CHAPTER

14 Redis

既然你已经学会了如何通过Node.js来使用两大主流数据库——MongoDB和MySQL，那么接下来是时候介绍Redis了。 259

Redis是一种数据库，不过更准确地来说，它更像一台结构化的数据服务器，从定义上来说相比MySQL更接近MongoDB。

和操作表中的行或者集合中的文档不同，在Redis中是通过键来访问数据的。因此，可以将Redis想象成是通过如下所示的JavaScript对象的方式来存储数据的： 260

```
{
    'key': 'some value'
  , 'key.2': 'some other value'
}
```

不过，正因为它是一个结构化的数据服务器，能存储的值自然不仅仅是简单的字符串。如下都是Redis支持的数据类型：

- 字符串（string）
- 列表（list）
- 数据集（set）

- 哈希（hash）
- 有序数据集（sorted set）

然而，Redis和MongoDB最显著的不同点就是，Redis中的文档结构总是扁平的。举例来说，即使一个键包含类似哈希的JavaScript对象，却不能包含像MongoDB支持的那种嵌套的数据结构。

另一个不同点在于持久化数据方式的不同。Redis设计的初衷是内存存储，搭配可配置的磁盘持久化思路，所以速度很快。有一点很重要，需要记住：持久化到磁盘是很重要的，因为任何存储在内存中的东西都是不稳定的，而且会随着系统崩溃或者重启而受到影响。

这就意味着，对于敏感的数据系统而言（如处理金融交易），在开发之前，使用以及配置Redis都要非常仔细。尽管Redis的数据集（我们所操作的所有数据）存储在内存中，但是它有多重策略来将数据备份到磁盘中。问题就在于，其默认的配置和行为并不像MySQL那样非常适合数据敏感的系统。这就是Redis被冠以“仅仅算是一种数据库，但却不怎么耐用”的原因了。

如果你是第一次尝试部署Redis，我建议你先看看官网提供的几种不同持久化的选择：http://redis.io/topics/persistence。总的来说，你可以把Redis看作是简单、庞大、扁平（键—值）的JavaScript对象，其中值可以是特殊的数据类型（哈希、数据集、字符串等），它是为高速读写数据孕育而生的（所有数据都存储在内存中）。数据写入后的安全级别也是可配置的，但是，对某类系统而言，它并非是一个好的选择。

261 安装Redis

Redis官方以tar包的形式发布，包含所有源代码，支持Mac OS X和 Linux。如果你想自己编译，可以直接通过http://redis.io/download来下载最新的稳定版本。

而对于Mac来说，最简单的安装Redis的方式是通过homebrew：

```
$ brew install redis
```

或者通过ports：

```
$ sudo port install redis
```

与从源代码安装不同，这两个包管理器会在载入完毕后，自动安装。通过如下命令可以运行Redis服务器：

```
$ nohup redis-server &
```

对于Windows，也有非官方但维护得很好的版本。通过上面的 URL来查看最新文档了解对

于Windows的支持情况。

Redis查询语言

要开始学习Redis查询语言（换句话说，就好比是Redis中的SQL），首先确保服务器正常运行，随后执行如下命令：

```
$ redis-cli
```

和执行node一样，redis-cli其实就是和Redis服务器之间建立了telnet连接。换句话说，当建立到Redis的TCP连接时，接下来要执行的命令和Node客户端执行的几乎是一样的。

第一个要执行的命令是KEYS。在Redis中，命令都是大小写不敏感的，不过通常都约定了用大写方式。

和函数调用一样，命令可以接收任意数量的参数。如果执行KEYS时不添加任何参数，Redis就会抛出如下错误：

```
redis 127.0.0.1:6379> KEYS
(error) ERR wrong number of arguments for 'keys' command
```

KEYS接收一个匹配键的模式，并返回匹配到的键。*表示匹配所有键：

```
redis 127.0.0.1:6379> KEYS *
(empty list or set)
```

262

由于Redis刚装好，所以上述代码返回空。

现在我们来用SET命令将一个字符串赋值给某个键。Redis会返回OK：

```
redis 127.0.0.1:6379> SET my.key test
OK
```

运行GET my.key会返回刚刚保存的值：

```
redis 127.0.0.1:6379> GET my.key
"test"
```

再次执行KEYS *就会返回新的键：

```
redis 127.0.0.1:6379> KEYS *
1) "my.key"
```

绝大部分Redis指令都是依赖于数据类型的。使用GET和SET可以操作字符串，但是，使用GET却不能获取哈希类型的值。

数据类型

Redis简单的设计带来的最基本的好处之一就是开发者可以很容易地预测出性能。数据库

并不是一个黑盒，而是一个简单的进程，它将某些已知的数据结构存储到内容中，让其他程序通过简单的协议就能够获取到。

如果你在Redis官方文档手册中查看HEXISTS命令，你就会发现其中有一部分是关于时间复杂度的。

HEXISTS命令的时间复杂度是$O(1)$，也就是固定的时长。这就意味着，不管数据集有多大，执行HEXISTS命令总是需要这些时间。

如果查看SMEMBERS，就会发现它的时间复杂度是$O(n)$，也就是线性的时长，所需时间是随着数据集的大小而改变的。这就意味着Redis完成该指令所需的时间直接取决于数据有多少量。

由于Redis的对象模型大致就是一个大的扁平的JSON对象，所以，理解不同数据类型最简单的方式就是把它们想象成JavaScript中的数据类型。对于每一种数据类型，下面都会介绍与之类似的JS世界中的类型。

263 字符串

Redis中的字符串类似于JavaScript中的Number和String。

除了使用SET和GET外，还可以对数字进行递增和递减：

```
redis 127.0.0.1:6379> SET online.users 0
OK
redis 127.0.0.1:6379> INCR online.users
(integer) 1
redis 127.0.0.1:6379> INCR online.users
(integer) 2
```

哈希

在Redis中，哈希类似于子对象。不过和MongoDB不同的是，这些子对象只能局限于字符串形式的键和值。

要在Redis中存储用户信息，如下所示：

```
{
    "name": "Guillermo"
  , "last": "Rauch"
  , "age": "21"
}
```

由于键和值都是字符串（或者数字），所以，用哈希来描述这种数据结构最合适不过了。

正如此前所述，在Redis中，大对象中的所有数据都是通过唯一的键来获取的。要存储用户信息，我们需要将用户ID作为键的一部分来唯一确定存储的值。Redis数据库存储的数据如

下所示：

```
{
    "profile:1": { name: "Guillermo", "last": "Rauch", . . . }
  , "profile:2": { name: "Tobi", "last": "Rauch", . . . }
}
```

上述例子中是否使用冒号（:）取决于你自己。你也可以使用点、下画线或者干脆不用。只要保证在操作文档时每个键都能唯一，避免冲突，同时键名又包含了足够多的信息，能够从应用层面简单地获取到，就可以了。

操作哈希的基本命令是HSET：

```
redis 127.0.0.1:6379> HSET profile.1 name Guillermo
(integer) 1
```

这个命令就等于在JavaScript中设置一个键：

```
obj['profile.1'].name = 'Guillermo';
```

264

要获取一个指定哈希中所有的键和值，可以使用HGETALL，并提供一个键名：

```
redis 127.0.0.1:6379> HGETALL profile.1
1) "name"
2) "Guillermo"
```

Redis会返回一个包含了修改过的键和值的列表：

```
redis 127.0.0.1:6379> HSET profile.1 last Rauch
(integer) 1
redis 127.0.0.1:6379> HGETALL profile.1
1) "name"
2) "Guillermo"
3) "last"
4) "Rauch"
```

要在哈希中删除一个键，可以调用HDEL：

```
redis 127.0.0.1:6379> HSET profile.1 programmer 1
(integer) 1
redis 127.0.0.1:6379> HGETALL profile.1
1) "name"
2) "Guillermo"
3) "last"
4) "Rauch"
5) "programmer"
6) "1"
redis 127.0.0.1:6379> HDEL profile.1 programmer
(integer) 1
redis 127.0.0.1:6379> HGETALL profile.1
1) "name"
2) "Guillermo"
```

```
3) "last"
4) "Rauch"
```

在JavaScript中，上述指令就等同于在哈希中使用了delete操作符：

```
delete obj['profile.1'].programmer
```

还可以使用HEXISTS来检查指定的字段是否存在：

```
redis 127.0.0.1:6379> HEXISTS profile.1 programmer
(integer) 0
```

上述命令等同于检查某个值是否不等于undefined：

```
'undefined' != typeof obj['profile.1'].programmer
```

265 列表

Redis中的列表就等同于JavaScript中的字符串数组。

对于列表，在Redis中有两个基本的操作命令是RPUSH（push到右侧，也就是列表的尾端）和LPUSH（push到左侧，也就是列表的顶端）。

操作列表和操作哈希类似：

```
redis 127.0.0.1:6379> RPUSH profile.1.jobs "job 1"
(integer) 1
redis 127.0.0.1:6379> RPUSH profile.1.jobs "job 2"
(integer) 2
```

然后可以获取指定返回的数组：

```
redis 127.0.0.1:6379> LRANGE profile.1.jobs 0 -1
1) "job 1"
2) "job 2"
```

LPUSH也类似：

```
redis 127.0.0.1:6379> LPUSH profile.1.jobs "job 0"
(integer) 3
redis 127.0.0.1:6379> LRANGE profile.1.jobs 0 -1
1) "job 0"
2) "job 1"
3) "job 2"
```

RPUSH等同于在JavaScript中push一个元素到数组中：

```
obj['profile.1.jobs'].push('job 2');
```

LPUSH等同于unshift：

```
obj['profile.1.jobs'].unshift('job 2');
```

LRANGE命令返回一个在列表中指定范围的元素。它和JavaScript中数组的slice类似但不完全一样。特别是，当第二个参数是-1时，它会返回列表中所有的值。

数据集

数据集处于列表和哈希之间。它拥有哈希的属性，即数据集中的每一项（或者哈希中的键）都是唯一不重复的。既然是等同于哈希中的键，操作数据集中的元素都只需要固定时长（也就是说，无论数据集多大，删除、添加、查找数据集中的元素都只需同等的时间）。

不像哈希但类似列表的是，数据集保存的是单个值（字符串），没有键。不过，数据集还有它专属的有意思的特性。Redis允许在数据集、联合（union）、获取到的随机元素等之间做交集操作。 266

要添加一个元素到数据集中，可以使用SADD：

```
redis 127.0.0.1:6379> SADD myset "a member"
(integer) 1
```

获取数据集的所有元素，可以使用SMEMBERS：

```
redis 127.0.0.1:6379> SMEMBERS myset
1) "a member"
```

以相同值再次调用SADD不会发生任何事情：

```
redis 127.0.0.1:6379> SADD myset "a member"
(integer) 0
redis 127.0.0.1:6379> SMEMBERS myset
1) "a member"
```

移除数据集中的某个元素，可以使用SREM：

```
redis 127.0.0.1:6379> SREM myset "a member"
(integer) 1
```

有序数据集

有序数据集拥有所有数据集的特性，不过，顾名思义，它是有序的。在Redis中，使用有序数据集的情况较少，属于高级用法。

Redis和Node

既然这些数据结构JavaScript都有（或者可以很容易地构建出来），并且操作它们也不需要什么协议或者服务器，那么Redis的作用究竟在哪里呢？

其中有一个原因就是当关闭Node进程后，所有在内存中的数据都会消失。

在第9章中，我们介绍过如何使用Redis来存储用户的session数据。要是把这些数据存储在

Node进程中，会有两大缺点，如下所示。

- 应用程序永远都无法享受到多线程带来的好处。随着应用程序负载不断增长，单进程无法承受所有的负载，我们需要将应用程序扩展到多进程或者多台计算机。
- 每次重启应用都会丢失session数据：比如，在部署新代码的时候总是需要重启的。

