[实现一个音乐播放器] Node 的事件机制 - EventEmitter

> 本节目标：【实现一个音乐播放器】一生二再生万物，一切皆事件，Node **最野性的魅力就来自于对于事件队列的精妙处理。**

事件，是用户与浏览器互动过程中，最高频的一种交互机制，用户无论是鼠标点击，滚动，拖拽，还是一个表单文件上传行为，都通过事件的形式来与应用运行环境互动。事件有它的触发者，也有它的接收者或者处理者，连接这两者以及赋能二者能力的就是事件机制。

对于一个异步行为，浏览器不知道用户什么时候点击网页按钮，用户同样不知道点击按钮后浏览器什么时候给予回应，有了事件机制，这件事情就变得很容易，比如监听一个按钮的点击行为：

```
var btn = document.getElementById('btn')

btn.addEventListener('click', function (e) {
  // 按钮点击事件被监听到，开始处理事务
}, false)
```

事件如何处理不仅在浏览器需要考虑，服务器也有类似的场景，服务器既不知道一个请求什么时候会到来，请求处理程序也不知道背后的数据库查询行为什么时候成功返回，那么这些异步场景就需要一种机制来连接和通知彼此，在 Node 里面，很多操作都会触发事件，例如 net.Server 会在每一次有客户端连接到它时触发事件，又如 fs.readStream 会在文件打开时触发事件，所有具备触发事件能力的接口对象都是 events 的实例，在 Node 里面，事件能跑起来需要两个关键组成，分别就是 EventEmitter 和回调函数，前者负责生成实例，而后者负责执行特定任务。

回调函数与事件驱动

在聊 EventEmitter 之前，我们先看下回调，回调是异步编程模型里面最常见的一种方法，在 Node 里面也是如此，可以将后续逻辑封装在回调函数中作为当前函数的参数，逐层嵌套，逐层执行，最终让程序按照我们期望的方式走完流程，那么一个回调函数长什么样子呢?

```
function doSomeThing (thing) {
  console.log(thing)
}

function comeTo (place, cb) {
  const thing = '到 ' + place + ' 学习 Node'
  cb(thing)
}

comeTo('Juejin', doSomeThing)
// 到 Juejin 学习 Node
```

可以发现回调在 JS 里面，本质上是控制权的后移，把一个提前声明好的函数以参数的形式交给当前函数，当前函数在某个时刻再调用传入这个函数，同时对这个函数可以传入一些新的参数数据，那么这个函数 cb 就是我们所说的回调函数，这个回调函数不会马上执行。如果这个回调函数结合事件来执行，当某个事件发生的时候再调用回调函数，这种函数执行的方式叫做事件驱动，所以我们常看到 Node 的一大卖点就是事件驱动（event-driven），它起一个服务器的代码，请求的接收与响应本身也是回调函数来实现：

```
require('http').createServer((req, res) => {
  res.write('Hi')
  res.end()
}).listen(3333, '127.0.0.1', () => {})
```

EventEmitter 的基本用法

在 Node 里面，events 模块提供了 EventEmitter 的 Class 类，可以直接创建一个事件实例：

```
// events 是 Node 的 built-in 模块，它提供了 EventEmitter 类
const EventEmitter = require('events')
// 创建 EventEmitter 的事件实例
const ee = new EventEmitter()

// 为实例增加 open 事件的监听以及注册回调函数，事件名甚至可以是中文
ee.on('open', (error, result) => {
 console.log('事件发生了，第一个监听回调函数执行')
})

// 为实例再增加一个 增加 open 事件的监听器
ee.on('open', (error, result) => {
 console.log('事件发生了，第二个监听回调函数执行')
})

// 通过 emit 来发出事件，所有该事件队列里的回调函数都会顺序执行
ee.emit('open')
console.log('触发后，隔一秒再触发一次')
setTimeout(() => {
 ee.emit('open')
}, 1000)
// 事件发生了，第一个监听回调函数执行
// 事件发生了，第二个监听回调函数执行
// 触发后，隔一秒再触发一次
// 事件发生了，第一个监听回调函数执行
// 事件发生了，第二个监听回调函数执行
```

