node原理介绍

like@tuniu

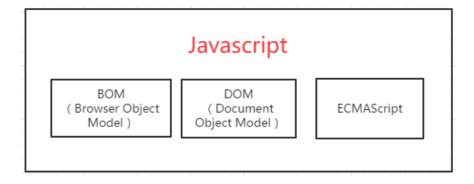
2016-11-22

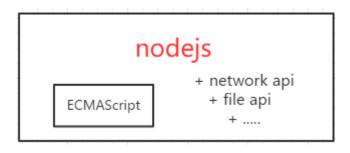
node

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

high performance web server: event-driven,

non-blocking I/O. e.g: Nginx





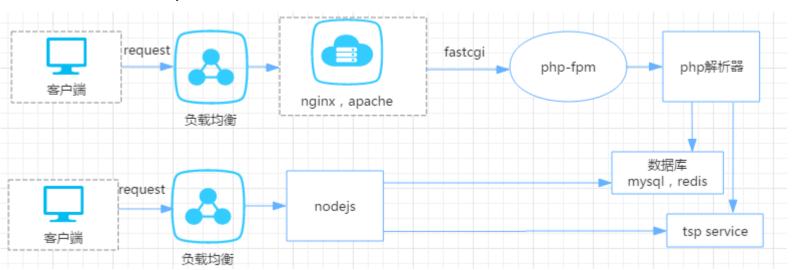
node的特点

- 异步 I/O: fs.readFile, 多个操作, (io >> data processing) max(m1,m2...) < m1 + m2 + ...
- 适用于I/O bound(I/O密集型)
- 单线程: 优点: 状态同步, 死锁, 线程上下文交换

缺点:无法利用多核CPU。错误会引起整个应用退出,大量计算占用CPU导致无法继续调用异步I/O。

多进程: child process, cluster...

- 跨平台: libuv
- 事件与回调函数:结合异步 I/O,将事件点暴露给业务逻辑。
- web应用



基本架构

stream.js

```
▼ node-0.10.16
▼ node-0.10.16
                                   benchmark
  benchmark
                                   ▶ deps

▼ deps

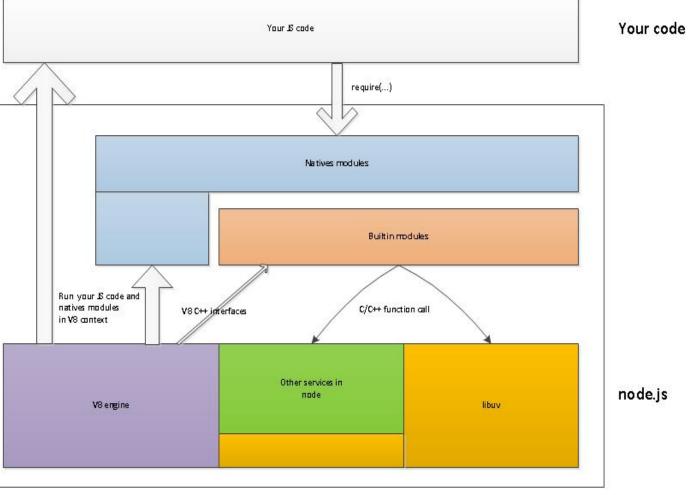
                                   ▶ doc
   cares
                                   ▼ lib
    http_parser
                                        _debugger.js
                                        _linklist.js
    ▶ npm
                                        _stream_duplex.js
    openssi
                                        _stream_passthrough.js

    uv

                                        _stream_readable.js
    ▶ v8
                                        _stream_transform.js
   ▶ zlib
                                        _stream_writable.js
  ▶ doc
                                        assert.is
  ▶ lib
                                        buffer.js
  ▶ src
                                        child process.js
 ▶ test
                                        cluster.js
                                        console.js
 ▶ tools
                                        constants.js
    .gitattributes
                                        crypto.js
    .gitignore
                                        dgram.js
     .mailmap
                                        dns.js
    .travis.yml
                                        domain.js
    AUTHORS
                                        events.js
    BSDmakefile
                                        freelist.js
                                        fs.js
    ChangeLog
                                        http.js
    common.gypi
                                        https.js
    configure
                                        module.js
    CONTRIBUTING.md
                                        net.js
    LICENSE
                                        os.js
    Makefile
                                        path.js
    node.gyp
                                        punycode.js
    README.md
                                        querystring.js
    vcbuild.bat
                                        readline.js
                                        repl.js
```

```
▼ node-0.10.16
 ▶ benchmark
 ▶ deps
 ▶ doc
 ▶ lib
 ▼ src
   ▶ res
      cares_wrap.cc
      fs_event_wrap.cc
      handle_wrap.cc
      handle wrap.h
      macros.py
      ngx-queue.h
      node.cc
      node.d
      node.h
      node.js
      node_buffer.cc
      node_buffer.h
      node constants.cc
      node constants.h
      node_counters.cc
      node_counters.h
      node_crypto.cc
      node_crypto.h
      node_crypto_groups.h
      node_dtrace.cc
      node_dtrace.h
      node_extensions.cc
      node_extensions.h
      node file.cc
      node_file.h
      node_http_parser.cc
      node_http_parser.h
      node_internals.h
      node_javascript.cc
      node_javascript.h
```

```
var fs = require('fs');
fs.readFile('./package.json',(err, data)=>{
  console.log('读取文件1完成');
});
```



