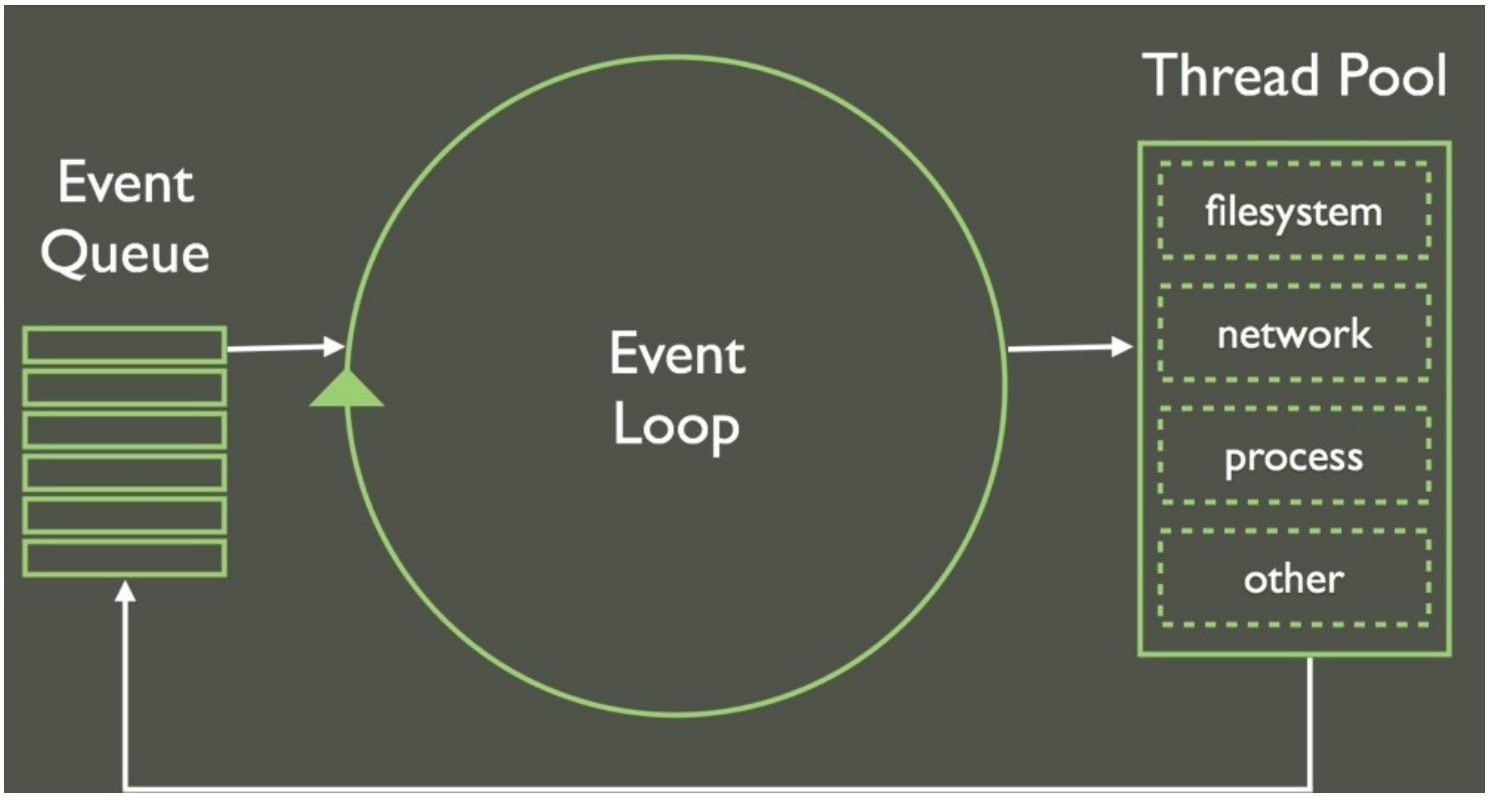


【译】Don't Block the Event Loop

paranoi...
人生天地间 忽如远行客
关注他

> 原文：nodejs.org/en/docs/guid...
> PR：github.com/nodejs/nodej...

你是否应该读这篇指南?

