- 创建多进程
- 进程间通信
  - socket
  - 管道
  - 信号
- cluster
- 进程管理：pm2与egg-cluster
  - pm2
  - egg-cluster

对于前端开发同学，一定很清楚js是单线程非阻塞的，这决定了NodeJS能够支持高性能的服务的开发。 JavaScript的单线程非阻塞特性让NodeJS适合IO密集型应用，因为JavaScript在访问磁盘/数据库/RPC等时候不需要阻塞等待结果，而是可以异步监听结果，同时继续向下执行。
但js不适合计算密集型应用，因为当JavaScript遇到耗费计算性能的任务时候，单线程的缺点就暴露出来了。后面的任务都要被阻塞，直到耗时任务执行完毕。
为了优化NodeJS不适合计算密集型任务的问题，NodeJS提供了多线程和多进程的支持。
多进程和多线程从两个方面对计算密集型任务进行了优化，**异步和并发**：
1. 异步，对于耗时任务，可以新建一个线程或者进程来执行，执行完毕再通知主线程/进程。
看下面例子，这是一个koa接口，里面有耗时任务，会阻塞其他任务执行。

```
const Koa = require('koa');
const app = new Koa();

app.use(async ctx => {
const url = ctx.request.url;
if (url === '/') {
ctx.body = 'hello';
}

if (url === '/compute') {
```

```
let sum = 0;
for (let i = 0; i < 1e20; i++) {
sum += i;
}
ctx.body = `${sum}`;
}
});

app.listen(3000, () => {
console.log('http://localhost:300/ start')
});
```

可以通过多线程和多进程来解决这个问题。

NodeJS提供多线程模块`worker_threads`，其中`Woker`模块用来创建线程，`parentPort`用在子线程中，可以获取主线程引用，子线程通过`parentPort.postMessage`发送数据给主线程，主线程通过`worker.on`接受数据。

```
//api.js
const Koa = require('koa');
const app = new Koa();

const {Worker} = require('worker_threads');

app.use(async (ctx) => {
const url = ctx.request.url;
if (url === '/') {
ctx.body = 'hello';
}

if (url === '/compute') {
const sum = await new Promise(resolve => {
const worker = new Worker(__dirname + '/compute.js');
//接收信息
worker.on('message', data => {
resolve(data);
})

});
ctx.body = `${sum}`;
```

```
}
})

app.listen(3000, () => {
console.log('http://localhost:3000/ start')
});



//computer.js
const {parentPort} = require('worker_threads')
let sum = 0;
for (let i = 0; i < 1e20; i++) {
sum += i;
}

//发送信息
parentPort.postMessage(sum);
```

下面是使用多进程解决耗时任务的方法，多进程模块`child_process`提供了`fork`方法（后面会介绍更多创建子进程的方法），可以用来创建子进程，主进程通过fork返回值（`worker`）持有子进程的引用，并通过`worker.on`监听子进程发送的数据，子进程通过`process.send`给父进程发送数据。

```
//api.js
const Koa = require('koa');
const app = new Koa();

const {fork} = require('child_process');

app.use(async ctx => {
const url = ctx.request.url;
if (url === '/') {
ctx.body = 'hello';
}

if (url === '/compute') {
const sum = await new Promise(resolve => {
const worker = fork(__dirname + '/compute.js');
worker.on('message', data => {
resolve(data);
```

```
});
});
ctx.body = `${sum}`;
}
});

app.listen(300, () => {
console.log('http://localhost:300/ start');
});

//computer.js
let sum = 0;
for (let i = 0; i < 1e20; i++) {
sum += i;
}
process.send(sum);
```

