[TOC]

CSS

%百分比的使用

- `width/height` 基于父元素。对于一些需要占满的元素，例如 nav、footer，直接设置 `width:100%`，当父元素也没有设置具体高度时，此时子元素就算设置了百分比高度仍会变成 auto。故在使用百分比时，同一个选择器控制的元素样式可能会不同，因为他们的父元素宽高不同。对于父元素也设置百分比的元素，依然是按照父元素的百分比计算，即和爷爷也有百分比关系

```
.grandpa {
    width:100px;
}
.father {
    with:50%; //100 * 0.5 = 50px
}
.son{
    width:50%; //50px * 0.5 = 25px
}
```

- `left/right/top/bottom` 基于父元素。对于一些左右布局，可以使用 `position: left:50%` 加 `margin-left` 偏移

- `transform: translateX(-50%)` transform 表示移动，translate(x，y) 表示 X，Y 轴移动距离，translateX，translateY 分别表示 X，Y 轴移动距离，百分比基于自身宽高

```
  // 实现水平垂直居中
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
```

- `margin/padding left/right/top/right` 都是基于父元素宽度，和**父元素高度**没有关系

```
.father {
  width: 200px;
  height: 100px;
  border: 1px solid #000;
}
.son {
  width: 80%;
  height: 80%;
  margin: 10% 10%;//20px 20px 20px 20px
```

```
  background-color: #ff0000;
}
```

- `border-radius` 基于自身宽高，设置 `border-radius:50%` 可以画出圆
- `background-position:@prams @prams` 表示背景图片基于容器的水平垂直位置，不设置时，图片位于元素左上角，参数可使用 center/left 等，也可以使用百分比，百分比表示：（容器的宽高—背景图片的宽高）*百分比，故 50%时，元素居中

```
// 参数二默认为 center，二者等价，都表示水平垂直居中
background-position:50%;
background-position:center;
```

- `line-height` 基于自身`font-size`大小
- `vertical-align` 对行内元素 inline 和 table-cell 起效（img、span、input、td、button、strong、em），行内元素垂直方向对齐方式：基线对齐。各行内元素的基线皆不同，其中 x 字母、图片、设置 overflow:hidden 的行内元素基线在最底部。 vertical-align 决定当前元素在基线基础上，相对父元素字体垂直移动距离，正值上移，负值下移。还有一些可选值，top 表示与元素最高点对齐，text-top 表示与父元素字体的顶端对齐

```
vertical-align: 50%;// 相对于基线上移 15px
line-height: 30px;
```

- ::after 百分比根据元素本身

图片元素的间隙

往 div 中放入图片，图片底部距离 div 下方会有一个间隙，两个图片水平之间也会有间隙

- 垂直方向 行内元素基线对齐后还要和**父元素的字体基线**保持一致，故图片基线由基于元素底部变成基于 x 字母，去掉空隙需操作父元素的字体，或者把图片设置成块级元素，或者设`vertical-align`属性

```
float: left; /* 设置图片浮动属性，默认变成块级元素 *
vertical-align: top;/* 设置图片垂直对齐方式 */
font-size: 0; /* 设置父元素文本大小为 0 */
display: block; /* 设置图片为块级元素 */
line-height: 0; /* 设置父元素行高为 0 */
```

- 水平方向 空格和换行字符被浏览器解析为空格，可以将图片元素不留空格连写、可以设置块级、浮动

鼠标事件

- `cursor` 鼠标指针放入元素范围内时显示的形状，常见：`cursor:pointer`

- pointer-events 设置元素是否鼠标可点击，被设置 pointer-events: none 的元素不仅无法被点击，而且没有鼠标样式，且其子元素也无法被点击，若子元素需被点击，独自设置 pointer-events: initial ，即属性初始化，默认可以被点击

文字设置

- white-space 文字空格。多个空格默认为一个，使用 white-space 属性操作空格，常用取值为：white-space:pre 和 ``white-space:pre-wrap`，其中 pre 不允许自动换行，原原本本地显示文本，可能超出容器，而 pre-wrap 会根据容器宽度自动换行。二者都保留行尾空格，但 pre-wrap 行尾空格也不自动换行，即超过容器，不影响下一行

```
.pre-article {
  font-family: inherit;
  word-break: break-all;//任意字符换行
  white-space: pre-wrap;//处理空格，是否换行
  }
```

- word-break 文字断行，对于英文单词不进行断行，break-all 表示全部字母断行
- 超出 n 行后显示省略
  - 一行：设置文字不换行、超出隐藏、超出使用省略号展示

```
white-space:nowrap;
overflow:hidden;
text-overflow:ellipsis;
```

  - n 行：超出隐藏、超出使用省略号展示

```
display:-webkit-box;
-webkit-box-orient:vertical;
-webkit-line-clamp:n;
overflow:hidden;
text-overflow:ellipsis;
```

- 文本左右两端对齐 text-align 表示文本对齐方式，一般设置 left、right、center。此外还有 justify，表示文本两端对齐，但当文本只有一行时，不起作用。此外，当设置 text-algin:justify 时，可以使用 text-justify 属性， 此属性只在 IE 浏览器起效。

```
p {
    text-align:justify; // 水平文字对齐方式为两端对齐
    text-justify:distribute // 兼容 IE 浏览器
}

// 使用 text-algin  表单的 span 两端对齐
```

```
span {
    display:inline-block; // 行内盒子，可设置宽高
    vertical-align:middle; // 和表单的基线对齐
    text-align:justify;
    text-align-last:justify; // 表示最后一行两端对齐，但在 Safari 浏览器中不会生效，需使用伪元素
      &::after{ // 以防 text-align-last 不起作用，故设置伪元素，假装其有换行
          content: " ";
          display:inline-block;
          width: 100%
      }
    }
```

rgba 和 opacity

opacity 后代元素都会继承 opacity 属性，而 rgba 后代元素不会继承不透明属性，故使用 rgba 代替 opacity，取值皆是从 0.0 （完全透明）到 1.0（完全不透明）

```
// 设置分页器样式
    &-paginat {
        display: flex;
        width: 100%;
        position: absolute;
        bottom: 84px;
        justify-content: center;

        .swiper-pagination-bullet {
            width: 21px;
            height: 21px;
            margin-right: 9px;
            border: 2px solid #FFF;
            border-radius: 50%;
            background: rgba(0, 0, 0, 0);

            &-active {
                background: #6B91CE;
            }

            &:last-child {
                margin-right: 0;
            }
        }
    }
```

图片相关

图片可以作为背景（background-img）插入元素，可以通过 img 标签作为行内元素插入 dom

```
// 使用 background 设置图片背景
// 1，联用，不规定顺序，但是同一个属性不能分开，（位置 left 和 top 不能分开
background:10px #ccc 10px 报错）
background: url('') repeat #fff fixed center// 图片 重复 颜色 是否固定 位置
// 2，单用
background-image:url('')
```

