目录

[Natives modules 1](#_Toc15945)

[Builtin modules 胶水层 1](#_Toc6843)

[底层实现 2](#_Toc8968)

[Node的优缺点 2](#_Toc1960)

[异步IO 3](#_Toc7649)

[总结： 3](#_Toc6362)

[Libuv库 3](#_Toc4406)

[事件驱动 4](#_Toc17850)

[单线程如何实现高并发 4](#_Toc786)

[NodeJs应用场景 5](#_Toc24396)

[Node全局对象 5](#_Toc24327)

[常见全局变量 5](#_Toc28430)

[process 5](#_Toc27652)

[API 7](#_Toc7663)

[Path模块 7](#_Toc15859)

[Buffer 7](#_Toc27656)

[创建Buffer实例 8](#_Toc26903)

[FS模块 8](#_Toc1777)

[API 9](#_Toc11648)

[链表 12](#_Toc15338)

[数组的缺点： 12](#_Toc17177)

[链表分类 12](#_Toc11318)

[网络通信 12](#_Toc29459)

[通信原理 12](#_Toc23890)

[通信必要条件 12](#_Toc20924)

[通讯方式 13](#_Toc19449)

[网络层次模型 14](#_Toc17414)

# Natives modules

提供应用程序可直接调用的库如fs，path等

# Builtin modules 胶水层

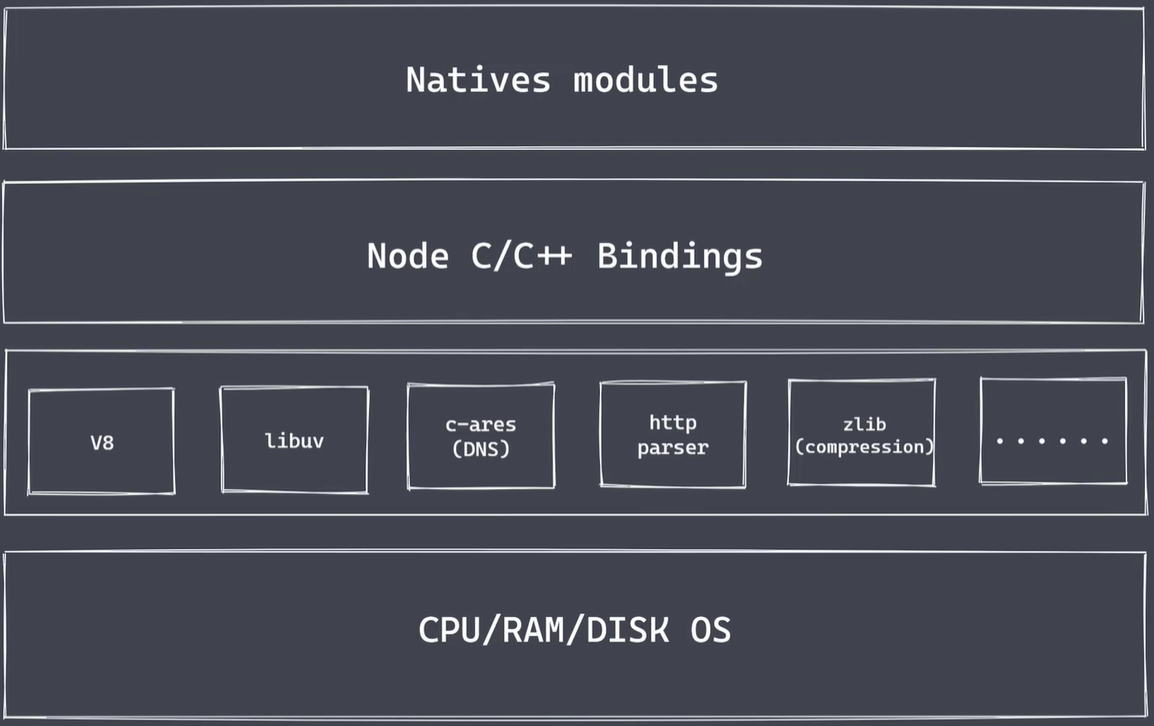
当处理硬件层的时候需要用这个模块

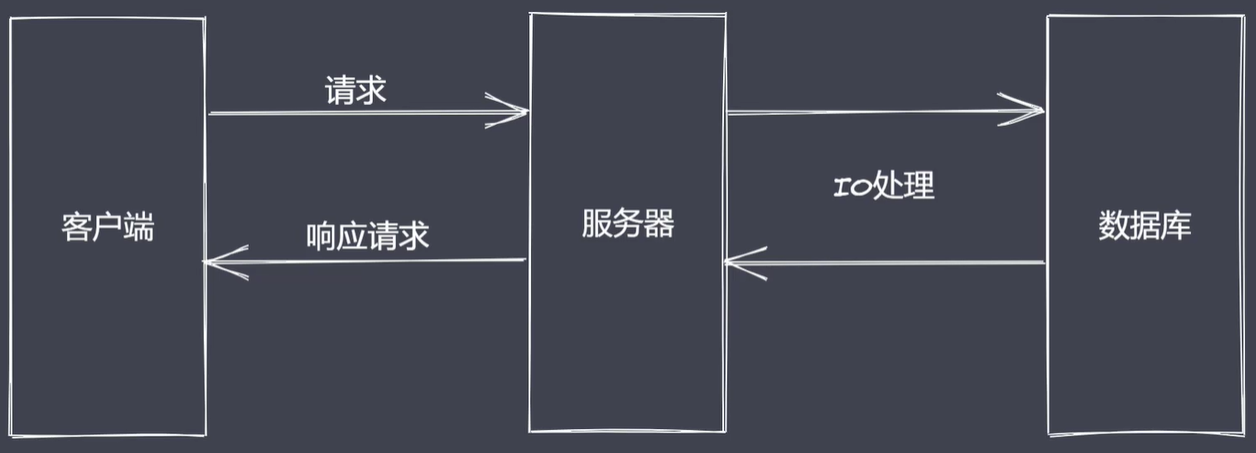
# 底层实现

V8：执行js代码，提供桥梁接口，实现功能的是C或者C++，V8引擎的作用是当执行js代码之后，转换成C++,C的功能

Libuv:事件循环，事件队列，异步IO

第三方模块





# Node的优缺点

普通的处理并发：用多线程，

NODE处理并发：Reactor模式：应答者模式，单线程完成多线程工作，实现异步IO和事件驱动

通常用NODE处理IO密集型高并发请求

# 异步IO

阻塞IO和非阻塞IO

当调用非阻塞IO时，应用层会不停地轮询

所以我们期望实现无需主动判断的非阻塞IO

轮询：应用层会重复的调用IO操作，来判断IO是否结束，

一次IO（请求）,客户端会不停地询问是否结束，结束后会将IO的返回结果返回给客户端，我们期望实现无需主动判断的非阻塞IO，就是发起IO后，直接开始下一个IO，而无需不停询问

