The Little Redis Book

by Karl Seguin

traduzione di
Sandro Conforto

Il Libro

Licenza

Little Redis Book è rilasciato sotto licenza *Attribution-NonCommercial 3.0 Unported*. Non dovresti aver pagato per questo libro.

Sei libero di copiare, distribuire, modificare o mostrare questo libro. Comunque, è richiesto di attribuire sempre il libro all'autore Karl Seguin, e al traduttore Sandro Conforto e di non utilizzare il libro per scopi commerciali.

Il *testo completo* della **licenza** è disponibile su: `http://creativecommons.org/licenses/by-nc/3.0/legalcode`

L'Autore

Karl Seguin è uno sviluppatore con esperienza in vari campi e tecnologie. Contribuisce attivamente a progetti Open-Source, è un technical writer e a volte uno speaker. Ha scritto vari articoli e alcuni strumenti, riguardanti Redis. Redis fornisce classifiche e le statistiche per un suo servizio dedicato a sviluppatori occasionali di giochi: mogade.com.

Karl ha scritto inoltre The Little MongoDB Book, un libro gratuito e popolare riguardante MongoDB.

Il suo blog si trova a `http://openmymind.net` e i suoi tweets su @karlseguin

Il Traduttore

Sandro Conforto si occupa professionalmente dello sviluppo di applicazioni iPad in ambito finanziario. Si dedica inoltre ad Android e a J2EE. E' da sempre appassionato di Open Source, Linux, e di linguaggi dinamici. Vede lo sviluppo software prima di tutto come un`attività creativa. Quando non si dedica all' informatica ama andare in snow board e suonare la sua chitarra classica ed elettrica.

Ringraziamenti

Il traduttore ringrazia di cuore Salvatore Sanfilippo, autore di Redis, per l'incoraggiamento nella stesura, e per aver dato la sua validazione alla bozza finale. Grazie Salvatore!

L'autore ringrazia in special modo Perry Neal per aver *prestato* i suoi occhi, la sua mente, e la sua passione contribuendo alla riuscita di questo libro.

Ultima Versione

L'ultima versione di questo libro è disponibile sul repository *github*:
http://github.com/sandroconforto/the-little-redis-book

E' ben accetto, per ogni contribuzione al libro, il fork e il *pull request* del *repository*.

Introduzione

Durante gli ultimi due anni, le tecniche e gli strumenti usati per la persistenza e la ricerca dei dati hanno avuto una crescita incredibile. Anche se, per il momento, non saranno sicuramente abbandonati i databases relazionali, possiamo però affermare che il panorama attorno alle basi di dati è cambiato radicalmente e non rimarrà più lo stesso.

Tra tutti i nuovi strumenti, a mio parere, Redis si è attestato come il più interessante. Come mai? Prima di tutto dato che è incredibilmente facile da imparare: qualche ora con Redis è sufficiente per cominciare a trovarsi a proprio agio. E poi, risolve un insieme di problemi specifico, ma rimanendo al contempo abbastanza generico. Cioè? cosa si intende esattamente con questo? Redis non si propone di permettere ogni tipo di operazione con ogni tipo di dato: fornisce invece i tipi dato base cui sono abituati tutti gli sviluppatori, e permette in modo semplice di combinarli in modo più complesso. Dopo un breve apprendimento, diventerà evidente a cosa sia adatto e a cosa meno. E non appena lo sarà, dal punto di vista dello sviluppo, sarà una gran cosa.

Anche se è possibile creare un sistema completo usando unicamente Redis, penso che la maggior parte delle persone lo troverà più un complemento alla propria soluzione di persistenza - sia essa fornita da un tradizionale database relazionale, un database documentale, o qualche altra cosa ancora. Un complemento che di solito permette di fornire delle specifiche funzionalità. A tal proposito risulta abbastanza simile a un sistema di indicizzazione: non si baserebbe mai la propria applicazione unicamente su Lucene, ma per una ricerca efficiente, Lucene fornisce un esperienza d'uso veramente gratificante, sia per l'utente che per l'amministratore. D'altronde le similitudini tra un sistema di indicizzazione e Redis terminano qui.

