JavaScript的运行机制

1. 课程目标

1. 了解进程与线程的基础概念，明确在浏览器中的进程与线程机制；
2. 了解浏览器与Node中的事件循环；

2. 课程大纲

1. 进程与线程；
2. 事件循环；

3. 进程与线程

3.1. 什么是进程

CPU是计算机的核心，承担所有的计算任务。

官网说法，进程是CPU资源分配的最小单位。

字面意思就是进行中的程序，我将它理解为一个可以独立运行且拥有自己的资源空间的任务程序，进程包括运行中的程序和程序所使用到的内存和系统资源。

CPU可以有很多进程，我们的电脑每打开一个软件就会产生一个或多个进程，为什么电脑运行的软件多就会卡，是因为CPU给每个进程分配资源空间，但是一个CPU一共就那么多资源，分出去越多，越卡，每个进程之间是相互独立的，CPU在运行一个进程时，其他的进程处于非运行状态，CPU使用 时间片轮转调度算法 来实现同时运行多个进程。

3.2. 什么是线程

线程是CPU调度的最小单位。

线程是建立在进程的基础上的一次程序运行单位，通俗点解释线程就是程序中的一个执行流，一个进程可以有多个线程。

一个进程中只有一个执行流称作单线程，即程序执行时，所走的程序路径按照连续顺序排下来，前面的必须处理好，后面的才会执行。

一个进程中有多个执行流称作多线程，即在一个程序中可以同时运行多个不同的线程来执行不同的任务， 也就是说允许单个程序创建多个并行执行的线程来完成各自的任务。

3.3. 进程和线程的区别

进程是操作系统分配资源的最小单位，线程是程序执行的最小单位。

一个进程由一个或多个线程组成，线程可以理解为是一个进程中代码的不同执行路线。

进程之间相互独立，但同一进程下的各个线程间共享程序的内存空间(包括代码段、数据集、堆等)及一些进程级的资源(如打开文件和信号)。

调度和切换：线程上下文切换比进程上下文切换要快得多。

3.4. 多进程和多线程

- 多进程：多进程指的是在同一个时间里，同一个计算机系统中如果允许两个或两个以上的进程处于运行状态。多进程带来的好处是明显的，比如大家可以在网易云听歌的同时打开编辑器敲代码，编辑器和网易云的进程之间不会相互干扰；
- 多线程：多线程是指程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务，也就是说允许单个程序创建多个并行执行的线程来完成各自的任务；

3.5. JS为什么是单线程

JS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?

还有人说js还有Worker线程，对的，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，但是子线程是完 全受主线程控制的，而且不得操作DOM。

所以，这个标准并没有改变JavaScript是单线程的本质。

3.6. 浏览器

拿Chrome来说，我们每打开一个Tab页就会产生一个进程。

3.6.1. 浏览器包含哪些进程

1. Browser进程
    1. 浏览器的主进程(负责协调、主控)，该进程只有一个；
    2. 负责浏览器界面显示，与用户交互。如前进，后退等；
    3. 负责各个页面的管理，创建和销毁其他进程；
    4. 将渲染(Renderer)进程得到的内存中的Bitmap(位图)，绘制到用户界面上；
    5. 网络资源的管理，下载等；
2. 第三方插件进程
    1. 每种类型的插件对应一个进程，当使用该插件时才创建；
3. GPU进程
    1. 该进程也只有一个，用于3D绘制等等；
4. 渲染进程
    1. 即通常所说的浏览器内核(Renderer进程，内部是多线程)；
    2. 每个Tab页面都有一个渲染进程，互不影响；
    3. 主要作用为页面渲染，脚本执行，事件处理等；

3.6.2. 为什么浏览器要多进程

假设浏览器是单进程，那么某个Tab页崩溃了，就影响了整个浏览器，体验有多差？同理如果插件崩溃了也会影响整个浏览器。

浏览器进程有很多，每个进程又有很多线程，都会占用内存

3.6.3. 渲染进程

页面的渲染，JS的执行，事件的循环，都在渲染进程内执行，所以我们要重点了解渲染进程

渲染进程是多线程的，看渲染进程的一些常用较为主要的线程：

3.6.3.1. GUI渲染线程

1. 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)；
2. 当我们修改了一些元素的颜色或者背景色，页面就会重绘(Repaint)；
3. 当我们修改元素的尺寸，页面就会回流(Reflow)；
4. 当页面需要Repaing和Reflow时GUI线程执行，绘制页面；
5. 回流(Reflow)比重绘(Repaint)的成本要高，我们要尽量避免Reflow和Repaint；
6. GUI渲染线程与JS引擎线程是互斥的：
    1. 当JS引擎执行时GUI线程会被挂起(相当于被冻结了)；
    2. 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；

