Node.js 调试 GC 以及内存暴涨的分析

[nodejs](https://leokongwq.github.io/categories/nodejs/)

**前言**

最近写的一个功能在本地开发的时候没有明显问题，但是到真实环境测试的时候发现内存不断增长，并且增长很快，同时 CPU 占用也很高，接近单核心的 100% 。这对于一个大部分都是 IO 操作的进程显然是有问题的。所以尝试分析内存和 CPU 异常的原因。最终发现是因为生产者和消费者速度差异引起的缓冲区暴增。在 MySQL 连接对象的 Queue 中积压了大量的 Query，而不是内存泄漏。

**查看 Node.js 进程的 GC log**

node --trace\_gc --trace\_gc\_verbose test.js

61 ms: Scavenge 2.2 (36.0) -> 1.9 (37.0) MB, 1 ms [Runtime::PerformGC].

Memory allocator, used: 38780928, available: 1496334336

New space, used: 257976, available: 790600

Old pointers, used: 1556224, available: 0, waste: 0

Old data space, used: 1223776, available: 4768, waste: 0

Code space, used: 1019904, available: 0, waste: 0

Map space, used: 131072, available: 0, waste: 0

Cell space, used: 98304, available: 0, waste: 0

Large object space, used: 0, available: 1495269120

97 ms: Mark-sweep 11.7 (46.1) -> 5.4 (40.7) MB, 10 ms [Runtime::PerformGC] [GC in old space requested].

Memory allocator, used: 42717184, available: 1492398080

New space, used: 0, available: 1048576

Old pointers, used: 1390648, available: 165576, waste: 0

Old data space, used: 1225920, available: 2624, waste: 0

Code space, used: 518432, available: 501472, waste: 0

Map space, used: 60144, available: 70928, waste: 0

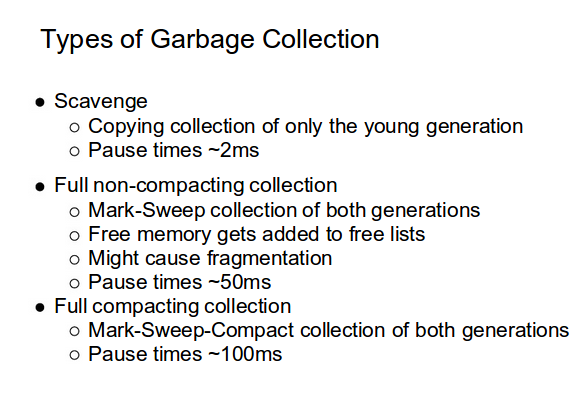
Cell space, used: 23840, available: 74464, waste: 0

Large object space, used: 3935872, available: 1491332864

**关于 Node.js 的 GC**

Node.js 的 GC 方式为分代 GC (Generational GC)。对象的生命周期由它的大小决定。对象首先进入占用空间很少的 new space (8MB)。大部分对象会很快失效，会频繁而且快速执行 Young GC (scavenging)*直接*回收这些少量内存。假如有些对象在一段时间内不能被回收，则进入 old space (64-128KB chunks of 8KB pages)。这个区域则执行不频繁的 Old GC/Full GC (mark-sweep, compact or not)，并且耗时比较长。(Node.js 的 GC 有两类：Young GC： 频繁的小量的回收；Old GC： 长时间存在的数据)

Node.js 最新增量 GC 方式虽然不能降低总的 GC 时间，但是避免了过大的停顿，一般大停顿也限制在了几十 ms 。

[](https://leokongwq.github.io/2016/11/08/nodejs-gc/v8_gc.png)

为了减少 Full GC 的停顿，可以限制 new space 的大小

--max-new-space-size=1024 (单位为 KB)

手动在代码中操作 GC (不推荐)

node --expose-gc test.js

修改 Node.js 默认 heap 大小

node --max-old-space-size=2048 test.js (单位为 MB)

Dump 出 heap 的内容到 Chrome 分析：

**安装库**

<https://github.com/bnoordhuis/node-heapdump>

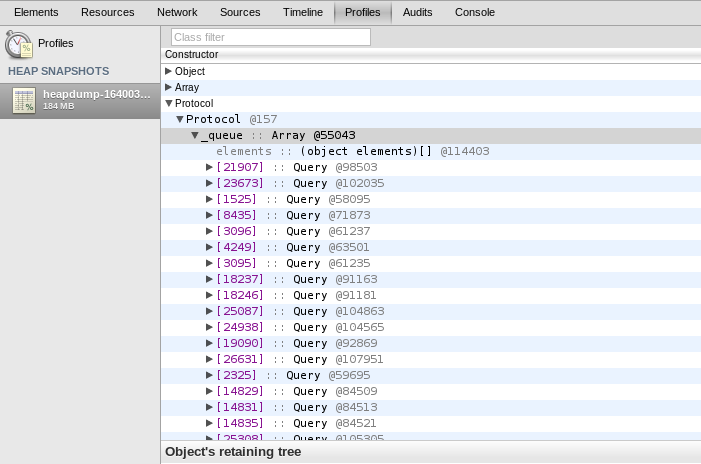
在应用的开始位置添加

var heapdump = require('heapdump');

