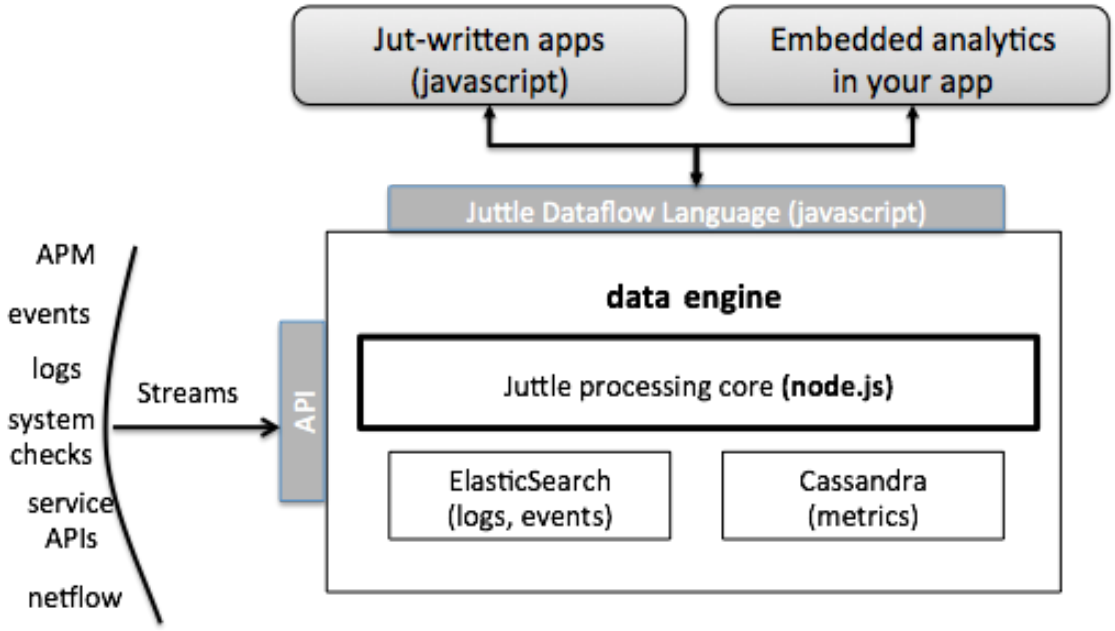
如何更好的利用Node.js的性能极限

作者 **张天雷** 发布于 2015年9月18日 | 1 **讨论**

通过使用非阻塞、事件驱动的I/O操作，Node.js为构建和运行大规模网络应用及服务提供了很好的平台，也受到了广泛的欢迎。其主要特性表现为能够处理庞大的并且高吞吐量的并发连接，从而构建高性能、高扩展性的互联网应用。然而，Node.js单线程的的工作方式及有限的可管理内存使得其计算性能十分有限，限制了某些场景中的应用。近日，Jut开发团队的工程师Dave Galbraith分享了他们所遇到的Node.js的限制以及超越这些限制的方法。接下来，本文就详细分析其所遇到的问题及解决思路。

首先，Jut团队所研发的产品称为操作数据中心（operations data hub）。该产品是一个专门为研发团队所设计的流分析平台，主要用于收集日志以及事件等操作数据，然后根据整体做分析和关联。其核心功能就是要能够同时处理实时数据、历史数据、结构和非结构数据。具体产品架构如下图所示。



从上图可以看出，该产品的核心就是数据引擎，包括底层大数据后端和JPC（Juttle Processing Core）两部分。其中，整体系统需要依赖ElasticSearch和Cassandra等这些大数据后端分系统，进行历史数据的处理和存储以及一般数据的复原和管理；JPC采用了Node.js，完成同等对待历史数据和实时数据、利用日志数据/度量数据/事件数据提出问题以及发送实时数据到浏览器来利用d3进行可视化等。而且，JPC负责运行Juttle程序。当用户点击Juttle程序时，浏览器把程序发送到JPC，将其转换为JavaScript进行执行。Galbraith提出，Jut团队选择JPC中使用Node.js的原因包括采用JavaScript等高级编程语言可以快速完成建模和迭代过程；鉴于程序前端采用JavaScript实现，后端同样采用JavaScript可以方便前后端配合和沟通；Node.js拥有强大的开源社区，使得开发团队可以有效利用社区的力量等三个方面。JPC就利用了社区中103个NPM包，同时也共享了自己开发的7个包。

尽管Node.js拥有着非常好的特性，JPC的开发团队还是遇到了一些Node.js不能直接解决的问题：

1. Node.js的应用程序都是单线程的。这就意味着即使计算机是多核或多处理器的，node.js的应用程序也只能利用其中一个，大大限制了系统性能。
2. 随着堆栈变大，Node.js的垃圾收集器变得非常低效。随着堆栈使用空间超过1GB，垃圾收集的过

