黑马程序员
www.itheima.com

Node 服务端编程

讲师：汪磊

课程大纲

- Node 是什么？带来了什么？能用来做什么？
- 如何搭建一个 Node 的开发和生产环境？
- 快速上手使用 Node ，操作入门
- 常见的核心模块和对象的使用
- 数据库的概念和通过 Node 操作数据库
- 服务器端开发的基本概念以及如何通过 Node 实现
- 常用的快速开发 Node 应用程序框架
- 综合项目实战

快速入门

了解 Node 中全局作用域及全局对象和函数

基础概念概要

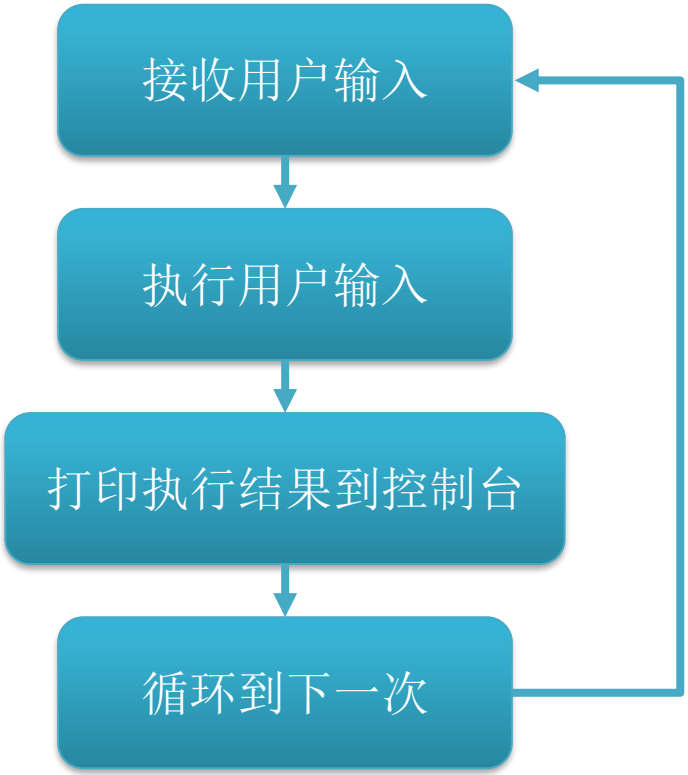
- Node 命令的基本用法
- REPL 环境
- 全局对象
- 全局变量
- 全局函数
- 异步操作之回调函数

Node 命令基本用法

- 进入 REPL 环境：
  - `$ node`
- 执行脚本字符串：
  - `$ node -e 'console.log("Hello World")'`
- 运行脚本文件：
  - `$ node index.js`
  - `$ node path/index.js`
  - `$ node path/index`
- 查看帮助：
  - `$ node --help`

REPL 环境

- REPL 全称（Read, Eval, Print, Loop）



REPL 环境操作

- 进入 REPL：
  - `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

全局作用域成员

了解 Node 的最基本成员以及核心概念

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
  - 用于操作二进制数据
  - 以后介绍

练习

- 绘制字符画动画
  - http://www.degraeve.com/img2txt.php 将图片转换为字符画
- 清空控制台
  - `process.stdout.write('\033[2J');`
  - `process.stdout.write('\033[0f');`
  - 或
  - `process.stdout.getWindowSize();`

Node 调试

任何一个平台的开发都离不开调试

只会写 Code 的 Coder 都不是好 Coder

Node 调试

- 最方便也是最简单的：console.log()
- Node 原生的调试
  - https://nodejs.org/api/debugger.html
- 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
  - 推荐 Visual Studio Code
  - WebStorm

学习目标

- 了解 REPL 环境
- 全局对象基本使用
- 学会调试 Node 程序

练习

- 连续按两次 ^C 推出
  - 放在用户操作不当
- 人机交互
  - 接收用户输入
  - 用户输入无状态

异步编程

异步操作

- Node 采用 Chrome V8 引擎处理 JavaScript 脚本，V8 最大特点就是单线程运行，一次只能运行一个任务。
- Node 大量采用异步操作（asynchronous operation），即任务不是马上执行，而是插在任务队列的尾部，等到前面的任务运行完后再执行。
- 提高代码的响应能力。

