10.1 部署

目前，要想运行Node.js应用，只需要在命令行中运行如下代码：

```
node script_name.js
```

在开发阶段采用这种方式还不错。然而，要想部署应用到生产服务器中，则需要添加额外的可靠层，当应用崩溃时能够提供帮助。虽然我们力争避免服务器出现bug，但难免会有意外，因此需要服务器尽快从问题中恢复。

接下来让我们看一些解决方案。

端口号

本书中大部分示例应用使用的端口号数字比较大，基本上都是8080。因为某些操作系统（特别是UNIX/Mac OS X系统）小于1024的端口号需要超级用户权限。因此，对于简单的调试和测试，8080端口比一般的80端口更合适。

但是，当我们部署应用时，通常又想要它们运行在80端口上，因为80端口是Web浏览器的默认端口。因此，需要以超级用户身份运行脚本。有两种方式可以完成这件事：

1）以超级用户的身份登录系统并运行程序。

2）使用sudo命令，在实际运行时能够以超级用户权限执行进程。

对于以上两种策略，我一般都使用后者。

10.1.1 级别：基础

最基本的级别就是使用无限循环运行脚本。该循环会在Node进程崩溃时自动重启Node进程。在UNIX和Mac计算机上运行bash，

其脚本如下所示：

```
#!/bin/bash
while true
do
    node script_name.js
done
```

在Windows平台上可以使用batch文件完成类似的工作，如下所示（保存该脚本到run_script.bat文件或其他类似文件中并运行）：

```
: loop
node script_name.js
goto loop
: end
```

这些脚本会确保Node进程在崩溃或者意外终止时立即重启。

更进一步，我们可以使用tee命令将node进程的标准输出写入到日志文件中。通常在命令行中使用|（"pipe"）操作符，将进程的输出pipe到tee命令中，然后tee再重新打印输出到标准输出，并写入到特定的日志文件中，如下所示：

```
node hello_world.js | tee -a /var/server/logs/important.log
```

tee的-a参数表示添加到特定日志文件而不是简单地覆盖这个文件。因此，可以通过运行如下脚步进一步改进部署：

```
bash ./run_script.sh | tee -a /var/server/logs/important.log
```

Window平台可以使用下面的命令：

```
run_script.bat | tee -a /var/server/logs/important.log
```

要获取更多关于tee的信息并真正运用该技术，请参见：

Windows下的Tee（和其他有用的命令）

Windows下的shell并不强大（虽然PowerShell正朝着正确的方向发展）。为了解决这个问题，人们通常会下载外部工具包，让

shell拥有UNIX平台上（如Linux或Mac OS X）一些非常棒的脚本功能。

对于本章提到的tee编程，有两个工具可供选择：

- http://getgnuwin32.sourceforge.net/
- http://unxutils.sourceforge.net/

也可以搜索"Windows tee"，查找针对Windows的特定tee工具。事实上，许多本书中描述的类UNIX工具基本上在Windows上都有同样功能的替代品或者版本。

使用screen监控输出

虽然在Windows上总是处于登录状态并很容易回到桌面查看Node服务器的运行状况，但是在UNIX服务器上（例如Linux或Mac OS X）经常需要注销而服务器仍然保持运行。问题在于当这样做时，会失去标准输出（stdout）处理，输出也会丢失，除非将输出写入到一个日志文件中。即使这样，日志文件也经常会错过一些重要信息，这些重要信息主要是Node或者操作系统打印出来的相关问题的内容。

为了解决这个问题，大多数UNIX系统都支持一个叫做screen的神奇小工具，即使已经注销，它也能够运行程序，就像一直拥有一个控制终端一样（也称之为tty）。不同平台下的安装不一样，但Mac OS X已经默认安装，而大多数的Linux平台可以通过类似apt-get screen的操作获取它。

现在，在运行Node服务器之前，可以运行screen并在其中运行Node。要离开screen，可以按下键序列Ctrl+A Ctrl+D。要重新进入screen，可以运行命令screen–r（恢复）。如果有多个screen session在运行，每个screen都有一个名字，想要恢复则需要输入对应session的名字。

我们可以使用screen运行每个独立的Node服务器，这是每隔一

段时间回来并查看运行情况的最佳方式。

10.1.2 级别：Ninja

前面的脚本对运行中的应用非常有用，可以在不会经常崩溃的低流量网站上使用它们。但是，它们存在两个严重的缺陷：

1）如果一个应用进入持续崩溃的状态，即使重启了，这些脚本也会盲目地不断地重启它，导致变成“丢失的进程”。服务器永远不可能正常启动，一直处于“假死”状态。

2）虽然这些脚本可以保持进程启动和运行，但当应用遇到麻烦时很难通知我们，特别是它们使用太多内存的时候。同时，还存在Node进程消耗过多内存或者遇到其他系统资源问题的情况。如果能够检测出这些情况并终止进程，随后再次启动，那就非常棒了。