3.6.3.2. JS引擎线程

1. JS引擎线程就是JS内核，负责处理Javascript脚本程序(例如V8引擎)；
2. JS引擎线程负责解析Javascript脚本，运行代码；
3. JS引擎一直等待着任务队列中任务的到来，然后加以处理：
    1. 浏览器同时只能有一个JS引擎线程在运行JS程序，所以js是单线程运行的；
    2. 一个Tab页(renderer进程)中无论什么时候都只有一个JS线程在运行JS程序；
4. GUI渲染线程与JS引擎线程是互斥的，js引擎线程会阻塞GUI渲染线程
    1. 就是我们常遇到的JS执行时间过长，造成页面的渲染不连贯，导致页面渲染加载阻塞(就是加载慢)；
    2. 例如浏览器渲染的时候遇到<script>标签，就会停止GUI的渲染，然后js引擎线程开始工作，执行里面的js代码，等js执行完毕，js引擎线程停止工作，GUI继续渲染下面的内容。所以如果js执行时间太长就会造成页面卡顿的情况；（所以有了 `defer` 和 `async` 标签）

3.6.3.3. 事件触发线程

1. 属于浏览器而不是JS引擎，用来控制事件循环，并且管理着一个事件队列(task queue)；

2. 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；
3. 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；
4. 因为JS是单线程，所以这些待处理队列中的事件都得排队等待JS引擎处理；

3.6.3.4. 定时触发器线程

1. setInterval与setTimeout所在线程；
2. 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，如果处于阻塞线程状态就会影响记计时的准确)；
3. 通过单独线程来计时并触发定时(计时完毕后，添加到事件触发线程的事件队列中，等待JS引擎空闲后执行)，这个线程就是定时触发器线程，也叫定时器线程；
4. 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；

3.6.3.5. 异步http请求线程

1. 在XMLHttpRequest在连接后是通过浏览器新开一个线程请求；
2. 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；
3. 简单说就是当执行到一个http异步请求时，就把异步请求事件添加到异步请求线程，等收到响应(准确来说应该是http状态变化)，再把回调函数添加到事件队列，等待js引擎线程来执行；

4. 事件循环(Event Loop)基础

JS分为同步任务和异步任务。同步任务都在主线程(这里的主线程就是JS引擎线程)上执行，会形成一个执行栈。

主线程之外，事件触发线程管理着一个任务队列，只要异步任务有了运行结果，就在任务队列之中放一个事件回调。一旦执行栈中的所有同步任务执行完毕(也就是JS引擎线程空闲了)，系统就会读取任务队列，将可运行的异步任务(任务队列中的事件回调，只要任务队列中有事件回调，就说明可以执行)添加到执行栈中，开始执行

