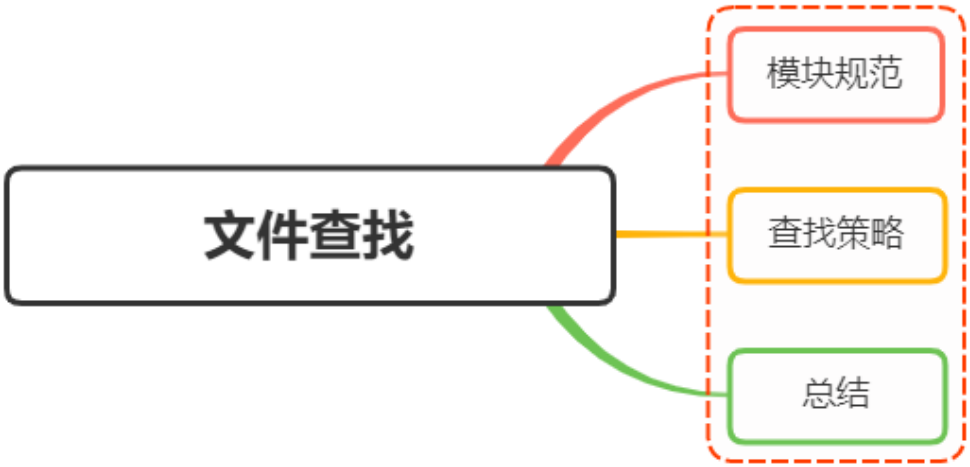


7.1. 模块规范

`NodeJS` 对 `CommonJS` 进行了支持和实现，让我们在开发 `node` 的过程中可以方便的进行模块化开发：

- 在Node中每一个js文件都是一个单独的模块
- 模块中包括CommonJS规范的核心变量：exports、module.exports、require
- 通过上述变量进行模块化开发

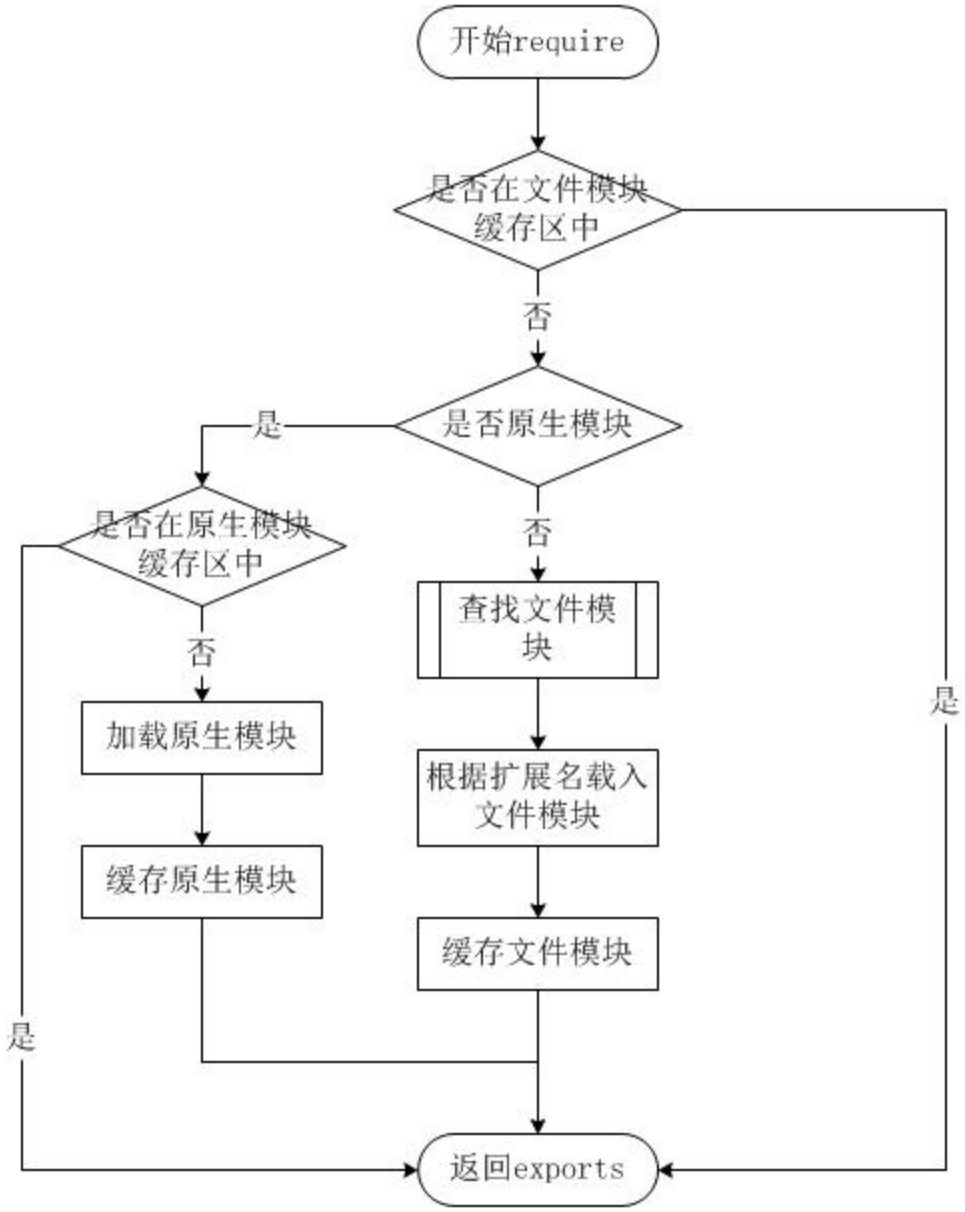
而模块化的核心是导出与导入，在 `Node` 中通过 `exports` 与 `module.exports` 负责对模块中的内容进行导出，通过 `require` 函数导入其他模块（自定义模块、系统模块、第三方库模块）中的内容

7.2. 查找策略

`require` 方法接收一下几种参数的传递：

- 原生模块：http、fs、path等
- 相对路径的文件模块：./mod或../mod
- 绝对路径的文件模块：/pathtomodule/mod
- 目录作为模块：./dirname
- 非原生模块的文件模块：mod

