

# 事件循环专题

---

1. 为什么是事件循环 (Event Loop)

2. 事件循环是什么

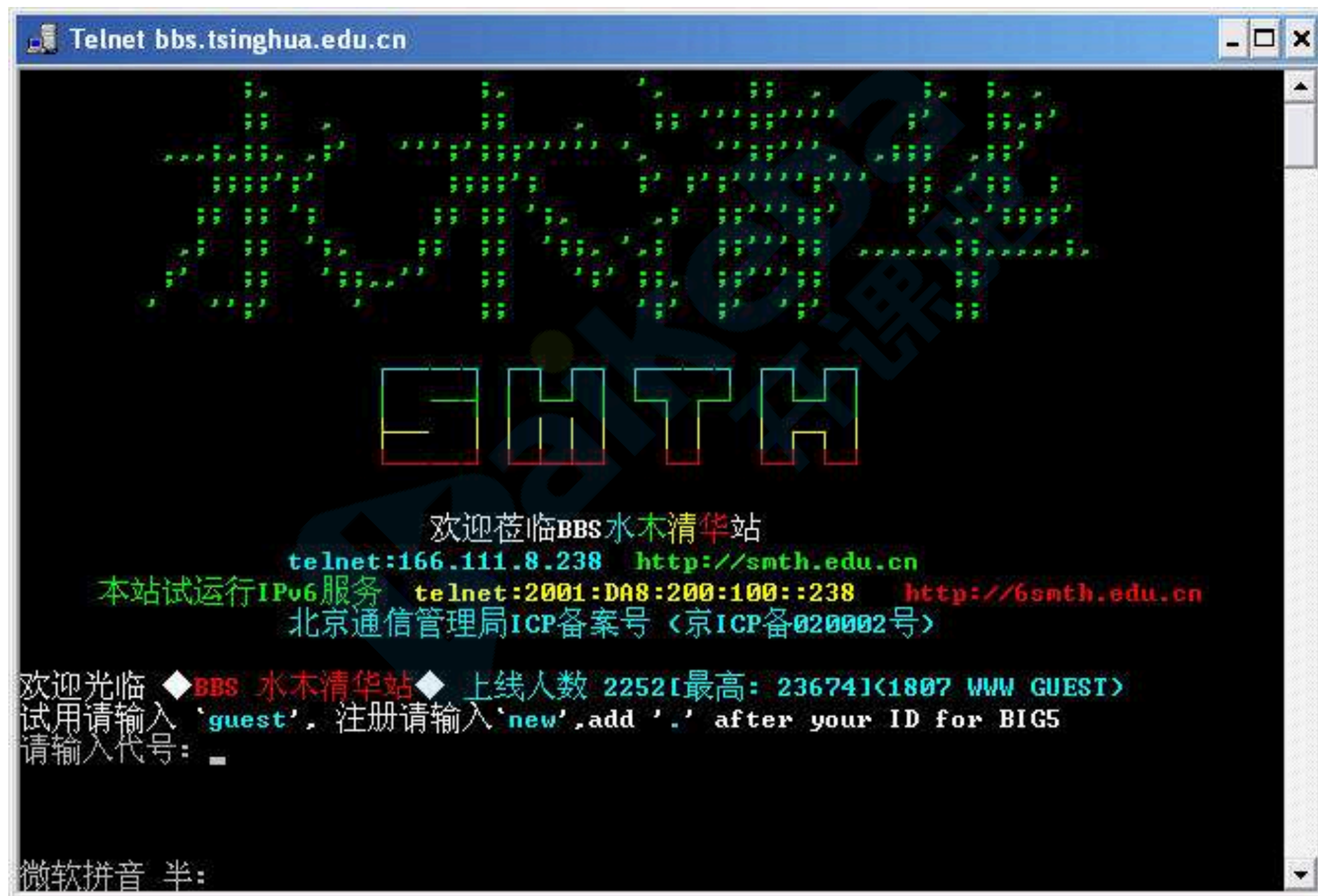
3. 浏览器与 Node.js 的事件循环差异

4. 题目与实例

# 为什么是事件循环

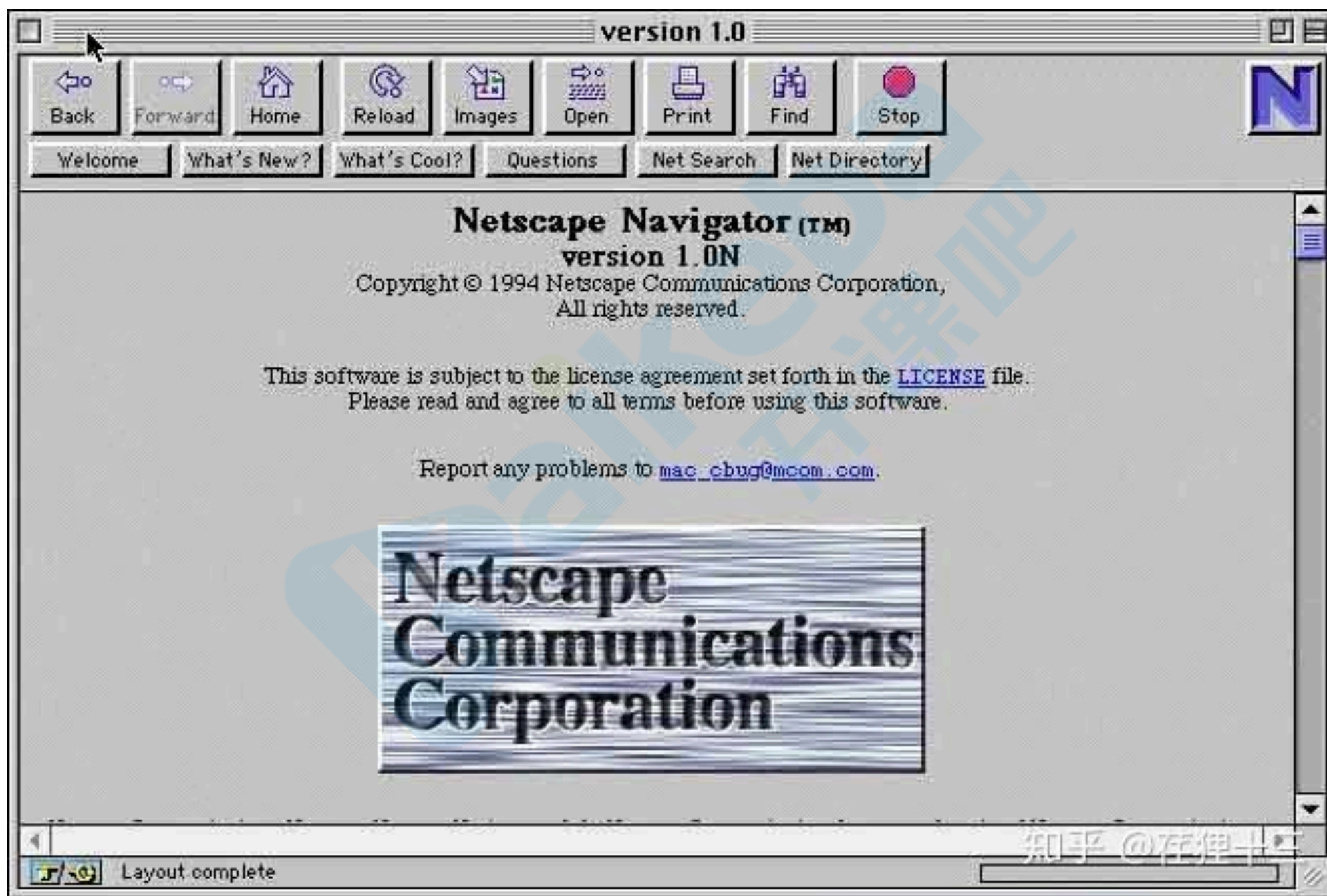


# 为什么是事件循环





# 为什么是事件循环



跨时代的产品 - 第一款浏览器 (1994)

# 为什么是事件循环

---

网景是一家浏览器公司

他们为什么要推出 JavaScript? (1995)

# 为什么是事件循环

---

## 403 Forbidden

You don't have permission to access the URL on this server. Sorry for the inconvenience.  
Please report this message and include the following information to us.  
Thank you very much!

