

[压测 Cluster 的并发负载]

Node 的集群 – cluster

本节目标：【压测 cluster 的集群负载能力】 - 所谓双拳难敌四手，cluster 的集群扩展可以分摊利用多核，健壮可扩展有了可能。

我们都知道 Node 是事件驱动的异步服务模型，高效的同时也很脆弱，因为所有的事情都是在一个单线程中完成的，一旦这个单线程挂了，那么整个服务就挂了，或者有点这个单线程里有个非常耗时的同步任务，那么其他的请求进来也会阻滞在这里了，这时候我们就希望能充分利用计算机的多核优势，多起几个独立的进程，每个进程都像是一个伏地魔的一个魂，让我们的服务有多条命，就算是一个挂了，整个服务还不至于瘫痪，而且还可以把压力分摊到每个进程上面，整体服务更加健壮，也能支撑更多的并发。

幸运的是，在 Node 里面，提供 cluster 这个模块，来实现服务集群的扩展，具体怎么用呢，我们先起一个简单的服务器来返回一段文本，同时里面放一个略大的数组来阻滞下代码运行。

起一个简单的 HTTP Server

先来实现一个略微耗时的任务：

```
const t1 = Date.now()
// 来用一个 1 百万长度的数组来模拟耗时操作
for (var i = 0; i < 1000000; i++) {}
const t2 = Date.now()
// 最后打印下耗时操作作用时
console.log('耗时', t2 - t1, '毫秒')
```

我的电脑打印后是这样的结果：耗时 3 毫秒

然后我们起一个 Server，把任务丢进去作为响应返回：

```
// 通过 http 创建一个服务器实例
require('http').createServer((req, res) => {
  for (var i = 0; i < 1000000; i++) {}
  // 返回一段文本
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('经过一个耗时操作，这是返回的一段文本\n')
}).listen(5000, '127.0.0.1', () => console.log('服务启动了'))
```

在命令行 `node server.js` 服务开起来后，我们压测一下，压测的话，大家可以使用 [Apache ab](http://httpd.apache.org/docs/2.2/programs/ab.html) (<http://httpd.apache.org/docs/2.2/programs/ab.html>)、[siege](https://www.joedog.org/siege-home/) (<https://www.joedog.org/siege-home/>)、[wrk](https://github.com/wg/wrk) (<https://github.com/wg/wrk>) 等等，具体教程大家参考官方文档，我们这里使用一个 Node 的简单压测工具 [autocannon](https://github.com/mcollina/autocannon) (<https://github.com/mcollina/autocannon>)，首先把它安装到本地：

```
# 安装 autocannon 到全局
npm i autocannon -g
```

安装后，通过 `node server.js` 开启服务，同时再开一个命令行窗口，输入下面命令运行：

```
autocannon -c 1000 -p 10 http://127.0.0.1:5000
```

这些参数意思是：

-c 是并发连接的数量，默认 10，我们指定为 1000

-p 指定每个连接的流水线请求数，默认是 1，我们指定为 10

在我电脑上我压测了 3 次，结果如下：

```
➔ ~ autocannon -c 1000 -p 10
http://127.0.0.1:5000
Running 10s test @ http://127.0.0.1:5000
1000 connections with 10 pipelining factor
# A 接口的延迟程度
Stat      2.5% 50%  97.5%   99%      Avg      Stdev
Max
Latency 0 ms 0 ms 4061 ms 4652 ms 368.29 ms
1248.24 ms 9176.69 ms
# B 每秒能处理的请求数 TPS
Stat      1%      2.5%   50%    97.5%  Avg
Stdev  Min
Req/Sec  1591    1591    1918    1940    1888.1
101.11  1591
# C 每秒返回的字节数
Bytes/Sec 288 kB 288 kB 347 kB 351 kB 342 kB 18.3
kB 288 kB

19k requests in 10.15s, 3.42 MB read
760 errors (730 timeouts)

➔ ~ autocannon -c 1000 -p 10
http://127.0.0.1:5000
Stat      2.5% 50%  97.5%   99%      Avg      Stdev
Max
Latency 0 ms 0 ms 4895 ms 4908 ms 373.95 ms
1226.24 ms 5582.86 ms
Stat      1%      2.5%   50%    97.5%  Avg
Stdev  Min
Req/Sec  1641    1641    1910    1970    1872.1
```

```

112.94 1641
Bytes/Sec 297 kB 297 kB 346 kB 357 kB 339 kB 20.4
kB 297 kB
19k requests in 10.16s, 3.39 MB read
875 errors (870 timeouts)

➔ ~ autocannon -c 1000 -p 10
http://127.0.0.1:5000
Stat      2.5% 50% 97.5% 99%      Avg      Stdev
Max
Latency 0 ms 0 ms 4729 ms 4748 ms 370.43 ms
1208.29 ms 5539.08 ms
Stat      1%      2.5% 50%      97.5% Avg
Stdev    Min
Req/Sec   1631    1631    1961    1980    1928.1
101.4    1631
Bytes/Sec 295 kB 295 kB 355 kB 358 kB 349 kB 18.3
kB 295 kB
19k requests in 10.14s, 3.49 MB read
600 errors (590 timeouts)