我们来看一段简单的代码：

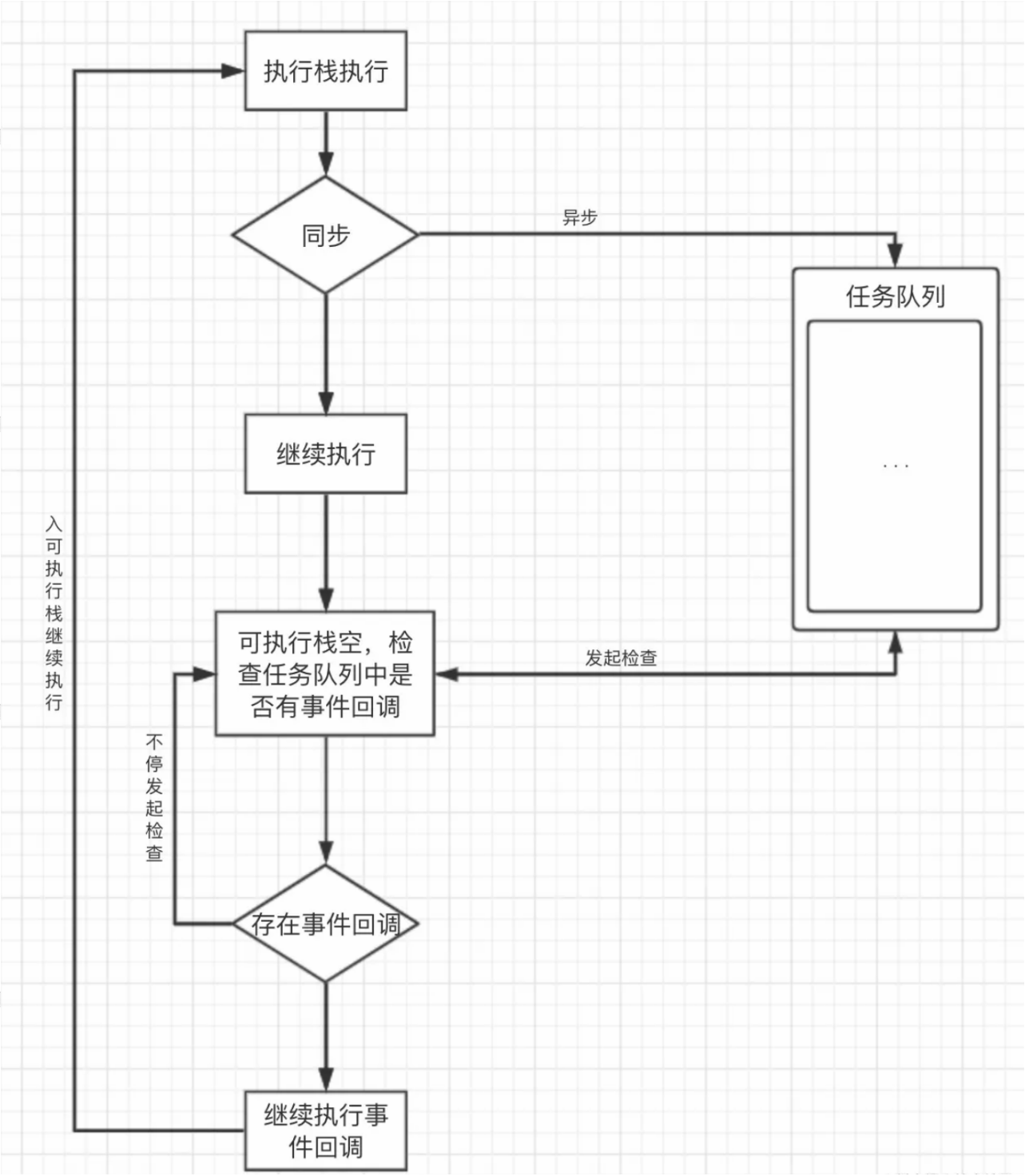
```
let setTimeoutCallBack = function() {
  console.log('');
};
let httpCallback = function() {
  console.log('http');
}

//
console.log('1');

//
setTimeout(setTimeoutCallBack,1000);

// http
ajax.get('/info',httpCallback);

//
console.log('2');
```

以上代码的执行过程：

1. JS是按照顺序从上往下依次执行的，可以先理解为这段代码时的执行环境就是主线程，也就是也就是当前执行栈；
2. 首先，执行 `console.log('我是同步任务1')` ；
3. 接着，执行到 `setTimeout` 时，会移交给定时器线程，通知定时器线程 1s 后将 `setTimeoutCallBack` 这个回调交给事件触发线程处理，在 1s 后事件触发线程会收到 `setTimeoutCallBack` 这个回调并把它加入到事件触发线程所管理的事件队列中等待执行；
4. 接着，执行http请求，会移交给异步http请求线程发送网络请求，请求成功后将 `httpCallback` 这个回调交由事件触发线程处理，事件触发线程收到 httpCallback 这个回调后把它加入到事件触发线程所管理的事件队列中等待执行；
5. 再接着执行 `console.log('我是同步任务2')` ；
6. 至此主线程执行栈中执行完毕，JS引擎线程已经空闲，开始向事件触发线程发起询问，询问事件触发线程的事件队列中是否有需要执行的回调函数，如果有将事件队列中的回调事件加入执行栈中，开始执行回调，如果事件队列中没有回调，JS引擎线程会一直发起询问，直到有为止；

