- Node.js 作为Web架构中间层的使用
- process模块
- 多进程实现
- 进程通信
- 进程守护

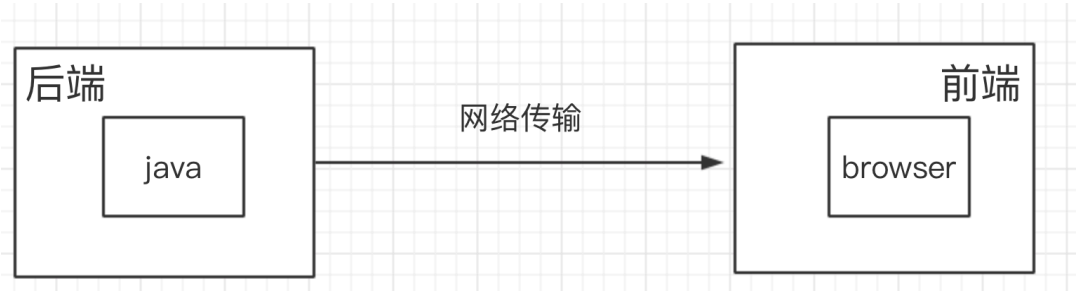
NodeJS作为Web架构中间层的使用

背景

传统做法我们用mock服务器搭建前端数据模拟服务，前后端开发过程中只需要定义好接口规范，就可以做各自的开发任务。联调时，按照之前定义的开发规范进行数据联调就可以。前后端的职能更加清晰：后端提供数据前端接收数据，返回数据处理业务逻辑渲染在浏览器上。

从上面看前后端是分离了，分工也明确了，但是我们会发现一些问题：服务端和客户端各层职责重叠，各搞各的，很难统一具体要做的事情。有可能会有一些性能上的问题。最具体的表现就是我们常用的SPA应用。

传统的前后端



传统的前后端分离问题

性能问题：

- 渲染、数据都在客户端做，影响性能
- 在低速网络情况下体验更差
- 需等待资源到齐后才能进行，会有短暂白屏与闪动，尤其是网页有js时生成的体验更差

重用问题：

- 模版无法重用，造成维护上问题；
- 逻辑无法重用，前端的校验后端仍须再做一次；

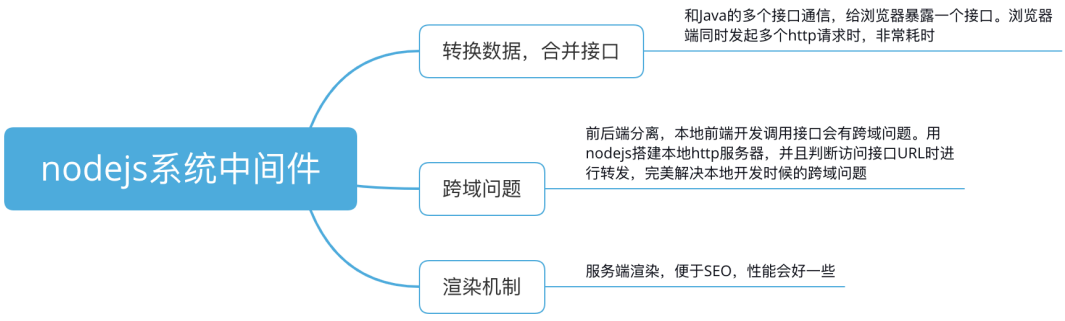
跨终端问题：

- 业务太靠前，导致不同端重复实现；
- 逻辑太靠前，造成维护上的不易；

虽然使用CDN可以提高网站的响应速度，但是面对上述问题，尤其是用户关心的页面体验问题，传统的网站架构依然有问题。如果我们使用node作为中间件会有什么样的效果呢?

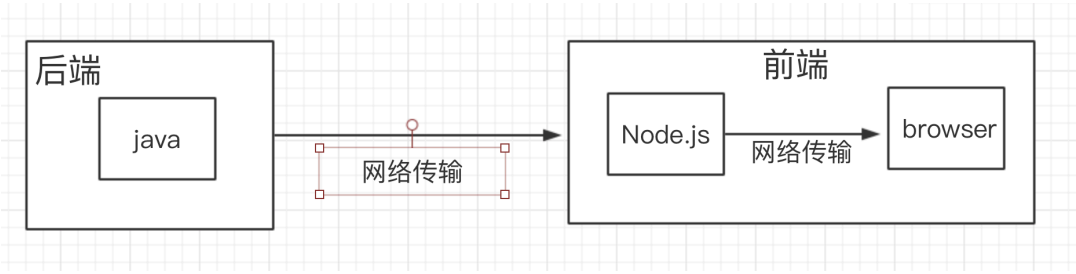
nodejs作为系统中间件的理解

在浏览器端和java端使用nodejs作为中间件，node调用java后端发布的接口，同时node发布http接口给浏览器端调用 。



重新定义的前后端

有了nodejs 后，从工作职能上我们可以重新定义前后端的范畴：



后端	前端	
服务端		浏览器
java	NodeJs	html+js+css
服务层 提供数据接口 维持数据稳定 封装业务逻辑	跑在服务器上的JS 转发数据，串接服务 路由设计，控制逻辑 渲染页面，体验优化 更多的可能	跑在浏览器上的JS CSS、JS加载并运行 DOM操作 任何的前端框架与工具 共用模版、路由

从图中我们可以看到，在服务器和浏览器之间增加了一个中间层，前端比之前多了node。

使用NodeJS作为Web中间层的优势

1. 跨系统、跨终端均可重用页面数据校验、逻辑代码

2. 只需在中间件中做一次数据校验，避免前后端重复校验，在保证数据的有效性同时降低了团队整体开发工作量；

3. 处理数据逻辑，前端开发人员可以专注页面渲染，这样不仅分工更明确，项目协作效率更高，更重要的是页面加载更快，用户体验更好，避免了浏览器长时间显示空白。

为什么选择中间层是node呢？因为我们把中间层归在了前端的范畴，对前端来说，nodejs还是js，从语法角度来说，上手起来容易，其

次开发转移成本相对较低。

那么中间层能给我们带来什么？我们知道引入 node的开发成本还是很大的，首先就是多了一层服务，其他不说，单凭传输时间，就多了一层传输时间！我们看下什么应用场景下node能给我们带来更大的好处。

