**初级教程**

引入模块：require(‘模块’)，获取模块，使用相应的功能。

**os模块**

提供操作系统基本信息。

var os = require("os");

var result = //os.platform(); //查看操作系统平台

//os.release(); //查看操作系统版本

//os.type(); //查看操作系统名称

os.arch(); //查看操作系统CPU架构

**process**

是一个内置全局对象，可以在代码任意位置访问该对象。它代表node.js代码宿主的操作系统进程对象。

使用process对象可以截获进程的异常、退出等事件，也可以获取进程的当前目录、环境变量、内存占用等信息，还可以执行进程退出、工作目录切换等操作。

查看应用程序当前目录：process.cwd();

改变应用程序目录：process.chdir(‘目录’);

stdout是标准输出流。

console.log = function(d){

process.stdout.write(d+'\n');

}

上面代码的作用就是就内容打印在输出设备上。

stderr是标准错误流，用来打印错误信息，可以通过它来捕获错误流信息。

stdin是进程的输入流，可以通过注册事件的方式获取输入内容：

process.stdin.on('readable', function() {

var chunk = process.stdin.read();

if (chunk !== null) {

process.stdout.write('data: ' + chunk);

}

});

要在程序内杀死进程，退出程序，可以用process.exit(code);参数code为默认返回的代码，若省略则默认返回0。

使用process.on()方法可以监听进程事件。

exit事件

当进程要退出之前，会触发exit事件。通过监听exit事件，我们可就以在进程退出前进行一些清理工作：

//参数code表示退出码

process.on("exit",function(code){

//进行一些清理工作

console.log("I am tired...")

});

var tick = Date.now();

console.log(tick);

uncaughtException事件

如果进程发生了未捕捉的异常，会触发uncaughtException事件。通过监听这个事件，你可以 让进程优雅的退出：

//参数err表示发生的异常

process.on("uncaughtException",function(err){

console.log(err);

});

//故意抛出一个异常

throw new Error("我故意的...");

为流设置编码：process.stdin.setEncoding(“编码”);

process.stdout.setEncoding(“编码”)

process.stderr.setEncoding(“编码”)

**fs模块**

支持I/O操作，fs模块的文件I/O是对POSIX函数的简单封装。

文件写入：fs.writeFile(filename, data, callback)，若文件已存在则覆盖。

**var** fs = require("fs");  
  
fs.writeFile("C:/Users/Administrator/Desktop/hehe.txt","hello world",**function**(err){  
 **if**(err) **throw** err;  
 console.log("文件被成功保存！");  
});

数据参数可以是string或者是Buffer,编码格式参数可选，默认为"utf8"，回调函数只有一个参数err。

若文件已存在只想添加一部分内容时，使用appendFile函数，它可以将新的内容添加到已有的文件中，如果文件不存在，则会创建一个新的文件。

例：fs.appendFile(文件名,数据,编码,回调函数(err));

**var** fs= require("fs");

fs.appendFile('C:/Users/Administrator/Desktop/hehe.txt', 'data to append', **function** (err) {  
 **if** (err) **throw** err;  
  
 //数据被添加到文件的尾部  
 console.log('The "data to append" was appended to file!');  
});

检查一个文件是否存在：fs.exists(文件,回调函数(exists))。回调函数有一个参数，为布尔类型，通过它来表示文件是否存在。

修改文件名：

fs.rename(旧文件，新文件，回调函数(err){  
 **if** (err) **throw** err;  
 console.log('Successful modification,');  
});

可以通过重命名文件来达到移动文件的目的。

读取文件：例：fs.readFile(文件,编码,回调函数);返回的是二进制码。

删除文件：例：fs.unlink(文件,回调函数(err));

创建路径：fs.mkdir(路径，权限，回调函数(err));

删除路径：fs.rmdir(路径，回调函数(err));

读取指定目录下的所有文件：fs.readdir(目录,回调函数(err,files));