`require` 参数较为简单，但是内部的加载却是十分复杂的，其加载优先级也各自不同，如下图：



从上图可以看见，文件模块存在缓存区，寻找模块路径的时候都会优先从缓存中加载已经存在的模块

7.2.1. 原生模块

而像原生模块这些，通过 `require` 方法在解析文件名之后，优先检查模块是否在原生模块列表中，如果在则从原生模块中加载

7.2.2. 绝对路径、相对路径

如果 `require` 绝对路径的文件，则直接查找对应的路径，速度最快

相对路径的模块则相对于当前调用 `require` 的文件去查找

如果按确切的文件名没有找到模块，则 `NodeJs` 会尝试带上 `.js` 、 `.json` 或 `.node` 拓展名再加载

7.2.3. 目录作为模块

默认情况是根据根目录中 `package.json` 文件的 `main` 来指定目录模块，如：

```JSON
{ "name" : "some-library",
  "main" : "main.js" }
```

如果这是在 `./some-library node_modules` 目录中，则 `require('./some-library')` 会试图加载 `./some-library/main.js`

如果目录里没有 `package.json` 文件，或者 `main` 入口不存在或无法解析，则会试图加载目录下的 `index.js` 或 `index.node` 文件

7.2.4. 非原生模块

在每个文件中都存在 `module.paths` ，表示模块的搜索路径， `require` 就是根据其来寻找文件

在 `window` 下输出如下：

```JavaScript
[ 'c:\\nodejs\\node_modules',
'c:\\node_modules' ]
```

可以看出 `module path` 的生成规则为：从当前文件目录开始查找 `node_modules` 目录；然后依次进入父目录，查找父目录下的 `node_modules` 目录，依次迭代，直到根目录下的 `node_modules` 目录

当都找不到的时候，则会从系统 `NODE_PATH` 环境变量查找

7.2.4.1. 举个例子

如果在 `/home/ry/projects/foo.js` 文件里调用了 `require('bar.js')` ，则 Node.js 会按以下顺序查找：

- /home/ry/projects/node_modules/bar.js
- /home/ry/node_modules/bar.js
- /home/node_modules/bar.js

- /node_modules/bar.js

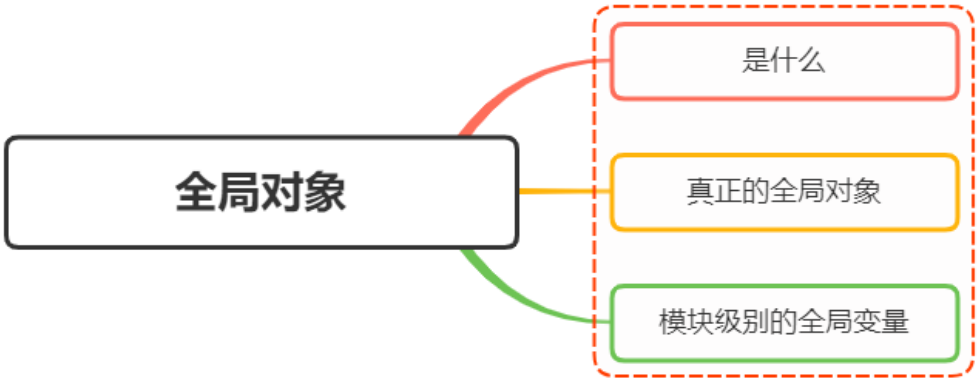
这使得程序本地化它们的依赖，避免它们产生冲突

7.3. 总结

通过上面模块的文件查找策略之后，总结下文件查找的优先级：

- 缓存的模块优先级最高
- 如果是内置模块，则直接返回，优先级仅次缓存的模块
- 如果是绝对路径 / 开头，则从根目录找
- 如果是相对路径 ./开头，则从当前require文件相对位置找
- 如果文件没有携带后缀，先从js、json、node按顺序查找
- 如果是目录，则根据 package.json的main属性值决定目录下入口文件，默认情况为 index.js
- 如果文件为第三方模块，则会引入 node_modules 文件，如果不在当前仓库文件中，则自动从上级递归查找，直到根目录

8. 说说 Node有哪些全局对象?



8.1. 是什么

在浏览器 `JavaScript` 中，通常 `window` 是全局对象， 而 `Nodejs` 中的全局对象是 `global`

在 `NodeJS` 里，是不可能在最外层定义一个变量，因为所有的用户代码都是当前模块的，只在当前模块里可用，但可以通过 `exports` 对象的使用将其传递给模块外部

所以，在 `NodeJS` 中，用 `var` 声明的变量并不属于全局的变量，只在当前模块生效

像上述的 `global` 全局对象则在全局作用域中，任何全局变量、函数、对象都是该对象的一个属性值

8.2. 有哪些

将全局对象分成两类：

- 真正的全局对象
- 模块级别的全局变量

8.2.1. 真正的全局对象

下面给出一些常见的全局对象：

- Class:Buffer
- process
- console
- clearInterval、setInterval
- clearTimeout、setTimeout
- global

8.2.1.1. Class:Buffer

可以处理二进制以及非 `Unicode` 编码的数据

在 `Buffer` 类实例化中存储了原始数据。 `Buffer` 类似于一个整数数组，在V8堆原始存储空间给它分配了内存

一旦创建了 `Buffer` 实例，则无法改变大小

8.2.1.2. process

进程对象，提供有关当前进程的信息和控制

包括在执行 `node` 程序进程时，如果需要传递参数，我们想要获取这个参数需要在 `process` 内置对象中

启动进程：

```
Plain Text | 复制代码
1    node index.js 参数1 参数2 参数3
```

index.js文件如下：

```javascript
process.argv.forEach((val, index) => {
  console.log(`${index}: ${val}`);
});
```

输出如下：

```
/usr/local/bin/node
/Users/mjr/work/node/process-args.js
参数1
参数2
参数3
```

除此之外，还包括一些其他信息如版本、操作系统等

```
process {
  version: 'v12.16.1',
  versions: {
    node: '12.16.1',
    v8: '7.8.279.23-node.31',
    uv: '1.34.0',
    zlib: '1.2.11',
    brotli: '1.0.7',
    ares: '1.15.0',
    modules: '72',
    nghttp2: '1.40.0',
    napi: '5',
    llhttp: '2.0.4',
    http_parser: '2.9.3',
    openssl: '1.1.1d',
    cldr: '35.1',
    icu: '64.2',
    tz: '2019c',
    unicode: '12.1'
  },
  arch: 'x64',
  platform: 'win32',
  release: {
    name: 'node',
    lts: 'Erbium',
    sourceUrl: 'https://nodejs.org/download/release/v12.16.1/node-v12.16.1.tar.gz',
    headersUrl: 'https://nodejs.org/download/release/v12.16.1/node-v12.16.1-headers
.tar.gz',
    libUrl: 'https://nodejs.org/download/release/v12.16.1/win-x64/node.lib'
  },
  _rawDebug: [Function: _rawDebug],
  moduleLoadList: [
    'Internal Binding native_module',
    'Internal Binding errors',
    'Internal Binding buffer',
```

