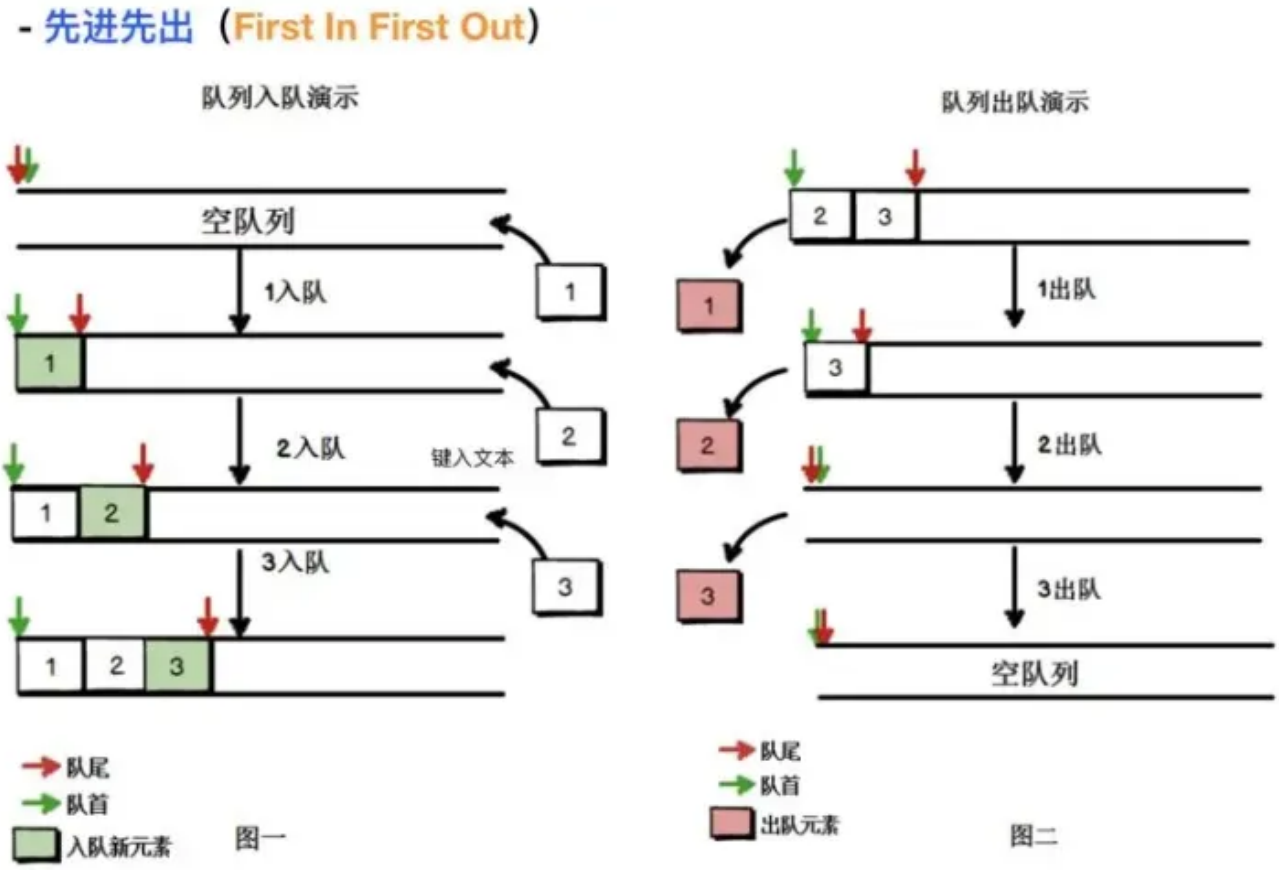
最新添加的元素必须排在队列的末尾



2.5. 链表（Linked List）

链表也是一种列表，已经设计了数组，为什么还需要链表呢?
JavaScript中数组的主要问题时，它们被实现成了对象，
与其他语言（比如C++和Java）的数组相对，效率很低。
如果你发现数组在实际使用时很慢，就可以考虑使用链表来代替它。

