

Node.js是单线程应用程序,但是通过事件和回调支持并发,所以性能非常高

Node.js的每一个API都是异步的,并作为一个独立线程运行使用异步函数

Node.js的每一个API都是异步的,并作为一个独立线程运行,使用异步函数调用,并处理并发

Node.js基本上所有的事件机制都是用设计模式模式中观察者模式实现.

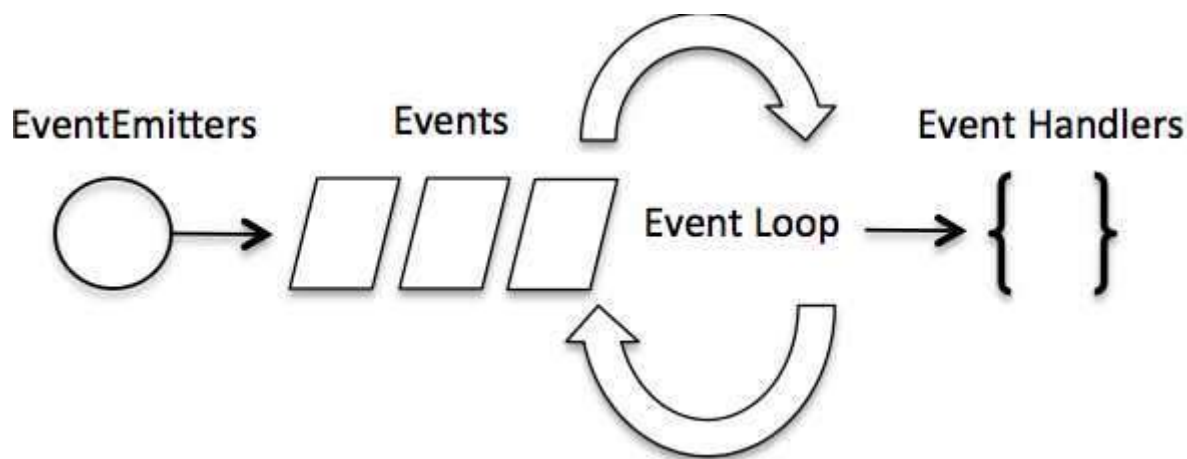
Node.js单线程类似进入一个While(true)的事件循环,直到没有事件观察者退出,每个异步事件都生成一个事件观察者,如果有事件发生就调用该回调函数

事件驱动程序:

Node.js使用事件驱动模型,当webserver接收到请求,就把它关闭然后进行处理,然后去服务下一个web请求.当这个请求完成,它被放回处理队列,当到达队列开头,这个结果被返回给用户

这个模型非常高效可扩展性非常强,因为webserver一直接收请求而不等待任何读写操作(非阻塞式IO或者事件驱动IO)

在事件驱动模型中,会生成一个主循环来监听事件,当检测到事件时触发回调函数.



意思就好像：购票，如果柜台买票，每个人都是一个事件，柜台在处理当前事件时阻塞后面的每一个事件，当时当你网上购票的时候，也许同时会有一个或者多个人同时在买票，但是你们都不会因为彼此的操作而影响到对方，就称非阻塞式IO或者事件驱动IO。

Node.js内置事件：我们可以通过引入events模块，并通过实例化EventEmitter类来绑定和监听事件，如：

```
var events = require('event');  
var EventEmitter = new events.EventEmitter();
```

程序绑定事件处理程序

```
eventEmitter.on('eventName', eventHandler);
```

通过程序触发事件：

```
eventEmitter.emit('eventName');
```

main.js

```
// 引入 events 模块  
var events = require('events');  
// 创建 EventEmitter 对象  
var EventEmitter = new events.EventEmitter();  
  
// 创建事件处理程序  
var connectHandler = function connected() {  
    console.log('连接成功。');  
  
    // 触发 data_received 事件  
    EventEmitter.emit('data_received');  
}  
  
// 绑定 connection 事件处理程序  
EventEmitter.on('connection', connectHandler);  
  
// 使用匿名函数绑定 data_received 事件  
EventEmitter.on('data_received', function(){  
    console.log('数据接收成功。');  
});
```

```
// 触发 connection 事件
eventEmitter.emit('connection');
```

```
console.log("程序执行完毕。");
```

执行后如下图:



```
C:\Users\Node>node main.js
连接成功。
数据接收成功。
程序执行完毕。
```

Node.js所有的异步I/O操作再完成时都会发送一个事件到事件队列.