URL: http://www.songtaste.com:8101/  
Server: console1.cn26  
Date: 2014/12/09 10:45:51

---

Powered by Tengine

# 为什么是事件循环

---

JavaScript 是为了服务浏览器上的网页推出的

浏览器选择调度 JavaScript 脚本执行

# 为什么是事件循环

---





# 为什么是事件循环

---

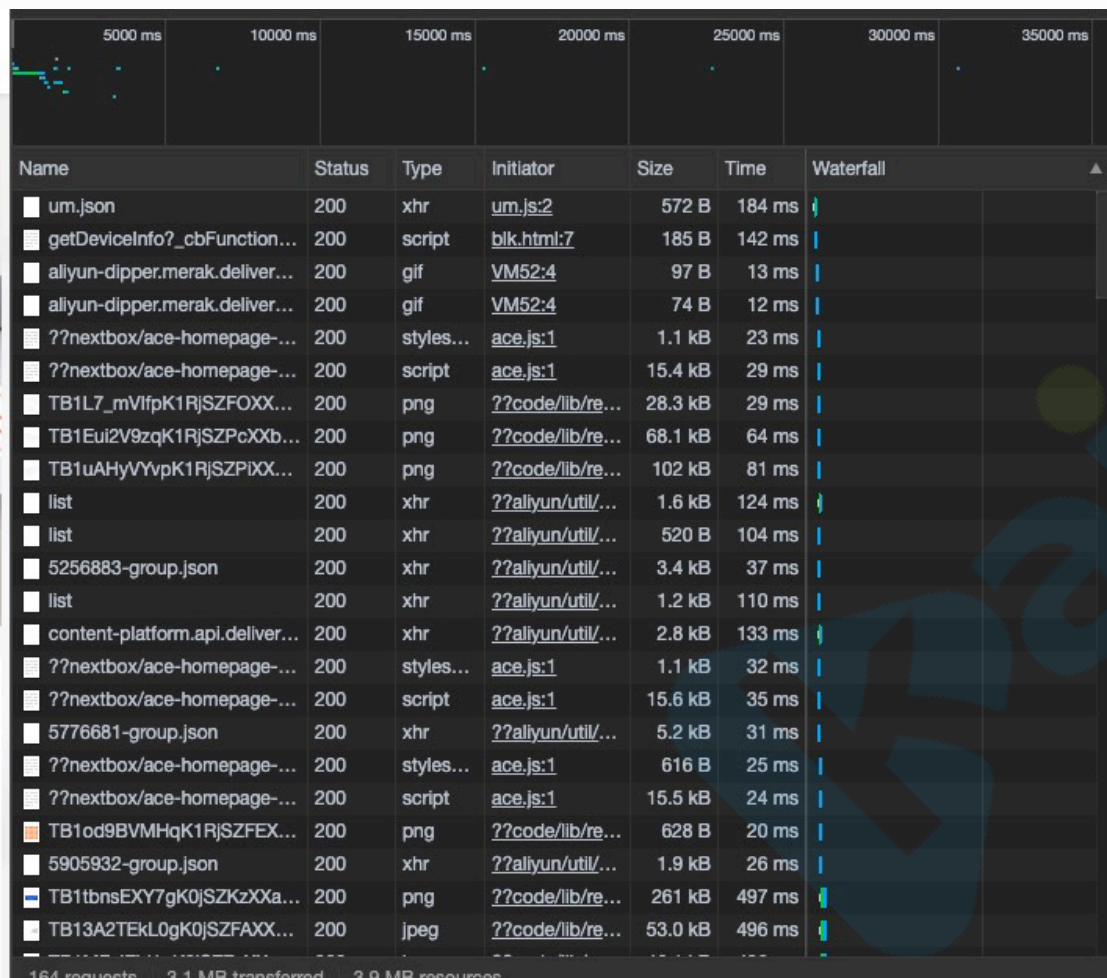
To coordinate events, user interaction, scripts, rendering, networking, and so forth, user agents must use event loops as described in this section. Each agent has an associated event loop, which is unique to that agent.

根据标准中对事件循环的定义描述，我们可以发现事件循环本质上是 user agent (如浏览器端) 用于协调用户交互（鼠标、键盘）、脚本（如 JavaScript）、渲染（如 HTML DOM、CSS 样式）、网络等行为的一个机制。

了解到这个定义之后，我们就能够清楚的明白，与其说是 JavaScript 提供了事件循环，不如说是嵌入 JavaScript 的 user agent 需要通过事件循环来与多种事件源交互。

# 事件循环是什么

打开网页的过程中可能触发非常多的事件



Name	Status	Type	Initiator	Size	Time	Waterfall
um.json	200	xhr	um.js:2	572 B	184 ms	
getDeviceInfo?_cbFunction...	200	script	blk.html:7	185 B	142 ms	
aliyun-dipper.merak.deliver...	200	gif	VM52:4	97 B	13 ms	
aliyun-dipper.merak.deliver...	200	gif	VM52:4	74 B	12 ms	
??nextbox/ace-homepage-...	200	styles...	ace.js:1	1.1 kB	23 ms	
??nextbox/ace-homepage-...	200	script	ace.js:1	15.4 kB	29 ms	
TB1L7_mVlfpK1RjSZFOXX...	200	png	??code/lib/re...	28.3 kB	29 ms	
TB1Eui2V9zqK1RjSZPcXXb...	200	png	??code/lib/re...	68.1 kB	64 ms	
TB1uAHyVYvpK1RjSZPiXX...	200	png	??code/lib/re...	102 kB	81 ms	
list	200	xhr	??aliyun/util/...	1.6 kB	124 ms	
list	200	xhr	??aliyun/util/...	520 B	104 ms	
5256883-group.json	200	xhr	??aliyun/util/...	3.4 kB	37 ms	
list	200	xhr	??aliyun/util/...	1.2 kB	110 ms	
content-platform.api.deliver...	200	xhr	??aliyun/util/...	2.8 kB	133 ms	
??nextbox/ace-homepage-...	200	styles...	ace.js:1	1.1 kB	32 ms	
??nextbox/ace-homepage-...	200	script	ace.js:1	15.6 kB	35 ms	
5776681-group.json	200	xhr	??aliyun/util/...	5.2 kB	31 ms	
??nextbox/ace-homepage-...	200	styles...	ace.js:1	616 B	25 ms	
??nextbox/ace-homepage-...	200	script	ace.js:1	15.5 kB	24 ms	
TB1od9BVMHqK1RjSZFEX...	200	png	??code/lib/re...	628 B	20 ms	
5905932-group.json	200	xhr	??aliyun/util/...	1.9 kB	26 ms	
TB1tbnsEXY7gK0JSZKzXXa...	200	png	??code/lib/re...	261 B	497 ms	
TB13A2TEkL0gK0JSZFAXX...	200	jpeg	??code/lib/re...	53.0 kB	496 ms	