如果你编写的代码并不只是一行命令调用那么简单，那么阅读本篇指南可以帮助你写出高性能、更安全的程序。

此文档是从 Node 服务器开发的角度编写的，但这些概念也同样适用于复杂的 Node 应用程序。文章中如有涉及到不同操作系统的细节，仅以 Linux 系统为代表。

TL; DR

Node.js 通过 Event Loop （事件循环）机制（初始化和回调）的方式运行 JavaScript 代码，并且提供了一个 Worker Pool （线程池）处理诸如 文件 I/O 等高成本的任务。

Node 的伸缩性非常好，某些场景下它甚至比类似 Apache 等更重量级的解决方案表现更优异。Node 可伸缩性的秘诀则在于它仅使用了极少数的线程就可以处理大量客户端连接。 由于 Node 只需占用很少的线程，它得以将更多的系统 CPU 时间和内存投入在处理客户端任务而不是线程的空间和时间消耗上（内存，上下文切换）。

但是 Node 只有少量线程这个特效也是把双刃剑，你必须非常小心的组织你的应用程序以便合理的使用它们。

这里有一个很好的经验法则，能使您的 Node 服务器变快： *在任何时候，当分配到每个客户端的任务是“轻量”的情况下，Node 是非常快的。*

这条法则可以应用于 Event Loop 中的回调任务，以及在工作 Worker Pool 上的任务。

为什么不要阻塞你的 Event Loop 和 Worker Pool?

Node 是用很少量的线程来处理大量客户端请求的。 在 Node 中，有两种类型的线程：一个 Event Loop （事件循环线程，也被称为主循环，主线程，事件线程等）。另外一个是在 Worker Pool （工作线程池）里的 `k` 个工作线程（也被称为线程池）。

如果一个线程执行一个回调函数（Event Loop）或者任务（Worker Pool）需要耗费很长时间，我们称之为“阻塞”。 当一个线程在处理某一个客户端请求时被阻塞了，它就无法处理其它客户端的请求了。 这里给出两个不能阻塞 Event Loop 和 Worker Pool 的理由：

1. 性能：如果你在任意类型的线程上频繁处理繁重的任务，那么你的服务器的 *吞吐量*（请求/秒）将面临严峻考验。
2. 安全性：如果对于特定的输入，你的某种类型的线程可能会被阻塞，那么恶意攻击者可以通过构造类似这样的“恶意输入”，故意让你的线程阻塞，然后使其它客户端请求得不到处理。这就是 DoS attack（拒绝服务攻击）。

对 Node 的快速回顾

Node 使用事件驱动机制：它有一个 Event Loop 负责任务编排，和一个专门处理繁重任务的 Worker Pool。

哪种代码运行在 Event Loop 上?

当 Node 程序运行时，程序首先完成初始化部分，即处理 `require` 加载的模块和注册事件回调。然后，Node 应用程序进入事件循环阶段，通过执行对应回调函数来对客户端请求做出回应。 此回调将同步执行，并且可能在完成之后继续注册新的异步请求。 这些异步请求的回调也会在 Event Loop 中被处理。

Event Loop 中同样也包含很多非阻塞异步请求的回调，如网络 I/O。

总体来说，Event Loop 执行事件的回调函数，并且负责处理类似网络 I/O 的非阻塞异步请求。

哪种代码运行在 Worker Pool?

Node 的 Worker Pool 是通过 libuv (相关文档) 来实现的，它对外提供了一个通用的任务处理 API。

Node 使用 Worker Pool 来处理“高成本”的任务。 这包括一些操作系统并没有提供非阻塞版本的 I/O 操作，以及一些 CPU 密集型的任务。

Node 模块中有如下这些 API 用到了工作线程池：

1. I/O 密集型任务：
    1. DNS： `dns.lookup()` ， `dns.lookupService()` 。
    2. 文件系统：所有的文件系统 API。除 `fs.FSWatcher()` 和那些显式同步调用的 API 之外，都使用 libuv 的线程池。

1. CPU 密集型任务：
    1. Crypto： `crypto.pbkdf2()` ， `crypto.randomBytes()` ， `crypto.randomFill()` 。

在许多 Node 应用程序中，这些 API 是工作线程池任务的唯一来源。此外应用程序和模块可以使用 C++ 插件 向 Worker Pool 提交其它任务。

为了完整性考虑，我们必须要说明，当你在 Event Loop 的一个回调中调用这些 API 时，Event Loop 将不得不为此花费少量的额外开销，因为它必须要进入对应 API 与 C++ 桥接通讯的 Node C++ binding 中，从而向 Worker Pool 提交一个任务。 和整个任务的成本相比，这些开销微不足道。这就是为什么 Event Loop 总是将这些任务转交给 Worker Pool。 当向 Worker Pool 中提交了某个任务，Node 会在 C++ binding 中为对应的 C++ 函数提供一个指针。

Node 怎么决定下一步该运行哪些代码?

