# 一 简单使用

Seneca是一个用来快速构建基于消息的微服务系统的工具集，你不需要知道服务本身部署在何处，不需要知道具体有多少服务，任何业务逻辑之外的服务，如数据库、缓存、第三方集成都被隐藏在微服务之后。

Seneca三大核心功能：

模式匹配：不同于脆弱的服务发现，模式匹配可以告诉真正的消息是什么；

无依赖传输：可以使用多种方式在服务间发送消息，且都隐藏在业务逻辑后；

组件化：功能被表示为一组可以一起组成微服务的插件。

在Seneca中，消息就是一个可以有任何你喜欢的内部结构的JSON对象，通过HTTP/TCP/消息队列、发布订阅或者任何能传输数据的方式进行传输。作为消息生产者的开发者来讲，只需要将消息发送出去即可，完全不用关心哪些服务来接收。

然后，你又想告诉这个世界，你想要接收一些消息，这也很简单，你只需在 Seneca 中作一点匹配模式配置即可，匹配模式也很简单，只是一个键值对的列表，这些键值对被用于匹配 JSON 消息的极组属性。

## 模式 Patterns

const seneca = require('seneca')();  
  
seneca.add(  
 'role:math, cmd:sum',  
 (msg, reply) => {  
 reply(  
 null,  
 {  
 answer: ( msg.left + msg.right )  
 }  
 )  
 }  
);  
  
seneca.act(  
 {  
 role: 'math',  
 cmd: 'sum',  
 left: 1,  
 right: 2  
 },  
 (err, result) => {  
 if (err) {  
 return console.error(err);  
 }  
 console.log(result);  
 }  
);

输出：

{ answer: 3 }

{"kind":"notice","notice":"hello seneca mn4epga9po65/1522746421003/3248/3.4.3/-","level":"info","seneca":"mn4epga9po65/1522746421003/3248/3.4.3/-","when":1522746422106}

到目前为止，所有这一切都发生在同一个进程中，没有网络流量产生，进程内的函数调用也是基于消息传输。

seneca.add 方法，添加了一个新的动作模式（\_Action Pattern\_）至 Seneca 实例中，它有两个参数：

pattern ：用于匹配 Seneca 实例中 JSON 消息体的模式；

action ：当模式被匹配时执行的操作

（role 与 cmd 这两个属性没有什么特别，只是恰好被你用于匹配模式）

seneca.act 方法同样有两个参数：

msg ：发送的消息主体，即作为纯对象提供的待匹配的入站消息；

respond ：用于接收并处理响应信息的回调函数（没有响应也会执行）。

上述案例展示了如何定义并创建一个 Action 以及如何呼起一个 Action，但它们都发生在一个进程中，接下来，我们很快就会展示如何拆分成不同的代码和多个进程。

## 匹配模式如何工作

在上述代码中再添加一个案例：

const seneca = require('seneca')();  
  
seneca.add('role:math, cmd:sum', (msg, reply) => {  
 reply(null, { answer: ( msg.left + msg.right )})  
});  
  
seneca.add('role:math, cmd:product', (msg, reply) => {  
 reply(null, { answer: ( msg.left \* msg.right )})  
});  
  
seneca.act({role: 'math', cmd: 'sum', left: 1, right: 2}, console.log)  
 .act({role: 'math', cmd: 'product', left: 3, right: 4}, console.log)

我们发现， seneca.act 是可以进行链式调用的，Seneca 提供了一个链式API，调式调用是顺序执行的，但是不是串行，所以，返回的结果的顺序可能与调用顺序并不一样。

## 扩展模式以增加新功能

下面让我们扩展一下 role: math, cmd: sum 操作，它只接收整型数字，那么，怎么做？

seneca.add({role: 'math', cmd: 'sum', integer: true}, function (msg, respond) {

var sum = Math.floor(msg.left) + Math.floor(msg.right)

respond(null, {answer: sum})

})

现在，下面这条消息：

{role: 'math', cmd: 'sum', left: 1.5, right: 2.5, integer: true}

将得到下面这样的结果：

{answer: 3} // == 1 + 2，小数部分已经被移除了

现在，你的两个模式都存在于系统中了，而且还存在交叉部分，那么 Seneca 最终会将消息匹配至哪条模式呢？原则是：更多匹配项目被匹配到的优先，被匹配到的属性越多，则优先级越高。

const seneca = require('seneca')()  
  
seneca.add({role: 'math', cmd: 'sum'}, function (msg, respond) {  
 var sum = msg.left + msg.right  
 respond(null, {answer: sum})  
})  
  