因此，在下一节，我们会看到关于两个部署的高级选项，在这一点上，UNIX和Windows的区别开始变大。

Linux/Mac

以前版本的脚本（以伪代码展示）非常有效：

```
while 1
      run node
end
```

现在可以升级这些脚本来做更多的事情，脚本如下所示：

```
PID = run node
while 1
      sleep a few seconds
```

```
        if PID not running
           restart = yes
        else
          check node memory usage
          if using too much memory
             restart = yes

        if restart and NOT too many restarts
           PID = restart node
end
```

要使上面的代码工作，有两个关键任务需要执行：

1）获得最新启动的Node服务器的进程ID（pid）。

2）弄清楚Node进程使用了多少内存。

关于第一点，可使用pgrep命令，Mac和大部分UNIX平台都支持。当带有-n参数时，它会返回给定进程名的最新实例的进程ID。在计算机运行多个Node服务器的情况下，也可以使用-f参数来告诉它要匹配的脚本的名字。因此，要获得最新创建的运行script_name.js的Node进程，可以使用：

```
pgrep -n -f "node script_name.js"
```

现在，要获得进程的内存使用量，需要用到两个命令。首先是ps命令，当提供wux（在某些平台是wup，请检查相关文档）参数时，会显示进程的虚拟内存消耗量，同时显示当前驻留内存的大小（这是我们想要的）。ps wux $pid的输出如下所示：

```
USER    PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
marcw  1974   6.6  3.4  4507980 571888   ??  R    12:15AM  52:02.42 command name
```

我们需要这个输出的第二行第六列（RSS）。要得到该数据，首先需要运行awk命令获得第二行，然后再次运行获得第六列，如下所示：

```
ps wux $PID | awk 'NR>1' | awk '{print $6}'
```

获取当前进程的内存消耗量后，就可以判断进程是否消耗太多内

存，以及有没有必要终止并重启该进程。

在这里我不会打印shell脚本的所有代码，你可以查看GitHub仓库第10章中的代码，叫做node_ninja_runner.sh。可以使用如下方式运行：

```
node_ninja_runner.sh server.js [extra params]
```

当然，如果要监听例如80或443端口号，必须确保已经提升为超级用户权限。

Windows

在Windows上，可靠地运行并监控应用的最佳方式是使用Windows服务。Node.js本身并没有设置成Windows服务，但幸运的是，可以使用一些技巧使它的行为看起来像是一个Windows服务。

通过使用一个叫做nssm.exe（Non-Sucking Service Manager）的小程序和一个叫做winser的npm模块，可以将Node的Web应用作为服务进行安装并通过Windows管理控制台（Windows Management Console）管理它们。要完成这些设置，需要做两件事情：

1）在package.json中为Web应用安装新的自定义操作。

2）下载和安装winser，并让其工作。

对于第一步，仅需添加如下配置到package.json文件：

```
{
  "name": "Express-Demo",
  "description": "Demonstrates Using Express and Connect",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "3.x"
  },
   "scripts": {
    "start": "node 01_express_basic.js"
  }
}
```

对于第二步，只需如下进行：

```
C:\Users\Mark\LearningNode\Chapter10> npm install winser
```

（或者添加winser到依赖中）然后，当准备好将应用安装成一个服务时，我们可以前往包含package.json文件的项目文件夹下并运行

```
C:\Users\Mark\LearningNode\Chapter10> node_modules\.bin\winser -i
The program Express-Demo was installed as a service.
```

要卸载服务，只需运行winser–r，如下所示：

```
C:\Users\Mark\LearningNode\Chapter10> node_modules\.bin\winser -r
The service for Express-Demo was removed.
```

现在可以前往Windows管理控制台，启动、停止、暂停，或者根据需求管理该服务！

10.2 多处理器部署：使用代理

之前提到多次，Node.js作为一个单线程的进程其运行会非常高效：脚本的所有代码都在同一个线程里执行，并使用异步回调来提高CPU效率。

那么，你一定会问：那在多CPU的系统中会怎么处理呢？如何获取服务器的最大能力？许多现代服务器都是强大的8-16核机器，可以的话我们都想使用。

幸好，这个问题的答案相当简单：即在每个想利用的内核上都运行node进程（见图10.1）。我们可以选择众多策略中的某一种将请求路由到各个不同的node进程中，就像匹配需求一样。