- 设置背景和容器契合 直接使用 img(width:100%)，图片不一定完全覆盖容器，即高度不够；在 div 中使用 background，全覆盖和原比例都可以做到
    - background-size
        - cover 将图片按照原有比例放大至填满容器，设置 width:100% 或 height：100%，超过部分不可见，图片不变形
        - 100% 图片在 x 轴方向缩放至填满容器，y 轴不管，图片不变形
        - 100% 100% 图片宽度长度都按容器比例撑满，图片变形

```
background-repeat: no-repeat;
background-size: cover;
background-size: 100%;
background-size: 100% 100%;
```

    - img

```
display:block
width:100% // 根据父元素宽度缩放
```

display:flex

当弹性盒子设置不换行 no-wrap 时，子元素跟随弹性盒子宽度缩放，能缩则按子元素宽度等比缩放，即若设置宽度为 1：2：3，最后宽度也 1：2：3.但子元素最多缩放至**其子元素最大宽度**，例如子元素里有张图片，则最多缩到图片宽度。当出现不能缩的元素，比例改变，剩下能缩的按照比例缩，若最后所有子元素缩小完毕，弹性盒子仍不能容下，则超出（若纯文本盒子，最后缩小到一个字一行）.弹性盒子拥有六大属性，子元素也有六大属性 **弹性盒子**

- flex-direction：子元素排列主轴与顺序，取值：[row/column]-reverse
- flex-wrap：子元素是否换行， 及换行方向（从顶端排到下 or 底部排到上），取值：nowrap/wrap-reverse
- flex-flow：不常用，简写上面两个属性，默认为：flex-flow:row nowrap
- justify-content：主轴上的对齐方式，flex-satrt/end，center，space-between，space-around（每个项目两侧的间隔相等，项目之间的间隔比项目与边框的间隔大一倍）
- align-items：项目在交叉轴上的对齐方式，除了左右中对齐外，还有基线对齐（baseline），stretch（项目未设置高度或设为 auto，占满整个容器的高）
- align-content：功能和 align-items 差不多，但只有项目不止一行时起作用，取值为 flex-[start/end]、center、space-[between/around]、stretch、center **子元素**

- order：排列顺序
- flex-grow：放大比例，默认为 0
- flex-shrink：缩小比例，默认为 1
- flex-basis：在主轴占的空间，默认为原本项目大小
- flex：综合前面三个属性，默认为 0 1 auto，该属性有两个快捷值：auto (1 1 auto) 和 none (0 0 auto)
- align-self：不遵从父元素的对齐（align-item），自定义对齐方式，取值和 align-items 一样有六个 **弹性盒子缺点**
- flex 盒子的内容可以溢出，即内容宽度可能超过父元素宽度（有些内容溢出不涉及宽度变化：比如 { width: 10px; // 固定宽度 white-space: nowrap; // 但不换行} ）
- 若子元素写了 { overflow: hidden; }，则宽度不会溢出，文字内容也可以点点点。但会影响一些绝对定位等位置需要超出该元素的内容。 **检查弹性盒子是否溢出** justify-content: center，若盒子移位了，表示溢出

position

设置元素位置，默认为 static，即静止不移动，当 position 设置为 relative/absolute/fixed 时，通过 top/left/right/bottom 可改变元素位置，其中 absolute/fixed 是脱离文档流的，但 relative 是不会脱离文档流，即 relative 会占据着移动之前的位置，但是 absolute 和 fixed 就不会）

min-width、width 和 max-width

优先级：min-width = max-width > width 当 max-width > width > min-width 时，宽度的标准是 width，否则取最大/小宽度，min-width、max-width 设置百分比时，都会继承父元素的当前显示宽度

position 和 margin 冲突

绝对定位是根据相对于父元素的 top/left/right/bottom 来定位的，而 margin 是根据自身当前位置来定位的，故设置 margin 失效 1，元素在绝对定位以后，left/rigth/top/bottom 是没有优先等级的，不像 margin-left 作用的时候 margin-right 没用，如果现在 left:0,right:0,两方实力相当，浏览器没办法，都得满足

```
position:absolute
left:0 // 与 right:0 优先级相同
right:0
margin:0 auto
```

2，当它距离父元素 left:50%,top:50%，那就是父元素一半的距离，因为要实现居中即自身的中点在父元素的中间才算，所以 margin-left/margin-top 负的自身宽/高的一半，那么正好水平垂直居中，但是由于 margin 相对于父元素，故不得使用百分比。

```
position:absolute
left:50%
right:50%
margin-left:-50px;// 具体数值
margin-top:-50px;
```

垂直水平居中

- 水平居中，直接设置 margin

```
margin:0 auto
```

- 垂直居中
  - position 结合 margin

```
{
    position:relative;
    top:50%;
    margin-top:-50px;
}
```

  - transform 结合 position

```
{
    position:relative;
    transform:translateX(-50%)
}
```

  - 弹性盒子

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

  - 表格元素：显示设置父元素为：table，子元素为：cell-table，这样就可以使用 vertical-align: center，实现垂直居中

```
.parent{
    display:table
}
.son{
    display:table-cell;
    vertical-align:middle;
    text-align:center
}
```

css 样式初始化

初始化可以解决浏览器的兼容问题，因为不同浏览器对有些标签的默认值是不同的

- 通配符初始化，简单粗暴，但初始化所有标签，浪费性能

```
*{
    paddind:0;
    margin:0
}
```

伪元素和伪类

伪类通过添加类来实现；伪元素通过添加实际的元素来实现，伪元素创建了一个不在文档树上但实际存在的新的元素，所以不能通过 js 来操作，仅仅是在 CSS 渲染层加入，要配合 content 属性一起使用

- 伪元素失效 在 input、radio、select 等表单标签中，伪元素失效，::before 的定义：在指定元素的内容之前插入内容。注意：是元素内容之前，而不是元素之前。而 input 并不是容器，所以没有内容之前一说，所以就无效了。
- 伪元素使用，必须设置 content ，其余和正常项目一致，常使用 position 改变伪元素位置

```
&::before{
    content:'';
    position: absolute;
    top: 37px;
    left: -15px;
}
```

盒子两端对齐

即元素每一行两端对齐，但是最后一行靠左，类似于文字的 justify 两端对齐，最佳实现效果是根据容器的宽度排列，决定元素之间的间隙与单行个数

- 使用 margin-right 搭配 float，改变宽度时无法兼容

```
{
    margin-right:20px;
    float:left;
    &:nth-child(3+3n){
        margin-right:0;
    }
}
```

- 弹性盒子添加空元素，宽度与项目保持一致，高度设为 0

```
{
    display:flex;
    justify-content:space-between;
    flex-wrap:wrap;
}
.item-empty {
      height: 0px;
      width: 400px;
```