常见的业务场景

- **接口数据可靠性修复**

有时服务端返回的数据并不是前端想要的结构，前端只需展现。但是后端经常在提供数据后，前端还需处理这些数据。

比如有时会碰到这样的问题：服务端返回的某个字段为null或数据结构太深，前端需不断写代码去判断数据结构是否返回了正确的东西，而不是null 或者undefined，对于这种情况，前端其实不应该重复校验数据的格式，而且这也不是浏览器端js需要做的事情。如果我们在中间层做接口转发，在转发的过程中做数据处理。这样就不用担心数据返回的问题：

```
router.get('/detail', (req, res, next) => {
    httpRequest.get('/detail', (data) => {
      // todo 处理数据
      res.send(data);
    })
})
```

- **页面性能优化和SEO**

有时做单页面应用时，经常会碰到首屏加载性能问题，如果我们有了中间层node，那么我们可以把首屏渲染的任务交给node去做，次屏的渲染依然走之前的浏览器渲染。

服务端渲染可以大幅提高首屏渲染的时间，减少用户的等待时间。这种形式应用最广的就是Vue的服务端渲染。其次对于单页面的SEO优

化也是很好地处理方式。

- **高并发场景**

Node.js中的进程和线程

node是js在服务端的运行环境，构建在chrome的V8引擎之上，基于事件驱动、非阻塞I/O模型，充分利用操作系统提供的异步 I/O 进行多任务的执行，适合于I/O密集型的应用场景，因为异步，程序无需阻塞等待结果返回，同时基于回调通知机制，原本同步模式等待的时间，则可以处理其它任务。

在单核CPU系统上我们采用单进程 + 单线程的模式来开发。在多核CPU系统上，可以通过`child_process.fork`开启多个进程（node之后版本新增Cluster来实现多进程架构），即多进程 + 单线程模式。

注意：开启多进程并不是为了解决高并发，主要是解决了单进程模式下nodejs CPU 利用率不足的情况，充分利用了多核CPU的性能。

node单线程

node单线程是指js执行是单线程，即我们所编写的代码运行在单线程上，实际上node不是真正的单线程。node.js启动后会创建V8实例，V8实例是多线程的，V8中的线程有：

- 主线程：获取代码、编译执行
- 编译线程：主线程执行的时候，可以优化代码
- Profiler线程：记录哪些方法耗时，为优化提供支持
- 其他线程：用于垃圾回收清除工作，因为是多个线程，所以可以并行清除

node中进程的概念

- node.js中每个应用程序都是一个进程类的实例对象

- process代表应用程序,是一个全局对象，通过它可获取node应用程序以及运行该程序的用户、环境等各种信息属性、方法和事件
- 启动一个服务、运行一个实例，就是开一个服务进程，node.js里通过 `node app.js` 开启一个服务进程，多进程即进程的复制（fork），fork出的每个进程都拥有自己的独立空间地址、数据栈，一个进程无法访问另一个进程里定义的变量、数据结构，只有建立IPC 通信，进程之间才可数据共享。

node.js开启服务进程demo

```
const http = require('http');

http.createServer().listen(3000, () => {
    process.title = 'Node.js进程学习' // 进程命
    console.log(`process.pid: `, process.pid);
})
```

运行后，在 Mac 系统自带的监控工具“活动监视器”可看到我们刚开启的 node.js进程

线程

我们知道线程是属于进程的，被包含于进程之中。**一个线程只能属于一个进程，但是一个进程是可拥有多个线程的。**

什么是单线程

- 单线程意思就是一个进程只开一个线程，相当于一个痴情的少年，对一个妹子一心一意，用情专一

node特性

- node的最大特性即单线程，遵循的是**单线程单进程**的模式，单线程是指js的引擎只有一个实例，且在nodejs的主线程中执行，同时node以事件驱动的方式处理IO等异步操作。
- node的单线程模式，只维持一个主线程，减少了线程间上下文

切换所带来的性能开销，但是单线程使得主线程不能进行CPU密集型操作，否则会阻塞主线程。

单线程说明：

1. node.js虽是单线程模型，但是其基于事件驱动、异步非阻塞模式，可应用于高并发场景，避免了线程创建以及线程上下文切换所产生的资源开销。

2. 当项目中需要有大量计算，CPU 耗时的操作时，可以考虑开启多进程实现。

3. node.js开发中，错误会引起整个应用退出，应用的健壮性是值得考验的，尤其是错误的异常抛出，以及进程守护是必须要做的。

单线程会带来的问题：

- 无法利用多核CPU
- 错误就会引起整个应用退出（整个应用就一个进程，挂了话直接就挂了）
- 大量计算长时间占用CPU，导致阻塞线程内的其他操作（异步IO发不出调用，已完成的异步IO回调不能及时执行）。

经典计算耗时造成线程阻塞的例子

```
const http = require('http');
const longComputation = () => {
  let sum = 0;
  for (let i = 0; i < 1e10; i++) {
    sum += i;
  };
  return sum;
};
const server = http.createServer();
server.on('request', (req, res) => {
  if (req.url === '/compute') {
    console.info('计算开始',new Date());
    const sum = longComputation();
    console.info('计算结束',new Date());
    return res.end(`Sum is ${sum}`);
  } else {
    res.end('Ok')
  }
});

server.listen(3000);
//结果
//计算开始 2019-08-25T08:35:38.911Z
//计算结束 2019-08-25T08:35:50.744Z
```

当我们访问127.0.0.1:3000/compute时，如果想调用其他路由比如127.0.0.1/大约需要10秒左右，即用户请求完第一个compute接口后需要等待10秒左右，这对用户是极其不友好的。后面我们会通过创建多进程的方式child_process.fork和cluster来解决这个问题。

多线程

多线程就是没有一个进程只开一个线程的限制，相当于一个风流少年除了爱慕自己班的某个妹子，还想着隔壁班的妹子。Java 就是多线程编程语言的一 种，可以有效避免代码阻塞导致后续请求无法处理。