一个事件实例上有如下的属性和方法：

- addListener(event, listener): 向事件队列后面再增加一个监

听器
- emit(event, [arg1], [arg2], [...]): 向事件队列触发一个事件，同时可以对该事件传过去更多的数据
- listeners(event): 返回事件队列中特定的事件监听对象
- on(event, listener): 针对一个特定的事件注册监听器，该监听器就是一个回调函数
- once(event, listener): 与 on 一样，只不过它只会执行一次，只生效一次
- removeAllListeners([event]): 移除所有指定事件的监听器，不指定的话，移除所有监听器，也就是清空事件队列
- removeListener(event, listener): 只移除特定事件监听器
- setMaxListeners(n): 设置监听器数组的最大数量，默认是 10，超过 10 个会收到 Node 的警告

定制自己的 events

如果我们在设计一款游戏，来监听一个玩家每一局干掉敌人，比如僵尸的个数，不同的个数会有不同的奖励机制，我们的代码可能会这样写：

```
// 声明一个玩家类
class Player {
  // 给他初始的名字和分数
  constructor (name) {
    this.name = name
    this.score = 0
  }

  // 每一局打完，统计干掉游戏目标个数，来奖励分值
  killed (target, number) {
    if (target !== 'zombie') return
    if (number < 10) {
      this.score += 10 * number
```

```
    } else if (number < 20) {
      this.score += 8 * number
    } else if (number < 30) {
      this.score += 5 * number
    }

    console.log(`${this.name} 成功击杀 ${number} 个
${target}，总得分 ${this.score}`)
  }
}
// 创建一个玩家人物
let player = new Player('Nil')

// 玩了 3 局，每一局都有收获
player.killed('zombie', 5)
player.killed('zombie', 12)
player.killed('zombie', 22)
// Nil 成功击杀 5 个 zombie，总得分 50
// Nil 成功击杀 12 个 zombie，总得分 146
// Nil 成功击杀 22 个 zombie，总得分 256
```

这样的代码简单易懂，逻辑都控制在 killed 方法里面，但是扩展性不是很好，比如我想要在 killed 的时候，多做一些其他事情，我不得不去重写或者覆盖这个 killed 方法，定制程度更弱一些，如果游戏目标除了僵尸，还有吸血鬼、灵兽、虫子等等，他们的激励策略都不相同，通过几个方法来定制加分策略会更硬编码一些，那如果我们换用 events 的事件来简单实现下呢：

```
const EventEmitter = require('events')

// 声明玩家类，让它继承 EventEmitter
class Player extends EventEmitter {
  constructor (name) {
```

```
    super()
    this.name = name
    this.score = 0
  }
}

let player = new Player('Nil')

// 每一个创建的玩家实例，都可以添加监听器
// 也可以定义需要触发事件的名称，为其注册回调
player.on('zombie', function (number) {
  if (number < 10) {
    this.score += 10 * number
  } else if (number < 20) {
    this.score += 8 * number
  } else if (number < 30) {
    this.score += 5 * number
  }

  console.log(`${this.name} 成功击杀 ${number} 个 zombie，总得分 ${this.score}`)
})

// 可以触发不同的事件类型
player.emit('zombie', 5)
player.emit('zombie', 12)
player.emit('zombie', 22)
// Nil 成功击杀 5 个 zombie，总得分 50
// Nil 成功击杀 12 个 zombie，总得分 146
// Nil 成功击杀 22 个 zombie，总得分 256
```

通过 Node 内建的 events，我们可以通过继承它来实现更灵活的类控制，给予类实例更多的控制颗粒度，即便是游戏规则变更，从代码的耦合度和维护性上看，后面这一种实现都会更轻量更灵活。

编程练习 - 命令行搜歌播放工具

能动手就不吵吵，Events 看着比较简单，我们应用到案例中感受一下，现在我们一起来开发一个工具，可以在命令行窗口中搜歌和播放歌曲，依然按照第六节的 NPM 发包流程，来创建这个项目，它的结构如下：

```
~ tree -L 2
.
├── README.md
├── bin
|   └── souge
├── index.js
├── lib
|   ├── choose.js
|   ├── find.js
|   ├── names.js
|   ├── play.js
|   ├── request.js
|   └── search.js
├── package-lock.json
└── package.json
```

