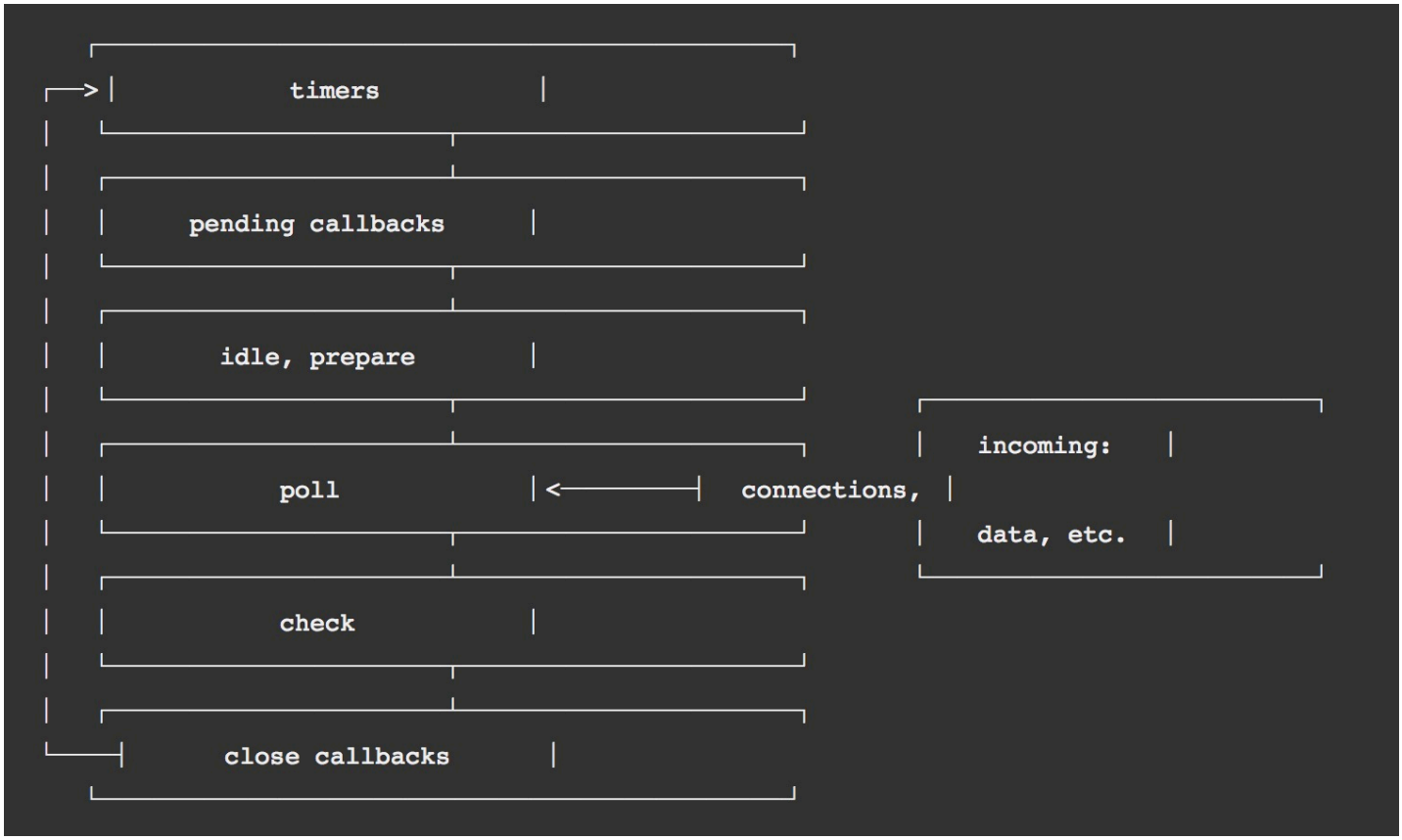
# The Node.js Event Loop, Timers and process.nextTick()

## 什么是Event loop

Event Loop使Node.js可以做非阻塞 I/O操作，尽管实际上JavaScript是单线程的 -- 尽可能的通过下发操作给操作系统内核。 大部分现代内核是多线程支持的，它们可以在后台处理多个操作。当这些操作中有一个完成时，内核通知Node.js执行已经被添加到了 poll 队列中对应的回调函数。我们会在之后更多的讨论这方面的细节。

