目录

HTML

如何理解 html 语义化

1. 让人更容易读懂（增加代码可读性）
2. **去掉或丢失样式的时候能够让页面呈现出清晰结构**
3. 让搜素引擎更容易理解（SEO 优化）

块级元素和内联元素

1. display:block/table. 有 div p h1 table ul 等
2. display:inline/inline-block. 有 span img input 等

p 标签里面不能嵌套 ul、div 等块级元素原因

不符合**语义化**规定，w3c 规定，对于 P 元素，它指定了以下内容，这表明 P 元素只允许包含**内联元素**（包括 p 元素本身也不行）。

```
<!ELEMENT P - O (%inline;)* -- paragraph -->
```

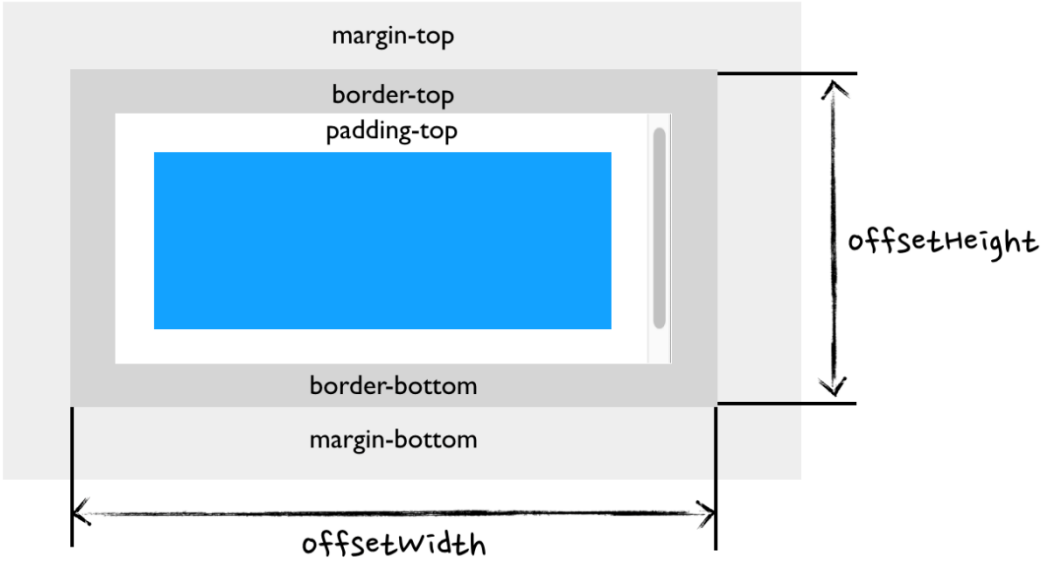
简而言之，不可能在 DOM 中放置`<div>`元素，因为开放的`<div>`标签会自动 closures<p>元素。

```
<p>
    11
    <div href='www.baidu.com'>baidu1</div>
    22
</p>
```

```
<p> 11 </p>
<div href="www.baidu.com">baidu1</div>
" 22 "
<p></p>
```

代码中的 offsetWidth 多大

offsetWidth 属性是一个只读属性，它返回该元素的像素宽度，宽度包含内边距（padding）和边框（border）




```
<!DOCTYPE html>
<html>
<head>
    <title>盒模型</title>
    <style type="text/css">
        #div1 {
            width: 100px;
            padding: 10px;
            border: 1px solid #ccc;
            margin: 10px;
            /* box-sizing: border-box; */
        }
    </style>
</head>
<body>
    <div id="div1">
```

```
        this is div1
    </div>
    <script>
      console.log(document.getElementById('div1').offsetWidth)
// 122
    </script>
</body>
</html>
```

将代码中的注释 box-sizing: border-box;打开，结果是 100。

box-sizing 的属性有两个 border-box（IE 盒模型/怪异盒模型）/content-box（标准盒模型）

margin 重叠问题

```
<!DOCTYPE html>
<html>
<head>
    <style type="text/css">
        p {
            font-size: 16px;
            line-height: 1;
            margin-top: 10px;
            margin-bottom: 15px;
        }
    </style>
</head>
<body>

    <p>AAA</p>
    <p></p>
    <p></p>
    <p></p>
    <p>BBB</p>

</body>
</html>
```

margin-top 和 margin-bottom 重叠，空白 p 被忽略，所以最后相距 15px

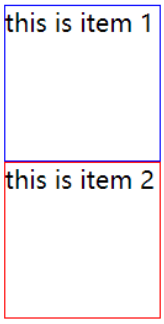
margin 负值问题

1. margin-top 和 margin-left 为负值时，元素向上或者向左移动
2. margin-bottom 为负值时，**下方元素上移，自身不受影响**
3. margin-right 为负值时，**右侧元素左移，自身不受影响**

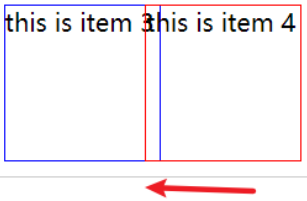
```
<!DOCTYPE html>
<html>
<head>
    <title>margin 负值</title>
    <style type="text/css">
        body {
            margin: 20px;
        }

        .float-left {
            float: left;
        }
        .clearfix:after {
            content: '';
            display: table;
            clear: both;
        }

        .container {
            border: 1px solid #ccc;
            padding: 10px;
        }
        .container .item {
            width: 100px;
            height: 100px;
        }
        .container .border-blue {
            border: 1px solid blue;
            margin-right:-10px;
        }
        .container .border-red {
            border: 1px solid red;
        }
    </style>
</head>
<body>
```

```
    <p>用于测试 margin top bottom 的负数情况</p>
    <div class="container">
        <div class="item border-blue">
            this is item 1
        </div>
        <div class="item border-red">
            this is item 2
        </div>
    </div>

    <p>用于测试 margin left right 的负数情况</p>
    <div class="container clearfix">
        <div class="item border-blue float-left">
            this is item 3
        </div>
        <div class="item border-red float-left">
            this is item 4
        </div>
    </div>

</body>
</html>
```

用于测试 margin top bottom 的负数情况



用于测试 margin left right 的负数情况



BFC 概念

在页面中元素都有一个隐含的属性叫作 Block Formatting Context，即**块级格式化上下文**，简称 BFC。该属性能够设置打开或关闭，默认是关闭的。

页面上的一个隔离渲染区域，容器里面的子元素不会在布局上影响到外面的元素.

一旦开启元素的 BFC 后，元素将会具备如下特性：

- 父元素的垂直外边距不会和子元素重叠
- 开启 BFC 的元素不会被浮动元素所覆盖
- 开启 BFC 的元素能够包含浮动的子元素

普通文档流布局规则

1. 浮动的元素是不会被父级计算高度
2. 非浮动元素会覆盖浮动元素的位置
3. **margin 会传递给父级**[根据规范，一个盒子如果没有**上补白**和**上边框**，那么它的**上边距**应该和其文档流中的第一个孩子元素的上边距**重叠**。]
4. 两个相邻元素上下 margin 会重叠

BFC 布局规则

1. 浮动的元素会被父级计算高度（父级触发了 BFC）
2. 非浮动元素不会覆盖浮动元素位置（非浮动元素触发了 BFC）
3. margin 不会传递给父级（父级触发了 BFC）
4. 两个相邻元素上下 margin 会重叠（给其中一个元素增加一个父级，然后让他的父级触发）【margin 重叠三个条件:同属于一个 BFC;相邻;块级元素】

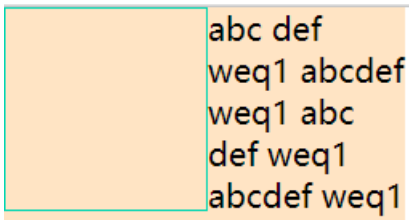
产生方式

- float 不为 none
- overflow 不为 visible
- position 不为 relative 和 static
- display 为 table-cell table-caption inline-block 之一
- 根元素

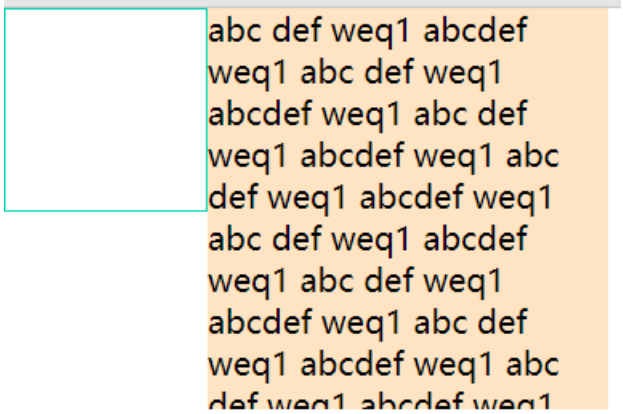
BFC 作用 :**多栏布局,清除浮动,上下 margin 重叠**

```
<!DOCTYPE html>
<html>
```

```
    <head>
        <meta charset="utf-8" />
        <meta name="viewport" content="width=device-width,
initial-scale=1">
        <title></title>
        <style>
            *{margin:0;padding:0}
            .k1{
                width:100px;
                height:100px;
                border:1px solid #00D6B2;
                float:left;

            }
            .k2{
                width:200px;
                height:200px;
                background-color: bisque;
                /* overflow: hidden; */

            }
        </style>
    </head>
    <body>
        <div class="k1">

        </div>
        <div class="k2">
            abc def weq1 abcdef weq1
            abc def weq1 abcdef weq1
            abc def weq1 abcdef weq1
            abc def weq1 abcdef weq1
            abc def weq1 abcdef weq1
            abc def weq1 abcdef weq1
            abc def weq1 abcdef weq1
            abc def weq1 abcdef weq1
            abc def weq1 abcdef weq1
            abc def weq1 abcdef weq1
            abc def weq1 abcdef weq1
        </div>

    </body>
</html>
```



放开 k2 中/* overflow: hidden; */



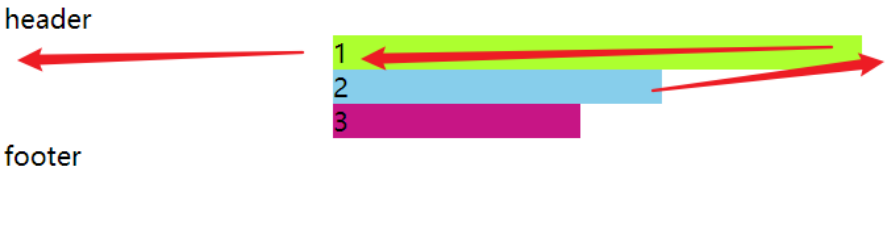
圣杯布局和双飞翼布局

圣杯布局是利用父容器的左、右内边距+两个列的相对定位

```
<style>
*{
    margin:0;
    padding:0
}
#container {
  padding-left: 200px;
  padding-right: 150px;
}
#container .column {
  float: left;
}

#center {
  width: 100%;
  background:greenyellow
```

```
}


#left {
  width: 200px;
  margin-left: -100%;
  position: relative;
  right: 200px;
  background: skyblue;
}
#right {
  width: 150px;
  margin-right: -150px;
  background: mediumvioletred;
}

#footer {
  clear: both;
}

</style>
<div id="header">header</div>
<div id="container">
  <div id="center" class="column">1</div>
  <div id="left" class="column">2</div>
  <div id="right" class="column">3</div>
</div>
<div id="footer">footer</div>
```

div 内容为 2 的位置因为浮动本应该在 1 之后，然后设置 margin-left：-100%；相当于向左移动了整个#container 位置，还需要在通过定位移动 200px



div 内容为 3 的位置，设置了 margin-right 负值，本来应该影响右侧内容，但是它的右侧没内容，假设有，右侧内容会慢慢左移，当 margin-right 负值等于 div3 的大小，就**相当于完全遮盖住**，外界感觉 div3 相当于没了宽度，**没了宽度的 div3 自然可以移动上去**。

双飞翼布局：多一个 div 包裹，中间 div 用 margin 不是 padding，左侧不需要再借助定位改变位置，右侧不需要通过 margin-right 直接使用 margin-left 即可

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>双飞翼布局</title>
    <style type="text/css">
        body {
            min-width: 550px;
        }
        .col {
            float: left;
        }

        #main {
            width: 100%;
            height: 200px;
            background-color: #ccc;
        }
        #main-wrap {
            margin: 0 190px 0 190px;
        }

        #left {
            width: 190px;
            height: 200px;
            background-color: #0000FF;
            margin-left: -100%;
        }
        #right {
            width: 190px;
            height: 200px;
            background-color: #FF0000;
```

```
            margin-left: -190px;
        }
    </style>
</head>
<body>
    <div id="main" class="col">
        <div id="main-wrap">
            this is main
        </div>
    </div>
    <div id="left" class="col">
        this is left
    </div>
    <div id="right" class="col">
        this is right
    </div>
</body>
</html>
```

清除浮动

1. overflow:hidden
2. 父级设置固定高度
3. clear:both;兼容性好，需要一个空 div，语义化不好
4. 万能清除法

