

6 TCP

传输控制协议（TCP）是一个面向连接的协议，它保证了两台计算机之间数据传输的可靠性和顺序。 ◀ 69

换句话说，TCP是一种传输层协议，它可以让你将数据从一台计算机完整有序地传输到另一台计算机。

正是由于这些特点，很多我们现在使用的如HTTP这样的协议都是基于TCP协议的。当传输一个页面的HTML文件时，肯定是希望它传输到目的地时能够与传输前一致，要是出什么问题，就应该抛出错误。哪怕有一个字符（字节）传输错位了，浏览器都有可能无法渲染这个页面。

Node.js这个框架的出发点就是为了网络应用开发所设计的。如今，网络应用都是用TCP/IP协议进行通信的。所以，了解TCP/IP协议是如何工作的，以及Node.js是如何通过简单的API对其进行封装的，都是非常有帮助的。

首先要介绍的是该协议的特点。举例来说，使用TCP在两台电脑之间进行数据传输是如何做到的。当传输两条消息时，传输到目的地时还能保持发送前的顺序吗？理解协议本身对于理解使用该协议的软件也是很重要的。比如，大部分时候，连接如MySQL等的数据库以及与数

数据库进行通信使用的都是TCP套接字。

Node HTTP服务器是构建于Node TCP服务器之上的。从编程角度来说，也就是Node中的`http.Server`继承自`net.Server`（`net`是TCP模块）。

除了Web浏览器和服务端（HTTP）之外，很多我们日常使用的如邮件客户端（SMTP/IMAP/POP）、聊天程序（IRC/XMPP）以及远程shell（SSH）等都基于TCP协议。

尽可能多地了解TCP以及如何使用相关的Node.js API对书写和理解网络程序会大有裨益。

70 TCP有哪些特性

若只是使用TCP的话，那无须理解它内部的工作原理，以及其实现机制。

不过，理解这些东西对分析更高层的协议和服务端，如Web服务器、数据库等的问题有很大的帮助。

TCP的首要特性就是它是面向连接的。

面向连接的通信和保证顺序的传递

说到TCP，可以将客户端和服务端端的通信看作是一个连接或者数据流。这对开发面向服务的应用和流应用是很好的抽象，因为TCP协议做基于的IP协议是面向无连接的。

IP是基于数据报的传输。这些数据报是独立进行传输的，送达的顺序也是无序的。

那么TCP又是如何保证这些独立的数据报送达的时候是有序的呢？

使用IP协议意味着数据包送达时是无序的，这些数据包不属于任何的数据流或者连接，那么当使用TCP/IP和服务端建立连接后，是怎样做到让数据包送达时是有序的呢？

要回答上述问题其实就等于在解释为什么会有TCP。当在TCP连接内进行数据传递时，发送的IP数据报包含了标识该连接以及数据流顺序的信息。

假设一条消息分割为四个部分。当服务器从连接A收到第一部分和第四部分后，它就知道还要等待其他数据报中的第二部分和第三部分。

要是用Node来写一个TCP服务器，就完全没必要去考虑这些复杂的内部实现了。只要考虑连接以及往套接字中写数据即可。接收方会按序接收到传输的信息，要是发生网络错误，连接会失效或者终止。

面向字节

TCP对字符以及字符编码是完全无知的。正如第4章介绍的，不同的编码会导致传输的字节数不同。

所以，TCP允许数据以ASCII字符（每个字符一个字节）或者 Unicode（即每个字符四个字节）进行传输。

正是因为对消息格式没有严格的约束，使得TCP有很好的灵活性。

71

可靠性

由于TCP是基于底层不可靠的服务，因此，它必须要基于确认和超时实现一系列的机制来达到可靠性的要求。

当数据发送出去后，发送方就会等待一个确认消息（表示数据包已经收到的简短的确认消息）。如果过了指定的窗口时间，还未收到确认消息，发送方就会对数据进行重发。

这种机制有效地解决了如网络错误或者网络阻塞这样的不可预测的情况。

流控制

要是两台互相通信的计算机中，有一台速度远快于另一台的话，会怎么样呢？

TCP会通过一种叫流控制的方式来确保两点之间传输数据的平衡。

拥堵控制

TCP有一种内置的机制能够控制数据包的延迟率及丢包率不会太高，以此来确保服务的质量（QoS）。

举例来说，和流控制机制能够避免发送方压垮接收方一样，TCP会通过控制数据包的传输速率来避免拥堵的情况。

好了，介绍完TCP的基本工作原理，到实践的时候了。为了测试 TCP服务器，我们可以使用Telnet工具。

Telnet

Telnet是一个早期的网络协议，旨在提供双向的虚拟终端。在SSH出现前，它作为一种控制远程计算机的方式被广泛使用，如远程服务器管理。它是TCP协议上层的协议（不要惊讶）。

尽管从21世纪初就基本不用Telnet了，但如今绝大部分主流的操作系统都内置了telnet客户端（见图6-1）：

```
$ telnet
```

绝大部分Telnet使用的是23号端口。要是通过该端口连接到服务器（telnet host.com 23 或者就简单地写成telnet host.com），就说明在通过TCP使用Telnet协议。

72

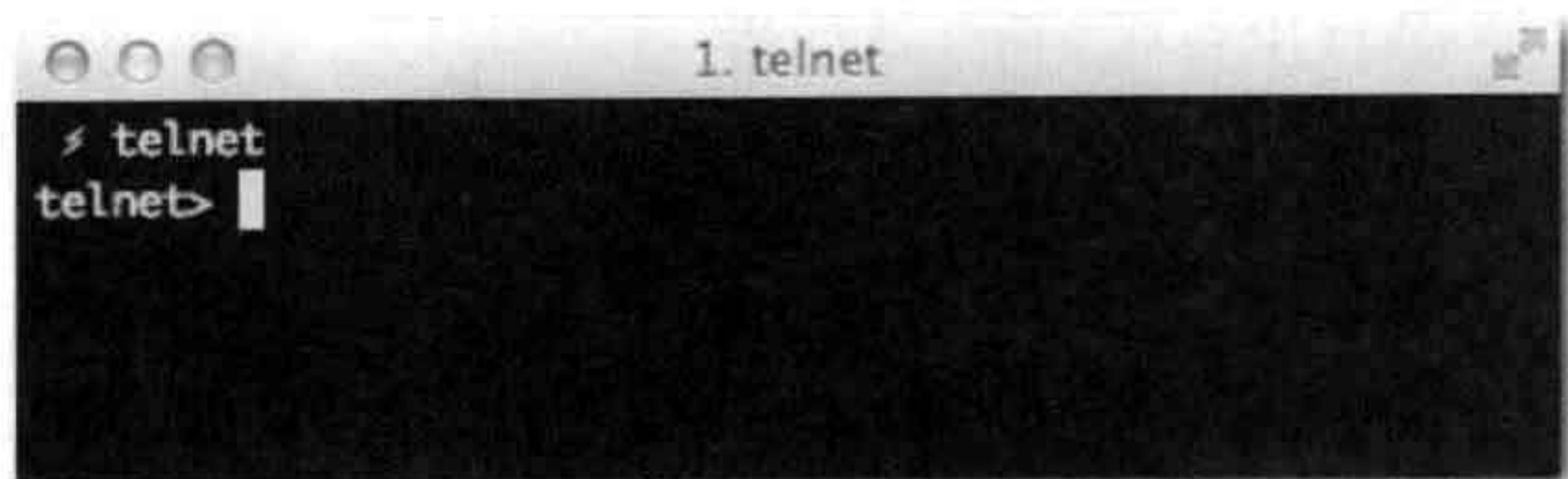


图6-1：运行telnet工具

不过，在本例中，telnet客户端还有更加有意思的功能。发送数据时，客户端要是发现服务器使用的并不是Telnet，这时，它不会关闭连接或者显示错误信息，相反，它会自动降级到低层的纯TCP模式。

