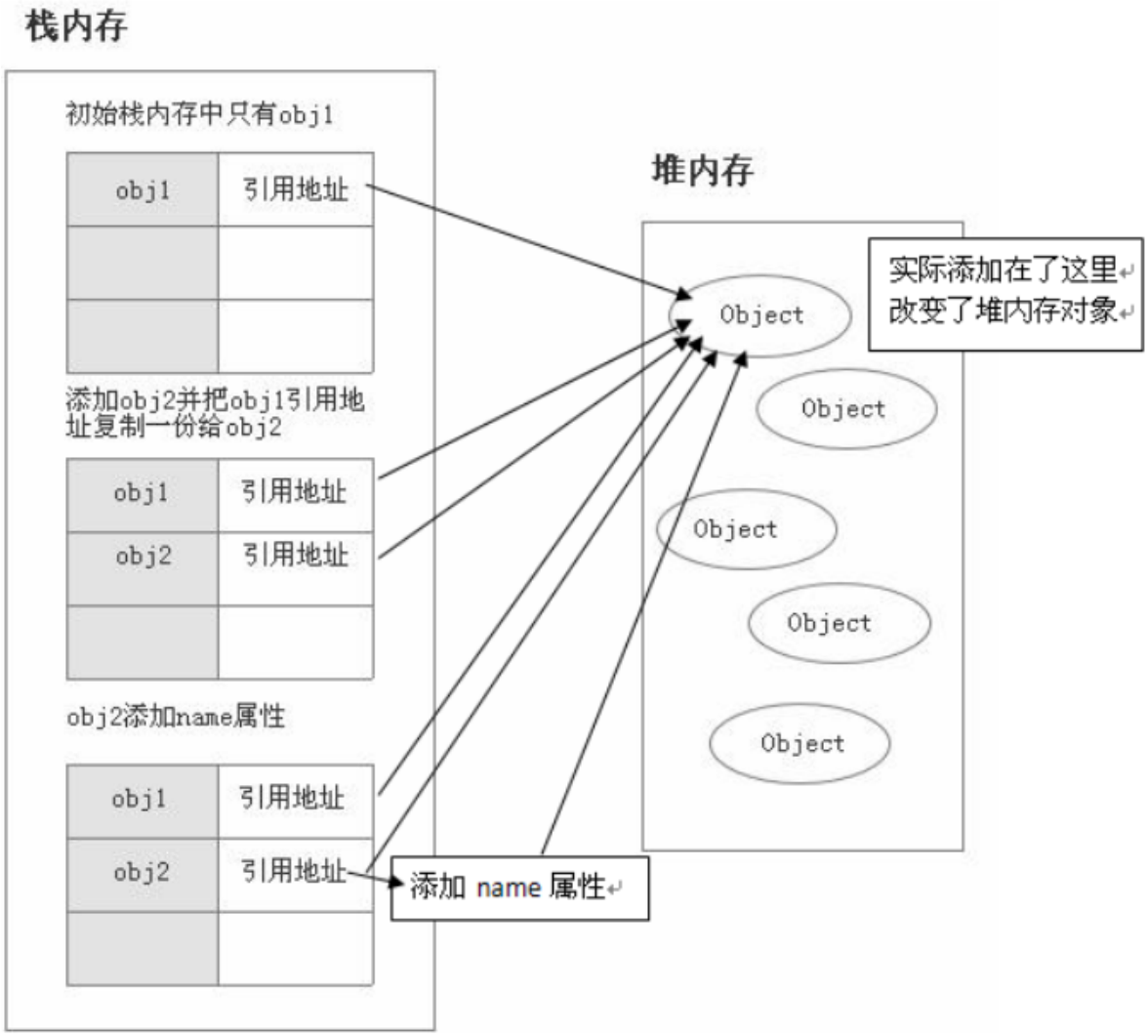
引用类型

```
var obj1 = {}
var obj2 = obj1;
obj2.name = "Xxx";
console.log(obj1.name); // xxx
```

引用类型数据存放在内对内中，每个堆内存中有一个引用地址，该引用地址存放在栈中

`obj1`是一个引用类型，在赋值操作过程汇总，实际是将堆内存对象在栈内存的引用地址复制了一份给了`obj2`，实际上他们共同指向了同一个堆内存对象，所以更改`obj2`会对`obj1`产生影响

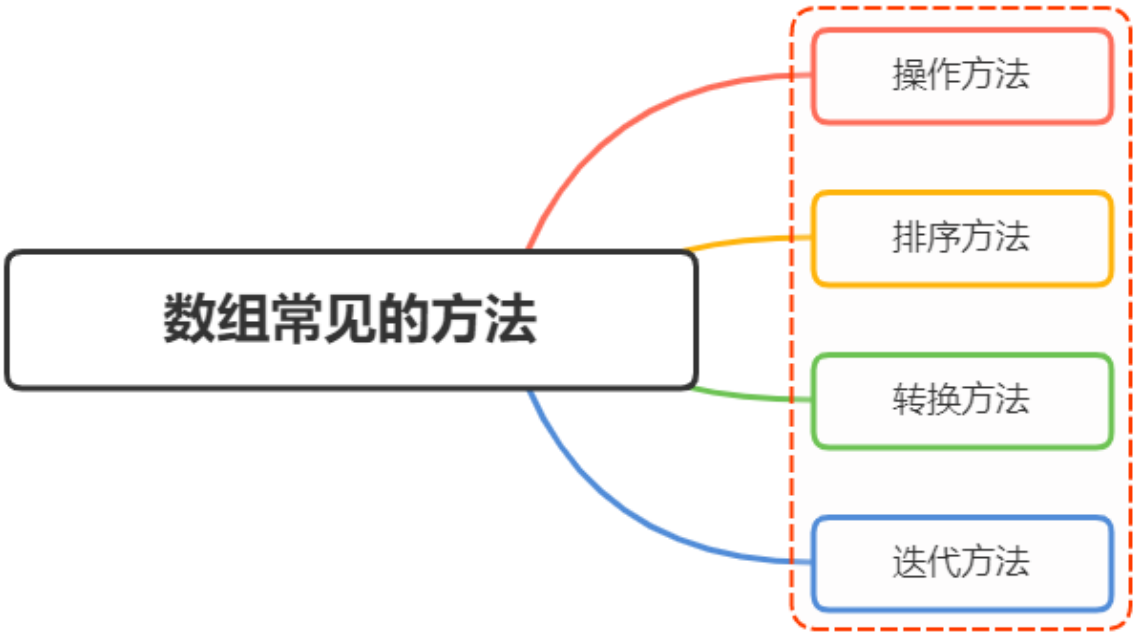
下图演示这个引用类型赋值过程



小结

- 声明变量时不同的内存地址分配：
  - 简单类型的值存放在栈中，在栈中存放的是对应的值
  - 引用类型对应的值存储在堆中，在栈中存放的是指向堆内存的地址
- 不同的类型数据导致赋值变量时的不同：
  - 简单类型赋值，是生成相同的值，两个对象对应不同的地址
  - 复杂类型赋值，是将保存对象的内存地址赋值给另一个变量。也就是两个变量指向堆内存中同一个对象

02.数组的常用方法有哪些?



一、操作方法

数组基本操作可以归纳为 增、删、改、查，需要留意的是哪些方法会对原数组产生影响，哪些方法不会

下面对数组常用的操作方法做一个归纳

增

下面前三种是对原数组产生影响的增添方法，第四种则不会对原数组产生影响

- push()
- unshift()
- splice()
- concat()

push()

`push()`方法接收任意数量的参数，并将它们添加到数组末尾，返回数组的最新长度

```
let colors = []; // 创建一个数组
let count = colors.push("red", "green"); // 推入两项
console.log(count) // 2
```

unshift()

unshift()在数组开头添加任意多个值，然后返回新的数组长度

```
let colors = new Array(); // 创建一个数组
let count = colors.unshift("red", "green"); // 从数组开头推入两项
alert(count); // 2
```

splice

传入三个参数，分别是开始位置、0（要删除的元素数量）、插入的元素，返回空数组

```
let colors = ["red", "green", "blue"];
let removed = colors.splice(1, 0, "yellow", "orange")
console.log(colors) // red,yellow,orange,green,blue
console.log(removed) // []
```

concat()

首先会创建一个当前数组的副本，然后再把它的参数添加到副本末尾，最后返回这个新构建的数组，不会影响原始数组

```
let colors = ["red", "green", "blue"];
let colors2 = colors.concat("yellow", ["black", "brown"]);
console.log(colors); // ["red", "green","blue"]
console.log(colors2); // ["red", "green", "blue", "yellow", "black", "brown"]
```

删

下面三种都会影响原数组，最后一项不影响原数组：

- pop()
- shift()
- splice()
- slice()

pop()

`pop()` 方法用于删除数组的最后一项，同时减少数组的 `length` 值，返回被删除的项

```
let colors = ["red", "green"]
let item = colors.pop(); // 取得最后一项
console.log(item) // green
console.log(colors.length) // 1
```

shift()

`shift()` 方法用于删除数组的第一项，同时减少数组的 `length` 值，返回被删除的项

```
let colors = ["red", "green"]
let item = colors.shift(); // 取得第一项
console.log(item) // red
console.log(colors.length) // 1
```

splice()

传入两个参数，分别是开始位置，删除元素的数量，返回包含删除元素的数组

```
let colors = ["red", "green", "blue"];
let removed = colors.splice(0,1); // 删除第一项
console.log(colors); // green,blue
console.log(removed); // red，只有一个元素的数组
```

slice()

slice() 用于创建一个包含原有数组中一个或多个元素的新数组，不会影响原始数组

```
let colors = ["red", "green", "blue", "yellow", "purple"];
let colors2 = colors.slice(1);
let colors3 = colors.slice(1, 4);
console.log(colors)   // red,green,blue,yellow,purple
concole.log(colors2); // green,blue,yellow,purple
concole.log(colors3); // green,blue,yellow
```

改

即修改原来数组的内容，常用`splice`

splice()

传入三个参数，分别是开始位置，要删除元素的数量，要插入的任意多个元素，返回删除元素的数组，对原数组产生影响

```
let colors = ["red", "green", "blue"];
let removed = colors.splice(1, 1, "red", "purple"); // 插入两个值，删除一个元素
console.log(colors); // red,red,purple,blue
console.log(removed); // green，只有一个元素的数组
```

查

即查找元素，返回元素坐标或者元素值

- indexOf()
- includes()
- find()

indexOf()

返回要查找的元素在数组中的位置，如果没找到则返回 -1

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
numbers.indexOf(4) // 3
```

includes()

返回要查找的元素在数组中的位置，找到返回`true`，否则`false`

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
numbers.includes(4) // true
```

find()

返回第一个匹配的元素

```
const people = [
    {
        name: "Matt",
        age: 27
    },
    {
        name: "Nicholas",
        age: 29
    }
];
people.find((element, index, array) => element.age < 28) // // {name: "Matt",
age: 27}
```

二、排序方法

数组有两个方法可以用来对元素重新排序：

- reverse()
- sort()

reverse()

顾名思义，将数组元素方向反转

```
let values = [1, 2, 3, 4, 5];
values.reverse();
alert(values); // 5,4,3,2,1
```

sort()

sort()方法接受一个比较函数，用于判断哪个值应该排在前面

```
function compare(value1, value2) {
    if (value1 < value2) {
        return -1;
    } else if (value1 > value2) {
        return 1;
    } else {
        return 0;
    }
}
let values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values); // 0,1,5,10,15
```

三、转换方法

常见的转换方法有：

join()

join() 方法接收一个参数，即字符串分隔符，返回包含所有项的字符串

```
let colors = ["red", "green", "blue"];
alert(colors.join(",")); // red,green,blue
alert(colors.join("||")); // red||green||blue
```

四、迭代方法

常用来迭代数组的方法（都不改变原数组）有如下：

- some()
- every()
- forEach()
- filter()
- map()

some()

对数组每一项都运行传入的函数，如果有一项函数返回 true ，则这个方法返回 true

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let someResult = numbers.every((item, index, array) => item > 2);
console.log(someResult) // true
```

every()

对数组每一项都运行传入的函数，如果对每一项函数都返回 true ，则这个方法返回 true

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let everyResult = numbers.every((item, index, array) => item > 2);
console.log(everyResult) // false
```

forEach()

对数组每一项都运行传入的函数，没有返回值

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
numbers.forEach((item, index, array) => {
    // 执行某些操作
});
```

filter()

对数组每一项都运行传入的函数，函数返回 `true` 的项会组成数组之后返回

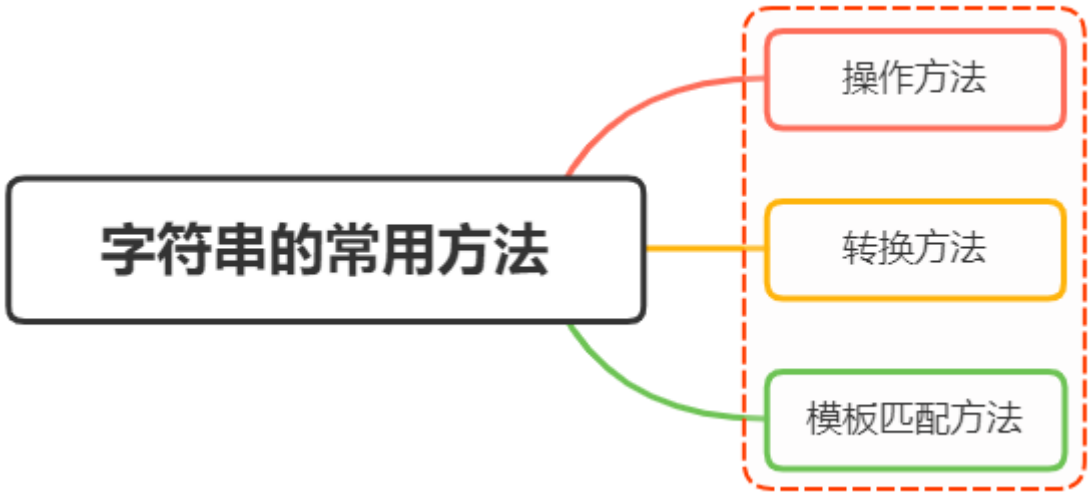
```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let filterResult = numbers.filter((item, index, array) => item > 2);
console.log(filterResult); // 3,4,5,4,3
```

map()

对数组每一项都运行传入的函数，返回由每次函数调用的结果构成的数组

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let mapResult = numbers.map((item, index, array) => item * 2);
console.log(mapResult) // 2,4,6,8,10,8,6,4,2
```

03.JavaScript字符串的常用方法有哪些?



一、操作方法

我们也可将字符串常用的操作方法归纳为增、删、改、查，需要知道字符串的特点是一旦创建了，就不可变

增

这里增的意思并不是说直接增添内容，而是创建字符串的一个副本，再进行操作

除了常用`+`以及`${}`进行字符串拼接之外，还可通过`concat`

concat

用于将一个或多个字符串拼接成一个新字符串

```
let stringValue = "hello ";
let result = stringValue.concat("world");
console.log(result); // "hello world"
console.log(stringValue); // "hello"
```

删

这里的删的意思并不是说删除原字符串的内容，而是创建字符串的一个副本，再进行操作

常见的有：

- slice()
- substr()
- substring()

这三个方法都返回调用它们的字符串的一个子字符串，而且都接收一或两个参数。

```
let stringValue = "hello world";
console.log(stringValue.slice(3)); // "lo world"
console.log(stringValue.substring(3)); // "lo world"
console.log(stringValue.substr(3)); // "lo world"
console.log(stringValue.slice(3, 7)); // "lo w"
console.log(stringValue.substring(3,7)); // "lo w"
console.log(stringValue.substr(3, 7)); // "lo worl"
```

改

这里改的意思也不是改变原字符串，而是创建字符串的一个副本，再进行操作

常见的有：

- trim()、trimLeft()、trimRight()
- repeat()
- padStart()、padEnd()
- toLowerCase()、 toUpperCase()

trim()、trimLeft()、trimRight()

删除前、后或前后所有空格符，再返回新的字符串

```
let stringValue = " hello world ";
let trimmedStringValue = stringValue.trim();
console.log(stringValue); // " hello world "
console.log(trimmedStringValue); // "hello world"
```

repeat()

接收一个整数参数，表示要将字符串复制多少次，然后返回拼接所有副本后的结果

```
let stringValue = "na ";
let copyResult = stringValue.repeat(2) // na na
```

padEnd()

复制字符串，如果小于指定长度，则在相应一边填充字符，直至满足长度条件

```
let stringValue = "foo";
console.log(stringValue.padStart(6)); // "  foo"
console.log(stringValue.padStart(9, ".")); // "......foo"
```

toLowerCase()、 toUpperCase()

大小写转化

```
let stringValue = "hello world";
console.log(stringValue.toUpperCase()); // "HELLO WORLD"
console.log(stringValue.toLowerCase()); // "hello world"
```

查

除了通过索引的方式获取字符串的值，还可通过：

- chatAt()
- indexOf()
- startWith()
- includes()

charAt()

返回给定索引位置的字符，由传给方法的整数参数指定

```
let message = "abcde";
console.log(message.charAt(2)); // "c"
```

indexOf()

从字符串开头去搜索传入的字符串，并返回位置（如果没找到，则返回 -1 ）

```
let stringValue = "hello world";
console.log(stringValue.indexOf("o")); // 4
```

startWith()、includes()

从字符串中搜索传入的字符串，并返回一个表示是否包含的布尔值

```
let message = "foobarbaz";
console.log(message.startsWith("foo")); // true
console.log(message.startsWith("bar")); // false
console.log(message.includes("bar")); // true
console.log(message.includes("qux")); // false
```

二、转换方法

split

把字符串按照指定的分割符，拆分成数组中的每一项

```
let str = "12+23+34"
let arr = str.split("+") // [12,23,34]
```

三、模板匹配方法

针对正则表达式，字符串设计了几个方法：

- match()
- search()
- replace()

match()

接收一个参数，可以是一个正则表达式字符串，也可以是一个`RegExp`对象，返回数组

```
let text = "cat, bat, sat, fat";
let pattern = /.at/;
let matches = text.match(pattern);
console.log(matches[0]); // "cat"
```

search()

接收一个参数，可以是一个正则表达式字符串，也可以是一个`RegExp`对象，找到则返回匹配索引，否则返回 -1

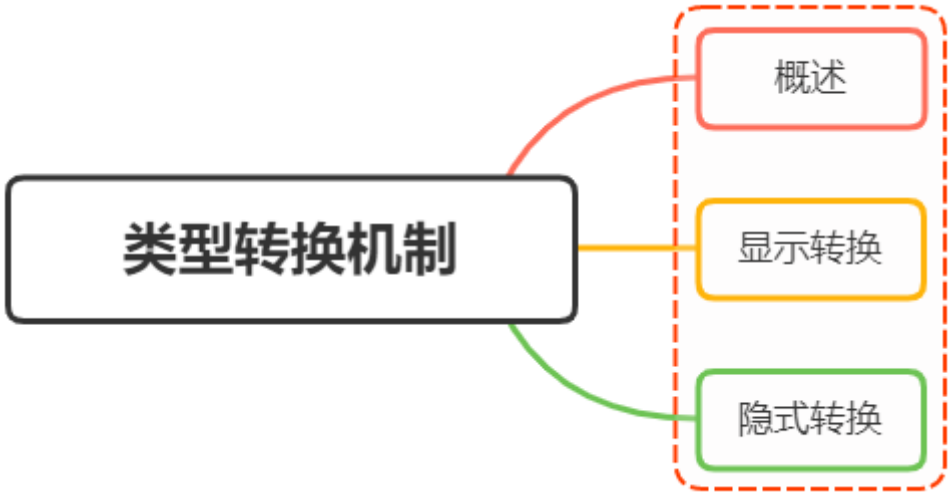
```
let text = "cat, bat, sat, fat";
let pos = text.search(/at/);
console.log(pos); // 1
```

replace()

接收两个参数，第一个参数为匹配的内容，第二个参数为替换的元素（可用函数）

```
let text = "cat, bat, sat, fat";
let result = text.replace("at", "ond");
console.log(result); // "cond, bat, sat, fat"
```

04.谈谈 JavaScript 中的类型转换机制



一、概述

前面我们讲到，`JS`中有六种简单数据类型：`undefined`、`null`、`boolean`、`string`、`number`、`symbol`，以及引用类型：`object`

但是我们在声明的时候只有一种数据类型，只有到运行期间才会确定当前类型

```
let x = y ? 1 : a;
```

上面代码中，`x`的值在编译阶段是无法获取的，只有等到程序运行时才能知道

虽然变量的数据类型是不确定的，但是各种运算符对数据类型是有要求的，如果运算子的类型与预期不符合，就会触发类型转换机制

常见的类型转换有：

- 强制转换（显示转换）
- 自动转换（隐式转换）

二、显示转换

显示转换，即我们很清楚可以看到这里发生了类型的转变，常见的方法有：

- Number()
- parseInt()
- String()
- Boolean()

Number()

将任意类型的值转化为数值

先给出类型转换规则：

原始值	转换结果
Undefined	NaN
Null	0
true	1
false	0
String	根据语法和转换规则来转换
Symbol	Throw a TypeError exception
Object	先调用toPrimitive，再调用toNumber

实践一下：

```
Number(324) // 324

// 字符串：如果可以被解析为数值，则转换为相应的数值
Number('324') // 324

// 字符串：如果不可以被解析为数值，返回 NaN
Number('324abc') // NaN

// 空字符串转为0
Number('') // 0

// 布尔值：true 转成 1，false 转成 0
Number(true) // 1
Number(false) // 0

// undefined：转成 NaN
Number(undefined) // NaN

// null：转成0
Number(null) // 0

// 对象：通常转换成NaN(除了只包含单个数值的数组)
Number({a: 1}) // NaN
Number([1, 2, 3]) // NaN
Number([5]) // 5
```

从上面可以看到，`Number`转换的时候是很严格的，只要有一个字符无法转成数值，整个字符串就会被转为`NaN`

parseInt()

`parseInt`相比`Number`，就没那么严格了，`parseInt`函数逐个解析字符，遇到不能转换的字符就停下来

```
parseInt('32a3') //32
```

String()

可以将任意类型的值转化成字符串

给出转换规则图：

原始值	转换结果
Undefined	'Undefined'
Boolean	'true' or 'false'
Number	对应的字符串类型
String	String
Symbol	Throw a TypeError exception
Object	先调用toPrimitive，再调用toNumber

实践一下：

```
// 数值：转为相应的字符串
String(1) // "1"

//字符串：转换后还是原来的值
String("a") // "a"

//布尔值：true转为字符串"true"，false转为字符串"false"
String(true) // "true"

//undefined：转为字符串"undefined"
String(undefined) // "undefined"

//null：转为字符串"null"
String(null) // "null"

//对象
String({a: 1}) // "[object Object]"
String([1, 2, 3]) // "1,2,3"
```

Boolean()

可以将任意类型的值转为布尔值，转换规则如下：

数据类型	转换为 true 的值	转换为 false 的值
Boolean	true	false
String	非空字符串	""（空字符串）
Number	非零数值（包括无穷值）	0、NaN（参见后面的相关内容）
Object	任意对象	null
Undefined	N/A（不存在）	undefined

实践一下：

```
Boolean(undefined) // false
Boolean(null) // false
Boolean(0) // false
Boolean(NaN) // false
Boolean('') // false
Boolean({}) // true
Boolean([]) // true
Boolean(new Boolean(false)) // true
```

三、隐式转换

在隐式转换中，我们可能最大的疑惑是：何时发生隐式转换?

我们这里可以归纳为两种情况发生隐式转换的场景：

- 比较运算（`==`、`!=`、`>`、`<`）、`if`、`while`需要布尔值地方
- 算术运算（`+`、`-`、`*`、`/`、`%`）

除了上面的场景，还要求运算符两边的操作数不是同一类型

自动转换为布尔值

在需要布尔值的地方，就会将非布尔值的参数自动转为布尔值，系统内部会调用`Boolean`函数

可以得出个小结：

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

除了上面几种会被转化成`false`，其他都换被转化成`true`

自动转换成字符串

遇到预期为字符串的地方，就会将非字符串的值自动转为字符串

具体规则是：先将复合类型的值转为原始类型的值，再将原始类型的值转为字符串

常发生在`+`运算中，一旦存在字符串，则会进行字符串拼接操作

```
'5' + 1 // '51'
'5' + true // "5true"
'5' + false // "5false"
'5' + {} // "5[object Object]"
'5' + [] // "5"
'5' + function (){} // "5function (){}"
'5' + undefined // "5undefined"
'5' + null // "5null"
```

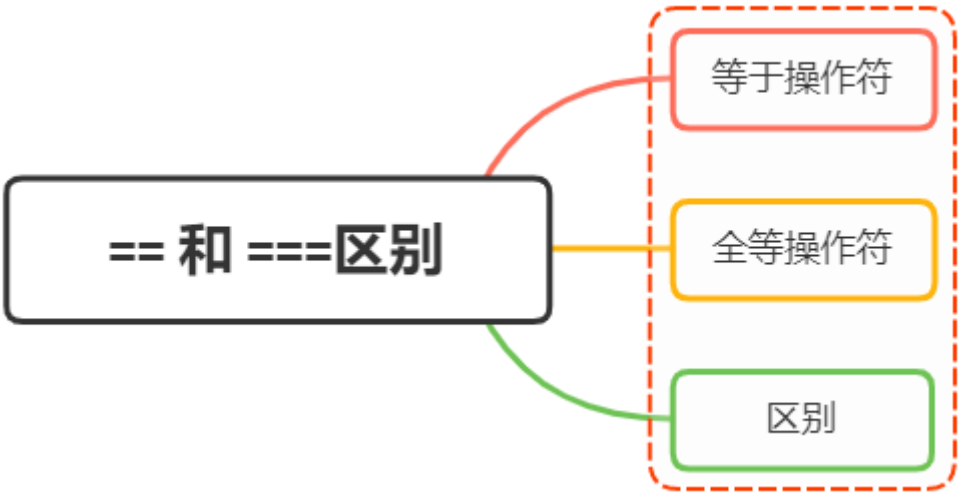
自动转换成数值

除了`+`有可能把运算子转为字符串，其他运算符都会把运算子自动转成数值

```
'5' - '2' // 3
'5' * '2' // 10
true - 1  // 0
false - 1 // -1
'1' - 1   // 0
'5' * []    // 0
false / '5' // 0
'abc' - 1   // NaN
null + 1 // 1
undefined + 1 // NaN
```

`null`转为数值时，值为`0`。`undefined`转为数值时，值为`NaN`

05.== 和 ===区别，分别在什么情况使用



一、等于操作符

等于操作符用两个等于号（==）表示，如果操作数相等，则会返回`true`

前面文章，我们提到在`JavaScript`中存在隐式转换。等于操作符（==）在比较中会先进行类型转换，再确定操作数是否相等

遵循以下规则：

如果任一操作数是布尔值，则将其转换为数值再比较是否相等

```
let result1 = (true == 1); // true
```

如果一个操作数是字符串，另一个操作数是数值，则尝试将字符串转换为数值，再比较是否相等

```
let result1 = ("55" == 55); // true
```

如果一个操作数是对象，另一个操作数不是，则调用对象的 `valueOf()` 方法取得其原始值，再根据前面的规则进行比较

```
let obj = {valueOf:function(){return 1}}
let result1 = (obj == 1); // true
```

`null` 和 `undefined` 相等

```
let result1 = (null == undefined ); // true
```

如果有任一操作数是 `NaN` ，则相等操作符返回 `false`

```
let result1 = (NaN == NaN ); // false
```

如果两个操作数都是对象，则比较它们是不是同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回 `true`

```
let obj1 = {name:"xxx"}
let obj2 = {name:"xxx"}
let result1 = (obj1 == obj2 ); // false
```

下面进一步做个小结：

- 两个都为简单类型，字符串和布尔值都会转换成数值，再比较
- 简单类型与引用类型比较，对象转化成其原始类型的值，再比较
- 两个都为引用类型，则比较它们是否指向同一个对象
- null 和 undefined 相等
- 存在 NaN 则返回 false

二、全等操作符

全等操作符由 3 个等于号（ === ）表示，只有两个操作数在不转换的前提下相等才返回 `true`。即类型相同，值也需相同

```
let result1 = ("55" === 55); // false，不相等，因为数据类型不同
let result2 = (55 === 55); // true，相等，因为数据类型相同值也相同
```

`undefined` 和 `null` 与自身严格相等

```
let result1 = (null === null)  //true
let result2 = (undefined === undefined)  //true
```

三、区别

相等操作符（==）会做类型转换，再进行值的比较，全等运算符不会做类型转换

```
let result1 = ("55" === 55); // false，不相等，因为数据类型不同
let result2 = (55 === 55); // true，相等，因为数据类型相同值也相同
```

`null` 和 `undefined` 比较，相等操作符（==）为`true`，全等为`false`

```
let result1 = (null == undefined ); // true
let result2 = (null  === undefined); // false
```

小结

相等运算符隐藏的类型转换，会带来一些违反直觉的结果

```
'' == '0' // false
0 == '' // true
0 == '0' // true

false == 'false' // false
false == '0' // true

false == undefined // false
false == null // false
null == undefined // true

' \t\r\n' == 0 // true
```

但在比较`null`的情况的时候，我们一般使用相等操作符`==`

```
const obj = {};

if(obj.x == null){
  console.log("1");  //执行
}
```

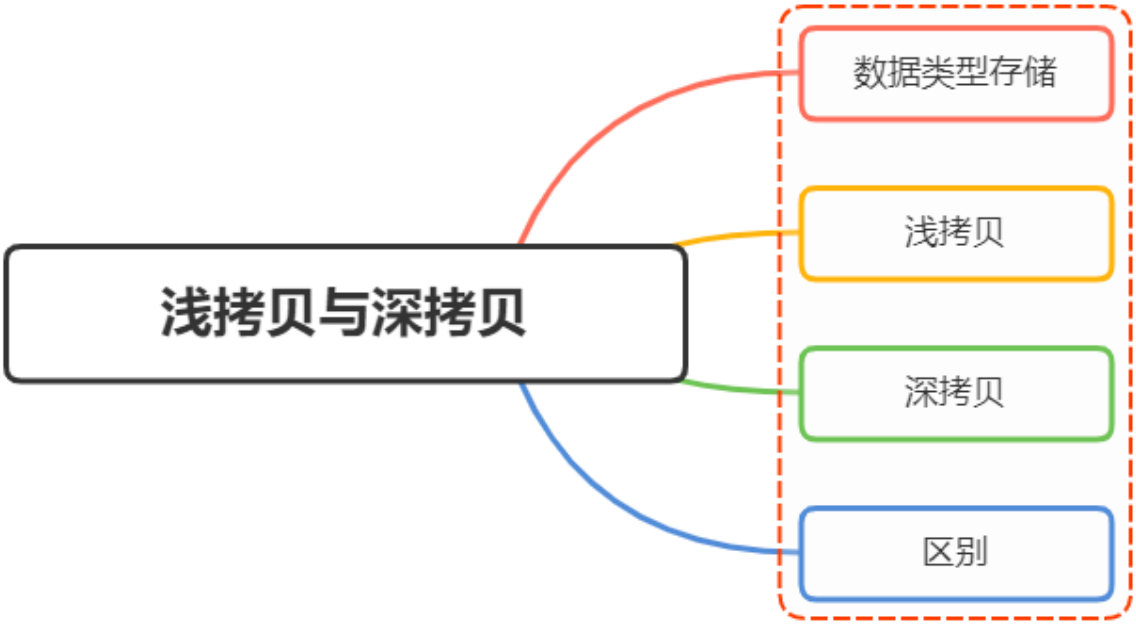
等同于下面写法

```
if(obj.x === null || obj.x === undefined) {
    ...
}
```

使用相等操作符（==）的写法明显更加简洁了

所以，除了在比较对象属性为`null`或者`undefined`的情况下，我们可以使用相等操作符（==），其他情况建议一律使用全等操作符（===）

06.深拷贝浅拷贝的区别？如何实现一个深拷贝？



一、数据类型存储

前面文章我们讲到，`JavaScript`中存在两大数据类型：

- 基本类型
- 引用类型

基本类型数据保存在在栈内存中

引用类型数据保存在堆内存中，引用数据类型的变量是一个指向堆内存中实际对象的引用，存在栈中

二、浅拷贝

浅拷贝，指的是创建新的数据，这个数据有着原始数据属性值的一份精确拷贝

如果属性是基本类型，拷贝的就是基本类型的值。如果属性是引用类型，拷贝的就是内存地址

即浅拷贝是拷贝一层，深层次的引用类型则共享内存地址

下面简单实现一个浅拷贝

```
function shallowClone(obj) {
    const newObj = {};
    for(let prop in obj) {
        if(obj.hasOwnProperty(prop)){
            newObj[prop] = obj[prop];
        }
    }
    return newObj;
}
```

在`JavaScript`中，存在浅拷贝的现象有：

- `Object.assign`
- `Array.prototype.slice()`，`Array.prototype.concat()`
- 使用拓展运算符实现的复制

Object.assign

```
var obj = {
    age: 18,
    nature: ['smart', 'good'],
    names: {
        name1: 'fx',
        name2: 'xka'
    },
    love: function () {
        console.log('fx is a great girl')
    }
}
var newObj = Object.assign({}, fxObj);
```

slice()

```
const fxArr = ["One", "Two", "Three"]
const fxArrs = fxArr.slice(0)
fxArrs[1] = "love";
console.log(fxArr) // ["One", "Two", "Three"]
console.log(fxArrs) // ["One", "love", "Three"]
```

concat()

```
const fxArr = ["One", "Two", "Three"]
const fxArrs = fxArr.concat()
fxArrs[1] = "love";
console.log(fxArr) // ["One", "Two", "Three"]
console.log(fxArrs) // ["One", "love", "Three"]
```

拓展运算符

```
const fxArr = ["One", "Two", "Three"]
const fxArrs = [...fxArr]
fxArrs[1] = "love";
console.log(fxArr) // ["One", "Two", "Three"]
console.log(fxArrs) // ["One", "love", "Three"]
```

三、深拷贝

深拷贝开辟一个新的栈，两个对象属完成相同，但是对应两个不同的地址，修改一个对象的属性，不会改变另一个对象的属性

常见的深拷贝方式有：

- _.cloneDeep()
- jQuery.extend()
- JSON.stringify()
- 手写循环递归

_.cloneDeep()

```
const _ = require('lodash');
const obj1 = {
    a: 1,
    b: { f: { g: 1 } },
    c: [1, 2, 3]
};
const obj2 = _.cloneDeep(obj1);
console.log(obj1.b.f === obj2.b.f);// false
```

jQuery.extend()

```
const $ = require('jquery');
const obj1 = {
    a: 1,
    b: { f: { g: 1 } },
    c: [1, 2, 3]
};
const obj2 = $.extend(true, {}, obj1);
console.log(obj1.b.f === obj2.b.f); // false
```

JSON.stringify()

```
const obj2=JSON.parse(JSON.stringify(obj1));
```

但是这种方式存在弊端，会忽略`undefined`、`symbol`和`函数`

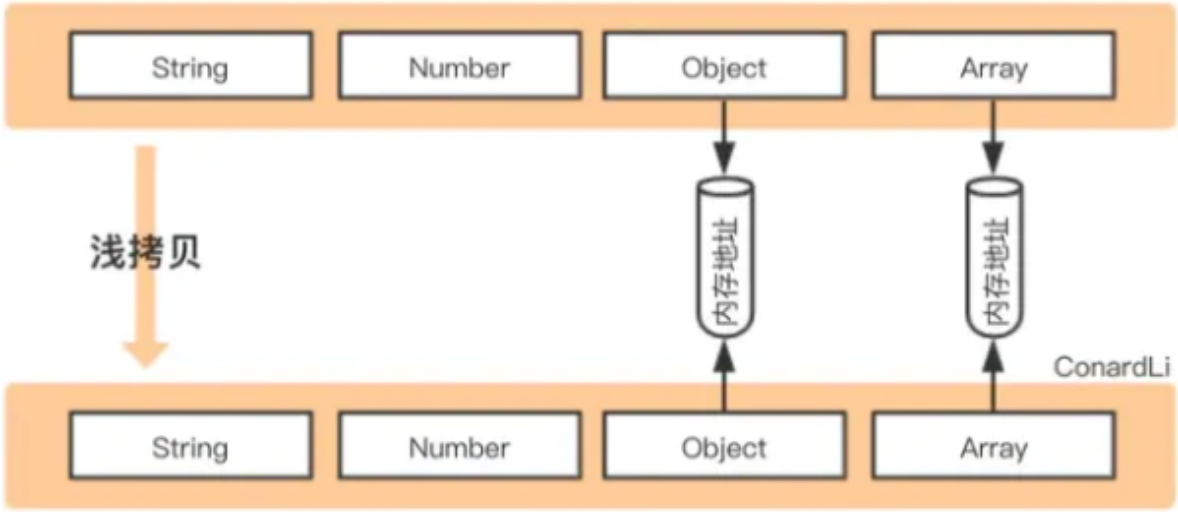
```
const obj = {
    name: 'A',
    name1: undefined,
    name3: function() {},
    name4:  Symbol('A')
}
const obj2 = JSON.parse(JSON.stringify(obj));
console.log(obj2); // {name: "A"}
```

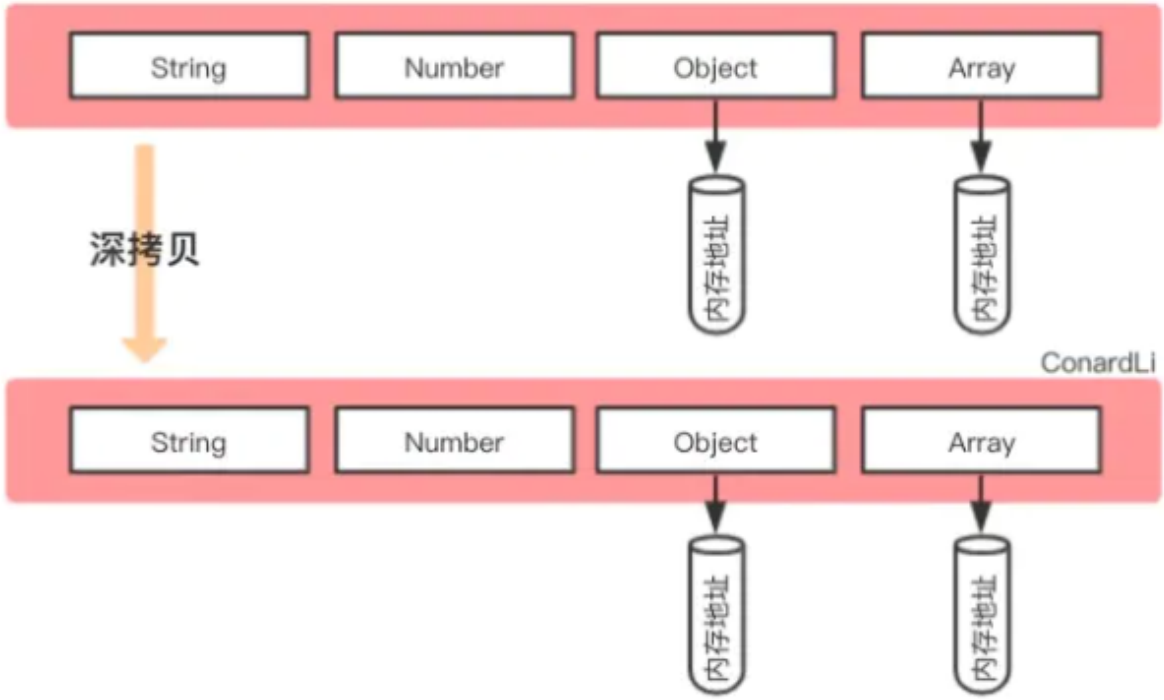
循环递归

```
function deepClone(obj, hash = new WeakMap()) {
  if (obj === null) return obj; // 如果是null或者undefined我就不进行拷贝操作
  if (obj instanceof Date) return new Date(obj);
  if (obj instanceof RegExp) return new RegExp(obj);
  // 可能是对象或者普通的值  如果是函数的话是不需要深拷贝
  if (typeof obj !== "object") return obj;
  // 是对象的话就要进行深拷贝
  if (hash.get(obj)) return hash.get(obj);
  let cloneObj = new obj.constructor();
  // 找到的是所属类原型上的constructor,而原型上的 constructor指向的是当前类本身
  hash.set(obj, cloneObj);
  for (let key in obj) {
    if (obj.hasOwnProperty(key)) {
      // 实现一个递归拷贝
      cloneObj[key] = deepClone(obj[key], hash);
    }
  }
  return cloneObj;
}
```

四、区别

下面首先借助两张图，可以更加清晰看到浅拷贝与深拷贝的区别





从上图发现，浅拷贝和深拷贝都创建出一个新的对象，但在复制对象属性的时候，行为就不一样

浅拷贝只复制属性指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存，修改对象属性会影响原对象

```
// 浅拷贝
const obj1 = {
    name : 'init',
    arr : [1,[2,3],4],
};
const obj3=shallowClone(obj1) // 一个浅拷贝方法
obj3.name = "update";
obj3.arr[1] = [5,6,7] ; // 新旧对象还是共享同一块内存

console.log('obj1',obj1) // obj1 { name: 'init',  arr: [ 1, [ 5, 6, 7 ], 4 ] }
console.log('obj3',obj3) // obj3 { name: 'update', arr: [ 1, [ 5, 6, 7 ], 4 ] }
```

但深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象

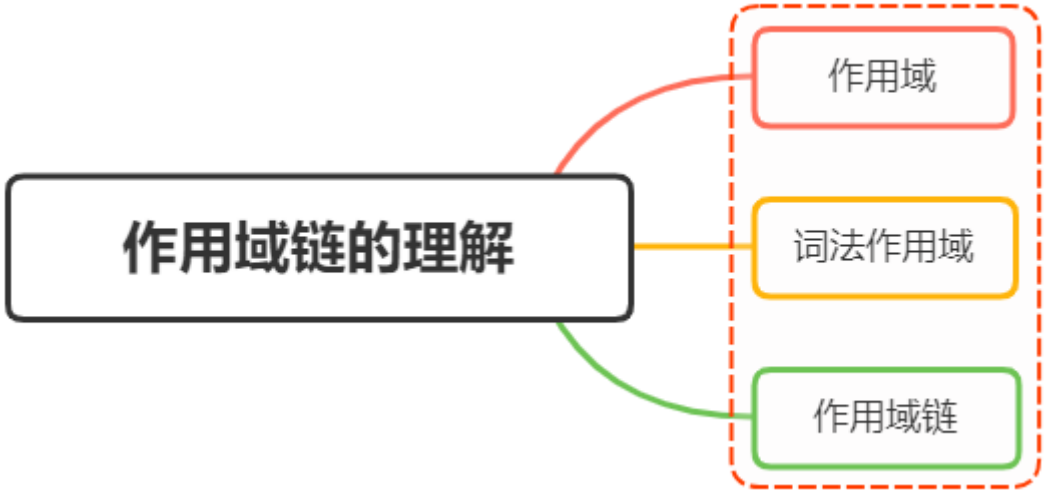
```
// 深拷贝
const obj1 = {
    name : 'init',
    arr : [1,[2,3],4],
};
const obj4=deepClone(obj1) // 一个深拷贝方法
obj4.name = "update";
obj4.arr[1] = [5,6,7] ; // 新对象跟原对象不共享内存

console.log('obj1',obj1) // obj1 { name: 'init', arr: [ 1, [ 2, 3 ], 4 ] }
console.log('obj4',obj4) // obj4 { name: 'update', arr: [ 1, [ 5, 6, 7 ], 4 ] }
```

小结

前提为拷贝类型为引用类型的情况下：

- 浅拷贝是拷贝一层，属性为对象时，浅拷贝是复制，两个对象指向同一个地址
- 深拷贝是递归拷贝深层次，属性为对象时，深拷贝是新开栈，两个对象指向不同的地址

07.说说你对作用域链的理解



一、作用域

作用域，即变量（变量作用域又称上下文）和函数生效（能被访问）的区域或集合

换句话说，作用域决定了代码区块中变量和其他资源的可见性

举个例子

```
function myFunction() {
    let inVariable = "函数内部变量";
}
myFunction();//要先执行这个函数，否则根本不知道里面是啥
console.log(inVariable); // Uncaught ReferenceError: inVariable is not defined
```

上述例子中，函数`myFunction`内部创建一个`inVariable`变量，当我们在全局访问这个变量的时候，系统会报错

这就说明我们在全局是无法获取到（闭包除外）函数内部的变量

我们一般将作用域分成：

- 全局作用域
- 函数作用域
- 块级作用域

全局作用域

任何不在函数中或是大括号中声明的变量，都是在全局作用域下，全局作用域下声明的变量可以在程序的任意位置访问

```
// 全局变量
var greeting = 'Hello World!';
function greet() {
  console.log(greeting);
}
// 打印 'Hello World!'
greet();
```

函数作用域

函数作用域也叫局部作用域，如果一个变量是在函数内部声明的它就在一个函数作用域下面。这些变量只能在函数内部访问，不能在函数以外去访问

```
function greet() {
  var greeting = 'Hello World!';
  console.log(greeting);
}
// 打印 'Hello World!'
greet();
// 报错： Uncaught ReferenceError: greeting is not defined
console.log(greeting);
```

可见上述代码中在函数内部声明的变量或函数，在函数外部是无法访问的，这说明在函数内部定义的变量或者方法只是函数作用域

块级作用域

ES6引入了`let`和`const`关键字,和`var`关键字不同，在大括号中使用`let`和`const`声明的变量存在于块级作用域中。在大括号之外不能访问这些变量

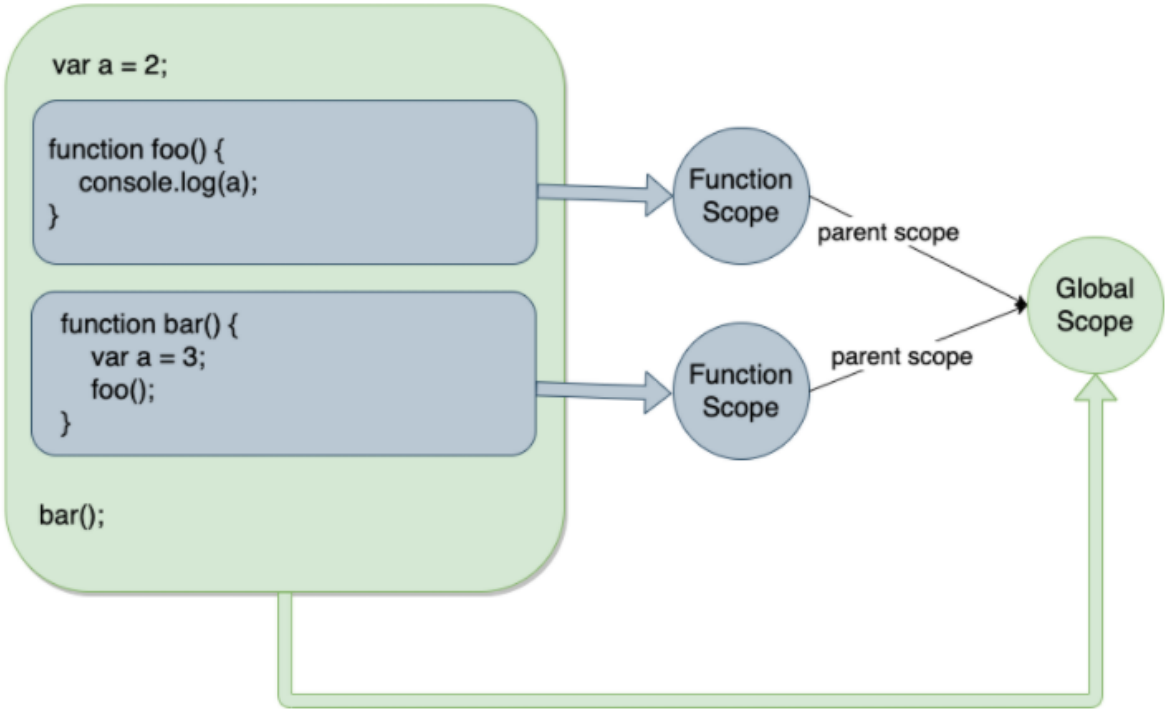
```
{
  // 块级作用域中的变量
  let greeting = 'Hello World!';
  var lang = 'English';
  console.log(greeting); // Prints 'Hello World!'
}
// 变量 'English'
console.log(lang);
// 报错：Uncaught ReferenceError: greeting is not defined
console.log(greeting);
```

二、词法作用域

词法作用域，又叫静态作用域，变量被创建时就确定好了，而非执行阶段确定的。也就是说我们写好代码时它的作用域就确定了，`JavaScript` 遵循的就是词法作用域

```
var a = 2;
function foo(){
    console.log(a)
}
function bar(){
    var a = 3;
    foo();
}
n()
```

上述代码改变成一张图



由于`JavaScript`遵循词法作用域，相同层级的 `foo` 和 `bar` 就没有办法访问到彼此块作用域中的变量，所以输出2

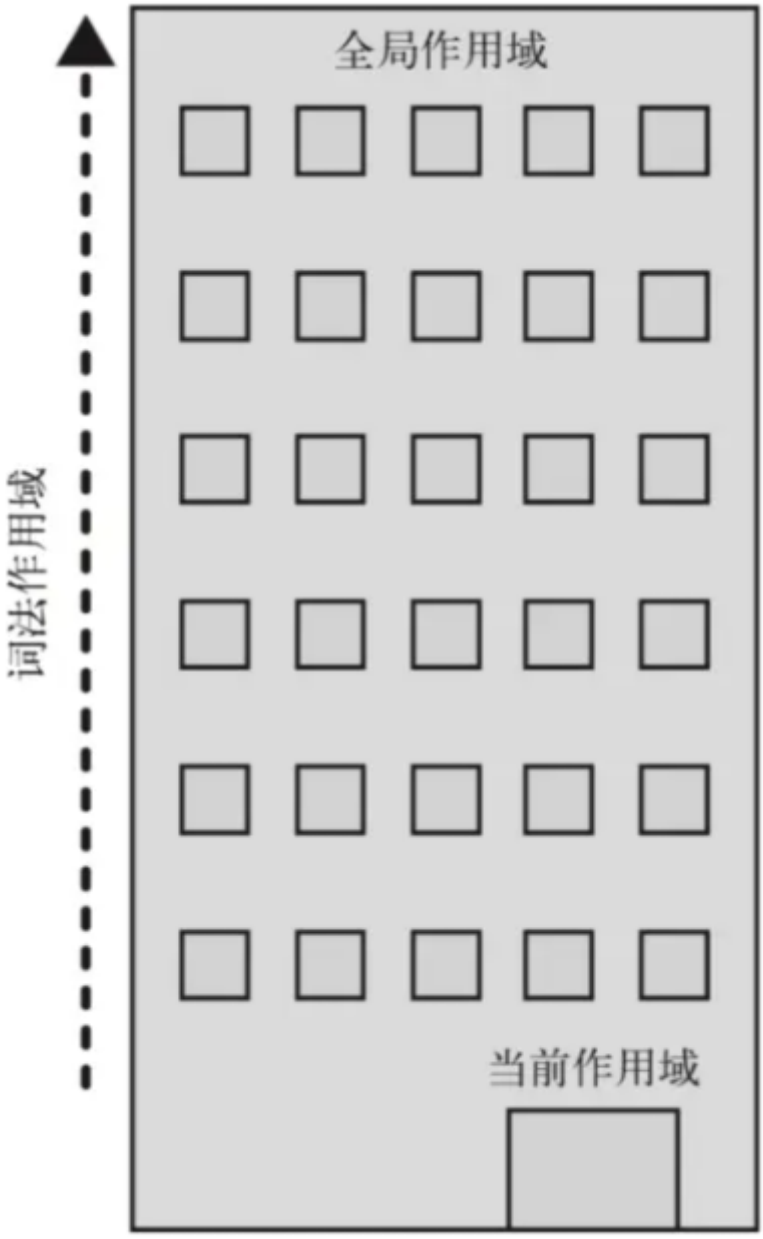
三、作用域链

当在`Javascript`中使用一个变量的时候，首先`Javascript`引擎会尝试在当前作用域下去寻找该变量，如果没找到，再到它的上层作用域寻找，以此类推直到找到该变量或是已经到了全局作用域

如果在全局作用域里仍然找不到该变量，它就会在全局范围内隐式声明该变量(非严格模式下)或是直接报错

这里拿《你不知道的Javascript(上)》中的一张图解释：

把作用域比喻成一个建筑，这份建筑代表程序中的嵌套作用域链，第一层代表当前的执行作用域，顶层代表全局作用域



变量的引用会顺着当前楼层进行查找，如果找不到，则会往上一层找，一旦到达顶层，查找的过程都会停止

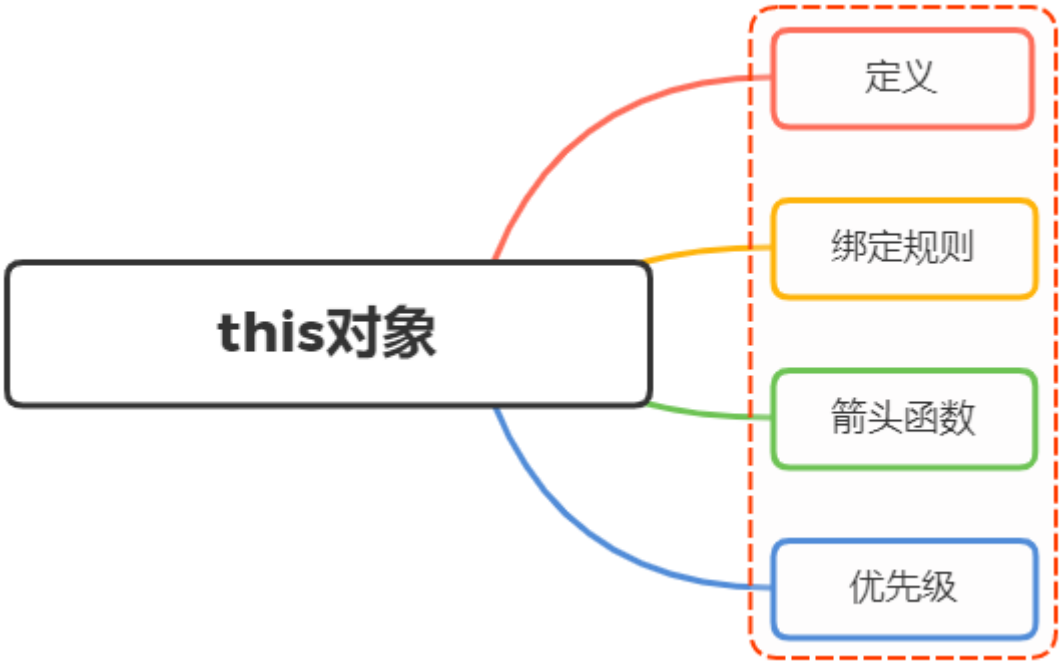
下面代码演示下：

```
var sex = '男';
function person() {
    var name = '张三';
    function student() {
        var age = 18;
        console.log(name); // 张三
        console.log(sex); // 男
    }
    student();
    console.log(age); // Uncaught ReferenceError: age is not defined
}
person();
```

上述代码主要主要做了以下工作：

- `student`函数内部属于最内层作用域，找不到`name`，向上一层作用域`person`函数内部找，找到了输出“张三”
- `student`内部输出cat时找不到，向上一层作用域`person`函数找，还找不到继续向上一层找，即全局作用域，找到了输出“男”
- 在`person`函数内部输出`age`时找不到，向上一层作用域找，即全局作用域，还是找不到则报错

08.谈谈this对象的理解



一、定义

函数的`this`关键字在`JavaScript`中的表现略有不同，此外，在严格模式和非严格模式之间也会有一些差别

在绝大多数情况下，函数的调用方式决定了`this`的值（运行时绑定）

`this`关键字是函数运行时自动生成的一个内部对象，只能在函数内部使用，总指向调用它的对象

举个例子：

```
function baz() {
    // 当前调用栈是：baz
    // 因此，当前调用位置是全局作用域

    console.log( "baz" );
    bar(); // <-- bar的调用位置
}

function bar() {
    // 当前调用栈是：baz --> bar
    // 因此，当前调用位置在baz中

    console.log( "bar" );
    foo(); // <-- foo的调用位置
}

function foo() {
    // 当前调用栈是：baz --> bar --> foo
    // 因此，当前调用位置在bar中

    console.log( "foo" );
}

baz(); // <-- baz的调用位置
```

同时，`this`在函数执行过程中，`this`一旦被确定了，就不可以再更改

```
var a = 10;
var obj = {
  a: 20
}

function fn() {
  this = obj; // 修改this，运行后会报错
  console.log(this.a);
}

fn();
```

二、绑定规则

根据不同的使用场合，`this`有不同的值，主要分为下面几种情况：

- 默认绑定
- 隐式绑定
- new绑定
- 显示绑定

默认绑定

全局环境中定义`person`函数，内部使用`this`关键字

```
var name = 'Jenny';
function person() {
    return this.name;
}
console.log(person());  //Jenny
```

上述代码输出`Jenny`，原因是调用函数的对象在游览器中位`window`，因此`this`指向`window`，所以输出`Jenny`

注意：

严格模式下，不能将全局对象用于默认绑定，this会绑定到`undefined`，只有函数运行在非严格模式下，默认绑定才能绑定到全局对象

隐式绑定

函数还可以作为某个对象的方法调用，这时`this`就指这个上级对象

```
function test() {
  console.log(this.x);
}

var obj = {};
obj.x = 1;
obj.m = test;

obj.m(); // 1
```

这个函数中包含多个对象，尽管这个函数是被最外层的对象所调用，`this`指向的也只是它上一级的对象

```
var o = {
    a:10,
    b:{
        fn:function(){
            console.log(this.a); //undefined
        }
    }
}
o.b.fn();
```

上述代码中，`this`的上一级对象为`b`，`b`内部并没有`a`变量的定义，所以输出`undefined`

这里再举一种特殊情况

```
var o = {
    a:10,
    b:{
        a:12,
        fn:function(){
            console.log(this.a); //undefined
            console.log(this); //window
        }
    }
}
var j = o.b.fn;
j();
```

此时`this`指向的是`window`，这里的大家需要记住，`this`永远指向的是最后调用它的对象，虽然`fn`是对象`b`的方法，但是`fn`赋值给`j`时候并没有执行，所以最终指向`window`

new绑定

通过构建函数`new`关键字生成一个实例对象，此时`this`指向这个实例对象

```
function test() {
  this.x = 1;
}

var obj = new test();
obj.x // 1
```

上述代码之所以能过输出1，是因为`new`关键字改变了`this`的指向

这里再列举一些特殊情况：

`new`过程遇到`return`一个对象，此时`this`指向为返回的对象

```
function fn()
{
    this.user = 'xxx';
    return {};
}
var a = new fn();
console.log(a.user); //undefined
```

如果返回一个简单类型的时候，则`this`指向实例对象

```
function fn()
{
    this.user = 'xxx';
    return 1;
}
var a = new fn;
console.log(a.user); //xxx
```

注意的是`null`虽然也是对象，但是此时`new`仍然指向实例对象

```
function fn()
{
    this.user = 'xxx';
    return null;
}
var a = new fn;
console.log(a.user); //xxx
```

显示修改

`apply()、call()、bind()`是函数的一个方法，作用是改变函数的调用对象。它的第一个参数就表示改变后的调用这个函数的对象。因此，这时`this`指的就是这第一个参数

```
var x = 0;
function test() {
  console.log(this.x);
}

var obj = {};
obj.x = 1;
obj.m = test;
obj.m.apply(obj) // 1
```

关于`apply、call、bind`三者的区别，我们后面再详细说

三、箭头函数

在 ES6 的语法中还提供了箭头函语法，让我们在代码书写时就能确定 `this` 的指向（编译时绑定）

举个例子：

```
const obj = {
  sayThis: () => {
    console.log(this);
  }
};

obj.sayThis(); // window 因为 JavaScript 没有块作用域，所以在定义 sayThis 的时候，里面的 this 就绑到 window 上去了
const globalSay = obj.sayThis;
globalSay(); // window 浏览器中的 global 对象
```

虽然箭头函数的`this`能够在编译的时候就确定了`this`的指向，但也需要注意一些潜在的坑

下面举个例子：

绑定事件监听

```
const button = document.getElementById('mngb');
button.addEventListener('click', ()=> {
    console.log(this === window) // true
    this.innerHTML = 'clicked button'
})
```

上述可以看到，我们其实是想要`this`为点击的`button`，但此时`this`指向了`window`

包括在原型上添加方法时候，此时`this`指向`window`

```
Cat.prototype.sayName = () => {
    console.log(this === window) //true
    return this.name
}
const cat = new Cat('mm');
cat.sayName()
```

同样的，箭头函数不能作为构建函数

四、优先级

隐式绑定 VS 显式绑定

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

var obj1 = {
    a: 2,
    foo: foo
};

var obj2 = {
    a: 3,
    foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

显然，显示绑定的优先级更高

new绑定 VS 隐式绑定

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

var obj1 = {
    foo: foo
};

var obj2 = {};

obj1.foo( 2 );
console.log( obj1.a ); // 2
```

```
obj1.foo.call( obj2, 3 );
console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );
console.log( obj1.a ); // 2
console.log( bar.a ); // 4
```

可以看到，new绑定的优先级 > 隐式绑定

new绑定 VS 显式绑定

因为new和apply、call无法一起使用，但硬绑定也是显式绑定的一种，可以替换测试

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

var obj1 = {};

var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2

var baz = new bar( 3 );
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

bar被绑定到obj1上，但是new bar(3) 并没有像我们预计的那样把obj1.a修改为3。但是，new修改了绑定调用bar()中的this

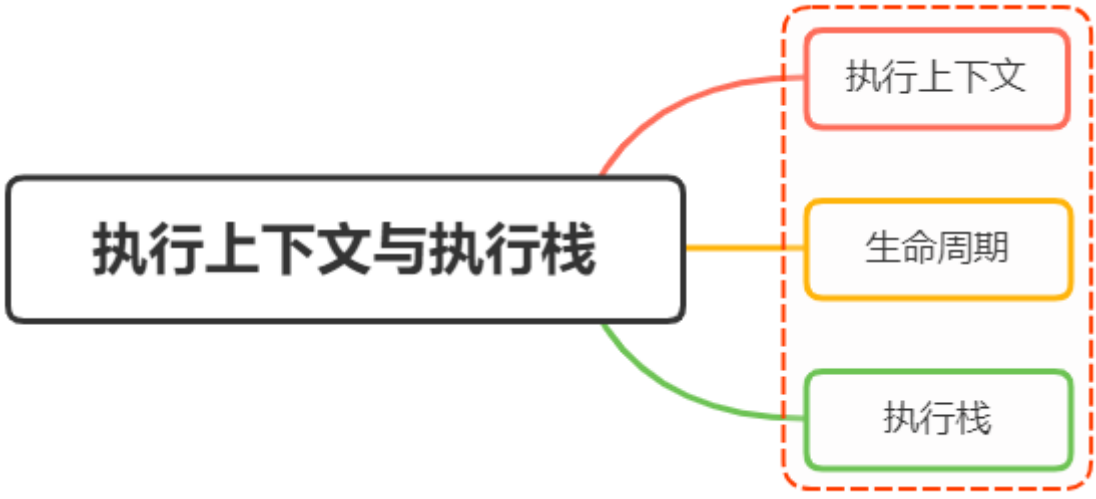
我们可认为new绑定优先级 > 显式绑定

综上，new绑定优先级 > 显示绑定优先级 > 隐式绑定优先级 > 默认绑定优先级

相关链接

- https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this

09.JavaScript中执行上下文和执行栈是什么



一、执行上下文

简单的来说，执行上下文是一种对`Javascript`代码执行环境的抽象概念，也就是说只要有`Javascript`代码运行，那么它就一定是运行在执行上下文中

执行上下文的类型分为三种：

- 全局执行上下文：只有一个，浏览器中的全局对象就是`window`对象，`this`指向这个全局对象
- 函数执行上下文：存在无数个，只有在函数被调用的时候才会被创建，每次调用函数都会创建一个新的执行上下文
- Eval 函数执行上下文： 指的是运行在`eval`函数中的代码，很少用而且不建议使用

下面给出全局上下文和函数上下文的例子：