```
.类名:after {
  content: "";
  clear: both;
  display: block;
  height: 0;
  overflow: hidden;
  visibility: hidden;
  zoom:1;
}
```

flex 布局

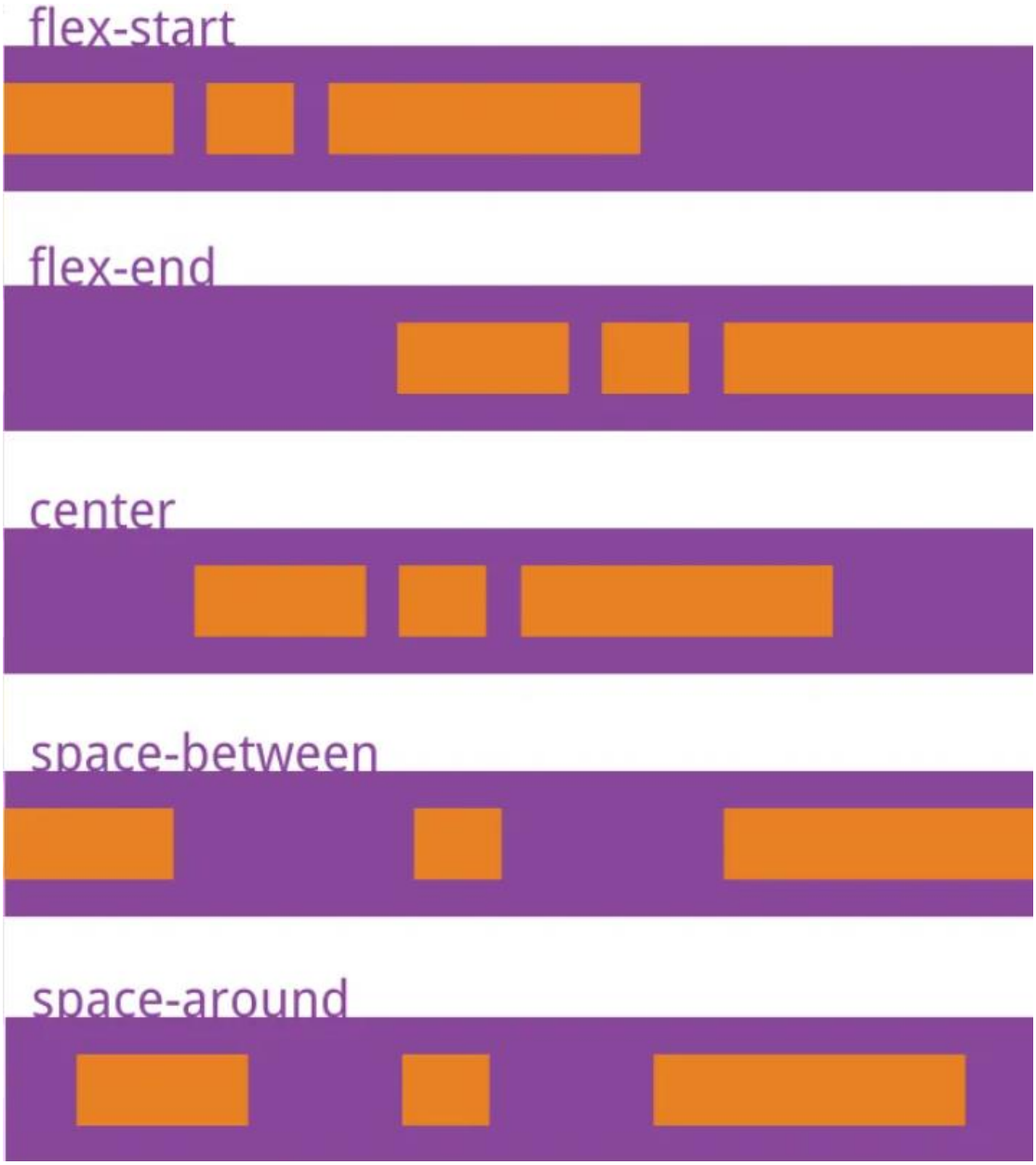
- flex-direction：设置主轴的方向
- justify-content：设置主轴上的子元素排列方式
- flex-wrap：设置子元素是否换行
- align-content：设置侧轴的子元素的排列方式（多行）

- **align-items**:设置侧轴上的子元素排列方式（单行）
- **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。
- **flex-flow**:复合属性，相当于同时设置了 flex-direction 和 flex-wrap
- **order**：定义项目的排列顺序。数值越小，排列越靠前，默认为 0。

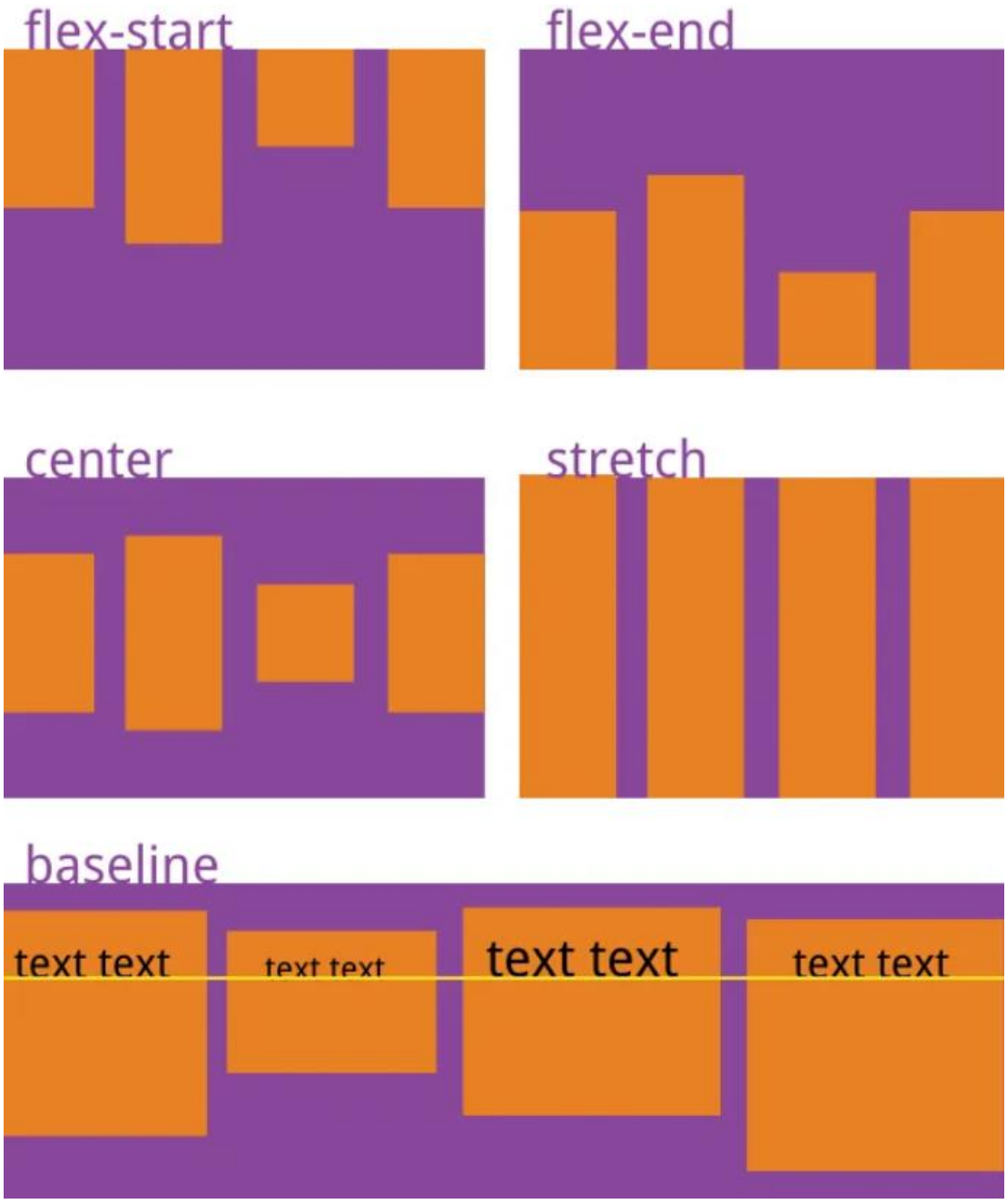
```
.box {
 justify-content: flex-start | flex-end | center | space-between
| space-around;
}
```

space-between：两端对齐，项目之间的间隔都相等。

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边框的间隔大一倍。

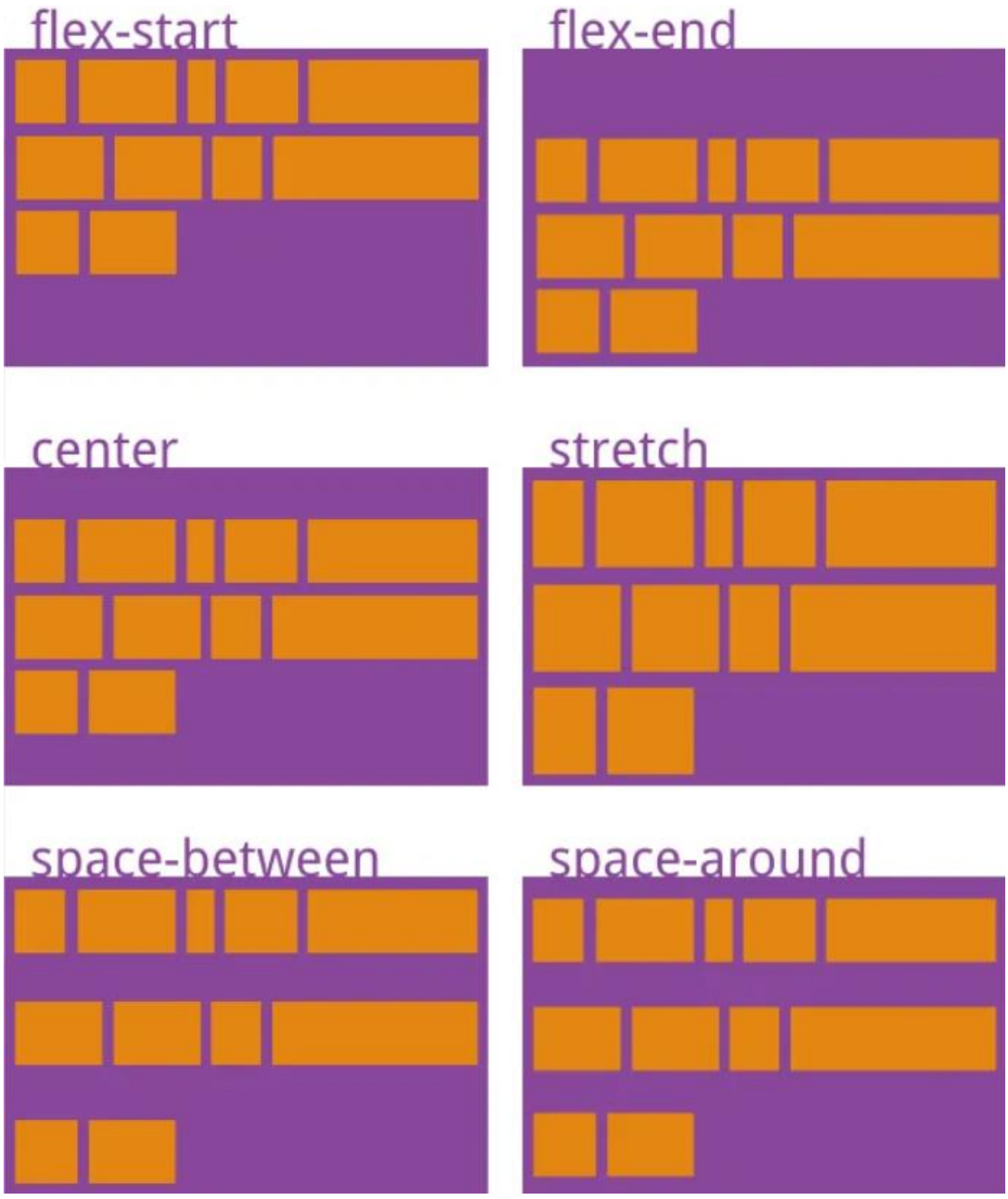



