Practical Perl Tools
Redis Meet Perl

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration* with Perl (the second edition of the Otter Book) available at purveyors of fine dead trees everywhere. He has spent the past 26+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA 2005 conference and one of the LISA 2006 Invited Talks co-chairs. David is honored to be the recipient of the 2009 SAGE Outstanding Achievement award and to serve on the USENIX Board of Directors.
dnb@ccs.neu.edu

One tool that you may have heard all of the cool kids(tm) are using today is Redis (http://redis.io). If you haven't heard of Redis, this column may introduce you to a lovely tool you can add to your repertoire. First we'll talk a little bit about what Redis is and why people like it. Once we do that, we can get into the Perl side of things. Just a quick warning for those of you who have seen this happen before in this column: the Perl stuff we're going to look at is a pretty straightforward layer on top of the basic Redis functionality. I'll consider an "Oh, is that all there is to it" reaction to be a good sign. But I just wanted to warn you lest you were hoping for gnarly Perl to impress your friends with at parties.

What Be Redis?

Redis is one of those packages that gets lumped into the NoSQL gang. These are software packages designed to help with certain scaling problems because they provide really simple but really fast storage and retrieval of data often with a little bit of "make it easy to distribute the data over several servers" thrown in. What they trade off in complexity around the storage and retrieval of data (ACID compliance, full query languages) is sheer performance and ease of use by other applications.

In many cases these software packages act like a "key-value" storage mechanism (i.e., like a Perl hash, you can store a value under an associated key for retrieval by that key later). An example of another well-known key-value store is the memcached package, something we may visit in a future column. Redis is a bit spiffier than a number of these packages because it actually understands more data types natively than most of the other players in the space.

In addition to this functionality, I believe Redis is well-liked by the people at those parties I mentioned before because it is fast, performant, stable, well-engineered, and quite easy to get started with right out of the box, even if you've never touched the thing. This may sound like what you hope every tool would be, but in my experience finding one isn't as easy as one would hope.

Redis Basics

Okay, so let's get into the fundamentals of using Redis. Redis gets packaged pretty easily so that you can usually find a way to install it quickly on the operating system of your choice. A second ago I grabbed the latest version to my laptop with "brew redis," but your copy could be almost an apt-get, yum, or wget/curl away (the source looks pretty easy to compile, although I have not done it). Given all of this, watch me hand wave about the installation (waves hands) so we can move right on to using the software. Redis comes with a massively commented sample config file (redis.conf), so feel free to bend it to your will. For this column, we're just going to assume you brought a Redis server up in its stock configuration (i.e., redis-server /path/to/redis.conf).

Redis comes with a command-line client (redis-cli), so we are going to play with it a bit before we show the Perl equivalent. This is similar to the approach we took when talking about RRDtool many moons ago. Let's use that client to do what every other key-value store can do first:

```
127.0.0.1:6379> SET usenix:address "2560 Ninth Street, Suite 215"
OK
127.0.0.1:6379> SET usenix:state "CA"
OK
127.0.0.1:6379> SET usenix:zip "94710"
OK
127.0.0.1:6379> GET usenix:address
"2560 Ninth Street, Suite 215"
```

Nothing exciting, right? If I tossed a ton of clients or millions of records and it did that, that would be cool but probably not all that exciting. A half a notch more exciting would be something like this:

```
127.0.0.1:6379> SET usenix:members 10
OK
127.0.0.1:6379> INCR usenix:members
(integer) 11
```

So why is that more exciting? Surely I could just do a GET and then a SET of the number of members + 1 (not the real number of members, by the way) instead of using a special increment operator. I could do that, but what if my typing or my script is kinda slow and some other person attempts to do the same operation? It is conceivable there will be a race condition in which I'll wind up incrementing a number that isn't the current one. INCR performs the operation in an "atomic" fashion, which means that you can have many separate clients incrementing the value and you don't have to worry about them stepping all over each other. There are a number of other fun things we can do to simple strings (append to them, get substrings, treat them like bit vectors, etc.). But let's get beyond strings…