## 总结：

IO是应用程序的瓶颈所在（一次请求）

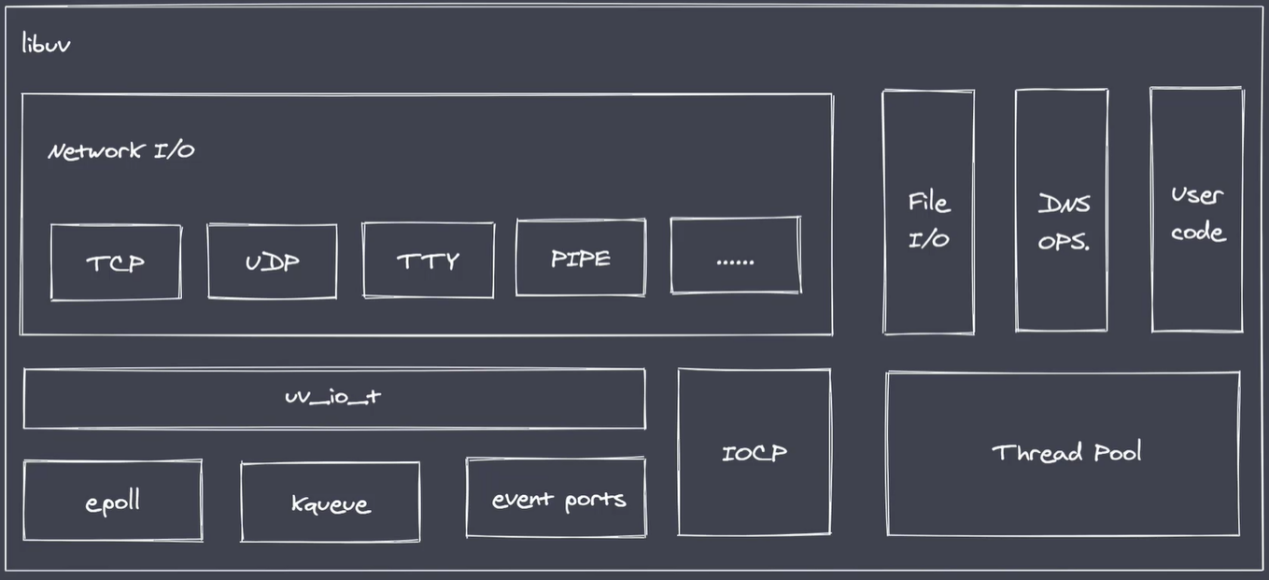
异步IO提高性能无需原地等待返回结果，可以直接开启下一个IO

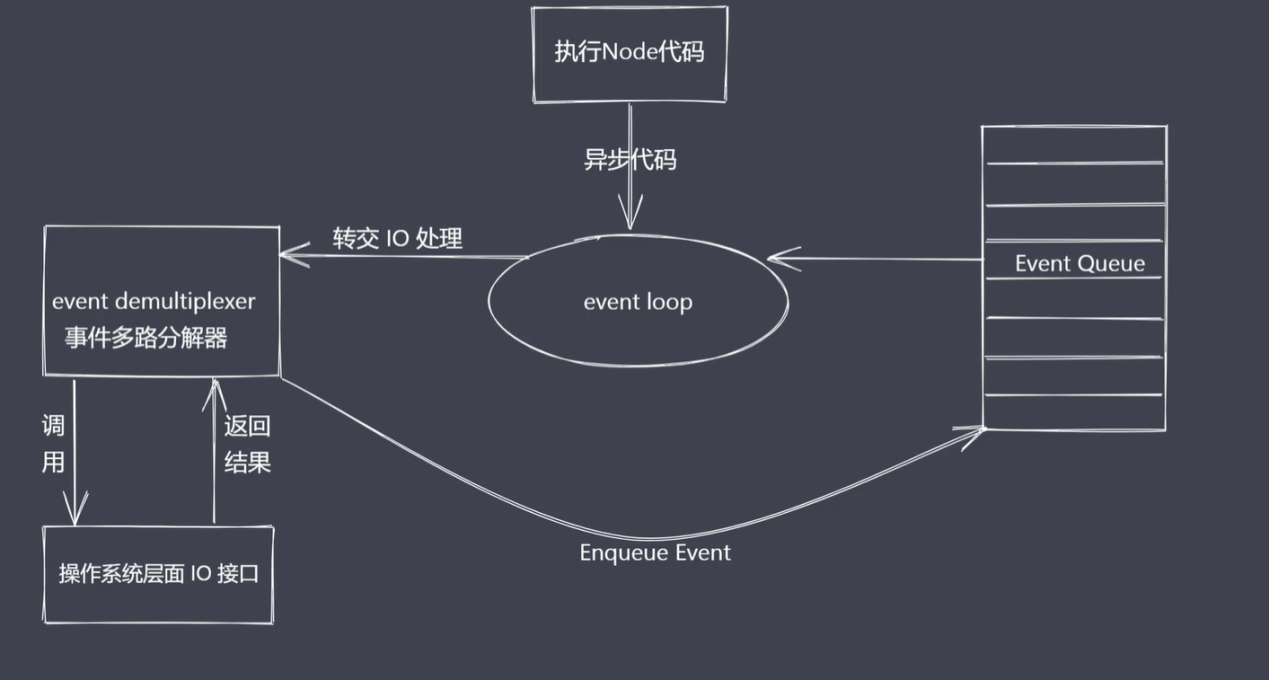
IO操作属于操作系统级别，平台都有对应的实现

Node.js的单线程通过事件驱动，以及libuv库实现非阻塞的异步IO

# Libuv库

是几种不同的异步IO实现方式的抽象封装层，他本身是默认有4个线程，js的单线程指的是主线程是单线程





# 事件驱动

是软件开发的一种通用的模式

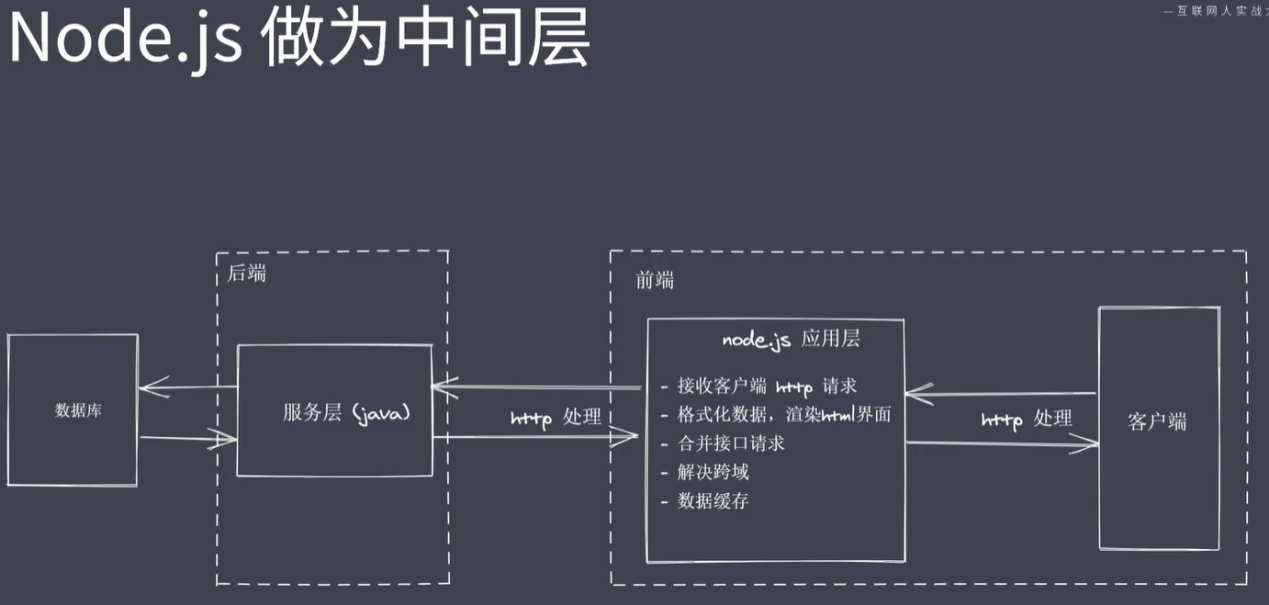
常见的三种模式分别是事件驱动，发布订阅，观察者模式

他们共同的特征是：主体发布消息，其他实例接收消息。

# 单线程如何实现高并发

通过异步IO，事件循环，事件驱动架构，通过回调通知的方式实现非阻塞的调用和实现并发

# NodeJs应用场景



操作数据库提供API服务

实时聊天应用程序

更加适合IO密集型的任务

# Node全局对象

Global对象

Global的根本作用是作为宿主

## 常见全局变量

\_\_filename: 正在执行脚本文件的绝对路径

\_\_dirname: 正在执行脚本所在的目录

timer类函数