}

```
- 当一列只有 2/3 个时，使用伪元素

```less
  &::after {
              height: 0;
              width: 20%;
              min-width: 223px;
              content: "";
          }
```

js

深拷贝和浅拷贝

- json 化实现 先使用 JSON.stringify 将对象变成 json 字符串，再使用 JSON.parse 将 json 字符串转为新对象，缺点是遇到无法转换为 json 格式的属性时，例如 function、RegExp、undefined 等数据，转化直接忽略，即深拷贝的数据会丢失，可以使用 JSON.stringify() 对特殊类型进行格式化

```
let _obj = JSON.stringify(obj)
let obj = JSON.parse(_obj)
```

- Object.assign
- 递归实现 判断参数类型，判断参数属性类型，若参数属性为对象则递归调用，参数为非引用值则直接赋值

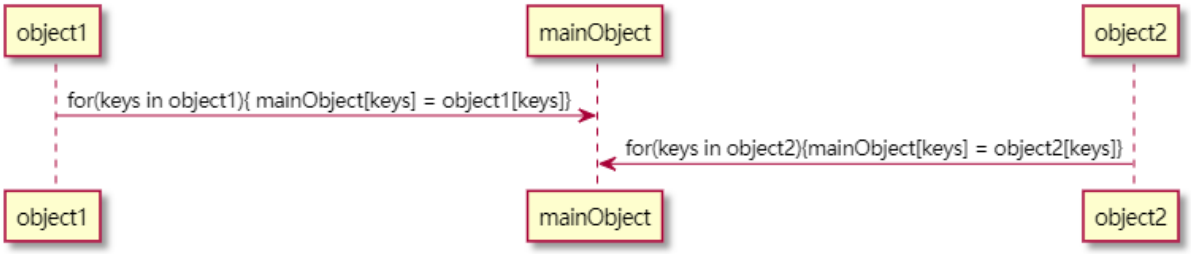
```
function deepClone(obj){
    let result
    if(typeof obj == 'object'){
        result = Array.isArray(obj) ? [] : {}
        for(let key in obj){
```
```

```
            if(typeof obj[key] !== 'object'){
                result[key] = obj[key]
            }else{
               result[key] =  deepClone(obj[key])
            }

        }
    }else{
        result = obj
    }
    return result
}
```

Object.assign(mainObject，...)

参数一是目标对象，接收后面参数对象**可枚举**的属性（一般对象的属性都是可枚举的，除非对象的 enumerable 属性为 false），目标对象有该属性则覆盖，无则添加，有一些拷贝的性质，但若参数的属性值为对象，则单凭赋值还是指向同一个地址，实现的是只有一层的拷贝



检测数据类型与 toString()

- typeof --- typeof xxx 判断数据类型，对于非引用类型可显示（string、boolean、number），对于数组、对象、实例、null 都识别为 object，对于 undefined 识别为 undefined，对于函数识别为 function
- instanceof --- xxx instanceof Object 返回一个 boolean 值，查看对象 B prototype 指向的原型对象是否在对象 A 的 prototype 原型链上，若对象 B 的 prototype 为 null 将会报错，类似于空指针异常，不可以检测非引用类型，因为没有原型对象，对象 A 必须是对象
- constructor 对象的 constructor 指向创建该对象的构造函数，但是不常用这个判断对象类型，因为 contructor 的指向是可以通过赋值操作被改变的，其中 null、undefined 没有 construtor，其余可以被检测出来
- Object.prototype.toString.call(obj) 返回一个形如'[Object type]'的字符串，例如'[Object String]'，Array、String 中的 toString 方法是被修改过的，故不能直接使用 toStirng 检测，而要使用 Object 对象原型的 toString 检测

Object 的内部方法

- Object.defineProperty(obj，property，descriptor) 操作对象的属性，有则修改，无则添加，参数三为一个对象，控制**属性描述符**对象，属性描述符对象有六个属性，不止控制属性的 value（属性值），还有 writable（是否可改），enumerable（是否可枚举）等，通过 defineProperty 定义的属性，与普通定义的属性不同，因为它是默认不可枚举、不可修改的、不可删除的。

```
Object.defineProperty(obj, 'name',{
    value:'',
    writable:true, // 是否可修改
    enumerable:true, // 是否可枚举
    configurable:true// 是否可修改
})
Object.defineProperty(obj, 'name',{})// 定义并传空，默认描述符如下
obj.descriptor === {
    value:undefined, // 默认未定义
    writable:false, // 不可修改
    enumerable:false, // 不可枚举
    configurable:false// 不可修改
    set:undefined, // 默认未定义
    get:undefined, // 默认未定义
}
```

- Object.defineProperties(obj， properties) 批量操作对象的属性，参数二对象每一个属性都操作一个对象属性，属性名为对象属性名，属性值为对象的属性描述符

```
Object.defineProperties(obj, {
    'name':{
        vaule:,
        writable:
    },
    'age':{
        vaule:,
        writable:
    }
})
```

- Object.getOwnPropertyDescriptor(obj，property) 获取对象对于属性的属性描述符对象，返回一个对象

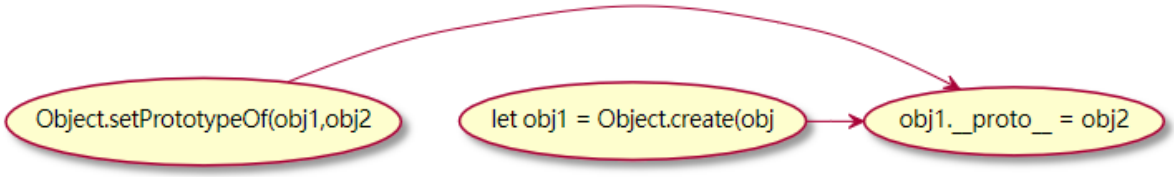
```
let descriptor = Object.getOwnPropertyDescriptor(obj, 'name')
descriptor// 包含六个属性的 name 属性描述符对象
```

- Object.getOwnPropertyDescriptors(obj) 获取对象所有属性的属性描述符对象，返回一个对象，每一个属性都表示一个对象属性的描述符对象

```
let descriptors = Object.getOwnPropertyDescriptors(obj)
descriptors === {
    'name':{
        // 包含六个属性的 name 属性描述符对象
    },
    'age':{
        // 包含六个属性的 name 属性描述符对象
```

```
    }
    }
```

- Object.getPrototypeOf(obj) 返回对象的原型对象，若没有继承的原型对象，则返回 null，



- Object.setPrototypeOf(obj1,obj2) 设置一个对象的原型对象，等价于 obj1.**proto** = obj2
- Object.create(obj) 其中参数 1 表示被创建出来新对象的原型对象，等价于 obj1.**proto** = obj2
- Object.preventExtensions(obj) 将传入对象设置为不可扩展的，即可以操作原有属性但不能再添加新属性，es5 添加新属性会抛出错误，es6 后使用赋值添加新属性 (obj.age = '') 不报错也没有效果，但若是使用 defineProtoype 定义属性会报错。此外，Object.isExtensible() 用于检查对象是否可扩展，传入一个对象返回 boolean 值，若传入一个非对象的变量，es5 抛出错误，es6 则将变量强制转为对象并返回 false
- Object.keys(obj) 返回**可枚举**属性组成的数组

reflect 对象

ES6 提供的一个将常见 js 对象内部方法（Object.xxx(obj) or Object.prototype.xxx(obj)）封装并反射出来的对象（reflect.xxx(obj)），原因是：1，内部的方法不希望被暴露；2，reflect 返回值更合理，使用 defineProperty 方法 Object 报错而 reflect 只是返回 false；3，Object 存在命令式，例如：delete obj.name，不符合面向对象的思想，reflect 是对象，纯函数式调用方法，变成 reflect.deleteProperty(obj，name)。reflect 对象拥有 13 个方法，且对第一个参数严格控制，若不传对象/函数，报错。