抽象来说，Event Loop 和 Worker Pool 分别为等待中的事件回调和等待中的任务维护一个队列。

而事实上，Event Loop 本身并不维护队列，它持有一堆要求操作系统使用诸如 epoll (Linux)，kqueue (OSX)，event ports (Solaris) 或者 IOCP (Windows) 等机制去监听的文件描述符。 这些文件描述符可能代表一个 Socket （网络套接字），一个监听的文件等等。 当操作系统确定某个文件的描述符发生变化，Event Loop 将把它转换成合适的事件，然后触发与该事件对应的回调函数。 你可以通过 这里 学习到更多有关这个过程的知识。

相对而言，Worker Pool 则使用一个真实的队列，里边装的都是要被处理的任务。 一个工作线程从这个队列中取出一个任务，开始处理它。当完成之后这个工作线程向 Event Loop 中发出一个“至少有一个任务完成了”的消息。

对于应用设计而言，这意味着什么?

在类似 Apache 这种“一个客户端连接一个线程”的系统中，每个处理中的客户端都被分配了一个独立的线程。 如果处理某个客户端的线程阻塞了，操作系统会中断它，并给予下一个客户端请求执行的机会。 操作系统必须确保一个只需要少量开销的客户端请求不会被其他需要大量开销的客户端请求影响。

因为 Node 用少量的线程处理许多客户端连接，如果在处理某个客户端的时候阻塞了，在该客户端请求的回调或任务完成之前，其他等待中的任务可能都不会得到执行机会。 *因此，保证每个客户端请求得到公平的执行机会变成了应用程序的责任。* 这意味着，对于任意一个客户端，你不应该在一个回调或任务中做太多的事情。

这既是 Node 服务能够保持良好伸缩性的原因，同时也要求应用程序必须自己确保公平调度。 下一部分将探讨如何确保 Event Loop 和 Worker Pool 的公平调度。

不要阻塞你的 Event Loop

Event Loop 关注着每个新的客户端连接，协调产生一个回应。 所有这些进入的请求和输出的应答都要通过 Event Loop。 这意味着如果你的 Event Loop 在某个地方花费太多的时间，所有当前等待和未来新的客户端请求都得不到处理机会了。

因此，你应该保证永远不要阻塞 Event Loop。 换句话说，每个 JavaScript 回调应该快速完成。这些对于 `await` ， `Promise.then` 也同样适用。

一个能确保做到这一点的方法是分析关于你回调代码的 "计算复杂度"。 如果你的回调函数在任意的参数输入下执行步骤数量都相同，那么你总能保证每个等待中的请求得到一个公平的执行机会。如果回调根据其参数不同所需要的执行步骤数量也不同， 则应深入评估参数复杂度增长的情况下请求的可能执行时间增长情况。

例子 1：固定执行时间的回调。

```
});
```

例子 2：一个 `O(n)` 回调。该回调对于小的输入 `n` 执行很快，但是 `n` 如果很大，会执行得很慢。

```
app.get('/countToN', (req, res) => {
  let n = req.query.n;

  // 先执行 n 次循环，才给其他请求执行机会
  for (let i = 0; i < n; i++) {
    console.log(`Iter {$i}`);
  }

  res.sendStatus(200);
});
```

例子 3：一个 `O(n^2)` 函数回调。 该回调对于小的输入 `n` 同样执行很快， 但是 `n` 如果很大，会比之前 `O(n)` 那个例子慢得多。

```
app.get('/countToN2', (req, res) => {
  let n = req.query.n;

  // 先执行 n^2 次循环，才给其他请求执行机会
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++) {
      console.log(`Iter ${i}.${j}`);
    }
  }

  res.sendStatus(200);
});
```

你应当注意些什么呢?

