link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=132 sentences=413, words=2773

1. 进程和线程

1.1 进程

- CPU承担了所有的计算任务
- 进程是CPU资源分配的最小单位
- 在同一个时间内，单个CPU只能执行一个任务，只能运行一个进程
- 如果有一个进程正在执行，其它进程就得暂停
- CPU使用了时间片轮转的算法实现多进程的调度

1.2. 线程

- 线程是 CPU调度的最小单位
- 一个进程可以包括多个线程，这些线程共享这个进程的资源

2. chrome浏览器进程

- 浏览器是多进程的
- 每一个TAB页就是一个进程
- `&#x6D4F;&#x89C8;&#x5668;&#x4E3B;&#x8FDB;&#x7A0B;`
    - 控制其它子进程的创建和销毁
    - 浏览器界面显示，比如用户交互、前进、后退等操作
    - 将渲染的内容绘制到用户界面上
- `&#x6E32;&#x67D3;&#x8FDB;&#x7A0B;`就是我们说的浏览器内核
    - 负责页面的渲染、脚本执行、事件处理
    - 每个TAB页都有一个渲染进程
    - 每个渲染进程中有主线程和合成线程等
- `&#x7F51;&#x7EDC;&#x8FDB;&#x7A0B;` 处理网络请求、文件访问等操作
- GPU进程 用于3D绘制
- 第三方插件进程

3. 渲染进程

- GUI渲染线程
    - 渲染、布局和绘制页面
    - 当页面需要重绘和回流时，此线程就会执行
    - 与JS引擎互斥
- JS引擎线程
    - 负责解析执行JS脚本
    - 只有一个JS引擎线程(单线程)
    - 与GUI渲染线程互斥
- 事件触发线程
    - 用来控制事件循环(鼠标点击、setTimeout、Ajax等)
    - 当事件满足触发条件时，把事件放入到JS引擎所有的执行队列中
- 定时器触发线程
    - setInterval和setTimeout所在线程
    - 定时任务并不是由JS引擎计时，而是由定时触发线程来计时的
    - 计时完毕后会通知事件触发线程
- 异步HTTP请求线程
    - 浏览器有一个单独的线程处理AJAX请求
    - 当请求完毕后，如果有回调函数，会通知事件触发线程
- IO线程
    - 接收其它进程发过来的消息

4. 事件环

4.1. 单线程顺序执行

- 无法和外界进行交互

4.1.1 main.js

```
(function mainThread() {
    let task1 = 1 + 2;
    let task2 = 2 + 3;
    let task3 = 3 + 4;
    console.log(task1, task2, task3);
})()
```

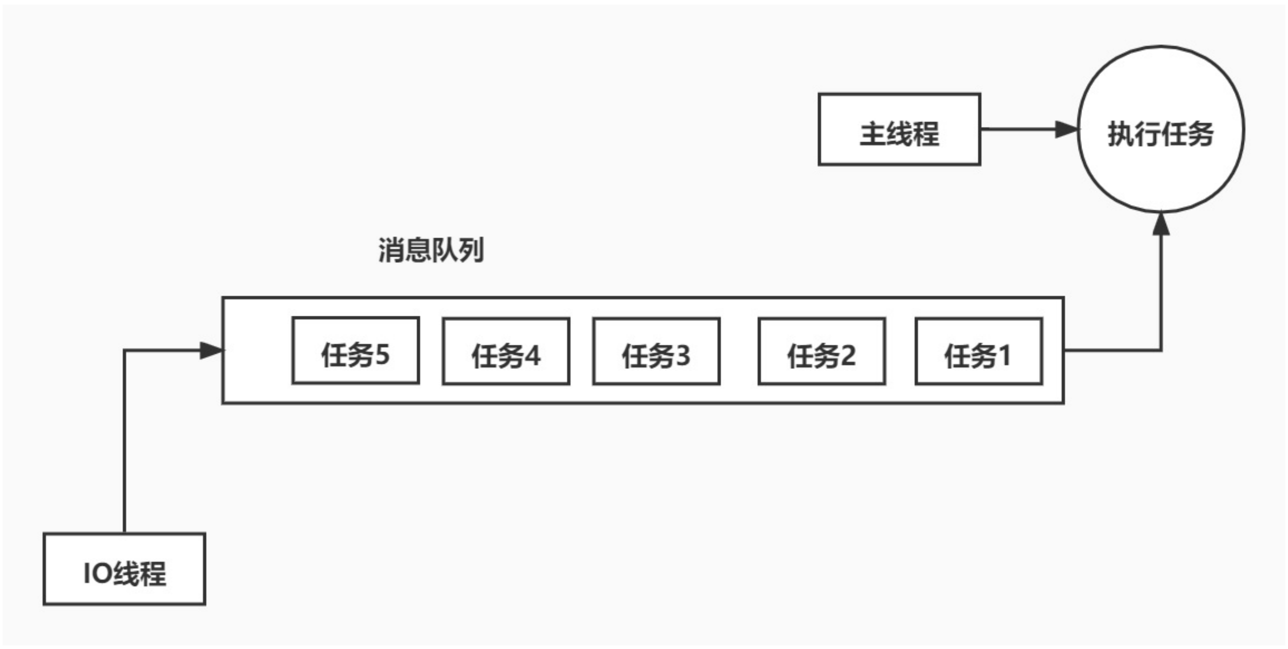
4.2.事件循环

- 引入事件循环可以不断接收新的任务并执行
- 引入输入事件可以接收用户的输入

4.2.1 main.js

```
let readline = require('readline-sync');
(function mainThread() {
    while (true) {
        var num1 = readline.question('num1: ');
        var num2 = readline.question('num2: ');
        let ret = eval(num1 + "+" + num2);
        console.log(ret);
    }
})();
```