所以，要是telnet到Web服务器会怎么样呢？要一探究竟，我们来看下面这个例子。

首先，我们用Node.js来写一个简单的hello world Web服务器，并监听 3000端口：

```
# web-server.js
require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('<h1>Hello world</h1>');
}).listen(3000);
```

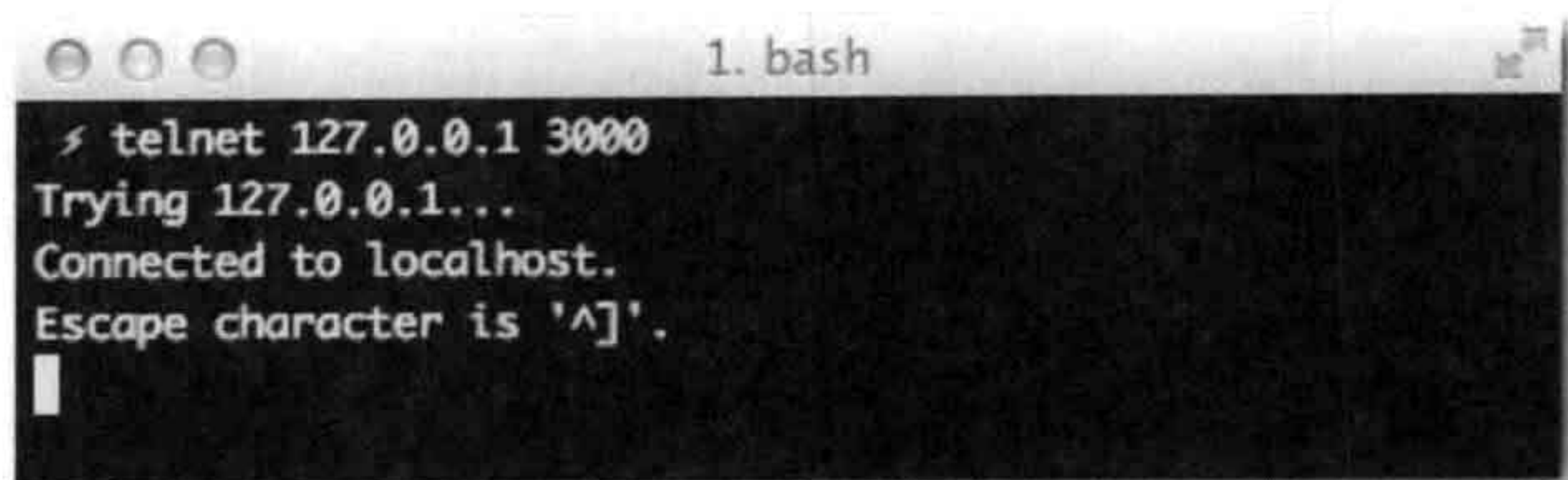
接下来通过node server.js运行上述代码。要确保运行正常，可以打开一个典型的HTTP客户端——浏览器来查看，如图6-2所示。



图6-2：浏览器通过本地3000端口建立了一个TCP连接，然后开始使用HTTP协议进行通信

现在来实现客户端部分。使用telnet来建立一个连接（见图6-3）：

```
$ telnet localhost 3000
```



73

图6-3：telnet允许在终端手动建立一个TCP连接

尽管根据图6-3中所示的结果，看上去已经工作正常了，但是，服务器端的“Hello World”消息并未到客户端这里。原因在于，要往TCP连接中写数据，必须首先创建一个HTTP请求，这就是套接字（socket）。在终端输入GET / HTTP/1.1然后按两下回车键。

如图6-4所示，这个时候，服务器端的响应就出现了！

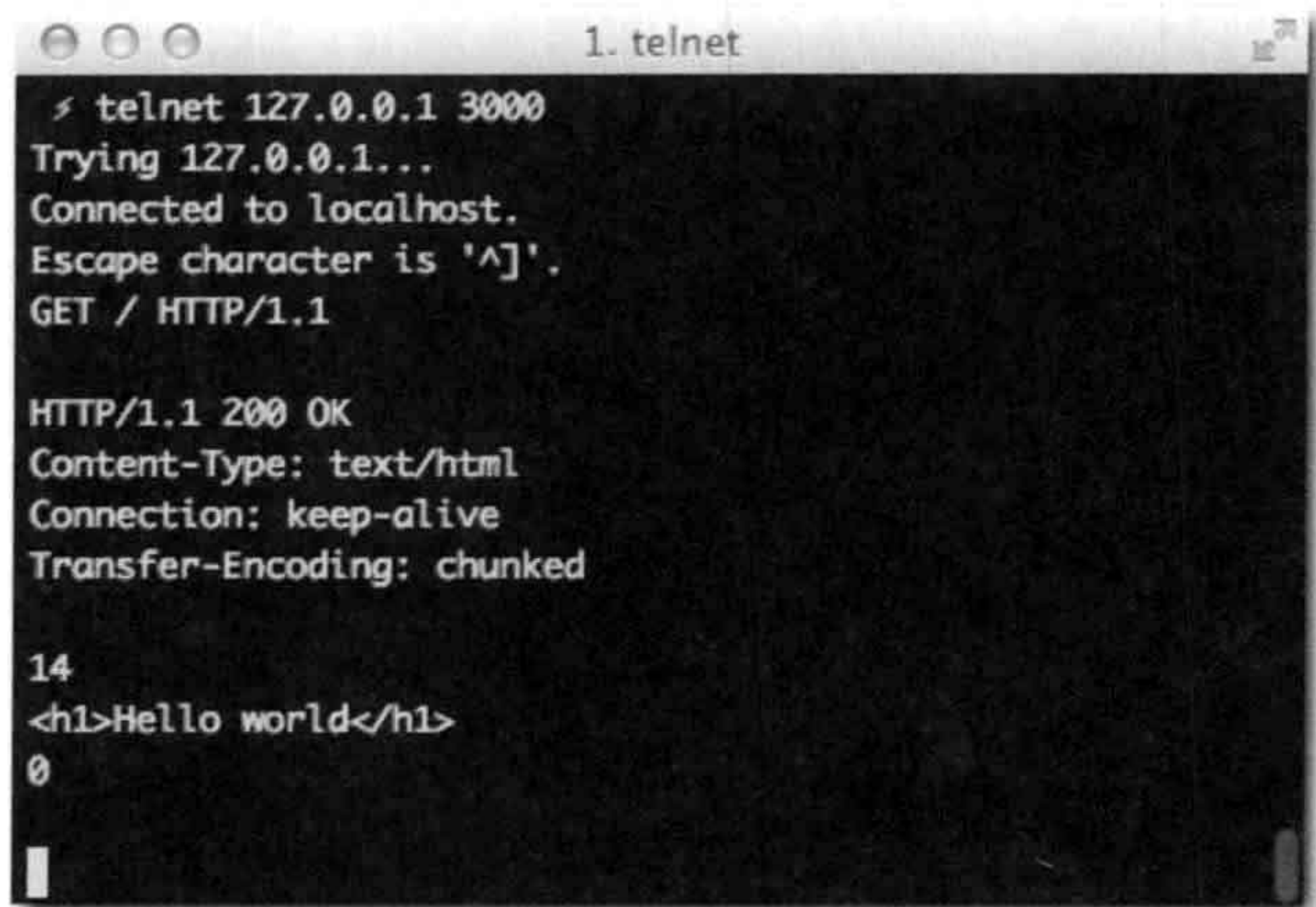


图6-4：Mac上IRC客户端（Textual.app）实践。Textual.app实现了基于TCP套接字的IRC协议

我们来总结一下：

- 成功建立了一个TCP连接。
- 创建了一个HTTP请求。
- 接收到了一个HTTP响应。
- 测试了一些TCP的特性。到达的数据和在Node.js中写的一样：先写了Content-Type响应头，然后是响应体，最后所有的信息都按序到达。

