NodeJs原理详解

#2021#

Buffer

在讲Nodejs的Buffer之前, 先来看几个基本概念

背景知识

1. ArrayBuffer

ArrayBuffer 对象用来表示通用的、固定长度的原始二进制数据缓冲区。
ArrayBuffer 不能直接操作，而是要通过类型数组对象 或 DataView 对象来操作，它们会将缓冲区中的数据表示为特定的格式，并通过这些格式来读写缓冲区的内容。

你可以把它理解为一块内存, 具体存什么需要其他的声明。

```
new ArrayBuffer(length)

// 参数：length 表示要创建的 ArrayBuffer 的大小，单位为字节。
// 返回值：一个指定大小的 ArrayBuffer 对象，其内容被初始化为 0。
// 异常：如果 length 大于 Number.MAX_SAFE_INTEGER (>= 2 ** 53) 或为负数，则抛出一个
RangeError 异常。
```

Ex. 比如这段代码, 可以执行一下看看输出什么

```
var buffer = new ArrayBuffer(8);
var view = new Int16Array(buffer);

console.log(buffer);
console.log(view);
```

2. Unit8Array

Uint8Array 数组类型表示一个 8 位无符号整型数组，创建时内容被初始化为 0。
创建完后，可以以对象的方式或使用数组下标索引的方式引用数组中的元素。

```
// 来自长度
var uint8 = new Uint8Array(2);
uint8[0] = 42;
console.log(uint8[0]); // 42
console.log(uint8.length); // 2
console.log(uint8.BYTES_PER_ELEMENT); // 1

// 来自数组
var arr = new Uint8Array([21,31]);
console.log(arr[1]); // 31

// 来自另一个 TypedArray
var x = new Uint8Array([21, 31]);
var y = new Uint8Array(x);
console.log(y[0]); // 21
```

3. ArrayBuffer 和 TypedArray的关系

TypedArray: Unit8Array, Int32Array这些都是TypedArray, 那些 Uint32Array 也好，Int16Array 也好，都是给 ArrayBuffer 提供了一个 “View”，MDN上的原话叫做 “Multiple views on the same data”，对它们进行下标读写，最终都会反应到它所建立在的 ArrayBuffer 之上。

ArrayBuffer 本身只是一个 0 和 1 存放在一行里面的一个集合，ArrayBuffer 不知道第一个和第二个元素在数组中该如何分配。 // 看array-buffer.png

为了能提供上下文，我们需要将其封装在一个叫做 View 的东西里面。这些在数据上的 View 可以被添加进确定类型的数组，而且我们有很多种确定类型的数据可以使用。

3.1 例如，你可以使用一个 Int8 的确定类型数组来分离存放 8 位二进制字节。 // 看图 int8-array.png
3.2 例如，你可以使用一个无符号的 Int16 数组来分离存放 16 位二进制字节。 // 看图 unit16-array.png

4. 总结

总之, ArrayBuffer 基本上扮演了一个原生内存的角色.

NodeJs Buffer

Buffer 类以一种更优化、更适合 Node.js 用例的方式实现了 Uint8Array API.

Buffer 类的实例类似于整数数组，但 Buffer 的大小是固定的、且在 V8 堆外分配物理内存。Buffer 的大小在被创建时确定，且无法调整。

基本使用

```
// 创建一个长度为 10、且用 0 填充的 Buffer。
const buf1 = Buffer.alloc(10);

// 创建一个长度为 10、且用 0x1 填充的 Buffer。
const buf2 = Buffer.alloc(10, 1);

// 创建一个长度为 10、且未初始化的 Buffer。
// 这个方法比调用 Buffer.alloc() 更快，
// 但返回的 Buffer 实例可能包含旧数据，
// 因此需要使用 fill() 或 write() 重写。
const buf3 = Buffer.allocUnsafe(10);

// 创建一个包含 [0x1, 0x2, 0x3] 的 Buffer。
const buf4 = Buffer.from([1, 2, 3]);

// 创建一个包含 UTF-8 字节  的 Buffer。
```

```
const buf5 = Buffer.from('tést');
```

tips

1. 当调用 Buffer.allocUnsafe() 时，被分配的内存段是未初始化的（没有用 0 填充）。

虽然这样的设计使得内存的分配非常快，但已分配的内存段可能包含潜在的敏感旧数据。 使用通过 Buffer.allocUnsafe() 创建的没有被完全重写内存的 Buffer ，在 Buffer内存可读的情况下，可能泄露它的旧数据。
虽然使用 Buffer.allocUnsafe() 有明显的性能优势，但必须额外小心，以避免给应用程序引入安全漏洞。

Buffer 与字符编码

Buffer 实例一般用于表示编码字符的序列，比如 UTF-8 、 UCS2 、 Base64 、或十六进制编码的数据。 通过使用显式的字符编码，就可以在 Buffer 实例与普通的 JavaScript 字符串之间进行相互转换。

```
const buf = Buffer.from('hello world', 'ascii');

console.log(buf)

// 输出 68656c6c6f20776f726c64
console.log(buf.toString('hex'));

// 输出 aGVsbG8gd29ybGQ=
console.log(buf.toString('base64'));
```