- get(obj，attribute，receiver) get 和 set 的最后一个参数都是用来绑定 this 的，当属性部署了读取函数(get name(){})/赋值函数(set name(){})时起作用
- set(obj，attribute，value，receiver)
- has(obj，attribute) 返回一个 boolean 值，和 attribute in obj 效果相同
- deleteProperty(obj，attribute) 删除对象的某个属性
- construct(obj，args) 参数一必须是函数，等同于 new 实例化。reflect.construct(Fun，'lyf') 等价于 new Fun('lyf')，不常用
- apply(fun，receiver，args) 参数一是函数，参数二表示绑定的 this，参数三表示传入函数的实参，与 fun.apply 一致，实参传入一个数组
- getPrototypeOf(obj) 获取对象的原型对象，返回 boolean 值
- setPrototypeof(obj，newProto) 设置对象的原型对象，返回一个 boolean 值
- defineProperty(obj，attribute，descriptor) 设置对象的标识属性
- getOwnPropertyDescriptor(obj，)
- isExtensible(obj) 检查对象是否可扩展
- preventExtensions(obj) 将对象设置为不可扩展
- ownKeys(obj) 返回一个包含所有属性名的数组

reflect 和 Object 的对比

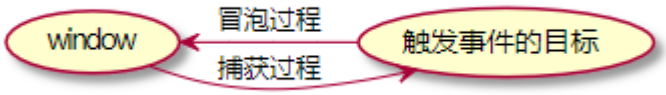
reflect	Object	功能
reflect.has(attribute)	attribute in obj	检查属性是否在对象中
reflect.set(obj，attribute，value，receiver)	obj[attribute] = value	设置对象属性，常用后者
reflect.get(obj，attribute，reciver)	obj[attribute]	获取对象属性，常用后者
reflect.deleteProperty(obj，attribute)	delete obj[attribute]	删除对象的属性，使用前者*
reflect.ownKeys(obj)	Object.keys(obj)	皆返回一个属性组成的数组，前者返回所有属性，后者只返回可枚举属性
reflect.defineProperty(obj，attribute，descriptor)	Object.defineProperty(obj，attribute，descriptor)	全面地**描述**属性，前者返回 Boolean 值，后者返回设置好的对象
reflect.getOwnPropertyDescriptor(obj，attribute)	Object.getOwnPropertyDescriptor(obj，attribute)	返回对象某属性的属性描述符对象，二者几乎一致，无该属性则返回 undefined
reflect.getPrototyOf(obj)	Object.getPrototyOf(obj)	返回原型对象，如无返回 null，二者几乎一致
reflect.setPrototyOf(obj，prototype)	Object.setPrototyOf(obj，prototype)	设置对象的原型对象，前者返回 Boolean 值，后者返回新对象
reflect.isExtensible(obj)	Object.isExtensible(obj)	检查对象是否可扩展，二者几乎一致
reflect.preventExtensions(obj)	Object.preventExtensions(obj)	将对象设置为不可扩展的，前者返回 Boolean 值，后者返回设置好的对象
reflect.apply(fun，receiver，args)	Function.prototype.apply(receivers，args)	二者功能几乎一致，常使用后者

reflect	Object	功能
reflect.construct(Fun，args)	new Fun(args)	构造函数实例，常使用后者

js 中的事件

在页面中，一些用户触发的行为，浏览器对象 BOM 触发的行为，可以称为事件，js 捕获事件并触发对应的回调函数，称为 js 事件

- 事件流 过程是根节点到目标节点，再从目标节点到根节点，例如一个 div 触发点击事件，流程为：document->html->body->div(click)->body->html->document，前面为捕获事件，后面为冒泡事件



- 监听事件 1，html 属性 属性名为 on + event，绑定的事件需**调用**，类似于事件触发时，调用一个早已准备好的函数

```
<div onclick="clickHandle()"></div>
```

2，dom 属性 在 dom 中获取元素并在事件属性中绑定函数，需确保 dom 中有此元素，故在 window.onload 中执行，属性值一般是唯一的，所以只能绑定一个函数

```
window.onload = function(){
document.getElementById('div1').onclick = clickHandle// 需注意在
window.onload 中，document 才能获取到页面节点
}
```

3，标准 dom 在 dom 中获取元素并在事件属性中绑定函数，需确保 dom 中有此元素，故在 window.onload 中执行，可以绑定多个事件函数，顺序是先绑定先执行 element.addEventListener(event，callback，isCapture) 参数 1 event 事件名的字符串，只能绑定确切存在事件，例如 click，不能自创事件，啥'show' 参数 2 callback 事件触发时绑定的回调函数，内有一个参数，代表触发事件的元素 参数 3 isCapture 表示事件是在捕获还是冒泡环节触发，boolean 值，默认为 false，即不再捕获阶段触发，而是冒泡

```
window.onload = function(){
document.getElementById('div1').addEventListener('click',(e)=>
{console.log(e)},true)
}
```

- 移除事件 只有使用标准 dom 绑定的事件函数可以移除，且必须移除同一个回调函数(地址) element.removeEventListener(event，callback，isCapture) 参数 1 event 必选，表示需移除的事件 参数

2 callback 必选，表示事件 参数 3 表示要移除的事件的触发阶段，即 addEventListener 的参数 3

```
element.addEventListener('click',function(){1},false)// 函数在内部，其他地方无法获取
element.addEventListener('click',function(){2},false)
element.addEventListener('click',function(){3},false)
element.addEventListener('click',function(){4},false)

elsment.removeEventListener('click',function{1})// 不会移除，因为这只是长的一样的函数
```

toString 和 valueOf 的区别

toString 返回一个表示该对象的字符串。valueOf 返回一个对象的**原始值**，即调用此方法的是什么类型，转换后仍是该类型。

类型	值	valueOf	值	toString	值
Array	[1,2,3]	数组	[1,2,3]	字符串	**'1,2,3'**
Boolean	true	布尔值	true	字符串	'true'
Number	123	数值	123	字符串	'123'
String	'123'	字符串	'123'	字符串	'123'
Object	{name:'lyf'}	对象	{name:'lyf'}	字符串	**'[object Object]'**
Date*	new Date()	数值时间戳	**1213145466**	字符串标准日期格式	'Fri Dec 23 2016 11:24:47 GMT+0800 (中国标准时间)'
Function	function fn(){}	函数	function fn(){}	函数字符串	'function fn(){}'

prototype 和 **proto** 的区别和联系

- **proto** 对象皆具有 **proto**属性，指向**构造该对象**的构造函数的原型对象，而原型对象是对象，也会有 **proto** 属性，指向构造其的构造函数的原型对象，以此生成 **proto__.proto.proto** 原型链。
- protptype 函数才拥有 prototype 属性，指向自身的原型对象，故对象 **proto** 和函数 prototype 有可能是等价的，当且仅当该对象是该函数构造的。在 prototype 对象上设置的属性，通过 new 实例化的对象可以访问，故 prototype 对象上的属性需严谨，具有一定的公共性。
- constructor prototype 对象有一个属性 constructor，指向构造函数（Object.prototype.constructor == object）。
- isPrototypeOf prototype 对象有一个 isPrototypeOf() 方法，判断**参数实例**是不是构造函数的实例。

