第五章：使用事件发射器模式简化事件绑定

本章内容：

- 事件发射器模式（Event Emitter Pattern）介绍
- 事件监听器的绑定和取消
- 创建自己的事件发射器

在 Node 里，很多对象都会发射事件。比如，一个 TCP 服务器，每当有客户端请求连接就会发射“connect”事件，又比如，每当读取一整块数据，文件系统就会发射一个“data”事件。这些对象在 Node 里被称为事件发射器（event emitter）。事件发射器允许程序员订阅他们感兴趣的事件，并将回调函数绑定到相关的事件上，这样每当事件发射器发射事件时回调函数就会被调用。发布/订阅模式非常类似传统的 GUI 模式，比如按钮被点击时程序就会收到相应的通知。使用这种模式，服务端程序可以在一些事件发生时作出反应，比如有客户端连接，socket 上有可用数据，或者文件被关闭的时候。

还可以创建自己的事件发射器，事实上，Node 专门提供了一个 EventEmitter 伪类，可以把它当作基类来创建自己的事件发射器。

理解回调模式

异步编程不使用函数返回值来表明函数调用的结束，而是采用后继传递风格。

> *“后继传递风格”（CPS：Continuation-passing style）是一种编程风格，流程控制被显式传递给下一步操作……*
>
> *CPS 风格的函数会接受一个函数作为额外参数，这个函数用来显式指出程序控制的下个流程，当 CPS 函数计算出它的“返回值”，它就会调用那个代表了程序下个流程的函数，并将 CPS 函数的“返回值”作为其参数。*
>
> *出自维基百科——http://en.wikipedia.org/wiki/Continuation-passing_style*

这种编程风格里，每个函数在执行结束后都会调用一个回调函数，这样程序就可以继续运行。后面你会明白，JavaScript 非常适合这种编程风格，下面是个 Node 下将文件加载到内存的例子：

```
var fs = require('fs');
fs.readFile('/etc/passwd', function(err, fileContent) {
    if (err) {
        throw err;
    }
    console.log('file content', fileContent.toString());
});
```

这个例子里，你传递了一个内联匿名函数作为 fs.readFile 的第二个参数，其实这就是在使用 CPS 编程，因为你把程序执行的后续流程交给了那个回调函数。

如你所见，回调函数的第一个参数是个错误对象，如果程序发生错误，这个参数将会是

一个 Error 类的实例，这是 Node 里 CPS 编程的一个常见模式。

理解事件发射器模式

标准回调模式里，把一个函数作为参数传递给将被执行的函数，这种模式在客户端需要在函数完成后被通知的场景下工作的很好。但是如果函数的执行过程中发生了多个事件或事件重复发生了多次，这种模式就不太适合了。比如，你想在 socket 每次收到可用数据时得到通知，这种场景你会发现标准回调模式不太好用，这时事件发射器模式就派上用场了，你可以用一套标准接口来清晰的分离事件发生器和事件监听器。

使用事件发生器模式时，会涉及到两个或多个对象——事件发射器和一个或多个事件监听器。

事件发射器，顾名思义，是个可以产生事件的对象。而事件监听器则是绑定到事件发射器上的代码，用来监听特定类型的事件，就像下面的例子：

```
var req = http.request(options, function(response) {
    response.on("data", function(data) {
        console.log("some data from the response", data);
    });
    response.on("end", function() {
        console.log("response ended");
    });
});
req.end();
```

这段代码演示了用 Node 的 http.request API（见后面章节）创建一个 HTTP 请求来访问远程 HTTP 服务器时的两个必要步骤。第一行采用了“后继传递风格”（CPS：Continuation-passing style），传递了一个当 HTTP 响应时会被调用的内联函数。HTTP 请求 API 在这儿使用 CPS 是因为程序需要在 http.request 函数执行完毕后才继续执行后续操作。

当 http.request 执行完毕，就会调用那个匿名回调函数，然后将 HTTP 响应对象作为参数传递给它，这个 HTTP 响应对象是个事件发射器，根据 Node 文档，它可以发射包括 data，end 在内的很多事件，你注册的那些回调函数会在每次事件发生时被调用。

作为一条经验，当你需要在请求的操作完成后重新获取执行权时使用 CPS 模式，以及当事件可以发生多次时使用事件发射器模式。

理解事件类型

被发射的事件都有一个用字符串表示的类型，前面的例子包含“data”和“end”两个事件类型，它们是由事件发射器来定义的任意字符串，不过约定俗成的是，事件类型通常都由不包含空字符的小写单词组成。

不能用代码来推断出事件发射器能产生哪些类型的事件，因为事件发射器 API 并没有内省机制，因此你使用的 API 应该有文档来表明它能发射那些类型的事件。

一旦事件发生，事件发射器就会调用跟事件相关的监听器，并将相关数据作为参数传递给监听器。在前面 http.request 那个例子里，“data”事件回调函数接受一个 data 对象作为它第一个也是唯一的参数，而“end”不接受任何数据，这些参数作为 API 契约的一部分也是由 API 的作者主观定义的，这些回调函数的参数签名也会在每个事件发射器的 API 文档里

