

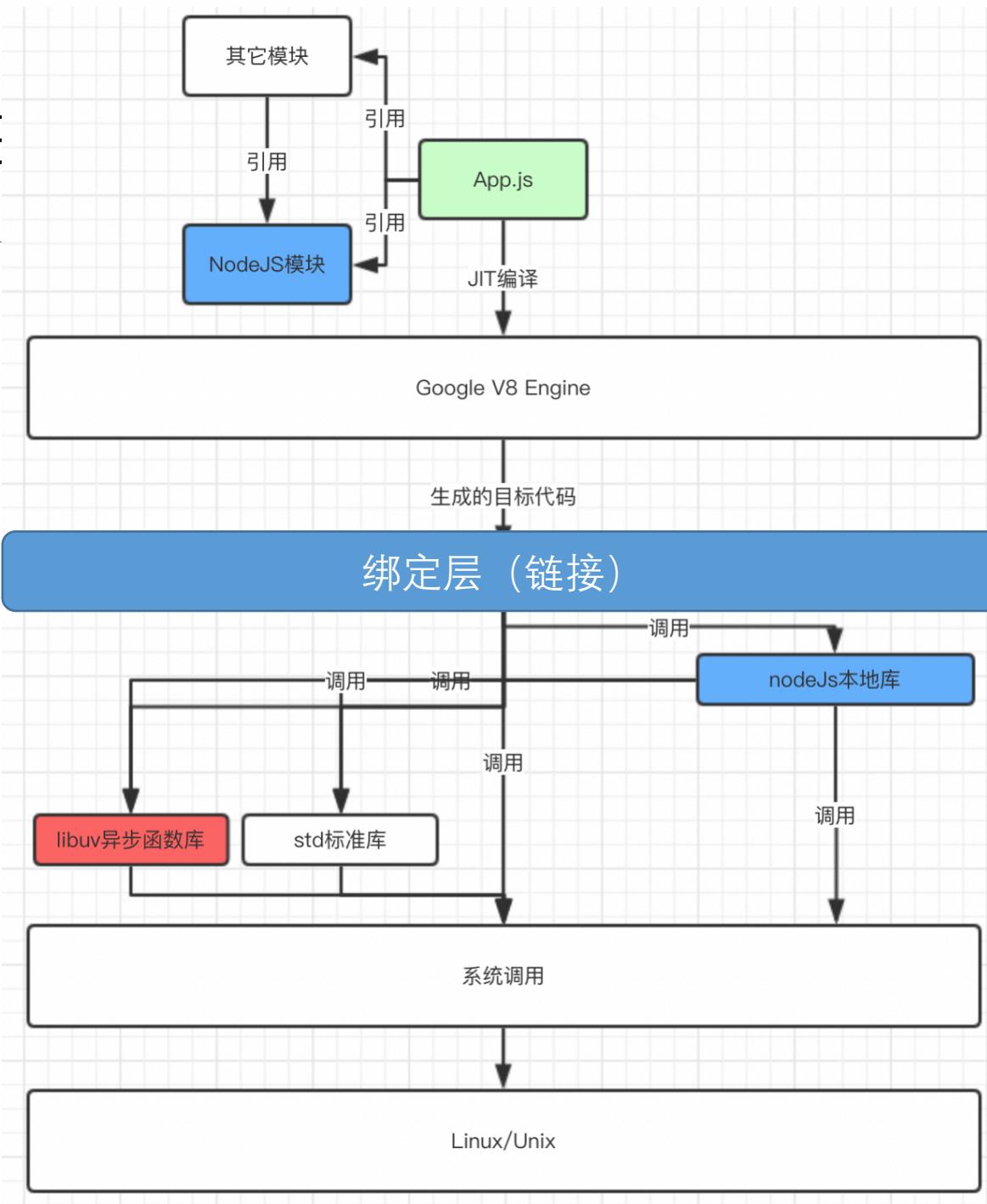
Node.js(MEVN/LAMP)

<https://nodejs.org/en/:since 2009>

- 本质上：一个服务器端的JS JIT解释器+JS附加库
- 适合用于高IO负载，低CPU负载的，DIRT型应用
- 成为革命性的JS全栈开发的关键技术
- 事件驱动方式
- 异步操作
- JavaScript
- V8引擎