4.3. 消息队列+IO线程



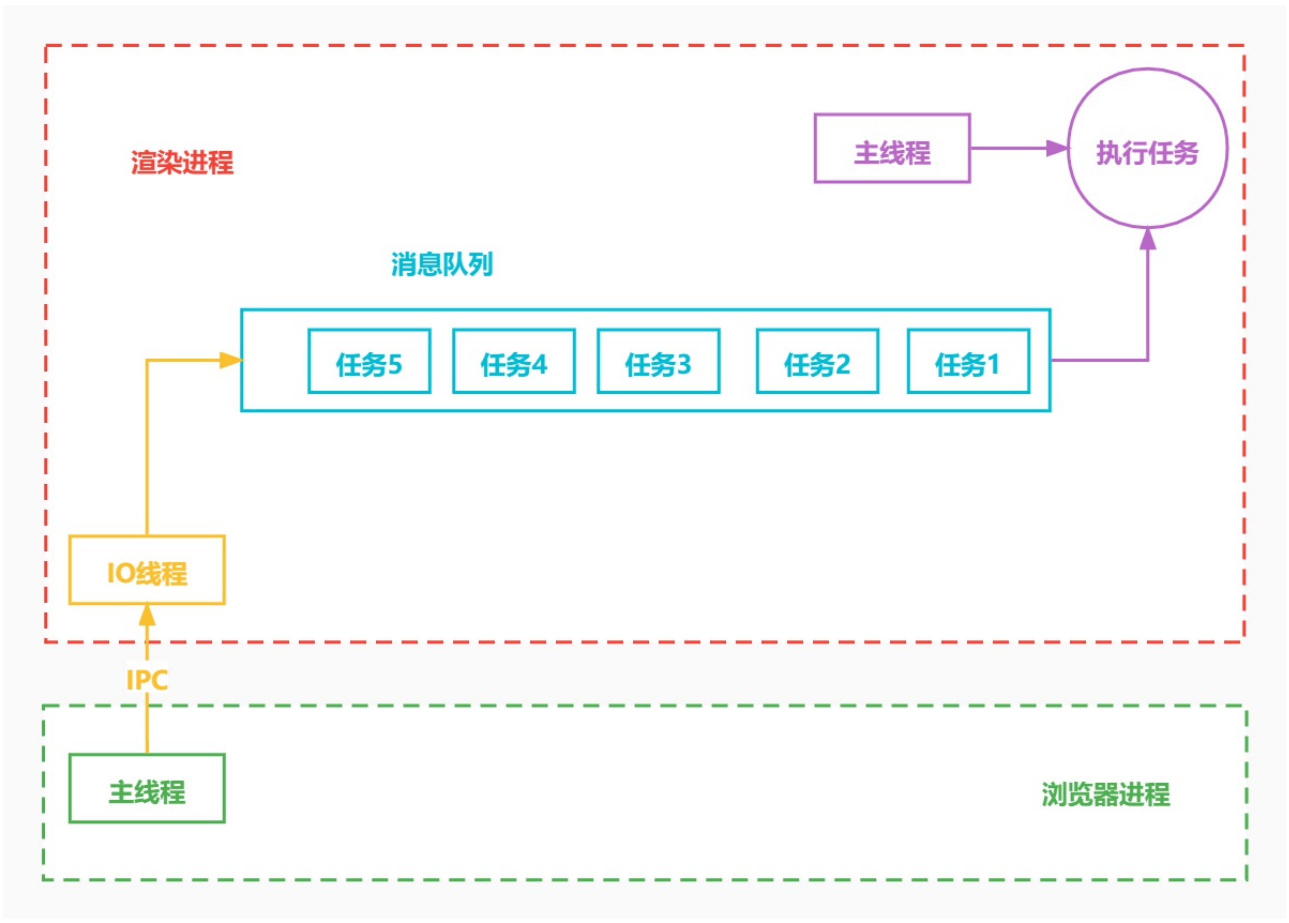
4.3.1 messageQueue.js

```
class MessageQueue {
    constructor() {
        this.messages = [];
    }
    put(message) {
        this.messages.push(message);
    }
    get() {
        return this.messages.pop();
    }
}
module.exports = new MessageQueue();
```

4.3.2 main.js

```
const messageQueue = require('./messageQueue');
(function mainThread() {
    setInterval(() => {
        let task = messageQueue.get();
        if (task) task();
    }, 1000);
})();

(function IOThread() {
    let counter = 1;
    setInterval(() => {
        messageQueue.put(()=>console.log(`task` + counter++));
    }, 1000);
})();
```

4.4.跨进程通信



4.4.1 render.js

render.js

```
const { fork } = require('child_process');
const messageQueue = require('./messageQueue');
(function mainThread() {
    setInterval(() => {
        let task = messageQueue.get();
        if (task) task();
    }, 1000);
})();

(function IOThread() {
    let browser = fork('./browser.js');
    browser.on('message', function ({ data }) {
        messageQueue.put(() => {
            console.log(data);
        });
    });
    browser.send({ type: 'click', data: 'clicked' });
})();
```

4.4.2 browser.js

- 浏览器主进程 browser.js

```
process.on('message', function ({ data }) {
  setTimeout(() => {
      process.send({ data });
  }, 100);
});
```

4.5. 微任务

- 微任务可以让新任务尽快的执行,会在主函数结束后，宏任务执行前执行

4.5.1 render.js

render.js

```
const { fork } = require('child_process');
+const { macroTaskQueue, microTaskQueue } = require('./messageQueue');
(function mainThread() {
    setInterval(() => {
+       let macroTask = macroTaskQueue.get();
+       if (macroTask) macroTask();
+       let microTask;
+       while (microTask = microTaskQueue.get()) {
+           microTask();
+       }
    }, 1000);
})();

(function IOThread() {
    let browser = fork('./browser.js');
    browser.on('message', function ({ data }) {
        macroTaskQueue.put(() => {
            console.log(data);
+           microTaskQueue.put(() => {
+               console.log('microTask1');
+               microTaskQueue.put(() => {
+                   console.log('microTask2');
+               });
+           });
        });
    });
    browser.send({ type: 'click', data: 'clicked' });
})();
```

4.5.2 browser.js

browser.js

```
process.on('message', function ({ data }) {
    setTimeout(() => {
        process.send({ data });
    }, 100);
});
```