Node 使用谷歌的 V8 引擎处理 JavaScript，对于大部分操作确实很快。 但有个例外是正则表达式以及 JSON 的处理，下面会讨论。

但是，对于复杂的任务你还应当考虑限定输入范围，直接拒绝会导致太长执行时间的输入。 那样的话，即便你的输入相当长而且复杂，因为你限定了接受范围，你也可以确保回调函数的执行时间在你预估的最差情况范围之内。 然后你可以评估此回调函数的最糟糕执行时间，根据你的业务场景决定此运行时间是否可以接受。

阻塞 Event Loop：REDOS

一个灾难性地阻塞 Event Loop 的常见错误是使用“有漏洞”的 正则表达式。

避免易受攻击的正则表达式

一个正则表达式是一定的规则去尝试匹配一个输入的字符串。 我们通常认为正则表达式的匹配需要扫描一次输入字符串—— `O(n)` 时间，其中 `n` 是输入字符串的长度。 在大部分情况下，一次扫描的确足够。 不幸的是，在某些情况下，正则表达式匹配扫描随着输入字符串呈指数增长——时间是 `O(2^n)` 。 指数级的扫描时间消耗意味着：如果引擎原来需要 `x` 时间来确定匹配，我们的输入仅仅只增加一个字符，它将需要 `2 * x` 的时间。 由于扫描的消耗与所需时间呈线性关系，因此，这种正则匹配将阻塞 Event Loop。

易受攻击的正则表达式 是指执行时间随输入指数级增长的情况， 它使您的应用程序在“恶意输入”的情况下面临 REDOS （正则表达式拒绝服务攻击）的风险。 一个正则表达式是否易受攻击的（例如，正则表达式引擎需要指数级的时间复杂度来执行），实际上是一个很难回答的问题。并且根据您是用 Perl、Python、Ruby、Java、JavaScript 等不同的语言情况也有所不同，但这里有一些在所有语言里都适用的经验法则：

1. 避免嵌套量词，如 `(a+)*` 。Node 的正则表达式引擎可能可以快速处理某些例子，但某些则可能是易受攻击的。
2. 避免带有“或”的重叠情况，如 `(a|a)*` 。同样，Node 只能保证某些场景下可以快速匹配。
3. 避免使用回溯，如 `(a.*) \1` 。没有正则表达式引擎可以保证在线性时间内执行这种匹配。
4. 如果您只需要做简单的字符串匹配，请使用 `indexOf` 或其他等价 API。这些是更好的选择，它们永远不会超过 `O(n)` 的时间。

如果您不确定正则表达式是否易受攻击，请记住：即使对于一个易受攻击的正则表达式和长输入字符串，Node 通常也仍然可以确保结果正确。 而指数爆炸的场景其实是出现在用户输入并不匹配正则的特征，但是 Node 却必须要去尝试执行非常多次的扫描才能最终得出不匹配的结论。

一个 REDOS 例子

下面是一个给服务器带来 REDOS 风险的示例示例：

```
app.get('/redos-me', (req, res) => {
  let filePath = req.query.filePath;

  // REDOS
  if (fileName.match(/(\/.+)+$/)) {
    console.log('valid path');
  }
  else {
    console.log('invalid path');
  }

  res.sendStatus(200);
});
```

这个有漏洞的正则例子是一个(糟糕的！)检查 Linux 上合法路径的例子。 它匹配的字符串是以 "/" 分隔的名称序列，如 "/a/b/c"。 这是非常危险的，因为它违反了规则 1：它有一个双重嵌套的量词。

假设客户端查询的是路径 `///.../\n` （100个“/”后跟一个与正则表达式的“.”不匹配的换行符），则 Event Loop 将持续执行且无法停止，从而阻止 Event Loop。 这类客户端发起的 REDOS 攻击会导致所有其它客户端在此正则表达式匹配完成之前得不到任何执行机会。

因此，您应该警惕使用复杂的正则表达式来验证用户输入的场景。

关于如何抵制 REDOS 的资源

这里提供了你一些工具帮助你检查你的正则表达式是否安全，像：

- safe-regex
- rxxr2

但是上述模块都无法保证能够捕获全部的正则表达式漏洞。

另一个方案是使用一个不同的正则表达式引擎。 你可以使用 node-re2 模块，它使用谷歌的超快正则表达式引擎 RE2。但注意，RE2 对 Node 正则表达式不是 100% 兼容，所以如果你想用