可以发现：

1. 定时触发线程只管理定时器且只关注定时不关心结果，定时结束就把回调扔给事件触发线程；
2. 异步http请求线程只管理http请求同样不关心结果，请求结束把回调扔给事件触发线程；
3. 事件触发线程只关心异步回调入事件队列；
4. JS引擎线程只会执行执行栈中的事件，执行栈中的代码执行完毕，就会读取事件队列中的事件并添加到执行栈中继续执行；
5. 反复执行，就是我们所谓的事件循环(Event Loop)；



1. 执行栈开始顺序执行；
2. 判断是否为同步，异步则进入异步线程，最终事件回调给事件触发线程的任务队列等待执行，同步继续执行；
3. 执行栈空，询问任务队列中是否有事件回调；
4. 任务队列中有事件回调则把回调加入执行栈末尾继续从第一步开始执行；
5. 任务队列中没有事件回调则不停发起询问；

5. 宏任务& 微任务

宏任务 -> GUI渲染 -> 宏任务 -> ... 复制代码

5.1. 宏任务(macrotask)

在ECMAScript中， macrotask 也被称为task。

我们可以将每次执行栈执行的代码当做是一个宏任务(包括每次从事件队列中获取一个事件回调并放到执行栈中执行)， 每一个宏任务会从头到尾执行完毕，不会执行其他、

由于JS引擎线程和GUI渲染线程是互斥的关系，浏览器为了能够使宏任务和DOM任务有序的进行，会在一个宏任务执行结果后，在下一个宏任务执行前，GUI渲染线程开始工作，对页面进行渲染：

```
 -> GUI ->  -> ...
```

常见的宏任务：

1. 主代码块；
2. setTimeout；
3. setInterval；
4. setImmediate () –Node；
5. requestAnimationFrame () –浏览器

5.2. 微任务(microtask)

ES6新引入了Promise标准，同时浏览器实现上多了一个 microtask 微任务概念，在ECMAScript中， microtask 也被称为jobs。

我们已经知道宏任务结束后，会执行渲染，然后执行下一个宏任务， 而微任务可以理解成在当前宏任务执行后立即执行的任务。

当一个宏任务执行完，会在渲染前，将执行期间所产生的所有微任务都执行完：

```
 ->  -> GUI ->  -> ...
```

常见微任务

1. process.nextTick () –Node；
2. Promise.then()；
3. catch；
4. finally；
5. Object.observe；
6. MutationObserver；

5.3. 区分宏任务&微任务

打开新的空白窗口，在console中输入以下代码

```
window.open();

document.body.style = 'background:black';
document.body.style = 'background:red';
```

```
document.body.style = 'background:blue';
document.body.style = 'background:pink';
```

背景直接渲染了粉红色，根据上文里讲浏览器会先执行完一个宏任务，再执行当前执行栈的所有微任务，然后移交GUI渲染，上面四行代码均属于同一次宏任务，全部执行完才会执行渲染，渲染时GUI线程会将所有UI改动优化合并，所以视觉上，只会看到页面变成粉红色。

```
document.body.style = 'background:blue';
setTimeout(()=>{
    document.body.style = 'background:black'
},200)
```

页面会先卡一下蓝色，再变成黑色背景。之所以会卡一下蓝色，是因为以上代码属于两次宏任务，第一次宏任务执行的代码是将背景变成蓝色，然后触发渲染，将页面变成蓝色，再触发第二次宏任务将背景变成黑色。

