因为操作都是异步 所以需要包装

---

没window or document 更多的是文件的读取和写入

装饰器:

ES7 当装饰对象是类的时候 我们操作的就是这个类的本身 即装饰器函数的第一个参数

---

导入

require

`require`函数用来在一个模块中引入另外一个模块。传入一个模块名，返回一个模块导出对象。用法：`let cc = require("模块名")`，其中模块名可以用绝对路径也可以用相对路径,模块的后缀名.js可以省略。例如：

```
let cc1 = require('./main.js')
let cc2 = require('home/src/main.js')
let cc3 = require('./main')
```

require默认是.js文件

---

导出

exports --> 默认导出的对象

```
let a = 10
export.a = a  //exports 默认是一个空对象
    {
            a : 10
    }
```

modules.exports.c = c /module.export 最终会指向exports

exports = module.exports

最终导出的时候export会寻找module.export

```
export = {
    user:"小明"
}
```

module.exports可以导出一个对象也可以指导出一个字符串，而exports只能通过exports.属性名 = 值

但export 只能通关exports.a

**module.exports**能导出对象

👆👆👆

总结：以module.exports为准

---

导入模块只会导入一次

```
let a = require('./index1')
// 1在没有任何内容导出去的情况
// 2require获取文件路径时，可
```

```
let b = require('./index1')
```

index1中有console.log(123)
最终只会输出一个123

---

**npm**

```
npm list 查看安装了哪些
npm root 包安装路径
npm config set registry https://registry.npm.taobao.org

npm install -g cnpm --registry=https://registry.npm.taobao.org

 --save-dev //安装到开发环境
 --save     //安装到生产环境
```

---

文件读写

```
var fs = require("fs");
fs.open(path,callback)
fs.openSync(path)    //同步
var fd = fs.openSync("hello.txt","r")  //r read
var content = fs.readFileSync("helloText",{flag:"r",encodeing:"utf-8"})
//read一般异步
fs.readFile('hello.txt',{flag:"r",encodeing:"utf-8"},function() {})
Buffer.alloc(20)   开辟20的内存

文件打开错误 可以试着用trim
```

写文件

```
fs.writeFile("test.txt","晚饭吃啥",{flag:true,encoding:"utf-8"},function(err){
```

```
2
3 })
```

封装异步

```
function toWrite(path,word) {
    return new Promise (function(resolve,reject){
            fs.writeFile(path,word,{flag:true,encoding:"utf-8"},function(err){
                if(err){
                    reject()
                }
                else {
                    resolve()
                }
            })
    })
}
```

unlink删除

**Buffer(缓存区) （了解 ）**

数组不能进行二进制操作

js数组不能像java python 高校

buffer内存空间开辟出固 定大小的内存

```
var str = "helloworld"
//字符串转成buffer类型
let buf = Buffer.from(str)
console.log(buf.toString())
//buffer 是二进制的连续的
//开辟一个空的buffer
let buf1 = Buffer.alloc(10)
buf1[0] = 10
======================
let buf2 = Buffer.allocUnsafe(10)
//效率高 任意操作 但容易内存泄漏
```

---

目录读取

```
fs.readdir(path,callback(err,files))              //异步
fs.readdir('../fs')
```

删除

```
rmdir(path,callback())                    //异步
```

---

读入

```
let readLine = require("readline") //导入
let r1 = readline.createIneterface({
    output:process.stdin,
    input:process.stdin

}) //实例化
//例子 提问时间
r1.question("你的名字是",function(answer){
    console.log("答复: ",answer)
    r1.close()     //结束
})
r1.on('close',function() { //监听结束事件
    procss.exit(0)
})
```

**作用： 接收输入的结果 配合fswrite填写到文件里**

---

stream(流)

# Node.js Stream(流)

Stream 是一个抽象接口，Node 中有很多对象实现了这个接口。例如，对http 服务器发起请求的request 对象就是一个 Stream，还有stdout（标准输出）。

Node.js，Stream 有四种流类型：

- **Readable** - 可读操作。
- **Writable** - 可写操作。
- **Duplex** - 可读可写操作.
- **Transform** - 操作被写入数据，然后读出结果。

所有的 Stream 对象都是 EventEmitter 的实例。常用的事件有：

- **data** - 当有数据可读时触发。
- **end** - 没有更多的数据可读时触发。
- **error** - 在接收和写入过程中发生错误时触发。
- **finish** - 所有数据已被写入到底层系统时触发。

读取大数据的时候 像水流一样 一点一点读进来 不会怕内存不够

手里有一块砖头 我想拿手机 就把砖头扔掉拿手机

创建写入流

```
let ws = fs.createWriteStream(path,[可选配置操作])
```

```
//文件打开
ws.on('open',function() {

})
//写入状态准备好
ws.on("ready",function() {

})
//文件写入完成 关闭
ws.on("close",function() {

})
//写入的内容 多次慢慢写入
ws.write("hello1",function(err){

})
ws.write("hello2",function(err){

