第1章 Node简介

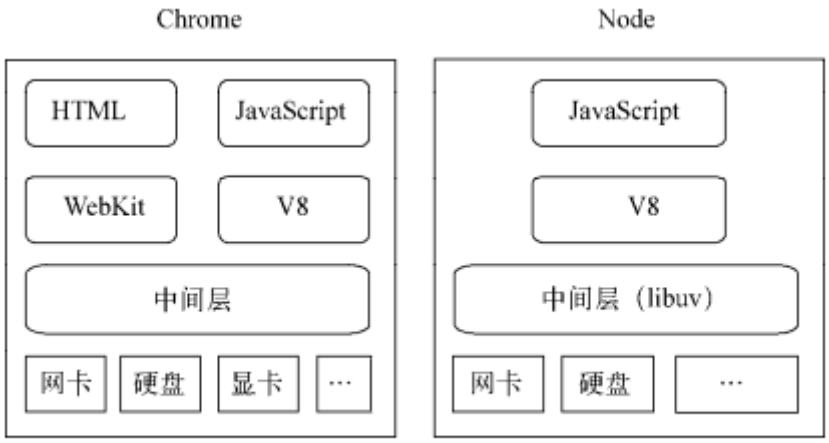


图1-1　Chrome浏览器和Node的组件构成

Node 相较于其他服务端语言的特点

- 异步I/O：解决了单线程上CPU与I/O之间阻塞无法重叠利用的问题（其他语言很少有异步）
  - ！！！**弥补了单线程无法利用多核CPU的缺点**（提供了类似Web Worker的子进程）
- 事件与回调函数（在Javascript中，函数是一等公民，将函数作为对象传参调用是一大特点）
- 单线程（对比多线程）【JavaScript是单线程的，但是Node是多线程的，只是IO线程使用的CPU较少】
  - Node是单线程吗？**Node的单线程仅仅是Javascript执行在单线程中**，而无论是Linux还是Windows，内部完成I/O任务的另有线程池
  - 优点：不必时时在意状态的同步问题，没有死锁的存在，没有线程上下文交换所带来的性能消耗
  - 缺点
    - 无法利用多核CPU
    - 错误会引起整个应用退出
    - 大量计算占用CPU导致无法继续调用异步I/0【浏览器中JS长时间执行会阻塞UI的渲染、响应】

> 浏览器中Web Worker：创建工作线程来进行计算，解决Javascript大计算阻塞UI渲染的问题
>
> - 用消息传递来传递运行结果 => 不能访问到主线程中的UI
>
> Node中的子进程 child_process：将计算分发到子进程，通过进程间的事件消息来传递结果
>
> - （与WebWorkers思路相同）
> - Master-Worker 管理各个工作进程

- 跨平台

第2章 模块机制

2.1 CommonJS（用于浏览器前端）

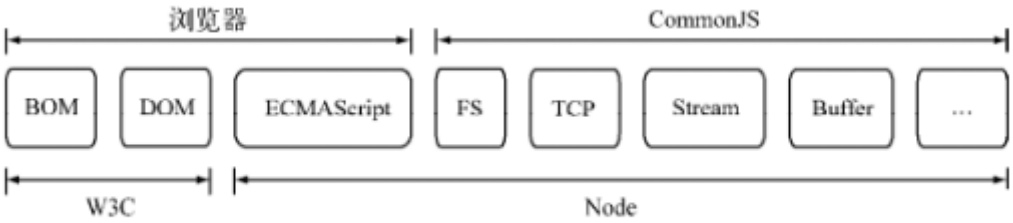


图2-2　Node与浏览器以及W3C组织、CommonJS组织、ECMAScript之间的关系

2.2 Node 的模块实现

引入模块：路径分析、文件定位、编译执行

两类模块

- 核心模块：编译成二进制执行文件，直接内存加载（第一步）
- 文件模块：**运行时动态加载 【边村】**

【会缓存编译、执行后的对象】

模块加载快慢：缓存加载、核心模块加载、文件模块加载、自定义模块加载

- **核心模块**，eg：http
- 以. 、.. 、 /开始的标识符，都被当作**文件模块**处理
- **自定义模块**：非核心、非路径形式，可能是文件/包，查找费时

1、路径分析：类似JS原型链的查找，向上逐级递归查找node_modules，直到有合适的文件

2、文件定位 = 文件扩展名 、目录分析和包

- 目录：有无package.json、有无main属性指定的文件名（扩展名分析），否则找index
- 性能优化：在引入时，尽可能不要直接用目录，可以是 目录/index.js

