The Little Redis Book

by Karl Seguin

traducido por
Raúl Expósito

Sobre este libro

Licencia

Sobre El Autor

Karl Seguin es un desarrollador con experiencia en varias áreas y tecnologias. Es un contribuidor activo en proyectos Open Source, un escritor técnico y un orador ocasional. Ha escrito varios artículos, al igual que varias herramientas, sobre Redis. Redis posibilita tanto el ranking como las estadísticas de su servicio para desarrolladores ocasionales de videojuegos: mogade.com.

Karl escribió El Pequeño Libro de MongoDB, el popular y gratuito libro sobre MongoDB.

Puede encontrase su blog en http://openmymind.net y twittea vía @karlseguin

Sobre El Traductor

Este libro ha sido traducido por Raúl Expósito. Puedes visitar su página web en http://raulexposito.com/

Agradecimientos

Quiero agradecerle especialmente a Perry Neal el haberme prestado su visión, su mentalidad y su pasión. Me brindaste una ayuda que no tiene precio. Gracias.

Última versión

La última versión del código fuente de este libro puede encontrarse en: http://github.com/raulexposito/the-little-redis-book

Introducción

En el último par de años, las técnicas y herramientas utilizadas para el almacenamiento y consulta de datos han crecido a un ritmo increíble. Es seguro afirmar que las bases de datos relacionales ya no van a ir a ninguna parte, del mismo modo que es posible decir que el ecosistema generado en torno a los datos nunca volverá a ser el mismo.

De todas las nuevas herramientas y soluciones, para mí, Redis ha sido la más emocionante. ¿Por qué?. El primer lugar porque es increíblemente fácil de aprender. La unidad correcta a emplear es horas cuando se habla de la cantidad de tiempo que se necesita hasta sentirse cómodo con Redis. En segundo lugar, porque soluciona un conjunto específico de problemas a la vez que se mantiene bastante genérico. ¿Qué significa esto exactamente? Redis no intenta hacer de todo con todo tipo de datos. A medida que vayas aprendiendo Redis, verás de un modo cada vez más evidente qué funciona y qué no funciona con él. Y lo que funciona, como desarrollador, genera una grata experiencia.

Mientras que es posible construir un sistema completo utilizando únicamente Redis, creo que mucha gente lo considerará como un complemento a aquello que esté utilizando para almacenar datos de forma genérica - que puede ser una base de datos relacional tradicional, un sistema orientado a documentos, o cualquier otra cosa. Es el tipo de solución que se emplea para implementar características específicas. De este modo, es similar a un motor de indexado. No tratarías de construir tu aplicación completa con Lucene, pero cuando necesitas realizar buenas búsquedas, da una experiencia mucho más satisfactoria - tanto para tí como para tus usuarios. Por supuesto, las similitudes entre Redis y los motores de indexado terminan aquí.

El objetivo de este libro es permitirte asentar las bases que necesitarás para ser un maestro de Redis. Nos centraremos en cinco estructuras de datos de Redis y veremos varias aproximaciones a modelos de datos. También veremos algunos asuntos administrativos y técnicas de depuración.

Comienzo

Todos tenemos un modo distinto de aprender: unos prefieren ensuciarse las manos, otros ver vídeos y a otros les gusta leer. Nada te ayudará mejor a entender Redis que experimentar con él. Redis es realmente sencillo de instalar y viene con una shell sencilla que nos dará todo lo que necesitemos. Vamos a emplear un par de minutos en tenerlo instalado y funcionando en nuestro equipo.

En Windows

Redis no da soporte oficial en Windows, pero existen opciones disponibles. Probablemente no quieras ejecutarlo en un entorno de producción, pero no he sufrido ninguna limitación mientras lo utilizaba para desarrollar.

Puedes obtener una versión de Microsoft Open Technologies, Inc. en https://github.com/MSOpenTech/redis. En el momento de escribir estas líneas esta opción no está preparada para un uso real en sistemas de producción.

Otra solución, que ha estado disponible durante algún tiempo, puede obtenerse en https://github.com/dmajkic/redis/downloads. Puedes descargar la versión más acutalizada (que debería estar en la parte superior de la lista). Descomprime el zip y, dependiendo de tu arquitectura, abre el directorio de 64bit o 32bit.

En *nix y MacOSX

Para los usuario de *nix y Mac, la construcción desde el código fuente es la mejor opción. Las instrucciones, junto con la última versión, están disponibles en http://redis.io/download. En el momento de escribir esto la última versión es la 2.6.2; para instalar esta versión debemos ejecutar:

```
wget http://redis.googlecode.com/files/redis-2.6.2.tar.gz
tar xzf redis-2.6.2.tar.gz
cd redis-2.6.2
make
```

(De forma alternativa, Redis está disponible a través de varios gestores de paquetes. Por ejemplo, los usuarios de MacOSX que tengan Homebrew instalado pueden simplemente ejecutar el comando `brew install redis`.)

Si lo construyes desde el código fuente, los ejecutables binarios se encontrarán en el directorio src. Navega hacia el directorio src ejecutando `cd src`.

Ejecutando y Conectando con Redis

Si todo ha funcionado los binarios de Redis deberían estar a tu alcance. Redis tiene un puñado de ejecutables. Nos vamos a centrar en el servidor Redis y en la interfaz por línea de comandos de Redis (un cliente similar a MSDOS). Vamos a comenzar por el servidor. En Windows hay que hacer doble click en `redis-server`. En *nix/MacOSX ejecuta `./redis-server`.

Si lees el mensaje de arranque verás una advertencia que indica que el fichero `redis.conf` no existe. Redis en su lugar empleará valores prefedinidos, lo cual irá bien para lo que vamos a hacer.

El siguiente paso es ejecutar la consola de Redis, haciendo doble click sobre `redis-cli` (Windows) o ejecutando `./redis-cli` (*nix/MacOSX). Conectará con el servidor que se ejecuta en la misma máquina en el puerto por defecto (6379).

Puedes probar que todo funciona correctamente escribiendo `info` en la interfaz por línea de comandos. Veremos un montón de pares clave-valor que indicarán cuál es el estado del servidor.

Si tienes problemas con la configuración indicada anteriormente te sugiero que busques ayuda en el grupo oficial de soporte de Redis.

Controladores para Redis

Como pronto aprenderás, la mejor manera de describir el API de Redis es como un conjunto explícito de funciones. Esto significa que cuando estás utilizando la línea de comandos, o un controlador para tu lenguaje favorito, las cosas se hacen de una manera muy similar. Por lo tanto, no deberías tener problemas si prefieres trabajar con Redis desde un lenguaje de programación. Si quieres, echa un vistazo a la página de clientes y descarga el controlador adecuado.

Capítulo 1 - Los Fundamentos

¿Qué hace a Redis especial?, ¿Qué tipos de problemas soluciona?, ¿Qué deberían esperar los desarrolladores cuando lo utilizan?. Antes de poder responder a estas preguntas, necesitamos entender qué es Redis.

Redis suele ser descrito como un almacén en memoria de conjuntos clave-valor. Yo no creo que ésta sea una descripción adecuada. Redis mantiene todos los datos en memoria (más sobre esto en breve), y los escribe en disco para persistirlos, pero es mucho más que un simple almacén de conjuntos clave-valor. Es importante ir un paso más allá de este concepto erróneo, porque sino tu perspectiva sobre Redis y los problemas que soluciona estarán demasiado acotados.