Node.js 目前支持的字符编码包括：

1. ‘ascii’ - 仅支持 7 位 ASCII 数据。如果设置去掉高位的话，这种编码是非常快的。
2. ‘utf8’ - 多字节编码的 Unicode 字符。许多网页和其他文档格式都使用 UTF-8 。
3. ‘utf16le’ - 2 或 4 个字节，小字节序编码的 Unicode 字符。支持代理对（U+10000 至 U+10FFFF）。
4. ‘ucs2’ - ‘utf16le’ 的别名。

5. ‘base64’ - Base64 编码。当从字符串创建 Buffer 时，按照 RFC4648 第 5 章的规定，这种编码也将正确地接受 “URL 与文件名安全字母表”。
6. ‘latin1’ - 一种把 Buffer 编码成一字节编码的字符串的方式（由 IANA 定义在 RFC1345 第 63 页，用作 Latin-1 补充块与 C0/C1 控制码）。
7. ‘binary’ - ‘latin1’ 的别名。
8. ‘Hex’ - 将每个字节编码为两个十六进制字符。

Buffer 内存管理

在介绍 Buffer 内存管理之前，我们要先来介绍一下 Buffer 内部的 8K 内存池。

8K 内存池

1. 在 Node.js 应用程序启动时，为了方便地、高效地使用 Buffer，会创建一个大小为 8K 的内存池。

```
Buffer.poolSize = 8 * 1024; // 8K
var poolSize, poolOffset, allocPool;

// 创建内存池
function createPool() {
  poolSize = Buffer.poolSize;
  allocPool = createUnsafeArrayBuffer(poolSize);
  poolOffset = 0;
}

createPool();
```

2. 在 createPool() 函数中，通过调用 createUnsafeArrayBuffer() 函数来创建 poolSize（即8K）的 ArrayBuffer 对象。createUnsafeArrayBuffer() 函数的实现如下：

```
function createUnsafeArrayBuffer(size) {
  zeroFill[0] = 0;
  try {
```

```
    return new ArrayBuffer(size); // 创建指定size大小的ArrayBuffer对象，其内容被初始化为0。
  } finally {
    zeroFill[0] = 1;
  }
}
```

这里你只需知道 Node.js 应用程序启动时，内部有个 8K 的内存池即可。

3. 那接下来我们要介绍哪个对象呢？在前面的预备知识部分，我们简单介绍了 ArrayBuffer 和 Unit8Array 相关的基础知识，而 ArrayBuffer 的应用在 8K 的内存池部分的已经介绍过了。那接下来当然要轮到 Unit8Array 了，我们再来回顾一下它的语法：

```
Uint8Array(length);
Uint8Array(typedArray);
Uint8Array(object);
Uint8Array(buffer [, byteOffset [, length]]);
```

其实除了 Buffer 类外，还有一个 FastBuffer 类，该类的声明如下：

```
class FastBuffer extends Uint8Array {
  constructor(arg1, arg2, arg3) {
    super(arg1, arg2, arg3);
  }
}
```

是不是知道 Uint8Array 用在哪里了，在 FastBuffer 类的构造函数中，通过调用 Uint8Array(buffer [, byteOffset [, length]]) 来创建 Uint8Array 对象。

4. 那么现在问题来了，FastBuffer 有什么用？它和 Buffer 类有什么关系？带着这两个问题，我们先来一起分析下面的简单示例：

```
```

```
const buf = Buffer.from('semlinker');
console.log(buf); // <Buffer 73 65 6d 6c 69 6e 6b 65 72>
```

为什么输出了一串数字, 我们创建的字符串呢? 来看一下源码

```
/**
 * Functionally equivalent to Buffer(arg, encoding) but throws a TypeError
 * if value is a number.
 * Buffer.from(str[, encoding])
 * Buffer.from(array)
 * Buffer.from(buffer)
 * Buffer.from(arrayBuffer[, byteOffset[, length]])
 **/
Buffer.from = function from(value, encodingOrOffset, length) {
  if (typeof value === "string") return fromString(value, encodingOrOffset);
  // 处理其它数据类型，省略异常处理等其它代码
  if (isAnyArrayBuffer(value))
    return fromArrayBuffer(value, encodingOrOffset, length);
  var b = fromObject(value);
};
```

可以看出 Buffer.from() 工厂函数，支持基于多种数据类型（string、array、buffer 等）创建 Buffer 对象。对于字符串类型的数据，内部调用 fromString(value, encodingOrOffset) 方法来创建 Buffer 对象。