*// 下面两条消息都匹配 role: math, cmd: sum*seneca.act({role: 'math', cmd: 'sum', left: 1.5, right: 2.5}, console.log)  
seneca.act({role: 'math', cmd: 'sum', left: 1.5, right: 2.5, integer: true}, console.log)  
  
setTimeout(() => {  
 seneca.add({role: 'math', cmd: 'sum', integer: true}, function (msg, respond) {  
 var sum = Math.floor(msg.left) + Math.floor(msg.right)  
 respond(null, { answer: sum })  
 })  
  
 *// 下面这条消息同样匹配 role: math, cmd: sum* seneca.act({role: 'math', cmd: 'sum', left: 1.5, right: 2.5}, console.log)  
  
 *// 但是，也匹配 role:math,cmd:sum,integer:true  
 // 但是因为更多属性被匹配到，所以，它的优先级更高* seneca.act({role: 'math', cmd: 'sum', left: 1.5, right: 2.5, integer: true}, console.log)  
}, 100)

输出结果如下：

null { answer: 4 }

null { answer: 4 }

null { answer: 4 }

null { answer: 3 }

在上面的代码中，因为系统中只存在 role: math, cmd: sum 模式，所以，都匹配到它，但是当 100ms 后，我们给系统中添加了一个 role: math, cmd: sum, integer: true 模式之后，结果就不一样了，匹配到更多的操作将有更高的优先级。

这种设计，可以让我们的系统可以更加简单的添加新的功能，不管是在开发环境还是在生产环境中，你都可以在不需要修改现有代码的前提下即可更新新的服务，你只需要先好新的服务，然后启动新服务即可。

## 基于模式的代码复用

const seneca = require('seneca')()  
  
seneca.add('role: math, cmd: sum', function (msg, respond) {  
 var sum = msg.left + msg.right  
 respond(null, {answer: sum})  
})  
  
seneca.add('role: math, cmd: sum, integer: true', function (msg, respond) {  
 *// 复用 role:math, cmd:sum* this.act({  
 role: 'math',  
 cmd: 'sum',  
 left: Math.floor(msg.left),  
 right: Math.floor(msg.right)  
 }, respond)  
})  
  
*// 匹配 role:math,cmd:sum*seneca.act('role: math, cmd: sum, left: 1.5, right: 2.5',console.log)  
  
*// 匹配 role:math,cmd:sum,integer:true*seneca.act('role: math, cmd: sum, left: 1.5, right: 2.5, integer: true', console.log)

在上面的示例代码中，我们使用了 this.act 而不是前面的 seneca.act，那是因为，在 action 函数中，上下文关系变量 this ，引用了当前的 seneca 实例，这样你就可以在任何一个 action 函数中，访问到该 action 调用的整个上下文。

在上面的代码中，我们使用了 JSON 缩写形式来描述模式与消息：

{role: 'math', cmd: 'sum', left: 1.5, right: 2.5}

缩写模式为：'role: math, cmd: sum, left: 1.5, right: 2.5'

jsonic 这种格式，提供了一种以字符串字面量来表达对象的简便方式，这使得我们可以创建更加简单的模式和消息。

## 模式是唯一的

定义的 Action 模式都是唯一的，它们只能触发一个函数，模式的解析规则如下：

属性越多优先级更高；

若模式具有相同的数量的属性，则按字母顺序匹配；

案例：

a: 1, b: 2 优先于 a: 1， 因为它有更多的属性；

a: 1, b: 2 优先于 a: 1, c: 3，因为 b 在 c 字母的前面；

a: 1, b: 2, d: 4 优先于 a: 1, c: 3, d:4，因为 b 在 c 字母的前面；

a: 1, b:2, c:3 优先于 a:1, b: 2，因为它有更多的属性；

a: 1, b:2, c:3 优先于 a:1, c:3，因为它有更多的属性。

很多时间，提供一种可以让你不需要全盘修改现有 Action 函数的代码即可增加它功能的方法是很有必要的，比如，你可能想为某一个消息增加更多自定义的属性验证方法，捕获消息统计信息，添加额外的数据库结果中，或者控制消息流速等。