```
.box { align-items: flex-start | flex-end | center | baseline |
stretch; }
```



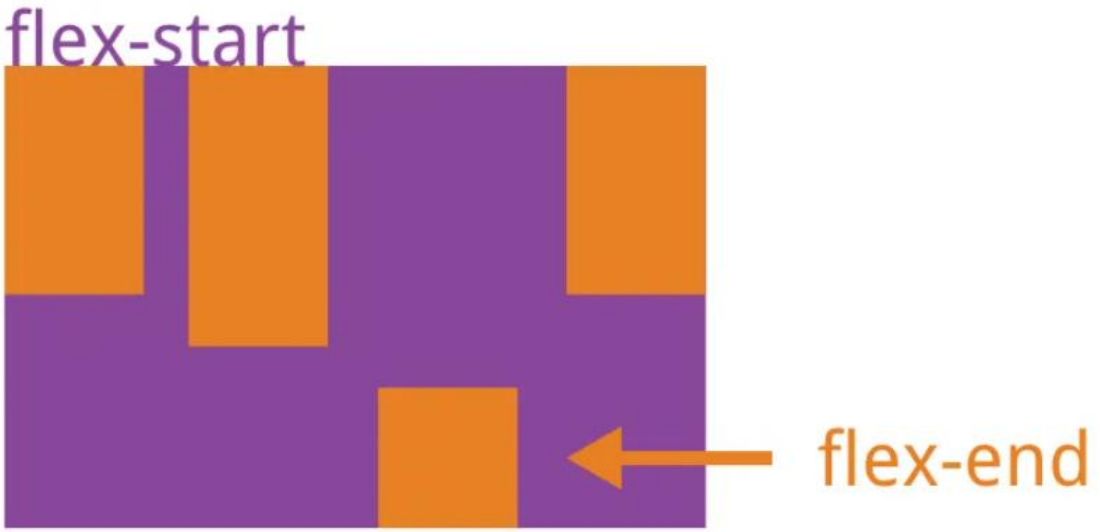
stretch（**默认值**）：如果项目**未设置高度或设为** auto，将占满整个容器的高度。

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



align-content 属性定义了**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。

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



align-self 属性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。

水平垂直居中设置方案

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>水平垂直居中方案</title>
    <style type="text/css">
        .container {
            border: 1px solid #ccc;
            margin: 10px;
            padding: 10px;
            height: 200px;
        }
        .item {
            background-color: #ccc;
        }
        /*行内元素*/
        .container-1{
            text-align: center;
            line-height: 200px;
            height: 200px;
        }
```

```
        .container-2 {
            position: relative;
        }
        /*需要知道宽高*/
        .container-2 .item {
            width: 300px;
            height: 100px;
            position: absolute;
            left: 50%;
            margin-left: -150px;
            top: 50%;
            margin-top: -50px;
        }

        .container-3 {
            position: relative;
        }
       /*不需要知道宽高，但是不兼容低版本浏览器*/
        .container-3 .item {
            width: 200px;
            height: 80px;
            position: absolute;
            left: 50%;
            top: 50%;
            transform: translate(-50%, -50%)
        }

        .container-4 {
            position: relative;
        }
        /*比较优秀的处理方案*/
        .container-4 .item {
            width: 100px;
            height: 50px;
            position: absolute;
            top: 0;
            left: 0;
            bottom: 0;
            right: 0;
            margin: auto;
        }
       .container-5 {
           position: relative;
          justify-content: center;
```

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

        }
        .container-5 .item {
            width: 100px;
            height: 50px;
        }

    </style>
</head>
<body>
    <div class="container container-1">
        <span>一段文字</span>
    </div>

    <div class="container container-2">
        <div class="item">
            this is item
        </div>
    </div>

    <div class="container container-3">
        <div class="item">
            this is item
        </div>
    </div>

    <div class="container container-4">
        <div class="item">
            this is item
        </div>
    </div>
   <div class="container container-5">
        <div class="item">
            this is item
        </div>
    </div>

</body>
</html>
```

line-height 继承

1. 写具体数值，如 30px，则继承父级该值
2. 写比例如 1/2/3.5 等,则继承该比例（自己的 font-size*父级中的比例）
3. 写**百分比**,如 200%,则继承计算出来的结果(父级的 font-size*200%)

rem em vw vh

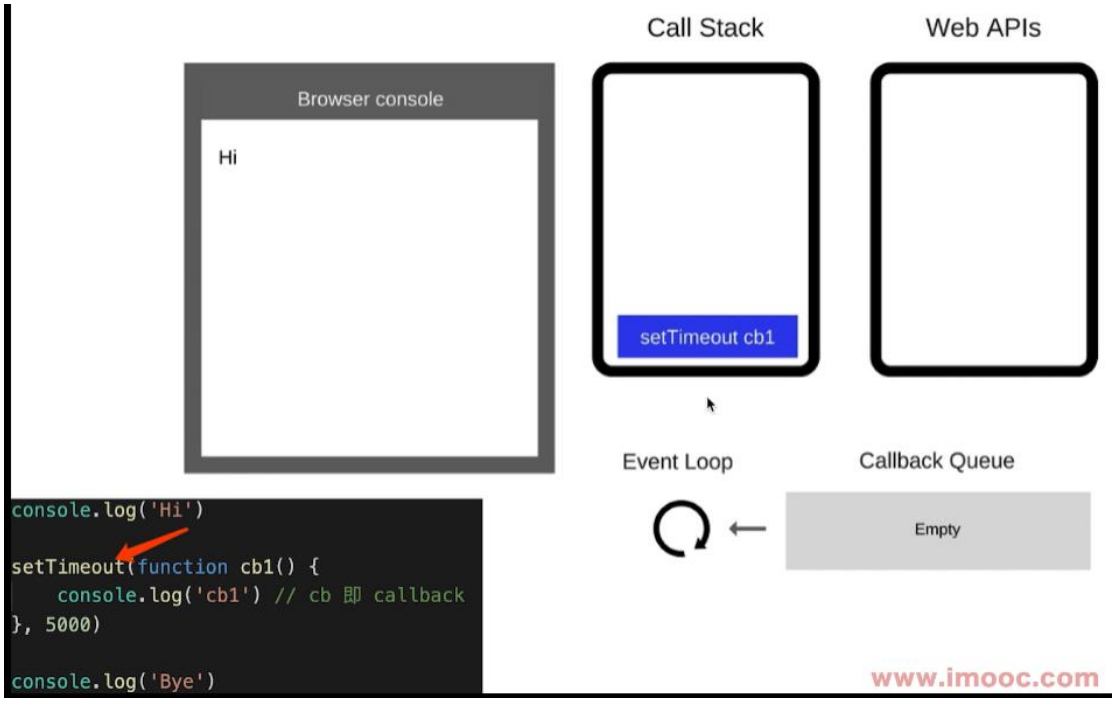
1. rem: 相对大小，但相对的只是 HTML 根元素
2. em: 继承父级元素的字体大小
3. vw:window.innerWidth = 100vw
4. vh:window.innerHeight = 100vh
5. vmax:取 vh/vw 中大值
6. vmin: 取 vh/vw 中小

JS

event loop

event loop 就是**异步回调**的实现原理

1. 同步代码，一行还在 Call Stack 中执行
2. 遇到异步，先“记录”下，等待时机（定时器、网络请求，dom 事件等）
3. 时机到了，移动到 Callback Queue
4. 如 Call Stack 为空（即同步代码执行完）
5. 轮询查找 Callback Queue，如有则移动到 Call Stack 执行
6. 然后继续轮询查找（永动机一样）



Promise 题打印出什么

- 三种状态 pending resolved rejected
- then 函数正常返回 resolved，里面有报错返回 rejected[如果有返回值，那么对应的 then 或者 catch 能拿到]
- catch 函数正常返回 resolved，里面有报错返回 rejected[同上]

```
Promise.resolve().then(() => {
    console.log(1) //1
  }).catch(() => {
      console.log(2)
  }).catch(() => {
      console.log(2)
}).catch(() => {
      console.log(2)
  }).then((val) => {
    console.warn(val)//undefined
      console.log(3)//3
})
```

```
Promise.resolve().then(() => { // 返回 rejected 状态的 promise
    console.log(1)
    throw new Error('erro1')
}).catch((e) => { // 返回 resolved 状态的 promise
  console.log(e)  //Error: erro1
    console.log(2) // 2
}).catch(() => {
    console.log(2)
}).catch(() => {
    console.log(2)
}).then(() => {
    console.log(3)//3
})
```

async await

async 函数返回结果都是 **Promise 对象**（如果函数内没返回 Promise ，则自动封装一下）

```
async function fn2() {
    return new Promise(() => {})
}
console.log( fn2() ) //Promise {<pending>}

async function fn1() {
    return 100
}
console.log( fn1() ) // 相当于 Promise.resolve(100)
```

await 相当于 Promise 的 then.

await 后面跟 Promise 对象：**会阻断后续代码**，等待状态变为 resolved ，才获取结果并继续执行

await 后续跟非 Promise 对象：会直接返回该值，不过也是要等其他同步代码执行完后。如 const f = await 400 ，内部相当于做了转换 const f= await Promise.resolve(400)

```
!(async function () {
    const p1 = new Promise(() => {})
    await p1
    console.log('p1') // 不会执行
})()

!(async function () {
    const p2 = Promise.resolve(100)
    const res = await p2
    console.log(res) // 100
})()

!(async function () {
    const res = await 100
    console.log(res) // 100
})()

!(async function () {
    const p3 = Promise.reject('some err')
    const res = await p3
    // new_file.html:34 Uncaught (in promise) some err
    console.log(res) // 不会执行
})()
```

使用 try…catch 捕获异常

```
(async function () {
    const p4 = Promise.reject('some err')
    try {
        const res = await p4
        console.log(res)
    } catch (ex) {
        console.error(ex)
    }
})()
```

async await 异步本质

await 是同步写法，但本质还是**异步调用**。即只要遇到了 await ，后面的代码都相当于放在 callback 里。

await 虽然会异步调用，但是它后面的函数是立刻执行的。

```
async function async1 () {
  console.log('async1 start')
  await async2()
  console.log('async1 end') // 关键在这一步，它相当于放在 callback 中，最后执行
}

async function async2 () {
  console.log('async2')
}

console.log('script start')
async1()
console.log('script end')
```

event loop 宏任务 微任务 和 dom 渲染的关联

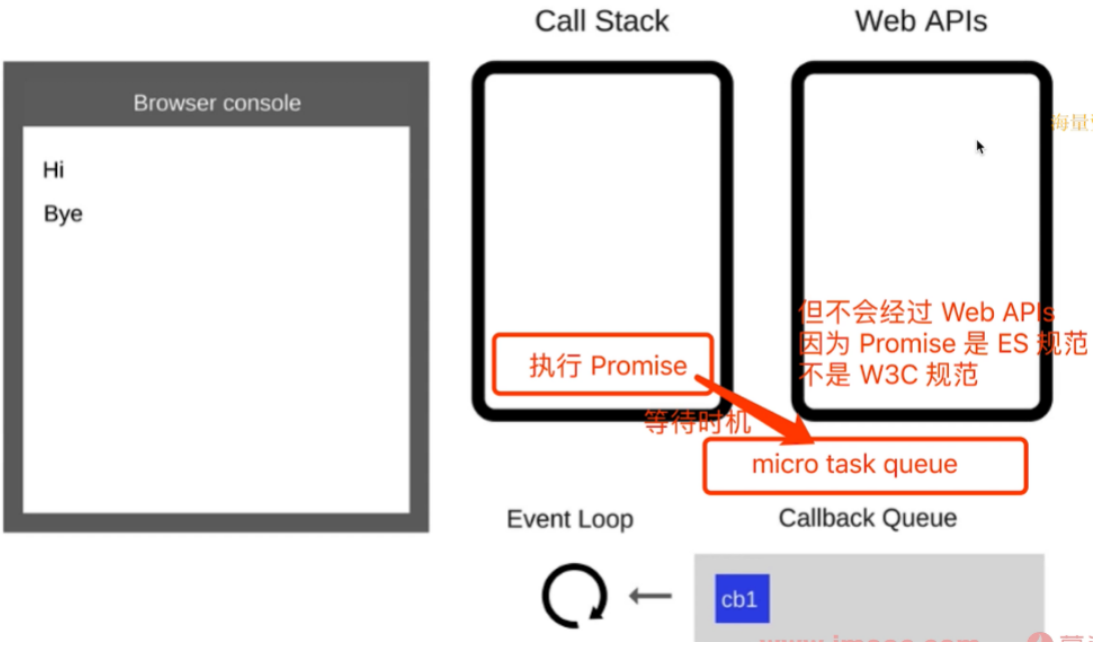
每一次 call stack（当前轮询结束） 结束，都会触发 DOM 渲染（不一定非得渲染，就是给一次 DOM 渲染的机会！！！）然后再进行 event loop。

宏任务：**DOM 渲染后再触发，**ES 语法没有，JS 引擎不处理，**浏览器（或 nodejs）干预处理。**

微任务：**DOM 渲染前会触发，ES 语法标准之内，**JS 引擎来统一处理。即，不用浏览器有任何关于，即可一次性处理完，**更快更及时。**

宏任务包含：setTimeout setInterval DOM 事件 ajax 等

微任务包含： Promise async await 等




```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <script
src="http://libs.baidu.com/jquery/2.0.0/jquery.min.js"></script>
  <body>
    <div id="container">

    </div>
  </body>
</html>
<script>
// 修改 DOM
const $p1 = $('<p>一段文字</p>')
const $p2 = $('<p>一段文字</p>')
const $p3 = $('<p>一段文字</p>')
$('#container')
    .append($p1)
    .append($p2)
    .append($p3)

// 微任务：渲染之前执行（DOM 结构已更新）
Promise.resolve().then(() => {
    const length = $('#container').children().length
    alert(`micro task ${length}`)
```

```
})