使用条件:
链表几乎可以用在任何可以使用一维数组的情况中。
如果需要随机访问，数组仍然是更好的选择。

2.6. 字典

字典是一种以键–值对存储数据的数据结构，js中的Object类就是以字典的形式设计的。JavaScript可以通过实现字典类，让这种字典类型的对象使用起来更加简单，字典可以实现对象拥有的常见功能，并相应拓展自己想要的功能，而对象在JavaScript编写中随处可见，所以字典的作用也异常明显了。

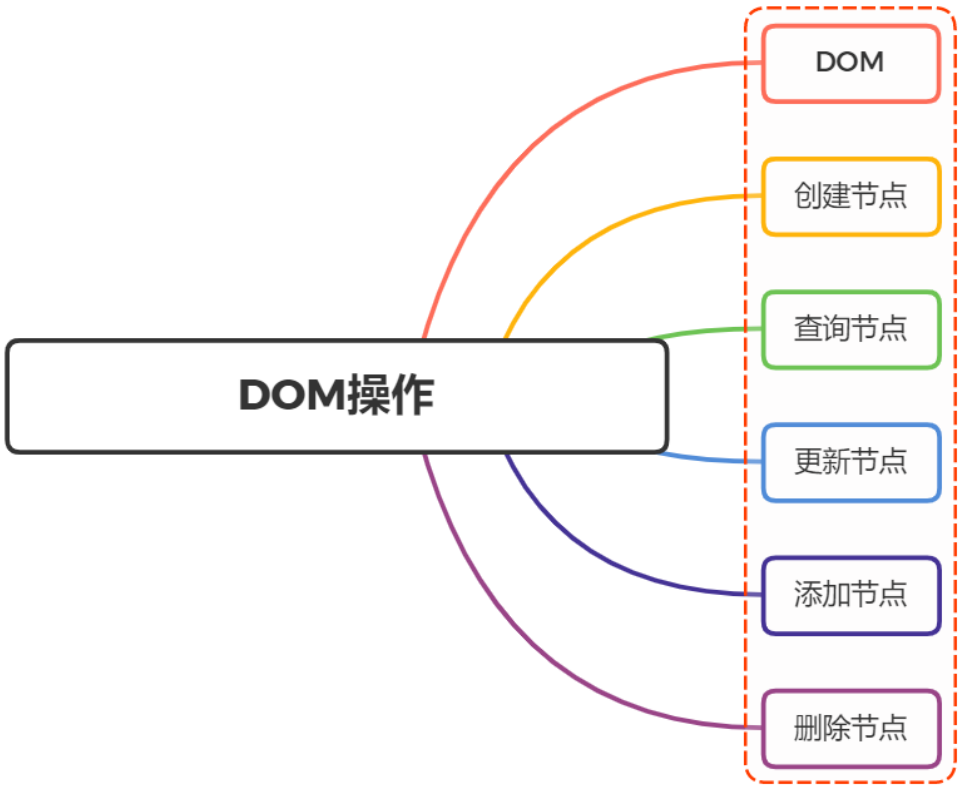
2.7. 散列表

也称为哈希表，特点是在散列表上插入、删除和取用数据都非常快。
为什么要设计这种数据结构呢?
用数组或链表存储数据，如果想要找到其中一个数据，需要从头进行遍历，因为不知道这个数据存储到了数组的哪个位置。

散列表在JavaScript中可以基础数组去进行设计。
数组的长度是预先设定的，所有元素根据和该元素对应的键，保存在数组的特定位置，这里的键和对象的键是类型的概念。
使用散列表存储数组时，通过一个散列函数将键映射为一个数字，这个数字的范围是0到散列表的长度。

即使使用一个高效的散列函数，依然存在将两个键映射为同一个值得可能，这种现象叫做碰撞。常见碰撞的处理方法有：开链法和线性探测法（具体概念有兴趣的可以网上自信了解）
使用条件：
可以用于数据的插入、删除和取用，不适用于查找数据