La realidad es que Redis ofrece cinco estructuras diferentes de datos, una de las cuales es la típica estructura clave-valor. Entender estas cinco estructuras de datos, cómo trabajan, qué métodos exponen y qué puedes modelar con ellos es la clave para entender Redis. Sin embargo, primero vamos a ver qué son éstas estructuras de datos.

Si estuviéramos aplicando este concepto de estructura de datos en el mundo de las bases de datos relacionales, podríamos decir que estas bases de datos ofrecen una única estructura de datos - tablas. Las tablas son tan complejas como flexibles. Sin embargo, esta naturaleza genérica no está libre de inconvenientes. En concreto, no todo es tan simple, o tan rápido, como debería ser. ¿Qué ocurriría si, en vez de tener una estructura que trate de encajar con todo, utilizamos estructuras más especializadas?. Puede que siga habiendo cosas que no podamos hacer (o, al menos, no podamos hacer bien) pero, seguramente ¿no podremos haber ganado en simplicidad y velocidad?

¿Usar estructuras de datos específicas para resolver problemas específicos?, ¿no es así como programamos?. No utilizas una tabla hash para cada dato, al igual que tampoco usas una variable escalar. Para mí, esto es lo que define la aproximación de Redis. Si estás trabajando con escalares, listas, tablas hash o conjuntos, ¿por qué no almacenarlos en escalares, listas, tablas hash o conjuntos?, ¿por qué comprobar la existencia de un elemento tiene que ser más complicado que invocar `exists(key)` o más lento que O(1)? (búsqueda que no se ralentiza sin importar el número de elementos que haya).

Los Ladrillos

Bases de datos

Redis tiene el mismo concepto de base de datos que aquel con el que estés familiarizado. Una base de datos contiene un conjunto de datos. El caso de uso típico de una base de datos es conservar todos los datos de una aplicación juntos y mantenerlos separados de los de otras aplicaciones.

En Redis, las bases de datos están identificadas simplemente con un número, siendo la base de datos por defecto el número 0. Si quieres cambiar a otra base de datos puedes hacerlo a través del comando select. En la interfaz por línea de comandos, escribe select 1. Redis debería responderte con un mensaje de OK y tu prompt debería cambiar por algo como redis 127.0.0.1:6379[1]>. Si quieres volver a la base de datos por defecto, sólo introduce select 0 en la interfaz por línea de comandos.

Comandos, Claves y Valores

Aunque Redis es más que simplemente un almacén de conjuntos clave-valor, en su núcleo, cada una de las cinco estructuras de datos de Redis tiene al menos una clave y un valor. Es indispensable que entendamos los conceptos de clave y valor antes de continuar con el resto de elementos de información.

Las claves son lo que vamos a utilizar para identificar conjuntos de datos. Vamos a tratar mucho con claves, pero por ahora, es suficiente con saber que una clave tiene un aspecto similar a users:leto. Uno podría esperar razonablemente que una clave como esta contuviese información sobre un usuario llamado leto. La coma no tiene ningún significado en especial, pero en lo relativo a Redis, es utilizado como un separador común para organizar las claves.

Los valores representan los datos que se encuentran relacionados con la clave. Pueden ser cualquier cosa. A veces almacenarás cadenas de texto, a veces números enteros, a veces almacenarás objetos serializados (como JSON, XML, o cualquier otro formato). La mayor parte de las veces Redis tratará los valores como arrays de bytes y no se preocupará de su tipo. Hay que tener en cuenta que los distintos controladores gestionan la serialización de modos distintos (algunos te la dejarán a tí) de tal modo que en este libro sólo hablaremos de cadenas de texto, números enteros y objetos JSON.

Vamos a mancharnos un poco las manos. Escribe el siguiente comando:

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

Esta es la anatomía básica de un comando de Redis. En primer lugar tenemos el comando a utilizar, en este caso set. Después tenemos sus parámetros. El comando set recibe dos parámetros: la clave sobre la cual establecemos el valor y el valor que estamos estableciendo. Muchos comandos, aunque no todos, reciben una clave (y cuando lo hacen, es a menudo el primer parámetro). ¿Puedes adivinar cómo recuperar este valor?. Espero que hayas dicho (¡aunque no te preocupes si no estabas seguro!):

```
get users:leto
```

Ve más allá y juega con otras combinaciones. Las claves y los valores son conceptos fundamentales, y los comandos get y set son el camino más sencillo para jugar con ellos. Crea más usuarios, prueba distintos tipos de claves y prueba distintos valores.

Consultas

A medida que vayamos avanzando, dos cosas tienen que ir quedando claras. En lo relativo a Redis, las claves lo son todo y los valores no son nada. O, dicho de otro modo, Redis no te permitirá consultar valores de un objeto. De este modo, no podremos buscar a los usuarios que viven en el planeta dune.

Para muchos ésto puede ser motivo de preocupación. Vivimos en un mundo donde la consulta de datos es tan flexible y poderosa que la aproximación de Redis parece primitiva y poco práctica. No permitas que esto te transtorne demasiado. Recuerda que Redis no es una solución para todo. Habrá cosas que simplemente no encajen (debido a las limitaciones a la hora de realiazr consultas). Sin embargo, considera que en algunos casos encontrarás nuevas maneras de modelar tus datos.

Veremos ejemplos más concretos a medida que avancemos, pero es importante que entendamos la realidad más básica de Redis. Esto nos ayudará a entender por qué los valores pueden no ser nada - Redis nunca necesita leerlos o entenderlos. Además, nos ayuda a preparar nuestras mentes a pensar cómo modelar en este nuevo mundo.

Memoria y Persistencia

Anteriormente mencionamos que Redis es un almacén de persistencia en memoria. Con respecto a la persistencia, Redis va creando fotos de la base de datos que almacena en disco en base a cuántas claves han cambiado. Es posible configurar este comportamiento de tal modo que si X claves han cambiado, entonces se almacene la base de datos cada Y segundos. Por defecto, Redis guarda la base de datos cada 60 segundos si han cambiado 1000 o más claves.

De forma alternativa (o en conjunto con la captura de fotografías), Redis puede funcionar en modo diferencial. Cada vez que una clave cambia, se actualiza un fichero con únicamente el diferencial en disco. En algunos casos es aceptable perder 60 segundos de datos con el objetivo de obtener más rendimiento. En algunos casos esta pérdida no es aceptable. Redis te da la opción. En el capítulo 6 veremos una tercera opción, que consiste en descargar persistencia en un nodo esclavo.

Con respecto a la memoria, Redis mantiene todos los datos en memoria, con lo que esto tiene una implicación obvia: la memoria RAM es a día de hoy la parte más cara del hardware del servidor.

Siento que muchos desarrolladores han perdido la percepción de qué poco espacio ocupan los datos. La obra completa de William Shakespeare ocupa aproximadamente unos 5,5 MB. En términos de escalabilidad, otras soluciones tienen a marcar límites en lo relativo a la CPU o IO. La limitación que necesites (en RAM o IO) dependerá de qué estés almacenando y consultando. A menos que estés almacenando archivos multimedia en Redis, la memoria probablemente no será un problema. Para aplicaciones donde esto sea un problema seguramente necesites otras soluciones,

Redis incluyó soporte para memoria virtual. Sin embargo, esta característica se ha considerado como un error (por los propios desarrolladores de Redis) y su uso ha sido descatalogado.

