The Little Redis Book

by Karl Seguin

About This Book

License

About The Author

Karl Seguin is a developer with experience across various fields and technologies. He's an active contributor to Open-Source Software projects, a technical writer and an occasional speaker. He's written various articles, as well as a few tools, about Redis. Redis powers the ranking and statistics of his free service for casual game developers: mogade.com.

Karl wrote The Little MongoDB Book, the free and popular book about MongoDB.

His blog can be found at http://openmymind.net and he tweets via [@karlseguin](http://twitter.com/karlseguin)

With Thanks To

A special thanks to Perry Neal for lending me his eyes, mind and passion. You provided me with invaluable help. Thank you.

Latest Version

The latest source of this book is available at: http://github.com/karlseguin/the-little-redis-book

Introduction

Over the last couple years, the techniques and tools used for persisting and querying data have grown at an incredible pace. While it's safe to say that relational databases aren't going anywhere, we can also say that the ecosystem around data is never going to be the same.

Of all the new tools and solutions, for me, Redis has been the most exciting. Why? First because it's unbelievably easy to learn. Hours is the right unit to use when talking about length of time it takes to get comfortable with Redis. Secondly, it solves a specific set of problems while at the same time being quite generic. What exactly does that mean? Redis doesn't try to be all things to all data. As you get to know Redis, it'll become increasingly evident what does and what does not belong in it. And when it does, as a developer, it's a great experience.

While you can build a complete system using Redis only, I think most people will find that it supplements their more generic data solution - whether that be a traditional relational database, a document-oriented system, or something else. It's the kind of solution you use to implement specific features. In that way, it's similar to an indexing engine. You wouldn't build your entire application on Lucene. But when you need good search, it's a much better experience - for both you and your users. Of course, the similarities between Redis and indexing engines end there.

The goal of this book is to build the foundation you'll need to master Redis. We'll focus on learning Redis' five data structures and look at various data modeling approaches. We'll also touch on some key administrative details and debugging techniques.

Getting Started

We all learn differently: some like to get their hands dirty, some like to watch videos, and some like to read. Nothing will help you understand Redis more than actually experiencing it. Redis is easy to install and comes with a simple shell that'll give us everything we need. Let's take a couple minutes and get it up and running on our machine.

On Windows

Redis itself doesn't officially support Windows, but there are options available. You wouldn't run these in production, but I've never experienced any limitations while doing development.

A port by Microsoft Open Technologies, Inc. can be found at https://github.com/MSOpenTech/redis. As of this writing the solution is not ready for use in production systems.

Another solution, which has been available for some time, can be found at https://github.com/dmajkic/redis/downloads. You can download the most up to date version (which should be at the top of the list). Extract the zip file and, based on your architecture, open either the `64bit` or `32bit` folder.

On *nix and MacOSX

For *nix and Mac users, building it from source is your best option. The instructions, along with the latest version number, are available at http://redis.io/download. At the time of this writing the latest version is 3.0.3; to install this version we would execute:

```
wget http://download.redis.io/releases/redis-3.0.3.tar.gz
tar xzf redis-3.0.3.tar.gz
cd redis-3.0.3
make
```

(Alternatively, Redis is available via various package managers. For example, MacOSX users with Homebrew installed can simply type `brew install redis`.)

If you built it from source, the binary outputs have been placed in the `src` directory. Navigate to the `src` directory by executing `cd src`.

Running and Connecting to Redis

If everything worked, the Redis binaries should be available at your fingertips. Redis has a handful of executables. We'll focus on the Redis server and the Redis command line interface (a DOS-like client). Let's start the server. In Windows, double click `redis-server`. On *nix/MacOSX run `./redis-server`.

If you read the start up message you'll see a warning that the `redis.conf` file couldn't be found. Redis will instead use built-in defaults, which is fine for what we'll be doing.

Next start the Redis console by either double clicking `redis-cli` (Windows) or running `./redis-cli` (*nix/MacOSX). This will connect to the locally-running server on the default port (6379).

You can test that everything is working by entering `info` into the command line interface. You'll hopefully see a bunch of key-value pairs which provide a great deal of insight into the server's status.

If you are having problems with the above setup I suggest you seek help in the official Redis support group.

Redis Drivers

As you'll soon learn, Redis' API is best described as an explicit set of functions. It has a very simple and procedural feel to it. This means that whether you are using the command line tool, or a driver for your favorite language, things are very similar. Therefore, you shouldn't have any problems following along if you prefer to work from a programming language. If you want, head over to the client page and download the appropriate driver.

Chapter 1 - The Basics

What makes Redis special? What types of problems does it solve? What should developers watch out for when using it? Before we can answer any of these questions, we need to understand what Redis is.

Redis is often described as an in-memory persistent key-value store. I don't think that's an accurate description. Redis does hold all the data in memory (more on this in a bit), and it does write that out to disk for persistence, but it's much more than a simple key-value store. It's important to step beyond this misconception otherwise your perspective of Redis and the problems it solves will be too narrow.

The reality is that Redis exposes five different data structures, only one of which is a typical key-value structure. Understanding these five data structures, how they work, what methods they expose and what you can model with them is the key to understanding Redis. First though, let's wrap our heads around what it means to expose data structures.

If we were to apply this data structure concept to the relational world, we could say that databases expose a single data structure - tables. Tables are both complex and flexible. There isn't much you can't model, store or manipulate with tables. However, their generic nature isn't without drawbacks. Specifically, not everything is as simple, or as fast, as it ought to be. What if, rather than having a one-size-fits-all structure, we used more specialized structures? There might be some things we can't do (or at least, can't do very well), but surely we'd gain in simplicity and speed?

Using specific data structures for specific problems? Isn't that how we code? You don't use a hashtable for every piece of data, nor do you use a scalar variable. To me, that defines Redis' approach. If you are dealing with scalars, lists, hashes, or sets, why not store them as scalars, lists, hashes and sets? Why should checking for the existence of a value be any more complex than calling `exists(key)` or slower than O(1) (constant time lookup which won't slow down regardless of how many items there are)?

The Building Blocks

Databases

Redis has the same basic concept of a database that you are already familiar with. A database contains a set of data. The typical use-case for a database is to group all of an application's data together and to keep it separate from another application's.