```
// global context

var sayHello = 'Hello';

function person() {          // execution context

    var first = 'David',
        last  = 'Shariff';

    function firstName() {  // execution context
        return first;
    }

    function lastName() {   // execution context
        return last;
    }

    alert(sayHello + firstName() + ' ' + lastName());

}
```

紫色框住的部分为全局上下文，蓝色和橘色框起来的是不同的函数上下文。只有全局上下文（的变量）能被其他任何上下文访问

可以有任意多个函数上下文，每次调用函数创建一个新的上下文，会创建一个私有作用域，函数内部声明的任何变量都不能在当前函数作用域外部直接访问

二、生命周期

执行上下文的生命周期包括三个阶段：创建阶段 → 执行阶段 → 回收阶段

创建阶段

创建阶段即当函数被调用，但未执行任何其内部代码之前

创建阶段做了三件事：

- 确定 this 的值，也被称为`This Binding`
- LexicalEnvironment（词法环境） 组件被创建

- VariableEnvironment（变量环境） 组件被创建

伪代码如下：

```
ExecutionContext = {
  ThisBinding = <this value>,      // 确定this
  LexicalEnvironment = { ... },    // 词法环境
  VariableEnvironment = { ... },   // 变量环境
}
```

This Binding

确定`this`的值我们前面讲到，`this`的值是在执行的时候才能确认，定义的时候不能确认

词法环境

词法环境有两个组成部分：

- 全局环境：是一个没有外部环境的词法环境，其外部环境引用为`null`，有一个全局对象，`this`的值指向这个全局对象
- 函数环境：用户在函数中定义的变量被存储在环境记录中，包含了`arguments`对象，外部环境的引用可以是全局环境，也可以是包含内部函数的外部函数环境

伪代码如下：

```
GlobalExectionContext = {  // 全局执行上下文
  LexicalEnvironment: {        // 词法环境
    EnvironmentRecord: {      // 环境记录
      Type: "Object",            // 全局环境
      // 标识符绑定在这里
      outer: <null>             // 对外部环境的引用
  }
}

FunctionExectionContext = { // 函数执行上下文
  LexicalEnvironment: {      // 词法环境
    EnvironmentRecord: {     // 环境记录
      Type: "Declarative",       // 函数环境
      // 标识符绑定在这里       // 对外部环境的引用
      outer: <Global or outer function environment reference>
  }
}
```

变量环境

变量环境也是一个词法环境，因此它具有上面定义的词法环境的所有属性

在 ES6 中，词法环境和变量环境的区别在于前者用于存储函数声明和变量（`let`和`const`）绑定，而后者仅用于存储变量（`var`）绑定

举个例子

```
let a = 20;
const b = 30;
var c;

function multiply(e, f) {
 var g = 20;
 return e * f * g;
}

c = multiply(20, 30);
```

执行上下文如下：

```
GlobalExectionContext = {

  ThisBinding: <Global Object>,

  LexicalEnvironment: {  // 词法环境
    EnvironmentRecord: {
      Type: "Object",
      // 标识符绑定在这里
      a: < uninitialized >,
      b: < uninitialized >,
      multiply: < func >
    }
    outer: <null>
  },

  VariableEnvironment: {  // 变量环境
    EnvironmentRecord: {
      Type: "Object",
      // 标识符绑定在这里
      c: undefined,
    }
    outer: <null>
  }
}

FunctionExectionContext = {

  ThisBinding: <Global Object>,

  LexicalEnvironment: {
    EnvironmentRecord: {
      Type: "Declarative",
      // 标识符绑定在这里
      Arguments: {0: 20, 1: 30, length: 2},
    },
    outer: <GlobalLexicalEnvironment>
  },

  VariableEnvironment: {
    EnvironmentRecord: {
      Type: "Declarative",
      // 标识符绑定在这里
      g: undefined
```

```
    },
    outer: <GlobalLexicalEnvironment>
  }
}
```

留意上面的代码，`let`和`const`定义的变量`a`和`b`在创建阶段没有被赋值，但`var`声明的变量从在创建阶段被赋值为`undefined`

这是因为，创建阶段，会在代码中扫描变量和函数声明，然后将函数声明存储在环境中

但变量会被初始化为`undefined`(`var`声明的情况下)和保持`uninitialized`(未初始化状态)(使用`let`和`const`声明的情况下)

这就是变量提升的实际原因

执行阶段

在这阶段，执行变量赋值、代码执行

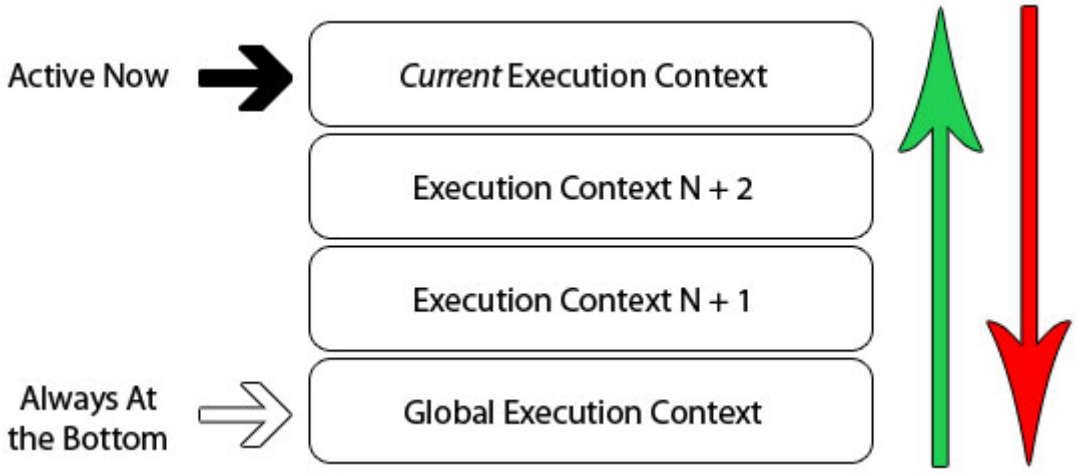
如果`Javascript`引擎在源代码中声明的实际位置找不到变量的值，那么将为其分配`undefined`值

回收阶段

执行上下文出栈等待虚拟机回收执行上下文

二、执行栈

执行栈，也叫调用栈，具有 LIFO（后进先出）结构，用于存储在代码执行期间创建的所有执行上下文



当`Javascript`引擎开始执行你第一行脚本代码的时候，它就会创建一个全局执行上下文然后将它压到执行栈中

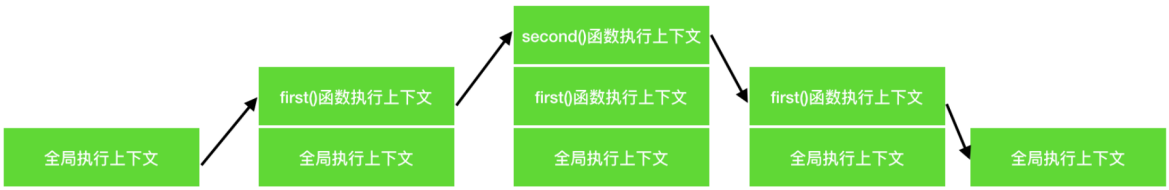
每当引擎碰到一个函数的时候，它就会创建一个函数执行上下文，然后将这个执行上下文压到执行栈中

引擎会执行位于执行栈栈顶的执行上下文(一般是函数执行上下文)，当该函数执行结束后，对应的执行上下文就会被弹出，然后控制流程到达执行栈的下一个执行上下文

举个例子：

```
let a = 'Hello World!';
function first() {
  console.log('Inside first function');
  second();
  console.log('Again inside first function');
}
function second() {
  console.log('Inside second function');
}
first();
console.log('Inside Global Execution Context');
```

转化成图的形式



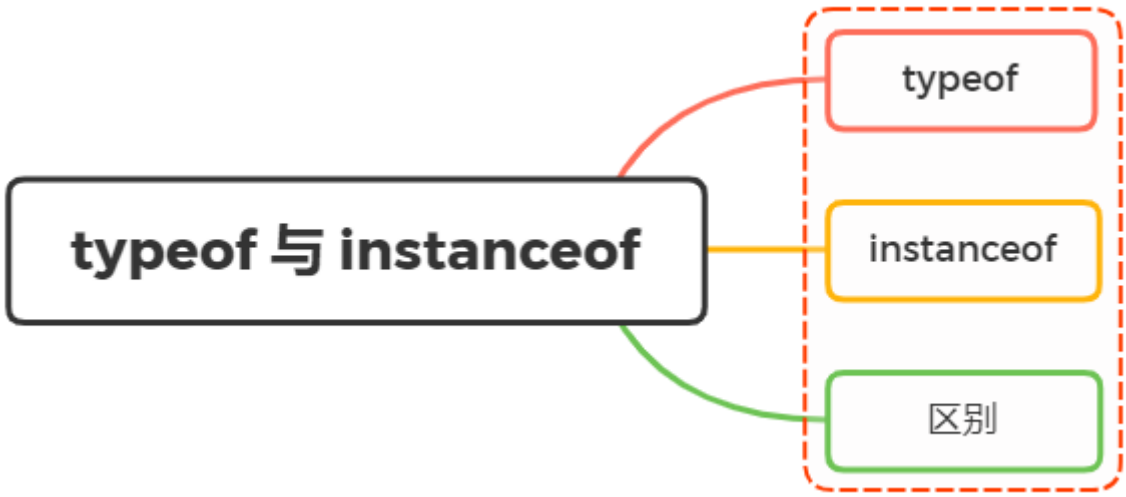
简单分析一下流程：

- 创建全局上下文请压入执行栈
- `first`函数被调用，创建函数执行上下文并压入栈
- 执行`first`函数过程遇到`second`函数，再创建一个函数执行上下文并压入栈
- `second`函数执行完毕，对应的函数执行上下文被推出执行栈，执行下一个执行上下文`first`函数
- `first`函数执行完毕，对应的函数执行上下文也被推出栈中，然后执行全局上下文
- 所有代码执行完毕，全局上下文也会被推出栈中，程序结束

参考文献

- https://zhuanlan.zhihu.com/p/107552264

10.typeof 与 instanceof 区别



一、typeof

`typeof` 操作符返回一个字符串，表示未经计算的操作数的类型

使用方法如下：

```
typeof operand
typeof(operand)
```

`operand`表示对象或原始值的表达式，其类型将被返回

举个例子

```
typeof 1 // 'number'
typeof '1' // 'string'
typeof undefined // 'undefined'
typeof true // 'boolean'
typeof Symbol() // 'symbol'
typeof null // 'object'
typeof [] // 'object'
typeof {} // 'object'
typeof console // 'object'
typeof console.log // 'function'
```

从上面例子，前6个都是基础数据类型。虽然`typeof null`为`object`，但这只是`JavaScript`存在的一个悠久`Bug`，不代表`null`就是引用数据类型，并且`null`本身也不是对象

所以，`null`在`typeof`之后返回的是有问题的结果，不能作为判断`null`的方法。如果你需要在`if`语句中判断是否为`null`，直接通过`===null`来判断就好

同时，可以发现引用类型数据，用`typeof`来判断的话，除了`function`会被识别出来之外，其余的都输出`object`

如果我们想要判断一个变量是否存在，可以使用`typeof`：(不能使用`if(a)`， 若`a`未声明，则报错)

```
if(typeof a != 'undefined'){
    //变量存在
}
```

二、instanceof

`instanceof`运算符用于检测构造函数的`prototype`属性是否出现在某个实例对象的原型链上

使用如下：

```
object instanceof constructor
```

`object`为实例对象，`constructor`为构造函数

构造函数通过`new`可以实例对象，`instanceof`能判断这个对象是否是之前那个构造函数生成的对象

```
// 定义构建函数
let Car = function() {}
let benz = new Car()
benz instanceof Car // true
let car = new String('xxx')
car instanceof String // true
let str = 'xxx'
str instanceof String // false
```

关于`instanceof`的实现原理，可以参考下面：

```
function myInstanceof(left, right) {
    // 这里先用typeof来判断基础数据类型，如果是，直接返回false
    if(typeof left !== 'object' || left === null) return false;
    // getProtypeOf是Object对象自带的API，能够拿到参数的原型对象
    let proto = Object.getPrototypeOf(left);
    while(true) {
        if(proto === null) return false;
        if(proto === right.prototype) return true;//找到相同原型对象，返回true
        proto = Object.getPrototypeof(proto);
    }
}
```

也就是顺着原型链去找，直到找到相同的原型对象，返回`true`，否则为`false`

三、区别

`typeof`与`instanceof`都是判断数据类型的方法，区别如下：

- `typeof`会返回一个变量的基本类型，`instanceof`返回的是一个布尔值
- `instanceof` 可以准确地判断复杂引用数据类型，但是不能正确判断基础数据类型
- 而`typeof` 也存在弊端，它虽然可以判断基础数据类型（`null` 除外），但是引用数据类型中，除了`function` 类型以外，其他的也无法判断

可以看到，上述两种方法都有弊端，并不能满足所有场景的需求

如果需要通用检测数据类型，可以采用`Object.prototype.toString`，调用该方法，统一返回格式`“[object Xxx]”`的字符串

如下

```
Object.prototype.toString({})       // "[object Object]"
Object.prototype.toString.call({})  // 同上结果，加上call也ok
Object.prototype.toString.call(1)    // "[object Number]"
Object.prototype.toString.call('1')  // "[object String]"
Object.prototype.toString.call(true)  // "[object Boolean]"
Object.prototype.toString.call(function(){})  // "[object Function]"
Object.prototype.toString.call(null)   //"[object Null]"
Object.prototype.toString.call(undefined) //"[object Undefined]"
Object.prototype.toString.call(/123/g)    //"[object RegExp]"
Object.prototype.toString.call(new Date()) //"[object Date]"
Object.prototype.toString.call([])       //"[object Array]"
Object.prototype.toString.call(document)  //"[object HTMLDocument]"
Object.prototype.toString.call(window)   //"[object Window]"
```

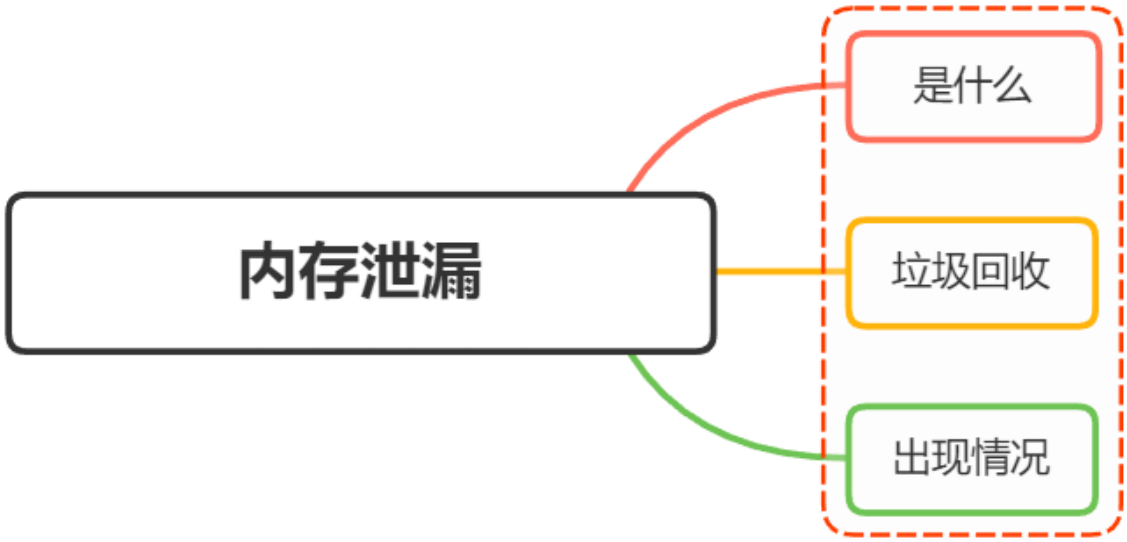
了解了`toString`的基本用法，下面就实现一个全局通用的数据类型判断方法

```
function getType(obj){
  let type  = typeof obj;
  if (type !== "object") {    // 先进行typeof判断，如果是基础数据类型，直接返回
    return type;
  }
  // 对于typeof返回结果是object的，再进行如下的判断，正则返回结果
  return Object.prototype.toString.call(obj).replace(/^\[object (\S+)\]$/,
'$1');
}
```

使用如下

```
getType([])     // "Array" typeof []是object，因此toString返回
getType('123')  // "string" typeof 直接返回
getType(window) // "Window" toString返回
getType(null)   // "Null"首字母大写，typeof null是object，需toString来判断
getType(undefined)   // "undefined" typeof 直接返回
getType()            // "undefined" typeof 直接返回
getType(function(){}) // "function" typeof能判断，因此首字母小写
getType(/123/g)      //"RegExp" toString返回
```

11.说说 JavaScript 中内存泄漏的几种情况



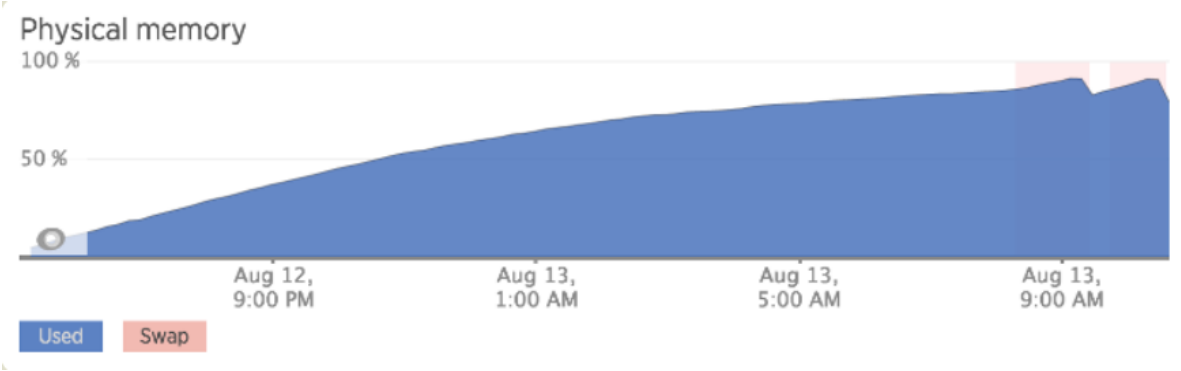
一、是什么

内存泄漏（Memory leak）是在计算机科学中，由于疏忽或错误造成程序未能释放已经不再使用的内存

并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费

程序的运行需要内存。只要程序提出要求，操作系统或者运行时就必须供给内存

对于持续运行的服务进程，必须及时释放不再用到的内存。否则，内存占用越来越高，轻则影响系统性能，重则导致进程崩溃



在`C`语言中，因为是手动管理内存，内存泄露是经常出现的事情。

```
char * buffer;
buffer = (char*) malloc(42);

// Do something with buffer

free(buffer);
```

上面是 C 语言代码，`malloc`方法用来申请内存，使用完毕之后，必须自己用`free`方法释放内存。

这很麻烦，所以大多数语言提供自动内存管理，减轻程序员的负担，这被称为"垃圾回收机制"

二、垃圾回收机制

Javascript 具有自动垃圾回收机制（GC：Garbage Collecation），也就是说，执行环境会负责管理代码执行过程中使用的内存

原理：垃圾收集器会定期（周期性）找出那些不在继续使用的变量，然后释放其内存

通常情况下有两种实现方式：

- 标记清除
- 引用计数

标记清除

`JavaScript`最常用的垃圾收回机制

当变量进入执行环境是，就标记这个变量为“进入环境“。进入环境的变量所占用的内存就不能释放，当变量离开环境时，则将其标记为“离开环境“

垃圾回收程序运行的时候，会标记内存中存储的所有变量。然后，它会将所有在上下文中的变量，以及被在上下文中的变量引用的变量的标记去掉

在此之后再被加上标记的变量就是待删除的了，原因是任何在上下文中的变量都访问不到它们了

随后垃圾回收程序做一次内存清理，销毁带标记的所有值并收回它们的内存

举个例子：

```
var m = 0,n = 19 // 把 m,n,add() 标记为进入环境。
add(m, n) // 把 a, b, c标记为进入环境。
console.log(n) // a,b,c标记为离开环境，等待垃圾回收。
function add(a, b) {
  a++
  var c = a + b
  return c
}
```

引用计数

语言引擎有一张"引用表"，保存了内存里面所有的资源（通常是各种值）的引用次数。如果一个值的引用次数是`0`，就表示这个值不再用到了，因此可以将这块内存释放

如果一个值不再需要了，引用数却不为`0`，垃圾回收机制无法释放这块内存，从而导致内存泄漏

```
const arr = [1, 2, 3, 4];
console.log('hello world');
```

面代码中，数组`[1, 2, 3, 4]`是一个值，会占用内存。变量`arr`是仅有的对这个值的引用，因此引用次数为`1`。尽管后面的代码没有用到`arr`，它还是会持续占用内存

如果需要这块内存被垃圾回收机制释放，只需要设置如下：

```
arr = null
```

通过设置`arr`为`null`，就解除了对数组`[1,2,3,4]`的引用，引用次数变为 0，就被垃圾回收了

小结

有了垃圾回收机制，不代表不用关注内存泄露。那些很占空间的值，一旦不再用到，需要检查是否还存在对它们的引用。如果是的话，就必须手动解除引用

三、常见内存泄露情况

意外的全局变量

```
function foo(arg) {
    bar = "this is a hidden global variable";
}
```

另一种意外的全局变量可能由`this`创建：

```
function foo() {
    this.variable = "potential accidental global";
}
// foo 调用自己，this 指向了全局对象（window）
foo();
```

上述使用严格模式，可以避免意外的全局变量

定时器也常会造成内存泄露

```
var someResource = getData();
setInterval(function() {
    var node = document.getElementById('Node');
    if(node) {
        // 处理 node 和 someResource
        node.innerHTML = JSON.stringify(someResource));
    }
}, 1000);
```

如果`id`为Node的元素从`DOM`中移除，该定时器仍会存在，同时，因为回调函数中包含对`someResource`的引用，定时器外面的`someResource`也不会被释放

包括我们之前所说的闭包，维持函数内局部变量，使其得不到释放

```
function bindEvent() {
  var obj = document.createElement('XXX');
  var unused = function () {
    console.log(obj, '闭包内引用obj obj不会被释放');
  };
  obj = null; // 解决方法
}
```

没有清理对`DOM`元素的引用同样造成内存泄露

```
const refA = document.getElementById('refA');
document.body.removeChild(refA); // dom删除了
console.log(refA, 'refA'); // 但是还存在引用能console出整个div 没有被回收
refA = null;
console.log(refA, 'refA'); // 解除引用
```

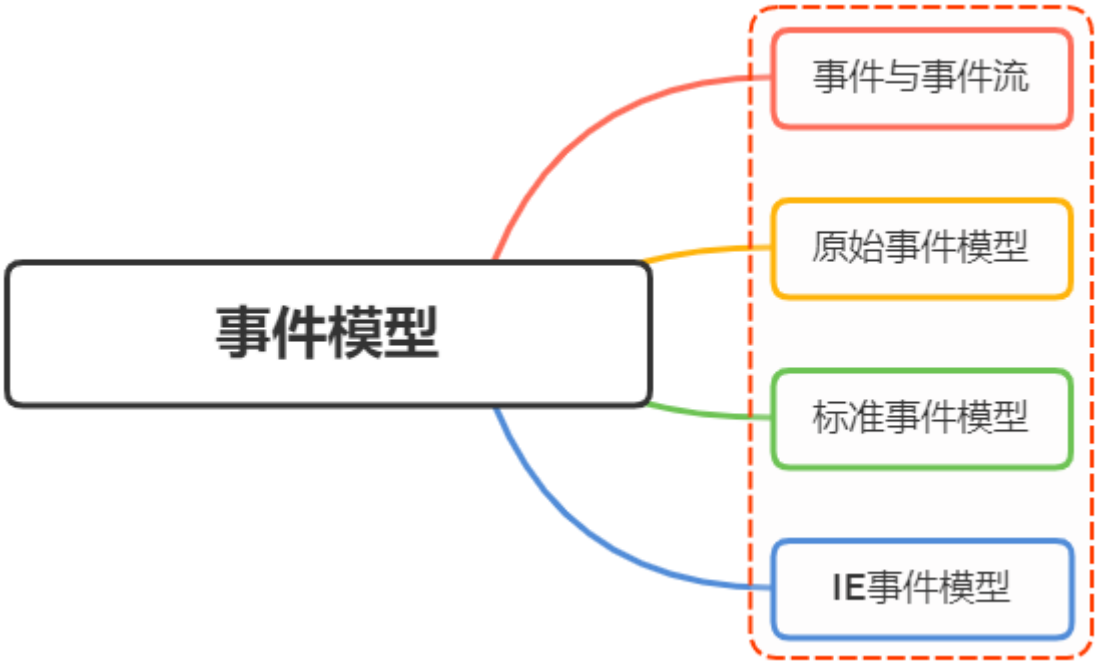
包括使用事件监听`addEventListener`监听的时候，在不监听的情况下使用`removeEventListener`取消对事件监听

参考文献

- http://www.ruanyifeng.com/blog/2017/04/memory-leak.html
- https://zh.wikipedia.org/wiki

4.webAPIs

01.说说JavaScript中的事件模型



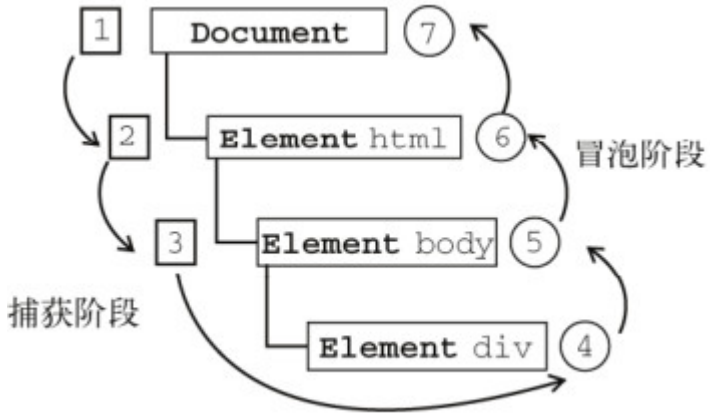
一、事件与事件流

`javascript`中的事件，可以理解就是在`HTML`文档或者浏览器中发生的一种交互操作，使得网页具备互动性， 常见的有加载事件、鼠标事件、自定义事件等

由于`DOM`是一个树结构，如果在父子节点绑定事件时候，当触发子节点的时候，就存在一个顺序问题，这就涉及到了事件流的概念

事件流都会经历三个阶段：

- 事件捕获阶段(capture phase)
- 处于目标阶段(target phase)
- 事件冒泡阶段(bubbling phase)



事件冒泡是一种从下往上的传播方式，由最具体的元素（触发节点）然后逐渐向上传播到最不具体的那个节点，也就是`DOM`中最高层的父节点

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Event Bubbling</title>
    </head>
    <body>
        <button id="clickMe">Click Me</button>
    </body>
</html>
```

然后，我们给`button`和它的父元素，加入点击事件

```
var button = document.getElementById('clickMe');

button.onclick = function() {
  console.log('1.Button');
};
document.body.onclick = function() {
  console.log('2.body');
};
document.onclick = function() {
  console.log('3.document');
};
window.onclick = function() {
  console.log('4.window');
};
```

点击按钮，输出如下

```
1.button
2.body
3.document
4.window
```

点击事件首先在`button`元素上发生，然后逐级向上传播

事件捕获与事件冒泡相反，事件最开始由不太具体的节点最早接受事件, 而最具体的节点（触发节点）最后接受事件

二、事件模型

事件模型可以分为三种：

- 原始事件模型（DOM0级）
- 标准事件模型（DOM2级）
- IE事件模型（基本不用）

原始事件模型

事件绑定监听函数比较简单, 有两种方式：

- HTML代码中直接绑定

```
<input type="button" onclick="fun()">
```

- 通过`JS`代码绑定

```
var btn = document.getElementById('.btn');
btn.onclick = fun;
```

特性

- 绑定速度快

`DOM0`级事件具有很好的跨浏览器优势，会以最快的速度绑定，但由于绑定速度太快，可能页面还未完全加载出来，以至于事件可能无法正常运行

- 只支持冒泡，不支持捕获
- 同一个类型的事件只能绑定一次

```
<input type="button" id="btn" onclick="fun1()">

var btn = document.getElementById('.btn');
btn.onclick = fun2;
```

如上，当希望为同一个元素绑定多个同类型事件的时候（上面的这个`btn`元素绑定2个点击事件），是不被允许的，后绑定的事件会覆盖之前的事件

删除`DOM0`级事件处理程序只要将对应事件属性置为`null`即可

```
btn.onclick = null;
```

标准事件模型

在该事件模型中，一次事件共有三个过程:

- 事件捕获阶段：事件从`document`一直向下传播到目标元素, 依次检查经过的节点是否绑定了事件监听函数，如果有则执行
- 事件处理阶段：事件到达目标元素, 触发目标元素的监听函数
- 事件冒泡阶段：事件从目标元素冒泡到`document`, 依次检查经过的节点是否绑定了事件监听函数，如果有则执行

事件绑定监听函数的方式如下:

```
addEventListener(eventType, handler, useCapture)
```

事件移除监听函数的方式如下:

```
removeEventListener(eventType, handler, useCapture)
```

参数如下:

- `eventType`指定事件类型(不要加on)
- `handler`是事件处理函数

- `useCapture`是一个`boolean`用于指定是否在捕获阶段进行处理，一般设置为`false`与IE浏览器保持一致

举个例子：

```
var btn = document.getElementById('.btn');
btn.addEventListener(‘click’, showMessage, false);
btn.removeEventListener(‘click’, showMessage, false);
```

特性

- 可以在一个`DOM`元素上绑定多个事件处理器，各自并不会冲突

```
btn.addEventListener(‘click’, showMessage1, false);
btn.addEventListener(‘click’, showMessage2, false);
btn.addEventListener(‘click’, showMessage3, false);
```

- 执行时机

当第三个参数(`useCapture`)设置为`true`就在捕获过程中执行，反之在冒泡过程中执行处理函数

下面举个例子：

```
<div id='div'>
    <p id='p'>
        <span id='span'>Click Me!</span>
    </p >
</div>
```

设置点击事件

```
var div = document.getElementById('div');
var p = document.getElementById('p');

function onClickFn (event) {
    var tagName = event.currentTarget.tagName;
    var phase = event.eventPhase;
    console.log(tagName, phase);
}

div.addEventListener('click', onClickFn, false);
p.addEventListener('click', onClickFn, false);
```

上述使用了`eventPhase`，返回一个代表当前执行阶段的整数值。1为捕获阶段、2为事件对象触发阶段、3为冒泡阶段

点击`Click Me!`，输出如下

```
P 3
DIV 3
```

可以看到，`p`和`div`都是在冒泡阶段响应了事件，由于冒泡的特性，裹在里层的`p`率先做出响应

如果把第三个参数都改为`true`

```
div.addEventListener('click', onClickFn, true);
p.addEventListener('click', onClickFn, true);
```

输出如下

```
DIV 1
P 1
```

两者都是在捕获阶段响应事件，所以`div`比`p`标签先做出响应

IE事件模型

IE事件模型共有两个过程:

- 事件处理阶段：事件到达目标元素, 触发目标元素的监听函数。
- 事件冒泡阶段：事件从目标元素冒泡到`document`，依次检查经过的节点是否绑定了事件监听函数，如果有则执行

事件绑定监听函数的方式如下:

```
attachEvent(eventType, handler)
```

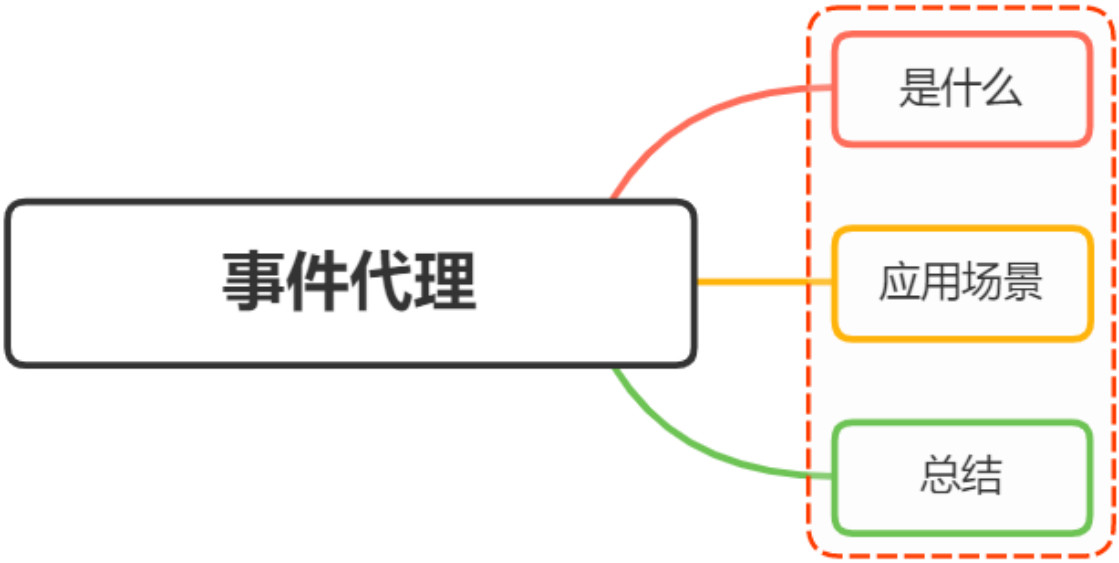
事件移除监听函数的方式如下:

```
detachEvent(eventType, handler)
```

举个例子:

```
var btn = document.getElementById('.btn');
btn.attachEvent('onclick', showMessage);
btn.detachEvent('onclick', showMessage);
```

02.解释下什么是事件代理？应用场景？



一、是什么

事件代理，俗地来讲，就是把一个元素响应事件（`click`、`keydown`.....）的函数委托到另一个元素

前面讲到，事件流的都会经过三个阶段： 捕获阶段 -> 目标阶段 -> 冒泡阶段，而事件委托就是在冒泡阶段完成

事件委托，会把一个或者一组元素的事件委托到它的父层或者更外层元素上，真正绑定事件的是外层元素，而不是目标元素

当事件响应到目标元素上时，会通过事件冒泡机制从而触发它的外层元素的绑定事件上，然后在外层元素上去执行函数

下面举个例子：

比如一个宿舍的同学同时快递到了，一种笨方法就是他们一个个去领取

较优方法就是把这件事情委托给宿舍长，让一个人出去拿好所有快递，然后再根据收件人一一分发给每个同学

在这里，取快递就是一个事件，每个同学指的是需要响应事件的 `DOM` 元素，而出去统一领取快递的宿舍长就是代理的元素

所以真正绑定事件的是这个元素，按照收件人分发快递的过程就是在事件执行中，需要判断当前响应的事件应该匹配到被代理元素中的哪一个或者哪几个

二、应用场景

如果我们有一个列表，列表之中有大量的列表项，我们需要在点击列表项的时候响应一个事件

```
<ul id="list">
  <li>item 1</li>
  <li>item 2</li>
  <li>item 3</li>
  ......
  <li>item n</li>
</ul>
```

如果给每个列表项一一都绑定一个函数，那对于内存消耗是非常大的

```
// 获取目标元素
const lis = document.getElementsByTagName("li")
// 循环遍历绑定事件
for (let i = 0; i < lis.length; i++) {
    lis[i].onclick = function(e){
        console.log(e.target.innerHTML)
    }
}
```

这时候就可以事件委托，把点击事件绑定在父级元素 `ul` 上面，然后执行事件的时候再去匹配目标元素

```
// 给父层元素绑定事件
document.getElementById('list').addEventListener('click', function (e) {
    // 兼容性处理
    var event = e || window.event;
    var target = event.target || event.srcElement;
    // 判断是否匹配目标元素
    if (target.nodeName.toLocaleLowerCase === 'li') {
        console.log('the content is: ', target.innerHTML);
    }
});
```

还有一种场景是上述列表项并不多，我们给每个列表项都绑定了事件

但是如果用户能够随时动态的增加或者去除列表项元素，那么在每一次改变的时候都需要重新给新增的元素绑定事件，给即将删去的元素解绑事件

如果用了事件委托就没有这种麻烦了，因为事件是绑定在父层的，和目标元素的增减是没有关系的，执行到目标元素是在真正响应执行事件函数的过程中去匹配的

举个例子：

下面`html`结构中，点击`input`可以动态添加元素

```
<input type="button" name="" id="btn" value="添加" />
<ul id="ul1">
    <li>item 1</li>
    <li>item 2</li>
    <li>item 3</li>
    <li>item 4</li>
</ul>
```

使用事件委托

```
const oBtn = document.getElementById("btn");
const oUl = document.getElementById("ul1");
const num = 4;

//事件委托，添加的子元素也有事件
oUl.onclick = function (ev) {
    ev = ev || window.event;
    const target = ev.target || ev.srcElement;
    if (target.nodeName.toLowerCase() == 'li') {
        console.log('the content is: ', target.innerHTML);
    }

};

//添加新节点
oBtn.onclick = function () {
    num++;
    const oLi = document.createElement('li');
    oLi.innerHTML = `item ${num}`;
    oUl.appendChild(oLi);
};
```

可以看到，使用事件委托，在动态绑定事件的情况下是可以减少很多重复工作的

三、总结

适合事件委托的事件有：`click`，`mousedown`，`mouseup`，`keydown`，`keyup`，`keypress`

从上面应用场景中，我们就可以看到使用事件委托存在两大优点：

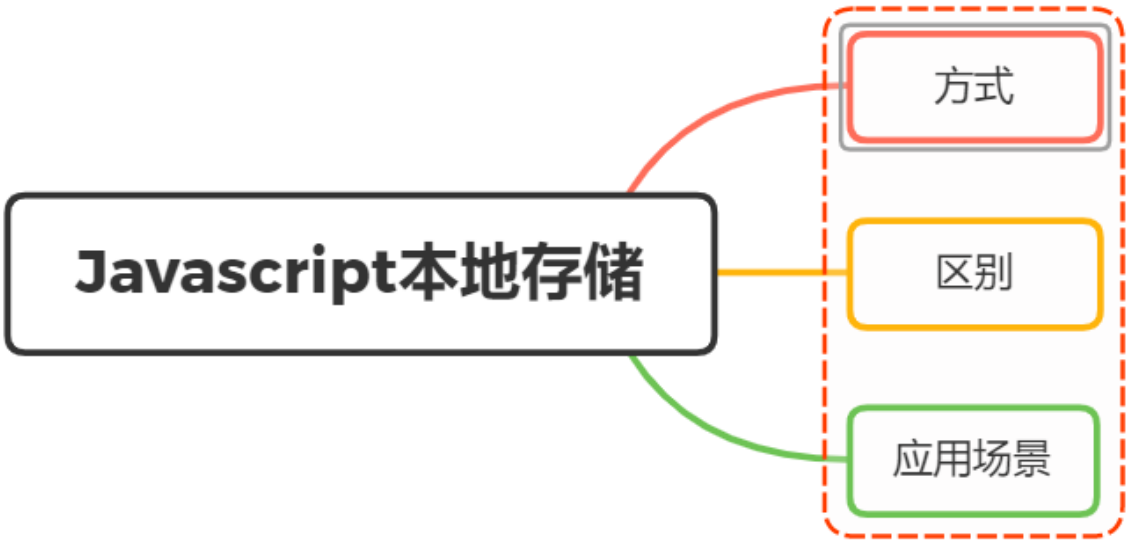
- 减少整个页面所需的内存，提升整体性能
- 动态绑定，减少重复工作

但是使用事件委托也是存在局限性：

- `focus`、`blur`这些事件没有事件冒泡机制，所以无法进行委托绑定事件
- `mousemove`、`mouseout`这样的事件，虽然有事件冒泡，但是只能不断通过位置去计算定位，对性能消耗高，因此也是不适合于事件委托的

如果把所有事件都用事件代理，可能会出现事件误判，即本不该被触发的事件被绑定上了事件

03.Javascript本地存储的方式有哪些？区别及应用场景？



一、方式

`javaScript`本地缓存的方法我们主要讲述以下四种：

- cookie
- sessionStorage
- localStorage
- indexedDB

cookie

`Cookie`，类型为「小型文本文件」，指某些网站为了辨别用户身份而储存在用户本地终端上的数据。是为了解决`HTTP`无状态导致的问题

作为一段一般不超过 4KB 的小型文本数据，它由一个名称（Name）、一个值（Value）和其它几个用于控制`cookie`有效期、安全性、使用范围的可选属性组成

但是cookie在每次请求中都会被发送，如果不使用HTTPS并对其加密，其保存的信息很容易被窃取，导致安全风险。举个例子，在一些使用cookie保持登录态的网站上，如果cookie被窃取，他人很容易利用你的cookie来假扮成你登录网站

关于cookie常用的属性如下：

- Expires 用于设置 Cookie 的过期时间

```
Expires=Wed, 21 Oct 2015 07:28:00 GMT
```

- Max-Age 用于设置在 Cookie 失效之前需要经过的秒数（优先级比Expires高）

```
Max-Age=604800
```

- Domain指定了 Cookie 可以送达的主机名
- Path指定了一个 URL路径，这个路径必须出现在要请求的资源的路径中才可以发送 Cookie 首部

```
Path=/docs   # /docs/Web/ 下的资源会带 Cookie 首部
```

- 标记为 Secure的 Cookie只应通过被HTTPS协议加密过的请求发送给服务端

通过上述，我们可以看到cookie又开始的作用并不是为了缓存而设计出来，只是借用了cookie的特性实现缓存

关于cookie的使用如下：

```
document.cookie = '名字=值';
```

关于cookie的修改，首先要确定domain和path属性都是相同的才可以，其中有一个不同得时候都会创建出一个新的cookie

```
Set-Cookie:name=aa; domain=aa.net; path=/  # 服务端设置
document.cookie =name=bb; domain=aa.net; path=/  # 客户端设置
```

最后cookie的删除，最常用的方法就是给cookie设置一个过期的事件，这样cookie过期后会被浏览器删除

localStorage

HTML5新方法，IE8及以上浏览器都兼容

特点

- 生命周期：持久化的本地存储，除非主动删除数据，否则数据是永远不会过期的
- 存储的信息在同一域中是共享的
- 当本页操作（新增、修改、删除）了localStorage的时候，本页面不会触发storage事件,但是别的页面会触发storage事件。
- 大小：5M（跟浏览器厂商有关系）
- localStorage本质上是对字符串的读取，如果存储内容多的话会消耗内存空间，会导致页面变卡
- 受同源策略的限制

下面再看看关于`localStorage`的使用

设置

```
localStorage.setItem('username','cfangxu');
```

获取

```
localStorage.getItem('username')
```

获取键名

```
localStorage.key(0) //获取第一个键名
```

删除

```
localStorage.removeItem('username')
```

一次性清除所有存储

```
localStorage.clear()
```

`localStorage` 也不是完美的，它有两个缺点：

- 无法像`Cookie`一样设置过期时间
- 只能存入字符串，无法直接存对象

```
localStorage.setItem('key', {name: 'value'});
console.log(localStorage.getItem('key')); // '[object, Object]'
```

sessionStorage

`sessionStorage`和 `localStorage`使用方法基本一致，唯一不同的是生命周期，一旦页面（会话）关闭，`sessionStorage` 将会删除数据

扩展的前端存储方式

`indexedDB`是一种低级API，用于客户端存储大量结构化数据(包括, 文件/ blobs)。该API使用索引来实现对该数据的高性能搜索

虽然 `Web Storage`对于存储较少量的数据很有用，但对于存储更大量的结构化数据来说，这种方法不太有用。`IndexedDB`提供了一个解决方案

优点：

- 储存量理论上没有上限
- 所有操作都是异步的，相比 `LocalStorage` 同步操作性能更高，尤其是数据量较大时
- 原生支持储存`JS`的对象
- 是个正经的数据库，意味着数据库能干的事它都能干

缺点：

- 操作非常繁琐
- 本身有一定门槛

关于`indexedDB`的使用基本使用步骤如下：

- 打开数据库并且开始一个事务
- 创建一个`object store`
- 构建一个请求来执行一些数据库操作，像增加或提取数据等。
- 通过监听正确类型的`DOM`事件以等待操作完成。
- 在操作结果上进行一些操作（可以在`request`对象中找到）

关于使用`indexdb`的使用会比较繁琐，大家可以通过使用`Godb.js`库进行缓存，最大化的降低操作难度

二、区别

关于`cookie`、`sessionStorage`、`localStorage`三者的区别主要如下：

- 存储大小：`cookie`数据大小不能超过`4k`，`sessionStorage`和`localStorage`虽然也有存储大小的限制，但比`cookie`大得多，可以达到5M或更大
- 有效时间：`localStorage`存储持久数据，浏览器关闭后数据不丢失除非主动删除数据；`sessionStorage`数据在当前浏览器窗口关闭后自动删除；`cookie`设置的`cookie`过期时间之前一直有效，即使窗口或浏览器关闭
- 数据与服务器之间的交互方式，`cookie`的数据会自动的传递到服务器，服务器端也可以写`cookie`到客户端；`sessionStorage`和`localStorage`不会自动把数据发给服务器，仅在本地保存

三、应用场景

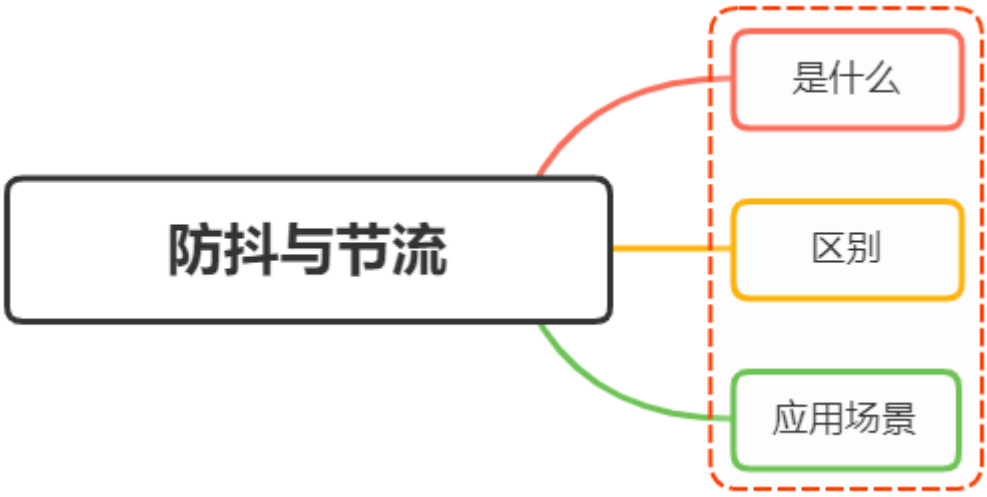
在了解了上述的前端的缓存方式后，我们可以看看针对不对场景的使用选择：

- 标记用户与跟踪用户行为的情况，推荐使用`cookie`
- 适合长期保存在本地的数据（令牌），推荐使用`localStorage`
- 敏感账号一次性登录，推荐使用`sessionStorage`
- 存储大量数据的情况、在线文档（富文本编辑器）保存编辑历史的情况，推荐使用`indexedDB`

相关连接

- https://mp.weixin.qq.com/s/mROjtpoXarN--UDfEMqwhQ
- https://github.com/chenstarx/GoDB.js

04.什么是防抖和节流？有什么区别？如何实现？



一、是什么

本质上是优化高频率执行代码的一种手段

如：浏览器的 `resize`、`scroll`、`keypress`、`mousemove` 等事件在触发时，会不断地调用绑定在事件上的回调函数，极大地浪费资源，降低前端性能

为了优化体验，需要对这类事件进行调用次数的限制，对此我们就可以采用 `throttle`（防抖）和 `debounce`（节流）的方式来减少调用频率

定义

- 节流: n 秒内只运行一次，若在 n 秒内重复触发，只有一次生效
- 防抖: n 秒后在执行该事件，若在 n 秒内被重复触发，则重新计时

一个经典的比喻:

想象每天上班大厦底下的电梯。把电梯完成一次运送，类比为一次函数的执行和响应

假设电梯有两种运行策略 `debounce` 和 `throttle`，超时设定为15秒，不考虑容量限制

电梯第一个人进来后，15秒后准时运送一次，这是节流

电梯第一个人进来后，等待15秒。如果过程中又有人进来，15秒等待重新计时，直到15秒后开始运送，这是防抖

代码实现

节流

完成节流可以使用时间戳与定时器的写法

使用时间戳写法，事件会立即执行，停止触发后没有办法再次执行

```
function throttled1(fn, delay = 500) {
    let oldtime = Date.now()
    return function (...args) {
        let newtime = Date.now()
        if (newtime - oldtime >= delay) {
            fn.apply(null, args)
            oldtime = Date.now()
        }
    }
}
```

使用定时器写法，`delay`毫秒后第一次执行，第二次事件停止触发后依然会再一次执行

```
function throttled2(fn, delay = 500) {
    let timer = null
    return function (...args) {
        if (!timer) {
            timer = setTimeout(() => {
                fn.apply(this, args)
                timer = null
            }, delay);
        }
    }
}
```

可以将时间戳写法的特性与定时器写法的特性相结合，实现一个更加精确的节流。实现如下

```
function throttled(fn, delay) {
    let timer = null
    let starttime = Date.now()
    return function () {
        let curTime = Date.now() // 当前时间
        let remaining = delay - (curTime - starttime)  // 从上一次到现在，还剩下多少多余时间
        let context = this
        let args = arguments
        clearTimeout(timer)
        if (remaining <= 0) {
            fn.apply(context, args)
            starttime = Date.now()
        } else {
            timer = setTimeout(fn, remaining);
        }
    }
}
```

防抖

简单版本的实现

```
function debounce(func, wait) {
    let timeout;

    return function () {
        let context = this; // 保存this指向
        let args = arguments; // 拿到event对象

        clearTimeout(timeout)
        timeout = setTimeout(function(){
            func.apply(context, args)
        }, wait);
    }
}
```

防抖如果需要立即执行，可加入第三个参数用于判断，实现如下：

```
function debounce(func, wait, immediate) {

    let timeout;

    return function () {
        let context = this;
        let args = arguments;

        if (timeout) clearTimeout(timeout); // timeout 不为null
        if (immediate) {
            let callNow = !timeout; // 第一次会立即执行，以后只有事件执行后才会再次触发
            timeout = setTimeout(function () {
                timeout = null;
            }, wait)
            if (callNow) {
                func.apply(context, args)
            }
        }
        else {
            timeout = setTimeout(function () {
                func.apply(context, args)
            }, wait);
        }
    }
}
```

二、区别

相同点：

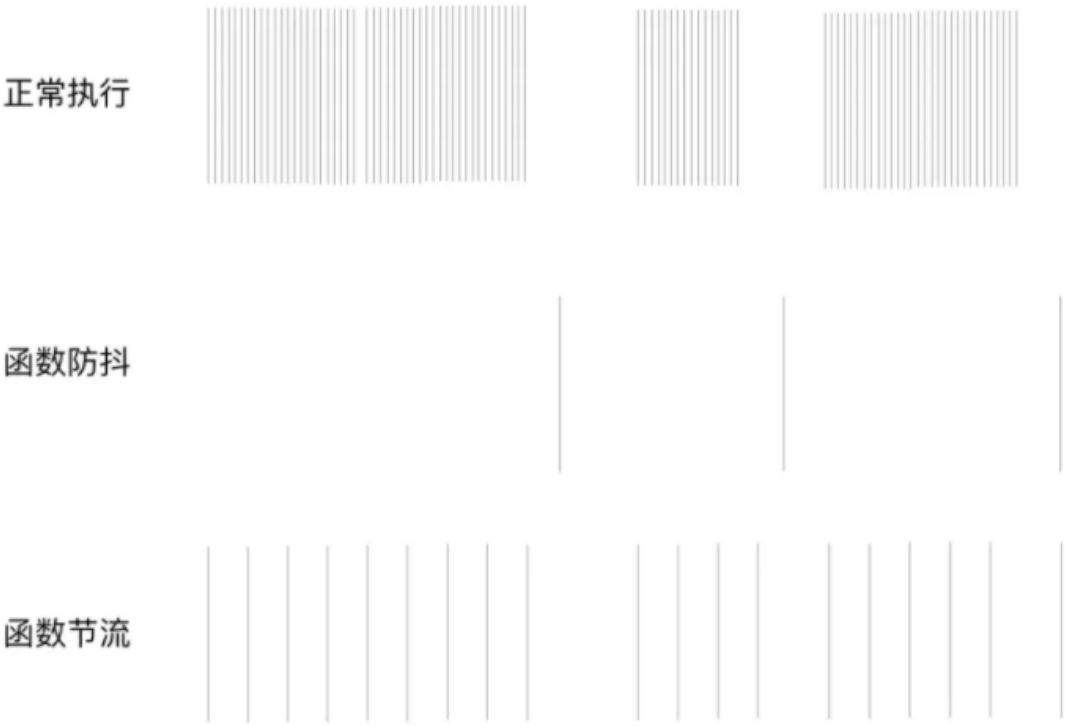
- 都可以通过使用 `setTimeout` 实现
- 目的都是，降低回调执行频率。节省计算资源

不同点：

- 函数防抖，在一段连续操作结束后，处理回调，利用`clearTimeout`和 `setTimeout`实现。函数节流，在一段连续操作中，每一段时间只执行一次，频率较高的事件中使用来提高性能
- 函数防抖关注一定时间连续触发的事件，只在最后执行一次，而函数节流一段时间内只执行一次

例如，都设置时间频率为500ms，在2秒时间内，频繁触发函数，节流，每隔 500ms 就执行一次。防抖，则不管调动多少次方法，在2s后，只会执行一次

如下图所示：



三、应用场景

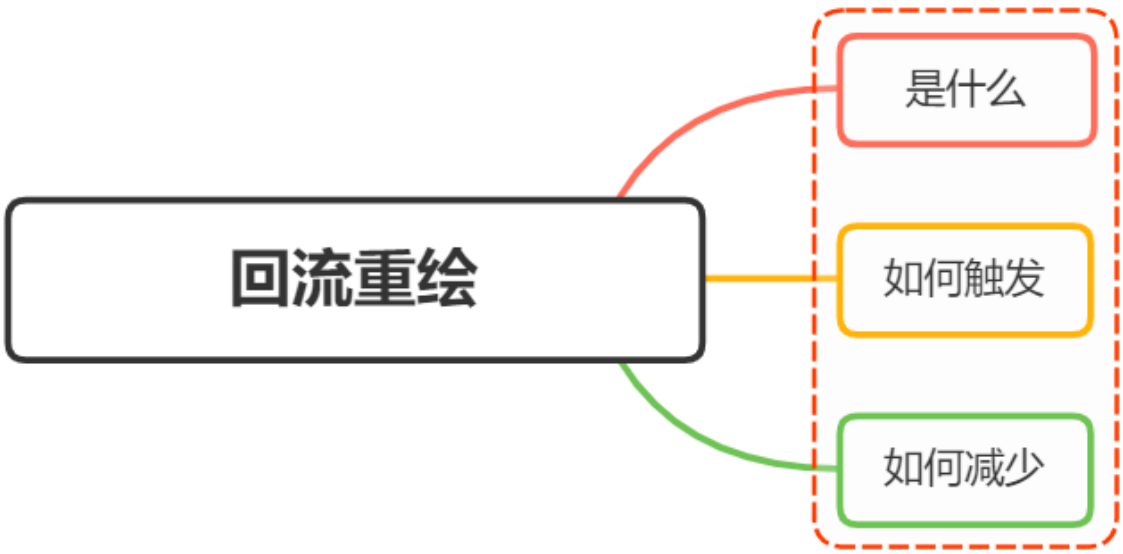
防抖在连续的事件，只需触发一次回调的场景有：

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

节流在间隔一段时间执行一次回调的场景有：

- 滚动加载，加载更多或滚到底部监听
- 搜索框，搜索联想功能

05.怎么理解回流跟重绘？什么场景下会触发？

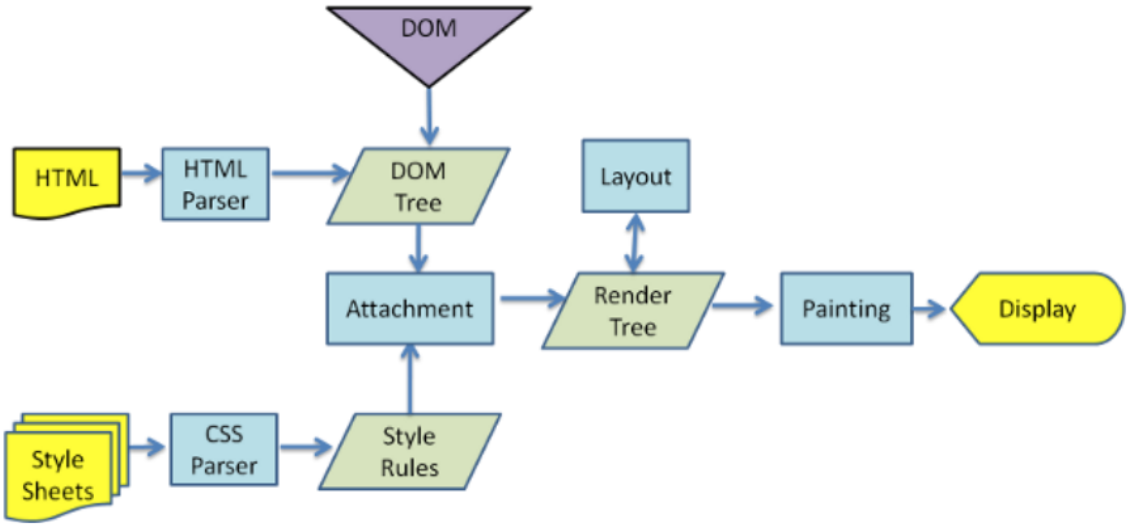


一、是什么

在`HTML`中，每个元素都可以理解成一个盒子，在浏览器解析过程中，会涉及到回流与重绘：

- 回流：布局引擎会根据各种样式计算每个盒子在页面上的大小与位置
- 重绘：当计算好盒模型的位置、大小及其他属性后，浏览器根据每个盒子特性进行绘制

具体的浏览器解析渲染机制如下所示：



- 解析HTML，生成DOM树，解析CSS，生成CSSOM树
- 将DOM树和CSSOM树结合，生成渲染树(Render Tree)
- Layout(回流):根据生成的渲染树，进行回流(Layout)，得到节点的几何信息（位置，大小）
- Painting(重绘):根据渲染树以及回流得到的几何信息，得到节点的绝对像素
- Display:将像素发送给GPU，展示在页面上

在页面初始渲染阶段，回流不可避免的触发，可以理解成页面一开始是空白的元素，后面添加了新的元素使页面布局发生改变

当我们对`DOM`的修改引发了`DOM`几何尺寸的变化（比如修改元素的宽、高或隐藏元素等）时，浏览器需要重新计算元素的几何属性，然后再将计算的结果绘制出来

当我们对`DOM`的修改导致了样式的变化（`color`或`background-color`），却并未影响其几何属性时，浏览器不需重新计算元素的几何属性、直接为该元素绘制新的样式，这里就仅仅触发了回流

二、如何触发

要想减少回流和重绘的次数，首先要了解回流和重绘是如何触发的

回流触发时机

回流这一阶段主要是计算节点的位置和几何信息，那么当页面布局和几何信息发生变化的时候，就需要回流，如下面情况：

- 添加或删除可见的DOM元素
- 元素的位置发生变化
- 元素的尺寸发生变化（包括外边距、内边框、边框大小、高度和宽度等）
- 内容发生变化，比如文本变化或图片被另一个不同尺寸的图片所替代
- 页面一开始渲染的时候（这避免不了）

- 浏览器的窗口尺寸变化（因为回流是根据视口的大小来计算元素的位置和大小的）

还有一些容易被忽略的操作：获取一些特定属性的值

> offsetTop、offsetLeft、 offsetWidth、offsetHeight、scrollTop、scrollLeft、scrollWidth、scrollHeight、clientTop、clientLeft、clientWidth、clientHeight

这些属性有一个共性，就是需要通过即时计算得到。因此浏览器为了获取这些值，也会进行回流

除此还包括`getComputedStyle`方法，原理是一样的

重绘触发时机

触发回流一定会触发重绘

可以把页面理解为一个黑板，黑板上有一朵画好的小花。现在我们要把这朵从左边移到了右边，那我们要先确定好右边的具体位置，画好形状（回流），再画上它原有的颜色（重绘）

除此之外还有一些其他引起重绘行为：

- 颜色的修改
- 文本方向的修改
- 阴影的修改

浏览器优化机制

由于每次重排都会造成额外的计算消耗，因此大多数浏览器都会通过队列化修改并批量执行来优化重排过程。浏览器会将修改操作放入到队列里，直到过了一段时间或者操作达到了一个阈值，才清空队列

当你获取布局信息的操作的时候，会强制队列刷新，包括前面讲到的`offsetTop`等方法都会返回最新的数据

因此浏览器不得不清空队列，触发回流重绘来返回正确的值

三、如何减少

我们了解了如何触发回流和重绘的场景，下面给出避免回流的经验：

- 如果想设定元素的样式，通过改变元素的 `class` 类名 (尽可能在 DOM 树的最里层)
- 避免设置多项内联样式
- 应用元素的动画，使用 `position` 属性的 `fixed` 值或 `absolute` 值(如前文示例所提)
- 避免使用 `table` 布局，`table` 中每个元素的大小以及内容的改动，都会导致整个 `table` 的重新计算
- 对于那些复杂的动画，对其设置 `position: fixed/absolute`，尽可能地使元素脱离文档流，从而减少对其他元素的影响
- 使用css3硬件加速，可以让`transform`、`opacity`、`filters`这些动画不会引起回流重绘
- 避免使用 CSS 的 `JavaScript` 表达式

在使用 `JavaScript` 动态插入多个节点时, 可以使用`DocumentFragment`. 创建后一次插入. 就能避免多次的渲染性能

但有时候，我们会无可避免地进行回流或者重绘，我们可以更好使用它们

例如，多次修改一个把元素布局的时候，我们很可能会如下操作

```
const el = document.getElementById('el')
for(let i=0;i<10;i++) {
    el.style.top  = el.offsetTop  + 10 + "px";
    el.style.left = el.offsetLeft + 10 + "px";
}
```