74 基于TCP的聊天程序

正如此前介绍的，TCP的主要目的就是为两台计算机通过提供可靠的网络进行通信。

本章选择一个聊天应用作为TCP的“Hello World”程序，因为，它是展示TCP最简单的方式之一。

下面，我们来创建一个基本的TCP服务器，任何人都可以连接到该服务器，无须实现任何协议或者指令：

- 成功连接到服务器后，服务器会显示欢迎信息，并要求输入用户名。同时还会告诉你当前还有多少其他客户端也连接到了该服务器。
- 输入用户名，按下回车键后，就认为成功连接上了。
- 连接后，就可以通过输入信息再按下回车键，来向其他客户端进行消息的收发。

为什么要按下回车键呢？事实上，Telnet中输入的任何信息都会立刻发送到服务器。按下回车键是为了输入\n字符。在Node服务器端，通过\n来判断消息是否已完全到达。所以，这其实是作为一个分隔符在使用。

换句话说，这里按下回车键和输入字符a没有什么区别。

创建模块

按照惯例，我们先来创建一个项目目录和package.json文件：

```
# package.json
{
  "name": "tcp-chat"
  , "description": "Our first TCP server"
  , "version": "0.0.1"
}
```

接着运行npm install测试一下。结果会输出一个空行，因为项目没有任何依赖。

理解NET.SERVER API

接下来，创建一个包含如下代码的index.js文件：

```
/**
 * 模块依赖
 */

var net = require('net')
/**
 * 创建服务器
 */
```



```
var server = net.createServer(function (conn) {  
  // handle connection  
  console.log('\033[90m  new connection!\033[39m');  
});  
  
/**  
 * 监听  
 */  
  
server.listen(3000, function () {  
  console.log('\033[96m  server listening on *:3000\033[39m');  
});
```

注意，上述代码中为`createServer`指定了一个回调函数。该函数在每次有新的连接建立时都会被执行。

为了验证，我们来运行上述代码，启动一个TCP服务器。当`listen`执行时，它会将服务器绑定到3000端口，并最终在终端打印出一段消息。

```
$ node index.js
```

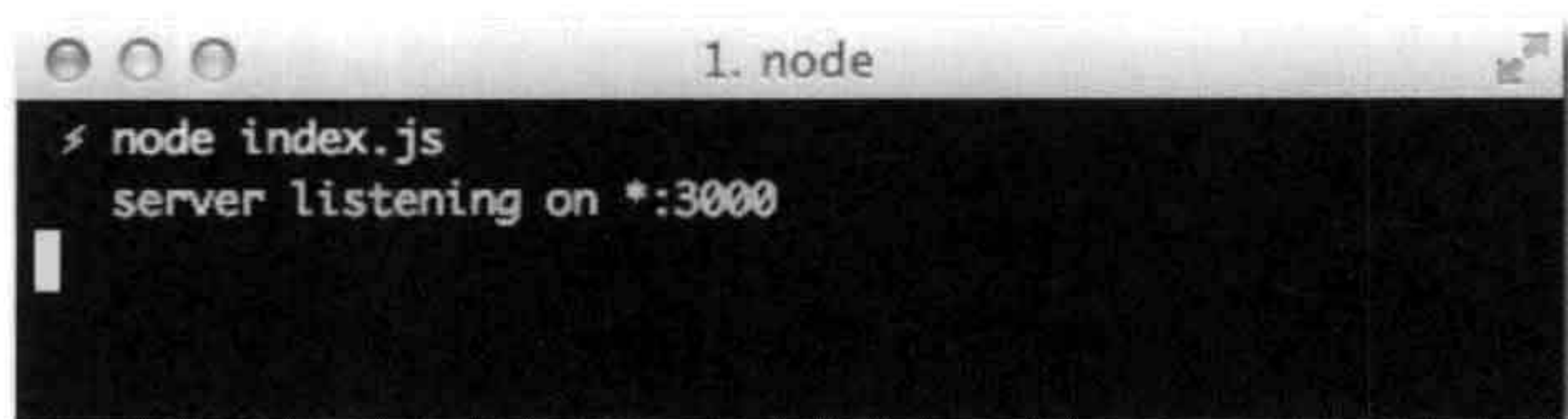


图6-5：服务器会绑定到3000端口，然后显示一段成功启动的信息
现在，我们尝试用telnet来进行连接：

```
$ telnet 127.0.0.1 3000
```

在图6-6中，能看到该命令，而且，“new connection!”消息也会显示出来。

如你所见，这个例子和此前HTTP的hello world程序类似。现在再去理解“HTTP是建立在TCP协议之上的”就不难了吧。不过，本例中，我们会创建自己的协议。

`createServer`回调函数会接收一个对象，该对象是Node中一个很常见的实例：流（Stream）。本例中，它传递的是`net.Stream`，该对象通常是既可读又可写的。

最后，还有一个重要的方法就是`listen`，它可以将服务器绑定到某个端口上。由于该方法也是异步的，所以也接收一个回调函数。

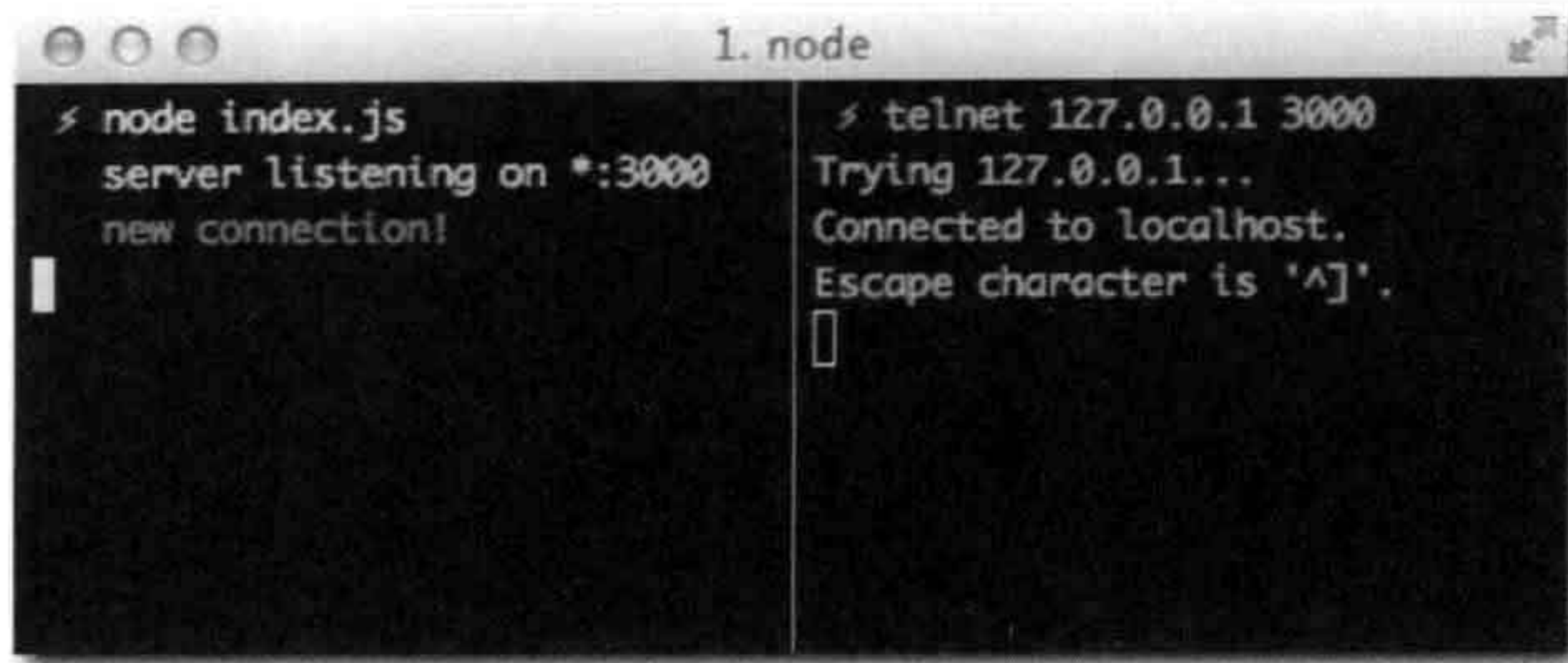


图6-6：左侧显示的是服务器进程的状态。右侧则是客户端，连接到服务器端后，服务器端就输出了“新连接（new connection）！”