3. DOM常见的操作有哪些?



3.1. DOM

文档对象模型 (DOM) 是 `HTML` 和 `XML` 文档的编程接口

它提供了对文档的结构化的表述，并定义了一种方式可以使从程序中对该结构进行访问，从而改变文档的结构，样式和内容

任何 `HTML` 或 `XML` 文档都可以用 `DOM` 表示为一个由节点构成的层级结构

节点分很多类型，每种类型对应着文档中不同的信息和（或）标记，也都有自己不同的特性、数据和方法，而且与其他类型有某种关系，如下所示：

HTML | 复制代码

```html
<html>
    <head>
        <title>Page</title>
    </head>
    <body>
        <p>Hello World!</p >
    </body>
</html>
```

`DOM` 像原子包含着亚原子微粒那样，也有很多类型的 `DOM` 节点包含着其他类型的节点。接下来我们先看看其中的三种：

HTML | 复制代码

```html
<div>
    <p title="title">
        content
    </p >
</div>
```

上述结构中， `div` 、 `p` 就是元素节点， `content` 就是文本节点， `title` 就是属性节点

3.2. 操作

日常前端开发，我们都离不开 `DOM` 操作

在以前，我们使用 `Jquery` ， `zepto` 等库来操作 `DOM` ，之后在 `vue` ， `Angular` ， `React` 等框架出现后，我们通过操作数据来控制 `DOM` （绝大多数时候），越来越少的去直接操作 `DOM`

但这并不代表原生操作不重要。相反， `DOM` 操作才能有助于我们理解框架深层的内容

下面就来分析 `DOM` 常见的操作，主要分为：

- 创建节点

- 查询节点
- 更新节点
- 添加节点
- 删除节点

3.2.1. 创建节点

3.2.1.1. createElement

创建新元素，接受一个参数，即要创建元素的标签名

```javascript
const divEl = document.createElement("div");
```

3.2.1.2. createTextNode

创建一个文本节点

```javascript
const textEl = document.createTextNode("content");
```

3.2.1.3. createDocumentFragment

用来创建一个文档碎片，它表示一种轻量级的文档，主要是用来存储临时节点，然后把文档碎片的内容一次性添加到 `DOM` 中

```javascript
const fragment = document.createDocumentFragment();
```

当请求把一个 `DocumentFragment` 节点插入文档树时，插入的不是 `DocumentFragment` 自身，而是它的所有子孙节点

3.2.1.4. createAttribute

创建属性节点，可以是自定义属性

```javascript
const dataAttribute = document.createAttribute('custom');
consle.log(dataAttribute);
```

3.2.2. 获取节点

3.2.2.1. querySelector

传入任何有效的 `css` 选择器，即可选中单个 `DOM` 元素（首个）：

```javascript
document.querySelector('.element')
document.querySelector('#element')
document.querySelector('div')
document.querySelector('[name="username"]')
document.querySelector('div + p > span')
```