程开始变得非常慢，会严重影响程序的性能。

3. 因为以上的问题，Node.js限制了堆栈所能使用的空间为1.5GB。一旦超过该范围，系统就会出错。

为了保证Jut系统的高效性，Jut团队想出了一些解决方案。

首先，针对Node.js单线程引起的性能低下问题，Jut团队采用了尽量避免利用Node.js进行计算的方式。JPC会把Juttle流图切割为一些子图，然后在Jut平台的更深层再进行高效执行。以ElasticSearch为例，在未优化之前，数据请求的流程为：ElasticSearch把相关数据从磁盘中取出->编码为JSON->通过HTTP协议发送给JPC->JPC解码JSON文件，执行预想的计算。然而，ElasticSearch拥有一种聚合（Aggregation）功能，能够跨数据集执行计算。这样，一次大的请求就可以优化为一个ElasticSearch聚合，避免了中间多次JSON转换以及Node.js针对大规模数据进行计算的过程。而且，ElasticSearch和Cassandra都是采用Java编写，可以有效利用多核或多处理器资源，实现高效率并行计算。总之，通过尽量避免在Node.js中进行计算的方式，Jut团队有效提高了系统的性能。

其次，关于堆栈空间问题。每当用户让Node.js服务器向其他服务器发送请求时，用户都会提供一些相应的函数，来对未来返回的数据进行处理。Node.js就会把这些函数放到event loop中，等待数据返回，然后调用相应的函数进行处理。这种类似中断的处理方式，可以大大提高单线程Node.js的效率。然而，一旦event loop中其中一个函数计算的时间过长，系统就会出现问题。以用户向Node.js发送从其他服务器中请求若干行的数据，然后对这些数据进行数学计算为例。如果请求的数据超过了1.5GB堆栈大小的限制，计算过程就会占用Node.js很长一段时间，甚至无法完成。由于Node.js为单线程，在这段时间内，新的请求或者新返回的数据只能放置在event loop的待办列表中。这样，Node.js服务器的反应时间将会大大增加，影响其他请求的正常处理。

为了解决该问题，Jut在任何可能的地方实现了分页（paging）。这就意味着，系统将不会一次读取大量数据，而是将其划分为若干小的请求。在这些请求中间，系统还可以处理器新的请求。当然，多次请求都需要一定的通信代价的。经过Jut团队的摸索，20000个点是比较合适的规模——系统仍然能够在若干毫秒中执行完毕，而且一般的请求也不需要进行大量分割。

针对这些问题，Galbraith分享了一个具体的使用案例。作为Jut的忠实客户，NPM一直伴随着Jut从alpha版本一直走到了现在的beta版本。NPM一个具体的任务就是找到所有包中过去两周下载量最大的前十名，然后在网站中以表格形式的显示。Juttle程序可以利用非常简单的代码完成该任务：

```
read -last :2 weeks: | reduce count() by package | sort count -desc | head 10 | @table
```

但是，Jut第一次跑该程序的时候就遇到了问题。经过调试发现，问题的原因在于JPC优化了read和reduce操作，将其合并为一个ElasticSearch聚合操作。由于聚合操作本身并不支持分页，而NPM的包数要超过数百万个，ElasticSearch就返回了一个超过百万个数组的巨大响应结果，总大小在几百MB。收到该响应后，JPC就试图一次处理完毕，导致内存空间使用超过了1.5GB的限制。垃圾收集器开始不断尝试回收空间。结果，处理时间超过了JPC内置的监控服务认为出现异常的阈值——60s。监控服务直接重启JPC，导致了NPM的任务一直无法完成。

为了解决该问题，Jut团队采用了模仿ElasticSearch针对聚合进行分页的方法。针对返回的包含大量信息的结果，JPC将其切分为可以方便处理的小块，一个个处理。在一些公开库的帮助下，修改后的JavaScript代码如

下：

```
var points = perform_elasticsearch_aggregtion();`
Promise.each(_.range(points.length / 20000), function processChunk(n) {
        return Promise.try(function() {
        process(points.splice(0, 20000));
        }).delay(1);
    });
```

其中Promise.each(param1,param2)负责针对第一个参数param1中的每一个元素调用第二个参数中的函数param2；_.range(num)函数接收一个数字num，返回该数字大小的数组。以包含100万个点为例，上述程序需要调用processChunk()函数50（points.length/20000=1000000/20000=50）次。每次调用负责把20000个点拉出数组，然后调用process()函数进行处理。一旦处理完毕，垃圾收集器就可以对这20000个点占用的空间进行回收。Promise.try()以一个函数作为参数，返回能够控制其参数中函数执行的对象。该对象的.delay(1)方法表示在多次调用中间允许处理器1ms的暂停去处理其他请求。经过这样的修改，程序只花费了大概20s的时间就完成了之前NPM的任务。而且，在此期间，服务器还对其他请求进行了响应。

感谢徐川对本文的审校。