```

```
})
ws.write("hello3",function(err){

})
//结束
wx.end(function(){
    console.log('文件写入完成')
})
```

读入流

创建读取流的实例

```
var rs = fs.createReadStream(path,config)
console.log(rs)
//读取文件打开
rs.on('open',function() {

})
//读取流结束
rs.on("close",function() {

})
rs.on('data',function(chunk) {
    console.log(chunk)
    //这里可以边读边将chunk写进去
})
//更简单
rs.pipe(ws) //从读到写
```

---

事件循环

事件驱动

Node.js 单线程类似进入一个while(true)的事件循环，直到没有事件观察者退出，每个异步事件都生成一个事件观察者，如果有事件发生就调用该回调函数.

# 设计观察者模式

```
lcEvent.on('fileSuccess',function(eventMsg){
    console.log("3查看所有用户学校的详细信息")
})

let lcEvent = {
    event:{
        //fileSuccess:[fn,fn,fn]
    },
    on:function(eventName,eventFn){
        if(this.event[eventName]){
            this.event[eventName].push(eventFn)
        }else{
            this.event[eventName] = []
            this.event[eventName].push(eventFn)
        }
    }
}

    ſ
},
emit:function(eventName,eventMsg){
    if(this.event[eventName]){
        this.event[eventName].forEach(itemFn => {
            itemFn(eventMsg)
        });
    }
}
```

对fileSuccess会触发的函数进行foreach

```
fs.readFile("hello.txt",{flag:"r",encoding:"utf-8"},function(err,data){
    if(err){
        console.log(err)
    }else{
        console.log(data)
        lcEvent.emit('fileSuccess',data)
        //1数据库查看所有的用详细信息
        //2统计年龄比例
        //3查看所有用户学校的详细信息
    }
})
```

```
lcEvent.on('fileSuccess',function(eventMsg){
    console.log("1数据库查看所有的用户详细信息")
})
lcEvent.on('fileSuccess',function(eventMsg){
    console.log("2统计用户年龄比例")
})
lcEvent.on('fileSuccess',function(eventMsg){
    console.log("3查看所有用户学校的详细信息")
})
```

//对fileSuccess的三个事件进行了注册

理解：将所有的事件方法注册在abcEvent.on,将所有的事件注册在abcEvent.event 如果外部有什么条件达成了 就用abcEvent的emit触发

emit处理一下需要携带的参数 并且 循环一下 是否所需触发的事件在abcEvent中 如果有的话就会到abcEvent.on中对其进行处理

**abcEvent.on是用来注册事件的**

**on就是注册事件和携带回调函数的**

**emit是用来触发事件的 和 携带事件所需参数**

注意( 在emit触发之前 所有的on会自己去注册事件到(提前)abcEvent.event中 emit触发了就会到event中去寻找并一个个触发 )

I made it

```
let  abcEvent = {
    count:1,
    event:{
        //各种fn
    },
    on(eventName,eventFn) {
        if(this.event[eventName])
        {
            this.event[eventName].push(eventFn)
        }
        else{
            this.event[eventName] = [eventFn]
        }
    },
    emit(eventName,msg) {
        if(this.event[eventName])
```

```
        {
            this.event[eventName].forEach(element => {
                element(msg)
            });
        }
    }
}
abcEvent.on('success',function() {
    console.log(abcEvent.count++)

})
abcEvent.on('success',function() {
    console.log(abcEvent.count++)

})
abcEvent.on('success',function() {
    console.log(abcEvent.count++)

})
abcEvent.on('wrong',function() {
    console.log(abcEvent.count++)
    console.log("我失败了")
})
let a = 13 + 24
if(a == 37)
{
    abcEvent.emit('success')
}
console.log(a)
```

引入事件模块

```
let events = require("events")
let ee = new events.EventEmitter();
ee.on('helloSuccess',function() {

})
ee.on('helloSuccess',function() {

})
```

## path

```
let path = require("path")
```

```
//获取路径信息的扩展名
let baidu = "http://www.baidu.com"
let info = path.extname(strPath)
//info = .com

let arr = ['/..','','zhongji']
path.resolve(...arr)       //
path.join()        //合成路径
console.log(__dirname)  //输出当前完整路径
console.log(__filename)  //输出当前执行文件目录
console.log(path.parse(__filename)) //返回文件的各个信息
```

## os

获取操作系统的信息

```
os.cpus()          //获取操作系统的cpu相关信息
os.totalmen()      //获取操作系统的cpu相关信息
os.freemen()       //闲置内存
```

---

创建目录

```
function fsDir(path) {
    return new Promise((resolve,reject)=>{
            fs.mkdir(path,function(err) {
                    if(err) reject()
                    else resolve("创建成功")
            })
    })
}

```

---

---