是时候来会一会 fromString() 方法了，它内部实现如下：

```
function fromString(string, encoding) {
  var length;
  if (typeof encoding !== "string" || encoding.length === 0) {
    if (string.length === 0) return new FastBuffer();
    // 若未设置编码，则默认使用utf8编码。
    encoding = "utf8";
```

```
    // 使用 buffer binding 提供的方法计算string的长度
    length = byteLengthUtf8(string);
  } else {
    // 基于指定的 encoding 计算string的长度
    length = byteLength(string, encoding, true);
    if (length === -1)
      throw new errors.TypeError("ERR_UNKNOWN_ENCODING", encoding);
    if (string.length === 0) return new FastBuffer();
  }

  // 当字符串所需字节数大于4KB，则直接进行内存分配
  if (length >= Buffer.poolSize >>> 1)
    // 使用 buffer binding 提供的方法，创建buffer对象
    return createFromString(string, encoding);

  // 当剩余的空间小于所需的字节长度，则先重新申请8K内存
  if (length > poolSize - poolOffset)
    // allocPool = createUnsafeArrayBuffer(8K); poolOffset = 0;
    createPool();
  // 创建 FastBuffer 对象，并写入数据。
  var b = new FastBuffer(allocPool, poolOffset, length);
  const actual = b.write(string, encoding);
  if (actual !== length) {
    // byteLength() may overestimate. That's a rare case, though.
    b = new FastBuffer(allocPool, poolOffset, actual);
  }
  // 更新pool的偏移
  poolOffset += actual;
  alignPool();
  return b;
```

所以我们得到这样的结论

1. 当未设置编码的时候，默认使用 utf8 编码；
2. 当字符串所需字节数大于4KB，则直接进行内存分配；
3. 当字符串所需字节数小于4KB，但超过预分配的 8K 内存池的剩余空间，则重新申请 8K 的内存池；
4. 调用 new FastBuffer(allocPool, poolOffset, length) 创建 FastBuffer 对象，进行数据存储，数

据成功保存后，会进行长度校验、更新 poolOffset 偏移量和字节对齐等操作。

Stream

在构建较复杂的系统时，通常将其拆解为功能独立的若干部分。这些部分的接口遵循一定的规范，通过某种方式相连，以共同完成较复杂的任务。譬如，shell通过管道|连接各部分，其输入输出的规范是文本流。

在Node.js中，内置的Stream模块也实现了类似功能，各部分通过.pipe()连接。

Stream提供了以下四种类型的流：

```
var Stream = require('stream')

var Readable = Stream.Readable
var Writable = Stream.Writable
var Duplex = Stream.Duplex
var Transform = Stream.Transform
```

使用Stream可实现数据的流式处理，如：

```
var fs = require('fs')
// `fs.createReadStream`创建一个`Readable`对象以读取`bigFile`的内容，并输出到标准输出
// 如果使用`fs.readFile`则可能由于文件过大而失败
fs.createReadStream(bigFile).pipe(process.stdout)
```

Readable

创建可读流。

实例：流式消耗迭代器中的数据。

代码实现。

实际使用时，new ToReadable(iterator)会返回一个可读流，下游可以流式的消耗迭代器中的数据。

执行上述代码，将会有100亿个随机数源源不断地写进标准输出流。

创建可读流时，需要继承Readable，并实现_read方法。

- _read方法是从底层系统读取具体数据的逻辑，即生产数据的逻辑。
- 在_read方法中，通过调用push(data)将数据放入可读流中供下游消耗。
- 在_read方法中，可以同步调用push(data)，也可以异步调用。
- 当全部数据都生产出来后，必须调用push(null)来结束可读流。
- 流一旦结束，便不能再调用push(data)添加数据。

可以通过监听data事件的方式消耗可读流。

- 在首次监听其data事件后，readable便会持续不断地调用_read()，通过触发data事件将数据输出。
- 第一次data事件会在下一个tick中触发，所以，可以安全地将数据输出前的逻辑放在事件监听后（同一个tick中）。
- 当数据全部被消耗时，会触发end事件。

上面的例子中，process.stdout代表标准输出流，实际是一个可写流。下小节中介绍可写流的用法。

Writable

创建可写流。

前面通过继承的方式去创建一类可读流，这种方法也适用于创建一类可写流，只是需要实现的是*write(data, enc, next)*方法，而不是read()方法。

有些简单的情况下不需要创建一类流，而只是一个流对象，可以用如下方式去做：

代码实现。

- 上游通过调用writable.write(data)将数据写入可写流中。write()方法会调用_write()将data写入

底层。

- 在_write中，当数据成功写入底层后，必须调用next(err)告诉流开始处理下一个数据。
- next的调用既可以是同步的，也可以是异步的。
- 上游必须调用writable.end(data)来结束可写流，data是可选的。此后，不能再调用write新增数据。
- 在end方法调用后，当所有底层的写操作均完成时，会触发finish事件。

Duplex

创建可读可写流。

Duplex实际上就是继承了Readable和Writable的一类流。 所以，一个Duplex对象既可当成可读流来使用（需要实现*read*方法），也可当成可写流来使用（需要实现write方法）。

代码实现。

上面的代码中实现了*read*方法，所以可以监听*data*事件来消耗*Duplex*产生的数据。 同时，又实现了write方法，可作为下游去消耗数据。

因为它既可读又可写，所以称它有两端：可写端和可读端。 可写端的接口与Writable一致，作为下游来使用；可读端的接口与Readable一致，作为上游来使用。

Transform

在上面的例子中，可读流中的数据（0, 1）与可写流中的数据（'a', 'b'）是隔离开的，但在Transform中可写端写入的数据经变换后会自动添加到可读端。 Tranform继承自Duplex，并已经实现了*read*和write方法，同时要求用户实现一个_transform方法。

代码实现。

数据类型

前面几节的例子中，经常看到调用data.toString()。这个toString()的调用是必需的吗?