```

压测的指标分为三部分，也就是 ABC，延迟越低，TPS 越高，每秒返回的字节数越多，就说明服务的响应能力越好，性能越好。

压测的结果不太稳定，但大体上可以看到，我们单核跑这个服务时候，延迟 4 秒多才能有返回，同时每秒处理的请求个数有 2 千上下，能吞吐响应的字节数，在二三百 KB 之内徘徊，在 10 秒内能响应的请求数有 2 万左右，同时还伴随有几百个超时错误，这样的结果不是太理想，我们改用 cluster 起服务下看看效果。

通过 cluster 启动 HTTP 服务

Node cluster 的用法非常简单，启动服务文件的时候，判断是否是 Master 模式，如果是则直接调用 cluster fork 来创建多个服务实例，如果不是 Master，就直接启动一个服务器实例，我们稍后再来了解这些概念，先看下被 cluster 优化后的代码：

```
const http = require('http')
// 加载拿到 cluster 模块
const cluster = require('cluster')
// 通过 os 模块拿到当前计算机上的 cpu
const cpus = require('os').cpus()

// cluster 能拿到当前是否是 master 模式
if (cluster.isMaster) {
  // master 下，对每个 cpu 都 fork 一个进程
  // 相当于是把 cpu 个数都吃满，充分利用多核优势
  for (let i = 0; i < cpus.length; i++) {
    cluster.fork()
  }
} else {
  // 如果不是 master 模式，则每个子进程都会启动这个服务
  // 相当于有多少个 cpu，fork 了多少个进程，这里就会有多少个服务器
  http.Server((req, res) => {
    for (var i = 0; i < 1000000; i++) {}
    res.statusCode = 200
    res.setHeader('Content-Type', 'text/plain')
    res.end('经过一个耗时操作，这是返回的一段文本\n')
  }).listen(5000, () => console.log('服务启动了'))
}
```

然后依然在命令行窗口中执行 node server.js，然后新开一个窗口，进行压测：

➔ ~ autocannon -c 1000 -p 10

http://127.0.0.1:5000

Stat	2.5%	50%	97.5%	99%	Avg	Stdev
------	------	-----	-------	-----	-----	-------

Max

Latency	0 ms	0 ms	1132 ms	1164 ms	101.92 ms	354.6 ms
---------	------	------	---------	---------	-----------	----------

Stat	1%	2.5%	50%	97.5%	Avg
------	----	------	-----	-------	-----

Stdev	Min
-------	-----

Req/Sec	6371	6371	7263	7383	7187.6
---------	------	------	------	------	--------

Bytes/Sec	1.15 MB	1.15 MB	1.31 MB	1.34 MB	1.3 MB
-----------	---------	---------	---------	---------	--------

	51.9 kB	1.15 MB
--	---------	---------

72k requests in 10.17s, 13 MB read

640 errors (640 timeouts)

➔ ~ autocannon -c 1000 -p 10

http://127.0.0.1:5000

Stat	2.5%	50%	97.5%	99%	Avg	Stdev
------	------	-----	-------	-----	-----	-------

Max

Latency	0 ms	0 ms	1195 ms	1274 ms	113.6 ms	376.96 ms
---------	------	------	---------	---------	----------	-----------

Stat	1%	2.5%	50%	97.5%	Avg
------	----	------	-----	-------	-----

Stdev	Min
-------	-----

Req/Sec	6459	6459	7391	7471	7294.4
---------	------	------	------	------	--------

Bytes/Sec	1.17 MB	1.17 MB	1.34 MB	1.35 MB	1.32 MB
-----------	---------	---------	---------	---------	---------

	51.4 kB	1.17 MB
--	---------	---------

73k requests in 10.17s, 13.2 MB read

560 errors (560 timeouts)