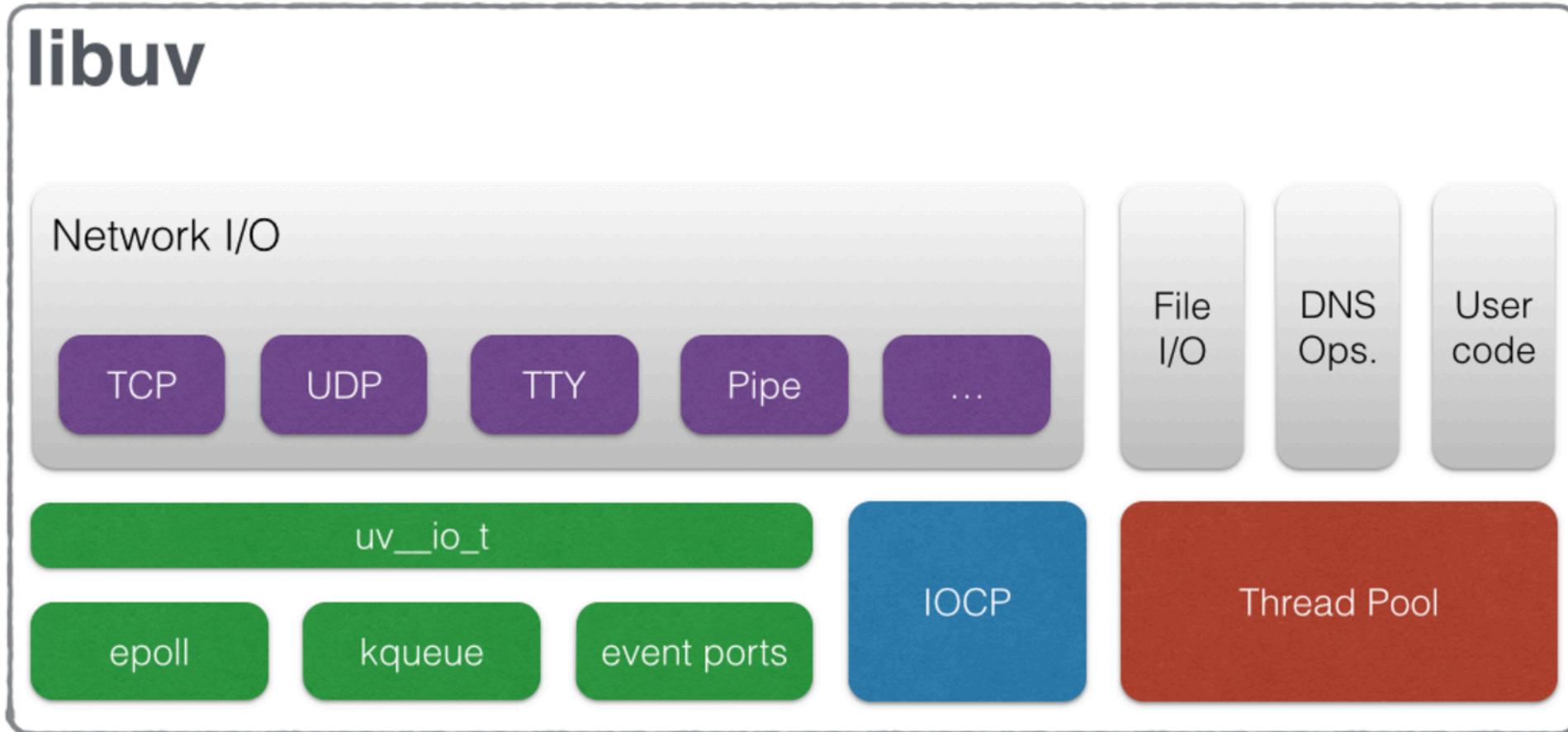
原理工作



V8 Javascript engine

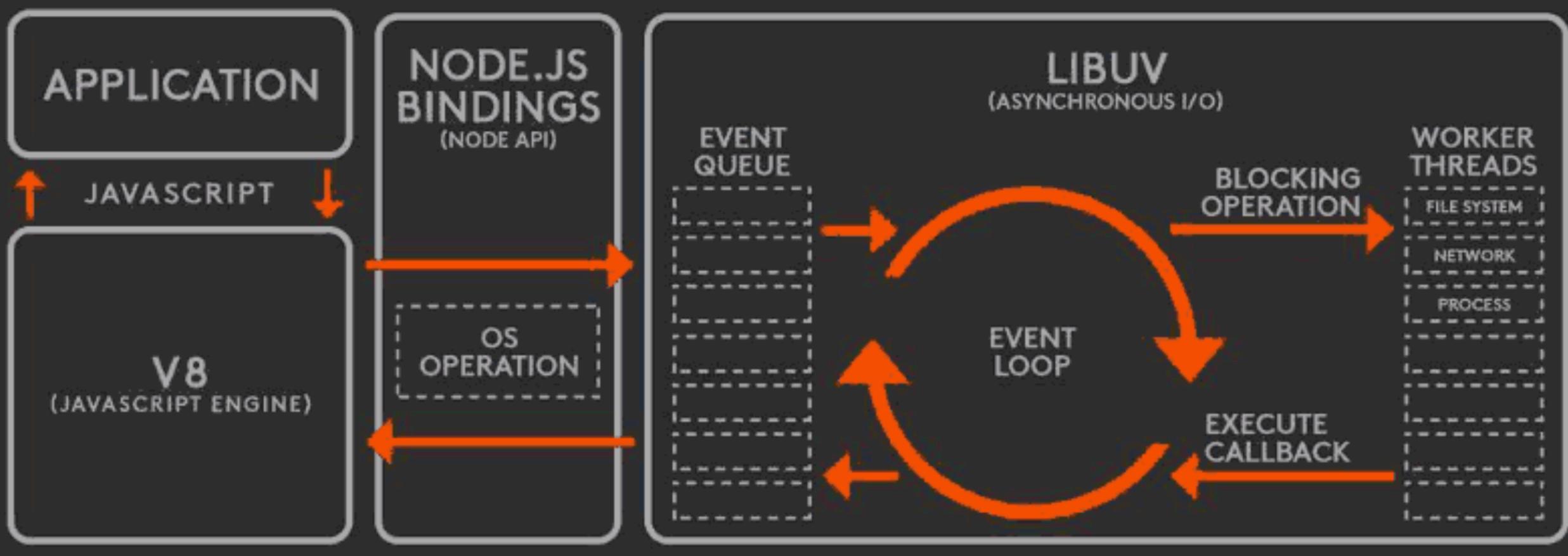
- JavaScript的即时编译器
- 最早应用于Chrome
- 作用类似于Java的JIT/AOT
- 让JavaScript可以压榨电脑的硬件