**url模块**

parse函数作用是解析url，返回一个json的数组：

**var** url = require('url');  
**var** inf = url.*parse*('http://www.baidu.com');  
console.log(inf)

parse函数第二个参数是一个布尔类型的值。

parse函数的第三个参数也是布尔类型的，当参数为true，解析时会将url的"//"和第一个"/"之间的部分解析为主机名。

format函数功能与parse相反，它的参数是一个JSON对象，返回一个url地址。

**var** url = require('url');  
**var** h = url.format({  
 protocol: 'http:',  
 hostname:'www.baidu.com',  
 port:'80',  
 pathname :'/news',  
 query:{page:1}  
});

resolve函数的参数是两个路径。第一个路径是开始路径或者说是当前路径，第二个则是想要去往的路径，返回值是一个组装好的url。

**var** url = require('url');  
  
url.resolve('http://baidu.com/', '/1') // 'http://baidu.com/1'

**path模块**

包含一套用于处理和转换文件路径的工具集，用于处理目录的对象，提高开发效率。

normalize函数能将不符合规范的路径装换成标准路径，解析路径中的.与..，还能去掉多余的斜杠。

join函数将传入的多个路径拼接为标准路径并将其格式化，返回规范的路径，避免手工拼接路径字符串的繁琐。

var path = require('path');

var data = path.join('///you', '/are', '//beautiful');

console.log(data);

dirname用来返回路径中的目录名。

var path = require('path');

var data = path.dirname('/foo/strong/cool/nice');

console.log(data);

打印出’/foo/strong/cool’

basename函数可返回路径中的最后一部分，并可以对其进行条件排除。例子：

例1：path.basename('路径字符串');

例2：path.basename('路径字符串', '[ext]');//排除[ext]后缀字符串;

var path = require('path');

var data1 = path.basename('/foo/strong/basename/index.html');

var data2 = path.basename('/foo/strong/basename/index.html','.html');

分别返回index.html和index。

extname函数返回路径中文件的扩展名（以最后一个.开始，返回.及其以后的字符串，没有则返回空）。

**Query String模块**

用于实现URL参数字符串与参数对象之间的相互转换。

stringify函数：

var querystring= require('querystring');

var result = querystring.stringify({foo:'bar',cool:['xux', 'yys']});

运行结果：foo=bar&cool=xux&cool=yys

自定义分隔符和分配符：

querystring.stringify("对象"，"分隔符"，"分配符")

var querystring = require('querystring');

var result = querystring.stringify({foo:'bar',cool:['xux', 'yys']},'\*','$');

返回foo$bar\*cool$xux\*cool$yys

parse函数：反序列化字符串，转化得到一个对象。

querystring.parse("对象"，"分隔符"，"分配符")

**util模块**

是node.js的一个核心模块，提供常用的函数集合，用于弥补javascript的一些功能过于精简的不足。并且还提供了一系列常用的工具，用来对数据输出和验证。

inspect函数：util.inspect(object,[showHidden],[depth],[colors])是一个将任意对象转换为字符串的函数，通常用于调试和错误输出。它至少接受一个参数object，即要转换的对象。

format函数：根据第一个参数，返回一个格式化字符串。第一个参数是可包含零个或多个占位符的字符串。每个占位符被替换为与其对应的转换后的值。支持的占位符有："%s(字符串)"、"%d(数字<整型和浮点型>)"、"%j(JSON)"、"%(单独一个百分号则不作为一个参数)"。

isArray函数判断对象是否为数组类型。

isDate函数判断对象是否为日期类型。

isRegExp函数判断对象是否为正则类型。

**子进程：child\_process模块**

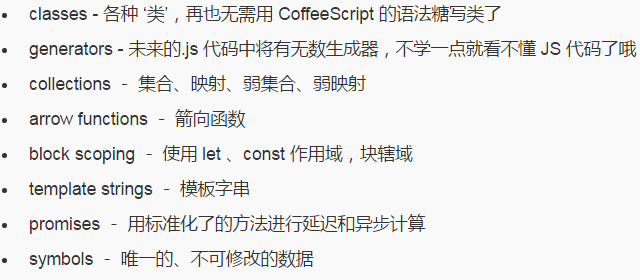
通过多进程来实现对多核CPU的利用。child\_process模块提供了四个创建子进程的函数：spawn，exec，execFile和fork。

spawn函数：用给定的命令发布一个子进程，只能运行指定的程序，参数需要在列表中给出。

**var** child\_process = require('child\_process');  
**var** child = child\_process.spawn( command );  
child.stdout.on('data', **function**(data) {  
 console.log(data);  
});