接下来面临的问题是，系统不是只有一个，而是n个Node进程在运行，而这些进程必须监听不同的端口（不可能要求多个用户监听同一个端口号）。那如何才能将来自mydomain:80的流量转入这些服务器呢？

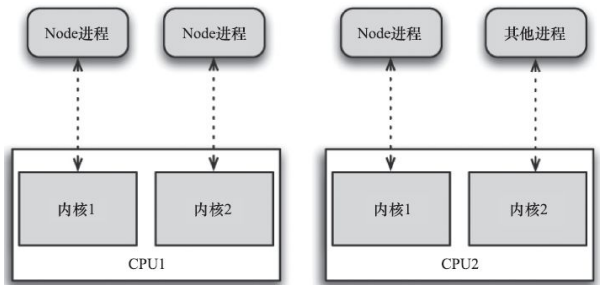

图10.1 为相同的应用运行多个node进程

我们可以通过实现一个简单的循环负载均衡器来解决这个问题，它提供一个运行特定Web应用的node服务器列表。接下来，每次将传入到该域的请求重定向到其中一个服务器中。当然，也可以使用更多高级策略，比如监听加载、可用性以及响应性。

要实现负载均衡器，首先要收集一个运行中的服务器列表。假设服务器有四个内核，分配其中三个给这个应用（保留第四个运行其他的系统服务）。在8081、8082、8083端口上运行这些应用服务器。那么，服务器列表就是：

```
{ "servers": [ {
    "host": "localhost",
    "port": "8081"
}, {
    "host": "localhost",
    "port": "8082"
}, {
    "host": "localhost",
    "port": "8083"
} ] }
```

这些简单服务器的代码展示在代码清单10.1中。这是本书中最简单的Web服务器，只有一些简单的代码用来从命令行获取需要监听的端口号（可以在第11章深入了解命令行参数）。

代码清单10.1　简单的HTTP服务器

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

if (process.argv.length != 3) {
    console.log("Need a port number");
    process.exit(-1);
}

var s = http.createServer(function (req, res) {

    res.end("I listened on port " + process.argv[2]);
});

s.listen(process.argv[2]);
```

我们可以在UNIX/Mac平台通过输入以下命令启动三个服务器：

```
$ node simple.js 8081 &
$ node simple.js 8082 &
$ node simple.js 8083 &
```

对于Windows平台，可以简单地在三个不同的命令提示符中运行下面的三个命令来启动服务器：

```
node simple.js 8081
node simple.js 8082
node simple.js 8083
```

现在可以使用npm模块http-proxy来构建自己的代理服务器。package.json如下所示：

```
{
  "name": "Round-Robin-Demo",
  "description": "A little proxy server to round-robin requests",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "http-proxy": "0.8.x"
  }
}
```

代理服务器的代码相当简单，如代码清单10.2所示。它维护了一个可用的服务器数组，然后遍历这些可用的服务器，将每个传入到该服务的请求都转发到服务器中。

代码清单10.2　循环代理负载均衡器（roundrobin.js）

```
var httpProxy = require('http-proxy'),
    fs = require('fs');

var servers = JSON.parse(fs.readFileSync('server_list.json')).servers;

var s = httpProxy.createServer(function (req, res, proxy) {
    var target = servers.shift();           // 1. Remove first server
    proxy.proxyRequest(req, res, target);   // 2. Re-route to that server
    servers.push(target);                   // 3. Add back to end of list
});