多线程说明：

多线程的代价在于创建新的线程和执行时上下文线程的切换开销，由于每创建一个线程就会占用一定的内存，当应用程序并发大了之后，内存将会很快耗尽。 还是推荐使用多进程来处理。

process模块

背景

node中，只有一个线程执行所有操作，如果某个操作需要大量消耗CPU资源的情况下，后续操作都需要等待。

子进程的作用

- 进程可执行系统shell命令，可使用操作系统的一些功能
- 如果有很耗时的任务，可通过子进程来避免阻塞事件循环

process进程中常用属性及方法

node中进程process是一个全局对象，无需require直接使用，通过它可开启多个子进程，子进程可通过互相通信来实现信息交换。

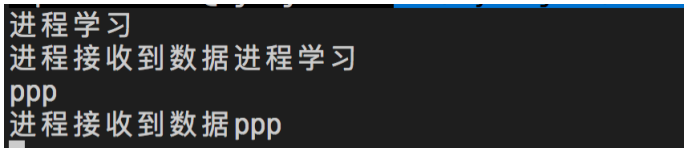
属性

- stdin：标准输入可读流
- stdout：标准输入可写流
- stderr：标准错误输出流
- argv：终端输入参数数组，第一个参数是node，第二个参数是当前执行的.js文件名，之后是执行的参数列表
- env：操作系统环境信息，环境变量，例如通过 `process.env.NODE_ENV` 获取不同环境项目配置信息
- pid：应用程序进程id
- platform：获取当前进程运行的操作系统平台
- process.title：指定进程名称，有时需给进程指定一个名称

stdin以及stdout

```
process.stdin.on('data', (chunk) => {
  process.stdout.write('进程接收到数据' + chunk)
})
```

运行结果：



方法

- process.memoryUsage() 查看内存使用信息
- process.nextTick() 在Event Loop时经常会提到，用于延迟回调函数的执行，是当前eventloop执行完毕时执行的回调函数
- process.chdir() 用于修改node应用程序中使用的当前工作目录
- process.cwd() 进程当前工作目录
- process.kill() 杀死进程
- process.uncaughtException() 当应用程序抛出一个未被捕获的异常时触发进程对象的uncaughtException事件
- uptime()：当前进程已运行时间，例如：pm2守护进程的uptime值

事件

process.on(‘uncaughtException’, cb) 捕获异常信息、process.on(‘exit’, cb) 进程退出监听

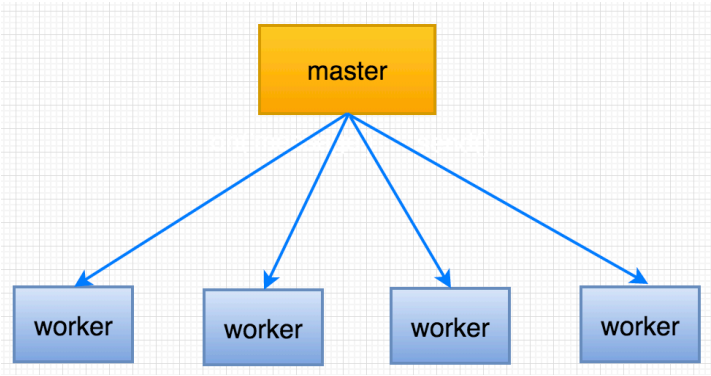
多进程实现

我们知道一个服务会占用一个进程，一个进程是挂在cpu上的，如果我们的服务是多进程的，那么多个进程是可占用多个cpu的。进程有多种创建方式，主要围绕child_process和cluster讲解。子进程和父进程具有相同的代码段、数据段、堆栈，但是它们的内存空间不共享。进程分为master进程和worker进程。

- master负责调度或管理worker进程
- worker负责具体业务处理

在服务器层面，worker可以是一个服务进程，负责处理来自客户端

的请求，多个worker便相当于多个服务器，从而构成一个服务器集群。master负责创建worker，将来自客户端的请求分配到各个服务器上去处理，并监控worker的运行状态以及进行管理等操作。



child_process模块

背景

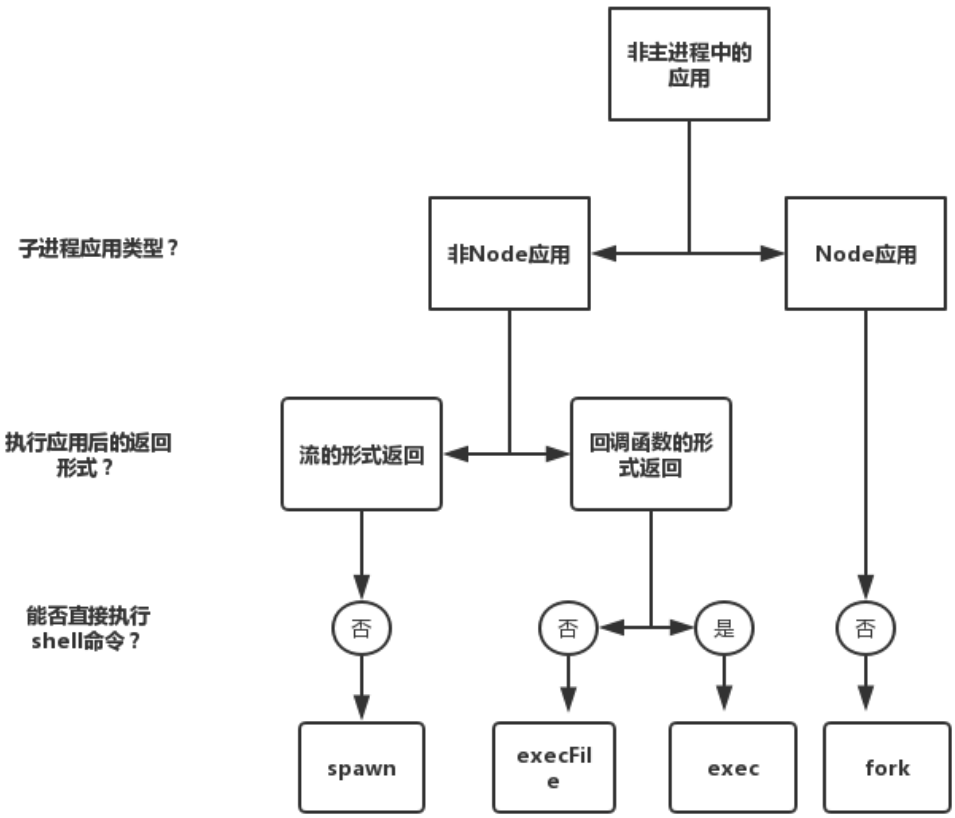
node中，只有一个线程执行所有操作，如果某个操作需要大量消耗CPU资源，那么后续操作都需等待。