8.2.1.3. console

用来打印 `stdout` 和 `stderr`

最常用的输入内容的方式：console.log

```
JavaScript | 复制代码
1  console.log("hello");
```

清空控制台：console.clear

```
JavaScript | 复制代码
1  console.clear
```

打印函数的调用栈：console.trace

```javascript
function test() {
    demo();
}

function demo() {
    foo();
}

function foo() {
    console.trace();
}

test();
```

```
Trace
    at foo (E:\Users\user\Desktop\111\index.js:10:13)
    at demo (E:\Users\user\Desktop\111\index.js:6:5)
    at test (E:\Users\user\Desktop\111\index.js:2:5)
    at Object.<anonymous> (E:\Users\user\Desktop\111\index.js:13:3)
    at Module._compile (internal/modules/cjs/loader.js:1158:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1178:10)
    at Module.load (internal/modules/cjs/loader.js:1002:32)
    at Function.Module._load (internal/modules/cjs/loader.js:901:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)
    at internal/main/run_main_module.js:18:47
```

8.2.1.4. clearInterval、setInterval

设置定时器与清除定时器

```javascript
setInterval(callback, delay[, ...args])
```

`callback` 每 `delay` 毫秒重复执行一次

`clearInterval` 则为对应发取消定时器的方法

8.2.1.5. clearTimeout、setTimeout

设置延时器与清除延时器

```
JavaScript | 复制代码
1  setTimeout(callback,delay[,...args])
```

`callback` 在 `delay` 毫秒后执行一次

`clearTimeout` 则为对应取消延时器的方法

8.2.1.6. global

全局命名空间对象，墙面讲到的 `process` 、 `console` 、 `setTimeout` 等都有放到 `global` 中

```
JavaScript | 复制代码
1  console.log(process === global.process) // true
```

8.2.2. 模块级别的全局对象

这些全局对象是模块中的变量，只是每个模块都有，看起来就像全局变量，像在命令交互中是不可以使用，包括：

- __dirname
- __filename
- exports
- module
- require

8.2.2.1. __dirname

获取当前文件所在的路径，不包括后面的文件名

从 `/Users/mjr` 运行 `node example.js` ：

```
JavaScript | 复制代码
1  console.log(__dirname);
2  // 打印: /Users/mjr
```

8.2.2.2. __filename

获取当前文件所在的路径和文件名称，包括后面的文件名称

从 `/Users/mjr` 运行 `node example.js`：

```JavaScript
console.log(__filename);
// 打印: /Users/mjr/example.js
```

8.2.2.3. exports

`module.exports` 用于指定一个模块所导出的内容，即可以通过 `require()` 访问的内容

```JavaScript
exports.name = name;
exports.age = age;
exports.sayHello = sayHello;
```

8.2.2.4. module

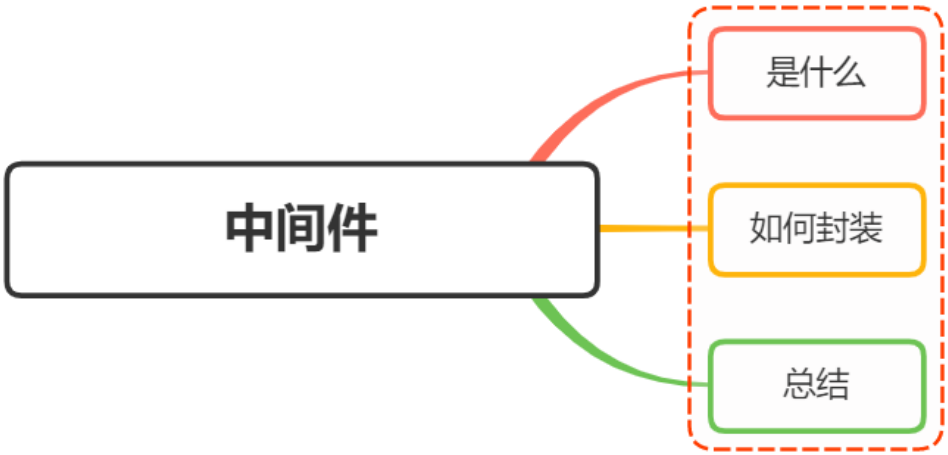
对当前模块的引用，通过 `module.exports` 用于指定一个模块所导出的内容，即可以通过 `require()` 访问的内容

8.2.2.5. require

用于引入模块、 `JSON` 、或本地文件。 可以从 `node_modules` 引入模块。

可以使用相对路径引入本地模块或 `JSON` 文件，路径会根据 `__dirname` 定义的目录名或当前工作目录进行处理

9. 说说对中间件概念的理解，如何封装 node 中间件?