node-re2 模块来处理你的正则表达式的话，请检仔细查你的表达式。 这里尤其值得提醒的是，一些特殊的复杂正则表达式不被 node-re2 支持。

如果你想匹配一些较为“明显”的东西，如网络路径或者是文件路径，请在 正则表达式库 中寻找到对应例子，或者使用一个 npm 的模块，如 ip-regex。

阻塞 Event Loop：Node 的核心模块

一些 Node 的核心模块有同步的高开销的 API 方法，包含：

- crypto 加密
- zlib 压缩
- fs 文件系统
- child_process 子进程

这些 API 是高开销的，因为它们包括了非常巨大的计算（如加密、压缩上），需要 I/O（如文件 I/O），或者两者都有潜在包含（如子进程处理）。这些 API 本意是为脚本提供方便，并非让你在服务器上下文中使用。如果你在 Event Loop 中使用它们，则需要花费比一般的 JavaScript 更长的执行时间，从而可能导致 Event Loop。

对于一个服务器而言，*你不应当使用以下同步的 API 函数*:

- 加密：
  - `crypto.randomBytes` （同步版本）
  - `crypto.randomFillSync`
  - `crypto.pbkdf2Sync`
  - 同时你应当非常小心对大数据输入进行加密和解密的情况。
- 压缩：
  - `zlib.inflateSync`
  - `zlib.deflateSync`
- 文件系统：
  - 不能使用同步的文件系统 API 函数。举个例子，如果你的程序运行于一个分布式文件系统，像 NFS，则访问时间可能会发生很大变化。
- 子进程：
  - `child_process.spawnSync`
  - `child_process.execSync`
  - `child_process.execFileSync`

此列表对于 Node 9 都是有效的。

阻塞 Event Loop：JSON DOS

`JSON.parse` 以及 `JSON.stringify` 是其它潜在高开销的操作。 这些操作的复杂度是 `O(n)` ，对于大型的 `n` 输入，消耗的时间可能惊人的长。

如果您在服务器上操作 JSON 对象（特别是来自客户端的输入），则应谨慎处理在 Event Loop 上消费的对象或字符串的大小。

关于 JSON 阻塞 Event Loop 的示例：我们创建一个大小为 2^21 的 JSON 的对象，然后用 `JSON.stringify` 序列化它；在此字符串上运行 `indexOf` 函数，然后使用 `JSON.parse` 解析它。

```
var obj = { a: 1 };
var niter = 20;

var before, res, took;

for (var i = 0; i < len; i++) {
  obj = { obj1: obj, obj2: obj }; // 每个循环里面将对象 size 加倍
}

before = process.hrtime();
res = JSON.stringify(obj);
took = process.hrtime(n);
console.log('JSON.stringify took ' + took);

before = process.hrtime();
res = str.indexOf('nomatch');
took = process.hrtime(n);
console.log('Pure indexof took ' + took);

before = process.hrtime();
res = JSON.parse(str);
took = process.hrtime(n);
console.log('JSON.parse took ' + took);
```

有一些 npm 的模块提供了异步的 JSON API 函数，参考：

- JSONStream，有流式操作的 API。
- Big-Friendly JSON，有流式 API 和使用下文所概述的任务拆分思想的异步 JSON 标准 API。

不要让复杂的计算阻塞 Event Loop

假设你想在 JavaScript 处理一个复杂的计算，而又不想阻塞 Event Loop。 你有两种选择：partitioning（任务拆分）或 offloading（任务分流）。

任务拆分

你可以把你的复杂计算 *拆分开*，然后让每个计算分别运行在 Event Loop 中，不过你要定期地让其它一些等待的事件得到执行机会。 在 JavaScript 中，用闭包很容易实现保存执行的上下文，请看如下的 2 个例子。

举个例子，假设你想计算 `1` 到 `n` 的平均值。

例子1：不分区算平均数，开销是 `O(n)`

```
for (let i = 0; i < n; i++)
  sum += i;
let avg = sum / n;
console.log('avg: ' + avg);
```

例子2：分区算平均值，每个 `n` 的异步步骤开销为 `O(1)`。

```
function asyncAvg(n, avgCB) {
  // Save ongoing sum in JS closure.
  var sum = 0;
  function help(i, cb) {
```

```
      cb(sum);
      return;
    }

    // "Asynchronous recursion".
    // Schedule next operation asynchronously.
    setImmediate(help.bind(null, i+1, cb));
  }

  // Start the helper, with CB to call avgCB.
  help(1, function(sum){
      var avg = sum/n;
      avgCB(avg);
  });
}

asyncAvg(n, function(avg){
  console.log('avg of 1-n: ' + avg);
});
```

