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 266
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。

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
```

271

```
 */

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);
};

272 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) {
```

```
    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
```

273

```
 */

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。