1. 并发，为了可以更好地利用多核能力，通常会对同一个脚本创建多进程和多线程，数量和CPU核数相同，这样可以让任务并发执行，最大程度提升了任务执行效率。

本文重点讲解多进程的使用。

从实际应用角度，如果我们希望使用多进程，让我们的应用支持并发执行，提升应用性能，那么首先要创建多进程，然后进程运行的过程中难免涉及到进程之间的通信，包括父子进程通信和兄弟进程之间的通信，另外还有很重要的一点是进程的管理，因为创建了多个进程，那么来了一个任务应该交给哪个进程去执行呢？进程必然要支持后台执行（守护进程），这个又怎么实现呢？进程崩溃如何重启？重启过于频繁的不稳定进程又如何限制？如何操作进程的启动、停止、重启？

这一系列的进程管理工作都有相关的工具支持。

接下来就按照上面说明的创建进程、进程间通信、进程管理（cluster集群管理、进程管理工具：pm2和egg-cluster）。

创建多进程

`child_process`模块用来创建子进程，该模块提供了4个方法用于创建子进程

```
const {spawn, fork, exec, execFile} = require('child_process');复制代码
child_process.spawn(command[, args][, options])
child_process.fork(modulePath[, args][, options])
child_process.exec(command[, options][, callback])
child_process.execFile(file[, args][, options][, callback])
```

`spawn`会启动一个shell，并在shell上执行命令；`spawn`会在父子进程间建立IO流`stdin`、`stdout`、`stderr`；`spawn`返回一个子进程的引用，通过这个引用可以监听子进程状态，并接收子进程的输入流。

```
const { spawn } = require('child_process'); const ls = spawn('ls', ['-lh', '/usr']); ls.stdout.on('data', (data) => { console.log(`stdout: ${data}`); }); ls.stderr.on('data', (data) => { console.error(`stderr: ${data}`); }); ls.on('close', (code) => { console.log(`child process exited with code ${code}`); });
```

`fork`、`exec`和`execFile`都是基于`spawn`扩展的。

`exec`与`spawn`不同，它接收一个回调作为参数，回调中会传入报错和IO流

```
const { exec } = require('child_process'); exec('cat ./test.txt', (error, stdout, stderr) => { if (error) { console.error(`exec error: ${error}`); return; } console.log(`stdout: ${stdout}`); console.error(`stderr: ${stderr}`); });
```

`execFile`和`exec`不同的是，它不会创建一个shell，而是直接执行可执行文件，因此效率比`exec`稍高一些，另外，它传入的第一个参数是可执行文件，第二个参数是执行可执行文件的参数。参考nodejs进阶视频讲解：进入学习

```
const { execFile } = require('child_process'); execFile('cat', ['./test.txt'], (error, stdout, stderr) => { if (error) { console.error(`exec error: ${error}`); return; } console.log(stdout); });
```

`fork`支持传入一个NodeJS模块路径，而非shell命令，返回一个子进程引用，这个子进程的引用和父进程建立了一个内置的IPC通道，可以让父子进程通信。

```
// parent.js var child_process = require('child_process'); var child = child_process.fork('./child.js'); child.on('message', function(m){ console.log('message from child: ' + JSON.stringify(m)); }); child.send({from: 'parent'}); // child.js process.on('message', function(m){ console.log('message from parent: ' + JSON.stringify(m)); }); process.send({from: 'child'});
```

对于上面几个创建子进程的方法，有对应的同步版本。

`spawnSync`、`execSync`、`execFileSync`。

进程间通信

进程间通信分为父子进程通信和兄弟进程通信，当然也可能涉及远程进程通信，这个会在后面提到，本文主要关注本地进程的通信。

父子进程通信可以通过标准IO流传递json

```
// 父进程
const { spawn } = require('child_process');

child = spawn('node', ['./stdio-child.js']);
child.stdout.setEncoding('utf8');
// 父进程-发
child.stdin.write(JSON.stringify({
type: 'handshake',
payload: '你好吖'
```

```
}));
// 父进程-收
child.stdout.on('data', function (chunk) {
let data = chunk.toString();
let message = JSON.parse(data);
console.log(`${message.type} ${message.payload}`);
});

// ./stdio-child.js
// 子进程-收
process.stdin.on('data', (chunk) => {
let data = chunk.toString();
let message = JSON.parse(data);
switch (message.type) {
case 'handshake':
// 子进程-发
process.stdout.write(JSON.stringify({
type: 'message',
payload: message.payload + ' : hoho'
}));
break;
default:
break;
}
});
```