如果页面上没有指定的元素时，返回 `null`

3.2.2.2. querySelectorAll

返回一个包含节点子树内所有与之相匹配的 `Element` 节点列表，如果没有相匹配的，则返回一个空节点列表

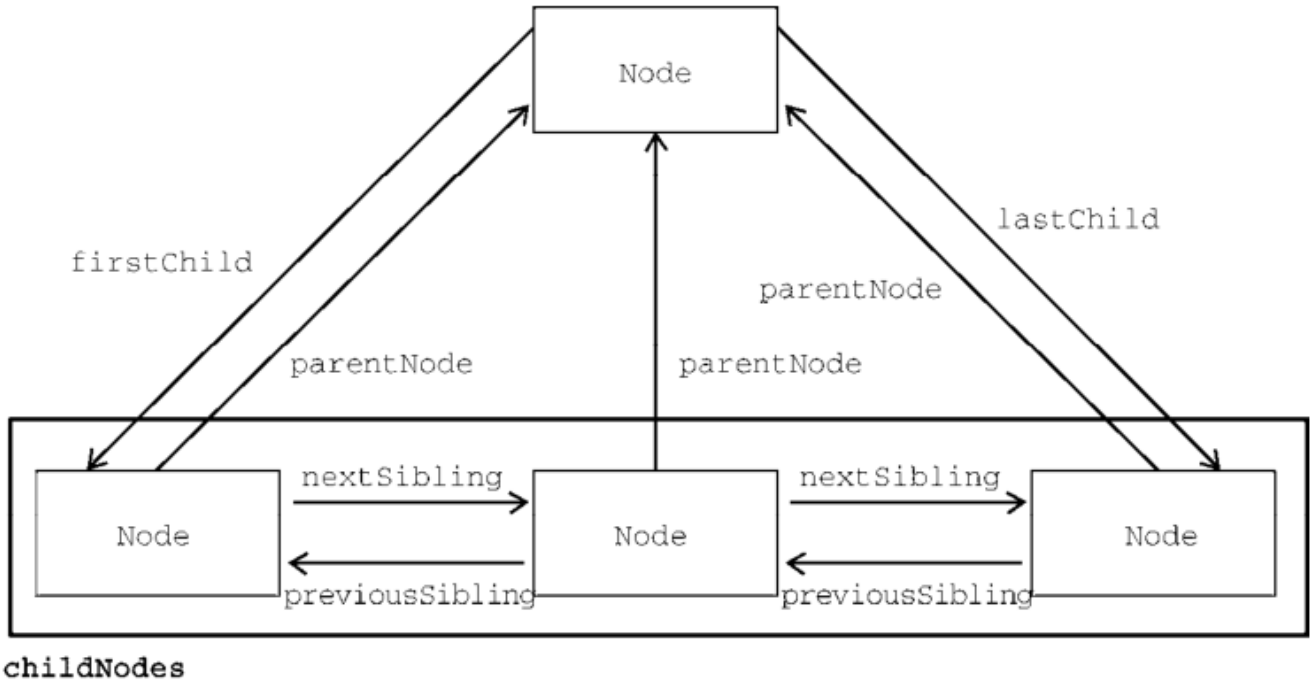
```javascript
const notLive = document.querySelectorAll("p");
```

需要注意的是，该方法返回的是一个 `NodeList` 的静态实例，它是一个静态的“快照”，而非“实时”的查询

关于获取 `DOM` 元素的方法还有如下，就不一一述说

```javascript
document.getElementById('id属性值');返回拥有指定id的对象的引用
document.getElementsByClassName('class属性值');返回拥有指定class的对象集合
document.getElementsByTagName('标签名');返回拥有指定标签名的对象集合
document.getElementsByName('name属性值'); 返回拥有指定名称的对象结合
document/element.querySelector('CSS选择器');  仅返回第一个匹配的元素
document/element.querySelectorAll('CSS选择器');    返回所有匹配的元素
document.documentElement;  获取页面中的HTML标签
document.body; 获取页面中的BODY标签
document.all[''];  获取页面中的所有元素节点的对象集合型
```

除此之外，每个 `DOM` 元素还有 `parentNode` 、 `childNodes` 、 `firstChild` 、 `lastChild` 、 `nextSibling` 、 `previousSibling` 属性，关系图如下图所示



3.2.3. 更新节点

3.2.3.1. innerHTML

不但可以修改一个 `DOM` 节点的文本内容，还可以直接通过 `HTML` 片段修改 `DOM` 节点内部的子树

```javascript
// 获取<p id="p">...</p >
var p = document.getElementById('p');
// 设置文本为abc:
p.innerHTML = 'ABC'; // <p id="p">ABC</p >
// 设置HTML:
p.innerHTML = 'ABC <span style="color:red">RED</span> XYZ';
// <p>...</p >的内部结构已修改
```

3.2.3.2. innerText、textContent

自动对字符串进行 `HTML` 编码，保证无法设置任何 `HTML` 标签

```
// 获取<p id="p-id">...</p >
var p = document.getElementById('p-id');
// 设置文本:
p.innerText = '<script>alert("Hi")</script>';
// HTML被自动编码，无法设置一个<script>节点:
// <p id="p-id">&lt;script&gt;alert("Hi")&lt;/script&gt;</p >
```

