浏览器中的事件循环机制

hjp

2024/07/25

先看一段代码

```
// 下面代码的输出结果是什么?

console.log('script start');

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

new Promise(resolve ⇒ {
  console.log('Promise')
  resolve('resolve')
}).then(res ⇒ {
  console.log(res)
})

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

▶ 答案

单线程的JavaScript

JavaScript语言的一大特点就是单线程，也就是说，同一个时间只能做一件事。那么，为什么JavaScript不能有多个线程呢?

JavaScript的单线程，与它的用途有关。作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。比如，假定JavaScript同时有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准?

所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也很难会改变。

为了利用多核CPU的计算能力，HTML5提出Web Worker标准，允许JavaScript脚本创建多个线程，但是子线程完全受主线程控制，且不得操作DOM。所以，这个新标准并没有改变JavaScript单线程的本质。

任务队列

单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。如果前一个任务耗时很长，后一个任务就不得不一直等着。

如果排队是因为计算量大，CPU忙不过来，倒也算了，但是很多时候CPU是闲着的，因为IO设备（输入输出设备）很慢（比如Ajax操作从网络读取数据），不得不等着结果出来，再往下执行。

JavaScript语言的设计者意识到，这时主线程完全可以不管IO设备，挂起处于等待中的任务，先运行排在后面的任务。等到IO设备返回了结果，再回过头，把挂起的任务继续执行下去。

于是，所有任务可以分成两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）。同步任务指的是，在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；异步任务指的是，不进入主线程、而进入"任务队列"（task queue）的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

异步执行机制

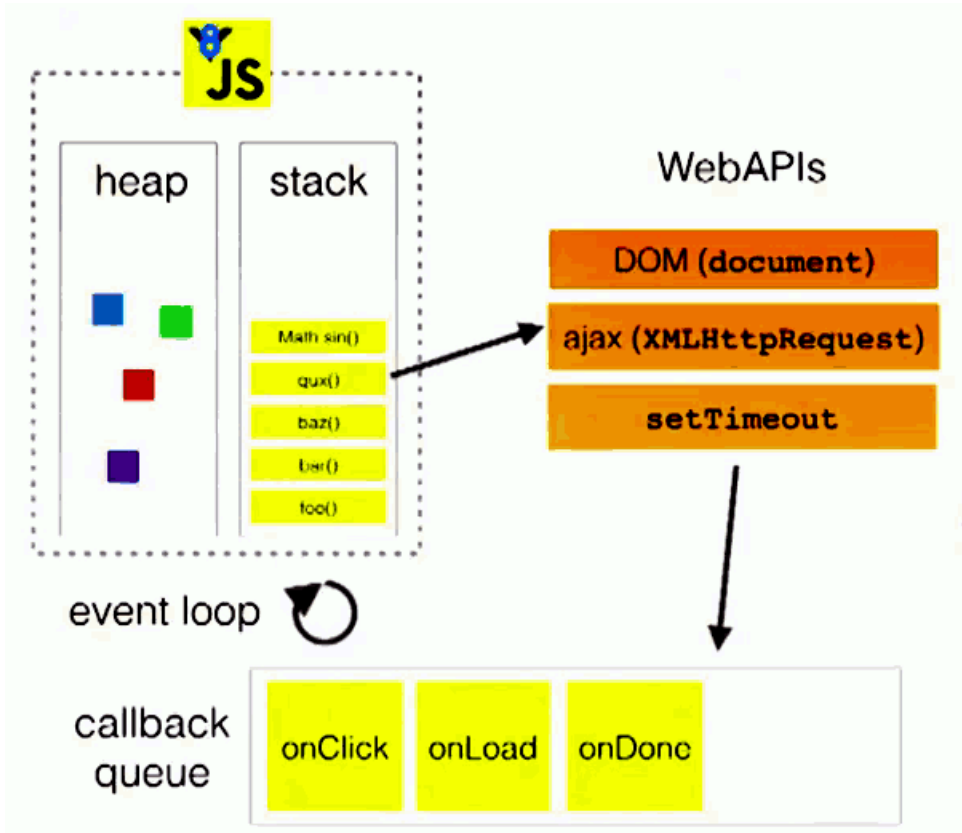
（1）所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。

（2）主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。

（3）一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。

（4）主线程不断重复上面的第三步。

Event Loop运行机制图



知识点

我们知道，JS 引擎只知道执行 JS，渲染引擎只知道渲染，它们两个并不知道彼此，该怎么配合呢？

答案就是Event Loop。

Event Loop是宿主环境为了集合渲染和 JS 执行，也为了处理 JS 执行时的高优先级任务而设计的机制。宿主环境包括但不限于浏览器、Node等等。

不同的宿主环境有不同的设计：浏览器里面主要是调度渲染和 JS 执行，还有 worker；node 里面主要是调度各种io。

浏览器中的Event Loop

1. check

浏览器里面执行一个 JS 任务就是一个 event loop，每个 loop 结束会检查下是否需要渲染，是否需要处理 worker 的消息，通过这种每次 loop 结束都 check 的方式来综合渲染、JS 执行、worker 等，让它们都能在一个线程内得到执行（渲染其实是在别的线程，但是会和 JS 线程相互阻塞）。

这样就解决了渲染、JS 执行、worker 这三者的调度问题。

但是这样有没有问题?

我们会在任务队列中不断的放新的任务，这样如果有更高优的任务是不是要等所有任务都执行完才能被执行。如果是“急事”呢?

所以这样还不行，我们要给 event loop 加上“急事”处理的快速通道，这就是微任务 micro tasks。

浏览器中的Event Loop

2. micro tasks

任务还是每次取一个执行，执行完检查下要不要渲染，处理下 worker 消息，但是也给高优先级的“急事”加入了插队机制，会在执行完任务之后，把所有的急事（micro task）全部处理完。 这样，event loop貌似就挺完美的了，每次都会检查是否要渲染，也能更快的处理 JS 的“急事”。

浏览器中的Event Loop

3. requestAnimationFrame