每次循环都需要获取多次`offset`属性，比较糟糕，可以使用变量的形式缓存起来，待计算完毕再提交给浏览器发出重计算请求

```
// 缓存offsetLeft与offsetTop的值
const el = document.getElementById('el')
let offLeft = el.offsetLeft, offTop = el.offsetTop

// 在JS层面进行计算
for(let i=0;i<10;i++) {
  offLeft += 10
  offTop  += 10
}

// 一次性将计算结果应用到DOM上
el.style.left = offLeft + "px"
el.style.top = offTop  + "px"
```

我们还可避免改变样式，使用类名去合并样式

```
const container = document.getElementById('container')
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
```

使用类名去合并样式

```
<style>
    .basic_style {
        width: 100px;
        height: 200px;
        border: 10px solid red;
        color: red;
    }
</style>
<script>
    const container = document.getElementById('container')
    container.classList.add('basic_style')
</script>
```

前者每次单独操作，都去触发一次渲染树更改（新浏览器不会），

都去触发一次渲染树更改，从而导致相应的回流与重绘过程

合并之后，等于我们将所有的更改一次性发出

我们还可以通过通过设置元素属性`display: none`，将其从页面上去掉，然后再进行后续操作，这些后续操作也不会触发回流与重绘，这个过程称为离线操作

```
const container = document.getElementById('container')
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
```

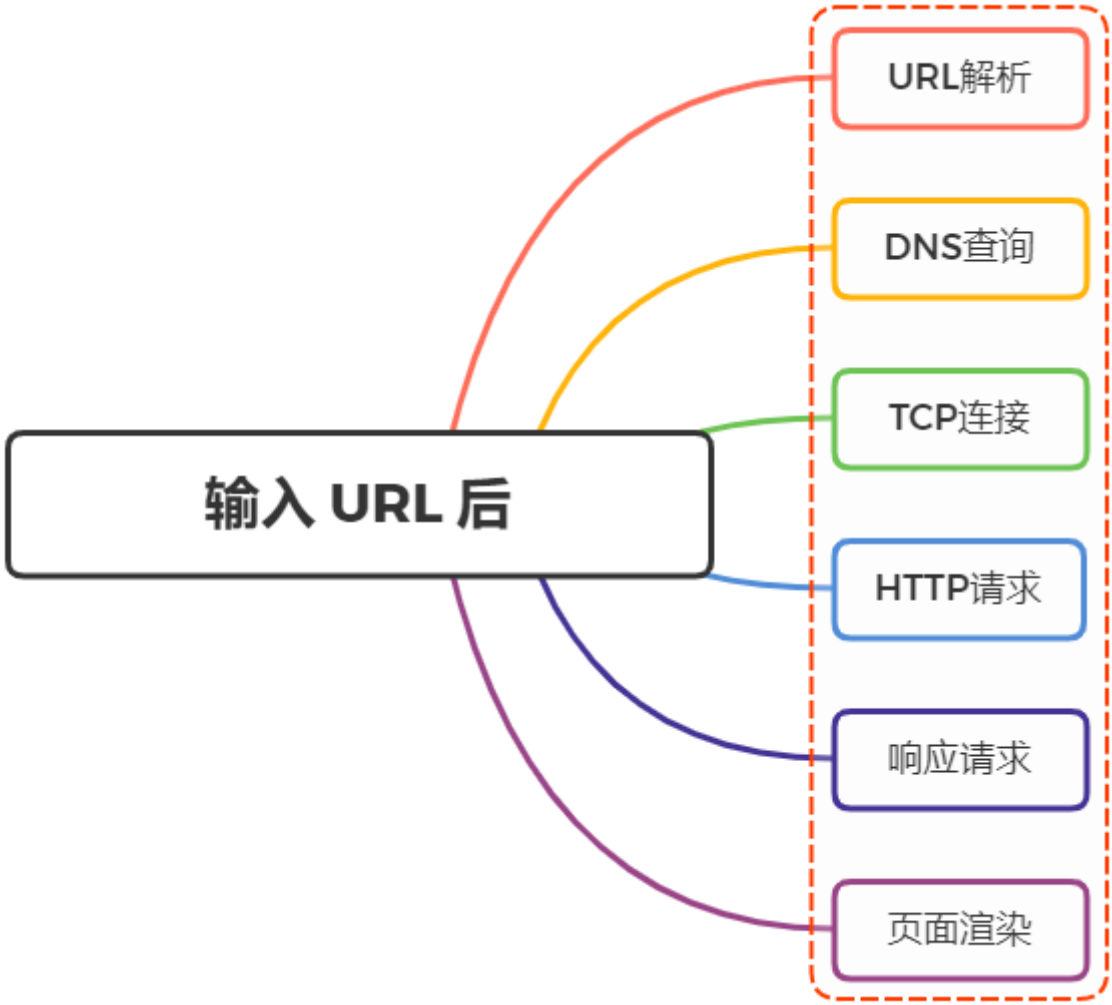
离线操作后

```
let container = document.getElementById('container')
container.style.display = 'none'
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
...（省略了许多类似的后续操作）
container.style.display = 'block'
```

参考文献

- https://juejin.cn/post/6844903942137053192
- https://segmentfault.com/a/1190000017329980

06.说说地址栏输入 URL 敲下回车后发生了什么



一、简单分析

简单的分析，从输入 `URL` 到回车后发生的行为如下：

- URL解析
- DNS 查询
- TCP 连接
- HTTP 请求
- 响应请求
- 页面渲染

二、详细分析

URL解析

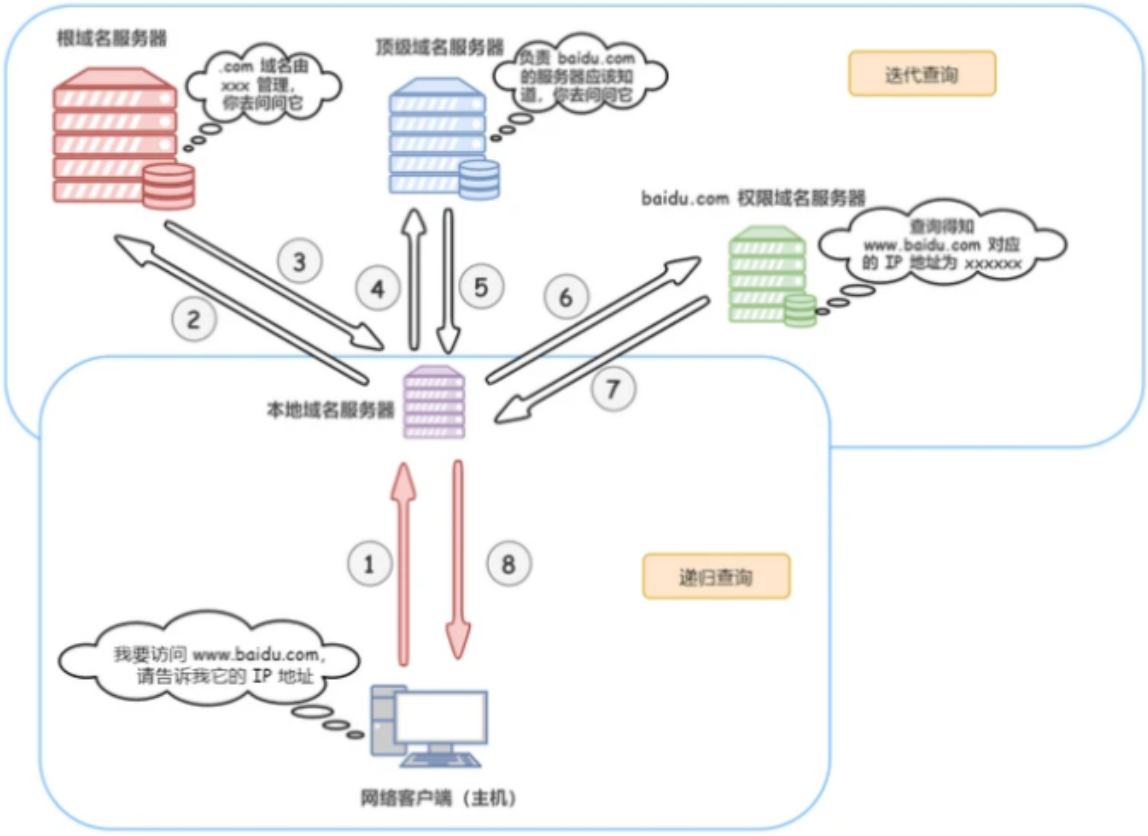
首先判断你输入的是一个合法的 `URL` 还是一个待搜索的关键词，并且根据你输入的内容进行对应操作

`URL` 的解析第过程中的第一步，一个 `url` 的结构解析如下：

DNS查询

在之前文章中讲过 `DNS` 的查询，这里就不再讲述了

整个查询过程如下图所示：

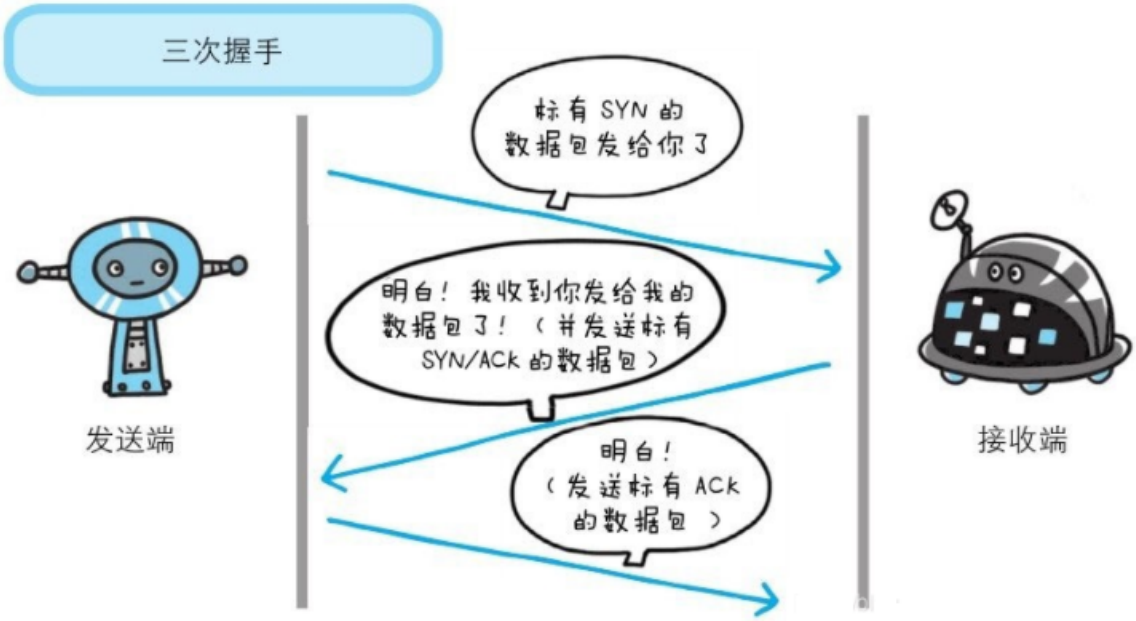


最终，获取到了域名对应的目标服务器`IP`地址

TCP连接

在之前文章中，了解到`tcp`是一种面向有连接的传输层协议

在确定目标服务器服务器的`IP`地址后，则经历三次握手建立`TCP`连接，流程如下：



发送 http 请求

当建立`tcp`连接之后，就可以在这基础上进行通信，浏览器发送`http`请求到目标服务器

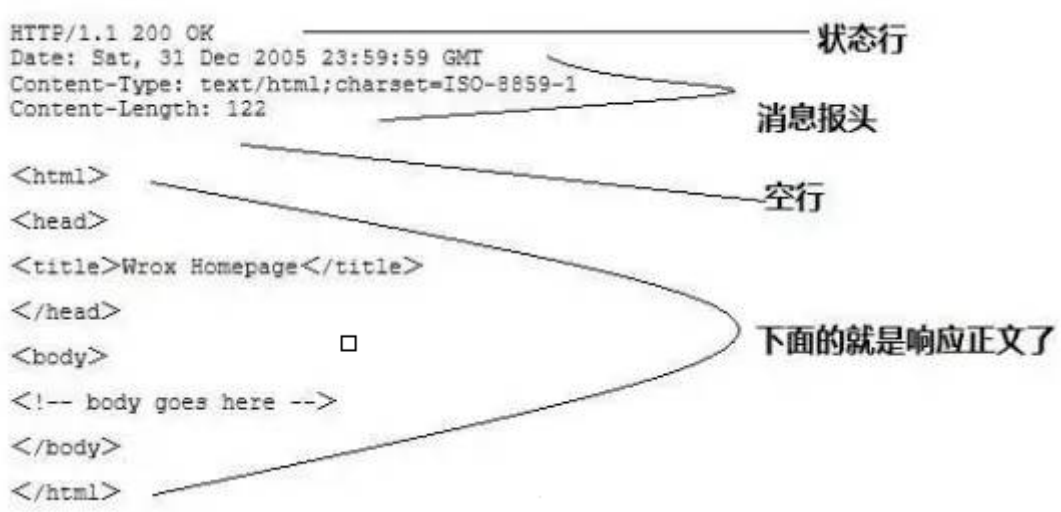
请求的内容包括：

- 请求行
- 请求头
- 请求主体

响应请求

当服务器接收到浏览器的请求之后，就会进行逻辑操作，处理完成之后返回一个`HTTP`响应消息，包括：

- 状态行
- 响应头
- 响应正文



在服务器响应之后，由于现在`http`默认开始长连接`keep-alive`，当页面关闭之后，`tcp`链接则会经过四次挥手完成断开

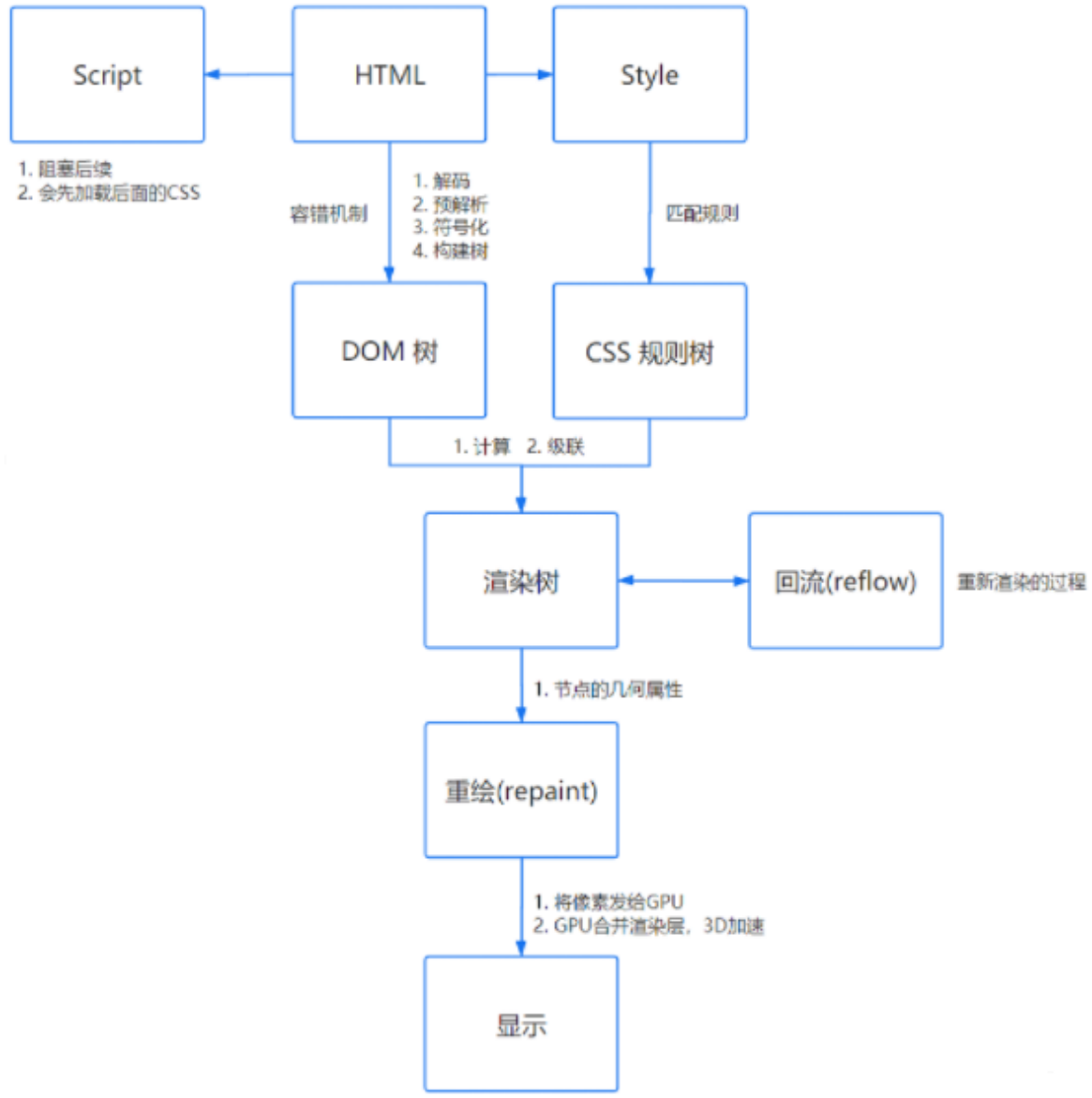
页面渲染

当浏览器接收到服务器响应的资源后，首先会对资源进行解析：

- 查看响应头的信息，根据不同的指示做对应处理，比如重定向，存储cookie，解压gzip，缓存资源等等
- 查看响应头的 Content-Type的值，根据不同的资源类型采用不同的解析方式

关于页面的渲染过程如下：

- 解析HTML，构建 DOM 树
- 解析 CSS ，生成 CSS 规则树
- 合并 DOM 树和 CSS 规则，生成 render 树
- 布局 render 树 （ Layout / reflow ），负责各元素尺寸、位置的计算
- 绘制 render 树 （ paint ），绘制页面像素信息
- 浏览器会将各层的信息发送给 GPU，GPU 会将各层合成 （ composite ），显示在屏幕上

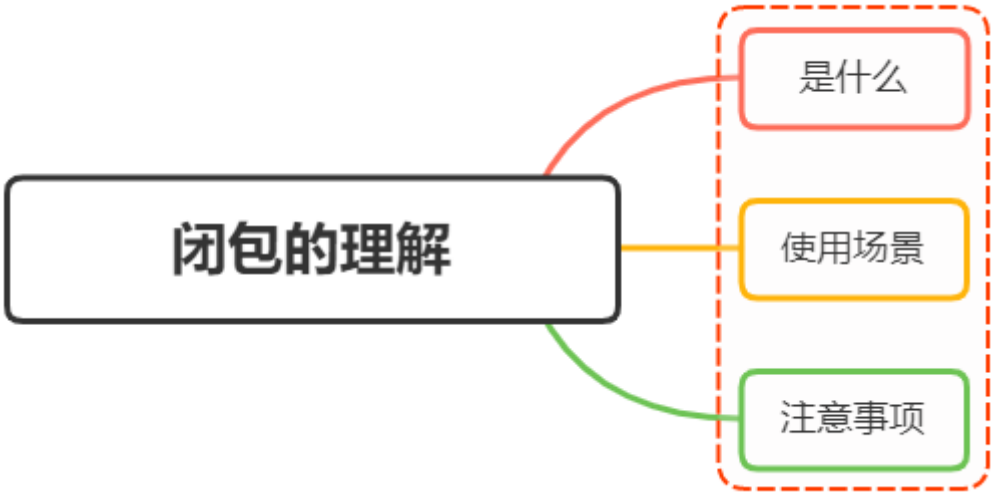


参考文献

- https://github.com/febobo/web-interview/issues/141
- https://zhuanlan.zhihu.com/p/80551769

5.JS高级

01.说说你对闭包的理解



一、是什么

一个函数和对其周围状态（lexical environment，词法环境）的引用捆绑在一起（或者说函数被引用包围），这样的组合就是闭包（closure）

也就是说，闭包让你可以在一个内层函数中访问到其外层函数的作用域

在 `JavaScript` 中，每当创建一个函数，闭包就会在函数创建的同时被创建出来，作为函数内部与外部连接起来的一座桥梁

下面给出一个简单的例子

```
function init() {
    var name = "Mozilla"; // name 是一个被 init 创建的局部变量
    function displayName() { // displayName() 是内部函数，一个闭包
        alert(name); // 使用了父函数中声明的变量
    }
    displayName();
}
init();
```

`displayName()` 没有自己的局部变量。然而，由于闭包的特性，它可以访问到外部函数的变量

二、使用场景

任何闭包的使用场景都离不开这两点：

- 创建私有变量
- 延长变量的生命周期

> 一般函数的词法环境在函数返回后就被销毁，但是闭包会保存对创建时所在词法环境的引用，即便创建时所在的执行上下文被销毁，但创建时所在词法环境依然存在，以达到延长变量的生命周期的目的

下面举个例子：

在页面上添加一些可以调整字号的按钮

```
function makeSizer(size) {
  return function() {
    document.body.style.fontSize = size + 'px';
  };
}

var size12 = makeSizer(12);
var size14 = makeSizer(14);
var size16 = makeSizer(16);

document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;
```

柯里化函数

柯里化的目的在于避免频繁调用具有相同参数函数的同时，又能够轻松的重用

```
// 假设我们有一个求长方形面积的函数
function getArea(width, height) {
    return width * height
}
// 如果我们碰到的长方形的宽老是10
const area1 = getArea(10, 20)
const area2 = getArea(10, 30)
const area3 = getArea(10, 40)

// 我们可以使用闭包柯里化这个计算面积的函数
function getArea(width) {
    return height => {
        return width * height
    }
}

const getTenWidthArea = getArea(10)
// 之后碰到宽度为10的长方形就可以这样计算面积
const area1 = getTenWidthArea(20)

// 而且如果遇到宽度偶尔变化也可以轻松复用
const getTwentyWidthArea = getArea(20)
```

使用闭包模拟私有方法

在 `JavaScript` 中，没有支持声明私有变量，但我们可以使用闭包来模拟私有方法

下面举个例子：

```
var Counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  }
})();

var Counter1 = makeCounter();
var Counter2 = makeCounter();
console.log(Counter1.value()); /* logs 0 */
Counter1.increment();
Counter1.increment();
console.log(Counter1.value()); /* logs 2 */
Counter1.decrement();
console.log(Counter1.value()); /* logs 1 */
console.log(Counter2.value()); /* logs 0 */
```

上述通过使用闭包来定义公共函数，并令其可以访问私有函数和变量，这种方式也叫模块方式

两个计数器 `Counter1` 和 `Counter2` 是维护它们各自的独立性的，每次调用其中一个计数器时，通过改变这个变量的值，会改变这个闭包的词法环境，不会影响另一个闭包中的变量

其他

例如计数器、延迟调用、回调等闭包的应用，其核心思想还是创建私有变量和延长变量的生命周期

三、注意事项

如果不是某些特定任务需要使用闭包，在其它函数中创建函数是不明智的，因为闭包在处理速度和内存消耗方面对脚本性能具有负面影响

例如，在创建新的对象或者类时，方法通常应该关联于对象的原型，而不是定义到对象的构造器中。

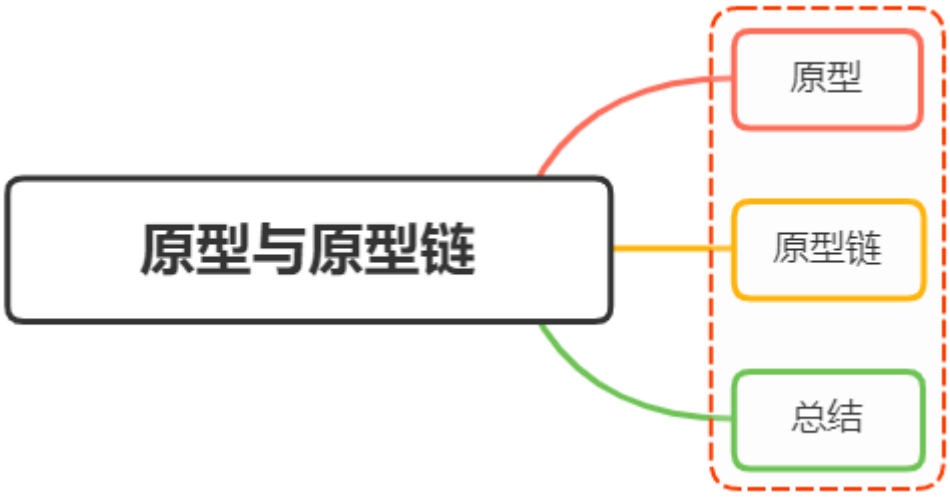
原因在于每个对象的创建，方法都会被重新赋值

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
  this.getName = function() {
    return this.name;
  };

  this.getMessage = function() {
    return this.message;
  };
}
```

上面的代码中，我们并没有利用到闭包的好处，因此可以避免使用闭包。修改成如下：

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype.getName = function() {
  return this.name;
};
MyObject.prototype.getMessage = function() {
  return this.message;
};
```

02.JavaScript原型，原型链？有什么特点?



一、原型

`JavaScript` 常被描述为一种基于原型的语言——每个对象拥有一个原型对象

当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾

准确地说，这些属性和方法定义在Object的构造器函数（constructor functions）之上的`prototype`属性上，而非实例对象本身

下面举个例子：

函数可以有属性。 每个函数都有一个特殊的属性叫作原型`prototype`

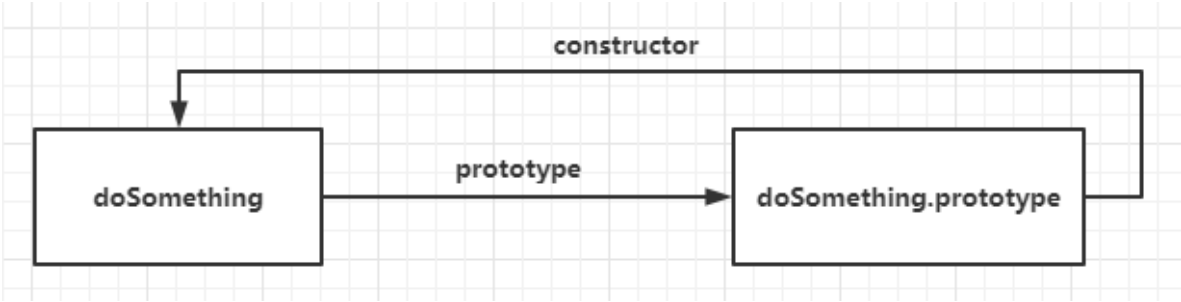
```
function doSomething(){}
console.log( doSomething.prototype );
```

控制台输出

```
{
    constructor: ƒ doSomething(),
    __proto__: {
        constructor: ƒ Object(),
        hasOwnProperty: ƒ hasOwnProperty(),
        isPrototypeOf: ƒ isPrototypeOf(),
        propertyIsEnumerable: ƒ propertyIsEnumerable(),
        toLocaleString: ƒ toLocaleString(),
        toString: ƒ toString(),
        valueOf: ƒ valueOf()
    }
}
```

上面这个对象，就是大家常说的原型对象

可以看到，原型对象有一个自有属性`constructor`，这个属性指向该函数，如下图关系展示



二、原型链

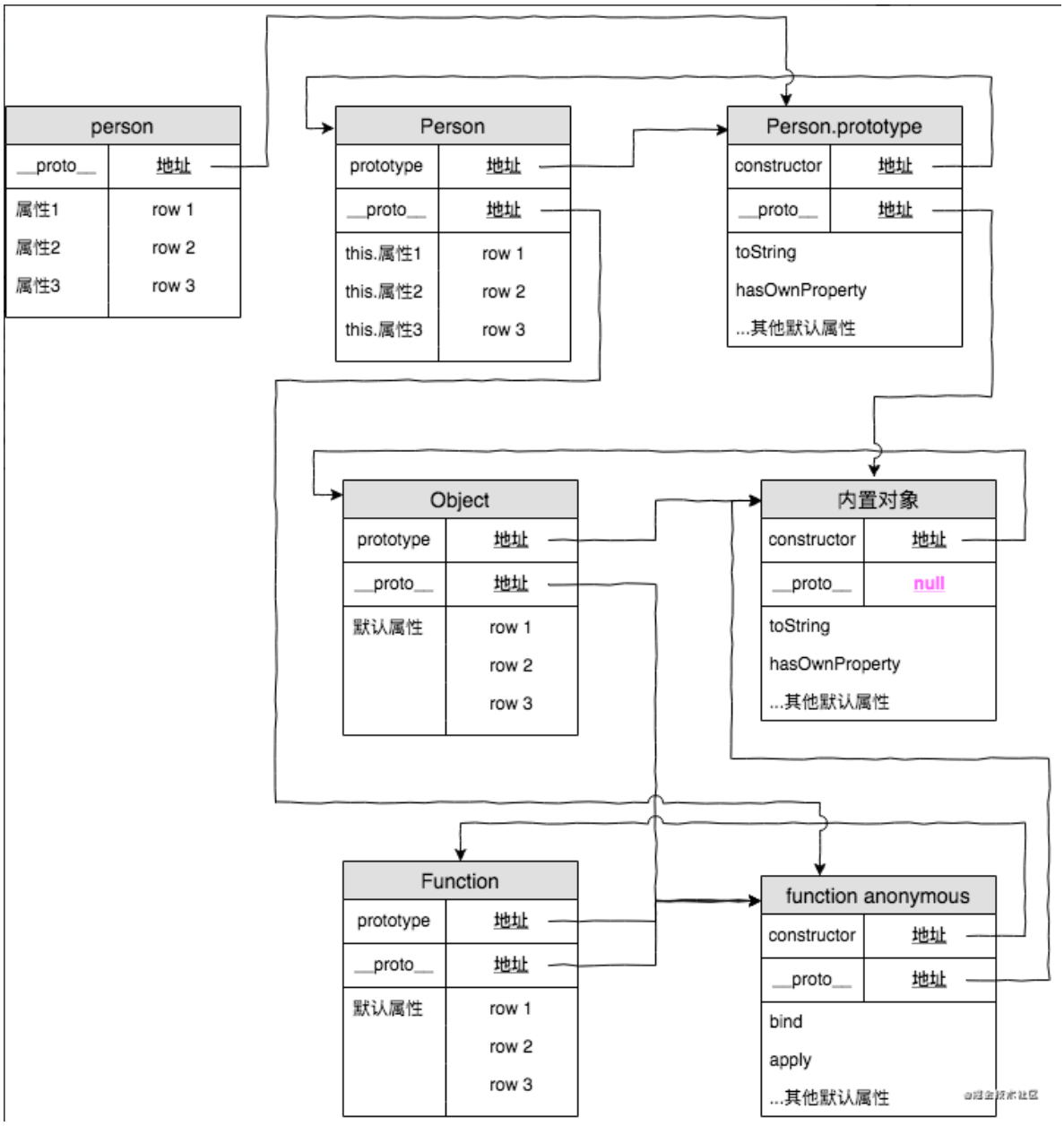
原型对象也可能拥有原型，并从中继承方法和属性，一层一层、以此类推。这种关系常被称为原型链 (prototype chain)，它解释了为何一个对象会拥有定义在其他对象中的属性和方法

在对象实例和它的构造器之间建立一个链接（它是`__proto__`属性，是从构造函数的`prototype`属性派生的），之后通过上溯原型链，在构造器中找到这些属性和方法

下面举个例子：

```
function Person(name) {
    this.name = name;
    this.age = 18;
    this.sayName = function() {
        console.log(this.name);
    }
}
// 第二步 创建实例
var person = new Person('person')
```

根据代码，我们可以得到下图



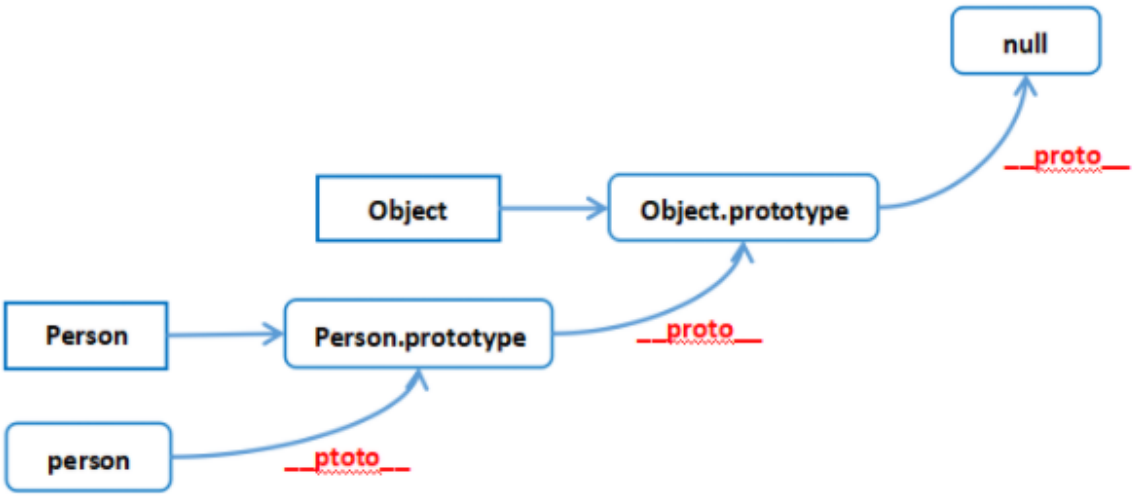
下面分析一下：

- 构造函数 `Person` 存在原型对象 `Person.prototype`
- 构造函数生成实例对象 `person`， `person` 的 `__proto__` 指向构造函数 `Person` 原型对象
- `Person.prototype.__proto__` 指向内置对象，因为 `Person.prototype` 是个对象，默认是由 `Object` 函数作为类创建的，而 `Object.prototype` 为内置对象
- `Person.__proto__` 指向内置匿名函数 `anonymous`，因为 Person 是个函数对象，默认由 Function 作为类创建
- `Function.prototype` 和 `Function.__proto__` 同时指向内置匿名函数 `anonymous`，这样原型链的终点就是 `null`

三、总结

下面首先要看几个概念：

`__proto__` 作为不同对象之间的桥梁，用来指向创建它的构造函数的原型对象的



每个对象的`__proto__`都是指向它的构造函数的原型对象`prototype`的

```
person1.__proto__ === Person.prototype
```

构造函数是一个函数对象，是通过`Function`构造器产生的

```
Person.__proto__ === Function.prototype
```

原型对象本身是一个普通对象，而普通对象的构造函数都是`Object`

```
Person.prototype.__proto__ === Object.prototype
```

刚刚上面说了，所有的构造器都是函数对象，函数对象都是`Function`构造产生的

```
Object.__proto__ === Function.prototype
```

`Object`的原型对象也有`__proto__`属性指向`null`，`null`是原型链的顶端

```
Object.prototype.__proto__ === null
```

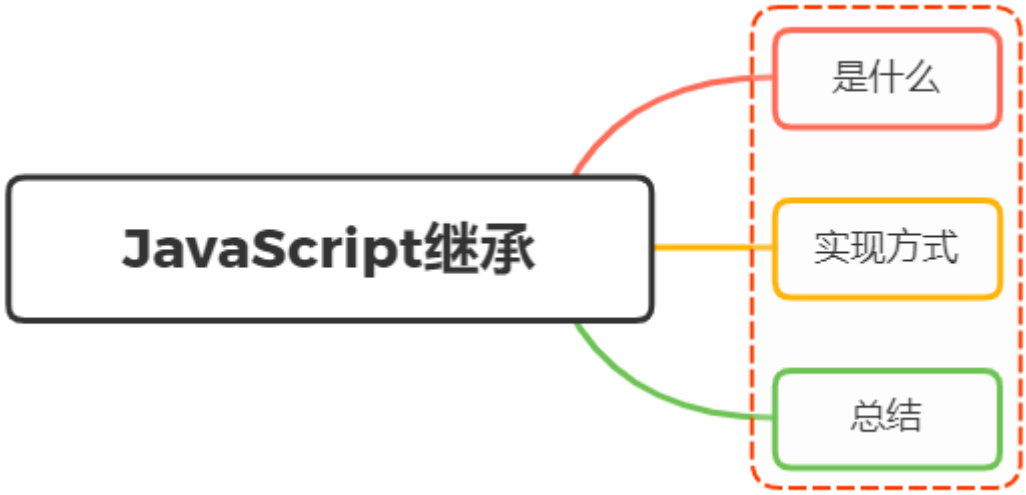
下面作出总结：

- 一切对象都是继承自`Object`对象，`Object`对象直接继承根源对象`null`
- 一切的函数对象（包括`Object`对象），都是继承自`Function`对象
- `Object`对象直接继承自`Function`对象
- `Function`对象的`__proto__`会指向自己的原型对象，最终还是继承自`Object`对象

参考文献

- https://juejin.cn/post/6870732239556640775#heading-7
- https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

03.avascript如何实现继承



一、是什么

继承（inheritance）是面向对象软件技术当中的一个概念。

如果一个类别B“继承自”另一个类别A，就把这个B称为“A的子类”，而把A称为“B的父类别”也可以称“A是B的超类”

- 继承的优点

继承可以使得子类具有父类别的各种属性和方法，而不需要再次编写相同的代码

在子类别继承父类别的同时，可以重新定义某些属性，并重写某些方法，即覆盖父类别的原有属性和方法，使其获得与父类别不同的功能

虽然`JavaScript`并不是真正的面向对象语言，但它天生的灵活性，使应用场景更加丰富

关于继承，我们举个形象的例子：

定义一个类（Class）叫汽车，汽车的属性包括颜色、轮胎、品牌、速度、排气量等

```
class Car{
    constructor(color,speed){
        this.color = color
        this.speed = speed
        // ...
    }
}
```

由汽车这个类可以派生出“轿车”和“货车”两个类，在汽车的基础属性上，为轿车添加一个后备厢、给货车添加一个大货箱

```
// 货车
class Truck extends Car{
    constructor(color,speed){
        super(color,speed)
        this.Container = true // 货箱
    }
}
```

这样轿车和货车就是不一样的，但是二者都属于汽车这个类，汽车、轿车继承了汽车的属性，而不需要再次在“轿车”中定义汽车已经有的属性

在“轿车”继承“汽车”的同时，也可以重新定义汽车的某些属性，并重写或覆盖某些属性和方法，使其获得与“汽车”这个父类不同的属性和方法

```
class Truck extends Car{
    constructor(color,speed){
        super(color,speed)
        this.color = "black" //覆盖
        this.Container = true // 货箱
    }
}
```

从这个例子中就能详细说明汽车、轿车以及卡车之间的继承关系

二、实现方式

下面给出`JavaScripy`常见的继承方式：

- 原型链继承
- 构造函数继承（借助 call）
- 组合继承
- 原型式继承
- 寄生式继承
- 寄生组合式继承

原型链继承

原型链继承是比较常见的继承方式之一，其中涉及的构造函数、原型和实例，三者之间存在着一定的关系，即每一个构造函数都有一个原型对象，原型对象又包含一个指向构造函数的指针，而实例则包含一个原型对象的指针

举个例子

```
 function Parent() {
    this.name = 'parent1';
    this.play = [1, 2, 3]
  }
  function Child() {
    this.type = 'child2';
  }
  Child1.prototype = new Parent();
  console.log(new Child())
```

上面代码看似没问题，实际存在潜在问题

```
var s1 = new Child2();
var s2 = new Child2();
s1.play.push(4);
console.log(s1.play, s2.play); // [1,2,3,4]
```

改变`s1`的`play`属性，会发现`s2`也跟着发生变化了，这是因为两个实例使用的是同一个原型对象，内存空间是共享的

构造函数继承

借助 `call` 调用 `Parent` 函数

```
function Parent(){
    this.name = 'parent1';
}

Parent.prototype.getName = function () {
    return this.name;
}

function Child(){
    Parent1.call(this);
    this.type = 'child'
}

let child = new Child();
console.log(child);  // 没问题
console.log(child.getName());  // 会报错
```

可以看到，父类原型对象中一旦存在父类之前自己定义的方法，那么子类将无法继承这些方法

相比第一种原型链继承方式，父类的引用属性不会被共享，优化了第一种继承方式的弊端，但是只能继承父类的实例属性和方法，不能继承原型属性或者方法

组合继承

前面我们讲到两种继承方式，各有优缺点。组合继承则将前两种方式继承起来

```
function Parent3 () {
    this.name = 'parent3';
    this.play = [1, 2, 3];
}

Parent3.prototype.getName = function () {
    return this.name;
}
function Child3() {
    // 第二次调用 Parent3()
    Parent3.call(this);
    this.type = 'child3';
}

// 第一次调用 Parent3()
Child3.prototype = new Parent3();
// 手动挂上构造器，指向自己的构造函数
Child3.prototype.constructor = Child3;
var s3 = new Child3();
var s4 = new Child3();
s3.play.push(4);
console.log(s3.play, s4.play);  // 不互相影响
console.log(s3.getName()); // 正常输出'parent3'
```

```
console.log(s4.getName()); // 正常输出'parent3'
```

这种方式看起来就没什么问题，方式一和方式二的问题都解决了，但是从上面代码我们也可以看到`Parent3`执行了两次，造成了多构造一次的性能开销

原型式继承

这里主要借助`Object.create`方法实现普通对象的继承

同样举个例子

```
let parent4 = {
    name: "parent4",
    friends: ["p1", "p2", "p3"],
    getName: function() {
      return this.name;
    }
  };

  let person4 = Object.create(parent4);
  person4.name = "tom";
  person4.friends.push("jerry");

  let person5 = Object.create(parent4);
  person5.friends.push("lucy");

  console.log(person4.name); // tom
  console.log(person4.name === person4.getName()); // true
  console.log(person5.name); // parent4
  console.log(person4.friends); // ["p1", "p2", "p3","jerry","lucy"]
  console.log(person5.friends); // ["p1", "p2", "p3","jerry","lucy"]
```

这种继承方式的缺点也很明显，因为`Object.create`方法实现的是浅拷贝，多个实例的引用类型属性指向相同的内存，存在篡改的可能

寄生式继承

寄生式继承在上面继承基础上进行优化，利用这个浅拷贝的能力再进行增强，添加一些方法

```
let parent5 = {
    name: "parent5",
    friends: ["p1", "p2", "p3"],
    getName: function() {
        return this.name;
    }
};

function clone(original) {
    let clone = Object.create(original);
    clone.getFriends = function() {
        return this.friends;
    };
    return clone;
```

```
}

let person5 = clone(parent5);

console.log(person5.getName()); // parent5
console.log(person5.getFriends()); // ["p1", "p2", "p3"]
```

其优缺点也很明显，跟上面讲的原型式继承一样

寄生组合式继承

寄生组合式继承，借助解决普通对象的继承问题的`Object.create`方法，在亲全面几种继承方式的优缺点基础上进行改造，这也是所有继承方式里面相对最优的继承方式

```
function clone (parent, child) {
    // 这里改用 Object.create 就可以减少组合继承中多进行一次构造的过程
    child.prototype = Object.create(parent.prototype);
    child.prototype.constructor = child;
}

function Parent6() {
    this.name = 'parent6';
    this.play = [1, 2, 3];
}
Parent6.prototype.getName = function () {
    return this.name;
}
function Child6() {
    Parent6.call(this);
    this.friends = 'child5';
}

clone(Parent6, Child6);

Child6.prototype.getFriends = function () {
    return this.friends;
}

let person6 = new Child6();
console.log(person6); //{friends:"child5",name:"child5",play:
[1,2,3],__proto__:Parent6}
console.log(person6.getName()); // parent6
console.log(person6.getFriends()); // child5
```

可以看到 person6 打印出来的结果，属性都得到了继承，方法也没问题

文章一开头，我们是使用`ES6`中的`extends`关键字直接实现`JavaScript`的继承

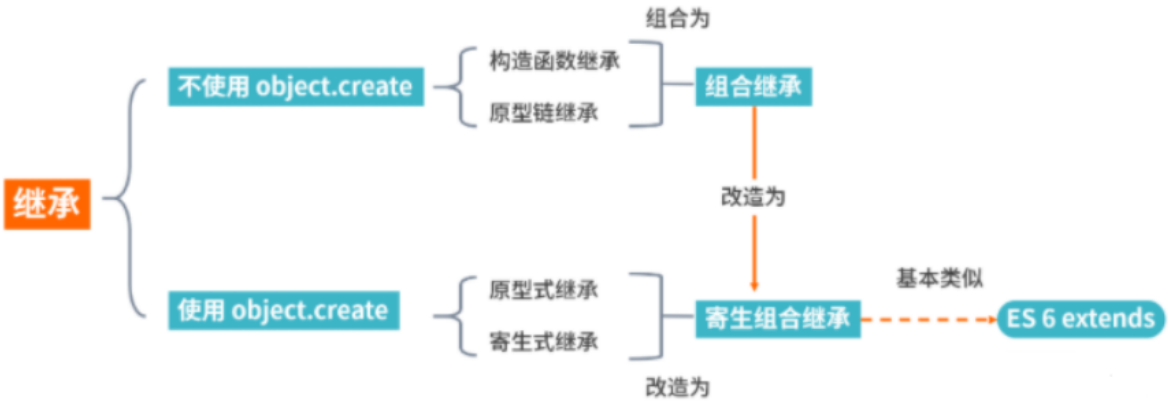
```
class Person {
  constructor(name) {
    this.name = name
  }
  // 原型方法
```

```
  // 即 Person.prototype.getName = function() { }
  // 下面可以简写为 getName() {...}
  getName = function () {
    console.log('Person:', this.name)
  }
}
class Gamer extends Person {
  constructor(name, age) {
    // 子类中存在构造函数，则需要在使用“this”之前首先调用 super()。
    super(name)
    this.age = age
  }
}
const asuna = new Gamer('Asuna', 20)
asuna.getName() // 成功访问到父类的方法
```

利用 `babel` 工具进行转换，我们会发现 `extends` 实际采用的也是寄生组合继承方式，因此也证明了这种方式是较优的解决继承的方式

三、总结

下面以一张图作为总结：

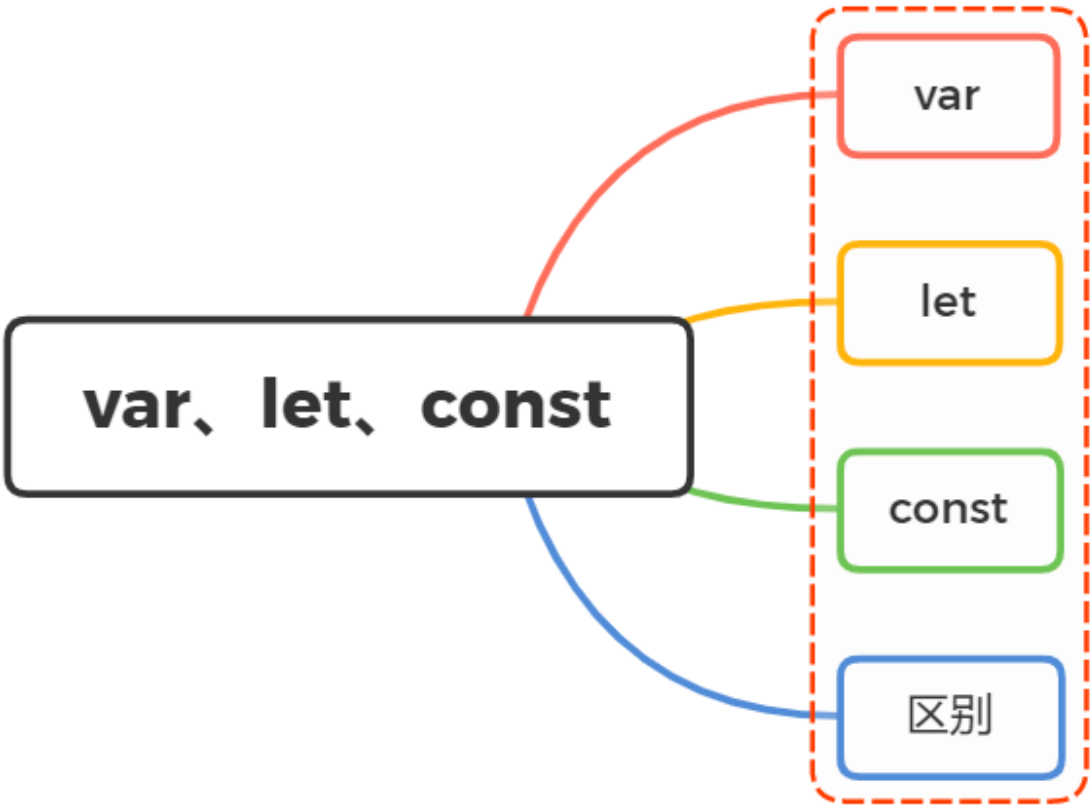


通过 `Object.create` 来划分不同的继承方式，最后的寄生式组合继承方式是通过组合继承改造之后的最优继承方式，而 `extends` 的语法糖和寄生组合继承的方式基本类似

相关链接

https://zh.wikipedia.org/wiki/%E7%BB%A7%E6%89%BF

04.说说var、let、const之间的区别



一、var

在ES5中，顶层对象的属性和全局变量是等价的，用`var`声明的变量既是全局变量，也是顶层变量

注意：顶层对象，在浏览器环境指的是`window`对象，在`Node`指的是`global`对象

```
var a = 10;
console.log(window.a) // 10
```

使用`var`声明的变量存在变量提升的情况

```
console.log(a) // undefined
var a = 20
```

在编译阶段，编译器会将其变成以下执行

```
var a
console.log(a)
a = 20
```

使用`var`，我们能够对一个变量进行多次声明，后面声明的变量会覆盖前面的变量声明

```
var a = 20
var a = 30
console.log(a) // 30
```

在函数中使用使用`var`声明变量时候，该变量是局部的

```
var a = 20
function change(){
    var a = 30
}
change()
console.log(a) // 20
```

而如果在函数内不使用`var`，该变量是全局的

```
var a = 20
function change(){
   a = 30
}
change()
console.log(a) // 30
```

二、let

`let`是`ES6`新增的命令，用来声明变量

用法类似于`var`，但是所声明的变量，只在`let`命令所在的代码块内有效

```
{
    let a = 20
}
console.log(a) // ReferenceError: a is not defined.
```

不存在变量提升

```
console.log(a) // 报错ReferenceError
let a = 2
```

这表示在声明它之前，变量`a`是不存在的，这时如果用到它，就会抛出一个错误

只要块级作用域内存在`let`命令，这个区域就不再受外部影响

```
var a = 123
if (true) {
    a = 'abc' // ReferenceError
    let a;
}
```

使用`let`声明变量前，该变量都不可用，也就是大家常说的“暂时性死区”

最后，`let`不允许在相同作用域中重复声明

```
let a = 20
let a = 30
// Uncaught SyntaxError: Identifier 'a' has already been declared
```

注意的是相同作用域，下面这种情况是不会报错的

```
let a = 20
{
    let a = 30
}
```

因此，我们不能在函数内部重新声明参数

```
function func(arg) {
  let arg;
}
func()
// Uncaught SyntaxError: Identifier 'arg' has already been declared
```

三、const

`const`声明一个只读的常量，一旦声明，常量的值就不能改变

```
const a = 1
a = 3
// TypeError: Assignment to constant variable.
```

这意味着，`const`一旦声明变量，就必须立即初始化，不能留到以后赋值

```
const a;
// SyntaxError: Missing initializer in const declaration
```

如果之前用`var`或`let`声明过变量，再用`const`声明同样会报错

```
var a = 20
let b = 20
const a = 30
const b = 30
// 都会报错
```

`const`实际上保证的并不是变量的值不得改动，而是变量指向的那个内存地址所保存的数据不得改动

对于简单类型的数据，值就保存在变量指向的那个内存地址，因此等同于常量

对于复杂类型的数据，变量指向的内存地址，保存的只是一个指向实际数据的指针，`const`只能保证这个指针是固定的，并不能确保改变量的结构不变

```
const foo = {};

// 为 foo 添加一个属性，可以成功
foo.prop = 123;
foo.prop // 123

// 将 foo 指向另一个对象，就会报错
foo = {}; // TypeError: "foo" is read-only
```

其它情况，`const`与`let`一致

四、区别

`var`、`let`、`const`三者区别可以围绕下面五点展开：

- 变量提升
- 暂时性死区
- 块级作用域
- 重复声明
- 修改声明的变量
- 使用

变量提升

`var`声明的变量存在变量提升，即变量可以在声明之前调用，值为`undefined`

`let`和`const`不存在变量提升，即它们所声明的变量一定要在声明后使用，否则报错

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

// let
console.log(b)  // Cannot access 'b' before initialization
let b = 10

// const
console.log(c)  // Cannot access 'c' before initialization
const c = 10
```

暂时性死区

`var`不存在暂时性死区

`let`和`const`存在暂时性死区，只有等到声明变量的那一行代码出现，才可以获取和使用该变量

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

// let
console.log(b)  // Cannot access 'b' before initialization
let b = 10

// const
console.log(c)  // Cannot access 'c' before initialization
const c = 10
```

块级作用域

`var`不存在块级作用域

`let`和`const`存在块级作用域

```
```

```
// var
{
    var a = 20
}
console.log(a)  // 20

// let
{
    let b = 20
}
console.log(b)  // Uncaught ReferenceError: b is not defined

// const
{
    const c = 20
}
console.log(c)  // Uncaught ReferenceError: c is not defined
```

重复声明

`var`允许重复声明变量

`let`和`const`在同一作用域不允许重复声明变量

```
// var
var a = 10
var a = 20 // 20

// let
let b = 10
let b = 20 // Identifier 'b' has already been declared

// const
const c = 10
const c = 20 // Identifier 'c' has already been declared
```

修改声明的变量

`var`和`let`可以

`const`声明一个只读的常量。一旦声明，常量的值就不能改变

```
// var
var a = 10
a = 20
console.log(a)  // 20

//let
let b = 10
b = 20
console.log(b)  // 20

// const
```

```
const c = 10
c = 20
console.log(c) // Uncaught TypeError: Assignment to constant variable
```

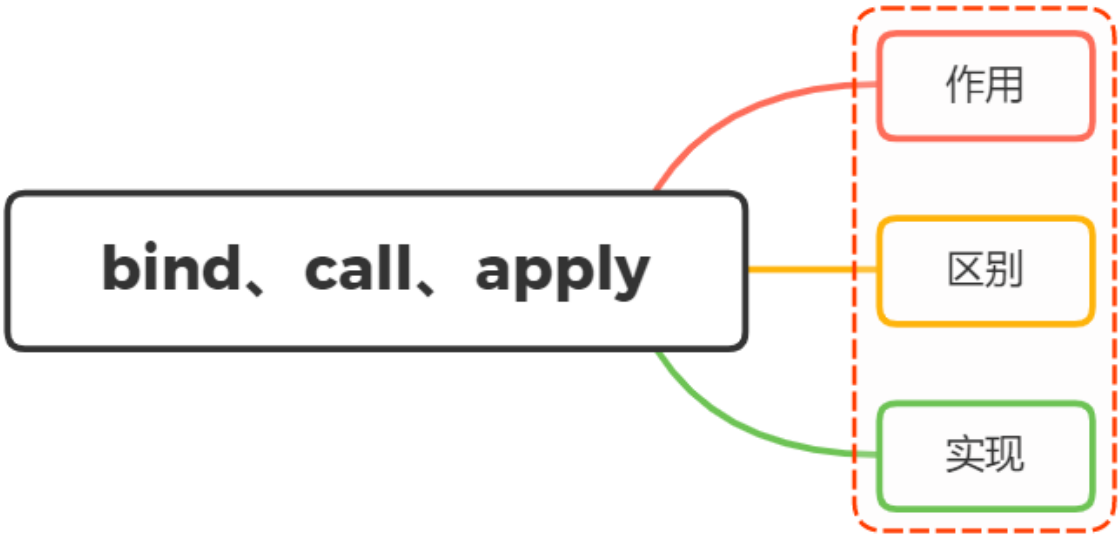
使用

能用`const`的情况尽量使用`const`，其他情况下大多数使用`let`，避免使用`var`

参考文献

- https://es6.ruanyifeng.com/

05.bind、call、apply 区别？如何实现一个bind?



