Node

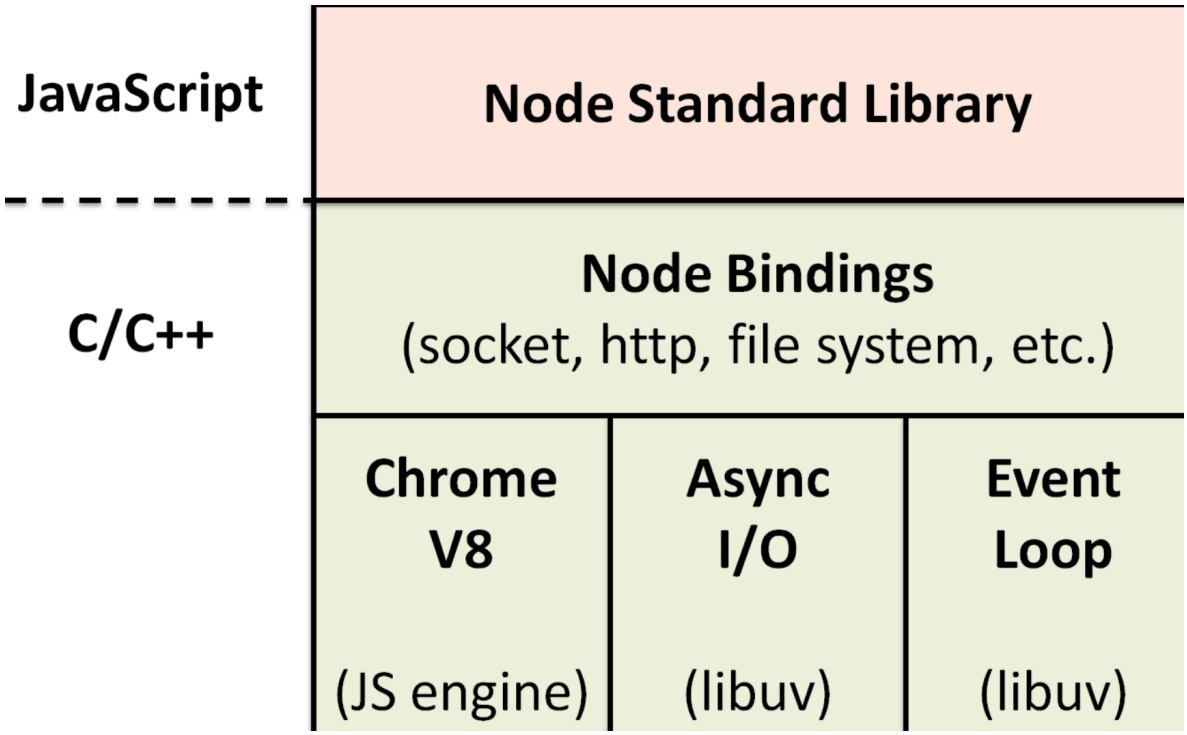
为什么要用node?

总结起来node有以下几个特点:简单强大，轻量可扩展．简单体现在node使用的是javascript,json来进行编码，人人都会；强大体现在非阻塞IO,可以适应分块传输数据，较慢的网络环境，尤其擅长高并发访问；轻量体现在node本身既是代码，又是服务器，前后端使用统一语言;可扩展体现在可以轻松应对多实例，多服务器架构，同时有海量的第三方应用组件

async 函数的实现原理，就是将 Generator 函数和自动执行器，包装在一个函数里。

node的构架是什么样子的?

主要分为三层，应用app >> V8及node内置架构 >> 操作系统. V8是node运行的环境，可以理解为node虚拟机．node内置架构又可分为三层: 核心模块(javascript实现) >> c++绑定 >> libuv + CAes + http.



node有哪些核心模块?

EventEmitter, Stream, FS, Net和全局对象

node有哪些全局对象?:process, console, Buffer和exports

process有哪些常用方法?

process.stdin, process.stdout, process.stderr, process.on, process.env, process.argv, process.arch, process.platform, process.exit

console有哪些常用方法?

console.log/console.info, console.error/console.warning, console.time/ console.timeEnd, console.trace, console.table

node有哪些定时功能?

setTimeout/clearTimeout, setInterval/clearInterval, setImmediate/clearImmediate, process.nextTick

node中的事件循环是什么样子的?

event loop其实就是一个事件队列，先加入先执行，执行完一次队列，再次循环遍历看有没有新事件加入队列．执行中的叫IO events, setImmediate是在当前队列立即执行,setTimout/setInterval是把执行定时到下一个队列，process.nextTick是在当前执行完，下次遍历前执行．所以总体顺序是: IO events >> setImmediate >> setTimeout/setInterval >> process.nextTick

node中的Buffer如何应用?