// 宏任务：渲染之后执行（DOM 结构已更新）
setTimeout(() => {
    const length = $('#container').children().length
    alert(`macro task ${length}`)
})
</script>
```

JavaScript 有哪些数据类型，它们的区别？

JavaScript 共有八种数据类型，分别是 Undefined、Null、Boolean、Number、String、Object、Symbol、BigInt。

其中 Symbol 和 BigInt 是 ES6 中新增的数据类型：

●Symbol 代表创建后独一无二且不可变的数据类型，它主要是为了解决可能出现的**全局变量冲突**的问题。

●BigInt 是一种数字类型的数据，它可以表示任意精度格式的整数，使用 BigInt 可以安全地存储和操作大整数，即使这个数已经超出了 Number 能够表示的安全整数范围。

Null 不同人有不同的观点，有认为是值类型，有认为是引用类型的（指针指向空地址）

Function 也可以单独抽离出来，是个特殊的引用类型，一般不需要存值，也没拷贝一说。

值类型存储在栈中，自上而下排列

栈	
key	value
a	100

栈	
key	value
a	100
b	100

栈	
key	value
a	200
b	100

引用类型存储在堆中，堆自下而上排列

栈	
key	value
a	内存地址1

栈	
key	value
a	内存地址1
b	内存地址1

栈	
key	value
a	内存地址1
b	内存地址1

内存地址1	{ age: 20 }
key	val
堆	

内存地址1	{ age: 20 }
key	val
堆	

内存地址1	{ age: 21 }
key	val
堆	

判断数据类型方法

1. typeof:判断基本数据类型以及函数，无法判断对象具体类型和 null
2. instanceof：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。
   - 缺点： instanceof 只能用来判断对象类型，原始类型不可以。并且所有对象类型 instanceof Object 都是 true
   - 优点：instanceof 可以弥补 Object.prototype.toString.call()不能判断自定义实例化对象的缺点。
3. constructor：可以判断包括简单类型、引用类型的构造函数，但是存在构造函数被修改过的风险

```
(2).constructor
//ƒ Number() { [native code] }
```

4. Object.prototype.toString.call()：相对来说比较合适，但是不能检测出自定义对象具体类型。

null 和 undefined 区别

首先 Undefined 和 Null 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 undefined 和 null。

undefined 代表的含义是**未定义**，null 代表的含义是**空对象**。一般变量声明了但还没有定义的时候会返回 undefined，null 主要用于赋值给一些可能会返回对象的变量，作为初始化。

undefined 在 JavaScript 中不是一个保留字，这意味着可以使用 undefined 来作为一个变量名，但是这样的做法是非常危险的，它会影响对 undefined 值的判断。**可以通过一些方法获得安全的 undefined 值，比如说 void 0。**

当对这两种类型使用 typeof 进行判断时，Null 类型化会返回“object”，这是一个历史遗留的问题。当使用双等号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

Object.is() 与比较操作符 “===”、“==” 的区别？

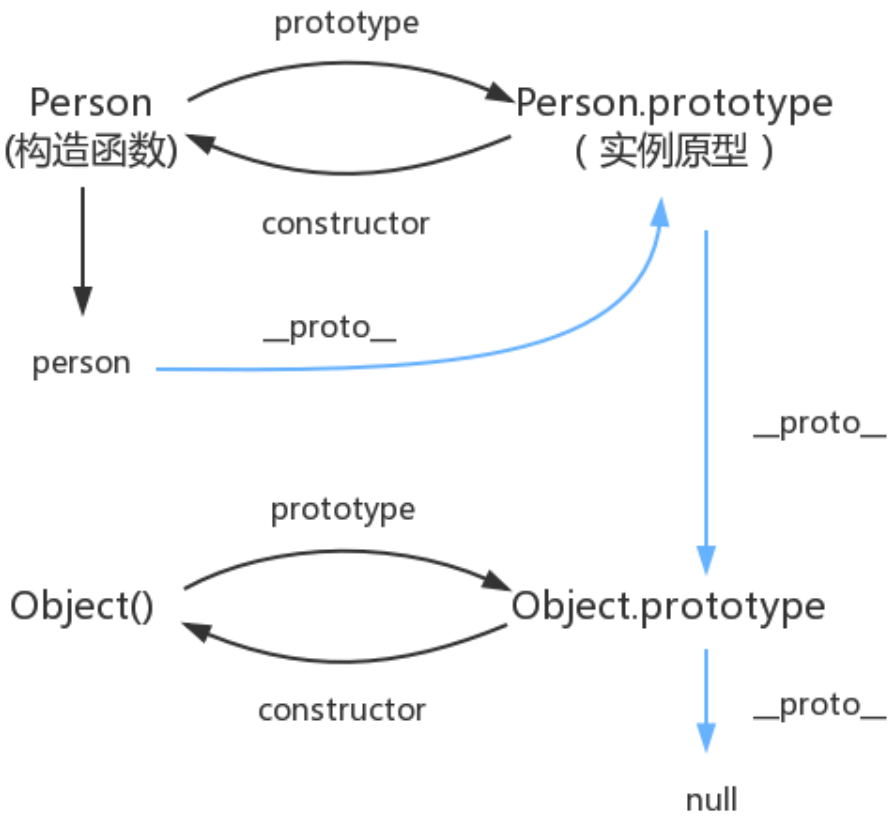
使用双等号（==）进行相等判断时，如果两边的类型不一致，则会进行强制类型转化后再进行比较。

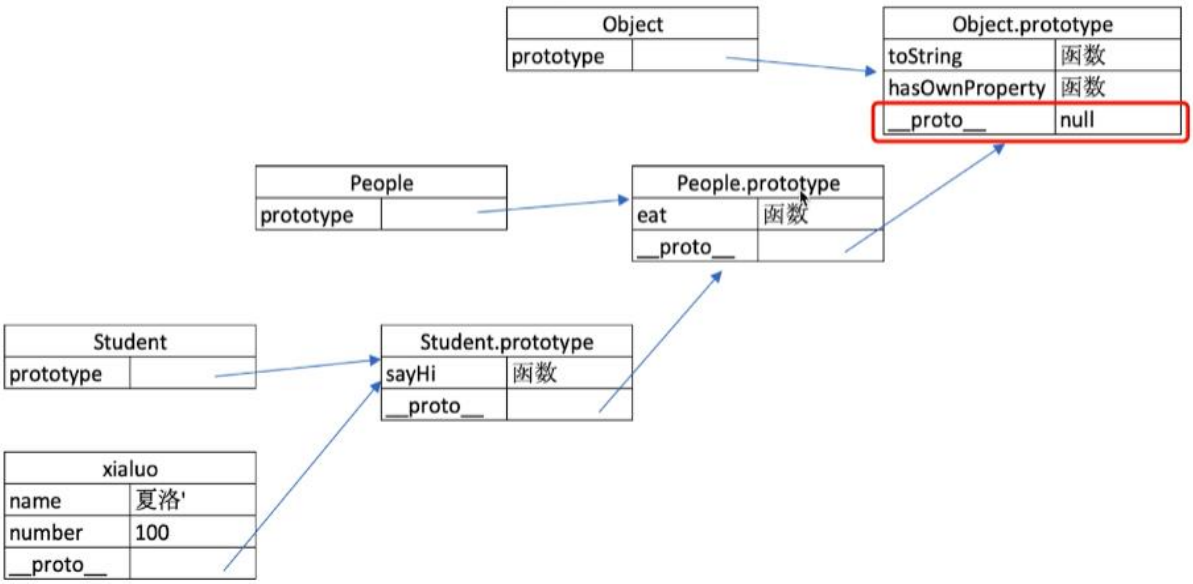
使用三等号（===）进行相等判断时，如果两边的类型不一致时，不会做强制类型准换，直接返回 false。

使用 Object.is 来进行相等判断时，一般情况下和三等号的判断相同，它处理了一些特殊的情况，比如 -0 和 +0 不再相等，两个 NaN 是相等的。

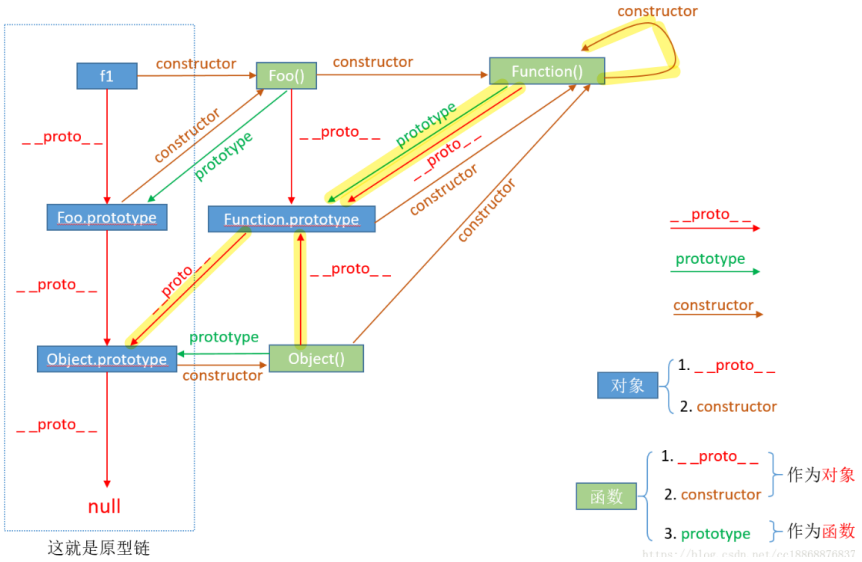
Class 和实例的关系以及原型链

1. 每个 class 都有显示原型 prototype【class 还有 constructor 和 __proto__】
2. 每个实例都有隐式原型__proto__【实例无 prototype，有 constructor】
3. 实例的__proto__指向对应 class 的 prototype





function Foo() { ...};
let f1 = new Foo();



这就是原型链

特殊之处：Function 的构造函数是自己；Function 的 prototype 和__proto__都指向 Function.prototype，Function 作为**对象**时，它的__proto__指向它**构造函数的 prototype**。Object 的构造函数也是 Function,同时一切对象又是 Object 创建。

闭包及其应用场景

在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

```
// 函数作为返回值
function create() {
    const a = 100
    return function () {
        console.log(a)
    }
```

```
}

const fn = create()
const a = 200
fn() // 100
```

```
// 函数作为参数被传递
function print(fn) {
    const a = 200
    fn()
}
const a = 100
function fn() {
    console.log(a)
}
print(fn) // 100
```

所有的**自由变量**的查找，**是在函数定义的地方，向上级作用域查找**，不是在执行的地方！！！

1. **作为私有仓库隐藏数据**
2. 集中管理，可以方便逻辑处理，修改时不需要到处修改某个变量
3. 让需要的变量在内存中存活，延续生命，不被收回
4. 点击对应的按钮序号等

```
// 闭包隐藏数据，只提供 API
function createCache() {
    const data = {} // 闭包中的数据，被隐藏，不被外界访问
    return {
        set: function (key, val) {
            data[key] = val
        },
        get (key) {
            return data[key]
        }
    }
}

const c = createCache()
c.set('a', 100)
c.set('a', 160)
console.log( c.get('a') )
```

this 指向总结

第一种是函数调用模式，当一个函数不是一个对象的属性时，直接作为函数来调用时，this 指向全局对象。【在严格模式环境中，默认绑定的 this 指向 undefined】

```
function fn() {
    console.log(this); //window
};

function fn1() {
    "use strict";
    console.log(this); //undefined
};

var name = '听风是风';

fn();
fn1();
```

```
// 如果在严格模式下调用不在严格模式中的函数，并不会影响 this 指向
var name = '听风是风';
function fn() {
    console.log(this); //window
    console.log(this.name); //听风是风
};

(function () {
    "use strict";
    fn();
}());
```

第二种是方法调用模式，如果一个函数作为一个对象的方法来调用时，this 指向这个对象。

```
//如果函数调用前存在多个对象，this 指向距离调用自己最近的对象
function fn() {
    console.log(this.name);
};
let obj = {
    name: '行星飞行',
    func: fn,
};
let obj1 = {
    name: '听风是风',
    o: obj
```

```
};
obj1.o.func() //行星飞行
```

```
//obj 对象虽然 obj1 的属性，但它两原型链并不相同，并不是父子关系，由于 obj 未提供 name 属性，所以是 undefined。
function fn() {
    console.log(this.name);
};
let obj = {
    func: fn,
};
let obj1 = {
    name: '听风是风',
    o: obj
};
obj1.o.func() //? ?
```

```
// 虽然 obj 对象并没有 name 属性，但顺着原型链，找到了产生自己的构造函数 Fn，由于 Fn 原型链存在 name 属性，所以输出时间跳跃了。
function Fn() {};
Fn.prototype.name = '时间跳跃';

function fn() {
    console.log(this.name);
};

let obj = new Fn();
obj.func = fn;

let obj1 = {
    name: '听风是风',
    o: obj
};
obj1.o.func() //?
```

第三种是构造器调用模式，如果一个函数用 new 调用时，函数执行前会新创建一个对象，this 指向这个新创建的对象。

第四种是 apply 、 call 和 bind 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。如果在使用 call 之类的方法改变 this 指向时，指向参数提供的是 null 或者 undefined，那么 this 将指向全局对象。另外，在 js API 中部分方法也内置了显式绑定，以 forEach 为例：