有说明。

事件发射器虽然是个为所有类型事件服务的接口，不过“error”事件是 Node 里的一个特殊实现。Node 里的大多数事件发射器都会在程序发生错误时产生“error”事件，如果程序没有监听某个事件发射器的 “error”事件，事件发射器将会注意到并在错误发生时向上抛出一个未捕获异常。

你可以在 Node PERL 里运行下面的代码来测试下效果，它模拟了一个能产生两种事件的事件发射器：

```
var em = new (require('events').EventEmitter)();
em.emit('event1');
em.emit('error', new Error('My mistake'));
```

你将会看到下面的输出：

```
var em = new (require('events').EventEmitter)();
undefined
> em.emit('event1');
false
> em.emit('error', new Error('My mistake'));
Error: My mistake
at repl:1:18
at REPLServer.eval (repl.js:80:21)
at repl.js:190:20
at REPLServer.eval (repl.js:87:5)
at Interface.<anonymous> (repl.js:182:12)
at Interface.emit (events.js:67:17)
at Interface._onLine (readline.js:162:10)
at Interface._line (readline.js:426:8)
at Interface._ttyWrite (readline.js:603:14)
at ReadStream.<anonymous> (readline.js:82:12)
>
```

代码第 2 行，随便发射了一个叫“event1”的事件，没有任何效果，但是当发射“error”事件时，错误被抛出到堆栈。如果程序不是运行在 PERL 命令行环境里，程序将会因为未捕获的异常而崩溃。

使用事件发射器 API

任何实现了事件发射器模式的对象（比如 TCP Socket，HTTP 请求等）都实现了下面的一组方法：

- .addListener 和.on —— 为指定类型的事件添加事件监听器
- .once —— 为指定类型的事件绑定一个仅执行一次的事件监听器
- .removeEventListener —— 删除绑定到指定事件上的某个监听器
- .removeAllEventListener —— 删除绑定到指定事件上的所有监听器

下面我们具体介绍它们。

使用.addListener()或.on()绑定回调函数

通过指定事件类型和回调函数，你可以注册当事件发生时被执行的操作。比如，文件读取数据流时如果有可用的数据块，就会发射一个“data”事件，下面代码展示如何通过传入一个回调函数来让程序告诉你发生了 data 事件。

```
function receiveData(data){
    console.log("got data from file read stream: %j", data);
}
readStream.addListener(“data”, receiveData);
```

你也可以使用.on，它只是.addListener 的简写方式，下面的代码和上面的是一样的：

```
function receiveData(data){
    console.log("got data from file read stream: %j", data);
}
readStream.on(“data”, receiveData);
```

前面代码，使用事先定义的一个的命名函数作为回调函数，你也可以使用一个内联匿名函数来简化代码：

```
readStream.on("data", function(data) {
    console.log("got data from file read stream: %j", data);
});
```

前面说过，传递给回调函数的参数个数和签名依赖于具体的事件发射器对象和事件类型，它们并不是被标准化的，“data”事件可能传递的是一个数据缓冲对象，“error”事件传递一个错误对象，数据流的“end”事件不向事件监听器传递任何数据。

绑定多个事件监听器

事件发射器模式允许多个事件监听器监听同一个事件发射器的同一事件类型，比如:

```
readStream.on("data", function(data) {
    console.log('I have some data here.');
});
readStream.on("data", function(data) {
    console.log('I have some data here too.');
});
```

这个例子里，readStream的“data”事件类型上绑定了两个函数，每当readStream对象发射“data”事件，你就会看到下面的输出：

```
I have some data here.
I have some data here too.
```

事件发射器负责按监听器的注册顺序调用指定事件类型上绑定的所有监听器，也就是说：

- 当事件发生后事件监听器可能不会被立刻调用，也许会有其它事件监听器在它之前被调用。
- 异常被抛出到堆栈是不正常的行为，可能是因为代码里有bug，当事件被发射时，如果有一个事件监听器在被调用时抛出了异常，可能会导致一些事件监听器永远不会被调用。这种情况下，事件发射器会捕获到异常，也许还会处理它。

看下面这个例子：

```
readStream.on("data", function(data) {
    throw new Error("Something wrong has happened");
});
readStream.on("data", function(data) {
    console.log('I have some data here too.');
});
```

因为第一个监听器抛出了异常，因此第二个监听器不会被调用。

用.removeListener()从事件发射器移除一个事件监听器

如果当你不再关心一个对象的某个事件时，你可以通过指定事件类型和回调函数来取消已注册的事件监听器，像这样：

```
function receiveData(data){
    console.log("got data from file read stream: %j", data);
}
readStream.on("data", receiveData);
// ...
readStream.removeListener("data", receiveData);
```