In Redis, databases are simply identified by a number with the default database being number `0`. If you want to change to a different database you can do so via the `select` command. In the command line interface, type `select 1`. Redis should reply with an OK message and your prompt should change to something like `redis 127.0.0.1:6379[1]>`. If you want to switch back to the default database, just enter `select 0` in the command line interface.

Commands, Keys and Values

While Redis is more than just a key-value store, at its core, every one of Redis' five data structures has at least a key and a value. It's imperative that we understand keys and values before moving on to other available pieces of information.

Keys are how you identify pieces of data. We'll be dealing with keys a lot, but for now, it's good enough to know that a key might look like `users:leto`. One could reasonably expect such a key to contain information about a user named `leto`. The colon doesn't have any special meaning, as far as Redis is concerned, but using a separator is a common approach people use to organize their keys.

Values represent the actual data associated with the key. They can be anything. Sometimes you'll store strings, sometimes integers, sometimes you'll store serialized objects (in JSON, XML or some other format). For the most part, Redis treats values as a byte array and doesn't care what they are. Note that different drivers handle serialization differently (some leave it up to you) so in this book we'll only talk about string, integer and JSON.

Let's get our hands a little dirty. Enter the following command:

```
set users:leto '{"name": "leto", "planet": "dune", "likes": ["spice"]}'
```

This is the basic anatomy of a Redis command. First we have the actual command, in this case `set`. Next we have its parameters. The `set` command takes two parameters: the key we are setting and the value we are setting it to. Many, but not all, commands take a key (and when they do, it's often the first parameter). Can you guess how to retrieve this value? Hopefully you said (but don't worry if you weren't sure!):

```
get users:leto
```

Go ahead and play with some other combinations. Keys and values are fundamental concepts, and the `get` and `set` commands are the simplest way to play with them. Create more users, try different types of keys, try different values.

Querying

As we move forward, two things will become clear. As far as Redis is concerned, keys are everything and values are nothing. Or, put another way, Redis doesn't allow you to query an object's values. Given the above, we can't find the user(s) which live on planet dune.

For many, this will cause some concern. We've lived in a world where data querying is so flexible and powerful that Redis' approach seems primitive and unpragmatic. Don't let it unsettle you too much. Remember, Redis isn't a one-size-fits-all solution. There'll be things that just don't belong in there (because of the querying limitations). Also, consider that in some cases you'll find new ways to model your data.

We'll look at more concrete examples as we move on, but it's important that we understand this basic reality of Redis. It helps us understand why values can be anything - Redis never needs to read or understand them. Also, it helps us get our minds thinking about modeling in this new world.

Memory and Persistence

We mentioned before that Redis is an in-memory persistent store. With respect to persistence, by default, Redis snapshots the database to disk based on how many keys have changed. You configure it so that if X number of keys change, then save the database every Y seconds. By default, Redis will save the database every 60 seconds if 1000 or more keys have changed all the way to 15 minutes if 9 or less keys has changed.

Alternatively (or in addition to snapshotting), Redis can run in append mode. Any time a key changes, an append-only file is updated on disk. In some cases it's acceptable to lose 60 seconds worth of data, in exchange for performance, should there be some hardware or software failure. In some cases such a loss is not acceptable. Redis gives you the option. In chapter 6 we'll see a third option, which is offloading persistence to a slave.

With respect to memory, Redis keeps all your data in memory. The obvious implication of this is the cost of running Redis: RAM is still the most expensive part of server hardware.

I do feel that some developers have lost touch with how little space data can take. The Complete Works of William Shakespeare takes roughly 5.5MB of storage. As for scaling, other solutions tend to be IO- or CPU-bound. Which limitation (RAM or IO) will require you to scale out to more machines really depends on the type of data and how you are storing and querying it. Unless you're storing large multimedia files in Redis, the in-memory aspect is probably a non-issue. For apps where it is an issue you'll likely be trading being IO-bound for being memory bound.

Redis did add support for virtual memory. However, this feature has been seen as a failure (by Redis' own developers) and its use has been deprecated.

(On a side note, that 5.5MB file of Shakespeare's complete works can be compressed down to roughly 2MB. Redis doesn't do auto-compression but, since it treats values as bytes, there's no reason you can't trade processing time for RAM by compressing/decompressing the data yourself.)

Putting It Together

We've touched on a number of high level topics. The last thing I want to do before diving into Redis is to bring some of those topics together. Specifically, query limitations, data structures and Redis' way to store data in memory.

When you add those three things together you end up with something wonderful: speed. Some people think "Of course Redis is fast, everything's in memory." But that's only part of it. The real reason Redis shines versus other solutions is its specialized data structures.

How fast? It depends on a lot of things - which commands you are using, the type of data, and so on. But Redis' performance tends to be measured in tens of thousands, or hundreds of thousands of operations **per second**. You can run `redis-benchmark` (which is in the same folder as the `redis-server` and `redis-cli`) to test it out yourself.

I once changed code which used a traditional model to using Redis. A load test I wrote took over 5 minutes to finish using the relational model. It took about 150ms to complete in Redis. You won't always get that sort of massive gain, but it hopefully gives you an idea of what we are talking about.

It's important to understand this aspect of Redis because it impacts how you interact with it. Developers with an SQL background often work at minimizing the number of round trips they make to the database. That's good advice for any system, including Redis. However, given that we are dealing with simpler data structures, we'll sometimes need to hit the Redis server multiple times to achieve our goal. Such data access patterns can feel unnatural at first, but in reality it tends to be an insignificant cost compared to the raw performance we gain.

In This Chapter

Although we barely got to play with Redis, we did cover a wide range of topics. Don't worry if something isn't crystal clear - like querying. In the next chapter we'll go hands-on and any questions you have will hopefully answer themselves.

The important takeaways from this chapter are:

- Keys are strings which identify pieces of data (values)
- Values are arbitrary byte arrays that Redis doesn't care about
- Redis exposes (and is implemented as) five specialized data structures
- Combined, the above make Redis fast and easy to use, but not suitable for every scenario

Chapter 2 - The Data Structures

It's time to look at Redis' five data structures. We'll explain what each data structure is, what methods are available and what type of feature/data you'd use it for.

