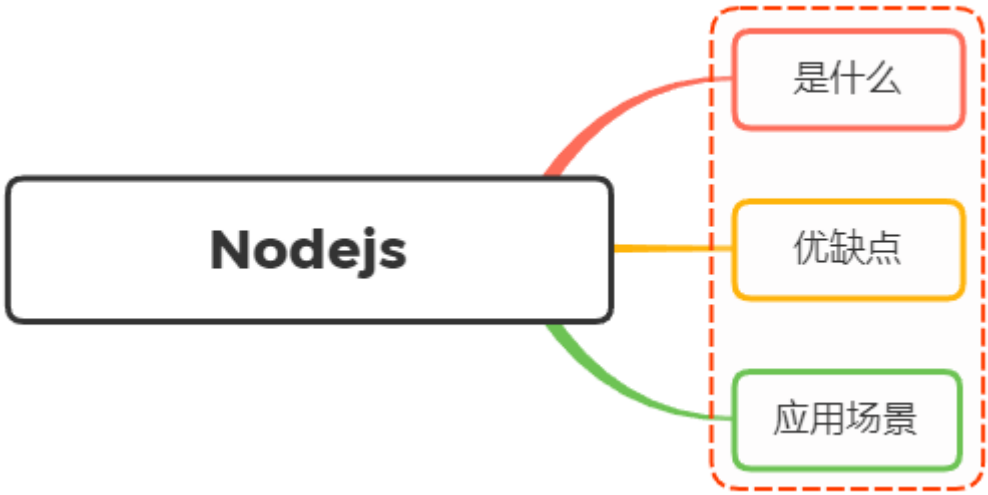
# Node.js面试真题（14题）

## 1. 说说你对Node.js 的理解？优缺点？应用场景？




### 1.1. 是什么

`Node.js` 是一个开源与跨平台的 `JavaScript` 运行时环境

在浏览器外运行 V8 JavaScript 引擎（Google Chrome 的内核），利用事件驱动、非阻塞和异步输入输出模型等技术提高性能

可以理解为 `Node.js` 就是一个服务器端的、非阻塞式I/O的、事件驱动的 `JavaScript` 运行环境

#### 1.1.1. 非阻塞异步

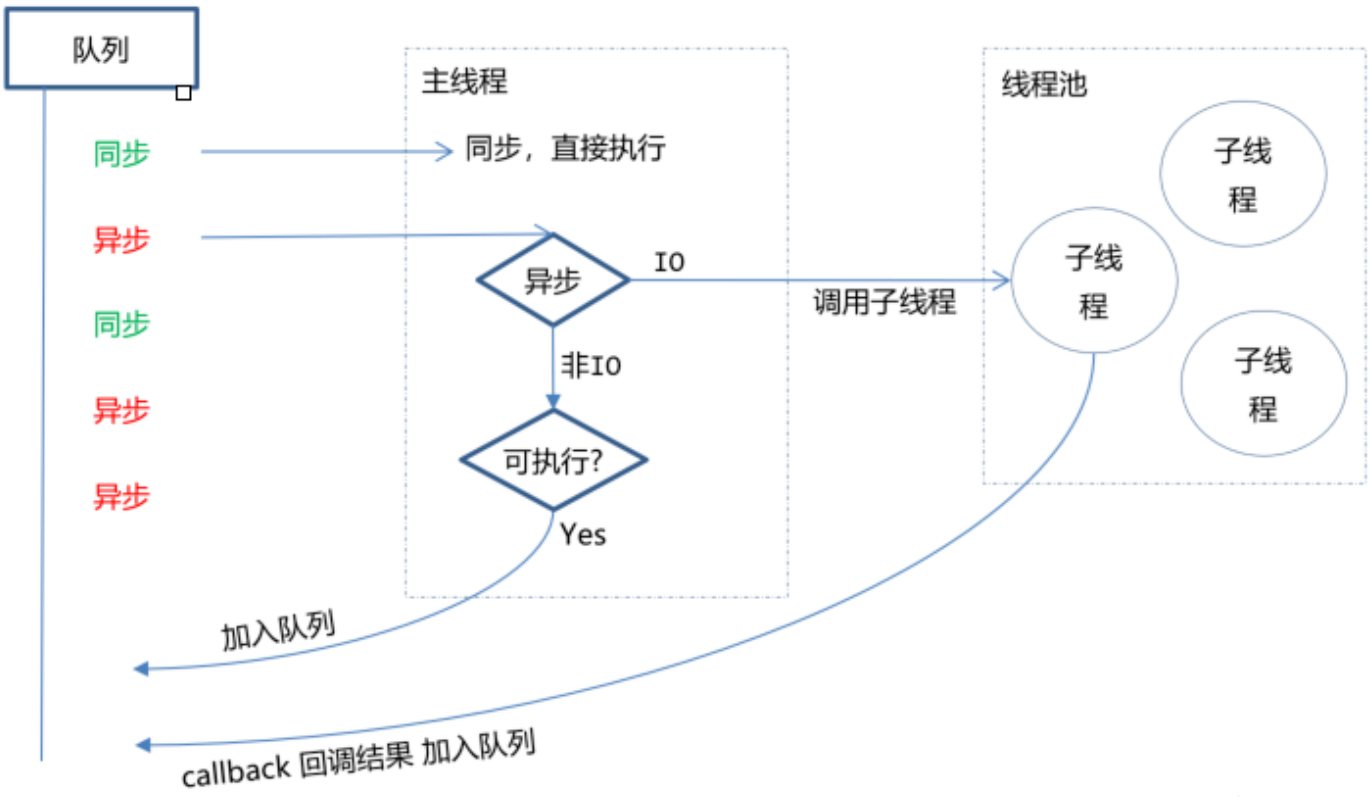
`Nodejs` 采用了非阻塞型 `I/O` 机制，在做 `I/O` 操作的时候不会造成任何的阻塞，当完成之后，以时间的形式通知执行操作

例如在执行了访问数据库的代码之后，将立即转而执行其后面的代码，把数据库返回结果的处理代码放在回调函数中，从而提高了程序的执行效率

#### 1.1.2. 事件驱动

事件驱动就是当进来一个新的请求的时，请求将会被压入一个事件队列中，然后通过一个循环来检测队列中的事件状态变化，如果检测到有状态变化的事件，那么就执行该事件对应的处理代码，一般都是回调函数

比如读取一个文件，文件读取完毕后，就会触发对应的状态，然后通过对应的回调函数来进行处理




## 1.2. 优缺点

优点：

- 处理高并发场景性能更佳
- 适合I/O密集型应用，值的是应用在运行极限时，CPU占用率仍然比较低，大部分时间是在做 I/O硬盘内存读写操作

因为 `Nodejs` 是单线程，带来的缺点有：

- 不适合CPU密集型应用
- 只支持单核CPU，不能充分利用CPU
- 可靠性低，一旦代码某个环节崩溃，整个系统都崩溃

## 1.3. 应用场景

借助 `Nodejs` 的特点和弊端，其应用场景分类如下：

- 善于 `I/O` ，不善于计算。因为Nodejs是一个单线程，如果计算（同步）太多，则会阻塞这个线程
- 大量并发的I/O，应用程序内部并不需要进行非常复杂的处理
- 与 websocket 配合，开发长连接的实时交互应用程序

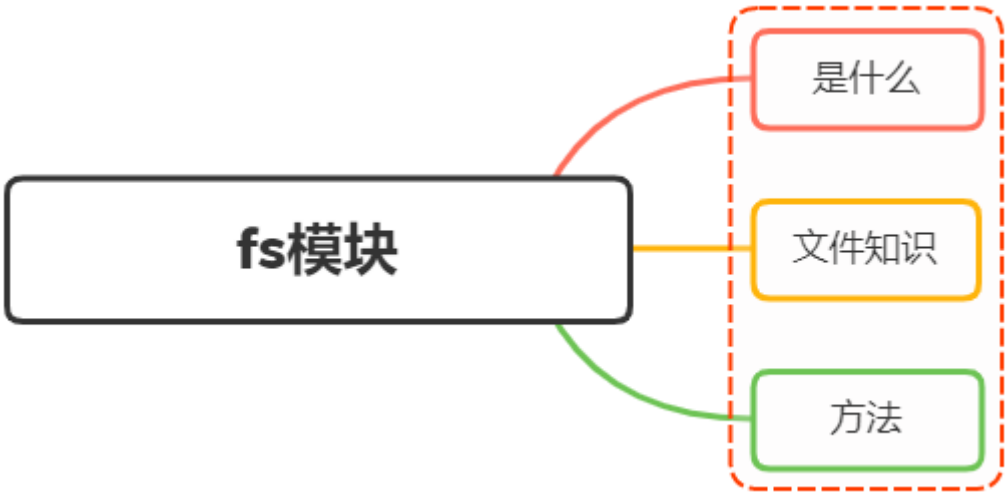
具体场景可以表现为如下：

- 第一大类：用户表单收集系统、后台管理系统、实时交互系统、考试系统、联网软件、高并发量的web应用程序
- 第二大类：基于web、canvas等多人联网游戏

- 第三大类：基于web的多人实时聊天客户端、聊天室、图文直播
- 第四大类：单页面浏览器应用程序
- 第五大类：操作数据库、为前端和移动端提供基于 `json` 的API

其实， `Nodejs` 能实现几乎一切的应用，只考虑适不适合使用它

# 2. 说说对 Node 中的 fs模块的理解? 有哪些常用方法




## 2.1. 是什么

fs（filesystem），该模块提供本地文件的读写能力，基本上是 `POSIX` 文件操作命令的简单包装

可以说，所有与文件的操作都是通过 `fs` 核心模块实现

导入模块如下：

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

这个模块对所有文件系统操作提供异步（不具有 `sync` 后缀）和同步（具有 `sync` 后缀）两种操作方式，而供开发者选择

### 2.1.1. 文件知识

在计算机中有关于文件的知识：

- 权限位 mode
- 标识位 flag
- 文件描述为 fd

### 2.1.2. 权限位 mode

| 权限分配 | 文件所有者 | | | 文件所属组 | | | 其他用户 | | |
|---|---|---|---|---|---|---|---|---|---|
| 权限项 | 读 | 写 | 执行 | 读 | 写 | 执行 | 读 | 写 | 执行 |
| 字符表示 | r | w | x | r | w | x | r | w | x |
| 数字表示 | 4 | 2 | 1 | 4 | 2 | 1 | 4 | 2 | 1 |

针对文件所有者、文件所属组、其他用户进行权限分配，其中类型又分成读、写和执行，具备权限位4、2、1，不具备权限为0

如在 `linux` 查看文件权限位：

```
1 drwxr-xr-x  1  PandaShen  197121  0  Jun 28 14:41  core
2 -rw-r--r--  1  PandaShen  197121  293  Jun 23 17:44  index.md
```

在开头前十位中， `d` 为文件夹， `-` 为文件，后九位就代表当前用户、用户所属组和其他用户的权限位，按每三位划分，分别代表读（r）、写（w）和执行（x），- 代表没有当前位对应的权限

### 2.1.3. 标识位

标识位代表着对文件的操作方式，如可读、可写、即可读又可写等等，如下表所示：

| 符号 | 含义 |
|---|---|
| r | 读取文件，如果文件不存在则抛出异常。 |
| r+ | 读取并写入文件，如果文件不存在则抛出异常。 |
| rs | 读取并写入文件，指示操作系统绕开本地文件系统缓存。 |
| w | 写入文件，文件不存在会被创建，存在则清空后写入。 |

| | |
|---|---|
| wx | 写入文件，排它方式打开。 |
| w+ | 读取并写入文件，文件不存在则创建文件，存在则清空后写入。 |
| wx+ | 和 w+ 类似，排他方式打开。 |
| a | 追加写入，文件不存在则创建文件。 |
| ax | 与 a 类似，排他方式打开。 |
| a+ | 读取并追加写入，不存在则创建。 |
| ax+ | 与 a+ 类似，排他方式打开。 |

### 2.1.4. 文件描述为 fd

操作系统会为每个打开的文件分配一个名为文件描述符的数值标识，文件操作使用这些文件描述符来识别与追踪每个特定的文件

`Window` 系统使用了一个不同但概念类似的机制来追踪资源，为方便用户， `NodeJS` 抽象了不同操作系统间的差异，为所有打开的文件分配了数值的文件描述符

在 `NodeJS` 中，每操作一个文件，文件描述符是递增的，文件描述符一般从 `3` 开始，因为前面有 `0` 、 `1` 、 `2` 三个比较特殊的描述符，分别代表 `process.stdin` （标准输入）、 `process.stdout` （标准输出）和 `process.stderr` （错误输出）

## 2.2. 方法

下面针对 `fs` 模块常用的方法进行展开：

- 文件读取
- 文件写入

- 文件追加写入
- 文件拷贝
- 创建目录

## 2.2.1. 文件读取

### 2.2.1.1. fs.readFileSync

同步读取，参数如下：

- 第一个参数为读取文件的路径或文件描述符
- 第二个参数为 options，默认值为 null，其中有 encoding（编码，默认为 null）和 flag（标识位，默认为 r），也可直接传入 encoding

结果为返回文件的内容

```
1 const fs = require("fs");
2 let buf = fs.readFileSync("1.txt");
3 let data = fs.readFileSync("1.txt", "utf8");
4 console.log(buf); // <Buffer 48 65 6c 6c 6f>
5 console.log(data); // Hello
```