Il fine di questo libro è porre le basi per padroneggiare Redis: ci si concentrerà nell'imparare le cinque strutture dati fondamentali e nel conoscere i vari approcci alla modellazione dati. Si introdurranno poi alcuni dettagli amministrativi e infine alcune tecniche di debugging.

Come Iniziare

Ognuno di noi ha un modo di imparare differente: alcuni preferiscono "sporcarsi subito le mani", altri preferiscono guardare dei video, altri ancora prediligono leggere. Niente aiuterà di più la comprensione di Redis che non provarlo. E' facile da installare e di base è fornito di una shell per cominciare subito a sperimentare. Un paio di minuti e sarà già pronto e funzionante sulla propria macchina.

Su Windows

Redis non supporta ufficialmente Windows, ma c'è comunque questa possibilità: anche se non userei questa versione in produzione, non vi sono limitazioni note per fare delle prove di sviluppo.

E' possibile scaricare da `https://github.com/dmajkic/redis/downloads` la versione più aggiornata (in cima alla lista).

Estrarre lo zip file e accedere, a seconda dell'architettura, alla cartella `64bit` o `32bit`.

Su *nix e MacOSX

Per gli utenti *nix e Mac, compilare i sorgenti dovrebbe essere la migliore opzione. Le istruzioni, assieme all'ultima versione, sono disponibili su `http://redis.io/download`. Al momento l'ultima versione è la 2.6.4; per installare tale versione eseguire da terminale:

```
wget http://redis.googlecode.com/files/redis-2.4.6.tar.gz
tar xzf redis-2.4.6.tar.gz
cd redis-2.4.6
make
```

(Alternativamente Redis si può trovare attraverso vari package managers. Per esempio, per gli utenti Mac che abbiano Homebrew installato potrà fare al caso un semplice `brew install redis`.)

Se si sono compilati i sorgenti, gli eseguibili si troveranno entro la cartella `src`. Entrare in `src` (`cd src`).

Esecuzione e Connessione a Redis

Se è andato tutto a buon fine, gli eseguibili dovrebbero essere nella cartella corrente. Ci si concentrerà dapprima su Redis server e sull'interfaccia a linea di comando. Per far partire il server sotto Windows doppio click su `redis-server`, sotto *nix/MacOSX eseguire `./redis-server` da terminale.

Nel messaggio di avvio Redis avvisa la mancanza del file `redis.conf`. Redis userà in questo caso delle impostazioni di default, perfette per i nostri scopi.

Si esegua quindi l'interfaccia a linea di comando facendo doppio click su `redis-cli` (Windows) o eseguendo `redis-cli` (*nix/MacOSX). Così facendo si effettuerà una connessione al server locale sulla porta di default (6379).

Si può testare che tutto funzioni impartendo il comando `info`. Dovrebbero essere così mostrate una serie di coppie chiave-valore che danno a colpo d'occhio le caratteristiche e lo stato del server.

In caso di problemi è possibile cercare aiuto nel gruppo di supporto ufficiale di Redis.

I Drivers per Linguaggio

Come si scoprirà presto, l`approccio di programmazione in Redis è di tipo procedurale. In questo modo, sia che si usi l'interfaccia a linea di comando, sia che si usi un driver per il proprio linguaggio favorito, non ci saranno molte differenze. Non ci saranno problemi, quindi, nel seguire questa guida adottando un qualsiasi linguaggio di programmazione. E' possibile reperire i vari drivers a partire dalla pagina dei clients.

Il Driver Ruby

Dato che saranno presenti in questa guida degli esempi nel linguaggio Ruby, si riporta brevemente la procedura per usare Redis con esso. Dopo aver installato il linguaggio (www.ruby-lang.org/it/), si eseguano i seguenti comandi:

```
gem install redis
gem install SystemTimer
```

Si entri nell'ambiente interattivo (`irb`) e si digitino le seguenti istruzioni:

```
require 'rubygems'
require 'redis'
redis = Redis.new
redis.ping
```

se il server risponde con la stringa `PONG` tutto è a posto. In caso contrario, si può fare riferimento alla pagina del client.

