Js异步机制的实现

Js异步机制

JavaScript 是一门单线程语言，所谓单线程，就是指一次只能完成一件任务，如果有多个任务，就必须排队，前面一个任务完成，再执行后面一个任务，以此类推。这种模式的好处是实现起来比较简单，执行环境相对单纯，坏处是只要有一个任务耗时很长，后面的任务都必须排队等着，会拖延整个程序的执行。常见的浏览器无响应也就是假死状态，往往就是因为某一段 Javascript 代码长时间运行比如死循环，导致整个页面卡在这个地方，其他任务无法执行。

执行机制

为了解决上述问题， Javascript 将任务的执行模式分为两种：同步 Synchronous 与异步 Asynchronous ，同步或非同步，表明着是否需要将整个流程按顺序地完成，阻塞或非阻塞，意味着你调用的函数会不会立刻告诉你结果。

同步

同步模式就是同步阻塞，后一个任务等待前一个任务结束，然后再执行，程序的执行顺序与任务的排列顺序是一致的、同步的。

```
var i = 100;
while(--i) { console.log(i); }
console.log("while 执行完毕我才能执行");
```

异步

异步执行就是非阻塞模式执行，每一个任务有一个或多个回调函数 callback ，前一个任务结束后，不是执行后一个任务，而是执行回调函数，后一个任务则是不等前一个任务结束就执行，所以程序的执行顺序与任务的排列顺序是不一致的、异步的。浏览器对于每个 Tab 只分配了一个 Js 线程，主要任务是与用户交互以及操作 DOM 等，而这也就决定它只能为单线程，否则会带来很复杂的同步问题，例如假定 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，这时浏览器无法确定以哪个线程的操作为准。

```
setTimeout(() => console.log("我后执行"), 0);
// 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4
console.log("我先执行");
```

异步机制

首先来看一个例子，与上文一样来测试一个异步执行的操作

```
setTimeout(() => console.log("我在很长时间之后才执行"), 0);
var i = 3000000000;
while(--i) { }
console.log("循环执行完毕");
```

本地测试，设置的 setTimeout 回调函数大约在 30s 之后才执行，远远大于 4ms ，我在主线程设置了一个非常大的循环来阻塞 Js 主线程，注意我并没有设置一个死循环，假如我在此处设置死循环来阻塞主线程，那么设置的 setTimeout 回调函数将永远不会执行，此外由于渲染线程与 JS 引擎线程是互斥的， Js 线程在处理任务时渲染线程会被挂起，整个页面都将被阻塞，无法刷新甚至无法关闭，只能通过使用任务管理器结束 Tab 进程的方式关闭页面。

Js 实现异步是通过一个执行栈与一个任务队列来完成异步操作的，所有同步任务都是在主线程上执行的，形成执行栈，任务队列中存放各种事件回调（也可以称作消息），当执行栈中的任务处理完成后，主线程就开始读取任务队列中的任务并执行，不

目录

断往复循环。

例如上例中的 setTimeout 完成后的事件回调就存在任务队列中，这里需要说明的是浏览器定时计数器并不是由 JavaScript 引擎计数的，因为 JavaScript 引擎是单线程的，如果线程处于阻塞状态就会影响记计时的准确，计数是由浏览器线程进行计数的，当计数完毕，就将事件回调加入任务队列，同样 HTTP 请求在浏览器中也存在单独的线程，也是执行完毕后将事件回调置入任务队列。通过这个流程，就能够解释为什么上例中 setTimeout 的回调一直无法执行，是由于主线程也就是执行栈中的代码没有完成，不会去读取任务队列中的事件回调来执行，即使这个事件回调早已在任务队列中。

Event Loop

主线程从任务队列中读取事件，这个过程是循环不断的，所以整个的这种运行机制又称为 Event Loop ， Event Loop 是一个执行模型，在不同的地方有不同的实现，浏览器和 NodeJS 基于不同的技术实现了各自的 Event Loop 。浏览器的 Event Loop 是在 HTML5 的规范中明确定义， NodeJS 的 Event Loop 是基于 libuv 实现的。