process:提供与当前进程互动的接口

require

module,exports

### process

获取进程信息

执行进程操作

#### 资源信息

process.memoryUsage() // 获取资源信息内存

process.cpuUsage() // 获取资源信息 CPU

#### 运行环境

process.cud() // 获取运行目录

process.version process.versions // 版本

process.arch // cpu架构

process.env.NODE\_ENV // 用户环境

process.env.PATH // 系统环境变量路径

process.env.USERPROFILE // 管理员目录 （或者HOME）

process.platform // 平台信息

#### 运行状态

启动参数，PID,运行时间

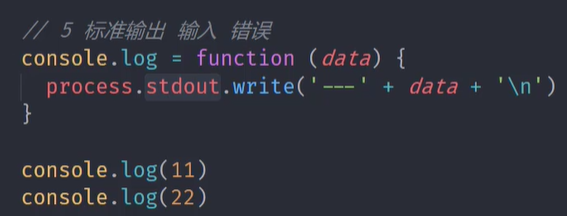
process.argv // 返回数组，NODE目录和当前脚本路径

process.pid

process.uptime() // 执行脚本消耗的时间

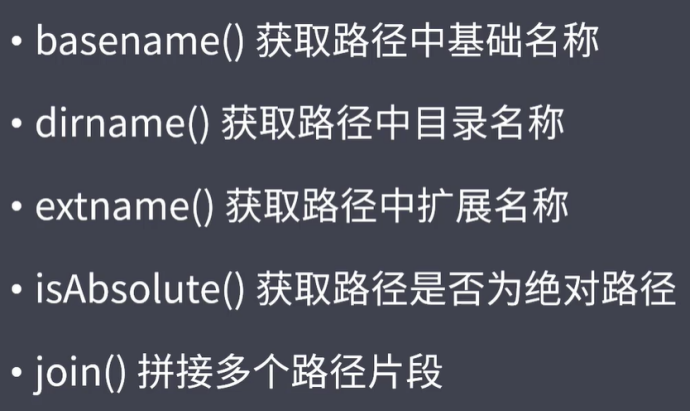


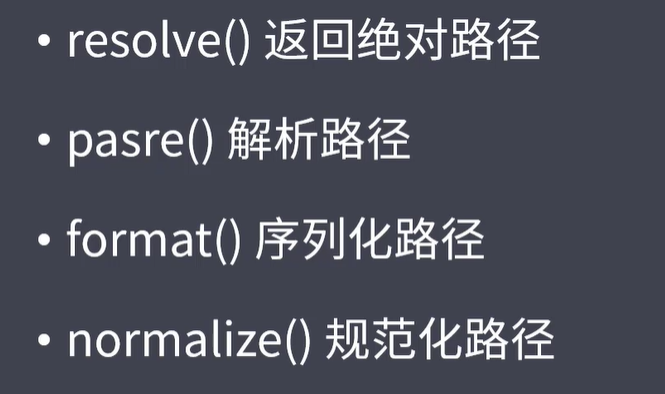
process.exit()