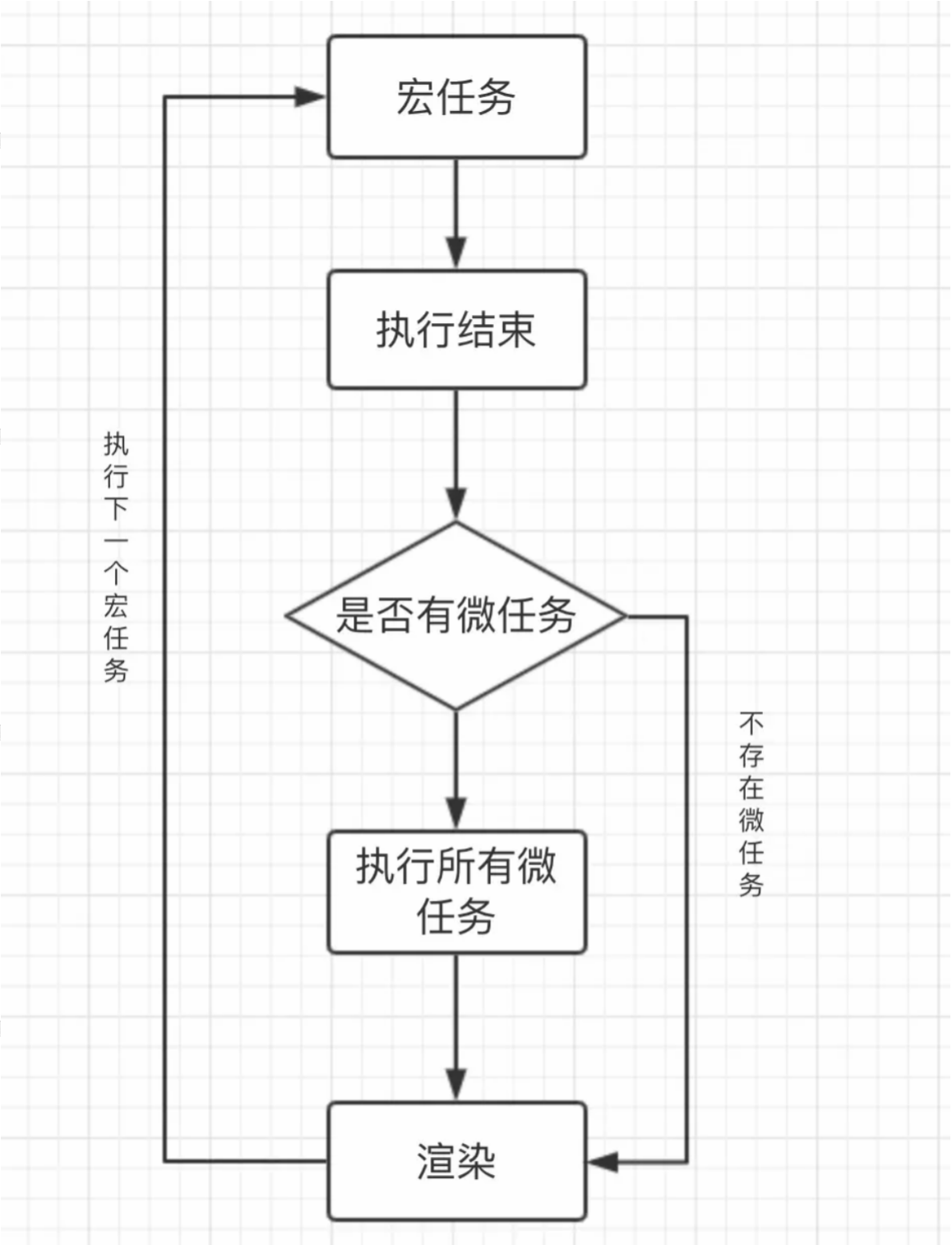
```
document.body.style = 'background:blue'
console.log(1);
Promise.resolve().then(()=>{
    console.log(2);
    document.body.style = 'background:pink'
});
console.log(3);
```

输出 1 3 2 , 是因为 promise 对象的 then 方法的回调函数是异步执行，所以 2 最后输出

页面的背景色直接变成粉色，没有经过蓝色的阶段，是因为，我们在宏任务中将背景设置为蓝色，但在进行渲染前执行了微任务， 在微任务中将背景变成了粉色，然后才执行的渲染。