接收连接

正如此前项目描述中的，一旦连接建立，就会向客户端回写欢迎语和当前连接数。

我们先在回调函数外部添加一个计数器：

```
/**
 * 追踪连接数
 */
```

```
var count = 0;
```

接着，我们需要修改回调函数内容，把计数器递增和打印出欢迎语的逻辑添加上去：

```
var server = net.createServer(function (conn) {
  conn.write(
    '\n > welcome to \033[92mnode-chat\033[39m!'
    + '\n > ' + count + ' other people are connected at this time.'
    + '\n > please write your name and press enter: '
  );
  count++;
});
```

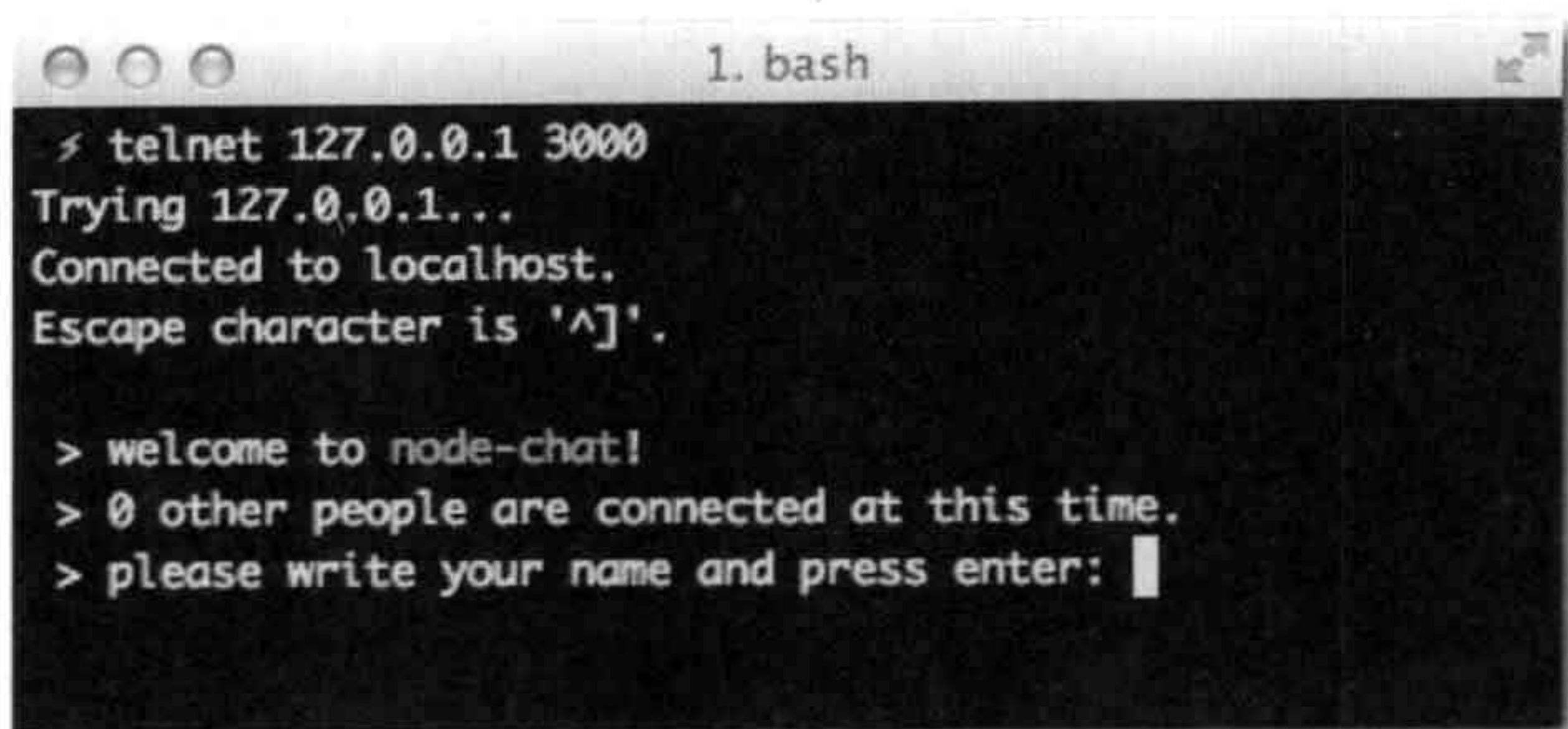
如上述代码所示，我们仍旧使用shell转义码来控制输出文本的颜色。

接着重启服务器进行测试：

```
$ node index
```

再次通过telnet去连接（见图6-7）：

```
$ telnet 127.0.0.1 3000
```

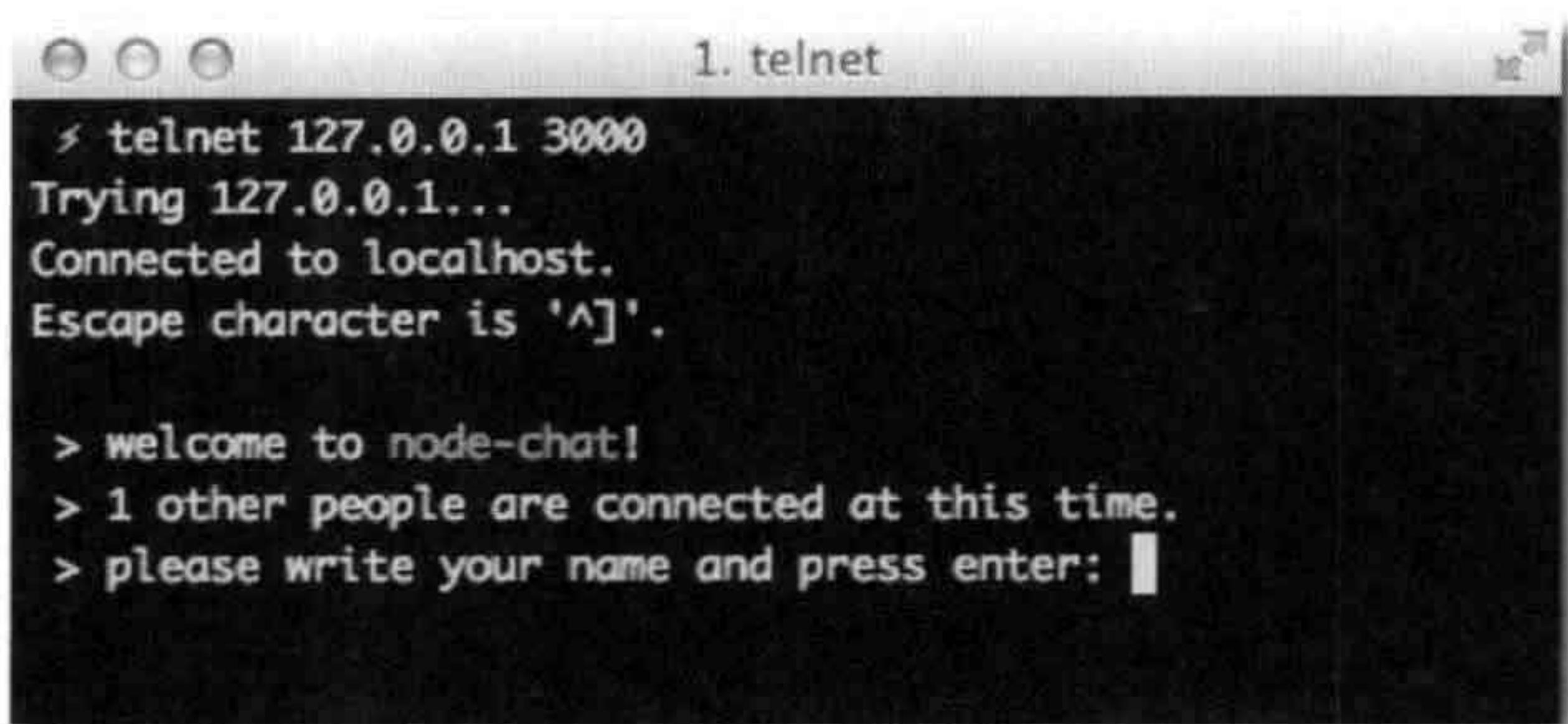



```
1. bash
> telnet 127.0.0.1 3000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

> welcome to node-chat!
> 0 other people are connected at this time.
> please write your name and press enter: |
```