# API

## Path模块





basename()返回接收路径中的最后一个部分，可能是文件名称，可能是路径。

第二个参数表示扩展名，如果没有设置则返回完整的文件名称带后缀。

第二个参数作为后缀时，如果没有在当前路径中被匹配到则被忽略。

extname() 传入文件路径，返回.js / .css /

parse() 接收一个路径，返回一个对象，root dir base ext name 注意传入绝对路径和相对路径的区别

format() 是parse的逆操作

## Buffer

buffer全局变量（缓冲区）可以让js直接操作二进制

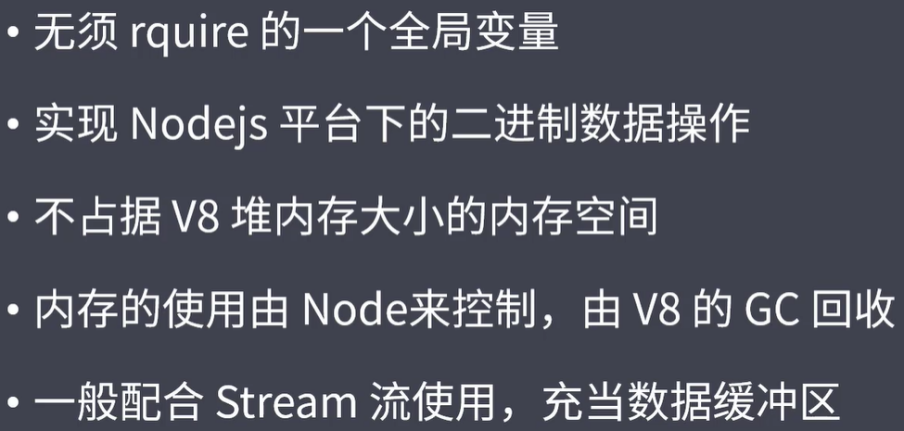
IO行为操作的就是二进制数据

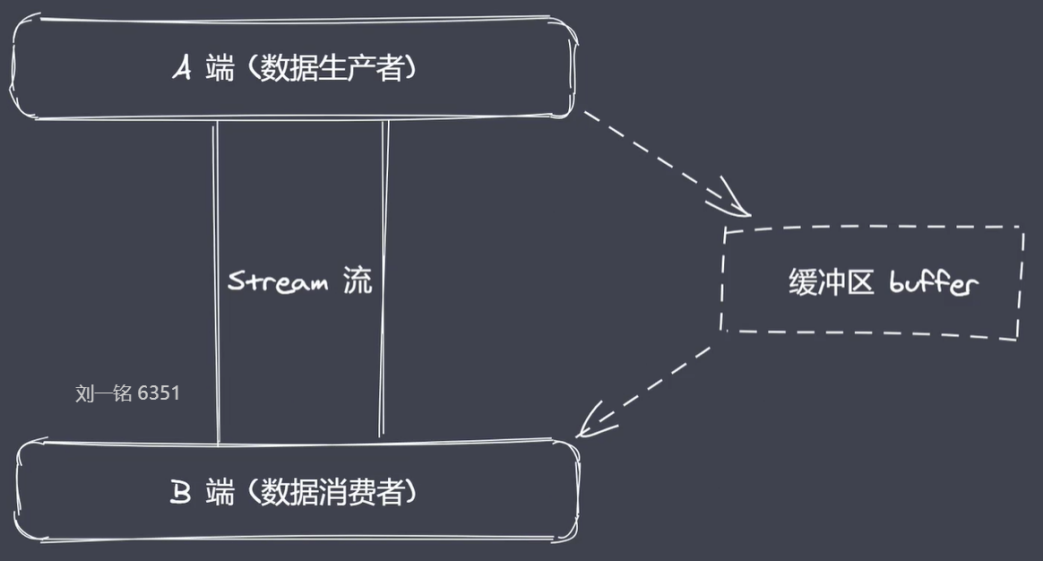