```
let obj1 = {
    name: '听风是风'
};
```

```
let obj2 = {
    name: '时间跳跃'
};
var name = '行星飞行';

function fn() {
    console.log(this.name);
};
fn.call(undefined); //行星飞行
fn.apply(null); //行星飞行
fn.bind(undefined)(); //行星飞行
let obj = {
    name: '听风是风'
};

[1, 2, 3].forEach(function () {
    console.log(this.name);//听风是风*3
}, obj);
```

这四种方式，使用构造器调用模式的优先级最高，然后是 apply、call 和 bind 调用模式，然后是方法调用模式，然后是函数调用模式。

this 的隐式丢失是怎么回事

在特定情况下会存在隐式绑定丢失的问题，最常见的就是作为**参数传递**以及**变量赋值**，先看参数传递：

```
var name = '行星飞行';
let obj = {
    name: '听风是风',
    fn: function () {
        console.log(this.name);
    }
};

function fn1(param) {
    param();
};
fn1(obj.fn);//行星飞行
```

将 obj.fn 也就是一个函数传递进 fn1 中执行，这里只是单纯传递了一个函数而已，**this 并没有跟函数绑在一起**，所以 this 丢失这里指向了 window。

```
var name = '行星飞行 2';
let obj = {
    name: '听风是风 2',
```

```
    fn: function () {
        console.log(this.name);
    }
};
let fn1 = obj.fn;
fn1(); //行星飞行 2
```

其实本质上与传参相同。

注意，隐式绑定丢失并不是都会指向全局对象，比如下面的例子：

```
var name = '行星飞行';
let obj = {
    name: '听风是风',
    fn: function () {
        console.log(this.name);
    }
};
let obj1 = {
    name: '时间跳跃'
}
obj1.fn = obj.fn;
obj1.fn(); //时间跳跃
```

虽然丢失了 obj 的隐式绑定，但是在赋值的过程中，又建立了新的隐式绑定，这里 this 就指向了对象 obj1。

JS 基本类型的装箱与拆箱

装箱的概念：把基本类型转化为相应的对象。而装箱又分为显式与隐式

在 JavaScript 中，基本类型是没有属性和方法的，但是为了便于操作基本类型的值，在调用基本类型的属性或方法时 JavaScript 会在后台隐式地将基本类型的值转换为对象

```
var s1  = new String("some text");
var s2 = s1.substring(2);
s1 = null;
```

JavaScript 也可以使用 Object 函数显式地将基本类型转换为包装类型：

```
Object(1)
```

拆箱：将引用类型对象转换为对应的值类型对象，它是通过引用类型的 valueOf()或者 toString()方法来实现的。

二者并存的情况下，在数值运算中，优先调用了 valueOf，字符串运算中，优先调用了 toString。

```
class A {
    valueOf() {
        return 2
    }
    toString() {
```

```
        return '哈哈哈'
    }
}
let a = new A()

console.log(String(a))  // '哈哈哈'    => (toString)
console.log(Number(a))  // 2          => (valueOf)
console.log(a + '22')   // '222'      => (valueOf)
console.log(a == 2)     // true       => (valueOf)
console.log(a === 2)    // false      => (严格等于不会触发隐式转换)
```

Symbol.toPrimitive 是一个内置的 Symbol 值，它是作为对象的函数值属性存在的，当一个对象转换为对应的原始值时，会调用此函数。

作用：同 valueOf()和 toString()一样，但是优先级要高于这两者；

new String 和 String 区别

```
let a1 ='a1'
let a2 = new String('a2')
let a3 = String('a3')
a1.name='zs'
a2.name='ls'
a3.name='ww'
console.log(a1)
console.log(a2)
console.log(a3)
console.log(a1.name)
console.log(a2.name)
console.log(a3.name)
console.log(typeof a1)
console.log(typeof a2)
console.log(typeof a3)
console.log(a2+'333')
console.log(a2.name)
// a1
// String {'a2', name: 'ls'}
// a3
// undefined
// ls
// undefined
// string
// object
// string
//a2333
```

```
//ls
```

Js 单线程详解

JavaScript 和 DOM 渲染共用同一个线程。在浏览器中，有一个称为**“主线程”**的线程，它负责处理 JavaScript 代码的执行、布局和渲染。当 JavaScript 代码执行时，它会阻塞主线程，这意味着浏览器无法同时执行其他任务，包括 DOM 渲染。因此，JavaScript 代码的执行速度对网页的性能有很大的影响。为了避免这种情况，可以使用 Web Workers 在后台运行 JavaScript 代码，或者使用异步编程技术来避免阻塞主线程。

浏览器渲染进程是多线程的，它包括下面几个线程：

- **GUI 渲染线程**
  - 负责渲染浏览器界面，解析 HTML，CSS，构建 DOM 树和 RenderObject 树，布局和绘制等。
    1. 解析 html 代码(HTML 代码本质是字符串)转化为浏览器认识的节点，生成 DOM 树，也就是 DOM Tree
    2. 解析 css，生成 CSSOM(CSS 规则树)
    3. 把 DOM Tree 和 CSSOM 结合，生成 Rendering Tree(渲染树)
  - 当界面需要重绘（Repaint）或由于某种操作引发回流(reflow)时，该线程就会执行
  - **GUI 渲染线程与 JS 引擎线程是互斥的**，当 JS 引擎执行时 GUI 线程会被挂起（相当于被冻结了），GUI 更新会被保存在一个队列中等到 JS 引擎空闲时立即被执行。
- **JS 引擎线程**
  - 也称为 **JS 内核**，负责处理 Javascript 脚本程序。（例如 V8 引擎）
  - JS 引擎线程负责解析 Javascript 脚本，运行代码。
  - JS 引擎一直等待着任务队列中任务的到来，然后加以处理，一个 Tab 页（renderer 进程）无论什么时候都只有一个 JS 线程在运行，JS 程序同样注意，GUI 渲染线程与 JS 引擎线程是互斥的，如果 JS 执行时间过长，会造成页面的渲染不连贯，页面渲染加载阻塞。
- **事件触发线程**
  - 归属于浏览器而不是 JS 引擎，用来控制事件循环（可以理解，JS 引擎自己都忙不过来，需浏览器另开线程协助）,用来控制事件循环，并且管理着一个事件队列(task queue)
  - 当 js 执行碰到事件绑定和一些异步操作(如 setTimeOut，也可来自浏览器内核的其他线程，如鼠标点击、AJAX 异步请求等)，会走事件触发线程将对应的事件添加到对应的线程中 (比如定时器操作，便把定时器事件添加到定时器线程)，等异步事件有了结果，便把他们的回调操作添加到事件队列，等待 js 引擎线程空闲时来处理。

    - 当对应的事件符合触发条件被触发时，该线程会把事件添加到待处理队列的队尾，等待 JS 引擎的处理
    - 注意，由于 JS 的单线程关系，所以这些待处理队列中的事件都得排队等待 JS 引擎处理（当 JS 引擎空闲时才会去执行）
- **定时触发器线程**
    - setInterval 与 setTimeout 所在线程
    - 浏览器定时计数器并不是由 JS 引擎计数的,（因为 JS 引擎是单线程的, 如果处于阻塞线程状态就会影响记计准确）。因此通过单独线程来计时并触发定时（计时完毕后，添加到事件队列中，等待 JS 引擎空闲后执行）
    - 注意，W3C 在 HTML 标准中规定，规定要求 setTimeout 中低于 4ms 的时间间隔算为 4ms。
- **异步 http 请求线程**
    - 在 XMLHttpRequest 在连接后是通过浏览器新开一个线程请求
    - 将检测到状态变更时，如果设置有回调函数，异步线程就产生状态变更事件，将这个回调再放入事件队列中。再由 JavaScript 引擎执行。

Dom 理解

文档对象模型（DOM）是 HTML 和 XML 文档的编程接口。Dom 的数据结构是一颗树。

Dom 节点操作

注意 appendChild

```
const div1 = document.getElementById('div1')
const div2 = document.getElementById('div2')

// 新建节点
const newP = document.createElement('p')
newP.innerHTML = 'this is newP'
// 插入节点
div1.appendChild(newP)

// 移动节点
const p1 = document.getElementById('p1')
div2.appendChild(p1)

// 获取父元素
console.log( p1.parentNode )
```

```
// 获取子元素列表
const div1ChildNodes = div1.childNodes
console.log( div1.childNodes )
const div1ChildNodesP =
Array.prototype.slice.call(div1.childNodes).filter(child => {
    if (child.nodeType === 1) {
        return true
    }
    return false
})
console.log('div1ChildNodesP', div1ChildNodesP)

div1.removeChild( div1ChildNodesP[0] )
```

Dom 性能优化

1. 缓存结果如 length 等
2. 文档碎片一次性添加

Bom 查看浏览器和地址栏和历史信息

navigator.userAgent：判断浏览器类型

location：获取地址栏 url 具体信息，有 location.href location.search location.host 等等可以具体到 url 的任何一部分

history：历史记录，history.pushState()或 history.replaceState()H5 新增，不会触发 popstate 事件

property 和 attribute 使用

Attribute：HTML 属性，书写在标签内的属性，使用 setAttribute()和 getAttribute()进行设置和获取。

Property：DOM 属性，html 标签对应的 DOM 节点属性，使用 .属性名 或者['属性名']进行设置和获取。

简单理解，Attribute 就是 dom 节点自带的属性，例如 html 中常用的 id、class、title、align 等;而 Property 是这个 DOM 元素作为对象，其附加的内容，例如 childNodes、firstChild 等。

另外，常用的 Attribute，例如 id、class 等，已经被作为 Property 附加到 DOM 对象上，可以和 Property 一样取值和赋值。但是自定义的 Attribute，就不会有这样的优待。

两者都可能造成重新渲染，优先使用 Property。

不管是修改 attribute 还是 property 都会影响到对方的属性值，但是**输入框的 input value 比较特殊例外！**

封装一个通用的事件监听函数

事件代理原理：事件冒泡

Element.matches()：如果元素被指定的选择器字符串选择，Element.matches() 方法返回 true；否则返回 false。

```
function bindEvent(elem, type, selector, fn) {
    if (fn == null) {
        fn = selector
        selector = null
    }
    elem.addEventListener(type, event => {
        const target = event.target
        if (selector) {
            // 代理绑定
            if (target.matches(selector)) {
                fn.call(target, event)
            }
        } else {
            // 普通绑定
            fn.call(target, event)
        }
    })
}

// 普通绑定
const btn1 = document.getElementById('btn1')
bindEvent(btn1, 'click', function (event) {
    // console.log(event.target) // 获取触发的元素
    event.preventDefault() // 阻止默认行为
    alert(this.innerHTML)
})

// 代理绑定
const div3 = document.getElementById('div3')
bindEvent(div3, 'click', 'a', function (event) {
    event.preventDefault()
    alert(this.innerHTML)
})
```

阻止事件冒泡和默认事件

event.stopPropagation()

event.stopImmediatePropagation():阻止事件冒泡并且阻止该元素上同事件类型的监听器被触发 【除了停止事件继续捕捉或冒泡传递外，也阻止事件被传入同元素中注册的其它相同事件类型监听器。】
event.preventDefalut():阻止默认事件

封装一个 ajax 函数

```
function ajax(url) {
    const p = new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest()
        xhr.open('GET', url, true)
        xhr.onreadystatechange = function () {
            if (xhr.readyState === 4) {
                if (xhr.status === 200) {
                    resolve(
                        JSON.parse(xhr.responseText)
                    )
                } else if (xhr.status === 404 || xhr.status === 500) {
                    reject(new Error('404 not found'))
                }
            }
        }
        xhr.send(null)
    })
    return p
}

const url = '/data/test.json'
ajax(url)
.then(res => console.log(res))
.catch(err => console.error(err))
```

跨域以及解决办法

同源策略/SOP（Same origin policy）是一种约定，是浏览器最核心基本的安全功能，缺少了同源策略，很容易受到 XSS、CSFR 等攻击。所谓同源是指"**协议+域名+端口**"三者相同。

解决跨域方案：

1. jsonp
2. cors
3. nginx 反向代理

4. websocket 协议

```
var script = document.createElement('script');
script.type = 'text/javascript';

// 传参一个回调函数名给后端，方便后端返回时执行这个在前端定义的回调函数
script.src = 'http://localhost:9000/t1?callback=handleCallback';
document.head.appendChild(script);

// 回调执行函数
function handleCallback(res) {
    console.log(res)
    alert(JSON.stringify(res));
}