回调函数的设计

- 对于一个函数如果需要定义回调函数：
  - 回调函数一定作为参数的最后一个参数出现：
    - `function foo1(name, age, callback) { }`
    - `function foo2(value, callback1, callback2) { }`
  - 回调函数的第一个参数默认接收错误信息，第二个参数才是真正的回调数据（便于外界获取调用的错误情况）：

```
foo1('赵小黑', 19, function(error, data) {
  if(error)  throw error;
  console.log(data);
});
```

- Node 统一约定

强调 错误优先

- 因为之后的操作大多数都是异步的方式，无法通过 try catch 捕获异常，SO
- 错误优先的回调函数
  - 第一个参数为上一步的错误信息

什么是异步

- 现实生活中：
- 程序世界中：
  - setTimeout()
  - $.ajax()

异步操作回调

- 由于系统永远不知道用户什么时候会输入内容，所以代码不能永远停在一个地方；
- Node 中的操作方式就是以异步回调的方式解决无状态的问题；

异步回调的问题

```
do1(function() {
  do2(function() {
    do3(function() {
      do4(function() {
        do5(function() {
          do6()
        });
      });
    });
  });
});
```

异步回调的问题

- 相比较于传统的代码：
  - 异步事件驱动的代码不容易阅读
  - 不容易调试
  - 不容易维护

非阻塞 I/O

Node 的核心特性

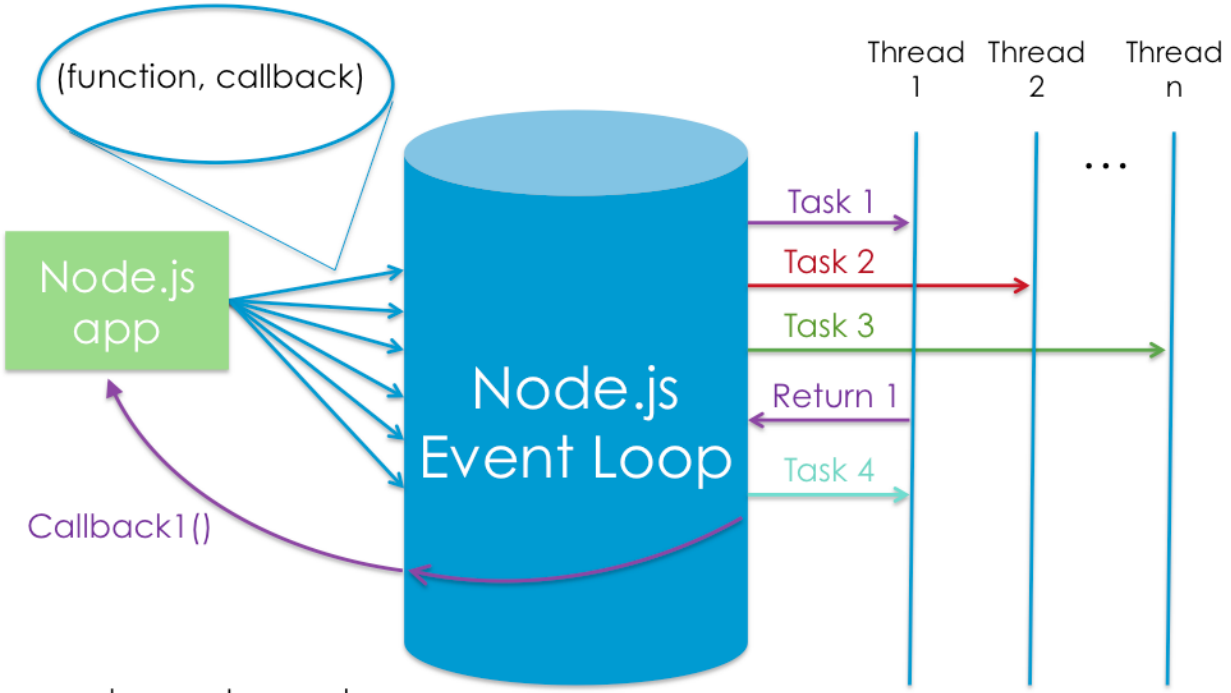
什么是 I/O

- I/O：【input/output】
- 可以理解为从输入到输出之间的转化过程
- 例如：
  - 敲击键盘（输入）看到编辑器中多出字符（输出）
  - 移动鼠标（输入）看到光标移动（输出）
  - 小霸王
  - cmd 中 执行命令
    - dir