Stream流操作 用来处理数据的

可以把数据分段，避免出现问题

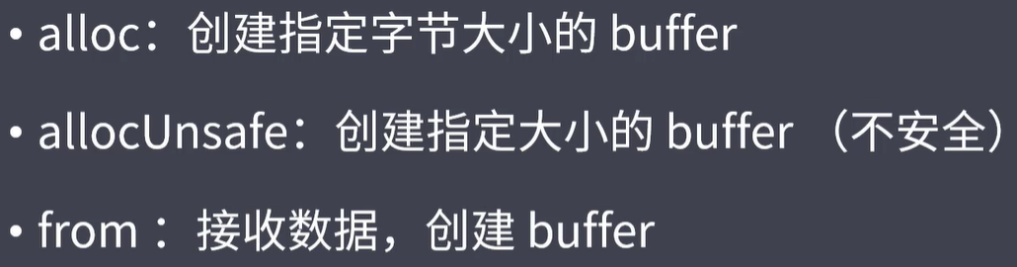
流操作配合管道实现数据分段传输

他是一片内存空间





### 创建Buffer实例



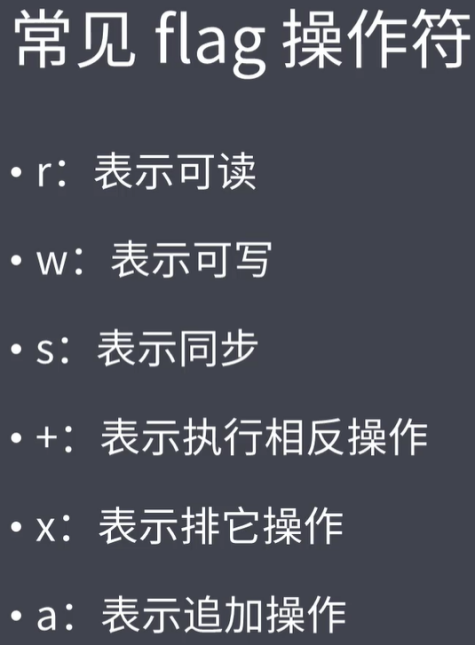
## FS模块

权限位：用户对文件所具备的操作权限 rwx



标志位

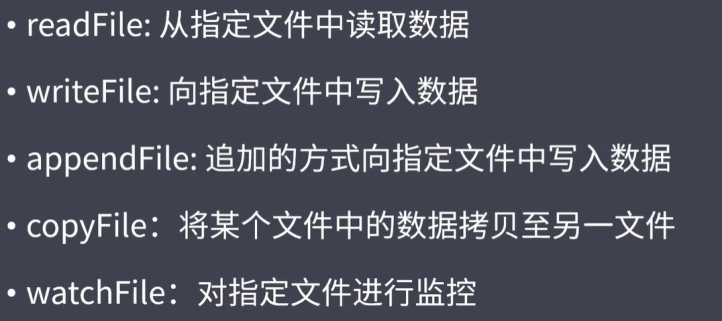
flag表示对文件的操作方式

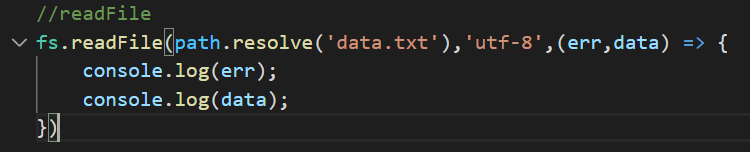


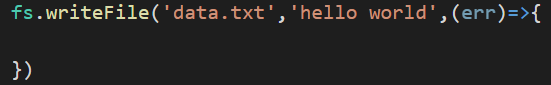
文件描述符fd