➔ ~ autocannon -c 1000 -p 10

http://127.0.0.1:5000

Stat	2.5%	50%	97.5%	99%	Avg	Stdev
------	------	-----	-------	-----	-----	-------

```
Max
Latency 0 ms 0 ms 1355 ms 1443 ms 119.83 ms
396.95 ms 9926.91 ms
Stat      1%      2.5%      50%      97.5%      Avg
Stdev    Min
Req/Sec   6359      6359      7259      7371      7176.8
280.34   6358
Bytes/Sec 1.15 MB 1.15 MB 1.31 MB 1.33 MB 1.3 MB
50.9 kB 1.15 MB
72k requests in 10.19s, 13 MB read
800 errors (800 timeouts)
```

同样压测了 3 次，发现超时错误的次数依然是几百个，但是服务的整体响应能力，从 10 秒响应的 2 万个，增长到了 7 万多个，翻了 3 倍多，同时延迟时间也从 4 秒降到了 1 秒多，每秒的处理次数也从 2 千增长到了 7 千，响应的字节数从二三百 KB 增长到了 1 MB 多，整体的服务性能改善还是非常可观的，这就是 Node cluster 带给我们的收益，想想还是很激动的。

关于 cluster

在刚才的测试里面，起到关键作用的一句代码就是 `cluster.fork()`，通过 fork 按照 cpu 的个数，创建了多个子进程，也就是 child process，我们管它叫 worker，这些 worker 会共享同一个服务器端口，也就是 server port，而能做到这一点离不开主进程的调度，也就是 master process。

对于 cluster 模块，它里面有几个事件，其中比较常见的一个是 online 事件，当 worker 被 fork 出来发送 online message，而 exit 会在一个 worker 进程杀掉挂掉的时候会被触发，我们来一段代码感受一下：

```
const cluster = require('cluster')
```

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

// 通过 if else 区分下主进程和子进程各自的启动逻辑
if (cluster.isMaster) masterProcess()
else childProcess()

function masterProcess () {
  // 可以选择只启动 2 个 worker
  for (let i = 0; i < 2; i ++) {
    let worker = cluster.fork()
  }

  // 进程创建成功 则触发 online 事件
  cluster.on('online', worker => {
    console.log('子进程 ' + worker.process.pid + ' 创建成功')
  })

  // 进程退出 则触发 exit 事件
  cluster.on('exit', (worker, code, signal) => {
    console.log(`子进程 ${worker.process.pid} 退出`)
  })
}

function childProcess () {
  console.log(`子进程开始 ${process.pid} 开始启动服务器...`)

  http.Server((req, res) => {
    res.statusCode = 200
    res.setHeader('Content-Type', 'text/plain')
    console.log('来自子进程 id 为 ' +
```



```
cluster.worker.id + ' 的响应')
    res.end('Hello Juejin!')
    process.exit(1)
  }).listen(5000, () => {
    console.log('子进程 ' + process.pid + ' 已成功监
听 5000 端口')
  })
}
```

当我们访问服务的时候，可以拿到返回的 Hello Juejin，但同时服务器也退出了，退出的时候，自然 cluster 启动的子进程也会退出，所以打印了如下的这段日志：

```
~ curl http://127.0.0.1:5000

子进程 16725 创建成功
子进程 16726 创建成功
子进程开始 16726 开始启动服务器...
子进程开始 16725 开始启动服务器...
子进程 16726 已成功监听 5000 端口
子进程 16725 已成功监听 5000 端口
来自子进程 id 为 2 的响应
子进程 16726 退出
# 访问浏览器 http://127.0.0.1:5000 可能会多出一个响应
来自子进程 id 为 1 的响应
子进程 16725 退出
```

浏览器请求的时候，可能会多发一个 favicon 的请求，等于是两个请求，第一个子进程退出后，第二个子进程会接管之后而来的其他请求，响应后也会退出，所以会多打印两行日志。

那 worker 负责干活，master 呢？master 在这里的作用，就是启动多个 worker，然后来调度这些 worker，然后在主进程和子进程之间通过 IPC 实现进程间的通信，但是子进程之间的任务怎么分配

呢？我们上面的代码案例中，如果把 `process.exit(1)` 拿掉后，然后不断的刷新浏览器，会发现实际上真正干活的子进程，一会是 1 一会是 2，并没有什么明显的规律，只是看上去大概符合 1:1 的平均分配，这里的分配就是 `cluster` 底层做的，用的调度算法是 RR 算法，也就是 Round-Robin 算法，调用的地方在 `lib/internal/cluster/child.js` 源码 [93 行](https://github.com/nodejs/node/blob/v10.x/lib/internal/cluster/child.js#L93) `cluster.getServer` (<https://github.com/nodejs/node/blob/v10.x/lib/internal/cluster/child.js#L93>) 这里：