Buffer是用来处理二进制数据的，比如图片，mp3,数据库文件等.Buffer支持各种编码解码，二进制字符串互转．

什么是EventEmitter?

EventEmitter是node中一个实现观察者模式的类，主要功能是监听和发射消息，用于处理多模块交互问题.

如何实现一个EventEmitter?

主要分三步：定义一个子类，调用构造函数，继承EventEmitter

```
var util = require('util');
var EventEmitter = require('events').EventEmitter;
function MyEmitter() {
    EventEmitter.call(this);
} // 构造函数
util.inherits(MyEmitter, EventEmitter); // 继承
var em = new MyEmitter();
em.on('hello', function (data) {
    console.log('收到事件hello的数据:', data);
}); // 接收事件，并打印到控制台
em.emit('hello', 'EventEmitter传递消息真方便!');
```

EventEmitter有哪些典型应用?

1) 模块间传递消息 2) 回调函数内外传递消息 3) 处理流数据，因为流是在EventEmitter基础上实现的. 4) 观察者模式发射触发机制相关应用

怎么捕获EventEmitter的错误事件?

监听error事件即可．如果有多个EventEmitter,也可以用domain来统一处理错误事件.

```
var domain = require('domain');
var myDomain = domain.create();
myDomain.on('error', function (err) {
    console.log('domain接收到的错误事件:', err);
}); // 接收事件并打印
myDomain.run(function () {
    var emitter1 = new MyEmitter();
    emitter1.emit('error', '错误事件来自emitter1');
    emitter2 = new MyEmitter();
    emitter2.emit('error', '错误事件来自emitter2');
});
```

EventEmitter中的newListenser事件有什么用处?

newListener可以用来做事件机制的反射，特殊应用，事件管理等．当任何on事件添加到EventEmitter时，就会触发newListener事件，基于这种模式，我们可以做很多自定义处理.

```
var emitter3 = new MyEmitter();
emitter3.on('newListener', function (name, listener) {
    console.log("新事件的名字:", name);
    console.log("新事件的代码:", listener);
    setTimeout(function () {
        console.log("我是自定义延时处理机制");
    }, 1000);
});
emitter3.on('hello', function () {
    console.log('hello  node');
});
```

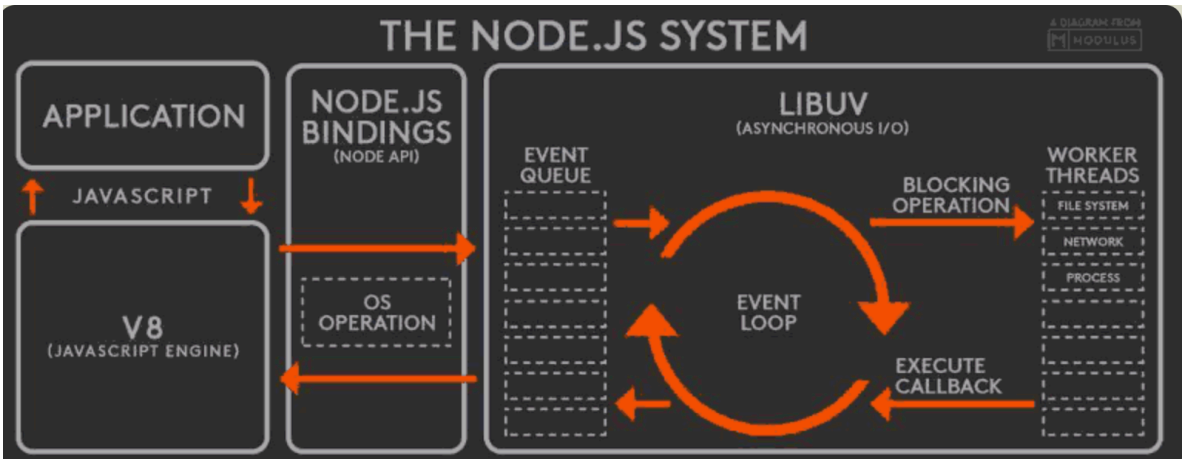
Nodejs的Eventloop

(1）V8引擎解析JavaScript脚本。

（2）解析后的代码，调用Node API。

（3）libuv库负责Node API的执行。它将不同的任务分配给不同的线程，形成一个Event Loop（事件循环），以异步的方式将任务的执行结果返回给V8引擎。

（4）V8引擎再将结果返回给用户。



什么是Stream?

stream是基于事件EventEmitter的数据管理模式．由各种不同的抽象接口组成，主要包括可写，可读，可读写，可转换等几种类型

Stream有什么好处?