(Como nota aparte, comentar que los 5,5 MB de la obra completa de Shakespeare puede ser comprimida a unos 2 MB. Redis no hará ninguna autocompresión pero, puesto que trata los valores como bytes, no hay motivo por el cual no puedas comprimir y descomprimir los datos tú mismo.)

Juntándolo Todo

Hemos visto unos cuantos elementos de alto nivel. Lo último que me gustaría hacer antes de bucear en Redis es ver estos elementos juntos. En concreto, las limitaciones en las consultas, las estructuras de datos y la manera que tiene Redis de almacenar la información en memoria.

Cuando pones esas tres cosas juntas ver algo maravilloso: velocidad. Mucha gente piensa "Claro que Redis es rápido, todo está en memoria". Pero eso es sólo una parte. El motivo principal que hace que Redis brille ante otras soluciones son sus estructuras de datos especializadas.

¿Cómo de rápido? Depende de muchas cosas - de qué comandos estés utilizando, de los tipos de los datos, etc. Pero el rendimiento de Redis tiende a ser medido en decenas de miles, o cientos de miles de operaciones **por segundo**. Puedes ejecutar `redis-benchmark` (el cual está en el mismo directorio que `redis-server` y `redis-cli`) para probarlo por tí mismo.

Una vez migré un código que utilizaba un modelo tradicional a Redis. Una prueba de carga que escribí tardó 5 minutos en terminar utilizando el modelo relacional. Tardó unos 150ms en terminar en Redis. Nunca obtendrás este tipo de ganancia tan abultada pero espero que te dé una idea acerca de qué estamos hablando.

Es importante entender este aspecto de Redis ya que tiene impacto en cómo interactuar con él. Los desarrolladores acostumbrados a SQL a menudo trabajan minimizando el número de operaciones que deben realizar con la base de datos. Este es un buen consejo para cualquier sistema, incluido Redis. Sin embargo, debido a que estamos trabajando con unas estructuras de datos muy sencillas, a menudo necesitaremos operar con el servidor de Redis varias veces para cumplir nuestro objetivo. Estos patrones de acceso a datos pueden parecer poco naturales al principio, pero tienen un coste insignificante comparado con el rendimiento que podemos ganar.

En Este Capítulo

A pesar de que apenas hemos llegado a jugar con Redis hemos cubierto un amplio abanico de temas. No te preocupes si algo no está del todo claro - como las consultas. En el siguiente capítulo vamos a ponernos en práctica y posiblemente las dudas se respondan solas.

Lo más importante que nos llevamos de este capítulo es que:

- Las claves son cadenas de texto que identifican bloques de datos (valores).
- Los valores son bloques de bytes que Redis trata indiferentemente.
- Redis expone (y, de hecho, está implementado como) cinco estructuras de datos especializadas.
- Estas características combinadas hacen que Redis sea rápido y fácil de usar, aunque no es válido para todos los escenarios posibles.

Capítulo 2 - Las Estructuras de Datos

Es el momento de echar un vistazo a las cinco estructuras de datos de Redis. Vamos a explicar qué es cada estructura de datos, qué métodos están disponibles y para qué tipo de funcionalidad/datos lo puedes utilizar.

Los únicos elementos que hemos visto hasta ahora son comandos, claves y valores. Hasta ahora, no hemos concretado nada acerca de las estructuras de datos. Cuando empleábamos el comando set, ¿Cómo sabía Redis qué tipo de datos debía utilizar? Cada comando es específico para cada estructura de datos. Por ejemplo, cuando utilizas el comando set estás almacenando el valor en una estructura de cadenas de texto. Cuando utilizas hset lo estás almacenando en una tabla hash. Dado el pequeño tamaño del vocabulario de Redis, es muy manejable.

La web de Redis tiene una magnífica documentación de referencia. No hay motivo por el cual repetir el trabajo que ellos ya han realizado. Nosotros sólo cubriremos los comandos más importantes para poder entender el propósito de la estructura de datos.

No hay nada más importante que divertirse y probar cosas. Siempre podrás dejar la base de datos en su estado original con el comando flushdb, ¡así que no seas tímido y haz locuras!

Cadenas de Texto

Las cadenas de texto son las estructuras de datos más básicas disponibles en Redis. Cuando piensas en un par clave-valor, estás pensando en cadenas de texto. Independientemente al nombre de la clave, el valor puede ser cualquier cosa. Yo prefiero llamarlos "escalares", pero es simplemente una preferencia personal.

Ya hemos visto un caso de uso común de empleo de cadenas de texto, almacenando instancias de objetos por clave. Esto es algo de lo que harás mucho uso:

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

Adicionalmente, Redis te permite realizar algunas operaciones comunes. Por ejemplo, se puede utilizar strlen <key> para calcular la longitud del valor de la clave; getrange <key> <start> <end> devuelve la subcadena de texto en el rango indicado; append <key> <value> añade el valor al final del valor existente (o crea uno si no existe ninguno todavía). Vamos un paso más allá a probar esto. Esto es lo que obtendremos:

```
> strlen users:leto
(integer) 42

> getrange users:leto 27 40
"likes: [spice]"

> append users:leto " OVER 9000!!"
(integer) 54
```

Ahora puede que estés pensando, esto es estupendo, pero no tiene sentido. No tiene sentido mirar en un rango específico de un objeto JSON o añadir un valor. Tienes razón, la lección trata de enseñar que algunos comandos, especialmente al tratar con cadenas de texto, sólo tienen sentido en tipos de datos específicos.

Anteriormente aprendimos que Redis no tiene en cuenta los valores. La mayor parte del tiempo esto es cierto. Sin embargo, unos pocos comandos con cadenas de texto son específicos para algunos tipos o estructuras de valores. Por ejemplo, existen los comandos incr, incrby, decr y decrby, los cuales incrementan o decrementan el valor de una cadena de texto:

```
> 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
```

Como puedes imaginar, las cadenas de texto de Redis son muy buenas para análisis. Prueba a incrementar users:leto (un valor no entero) y verás qué ocurre (deberías ver un error).

Hay un ejemplo más avanzado con los comandos setbit y getbit. Hay una entrada maravillosa de cómo Spool utiliza estos dos comandos para, de una forma efectiva, responder a la pregunta "¿cuántos usuarios únicos tuvimos hoy?". Para 128 millones de usuarios, un portátil convencional devuelve la respuesta en menos de 50ms utilizando sólo 16MB de memoria.

No es importante entender cómo funcionan los mapas de bits, o cómo Spool los utiliza, sino entender que las cadenas de texto de Redis son más potentes de lo que pueden parecer a simple vista. Una vez más, los casos de uso más comunes son los que comentamos anteriormente: el almacenamiento de objetos (sean complejos o no) y los contadores. Además, puesto que recuperar un valor a través de su clave es tan rápido, las cadenas de texto se emplean a menudo para cachear datos.

Hashes

Los hashes son un buen ejemplo de por qué no es acertado decir que Redis es un almacén clave-valor. Como posiblemente sepas, en cierta manera, los hashes son como las cadenas de texto. La diferencia más importante es que los hashes incluyen un nivel extra de indirección: un campo.

En este caso, los equivalentes en hash de set y get son:

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

Podemos dar valor a varios campos a la vez, obtener varios campos a la vez, recuperar todos los campos y sus valores, mostrar todos los campos o borrar un campo específico.

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

```
hkeys users:goku
hdel users:goku age
```

Como puedes ver, los hashes nos dan algo más de control de lo que nos daban las cadenas de texto. En lugar de almacenar un usuario como un valor serializado podemos almacenar un hash con una representación más acertada. El beneficio que se obtiene de ellos es el poder acceder y actualizar/borrar trozos concretos de datos sin tener que recuperar o escribir el valor entero.