操作系统分配给被打开文件的标识

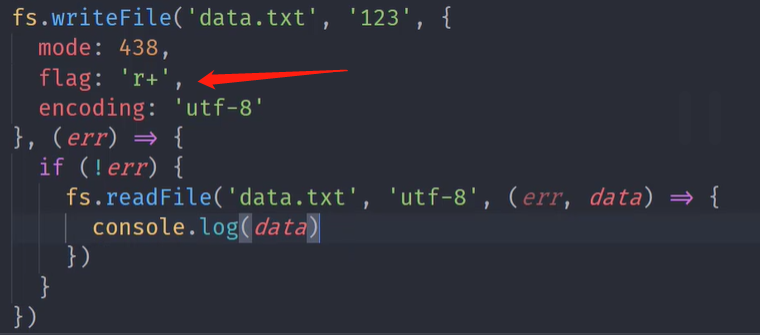
### API







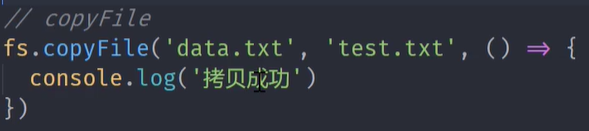
写操作如果文件不存在则执行创建操作



r+的操作就是直接在后面加，不是清空再写入



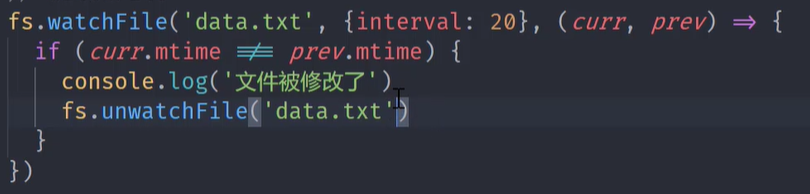
appendFile是在尾部追加，不删除之前的



把data.txt的内容拷贝到test.txt

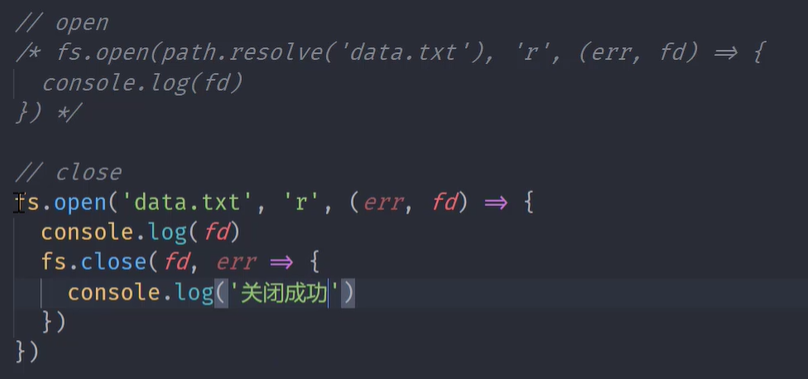


每20毫秒进行判断文件是否被修改

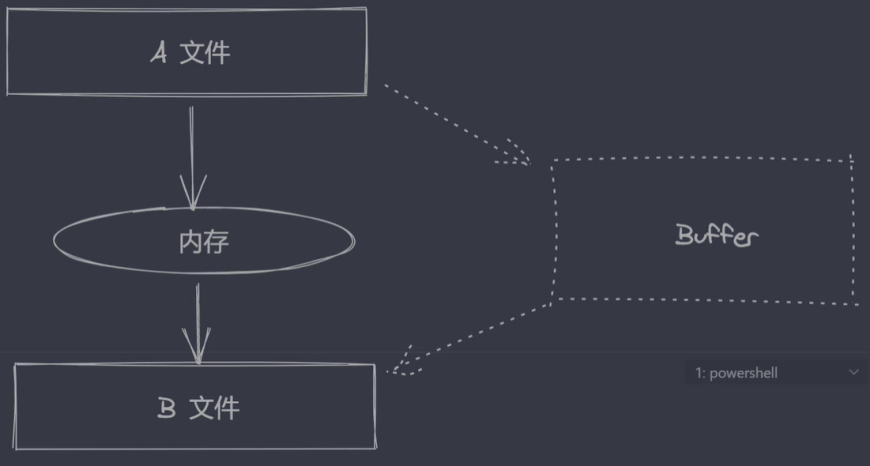


用unwatchFile进行结束监控