4.5.3 messageQueue.js

messageQueue.js

```
class MessageQueue {
    constructor() {
        this.messages = [];
    }
    put(message) {
        this.messages.push(message);
    }
    get() {
        return this.messages.pop();
    }
}
+exports.macroTaskQueue = new MessageQueue();
+exports.microTaskQueue = new MessageQueue();
```

4.6. setTimeout

- 延迟队列
- 定时器是不精确的
- 嵌套5次以上定时器最短时间间隔设置为4ms
- 未激活页面最小间隔是1000ms
- 延迟最大值2147483647ms
- setTimeout中的this 会被设置为全局 window，如果是严格模式，会被设置为 undefined

4.6.1 DelayTask.js

DelayTask.js

```
let timerCounter = 1;
class DelayTask {
    constructor(callback, delayTime) {
        this.id = timerCounter++;
        this.startTime = Date.now();
        this.callback = callback;
        this.delayTime = delayTime;
    }

}
module.exports = DelayTask;
```

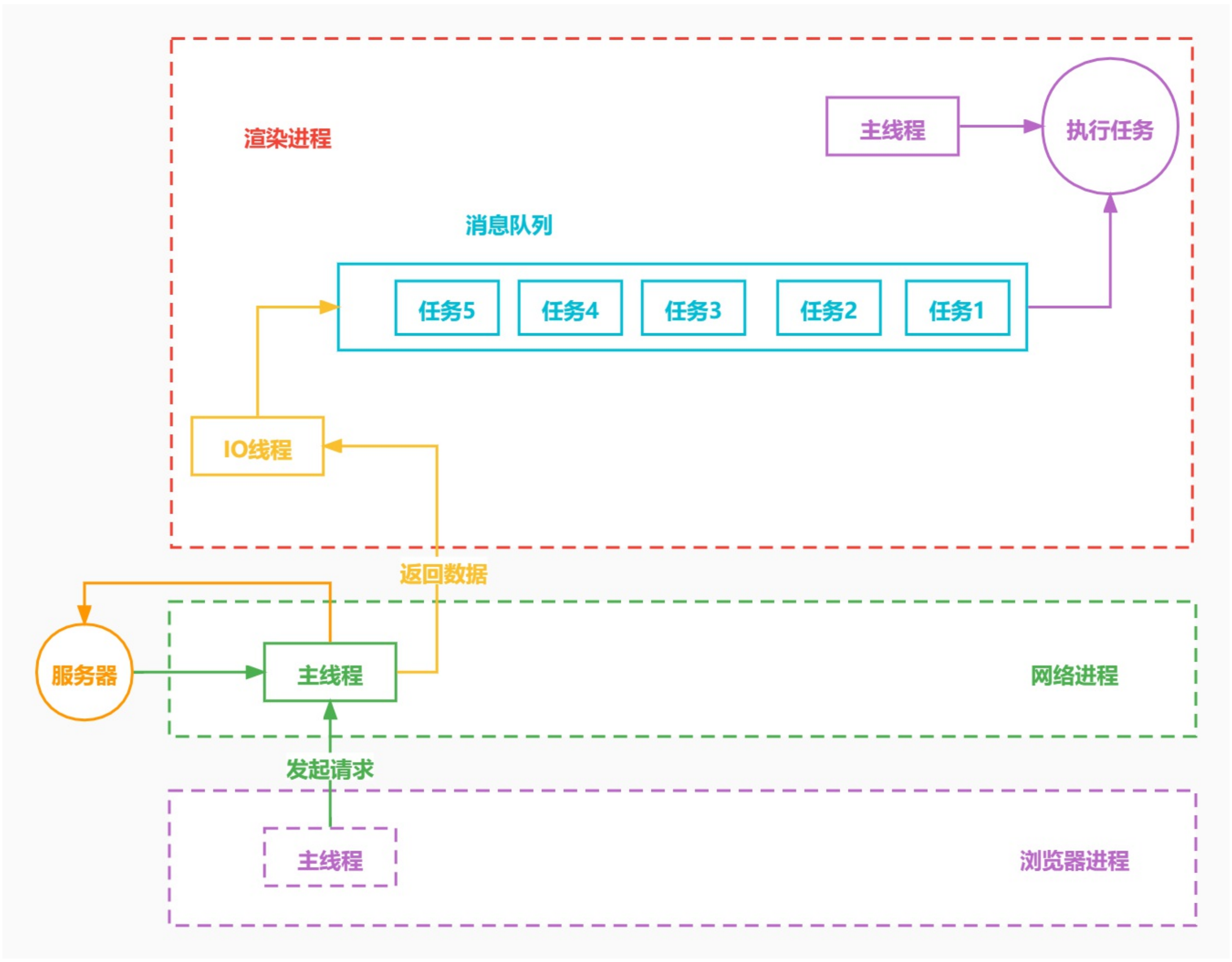
4.6.2 render.js

render.js

```
const { fork } = require('child_process');
const { macroTaskQueue, microTaskQueue } = require('./messageQueue');
+const DelayTask = require('./DelayTask');
+let delayTaskQueue = [];
(function mainThread() {
    setInterval(() => {
        let macroTask = macroTaskQueue.get();
        if (macroTask) macroTask();
+       processDelayTask();
        let microTask;
        while (microTask = microTaskQueue.get()) {
            microTask();
        }
    }, 16);
})();
+function setTimeout(callback, delayTime) {
+    delayTaskQueue.push(new DelayTask(callback, delayTime));
+}
+function clearTimeout(timeId) {
+    delayTaskQueue = delayTaskQueue.filter(delayTask => {
+        return delayTask.id !== timeId;
+    });
+}
+function processDelayTask() {
+    delayTaskQueue = delayTaskQueue.filter(delayTask => {
+        const { callback, startTime, delayTime } = delayTask;
+        if (Date.now() > startTime + delayTime) {
+            macroTaskQueue.put(callback);
+            return false;
+        }
+        return true;
+    });
+}

(function IOThread() {
    let browser = fork('./browser.js');
+   console.time('cost');
    browser.on('message', function ({ data }) {
        console.log(data);
+        setTimeout(() => {
+            console.timeEnd('cost');
+        }, 1000);
    });
    browser.send({ type: 'click', data: 'clicked' });
})();
```