### 2.2.1.2. fs.readFile

异步读取方法 `readFile` 与 `readFileSync` 的前两个参数相同，最后一个参数为回调函数，函数内有两个参数 `err` （错误）和 `data` （数据），该方法没有返回值，回调函数在读取文件成功后执行

```
1 const fs = require("fs");
2 fs.readFile("1.txt", "utf8", (err, data) => {
3    if(!err){
4        console.log(data); // Hello
5    }
6 });
```

## 2.2.2. 文件写入

### 2.2.2.1. writeFileSync

同步写入，有三个参数：

- 第一个参数为写入文件的路径或文件描述符
- 第二个参数为写入的数据，类型为 String 或 Buffer

- 第三个参数为 options，默认值为 null，其中有 encoding（编码，默认为 utf8）、 flag（标识位，默认为 w）和 mode（权限位，默认为 0o666），也可直接传入 encoding

```
const fs = require("fs");
fs.writeFileSync("2.txt", "Hello world");
let data = fs.readFileSync("2.txt", "utf8");
console.log(data); // Hello world
```

#### 2.2.2.2. writeFile

异步写入，`writeFile` 与 `writeFileSync` 的前三个参数相同，最后一个参数为回调函数，函数内有一个参数 `err` （错误），回调函数在文件写入数据成功后执行

```
const fs = require("fs");
fs.writeFile("2.txt", "Hello world", err => {
    if (!err) {
        fs.readFile("2.txt", "utf8", (err, data) => {
            console.log(data); // Hello world
        });
    }
});
```

### 2.2.3. 文件追加写入

#### 2.2.3.1. appendFileSync

参数如下：

- 第一个参数为写入文件的路径或文件描述符
- 第二个参数为写入的数据，类型为 String 或 Buffer
- 第三个参数为 options，默认值为 null，其中有 encoding（编码，默认为 utf8）、 flag（标识位，默认为 a）和 mode（权限位，默认为 0o666），也可直接传入 encoding

```
const fs = require("fs");
fs.appendFileSync("3.txt", " world");
let data = fs.readFileSync("3.txt", "utf8");
```

#### 2.2.3.2. appendFile

异步追加写入方法 `appendFile` 与 `appendFileSync` 的前三个参数相同，最后一个参数为回调函数，函数内有一个参数 `err` （错误），回调函数在文件追加写入数据成功后执行

```
const fs = require("fs");
fs.appendFile("3.txt", " world", err => {
    if (!err) {
        fs.readFile("3.txt", "utf8", (err, data) => {
            console.log(data); // Hello world
        });
    }
});
```

## 2.2.4. 文件拷贝

### 2.2.4.1. copyFileSync

同步拷贝

```
const fs = require("fs");
fs.copyFileSync("3.txt", "4.txt");
let data = fs.readFileSync("4.txt", "utf8");
console.log(data); // Hello world
```

### 2.2.4.2. copyFile

异步拷贝

```
const fs = require("fs");
fs.copyFile("3.txt", "4.txt", () => {
    fs.readFile("4.txt", "utf8", (err, data) => {
        console.log(data); // Hello world
    });
});
```

## 2.2.5. 创建目录

### 2.2.5.1. mkdirSync

同步创建，参数为一个目录的路径，没有返回值，在创建目录的过程中，必须保证传入的路径前面的文件目录都存在，否则会抛出异常

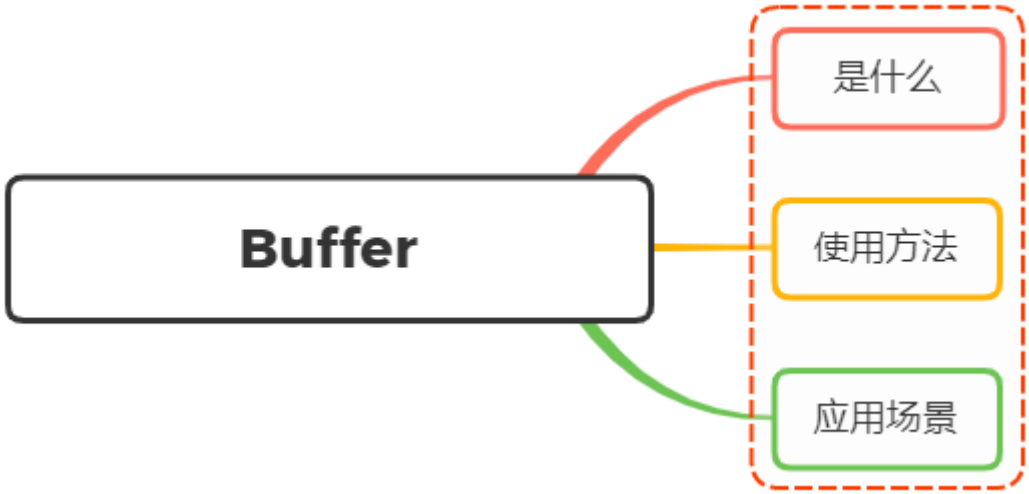
```
// 假设已经有了 a 文件夹和 a 下的 b 文件夹
fs.mkdirSync("a/b/c")
```

#### 2.2.5.2. mkdir

异步创建，第二个参数为回调函数

```
fs.mkdir("a/b/c", err => {
    if (!err) console.log("创建成功");
});
```

# 3. 说说对 Node 中的 Buffer 的理解？应用场景？




## 3.1. 是什么

在 `Node` 应用中，需要处理网络协议、操作数据库、处理图片、接收上传文件等，在网络流和文件的操作中，要处理大量二进制数据，而 `Buffer` 就是在内存中开辟一片区域（初次初始化为8KB），用来存放二进制数据

在上述操作中都会存在数据流动，每个数据流动的过程中，都会有一个最小或最大数据量

如果数据到达的速度比进程消耗的速度快，那么少数早到达的数据会处于等待区等候被处理。反之，如果数据到达的速度比进程消耗的数据慢，那么早先到达的数据需要等待一定量的数据到达之后才能被处理

这里的等待区就指的缓冲区（Buffer），它是计算机中的一个小物理单位，通常位于计算机的 `RAM` 中

简单来讲， `Nodejs` 不能控制数据传输的速度和到达时间，只能决定何时发送数据，如果还没到发送时间，则将数据放在 `Buffer` 中，即在 `RAM` 中，直至将它们发送完毕

上面讲到了 `Buffer` 是用来存储二进制数据，其的形式可以理解成一个数组，数组中的每一项，都可以保存8位二进制： `00000000` ，也就是一个字节

例如：

```
const buffer = Buffer.from("why")
```

其存储过程如下图所示：


## 3.2. 使用方法

`Buffer` 类在全局作用域中，无须 `require` 导入

创建 `Buffer` 的方法有很多种，我们讲讲下面的两种常见的形式：

- Buffer.from()
- Buffer.alloc()

### 3.2.1. Buffer.from()

```
const b1 = Buffer.from('10');
const b2 = Buffer.from('10', 'utf8');
const b3 = Buffer.from([10]);
const b4 = Buffer.from(b3);
console.log(b1, b2, b3, b4); // <Buffer 31 30> <Buffer 31 30> <Buffer 0a>
<Buffer 0a>
```

### 3.2.2. Buffer.alloc()

```
const bAlloc1 = Buffer.alloc(10); // 创建一个大小为 10 个字节的缓冲区
const bAlloc2 = Buffer.alloc(10, 1); // 建一个长度为 10 的 Buffer,其中全部填充了值
为
1
 的字节
console.log(bAlloc1); // <Buffer 00 00 00 00 00 00 00 00 00 00>
console.log(bAlloc2); // <Buffer 01 01 01 01 01 01 01 01 01 01>
```

在上面创建 `buffer` 后，则能够 `toString` 的形式进行交互，默认情况下采取 `utf8` 字符编码形式，如下

```
const buffer = Buffer.from("你好");
console.log(buffer);
// <Buffer e4 bd a0 e5 a5 bd>
const str = buffer.toString();
console.log(str);
// 你好
```

如果编码与解码不是相同的格式则会出现乱码的情况，如下：

```
const buffer = Buffer.from("你好","utf-8 ");
console.log(buffer);
// <Buffer e4 bd a0 e5 a5 bd>
const str = buffer.toString("ascii");
console.log(str);
// d= e%=
```

当设定的范围导致字符串被截断的时候，也会存在乱码情况，如下：

```
const buf = Buffer.from('Node.js 技术栈', 'UTF-8');
console.log(buf)          // <Buffer 4e 6f 64 65 2e 6a 73 20 e6 8a 80 e6 9c af
e6 a0 88>
console.log(buf.length)   // 17
console.log(buf.toString('UTF-8', 0, 9))  // Node.js
console.log(buf.toString('UTF-8', 0, 11)) // Node.js 技
```

所支持的字符集有如下：

- ascii：仅支持 7 位 ASCII 数据，如果设置去掉高位的话，这种编码是非常快的
- utf8：多字节编码的 Unicode 字符，许多网页和其他文档格式都使用 UTF-8
- utf16le：2 或 4 个字节，小字节序编码的 Unicode 字符，支持代理对（U+10000至 U+10FFFF）
- ucs2，utf16le 的别名
- base64：Base64 编码
- latin：一种把 Buffer 编码成一字节编码的字符串的方式
- binary：latin1 的别名，
- hex：将每个字节编码为两个十六进制字符

## 3.3. 应用场景

`Buffer` 的应用场景常常与流的概念联系在一起，例如有如下：

- I/O操作
- 加密解密
- zlib.js

### 3.3.1. I/O操作

通过流的形式，将一个文件的内容读取到另外一个文件

```
1 const fs = require('fs');
2 const inputStream = fs.createReadStream('input.txt'); // 创建可读流
3 const outputStream = fs.createWriteStream('output.txt'); // 创建可写流
4 inputStream.pipe(outputStream); // 管道读写
```

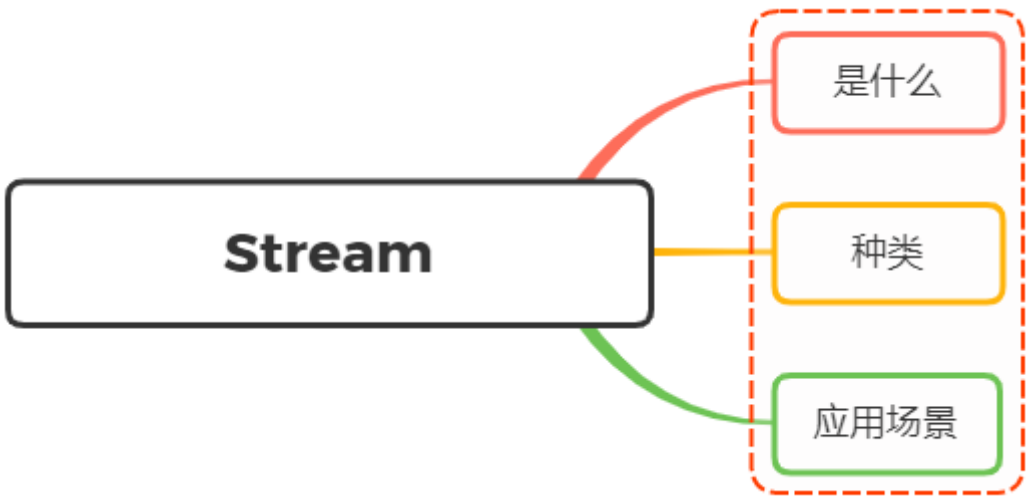
### 3.3.2. 加解密

在一些加解密算法中会遇到使用 `Buffer`，例如 `crypto.createCipheriv` 的第二个参数 `key` 为 `string` 或 `Buffer` 类型

### 3.3.3. zlib.js

`zlib.js` 为 `Node.js` 的核心库之一，其利用了缓冲区（`Buffer`）的功能来操作二进制数据流，提供了压缩或解压功能

## 4. 说说对 Node 中的 Stream 的理解？应用场景？




### 4.1. 是什么

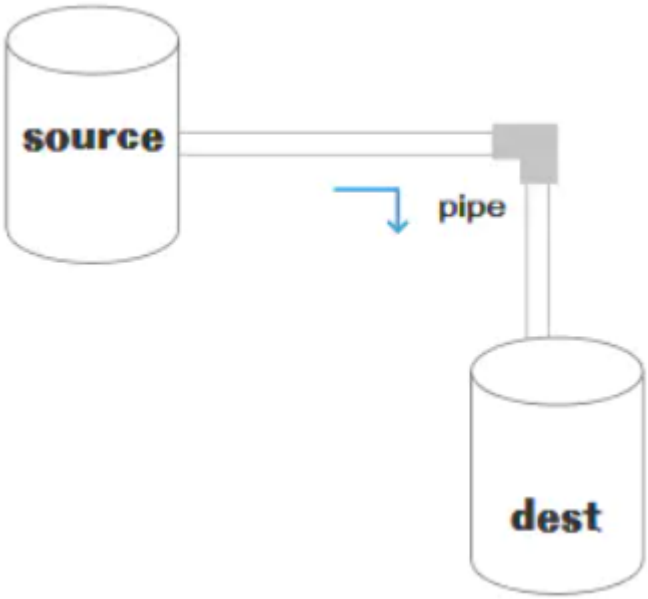
流（Stream），是一个数据传输手段，是端到端信息交换的一种方式，而且是有顺序的,是逐块读取数据、处理内容，用于顺序读取输入或写入输出