一、作用

`call`、`apply`、`bind`作用是改变函数执行时的上下文，简而言之就是改变函数运行时的`this`指向

那么什么情况下需要改变`this`的指向呢？下面举个例子

```
const name="lucy";
const obj={
    name:"martin",
    say:function () {
        console.log(this.name);
    }
};
obj.say(); //martin，this指向obj对象
setTimeout(obj.say,0); //lucy，this指向window对象
```

从上面可以看到，正常情况`say`方法输出`martin`

但是我们把 say 放在 setTimeout 方法中，在定时器中是作为回调函数来执行的，因此回到主栈执行时是在全局执行上下文的环境中执行的，这时候 this 指向 window，所以输出 lucy

我们实际需要的是 this 指向 obj 对象，这时候就需要该改变 this 指向了

```
setTimeout(obj.say.bind(obj),0); //martin，this指向obj对象
```

二、区别

下面再来看看 apply、call、bind 的使用

apply

apply 接受两个参数，第一个参数是 this 的指向，第二个参数是函数接受的参数，以数组的形式传入

改变 this 指向后原函数会立即执行，且此方法只是临时改变 this 指向一次

```
function fn(...args){
    console.log(this,args);
}
let obj = {
    myname:"张三"
}

fn.apply(obj,[1,2]); // this会变成传入的obj，传入的参数必须是一个数组；
fn(1,2) // this指向window
```

当第一个参数为 null、undefined 的时候，默认指向 window(在浏览器中)

```
fn.apply(null,[1,2]); // this指向window
fn.apply(undefined,[1,2]); // this指向window
```

call

call 方法的第一个参数也是 this 的指向，后面传入的是一个参数列表

跟 apply 一样，改变 this 指向后原函数会立即执行，且此方法只是临时改变 this 指向一次

```
function fn(...args){
    console.log(this,args);
}
let obj = {
    myname:"张三"
}

fn.call(obj,1,2); // this会变成传入的obj，传入的参数必须是一个数组；
fn(1,2) // this指向window
```

同样的，当第一个参数为 null、undefined 的时候，默认指向 window(在浏览器中)

```
fn.call(null,[1,2]); // this指向window
fn.call(undefined,[1,2]); // this指向window
```

bind

bind方法和call很相似，第一参数也是`this`的指向，后面传入的也是一个参数列表(但是这个参数列表可以分多次传入)

改变`this`指向后不会立即执行，而是返回一个永久改变`this`指向的函数

```
function fn(...args){
    console.log(this,args);
}
let obj = {
    myname:"张三"
}

const bindFn = fn.bind(obj); // this 也会变成传入的obj ，bind不是立即执行需要执行一次
bindFn(1,2) // this指向obj
fn(1,2) // this指向window
```

小结

从上面可以看到，`apply`、`call`、`bind`三者的区别在于：

- 三者都可以改变函数的`this`对象指向
- 三者第一个参数都是`this`要指向的对象，如果如果没有这个参数或参数为`undefined`或`null`，则默认指向全局`window`
- 三者都可以传参，但是`apply`是数组，而`call`是参数列表，且`apply`和`call`是一次性传入参数，而`bind`可以分为多次传入
- `bind`是返回绑定this之后的函数，`apply`、`call`则是立即执行

三、实现

实现`bind`的步骤，我们可以分解成为三部分：

- 修改`this`指向
- 动态传递参数

```
// 方式一：只在bind中传递函数参数
fn.bind(obj,1,2)()

// 方式二：在bind中传递函数参数，也在返回函数中传递参数
fn.bind(obj,1)(2)
```

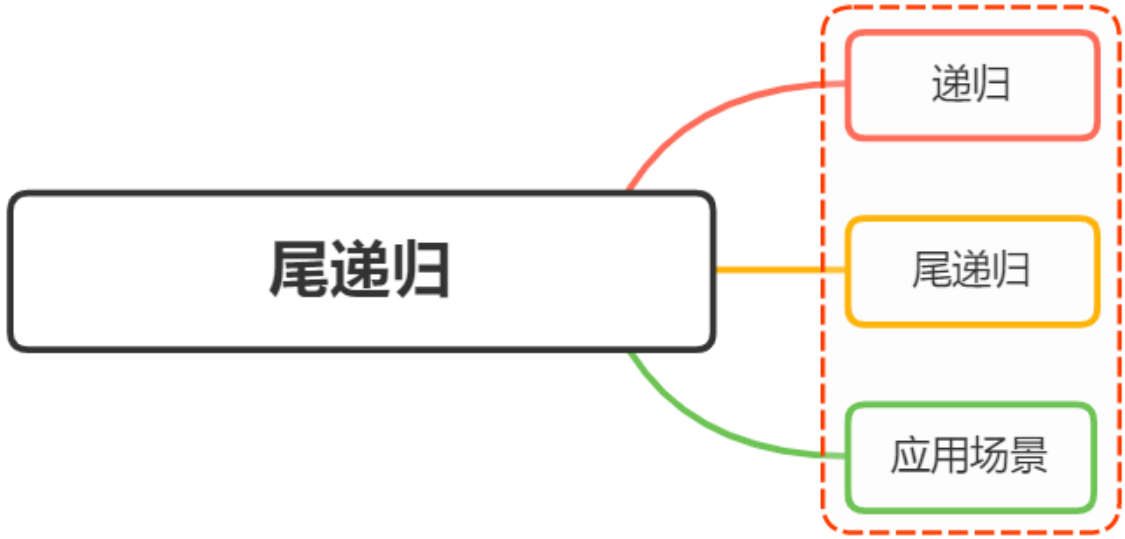
- 兼容`new`关键字

整体实现代码如下：

```
Function.prototype.myBind = function (context) {
    // 判断调用对象是否为函数
```

```
    if (typeof this !== "function") {
        throw new TypeError("Error");
    }

    // 获取参数
    const args = [...arguments].slice(1),
          fn = this;

    return function Fn() {

        // 根据调用方式，传入不同绑定值
        return fn.apply(this instanceof Fn ? new fn(...arguments) : context,
args.concat(...arguments));
    }
}
```

06.举例说明你对尾递归的理解，有哪些应用场景



一、递归

递归（英语：Recursion）

在数学与计算机科学中，是指在函数的定义中使用函数自身的方法

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数

其核心思想是把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解

一般来说，递归需要有边界条件、递归前进阶段和递归返回阶段。当边界条件不满足时，递归前进；当边界条件满足时，递归返回

下面实现一个函数 `pow(x, n)`，它可以计算 `x` 的 `n` 次方

使用迭代的方式，如下：

```
function pow(x, n) {
  let result = 1;

  // 再循环中，用 x 乘以 result n 次
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
```

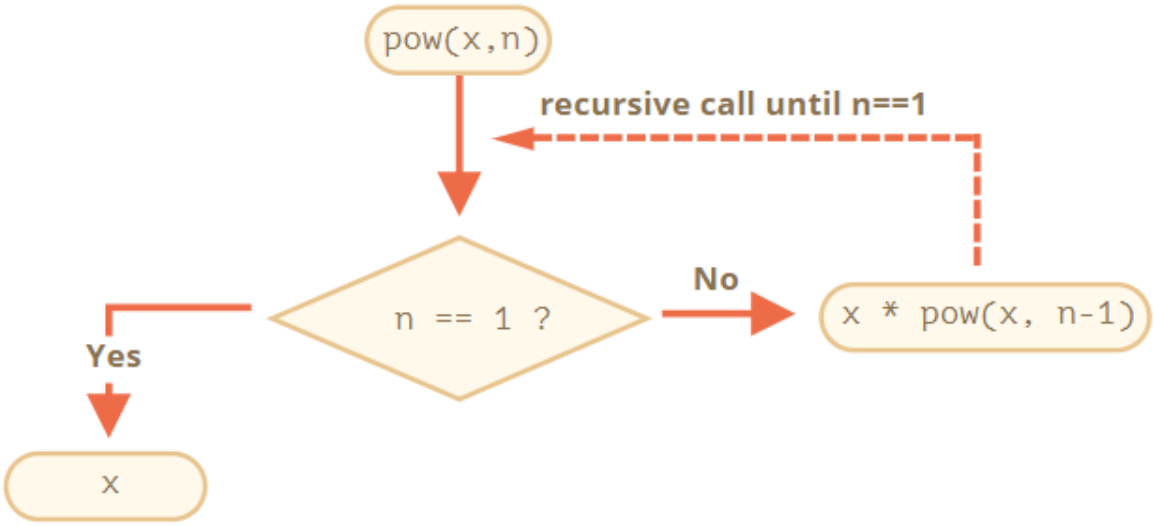
使用递归的方式，如下：

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}
```

`pow(x, n)` 被调用时，执行分为两个分支：

```
              if n==1  = x
              /
pow(x, n) =
              \
               else     = x * pow(x, n - 1)
```

也就是说`pow` 递归地调用自身 直到 `n == 1`



为了计算 `pow(2, 4)`，递归变体经过了下面几个步骤：

1. `pow(2, 4) = 2 * pow(2, 3)`
2. `pow(2, 3) = 2 * pow(2, 2)`
3. `pow(2, 2) = 2 * pow(2, 1)`
4. `pow(2, 1) = 2`

因此，递归将函数调用简化为一个更简单的函数调用，然后再将其简化为一个更简单的函数，以此类推，直到结果

二、尾递归

尾递归，即在函数尾位置调用自身（或是一个尾调用本身的其他函数等等）。尾递归也是递归的一种特殊情形。尾递归是一种特殊的尾调用，即在尾部直接调用自身的递归函数

尾递归在普通尾调用的基础上，多出了2个特征：

- 在尾部调用的是函数自身
- 可通过优化，使得计算仅占用常量栈空间

在递归调用的过程当中系统为每一层的返回点、局部量等开辟了栈来存储，递归次数过多容易造成栈溢出

这时候，我们就可以使用尾递归，即一个函数中所有递归形式的调用都出现在函数的末尾，对于尾递归来说，由于只存在一个调用记录，所以永远不会发生"栈溢出"错误

实现一下阶乘，如果用普通的递归，如下：

```
function factorial(n) {
  if (n === 1) return 1;
  return n * factorial(n - 1);
}

factorial(5) // 120
```

如果`n`等于5，这个方法要执行5次，才返回最终的计算表达式，这样每次都要保存这个方法，就容易造成栈溢出，复杂度为`O(n)`

如果我们使用尾递归，则如下：

```
function factorial(n, total) {
  if (n === 1) return total;
  return factorial(n - 1, n * total);
}

factorial(5) // 120
```

可以看到，每一次返回的就是一个新的函数，不带上一个函数的参数，也就不需要储存上一个函数了。尾递归只需要保存一个调用栈，复杂度 O(1)

二、应用场景

数组求和

```
function sumArray(arr, total) {
    if(arr.length === 1) {
        return total
    }
    return sum(arr, total + arr.pop())
}
```

使用尾递归优化求斐波那契数列

```
function factorial2 (n, start = 1, total = 1) {
    if(n <= 2){
        return total
    }
    return factorial2 (n -1, total, total + start)
}
```

数组扁平化

```
let a = [1,2,3, [1,2,3, [1,2,3]]]
// 变成
let a = [1,2,3,1,2,3,1,2,3]
// 具体实现
function flat(arr = [], result = []) {
    arr.forEach(v => {
        if(Array.isArray(v)) {
            result = result.concat(flat(v, []))
        }else {
            result.push(v)
        }
    })
    return result
}
```

数组对象格式化

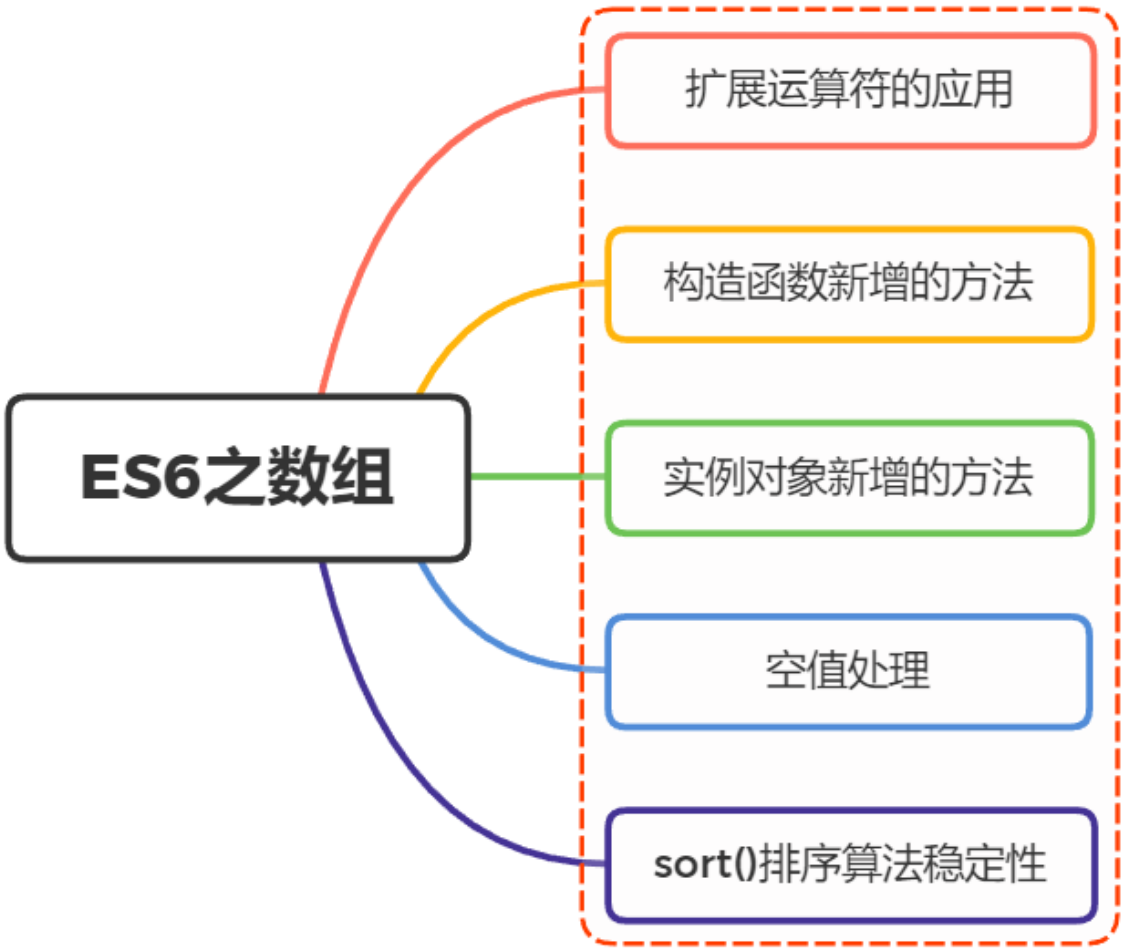
```
let obj = {
    a: '1',
    b: {
        c: '2',
        D: {
            E: '3'
        }
    }
}
// 转化为如下：
let obj = {
    a: '1',
    b: {
        c: '2',
        d: {
            e: '3'
        }
    }
}

// 代码实现
function keysLower(obj) {
    let reg = new RegExp("([A-Z]+)", "g");
    for (let key in obj) {
        if (obj.hasOwnProperty(key)) {
            let temp = obj[key];
            if (reg.test(key.toString())) {
                // 将修改后的属性名重新赋值给temp，并在对象obj内添加一个转换后的属性
                temp = obj[key.replace(reg, function (result) {
                    return result.toLowerCase()
```

```
                })] = obj[key];
                // 将之前大写的键属性删除
                delete obj[key];
            }
            // 如果属性是对象或者数组，重新执行函数
            if (typeof temp === 'object' || Object.prototype.toString.call(temp)
=== '[object Array]') {
                keysLower(temp);
            }
        }
    }
    return obj;
};
```

参考文献

- https://zh.wikipedia.org/wiki/%E5%B0%BE%E8%B0%83%E7%94%A8

07.数组新增了哪些扩展?



一、扩展运算符的应用

ES6通过扩展元素符`...`，好比`rest`参数的逆运算，将一个数组转为用逗号分隔的参数序列

```
console.log(...[1, 2, 3])
// 1 2 3

console.log(1, ...[2, 3, 4], 5)
// 1 2 3 4 5

[...document.querySelectorAll('div')]
// [<div>, <div>, <div>]
```

主要用于函数调用的时候，将一个数组变为参数序列

```
function push(array, ...items) {
  array.push(...items);
}

function add(x, y) {
  return x + y;
}

const numbers = [4, 38];
add(...numbers) // 42
```

可以将某些数据结构转为数组

```
[...document.querySelectorAll('div')]
```

能够更简单实现数组复制

```
const a1 = [1, 2];
const [...a2] = a1;
// [1,2]
```

数组的合并也更为简洁了

```
const arr1 = ['a', 'b'];
const arr2 = ['c'];
const arr3 = ['d', 'e'];
[...arr1, ...arr2, ...arr3]
// [ 'a', 'b', 'c', 'd', 'e' ]
```

注意：通过扩展运算符实现的是浅拷贝，修改了引用指向的值，会同步反映到新数组

下面看个例子就清楚多了

```
const arr1 = ['a', 'b',[1,2]];
const arr2 = ['c'];
const arr3  = [...arr1,...arr2]
arr[1][0] = 9999 // 修改arr1里面数组成员值
console.log(arr[3]) // 影响到arr3,['a','b',[9999,2],'c']
```

扩展运算符可以与解构赋值结合起来，用于生成数组

```
const [first, ...rest] = [1, 2, 3, 4, 5];
first // 1
rest  // [2, 3, 4, 5]

const [first, ...rest] = [];
first // undefined
rest  // []

const [first, ...rest] = ["foo"];
first  // "foo"
rest   // []
```

如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错

```
const [...butLast, last] = [1, 2, 3, 4, 5];
// 报错

const [first, ...middle, last] = [1, 2, 3, 4, 5];
// 报错
```

可以将字符串转为真正的数组

```
[...'hello']
// [ "h", "e", "l", "l", "o" ]
```

定义了遍历器（Iterator）接口的对象，都可以用扩展运算符转为真正的数组

```
let nodeList = document.querySelectorAll('div');
let array = [...nodeList];

let map = new Map([
  [1, 'one'],
  [2, 'two'],
  [3, 'three'],
]);

let arr = [...map.keys()]; // [1, 2, 3]
```

如果对没有 Iterator 接口的对象，使用扩展运算符，将会报错

```
const obj = {a: 1, b: 2};
let arr = [...obj]; // TypeError: Cannot spread non-iterable object
```

二、构造函数新增的方法

关于构造函数，数组新增的方法有如下：

- Array.from()
- Array.of()

Array.from()

将两类对象转为真正的数组：类似数组的对象和可遍历（iterable）的对象（包括 ES6 新增的数据结构 Set 和 Map）

```
let arrayLike = {
    '0': 'a',
    '1': 'b',
    '2': 'c',
    length: 3
};
let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']
```

还可以接受第二个参数，用来对每个元素进行处理，将处理后的值放入返回的数组

```
Array.from([1, 2, 3], (x) => x * x)
// [1, 4, 9]
```

Array.of()

用于将一组值，转换为数组

```
Array.of(3, 11, 8) // [3,11,8]
```

没有参数的时候，返回一个空数组

当参数只有一个的时候，实际上是指定数组的长度

参数个数不少于 2 个时，Array()才会返回由参数组成的新数组

```
Array() // []
Array(3) // [, , ,]
Array(3, 11, 8) // [3, 11, 8]
```

三、实例对象新增的方法

关于数组实例对象新增的方法有如下：

- copyWithin()
- find()、findIndex()
- fill()
- entries()，keys()，values()
- includes()
- flat()，flatMap()

copyWithin()

将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组

参数如下：

- target（必需）：从该位置开始替换数据。如果为负值，表示倒数。
- start（可选）：从该位置开始读取数据，默认为 0。如果为负值，表示从末尾开始计算。

- end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示从末尾开始计算。

```
[1, 2, 3, 4, 5].copyWithin(0, 3) // 将从 3 号位直到数组结束的成员（4 和 5），复制到从 0 号位开始的位置，结果覆盖了原来的 1 和 2
// [4, 5, 3, 4, 5]
```

find()、findIndex()

`find()`用于找出第一个符合条件的数组成员

参数是一个回调函数，接受三个参数依次为当前的值、当前的位置和原数组

```
[1, 5, 10, 15].find(function(value, index, arr) {
  return value > 9;
}) // 10
```

`findIndex`返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回`-1`

```
[1, 5, 10, 15].findIndex(function(value, index, arr) {
  return value > 9;
}) // 2
```

这两个方法都可以接受第二个参数，用来绑定回调函数的`this`对象。

```
function f(v){
  return v > this.age;
}
let person = {name: 'John', age: 20};
[10, 12, 26, 15].find(f, person);    // 26
```

fill()

使用给定值，填充一个数组

```
['a', 'b', 'c'].fill(7)
// [7, 7, 7]

new Array(3).fill(7)
// [7, 7, 7]
```

还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置

```
['a', 'b', 'c'].fill(7, 1, 2)
// ['a', 7, 'c']
```

注意，如果填充的类型为对象，则是浅拷贝

entries()，keys()，values()

`keys()`是对键名的遍历、`values()`是对键值的遍历，`entries()`是对键值对的遍历

```
or (let index of ['a', 'b'].keys()) {
  console.log(index);
}
// 0
// 1

for (let elem of ['a', 'b'].values()) {
  console.log(elem);
}
// 'a'
// 'b'

for (let [index, elem] of ['a', 'b'].entries()) {
  console.log(index, elem);
}
// 0 "a"
```

includes()

用于判断数组是否包含给定的值

```
[1, 2, 3].includes(2)     // true
[1, 2, 3].includes(4)     // false
[1, 2, NaN].includes(NaN) // true
```

方法的第二个参数表示搜索的起始位置，默认为`0`

参数为负数则表示倒数的位置

```
[1, 2, 3].includes(3, 3);  // false
[1, 2, 3].includes(3, -1); // true
```

flat()，flatMap()

将数组扁平化处理，返回一个新数组，对原数据没有影响

```
[1, 2, [3, 4]].flat()
// [1, 2, 3, 4]
```

`flat()`默认只会“拉平”一层，如果想要“拉平”多层的嵌套数组，可以将`flat()`方法的参数写成一个整数，表示想要拉平的层数，默认为1

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

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

`flatMap()`方法对原数组的每个成员执行一个函数相当于执行`Array.prototype.map()`，然后对返回值组成的数组执行`flat()`方法。该方法返回一个新数组，不改变原数组

```
// 相当于 [[2, 4], [3, 6], [4, 8]].flat()
[2, 3, 4].flatMap((x) => [x, x * 2])
// [2, 4, 3, 6, 4, 8]
```

`flatMap()`方法还可以有第二个参数，用来绑定遍历函数里面的`this`

四、数组的空位

数组的空位指，数组的某一个位置没有任何值

ES6 则是明确将空位转为`undefined`，包括`Array.from`、扩展运算符、`copyWithin()`、`fill()`、`entries()`、`keys()`、`values()`、`find()`和`findIndex()`

建议大家在日常书写中，避免出现空位

五、排序稳定性

将`sort()`默认设置为稳定的排序算法

```
const arr = [
  'peach',
  'straw',
  'apple',
  'spork'
];

const stableSorting = (s1, s2) => {
  if (s1[0] < s2[0]) return -1;
  return 1;
};

arr.sort(stableSorting)
// ["apple", "peach", "straw", "spork"]
```

排序结果中，`straw`在`spork`的前面，跟原始顺序一致

参考文献

- https://es6.ruanyifeng.com/#docs/array

08.对象新增了哪些扩展?

一、属性的简写

ES6中，当对象键名与对应值名相等的时候，可以进行简写

```
const baz = {foo:foo}

// 等同于
const baz = {foo}
```

方法也能够进行简写

```
const o = {
  method() {
    return "Hello!";
  }
};

// 等同于

const o = {
  method: function() {
    return "Hello!";
  }
}
```

在函数内作为返回值，也会变得方便很多

```
function getPoint() {
  const x = 1;
  const y = 10;
  return {x, y};
}

getPoint()
// {x:1, y:10}
```

注意：简写的对象方法不能用作构造函数，否则会报错

```
const obj = {
  f() {
    this.foo = 'bar';
  }
};

new obj.f() // 报错
```

二、属性名表达式

ES6 允许字面量定义对象时，将表达式放在括号内

```
let lastWord = 'last word';

const a = {
  'first word': 'hello',
  [lastWord]: 'world'
};

a['first word'] // "hello"
a[lastWord] // "world"
a['last word'] // "world"
```

表达式还可以用于定义方法名

```
let obj = {
  ['h' + 'ello']() {
    return 'hi';
  }
};

obj.hello() // hi
```

注意，属性名表达式与简洁表示法，不能同时使用，会报错

```
// 报错
const foo = 'bar';
const bar = 'abc';
const baz = { [foo] };

// 正确
const foo = 'bar';
const baz = { [foo]: 'abc'};
```

注意，属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串`[object Object]`

```
const keyA = {a: 1};
const keyB = {b: 2};

const myObject = {
  [keyA]: 'valueA',
  [keyB]: 'valueB'
};

myObject // Object {[object Object]: "valueB"}
```

三、super关键字

`this`关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字`super`，指向当前对象的原型对象

```
const proto = {
  foo: 'hello'
};

const obj = {
  foo: 'world',
  find() {
    return super.foo;
  }
};

Object.setPrototypeOf(obj, proto); // 为obj设置原型对象
obj.find() // "hello"
```

四、扩展运算符的应用

在解构赋值中，未被读取的可遍历的属性，分配到指定的对象上面

```
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
x // 1
y // 2
z // { a: 3, b: 4 }
```

注意：解构赋值必须是最后一个参数，否则会报错

解构赋值是浅拷贝

```
let obj = { a: { b: 1 } };
let { ...x } = obj;
obj.a.b = 2; // 修改obj里面a属性中键值
x.a.b // 2，影响到了结构出来x的值
```

对象的扩展运算符等同于使用`Object.assign()`方法

五、属性的遍历

ES6 一共有 5 种方法可以遍历对象的属性。

- for...in：循环遍历对象自身的和继承的可枚举属性（不含 Symbol 属性）
- Object.keys(obj)：返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含 Symbol 属性）的键名
- Object.getOwnPropertyNames(obj)：回一个数组，包含对象自身的所有属性（不含 Symbol 属性，但是包括不可枚举属性）的键名
- Object.getOwnPropertySymbols(obj)：返回一个数组，包含对象自身的所有 Symbol 属性的键名
- Reflect.ownKeys(obj)：返回一个数组，包含对象自身的（不含继承的）所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举

上述遍历，都遵守同样的属性遍历的次序规则：

- 首先遍历所有数值键，按照数值升序排列
- 其次遍历所有字符串键，按照加入时间升序排列
- 最后遍历所有 Symbol 键，按照加入时间升序排

```
Reflect.ownKeys({ [Symbol()]:0, b:0, 10:0, 2:0, a:0 })
// ['2', '10', 'b', 'a', Symbol()]
```

六、对象新增的方法

关于对象新增的方法，分别有以下：

- Object.is()
- Object.assign()
- Object.getOwnPropertyDescriptors()
- Object.setPrototypeOf()，Object.getPrototypeOf()

- Object.keys()，Object.values()，Object.entries()
- Object.fromEntries()

Object.is()

严格判断两个值是否相等，与严格比较运算符（===）的行为基本一致，不同之处只有两个：一是`+0`不等于`-0`，二是`NaN`等于自身

```
+0 === -0 //true
NaN === NaN // false

Object.is(+0, -0) // false
Object.is(NaN, NaN) // true
```

Object.assign()

`Object.assign()`方法用于对象的合并，将源对象`source`的所有可枚举属性，复制到目标对象`target`

`Object.assign()`方法的第一个参数是目标对象，后面的参数都是源对象

```
const target = { a: 1, b: 1 };

const source1 = { b: 2, c: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
target // {a:1, b:2, c:3}
```

注意：`Object.assign()`方法是浅拷贝，遇到同名属性会进行替换

Object.getOwnPropertyDescriptors()

返回指定对象所有自身属性（非继承属性）的描述对象

```
const obj = {
  foo: 123,
  get bar() { return 'abc' }
};

Object.getOwnPropertyDescriptors(obj)
// { foo:
//    { value: 123,
//      writable: true,
//      enumerable: true,
//      configurable: true },
//   bar:
//    { get: [Function: get bar],
//      set: undefined,
//      enumerable: true,
//      configurable: true } }
```

Object.setPrototypeOf()

`Object.setPrototypeOf`方法用来设置一个对象的原型对象

```
Object.setPrototypeOf(object, prototype)

// 用法
const o = Object.setPrototypeOf({}, null);
```

Object.getPrototypeOf()

用于读取一个对象的原型对象

```
Object.getPrototypeOf(obj);
```

Object.keys()

返回自身的（不含继承的）所有可遍历（enumerable）属性的键名的数组

```
var obj = { foo: 'bar', baz: 42 };
Object.keys(obj)
// ["foo", "baz"]
```

Object.values()

返回自身的（不含继承的）所有可遍历（enumerable）属性的键对应值的数组

```
const obj = { foo: 'bar', baz: 42 };
Object.values(obj)
// ["bar", 42]
```

Object.entries()

返回一个对象自身的（不含继承的）所有可遍历（enumerable）属性的键值对的数组

```
const obj = { foo: 'bar', baz: 42 };
Object.entries(obj)
// [ ["foo", "bar"], ["baz", 42] ]
```

Object.fromEntries()

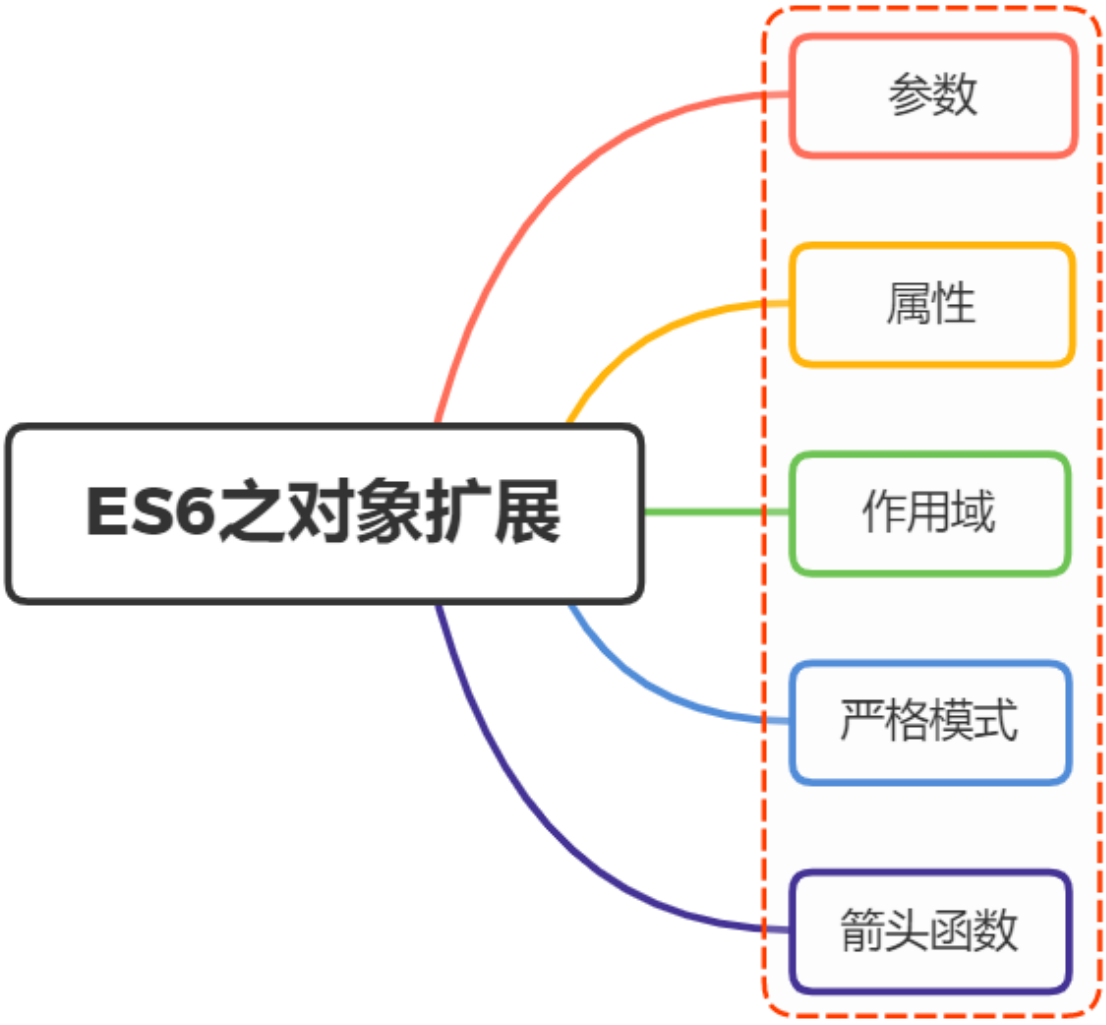
用于将一个键值对数组转为对象

```
Object.fromEntries([
  ['foo', 'bar'],
  ['baz', 42]
])
// { foo: "bar", baz: 42 }
```

参考文献

- https://es6.ruanyifeng.com/#docs/object

09.函数新增了哪些扩展?



一、参数

ES6 允许为函数的参数设置默认值

```
function log(x, y = 'World') {
  console.log(x, y);
}

console.log('Hello') // Hello World
console.log('Hello', 'China') // Hello China
console.log('Hello', '') // Hello
```

函数的形参是默认声明的，不能使用`let`或`const`再次声明

```
function foo(x = 5) {
    let x = 1; // error
    const x = 2; // error
}
```

参数默认值可以与解构赋值的默认值结合起来使用

```
function foo({x, y = 5}) {
  console.log(x, y);
}

foo({}) // undefined 5
foo({x: 1}) // 1 5
foo({x: 1, y: 2}) // 1 2
foo() // TypeError: Cannot read property 'x' of undefined
```

上面的`foo`函数，当参数为对象的时候才能进行解构，如果没有提供参数的时候，变量`x`和`y`就不会生成，从而报错，这里设置默认值避免

```
function foo({x, y = 5} = {}) {
  console.log(x, y);
}

foo() // undefined 5
```

参数默认值应该是函数的尾参数，如果不是非尾部的参数设置默认值，实际上这个参数是没发省略的

```
function f(x = 1, y) {
  return [x, y];
}

f() // [1, undefined]
f(2) // [2, undefined]
f(, 1) // 报错
f(undefined, 1) // [1, 1]
```

二、属性

函数的length属性

`length`将返回没有指定默认值的参数个数

```
(function (a) {}).length // 1
(function (a = 5) {}).length // 0
(function (a, b, c = 5) {}).length // 2
```

`rest` 参数也不会计入 `length` 属性

```
(function(...args) {}).length // 0
```

如果设置了默认值的参数不是尾参数，那么 `length` 属性也不再计入后面的参数了

```
(function (a = 0, b, c) {}).length // 0
(function (a, b = 1, c) {}).length // 1
```

name属性

返回该函数的函数名

```
var f = function () {};

// ES5
f.name // ""

// ES6
f.name // "f"
```

如果将一个具名函数赋值给一个变量，则 `name` 属性都返回这个具名函数原本的名字

```
const bar = function baz() {};
bar.name // "baz"
```

`Function` 构造函数返回的函数实例， `name` 属性的值为 `anonymous`

```
(new Function).name // "anonymous"
```

`bind` 返回的函数， `name` 属性值会加上 `bound` 前缀

```
function foo() {};
foo.bind({}).name // "bound foo"

(function(){}).bind({}).name // "bound "
```

三、作用域

一旦设置了参数的默认值，函数进行声明初始化时，参数会形成一个单独的作用域

等到初始化结束，这个作用域就会消失。这种语法行为，在不设置参数默认值时，是不会出现的

下面例子中， `y=x` 会形成一个单独作用域， `x` 没有被定义，所以指向全局变量 `x`

```
let x = 1;

function f(y = x) {
  // 等同于 let y = x
  let x = 2;
  console.log(y);
}

f() // 1
```

四、严格模式

只要函数参数使用了默认值、解构赋值、或者扩展运算符，那么函数内部就不能显式设定为严格模式，否则会报错

```
// 报错
function doSomething(a, b = a) {
  'use strict';
  // code
}

// 报错
const doSomething = function ({a, b}) {
  'use strict';
  // code
};

// 报错
const doSomething = (...a) => {
  'use strict';
  // code
};

const obj = {
  // 报错
  doSomething({a, b}) {
    'use strict';
    // code
  }
};
```

五、箭头函数

使用“箭头”（`=>`）定义函数

```
var f = v => v;

// 等同于
var f = function (v) {
  return v;
};
```

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分

```
var f = () => 5;
// 等同于
var f = function () { return 5 };

var sum = (num1, num2) => num1 + num2;
// 等同于
var sum = function(num1, num2) {
  return num1 + num2;
};
```

如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用`return`语句返回

```
var sum = (num1, num2) => { return num1 + num2; }
```

如果返回对象，需要加括号将对象包裹

```
let getTempItem = id => ({ id: id, name: "Temp" });
```

注意点：

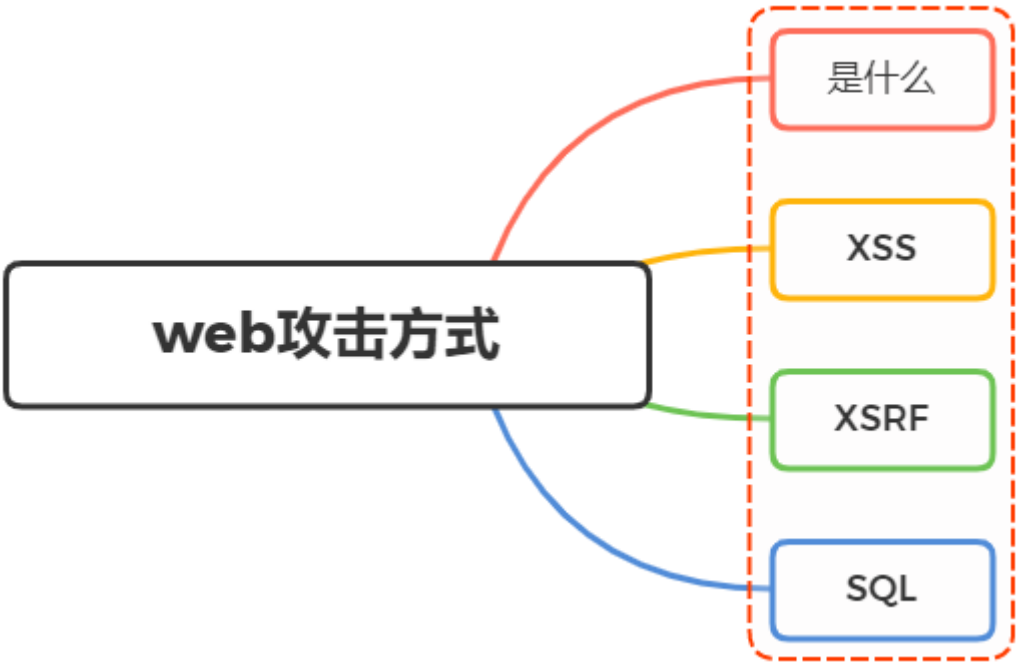
- 函数体内的`this`对象，就是定义时所在的对象，而不是使用时所在的对象
- 不可以当作构造函数，也就是说，不可以使用`new`命令，否则会抛出一个错误
- 不可以使用`arguments`对象，该对象在函数体内不存在。如果要用，可以用`rest`参数代替
- 不可以使用`yield`命令，因此箭头函数不能用作 Generator 函数

参考文献

- https://es6.ruanyifeng.com/#docs/function

10.web常见的攻击方式有哪些？如何防御？

面试官：web常见的攻击方式有哪些？如何防御？



一、是什么

Web攻击（WebAttack）是针对用户上网行为或网站服务器等设备进行攻击的行为

如植入恶意代码，修改网站权限，获取网站用户隐私信息等等

Web应用程序的安全性是任何基于Web业务的重要组成部分

确保Web应用程序安全十分重要，即使是代码中很小的 bug 也有可能导致隐私信息被泄露

站点安全就是为保护站点不受未授权的访问、使用、修改和破坏而采取的行为或实践

我们常见的Web攻击方式有

- XSS (Cross Site Scripting) 跨站脚本攻击
- CSRF（Cross-site request forgery）跨站请求伪造
- SQL注入攻击

二、XSS

XSS，跨站脚本攻击，允许攻击者将恶意代码植入到提供给其它用户使用的页面中

`xss`涉及到三方，即攻击者、客户端与`web`应用

`xss`的攻击目标是为了盗取存储在客户端的`cookie`或者其他网站用于识别客户端身份的敏感信息。一旦获取到合法用户的信息后，攻击者甚至可以假冒合法用户与网站进行交互

举个例子：

一个搜索页面，根据`url`参数决定关键词的内容

```
<input type="text" value="<%= getParameter("keyword") %>">
<button>搜索</button>
<div>
  您搜索的关键词是：<%= getParameter("keyword") %>
</div>
```

这里看似并没有问题，但是如果不按套路出牌呢？

用户输入`"><script>alert('XSS');</script>`，拼接到 HTML 中返回给浏览器。形成了如下的 HTML：

```
<input type="text" value=""><script>alert('XSS');</script>">
<button>搜索</button>
<div>
  您搜索的关键词是："><script>alert('XSS');</script>
</div>
```

浏览器无法分辨出 `<script>alert('XSS');</script>` 是恶意代码，因而将其执行，试想一下，如果是获取 `cookie` 发送对黑客服务器呢？

根据攻击的来源，`XSS` 攻击可以分成：

- 存储型
- 反射型
- DOM 型

存储型

存储型 XSS 的攻击步骤：

1. 攻击者将恶意代码提交到目标网站的数据库中
2. 用户打开目标网站时，网站服务端将恶意代码从数据库取出，拼接在 HTML 中返回给浏览器
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

这种攻击常见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信等

反射型 XSS

反射型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码
2. 用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

反射型 XSS 跟存储型 XSS 的区别是：存储型 XSS 的恶意代码存在数据库里，反射型 XSS 的恶意代码存在 URL 里。

反射型 XSS 漏洞常见于通过 URL 传递参数的功能，如网站搜索、跳转等。

由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。

POST 的内容也可以触发反射型 XSS，只不过其触发条件比较苛刻（需要构造表单提交页面，并引导用户点击），所以非常少见

DOM 型 XSS

DOM 型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码
2. 用户打开带有恶意代码的 URL
3. 用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞

XSS的预防

通过前面介绍，看到`XSS`攻击的两大要素：

- 攻击者提交而恶意代码
- 浏览器执行恶意代码

针对第一个要素，我们在用户输入的过程中，过滤掉用户输入的恶劣代码，然后提交给后端，但是如果攻击者绕开前端请求，直接构造请求就不能预防了

而如果在后端写入数据库前，对输入进行过滤，然后把内容给前端，但是这个内容在不同地方就会有不同显示

例如：

一个正常的用户输入了 `5 < 7` 这个内容，在写入数据库前，被转义，变成了 `5 < 7`

在客户端中，一旦经过了 `escapeHTML()`，客户端显示的内容就变成了乱码(`5 < 7`)

在前端中，不同的位置所需的编码也不同。

- 当 `5 < 7` 作为 HTML 拼接页面时，可以正常显示：

```
<div title="comment">5 &lt; 7</div>
```

- 当 `5 < 7` 通过 Ajax 返回，然后赋值给 JavaScript 的变量时，前端得到的字符串就是转义后的字符。这个内容不能直接用于 Vue 等模板的展示，也不能直接用于内容长度计算。不能用于标题、alert 等

可以看到，过滤并非可靠的，下面就要通过防止浏览器执行恶意代码：

在使用 `.innerHTML`、`.outerHTML`、`document.write()` 时要特别小心，不要把不可信的数据作为 HTML 插到页面上，而应尽量使用 `.textContent`、`.setAttribute()` 等

如果用 `Vue/React` 技术栈，并且不使用 `v-html`/`dangerouslySetInnerHTML` 功能，就在前端 `render` 阶段避免 `innerHTML`、`outerHTML` 的 XSS 隐患

DOM 中的内联事件监听器，如 `location`、`onclick`、`onerror`、`onload`、`onmouseover` 等，`<a>` 标签的 `href` 属性，JavaScript 的 `eval()`、`setTimeout()`、`setInterval()` 等，都能把字符串作为代码运行。如果不可信的数据拼接到字符串中传递给这些 API，很容易产生安全隐患，请务必避免

```
<!-- 链接内包含恶意代码 -->
< a href=" ">1</ a>

<script>
// setTimeout()/setInterval() 中调用恶意代码
setTimeout("UNTRUSTED")
setInterval("UNTRUSTED")

// location 调用恶意代码
location.href = 'UNTRUSTED'

// eval() 中调用恶意代码
eval("UNTRUSTED")
```

三、CSRF

CSRF（Cross-site request forgery）跨站请求伪造：攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发送跨站请求

利用受害者在被攻击网站已经获取的注册凭证，绕过后台的用户验证，达到冒充用户对被攻击的网站执行某项操作的目

一个典型的CSRF攻击有着如下的流程：

- 受害者登录a.com，并保留了登录凭证（Cookie）
- 攻击者引诱受害者访问了b.com
- b.com 向 a.com 发送了一个请求：a.com/act=xx。浏览器会默认携带a.com的Cookie
- a.com接收到请求后，对请求进行验证，并确认是受害者的凭证，误以为是受害者自己发送的请求
- a.com以受害者的名义执行了act=xx
- 攻击完成，攻击者在受害者不知情的情况下，冒充受害者，让a.com执行了自己定义的操作

`csrf`可以通过`get`请求，即通过访问`img`的页面后，浏览器自动访问目标地址，发送请求

同样，也可以设置一个自动提交的表单发送`post`请求，如下：

```
<form action="http://bank.example/withdraw" method=POST>
    <input type="hidden" name="account" value="xiaoming" />
    <input type="hidden" name="amount" value="10000" />
    <input type="hidden" name="for" value="hacker" />
</form>
<script> document.forms[0].submit(); </script>
```

访问该页面后，表单会自动提交，相当于模拟用户完成了一次`POST`操作

还有一种为使用`a`标签的，需要用户点击链接才会触发

访问该页面后，表单会自动提交，相当于模拟用户完成了一次POST操作

```
< a href="http://test.com/csrf/withdraw.php?amount=1000&for=hacker"
taget="_blank">
    重磅消息！！
<a/>
```

CSRF的特点

- 攻击一般发起在第三方网站，而不是被攻击的网站。被攻击的网站无法防止攻击发生
- 攻击利用受害者在被攻击网站的登录凭证，冒充受害者提交操作；而不是直接窃取数据
- 整个过程攻击者并不能获取到受害者的登录凭证，仅仅是“冒用”
- 跨站请求可以用各种方式：图片URL、超链接、CORS、Form提交等等。部分请求方式可以直接嵌入在第三方论坛、文章中，难以进行追踪

CSRF的预防

CSRF通常从第三方网站发起，被攻击的网站无法防止攻击发生，只能通过增强自己网站针对CSRF的防护能力来提升安全性

防止`csrf`常用方案如下：

- 阻止不明外域的访问
  - 同源检测
  - Samesite Cookie
- 提交时要求附加本域才能获取的信息
  - CSRF Token
  - 双重Cookie验证

这里主要讲讲`token`这种形式，流程如下：

- 用户打开页面的时候，服务器需要给这个用户生成一个Token
- 对于GET请求，Token将附在请求地址之后。对于 POST 请求来说，要在 form 的最后加上

```
<input type=”hidden” name=”csrftoken” value=”tokenvalue”/>
```

- 当用户从客户端得到了Token，再次提交给服务器的时候，服务器需要判断Token的有效性

四、SQL注入

Sql 注入攻击，是通过将恶意的`Sql`查询或添加语句插入到应用的输入参数中，再在后台`Sql`服务器上解析执行进行的攻击

流程如下所示：

- 找出SQL漏洞的注入点
- 判断数据库的类型以及版本
- 猜解用户名和密码
- 利用工具查找Web后台管理入口
- 入侵和破坏

预防方式如下：

- 严格检查输入变量的类型和格式
- 过滤和转义特殊字符
- 对访问数据库的Web应用程序采用Web应用防火墙

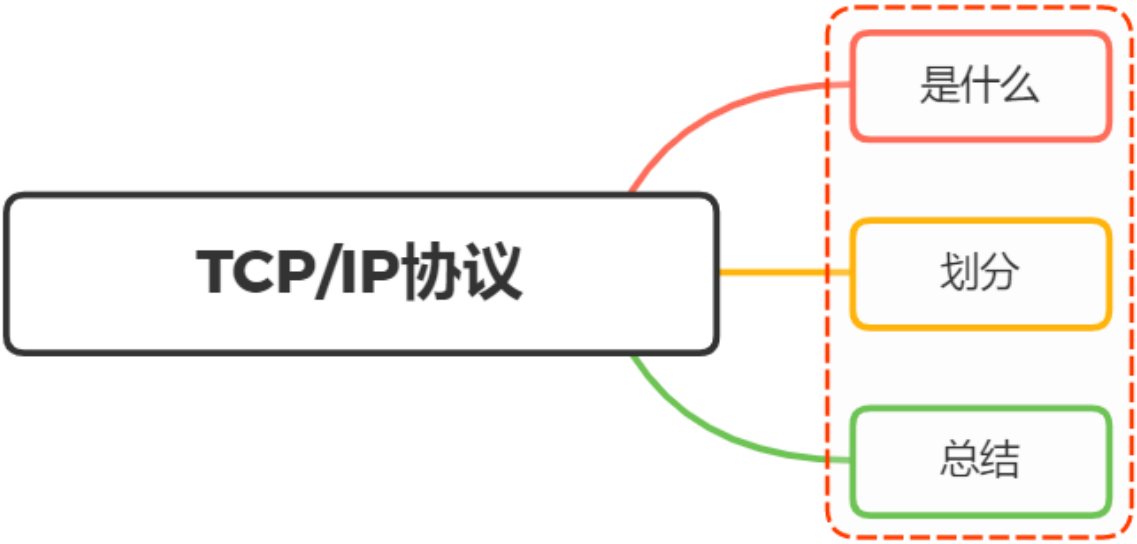
上述只是列举了常见的`web`攻击方式，实际开发过程中还会遇到很多安全问题，对于这些问题， 切记不可忽视

参考文献

- https://tech.meituan.com/2018/09/27/fe-security.html
- https://developer.mozilla.org/zh-CN/docs/learn/Server-side/First_steps/Website_security

6.ajax

01.如何理解TCP/IP协议



一、是什么

TCP/IP，**传输控制协议/网际协议**，是指能够在多个不同网络间实现信息传输的协议簇

- TCP（传输控制协议）

一种面向连接的、可靠的、基于字节流的传输层通信协议

- IP（网际协议）

用于封包交换数据网络的协议

TCP/IP协议不仅仅指的是`TCP`和`IP`两个协议，而是指一个由`FTP`、`SMTP`、`TCP`、`UDP`、`IP`等协议构成的协议簇，

只是因为在`TCP/IP`协议中`TCP`协议和`IP`协议最具代表性，所以通称为TCP/IP协议族（英语：TCP/IP Protocol Suite，或TCP/IP Protocols）

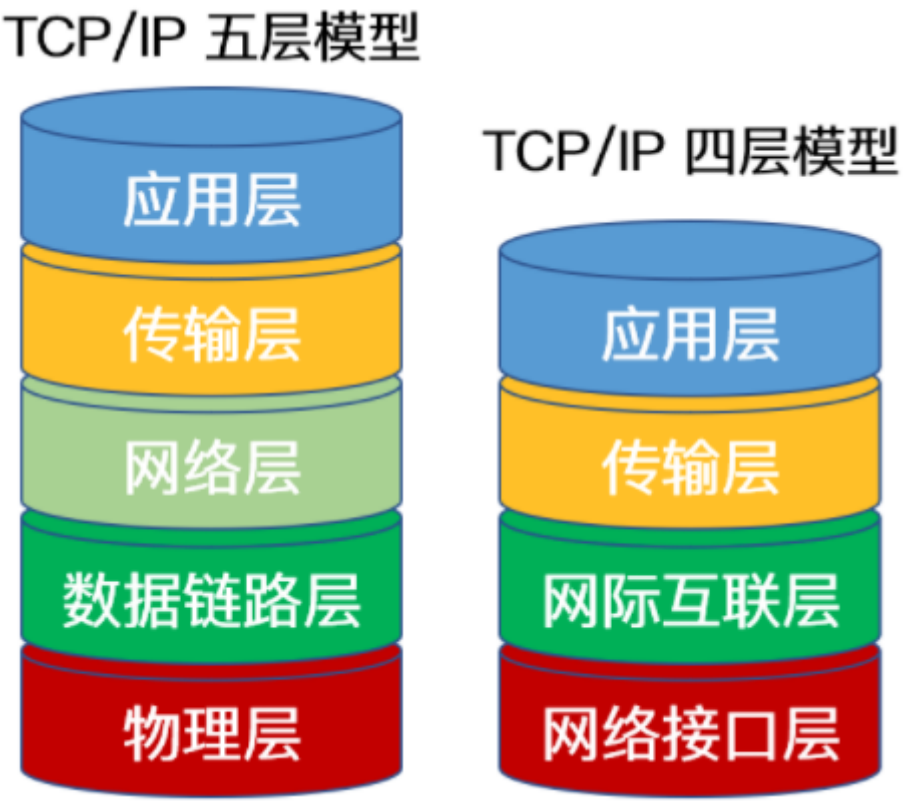
二、划分

TCP/IP协议族按层次分别了五层体系或者四层体系

五层体系的协议结构是综合了 OSI 和 TCP/IP 优点的一种协议，包括应用层、传输层、网络层、数据链路层和物理层

五层协议的体系结构只是为介绍网络原理而设计的，实际应用还是 TCP/IP 四层体系结构，包括应用层、传输层、网络层（网际互联层）、网络接口层

如下图所示：



五层体系

应用层

`TCP/IP` 模型将 `OSI` 参考模型中的会话层、表示层和应用层的功能合并到一个应用层实现，通过不同的应用层协议为不同的应用提供服务

如：`FTP`、`Telnet`、`DNS`、`SMTP` 等

传输层

该层对应于 OSI 参考模型的传输层，为上层实体提供源端到对端主机的通信功能

传输层定义了两个主要协议：传输控制协议（TCP）和用户数据报协议（UDP）

其中面向连接的 TCP 协议保证了数据的传输可靠性，面向无连接的 UDP 协议能够实现数据包简单、快速地传输

网络层

负责为分组网络中的不同主机提供通信服务，并通过选择合适的路由将数据传递到目标主机

在发送数据时，网络层把运输层产生的报文段或用户数据封装成分组或包进行传送

数据链路层

数据链路层在两个相邻节点传输数据时，将网络层交下来的IP数据报组装成帧，在两个相邻节点之间的链路上传送帧

物理层

保数据可以在各种物理媒介上进行传输，为数据的传输提供可靠的环境

四层体系

TCP/IP 的四层结构则如下表所示：

层次名称	单位	功能	协议
网络接口层	帧	负责实际数据的传输，对应OSI参考模型的下两层	HDLC（高级链路控制协议）PPP（点对点协议）SLIP（串行线路接口协议）
网络层	数据报	负责网络间的寻址数据传输，对应OSI参考模型的第三层	IP（网际协议） ICMP（网际控制消息协议）ARP（地址解析协议） RARP（反向地址解析协议）
传输层	报文段	负责提供可靠的传输服务，对应OSI参考模型的第四层	TCP（控制传输协议） UDP（用户数据报协议）
应用层		负责实现一切与应用程序相关的功能，对应OSI参考模型的上三层	FTP（文件传输协议） HTTP（超文本传输协议）DNS（域名服务器协议）SMTP（简单邮件传输协议）NFS（网络文件系统协议）

三、总结

OSI 参考模型与 TCP/IP 参考模型区别如下：

相同点：

- OSI 参考模型与 TCP/IP 参考模型都采用了层次结构
- 都能够提供面向连接和无连接两种通信服务机制

不同点：

- OSI 采用的七层模型； TCP/IP 是四层或五层结构
- TCP/IP 参考模型没有对网络接口层进行细分，只是一些概念性的描述； OSI 参考模型对服务和协议做了明确的区分
- OSI 参考模型虽然网络划分为七层，但实现起来较困难。TCP/IP 参考模型作为一种简化的分层结构是可以的
- TCP/IP协议去掉表示层和会话层的原因在于会话层、表示层、应用层都是在应用程序内部实现的，最终产出的是一个应用数据包，而应用程序之间是几乎无法实现代码的抽象共享的，这也就造成 `OSI` 设想中的应用程序维度的分层是无法实现的

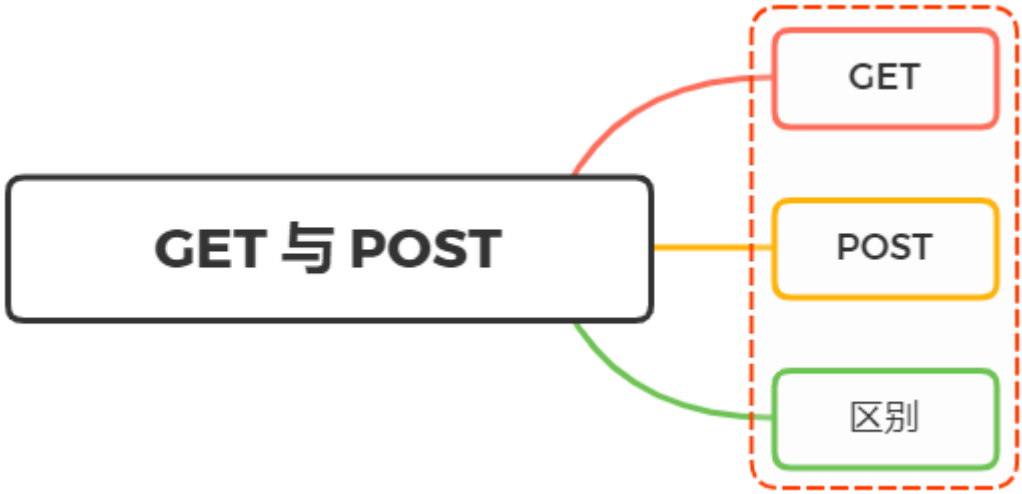
三种模型对应关系如下图所示：

区域	TCP/IP 四层模型	TCP/IP 五层模型	OSI 七层模型	单位	地址	功能	对应设备	协议
计算机高层	应用层	应用层	应用层	应用进程	进程号	应用程序与协议	应用程序（eg：FTP、HTTP）	FTP、NFS
			表示层			数据加密、压缩	编码解码、加密解密	Telnet、SNMP
			会话层			会话的开始、恢复、释放、同步	建立会话，session验证、断点传输	SMTP、DNS
网络低层	传输层	传输层	传输层	报文/数据段	端口号	端到端的可靠透明传输、保证数据完整性	进程与端口	TCP、UDP
	网络层	网络层	网络层	包/分组	IP地址	服务选择、路径选择、多路复用等（如何选择发送路径、方式）	路由器、防火墙、多层交换机	IP、ICMP、ARP
	网络接口层	数据链路层	数据链路层	帧	mac地址	差错控制、流量控制（规定如何进行01发送不会造成错误）	网卡、网桥、交换机	PPP、SLIP
		物理层	物理层	比特流	bit	光纤、电缆、双绞线连接，传送0/1电信号	中继器、集线器、网线	IEEExxxx

参考文献

- https://zh.wikipedia.org/wiki/TCP/IP%E5%8D%8F%E8%AE%AE%E6%97%8F
- https://zhuanlan.zhihu.com/p/103162095
- https://segmentfault.com/a/1190000039204681
- https://leetcode-cn.com/leetbook/detail/networks-interview-highlights/
- https://vue3js.cn/interview

02.说一下 GET 和 POST 的区别



一、是什么

`GET` 和 `POST`，两者是 `HTTP` 协议中发送请求的方法

GET

`GET`方法请求一个指定资源的表示形式，使用GET的请求应该只被用于获取数据

POST

`POST`方法用于将实体提交到指定的资源，通常导致在服务器上的状态变化或**副作用**

本质上都是`TCP`链接，并无差别

但是由于`HTTP`的规定和浏览器/服务器的限制，导致他们在应用过程中会体现出一些区别

二、区别

从`w3schools`得到的标准答案的区别如下：

- GET在浏览器回退时是无害的，而POST会再次提交请求。
- GET产生的URL地址可以被Bookmark，而POST不可以。
- GET请求会被浏览器主动cache，而POST不会，除非手动设置。
- GET请求只能进行url编码，而POST支持多种编码方式。
- GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
- GET请求在URL中传送的参数是有长度限制的，而POST没有。
- 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。
- GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
- GET参数通过URL传递，POST放在Request body中

参数位置

貌似从上面看到`GET`与`POST`请求区别非常大，但两者实质并没有区别

无论`GET`还是`POST`，用的都是同一个传输层协议，所以在传输上没有区别

当不携带参数的时候，两者最大的区别为第一行方法名不同

> POST /uri HTTP/1.1 \r\n
>
> GET /uri HTTP/1.1 \r\n

当携带参数的时候，我们都知道`GET`请求是放在`url`中，`POST`则放在`body`中

`GET` 方法简约版报文是这样的

```
GET /index.html?name=qiming.c&age=22 HTTP/1.1
Host: localhost
```

`POST`方法简约版报文是这样的

```
POST /index.html HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded

name=qiming.c&age=22
```

注意：这里只是约定，并不属于`HTTP`规范，相反的，我们可以在`POST`请求中`url`中写入参数，或者`GET`请求中的`body`携带参数

参数长度

`HTTP`协议没有`Body`和`URL`的长度限制，对`URL`限制的大多是浏览器和服务器的原因

`IE`对`URL`长度的限制是2083字节(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持

这里限制的是整个`URL`长度，而不仅仅是参数值的长度

服务器处理长`URL`要消耗比较多的资源，为了性能和安全考虑，会给`URL`长度加限制

安全

`POST`比`GET`安全，因为数据在地址栏上不可见

然而，从传输的角度来说，他们都是不安全的，因为`HTTP`在网络上是明文传输的，只要在网络节点上捉包，就能完整地获取数据报文

只有使用`HTTPS`才能加密安全

数据包

对于`GET`方式的请求，浏览器会把`http header`和`data`一并发送出去，服务器响应200（返回数据）

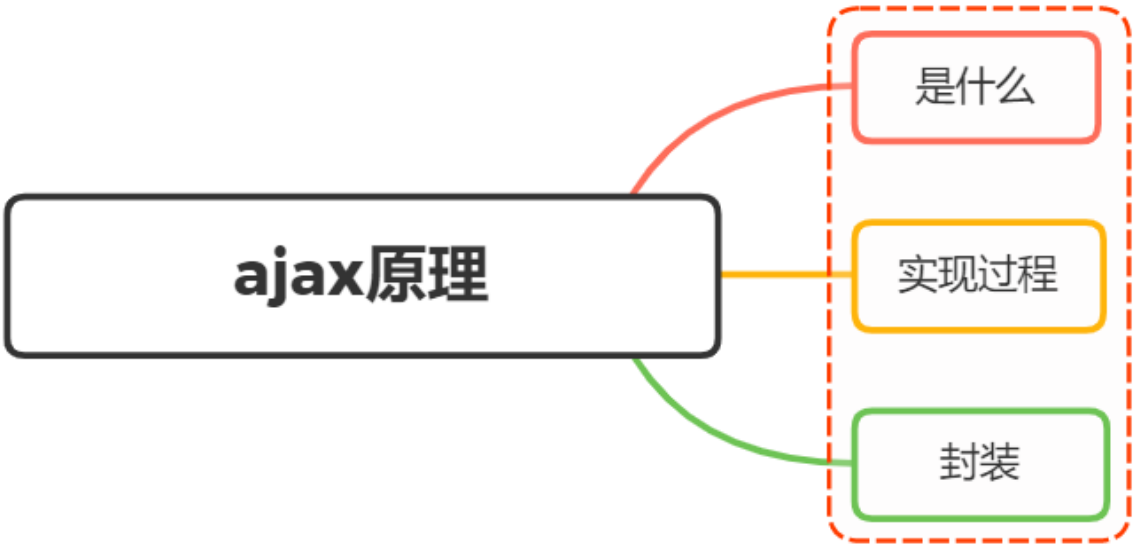
对于`POST`，浏览器先发送`header`，服务器响应100 `continue`，浏览器再发送`data`，服务器响应200 ok

并不是所有浏览器都会在`POST`中发送两次包，`Firefox`就只发送一次

参考文献

- https://mp.weixin.qq.com/s?__biz=MzI3NzIzMzg3Mw==&mid=100000054&idx=1&sn=71f6c214f3833d9ca20b9f7dcd9d33e4#rd
- https://blog.fundebug.com/2019/02/22/compare-http-method-get-and-post/
- https://www.w3school.com.cn/tags/html_ref_httpmethods.asp
- https://vue3js.cn/interview

03.ajax原理是什么？如何实现？



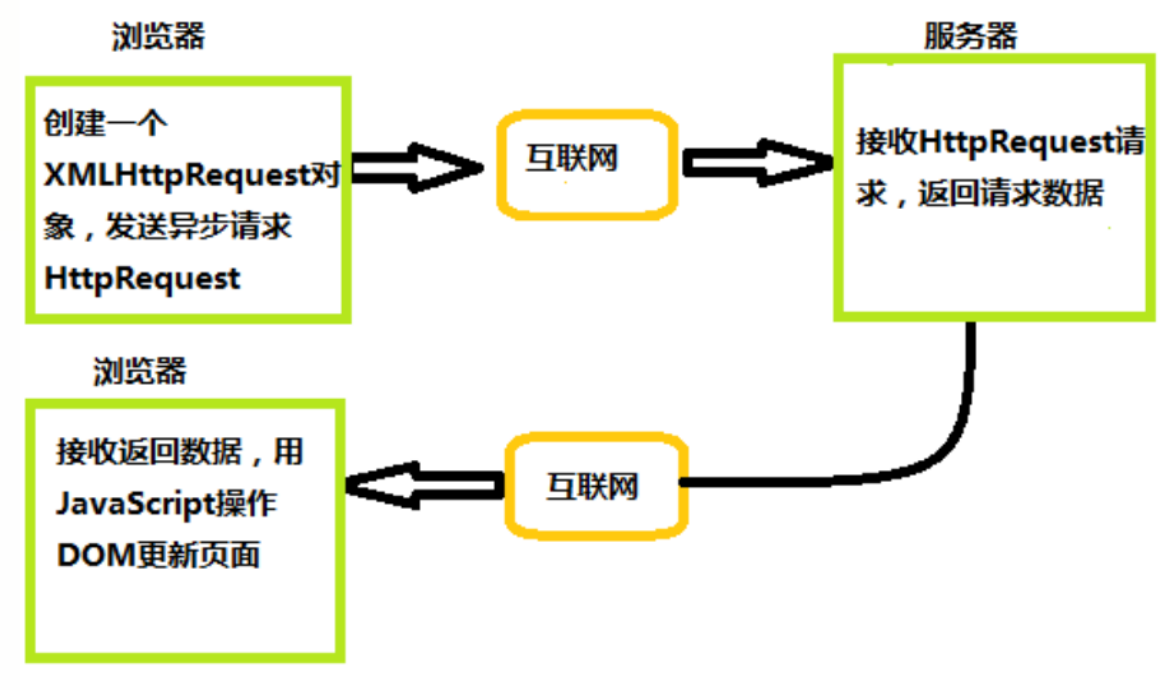
一、是什么

`AJAX`全称(Async Javascript and XML)

即异步的`JavaScript`和`XML`，是一种创建交互式网页应用的网页开发技术，可以在不重新加载整个网页的情况下，与服务器交换数据，并且更新部分网页

`Ajax`的原理简单来说通过`XmlHttpRequest`对象来向服务器发异步请求，从服务器获得数据，然后用`JavaScript`来操作`DOM`而更新页面

流程图如下：



下面举个例子：

领导想找小李汇报一下工作，就委托秘书去叫小李，自己就接着做其他事情，直到秘书告诉他小李已经到了，最后小李跟领导汇报工作

`Ajax`请求数据流程与“领导想找小李汇报一下工作”类似，上述秘书就相当于`XMLHttpRequest`对象，领导相当于浏览器，响应数据相当于小李

浏览器可以发送`HTTP`请求后，接着做其他事情，等收到`XHR`返回来的数据再进行操作

二、实现过程

实现`Ajax`异步交互需要服务器逻辑进行配合，需要完成以下步骤：

- 创建`Ajax`的核心对象`XMLHttpRequest`对象
- 通过`XMLHttpRequest`对象的`open()`方法与服务端建立连接
- 构建请求所需的数据内容，并通过`XMLHttpRequest`对象的`send()`方法发送给服务器端
- 通过`XMLHttpRequest`对象提供的`onreadystatechange`事件监听服务器端你的通信状态
- 接受并处理服务端向客户端响应的数据结果
- 将处理结果更新到`HTML`页面中

创建XMLHttpRequest对象

通过`XMLHttpRequest()`构造函数用于初始化一个`XMLHttpRequest`实例对象

```
const xhr = new XMLHttpRequest();
```

与服务器建立连接

通过 `XMLHttpRequest` 对象的 `open()` 方法与服务器建立连接

```
xhr.open(method, url, [async][, user][, password])
```

参数说明：

- `method`：表示当前的请求方式，常见的有`GET`、`POST`
- `url`：服务端地址
- `async`：布尔值，表示是否异步执行操作，默认为`true`
- `user`：可选的用户名用于认证用途；默认为`null
- `password`：可选的密码用于认证用途，默认为`null

给服务端发送数据

通过 `XMLHttpRequest` 对象的 `send()` 方法，将客户端页面的数据发送给服务端