Node.js里面的许多对象都会分发事件:一个net.Server对象会在每次有新链接时分发一个事件,一个fs.readStream对象会在文件被打开的时候发出一个事件.所有这些产生事件的对象都是

events.EventEmitter的实例

EventEmitter类:events模块只提供了一个对

象:events.EventEmitter的核心就是事件触发与事件监听功能的封装.可以通过require("events")来访问该模块

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

```
var eventEmitter = new events.EventEmitter();
```

EventEmitter 对象如果在实例化时发生错误,会触发error事件.

当添加新的监听器时,newListener事件会触发,当监听器被移除时,removeListener事件被触发

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

```
var event = new EventEmitter();
```

```
event.on('some_event', function() {
```

```
    console.log('some_event 事件触发');
```

```
});
```

```
setTimeout(function() {
```

```
    console.log('hello,我被触发啦');
```

```
    event.emit('some_event');
```

```
}, 1000);
```

如下图：

```
C:\Users\Node>node event.js  
hello, 我被触发啦  
some_event 事件触发
```

注册事件：`event.on("事件名称", 回调函数)`，上述代码`event`对象注册了事件`some_event`的一个监听器，通过`setTimeout`在1000毫秒以后向`event`对象发送事件`some_event`，此时会调用`some_event`的监听器

`EventEmitter`的每个事件由一个事件名和若干个参数组成，事件名是一个字符串，通常表达一定的语义。对于每个事件，`EventEmittr`支持若干个事件监听器。

当事件触发时，注册这个事件的监听器被一次调用，事件参数作为回调参数传递。

注册两个监听事件，采用事件参数作为回调函数参数传递

`event.js`

```
var events = require('events');  
var emitter = new events.EventEmitter();  
emitter.on('someEvent', function(arg1, arg2) {  
    console.log('listener1', arg1, arg2);  
});  
emitter.on('someEvent', function(arg1, arg2) {  
    console.log('listener2', arg1, arg2);  
});  
emitter.emit('someEvent', 'arg1 参数', 'arg2 参数');
```

如下图：

```
C:\Users\Node>node eventCS.js  
listener1 arg1参数 arg2参数  
listener2 arg1参数 arg2参数
```

上述例子中，`emitter`为`someEvent`注册了两个事件监听器，然后触发了`someEvent`事件。运行后，可以看到两个事件监听器回调函数被先后调用

EventEmitter提供了多个属性,如on和emit, on用于绑定事件函数, emit属性用于触发一个事件.EventEmitter的属性

1.

on(event, listener):为指定事件注册一个监听器,接收一个字符串event和一个回调函数.如下:

```
emitter.on('eventName',function(){  
  console.log('message');  
});
```

2.emit(event, [arg1], [arg2], [...])

按参数的顺序执行每个监听器,如果事件有注册监听返回true,否则false

3.listeners(event):返回指定事件的监听器组数.

4.listenerCount(emitter,event) 或

listenerCount(event):返回指定事件的监听器数量.

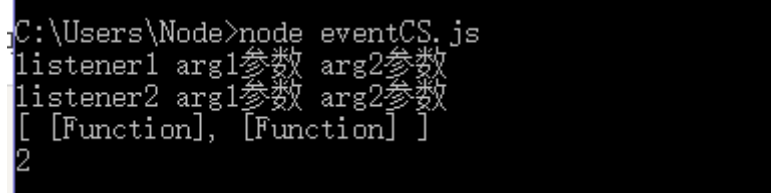
案例:

eventCS.js

```
var events = require('events');  
var emitter =new  events.EventEmitter();  
emitter.on('someEvent',function (arg1,arg2) {  
  console.log('listener1',arg1,arg2);  
});  
emitter.on('someEvent',function (arg1,arg2) {  
  console.log('listener2',arg1,arg2);  
  
});  
emitter.emit('someEvent','arg1参数','arg2参数');  
console.log(emitter.listeners('someEvent'));
```

```
console.log(emitter.listenerCount('someEvent'));
```