Object 对象上操作原型对象的方法

getPrototypeOf() 返回传入参数的构造函数的 prototype。 setPrototypeOf() 传入两个参数，参数 1 是实例，参数 2 是一个对象，表示将参数 2 设置为参数 1 的原型对象。 create() 将参数作为新创建对象的原型对象。 若要获取对象自身的可枚举属性，使用 Object.keys()。 对于一个实例 obj，判断一个属性是原型还是自身定义的，

使用 obj.hasOwnProperty()，传入一个字符串参数，根据参数是不是实例自身属性返回一个 boolean 值，与之对应的操作符是 in，'name' in obj 不论参数在原型还是自身，都会显示 true，for-in 也是 in 的一种体现，遍历时也会返回原型的属性。

```
let obj = new Object() // 构造函数为 Object，原型对象由 prototype 获取
obj.__proto__ === 指向 Object 的原型对象 === Object.prototype
arr.__proto__.__proto__ === Array.prototype.__proto__ === Object.prototype

Array.prototype.constructor === Array

Array.prototype.isPrototypeOf(arr) // true

Object.getPrototypeOf(arr) == Array.prototype // true

let obj1 = {name:'lyf'} // 原型对象，包含属性 name
let obj = {age:111}
Object.setPrototypeOf(obj, obj1)
obj.__proto__ == obj1 // true
obj.hasOwnProperty('name') // false
'name' in obj // true
Objetc.keys(obj) // ['age']

let obj = Object.create(obj1)
obj.name  // 'lyf'
obj.__proto__ == obj1 // true
```

Vue

Vue 事件

通过 Vue 发送的事件，而不是 dom 事件，一般使用 emit 发送，on 接收，off 去除，和标准 dom 事件监听 addEventListener、removeEventListener 相对应

- vue.$emit(event，attributes) 参数 1 event 为表示事件的字符串，类似于 dom 事件的 'click'、'mousedown'，但是这个没有固定的限制 参数 2 attributes 表示传给监听事件回调函数的参数 本质上是给本组件监听，可以在本组件使用 vue.$on 和 vue.$once 监听，在父组件则是使用 v-on 监听子组件传出的事件，并调用回调函数，调用互不影响

```
// son.js
handle(){
    // 子组件发送了四个同名称事件 sonEvent
    this.$emit('sonEvent',[1,2,3])
    this.$emit('sonEvent',[1])
    this.$emit('sonEvent',[2])
    this.$emit('sonEvent',[3])
}
mounted(){
```

```
    this.$on('sonEvent',(data)=>{ // 触发四次
        console.log(data)
    })
    this.$on('sonEvent',(data)=>{ // 触发四次
        console.log('xxx')
    })
}

// farther.js
<son @sonEvent="handle"></son> // 监听子组件传出事件 sonEvent
handle(data){
    console.log(data) // [1,2,3] 触发四次
}
```

- vue.$on(event，callback) 在本组件上直接使用 this.$on() 调用。可以监听组件使用 emit 所发出的事件，可以多次监听同一个事件回调不同的函数。此外还有 this.$once，表示只监听一次，监听完销毁。
- vue.$off(event，callback) 移除事件监听器，可以移除所有事件的所有监听器，移除某个事件的所有监听器，移除某个事件的某个监听器
    - 当没有传 event 和 callback 时，移除所有监听器
    - 当只传 event 时，移除 event 的所有监听事件
    - 当传入 event 和 callback 时，只移除和 this.$on/$once 中参数一致的监听器

自定义指令

可以全局和局部自定义指令，使用 directive 注册一个指令，并设置其钩子函数，全局则是调用 Vue 原型方法 Vue.directive('orderName',{pFunction1,pFunction2})，局部则是属性对象 directives:{orderName1:{pFunction1,pFunction2},orderName2:{pFunction1,pFunction2}}，在 dom 元素中使用 v-orderName 即可，也可绑定属性

自定义指令的钩子函数(pFunction) 五个

- bind 绑定时调用，只调用一次，类似于 created 周期函数，此时元素还没有渲染
- inserted 插入时调用
- update 更新时调用，此时 dom 更新前
- componentUpdate 在 dom 更新完毕后调用
- unbind 解绑时调用，例如元素销毁

钩子函数的参数

使用这些钩子函数时，固定有几个参数

- el 表示绑定指令的元素，可以直接操作修改元素
- binding 一个**对象**，包含一些具体的属性
    - name 为指令名称，不含 v-
    - value 为指令绑定的值，v-xxx="name"，自定义指令没有:绑定符号，故需要此属性找寻 data 中的 name 属性或 name 函数，无则警告
    - expression 为指令绑定的字符串，以上直接返回 'name'
    - arg 为指令传入的参数，使用 ':' 符号绑定，例如 v-xxx:name，即传给指令一个参数 'name'

    - modifiers 一个对象，使用 '.' 绑定，不限个数，例如 v-xxx.a.b，显示 {a:true， b:true}
- vnode
- oldValue

```
// 自定义指令
directives:{
    input:{
        inserted:function(el){
            console.log('bind')
            el.focus()
        }
    }
}
```

ES6

class 类

定义一个类，直接 class className{constructor(attributes){}}，实例化一个类，直接 new className(attributes)

- 静态 static 没有被修饰符修饰的属性直接在实例上，static 修饰的属性不会被实例继承，只能被类调用

```
class FullName{
    static name = "lyf";
    anthorName = "anthor";
    constructor(firstName, lastName){
        this.firstName = firstName
        this.lastName = lastName
    }
    static sortFullName(){
        return this.firstName + this.lastName
    }
    reserveFullName(){
        return this.lastName + this.firstName
    }
}
let fullName = new FullName('l','yf')
// fullName 的结构
{
    // 不包括静态属性和方法
    anthorName:'anthor',// 类的公有属性
    firstName: 'l', // 传入的实例属性
    lastName: 'yf', // 传入的实例属性
    Prototype:{ // 原型上挂载类的公有方法和 constructor
        constructor: class FullName,
        reserveFullName: ƒ reserveFullName(),
        Prototype:Object// 原型的原型才是 Object 的原型对象
    }
```

```
  }
```