`Node.js` 中很多对象都实现了流，总之它是会冒数据（以 `Buffer` 为单位）

它的独特之处在于，它不像传统的程序那样一次将一个文件读入内存，而是逐块读取数据、处理其内容，而不是将其全部保存在内存中

流可以分成三部分： `source` 、 `dest` 、 `pipe`

在 `source` 和 `dest` 之间有一个连接的管道 `pipe` ,它的基本语法是 `source.pipe(dest)` ， `source` 和 `dest` 就是通过pipe连接，让数据从 `source` 流向了 `dest` ，如下图所示：




## 4.2. 种类

在 `NodeJS` ，几乎所有的地方都使用到了流的概念，分成四个种类：

- 可写流：可写入数据的流。例如 fs.createWriteStream() 可以使用流将数据写入文件
- 可读流： 可读取数据的流。例如fs.createReadStream() 可以从文件读取内容
- 双工流： 既可读又可写的流。例如 net.Socket
- 转换流： 可以在数据写入和读取时修改或转换数据的流。例如，在文件压缩操作中，可以向文件写入压缩数据，并从文件中读取解压数据

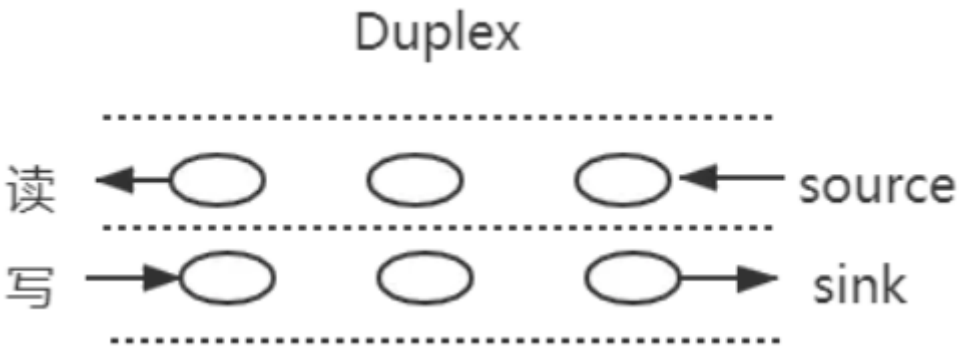
在 `NodeJS` 中 `HTTP` 服务器模块中， `request` 是可读流， `response` 是可写流。还有 `fs` 模块，能同时处理可读和可写文件流

可读流和可写流都是单向的，比较容易理解，而另外两个是双向的

### 4.2.1. 双工流

之前了解过 `websocket` 通信，是一个全双工通信，发送方和接受方都是各自独立的方法，发送和接收都没有任何关系

如下图所示：

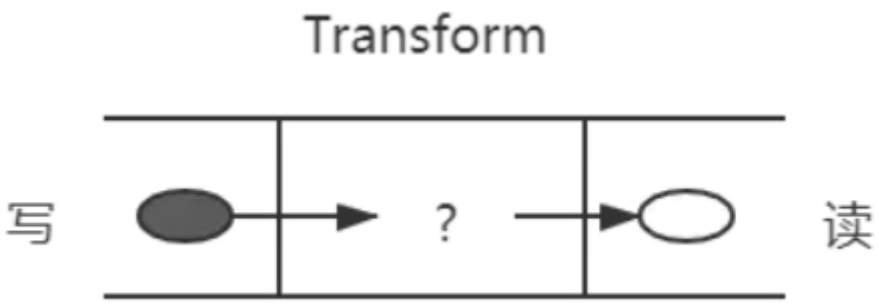



基本代码如下：

```
const { Duplex } = require('stream');
const myDuplex = new Duplex({
  read(size) {
    // ...
  },
  write(chunk, encoding, callback) {
    // ...
  }
});
```

### 4.2.2. 双工流

双工流的演示图如下所示：




除了上述压缩包的例子，还比如一个 `babel` ，把 `es6` 转换为，我们在左边写入 `es6` ，从右边读取 `es5`

基本代码如下所示：

```
const { Transform } = require('stream');
const myTransform = new Transform({
  transform(chunk, encoding, callback) {
    // ...
  }
});
```

## 4.3. 应用场景

`stream` 的应用场景主要就是处理 `IO` 操作，而 `http` 请求和文件操作都属于 `IO` 操作

试想一下，如果一次 `IO` 操作过大，硬件的开销就过大，而将此次大的 `IO` 操作进行分段操作，让数据像水管一样流动，直到流动完成

常见的场景有：

- get请求返回文件给客户端
- 文件操作
- 一些打包工具的底层操作

### 4.3.1. get请求返回文件给客户端

使用 `stream` 流返回文件， `res` 也是一个 `stream` 对象，通过 `pipe` 管道将文件数据返回

```
const server = http.createServer(function (req, res) {
    const method = req.method; // 获取请求方法
    if (method === 'GET') { // get 请求
        const fileName = path.resolve(__dirname, 'data.txt');
        let stream = fs.createReadStream(fileName);
        stream.pipe(res); // 将 res 作为 stream 的 dest
    }
});
server.listen(8000);
```

### 4.3.2. 文件操作

创建一个可读数据流 `readStream` ，一个可写数据流 `writeStream` ，通过 `pipe` 管道把数据流转过去

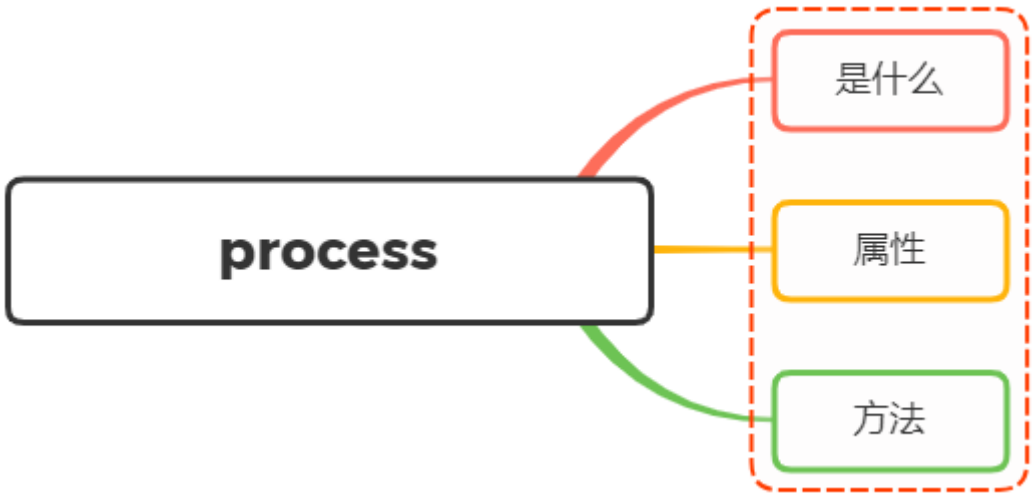
```
const fs = require('fs')
const path = require('path')
// 两个文件名
const fileName1 = path.resolve(__dirname, 'data.txt')
const fileName2 = path.resolve(__dirname, 'data-bak.txt')
// 读取文件的 stream 对象
const readStream = fs.createReadStream(fileName1)
// 写入文件的 stream 对象
const writeStream = fs.createWriteStream(fileName2)
// 通过 pipe执行拷贝，数据流转
readStream.pipe(writeStream)
```

```
// 数据读取完成监听，即拷贝完成
readStream.on('end', function () {
    console.log('拷贝完成')
})
```

### 4.3.3. 一些打包工具的底层操作

目前一些比较火的前端打包构建工具，都是通过 `node.js` 编写的，打包和构建的过程肯定是文件频繁操作的过程，离不来 `stream` ，如 `gulp`

## 5. 说说对 Node 中的 process 的理解？有哪些常用方法？




### 5.1. 是什么

`process` 对象是一个全局变量，提供了有关当前 `Node.js` 进程的信息并对其进行控制，作为一个全局变量

我们都知道，进程计算机系统进行资源分配和调度的基本单位，是操作系统结构的基础，是线程的容器

当我们启动一个 `js` 文件，实际就是开启了一个服务进程，每个进程都拥有自己的独立空间地址、数据栈，像另一个进程无法访问当前进程的变量、数据结构，只有数据通信后，进程之间才可以数据共享

由于 `JavaScript` 是一个单线程语言，所以通过 `node xxx` 启动一个文件后，只有一条主线程

### 5.2. 属性与方法

关于 `process` 常见的属性有如下：

- process.env：环境变量，例如通过 `process.env.NODE_ENV 获取不同环境项目配置信息
- process.nextTick：这个在谈及 `EventLoop` 时经常为会提到
- process.pid：获取当前进程id

- process.ppid：当前进程对应的父进程
- process.cwd()：获取当前进程工作目录，
- process.platform：获取当前进程运行的操作系统平台
- process.uptime()：当前进程已运行时间，例如：pm2 守护进程的 uptime 值
- 进程事件： process.on(‘uncaughtException’,cb) 捕获异常信息、 process.on(‘exit’,cb）进程推出监听
- 三个标准流： process.stdout 标准输出、 process.stdin 标准输入、 process.stderr 标准错误输出
- process.title 指定进程名称，有的时候需要给进程指定一个名称

下面再稍微介绍下某些方法的使用：

### 5.2.1. process.cwd()

返回当前 `Node` 进程执行的目录

一个 `Node` 模块 `A` 通过 NPM 发布，项目 `B` 中使用了模块 `A`。在 `A` 中需要操作 `B` 项目下的文件时，就可以用 `process.cwd()` 来获取 `B` 项目的路径

### 5.2.2. process.argv

在终端通过 Node 执行命令的时候，通过 `process.argv` 可以获取传入的命令行参数，返回值是一个数组：

- 0: Node 路径（一般用不到，直接忽略）
- 1: 被执行的 JS 文件路径（一般用不到，直接忽略）
- 2~n: 真实传入命令的参数

所以，我们只要从 `process.argv[2]` 开始获取就好了

```
1 const args = process.argv.slice(2);
```

### 5.2.3. process.env

返回一个对象，存储当前环境相关的所有信息，一般很少直接用到。

一般我们会在 `process.env` 上挂载一些变量标识当前的环境。比如最常见的用 `process.env.NODE_ENV` 区分 `development` 和 `production`

在 `vue-cli` 的源码中也经常会看到 `process.env.VUE_CLI_DEBUG` 标识当前是不是 `DEBUG` 模式

### 5.2.4. process.nextTick()

我们知道 `NodeJs` 是基于事件轮询，在这个过程中，同一时间只会处理一件事情

在这种处理模式下， `process.nextTick()` 就是定义出一个动作，并且让这个动作在下一个事件轮询的时间点上执行

例如下面例子将一个 `foo` 函数在下一个时间点调用

```
function foo() {
    console.error('foo');
}
process.nextTick(foo);
console.error('bar');
```

输出结果为 `bar` 、 `foo`

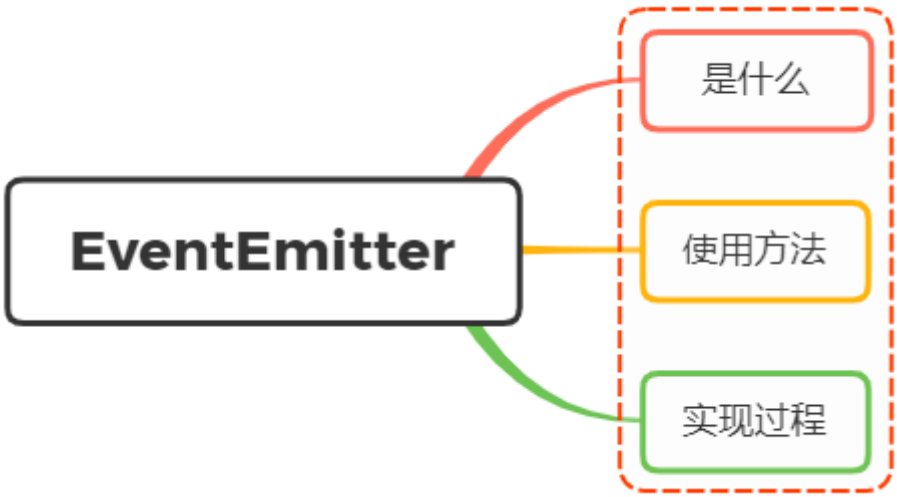
虽然下述方式也能实现同样效果：

```
setTimeout(foo, 0);
console.log('bar');
```

两者区别在于：

- process.nextTick()会在这一次event loop的call stack清空后（下一次event loop开始前）再调用callback
- setTimeout()是并不知道什么时候call stack清空的，所以何时调用callback函数是不确定的

# 6. 说说Node中的EventEmitter? 如何实现一个EventEmitter?




## 6.1. 是什么

我们了解到，`Node` 采用了事件驱动机制，而 `EventEmitter` 就是 `Node` 实现事件驱动的基础

在 `EventEmitter` 的基础上，`Node` 几乎所有的模块都继承了这个类，这些模块拥有了自己的事件，可以绑定／触发监听器，实现了异步操作

`Node.js` 里面的许多对象都会分发事件，比如 fs.readStream 对象会在文件被打开的时候触发一个事件

这些产生事件的对象都是 events.EventEmitter 的实例，这些对象有一个 eventEmitter.on() 函数，用于将一个或多个函数绑定到命名事件上