图6-7：现在客户端连接成功后能够接收到更多信息了

如图6-8所示，当第二个客户端连接进去后，计数器就增加了一个！



```
1. telnet
> telnet 127.0.0.1 3000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

> welcome to node-chat!
> 1 other people are connected at this time.
> please write your name and press enter: |
```

图6-8：连接计数器正确反映了当前连接数

当客户端请求关闭连接时，计数器变量就要进行递减操作：

```
conn.on('close', function () {
  count--;
});
```

当底层套接字关闭时，Node.js会触发close事件。Node.js中有两个和连接终止相关的事件：end和close。前者是当客户端显示关闭TCP连接时触发。比如，当你关闭telnet时，它会发送一个名为“FIN”的包给服务器，意味着要结束连接。

当连接发生错误时（触发error事件），end事件不会触发，因为服务器端并未收到“FIN”包信息。不过这两种情况下，close事件都会被触发，所以，上述例子中使用close事件会比较好。

在Mac上，可以通过按下Alt + [来结束一个telnet连接，Windows上可以用Ctrl +]。

data事件

我们已经能在客户端打印出一些信息了，接着我们来看看如何处理客户端发送的数据。

首先要处理的数据是用户输入的呢称（nickname），所以，我们从监听data事件开始。与

其他Node中的API一样，`net.Stream`同时也是一个`EventEmitter`。

为了进行测试，我们在服务器端的控制台输出客户端发来的数据：

```
var server = net.createServer(function (conn) {
  conn.write(
    '\n > welcome to \033[92mnode-chat\033[39m!'
    + '\n > ' + count + ' other people are connected at this time.'
    + '\n > please write your name and press enter: '
  );
  count++;

  conn.on('data', function (data) {
    console.log(data);
  });
  conn.on('close', function () {
    count--;
  });
});
```

然后，我们启动该服务器，并使用客户端进行连接。如图6-9所示，在客户端输入一些数据。看左侧部分，当输入数据时，服务器端就直接通过`console.log`将其打印出来了。

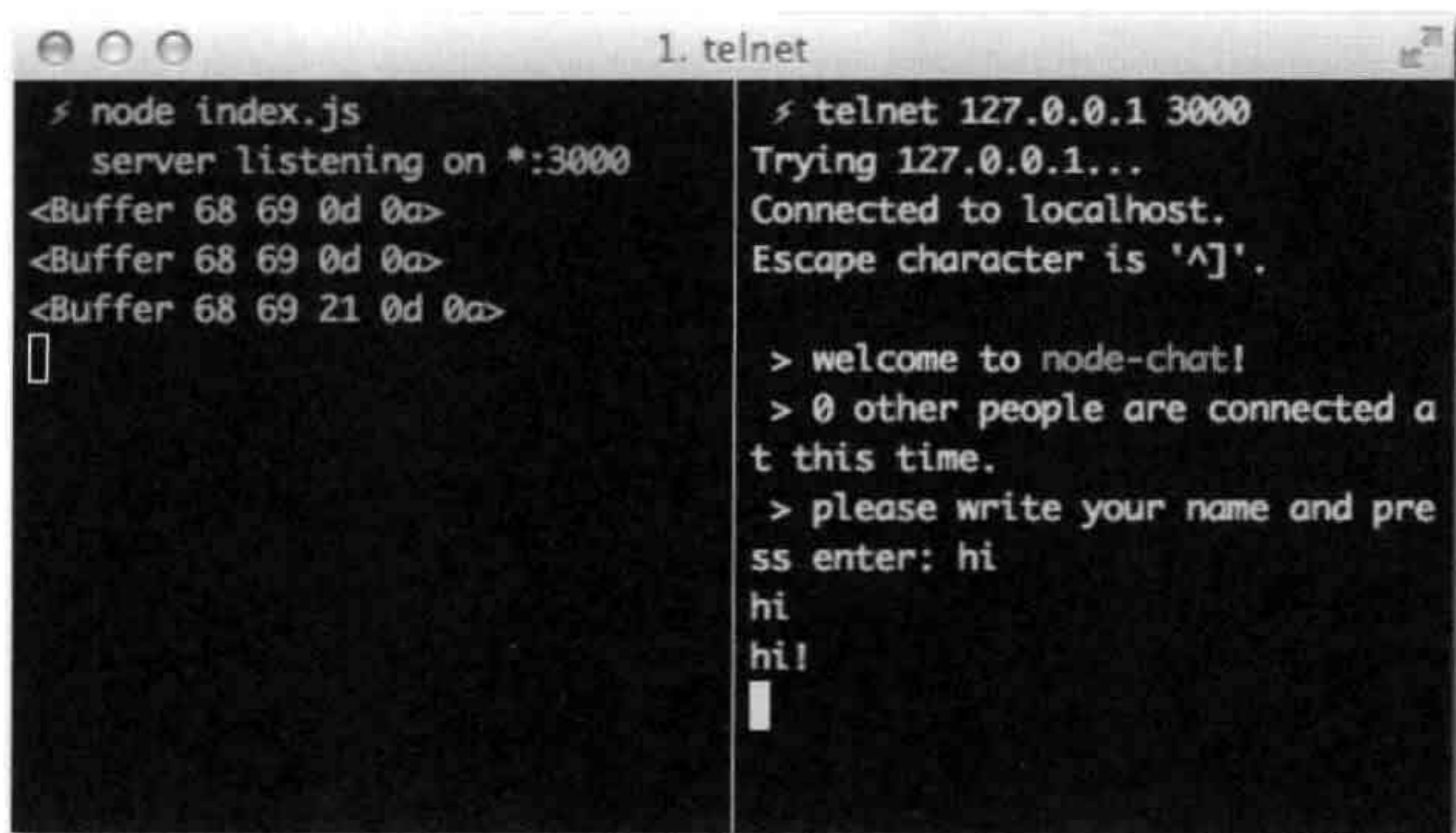


图6-9：图中左侧部分显示了右侧发送过来数据相对应的Buffer对象

如图6-9中所示，我们接收到的数据是一个Buffer。还记得此前介绍的，TCP是面向字节的协议吗？这里可以看得出来Node遵循了TCP的这一习惯！

这个时候，有多种选择可以获取字符串形式的数据。可以通过调用Buffer对象上的`.toString('utf8')`来获取utf8编码的字符串。

不过，由于我们这里无须获取utf8之外其他编码格式的数据，我们可以通过`net.Stream#setEncoding`方法来设置编码（如图6-10所示）：


```
# index.js
...
conn.setEncoding('utf8');
```

