# NodeJS如何能够构建高性能网络服务器

##### Nodejs简介

NodeJS是一个JS的服务端运行平台,基于google的v8引擎,该引擎对js进行了一些封装,以解决原生的js对一些操作的支持不足,并且采用JIT的编译技术使得Nodejs编写出来的代码性能很高,Nodejs由于基于javascript,所以是单线程的,事件驱动模型,它的I/O操作在底层调用了一个基于c的库(Libeio),该库是一个功能强大,高性能的异步I/O库.跟面向对象的编程风格不同,Nodejs采用按模块划分来组织一个功能单元集合.

Nodejs发展非常迅猛,由于是开源并且基于javascript的,使得大量js爱好者投身其中,第三方的模块增长速度惊人,从2012年-2013年,其第三方模块增加了3万多,截止目前,模块数量



官方对它的作用定义是可以用来构建高性能的网络服务器.

##### Nodejs核心API,模块简介

1,全局对象

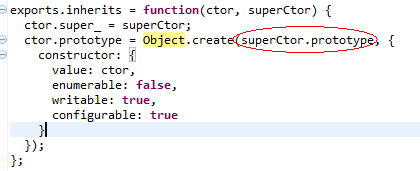
在js中全局对象是window,在Nodejs中叫做global,所有全局变量,除了global自身,都是global对象的属性.

2,工具集

主要是2个,熟悉javascript的都应该猜得到,有2个东西在扩展js的时候是必须考虑的,

一个是继承,一个是对象-字符串的转换

Nodejs提供了util.inherits函数来进行实现继承,简单看下源码

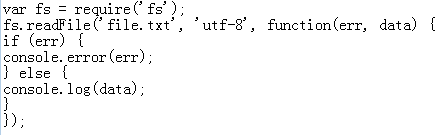


通过图中画出的部分可以看出,Nodejs的继承采用标准的原型继承(ECMAScript新版规范),子类只会继承父类原型上的属性,由于没有使用对象冒充,无法继承父类构造定义的属性.

util.inspect函数可以把一个对象,转换成其对应的JSON,可替代js的eval函数.

3,文件系统

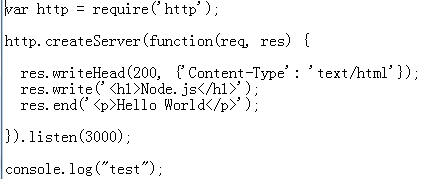
Nodejs的文件操作主要由fs模块封装,这里需要注意的是,此模块不止有异步的操作,还提供了同步的操作,同步的操作会阻塞主程序的运行,简单看一下Nodejs的调用风格:

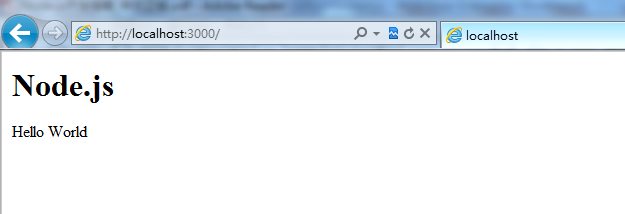


看起来很清爽,只需要提供一个路径,一个可选的编码参数,一个回调函数

4,HTTP模块

Nodejs提供了一个http server的基本实现,主要操作对象是http.Server,它提供了一套底层的API,仅仅是流控制和简单的消息解析,所有高层功能都需要通过它的基础上来实现.实现代码如下:





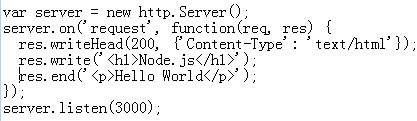
这里注意一个细节, console.log("test");会直接在后台输出,http.server是一个基于事件的服务器,所有的请求都会被封装成独立的事件,开发者只需要对相应的事件编写相应的回调函数,就可以完成所有的HTTP服务器功能,这里基本的事件有以下3个