特点

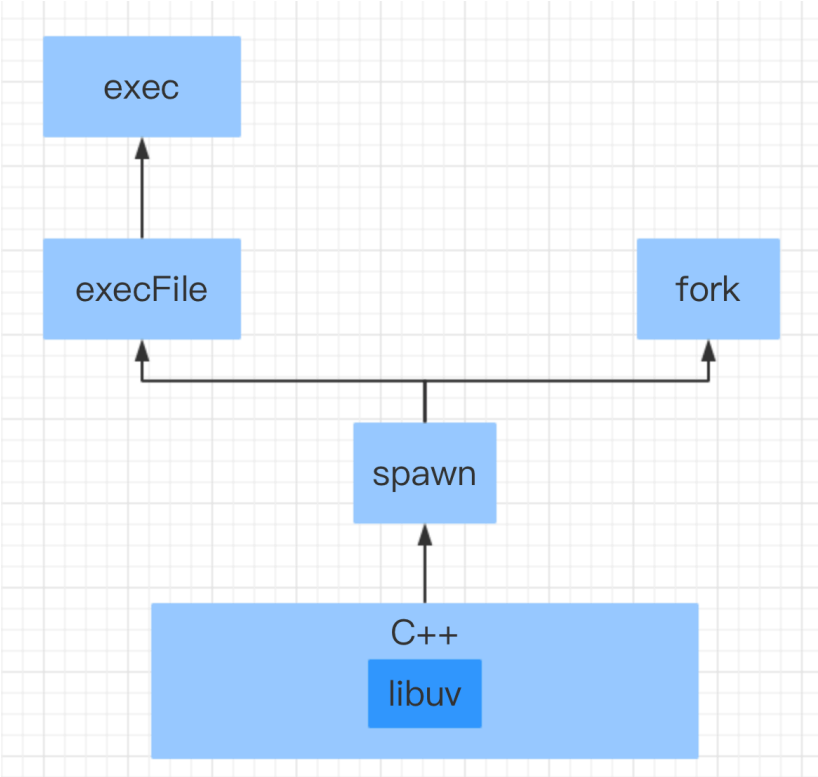
可创建多进程，充分利用单机的多核计算资源

概念

child_process是node的内置模块，用于创建子进程，该模块提供了4种方法创建子进程,实现了单机node集群,可以用下面的图来描述这4种方法的区别。


非主进程中的应用
子进程应用类型？
非Node应用
Node应用
执行应用后的返回形式？
流的形式返回
回调函数的形式返回
能否直接执行shell命令？
否
否
是
否
spawn
execFile
exec
fork

四个方法之间的关系



从图可知，这些方法都是对spawn方法的复用，spawn方法底层调用了libuv进行进程管理。

spawn

- 适用于**进程输入、输出数据量比较大**的情况（它支持steam的方式，而exec/execFile都是Buffer，不支持stream）。
- 执行**结果以流的形式返回**
- spawn创建的子进程，继承自EventEmitter。同时子进程具有三个输入输出流：stdin、stdout、stderr，可实时获取子进程的输入输出和错误信息。

看个例子：

spawn.js
```
// parent.js
let child_process = require('child_process');
const p = child_process.spawn(
  'node', // 需要执行的命令
  ['spawn_child.js', 'a', 'b'], // 传递的参数
  {}
);
console.log('child pid:', p.pid);
p.on('exit', code => {
  console.log('exit:', code);
});
// 父进程的输入直接pipe给子进程（子进程可以通过proce
process.stdin.pipe(p.stdin);

// 子进程的错误输出pipe给父进程的错误输出
p.stderr.pipe(process.stderr);

// 子进程的输出pipe给父进程的输出
p.stdout.pipe(process.stdout);
```

spawn_child.js
```
// child.js
console.log('child argv: ', process.argv);
process.stdin.pipe(process.stdout);
```

输出结果：

```
qiaochunmei@qiaochunmeideMacBook-Pro ~/myProject/node-process-study node spawn.js
child pid: 21987
child argv:  [ '/usr/local/bin/node',
  '/Users/qiaochunmei/myProject/node-process-study/spawn_child.js',
  'a',
  'b' ]
node进程
node进程
```

exec/execFile

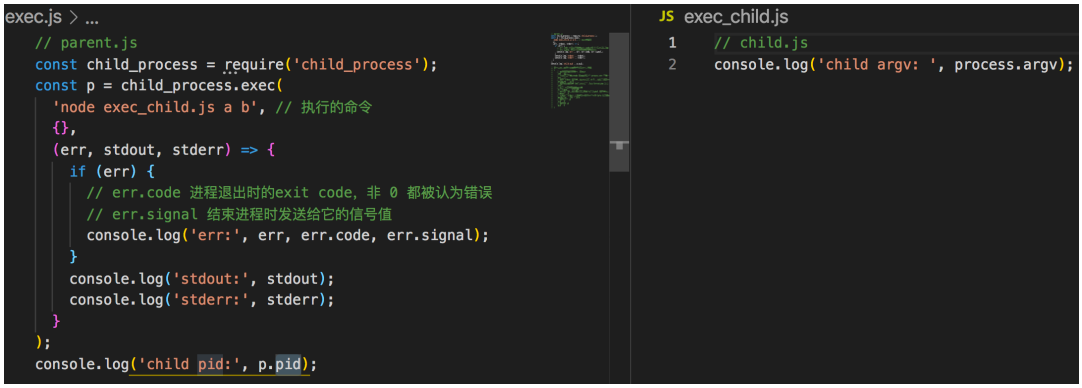
execFile(file, args, options, callback)和exec(command, options, callback)执行结果均以回调形式返回。

- exec 和 execFile 类似，使用一个 Buffer 来存储进程执行后的标准输出结果，在callback里面获取到;
- exec会首先创建一个新的shell进程出来，然后执行command；execFile则是直接将可执行的file创建为新进程执行。 所以，execfile会比exec高效一些;
- exec适合执行shell命令， 然后获取输出，但是 execFile不是，因为它实际上只接受了一个可执行的命令，然后执行;

注意：

为什么把这两个放到一起，是因为exec最后调用的就是execFile方法，相同点是执行的是非node应用，且执行后结果以回调函数形式返

回。不同点exec是直接执行的一段shell命令，而execFile是执行的一个应用。



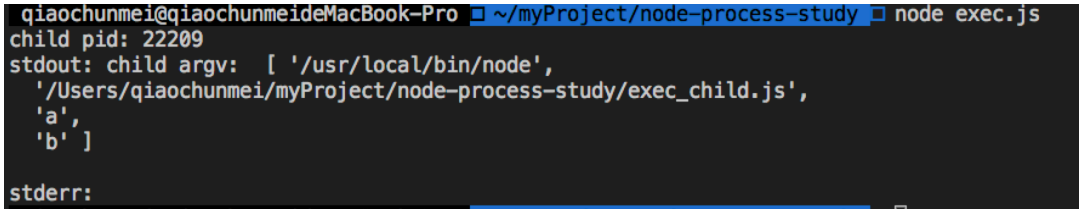
exec.js > ...

```
// parent.js
const child_process = require('child_process');
const p = child_process.exec(
  'node exec_child.js a b', // 执行的命令
  {},
  (err, stdout, stderr) => {
    if (err) {
      // err.code 进程退出时的exit code, 非 0 都被认为错误
      // err.signal 结束进程时发送给它的信号值
      console.log('err:', err, err.code, err.signal);
    }
    console.log('stdout:', stdout);
    console.log('stderr:', stderr);
  }
);
console.log('child pid:', p.pid);
```

exec_child.js

```
// child.js
console.log('child argv: ', process.argv);
```

输出结果：



```
qiaochunmei@qiaochunmeideMacBook-Pro ~/myProject/node-process-study node exec.js
child pid: 22209
stdout: child argv:  [ '/usr/local/bin/node',
  '/Users/qiaochunmei/myProject/node-process-study/exec_child.js',
  'a',
  'b' ]

stderr:
```