<http://libuv.org/>



THE NODE.JS SYSTEM

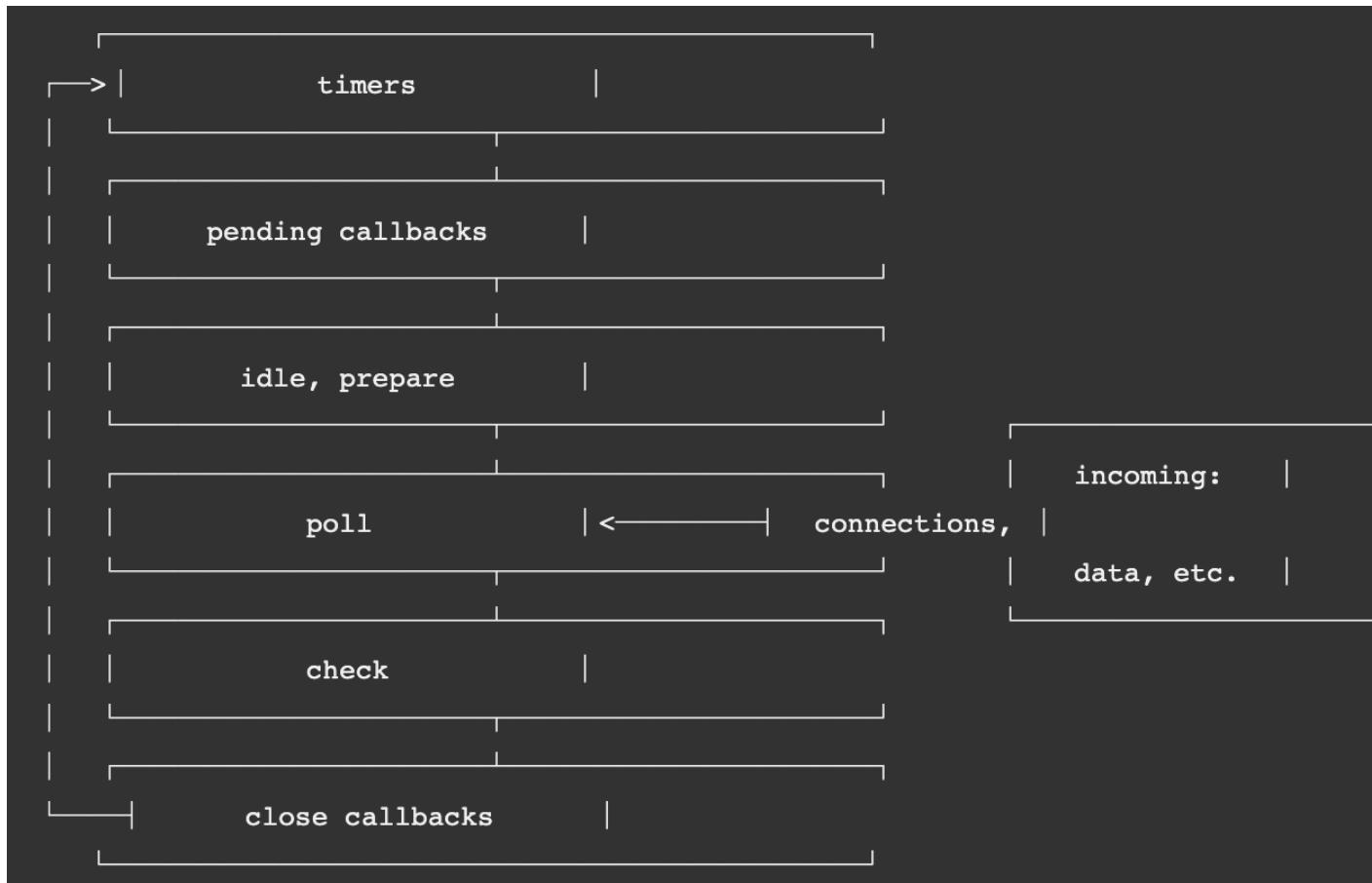
A DIAGRAM FROM
MODULUS



Event Loop

官网的有关知识：

<https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/#setimmediate-vs-settimeout>



Phases

- **timers:**
 - this phase executes callbacks scheduled by `setTimeout()` and `setInterval()`.
- **pending callbacks:**
 - executes I/O callbacks deferred to the next loop iteration.
- **idle, prepare:**
 - only used internally.
- **poll:**
 - retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and `setImmediate()`); node will block here when appropriate.
- **check:**
 - `setImmediate()` callbacks are invoked here.
- **close callbacks:**
 - some close callbacks, e.g. `socket.on('close', ...)`.

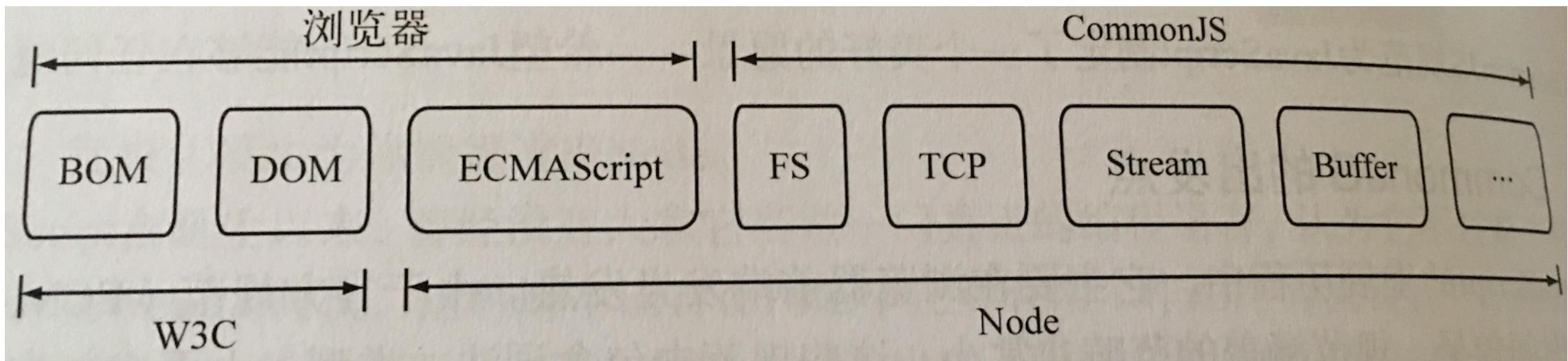
分析执行结果

- const fs =require('fs')
setImmediate(()=>*console.log("immediate process")*)
process.nextTick(()=>*console.log("next tick process")*)

setTimeout(()=>console.log('time out process'),0)
fs.open("abc.txt",*(err,fd)=>console.log("i/o callback",fd))*
for (let i = 0; i < 10000; i++) {
 console.log(i)
}