```
xhr.send([body])
```

`body`：在 `XHR` 请求中要发送的数据体，如果不传递数据则为 `null`

如果使用`GET`请求发送数据的时候，需要注意如下：

- 将请求数据添加到`open()`方法中的`url`地址中
- 发送请求数据中的`send()`方法中参数设置为`null`

绑定onreadystatechange事件

`onreadystatechange` 事件用于监听服务器端的通信状态，主要监听的属性为 `XMLHttpRequest.readyState`，

关于`XMLHttpRequest.readyState`属性有五个状态，如下图显示

值	状态	描述
0	UNSENT(未打开)	open()方法还未被调用
1	OPENED(未发送)	send()方法还未被调用
2	HEADERS_RECEIVED(以获取响应头)	send()方法已经被调用，响应头和响应状态已经返回
3	LOADING(正在下载响应体)	响应体下载中；responseText中已经获取部分数据
4	DONE(请求完成)	整个请求过程已完毕

只要 `readyState`属性值一变化，就会触发一次 `readystatechange` 事件

`XMLHttpRequest.responseText`属性用于接收服务器端的响应结果

举个例子：

```
const request = new XMLHttpRequest()
request.onreadystatechange = function(e){
    if(request.readyState === 4){ // 整个请求过程完毕
        if(request.status >= 200 && request.status <= 300){
            console.log(request.responseText) // 服务端返回的结果
        }else if(request.status >=400){
            console.log("错误信息: " + request.status)
        }
    }
}
request.open('POST','http://xxxx')
request.send()
```

三、封装

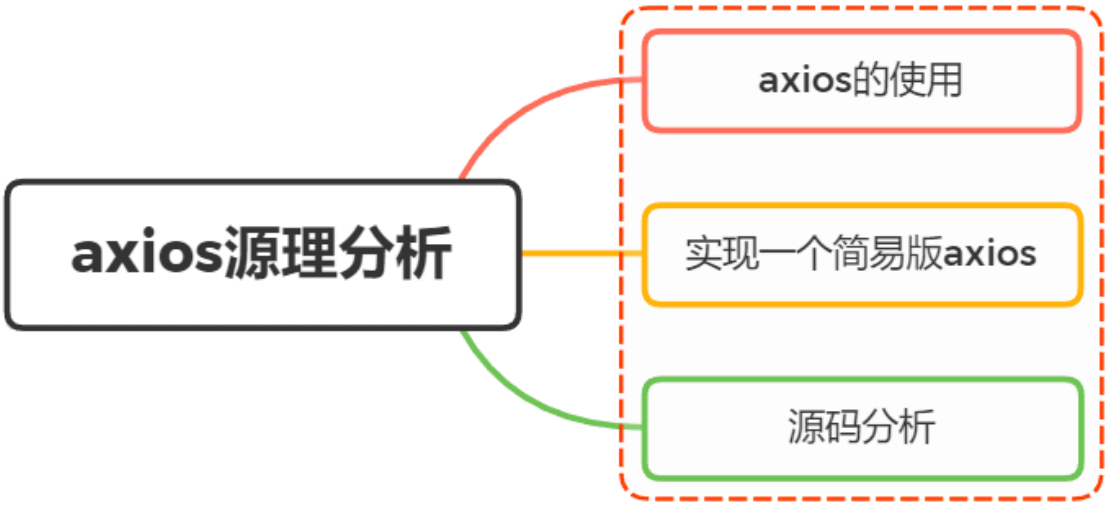
通过上面对`XMLHttpRequest`对象的了解，下面来封装一个简单的`ajax`请求

```
//封装一个ajax请求
function ajax(options) {
    //创建XMLHttpRequest对象
    const xhr = new XMLHttpRequest()


    //初始化参数的内容
    options = options || {}
    options.type = (options.type || 'GET').toUpperCase()
    options.dataType = options.dataType || 'json'
    const params = options.data

    //发送请求
    if (options.type === 'GET') {
        xhr.open('GET', options.url + '?' + params, true)
        xhr.send(null)
    } else if (options.type === 'POST') {
        xhr.open('POST', options.url, true)
        xhr.send(params)

    //接收请求
    xhr.onreadystatechange = function () {
        if (xhr.readyState === 4) {
            let status = xhr.status
            if (status >= 200 && status < 300) {
                options.success && options.success(xhr.responseText,
xhr.responseXML)
            } else {
                options.fail && options.fail(status)
            }
        }
    }
}
```

使用方式如下

```
ajax({
    type: 'post',
    dataType: 'json',
    data: {},
    url: 'https://xxxx',
    success: function(text,xml){//请求成功后的回调函数
        console.log(text)
    },
    fail: function(status){////请求失败后的回调函数
        console.log(status)
    }
})
```

04.你了解axios的原理吗？有看过它的源码吗？



一、axios的使用

关于`axios`的基本使用，上篇文章已经有所涉及，这里再稍微回顾下：

发送请求

```
import axios from 'axios';

axios(config) // 直接传入配置
axios(url[, config]) // 传入url和配置
axios[method](url[, option]) // 直接调用请求方式方法，传入url和配置
axios[method](url[, data[, option]]) // 直接调用请求方式方法，传入data、url和配置
axios.request(option) // 调用 request 方法

const axiosInstance = axios.create(config)
// axiosInstance 也具有以上 axios 的能力

axios.all([axiosInstance1, axiosInstance2]).then(axios.spread(response1,
response2))
// 调用 all 和传入 spread 回调
```

请求拦截器

```
axios.interceptors.request.use(function (config) {
    // 这里写发送请求前处理的代码
    return config;
}, function (error) {
    // 这里写发送请求错误相关的代码
    return Promise.reject(error);
});
```

响应拦截器

```
axios.interceptors.response.use(function (response) {
    // 这里写得到响应数据后处理的代码
    return response;
}, function (error) {
    // 这里写得到错误响应处理的代码
    return Promise.reject(error);
});
```

取消请求

```
// 方式一
const CancelToken = axios.CancelToken;
const source = CancelToken.source();

axios.get('xxxx', {
  cancelToken: source.token
})
// 取消请求 (请求原因是可选的)
source.cancel('主动取消请求');

// 方式二
const CancelToken = axios.CancelToken;
let cancel;

axios.get('xxxx', {
  cancelToken: new CancelToken(function executor(c) {
    cancel = c;
  })
});
cancel('主动取消请求');
```

二、实现一个简易版axios

构建一个`Axios`构造函数，核心代码为`request`

```
class Axios {
```

```
    constructor() {

    }

    request(config) {
        return new Promise(resolve => {
            const {url = '', method = 'get', data = {}} = config;
            // 发送ajax请求
            const xhr = new XMLHttpRequest();
            xhr.open(method, url, true);
            xhr.onload = function() {
                console.log(xhr.responseText)
                resolve(xhr.responseText);
            }
            xhr.send(data);
        })
    }
}
```

导出`axios`实例

```
// 最终导出axios的方法，即实例的request方法
function CreateAxiosFn() {
    let axios = new Axios();
    let req = axios.request.bind(axios);
    return req;
}

// 得到最后的全局变量axios
let axios = CreateAxiosFn();
```

上述就已经能够实现`axios({ })`这种方式的请求

下面是来实现下`axios.method()`这种形式的请求

```
// 定义get,post...方法，挂在到Axios原型上
const methodsArr = ['get', 'delete', 'head', 'options', 'put', 'patch', 'post'];
methodsArr.forEach(met => {
    Axios.prototype[met] = function() {
        console.log('执行'+met+'方法');
        // 处理单个方法
        if (['get', 'delete', 'head', 'options'].includes(met)) { // 2个参数
(url[, config])
            return this.request({
                method: met,
                url: arguments[0],
                ...arguments[1] || {}
            })
        } else { // 3个参数(url[,data[,config]])
            return this.request({
                method: met,
                url: arguments[0],
                data: arguments[1] || {},
                ...arguments[2] || {}
            })
        }
```

```
    }
})
```

将`Axios.prototype`上的方法搬运到`request`上

首先实现个工具类，实现将`b`方法混入到`a`，并且修改`this`指向

```
const utils = {
  extend(a,b, context) {
    for(let key in b) {
      if (b.hasOwnProperty(key)) {
        if (typeof b[key] === 'function') {
          a[key] = b[key].bind(context);
        } else {
          a[key] = b[key]
        }
      }

    }
  }
}
```

修改导出的方法

```
function CreateAxiosFn() {
  let axios = new Axios();

  let req = axios.request.bind(axios);
  // 增加代码
  utils.extend(req, Axios.prototype, axios)

  return req;
}
```

构建拦截器的构造函数

```
class InterceptorsManage {
  constructor() {
    this.handlers = [];
  }

  use(fullfield, rejected) {
    this.handlers.push({
      fullfield,
      rejected
    })
  }
}
```

实现`axios.interceptors.response.use`和`axios.interceptors.request.use`

```
class Axios {
    constructor() {
        // 新增代码
        this.interceptors = {
            request: new InterceptorsManage,
            response: new InterceptorsManage
        }
    }

    request(config) {
        ...
    }
}
```

执行语句`axios.interceptors.response.use`和`axios.interceptors.request.use`的时候，实现获取`axios`实例上的`interceptors`对象，然后再获取`response`或`request`拦截器，再执行对应的拦截器的`use`方法

把`Axios`上的方法和属性搬到`request`过去

```
function CreateAxiosFn() {
  let axios = new Axios();

  let req = axios.request.bind(axios);
  // 混入方法， 处理axios的request方法，使之拥有get,post...方法
  utils.extend(req, Axios.prototype, axios)
  // 新增代码
  utils.extend(req, axios)
  return req;
}
```

现在`request`也有了`interceptors`对象，在发送请求的时候，会先获取`request`拦截器的`handlers`的方法来执行

首先将执行`ajax`的请求封装成一个方法

```
request(config) {
    this.sendAjax(config)
}
sendAjax(config){
    return new Promise(resolve => {
        const {url = '', method = 'get', data = {}} = config;
        // 发送ajax请求
        console.log(config);
        const xhr = new XMLHttpRequest();
        xhr.open(method, url, true);
        xhr.onload = function() {
            console.log(xhr.responseText)
            resolve(xhr.responseText);
        };
        xhr.send(data);
    })
}
```

获得`handlers`中的回调

```
request(config) {
    // 拦截器和请求组装队列
    let chain = [this.sendAjax.bind(this), undefined] // 成对出现的，失败回调暂时不处理

    // 请求拦截
    this.interceptors.request.handlers.forEach(interceptor => {
        chain.unshift(interceptor.fullfield, interceptor.rejected)
    })

    // 响应拦截
    this.interceptors.response.handlers.forEach(interceptor => {
        chain.push(interceptor.fullfield, interceptor.rejected)
    })

    // 执行队列，每次执行一对，并给promise赋最新的值
    let promise = Promise.resolve(config);
    while(chain.length > 0) {
        promise = promise.then(chain.shift(), chain.shift())
    }
    return promise;
}
```

`chains`大概是`['fulfilled1','reject1','fulfilled2','reject2','this.sendAjax','undefined','fulfilled2','reject2','fulfilled1','reject1']`这种形式

这样就能够成功实现一个简易版`axios`

三、源码分析

首先看看目录结构

```
├── /lib/                       # 项目源码目
│ ├── /adapters/                # 定义发送请求的适配器
│ │ ├── http.js                 # node环境http对象
│ │ └── xhr.js                  # 浏览器环境XML对象
│ ├── /cancel/                  # 定义取消功能
│ ├── /helpers/                 # 一些辅助方法
│ ├── /core/                    # 一些核心功能
│ │ ├── Axios.js                # axios实例构造函数
│ │ ├── createError.js          # 抛出错误
│ │ ├── dispatchRequest.js      # 用来调用http请求适配器方法发送请求
│ │ ├── InterceptorManager.js   # 拦截器管理器
│ │ ├── mergeConfig.js          # 合并参数
│ │ ├── settle.js               # 根据http响应状态，改变Promise的状态
│ │ └── transformData.js        # 改变数据格式
│ ├── axios.js                  # 入口，创建构造函数
│ ├── defaults.js               # 默认配置
│ └── utils.js                  # 公用工具
```

`axios`发送请求有很多实现的方法，实现入口文件为`axios.js`

```
function createInstance(defaultConfig) {
  var context = new Axios(defaultConfig);

  // instance指向了request方法，且上下文指向context，所以可以直接以 instance(option) 方式调用
  // Axios.prototype.request 内对第一个参数的数据类型判断，使我们能够以 instance(url, option) 方式调用
  var instance = bind(Axios.prototype.request, context);

  // 把Axios.prototype上的方法扩展到instance对象上，
  // 并指定上下文为context，这样执行Axios原型链上的方法时，this会指向context
  utils.extend(instance, Axios.prototype, context);

  // Copy context to instance
  // 把context对象上的自身属性和方法扩展到instance上
  // 注：因为extend内部使用的forEach方法对对象做for in 遍历时，只遍历对象本身的属性，而不会遍历原型链上的属性
  // 这样，instance 就有了  defaults、interceptors 属性。
  utils.extend(instance, context);
  return instance;
}

// Create the default instance to be exported 创建一个由默认配置生成的axios实例
var axios = createInstance(defaults);

// Factory for creating new instances 扩展axios.create工厂函数，内部也是createInstance
axios.create = function create(instanceConfig) {
  return createInstance(mergeConfig(axios.defaults, instanceConfig));
};

// Expose all/spread
axios.all = function all(promises) {
  return Promise.all(promises);
};

axios.spread = function spread(callback) {
  return function wrap(arr) {
    return callback.apply(null, arr);
  };
};
module.exports = axios;
```

主要核心是`Axios.prototype.request`，各种请求方式的调用实现都是在`request`内部实现的， 简单看下`request`的逻辑

```
Axios.prototype.request = function request(config) {
  // Allow for axios('example/url'[, config]) a la fetch API
  // 判断 config 参数是否是 字符串，如果是则认为第一个参数是 URL，第二个参数是真正的config
  if (typeof config === 'string') {
    config = arguments[1] || {};
    // 把 url 放置到 config 对象中，便于之后的 mergeConfig
    config.url = arguments[0];
  } else {
```

```
    // 如果 config 参数是否是 字符串，则整体都当做config
    config = config || {};
  }
  // 合并默认配置和传入的配置
  config = mergeConfig(this.defaults, config);
  // 设置请求方法
  config.method = config.method ? config.method.toLowerCase() : 'get';
  /*
    something... 此部分会在后续拦截器单独讲述
  */
};

// 在 Axios 原型上挂载 'delete', 'get', 'head', 'options' 且不传参的请求方法，实现内部也是 request
utils.forEach(['delete', 'get', 'head', 'options'], function forEachMethodNoData(method) {
  Axios.prototype[method] = function(url, config) {
    return this.request(utils.merge(config || {}, {
      method: method,
      url: url
    }));
  };
});

// 在 Axios 原型上挂载 'post', 'put', 'patch' 且传参的请求方法，实现内部同样也是 request
utils.forEach(['post', 'put', 'patch'], function forEachMethodWithData(method) {
  Axios.prototype[method] = function(url, data, config) {
    return this.request(utils.merge(config || {}, {
      method: method,
      url: url,
      data: data
    }));
  };
});
```

`request`入口参数为`config`，可以说`config`贯彻了`axios`的一生

`axios` 中的 `config`主要分布在这几个地方：

- 默认配置 `defaults.js`
- `config.method`默认为 `get`
- 调用 `createInstance` 方法创建 `axios`实例，传入的`config`
- 直接或间接调用 `request` 方法，传入的 `config`

```
// axios.js
// 创建一个由默认配置生成的axios实例
var axios = createInstance(defaults);

// 扩展axios.create工厂函数，内部也是 createInstance
axios.create = function create(instanceConfig) {
  return createInstance(mergeConfig(axios.defaults, instanceConfig));
};

// Axios.js
// 合并默认配置和传入的配置
config = mergeConfig(this.defaults, config);
// 设置请求方法
```

```
config.method = config.method ? config.method.toLowerCase() : 'get';
```

从源码中，可以看到优先级：默认配置对象`default` < `method:get` < `Axios`的实例属性`this.default` < `request`参数

下面重点看看`request`方法

```
Axios.prototype.request = function request(config) {
  /*
    先是 mergeConfig ... 等，不再阐述
  */
  // Hook up interceptors middleware 创建拦截器链. dispatchRequest 是重中之重，后续重点
  var chain = [dispatchRequest, undefined];

  // push各个拦截器方法 注意：interceptor.fulfilled 或 interceptor.rejected 是可能为undefined
  this.interceptors.request.forEach(function unshiftRequestInterceptors(interceptor) {
    // 请求拦截器逆序 注意此处的 forEach 是自定义的拦截器的forEach方法
    chain.unshift(interceptor.fulfilled, interceptor.rejected);
  });

  this.interceptors.response.forEach(function pushResponseInterceptors(interceptor) {
    // 响应拦截器顺序 注意此处的 forEach 是自定义的拦截器的forEach方法
    chain.push(interceptor.fulfilled, interceptor.rejected);
  });

  // 初始化一个promise对象，状态为resolved，接收到的参数为已经处理合并过的config对象
  var promise = Promise.resolve(config);

  // 循环拦截器的链
  while (chain.length) {
    promise = promise.then(chain.shift(), chain.shift()); // 每一次向外弹出拦截器
  }
  // 返回 promise
  return promise;
};
```

拦截器`interceptors`是在构建`axios`实例化的属性

```
function Axios(instanceConfig) {
  this.defaults = instanceConfig;
  this.interceptors = {
    request: new InterceptorManager(), // 请求拦截
    response: new InterceptorManager() // 响应拦截
  };
}
```

`InterceptorManager`构造函数

```
// 拦截器的初始化 其实就是一组钩子函数
function InterceptorManager() {
  this.handlers = [];
```

```
}

// 调用拦截器实例的use时就是往钩子函数中push方法
InterceptorManager.prototype.use = function use(fulfilled, rejected) {
  this.handlers.push({
    fulfilled: fulfilled,
    rejected: rejected
  });
  return this.handlers.length - 1;
};

// 拦截器是可以取消的，根据use的时候返回的ID，把某一个拦截器方法置为null
// 不能用 splice 或者 slice 的原因是 删除之后 id 就会变化，导致之后的顺序或者是操作不可控
InterceptorManager.prototype.eject = function eject(id) {
  if (this.handlers[id]) {
    this.handlers[id] = null;
  }
};

// 这就是在 Axios的request方法中 中循环拦截器的方法 forEach 循环执行钩子函数
InterceptorManager.prototype.forEach = function forEach(fn) {
  utils.forEach(this.handlers, function forEachHandler(h) {
    if (h !== null) {
      fn(h);
    }
  });
}
```

请求拦截器方法是被 `unshift` 到拦截器中，响应拦截器是被 `push` 到拦截器中的。最终它们会拼接上一个叫 `dispatchRequest` 的方法被后续的 `promise` 顺序执行

```
var utils = require('./../utils');
var transformData = require('./transformData');
var isCancel = require('../cancel/isCancel');
var defaults = require('../defaults');
var isAbsoluteURL = require('./../helpers/isAbsoluteURL');
var combineURLs = require('./../helpers/combineURLs');

// 判断请求是否已被取消，如果已经被取消，抛出已取消
function throwIfCancellationRequested(config) {
  if (config.cancelToken) {
    config.cancelToken.throwIfRequested();
  }
}

module.exports = function dispatchRequest(config) {
  throwIfCancellationRequested(config);

  // 如果包含baseUrl，并且不是config.url绝对路径，组合baseUrl以及config.url
  if (config.baseURL && !isAbsoluteURL(config.url)) {
    // 组合baseURL与url形成完整的请求路径
    config.url = combineURLs(config.baseURL, config.url);
  }

  config.headers = config.headers || {};
```

```
  // 使用/lib/defaults.js中的transformRequest方法，对config.headers和config.data进行格式化
  // 比如将headers中的Accept，Content-Type统一处理成大写
  // 比如如果请求正文是一个Object会格式化为JSON字符串，并添加application/json;charset=utf-8的Content-Type
  // 等一系列操作
  config.data = transformData(
    config.data,
    config.headers,
    config.transformRequest
  );

  // 合并不同配置的headers，config.headers的配置优先级更高
  config.headers = utils.merge(
    config.headers.common || {},
    config.headers[config.method] || {},
    config.headers || {}
  );

  // 删除headers中的method属性
  utils.forEach(
    ['delete', 'get', 'head', 'post', 'put', 'patch', 'common'],
    function cleanHeaderConfig(method) {
      delete config.headers[method];
    }
  );

  // 如果config配置了adapter，使用config中配置adapter的替代默认的请求方法
  var adapter = config.adapter || defaults.adapter;

  // 使用adapter方法发起请求（adapter根据浏览器环境或者Node环境会有不同）
  return adapter(config).then(
    // 请求正确返回的回调
    function onAdapterResolution(response) {
      // 判断是否以及取消了请求，如果取消了请求抛出以取消
      throwIfCancellationRequested(config);

      // 使用/lib/defaults.js中的transformResponse方法，对服务器返回的数据进行格式化
      // 例如，使用JSON.parse对响应正文进行解析
      response.data = transformData(
        response.data,
        response.headers,
        config.transformResponse
      );

      return response;
    },
    // 请求失败的回调
    function onAdapterRejection(reason) {
      if (!isCancel(reason)) {
        throwIfCancellationRequested(config);

        if (reason && reason.response) {
          reason.response.data = transformData(
            reason.response.data,
            reason.response.headers,
            config.transformResponse
          );
```

```
        }
      }
      return Promise.reject(reason);
    }
  );
};
```

再来看看`axios`是如何实现取消请求的，实现文件在`CancelToken.js`

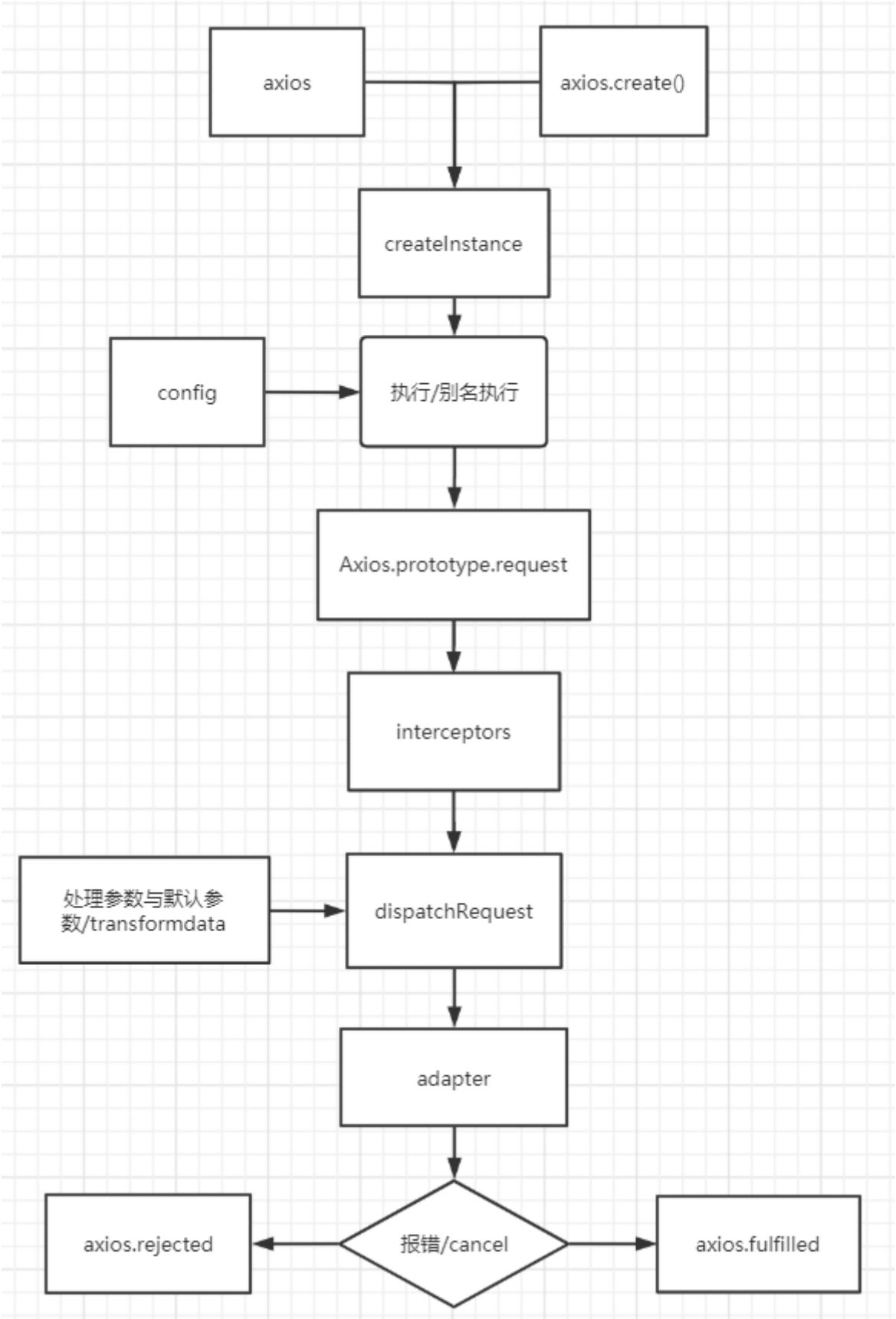
```
function CancelToken(executor) {
  if (typeof executor !== 'function') {
    throw new TypeError('executor must be a function.');
  }
  // 在 CancelToken 上定义一个 pending 状态的 promise ，将 resolve 回调赋值给外部变量 resolvePromise
  var resolvePromise;
  this.promise = new Promise(function promiseExecutor(resolve) {
    resolvePromise = resolve;
  });

  var token = this;
  // 立即执行 传入的 executor函数，将真实的 cancel 方法通过参数传递出去。
  // 一旦调用就执行 resolvePromise 即前面的 promise 的 resolve，就更改promise的状态为resolve。
  // 那么xhr中定义的 CancelToken.promise.then方法就会执行，从而xhr内部会取消请求
  executor(function cancel(message) {
    // 判断请求是否已经取消过，避免多次执行
    if (token.reason) {
      return;
    }
    token.reason = new Cancel(message);
    resolvePromise(token.reason);
  });
}

CancelToken.source = function source() {
  // source 方法就是返回了一个 CancelToken 实例，与直接使用 new CancelToken 是一样的操作
  var cancel;
  var token = new CancelToken(function executor(c) {
    cancel = c;
  });
  // 返回创建的 CancelToken 实例以及取消方法
  return {
    token: token,
    cancel: cancel
  };
};
```

实际上取消请求的操作是在`xhr.js`中也有响应的配合的

```
if (config.cancelToken) {
    config.cancelToken.promise.then(function onCanceled(cancel) {
        if (!request) {
            return;
        }
        // 取消请求
        request.abort();
        reject(cancel);
    });
}
```

巧妙的地方在 `CancelToken` 中 `executor` 函数，通过 `resolve` 函数的传递与执行，控制 `promise` 的状态

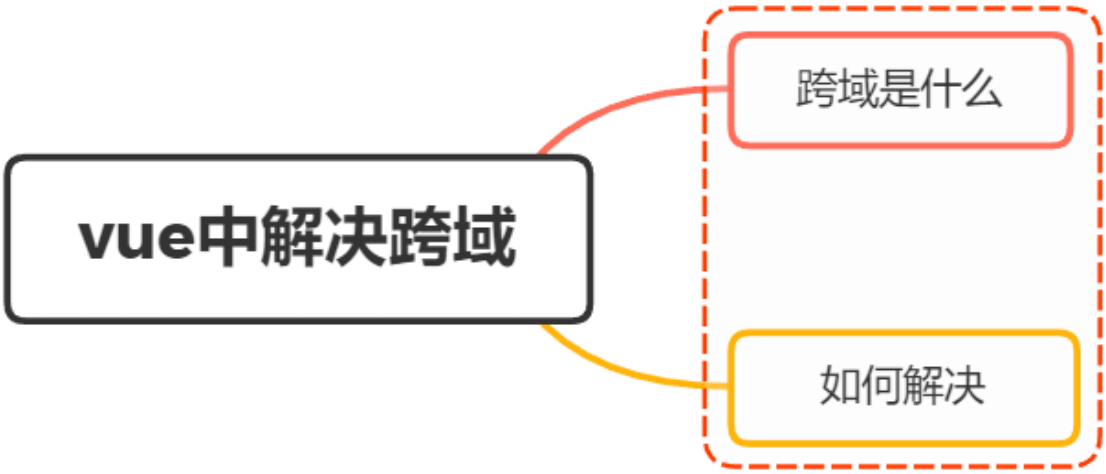
小结



参考文献

- https://juejin.cn/post/6856706569263677447#heading-4
- https://juejin.cn/post/6844903907500490766
- https://github.com/axios/axios

05.Vue项目中你是如何解决跨域的呢?



一、跨域是什么

跨域本质是浏览器基于**同源策略**的一种安全手段

同源策略（Sameoriginpolicy），是一种约定，它是浏览器最核心也最基本的安全功能

所谓同源（即指在同一个域）具有以下三个相同点

- 协议相同（protocol）
- 主机相同（host）
- 端口相同（port）

反之非同源请求，也就是协议、端口、主机其中一项不相同的时候，这时候就会产生跨域

> 一定要注意跨域是浏览器的限制，你用抓包工具抓取接口数据，是可以看到接口已经把数据返回回来了，只是浏览器的限制，你获取不到数据。用postman请求接口能够请求到数据。这些再次印证了跨域是浏览器的限制。

二、如何解决

解决跨域的方法有很多，下面列举了三种：

- JSONP
- CORS
- Proxy

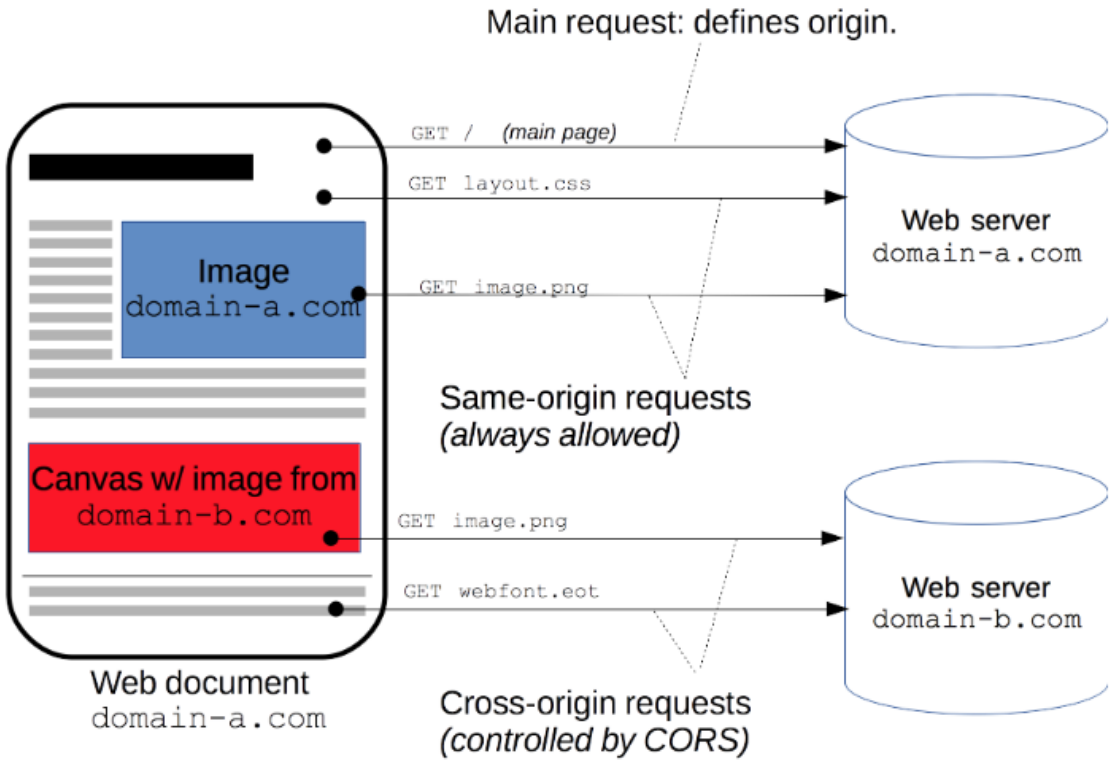
而在 `vue` 项目中，我们主要针对 `CORS` 或 `Proxy` 这两种方案进行展开

CORS

CORS （Cross-Origin Resource Sharing，跨域资源共享）是一个系统，它由一系列传输的HTTP头组成，这些HTTP头决定浏览器是否阻止前端 JavaScript 代码获取跨域请求的响应

`CORS` 实现起来非常方便，只需要增加一些 `HTTP` 头，让服务器能声明允许的访问来源

只要后端实现了 `CORS`，就实现了跨域



以`koa`框架举例

添加中间件，直接设置`Access-Control-Allow-Origin`响应头

```
app.use(async (ctx, next)=> {
  ctx.set('Access-Control-Allow-Origin', '*');
  ctx.set('Access-Control-Allow-Headers', 'Content-Type, Content-Length,
Authorization, Accept, X-Requested-With , yourHeaderFeild');
  ctx.set('Access-Control-Allow-Methods', 'PUT, POST, GET, DELETE, OPTIONS');
  if (ctx.method == 'OPTIONS') {
    ctx.body = 200;
  } else {
    await next();
  }
})
```

ps: `Access-Control-Allow-Origin` 设置为*其实意义不大，可以说是形同虚设，实际应用中，上线前我们会将`Access-Control-Allow-Origin` 值设为我们目标`host`

Proxy

代理（Proxy）也称网络代理，是一种特殊的网络服务，允许一个（一般为客户端）通过这个服务与另一个网络终端（一般为服务器）进行非直接的连接。一些网关、路由器等网络设备具备网络代理功能。一般认为代理服务有利于保障网络终端的隐私或安全，防止攻击

方案一

如果是通过`vue-cli`脚手架工具搭建项目，我们可以通过`webpack`为我们起一个本地服务器作为请求的代理对象

通过该服务器转发请求至目标服务器，得到结果再转发给前端，但是最终发布上线时如果web应用和接口服务器不在一起仍会跨域

在`vue.config.js`文件，新增以下代码

```
amodule.exports = {
    devServer: {
        host: '127.0.0.1',
        port: 8084,
        open: true,// vue项目启动时自动打开浏览器
        proxy: {
            '/api': { // '/api'是代理标识，用于告诉node，url前面是/api的就是使用代理的
                target: "http://xxx.xxx.xx.xx:8080", //目标地址，一般是指后台服务器地
址
                changeOrigin: true, //是否跨域
                pathRewrite: { // pathRewrite 的作用是把实际Request Url中
的'/api'用""代替
                    '^/api': ""
                }
            }
        }
    }
}
```

通过`axios`发送请求中，配置请求的根路径

```
axios.defaults.baseURL = '/api'
```

方案二

此外，还可通过服务端实现代理请求转发

以`express`框架为例

```
var express = require('express');
const proxy = require('http-proxy-middleware')
const app = express()
app.use(express.static(__dirname + '/'))
app.use('/api', proxy({ target: 'http://localhost:4000', changeOrigin: false
                      }));
module.exports = app
```

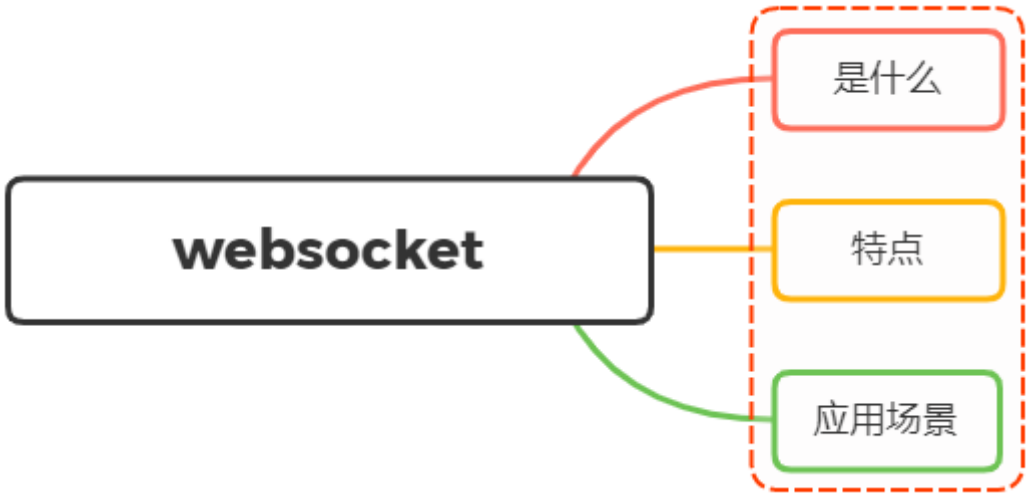
方案三

通过配置`nginx`实现代理

```
server {
    listen       80;
    # server_name www.josephxia.com;
    location / {
        root   /var/www/html;
        index  index.html index.htm;
        try_files $uri $uri/ /index.html;
    }
    location /api {
        proxy_pass  http://127.0.0.1:3000;
        proxy_redirect   off;
        proxy_set_header  Host       $host;
```

```
        proxy_set_header  X-Real-IP      $remote_addr;
        proxy_set_header  X-Forwarded-For  $proxy_add_x_forwarded_for;
    }
}
```

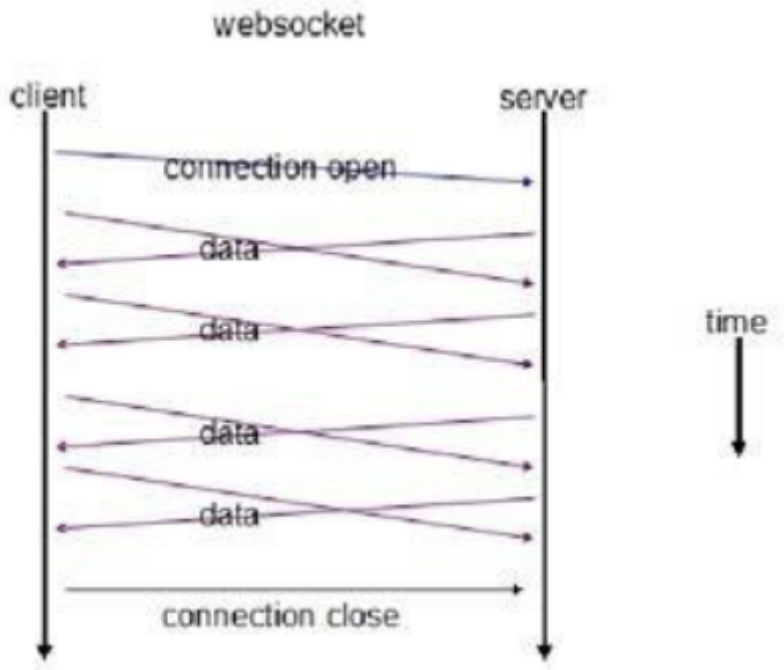
06.说说对WebSocket的理解？应用场景？



一、是什么

WebSocket，是一种网络传输协议，位于`OSI`模型的应用层。可在单个`TCP`连接上进行全双工通信，能更好的节省服务器资源和带宽并达到实时通迅

客户端和服务器只需要完成一次握手，两者之间就可以创建持久性的连接，并进行双向数据传输



从上图可见，`websocket`服务器与客户端通过握手连接，连接成功后，两者都能主动的向对方发送或接受数据

而在`websocket`出现之前，开发实时`web`应用的方式为轮询

不停地向服务器发送 HTTP 请求，问有没有数据，有数据的话服务器就用响应报文回应。如果轮询的频率比较高，那么就可以近似地实现“实时通信”的效果

轮询的缺点也很明显，反复发送无效查询请求耗费了大量的带宽和 `CPU` 资源

二、特点

全双工

通信允许数据在两个方向上同时传输，它在能力上相当于两个单工通信方式的结合

例如指 A→B 的同时 B→A ，是瞬时同步的

二进制帧

采用了二进制帧结构，语法、语义与 HTTP 完全不兼容，相比 `http/2` ，`WebSocket` 更侧重于“实时通信”，而 `HTTP/2` 更侧重于提高传输效率，所以两者的帧结构也有很大的区别

不像 `HTTP/2` 那样定义流，也就不存在多路复用、优先级等特性

自身就是全双工，也不需要服务器推送

协议名

引入 `ws` 和 `wss` 分别代表明文和密文的 `websocket` 协议，且默认端口使用80或443，几乎与 `http` 一致

```
ws://www.chrono.com
ws://www.chrono.com:8080/srv
wss://www.chrono.com:445/im?user_id=xxx
```

握手

`WebSocket` 也要有一个握手过程，然后才能正式收发数据

客户端发送数据格式如下：

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

- Connection：必须设置Upgrade，表示客户端希望连接升级
- Upgrade：必须设置Websocket，表示希望升级到Websocket协议
- Sec-WebSocket-Key：客户端发送的一个 base64 编码的密文，用于简单的认证秘钥。要求服务端必须返回一个对应加密的“Sec-WebSocket-Accept应答，否则客户端会抛出错误，并关闭连接
- Sec-WebSocket-Version ： 表示支持的Websocket版本

服务端返回的数据格式：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=Sec-WebSocket-Protocol: chat
```

- HTTP/1.1 101 Switching Protocols：表示服务端接受 WebSocket 协议的客户端连接
- Sec-WebSocket-Accep：验证客户端请求报文，同样也是为了防止误连接。具体做法是把请求头里“Sec-WebSocket-Key”的值，加上一个专用的 UUID，再计算摘要

优点

- 较少的控制开销：数据包头部协议较小，不同于http每次请求需要携带完整的头部
- 更强的实时性：相对于HTTP请求需要等待客户端发起请求服务端才能响应，延迟明显更少
- 保持创连接状态：创建通信后，可省略状态信息，不同于HTTP每次请求需要携带身份验证
- 更好的二进制支持：定义了二进制帧，更好处理二进制内容
- 支持扩展：用户可以扩展websocket协议、实现部分自定义的子协议
- 更好的压缩效果：Websocket在适当的扩展支持下，可以沿用之前内容的上下文，在传递类似的数据时，可以显著地提高压缩率

二、应用场景

基于`websocket`的事实通信的特点，其存在的应用场景大概有：

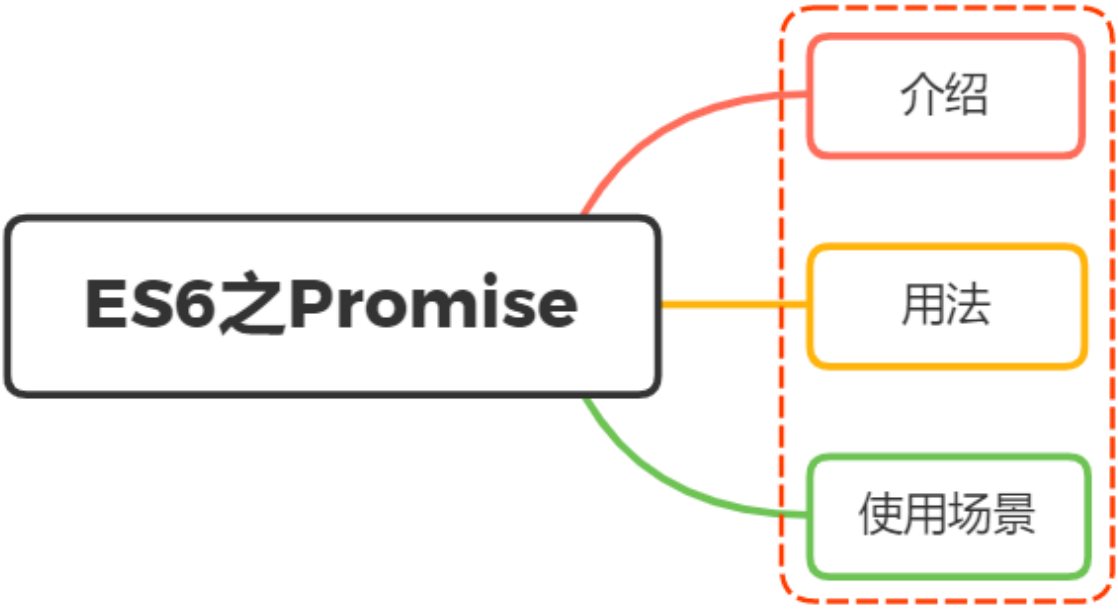
- 弹幕
- 媒体聊天
- 协同编辑
- 基于位置的应用
- 体育实况更新
- 股票基金报价实时更新

参考文献

- https://zh.wikipedia.org/wiki/WebSocket
- https://www.oschina.net/translate/9-killer-uses-for-websockets
- https://vue3js.cn/interview

7.nodejs

01.你是怎么理解ES6中 Promise的？使用场景？



一、介绍

`Promise`，译为承诺，是异步编程的一种解决方案，比传统的解决方案（回调函数）更加合理和更加强大

在以往我们如果处理多层异步操作，我们往往会像下面那样编写我们的代码

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('得到最终结果: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

阅读上面代码，是不是很难受，上述形成了经典的回调地狱

现在通过`Promise`的改写上面的代码

```
doSomething().then(function(result) {
  return doSomethingElse(result);
})
.then(function(newResult) {
  return doThirdThing(newResult);
})
.then(function(finalResult) {
  console.log('得到最终结果: ' + finalResult);
})
.catch(failureCallback);
```

瞬间感受到`promise`解决异步操作的优点：

- 链式操作减低了编码难度
- 代码可读性明显增强

下面我们正式来认识`promise`：

状态

`promise`对象仅有三种状态

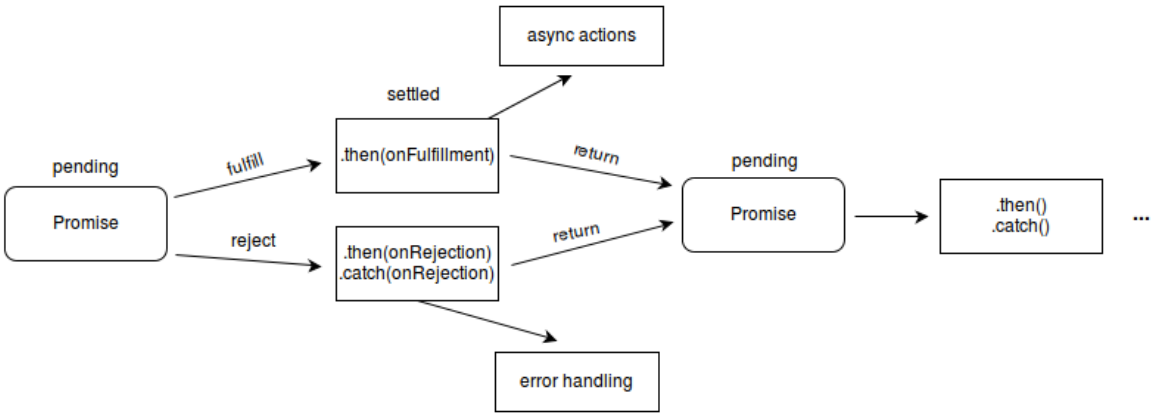
- `pending`（进行中）
- `fulfilled`（已成功）
- `rejected`（已失败）

特点

- 对象的状态不受外界影响，只有异步操作的结果，可以决定当前是哪一种状态
- 一旦状态改变（从`pending`变为`fulfilled`和从`pending`变为`rejected`），就不会再变，任何时候都可以得到这个结果

流程

认真阅读下图，我们能够轻松了解`promise`整个流程



二、用法

`Promise`对象是一个构造函数，用来生成`Promise`实例

```
const promise = new Promise(function(resolve, reject) {});
```

`Promise`构造函数接受一个函数作为参数，该函数的两个参数分别是`resolve`和`reject`

- `resolve`函数的作用是，将`Promise`对象的状态从“未完成”变为“成功”
- `reject`函数的作用是，将`Promise`对象的状态从“未完成”变为“失败”

实例方法

`Promise`构建出来的实例存在以下方法：

- then()
- then()
- catch()

- finally()

then()

`then`是实例状态发生改变时的回调函数，第一个参数是`resolved`状态的回调函数，第二个参数是`rejected`状态的回调函数

`then`方法返回的是一个新的`Promise`实例，也就是`promise`能链式书写的原因

```
getJSON("/posts.json").then(function(json) {
  return json.post;
}).then(function(post) {
  // ...
});
```

catch

`catch()`方法是`.then(null, rejection)`或`.then(undefined, rejection)`的别名，用于指定发生错误时的回调函数

```
getJSON('/posts.json').then(function(posts) {
  // ...
}).catch(function(error) {
  // 处理 getJSON 和 前一个回调函数运行时发生的错误
  console.log('发生错误!', error);
});
```

`Promise`对象的错误具有"冒泡"性质，会一直向后传递，直到被捕获为止

```
getJSON('/post/1.json').then(function(post) {
  return getJSON(post.commentURL);
}).then(function(comments) {
  // some code
}).catch(function(error) {
  // 处理前面三个Promise产生的错误
});
```

一般来说，使用`catch`方法代替`then()`第二个参数

`Promise`对象抛出的错误不会传递到外层代码，即不会有任何反应

```
const someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};
```

浏览器运行到这一行，会打印出错误提示`ReferenceError: x is not defined`，但是不会退出进程

`catch()`方法之中，还能再抛出错误，通过后面`catch`方法捕获到

finally()

`finally()`方法用于指定不管 Promise 对象最后状态如何，都会执行的操作

```
promise
.then(result => {···})
.catch(error => {···})
.finally(() => {···});
```

构造函数方法

`Promise`构造函数存在以下方法：

- all()
- race()
- allSettled()
- resolve()
- reject()
- try()

all()

`Promise.all()`方法用于将多个`Promise`实例，包装成一个新的`Promise`实例

```
const p = Promise.all([p1, p2, p3]);
```

接受一个数组（迭代对象）作为参数，数组成员都应为`Promise`实例

实例`p`的状态由`p1`、`p2`、`p3`决定，分为两种：

- 只有`p1`、`p2`、`p3`的状态都变成`fulfilled`，`p`的状态才会变成`fulfilled`，此时`p1`、`p2`、`p3`的返回值组成一个数组，传递给`p`的回调函数
- 只要`p1`、`p2`、`p3`之中有一个被`rejected`，`p`的状态就变成`rejected`，此时第一个被`reject`的实例的返回值，会传递给`p`的回调函数

注意，如果作为参数的`Promise`实例，自己定义了`catch`方法，那么它一旦被`rejected`，并不会触发`Promise.all()`的`catch`方法

```
const p1 = new Promise((resolve, reject) => {
  resolve('hello');
})
.then(result => result)
.catch(e => e);

const p2 = new Promise((resolve, reject) => {
  throw new Error('报错了');
})
.then(result => result)
.catch(e => e);

Promise.all([p1, p2])
.then(result => console.log(result))
.catch(e => console.log(e));
```

```
// ["hello", Error: 报错了]
```

如果 `p2` 没有自己的 `catch` 方法，就会调用 `Promise.all()` 的 `catch` 方法

```
const p1 = new Promise((resolve, reject) => {
  resolve('hello');
})
.then(result => result);

const p2 = new Promise((resolve, reject) => {
  throw new Error('报错了');
})
.then(result => result);

Promise.all([p1, p2])
.then(result => console.log(result))
.catch(e => console.log(e));
// Error: 报错了
```

race()

`Promise.race()` 方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例

```
const p = Promise.race([p1, p2, p3]);
```

只要 `p1`、`p2`、`p3` 之中有一个实例率先改变状态，`p` 的状态就跟着改变

率先改变的 Promise 实例的返回值则传递给 `p` 的回调函数

```
const p = Promise.race([
  fetch('/resource-that-may-take-a-while'),
  new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error('request timeout')), 5000)
  })
]);

p
.then(console.log)
.catch(console.error);
```

allSettled()

`Promise.allSettled()` 方法接受一组 Promise 实例作为参数，包装成一个新的 Promise 实例

只有等到所有这些参数实例都返回结果，不管是 `fulfilled` 还是 `rejected`，包装实例才会结束

```
const promises = [
  fetch('/api-1'),
  fetch('/api-2'),
  fetch('/api-3'),
];

await Promise.allSettled(promises);
removeLoadingIndicator();
```

resolve()

将现有对象转为 `Promise` 对象

```
Promise.resolve('foo')
// 等价于
new Promise(resolve => resolve('foo'))
```

参数可以分成四种情况，分别如下：

- 参数是一个 Promise 实例，`promise.resolve` 将不做任何修改、原封不动地返回这个实例
- 参数是一个 `thenable` 对象，`promise.resolve` 会将这个对象转为 `Promise` 对象，然后就立即执行 `thenable` 对象的 `then()` 方法
- 参数不是具有 `then()` 方法的对象，或根本就不是对象，`Promise.resolve()` 会返回一个新的 Promise 对象，状态为 `resolved`
- 没有参数时，直接返回一个 `resolved` 状态的 Promise 对象

reject()

`Promise.reject(reason)` 方法也会返回一个新的 Promise 实例，该实例的状态为 `rejected`

```
const p = Promise.reject('出错了');
// 等同于
const p = new Promise((resolve, reject) => reject('出错了'))

p.then(null, function (s) {
  console.log(s)
});
// 出错了
```

`Promise.reject()` 方法的参数，会原封不动地变成后续方法的参数

```
Promise.reject('出错了')
.catch(e => {
  console.log(e === '出错了')
})
// true
```

三、使用场景

将图片的加载写成一个`Promise`，一旦加载完成，`Promise`的状态就发生变化

```
const preloadImage = function (path) {
  return new Promise(function (resolve, reject) {
    const image = new Image();
    image.onload  = resolve;
    image.onerror = reject;
    image.src = path;
  });
};
```

通过链式操作，将多个渲染数据分别给个`then`，让其各司其职。或当下个异步请求依赖上个请求结果的时候，我们也能够通过链式操作友好解决问题

```
// 各司其职
getInfo().then(res=>{
    let { bannerList } = res
    //渲染轮播图
    console.log(bannerList)
    return res
}).then(res=>{

    let { storeList } = res
    //渲染店铺列表
    console.log(storeList)
    return res
}).then(res=>{
    let { categoryList } = res
    console.log(categoryList)
    //渲染分类列表
    return res
})
```

通过`all()`实现多个请求合并在一起，汇总所有请求结果，只需设置一个`loading`即可

```
function initLoad(){
    // loading.show() //加载loading
    Promise.all([getBannerList(),getStoreList(),getCategoryList()]).then(res=>{
        console.log(res)
        loading.hide() //关闭loading
    }).catch(err=>{
        console.log(err)
        loading.hide()//关闭loading
    })
}
//数据初始化
initLoad()
```

通过`race`可以设置图片请求超时

```
//请求某个图片资源
function requestImg(){
    var p = new Promise(function(resolve, reject){
        var img = new Image();
        img.onload = function(){
```

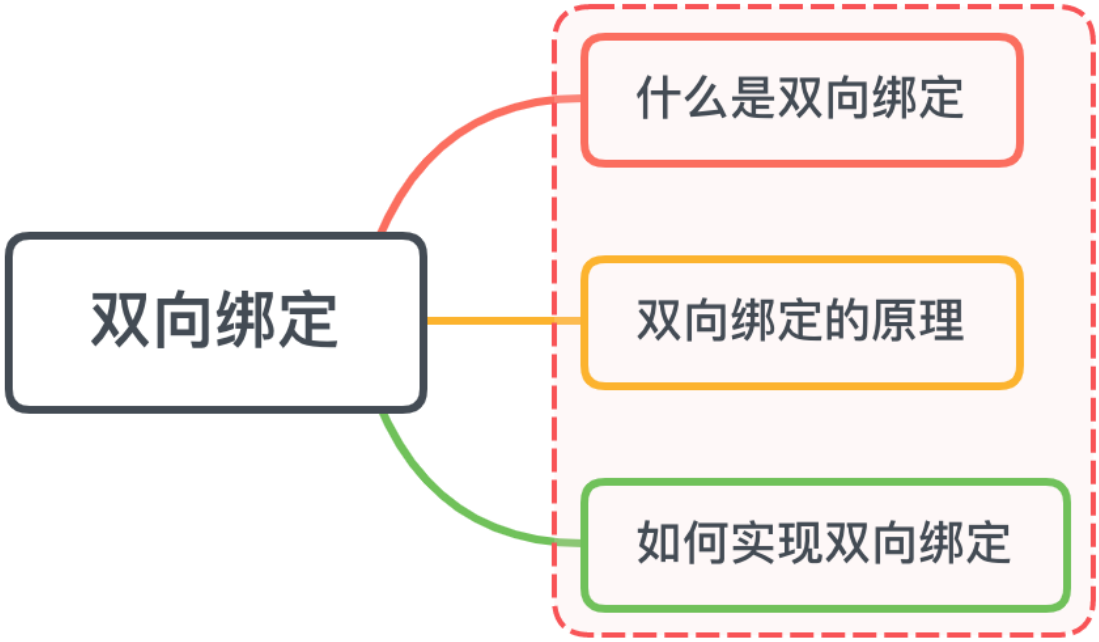
```
            resolve(img);
        }
        //img.src = "https://b-gold-cdn.xitu.io/v3/static/img/logo.a7995ad.svg";
正确的
        img.src = "https://b-gold-cdn.xitu.io/v3/static/img/logo.a7995ad.svg1";
    });
    return p;
}

//延时函数，用于给请求计时
function timeout(){
    var p = new Promise(function(resolve, reject){
        setTimeout(function(){
            reject('图片请求超时');
        }, 5000);
    });
    return p;
}

Promise
.race([requestImg(), timeout()])
.then(function(results){
    console.log(results);
})
.catch(function(reason){
    console.log(reason);
});
```

参考文献

- https://es6.ruanyifeng.com/#docs/promise

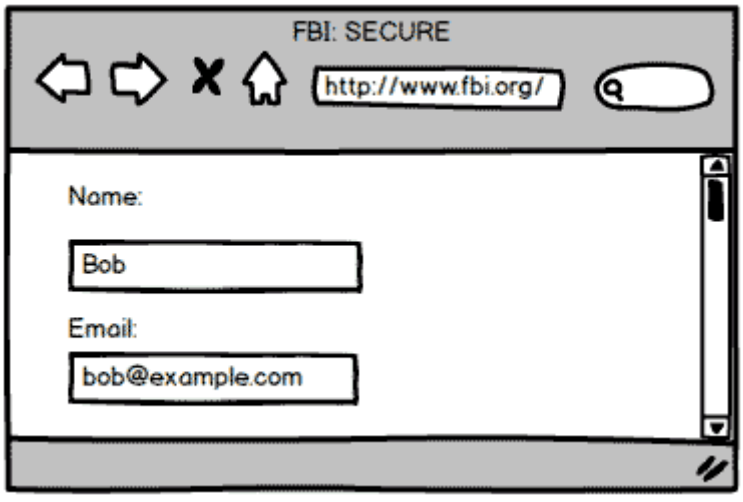
8.vue

01.说说你对双向绑定的理解

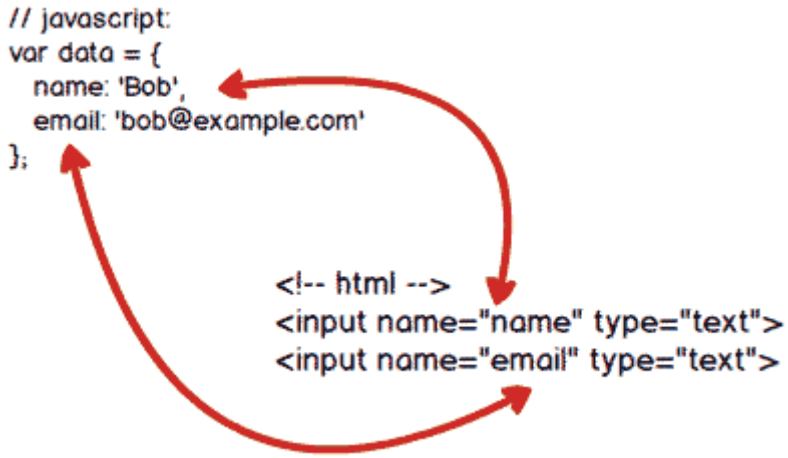


一、什么是双向绑定

我们先从单向绑定切入单向绑定非常简单，就是把`Model`绑定到`View`，当我们用`JavaScript`代码更新`Model`时，`View`就会自动更新双向绑定就很容易联想到了，在单向绑定的基础上，用户更新了`View`，`Model`的数据也自动被更新了，这种情况就是双向绑定举个栗子



当用户填写表单时，`View`的状态就被更新了，如果此时可以自动更新`Model`的状态，那就相当于我们把`Model`和`View`做了双向绑定关系图如下



二、双向绑定的原理是什么

我们都知道 `Vue` 是数据双向绑定的框架，双向绑定由三个重要部分构成

- 数据层（Model）：应用的数据及业务逻辑
- 视图层（View）：应用的展示效果，各类UI组件
- 业务逻辑层（ViewModel）：框架封装的核心，它负责将数据与视图关联起来

而上面的这个分层的架构方案，可以用一个专业术语进行称呼：`MVVM` 这里的控制层的核心功能便是 “数据双向绑定” 。自然，我们只需弄懂它是什么，便可以进一步了解数据绑定的原理

理解ViewModel

它的主要职责就是：

- 数据变化后更新视图
- 视图变化后更新数据

当然，它还有两个主要部分组成

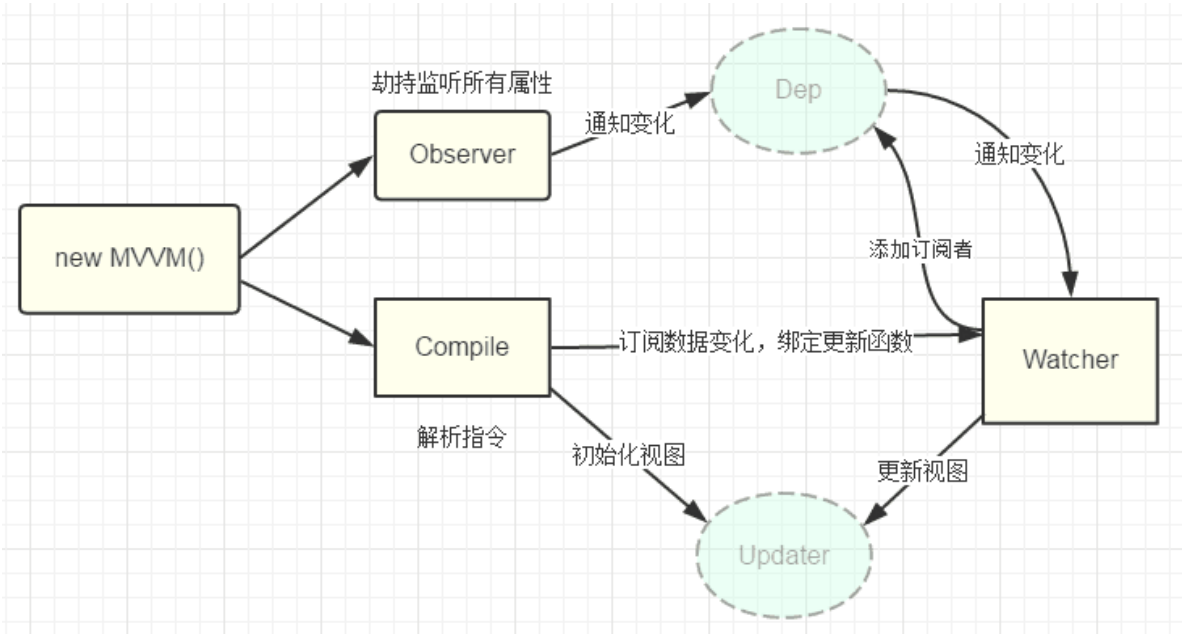
- 监听器（Observer）：对所有数据的属性进行监听
- 解析器（Compiler）：对每个元素节点的指令进行扫描跟解析,根据指令模板替换数据,以及绑定相应的更新函数