s.listen(8080);
```

为了让所有代码生效，需要运行node roundrobin.js，接下来就可以开始查询，如下所示：

```
curl -X GET http://localhost:8080
```

如果多次运行该命令，我们可以看到输出结果为当前服务器监听的实际端口号。

多服务器和会话

在不同的CPU和服务器上运行多个服务器是分散负载的好办法，这给我们提供了一个提高可扩展性的低廉途径。（服务器超负荷了？那就再加个服务器！）但这会产生并发症，即需要在有效使用这些服务器前能够定位到它们：目前seesion数据通过本地MemoryStore对象存储在每个进程中（请参见7.4.5节）。这会导致一个严重的问题，如图10.2所示：

因为每个Node服务器都在自己的内存存储中储存本身的session记录，当两个来自相同客户端的请求分配到两个不同的node进程时，它们对来自客户端的session数据的当前状态会产生疑惑并感到迷茫。我们有两种办法来解决这个问题：

1）实现一个更高级的路由，可以记住来自特定客户端的请求发送到哪一台Node服务器，并确保对于同一个客户端的所有请求持续地以相同方式进行路由。

2）将session数据使用的内存存储放到存储池中，这样所有Node进程都可以访问它。

我倾向于选择第二种解决方案，因为它是一个容易实现且并不复杂的解决方案。

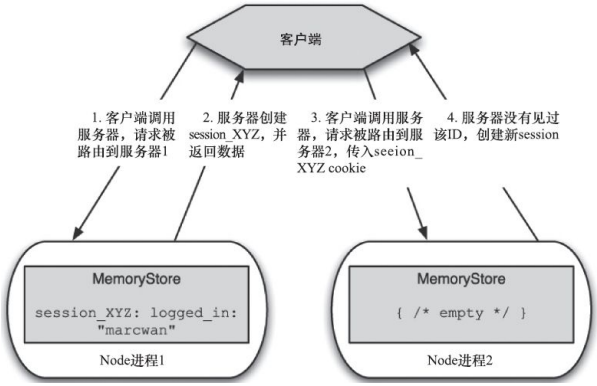

图10.2　多服务器实例和session数据

要实现该解决方案，我们首先需要选择一个内存存储池。最佳候选方案是memcached和Redis，两者都是基于内存的键/值存储（memory-based key/value stores），可以在不同计算机之间传递数据。本章使用的是memcached，因为它完全满足我们的需要而且非常轻量。设置基本上如图10.3所示。

在Windows上安装

对于Windows用户，可以安装memcached的二进制安装包，在互联网上可以搜索到很多安装包。我们也不需要最新或最强大的版本，1.2.x或之后的版本就可以。为了将其安装成服务，需要运行下面的脚本：

```
c:\memcached\memcached -d install
```

然后可以运行memcached服务，输入如下代码：

```
net start memcached
```

最后，我们可以编辑HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/memcachedServer并修改ImagePath为：

```
"C:/memcached/memcached.exe" -d runservice -m 25
```

设置memcached可用的内存量为25MB，通常这对开发已经足够了（当然，可以设置成任何想要的值）。它的监听端口为11211。

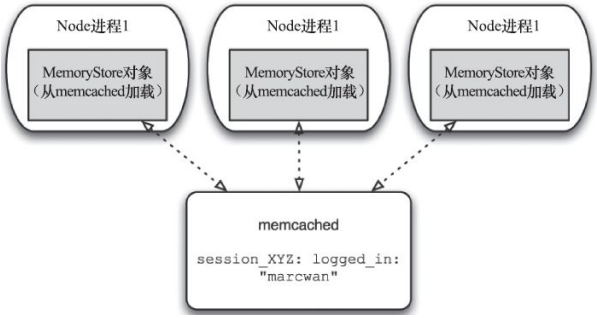

图10.3　使用基于内存的键/值存储来实现session

在Mac和类UNIX系统上安装

如果我们的系统有某种包管理器，则通常可以像下面一样使用（这可能需要sudo）：

```
apt-get install memcached
```

这样就可以了。如果我们想要通过源代码构建，首先需要从http://libevent.org上获取libevent。下载、构建以及安装（以超级用户身份）这个库。它应该安装在/usr/local/lib目录下。

接下来，访问memcached.org并下载最新的源代码包（.tar.gz文件）并再次以超级用户的身份配置、构建和安装memcached。

安装后要运行memcached，可以运行如下命令：

```
/usr/local/bin/memcached -m 100 -p 11211 -u daemon
```

该命令告诉服务使用100MB内存运行并监听11211端口，以daemon身份运行。通常以root身份运行会被拒绝。

在express中使用memcached

现在我们已经安装并运行memcached（假设是localhost的11211端口），那接下来需要为它配一个MemoryStore对象，这样seesion数据才可以使用memcached。幸运的是，Node.js社区相当活跃，已经有个叫做connect-memcached的npm模块。因此，我们只需添加下面的配置到package.json文件依赖中：

```
"connect-memcached": "0.0.x"
```

之后，可以修改创建session的代码，如下所示：

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

// pass the express obj so connect-memcached can inherit from MemoryStore
var MemcachedStore = require('connect-memcached')(express);
var mcds = new MemcachedStore({ hosts: "localhost:11211" });

var app = express()
    .use(express.logger('dev'))
    .use(express.cookieParser())
    .use(express.session({ secret: "cat on keyboard",
                           cookie: { maxAge: 1800000 },
                           store: mcds}))
    .use(function(req, res){
        var x = req.session.last_access;
        req.session.last_access = new Date();
        res.end("You last asked for this page at: " + x);
    })
    .listen(8080);
```