CPU/RAM/DISK OS(kernel & other libraries)

Host environment

node模块机制

CommonJs规范: 1.模块引用 2.模块定义 3.模块标识

模块查找与引入流程:文件模块,核心模块,内建模块,

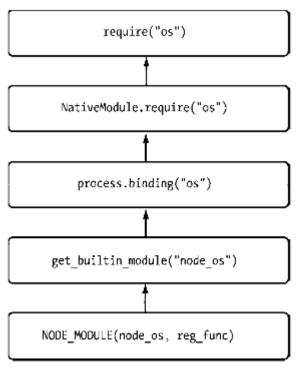


图2-5 os原生模块的引入流程

缓存:核心模块,文件模块。 浏览器仅仅缓存文件,而Node缓存的 是编译和执行之后的对象。

核心模块的引入流程

JavaScript核心模块的编译过程:

Node采用了V8附带的js2c.py工具,将所有内置的JavaScript代码(src/node.js和lib/*.js)转换成C++里的数组,生成node_natives.h头文件,在启动Node进程时,JavaScript代码直接加载进内存中。

对不同文件的处理

Module.js:

```
// Native extension for .js
Module._extensions['.js'] = function(module, filename) {
  var content = NativeModule.require('fs').readFileSync(filename, 'utf8');
  module._compile(stripBOM(content), filename);
// Native extension for .json
Module._extensions['.json'] = function(module, filename) {
  var content = NativeModule.require('fs').readFileSync(filename, 'utf8');
  try {
    module.exports = JSON.parse(stripBOM(content));
  } catch (err) {
    err.message = filename + ': ' + err.message;
    throw err;
//Native extension for .node
Module._extensions['.node'] = process.dlopen;
```

I/0模型

《UNIX网络编程:卷一》第六章——I/O复用。书中向我们提及 了5种类UNIX下可用的I/O模型:

- 1.阻塞式I/O;
- 2.非阻塞式I/O;
- 3.I/O复用(select, poll, epoll...);
- 4.信号驱动式I/O(SIGIO);
- 5.异步I/O(POSIX的aio_系列函数);

Nginx: 异步,非阻塞,使用io复用epoll

apache: 多进程,多线程,每个线程一个请求。

同步,上下文切换,管理。。。

高性能网络库:

libevent: e.g memcached

libev

libuv

阻塞io, 非阻塞io

io操作:

- (1) 等待数据准备好;
- (2) 从内核向进程复制数据。

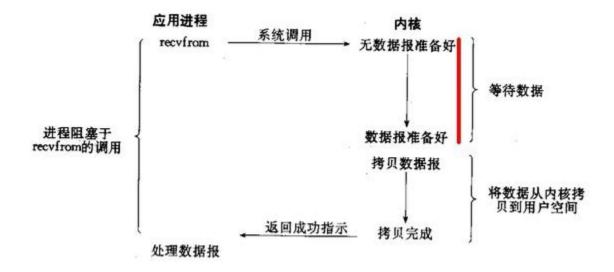


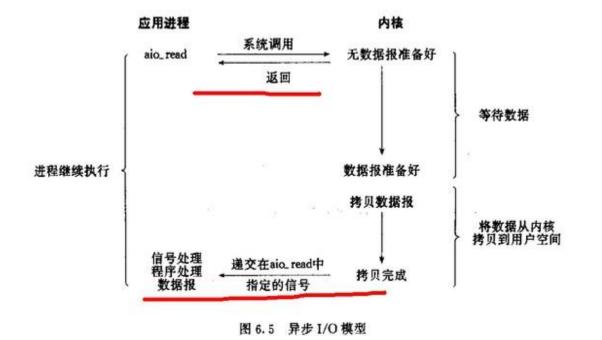
图 6.1 阻塞 I/O 模型



图 6.2 非阻塞 I/O 模型

异步io, 同步io

I/O多路复用:虽然I/O多路复用的函数也是阻塞的,但是其与以上两种还是有不同的,I/O多路复用是阻塞在select,epoll这样的系统调用之上,而没有阻塞在真正的I/O系统调用如recvfrom之上。



同步,异步:访问数据的方式,同步需要主动读写数据,在读写数据的过程中还是会阻塞;异步只需要I/O操作完成的通知,并不主动读写数据,由操作

系统内核完成数据的读写。

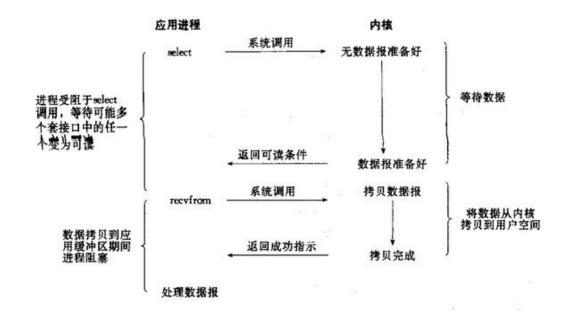
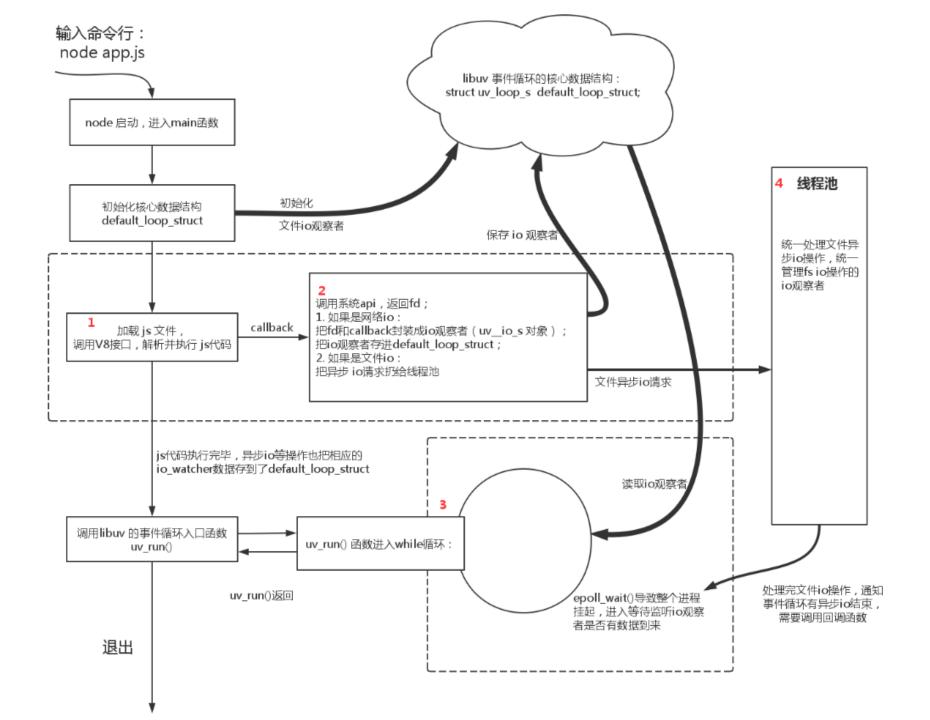


图 6.3 I/O 复用模型

异步I/O: 这类函数的工作机制是告知内核启动某个操作, 并让内核在整个操作(包括将数据从内核拷贝到用户空间)完成后通知我们。

工作流程

timers... fs.readFileSync...



