第七节 Nodejs 事件模块

1.手写events模块

```
function MyEvent() {
  this._event = Object.create(null)
}
MyEvent.prototype.on = function(type,callback) {
  if (this._events[type]) {
    this._events[type].push(callback)
  } else {
    this._events[type] = [callback]
  }
}
MyEvent.prototype.emit = function (type, ...args) {
  if (this._events && this._events[type].length) {
    this._events[type].forEach((callback) => {
      callback.call(this, ...args)
    })
  }
}
MyEvent.prototype.off = function (type, callback) {
  // 判断当前 type 事件监听是否存在，如果存在则取消指定的监听
```

```
  if (this._events && this._events[type]) {
    this._events[type] = this._events[type].filter((item) => {
      return item !== callback && item.link !== callback
    })
  }
}

MyEvent.prototype.once = function(type, callback) {
  let foo = function(...args) {
    callback.call(this,...args)
    this.off(type, foo)
  }
  foo.link = callback
  this.on(type,foo)
}
```

2.浏览器事件循环

完整的事件循环执行顺序

- 从上至下执行所有同步代码
- 执行过程在讲遇到的宏任务与微任务添加到相应的队列
- 同步代码执行完毕之后， 执行满足条件的微任务回调
- 微任务队列执行完毕后，执行所有满足需求的延迟队列的任务回调
- 再次循环上次操作
- 注意： 每执行一个宏任务之后就会立刻检查微任务队列

```
setTimeout(() => {
  console.log('s1')
  Promise.resolve().then(() => {
    console.log('p2')
  })
  Promise.resolve().then(() => {
    console.log('p3')
  })
})

Promise.resolve().then(() => {
  console.log('p1')
  setTimeout(() => {
```

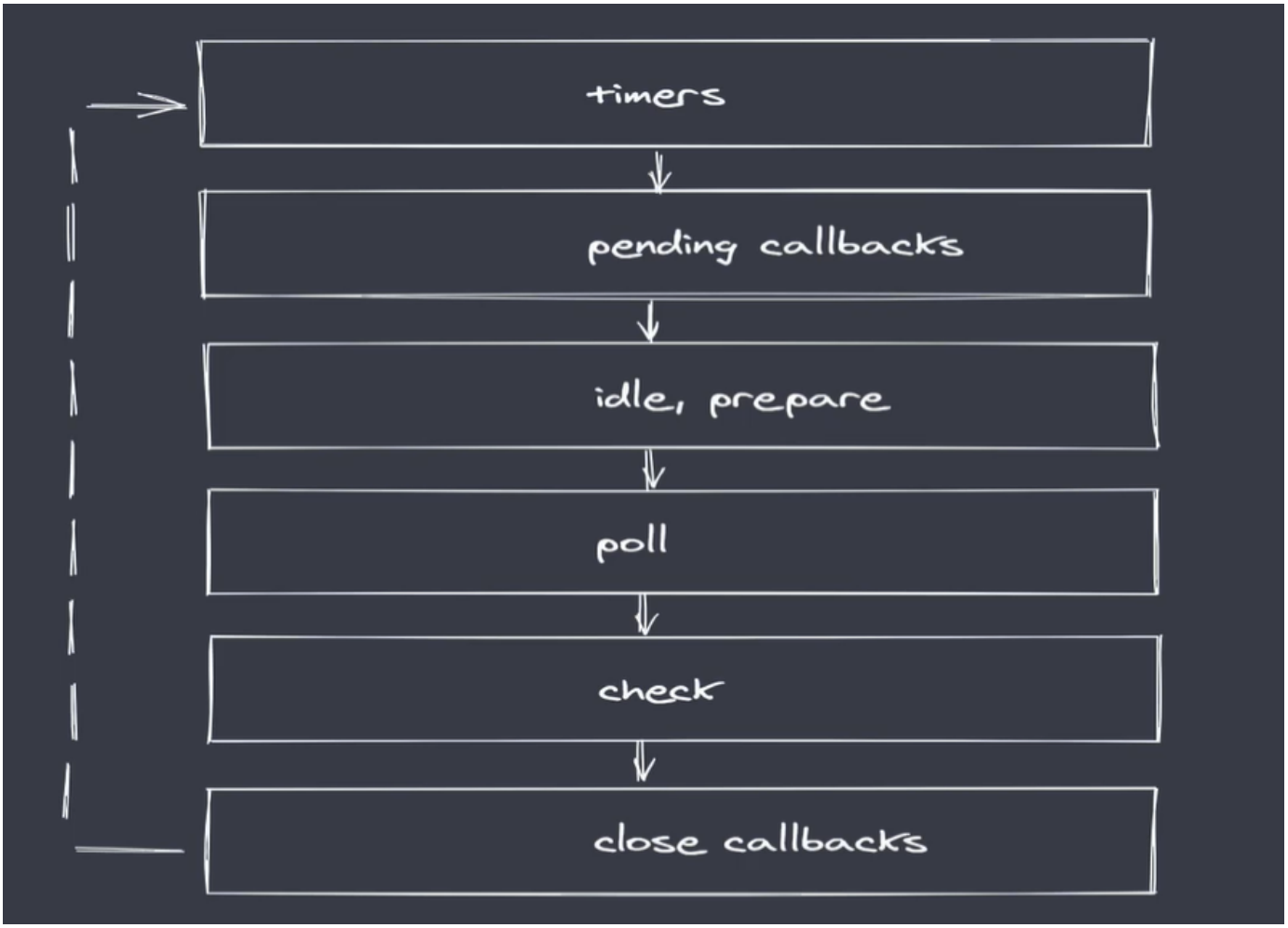
```
    console.log('s2')
  })
  setTimeout(() => {
    console.log('s3')
  })
})

// p1 s1 p2 p3 s2 s3
```