## Event Loop Explained

当Node.js开始执行，它初始化了Event loop，执行输入提供的脚本里可能使用异步api、定时器或者调用process.nextTick()，这时就开始处理Event loop。 下面的图表简单展示了event loop执行顺序概况。




*注意:每一个块都被当做一个event loop阶段*

每一个阶段都有一个FIFO队列来执行回调。通常,当event loop进度到一个给定的阶段时，每个阶段都有特定处理的事情，它将执行特定于该阶段的任何操作,然后再该阶段的队列中执行回调，直到队列为空或到执行最大回调数。当队列耗尽或者回调数到达限制，event loop将移至下一阶段，以此类推。 由于任何这些操作都可以安排更多操作，并且在loop阶段处理的新事件由内核排队，轮询事件可以在处理轮询事件时加入队列。因此，长时间执行的回调能使poll阶段执行的比timer的阈值长的多。在timers和poll段落有更多的描述。

## 阶段概览

- timers: 这个阶段执行setTimeout和setInterval预先设置的回调函数
- pending callbacks: 执行延迟到下一次循环迭代的 I/O 回调函数
- idle,prepare: 只在内部使用
- poll：取到新的I/O事件，执行I/O先关的回调（除了close callbacks和由定时器或setImmediate调度以外的几乎所有回调），在这里将会在适当的时候阻塞。
- check：setImmediate()回调函数会在这里执行。
- close callbacks: 一些close callbacks，例如：socket.on('close',...) . 在每次event loop执行间隔，Node.js检查是否在等待的异步I/O或定时器，如果没有就关闭循环。

## 阶段详情

### 定时器

定时器有指定的阈值，在阈值之后可以执行提供的回调，而不是你想要执行它的确切时间。定时器回调在指定的时间过去后会尽早的被安排执行；但是，操作系统调度或其他回调的允许可能会延迟它们。

*注意：严格来说，poll阶段控制什么时候执行timers。*

例如，假设您计划在100毫秒阈值后执行超时，那么您的脚本将异步读取一个耗时95毫秒的文件：

```
const fs = require('fs');

function someAsyncOperation(callback) {
  // Assume this takes 95ms to complete
  fs.readFile('/path/to/file', callback);
}

const timeoutScheduled = Date.now();

setTimeout(() => {
  const delay = Date.now() - timeoutScheduled;

  console.log(`${delay}ms have passed since I was scheduled`);
}, 100);


// do someAsyncOperation which takes 95 ms to complete
someAsyncOperation(() => {
  const startCallback = Date.now();

  // do something that will take 10ms...
  while (Date.now() - startCallback < 10) {
    // do nothing
  }
});
```

当event loop进入到poll阶段，存在一个空的队列（fs.readFile()还没有完成），所以它会逗留几毫秒直到下一个最近的定时器到达阈值。在这里等待了95ms，fs.readFile()完成文件读取,接着会消耗10ms把对应的回调添加到poll队列并执行。当回调执行完成，poll队列中没有其他回调函数，所以event loop会查看如果有达到时间最近定时器的阈值，就回到计时器阶段以执行计时器的回调。在此示例中，您将看到正在调度的计时器与正在执行的回调之间的总延迟将为105毫秒。

## pending callbacks

此阶段执行某些系统操作（例如TCP错误类型）的回调。例如，如果TCP套接字在尝试连接时收到ECONNREFUSED，在某些*nix系统希望等待报告错误。这将排队等待在挂起的pending回调阶段执行。

## poll

poll阶段有两个主要功能：

- 计算它应该阻止和轮询I / O的时间，然后
- 处理轮询队列中的事件 当event loop进入到poll阶段并且没有定时器时，将执行以下两个分支之一：
- 如果poll队列不为空，则事件循环将遍历其同步执行它们的回调队列，直到队列已用尽或者达到系统相

关的硬限制。

- 如果poll队列为空，将执行以下两个分支之一：
  - 如果setImmediate（）已调度脚本，则事件循环将结束轮询阶段并继续执行检查阶段以执行这些调度脚本。
  - 如果setImmediate（）尚未调度脚本，则事件循环将等待将回调添加到队列，然后立即执行它们。