common JS—commonjs.org

- ES不存在“标准库”
- common JS 相当于STL,JDK的类库的概念，但也未标准化
- 最杰出之处，提出了“模块”概念



模块

- 内建模块，如`console,process,globals`
- 自含模块：如：`fs,http,net,buffer....`
- 第三方模块：如：`cheerio,express,mongoose`
- 自定义模块：“`./xx.js`”

模块的开发使用

```
//util1.js  
//module.exports对象，添加属性  
exports.sum=function(a,b){return a+b;}  
exports.multiply=function(a,b){return a*b;}
```

```
//util3.js  
//替换module.exports对象（使用构造函数方式）  
module.exports=function(){  
    this.add=function(a,b){return a+b;}  
    this.multiply=function(a,b){return a*b;}  
}
```

```
//util2.js  
//替换module.exports对象（使用对象字面声明方式）  
module.exports={  
    sum:function(a,b){return a+b;},  
    multiply:function(a,b){return a*b;}  
}
```

```
//myapp.js  
  
var util1=require("./util1.js");  
var s1=util1.sum(1,3);//4  
  
var math=require("./util2.js");//对象在模块中已被实例化  
var s2=math.sum(1,5);//6  
  
var MyMath=require("./util3.js");//获取构造函数  
var math1=new MyMath();  
var s3=math1.multiply(3,4);//12  
console.log(s3);
```

module注意的几点事项

- 每次**require**操作是同步操作
- **require**操作会让“**exports=**”右边的代码获得一次执行机会
- 取得的对象，都是单例（系统会进行缓存）