非阻塞式数据处理提升效率，片断处理节省内存，管道处理方便可扩展等.

Stream有哪些典型应用?

文件，网络，数据转换，音频视频等

怎么捕获Stream的错误事件?

监听error事件，方法同EventEmitter.

有哪些常用Stream,分别什么时候使用?

Readable为可被读流，在作为输入数据源时使用；Writable为可被写流,在作为输出源时使用；Duplex为可读写流,它作为输出源接受被写入，同时又作为输入源被后面的流读出．Transform机制和Duplex一样，都是双向流，区别时Transfrom只需要实现一个函数_transfrom(chunk, encoding, callback);而Duplex需要分别实现_read(size)函数和_write(chunk, encoding, callback)函数

实现一个Writable Stream

三步走:1)构造函数call Writable 2)　继承Writable 3) 实现_write(chunk, encoding, callback)函数

```
var Writable = require('stream').Writable;
var util = require('util');
function MyWritable(options) {
    Writable.call(this, options);
} // 构造函数
util.inherits(MyWritable, Writable); // 继承自Writable
MyWritable.prototype._write = function (chunk, encoding, callback) {
    console.log("被写入的数据是:", chunk.toString()); // 此处可对写入的数据进行处理
    callback();
};
process.stdin.pipe(new MyWritable()); // stdin作为输入源，MyWritable作为输出源
```

内置的fs模块架构是什么样子的?

fs模块主要由下面几部分组成: 1) POSIX文件Wrapper,对应于操作系统的原生文

件操作 2) 文件流 fs.createReadStream和fs.createWriteStream 3) 同步文件读写,fs.readFileSync和fs.writeFileSync 4) 异步文件读写, fs.readFile和fs.writeFile

读写一个文件有多少种方法?

总体来说有四种: 1) POSIX式低层读写 2) 流式读写 3) 同步文件读写 4) 异步文件读写

怎么读取json配置文件?

主要有两种方式,

第一种是利用node内置的require('data.json')机制，直接得到js对象;

第二种是读入文件入内容，然后用JSON.parse(content)转换成js对象.

二者的区别是require机制情况下，如果多个模块都加载了同一个json文件，那么其中一个改变了js对象，其它跟着改变，这是由node模块的缓存机制造成的，只有一个js模块对象; 第二种方式则可以随意改变加载后的js变量，而且各模块互不影响，因为他们都是独立的，是多个js对象.

fs.watch和fs.watchFile有什么区别，怎么应用?

二者主要用来监听文件变动. fs.watch利用操作系统原生机制来监听，可能不适用网络文件系统; fs.watchFile则是定期检查文件状态变更，适用于网络文件系统，但是相比fs.watch有些慢，因为不是实时机制.

node的网络模块架构是什么样子的?

node全面支持各种网络服务器和客户端，包括tcp, http/https, tcp, udp, dns, tls/ssl等

node是怎样支持https,tls的

主要实现以下几个步骤即可:

1) openssl生成公钥私钥

2) 服务器或客户端使用https替代http

3) 服务器或客户端加载公钥私钥证书

实现一个简单的http服务器?

思路是加载http模块，创建服务器，监听端口

```
var http = require('http'); // 加载http模块
http.createServer(function (req, res) {
    res.writeHead(200, {
        'Content-Type': 'text/html'
    }); // 200代表状态成功, 文档类型是给浏览器识别用的
    res.write('<meta charset="UTF-8"> <h1>标题啊! </h1> <font color="red">这么原生</font>');
    // 返回给客户端的html数据
    res.end(); // 结束输出流
}).listen(3000); // 绑定3000, 查看效果请访问 http://localhost:3000
```

为什么需要child-process?

node是异步非阻塞的，这对高并发非常有效. 可是我们还有其它一些常用需求，比如和操作系统shell命令交互，调用可执行文件，创建子进程进行阻塞式访问或高CPU计算等，child-process就是为满足这些需求而生的. child-process顾名思义，就是把node阻塞的工作交给子进程去做.

exec,execFile,spawn和fork都是做什么用的?

exec可以用操作系统原生的方式执行各种命令，如管道 cat ab.txt | grep hello; execFile是执行一个文件; spawn是流式和操作系统进行交互; fork是两个node程序(javascript)之间时行交互.