The only Redis constructs we've seen so far are commands, keys and values. So far, nothing about data structures has been concrete. When we used the `set` command, how did Redis know what data structure to use? It turns out that every command is specific to a data structure. For example when you use `set` you are storing the value in a string data structure. When you use `hset` you are storing it in a hash. Given the small size of Redis' vocabulary, it's quite manageable.

Redis' website has great reference documentation. There's no point in repeating the work they've already done. We'll only cover the most important commands needed to understand the purpose of a data structure.

There's nothing more important than having fun and trying things out. You can always erase all the values in your database by entering `flushdb`, so don't be shy and try doing crazy things!

Strings

Strings are the most basic data structures available in Redis. When you think of a key-value pair, you are thinking of strings. Don't get mixed up by the name, as always, your value can be anything. I prefer to call them "scalars", but maybe that's just me.

We already saw a common use-case for strings, storing instances of objects by key. This is something that you'll make heavy use of:

```
set users:leto '{"name": leto, "planet": dune, "likes": ["spice"]}'
```

Additionally, Redis lets you do some common operations. For example `strlen <key>` can be used to get the length of a key's value; `getrange <key> <start> <end>` returns the specified range of a value; `append <key> <value>` appends the value to the existing value (or creates it if it doesn't exist already). Go ahead and try those out. This is what I get:

```
> strlen users:leto
(integer) 50

> getrange users:leto 31 48
"\"likes\": [\"spice\"]"

> append users:leto " OVER 9000!!"
(integer) 62
```

Now, you might be thinking, that's great, but it doesn't make sense. You can't meaningfully pull a range out of JSON or append a value. You are right, the lesson here is that some of the commands, especially with the string data structure, only make sense given specific type of data.

Earlier we learnt that Redis doesn't care about your values. Most of the time that's true. However, a few string commands are specific to some types or structure of values. As a vague example, I could see the above append and

getrange commands being useful in some custom space-efficient serialization. As a more concrete example I give you the `incr`, `incrby`, `decr` and `decrby` commands. These increment or decrement the value of a string:

```
> incr stats:page:about
(integer) 1
> incr stats:page:about
(integer) 2

> incrby ratings:video:12333 5
(integer) 5
> incrby ratings:video:12333 3
(integer) 8
```

As you can imagine, Redis strings are great for analytics. Try incrementing `users:leto` (a non-integer value) and see what happens (you should get an error).

A more advanced example is the `setbit` and `getbit` commands. There's a wonderful post on how Spool uses these two commands to efficiently answer the question "how many unique visitors did we have today". For 128 million users a laptop generates the answer in less than 50ms and takes only 16MB of memory.

It isn't important that you understand how bitmaps work, or how Spool uses them, but rather to understand that Redis strings are more powerful than they initially seem. Still, the most common cases are the ones we gave above: storing objects (complex or not) and counters. Also, since getting a value by key is so fast, strings are often used to cache data.

Hashes

Hashes are a good example of why calling Redis a key-value store isn't quite accurate. You see, in a lot of ways, hashes are like strings. The important difference is that they provide an extra level of indirection: a field. Therefore, the hash equivalents of `set` and `get` are:

```
hset users:goku powerlevel 9000
hget users:goku powerlevel
```

We can also set multiple fields at once, get multiple fields at once, get all fields and values, list all the fields or delete a specific field:

```
hmset users:goku race saiyan age 737
hmget users:goku race powerlevel
hgetall users:goku
hkeys users:goku
hdel users:goku age
```

As you can see, hashes give us a bit more control over plain strings. Rather than storing a user as a single serialized value, we could use a hash to get a more accurate representation. The benefit would be the ability to pull and update/delete specific pieces of data, without having to get or write the entire value.

Looking at hashes from the perspective of a well-defined object, such as a user, is key to understanding how they work. And it's true that, for performance reasons, more granular control might be useful. However, in the next chapter we'll look at how hashes can be used to organize your data and make querying more practical. In my opinion, this is where hashes really shine.

Lists

Lists let you store and manipulate an array of values for a given key. You can add values to the list, get the first or last value and manipulate values at a given index. Lists maintain their order and have efficient index-based operations. We could have a `newusers` list which tracks the newest registered users to our site:

```
lpush newusers goku
ltrim newusers 0 49
```

First we push a new user at the front of the list, then we trim it so that it only contains the last 50 users. This is a common pattern. `ltrim` is an O(N) operation, where N is the number of values we are removing. In this case, where we always trim after a single insert, it'll actually have a constant performance of O(1) (because N will always be equal to 1).

This is also the first time that we are seeing a value in one key referencing a value in another. If we wanted to get the details of the last 10 users, we'd do the following combination:

```
ids = redis.lrange('newusers', 0, 9)
redis.mget(*ids.map {|u| "users:#{u}"})
```

The above is a bit of Ruby which shows the type of multiple roundtrips we talked about before.

Of course, lists aren't only good for storing references to other keys. The values can be anything. You could use lists to store logs or track the path a user is taking through a site. If you were building a game, you might use one to track queued user actions.

Sets

Sets are used to store unique values and provide a number of set-based operations, like unions. Sets aren't ordered but they provide efficient value-based operations. A friend's list is the classic example of using a set:

```
sadd friends:leto ghanima paul chani jessica
sadd friends:duncan paul jessica alia
```

Regardless of how many friends a user has, we can efficiently tell (O(1)) whether userX is a friend of userY or not:

```
sismember friends:leto jessica
sismember friends:leto vladimir
```

Furthermore we can see whether two or more people share the same friends:

```
sinter friends:leto friends:duncan
```

and even store the result at a new key:

```
sinterstore friends:leto_duncan friends:leto friends:duncan
```

Sets are great for tagging or tracking any other properties of a value for which duplicates don't make any sense (or where we want to apply set operations such as intersections and unions).