在shell中，用管道（|）连接上下游。上游输出的是文本流（标准输出流），下游输入的也是文本流（标准输入流）

对于可读流来说，push(data)时，data只能是String或Buffer类型，而消耗时data事件输出的数据都是Buffer类型。对于可写流来说，write(data)时，data只能是String或Buffer类型，_write(data)调用时传进来的data都是Buffer类型。

也就是说，流中的数据默认情况下都是Buffer类型。产生的数据一放入流中，便转成Buffer被消耗；写入的数据在传给底层写逻辑时，也被转成Buffer类型。

但每个构造函数都接收一个配置对象，有一个objectMode的选项，一旦设置为true，就能出现"种瓜得瓜，种豆得豆"的效果。

1. Readable未设置objectMode时：

```
const Readable = require('stream').Readable

const readable = Readable()

readable.push('a')
readable.push('b')
readable.push(null)

readable.on('data', data => console.log(data))
```

2. Readable设置objectMode后：

```
const Readable = require('stream').Readable

const readable = Readable({ objectMode: true })

readable.push('a')
readable.push('b')
readable.push({})
readable.push(null)
```

```
readable.on('data', data => console.log(data))
```

可见，设置objectMode后，push(data)的数据被原样地输出了。此时，可以生产任意类型的数据。

Events

Events模块是node的核心模块之一，几乎所有常用的node模块都继承了events模块，比如http、fs等。

模块本身非常简单，API虽然也不少，但常用的就那么几个，这里举几个简单例子。

例子1：单个事件监听器

```
var EventEmitter = require('events');

class Man extends EventEmitter {}

var man = new Man();

man.on('wakeup', function(){
    console.log('man has woken up');
});

man.emit('wakeup');
// 输出如下:
// man has woken up
```

例子2：同个事件，多个事件监听器

可以看到，事件触发时，事件监听器按照注册的顺序执行。

```
var EventEmitter = require('events');

class Man extends EventEmitter {}

var man = new Man();

man.on('wakeup', function(){
    console.log('man has woken up');
});

man.on('wakeup', function(){
    console.log('man has woken up again');
});

man.emit('wakeup');

// 输出如下:
// man has woken up
// man has woken up again
```

例子3：只运行一次的事件监听器

```
var EventEmitter = require('events');

class Man extends EventEmitter {}

var man = new Man();

man.on('wakeup', function(){
    console.log('man has woken up');
});

man.once('wakeup', function(){
    console.log('man has woken up again');
});
```

```
man.emit('wakeup');
man.emit('wakeup');

// 输出如下:
// man has woken up
// man has woken up again
// man has woken up
```

例子4：注册事件监听器前，事件先触发

可以看到，注册事件监听器前，事件先触发，则该事件会直接被忽略。

```
var EventEmitter = require('events');

class Man extends EventEmitter {}

var man = new Man();

man.emit('wakeup', 1);

man.on('wakeup', function(index){
    console.log('man has woken up ->' + index);
});

man.emit('wakeup', 2);
// 输出如下:
// man has woken up ->2
```

例子5：异步执行，还是顺序执行

例子很简单，但非常重要。究竟是代码1先执行，还是代码2先执行，这点差异，无论对于我们理解别人的代码，还是自己编写node程序，都非常关键。

实践证明，代码1先执行了

```
var EventEmitter = require('events');

class Man extends EventEmitter {}

var man = new Man();

man.on('wakeup', function(){
    console.log('man has woken up'); // 代码1
});

man.emit('wakeup');

console.log('woman has woken up');  // 代码2

// 输出如下:
// man has woken up
// woman has woken up
```

例子6：移除事件监听器

```
var EventEmitter = require('events');

function wakeup(){
    console.log('man has woken up');
}

class Man extends EventEmitter {}

var man = new Man();

man.on('wakeup', wakeup);
man.emit('wakeup');

man.removeListener('wakeup', wakeup);
man.emit('wakeup');
```

```
// 输出如下:
// man has woken up
```

手写实现EventEmitter

event.js

全局对象解析

JavaScript 中有一个特殊的对象，称为全局对象（Global Object），它及其所有属性都可以在程序的任何地方访问，即全局变量。

在浏览器 JavaScript 中，通常 window 是全局对象， 而 Node.js 中的全局对象是 global，所有全局变量（除了 global 本身以外）都是 global 对象的属性。

在 Node.js 我们可以直接访问到 global 的属性，而不需要在应用中包含它。

全局对象和全局变量

Global 最根本的作用是作为全局变量的宿主。按照 ECMAScript 的定义，满足以下条 件的变量是全局变量：

在最外层定义的变量；
全局对象的属性；
隐式定义的变量（未定义直接赋值的变量）。
当你定义一个全局变量时，这个变量同时也会成为全局对象的属性，反之亦然。需要注 意的是，在 Node.js 中你不可能在最外层定义变量，因为所有用户代码都是属于当前模块的， 而模块本身不是最外层上下文。