这个例子里，最后一行把一个可能在将来被随时调用的事件监听器从事件发射器对象移除了。

为了删除监听器，你必须给回调函数命名，因为在添加和删除的时候需要回调函数的名字。

使用.once()让回调函数最多执行一次

如果你想监听一个最多执行一次的事件，或者只对某个事件发生的第一次感兴趣，可以用.once()函数：

```
function receiveData(data){
    console.log("got data from file read stream: %j", data);
}
readStream.once("data", receiveData);
```

上面的代码，receiveData函数只会被调用一次。如果readStream对象发射了data事件，receiveData回调函数将会而且仅会被触发一次。

它其实只是个方便方法，因为很简单的就能实现它，像这样：

```
var EventEmitter = require("events").EventEmitter;
EventEmitter.prototype.once = function(type, callback) {
    var that = this;
    this.on(type, function listener() {
        that.removeListener(type, listener);
        callback.apply(that, arguments);
    });
};
```

上面代码里，你重新定了EventEmitter.prototype.once函数，同时也重定义了每个继承自

EventEmitter的所有对象的once函数。代码只是简单的使用.on()方法，一旦收到了事件，就用.removeEventListener()取消回调函数的注册，并调用原来的回调函数。

注意：前面代码里使用了function.apply()方法，它接受一个对象并把它作为内含的this变量，以及一个参数数组。前面例子里，通过事件发射器把未修改过的参数数组透明地传递给回调函数。

用.removeAllListeners()从事件发射器移除所有事件监听器

你可以像下面那样从事件发射器移除所有注册到指定事件类型上的所有监听器：

```
emitter.removeAllListeners(type);
```

比如，你可以这样取消所有进程中断信号的监听器：

```
process.removeAllListeners("SIGTERM");
```

注意：作为一条经验，推荐你只在确切知道删除了什么内容时才使用这个函数，否则，应该让应用程序其它部分来删除事件监听器集合，或者也可以让程序的那些部分自己负责移除监听器。但不管怎样，在某些罕见的场景下，这个函数还是很有用的，比如当你准备有序的关闭一个事件发射器或者关闭整个进程的时候。

创建事件发射器

事件发射器用一个很棒的方式让编程接口变得更通用，在一个常见易懂的编程模式里，客户端直接调用各种函数，而在事件发射器模式中，客户端被绑定到各种事件上，这会让你的程序变得更灵活。（译者注：这句不太自信，贴出原文：The event emitter provides a great way of making a programming interface more generic. When you use a common understood pattern, clients bind to events instead of invoking functions, making your program more flexible.）

此外，通过使用事件发射器，你还可以获得许多特性，比如在同一事件上绑定多个互不相关的监听器。

从 Node 事件发射器继承

如果你对 Node 的事件发射器模式感兴趣，并打算用到自己的应用程序里，你可以通过继承 EventEmitter 来创建一个伪类：

```
util = require('util');
var EventEmitter = require('events').EventEmitter;
// 这是MyClass的构造函数:
var MyClass = function() {
}
util.inherits(MyClass, EventEmitter);
```

注意：util.inherits建立了MyClass的原形链，让你的MyClass实例可以使用EventEmitter的原形方法。

发射事件

通过继承自 EventEmitter，MyClass 可以像这样发射事件了：

```
MyClass.prototype.someMethod = function() {
    this.emit("custom event", "argument 1", "argument 2");
};
```

上面的代码，当 someMethond 方法被 MyClass 的实例调用时，就会发射一个叫“cuteom event”的事件，这个事件还会发射两个字符串作为数据：“argument 1”和“argument 2”，它们将会作为参数传递给事件监听器。

MyClass 实例的客户端可以像这样监听“custom event”事件：

```
var myInstance = new MyClass();
myInstance.on('custom event', function(str1, str2) {
    console.log('got a custom event with the str1 %s and str2 %s!', str1, str2);
});
```

再比如，你可以这样创建一个每秒发射一次“tick”事件的Ticker类：

```
var util = require('util'),
EventEmitter = require('events').EventEmitter;
var Ticker = function() {
    var self = this;
    setInterval(function() {
        self.emit('tick');
    }, 1000);
};
util.inherits(Ticker, EventEmitter);
```

用Ticker类的客户端可以展示如何使用Ticker类和监听“tick”事件，

```
var ticker = new Ticker();
ticker.on("tick", function() {
    console.log("tick");
});
```

小结

事件发射器模式是种可重入模式（recurrent pattern），可以用它将事件发射器对象从一组特定事件的代码中解耦合。

可以用 event_emitter.on() 来为特定类型的事件注册监听器，并用 event_emitter.removeListener()来取消注册。

还可以通过继承 EventEmitter 和简单的使用.emit()函数来创建自己的事件发射器。

译者注：

本文对应原文第二部分第五章：Node Core API Basics：Using the Event Emitter Pattern

本系列文章列表和翻译进度，请移步：Node.js 高级编程：用 Javascript 构建可伸缩应用（〇）