我们可以这样理解下，echo是linux系统的一个自带命令，我们直接可以在命令行执行：

```
echo hello world
```

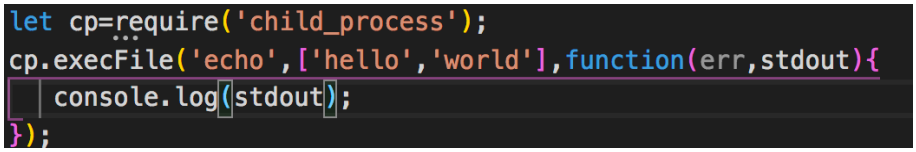
结果在命令行中会打印出hello world

通过exec来实现

```
let cp=require('child_process');
cp.exec('echo hello world',function(err,stdout){
  console.log(stdout);
});
```

执行这个js，结果会输出hello world,我们会发现exec的第一个参数，跟shell命令完全相似

通过execFile来实现



```
let cp=require('child_process');
cp.execFile('echo',['hello','world'],function(err,stdout){
  console.log(stdout);
});
```

execFile执行名为echo的应用，然后传入参数。execFlie会在process.env.PATH的路径中依次寻找是否有名为'echo'的应用，找到后就会执行。默认process.env.PATH路径中包含了'usr/local/bin',而这个'usr/local/bin'目录中就存在名为'echo'的程序，传入hello和world两个参数，执行后返回。

注意：

像exec那样，直接执行一段shell是非常不安全的，而execFile在传入参数的同时，会检测传入实参执行的安全性，如果存在安全性问题，会抛出异常。除了execFile外，spawn和fork也都不能直接执行shell，因此安全性较高。

fork

fork创建子进程会衍生新的node进程，产生一个新的V8实例，所以执行fork方法需要指定一个js文件。通过fork创建子进程后，父子进程之间直接会创建一个IPC（进程间通信）通道，方便父子进程直接通信，在js层使用 `process.send(message)` 和 `process.on('message', msg => {})` 进行通信，下面会具体说到。

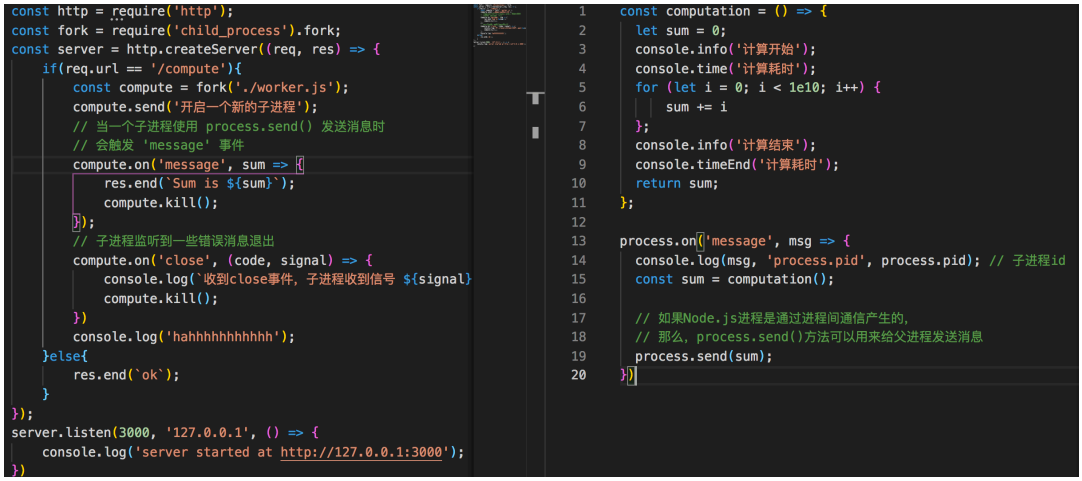
调用fork的进程为父进程，fork 出来的是子进程。

子进程的输入/输出操作执行完毕后，不会自动退出，必须用process.exit()方法显式退出

fork开启子进程 Demo

fork开启子进程解决文章刚开始计算耗时造成线程阻塞问题

在进行compute计算时创建子进程，子进程计算完成通过`send`方法将结果发送给主进程，主进程通过`message`监听到信息后处理并退出。



```
const http = require('http');
const fork = require('child_process').fork;
const server = http.createServer((req, res) => {
    if(req.url == '/compute'){
        const compute = fork('./worker.js');
        compute.send('开启一个新的子进程');
        // 当一个子进程使用 process.send() 发送消息时
        // 会触发 'message' 事件
        compute.on('message', sum => {
            res.end(`Sum is ${sum}`);
            compute.kill();
        });
        // 子进程监听到一些错误消息退出
        compute.on('close', (code, signal) => {
            console.log(`收到close事件，子进程收到信号 ${signal}
            compute.kill();
        })
        console.log('hahhhhhhhhhh');
    }else{
        res.end(`ok`);
    }
});
server.listen(3000, '127.0.0.1', () => {
    console.log('server started at http://127.0.0.1:3000');
})
```

```
const computation = () => {
  let sum = 0;
  console.info('计算开始');
  console.time('计算耗时');
  for (let i = 0; i < 1e10; i++) {
      sum += i
  };
  console.info('计算结束');
  console.timeEnd('计算耗时');
  return sum;
};

process.on('message', msg => {
  console.log(msg, 'process.pid', process.pid); // 子进程id
  const sum = computation();

  // 如果Node.js进程是通过进程间通信产生的，
  // 那么，process.send()方法可以用来给父进程发送消息
  process.send(sum);
})
```