## 6.2. 使用方法

`Node` 的 `events` 模块只提供了一个 `EventEmitter` 类，这个类实现了 `Node` 异步事件驱动架构的基本模式——观察者模式

在这种模式中，被观察者(主体)维护着一组其他对象派来(注册)的观察者，有新的对象对主体感兴趣就注册观察者，不感兴趣就取消订阅，主体有更新的话就依次通知观察者们

基本代码如下所示：

```
const EventEmitter = require('events')
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter()
function callback() {
    console.log('触发了event事件! ')
}
myEmitter.on('event', callback)
myEmitter.emit('event')
myEmitter.removeListener('event', callback);
```

通过实例对象的 `on` 方法注册一个名为 `event` 的事件，通过 `emit` 方法触发该事件，而 `removeListener` 用于取消事件的监听

关于其常见的方法如下：

- emitter.addListener/on(eventName, listener)：添加类型为 eventName 的监听事件到事件数组尾部
- emitter.prependListener(eventName, listener)：添加类型为 eventName 的监听事件到事件数组头部
- emitter.emit(eventName[, ...args])：触发类型为 eventName 的监听事件
- emitter.removeListener/off(eventName, listener)：移除类型为 eventName 的监听事件
- emitter.once(eventName, listener)：添加类型为 eventName 的监听事件，以后只能执行一次并删除

- emitter.removeAllListeners([eventName])： 移除全部类型为 eventName 的监听事件

## 6.3. 实现过程

通过上面的方法了解， `EventEmitter` 是一个构造函数，内部存在一个包含所有事件的对象

```
class EventEmitter {
    constructor() {
        this.events = {};
    }
}
```

其中 `events` 存放的监听事件的函数的结构如下：

```
{
  "event1": [f1,f2,f3],
  "event2": [f4,f5],
  ...
}
```

然后开始一步步实现实例方法，首先是 `emit` ，第一个参数为事件的类型，第二个参数开始为触发事件函数的参数，实现如下：

```
emit(type, ...args) {
    this.events[type].forEach((item) => {
        Reflect.apply(item, this, args);
    });
}
```

当实现了 `emit` 方法之后，然后实现 `on` 、 `addListener` 、 `prependListener` 这三个实例方法，都是添加事件监听触发函数，实现也是大同小异

```
on(type, handler) {
    if (!this.events[type]) {
        this.events[type] = [];
    }
    this.events[type].push(handler);
}
addListener(type,handler){
    this.on(type,handler)
```

```
}
prependListener(type, handler) {
    if (!this.events[type]) {
        this.events[type] = [];
    }
    this.events[type].unshift(handler);
}
```

紧接着就是实现事件监听的方法 `removeListener/on`

```
removeListener(type, handler) {
    if (!this.events[type]) {
        return;
    }
    this.events[type] = this.events[type].filter(item => item !== handler);
}
off(type,handler){
    this.removeListener(type,handler)
}
```

最后再来实现 `once` 方法， 再传入事件监听处理函数的时候进行封装，利用闭包的特性维护当前状态，通过 `fired` 属性值判断事件函数是否执行过

```
once(type, handler) {
    this.on(type, this._onceWrap(type, handler, this));
  }
  _onceWrap(type, handler, target) {
    const state = { fired: false, handler, type , target};
    const wrapFn = this._onceWrapper.bind(state);
    state.wrapFn = wrapFn;
    return wrapFn;
  }
  _onceWrapper(...args) {
    if (!this.fired) {
      this.fired = true;
      Reflect.apply(this.handler, this.target, args);
      this.target.off(this.type, this.wrapFn);
    }
 }
```

完整代码如下：

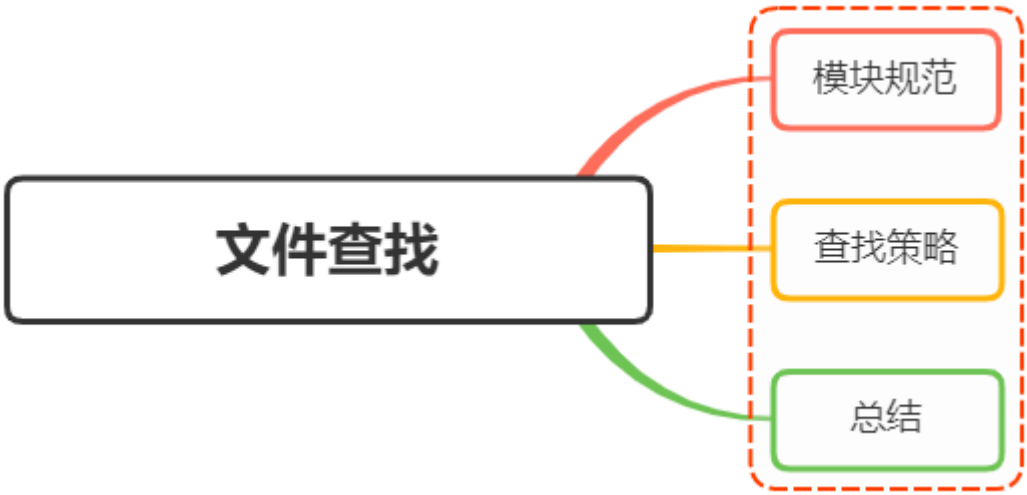
```
class EventEmitter {
    constructor() {
        this.events = {};
    }
    on(type, handler) {
        if (!this.events[type]) {
            this.events[type] = [];
        }
        this.events[type].push(handler);
    }
    addListener(type,handler){
        this.on(type,handler)
    }
    prependListener(type, handler) {
        if (!this.events[type]) {
            this.events[type] = [];
        }
        this.events[type].unshift(handler);
    }
    removeListener(type, handler) {
        if (!this.events[type]) {
            return;
        }
        this.events[type] = this.events[type].filter(item => item !== handler);
    }
    off(type,handler){
        this.removeListener(type,handler)
    }
    emit(type, ...args) {
        this.events[type].forEach((item) => {
            Reflect.apply(item, this, args);
        });
    }
    once(type, handler) {
        this.on(type, this._onceWrap(type, handler, this));
    }
    _onceWrap(type, handler, target) {
        const state = { fired: false, handler, type , target};
        const wrapFn = this._onceWrapper.bind(state);
        state.wrapFn = wrapFn;
        return wrapFn;
    }
    _onceWrapper(...args) {
        if (!this.fired) {
            this.fired = true;
            Reflect.apply(this.handler, this.target, args);
            this.target.off(this.type, this.wrapFn);
```

```
        }
    }
}
```

测试代码如下：

```
const ee = new EventEmitter();
// 注册所有事件
ee.once('wakeUp', (name) => { console.log(
${name} 1
); });
ee.on('eat', (name) => { console.log(
${name} 2
) });
ee.on('eat', (name) => { console.log(
${name} 3
) });
const meetingFn = (name) => { console.log(
${name} 4
) };
ee.on('work', meetingFn);
ee.on('work', (name) => { console.log(
${name} 5
) });
ee.emit('wakeUp', 'xx');
ee.emit('wakeUp', 'xx');          // 第二次没有触发
ee.emit('eat', 'xx');
ee.emit('work', 'xx');
ee.off('work', meetingFn);         // 移除事件
ee.emit('work', 'xx');            // 再次工作
```

## 7. 说说 Node 文件查找的优先级以及 Require 方法的文件查找策略?




## 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

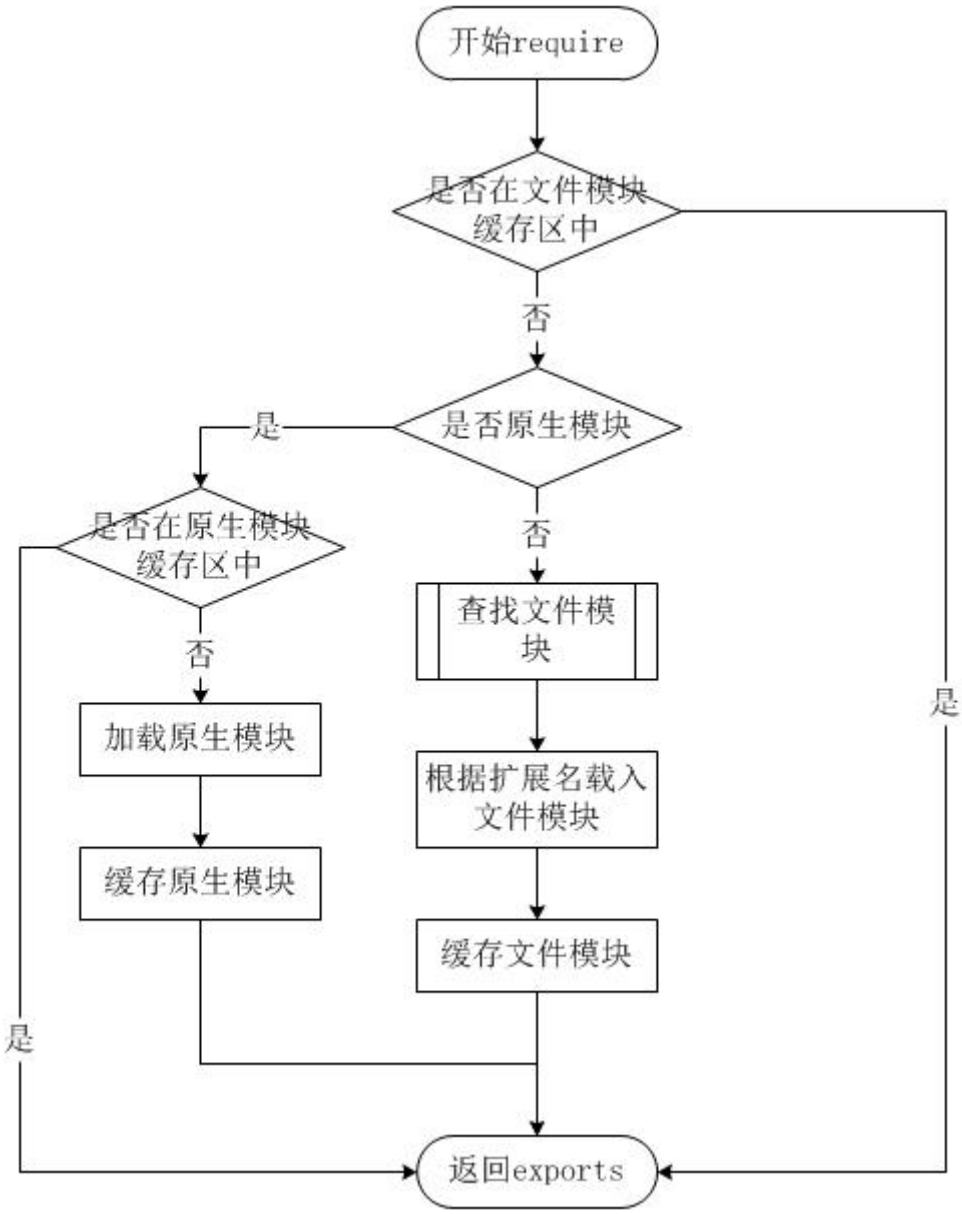
而模块化的核心是导出与导入，在 `Node` 中通过 `exports` 与 `module.exports` 负责对模块中的内容进行导出，通过 `require` 函数导入其他模块（自定义模块、系统模块、第三方库模块）中的内容

## 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:

```
{ "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：

```
[ '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

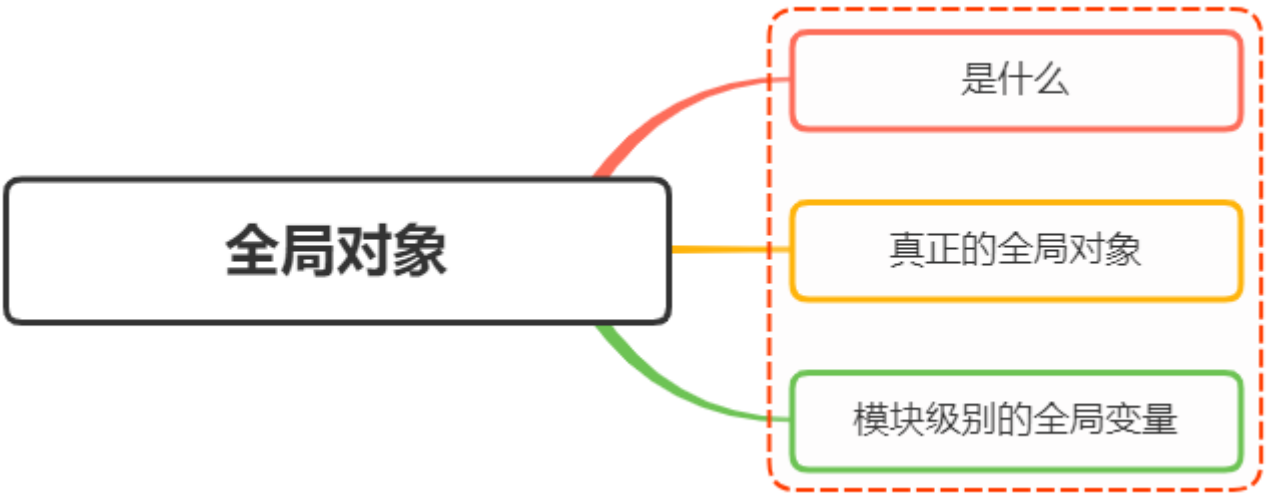
## 7.3. 总结

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：

```
1  node index.js 参数1 参数2 参数3
```

index.js文件如下：

```
1 process.argv.forEach((val, index) => {
2   console.log(
3 ${index}: ${val}
4 );
5 });
```

