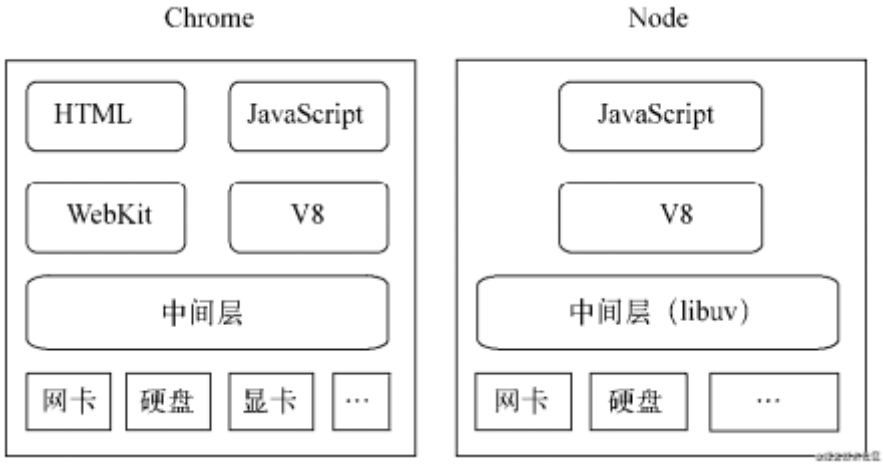
一、Node简介

为什么会选择`JavaScript`：考虑到**高性能、事驱动、没有历史包袱**这3个主要因素，JavaScript成为了Node的实现语言。

1.1 Node给JavaScript 带来的意义

`Node`不处理`UI`，但用与浏览器相同的机制和原理运行。`Node`打破了过去`JavaScript`只能在浏览器中运行的局面。前后端编程环境统一，可以大大降低前后端转换需要的上下文交换代价。



1.2 Node的特点

1.2.1 异步IO

在`Node`中，绝大多数的操作都以异步的方式进行调用。

下面的两个文件读取任务的耗时取决于最慢的那个文件读取的耗时

```
fs.readFile('/path1', (err, file) => {
  console.log('读取文件1完成');
})
fs.readFile('/path2', (err, file) => {
  console.log('读取文件2完成')
})
```

而对于同步`I/O`而言，它们的耗时是两个任务的耗时之和。

1.2.2 事件与回调函数

```
var http = require('http')
var querystring = require('querystring')

// 侦听服务器的request事件
http.createServer((req, res) => {
  var postData = ''
  req.setEncoding('utf8')
  // 侦听请求的data事件
  req.on('data', (chunk) => {
```

```
    postData += chunk;
  });
  // 侦听请求的end事件
  req.on('end', () => {
    res.end(postData);
  })
}).listen(8080)
console.log('服务器启动完成')
```

与其他的Web后端编程语言相比，Node除了异步和事件外，回调函数是一大特色。纵观下来，回调函数也是最好的接受异步调用返回数据的方式。

1.2.3单线程

`Node`保持了`JavaScript`在浏览器中单线程的特点。而且在`Node`中，`JavaScript`与其他线程是无法共享任何状态的。单线程最大的好处是不用像多线程编程那样处处在意状态的同步问题，这里没有死锁的存在，也没有线程上下文交换所带来的的性能上的开销。

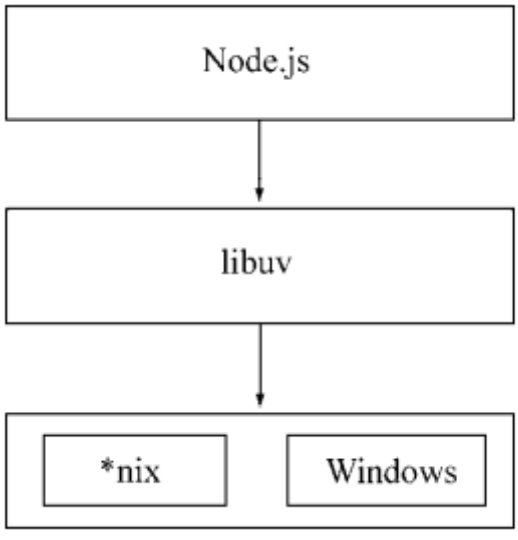
弱点

- 无法利用多核`CPU`
- 错误会引起整个应用的退出，应用的健壮性值得考虑
- 大量计算占用`CPU`导致无法继续调用异步`I/O`。在Node中，长时间的CPU占用也会导致后续的异步`I/O`发不出调用，已完成的异步I/O的回调函数也会得不到及时执行。

优化策略

在浏览器中，`Web Workers`能够创建工作线程来进行计算，以解决`JavaScript`大计算阻塞`UI`渲染的问题。`Node`**采用了与**`Web Workers`**相同的思路来解决单线程中大计算量的问题：**`child_process`**。子进程的出现，意味着Node可以从容地应对单线程在健壮性和无法利用多核**`CPU`**方面的问题。通过将计算分发到各个子进程，可以将大量计算分解掉，然后再通过进程之间的事件消息来传递结果，可以很好地保持应用模型的简单与低依赖。通过Master-Worker的管理方式，也可以很好地管理各个工作进程，以达到更高的健壮性。**

1.2.4 跨平台



兼容`Windows`和`*nix`平台主要得益于Node在架构层面的改动，它在操作系统和Node上层模块系统之间构建了一层平台层架构，即`libuv`。

1.3 Node的应用场景

1.3.1 I/O密集型

说Node擅长I/O密集型的应用场景基本上是没人反对的。Node面向网络且最擅长并行I/O，能够有效地组织起更多的硬件资源，从而提供更多好的服务。

I/O密集的优势主要在于Node利用事件循环的处理能力，而不是启动每一个线程为每一个请求服务，资源占用更少

1.3.2 CPU密集型

在CPU密集的应用场景中，Node是否能胜任呢？实际上，V8的执行效率是十分高的。单以执行效率来做评判，V8的执行效率是毋庸置疑的。

存在的问题

CPU密集型应用给Node带来的挑战主要是：**由于JavaScript单线程的原因，如果有长时间运行的计算（比如大循环），将会导致CPU时间片不能释放，使得后续I/O无法发起。**

但是适当调整和分解大型运算任务为多个小任务，使得运算能够及时释放，不阻塞I/O调用的发起，这样既可同时享受到并行异步I/O的好处，又能充分利用CPU。

关于CPU密集型应用，Node的异步I/O已经解决了在单线程上CPU与I/O之间阻塞无法重叠利用的问题，**I/O阻塞造成的性能浪费远比CPU的影响小。**

对于长时间运行的计算，如果它的耗时超过普通阻塞I/O的耗时，那么应用场景就需要重新评估，因为这类计算比阻塞I/O还影响效率，甚至说就是一个纯计算的场景，根本没有I/O。此类应用场景或许应当采用多线程的方式进行计算。

Node虽然没有提供多线程用于计算支持，但是还是有以下两种方式来充分利用CPU

- Node可以通过编写C/C++扩展的方式更搞笑地利用CPU
- 如果单线程的Node不能满足需求，甚至用了C/C++扩展后还觉得不够，那么通过子进程的方式，将一部分Node进程当作常驻服务进程用于计算，然后利用进程间的消息来传递结果，将计算和I/O分离，这样还能充分利用CPU

1.3.3 与遗留系统和平相处

在Node中，语言层面即可天然并行的特性在这种场景中显得十分有效。对于已有的稳定系统，并非意味着我们要抛弃

1.3.4 分布式应用

阿里巴巴开发了中间层应用NodeFox、ITier，将数据库集群做了划分和映射，查询调用依旧是针对单张表进行SQL查询，中间层分解查询SQL，并行地去多台数据库中获取数据并合并。

1.4 Node的使用者

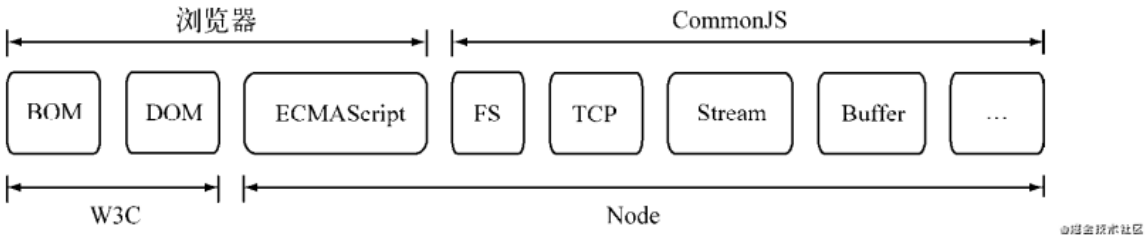
- 前后端编程语言环境统一
- Node带来的高性能I/O用于实时应用
- 并行I/O使得使用者可以更高效地利用分布式环境
- 并行I/O，有效利用稳定接口提升Web渲染能力
- 云计算平台提供Node支持
- 游戏开发领域
- 工具类应用

二、模块机制

2.1 CommonJS规范

`CommonJS`规范为JavaScript制定了一个美好的愿景 —— 希望JavaScript能够在任何地方运行。

Node能以一种比较成熟的姿态出现，离不开`CommonJS`规范的影响。



2.1.1 CommonJS的模块规范

- 模块引用

  ```
  var math = require('math')
  ```

- 模块定义

  将方法挂载在exports对象上作为属性即可定义导出的方式

- 模块标识

  模块标识其实就是传递给`require()`方法的参数,它必须是符合小驼峰命名的字符串,或者以...开头的相对路径,或者绝对路径。它可以没有文件名后缀`.js`.

`CommonJS`构建的这套模块导出和引入机制使得用户完全不必考虑变量污染。

2.2 Node的模块实现

在Node中引入模块，需要经历如下3个步骤

- 路径分析
- 文件定位
- 编译执行

在Node中，模块分为两类:

- Node提供的模块——核心模块

  核心模块引入时,文件定位和编译执行这两个步骤可以省略掉,并且在路径分析中优先判断,所以它的加载速度是最快的。

- 用户编写的模块——文件模块

  文件模块则是在运行时动态加载,需要完整的路径分析、文件定位、编译执行过程,速度比核心模块慢。

2.2.1 优先从缓存加载

`Node`对引入过的模块都会进行缓存,以减少二次引入时的开销。不同的地方在于,浏览器仅仅缓存文件,而Node缓存的是编译和执行之后的对象。不论是核心模块还是文件模块,`require()`方法对相同模块的二次加载都一律采用缓存优先的方式,这是第一优先级的。不同之处在于核心模块的缓存检查先于文件模块的缓存检查。

> 从缓存加载的优化策略使得二次引入时，不需要路径分析、文件定位和编译执行的过程，大大提高了再次加载模块时的效率。

2.2.2 路径分析和文件定位

模块标识符

模块标识符在Node中主要分为以下几类：

- 核心模块：http、fs、path等，其优先级仅次于缓存加载，它在Node的源代码编译过程中已经编译为二进制代码，其加载过程最快。
- .或..开始的相对路径文件模块、以/开始的绝对路径文件模块。文件模块给Node指定了确切的文件位置，所以查找过程可以节约大量时间，加载速度慢于核心模块。
- 非路径形式的文件模块，如自定义的connect模块。它是一种特殊的文件模块，可能是一个文件或者包的形式。这类模块查找最费时，也是最慢的一种。

模块路径

`模块路径是Node在定位自定义模块的具体文件时指定的查找策略，具体表现为一个路径组成的数组。`

关于这个路径的生成规则，可以手动尝试一番。

（1）创建module_path.js文件，其内容为`console.log(module.paths);`。

（2）将其放到任意一个目录中然后执行node module_path.js。

在Linux下，可能会有这样一个数组输出：

```
[ '/home/jackson/research/node_modules',
'/home/jackson/node_modules',
'/home/node_modules',
'/node_modules' ]
```

在Windows下，也许是这样：

```
[ 'c:\\nodejs\\node_modules', 'c:\\node_modules' ]
```

可以看出，模块路径的生成规则如下所示。

- 当前文件目录下的node_modules目录。
- 父目录下的node_modules目录。
- 父目录的父目录下的node_modules目录。
- 沿路径向上逐级递归，直到根目录下的node_modules目录。
  它的生成方式与JavaScript的原型链或作用域的查找方式十分类似。**在加载的过程中，Node会逐个尝试模块路径中的路径，直到找到目标文件为止。也就是说当前文件的路径越深，模块查找耗时会越多，这是自定义模的加载速度是最慢的原因。**

文件定位

- 文件扩展名分析：require()在分析标识符的过程中，会出现标识符中不包含文件扩展名的情况。CommonJs模块规范也允许标识符中不包含文件扩展名，这种情况下，Node会按.js、.json、.node的次序不足扩展名，依次尝试。但是尝试的过程中，需要调用fs模块同步阻塞式地判断文件是否存在，这里会引起性能问题。**所以，如果是.node和.json文件，在传递给require()的标识符中，尽量带上扩展名。**
- 目录分析和包：在分析标识符的过程中，可能没有查找到对应文件，但是却得到一个目录，此时Node会将目录当作一个包来处理。在这个过程中，Node对CommonJS包规范进行了一定程度的支持。首先，Node在当前目录下查找package.json，通过JSON.parse()解析出包描述对象，从中取出main属性指定的文件名进行定位。而如果main属性指定的文件名错误，或者压根没有package.json文件，Node会将index当作默认文件名，然后依次查找index.js、index.json、index.node。如果在目录分析的过程中没有定位成功任何文件，则自定义模块进入下一个模块路径进行查找。如果模块路径数组都被遍历完毕，依然没有查找到目标文件，则会抛出查找失败的异常。

2.2.3 模块编译

编译和执行是引入文件模块的最后一个阶段。定位到具体的文件后，Node会新建一个模块对象，然后根据路径载入并编译。对于不同的文件扩展名，其载入方法也有所不同，具体如下所示：

- .js文件，通过 `fs模块` 同步读取文件后编译执行
- .node文件，这是用C/C++编写的扩展文件，通过 `dlopen()` 方法加载最后编译生成的文件。
- .json文件，通过 `fs模块` 同步读取文件后，用 `JSON.parse()` 解析返回结果。
- 其他扩展名文件，都被当做.js文件载入。
  每一个编译成功的模块都会将其文件路径作为索引缓存在Module._cache对象上，以提高二次引入的性能。

JavaScript模块的编译

在编译的过程中，Node会对获取的JavaScript文件内容进行头尾包装。在头部添加了了`(function (exports, require, module, __filename, __dirname) {\n`，在尾部添加`\n});`。一个正常的JavaScript文件会被包装成如下的样子：

```
(function (exports, require, module, __filename, __dirname) {
     var math = require('math');
     exports.area = function (radius) {
     return Math.PI * radius * radius;
     };
});
```

这样每个模块文件之间都进行了作用域隔离。包装之后的代码会通过vm原生模块的`runInThisContext()`方法执行（`类似于eval，只是具有明确上下文，不污染全局`），返回一个具体的`function`对象。最后，将当前模块对象的`exports`属性、`require()`方法、`module`（模块对象自身），以及在文件定位中得到的完整文件路径（`__filename`）和文件目录（`__dirname`）作为参数传递给这个function()执行。

> 这就是这些变量（`exports`属性、`require()`方法、`module`、`__filename`和`__dirname`）并没有定义，在每个模块文件中却存在的原因。另外，在执行之后，模块的exports属性会被返回给了调用方。export属性上的任何方法和属性都可以被外部调用到，但是模块中的其余变量或属性则不可直接被调用。

C/C++模块的编译

Node调用`process.dlopen()`方法进行加载和执行（.node模块文件并不需要编译，因为它是编写C/C++模块之后编译生成的，所以这里只有加载和执行的过程）。在执行的过程中，模块的exports对象与.node模块产生联系，然后返回给调用者。

> 在Node的架构下，dlopen()方法在Windows和*nix平台下分别有不同的实现，通过libun兼容层进行了封装。

JSON文件的编译

Node利用`fs模块`同步读取JSON文件的内容之后，调用JSON.parse()方法得到对象，然后再将它赋给模块对象的exports，以供外部调用。

2.3 核心模块

> 待更新。。。

三、异步I/O

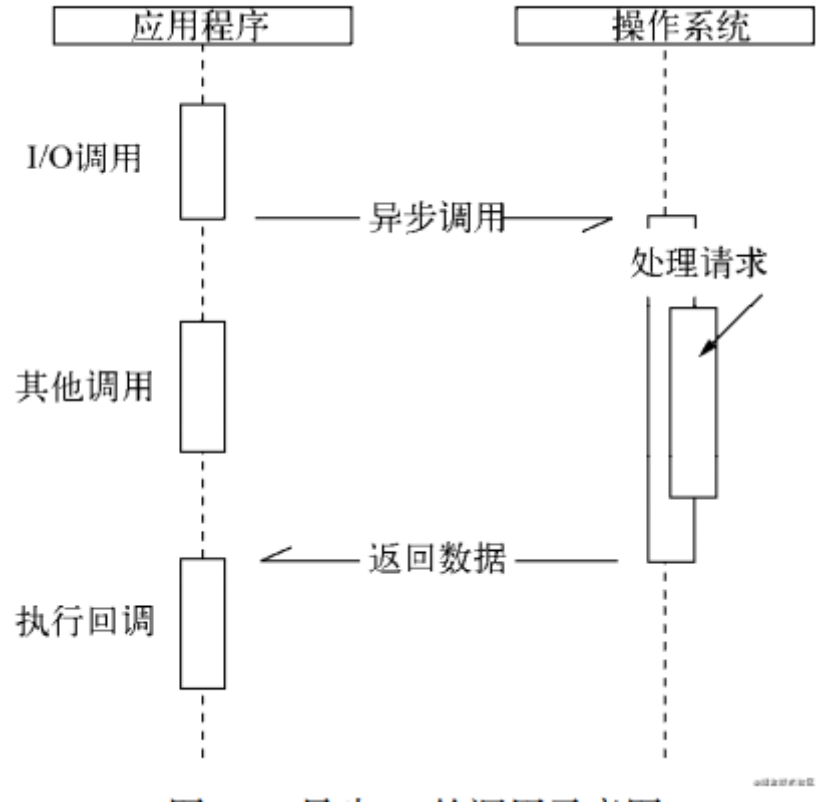
3.1 异步I/O的优越性

3.1.1 用户体验

前端通过异步请求可以消除掉UI阻塞的现象，后端也可以通过异步I/O消除I/O阻塞现象。只有后端能够快速响应资源，才能让前端的体验变好。

3.1.2 资源分配

> 在计算机资源中，通常I/O与CPU计算之间是可以并行进行的。但是同步的编程模型导致的问题是，I/O的进行会让后续任务等待，这造成资源不能被更好地利用。
> **单线程同步编程模型会因阻塞I/O导致硬件资源得不到更优的使用。多线程编程模型也因为编程中的死锁、状态同步和过多资源占用等问题让开发人员头疼。**
> **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、状态同步和过多资源占用问题；利用异步I/O，让单线程远离阻塞。以更好地使用CPU。**



为了弥补单线程无法利用多核CPU的缺点，Node提供了类似前端浏览器中Web Workers的子进程，该子进程可以通过工作进程高效地利用CPU和I/O。

3.2 异步I/O的实现现状

3.2.1 异步I/O和非阻塞I/O

从实际效果而言，异步和非阻塞都达到了并行I/O的目的。但是从计算机内核I/O而言，异步/同步和阻塞/非阻塞是两回事。

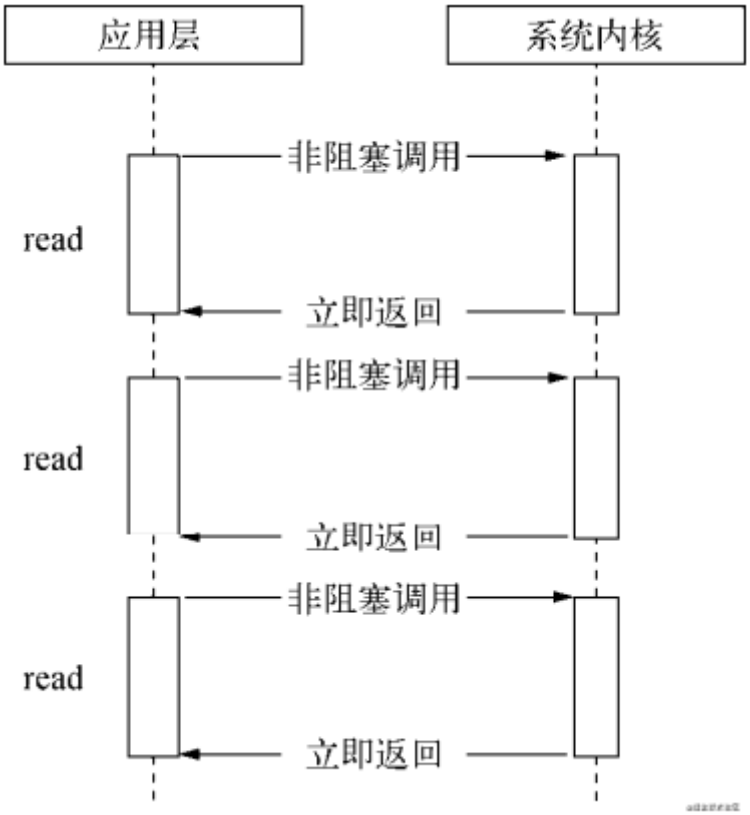
操作系统内核对于I/O只有两种方式：**阻塞与非阻塞**。阻塞I/O造成CPU等待I/O，浪费等待时间，CPU无法得到充分利用。为了提高性能，内核提供了非阻塞I/O。非阻塞I/O跟阻塞I/O的差别为调用之后会立即返回。

