你有关心过你的 Node.js 程序内存占用情况吗

Node.js 的内存限制到底是多大?

> 对 Node.js 的 API熟练程度可以让你走的快，但对 Node.js程序占用内存的深入理解可以让你走的更远 – Max

首先我们先检测一下内存使用情况，通过 `process.memoryUsage()` 每秒更新一次

```
setInterval(() => { console.log('Memory Usage:', process.memoryUsage());
}, 1000);
```

由于打印出来的是字节，可阅读性不好，我们将内存情况格式化成 MB:

```
function formatMemoryUsageInMB(memUsage) {
    return {
        rss: convertToMB(memUsage.rss),
        heapTotal: convertToMB(memUsage.heapTotal),
        heapUsed: convertToMB(memUsage.heapUsed),
        external: convertToMB(memUsage.external)
    };
}

const convertToMB = value => {
    return (value / 1024 / 1024).toFixed(2) + ' MB';
};

const logInterval = setInterval(() => {
    const memoryUsageMB =
formatMemoryUsageInMB(process.memoryUsage());
```

```
    console.log(`Memory Usage (MB):`, memoryUsageMB);
}, 1000);
```

然后我们可以每秒可以得到下面的输出

```
Memory Usage (MB): {
  rss: '30.96 MB', // 整个程序实际占用的操作系统内存，包括代码，数据，共享库等
  heapTotal: '6.13 MB', // JS 对象，数组等 Nodejs 动态分配的内存区域占用
                        // v8 将堆划分为 新生态，老生态进行不同的垃圾回收策略
  heapUsed: '5.17 MB',
  external: '0.39 MB'
}

Memory Usage (MB): {
  rss: '31.36 MB',
  heapTotal: '6.13 MB',
  heapUsed: '5.23 MB',
  external: '0.41 MB'
}
```

我们都知道 V8引擎的内存使用是受限的，不仅受到操作系统内存管理和资源分配策略的制约，更受到自身设定的限制，。

通过 `os.freemem()` 可以看到操作系统有多少空闲内存，但不代表都可以被 Node.js 程序占用。

```
console.log('Free memory:', os.freemem());
```

道对于64位系统，Nodejs V8 的默认最大老生代堆大小约为 1.4GB。这意味着即使你的操作系统有更多可用内存，V8也不会自动占用超过这个限制。

提示: 这个限制可以通过设置环境变量或者在启动 Node.js 时指定参数来改变。

例如，如果你想让V8使用更大的堆，你可以使用`--max-old-space-size`选项来指定：

```
node --max-old-space-size=4096 your_script.js
```

这个值需要根据你的实际情况和场景来设置，比如你只有一台大内存的机器，单机部署，和你有很多台小机器内存机器，分布式部署，这个值的设置肯定是不同的。

我们做个测试，放一个数组，无限向其中追加数据，直到内存溢出，让我们观察下什么时候溢出

```
const array = [];
while (true) {
    for (let i = 0; i < 100000; i++) {
        array.push(i);
    }
    const memoryUsageMB =
formatMemoryUsageInMB(process.memoryUsage());
    console.log(`Memory Usage (MB):`, memoryUsageMB);
}
```

这是我们直接运行程序得到的结果，在追加了一会儿数据后，程序崩溃

```
Memory Usage (MB): {
  rss: '2283.64 MB',
  heapTotal: '2279.48 MB',
  heapUsed: '2248.73 MB',
  external: '0.40 MB'
}
Memory Usage (MB): {
  rss: '2283.64 MB',
  heapTotal: '2279.48 MB',
  heapUsed: '2248.74 MB',
```

```
  external: '0.40 MB'
}


#
# Fatal error in , line 0
# Fatal JavaScript invalid size error 169220804
#
#
#
#FailureMessage Object: 0x7ff7b0ef8070
```