79

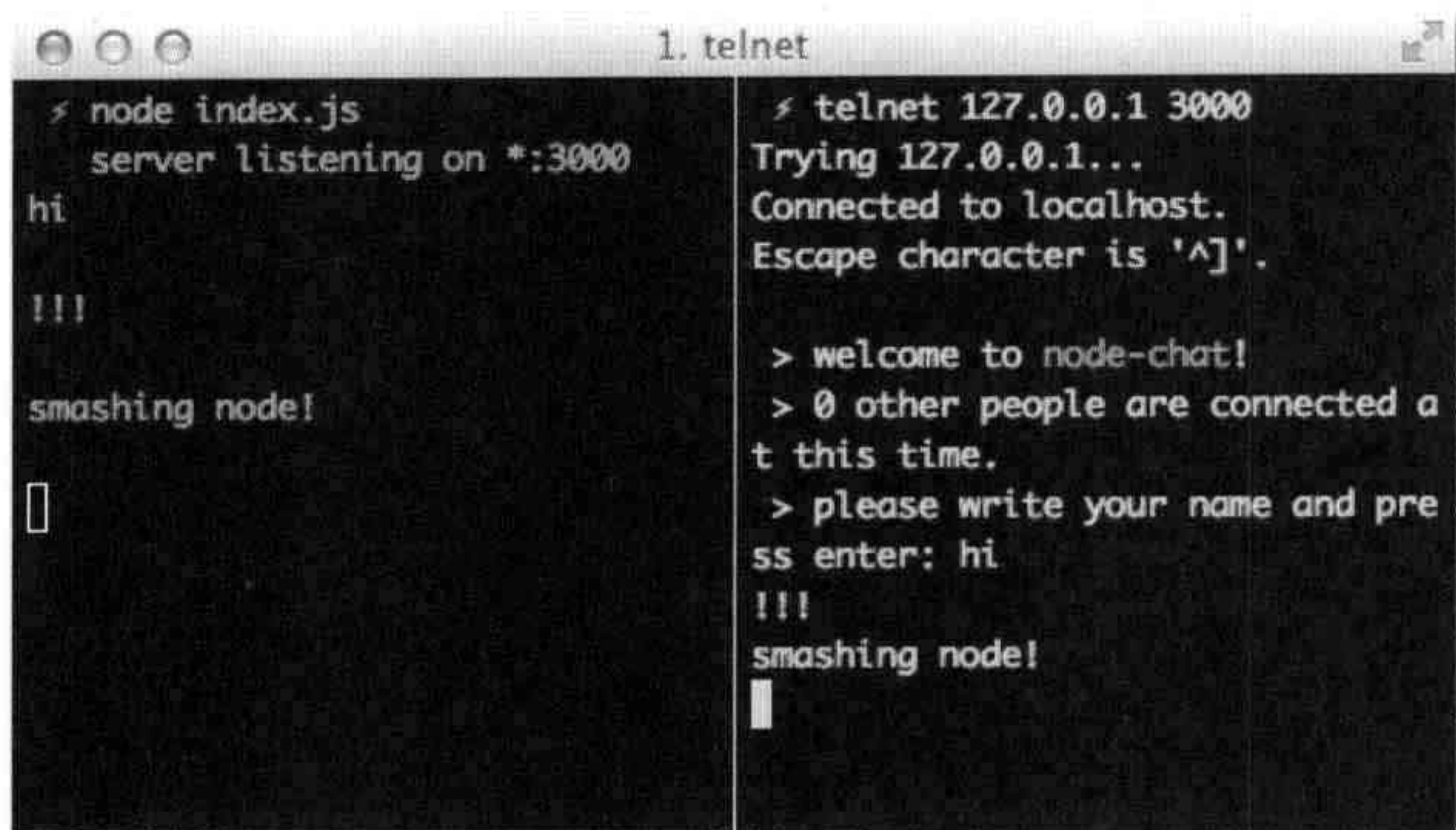


图6-10：聊天信息现在以utf8编码的字符串形式输出在左侧

好了，至此，我们已经可以让客户端和服务端进行交互了，接下来我们要让更多的客户端加入聊天。

状态以及记录连接情况

此前定义的计数器通常称为状态。因为，在本例中，两个不同连接的用户需要修改同一个状态变量，这在Node中称为共享状态的并发。

为了能够向其他连接进来的客户端发送和广播消息，我们需要对该状态进行扩展，来追踪到底谁连接进来了。

当客户端输入了昵称后，就认为该客户端已经连接成功，并可以接收消息了。

首先，我们要记录设置了昵称的用户。为此，我们需要引入一个新的状态变量，`users`：

```
var count = 0
    , users = {}
```

然后，在每个连接中，再引入一个`nickname`变量：

```
conn.setEncoding('utf8');
```

```
// 代表当前连接的昵称
var nickname;
```

```
conn.on('data', function (data) {
```

接收到数据时，确保将`\r\n`（相当于按下回车键）清除：

```
// 删除回车符
data = data.replace('\r\n', '');
```

80

对于尚未注册的用户，需要进行校验。如果昵称可用，则通知其他客户端当前用户已经连接进来了（见图6-11）：

```
// 接收到的第一份数据应当是用户输入的昵称
if (!nickname) {
  if (users[data]) {
    conn.write('\033[93m> nickname already in use. try again:\033[39m ');
    return;
  } else {
    nickname = data;
    users[nickname] = conn;

    for (var i in users) {
      users[i].write('\033[90m > ' + nickname + ' joined the room\033[39m\n');
    }
  }
}
```

```
1. telnet
$ telnet 127.0.0.1 3000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

> welcome to node-chat!
> 0 other people are connected at this time.
> please write your name and press enter: test
> test joined the room
> woot joined the room
[]

$ telnet 127.0.0.1 3000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

> welcome to node-chat!
> 1 other people are connected at this time.
> please write your name and press enter: test
> nickname already in use. try again: woot
> woot joined the room
[]
```

图6-11：消息已经广播给所有加入聊天服务器的客户端了

如果是已经验证通过的用户，那么接下来收到的数据就认为是聊天消息，需要显示给所有其他客户端：

```
else {
  // 否则，视为聊天消息
  for (var i in users) {
    if (i !== nickname) {
      users[i].write('\033[96m > ' + nickname + ':\033[39m ' + data + '\n');
    }
  }
}
```



```

    }
}