输出如下：

```
1 /usr/local/bin/node
2 /Users/mjr/work/node/process-args.js
3 参数1
4 参数2
5 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

```
1 console.log("hello");
```

清空控制台：console.clear

```
1 console.clear
```

打印函数的调用栈：console.trace

```
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

```
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

```
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

```
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`：

```
console.log(__dirname);
// 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`：

```
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

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

#### 8.2.2.4. module

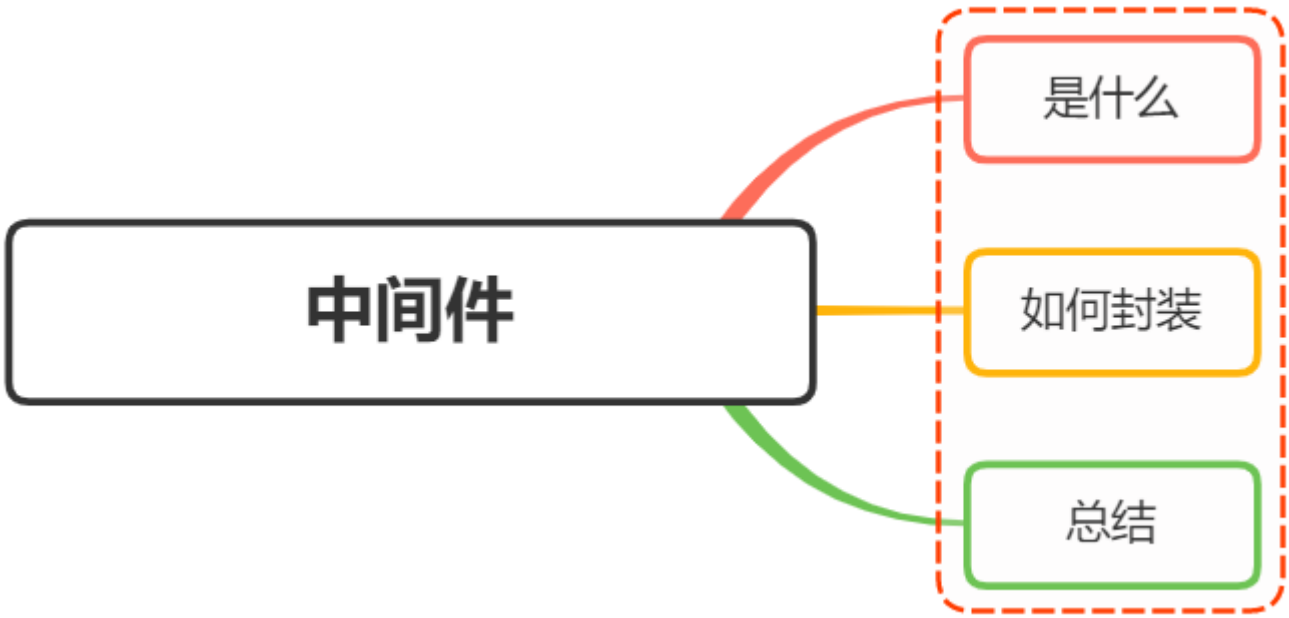
对当前模块的引用，通过 `module.exports` 用于指定一个模块所导出的内容，即可以通过 `require()` 访问的内容

#### 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?

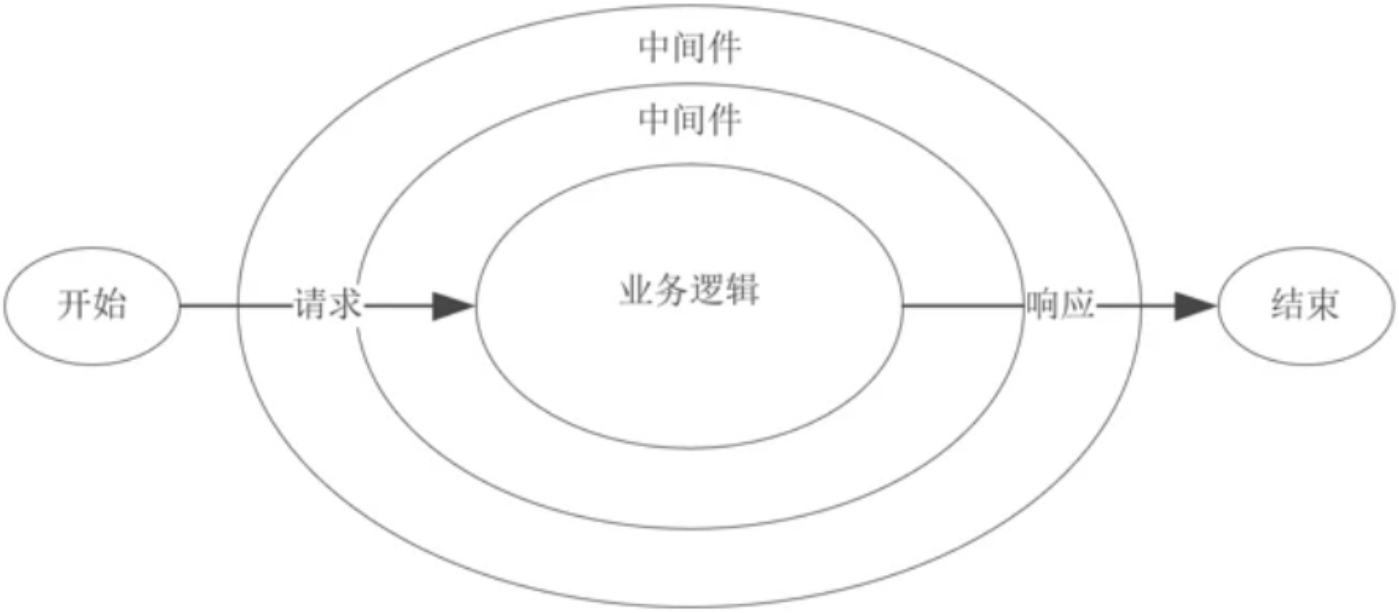



### 9.1. 是什么

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



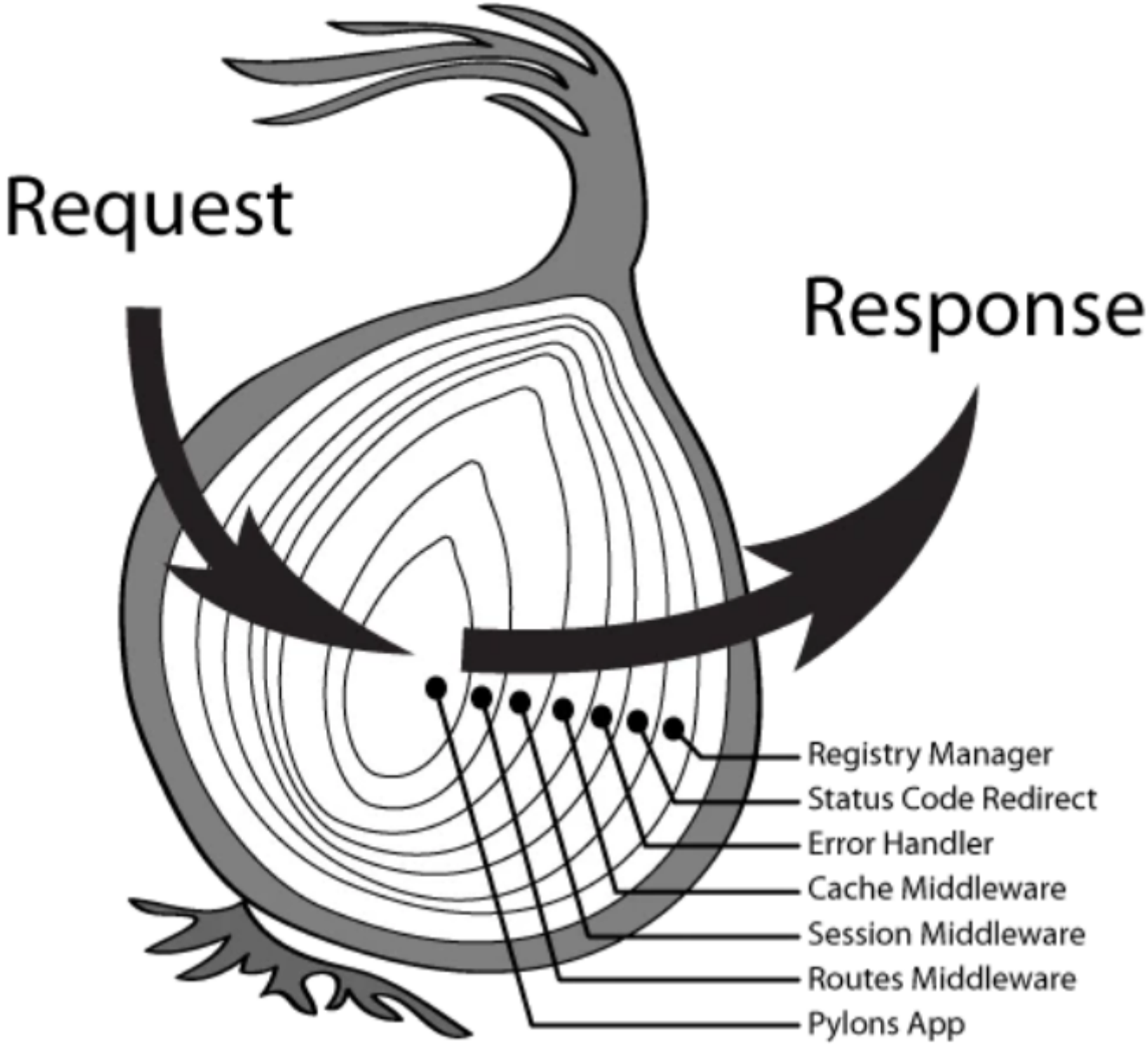

在这些中间件函数中，我们可以执行业务逻辑代码，修改请求和响应对象、返回响应数据等操作

## 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

```
// 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

```
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

```
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

```
// 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