这个原则也可以应用到数组迭代和其它类似场景。

任务分流

如果你需要做更复杂的任务，拆分可能也不是一个好选项。这是因为拆分之后任务仍然在 Event Loop 中执行，并且你无法利用机器的多核硬件能力。 *请记住，Event Loop 只负责协调客户端的请求，而不是独自执行完所有任务。* 对一个复杂的任务，最好把它从 Event Loop 转移到工作线程池上。

如何进行任务分流?

你有两种方式将任务转移到工作线程池执行。

1. 你可以通过开发 C++ 插件 的方式使用内置的 Node Worker Pool。稍早之前的 Node 版本，通过使用 NAN 的方式编译你的 C++ 插件，在新版的 Node 上使用 N-API。 node-webworker-threads 提供了一个仅用 JavaScript 就可以访问 Node 的 Worker Pool 的方式。
2. 您可以创建和管理自己专用于计算的 Worker Pool，而不是使用 Node 自带的负责的 I/O 的 Worker Pool。最直接的方法就是使用 Child Process 或者是 cluster。

你 *不* 应该直接为每个请求都创建一个 子进程 。 因为客户端请求的频率可能远远高于你的服务器能创建和管理子进程的频率，这种情况你的服务器就变成了一个 fork bomb（fork 炸弹）。

转移到 Worker Pool 的缺陷

这种方法的缺点是它增大了 *通信开销* 。 因为 Node 仅允许 Event Loop 去访问应用程序的“命名空间”（保存着 JavaScript 状态）。 在 Worker Pool 中是无法操作 Event Loop 的命名空间中的 JavaScript 对象的。 因此，您必须序列化和反序列化任何要在线程间共享的对象。 然后，Worker Pool 可以对属于自己的这些对象的副本进行操作，并将修改后的对象（或“补丁”） 返回到 Event Loop。

有关序列化问题，请参阅 JSON 文档部分。

一些关于分流的建议

您可能希望区分 CPU 密集型和 I/O 密集型任务，因为它们具有明显不同的特性。

CPU 密集型任务只有在该 Worker 线程被调度到时候才得到执行机会，并且必须将该任务分配到机器的某一个 逻辑核心中。

如果你的机器有 4 个逻辑核心和 5 个工作线程，那这些工作线程中的某一个则无法得到执行机会。 因此，您实质上只是在为该工作线程白白支付开销（内存和调度开销），却无法得到任何返回。

I/O 密集型任务通常包括查询外部服务提供程序（DNS、文件系统等）并等待其响应。 当 I/O 密集型任务的工作线程正在等待其响应时，它没有其它工作可做，并且可以被操作系统重新调度，从而使另一个 Worker 有机会提交它的任务。 因此，*即使关联的线程并没有被占有，I/O 密集型任务也可以持续运行*。 像数据库和文件系统这样的外部服务提供程序已经经过高度优化，可以同时处理许多并发的请求。 例如，文件系统会检查一大组并发等待的写入和读取请求，以合并冲突更新并以最佳顺序读取文件（请参阅 这些幻灯片）。

如果只依赖一个 Worker Pool（例如 Node Worker Pool），则 CPU 密集和 I/O 密集的任务的不同特效可能会损害应用程序的性能。

因此，您可能希望一个维护单独的计算 Worker Pool。

分流：总结

对于简单的任务：比如遍历任意长数组的元素，拆分可能是一个很好的选择。 如果计算更加复杂，则分流是一种更好的方法：通信成本（即在 Event Loop 和 Worker Pool 之间传递序列化对象的开销）被使用多个物理内核的好处抵消。 但是，如果你的服务器严重依赖复杂的计算，则应该重新考虑 Node 是否真的很适合该场景？Node 擅长于 I/O 密集型任务，但对于昂贵的计算，它可能不是最好的选择。

如果采用分流方法，请参阅“不阻塞工作线程池”一节。

不要阻塞你的 Worker Pool

Node 由 `k` 个工作线程组成了 Worker Pool。 如果您使用上面讨论过的任务分流思想，则可能有一个单独的计算 Worker Pool，它也适用于该原则。 在这两种情况下，一般 `k` 比您可能需要同时处理的客户端请求数量要小得多。 这与 Node 的“一个线程处理许多客户端连接”的哲学是一致的，这也是它的可伸缩性秘诀。