轮询队列为空后，事件循环将检查已达到时间阈值的计时器。如果一个或多个计时器准备就绪，事件循环将回绕到计时器阶段以执行那些计时器的回调。

## check

此阶段允许在轮询阶段完成后立即执行回调。如果轮询阶段变为空闲并且脚本已使用setImmediate（）排队，则事件循环可以继续到检查阶段而不是等待。

setImmediate（）实际上是一个特殊的计时器，它在event loop的一个单独阶段运行。它使用libuv API来调度在轮询阶段完成后执行的回调。

通常，在执行代码时，事件循环最终会到达轮询阶段，它将等待传入连接，请求等。但是，如果已使用setImmediate（）调度回调并且轮询阶段变为空闲，则将结束并继续进入检查阶段，而不是一直等待poll events。

## close callbacks

如果一个socket或者handle突然关闭（例如：socket.destroy()），在这个阶段会触发 ‘close’事件，否则它将通过process.nextTick()触发。

## setImmediate() VS setTimeout()

setImmediate和setTimeout（）类似，但在不同的调用场景有不同的运行方式。

- setImmediate()设计用于在poll阶段完成后执行一次脚本调用。
- setTimeout()调度在经过最小阈值（毫秒）后运行的脚本。

执行定时器的顺序将根据调用它们的上下文而有所不同。如果从主模块中调用两者，则时机将受到进程性能的限制（可能受到计算机上运行的其他应用程序的影响）。

例如，如果我们运行不在I / O周期内的以下脚本（即主模块），则执行两个定时器的顺序是不确定的，因为它受进程性能的约束：

```
// timeoutvsimmediate.js
setTimeout(() => {
  console.log('timeout');
}, 0);



setImmediate(() => {
  console.log('immediate');
});
```

```
$ node timeoutvsimmediate.js
timeout
immediate



$ node timeoutvsimmediate.js
immediate
timeout
```

但是，如果你把这两个函数放入一个 I/O 循环内调用，setImmediate 总是被优先调用：

```
// timeoutvsimmediate.js
const fs = require('fs');



fs.readFile(_filename, () => {
  setTimeout(() => {
    console.log('timeout');
  }, 0);
  setImmediate(() => {
    console.log('immediate');
  });
});
```

```
$ node timeoutvs_immediate.js
immediate
timeout



$ node timeoutvsimmediate.js
immediate
timeout
```

使用 setImmediate() 超过 setTimeout() 的主要优点是 setImmediate() 在任何计时器（如果在 I/O 周期内）都将始终执行，而不依赖于存在多少个计时器。

## process.nextTick()

### 理解process.nextTick()

您可能已经注意到 process.nextTick() 在关系图中没有显示，即使它是异步 API 的一部分。这是因为 process.nextTick() 在技术上不是事件循环的一部分。相反，无论事件循环的当前阶段如何，都将在当前操作完成后处理 nextTickQueue。 回顾我们的关系图，在给定的阶段中任何时候您调用 process.nextTick()时，所有传递到 process.nextTick() 的回调将在事件循环继续之前得到解决。这可能会造成一些糟糕的情况, 因为它允许您通过进行递归 process.nextTick() 来“饿死”您的 I/O 调用，阻止事件循环到达 轮询 阶段。

### 为什么会允许这样?

为什么这样的事情会包含在 Node.js 中？它的一部分是一个设计理念，其中 API 应该始终是异步的，即使它不必是。以此代码段为例：

```
function apiCall(arg, callback) {
  if (typeof arg !== 'string')
    return process.nextTick(callback,
                            new TypeError('argument should be string'));
}
```

这个代码片段进行参数检查，如果不正确，它会将错误传递给回调。最近更新的API允许传递参数到 process.nextTick(),允许将回调后的任何参数作为回调的参数，因此您不必嵌套函数。 我们正在做的是将错误传回给用户，但它会在其余的用户代码执行之后进行。通过使用process.nextTick()我们确保apiCall()始终在用户代码执行之后和允许时间循环之前运行其回调。为了实现这一点，允许JS调用堆栈展开然后立即执行提供的回调，这允许一个人对process.nextTick（）进行递归调用而不会达到RangeError：超出v8的最大调用堆栈大小。 这种理念可能会导致一些潜在的问题。以此片段为例：

```
let bar;



// this has an asynchronous signature, but calls callback synchronously
function someAsyncApiCall(callback) { callback(); }



// the callback is called before someAsyncApiCall completes.
someAsyncApiCall(() => {
  // since someAsyncApiCall has completed, bar hasn't been assigned any value
  console.log('bar', bar); // undefined
});



bar = 1;
```