控制台的输入和输出

- 输出: `console.[log|info]/[warn|error]`
- `util.format`

```
//控制台输入
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
rl.on('line', (cmd) => {
  console.log(`You just typed: ${cmd}`);
});
```

函数式异步编程

```
1 //传入函数形式参数
2 function systemCall(callback) {
3     //模拟进行系统调用,耗时2ms
4     setTimeout(()=>{
5         let fd=4;//模拟系统调用的现场信息
6         callback(fd);
7     },2);
8 }
9 function callbackHandler(fd) {
10    console.log("系统中获取文件句柄",fd);
11 }
12 //异步函数
13 systemCall(callbackHandler);
14 console.log("end....");
15
16 //运行结果:
17 // end....
18 // 系统中获取文件句柄 4
```

```
1 //实际例子
2 const fs=require("fs");
3 fs.open("./app.js",'r', (err,fd)=> {
4     console.log(fd);
5 })
```

事件驱动

- 所有的回调函数，都是事件发生时，自动回调
- 事件的现场信息，将会以参数形式，传入到回调函数中。

EventEmitter

- node中的很多对象都是EventEmitter实例，工作原理如下：

```
var Emitter=require('events');
var mye=new Emitter();
var mye1=new Emitter();

mye.on('kada',function(x){console.log('recv.....'+x);});
mye.on('kada',function(x){console.log('recv.....'+x);});
mye.emit('kada',"info");
```

```
> process instanceof EventEmitter
true
```

```
//注册warning的回调处理函数
process.on('warning',function(w){console.log('I receiving '+w.message);});
//发射事件
process.emitWarning("warning....");
```

一个小例子

```
const Emitter=require('events');
var server=new Emitter();
server.on("data", (msg)=>{
    console.log(msg);
});
server.on("close", ()=>{
    console.log("系统关闭");
});

//模拟系统底层事件，触发回调处理
server.listen=function () {
    let fd= setInterval(()=>{
        //模拟底层产生的消息
        let msg=Math.random();
        this.emit("data",msg);
    },1000);
    setTimeout(()=>{
        clearInterval(fd);
        this.emit("close");
    },5000)
};
server.listen();
```

核心模块

global

- 是一个“垫底对象”
- console
- __dirname,__filename
- module,exports(repl无效)module.exports==exports
- require()
- process
- 公用函数
- ...

process对象

- 代表着nodejs的主进程:
 - exit(0),kill(pid),pid,getuid(),getgid()
 - env,argv,
 - cwd(),chdir(directory)
 - stdin,stdout
- 自身是EventEmitter
 - exit,beforeExit(exit()),message(send()),warning(emitWarning())

child_process模块

- 由node主进程打开子进程的工具
- 以下命令，会传入回调
 - child.exec[File]('命令', (err, stdout, stderr)=>{})
- 以下命令，会返回命令执行结果：
 - try{out=child.exec[File]Sync()}catch(e){}
- 以下命令，会返回childProcess，可以在其中对其stdout进行监听
 - child.spawn('命令')
 - child.fork('node模块')

Buffer

- 内建模块
- 描述内存区的概念，是ES6的Uint8Array的子类
- 提供大量语法糖，完成数据的存取操作
- 涉及encoding操作有：
`utf8, ascii, utf16le, base64, binary, hex`
- 内存分配好，无法变长

分配内存

- `Buffer.from(array/ArrayBuffer/string);`
- `Buffer.alloc(size[fill],[encoding]);`

写入及读取

- buf.fill();
- buf.write(string)
- buf.writeInt8()/writeInt32BE()...
- buf.length
- buf[index],buf.readInt8()/buf.toString("utf8/ucs2");
- buf.slice(start,end)

例子

```
//分配n字节，初值为0  
let buf=Buffer.alloc(18);  
//从0位置写入32位整数，小端序(count为写入个数)  
let count=buf.writeInt32LE(123,0);  
//从count位置写入32位整数，小端序  
buf.writeDoubleBE(12.34,count);  
  
//从0位置读入32位整数，小端序  
let m=buf.readInt32LE(0);  
//继续读入  
m=buf.readDoubleBE(4);  
  
//从0位置写入文字  
let len=buf.write("中国",0,"UTF-8");  
//读字符串的方法  
let s=buf.toString("UTF-8",0,len);  
console.log(s);
```