下面的示例代码中，加法操作期望 left 和 right 属性是有限数，此外，为了调试目的，将原始输入参数附加到输出的结果中也是很有用的，您可以使用以下代码添加验证检查和调试信息：

const seneca = require('seneca')()  
  
seneca  
 .add(  
 'role:math,cmd:sum',  
 function(msg, respond) {  
 var sum = msg.left + msg.right  
 respond(null, {  
 answer: sum  
 })  
 })  
  
 *// 重写 role:math,cmd:sum with ，添加额外的功能* .add(  
 'role:math,cmd:sum',  
 function(msg, respond) {  
  
 *// bail out early if there's a problem* if (!Number.isFinite(msg.left) ||  
 !Number.isFinite(msg.right)) {  
 return respond(new Error("left 与 right 值必须为数字。"))  
 }  
  
 *// 调用上一个操作函数 role:math,cmd:sum* this.prior({  
 role: 'math',  
 cmd: 'sum',  
 left: msg.left,  
 right: msg.right,  
  
 }, function(err, result) {  
 if (err) return respond(err)  
  
 result.info = msg.left + '+' + msg.right  
 respond(null, result)  
 })  
 })  
  
 *// 增加了的 role:math,cmd:sum* .act('role:math,cmd:sum,left:1.5,right:2.5',  
 console.log *// 打印 { answer: 4, info: '1.5+2.5' }* )

seneca 实例提供了一个名为 prior 的方法，让可以在当前的 action 方法中，调用被其重写的旧操作函数。

## 使用插件组织模式

一个 seneca 实例，其实就只是多个 Action Patterm 的集合而已，你可以使用命名空间的方式来组织操作模式，例如在前面的示例中，我们都使用了 role: math，为了帮助日志记录和调试， Seneca 还支持一个简约的插件支持。

同样，Seneca插件只是一组操作模式的集合，它可以有一个名称，用于注释日志记录条目，还可以给插件一组选项来控制它们的行为，插件还提供了以正确的顺序执行初始化函数的机制，例如，您希望在尝试从数据库读取数据之前建立数据库连接。

简单来说，Seneca插件就只是一个具有单个参数选项的函数，你将这个插件定义函数传递给 seneca.use 方法，下面这个是最小的Seneca插件（其实它什么也没做！）：

function math(options) {  
  
 this.add('role:math,cmd:sum', function (msg, respond) {  
 respond(null, { answer: msg.left + msg.right })  
 })  
  
 this.add('role:math,cmd:product', function (msg, respond) {  
 respond(null, { answer: msg.left \* msg.right })  
 })  
  
}  
  
require('seneca')()  
 .use(math)  
 .act('role:math,cmd:sum,left:1,right:2', console.log)

Seneca 还提供了详细日志记录功能，可以提供为开发或者生产提供更多的日志信息，通常的，日志级别被设置为 INFO，它并不会打印太多日志信息，如果想看到所有的日志信息，试试以下面这样的方式启动你的服务：

node minimal-plugin.js --seneca.log.all

会不会被吓一跳？当然，你还可以过滤日志信息：

node minimal-plugin.js --seneca.log.all | grep plugin:define

通过日志我们可以看到， seneca 加载了很多内置的插件，比如 basic、transport、web 以及 mem-store，这些插件为我们提供了创建微服务的基础功能，同样，你应该也可以看到 minimal\_plugin 插件。

查看上述代码的日志：所有的该插件的日志都被自动的添加了 plugin 属性。

在 Seneca 的世界中，我们通过插件组织各种操作模式集合，这让日志与调试变得更简单，然后你还可以将多个插件合并成为各种微服务，在接下来的章节中，我们将创建一个 math 服务。

插件通过需要进行一些初始化的工作，比如连接数据库等，但是，你并不需要在插件的定义函数中去执行这些初始化，定义函数被设计为同步执行的，因为它的所有操作都是在定义一个插件，事实上，你不应该在定义函数中调用 seneca.act 方法，只调用 seneca.add 方法。

要初始化插件，你需要定义一个特殊的匹配模式 init: <plugin-name>，对于每一个插件，将按顺序调用此操作模式，init 函数必须调用其 callback 函数，并且不能有错误发生，如果插件初始化失败，则 Seneca 会立即退出 Node 进程。所以的插件初始化工作都必须在任何操作执行之前完成。

为了演示初始化，让我们向 math 插件添加简单的自定义日志记录，当插件启动时，它打开一个日志文件，并将所有操作的日志写入文件，文件需要成功打开并且可写，如果这失败，微服务启动就应该失败。

const fs = require('fs')  
  
function math(options) {  
  
 *// 日志记录函数，通过 init 函数创建* var log  
  
 *// 将所有模式放在一起会上我们查找更方便* this.add('role:math,cmd:sum', sum)  
 this.add('role:math,cmd:product', product)  
  