267

Redis还有许多其他非常重要的好处，比如：多种编程语言间的协同、持久性以及其他一些通过书写完整的数据存储方案才能获取的特性。

使用node-redis实现一个社交图谱

作为一个使用Redis和Node的示例程序，我们可以通过使用数据集和交集的强大功能，来构建一个关注（follows）和粉丝（followers）的社交图谱，和Twitter非常类似。

初始化应用

我们选择一个名为node_redis（https://github.com/ mranney/node_redis）项目作为Redis客户端，其NPM的项目名就叫redis：

package.json

```
{
    "name": "sample-social-graph"
  , "version": "0.0.1"
  , "dependencies": {
        "redis": "0.7.1"
    }
}
```

连接redis

node-redis遵循了和我们在第13章中介绍的MySQL客户端类似的设计。

首先，通过require来引入该模块，然后通过createClient来初始化客户端：

```
/**
 * 模块依赖
 */

var redis = require('redis')

/**
 * 创建客户端
 */

var client = redis.createClient();
```

客户端将所有的Redis命令都以函数的方式提供出来。比如，要使用SET，可以这样：

268

```
client.set('my key', 'my value', function (err) {
  // . . .
});
```

其他如SMEMBERS、HEXISTS命令等也是相同的使用方式。

定义模型

首先，我们需要为每个用户关注的人和粉丝创建不同的Redis数据集，其中ID作为键的一部分：

```
user:<id>:follows
user:<id>:followers
```

当某个用户（id "1"）关注了另一个用户（id "2"）时，我们需要执行如下操作：

```
- Add user id "2" to the user:1:follows
- Add user id "1" to the user:2:followers
```

除此之外，我们将用户信息存储在哈希中：

```
user:<id>:data
```

接下来，从定义基本的模型开始：

```
/**
 * 用户模型
 */

function User (id, data) {
  this.id = id;
  this.data = data;
}
```

每个User实例都会包含一个id，来标识该用户及其数据。

我们也可以提供一个静态方法find，用来从Redis查询结果中构建一个User实例：

```
User.find = function (id, fn) {
  client.hgetall('user:' + id + ':data', function (err, obj) {
    if (err) return fn(err);
    fn(null, new User(id, obj));
  });
};
```

创建及修改用户信息

模型需要有查询一个用户、修改该用户的信息以及重新保存到Redis的能力。我们需要能够运行new User，设置一些数据，然后将其保存到Redis中。

269 幸运的是，通过Node来操作哈希要比在Redis命令行中更简单。hgetall和hmset函数可以直接操作JavaScript原生对象，这就是为什么操作起来更容易的原因：

```
client.hmset('somehash', { a: 'key', another: 'key' });
```

上述代码等同于如下所示的在Redis命令行（CLI）执行的命令：

```
HMSET somehash "key" "value" "anotherkey" "anothervalue"
```

当通过hgetall获取到数据后，回调函数中的第二个参数就是 JavaScript对象：

```
client.hgetall('somehash', function (err, obj) {
  // obj.a == 'key'
});
```

所以，我们可以给模型添加一个save方法，该方法执行hmset来创建和修改用户信息：

```
User.prototype.save = function (fn) {
  if (!this.id) {
    this.id = String(Math.random()).substr(3);
  }

  client.hmset('user:' + this.id + ':data', this.data, fn)
};
```

定义图谱方法

对于一个指定用户，我们需要做的操作是：关注另一个用户，以及取消关注另一个用户。

```
User.prototype.follow = function (user_id, fn) {
  client.multi()
    .sadd('user:' + user_id + ':followers', this.id)
    .sadd('user:' + this.id + ':follows', user_id)
    .exec(fn);
};

User.prototype.unfollow = function (user_id, fn) {
  client.multi()
    .srem('user:' + user_id + ':followers', this.id)
    .srem('user:' + this.id + ':follows', user_id)
    .exec(fn);
};
```

注意了，和此前的例子不同，这一次调用的是client.multi。调用了multi就意味着告诉Redis客户端，所有的命令都必须要等到exec执行后才能执行，它们作为事务的一部分，应当需要一起执行。

270 如果修改关注列表的时候，发生了某些事件导致了粉丝列表修改失败，这时就会导致数据的不一致。所以，这两部分的修改需要放到同一个事务中来处理。

最后，还需要定义两个获取关注的人和粉丝的方法：

```
User.prototype.getFollowers = function (fn) {
  client.smembers('user:' + this.id + ':followers', fn);
};

User.prototype.getFollows = function (fn) {
  client.smembers('user:' + this.id + ':follows', fn);
};
```

计算交集

除了关注的人和粉丝之外，我们还需要计算第三种关系：朋友。

我们可以说两者互相关注的是朋友。换句话说，如果某个用户的ID同时出现在另一个用户的关注和粉丝列表中，那么他们就是朋友。

添加一个getFriends很容易，直接使用SINTER命令来计算两个数据集的交集即可：

```
User.prototype.getFriends = function (fn) {
  client.sinter('user:' + this.id + ':follows', 'user:' + this.id + ':followers',
  fn);
};
```

测试

要测试模型，我们需要创建一些对Web应用来说有代表性的用户数据。

第一步要做的就是为了更好地重用，需要把此前我们书写的模型代码放到单独的文件中，这样就可以简单地通过require来引入了。注意了，我添加了一行module.exports代码：

---

model.js

```
/**
 * 模块依赖
 */

var redis = require('redis')

/**
 * 模块导出
 */

module.exports = User;

/**
 * 创建客户端
```

```
 */

var client = redis.createClient();

/**
 * 用户模型
 */

function User (id, data) {
  this.id = id;
  this.data = data;
}

User.prototype.save = function (fn) {
  if (!this.id) {
    this.id = String(Math.random()).substr(3);
  }

  client.hmset('user:' + this.id + ':data', this.data, fn)
};

User.prototype.follow = function (user_id, fn) {
  client.multi()
    .sadd('user:' + user_id + ':followers', this.id)
    .sadd('user:' + this.id + ':follows', user_id)
    .exec(fn);
};

User.prototype.unfollow = function (user_id, fn) {
  client.multi()
    .srem('user:' + user_id + ':followers', this.id)
    .srem('user:' + this.id + ':follows', user_id)
    .exec(fn);
};

User.prototype.getFollowers = function (fn) {
  client.smembers('user:' + this.id + ':followers', fn);
};

User.prototype.getFollorws = function (fn) {
  client.smembers('user:' + this.id + ':follows', fn);
};

User.prototype.getFriends = function (fn) {
  client.sinter('user:' + this.id + ':follows', 'user:' + this.id + ':followers',
 fn);
};

User.find = function (id, fn) {
  client.hgetall('user:' + id + ':data', function (err, obj) {
```

272

```
    if (err) return fn(err);
    fn(null, new User(id, obj));
  });
};
```

确保要运行npm install redis来安装本例依赖的唯一模块，与此同时通过运行redis-cli来确保Redis服务器已经运行了。

接下来，我们需要创建一个测试脚本并引入模型。在同一目录下，创建一个test.js文件：

test.js

```
/**
 * 模块依赖
 */

var User = require('./model')
```

如果现在通过node test执行上述文件，你会发现它的进程会挂在那里，不会自己退出。这是因为模型正在和Redis建立连接。

接着，我们要创建一些测试用户。为了更好地组织代码，避免过多的嵌套回调函数，我们定义一个create方法，该方法接收一个包含email和用户数据的对象。本例中，email地址会作为唯一的id存储到Redis中。

test.js

```
/**
 * 模块依赖
 */

var User = require('./model')

/**
 * 创建测试用户
 */

var testUsers = {
    'mark@facebook.com': { name: 'Mark Zuckerberg' }
  , 'bill@microsoft.com': { name: 'Bill Gates' }
  , 'jeff@amazon.com': { name: 'Jeff Bezos' }
  , 'fred@fedex.com': { name: 'Fred Smith' }
};

/**
 * 用于创建用户的函数
```

```
 */

function create (users, fn) {
  var total = Object.keys(users).length;
  for (var i in users) {
    (function (email, data) {
      var user = new User(email, data);
      user.save(function (err) {
        if (err) throw err;
        --total || fn();
      });
    })(i, users[i]);
  }
}

/**
 * 创建测试用户
 */

create(testUsers, function () {
  console.log('all users created');
  process.exit();
});
```

这个时候运行node test，应当就已经成功添加了四个用户。可以使用redis-cli来检查是否正确：

```
redis-cli
redis 127.0.0.1:6379> KEYS *
1) "user:fred@fedex.com:data"
2) "user:jeff@amazon.com:data"
3) "user:bill@microsoft.com:data"
4) "user:mark@facebook.com:data"
redis 127.0.0.1:6379> HGETALL "user:fred@fedex.com:data"
1) "name"
2) "Fred Smith"
```

注意了，上述HGETALL命令和User.find函数的作用是一样的：它根据id获取对应用户的数据。

274 为了避免混淆，每次执行完node test，都建议将Redis数据库清空以确保由不同测试用例创建的数据不会相互混淆。清空数据库可以使用如下命令：

```
redis-cli FLUSHALL
```

至此，用户已经成功创建好了，接下来我们需要根据email来获取User对象，这样就可以通过此前在模型中定义的方法来建立用户之间的关系。这个处理过程称为水合（hydration），我们要建立一个hydrate方法，并在create回调函数中使用：

test.js

```
/**
 * 用于水合用户的函数
 */

function hydrate (users, fn) {
  var total = Object.keys(users).length;
  for (var i in users) {
    (function (email) {
      User.find(email, function (err, user) {
        if (err) throw err;
        users[email] = user;
        --total || fn();
      });
    })(i);
  }
}

/**
 * 创建测试用户
 */

create(testUsers, function () {
  hydrate(testUsers, function () {
    console.log(testUsers);
  });
});
```

如图14-1所示，若再次运行测试脚本，你会发现testUsers对象包含了User对象（该对象包含在User构造器中传递的id和data属性）。

```
{ 'mark@facebook.com': { id: 'mark@facebook.com', data: { name: 'Mark Zuckerberg' }
 },
 'bill@microsoft.com': { id: 'bill@microsoft.com', data: { name: 'Bill Gates' } },
 'jeff@amazon.com': { id: 'jeff@amazon.com', data: { name: 'Jeff Bezos' } },
 'fred@fedex.com': { id: 'fred@fedex.com', data: { name: 'Fred Smith' } } }
```