三、实现双向绑定

我们还是以 `Vue` 为例，先来看看 `Vue` 中的双向绑定流程是什么的

1. `new Vue()` 首先执行初始化，对 `data` 执行响应化处理，这个过程发生 `Observe` 中
2. 同时对模板执行编译，找到其中动态绑定的数据，从 `data` 中获取并初始化视图，这个过程发生在 `Compile` 中
3. 同时定义一个更新函数和 `Watcher`，将来对应数据变化时 `Watcher` 会调用更新函数
4. 由于 `data` 的某个 `key` 在一个视图中可能出现多次，所以每个 `key` 都需要一个管家 `Dep` 来管理多个 `Watcher`
5. 将来data中数据一旦发生变化，会首先找到对应的 `Dep`，通知所有 `Watcher` 执行更新函数

流程图如下：



实现

先来一个构造函数：执行初始化，对 `data` 执行响应化处理

```
class Vue {
  constructor(options) {
    this.$options = options;
    this.$data = options.data;

    // 对data选项做响应式处理
    observe(this.$data);

    // 代理data到vm上
    proxy(this);

    // 执行编译
    new Compile(options.el, this);
  }
}
```

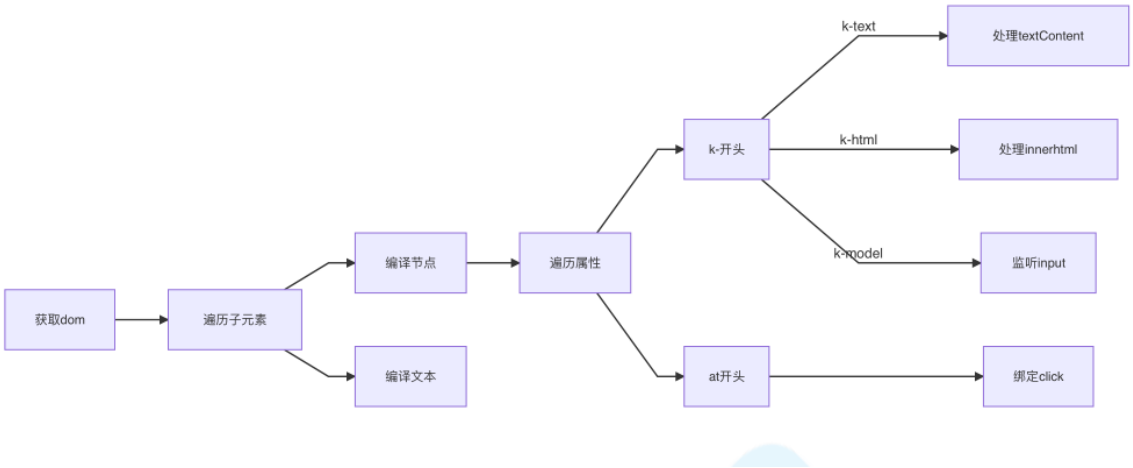
对 `data` 选项执行响应化具体操作

```
function observe(obj) {
  if (typeof obj !== "object" || obj == null) {
    return;
  }
  new Observer(obj);
}

class Observer {
  constructor(value) {
    this.value = value;
    this.walk(value);
  }
  walk(obj) {
    Object.keys(obj).forEach((key) => {
      defineReactive(obj, key, obj[key]);
```

```
    });
  }
}
```

编译 Compile

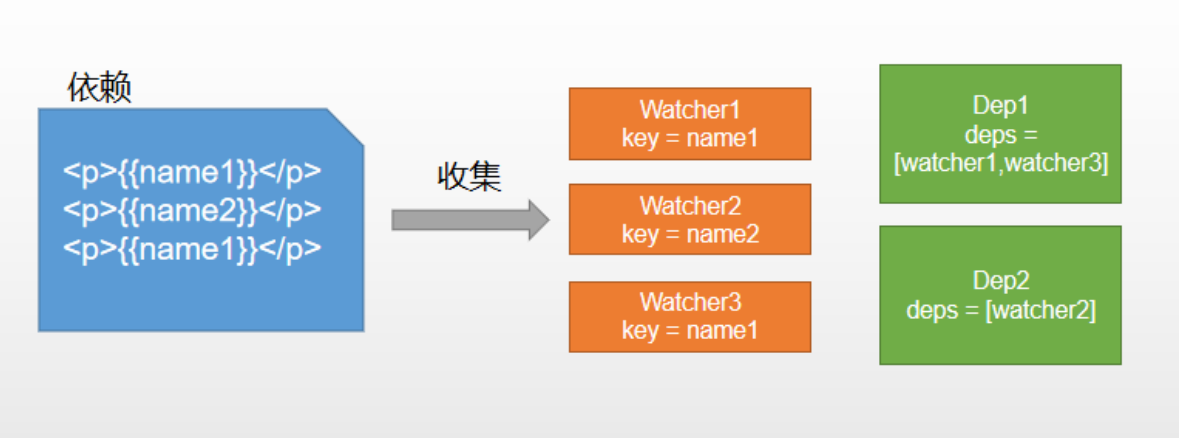
对每个元素节点的指令进行扫描跟解析,根据指令模板替换数据,以及绑定相应的更新函数




```
class Compile {
  constructor(el, vm) {
    this.$vm = vm;
    this.$el = document.querySelector(el);  // 获取dom
    if (this.$el) {
      this.compile(this.$el);
    }
  }
  compile(el) {
    const childNodes = el.childNodes;
    Array.from(childNodes).forEach((node) => { // 遍历子元素
      if (this.isElement(node)) {   // 判断是否为节点
        console.log("编译元素" + node.nodeName);
      } else if (this.isInterpolation(node)) {
        console.log("编译插值文本" + node.textContent);  // 判断是否为插值文本 {{}}
      }
      if (node.childNodes && node.childNodes.length > 0) {  // 判断是否有子元素
        this.compile(node);  // 对子元素进行递归遍历
      }
    });
  }
  isElement(node) {
    return node.nodeType == 1;
  }
  isInterpolation(node) {
    return node.nodeType == 3 && /\{\{(.*)\}\}/.test(node.textContent);
  }
}
```

依赖收集

视图中会用到`data`中某`key`，这称为依赖。同一个`key`可能出现多次，每次都需要收集出来用一个`Watcher`来维护它们，此过程称为依赖收集多个`Watcher`需要一个`Dep`来管理，需要更新时由`Dep`统一通知



实现思路

1. `defineReactive`时为每一个`key`创建一个`Dep`实例
2. 初始化视图时读取某个`key`，例如`name1`，创建一个`watcher1`
3. 由于触发`name1`的`getter`方法，便将`watcher1`添加到`name1`对应的Dep中
4. 当`name1`更新，`setter`触发时，便可通过对应`Dep`通知其管理所有`Watcher`更新

```
// 负责更新视图
class Watcher {
  constructor(vm, key, updater) {
    this.vm = vm
    this.key = key
    this.updaterFn = updater

    // 创建实例时，把当前实例指定到Dep.target静态属性上
    Dep.target = this
    // 读一下key，触发get
    vm[key]
    // 置空
    Dep.target = null
  }

  // 未来执行dom更新函数，由dep调用的
  update() {
    this.updaterFn.call(this.vm, this.vm[this.key])
  }
}
```

声明`Dep`

```
class Dep {
  constructor() {
    this.deps = [];  // 依赖管理
  }
  addDep(dep) {
    this.deps.push(dep);
  }
  notify() {
    this.deps.forEach((dep) => dep.update());
  }
}
```

创建`watcher`时触发`getter`

```
class Watcher {
  constructor(vm, key, updateFn) {
    Dep.target = this;
    this.vm[this.key];
    Dep.target = null;
  }
}
```

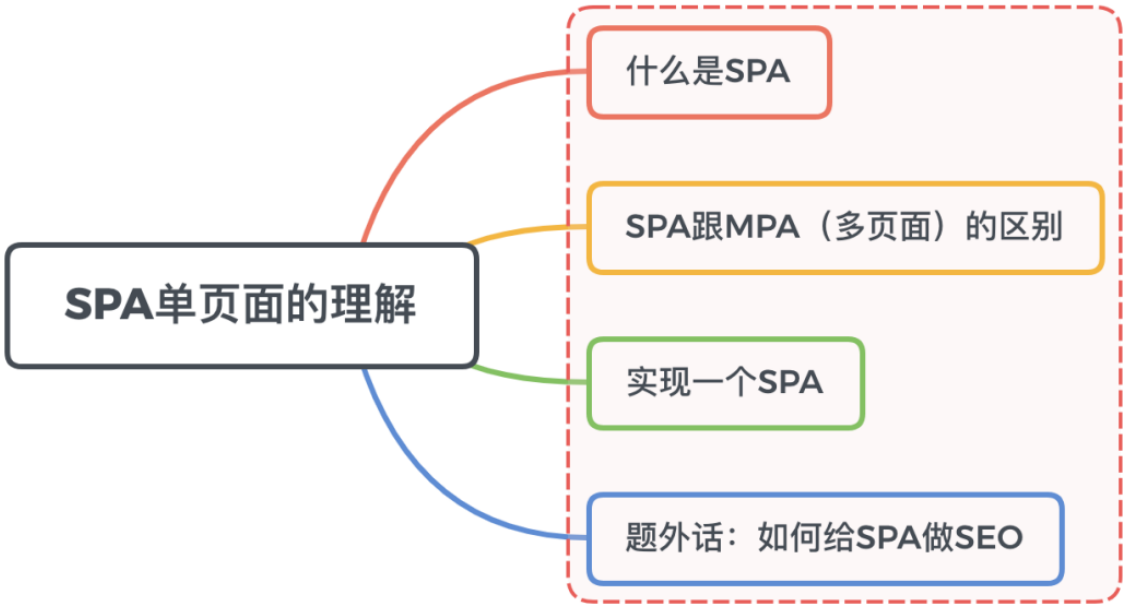
依赖收集，创建`Dep`实例

```
function defineReactive(obj, key, val) {
  this.observe(val);
  const dep = new Dep();
  Object.defineProperty(obj, key, {
    get() {
      Dep.target && dep.addDep(Dep.target);// Dep.target也就是Watcher实例
      return val;
    },
    set(newVal) {
      if (newVal === val) return;
      dep.notify(); // 通知dep执行更新方法
    },
  });
}
```

参考文献

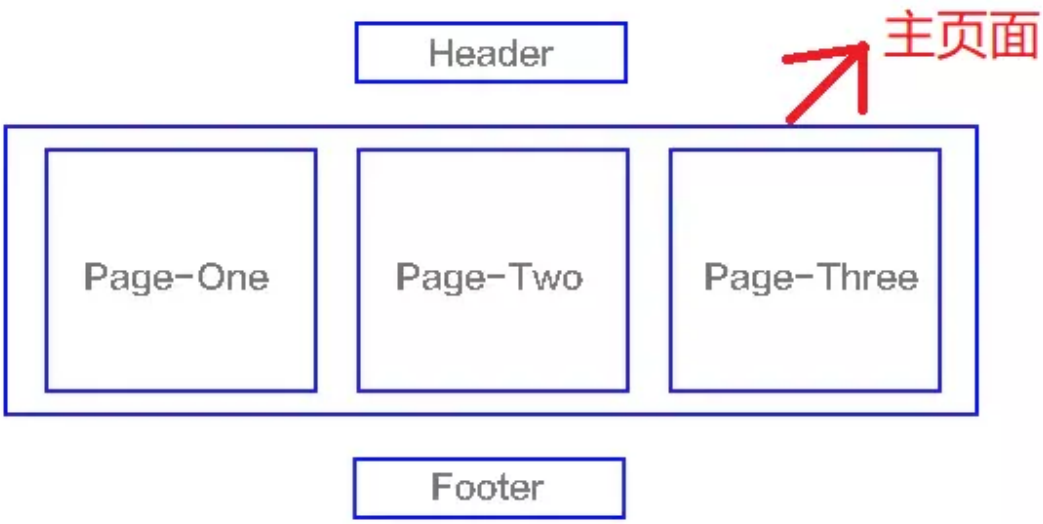
- https://www.liaoxuefeng.com/wiki/1022910821149312/1109527162256416
- https://juejin.cn/post/6844903942254510087#heading-9

02.你对SPA单页面的理解，它的优缺点分别是什么？如何实现SPA应用呢



一、什么是SPA

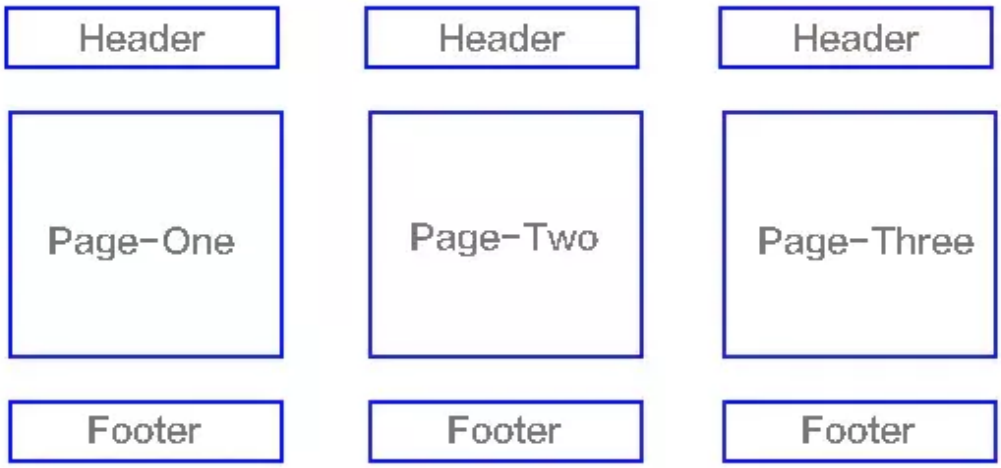
SPA（single-page application），翻译过来就是单页应用`SPA`是一种网络应用程序或网站的模型，它通过动态重写当前页面来与用户交互，这种方法避免了页面之间切换打断用户体验在单页应用中，所有必要的代码（`HTML`、`JavaScript`和`CSS`）都通过单个页面的加载而检索，或者根据需要（通常是为响应用户操作）动态装载适当的资源并添加到页面页面在任何时间点都不会重新加载，也不会将控制转移到其他页面举个例子来讲就是一个杯子，早上装的牛奶，中午装的是开水，晚上装的是茶，我们发现，变的始终是杯子里的内容，而杯子始终是那个杯子结构如下图



我们熟知的JS框架如`react`,`vue`,`angular`,`ember`都属于`SPA`

二、SPA和MPA的区别

上面大家已经对单页面有所了解了，下面来讲讲多页应用MPA（MultiPage-page application），翻译过来就是多页应用在`MPA`中，每个页面都是一个主页面，都是独立的当我们在访问另一个页面的时候，都需要重新加载`html`、`css`、`js`文件，公共文件则根据需求按需加载如下图



单页应用与多页应用的区别

	单页面应用（SPA）	多页面应用（MPA）
组成	一个主页面和多个页面片段	多个主页面
刷新方式	局部刷新	整页刷新
url模式	哈希模式	历史模式
SEO搜索引擎优化	难实现，可使用SSR方式改善	容易实现
数据传递	容易	通过url、cookie、localStorage等传递
页面切换	速度快，用户体验良好	切换加载资源，速度慢，用户体验差
维护成本	相对容易	相对复杂

单页应用优缺点

优点：

- 具有桌面应用的即时性、网站的可移植性和可访问性
- 用户体验好、快，内容的改变不需要重新加载整个页面
- 良好的前后端分离，分工更明确

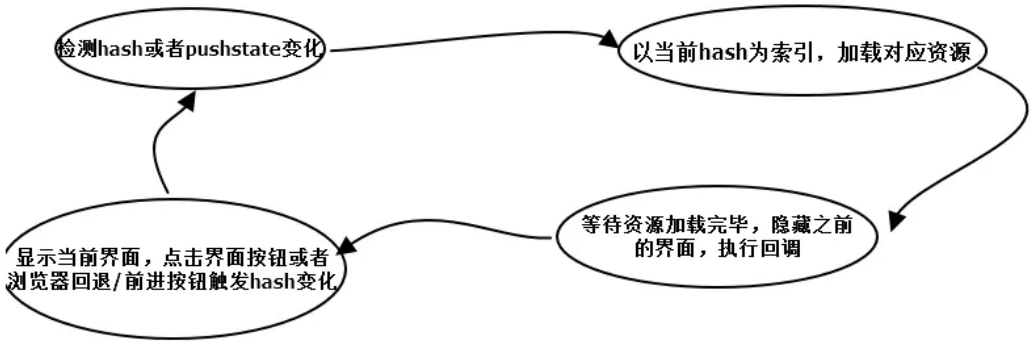
缺点：

- 不利于搜索引擎的抓取
- 首次渲染速度相对较慢
-

三、实现一个SPA

原理

1. 监听地址栏中`hash`变化驱动界面变化
2. 用`pushsate`记录浏览器的历史，驱动界面发送变化



实现

`hash` 模式

核心通过监听`url`中的`hash`来进行路由跳转

```
// 定义 Router
class Router {
    constructor () {
        this.routes = {}; // 存放路由path及callback
        this.currentUrl = '';

        // 监听路由change调用相对应的路由回调
        window.addEventListener('load', this.refresh, false);
        window.addEventListener('hashchange', this.refresh, false);
    }

    route(path, callback){
        this.routes[path] = callback;
    }

    push(path) {
        this.routes[path] && this.routes[path]()
    }
}

// 使用 router
window.miniRouter = new Router();
miniRouter.route('/', () => console.log('page1'))
miniRouter.route('/page2', () => console.log('page2'))

miniRouter.push('/') // page1
miniRouter.push('/page2') // page2
```

history模式

`history`模式核心借用`HTML5 history api`，`api`提供了丰富的`router`相关属性先了解一个几个相关的api

- `history.pushState`浏览器历史纪录添加记录
- `history.replaceState`修改浏览器历史纪录中当前纪录
- `history.popState`当`history`发生变化时触发

```
// 定义 Router
```

```
class Router {
    constructor () {
        this.routes = {};
        this.listerPopState()
    }

    init(path) {
        history.replaceState({path: path}, null, path);
        this.routes[path] && this.routes[path]();
    }

    route(path, callback){
        this.routes[path] = callback;
    }

    push(path) {
        history.pushState({path: path}, null, path);
        this.routes[path] && this.routes[path]();
    }

    listerPopState () {
        window.addEventListener('popstate' , e => {
            const path = e.state && e.state.path;
            this.routers[path] && this.routers[path]()
        })
    }
}

// 使用 Router

window.miniRouter = new Router();
miniRouter.route('/', ()=> console.log('page1'))
miniRouter.route('/page2', ()=> console.log('page2'))

// 跳转
miniRouter.push('/page2')  // page2
```

四、题外话：如何给SPA做SEO

下面给出基于 `Vue` 的 `SPA` 如何实现 `SEO` 的三种方式

1. **SSR服务端渲染**

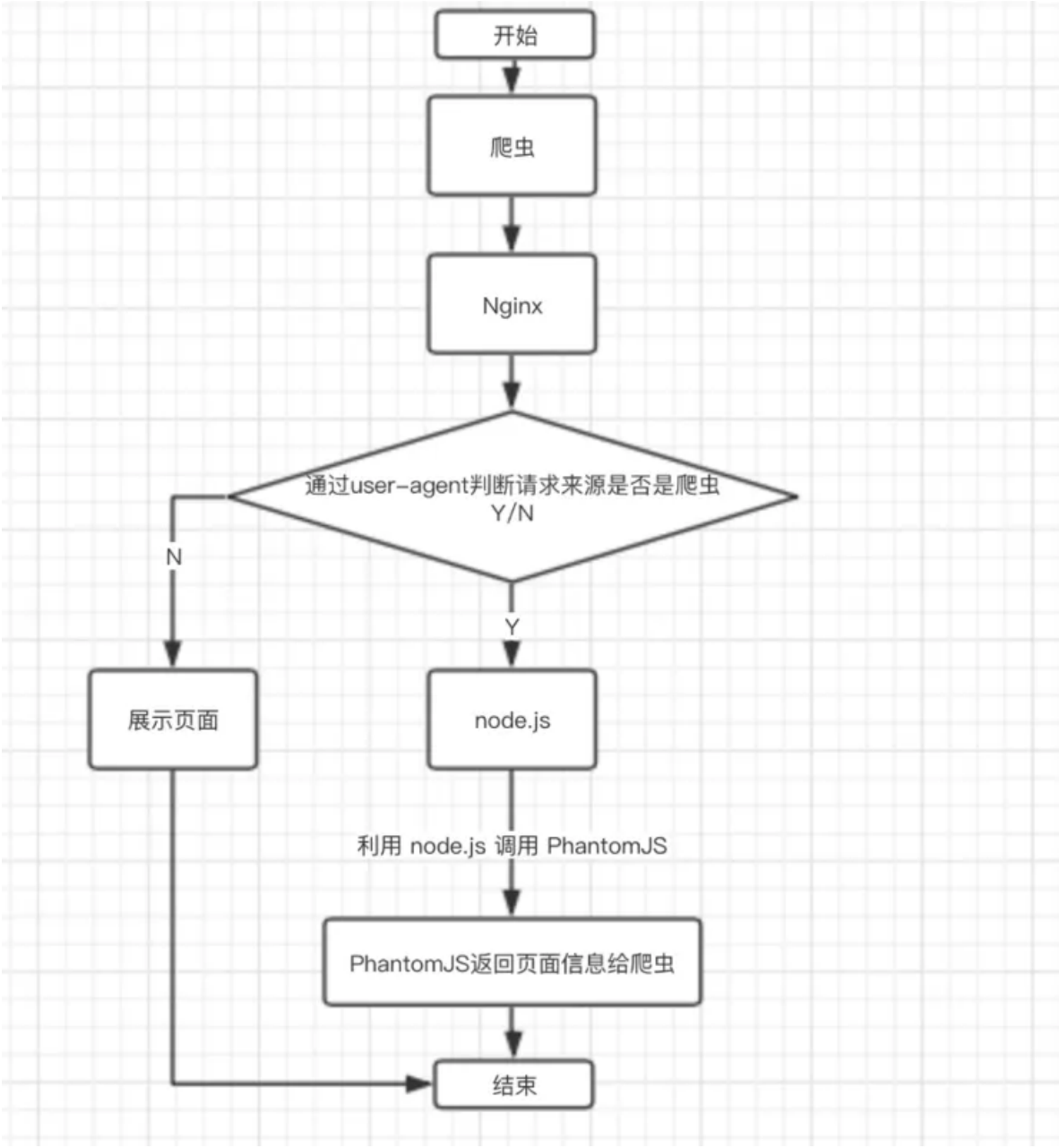
将组件或页面通过服务器生成html，再返回给浏览器，如 `nuxt.js`

2. **静态化**

目前主流的静态化主要有两种：（1）一种是通过程序将动态页面抓取并保存为静态页面，这样的页面的实际存在于服务器的硬盘中（2）另外一种是通过WEB服务器的 `URL Rewrite` 的方式，它的原理是通过web服务器内部模块按一定规则将外部的URL请求转化为内部的文件地址，一句话来说就是把外部请求的静态地址转化为实际的动态页面地址，而静态页面实际是不存在的。这两种方法都达到了实现URL静态化的效果

3. **使用 `Phantomjs` 针对爬虫处理**

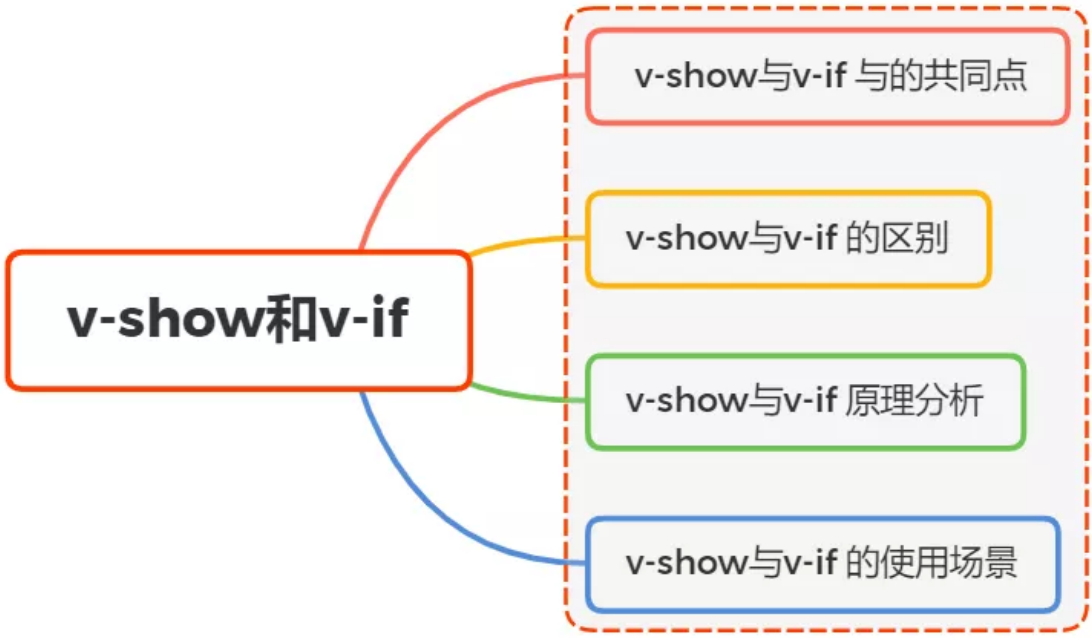
原理是通过 `Nginx` 配置，判断访问来源是否为爬虫，如果是则搜索引擎的爬虫请求会转发到一个 `node server`，再通过 `PhantomJS` 来解析完整的 `HTML`，返回给爬虫。下面是大致流程图



参考文献

- https://segmentfault.com/a/1190000019623624
- https://juejin.cn/post/6844903512107663368
- https://www.cnblogs.com/constantince/p/5586851.html

03.v-show和v-if有什么区别？使用场景分别是什么？



一、v-show与v-if的共同点

我们都知道在 `vue` 中 `v-show` 与 `v-if` 的作用效果是相同的(不含v-else)，都能控制元素在页面是否显示

在用法上也是相同的

```
<Model v-show="isShow" />
<Model v-if="isShow" />
```

- 当表达式为 `true` 的时候，都会占据页面的位置
- 当表达式都为 `false` 时，都不会占据页面位置

二、v-show与v-if的区别

- 控制手段不同
- 编译过程不同
- 编译条件不同

控制手段：`v-show`隐藏则是为该元素添加`css--display:none`，`dom`元素依旧还在。`v-if`显示隐藏是将`dom`元素整个添加或删除

编译过程：`v-if`切换有一个局部编译/卸载的过程，切换过程中合适地销毁和重建内部的事件监听和子组件；`v-show`只是简单的基于css切换

编译条件：`v-if`是真正的条件渲染，它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。只有渲染条件为假时，并不做操作，直到为真才渲染

- `v-show`由`false`变为`true`的时候不会触发组件的生命周期
- `v-if`由`false`变为`true`的时候，触发组件的`beforeCreate`、`create`、`beforeMount`、`mounted`钩子，由`true`变为`false`的时候触发组件的`beforeDestory`、`destoryed`方法

性能消耗：`v-if`有更高的切换消耗；`v-show`有更高的初始渲染消耗；

三、v-show与v-if原理分析

具体解析流程这里不展开讲，大致流程如下

- 将模板`template`转为`ast`结构的`JS`对象
- 用`ast`得到的`JS`对象拼装`render`和`staticRenderFns`函数
- `render`和`staticRenderFns`函数被调用后生成虚拟`VNODE`节点，该节点包含创建`DOM`节点所需信息
- `vm.patch`函数通过虚拟`DOM`算法利用`VNODE`节点创建真实`DOM`节点

v-show原理

不管初始条件是什么，元素总是会被渲染

我们看一下在`vue`中是如何实现的

代码很好理解，有`transition`就执行`transition`，没有就直接设置`display`属性

```
// https://github.com/vuejs/vue-next/blob/3cd30c5245da0733f9eb6f29d220f39c46518162/packages/runtime-dom/src/directives/vShow.ts
export const vShow: ObjectDirective<VShowElement> = {
  beforeMount(el, { value }, { transition }) {
    el._vod = el.style.display === 'none' ? '' : el.style.display
    if (transition && value) {
      transition.beforeEnter(el)
    } else {
      setDisplay(el, value)
    }
  },
  mounted(el, { value }, { transition }) {
    if (transition && value) {
      transition.enter(el)
    }
  },
  updated(el, { value, oldValue }, { transition }) {
    // ...
  },
  beforeUnmount(el, { value }) {
```

```
    setDisplay(el, value)
  }
}
```

v-if原理

`v-if` 在实现上比 `v-show` 要复杂的多，因为还有 `else` `else-if` 等条件需要处理，这里我们也只摘抄源码中处理 `v-if` 的一小部分

返回一个 `node` 节点， `render` 函数通过表达式的值来决定是否生成 `DOM`

```
// https://github.com/vuejs/vue-next/blob/cdc9f336fd/packages/compiler-core/src/transforms/vIf.ts
export const transformIf = createStructuralDirectiveTransform(
  /^(if|else|else-if)$/,
  (node, dir, context) => {
    return processIf(node, dir, context, (ifNode, branch, isRoot) => {
      // ...
      return () => {
        if (isRoot) {
          ifNode.codegenNode = createCodegenNodeForBranch(
            branch,
            key,
            context
          ) as IfConditionalExpression
        } else {
          // attach this branch's codegen node to the v-if root.
          const parentCondition = getParentCondition(ifNode.codegenNode!)
          parentCondition.alternate = createCodegenNodeForBranch(
            branch,
            key + ifNode.branches.length - 1,
            context
          )
        }
      }
    })
  }
)
```

四、v-show与v-if的使用场景

`v-if` 与 `v-show` 都能控制 `dom` 元素在页面的显示

`v-if` 相比 `v-show` 开销更大的（直接操作 `dom` 节点增加与删除）

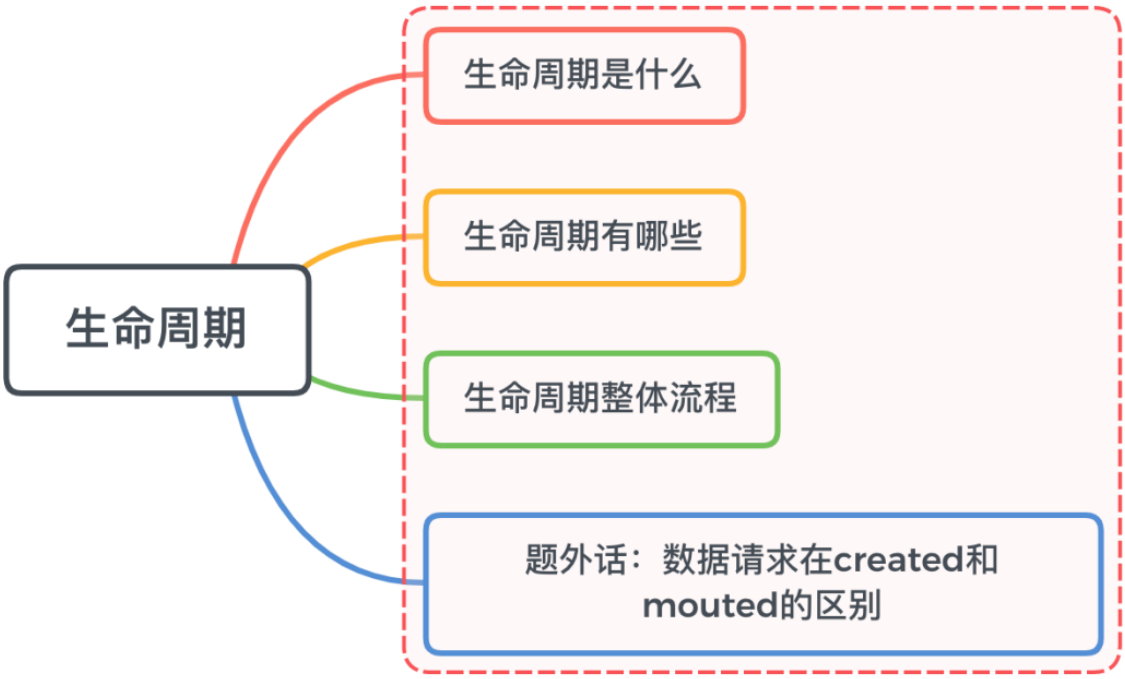
如果需要非常频繁地切换，则使用 v-show 较好

如果在运行时条件很少改变，则使用 v-if 较好

参考文献

- https://www.jianshu.com/p/7af8554d8f08
- https://juejin.cn/post/6897948855904501768
- https://vue3js/docs/zh

04.请描述下你对vue生命周期的理解？在created和mounted这两个生命周期中请求数据有什么区别呢?



一、生命周期是什么

生命周期`（Life Cycle）`的概念应用很广泛，特别是在政治、经济、环境、技术、社会等诸多领域经常出现，其基本涵义可以通俗地理解为“从摇篮到坟墓”`（Cradle-to-Grave）`的整个过程在`Vue`中实例从创建到销毁的过程就是生命周期，即指从创建、初始化数据、编译模板、挂载Dom→渲染、更新→渲染、卸载等一系列过程我们可以把组件比喻成工厂里面的一条流水线，每个工人（生命周期）站在各自的岗位，当任务流转到工人身边的时候，工人就开始工作PS：在`Vue`生命周期钩子会自动绑定`this`上下文到实例中，因此你可以访问数据，对`property`和方法进行运算这意味着**你不能使用箭头函数来定义一个生命周期方法**(例如`created: () => this.fetchTodos()`)

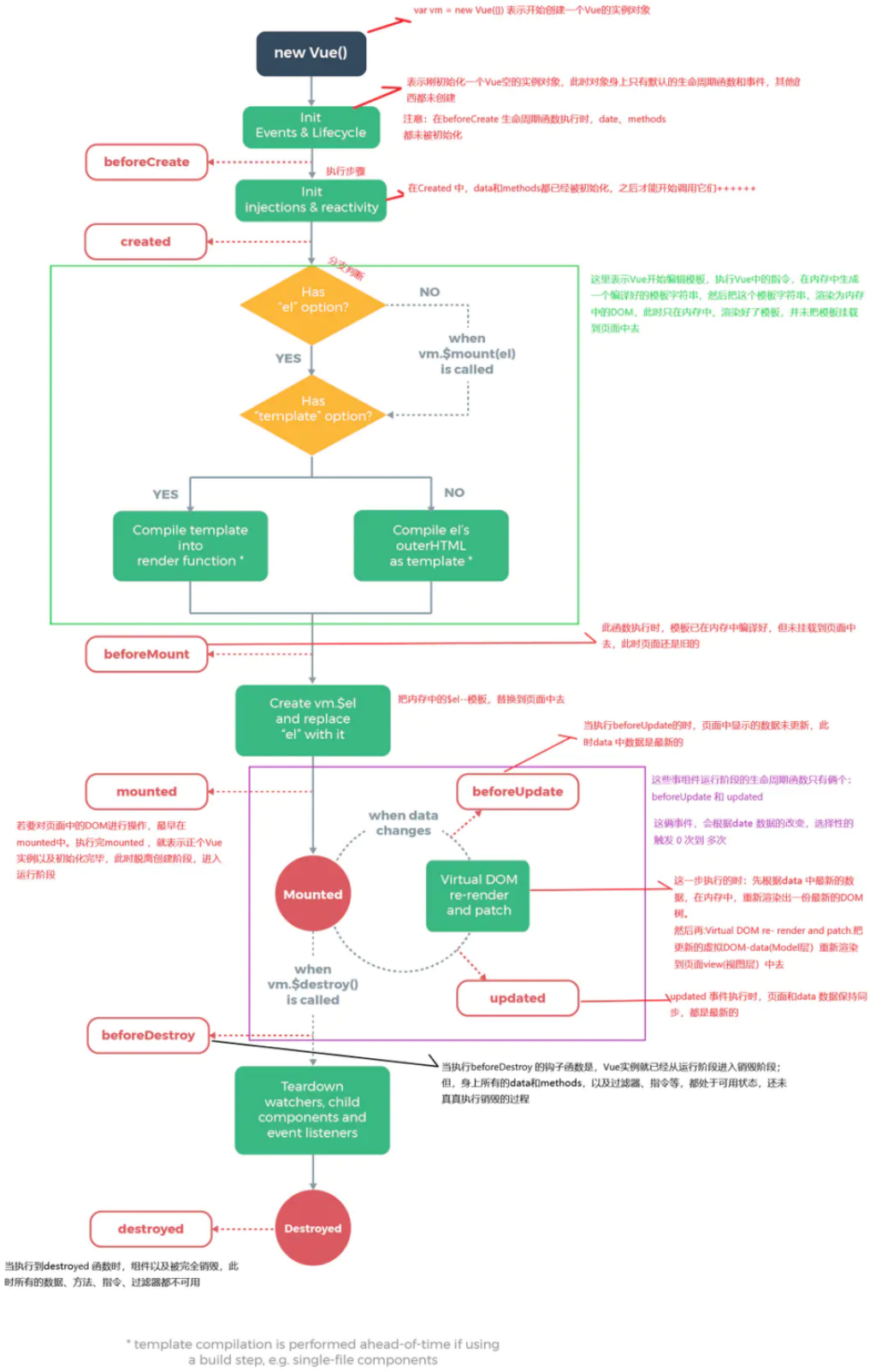
二、生命周期有哪些

Vue生命周期总共可以分为8个阶段：创建前后, 载入前后,更新前后,销毁前销毁后，以及一些特殊场景的生命周期

生命周期	描述
beforeCreate	组件实例被创建之初
created	组件实例已经完全创建
beforeMount	组件挂载之前
mounted	组件挂载到实例上去之后
beforeUpdate	组件数据发生变化，更新之前
updated	组件数据更新之后
beforeDestroy	组件实例销毁之前
destroyed	组件实例销毁之后
activated	keep-alive 缓存的组件激活时
deactivated	keep-alive 缓存的组件停用时调用
errorCaptured	捕获一个来自子孙组件的错误时被调用

三、生命周期整体流程

`vue` 生命周期流程图



具体分析

beforeCreate -> created

- 初始化 `vue` 实例，进行数据观测

created

- 完成数据观测，属性与方法的运算， `watch` 、 `event` 事件回调的配置

- 可调用`methods`中的方法，访问和修改data数据触发响应式渲染`dom`，可通过`computed`和`watch`完成数据计算
- 此时`vm.$el`并没有被创建

created -> beforeMount

- 判断是否存在`el`选项，若不存在则停止编译，直到调用`vm.$mount(el)`才会继续编译
- 优先级：`render` > `template` > `outerHTML`
- `vm.el`获取到的是挂载`DOM`的

beforeMount

- 在此阶段可获取到`vm.el`
- 此阶段`vm.el`虽已完成DOM初始化，但并未挂载在`el`选项上

beforeMount -> mounted

- 此阶段`vm.el`完成挂载，`vm.$el`生成的`DOM`替换了`el`选项所对应的`DOM`

mounted

- `vm.el`已完成`DOM`的挂载与渲染，此刻打印`vm.$el`，发现之前的挂载点及内容已被替换成新的DOM

beforeUpdate

- 更新的数据必须是被渲染在模板上的（`el`、`template`、`render`之一）
- 此时`view`层还未更新
- 若在`beforeUpdate`中再次修改数据，不会再次触发更新方法

updated

- 完成`view`层的更新
- 若在`updated`中再次修改数据，会再次触发更新方法（`beforeUpdate`、`updated`）

beforeDestroy

- 实例被销毁前调用，此时实例属性与方法仍可访问

destroyed

- 完全销毁一个实例。可清理它与其它实例的连接，解绑它的全部指令及事件监听器
- 并不能清除DOM，仅仅销毁实例

使用场景分析

生命周期	描述
beforeCreate	执行时组件实例还未创建，通常用于插件开发中执行一些初始化任务
created	组件初始化完毕，各种数据可以使用，常用于异步数据获取
beforeMount	未执行渲染、更新，dom未创建
mounted	初始化结束，dom已创建，可用于获取访问数据和dom元素
beforeUpdate	更新前，可用于获取更新前各种状态
updated	更新后，所有状态已是最新
beforeDestroy	销毁前，可用于一些定时器或订阅的取消
destroyed	组件已销毁，作用同上

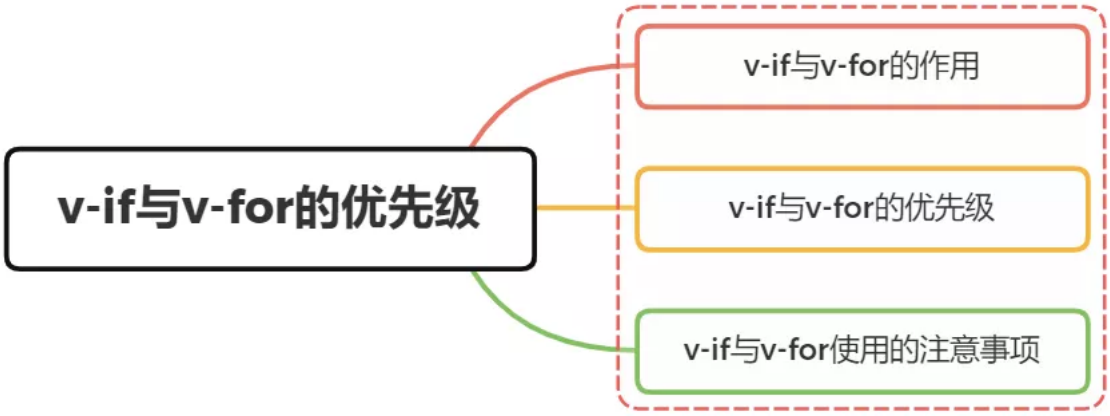
四、题外话：数据请求在created和mouted的区别

`created`是在组件实例一旦创建完成的时候立刻调用，这时候页面`dom`节点并未生成`mounted`是在页面`dom`节点渲染完毕之后就立刻执行的触发时机上`created`是比`mounted`要更早的两者相同点：都能拿到实例对象的属性和方法讨论这个问题本质就是触发的时机，放在`mounted`请求有可能导致页面闪动（页面`dom`结构已经生成），但如果在页面加载前完成则不会出现此情况建议：放在`create`生命周期当中

参考文献

- https://juejin.cn/post/6844903811094413320
- https://baike.baidu.com/
- http://cn.vuejs.org/

05.v-if和v-for的优先级是什么?



一、作用

`v-if`指令用于条件性地渲染一块内容。这块内容只会在指令的表达式返回`true`值的时候被渲染

`v-for`指令基于一个数组来渲染一个列表。`v-for`指令需要使用`item in items`形式的特殊语法，其中`items`是源数据数组或者对象，而`item`则是被迭代的数组元素的别名

在`v-for`的时候，建议设置`key`值，并且保证每个`key`值是独一无二的，这便于`diff`算法进行优化

两者在用法上

```
<Modal v-if="isShow" />

<li v-for="item in items" :key="item.id">
    {{ item.label }}
</li>
```

二、优先级

`v-if`与`v-for`都是`vue`模板系统中的指令

在`vue`模板编译的时候，会将指令系统转化成可执行的`render`函数

示例

编写一个`p`标签，同时使用`v-if`与`v-for`

```
<div id="app">
    <p v-if="isShow" v-for="item in items">
        {{ item.title }}
    </p>
</div>
```

创建`vue`实例，存放`isShow`与`items`数据

```
const app = new Vue({
  el: "#app",
  data() {
    return {
      items: [
        { title: "foo" },
        { title: "baz" }]
    }
  },
  computed: {
    isShow() {
      return this.items && this.items.length > 0
    }
  }
})
```

模板指令的代码都会生成在`render`函数中，通过`app.$options.render`就能得到渲染函数

```
ƒ anonymous() {
  with (this) { return
    _c('div', { attrs: { "id": "app" } },
    _l((items), function (item)
    { return (isShow) ? _c('p', [_v("\n" + _s(item.title) + "\n")]) : _e() }),
0) }
}
```

`_l`是`vue`的列表渲染函数，函数内部都会进行一次`if`判断

初步得到结论：`v-for`优先级是比`v-if`高

再将`v-for`与`v-if`置于不同标签

```
<div id="app">
    <template v-if="isShow">
        <p v-for="item in items">{{item.title}}</p>
    </template>
</div>
```

再输出下`render`函数

```
ƒ anonymous() {
  with(this){return
    _c('div',{attrs:{"id":"app"}},
    [(isShow)?[_v("\n"),
    _l((items),function(item){return _c('p',[_v(_s(item.title))])})]:_e()],2)}
}
```

这时候我们可以看到，`v-for`与`v-if`作用在不同标签时候，是先进行判断，再进行列表的渲染

我们再在查看下`vue`源码

源码位置：`\vue-dev\src\compiler\codegen\index.js`

```
export function genElement (el: ASTElement, state: CodegenState): string {
  if (el.parent) {
    el.pre = el.pre || el.parent.pre
  }
  if (el.staticRoot && !el.staticProcessed) {
    return genStatic(el, state)
  } else if (el.once && !el.onceProcessed) {
    return genOnce(el, state)
  } else if (el.for && !el.forProcessed) {
    return genFor(el, state)
  } else if (el.if && !el.ifProcessed) {
    return genIf(el, state)
  } else if (el.tag === 'template' && !el.slotTarget && !state.pre) {
    return genChildren(el, state) || 'void 0'
  } else if (el.tag === 'slot') {
    return genSlot(el, state)
  } else {
    // component or element
    ...
}
```

在进行`if`判断的时候，`v-for`是比`v-if`先进行判断

最终结论：`v-for`优先级比`v-if`高

三、注意事项

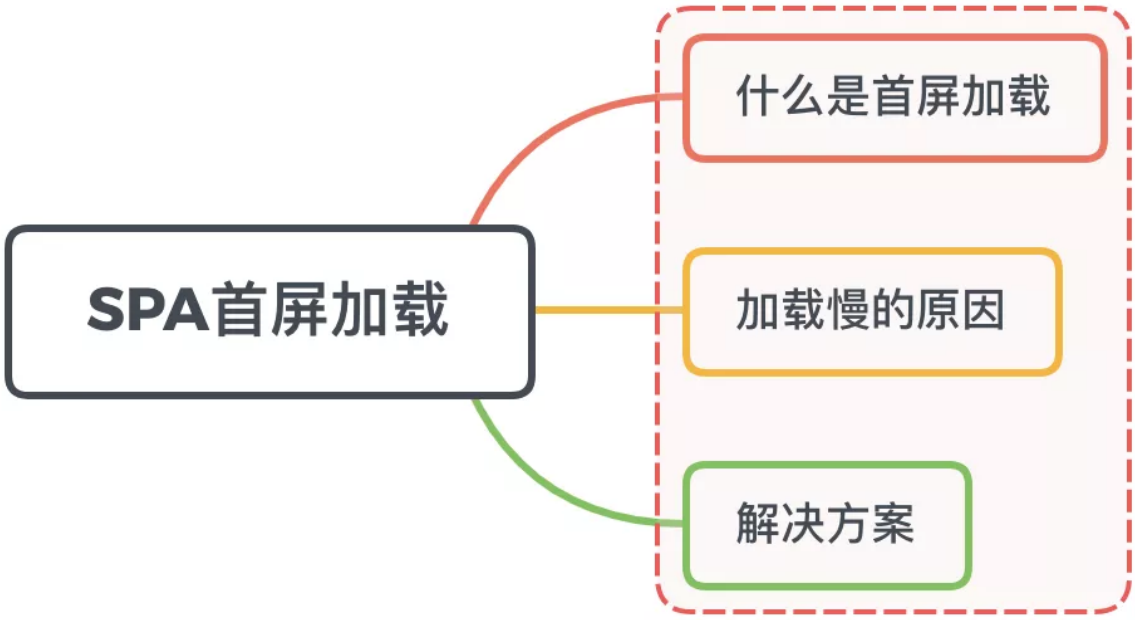
1. 永远不要把 `v-if` 和 `v-for` 同时用在同一个元素上，带来性能方面的浪费（每次渲染都会先循环再进行条件判断）
2. 如果避免出现这种情况，则在外层嵌套`template`（页面渲染不生成`dom`节点），在这一层进行v-if判断，然后在内部进行v-for循环

```
<template v-if="isShow">
    <p v-for="item in items">
</template>
```

3. 如果条件出现在循环内部，可通过计算属性`computed`提前过滤掉那些不需要显示的项

```
computed: {
    items: function() {
      return this.list.filter(function (item) {
        return item.isShow
      })
    }
}
```

06.SPA首屏加载速度慢的怎么解决?



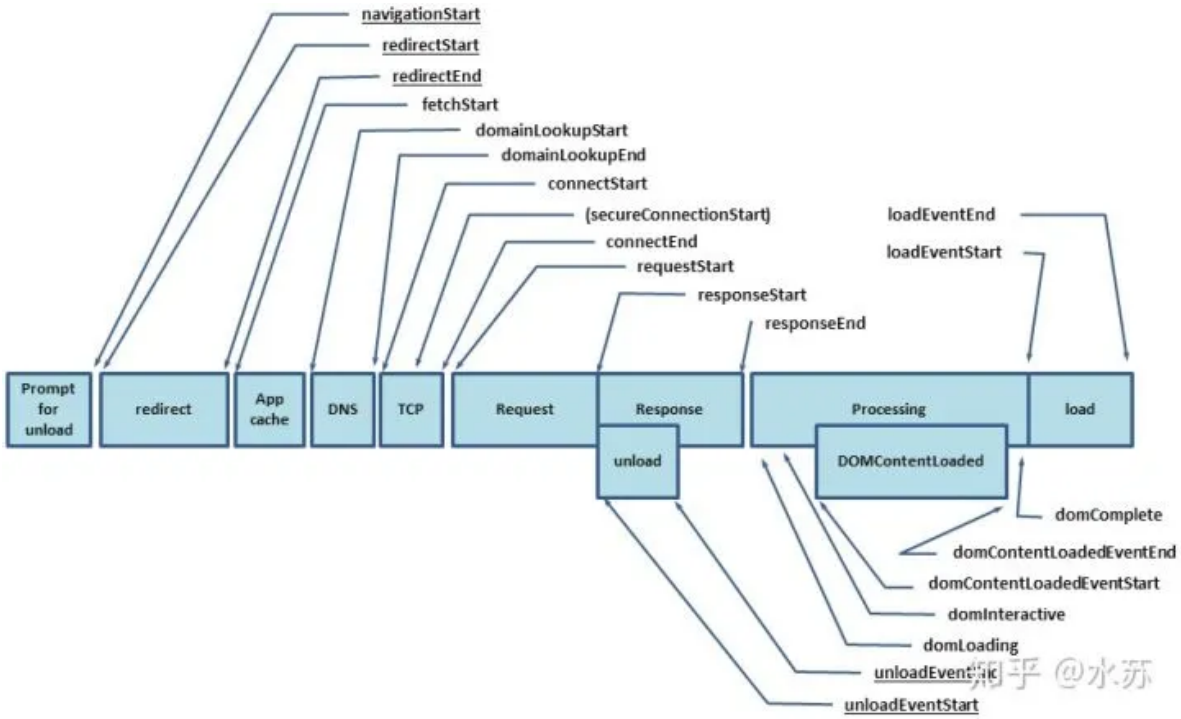
一、什么是首屏加载

首屏时间（First Contentful Paint），指的是浏览器从响应用户输入网址地址，到首屏内容渲染完成的时间，此时整个网页不一定要全部渲染完成，但需要展示当前视窗需要的内容

首屏加载可以说是用户体验中**最重要**的环节

关于计算首屏时间

利用`performance.timing`提供的数据：



通过`DOMContentLoad`或者`performance`来计算出首屏时间

```
// 方案一：
document.addEventListener('DOMContentLoaded', (event) => {
    console.log('first contentful painting');
});
// 方案二：
performance.getEntriesByName("first-contentful-paint")[0].startTime

// performance.getEntriesByName("first-contentful-paint")[0]
// 会返回一个 PerformancePaintTiming的实例，结构如下：
{
  name: "first-contentful-paint",
  entryType: "paint",
  startTime: 507.80000002123415,
  duration: 0,
};
```

二、加载慢的原因

在页面渲染的过程，导致加载速度慢的因素可能如下：

- 网络延时问题
- 资源文件体积是否过大
- 资源是否重复发送请求去加载了
- 加载脚本的时候，渲染内容堵塞了

三、解决方案

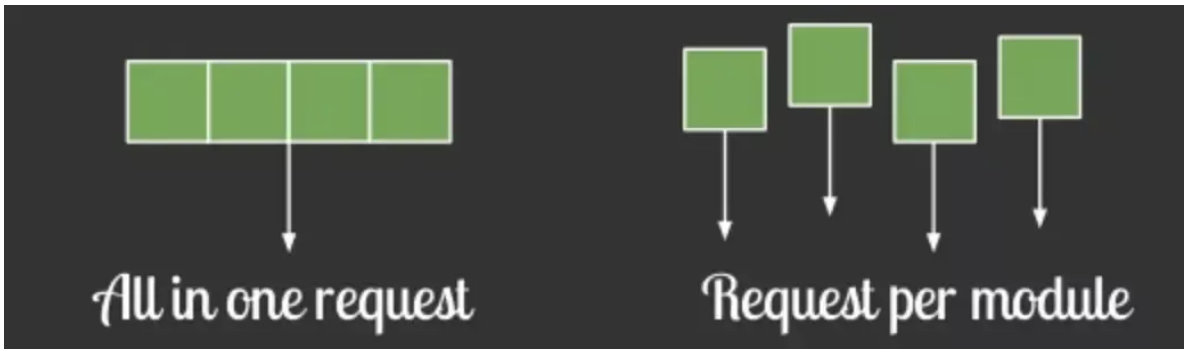
常见的几种SPA首屏优化方式

- 减小入口文件积
- 静态资源本地缓存

- UI框架按需加载
- 图片资源的压缩
- 组件重复打包
- 开启GZip压缩
- 使用SSR

减小入口文件体积

常用的手段是路由懒加载，把不同路由对应的组件分割成不同的代码块，待路由被请求的时候会单独打包路由，使得入口文件变小，加载速度大大增加



在`vue-router`配置路由的时候，采用动态加载路由的形式

```
routes:[
    path: 'Blogs',
    name: 'ShowBlogs',
    component: () => import('./components/ShowBlogs.vue')
]
```

以函数的形式加载路由，这样就可以把各自的路由文件分别打包，只有在解析给定的路由时，才会加载路由组件

静态资源本地缓存

后端返回资源问题：

- 采用`HTTP`缓存，设置`Cache-Control`，`Last-Modified`，`Etag`等响应头
- 采用`Service Worker`离线缓存

前端合理利用`localStorage`

UI框架按需加载

在日常使用`UI`框架，例如`element-UI`、或者`antd`，我们经常性直接饮用整个`UI`库

```
import ElementUI from 'element-ui'
Vue.use(ElementUI)
```

但实际上我用到的组件只有按钮，分页，表格，输入与警告 所以我们要按需引用

```
import { Button, Input, Pagination, Table, TableColumn, MessageBox } from
'element-ui';
Vue.use(Button)
Vue.use(Input)
Vue.use(Pagination)
```

组件重复打包

假设`A.js`文件是一个常用的库，现在有多个路由使用了`A.js`文件，这就造成了重复下载

解决方案：在`webpack`的`config`文件中，修改`CommonsChunkPlugin`的配置

```
minChunks: 3
```

`minChunks`为3表示会把使用3次及以上的包抽离出来，放进公共依赖文件，避免了重复加载组件

图片资源的压缩

图片资源虽然不在编码过程中，但它却是对页面性能影响最大的因素

对于所有的图片资源，我们可以进行适当的压缩

对页面上使用到的`icon`，可以使用在线字体图标，或者雪碧图，将众多小图标合并到同一张图上，用以减轻`http`请求压力。

开启GZip压缩

拆完包之后，我们再用`gzip`做一下压缩 安装`compression-webpack-plugin`

```
cnmp i compression-webpack-plugin -D
```

在`vue.congig.js`中引入并修改`webpack`配置

```
const CompressionPlugin = require('compression-webpack-plugin')

configureWebpack: (config) => {
        if (process.env.NODE_ENV === 'production') {
            // 为生产环境修改配置...
            config.mode = 'production'
            return {
                plugins: [new CompressionPlugin({
                    test: /\.js$|\.html$|\.css/, //匹配文件名
                    threshold: 10240, //对超过10k的数据进行压缩
                    deleteOriginalAssets: false //是否删除原文件
                })]
            }
        }
```

在服务器我们也要做相应的配置 如果发送请求的浏览器支持`gzip`，就发送给它`gzip`格式的文件 我的服务器是用`express`框架搭建的 只要安装一下`compression`就能使用

```
const compression = require('compression')
app.use(compression())  // 在其他中间件使用之前调用
```

使用SSR

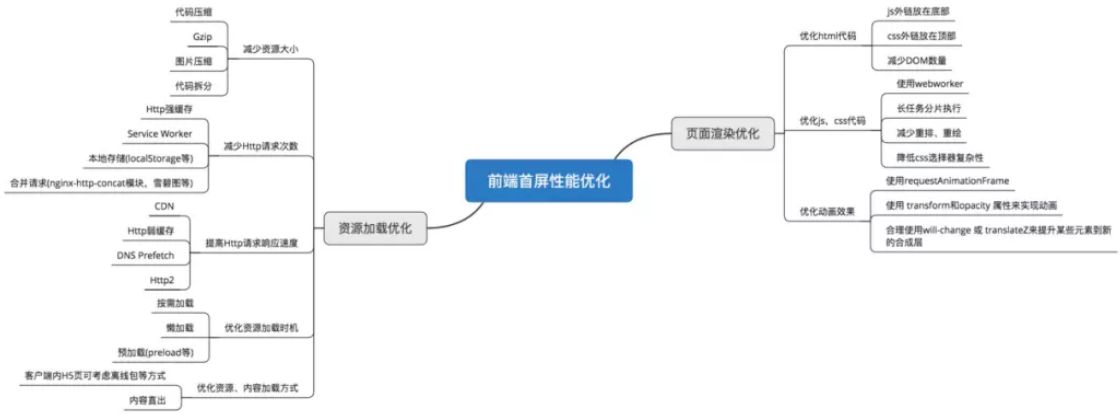
SSR（Server side ），也就是服务端渲染，组件或页面通过服务器生成html字符串，再发送到浏览器

从头搭建一个服务端渲染是很复杂的，`vue`应用建议使用`Nuxt.js`实现服务端渲染

小结：

减少首屏渲染时间的方法有很多，总的来讲可以分成两大部分： 资源加载优化 和 页面渲染优化

下图是更为全面的首屏优化的方案

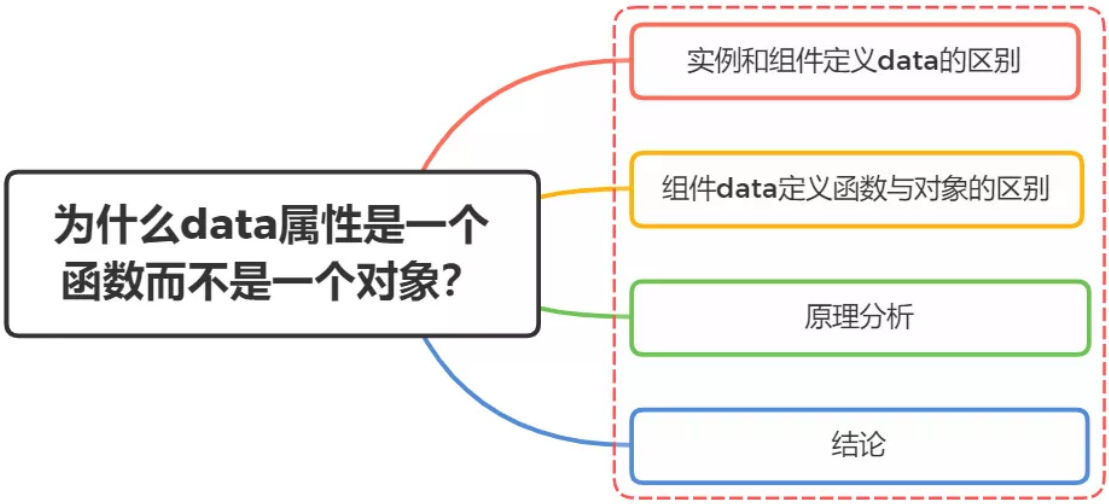


大家可以根据自己项目的情况选择各种方式进行首屏渲染的优化

参考文献

- https://zhuanlan.zhihu.com/p/88639980?utm_source=wechat_session
- https://www.chengrang.com/how-browsers-work.html
- https://juejin.cn/post/6844904185264095246
- https://vue3js.cn/docs/zh

07.为什么data属性是一个函数而不是一个对象



一、实例和组件定义data的区别

`vue`实例的时候定义`data`属性既可以是一个对象，也可以是一个函数

```
const app = new Vue({
    el:"#app",
    // 对象格式
    data:{
        foo:"foo"
    },
    // 函数格式
    data(){
        return {
            foo:"foo"
        }
    }
})
```

组件中定义`data`属性，只能是一个函数

如果为组件`data`直接定义为一个对象

```
Vue.component('component1',{
    template:`<div>组件</div>`,
    data:{
        foo:"foo"
    }
})
```

则会得到警告信息

```
▶[Vue warn]: The "data" option should be a function that returns a per-instance value in component definitions.   vue.js:634
```

警告说明：返回的`data`应该是一个函数在每一个组件实例中

二、组件data定义函数与对象的区别

上面讲到组件`data`必须是一个函数，不知道大家有没有思考过这是为什么呢?

在我们定义好一个组件的时候，`vue`最终都会通过`Vue.extend()`构成组件实例

这里我们模仿组件构造函数，定义`data`属性，采用对象的形式

```
function Component(){

}
Component.prototype.data = {
    count : 0
}
```

创建两个组件实例

```
const componentA = new Component()
const componentB = new Component()
```

修改`componentA`组件`data`属性的值，`componentB`中的值也发生了改变

```
console.log(componentB.data.count)  // 0
componentA.data.count = 1
console.log(componentB.data.count)  // 1
```

产生这样的原因这是两者共用了同一个内存地址，`componentA`修改的内容，同样对`componentB`产生了影响

如果我们采用函数的形式，则不会出现这种情况（函数返回的对象内存地址并不相同）

```
function Component(){
    this.data = this.data()
}
Component.prototype.data = function (){
    return {
        count : 0
    }
}
```

修改`componentA`组件`data`属性的值，`componentB`中的值不受影响

```
console.log(componentB.data.count)  // 0
componentA.data.count = 1
console.log(componentB.data.count)  // 0
```

`vue`组件可能会有很多个实例，采用函数返回一个全新`data`形式，使每个实例对象的数据不会受到其他实例对象数据的污染

三、原理分析

首先可以看看`vue`初始化`data`的代码，`data`的定义可以是函数也可以是对象

源码位置：`/vue-dev/src/core/instance/state.js`

```
function initData (vm: Component) {
  let data = vm.$options.data
  data = vm._data = typeof data === 'function'
    ? getData(data, vm)
    : data || {}
    ...
}
```

`data`既能是`object`也能是`function`，那为什么还会出现上文警告呢?