164 requests 3.1 MB transferred 3.9 MB resources



# 事件循环是什么

---

大部分事件都是异步的





# 事件循环是什么

---

让事件都进入队列排队



# 事件循环是什么

---

各种浏览器事件同时触发时，肯定有一个先来后到的排队问题。决定这些事件如何排队触发的机制，就是事件循环。这个排队行为以 JavaScript 开发者的角度来看，主要是分成两个队列：

- 一个是 JavaScript 外部的队列。外部的队列主要是浏览器协调的各类事件的队列，标准文件中称之为 **Task Queue**。下文中为了方便理解统一称为**外部队列**。
- 另一个是 JavaScript 内部的队列。这部分主要是 JavaScript 内部执行的任务队列，标准中称之为 **Microtask Queue**。下文中为了方便理解统一称为**内部队列**。

值得注意的是，虽然为了好理解我们管这个叫队列 (Queue)，但是本质上是有序集合 (Set)，因为传统的队列都是先进先出 (FIFO) 的，而这里的队列则不然，排到最前面但是没有满足条件也是不会执行的（比如外部队列里只有一个 `setTimeout` 的定时任务，但是时间还没有到，没有满足条件也不会把他出列来执行）。



# 事件循环是什么

---

## 外部队列

外部队列 (Task Queue [3])，顾名思义就是 JavaScript 外部的事件的队列，这里我们可以先列举一下浏览器中这些外部事件源 (Task Source)，他们主要有：

- DOM 操作 (页面渲染)
- 用户交互 (鼠标、键盘)
- 网络请求 (Ajax 等)
- History API 操作
- 定时器 (setTimeout 等) [4]

可以观察到，这些外部的事件源可能很多，为了方便浏览器厂商优化，HTML 标准中明确指出一个事件循环由一个或多个外部队列，而每一个外部事件源都有一个对应的外部队列。不同事件源的队列可以有不同的优先级（例如在网络事件和用户交互之间，浏览器可以优先处理鼠标行为，从而让用户感觉更加流畅）。

# 事件循环是什么

---

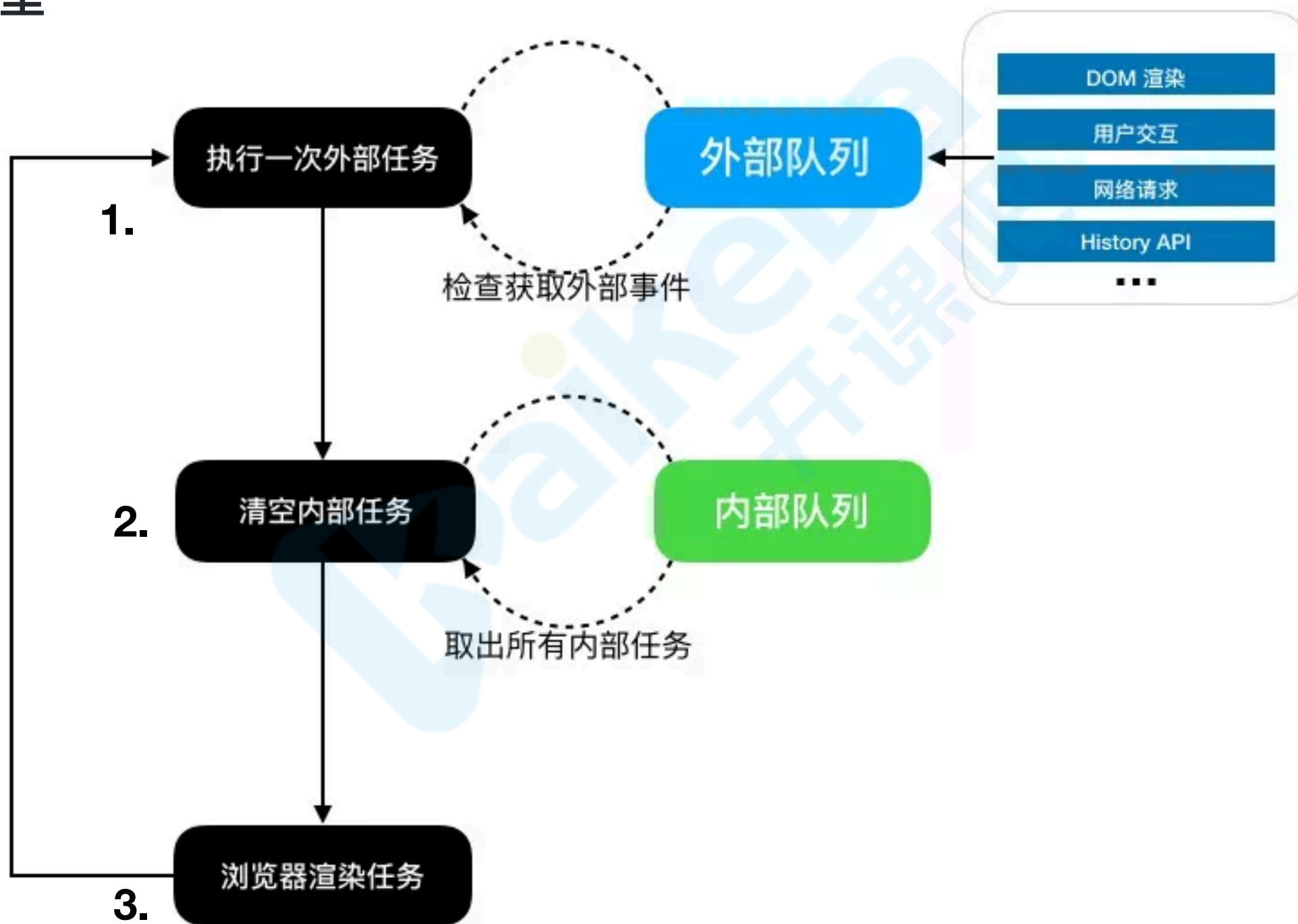
## 内部队列

内部队列（Microtask Queue），即 JavaScript 语言内部的事件队列，在 HTML 标准中，并没有明确规定这个队列的事件源，通常认为有以下几种：

- Promise [5] 的成功 (.then) 与失败 (.catch)
- MutationObserver [6]
- Object.observe (已废弃) [7]

# 事件循环是什么

## 处理模型



# 事件循环是什么

## 案例分析

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

```
setTimeout(function() {  
  console.log('2 setTimeout');  
}, 0);
```

```
Promise.resolve().then(function() {  
  console.log('3 promise1');  
}).then(function() {  
  console.log('4 promise2');  
});
```

```
console.log('5 script end');
```