输出：

```
qiaochunmei@bjdhj-1-10 □ ~/myProject/node-process-study □ node master.js
server started at http://127.0.0.1:3000
hahhhhhhhhhh
开启一个新的子进程 process.pid 3881
计算开始
计算结束
计算耗时： 11551.748ms
收到close事件，子进程收到信号 SIGTERM 而终止，退出码 null
```

cluster模块

cluster意思是集成，主要集成了两个方面，第一集成了child_process.fork方法创建node子进程，第二集成了根据多核CPU创建子进程后，自动控制负载均衡。它是node本身的一个模块，用于多核CPU环境下多进程的负载均衡。`cluster`模块可创建共享服务器端口的子进程。

特点：

- 可以共享tcp连接
- 自带负载均衡(内置一个负载均衡器，采用Round-robin算法协调各个worker进程之间的负载)
- 通过主进程监听端口和分发请求，子进程负责请求的处理

cluster 开启子进程Demo

创建与CPU数目相同的服务端实例，来处理客户端请求。注意，它们

监听的都是同样的端口

```
const http = require('http');
const numCPUs = require('os').cpus().length;
const cluster = require('cluster');
if(cluster.isMaster){
    console.log('Master proces id is',process.pid);
    // fork workers,衍生工作进程。
    for(let i= 0;i<numCPUs;i++){
        cluster.fork();
    }
    cluster.on('online',function(worker){
        console.log('worker' + worker.process.pid + 'is online');
    })
    cluster.on('exit', (worker, code, signal) => {
        console.log(`工作进程 ${worker.process.pid} 已退出`);
        console.log('开始一个新进程');
        cluster.fork();
    });
}else{
    // Worker可以共享同一个TCP连接
    // 这里是一个http服务器
    http.createServer(function(req,res){
        res.writeHead(200);
        res.end('hello word');
    }).listen(8000);
    console.log(`工作进程 ${process.pid} 已启动`);
}
```

输出：

```
qiaochunmei@qiaochunmeideMacBook-Pro □ ~/myProject/node-process-study □ node cluster.js
Master process id is 22142
worker22143is online
worker22145is online
worker22144is online
工作进程 22145 已启动
工作进程 22143 已启动
worker22146is online
工作进程 22144 已启动
工作进程 22146 已启动
```

了解cluster模块，主要搞清楚三个问题：

1. master、worker如何通信?
2. 多个server实例，如何实现端口共享?
3. 多个server实例，来自客户端的请求如何分发到多个worker?

master、worker如何通信?

- master通过cluster.fork()创建 worker进程。cluster.fork()内部通过child_process.fork()创建子进程。

- master、worker进程是父、子进程关系，通过IPC通道通信

多个server实例，如何实现端口共享?

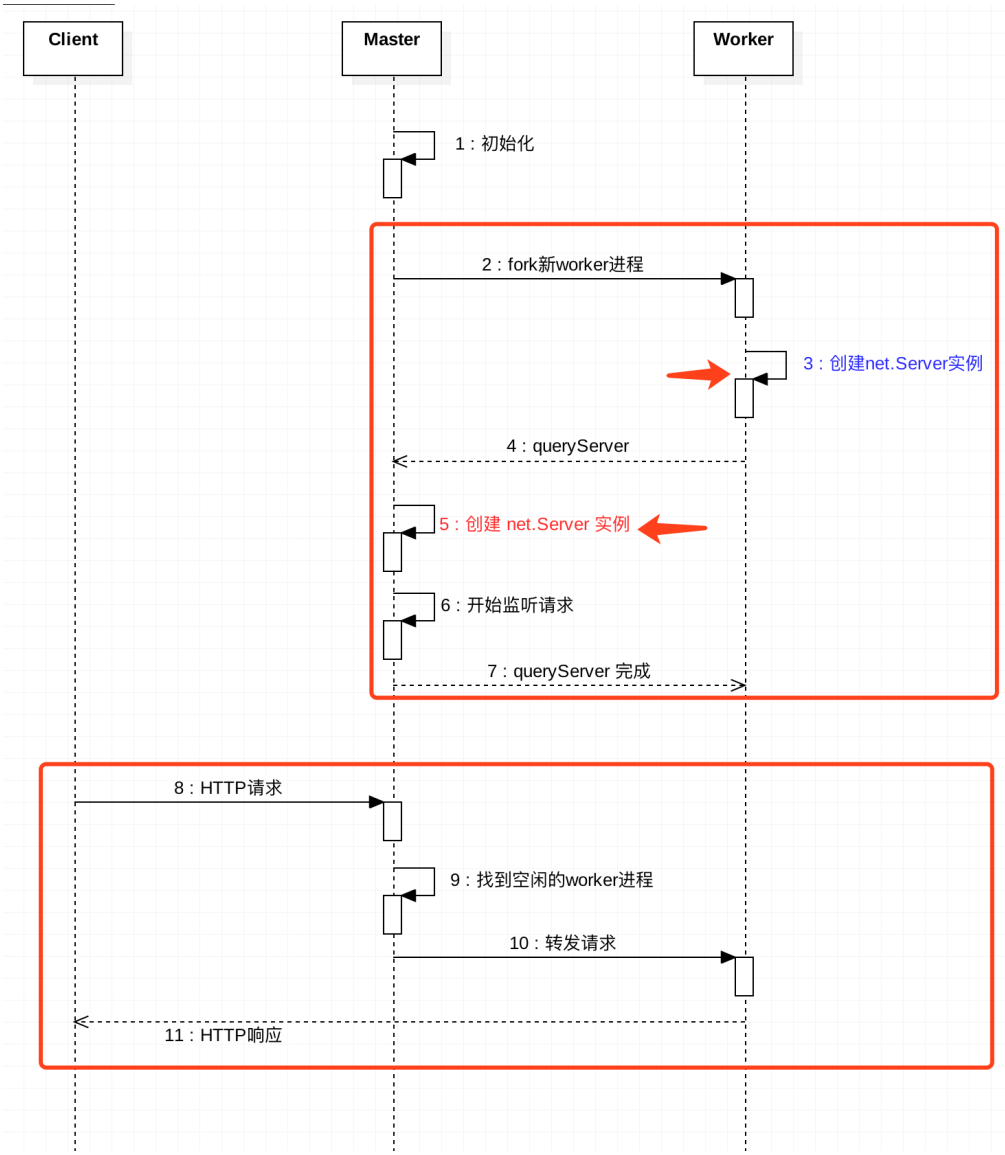
在上述例子中，多个woker中创建的server监听了同个端口8000。通常来说，多个进程监听同个端口，系统会报错。为什么上面的例子可以呢?