事件驱动和非阻塞机制

1 Node apps pass async tasks to the event loop, along with a callback

2 The event loop efficiently manages a thread pool and executes tasks efficiently…



3 …and executes each callback as tasks complete

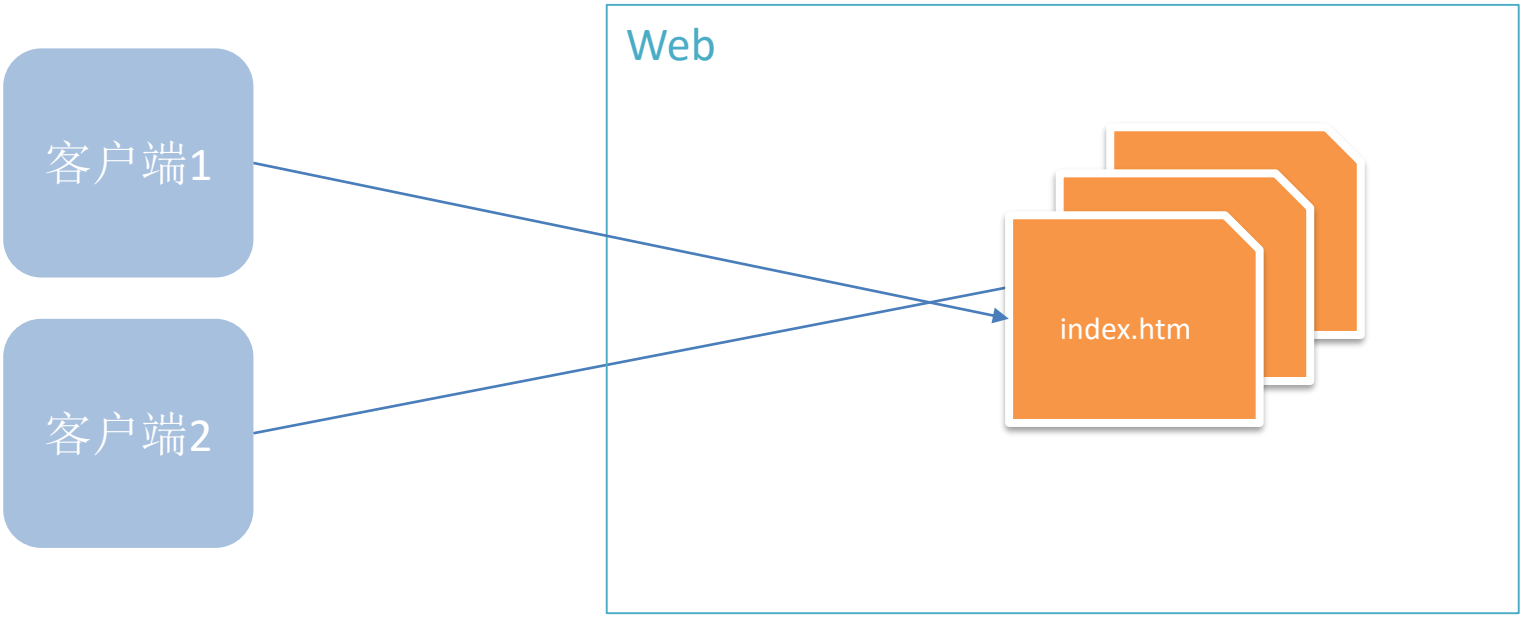
太抽象，要栗子

非阻塞的必要性

- 花店的栗子
- 以及：

Web 开发中的非阻塞

- 早些时候，当 Web 还是一个个的页面的时候：

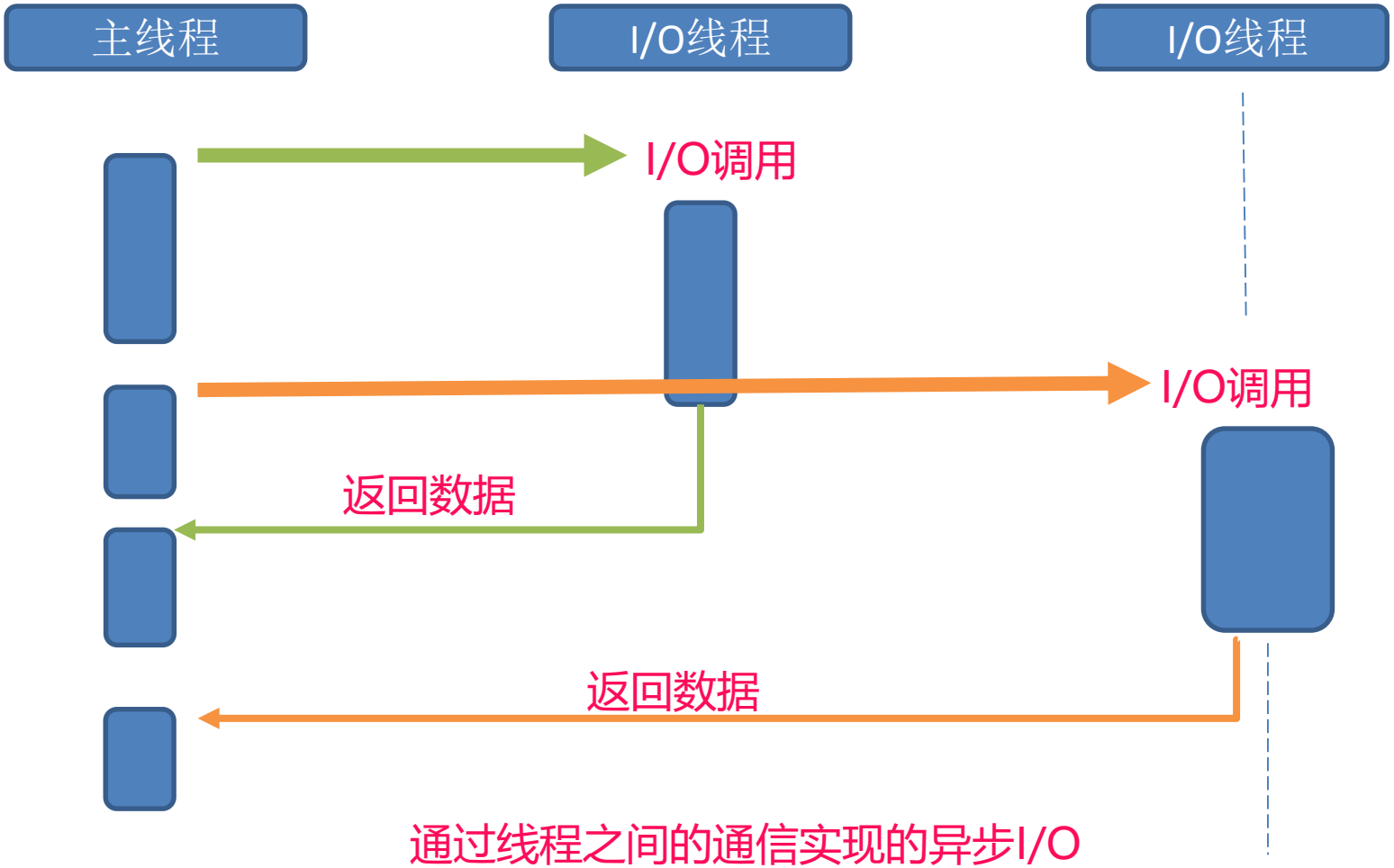


- 这种时候是没有必要的

Web 开发中的非阻塞

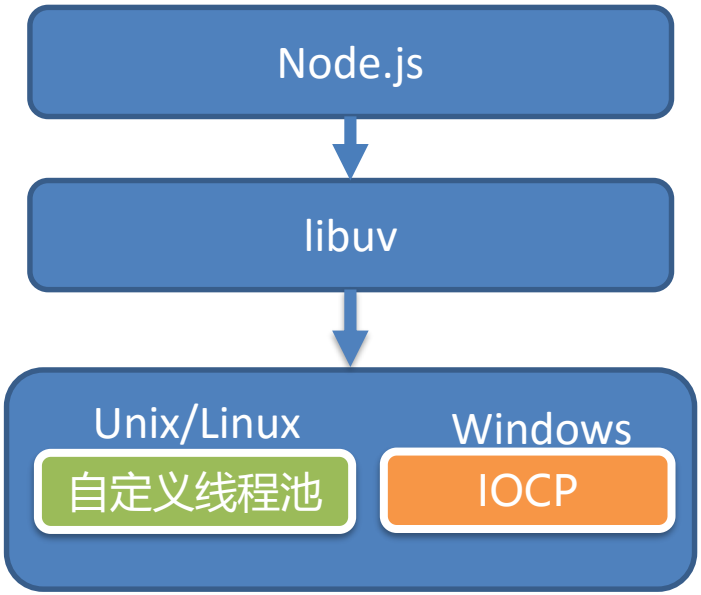
- 再看看当下：
  - 各种业务需求
  - 各种设备访问
- 一切为了高效

Node 中的异步 I / O



平台实现差异

- 由于 Windows 和 *nix 平台的差异，Node 提供了 libuv 作为抽象封装层，保证上层的 Node 与下层的自定义线程池及 IOCP 之间各自独立



如何提高一个人的工作效率

- 我曾经日过：

非阻塞的优势

- 提高代码的响应效率
- 充分利用单核 CPU 的优势
- 改善 I/O 的不可预测带来的问题
- 如何提高一个人的工作效率

什么进程

- 每一个 正在运行 的应用程序都称之为进程。
- 每一个应用程序都至少有一个进程
- 进程是用来给应用程序提供一个运行的环境
- 进程是操作系统为应用程序分配资源的一个单位

什么是线程

- 用来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

什么原因让多线程没落

- 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：

学习目标

- 学会定义错误优先的回调函数
- `function callback(error, data) {`
- `  console.log(data);`
- `}`
- 了解什么是事件驱动和非阻塞 I/O
- 理解 Node 为什么那么快

