浏览器

1 事件机制

事件触发三阶段

- `document` 往事件触发处传播，遇到注册的捕获事件会触发
- 传播到事件触发处时触发注册的事件
- 从事件触发处往 `document` 传播，遇到注册的冒泡事件会触发

> 事件触发一般来说会按照上面的顺序进行，但是也有特例，如果给一个目标节点同时注册冒泡和捕获事件，事件触发会按照注册的顺序执行

```
// 以下会先打印冒泡然后是捕获
node.addEventListener('click',(event) =>{
    console.log('冒泡')
},false);
node.addEventListener('click',(event) =>{
    console.log('捕获 ')
},true)
```

注册事件

- 通常我们使用 `addEventListener` 注册事件，该函数的第三个参数可以是布尔值，也可以是对象。对于布尔值 `useCapture` 参数来说，该参数默认值为 `false` 。 `useCapture` 决定了注册的事件是捕获事件还是冒泡事件
- 一般来说，我们只希望事件只触发在目标上，这时候可以使用 `stopPropagation` 来阻止事件的进一步传播。通常我们认为 `stopPropagation` 是用来阻止事件冒泡的，其实该函数也可以阻止捕获事件。`stopImmediatePropagation` 同样也能实现阻止事件，但是还能阻止该事件目标执行别的注册事件

```
node.addEventListener('click',(event) =>{
    event.stopImmediatePropagation()
    console.log('冒泡')
},false);
// 点击 node 只会执行上面的函数，该函数不会执行
node.addEventListener('click',(event) => {
    console.log('捕获 ')
},true)
```

事件代理

> 如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话应该注册在父节点上

```
<ul id="ul">
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
<script>
    let ul = document.querySelector('##ul')
    ul.addEventListener('click', (event) => {
        console.log(event.target);
    })
</script>
```

> 事件代理的方式相对于直接给目标注册事件来说，有以下优点

- 节省内存
- 不需要给子节点注销事件

2 跨域

> 因为浏览器出于安全考虑，有同源策略。也就是说，如果协议、域名或者端口有一个不同就是跨域，`Ajax` 请求会失败

JSONP

> `JSONP` 的原理很简单，就是利用 `<script>` 标签没有跨域限制的漏洞。通过 `<script>` 标签指向一个需要访问的地址并提供一个回调函数来接收数据当需要通讯时

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

- `JSONP` 使用简单且兼容性不错，但是只限于 `get` 请求

CORS

- `CORS` 需要浏览器和后端同时支持
- 浏览器会自动进行 `CORS` 通信，实现 `CORS` 通信的关键是后端。只要后端实现了 `CORS`，就实现了跨域。
- 服务端设置 `Access-Control-Allow-Origin` 就可以开启 `CORS`。 该属性表示哪些域名可以访问资源，如果设置通配符则表示所有网站都可以访问资源

document.domain

- 该方式只能用于二级域名相同的情况下，比如 `a.test.com` 和 `b.test.com` 适用于该方式。
- 只需要给页面添加 `document.domain = 'test.com'` 表示二级域名都相同就可以实现跨域

postMessage

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

```
// 发送消息端
window.parent.postMessage('message',
'http://blog.poetries.com');

// 接收消息端
var mc = new MessageChannel();
mc.addEventListener('message', (event) => {
    var origin = event.origin || event.originalEvent.origin;
    if (origin === 'http://blog.poetries.com') {
        console.log('验证通过')
    }
});
```

3 Event loop

JS中的event loop

> 众所周知 `JS` 是门非阻塞单线程语言，因为在最初 `JS` 就是为了和浏览器交互而诞生的。如果 `JS` 是门多线程的语言话，我们在多个线程中处理 `DOM` 就可能会发生问题（一个线程中新加节点，另一个线程中删除节点）

- `JS` 在执行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 `Task`（有多种 `task`） 队列中。一旦执行栈为空， `Event` `Loop` 就会从 `Task` 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 `JS` 中的异步还是同步行为

```
console.log('script start');

setTimeout(function() {
  console.log('setTimeout');
}, 0);

console.log('script end');
```