完成文件复制

- `fs.read()`
- `Chunk`
- `fs.write()`

stream module

概览

- 单工
 - Writable, Readable,
- 双工
 - Duplex, Transform
- 都默认有缓冲区
 - stream.[writableBuffer|readableBuffer]
 - stream.[writable|readable]highWaterMark

stream.Readable

常见的流形式

- [客户端的 HTTP 响应](#)
- [服务器的 HTTP 请求](#)
- [fs 的读取流](#)
- [zlib 流](#)
- [crypto 流](#)
- [TCP socket](#)
- [子进程 stdout 与 stderr](#)
- [process.stdin](#)

基本总结

- 暂停模式
 - 流打开时，会有就绪事件**readable**
 - 流处于“暂停”状态，可以使用**read([size])**方法读入到**buf**中
 - 返回**buf**会小于等于**size**或**highWarterMark**，如果结尾时，会返回**null**
- 流动模式
 - 当监听“**data**”事件时和调用**resume()**，流会转换成为“流动模式”
 - 流动过程中，可以使用**pasue()**方法，使用流转换成为“暂停模式”
- **end**事件
 - 流读到了结尾（无论何总模式）都会收到**end**事件

流动模式

- 在**data**事件监听和**resume()**方法调用时，数据将流入缓存区，直到遇到结尾**EOF**，并触发**end**事件
- 可以使用**pipe(writable)**，进行流连接，此时会自动将数据流向目标流，并内部处理“背压问题”
- 流动的过程中，可以使用**pause()/resume()**进行模式切换

stream.Writable

常见的种类

- [客户端的 HTTP 请求](#)
- [服务器的 HTTP 响应](#)
- [fs 的写入流](#)
- [zlib 流](#)
- [crypto 流](#)
- [TCP socket](#)
- [子进程 stdin](#)
- [process.stdout、process.stderr](#)

事件与方法的对应

常用的方法,属性

- write(), end(),
 - destroy(),
 - setDefaultEncoding()
-
- writableHighWaterMark (缓存大小)
 -

事件

- `close`: `destroy()`或者`{emitClose:true}`触发
- `drain`: 当流的缓存消耗干净，触发
- `finish`: 当`end()`
- `pipe/unpipe`: 当`input.[un]pipe(output)`

文件复制（解决背压）

```
1  /**  
2   * 文件复制（解决背压问题）  
3  */  
4  const fs=require('fs')  
5  let input=fs.createReadStream('aa.txt')  
6  let output=fs.createWriteStream('bb.txt')  
7  //输入流关闭时，关闭输出流（输入流自动关闭）  
8  input.on('close',()=>output.destroy())  
9  //转换到"流动模式"  
10 input.on('data',chunk=>{  
11   let b=output.write(chunk)  
12   //写入缓存满时，input转入"暂停模式"，drain时，再次转入"流动模式"  
13   if(!b){  
14     input.pause()  
15     output.once('drain',()=>{  
16       input.resume()  
17     })  
18   }  
19 })
```

```
/**  
 * 文件复制（解决背压问题）  
 */  
fs.createReadStream('aa.txt').pipe(fs.createWriteStream('bb1.txt'))
```

FileSystem Module

概况

- 基于底层系统调用(POSIX)的异步（同步）封装
- 回调函数的第一个参数为error
- 提供的API：
 - `fs.Stats`类：表示文件状态
 - `fs.WriteStream`/`fs.ReadStream`：附加了新的事件和属性
 - `fs.Dirent`类，表示目录状态信息
 - `fs.`[若干方法]：与文件有关的系统调用

基本操作演示

```
// 异步
fs.open('aa.txt', 'w', function (err, fd) {
  if(!err) console.log(fd)
})
```

// 同步

```
try {
  let fd = fs.openSync('aa.txt')
  console.log(fd)
} catch (e) {
  // catch exception
}
```

- `'a'` - 打开文件用于追加。如果文件不存在，则创建该文件。
- `'ax'` - 与 `'a'` 相似，但如果路径已存在则失败。
- `'a+'` - 打开文件用于读取和追加。如果文件不存在，则创建该文件。
- `'ax+'` - 与 `'a+'` 相似，但如果路径已存在则失败。
- `'as'` - 以同步模式打开文件用于追加。如果文件不存在，则创建该文件。
- `'as+'` - 以同步模式打开文件用于读取和追加。如果文件不存在，则创建该文件。
- `'r'` - 打开文件用于读取。如果文件不存在，则出现异常。
- `'r+'` - 打开文件用于读取和写入。如果文件不存在，则出现异常。
- `'rs+'` - 以同步模式打开文件用于读取和写入。指示操作系统绕过本地的文件系统缓存。