More Redis Data Types

Although constructing all sorts of data structures in your application with just plain strings is possible, Redis makes it even easier for the programmer by internally supporting some of the more popular ones. For example, Redis handles lists for you trivially:

```
127.0.0.1:6379> LPUSH usenix:conferences LISA
(integer) 1
127.0.0.1:6379> LPUSH usenix:conferences OSDI
(integer) 2

127.0.0.1:6379> LRANGE usenix:conferences 0 -1
1) "OSDI"
2) "LISA"

127.0.0.1:6379> RPUSH usenix:conferences Security
(integer) 3
127.0.0.1:6379> RPUSH usenix:conferences FAST
(integer) 4

127.0.0.1:6379> LRANGE usenix:conferences 0 -1
1) "OSDI"
2) "LISA"
3) "Security"
4) "FAST"
```

The LPUSH command adds items to the left of the list (the front); RPUSH adds them to the right (the end). The LRANGE operator can be used to return parts of the list (using 0 and -1 means start at the first element and go to the end). There is a whole host of other list-related commands, including:

```
127.0.0.1:6379> RPOP usenix:conferences
"FAST"

127.0.0.1:6379> LRANGE usenix:conferences 0 -1
1) "OSDI"
2) "LISA"
3) "Security"
```

Here we've treated the list like a stack and popped the last element off the end using RPOP.

Let's Take a Perl Break

There are more data structures we should look at, but let's take a break to look at some Perl code. Here's a translation of the initial redis-cli example to Perl. In this column, we're going to be using the most popular Redis module (called Redis), although there are number of other choices available on CPAN:

```
use Redis;

my $redis = Redis->new;

# using usenix-from-perl as part of the key name just so it
# is clear on the server that the data is coming from this
# script and not the redis-cli command lines.
$redis->set( 'usenix-from-perl:address' =>
             '2560 Ninth Street, Suite 215' );
$redis->set( 'usenix-from-perl:state'   => 'CA' );
$redis->set( 'usenix-from-perl:zip'     => '94710' );

print $redis->get('usenix-from-perl:address'), "\n";
```

This prints out '2560 Ninth Street, Suite 215' as you'd expect. Perl code that uses Redis is an easy leap from the command-line example, no? Let's go back to the Redis data structures because we're going to run into a few of the even cooler Redis features shortly.

Two More Data Structures

I would be remiss if I didn't mention the two remaining supported data structures. The first, very familiar to Perl folks and mentioned early in this column, is the hash. It isn't immediately apparent what a hash type might mean when it comes to talking about a key-value store (which sounds like a hash already). Redis hashes are probably most equivalent to the Perl hash-of-hashes data structure. Each key in Redis is connected to a value that has associated fields (each with its own values). For example, let's make a hash that lists the cities where the LISA conference will be held in upcoming years:

```
127.0.0.1:6379> HSET usenix:lisa-conference 2014 "Seattle"
(integer) 1
127.0.0.1:6379> HSET usenix:lisa-conference 2015 "D.C."
(integer) 1
127.0.0.1:6379> HMSET usenix:lisa-conference 2016 "Boston" 2017
"San Francisco"
OK
```

Here we've used two different commands to populate the usenix:lisa-conference hash (which contains the years as fields). The first, HSET, sets a single field at a time. The second, HMSET, lets us set multiple fields at a time. Retrieving the info in each field can be done with (I bet you are seeing the pattern in names) HGET:

```
127.0.0.1:6379> HGET usenix:lisa-conference 2017
"San Francisco"
```

If we want to retrieve multiple fields, we can use HMGET with a list of the fields we want to retrieve:

```
127.0.0.1:6379> HMGET usenix:lisa-conference 2014 1016
1) "Seattle"
2) (nil)

127.0.0.1:6379> HMGET usenix:lisa-conference 2014 2016
1) "Seattle"
2) "Boston"
```

In the example above, I left in my typo so you can see what gets returned if you ask for a field that hasn't been set previously. (I don't actually know where LISA was in the year 1016; I only started attending in the 1900s.)