275

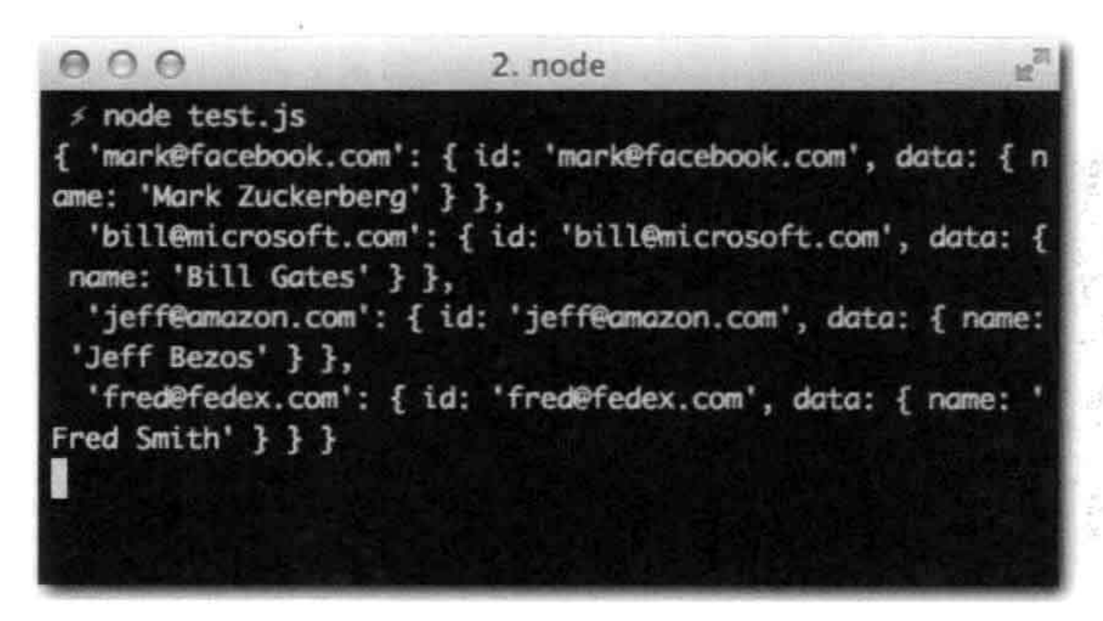


图14-1：以JSON格式输出的创建好的用户数据

水合后，就可以使用模型上多种不同的方法了：

test.js

```
create(testUsers, function () {
  hydrate(testUsers, function () {
    testUsers['bill@microsoft.com'].follow('jeff@amazon.com', function (err) {
      if (err) throw err;
      console.log('+ bill followed jeff');

      testUsers['jeff@amazon.com'].getFollowers(function (err, users) {
        if (err) throw err;
        console.log("jeff's followers", users);

        testUsers['jeff@amazon.com'].getFriends(function (err, users) {
          if (err) throw err;
          console.log("jeff's friends", users);

          testUsers['jeff@amazon.com'].follow('bill@microsoft.com'
          , function (err) {
            if (err) throw err;

            console.log('+ jeffed follow bill');
            testUsers['jeff@amazon.com'].getFriends(function (err, users) {
              if (err) throw err;

              console.log("jeff's friends", users);
              process.exit(0);
            });
          });
        });
      });
    });
  });
});
```

276

在图14-2中，如控制台输出所示，它展示了Redis中的交集操作的结果。

图14-2：社交图谱中的粉丝和朋友

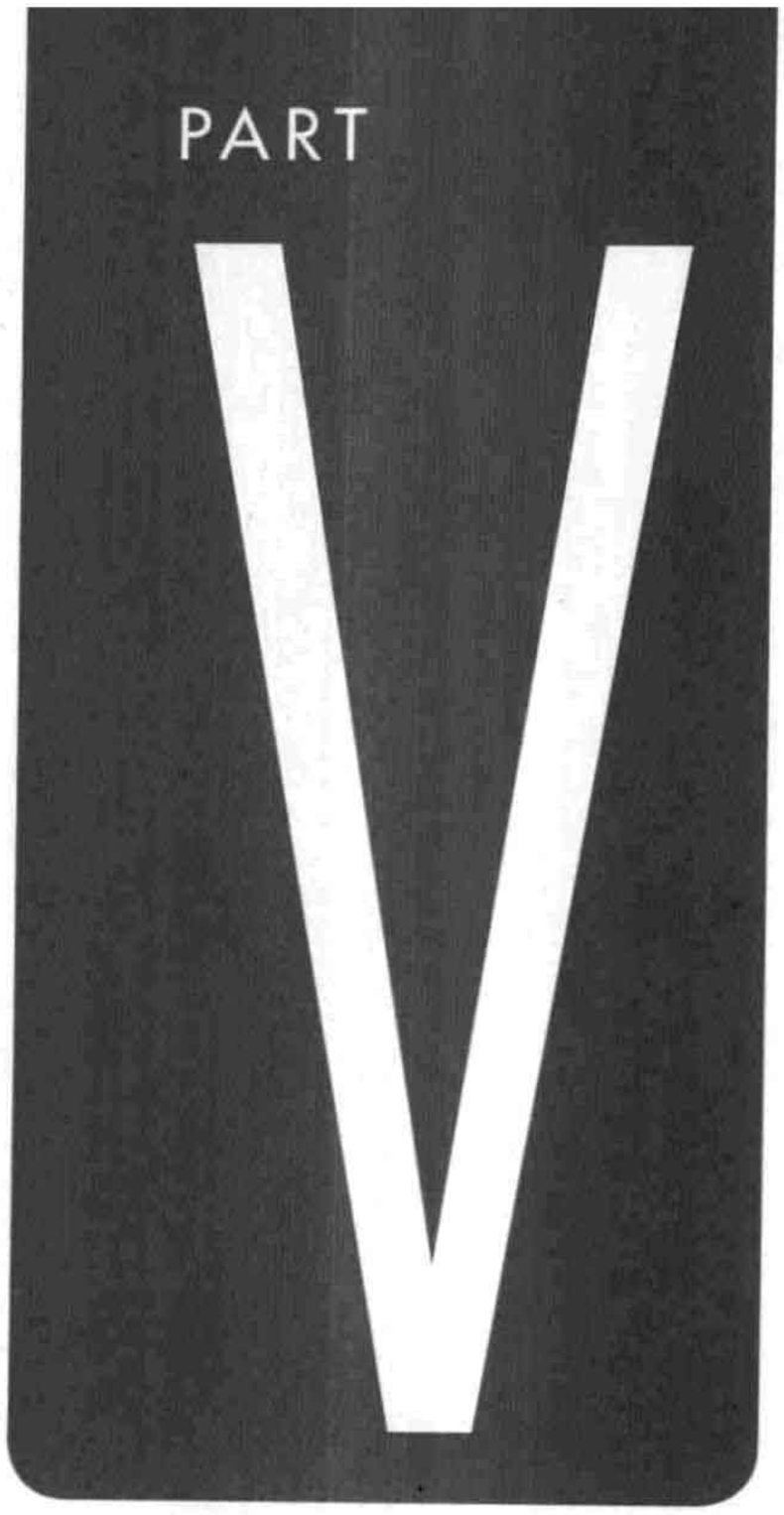
小结

在我看来Redis是最重要的、生机盎然的数据库之一，这也是我在本章一开始花笔墨来介绍它的基本技术点的原因。

因为它就像是结构化的数据服务器，既可以用作普通的数据库，也可作为一种黏合剂来协调多个小程序。

比如，在第9章中，我们介绍了如何使用`connect-redis`来当重启Node进程时保持session数据的持久化。当然了，Node本身也可以将这些数据存储在内存中，不过通过将数据分离到另一个进程，并通过简单的TCP协议来连接操作，就可以获得灵活独立的好处：不管程序本身有没有运行，数据都在那儿。

大量程序和Web应用都有简单的数据模型，数据集也非常适合存储在内存中。对于这些使用场景，我推荐你首先考虑Redis，因为它简单、可靠，而且通过Node.js非常易于使用。本章中的示例程序就是一个很好的例子：我们书写了一个完整的模型，该模型仅通过使用 Redis驱动器就包含了所有当今社交应用所需的重要功能。



测试

CHAPTER 15 代码共享
CHAPTER 16 测试

CHAPTER

15 代码共享

在本书介绍部分，我就提过，Node.js最重要的特点之一就是：它所使用的语言——JavaScript，是浏览器唯一支持的语言。 279

尽管现在我们已经写了很多独立的JavaScript代码，减少了在开发 Web应用时总要在不同语言之间切换上下文的痛苦。我们还没有享受到一次编码处处运行的好处。

本章分析适合代码共享的最佳用例，以及如何解决常见的语言兼容性问题。本章还会介绍如何书写模块化Node代码的最佳实践，并将其通过browserbuild编译后在浏览器中直接运行。

什么样的代码可以共享

280

回答代码是否可以在浏览器和服务器端共享的最简单的方式就是把问题拆为如下两个问题：

- 是否值得在两个环境中运行同一份代码？
- 代码依赖的API在两个环境中是否都有？如果不是，那么是否能够很容易地找到替换它们的方法（称为模拟实现法）。

回答第一个问题比较简单，答案本身取决于程序和项目本身。回答第二个问题就有点困难了。

在第2章中，我们介绍了有些特定的和JavaScript有关联的API并非是语言的一部分，而是浏览器提供的标准API。例如：XMLHttpRequest、WebSocket、DOM、2d画布 API，等等。

尽管Node.js核心并未提供XMLHttpRequest API，但是我们可以通过使用Node.js的HTTP客户端API来替代。在NPM有个名为xmlhttprequest的项目就是起这个作用的。

在其他的场景中，某些模块只和原生API交互，这类代码就很容易写成可以在两个环境中运行。

这类例子如下所示。

- 日期操作工具集：它们通常简单地处理原生Date API或者对其做相应扩展。
- 模板引擎：它们通常接收一个字符串，通过正则表达式或者循环来解析，并生成一个（编译）函数，用于输出编译后的字符串。
- 数学及加密库：它们通常处理Number和与数学相关的内容。
- 面向对象框架：它们提供一些在JavaScript中书写类的语法糖。换句话说，提供一些API，使得在JavaScript中能和在其他典型的面向对象语言中一样书写面向对象代码。

书写兼容的JavaScript代码

第一个挑战就是要解决在Node.js模块系统中书写的JavaScript代码无法在浏览器中运行的问题。

导出模块

第一个问题是浏览器环境中没有module全局变量。哪怕文件本身没有依赖，但是，要想让它能为其他Node程序所用，需要通过module.exports或者exports。

281

假设，我们已熟悉一个简单的两数求和的函数：

add.js

```
module.exports = function (a, b) {
  return a + b;
}
```

在Node中，只需简单地从另外一个文件中调用require('./add')就可以了。在浏览器中，我们更希望将其暴露为一个全局变量add。

为了避免修改已有代码，我们可以在浏览器中，通过伪造一个module对象来模拟module.exports：

```
if ('undefined' == typeof module) {
  module = { exports: {} };
}
```

模块代码的最后，对象生成后，就可以将其暴露给window全局对象了。

```
if ('undefined' != typeof window) {
  window.add = module.exports;
}
```

和Node.js不同，在浏览器中，文件都是在全局作用于下执行的。因此，我们需要为程序引入一个全局的module变量。

将模块代码包裹在一个自执行的函数中是个不错的主意。

```
(function (module) {
  module.exports = function (a, b) {
    return a + b;
  }

  if ('undefined' != typeof window) {
    window.add = module.exports;
  }
})('undefined' == typeof module ? { module: { exports: {} } } : module);
```

太好了！模块还是可以通过require获得：

```
$ node
> require('./add')(1,2)
3
```

同时，还能在DOM中引入：

282

add.html

```
<script src="add.js"><script>
<script>
  console.log(add(1,2));
</script>
```

模拟实现ECMA API

下一个挑战就是，一些主流浏览器中的特性在其他浏览器及JavaScript引擎中都没有。

有时，Node中的v8引擎甚至比诸多电脑所用的Google Chrome浏览器的引擎还要新。然而，对于Function#bind这样的API，一些主流引擎，如Safari引擎JavaScriptCore VM就还不支持。所以，要想让代码处处运行，对于使用了哪些功能要特别小心。