1	script start
5	script end
3	promise1
4	promise2
2	setTimeout

### 外部队列

执行脚本

1

5

2 setTimeout

### 内部队列

3 promise

4 promise

对应的处理过程则是：

1. 执行 console.log （输出 script start）
2. 遇到 setTimeout 加入外部队列
3. 遇到两个 Promise 的 then 加入内部队列
4. 遇到 console.log 直接执行 （输出 script end）
5. 内部队列中的任务挨个执行完 （输出 promise1 和 promise2）
6. 外部队列中的任务执行 （输出 setTimeout）

# 事件循环是什么

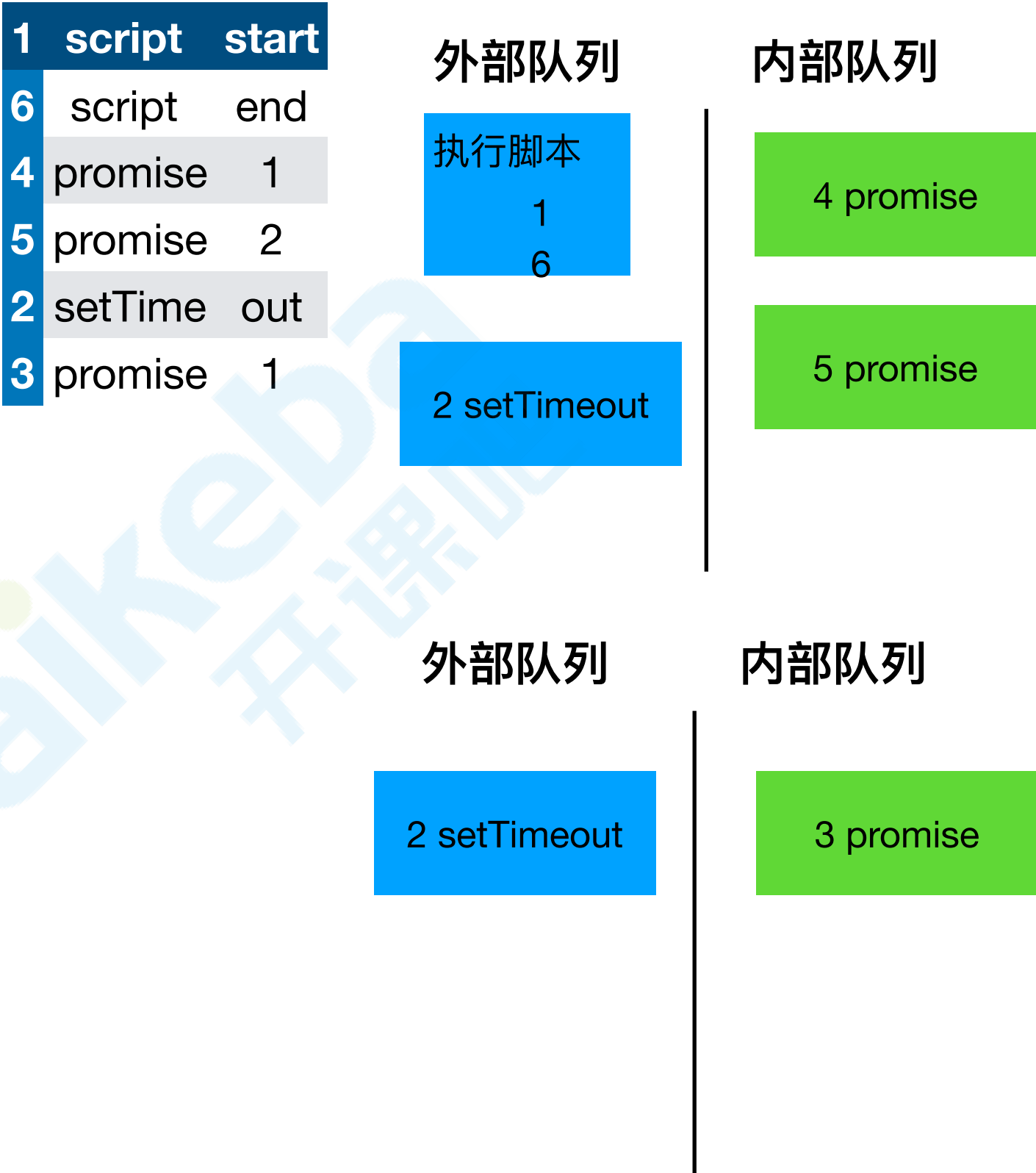
## 案例分析

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

setTimeout(function() {
  console.log('2 setTimeout');
  Promise.resolve().then(function() {
    console.log('3 promise1');
  });
}, 0);

Promise.resolve().then(function() {
  console.log('4 promise1');
}).then(function() {
  console.log('5 promise2');
});

console.log('6 script end');
```





# 事件循环是什么

---

## 案例分析

```
setTimeout(() => {  
  console.log('setTimeout1')  
  // promise  
})
```

```
Promise.resolve().then(() => {  
  console.log('promise1')  
})
```

```
setTimeout(() => {  
  console.log('setTimeout2')  
})
```

```
Promise.resolve().then(() => {  
  console.log('promise2')  
})
```

```
Promise.resolve().then(() => {  
  console.log('promise3')  
})
```

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

# 事件循环是什么

## HTML 案例分析

```
<html>
  <body>
    <pre id="main"></pre>
  </body>
  <script>
    const main = document.querySelector('#main');
    const callback = (i, fn) => () => {
      console.log(i)
      main.innerHTML += fn(i);
    };
    let i = 1;
    while(i++ < 5000) {
      setTimeout(callback(i, (i) => '\n' + i + '<'))
    }

    while(i++ < 10000) {
      Promise.resolve().then(callback(i, (i) => i + ','))
    }
    console.log(i)
    main.innerHTML += '[end ' + i + ']\n'
  </script>
</html>
```

通过这个例子，我们就可以发现，渲染过程很明显分成三个阶段：

1. JavaScript 执行完毕 innerText 首先加上 [end 10001]
2. 内部队列：Promise 的 then 全部任务执行完毕，往 innerText 上追加了很长一段字符串
3. HTML 渲染：1 和 2 追加到 innerText 上的内容同时渲染
4. 外部队列：挨个执行 setTimeout 中追加到 innerText 的内容
5. HTML 渲染：将 4 中的内容渲染。
6. 回到第 4 步走外部队列的流程（内部队列已清空）

# 事件循环是什么

---

## script 事件是外部队列

浏览器

Javascript

Java applet

vbscript

To coordinate events, user interaction, **scripts**, rendering, networking, and so forth, user agents must use event loops as described in this section. Each agent has an associated event loop, which is unique to that agent.

看到这里，大家可能就反应过来了，**scripts 执行也是一个事件**，我们只要归类一下就会发现 JavaScript 的执行也是一个浏览器发起的外部事件。所以本质的执行顺序还是：

1. 一次外部事件
2. 所有内部事件
3. HTML 渲染
4. 回到到 1

# 浏览器与 Node.js 的事件循环差异

---

根据本文开头我们讨论的事件循环起源，很容易理解为什么浏览器与 Node.js 的事件循环会存在差异。如果说浏览端是将 JavaScript 集成到 HTML 的事件循环之中，那么 Node.js 则是将 JavaScript 集成到 libuv 的 I/O 循环之中。

简而言之，二者都是把 JavaScript 集成到他们各自的环境中，但是 HTML (浏览器端) 与 libuv (服务端) 面对的场景有很大的差异。首先能直观感受到的区别是：

1. 事件循环的过程没有 HTML 渲染。只剩下了外部队列和内部队列这两个部分。
2. 外部队列的事件源不同。Node.js 端没有了鼠标等外设但是新增了文件等 IO。
3. 内部队列的事件仅剩下 Promise 的 then 和 catch。

至于内在的差异，有一个很重要的地方是 Node.js (libuv) 在最初设计的时候是允许执行多次外部的事件再切换到内部队列的，而浏览器端一次事件循环只允许执行一次外部事件。这个经典的内在差异，可以通过一个例子来观察。

# 浏览器与 Node.js 的事件循环差异