- 私有属性 ES6 没有私有属性的概念，使用形如'#attribute'，即在属性前加#符号，表达私有属性。虽然只有在类内部才能读取该属性，但其在实例中仍可以打印显示，只是#开头的属性违法，无法通过 class.#attribute 访问。这只是一种巧妙，使用其他符号定义属性也是一样的，只是约定是#，其他符号都犯法。此外，js 命名规则是只能使用字母、$符号、下划线开头

```
class Foo {
  #a;// 私有属性
  #b;// 私有属性
  #sum() { return this.#a + this.#b; }// 私有属性表示私有方法，故在第一层
  printSum() { console.log(this.#sum()); }
  constructor(a,  b) { this.#a = +a; this.#b = +b; }
}
let foo = new Foo(12,2)
foo  === {
    #a : 12,// 无法访问
    #b : 2,// 无法访问
    #sum : f;// 无法访问，虽是方法，但在第一层
    prototype:{
        constructor : f;// 只能通过自身属性（包括原型上的）去修改、展示内部私有属
性
        printSum : f// 只能通过自身属性（包括原型上的）去修改、展示内部私有属性
    }
}
```

- 继承 通过 ES6 的关键字 extends 继承，若在子类中自定义 constructor 构造函数，必须使用 super 关键字继承父类的构造函数，因为 extends 的机制是先解析父类的属性，再用子类的构造函数覆盖，super 使子类先继承父类的构造函数，必须设置在 constructor 最前面。super 作为函数调用只能在构造函数中，作为对象调用不受限，super 作为对象时在**普通方法**中指向父类的原型对象，即父类的静态方法；在**静态方法**中指向父类，即父类的普通方法。

```
class Farther {
    constructor(x, y){
        this.x = x;
        this.y = y
    }
}
class Son extends Farther{} // 没有构造函数，可以不使用 super
class Son extends Farther{
    constructor(x, y){
        // 有构造函数，必须使用 super
        this.x = 111 // 报错，因为没有经过父类
        super(x) // 将传入实例的参数赋值给父类，this.x = x
        this.x = 1111 // 在此之前 this.x = x，在此之后 x == 1111
```

```
    }
}
```

HTTP 请求

超文本传输协议，基于 tcp，有 0.9、1、1.1、2 等版本，常见发送 HTTP 请求的方式有 AJAX 和 axios

AJAX 请求

创建一个 ajax 请求一般分为五步。

- 1、创建对象

```
let xhr = new XMLHttpRequest()
```

- 2、使用 open 设置请求

```
xhr.open(method, url, isAsycn)
/*
* @params GET or POST
* @params  请求地址
* @params 是否异步，默认为 true，即 send 后不必等到其执行完毕
*/
```

- 3、使用 send 发送请求

```
xhr.send(requestData)
/*
* @params 请求时传的参数，因为默认传的内容为文本格式，故需要设置请求头，可传字符串和对象
* /
```

- 4、监听状态变化，一旦状态变化调用函数

```
xhr.onreadystatechange = function(){

}
```

- 5、根据返回的请求状态表示请求成功

```
  xhr.onreadystatechange = function(){
      if(xhr.readyState == 4 && xhr.status == 200){
          // 请求成功后的操作，一般操作 responseText
```

```
        }
    }
```

- 6、其余设置

解决传参不便的问题，需设置请求头

```
xhr.RequestHeader('Content-Type','application/x-www-form-urlencoded;charset=UTF-8')
```

axios

axios 是一个外部库，使用前需安装或引入。常用的 axios 方法参数都是不定的，可以只传一个 url 字符串，将参数拼至 url 后。也可以传多个参数，常见的方法有：axios()、axios.request()、axios.post()、axios.get()、axios.all()、axios.create()

```
// 安装或引入 axios
npm install axios
import axios from 'axios';

<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

请求配置项

当直接使用 axios(paramObj) 时，传入一个定义关于请求的对象。

- transformRequest、transformResponse 的格式是 数组嵌套函数，其中函数的参数就是请求参数，需返回一个值，表示操作后的请求参数:[function(data){return data}]。
- params 和 data 的区别：params 拼接在 url 后，但是 data 作为一个请求体被发送。get 请求没有 data 属性，该属性仅存在 put/post/patch 中

```
{
    url:'',
    method:''
    baseURL:'',
    // 在请求发送之前，对数据进行改动。例如：将 undefined、null 转为空，或者将 Array -> &字符串
    transformRequest:[function(data){
    // 此处根据需求改变数据
    return data;
    }],
    // 在请求响应到达`then/catch`之前，对数据进行改动
    transformResponse:[function(data){
    // 此处根据需求改变数据
    return data;
    }],
```