使用`fork`创建的子进程，父子进程之间会建立内置IPC通道（不知道该IPC通道底层是使用管道还是socket实现）。（代码见“创建多进程小节”）

因此父子进程通信是NodeJS原生支持的。

下面我们看兄弟进程如何通信。

通常进程通信有几种方法：共享内存、消息队列、管道、socket、信号。

其中对于共享内存和消息队列，NodeJS并未提供原生的进程间通信支持，需要依赖第三方实现，比如通过C++`shared-memory-disruptor addon`插件实现共享内存的支持、通过redis、MQ实现消息队列的支持。

下面介绍在NodeJS中通过socket、管道、信号实现的进程间通信。

socket

socket是应用层与TCP/IP协议族通信的中间抽象层，是一种操作系统提供的进程间通信机制，是操作系统提供的，工作在传输层的网络操作API。

socket提供了一系列API，可以让两个进程之间实现客户端–服务端模式的通信。

通过socket实现IPC的方法可以分为两种：

1. TCP/UDP socket，原本用于进行网络通信，实际就是两个远程进程间的通信，但两个进程既可以是远程也可以是本地，使用socket进行通信的方式就是一个进程建立server，另一个进程建立client，然后通过socket提供的能力进行通信。

2. UNIX Domain socket，这是一套由操作系统支持的、和socket很相近的API，但用于IPC，名字虽然是UNIX，实际Linux也支持。socket 原本是为网络通讯设计的，但后来在 socket 的框架上发展出一种 IPC 机制，就是 UNIX domain socket。虽然网络 socket 也可用于同一台主机的进程间通讯(通过 loopback 地址 127.0.0.1)，但是 UNIX domain socket 用于 IPC 更有效率：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。这是因为，IPC 机制本质上是可靠的通讯，而网络协议是为不可靠的通讯设计的。

开源的node–ipc方案就是使用了socket方案

NodeJS如何使用socket进行通信呢？答案是通过net模块实现，看下面的例子。

```
// server
const net = require('net');

net.createServer((stream => {
stream.end(`hello world!\n`);
})).listen(3302, () => {
console.log(`running ...`);
});

// client
const net = require('net');

const socket = net.createConnection({port: 3302});

socket.on('data', data => {
console.log(data.toString());
});
```

UNIX Domain socket在NodeJS层面上提供的API和TCP socket类似，只是listen的是一个文件描述符，而不是端口，相应的，client连接的也是一个文件描述符（`path`）。

```
// 创建进程
const net = require('net')
const unixSocketServer = net.createServer(stream => {
stream.on('data', data => {
```

```
console.log(`receive data: ${data}`)
})
});

unixSocketServer.listen('/tmp/test', () => {
console.log('listening...');
});

// 其他进程

const net = require('net')

const socket = net.createConnection({path: '/tmp/test'})

socket.on('data', data => {
console.log(data.toString());
});

socket.write('my name is vb');

// 输出结果

listening...
```

管道

管道是一种操作系统提供的进程通信方法，它是一种半双工通信，同一时间只能有一个方向的数据流。

管道本质上就是内核中的一个缓存，当进程创建一个管道后，Linux会返回两个文件描述符，一个是写入端的描述符（fd[1]），一个是输出端的描述符（fd[0]），可以通过这两个描述符往管道写入或者读取数据。