```
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();
        }
    }
}
```

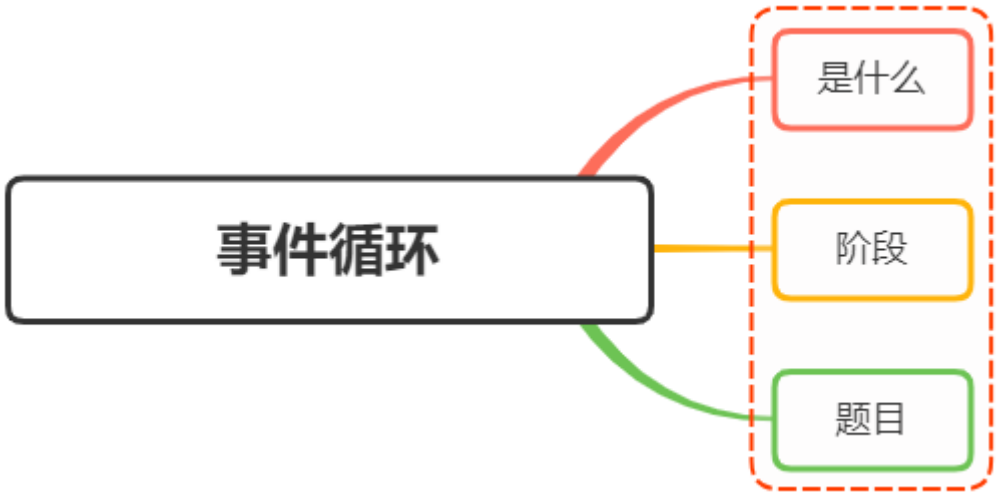
## 9.3. 总结

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

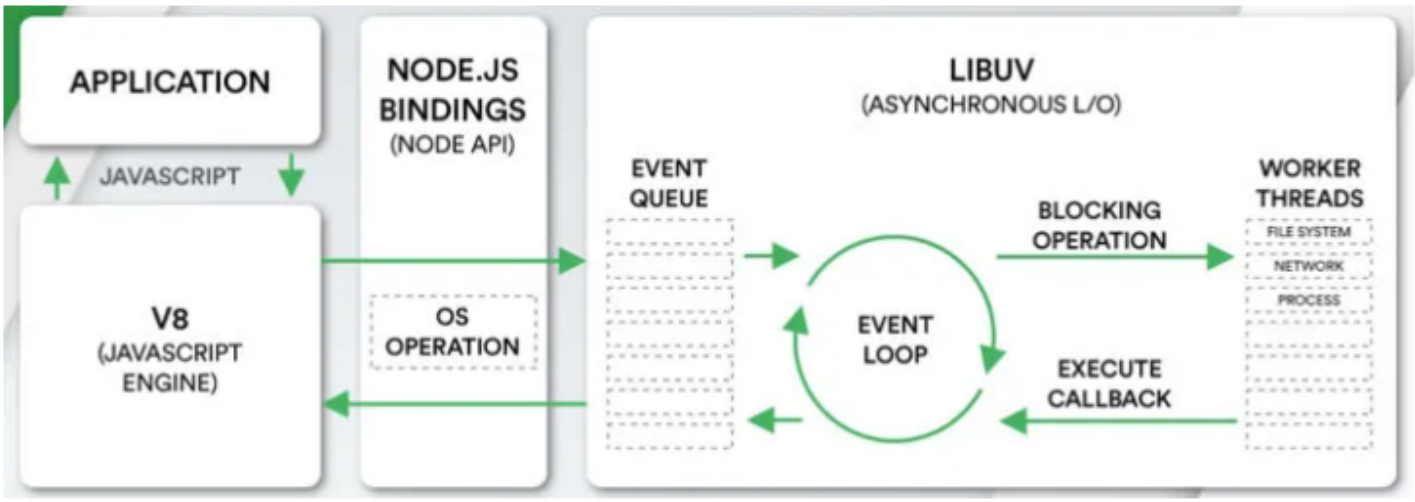
# 10. 说说对Nodejs中的事件循环机制理解?




## 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

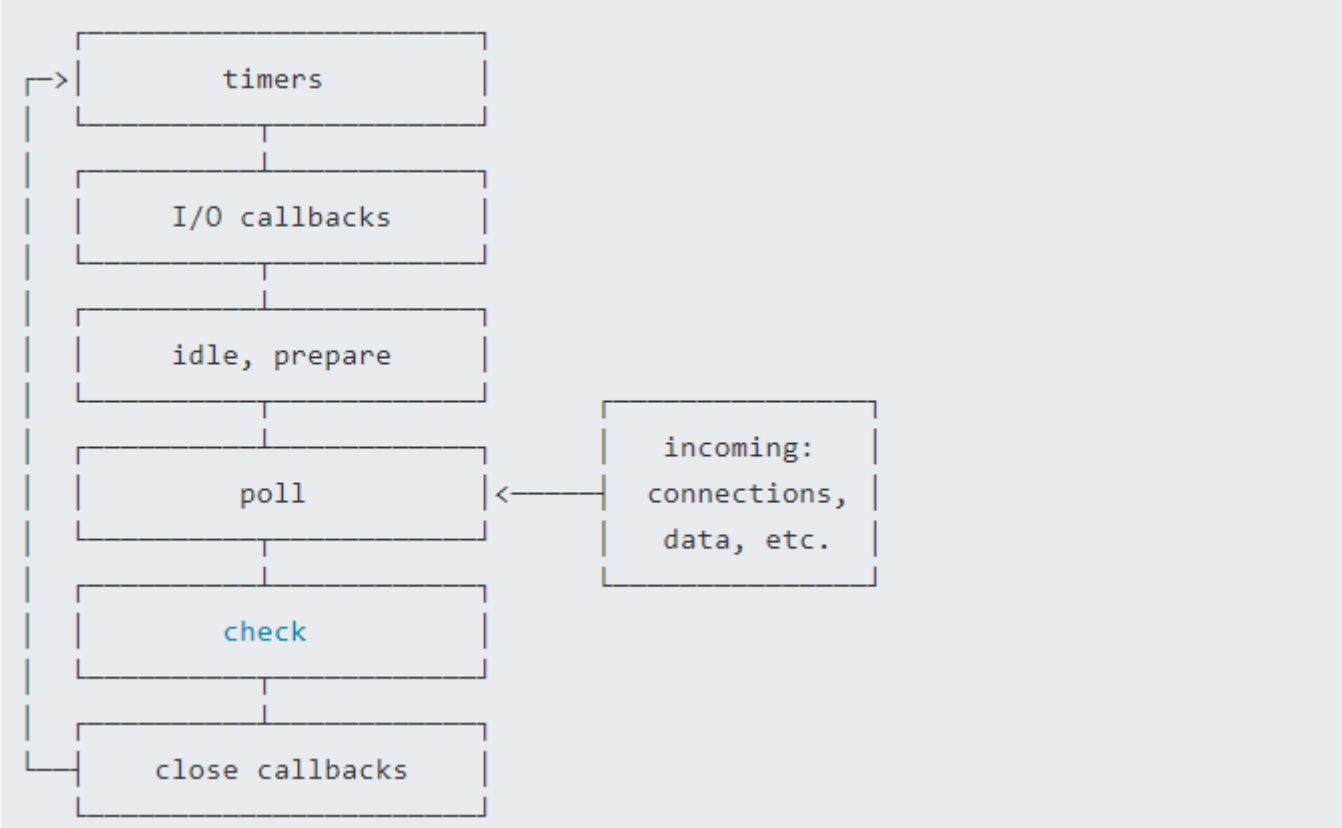
而在 `NodeJS` 中，事件循环是基于 `libuv` 实现， `libuv` 是一个多平台的专注于异步IO的库，如下图最右侧所示：




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：

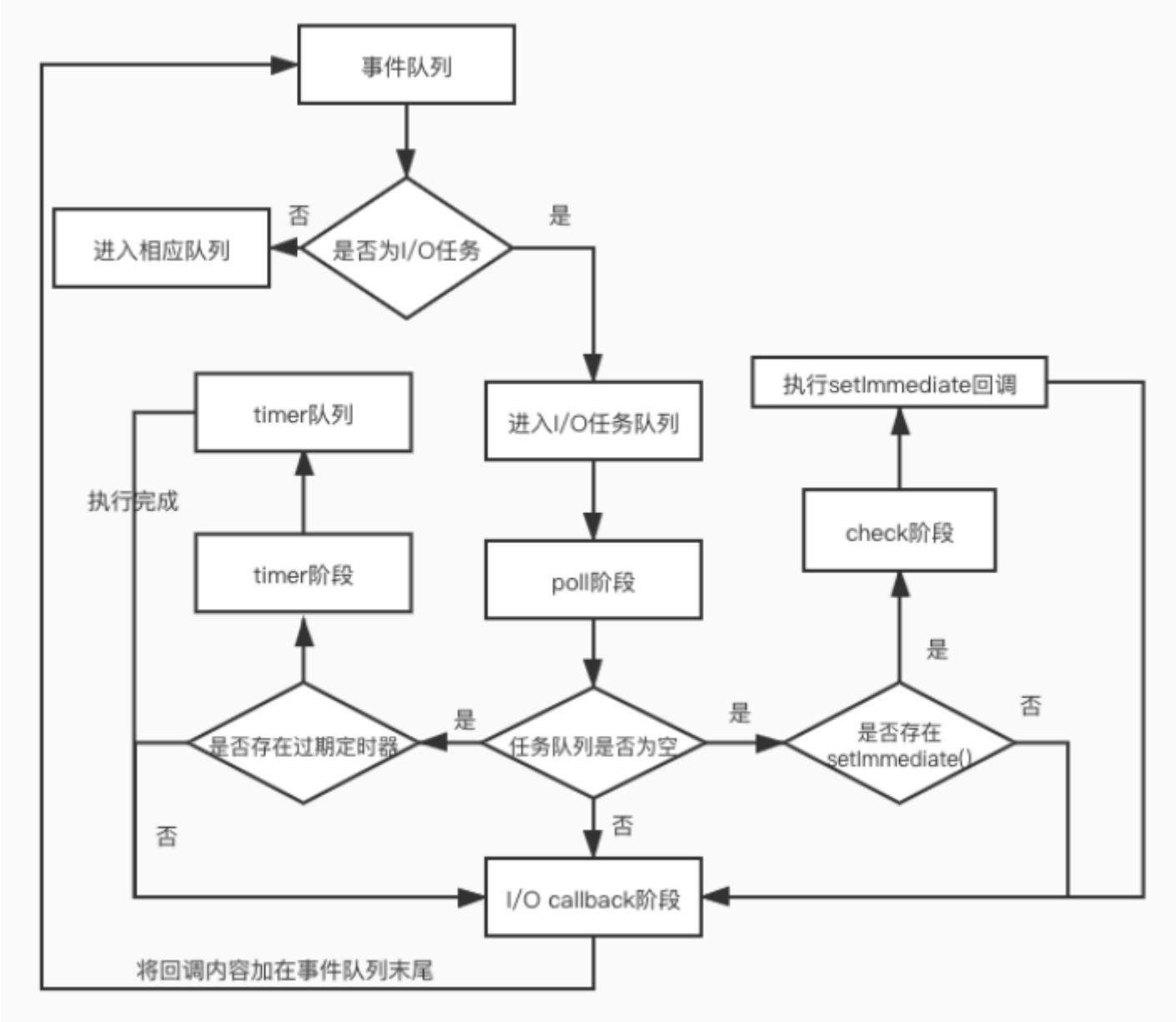



- timers阶段：这个阶段执行timer（setTimeout、setInterval）的回调
- 定时器检测阶段(timers)：本阶段执行 timer 的回调，即 setTimeout、setInterval 里面的回调函数
- I/O事件回调阶段(I/O callbacks)：执行延迟到下一个循环迭代的 I/O 回调，即上一轮循环中未被执行的一些I/O回调
- 闲置阶段(idle, prepare)：仅系统内部使用
- 轮询阶段(poll)：检索新的 I/O 事件;执行与 I/O 相关的回调（几乎所有情况下，除了关闭的回调函数，那些由计时器和 setImmediate() 调度的之外），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

```
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：

```
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

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

```
6 });
```

输出情况如下：

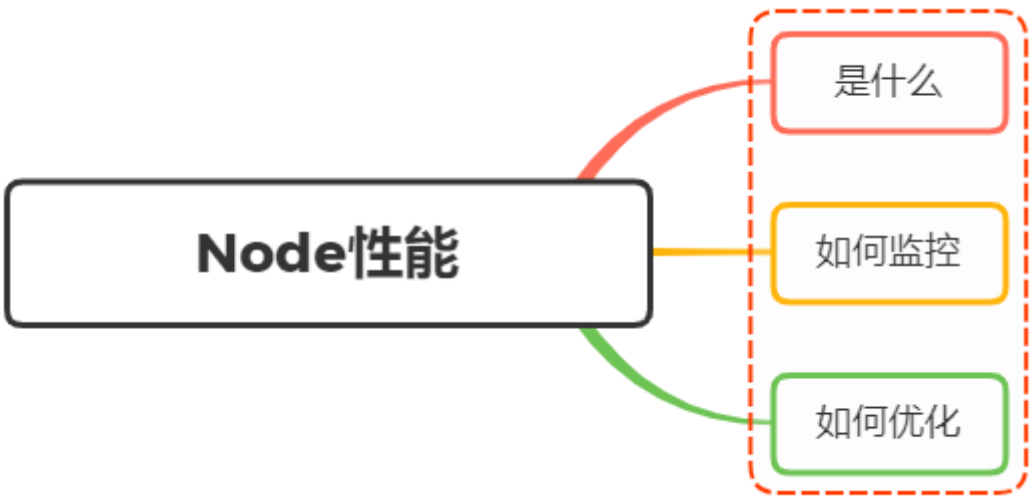
```
1 情况一：
2 setTimeout
3 setImmediate
4 情况二：
5 setImmediate
6 setTimeout
```

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

```
// /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

这里采用 `Easy-Monitor 2.0`，其是轻量级的 `Node.js` 项目内核性能监控 + 分析工具，在默认模式下，只需要在项目入口文件 `require` 一次，无需改动任何业务代码即可开启内核级别的性能监控分析

使用方法如下：

在你的项目入口文件中按照如下方式引入，当然请传入你的项目名称：

```
1 const easyMonitor = require('easy-monitor');
2 easyMonitor('你的项目名称');
```

打开你的浏览器，访问 `http://localhost:12333`，即可看到进程界面

关于定制化开发、通用配置项以及如何动态更新配置项详见官方文档

## 11.3. 如何优化

关于 `Node` 的性能优化的方式有：

- 使用最新版本Node.js
- 正确使用流 Stream
- 代码层面优化
- 内存管理优化

### 11.3.1. 使用最新版本Node.js

每个版本的性能提升主要来自于两个方面：

- V8 的版本更新
- Node.js 内部代码的更新优化

### 11.3.2. 正确使用流 Stream

在 `Node` 中，很多对象都实现了流，对于一个大文件可以通过流的形式发送，不需要将其完全读入内存

```
const http = require('http');
const fs = require('fs');
// bad
http.createServer(function (req, res) {
    fs.readFile(__dirname + '/data.txt', function (err, data) {
        res.end(data);
    });
});
// good
http.createServer(function (req, res) {
    const stream = fs.createReadStream(__dirname + '/data.txt');
    stream.pipe(res);
});
```

### 11.3.3. 代码层面优化

合并查询，将多次查询合并一次，减少数据库的查询次数

```
// bad
for user_id in userIds
     let account = user_account.findOne(user_id)
// good
const user_account_map = {}   // 注意这个对象将会消耗大量内存。
user_account.find(user_id in user_ids).forEach(account){
    user_account_map[account.user_id] =  account
}
for user_id in userIds
    var account = user_account_map[user_id]
```

### 11.3.4. 内存管理优化

在 V8 中，主要将内存分为新生代和老生代两代：

- 新生代：对象的存活时间较短。新生对象或只经过一次垃圾回收的对象
- 老生代：对象存活时间较长。经历过一次或多次垃圾回收的对象

若新生代内存空间不够，直接分配到老生代

通过减少内存占用，可以提高服务器的性能。如果有内存泄露，也会导致大量的对象存储到老生代中，服务器性能会大大降低

如下面情况：

```
const buffer = fs.readFileSync(__dirname + '/source/index.htm');
app.use(
    mount('/', async (ctx) => {
        ctx.status = 200;
        ctx.type = 'html';
        ctx.body = buffer;
        leak.push(fs.readFileSync(__dirname + '/source/index.htm'));
    })
);
const leak = [];
```