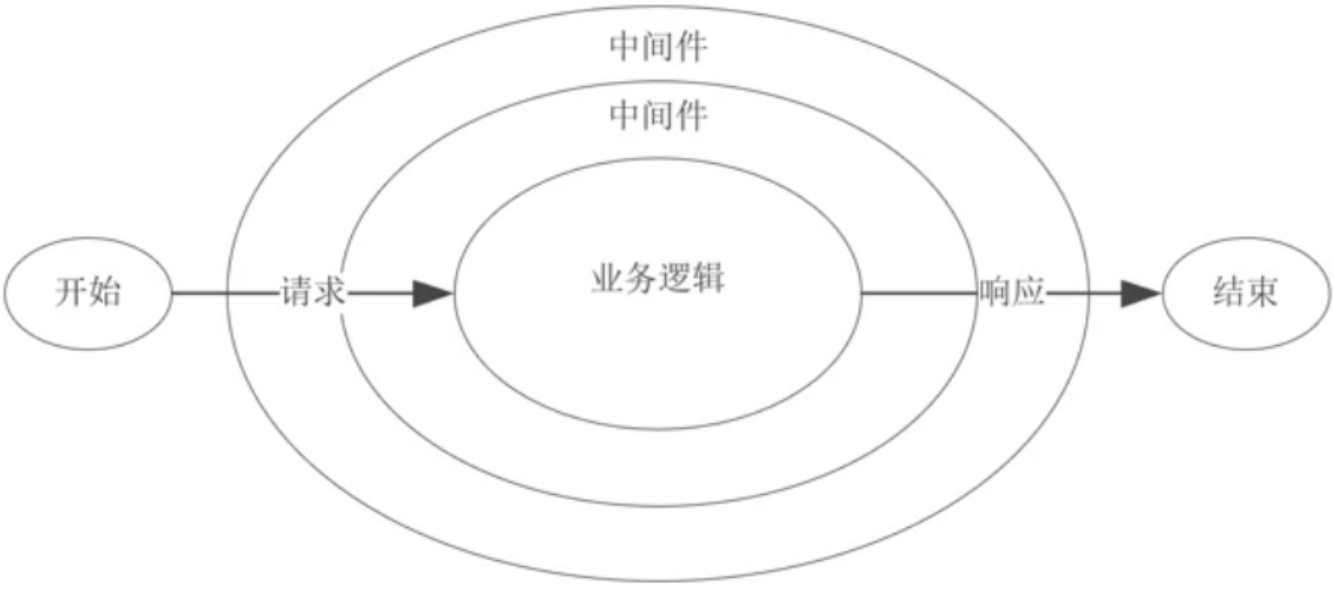


9.1. 是什么

中间件（Middleware）是介于应用系统和系统软件之间的一类软件，它使用系统软件所提供的基础服务（功能），衔接网络上应用系统的各个部分或不同的应用，能够达到资源共享、功能共享的目的

在 `NodeJS` 中，中间件主要是指封装 `http` 请求细节处理的方法

例如在 `express` 、 `koa` 等 `web` 框架中，中间件的本质为一个回调函数，参数包含请求对象、响应对象和执行下一个中间件的函数



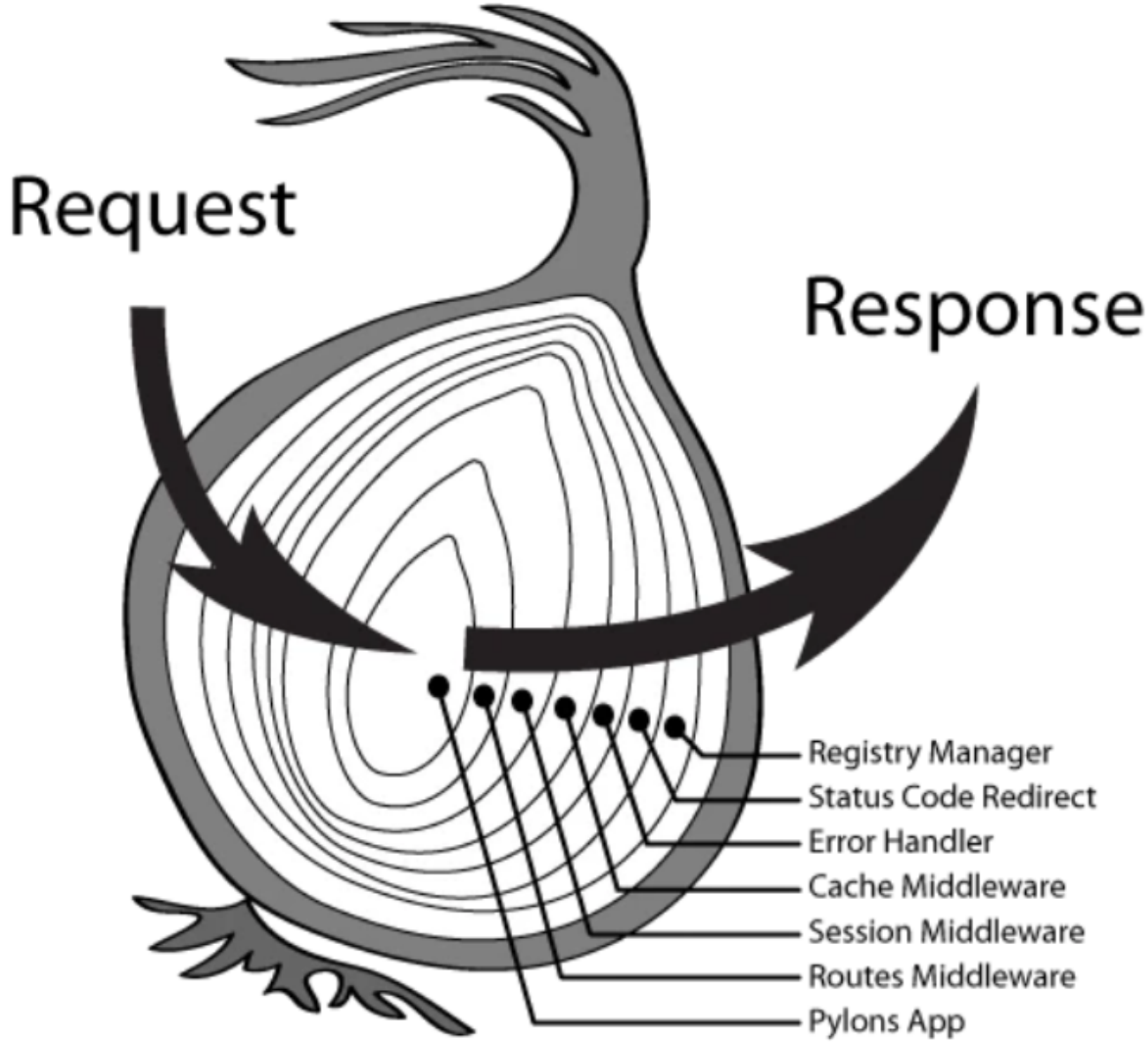
在这些中间件函数中，我们可以执行业务逻辑代码，修改请求和响应对象、返回响应数据等操作

9.2. 封装

`koa` 是基于 `NodeJS` 当前比较流行的 `web` 框架，本身支持的功能并不多，功能都可以通过中间件拓展实现。通过添加不同的中间件，实现不同的需求，从而构建一个 `Koa` 应用