现在，所有的Node服务器的session数据都配置成使用同一个MemoryStore，无论请求是哪个服务器在处理，都可以获取到正确的信息。

10.3 虚拟主机

多年来，在同一个服务器上运行多个网站对于Web应用平台都是一个主要的需求。幸运的是，当通过express构建应用时，Node.js在这方面为我们提供了两种相当可行的解决方案。

通过向HTTP请求中添加"Host:"头来实现虚拟主机，这是HTTP/1.1的主要特性之一（见图10.4）。

10.3.1 内置支持

express直接内置了运行多虚拟主机的功能。要让它工作，需要为每一个支持的主机创建一个express服务器，然后为虚拟服务器提供一个“主”服务器，用来将请求转发到合适的虚拟服务器。最后，使用vhost连接中间件组件将所有的服务器都组合起来，如代码清单10.3所示：

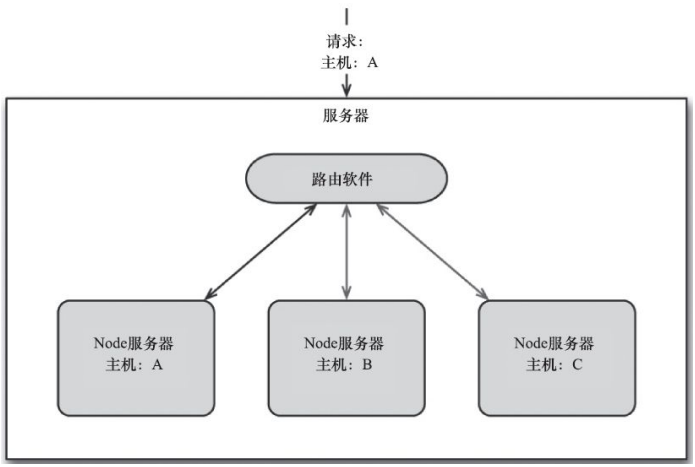

图10.4 虚拟主机的基本概念

代码清单10.3 express中的虚拟主机（vhost_server.js）

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

var one = express();
one.get("/", function(req, res){
    res.send("This is app one!")
});

// App two
var two = express();
two.get("/", function(req, res){
    res.send("This is app two!")
});

// App three
var three = express();
three.get("/", function(req, res){
    res.send("This is app three!")
});

// controlling app
var master_app = express();

master_app.use(express.logger('dev'));
master_app.use(express.vhost('app1', one))

master_app.use(express.vhost('app2', two));
master_app.use(express.vhost('app3', three));

master_app.listen(8080);
```

测试虚拟主机

测试虚拟机是个不小的挑战，因为所有服务器都运行在localhost上，而我们需要获取服务器并识别出具体请求的服务器名。

如果通过命令行使用curl来测试，只需添加一个头信息到请求中来指定我们请求的是哪个主机，如下所示：

```
curl –X GET -H "Host: hostname1.com" http://localhost:8080
```

然而，如果想在浏览器中进行测试，则需要修改计算机的DNS条目。最常见的方法就是编辑/etc/hosts（UNIX/Mac机器）或者C:\Windows\System32\drivers\etc\hosts，这需要高级权限才能完成。在UNIX/Mac上，使用sudo打开编辑器；而在Windows上，只需简单地使用Notepad，但要确保使用管理员权限打开。

然后我们就可以使用如下格式在文件中添加条目了：

```
127.0.0.1       hostname
```

这里，127.0.0.1是本地计算机的IPv4地址，接下来，只需把想要的名称映射到该地址即可。我们可以添加任意多的主机映射（例如，app1、app2、app3等）。

对于这两种方式，无论是命令行或者是浏览器，用来测试虚拟主机都没有问题。

10.3.2 代理服务器支持

虽然express内置支持虚拟主机，但一个进程运行多个Web应用程序还是会存在风险。如果有人胡作非为或者出现问题，则所有的站点都会出现问题。

为了解决这种情况，我们来看看另外一种管理虚拟主机的方法，这再次使用到了http-proxy模块。该模块功能非常强大，可以很轻松地将请求路由到不同的虚拟主机，只需要几行代码即可。事实上，这种路由方式是基于站点主机名的，我们可以运行代码清单10.4，如下所示：

代码清单10.4 代理服务器虚拟主机（proxy_vhosts_server.js）