Sorted Sets

The last and most powerful data structure are sorted sets. If hashes are like strings but with fields, then sorted sets are like sets but with a score. The score provides sorting and ranking capabilities. If we wanted a ranked list of friends, we might do:

```
zadd friends:duncan 70 ghanima 95 paul 95 chani 75 jessica 1 vladimir
```

Want to find out how many friends duncan has with a score of 90 or over?

```
zcount friends:duncan 90 100
```

How about figuring out chani's rank?

```
zrevrank friends:duncan chani
```

We use `zrevrank` instead of `zrank` since Redis' default sort is from low to high (but in this case we are ranking from high to low). The most obvious use-case for sorted sets is a leaderboard system. In reality though, anything you want sorted by some integer, and be able to efficiently manipulate based on that score, might be a good fit for a sorted set.

In This Chapter

That's a high level overview of Redis' five data structures. One of the neat things about Redis is that you can often do more than you first realize. There are probably ways to use string and sorted sets that no one has thought of yet. As long as you understand the normal use-case though, you'll find Redis ideal for all types of problems. Also, just because Redis exposes five data structures and various methods, don't think you need to use all of them. It isn't uncommon to build a feature while only using a handful of commands.

Chapter 3 - Leveraging Data Structures

In the previous chapter we talked about the five data structures and gave some examples of what problems they might solve. Now it's time to look at a few more advanced, yet common, topics and design patterns.

Big O Notation

Throughout this book we've made references to the Big O notation in the form of O(n) or O(1). Big O notation is used to explain how something behaves given a certain number of elements. In Redis, it's used to tell us how fast a command is based on the number of items we are dealing with.

Redis documentation tells us the Big O notation for each of its commands. It also tells us what the factors are that influence the performance. Let's look at some examples.

The fastest anything can be is O(1) which is a constant. Whether we are dealing with 5 items or 5 million, you'll get the same performance. The `sismember` command, which tells us if a value belongs to a set, is O(1). `sismember` is a powerful command, and its performance characteristics are a big reason for that. A number of Redis commands are O(1).

Logarithmic, or O(log(N)), is the next fastest possibility because it needs to scan through smaller and smaller partitions. Using this type of divide and conquer approach, a very large number of items quickly gets broken down in a few iterations. zadd is a O(log(N)) command, where N is the number of elements already in the sorted set.

Next we have linear commands, or O(N). Looking for a non-indexed column in a table is an O(N) operation. So is using the `ltrim` command. However, in the case of `ltrim`, N isn't the number of elements in the list, but rather the elements being removed. Using `ltrim` to remove 1 item from a list of millions will be faster than using `ltrim` to remove 10 items from a list of thousands. (Though they'll probably both be so fast that you wouldn't be able to time it.)

`zremrangebyscore` which removes elements from a sorted set with a score between a minimum and a maximum value has a complexity of O(log(N)+M). This makes it a mix. By reading the documentation we see that N is the number of total elements in the set and M is the number of elements to be removed. In other words, the number of elements that'll get removed is probably going to be more significant, in terms of performance, than the total number of elements in the set.

The `sort` command, which we'll discuss in greater detail in the next chapter has a complexity of O(N+M*log(M)). From its performance characteristic, you can probably tell that this is one of Redis' most complex commands.

There are a number of other complexities, the two remaining common ones are O(N^2) and O(C^N). The larger N is, the worse these perform relative to a smaller N. None of Redis' commands have this type of complexity.

It's worth pointing out that the Big O notation deals with the worst case. When we say that something takes O(N), we might actually find it right away or it might be the last possible element.

Pseudo Multi Key Queries

A common situation you'll run into is wanting to query the same value by different keys. For example, you might want to get a user by email (for when they first log in) and also by id (after they've logged in). One horrible solution is to duplicate your user object into two string values:

```
set users:leto@dune.gov '{"id": 9001, "email": "leto@dune.gov", ...}'
set users:9001 '{"id": 9001, "email": "leto@dune.gov", ...}'
```

This is bad because it's a nightmare to manage and it takes twice the amount of memory.

It would be nice if Redis let you link one key to another, but it doesn't (and it probably never will). A major driver in Redis' development is to keep the code and API clean and simple. The internal implementation of linking keys (there's a lot we can do with keys that we haven't talked about yet) isn't worth it when you consider that Redis already provides a solution: hashes.

Using a hash, we can remove the need for duplication:

```
set users:9001 '{"id": 9001, "email": "leto@dune.gov", ...}'
hset users:lookup:email leto@dune.gov 9001
```

What we are doing is using the field as a pseudo secondary index and referencing the single user object. To get a user by id, we issue a normal `get`:

```
get users:9001
```

To get a user by email, we issue an `hget` followed by a `get` (in Ruby):

```
id = redis.hget('users:lookup:email', 'leto@dune.gov')
user = redis.get("users:#{id}")
```

This is something that you'll likely end up doing often. To me, this is where hashes really shine, but it isn't an obvious use-case until you see it.

References and Indexes

We've seen a couple examples of having one value reference another. We saw it when we looked at our list example, and we saw it in the section above when using hashes to make querying a little easier. What this comes down to is essentially having to manually manage your indexes and references between values. Being honest, I think we can say that's a bit of a downer, especially when you consider having to manage/update/delete these references manually. There is no magic solution to solving this problem in Redis.

We already saw how sets are often used to implement this type of manual index:

```
sadd friends:leto ghanima paul chani jessica
```

Each member of this set is a reference to a Redis string value containing details on the actual user. What if `chani` changes her name, or deletes her account? Maybe it would make sense to also track the inverse relationships:

```
sadd friends_of:chani leto paul
```

Maintenance cost aside, if you are anything like me, you might cringe at the processing and memory cost of having these extra indexed values. In the next section we'll talk about ways to reduce the performance cost of having to do extra round trips (we briefly talked about it in the first chapter).

If you actually think about it though, relational databases have the same overhead. Indexes take memory, must be scanned or ideally seeked and then the corresponding records must be looked up. The overhead is neatly abstracted away (and they do a lot of optimizations in terms of the processing to make it very efficient).

Again, having to manually deal with references in Redis is unfortunate. But any initial concerns you have about the performance or memory implications of this should be tested. I think you'll find it a non-issue.