3、编译执行：将编译成功的模块路径作为索引缓存起来，提高二次引入性能

- Javascript模块的编译：对JS文件内容左头尾包装 (function(){})，作用域隔离，用vm原生模块的runInThisContext()执行

性能优化1：

在模块中，可以不加扩展名，Node会按照.js、.json、.node的次序补充扩展名，依次尝试

- 需要调用fs模块同步阻塞式地判断文件是否存在（单线程，有性能消耗）
- 优化点
  - .node、.json文件，传递给require()的标识符中带上扩展名，加快速度
  - 同步配合缓存，直接省略引入模块的三步骤，加载缓存中编译后的对象

第3章 异步I/O

为什么需要 异步I/O

- 用户体验：后端获取速度慢也会影响用户体验，分布式数据需要异步IO
- 资源浪费

> 为什么前端需要异步编程

> - 前端编程算是一种GUI编程，里面有很多的Ajax和事件处理，都是典型的异步应用场景

多线程

- 优点：可以并行，在多核CPU上能够有效提升CPU
- 缺点：创建线程、执行期间线程上下文切换开销较大，存在死锁、状态同步等问题

单线程

- 优点：顺序执行，符合编程人员思维方式（主流、易于表达）
- 缺点：通常计算机的I/O与CPU计算之间是可以并行进行的，但同步中IO会因为阻塞让CPU等待，硬件资源浪费

Node的做法：

Node在两者之间给出了它的方案：利用单线程，远离多线程死锁、状态同步等问题；利用异步I/O，让单线程远离阻塞，以更好地使用CPU。

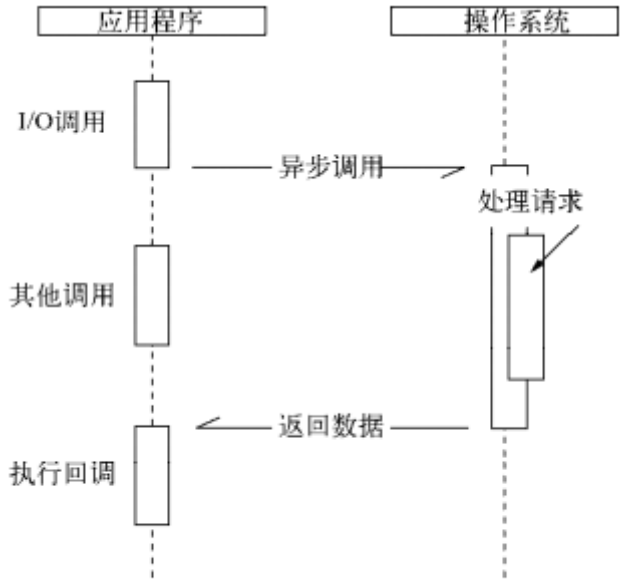


图3-1　异步I/O的调用示意图

操作系统内核对于IO只有两种方式：阻塞/非阻塞

阻塞

- 调用阻塞IO时，应用程序需要等待IO完成才返回 =》造成CPU阻塞

非阻塞

- 非阻塞IO调用之后会立即返回（不带数据直接返回），之后通过文件描述符再次读取数据
- 再次读取，重新调用IO操作来确认是否完成（**轮询**），轮询需要CPU处理状态判断，也是一种资源浪费

Node的异步IO

完成整个异步IO环节：**事件循环、观察者、请求对象、IO线程池等**

- 进程启动，Node创建事件循环，每执行一次循环：查看观察者处是否有事件需要处理；若有则回调，继续循环；若不再有事件处理了则退出进程

- 观察者：告诉Node是否有需要处理的事件，通常一个观察者可能会有多个事件
  - 事件的生产者：异步IO、网络请求
  - 事件的消费者：事件循环从观察者那里取出事件并处理

事件循环

在进程启动时，Node便会创建一个类似于while(true)的循环，每执行一次循环体的过程我们称为Tick。每个Tick的过程就是查看是否有事件待处理，如果有，就取出事件及其相关的回调函数。如果存在关联的回调函数，就执行它们。然后进入下个循环，如果不再有事件处理，就退出进程。流程图如图3-11所示。