在浏览器中的 Event Loop 由执行栈 Execution Stack 、后台线程 Background Threads 、宏队列 Macrotask Queue 、微队列 Microtask Queue 组成。

执行栈就是在主线程执行同步任务的数据结构，函数调用形成了一个由若干帧组成的栈。

后台线程就是浏览器实现对于 setTimeout 、 setInterval 、 XMLHttpRequest 等等的执行线程。

宏队列，一些异步任务的回调会依次进入宏队列，等待后续被调用，包括 setTimeout 、 setInterval 、 setImmediate(Node) 、 requestAnimationFrame 、 UI rendering 、 I/O 等操作

微队列，另一些异步任务的回调会依次进入微队列，等待后续调用，包括 Promise 、 process.nextTick(Node) 、 Object.observe 、 MutationObserver 等操作

当 Js 执行时，进行如下流程

1. 首先将执行栈中代码同步执行，将这些代码中异步任务加入后台线程中
2. 执行栈中的同步代码执行完毕后，执行栈清空，并开始扫描微队列
3. 取出微队列队首任务，放入执行栈中执行，此时微队列是进行了出队操作
4. 当执行栈执行完成后，继续出队微队列任务并执行，直到微队列任务全部执行完毕
5. 最后一个微队列任务出队并进入执行栈后微队列中任务为空，当执行栈任务完成后，开始扫面微队列为空，继续扫描宏队列任务，宏队列出队，放入执行栈中执行，执行完毕后继续扫描微队列为空则扫描宏队列，出队执行
6. 不断往复...

实例

```
// Step 1
console.log(1);

// Step 2
setTimeout(() => {
  console.log(2);
  Promise.resolve().then(() => {
    console.log(3);
  });
}, 0);

// Step 3
new Promise((resolve, reject) => {
  console.log(4);
  resolve();
}).then(() => {
  console.log(5);
})
```

目录

```
// Step 4
setTimeout(() => {
  console.log(6);
}, 0);

// Step 5
console.log(7);

// Step N
// ...

// Result
/*
  1
  4
  7
  5
  2
  3
  6
*/
```

Step 1

```
// 执行栈 console
// 微队列 []
// 宏队列 []
console.log(1); // 1
```

Step 2

```
// 执行栈 setTimeout
// 微队列 []
// 宏队列 [setTimeout1]
setTimeout(() => {
  console.log(2);
  Promise.resolve().then(() => {
    console.log(3);
  });
}, 0);
```

Step 3

```
// 执行栈 Promise
// 微队列 [then1]
// 宏队列 [setTimeout1]
new Promise((resolve, reject) => {
  console.log(4); // 4 // Promise是个函数对象，此处是同步执行的 // 执行
  resolve();
}).then(() => {
  console.log(5);
})
```

Step 4

```
// 执行栈 setTimeout
// 微队列 [then1]
// 宏队列 [setTimeout1 setTimeout2]
setTimeout(() => {
  console.log(6);
}, 0);
```

Step 5

目录

```
// 执行栈 console
// 微队列 [then1]
// 宏队列 [setTimeout1 setTimeout2]
console.log(7); // 7
```

Step 6

```
// 执行栈 then1
// 微队列 []
// 宏队列 [setTimeout1 setTimeout2]
console.log(5); // 5
```

Step 7

```
// 执行栈 setTimeout1
// 微队列 [then2]
// 宏队列 [setTimeout2]
console.log(2); // 2
Promise.resolve().then(() => {
    console.log(3);
});
```

Step 8

```
// 执行栈 then2
// 微队列 []
// 宏队列 [setTimeout2]
console.log(3); // 3
```

Step 9

```
// 执行栈 setTimeout2
// 微队列 []
// 宏队列 []
console.log(6); // 6
```

参考


```
https://www.jianshu.com/p/1a35857c78e5
https://segmentfault.com/a/1190000016278115
https://segmentfault.com/a/1190000012925872
https://www.cnblogs.com/sunidol/p/11301808.html
http://www.ruanyifeng.com/blog/2014/10/event-loop.html
https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/EventLoop
```


目录