> 操作系统对计算机进行了抽象，将所有输入输出设备抽象为文件。内核在进行文件I/O操作时，通过文件描述符进行管理，而文件描述符类似于应用程序与系统内核之间的凭证。应用程序如果需要进行I/O调用，需要先打开文件描述符，然后再根据文件描述符去实现文件的数据读写。此处非阻塞I/O与阻塞I/O的区别在于阻塞I/O完成整个获取数据的过程，而非阻塞I/O则不带数据直接返回，要获取数据，还需要通过文件描述符再次读取。

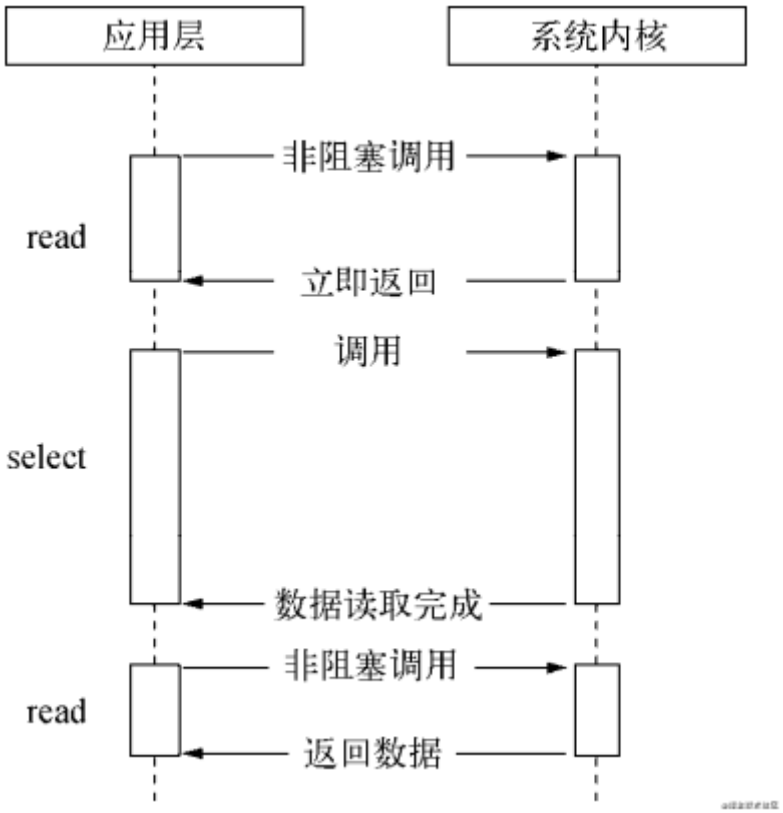
非阻塞I/O返回之后，CPU的时间片可以用来处理其他事务，此时的性能提升是明显的。但是非阻塞I/O也存在一些问题，由于完整的I/O并没有完成，立即返回的并不是业务层期望的数据，而仅仅是当前调用的状态。为了获取完整的数据，应用程序需要通过轮询来确认是否完成。

轮巡

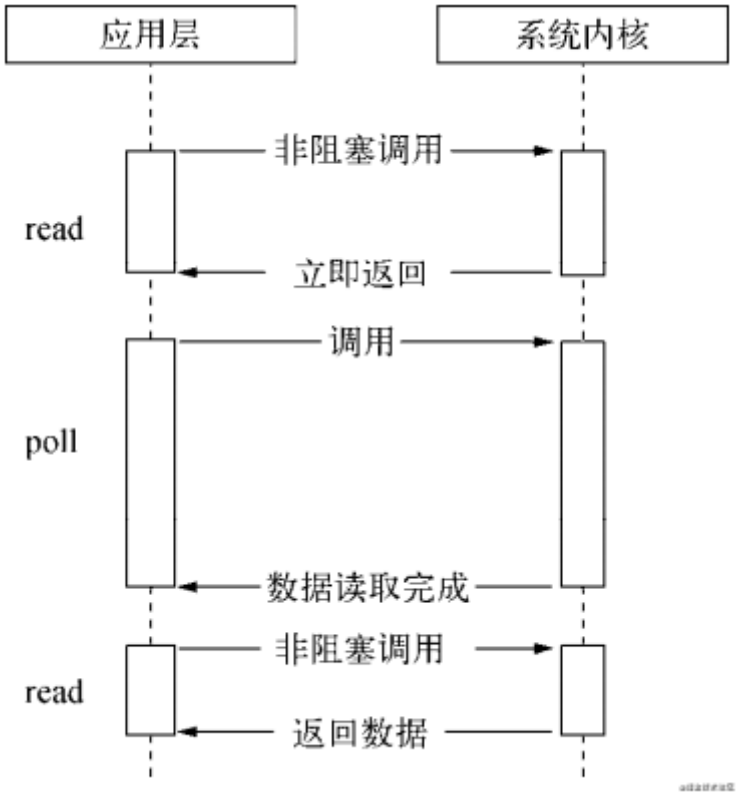
- read：通过重复调用来检查I/O的状态来完成完整数据的读取。



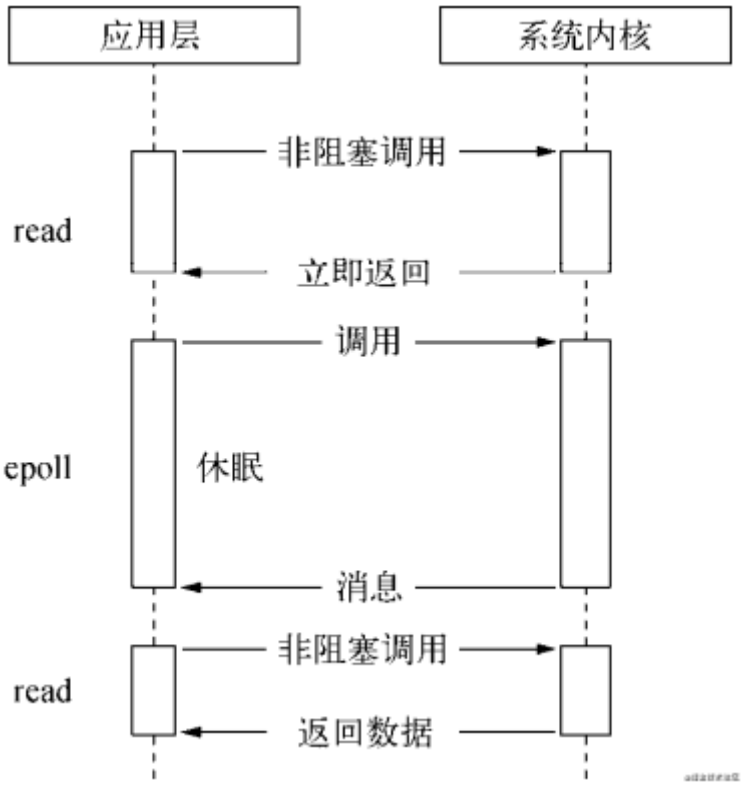
- select：在read的基础上进行改进，通过对文件描述符上的事件状态来进行判断，但是最多可以同时检查1024个文件描述符。



- poll：在select的基础上改进，采用链表的方式避免数组长度的限制，其次可以避免一些不必要的检查。



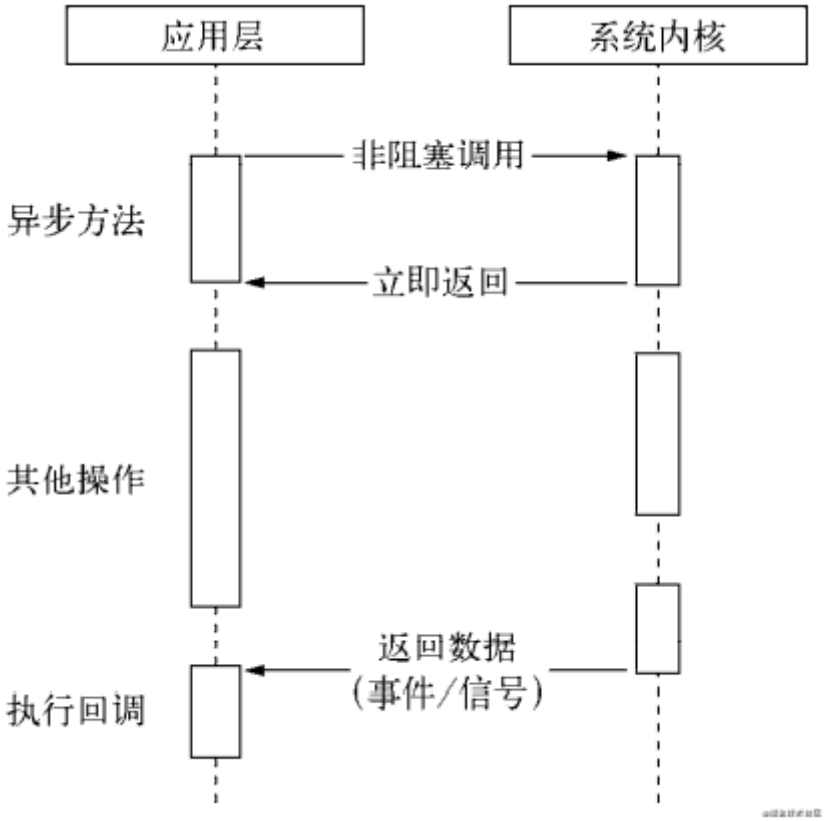
- epoll：**该方案是Linux下效率最高的I/O事件通知机制，在进入轮巡的时候，如果没有检查到I/O事件，将会进行休眠，直到事件发生将它唤醒。真实利用了事件通知、执行回调的方式，而不是遍历查询，执行效率高。**



3.2.2 理想的非阻塞异步I/O

尽管epoll已经利用了事件来降低CPU耗用，但是休眠期间CPU几乎是闲置的，对于当前线程来说利用率不够。

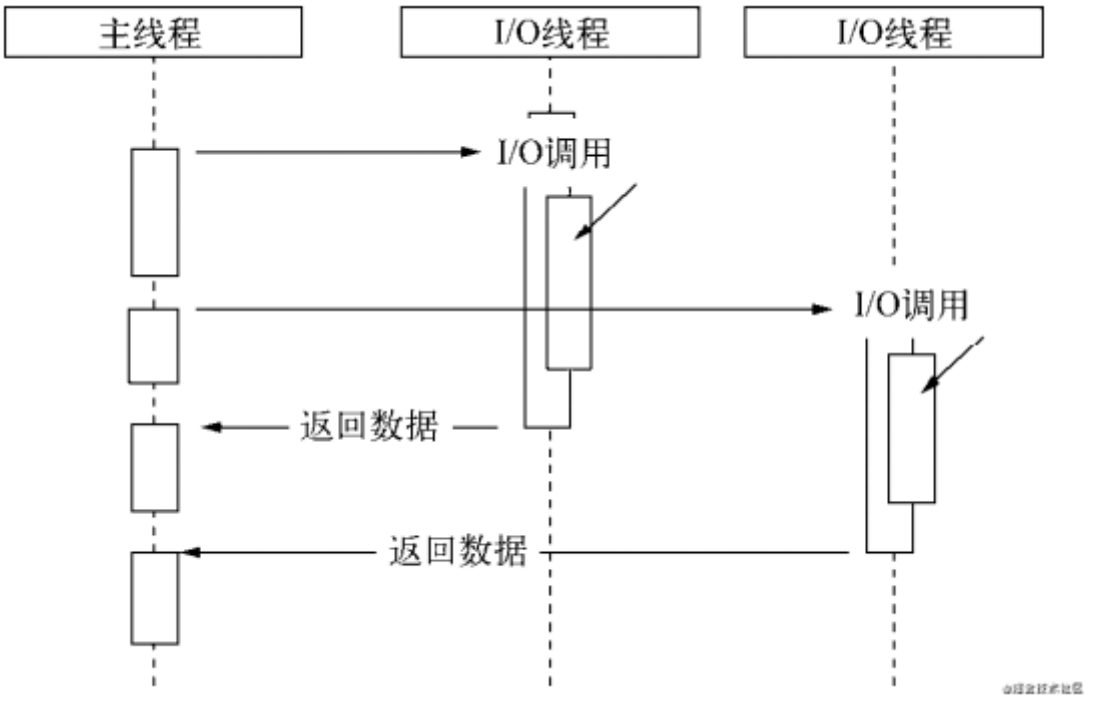
理想的非阻塞异步I/O应该是应用程序发起非阻塞调用，无需通过遍历或者事件唤醒等方式轮询，可以直接处理下一任务，只需在I/O完成后通过信号或回调将数据传递给应用程序即可。



> 幸运的是，在Linux下存在这样一种方式，它原生提供的一种异步I/O方式（AIO）就是通过信号或回调来传递数据的。但是不幸的是，只有Linux下有，并且AIO只支持内核I/O中的O_DIRECT方式读取，导致无法利用系统缓存。

3.2.3 现实的异步I/O

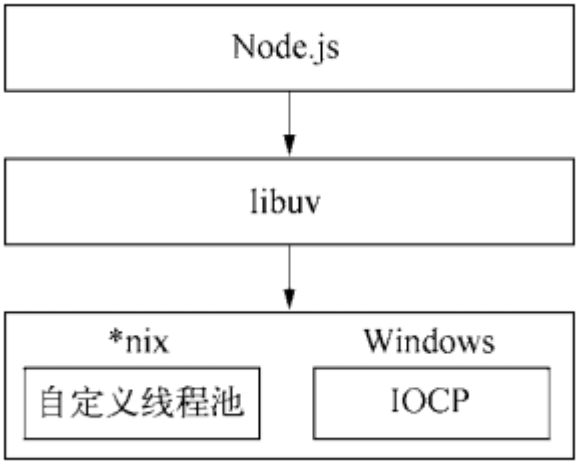
可以通过让部分线程进行阻塞I/O或者非阻塞I/O加轮询技术来完成数据读取，让一个线程进行计算处理，最终通过线程之间的通信将I/O得到的数据进行传递，这就模拟实现了异步I/O。



最初，Node在*nix平台下采用了libeio（libeio实际上依然是采用线程池与阻塞I/O模拟异步I/O）配合libev实现I/O部分，实现了异步I/O，在Node v0.9.3中，自行实现了线程池来完成异步I/O。

Windows下的IOCP在某种程度上提供了理想的异步I/O：调用异步方法，等到I/O完成之后的通知，执行回调，用户无需考虑轮询。但是它内部依然是线程池原理，不同之处在于这些线程池由系统内核接手管理。

> 由于Windows平台和*nix平台的差异，Node提供了libuv作为抽象封装层，使得所有平台兼容性的判断都由这一层来完成，并保证上层的Node与下层的自定义线程池及IOCP之间各自独立。Node在编译期间会判断平台条件，选择性编译unix目录或是win目录下的源文件到目标程序中。



需要强调的一点是，I/O不仅仅只限于磁盘文件的读写。*nix将计算机抽象了一番，磁盘文件、硬件、套接字等几乎所有计算机资源都被抽象为了文件，因此这里描述的阻塞和非阻塞情况同样适用于套接字等。另外一个需要强调的就是时常提到Node是单线程的，这里的单线程仅仅只是JavaScript执行在单线程中而已。在Node中，无论是*nix还是Windows平台，内部完成I/O任务的另有线程池。

3.3 Node中的异步I/O

完成整个异步`I/O`环节的有事件循环、观察者、请求对象、`I/O`线程池等。

3.3.1 事件循环

Node自身的执行模型——事件循环，正是它使得回调函数十分普遍。

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，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。如果存在相关联的回调函数，就执行它们。然后进入下个循环，直至不再有事件处理，退出进程。

3.3.2 观察者

每个事件循环中有一个或者多个观察者，而判断是否有事件要处理的过程，就是向这些观察者询问是否有要处理的事件。

浏览器采用了类似机制。事件可能来自用户的点击或者加在某些文件时产生，而这些产生的事件都有对应的观察者。在Node中，事件主要来源于网络请求、文件I/O等，这些事件对应观察者有文件I/O观察者、网络I/O观察者等。

事件循环是一个典型的**生产者/消费者模型。**异步I/O、网络请求等则是事件的生产者，源源不断为Node提供不同类型的事件，这些事件被传递到对应的观察者那里，事件循环则从观察者那里取出事件并处理。

在`Windows`下，这个循环基于`IOCP`创建，而在`*nix`下则基于多线程创建。

3.3.3 请求对象

对于一般的（非异步）回调函数，函数由我们自行调用，如下所示：

```
const forEach = function (list, callback) {
    for (let i = 0; i < list.length; i++) {
        callback(list[i], i, list);
    }
};
```

对于Node的异步I/O调用而言，回调函数却不由开发者来调用。

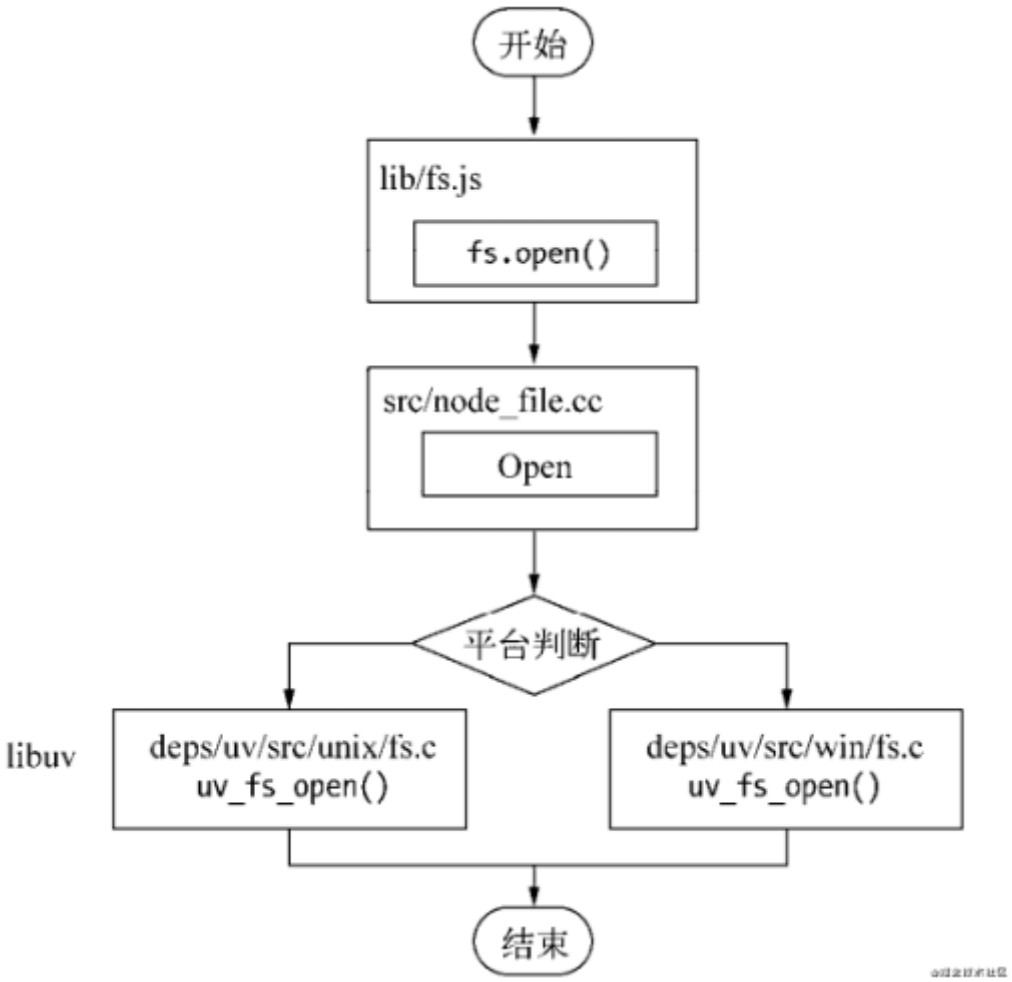
事实上，从JavaScript发起调用到内核执行完I/O操作的过渡过程中，存在一种中间产物，即请求对象。