Mirando los hashes desde la perspectiva de objetos bien definidos, como usuario, es primordial entender cómo funcionan. Y es cierto que, por motivos de rendimiento, puede ser útil tener un control con un grano más fino. Sin embargo, en el próximo capítulo veremos cómo los datos pueden utilizarse para organizar los datos y lanzar consultas de un modo más práctico. En mi opinión, este es el escenario en el que los hashes realmente brillan.

Listas

Las listas permiten almacenar y manipular un conjunto de valores dada una clave concreta. Puedes añadir valores a la lista, recuperar el primer o el último valor y manipular valores de una posición concreta. Las listas mantienen un orden y son eficientes al realizar operaciones basadas en su índice. Podemos tener una lista llamada newusers en la que guardar los usuarios registrados más recientemente en nuestra web:

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

En primer lugar colocamos un nuevo usuario en el comienzo de la lista, y después la truncamos para que sólo contenga los últimos 50 usuarios. Este es un patrón común. ltrim es una operación con una complejidad O(N), donde N es el número de valores que estamos eliminando. En este caso, en el que estamos truncando tras insertar un único elemento, se obtiene un rendimiento constante de O(1) (porque N es siempre igual a 1).

Esta es además la primera vez que vamos a ver el valor de una clave referenciando el valor de otra. Si queremos recuperar los detalles de los últimos 10 usuarios, podemos hacer la siguiente combinación:

```
keys = redis.lrange('newusers', 0, 10)
redis.mget(*keys.map {|u| "users:#{u}"})
```

Por supuesto, las listas no son buenas únicamente para almacenar referencias a otras claves. Los valores pueden ser cualquier cosa. Puedes usar listas para almacenar trazas de log o registrar el camino que está siguiendo un usuario a través de una aplicación. Si estás construyendo un juego puedes usarlo para registrar las acciones que realiza un usuario.

Conjuntos

Los conjuntos se emplean para almacenar valores únicos y facilitan un número de operaciones útiles para tratar con ellos, como las uniones. Los conjuntos no mantienen orden pero brindan operaciones eficientes basándose en los valores. Una lista de amigos es el ejemplo clásico de uso de un conjunto.

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

Independientemente de cuántos amigos tenga un usuario, podemos decir eficientemente (O(1)) cuándo un el usuarioX es amigo del usuarioY o no.

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

Es más, podemos saber cuándo dos o más personas tienen los mismos amigos:

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

E incluso almacenar el resultado en una nueva clave:

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

Los conjuntos son muy buenos para etiquetar o registrar propiedades de un valor para los cuales los duplicados no tengan sentido (o para aquellos escenarios en los que queramos realizar operaciones como intersecciones o uniones).

Conjuntos Ordenados

La última y más poderosa estructura de datos son los conjuntos ordenados. Si los hashes son como las cadenas de texto pero con campos, entonces los conjuntos ordenados son como los conjuntos pero con una puntuación. La puntuación facilita la ordenación y la capacidad de establecer un ranking. Si queremos tener una lista de amigos con ranking, podemos hacer:

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

¿Cómo encontramos cuántos amigos tiene duncan con una puntuación de 90 o más?

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

¿Cómo podemos saber cuál es el ranking de chani?

```
zrevrank friends:duncan chani
```

Utilizamos zrevrank en lugar de zrank porque la ordenación por defecto de Redis ordena de menor a mayor valor (pero, en este caso, queremos ordenar de mayor a menor valor). El caso de uso más obvio para los conjuntos ordenados son los sistemas de clasificación. En realidad, cualquier cosa que quieras ordenar en base a un valor entero, y que sea capaz de ser manipulable eficientemente en base a su puntuación, puede encajar bien en un conjunto ordenado.

En Este Capítulo

Hemos echado un vistazo general a las cinco estructuras de datos de Redis. Una de las mejores cosas de Redis es que puedes hacer más cosas de las que en un primer momento pensaste. Seguramente haya maneras de usar las

cadenas de texto y los conjuntos que todavía no haya pensado nadie. A medida que vayas entendiendo los casos de uso habituales encontrarás a Redis perfecto para todo tipo de problemas. Además, aunque Redis exponga cinco estructuras de datos y varios métodos, no creas que vas a necesitar utilizar todos ellos. No es raro construir una nueva funcionalidad que utilice únicamente unos pocos comandos.

Capítulo 3 - Aprovechando las Estructuras de Datos

En el capítulo anterior hablamos acerca de las cinco estructuras de datos y dimos algunos ejemplos de los problemas que podian resolver. Es el momento de ver algo un poco más avanzado, aunque común, como son algunos tópicos y los patrones de diseño.

Cota Superior Asintótica

A lo largo de este libro hemos hecho referencias a la cota superior asintótica usando la sintaxis O(n) u O(1). La cota superior asintótica se utiliza para explicar cómo se comporta algo dado un cierto número de elementos. En Redis, se utiliza para decirnos cómo de rápido es un comando o el número de elementos con los que vamos a tratar.

La documentación de Redis indica la cota superior asintótica para cada uno de sus comandos. Esto nos indica qué factores influyen en el rendimiento. Vamos a ver algunos ejemplos.

Lo más rápido que se puede ser es O(1), que es una constante. Independientemente de estemos tratando con 5 elementos o con 5 millones, obtendrás el mismo rendimiento. El comando sismember, que nos indica si un valor pertenece a un conjunto, es O(1). sismember es un comando potente, y sus características en lo relativo al rendimiento son un buen motivo de ello. Varios comandos de Redis son O(1).

La función logarítmica, u O(log(N)), es el siguiente caso más rápido porque necesita buscar a través de particiones cada vez más pequeñas. Usando este tipo y una aproximación del tipo divide y vencerás, un número muy grande de items se descomponen en unas pocas iteraciones. zadd es un comando O(log(N)) donde N es el número de elementos que ya existen en el conjunto.

Aparte existen comandos lineares, u O(n). Buscar dentro una columna sin indexación en una tabla es una operación O(N). Esto es lo que ocurre con el comando ltrim. Sin embargo, en el caso de ltrim, N no es el número de elementos en la lista, sino el número de elementos que van a ser eliminados. Emplear ltrim para eliminar un item de una lista de millones será más rápido que usar ltrim para eliminar 10 items de una lista de cientos. (O eso creo, ya que probablemente ambas operaciones serán tan rápidas que no seremos capaces de cronometrarlas).

zremrangebyscore, que elimina elementos de un conjunto ordenado con una puntuación que se encuentre dentro de unos valores máximo y mínimo, tiene una complejidad de O(log(N)+M). Es una especie de mezcla. Leyendo la documentación se puede observar que N es el número total de elementos en el conjunto y M es el número de elementos que van a ser eliminados. En otras palabras, el número de elementos que no van a ser eliminados es más relevante, en términos de rendimiento, que el número total de elementos del conjunto.

El comando sort, del cual hablaremos en más detalle en el próximo capítulo, tiene una complejidad de O(N+M*log(M)). Probablemente sea uno de los comandos más complejos de Redis debido a sus características de rendimiento.

Hay, además otras complejidades. Dos más comunes que podríamos mencionar son O(N^2) y O(C^N). Cuando más largo sea N, peor será su rendimiento con respecto a una N más pequeña. Ninguno de los comandos de Redis tiene este tipo de complejidad.