4.7.XMLHttpRequest

- XMLHttpRequest是由浏览器进程或发起请求，然后再将执行结果利用 IPC 的方式通知渲染进程，之后渲染进程再将对应的消息添加到消息队列中



4.7.1 render.js

render.js

```
const { fork } = require('child_process');
const { macroTaskQueue, microTaskQueue } = require('./messageQueue');
(function mainThread() {
    setInterval(() => {
        let macroTask = macroTaskQueue.get();
        if (macroTask) macroTask();
        let microTask;
        while (microTask = microTaskQueue.get()) {
            microTask();
        }
    }, 16);
})();

(function IOThread() {
    let browser = fork('./browser.js');
    console.time('cost');
    browser.on('message', function () {
+       let xhr = new XMLHttpRequest();
+       xhr.open('GET', 'http://localhost:3000/data');
+       xhr.onload = function () {
+           console.log(xhr.response);
+       }
+       xhr.send();
    });
    browser.send({ type: 'click', data: 'clicked' });
})();

+class XMLHttpRequest {
+    constructor() {
+        this.options = {};
+    }
+    open(method, url) {
+        this.options.method = method;
+        this.options.url = url;
+    }
+    send() {
+        let child = fork('./XMLHttpRequest.js');
+        child.on('message', (message) => {
+            if (message.type === 'response') {
+                this.response = message.data;
+                macroTaskQueue.put(this.onload);
+            }
+        });
+        child.send({ type: 'send', options: this.options });
+    }
+}
```

4.7.2 XMLHttpRequest.js

XMLHttpRequest.js

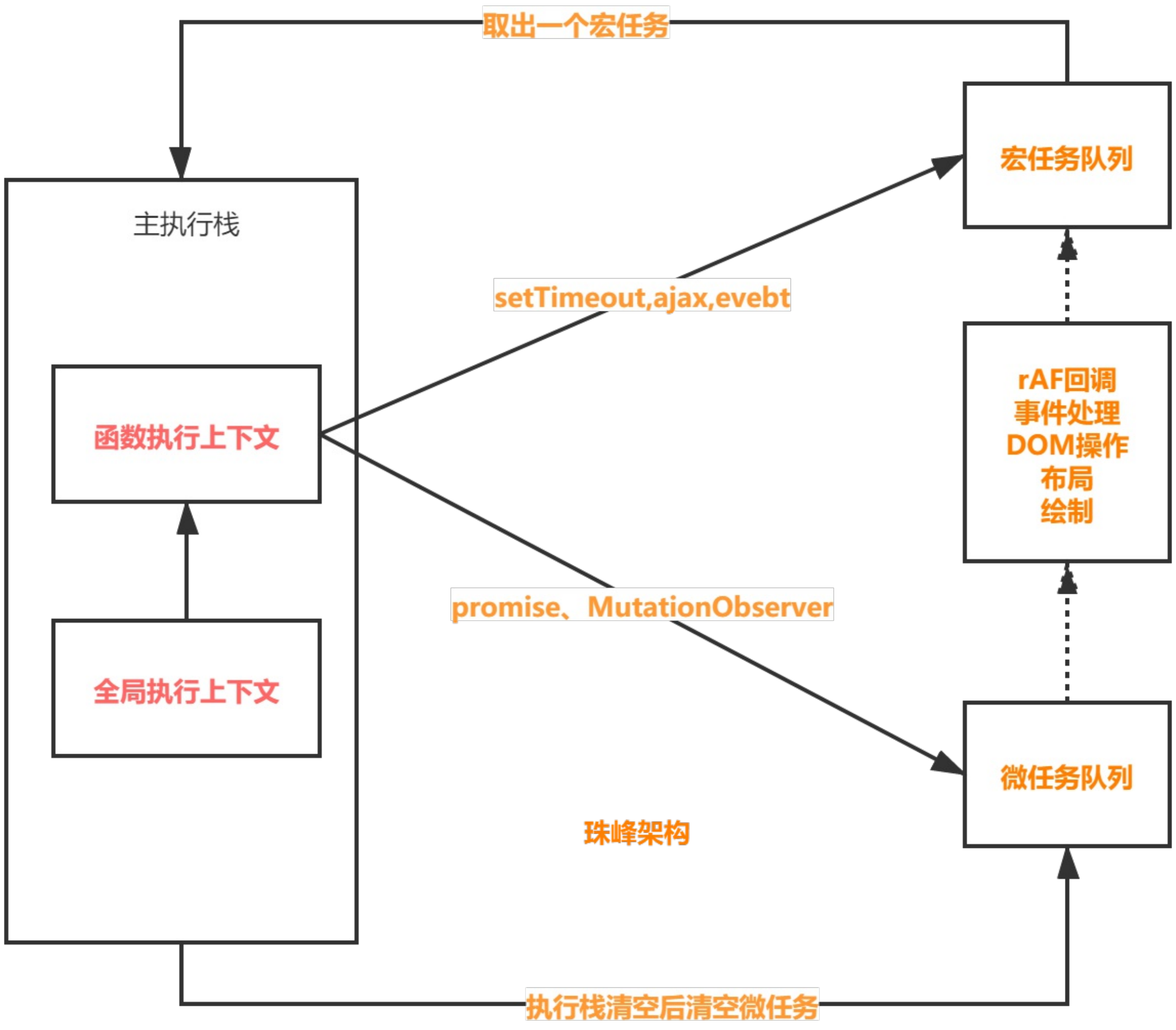
```
let url = require('url');
let http = require('http');
process.on('message', function (message) {
    let { type, options } = message;
    if (type == 'send') {
        let urlObj = url.parse(options.url);
        const config = {
            hostname: urlObj.hostname,
            port: urlObj.port,
            path: urlObj.path,
            method: options.method
        };
        var req = http.request(config, (res) => {
            let chunks = [];
            res.on('data', (chunk) => {
                chunks.push(chunk);
            });
            res.on('end', () => {
                process.send({
                    type: 'response',
                    data: JSON.parse(Buffer.concat(chunks).toString())
                });
                process.exit();
            });
        });
        req.on('error', (err) => {
            console.error(err);
        });
        req.end();
    }
});
```