request 客户端请求到来触发

connection TCP连接建立触发

close 当服务器关闭时触发

http.createServer其实是一个request监听写法上的简化形式,相当于这样的实现



Request事件发送的第一个参数 req是请求信息对象,它提供3个事件来控制http请求体的传输

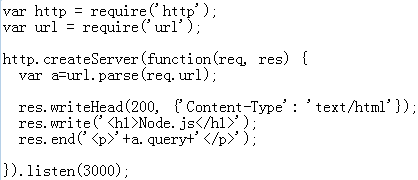
data当请求体到来时触发

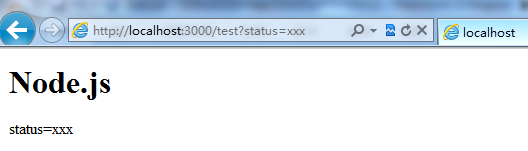
end当请求体传输完成时触发

close当用户当前请求终止时触发

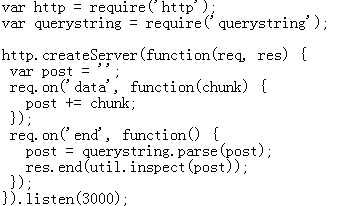
Nodejs对于get和post请求要分开来处理

get请求,利用url.parse可以解析访问路径



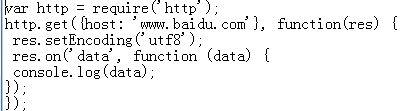


post相对较麻烦,post的内容都在请求体中



可以看到需要利用req提供的事件来进行post请求的解析

Nodejs的客户端提供了http.request和http.get来向服务端发起请求,后者是前者的简化版,它自动将请求方法设为get,并且不用手动调用req.end方法



回调里的res对象跟上面的req对象一样,也是提供三个事件来表示,数据到达,传输结束和连接终止

5,Socket.io

Sokcet.io是Nodejs的一个亮点所在,它提供了一套完整的实时通信封装以解决各个浏览器兼容性的问题,并简化开发,很轻松的解决了服务端-浏览器的推送问题.

利用Nodejs可以很轻松开发出兼容所有浏览器的webIM应用,下面是一个Express结合Socket的简单示例

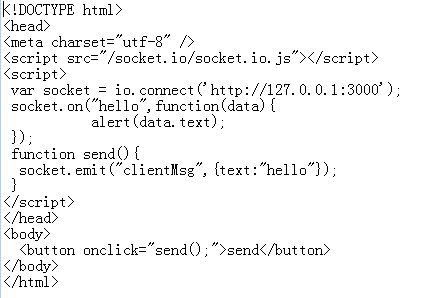
服务端代码如下:

(省略部分代码,只给出关键代码)



服务端监听连接后获得一个客户端的socket对象,用此对象做事件监听即可,以上代码,监听来自客户端的clientMsg事件,打印信息以后发送一个hello事件到客户端,内容为一个JSON对象

客户端代码如下:



需要引入一个模块下的js文件,然后连接服务端,之后开启监听,以上监听来自服务端的hello事件打印服务端发来的信息,然后用一个按钮来向服务端发送clientMsg事件,并发送信息.

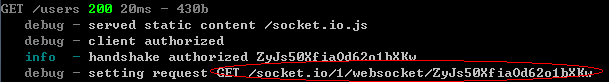
效果如下:



后台打印

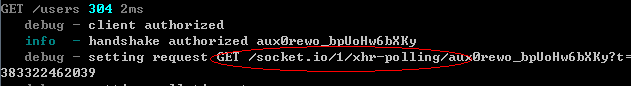
从控制台加载信息可以看出Socket.io解决浏览器兼容问题的方案

Chrome后台输出如下:



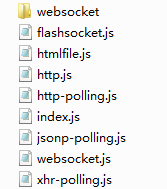
可以看出对于chrome使用了websocket

IE 9输出如下:



对于IE9使用了ajax轮询

从Socket.io源码的文件中可以找到Nodejs会采取的方法



主要是以下几种:

Web Socket

Adobe Flash Socket

http长连接

http 轮询

Ajax多部轮询

Jsonp轮询

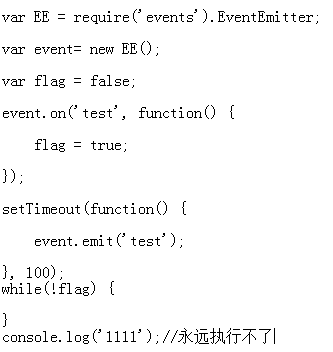
正是因为Nodejs提供了这些封装实现,大大简化了开发流程,开发人员可以专注开发,不用担心兼容问题.

##### Nodejs事件轮询原理

由于采用事件驱动的设计模型,javascript天生就是一个事件驱动的语言,比较容易适合来做异步的编程,其次,NodeJS使用了一个事件循环的架构来实现并行操作,使用这样的架构首先少了多线程的开销,其次可以屏蔽掉一些并发编程的复杂点和陷阱.