对于缺失的这类功能，有两个解决办法：通过扩展原型来提供这类功能的实现或者使用工具函数。

扩展原型

假设我们书写了一个同时在Node和浏览器中运行的模块，它用到了Function#bind方法。

```
var myfn = fn.bind(this);
```

由于bind方法并不一定在所有环境中都存在，所以当它不存在时，我们要创建一个出来：

```
if (!Function.prototype.bind) {
  Function.prototype.bind = function () {
    // code that replicates bind behavior
  }
}
```

有一个名为es5-shim的项目做了类似上面的事情，它把所有浏览器中缺少的ECMA标准API都实现了。要了解该项目详情，可以参考https://github.com/kriskowal/es5-shim。

这种技术有个很明显的好处就是在加入填补的方法后，几乎不需要修改源代码。

283

缺点就是这么做会破坏原型对象，影响其他使用者，可能还会更糟。

工具函数

另一种解决方法就是定义简单的函数，接收原生对象作为参数，如果该对象上的函数已经实现了，就直接使用，否则就实现一次。

例如，Object.keys就是个很好的例子，在Node中会经常使用此函数，但是在很多浏览器中却还没有提供其实现。

接着，我们就通过如下方式来定义一个工具函数：

```
var keys = Object.keys || function (obj) {
  var ret = [];
  for (var i in obj) {
    if (Object.prototype.hasOwnProperty.call(obj, i)) {
      ret.push(i);
    }
  }
  return ret;
};
```

这种技术的好处就是随处都能用，没有隐患，并且对于开发者来说，能够一眼就看明白哪些方法是模拟的，哪些是原生的，这对于在某些浏览器中检测代码性能很有帮助。

缺点就是，我们要记住是用工具函数而不是用原生的。

除此之外，有的时候，最终书写出来的代码会有些冗长。比如，旧版本的IE不支持Array#forEach、Array#map及Array#filter，要想写出兼容的代码，可能要写成如下形式：

```
arr.filter(function () {}).map(function(){ }).forEach(function () {})
```

不过，上述代码远没有使用工具函数来得清晰：

```
each(map(filter(arr, function() {}), function () {}), function (){})
```

模拟实现Node API

有些如EventEmitter这样的Node API一般会在自定义的类中使用。幸运的是，Node社区书写了可以在所有环境下运行的Node API。

EventEmitter

284

对node EventEmitter的实现可以参考https://github.com/Wolfy87/EventEmitter和https://github.com/tmpvar/node-eventemitter。

assert

支持浏览器的assert模块可以在这里找到： https://github.com/Jxck/assert。

模拟实现浏览器端API

很多情况下，我们希望在Node中也能运行浏览器支持的方法。

XMLHttpRequest

node-XMLHttpRequest项目（https://github.com/driverdan/node-XMLHttpRequest）为Node提供了XMLHttpRequest的模拟实现，通过如下方式就可以引入该模块了：

```
var XMLHttpRequest = require('xmlhttprequest')
```

DOM

一个完整并且测试完全的DOM I、DOM II以及 DOM III的实现在这里：https://github.com/tmpvar/jsdom。

WebSocket

Node中也有WebSocket客户端的API实现（https://github.com/einaros/ws）：

```
var WebSocket = require('ws')
```

node-canvas

用于图片操作的2D画布上下文也有在Node中的实现，参考node-canvas：https://github.com/learnboost/node-canvas。

跨浏览器的继承实现

在模块中，经常需要将一个类继承自另一个类。第2章中介绍的.__proto__可以用于实现继承，但却很难模拟实现。

一个简单的解决方法是通过如下方式定义一个工具函数：

```
/**
 * 用于实现继承的工具函数
 *
 * @param {Function} 构造器
 * @param {Function} 父类构造器
 * @api private
 */

function inherits (a, b) {
  function c () {};
  c.prototype = b.prototype;
  a.prototype = new c;
};
```

285

然后就可以在Node和浏览器中使用了。

```
function A () {}
function B () {}

inherits(A, B);
// instead of A.prototype.__proto__ = B.prototype
```

集成到一起：browserbuild

通过自执行函数将模块代码包裹起来，同时到处都得执行typeof检测多少有点麻烦。

我创建了一个名为browserbuild的项目，它的作用就是将以Node风格书写的代码（即使用了require、module.exports以及exports，并且代码都分属不同文件），通过运行简单的命令，就编译为浏览器端可执行的版本。

browserbuild可以允许你以Node风格书写此前的求和模块，通过编译后，就会生成一个可以直接在浏览器端通过<script>嵌入执行的版本。

browserbuild同时还是个NPM模块，它提供了browserbuild命令行脚本。要全局安装该模块，可以采用如下方式：

```
npm install -g browserbuild
browserbuild --version
```

上述第二行命令用于展示所安装browserbuild的版本号。这里所用的版本号是0.4.8。

还可以直接安装到工作目录中，通过.bin目录来访问：

```
npm install browserbuild
./node_modules/browserbuild/.bin/browserbuild --version
```

基础案例

286

本例中，我们要书写一个模块，该模块依赖于日志工具。

首先，书写一个main.js文件：

main.js

```
var log = require('./log'):

module.exports = function () {
log('Executed my module');
}
```

在使用特定模块时，main文件不管是在NPM模块系统中，还是在单独的文件中，都可以通过require来引入。

log.js

```
module.exports = function (str) {
  return console.log(str);
}
```

在Node中，可以直接写成；

node.js

```
var mymodule = require('./main')
mymodule();
```

对于浏览器端，我们需要其导出成全局的mymodule变量。所以，我们需要在工作目录下运行browserbuild命令，同时提供全局变量名以及main文件。

```
browserbuild --main main --global mymodule main.js log.js > out.js.
```

上述代码的含义是告诉browserbuild：编译main.js和log.js，并将main模块导出为mymodule全局变量。

除此之外，注意，命令行最后我们使用了>compiled.js。这是将输出的结果保存到compiled.js文件中。

要将其作为库引入到浏览器中，只需添加一个<script>标签，并指向compiled.js脚本即可：

```
<script src='compiled.js">
```

287 生成的脚本如下所示：

compiled.js

```
(function(){var global = this;function debug(){return debug};function require(p,
 parent){ var path = require.resolve(p) , mod = require.modules[path]; if (!mod)
 throw new Error('failed to require "' + p + '" from ' + parent); if (!mod.exports)
 { mod.exports = {}; mod.call(mod.exports, mod, mod.exports, require.relative(path),
 global); } return mod.exports;}require.modules = {};require.resolve =
 function(path){ var orig = path , reg = path + '.js' , index = path + '/index.js';
 return require.modules[reg] && reg || require.modules[index] && index ||
 orig;};require.register = function(path, fn){ require.modules[path] = fn;};require.
 relative = function(parent) { return function(p){ if ('debug' == p) return debug;
 if ('.' != p.charAt(0)) return require(p); var path = parent.split('/') , segs =
 p.split('/'); path.pop(); for (var i = 0; i < segs.length; i++) { var seg =
 segs[i]; if ('..' == seg) path.pop(); else if ('.' != seg) path.push(seg); } return
 require(path.join('/'), parent); };};require.register("main.js", function(module,
 exports, require, global){

var log = require('./log');

module.exports = function () {
  log('Executed my module');
}

});require.register("log.js", function(module, exports, require, global){
module.exports = function (str) {
  return console.log(str);
}

});mymodule = require('main');
})();
```

上述脚本中，第一部分实现了一个require函数供后续代码使用。编译后的脚本全部压缩在一行代码中，这样文件长度对于编译过程的影响可以减到最小。

另一部分比较有意思的代码是，require.register函数提供了一个global参数。这样做是为了避免你去检测window对象是否存在，以此来提供另一个global变量。书写代码时可以直接看作是Node.js的global对象，要是在浏览器中，该对象会变成window对象。

最后，上述代码的最后一部分是导出了全局变量（mymodule），这也解释了运行browserbuild的时候要指定是main模块的原因：

```
mymodule = require('main');
```

browserbuild另一个有意思的特性是if node代码块，它允许你使用JavaScript注释来告诉编译器，该代码块中的代码不需要编译到浏览器端版本中： 288

nodeonly.js

```
// if node
process.exit(1);
// end

console.log('browser and node');
```

如果运行如下命令编译上述代码，if node代码块中的代码就不会被编译进去：

```
$ browserbuild --main nodeonly nodeonly.js
```

你会注意到process.exit代码行不在里面。

```
// …
require.register("nodeonly.js", function(module, exports, require, global){
console.log('browser and node');
});nodeonly = require('nodeonly');
})();
```

要获取更多browserbuild命令的选项，可以执行browserbuild --help，或者参看项目主页： http://github.com/learnboost/browserbuild。

小结

正如全书一直在强调的，Node.js做了一项很伟大的工作就是将 JavaScript带到了服务器端。

这一创新依赖于其模块系统，而浏览器端则没有对应的默认模块系统。

本章从介绍如何利用运行时的方法检测来书写同时能在服务器端和浏览器端运行的代码。通过执行typeof检查，可以实现模块系统的特性检查，并为浏览器端也提供一个导出机制，比如通过全局对象。

然而，在所有的文件中都要进行手动地通过自执行的函数将代码包裹起来，并执行typeof进行检查，这显然有悖于Node中require系统的简洁性，browserbuild就是为了解决这类问题而生的。有了它，就可以很轻松地书写Node.js模块，并编译到浏览器端进行运行了。

这种方法的显著的好处就是，模块是完全以全局变量的方式在浏览器端导出的，就和jQuery和IO一样，也就是说，并没有引入一个特定的模块系统API来让终端用户在浏览器环境中使用。

CHAPTER

16 测试

至此，每次书写完Node程序，都要通过执行程序来检测是否结果和我们预期的一样，从而判断工作是否正常。随着时间越来越长，这种确保程序工作正常并且没有引入新bug的测试方法是非常低效的。

自动化测试是一个过程，它通过执行一系列程序来检验函数工作是否正确。本章首先要介绍的自动化测试的方法是为每个测试创建一段小的 node程序，然后使用原生的`assert`模块来检验。

接着，我们会通过使用一个名为expect.js的项目来优化书写断言（assertion）的过程。紧接着，我们还会介绍如何使用测试框架 Mocha来组织测试代码。

最后，对于在Node和浏览器端都要运行的代码，本章会介绍如何将已有的测试也用到浏览器端。

简单测试

开始前，我们先要确定测试目标。换句话说，我们得确定哪些脚本或者功能需要测试。

测试目标

本章的测试目标是我们在第9章中书写的查询推文的应用。

这里，我们书写一个程序，来断言提交查询时，看看是否能够找到对应的HTML代码来证明查询的推文内容返回了。本例子，推文列表是通过一个或者多个<li>元素来构建的。通过断言查询关键字是否存在，以及<li>字符串是否为HTTP响应内容的一部分来判断应用程序是否工作正常，应当已经很充分了。

要开始自动化测试前，我们先运行应用，确保它能正常工作。然后，通过浏览器访问http://localhost:3000。

测试策略

最基本的测试方法就是书写一个新的node程序，当测试通过时就以状态码0退出程序，失败则以状态码1退出。

如果有未捕获的异常（uncaught exception）抛出，Node会自动以失败错误码退出程序，和我们的设计相符。除此之外，还能获得堆栈踪迹来帮助调试错误原因。因此，书写测试代码时，目标就是要断言条件是否通过，不通过就抛出异常。Node自带了一个assert模块，顾名思义，它的作用就是检查判断条件是否通过，如果不通过，就抛出一个AssertionError异常。