```
setTimeout(()=>{  
  console.log('timer1');  
  Promise.resolve().then(function() {  
    console.log('promise1');  
  });  
});
```

```
setTimeout(()=>{  
  console.log('timer2');  
  Promise.resolve().then(function() {  
    console.log('promise2');  
  });  
});
```

究其原因，主要是因为浏览器端有外部队列一次事件循环只能执行一个的限制，而在 Node.js 中则放开了这个限制，允许外部队列中所有任务都执行完再切换到内部队列。所以他们的情况对应为：

- 浏览器端

- i. 外部队列：代码执行，两个 timeout 加入外部队列
- ii. 内部队列：空
- iii. 外部队列：第一个 timeout 执行，promise 加入内部队列  
内部队列：执行第一个 promise
- iv. 外部队列：第二个 timeout 执行，promise 加入内部队列
- v. 内部队列：执行第二个 promise

- Node.js 服务端

- i. 外部队列：代码执行，两个 timeout 加入外部队列
- ii. 内部队列：空
- iii. 外部队列：两个 timeout 都执行完
- iv. 内部队列：两个 promise 都执行完



# 浏览器与 Node.js 的事件循环差异

---

虽然 Node.js 的这个问题在 11 之后的版本里修复了，但是为了继续探究这个影响，我们引入一个新的外部事件 `setImmediate`。这个方法目前是 Node.js 独有的，浏览器端没有。

`setImmediate` 的引入是为了解决 `setTimeout` 的精度问题，由于 `setTimeout` 指定的延迟时间是毫秒（ms）但实际一次时间循环的时间可能是纳秒级的，所以在一次事件循环的多个外部队列中，找到某一个队列直接执行其中的 `callback` 可以得到比 `setTimeout` 更早执行的效果。

我们继续以开始的场景构造一个例子，并在 Node.js 10.x 的版本上执行（存在一次事件循环执行多次外部事件）：

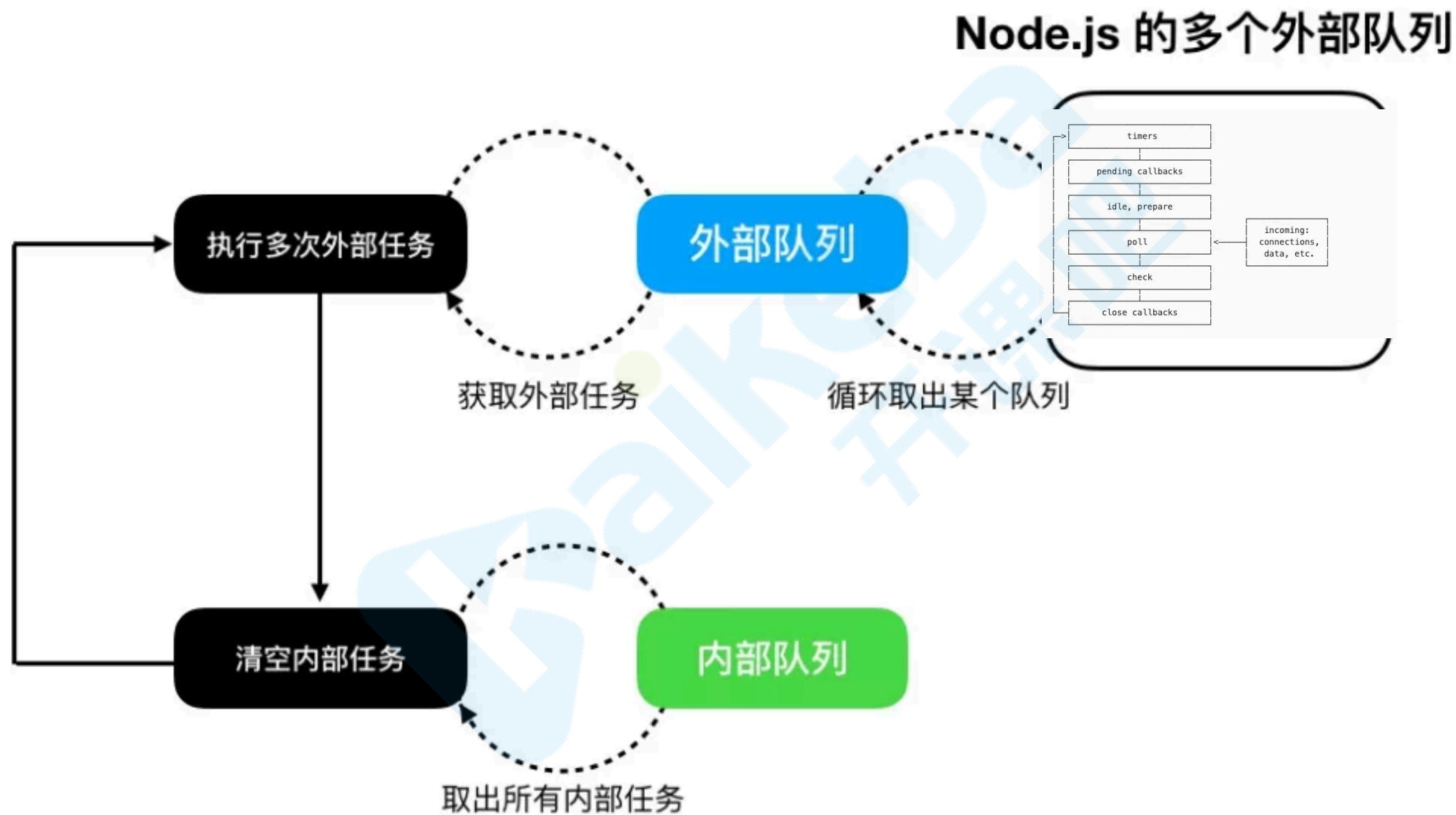
```
setTimeout(()=>{
  console.log('setTimeout1');
  Promise.resolve().then(() => console.log('promise1'));
});
```

```
setTimeout(()=>{
  console.log('setTimeout2');
  Promise.resolve().then(() => console.log('promise2'));
});
```

```
setImmediate() => {
  console.log('setImmediate1');
  Promise.resolve().then(() => console.log('promise3'));
};
```