注意： 永远使用 var 定义变量以避免引入全局变量，因为全局变量会污染 命名空间，提高代码的耦合风险。

__filename

filename 表示当前正在执行的脚本的文件名。它将输出文件所在位置的绝对路径，且和命令行参数所指定的文件名不一定相同。 如果在模块中，返回的值是模块文件的路径。

```
console.log( __filename );
```

__dirname

dirname 表示当前执行脚本所在的目录。

```
console.log( __dirname );
```

setTimeout(cb, ms)

setTimeout(cb, ms) 全局函数在指定的毫秒(ms)数后执行指定函数(cb)。： setTimeout() 只执行一次指定函数。

返回一个代表定时器的句柄值。

```
function printHello(){
   console.log( "Hello, World!");
}
// 两秒后执行以上函数
setTimeout(printHello, 2000);
```

clearTimeout

setInterval

clearInterval

console

process

Process 是一个全局变量，即 global 对象的属性。

它用于描述当前Node.js 进程状态的对象，提供了一个与操作系统的简单接口。通常在你写本地命令行程序的时候，少不了要 和它打交道。下面将会介绍 process 对象的一些最常用的成员方法。

1. exit
   当进程准备退出时触发。

2. beforeExit
   当 node 清空事件循环，并且没有其他安排时触发这个事件。通常来说，当没有进程安排时 node 退出，但是 'beforeExit' 的监听器可以异步调用，这样 node 就会继续执行。

3. uncaughtException
   当一个异常冒泡回到事件循环，触发这个事件。如果给异常添加了监视器，默认的操作（打印堆栈跟踪信息并退出）就不会发生。

4. Signal 事件
   当进程接收到信号时就触发。信号列表详见标准的 POSIX 信号名，如 SIGINT、SIGUSR1 等。

```
process.on('exit', function(code) {
  // 以下代码永远不会执行
  setTimeout(function() {
    console.log("该代码不会执行");
  }, 0);

  console.log('退出码为:', code);
});
console.log("程序执行结束");
```

退出的状态码

1. Uncaught Fatal Exception
   有未捕获异常，并且没有被域或 uncaughtException 处理函数处理。

   3 Internal JavaScript Parse Error
   JavaScript的源码启动 Node 进程时引起解析错误。非常罕见，仅会在开发 Node 时才会有。

   4 Internal JavaScript Evaluation Failure
   JavaScript 的源码启动 Node 进程，评估时返回函数失败。非常罕见，仅会在开发 Node 时才会有。

   5 Fatal Error
   V8 里致命的不可恢复的错误。通常会打印到 stderr ，内容为： FATAL ERROR

   6 Non-function Internal Exception Handler
   未捕获异常，内部异常处理函数不知为何设置为on-function，并且不能被调用。

   7 Internal Exception Handler Run-Time Failure
   未捕获的异常， 并且异常处理函数处理时自己抛出了异常。例如，如果 process.on(‘uncaughtException’) 或 domain.on(‘error’) 抛出了异常。

   9 Invalid Argument
   可能是给了未知的参数，或者给的参数没有值。

   10 Internal JavaScript Run-Time Failure
   JavaScript的源码启动 Node 进程时抛出错误，非常罕见，仅会在开发 Node 时才会有。

   12 Invalid Debug Argument
   设置了参数–debug 和/或 –debug-brk，但是选择了错误端口。

   >128 Signal Exits
   如果 Node 接收到致命信号，比如SIGKILL 或 SIGHUP，那么退出代码就是128 加信号代码。这是标准的 Unix 做法，退出信号代码放在高位。

```
// 输出到终端
process.stdout.write("Hello World!" + "\n");

// 通过参数读取
process.argv.forEach(function(val, index, array) {
   console.log(index + ': ' + val);
});

// 获取执行路局
console.log(process.execPath);

// 平台信息
console.log(process.platform);
```

看看这段代码输出什么

```
console.log(this);

module.exports.foo = 5;

console.log(this);
```