作为例子，我们书写一个测试程序，当时间戳是偶数时成功，是奇数时失败，同时抛出堆栈信息。

```
/**
 * 模块依赖
 */

var assert = require('assert');

/**
 * 断言条件
 */

var now = Date.now();
console.log(now);
assert.ok(now % 2 == 0);
```

291 assert.ok可以用来判断提供的值是否为真（哪怕真是值不是true，我们还是要断言它是true）。当数字除以2没有余数时，那么该数字就是偶数。

现在，我们多运行几次上述程序，看看时间戳：

```
 ∞ simple-testing node assert-example.js
1325520251830
 ∞ simple-testing node assert-example.js
1325520252742
 ∞ simple-testing node assert-example.js
1325520253637

node.js:134
        throw e; // process.nextTick error, or 'error' event on first tick
        ^
AssertionError: true == false
    at Object.<anonymous> (assert-example.js:14:8)
    at Module._compile (module.js:411:26)
    at Object..js (module.js:417:10)
    at Module.load (module.js:343:31)
    at Function._load (module.js:302:12)
    at Array.<anonymous> (module.js:430:10)
    at EventEmitter._tickCallback (node.js:126:26)
```

前两次中，由于时间戳是偶数，所以测试通过了。第三次，时间戳变成了奇数，所以抛出了异常，输出了堆栈信息。

测试程序

现在，我们使用superagent，发送GET请求来查询bieber关键字，然后分析响应结果：

```
/**
 * 模块依赖
 */

var request = require('superagent')
  , assert = require('assert')

/**
 * 测试 /search?q=<tweet>
 */

request.get('http://localhost:3000')
  .data({ q: 'bieber' })
  .exec(function (res) {

  // 断言响应状态码是否正确
  assert.ok(200 == res.status);

  // 断言关键字是否存在
  assert.ok(~res.text.toLowerCase().indexOf('bieber'));

  // 断言列表项是否存在
  assert.ok(~res.text.indexOf('<li>'));
});
```

292

- throwException：断言Function在调用时是否会抛出异常。

```
expect(fn).to.throwException();
expect(fn2).to.not.throwException();
```

- within：断言数组是否在某个区间内。

```
expect(1).to.be.within(0, Infinity);
```

- greaterThan/above：断言 >。

```
expect(3).to.be.above(0); expect(5).to.be.greaterThan(3);
```

- lessThan/below：断言 <。

```
expect(0).to.be.below(3); expect(1).to.be.lessThan(3);
```

改进了断言风格，接下来，我们要通过一个名为Mocha的框架重构测试代码的组织方式。

Mocha

Mocha是一个测试框架，简化了书写测试代码的过程，提供划分测试集、运行并同时输出有助于开发者理解的结果页。

相比把一份份测试代码写在不同文件中，会导致的文件系统中存有大量无序文件，通过Mocha，可以书写出如下形式的测试代码：

test.js

```
describe('a topic', function () {

  it('should test something', function () {

  });

  describe('another topic', function () {

    it('should test something else', function () {

    });

  });

});
```

295

与expect.js类似，在test.js中描述和组织测试代码的方式非常自然直观。

要运行测试代码只需通过mocha命令即可。在这之前，需要确保运行npm install -g mocha安装了mocha。然后，就通过如下方式来运行：

- be/equal：类似===。

```
expect(1).to.be(1);
expect(NaN).not.to.equal(NaN);
expect(1).not.to.be(true);
expect('1').to.not.be(1);
```

- eql：断言非严格相等，支持对象。

```
expect({ a: 'b' }).to.eql({ a: 'b' });
expect(1).to.eql('1');
```

- a/an：断言所属类型，支持数组和instanceof。

```
// typeof with optional array
expect(5).to.be.a('number');
expect([]).to.be.an('array'); // works
expect([]).to.be.an('object'); // works too, since it uses typeof`
// constructors
expect(5).to.be.a(Number);
expect([]).to.be.an(Array);
expect(tobi).to.be.a(Ferret);
expect(person).to.be.a(Mammal);
```

- match：断言字符串是否匹配一段正则表达式。

```
expect(program.version).to.match(/[0-9]+\.[0-9]+\.[0-9]+/);
```

- contain：断言字符串是否包含另一个字符串，内部使用indexOf方法。

```
expect([1, 2]).to.contain(1);
expect('hello world').to.contain('world');
```

- length：断言数组长度，内部使用.length方法。

```
expect([]).to.have.length(0);
expect([1,2,3]).to.have.length(3);
```

- empty：断言数组是否为空。

```
expect([]).to.be.empty();
expect([1,2,3]).to.not.be.empty();
```

- property：断言某个自身属性（或值）是否存在。

```
expect(window).to.have.property('expect');expect(window).to.have.
 property('expect', expect)
expect({a: 'b'}).to.have.property('a');
```

- key/keys：断言键是否存在，支持only修饰符。 294

```
js expect({ a: 'b' }).to.have.key('a');
expect({ a: 'b', c: 'd' }).to.only.have.keys('a', 'c');
expect({ a: 'b', c: 'd' }).to.only.have.keys(['a', 'c']);
expect({ a: 'b', c: 'd' }).to.not.only.have.key('a');
```

注意了，如果请求抛出异常，我们直接不做处理，往上抛即可，最终它会变成一个未捕获的异常，导致程序失败并退出。

记住在运行测试前要先安装superagent：

```
npm install superagent@0.4.1
```

expect.js

在此前的例子中，我们使用了assert.ok以及基本的JavaScript表达式。

在看测试代码时，你也许会发现很难看懂到底在测试什么。比如，断言一个字符串是否包含另一个字符串时，最简单的方法就是使用indexOf方法和~操作符，正如此前例子中处理的那样。

expect.js提供了一个简单的expect函数，可以将：

```
assert.ok(~res.text.indexOf('<li>'));
```

改写成：

```
expect(res.text).to.contain('<li>');
```

通过近自然语言的表达，使得书写和理解测试代码变得更加容易。

expect.js可以通过NPM获取，其名为expect.js，同时，它的官方文档是https://github.com/learnboost/expect.js。

接下来，会介绍一些expect.js的基础API。

API一览

通过引入expect.js模块就可以使用expect函数：

```
var expect = require('expect.js')
```

293 expect.js可以和任何模块一起使用，如此前用到的assert。与所有assert模块提供的函数类似，在expect.js中，当断言失败时会抛出AssertionError。

下面是一些expect.js 0.1.2中最有用的方法。

- ok：断言值是否为真。

```
expect(1).to.be.ok();
expect(true).to.be.ok();
expect({}).to.be.ok();
expect(0).to.not.be.ok();
```

```
mocha test.js
```

Mocha使用多种报告形式来展示测试通过和运行结果。

```
mocha test.js

  ..

    2 tests complete (0ms)
```

比如，有种报告形式是列表：

```
mocha -R list test.js

    a topic should test something: 0ms
    a topic another topic should test something else: 0ms

    2 tests complete (1ms)
```

正如下面将要介绍的，Mocha还有一种HTML报告形式，可以让你直接在浏览器查看运行。

测试异步代码

Mocha默认会在一个测试用例执行完毕后立刻执行另一个。然而，很多时候，当有异步事件发生时，我们希望能够延缓下一个测试用例的执行。

考虑如下例子：

```
it('should not throw', function () {
  setTimeout(function () {
    throw new Error('An error!');
  }, 100);
});
```

上述测试代码总会通过。原因就在于设置了计时器后代码就执行完毕了，Mocha会继续去执行下一个。由于在计时器设置后，没有立刻抛出异常，所以测试通过。

对于这类异常要在将来才会抛出（异步的行为）的测试，我们需要告诉Mocha，等到我们通知它说完成了才能认为该测试完成了。

要解决这个问题，我们只要简单地在回调函数中添加一个参数就可以了。也就是说，正如第2章中介绍的，我们将函数的参数数量（或者是`function#length`）设置为1： 296

```
it('should not throw', function (done) {
  setTimeout(function () {
    assert.ok(1 == 1);
  }, 100);
});
```

现在，测试代码看起来就像中间件一样。Mocha会一直等到done函数调用之后才会认为该测试已经结束。如果该函数在两秒内（默认）没有调用，就会抛出一个超时异常来告诉你哪个测试卡住了。你也可以通过mocha命令的-t选项来修改这个超时时长。还可以通过如下方式来自定义超时时长：

```
it('will fail', function () {
  this.timeout(100);
  setTimeout(function () {
    // the test will timeout before this occurs
  }, 1000);
});
```

为了让这个测试用例能够工作，我们在断言结束后调用done方法：

```
it('should not throw', function (done) {
  setTimeout(function () {
    assert.ok(1 == 1);
    done();
  }, 100);
});
```

我们可以使用Mocha将此前测试查询Bieber推文应用的代码改写为如下形式：

```
it('should find bieber tweets', function (done) {
  request.get('http://localhost:3000')
    .data({ q: 'bieber' })
    .exec(function (res) {

      //断言状态码是否正确
      assert.ok(200 == res.status);

      //断言查询关键字是否存在
      assert.ok(~res.text.toLowerCase().indexOf('bieber'));

      //断言列表项是否存在
      assert.ok(~res.text.indexOf('<li>'));

      done();
    });
});
```

297

有时，一个测试用例只有在当多个异步操作一起完成时才算通过。

这时我们就可以使用一个计数器来解决这个问题：

```
it('should complete three requests', function (done) {
  var total = 3;
  request.get('http://localhost:3000/1', function (res) {
    if (200 != res.status) throw new Error('Request error');--total || done();
  });
```

```
  request.get('http://localhost:3000/2', function (res) {
    if (200 != res.status) throw new Error('Request error');--total || done();
  });
  request.get('http://localhost:3000/3', function (res) {
    if (200 != res.status) throw new Error('Request error');;
    --total || done();
  });
});
```

注意了，Mocha非常“聪明”，它能够识别出未捕获的异常属于哪个测试用例。这是因为Mocha任何时候只执行一个测试用例，所以它能够准确地将通过`process.on('uncaughtException')`处理器捕获的未捕获的错误（uncaught Exception）链接到正确的测试用例上。

BDD风格

前面小节中使用的测试代码书写风格称为：行为驱动开发（BDD）。

在接下来的例子中，我们给jade提供一个包含段落的模板来测试Jade的处理行为。过程中还会用到expect.js。

首先，安装Jade、Mocha和expect.js：

```
$ npm install expect.js jade mocha
```

bdd.js

```
var expect = require('expect.js')
  , jade = require('jade');

describe('jade.render', function () {
  it('should render a paragraph', function () {
    expect(jade.render('p A paragraph')).to.be('<p>A paragraph</p>');
  });
});
```

由于Mocha安装在项目路径中时并非是全局安装，所以，要在`./node_modules/bin`找到它并执行：

```
$ ./node_modules/.bin/mocha bdd.js
```

TDD风格

298

接下来要介绍的是测试驱动开发（TDD）。它和BDD类似，但是组织方式是使用测试集（suite）和测试（test）。

每个测试集都有setup和teardown函数。这些方法会在测试集中的测试执行前执行，它们的作用是为了避免代码重复以及最大限度使得测试之间相互独立。

现在，测试代码看起来就像中间件一样。Mocha会一直等到done函数调用之后才会认为该测试已经结束。如果该函数在两秒内（默认）没有调用，就会抛出一个超时异常来告诉你哪个测试卡住了。你也可以通过mocha命令的-t选项来修改这个超时时长。还可以通过如下方式来自定义超时时长：

```
it('will fail', function () {
  this.timeout(100);
  setTimeout(function () {
    // the test will timeout before this occurs
  }, 1000);
});
```