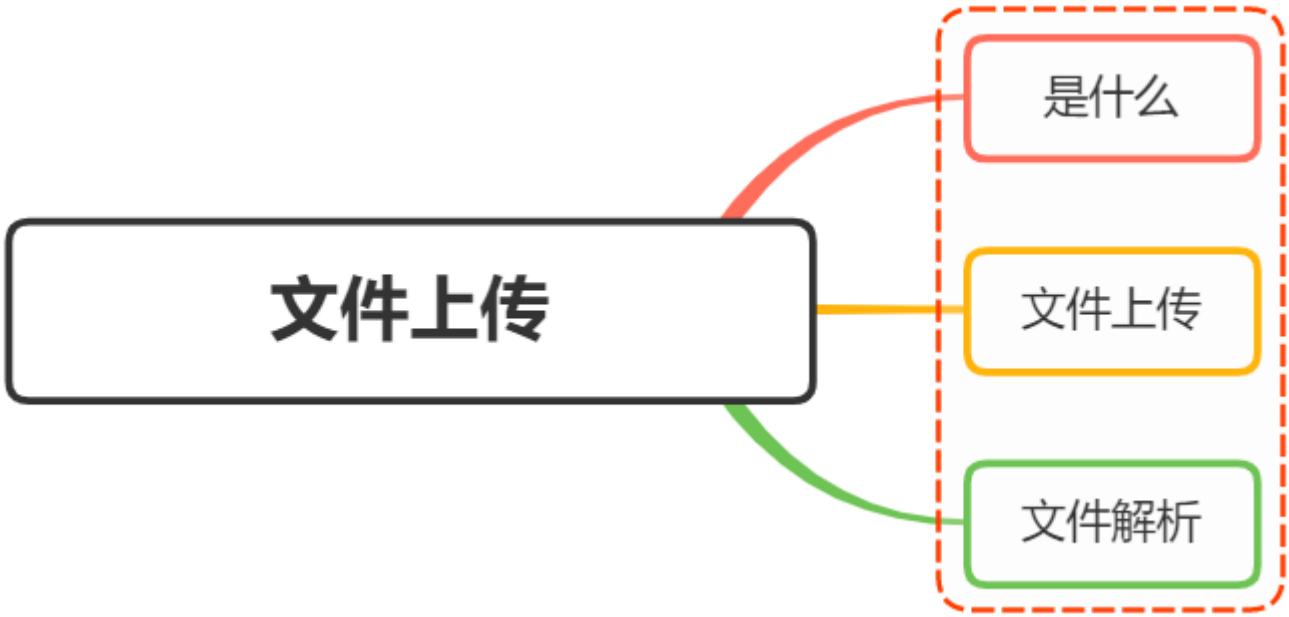
`leak` 的内存非常大，造成内存泄露，应当避免这样的操作，通过减少内存使用，是提高服务性能的手段之一

而节省内存最好的方式是使用池，其将频用、可复用对象存储起来，减少创建和销毁操作

例如有个图片请求接口，每次请求，都需要用到类。若每次都需要重新new这些类，并不是很合适，在大量请求时，频繁创建和销毁这些类，造成内存抖动

使用对象池的机制，对这种频繁需要创建和销毁的对象保存在一个对象池中。每次用到该对象时，就取对象池空闲的对象，并对它进行初始化操作，从而提高框架的性能

## 12. 如何实现文件上传？说说你的思路




## 12.1. 是什么

文件上传在日常开发中应用很广泛，我们发微博、发微信朋友圈都会用到了图片上传功能

因为浏览器限制，浏览器不能直接操作文件系统的，需要通过浏览器所暴露出来的统一接口，由用户主动授权发起来访问文件动作，然后读取文件内容进指定内存里，最后执行提交请求操作，将内存里的文件内容数据上传到服务端，服务端解析前端传来的数据信息后存入文件里

对于文件上传，我们需要设置请求头为 `content-type:multipart/form-data`

multipart互联网上的混合资源，就是资源由多种元素组成，form-data表示可以使用HTML Forms 和 POST 方法上传文件

结构如下：

```
POST /t2/upload.do HTTP/1.1
User-Agent: SOHUWapRebot
Accept-Language: zh-cn,zh;q=0.5
Accept-Charset: GBK,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Content-Length: 60408
Content-Type:multipart/form-data; boundary=ZnGpDtePMx0KrHh_G0X99Yef9r8JZsRJSXC
Host: w.sohu.com
--ZnGpDtePMx0KrHh_G0X99Yef9r8JZsRJSXC
Content-Disposition: form-data; name="city"
Santa colo
--ZnGpDtePMx0KrHh_G0X99Yef9r8JZsRJSXC
Content-Disposition: form-data;name="desc"
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 8bit
...
--ZnGpDtePMx0KrHh_G0X99Yef9r8JZsRJSXC
```

```
Content-Disposition: form-data;name="pic"; filename="photo.jpg"
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary
... binary data of the jpg ...
--ZnGpDtePMx0KrHh_G0X99Yef9r8JZsRJSXC--
```

`boundary` 表示分隔符，如果要上传多个表单项，就要使用 `boundary` 分割，每个表单项由 `---XXX` 开始，以 `---XXX` 结尾

而 `xxx` 是即时生成的字符串，用以确保整个分隔符不会在文件或表单项的内容中出现

每个表单项必须包含一个 `Content-Disposition` 头，其他的头信息则为可选项， 比如 `Content-Type`

`Content-Disposition` 包含了 `type` 和 一个名字为 `name` 的 `parameter` ， `type` 是 `form-data` ， `name` 参数的值则为表单控件（也即 field）的名字，如果是文件，那么还有一个 `filename` 参数，值就是文件名

```
Content-Disposition: form-data; name="user"; filename="logo.png"
```

至于使用 `multipart/form-data` ，是因为文件是以二进制的形式存在，其作用是专门用于传输大型二进制数据，效率高

### 12.1.1. 如何实现

关于文件的上传的上传，我们可以分成两步骤：

- 文件的上传
- 文件的解析

### 12.1.2. 文件上传

传统前端文件上传的表单结构如下：

```
<form action="http://localhost:8080/api/upload" method="post"
enctype="multipart/form-data">
    <input type="file" name="file" id="file" value="" multiple="multiple" />
    <input type="submit" value="提交"/>
</form>
```

`action` 就是我们的提交到的接口， `enctype="multipart/form-data"` 就是指定上传文件格式， `input` 的 `name` 属性一定要等于 `file`

### 12.1.3. 文件解析

在服务器中，这里采用 `koa2` 中间件的形式解析上传的文件数据，分别有下面两种形式：

- koa-body
- koa-multer

#### 12.1.3.1. koa-body

安装依赖

```
npm install koa-body
```

引入 `koa-body` 中间件

```
const koaBody = require('koa-body');
app.use(koaBody({
    multipart: true,
    formidable: {
        maxFileSize: 200
*1024*
1024    // 设置上传文件大小最大限制，默认2M
    }
}));
```

获取上传的文件

```
const file = ctx.request.files.file; // 获取上传文件
```

获取文件数据后，可以通过 `fs` 模块将文件保存到指定目录

```
router.post('/uploadfile', async (ctx, next) => {
  // 上传单个文件
  const file = ctx.request.files.file; // 获取上传文件
  // 创建可读流
  const reader = fs.createReadStream(file.path);
  let filePath = path.join(__dirname, 'public/upload/') +
/${file.name}
;
  // 创建可写流
```

```
  const upStream = fs.createWriteStream(filePath);
  // 可读流通过管道写入可写流
  reader.pipe(upStream);
  return ctx.body = "上传成功! ";
});
```

### 12.1.3.2. koa-multer

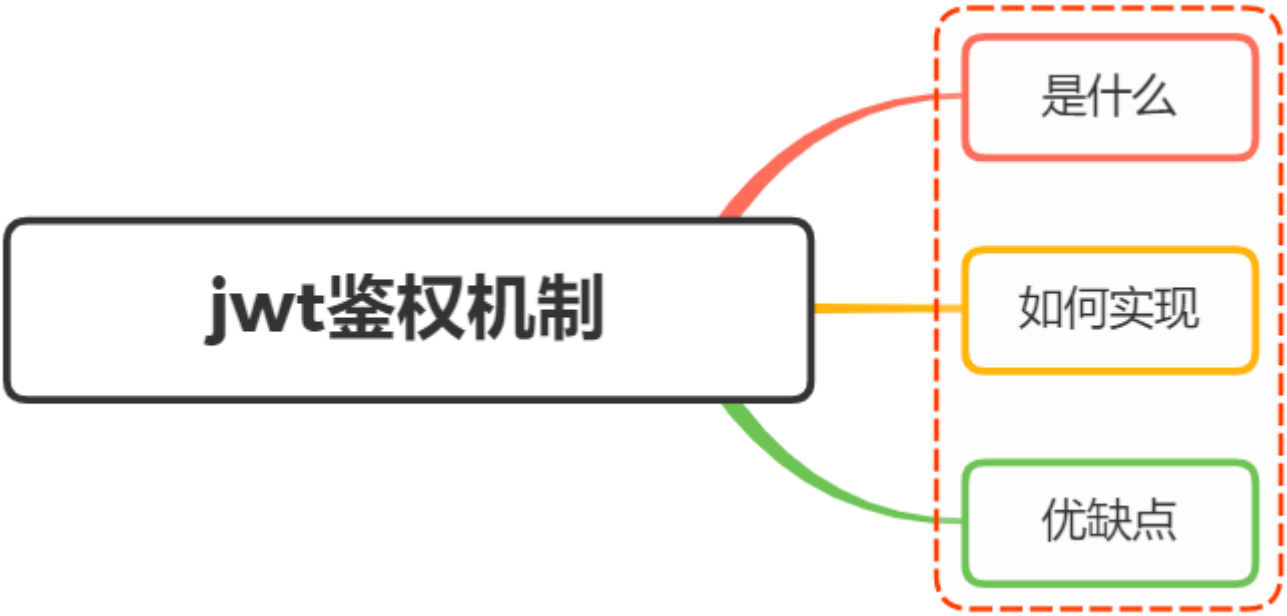
安装依赖：

```
npm install koa-multer
```

使用 `multer` 中间件实现文件上传

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, "./upload/")
  },
  filename: (req, file, cb) => {
    cb(null, Date.now() + path.extname(file.originalname))
  }
})
const upload = multer({
  storage
});
const fileRouter = new Router();
fileRouter.post("/upload", upload.single('file'), (ctx, next) => {
  console.log(ctx.req.file); // 获取文件
})
app.use(fileRouter.routes());
```

# 13. 如何实现jwt鉴权机制？说说你的思路




## 13.1. 是什么

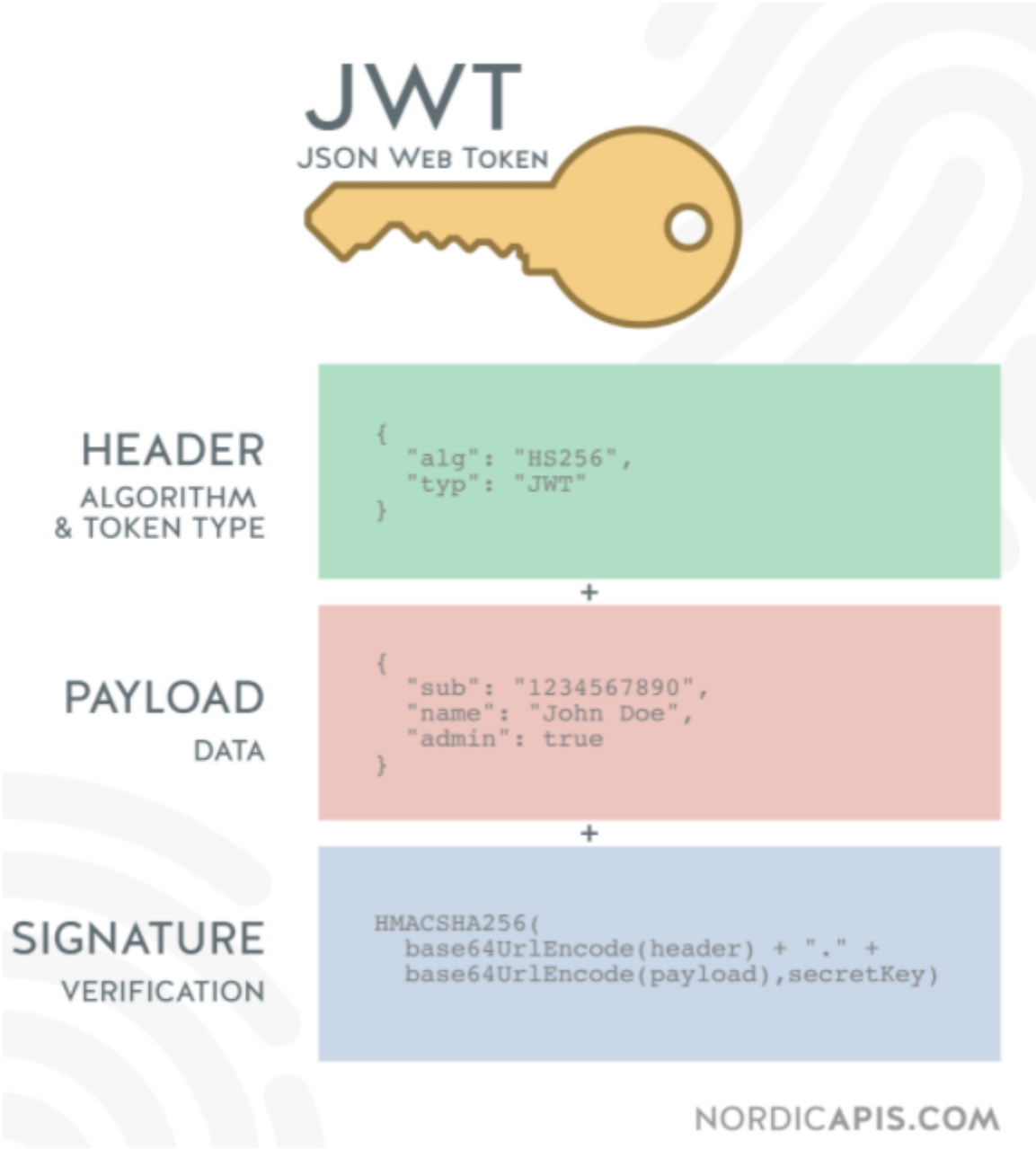
JWT（JSON Web Token），本质就是一个字符串书写规范，如下图，作用是用来在用户和服务器之间传递安全可靠的信息

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

在目前前后端分离的开发过程中，使用 `token` 鉴权机制用于身份验证是最常见的方案，流程如下：

- 服务器当验证用户账号和密码正确的时候，给用户颁发一个令牌，这个令牌作为后续用户访问一些接口的凭证
- 后续访问会根据这个令牌判断用户时候有权限进行访问

`Token`，分成了三部分，头部（Header）、载荷（Payload）、签名（Signature），并以 `.` 进行拼接。其中头部和载荷都是以 `JSON` 格式存放数据，只是进行了编码




### 13.1.1. header

每个JWT都会带有头部信息，这里主要声明使用的算法。声明算法的字段名为 `alg` ，同时还有一个 `typ` 的字段，默认 `JWT` 即可。以下示例中算法为HS256