两者的区别在于读取属性时， `innerText` 不返回隐藏元素的文本，而 `textContent` 返回所有文本

3.2.3.3. style

`DOM` 节点的 `style` 属性对应所有的 `CSS` ，可以直接获取或设置。遇到 `-` 需要转化为驼峰命名

```javascript
// 获取<p id="p-id">...</p >
const p = document.getElementById('p-id');
// 设置CSS:
p.style.color = '#ff0000';
p.style.fontSize = '20px'; // 驼峰命名
p.style.paddingTop = '2em';
```

3.2.4. 添加节点

3.2.4.1. innerHTML

如果这个DOM节点是空的，例如，`<div></div>`，那么，直接使用`innerHTML = '<span>child</span>'`就可以修改`DOM`节点的内容，相当于添加了新的`DOM`节点

如果这个DOM节点不是空的，那就不能这么做，因为`innerHTML`会直接替换掉原来的所有子节点

3.2.4.2. appendChild

把一个子节点添加到父节点的最后一个子节点

举个例子

JavaScript 复制代码

```
<!-- HTML结构 -->
<p id="js">JavaScript</p >
<div id="list">
    <p id="java">Java</p >
    <p id="python">Python</p >
    <p id="scheme">Scheme</p >
</div>
```

添加一个`p`元素

JavaScript 复制代码

```
const js = document.getElementById('js')
js.innerHTML = "JavaScript"
const list = document.getElementById('list');
list.appendChild(js);
```

现在`HTML`结构变成了下面

JavaScript 复制代码

```
<!-- HTML结构 -->
<div id="list">
    <p id="java">Java</p >
    <p id="python">Python</p >
    <p id="scheme">Scheme</p >
    <p id="js">JavaScript</p >  <!-- 添加元素 -->
</div>
```

上述代码中，我们是获取`DOM`元素后再进行添加操作，这个`js`节点是已经存在当前文档树中，因此这个节点首先会从原先的位置删除，再插入到新的位置

如果动态添加新的节点，则先创建一个新的节点，然后插入到指定的位置

```javascript
const list = document.getElementById('list'),
const haskell = document.createElement('p');
haskell.id = 'haskell';
haskell.innerText = 'Haskell';
list.appendChild(haskell);
```

3.2.4.3. insertBefore

把子节点插入到指定的位置，使用方法如下：

```javascript
parentElement.insertBefore(newElement, referenceElement)
```

子节点会插入到 `referenceElement` 之前

3.2.4.4. setAttribute

在指定元素中添加一个属性节点，如果元素中已有该属性改变属性值

```javascript
const div = document.getElementById('id')
div.setAttribute('class', 'white');//第一个参数属性名，第二个参数属性值。
```