模块化结构

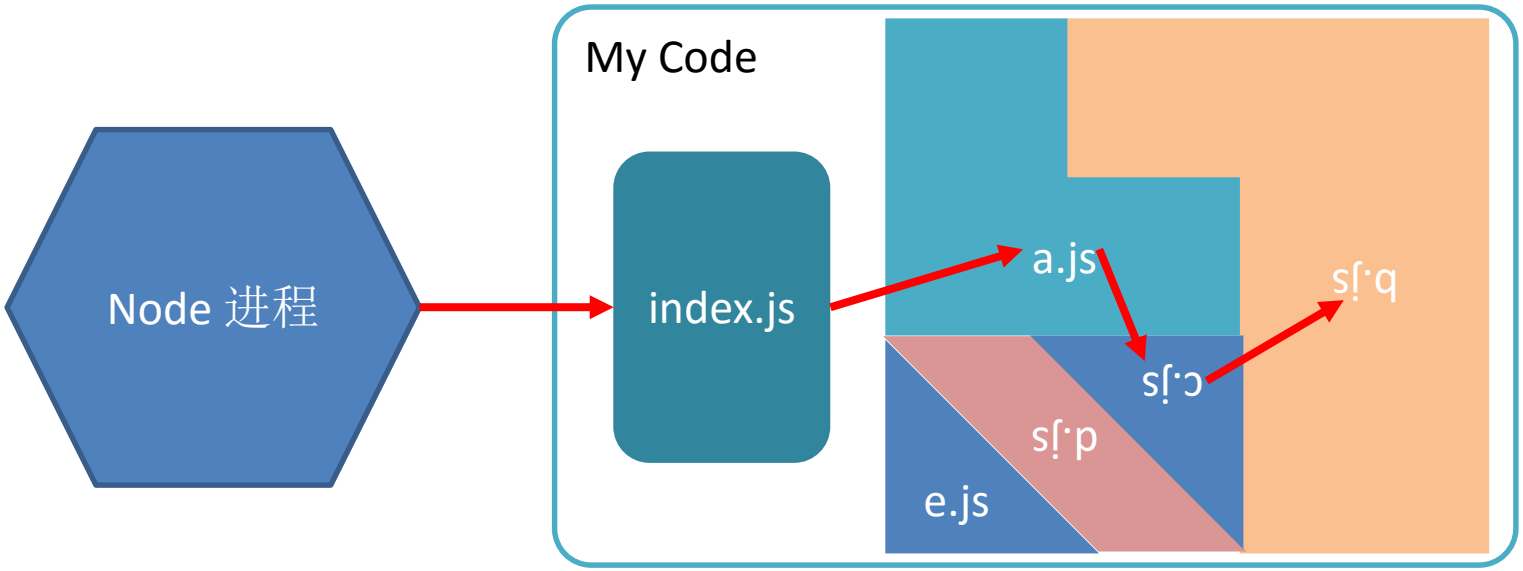
Node 实现 CommonJS 规范，所以可以使用模块化的方式组织代码结构

练习

- 计算器
  - `$ node calculator.js 111 + 1111`
  - `1222`
- 抽象模块

模块化代码结构

- Node 采用的模块化结构是按照 CommonJS 规范
- 模块与文件是一一对应关系，即加载一个模块，实际上就是加载对应的一个模块文件。



CommonJS 规范概述

- CommonJS 就是一套约定标准，不是技术；
- 用于约定我们的代码应该是怎样的一种结构；
- http://wiki.commonjs.org/wiki/CommonJS