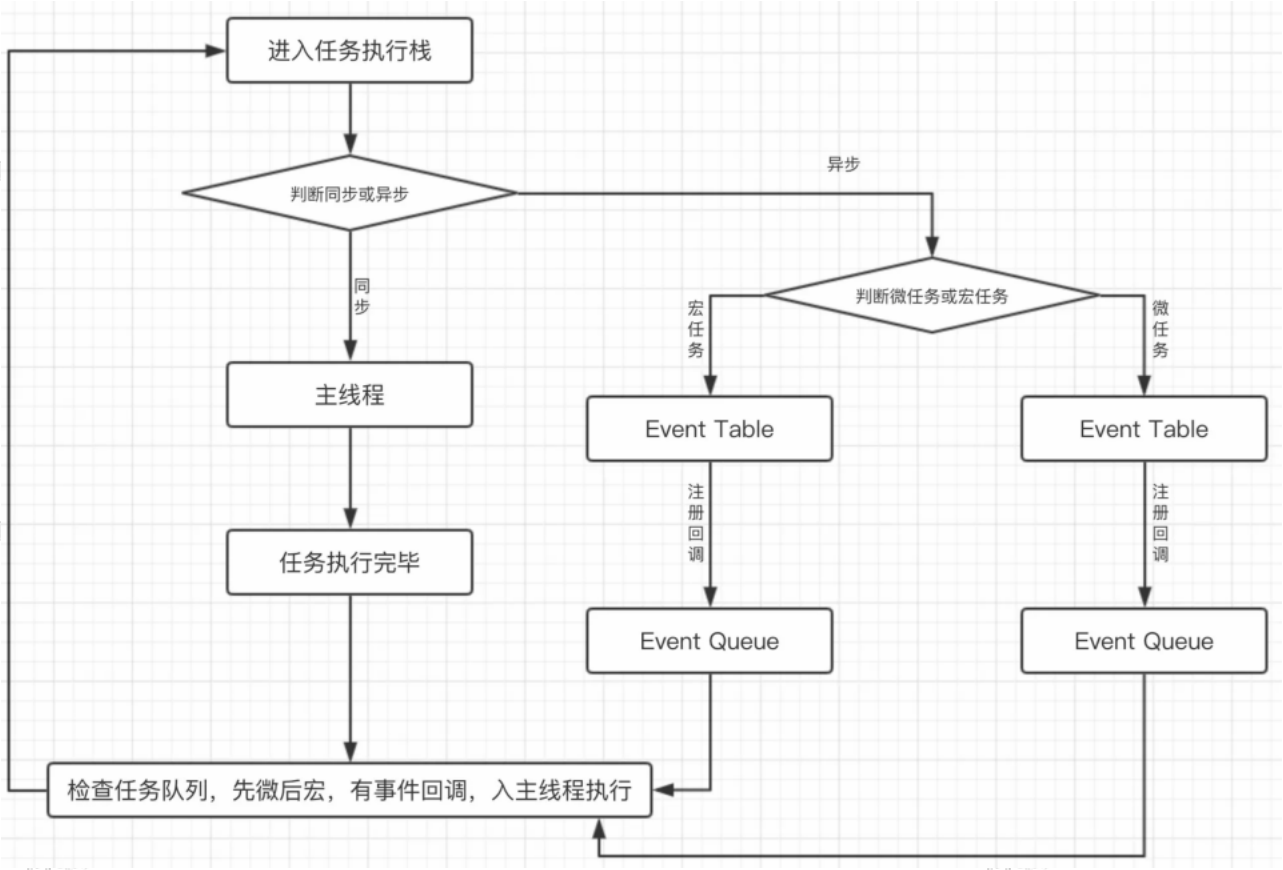
5.4. 注意点

1. 浏览器会先执行一个宏任务，紧接着执行当前执行栈产生的微任务，再进行渲染，然后再执行下一个宏任务；
2. 微任务和宏任务不在一个任务队列，不在一个任务队列：
    1. 例如 `setTimeout` 是一个宏任务，它的事件回调在宏任务队列， `Promise.then()` 是一个微任务，它的事件回调在微任务队列，二者并不是一个任务队列；
    2. 以Chrome 为例，有关渲染的都是在渲染进程中执行，渲染进程中的任务（DOM树构建，js解析…等等）需要主线程执行的任务都会在主线程中执行，而浏览器维护了一套事件循环机制，主线程上的任务都会放到消息队列中执行，主线程会循环消息队列，并从头部取出任务进行执行，如果执行过程中产生其他任务需要主线程执行的，渲染进程中的其他线程会把该任务塞入到消息队列的尾部，消息队列中的任务都是宏任务；
    3. 微任务是如何产生的呢？当执行到script脚本的时候，js引擎会为全局创建一个执行上下文，在该执行上下文中维护了一个微任务队列，当遇到微任务，就会把微任务回调放在微队列中，当所有的js代码执行完毕，在退出全局上下文之前引擎会去检查该队列，有回调就执行，没有就退出执行上下文，这也就是为什么微任务要早于宏任务，也是大家常说的，每个宏任务都有一个微任务队列（由于定时器是浏览器的API，所以定时器是宏任务，在js中遇到定时器会也是放入到浏览器的队列中）；