Capitolo 1 - Fondamenti

Cosa distingue e rende speciale Redis? Che tipo di problemi risolve? Di cosa si dovrebbero preoccupare gli sviluppatori nell'usarlo? Prima di rispondere a tali domande, si dovrebbe aver ben chiaro cosa Redis sia.

Si sente spesso dire che Redis sia un datastore chiave-valore persistito, ma non è propriamente così: anche se Redis mantiene i dati in memoria e li persiste su disco, d`altra parte è molto più di un datastore chiave-valore. E' importante abbandonare questo pregiudizio in quanto, in caso contrario, la propria prospettiva riguardo a Redis, e ai problemi che può risolvere, risulterà troppo ristretta.

In effetti Redis espone cinque differenti strutture dati, delle quali solo una è una tipica struttura chiave-valore. Capire queste cinque strutture, come funzionino, quali metodi espongano e cosa si possa modellare con esse è la chiave per capire Redis. Per prima cosa, vediamo come siano queste strutture dati.

Se dovessimo applicare il concetto di struttura dati al mondo relazionale, potremmo dire che i database espongono una e unica struttura dati: la tabella. Le tabelle in tal modo devono gestire sia casi complessi che semplice quindi essere flessibili. Non esiste molto che non possa essere modellato con esse. D'altra parte, la loro struttura, così generica, può anche dar luogo a problematiche. In particolare, niente è così semplice o veloce, come potrebbe essere. Se invece avessimo a disposizione, invece di una unica struttura per tutto, di strutture dati più specifiche? Ci potrebbero essere delle cose che non si possono più fare (o almeno, non altrettanto bene), ma sicuramente si guadagnerebbe in semplicità e velocità!

Usare strutture dati specifiche per specifici problemi? Ma non è esattamente quello che facciamo quando programmiamo? A mio parere, questo è proprio l`approccio di Redis. Se ci si trova ad operare con scalari, liste, 'hashes', insiemi, perché non persisterle appunto come scalari, liste, `hashes' e insiemi? Ancora, perché verificare la presenza di una chiave deve essere più complesso di un `exists(key)` o più lento di O(1) (cioè un tempo constante, indipendente da quanti elementi sono presenti)?

Basi di Dati

Redis si fonda sugli stessi concetti di un database: persiste un insieme di dati, mantiene inoltre coerenti i dati di un'applicazione tenendoli separati da quelli di una seconda.

In Redis, i database sono identificati da un numero, quello di default è il numero `0`. Per selezionare un database differente è sufficiente usare il comando `select` seguito dal numero identificativo: ad esempio digitare `select 1`. Redis dovrebbe restituire il messaggio `OK` e il prompt dovrebbe diventare qualcosa del tipo `redis 127.0.0.1:6379[1]>`. Per tornare al database di default è sufficiente digitare `select 0`.

Comandi, Chiavi e Valori

Anche se Redis è più di un datastore chiave-valore, di base, ognuna delle cinque strutture dati Redis è costituita da almeno una chiave e un valore. E' quindi fondamentale capire il modello chiave-valore prima di proseguire.

Le chiavi identificano porzioni di dati. Nel seguito avremo parecchio a che fare con le chiavi ma, per il momento, è sufficiente sapere che una chiave può essere ad esempio `users:leto`. Se si usa una chiave siffatta, ci si può ragionevolmente attendere che tale chiave si riferisca alle informazioni di un utente (user) chiamato `leto`. I due punti non hanno uno speciale significato, ma usare un separatore per organizzare le chiavi è un ottimo modo per strutturare i propri dati.