正如在上文中讨论的，每个工作线程必须完成当前任务，才能继续执行 Worker Pool 队列中的下一项。

那么，处理客户请求所需的任务成本将会在不同的客户端输入场景下发生很大变化。 有些任务可以快速完成（例如读取小文件或缓存文档，或者生成少量的随机字节），而另一些则需要更长的时间（例如读取较大或未缓存的文件，或生成更多的随机字节）。 您的目标应该是使用 *任务拆分* 来*尽量缩小不同请求任务执行时间的动态变化*，

最小化任务时间的变化

如果工作线程的当前任务比其它任务开销大很多，则他无法处理其它等待中任务。 换言之，*每个相对长的任务会直接减少了 Worker Pool 的可用线程数量，直到它的任务完成*。 这是不可取的。因为从某种程度上说， Worker Pool 中的工作线程越多，Worker Pool 的吞吐量（任务/秒）就越大，因此服务器吞吐量（客户端请求/秒）就越大。 一个具有相对昂贵开销任务的客户端请求将减少 Worker Pool 整体的吞吐量，从而降低服务器的吞吐量。

为避免这种情况，应尽量减少提交给 Worker Pool 的不同任务在执行时间上的变化。 虽然一般而言将 I/O 请求（DB、FS 等）访问的外部系统视为黑盒在程序开发的某种角度是适当的；但您也应该知道这些 I/O 请求的相对成本，并应避免提交您预估可能特别耗时的任务。

动态执行时间示例: 长时间运行的文件系统读取

假设您的服务器必须读取文件来处理某些客户端请求。 在了解 Node 的 文件系统 的 API 之后，您选择使用 `fs.readFile()` 进行简单操作。 但是， `fs.readFile()` 是（当前）未拆分任务的：它提交一个 `fs.read()` 任务来读取整个文件。 如果您为某些用户读取较短的文件，并为其它人读取较长的文件， `fs.readFile()` 可能会在任务长度上引入显著的变化，从而损害 Worker Pool 吞吐量。

对于最坏的情况，假设攻击者可以促使您的服务器读取 *任意* 文件（这是一个 目录遍历漏洞）。 如果您的服务器运行的是 Linux，攻击者可以命名一个非常慢的文件：/dev/random。 假设某种情况下 `/dev/random` 是极其缓慢的；每个工作线程都被要求读取 `/dev/random` ，这样下去将永远无法完成这项任务。 然后，攻击者提交 `k` 个请求，每一个被分配给一个工作线程，则其它需要使用工作线程的客户端请求将得不到执行机会。

动态执行时间示例: 长时间运行的加密操作

假设您的服务器使用 crypto.randomBytes() 来生成密码学上安全的随机字节。 `crypto.randomBytes()` 是不拆分任务的：它创建单个 `randomBytes()` 任务，以生成您请求的字节数。 如果为某些用户创建的字节数较少，为其它请求创建字节数较多；则 `crypto.randomBytes()` 是任务长度变化的另一个来源。

任务拆分

具有可变时间成本的任务可能会损害 Worker Pool 的吞吐量。 为了尽量减少任务时间的长度变化，应尽可能将每个任务 *划分* 为开销接近一致的子任务。 当每个子任务完成时，它应该提交下一个子任务；并且当最终的子任务完成时，它应该通知提交者。

继续使用 `fs.readFile()` 的示例，更好的方案是使用 `fs.read()` （手动拆分）或 `ReadStream` （自动拆分）。

同样的原理也适用于 CPU 密集型任务； `asyncAvg` 示例可能不适用于事件循环，但它非常适合于 Worker Pool。

将任务拆分为子任务时，较短的任务将拆分为少量的子任务，而更长的任务将拆分为更多的子任务。 在较长任务的每个子任务之间，分配给它的工作线程可以调度执行另一个更短的任务拆分出来的子任务，从而提高 Worker Pool 的总体任务吞吐量。

请注意：完成的子任务数对于 Worker Pool 的吞吐量不是一个有用的度量指标。 相反，请关注完成的 *任务* 数。

避免任务拆分

我们需要明确任务拆分的目的是尽量减少任务执行时间的动态变化。 但是如果你可以人工区分较短的任务和较长的任务（例如，对数组求和或排序），则可以手动为每个类型的任务创建一个 Worker Pool。 将较短的任务和更长的任务分别路由到各自的 Worker Pool，也是减少任务时间动态变化的另一种方法。