1. 首先执行一个宏任务，执行结束后判断是否存在微任务；
2. 有微任务先执行所有的微任务，再渲染，没有微任务则直接渲染；
3. 然后再接着执行下一个宏任务；

6. 完整的Event Loop



1. 整体的script(作为第一个宏任务)开始执行的时候，会把所有代码分为同步任务、异步任务两部分，同步任务会直接进入主线程依次执行，异步任务会再分为宏任务和微任务；
2. 宏任务进入到 `Event Table` 中，并在里面注册回调函数，每当指定的事件完成时， `Event Table` 会将这个函数移到 `Event Queue` 中；
3. 微任务也会进入到另一个 `Event Table` 中，并在里面注册回调函数，每当指定的事件完成时， `Event Table` 会将这个函数移到 `Event Queue` 中；
4. 当主线程内的任务执行完毕，主线程为空时，会检查微任务的 `Event Queue` ，如果有任务，就全部执行，如果没有就执行下一个宏任务；
5. 上述过程会不断重复，这就是Event Loop；

7. Promise&async/aweait

7.1. Promise

`new Promise(() => {}).then()` 中，前面的 new Promise() 这一部分是一个构造函数，这是一个同步任务，后面的 .then() 才是一个异步微任务：

```
new Promise((resolve) => {
        console.log(1)
  resolve()
}).then(()=>{
        console.log(2)
})
console.log(3)
// 1 3 2
```

7.2. async/await 函数

async/await本质上还是基于Promise的一些封装，而Promise是属于微任务的一种

所以在使用await关键字与Promise.then效果类似，await 以前的代码，相当于与 new Promise 的同步代码，await 以后的代码相当于 Promise.then的异步

```
setTimeout(() => console.log(4))

async function test() {
  console.log(1)
  await Promise.resolve()
  console.log(3)
}

test()

console.log(2)
// 1 2 3 4
```

7.3. demo

```
function test() {
  console.log(1)
  setTimeout(function () {         // timer1
    console.log(2)
  }, 1000)
}

test();

setTimeout(function () {                 // timer2
  console.log(3)
})

new Promise(function (resolve) {
  console.log(4)
  setTimeout(function () {         // timer3
    console.log(5)
  }, 100)
  resolve()
}).then(function () {
  setTimeout(function () {         // timer4
    console.log(6)
  }, 0)
  console.log(7)
})

console.log(8)

// 14873652
```