```

使用 `i !== nickname` 来确保消息只发送给除了自己以外的其他客户端。

图6-12展示了两台客户端连接进来之后，互相聊天的场景。

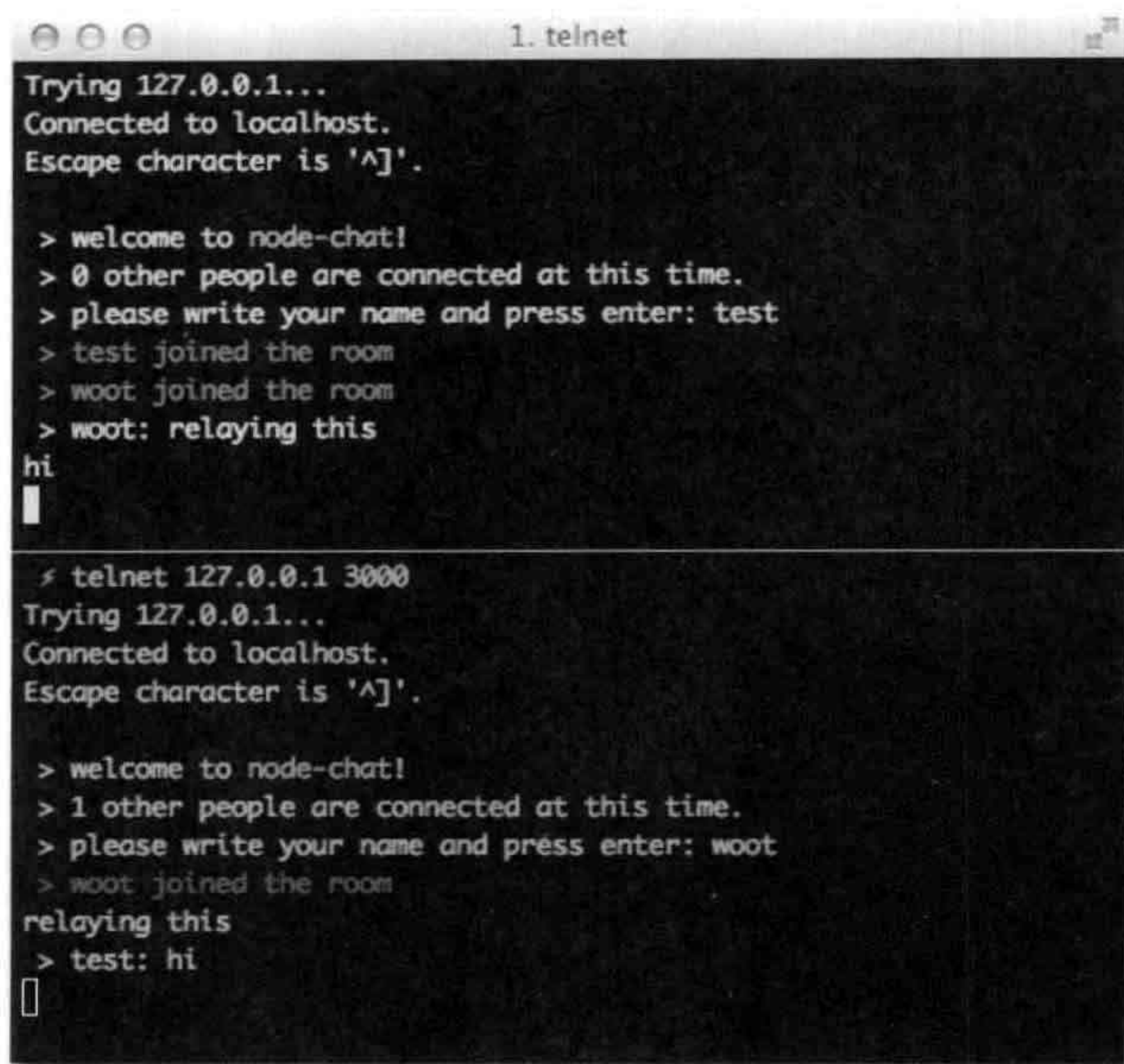


图6-12：展示了两台客户端连进来之后，互相聊天的场景
成功交换聊天消息后，我们来圆满完成这个程序。

圆满完成此程序

当有人断开连接时，我们需要清除 `users` 数组中对应的元素：

```

conn.on('close', function () {
  count--;
  delete users[nickname];
});

```

用户断开时通知其他用户也是个不错的想法。由于我们时不时就需要给所有的用户广播消息，所以，把这部分逻辑抽象出来也不错：

```

// ...
function broadcast (msg, exceptMyself) {
  for (var i in users) {
    if (!exceptMyself || i !== nickname) {
      users[i].write(msg);
    }
  }
}

```



```
}
```

```
conn.on('data', function (data) {
  // . . .
```

这样一来，就可以重用上面的函数进行消息广播了，简洁易懂，一目了然：

```
broadcast('\033[90m > ' + nickname + ' joined the room\033[39m\n');
// . . .
broadcast('\033[96m > ' + nickname + ':\033[39m ' + data + '\n', true);
```

我们把它加到close处理器中（见图6-13）：

```
conn.on('close', function () {
  // . . .
  broadcast('\033[90m > ' + nickname + ' left the room\033[39m\n');
});
```

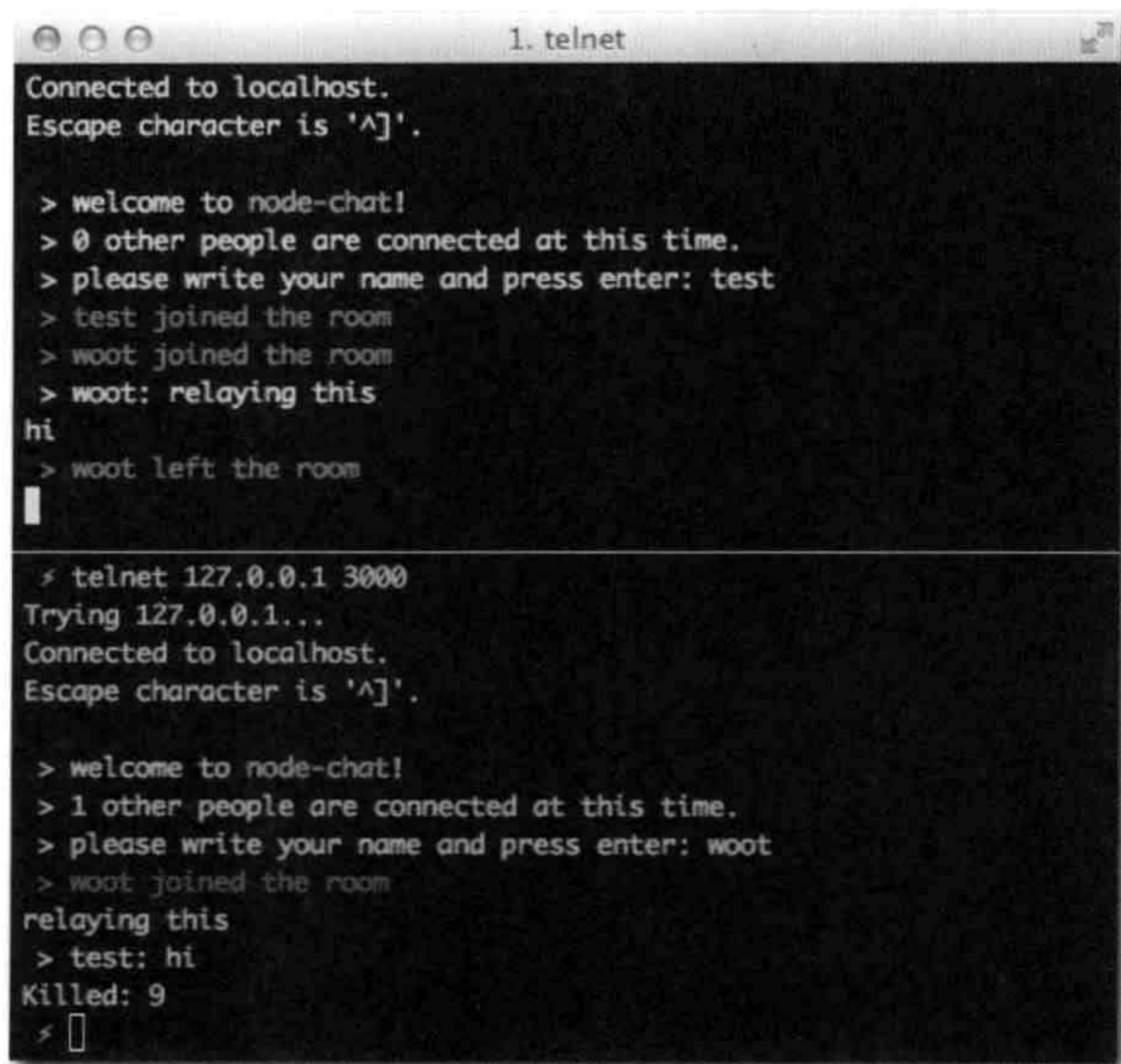


图6-13：在关掉第一个客户端时，会强制关闭该连接，接着其他客户端就会收到当前用户离开的消息完成！

83 在成功实现了一个TCP服务器后，我们需要进一步学习在Node.js中如何实现一个TCP客户端。

客户端API和其他像Twitter这样用来查询Web服务的HTTP客户端类似，因此，完全明白这些非常重要。

一个IRC客户端程序

IRC是因特网中继聊天（Internet Relay Chat）的缩写，它也是一项常用的基于TCP的协议。它通常被使用在如图6-14所示的客户端程序中，作为连接到IRC服务器的客户端。

