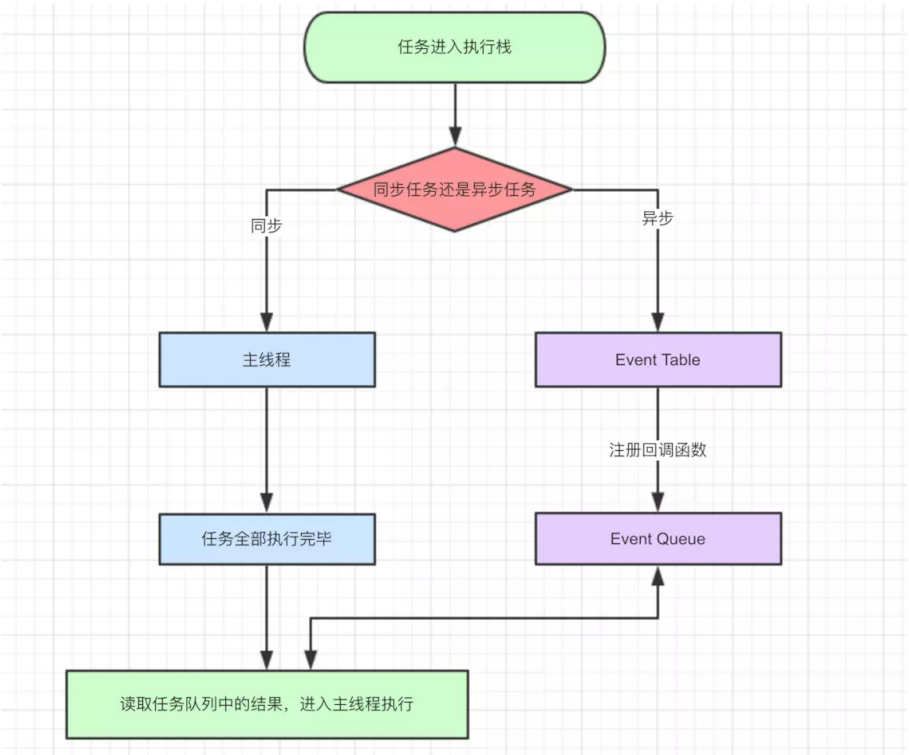
1. JS的调用堆栈



- 同步任务进入主线程，异步任务进入Event Table，并注册函数。
- 当异步任务完成时，Event Table将这个函数移入Event Queue。
- 主线程任务执行完毕，会去Event Queue读取函数进入主线程。
- 上述任务不断重复，即常说的Event Loop（事件循环）。

```
let data = [];
$.ajax({
    url:www.javascript.com,
    data:data,
    success:() => {
        console.log('发送成功!');
    }
})
console.log('代码执行结束');
```

上面是一段简易的 `ajax` 请求代码：

- ajax进入Event Table，注册回调函数 `success`。
- 执行 `console.log('代码执行结束')`。
- ajax事件完成，回调函数 `success` 进入Event Queue。
- 主线程从Event Queue读取回调函数 `success` 并执行。

```javascript
setTimeout(() => {
    task()
},3000)

sleep(10000000)
```

乍一看其实差不多嘛，但我们把这段代码在chrome执行一下，却发现控制台执行 `task()` 需要的时间远远超过3秒，说好的延时三秒，为啥现在需要这么长时间啊?

这时候我们需要重新理解 `setTimeout` 的定义。我们先说上述代码是怎么执行的：

- `task()` 进入Event Table并注册,计时开始。
- 执行 `sleep` 函数，很慢，非常慢，计时仍在继续。
- 3秒到了，计时事件 `timeout` 完成， `task()` 进入Event Queue，但是 `sleep` 也太慢了吧，还没执行完，只好等着。
- `sleep` 终于执行完了， `task()` 终于从Event Queue进入了主线程执行。

上述的流程走完，我们知道 `setTimeout` 这个函数，是经过指定时间后，把要执行的任务(本例中为 `task()`)加入到Event Queue中，又因为是单线程任务要一个一个执行，如果前面的任务需要的时间太久，那么只能等着，导致真正的延迟时间远远大于3秒。

我们还经常遇到 `setTimeout(fn,0)` 这样的代码，0秒后执行又是什么意思呢? 是不是可以立即执行呢?

答案是不会的， `setTimeout(fn,0)` 的含义是，指定某个任务在主线程最早可得的空闲时间执行，意思就是不用再等多少秒了，只要主线程执行栈内的同步任务全部执行完成，栈为空就马上执行。