#### 文件打开和关闭

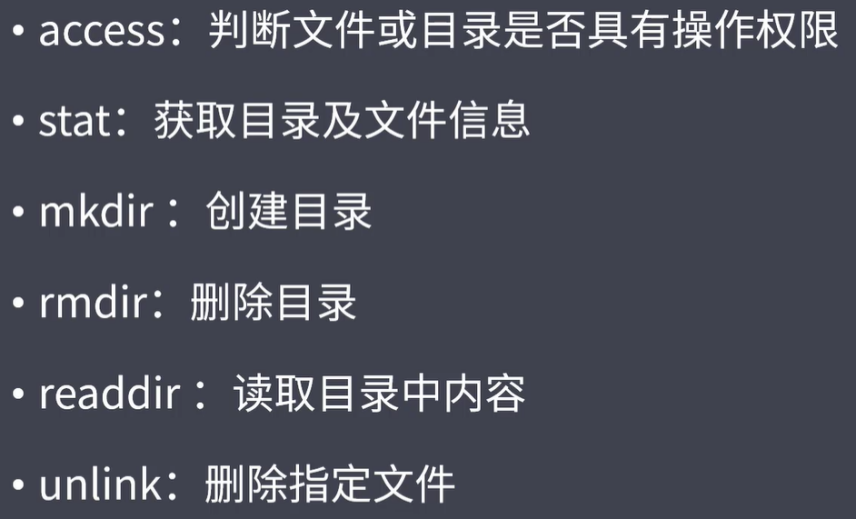


#### 大文件读写





#### 目录操作API



# 链表

## 数组的缺点：

数组长度有上限

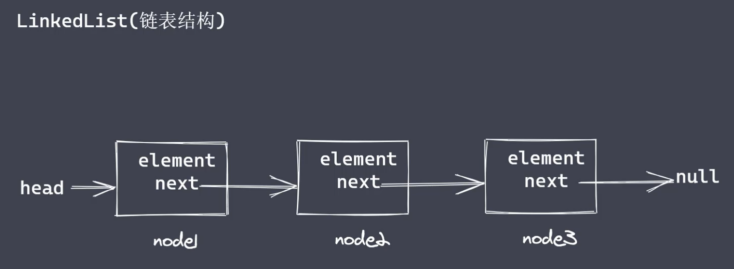
数组存在塌陷（空位）问题

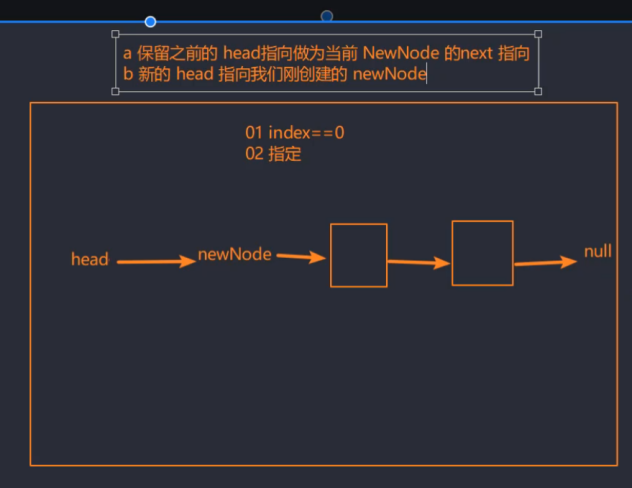
## 链表分类

双向

单向

循环





# 网络通信

## 通信原理

### 通信必要条件

1 主机之间需要有传输介质

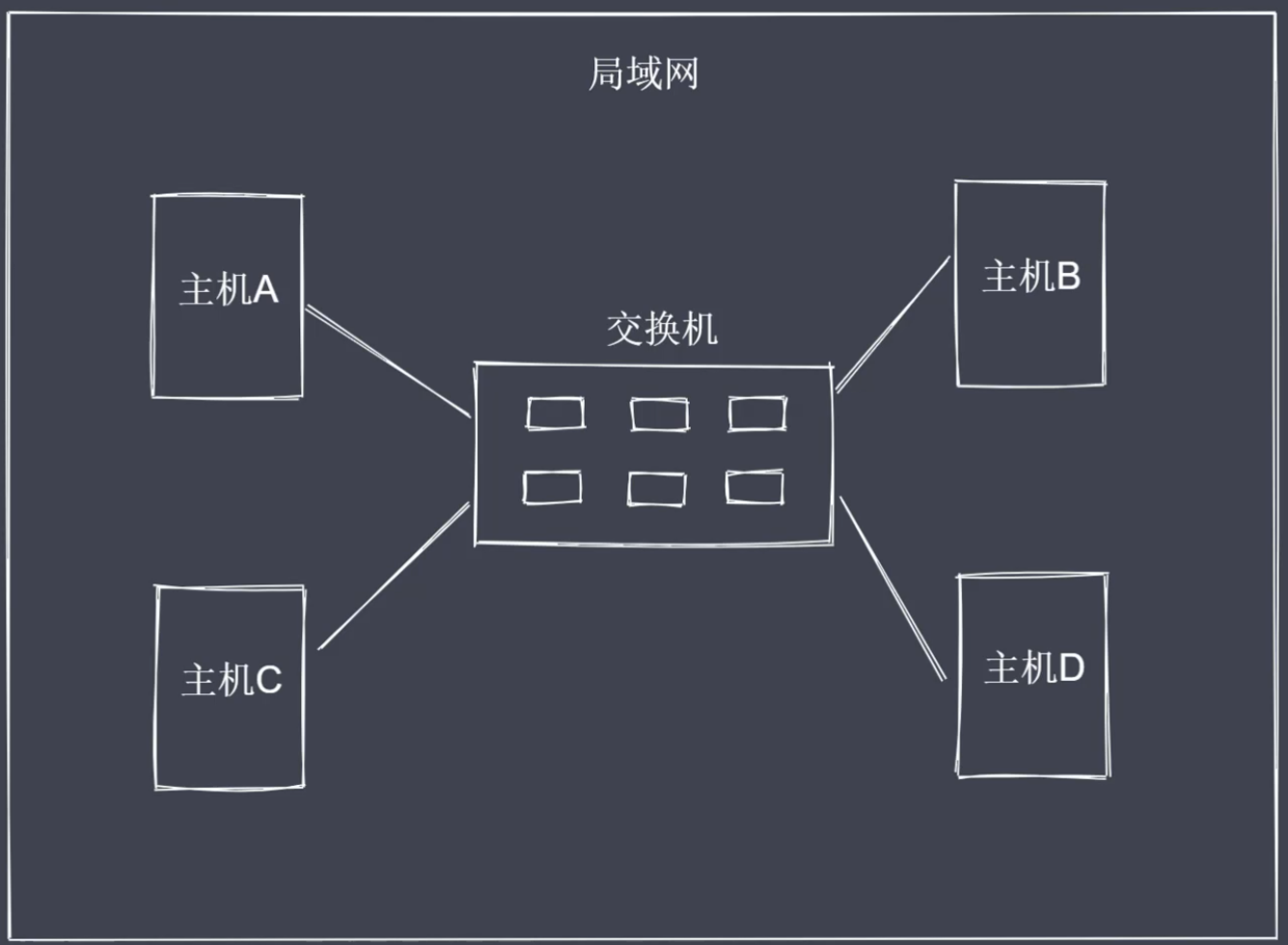
2 主机上必须要有网卡设备，用来信号的调制和解调制，其中，把二进制的信息转为高低电压的过程就是数据的调制功能