You can probably guess this, but just to make it explicit, the Perl equivalents of the commands above are direct translations:

```
$redis->hset( 'usenix-from-perl:lisa-conference',
'2014' => 'Seattle' );
$redis->hmset( 'usenix-from-perl:lisa-conference',
'2016' => 'Boston', '2017' => 'San Francisco' );
my $location = $redis->hget( 'usenix-from-perl:lisa-conference',
'2017' );
```

One last data structure type and then I want to show you two more magical things Redis can do with its data storage. The last data type is one that doesn't really have a direct analog to Perl's built-in data types: sets. Redis implements sets in two flavors: standard/unordered and sorted. The standard set is basically an unordered collection of elements that can be added to, subtracted from, tested for membership, and so on. And just like your junior high school days, you can perform operations between sets, such as finding their union or intersection. Let's use a set to keep track of the current USENIX board members. First we'll add them to the set:

```
127.0.0.1:6379> SADD usenix:board margo
(integer) 1
127.0.0.1:6379> SADD usenix:board john
(integer) 1
127.0.0.1:6379> SADD usenix:board carolyn
(integer) 1
127.0.0.1:6379> SADD usenix:board brian
(integer) 1
127.0.0.1:6379> SADD usenix:board david
(integer) 1
127.0.0.1:6379> SADD usenix:board niels
(integer) 1
127.0.0.1:6379> SADD usenix:board sasha
(integer) 1
127.0.0.1:6379> SADD usenix:board dan
(integer) 1
```

Now, if we show the members of the set, you'll see that they come back in a different order than they were added to the set (in this way, the lack of preserved order does resemble keys in a Perl hash):

```
127.0.0.1:6379> SMEMBERS usenix:board
1) "dan"
2) "john"
3) "carolyn"
4) "david"
5) "margo"
6) "niels"
7) "brian"
8) "sasha"
```

Once we have constructed our set, we can query to see whether an element is in it:

```
127.0.0.1:6379> SISMEMBER usenix:board niels
(integer) 1
127.0.0.1:6379> SISMEMBER usenix:board santa
(integer) 0
```

This test is fast, even with large sets (O(1) for you CS geeks). If we had defined multiple sets in this example, we could have determined how they differ, their intersections, and other fun things with a single Redis command.

Redis has a variation on sets called sorted sets. Sorted sets are like standard sets, except each member of the set has an associated "score." The score should be, according to the doc, "the string representation of a numeric value, and accepts double precision floating point numbers." The members of the set are kept in a sorted order based on this score. So, for example, if you wanted to model a leaderboard, you could create a sorted set using each member's score. As you rewrite each person in the set back to Redis with a new score, the members of the set rearrange themselves to stay sorted. This type of functionality can make parts of your application trivial to write. Let's do a toy example. If we had a bunch of different registration systems reporting back to a Redis instance the number of signups for each conference like so (yes, very fake numbers), we might have each system submit one of these commands:

```
127.0.0.1:6379> ZADD usenix:attendees 100 LISA
(integer) 1
127.0.0.1:6379> ZADD usenix:attendees 50 OSDI
(integer) 1
127.0.0.1:6379> ZADD usenix:attendees 30 FAST
(integer) 1
127.0.0.1:6379> ZADD usenix:attendees 75 Security
(integer) 1
127.0.0.1:6379> ZADD usenix:attendees 60 NSDI
(integer) 1
```

Now we can see the top three conferences listed in order of their attendance:

```
127.0.0.1:6379> ZRANGE usenix:attendees 0 2
1) "FAST"
2) "OSDI"
3) "NSDI"
```