 *// 这就是那个特殊的初始化操作* this.add('init:math', init)  
  
 function init(msg, respond) {  
 *// 将日志记录至一个特写的文件中* fs.open(options.logfile, 'a', function (err, fd) {  
  
 *// 如果不能读取或者写入该文件，则返回错误，这会导致 Seneca 启动失败* if (err) return respond(err)  
  
 log = makeLog(fd)  
 respond()  
 })  
 }  
  
 function sum(msg, respond) {  
 var out = { answer: msg.left + msg.right }  
 log('sum '+msg.left+'+'+msg.right+'='+out.answer+'\n')  
 respond(null, out)  
 }  
  
 function product(msg, respond) {  
 var out = { answer: msg.left \* msg.right }  
 log('product '+msg.left+'\*'+msg.right+'='+out.answer+'\n')  
 respond(null, out)  
 }  
  
 function makeLog(fd) {  
 return function (entry) {  
 fs.write(fd, new Date().toISOString()+' '+entry, null, 'utf8', function (err) {  
 if (err) return console.log(err)  
  
 *// 确保日志条目已刷新* fs.fsync(fd, function (err) {  
 if (err) return console.log(err)  
 })  
 })  
 }  
 }  
}  
  
require('seneca')()  
 .use(math, {logfile:'./math.log'})  
 .act('role:math,cmd:sum,left:1,right:2', console.log)

在上面这个插件的代码中，匹配模式被组织在插件的顶部，以便它们更容易被看到，函数在这些模式下面一点被定义，您还可以看到如何使用选项提供自定义日志文件的位置（不言而喻，这不是生产日志！）。

初始化函数 init 执行一些异步文件系统工作，因此必须在执行任何操作之前完成。 如果失败，整个服务将无法初始化。要查看失败时的操作，可以尝试将日志文件位置更改为无效的，例如 /math.log。

## 创建微服务

现在让我们把 math 插件变成一个真正的微服务。首先，你需要组织你的插件。 math 插件的业务逻辑 ---- 即它提供的功能，与它以何种方式与外部世界通信是分开的，你可能会暴露一个Web服务，也有可能在消息总线上监听。

将业务逻辑（即插件定义）放在其自己的文件中是有意义的。 Node.js 模块即可完美的实现，创建一个名为 math.js 的文件，内容如下：

module.exports = function math(options) {  
  
 this.add('role:math,cmd:sum', function sum(msg, respond) {  
 respond(null, { answer: msg.left + msg.right })  
 })  
  
 this.add('role:math,cmd:product', function product(msg, respond) {  
 respond(null, { answer: msg.left \* msg.right })  
 })  
  
 this.wrap('role:math', function (msg, respond) {  
 msg.left = Number(msg.left).valueOf()  
 msg.right = Number(msg.right).valueOf()  
 this.prior(msg, respond)  
 })  
}

使用：

require('seneca')()  
 .use(require('./math.js')) *//也可以写为：.use('math') // 在当前目录下找到 `./math.js`* .act('role:math,cmd:sum,left:1,right:2', console.log)

seneca.wrap 方法可以匹配一组模式，同使用相同的动作扩展函数覆盖至所有被匹配的模式，这与为每一个组模式手动调用 seneca.add 去扩展可以得到一样的效果，它需要两个参数：

pin ：模式匹配模式

action ：扩展的 action 函数

pin 是一个可以匹配到多个模式的模式，它可以匹配到多个模式，比如 role:math 这个 pin 可以匹配到 role:math, cmd:sum 与 role:math, cmd:product。

在上面的示例中，我们在最后面的 wrap 函数中，确保了，任何传递给 role:math 的消息体中 left 与 right 值都是数字，即使我们传递了字符串，也可以被自动的转换为数字。

有时，查看 Seneca 实例中有哪些操作是被重写了是很有用的，你可以在启动应用时，加上 --seneca.print.tree 参数即可，我们先创建一个 math-tree.js 文件，填入以下内容：

require('seneca')()

.use('math')

但是，所有的操作都还存在于同一个进程中，先创建一个名为 math-service.js 的文件:

math-service.js

require('seneca')()

.use('math')

.listen()