> 不同的任务源会被分配到不同的 `Task` 队列中，任务源可以分为 微任务（`microtask`）和 宏任务（`macrotask`）。在 `ES6` 规范中，`microtask` 称为 jobs，macrotask 称为 task

```
console.log('script start');

setTimeout(function() {
  console.log('setTimeout');
}, 0);

new Promise((resolve) => {
    console.log('Promise')
    resolve()
}).then(function() {
  console.log('promise1');
}).then(function() {
  console.log('promise2');
});

console.log('script end');
// script start => Promise => script end => promise1 =>
promise2 => setTimeout
```

> 以上代码虽然 `setTimeout` 写在 `Promise` 之前，但是因为 `Promise` 属于微任务而 `setTimeout` 属于宏任务

微任务

- `process.nextTick`
- `promise`
- `Object.observe`
- `MutationObserver`

宏任务

- `script`
- `setTimeout`

- `setInterval`
- `setImmediate`
- `I/O`
- `UI rendering`

> 宏任务中包括了 `script` ，浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务

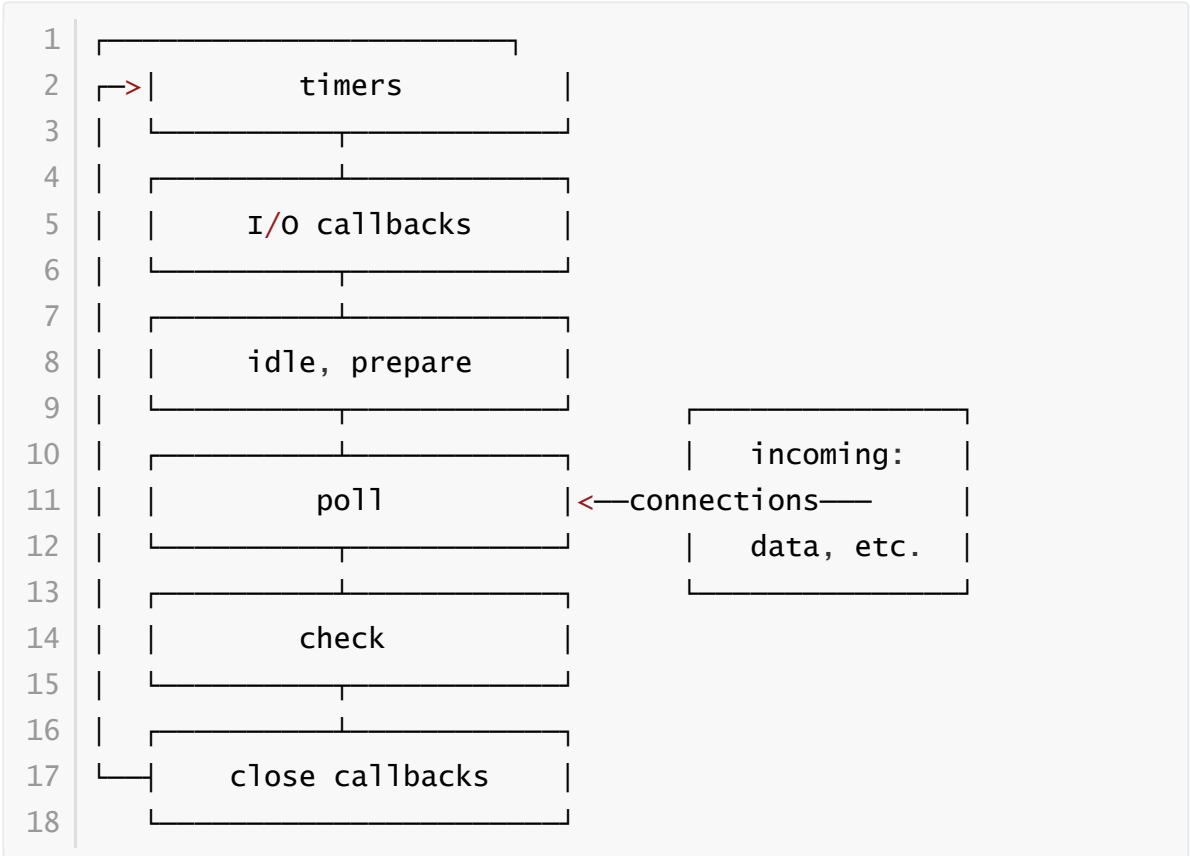
所以正确的一次 Event loop 顺序是这样的

- 执行同步代码，这属于宏任务
- 执行栈为空，查询是否有微任务需要执行
- 执行所有微任务
- 必要的话渲染 UI
- 然后开始下一轮 `Event loop`，执行宏任务中的异步代码

> 通过上述的 `Event loop` 顺序可知，如果宏任务中的异步代码有大量的计算并且需要操作 `DOM` 的话，为了更快的响应界面响应，我们可以把操作 `DOM` 放入微任务中

Node 中的 Event loop

- `Node` 中的 `Event loop` 和浏览器中的不相同。
- `Node` 的 `Event loop` 分为`6`个阶段，它们会按照顺序反复运行



timer

- `timers` 阶段会执行 `setTimeout` 和 `setInterval`
- 一个 timer 指定的时间并不是准确时间，而是在达到这个时间后尽快执行回调，可能会因为系统正在执行别的事务而延迟

I/O

- `I/O` 阶段会执行除了 `close` 事件，定时器和 `setImmediate` 的回调

poll

- `poll` 阶段很重要，这一阶段中，系统会做两件事情
  - 执行到点的定时器
  - 执行 `poll` 队列中的事件