```
setImmediate() => {
  console.log('setImmediate2');
  Promise.resolve().then(() => console.log('promise4'));
};
```

# 浏览器与 Node.js 的事件循环差异



我们可以推测出 Node.js 中的事件循环与浏览器类似，也是外部队列与内部队列的循环，而 `setImmediate` 在另外一个外部队列中。

# 浏览器与 Node.js 的事件循环差异



该水印由迅读PDF生成，  
如果想去掉该水印，请访问并下载：  
<http://www.pdfxd.com>

## 外部队列

执行脚本

1  
5

2 setTimeout

## 内部队列

3 promise

4 promise

在v11版本之前是先清空外部队列再清空内部队列的，在v11之后和浏览器对齐了，setImmediate发生在外部队列的check阶段

```
setTimeout(()=>{ // 定时器  
  console.log('1 setTimeout1');  
  Promise.resolve().then(() => console.log('2 promise1'));  
});
```

```
setImmediate(() => { // 立执行  
  console.log('3 setImmediate1');  
  Promise.resolve().then(() => console.log('4 promise3'));  
});
```

```
setTimeout(()=>{ // 定时器  
  console.log('5 setTimeout2');  
  Promise.resolve().then(() => console.log('6 promise2'));  
});
```

```
setImmediate(() => { // 立执行  
  console.log('7 setImmediate2');  
  Promise.resolve().then(() => console.log('8 promise4'));  
});
```

# 浏览器与 Node.js 的事件循环差异

---

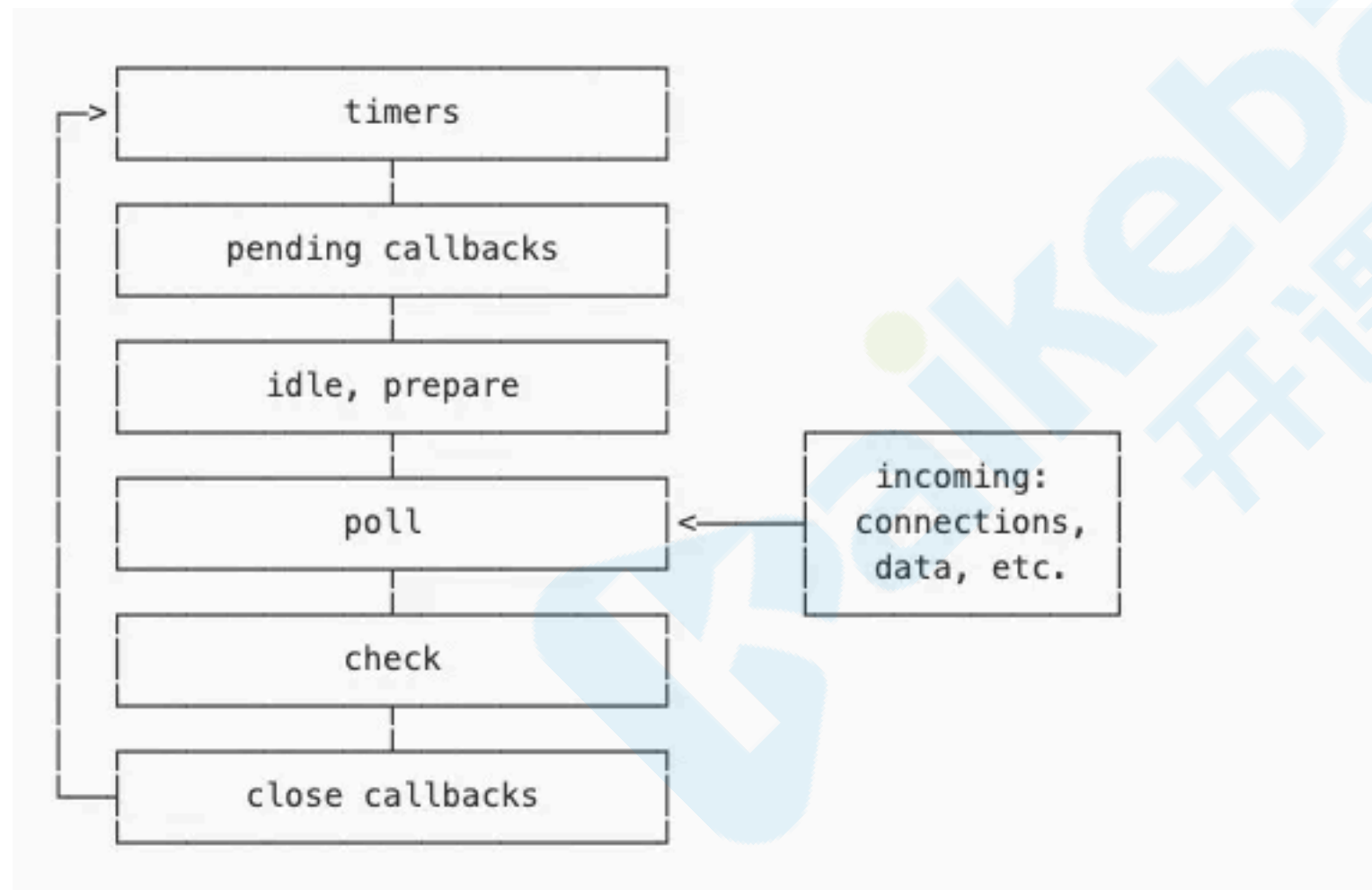
接下来，我们再来看一下当 Node.js 在与浏览器端对齐了事件循环的事件之后，这个例子的执行结果为：

```
setImmediate1  
promise3  
setImmediate2  
promise4  
setTimeout1  
promise1  
setTimeout2  
promise2
```

其中主要有两点需要关注，一是外部列队在每次事件循环只执行了一个，另一个是 Node.js 的固定了多个外部队列的优先级。setImmediate 的外部队列没有执行完的时候，是不会执行 timeout 的外部队列的。

# 浏览器与 Node.js 的事件循环差异

了解了这个点之后，Node.js 的事件循环就变得很简单了，我们可以看下 Node.js 官方文档中对于事件循环顺序的展示：



其中 **check** 阶段是用于执行 **setImmediate** 事件的。结合本文上面的推论我们可以知道，Node.js 官方这个所谓事件循环过程，其实只是完整的事件循环中 Node.js 的多个外部队列相互之间的优先级顺序。



# 浏览器与 Node.js 的事件循环差异

## Output

我们可以再加入一个 poll 阶段的例子来看这个循环：