为了让这个测试用例能够工作，我们在断言结束后调用done方法：

```
it('should not throw', function (done) {
  setTimeout(function () {
    assert.ok(1 == 1);
    done();
  }, 100);
});
```

我们可以使用Mocha将此前测试查询Bieber推文应用的代码改写为如下形式：

```
it('should find bieber tweets', function (done) {
  request.get('http://localhost:3000')
    .data({ q: 'bieber' })
    .exec(function (res) {

      //断言状态码是否正确
      assert.ok(200 == res.status);

      //断言查询关键字是否存在
      assert.ok(~res.text.toLowerCase().indexOf('bieber'));

      //断言列表项是否存在
      assert.ok(~res.text.indexOf('<li>'));

      done();
    });
});
```

297

有时，一个测试用例只有在当多个异步操作一起完成时才算通过。

这时我们就可以使用一个计数器来解决这个问题：

```
it('should complete three requests', function (done) {
  var total = 3;
  request.get('http://localhost:3000/1', function (res) {
    if (200 != res.status) throw new Error('Request error');--total || done();
  });
```

2 告诉Mocha采用什么样的测试风格（TDD、BDD或者export）。

3 载入测试代码。

4 运行Mocha。

expect.js可以在所有浏览器中运行，它可以很好地和Mocha配合使用。

建立项目

我们从创建一个test/文件夹开始，该文件夹包含Mocha在浏览器端运行所需文件（mocha.css和mocha.js）。

我们可以从node_modules/mocha目录中将这两个文件复制过来，或者从git仓库中直接下载。要了解如何下载这些文件，可以参考 http://mochajs.com。

我们还需要载入jQuery、expect.js以及测试代码，然后调用mocha.setup来设置测试风格。

最终，test/index.html代码如下所示：

test/index.html

```
<!doctype html>
<html>
  <head>
    <title>my tests</title>
    <link href="/mocha.css" rel="stylesheet" media="screen" />
    <script src="/jquery.js"></script>
    <script src="/mocha.js"></script>
    <script src="/expect.js"></script>
    <script>mocha.setup('bdd');</script>
    <script src="/my-test.js"></script>
    <script>window.onload = function () { mocha.run(); };</script>
  </head>
  <body>
    <div id="mocha"></div>
  </body>
</html>
```

300

如上述代码所示，Mocha会执行my-test文件，该文件中以BDD的风格做了一些简单的测试：

my-test.js

```
describe('my tests', function () {
  it('should not throw', function () {
    expect(1 + 1).to.be(2);
  });
});
```

托管目录

最简单的以网站的形式托管整个目录的方式是使用serve(1)命令。这是一个简单的命令行程序，serve内部使用connect的static中间件来托管指定的目录：

```
$ serve .
```

之后，简单地通过浏览器访问http://localhost:3000就可以了。

小结

本章一开始介绍了如何以最简单的方式书写测试程序：运行一个简单的测试脚本并查看其运行结果是否成功。

接着，为了验证在测试脚本中，某段条件判断是否通过，本章介绍了一个Node.js核心模块assert。

在这基础上，本章又介绍了使用expect.js和Mocha来提升测试代码风格以及组织形式。Expect可以让书写的测试代码清晰易懂，而Mocha提供了很好的组织测试代码的方法，同时还允许将测试代码在浏览器端运行。除此之外，对于测试异步代码，也游刃有余。

索引

SYMBOLS AND NUMERICS

> (angle bracket), defined, 9
$ (dollar sign) prefix, 8
./, (dot slash) adding to require parameters, 43
\r\n delimiter, 85
[] (square brackets) for uploading multiple files, 134
404 (Not Found) status code, 117

A

absolute modules, defined, 41
accepts extension for Request, 156
accessors in v8, 25–26
acknowledgment, Socket.IO, 190–191
acknowledgments, TCP, 71
addEventListener API, 45–46
addMessage function, 187, 190
AJAX, 164–166, 180, 245
Amazon Linux, 9
angle bracket (>), defined, 9
ANSI escape codes (website), 66
Apache/PHP compared to Node, 28–29
app. users function, 212
app.configure function, 153
app.enable function, 153
app.error function, 155
apply method, 18
app.set function, 147, 148
app.use function, 159, 161
argv API, 63–64, 108
arity of a function, 19, 296
array JavaScript type, 16, 17
arrays, 23, 239–240, 242–243
assert module, 284, 290–291, 293
AssertionError, 290
asynchronous code
  advantage for Node, 32
  serial execution, 58, 59
  synchronous code, compared to, 30, 53–55
  testing, 295–297
authentication, MongoDB access, 217–218

B

base64, defined, 48
basicAuth middleware, 141–143
BDD (behavior-driven development), 297
belongsTo relationship type, 250
binary data, representing in JavaScript, 47–48
bind method, 24
block body declaration, 238
blocking code, 29–31, 32, 33, 46, 54
bodyParser middleware, 131–134, 208, 214, 251
boolean JavaScript type, 16
broadcast function, 171, 173–174, 186, 187
broadcasting issue, WebSocket, 177, 185–190
browser. *See also* code sharing
  Chrome, 1, 22
  Mocha, compiling for, 299–300
  Node's connection features, 94
  shimming browser APIs, 284
  WebSocket issues, 166, 177, 180
browserbuild project, 279, 285–288
Buffer object, 48
buffers, 47–48
bundling up client-side JavaScript, 128
byte orientation in TCP, 70–71

C

call method, 18
call stacks in v8, 32
case sensitive routes in Express, 153
casting in Mongoose, 225–226
Chrome, Google, 1, 22
chunked value for transfer encoding, 92–93
classes, 20, 46, 220, 254
CLI (command line interface), exploring APIs for, 63–66
client. *See also* browser
  bundling up JavaScript, 128
  createClient in MySQL, 234
  http.ClientRequest object, 105, 106
  IRC, 83–85
  mongo client, 207
  Socket.IO, 183–185
  telnet client, 71–73
  WebSocket, 169
client.end function, 236
client.query function, 236
close event, 77, 177, 180
closures, JavaScript, 19

cloud deployments, non-blocking IO advantages for, 33
code sharing (browser/server)
  browser APIs, shimming, 284
  browserbuild project, 285–288
  cross-browser inheritance, 284–285
  ECMA APIs, shimming, 282–283
  introduction, 279–280
  modules, exposing, 280–282
  Node APIs, shimming, 283–284
  summary, 288
Collection.insert function, 214
collections of documents in MongoDB, 205, 213–216
command line interface (CLI), exploring APIs for, 63–66
complex JavaScript types, 16
concurrency. *See* shared-state concurrency
configure function, 153
congestion control in TCP, 71
Connect API. *See also* middleware
  Connect-based website, creating, 119–120
  HTTP-based website, creating, 116–119
  introduction, 115–116
  summary, 144
connect event, 84, 180, 185–186
connect function, 84, 184, 220
connection event, 168, 181
Connection HTTP header, 92, 94
connections
  client/server communications as, 70
  HTTP, 93–94
  MongoDB access, 212–214
  node-mysql, 234
  redis, 267–268
  sequelize, 248–249
  tracking in TCP, 79–81
  WebSocket, 177
console object, 40
Content-Type HTTP header, 89, 99–100
contracts, event, defined, 47
controllers, separating HTML code from in Express, 146
convenience methods, Express, 155–157, 253
Cookie header, 134
createClient API, 234
createServer function
  Connect style, 120
  Express style, 147
  HTTP server, 112, 117
  TCP, 75
  WebSocket, 168
Crockford, Douglas (author)
  JavaScript: The Good Parts, 15
cross-browser inheritance, 284–285
crypto libraries, code sharing, 280
current working directory, 64–65

D

Dahl, Ryan (developer), 1
data deletion route, sequelize, 254–255
data event, 60–61, 77–79
data stream, client/server communication as, 70
data types, 207, 262–266
database access, non-blocking IO advantage for, 33. *See also* MongoDB; MySQL; Redis
datagram, defined, 70
:date dynamic token, 130
date manipulation toolkits, code sharing, 280
db.getLastError function, 220
db.query function, 241
default logging format, 129
delete class, 254
DELETE request, 99
deletion route, data, sequelize, 254–255
destroy command, 255
dev logging format, 129
_dirname, 64
disconnect event, 180
disconnecting and close event in WebSocket, 177
dispatchEvent API, 45–46
documents, MongoDB data as, 205, 206, 213–217, 222
dollar sign ($) prefix, 8
DOM API, code sharing (website), 282, 284
dot notation for nested keys in MongoDB, 222
dot slash (./), adding to require parameters, 43
driver APIs, defined, 208. *See also* node-mongodb-native driver; node-mysql driver; node-redis driver

E

ECMA APIs, shimming, 282–283
ECMAScript, 15
ejs template engine, 146–147, 154
elect function, 192
embedded documents, defining in Mongoose, 222
encoded string as parameter of event, 57
end event, 77
end method, 107
ensureIndex function, 216
environmental variables, 65
error event, 34, 47, 77
error handling
  AssertionError in testing, 290
  db.getLastError function, 220
  Express API, 150, 151, 155
  MySQL, 236
  Node, 34–35
  shared-state concurrency, 33–35
  stack traces, 35–37
es5-shim project (website), 282
event loop
  introduction, 1
  IO, relationship to, 32
  keeping it running with Stream, 56
  non-blocking code, relationship to, 30–31
  single thread of execution, 31
EventEmitter API, 45–46, 78, 284

events
close, 77, 177, 180
connect, 84, 180, 185–186
connection, 168, 181
data, 60–61, 77–79
disconnect, 180
encoded string as parameter, 57
end, 77
error, 34, 47, 77
join, 186
Node JavaScript additions, 45–47
open, 180
Socket.IO, 180–181, 185
exception, capturing, 21–22
executing a file, 10
exit API, 65
expect function, 292–293
expect.js project (website), 292–294
exports global object, 41, 44
exports style in Mocha, 298–299
exposing APIs, Node JavaScript additions, 44–45
Express API, 157–158. *See also* routes and route handlers
convenience methods, 155–157
error handling, 150, 151, 155
HTML, 146–147
introduction, 145
middleware, 159–160
module, creating, 146
MongoDB, 208–212, 213
node-mysql, 232–233
organization strategies, 160–162
search module, 150–151, 152
sequelize, 245–248
settings, 153–154
setup, 147–148
summary, 162
template engines, 146, 147, 154, 209
in WebSocket API, 167

F

file descriptors, 31, 56
filesystem, 92. *See also* fs module
filter method, 23
find method in sequelize, 253–254
findById query operation, 224
findOne command, 216
flag, defined, 186
flow control, 71, 121, 158
FLUSHALL command, 274
forEach method, 23
formidable module, 131
404 (Not Found) status code, 103, 117
framing, defined, 166
fs module
interacting with, 61–63
overview, 41
Stream, 66–68
sync compared to async style, 54
watch, 67–68
fs.createReadStream function, 67
fs.stat function, 58, 117
fs.watch function, 68
fs.WriteStream function, 67
function JavaScript type, 16
function name compared to variable name, 18
functions, overview, 18–19. *See also specific functions and methods*

G

-g flag, 13
GET command, 262
Get request, 99, 108, 109–110
GitHub, 3
global object, 40–41
Google Chrome, 1, 22
Grooveshark API (website), 192–196