Round Trips and Pipelining

We already mentioned that making frequent trips to the server is a common pattern in Redis. Since it is something you'll do often, it's worth taking a closer look at what features we can leverage to get the most out of it.

First, many commands either accept one or more set of parameters or have a sister-command which takes multiple parameters. We saw `mget` earlier, which takes multiple keys and returns the values:

```
ids = redis.lrange('newusers', 0, 9)
redis.mget(*ids.map {|u| "users:#{u}"})
```

Or the `sadd` command which adds 1 or more members to a set:

```
sadd friends:vladimir piter
sadd friends:paul jessica leto "leto II" chani
```

Redis also supports pipelining. Normally when a client sends a request to Redis it waits for the reply before sending the next request. With pipelining you can send a number of requests without waiting for their responses. This reduces the networking overhead and can result in significant performance gains.

It's worth noting that Redis will use memory to queue up the commands, so it's a good idea to batch them. How large a batch you use will depend on what commands you are using, and more specifically, how large the parameters are. But, if you are issuing commands against ~50 character keys, you can probably batch them in thousands or tens of thousands.

Exactly how you execute commands within a pipeline will vary from driver to driver. In Ruby you pass a block to the `pipelined` method:

```
redis.pipelined do
  9001.times do
    redis.incr('powerlevel')
  end
end
```

As you can probably guess, pipelining can really speed up a batch import!

Transactions

Every Redis command is atomic, including the ones that do multiple things. Additionally, Redis has support for transactions when using multiple commands.

You might not know it, but Redis is actually single-threaded, which is how every command is guaranteed to be atomic. While one command is executing, no other command will run. (We'll briefly talk about scaling in a later chapter.) This is particularly useful when you consider that some commands do multiple things. For example:

`incr` is essentially a `get` followed by a `set`

`getset` sets a new value and returns the original

`setnx` first checks if the key exists, and only sets the value if it does not

Although these commands are useful, you'll inevitably need to run multiple commands as an atomic group. You do so by first issuing the `multi` command, followed by all the commands you want to execute as part of the transaction, and finally executing `exec` to actually execute the commands or `discard` to throw away, and not execute the commands. What guarantee does Redis make about transactions?

- The commands will be executed in order
- The commands will be executed as a single atomic operation (without another client's command being executed halfway through)
- That either all or none of the commands in the transaction will be executed

You can, and should, test this in the command line interface. Also note that there's no reason why you can't combine pipelining and transactions.

```
multi
hincrby groups:1percent balance -9000000000
hincrby groups:99percent balance 9000000000
exec
```

Finally, Redis lets you specify a key (or keys) to watch and conditionally apply a transaction if the key(s) changed. This is used when you need to get values and execute code based on those values, all in a transaction. With the code above, we wouldn't be able to implement our own `incr` command since they are all executed together once `exec` is called. From code, we can't do:

```
redis.multi()
current = redis.get('powerlevel')
redis.set('powerlevel', current + 1)
redis.exec()
```

That isn't how Redis transactions work. But, if we add a `watch` to `powerlevel`, we can do:

```
redis.watch('powerlevel')
current = redis.get('powerlevel')
redis.multi()
```

```
redis.set('powerlevel', current + 1)
redis.exec()
```

If another client changes the value of `powerlevel` after we've called `watch` on it, our transaction will fail. If no client changes the value, the set will work. We can execute this code in a loop until it works.

Keys Anti-Pattern

In the next chapter we'll talk about commands that aren't specifically related to data structures. Some of these are administrative or debugging tools. But there's one I'd like to talk about in particular: the `keys` command. This command takes a pattern and finds all the matching keys. This command seems like it's well suited for a number of tasks, but it should never be used in production code. Why? Because it does a linear scan through all the keys looking for matches. Or, put simply, it's slow.

How do people try and use it? Say you are building a hosted bug tracking service. Each account will have an `id` and you might decide to store each bug into a string value with a key that looks like `bug:account_id:bug_id`. If you ever need to find all of an account's bugs (to display them, or maybe delete them if they delete their account), you might be tempted (as I was!) to use the `keys` command:

```
keys bug:1233:*
```

The better solution is to use a hash. Much like we can use hashes to provide a way to expose secondary indexes, so too can we use them to organize our data:

```
hset bugs:1233 1 '{"id":1, "account": 1233, "subject": "..."}'
hset bugs:1233 2 '{"id":2, "account": 1233, "subject": "..."}'
```

To get all the bug ids for an account we simply call `hkeys bugs:1233`. To delete a specific bug we can do `hdel bugs :1233 2` and to delete an account we can delete the key via `del bugs:1233`.

In This Chapter

This chapter, combined with the previous one, has hopefully given you some insight on how to use Redis to power real features. There are a number of other patterns you can use to build all types of things, but the real key is to understand the fundamental data structures and to get a sense for how they can be used to achieve things beyond your initial perspective.

Chapter 4 - Beyond The Data Structures

While the five data structures form the foundation of Redis, there are other commands which aren't data structure specific. We've already seen a handful of these: `info`, `select`, `flushdb`, `multi`, `exec`, `discard`, `watch` and `keys`. This chapter will look at some of the other important ones.

Expiration

Redis allows you to mark a key for expiration. You can give it an absolute time in the form of a Unix timestamp (seconds since January 1, 1970) or a time to live in seconds. This is a key-based command, so it doesn't matter what type of data structure the key represents.

```
expire pages:about 30
expireat pages:about 1356933600
```

The first command will delete the key (and associated value) after 30 seconds. The second will do the same at 12:00 a.m. December 31st, 2012.

This makes Redis an ideal caching engine. You can find out how long an item has to live until via the `ttl` command and you can remove the expiration on a key via the `persist` command:

```
ttl pages:about
persist pages:about
```

Finally, there's a special string command, `setex` which lets you set a string and specify a time to live in a single atomic command (this is more for convenience than anything else):

```
setex pages:about 30 '<h1>about us</h1>....'
```

Publication and Subscriptions

Redis lists have an `blpop` and `brpop` command which returns and removes the first (or last) element from the list or blocks until one is available. These can be used to power a simple queue.