Oh, wait, that's not right. That is indeed the first three conferences in the list in ascending order. We actually wanted to see the list sorted in descending order:

```
127.0.0.1:6379> ZREVRANGE usenix:attendees 0 2
1) "LISA"
2) "Security"
3) "NSDI"
```

Ah, much better. Sorted sets also make it super easy and super fast to find out where a particular member lives in the set:

```
127.0.0.1:6379> ZRANK usenix:attendees NSDI
(integer) 3
```

This says that NSDI can be found in the third place in the ranked order. Like the other data types, there are a whole slew of sorted set commands available. Instead of dwelling on them, let's finish the column by looking at two kinds of behind-the-scenes magic we can invoke.

Caching and Sub-ing

The first functionality I want to mention shows up in other key-value stores, but I still think it is kind of magic. Redis lets you set a time-to-live on any key. You can either set that value as the number of seconds a key should stick around via EXPIRE (which you can refresh using another EXPIRE), or provide a specific time for the expiration using EXPIREAT. Congratulations, you have self-maintaining cache.

The second thing Redis does that might make you squeal in delight is provide a special pub-sub mode. Pub-sub (i.e., publish-subscribe) is described in Wikipedia as

> "a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers. Instead, published messages are characterized into classes, without knowledge of what, if any, subscribers there may be. Similarly, subscribers express interest in one or more classes, and only receive messages that are of interest, without knowledge of what, if any, publishers there are."

In Redis (and in other contexts) the classes are called channels. A client will connect to the server and SUBSCRIBE to a set of channels either directly by name (e.g., SUBSCRIBE usenix) or via a globbing pattern like PSUBSCRIBE *usenix*' (for all channels with usenix in their name). If other clients use the PUBLISH usenix 'some message' command, the subscribed clients to that channel will get this message.

Because we're getting close to the end of the column, a Perl column, let's see a demonstration of this mode via a Perl sample. The one thing that makes this sample a little more complex than the previous translations is that pub-sub is coded up using callbacks. Callbacks are little snippets of code that are called when a message is received (vs. having some function you call that returns a value). As the Perl Redis module documentation states:

> "All Pub/Sub commands receive a callback as the last parameter. This callback receives three arguments:

- The published message.
- The topic over which the message was sent.
- The subscribed topic that matched the topic for the message. With 'subscribe' these last two are the same, always. But with 'psubscribe', this parameter tells you the pattern that matched."

The first thing we might write is a script that subscribes to some channels. It is as simple as this:

```
use Redis;
my $redis = Redis->new;

# subscribe to the LISA and Security channels
$redis->subscribe( 'LISA', 'Security',
    sub my ( $message, $channel, $subscribedchannel ) = @;
      print STDERR "Someone said something interesting:
           $message\n"; }, );
# loop, waiting for callbacks, or for 60 seconds to pass
$redis->wait_for_messages(60) while 1;
```

The code above connects to the server and subscribes to the LISA and Security channels. When we subscribe, we specify a subroutine that runs when a message comes in (it receives the arguments we just quoted from the doc) and prints out what it receives. Because this is a callback, if we just did a SUBSCRIBE, the script would subscribe to the channel and exit; it never would have the pleasure of smelling a freshly received message. We must tell it to wait around for messages and process callbacks, hence the last line that loops forever waiting for messages.

The script that actually publishes a message is trivial:

```
use Redis;

my $redis = Redis->new;
$redis->publish('LISA', 'Perl is great!');
```

If we launch the client and then run this script, it predictably prints out:

```
Someone said something interesting: Perl is great!
```

So with that example, I want to wrap up the column. We've only looked at a small subset of the commands Redis offers (for example, there are indeed commands for deleting info), and we haven't even mentioned some of the support for high-performance use, such as its pipelining feature. I'm hoping this brief look, along with a sense of how easy it is work with Redis from Perl once you know the commands, inspires you to go digging for more.

Take care and I'll see you next time.