4.7.3 server.js

server.js

```
var http = require("http");
var server = http.createServer();
server.on("request", function (request, response) {
    response.end(JSON.stringify({ message: 'hello' }));
})
server.listen(3000, function () {
    console.log("服务已经在3000端口启动!")
});
```

5.任务分类



5.1 宏任务

- 页面的大部分任务是在主任务上执行的，比如下面这些都是宏任务
    - 渲染事件(DOM解析、布局、绘制)
    - 用户交互(鼠标点击、页面缩放)
    - JavaScript脚本执行
    - 网络请求
    - 文件读写
- 宏任务会添加到消息到消息队列的尾部，当主线程执行到该消息的时候就会执行
- 每次从事件队列中获取一个事件回调并且放到执行栈中的就是一个宏任务，宏任务执行过程中不会执行其它内容
- 每次宏任务执行完毕后会进行GUI渲染线程的渲染，然后再执行下一个宏任务
- 宏任务: script（整体代码）, setTimeout, setInterval, setImmediate, I/O, UI rendering
- 宏任务颗粒度较大，不适合需要精确控制境的任务
- 宏任务是由宿主方控制的

5.2 微任务

- 宏任务结束后会进行渲染然后执行下一个宏任务
- 微任务是当前宏任务执行后立即执行的宏任务
- 当宏任务执行完，就到达了检查点,会先将执行期间所产生的所有微任务都执行完再去进行渲染
- 微任务是由V8引擎控制的，在创建全局执行上下文的时候，也会在V8引擎内部创建一个微任务队列
- 微任务: process.nextTick（Nodejs）, Promises, Object.observe, MutationObserver

5.3 MutationObserver

- MutationObserver (https://developer.mozilla.org/zh-CN/docs/Web/API/MutationObserver)创建并返回一个新的 MutationObserver 它会在指定的DOM发生变化时被调用
- MutationObserver采用了异步 + 微任务的方案
- 异步是为了提升同步操作带来的性能问题
- 微任务是为了解决实时响应的问题

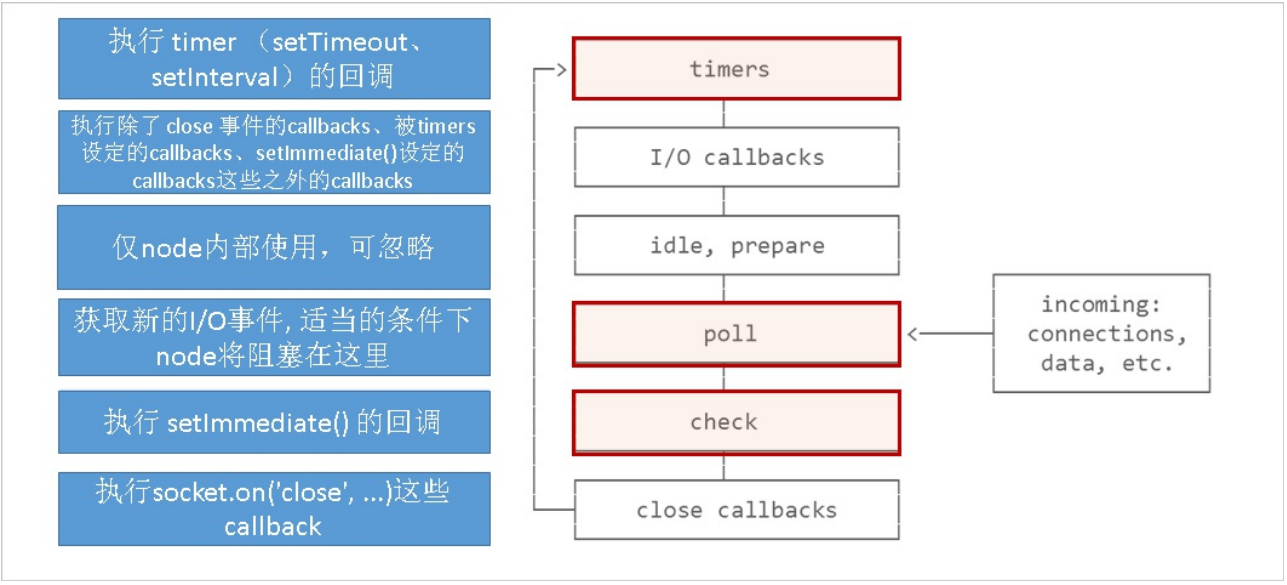
```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Documenttitle>
head>

<body>
    <div id="tree">div>
    <button onclick="start()">开始监听button>
    <button onclick="changeAttribute()">修改属性button>
    <button onclick="addChild()">添加子节点，3秒后删除button>
    <script>
        var targetNode = document.getElementById('tree');
        var config = { attributes: true, childList: true, subtree: true };
        var callback = function (mutationsList) {
            for (var mutation of mutationsList) {
                console.log(mutation);
                if (mutation.type == 'childList') {
                    console.log('添加或删除子节点');
                    console.log(mutation.addedNodes);
                    console.log(mutation.removedNodes);
                } else if (mutation.type == 'attributes') {
                    console.log('属性 ' + mutation.attributeName + ' 被改变了');
                }
            }
        };

        var observer = new MutationObserver(callback);
        function start() {
            observer.observe(targetNode, config);
        }
        function changeAttribute() {
            targetNode.setAttribute('data-name', '树');
        }
        function addChild() {
            let child = document.createElement('div');
            child.innerHTML = '子节点';
            targetNode.appendChild(child);
            setTimeout(() => {
                targetNode.removeChild(child);
            }, 2000);
        }
    script>
body>

html>
```