//script.src =
'http://www.domain2.com:8080/login?user=admin&callback=handleCal
lback';
//handleCallback({"status": true, "user": "admin"})
```

Jquery 中的 jsonp 本质也是如此，与 ajax 无关。

cookie localStorage sessionStorage 存储优缺点

cookie 原本是前后端一些信息交互，后在特定情况下作为存储，容量小，且 api 操作不优雅，**每次请求会被携带**。通常后端也可以使用，并且结合 session 做一些存储和验证。

localStorage sessionStorage 可以存储 5M 以上，且 api get set 适合开发，一个永久存储一个会话期存储，且都不会被 http 请求携带。

从输入 URL 到渲染页面的整个过程

1. DNS 解析：域名=>IP 地址
2. 浏览器根据 IP 地址向服务器发起 http 请求
3. 服务器处理 http 请求，并将对应资源返回给浏览器
4. 根据 HTML 代码生成 DOM Tree，根据 CSS 代码生成 CSSOM（css 对象模型）
5. 将 DOM Tree 和 CSSOM 整合形成 Render Tree
6. 根据 Render Tree 渲染页面
7. 遇到 script 标签停止渲染，加载并执行 js，完成后再继续执行
8. 直至整个 Render 渲染完成

为什么 css 在页面 head，js 在 body 尾部

CSS 在加载过程中，不影响 HTML 的解析。但影响 HTML 渲染。

假如将 css 放在 body 的尾部，会产生一种情况。解析 html 生成 DOM 树，而后没有 css。所以直接生成渲染树，然后生成布局树渲染网页。直到解析到 css 时，生成 CSSOM。CSSOM 与 DOM 树合并生成渲染树，然后再一次生成布局树渲染网页。

这样会渲染树多生成了一次，并且页面也多渲染了一次。造成性能损耗。

JS 在加载时，HTML 会暂停解析。

如果 JS 在页面的首部，那么会造成一种情况。长时间加载 JS 导致 HTML 无法解析，页面长时间无法响应。所以 JS 放在 body 尾部。

或者给 JS 添加 defer/async

defer 和 async 都是异步加载 JS 的方法：

不同：defer 脚本加载完后立即执行。

async 脚本是 html 完全解析生成 DOM 树后立即执行。

如果多个 JS 脚本添加 defer 执行顺序是无序的。而 async 是按加载顺序执行的。

Window.onload 和 DOMContentLoaded（即 ready）区别

如非必要，优先使用 DOMContentLoaded（jquery），体验感更好

```
window.addEventListener('load', function () {
    // 页面的全部资源加载完才会执行，包括图片、视频等
})
document.addEventListener('DOMContentLoaded', function () {
    // DOM 渲染完即可执行，此时图片、视频还可能没有加载完
})
```

浅谈前端性能优化

优化原则：多使用内存，缓存，减少 cpu 计算量，减少网络加载耗时【空间换时间】

1. 减少资源体积：代码压缩
2. 减少访问次数：代码合并，SSR 服务端渲染（网页和数据一起渲染），缓存【如 webpack 打包文件对应的名称和 hash 值没变，会走 304】
3. 使用更快的网络：CDN
4. css 放在头部，js 放在底部
5. 尽早的执行 js，用 DOMContnetLoaded 触发
6. 懒加载
7. 对 DOM 查询进行缓存

8. 利用 fragment 文档碎片一次插入多条
9. 节流（拖拽） 防抖（输入框）
10. 根据项目需要，按需加载

前端防护 xss 和 xsrf 攻击

【Cross Site Script】跨站脚本攻击 恶意攻击者往 Web 页面里插入恶意 Script 代码，当用户浏览该页之时，嵌入其中 Web 里面的 Script 代码会被执行，从而达到恶意攻击用户的目的。可以直接安装 xss 对应插件。

```
function escape(str) {
  str = str.replace(/&/g, '&amp;')// h5 之后可以不做转义，且&符号转义要放在第一个否则对其他的有干扰
  str = str.replace(/</g, '&lt;')
  str = str.replace(/>/g, '&gt;')
  str = str.replace(/"/g, '&quto;')
  str = str.replace(/'/g, '&#39;')
  str = str.replace(/`/g, '&#96;')
  str = str.replace(/\//g, '&#x2F;')
  str = str.replace(/ /g, '&#39;')
  return str
}
escape('<script>alert(1)</script>')
//&lt;script&gt;alert(1)&lt;&#x2F;script&gt;
```

CSP: Content-Security-Policy 内容安全策略(白名单制度)

- 设置 HTTP 的 Content-Security-Policy 头部字段
- 设置网页的<meta>标签。

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title></title>
        <!-- 前端页面上设置，加了 jquery 就访问不了了-->
        <meta http-equiv="Content-Security-Policy"
content="form-action 'self';default-src 'self';">
    </head>
    <body>
        <script
src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-
2.1.1.min.js"></script>
        <script type="text/javascript">
```

```
            console.log($)
        </script>
    </body>
</html>
```

XSRF 攻击：【Cross Site Request Forgery】跨站点伪造请求

通过在访问用户被认为已经通过身份验证的 Web 应用程序的页面中包含恶意代码或链接来工作。

1. 在请求地址中添加 token+(验证码) 并验证：限制从别的网址登录的可能
2. 在 HTTP 头中自定义属性并验证
3. 验证 HTTP Referer 字段（ req.headers.referer ）
4. Get 请求不对数据进行修改
5. 不让第三方网站访问到用户 Cookie =>后台 cookie 配置设置 same-site 属性进行控制
6. CORS 是否必要[JSONP]，是否兼容老浏览器

数组的 API，哪些是纯函数

不改变原数组（无副作用），返回新数组。

```
// concat
const arr1 = arr.concat([50, 60, 70])
// map
const arr2 = arr.map(num => num * 10)
// filter
const arr3 = arr.filter(num => num > 25)
// slice
const arr4 = arr.slice()
```

非纯函数：

push pop shift unshift splice

forEach

some every

reduce

[10, 20, 30].map(parseInt)

```
const res = [10, 20, 30].map(parseInt)
console.log(res)//10 NaN NaN
// 拆解 parseInt 等同于下面的函数 return 简写
```

```
[10, 20, 30].map((num, index) => {
    return parseInt(num, index)
})
```
parseInt(string, radix):radix 从 2-36 表示进制

Vue

React

手写代码

手写 promise

then 和 catch 返回的是 promise 对象，如果 return 的不是 promise，也会转换成 promise，catch 是 then 的语法糖

如果是异步，需要考虑把回调函数**存储在数组**中，在合适的时候去遍历这个数组，因为可能存在同一个 promise 被多次调用这个 then，如 p1.then()调用了多次，所以使用数组

下面这个 promise 问题基本实现功能，存在一定的问题

```
class MyPromise {
    state = 'pending' // 状态，'pending' 'fulfilled' 'rejected'
    value = undefined // 成功后的值
    reason = undefined // 失败后的原因

    resolveCallbacks = [] // pending 状态下，存储成功的回调
    rejectCallbacks = [] // pending 状态下，存储失败的回调

    constructor(fn) {
        const resolveHandler = (value) => {
            // 加 setTimeout ，参考
https://coding.imooc.com/learn/questiondetail/257287.html
            setTimeout(() => {
                if (this.state === 'pending') {
                    this.state = 'fulfilled'
                    this.value = value
                    this.resolveCallbacks.forEach(fn =>
fn(value))
                }
            })
        }

        const rejectHandler = (reason) => {
            setTimeout(() => {
                if (this.state === 'pending') {
                    this.state = 'rejected'
                    this.reason = reason
```

```
                    this.rejectCallbacks.forEach(fn =>
fn(reason))
                }
            })
        }

        try {
            fn(resolveHandler, rejectHandler)
        } catch (err) {
            rejectHandler(err)
        }
    }

    then(fn1, fn2) {
        fn1 = typeof fn1 === 'function' ? fn1 : (v) => v
        fn2 = typeof fn2 === 'function' ? fn2 : (e) => e

        if (this.state === 'pending') {
            const p1 = new MyPromise((resolve, reject) => {
                this.resolveCallbacks.push(() => {
                    try {
                        const newValue = fn1(this.value)
                        resolve(newValue)
                    } catch (err) {
                        reject(err)
                    }
                })

                this.rejectCallbacks.push(() => {
                    try {
                        const newReason = fn2(this.reason)
                        reject(newReason)
                    } catch (err) {
                        reject(err)
                    }
                })
            })
            return p1
        }

        if (this.state === 'fulfilled') {
            const p1 = new MyPromise((resolve, reject) => {
                try {
                    const newValue = fn1(this.value)
```

```
                    resolve(newValue)
                } catch (err) {
                    reject(err)
                }
            })
            return p1
        }

        if (this.state === 'rejected') {
            const p1 = new MyPromise((resolve, reject) => {
                try {
                    const newReason = fn2(this.reason)
                    reject(newReason)
                } catch (err) {
                    reject(err)
                }
            })
            return p1
        }
    }

    // 就是 then 的一个语法糖，简单模式
    catch(fn) {
        return this.then(null, fn)
    }
}

MyPromise.resolve = function (value) {
    return new MyPromise((resolve, reject) => resolve(value))
}
MyPromise.reject = function (reason) {
    return new MyPromise((resolve, reject) => reject(reason))
}

MyPromise.all = function (promiseList = []) {
    const p1 = new MyPromise((resolve, reject) => {
        const result = [] // 存储 promiseList 所有的结果
        const length = promiseList.length
        let resolvedCount = 0

        promiseList.forEach(p => {
            p.then(data => {
                result.push(data)
```

```
                // resolvedCount 必须在 then 里面做 ++
                // 不能用 index
                resolvedCount++
                if (resolvedCount === length) {
                    // 已经遍历到了最后一个 promise
                    resolve(result)
                }
            }).catch(err => {
                reject(err)
            })
        })
    })
    return p1
}

MyPromise.race = function (promiseList = []) {
    let resolved = false // 标记
    const p1 = new Promise((resolve, reject) => {
        promiseList.forEach(p => {
            p.then(data => {
                if (!resolved) {
                    resolve(data)
                    resolved = true
                }
            }).catch((err) => {
                reject(err)
            })
        })
    })
    return p1
}
```

手写深拷贝

最重要的是**递归**那步

```
/**
 * 深拷贝
 */

const obj1 = {
    age: 20,
    name: 'xxx',
    address: {
```

```
        city: 'beijing'
    },
    arr: ['a', 'b', 'c']
}

const obj2 = deepClone(obj1)
obj2.address.city = 'shanghai'
obj2.arr[0] = 'a1'
console.log(obj1.address.city)
console.log(obj1.arr[0])

/**
 * 深拷贝
 * @param {Object} obj 要拷贝的对象
 */
function deepClone(obj = {}) {
    if (typeof obj !== 'object' || obj == null) {
        // obj 是 null ，或者不是对象和数组，直接返回
        return obj
    }

    // 初始化返回结果
    let result
    if (obj instanceof Array) {
        result = []
    } else {
        result = {}
    }

    for (let key in obj) {
        // 保证 key 不是原型的属性
        if (obj.hasOwnProperty(key)) {
            // 递归调用！！！
            result[key] = deepClone(obj[key])
        }
    }

    // 返回结果
    return result
}
```

手写 jquery，考虑插件和扩展性

```
class jQuery {
        constructor(selector) {
            const result = document.querySelectorAll(selector)
            const length = result.length
            for (let i = 0; i < length; i++) {
                this[i] = result[i]
            }
            this.length = length
            this.selector = selector
        }
        get(index) {
            return this[index]
        }
        each(fn) {
            for (let i = 0; i < this.length; i++) {
                const elem = this[i]
                fn(elem)
            }
        }
        on(type, fn) {
            return this.each(elem => {
                elem.addEventListener(type, fn, false)
            })
        }
        // 扩展很多 DOM API
    }

    // 插件
    jQuery.prototype.dialog = function (info) {
        alert(info)
    }

    // “造轮子”
    class myJQuery extends jQuery {
        constructor(selector) {
            super(selector)
        }
        // 扩展自己的方法
        addClass(className) {

        }
```

```
        style(data) {

        }
    }

    // const $p = new jQuery('p')
    // $p.get(1)
    // $p.each((elem) => console.log(elem.nodeName))
    // $p.on('click', () => alert('clicked'))
```

手写 bind/call 函数

call/apply/bind 的核心理念：借。借助已实现的方法，改变方法中数据的 this 指向，减少重复代码，节省内存。（call 隐式绑定的方式调用函数）

```
// 模拟 bind
Function.prototype.bind1 = function () {
    // 将参数拆解为数组
    const args = Array.prototype.slice.call(arguments)

    // 获取 this（数组第一项）
    const t = args.shift()

    // fn1.bind(...) 中的 fn1
    const self = this

    // 返回一个函数
    return function () {
        return self.apply(t, args)
    }
}

function fn1(a, b, c) {
    console.log('this', this)
    console.log(a, b, c)
    return 'this is fn1'
}

const fn2 = fn1.bind1({x: 100}, 10, 20, 30)
const res = fn2()
console.log(res)
Function.prototype.myCall = function(context) {
      console.log(this) //fn a()
```

```
      if (typeof this !== 'function') {
      throw new TypeError('Error')
      }
      context = context || window
      // console.log(context)//obj
      context.fn = this

      // 防止覆盖掉原有属性可用 symbol
      // const key = Symbol()
      // context[key] = this
      console.log(typeof context.fn)//function

      const args = [...arguments].slice(1)
      console.log(args)
      //通过隐式绑定的方式调用函数
      const result = context.fn(...args)
       //删除添加的属性
      delete context.fn
       //返回函数调用的返回值
      return result
}

let obj ={
  name:333
}

function a(){
  console.log(this.name)
}
a.myCall(obj)
```