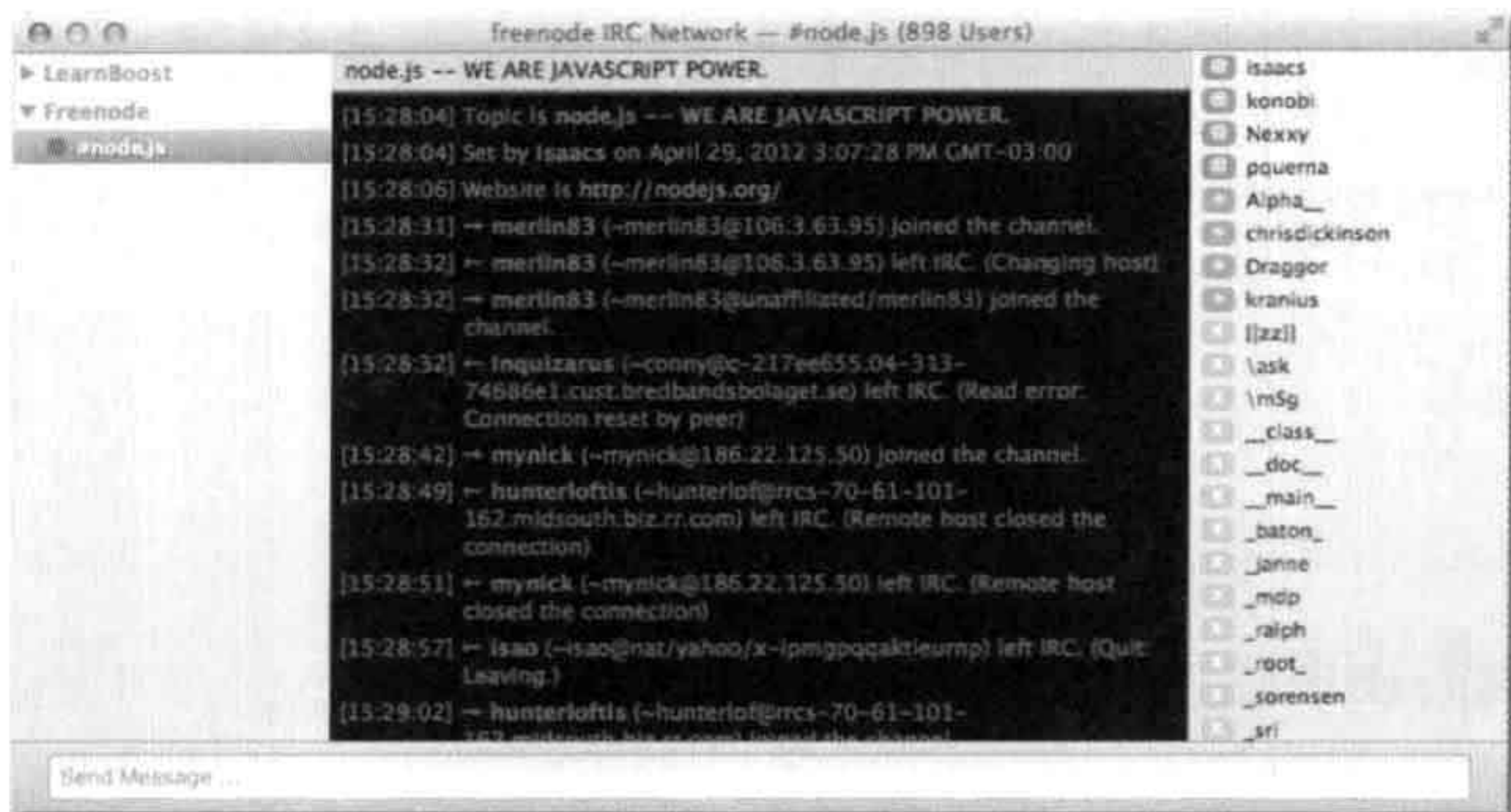


图6-14：正在运行的Mac上一款IRC客户端程序（Linkinus）。Linkinus实现了基于TCP套接字的IRC协议

此前我们成功地构建了一个TCP服务器，这次我们来写一个TCP客户端。

构建一个实现IRC协议的客户端意味着，需要实现通过一组命令来实现与IRC服务器进行“通信”，进行数据的交换。

比如，要设置昵称，客户端可以发送如下指令：

```
NICK mynick
```

IRC是一项非常直观、简单的协议。通过一些简单的命令就可以和现有的应用以及服务器（如图6-14中展示的）进行通信。

下面我们就来学习如何用Node.js书写一个非常基本的客户端，实现加入一个聊天室以及回复消息功能。

创建模块

和往常一样，我们先来创建一个项目目录，并在该目录下创建一个package.json文件：

```
{
  "name": "irc-client"
  , "description": "Our first TCP client"
  , "version": "0.0.1"
}
```

接着运行npm install。因为项目没有任何依赖的模块，所以控制台应当会输出一个空行。

理解NET#STREAM API

和createServer一样，net API提供了另外一个名为connect的方法，如下所示：

```
net.connect(port[, host], callback))
```

如果提供了回调函数，就等于是监听了该方法返回的对象上的connect事件。

```
var client = net.connect(3000, 'localhost');
client.on('connect', function () {});
```

上述代码和下面的代码是一样的：

```
net.connect(300, 'localhost', function () {});
```

另外，和此前使用的API类似，我们还可以监听data和close事件。

实现部分IRC协议

我们先来初始化IRC客户端。然后尝试去登录irc.freenode.net中的#node.js频道：

```
var client = net.connect(6667, 'irc.freenode.net')
```

设置编码为utf-8：

```
client.setEncoding('utf-8')
```

连接上服务器后，发送自己的昵称。除此之外，还要写上服务器要求的USER命令。采用类似如下所示的方式来发送数据：

```
NICK mynick
USER mynick 0 * :realname
JOIN #node.js
```

85

具体代码如下所示：

```
client.on('connect', function () {
  client.write('NICK mynick\r\n');
  client.write('USER mynick 0 * :realname\r\n');
  client.write('JOIN #node.js\r\n');
});
```

这里要注意，需要在每条命令后加上\r\n分割符。这和此前在Telnet下按下回车键是一样的。\\r\\n也是HTTP协议用来区分头信息的分隔符。

测试实际的IRC服务器

打开一个IRC客户端（如Windows上的mIRC、Linux上的xChat或者Mac上的Colloquy/Linkinus），并将其指向：


```
irc.freenode.net  
#node.js
```

启动后，观察mynick是否连接上了：

```
![] (http://f.cl.ly/items/1b3g3i1w120Z2U082I3G/Image%202011.11.07%202:31:35%20AM.png)
```

小结

本章介绍了一个简单的网络客户端的实现，成功使其与一台并非自己实现的TCP服务器进行通信。

作为习题，你可以尝试着去完成如下部分：监听data数据并尝试解析接收到的数据，将其他用户发送到#node.js频道的消息打印出来。你还可以进一步尝试着结合现有的代码去实现一个IRC机器人，来自动对命令做出响应。比如，当某人说日期（data）时（可以在data事件中检测到），你就可以输出new Date()的结果。

紧接着在下一章，你会学到HTTP这种Node.js因此闻名的Web协议。现在你对TCP中数据传递、数据块的构建已经很清楚了，那么接下来学习TCP上层的HTTP API，一定会让你对Node.js中的核心功能有更加深入的了解。