3 主机之间需要协商网络速率

### 通讯方式

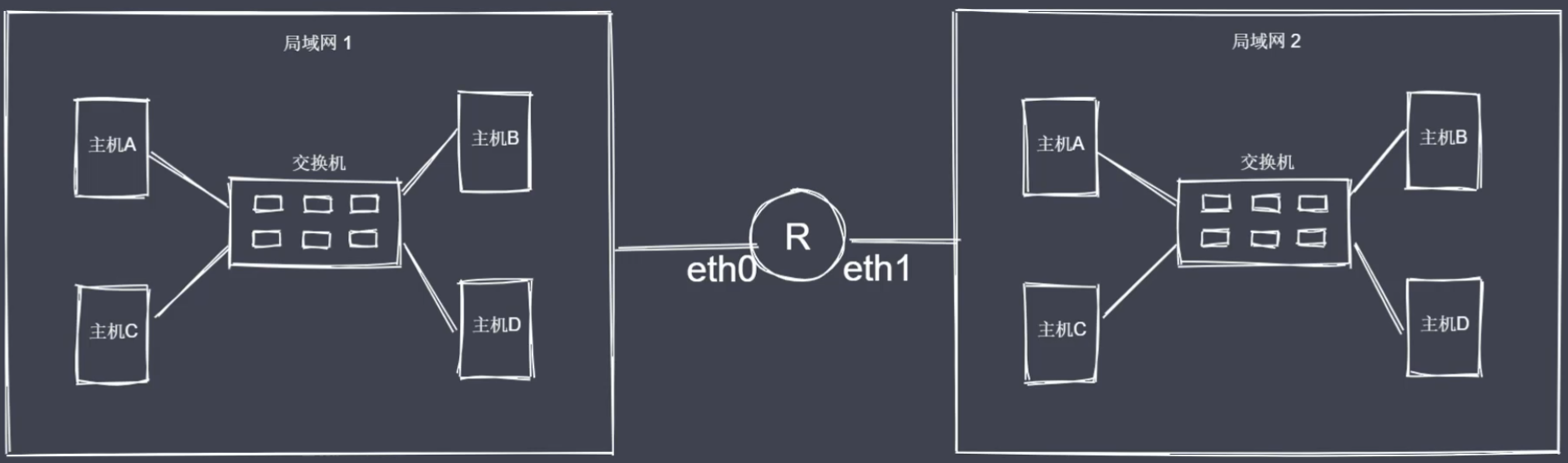
交换机通讯

1 每台主机都有个Mac地址，唯一标识网卡地址，物理地址

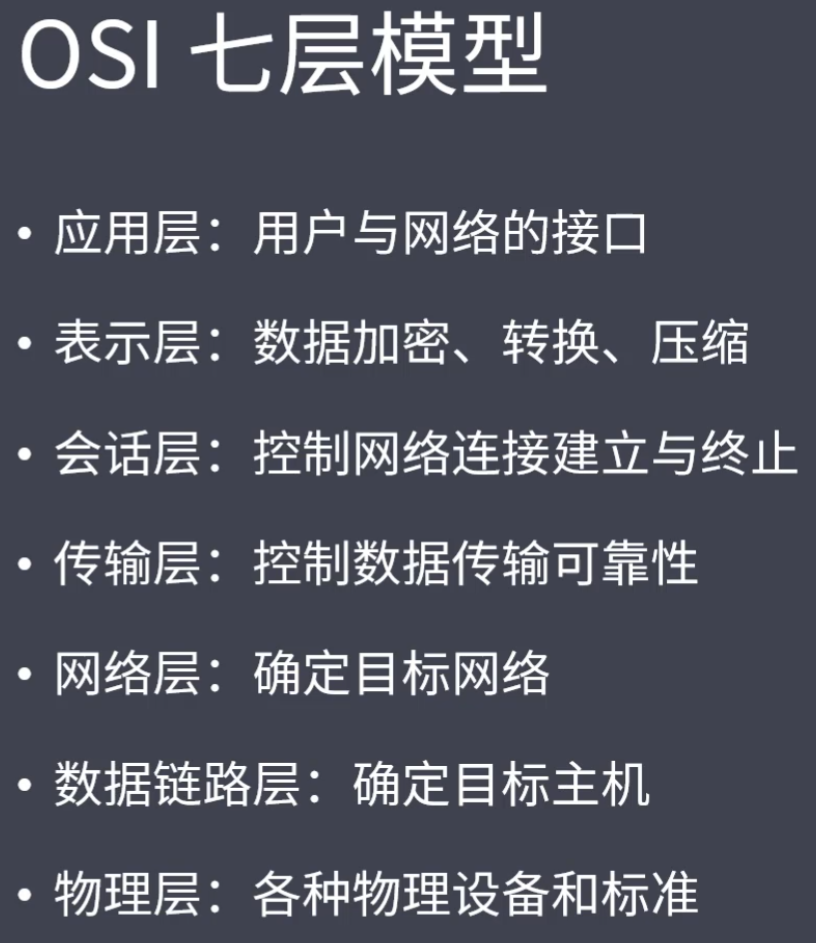


2 局域网存在大量主机会造成广播风暴，主机A发送消息到交换机，BCD接受消息，然后根据Mac地址把不要接收的数据扔掉。

路由器通讯



### 网络层次模型



#### 传输数据时，5个层分别发生了什么

1 应用层：产生真正的传输数据

2 传输层：有基于端口的TCP与UDP协议，端口是主机上用于唯一确定一个应用进程，数据在这里会被包裹上目标应用端口和应用在当前主机上的源端口

3 网络层：通过IP协议确定目标主机所在的网络，数据在这里会被包裹上目标主机的IP和源IP

4 数据链路层：通过Mac地址找到目标主机，数据在这里会被包裹上目标主机的Mac地址和当前主机的源Mac地址

5 物理层：经过网卡的调制，二进制数据会变成高低电压传输到目标主机的网卡

## Http协议

const exec = require('child\_process').exec

export **function** start () {

 // 任何你期望执行的cmd命令，ls都可以

 let cmdStr1 = 'your command code'

 let cmdPath = './file/'

 // 子进程名称

 let workerProcess

 runExec(cmdStr1)

**function** runExec (cmdStr) {

  workerProcess = exec(cmdStr, { cwd: cmdPath })

  // 打印正常的后台可执行程序输出

  workerProcess.stdout.on('data', **function** (data) {

   console.log('stdout: ' + data)

  })

  // 打印错误的后台可执行程序输出

  workerProcess.stderr.on('data', **function** (data) {

   console.log('stderr: ' + data)

  })

  // 退出之后的输出

  workerProcess.on('close', **function** (code) {

   console.log('out code：' + code)

  })

 }

}