I valori rappresentano invece il dato effettivo associato alla chiave. Potrebbero assumere qualsiasi formato: a volte stringhe, altre volte interi, altre volte ancora oggetti serializzati (in JSON, XML o qualche altro formato ancora). Per lo più i valori vengono considerati come array di byte, non importando la loro rappresentazione effettiva. Si noti comunque che i vari drivers trattano la serializzazione in maniera differente tra loro (a volte delegandola direttamente all'utente) per cui in questo libro tratteremo unicamente le stringhe, gli interi e il formato JSON.

Sporchiamoci, come si dice, un po' le mani sulla tastiera e impartiamo il seguente comando:

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

Questo esempio presenta il tipico formato di un comando Redis: per primo abbiamo l'effettivo comando (`set`) Sono quindi presenti i parametri. Il comando `set` possiede due parametri: la chiave che stiamo impostando (`users:leto`) e il valore che stiamo assegnando alla chiave stessa (`"{name: leto, planet: dune, likes: [spice]}"`). Molti comandi, anche se non tutti, accettano come parametro una chiave (e in questo caso è spesso il primo parametro). Dato che si è usato il comando `set` per impostare un valore, che comando si userà mai per ritrovare tale dato? Probabilmente verrà subito in mente il comando `get`:

```
get users:leto
```

Si proceda e si provi con altre combinazioni. Chiavi e valori sono concetti fondamentali, e i comandi `get` e `set`sono il modo più semplice per manipolarli. Si creino più utenti, si provino altri tipi di chiave e altri tipi di valori.

Interrogazioni

Mentre procederemo, due cose diventeranno chiare. Per quanto riguarda Redis, le chiavi hanno importanza primaria, i valori secondaria. Per esempio, Redis non permette una ricerca

all'interno dell'insieme degli oggetti memorizzati come valori. Posto questo, non è possibile trovare gli utenti che siano abitanti del pianeta `dune`, almeno non direttamente.

Per alcuni, la cosa causerà qualche preoccupazione. Viviamo in un mondo in cui la ricerca dei dati è così flessibile e potente che l'approccio di Redis potrà sembrare primitivo e per niente pragmatico: non ci si lasci traviare da questa prima impressione. Come si è detto Redis non è stato pensato per essere una soluzione a tutti i problemi. Ci saranno delle cose che non sarà adatto a risolvere (in particolare data questa limitazione nella ricerca dei dati). Si tenga comunque presente che, in molti casi, un diverso modo di strutturare e modellare i propri dati farà fronte a dette limitazioni.

Faremo ulteriori esempi ma, per il momento, è importante capire queste scelte di base che hanno guidato lo sviluppo di Redis. Aver capito che Redis si limita a memorizzare, e non a interpretare, i valori ci aiuterà sicuramente a creare e a sviluppare una migliore *forma mentis* per modellare le entità in questo nuovo ambito.

Memoria e Persistenza

Abbiamo accennato che Redis è un data store in memoria persistente. Per quanto riguarda la persistenza, di default, Redis esegue un'immagine su disco del datastore in base al numero di chiavi che sono cambiate. In particolare è possibile configurare in modo che se X è il numero di chiavi che cambia, allora avvenga un salvataggio ogni Y secondi. Di default, Redis salverà il database ogni 60 secondi se 1000 o più chiavi sono cambiate, oppure ogni 15 minuti se sono cambiate nel frattempo 9 chiavi o meno.

In alternativa (o anche in aggiunta al salvataggio periodico), Redis può funzionare in modalità `aggiunta` (*append*). Ogni qualvolta una chiave subisca un cambiamento, un file viene aggiornato su disco. Vale a dire, a volte è accettabile sacrificare, in caso di un guasto hardware o software, 60 secondi di dati, per avere migliori prestazione. In altri casi questa perdita non è accettabile: Redis delega all'utilizzatore la scelta. Nel quinto capitolo verrà presentata una terza possibilità, delegare la persistenza a un sistema secondario (*slave*)

Per quanto riguarda la memoria, Redis mantiene tutti i propri dati in memoria. Un'ovvia implicazione di questo è che i costi di un sistema Redis si spostano verso la RAM, ad oggi ancora la parte più costosa di un server.