Beyond this, Redis has first-class support for publishing messages and subscribing to channels. You can try this out yourself by opening a second `redis-cli` window. In the first window subscribe to a channel (we'll call it `warnings`):

```
subscribe warnings
```

The reply is the information of your subscription. Now, in the other window, publish a message to the `warnings` channel:

```
publish warnings "it's over 9000!"
```

If you go back to your first window you should have received the message to the `warnings` channel.

You can subscribe to multiple channels (`subscribe channel1 channel2 ...`), subscribe to a pattern of channels (`psubscribe warnings:*`) and use the `unsubscribe` and `punsubscribe` commands to stop listening to one or more channels, or a channel pattern.

Finally, notice that the `publish` command returned the value `1`. This indicates the number of clients that received the message.

Monitor and Slow Log

The `monitor` command lets you see what Redis is up to. It's a great debugging tool that gives you insight into how your application is interacting with Redis. In one of your two redis-cli windows (if one is still subscribed, you can either use the `unsubscribe` command or close the window down and re-open a new one) enter the `monitor` command. In the other, execute any other type of command (like `get` or `set`). You should see those commands, along with their parameters, in the first window.

You should be wary of running monitor in production, it really is a debugging and development tool. Aside from that, there isn't much more to say about it. It's just a really useful tool.

Along with `monitor`, Redis has a `slowlog` which acts as a great profiling tool. It logs any command which takes longer than a specified number of **micro**seconds. In the next section we'll briefly cover how to configure Redis, for now you can configure Redis to log all commands by entering:

```
config set slowlog-log-slower-than 0
```

Next, issue a few commands. Finally you can retrieve all of the logs, or the most recent logs via:

```
slowlog get
slowlog get 10
```

You can also get the number of items in the slow log by entering `slowlog len`

For each command you entered you should see four parameters:

- An auto-incrementing id
- A Unix timestamp for when the command happened
- The time, in microseconds, it took to run the command
- The command and its parameters

The slow log is maintained in memory, so running it in production, even with a low threshold, shouldn't be a problem. By default it will track the last 1024 logs.

Sort

One of Redis' most powerful commands is `sort`. It lets you sort the values within a list, set or sorted set (sorted sets are ordered by score, not the members within the set). In its simplest form, it allows us to do:

```
rpush users:leto:guesses 5 9 10 2 4 10 19 2
sort users:leto:guesses
```

Which will return the values sorted from lowest to highest. Here's a more advanced example:

```
sadd friends:ghanima leto paul chani jessica alia duncan
sort friends:ghanima limit 0 3 desc alpha
```

The above command shows us how to page the sorted records (via `limit`), how to return the results in descending order (via `desc`) and how to sort lexicographically instead of numerically (via `alpha`).

The real power of `sort` is its ability to sort based on a referenced object. Earlier we showed how lists, sets and sorted sets are often used to reference other Redis objects. The `sort` command can dereference those relations and sort by the underlying value. For example, say we have a bug tracker which lets users watch issues. We might use a set to track the issues being watched:

```
sadd watch:leto 12339 1382 338 9338
```

It might make perfect sense to sort these by id (which the default sort will do), but we'd also like to have these sorted by severity. To do so, we tell Redis what pattern to sort by. First, let's add some more data so we can actually see a meaningful result:

```
set severity:12339 3
set severity:1382 2
set severity:338 5
set severity:9338 4
```

To sort the bugs by severity, from highest to lowest, you'd do:

```
sort watch:leto by severity:* desc
```

Redis will substitute the * in our pattern (identified via by) with the values in our list/set/sorted set. This will create the key name that Redis will query for the actual values to sort by.

Although you can have millions of keys within Redis, I think the above can get a little messy. Thankfully `sort` can also work on hashes and their fields. Instead of having a bunch of top-level keys you can leverage hashes:

```
hset bug:12339 severity 3
hset bug:12339 priority 1
hset bug:12339 details '{"id": 12339, ....}'

hset bug:1382 severity 2
hset bug:1382 priority 2
hset bug:1382 details '{"id": 1382, ....}'

hset bug:338 severity 5
hset bug:338 priority 3
hset bug:338 details '{"id": 338, ....}'
```

```
hset bug:9338 severity 4
hset bug:9338 priority 2
hset bug:9338 details '{"id": 9338, ....}'
```

Not only is everything better organized, and we can sort by `severity` or `priority`, but we can also tell `sort` what field to retrieve:

```
sort watch:leto by bug:*->priority get bug:*->details
```

The same value substitution occurs, but Redis also recognizes the -> sequence and uses it to look into the specified field of our hash. We've also included the `get` parameter, which also does the substitution and field lookup, to retrieve bug details.

Over large sets, `sort` can be slow. The good news is that the output of a `sort` can be stored:

```
sort watch:leto by bug:*->priority get bug:*->details store watch_by_priority:leto
```

Combining the `store` capabilities of `sort` with the expiration commands we've already seen makes for a nice combo.

Scan

In the previous chapter, we saw how the keys command, while useful, shouldn't be used in production. Redis 2.8 introduces the `scan` command which is production-safe. Although `scan` fulfills a similar purpose to `keys` there are a number of important difference. To be honest, most of the *differences* will seem like *idiosyncrasies*, but this is the cost of having a usable command.

First amongst these differences is that a single call to `scan` doesn't necessarily return all matching results. Nothing strange about paged results; however, `scan` returns a variable number of results which cannot be precisely controlled. You can provide a `count` hint, which defaults to 10, but it's entirely possible to get more or less than the specified `count`.

Rather than implementing paging through a `limit` and `offset`, `scan` uses a `cursor`. The first time you call `scan` you supply `0` as the cursor. Below we see an initial call to `scan` with an pattern match (optional) and a count hint (optional):

```
scan 0 match bugs:* count 20
```

As part of its reply, `scan` returns the next cursor to use. Alternatively, `scan` returns `0` to signify the end of results. Note that the next cursor value doesn't correspond to the result number or anything else which clients might consider useful.