```
var httpProxy = require('http-proxy');

var options = {
  hostnameOnly: true,
  router: {
    'app1': '127.0.0.1:8081',
    'app2': '127.0.0.1:8082',
    'app3': '127.0.0.1:8083'
  }
}

var proxyServer = httpProxy.createServer(options);
proxyServer.listen(8080);
```

我们为代理服务器提供主机列表，用来寻找主机并将请求路由到这些主机中。要测试该功能，只需简单地运行三个不同的应用服务器，然后运行代理服务器，请求就会自动寻找到目标。

10.4 使用HTTPS/SSL保障项目安全

对于应用中处理敏感数据的那部分，例如用户密码、个人数据、银行账户或付款信息，实际上任何在本质上是私人的东西，我们都应该使用SSL加密的HTTPS传输来保护应用。虽然创建SSL连接和加密传输会带来额外的开销，但是安全上得到的收益却更重要。

要为应用添加SSL/HTTPS支持，首先需要生成一些测试证书，并为应用程序添加加密传输的支持。对于后者，我们也有两种方式来实现：express的内置支持或者使用代理服务器。

10.4.1 生成测试证书

要让加密传输在我们的开发机器上正常工作，需要生成一些测试证书，包括两个文件——privkey.pem和newcert.pem——私有密钥和对应的证书。所有的UNIX/Mac机器都带有叫做openssl的工具，可以用它来生成这两个文件。

对于Windows用户，可以访问http://gnuwin32.sourceforge.net/packages/openssl.htm并下载Win32版本的openssl.exe安装程序。然后就可以和其他平台一样运行命令了。

要生成这两个证书文件，运行下面的三个命令：

```
openssl genrsa -out privkey.pem 1024
openssl req -new -key privkey.pem -out certreq.csr
openssl x509 -req -days 3650 -in certreq.csr -signkey privkey.pem -out newcert.pem
```

有了这两个文件，就可以在应用中用它们来工作。注意，绝不能在生产环境中使用它们，如果尝试在浏览器中查看受这些证书保护的站点，就会听到异常大声的危险声音，警告这个网站是不被信任的。通常必须从受信任的证书颁发机构购买生产环境的证书。

10.4.2 内置支持

不必惊讶，通过Node提供的内置https模块，Node.js和express提供了对SSL/HTTPS数据流的支持。实际上我们是运行一个https模块服务器来监听HTTPS端口（默认是443，但在开发阶段，可以用8443替代，以避免需要提升Node进程的权限），然后当加密数据流（encrypted stream）经过握手和创建之后，https模块会将请求传输到express服务器。

我们创建HTTPS服务器，并将对站点签名的私有密钥和证书文件的地址作为可选参数传入。也可以将它传给express服务器，它能在加密建立之后发送数据。具体实现如代码清单10.5所示：

代码清单10.5　express/https模块的SSL支持（https_express_server.js）

```
var express = require('express'),
    https = require('https'),
    fs = require('fs');

// 1. Load certificates and create options
var privateKey = fs.readFileSync('privkey.pem').toString();
var certificate = fs.readFileSync('newcert.pem').toString();
var options = {
    key : privateKey,
    cert : certificate
}
// 2. Create express app and set up routing, etc.
var app = express();
app.get("*", function (req, res) {
      res.end("Thanks for calling securely!\n");
});


// 3. start https server with options and express app.
https.createServer(options, app).listen(8443, function(){
    console.log("Express server listening on port 8443");
});
```

可以通过在Web浏览器的地址栏打开https://localhost:8443来访问这些加密页面。首次浏览它们时，会提示它们是不安全的。

10.4.3　代理服务器支持

正如在本章中探究的其他特性一样，我们也可以使用强大的http-proxy模块来处理SSL/HTTPS传输。对比内置的HTPPS支持，使用这个模块有两个明显的优势，它能够让应用服务器像普通的HTTP服务器一样运行，让它们“隐藏”在HTTPS代理服务器背后并让我们从乏味的加密工作中释放出来。也可以运行使用之前见过的循环负载均衡，或者看看有没有其他创建方式来设置我们的配置。

这种支持的使用方法和之前express的例子没有太大差异：创建一个Node的内置HTTPS服务器类的实例，再创建一个用来路由请求到普通HTTP服务器（应用服务器）的代理服务器的实例。然后运行HTTPS服务器，当它建立安全连接后，会将请求传给代理服务器，再传递给应用服务器。具体实现如代码清单10.6所示。

代码清单10.6　http-proxy SSL支持（https_proxy_server.js）

```
var fs = require('fs'),
    http = require('http'),
    https = require('https'),
    httpProxy = require('http-proxy');