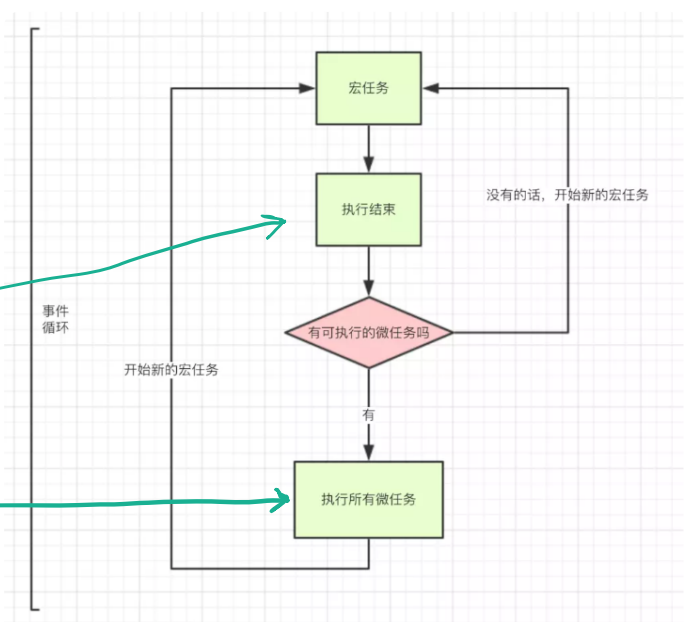
Macro task (宏任务)　整体代码script. setTimeout. setInterval.

Micro task (微任务)　Promise. process.nextTick.

```javascript
setTimeout(function() {
    console.log('setTimeout');
})

new Promise(function(resolve) {
    console.log('promise');
}).then(function() {
    console.log('then');
})

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



- 这段代码作为宏任务，进入主线程。
- 先遇到 `setTimeout` ，那么将其回调函数注册后分发到宏任务Event Queue。(注册过程与上同，下文不再描述)
- 接下来遇到了 `Promise` ， `new Promise` 立即执行， `then` 函数分发到微任务Event Queue。
- 遇到 `console.log()` ，立即执行。
- 好啦，整体代码script作为第一个宏任务执行结束，看看有哪些微任务?我们发现了 `then` 在微任务Event Queue里面，执行。
- ok，第一轮事件循环结束了，我们开始第二轮循环，当然要从宏任务Event Queue开始。我们发现了宏任务Event Queue中 `setTimeout` 对应的回调函数，立即执行。
- 结束。

```
console.log('1');

setTimeout(function() {
    console.log('2');
    process.nextTick(function() {
        console.log('3');
    })
    new Promise(function(resolve) {
        console.log('4');
        resolve();
    }).then(function() {
        console.log('5')
    })
})
process.nextTick(function() {
    console.log('6');
})
new Promise(function(resolve) {
    console.log('7');
    resolve();
}).then(function() {
    console.log('8')
})

setTimeout(function() {
    console.log('9');
    process.nextTick(function() {
        console.log('10');
    })
    new Promise(function(resolve) {
        console.log('11');
        resolve();
    }).then(function() {
        console.log('12')
    })
})
```

第一轮事件循环流程分析如下：

- 整体script作为第一个宏任务进入主线程，遇到 `console.log` ，输出1。
- 遇到 `setTimeout` ，其回调函数被分发到宏任务Event Queue中。我们暂且记为 `setTimeout1` 。
- 遇到 `process.nextTick()` ，其回调函数被分发到微任务Event Queue中。我们记为 `process1` 。
- 遇到 `Promise` ， `new Promise` 直接执行，输出7。 `then` 被分发到微任务Event Queue中。我们记为 `then1` 。
- 又遇到了 `setTimeout` ，其回调函数被分发到宏任务Event Queue中，我们记为 `setTimeout2` 。

宏任务Event Queue	微任务Event Queue
setTimeout1	process1
setTimeout2	then1

- 上表是第一轮事件循环宏任务结束时各Event Queue的情况，此时已经输出了1和7。
- 我们发现了 `process1` 和 `then1` 两个微任务。
- 执行 `process1` ,输出6。
- 执行 `then1` ，输出8。

好了，第一轮事件循环正式结束，这一轮的结果是输出1，7，6，8。那么第二轮时间循环从 `setTimeout1` 宏任务开始：

- 首先输出2。接下来遇到了 `process.nextTick()` ，同样将其分发到微任务Event Queue中，记为 `process2` 。 `new Promise` 立即执行输出4， `then` 也分发到微任务Event Queue中，记为 `then2` 。

宏任务Event Queue	微任务Event Queue
setTimeout2	process2
	then2

- 第二轮事件循环宏任务结束，我们发现有 `process2` 和 `then2` 两个微任务可以执行。
- 输出3。
- 输出5。
- 第二轮事件循环结束，第二轮输出2，4，3，5。
- 第三轮事件循环开始，此时只剩setTimeout2了，执行。
- 直接输出9。
- 将 `process.nextTick()` 分发到微任务Event Queue中。记为 `process3` 。
- 直接执行 `new Promise` ，输出11。
- 将 `then` 分发到微任务Event Queue中，记为 `then3` 。

宏任务Event Queue	微任务Event Queue
	process3
	then3

- 第三轮事件循环宏任务执行结束，执行两个微任务 `process3` 和 `then3` 。
- 输出10。
- 输出12。
- 第三轮事件循环结束，第三轮输出9，11，10，12。

整段代码，共进行了三次事件循环，完整的输出为1，7，6，8，2，4，3，5，9，11，10，12。
(请注意，node环境下的事件监听依赖libuv与前端环境不完全相同，输出顺序可能会有误差)

Javascript 是一门单线程语言.

Event Loop 是Javascript的执行机制.