```
    // 操作自定义请求头信息
    headers: {'X-Requested-With':'XMLHttpRequest'},
    // 让参数 (params) 序列化
    paramsSerializer: function(params){
    return Qs.stringify(params, {arrayFormat:'brackets'})
    },
    // 请求参数，类型为一个纯对象，或 URLSearchParams 对象
    params: {
    ID:
    },
    data {
    firstName:
    },
    // 表示请求发出的延迟毫秒数，如果请求花费的时间超过延迟的时间，那么请求会被终止
    timeout:1000,
    // 表明是否是跨域请求，默认为 false，表示不是跨域请求
    withCredentials:false,
    // 表示返回数据的格式，可选值为：arraybuffer、blob、document、json、text、
stream，默认为 json
    responseType:'json',
    // 响应内容的最大值
    maxContentLength:2000,
    // 上传进度事件，常用于上传进度条
    onUploadProgress:function(progressEvent){},
    // 下载进度的事件，常用于下载进度条
    onDownloadProgress:function(progressEvent){},
    // 表明 HTTP 基础的认证和证书，这会设置一个 authorization 头 (header) ，并覆盖你在
header 设置的 Authorization 头信息
    auth: {
    username:"zhangsan",
    password: "s00sdkf"
    },
    // 适配器选项允许自定义处理请求，这会使得测试变得方便
    // 返回一个 promise，并提供验证返回
    adapter: function(config){
    /*.........*/
    },
    // 用作 xsrf token 的值的 cookie 的名称
    xsrfCookieName: 'XSRF-TOKEN',// default
    xsrfHeaderName:'X-XSRF-TOKEN',// default
    // validateStatus 定义了是否根据 http 相应状态码，来 resolve 或者 reject promise
    // 如果 validateStatus 返回 true(或者设置为`null`或者`undefined`),那么 promise
的状态将会是 resolved，否则其状态就是 rejected
    validateStatus:function(status){
    return status >= 200 && status <300;// default
    },
    // 表示 nodejs 中重定向的最大数量，默认为 5
    maxRedirects: 5, //default
    // httpAgent/httpsAgent 定义了当发送 http/https 请求要用到的自定义代理
    // keeyAlive 在选项中没有被默认激活
    httpAgent: new http.Agent({keeyAlive:true}),
    httpsAgent: new https.Agent({keeyAlive:true}),
    // proxy 定义了主机名字和端口号，
    // `auth`表明 http 基本认证应该与 proxy 代理链接，并提供证书
```

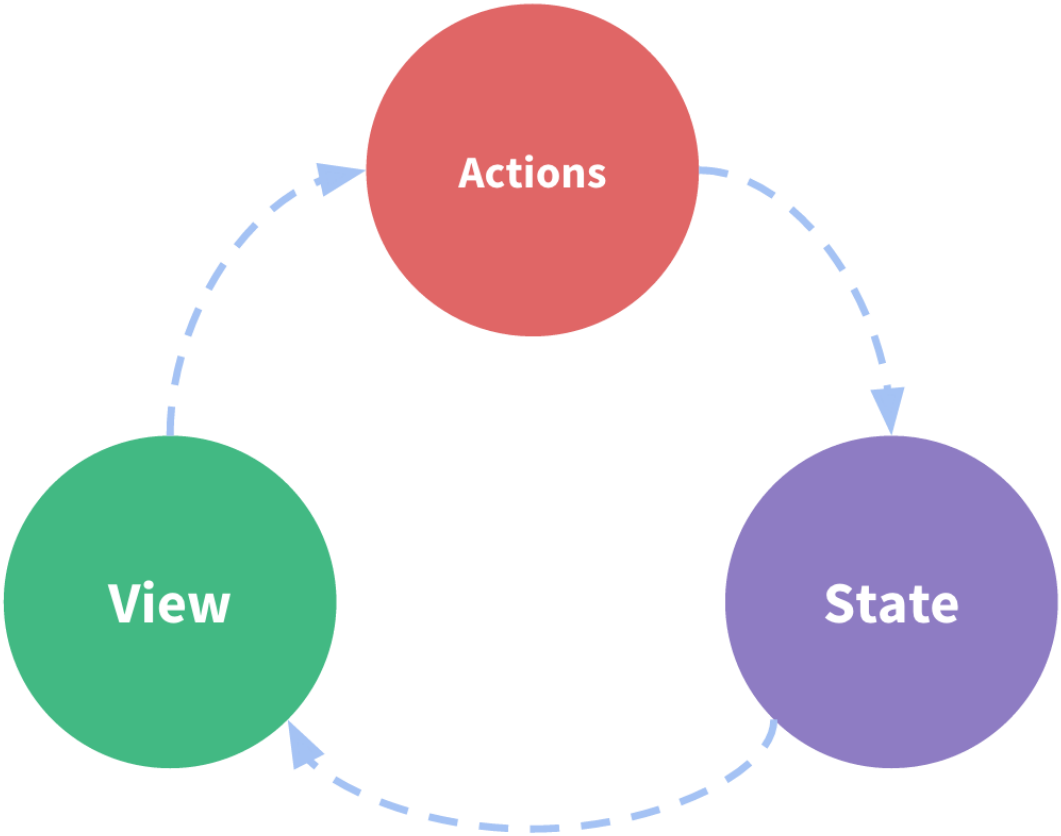
```
    // 这将会设置一个`Proxy-Authorization` header，并且会覆盖掉已经存在的`Proxy-
Authorization`  header
    proxy: {
    host:'127.0.0.1',
    port: 9000,
    auth: {
        username:'skda',
        password:'radsd'
    }
    },
    // `cancelToken`定义了一个用于取消请求的 cancel token
    // 详见 cancelation 部分
    cancelToken: new cancelToken(function(cancel){})
}
```

常见的请求状态码

状态码	功能
100	继续请求
200	请求成功
202	请求已接收，但未处理
204	请求已处理，但是没有返回任何实体内容，可能只更新了头部信息
301	请求已转移 url
404	请求失败，资源没有找到或不存在
500	服务器出错，无法完成请求
503	服务器由于过载或维护，无法完成当前请求

Vuex

可以调用 Vuex.store({}) 生成一个单一状态管理树，即一个应用只有一个仓库 store，但是所有 vue 实例共享仓库里的状态、方法，状态管理包含三部分，state，view 和 actions。



- 创建仓库 需安装并引入 vuex，进行注册，并挂载在 Vue 中

```
// store.js
import Vuex from 'vuex'
Vue.use(Vuex) // 注册插件
const store = new Vuex.Store({ // 初始化仓库
    state:{},
    mutations:{},
    actions:{},
    modules:{}
})
export default store

// index.js
import store from './store.js'
new Vue({
    store,
    render:h => h(App)
}).$mount('#app')
```

- 仓库属性

state

类似于 Vue 实例中的 data，但是在 vuex 中响应式，故需要实时监听，常常在 computed 中监听，若单纯在 data 中引入，原始值不响应，引用值的属性修改时响应(因为指向一个地址)

```
state:{
    count:10
}
```

getters

类似于 Vuex 的计算属性，也是监听一个函数的返回值;默认参数 1 为 state，操作后返回符合结果，无法获取到模块化的 getters

```
getters:{
    showName(state){ // 此处可以定义其他参数，但是都是本 store 中的数据，因为不能从外部调用
        return state.names.filter(item=>{item.age>10})
    }
}
computed:{
    showName(){
        return this.$store.getters.showName // 并不调用，而是直接获取
    }
}
```

mutations

类似于 Vue 实例中的 methods，但不能和 methods 一样方法间相互调用，且只能实现同步操作，无法异步。事件参数默认为(state.payload),故在提交事件时，只能提交一个参数，如需提交多个参数，使用对象。

```
this.$store.commit("moduleA/increment", 10, "10"); // '10' 无效
this.$store.commit("moduleA/increment", {num:10,str:"10"});
```

actions

解决 mutations 无法异步，用于异步完将 commit 到 mutations，虽然此处可以修改 state 的值，但是一般只在 mutations 修改 modules 为一个对象，因为是 vuex 单一树，故 store 只有一个，但是可以进行分割，在定义了命名空间时，各个模块之间数据是不共通的。在有模块化的 vuex 实例中，获取模块内的状态操作需加模块名前缀

```
let moduleA = {
    state:{
        count:10
    },
    mutations:{
        increment(state){
            state.count++
        }
```

```
    }
}

let store = new Vuex.Store({
    modules:{moduleA},
    state:{
        count:1
    }
})
```

- 在 Vue 实例中使用仓库 因为 vuex 在 Vue 中挂载了，故使用 this.$store 可以访问。且 vuex 内部有 mapXxxx 方法帮助 state 和 getters 快速生成计算属性，帮助 mapMutations 和 mapActions 快速生成函数

```
// App.vue
computed:{
    count(){
        return this.$store.state.count // 单模块
        return this.$store.state.app.count // 多模块，获取 app 模块数据
    }
}

// 辅助函数 mapState
import {mapState} from 'vuex'
computed: mapState(['count'])
computed:{
    ...mapState(['count'])
}