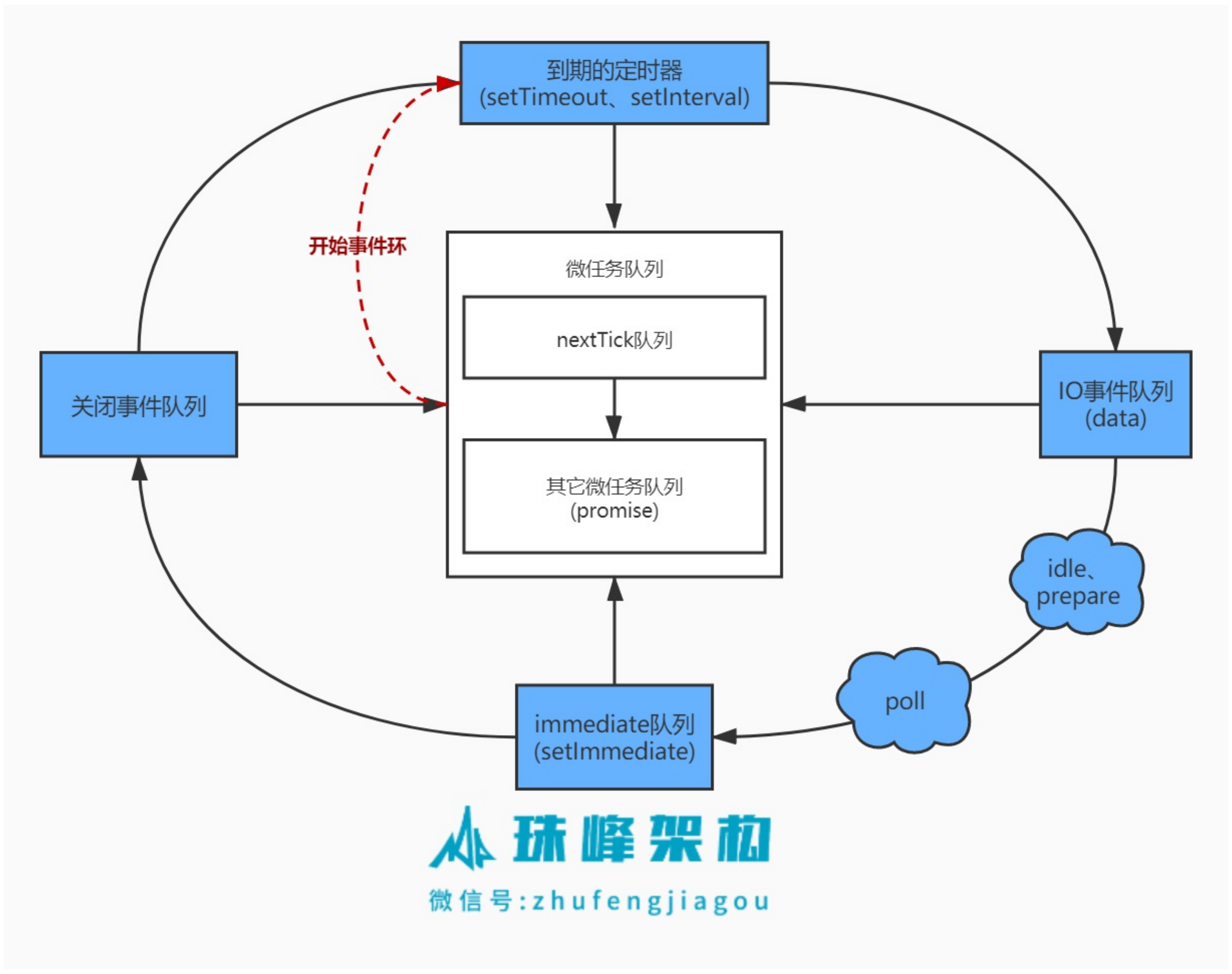
6. Node中的EventLoop

- Node.js采用V8作为js的解析引擎，而I/O处理方面使用了自己设计的libuv
- libuv是一个基于事件驱动的跨平台抽象层，封装了不同操作系统一些底层特性，对外提供统一的API
- 事件循环机制也是它里面的实现
    - V8引擎解析JavaScript脚本并调用Node API
    - libuv库负责Node API的执行。它将不同的任务分配给不同的线程,形成一个Event Loop（事件循环），以异步的方式将任务的执行结果返回给V8引擎
    - V8引擎再将结果返回给用户

6.1 libuv

- 同步执行全局的脚本
- 执行所有的微任务，先执行nextTick中的所有的任务，再执行其它微任务
- 开始执行宏任务，共有6个阶段，从第1个阶段开始，会执行每一个阶段所有的宏任务


执行 timer （setTimeout、setInterval）的回调
执行除了 close 事件的callbacks、被timers设定的callbacks、setImmediate()设定的callbacks这些之外的callbacks
仅node内部使用，可忽略
获取新的I/O事件，适当的条件下node将阻塞在这里
执行 setImmediate() 的回调
执行socket.on('close', ...)这些callback
timers
I/O callbacks
idle, prepare
poll
check
close callbacks
incoming:
connections,
data, etc.



- setTimeout/setInterval取值范围是[1,2的32次方-1],超出范围初始化为1，所以 setTimeout(fn,0) = setTimeout(fn,1)

```
setTimeout(function  () {
  console.log('timeout');
},0);
setImmediate(function  () {
  console.log('immediate');
});
```

```
const fs = require('fs')
fs.readFile(__filename, () => {
    setTimeout(() => {
        console.log('timeout');
    }, 0)
    setImmediate(() => {
        console.log('immediate')
    })
})
```

