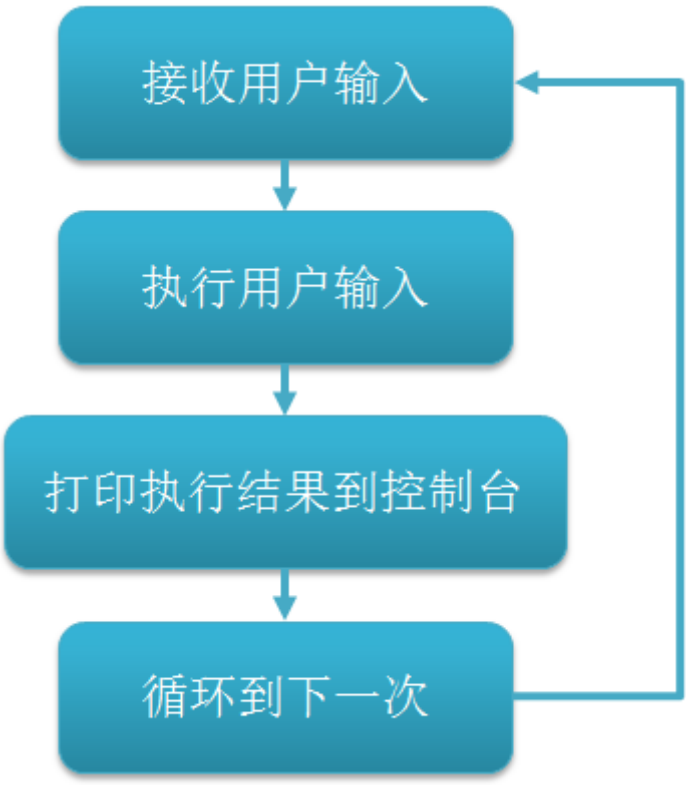
1 - 使用node

基础概念概要：

- Node命令的基本用法
- REPL环境
- 全局对象
- 全局变量
- 全局函数
- 异步操作之回调函数

1-1REPL 环境操作：

+ REPL 全称（Read, Eval, Print, Loop）



+ 进入REPL（命令行输入）
    - node
    - node --use_strict
+ REPL环境中
    - 类似 Chrome Developer Tools → Consoles
    - 特殊变量下划线（_）表示上一个命令的返回结果
    - 通过 .exit 或执行 process.exit() 退出 REPL 交互

1-2全局作用域成员

+全局对象

- global： 类似于客户端 JavaScript 运行环境中的 window
- process： 用于获取当前的 Node 进程信息，一般用于获取环境变量之类的信息
- console： Node 中内置的 console 模块，提供操作控制台的输入输出功能，常见使用方式与客户端类似

+ 全局函数

- setInterval(callback, millisecond)
- clearInterval(timer)
- setTimeout(callback, millisecond)
- clearTimeout(timer)
- Buffer：Class

全局函数	描述
console	**用于提供控制台标准输出**
console.log([data][, ...])	向标准输出流打印字符并以换行符结束。
console.info([data][, ...])	该命令的作用是返回信息性消息，这个命令与console.log差别 余的会显示一个蓝色的惊叹号。
console.warn([data][, ...])	输出警告消息。控制台出现有黄色的惊叹号。
console.dir(obj[, options])	用来对一个对象进行检查（inspect），并以易于阅读和打印的
console.time(label)	输出开始时间，表示计时开始。
console.timeEnd(label)	输出结束时间，表示计时结束。
console.trace(message[, ...])	当前执行的代码在堆栈中的调用路径，这个测试函数运行很有 console.trace 就行了。
console.assert(value[, message][, ...])	用于判断某个表达式或变量是否为真，接收两个参数，第一个 一个参数为false，才会输出第二个参数，否则不会有任何结果
process	**描述当前Node.js进程状态的对象**
stdout	标准输出流。
stdin	标准输入流。
stderr	标准错误流。
argv	argv 属性返回一个数组，由命令行执行脚本时的各个参数组成 本文件名，其余成员是脚本文件的参数。
execPath	返回执行当前脚本的 Node 二进制文件的绝对路径。