// 1. Get certificates ready.
var options = {
    https: {
        key: fs.readFileSync('privkey.pem', 'utf8'),
        cert: fs.readFileSync('newcert.pem', 'utf8')
    }
};

// 2. Create an instance of HttpProxy to use with another server
var proxy = new httpProxy.HttpProxy({
    target: {
        host: 'localhost',
        port: 8081
    }
});

// 3. Create https server and start accepting connections.
https.createServer(options.https, function (req, res) {
    proxy.proxyRequest(req, res)
}).listen(8443);
```

上示代码看起来有很多的路由和重定向，但Node开发团队非常注重性能，这些组件也相当轻便，所以在为Web应用处理请求时不会增加太多明显的延迟。

10.5 多平台开发

Node.js的另一优势是，它不仅能很好地支持类UNIX和Mac系统，也可以运行在Windows机器上。在编写本书时，对所有的示例都测试过，并且在Windows上运行这些示例也没有任何问题。

实际上，我们可以让人们在任何他想要的平台上进行项目开发，只要做好应付两个可能突发的小问题的准备即可：配置上的不同和路径区别。

10.5.1 位置和配置文件

Windows和类UNIX操作系统在不同的地方保存文件。一个能减轻该问题带来的影响的方法就是使用配置文件来存储这些地址。这样，代码会变得足够灵活，它知道去哪里查找东西，而不必根据不同平台分别处理。

实际上，在第8章和第9章中就开始使用这种技术，我们将数据库配置信息保存到一个叫做local.config.js的文件。现在可以继续使用这种技术，并扩大它的使用范围，通常可以保存任何会影响应用运行的信息到这个文件。事实上，这种技术并不仅限于保存文件的位置或路径，还可以在文件中配置端口号或构建类型：

```
exports.config = {
    db_config: {
        host: "localhost",
        user: "root",
        password: "",
        database: "PhotoAlbums",

        pooled_connections: 125,
        idle_timeout_millis: 30000
    },

    static_content: "../static/",
    build_type: "debug"      // show extra output
};
```

实际上，通常我们需要做的就是确认该文件处于版本控制系统之下（例如Github），而且是以local.config.js-base而不是local.config.js保存在源代码树中。要运行应用，只需拷贝这个基础文件并修改成local.config.js，针对本地的运行设置更新成合适的值，然后运行应用。任何时候想使用这些本地配置变量，只需添加如下代码：

```
var local = require('local.config.js');
```

然后，在代码中直接引用变量local.config.variable_xyz即可。

10.5.2 处理路径差异

Windows和类UNIX操作系统的另外一个主要区别就是路径。当面对诸如需要require处理程序，而这些处理程序却存放在项目的子文件夹下等问题时，我们应该如何编码呢？

好消息是，绝大多数情况下，我们会发现Node接受斜杠（/）字符。当我们从相对路径（例如"../path/to/sth"）引用模块时，Node会和你期望的一样去“工作”。即使是使用fs模块的API，大部分的API也能够处理两个平台之间不同的路径类型。

对于那些必须去处理不同路径格式的情况，可以使用Node.js的path.sep属性，它能够灵活使用数组的连接和分割，例如：

```
var path = require('path');
var comps = [ '..', 'static', 'photos' ];
console.log(comps.join(path.sep));
```

Node.js的process全局对象总是能够告诉我们当前的运行平台，可以通过如下代码进行查询：

```
if (process.platform === 'win32') {
    console.log('Windows');
} else {
    console.log('You are running on: ' + process.platform);
}
```

10.6 小结

在本章中，我们了解了如何在生产环境中运行Node应用，涵盖了脚本执行、负载均衡和多进程应用。本章还展示了如何基于SSL使用HTTPS保护Web应用安全，最后描述了多平台开发，结论就是它们并没有想象中的那么吓人。

现在我们有一系列漂亮的工具集合来构建和运行异步应用。不过，在下一章，我们会看到Node.js更酷的用法，会发现Node.js还有一堆同步API，能让Node.js完成命令行编程和脚本编程。

第11章　命令行编程

本书花了大量的篇幅和时间来阐述Node.js平台的强大之处，它可以创建异步、非阻塞应用。而在Node中使用传统的同步、阻塞IO编程同样出色——编写命令行程序。事实表明，编写JavaScript脚本十分有趣，很多人开始使用Node编写日常脚本和其他小程序。

在本章中，我们会学习到在Node中运行命令行脚本的基本知识，包括一些常用的同步文件系统API等。然后会学习如何利用缓冲和无缓冲输入与用户交互，最后会看一下进程的创建和执行。

11.1 运行命令行脚本

在类UNIX、Mac操作系统以及Windows平台下，从命令行运行Node.js脚本时有很多选项。因此，Node.js脚本表现得与操作系统命令几乎一模一样，并且可以将命令行参数传递给Node。

11.1.1 UNIX和Mac

在类UNIX操作系统中，如Linux或者Mac OS X，绝大部分用户使用的是Bourne Shell，即sh（或者另外一个流行的变型——bash）。尽管shell脚本语言功能齐全、性能强大，但是如果有时使用JavaScript来编写脚本的话，则会更有趣。

在shell中运行Node程序有两种方式。其中最简单的运行方式和平常运行其他程序没有任何不同——指定node程序和需要运行的脚本（事实上，.js扩展名一般是可选的）：