这对于在 NFS 挂载上打开文件时非常有用，因为它允许跳过可能过时的本地缓存。它对 I/O 性能有显著提升。

这不会将 `fs.open()` 或 `fsPromises.open()` 转换为同步的阻塞调用。如果需要同步的操作，可以使用 `fs.readFileSync()` 或 `fsPromises.readFile()`。
- `'w'` - 打开文件用于写入。如果文件不存在则创建文件，如果文件已存在则截断文件。
- `'wx'` - 与 `'w'` 相似，但如果路径已存在则失败。
- `'w+'` - 打开文件用于读取和写入。如果文件不存在则创建文件，如果文件已存在则截断文件。
- `'wx+'` - 与 `'w+'` 相似，但如果路径已存在则失败。

Stats信息

```
Stats {
  dev: 16777220,
  mode: 33188,
  nlink: 1,
  uid: 501,
  gid: 20,
  rdev: 0,
  blksize: 4096,
  ino: 4266291,
  size: 12257280,
  blocks: 23944,
  atimeMs: 1571102300347.0676,
  mtimeMs: 1571101635895.1665,
  ctimeMs: 1571101635895.1665,
  birthtimeMs: 1570946640568.237,
  atime: 2019-10-15T01:18:20.347Z,
  mtime: 2019-10-15T01:07:15.895Z,
  ctime: 2019-10-15T01:07:15.895Z,
  birthtime: 2019-10-13T06:04:00.568Z }
```

```
16 //fs.Stats的演示
17 //以fd为基础的操作，函数如果以f开头
18 fs.open('aa.txt',function (err,fd) {
19   fs.fstat(fd,function (err,stat) {
20     console.log(stat)
21     fs.close(fd,err=>console.log("file closed."))
22   })
23 })
24 //以文件名为基础的操作
25 fs.stat('aa.txt',function (err,stat) {
26   if(!err) console.log(stat)
27 })
```


Demo

- 异步完成“目录创建，文件创建，文件写入”的全过程
- 递归列出所有文件及目录
- *递归删除所有文件

```
function dir1(path){  
    var isFile=fs.statSync(path).isFile();  
    if(isFile) return;  
    fs.readdir(path, function(args,x){  
        x.forEach((item)=>{  
            console.log(path+"/"+item);  
            dir1(path+"/"+item);  
        })  
    })  
}  
  
dir1("target");
```

net module

简介

- 提供了TCP协议的API
- Socket
 - duplex stream && EventEmitter
- Server

net.Socket

- new net.Socket().connect(port, ip, cb);
- net.createConnection(port, ip, cb);

```
const net=require('net')
let socket=new net.Socket();
// "输出流"的文字编码
socket.setEncoding('UTF-8')
var result='';
socket.on('data',function (resp) {
    console.log(resp)
    // socket.destroy();不可以在这里处理关闭事件（回调会多次发生）
})
socket.on('end',()=>console.log("end"))
socket.on('close',()=>console.log("close"))

//连接成功以后，发送http请求头
socket.connect(80,'www.baidu.com',function () {
    socket.write('GET / HTTP/1.1\n')
    socket.write('Host: www.baidu.com\n')
    socket.write('\n')
    socket.end(); //半关闭socket，服务器会首先关闭socket
})
```

TCP的半关闭问题

- 原理：
 - TCP的一方发送**FIN**包，表示自身不再发送，但仍可以接收
 - TCP的接收到**FIN**包，可以决定**[关闭|发送]**
- `socket.end()`方法：
 - 会发送**FIN**包，并将“**自身socket**”的输出关闭，表示自己没有数据发送，但仍可以接收数据
 - “**对方socket**”会收到**FIN**包，并触发“**end**”事件，可以处理“**继续发送或关闭**”

net.Server

- an [EventEmitter](#)
 - 最重要的connection事件，回调函数(cb)参数类型为Socket
 - `server.on(connection, socket=>{})`;
- 创建方式
 - `server=net.createServer([cb])`;
 - cb指定时，可以不使用connection事件
- 启动方式
 - `server.listen(port, cb)`;
 - cb指定时，可以不使用connection事件

```
var net=require("net");
var clientList=new Map();
var server=net.createServer();
server.on("connection",function(sk){
    sk.write("hello\n");
    sk.on('data',function(chunk){
        var s=new String(chunk);
        console.log(s+".....");
        sk.write("hell:"+s);
        //sk.end();关闭socket
        if(s.indexOf("bye")>=0){
            sk.destroy();//关闭socket
        }
    }).on("close",function(){
        console.log("server-client is closed");
        server.close(function(){
            console.log("server is closed");
        })
    });
});
server.listen(7999);
```