3.Nodejs事件循环机制

3.1 有6个事件循环队列

- timers: 执行setTimeout与setInterval回调
- pednding callbacks: 执行操作系统的回调， 例如 tcp udp
- idle, prepare: 只在系统内容进行使用
- poll: 执行于io相关的回调
- check: 执行setImmediate中的回调
- close callback: 执行close事件的回调



3.2 Nodejs完整事件环

- 执行同步代码将不同的任务添加至相应的度列
- 所有同步代码执行完后会执行满足条件的微任务
- 所有微任务代码执行后会执行timer队列中满足的宏任务
- timer中的所有宏任务执行完成后就会一次切换队列
- 注意： 在完成队列切换之前会先清空微任务代码
- 注意： process.nextTIck是一个node平台的一个微任务， 优先级高于promise等其他微任务

```
setTimeout(() => {
  console.log('s1')
})

Promise.resolve().then(() => {
  console.log('p1')
})

```

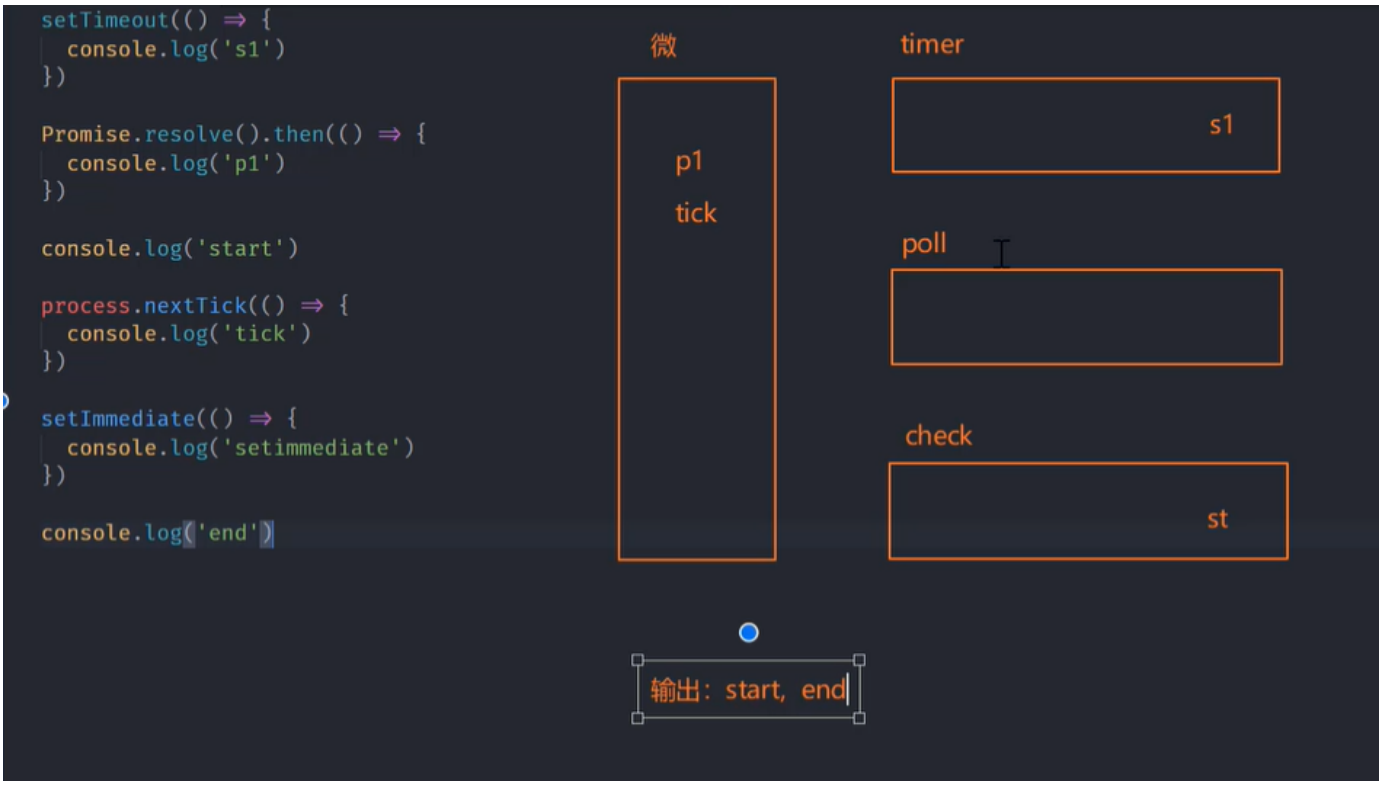
```
console.log('start')

process.nextTick(() => {
  console.log('tick')
})

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

console.log('end')

// start,  end, tick, p1, s1,  st
```



4.Nodejs与浏览器事件循环区别

- 任务队列数是不同的
  - 浏览器中只有3（2）个任务队列
  - Nodejs中有6个事件队列
- Nodejs微任务执行时机不同
  - 二者都会在同步代码执行完毕后执行微任务
  - 浏览器平台下每当一个宏任务执行完毕之后就清空微任务

    - Nodejs平台在事件队列切换时会去清空微任务
    - https://juejin.cn/post/6844903761949753352
- 微任务优先级不同
    - 浏览器事件循环中，微任务存放于微任务队列中， 先进先出
    - Nodejs中process.nextTick先于promise.then

5.Nodejs事件循环常见问题

5.1 setTimeout执行时间有延迟

```
setTimeout(() => {
  console.log('timeout')
}, 0)

setImmediate(() => {
  console.log('immdieate')
})
// 执行循序可能不确定
```

5.2 6个事件队列是从上往下执行的

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

fs.readFile('./m1.js', () => {
  setTimeout(() => {
    console.log('timeout')
  }, 0)

  setImmediate(() => {
    console.log('immdieate')
  })
})
// 因为读取文件的回调会放到poll队列中， 所以像一个事件队列是check
// immdieate timeout
```