Merece la pena señalar que la cota superior asintótica se refiere siempre al peor caso. Cuando decimos que algo tiene O(N), en realidad decimos que podría encontrar el dato de inmediato o encontrarlo en el último elemento posible.

Pseudo Consultas Multi Clave

Una situación con la que tropezarás a menudo será la de tratar de recuperar el mismo valor usando claves diferentes. Por ejemplo, puede que quieras recuperar un usuario en base a su email (para el caso del login) y también por id (una vez se haya logado). Una solución terrible es duplicar el usuario usando dos valores de cadena de texto:

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

Esto es malo porque es una pesadilla de gestionar y utiliza el doble de memoria.

Sería bastante interesante que Redis dejara enlazar una clave con otra, pero no lo hace (y, seguramente, nunca lo hará). Una de las reglas más importantes en el desarrollo de Redis es mantener tanto el código como el API limpios y simples. La implementación interna que supone enlazar claves (hay un montón de cosas que se pueden hacer con claves de las cuales no hemos hablado todavía) no vale la pena cuando Redis ya incluye una solución: los hashes.

Gracias a los hashes podemos suprimir la necesidad de tener duplicados

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

Lo que estamos haciendo es utilizar el campo como un pseudo índice secundario y referenciar al objeto usuario. Para recuperar un usuario por id podemos emplear un get normal:

```
get users:9001
```

Para recuperar a un usuario por email, podemos utilizar un hget seguido por un get (en Ruby):

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

Esto es algo que seguramente acabes haciendo a menudo. Para mí, este es el escenario en el cual los hashes son realmente brillantes, pero no se convierte en un caso de uso obvio hasta que no lo utilizas.

Referencias e Índices

Hemos visto un par de ejemplos donde un valor referencia a otro. Lo vimos cuando hicimos nuestro ejemplo de la lista, y lo vimos también en la sección en la que empleabamos hashes para hacer las consultas un poco más sencillas. Lo que viene ahora es cómo gestionar de forma manual tus índices y referencias entre valores. Siendo sincero, creo que podemos decir que es un poco decepcionante, especialmente cuando tienes que tener en cuelta que debes gestionar, actualizar y borrar estas referencias de forma manual. No hay una solución mágica que resuelva este problema en Redis.

Ya hemos visto cómo los conjuntos son utilizados a menudo para implementar este tipo de índice manual:

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

Cada miembro de este conjunto es una referencia a una cadena de texto de Redis que contiene los detalles del usuario actual. ¿Qué ocurre si chani cambia de nombre, o borra su cuenta?. Quizá tendría sentido tener en cuenta también las relaciones inversas:

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

Costes de mantenimiento aparte, si eres como yo, puede que te estremezcas pensando en el coste extra de procesamiento y memoria que supone tener estos índices extra. En la siguiente sección hablaremos de formas de reducir los costes de rendimiento dando unos rodeos extra (hablamos brevemente acerca de esto en el primer capítulo).

Si piensas en ello, las bases de datos relacionales tienen la misma sobrecarga. Los índices consumen memoria, tienen que ser recorridos constantemente para que los registros estén en su lugar. Esta sobrecarga está pulcramente localizada (y, de hecho, se hacen muchas optimizaciones para conseguir que su procesamiento se haga de un modo lo más eficiente posible).

De nuevo, tener que tratar manualmente con la referencias en Redis es funesto. Pero con los conceptos que ya posees sobre el rendimiento o el uso de la memoria creo que lo percibirás como algo importante.

Rodeos Extra y Tuberías

Ya hemos comentado que hacer peticiones frecuentemente al servidor es un patrón común de Redis. Dado que es algo que vas a hacer a menudo, es importante echarle un vistazo a qué funcionalidades pueden ayudarte a conseguir el mejor resultado.

Para empezar, muchos comandos aceptan uno o más conjuntos de parámetros o, incluso, aceptan varios parámetros. Anteriormente vimos el comando mget, que recibe varias claves y devuelve los valores:

```
keys = redis.lrange('newusers', 0, 10)
redis.mget(*keys.map {|u| "users:#{u}"})
```

O el comando sadd, el cual añade uno o más miembros a un conjunto:

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

Redis, además, permite el uso de tuberías. Por lo general, cuando un cliente envía una petición a Redis espera la respuesta antes de enviar la siguiente petición. Al usar tuberías puedes enviar varias peticiones sin esperar por sus respuestas. Esto reduce la carga de la red y puede dar, como resultado, una ganancia en rendimiento importante.

Es importante tener en cuenta que Redis utilizará la memoria para encolar comandos, así que es buena idea hacer lotes con ellos. El cómo de grande será el lote dependerá de qué comandos estés usando y, más específicamente, de cómo de largos sean los parámetros. Si estás ejecutando comandos de unos 50 caracteres seguramente podrás hacer lotes de miles o decenas de miles de ellos.

En concreto, el cómo ejecutes comandos a través de una tubería variará entre drivers. En Ruby debes pasarle un bloque al método pipelined:

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

¡Como puedes imaginar, las tuberías pueden acelerar la carga de un lote!

Transacciones

Cada comando de Redis es atómico, incluyendo los que hacen varias cosas. Además, Redis tiene soporte para transacciones aún cuando utilices varios comandos.

Puede que no lo sepas, pero Redis en la actualidad se ejecuta en un único hilo de ejecución, lo que garantiza que cada comando sea atómico. Cuando se está ejecutando un comando, ningún otro puede ejecutarse a la vez (hablaremos brevemente sobre escalabilidad en un capítulo posterior). Esto es particualmente útil cuando consideres que algunos comandos hacen varias cosas. Por ejemplo:

incr es básicamente un get seguido de un set.

getset establece un nuevo valor y devuelve el original.

setnx primero comprueba que la clave exista, y sólo le pone un valor si no existe.

Aunque estos comandos son útiles, inevitablemente necesitarás ejecutar varios comandos en un grupo que se ejecute de forma atómica. Puedes conseguirlo incluyendo antes el comando multi, seguido de todos los comandos que quieras ejecutar dentro de la misma transacción. Finalmente usarás el comando exec para ejecutar los comandos o discard para ignorarlos. ¿Qué garantiza Redis con respecto a la transaccionalidad?

- Los comandos se ejecutarán en orden.
- Los comandos se ejecutarán como una operación atómica única (sin que los comandos de otro cliente se puedan ejecutar entremedias).
- Que los comandos de la transacción se ejecutarán o todos o ninguno.

Puedes, y deberías, probar esto en la interfaz por línea de comandos. Observa que no hay razón por la cual no puedas combinar el uso de tuberías y transacciones.

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

Finalmente, Redis te permite especificar una clave (o claves) que vigilar y lanzar una transaccion si la clave ha cambiado. Esto se utiliza cuando necesitas recuperar valores y ejecutar código basado en esos valores, todo dentro de una misma transacción. Con el código anterior, no seríamos capaces de implementar nuestro propio comando incr ya que todos los comandos son ejecutados juntos cuando se invoca a exec. Programáticamente, no podemos hacer:

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

Esta no es la manera de trabajar de las transacciones de Redis. Pero, si añadimos watch a powerlevel, podemos hacer:

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

Si otro cliente cambiase el valor de powerlevel tras haber invocado nosotros a watch sobre ella, nuestra transacción fallaría. Si ningún cliente cambia el valor, el conjunto funcionará. Podemos ejecutar este código en un bucle hasta que funcione.