NodeJS中也是通过net模块实现管道通信，与socket区别是server listen的和client connect的都是特定格式的管道名。

管道的通信效率比较低下，一般不用它作为进程通信方案。

下面是使用net实现进程通信的示例。

```
var net = require('net');

var PIPE_NAME = "mypipe";
```

```
var PIPE_PATH = "\\.\pipe\" + PIPE_NAME;

var L = console.log;

var server = net.createServer(function(stream) {
L('Server: on connection')

stream.on('data', function(c) {
L('Server: on data:', c.toString());
});

stream.on('end', function() {
L('Server: on end')
server.close();
});

stream.write('Take it easy!');
});

server.on('close',function(){
L('Server: on close');
})

server.listen(PIPE_PATH,function(){
L('Server: on listening');
})

// == Client part == //
var client = net.connect(PIPE_PATH, function() {
L('Client: on connection');
})

client.on('data', function(data) {
L('Client: on data:', data.toString());
client.end('Thanks!');
});
```

```
client.on('end', function() {
L('Client: on end');
})

// Server: on listening
// Client: on connection
// Server: on connection
// Client: on data: Take it easy!
// Server: on data: Thanks!
// Client: on end
// Server: on end
// Server: on close
```

信号

作为完整健壮的程序，需要支持常见的中断退出信号，使得程序能够正确的响应用户和正确的清理退出。

信号是操作系统杀掉进程时候给进程发送的消息，如果进程中没有监听信号并做处理，则操作系统一般会默认直接粗暴地杀死进程，如果进程监听信号，则操作系统不默认处理。

这种进程通信方式比较局限，只用在一个进程杀死另一个进程的情况。

在NodeJS中，一个进程可以杀掉另一个进程，通过制定要被杀掉的进程的id来实现：

`process.kill(pid, signal)/child_process.kill(pid, signal)`。

进程可以监听信号：

```
process.on('SIGINT', () => { console.log('ctl + c has pressed');});复制代码
```

cluster

现在设想我们有了一个启动server的脚步，我们希望能更好地利用多核能力，启动多个进程来执行server脚本，另外我们还要考虑如何给多个进程分配请求。

上面的场景是一个很常见的需求：多进程管理，即一个脚本运行时候创建多个进程，那么如何对多个进程进行管理?

实际上，不仅是在server的场景有这种需求，只要是多进程都会遇到这种需求。而server的多进程还会遇到另一个问题：同一个server脚本监听的端口肯定相同，那启动多个进程时候，端口一定会冲突。

为了解决多进程的问题，并解决server场景的端口冲突问题，NodeJS提供了cluster模块。

这种同样一份代码在多个实例中运行的架构叫做集群，cluster就是一个NodeJS进程集群管理的工具。

cluster提供的能力：

1. 创建子进程

2. 解决多子进程监听同一个端口导致冲突的问题

3. 负载均衡

cluster主要用于server场景，当然也支持非server场景。

先来看下cluster的使用

```
import cluster from 'cluster';
import http from 'http';
import { cpus } from 'os';
import process from 'process';

const numCPUs = cpus().length;

if (cluster.isPrimary) {
console.log(`Primary ${process.pid} is running`);

// Fork workers.
for (let i = 0; i < numCPUs; i++) {
cluster.fork();
}

cluster.on('exit', (worker, code, signal) => {
console.log(`worker ${worker.process.pid} died`);
});
} else {
// Workers can share any TCP connection
// In this case it is an HTTP server
http.createServer((req, res) => {
res.writeHead(200);
res.end('hello world\n');
}).listen(8000);

console.log(`Worker ${process.pid} started`);
}
```

可以看到使用`cluster.fork`创建了子进程，实际上cluster.fork调用了child_process.fork来创建子进程。创建好后，cluster会自动进行负载均衡。

cluster支持设置负载均衡策略，有两种策略：轮询和操作系统默认策略。可以通过设置`cluster.schedulingPolicy = cluster.SCHED_RR;`指定轮询策略，设置`cluster.schedulingPolicy = cluster.SCHED_NONE;`指定用操作系统默认策略。也可以设置环境变量`NODE_CLUSTER_SCHED_POLICY`为`rr/none`来实现。

让人比较在意的是，cluster是如何解决端口冲突问题的呢?