- 并且当 `poll` 中没有定时器的情况下，会发现以下两件事情
  - 如果 `poll` 队列不为空，会遍历回调队列并同步执行，直到队列为空或者系统限制
  - 如果 `poll` 队列为空，会有两件事发生
  - 如果有 `setImmediate` 需要执行，`poll` 阶段会停止并且进入到 `check` 阶段执行 `setImmediate`
  - 如果没有 `setImmediate` 需要执行，会等待回调被加入到队列中并立即执行回调
  - 如果有别的定时器需要被执行，会回到 `timer` 阶段执行回调。

check

- `check` 阶段执行 `setImmediate`

close callbacks

- `close callbacks` 阶段执行 `close` 事件
- 并且在 `Node` 中，有些情况下的定时器执行顺序是随机的

```
setTimeout(() => {
    console.log('setTimeout');
}, 0);
setImmediate(() => {
    console.log('setImmediate');
})
// 这里可能会输出 setTimeout，setImmediate
// 可能也会相反的输出，这取决于性能
// 因为可能进入 event loop 用了不到 1 毫秒，这时候会执行 setImmediate
// 否则会执行 setTimeout
```

> 上面介绍的都是 `macrotask` 的执行情况，`microtask` 会在以上每个阶段完成后立即执行

```
setTimeout(()=>{
    console.log('timer1')

    Promise.resolve().then(function() {
        console.log('promise1')
    })
}, 0)

setTimeout(()=>{
    console.log('timer2')

    Promise.resolve().then(function() {
        console.log('promise2')
    })
}, 0)

// 以上代码在浏览器和 node 中打印情况是不同的
// 浏览器中一定打印 timer1, promise1, timer2, promise2
// node 中可能打印 timer1, timer2, promise1, promise2
// 也可能打印 timer1, promise1, timer2, promise2
```

> `Node` 中的 `process.nextTick` 会先于其他 `microtask` 执行

```
setTimeout(() => {
 console.log("timer1");

 Promise.resolve().then(function() {
   console.log("promise1");
 });
}, 0);

process.nextTick(() => {
 console.log("nextTick");
});
// nextTick, timer1, promise1
```