```
cluster._getServer = function(obj, options, cb) {
  let address = options.address;

  // Resolve unix socket paths to absolute paths
  address = path.resolve(address);

  const indexesKey = [address,
                      options.port,
                      options.addressType,
                      options.fd ].join(':');

  if (indexes[indexesKey] === undefined)
    indexes[indexesKey] = 0;
  else
    indexes[indexesKey]++;

  const message = util._extend({
    act: 'queryServer',
    index: indexes[indexesKey],
    data: null
  }, options);

  message.address = address;
```

```
if (obj._getServerData)
  message.data = obj._getServerData();

send(message, (reply, handle) => {
  if (typeof obj._setServerData === 'function')
    obj._setServerData(reply.data);

  if (handle)
    shared(reply, handle, indexesKey, cb); //
Shared listen socket.
  else
    rr(reply, indexesKey, cb); // Round-robin.
});

obj.once('listening', () => {
  cluster.worker.state = 'listening';
  const address = obj.address();
  message.act = 'listening';
  message.port = address && address.port ||
options.port;
  send(message);
});
};
```

cluster 如果挂了怎么办

我们上面代码案例中通过 `process.exit` 来退出程序了，如果是其他异常导致子进程异常呢，来看如下代码：

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

if (cluster.isMaster) masterProcess()
else childProcess()

function masterProcess () {
  // 只启动 1 个 worker
  const worker = cluster.fork()
  cluster.on('exit', (worker, code, signal) => {
    console.log(`子进程 ${worker.process.pid} 挂了`)
  })
}

function childProcess () {
  http.Server((req, res) => {
    console.log('子进程 ' + cluster.worker.id + ' 在响应')
    // 此处发生异常
    throw new Error({})
    res.end('Hello Juejin!')
  }).listen(5000, () => {
    console.log('子进程 ' + process.pid + ' 监听中')
  })
}
```

我们访问 `http://127.0.0.1:5000`，会看到如下的服务报错：

```
子进程 20739 监听中
子进程 1 在响应
/Users/black/Downloads/node-
10.x/juejin/server.js:18
    throw new Error({})
    ^
Error: [object Object]
    at Server.http.Server
(/Users/black/Downloads/node-
10.x/juejin/server.js:18:11)
    at Server.emit (events.js:182:13)
    at parserOnIncoming (_http_server.js:652:12)
    at HTTPParser.parserOnHeadersComplete
(_http_common.js:109:17)
子进程 20739 挂了
```

可以看到，能通过 cluster 的 exit 事件监听到子进程挂掉，那么我们就可以在 exit 的时候，再启动一个进程，改下代码成这样子：

```

const cluster = require('cluster')
const http = require('http')

if (cluster.isMaster) masterProcess()
else childProcess()

function masterProcess () {
  // 只启动 1 个 worker
  cluster.fork()
  cluster.on('exit', (worker, code, signal) => {
    console.log(`子进程 ${worker.process.pid} 挂了`)
  })
  if (code !== 0 && !worker.suicide) {
    cluster.fork()
    console.log('再启动一个新的子进程')
  }
}

function childProcess () {
  http.Server((req, res) => {
    console.log('子进程 ' + cluster.worker.id + ' 在响应')
    throw new Error({})
    res.end('Hello Juejin!')
  }).listen(5000, () => {
    console.log('子进程 ' + process.pid + ' 监听中')
  })
}

```

同样的请求后，我们观察终端打印的日志如下：

```
子进程 20956 监听中
子进程 1 在响应
/Users/black/Downloads/node-
10.x/juejin/server.js:22
    throw new Error({})
    ^
Error: [object Object]
    at Server.http.Server
(/Users/black/Downloads/node-
10.x/juejin/server.js:22:11)
    at Server.emit (events.js:182:13)
    at parserOnIncoming (_http_server.js:652:12)
    at HTTPParser.parserOnHeadersComplete
(_http_common.js:109:17)
子进程 20956 挂了
再启动一个新的子进程
子进程 20960 监听中
```