Claves y Antipatrones

En el siguiente capítulo vamos a hablar de comandos que no están específicamente relacionados con las estructuras de datos. Algunos de ellos son herramientas de depuración o de administración. Pero hay una en particular de la que quiero hablar: el comando keys. Este comando recibe un patrón y busca todas las claves coincidentes con él. El uso de este comando encaja en algunas tareas, pero nunca debe ser utilizado en entornos de producción. ¿Por qué? Porque hace un escaneo lineal a través de todas las claves para ver si coinciden. O, simplemente, porque es lento.

¿Cómo lo utiliza la gente?. Imagina que estás construyendo un servicio de seguimiento de incidencias. Cada cuenta tiene un id y decides almacenar cada incidencia en una cadena de texto con una clave como la siguiente: `indicencia:id-cuenta:id-incidencia`. Si alguna vez necesitas buscar todas las incidencias de una cuenta (para mostrarlas, o para borrarlas si quieren borrar su cuenta), puedes estar tentado (¡y yo lo estuve!) de usar el comando keys:

```
keys indicencia:1233:*
```

La mejor solución es utilizar un hash. Desde que podemos usar hashes para poder disponer de indices secundarios podemos usarlos también para organizar nuestros datos:

```
hset indicencia:1233 1 "{id:1, cuenta: 1233, asunto: '...'}"
hset indicencia:1233 2 "{id:2, cuenta: 1233, asunto: '...'}"
```

Para recuperar todos los identificadores de las incidencias de una cuenta simplemente podemos invocar `hkeys indicencia:1233`. Para borrar una incidencia específica podemos hacer `hdel indicencia:1233 2` y para borrar una cuenta podemos utilizar `del indicencia:1233`.

En Este Capítulo

Este capítulo, combinado con el anterior, nos ha permitido ver cómo podemos utilizar las poderosas características de Redis. Hay una cantidad de patrones que puedes utilizar para construir todo tipo de cosas, pero la clave real es

entender las estructuras de datos fundamenteles y tener la percepción de cómo pueden usarse para alcanzar hitos que se encontraban tras nuestra perspectiva inicial.

Capítulo 4 - Más allá de las Estructuras de Datos

Mientras que las cinco estructuras de datos están en lo más profundo de Redis, hay otros comandos que no son específicos de ninguna estructura de datos. Ya hemos visto un conjunto útil de ellas: info, select, flushdb, multi, exec, discard, watch y keys. En este capítulo veremos otras igualmente importantes.

Caducidad

Redis permite incluir caducidad a una clave. Puedes darle una fecha absoluta indicando el tiempo absoluto (segundos desde el 1 de enero de 1970) o especificar un tiempo de vida en segundos. Este comando se basa únicamente en la clave, con lo que no importa el tipo de datos que represente.

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

El primer comando borrará la clave (y el valor asociado) tras 30 segundos. El segundo hará lo mismo el 31 de diciembre de 2012 a las 12:00 a.m.

Esto permite a Redis ser un motor de caché ideal. Puedes saber cuánto tiempo va a vivir un dato a través del comando ttl y puedes eliminar su caducidad a través del comando persist:

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

Finalmente hay un comando especial para cadenas de texto, setex, que permite definir una cadena de texto e indicar el tiempo que va a vivir en un comando atómico (esto es más por conveniencia que por otra cosa):

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

Publicación y Subscripciones

Las listas de Redis tienen un comando blpop y brpop que devuelve y elimina el primer (o último) elemento de la lista o se bloquea hasta que haya uno disponible. Es posible usarlo para mejorar una consulta simple.

Tras esto, Redis tiene soporte de primer nivel para publicar mensajes y subscribirse a canales. Puedes intentar esto abriendo una segunda ventana de redis-cli. En la primera ventana nos subscribimos a un canal (al que llamaremos warnings):

```
subscribe warnings
```

La respuesta es la información de tu subscripción. Ahora, en la otra ventana, publica un mensaje en el canal warnings:

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

Si vuelves a mirar en la primera ventana deberías haber recibido el mensaje en el canal warnings.

Puedes subscribirte a varios canales (subscribe channel1 channel2 …), subscribirte a un patrón de canales (psubscribe warnings:*) y usar los comandos unsubscribe y punsubscribe para dejar de escuchar uno o más canales, o un patrón de canales.

Para terminar, observa que el comando publish devuelve el valor 1. Esto indica el número de mensajes que reciben el mensaje.

Monitororización y Logs

El comando monitor te permite conocer el estado de Redis. Es una herramienta de depuración que te permite ver al detalle cómo está interactuando tu aplicación con Redis. En una de tus dos ventanas con redis-cli (si una de ellas todavía está subscrita, puedes o usar el comando unsubscribe o cerrar la ventana y abrir una nueva) invoca al comando monitor. En la otra, ejecuta cualquier tipo de comando (como un get o un set). Deberías ver estos comandos, junto con sus parámetros, en la primera ventana.

Deberias ser cuidadoso antes de ejecutar el monitor en producción, ya que realmente es una herramienta pensada para depuración y desarrollo. Aparte de esto, no hay mucho más que decir. Simplemente es una herramienta realmente útil.

Además de monitor, Redis tiene showlog que actúa como una gran herramienta de profiling. Registra cualquier comando que tarde más de un número indicado de **micro**segundos. En la siguiente sección cubriremos brevemente cómo configurar Redis, de momento puedes configurar que Redis registre todos los comandos introduciendo:

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

Después introduce unos pocos comandos. Podrás recuperar todos los registros, o los más recientes, usando:

```
slowlog get
slowlog get 10
```

Puedes además recuperar el número de elementos que tienes en el log con el comando slowlog len

Por cada comando que introduzcas verás cuatro parámetros:

- Un identificador autoincremental.
- Una marca de tiempo de Unix que indica cuándo se lanzó el comando.
- El tiempo, en microsegundos, que llevó ejecutar el comando.
- El comando en sí mismo y sus parámetros.

Este log se mantiene en memoria, así que ejecutarlo en producción, aún con una carga baja, no debería ser un problema. Por defecto almacena los últimos 1024 registros.

Ordenación

Uno de los comandos más potentes de Redis es sort. Permite ordenar los valores en una lista, un conjunto o un conjunto ordenado (los conjuntos ordenados están ordenados por puntuación, no los miembros que están dentro del conjunto).

De esta forma tan sencilla, nos permiten hacer:

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

Lo cual devolverá los valores ordenados del más bajo al más alto. Aquí hay un ejemplo más avanzado:

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

El comando anterior nos muestra cómo paginar los registros ordenados (a través de limit), cómo devolver los resultados en orden descendente (a través de desc) y cómo ordenarlos lexicográficamente en vez de numéricamente (a través de alpha).

El poder real de sort reside en su habilidad de ordenar basándose en un objeto tomado como referencia. Anteriormente mostramos cómo listas, conjuntos y conjuntos ordenados son utilizados a menudo para referenciar otros objetos de Redis. El comando sort puede dereferenciar esas relaciones y ordenar en base a un valor subyacente. Por ejemplo, imagina que tenemos un gestor de incidencias que permite a los usuarios consultar incidencias. Podemos usar un conjunto para almacenar qué incidencias están siendo vistas:

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

Perfectamente puede tener sentido ordenarlos por id (que es la ordenación que funcionará por defecto), pero nosotros queremos además ordenarlos por gravedad. Para lograr esto, le diremos a Redis de qué manera queremos ordenar. Antes de nada, vamos a añadir algunos datos de tal modo que podamos obtener un resultado con sentido:

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

Para ordenar estas incidencias por gravedad, de mayor a menor, haremos:

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

Redis sustituirá el * de nuestra forma de ordenar (identificado por el by) por los valores de nuestra lista/conjunto/conjunto ordenado. Esto creará los nombres de las claves que Redis utilizará para ordenar.