**Node.js 4.0**

新特性：



严格模式：与正常的JavaScript有些不同。首先，严格模式会将JavaScript陷阱直接变成明显的错误；其次，严格模式修正了一些引擎难以修正的错误，同样的代码有些时候严格模式会比非严格模式更快；再次，严格模式禁用了有可能在未来版本中定义的语法。

为整个script标签开启严格模式，需要在所有语句之前放一个特定的语句”use strict”;为某个函数开启严格模式需要把”use strict”声明放在函数体所有语句之前。

**let**允许把变量的作用域限制在块级作用域之中，它声明的变量不存在变量提升。而var申明的变量是函数级的。

**const**声明一个常量，遵循与变量相同的作用域规则。一个常量不能被重新赋值，也不能被重复声明。若声明时不对常量赋值，则该常量的值永远为undefined。常量不能与它所在的作用域内的其他变量或函数拥有相同的名称。

**类声明和类表达式**

ES6中的类实际上就是一个函数。正如函数的定义方式有函数声明和函数表达式两种，类的定义方式也有类声明和类表达式两种。

类声明：

'use strict'  
**class** People{  
 constructor(name,gender){  
 **this**.name = name;  
 **this**.gender = gender;  
  
  
 }  
 sayName (){  
 console.log("My name is: " + **this**.name);  
 }  
}  
  
**var** student = **new** People("Liuyawin","man");  
student.sayName();

类声明没有变量提升现象，必须先声明再使用，否则会抛出ReferenceError异常。

类表达式：

类表达式中，类名是可有可无的。如果定义了类名，则该类名只有在类体内部才能访问到。

'use strict';  
// 匿名类表达式  
**var** Polygon = **class** {  
 constructor(height, width) {  
 **this**.height = height;  
 **this**.width = width;  
 }  
};  
  
// 命名类表达式  
**var** Polygon = **class** Polygon {  
 constructor(height, width) {  
 **this**.height = height;  
 **this**.width = width;  
 }  
};

**构造函数**

类成员需要定义在一对花括号{}里，花括号里的代码和花括号组成了类体。类成员包括构造器和类方法（包括静态方法和实例方法）。

class根据constructor方法创建和初始化对象。

Constructor方法是类的默认方法，通过new关键字生成对象实例时，自动调用此方法。一个类只能有一个constructor方法。如果没有显示定义，一个空的constructor方法会被默认添加。

**class** Foo {  
 constructor() {  
 **return** Object.create(**null**);  
 }  
}  
**var** h = **new** Foo() **instanceof** Foo  
console.log(h);//false

constructor默认返回实例对象（即this），但是它也可以返回另一个对象。如上例所示，constructor函数返回一个新的对象，结果导致该实例对象不是Foo的实例。

constructor是一个特殊的类方法，它既不是静态方法也不是实例方法，它仅在实例化一个类的时候被调用。一个类有且只能有一个constructor方法，否则会抛出SyntaxError异常。

类和模块的内部默认就是严格模式，不需要使用use strict指定运行模式。

**静态方法**

static关键字定义一个静态方法。

静态方法是属于整个类的类方法，它在内存中的代码段会随类的定义而被分配和装载；实例方法是属于具体对象的方法，当对象被创建时，在对象的内存中会拥有此方法的专用代码段。应用静态方法可以用对象名加前缀，也可以使用类名加前缀；调用实例方法只能用对象名加方法。

非静态方法可以访问静态数据成员，也可访问非静态数据成员；静态方法只能访问静态数据成员。同样的非静态方法可以访问静态方法和非静态方法，静态方法只能访问静态方法。main方法是一个特殊的静态方法，是程序的入口点，每个程序中有且只能有一个。

**使用extends关键字创建子类。**

'use strict';  
**class** Animal {  
 constructor(name) {  
 **this**.name = name;  
 }  
  
 speak() {  
 console.log(**this**.name + ' makes a noise.');  
 }  
}  
  
**class** Dog **extends** Animal {  
 speak() {  
 console.log(**this**.name + ' barks.');  
 }  
}  
**var** dog = **new** Dog('NiNi');  
dog.speak();//NiNi barks.