我们看到代码中使用了`http.createServer`，并监听了端口8000，但实际上子进程并未监听8000，net模块的server.listen方法（http继承自net）判断在cluster子进程中不监听端口，而是创建一个socket并发送到父进程，以此将自己注册到父进程，所以只有父进程监听了端口，子进程通过socket和父进程通信，当一个请求到来后，父进程会根据轮询策略选中一个子进程，然后将请求的句柄（其实就是一个socket）通过进程通信发送给子进程，子进程拿到socket后使用这个socket和客户端通信，响应请求。

那么net中又是如何判断是否是在cluster子进程中的呢？`cluster.fork`对进程做了标识，因此net可以区分出来。

cluster是一个典型的master-worker架构，一个master负责管理worker，而worker才是实际工作的进程。

进程管理：pm2与egg-cluster

除了集群管理，在实际应用运行时候，还有很多进程管理的工作，比如：进程的启动、暂停、重启、记录当前有哪些进程、进程的后台运行、守护进程监听进程崩溃重启、终止不稳定进程（频繁崩溃重启）等等。

社区也有比较成熟的工具做进程管理，比如pm2和egg-cluster

pm2

pm2是一个社区很流行的NodeJS进程管理工具，直观地看，它提供了几个非常好用的能力：

1. 后台运行。
2. 自动重启。
3. 集群管理，支持cluster多进程模式。

其他的功能还包括0s reload、日志管理、终端监控、开发调试等等。

pm2的大概原理是，建立一个守护进程（daemon），用来管理机器上通过pm2启动的应用。当用户通过命令行执行pm2命令对应用进行操作时候，其实是在和daemon通信，daemon接收到指令后进行相应的操作。这时一种C/S架构，命令行相当于客户端（client），守护进程daemon相当于服务器（server），这种模式和docker的运行模式相同，docker也是有一个守护进程接收命令行的指令，再执行对应的操作。

客户端和daemon通过rpc进行通信，daemon是真正的“进程管理者”。

由于有守护进程，在启动应用时候，命令行使用pm2客户端通过rpc向daemon发送信息，daemon创建进程，这样进程不是由客户端创建的，而是daemon创建的，因此客户端退出也不会收到影响，这就是pm2启动的应用可以后台运行的原因。

daemon还会监控进程的状态，崩溃会自动重启（当然频繁重启的进程被认为是不稳定的进程，存在问题，不会一直重启），这样就实现了进程的自动重启。

pm2利用NodeJS的cluster模块实现了集群能力，当配置`exec_mode`为`cluster`时候，pm2就会自动使用cluster创建多个进程，也就有了负载均衡的能力。

egg-cluster

egg-cluster是egg项目开源的一个进程管理工具，它的作用和pm2类似，但两者也有很大的区别，比如pm2的进程模型是master-worker，master负责管理worker，worker负责执行具体任务。egg-cluster的进程模型是master-agent-worker，其中多出来的agent有什么作用呢?
有些工作其实不需要每个 Worker 都去做，如果都做，一来是浪费资源，更重要的是可能会导致多进程间资源访问冲突
既然有了pm2，为什么egg要自己开发一个进程管理工具呢？可以参考作者的回答

> 1. PM2 的理念跟我们不一致，它的大部分功能我们用不上，用得上的部分却又做的不够极致。
> 2. PM2 是AGPL 协议的，对企业应用不友好。

pm2虽然很强大，但还不能说完美，比如pm2并不支持master-agent-worker模型，而这个是实际项目中很常见的一个需求。因此egg-cluster基于实际的场景实现了进程管理的一系列功能。

答案

通过上面的介绍，我们知道了pm2使用cluster做集群管理，cluster又是使用child_process.fork来创建子进程，所以父子进程通信使用的是内置默认的IPC通道。