6.3 process.nextTick

- nextTick独立于Event Loop,有自己的队列，每个阶段完成后如果存在nextTick队列会全部清空，优先级高于微任务

```
setTimeout(() => {
    console.log('setTimeout1')
    Promise.resolve().then(function () {
        console.log('promise1')
    })
}, 0)
setTimeout(() => {
    console.log('setTimeout2')
    Promise.resolve().then(function () {
        console.log('promise2')
    })
}, 0)
setImmediate(() => {
    console.log('setImmediate1')
    Promise.resolve().then(function () {
        console.log('promise3')
    })
}, 0)

process.nextTick(() => {
    console.log('nextTick1');
    Promise.resolve().then(() => console.log('promise4'));
    process.nextTick(() => {
        console.log('nextTick2');
        Promise.resolve().then(() => console.log('promise5'));
        process.nextTick(() => {
            console.log('nextTick3')
            process.nextTick(() => {
                console.log('nextTick4')
            })
        })
    })
})
```

6.4 Node.js事件环

```
let fs = require('fs');
setTimeout(() => {
    console.log('1');
    let rs1 = fs.createReadStream(__filename);
    rs1.on('data', () => {
        rs1.destroy();
        setImmediate(() => console.log('setImmediate_a'));
        setTimeout(() => {
            console.log('setTimeout_a')
        });
        console.log('a');
    });
    rs1.on('close', () => console.log('end_a'));
    console.log('2');
    setImmediate(function () {
        console.log('setImmediate1');
        process.nextTick(() => console.log('nextTick1'));
    });
    setImmediate(function () {
        console.log('setImmediate2');
        process.nextTick(() => console.log('nextTick2'));
    });
    console.log('3');
    setTimeout(() => {
        console.log('setTimeout1');
        process.nextTick(() => {
            console.log('nextTick3')
            process.nextTick(() => console.log('nextTick4'));
        });
    });
    setTimeout(() => {
        console.log('setTimeout2');
    });
    console.log('4');
}, 1000);
```

7. 面试题

- Promise (https://static.zhufengpeixun.com/Promise_1642590129964.js)
- #sec-promise-resolve-functions (https://tc39.es/ecma262/#sec-promise-resolve-functions)
- promise-resolve.tq#151 (https://chromium.googlesource.com/v8/v8.git/+/refs/heads/9.0-lkgr/src/builtins/promise-resolve.tq#151)
- promise-jobs.tq#13 (https://chromium.googlesource.com/v8/v8.git/+/refs/heads/9.0-lkgr/src/builtins/promise-jobs.tq#13)
- promise-then.tq (https://chromium.googlesource.com/v8/v8.git/+/refs/heads/9.0-lkgr/src/builtins/promise-then.tq)
- promise-abstract-operations.tq (https://chromium.googlesource.com/v8/v8.git/+/refs/heads/9.0-lkgr/src/builtins/promise-abstract-operations.tq)
- promise-abstract-operations.tq#409 (https://chromium.googlesource.com/v8/v8.git/+/refs/heads/9.0-lkgr/src/builtins/promise-abstract-operations.tq#409)

7.1 原题

```
const Promise = require('./Promise');
Promise.resolve().then(() => {
    console.log(0);
    return new Promise((resolve)=>{
        resolve('a');
    })
}).then(res => {
    console.log(res)
})
Promise.resolve().then(() => {
    console.log(1);
}).then(() => {
    console.log(2);
}).then(() => {
    console.log(3);
}).then(() => {
    console.log(4);
}).then(() => {
    console.log(5);
})
```

7.2 改造

```
const Promise = require('./Promise');
let promise1 = Promise.resolve();
let promise2 = promise1.then(() => {
    console.log(0);
    let promise10 = Promise.resolve('a');
    return promise10;
})
let promise3 = promise2.then(res => {
    console.log(res)
})
let promise4 = Promise.resolve();
let promise5 = promise4.then(() => {
    console.log(1);
});
let promise6 = promise5.then(() => {
    console.log(2);
});
let promise7 = promise6.then(() => {
    console.log(3);
});
let promise8 = promise7.then(() => {
    console.log(4);
});
let promise9 = promise8.then(() => {
    console.log(5);
})
```

7.3 Promise.js

```
const PENDING = 'PENDING';
const FULFILLED = 'FULFILLED';
function resolvePromise(promise, x, resolve) {
    if (x && typeof x.then === 'function') {
        queueMicrotask(() => {
            console.log('resolvePromise: microtask', x.id);
            x.then(y => resolvePromise(promise, y, resolve));
        });
    } else {
        resolve(x)
    }
}
class Promise {
    static counter = 1;
    constructor(executor) {
        this.id = Promise.counter++;
        this.status = PENDING;
        this.value = undefined;
        this.reason = undefined;
        this.onResolvedCallbacks = [];
        const resolve = (value) => {
            if (value instanceof Promise) {
                return queueMicrotask(() => {
                    console.log('resolve: microtask', value.id);
                    value.then(resolve)
                });
            }
            if (this.status == PENDING) {
                this.value = value;
                this.status = FULFILLED
                this.onResolvedCallbacks.forEach(cb => cb(this.value))
            }
        }
        executor(resolve);
    }
    then(onFulfilled) {
        let newPromise = new Promise((resolve) => {
            if (this.status === FULFILLED) {
                queueMicrotask(() => {
                    console.log('FULFILLED then: microtask', this.id);
                    let x = onFulfilled(this.value);
                    resolvePromise(newPromise, x, resolve)
                })
            }
            if (this.status == PENDING) {
                this.onResolvedCallbacks.push(() => {
                    queueMicrotask(() => {
                        console.log('PENDING then: microtask', this.id);
                        let x = onFulfilled(this.value);
                        resolvePromise(newPromise, x, resolve)
                    })
                });
            }
        });
        return newPromise;
    }
    static resolve(value) {
        return new Promise((resolve) => {
            resolve(value)
        })
    }
}
module.exports = Promise;
```

7.4 讲解

- 第1个Promise.resolve()返回一个promise1
- 这个promise1会立刻调用它的resolve方法，把它的value值设置为undefined,状态设置为完成态,执行成功回调是空数组
- 接着调用promise1的then方法，因为此时promise1已经是完成态了，入队

```
console.log('FULFILLED then: microtask', this.id);
let x = onFulfilled(this.value);
resolvePromise(p1, x, resolve, reject)
```