Aunque tengas millones de claves en Redis, creo que lo anterior puede quedar un poco desordenado. Por suerte sort puede trabajar con hashes y sus campos. En lugar de tener un grupo de claves de alto nivel puedes utilizar 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, ....}"
```

No sólo está todo mejor organizado, sino que podemos ordenar por severity o priority y además podemos indicarle a sort qué campo queremos recuperar:

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

Se produce la misma sustitución de valores. Redis reconoce la secuencia -> y la utiliza para buscar el campo indicado en nuestro hash. Hemos añadido además el parámetro get para recuperar los detalles de la incidencia.

En conjuntos grandes, sort puede ser lento. La buena noticia es que se puede almacenar la salida de sort:

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

Combinar las capacidades de store y sort con la expiración de los comandos que vimos puede hacer una buena combinación.

En Este Capítulo

En este capítulo nos hemos centrado en comandos que no son específicos para ninguna estructura de datos. Como todo lo demás, su uso depende de las circuntancias. No es infrecuente construir una aplicación o funcionalidad que no necesite hacer uso de la caducidad, publicación/subscripción y/o ordenación. Pero es bueno saber que están ahí. Además, hemos visto algunos otros comandos. Hay más, y una vez que hayas digerido el material de este libro merecerá la pena que te des una vuelta por la lista completa.

Capítulo 5 - Scripts con Lua

Redis 2.6 incluye un intérprete de Lua que los desarrolladores pueden utilizar para escribir consultas más avanzadas que sean ejecutadas en Redis. No sería erróneo que pensases que esta funcionalidad es similar a la de los procedimientos almacenados de la mayoría de bases de datos relacionales.

El aspecto más dificil de dominar de esta funcionalidad es aprender Lua. Por suerte, Lua es similar a la mayoría de los lenaguajes de propósito general, está bien documentado, tiene una comunidad activa y es útil conocerlo más allá de su uso en Redis. Este capítulo no cubre Lua en detalle; pero veremos unos pocos ejemplos que espero sirvan de introducción sencilla.

¿Por qué?

Antes de ver cómo usar Lua, debes estar preguntándote por qué ibas a querer usarlo. A muchos desarrolladores no les gustan los procedimientos almacenados tradicionales, ¿es este caso diferente?. La respuesta corta es no. Usado indebidamente, el uso de Lua en Redis puede dar como resultado el que sea difícil probar código, que haya lógica de negocio acoplada con el acceso a datos o incluso que haya lógica duplicada.

Usado correctamente, en cambio, es una funcionalidad que puede simplificar el código e incrementar el rendimiento. Ambos beneficios pueden conseguirse agrupando varios comandos, con una lógica sencilla, en una función autocontenida. El código se hace más sencillo porque cada invocación a Lua se realiza sin interrupciones y esto consigue una forma limpia de crear comandos que se ejecuten de forma atómica (sobretodo al eliminar la necesidad de usar el incómodo comando watch). Puede incrementar el rendimiento eliminando la necesidad de tener resultados intermedios - la salida final puede ser devuelta con el script.

Los ejemplos de las próximas secciones ilustrarán mejor estos conceptos.

Eval

El comando eval recive un script de Lua (como cadena de texto), las claves con las que tiene que trabajar, y un conjunto opcional de argumentos arbitrarios. Vamos a echar un vistazo a un ejemplo sencillo (ejecutado desde Ruby, ya que ejecutar comandos multi-línea desde la línea de comandos de Redis no es divertido):

```
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'])
```

El código anterior recupera los detalles de todos los amigos masculinos de Leto. Observad que para invocar los comandos de Redis desde nuestro script hemos usado el método redis.call("command", ARG1, ARG2, ...).

Si eres nuevo en Lua deberías revisar cada línea con cuidado. Puede que te resulte útil saber que {} crea una table vacía (que puede actuar como array o como diccionario), #TABLE recupera el número de elementos que hay en TABLE, y .. se utiliza para concatenar cadenas de texto.

eval recibe cuatro parámetros. El segundo parámetro debería ser el número de claves; sin embargo el driver de Ruby automáticamete lo crea por nosotros. ¿Por qué es necesario? Observa cómo se ve lo anterior cuando se ejecuta desde la línea de comandos:

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

En el primer e incorrecto caso, ¿cómo sabe Redis cuáles de los parámetros son claves y cuáles simplemente son argumentos?. En el segundo caso no hay ambigüedad.

Esto nos genera una segunda pregunta: ¿por qué hay que indicar explícitamente las claves? Cada comando de Redis conoce, en tiempo de ejecución, qué claves va a necesitar. Esto permitirá a futuras herramientas, como Redis Cluster, distribuir peticiones entre varios servidores de Redes. Es posible que vayas visto que nuestro ejemplo anterior recupere las claves dinámicamente (sin tener que pasárselas a eval). Un hget es ejecutado sobre todos los amigos masculinos de Leto. Esto es así porque la necesidad de listar las claves antes de tiempo es más una sugerencia que una regla a seguir. El código anterior funcionará en una única instancia, incluso aunque esta tenga replicación, pero no funcionará en el no-todavía-liberado Redis Cluster.

Gestión de Scripts

Aunque los scripts son cacheados por Redis cuando se ejecutan a través de eval, no es buena idea enviar el cuerpo del script cada vez que lo quieres ejecutar. En su lugar, puedes registrar el script en Redis y ejecutarlo por su clave. Para lograr esto puedes usar el comando script load, el cual devuelve el SHA1 del script:

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

Una vez que hemos cargado el script podemos utilizar evalsha para ejecutarlo.

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

script kill, script flush y script exists son el resto de comandos que puedes utilizar para gestionar scripts en Lua. Se utilizan para finalizar un script en ejecución, eliminar todos los scripts de la caché interna y comprobar si un script determinado existe en la caché.

Librerías

La implementación de Lua de Redis contiene un conjunto de librerías útiles. Mientras que table.lib, string.lib y math.lib son muy prácticas, para mí cjson.lib es la que más merece señalar. En primer lugar, si te encuentras en la situación de tener que pasar varios argumentos a un script, verás que resulta mucho más claro pasar un objeto JSON:

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

El cual puedes deserializar con el script de Lua de este modo:

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

Por supuesto que la librería de JSON puede utilizarse para analizar valores almacenados en la propia Redis. Nuestro ejemplo anterior puede ser potencialmente reescrito de este modo:

```
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
```

En lugar de recuperar el género de un campo específico del hash, podemos recuperarlo de los datos almacenados del amigo. (Esta solución es mucho más lenta, y personalmente prefiero la original, pero esto muestra que es posible).

Atomicidad

Ya que Redis tiene un único hilo de ejecución, no tienes que preocuparte de que tu script de Lua pueda interrumpido por otro comando. Uno de los beneficios más obvios de esto es que las claves que tengan un TTL definido no caducarán en mitad de la ejecución. Si una clave está presente al arrancar el script, estará presente en cualquier punto de la ejecución, a no ser que lo borres.

Administración

En el siguiente capítulo vamos a hablar sobre la administración de Redis y su configuración en más detalle. De momento, simplemente basta con conocer que lua-time-limit define cuánto tiempo puede estar ejecutándose un script en Lua antes de terminar. El valor por defecto son unos generosos 5 segundos. Considera reducirlos.

En Este Capítulo

Este capítulo ha introducido las capacidades de scripting de Redis con respecto a Lua. Como con todo, no hay que abusar de esta funcionalidad. Sin embargo, usarlo con prudencia para implementar tus propios comandos personalizados no sólo simplificará tu código sino que te permitirá mejorar el rendimiento. Hacer scripts con Lua es como cualquier otro comando o funcionalidad de Redis: harás un uso limitado de ello al principio y verás que lo acabarás usando más y más cada día.

Capítulo 6 - Administración

Nuestro último capítulo está dedicado a algunos aspectos administrativos relacionados con la ejecución de Redis. De ninguna manera es una guía exhaustiva de administración de Redis. A lo más que llegaremos será a responder algunas preguntas básicas que seguramente tengan los nuevos usuarios que lleguen a Redis.

Configuración

Cuando ejecutas el servidor de Redis por primera vez aparece un mensaje de advertencia indicando que el fichero redis.conf no se puede encontrar. Este fichero puede utilizarse para configurar varias cosas de Redis. Un fichero redis.conf autodocumentado está disponible en cada versión de Redis. El fichero de ejemplo contiene los valores de configuración establecidos por defecto, así que es útil entender tanto cuáles son estas variables de configuración como cuáles son sus valores por defecto. Puedes encontrarlo en https://github.com/antirez/redis/raw/2.4.6/redis.conf.

Este es el fichero de configuración de Redis 2.4.6. Debes reemplazar "2.4.6" en la URL mencionada anteriormente por tu versión. Puedes encontrar tu versión ejecutando el comando info y buscando el primer valor.

No vamos a explicarlo puesto que está autodocumentado.

Aparte de configurar Redis a través del fichero redis.conf, el comando config set se puede utilizar para establecer un valor individual. De hecho, ya lo hemos utilizado para establecer la propiedad slowlog-log-slower-than a 0.

También existe el comando config get, el cual muestra el valor de una configuración concreta. Este comando permite el uso de máscaras. De esta manera, si queremos mostrar todo lo relacionado con el guardado de trazas de log, podemos hacer:

```
config get *log*
```

Autenticación

Se puede configurar Redis para que pida una contraseña. Esto se consigue a través de la variable de configuración requirepass (establecida a través del fichero redis.conf o a través del comando config set). Cuando se indica un valor en requirepass (que será la contraseña que se deba utilizar), los clientes necesitarán ejecutar el comando auth password.

Una vez que un cliente se ha autenticado pueden lanzar cualquier comando contra la base de datos. Esto incluye al comando flushall, que borrará cualquier clave de la base de datos. A través de la configuración, puedes renombrar comandos para alcanzar algo de seguridad a través de la ofuscación:

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

O puedes deshabilitar un comando estableciendo el nombre como una cadena de texto vacía.

Limitaciones de Tamaño

Cuando comienzas con Redis puede que te preguntes "¿cuántas claves puedo tener?". Puede que también te preguntes cuántos campos puede tener un hash (especialmente cuando quieres usarlo para organizar tus datos), o cuántos elementos pueden tener las listas y los conjuntos. Por cada instancia, las limitaciones para todos los elementos mencionados anteriormente es de miles de millones.

Replicación

Redis tiene soporte para replicación, en el sentido de que cuando escribes en una instancia de Redis (el maestro), una o más instancias (los esclavos) se mantienen actualizados con respecto al maestro. Para configurar un esclavo puedes usar o la variable de configuración slaveof o el comando slaveof (las instancias que se ejecuten sin esta configuración son o pueden ser maestros).

La replicación te ayuda a proteger tus datos copiándolos a distintos servidores. La replicación puede utilizarse, además, para mejorar el rendimiento ya que las lecturas pueden ser realizadas por los esclavos. Pueden responder con datos ligeramente desactualizados, pero para la mayoría de las aplicaciones merece la pena.

Desafortunadamente, la replicación de Redis no tiene recuperación automática ante caídas. Si el maestro cae, un esclavo necesita ser promocionado manualmente. Necesitarás de herramientas que provean alta disponibilidad y de scripts que automaticen este cambio para conseguir cierto grado de alta disponibilidad con Redis.

Copias de Seguridad

Hacer una copia de Redis es simplemente una cuestión de copiar una fotografía de Redis en el medio que quieras (S3, FTP, . . .). Redis por defecto guarda sus fotografías en un fichero llamado dump.rdb. En este punto, puedes simplemente hacer un scp, ftp o cp de este fichero.

No es infrecuente deshabilitar tanto la realización de fotografías como la generación de incrementales del maestro y dejar al esclavo que se encargue de ello. Esto ayuda a reducir la carga del maestro y te permite ser más agresivo en el esclavo sin dañar la respuesta del sistema en su conjunto.

Escalando y Redis Cluster

La replicación es la primera herramienta de la cual un sitio que va a crecer se puede aprovechar. Algunos comandos son computacionalmente más caros que otros (por ejemplo sort) y la descarga de su ejecución en un esclavo puede mantener la respuesta del sistema intacta para permitir que entren nuevas consultas.

Detrás de esto, los sistemas realmente escalables distribuyen sus claves a través de varias instancias (las cuales pueden estar funcionando en la misma máquina, recuerda, Redis tiene un único hilo de ejecución). Por el momento, esto es algo de lo que tendrás que preocuparte (aunque varios drivers de Redis facilitan algoritmos de hash consistentes). Pensar en tus datos en términos de distribución horizontal no es algo que podamos cubrir en este libro. Es algo de lo que probablemente no vayas a preocuparte por un tiempo, pero es algo de lo que tendrás que ocuparte más adelante.

La buena noticia es que se está trabajando en Redis Cluster. Esta funcionalidad ofrecerá no sólo escalabilidad horizontal o rebalanceo, sino que además brindará soporte automático ante caídas para dar alta disponibilidad.

La alta disponibilidad y la escalabilidad es algo a lo que se puede llegar a día de hoy si dedicas tiempo y esfuerzo en ello. Mirando al futuro, el cluster de Redis debería hacer las cosas mucho más sencillas.

En Este Capítulo

Dado el número de proyectos y sitios que utilizan Redis, no debería haber dudas en cuanto a que Redis está listo para funcionar en producción, y que lo ha estado todo este tiempo. Sin embargo, algunas herramientas, especialmente aquellas que giran en torno a la seguridad y la alta disponibilidad son todavía jóvenes. Redis Cluster, que esperemos está listo pronto, debería ayudar a alcanzar algunas metas en la gestión.

Conclusión

De varias maneras, Redis representa una simplificación de la forma en la que trabajamos con datos. Se desprende en gran parte de la complejidad y la abstración de otros sistemas. En algunos casos esto hace que Redis sea una mala elección. En otros puedes pensar que Redis fue construido justamente para almacenar tus datos.

En definitiva, se puede decir algo que dije al comienzo: es fácil aprender Redis. Hay muchas nuevas tecnologías y puede ser difícil averiguar en cuáles merece la pena invertir tiempo para aprenderlas. Cuando percibas los beneficios que Redis puede ofrecer con su simplicidad verás que es una de las mejores inversiones, en cuanto a aprendizaje, que tanto tú como tu equipo podeis hacer.