在进程运行一小段时间后执行：

kill -USR2 <pid>

这时候就会在当前目录下生成 heapdump-xxxxxxx.heapsnapshoot 文件。  
将这个文件 Down 下来，打开 Chrome 开发者工具中的 Profiles，将这个文件加载进去，就可以看到当前 Node.js heap 中的内容了。

[](https://leokongwq.github.io/2016/11/08/nodejs-gc/heapdump_chrome_profiler.png)

可以看到有很多 MySQL 的 Query 堆积在处理队列中。内存暴涨的原因应该是 MySQL 的处理速度过慢，而 Query 产生速度过快。  
所以解决方式很简单，降低 Query 的产生速度。内存暴涨还会引起 GC 持续执行，占用了大量 CPU 资源。

node-mysql 库中的相关代码，其实应该限制 \_queue 的 size，size 过大则抛出异常或者阻塞，就不会将错误扩大。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 | Protocol.prototype.\_enqueue = function(sequence) {  if (!this.\_validateEnqueue(sequence)) {  return sequence;  }   this.\_queue.push(sequence);   var self = this;  sequence  .on('error', function(err) {  self.\_delegateError(err, sequence);  })  .on('packet', function(packet) {  self.\_emitPacket(packet);  })  .on('end', function() {  self.\_dequeue();  });   if (this.\_queue.length === 1) {  this.\_parser.resetPacketNumber();  sequence.start();  }   return sequence; }; |

在不修改 node-mysql 的情况下，加入生产者和消费者的同步，调整之后，内存不再增长，一直保持在不到 100M 左右，CPU 也降低到 10% 左右。

**Node.js 调试工具 node-inspector**

**安装：**

npm install -g node-inspector

**启动自己的程序：**

node --debug test.js

node --debug-brk test.js (在代码第一行加断点)

**启动调试器界面：**

node-inspector

打开 <http://localhost:8080/debug?port=5858> 可以看到执行到第一行的断点。右边为局部变量和全局变量、调用栈和常见的断点调试按钮，查看程序步进执行情况。并且你可以修改正在执行的代码，比如在关键的位置增加 console.log 打印信息。

**Node.js 命令行调试工具**

以 DEBUG 模式启动 Node.js 程序，类似于 GDB：

node debug test.js

debug> help

Commands: run (r), cont (c), next (n), step (s), out (o), backtrace (bt), setBreakpoint (sb), clearBreakpoint (cb),

watch, unwatch, watchers, repl, restart, kill, list, scripts, breakOnException, breakpoints, version

**Node.js 其他常用命令参数**

node --max-stack-size 设置栈大小

node --v8-options 打印 V8 相关命令

node --trace-opt test.js

node --trace-bailout test.js 查找不能被优化的函数，重写

node --trace-deopt test.js 查找不能优化的函数

**Node.js 的 Profiling**

V8 自带的 prof 功能：

npm install profiler

node --prof test.js

会在当前文件夹下生成 v8.log

**安装 v8.log 转换工具**

sudo npm install tick -g

**在当前目录下执行**

node-tick-processor v8.log

可以关注其中 Javascript 各个函数的消耗和 GC 部分

[JavaScript]:

ticks total nonlib name

67 18.7% 20.1% LazyCompile: \*makeF /opt/data/app/test/test.js:6

62 17.3% 18.6% Function: ~ /opt/data/app/test/test.js:9

42 11.7% 12.6% Stub: FastNewClosureStub

38 10.6% 11.4% LazyCompile: \* /opt/data/app/test/test.js:1

[GC]:

ticks total nonlib name

27 7.5%

**参考以及一些有用的链接**

<https://bugzilla.mozilla.org/show_bug.cgi?id=634503>  
<http://cs.au.dk/~jmi/VM/GC.pdf>  
<http://lifecs.likai.org/2010/02/how-generational-garbage-collector.html>  
<http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)#Na.C3.AFve_mark-and-sweep>  
[http://en.wikipedia.org/wiki/Cheney’s\_algorithm](http://en.wikipedia.org/wiki/Cheney%E2%80%99s_algorithm)  
<https://github.com/bnoordhuis/node-heapdump>  
<http://mrale.ph/blog/2011/12/18/v8-optimization-checklist.html>  
<http://es5.github.com/>  
<http://blog.caustik.com/2012/04/08/scaling-node-js-to-100k-concurrent-connections/>  
<https://hacks.mozilla.org/2013/01/building-a-node-js-server-that-wont-melt-a-node-js-holiday-season-part-5/>  
<https://gist.github.com/2000999>  
<http://www.jiangmiao.org/blog/2247.html>  
<http://blog.caustik.com/2012/04/08/scaling-node-js-to-100k-concurrent-connections/>  
<http://blog.caustik.com/2012/04/11/escape-the-1-4gb-v8-heap-limit-in-node-js/>  
<https://developers.google.com/v8/embed#handles>  
<https://hacks.mozilla.org/2012/11/fully-loaded-node-a-node-js-holiday-season-part-2/>  
<https://code.google.com/p/v8/wiki/V8Profiler>