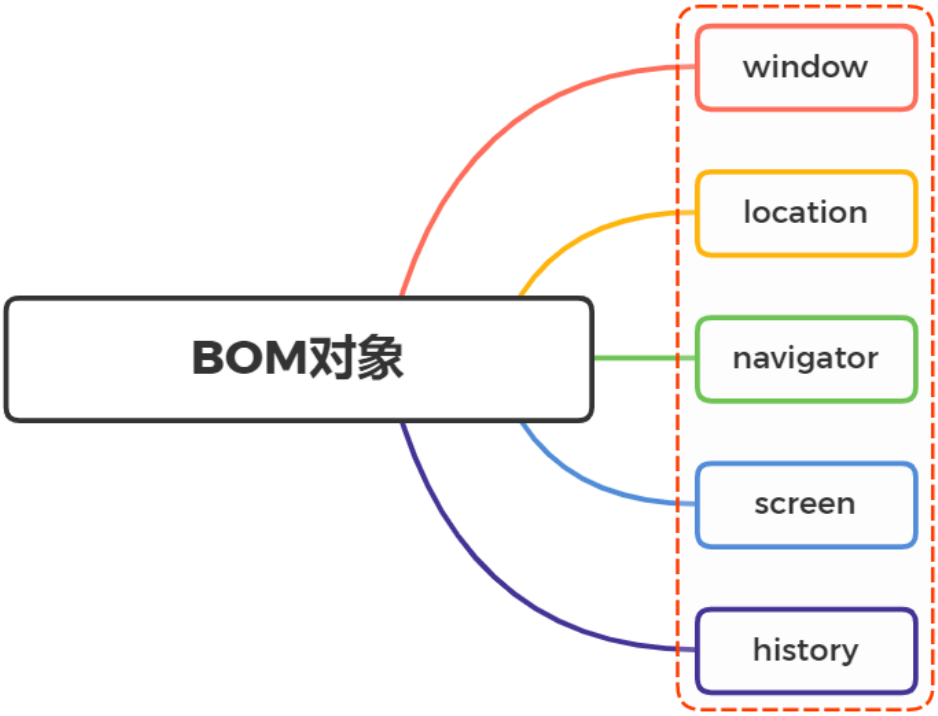
3.2.5. 删除节点

删除一个节点，首先要获得该节点本身以及它的父节点，然后，调用父节点的 `removeChild` 把自己删掉

```javascript
// 拿到待删除节点:
const self = document.getElementById('to-be-removed');
// 拿到父节点:
const parent = self.parentElement;
// 删除:
const removed = parent.removeChild(self);
removed === self; // true
```

删除后的节点虽然不在文档树中了，但其实它还在内存中，可以随时再次被添加到别的位置

4. 说说你对BOM的理解，常见的BOM对象你了解哪些?



4.1. 是什么

`BOM` (Browser Object Model)，浏览器对象模型，提供了独立于内容与浏览器窗口进行交互的对象

其作用就是跟浏览器做一些交互效果,比如如何进行页面的后退，前进，刷新，浏览器的窗口发生变化，滚动条的滚动，以及获取客户的一些信息如：浏览器品牌版本，屏幕分辨率

浏览器的全部内容可以看成 `DOM` ，整个浏览器可以看成 `BOM` 。区别如下：

DOM

- 文档对象模型
- DOM 就是把「文档」当做一个「对象」来看待
- DOM 的顶级对象是 document
- DOM 主要学习的是操作页面元素
- DOM 是 W3C 标准规范

BOM

- 浏览器对象模型
- 把「浏览器」当做一个「对象」来看待
- BOM 的顶级对象是 window
- BOM 学习的是浏览器窗口交互的一些对象
- BOM 是浏览器厂商在各自浏览器上定义的，兼容性较差

4.2. window

`Bom` 的核心对象是 `window` ，它表示浏览器的一个实例

在浏览器中，`window` 对象有双重角色，即是浏览器窗口的一个接口，又是全局对象

因此所有在全局作用域中声明的变量、函数都会变成 `window` 对象的属性和方法

JavaScript | 复制代码

```
var name = 'js每日一题';
function lookName(){
  alert(this.name);
}

console.log(window.name);  //js每日一题
lookName();                //js每日一题
window.lookName();         //js每日一题
```

关于窗口控制方法如下：

- `moveBy(x,y)`：从当前位置水平移动窗体x个像素，垂直移动窗体y个像素，x为负数，将向左移动窗体，y为负数，将向上移动窗体
- `moveTo(x,y)`：移动窗体左上角到相对于屏幕左上角的(x,y)点
- `resizeBy(w,h)`：相对窗体当前的大小，宽度调整w个像素，高度调整h个像素。如果参数为负值，将缩小窗体，反之扩大窗体
- `resizeTo(w,h)`：把窗体宽度调整为w个像素，高度调整为h个像素
- `scrollTo(x,y)`：如果有滚动条，将横向滚动条移动到相对于窗体宽度为x个像素的位置，将纵向滚动条移动到相对于窗体高度为y个像素的位置
- `scrollBy(x,y)`：如果有滚动条，将横向滚动条向左移动x个像素，将纵向滚动条向下移动y个像素