Io_watcher

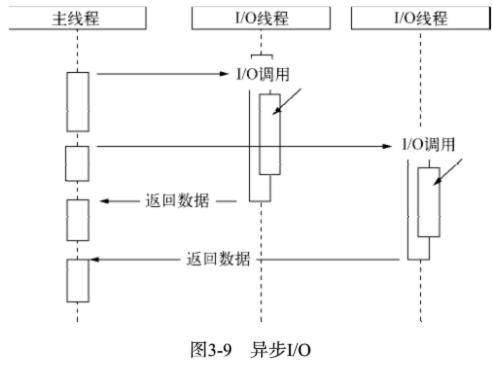
```
typedef struct uv io s uv io t;
struct uv io s {
uv io cb cb;
ngx queue t pending queue;
ngx queue t watcher queue;
unsigned int pevents; /* Pending event mask i.e. mask at next tick. */
unsigned int events; /* Current event mask. */
int fd;
UV IO PRIVATE FIELDS
};
fd: 文件描述符,操作系统对进程监听的网络端口、或者打开文件的一个标记
cb: 回调函数,当相应的io观察者监听的事件被激活之后,被libuv事件循环调用的回
调函数。C++回调函数,里面回调js的回调函数。
events: 交给libuv的事件循环(epoll wait)进行监听的事件
当js代码执行完毕,进入C++域,再进入到uv io poll的时候,就需要这几个步骤:
遍历 loop->watcher queue,取出所有io观察者,这里取出的w就是图6-3-1中调用
uv io start保存的io观察者 —— w。
取出了w之后,调用epoll ctl(),把w->fd(io观察者对应的fd)注册给系统的epoll机制,
那么epoll wait()时就监听对应的fd。
当epoll wait()返回了,拿出有事件到来的fd,这个时候loop->watchers 映射表就起到作
```

用了,通过fd拿出对应的io观察者 —— w,调用w->cb()。

io复用: epoll 线程池

- epoll_create, 创建一个epoll文件描述符
- epoll_ctl,用来添加/修改/删除需要侦听的文件描述符及其事件
- epoll_wait,接收发生在被侦听的描述符上的,用户感兴趣的IO事
- 件,
- 创建线程 线程从队列中取出,执行work().
- 线程中操作完成,通过pipe(),eventfd()通知epoll。
- epoll监听到事件,执行回调。js的回调函数都是在主线程中完成。
- 等待数据, 拷贝数据都是在线程池中完成。模拟了异步。

node异步的实现



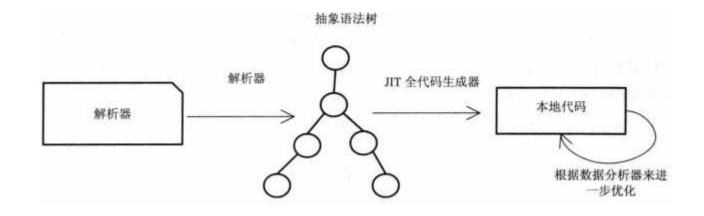
glibc的AIO便是典型的线程池模拟异步I/O Windows的IOCP:

理想的异步I/O:调用异步方法,等待I/O完成之后的通知,执行回调,用户无须考虑轮询。但是它的内部其实仍然是线程池原理,不同之处在于这些线程池由系统内核接手管理。

IOCP的异步I/O模型与Node的异步调用模型十分近似。在Windows平台下采用了IOCP实现异步I/O。

V8

V8采用JIT——即时编译技术,直接将js代码编译成本地平台的机器码。



Gc, memory,

js c++:

C++通过v8的Object类和FunctionTemplate类,创建对象、方法,设置属性、原型方法等,提供给运行时的 js代码调用。

MakeCallback(): 执行js异步回调函数时,从C++域陷入js域的函数

参考:

- 朴灵 深入浅出nodejs
- https://cnodejs.org/user/bigtree9307
- https://cnodejs.org/user/LanceHBZhang