实现一个简单的命令行交互程序?

那就用spawn吧.

```
var cp = require('child_process');
var child = cp.spawn('echo', ['你好', "钩子"]);
// 执行命令
child.stdout.pipe(process.stdout);
// child.stdout是输入流，process.stdout是输出流
// 这句的意思是将子进程的输出作为当前程序的输入流，然后重定向到当前程序的标准输出，即控制台
```

两个node程序之间怎样交互?

用fork嘛，上面讲过了．原理是子程序用process.on, process.send，父程序里用child.on,child.send进行交互.

```
//1) fork - parent.js
var cp = require('child_process');
var child = cp.fork('./fork-child.js');
child.on('message', function (msg) {
    console.log('老爸从儿子接受到数据:', msg);
});
child.send('我是你爸爸，送关怀来了!');

//2) fork - child.js
process.on('message', function (msg) {
    console.log("儿子从老爸接收到的数据:", msg);
    process.send("我不要关怀，我要银民币! ");
});
```

怎样让一个js文件变得像linux命令一样可执行

1) 在myCommand.js文件头部加入 #!/usr/bin/env node

2) chmod命令把js文件改为可执行即可

3) 进入文件目录，命令行输入myComand就是相当于node myComand.js了

child-process和process的stdin,stdout,stderror是一样的吗?

概念都是一样的，输入，输出，错误，都是流．区别是在父程序眼里，子程序的stdout是输入流，stdin是输出流

node中的异步和同步怎么理解

node是单线程的，异步是通过一次次的循环事件队列来实现的．同步则是说阻塞式的IO,这在高并发环境会是一个很大的性能问题，所以同步一般只在基础框

架的启动时使用，用来加载配置文件，初始化程序什么的.

有哪些方法可以进行异步流程的控制?

1) 多层嵌套回调 2) 为每一个回调写单独的函数，函数里边再回调 3) 用第三方框架比方async, q, promise等

怎样绑定node程序到80端口?

多种方式 1) sudo 2) apache/nginx代理 3) 用操作系统的firewall iptables进行端口重定向

有哪些方法可以让node程序遇到错误后自动重启?

1) runit 2) forever 3) nohup npm start &

怎样充分利用多个CPU?

一个CPU运行一个node实例

怎样调节node执行单元的内存大小?

用--max-old-space-size 和 --max-new-space-size 来设置 v8 使用内存的上限

程序总是崩溃，怎样找出问题在哪里?

1) node --prof 查看哪些函数调用次数多 2) memwatch和heapdump获得内存快照进行对比，查找内存溢出

有哪些常用方法可以防止程序崩溃?

1) try-catch-finally 2) EventEmitter/Stream error事件处理 3) domain统一控制 4) jshint静态检查 5) jasmine/mocha进行单元测试

怎样调试node程序?

node --debug app.js 和node-inspector

express和koa的区别：

Express 和 Koa 最明显的差别就是 Handler 的处理方法，一个是普通的回调函数，一个是利用生成器函数（Generator Function）来作为响应器。往里头儿说就是 Express 是在同一线程上完成当前进程的所有 HTTP 请求，而 Koa 利用 co 作为底层运行框架，利用 Generator 的特性，实现“协程响应”（并不能将 Generator 等价于协程，在 V8 的邮件列表中对 Generator 的定义基本是 `coroutine-like`），然而 co 这个库对 Generator 的使用方法并非当初 Generator 的设计初衷。详细可以看这里：Koa, co and coroutine

Koa 2.0 与 Koa 1.x 版本的最大区别就是使用了 ES7 中 Async/Await 的特性，代替了 co 的 Generator Function，好处是摆脱了 co 的“暧昧”实现方法，改而使用原生的 Coroutine-like(maybe) 语法。缺点是与从 connect/express 迁移到 Koa 1.x 一样