`Koa` 中间件采用的是洋葱圈模型，每次执行下一个中间件传入两个参数：

- ctx ： 封装了request 和 response 的变量
- next ： 进入下一个要执行的中间件的函数



下面就针对 `koa` 进行中间件的封装：

`Koa` 的中间件就是函数，可以是 `async` 函数，或是普通函数

```JavaScript
// async 函数
app.use(async (ctx, next) => {
  const start = Date.now();
  await next();
  const ms = Date.now() - start;
  console.log(`${ctx.method} ${ctx.url} - ${ms}ms`);
});

// 普通函数
app.use((ctx, next) => {
  const start = Date.now();
  return next().then(() => {
    const ms = Date.now() - start;
    console.log(`${ctx.method} ${ctx.url} - ${ms}ms`);
  });
});
```

下面则通过中间件封装 `http` 请求过程中几个常用的功能：

9.2.1. token校验

```JavaScript
module.exports = (options) => async (ctx, next) {
  try {
    // 获取 token
    const token = ctx.header.authorization
    if (token) {
      try {
          // verify 函数验证 token，并获取用户相关信息
          await verify(token)
      } catch (err) {
        console.log(err)
      }
    }
    // 进入下一个中间件
    await next()
  } catch (err) {
    console.log(err)
  }
}
```

9.2.2. 日志模块

JavaScript | 复制代码

```
const fs = require('fs')
module.exports = (options) => async (ctx, next) => {
  const startTime = Date.now()
  const requestTime = new Date()
  await next()
  const ms = Date.now() - startTime;
  let logout = `${ctx.request.ip} -- ${requestTime} -- ${ctx.method} -- ${ctx.url} -- ${ms}ms`;
  // 输出日志文件
  fs.appendFileSync('./log.txt', logout + '\n')
}
```

`Koa` 存在很多第三方的中间件，如 `koa-bodyparser` 、 `koa-static` 等

下面再来看看它们的大体的简单实现：

9.2.3. koa-bodyparser

`koa-bodyparser` 中间件是将我们的 `post` 请求和表单提交的查询字符串转换成对象，并挂在 `ctx.request.body` 上，方便我们在其他中间件或接口处取值

JavaScript | 复制代码

```javascript
// 文件：my-koa-bodyparser.js
const querystring = require("querystring");

module.exports = function bodyParser() {
    return async (ctx, next) => {
        await new Promise((resolve, reject) => {
            // 存储数据的数组
            let dataArr = [];

            // 接收数据
            ctx.req.on("data", data => dataArr.push(data));

            // 整合数据并使用 Promise 成功
            ctx.req.on("end", () => {
                // 获取请求数据的类型 json 或表单
                let contentType = ctx.get("Content-Type");

                // 获取数据 Buffer 格式
                let data = Buffer.concat(dataArr).toString();

                if (contentType === "application/x-www-form-urlencoded") {
                    // 如果是表单提交，则将查询字符串转换成对象赋值给 ctx.request.body
                    ctx.request.body = querystring.parse(data);
                } else if (contentType === "applaction/json") {
                    // 如果是 json，则将字符串格式的对象转换成对象赋值给 ctx.request.body
                    ctx.request.body = JSON.parse(data);
                }

                // 执行成功的回调
                resolve();
            });
        });

        // 继续向下执行
        await next();
    };
};
```

9.2.4. koa-static

`koa-static` 中间件的作用是在服务器接到请求时，帮我们处理静态文件

JavaScript | 复制代码

```
const fs = require("fs");
const path = require("path");
const mime = require("mime");
const { promisify } = require("util");

// 将 stat 和 access 转换成 Promise
const stat = promisify(fs.stat);
const access = promisify(fs.access)

module.exports = function (dir) {
    return async (ctx, next) => {
        // 将访问的路由处理成绝对路径，这里要使用 join 因为有可能是 /
        let realPath = path.join(dir, ctx.path);

        try {
            // 获取 stat 对象
            let statObj = await stat(realPath);

            // 如果是文件，则设置文件类型并直接响应内容，否则当作文件夹寻找 index.html
            if (statObj.isFile()) {
                ctx.set("Content-Type", `${mime.getType()};charset=utf8`);
                ctx.body = fs.createReadStream(realPath);
            } else {
                let filename = path.join(realPath, "index.html");

                // 如果不存在该文件则执行 catch 中的 next 交给其他中间件处理
                await access(filename);

                // 存在设置文件类型并响应内容
                ctx.set("Content-Type", "text/html;charset=utf8");
                ctx.body = fs.createReadStream(filename);
            }
        } catch (e) {
            await next();
        }
    }
}
```