建议这种方案的原因是，做任务拆分会导致额外的开销（创建工作线程，记录和操作 Worker Pool 任务队列），而避免拆分会为您节省这些额外成本，同时也会避免你在拆分任务的时候犯错误。

这种方案的缺点是：所有这些 Worker Pool 中的工作线程都将产生空间和时间开销，并将相互竞争 CPU 时间片。 请记住：每个 CPU 密集任务只在它被调度到的时候才会得到执行。 因此，您应该再仔细分析后才考虑此方案。

无论您只使用 Node Worker Pool 还是维护单独的 Worker Pool，都应着力优化 Worker Pool 的任务吞吐量。

为此，请使用任务拆分最小化任务执行时间的动态变化范围。

使用 npm 模块的风险

虽然 Node 核心模块为各种需求提供了基础支持，但有时还需要更多的功能。Node 的开发人员从 npm 生态系统 中获益良多，有成百上千个模块可以为你的应用开发提效。

但是，请记住，这些模块中的大多数是由第三方开发人员编写的；它们通常只能保证尽力做到最好。使用 npm 模块的开发人员应该关注如下两件事，尽管后者经常被遗忘。

1. 它是否拥有优秀的 API 设计?
2. 它的 API 可能会阻塞 Event Loop 或 Worker Pool 吗?

许多模块对它们 API 的开销没有任何考虑，这对社区使用者是不利的。

对于简单的 API，您可以估计 API 的成本；如字符串操作的成本并不难预估。 但在许多情况下却很难预估 API 可能的开销。

如果您调用的 API 可能会做一些昂贵的事情，请着重检查成本；或者请求开发人员给出相关文档，或者自己检查源代码（并提交一个 PR 说明开销）。

请记住：即使 API 是异步的，您也可能无法预估它的每个拆分的子任务需要在 Worker Pool 或 Event Loop 上花费多少时间。 例如，假设在上面给出的 `asyncAvg` 示例中，每个对 helper 函数的调用都累加一半的数字而不只是其中的一个。 那么这个函数仍然是异步的，但每个子任务的成本将是 `O(n)` 而不是 `O(1)`， 就使得对于任意的输入 `n` 不再那么安全。

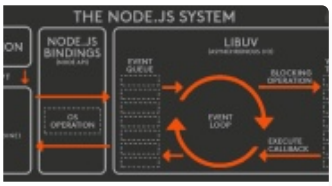
总结

Node 有两种类型的线程：一个 Event Loop 和 `k` 个 Worker。 事件循环负责 JavaScript 回调和非阻塞 I/O，工作线程执行与 C++ 代码对应的、完成异步请求的任务，包括阻塞 I/O 和 CPU 密集型工作。 这两种类型的线程一次都只能处理一个活动。 如果任意一个回调或任务需要很长时间，则运行它的线程将被 *阻塞*。 如果你的应用程序发起阻塞的回调或任务，在好的情况下这可能只会导致吞吐量下降（客户端/秒），而在最坏情况下可能会导致完全拒绝服务。

要编写高吞吐量、防 DoS 攻击的 web 服务，您必须确保不管在良性还是恶意输入的情况下，您的 Event Loop 和您的 Worker Pool 都不会阻塞。

编辑于 2018-09-13

推荐阅读



一次搞懂Event loop

/rTreasure

[Brief Talk] auto, auto&, const auto&以及其它形式...

在C++11中，有两个特性很受欢迎，一个是auto，另外一个就是for-range loop. 用法很简单，但是却非常提高生产力。随便翻开现在的C++11的开源代码，都少不了这个东西，如 https://github.com/...

蓝色 发表于蓝色的味道

详解JS运行机制和Event Loop

1 JS运行机制详解1.1 单线程的JSJavascript是一门单线程语言，在最新的HTML5中提出了Web-Worker，但javascript是单线程这一核心仍未改变。所以一切javascript版的"多线程"都是用单...

whynotgonow

微任务、宏任务与Event-Loop

慕课网 发表于猿论

1 条评论

切换为时间排序

写下你的评论...

isLishude 8 天前

任务拆分这个思路很赞

赞