事件循环1

I/O 操作的独特性使得 NodeJS 从一众编程语言中脱颖而出。我们经常可以听到 NodeJs 是一门建立在 Google V8 JavaScript 引擎之上的，无阻塞，基于事件循环的语言。

那么这些都意味着什么呢？事件循环是回答这一切问题的关键所在。在下面的文章中，我会详细介绍什么是事件循环，事件循环是如何影响我们的 web 应用，并且我们应该最大程度地利用好事件循环这个特性。下面我将一系列的文章来介绍该内容。在第一章中，我们将探讨 NodeJs 是如何工作，如何处理 I/O，并且是如何做到支持跨平台的。

**Reactor Pattern ( 反应器设计模式 )**

NodeJs 是一个基于，事件分发器 ( Event Demultiplexer ) 和 任务队列 ( Event Queue )。所有的 I/O 操作 ( 磁盘，网络等 ) 都会最终被抽象成一个 ( 完成/失败 ) 的触发行为（触发器），这种触发器被称为事件。事件的处理基于以下的算法：

1. 事件分发器用于接收 I/O 请求，并且把这些请求分配给合适的硬件
2. 当某个 I/O 操作被处理时（ 例如：磁盘, 网络请求 ），事件分发器会将为这次 I/O 操作注册一个回调函数来执行特定的操作。并把这个回调函数塞入一个队列。这个回调函数就被称为事件，这个装有回调函数的队列就叫做事件队列。
3. 当事件队列里的回调函数可以执行时，系统会按照接收的顺序依次处理事件队列中的回调函数，直到整个队列为空为止。
4. 当事件队列中没有可供处理的事件，或是事件分配器没有额外的待分配的请求时，整个程序就执行完成了，否则就系统将会从第一步开始继续循环执行。

这个用于协调整个事件机制的流程就被称之为事件循环

事件循环是一个单线程，半无限的循环。称它为半无限是因为在没有额外的事件可供处理时，程序就会结束，不会无限执行下去。就开发者所愿，此时程序会结束。

不要把事件循环和 NodeJs 中的事件触发器混为一谈。事件触发器和事件循环是两个大相近庭的概念。在最后一篇文章中，我将介绍事件触发器对事件处理函数，和事件循环的影响

上面的图片高度概括了 NodeJs 是如何工作的，并且向我们展示了 NodeJs 中的一个核心组件——反应器模式 ( Reactor 设计模式 )。不过真实情况要比这更复杂。

事件分配器不仅单纯用于分配各种类型的 I/O 请求（支持跨平台）。同样的，事件分配器也不仅仅是一个用于存放，消费各种事件的队列。I/O 请求也不是事件队列能接收的唯一类型事件。

**事件分配器 ( Event Demultiplexer )**

事件分配器是 Reactor 设计模式中的一个抽象概念。事件分配器在不同的操作系统中有不同的实现，名字也大相近庭。例如，例如 linux 系统中的 epoll，mac os 中的 kqueue，Solaris 中的 event ports 和 windows 系统中的 IOCP 都是事件分配器。NodeJs 借助这些实现来提供底层的，异步，非阻塞的 I/O 操作功能

**文件 I/O 操作的复杂性**

但是令人困惑的是，并非所有的 I/O 操作都可以使用上述系统底层实现来执行。即使是在相同操作系统平台上，支持不同类型的 I/O 操作也很复杂。通常，网络 I/O 可以使用这些epoll、kqueue、event ports和IOCP来进行非阻塞方式，但是文件输入输出要复杂得多。某些系统，比如Linux，不支持完全异步的文件系统访问。在MacOS系统中，文件系统事件(通知/信号)发送也有限制（此处可以了解更多信息）。糅合所有这些情形的差异和复杂性，以便提供完整的异步操作，是一项非常复杂的任务

**DNS 的复杂性**

和文件 I/O 类似的是，NodeJs api 提供的某些 dns 功能也具有相当的复杂性。例如，dns.lookup 这个功能，需要访问文件系统，去读取诸如 nsswitch.conf,resolv.conf 配置文件。那么此时文件系统的复杂性便会延续到 DNS 的实现中来。

**那我们如何来解决这些问题呢 ?**

因此我们引入了线程池来提供 I/O 功能，使用线程池处理的 I/O 操作不能直接访问底层硬件（譬如: epoll 等）。NodeJs 将大部分非阻塞，异步的 I/O 都借助系统提供的 epoll 自己处理掉了，对于那些阻塞的 I/O 或是复杂类型的 I/O 操作，则使用线程池来处理

**总结：**

诚如我们所见，在现实场景中，需要在不同的操作系统上去提供所有不同类型的 I/O ( 文件 io，网络io，dns 等 ) 操作是非常困难的。对于一些类型的 I/O 操作，我们可以使用本机硬件来实现，并保持完全异步。对于特定的 I/O 操作（高复杂度，无法异步，阻塞），我们就将其放入线程池，来实现异步。

PS：开发人员对Node的一个常见误解是，Node在线程池中执行所有的输入输出。

NodeJs 提供了一个抽象层，封装了跨平台，和平台内部的复杂性，并对外提供了统一的 api

**封装的艺术**

下文是来自 libuv 的官方文档:

Libuv是一个跨平台的支持库，最初为node js编写的。其专门针对异步 I/O 所设计。针对不同的 I/O 轮训机制，该库提供的不仅仅是一个简单的抽象：句柄 ( handles ) 和流 ( stream ) 为套接字 ( sockets ) 等进行了高度的抽象。也同时为 NodeJS 提供了跨平台 I/O 和 线程功能。

现在让我们来看一下 libuv 是如何构成的。下面的图标来自 libuv 官方文档，其向我们展示了 libuv 是如何将不同类型的 I/O 请求映射到同一套 api 的。

现在我们知道了，事件分配器不是独立的个体，而是暴露在 NodeJs 上层，一整套由 libuv 提供的 I/O API 的集合。Libuv 同时也为 NodeJs 提供了一整套的事件循环和事件队列机制

下面让我们来看一下事件队列

**事件队列**

事件队列是一个在队列为空前，按照先进先出规则按顺序处理其中事件的一个数据结构。此行为与 Reactor 模式描述得完全不同，那么区别在哪里呢？

1. 在 NodeJs 中包含有多个队列，不同类型的事件在他们自己的队列中排队
2. NodeJs 在处理完一个事件循环周期转入下一个周期之前，事件循环机制会去先处理 intermediate 队列中的事件，直到队列为空。

那么在 NodeJs 中一共有多少个队列，intermediate 队列又是何方神圣呢？

libuv 事件循环机制一共提供了 4种主要队列

1. Expired timers and intervals queue  ( setTimeout 和 setInterval 到期触发事件) 到期的 setTimeOut 和 setInterval 绑定的回调函数
2. IO Events Queue( IO 事件 ) 完成的 IO 事件
3. Immediates Queue 使用 setImmediate 注册的事件
4. Close Handlers Queue 任何关闭事件，例如 socket.on('close', ...)

需要注意的是，上面只是对事件队列类型的简单描述，其中一些种类实际上拥有不同的数据结构（ 定时器 timer，存放于最小堆中 ）

除了上述提到的 4 种队列以外，还有 2 种额外的队列，他们隶属于 intermediate queue。这两个队列不由 libuv 提供，而是由 NodeJs 原生提供，他们分别是：

1. Next Ticks Queue 由 process.nextTick 注册的回调函数
2. Other Microtasks Queue 微任务，譬如：promise 的 resolve 回调

**如何工作?**

正如下图展示的，NodeJs 首先会去检查定时器队列中的到期定时器，启动事件循环，并在该循环周期中遍历每个事件队列，同时维护所有需要处理的事件的引用计数器。若是再处理完 close 队列后，其他队列中没有待处理的事件时，循环就会终止。事件循环中的每个队列的处理可以被看作是事件循环的一个阶段。

上图红色标注出来的两个 intermediate 队列 ( Next Ticks Queue 和 Other Microtasks Queue )，每一个完整的事件循环周期，都会去检查这两个 intermediate 队列，并立即执行其中的事件，直到队列清空后再进入下一个事件循环周期。

例如：事件循环当前正在处理包含了5个事件的 immediates 队列时，加入两个其他事件至 next tick 队列，当事件机制处理完这5个 immediates 事件并转入 close 事件阶段前，会去处理 next tick 队列中的事件。处理完成后才会转入 close 事件阶段。

**Next tick 队列 vs 其他 Microtasks ( 微任务 )**

Next tick 队列的优先级要高于 Microtasks 队列。虽然 Next tick 和 Microtasks 是单次事件循环周期中相邻的两个阶段，且都隶属于 intermediate 队列。但是事件循环机制仍旧会处理完 Next tick 队列中的事件之后，再去处理 Microtasks 队列中的事件

需要注意的是，Next tick 优先级高于 promise resolved 事件，只适用于 v8 引擎的情况，其他引擎需要更具其各自的实现来确定。若是您使用了类似 bluebird 或是 q 的库，则上述优先级的顺序则不再适用，这些库对 promise 有各自的有别于 v8 的实现方式。

在处理 promise resolve 回调的时候，q 和 bluebird 也有各自不同的实现方式，在后面的文章中会介绍。

Intermediate 队列的引入，带来了一个新的问题，IO 饿死 ( IO starvation )。假设我们不停地使用 process.nextTick 函数往 Intermediate 队列 ( 此处具体为 Next Tick 队列 )中增加事件。那么事件循环机制将被迫无限期地去处理 Next Tick 队列中的事件，而造成整个事件循环周期的停顿。这种由于 Intermediate 队列长期无法被清空而造成事件循环无法继续前进的故障就叫做 IO 饿死。

为了避免这个问题，在 NodeJs 0.12 版本之前，我们可以使用 process.maxTickDepth 参数，设置 next tick 队列的最大深度。后续 NodeJs 版本中，由于各种原因移除了这个方法。

全文最后，相信大家应该对事件循环的定义，实现，和其处理异步 I/O 的方式有了一个大致的了解。下面图例展示了 Libuv 在整个 nodejs 架构中的位置。

热衷希望大家能对本文提出意见，在下面的文章中，我还将介绍

1. Timers, Immediates 和 process.nextTick
2. Resolved Promise 回调 和 process.nextTick
3. 如何处理 I/O
4. 事件循环的最佳实践