别急，继续看下文

组件在创建的时候，会进行选项的合并

源码位置：`/vue-dev/src/core/util/options.js`

自定义组件会进入`mergeOptions`进行选项合并

```
Vue.prototype._init = function (options?: Object) {
    ...
    // merge options
    if (options && options._isComponent) {
      // optimize internal component instantiation
      // since dynamic options merging is pretty slow, and none of the
      // internal component options needs special treatment.
      initInternalComponent(vm, options)
    } else {
      vm.$options = mergeOptions(
        resolveConstructorOptions(vm.constructor),
        options || {},
        vm
      )
    }
    ...
  }
```

定义`data`会进行数据校验

源码位置：`/vue-dev/src/core/instance/init.js`

这时候`vm`实例为`undefined`，进入`if`判断，若`data`类型不是`function`，则出现警告提示

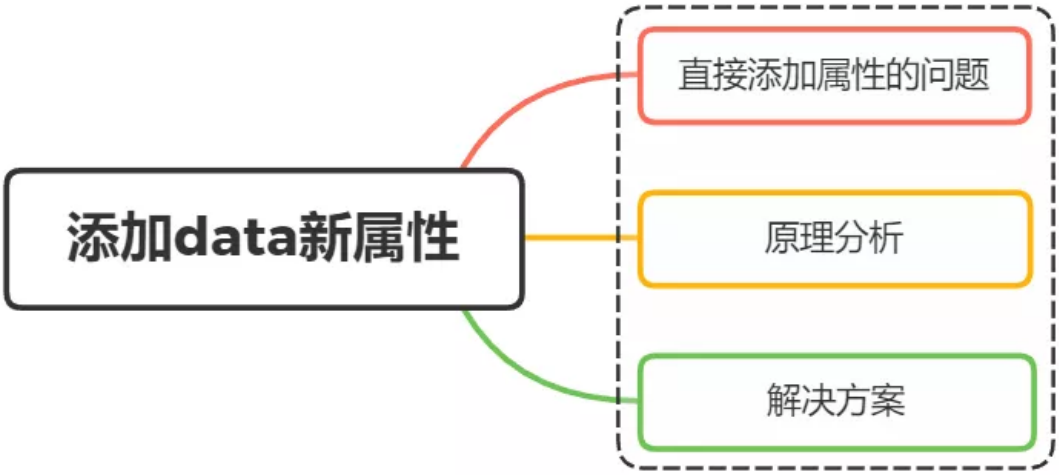
```
strats.data = function (
  parentVal: any,
  childVal: any,
  vm?: Component
): ?Function {
  if (!vm) {
    if (childVal && typeof childVal !== "function") {
      process.env.NODE_ENV !== "production" &&
        warn(
          'The "data" option should be a function ' +
            "that returns a per-instance value in component " +
            "definitions.",
          vm
        );
```

```
      return parentVal;
    }
    return mergeDataOrFn(parentVal, childVal);
  }
  return mergeDataOrFn(parentVal, childVal, vm);
};
```

四、结论

- 根实例对象`data`可以是对象也可以是函数（根实例是单例），不会产生数据污染情况
- 组件实例对象`data`必须为函数，目的是为了防止多个组件实例对象之间共用一个`data`，产生数据污染。采用函数的形式，`initData`时会将其作为工厂函数都会返回全新`data`对象

08.动态给vue的data添加一个新的属性时会发生什么？怎样解决?



一、直接添加属性的问题

我们从一个例子开始

定义一个`p`标签，通过`v-for`指令进行遍历

然后给`botton`标签绑定点击事件，我们预期点击按钮时，数据新增一个属性，界面也 新增一行

```
<p v-for="(value,key) in item" :key="key">
    {{ value }}
</p>
<button @click="addProperty">动态添加新属性</button>
```

实例化一个`vue`实例，定义`data`属性和`methods`方法

```
const app = new Vue({
    el:"#app",
    data:()=>{
        item:{
            oldProperty:"旧属性"
        }
    },
    methods:{
```

```
        addProperty(){
            this.items.newProperty = "新属性"  // 为items添加新属性
            console.log(this.items)  // 输出带有newProperty的items
        }
    }
})
```

点击按钮，发现结果不及预期，数据虽然更新了（`console`打印出了新属性），但页面并没有更新

二、原理分析

为什么产生上面的情况呢?

下面来分析一下

`vue2`是用过`Object.defineProperty`实现数据响应式

```
const obj = {}
Object.defineProperty(obj, 'foo', {
        get() {
            console.log(`get foo:${val}`);
            return val
        },
        set(newVal) {
            if (newVal !== val) {
                console.log(`set foo:${newVal}`);
                val = newVal
            }
        }
    })
}
```

当我们访问`foo`属性或者设置`foo`值的时候都能够触发`setter`与`getter`

```
obj.foo
obj.foo = 'new'
```

但是我们为`obj`添加新属性的时候，却无法触发事件属性的拦截

```
obj.bar  = '新属性'
```

原因是一开始`obj`的`foo`属性被设成了响应式数据，而`bar`是后面新增的属性，并没有通过`Object.defineProperty`设置成响应式数据

三、解决方案

`Vue`不允许在已经创建的实例上动态添加新的响应式属性

若想实现数据与视图同步更新，可采取下面三种解决方案:

- Vue.set()
- Object.assign()
- $forcecUpdated()

Vue.set()

Vue.set(target, propertyName/index, value)

参数

- `{Object | Array} target`
- `{string | number} propertyName/index`
- `{any} value`

返回值：设置的值

通过`Vue.set`向响应式对象中添加一个`property`，并确保这个新`property`同样是响应式的，且触发视图更新

关于`Vue.set`源码（省略了很多与本节不相关的代码）

源码位置：`src\core\observer\index.js`

```
function set (target: Array<any> | Object, key: any, val: any): any {
  ...
  defineReactive(ob.value, key, val)
  ob.dep.notify()
  return val
}
```

这里无非再次调用`defineReactive`方法，实现新增属性的响应式

关于`defineReactive`方法，内部还是通过`Object.defineProperty`实现属性拦截

大致代码如下：

```
function defineReactive(obj, key, val) {
    Object.defineProperty(obj, key, {
        get() {
            console.log(`get ${key}:${val}`);
            return val
        },
        set(newVal) {
            if (newVal !== val) {
                console.log(`set ${key}:${newVal}`);
                val = newVal
            }
        }
    })
}
```

Object.assign()

直接使用`Object.assign()`添加到对象的新属性不会触发更新

应创建一个新的对象，合并原对象和混入对象的属性

```
this.someObject = Object.assign({},this.someObject,
{newProperty1:1,newProperty2:2 ...})
```

$forceUpdate

如果你发现你自己需要在 `vue` 中做一次强制更新，99.9% 的情况，是你在某个地方做错了事

`$forceUpdate` 迫使 `vue` 实例重新渲染

PS：仅仅影响实例本身和插入插槽内容的子组件，而不是所有子组件。

小结

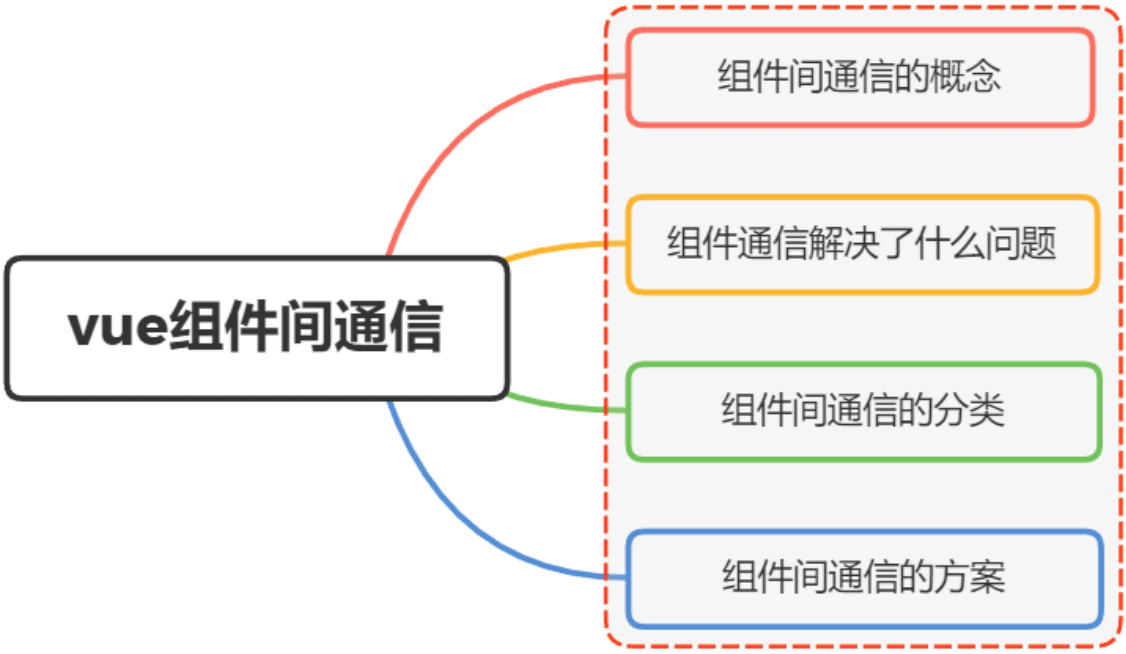
- 如果为对象添加少量的新属性，可以直接采用 `Vue.set()`
- 如果需要为新对象添加大量的新属性，则通过 `Object.assign()` 创建新对象
- 如果你实在不知道怎么操作时，可采取 `$forceUpdate()` 进行强制刷新 (不建议)

PS：`vue3` 是用过 `proxy` 实现数据响应式的，直接动态添加新属性仍可以实现数据响应式

参考文献

- https://cn.vuejs.org/v2/api/#Vue-set
- https://vue3js.cn/docs/zh

09.Vue组件之间的通信方式都有哪些?



一、组件间通信的概念

开始之前，我们把**组件间通信**这个词进行拆分

- 组件
- 通信

都知道组件是 `vue` 最强大的功能之一，`vue` 中每一个 `.vue` 我们都可以视之为一个组件通信指的是发送者通过某种媒体以某种格式来传递信息到收信者以达到某个目的。广义上，任何信息的交通都是通信**组件间通信**即指组件(`.vue`)通过某种方式来传递信息以达到某个目的举个栗子我们在使用 `UI` 框架中的 `table` 组件，可能会往 `table` 组件中传入某些数据，这个本质就形成了组件之间的通信

二、组件间通信解决了什么

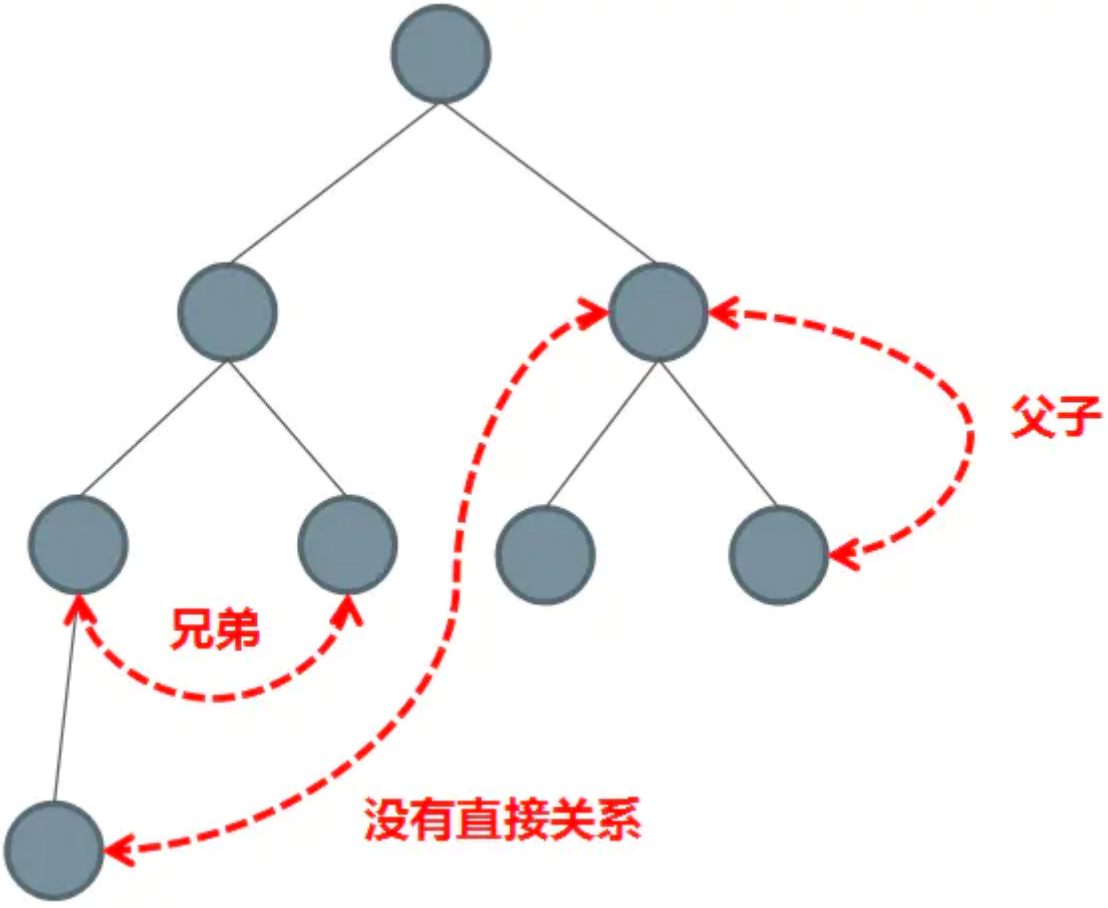
在古代，人们通过驿站、飞鸽传书、烽火报警、符号、语言、眼神、触碰等方式进行信息传递，到了今天，随着科技水平的飞速发展，通信基本完全利用有线或无线电完成，相继出现了有线电话、固定电话、无线电话、手机、互联网甚至视频电话等各种通信方式从上面这段话，我们可以看到通信的本质是信息同步，共享回到`vue`中，每个组件之间的都有独自的作用域，组件间的数据是无法共享的但实际开发工作中我们常常需要让组件之间共享数据，这也是组件通信的目的要让它们互相之间能进行通讯，这样才能构成一个有机的完整系统

二、组件间通信的分类

组件间通信的分类可以分成以下

- 父子组件之间的通信
- 兄弟组件之间的通信
- 祖孙与后代组件之间的通信
- 非关系组件间之间的通信

关系图:



三、组件间通信的方案

整理`vue`中8种常规的通信方案

1. 通过 props 传递
2. 通过 $emit 触发自定义事件
3. 使用 ref
4. EventBus
5. $parent 或$root

6. attrs 与 listeners
7. Provide 与 Inject
8. Vuex

props传递数据

- 适用场景：父组件传递数据给子组件
- 子组件设置 `props` 属性，定义接收父组件传递过来的参数
- 父组件在使用子组件标签中通过字面量来传递值

`Children.vue`

```
props:{
    // 字符串形式
 name:String // 接收的类型参数
    // 对象形式
    age:{
        type:Number, // 接收的类型为数值
        defaule:18,  // 默认值为18
       require:true // age属性必须传递
    }
}
```

`Father.vue` 组件

```
<Children name:"jack" age=18 />
```

$emit 触发自定义事件

- 适用场景：子组件传递数据给父组件
- 子组件通过 `$emit触发` 自定义事件，`$emit` 第二个参数为传递的数值
- 父组件绑定监听器获取到子组件传递过来的参数

`Chilfen.vue`

```
this.$emit('add', good)
```

`Father.vue`

```
<Children @add="cartAdd($event)" />
```

ref

- 父组件在使用子组件的时候设置`ref`
- 父组件通过设置子组件`ref`来获取数据

父组件

```
<Children ref="foo" />

this.$refs.foo  // 获取子组件实例，通过子组件实例我们就能拿到对应的数据
```

EventBus

- 使用场景：兄弟组件传值
- 创建一个中央时间总线`EventBus`
- 兄弟组件通过`$emit`触发自定义事件，`$emit`第二个参数为传递的数值
- 另一个兄弟组件通过`$on`监听自定义事件

`Bus.js`

```
// 创建一个中央时间总线类
class Bus {
  constructor() {
    this.callbacks = {};   // 存放事件的名字
  }
  $on(name, fn) {
    this.callbacks[name] = this.callbacks[name] || [];
    this.callbacks[name].push(fn);
  }
  $emit(name, args) {
    if (this.callbacks[name]) {
      this.callbacks[name].forEach((cb) => cb(args));
    }
  }
}

// main.js
Vue.prototype.$bus = new Bus() // 将$bus挂载到vue实例的原型上
// 另一种方式
Vue.prototype.$bus = new Vue() // Vue已经实现了Bus的功能
```

`Children1.vue`

```
this.$bus.$emit('foo')
```

`Children2.vue`

```
this.$bus.$on('foo', this.handle)
```

$parent 或$ root

- 通过共同祖辈`$parent`或者`$root`搭建通信侨联

兄弟组件

`this.$parent.on('add',this.add)`

另一个兄弟组件

`this.$parent.emit('add')`

$attrs 与$ listeners

- 适用场景：祖先传递数据给子孙
- 设置批量向下传属性 `$attrs` 和 `$listeners`
- 包含了父级作用域中不作为 `prop` 被识别 (且获取) 的特性绑定 (class 和 style 除外)。
- 可以通过 `v-bind="$attrs"` 传入内部组件

```
// child：并未在props中声明foo
<p>{{$attrs.foo}}</p>

// parent
<HelloWorld foo="foo"/>
```

```
// 给Grandson隔代传值，communication/index.vue
<Child2 msg="lalala" @some-event="onSomeEvent"></Child2>

// Child2做展开
<Grandson v-bind="$attrs" v-on="$listeners"></Grandson>

// Grandson使用
<div @click="$emit('some-event', 'msg from grandson')">
{{msg}}
</div>
```

provide 与 inject

- 在祖先组件定义`provide`属性，返回传递的值
- 在后代组件通过`inject`接收组件传递过来的值

祖先组件

```
provide(){
    return {
        foo:'foo'
    }
}
```

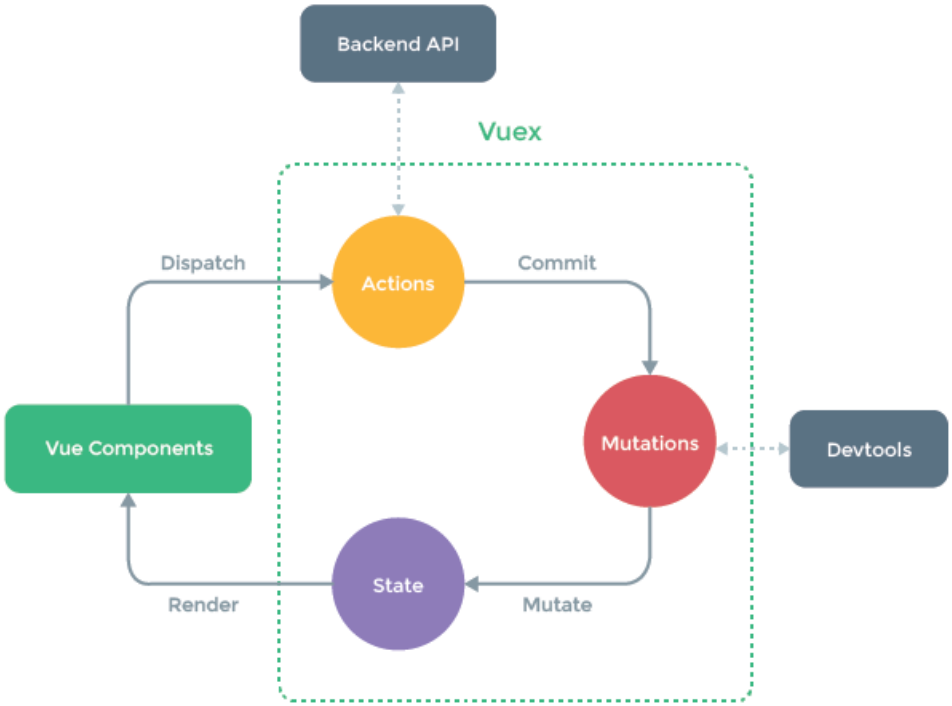
后代组件

```
inject:['foo'] // 获取到祖先组件传递过来的值
```

vuex

- 适用场景: 复杂关系的组件数据传递

- `Vuex`作用相当于一个用来存储共享变量的容器



- `state`用来存放共享变量的地方
- `getter`，可以增加一个`getter`派生状态，(相当于`store`中的计算属性），用来获得共享变量的值
- `mutations`用来存放修改`state`的方法。
- `actions`也是用来存放修改state的方法，不过`action`是在`mutations`的基础上进行。常用来做一些异步操作

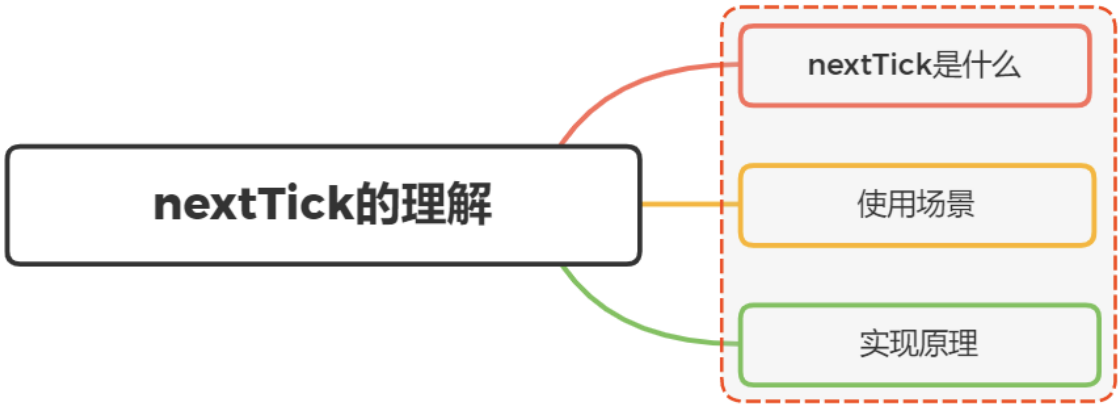
小结

- 父子关系的组件数据传递选择`props` 与 `$emit`进行传递，也可选择`ref`
- 兄弟关系的组件数据传递可选择`$bus`，其次可以选择`$parent`进行传递
- 祖先与后代组件数据传递可选择`attrs`与`listeners`或者 `Provide`与 `Inject`
- 复杂关系的组件数据传递可以通过`vuex`存放共享的变量

参考文献

- https://juejin.cn/post/6844903990052782094#heading-0
- https://zh.wikipedia.org/wiki/\%E9\%80\%9A\%E4\%BF\%A1
- https://vue3js.cn/docs/zh

10.Vue中的$nextTick有什么作用?



一、NextTick是什么

官方对其的定义

> 在下次 DOM 更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的 DOM

什么意思呢?

我们可以理解成，`Vue` 在更新 `DOM` 时是异步执行的。当数据发生变化，`Vue`将开启一个异步更新队列，视图需要等队列中所有数据变化完成之后，再统一进行更新

举例一下

`Html`结构

```
<div id="app"> {{ message }} </div>
```

构建一个`vue`实例

```
const vm = new Vue({
  el: '#app',
  data: {
    message: '原始值'
  }
})
```

修改`message`

```
this.message = '修改后的值1'
this.message = '修改后的值2'
this.message = '修改后的值3'
```

这时候想获取页面最新的`DOM`节点，却发现获取到的是旧值

```
console.log(vm.$el.textContent) // 原始值
```

这是因为`message`数据在发现变化的时候，`vue`并不会立刻去更新`Dom`，而是将修改数据的操作放在了一个异步操作队列中

如果我们一直修改相同数据，异步操作队列还会进行去重

等待同一事件循环中的所有数据变化完成之后，会将队列中的事件拿来进行处理，进行`DOM`的更新

为什么要有nexttick

举个例子

```
{{num}}
for(let i=0; i<100000; i++){
    num = i
}
```

如果没有 `nextTick` 更新机制，那么 `num` 每次更新值都会触发视图更新(上面这段代码也就是会更新10万次视图)，有了`nextTick`机制，只需要更新一次，所以`nextTick`本质是一种优化策略

二、使用场景

如果想要在修改数据后立刻得到更新后的`DOM`结构，可以使用`Vue.nextTick()`

第一个参数为：回调函数（可以获取最近的`DOM`结构）

第二个参数为：执行函数上下文

```
// 修改数据
vm.message = '修改后的值'
// DOM 还没有更新
console.log(vm.$el.textContent) // 原始的值
Vue.nextTick(function () {
  // DOM 更新了
  console.log(vm.$el.textContent) // 修改后的值
})
```

组件内使用 `vm.$nextTick()` 实例方法只需要通过 `this.$nextTick()`，并且回调函数中的 `this` 将自动绑定到当前的 `Vue` 实例上

```
this.message = '修改后的值'
console.log(this.$el.textContent) // => '原始的值'
this.$nextTick(function () {
    console.log(this.$el.textContent) // => '修改后的值'
})
```

`$nextTick()` 会返回一个 `Promise` 对象，可以是用`async/await`完成相同作用的事情

```
this.message = '修改后的值'
console.log(this.$el.textContent) // => '原始的值'
await this.$nextTick()
console.log(this.$el.textContent) // => '修改后的值'
```

三、实现原理

源码位置：`/src/core/util/next-tick.js`

`callbacks`也就是异步操作队列

`callbacks`新增回调函数后又执行了`timerFunc`函数，`pending`是用来标识同一个时间只能执行一次

```
export function nextTick(cb?: Function, ctx?: Object) {
  let _resolve;

  // cb 回调函数会经统一处理压入 callbacks 数组
  callbacks.push(() => {
    if (cb) {
      // 给 cb 回调函数执行加上了 try-catch 错误处理
      try {
        cb.call(ctx);
      } catch (e) {
        handleError(e, ctx, 'nextTick');
      }
    } else if (_resolve) {
      _resolve(ctx);
    }
  });

  // 执行异步延迟函数 timerFunc
  if (!pending) {
    pending = true;
    timerFunc();
  }

  // 当 nextTick 没有传入函数参数的时候，返回一个 Promise 化的调用
  if (!cb && typeof Promise !== 'undefined') {
    return new Promise(resolve => {
      _resolve = resolve;
    });
  }
}
```

`timerFunc`函数定义，这里是根据当前环境支持什么方法则确定调用哪个，分别有：

`Promise.then`、`MutationObserver`、`setImmediate`、`setTimeout`

通过上面任意一种方法，进行降级操作

```
export let isUsingMicroTask = false
if (typeof Promise !== 'undefined' && isNative(Promise)) {
  //判断1：是否原生支持Promise
  const p = Promise.resolve()
  timerFunc = () => {
    p.then(flushCallbacks)
    if (isIOS) setTimeout(noop)
  }
  isUsingMicroTask = true
} else if (!isIE && typeof MutationObserver !== 'undefined' && (
  isNative(MutationObserver) ||
  MutationObserver.toString() === '[object MutationObserverConstructor]'
)) {
  //判断2：是否原生支持MutationObserver
  let counter = 1
  const observer = new MutationObserver(flushCallbacks)
  const textNode = document.createTextNode(String(counter))
  observer.observe(textNode, {
    characterData: true
  })
```

```
  timerFunc = () => {
    counter = (counter + 1) % 2
    textNode.data = String(counter)
  }
  isUsingMicroTask = true
} else if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
  //判断3：是否原生支持setImmediate
  timerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else {
  //判断4：上面都不行，直接用setTimeout
  timerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}
```

无论是微任务还是宏任务，都会放到`flushCallbacks`使用

这里将`callbacks`里面的函数复制一份，同时`callbacks`置空

依次执行`callbacks`里面的函数

```
function flushCallbacks () {
  pending = false
  const copies = callbacks.slice(0)
  callbacks.length = 0
  for (let i = 0; i < copies.length; i++) {
    copies[i]()
  }
}
```

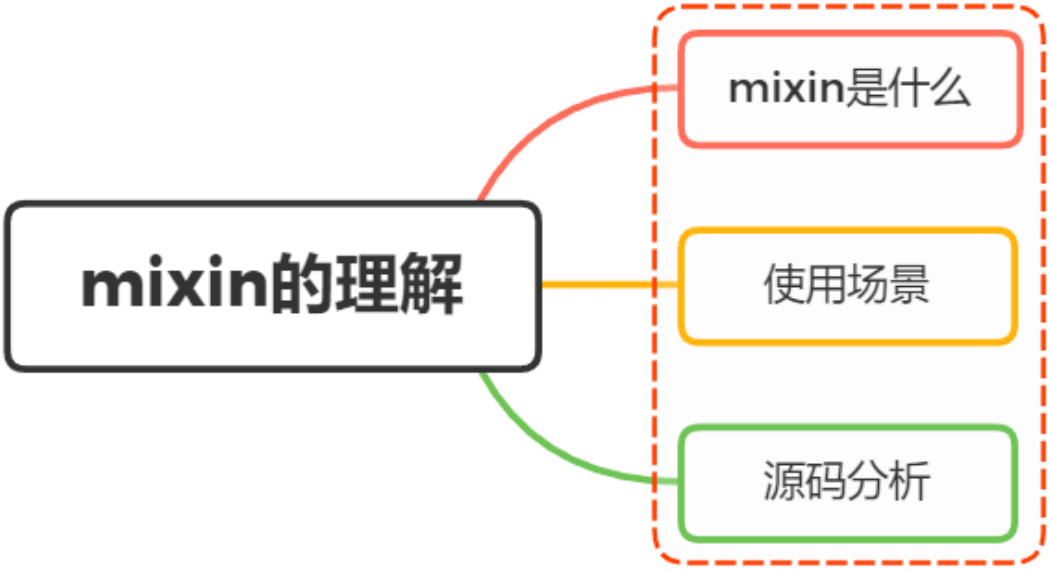
小结：

1. 把回调函数放入callbacks等待执行
2. 将执行函数放到微任务或者宏任务中
3. 事件循环到了微任务或者宏任务，执行函数依次执行callbacks中的回调

参考文献

- https://juejin.cn/post/6844904147804749832

11.说说你对vue的mixin的理解，有什么应用场景



一、mixin是什么

`Mixin`是面向对象程序设计语言中的类，提供了方法的实现。其他类可以访问`mixin`类的方法而不必成为其子类

`Mixin`类通常作为功能模块使用，在需要该功能时"混入"，有利于代码复用又避免了多继承的复杂

Vue中的mixin

先来看一下官方定义

> `mixin`（混入），提供了一种非常灵活的方式，来分发`Vue`组件中的可复用功能。

本质其实就是一个`js`对象，它可以包含我们组件中任意功能选项，如`data`、`components`、`methods`、`created`、`computed`等等

我们只要将共用的功能以对象的方式传入`mixins`选项中，当组件使用`mixins`对象时所有`mixins`对象的选项都将被混入该组件本身的选项中来

在`Vue`中我们可以**局部混入**跟**全局混入**

局部混入

定义一个`mixin`对象，有组件`options`的`data`、`methods`属性

```
var myMixin = {
  created: function () {
    this.hello()
  },
  methods: {
    hello: function () {
      console.log('hello from mixin!')
    }
  }
}
```

组件通过`mixins`属性调用`mixin`对象

```
Vue.component('componentA',{
  mixins: [myMixin]
})
```

该组件在使用的时候，混合了`mixin`里面的方法，在自动执行`create`生命钩子，执行`hello`方法

全局混入

通过`Vue.mixin()`进行全局的混入

```
Vue.mixin({
  created: function () {
      console.log("全局混入")
    }
})
```

使用全局混入需要特别注意，因为它会影响到每一个组件实例（包括第三方组件）

PS：全局混入常用于插件的编写

注意事项：

当组件存在与`mixin`对象相同的选项的时候，进行递归合并的时候组件的选项会覆盖`mixin`的选项

但是如果相同选项为生命周期钩子的时候，会合并成一个数组，先执行`mixin`的钩子，再执行组件的钩子

二、使用场景

在日常的开发中，我们经常会遇到在不同的组件中经常会需要用到一些相同或者相似的代码，这些代码的功能相对独立

这时，可以通过`Vue`的`mixin`功能将相同或者相似的代码提出来

举个例子

定义一个`modal`弹窗组件，内部通过`isShowing`来控制显示

```
const Modal = {
  template: '#modal',
  data() {
    return {
      isShowing: false
    }
  },
  methods: {
    toggleShow() {
      this.isShowing = !this.isShowing;
    }
  }
}
```

定义一个`tooltip`提示框，内部通过`isShowing`来控制显示

```
const Tooltip = {
  template: '#tooltip',
  data() {
    return {
      isShowing: false
    }
  },
  methods: {
    toggleShow() {
      this.isShowing = !this.isShowing;
    }
  }
}
```

通过观察上面两个组件，发现两者的逻辑是相同，代码控制显示也是相同的，这时候`mixin`就派上用场了

首先抽出共同代码，编写一个`mixin`

```
const toggle = {
  data() {
    return {
      isShowing: false
    }
  },
  methods: {
    toggleShow() {
      this.isShowing = !this.isShowing;
    }
  }
}
```

两个组件在使用上，只需要引入`mixin`

```
const Modal = {
  template: '#modal',
  mixins: [toggle]
};

const Tooltip = {
  template: '#tooltip',
  mixins: [toggle]
}
```

通过上面小小的例子，让我们知道了`Mixin`对于封装一些可复用的功能如此有趣、方便、实用

三、源码分析

首先从`Vue.mixin`入手

源码位置：/src/core/global-api/mixin.js

```
export function initMixin (Vue: GlobalAPI) {
  Vue.mixin = function (mixin: Object) {
    this.options = mergeOptions(this.options, mixin)
    return this
  }
}
```

主要是调用`merOptions`方法

源码位置：/src/core/util/options.js

```
export function mergeOptions (
  parent: Object,
  child: Object,
  vm?: Component
): Object {

if (child.mixins) { // 判断有没有mixin 也就是mixin里面挂mixin的情况 有的话递归进行合并
    for (let i = 0, l = child.mixins.length; i < l; i++) {
    parent = mergeOptions(parent, child.mixins[i], vm)
    }
}

  const options = {}
  let key
  for (key in parent) {
    mergeField(key) // 先遍历parent的key 调对应的strats[XXX]方法进行合并
  }
  for (key in child) {
    if (!hasOwn(parent, key)) { // 如果parent已经处理过某个key 就不处理了
      mergeField(key) // 处理child中的key 也就parent中没有处理过的key
    }
  }
  function mergeField (key) {
    const strat = strats[key] || defaultStrat
    options[key] = strat(parent[key], child[key], vm, key) // 根据不同类型的
options调用strats中不同的方法进行合并
  }
  return options
}
```

从上面的源码，我们得到以下几点：

- 优先递归处理`mixins`
- 先遍历合并`parent`中的`key`，调用`mergeField`方法进行合并，然后保存在变量`options`
- 再遍历`child`，合并补上`parent`中没有的`key`，调用`mergeField`方法进行合并，保存在变量`options`
- 通过`mergeField`函数进行了合并

下面是关于`Vue`的几种类型的合并策略

- 替换型
- 合并型
- 队列型
- 叠加型

替换型

替换型合并有`props`、`methods`、`inject`、`computed`

```
strats.props =
strats.methods =
strats.inject =
strats.computed = function (
  parentVal: ?Object,
  childVal: ?Object,
  vm?: Component,
  key: string
): ?Object {
  if (!parentVal) return childVal // 如果parentVal没有值，直接返回childVal
  const ret = Object.create(null) // 创建一个第三方对象 ret
  extend(ret, parentVal) // extend方法实际是把parentVal的属性复制到ret中
  if (childVal) extend(ret, childVal) // 把childVal的属性复制到ret中
  return ret
}
strats.provide = mergeDataOrFn
```

同名的`props`、`methods`、`inject`、`computed`会被后来者代替

合并型

和并型合并有：`data`

```
strats.data = function(parentVal, childVal, vm) {
    return mergeDataOrFn(
        parentVal, childVal, vm
    )
};

function mergeDataOrFn(parentVal, childVal, vm) {
    return function mergedInstanceDataFn() {
        var childData = childVal.call(vm, vm) // 执行data挂的函数得到对象
        var parentData = parentVal.call(vm, vm)
        if (childData) {
            return mergeData(childData, parentData) // 将2个对象进行合并

        } else {
            return parentData // 如果没有childData 直接返回parentData
        }
    }
}

function mergeData(to, from) {
    if (!from) return to
    var key, toVal, fromVal;
    var keys = Object.keys(from);
    for (var i = 0; i < keys.length; i++) {
        key = keys[i];
        toVal = to[key];
        fromVal = from[key];
        // 如果不存在这个属性，就重新设置
        if (!to.hasOwnProperty(key)) {
            set(to, key, fromVal);
```

```
        }
        // 存在相同属性，合并对象
        else if (typeof toVal =="object" && typeof fromVal =="object") {
            mergeData(toVal, fromVal);
        }
    }
    return to
}
```

`mergeData`函数遍历了要合并的 data 的所有属性，然后根据不同情况进行合并：

- 当目标 data 对象不包含当前属性时，调用 `set` 方法进行合并（set方法其实就是一些合并重新赋值的方法）
- 当目标 data 对象包含当前属性并且当前值为纯对象时，递归合并当前对象值，这样做是为了防止对象存在新增属性

队列性

队列性合并有：全部生命周期和`watch`

```
function mergeHook (
  parentVal: ?Array<Function>,
  childVal: ?Function | ?Array<Function>
): ?Array<Function> {
  return childVal
    ? parentVal
      ? parentVal.concat(childVal)
      : Array.isArray(childVal)
        ? childVal
        : [childVal]
    : parentVal
}

LIFECYCLE_HOOKS.forEach(hook => {
  strats[hook] = mergeHook
})

// watch
strats.watch = function (
  parentVal,
  childVal,
  vm,
  key
) {
  // work around Firefox's Object.prototype.watch...
  if (parentVal === nativeWatch) { parentVal = undefined; }
  if (childVal === nativeWatch) { childVal = undefined; }
  /* istanbul ignore if */
  if (!childVal) { return Object.create(parentVal || null) }
  {
    assertObjectType(key, childVal, vm);
  }
  if (!parentVal) { return childVal }
  var ret = {};
  extend(ret, parentVal);
  for (var key$1 in childVal) {
    var parent = ret[key$1];
```

```
    var child = childVal[key$1];
    if (parent && !Array.isArray(parent)) {
      parent = [parent];
    }
    ret[key$1] = parent
      ? parent.concat(child)
      : Array.isArray(child) ? child : [child];
  }
  return ret
};
```

生命周期钩子和`watch`被合并为一个数组，然后正序遍历一次执行

叠加型

叠加型合并有：`component`、`directives`、`filters`

```
strats.components=
strats.directives=

strats.filters = function mergeAssets(
    parentVal, childVal, vm, key
) {
    var res = Object.create(parentVal || null);
    if (childVal) {
        for (var key in childVal) {
            res[key] = childVal[key];
        }
    }
    return res
}
```

叠加型主要是通过原型链进行层层的叠加

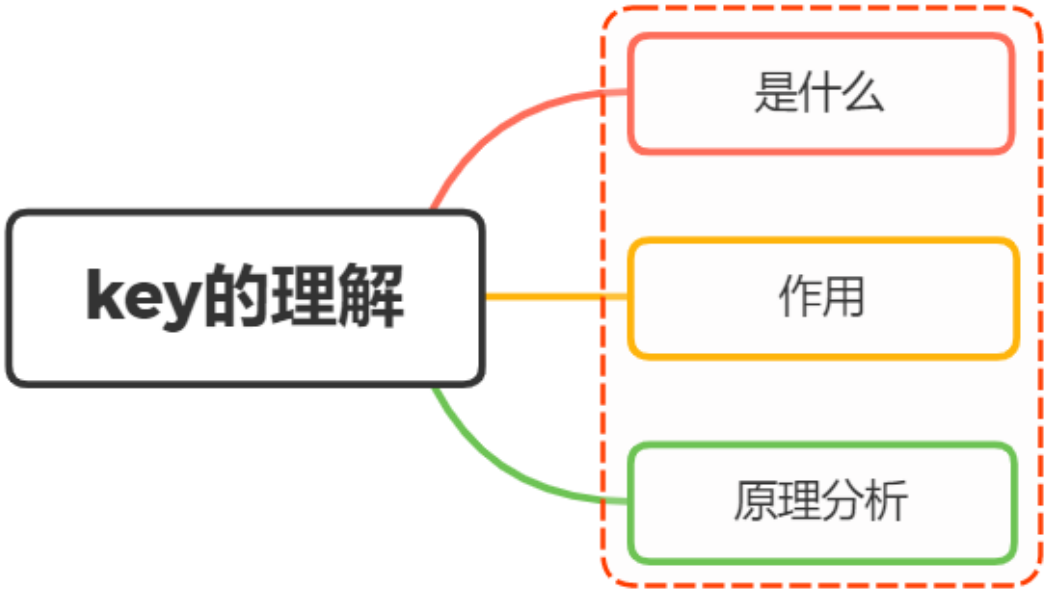
小结：

- 替换型策略有`props`、`methods`、`inject`、`computed`，就是将新的同名参数替代旧的参数
- 合并型策略是`data`，通过`set`方法进行合并和重新赋值
- 队列型策略有生命周期函数和`watch`，原理是将函数存入一个数组，然后正序遍历依次执行
- 叠加型有`component`、`directives`、`filters`，通过原型链进行层层的叠加

参考文献

- https://zhuanlan.zhihu.com/p/31018570
- https://juejin.cn/post/6844904015495446536#heading-1
- https://juejin.cn/post/6844903846775357453
- https://vue3js.cn/docs/zh

12.你知道vue中key的原理吗？说说你对它的理解



一、Key是什么

开始之前，我们先还原两个实际工作场景

1. 当我们在使用`v-for`时，需要给单元加上`key`

```
<ul>
    <li v-for="item in items" :key="item.id">...</li>
</ul>
```

2. 用`+new Date()`生成的时间戳作为`key`，手动强制触发重新渲染

```
<Comp :key="+new Date()" />
```

那么这背后的逻辑是什么，`key`的作用又是什么？

一句话来讲

> key是给每一个vnode的唯一id，也是diff的一种优化策略，可以根据key，更准确， 更快的找到对应的vnode节点

场景背后的逻辑

当我们在使用`v-for`时，需要给单元加上`key`

- 如果不用key，Vue会采用就地复地原则：最小化element的移动，并且会尝试尽最大程度在同适当的地方对相同类型的element，做patch或者reuse。
- 如果使用了key，Vue会根据keys的顺序记录element，曾经拥有了key的element如果不再出现的话，会被直接remove或者destoryed

用`+new Date()`生成的时间戳作为`key`，手动强制触发重新渲染

- 当拥有新值的rerender作为key时，拥有了新key的Comp出现了，那么旧key Comp会被移除，新key Comp触发渲染

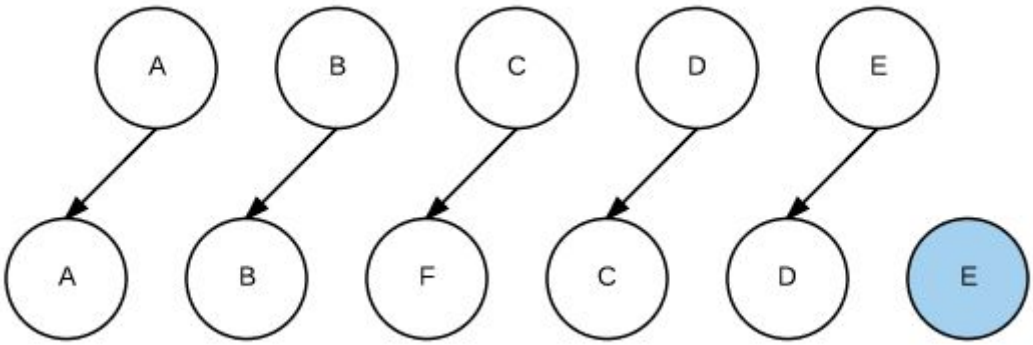
二、设置key与不设置key区别

举个例子：

创建一个实例，2秒后往`items`数组插入数据

```
<body>
  <div id="demo">
    <p v-for="item in items" :key="item">{{item}}</p>
  </div>
  <script src="../../dist/vue.js"></script>
  <script>
    // 创建实例
    const app = new Vue({
      el: '#demo',
      data: { items: ['a', 'b', 'c', 'd', 'e'] },
      mounted () {
        setTimeout(() => {
          this.items.splice(2, 0, 'f')  //
       }, 2000);
     },
   });
  </script>
</body>
```

在不使用`key`的情况，`vue`会进行这样的操作：



分析下整体流程：

- 比较A，A，相同类型的节点，进行`patch`，但数据相同，不发生`dom`操作
- 比较B，B，相同类型的节点，进行`patch`，但数据相同，不发生`dom`操作
- 比较C，F，相同类型的节点，进行`patch`，数据不同，发生`dom`操作
- 比较D，C，相同类型的节点，进行`patch`，数据不同，发生`dom`操作
- 比较E，D，相同类型的节点，进行`patch`，数据不同，发生`dom`操作
- 循环结束，将E插入到`DOM`中

一共发生了3次更新，1次插入操作

在使用`key`的情况：`vue`会进行这样的操作：

- 比较A，A，相同类型的节点，进行`patch`，但数据相同，不发生`dom`操作
- 比较B，B，相同类型的节点，进行`patch`，但数据相同，不发生`dom`操作
- 比较C，F，不相同类型的节点
  - 比较E、E，相同类型的节点，进行`patch`，但数据相同，不发生`dom`操作
- 比较D、D，相同类型的节点，进行`patch`，但数据相同，不发生`dom`操作
- 比较C、C，相同类型的节点，进行`patch`，但数据相同，不发生`dom`操作

- 循环结束，将F插入到C之前

一共发生了0次更新，1次插入操作

通过上面两个小例子，可见设置`key`能够大大减少对页面的`DOM`操作，提高了`diff`效率

设置key值一定能提高diff效率吗?

其实不然，文档中也明确表示

> 当 Vue.js 用 v-for 正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序， 而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素

这个默认的模式是高效的，但是只适用于不依赖子组件状态或临时 DOM 状态 (例如：表单输入值) 的列表渲染输出

建议尽可能在使用`v-for`时提供`key`，除非遍历输出的 DOM 内容非常简单，或者是刻意依赖默认行为以获取性能上的提升

三、原理分析

源码位置：core/vdom/patch.js

这里判断是否为同一个`key`，首先判断的是`key`值是否相等如果没有设置`key`，那么`key`为`undefined`，这时候`undefined`是恒等于`undefined`

```
function sameVnode (a, b) {
    return (
        a.key === b.key && (
            (
                a.tag === b.tag &&
                a.isComment === b.isComment &&
                isDef(a.data) === isDef(b.data) &&
                sameInputType(a, b)
            ) || (
                isTrue(a.isAsyncPlaceholder) &&
                a.asyncFactory === b.asyncFactory &&
                isUndef(b.asyncFactory.error)
            )
        )
    )
}
```

`updateChildren`方法中会对新旧`vnode`进行`diff`，然后将比对出的结果用来更新真实的`DOM`

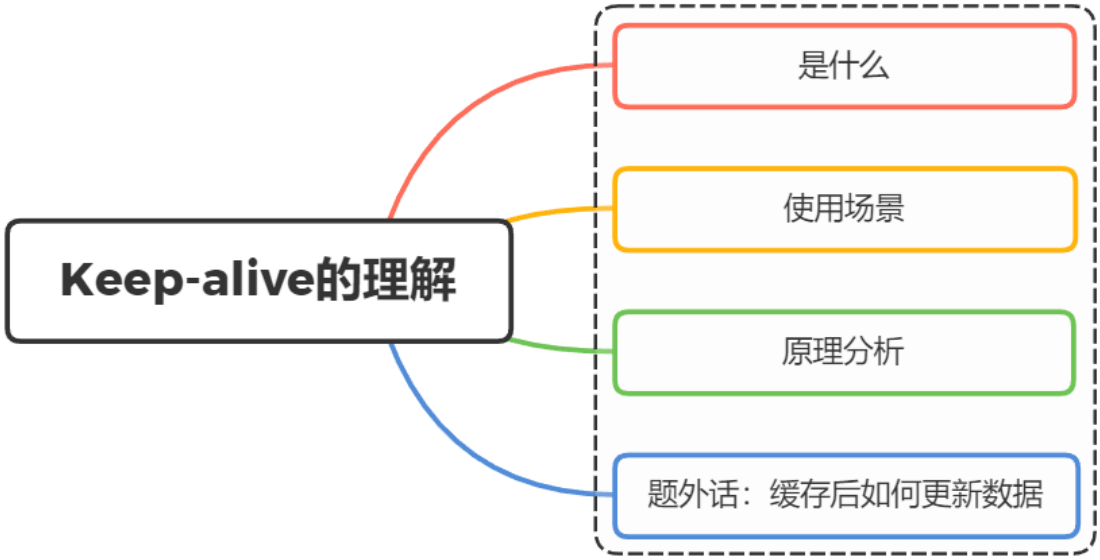
```
function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue,
removeOnly) {
    ...
    while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
        if (isUndef(oldStartVnode)) {
            ...
        } else if (isUndef(oldEndVnode)) {
            ...
        } else if (sameVnode(oldStartVnode, newStartVnode)) {
            ...
        } else if (sameVnode(oldEndVnode, newEndVnode)) {
            ...
```

```
        } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved right
            ...
        } else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode moved left
            ...
        } else {
            if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh,
oldStartIdx, oldEndIdx)
            idxInOld = isDef(newStartVnode.key)
                ? oldKeyToIdx[newStartVnode.key]
                : findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx)
            if (isUndef(idxInOld)) { // New element
                createElm(newStartVnode, insertedVnodeQueue, parentElm,
oldStartVnode.elm, false, newCh, newStartIdx)
            } else {
                vnodeToMove = oldCh[idxInOld]
                if (sameVnode(vnodeToMove, newStartVnode)) {
                    patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue,
newCh, newStartIdx)
                    oldCh[idxInOld] = undefined
                    canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm,
oldStartVnode.elm)
                } else {
                    // same key but different element. treat as new element
                    createElm(newStartVnode, insertedVnodeQueue, parentElm,
oldStartVnode.elm, false, newCh, newStartIdx)
                }
            }
            newStartVnode = newCh[++newStartIdx]
        }
    }
    ...
}
```

参考文献

- https://juejin.cn/post/6844903826693029895
- https://juejin.cn/post/6844903985397104648
- https://vue3js.cn/docs/zh

13.说说你对keep-alive的理解是什么?



一、Keep-alive 是什么

`keep-alive`是`vue`中的内置组件，能在组件切换过程中将状态保留在内存中，防止重复渲染`DOM`

`keep-alive`包裹动态组件时，会缓存不活动的组件实例，而不是销毁它们

`keep-alive`可以设置以下`props`属性：

- `include` - 字符串或正则表达式。只有名称匹配的组件会被缓存
- `exclude` - 字符串或正则表达式。任何名称匹配的组件都不会被缓存
- `max` - 数字。最多可以缓存多少组件实例

关于`keep-alive`的基本用法：

```
<keep-alive>
  <component :is="view"></component>
</keep-alive>
```

使用`includes`和`exclude`：

```
<keep-alive include="a,b">
  <component :is="view"></component>
</keep-alive>

<!-- 正则表达式 (使用 `v-bind`) -->
<keep-alive :include="/a|b/">
  <component :is="view"></component>
</keep-alive>

<!-- 数组 (使用 `v-bind`) -->
<keep-alive :include="['a', 'b']">
  <component :is="view"></component>
</keep-alive>
```

匹配首先检查组件自身的`name`选项，如果`name`选项不可用，则匹配它的局部注册名称 (父组件`components`选项的键值)，匿名组件不能被匹配

设置了 keep-alive 缓存的组件，会多出两个生命周期钩子（`activated`与`deactivated`）：

- 首次进入组件时：`beforeRouteEnter` > `beforeCreate` > `created` > `mounted` > `activated` > ... ... > `beforeRouteLeave` > `deactivated`
- 再次进入组件时：`beforeRouteEnter` > `activated` > ... ... > `beforeRouteLeave` > `deactivated`

二、使用场景

使用原则：当我们在某些场景下不需要让页面重新加载时我们可以使用`keepalive`

举个栗子:

当我们从`首页` -> `列表页` -> `商详页` -> `再返回`，这时候列表页应该是需要`keep-alive`

从`首页` -> `列表页` -> `商详页` -> `返回到列表页(需要缓存)` -> `返回到首页(需要缓存)` -> `再次进入列表页(不需要缓存)`，这时候可以按需来控制页面的`keep-alive`

在路由中设置`keepAlive`属性判断是否需要缓存

```
{
  path: 'list',
  name: 'itemList', // 列表页
  component (resolve) {
    require(['@/pages/item/list'], resolve)
 },
 meta: {
  keepAlive: true,
  title: '列表页'
 }
}
```

使用`<keep-alive>`

```
<div id="app" class='wrapper'>
    <keep-alive>
        <!-- 需要缓存的视图组件 -->
        <router-view v-if="$route.meta.keepAlive"></router-view>
     </keep-alive>
      <!-- 不需要缓存的视图组件 -->
     <router-view v-if="!$route.meta.keepAlive"></router-view>
</div>
```

三、原理分析

`keep-alive`是`vue`中内置的一个组件

源码位置：src/core/components/keep-alive.js

```
export default {
  name: 'keep-alive',
  abstract: true,

  props: {
    include: [String, RegExp, Array],
    exclude: [String, RegExp, Array],
    max: [String, Number]
  },
```

```
  created () {
    this.cache = Object.create(null)
    this.keys = []
  },

  destroyed () {
    for (const key in this.cache) {
      pruneCacheEntry(this.cache, key, this.keys)
    }
  },

  mounted () {
    this.$watch('include', val => {
      pruneCache(this, name => matches(val, name))
    })
    this.$watch('exclude', val => {
      pruneCache(this, name => !matches(val, name))
    })
  },

  render() {
    /* 获取默认插槽中的第一个组件节点 */
    const slot = this.$slots.default
    const vnode = getFirstComponentChild(slot)
    /* 获取该组件节点的componentOptions */
    const componentOptions = vnode && vnode.componentOptions

    if (componentOptions) {
      /* 获取该组件节点的名称，优先获取组件的name字段，如果name不存在则获取组件的tag */
      const name = getComponentName(componentOptions)

      const { include, exclude } = this
      /* 如果name不在inlcude中或者存在于exlude中则表示不缓存，直接返回vnode */
      if (
        (include && (!name || !matches(include, name))) ||
        // excluded
        (exclude && name && matches(exclude, name))
      ) {
        return vnode
      }

      const { cache, keys } = this
      /* 获取组件的key值 */
      const key = vnode.key == null
        // same constructor may get registered as different local components
        // so cid alone is not enough (#3269)
        ? componentOptions.Ctor.cid + (componentOptions.tag ?
`::${componentOptions.tag}` : '')
        : vnode.key
     /*  拿到key值后去this.cache对象中去寻找是否有该值，如果有则表示该组件有缓存，即命中缓存
*/
      if (cache[key]) {
        vnode.componentInstance = cache[key].componentInstance
        // make current key freshest
        remove(keys, key)
        keys.push(key)
      }
```

```
        /* 如果没有命中缓存，则将其设置进缓存 */
        else {
        cache[key] = vnode
        keys.push(key)
        // prune oldest entry
        /* 如果配置了max并且缓存的长度超过了this.max，则从缓存中删除第一个 */
        if (this.max && keys.length > parseInt(this.max)) {
          pruneCacheEntry(cache, keys[0], keys, this._vnode)
        }
      }

      vnode.data.keepAlive = true
    }
    return vnode || (slot && slot[0])
  }
}
```

可以看到该组件没有`template`，而是用了`render`，在组件渲染的时候会自动执行`render`函数

`this.cache`是一个对象，用来存储需要缓存的组件，它将以如下形式存储：

```
this.cache = {
    'key1':'组件1',
    'key2':'组件2',
    // ...
}
```

在组件销毁的时候执行`pruneCacheEntry`函数

```
function pruneCacheEntry (
  cache: VNodeCache,
  key: string,
  keys: Array<string>,
  current?: VNode
) {
  const cached = cache[key]
  /* 判断当前没有处于被渲染状态的组件，将其销毁*/
  if (cached && (!current || cached.tag !== current.tag)) {
    cached.componentInstance.$destroy()
  }
  cache[key] = null
  remove(keys, key)
}
```

在`mounted`钩子函数中观测`include`和`exclude`的变化，如下：

```
mounted () {
    this.$watch('include', val => {
        pruneCache(this, name => matches(val, name))
    })
    this.$watch('exclude', val => {
        pruneCache(this, name => !matches(val, name))
    })
}
```

如果 include 或 exclude 发生了变化，即表示定义需要缓存的组件的规则或者不需要缓存的组件的规则发生了变化，那么就执行 pruneCache 函数，函数如下：

```
function pruneCache (keepAliveInstance, filter) {
  const { cache, keys, _vnode } = keepAliveInstance
  for (const key in cache) {
    const cachedNode = cache[key]
    if (cachedNode) {
      const name = getComponentName(cachedNode.componentOptions)
      if (name && !filter(name)) {
        pruneCacheEntry(cache, key, keys, _vnode)
      }
    }
  }
}
```

在该函数内对 this.cache 对象进行遍历，取出每一项的 name 值，用其与新的缓存规则进行匹配，如果匹配不上，则表示在新的缓存规则下该组件已经不需要被缓存，则调用 pruneCacheEntry 函数将其从 this.cache 对象剔除即可

关于 keep-alive 的最强大缓存功能是在 render 函数中实现

首先获取组件的 key 值：

```
const key = vnode.key == null?
componentOptions.Ctor.cid + (componentOptions.tag ? `::${componentOptions.tag}`
: '')
: vnode.key
```

拿到 key 值后去 this.cache 对象中去寻找是否有该值，如果有则表示该组件有缓存，即命中缓存，如下：

```
/* 如果命中缓存，则直接从缓存中拿 vnode 的组件实例 */
if (cache[key]) {
    vnode.componentInstance = cache[key].componentInstance
    /* 调整该组件key的顺序，将其从原来的地方删掉并重新放在最后一个 */
    remove(keys, key)
    keys.push(key)
}
```

直接从缓存中拿 vnode 的组件实例，此时重新调整该组件 key 的顺序，将其从原来的地方删掉并重新放在 this.keys 中最后一个

this.cache 对象中没有该 key 值的情况，如下：

```
/* 如果没有命中缓存，则将其设置进缓存 */
else {
    cache[key] = vnode
    keys.push(key)
    /* 如果配置了max并且缓存的长度超过了this.max，则从缓存中删除第一个 */
    if (this.max && keys.length > parseInt(this.max)) {
        pruneCacheEntry(cache, keys[0], keys, this._vnode)
    }
}
```

表明该组件还没有被缓存过，则以该组件的`key`为键，组件`vnode`为值，将其存入`this.cache`中，并且把`key`存入`this.keys`中

此时再判断`this.keys`中缓存组件的数量是否超过了设置的最大缓存数量值`this.max`，如果超过了，则把第一个缓存组件删掉

四、思考题：缓存后如何获取数据

解决方案可以有以下两种：

- beforeRouteEnter
- actived

beforeRouteEnter

每次组件渲染的时候，都会执行`beforeRouteEnter`

```
beforeRouteEnter(to, from, next){
    next(vm=>{
        console.log(vm)
        // 每次进入路由执行
        vm.getData()  // 获取数据
    })
},
```

actived

在`keep-alive`缓存的组件被激活的时候，都会执行`actived`钩子

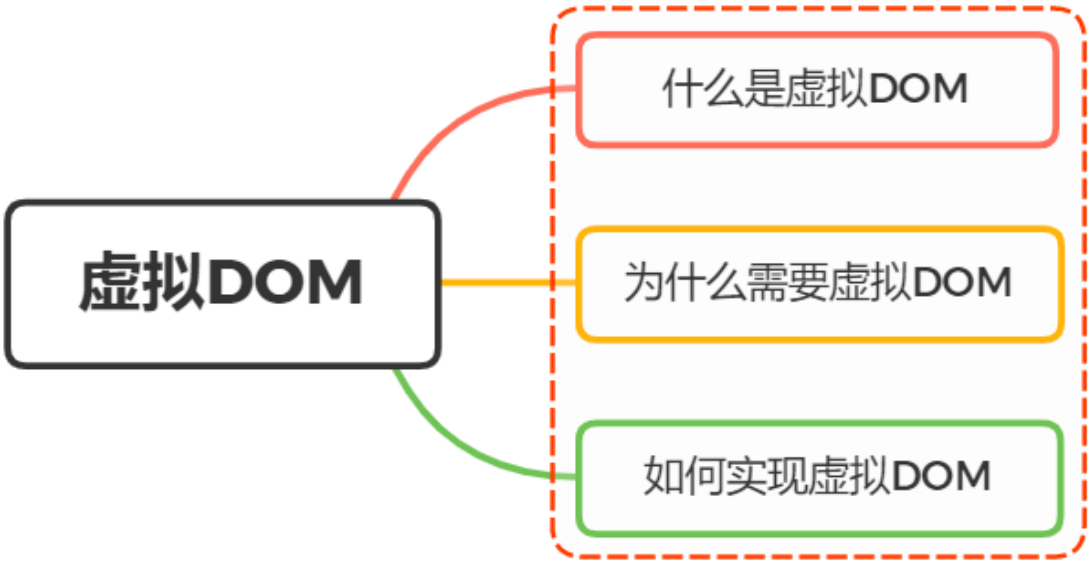
```
activated(){
   this.getData() // 获取数据
},
```

注意：服务器端渲染期间`avtived`不被调用

参考文献

- https://www.cnblogs.com/dhui/p/13589401.html
- https://www.cnblogs.com/wangjiachen666/p/11497200.html
- https://vue3js.cn/docs/zh

14.什么是虚拟DOM？如何实现一个虚拟DOM？说说你的思路



一、什么是虚拟DOM

虚拟 DOM （`Virtual DOM`）这个概念相信大家都不陌生，从 `React` 到 `Vue`，虚拟 `DOM` 为这两个框架都带来了跨平台的能力（`React-Native` 和 `Weex`）

实际上它只是一层对真实 `DOM` 的抽象，以 `JavaScript` 对象（`VNode` 节点）作为基础的树，用对象的属性来描述节点，最终可以通过一系列操作使这棵树映射到真实环境上

在 `Javascript` 对象中，虚拟 `DOM` 表现为一个 `Object` 对象。并且最少包含标签名（`tag`）、属性（`attrs`）和子元素对象（`children`）三个属性，不同框架对这三个属性的名命可能会有差别

创建虚拟 `DOM` 就是为了更好将虚拟的节点渲染到页面视图中，所以虚拟 `DOM` 对象的节点与真实 `DOM` 的属性一一照应

在 `vue` 中同样使用到了虚拟 `DOM` 技术

定义真实 `DOM`

```
<div id="app">
    <p class="p">节点内容</p>
    <h3>{{ foo }}</h3>
</div>
```