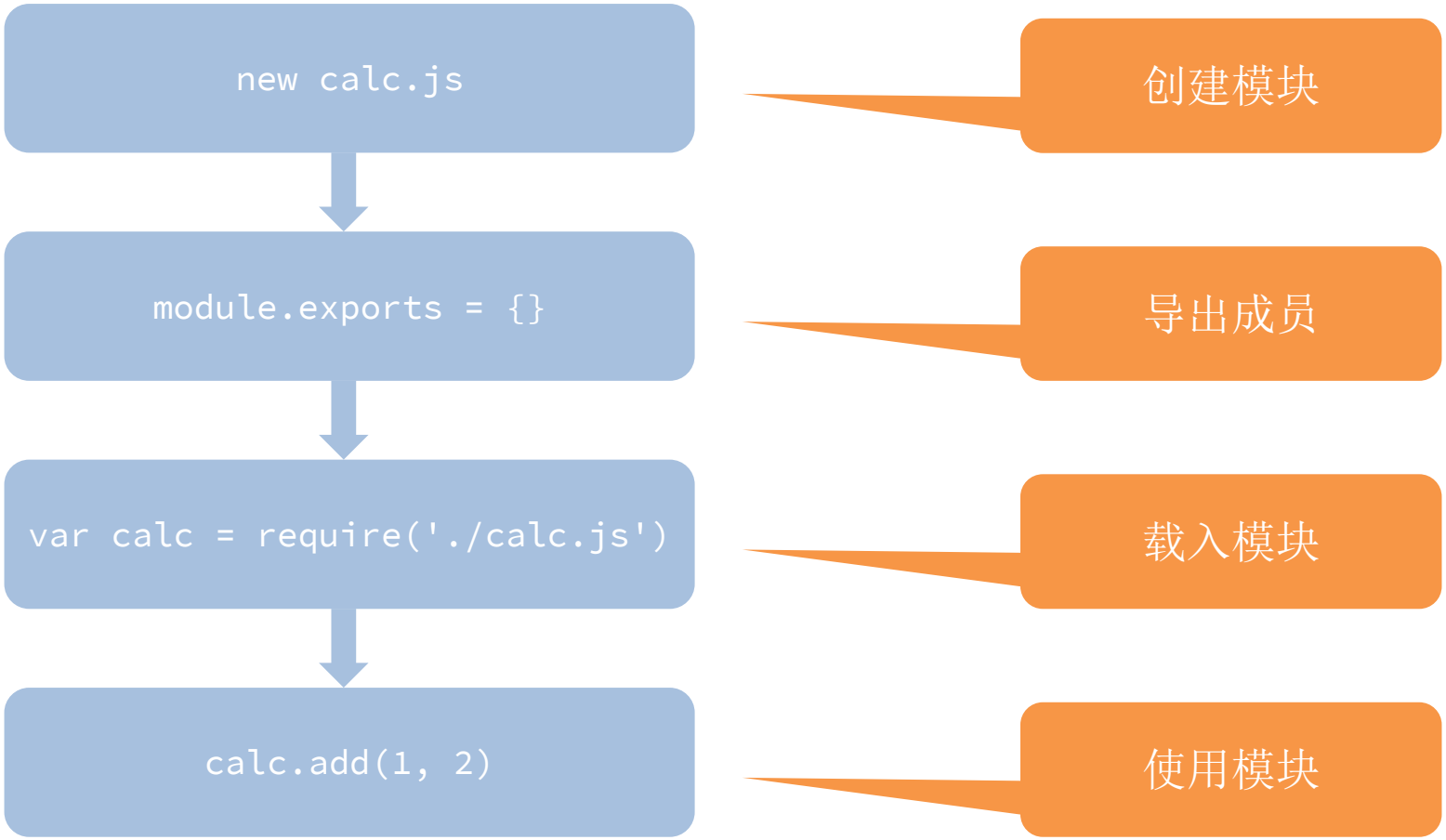
CommonJS 模块的特点

- 所有代码都运行在模块作用域，不会污染全局作用域。
- 模块可以多次加载，但是只会在第一次加载时运行一次，然后运行结果就被缓存了，以后再加载，就直接读取缓存结果。要想让模块再次运行，必须清除缓存。
- 模块加载的顺序，按照其在代码中出现的顺序。

模块的分类

- 文件模块
  - 就是我们自己写的功能模块文件
- 核心模块
  - Node 平台自带的一套基本的功能模块，也有人称之为 Node 平台的 API
- 第三方模块
  - 社区或第三方个人开发好的功能模块，可以直接拿回来用

模块化开发的流程


new calc.js
module.exports = {}
var calc = require('./calc.js')
calc.add(1, 2)
创建模块
导出成员
载入模块
使用模块

定义模块

模块内全局环境（伪）

- 我们在之后的文件操作中必须使用绝对路径
- `__dirname`
  - 用于获取当前文件所在目录的完整路径；
  - 在 REPL 环境无效；
- `__filename`
  - 用来获取当前文件的完整路径；
  - 在 REPL 环境同样无效；

模块内全局环境（伪）

- module
  - 模块对象
- exports
  - 映射到 module.exports 的别名
- require()
  - require.cache
  - require.extensions
  - require.main
  - require.resolve()

module 对象

- Node 内部提供一个 Module 构建函数。所有模块都是 Module 的实例，属性如下：
  - module.id 模块的识别符，通常是带有绝对路径的模块文件名。
  - module.filename 模块定义的文件的绝对路径。
  - module.loaded 返回一个布尔值，表示模块是否已经完成加载。
  - module.parent 返回一个对象，表示调用该模块的模块。
  - module.children 返回一个数组，表示该模块要用到的其他模块。
  - module.exports 表示模块对外输出的值。
- 载入一个模块就是构建一个 Module 实例。

模块的定义

- 一个新的 JS 文件就是一个模块；
- 一个合格的模块应该是有导出成员的，否则模块就失去了定义的价值；
- 模块内部是一个独立（封闭）的作用域（模块与模块之间不会冲突）；
- 模块之间必须通过导出或导入的方式协同；
- 导出方式：
  - `exports.name = value;`
  - `module.exports = { name: value };`