`window.open()` 既可以导航到一个特定的 `url`，也可以打开一个新的浏览器窗口

如果 `window.open()` 传递了第二个参数，且该参数是已有窗口或者框架的名称，那么就会在目标窗口加载第一个参数指定的URL

JavaScript | 复制代码

```
window.open('htttp://www.vue3js.cn','topFrame')
==> < a href=" " target="topFrame"></ a>
```

`window.open()` 会返回新窗口的引用，也就是新窗口的 `window` 对象

JavaScript | 复制代码

```
const myWin = window.open('http://www.vue3js.cn','myWin')
```

`window.close()` 仅用于通过 `window.open()` 打开的窗口

新创建的 `window` 对象有一个 `opener` 属性，该属性指向打开他的原始窗口对象

4.3. location

`url` 地址如下：

```javascript
http://foouser:barpassword@www.wrox.com:80/WileyCDA/?q=javascript#contents
```

`location` 属性描述如下：

属性名	例子	说明
hash	"#contents"	utl中#后面的字符，没有则返回空串
host	www.wrox.com:80	服务器名称和端口号
hostname	www.wrox.com	域名，不带端口号
href	http://www.wrox.com:80/WileyCDA/?q=javascript#contents	完整url
pathname	"/WileyCDA/"	服务器下面的文件路径
port	80	url的端口号，没有则为空
protocol	http:	使用的协议
search	?q=javascript	url的查询字符串，通常为？后面的内容

除了 `hash` 之外，只要修改 `location` 的一个属性，就会导致页面重新加载新 `URL`

`location.reload()`，此方法可以重新刷新当前页面。这个方法会根据最有效的方式刷新页面，如果页面自上一次请求以来没有改变过，页面就会从浏览器缓存中重新加载

如果要强制从服务器中重新加载，传递一个参数 `true` 即可

4.4. navigator

`navigator` 对象主要用来获取浏览器的属性，区分浏览器类型。属性较多，且兼容性比较复杂

下表列出了 `navigator` 对象接口定义的属性和方法：

属性/方法	说　明
`activeVrDisplays`	返回数组，包含 `ispresenting` 属性为 `true` 的 `VRDisplay` 实例
`appCodeName`	即使在非 Mozilla 浏览器中也会返回 `"Mozilla"`
`appName`	浏览器全名
`appVersion`	浏览器版本。通常与实际的浏览器版本不一致
`battery`	返回暴露 Battery Status API 的 `BatteryManager` 对象
`buildId`	浏览器的构建编号
`connection`	返回暴露 Network Information API 的 `NetworkInformation` 对象
`cookieEnabled`	返回布尔值，表示是否启用了 cookie
`credentials`	返回暴露 Credentials Management API 的 `CredentialsContainer` 对象
`deviceMemory`	返回单位为 GB 的设备内存容量
`doNotTrack`	返回用户的“不跟踪”（do-not-track）设置
`geolocation`	返回暴露 Geolocation API 的 `Geolocation` 对象
`getVRDisplays()`	返回数组，包含可用的每个 `VRDisplay` 实例
`getUserMedia()`	返回与可用媒体设备硬件关联的流
`hardwareConcurrency`	返回设备的处理器核心数量
`javaEnabled`	返回布尔值，表示浏览器是否启用了 Java
`language`	返回浏览器的主语言
`languages`	返回浏览器偏好的语言数组

locks	返回暴露 Web Locks API 的 LockManager 对象
mediaCapabilities	返回暴露 Media Capabilities API 的 MediaCapabilities 对象
mediaDevices	返回可用的媒体设备
maxTouchPoints	返回设备触摸屏支持的最大触点数
mimeTypes	返回浏览器中注册的 MIME 类型数组
onLine	返回布尔值，表示浏览器是否联网
oscpu	返回浏览器运行设备的操作系统和（或）CPU
permissions	返回暴露 Permissions API 的 Permissions 对象
platform	返回浏览器运行的系统平台
plugins	返回浏览器安装的插件数组。在 IE 中，这个数组包含页面中所有<embed>元素
product	返回产品名称（通常是"Gecko"）
productSub	返回产品的额外信息（通常是 Gecko 的版本）
registerProtocolHandler()	将一个网站注册为特定协议的处理程序
requestMediaKeySystemAccess()	返回一个期约，解决为 MediaKeySystemAccess 对象
sendBeacon()	异步传输一些小数据
serviceWorker	返回用来与 ServiceWorker 实例交互的 ServiceWorkerContainer
share()	返回当前平台的原生共享机制
storage	返回暴露 Storage API 的 StorageManager 对象
userAgent	返回浏览器的用户代理字符串
vendor	返回浏览器的厂商名称
vendorSub	返回浏览器厂商的更多信息
vibrate()	触发设备振动
webdriver	返回浏览器当前是否被自动化程序控制