示例fs.open()

以最简单的`fs.open()`为例，探索Node与底层之间是如何执行异步I/O调用以及回调函数究竟是如何被调用执行的：

```
fs.open = function(path, flags, mode, callback) {
    // ...
    binding.open(pathModule._makeLong(path), stringToFlags(flags), mode,
callback);
};
```

`fs.open()`的作用就是根据指定路径和参数去打开一个文件，从而得到一个文件描述符，这是后续所有I/O操作的初始化操作。



从JavaScript调用Node的核心模块，核心模块调用C++内建模块，内建模块通过libuv进行系统调用，这是Node经典的调用方式。这里libuv作为封装层，有两个平台的实现，实际上是调用了`uv_fs_open()`方法，在`uv_fs_open()`方法的调用过程中，创建了一个`FSReqWrap`请求对象。从JavaScript层传入的参数和当前方法都被封装在这个请求对象中，**其中回调函数则被设置在这个对象的`oncomplete_sym`属性上**：

```
req_wrap->object_->Set(oncomplete_sym, callback);
```

对象包装完毕后，在Windows下，则会调用`QueueUserWorkItem()`方法将这个`FSReqWrap`对象推入线程池中等待执行，该方法的代码如下所示：

```
QueueUserWorkItem($uv_fs_thread_proc, req, WT_EXECUTEDEFAULT)
```

`QueueUserWorkItem()`方法接受3个参数：第一个参数是将要执行的方法的引用（`uv_fs_thread_proc`），第二个参数是`uv_fs_thread_proc`方法运行时所需要的参数；第三个参数是执行的标志。当线程池中有可用线程时，会调用`uv_fs_thread_proc()`方法。`uv_fs_thread_proc()`方法会根据传入参数的类型调用相应的底层函数。以`uv_fs_open()`为例，实际上调用的底层函数是`fs_open()`方法。

至此，`JavaScript`调用立即返回，由`JavaScript`层面发起的异步调用的第一阶段到此结束。`JavaScript`线程可以继续执行当前任务的后续操作。当前`I/O`操作在线程池中等待执行，不管它是否为阻塞`I/O`，都不会影响`JavaScript`线程的后续执行，如此便达到了异步的目的。

> 请求对象是异步I/O过程中的重要中间产物，所有的状态都保存在这个对象中，包括送入线程池等待执行以及I/O操作完毕后的回调处理。

3.3.4 执行回调

组装好请求对象、送入I/O线程池等待执行，实际上完成的只是异步I/O的第一部分，回调通知则是第二部分。

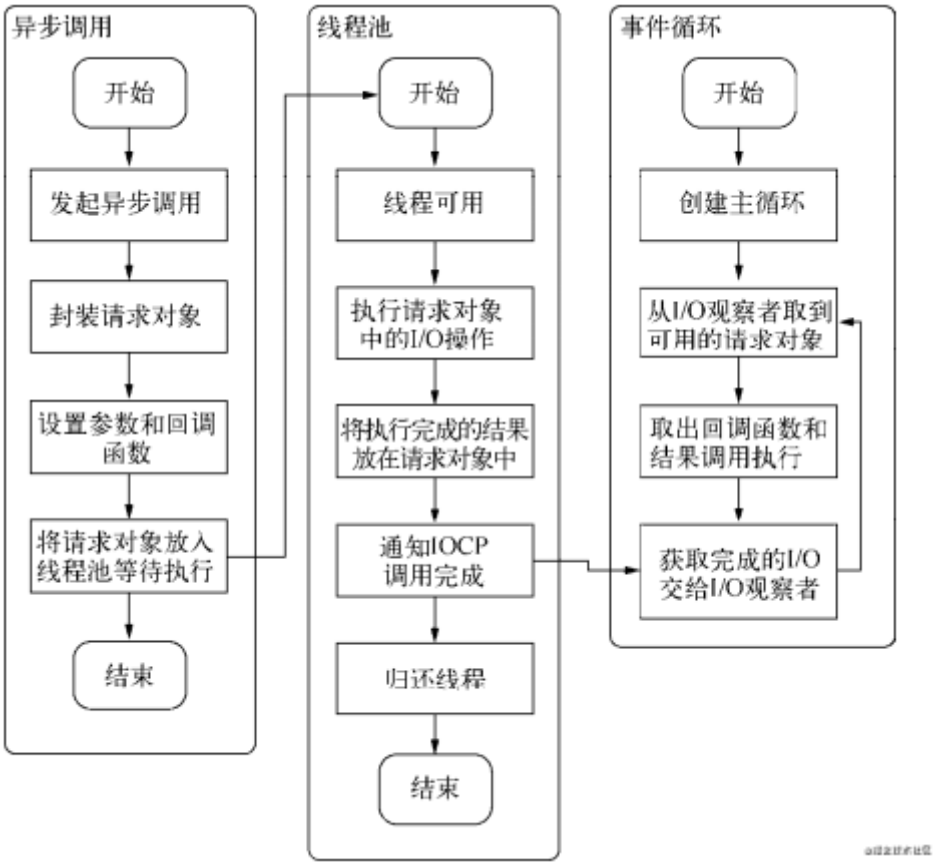
线程池中的I/O操作调用完毕之后，会将获取的结果储存在请求对象的result（`req->result`）属性上，然后调用PostQueuedCompletionStatus()通知IOCP，告知当前对象操作已经完成：

```
PostQueuedCompletionStatus((loop)->iocp, 0, 0, &((req)->overlapped))
```

`PostQueuedCompletionStatus()`方法的作用是向IOCP提交执行状态，并将线程归还线程池。通过`PostQueuedCompletionStatus()`方法提交的状态，可以通过`GetQueuedCompletionStatus()`提取。

在这个过程中，不可缺少的还有`I/O观察者`。在每次Tick的执行中，它都会调用IOCP相关的`GetQueuedCompletionStatus()`方法检查线程池中是否有执行完的请求，如果存在，会将请求对象加入到I/O观察者的队列中，然后将其当做事件处理。

`I/O观察者`回调函数的行为就是取出请求对象的`result`属性作为参数，取出`oncomplete_sym`属性作为方法，然后调用执行，以此达到调用JavaScript中传入的回调函数的目的。



3.3.5 小结

事实上，在Node中，除了JavaScript是单线程的以外，Node自身其实是多线程的，只是I/O线程所使用的CPU较少。另外除了用户代码无法并行执行以外，所有的I/O（磁盘I/O和网络I/O等）都是可以并行起来的。

3.4 非I/O的异步API

Node中还存在一些与I/O无关的异步API，比如：`setTimeout()`、`setInterval()`、`setImmediate()process.nextTick()`。

3.4.1 定时器

`setTimeout()`和`setInterval()`的实现原理与异步I/O比较类似，只是不需要I/O线程池的参与。调用setTimeout()或者setInterval()创建的定时器会被插入到定时器观察者内部的一个红黑树中。每次Tick执行时，会从该红黑树中迭代去除定时器对象，检查是否超过定时时间，如果超过，就形成一个事件，它的回调函数将立即执行。

3.4.2 process.nextTick()

`process.nextTick()`方法的操作相对较为轻量，具体代码如下：

```
process.nextTick = function(callback) {
     // on the way out, don't bother.
     // it won't get fired anyway
     if (process._exiting) return;
     if (tickDepth >= process.maxTickDepth)
         maxTickWarn();
     var tock = { callback: callback };
     if (process.domain) tock.domain = process.domain;
     nextTickQueue.push(tock);
     if (nextTickQueue.length) {
         process._needTickCallback();
     }
};
```

每次调用process.nextTick()方法，只会将回调函数放入队列中，在下一轮Tick时取出执行。定时器中采用红黑树的操作时间复杂度为O(lg(n))，nextTick()的时间复杂度为O(1)。相较之下，process.nextTick()更高效。

3.4.3 setImmediate()

`setImmediate()`方法和与`process.nextTick()`方法十分类似。但是两者之间有着细微差别。

```
process.nextTick(function () {
 console.log('nextTick延迟执行');
});
setImmediate(function () {
 console.log('setImmediate延迟执行');
});
console.log('正常执行');

执行结果:
正常执行
nextTick延迟执行
setImmediate延迟执行
```

process.nextTick()中的回调函数执行优先级要高于setImmediate()。因为事件循环对观察者的检查是有先后顺序的，process.nextTick()属于idle观察者，setImmediate()属于check观察者。在每一轮循环检查中，idle观察者先于I/O观察者，I/O观察者先于check观察者。

3.4.4 process.nextTick()和setImmediate()

- **在具体实现上，process.nextTick()的回调函数保存在一个数组中，setImmediate()的结果则是保存在链表中。**
- **在行为上，process.nextTick()在每轮循环中会将数组中的回调函数全部执行完，而setImmediate()在每轮循环中只执行链表中的一个回调函数。如下示例代码：**

```
// 加入两个nextTick()的回调函数
process.nextTick(function () {
 console.log('nextTick延迟执行1');
});
process.nextTick(function () {
 console.log('nextTic延迟执行2');
});
// 加入两个setImmediate()的回调函数
setImmediate(function () {
```

```
 console.log('setImmediate延迟执行1');
 // 进入下次循环
 process.nextTick(function () {
 console.log('强势插入');
 });
});
setImmediate(function () {
 console.log('setImmediate延迟执行2');
});
console.log('正常执行');

执行结果如下:
正常执行
nextTick延迟执行1
nextTick延迟执行2
setImmediate延迟执行1
强势插入
setImmediate延迟执行2
```

从执行结果上可以看出，当第一个setImmediate()的回调函数执行后，并没有立即执行第二个，而是进入了下一轮循环，再次按process.nextTick()优先、setImmediate()次后的顺序执行。**之所以这样合计，是为了保证每轮循环能够较快地执行结束，防止CPU占用过多而阻塞后续I/O调用的情况。**

四、异步编程

4.1 异步编程的优势和难点

4.1.1 优势

Node带来的最大特性莫过于基于事件驱动的非阻塞I/O模型，非阻塞I/O可以使CPU和I/O不需要互相依赖等待，让资源得到更好的利用。

根据Node实现异步I/O的原理，我们可以知道，利用事件循环的方式，JavaScript线程像是一个分配任务和处理结果的大管家，I/O线程池里的各个I/O线程都是小二，负责兢兢业业地完成分配的任务，彼此之间互不依赖，所以可以保持整体的高效率。这个模型的缺点在于管家无法承担过多的细节性任务，如果承担太多，则会影响到任务的调度，管家忙个不停，小二却得不到活干，结局则是整体效率的降低。

> 由于事件循环模型需要应对海量请求，海量请求同时作用在单线程上，就需要防止任何一个计算耗费过多的CPU时间片。建议对CPU的耗用不超过10ms，或者将大量的计算分解为诸多的小量计算，通过setImmediate()进行调度。

4.1.2 难点

异常处理

过去我们处理异常时，通常使用try/catch/final语句块进行异常捕获，示例代码如下：

```
try {
 JSON.parse(json);
} catch (e) {
 // TODO
}
```

但是这对于异步编程而言并不一定适用。

异步I/O的实现主要包含两个阶段：提交请求和处理结果。这两个阶段中间有事件循环的调度，两者彼此不关联。异步方法通常在第一个阶段提交请求后立即返回，因为一场并不一定发生在这个阶段，try/catch的功效在此处不会发挥任何作用。

```
var async = function (callback) {
 process.nextTick(callback);
};

try {
 async(callback);
} catch (e) {
 // TODO
}
```

调用async()方法后，callback被存放起来，直到下一个事件循环（Tick）才会被取出来执行。尝试对异步方法进行try/catch操作只能捕获档次事件循环内的异常，对callback执行时抛出的异常将无能为力。

Node在处理异常上形成了一种约定，将异常作为回调函数的第一个实参传回，如果为空值，则表明异步调用没有异常抛出。

函数嵌套过深

```
fs.readdir(path.join(__dirname, '..'), function (err, files) {
  files.forEach(function (filename, index) {
    fs.readFile(filename, 'utf8', function (err, file) {
      // TODO
    });
  });
});
```

> 该问题在新语法中得到了很好的解决

阻塞代码

JavaScript这门编程语言本身没有sleep()这样的线程沉睡功能，唯独能用于延时操作的只有setInterval()和setTimeout()这两个函数。但是这两个函数并不能阻塞后续代码的持续执行。

所以很多开发者为了实现sleep(1000)的效果，会写出下述代码：

```
// TODO
var start = new Date();
while (new Date() - start < 1000) {
 // TODO
}
// 需要阻塞的代码
```

但事实上，这段代码会持续占用CPU，与真正的线程沉睡相去甚远，完全破坏了事件循环调度，导致其余任何请求都会得不到响应。遇到这样的需求时，在统一规划业务逻辑之后，调用setTimeout()的效果会更好。

4.2 异步编程解决方案

- 事件发布/订阅模式
- Promise/Deferred模式
- 流程控制库

4.2.1 事件发布/订阅模式

事件监听器模式是一种广泛用于异步编程的模式，是回调函数的事件化，又称发布/订阅模式。

Node自身提供的events模块是发布/订阅模式的一个简单实现，Node中部分模块都继承自它，这个模块比前端浏览器中的大量DOM事件简单，不存在事件冒泡，也不存在preventDefault()、stopPropagation()和stopImmediatePropagation()等控制事件传递的方法。它具有addListener/on()、once()、removeListener()、removeAllListeners()和emit()等基本的事件监听模式的方法实现。事件发布/订阅模式的操作极其简单，示例代码如下：

```
// 订阅
emitter.on("event1", function (message) {
 console.log(message);
});
// 发布
emitter.emit('event1', "I am message!");
```

事件发布/订阅模式常常用来解耦业务逻辑，事件发布者无须关注订阅的侦听器如何实现业务逻辑，甚至不用关注有多少个侦听器存在，数据通过消息的方式可以很灵活地传递。 在一些典型场景中，可以通过事件发布/订阅模式进行组件封装，将不变的部分封装在组件内部，将容易变化、需自定义的部分通过事件暴露给外部处理，这是一种典型的逻辑分离方式。在这种事件发布/订阅式组件中，事件的设计非常重要，因为它关乎外部调用组件时是否优雅，从某种角度来说事件的设计就是组件的接口设计。

从另一个角度来看，**事件侦听器模式也是一种钩子（hook）机制，利用钩子导出内部数据或状态给外部的调用者。**Node中的很多对象大多具有黑盒的特点，功能点较少，如果不通过事件钩子的形式，我们就无法获取对象在运行期间的中间值或内部状态。**这种通过事件钩子的方式，可以使编程者不用关注组件是如何启动和执行的，只需关注在需要的事件点上即可。**

```
var options = {
 host: 'www.google.com',
 port: 80,
 path: '/upload',
 method: 'POST'
};
var req = http.request(options, function (res) {
 console.log('STATUS: ' + res.statusCode);
 console.log('HEADERS: ' + JSON.stringify(res.headers));
 res.setEncoding('utf8');
 res.on('data', function (chunk) {
 console.log('BODY: ' + chunk);
 });
 res.on('end', function () {
 // TODO
 });
});
req.on('error', function (e) {
 console.log('problem with request: ' + e.message);
});
// write data to request body
req.write('data\n');
req.write('data\n');
req.end();
```

在这段HTTP请求的代码中，开发者只需要将视线放在error、data、end这些业务事件点上即可，至于内部的流程如何，无需过于关注。

> Node对事件发布/订阅的机制做了一些额外的处理，这大多是基于健壮性而考虑的。下面为两个具体的细节点:
>
> - **如果对一个事件添加了超过10个侦听器，将会得到一条警告。**这一处设计与Node自身单线程运行有关，设计者认为侦听器太多可能导致内存泄漏，所以存在这样一条警告。调用emitter.setMaxListeners(0);可以将这个限制去掉。另一方面，由于事件发布会引起一系列侦听器执行，如果事件相关的侦听器过多，可能存在过多占用CPU的情景。
> - **为了处理异常，EventEmitter对象对error事件进行了特殊对待。**如果运行期间的错误触发了error事件，EventEmitter会检查是否有对error事件添加过侦听器。如果添加了，这个错误将会交由该侦听器处理，否则这个错误将会作为异常抛出。如果外部没有捕获这个异常，将会引起线程退出。一个健壮的EventEmitter实例应该对error事件做处理。

继承events模块

实现一个继承EventEmitter的类是十分简单的，以下代码是Node中Stream对象继承EventEmitter的例子:

```
var events = require('events');
function Stream() {
 events.EventEmitter.call(this);
}
util.inherits(Stream, events.EventEmitter);
```

Node在util模块中封装了继承的方法，所以此处可以很便利地调用。开发者可以通过这样的方式轻松继承EventEmitter类，利用事件机制解决业务问题。**在Node提供的核心模块中，有近半数都继承自EventEmitter。**

利用事件队列解决雪崩问题

在计算机中，缓存由于存放在内存中，访问速度十分快，常常用于加速数据访问，让绝大多数的请求不必重复去做一些低效的数据读取。所谓雪崩问题，就是在高访问量、大并发量的情况下缓存失效的情景，此时大量的请求同时涌入数据库中，数据库无法同时承受如此大的查询请求，进而往前影响到网站整体的响应速度。

以下是一条数据库查询语句的调用:

```
var select = function (callback) {
 db.select("SQL", function (results) {
 callback(results);
 });
};
```

如果站点刚好启动，这时缓存中是不存在数据的，而如果访问量巨大，同一句SQL会被发送到数据库中反复查询，会影响服务的整体性能。一种改进方案是添加一个状态锁，相关代码如下:

```
var status = "ready";
var select = function (callback) {
 if (status === "ready") {
   status = "pending";
   db.select("SQL", function (results) {
     status = "ready";
     callback(results);
   });
 }
};
```

但是在这种情景下，连续地多次调用select()时，只有第一次调用是生效的，后续的select()是没有数据服务的，这个时候可以引入事件队列，相关代码如下：

```
var proxy = new events.EventEmitter();
var status = "ready";
var select = function (callback) {
 proxy.once("selected", callback);
 if (status === "ready") {
   status = "pending";
   db.select("SQL", function (results) {
     proxy.emit("selected", results);
     status = "ready";
   });
 }
};
```

利用了once()方法，将所有请求的回调都压入事件队列中，利用其执行一次就会将监视器移除的特点，保证每一个回调只会被执行一次。对于相同的SQL语句，保证在同一个查询开始到结束的过程中永远只有一次。SQL在进行查询时，新到来的相同调用只需在队列中等待数据就绪即可，一旦查询结束，得到的结果可以被这些调用共同使用。这种方式能节省重复的数据库调用产生的开销。由于Node单线程执行的原因，此处无须担心状态同步问题。这种方式其实也可以应用到其他远程调用的场景中，即使外部没有缓存策略，也能有效节省重复开销。

4.3 异步并发控制

> 异步I/O与同步I/O的显著差距：同步I/O因为每个I/O都是彼此阻塞的，在循环体中，总是一个接着一个调用，不会出现耗用文件描述符太多的情况，同时性能也是低下的；对于异步I/O，虽然并发容易实现，但是由于太容易实现，依然需要控制。换言之，尽管是要压榨底层系统的性能，但还是需要给予一定的过载保护，以防止过犹不及。

4.3.1 bagpipe的解决方案

如何对既有的异步API添加过载保护，我们期望的当然不是去改动API。那么如何实现呢？bagpipe模块的解决思路是这样的:

- 通过一个队列来控制并发量。
- 如果当前活跃（指调用发起但未执行回调）的异步调用量小于限定值，从队列中取出执行。
- 如果活跃调用达到限定值，调用暂时存放在队列中。
- 每个异步调用结束时，从队列中取出新的异步调用执行。
  bagpipe的API主要暴露了一个push()方法和full事件，示例代码如下：

```
var Bagpipe = require('bagpipe');
// 设定最大并发数为10
var bagpipe = new Bagpipe(10);
for (var i = 0; i < 100; i++) {
 bagpipe.push(async, function () {
 // 异步回调执行
 });
}
bagpipe.on('full', function (length) {
 console.warn('底层系统处理不能及时完成，队列拥堵，目前队列长为:' + length);
});
```

push()

push()方法依然是通过函数变换的方式实现，假设第一个参数是方法，最后一个参数是回调函数，其余为其他参数，其核心实现如下：

```
/**
 * 推入方法，参数。最后一个参数为回调函数
 * @param {Function} method 异步方法
 * @param {Mix} args参数列表，最后一个参数为回调函数
 */
Bagpipe.prototype.push = function (method) {
 var args = [].slice.call(arguments, 1);
 var callback = args[args.length - 1];
 if (typeof callback !== 'function') {
   args.push(function () {});
 }
 if (this.options.disabled || this.limit < 1) {
   method.apply(null, args);
   return this;
 }
 // 队列长度也超过限制值时
 if (this.queue.length < this.queueLength || !this.options.refuse) {
   this.queue.push({
     method: method,
     args: args
   });
 } else {
   var err = new Error('Too much async call in queue');
   err.name = 'TooMuchAsyncCallError';
   callback(err);
 }
 if (this.queue.length > 1) {
   this.emit('full', this.queue.length);
 }
  this.next();
  return this;
};
```

next()

将调用推入队列后，调用一次next()方法尝试触发。next()方法的定义如下：

```
/*!
 * 继续执行队列中的后续动作
 */
Bagpipe.prototype.next = function () {
 var that = this;
 if (that.active < that.limit && that.queue.length) {
   var req = that.queue.shift();
   that.run(req.method, req.args);
 }
};
```

run()

next()方法主要判断活跃调用的数量，如果正常，将调用内部方法run()来执行真正的调用。这里为了判断回调函数是否执行，采用了一个注入代码的技巧，具体代码如下：

```
/*!
 * 执行队列中的方法
 */
Bagpipe.prototype.run = function (method, args) {
 var that = this;
 that.active++;
 var callback = args[args.length - 1];
 var timer = null;
 var called = false;
 // inject logic
 args[args.length - 1] = function (err) {
   // anyway, clear the timer
   if (timer) {
     clearTimeout(timer);
     timer = null;
   }
   // if timeout, don't execute
   if (!called) {
     that._next();
     callback.apply(null, arguments);
   } else {
     // pass the outdated error
     if (err) {
       that.emit('outdated', err);
     }
   }
 };
 var timeout = that.options.timeout;
 if (timeout) {
   timer = setTimeout(function () {
     // set called as true
     called = true;
     that._next();
     // pass the exception
     var err = new Error(timeout + 'ms timeout');
     err.name = 'BagpipeTimeoutError';
     err.data = {
       name: method.name,
       method: method.toString(),
       args: args.slice(0, -1)
     };
     callback(err);
   }, timeout);
 }
 method.apply(null, args);
};
```

用户传入的回调函数被真正执行前，被封装替换过。这个封装的回调函数内部的逻辑将活跃值的计数器减1后，主动调用next()执行后续等待的异步调用。bagpipe类似于打开了一道窗口，允许异步调用并行进行，但是严格限定上限。仅仅在调用push()时分开传递，并不对原有API有任何侵入。

拒绝模式

事实上，bagpipe还有一些深度的使用方式。对于大量的异步调用，也需要分场景进行区分，因为涉及并发控制，必然会造成部分调用需要进行等待。如果调用有实时方面的需求，那么需要快速返回，因为等到方法被真正执行时，可能已经超过了等待时间，即使返回了数据，也没有意义了。这种场景下需要快速失败，让调用方尽早返回，而不用浪费不必要的等待时间。bagpipe为此支持了拒绝模式。

拒绝模式的使用只要设置下参数即可，相关代码如下：

```
// 设定最大并发数为10
var bagpipe = new Bagpipe(10, {
 refuse: true
});
```

超时控制

> 造成队列拥塞的主要原因是异步调用耗时太久，调用产生的速度远远高于执行的速度。为了防止某些异步调用使用了太多的时间，我们需要设置一个时间基线，将那些执行时间太久的异步调用清理出活跃队列，让排队中的异步调用尽快执行。否则在拒绝模式下，会有太多的调用因为某个执行得慢，导致得到拒绝异常。相对而言，这种场景下得到拒绝异常显得比较无辜。为了公平地对待在实时需求场景下的每个调用，必须要控制每个调用的执行时间，将那些害群之马踢出队伍。
> bagpipe也提供了超时控制。超时控制是为异步调用设置一个时间阈值，如果异步调用没有在规定时间内完成，我们先执行用户传入的回调函数，让用户得到一个超时异常，以尽早返回。然后让下一个等待队列中的调用执行。

超时的设置如下：

```
// 设定最大并发数为10，超时时间为3秒
var bagpipe = new Bagpipe(10, {
 timeout: 3000
});
```

小结

异步调用的并发限制在不同场景下的需求不同：非实时场景下，让超出限制的并发暂时等待执行已经可以满足需求；但在实时场景下，需要更细粒度、更合理的控制。

五、内存控制

5.1 V8的垃圾回收机制与内存限制

5.1.1 V8的内存限制

在Node中通过JavaScript使用内存时会发现只能使用部分内存（64位系统下约为1.4 GB,32位系统下约为0.7 GB）。

5.1.2 V8的对象分配

在V8中，所有的JavaScript对象都是通过堆来进行分配的。Node提供了V8中内存使用量的查看方式，执行下面的代码，将得到输出的内存信息：

```
$ node
> process.memoryUsage();
{ rss: 14958592,
 heapTotal: 7195904,
 heapUsed: 2821496 }
```

在memoryUsage()方法返回的3个属性中，heapTotal和heapUsed是V8的堆内存使用情况，前者是已申请到的堆内存，后者是当前使用的量。

> V8为何要限制堆的大小，**表层原因为V8最初为浏览器而设计，不太可能遇到用大量内存的场景。对于网页来说，V8的限制值已经绰绰有余。深层原因是V8的垃圾回收机制的限制。**按官方的说法，以1.5 GB的垃圾回收堆内存为例，V8做一次小的垃圾回收需要50毫秒以上，做一次非增量式的垃圾回收甚至要1秒以上。这是垃圾回收中引起JavaScript线程暂停执行的时间，在这样的时间花销下，应用的性能和响应能力都会直线下降。这样的情况不仅仅后端服务无法接受，前端浏览器也无法接受。因此，在当时的考虑下直接限制堆内存是一个好的选择。
> 当然，这个限制也不是不能打开，V8依然提供了选项让我们使用更多的内存。Node在启动时可以传递`--max-old-space-size`（设置老生代内存空间的最大值）或`--max-new-space-size`（设置新生代内存空间的大小）来调整内存限制的大小，示例如下：

```
node --max-old-space-size=1700 test.js // 单位为MB
// 或者
node --max-new-space-size=1024 test.js // 单位为KB
```

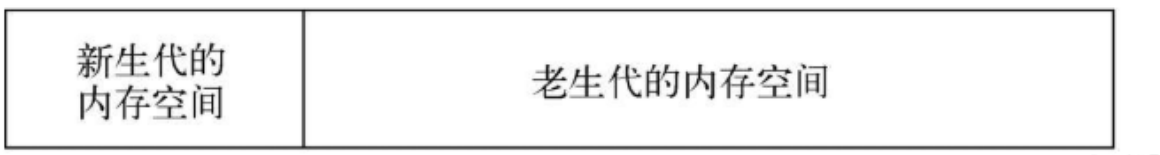
上述参数在V8初始化时生效，**一旦生效就不能再动态改变。**如果遇到Node无法分配足够内存给JavaScript对象的情况，可以用这个办法来放宽V8默认的内存限制，避免在执行过程中稍微多用了一些内存就轻易崩溃。

5.1.3 V8的垃圾回收机制

现代的垃圾回收算法中按对象的存活时间将内存的垃圾回收进行不同的分代，然后分别对不同分代的内存施以更高效的算法。

V8的内存分代

在V8中，主要将内存分为新生代和老生代两代。**新生代中的对象为存活时间较短的对象，老生代中的对象为存活时间较长或常驻内存的对象。**



V8堆的整体大小就是新生代所用内存空间加上老生代的内存空间。

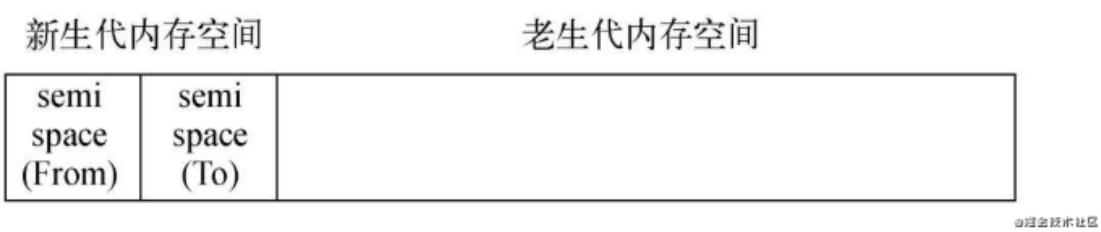
> 老生代的设置在64位系统下为1400 MB，在32位系统下为700 MB。对于新生代内存，它由两个reserved_semispace_size*所构成。按机器位数不同，reserved_semispace_size*在64位系统和32位系统上分别为16 MB和8 MB。所以新生代内存的最大值在64位系统和32位系统上分别为32MB和16 MB。

V8堆内存的最大保留空间，其公式为4 * reserved_semispace_size_ + max_old_generation_size_。因此，默认情况下，V8堆内存的最大值在64位系统上为1464 MB,32位系统上则为732 MB。这个数值可以解释为何在64位系统下只能使用约1.4 GB内存和在32位系统下只能使用约0.7 GB内存。

Scavenge算法

在分代的基础上，新生代中的对象主要通过Scavenge算法进行垃圾回收。在Scavenge的具体实现中，主要采用了Cheney算法。

> Cheney算法是一种采用复制的方式实现的垃圾回收算法。它将堆内存一分为二，每一部分空间称为semispace。在这两个semispace空间中，只有一个处于使用中，另一个处于闲置状态。处于使用状态的semispace空间称为From空间，处于闲置状态的空间称为To空间。当我们分配对象时，先是在From空间中进行分配。当开始进行垃圾回收时，会检查From空间中的存活对象，这些存活对象将被复制到To空间中，而非存活对象占用的空间将会被释放。完成复制后，From空间和To空间的角色发生对换。简而言之，在垃圾回收的过程中，就是通过将存活对象在两个semispace空间之间进行复制。Scavenge的缺点是只能使用堆内存中的一半，这是由划分空间和复制机制所决定的。但Scavenge由于只复制存活的对象，并且对于生命周期短的场景存活对象只占少部分，所以它在时间效率上有优异的表现。
> 由于Scavenge是典型的牺牲空间换取时间的算法，所以无法大规模地应用到所有的垃圾回收中。但可以发现，Scavenge非常适合应用在新生代中，因为新生代中对象的生命周期较短，恰恰适合这个算法。

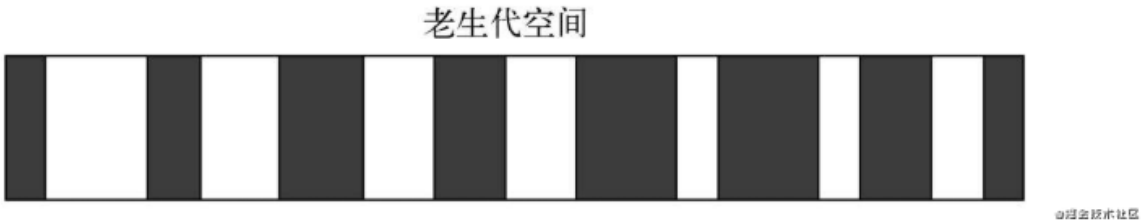


当一个对象经过多次复制依然存活时，它将会被认为是生命周期较长的对象。这种较长生命周期的对象随后会被移动到老生代中，采用新的算法进行管理。对象从新生代中移动到老生代中的过程称为晋升。**对象晋升的条件主要有两个，一个是对象是否经历过Scavenge回收，一个是To空间的内存占用比(25%)超过限制。**

Mark-Sweep & Mark-Compact

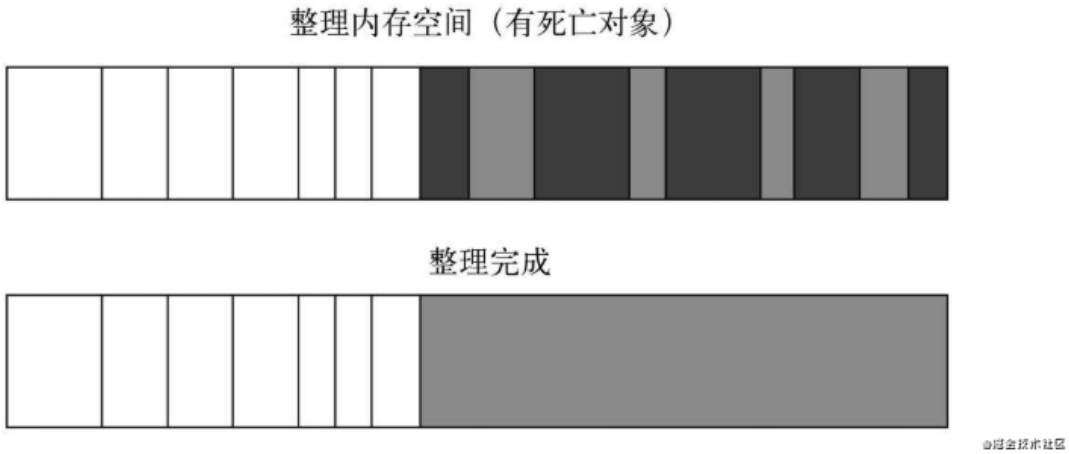
对于老生代中的对象，由于存活对象占较大比重，再采用Scavenge的方式会有两个问题：一个是存活对象较多，复制存活对象的效率将会很低；另一个问题依然是浪费一半空间的问题。这两个问题导致应对生命周期较长的对象时Scavenge会显得捉襟见肘。为此，**V8在老生代中主要采用了Mark-Sweep(标记清除)和Mark-Compact(标记整理)相结合的方式进行垃圾回收。**

- Mark-Sweep
  Mark-Sweep是标记清除的意思，它分为标记和清除两个阶段。与Scavenge相比，Mark-Sweep并不将内存空间划分为两半，所以不存在浪费一半空间的行为。与Scavenge复制活着的对象不同，**Mark-Sweep在标记阶段遍历堆中的所有对象，并标记活着的对象，在随后的清除阶段中，只清除没有被标记的对象。**可以看出，Scavenge中只复制活着的对象，而Mark-Sweep只清理死亡对象。活对象在新生代中只占较小部分，死对象在老生代中只占较小部分，这是两种回收方式能高效处理的原因。下图为Mark-Sweep在老生代空间中标记后的示意图，黑色部分标记为死亡的对象。



Mark-Sweep最大的问题是在进行一次标记清除回收后，内存空间会出现不连续的状态。这种内存碎片会对后续的内存分配造成问题，因为很可能出现需要分配一个大对象的情况，这时所有的碎片空间都无法完成此次分配，就会提前触发垃圾回收，而这次回收是不必要的。

- Mark-Compact
  为了解决Mark-Sweep的内存碎片问题，Mark-Compact被提出来。**Mark-Compact是标记整理的意思，是在Mark-Sweep的基础上演变而来的。它们的差别在于对象在标记为死亡后，在整理的过程中，将活着的对象往一端移动，移动完成后，直接清理掉边界外的内存。**下图为Mark-Compact完成标记并移动存活对象后的示意图，白色格子为存活对象，深色格子为死亡对象，浅色格子为存活对象移动后留下的空洞。



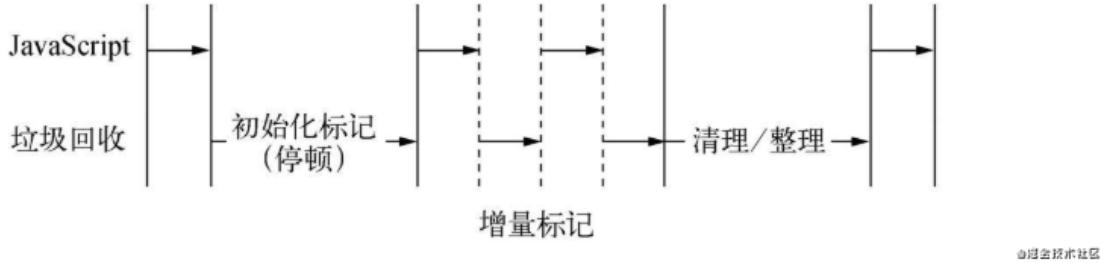
完成移动后，就可以直接清除最右边的存活对象后面的内存区域完成回收。

> 在Mark-Sweep和Mark-Compact之间，由于Mark-Compact需要移动对象，所以它的执行速度不可能很快，所以在取舍上，V8主要使用Mark-Sweep，在空间不足以对从新生代中晋升过来的对象进行分配时才使用Mark-Compact。

Incremental Marking

为了避免出现JavaScript应用逻辑与垃圾回收器看到的不一致的情况，垃圾回收的3种基本算法都需要将应用逻辑暂停下来，待执行完垃圾回收后再恢复执行应用逻辑，这种行为被称为“全停顿”（stop-the-world）。在V8的分代式垃圾回收中，一次小垃圾回收只收集新生代，由于新生代默认配置得较小，且其中存活对象通常较少，所以即便它是全停顿的影响也不大。但**V8的老生代通常配置得较大，且存活对象较多，全堆垃圾回收（full垃圾回收）的标记、清理、整理等动作造成的停顿就会比较可怕，需要设法改善。**

为了降低全堆垃圾回收带来的停顿时间，V8先从标记阶段入手，将原本要一口气停顿完成的动作改为增量标记（incremental marking），也就是拆分为许多小“步进”，每做完一“步进”就让JavaScript应用逻辑执行一小会儿，垃圾回收与应用逻辑交替执行直到标记阶段完成。



V8在经过增量标记的改进后，垃圾回收的最大停顿时间可以减少到原本的1/6左右。

> V8后续还引入了延迟清理（lazy sweeping）与增量式整理（incrementalcompaction），让清理与整理动作也变成增量式的。同时还计划引入并行标记与并行清理，进一步利用多核性能降低每次停顿的时间。

5.1.4 查看垃圾回收日志

查看垃圾回收日志的方式主要是在启动时添加--trace_gc参数。在进行垃圾回收时，将会从标准输出中打印垃圾回收的日志信息。下面是一段示例，执行结束后，将会在gc.log文件中得到所有垃圾回收信息：

```
node --trace_gc -e "var a = [];for (var i = 0; i < 1000000; i++) a.push(new
Array(100));" > gc.log
```

通过在Node启动时使用--prof参数，可以得到V8执行时的性能分析数据，其中包含了垃圾回收执行时占用的时间。

```
$ node --prof test01.js
```

5.2 内存指标

5.2.1 查看内存使用情况

查看进程的内存占用

调用process.memoryUsage()可以看到Node进程的内存占用情况，示例代码如下：

```
$ node
> process.memoryUsage()
{ rss: 13852672,
 heapTotal: 6131200,
 heapUsed: 2757120 }
```

rss是resident set size的缩写，即进程的常驻内存部分。进程的内存总共有几部分，一部分是rss，其余部分在交换区（swap）或者文件系统（filesystem）中。除了rss外，heapTotal和heapUsed对应的是V8的堆内存信息。**heapTotal是堆中总共申请的内存量，heapUsed表示目前堆中使用中的内存量。**这3个值的单位都是字节。

示例

为了更好地查看效果，我们格式化一下输出结果：

```
var showMem = function () {
 var mem = process.memoryUsage();
 var format = function (bytes) {
   return (bytes / 1024 / 1024).toFixed(2) + ' MB';
 };
 console.log('Process: heapTotal ' + format(mem.heapTotal) +
 ' heapUsed ' + format(mem.heapUsed) + ' rss ' + format(mem.rss));
 console.log('-----------------------------------------------------------');
};
```

同时，写一个方法用于不停地分配内存但不释放内存，相关代码如下：

```
var useMem = function () {
 var size = 20 * 1024 * 1024;
 var arr = new Array(size);
 for (var i = 0; i < size; i++) {
   arr[i] = 0;
 }
 return arr;
};
var total = [];
for (var j = 0; j < 15; j++) {
 showMem();
 total.push(useMem());
}
showMem();
```