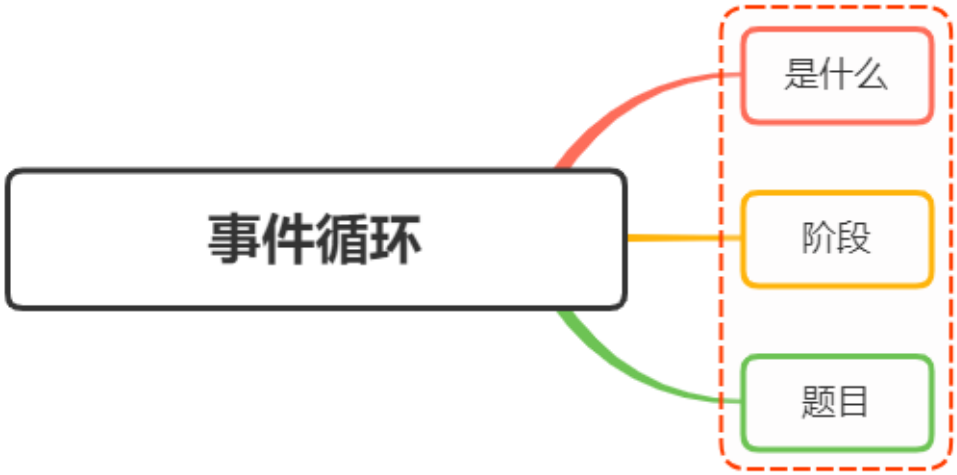
9.3. 总结

在实现中间件时候，单个中间件应该足够简单，职责单一，中间件的代码编写应该高效，必要的时候通过缓存重复获取数据

`koa` 本身比较简洁，但是通过中间件的机制能够实现各种所需要的功能，使得 `web` 应用具备良好的可拓展性和组合性

通过将公共逻辑的处理编写在中间件中，可以不用在每一个接口回调中做相同的代码编写，减少了冗杂代码，过程就如装饰者模式

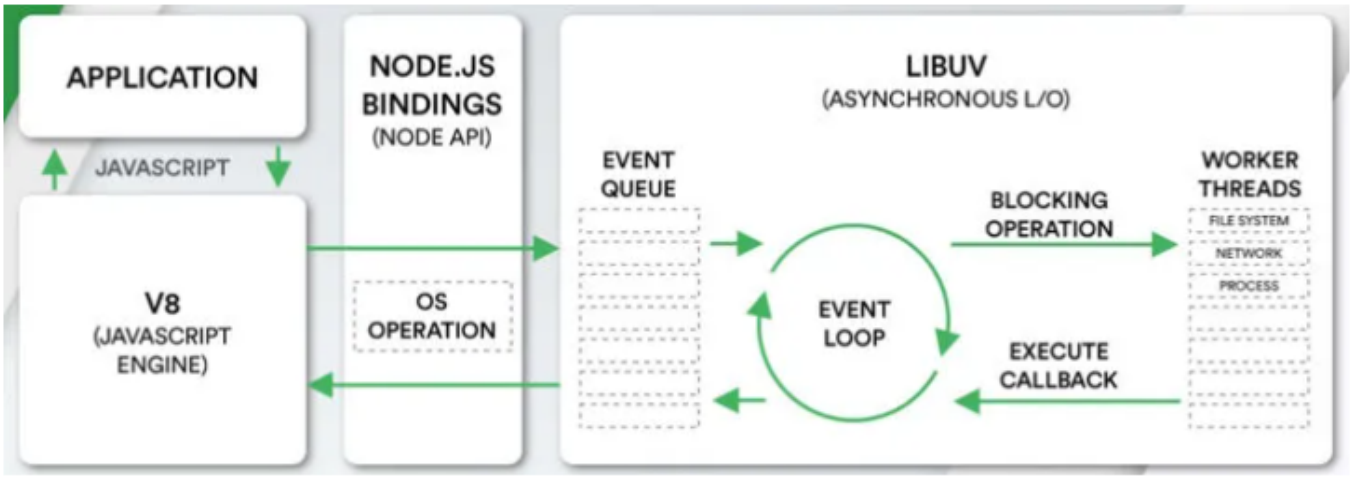
10. 说说对Nodejs中的事件循环机制理解?



10.1. 是什么

在浏览器事件循环中，我们了解到 `javascript` 在浏览器中的事件循环机制，其是根据 `HTML5` 定义的规范来实现

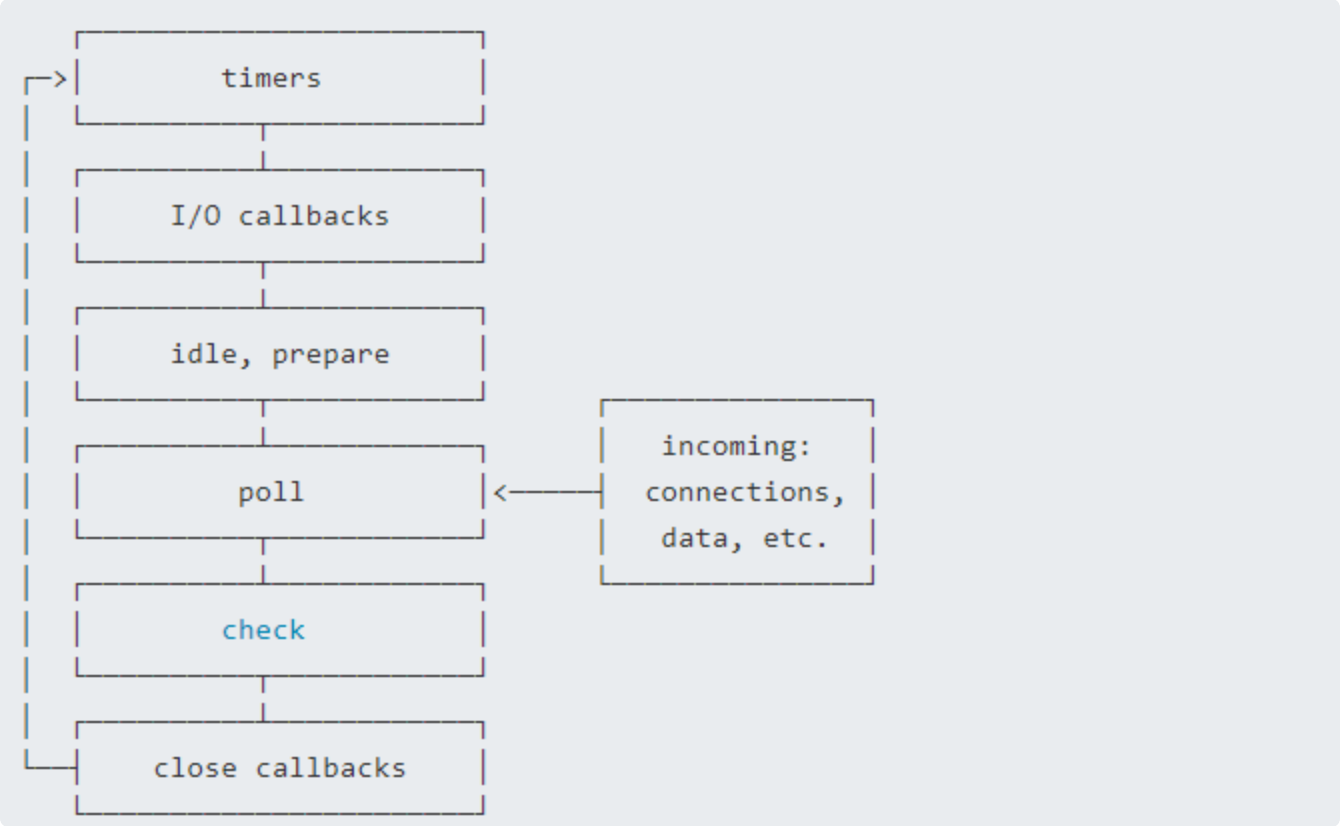
而在 `NodeJS` 中，事件循环是基于 `libuv` 实现， `libuv` 是一个多平台的专注于异步IO的库，如下图最右侧所示：