```
1 {  "alg": "HS256",  "typ": "JWT" }
```

因为JWT是字符串，所以我们还需要对以上内容进行Base64编码，编码后字符串如下：

```
1 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

### 13.1.2. payload

载荷即消息体，这里会存放实际的内容，也就是 `Token` 的数据声明，例如用户的 `id` 和 `name` ，默认情况下也会携带令牌的签发时间 `iat` ，通过还可以设置过期时间，如下：

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

同样进行Base64编码后，字符串如下：

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
```

### 13.1.3. Signature

签名是对头部和载荷内容进行签名，一般情况，设置一个 `secretKey` ，对前两个的结果进行 `HMACSHA25` 算法，公式如下：

```
Signature = HMACSHA256(base64Url(header)+.+base64Url(payload),secretKey)
```

一旦前面两部分数据被篡改，只要服务器加密用的密钥没有泄露，得到的签名肯定和之前的签名不一致

## 13.2. 如何实现

`Token` 的使用分成了两部分：

- 生成token：登录成功的时候，颁发token
- 验证token：访问某些资源或者接口时，验证token

### 13.2.1. 生成 token

借助第三方库 `jsonwebtoken` ，通过 `jsonwebtoken` 的 `sign` 方法生成一个 `token` ：

- 第一个参数指的是 Payload
- 第二个是秘钥，服务端特有
- 第三个参数是 option，可以定义 token 过期时间

```
const crypto = require("crypto"),
```

```
  jwt = require("jsonwebtoken");
// TODO:使用数据库
// 这里应该是用数据库存储，这里只是演示用
let userList = [];
class UserController {
  // 用户登录
  static async login(ctx) {
    const data = ctx.request.body;
    if (!data.name || !data.password) {
      return ctx.body = {
        code: "000002",
        message: "参数不合法"
      }
    }
    const result = userList.find(item => item.name === data.name &&
item.password === crypto.createHash('md5').update(data.password).digest('hex'))
    if (result) {
      // 生成token
      const token = jwt.sign(

        {
          name: result.name
        },
        "test_token", // secret
        { expiresIn: 60 * 60 } // 过期时间：60 * 60 s
      );
      return ctx.body = {
        code: "0",
        message: "登录成功",
        data: {
          token
        }
      };
    } else {
      return ctx.body = {
        code: "000002",
        message: "用户名或密码错误"
      };
    }
  }
}
module.exports = UserController;
```

在前端接收到 `token` 后，一般情况会通过 `localStorage` 进行缓存，然后将 `token` 放到 `HTTP` 请求头 `Authorization` 中，关于 `Authorization` 的设置，前面要加上 Bearer ，注意后面带有空格

```
axios.interceptors.request.use(config => {
  const token = localStorage.getItem('token');
  config.headers.common['Authorization'] = 'Bearer ' + token; // 留意这里的 Authorization
  return config;
})
```

### 13.2.2. 校验token

使用 `koa-jwt` 中间件进行验证，方式比较简单

```
/ 注意：放在路由前面
app.use(koajwt({
  secret: 'test_token'
}).unless({ // 配置白名单
  path: [/\/api\/register/, /\/api\/login/]
}))
```

- secret 必须和 sign 时候保持一致
- 可以通过 unless 配置接口白名单，也就是哪些 URL 可以不用经过校验，像登陆/注册都可以不用校验
- 校验的中间件需要放在需要校验的路由前面，无法对前面的 URL 进行校验

获取 `token` 用户的信息方法如下：

```
router.get('/api/userInfo',async (ctx,next) =>{
    const authorization =  ctx.header.authorization // 获取jwt
    const token = authorization.replace('Beraer ','')
    const result = jwt.verify(token,'test_token')
    ctx.body = result
```

注意：上述的 `HMA256` 加密算法为单秘钥的形式，一旦泄露后果非常的危险

在分布式系统中，每个子系统都要获取到秘钥，那么这个子系统根据该秘钥可以发布和验证令牌，但有些服务器只需要验证令牌

这时候可以采用非对称加密，利用私钥发布令牌，公钥验证令牌，加密算法可以选择 `RS256`

## 13.3. 优缺点

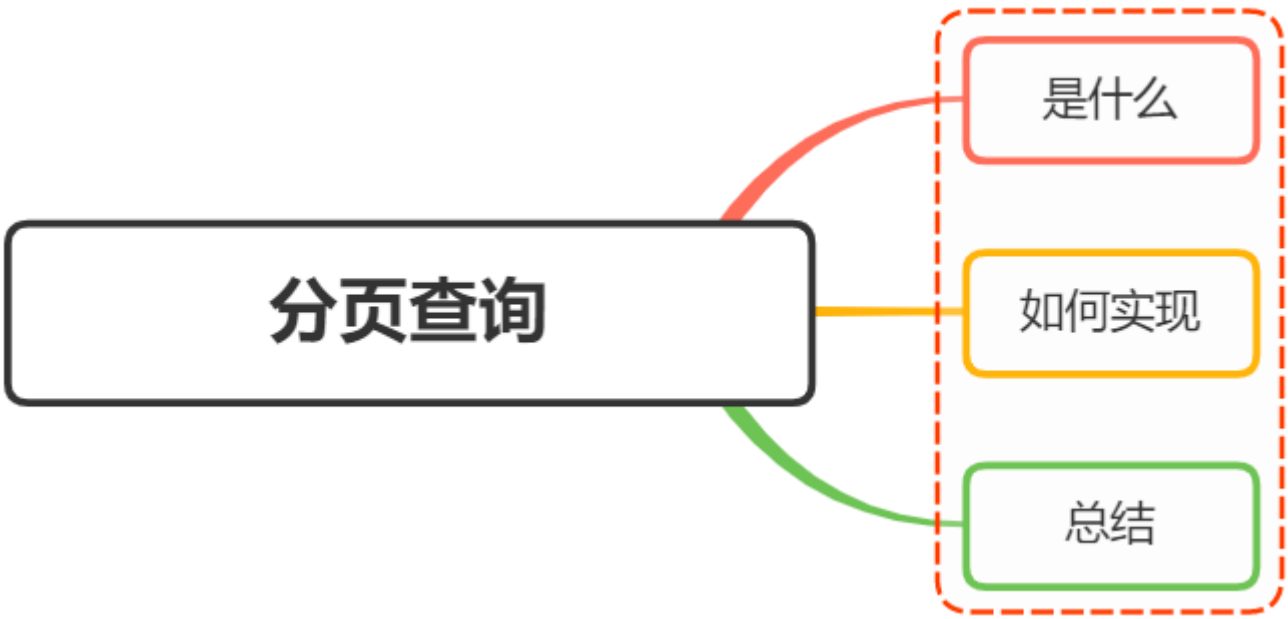
优点：

- json具有通用性，所以可以跨语言
- 组成简单，字节占用小，便于传输
- 服务端无需保存会话信息，很容易进行水平扩展
- 一处生成，多处使用，可以在分布式系统中，解决单点登录问题
- 可防护CSRF攻击

缺点：

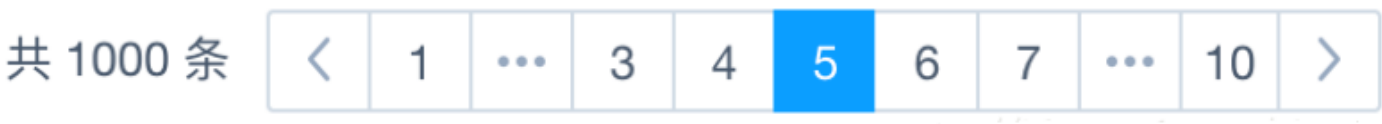
- payload部分仅仅是进行简单编码，所以只能用于存储逻辑必需的非敏感信息
- 需要保护好加密密钥，一旦泄露后果不堪设想
- 为避免token被劫持，最好使用https协议

## 14. 如果让你来设计一个分页功能, 你会怎么设计? 前后端如何交互?




### 14.1. 是什么

在我们做数据查询的时候，如果数据量很大，比如几万条数据，放在一个页面显示的话显然不友好，这时候就需要采用分页显示的形式，如每次只显示10条数据




要实现分页功能，实际上就是从结果集中显示第1(10条记录作为第1页，显示第11)20条记录作为第2页，以此类推

因此，分页实际上就是从结果集中截取出第M~N条记录

## 14.2. 如何实现

前端实现分页功能，需要后端返回必要的数据，如总的页数，总的数据量，当前页，当前的数据

```
{
 "totalCount": 1836,   // 总的条数
 "totalPages": 92,  // 总页数
 "currentPage": 1   // 当前页数
 "data": [     // 当前页的数据
   {
 ...
   }
]
```

后端采用 `mysql` 作为数据的持久性存储

前端向后端发送目标的页码 `page` 以及每页显示数据的数量 `pageSize` ，默认情况每次取10条数据，则每一条数据的起始位置 `start` 为：

```
const start = (page - 1) * pageSize
```

当确定了 `limit` 和 `start` 的值后，就能够确定 `SQL` 语句：

```
const sql = SELECT * FROM record limit ${pageSize} OFFSET ${start};
```

上诉 `SQL` 语句表达的意思为：截取从 `start` 到 `start` + `pageSize` 之间（左闭右开）的数据

关于查询数据总数的 `SQL` 语句为， `record` 为表名：

```
SELECT COUNT(*) FROM record
```

因此后端的处理逻辑为：

- 获取用户参数页码数page和每页显示的数目 pageSize ，其中page 是必须传递的参数，pageSize 为可选参数，默认为10
- 编写 SQL 语句，利用 limit 和 OFFSET 关键字进行分页查询

- 查询数据库，返回总数据量、总页数、当前页、当前页数据给前端

代码如下所示：

```
router.all('/api', function (req, res, next) {
  var param = '';
  // 获取参数
  if (req.method == "POST") {
    param = req.body;
  } else {
    param = req.query || req.params;
  }
  if (param.page == '' || param.page == null || param.page == undefined) {
    res.end(JSON.stringify({ msg: '请传入参数page', status: '102' }));
    return;
  }
  const pageSize = param.pageSize || 10;
  const start = (param.page - 1) * pageSize;
  const sql =
SELECT * FROM record limit ${pageSize} OFFSET ${start};
  pool.getConnection(function (err, connection) {
    if (err) throw err;
    connection.query(sql, function (err, results) {
      connection.release();
      if (err) {
        throw err
      } else {
        // 计算总页数
        var allCount = results[0][0]['COUNT(*)'];
        var allPage = parseInt(allCount) / 20;
        var pageStr = allPage.toString();
        // 不能被整除
        if (pageStr.indexOf('.') > 0) {
          allPage = parseInt(pageStr.split('.')[0]) + 1;
        }
        var list = results[1];
        res.end(JSON.stringify({ msg: '操作成功', status: '200', totalPages:
allPage, currentPage: param.page, totalCount: allCount, data: list }));
      }
    })
  })
});
```

## 14.3. 总结

通过上面的分析，可以看到分页查询的关键在于，要首先确定每页显示的数量 `pageSize` ，然后根据当前页的索引 `pageIndex` （从1开始），确定 `LIMIT` 和 `OFFSET` 应该设定的值：

- LIMIT 总是设定为 pageSize
- OFFSET 计算公式为 pageSize * (pageIndex - 1)

确定了这两个值，就能查询出第 `N` 页的数据