启动该脚本，即可启动微服务。它会启动一个进程，并通过 10101 端口监听HTTP请求，但不是一个 Web 服务器，在此时， HTTP 仅仅作为消息的传输机制。

访问 http://localhost:10101/act?ro... 即可看到结果，或者使用 curl 命令：

curl -d '{"role":"math","cmd":"sum","left":1,"right":2}' http://localhost:10101/act

两种方式都可以看到结果：

{"answer":3}

创建一个微服务客户端 math-client.js：

require('seneca')()

.client()

.act('role:math,cmd:sum,left:1,right:2',console.log)

打开一个新的终端，执行该脚本：

null { answer: 3 } { id: '7uuptvpf8iff/9wfb26kbqx55',

accept: '043di4pxswq7/1483589685164/65429/3.2.2/-',

track: undefined,

time:

{ client\_sent: '0',

listen\_recv: '0',

listen\_sent: '0',

client\_recv: 1483589898390 } }

在 Seneca 中，我们通过 seneca.listen 方法创建微服务，然后通过 seneca.client 去与微服务进行通信。在上面的示例中，我们使用的都是 Seneca 的默认配置，比如 HTTP 协议监听 10101 端口，但 seneca.listen 与 seneca.client 方法都可以接受下面这些参数，以达到定抽的功能：

port ：可选的数字，表示端口号；

host ：可先的字符串，表示主机名或者IP地址；

spec ：可选的对象，完整的定制对象

注意：在 Windows 系统中，如果未指定 host， 默认会连接 0.0.0.0，这是没有任何用处的，你可以设置 host 为 localhost。

只要 client 与 listen 的端口号与主机一致，它们就可以进行通信：

seneca.client(8080) → seneca.listen(8080)

seneca.client(8080, '192.168.0.2') → seneca.listen(8080, '192.168.0.2')

seneca.client({ port: 8080, host: '192.168.0.2' }) → seneca.listen({ port: 8080, host: '192.168.0.2' })

Seneca 为你提供的 无依赖传输 特性，让你在进行业务逻辑开发时，不需要知道消息如何传输或哪些服务会得到它们，而是在服务设置代码或配置中指定，比如 math.js 插件中的代码永远不需要改变，我们就可以任意的改变传输方式。

虽然 HTTP 协议很方便，但是并不是所有时间都合适，另一个常用的协议是 TCP，我们可以很容易的使用 TCP 协议来进行数据的传输，尝试下面这两个文件：

math-service-tcp.js :

require('seneca')()

.use('math')

.listen({type: 'tcp'})

math-client-tcp.js

require('seneca')()

.client({type: 'tcp'})

.act('role:math,cmd:sum,left:1,right:2',console.log)

默认情况下， client/listen 并未指定哪些消息将发送至哪里，只是本地定义了模式的话，会发送至本地的模式中，否则会全部发送至服务器中，我们可以通过一些配置来定义哪些消息将发送到哪些服务中，你可以使用一个 pin 参数来做这件事情。

让我们来创建一个应用，它将通过 TCP 发送所有 role:math 消息至服务，而把其它的所有消息都在发送至本地：

math-pin-service.js：

require('seneca')()

.use('math')

// 监听 role:math 消息

// 重要：必须匹配客户端

.listen({ type: 'tcp', pin: 'role:math' })

math-pin-client.js：

require('seneca')()

// 本地模式

.add('say:hello', function (msg, respond){ respond(null, {text: "Hi!"}) })

// 发送 role:math 模式至服务

// 注意：必须匹配服务端

.client({ type: 'tcp', pin: 'role:math' })

// 远程操作

.act('role:math,cmd:sum,left:1,right:2',console.log)

// 本地操作

.act('say:hello',console.log)

你可以通过各种过滤器来自定义日志的打印，以跟踪消息的流动，使用 --seneca... 参数，支持以下配置：

date-time： log 条目何时被创建；

seneca-id： Seneca process ID；

level：DEBUG、INFO、WARN、ERROR 以及 FATAL 中任何一个；

type：条目编码，比如 act、plugin 等；

plugin：插件名称，不是插件内的操作将表示为 root$；

case： 条目的事件：IN、ADD、OUT 等

action-id/transaction-id：跟踪标识符，\_在网络中永远保持一致\_；

pin：action 匹配模式；

message：入/出参消息体

如果你运行上面的进程，使用了 --seneca.log.all，则会打印出所有日志，如果你只想看 math 插件打印的日志，可以像下面这样启动服务：

node math-pin-service.js --seneca.log=plugin:math

## 结合web服务器