在 package.json 中，我们增加执行的脚本路径：

```
"bin": {
  "souge": "./bin/souge"
},
```

然后在 /bin/souge 里面，增加如下脚本代码：

```
#!/usr/bin/env node

const pkg = require('../package')
const emitter = require('..')

function printVersion() {
  console.log('souge ' + pkg.version)
  process.exit()
}

function printHelp(code) {
  const lines = [
    '',
    '  Usage:',
    '    souge [songName]',
    '',
    '  Options:',
    '    -v, --version    print the version of
vc',
    '    -h, --help       display this message',
    '',
    '  Examples:',
    '    $ souge Hello',
    ''
  ]

  console.log(lines.join('\n'))
  process.exit(code || 0)
}

const main = async (argv) => {
  if (!argv || !argv.length) {
    printHelp(1)
```

```
  }

  let arg = argv[0]

  switch(arg) {
    case '-v':
    case '-V':
    case '--version':
      printVersion()

      break
    case '-h':
    case '-H':
    case '--help':
      printHelp()

      break
    default:
      # 启动搜索逻辑，同时传入参数
      emitter.emit('search', arg)
      break
  }
}

main(process.argv.slice(2))
module.exports = main
```

我们只需要关注 main 函数就可以了，当通过 souge hello 类似这样执行时，会把 hello 这个参数带进去，没有匹配到既有的其他参数标识，就会通过 emitter 来触发一个搜索事件，而 emitter 实例我们是从外层的 index.js 里面拿到的，所以在 index.js 里面这样写，大家跟着我的注释来看代码：

```
// names 是拼接歌曲名称的一个方法
const names = require('./lib/names')
const EventEmitter = require('events')
// 声明一个继承 EventEmitter 的事件类
class Emitter extends EventEmitter {}
// 实例化一个事件实例
const emitter = new Emitter()

;[
  'search',
  'choose',
  'find',
  'play'
].forEach(key => {
  // 加载 search/choose/find/play 四个模块方法
  const fn = require(`./lib/${key}`)
  // 为 emitter 增加 4 个事件监听，key 就是模块名
  emitter.on(key, async function (...args) {
    // 在事件回调里面，调用模块方法，无脑传入事件入参
    const res = await fn(...args)
    // 执行模块方法后，再触发一个新事件 hanlder
    // 同时把多个参数，如 key/res 继续丢过去
    this.emit('handler', key, res, ...args)
  })
})

// 搜索后触发 afterSearch，它回调里面继续触发 choose 事
件
emitter.on('afterSearch', function (data, q) {
  if (!data || !data.result ||
!data.result.songs) {
    console.log(`没搜索到 ${q} 的相关结果`)
    return process.exit(1)
```

```
  }
  const songs = data.result.songs
  this.emit('choose', songs)
})

// 在歌曲被选中后，它回调里面继续触发 find 事件
emitter.on('afterChoose', function (answers,
songs) {
  const arr = songs.filter((song, i) => (
    names(song, i) === answers.song
  ))

  if (arr[0] && arr[0].id) {
    this.emit('find', arr[0].id)
  }
})

// 在歌曲被找到后，它回调里面触发 play 播放事件
emitter.on('afterFind', function (songs) {
  if (songs[0] && songs[0].url) {
    this.emit('play', songs[0].url)
  }
})

// 监听播放，并对播放结束后继续触发 playEnd
emitter.on('playing', function (player) {
  player.on('playend', (item) => {
    this.emit('playEnd')
  })
})

// 收到播放结束，退出程序
emitter.on('playEnd', function (player) {
```

```
  console.log('播放结束!')
  process.exit()
})

// 这里的 handler 精简了多个事件的判断
// 为不同的事件增加了不同的触发回调
emitter.on('handler', function (key, res,
...args) {
  switch (key) {
    case 'search':
      return this.emit('afterSearch', res,
args[0])
    case 'choose':
      return this.emit('afterChoose', res,
args[0])
    case 'find':
      return this.emit('afterFind', res)
    case 'play':
      return this.emit('playing', res)
  }
})

module.exports = emitter
```