如下图:



```
C:\Users\Node>node eventCS.js
listener1 arg1参数 arg2参数
listener2 arg1参数 arg2参数
[ [Function], [Function] ]
2
```

从图中可以看出监听器是回调函数, 数量为2

5.addListener(event, listener): 为指定事件添加一个监听器到监听器组数的尾部

案例:

event.js

```
var events = require('events');
var emitter = new events.EventEmitter();
emitter.on('someEvent', function (arg1, arg2) {
    console.log('listener1', arg1, arg2);
});
emitter.on('someEvent', function (arg1, arg2) {
    console.log('listener2', arg1, arg2);
});
emitter.addListener('someEvent', function
(arg1, arg2) {
    console.log('listener3', arg1, arg2);
});
emitter.emit('someEvent', 'arg1参数', 'arg2参数');
console.log(emitter.listeners('someEvent'));
console.log(emitter.listenerCount('someEvent'));
```

如下图:

```
C:\Users\Node>node eventCS.js
listener1 arg1参数 arg2参数
listener2 arg1参数 arg2参数
listener3 arg1参数 arg2参数
[ [Function], [Function], [Function] ]
3
```

可以看到, 监听器被添加到了原先的监听器数组的尾部, 如果把
`emitter.addListener('someEvent', function`
`(arg1, arg2) {`
`console.log('listener3', arg1, arg2);`放在第二个监
听器之前, 那么输出如下图:

```
C:\Users\Node>node eventCS.js
listener1 arg1参数 arg2参数
listener3 arg1参数 arg2参数
listener2 arg1参数 arg2参数
[ [Function], [Function], [Function] ]
3
```

发现: '尾部' 是指目前的从上到下执行的尾部, 而不是程序执行后的尾部

6. `once(event, listener)`: 为指定事件注册一个单词监听器, 即监听器最多只会触发一次, 触发后立即接触该监听器.

```
emitter.once(event, listener);
```

7. `removeListener(event, listener)`: 移除指定事件的某个监听器, 监听器必须是该事件已经注册过的监听器, 接收两个参数, 第一个事件名称, 第二个回调函数名称

案例: `removeListener.js`

```
var events = require('events');
var emitter = new events.EventEmitter();

var callback = function() {
```

```

console.log('监听上了!!');

};

emitter.on('someEvent',callback);
emitter.removeListener('someEvent',callback);

emitter.on('someEvent',function (arg1,arg2) {
    console.log('listener2',arg1,arg2);

});

emitter.emit('someEvent','arg1参数','arg2参数');
console.log(emitter.listeners('someEvent'));
console.log(emitter.listenerCount('someEvent'));

```

如下图：



```

C:\Users\Node>node removeListener.js
listener2 arg1参数 arg2参数
[ [Function] ]
1

```

从图中可以看到，监听器个数只剩下一个了，上述代码中注册了两个，但是第一个被移除了，所以就只剩下一个了

8.removeAllListeners([even])：移除所有事件的的所有监听器，如果指定事件，则移除指定事件的所有监听器。

案例：

removeAllListener.js

```

var events = require('events');
var emitter =new  events.EventEmitter();
emitter.on('someEvent',function (arg1,arg2) {
    console.log('listener2',arg1,arg2);

```



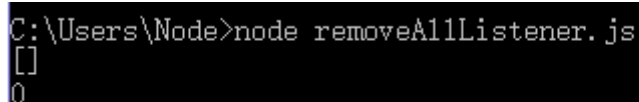
```

});
emitter.on('someEvent2',function (arg1,arg2) {
    console.log('listener2',arg1,arg2);

});
emitter.removeAllListeners();
emitter.emit('someEvent','arg1参数','arg2参数');
emitter.emit('someEvent2','arg1参数','arg2参数');
console.log(emitter.listeners('someEvent'));
console.log(emitter.listenerCount('someEvent'));

```

如下图:



```

C:\Users\Node>node removeAllListener.js
[]
0

```

如果只是想删除某个事件的所有监听器,则可以指定删除的事件名称,如把上述的:

`emitter.removeAllListeners();` 改为:

`emitter.removeAllListeners('someEvent');`

输出如下图:



```

C:\Users\Node>node removeAllListener.js
listener2 arg1参数 arg2参数
[]
0

```

从图中可以看到`someEvent`事件的所有监听器已经被删除为0个,而`someEvent2`的监听器还存在.