将以上代码存为outofmemory.js并执行它，得到的输出结果如下：

```
$ node outofmemory.js
Process: heapTotal 3.86 MB heapUsed 2.10 MB rss 11.16 MB
-----------------------------------------------------------
Process: heapTotal 357.88 MB heapUsed 353.95 MB rss 365.44 MB
-----------------------------------------------------------
Process: heapTotal 520.88 MB heapUsed 513.94 MB rss 526.30 MB
-----------------------------------------------------------
Process: heapTotal 679.91 MB heapUsed 673.86 MB rss 686.14 MB
-----------------------------------------------------------
Process: heapTotal 839.93 MB heapUsed 833.86 MB rss 846.16 MB
-----------------------------------------------------------
Process: heapTotal 999.94 MB heapUsed 993.86 MB rss 1006.93 MB
-----------------------------------------------------------
Process: heapTotal 1159.96 MB heapUsed 1153.86 MB rss 1166.95 MB
-----------------------------------------------------------
Process: heapTotal 1367.99 MB heapUsed 1361.86 MB rss 1375.00 MB
-----------------------------------------------------------
FATAL ERROR: CALL_AND_RETRY_2 Allocation failed - process out of memory
```

可以看到，每次调用useMem都导致了3个值的增长。在接近1500 MB的时候，无法继续分配内存，然后进程内存溢出了，连循环体都无法执行完成，仅执行了7次。

查看系统的内存占用

与process.memoryUsage()不同的是，os模块中的totalmem()和freemem()这两个方法用于查看操作系统的内存使用情况，它们分别返回系统的总内存和闲置内存，以字节为单位。

```
$ node
> os.totalmem()
8589934592
> os.freemem()
4527833088
>
```

5.2.2 堆外内存

通过process.memoryUsage()的结果可以看到，堆中的内存用量总是小于进程的常驻内存用量，这意味着Node中的内存使用并非都是通过V8进行分配的。我们将那些不是通过V8分配的内存称为堆外内存。

将前面的useMem()方法稍微改造一下，将Array变为Buffer，将size变大，每一次构造200 MB的对象，相关代码如下：

```
var useMem = function () {
 var size = 200 * 1024 * 1024;
 var buffer = new Buffer(size);
 for (var i = 0; i < size; i++) {
   buffer[i] = 0;
 }
 return buffer;
};
```

重新执行该代码，得到的输出结果如下所示：

```
$ node out_of_heap.js
Process: heapTotal 3.86 MB heapUsed 2.07 MB rss 11.12 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.94 MB rss 212.88 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.95 MB rss 412.89 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.95 MB rss 612.89 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.92 MB rss 812.89 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.92 MB rss 1012.89 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.84 MB rss 1212.91 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.84 MB rss 1412.91 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.84 MB rss 1612.91 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.84 MB rss 1812.91 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.84 MB rss 2012.91 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.84 MB rss 2212.91 MB
-----------------------------------------------------------------
```

```
Process: heapTotal 5.85 MB heapUsed 1.84 MB rss 2412.91 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.85 MB rss 2612.91 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.85 MB rss 2812.91 MB
-----------------------------------------------------------------
Process: heapTotal 5.85 MB heapUsed 1.85 MB rss 3012.91 MB
-----------------------------------------------------------------
```

我们看到15次循环都完整执行，并且三个内存占用值与前一个示例完全不同。在改造后的输出结果中，heapTotal与heapUsed的变化极小，唯一变化的是rss的值，并且该值已经远远超过V8的限制值。**这其中的原因是Buffer对象不同于其他对象，它不经过V8的内存分配机制，所以也不会有堆内存的大小限制。这意味着利用堆外内存可以突破内存限制的问题。**

> 为何Buffer对象并非通过V8分配？这在于Node并不同于浏览器的应用场景。在浏览器中，JavaScript直接处理字符串即可满足绝大多数的业务需求，而Node则需要处理网络流和文件I/O流，操作字符串远远不能满足传输的性能需求。

5.2.3 小结

Node的内存构成主要由通过V8进行分配的部分和Node自行分配的部分。受V8的垃圾回收限制的主要是V8的堆内存。

5.3 内存泄漏

造成内存泄漏的原因有如下几个:

- 缓存。
- 队列消费不及时。
- 作用域未释放。

5.3.1 慎将内存当做缓存

缓存在应用中的作用举足轻重，可以十分有效地节省资源。因为它的访问效率要比I/O的效率高，一旦命中缓存，就可以节省一次I/O的时间。但是在Node中，缓存并非物美价廉。一旦一个对象被当做缓存来使用，那就意味着它将会常驻在老生代中。**缓存中存储的键越多，长期存活的对象也就越多，这将导致垃圾回收在进行扫描和整理时，对这些对象做无用功。**另一个问题在于，**JavaScript开发者通常喜欢用对象的键值对来缓存东西，但这与严格意义上的缓存又有着区别，严格意义的缓存有着完善的过期策略，而普通对象的键值对并没有。**

缓存限制策略

为了解决缓存中的对象永远无法释放的问题，需要加入一种策略来限制缓存的无限增长。

比如以下程序可以实现对键值数量的限制：

```
var LimitableMap = function (limit) {
 this.limit = limit || 10;
 this.map = {};
 this.keys = [];
};
var hasOwnProperty = Object.prototype.hasOwnProperty;
LimitableMap.prototype.set = function (key, value) {
 var map = this.map;
 var keys = this.keys;
 if (!hasOwnProperty.call(map, key)) {
```

```
    if (keys.length === this.limit) {
      var firstKey = keys.shift();
      delete map[firstKey];
    }
    keys.push(key);
  }
  map[key] = value;
};
LimitableMap.prototype.get = function (key) {
  return this.map[key];
};
module.exports = LimitableMap;
```

可以看到，实现过程还是非常简单的。记录键在数组中，一旦超过数量，就以先进先出的方式进行淘汰。

缓存的解决方案

直接将内存作为缓存的方案要十分慎重。除了限制缓存的大小外，另外要考虑的事情是，进程之间无法共享内存。如果在进程内使用缓存，这些缓存不可避免地有重复，对物理内存的使用是一种浪费。

如何使用大量缓存，目前比较好的解决方案是采用进程外的缓存，进程自身不存储状态。外部的缓存软件有着良好的缓存过期淘汰策略以及自有的内存管理，不影响Node进程的性能。它的好处多多，在Node中主要可以解决以下两个问题：

- 将缓存转移到外部，减少常驻内存的对象的数量，让垃圾回收更高效。
- 进程之间可以共享缓存。
  目前，市面上较好的缓存有Redis和Memcached。

5.3.2 关注队列状态

在JavaScript中可以通过队列（数组对象）来完成许多特殊的需求，比如Bagpipe。队列在消费者-生产者模型中经常充当中间产物。这是一个容易忽略的情况，因为**在大多数应用场景下，消费的速度远远大于生产的速度，内存泄漏不易产生。但是一旦消费速度低于生产速度，将会形成堆积。**

举个实际的例子，有的应用会收集日志。如果欠缺考虑，也许会采用数据库来记录日志。**日志通常会是海量的，数据库构建在文件系统之上，写入效率远远低于文件直接写入，于是会形成数据库写入操作的堆积，而JavaScript中相关的作用域也不会得到释放，内存占用不会回落，从而出现内存泄漏。**

遇到这种场景，表层的解决方案是换用消费速度更高的技术。在日志收集的案例中，换用文件写入日志的方式会更高效。需要注意的是，如果生产速度因为某些原因突然激增，或者消费速度因为突然的系统故障降低，内存泄漏还是可能出现的。

深度的解决方案应该是监控队列的长度，一旦堆积，应当通过监控系统产生报警并通知相关人员。另一个解决方案是任意异步调用都应该包含超时机制，一旦在限定的时间内未完成响应，通过回调函数传递超时异常，使得任意异步调用的回调都具备可控的响应时间，给消费速度一个下限值。

对于Bagpipe而言，它提供了超时模式和拒绝模式。启用超时模式时，调用加入到队列中就开始计时，超时就直接响应一个超时错误。启用拒绝模式时，当队列拥塞时，新到来的调用会直接响应拥塞错误。这两种模式都能够有效地防止队列拥塞导致的内存泄漏问题。

5.4 大内存应用

在Node中，不可避免地还是会存在操作大文件的场景。由于Node的内存限制，操作大文件也需要小心，好在**Node提供了stream模块用于处理大文件。**

stream继承自EventEmitter，具备基本的自定义事件功能，同时抽象出标准的事件和方法。它分可读和可写两种。Node中的大多数模块都有stream的应用，比如fs的createReadStream()和createWriteStream()方法可以分别用于创建文件的可读流和可写流，process模块中的stdin和stdout则分别是可读流和可写流的示例。**由于V8的内存限制，我们无法通过fs.readFile()和fs.writeFile()直接进行大文件的操作，而改用fs.createReadStream()和fs.createWriteStream()方法通过流的方式实现对大文件的操作。**下面的代码展示了如何读取一个文件，然后将数据写入到另一个文件的过程：

```
var reader = fs.createReadStream('in.txt');
var writer = fs.createWriteStream('out.txt');
reader.on('data', function (chunk) {
 writer.write(chunk);
});
reader.on('end', function () {
 writer.end();
});
```

由于读写模型固定，上述方法有更简洁的方式，具体如下所示：

```
var reader = fs.createReadStream('in.txt');
var writer = fs.createWriteStream('out.txt');
reader.pipe(writer);
```

可读流提供了管道方法pipe()，封装了data事件和写入操作。通过流的方式，上述代码不会受到V8内存限制的影响，有效地提高了程序的健壮性。

> 如果不需要进行字符串层面的操作，则不需要借助V8来处理，可以尝试进行纯粹的Buffer操作，这不会受到V8堆内存的限制。但是这种大片使用内存的情况依然要小心，即使V8不限制堆内存的大小，物理内存依然有限制。

六、理解Buffer

JavaScript对于字符串（string）的操作十分友好，无论是宽字节字符串还是单字节字符串，都被认为是一个字符串。

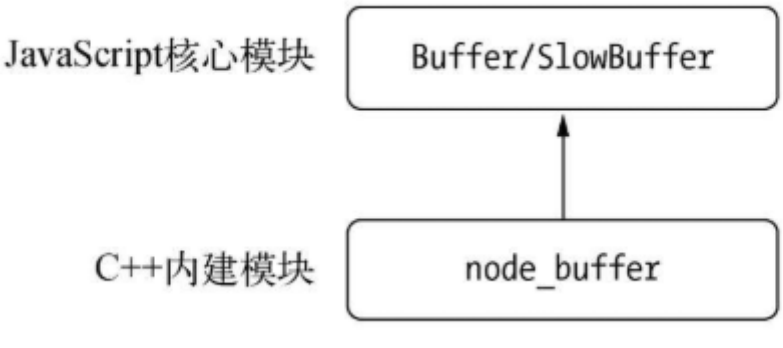
由于应用场景不同，在Node中，应用需要处理网络协议、操作数据库、处理图片、接收上传文件等，在网络流和文件的操作中，还要处理大量二进制数据，**JavaScript自有的字符串远远不能满足这些需求，于是Buffer对象应运而生。**

6.1 Buffer结构

Buffer是一个像Array的对象，但它主要用于操作字节。

6.1.1 模块结构

Buffer是一个典型的JavaScript与C++结合的模块，它将性能相关部分用C++实现，将非性能相关的部分用JavaScript实现。



6.1.2 Buffer对象

Buffer对象类似于数组，它的元素为16进制的两位数，即0到255的数值。

```
// 中文字在UTF-8编码下占用3个元素，字母和半角标点符号占用1个元素。
var str = "深入浅出node.js";
var buf = new Buffer(str, 'utf-8');
console.log(buf);
// => <Buffer e6 b7 b1 e5 85 a5 e6 b5 85 e5 87 ba 6e 6f 64 65 2e 6a 73>
```

6.1.3 Buffer内存分配

Buffer对象的内存分配不是在V8的堆内存中，而是在Node的C++层面实现内存的申请的。因为处理大量的字节数据不能采用需要一点内存就向操作系统申请一点内存的方式，这可能造成大量的内存申请的系统调用，对操作系统有一定压力。为此Node在内存的使用上应用的是在C++层面申请内存、在JavaScript中分配内存的策略。

为了高效地使用申请来的内存，Node采用了slab分配机制。slab是一种动态内存管理机制，最早诞生于SunOS操作系统（Solaris）中，目前在一些*nix操作系统中有广泛的应用，如FreeBSD和Linux。

简单而言，**slab就是一块申请好的固定大小的内存区域。**slab具有如下3种状态:

- full：完全分配状态。
- partial：部分分配状态。
- empty：没有被分配状态。
  Node以8 KB为界限来区分Buffer是大对象还是小对象

```
Buffer.poolSize = 8 * 1024;
```

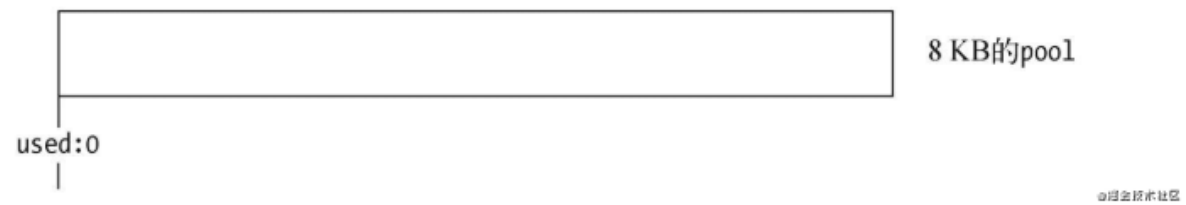
这个8 KB的值也就是每个slab的大小值，在JavaScript层面，以它作为单位单元进行内存的分配。

分配小Buffer对象

如果指定Buffer的大小少于8 KB，Node会按照小对象的方式进行分配。Buffer的分配过程中主要使用一个局部变量pool作为中间处理对象，处于分配状态的slab单元都指向它。以下是分配一个全新的slab单元的操作，它会将新申请的SlowBuffer对象指向它：

```
var pool;
function allocPool() {
 pool = new SlowBuffer(Buffer.poolSize);
 pool.used = 0;
}
```

下图为一个新构造的slab单元示例，slab处于empty状态：



构造小Buffer对象时的代码如下：

```
new Buffer(1024);
```

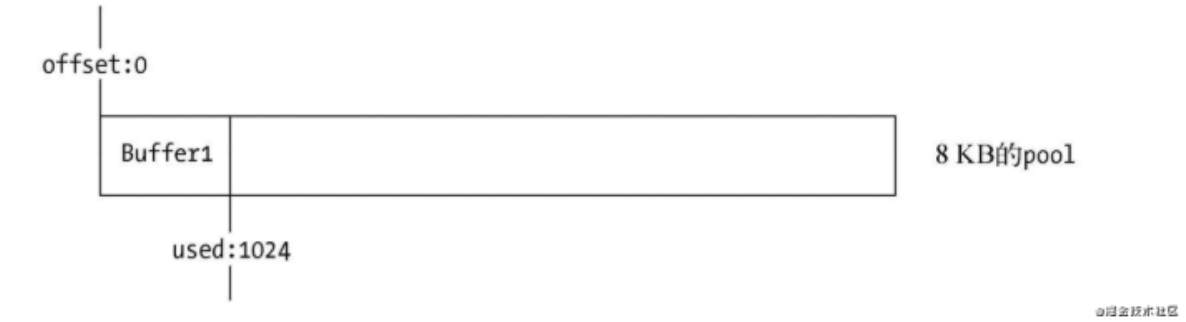
这次构造将会去检查pool对象，**如果pool没有被创建，将会创建一个新的slab单元指向它**：

```
if (!pool || pool.length - pool.used < this.length) allocPool();
```

同时当前Buffer对象的parent属性指向该slab，并记录下是从这个slab的哪个位置（offset）开始使用的，slab对象自身也记录被使用了多少字节，代码如下：

```
this.parent = pool;
this.offset = pool.used;
pool.used += this.length;
if (pool.used & 7) pool.used = (pool.used + 8) & ~7;
```

下图为从一个新的slab单元中初次分配一个Buffer对象的示意图：



这时候的slab状态为partial。

当再次创建一个Buffer对象时，构造过程中将会判断这个slab的剩余空间是否足够。如果足够，使用剩余空间，并更新slab的分配状态。下面的代码创建了一个新的Buffer对象，它会引起一次slab分配：

```
new Buffer(3000);
```

如果slab剩余的空间不够，将会构造新的slab，原slab中剩余的空间会造成浪费。例如，第一次构造1字节的Buffer对象，第二次构造8192字节的Buffer对象，由于第二次分配时slab中的空间不够，所以创建并使用新的slab，第一个slab的8 KB将会被第一个1字节的Buffer对象独占。下面的代码一共使用了两个slab单元：

```
new Buffer(1);
new Buffer(8192);
```

> 由于同一个slab可能分配给多个Buffer对象使用，只有这些小Buffer对象在作用域释放并都可以回收时，slab的8 KB空间才会被回收。尽管创建了1个字节的Buffer对象，但是如果不释放它，实际可能是8 KB的内存没有释放。

分配大Buffer对象

如果需要超过8 KB的Buffer对象，将会直接分配一个SlowBuffer对象作为slab单元，这个slab单元将会被这个大Buffer对象独占.

```
// Big buffer, just alloc one
this.parent = new SlowBuffer(this.length);
this.offset = 0;
```

这里的SlowBuffer类是在C++中定义的，虽然引用buffer模块可以访问到它，但是不推荐直接操作它，而是用Buffer替代。

小结

简单而言，真正的内存是在Node的C++层面提供的，JavaScript层面只是使用它。当进行小而频繁的Buffer操作时，采用slab的机制进行预先申请和事后分配，使得JavaScript到操作系统之间不必有过多的内存申请方面的系统调用。对于大块的Buffer而言，则直接使用C++层面提供的内存，而无需细腻的分配操作。

6.2 Buffer的转换

6.2.1 字符串转Buffer

字符串转Buffer对象主要是通过构造函数完成的：

```
new Buffer(str, [encoding]);
```

通过构造函数转换的Buffer对象，存储的只能是一种编码类型。encoding参数不传递时，默认按UTF-8编码进行转码和存储。

一个Buffer对象可以存储不同编码类型的字符串转码的值，调用write()方法可以实现该目的，代码如下：

```
buf.write(string, [offset], [length], [encoding])
```

由于可以不断写入内容到Buffer对象中，并且每次写入可以指定编码，所以Buffer对象中可以存在多种编码转化后的内容。需要小心的是，每种编码所用的字节长度不同，将Buffer反转回字符串时需要谨慎处理。

6.2.2 Buffer转字符串

实现Buffer向字符串的转换也十分简单，Buffer对象的toString()可以将Buffer对象转换为字符串，代码如下：

```
buf.toString([encoding], [start], [end])
```

比较精巧的是，可以设置encoding（默认为UTF-8）、start、end这3个参数实现整体或局部的转换。如果Buffer对象由多种编码写入，就需要在局部指定不同的编码，才能转换回正常的编码。

6.3 Buffer的拼接

Buffer在使用场景中，通常是以一段一段的方式传输。以下是常见的从输入流中读取内容的示例代码：

```
var fs = require('fs');
var rs = fs.createReadStream('test.md');
var data = '';
rs.on("data", function (chunk){
 data += chunk;
});
rs.on("end", function () {
 console.log(data);
});
```

上面这段代码常见于国外，用于流读取的示范，**data事件中获取的chunk对象即是Buffer对象。**

一旦输入流中有宽字节编码时，就会出现问题。如果你在通过Node开发的网站上看到乱码符号，那么该问题的起源多半来自于这里。

这里潜藏的问题在于如下这句代码：

```
data += chunk;
```

这句代码里隐藏了toString()操作，它等价于如下的代码：

```
data = data.toString() + chunk.toString();
```

在英文环境下，这个toString()不会造成任何问题。但对于宽字节的中文，却会形成问题。为了重现这个问题，下面通过模拟近似的场景，将文件可读流的每次读取的Buffer长度限制为11，代码如下：

```
var rs = fs.createReadStream('test.md', {highWaterMark: 11});
```