execArgv	返回一个数组，成员是命令行下执行脚本时，在Node可执行文
env	返回一个对象，成员为当前 shell 的环境变量
exitCode	进程退出时的代码，如果进程优通过 process.exit() 退出，不需
version	Node 的版本，比如v0.10.18。
versions	一个属性，包含了 node 的版本和依赖.
config	config一个包含用来编译当前 node 执行文件的 javascript 配置 的 "config.gypi" 文件相同。
pid	当前进程的进程号。
title	进程名，默认值为"node"，可以自定义该值。
arch	当前 CPU 的架构：'arm'、'ia32' 或者 'x64'。
platform	运行程序所在的平台系统 'darwin', 'freebsd', 'linux', 'sunos'
mainModule	require.main 的备选方法。不同点，如果主模块在运行时改变 以认为，这两者引用了同一个模块。
exit()	当进程准备退出时触发。
beforeExit()	当 node 清空事件循环，并且没有其他安排时触发这个事件
uncaughtException()	当一个异常冒泡回到事件循环，触发这个事件。如果给异常添加 退出）就不会发生。
Signal 事件()	当进程接收到信号时就触发。信号列表详见标准的 POSIX 信号
abort()	这将导致 node 触发 abort 事件。会让 node 退出并生成一个
chdir(directory)	改变当前工作进程的目录，如果操作失败抛出异常。
cwd()	返回当前进程的工作目录
exit([code])	使用指定的 code 结束进程。如果忽略，将会使用 code 0。
getgid()	获取进程的群组标识（参见 getgid(2)）。获取到得时群组的数 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和
setgid(id)	设置进程的群组标识（参见 setgid(2)）。可以接收数字 ID 或 为数字 ID 。 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和
getgroups()	返回进程的群组 iD 数组。POSIX 系统没有保证一定有，但是 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和
setgroups(groups)	设置进程的群组 ID。这是授权操作，所有你需要有 root 权限， 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和
initgroups(user, extra_group)	读取 /etc/group ，并初始化群组访问列表，使用成员所在的用

	权限，或者有 CAP_SETGID 能力。 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和
kill(pid[, signal])	发送信号给进程. pid 是进程id，并且 signal 是发送的信号的字 'SIGHUP'。如果忽略，信号会是 'SIGTERM'。
memoryUsage()	memoryUsage()返回一个对象，描述了 Node 进程所用的内存
nextTick(callback)	一旦当前事件循环结束，调用回到函数。
umask([mask])	设置或读取进程文件的掩码。子进程从父进程继承掩码。如果指 前掩码。
uptime()	返回 Node 已经运行的秒数。
hrtime()	返回当前进程的高分辨时间，形式为 [seconds, nanoseconds 期无关，因此不受时钟漂移的影响。主要用途是可以通过精确的 你可以将之前的结果传递给当前的 process.hrtime()，会返回

```
process.stdout.write('\033[2J');
```

清空控制台：

```
process.stdout.write('\033[0f');
process.stdout.getWindowSize();
```

V8 对 ES6支持情况分为三个级别：根本不支持，直接支持，严格模式支持

```
console.time(string);
   //code
   //计算这段代码执行的时间
console.timeEnd(stirng);

process.argv    //启动Node.js进程时的命令行参数
process.argit    //接受用户输入的内容
process.stdin    //标准输入
process.stdout    //标准输出
process.stdout.write('string');    //向屏幕输出提示信息
process.env    //返回一个包含用户环境信息的对象
process.env.PATH //获取环境变量
process.platform    //获取node.js进程运行的操作系统平台
```

//fs是FileSyStem模块

fs.stat(path, callback) //查询文件信息
fs.unlink(path, callback) //删除文件
fs.writeFile(path, callback) //创建文件

1-3Node调试工具：
- console.log()
- Node原生调试
+ 第三方模块提供的调试工具
    - $ npm install node-inspector -g
    - $ npm install devtool -g
- 开发工具的调试

$ node debug myscript.js

1-4异步操作：
- 异步操作回调
+ 回调函数设计
    - 回调函数一定参作为数的最后一个参数出现
    - 回调函数的第一个参数默认接受错误信息，第二个参数才是真正的回调数据

附加：
采用异步的方式，无法通过 try catch 捕获异常， 所以
错误优先的回调函数 第一个参数为上一步的错误信息

异步回调的问题
- 回调黑洞
- 不容易阅读
- 不容易调试
- 不容易维护

fs.readFile('path', 'utf-8', (error, data)=> {

```
    if(error) throw error;
    console.log(data)
})
```

1-5进程与线程

进程	线程
每一个正在运行的应用程序都称之为进程 每一个应用程序运行都至少有一个进程 提供给应用程序一个运行的环境 操作系统为应用程序分配资源的一个单位	执行应用程序中的代码 在一个进程内部，可以有很多的线程 在一个线程内部，同时只可以干一件事 而且传统的开发方式大部分都是 I/O 阻塞的 所以需要多线程来更好的利用硬件资源 给人带来一种错觉：线程越多越好