手写 debounce 防抖函数

闭包和定时器应用，防抖每次触发清除定时器

```
const input1 = document.getElementById('input1')
// 防抖
function debounce(fn, delay = 500) {
    // timer 是闭包中的
    let timer = null

    return function () {
        if (timer) {
            clearTimeout(timer)
```

```
        }
        timer = setTimeout(() => {
            fn.apply(this, arguments)
            timer = null
        }, delay)
    }
}

input1.addEventListener('keyup', debounce(function (e) {
    console.log(e.target)
    console.log(input1.value)
}, 600))
```

手写 throttle 节流函数

节流每次触发 timer 不为 null 直接返回

```
// 节流
function throttle(fn, delay = 100) {
    let timer = null

    return function () {
        if (timer) {
            return
        }
        timer = setTimeout(() => {
            fn.apply(this, arguments)
            timer = null
        }, delay)
    }
}
```

手写 lodash 中深度对比 isEqual

核心递归

```
// 判断是否是对象或数组
function isObject(obj) {
    return typeof obj === 'object' && obj !== null
}
// 全相等（深度）
function isEqual(obj1, obj2) {
    if (!isObject(obj1) || !isObject(obj2)) {
```

```
        // 值类型（注意，参与 equal 的一般不会是函数）
        return obj1 === obj2
    }
    if (obj1 === obj2) {
        return true
    }
    // 两个都是对象或数组，而且不相等
    // 1. 先取出 obj1 和 obj2 的 keys ，比较个数
    const obj1Keys = Object.keys(obj1)
    const obj2Keys = Object.keys(obj2)
    if (obj1Keys.length !== obj2Keys.length) {
        return false
    }
    // 2. 以 obj1 为基准，和 obj2 一次递归比较
    for (let key in obj1) {
        // 比较当前 key 的 val —— 递归！！！
        const res = isEqual(obj1[key], obj2[key])
        if (!res) {
            return false
        }
    }
    // 3. 全相等
    return true
}

// 测试
const obj1 = {
    a: 100,
    b: {
        x: 100,
        y: 200
    }
}
const obj2 = {
    a: 100,
    b: {
        x: 100,
        y: 200
    }
}
// console.log( obj1 === obj2 )
console.log( isEqual(obj1, obj2) )
```

Http

http 状态码

1: 服务端接受消息
2: 成功
3: 重定向之类
4: 一般客户端有问题
5: 服务端问题

状态码	状态码英文名称	中文描述
100	Continue	继续。客户端应继续其请求
200	OK	请求成功
204	No Content	无内容。服务器成功处理，但未返回内容。在未更新网页的情况下，可确保浏览器继续显示当前文档
301	-	永久重定向，结合返回的 Location，浏览器自己处理
302	-	临时重定向，其他同上
304	Not Modified	所请求的资源未修改，缓存
400	Bad Request	客户端请求的语法错误，服务器无法理解
401	Unauthorized	请求要求用户的身份认证
403	Forbidden	服务器理解请求客户端的请求，但是拒绝执行此请求
404	Not Found	服务器无法根据客户端的请求找到资源
405	Method Not	客户端请求中的方法被禁止，可能是请求方

状态码	状态码英文名称	中文描述
	Allowed	法错误
413	Request Entity Too Large	由于请求的实体过大，服务器无法处理，因此拒绝请求。【后端可配置放大实体】
415	Unsupported Media Type	服务器无法处理请求附带的媒体格式
500	Internal Server Error	服务器内部错误，无法完成请求
501	Not Implemented	服务器不支持请求的功能，无法完成请求
502	Bad Gateway	作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应
504	Gateway Time-out	充当网关或代理的服务器，未及时从远端服务器获取请求，网关超时。

RESTful API 和传统 API 设计区别

传统 api：把每个 url 当做一个功能。

Restful API：**把每个 url 当做一个唯一的资源。**

传统 api 设计一般只用到 post/get，url 中需要参数形式，且对同一个数据的增删改查 url 名字明显不同。

Restful API 的 url 同一个数据做增删改查 url 基本一致，体现在方法名上，post 新建 patch/put 更新 delete 删除 get 查询。

常见的 http headers

Request Header	规定
Accept	浏览器端接受的格式。
Accept-Encoding:	浏览器端接收的编码方式。
Accept-Language	浏览器端接受的语言，用于服务端判断多语言。
Cache-Control	控制缓存的时效性。
Connection	连接方式，如果是keep-alive，且服务端支持，则会复用连接。
Host	HTTP访问使用的域名。
If-Modified-Since	上次访问时的更改时间，如果服务端认为此时间后自己没有更新，则会给出304响应。
If-None-Match	次访问时使用的E-Tag，通常是页面的信息摘要，这个比更改时间更准确一些。
User-Agent	客户端标识，因为一些历史原因，这是一笔糊涂账，多数浏览器的这个字段都十分复杂，区别十分微妙。
Cookie	客户端存储的cookie字符串。

Content-type：告诉服务端传递的内容类型和一些其他信息。

```
axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded;charset=UTF-8';
```

Response Header	规定
Cache-Control	缓存控制，用于通知各级缓存保存的时间，例如max-age=0，表示不要缓存。
Connection	连接类型，Keep-Alive表示复用连接。
Content-Encoding	内容编码方式，通常是gzip。
Content-Length	内容的长度，有利于浏览器判断内容是否已经结束。
Content-Type	内容类型，所有请求网页的都是text/html。
Date	当前的服务器日期。
ETag	页面的信息摘要，用于判断是否需要重新到服务端取回页面。
Expires	过期时间，用于判断下次请求是否需要到服务端取回页面。
Keep-Alive	保持连接不断时需要的一些信息，如timeout=5, max=100。
Last-Modified	页面上次修改的时间。
Server	服务端软件的类型。
Set-Cookie	设置cookie，可以存在多个。
Via	服务端的请求链路，对一些调试场景至关重要的一个头。