搭配该代码的测试数据为李白的《静夜思》。执行该程序，将会得到以下输出：

```
床前明月���光，疑���地上霜；举头���明月，���头思故乡。
```

6.3.1 乱码是如何产生的

上面的诗歌中，“月”、“是”、“望”、“低”4个字没有被正常输出，取而代之的是3个�。产生这个输出结果的原因在于文件可读流在读取时会逐个读取Buffer。这首诗的原始Buffer应存储为：

```
<Buffer e5 ba 8a e5 89 8d e6 98 8e e6 9c 88 e5 85 89 ef bc 8c e7 96 91 e6 98 af
e5 9c b0 e4 b8 8a e9
9c 9c ef bc 9b e4 b8 be e5 a4 b4 e6 9c 9b e6 98 8e e6 9c 88 ...>
```

由于我们限定了Buffer对象的长度为11，因此只读流需要读取7次才能完成完整的读取，结果是以下几个Buffer对象依次输出：

```
<Buffer e5 ba 8a e5 89 8d e6 98 8e e6 9c>
<Buffer 88 e5 85 89 ef bc 8c e7 96 91 e6>
...
```

上文提到的buf.toString()方法默认以UTF-8为编码，中文字在UTF-8下占3个字节。所以第一个Buffer对象在输出时，只能显示3个字符，Buffer中剩下的2个字节（e6 9c）将会以乱码的形式显示。第二个Buffer对象的第一个字节也不能形成文字，只能显示乱码。于是形成一些文字无法正常显示的问题。

> 对于任意长度的Buffer而言，宽字节字符串都有可能存在被截断的情况，只不过Buffer的长度越大出现的概率越低而已，但该问题依然不可忽视。

6.3.2 setEncoding()与string_decoder()

可读流还有一个设置编码的方法setEncoding()，示例如下：

```
readable.setEncoding(encoding)
```

该方法的作用是让data事件中传递的不再是一个Buffer对象，而是编码后的字符串。为此，我们继续改进前面诗歌的程序，添加setEncoding()的步骤如下：

```
var rs = fs.createReadStream('test.md', { highWaterMark: 11});
rs.setEncoding('utf8');
```

重新执行程序，得到输出：

```
床前明月光，疑是地上霜；举头望明月，低头思故乡。
```

无论如何设置编码，触发data事件的次数依旧相同，这意味着设置编码并未改变按段读取的基本方式。**事实上，在调用setEncoding()时，可读流对象在内部设置了一个decoder对象。每次data事件都通过该decoder对象进行Buffer到字符串的解码，然后传递给调用者。是故设置编码后，data不再收到原始的Buffer对象。但是这依旧无法解释为何设置编码后乱码问题被解决掉了，因为在前述分析中，无论如何转码，总是存在宽字节字符串被截断的问题。最终乱码问题得以解决，还是在于decoder的神奇之处。**decoder对象来自于string_decoder模块StringDecoder的实例对象。它神奇的原理是什么，下面我们以代码来说明：

```
var StringDecoder = require('string_decoder').StringDecoder;
var decoder = new StringDecoder('utf8');
var buf1 = new Buffer([0xE5, 0xBA, 0x8A, 0xE5, 0x89, 0x8D, 0xE6, 0x98, 0x8E,
0xE6, 0x9C]);
console.log(decoder.write(buf1));
//=>床前明
var buf2 = new Buffer([0x88, 0xE5, 0x85, 0x89, 0xEF, 0xBC, 0x8C, 0xE7, 0x96,
0x91, 0xE6]);
console.log(decoder.write(buf2));
// => 月光，疑
```

我将前文提到的前两个Buffer对象写入decoder中。奇怪的地方在于“月”的转码并没有如平常一样在两个部分分开输出。**StringDecoder在得到编码后，知道宽字节字符串在UTF-8编码下是以3个字节的方式存储的，所以第一次write()时，只输出前9个字节转码形成的字符，“月”字的前两个字节被保留在StringDecoder实例内部。第二次write()时，会将这2个剩余字节和后续11个字节组合在一起，再次用3的整数倍字节进行转码。于是乱码问题通过这种中间形式被解决了。**

> 虽然string_decoder模块很奇妙，但是它也并非万能药，它目前只能处理UTF-8、Base64和UCS-2/UTF-16LE这3种编码。所以，通过setEncoding()的方式不可否认能解决大部分的乱码问题，但并不能从根本上解决该问题。

6.3.3 正确拼接Buffer

淘汰掉setEncoding()方法后，剩下的解决方案只有将多个小Buffer对象拼接为一个Buffer对象，然后通过iconv-lite一类的模块来转码这种方式。+=的方式显然不行，那么正确的Buffer拼接方法应该如下面展示的形式：

```
var chunks = [];
var size = 0;
res.on('data', function (chunk) {
 chunks.push(chunk);
 size += chunk.length;
});
res.on('end', function () {
 var buf = Buffer.concat(chunks, size);
 var str = iconv.decode(buf, 'utf8');
 console.log(str);
});
```

正确的拼接方式是用一个数组来存储接收到的所有Buffer片段并记录下所有片段的总长度，然后调用Buffer.concat()方法生成一个合并的Buffer对象。

6.4 Buffer与性能

在Web应用中，字符串转换到Buffer是时时刻刻发生的，提高字符串到Buffer的转换效率，可以很大程度地提高网络吞吐率。

通过预先转换静态内容为Buffer对象，可以有效地减少CPU的重复使用，节省服务器资源。在Node构建的Web应用中，可以选择将页面中的动态内容和静态内容分离，静态内容部分可以通过预先转换为Buffer的方式，使性能得到提升。由于文件自身是二进制数据，所以在不需要改变内容的场景下，尽量只读取Buffer，然后直接传输，不做额外的转换，避免损耗。

文件读取

Buffer的使用除了与字符串的转换有性能损耗外，在文件的读取时，**有一个highWaterMark设置对性能的影响至关重要。**在fs.createReadStream(path,opts)时，我们可以传入一些参数，代码如下：

```
{
 flags: 'r',
 encoding: null,
 fd: null,
 mode: 0666,
 highWaterMark: 64 * 1024
}
```

fs.createReadStream()的工作方式是在内存中准备一段Buffer，然后在fs.read()读取时逐步从磁盘中将字节复制到Buffer中。完成一次读取时，则从这个Buffer中通过slice()方法取出部分数据作为一个小Buffer对象，再通过data事件传递给调用方。如果Buffer用完，则重新分配一个；如果还有剩余，则继续使用。下面为分配一个新的Buffer对象的操作：

```
var pool;
function allocNewPool(poolSize) {
 pool = new Buffer(poolSize);
 pool.used = 0;
}
```

在理想的状况下，每次读取的长度就是用户指定的highWaterMark。但是有可能读到了文件结尾，或者文件本身就没有指定的highWaterMark那么大，这个预先指定的Buffer对象将会有部分剩余，不过好在这里的内存可以分配给下次读取时使用。pool是常驻内存的，只有当pool单元剩余数量小于128（kMinPoolSpace）字节时，才会重新分配一个新的Buffer对象。Node源代码中分配新的Buffer对象的判断条件如下所示：

```
if (!pool || pool.length - pool.used < kMinPoolSpace) {
 // discard the old pool
 pool = null;
 allocNewPool(this._readableState.highWaterMark);
}
```

这里与Buffer的内存分配比较类似，highWaterMark的大小对性能有两个影响的点：

- **highWaterMark设置对Buffer内存的分配和使用有一定影响。**
- **highWaterMark设置过小，可能导致系统调用次数过多。**

> 文件流读取基于Buffer分配，Buffer则基于SlowBuffer分配，这可以理解为两个维度的分配策略。如果文件较小（小于8 KB），有可能造成slab未能完全使用。由于fs.createReadStream()内部采用fs.read()实现，将会引起对磁盘的系统调用，对于大文件而言，highWaterMark的大小决定会触发系统调用和data事件的次数。
> **注：读取一个相同的大文件时，highWaterMark值的大小与读取速度的关系：该值越大，读取速度越快。**

七、网络编程

待更新。。。

八、构建Web应用

待更新。。。

九、玩转进程

9.1 服务模型的变迁

9.1.1 石器时代：同步

最早的服务器，其执行模型是同步的，它的服务模式是一次只为一个请求服务，所有请求都得按次序等待服务。这意味除了当前的请求被处理外，其余请求都处于耽误的状态。它的处理能力相当低下，假设每次响应服务耗用的时间稳定为N秒，这类服务的QPS为1/N。

9.1.2 青铜时代：复制进程

为了解决同步架构的并发问题，一个简单的改进是通过进程的复制同时服务更多的请求和用户。在进程复制的过程中，需要复制进程内部的状态，对于每个连接都进行这样的复制的话，相同的状态将会在内存中存在很多份，造成浪费。并且这个过程由于要复制较多的数据，启动是较为缓慢的。

9.1.3 白银时代：多线程

为了解决进程复制中的浪费问题，多线程被引入服务模型，让一个线程服务一个请求。线程相对进程的开销要小许多，并且线程之间可以共享数据，内存浪费的问题可以得到解决，并且利用线程池可以减少创建和销毁线程的开销。但是因为每个线程都拥有自己独立的堆栈，这个堆栈都需要占用一定的内存空间。另外，由于一个CPU核心在一个时刻只能做一件事情，操作系统只能通过将CPU切分为时间片的方法，让线程可以较为均匀地使用CPU资源，但是操作系统内核在切换线程的同时也要切换线程的上下文，当线程数量过多时，时间将会被耗用在上下文切换中。所以在大并发量时，多线程结构还是无法做到强大的伸缩性。

9.1.4 黄金时代：事件驱动

为了解决高并发问题，基于事件驱动的服务模型出现了，像Node与Nginx均是基于事件驱动的方式实现的，采用单线程避免了不必要的内存开销和上下文切换开销。**基于事件的服务模型存在两个问题：CPU的利用率和进程的健壮性。**

9.2 多进程架构

Node提供了`child_process`模块，并且也提供了`child_process.fork()`函数供我们实现进程的复制。

我们再一次将经典的示例代码存为worker.js文件，如下所示：

```
var http = require('http');
http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': 'text/plain'});
 res.end('Hello World\n');
}).listen(Math.round((1 + Math.random()) * 1000), '127.0.0.1');
```

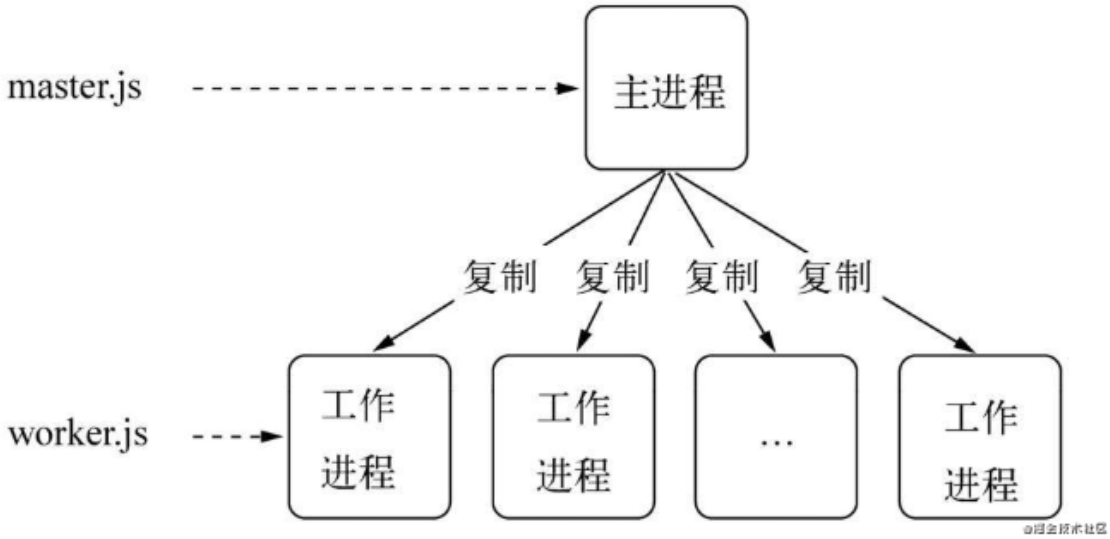
通过node worker.js启动它，将会侦听1000到2000之间的一个随机端口。将以下代码存为master.js，并通过node master.js启动它：

```
var fork = require('child_process').fork;
var cpus = require('os').cpus();
for (var i = 0; i < cpus.length; i++) {
 fork('./worker.js');
}
```

这段代码将会根据当前机器上的CPU数量复制出对应Node进程数。

9.2.1 Master-Worker模式

著名的Master-Worker模式，又称主从模式。进程分为两种：**主进程和工作进程。**这是典型的分布式架构中用于并行处理业务的模式，具备较好的可伸缩性和稳定性。**主进程不负责具体的业务处理，而是负责调度或管理工作进程，它是趋向于稳定的。工作进程负责具体的业务处理，因为业务的多种多样，甚至一项业务由多人开发完成，所以工作进程的稳定性值得开发者关注。**



> 通过fork()复制的进程都是一个独立的进程，这个进程中有着独立而全新的V8实例。它需要至少30毫秒的启动时间和至少10 MB的内存。尽管Node提供了fork()供我们复制进程使每个CPU内核都使用上，但是依然要切记fork()进程是昂贵的。好在Node通过事件驱动的方式在单线程上解决了大并发的问题，这里启动多个进程只是为了充分将CPU资源利用起来，而不是为了解决并发问题。

9.2.2 创建子进程

child_process模块提供了4个方法用于创建子进程:

- spawn()：**启动一个子进程来执行命令。**
- exec()：启动一个子进程来执行命令，**与spawn()不同的是其接口不同，它有一个回调函数获知子进程的状况。**
- execFile()：启动一个子进程来**执行可执行文件。**
- fork()：与spawn()类似，**不同点在于它创建Node的子进程只需指定要执行的JavaScript文件模块即可。**
  spawn()与exec()、execFile()不同的是，**后两者创建时可以指定timeout属性设置超时时间，一旦创建的进程运行超过设定的时间将会被杀死。**exec()与execFile()不同的是，**exec()适合执行已有的命令，execFile()适合执行文件。**

```
var cp = require('child_process');
cp.spawn('node', ['worker.js']);
cp.exec('node worker.js', function (err, stdout, stderr) {
 // some code
});
cp.execFile('worker.js', function (err, stdout, stderr) {
 // some code
});
cp.fork('./worker.js');
```

类　型	回调/异常	进程类型	执行类型	可设置超时
spawn()	×	任意	命令	×
exec()	√	任意	命令	√
execFile()	√	任意	可执行文件	√
fork()	×	Node	JavaScript文件	×

> 尽管4种创建子进程的方式有些差别，但事实上后面3种方法都是spawn()的延伸应用。

9.2.3 进程间通信

在Master-Worker模式中，要实现主进程管理和调度工作进程的功能，需要主进程和工作进程之间的通信。

WebWorker线程间通信

WebWorker允许创建工作线程并在后台运行，使得一些阻塞较为严重的计算不影响主线程上的UI渲染。

```
var worker = new Worker('worker.js');
worker.onmessage = function (event) {
 document.getElementById('result').textContent = event.data;
};
```

其中，worker.js如下所示：

```
var n = 1;
search: while (true) {
 n += 1;
 for (var i = 2; i <= Math.sqrt(n); i += 1) {
   if (n i == 0) %
   continue search;
   // found a prime
   postMessage(n);
 }
}
```

主线程与工作线程之间通过onmessage()和postMessage()进行通信。通过消息传递内容，而不是共享或直接操作相关资源，这是较为轻量和无依赖的做法。

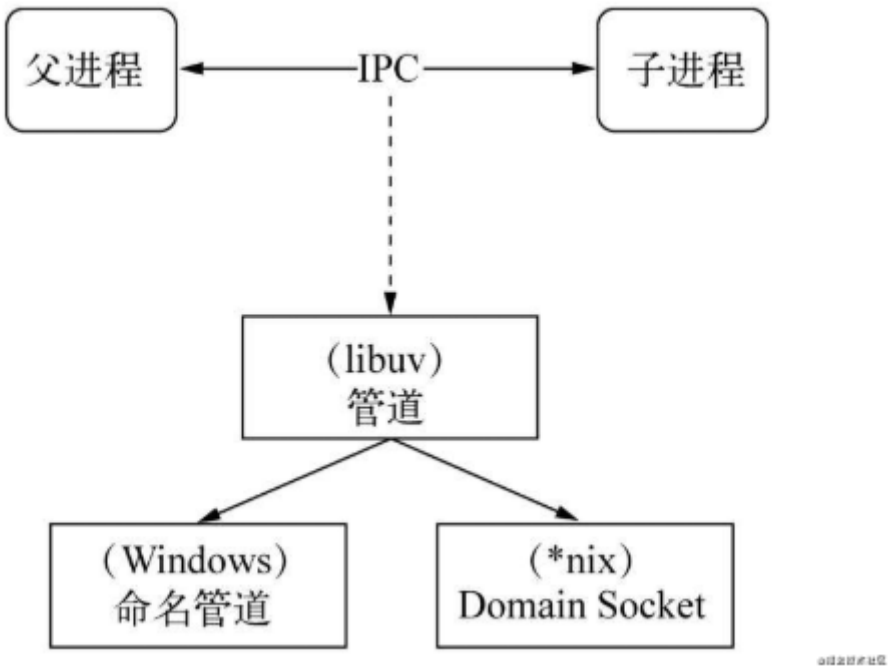
Node进程间通信

Node中，子进程对象则由send()方法实现主进程向子进程发送数据，message事件实现收听子进程发来的数据，与WebWorker API在一定程度上相似。

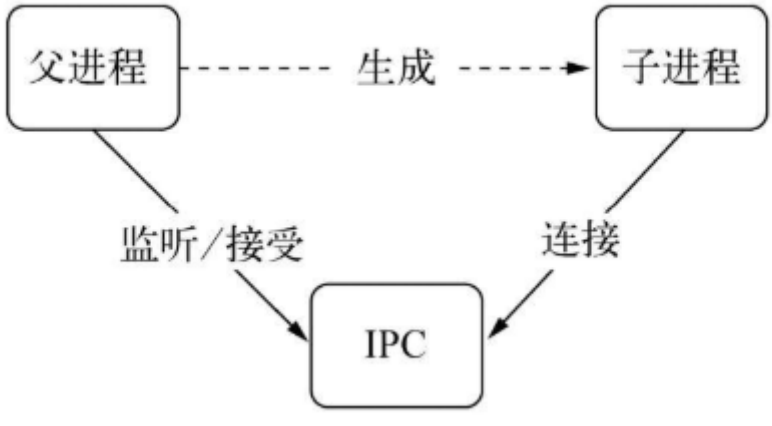
```
// parent.js
var cp = require('child_process');
var n = cp.fork(__dirname + '/sub.js');
n.on('message', function (m) {
 console.log('PARENT got message:', m);
});
n.send({hello: 'world'});

// sub.js
process.on('message', function (m) {
 console.log('CHILD got message:', m);
});
process.send({foo: 'bar'});
```

Node进程间通信原理

> 通过fork()或者其他API，创建子进程之后，为了实现父子进程之间的通信，父进程与子进程之间将会创建IPC通道。通过IPC通道，父子进程之间才能通过message和send()传递消息。
> IPC的全称是Inter-Process Communication，即进程间通信。**进程间通信的目的是为了让不同的进程能够互相访问资源并进行协调工作。**实现进程间通信的技术有很多，如命名管道、匿名管道、socket、信号量、共享内存、消息队列、DomainSocket等。**Node中实现IPC通道的是管道（pipe）技术。**但此管道非彼管道，在Node中管道是个抽象层面的称呼，**具体细节实现由libuv提供，在Windows下由命名管道（named pipe）实现，*nix系统则采用Unix Domain Socket实现。**表现在应用层上的进程间通信只有简单的message事件和send()方法，接口十分简洁和消息化。下图为IPC创建和实现的示意图：



父进程在实际创建子进程之前，会创建IPC通道并监听它，然后才真正创建出子进程，并通过环境变量（NODE_CHANNEL_FD）告诉子进程这个IPC通道的文件描述符。子进程在启动的过程中，根据文件描述符去连接这个已存在的IPC通道，从而完成父子进程之间的连接。下图为创建IPC管道的步骤示意图：