- 然后返回promise2
- 然后继续调用promise2的then方法，发现promise2还处于等待态，不能直接添加微任务，只能添加

```
() => {
  queueMicrotask(() => {
      console.log('PENDING then: microtask', this.id);
      let x = onFulfilled(this.value);
      resolvePromise(p1, x, resolve)
  })
}
```

到promise2的onResolvedCallbacks的尾部，并返回promise3，这个promise3后面没有再用到了，此时第一段代码结束

- 开始执行第二段代码
- 第1个Promise.resolve()返回一个promise4
- promise4会立刻调用它的resolve方法，把它的value值设置为undefined,状态设置为完成态,执行成功回调是空数组
- 接着调用promise4的then方法，因为此时promise4已经是完成态了，入队

```
queueMicrotask(() => {
  console.log('FULFILLED then: microtask',4);
  let x = onFulfilled(this.value);
  resolvePromise(newPromise, x, resolve)
})
```

- 然后返回promise5
- 然后继续调用promise5的then方法，发现promise5还处于等待态，不能直接添加微任务，只能添加

```
this.onResolvedCallbacks.push(() => {
 queueMicrotask(() => {
     console.log('PENDING then: microtask', this.id);
     let x = onFulfilled(this.value);
     resolvePromise(newPromise, x, resolve)
 })
});
```

- 然后返回promise6
- 然后继续调用promise6的then方法，发现promise6还处于等待态，不能直接添加微任务，只能添加

```
this.onResolvedCallbacks.push(() => {
 queueMicrotask(() => {
     console.log('PENDING then: microtask', this.id);
     let x = onFulfilled(this.value);
     resolvePromise(newPromise, x, resolve)
 })
});
```

- 然后返回promise7
- 然后继续调用promise7的then方法，发现promise7还处于等待态，不能直接添加微任务，只能添加

```
this.onResolvedCallbacks.push(() => {
 queueMicrotask(() => {
     console.log('PENDING then: microtask', this.id);
     let x = onFulfilled(this.value);
     resolvePromise(newPromise, x, resolve)
 })
});
```

- 然后返回promise8
- 然后继续调用promise8的then方法，发现promise8还处于等待态，不能直接添加微任务，只能添加

```
this.onResolvedCallbacks.push(() => {
 queueMicrotask(() => {
     console.log('PENDING then: microtask', this.id);
     let x = onFulfilled(this.value);
     resolvePromise(newPromise, x, resolve)
 })
});
```

- 然后返回promise9
- 然后继续调用promise9的then方法，发现promise9还处于等待态，不能直接添加微任务，只能添加

```
this.onResolvedCallbacks.push(() => {
 queueMicrotask(() => {
     console.log('PENDING then: microtask', this.id);
     let x = onFulfilled(this.value);
     resolvePromise(newPromise, x, resolve)
 })
});
```

- 此时，任务列队上有两个新任务 微任务队列 [`FULFILLED then: microtask 1`， `FULFILLED then: microtask 4`]
- 把第一个微任务出队 `FULFILLED then: microtask 1` 执行
- 输出 0
- 创建promise10并返回
- 在执行 `resolvePromise(promise2, promise10, resolve)`的时候，如果发现promise10是一个promise,并且入队

```
console.log('resolvePromise: microtask 10');
x.then(y => resolvePromise(promise, y, resolve));
```

- 此时第一个微任务执行完毕
- 此时任务队列有两个任务[`FULFILLED then: microtask 4`,`resolvePromise: microtask 10`]
- 然后执行 `FULFILLED then: microtask 4`,输出1
- 然后会将promise4变成完成态，执行成功回调，成功回调会把 `PENDING then: microtask 5`入队,此时本任务结束
- 此时任务队列[`resolvePromise: microtask 10`,`PENDING then: microtask 5`]
- 然后再执行 `resolvePromise: microtask 10`,因为promise10是直接成功的，直接执行成功回调，入队

```
console.log('FULFILLED then: microtask 10');
let x = onFulfilled(this.value);
resolvePromise(newPromise, x, resolve)
```

- 此时队列 [`PENDING then: microtask 5`,`FULFILLED then: microtask 10`]
- 执行 `PENDING then: microtask 5`,输出2
- 此时promise5变成完成态，执行成功回调，成功回调会把 `PENDING then: microtask 6`入队,此时本任务结束
- 此时队列 [`FULFILLED then: microtask 10`， `PENDING then: microtask 6`]
- 执行 `FULFILLED then: microtask 10`,它会让promise2变成成功态，并且把promise2的成功回调入队
- 此时队列 [`PENDING then: microtask 6`,`FULFILLED then: microtask 2`]
- 再执行 `PENDING then: microtask 6`,输出3，
- 此时promise6变成完成态，执行成功回调，成功回调会把 `PENDING then: microtask 7`入队,此时本任务结束
- 此时队列 [`FULFILLED then: microtask 2`,`PENDING then: microtask 7`]
- 执行 `FULFILLED then: microtask 2`，输出a
- 再执行 `PENDING then: microtask 7`,输出4
- 此时promise7变成完成态，执行成功回调，成功回调会把 `PENDING then: microtask 8`入队,此时本任务结束
- 此时队列 [`PENDING then: microtask 8`]
- 再执行 `PENDING then: microtask 8`,输出5

7.5 打印

```
FULFILLED then: microtask 1
0
FULFILLED then: microtask 4
1
resolvePromise: microtask 10
PENDING then: microtask 5
2
FULFILLED then: microtask 10
PENDING then: microtask 6
3
PENDING then: microtask 2
a
PENDING then: microtask 7
4
PENDING then: microtask 8
5
```

7.思考

```
const Promise = require('./Promise');
Promise.resolve().then(() => {
    console.log(0);
    return new Promise((resolve) => {
        resolve(new Promise((resolve) => {
            resolve('a');
        }));
    })
}).then(res => {
    console.log(res)
})
Promise.resolve().then(() => {
    console.log(1);
}).then(() => {
    console.log(2);
}).then(() => {
    console.log(3);
}).then(() => {
    console.log(4);
}).then(() => {
    console.log(5);
})
```