A typical flow might look like this:

```
scan 0 match bugs:* count 2
> 1) "3"
> 2) 1) "bugs:125"
scan 3 match bugs:* count 2
> 1) "0"
```

```
> 2) 1) "bugs:124"
>    2) "bugs:123"
```

Our first call returned a next cursor (3) and one result. Our subsequent call, using the next cursor, returned the end cursor (0) and the final two results. The above is a *typical* flow. Since the `count` is merely a hint, it's possible for `scan` to return a next `cursor` (not 0) with no actual results. In other words, an empty result set doesn't signify that no additional results exist. Only a 0 cursor means that there are no additional results.

On the positive side, `scan` is completely stateless from Redis' point of view. So there's no need to close a cursor and there's no harm in not fully reading a result. If you want to, you can stop iterating through results, even if Redis returned a valid next cursor.

There are two other things to keep in mind. First, `scan` can return the same key multiple times. It's up to you to deal with this (likely by keeping a set of already seen values). Secondly, `scan` only guarantees that values which were present during the entire duration of iteration will be returned. If values get added or removed while you're iterating, they may or may not be returned. Again, this comes from `scan`'s statelessness; it doesn't take a snapshot of the existing values (like you'd see with many databases which provide strong consistency guarantees), but rather iterates over the same memory space which may or may not get modified.

In addition to `scan`, `hscan`, `sscan` and `zscan` commands were also added. These let you iterate through hashes, sets and sorted sets. Why are these needed? Well, just like `keys` blocks all other callers, so does the hash command `hgetall` and the set command `smembers`. If you want to iterate over a very large hash or set, you might consider making use of these commands. `zscan` might seem less useful since paging through a sorted set via `zrangebyscore` or `zrangebyrank` is already possible. However, if you want to fully iterate through a large sorted set, `zscan` isn't without value.

In This Chapter

This chapter focused on non-data structure-specific commands. Like everything else, their use is situational. It isn't uncommon to build an app or feature that won't make use of expiration, publication/subscription and/or sorting. But it's good to know that they are there. Also, we only touched on some of the commands. There are more, and once you've digested the material in this book it's worth going through the full list.

Chapter 5 - Lua Scripting

Redis 2.6 includes a built-in Lua interpreter which developers can leverage to write more advanced queries to be executed within Redis. It wouldn't be wrong of you to think of this capability much like you might view stored procedures available in most relational databases.

The most difficult aspect of mastering this feature is learning Lua. Thankfully, Lua is similar to most general purpose languages, is well documented, has an active community and is useful to know beyond Redis scripting. This chapter won't cover Lua in any detail; but the few examples we look at should hopefully serve as a simple introduction.

Why?

Before looking at how to use Lua scripting, you might be wondering why you'd want to use it. Many developers dislike traditional stored procedures, is this any different? The short answer is no. Improperly used, Redis' Lua scripting can result in harder to test code, business logic tightly coupled with data access or even duplicated logic.

Properly used however, it's a feature that can simplify code and improve performance. Both of these benefits are largely achieved by grouping multiple commands, along with some simple logic, into a custom-build cohesive function. Code is made simpler because each invocation of a Lua script is run without interruption and thus provides a clean way to create your own atomic commands (essentially eliminating the need to use the cumbersome `watch` command). It can improve performance by removing the need to return intermediary results - the final output can be calculated within the script.

The examples in the following sections will better illustrate these points.

Eval

The `eval` command takes a Lua script (as a string), the keys we'll be operating against, and an optional set of arbitrary arguments. Let's look at a simple example (executed from Ruby, since running multi-line Redis commands from its command-line tool isn't fun):

```
script = <<-eos
  local friend_names = redis.call('smembers', KEYS[1])
  local friends = {}
  for i = 1, #friend_names do
    local friend_key = 'user:' .. friend_names[i]
    local gender = redis.call('hget', friend_key, 'gender')
    if gender == ARGV[1] then
      table.insert(friends, redis.call('hget', friend_key, 'details'))
    end
  end
  return friends
eos
Redis.new.eval(script, ['friends:leto'], ['m'])
```

The above code gets the details for all of Leto's male friends. Notice that to call Redis commands within our script we use the `redis.call("command", ARG1, ARG2, ...)` method.

If you are new to Lua, you should go over each line carefully. It might be useful to know that `{}` creates an empty `table` (which can act as either an array or a dictionary), #TABLE gets the number of elements in the TABLE, and `..` is used to concatenate strings.

`eval` actually take 4 parameters. The second parameter should actually be the number of keys; however the Ruby driver automatically creates this for us. Why is this needed? Consider how the above looks like when executed from the CLI:

```
eval "....." "friends:leto" "m"
vs
eval "....." 1 "friends:leto" "m"
```

In the first (incorrect) case, how does Redis know which of the parameters are keys and which are simply arbitrary arguments? In the second case, there is no ambiguity.

This brings up a second question: why must keys be explicitly listed? Every command in Redis knows, at execution time, which keys are going to needed. This will allow future tools, like Redis Cluster, to distribute requests amongst multiple Redis servers. You might have spotted that our above example actually reads from keys dynamically (without having them passed to `eval`). An `hget` is issued on all of Leto's male friends. That's because the need to list keys ahead of time is more of a suggestion than a hard rule. The above code will run fine in a single-instance setup, or even with replication, but won't in the yet-released Redis Cluster.

Script Management

Even though scripts executed via `eval` are cached by Redis, sending the body every time you want to execute something isn't ideal. Instead, you can register the script with Redis and execute it's key. To do this you use the `script load` command, which returns the SHA1 digest of the script:

```
redis = Redis.new
script_key = redis.script(:load, "THE_SCRIPT")
```

Once we've loaded the script, we can use `evalsha` to execute it:

```
redis.evalsha(script_key, ['friends:leto'], ['m'])
```

`script kill`, `script flush` and `script exists` are the other commands that you can use to manage Lua scripts. They are used to kill a running script, removing all scripts from the internal cache and seeing if a script already exists within the cache.