NodeJS是一个单线程的语言,缺少并行性,主程序将事件的回调压入事件队列等待处理,在事件队列中,处理的方式也是串行的,这样的机制可以使一个请求被接收后,只需要把这个请求的处理回调函数压入队列,主程序就可以继续往下执行,不需要阻塞等待,但是如果当事件队列中某个事件的处理无限慢时,将会导致NodeJS主线程阻塞,无法向下执行

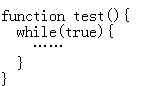
一个最直观的的例子:

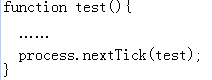


可以看到,主程序一直在等待事件回调结束完毕

而对于IO操作,异步IO操作是底层完成的,当你调用一个类似readFile之后只是相当于将请求发给了操作系统,然后就立即返回执行下面的语句,当这个IO完成以后底层的libeio会发送一个事件到事件队列,然后通过事件循环,该事件被触发,执行回调函数,Nodejs不需要轮询事件队列,当IO请求发送的时候,会把主线程的一个回调也发送过去,当操作系统完成IO后由libeio来调用主线程的这个回调,这个符合观察者设计模式,所以Nodejs的IO操作省去了轮询事件队列的代价,IO操作耗时以否都不影响主线程的执行,只有当回调函数在执行耗时计算的时候,会导致主线程阻塞.

对此,NodeJS提供了一个解决方案,使用process.nextTick()这个函数来解决一些需要持续运行的会导致主程序阻塞的逻辑, process.nextTick可以让当前的逻辑在下一个事件轮循的时间点上执行,这样可以避免如下的代码:

可以改写为:



这样就可以使test和正常的事件响应交叉执行,避免了阻塞整个主程序,这只是NodeJS提出的一个解决执行cpu运算密集行的任务的方案,并不能通过它来获得真正的多cpu下并行执行的好处,在真正做的时候还要根据具体业务需求,把一个很长的流程切分来处理.

##### Nodejs利用多核CPU编程

NodeJS在多核cpu下可以使用多进程的方式来按照cpu的数量扩展应用程序

可以利用多进程的方式来实现负载均衡

一般有如下几种方法

多进程绑定在同一个端口监听：

可以用NodeJS自带的集群模块来实现

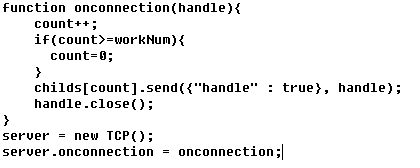


这种情况下负载均衡是由操作系统来分配的，但是这种方法，这个Cluster模块还有很多问题，这种方法实现的负载均衡稳定性也有问题，所以又有大牛想了一些其他的解决方案

每个进程绑定不同端口，用反向代理服务器来做负载均衡（我对这个不是很了解，大意就是稳定性，功能都能有保证，缺点是引入了一个间接层）

还有一种方法就是，一个线程负责接收，监听连接，然后把收到的连接平均分配到其他子进程

这种方法需要在tcp包装前就把tcp连接分配下去



缺点就是引如了额外的开销，而且不能与主流的web框架结合

以上除了使用cluster模块都是一些比较基础的实现，正常开发中还是用cluster的居多，cluster可以应对在使用短连接，并发非常高的情况.

总结：nodejs使用自己底层库实现的异步非阻塞IO来解决i/o密集型的任务，nodejs使用多进程的方式来应对在多核cpu下的程序扩展，但是方法五花八门，第三方模块更是多如牛毛，目前进行web开发比较主流的有Express(轻量级http server框架),html模板引擎,jade,esb等等,都是有利有弊，具体要结合实际情况来选择，虽然有多种优化手段，但是任然不能避免nodejs解决cpu密集型任务的弊端，正是由于这些原因，所以导致其编程思想的转变，java可能考虑的是这个io完成后我该做什么，而nodejs 确需要考虑在进行这个io的时候我还能做什么，但是其确实在编程的简便性上做到了很好(比如Socket.io提供的web客户端解决方案),屏蔽了并发编程的一些复杂性，比如，竞争，可见性，中断，可伸缩性等一些java并发编程的陷阱，学习成本也不是很高，但是当需要使用它在专有场合，能够有很好的表现的话，还是需要一定的掌握深度，以及带来另外的一些复杂性.