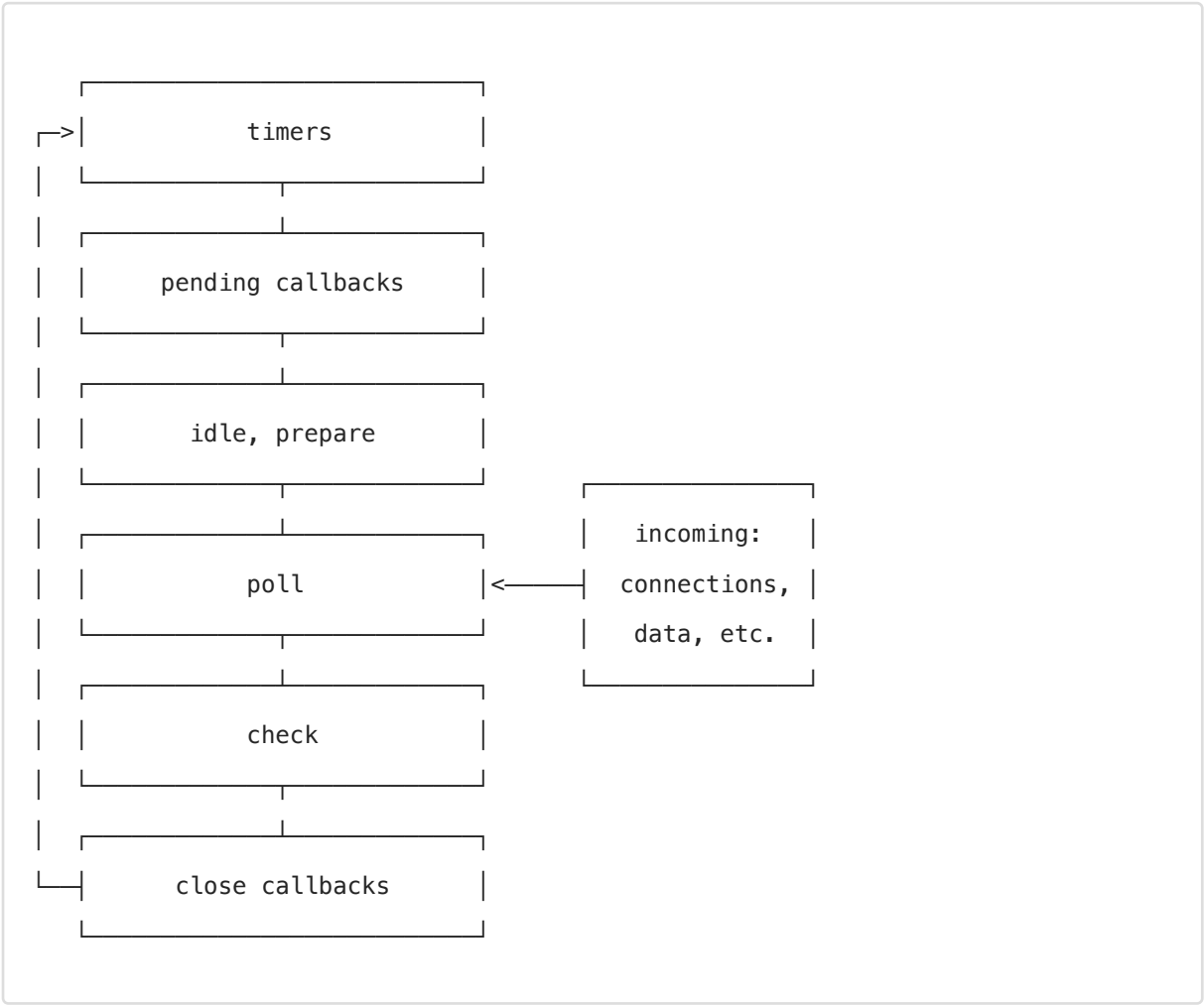
node.js 事件循环模型

什么是事件循环

事件循环使 Node.js 可以通过将操作转移到系统内核中来执行非阻塞 I/O 操作（尽管 JavaScript 是单线程的）。

由于大多数现代内核都是多线程的，因此它们可以处理在后台执行的多个操作。 当这些操作之一完成时，内核会告诉 Node.js，以便可以将适当的回调添加到轮询队列中以最终执行。

Node.js 启动时，它将初始化事件循环，处理提供的输入脚本，这些脚本可能会进行异步 API 调用，调度计时器或调用 process.nextTick， 然后开始处理事件循环。



每个阶段都有一个要执行的回调 FIFO 队列。 尽管每个阶段都有其自己的特殊方式，但是通常，当事件循环进入给定阶段时，它将执行该阶段特定的任何操作，然后在该阶段的队列中执行回调，直到队列耗尽或执行回调的最大数量为止。 当队列已为空或达到回调限制时，事件循环将移至下一个阶段，依此类推。

各阶段概览

1. timers：此阶段执行由 setTimeout 和 setInterval 设置的回调。
2. pending callbacks：执行推迟到下一个循环迭代的 I/O 回调。
3. idle, prepare, ：仅在内部使用。

4. poll：取出新完成的 I/O 事件；执行与 I/O 相关的回调（除了关闭回调，计时器调度的回调和 setImmediate 之外，几乎所有这些回调） 适当时，node 将在此处阻塞。
5. check：在这里调用 setImmediate 回调。
6. close callbacks：一些关闭回调，例如 socket.on('close', ...)。

在每次事件循环运行之间，Node.js 会检查它是否正在等待任何异步 I/O 或 timers，如果没有，则将其干净地关闭。

各阶段详细解析

timers 计时器阶段

计时器可以在回调后面指定时间阈值，但这不是我们希望其执行的确切时间。 计时器回调将在经过指定的时间后尽早运行。 但是，操作系统调度或其他回调的运行可能会延迟它们。– 执行的实际时间不确定

```
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
  }
});
```