实例化 `vue`

```
const app = new Vue({
    el:"#app",
    data:{
        foo:"foo"
    }
})
```

观察 `render` 的 `render`，我们能得到虚拟 `DOM`

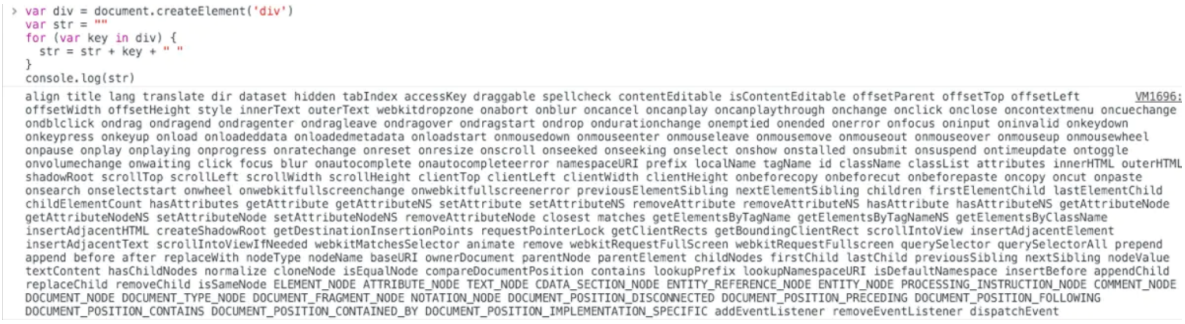
```
(function anonymous(
) {
    with(this){return _c('div',{attrs:{"id":"app"}},[_c('p',{staticClass:"p"},
                        [_v("节点内容")]),_v(" "),_c('h3',[_v(_s(foo))])])}})
```

通过 VNode，vue 可以对这颗抽象树进行创建节点,删除节点以及修改节点的操作，经过 diff 算法得出一些需要修改的最小单位,再更新视图，减少了 dom 操作，提高了性能

二、为什么需要虚拟DOM

DOM 是很慢的，其元素非常庞大，页面的性能问题，大部分都是由 DOM 操作引起的

真实的 DOM 节点，哪怕一个最简单的 div 也包含着很多属性，可以打印出来直观感受一下：



```
> var div = document.createElement('div')
  var str = ""
  for (var key in div) {
    str = str + key + " "
  }
  console.log(str)
  align title lang translate dir dataset hidden tabIndex accessKey draggable spellcheck contentEditable isContentEditable offsetParent offsetTop offsetLeft        VM1696:
  offsetWidth offsetHeight style innerText outerText webkitdropzone onabort onblur oncancel oncanplay oncanplaythrough onchange onclick onclose oncontextmenu oncuechange
  ondblclick ondrag ondragend ondragenter ondragleave ondragover ondragstart ondrop ondurationchange onemptied onended onerror onfocus oninput oninvalid onkeydown
  onkeypress onkeyup onload onloadeddata onloadedmetadata onloadstart onmousedown onmouseenter onmouseleave onmousemove onmouseout onmouseover onmouseup onmousewheel
  onpause onplay onplaying onprogress onratechange onreset onresize onscroll onseeked onseeking onselect onshow onstalled onsubmit onsuspend ontimeupdate ontoggle
  onvolumechange onwaiting click focus blur onautocomplete onautocompleteerror namespaceURI prefix localName tagName id className classList attributes innerHTML outerHTML
  shadowRoot scrollTop scrollLeft scrollWidth scrollHeight clientTop clientLeft clientWidth clientHeight onbeforecopy onbeforecut onbeforepaste oncopy oncut onpaste
  onsearch onselectstart onwheel onwebkitfullscreenchange onwebkitfullscreenerror previousElementSibling nextElementSibling children firstElementChild lastElementChild
  childElementCount hasAttributes getAttribute getAttributeNS setAttribute setAttributeNS removeAttribute removeAttributeNS hasAttribute hasAttributeNS getAttributeNode
  getAttributeNodeNS setAttributeNode setAttributeNodeNS removeAttributeNode closest matches getElementsByTagName getElementsByTagNameNS getElementsByClassName
  insertAdjacentHTML createShadowRoot getDestinationInsertionPoints requestPointerLock getClientRects getBoundingClientRect scrollIntoView insertAdjacentElement
  insertAdjacentText scrollIntoViewIfNeeded webkitMatchesSelector animate remove webkitRequestFullScreen webkitRequestFullscreen querySelector querySelectorAll prepend
  append before after replaceWith nodeType nodeName baseURI ownerDocument parentNode parentElement childNodes firstChild lastChild previousSibling nextSibling nodeValue
  textContent hasChildNodes normalize cloneNode isEqualNode compareDocumentPosition contains lookupPrefix lookupNamespaceURI isDefaultNamespace insertBefore appendChild
  replaceChild removeChild isSameNode ELEMENT_NODE ATTRIBUTE_NODE TEXT_NODE CDATA_SECTION_NODE ENTITY_REFERENCE_NODE ENTITY_NODE PROCESSING_INSTRUCTION_NODE COMMENT_NODE
  DOCUMENT_NODE DOCUMENT_TYPE_NODE DOCUMENT_FRAGMENT_NODE NOTATION_NODE DOCUMENT_POSITION_DISCONNECTED DOCUMENT_POSITION_PRECEDING DOCUMENT_POSITION_FOLLOWING
  DOCUMENT_POSITION_CONTAINS DOCUMENT_POSITION_CONTAINED_BY DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC addEventListener removeEventListener dispatchEvent
```

由此可见，操作 DOM 的代价仍旧是昂贵的，频繁操作还是会出现页面卡顿，影响用户的体验

举个例子：

你用传统的原生 api 或 jQuery 去操作 DOM 时，浏览器会从构建 DOM 树开始从头到尾执行一遍流程

当你在一次操作时，需要更新10个 DOM 节点，浏览器没这么智能，收到第一个更新 DOM 请求后，并不知道后续还有9次更新操作，因此会马上执行流程，最终执行10次流程

而通过 VNode，同样更新10个 DOM 节点，虚拟 DOM 不会立即操作 DOM，而是将这10次更新的 diff 内容保存到本地的一个 js 对象中，最终将这个 js 对象一次性 attach 到 DOM 树上，避免大量的无谓计算

> 很多人认为虚拟 DOM 最大的优势是 diff 算法，减少 JavaScript 操作真实 DOM 的带来的性能消耗。虽然这一个虚拟 DOM 带来的一个优势，但并不是全部。虚拟 DOM 最大的优势在于抽象了原本的渲染过程，实现了跨平台的能力，而不仅仅局限于浏览器的 DOM，可以是安卓和 IOS 的原生组件，可以是近期很火热的小程序，也可以是各种GUI

三、如何实现虚拟DOM

首先可以看看 vue 中 VNode 的结构

源码位置：src/core/vdom/vnode.js

```
export default class VNode {
  tag: string | void;
  data: VNodeData | void;
  children: ?Array<VNode>;
  text: string | void;
  elm: Node | void;
  ns: string | void;
  context: Component | void; // rendered in this component's scope
  functionalContext: Component | void; // only for functional component root
nodes
  key: string | number | void;
```

```
  componentOptions: VNodeComponentOptions | void;
  componentInstance: Component | void; // component instance
  parent: VNode | void; // component placeholder node
  raw: boolean; // contains raw HTML? (server only)
  isStatic: boolean; // hoisted static node
  isRootInsert: boolean; // necessary for enter transition check
  isComment: boolean; // empty comment placeholder?
  isCloned: boolean; // is a cloned node?
  isOnce: boolean; // is a v-once node?

  constructor (
    tag?: string,
    data?: VNodeData,
    children?: ?Array<VNode>,
    text?: string,
    elm?: Node,
    context?: Component,
    componentOptions?: VNodeComponentOptions
  ) {
    /*当前节点的标签名*/
    this.tag = tag
    /*当前节点对应的对象，包含了具体的一些数据信息，是一个VNodeData类型，可以参考VNodeData类型中的数据信息*/
    this.data = data
    /*当前节点的子节点，是一个数组*/
    this.children = children
    /*当前节点的文本*/
    this.text = text
    /*当前虚拟节点对应的真实dom节点*/
    this.elm = elm
    /*当前节点的名字空间*/
    this.ns = undefined
    /*编译作用域*/
    this.context = context
    /*函数化组件作用域*/
    this.functionalContext = undefined
    /*节点的key属性，被当作节点的标志，用以优化*/
    this.key = data && data.key
    /*组件的option选项*/
    this.componentOptions = componentOptions
    /*当前节点对应的组件的实例*/
    this.componentInstance = undefined
    /*当前节点的父节点*/
    this.parent = undefined
    /*简而言之就是是否为原生HTML或只是普通文本，innerHTML的时候为true，textContent的时候为false*/
    this.raw = false
    /*静态节点标志*/
    this.isStatic = false
    /*是否作为跟节点插入*/
    this.isRootInsert = true
    /*是否为注释节点*/
    this.isComment = false
    /*是否为克隆节点*/
    this.isCloned = false
    /*是否有v-once指令*/
    this.isOnce = false
  }
```

```
  // DEPRECATED: alias for componentInstance for backwards compat.
  /* istanbul ignore next https://github.com/answershuto/learnVue*/
  get child (): Component | void {
    return this.componentInstance
  }
}
```

这里对`VNode`进行稍微的说明：

- 所有对象的`context`选项都指向了`Vue`实例
- `elm`属性则指向了其相对应的真实`DOM`节点

`vue`是通过`createElement`生成`VNode`

源码位置：src/core/vdom/create-element.js

```
export function createElement (
  context: Component,
  tag: any,
  data: any,
  children: any,
  normalizationType: any,
  alwaysNormalize: boolean
): VNode | Array<VNode> {
  if (Array.isArray(data) || isPrimitive(data)) {
    normalizationType = children
    children = data
    data = undefined
  }
  if (isTrue(alwaysNormalize)) {
    normalizationType = ALWAYS_NORMALIZE
  }
  return _createElement(context, tag, data, children, normalizationType)
}
```

上面可以看到`createElement`方法实际上是对`_createElement`方法的封装，对参数的传入进行了判断

```
export function _createElement(
    context: Component,
    tag?: string | Class<Component> | Function | Object,
    data?: VNodeData,
    children?: any,
    normalizationType?: number
): VNode | Array<VNode> {
    if (isDef(data) && isDef((data: any).__ob__)) {
        process.env.NODE_ENV !== 'production' && warn(
            `Avoid using observed data object as vnode data:
${JSON.stringify(data)}\n` +
            'Always create fresh vnode data objects in each render!',
            context`
        )
        return createEmptyVNode()
    }
    // object syntax in v-bind
    if (isDef(data) && isDef(data.is)) {
```

```
        tag = data.is
    }
    if (!tag) {
        // in case of component :is set to falsy value
        return createEmptyVNode()
    }
    ...
    // support single function children as default scoped slot
    if (Array.isArray(children) &&
        typeof children[0] === 'function'
    ) {
        data = data || {}
        data.scopedSlots = { default: children[0] }
        children.length = 0
    }
    if (normalizationType === ALWAYS_NORMALIZE) {
        children = normalizeChildren(children)
    } else if ( === SIMPLE_NORMALIZE) {
        children = simpleNormalizeChildren(children)
    }
    // 创建VNode
    ...
}
```

可以看到`_createElement`接收5个参数：

- `context`表示`VNode`的上下文环境，是`Component`类型
- tag表示标签，它可以是一个字符串，也可以是一个`Component`
- `data`表示`VNode`的数据，它是一个`VNodeData`类型
- `children`表示当前`VNode`的子节点，它是任意类型的
- `normalizationType`表示子节点规范的类型，类型不同规范的方法也就不一样，主要是参考`render`函数是编译生成的还是用户手写的

根据`normalizationType`的类型，`children`会有不同的定义

```
if (normalizationType === ALWAYS_NORMALIZE) {
    children = normalizeChildren(children)
} else if ( === SIMPLE_NORMALIZE) {
    children = simpleNormalizeChildren(children)
}
```

`simpleNormalizeChildren`方法调用场景是`render`函数是编译生成的

`normalizeChildren`方法调用场景分为下面两种：

- `render`函数是用户手写的
- 编译`slot`、`v-for`的时候会产生嵌套数组

无论是`simpleNormalizeChildren`还是`normalizeChildren`都是对`children`进行规范（使`children`变成了一个类型为`VNode`的`Array`），这里就不展开说了

规范化`children`的源码位置在：src/core/vdom/helpers/normalzie-children.js

在规范化`children`后，就去创建`VNode`

```
let vnode, ns
// 对tag进行判断
```

```
if (typeof tag === 'string') {
  let Ctor
  ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
  if (config.isReservedTag(tag)) {
    // 如果是内置的节点，则直接创建一个普通VNode
    vnode = new VNode(
      config.parsePlatformTagName(tag), data, children,
      undefined, undefined, context
    )
  } else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {
    // component
    // 如果是component类型，则会通过createComponent创建VNode节点
    vnode = createComponent(Ctor, data, context, children, tag)
  } else {
    vnode = new VNode(
      tag, data, children,
      undefined, undefined, context
    )
  }
} else {
  // direct component options / constructor
  vnode = createComponent(tag, data, context, children)
}
```

`createComponent` 同样是创建 `VNode`

源码位置：src/core/vdom/create-component.js

```
export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag?: string
): VNode | Array<VNode> | void {
  if (isUndef(Ctor)) {
    return
  }
 // 构建子类构造函数
  const baseCtor = context.$options._base

  // plain options object: turn it into a constructor
  if (isObject(Ctor)) {
    Ctor = baseCtor.extend(Ctor)
  }

  // if at this stage it's not a constructor or an async component factory,
  // reject.
  if (typeof Ctor !== 'function') {
    if (process.env.NODE_ENV !== 'production') {
      warn(`Invalid Component definition: ${String(Ctor)}`, context)
    }
    return
  }

  // async component
  let asyncFactory
```

```
  if (isUndef(Ctor.cid)) {
    asyncFactory = Ctor
    Ctor = resolveAsyncComponent(asyncFactory, baseCtor, context)
    if (Ctor === undefined) {
      return createAsyncPlaceholder(
        asyncFactory,
        data,
        context,
        children,
        tag
      )
    }
  }

  data = data || {}

  // resolve constructor options in case global mixins are applied after
  // component constructor creation
  resolveConstructorOptions(Ctor)

  // transform component v-model data into props & events
  if (isDef(data.model)) {
    transformModel(Ctor.options, data)
  }

  // extract props
  const propsData = extractPropsFromVNodeData(data, Ctor, tag)

  // functional component
  if (isTrue(Ctor.options.functional)) {
    return createFunctionalComponent(Ctor, propsData, data, context, children)
  }

  // extract listeners, since these needs to be treated as
  // child component listeners instead of DOM listeners
  const listeners = data.on
  // replace with listeners with .native modifier
  // so it gets processed during parent component patch.
  data.on = data.nativeOn

  if (isTrue(Ctor.options.abstract)) {
    const slot = data.slot
    data = {}
    if (slot) {
      data.slot = slot
    }
  }

  // 安装组件钩子函数，把钩子函数合并到data.hook中
  installComponentHooks(data)

  //实例化一个VNode返回。组件的VNode是没有children的
  const name = Ctor.options.name || tag
  const vnode = new VNode(
    `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
    data, undefined, undefined, undefined, context,
    { Ctor, propsData, listeners, tag, children },
    asyncFactory
```

```
  )
  if (__WEEX__ && isRecyclableComponent(vnode)) {
    return renderRecyclableComponentTemplate(vnode)
  }

  return vnode
}
```

稍微提下`createComponent`生成`VNode`的三个关键流程：

- 构造子类构造函数`Ctor`
- `installComponentHooks`安装组件钩子函数
- 实例化`vnode`

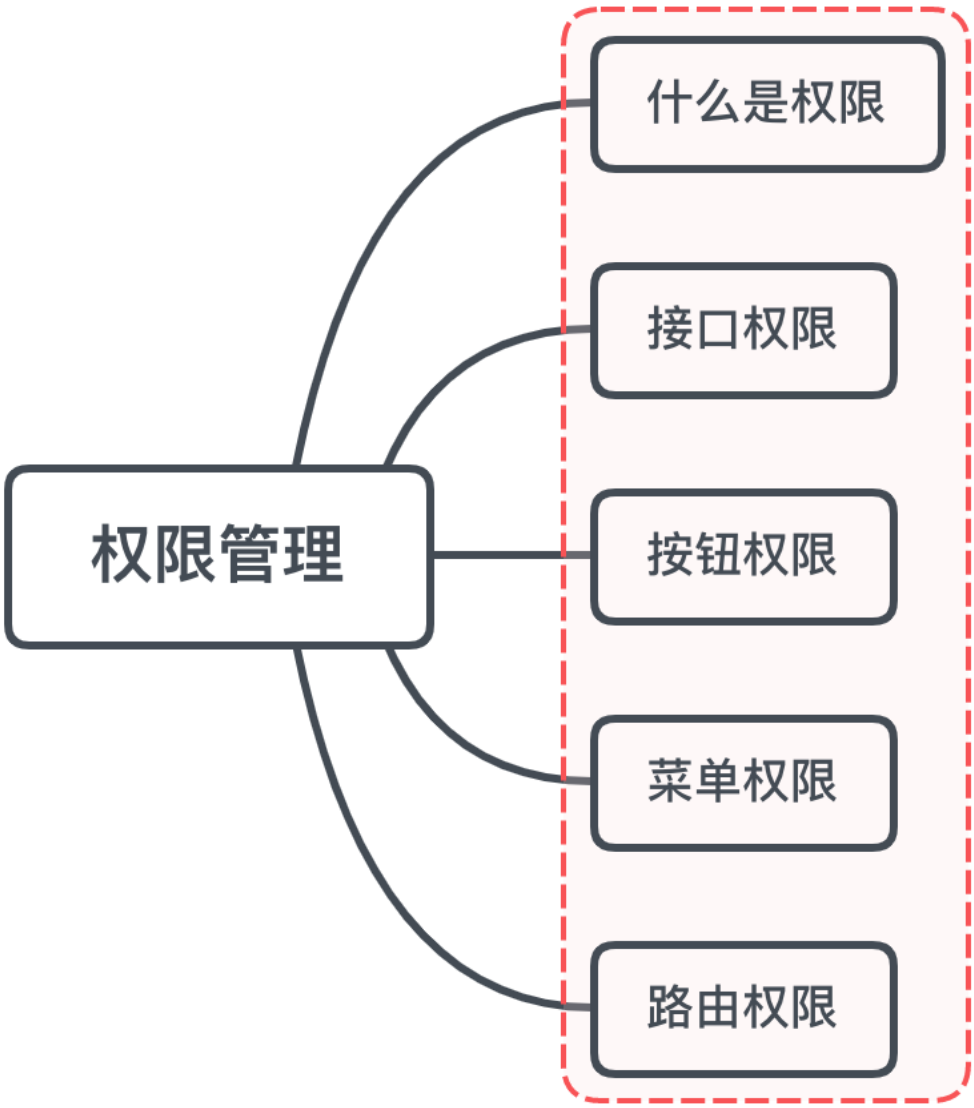
小结

`createElement`创建`VNode`的过程，每个`VNode`有`children`，`children`每个元素也是一个`VNode`，这样就形成了一个虚拟树结构，用于描述真实的`DOM`树结构

参考文献

- https://ustbhuangyi.github.io/vue-analysis/v2/data-driven/create-element.html#children-%E7%9A%84%E8%A7%84%E8%8C%83%E5%8C%96
- https://juejin.cn/post/6876711874050818061

15.vue要做权限管理该怎么做？如果控制到按钮级别的权限怎么做？



一、是什么

权限是对特定资源的访问许可，所谓权限控制，也就是确保用户只能访问到被分配的资源

而前端权限归根结底是请求的发起权，请求的发起可能有下面两种形式触发

- 页面加载触发
- 页面上的按钮点击触发

总的来说，所有的请求发起都触发自前端路由或视图

所以我们可以从这两方面入手，对触发权限的源头进行控制，最终要实现的目标是：

- 路由方面，用户登录后只能看到自己有权访问的导航菜单，也只能访问自己有权访问的路由地址，否则将跳转 `4xx` 提示页
- 视图方面，用户只能看到自己有权浏览的内容和有权操作的控件
- 最后再加上请求控制作为最后一道防线，路由可能配置失误，按钮可能忘了加权限，这种时候请求控制可以用来兜底，越权请求将在前端被拦截

二、如何做

前端权限控制可以分为四个方面：

- 接口权限

- 按钮权限
- 菜单权限
- 路由权限

接口权限

接口权限目前一般采用`jwt`的形式来验证，没有通过的话一般返回`401`，跳转到登录页面重新进行登录

登录完拿到`token`，将`token`存起来，通过`axios`请求拦截器进行拦截，每次请求的时候头部携带`token`

```
axios.interceptors.request.use(config => {
    config.headers['token'] = cookie.get('token')
    return config
})
axios.interceptors.response.use(res=>{},{response}=>{
    if (response.data.code === 40099 || response.data.code === 40098) { //token过期或者错误
        router.push('/login')
    }
})
```

路由权限控制

方案一

初始化即挂载全部路由，并且在路由上标记相应的权限信息，每次路由跳转前做校验

```
const routerMap = [
  {
    path: '/permission',
    component: Layout,
    redirect: '/permission/index',
    alwaysShow: true, // will always show the root menu
    meta: {
      title: 'permission',
      icon: 'lock',
      roles: ['admin', 'editor'] // you can set roles in root nav
    },
    children: [{
      path: 'page',
      component: () => import('@/views/permission/page'),
      name: 'pagePermission',
      meta: {
        title: 'pagePermission',
        roles: ['admin'] // or you can only set roles in sub nav
      }
    }, {
      path: 'directive',
      component: () => import('@/views/permission/directive'),
      name: 'directivePermission',
      meta: {
        title: 'directivePermission'
```

```
        // if do not set roles, means: this page does not require permission
      }
    }]
  }]
```

这种方式存在以下四种缺点：

- 加载所有的路由，如果路由很多，而用户并不是所有的路由都有权限访问，对性能会有影响。
- 全局路由守卫里，每次路由跳转都要做权限判断。
- 菜单信息写死在前端，要改个显示文字或权限信息，需要重新编译
- 菜单跟路由耦合在一起，定义路由的时候还有添加菜单显示标题，图标之类的信息，而且路由不一定作为菜单显示，还要多加字段进行标识

方案二

初始化的时候先挂载不需要权限控制的路由，比如登录页，404等错误页。如果用户通过URL进行强制访问，则会直接进入404，相当于从源头上做了控制

登录后，获取用户的权限信息，然后筛选有权限访问的路由，在全局路由守卫里进行调用`addRoutes`添加路由

```
import router from './router'
import store from './store'
import { Message } from 'element-ui'
import NProgress from 'nprogress' // progress bar
import 'nprogress/nprogress.css'// progress bar style
import { getToken } from '@/utils/auth' // getToken from cookie

NProgress.configure({ showSpinner: false })// NProgress Configuration

// permission judge function
function hasPermission(roles, permissionRoles) {
  if (roles.indexOf('admin') >= 0) return true // admin permission passed
directly
  if (!permissionRoles) return true
  return roles.some(role => permissionRoles.indexOf(role) >= 0)
}

const whiteList = ['/login', '/authredirect']// no redirect whitelist

router.beforeEach((to, from, next) => {
  NProgress.start() // start progress bar
  if (getToken()) { // determine if there has token
    /* has token*/
    if (to.path === '/login') {
      next({ path: '/' })
      NProgress.done() // if current page is dashboard will not trigger
afterEach hook, so manually handle it
    } else {
      if (store.getters.roles.length === 0) { // 判断当前用户是否已拉取完user_info信
息
        store.dispatch('GetUserInfo').then(res => { // 拉取user_info
          const roles = res.data.roles // note: roles must be a array! such as:
['editor','develop']
```

```
          store.dispatch('GenerateRoutes', { roles }).then(() => { // 根据roles权限生成可访问的路由表
            router.addRoutes(store.getters.addRouters) // 动态添加可访问路由表
            next({ ...to, replace: true }) // hack方法 确保addRoutes已完成 ,set the replace: true so the navigation will not leave a history record
          })
        }).catch((err) => {
          store.dispatch('FedLogOut').then(() => {
            Message.error(err || 'Verification failed, please login again')
            next({ path: '/' })
          })
        })
      } else {
        // 没有动态改变权限的需求可直接next() 删除下方权限判断 ↓
        if (hasPermission(store.getters.roles, to.meta.roles)) {
          next()//
        } else {
          next({ path: '/401', replace: true, query: { noGoBack: true }})
        }
        // 可删 ↑
      }
    }
  } else {
    /* has no token*/
    if (whiteList.indexOf(to.path) !== -1) { // 在免登录白名单，直接进入
      next()
    } else {
      next('/login') // 否则全部重定向到登录页
      NProgress.done() // if current page is login will not trigger afterEach hook, so manually handle it
    }
  }
})

router.afterEach(() => {
  NProgress.done() // finish progress bar
})
```

按需挂载，路由就需要知道用户的路由权限，也就是在用户登录进来的时候就要知道当前用户拥有哪些路由权限

这种方式也存在了以下的缺点：

- 全局路由守卫里，每次路由跳转都要做判断
- 菜单信息写死在前端，要改个显示文字或权限信息，需要重新编译
- 菜单跟路由耦合在一起，定义路由的时候还有添加菜单显示标题，图标之类的信息，而且路由不一定作为菜单显示，还要多加字段进行标识

菜单权限

菜单权限可以理解成将页面与理由进行解耦

方案一

菜单与路由分离，菜单由后端返回

前端定义路由信息

```
{
    name: "login",
    path: "/login",
    component: () => import("@/pages/Login.vue")
}
```

`name`字段都不为空，需要根据此字段与后端返回菜单做关联，后端返回的菜单信息中必须要有`name`对应的字段，并且做唯一性校验

全局路由守卫里做判断

```
function hasPermission(router, accessMenu) {
  if (whiteList.indexOf(router.path) !== -1) {
    return true;
  }
  let menu = Util.getMenuByName(router.name, accessMenu);
  if (menu.name) {
    return true;
  }
  return false;

}

Router.beforeEach(async (to, from, next) => {
  if (getToken()) {
    let userInfo = store.state.user.userInfo;
    if (!userInfo.name) {
      try {
        await store.dispatch("GetUserInfo")
        await store.dispatch('updateAccessMenu')
        if (to.path === '/login') {
          next({ name: 'home_index' })
        } else {
          //Util.toDefaultPage([...routers], to.name, router, next);
          next({ ...to, replace: true })//菜单权限更新完成,重新进一次当前路由
        }
      }
      catch (e) {
        if (whiteList.indexOf(to.path) !== -1) { // 在免登录白名单，直接进入
          next()
        } else {
          next('/login')
        }
      }
    } else {
      if (to.path === '/login') {
        next({ name: 'home_index' })
      } else {
        if (hasPermission(to, store.getters.accessMenu)) {
          Util.toDefaultPage(store.getters.accessMenu,to, routes, next);
        } else {
          next({ path: '/403',replace:true })
        }
      }
    }
```

```
  } else {
    if (whiteList.indexOf(to.path) !== -1) { // 在免登录白名单，直接进入
      next()
    } else {
      next('/login')
    }
  }
  let menu = Util.getMenuByName(to.name, store.getters.accessMenu);
  Util.title(menu.title);
});

Router.afterEach((to) => {
  window.scrollTo(0, 0);
});
```

每次路由跳转的时候都要判断权限，这里的判断也很简单，因为菜单的`name`与路由的`name`是一一对应的，而后端返回的菜单就已经是经过权限过滤的

如果根据路由`name`找不到对应的菜单，就表示用户有没权限访问

如果路由很多，可以在应用初始化的时候，只挂载不需要权限控制的路由。取得后端返回的菜单后，根据菜单与路由的对应关系，筛选出可访问的路由，通过`addRoutes`动态挂载

这种方式的缺点：

- 菜单需要与路由做一一对应，前端添加了新功能，需要通过菜单管理功能添加新的菜单，如果菜单配置的不对会导致应用不能正常使用
- 全局路由守卫里，每次路由跳转都要做判断

方案二

菜单和路由都由后端返回

前端统一定义路由组件

```
const Home = () => import("../pages/Home.vue");
const UserInfo = () => import("../pages/UserInfo.vue");
export default {
    home: Home,
    userInfo: UserInfo
};
```

后端路由组件返回以下格式

```
[
    {
        name: "home",
        path: "/",
        component: "home"
    },
    {
        name: "home",
        path: "/userinfo",
        component: "userInfo"
    }
]
```

在将后端返回路由通过`addRoutes`动态挂载之间，需要将数据处理一下，将`component`字段换为真正的组件

如果有嵌套路由，后端功能设计的时候，要注意添加相应的字段，前端拿到数据也要做相应的处理

这种方法也会存在缺点：

- 全局路由守卫里，每次路由跳转都要做判断
- 前后端的配合要求更高

按钮权限

方案一

按钮权限也可以用`v-if`判断

但是如果页面过多，每个页面页面都要获取用户权限`role`和路由表里的`meta.btnPermissions`，然后再做判断

这种方式就不展开举例了

方案二

通过自定义指令进行按钮权限的判断

首先配置路由

```
{
    path: '/permission',
    component: Layout,
    name: '权限测试',
    meta: {
        btnPermissions: ['admin', 'supper', 'normal']
    },
    //页面需要的权限
    children: [{
        path: 'supper',
        component: _import('system/supper'),
        name: '权限测试页',
        meta: {
            btnPermissions: ['admin', 'supper']
        } //页面需要的权限
    },
    {
        path: 'normal',
        component: _import('system/normal'),
        name: '权限测试页',
        meta: {
            btnPermissions: ['admin']
        } //页面需要的权限
    }]
}
```

自定义权限鉴定指令

```
import Vue from 'vue'
/**权限指令**/
const has = Vue.directive('has', {
    bind: function (el, binding, vnode) {
        // 获取页面按钮权限
        let btnPermissionsArr = [];
        if(binding.value){
            // 如果指令传值，获取指令参数，根据指令参数和当前登录人按钮权限做比较。
            btnPermissionsArr = Array.of(binding.value);
        }else{
            // 否则获取路由中的参数，根据路由的btnPermissionsArr和当前登录人按钮权限做比较。
            btnPermissionsArr = vnode.context.$route.meta.btnPermissions;
        }
        if (!Vue.prototype.$_has(btnPermissionsArr)) {
            el.parentNode.removeChild(el);
        }
    }
});
// 权限检查方法
Vue.prototype.$_has = function (value) {
    let isExist = false;
    // 获取用户按钮权限
    let btnPermissionsStr = sessionStorage.getItem("btnPermissions");
    if (btnPermissionsStr == undefined || btnPermissionsStr == null) {
        return false;
    }
    if (value.indexOf(btnPermissionsStr) > -1) {
        isExist = true;
    }
    return isExist;
};
export {has}
```

在使用的按钮中只需要引用`v-has`指令

```
<el-button @click='editClick' type="primary" v-has>编辑</el-button>
```

小结

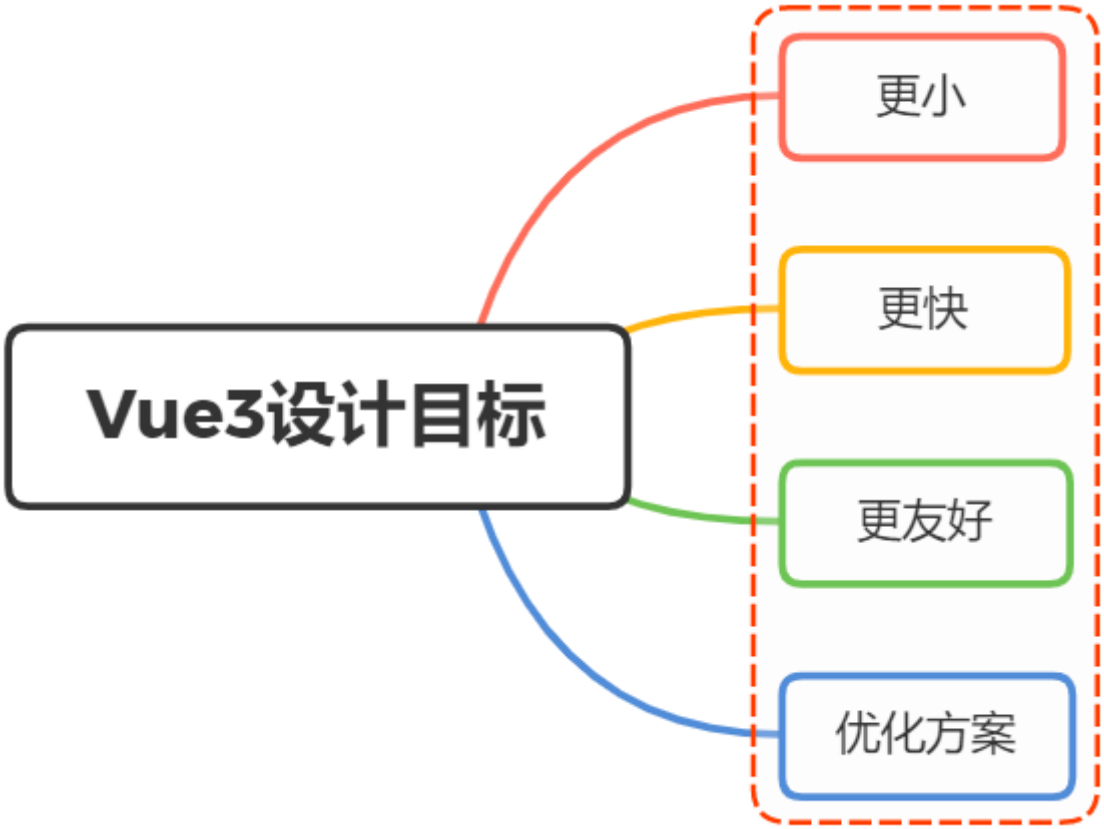
关于权限如何选择哪种合适的方案，可以根据自己项目的方案项目，如考虑路由与菜单是否分离

权限需要前后端结合，前端尽可能的去控制，更多的需要后台判断

参考文献

- https://mp.weixin.qq.com/s/b-D2eH1mLwL_FkaZwjueSw
- https://segmentfault.com/a/1190000020887109
- https://juejin.cn/post/6844903648057622536#heading-6

16.Vue3.0的设计目标是什么？做了哪些优化



一、设计目标

不以解决实际业务痛点的更新都是耍流氓，下面我们来列举一下`Vue3`之前我们或许会面临的问题

- 随着功能的增长，复杂组件的代码变得越来越难以维护
- 缺少一种比较「干净」的在多个组件之间提取和复用逻辑的机制
- 类型推断不够友好
- `bundle`的时间太久了

而`Vue3`经过长达两三年时间的筹备，做了哪些事情?

我们从结果反推

- 更小
- 更快
- TypeScript支持
- API设计一致性
- 提高自身可维护性
- 开放更多底层功能

一句话概述，就是更小更快更友好了

更小

`Vue3`移除一些不常用的`API`

引入`tree-shaking`，可以将无用模块"剪辑"，仅打包需要的，使打包的整体体积变小了

更快

主要体现在编译方面：

- diff算法优化
- 静态提升
- 事件监听缓存
- SSR优化

下篇文章我们会进一步介绍

更友好

`vue3`在兼顾`vue2`的`options API`的同时还推出了`composition API`，大大增加了代码的逻辑组织和代码复用能力

这里代码简单演示下：

存在一个获取鼠标位置的函数

```
import { toRefs, reactive } from 'vue';
function useMouse(){
    const state = reactive({x:0,y:0});
    const update = e=>{
        state.x = e.pageX;
        state.y = e.pageY;
    }
    onMounted(()=>{
        window.addEventListener('mousemove',update);
    })
    onUnmounted(()=>{
        window.removeEventListener('mousemove',update);
    })

    return toRefs(state);
}
```

我们只需要调用这个函数，即可获取`x`、`y`的坐标，完全不用关注实现过程

试想一下，如果很多类似的第三方库，我们只需要调用即可，不必关注实现过程，开发效率大大提高

同时，`VUE3`是基于`typescipt`编写的，可以享受到自动的类型定义提示

三、优化方案

`vue3`从很多层面都做了优化，可以分成三个方面：

- 源码
- 性能
- 语法 API

源码

源码可以从两个层面展开：

- 源码管理
- TypeScript

源码管理

`vue3`整个源码是通过`monorepo`的方式维护的，根据功能将不同的模块拆分到`packages`目录下面不同的子目录中

这样使得模块拆分更细化，职责划分更明确，模块之间的依赖关系也更加明确，开发人员也更容易阅读、理解和更改所有模块源码，提高代码的可维护性

另外一些`package`（比如`reactivity`响应式库）是可以独立于`Vue`使用的，这样用户如果只想使用`Vue3`的响应式能力，可以单独依赖这个响应式库而不用去依赖整个`Vue`

TypeScript

`Vue3`是基于`typeScript`编写的，提供了更好的类型检查，能支持复杂的类型推导

性能

`vue3`是从什么哪些方面对性能进行进一步优化呢?

- 体积优化
- 编译优化
- 数据劫持优化

这里讲述数据劫持:

在`vue2`中，数据劫持是通过`Object.defineProperty`，这个 API 有一些缺陷，并不能检测对象属性的添加和删除

```
Object.defineProperty(data, 'a',{
  get(){
    // track
  },
  set(){
    // trigger
  }
})
```

尽管`Vue`为了解决这个问题提供了`set`和`delete`实例方法，但是对于用户来说，还是增加了一定的心智负担

同时在面对嵌套层级比较深的情况下，就存在性能问题

```
default {
  data: {
    a: {
      b: {
          c: {
          d: 1
        }
      }
    }
  }
}
```

相比之下，`vue3`是通过`proxy`监听整个对象，那么对于删除还是监听当然也能监听到

同时`Proxy`并不能监听到内部深层次的对象变化，而`Vue3`的处理方式是在`getter`中去递归响应式，这样的好处是真正访问到的内部对象才会变成响应式，而不是无脑递归

语法 API

这里当然说的就是`composition API`，其两大显著的优化:

- 优化逻辑组织
- 优化逻辑复用

逻辑组织

一张图，我们可以很直观地感受到`Composition API`在逻辑组织方面的优势

相同功能的代码编写在一块，而不像`options API`那样，各个功能的代码混成一块

逻辑复用

在`vue2`中，我们是通过`mixin`实现功能混合，如果多个`mixin`混合，会存在两个非常明显的问题：命名冲突和数据来源不清晰

而通过`composition`这种形式，可以将一些复用的代码抽离出来作为一个函数，只要的使用的地方直接进行调用即可

同样是上文的获取鼠标位置的例子

```
import { toRefs, reactive, onUnmounted, onMounted } from 'vue';
function useMouse(){
    const state = reactive({x:0,y:0});
    const update = e=>{
        state.x = e.pageX;
        state.y = e.pageY;
    }
```

```
    onMounted(()=>{
        window.addEventListener('mousemove',update);
    })
    onUnmounted(()=>{
        window.removeEventListener('mousemove',update);
    })

    return toRefs(state);
}
```

组件使用

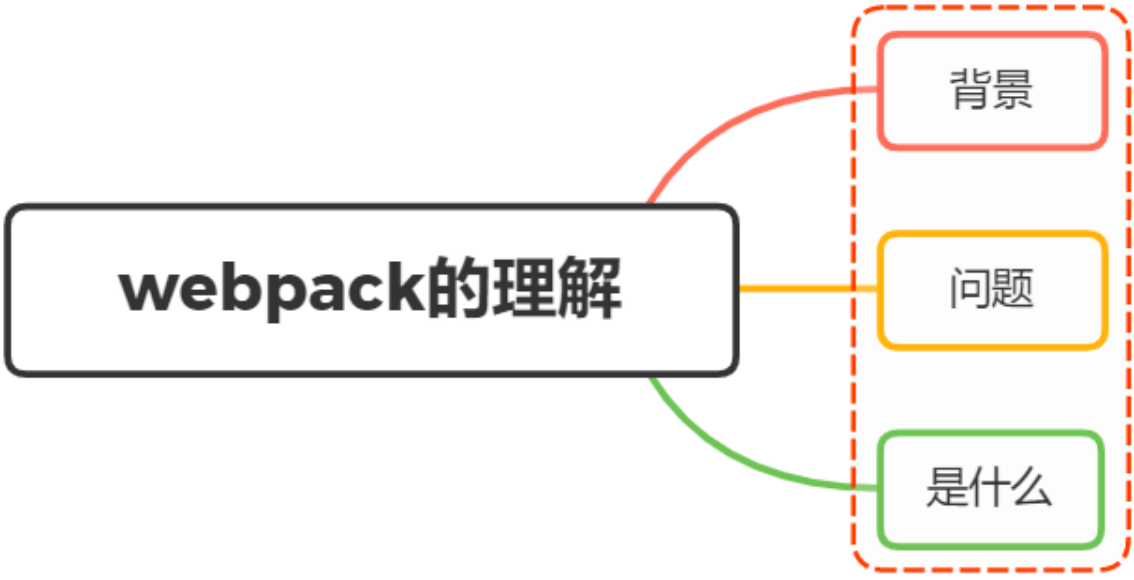
```
import useMousePosition from './mouse'
export default {
    setup() {
        const { x, y } = useMousePosition()
        return { x, y }
    }
}
```

可以看到，整个数据来源清晰了，即使去编写更多的 hook 函数，也不会出现命名冲突的问题

参考文献

- https://juejin.cn/post/6850418112878575629#heading-5
- https://vue3js.cn/docs/zh

17.说说你对webpack的理解？解决了什么问题？



一、背景

`Webpack` 最初的目标是实现前端项目的模块化，旨在更高效地管理和维护项目中的每一个资源

模块化

最早的时候，我们会通过文件划分的形式实现模块化，也就是将每个功能及其相关状态数据各自单独放到不同的`JS`文件中

约定每个文件是一个独立的模块，然后再将这些`js`文件引入到页面，一个`script`标签对应一个模块，然后调用模块化的成员

```
<script src="module-a.js"></script>
<script src="module-b.js"></script>
```

但这种模块弊端十分的明显，模块都是在全局中工作，大量模块成员污染了环境，模块与模块之间并没有依赖关系、维护困难、没有私有空间等问题

项目一旦变大，上述问题会尤其明显

随后，就出现了命名空间方式，规定每个模块只暴露一个全局对象，然后模块的内容都挂载到这个对象中

```
window.moduleA = {
  method1: function () {
    console.log('moduleA#method1')
  }
}
```

这种方式也并没有解决第一种方式的依赖等问题

再后来，我们使用立即执行函数为模块提供私有空间，通过参数的形式作为依赖声明，如下

```
// module-a.js
(function ($) {
  var name = 'module-a'

  function method1 () {
    console.log(name + '#method1')
    $('body').animate({ margin: '200px' })
  }

  window.moduleA = {
    method1: method1
  }
})(jQuery)
```

上述的方式都是早期解决模块的方式，但是仍然存在一些没有解决的问题。例如，我们是用过`script`标签在页面引入这些模块的，这些模块的加载并不受代码的控制，时间一久维护起来也十分的麻烦

理想的解决方式是，在页面中引入一个`JS`入口文件，其余用到的模块可以通过代码控制，按需加载进来

除了模块加载的问题以外，还需要规定模块化的规范，如今流行的则是`CommonJS`、`ES Modules`

二、问题

从后端渲染的`JSP`、`PHP`，到前端原生`JavaScript`，再到`jQuery`开发，再到目前的三大框架`Vue`、`React`、`Angular`

开发方式，也从`javascript`到后面的`es5`、`es6、7、8、9、10`，再到`typescript`，包括编写`CSS`的预处理器`less`、`scss`等

现代前端开发已经变得十分的复杂，所以我们开发过程中会遇到如下的问题：

- 需要通过模块化的方式来开发
- 使用一些高级的特性来加快我们的开发效率或者安全性，比如通过ES6+、TypeScript开发脚本逻辑，通过sass、less等方式来编写css样式代码
- 监听文件的变化来并且反映到浏览器上，提高开发的效率
- JavaScript 代码需要模块化，HTML 和 CSS 这些资源文件也会面临需要被模块化的问题
- 开发完成后我们还需要将代码进行压缩、合并以及其他相关的优化

而`webpack`恰巧可以解决以上问题

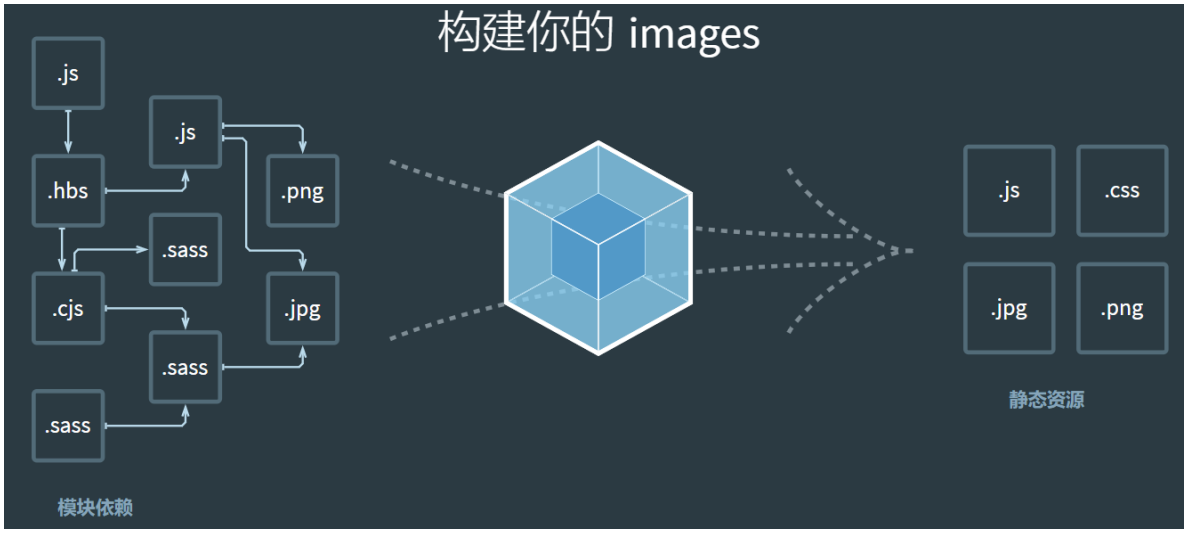
三、是什么

`webpack`是一个用于现代`JavaScript`应用程序的静态模块打包工具

- 静态模块

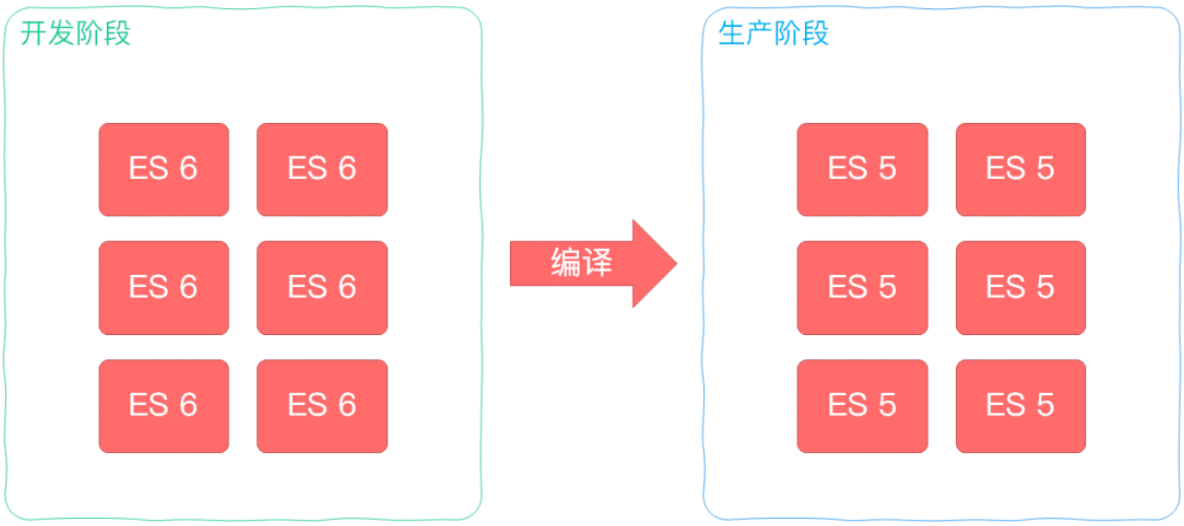
这里的静态模块指的是开发阶段，可以被`webpack`直接引用的资源（可以直接被获取打包进`bundle.js`的资源）

当`webpack`处理应用程序时，它会在内部构建一个依赖图，此依赖图对应映射到项目所需的每个模块（不再局限`js`文件），并生成一个或多个`bundle`

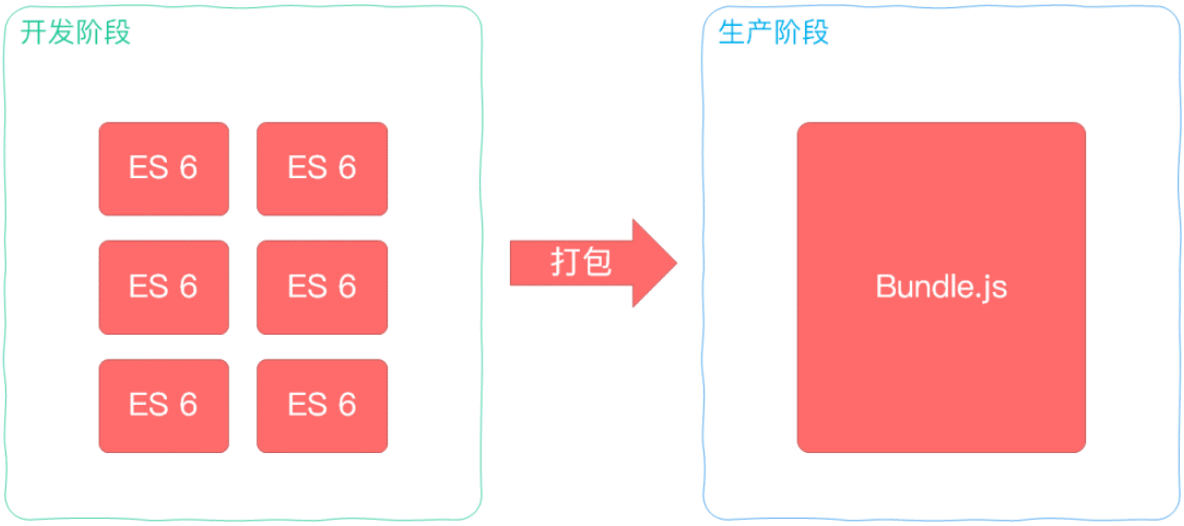


`webpack`的能力：

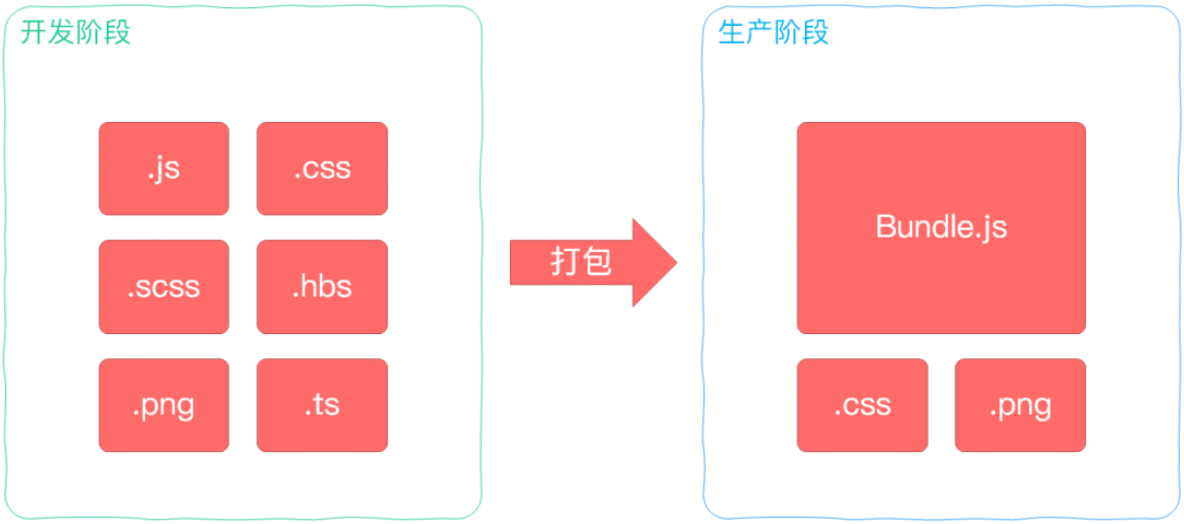
编译代码能力，提高效率，解决浏览器兼容问题



模块整合能力，提高性能，可维护性，解决浏览器频繁请求文件的问题



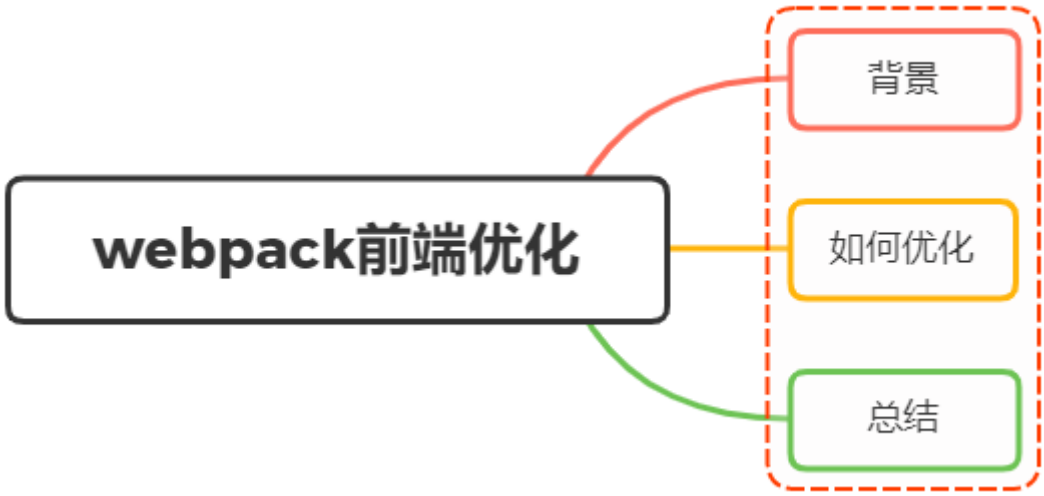
万物皆可模块能力，项目维护性增强，支持不同种类的前端模块类型，统一的模块化方案，所有资源文件的加载都可以通过代码控制



参考文献

- https://webpack.docschina.org/concepts/
- https://zhuanlan.zhihu.com/p/267875652

18.说说如何借助webpack来优化前端性能



一、背景

随着前端的项目逐渐扩大，必然会带来的一个问题就是性能

尤其在大型复杂的项目中，前端业务可能因为一个小小的数据依赖，导致整个页面卡顿甚至奔溃

一般项目在完成后，会通过`webpack`进行打包，利用`webpack`对前端项目性能优化是一个十分重要的环节

二、如何优化

通过`webpack`优化前端的手段有：

- JS代码压缩
- CSS代码压缩
- Html文件代码压缩
- 文件大小压缩
- 图片压缩
- Tree Shaking
- 代码分离
- 内联 chunk

JS代码压缩

`terser`是一个`JavaScript`的解释、绞肉机、压缩机的工具集，可以帮助我们压缩、丑化我们的代码，让`bundle`更小

在`production`模式下，`webpack`默认就是使用`TerserPlugin`来处理我们的代码的。如果想要自定义配置它，配置方法如下：

```
const TerserPlugin = require('terser-webpack-plugin')
module.exports = {
    ...
    optimization: {
        minimize: true,
        minimizer: [
            new TerserPlugin({
                parallel: true // 电脑cpu核数-1
            })
        ]
    }
}
```

属性介绍如下：

- extractComments：默认值为true，表示会将注释抽取到一个单独的文件中，开发阶段，我们可设置为 false ，不保留注释
- parallel：使用多进程并发运行提高构建的速度，默认值是true，并发运行的默认数量：os.cpus().length - 1
- terserOptions：设置我们的terser相关的配置：
    - compress：设置压缩相关的选项，mangle：设置丑化相关的选项，可以直接设置为true
    - mangle：设置丑化相关的选项，可以直接设置为true
    - toplevel：底层变量是否进行转换
    - keep_classnames：保留类的名称
    - keep_fnames：保留函数的名称

CSS代码压缩

`CSS`压缩通常是去除无用的空格等，因为很难去修改选择器、属性的名称、值等

CSS的压缩我们可以使用另外一个插件：`css-minimizer-webpack-plugin`

```
npm install css-minimizer-webpack-plugin -D
```

配置方法如下：

```
const CssMinimizerPlugin = require('css-minimizer-webpack-plugin')
module.exports = {
    // ...
    optimization: {
        minimize: true,
        minimizer: [
            new CssMinimizerPlugin({
                parallel: true
            })
        ]
    }
}
```

Html文件代码压缩

使用`HtmlWebpackPlugin`插件来生成`HTML`的模板时候，通过配置属性`minify`进行`html`优化

```
module.exports = {
    ...
    plugin:[
        new HtmlwebpackPlugin({
            ...
            minify:{
                minifyCSS:false, // 是否压缩css
                collapseWhitespace:false, // 是否折叠空格
                removeComments:true // 是否移除注释
            }
        })
    ]
}
```

设置了`minify`，实际会使用另一个插件`html-minifier-terser`

文件大小压缩

对文件的大小进行压缩，减少`http`传输过程中宽带的损耗

```
npm install compression-webpack-plugin -D
```

```
new ComepressionPlugin({
    test:/\.(css|js)$/,  // 哪些文件需要压缩
    threshold:500, // 设置文件多大开始压缩
    minRatio:0.7, // 至少压缩的比例
    algorithm:"gzip", // 采用的压缩算法
})
```

图片压缩

一般来说在打包之后，一些图片文件的大小是远远要比`js`或者`css`文件要来的大，所以图片压缩较为重要

配置方法如下：

```
module: {
  rules: [
    {
      test: /\.(png|jpg|gif)$/,
      use: [
        {
          loader: 'file-loader',
          options: {
            name: '[name]_[hash].[ext]',
            outputPath: 'images/',
```

```
          }
        },
        {
          loader: 'image-webpack-loader',
          options: {
            // 压缩 jpeg 的配置
            mozjpeg: {
              progressive: true,
              quality: 65
            },
            // 使用 imagemin**-optipng 压缩 png，enable: false 为关闭
            optipng: {
              enabled: false,
            },
            // 使用 imagemin-pngquant 压缩 png
            pngquant: {
              quality: '65-90',
              speed: 4
            },
            // 压缩 gif 的配置
            gifsicle: {
              interlaced: false,
            },
            // 开启 webp，会把 jpg 和 png 图片压缩为 webp 格式
            webp: {
              quality: 75
            }
          }
        }
      ]
    },
  ]
}
```

Tree Shaking

`Tree Shaking` 是一个术语，在计算机中表示消除死代码，依赖于`ES Module`的静态语法分析（不执行任何的代码，可以明确知道模块的依赖关系）

在`webpack`实现`Trss shaking`有两种不同的方案：

- usedExports：通过标记某些函数是否被使用，之后通过Terser来进行优化的
- sideEffects：跳过整个模块/文件，直接查看该文件是否有副作用

两种不同的配置方案， 有不同的效果

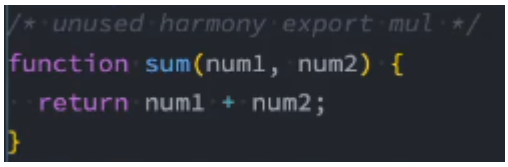
usedExports

配置方法也很简单，只需要将`usedExports`设为`true`

```
module.exports = {
    ...
    optimization:{
        usedExports
    }
}
```

使用之后，没被用上的代码在`webpack`打包中会加入`unused harmony export mul`注释，用来告知`Terser`在优化时，可以删除掉这段代码

如下面`sum`函数没被用到，`webpack`打包会添加注释，`terser`在优化时，则将该函数去掉



sideEffects

`sideEffects`用于告知`webpack compiler`哪些模块时有副作用，配置方法是在`package.json`中设置`sideEffects`属性

如果`sideEffects`设置为false，就是告知`webpack`可以安全的删除未用到的`exports`

如果有些文件需要保留，可以设置为数组的形式

```
"sideEffecis":[
    "./src/util/format.js",
    "*.css" // 所有的css文件
]
```

上述都是关于`javascript`的`tree shaking`，`css`同样也能够实现`tree shaking`

css tree shaking

`css`进行`tree shaking`优化可以安装`PurgeCss`插件

```
npm install purgecss-plugin-webpack -D
```

```
const PurgeCssPlugin = require('purgecss-webpack-plugin')
module.exports = {
    ...
    plugins:[
        new PurgeCssPlugin({
            path:glob.sync(`${path.resolve('./src')}/**/*`), {nodir:true}// src
里面的所有文件
            satelist:function(){
                return {
                    standard:["html"]
                }
            }
```

```
        })
    ]
}
```

- paths：表示要检测哪些目录下的内容需要被分析，配合使用glob
- 默认情况下，Purgecss会将我们的html标签的样式移除掉，如果我们希望保留，可以添加一个safelist的属性

代码分离

将代码分离到不同的`bundle`中，之后我们可以按需加载，或者并行加载这些文件

默认情况下，所有的`JavaScript`代码（业务代码、第三方依赖、暂时没有用到的模块）在首页全部都加载，就会影响首页的加载速度

代码分离可以分出出更小的`bundle`，以及控制资源加载优先级，提供代码的加载性能

这里通过`splitChunksPlugin`来实现，该插件`webpack`已经默认安装和集成，只需要配置即可

默认配置中，chunks仅仅针对于异步（async）请求，我们可以设置为initial或者all

```
module.exports = {
    ...
    optimization:{
        splitChunks:{
            chunks:"all"
        }
    }
}
```

`splitChunks`主要属性有如下：

- Chunks，对同步代码还是异步代码进行处理
- minSize： 拆分包的大小, 至少为minSize，如何包的大小不超过minSize，这个包不会拆分
- maxSize： 将大于maxSize的包，拆分为不小于minSize的包
- minChunks：被引入的次数，默认是1

内联chunk

可以通过`InlineChunkHtmlPlugin`插件将一些`chunk`的模块内联到`html`，如`runtime`的代码（对模块进行解析、加载、模块信息相关的代码），代码量并不大，但是必须加载的

```
const InlineChunkHtmlPlugin = require('react-dev-utils/InlineChunkHtmlPlugin')
const HtmlWebpackPlugin = require('html-webpack-plugin')
module.exports = {
    ...
    plugin:[
        new InlineChunkHtmlPlugin(HtmlWebpackPlugin,[/runtime.+\.js/]
}
```

三、总结

关于`webpack`对前端性能的优化，可以通过文件体积大小入手，其次还可通过分包的形式、减少http请求次数等方式，实现对前端性能的优化

参考文献

- https://zhuanlan.zhihu.com/p/139498741
- https://vue3js.cn/interview/