H

handshake, defined, 166
hash data type in Redis, 263–264
hasMany association, 250
HDEL command, 264
header extension for Request, 155
header extension for Response, 156
headers, HTTP, 89–93, 99–100
HEXLISTS command, 262, 264
HGETALL command, 264, 273, 274
hgetall function, 269
hidden option in static middleware, 127
hmset function, 269
homebrew package manager, 261
HSET command, 263
HTML. *See* browser
HTML5 WebSocket, 166–167
HTTP (Hypertext Transfer Protocol)
Connect API, compared to, 116–119
connections, 93–94
headers, 89–93, 99–100
introduction, 87
reloading with up, 111–112
request+response model inefficiencies, 164–166
structure, 88–89
summary, 112
superagent module, 110–111
TCP, relationship to, 69, 88, 93
Twitter web client exercise, 104–110
web server, creating, 93–104
WebSocket, relationship to, 167
http.ClientRequest object, 105, 106
http.request object, 105, 106, 108, 109
http.Server object, 69, 182, 183
http.ServerRequest constructor, 88
http.ServerResponse constructor, 88
:http-version dynamic token, 130

I

if node block in browserbuild, 288
index method, 223
index setup for Mongoose, 215–216, 222–223
info object, 241
inheritance, 20–21, 25, 284–285
in-memory store, Redis, 260
input and output, basics in Node, 57–59

insert command, 214–215
instanceof operator, 17, 21
interpolation feature in jade, 240
io.connect function, 184
IP (Internet Protocol), 70
IRC (Internet Relay Chat) client program, 83–85
is extension for Request, 156
isarray method, 23
items array, database data, 239–240, 242–243

J

jade template engine, 154, 209–210, 238–240
JavaScript. *See also* Node, JavaScript additions
  bundling up client-side, 128
  classes, 20
  defined, 15
  functions, 18–19
  inheritance, 20–21
  JSON, compared to, 235
  Redis, compared to, 266
  summary, 26
  try/catch, 21–22
  types, 16–17
  v8, 22–26
JavaScript: The Good Parts (Crockford), 15
join event, 186
jQuery listener for sequelize data, 250–251
JSON
  in Connect, 135
  creating package.json, 12–13
  data format, 104
  encoding specification from v8, 24
  JavaScript, compared to, 235
  MongoDB design, relationship to, 206
  in sequelize, 252–253
  Socket.IO's coordination of events, 181
  in WebSocket, 173, 177
json extension for Response, 156
jsonp function, 154

K

keep-alive value for Connection header, 94
KEYS command, 261, 262
keys method, 22–23
key-value basis for Redis, 260, 263
Kvalheim, Christian Amor (developer), 208

L

lastIndexof method, 24
length property, 19
Linux systems, installing Node.JS on, 8–9
list data type in Redis, 265
listen method, 76, 112
locals object, 150
logger middleware, 129–131
login route, MongoDB, 215
login system, creating with Connect, 134–140
long polling, defined, 180
LPUSH command, 265
LRANGE command, 265

M

Mac OS X, installing Node.JS on, 8
make test command, 9
map method, 23
math libraries, code sharing, 280
maxAge option in static middleware, 127
:method dynamic token, 130
methodOverride middleware, 141
middleware
  Connect API
    basicAuth, 141–143
    bodyParser, 131–134
    Cookie header, 134
    defined, 115
    introduction, 121–122
    logger, 129–131
    methodOverride, 141
    query, 128
    RedisStore, 140
    session, 134–140
    static, 120, 127–128
    writing reusable, 122–127
  Express API, 159–160
  MongoDB, 208, 214, 217–218
  Mongoose, 223
  next function, relationship to, 124, 158
  sequelize, 251
mocha command, 295
Mocha framework, 294–300
Model.count function, 225
models, database
  Mongoose, 219, 220–221, 223–224, 227–229
  Redis, 268
  sequelize, 249–250
module global object, 41, 44, 280–281
module system
  assert, 284, 290–291, 293
  code sharing, 280–282
  creating the module, 12–13, 53
  Express API, 146, 150–151
  formidable, 131
  fs, 41, 54, 61–63, 66–68
  HTTP, 95, 104–110
  input and output, 57–59
  Node JavaScript additions, 41–43
  querystring, 101–102, 106, 108
  refactoring, 59–61
  streams, 55–57
  superagent, 110–111, 146, 151, 291–292
  synchronous compared to asynchronous code, 53–55
  TCP (chat program), 74
mongo client, 207
mongod server, 207
MongoDB
  compared to other database programs, 205–207, 260, 263
  defined, 205
  installing, 207
  Mongoose, 220–229
  node-mongodb-native driver
    application setup, 208
    atomicity, 219
    connecting to, 212–214
    creating documents, 214–215

Express app, creating, 208–212, 213
finding documents, 215–217
middleware, 208, 214, 217–218
safe mode, 219–220
validation, 218–219
summary, 229
mongodb.Server, 212–213
Mongoose
atomicity, 219
automatic key populating, 225
casting, 225–226
defining the model, 220–221
embedded documents, defining, 222
example program, 226–229
index setup, 215–216, 222–223
inspecting the model state, 223–224
limiting, 225
middleware, 223
nested keys, defining, 222
querying, 224–225
selecting, 224
skipping, 225
sorting, 224
monkey-patch (overriding functions), 123
mounting with static middleware, 127
MSI installer in Windows, 8
multi function, 269
multiplexing, defined, 182
MySQL
introduction, 231
MongoDB, compared to, 205, 206
node-mysql driver
connecting to, 234
creating data, 238–242
Express app, 232–233
fetching data, 242–244
initializing the script, 234–238
setup, 232
node-sequelize ORM
connecting to, 248–249
creating data, 250–253
defining models, 249–250
Express app, 245–248
introduction, 244–245
other functionality, 256–257
removing data, 254–256
retrieving data, 253–254
setup, 245
synchronizing to the database, 250
summary, 257

N

name method, 24
namespaces in Socket.IO, 181–182
nested keys, defining in Mongoose, 222
net.connect function, 84
net.createServer, 75
net.Server API, 74–76
net.Stream API, 75, 84
next function and middleware, 124, 158
nextTick function, 40
node command, 2, 9–10
Node Package Manager (NPM), 2, 10–14, 53
node-canvas (canvas 2D context) (website), 284
NODE_ENV, 65, 153
Node.JS. *See also specific APIs and database programs*
code sharing, 279–288
data in memory disadvantage, 266–267
installing, 7–14
introduction, 1–3
JavaScript additions
buffers, 47–48
events, 45–47
exposing APIs, 44–45
global object, 40–41
introduction, 39
module system, 41–43
summary, 48
shared-state concurrency
blocking compared to non-blocking code, 29–31
error handling, 33–35
introduction, 27–29
single thread of execution, 31–33
stack traces, 35–37
state, 79
summary, 37
support resources, 3
testing
expect.js project, 292–294
introduction, 289
Mocha framework, 294–300
program, 291–292
strategy, 290–291
subject for, 290
summary, 300
node_modules directory, 41
node-mongodb-native driver
application setup, 208
atomicity, 219
connecting to, 212–214
creating documents, 214–215
Express app, creating, 208–212, 213
finding documents, 215–217
middleware, 208, 214, 217–218
safe mode, 219–220
validation, 218–219
node-mysql driver
connecting to, 234
creating data, 238–242
Express app, 232–233
fetching data, 242–244
initializing the script, 234–238
sequelize, relationship to, 245
setup, 232
node-redis driver
connecting to redis, 267–268
defining the model, 268
graph methods, 269–270
intersections, computing, 270
testing, 270–276
users, creating and modifying, 268–269
node-sequelize ORM
connecting to, 248–249
creating data, 250–253
defining models, 249–250
Express app, 245–248
introduction, 244–245
other functionality, 256–257
removing data, 254–256
retrieving data, 253–254
setup, 245
summary, 257

non-blocking code, 29–31, 32, 33, 46, 54
nonprinting characters, 66
NoSQL, 206, 231
NPM (Node Package Manager), 2, 10–14, 53
npm install command, 12
npm publish command, 12, 13
null JavaScript type, 16

O

object JavaScript type, 16
ObjectID Schema type, 221, 225, 226
Object.keys API, code sharing issue, 283
Object-Relational Mappers (ORMs), 207. *See also* node-sequelize ORM
octets, 48
ODM (Object Document Mapper), 207
once method, 142
onload handler, 174
OOP frameworks, code sharing, 280
open event, 180
option function, 63
ORMs (Object-Relational Mappers), 207. *See also* node-sequelize ORM

P

package.json file, 12–13
parse method, 24
PATCH request, 99
paused state, 56
persistence, Redis data, 260
PHP compared to Node, 28–31
piping streams, 93
PKG file in Mac OS X, 8
port method, 112
ports package manager, 261
positions object, 172–173
POST request, 46–47, 99, 131
PostgreSQL, 205
primitive JavaScript types, 16
private variable, defined, 19
process object, 40
process.argv API, 63–64, 108
process.cwd object, 64–65
process.env object, 65
process.exit object, 65
process.on handler, 297
programming requirements, setting, 52
progressive API, defined, 111
proto, 25, 284
prototype extension in code sharing, 282–283
prototypical inheritance, defined, 20
PUT request, 99

Q

quality of service (QoS) for TCP, 71
query middleware, 128
query string in URLs, 101
querying database, 215–216, 224–225, 261–262
query.limit extension, 225
query.select extension for MongoDB, 224
query.skip extension for MongoDB, 225
query.sort extension for MongoDB, 224
querystring module, 101–102, 106, 108
querystringparse module, 102

R

RAW TCP mode, 72
Read-Eval-Print Loop (REPL), 2, 9–10
ReadStream filesystem APIs, 92
reception acknowledgment in Socket.IO, 190–191
reconnection issue in WebSocket, 177
redirect extension for Response, 157
Redis
  data types, 262–266
  installing, 261
  introduction, 259–260
  middleware, 140
  node-redis driver, 266–276
  query language, 261–262
  summary, 276
redis-cli command line execution, 261
RedisStore middleware, 140
reduce method, 24
refactoring, 59–61, 226–227
:referrer dynamic token, 130
RegExp object, 157–158
relative modules, 42
:remote-addr dynamic token, 130
removeEventListener API, 45–46
render extension for Response, 156
render method, 149, 150
REPL (Read-Eval-Print Loop), 2, 9–10
:req dynamic token, 130
req.params object, 157
Request object
  Express extensions, 155–156
  HTTP, 88, 93, 104–106, 108, 109–110, 164–166
  Node/JavaScript code sharing, 280, 284
require global object
  browserbuild considerations, 286–287
  exposing APIs, 44–45
  Express, 160
  JSON file loading, 235
  keyword, 14
  module system role, 41
  relative modules, 42–43
:res dynamic token, 130
Response object
  Express extensions, 156–157
  HTTP, 88, 93, 105, 164–166
:response-time dynamic token, 130
REST principles, 245
resume state, 56
review command, 14
routes and route handlers
  case-sensitive, 153
  data deletion route in sequelize, 254–255
  defining in Express, 148–150
  error handling, 155–157

middleware, 159–160
MongoDB, 210–211
organization strategy role (route maps), 160–162
strict, 154
terms defined, 145, 146
web application capabilities, 157–158
RPUSH command, 265