当事件循环进入 poll 阶段时，它有一个空队列（fs.readFile 尚未完成），因此它将等待直到达到最快的计时器 timer 阈值为止。
等待 95 ms 过去时，fs.readFile 完成读取文件，并将需要 10ms 完成的其回调添加到轮询 (poll) 队列并执行。
回调完成后，队列中不再有回调，此时事件循环已达到最早计时器 (timer) 的阈值 (100ms)，然后返回到计时器 (timer) 阶段以执行计时器的回调。
在此示例中，您将看到计划的计时器与执行的回调之间的总延迟为 105ms。

pending callbacks 阶段

此阶段执行某些系统操作的回调，例如 TCP 错误。 平时无需关注

轮询 poll 阶段

轮询阶段具有两个主要功能：

1. 计算应该阻塞并 I/O 轮询的时间
2. 处理轮询队列 (poll queue) 中的事件

当事件循环进入轮询 (poll) 阶段并且没有任何计时器调度 (timers scheduled) 时，将发生以下两种情况之一：

1. 如果轮询队列 (poll queue) 不为空，则事件循环将遍历其回调队列，使其同步执行，直到队列用尽或达到与系统相关的硬限制为止 (到底是哪些硬限制？)。
2. 如果轮询队列为空，则会发生以下两种情况之一：
   2.1 如果已通过 setImmediate 调度了脚本，则事件循环将结束轮询 poll 阶段，并继续执行 check 阶段以执行那些调度的脚本。
   2.2 如果脚本并没有 setImmediate 设置回调，则事件循环将等待 poll 队列中的回调，然后立即执行它们。

一旦轮询队列 (poll queue) 为空，事件循环将检查哪些计时器 timer 已经到时间。 如果一个或多个计时器 timer 准备就绪，则事件循环将返回到计时器阶段，以执行这些计时器的回调。

检查阶段 check

此阶段允许在轮询 poll 阶段完成后立即执行回调。 如果轮询 poll 阶段处于空闲，并且脚本已使用 setImmediate 进入 check 队列，则事件循环可能会进入 check 阶段，而不是在 poll 阶段等待。

setImmediate 实际上是一个特殊的计时器，它在事件循环的单独阶段运行。 它使用 libuv API，该 API 计划在轮询阶段完成后执行回调。

通常，在执行代码时，事件循环最终将到达轮询 poll 阶段，在该阶段它将等待传入的连接，请求等。但是，如果已使用 setImmediate 设置回调并且轮询阶段变为空闲，则它将将结束并进入 check 阶段，而不是等待轮询事件。

close callbacks 阶段

如果套接字或句柄突然关闭（例如 socket.destroy），则在此阶段将发出 'close' 事件。 否则它将通过 process.nextTick 发出。

setImmediate 和 setTimeout 的区别

setImmediate 和 setTimeout 相似，但是根据调用时间的不同，它们的行为也不同。

- setImmediate 设计为在当前轮询 poll 阶段完成后执行脚本。
- setTimeout 计划在以毫秒为单位的最小阈值过去之后运行脚本。

Tips: 计时器的执行顺序将根据调用它们的上下文而有所不同。 如果两者都是主模块中调用的，则时序将受到进程性能的限制.

来看两个例子：

1. 在主模块中执行

   两者的执行顺序是不固定的, 可能timeout在前, 也可能immediate在前

2. 在同一个I/O回调里执行

   setImmediate总是先执行

问题：那为什么在外部 (比如主代码部分 mainline) 这两者的执行顺序不确定呢?

解答：在 主代码 部分执行 setTimeout 设置定时器 (此时还没有写入队列)，与 setImmediate 写入 check 队列。

Mainline 执行完开始事件循环，第一阶段是 timers，这时候 timers 队列可能为空，也可能有回调；
如果没有那么执行 check 队列的回调，下一轮循环在检查并执行 timers 队列的回调；
如果有就先执行 timers 的回调，再执行 check 阶段的回调。因此这是 timers 的不确定性导致的。