看到这里很迷茫？ 不是 1.4G 的内存限制吗？为什么占用了 2.多呢？实际上，Node.js的1.4GB 限制是基于V8引擎的一个历史限制，它适用于早期版本的V8和某些特定的配置。在现代版本的Node.js和V8中，Node.js默认会根据系统资源自动调整其内存使用。在某些情况下，它可能使用比1.4GB多得多的内存，特别是当处理大量数据或者运行内存密集型操作时。

当我们将内存限制设置为 512M 时候，差不多 rss 在 996 MB 的时候内存溢出。

```
Memory Usage (MB): {
  rss: '996.22 MB',
  heapTotal: '993.22 MB',
  heapUsed: '962.08 MB',
  external: '0.40 MB'
}
Memory Usage (MB): {
  rss: '996.23 MB',
  heapTotal: '993.22 MB',
  heapUsed: '962.09 MB',
  external: '0.40 MB'
}

<--- Last few GCs --->

[22540:0x7fd27684d000]     1680 ms: Mark-sweep 643.0 (674.4) -> 386.8 (419.4) MB, 172.2 / 0.0 ms  (average mu = 0.708, current mu = 0.668) allocation failure; scavenge might not succeed
[22540:0x7fd27684d000]     2448 ms: Mark-sweep 962.1 (993.2) -> 578.1 (610.7) MB, 240.7 / 0.0 ms  (average mu = 0.695, current mu = 0.687) allocation failure; scavenge might not succeed
```

<--- JS stacktrace --->

FATAL ERROR: Reached heap limit Allocation failed - JavaScript heap out of memory

```

总结：其实要说的准确点，Node.js 的内存限制是指 堆内存限制，也就是 V8 分配的 JS对象，数组这些的内存最大占用限制。

那堆内存大小决定了 Node.js 进程能占用多少内存吗？不！接着看。

我有一个 3G 的文件，能不能放到 Node.js 程序内存里？
---

那刚才测试也看到，数组最多放 2.多 G程序就崩了，那我有 3G 的文件，是不是 Node.js 程序不能一次性放内存里？

能放！

刚才通过 `process.memoryUsage()` 我们看到一个 `external` 内存，这个内存是显示被这个 Node.js 进程占用，但是又不是 V8 分配的。那只要将 3G 文件放这里就没有内存限制了。怎么做呢？可以通过 Buffer， Buffer 是一个 Node.js 的 C++ 扩展模块，是使用 C++ 分配内存的，并不是 JS 的对象和数据

来个 demo

```
setTimeout(()=>{
    let buffer = Buffer.alloc(1024 * 1024 * 3000);
}, 3000)
```

即使分配了 3G 内存，我们的程序还跑的好好的，我们的 Node.js 程序竟然占了 5G 多的内存，因为这个 external 不受 Node.js 限制，受操作系统对线程分配内存大小的限制 （所以也不能无脑放，即使Buffer 也会爆内存改为 Stream
```

处理大数据量是精髓）。

在Node.js中，`Buffer` 对象的生命周期是与 JavaScript 对象绑定的。当`Buffer`对象的JavaScript 引用被解除时，V8 垃圾回收器会标记这个对象为可回收，但`Buffer`对象的底层内存并不会立即释放。通常，当 C++ 扩展的析构函数被调用时（例如在Node.js的垃圾回收过程中），这部分内存才会被释放。然而，这个过程可能不会与V8的垃圾回收完全同步。

```
Memory Usage (MB): {
  rss: '2392.73 MB',
  heapTotal: '2392.57 MB',
  heapUsed: '2359.93 MB',
  external: '3000.41 MB'
}
Memory Usage (MB): {
  rss: '2392.75 MB',
  heapTotal: '2392.57 MB',
  heapUsed: '2359.94 MB',
  external: '3000.41 MB'
}
Memory Usage (MB): {
  rss: '2392.75 MB',
  heapTotal: '2392.57 MB',
  heapUsed: '2359.94 MB',
  external: '3000.41 MB'
}
```