4 Service Worker

> `Service workers` 本质上充当Web应用程序与浏览器之间的代理服务器，也可以在网络可用时作为浏览器和网络间的代理。它们旨在（除其他之外）使得能够创建有效的离线体验，拦截网络请求并基于网络是否可用以及更新的资源是否驻留在服务器上来采取适当的动作。他们还允许访问推送通知和后台同步API

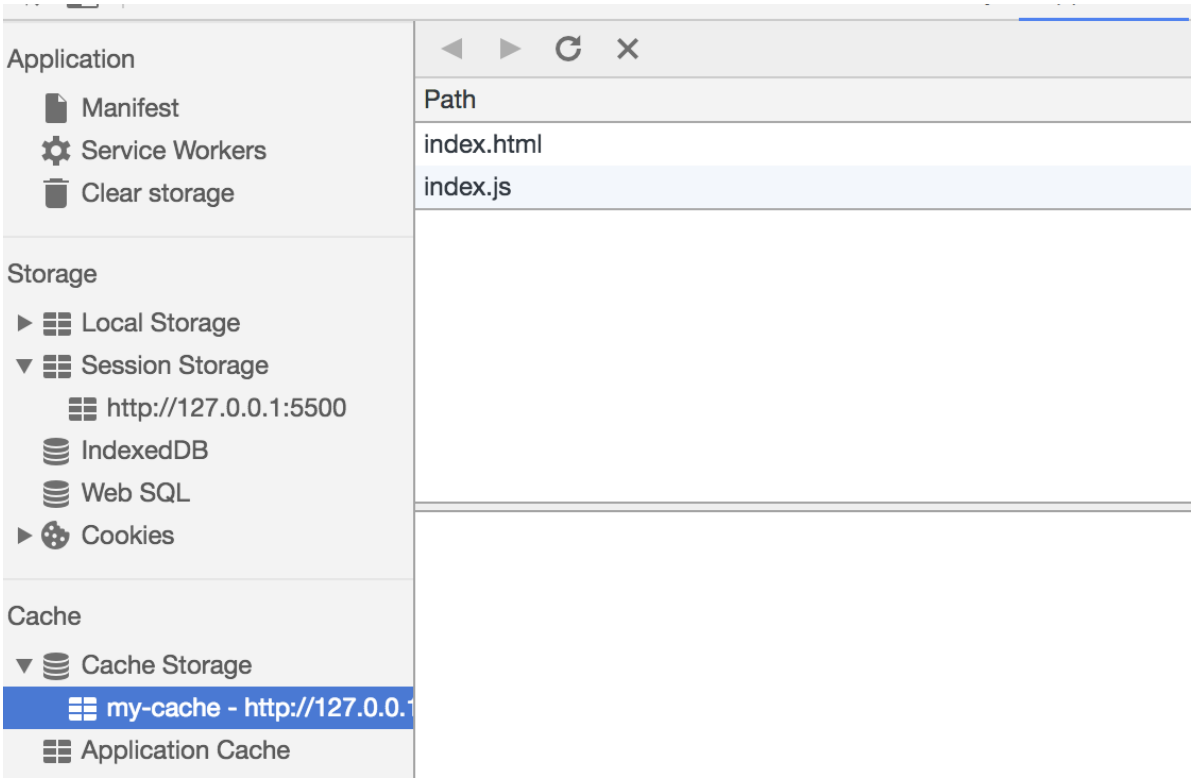
目前该技术通常用来做缓存文件，提高首屏速度

```
// index.js
if (navigator.serviceWorker) {
  navigator.serviceWorker
    .register("sw.js")
    .then(function(registration) {
      console.log("service worker 注册成功");
    })
    .catch(function(err) {
      console.log("servcie worker 注册失败");
    });
}
// sw.js
// 监听 `install` 事件，回调中缓存所需文件
self.addEventListener("install", e => {
  e.waitUntil(
    caches.open("my-cache").then(function(cache) {
      return cache.addAll(["./index.html", "./index.js"]);
    })
  );
});

// 拦截所有请求事件
// 如果缓存中已经有请求的数据就直接用缓存，否则去请求数据
self.addEventListener("fetch", e => {
  e.respondWith(
    caches.match(e.request).then(function(response) {
      if (response) {
        return response;
      }
      console.log("fetch source");
    })
  );
});
```