// getters
computed:{
    count(){
        return this.$store.getters.countCreamt // 单模块，modules 下
        }
}

// 辅助函数 mapGetters
computed:mapGetters(['getCount'])
computed:{
    ...mapGetters(['getCount'])
}
共有三种方式，如下：
//1.
commonGetter(){
    this.$store.getters['moduleA/moduleAGetter']
},
//2.
...mapGetters('moduleA',['moduleAGetter']),此处的 moduleA，不是以前缀的形式出现！！！
//3.别名状态下
...mapGetters({
    paramGetter:'moduleA/moduleAGetter'
```

```
})

// mutations
methods:{
    clickHandle(){
        this.$store.commit('addCount',10) // 单模块
        this.$store.commit('app/addCount',10) // 多模块，调用 app 模块下的
mutations
    }
}

// mapMutations
@click="addCount(5)" // 此时参数在绑定事件中携带
methods:{
    ...mapMutations(['addCount'])
}
// 辅助函数
...mapMutations('moduleA',['moduleAMutation']),
// 辅助函数别名
...mapMutations({
    changeNameMutation:'moduleA/moduleAMutation'
}),


// actions
methods:{
    clickHandle(){
        this.$store.dispatch('addCount',10) // 单模块
        this.$store.dispatch('app/addCount',10) // 多模块，调用 app 模块下的
actions
    }
}

@click="addCount(5)"
methods:{ ...mapActions(['addCount'])}
```

- 命名空间 当模块没有添加 namespaced 属性时，action 和 mutation 仍是全局的。添加 namespaced:true 表示该模块是命名空间，该模块的 getter、action 及 mutation 都会根据模块注册路径调整命名

零碎

Promise 调用的区别

Promise 是层级调用的，即 then 在同一层，调用就在一层，vue 刷新也只有一次

```
Promise(1).then(2)
Promise(3)
```

```
Promise(4)
// 执行顺序 1->3->4->2
```

CLS

全称 CommonLanguageSpecification，即公共语言规范，在定义类对象时作为文件名后缀。

IIFE

全称 Immediately Invoked Function Expression，即为立即调用的函数表达式。(()){}()

markdown 语法

- 页面内跳转 一个带 id 的 html 标签，代表要跳转到的地方： 跳转到的地方 当需要跳转时，使用 [](#标签id)，例如：点击跳转

热重载

也叫热更新，不需要刷新页面就更新

HTML 文档模式

在 html 文档的第一行通过 <!DOCTYPE> 声明文档模式，不是 HTML 标签，而是指示 浏览器编写的 HTML 版本、解析器使用的文档类型的指令。文档模式分为标准模式和混杂模式，一般是对 css 初始化的不同，例如标准盒子、图片是否有 3px 差异。文档类型定义 (DTD) 根据 DOCTYPE 标签决定，浏览器解析 DOCTYPE，在内部决定文档模式。

```
<!-- HTML5 -->
<!DOCTYPE html>

<!-- HTML 4.01 严格型 -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<!-- HTML 4.01 过渡型 -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<!-- HTML 4.01 框架集型-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">

<!-- XHTML 1.0 严格型 -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- XHTML 1.0 过渡型 -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<!-- XHTML 1.0 框架集型 -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

- 盒子模型 标准盒子(W3C)和怪异盒子(IE)，标准盒子 = content + padding + border + margin ,怪异盒子 = (content + padding + border) +margin，即怪异盒子将宽度视为 内容 + 内边距 + 边框

JSON 的方法

- JSON.stringify(obj， Array|callback， Number|String) 表示将 对象 变成 JSON 数据格式。 参数 1：表示需要 JSON 化的对象，是必须的。 参数 2：可选，函数 or 数组，用于过滤和自定义。当为过滤函数时，函数参数为参数 1 的 key 和值，必须拥有 return 值，常常 return val。若对象的属性为一个对象，即函数 return 一个对象，则下一次先遍历该对象内部，因为对象不能简单加 ""，类似于入栈，完了再出栈按照顺序遍历，直到所有遍历完退出；若为数组，表示需遍历的属性，不在列表的不被序列化。 参数 3：可选，表示字符串间距数值，每一级对比上一级缩进，可以传 '\t'等表示缩进的字符串

```
let obj = {
    name:'lyf',
    age:18,
    hobby:{
        sports:'football',
        foods:'beef'
    },
    sayName(){
        return this.name
    }
}
JSON.stringify(obj, function(key, val){// 3, hobby 对象入栈 --- 5, hooby 对象出栈, 继续遍历 sayName
    console.log("key is %s",key)// 0,初次遍历时 key 和 value 为 '' 和 object
    console.log("val is %s",val)// 1, 遍历 name 和 age --- 4, 遍历 hobby 对象的属性 sports 和 foods
    return val // 2, 遇到 hooby 对象
})
JSON.stringify(obj,  ["name", "info", "sex"]);
```

代码规范

- void 0 返回一个 undefined，比直接使用 undefined 严谨，因为在函数作用域中，undefined 是可以修改的，但是 null 不能修改。

```
function test(){
    let undefined = 10;
    return undefined
    }
  test() // 10
  function test(){
```

```
        let null = 10;
        return null
        }
    test() // error, 因为 null 不可做修饰符
```

Vue.use(plugins，...args)

这是 Vue 用于注册插件的方法。参数 1 plugins 为**对象**或**函数**，表示要注册的插件。 判断是否注册过该插件，有则返回避免重复注册。 参数 2~n 表示传入插件的数据，将其封装至数组中，且将 Vue 的 this 推入该数组头部。 若该插件是对象，且有 install 函数属性，使用 apply 调用该属性，且将数组作为参数数组传入; 若该插件是函数，使用 applay 调用，并将参数数组传入。

```
// Vue.use 源码
Vue.use = function (plugin: Function | Object) {
const installedPlugins = (this._installedPlugins || (this._installedPlugins = []))
// 判断是否注册过该插件，有则返回避免重复注册
if (installedPlugins.indexOf(plugin) > -1) {
    return this
}

// 将参数 2~n 封装至数组中
const args = toArray(arguments, 1)
args.unshift(this) // this 作为数组的第一项
if (typeof plugin.install === 'function') {
    plugin.install.apply(plugin, args)
} else if (typeof plugin === 'function') {
    plugin.apply(null, args)
}
installedPlugins.push(plugin)
return this
}
```

iview 和 Vue

iview 本就是为 Vue 所操作的组件库，故在 iview 的源码中，已经在 install 中注册了快捷组件使用，例如在 Vue 中可以直接使用 this.$Spin

```
// index.js
Vue.prototype.$Loading = LoadingBar;
Vue.prototype.$Message = Message;
Vue.prototype.$Modal = Modal;
Vue.prototype.$Notice = Notice;
Vue.prototype.$Spin = Spin;
```

如何在node_modules中进行查找文件

安装 Search node_modules 插件 按 F1 时输入 node 进入到 search node_modules 中，输入关键字 + 选中 + enter 进入文件夹