S

SADD command, 266
safe option, MongoDB driver, 219–220
Schema class, 220
schema-less database, MongoDB as, 205–206
scope definition in JavaScript, 19
<script> tag, 183–184
search command, 14
search module for Express Twitter app, 150–151, 152
Select command, 242–243
self-invoked function, defined, 19
semver versioning spec, 53
send extension for Response, 156
send object, 110–111, 166
sendfile extension for Response, 157
sequelize. *See* node-sequelize ORM
Sequelize constructor, 248–249
sequelize.define, 249
sequelize.sync, 250
serial execution, 58, 59
serve function, 118
serve(1) utility, 300
server. *See also* createServer function; HTTP
communications with client, 70
creating with Node.JS, 1–2, 10
MongoDB, 207, 212–213
net global object, 74–76
Server API, 69, 95–104, 182, 183
session data, 140, 266–267
session middleware, 134–140, 208
SET command, 262
set data type in Redis, 265–266
set object, 110–111
setEncoding method, 105
setImmediate API, 40
setTimeout function, 40
shared-state concurrency
blocking compared to non-blocking code, 29–31
error handling, 33–35
introduction, 27–29
single thread of execution, 31–33
stack traces, 35–37
state, 79
summary, 37
shimming of APIs, 280, 282–284
short logging format, 129
SIGKILL signal, 65
signals, 65–66
single thread of execution, 31–33
sio.listen function, 183
slash-r, slash-n (\r\n) delimiter, 85
SMEMBERS command, 262, 266
socket, TCP, 73
Socket.IO
chat program exercise
broadcasting, 185–190
client setup, 183–185
events, 185
reception acknowledgment, 190–191
server setup, 182–183
DJ-by-turns application exercise
extending chat, 191–192
Grooveshark API integration, 192–196
playing function, 196–201
introduction, 179
summary, 201
transports, 180–182
socket.io, program setup, 182
sorted sets in Redis, 266
sorting in database, 224
square brackets ([]) for uploading multiple files, 134
SREM command, 266
Stack Overflow (website), 3
stack trace, 24–25, 35–37
Stat object, 58
state variables and TCP, 79–81
static middleware, 120, 127–128
:status dynamic token, 130
stderr stream object, 56, 57
stdin stream object, 56, 59–60
stdio process, 51
stdout stream object, 51, 56, 57, 59–60
store for login sessions, setting up with Connect, 140
Stream objects
in first program module, 55–57
fs module, 66–68
net.createServer, 75–76
net.Stream API, 75, 84
types of, 51, 56–57, 59–60
strict routing in Express, 154
string data type in Redis, 263
string JavaScript type, 16
string methods in v8, 24
stringify method, 24, 106
String.prototype, 42
suite in TDD, 298
superagent module, 110–111, 146, 151, 291–292
synchronizing to database in sequelize, 250
synchronous code, 30, 53–55

T

tablescan lookup, defined, 216
TCP (Transmission Control Protocol)
characteristics of, 69–71
chat program exercise
creating the module, 74
data event, 77–79
disconnecting, 81–83
net.Server API, 74–76
receiving connections, 76–77
state variables, 79–81
HTTP, relationship to, 69, 88, 93
introduction, 69
IRC client program, 83–85
summary, 85
Telnet, 71–73
TDD (test-driven development), 298
telnet client, 71–73
template engines
code sharing, 280
ejs, 146–147, 154
jade, 154, 209–210, 238–240

tenants in cloud environment, 33
testing
  arrays, 23
  asynchronous code, 295–297
  expect.js project, 292–294
  introduction, 289
  Mocha framework, 294–300
  node-redis driver, 270–276
  program, 291–292
  strategy, 290–291
  subject for, 290
  summary, 300
test.js (Mocha), 295–300
text editor, 2
third-party modules, 41
time complexity, Redis, 262
timeouts
  setTimeout function, 40
  Socket.IO, 180
  TCP, 71
tiny logging format, 129
TinySong API, 193–196
Transfer-Encoding HTTP header, 92
Transmission Control Protocol. *See* TCP (Transmission Control Protocol)
transports, Socket.IO, 180–182
try/catch in JavaScript, 21–22
type options in Mongoose, 221
typeof operator, 17, 281

U

Ubuntu, 9
uncaughtException handler, 34
undefined JavaScript type, 16
Unix Standard Streams, 56
up executable, 111–112
updateAttributes method, 255
uploads, file, with bodyParser, 131–134
:url dynamic token, 130
url property, 97
/url property, 98
use function for static middleware, 120
:user-agent dynamic token, 130
utf8 string, 105

V

v8 JavaScript interpreter, 1, 17, 22–26, 32, 33
validate option in sequelize, 256
validation, MongoDB, 218–219
versioning spec for NPM, 53
view options parameter in Express, 147, 209, 232
vim text editor (website), 2
virtualized operating systems in cloud environment, 33

W

W3C, 166
watch function, 67–68, 112
Web 2.0, 164
web applications, rise of, 164
web browser. *See* browser
web server. *See* server
WebSocket. *See also* Socket.IO
  API compared to Protocol, 166
  background for, 163–167
  broadcasting issue, 177, 185–190
  client setups, 169
  close event and disconnecting, 177
  defined, 166
  echo example, 167–171
  JSON packet encoding/decoding, 173, 177
  mouse cursor example, 171–176
  Node/JavaScript code sharing, 280, 284
  reconnection issue, 177
  running the server, 170, 176
  server setups, 168, 172
  summary, 178
websocket.io, 167–168, 172
White, Nathan (developer), 207
window object, 40
Windows systems, installing Node.JS on, 8
working directory, CLI, 64–65
writeHead API, 90, 92

X

XCode, 8
XMLHttpRequest API, 280, 284

JS精品图书

《基于MVC的JavaScript Web富应用开发》

Alex MacCaw 著
李晶 张散集 译
2012年05月出版
ISBN：
978-7-121-10956-0
定 价：59.00元

《JavaScript语言精髓与编程实践》

周爱民 著
2012年03月出版
ISBN：
978-7-121-15640-3
定 价：79.00元

《JavaScript语言精粹（修订版）》

Douglas Crockford 著
赵泽欣 鄢学鹂 译
2012年09月出版
ISBN：
978-7-121-17740-8
定 价：49.00元

《用AngularJS开发下一代Web应用》

Green,B. Seshadri,S. 著
大漠穷秋 译
2013年10月出版
ISBN：
978-7-121-21574-2
定 价：55.00元

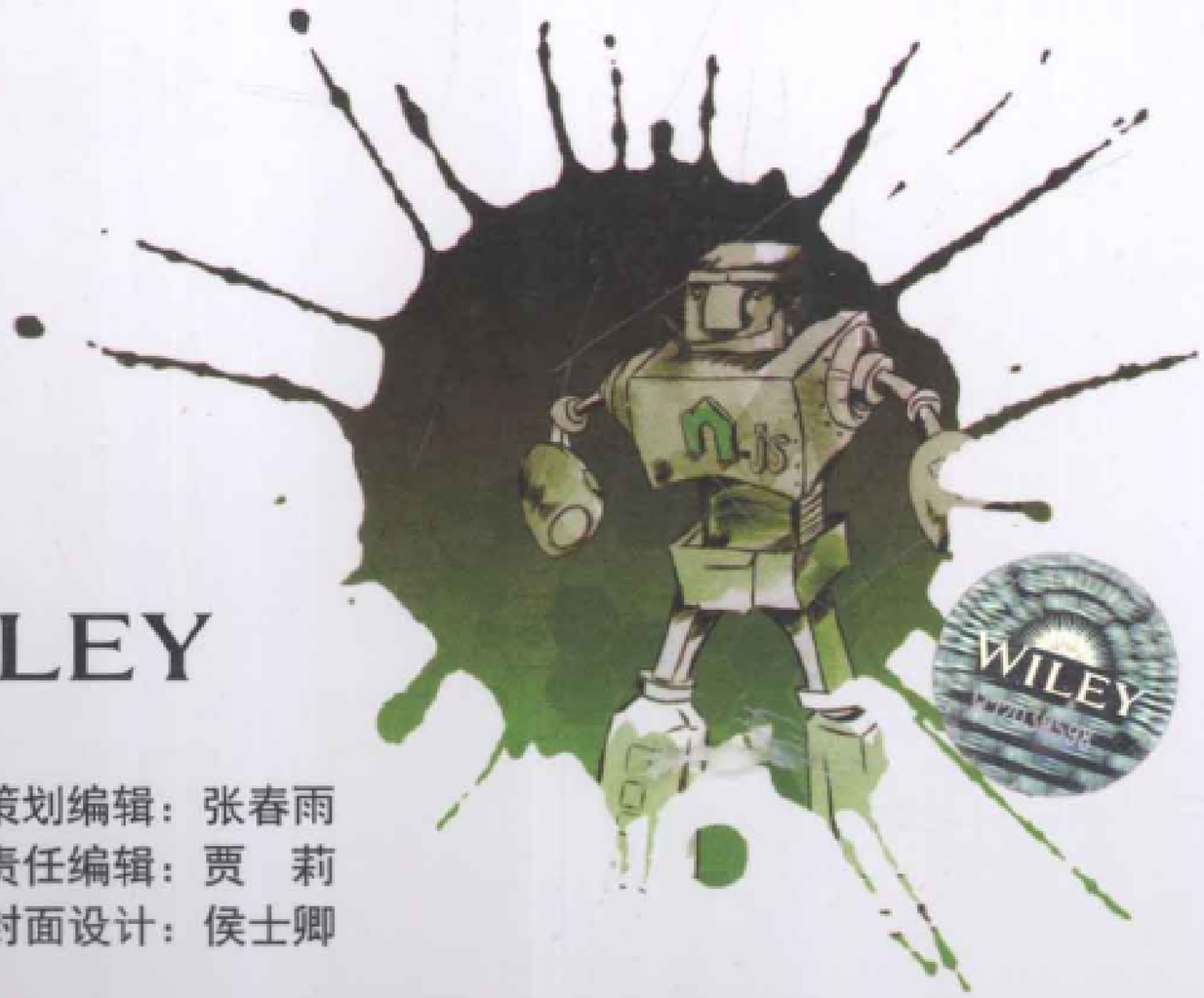
Broadview®
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

Node.js是一个由JavaScript书写而成的强大的Web开发框架，它让开发强壮的、伸缩性良好的服务器端Web应用变得更加简单、容易。本书向你展示了什么是Node以及如何让你在项目中使用它。本书包含大量实际应用中的示例程序，证明了为什么Node.js会快速成为Web开发首选工具，通过本书，你能够快速熟悉和掌握达到如下目标所需的Node知识和技能：

- **了解Node基于事件轮询的架构、无阻塞I/O以及事件驱动的编程方式**
- **精通Node.js的API**
- **轻松实现开发实时应用相关的技术，如Socket.IO和HTML5 WebSocket**
- **编写能够支持跨多台服务器的高并发应用**
- **通过Node来支持多种数据库以及数据存储工具**
- **编写在单台服务器情况下能够处理万级并发量的程序**
- **能够在一个包含更多Node知识和注解示例（含源代码）的网站上和其他开发者进行实时的沟通交流**

本书包含大量全彩插图和实用的源代码，绝对是一本革命性Web开发工具——Node的实用指南。

Guillermo Rauch（旧金山，加利福尼亚州）是一家位于旧金山，为当地教育提供相关服务的创业公司LearnBoost的CTO和联合创始人。Rauch还是几个知名Node.js项目的发明者，曾在JSConf和一些Node.js workshop做过演讲。



WILEY

策划编辑：张春雨
责任编辑：贾　莉
封面设计：侯士卿

上架建议：Web开发/JavaScript

ISBN 978-7-121-21769-2

9 787121 217692 >

定价：79.00元