> 打开页面，可以在开发者工具中的 `Application` 看到 `Service Worker` 已经启动了

> 在 Cache 中也可以发现我们所需的文件已被缓存



> 当我们重新刷新页面可以发现我们缓存的数据是从 `Service Worker` 中读取的

5 渲染机制

浏览器的渲染机制一般分为以下几个步骤

- 处理 `HTML` 并构建 `DOM` 树。
- 处理 `CSS` 构建 `CSSOM` 树。
- 将 `DOM` 与 `CSSOM` 合并成一个渲染树。
- 根据渲染树来布局，计算每个节点的位置。
- 调用 `GPU` 绘制，合成图层，显示在屏幕上

- 在构建 CSSOM 树时，会阻塞渲染，直至 CSSOM 树构建完成。并且构建 CSSOM 树是一个十分消耗性能的过程，所以应该尽量保证层级扁平，减少过度层叠，越是具体的 CSS 选择器，执行速度越慢
- 当 HTML 解析到 script 标签时，会暂停构建 DOM，完成后才会从暂停的地方重新开始。也就是说，如果你想首屏渲染的越快，就越不应该在首屏就加载 JS 文件。并且 CSS 也会影响 JS 的执行，只有当解析完样式表才会执行 JS，所以也可以认为这种情况下，CSS 也会暂停构建 DOM

图层

> 一般来说，可以把普通文档流看成一个图层。特定的属性可以生成一个新的图层。不同的图层渲染互不影响，所以对于某些频繁需要渲染的建议单独生成一个新图层，提高性能。但也不能生成过多的图层，会引起反作用

- 通过以下几个常用属性可以生成新图层
  - `3D`变换：`translate3d`、`translateZ`
  - `will-change`
  - `video`、`iframe`标签
  - 通过动画实现的`opacity`动画转换
  - `position: fixed`

重绘（Repaint）和回流（Reflow）

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

> 回流必定会发生重绘，重绘不一定会引发回流。回流所需的成本比重绘高的多，改变深层次的节点很可能导致父节点的一系列回流

- 所以以下几个动作可能会导致性能问题

  ：

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

很多人不知道的是，重绘和回流其实和`Event loop`有关

- 当`Event loop`执行完`Microtasks`后，会判断`document`是否需要更新。因为浏览器是`60Hz`的刷新率，每`16ms`才会更新一次。
- 然后判断是否有`resize`或者`scroll`，有的话会去触发事件，所以`resize`和`scroll`事件也是至少`16ms`才会触发一次，并且自带节流功能。
- 判断是否触发了`media query`
- 更新动画并且发送事件
- 判断是否有全屏操作事件
- 执行`requestAnimationFrame`回调
- 执行`IntersectionObserver`回调，该方法用于判断元素是否可见，可以用于懒加载上，但是兼容性不好
- 更新界面
- 以上就是一帧中可能会做的事情。如果在一帧中有空闲时间，就会去执行`requestIdleCallback`回调

减少重绘和回流

- 使用`translate`替代`top`

- 使用 `visibility` 替换`display: none` ，因为前者只会引起重绘，后者会引发回流（改变了布局）
- 不要使用 `table` 布局，可能很小的一个小改动会造成整个 table 的重新布局
- 动画实现的速度的选择，动画速度越快，回流次数越多，也可以选择使用 `requestAnimationFrame`
- `CSS` 选择符从右往左匹配查找，避免 `DOM` 深度过深
- 将频繁运行的动画变为图层，图层能够阻止该节点回流影响别的元素。比如对于`video`标签，浏览器会自动将该节点变为图层