歌曲的搜索播放主逻辑有了后，我们就可以各个击破了，首先是搜索逻辑，在 /lib/search.js 里面发一个请求，把歌曲名字带过去，开始搜索，这里借用了 imjad.cn (https://api.imjad.cn/cloudmusic.md) 的搜歌 API，大家如果自己学习，也可以自行搭建其他歌曲 API 服务，有很多开源的项目可以参考：

```
const request = require('./request')

module.exports = (name) => {
  const url = 'https://api.imjad.cn/cloudmusic/?
type=search&search_type=1&s=' + name

  return request(url)
}
```

这里请求模块，也就是 /lib/request.js 代码如下：

```
const https = require('https')

module.exports = (url) => new Promise((resolve,
reject) => {
  https.get(url, (req, res) => {
    let data = []

    req.on('data', chunk => {
      data.push(chunk)
    })
    req.on('end', () => {
      let body
      try {
        body = JSON.parse(data.join(''))
      } catch (err) {
        console.log('<== API 服务器可能挂了，稍后重
试！==>')
      }

      resolve(body)
    })
  })
})
```

这里的代码就很简单了，一个普通的 HTTP 请求，把收到的 Buffer 数据最终拼接后返回，在返回后，又会一路触发 afterSearch 和 choose 事件，在 choose 时候，会显示一个歌曲列表，代码也很简单：

```
const inquirer = require('inquirer')
const names = require('./names')

module.exports = (songs) => inquirer.prompt([{
  type: 'list',
  name: 'song',
  message: '共有 ' + songs.length + ' 个结果，按下回车播放',
  choices: songs.map((i, index) => names(i, index))
}])
```

拼接歌曲名称的代码 /lib/names.js 只有一句：

```
module.exports = (item, index) =>
  `${index + 1} ${item.name} ${item.ar[0].name} ${item.al.name}`
```

而 inquirer 是一个三方模块，赋予我们跟命令行窗口更友好的交互方式，通过 inquirer.prompt 我们了解到是哪一首歌曲被选中了，最后当歌曲被选中时，一路就触发 afterChoose 和 find 事件，在 find 时候，就用到了 /lib/find.js:

```
const request= require('./request')

module.exports = async (id) => {
  const url = 'https://api.imjad.cn/cloudmusic/?type=song&br=128000&id=' + id
  const { data } = await request(url)
  return data
}
```

这里依然是个简单的请求，获取到音乐文件流数据，最后在 afterFind 里面，触发 play 事件，播放音乐文件，播放代码在 /lib/play.js 里面：

```
const Player = require('player')

module.exports = (url) => {
  return new Promise((resolve, reject) => {
    // 实例化一个播放器，立刻启动播放
    const player = new Player(url)
    player.play()

    // 播放时候，触发 playing
    player.on('playing', function (item) {
      console.log('播放中!')
      resolve(player)
    })

    player.on('error', function (err) {
      // when error occurs
      console.log('播放出错!')
      reject(err)
    })
  })
}
```

这里封装了一个 Promise，包裹了 Player 的创建和播放过程，直到最终播放结束事件触发，整个过程完成，所以事件的整体触发顺序是：

- search 搜索启动
- afterSearch 搜索完成
- choose 歌曲选单

- afterChoose 选中动作触发
- find 去找寻对应歌曲
- afterFind 歌曲文件数据拿到
- play 开始播放
- playing 正在播放
- playEnd 播放结束

这样通过事件，我们就非常方便的管理了整个流程的多个状态，如果我们想要集成其他的事件，只要处理好事件触发顺序，就变得易如反掌了。

最后，我们可以把这个模块发布到 npm，大家也可以 `npm i souge -g` 来体验这个工具，最原始的代码在 [4liang/souge (https://github.com/4liang/souge)](https://github.com/4liang/souge)，考虑到学习，未对代码做过多优化，大家可以基于此修改后，来提 PR。