```
const fs = require('fs');
```

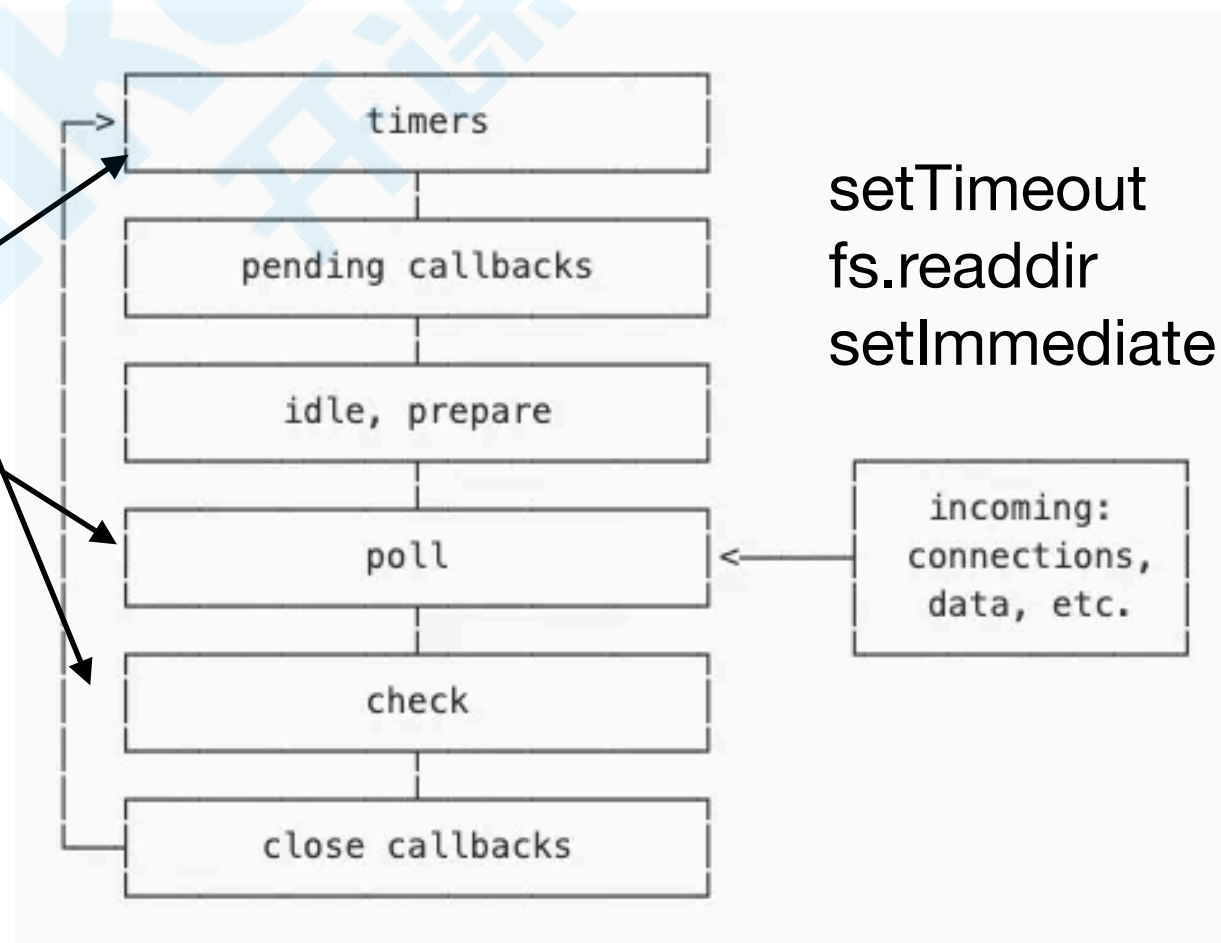
```
setImmediate(() => {  
  console.log('setImmediate');  
});
```

```
fs.readdir(__dirname, () => {  
  console.log('fs.readdir');  
});
```

```
setTimeout(()=>{  
  console.log('setTimeout');  
});
```

```
Promise.resolve().then(() => {  
  console.log('promise');  
});
```

promise  
setTimeout  
fs.readdir  
setImmediate



# 浏览器与 Node.js 的事件循环差异

我们可以再加入一个 poll 阶段的例子来看这个循环：

```
const fs = require('fs');

setImmediate(() => {
  console.log('setImmediate');
});

fs.readdir(__dirname, () => {
  console.log('fs.readdir');
});

setTimeout(()=>{
  console.log('setTimeout');
});