**map**

一个简单的键值映射，任何值（包括对象和原始值）都可以作为一个键或一个值。

**var** m = **new** Map();  
**var** o = {p: "Hello World"};  
m.set(o, "content")  
m.get(o) // "content"

set方法也可以接受一个数组作为参数，该数组的成员是一个个表示键值对的数组。

**var** map = **new** Map([["name", "张三"], ["title", "Author"]]);  
map.size // 2  
map.get("name") // "张三"  
map.get("title") // "Author"

map的键实际上是跟内存地址绑定的，内存地址不一样就视为两个键。注意：

1. 如果使用对象作为键名，就不用担心自己的属性与原作者的属性同名。
2. 如果map的键是一个简单的类型值（数字、字符串、布尔值），则只要两个值严格相等，map将其视为一个键，包括0和-0。
3. 另外，虽然NaN不严格相等于自身，但map将其视为同一个键。

**map的属性和方法：**

size属性返回Map结构的成员总数。即返回映射对象中的键/值对的数目。

set(key, value)方法设置key所对应的键值，然后返回整个Map结构。如果key已经有值，则键值会被更新，否则就新生成该键。

set方法返回的是Map本身，因此可以采用链式写法。

get(key)方法读取key对应的键值，如果找不到key，返回undefined。

has(key)方法返回一个布尔值，表示某个键是否在Map数据结构中。

delete(key)方法删除某个键，返回true。如果删除失败，返回false。

clear()方法清除所有成员，没有返回值。

**Map遍历方法：**

keys()：返回键名的遍历器。

values()：返回键值的遍历器。

entries()：返回所有成员的遍历器。

forEach()：遍历Map的所有成员。

WeakMap

与map类似，区别在于WeakMap的键名只能是对象。这样设计的目的在于：键名是对对象的弱引用（垃圾回收机制不将该引用考虑在内），所以其对应的对象可能会被自动回收。当对象回收后，WeakMap自动移除对应的键值对。

WeakMap没有遍历操作和清空方法。它只有四个可用方法：get()、set()、has()、delete()。

**Set**

类似于数组，但成员值都是唯一的，没有重复的值。

**Set实例的属性和方法：**

Set结构的实例有以下属性。

Set.prototype.constructor：构造函数，默认就是Set函数。

Set.prototype.size：返回Set实例的成员总数。

Set实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。下面先介绍四个操作方法。

add(value)：添加某个值，返回Set结构本身。

delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。

has(value)：返回一个布尔值，表示该值是否为Set的成员。

clear()：清除所有成员，没有返回值。 上面这些属性和方法的实例如下。

**var** s = **new** Set();  
s.add(1).add(2).add(2);  
// 注意2被加入了两次  
console.log(s.size); // 2  
console.log(s.has(1)); // true  
console.log(s.has(2)); // true  
console.log(s.has(3)); // false  
console.log(s.delete(2));  
console.log(s.has(2)); // false

**遍历操作**

keys()：返回一个键名的遍历器

values()：返回一个键值的遍历器

entries()：返回一个键值对的遍历器

forEach()：使用回调函数遍历每个成员

key方法、value方法、entries方法返回的都是遍历器对象。由于Set结构没有键名，只有键值（或者说键名和键值是同一个值），所以key方法和value方法的行为完全一致。

**let set** = **new** Set(['red', 'green', 'blue']);  
**for** ( **let** item of **set**.keys() ){  
 console.log(item);  
}// red green blue  
**for** ( **let** item of **set**.values() ){  
 console.log(item);  
}// red green blue  
**for** ( **let** item of **set**.entries() ){  
 console.log(item);  
}// ["red", "red"] ["green", "green"] ["blue", "blue"]

上面代码中，entries方法返回的遍历器，同时包括键名和键值，所以每次输出一个数组，它的两个成员完全相等。

Set结构的实例的forEach方法，用于对每个成员执行某种操作，没有返回值。

**let set** = **new** Set([1, 2, 3]);  
**set**.forEach((value, key) => console.log(value \* 2) )  
// 2 4 6

上面代码说明，forEach方法的参数就是一个处理函数。该函数的参数依次为键值、键名、集合本身（上例省略了该参数）。另外，forEach方法还可以有第二个参数，表示绑定的this对象。