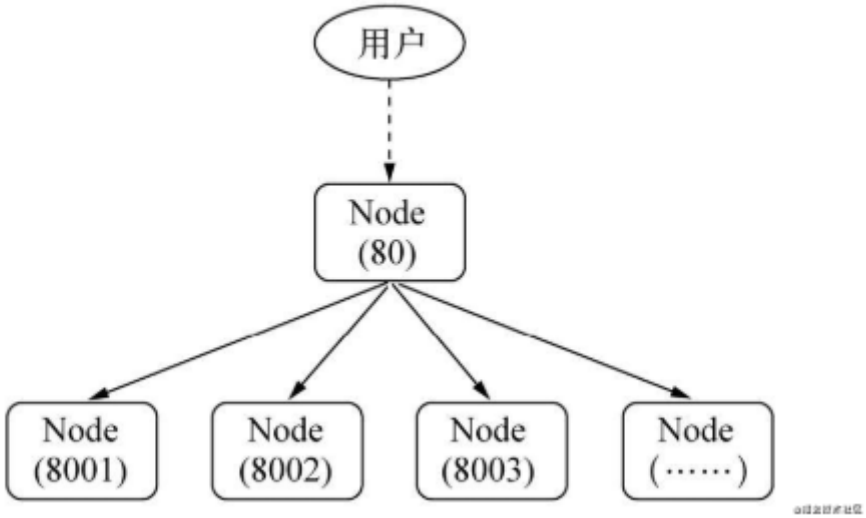


建立连接之后的父子进程就可以自由地通信了。**由于IPC通道是用命名管道或Domain Socket创建的，它们与网络socket的行为比较类似，属于双向通信。不同的是它们在系统内核中就完成了进程间的通信，而不用经过实际的网络层，非常高效。在Node中，IPC通道被抽象为Stream对象，在调用send()时发送数据（类似于write()），接收到的消息会通过message事件（类似于data）触发给应用层。**

> 只有启动的子进程是Node进程时，子进程才会根据环境变量去连接IPC通道，对于其他类型的子进程则无法实现进程间通信，除非其他进程也按约定去连接这个已经创建好的IPC通道。

9.2.4 句柄传递

当我们想让服务都监听到相同的端口时，这时只有一个工作进程能够监听到该端口上，其余的进程在监听的过程中都抛出了EADDRINUSE异常，这是端口被占用的情况，新的进程不能继续监听该端口了。要解决这个问题，通常的做法是**让每个进程监听不同的端口，其中主进程监听主端口（如80），主进程对外接收所有的网络请求，再将这些请求分别代理到不同的端口的进程上。**



> 通过代理，可以避免端口不能重复监听的问题，甚至可以在代理进程上做适当的负载均衡，使得每个子进程可以较为均衡地执行任务。**由于进程每接收到一个连接，将会用掉一个文件描述符，因此代理方案中客户端连接到代理进程，代理进程连接到工作进程的过程需要用掉两个文件描述符。操作系统的文件描述符是有限的，代理方案浪费掉一倍数量的文件描述符的做法影响了系统的扩展能力。**
> 为了解决上述这样的问题，Node在版本v0.5.9引入了进程间发送句柄的功能。send()方法除了能通过IPC发送数据外，还能发送句柄，第二个可选参数就是句柄，如下所示：

```
child.send(message, [sendHandle])
```

> 句柄是一种可以用来标识资源的引用，它的内部包含了指向对象的文件描述符。比如句柄可以用来标识一个服务器端socket对象、一个客户端socket对象、一个UDP套接字、一个管道等。
> 发送句柄意味着什么？在前一个问题中，我们**可以去掉代理这种方案，使主进程接收到socket请求后，将这个socket直接发送给工作进程，而不是重新与工作进程之间建立新的socket连接来转发数据。文件描述符浪费的问题可以通过这样的方式轻松解决。**

示例

主进程代码如下所示：

```
var child = require('child_process').fork('child.js');
// Open up the server object and send the handle
var server = require('net').createServer();
server.on('connection', function (socket) {
 socket.end('handled by parent\n');
});
server.listen(1337, function () {
 child.send('server', server);
});
```

子进程代码如下所示：

```
process.on('message', function (m, server) {
 if (m === 'server') {
   server.on('connection', function (socket) {
     socket.end('handled by child\n');
   });
 }
});
```

这个示例中直接将一个TCP服务器发送给了子进程。这是看起来不可思议的事情，我们先来测试一番，看看效果如何，如下所示：

```
// 先启动服务器
$ node parent.js
```

然后新开一个命令行窗口，用上curl工具，如下所示：

```
$ curl "http://127.0.0.1:1337/"
handled by parent
$ curl "http://127.0.0.1:1337/"
handled by child
$ curl "http://127.0.0.1:1337/"
handled by child
$ curl "http://127.0.0.1:1337/"
handled by parent
```

命令行中的响应结果也是很不可思议的，这里子进程和父进程都有可能处理我们客户端发起的请求。

对于主进程而言，我们甚至想要它更轻量一点，所以在将服务器句柄发送给子进程之后，就可以关掉服务器的监听，让子进程来处理请求。
对主进程进行改动：

```
// parent.js
var cp = require('child_process');
var child1 = cp.fork('child.js');
var child2 = cp.fork('child.js');
// Open up the server object and send the handle
var server = require('net').createServer();
server.listen(1337, function () {
 child1.send('server', server);
 child2.send('server', server);
 // 关掉
 server.close();
});
```

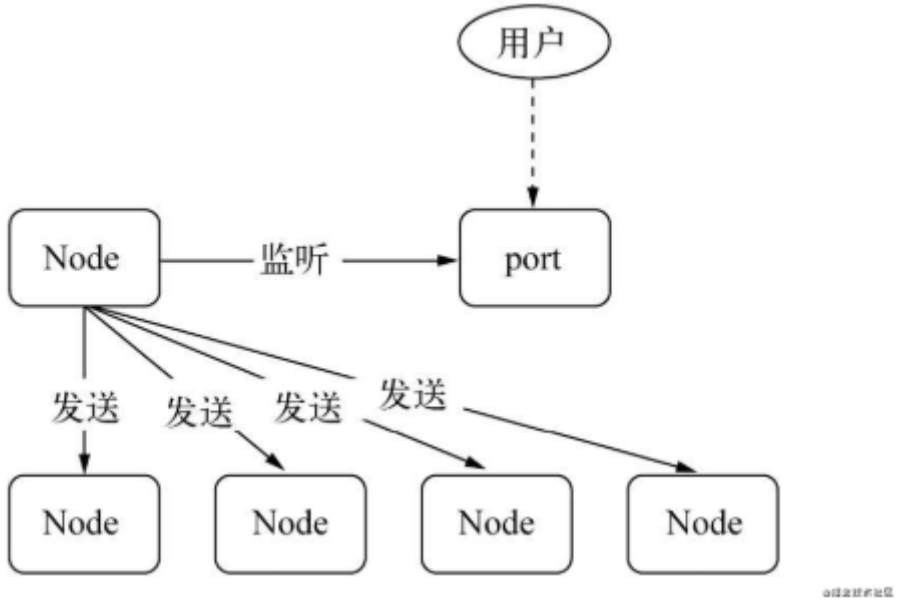
然后对子进程进行改动：

```
// child.js
var http = require('http');
var server = http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': 'text/plain'});
 res.end('handled by child, pid is ' + process.pid + '\n');
});
process.on('message', function (m, tcp) {
 if (m === 'server') {
   tcp.on('connection', function (socket) {
     server.emit('connection', socket);
   });
 }
});
```

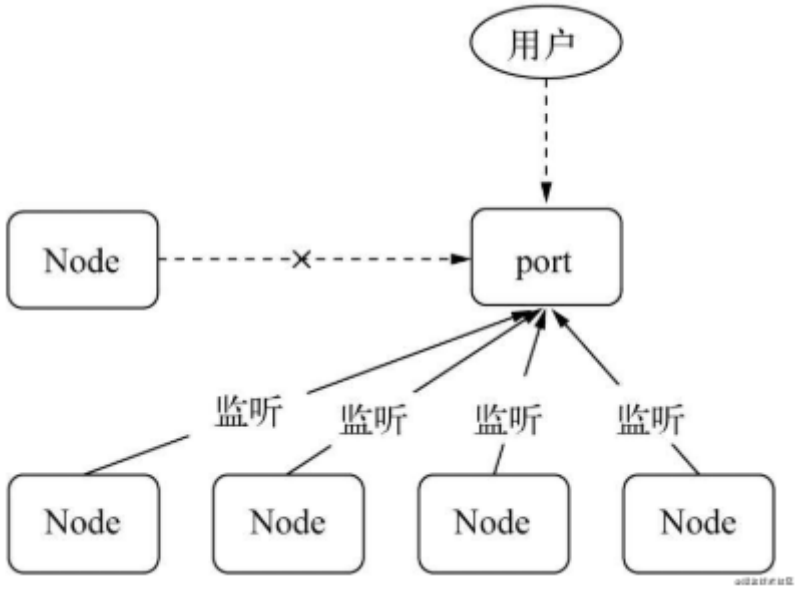
重新启动parent.js后，再次测试，如下所示：

```
$ curl "http://127.0.0.1:1337/"
handled by child, pid is 24852
$ curl "http://127.0.0.1:1337/"
handled by child, pid is 24851
```

这样一来，所有的请求都是由子进程处理了。整个过程中，服务的过程发生了一次改变，如图所示：



主进程发送完句柄并关闭监听之后，成为了如图所示的结构：



这时，多个子进程可以同时监听相同端口，再没有EADDRINUSE异常发生了。

句柄发送与还原

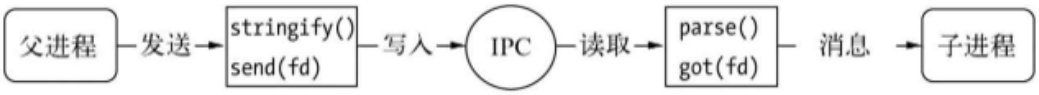
目前子进程对象send()方法可以发送的句柄类型包括如下几种:

- net.Socket。TCP套接字。
- net.Server。TCP服务器，任意建立在TCP服务上的应用层服务都可以享受到它带来的好处。
- net.Native。C++层面的TCP套接字或IPC管道。
- dgram.Socket。UDP套接字。
- dgram.Native。C++层面的UDP套接字。
  send()方法在将消息发送到IPC管道前，将消息组装成两个对象，一个参数是handle，另一个是message。

message参数如下：

```
{
 cmd: 'NODE_HANDLE',
 type: 'net.Server',
 msg: message
}
```

> 发送到IPC管道中的实际上是我们要发送的句柄文件描述符，文件描述符实际上是一个整数值。这个**message对象在写入到IPC管道时也会通过JSON.stringify()进行序列化。所以最终发送到IPC通道中的信息都是字符串**，send()方法能发送消息和句柄并不意味着它能发送任意对象。**连接了IPC通道的子进程可以读取到父进程发来的消息，将字符串通过JSON.parse()解析还原为对象后，才触发message事件将消息体传递给应用层使用。**在这个过程中，消息对象还要被进行过滤处理，message.cmd的值如果以NODE_为前缀，它将响应一个内部事件internalMessage。**如果message.cmd值为NODE_HANDLE，它将取出message.type值和得到的文件描述符一起还原出一个对应的对象。**



以发送的TCP服务器句柄为例，子进程收到消息后的还原过程如下所示：

```
function(message, handle, emit) {
 var self = this;
 var server = new net.Server();
 server.listen(handle, function() {
   emit(server);
 });
}
```

上面的代码中，**子进程根据message.type创建对应TCP服务器对象，然后监听到文件描述符上。**由于底层细节不被应用层感知，所以在子进程中，开发者会有一种服务器就是从父进程中直接传递过来的错觉。值得注意的是，**Node进程之间只有消息传递，不会真正地传递对象**，这种错觉是抽象封装的结果。

端口共同监听

问题1：为何监听相同端口会出现异常

独立启动的进程中，TCP服务器端socket套接字的文件描述符并不相同，导致监听到相同的端口时会抛出异常。

问题2：为何通过发送句柄后，多个进程可以监听到相同的端口而不引起EADDRINUSE异常

Node底层对每个端口监听都设置了SO_REUSEADDR选项，这个选项的涵义是不同进程可以就相同的网卡和端口进行监听，这个服务器端套接字可以被不同的进程复用，如下所示：

```
setsockopt(tcp->io_watcher.fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on))
```

由于独立启动的进程互相之间并不知道文件描述符，所以监听相同端口时就会失败。但**对于send()发送的句柄还原出来的服务而言，它们的文件描述符是相同的，所以监听相同端口不会引起异常。多个应用监听相同端口时，文件描述符同一时间只能被某个进程所用。换言之就是网络请求向服务器端发送时，只有一个幸运的进程能够抢到连接，也就是说只有它能为这个请求进行服务。这些进程服务是抢占式的。**

9.3 集群稳定之路

除了搭建集群，充分利用多核CPU资源以外，还有一些细节需要考虑：

- 性能问题。
- 多个工作进程的存活状态管理。
- 工作进程的平滑重启。
- 配置或者静态数据的动态重新载入。
- 其他细节。
  虽然创建了很多工作进程，但每个工作进程依然是在单线程上执行的，它的稳定性还不能得到完全的保障。因此需要建立起一个健全的机制来保障Node应用的健壮性。

9.3.1 进程事件

除了message事件外，Node还有如下这些事件：

- error：当子进程无法被复制创建、无法被杀死、无法发送消息时会触发该事件。
- exit：子进程退出时触发该事件，子进程如果是正常退出，这个事件的第一个参数为退出码，否则为null。如果进程是通过kill()方法被杀死的，会得到第二个参数，它表示杀死进程时的信号。
- close：在子进程的标准输入输出流中止时触发该事件，参数与exit相同。
- disconnect：在父进程或子进程中调用disconnect()方法时触发该事件，在调用该方法时将关闭监听IPC通道。
  除了send()外，还能通过kill()方法给子进程发送消息。**kill()方法并不能真正地将通过IPC相连的子进程杀死，它只是给子进程发送了一个系统信号。默认情况下，父进程将通过kill()方法给子进程发送一个SIGTERM信号。**

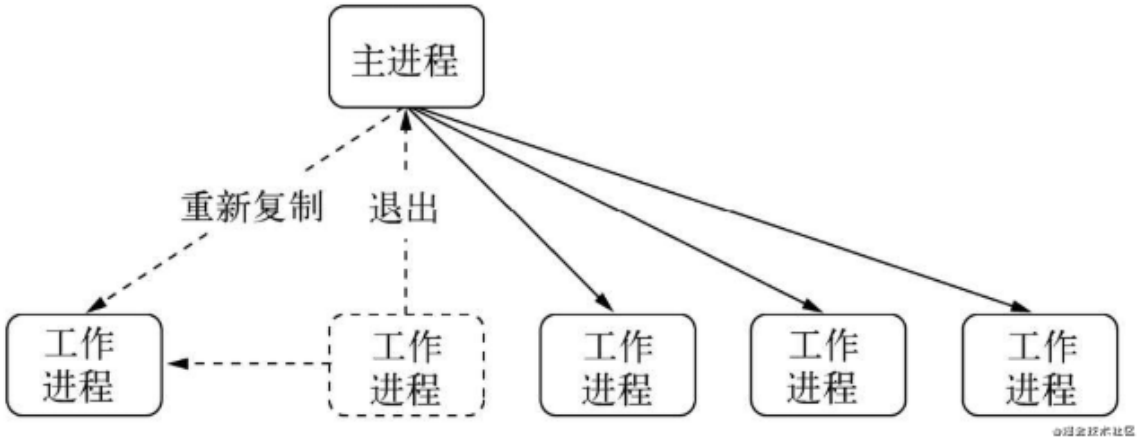
```
// 子进程
child.kill([signal]);
// 当前进程
process.kill(pid, [signal]);
```

Node提供了这些信号对应的信号事件，每个进程都可以监听这些信号事件。这些信号事件是用来通知进程的，每个信号事件有不同的含义，进程在收到响应信号时，应当做出约定的行为，如SIGTERM是软件终止信号，进程收到该信号时应当退出。

```
process.on('SIGTERM', function() {
 console.log('Got a SIGTERM, exiting...');
 process.exit(1);
});
console.log('server running with PID:', process.pid);
process.kill(process.pid, 'SIGTERM');
```

9.3.2 自动重启

有了父子进程之间的相关事件之后，就可以在这些关系之间创建出需要的机制了。至少我们能够通过监听子进程的exit事件来获知其退出的信息，接着前文的多进程架构，我们在主进程上要加入一些子进程管理的机制，比如重新启动一个工作进程来继续服务。



示例代码

```
// master.js
var fork = require('child_process').fork;
var cpus = require('os').cpus();
var server = require('net').createServer();
server.listen(1337);
var workers = {};
var createWorker = function () {
 var worker = fork(__dirname + '/worker.js');
 //退出时重新启动新的进程
 worker.on('exit', function () {
   console.log('Worker ' + worker.pid + ' exited.');
   delete workers[worker.pid];
   createWorker();
 });
 // 句柄转发
 worker.send('server', server);
 workers[worker.pid] = worker;
 console.log('Create worker. pid: ' + worker.pid);
};
for (var i = 0; i < cpus.length; i++) {
 createWorker();
}
// 进程自己退出时，让所有工作进程退出
process.on('exit', function () {
 for (var pid in workers) {
   workers[pid].kill();
 }
});
```

测试一下上面的代码，如下所示：

```
$ node master.js
Create worker. pid: 30504
Create worker. pid: 30505
Create worker. pid: 30506
Create worker. pid: 30507
```

通过kill命令杀死某个进程试试，如下所示：

```
$ kill 30506
```

结果是30506进程退出后，自动启动了一个新的工作进程30518，总体进程数量并没有发生改变，如下所示：

```
Worker 30506 exited.
Create worker. pid: 30518
```

在这个场景中我们主动杀死了一个进程，在实际业务中，可能有隐藏的bug导致工作进程退出，那么我们需要仔细地处理这种异常，如下所示：

```
// worker.js
var http = require('http');
var server = http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': 'text/plain'});
 res.end('handled by child, pid is ' + process.pid + '\n');
});
var worker;
process.on('message', function (m, tcp) {
 if (m === 'server') {
   worker = tcp;
   worker.on('connection', function (socket) {
     server.emit('connection', socket);
   });
 }
});
process.on('uncaughtException', function () {
 // 停止接收新的连接
 worker.close(function () {
   // 所有已有连接断开后，退出进程
   process.exit(1);
 });
});
```

上述代码的处理流程是，一旦有未捕获的异常出现，工作进程就会立即停止接收新的连接；当所有连接断开后，退出进程。主进程在侦听到工作进程的exit后，将会立即启动新的进程服务，以此保证整个集群中总是有进程在为用户服务的。

自杀信号

上述代码存在的问题是要等到已有的所有连接断开后进程才退出，**在极端的情况下，所有工作进程都停止接收新的连接，全处在等待退出的状态。但在等到进程完全退出才重启的过程中，所有新来的请求可能存在没有工作进程为新用户服务的情景，这会丢掉大部分请求。**

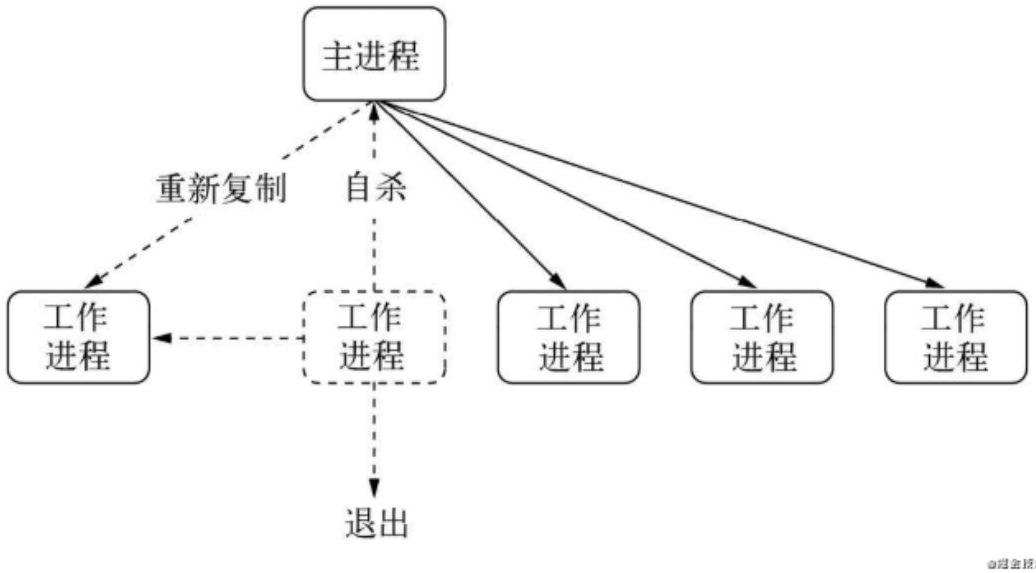
为此需要改进这个过程，**不能等到工作进程退出后才重启新的工作进程。当然也不能暴力退出进程，因为这样会导致已连接的用户直接断开。于是我们在退出的流程中增加一个自杀（suicide）信号。工作进程在得知要退出时，向主进程发送一个自杀信号，然后才停止接收新的连接，当所有连接断开后才退出。主进程在接收到自杀信号后，立即创建新的工作进程服务。**代码改动如下所示：

```
// worker.js
process.on('uncaughtException', function (err) {
 process.send({act: 'suicide'});
 // 停止接收新的连接
 worker.close(function () {
 // 所有已有连接断开后，退出进程
 process.exit(1);
 });
});
```