path,url

path

```
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..');  
// 返回: '/foo/bar/baz/asdf'
```

```
path.parse('/home/user/dir/file.txt');  
// 返回:  
// { root: '/',  
//   dir: '/home/user/dir',  
//   base: 'file.txt',  
//   ext: '.txt',  
//   name: 'file' }
```

```
path.dirname('/foo/bar/baz/asdf/quux');  
// 返回: '/foo/bar/baz/asdf'
```

```
path.extname('index.html');  
// 返回: '.html'  
  
path.extname('index.coffee.md');  
// 返回: '.md'  
  
path.extname('index.');//  
// 返回: '..'  
  
path.extname('index');//  
// 返回: ''  
  
path.extname('.index');//  
// 返回: ''  
  
path.extname('.index.md');//  
// 返回: '.md'
```

url模块

```
1 const url=require('url')
2 let s="protocol://username:password@hostname:port pathname?query#hash"
3 let urlString="https://john:123@www.johnyu.cn:8080/ab/xy?page=1&favs=foot&favs=swim#me";
4 //解析，并将query部分进行解析
5 let myurl=new url.parse(urlString, {parseQueryString: true})
6 console.log(myurl)
```

```
Url {
  protocol: 'https:',
  slashes: true,
  auth: 'john:123',
  host: 'www.johnyu.cn:8080',
  port: '8080',
  hostname: 'www.johnyu.cn',
  hash: '#me',
  search: '?page=1&favs=foot&favs=swim',
  query: 'page=1&favs=foot&favs=swim',
  pathname: '/ab/xy',
  path: '/ab/xy?page=1&favs=foot&favs=swim',
  href:
  'https://john:123@www.johnyu.cn:8080/ab/xy?page=1&favs=foot&favs=swim#me' }
```

http module

概述

- **http.ClientRequest:**
 - Emitter|Writable
 - 用于“封装客户端的http的请求API”
- **http.ServerResponse:**
 - Writable|Emitter
 - 用于封装“服务器端的http响应”
- **http.IncomingMessage**
 - Readable|Emitter
 - 用于封装“服务器端的http请求”
 - 用于封装“客户端的http响应”
- **http.Server**
 - Emitter
 - 最重要的request事件，用于完成http的服务

http client

```
var postData = querystring.stringify({
  'msg' : 'Hello World!'
});

var options = {
  hostname: 'www.google.com',
  port: 80,
  path: '/upload',
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': postData.length
  }
};
```

```
var req = http.request(options, (res) => {
  console.log(`STATUS: ${res.statusCode}`);
  console.log(`HEADERS: ${JSON.stringify(res.headers)}`);
  res.setEncoding('utf8');
  res.on('data', (chunk) => {
    console.log(`BODY: ${chunk}`);
  });
  res.on('end', () => {
    console.log('No more data in response.')
  })
});
```

```
req.on('error', (e) => {
  console.log(`problem with request: ${e.message}`);
});

// write data to request body
req.write(postData);
req.end();
```

http:Http通讯模块

```
var http=require("http");
var qs=require('querystring');
var urlParser=require('url');

var server=http.createServer();
server.on('request',function(req,res){
    //req is instanceof http.MessageIncomming
    //res is instacneof http.ServerResponse
    //when client request,will callback!
    //单线程, 闭包的形式
    res.end("complete! !");
});
server.listen(3001);
```

```
> url.parse('/crm/login?uname=john&psd=123');
Url {
  protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '?uname=john&psd=123',
  query: 'uname=john&psd=123',
  pathname: '/crm/login',
  path: '/crm/login?uname=john&psd=123',
  href: '/crm/login?uname=john&psd=123' }
```

http.ServerResponse

```
res.setHeader('Content-Type', 'text/html');
res.setHeader('X-Foo', 'bar');
res.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
//另一种设置头的方案
res.writeHead(200, {'Content-Type': 'text/plain', 'uname':'john'});
res.write("content");

//res.statusCode=404;
//res.statusMessage='Not found';

//重定向的方法
// res.statusCode=302;
// res.setHeader("Location","http://www.baidu.com");

res.end();
```

http.MessageIncoming:(request)

```
//全部的请求头(JSON形式)
var headers=req.headers;
res.write(JSON.stringify(headers));

//GET /status?name=ryan HTTP/1.1\r\n
var url=req.url;// '/status?name=ryan'，然后可以用url模块解析

//获取post请求中的请求体
req.setEncoding('utf8');
req.on('data',function(chunk){
    console.log(chunk);
});
req.on('end',function(){
    console.log("complete....");
});
```

url & querystring module

```
> require('url').parse('/status?name=ryan')
{
  href: '/status?name=ryan',
  search: '?name=ryan',
  query: 'name=ryan',
  pathname: '/status'
}
```

```
> require('url').parse('/status?name=ryan', true)
{
  href: '/status?name=ryan',
  search: '?name=ryan',
  query: {name: 'ryan'},
  pathname: '/status'
}
```

```
> require("querystring").parse('uname=john&age=100');
{ uname: 'john', age: '100' }
```

案例：静态服务器/文件下载服务

http模块的深入思考

- 没有实现路由！！！