事件循环

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。为了协调事件、用户交互、脚本、UI 渲染和网络处理等行为，防止主线程的不阻塞，Event Loop 的方案应用而生。Event Loop 包含两类：一类是基于 Browsing Context，一种是基于 Worker。二者的运行是独立的，也就是说，每一个 JavaScript 运行的"线程环境"都有一个独立的 Event Loop，每一个 Web Worker 也有一个独立的 Event Loop。

任务队列

根据规范，事件循环是通过任务队列的机制来进行协调的。一个 Event Loop 中，可以有一个或者多个任务队列(task queue)，一个任务队列便是一系列有序任务(task)的集合；每个任务都有一个任务源(task source)，源自同一个任务源的 task 必须放到同一个任务队列，从不同源来的则被添加到不同队列。setTimeout/Promise 等API便是任务源，而进入任务队列的是他们指定的具体执行任务。

在事件循环中，每进行一次循环操作称为 tick，每一次 tick 的任务处理模型是比较复杂的，但关键步骤如下：

在此次 tick 中选择最先进入队列的任务(oldest task)，如果有则执行(一次)

检查是否存在 Microtasks，如果存在则不停地执行，直至清空 Microtasks Queue

更新 render

主线程重复执行上述步骤

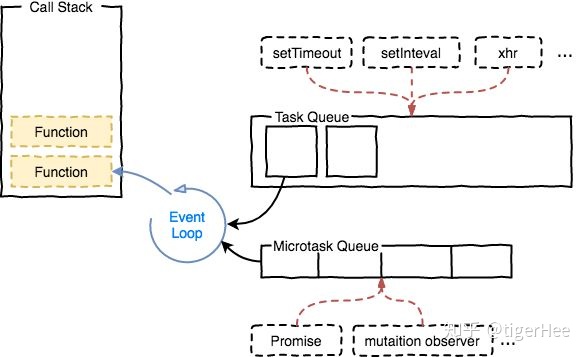
在上诉tick的基础上需要了解几点：

JS分为同步任务和异步任务

同步任务都在主线程上执行，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置一个事件。

一旦执行栈中的所有同步任务执行完毕（此时JS引擎空闲），系统就会读取任务队列，将可运行的异步任务添加到可执行栈中，开始执行。



宏任务

宏任务

(macro)task，可以理解是每次执行栈执行的代码就是一个宏任务（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）。

浏览器为了能够使得JS内部(macro)task与DOM任务能够有序的执行，会在一个(macro)task执行结束后，在下一个(macro)task 执行开始前，对页面进行重新渲染，流程如下：

(macro)task->渲染->(macro)task->...

宏任务包含：

script(整体代码)

setTimeout

setInterval

I/O

UI交互事件

postMessage

MessageChannel

setImmediate(Node.js 环境)

微任务

microtask,可以理解是在当前 task 执行结束后立即执行的任务。也就是说，在当前task任务后，下一个task之前，在渲染之前。

所以它的响应速度相比setTimeout（setTimeout是task）会更快，因为无需等渲染。也就是说，在某一个macrotask执行完后，就会将在它执行期间产生的所有microtask都执行完毕（在渲染前）。

(macro)task->(micro)task->渲染->(macro)task->...

微任务包含：

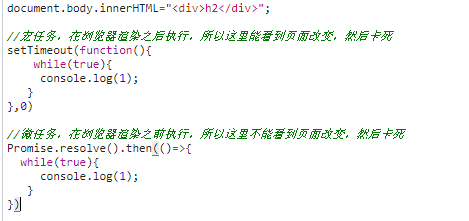
Promise.then

Object.observe

MutaionObserver

process.nextTick(Node.js 环境)