1. JS是顺序从上而下执行；
2. 执行到test()，test方法为同步，直接执行 `console.log(1)` 打印1；

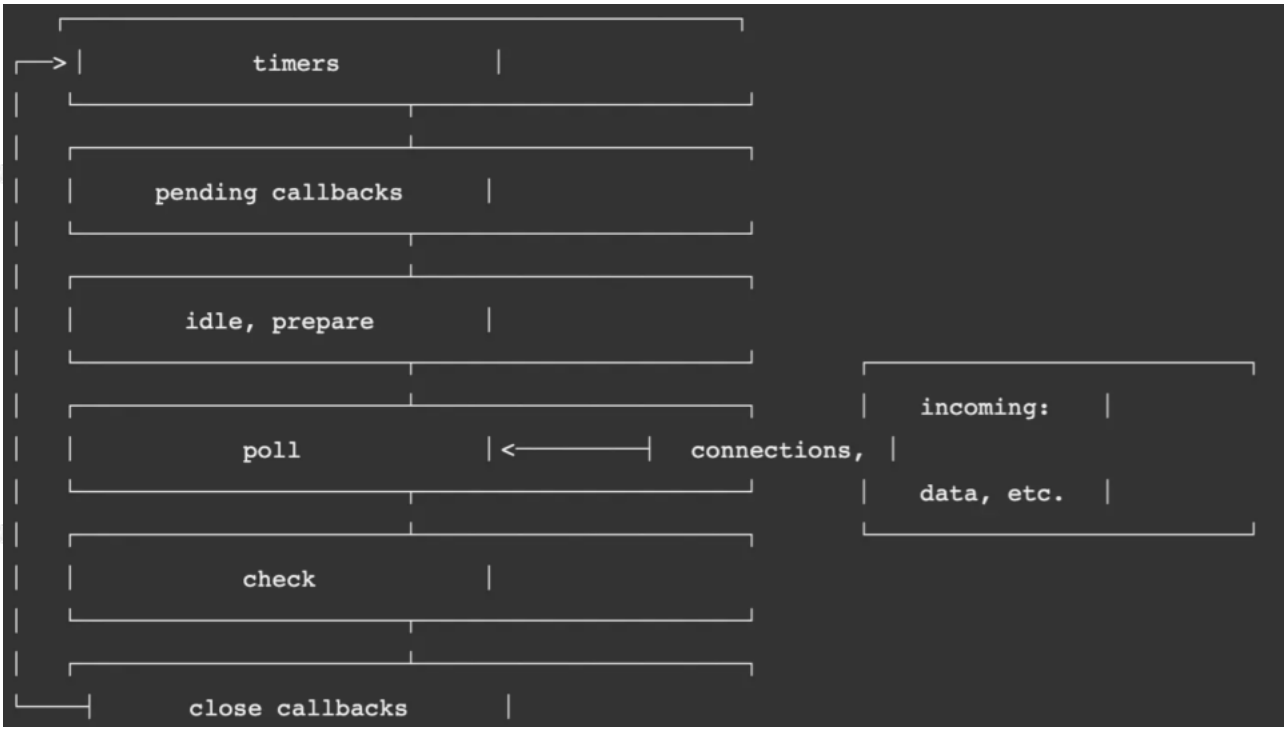
3. test方法中 `setTimeout` 为异步宏任务，回调我们把它记做timer1放入宏任务队列；
4. test方法下面有一个 `setTimeout` 为异步宏任务，回调我们把它记做timer2放入宏任务队列；
5. 执行promise， `new Promise` 是同步任务，直接执行，打印4；
6. `new Promise` 里面的 `setTimeout` 是异步宏任务，回调我们记做timer3放到宏任务队列；
7. `Promise.then` 是微任务，放到微任务队列；
8. console.log(8)是同步任务，直接执行，打印8；
9. 主线程任务执行完毕，检查微任务队列中有 `Promise.then` ；
10. 开始执行微任务，发现有 `setTimeout` 是异步宏任务，记做timer4放到宏任务队列；
11. 微任务队列中的 `console.log(7)` 是同步任务，直接执行，打印7；
12. 微任务执行完毕，第一次循环结束；
13. 检查宏任务队列，里面有timer1、timer2、timer3、timer4，四个定时器宏任务，按照定时器延迟时间得到可以执行的顺序，即Event Queue：timer2、timer4、timer3、timer1，依次拿出放入执行栈末尾执行；
14. 执行timer2， `console.log(3)` 为同步任务，直接执行，打印3；
15. 检查没有微任务，第二次Event Loop结束；
16. 执行timer4， `console.log(6)` 为同步任务，直接执行，打印6；
17. 检查没有微任务，第三次Event Loop结束；
18. 执行timer3， `console.log(5)` 同步任务，直接执行，打印5；
19. 检查没有微任务，第四次Event Loop结束；
20. 执行timer1， `console.log(2)` 同步任务，直接执行，打印2；
21. 检查没有微任务，也没有宏任务，第五次Event Loop结束；

8. NodeJS中的运行机制

虽然NodeJS中的JavaScript运行环境也是V8，也是单线程，但是，还是有一些与浏览器中的表现是不一样的。

其实nodejs与浏览器的区别，就是nodejs的宏任务分好几种类型，而这好几种又有不同的任务队列，而不同的任务队列又有顺序区别，而微任务是穿插在每一种宏任务之间的。

在node环境下，process.nextTick的优先级高于Promise，可以简单理解为在宏任务结束后会先执行微任务队列中的nextTickQueue部分，然后才会执行微任务中的Promise部分。



NodeJS的Event Loop：

1. Node会先执行所有类型为 timers 的 `MacroTask`，然后执行所有的 `MicroTask`（`NextTick` 例外）；
2. 进入 poll 阶段，执行几乎所有 `MacroTask`，然后执行所有的 `MicroTask`；
3. 再执行所有类型为 check 的 `MacroTask`，然后执行所有的 `MicroTask`；
4. 再执行所有类型为 `close callbacks` 的 `MacroTask`，然后执行所有的 `MicroTask`；
5. 至此，完成一个 Tick，回到 timers 阶段，重复执行；