9. `setMaxListeners(n)`: 默认情况下, `EventEmitters` 如果你添加的监听器超过10个就会输出警告信息. `setMaxListeners` 函数用于提高监听器的默认限制的数量.

案例: `setMaxListeners.js`

```

var events = require('events');

```

```

var emitter =new  events.EventEmitter();
emitter.setMaxListeners(2);
emitter.on('someEvent',function (arg1,arg2) {
    console.log('listener2',arg1,arg2);

});
emitter.on('someEvent',function (arg1,arg2) {
    console.log('listener2',arg1,arg2);

});
emitter.on('someEvent',function (arg1,arg2) {
    console.log('listener2',arg1,arg2);

});
emitter.on('someEvent',function (arg1,arg2) {
    console.log('listener2',arg1,arg2);

});

emitter.emit('someEvent','arg1参数','arg2参数');
console.log(emitter.listeners('someEvent'));
console.log(emitter.listenerCount('someEvent'));

```

输出如下图：



```

C:\Users\Node>node setMaxListeners.js
listener2 arg1参数 arg2参数
listener2 arg1参数 arg2参数
listener2 arg1参数 arg2参数
listener2 arg1参数 arg2参数
[ [Function], [Function], [Function], [Function] ]
4
(node:13360) MaxListenersExceededWarning: Possible EventEmitter memory leak detected. 3 someEvent listeners added. Use emitter.setMaxListeners() to increase limit

```

从图中可以看出，程序会输出所有注册的监听器，如果超过，就会给出警告，**注意**：设置默认的监听器个数需要在你注册监听器之前就设置

好,不然是不会起作用的,监听器的限制个数还是默认的10个

10. (事件) `newListener`: 该事件在添加监听器时被触发

11. (事件) `removeListener`: 从指定监听器数组中删除一个监听器, 此操作将会改变被删监听器之后的那些监听器的索引.

12. `error: EventEmitter` 定义了一个特殊的事件 `error`, 它包含了错误的语义, 当我们遇到异常的时候通常会触发 `error` 事件. 当 `error` 被触发时, `EventEmitter` 规定如果没有响应的监听器, `Node.js` 会把它当作异常, 退出程序并输出错误信息. 一般需要为会触发 `error` 事件的对象设置监听器, 避免遇到后整个正序崩溃.

继承 `EventEmitter`

大多数时候我们不会直接使用 `EventEmitter`, 而是在对象中继承它. 包括 `fs`, `net`, `http` 在内的. 只要支持事件响应的核心模块都是 `EventEmitter` 的子类. 原因: 首先, 具有某个实体功能的对象实现事件符合语义, 事件的监听和发生应该是一个对象的方法, 其次 `JavaScript` 的对象机制是基于原型的, 支持部分多重继承, 继承 `EventEmitter` 不会打乱对象原有的继承关系.

Node 应用程序的工程原理:

`t.js`

```
var fs = require("fs");
```

```
fs.readFile('t.txt', function (err, data) {  
  if (err) {  
    console.log(err.stack);  
  }  
})
```

```
        return;
    }
    console.log(data.toString());
});
console.log("程序执行完毕");
t.tx
你好,我叫马达斯佳
```

执行后输出:



```
C:\Users\Node>node t.js
程序执行完毕
你好,我叫马达斯佳
```

从图中可以看到程序是异步执行的, `fs.readFile()` 是异步函数用于读取文件. 如果读取文件过程中发生错误, 错误 `err` 对象就会输出错误信息.

如果没有发生错误, `readFile` 跳过 `err` 对象的输出, 文件内容就通过回调函数输出. 如果不是异步执行, 那么"程序执行完毕"应该是在 `t.txt` 内容输出完毕后会才会执行, 是按照从上到下的步骤去运行的, 如果程序中途出错, 那么下边的程序是无法继续进行的, 就像柜台买票, 如果此时第一个人办理的事务特别多, 特别繁琐 (如外国人买中国火车票), 那么就无法再进行后面的事务处理. 就会中断全部. 但是如果你网上购票, 另一方操作太慢, 但是这并不会影响到你正常买票