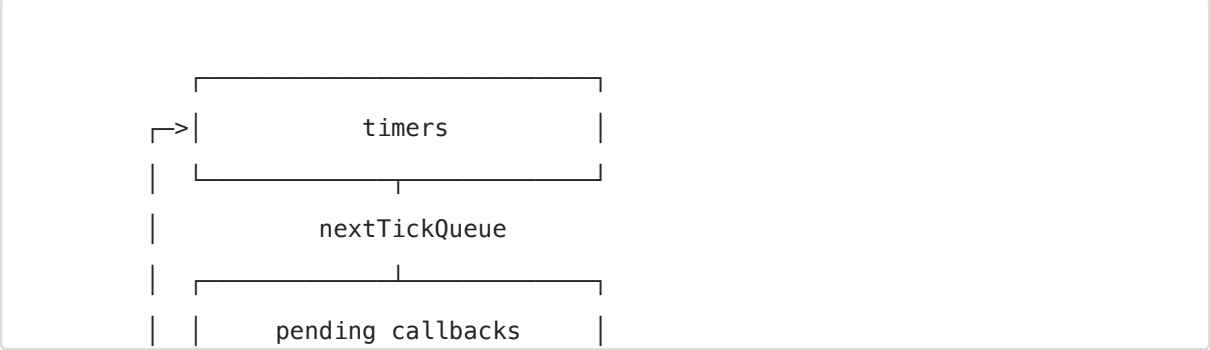
process.nextTick

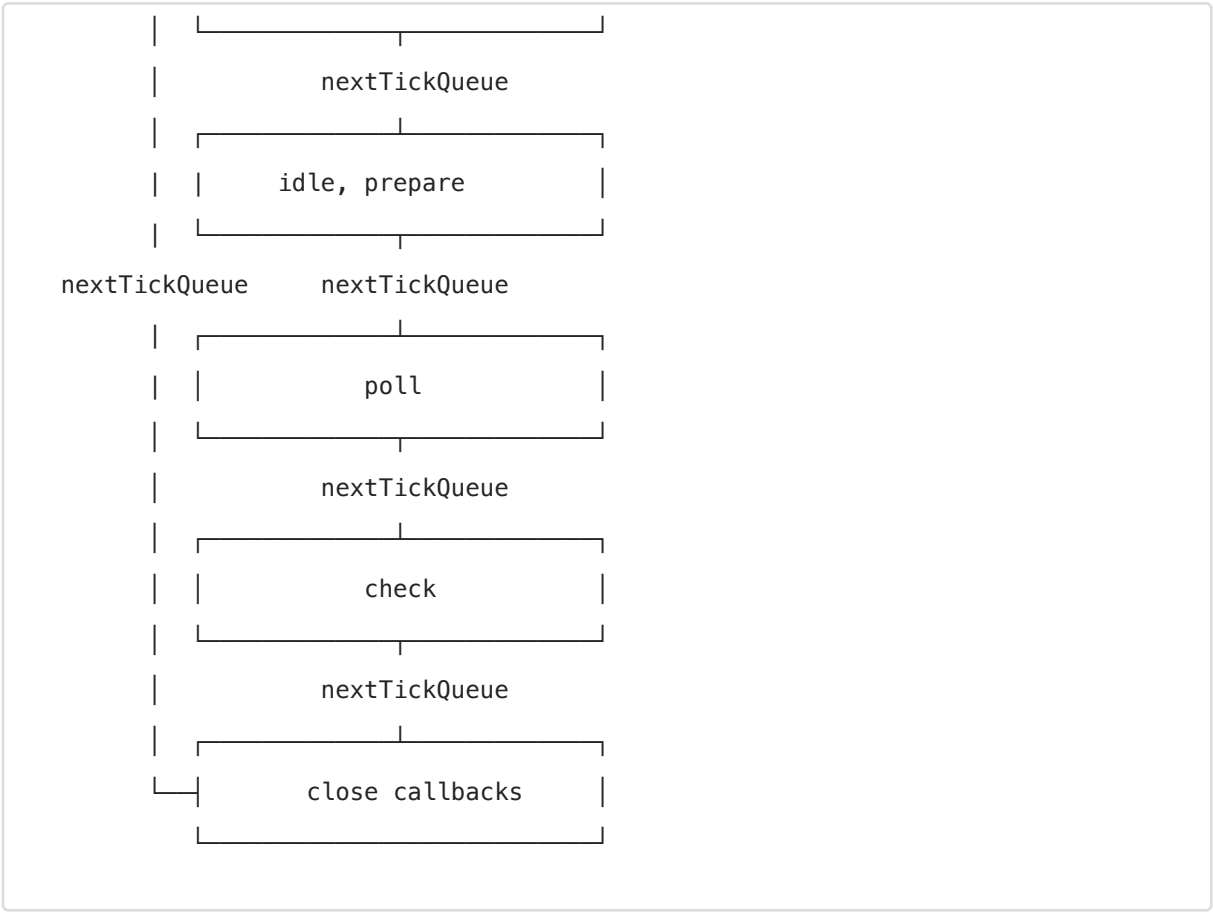
Process.nextTick 从技术上讲不是事件循环的一部分。 相反，无论事件循环的当前阶段如何，都将在当前操作完成之后处理 nextTickQueue

process.nextTick 和 setImmediate 的区别

- process.nextTick 在同一阶段立即触发
- setImmediate fires on the following iteration or 'tick' of the event loop (在事件循环接下来的阶段迭代中执行 - check 阶段)。

nextTick在事件循环中的位置





Microtasks 微任务

在 Node 领域，微任务是来自以下对象的回调：

1. process.nextTick()
2. then()

在主线结束后以及事件循环的每个阶段之后，立即运行微任务回调。

resolved 的 promise.then 回调像微处理一样执行，就像 process.nextTick 一样。 虽然，如果两者都在同一个微任务队列中，则将首先执行 process.nextTick 的回调。

优先级 process.nextTick > promise.then

看代码输出顺序

```
async function async1() {
    console.log('async1 start')
```

```
    await async2()
    console.log('async1 end')
}
async function async2() {
    console.log('async2')
}
console.log('script start')
setTimeout(function () {
    console.log('setTimeout0')
    setTimeout(function () {
        console.log('setTimeout1');
    }, 0);
    setImmediate(() => console.log('setImmediate'));
}, 0)

process.nextTick(() => console.log('nextTick'));
async1();
new Promise(function (resolve) {
    console.log('promise1')
    resolve();
    console.log('promise2')
}).then(function () {
    console.log('promise3')
})
console.log('script end')
```