上图 `EVENT_QUEUE` 给人看起来只有一个队列，但 `EventLoop` 存在6个阶段，每个阶段都有对应的一个先进先出的回调队列

10.2. 流程

上节讲到事件循环分成了六个阶段，对应如下：

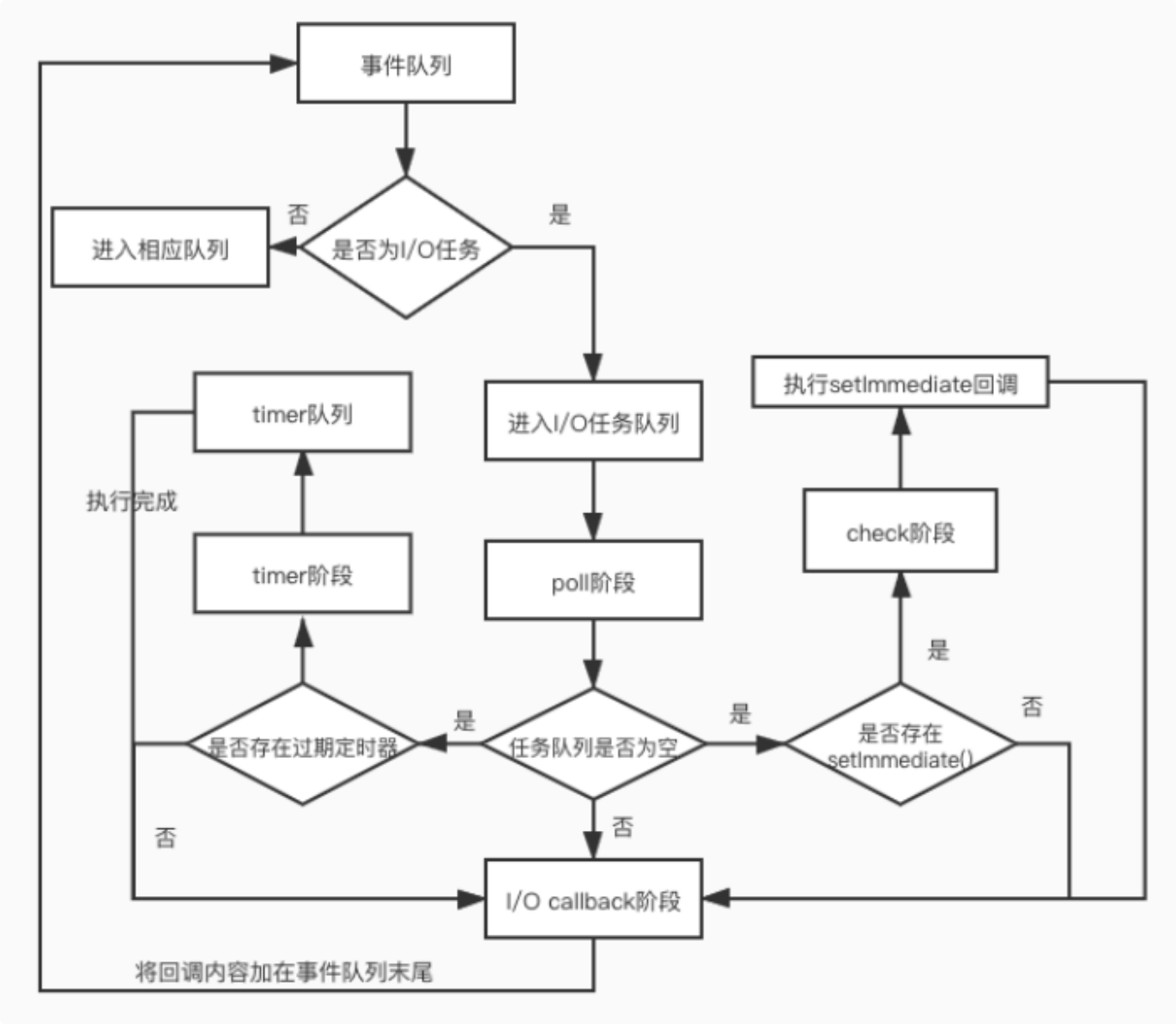


- timers阶段：这个阶段执行timer（setTimeout、setInterval）的回调
- 定时器检测阶段(timers)：本阶段执行 timer 的回调，即 setTimeout、setInterval 里面的回调函数
- I/O事件回调阶段(I/O callbacks)：执行延迟到下一个循环迭代的 I/O 回调，即上一轮循环中未被执行的一些I/O回调
- 闲置阶段(idle, prepare)：仅系统内部使用
- 轮询阶段(poll)：检索新的 I/O 事件;执行与 I/O 相关的回调（几乎所有情况下，除了关闭的回调函数，那些由计时器和 setImmediate() 调度的之外），其余情况 node 将在适当的时候在此阻塞
- 检查阶段(check)：setImmediate() 回调函数在这里执行
- 关闭事件回调阶段(close callback)：一些关闭的回调函数，如：socket.on('close', ...)

每个阶段对应一个队列，当事件循环进入某个阶段时, 将会在该阶段内执行回调，直到队列耗尽或者回调的最大数量已执行, 那么将进入下一个处理阶段

除了上述6个阶段，还存在 `process.nextTick` ，其不属于事件循环的任何一个阶段，它属于该阶段与下阶段之间的过渡, 即本阶段执行结束, 进入下一个阶段前, 所要执行的回调，类似插队

流程图如下所示：



在 `Node` 中，同样存在宏任务和微任务，与浏览器中的事件循环相似

微任务对应有：

- next tick queue：process.nextTick
- other queue：Promise的then回调、queueMicrotask

宏任务对应有：

- timer queue：setTimeout、setInterval
- poll queue：IO事件
- check queue：setImmediate

- close queue：close事件

其执行顺序为：

- next tick microtask queue
- other microtask queue
- timer queue
- poll queue
- check queue
- close queue