Libraries

Redis' Lua implementation ships with a handful of useful libraries. While `table.lib`, `string.lib` and `math.lib` are quite useful, for me, `cjson.lib` is worth singling out. First, if you find yourself having to pass multiple arguments to

a script, it might be cleaner to pass it as JSON:

```
redis.evalsha ".....", [KEY1], [JSON.fast_generate({gender: 'm', ghola: true})]
```

Which you could then deserialize within the Lua script as:

```
local arguments = cjson.decode(ARGV[1])
```

Of course, the JSON library can also be used to parse values stored in Redis itself. Our above example could potentially be rewritten as such:

```
local friend_names = redis.call('smembers', KEYS[1])
local friends = {}
for i = 1, #friend_names do
  local friend_raw = redis.call('get', 'user:' .. friend_names[i])
  local friend_parsed = cjson.decode(friend_raw)
  if friend_parsed.gender == ARGV[1] then
    table.insert(friends, friend_raw)
  end
end
return friends
```

Instead of getting the gender from specific hash field, we could get it from the stored friend data itself. (This is a much slower solution, and I personally prefer the original, but it does show what's possible).

Atomic

Since Redis is single-threaded, you don't have to worry about your Lua script being interrupted by another Redis command. One of the most obvious benefits of this is that keys with a TTL won't expire half-way through execution. If a key is present at the start of the script, it'll be present at any point thereafter - unless you delete it.

Administration

The next chapter will talk about Redis administration and configuration in more detail. For now, simply know that the `lua-time-limit` defines how long a Lua script is allowed to execute before being terminated. The default is generous 5 seconds. Consider lowering it.

In This Chapter

This chapter introduced Redis' Lua scripting capabilities. Like anything, this feature can be abused. However, used prudently in order to implement your own custom and focused commands, it won't only simplify your code, but will likely improve performance. Lua scripting is like almost every other Redis feature/command: you make limited, if any, use of it at first only to find yourself using it more and more every day.

Chapter 6 - Administration

Our last chapter is dedicated to some of the administrative aspects of running Redis. In no way is this a comprehensive guide on Redis administration. At best we'll answer some of the more basic questions new users to Redis are most likely to have.

Configuration

When you first launched the Redis server, it warned you that the `redis.conf` file could not be found. This file can be used to configure various aspects of Redis. A well-documented `redis.conf` file is available for each release of Redis. The sample file contains the default configuration options, so it's useful to both understand what the settings do and what their defaults are. You can find it at http://download.redis.io/redis-stable/redis.conf.

Since the file is well-documented, we won't be going over the settings.

In addition to configuring Redis via the `redis.conf` file, the `config set` command can be used to set individual values. In fact, we already used it when setting the `slowlog-log-slower-than` setting to 0.

There's also the `config get` command which displays the value of a setting. This command supports pattern matching. So if we want to display everything related to logging, we can do:

```
config get *log*
```

Authentication

Redis can be configured to require a password. This is done via the `requirepass` setting (set through either the `redis.conf` file or the `config set` command). When `requirepass` is set to a value (which is the password to use), clients will need to issue an `auth password` command.

Once a client is authenticated, they can issue any command against any database. This includes the `flushall` command which erases every key from every database. Through the configuration, you can rename commands to achieve some security through obfuscation:

```
rename-command CONFIG 5ec4db169f9d4dddacbfb0c26ea7e5ef
rename-command FLUSHALL 1041285018a942a4922cbf76623b741e
```

Or you can disable a command by setting the new name to an empty string.

Size Limitations

As you start using Redis, you might wonder "how many keys can I have?" You might also wonder how many fields can a hash have (especially when you use it to organize your data), or how many elements can lists and sets have? Per instance, the practical limits for all of these is in the hundreds of millions.

Replication

Redis supports replication, which means that as you write to one Redis instance (the master), one or more other instances (the slaves) are kept up-to-date by the master. To configure a slave you use either the `slaveof` configuration setting or the `slaveof` command (instances running without this configuration are or can be masters).

Replication helps protect your data by copying to different servers. Replication can also be used to improve performance since reads can be sent to slaves. They might respond with slightly out of date data, but for most apps that's a worthwhile tradeoff.

Unfortunately, Redis replication doesn't yet provide automated failover. If the master dies, a slave needs to be manually promoted. Traditional high-availability tools that use heartbeat monitoring and scripts to automate the switch are currently a necessary headache if you want to achieve some sort of high availability with Redis.

Backups

Backing up Redis is simply a matter of copying Redis' snapshot to whatever location you want (S3, FTP, …). By default Redis saves its snapshot to a file named `dump.rdb`. At any point in time, you can simply `scp`, `ftp` or `cp` (or anything else) this file.

It isn't uncommon to disable both snapshotting and the append-only file (aof) on the master and let a slave take care of this. This helps reduce the load on the master and lets you set more aggressive saving parameters on the slave without hurting overall system responsiveness.

Scaling and Redis Cluster

Replication is the first tool a growing site can leverage. Some commands are more expensive than others (`sort` for example) and offloading their execution to a slave can keep the overall system responsive to incoming queries.

Beyond this, truly scaling Redis comes down to distributing your keys across multiple Redis instances (which could be running on the same box, remember, Redis is single-threaded). For the time being, this is something you'll need to take care of (although a number of Redis drivers do provide consistent-hashing algorithms). Thinking about your data in terms of horizontal distribution isn't something we can cover in this book. It's also something you probably won't have to worry about for a while, but it's something you'll need to be aware of regardless of what solution you use.

The good news is that work is under way on Redis Cluster. Not only will this offer horizontal scaling, including rebalancing, but it'll also provide automated failover for high availability.

High availability and scaling is something that can be achieved today, as long as you are willing to put the time and effort into it. Moving forward, Redis Cluster should make things much easier.

In This Chapter

Given the number of projects and sites using Redis already, there can be no doubt that Redis is production-ready, and has been for a while. However, some of the tooling, especially around security and availability is still young. Redis

Cluster, which we'll hopefully see soon, should help address some of the current management challenges.

Conclusion

In a lot of ways, Redis represents a simplification in the way we deal with data. It peels away much of the complexity and abstraction available in other systems. In many cases this makes Redis the wrong choice. In others it can feel like Redis was custom-built for your data.

Ultimately it comes back to something I said at the very start: Redis is easy to learn. There are many new technologies and it can be hard to figure out what's worth investing time into learning. When you consider the real benefits Redis has to offer with its simplicity, I sincerely believe that it's one of the best investments, in terms of learning, that you and your team can make.