原因在于net.js源码中的listen方法通过listenInCluster方法来区分是父进程还是子进程，父进程会绑定端口号，而子进程不监听端口，只处理请求。

- master进程：在该端口上正常监听请求
- worker进程：创建server实例，然后通过IPC通道，向master发送消息，消息类型为queryServer，让master也创建server实例，并在该端口上监听请求，当请求进来时，master将请求转发给worker的server实例。

总结：master负责监听特定端口，并将客户请求转发给worker。

如下图所示：



多个server实例，来自客户端的请求如何分发到多个worker?

- 当worker创建server实例来监听请求时，都会通过IPC通道，在master上进行注册。当客户端请求到达，master将请求转发给对应的worker。
- 具体转发给哪个worker？是由转发策略决定的，可以通过环境变量NODE_CLUSTER_SCHED_POLICY设置，也可以在cluster.setupMaster(options)时传入。
- 默认的转发策略是轮询（SCHED_RR）。

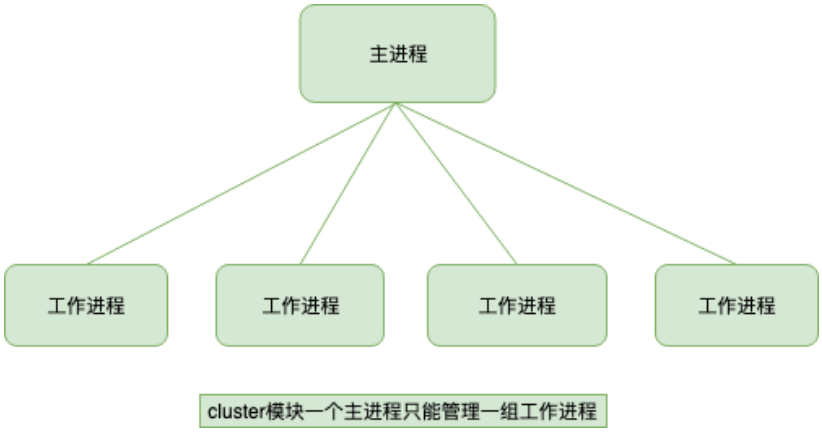
- 当有客户请求到达时，master会轮询一遍worker列表，找到第一个空闲的worker，然后将该请求转发给该worker。

cluster如何做到负载均衡

所有请求是通过master分配的，要保证服务器负载比较均衡的分配到各个worker上，就涉及到负载均衡策略。

- node.js默认采用的策略是Round-robin时间片轮转法。Round-robin是一种很常见的负载均衡算法，Nginx上也采用了它作为负载均衡策略之一
- 它原理很简单，每一次把来自用户的请求轮流分配给各个进程，从1开始，直到 N(worker 进程个数)，然后重新开始循环。这个算法的问题在于，它是假定各个进程或者说各个服务器的处理性能是一样的，但是如果请求处理间隔较长，就容易导致出现负载不均衡。因此通常在Nginx上采用另一种算法：**WRR**，加权轮转法。通过给各个服务器分配一定的权重，每次选出权重最大的，给其权重减 1，直到权重全部为0后，按照此时生成的序列轮询
- node中cluster实现了单机多进程上的负载均衡

cluster原理分析



- 通过调用fork方法创建子进程，该方法与child_process中的fork

是同一个方法。

- 采用的是经典的主从模型，cluster会创建一个master，然后根据你指定的数量复制出多个子进程，`cluster.isMaster`属性判断当前进程是否是主进程。由master进程来管理所有的子进程，主进程不负责具体的任务处理，主要负责调度和管理。
- 内置的负载均衡更好地处理了线程之间的压力，该负载均衡使用了`Round-robin`算法（也被称之为循环算法）。当使用Round-robin调度策略时，master接受所有传入的连接请求，然后将相应的TCP请求处理发送给选中的工作进程。

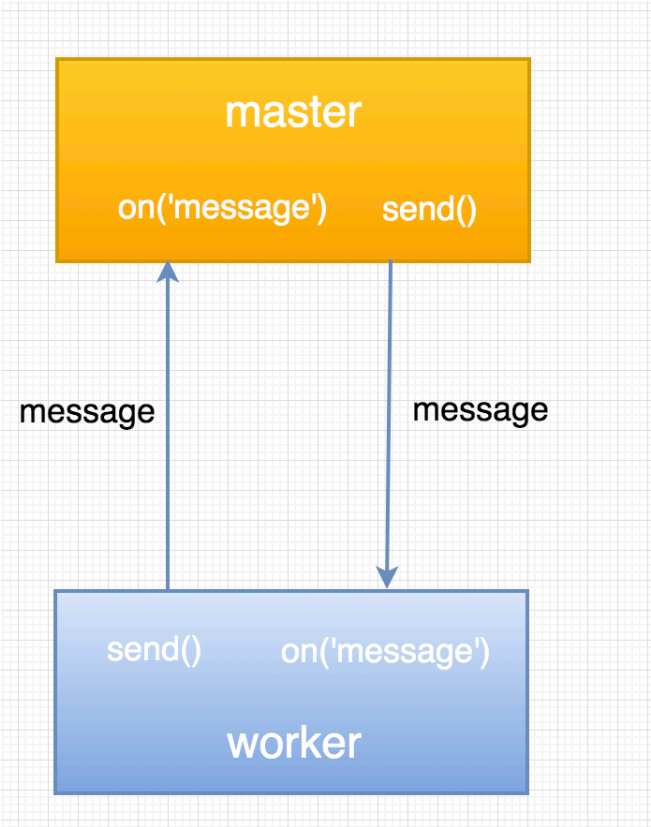
cluster模块的一个弊端

cluster内部隐时的构建TCP服务器的方式对使用者来说确实简单透明了很多，但是这种方式无法像使用child_process那样灵活，因为一个主进程只能管理一组相同的工作进程，而自行通过child_process来创建工作进程时，一个主进程可以控制多组进程。原因是child_process操作子进程时，可以隐式的创建多个TCP服务器。