Promise.resolve().then(() => {
  console.log('promise');
});
```

输出结果 (v12.x)：

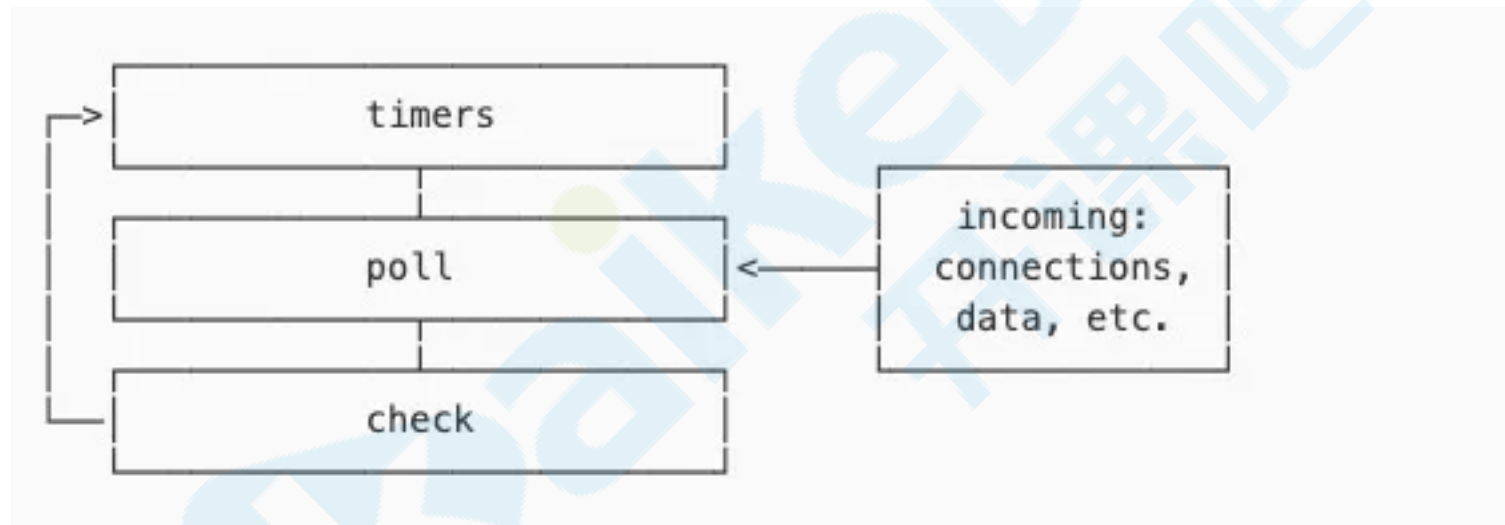
```
promise
setTimeout
fs.readdir
setImmediate
```

根据输出结果，我们可以知道梳理出来：

1. 外部队列：执行当前 script
2. 内部队列：执行 promise
3. 外部队列：执行 setTimeout
4. 内部队列：空
5. 外部队列：执行 fs.readdir
6. 内部队列：空
7. 外部队列：执行 check (setImmediate)

# 浏览器与 Node.js 的事件循环差异

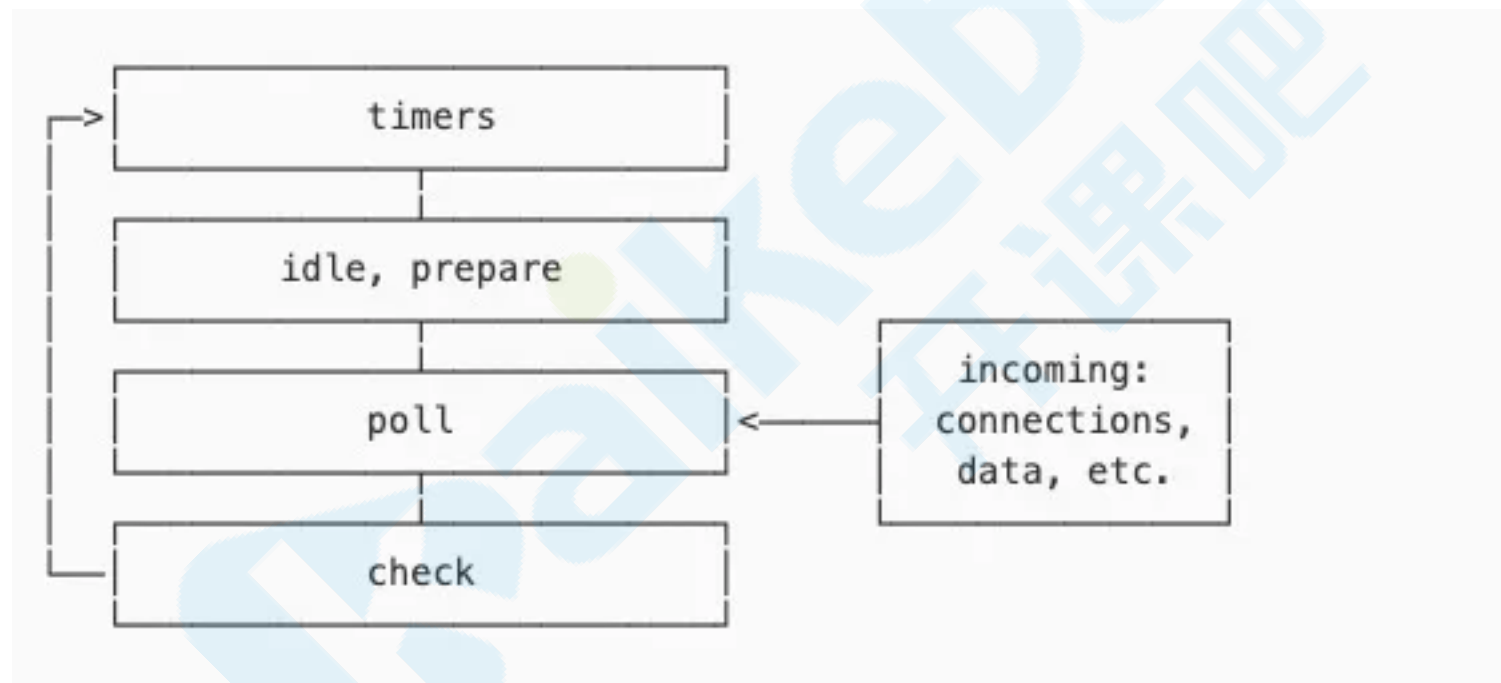
---



timer (setTimeout) 是第一阶段的原因在 libuv 的文档 [8] 中有描述 —— 为了减少时间相关的系统调用 (System Call)。setImmediate 出现在 check 阶段是蹭了 libuv 中 poll 阶段之后的检查过程 (这个过程放在 poll 中也很奇怪, 放在 poll 之后感觉比较合适)。

# 浏览器与 Node.js 的事件循环差异

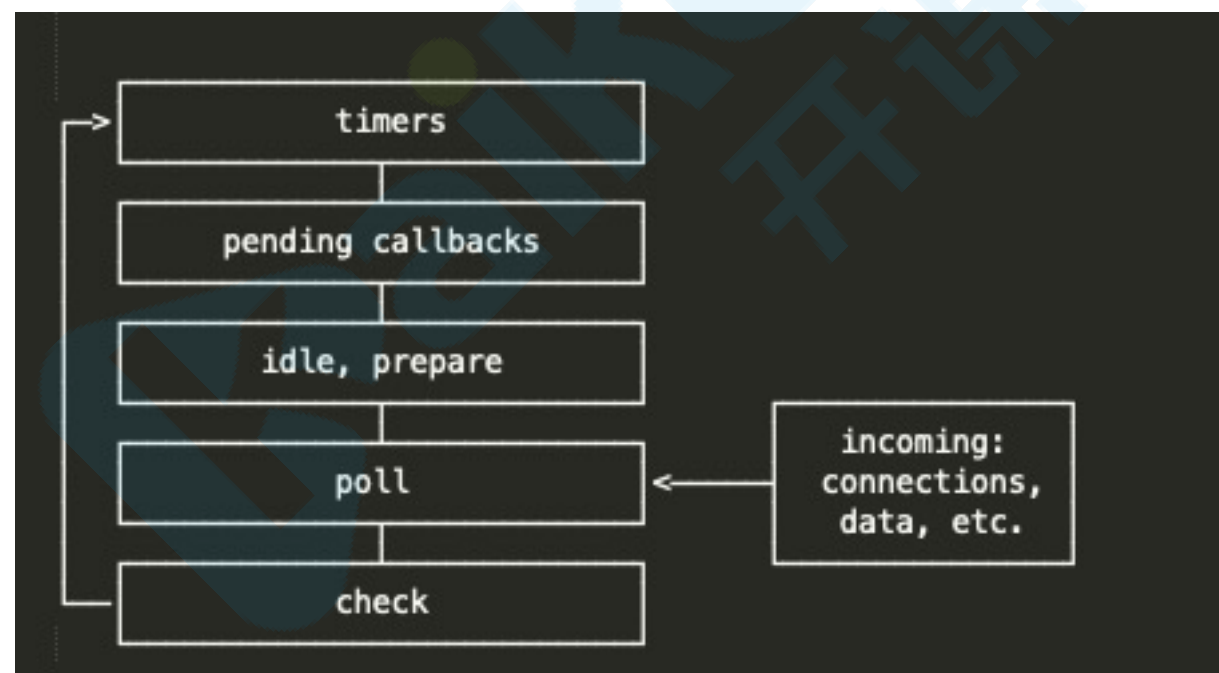
`idle`, `prepare` 对应的是 libuv 中的两个叫做 `idle` [10] 和 `prepare` [11] 的句柄。由于 I/O 的 `poll` 过程可能阻塞住事件循环，所以这两个句柄主要是用来触发 `poll`（阻塞）之前需要触发的回调：



由于 `poll` 可能 `block` 住事件循环，所以应当有一个外部队列专门用于执行 I/O 的 `callback`，并且优先级在 `poll` 以及 `prepare to poll` 之前。

# 浏览器与 Node.js 的事件循环差异

另外我们知道网络 IO 可能有非常多的请求同时进来，如果该阶段如果无限制的执行这些 callback，可能导致 Node.js 的进程卡死该阶段，其他外部队列的代码都没法执行了。所以当前外部队列在执行一定数量的 callback 之后会截断。由于截断的这个特性，这个专门执行 I/O callbacks 的外部队列也叫 pending callbacks：



# 实例与题目

当前位置: 首页 > 模拟考试 > 小车科目一 > 顺序练习

1/1455、驾驶机动车在道路上违反道路交通安全法的行为, 属于什么行为?

★ 收藏

- ☐ A、过失行为
- ☐ B、违规行为
- ☐ C、违章行为
- ☐ D、违法行为

单选题, 请选择你认为正确的答案!

上一题

下一题

显示详解

显示答题卡

☒ 答对自动下一题

答对: 0 题

答错: 0 题

正确率: 0%

[云同步做题进度](#)

<https://www.jiakaobaodian.com/mnks/exercise/0-car-kemu1.html?id=800000>