主进程将重启工作进程的任务，从exit事件的处理函数中转移到message事件的处理函数中，如下所示：

```
var createWorker = function () {
 var worker = fork(__dirname + '/worker.js');
 // 启动新的进程
 worker.on('message', function (message) {
   if (message.act === 'suicide') {
     createWorker();
   }
 });
 worker.on('exit', function () {
   console.log('Worker ' + worker.pid + ' exited.');
   delete workers[worker.pid];
 });
 worker.send('server', server);
 workers[worker.pid] = worker;
 console.log('Create worker. pid: ' + worker.pid);
};
```

与前一种方案相比，**创建新工作进程在前，退出异常进程在后。**在这个可怜的异常进程退出之前，总是有新的工作进程来替上它的岗位。至此我们完成了进程的平滑重启，一旦有异常出现，主进程会创建新的工作进程来为用户服务，旧的进程一旦处理完已有连接就自动断开。整个过程使得我们的应用的稳定性和健壮性大大提高。示意图如下图所示：



这里存在问题的是有可能我们的连接是长连接，不是HTTP服务的这种短连接，**等待长连接断开可能需要较久的时间。为此为已有连接的断开设置一个超时时间是必要的，在限定时间里强制退出**的设置如下所示：

```
process.on('uncaughtException', function (err) {
 process.send({act: 'suicide'});
 // 停止接收新的连接
 worker.close(function () {
   // 所有已有连接断开后，退出进程
   process.exit(1);
 });
 // 五秒后退出进程
 setTimeout(function () {
   process.exit(1);
 }, 5000);
});
```

限量重启

通过自杀信号告知主进程可以使得新连接总是有进程服务，但是依然还是有极端的情况。**工作进程不能无限制地被重启，如果启动的过程中就发生了错误，或者启动后接到连接就收到错误，会导致工作进程被频繁重启，这种频繁重启不属于我们捕捉未知异常的情况，因为这种短时间内频繁重启已经不符合预期的设置，极有可能是程序编写的错误。**

为了消除这种无意义的重启，在满足一定规则的限制下，不应当反复重启。比如在单位时间内规定只能重启多少次，超过限制就触发giveup事件，告知放弃重启工作进程这个重要事件。

```
// 重启次数
var limit = 10;
// 时间单位
var during = 60000;
var restart = [];
var isTooFrequently = function () {
  //记录重启时间
  var time = Date.now();
  var length = restart.push(time);
  if (length > limit) {
    // 取出最后10个记录
    restart = restart.slice(limit * -1);
  }
  // 最后一次重启到前10次重启之间的时间间隔
  return restart.length >= limit && restart[restart.length - 1] - restart[0] <
during;
};
var workers = {};
var createWorker = function () {
 // 检查是否太过于频繁
 if (isTooFrequently()) {
   // 触发giveup事件后不再重启
   process.emit('giveup', length, during);
   return;
 }
 var worker = fork(__dirname + '/worker.js');
 worker.on('exit', function () {
   console.log('Worker ' + worker.pid + ' exited.');
   delete workers[worker.pid];
 });
 // 重新启动新的进程
 worker.on('message', function (message) {
   if (message.act === 'suicide') {
```

```
      createWorker();
    }
  });
  // 句柄转发
  worker.send('server', server);
  workers[worker.pid] = worker;
  console.log('Create worker. pid: ' + worker.pid);
};
```

> giveup事件是比uncaughtException更严重的异常事件。uncaughtException只代表集群中某个工作进程退出，在整体性保证下，不会出现用户得不到服务的情况，但是这个**giveup事件则表示集群中没有任何进程服务**了，十分危险。为了健壮性考虑，我们应在giveup事件中添加重要日志，并让监控系统监视到这个严重错误，进而报警等。

9.3.3 负载均衡

在多进程之间监听相同的端口，使得用户请求能够分散到多个进程上进行处理，这带来的好处是可以将CPU资源都调用起来。这犹如饭店将客人的点单分发给多个厨师进行餐点制作。既然涉及多个厨师共同处理所有菜单，那么保证每个厨师的工作量是一门学问，既不能让一些厨师忙不过来，也不能让一些厨师闲着，这种保证多个处理单元工作量公平的策略叫负载均衡。

Node默认提供的机制是采用操作系统的抢占式策略。所谓的抢占式就是在一堆工作进程中，闲着的进程对到来的请求进行争抢，谁抢到谁服务。

一般而言，这种抢占式策略对大家是公平的，各个进程可以根据自己的繁忙度来进行抢占。但是**对于Node而言，需要分清的是它的繁忙是由CPU、I/O两个部分构成的，影响抢占的是CPU的繁忙度。对不同的业务，可能存在I/O繁忙，而CPU较为空闲的情况，这可能造成某个进程能够抢到较多请求，形成负载不均衡的情况。**

为此Node在v0.11中提供了一种**新的策略使得负载均衡更合理，这种新的策略叫Round-Robin，又叫轮叫调度。**轮叫调度的工作方式是**由主进程接受连接，将其依次分发给工作进程。分发的策略是在N个工作进程中，每次选择第i = (i + 1)mod n个进程来发送连接。**在cluster模块中启用它的方式如下：

```
// 启用Round-Robin
cluster.schedulingPolicy = cluster.SCHED_RR
// 不启用Round-Robin
cluster.schedulingPolicy = cluster.SCHED_NONE
```

或者在环境变量中设置NODE_CLUSTER_SCHED_POLICY的值，如下所示：

```
export NODE_CLUSTER_SCHED_POLICY=rr
export NODE_CLUSTER_SCHED_POLICY=none
```

Round-Robin非常简单，可以避免CPU和I/O繁忙差异导致的负载不均衡。Round-Robin策略也可以通过代理服务器来实现，但是它会导致服务器上消耗的文件描述符是平常方式的两倍。

9.3.4 状态共享

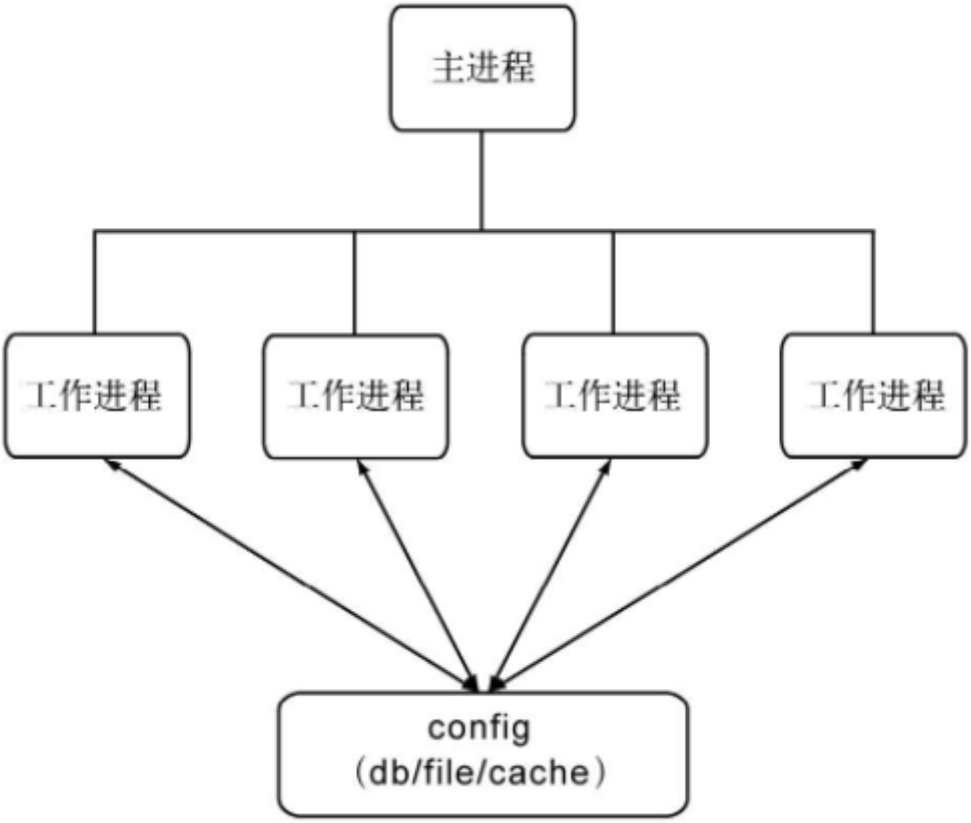
在Node进程中不宜存放太多数据，因为它会加重垃圾回收的负担，进而影响性能。同时，Node也不允许在多个进程之间共享数据。但在实际的业务中，往往需要共享一些数据，譬如配置数据，这在多个进程中应当是一致的。为此，在不允许共享数据的情况下，我们**需要一种方案和机制来实现数据在多个进程之间的共享。**

第三方数据存储

解决数据共享最直接、简单的方式就是通过第三方来进行数据存储，比如将数据存放到数据库、磁盘文件、缓存服务（如Redis）中，所有工作进程启动时将其读取进内存中。但这种方式存在的问题是如果数据发生改变，还需要一种机制通知到各个子进程，使得它们的内部状态也得到更新。

定时轮询

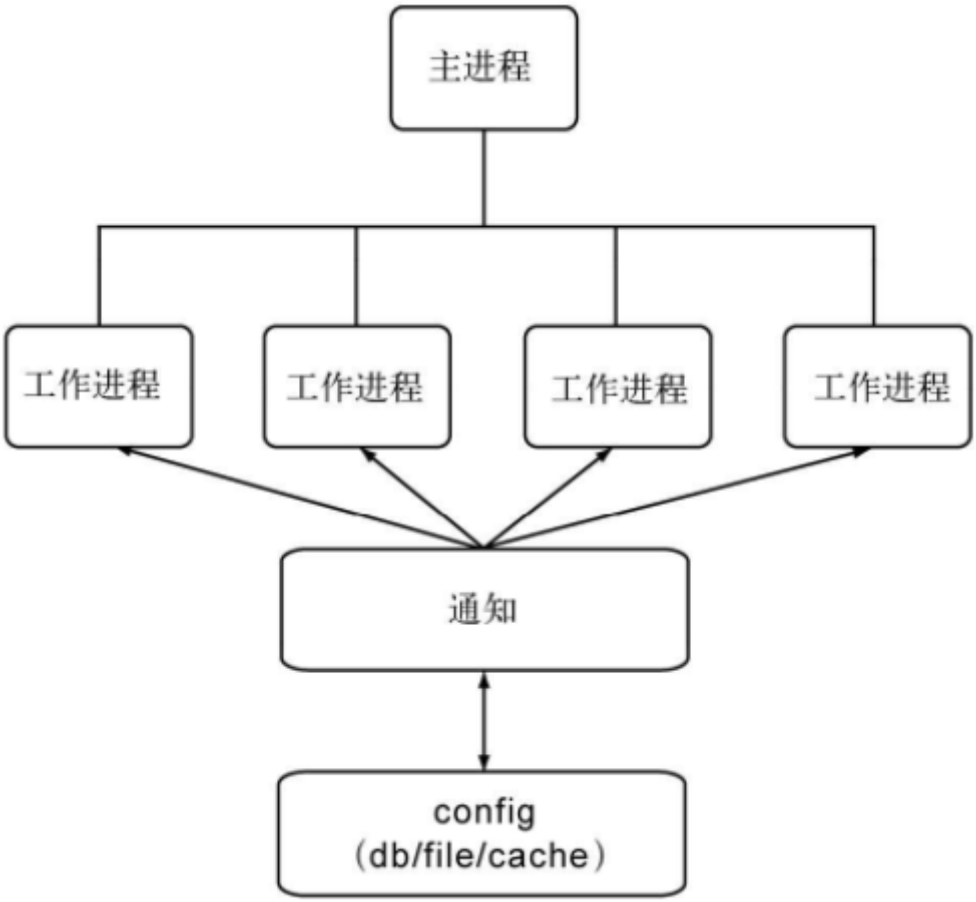
实现状态同步的机制有两种，一种是各个子进程去向第三方进行定时轮询，示意图如下图所示：



> 定时轮询带来的问题是轮询时间不能过密，如果子进程过多，会形成并发处理，如果数据没有发生改变，这些轮询会没有意义，白白增加查询状态的开销。如果轮询时间过长，数据发生改变时，不能及时更新到子进程中，会有一定的延迟。

主动通知

一种改进的方式是当数据发生更新时，主动通知子进程。当然，即使是主动通知，也需要一种机制来及时获取数据的改变。**这个过程仍然不能脱离轮询，但我们可以减少轮询的进程数量**，我们将这种用来发送通知和查询状态是否更改的进程叫做通知进程。为了不混合业务逻辑，可以将这个进程设计为只进行轮询和通知，不处理任何业务逻辑，示意图如下图所示：



> 这种推送机制**如果按进程间信号传递，在跨多台服务器时会无效，是故可以考虑采用TCP或UDP的方案**。进程在启动时从通知服务处除了读取第一次数据外，还将进程信息注册到通知服务处。一旦通过轮询发现有数据更新后，根据注册信息，将更新后的数据发送给工作进程。由于不涉及太多进程去向同一地方进行状态查询，状态响应处的压力不至于太过巨大，单一的通知服务轮询带来的压力并不大，所以可以将轮询时间调整得较短，一旦发现更新，就能实时地推送到各个子进程中。

9.4 Cluster模块

在v0.8版本之前，实现多进程架构必须通过child_process来实现，要创建单机Node集群，由于有这么多细节需要处理，对普通工程师而言是一件相对较难的工作，于是v0.8时直接引入了cluster模块，用以解决多核CPU的利用率问题，同时也提供了较完善的API，用以处理进程的健壮性问题。

对于之前提到的创建Node进程集群，cluster实现起来也是很轻松的事情，如下所示：

```
// cluster.js
var cluster = require('cluster');
cluster.setupMaster({
 exec: "worker.js"
});
var cpus = require('os').cpus();
for (var i = 0; i < cpus.length; i++) {
 cluster.fork();
}
```

执行`node cluster.js`将会得到与前文创建子进程集群的效果相同。就官方的文档而言，它更喜欢如下的形式作为示例：

```
var cluster = require('cluster');
var http = require('http');
```

```
var numCPUs = require('os').cpus().length;
if (cluster.isMaster) {
 // Fork workers
 for (var i = 0; i < numCPUs; i++) {
   cluster.fork();
 }
 cluster.on('exit', function(worker, code, signal) {
   console.log('worker ' + worker.process.pid + ' died');
 });
} else {
 // Workers can share any TCP connection
  // In this case its a HTTP server
 http.createServer(function(req, res) {
   res.writeHead(200);
   res.end("hello world\n");
 }).listen(8000);
}
```

在进程中判断是主进程还是工作进程，主要取决于环境变量中是否有NODE_UNIQUE_ID，如下所示：

```
cluster.isWorker = ('NODE_UNIQUE_ID' in process.env);
cluster.isMaster = (cluster.isWorker === false);
```

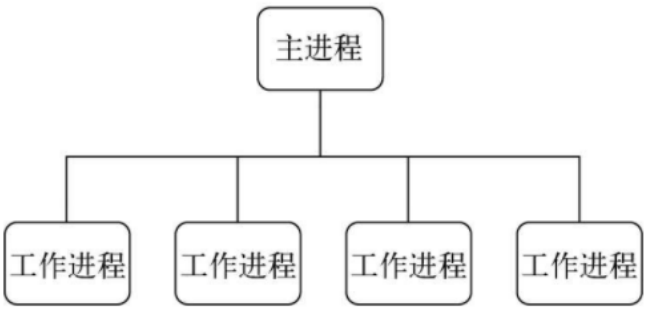
但是官方示例中忽而判断cluster.isMaster、忽而判断cluster.isWorker，对于代码的可读性十分差。我建议用cluster.setupMaster()这个API，将主进程和工作进程从代码上完全剥离，如同send()方法看起来直接将服务器从主进程发送到子进程那样神奇，剥离代码之后，甚至都感觉不到主进程中有任何服务器相关的代码。

通过cluster.setupMaster()创建子进程而不是使用cluster.fork()，程序结构不再凌乱，逻辑分明，代码的可读性和可维护性较好。

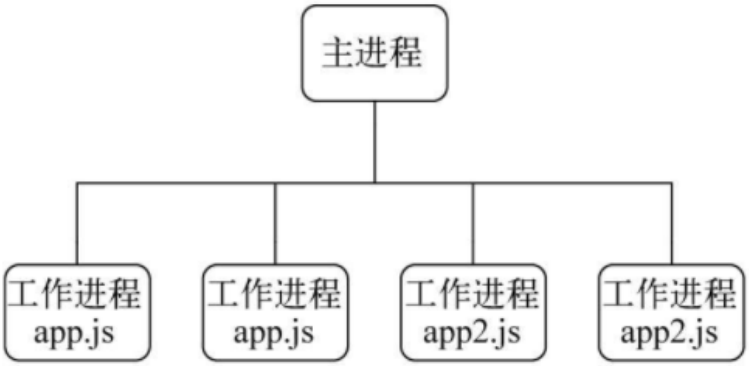
9.4.1 Cluster工作原理

事实上**cluster模块就是child_process和net模块的组合应用。**cluster启动时，如同在9.2.3节里的代码一样，它**会在内部启动TCP服务器，在cluster.fork()子进程时，将这个TCP服务器端socket的文件描述符发送给工作进程。如果进程是通过cluster.fork()复制出来的，那么它的环境变量里就存在NODE_UNIQUE_ID，如果工作进程中存在listen()侦听网络端口的调用，它将拿到该文件描述符，通过SO_REUSEADDR端口重用，从而实现多个子进程共享端口。**对于普通方式启动的进程，则不存在文件描述符传递共享等事情。

在cluster内部隐式创建TCP服务器的方式对使用者来说十分透明，但也正是这种方式使得它无法如直接使用child_process那样灵活。在cluster模块应用中，一个主进程只能管理一组工作进程，如图所示。



对于自行通过child_process来操作时，则可以更灵活地控制工作进程，甚至控制多组工作进程。其原因在于自行通过child_process操作子进程时，可以隐式地创建多个TCP服务器，使得子进程可以共享多个的服务器端socket，如图所示。



9.4.2 Cluster事件

对于健壮性处理，cluster模块也暴露了相当多的事件：

- fork：复制一个工作进程后触发该事件。
- online：复制好一个工作进程后，工作进程主动发送一条online消息给主进程，主进程收到消息后，触发该事件。
- listening：工作进程中调用listen()（共享了服务器端Socket）后，发送一条listening消息给主进程，主进程收到消息后，触发该事件。
- disconnect：主进程和工作进程之间IPC通道断开后会触发该事件。
- exit：有工作进程退出时触发该事件。
- setup:cluster.setupMaster()执行后触发该事件。

这些事件大多跟child_process模块的事件相关，在进程间消息传递的基础上完成的封装。这些事件对于增强应用的健壮性已经足够了。

9.5 总结

尽管Node从单线程的角度来讲它有够脆弱的：既不能充分利用多核CPU资源，稳定性也无法得到保障。但是群体的力量是强大的，通过简单的主从模式，就可以将应用的质量提升一个档次。在实际的复杂业务中，我们可能要启动很多子进程来处理任务，结构甚至远比主从模式复杂，但是每个子进程应当是简单到只做好一件事，然后通过进程间通信技术将它们连接起来即可。这符合Unix的设计理念，每个进程只做一件事，并做好一件事，将复杂分解为简单，将简单组合成强大。

尽管通过child_process模块可以大幅提升Node的稳定性，但是一旦主进程出现问题，所有子进程将会失去管理。在Node的进程管理之外，还需要用监听进程数量或监听日志的方式确保整个系统的稳定性，即使主进程出错退出，也能及时得到监控警报，使得开发者可以及时处理故障。