进程通信

无论是child_process还是cluster，都需要主进程和工作进程之间进行通信。

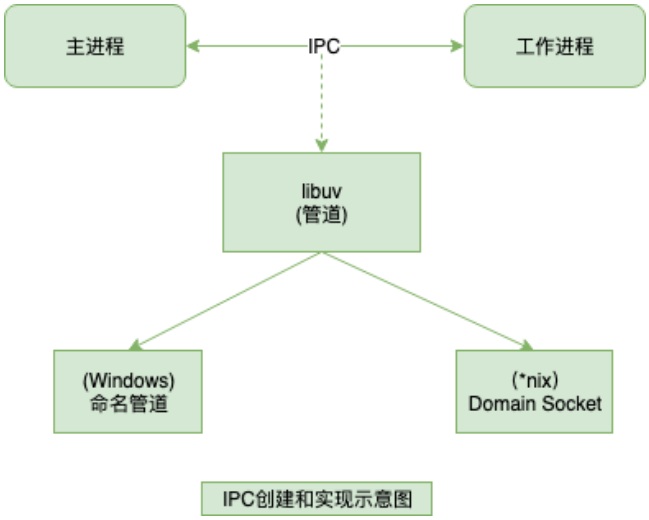
node父子进程之间可以通过on('message')和send()来实现通信，on('message')是监听message事件，当该进程收到其他进程发送的消息时，便会触发message事件，send()方法则是向其他进程发送信息。master进程中调用child_process的fork()方法后会得到一个子进程的实例，通过这个实例可以监听来自子进程的消息或者向子进程发送消息，worker进程则通过process对象监听来自父进程的消息或者向父进程发送消息。如下图所示：



进程间通信IPC

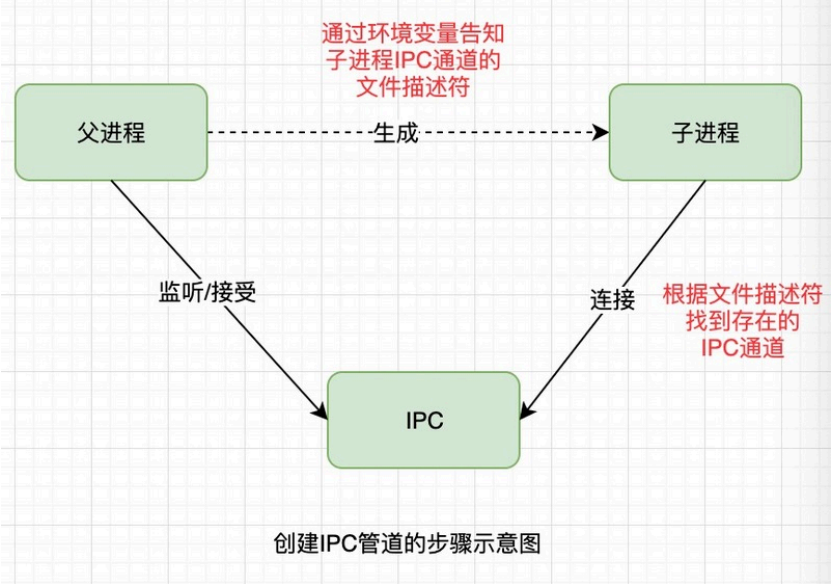
IPC的全称是Inter-Process Communication,即进程间通信。它的目的是为了让不同的进程能够互相访问资源并进行协调工作。node中实现IPC通道是依赖于libuv。windows下由命名管道(name pipe)实现，*nix系统则采用Unix Domain Socket实现。表现在应用层上的进程间通信就是message事件和send()方法。

IPC创建和实现示意图



IPC创建和实现示意图

IPC通信管道是如何创建的



创建IPC管道的步骤示意图

父进程在实际创建子进程之前，会创建IPC通道并监听它，然后才真正的创建出子进程，这个过程中也会通过环境变量（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。

注意：

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自动实现进程间通信，需要让其他进程也按照约定去连接这个已经创建好的IPC通道才行。

Node.js进程守护

什么是进程守护?

每次启动node程序都需要在命令窗口输入命令 `node app.js` 才能启动，但如果把命令窗口关闭则node程序服务就会立刻断掉。除此之外，当我们node服务意外崩溃了就不能自动重启进程了，所以需要通过某些方式来守护这个开启的进程，执行 node app.js 开启一个服务进程之后，还可以在这个终端上做些别的事情，且不会相互影响，当出现问题可以自动重启。

如何实现进程守护?

这里说一些第三方的进程守护工具，目前最常见的线上部署node项目的有forever，pm2这两种，都可以实现进程守护，底层也都是通过child_process模块和cluster模块实现的。

pm2的cluster模式也是基于child_process.fork进行的封装，和cluster API类似。直接通过 pm2 start index.js -i number, 直接启动多个node进程，结合cluster的优势，直接实现了一个比较完善的master-worker多进程模型，能满足我们的大部分需求。

使用场合:

- forever管理多个站点，每个站点访问量不大，不需要监控。
- pm2 网站访问量比较大，需要完整的监控界面。

总结

1.多进程 vs 多线程

属性	多进程	多线程	比较
数据	数据共享复杂，需要用IPC；数据是分开的，同步简单	因为共享进程数据，数据共享简单，同步复杂	各有千秋
CPU、内存	占用内存多，切换复杂，CPU利用率低	占用内存少，切换简单，CPU利用率高	多线程更好
销毁、切换	创建销毁、切换复杂，速度慢	创建销毁、切换简单，速度很快	多线程更好
coding	编码简单、调试方便	编码、调试复杂	编码、调试复杂
可靠性	进程独立运行，不会相互影响	线程同呼吸共命运	多进程更好
分布式	可用于多机多核分布式，易于扩展	只能用于多核分布式	多进程更好

2.实际生产中一个健壮的多进程模型需要考虑多个因素：

- 负载均衡，高并发是将请求平均分配给多个子进程。
- 子进程监听同一端口，减少句柄的浪费。
- 进程安全重启(平滑重启、限量重启)
- 工作进程存活状态管理
- 进程性能优化
- 多进程模式下定时任务处理等

3.child_process、cluster模块

- 无论是child_process还是cluster，都是为了解决node实例单线程运行，无法利用多核 CPU 的问题而出现的。核心就是父进程（即master进程）负责监听端口，接收到新的请求后将其分发给下面的worker进程。
- 利用child_process和cluster能够很好地实现Master-Worker模式多进程架构，实现单机服务器集群，充分利用多核CPU资源。通过进程通信能够实现进程管理、重启以及负载均衡，从而提高集群的稳定性和健壮性。

4.child_process中4个创建子进程方法比较

exec/execFile: 使用Buffer来存储进程的输出，可以在回调函数中获取

输出结果，不太适合数据量大的情况，可以执行任何命令； 不创建V8实例。

spawn: 支持stream方式操作输入输出，适合数据量大的情况； 可以执行任何命令； 不创建v8实例； 可以创建常驻的后台进程。适用于图像处理、二进制数据处理。

fork: spawn的一个特例； 只能执行node脚本； 会创建一个V8实例；会建立父子进程的IPC通道，能够进行通信。

以上四个方法在创建子进程后，均会返回子进程对象，他们的差别如下

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	×