http 缓存和刷新

将网络资源存储在本地，等待下次请求该资源时，如果命中就不需要到服务器重新请求该资源，直接在本地读取该资源。

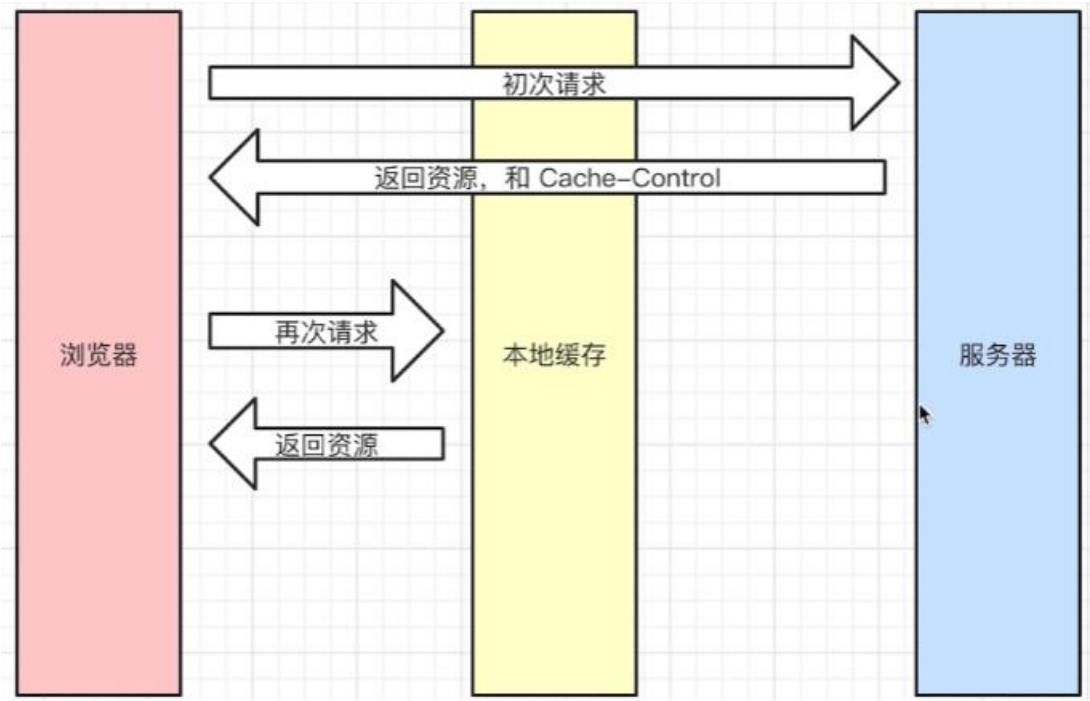
强缓存：1.cache-control 的某些属性值

max-age：最大缓存时间

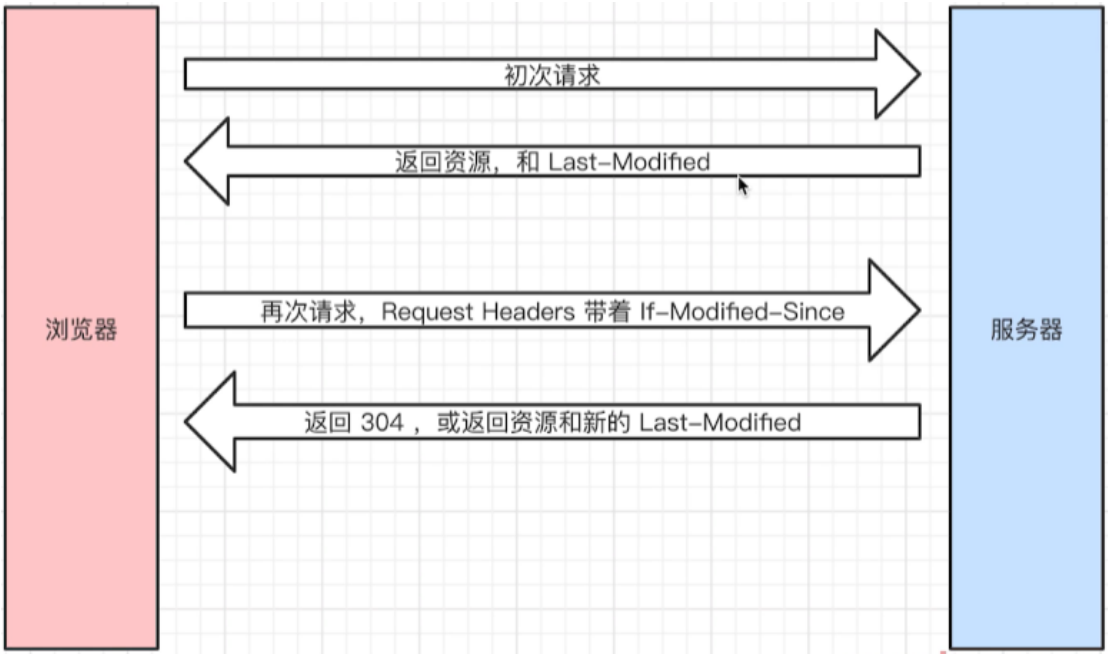
no-cache：不用本地强制缓存

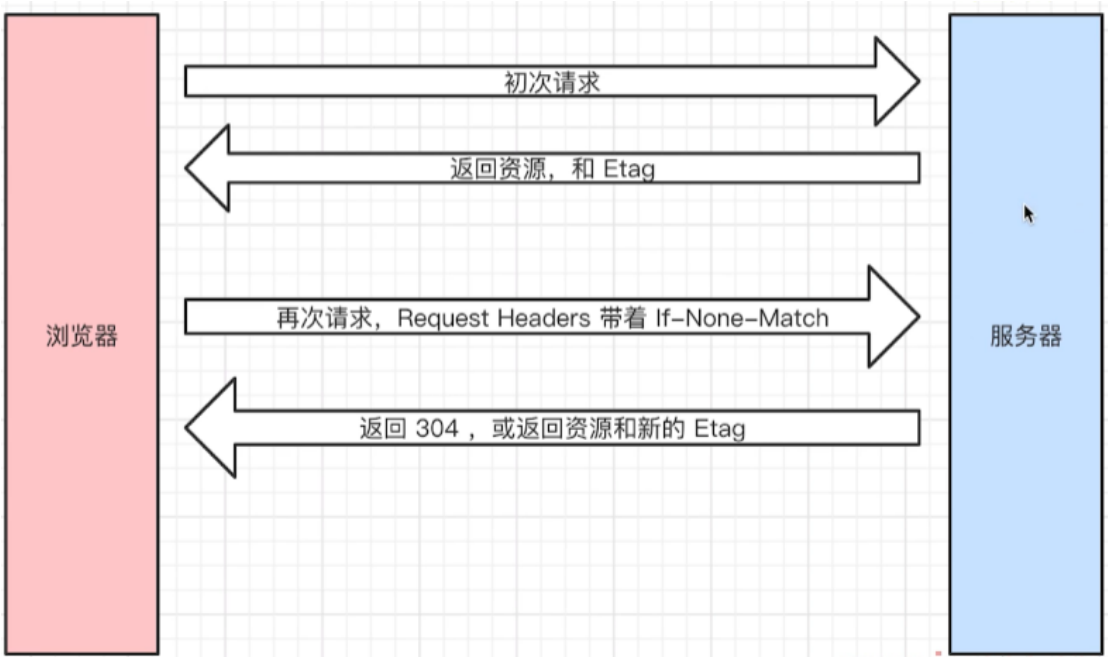
no-store：不用本地缓存，也不用服务端的一些缓存措施

2.expires（不推荐，优先级也低于 cache-control）



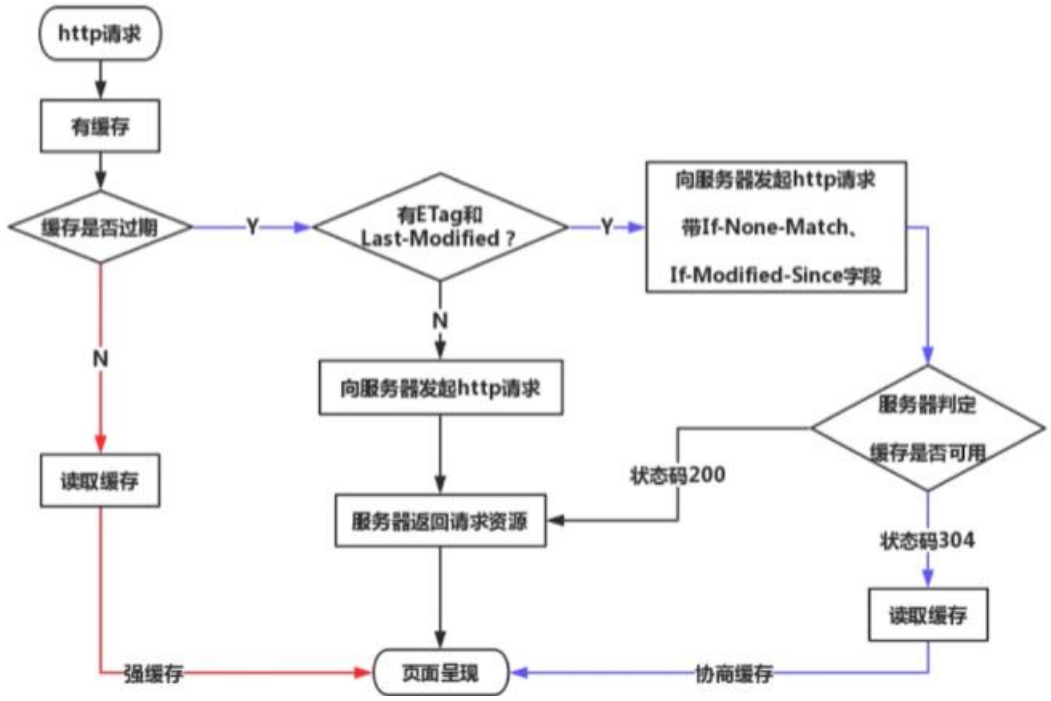
协商缓存：服务端缓存策略，通过【Last-Modified，If-Modified-Since】（最后修改时间）和【ETag、If-None-Match】（类似唯一指纹）这两对 Header 分别管理





Last-Modified 和 ETag 可同时存在，一般情况 ETag 更准确

总结



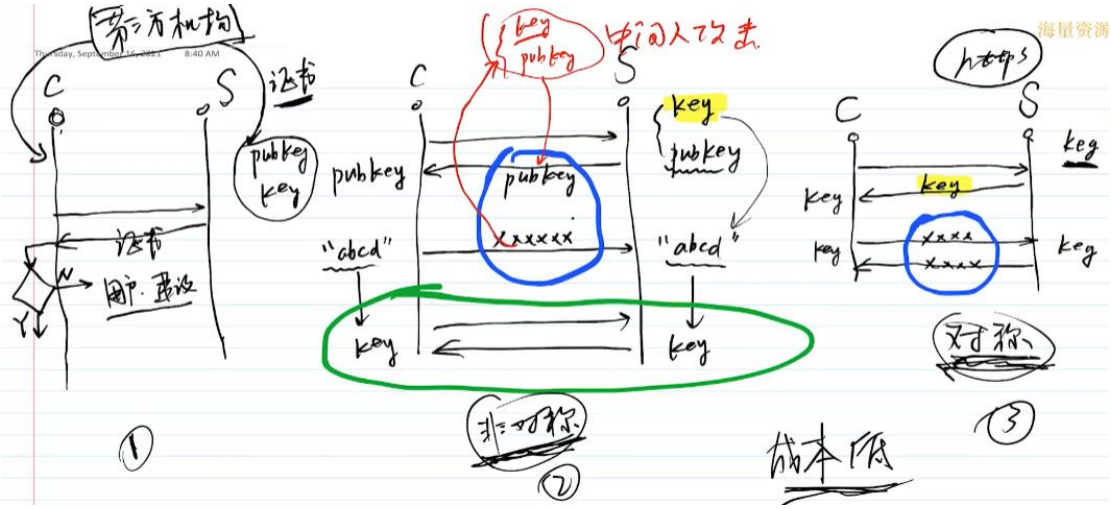
刷新操作

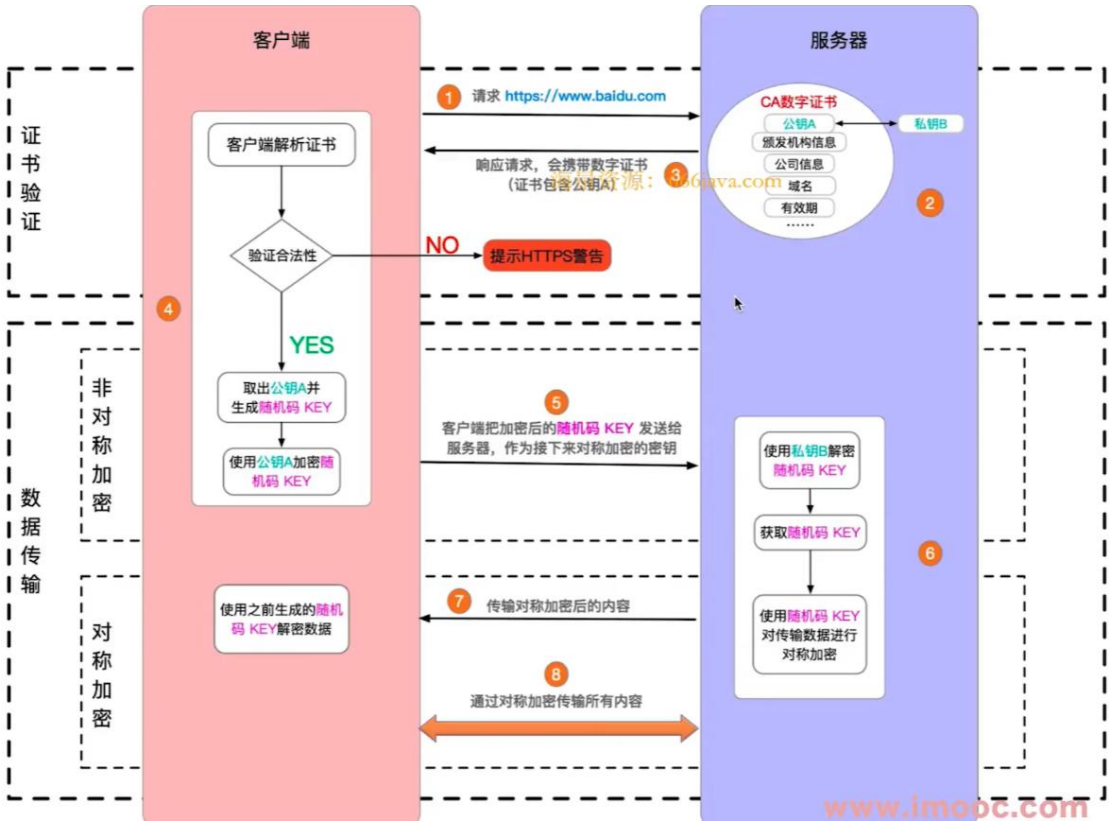
正常操作（浏览器切换页面）：强缓存和协商缓存都生效
F5 刷新：强缓存失效，协商缓存有效
Ctrl+F5:两者都失效

https 加密方式和证书

https 采用的是：结合**对称加密+非对称加密**这两种方式，可以用非对称加密的方式来传输对称加密过程中的密钥，之后可以采取对称加密的方式来传输数据了。

证书为了避免中间人攻击。





加密算法有哪几种

1. **对称加密算法**也叫共享密钥加密算法、单密钥加密算法。采用单密钥的加密方法，同一个密钥可以同时用作信息的加密和解密，即解密算法为加密算法的逆算法。因此在知道了加密算法后也就知道了解密算法。对称加密算法有 DES、3DES、AES 等。

2. **非对称加密算法**又叫公开密钥算法。采用的是公钥和私钥相结合的加密方法。公钥和私钥是两个完全不同的密钥，一个用于加密，一个用于解密。同

时这两个密钥在数学上是关联的。即解密算法不是加密算法的逆算法，因此在知道了加密算法后也无法知道解密算法，保证了安全性。有 RSA、ECC 等。

3. **数字摘要算法**又称哈希算法、散列算法，是一种单向算法，它通过对数据内容进行散列得到一个固定长度的密文信息（信息是任意长度，而摘要是定长）。即用户可以通过哈希算法对目标信息生成一段特定长度的唯一的 Hash 值，却不能通过这个 Hash 值重新获得目标信息。该算法不可逆。哈希算法有 MD5、SHA-1、SHA-256 等。

Node

构建工具

其他工具

Git 常用命令

1、第一次初始化

- git init
  git remote add origin git@github.com:帐号名/仓库名.git
  git pull origin master
  git push origin master # -f 强推
- git clone git@github.com:git 帐号名/仓库名.git

2、工作基本操作

- git checkout master 切到主分支
- git fetch origin 获取最新变更
- git checkout -b dev origin/master 基于主分支创建 dev 分支

3、查看仓库当前状态
git status

4、文件相关操作
将文件添加到仓库：

- git add 文件名 将工作区的某个文件添加到暂存区
- git add . 将当前工作区的所有文件都加入暂存区

将暂存区文件提交到本地仓库：

- git commit -m“提交说明”将暂存区内容提交到本地仓库

比较文件异同

- git diff 工作区与暂存区的差异
- git diff 分支名 工作区与某分支的差异，远程分支这样写：remotes/origin/分支名

5、查看历史记录

- git log 查看所有 commit 记录(SHA-A 校验和，作者名称，邮箱，提交时间，提交说明)
- git log fileName 查看某文件的修改记录

6、代码回滚

- git reset HEAD^ 恢复成上次提交的版本
- git reset HEAD^^ 恢复成上上次提交的版本，就是多个^，以此类推或用~次数

7、版本库相关操作

- 删除版本库文件：git rm 文件名
- **版本库里的版本替换工作区的版本：git checkout – test.txt**

8、远程仓库相关操作

同步远程仓库：git push -u origin master

拉取远程分支到本地仓库：

- git checkout -b 本地分支 远程分支 # 会在本地新建分支，并自动切换到该分支
- **git fetch origin 远程分支:本地分支 # 会在本地新建分支，但不会自动切换，还需 checkout**

9、分支相关操作

创建分支：git checkout -b dev -b 表示创建并切换分支

上面一条命令相当于一面的二条：

git branch dev 创建分支

git checkout dev 切换分支

10、git 相关配置

安装完 Git 后第一件要做的事，设置用户信息(global 可换成 local 在单独项目生效)：

- git config -global user.name “用户名” # 设置用户名
- git config -global user.email “用户邮箱” #设置邮箱
- git config -global user.name # 查看用户名是否配置成功
- git config -global user.email # 查看邮箱是否配置

11、撤消某次提交

- git revert HEAD # 撤销最近的一个提交
- git revert 版本号 # 撤销某次 commit

linux 常见命令

1.cd 跳转目录
```bash
cd /usr/local
```
2.ls 查看当前目录文件
3. mkdir 新建文件夹
```bash
mkdir xxx
```
4.创建级联目录
```bash
mkdir /a/b/c
```
5.创建文件 demo.txt
```bash
touch demo.txt
```
6.vi 编辑文件和保存
编辑文件用 vi 打开后， 点击 i 按钮开始编辑
```bash
vi /etc/bubby.txt
```
编辑完后，按 ESC 退出 insert 编辑状态，输入:wq 保存退出，如果不想保存，则是 esc 后输入 :q!
7.cat 查看文件
```bash
cat demo.txt
```
8.echo 追加内容/覆盖内容
+ `>>` 追加
+ `>` 覆盖
+ echo 后面内容加不加引号都可以执行
```bash
echo 'xxx' >> demo.txt
```
```bash
echo helloword > demo.txt
```
9.rm 删除文件和文件夹
```bash
rm demo.txt
```

```
+ 删除文件夹
```bash
rm -r tt
```
```bash
rm xxx.zip
```
+ 强制删除文件夹
```bash
rm -rf a
```
10.搜索 grep
+ ls 查看当前目录下所有的文件，过滤| => grep 查看 txt 后缀的文件
```
ls | grep *.txt
```
```