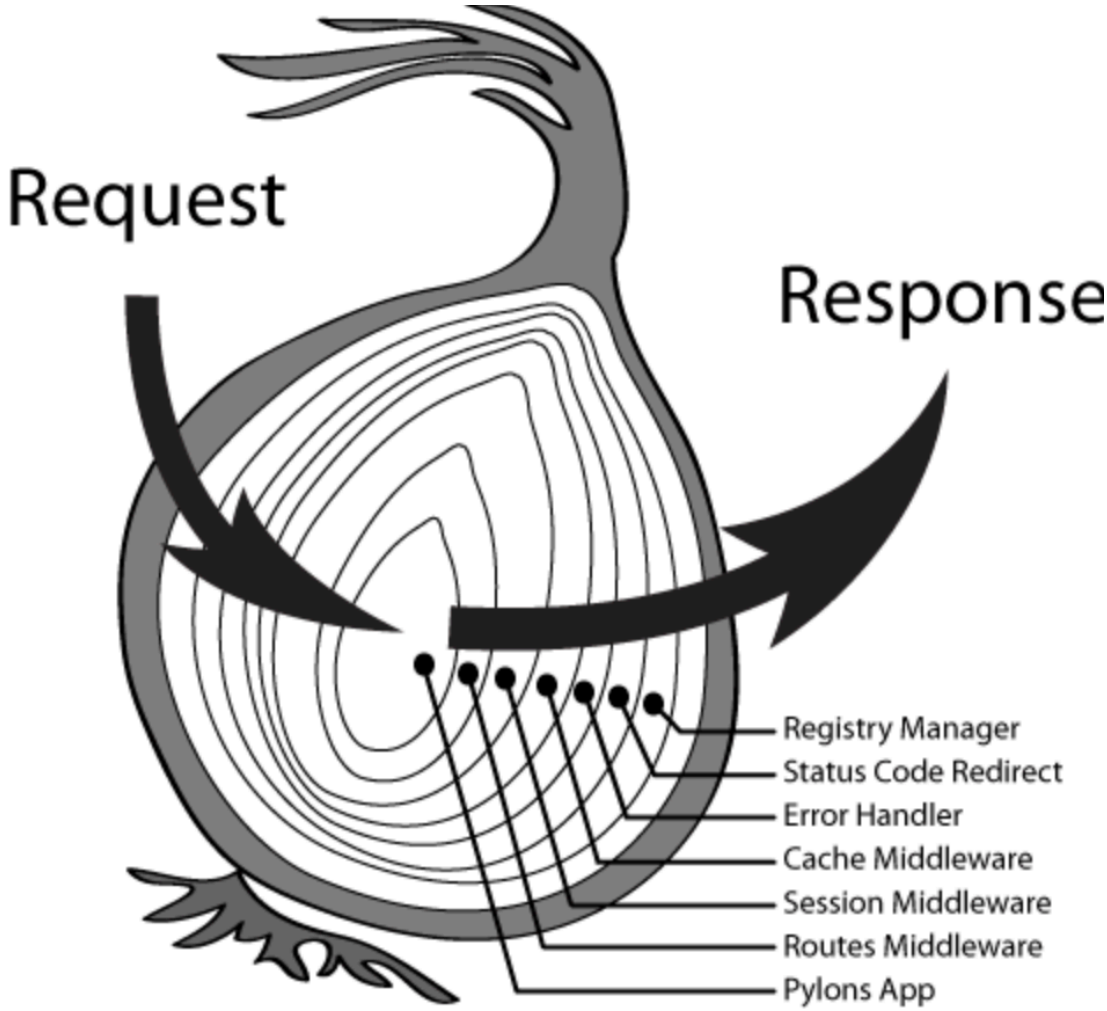
Koa2的特性：

- 只提供封装好http上下文、请求、响应，以及基于async/await的中间件容器。
- 利用ES7的async/await的来处理传统回调嵌套问题和代替koa@1的generator，但是需要在node.js 7.x的harmony模式下才能支持async/await。
- 中间件只支持 async/await 封装的，如果要使用koa@1基于generator中间

件，需要通过中间件koa-convert封装一下才能使用。koa-convert的思路就是对Generator封装一层Promise，使上一个中间件可以利用await next()的方式调用，对于Generator的执行，利用co库，从而达到了兼容的目的。

co 函数库是著名程序员 TJ Holowaychuk 于2013年6月发布的一个小工具，用于 Generator 函数的自动执行。**co** 函数库其实就是将两种自动执行器（**Thunk** 函数和 **Promise** 对象），包装成一个库。使用 co 的前提条件是，Generator 函数的 yield 命令后面，只能是 Thunk 函数或 Promise 对象。

koa的洋葱中间件：



源码非常的简单，实现的功能就是将所有的中间件串联起来，首先给倒数第一个中间件传入一个 `noop` 作为其 `next` ，再将这个整理后的倒数第一个中间作为 `next` 传入倒数第二个中间件，最终得到的 `next` 就是整理后的第一个中间件。说起来比较复杂，画图来看：