用户将someAsyncApiCall（）定义为具有异步签名，但它实际上是同步操作的。调用它时，在事件循环的同一阶段调用提供给someAsyncApiCall（）的回调，因为someAsyncApiCall（）实际上不会异步执行任何操作。因此，回调尝试引用bar，即使它在范围内可能没有该变量，因为该脚本无法运行完成。

通过将回调放在process.nextTick（）中，脚本仍然能够运行完成，允许在调用回调之前初始化所有变量，函数等。它还具有不允许事件循环继续的优点。在允许事件循环继续之前，向用户警告错误可能是有用的。以下是使用process.nextTick（）的前一个示例：

```
let bar;



// this has an asynchronous signature, but calls callback synchronously
function someAsyncApiCall(callback) { callback(); }



// the callback is called before someAsyncApiCall completes.
someAsyncApiCall(() => {
  // since someAsyncApiCall has completed, bar hasn't been assigned any value
  console.log('bar', bar); // undefined
});



bar = 1;
```

用户将 someAsyncApiCall() 定义为具有异步签名，但实际上它是同步运行的。当调用它时，提供给 someAsyncApiCall() 的回调在同一阶段调用事件循环，因为 someAsyncApiCall() 实际上并没有异步执行任何事情。因此，回调尝试引用 bar，在它作用域范围内可能还没有该变量，因为脚本无法运行到完成。

通过将回调置于 process.nextTick() 中，脚本仍具有运行完成的能力，允许在调用回调之前初始化所有变量、函数等。它还具有不允许事件循环继续的优点。在允许事件循环继续之前，对用户发出错误警报可能很有用。下面是使用 process.nextTick() 的上一个示例：

```
let bar;



function someAsyncApiCall(callback) {
  process.nextTick(callback);
}



someAsyncApiCall(() => {
  console.log('bar', bar); // 1
});



bar = 1;
```

这有是另外一个真实的例子：

```
const server = net.createServer(() => {}).listen(8080);



server.on('listening', () => {});
```

仅当传递端口时，端口立即绑定。因此，可以立即调用'listen'回调。问题是那时候不会设置.on（'listen'）回调。 为了解决这个问题，'listen'事件在nextTick（）中排队，以允许脚本运行完成。这允许用户设置他们想要的任何事件处理程序。

## process.nextTick() vs setImmediate()

就用户而言，我们有两个类似的呼叫，但它们的名称令人困惑。

- process.nextTick() 在同一阶段立即触发。
- 在下一次迭代或事件循环的“tick”时触发

本质上，两者的名字需要交换。process.nextTick() 比 setImmediate()触发的更“立即”，但过去的设计不太可能改变。进行这个切换会破坏npm上的大部分包。每天都会添加更多新模块，这意味着我们每天都在等待，更多的潜在破损发生。虽然它们令人困惑，但名称本身不会改变。

我们建议开发人员在所有情况下都使用setImmediate（），因为它更容易推理（并且它导致代码与更广泛的环境兼容，如浏览器JS。）

## 为什么使用process.nextTick()?

有两个主要原因： + 允许用户处理错误，清除任何不需要的资源，或者在事件循环继续之前再次尝试请求。 + 有时需要允许回调在调用堆栈展开之后但在事件循环继续之前运行。

一个例子是匹配用户的期望。简单的例子：

```
const server = net.createServer();
server.on('connection', (conn) => { });




server.listen(8080);
server.on('listening', () => { });
```

假设listen（）在事件循环开始时运行，但是监听回调放在setImmediate（）中。除非传递hostname，否则将立即绑定到端口。要是事件循环继续，它必须达到poll阶段，这意味着存在一个非零概率已经先收到链接，会在listening 事件前触发connection 事件。 另一个例子是运行一个函数构造函数，比如继承自EventEmitter，它想在构造函数中调用一个事件：

```
const EventEmitter = require('events');
const util = require('util');




function MyEmitter() {
  EventEmitter.call(this);
  this.emit('event');
}
util.inherits(MyEmitter, EventEmitter);




const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
```

您无法立即从构造函数中发出事件，因为脚本将不会处理到用户为该事件分配回调的位置.因此，在构造函数本身中，您可以使用process.nextTick（）设置回调以在构造函数完成后发出事件，从而提供预期的结果：

```
const EventEmitter = require('events');
const util = require('util');



function MyEmitter() {
  EventEmitter.call(this);



// use nextTick to emit the event once a handler is assigned
  process.nextTick(() => {
    this.emit('event');
  });
}
util.inherits(MyEmitter, EventEmitter);



const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
```