宏任务和微任务与渲染页面之间的顺序



运行机制

在事件循环中，每进行一次循环操作称为 tick，每一次 tick 的任务处理模型是比较复杂的，但关键步骤如下：

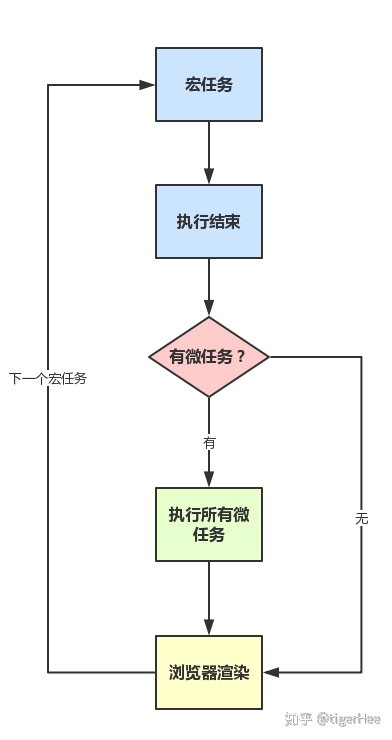
执行一个宏任务（栈中没有就从事件队列中获取）

执行过程中如果遇到微任务，就将它添加到微任务的任务队列中

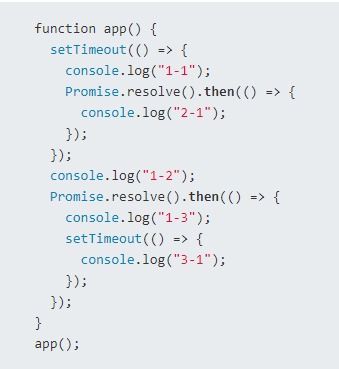
宏任务执行完毕后，立即执行当前微任务队列中的所有微任务（依次执行）

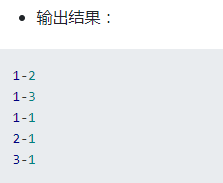
当前宏任务执行完毕，开始检查渲染，然后GUI线程接管渲染

渲染完毕后，JS线程继续接管，开始下一个宏任务（从事件队列中获取）



面试题：





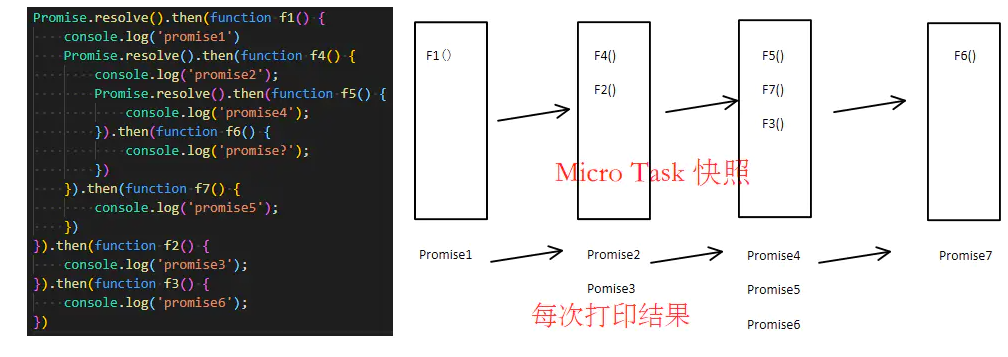


Promise嵌套then和多级then的解析

Promise链式调用then，只会先拿到第一个then扔到微任务里面去执行。因为第二个then要等到第一个then Resovle之后才会执行。当第一个then微任务执行完之后，再把第二个then扔到微任务里去。如果then里面有嵌套的promise，则会形成嵌套效果，如下



先看执行图



接下来是分析

最开始代码执行，遇到Promise，直接执行，将回调函数F1扔进了Micro Task中。执行栈为空，开始执行Micro Task中的代码，为第一个快照。

执行函数F1，打印出Promise1，执行Promise.resolve()，将函数F4扔进了Micro Task中；此时状态已更改为resolve，将then中的函数F2扔进Micro Task，为第二个快照。

执行函数F4，打印出promise2，执行Promise.resolve()，将函数F5扔进了Micro Task中；F4执行完毕，状态更改，将函数F7扔进Micro Task中。

执行函数F2，打印出promise3,状态更改，将函数F3扔进了扔进了Micro Task中，为第三个快照。

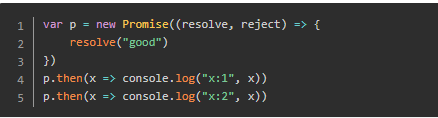
执行函数F5，打印出promise4, 状态更改，将函数F6扔进了Micro Task中；

执行函数F7，打印出promise5；

执行函数F3，打印出promise6，，为第四个快照。

执行函数F3，打印出promise?，结束。

如果不是链式调用，则没有这种嵌套的结果。



p这个promise注册了两个then，注意跟链式调用then的区别，这个就像发布订阅模式，有两个监听者，resolve时，这两个then同时执行