模块的定义

- 还有必要写自执行函数吗？

模块的定义

- module.exports 和 exports
- module.exports 是用于为模块导出成员的接口
- exports 是指向 module.exports 的别名，相当于在模块开始的时候执行：
  - `var exports = module.exports;`
- 一旦为 module.exports 赋值，就会切断之前两者的相关性；
- 最终模块的导出成员以 module.exports 为准

载入模块

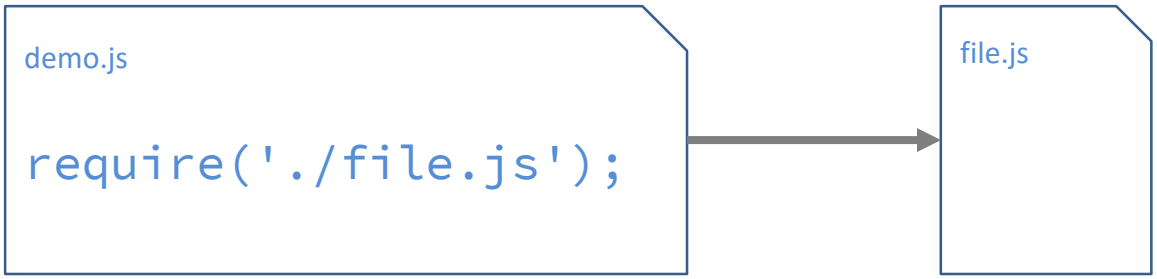
require 函数

require 简介

- Node 使用 CommonJS 模块规范，内置的 require 函数用于加载模块文件。
- require 的基本功能是，读入并执行一个 JavaScript 文件，然后返回该模块的 exports 对象。
- 如果没有发现指定模块，会报错。

模块的加载机制

- id: 路径的情况就是直接以相对路径的方式找文件



require 扩展名

- require 加载文件时可以省略扩展名：
  - `require('./module');`
  - `// 此时文件按 JS 文件执行`
  - `require('./module.js');`
  - `// 此时文件按 JSON 文件解析`
  - `require('./module.json');`
  - `// 此时文件预编译好的 C++ 模块执行`
  - `require('./module.node');`

require 加载文件规则

- 通过 ./ 或 ../ 开头：则按照相对路径从当前文件所在文件夹开始寻找模块；
  - `require('../file.js');` => 上级目录下找 file.js 文件
- 通过 / 开头：则以系统根目录开始寻找模块；
  - `require('/Users/iceStone/Documents/file.js');` => 以绝对路径的方式找

require 加载文件规则

- 如果参数字符串不以“./“或”/“开头，则表示加载的是一个默认提供的核心模块（位于 Node 的系统安装目录中）：
  - `require('fs');` => 加载核心模块中的文件系统模块
- 或者从当前目录向上搜索 node_modules 目录中的文件：
  - `require('my_module');` => 各级 node_modules 文件夹中搜索 my_module.js 文件；

require 加载目录规则

- 如果 require 传入的是一个目录的路径，会自动查看该目录的 package.json 文件，然后加载 main 字段指定的入口文件
- 如果package.json文件没有main字段，或者根本就没有package.json文件，则默认找目录下的 index.js 文件作为模块：
  - `require('./calcuator');` => 当前目录下找 calculator 目录中的 index.js 文件

模块的缓存

- 第一次加载某个模块时，Node 会缓存该模块。以后再加载该模块，就直接从缓存取出该模块的 `module.exports` 属性（不会再次执行该模块）
- 如果需要多次执行模块中的代码，一般可以让模块暴露行为（函数）
- 模块的缓存可以通过 require.cache 拿到，同样也可以删除

require 的实现机制

- 将传入的模块 ID 通过加载规则找到对应的模块文件
- 读取这个文件里面的代码
- 通过拼接的方式为该段代码构建私有空间
- 执行该代码
- 拿到 module.exports 返回

学习目标

- 如何定义和导入模块
- 使用模块化的方式组织代码结构
- 载入模块的规则
- 了解模块的加载机制