```
node do_something
```

第二种方式则是直接修改文件的第一行代码，使其变为可执行文件，如下所示：

```
#!/usr/local/bin/node
```

在计算机科学中，文件中第一行的前两个字符（#!）一般被称为shebang。在文件中存在shebang的情况下，类Unix操作系统的程序载入器会分析Shebang后的内容，将这些内容作为解释器指令，并调用该指令，将载有shebang的文件路径作为该解释器的参数。在前面的示例中，操作系统会在/usr/local/bin目录下查找可执行的node，运行Node并将文件路径作为参数传递给Node。Node解释器会忽略第一行中以#!开头的代码，因为它知道这些信息是给操作系统用的。

所以，可以在Node中写出如下所示的可执行代码：

```
#!/usr/local/bin/node
console.log("Hello World!");
```

要让该程序可运行，需要使用chmod命令，将其标记为可执行文件：

```
chmod 755 hello_world
```

现在就可以直接运行该脚本啦，输入

```
$ ./hello_world
Hello World!
$
```

但是，这种实现方式还有一点小问题：当脚本文件拷贝到其他电脑中，而node解释器却没有在/usr/local/bin目录下而是在/usr/bin目录下会怎样呢？通常的做法是修改shebang行的内容，如下所示：

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

env命令会查询PATH环境变量，找到第一个node程序实例然后运行。所有的UNIX/Mac平台在/usr/bin目录下都存在env命令，因此，只要系统PATH变量中包含node，这种方式会更便捷。

如果经常在各种操作系统的可执行目录或者其他安装软件的目录下闲逛，就可以发现一些常用的程序，实际上，就是一些脚本。

小贴士：Node解释器非常智能，即使在Windows平台下也可以省略#!语法，而写出来的脚本仍然可以被识别并运行！

11.1.2 Windows

微软的Windows平台从来就没有过一个出色的脚本环境。虽然在Windows Vista和Windows 7中都引入了PowerShell，相较于以前有了很大的提高，但还是不如类UNIX平台下的shell强大和易用。好消息是，这并不会产生太大的影响，因为我们只是想启动并运行脚

本，仅此而已。而所有现代（或半现代）版的Windows平台满足这些需求绰绰有余。

我们还可以使用批处理文件（带有.bat扩展名的可执行文件）运行这些脚本。比如前文中所见的hello_world.js文件，可以创建一个hello_world.bat文件来执行它，如下所示：

```
@echo off
node hello_world
```

默认情况下，Windows会输出批处理文件中所有的命令。@echo off用来禁止输出。现在，要运行程序，只需要简单地输入

```
C:\Users\Mark\LearningNode\Chapter11\> hello_world
Hello World!
C:\Users\Mark\LearningNode\Chapter11\>
```

但是，该程序中有一个bug！如果hello_world.bat和hello_world.js文件在同一个文件夹下，但却在其他目录下执行该批处理文件，如下所示：

```
C:\Users\Mark\LearningNode\Chapter11\> cd \
C:\> \Users\Mark\LearningNode\Chapter11\hello_world
module.js:340
    throw err;
          ^
Error: Cannot find module 'C:\hello_world'
    at Function.Module._resolveFilename (module.js:338:15)
    at Function.Module._load (module.js:280:25)
    at Module.runMain (module.js:492:10)
    at process.startup.processNextTick.process._tickCallback (node.js:244:9)
```

由于没有指定hello_world.js的完整路径，因此，Node找不到该文件。只需要修改hello_world.bat脚本就可以修复这个问题，如下所示：

```
@echo off
node %~d0\%~p0\hello_world
```

这里利用了两个便捷的批处理文件的宏：%~pXX和%~dXX，即分别指定第n个参数（即第0个或者运行脚本的名称）的盘符和路径。现在，在计算机中任何地方运行hello_world.bat文件都能正常工作：