4.5. screen

保存的纯粹是客户端能力信息，也就是浏览器窗口外面的客户端显示器的信息，比如像素宽度和像素高度

属　性	说　明
availHeight	屏幕像素高度减去系统组件高度（只读）
availLeft	没有被系统组件占用的屏幕的最左侧像素（只读）
availTop	没有被系统组件占用的屏幕的最顶端像素（只读）
availWidth	屏幕像素宽度减去系统组件宽度（只读）
colorDepth	表示屏幕颜色的位数；多数系统是 32（只读）
height	屏幕像素高度
left	当前屏幕左边的像素距离
pixelDepth	屏幕的位深（只读）
top	当前屏幕顶端的像素距离
width	屏幕像素宽度
orientation	返回 Screen Orientation API 中屏幕的朝向

4.6. history

`history` 对象主要用来操作浏览器 `URL` 的历史记录，可以通过参数向前，向后，或者向指定 `URL` 跳转

常用的属性如下：

- `history.go()`

接收一个整数数字或者字符串参数：向最近的一个记录中包含指定字符串的页面跳转，

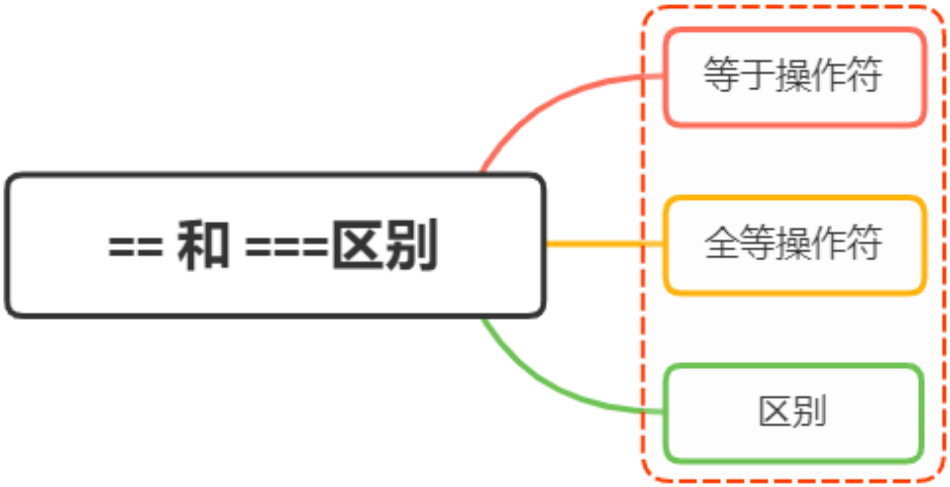
```
JavaScript | 复制代码
1  history.go('maixaofei.com')
```

当参数为整数数字的时候，正数表示向前跳转指定的页面，负数为向后跳转指定的页面

```
JavaScript | 复制代码
1  history.go(3) //向前跳转三个记录
2  history.go(-1) //向后跳转一个记录
```

- `history.forward()`：向前跳转一个页面
- `history.back()`：向后跳转一个页面
- `history.length`：获取历史记录数

5. == 和 ===区别，分别在什么情况使用



5.1. 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

JavaScript | 复制代码

```
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

JavaScript | 复制代码

```
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

JavaScript | 复制代码

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

`null` 和 `undefined` 相等

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

如果有任一操作数是 `NaN` ，则相等操作符返回 `false`

```javascript
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

5.2. 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

```javascript
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