10.3. 题目

通过上面的学习，下面开始看看题目

JavaScript | 复制代码

```javascript
async function async1() {
    console.log('async1 start')
    await async2()
    console.log('async1 end')
}

async function async2() {
    console.log('async2')
}

console.log('script start')

setTimeout(function () {
    console.log('setTimeout0')
}, 0)

setTimeout(function () {
    console.log('setTimeout2')
}, 300)

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

process.nextTick(() => console.log('nextTick1'));

async1();

process.nextTick(() => console.log('nextTick2'));

new Promise(function (resolve) {
    console.log('promise1')
    resolve();
    console.log('promise2')
}).then(function () {
    console.log('promise3')
})

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

分析过程：

- 先找到同步任务，输出script start
- 遇到第一个 setTimeout，将里面的回调函数放到 timer 队列中
- 遇到第二个 setTimeout，300ms后将里面的回调函数放到 timer 队列中
- 遇到第一个setImmediate，将里面的回调函数放到 check 队列中

- 遇到第一个 nextTick，将其里面的回调函数放到本轮同步任务执行完毕后执行
- 执行 async1函数，输出 async1 start
- 执行 async2 函数，输出 async2，async2 后面的输出 async1 end进入微任务，等待下一轮的事件循环
- 遇到第二个，将其里面的回调函数放到本轮同步任务执行完毕后执行
- 遇到 new Promise，执行里面的立即执行函数，输出 promise1、promise2
- then里面的回调函数进入微任务队列
- 遇到同步任务，输出 script end
- 执行下一轮回到函数，先依次输出 nextTick 的函数，分别是 nextTick1、nextTick2
- 然后执行微任务队列，依次输出 async1 end、promise3
- 执行timer 队列，依次输出 setTimeout0
- 接着执行 check 队列，依次输出 setImmediate
- 300ms后，timer 队列存在任务，执行输出 setTimeout2

执行结果如下：

Plain Text | 复制代码

```
script start
async1 start
async2
promise1
promise2
script end
nextTick1
nextTick2
async1 end
promise3
setTimeout0
setImmediate
setTimeout2
```

最后有一道是关于 `setTimeout` 与 `setImmediate` 的输出顺序

```JavaScript
setTimeout(() => {
  console.log("setTimeout");
}, 0);

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

输出情况如下：

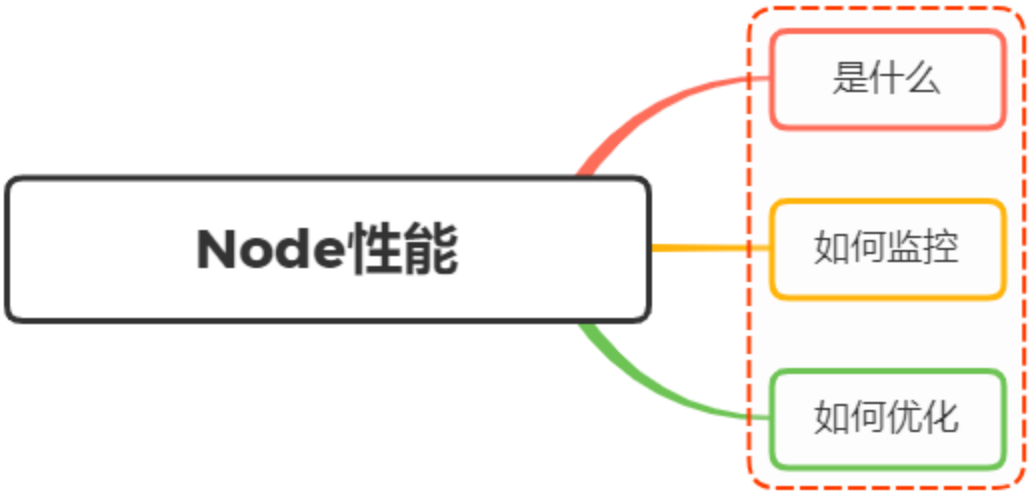
```
情况一：
setTimeout
setImmediate

情况二：
setImmediate
setTimeout
```

分析下流程：

- 外层同步代码一次性全部执行完，遇到异步API就塞到对应的阶段
- 遇到 `setTimeout`，虽然设置的是0毫秒触发，但实际上会被强制改成1ms，时间到了然后塞入 `times` 阶段
- 遇到 `setImmediate` 塞入 `check` 阶段
- 同步代码执行完毕，进入Event Loop
- 先进入 `times` 阶段，检查当前时间过去了1毫秒没有，如果过了1毫秒，满足 `setTimeout` 条件，执行回调，如果没过1毫秒，跳过
- 跳过空的阶段，进入check阶段，执行 `setImmediate` 回调

这里的关键在于这1ms，如果同步代码执行时间较长，进入 `Event Loop` 的时候1毫秒已经过了，`setTimeout` 先执行，如果1毫秒还没到，就先执行了 `setImmediate`

11. Node性能如何进行监控以及优化?



11.1. 是什么

`Node` 作为一门服务端语言，性能方面尤为重要，其衡量指标一般有如下：

- CPU
- 内存
- I/O
- 网络

11.1.1. CPU

主要分成了两部分：

- CPU负载：在某个时间段内，占用以及等待CPU的进程总数
- CPU使用率：CPU时间占用状况，等于 1 – 空闲CPU时间(idle time) / CPU总时间

这两个指标都是用来评估系统当前CPU的繁忙程度的量化指标

`Node` 应用一般不会消耗很多的 `CPU` ，如果 `CPU` 占用率高，则表明应用存在很多同步操作，导致异步任务回调被阻塞

11.1.2. 内存指标

内存是一个非常容易量化的指标。 内存占用率是评判一个系统的内存瓶颈的常见指标。 对于Node来说，内部内存堆栈的使用状态也是一个可以量化的指标

```JavaScript
// /app/lib/memory.js
const os = require('os');
// 获取当前Node内存堆栈情况
const { rss, heapUsed, heapTotal } = process.memoryUsage();
// 获取系统空闲内存
const sysFree = os.freemem();
// 获取系统总内存
const sysTotal = os.totalmem();

module.exports = {
  memory: () => {
    return {
      sys: 1 - sysFree / sysTotal,  // 系统内存占用率
      heap: heapUsed / headTotal,   // Node堆内存占用率
      node: rss / sysTotal,         // Node占用系统内存的比例
    }
  }
}
```

- rss：表示node进程占用的内存总量。
- heapTotal：表示堆内存的总量。
- heapUsed：实际堆内存的使用量。
- external ： 外部程序的内存使用量，包含Node核心的C++程序的内存使用量

在 `Node` 中，一个进程的最大内存容量为1.5GB。因此我们需要减少内存泄露

11.1.3. 磁盘 I/O

硬盘的 `IO` 开销是非常昂贵的，硬盘 IO 花费的 CPU 时钟周期是内存的 164000 倍

内存 `IO` 比磁盘 `IO` 快非常多，所以使用内存缓存数据是有效的优化方法。常用的工具如 `redis` 、 `memcached` 等

并不是所有数据都需要缓存，访问频率高，生成代价比较高的才考虑是否缓存，也就是说影响你性能瓶颈的考虑去缓存，并且而且缓存还有缓存雪崩、缓存穿透等问题要解决

11.2. 如何监控

关于性能方面的监控，一般情况都需要借助工具来实现