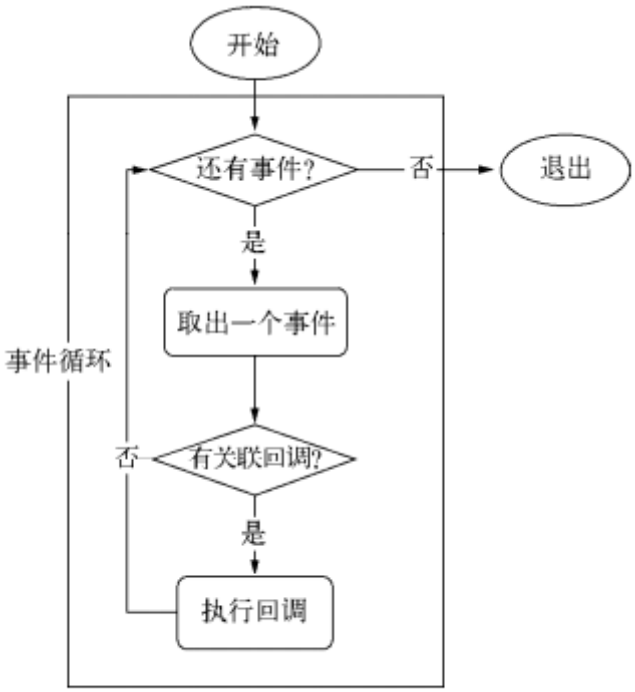
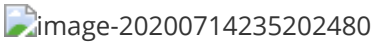


图3-11　Tick流程图

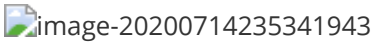
请求对象

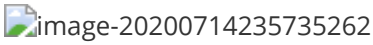
- **组装好请求对象，送入IO线程池**（JavaScript调用返回，不阻塞）
- 普通的回调函数一般由开发者自己调用，而Node中的 异步IO回调则不是



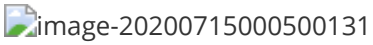
请求对象：请求对象是从JavaScript发起调用，到内核执行完IO操作的过程中产生的中间产物（保存着所有状态，包括送入线程池等待执行以及IO操作完毕后的回调处理）

例子：fs.open()，根据指定路径、参数打开一个文件，得到文件描述符





执行回调



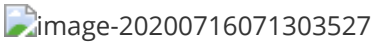
非IO的异步API

SetTimeout()

类似异步IO但无IO线程池，定时器会被插入定时器观察者内部的红黑树中。每次Tick执行时，从红黑树中迭代取出定时器对象，检查是否超时，超时则回调

定时器的问题

- 不精准（执行开始时间也受事件循环影响）

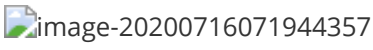


Process.nextTick

> 需要异步=》SetTimeout(fn, 0) 需要动用红黑树、创建定时器对象、迭代，浪费性能【操作时间复杂度O(lgn)】

Process.nextTick 只会将回调函数放入队列中，下一次Tick时取出执行【操作时间复杂度O(1）】

SetImediate()



具体表现：

Process.nextTick把回调函数保存在数组中，每次全部执行完，而setImmediate()的结果则保存在链表中，每次只执行一个（宏任务、微任务）

目的：让事件循环更快执行完（防止GPU占用过多阻塞IO）

Process.nextTick优先级高

事件驱动与高性能服务器

事件驱动的实质：事件触发+主循环

Node服务器的优点：通过事件驱动的方式处理请求，无需为请求创建额外线程（线程创建、销毁有开销，线程少，OS调度快，上下文切换代价低）

Nginx也是事件驱动，不是多线程

异步编程

Node的最大特性：基于事件驱动的非阻塞IO模型（CPU与IO并不相互等待，更好地利用资源）

为什么Node是单线程的：需要处理很多异步事件

难点

异步编程解决方案

事件发布/订阅模式

- events：on/emit
- hook：通过钩子导出内部数据、状态给调用者

优点：利用高阶函数，可以随意添加回调，增删、隔离业务逻辑，保持业务逻辑单元的职责单一

特点：执行流程需要被预先设定，即使是分支也需要

Promise/Deffered模式

链式调用的优点：在Node里面网络库是完全异步的，无法在编程层面实现像其他语言一样的同步调用， 用Promise的链式调用可以避免回调地狱（嵌套）

```
// 回调地狱
obj.api1(function (value1) {
    obj.api2(value1, function (value2) {
        obj.api3(value2, function (value3) {
            obj.api4(value3, function (value4) {
                callback(value4);
            });
        });
    });
});
```

```
// 事件机制处理（代码量剧增）
var emitter = new event.Emitter();
emitter.on("step1", function () {
    obj.api1(function (value1) {
        emitter.emit("step2", value1);
    });
});
emitter.on("step2", function (value1) {
    obj.api2(value1, function (value2) {
        emitter.emit("step3", value2);
    });
});
emitter.on("step3", function (value2) {
    obj.api3(value2, function (value3) {
        emitter.emit("step4", value3);
    });
});
emitter.on("step4", function (value3) {
    obj.api4(value3, function (value4) {
        callback(value4);
    });
});
```

```
promise()
    .then(obj.api1)
    .then(obj.api2)
    .then(obj.api3)
    .then(obj.api4)
    .then(function (value4) {
    // Do something with value4
    }, function (error) {
    // Handle any error from step1 through step4
    })
.done();
```