总结: Node.js 内存占用由 JS堆内存占用（由 V8 决定垃圾回收） + C++扩展模块内存分配占用（由C++扩展决定垃圾回收）

我们写个小 demo，3秒后分配 3G内存，并且每秒观察下状况， 看下 C++ 扩展的垃圾回收

```
setTimeout(() => {
let buffer = Buffer.alloc(1024 * 1024 * 3000);
}, 3000)
```

C++ 扩展模过了一会儿后发现没有 js 对象和这片内存绑定，所以回收了这片内存

```
Memory Usage (MB): {
  rss: '32.32 MB',
  heapTotal: '6.13 MB',
  heapUsed: '4.59 MB',
  external: '3000.41 MB'
}
Memory Usage (MB): {
  rss: '32.34 MB',
  heapTotal: '6.13 MB',
  heapUsed: '4.60 MB',
  external: '3000.41 MB'
}
Memory Usage (MB): {
  rss: '32.40 MB',
  heapTotal: '6.13 MB',
  heapUsed: '4.33 MB',
  external: '0.41 MB' // 回收了
}
```

为什么堆内存要分新生代和老生代啊?

分代垃圾回收策略在现代编程语言的实现中非常普遍！Generational Garbage Collection 这种策略在 Ruby 和 .NET ，Java 中都有类似的应用。 垃圾回收时候会 stop the world! 当然程序性能会受到影响，这种设计是出于对性能优化的考虑。

* 对象生命周期不同

在开发程序中，大部分变量都是临时变量，都是为了完成某个局部的计算任务，这类变量适合使用 Minor GC 也就是新生代GC，新生代内存中的对象主要通过 Scavenge 算法进行垃圾回收。Scavenge 算法，将堆内存一分为二，分别为 From 和 To（典型的空间换时间，但是由于存活时间短，所以不会占用大量内存）。

在进行内存分配时，是在 From 中进行分配的；而在垃圾回收时，会检查 From

中的存活对象，并将这些存活对象复制到 To 中，然后再将非存活对象进行释放，在下一轮回收时，将 To 中的存活对象复制到 From 中，而此时，To 就变成了 From，From 变成了 To。每次垃圾回收，From 和 To 会互换，这个算法在复制时只复制了存活对象，还避免了碎片的产生。

怎么判断这个变量是否存活呢？可以通过可达性分析，假设我们有以下几个对象：

* `globalObject`：全局对象
* `obj1`：一个由 `globalObject` 直接引用的对象
* `obj2`：一个由 `obj1` 引用的对象
* `obj3`：一个孤立的对象，没有被其他对象引用

在可达性分析中：

* `globalObject` 因为是根对象，肯定是可达的。
* `obj1` 由于被 `globalObject` 引用，所以也是可达的。
* `obj2` 因为被 `obj1` 引用，也是可达的。
* 而 `obj3` 由于没有任何与根对象或其他可达对象的引用路径，就被判定为不可达，是可以被回收的对象。

当然引用计数也可以是一个辅助手段，但是如果存在循环引用，就不能很好的识别对象是否真存活。

在老生代内存中，他们通常都是不怎么活动的，但是如果老生代内存也满，那就会触发老生代内存的清理（Major GC）通过 Mark-Sweep。

Mark-Sweep 算法分为标记和清除两个阶段。在标记阶段，V8 引擎会遍历堆中的所有对象，并标记活着的对象；在清除阶段，它只会清除没有被标记的对象。这种算法的优点是清除阶段所占用的时间较少，因为老生代中死亡对象占比较小。然而，它的缺点是只清除而不整理，可能会导致内存空间出现不连续的状态，不利于分配内存给大对象。

这个缺点会导致内存碎片的产生，所以需要配合另一个算法 Mark-Compact。将活着的对象都往一端移动，然后一次性清理掉边界右侧的无效内存空间，从而得到完整连续的可用内存空间。它解决了 Mark-Sweep 算法可能导致的内存碎片问题，但缺点是移动大量的存活对象会消耗比较多的时间。

都看到里了，点个赞吧