JS 执行完，开始渲染之前会有一个生命周期，就是 requestAnimationFrame，在这里面做一些计算最合适了，能保证一定是在渲染之前做的计算。

浏览器中的Event Loop

4. event loop 的问题

每一帧的计算和渲染是有固定频率的，如果 JS 执行时间过长，超过了一帧的刷新时间，那么就会导致渲染延迟，甚至掉帧（因为上一帧的数据还没渲染到界面就被覆盖成新的数据了），给用户的感受就是“界面卡了”。

什么情况会导致帧刷新拖延甚至帧数据被覆盖（丢帧）呢？每个 loop 在 check 渲染之前的每一个阶段都有可能，也就是 task、microtask、requestAnimationFrame、requestIdleCallback 都有可能导致阻塞了 check，这样等到了 check 的时候发现要渲染了，再去渲染的时候就晚了。

所以主线程 JS 代码不要做太多的计算（不像安卓会很自然的起一个线程来做），要做拆分，这也是为啥 ui 框架要做计算的 fiber 化，就是因为处理交互的时候，不能让计算阻塞了渲染，要递归改循环，通过链表来做计算的暂停恢复。对于一些计算复杂的操作，可以放到worker中执行。

除了 JS 代码要注意之外，如果浏览器能够提供 API 就是在每帧间隔来执行，那样岂不是就不会阻塞了，所以后来有了 requestIdleCallback。

浏览器中的Event Loop

5. requestIdleCallback

requestIdleCallback 会在每次 check 结束发现距离下一帧的刷新还有时间，就执行一下这个。如果时间不够，就下一帧再说。

如果每一帧都没时间呢，那也不行，所以提供了 timeout 的参数可以指定最长的等待时间，如果一直没时间执行这个逻辑，那么就算拖延了帧渲染也要执行。

这个 api 对于前端框架来说太需要了，框架就是希望计算不阻塞渲染，也就是在每一帧的间隔时间（idle时间）做计算，但是这个 api 毕竟是最近加的，有兼容问题，所以 react 自己实现了类似 idle callback 的fiber 机制，在执行逻辑之前判断一下离下一帧刷新还有多久，来判断是否执行逻辑。

总结

总之，浏览器里有 JS 引擎做 JS 代码的执行，利用注入的浏览器 API 完成功能，有渲染引擎做页面渲染，两者都比较纯粹，需要一个调度的方式，就是 event loop。

event loop 实现了 task 和 急事处理机制 microtask，而且每次 loop 结束会 check 是否要渲染，渲染之前会有 requestAnimationFrames 生命周期。

帧刷新尽量不被拖延否则会卡顿掉帧，所以就需要 JS 代码里面不要做过多计算，于是有了 requestIdleCallback 的 api，希望在每次 check 完发现还有时间就执行，没时间就不执行（这个deadline的时间也作为参数让 js 代码自己判断），为了避免一直没时间被冷落，还提供了 timeout 参数强制执行。

防止计算时间过长导致渲染掉帧是 ui 框架一直关注的问题，就是怎么不阻塞渲染，让逻辑能够拆成帧间隔时间内能够执行完的小块。浏览器提供了 idlecallback 的 api，很多 ui 框架也通过递归改循环然后记录状态等方式实现了计算量的拆分，目的只有一个：loop 内的逻辑执行不能阻塞 check，也就是不能阻塞渲染引擎做帧刷新。所以不管是 JS 代码宏微任务、 requestAnimationCallback、requestIdleCallback 都不能计算时间太长。这个问题是前端开发的持续性阵痛。

宏任务与微任务

宏任务

- <script>标签中的运行代码
- 事件触发的回调函数，例如DOM Events、I/O、requestAnimationFrame
- setTimeout、setInterval的回调函数

微任务

- promises：Promise.then、Promise.catch、Promise.finally
- MutationObserver
- queueMicrotask
- process.nextTick：Node独有

回头看

```
// 下面代码的输出结果是什么?

console.log('script start');

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

new Promise(resolve => {
  console.log('Promise')
  resolve('resolve')
}).then(res => {
  console.log(res)
})

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

引用

How JavaScript works: Event loop and the rise of Async programming + 5 ways to better coding with async/await

JavaScript 运行机制详解：再谈Event Loop

Event Loop 和 JS 引擎、渲染引擎的关系