多线程没落：

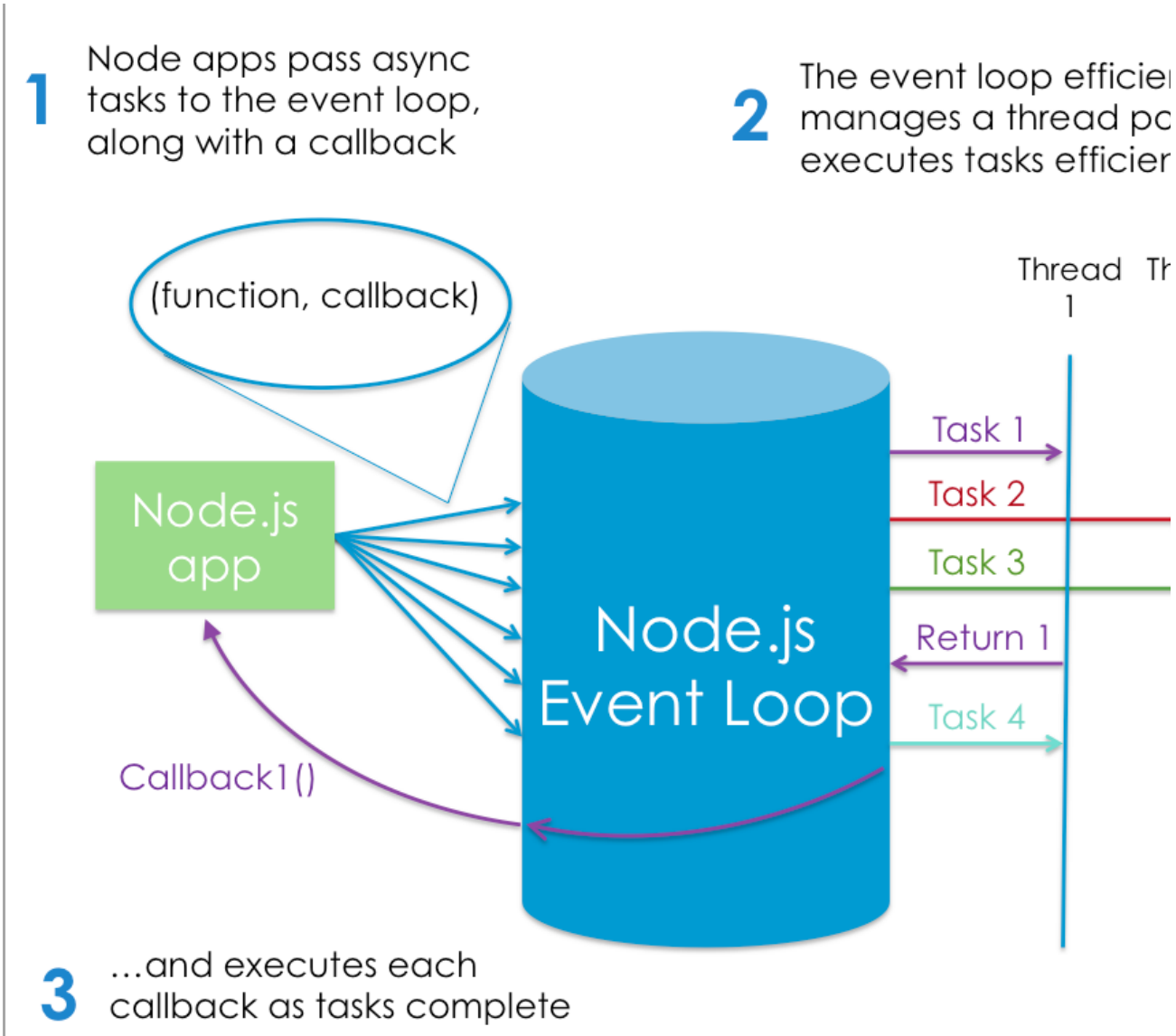
多线程都是假的，因为只一个 CPU（单核）

线程之间共享某些数据，同步某个状态都很麻烦

更致命的是： 创建线程耗费 ，线程数量有限

CPU 在不同线程之间转换，需要上下文转换，这个转换非常耗时

1-6事件驱动和非阻塞I/O机制



1：Node应用程序需要去完成一个块操作，传送一个伴随回调函数的异步任务给事件循环，然后继续执行
2：事件循环跟踪异步操作，当完成时执行异步操作所传送的回调函数
3：异步操作完成，事件循环返回回调函数的结果给应用程序

总结：

Node中将所有的阻塞操作交给了内部实现的线程池

Node本身主线程主要就是不断的往返调度