看到虽然子进程 20956 挂了，但是 子进程 20960 已经跑起来，可以继续接管后续的请求了。

有哪些能实现横向扩展 cluster 的工具

虽然我们 know cluster 的大概原理，但人肉来维护进程显然不是我们在学习 Node 初期可以深度掌握的技能，需要一些工具的配合，那么这里就给大家推荐两个工具，一个是 [pm2](https://github.com/Unitech/pm2) (<https://github.com/Unitech/pm2>)，一个是阿里的 Egg 框架自带的 egg-cluster，关于后者我们本册先不涉及，先来看下 pm2。

pm2 的安装特别简单：

```
# 安装到全局
npm i pm2 -g
```

[pm2 官方文档 \(https://pm2.io/doc\)](https://pm2.io/doc)也特别详尽，大家可以前往学习，我挑几个自己常用的介绍下。

pm2 启动服务器

推荐大家从配置文件启动，配置文件参考[官网 \(https://pm2.io/doc/en/runtime/guide/ecosystem-file/\)](https://pm2.io/doc/en/runtime/guide/ecosystem-file/)，从命令行启动非常简单：

```
pm2 start app.js -i 2
```

-i 后面跟的 2 表示启动 2 个 server 实例，如果输入 0 的话，则按照当前服务器它实际的 cpu 核数来启动多个 server，启动后，我们通过 pm2 ls 来看看已经启动的实例：

```
~ pm2 ls
```

Name	id	mode	status	↻	cpu	memory
app	0	N/A	cluster	online	0	19% 28.4 MB
app	1	N/A	cluster	online	0	0% 20.3 MB

pm2 实时扩容集群

如果我们发现线上的服务响应比较吃力，而 cpu 核数没有吃满的话，我们可以实时扩容集群，通过 scale 命令来实现：


```
pm2 scale app +1
[PM2] Scaling up application
```

Name	id	mode	status	↻	cpu	memory
app	0	N/A	cluster	online	0	0% 33.6 MB
app	1	N/A	cluster	online	0	0% 34.2 MB
app	2	N/A	cluster	online	0	0% 19.9 MB

这里的 +1 就是扩容一个服务实例，其实就是增加一个 cluster 的 worker 子进程，扩容后：

pm2 终止某个进程

有时候如果某个进程明显卡住了，或者线上负载不大，可以杀掉部分进程，通过：

```
e12-cluster git:(master) x pm2 stop 1
[PM2] Applying action stopProcessId on app [1]
(ids: 1)
[PM2] [app](1) ✓
```

Name	id	mode	status	↻	cpu	memory
app	0	N/A	cluster	online	0	0% 33.6 MB
app	1	N/A	cluster	stopped	0	0% 0 B
app	2	N/A	cluster	online	0	0% 33.6 MB

可以看到进程 ID 为 2 的 worker 已经是 stopped 状态。

pm2 平滑重启进程

有时候，如果想要某个比较吃内存的进程可以重启，或者想要所有的 worker 都重新启动，但是又希望不影响进程正常处理用户的请求，可以使用 pm2 的 gracefulReload 命令：

➔ pm2 reload app

Use --update-env to update environment variables

[PM2] Applying action reloadProcessId on app

[app](ids: 0,1,2)

[PM2] [app](1) ✓

[PM2] [app](0) ✓

[PM2] [app](2) ✓

➔ pm2 ls

Name	id	mode	status	↻	cpu	memory
app	0	N/A	cluster	online	1	6.8% 37.4 MB
app	1	N/A	cluster	online	0	6.8% 37.4 MB
app	2	N/A	cluster	online	1	6.4% 37.5 MB

这样所有的子进程又原地满血复活，当然也会存在说，某些进程上面的未处理连接或者任务的确很重，比如有一些大而重的文件 IO 或者数据库 IO 在等待，会导致 reload 失败，这时候也可以指定一个超时时间，命令会退化到 restart 模式，强制杀死进程再重启，或者我们可以在代码中再友好一些，当它收到 pm2 要重启的时候，在程序里面我们把一些任务清空掉然后让服务重启：

```
// pm2 会发出 SIGINT 事件，我们监听事件
process.on('SIGINT', function() {
  // 处理一些任务然后再信号交还给 PM2 来重启服务
  db.stop(function(err) {
    process.exit(err ? 1 : 0)
  })
})
```

小结

简单总结一下，我们现在了解到 cluster 可以分摊服务器的压力，可以最大的利用多核 CPU 的资源，从而实现并发和整体响应性能的提升，同时在服务的健壮性上，我们也可以通过监听子进程的异常来杀死或者启动一个新的子进程，从而实现了多进程多服务的有效负载。我们在生产环境中，也可以通过 pm2 这样的部署运维工具，来保持服务的自动重启和更简便的集群扩展，甚至可以使用它的高级功能如监控等等，对于一些不太复杂的系统我们就有这样的配套全家桶了。