**WeakSet**

WeakSet结构与Set类似，也是不重复的值的集合。但是，它与Set有两个区别。

首先，WeakSet的成员只能是对象，而不能是其他类型的值。

其次，WeakSet中的对象都是弱引用，即垃圾回收机制不考虑WeakSet对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于WeakSet之中。这个特点意味着，无法引用WeakSet的成员，因此WeakSet是不可遍历的。

WeakSet是一个构造函数，可以使用new命令，创建WeakSet数据结构。

作为构造函数，WeakSet可以接受一个数组或类似数组的对象作为参数。（实际上，任何具有iterable接口的对象，都可以作为WeakSet的参数。）该数组的所有成员，都会自动成为WeakSet实例对象的成员。

WeakSet结构有以下三个方法。

WeakSet.prototype.add(value)：向WeakSet实例添加一个新成员。

WeakSet.prototype.delete(value)：清除WeakSet实例的指定成员。

WeakSet.prototype.has(value)：返回一个布尔值，表示某个值是否在WeakSet实例之中。

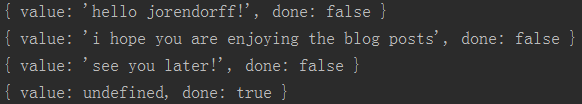
WeakSet没有size属性，也没有遍历方法。

WeakSet不能遍历，是因为成员都是弱引用，随时可能消失，遍历机制无法保存成员的存在，很可能刚刚遍历结束，成员就取不到了。WeakSet的一个用处，是储存DOM节点，而不用担心这些节点从文档移除时，会引发内存泄漏。

Generator

**function**\* quips(name) {  
 **yield** "hello " + name + "!";  
 **yield** "i hope you are enjoying the blog posts";  
 **if** (name.startsWith("X")) {  
 **yield** "it's cool how your name starts with X, " + name;  
 }  
 **yield** "see you later!";  
}  
  
**var** iter = quips("jorendorff");  
console.log(iter.next())  
console.log(iter.next())  
console.log(iter.next())  
console.log(iter.next())

打印结果为：



Generator看上去很像一个函数，这被称为Generator函数，它与我们常见的函数有很多共同点，但还可以看到下面两个差异：

通常的函数以function开始，但Generator函数以function\*开始。

在Generator函数内部，yield是一个关键字，和return有点像。不同点在于，所有函数（包括Generator函数）都只能返回一次，而在Generator函数中可以 yield任意次。yield表达式暂停了Generator函数的执行，然后可以从暂停的地方恢复执行。

常见的函数不能暂停执行，而Generator函数可以，这就是这两者最大的区别。

Generator函数的调用方法与普通函数一样：quips("jorendorff")，但调用一个 Generator函数时并没有立即执行，而是返回了一个Generator对象（上面代码中的iter），这时函数就立即暂停在函数代码的第一行。

每次调用Generator对象的.next()方法时，函数就开始执行，直到遇到下一个 yield 表达式为止。

这就是为什么我们每次调用iter.next()时都会得到一个不同的字符串，这些都是在函数内部通过yield表达式产生的值。

当执行最后一个iter.next()时，就到达了Generator函数的末尾，所以返回结果的.done属性值为true，并且.value属性值为undefined。

**var** events = require('events');  
**var** eventEmitter = **new** events.EventEmitter();  
  
**var** preConnect = **function**(){  
 console.log("连接之前执行！");  
}  
  
**var** connectHandler = **function** connected(){  
 console.log("连接成功！");

//触发data\_received事件  
 eventEmitter.emit('data\_received',"这是返回的数据");

}  
  
//绑定connection事件处理程序  
eventEmitter.on('connection',preConnect);  
eventEmitter.on('connection',connectHandler);  
  
//使用匿名函数绑定data\_received事件  
eventEmitter.on('data\_received',**function**(data){  
 console.log("数据接收成功:" + data)  
})  
  
//触发connection事件  
eventEmitter.emit('connection');  
  
console.log("程序执行完毕！");