Ad ogni modo si tenga presente quanto poco spazio i dati possano occupare: l`opera completa di Shakespeare occupa approssimativamente 5.5MB di spazio. Inoltre, per quanto riguarda la scalabilità, molte soluzioni tendono ad essere limitate dall'*input-output* o dalla potenza di calcolo (CPU). Quale limitazione (RAM o IO) richieda l'aggiunta di più server dipende dal tipo di dati che si deve memorizzare. A meno che non si debbano memorizzare grandi files multimediali con Redis, la limitazione sulla memoria probabilmente non sarà un problema. Per applicazioni in cui ci sia detta limitazione, si dovrebbe comunque valutare un compromesso tra essere limitati dall'IO o essere limitati dalla memoria RAM.

Redis aveva aggiunto un supporto per la memoria virtuale. Sfortunatamente questa soluzione

è stata deprecata in quanto considerata impraticabile dai suoi stessi sviluppatori.

Come nota a margine, i 5.5MB dell'opera completa di Shakespeare possono essere compressi all'incirca a 2MB. Redis non esegue un auto-compressione dei dati, ma dato che considera i dati a livello di bytes, non c'è motivo per cui non possano essere compressi applicativamente, scegliendo un compromesso tra il maggior tempo speso per comprimere/decomprimere e la quantità di RAM occupata.

Comporre il Tutto

Si sono toccati diversi argomenti ad un alto livello. Occorre ora mettere tutte queste cose assieme. In particolare, le limitazioni sulle interrogazioni, le strutture dati e il modo di memorizzare i dati in memoria in Redis.

Quando si mettono queste tre cose assieme si ottiene una cosa fantastica: velocità. Alcune persone potrebbero dire: "Per forza è veloce, tutto è in memoria". Ma questo è solo un aspetto. La vera ragione per cui Redis spicca rispetto ad altre soluzioni sono le sue strutture dati specializzate.

Ma quanto veloce? Dipende da un gran numero di fattori - che comandi si usino, i tipi di dato e così via. Ad ogni modo le prestazioni di Redis si possono misurare in decine o addirittura centinaia di migliaia di operazioni **per secondo**. Per rendersene conto è sufficiente eseguire `redis-benchmark` dalla stessa cartella in cui si trovano `redis-server` e `redis-cli`.

Una volta ho adattato il codice che usava un modello tradizionale per usare Redis. Occorrevano più di cinque minuti per eseguire un test di carico usando il modello relazionale. Per completare lo stesso test con Redis occorrevano solo 150ms! Non è detto che si abbia sempre questo tipo di miglioramento, ma questo esempio può dare un'idea di quello di cui stiamo parlando…

E' importante capire questo aspetto di Redis, perché può cambiare il modo di farne uso. Solitamente provenendo da un solido background SQL si cerca di minimizzare il numero di interazioni col database. Questo rimane comunque una buona pratica anche in Redis. D'altra parte, dato che si ha a che fare con strutture dati più semplici ed efficienti, a volte sarà necessario interagire col server un numero maggiore di volte per lo stesso scopo. Questo potrebbe risultare forzato in un primo momento: si tenga presente però che il costo di queste interazioni tenderà a diventare ininfluente se confrontato con il guadagno globale di prestazioni offerto da Redis.

In Questo Capitolo

Anche se in superficie, si sono toccati una gran varietà di argomenti. Non ci si preoccupi se non risulta ancora tutto chiaro (per esempio le interrogazioni). Nel prossimo capitolo verranno approfonditi e ogni dubbio verrà auspicabilmente fugato.

Le parti più importanti del capitolo sono:

- Le chiavi sono stringhe che identificano porzioni di dati (i valori)

- I valori sono qualsivoglia sequenze di byte che Redis non interpreta
- Redis espone (ed è implementato come) cinque strutture base specializzate
- Assieme, queste cose, fanno di Redis un data store veloce e facile da usare, ma ovviamente non adatto a tutti gli scenari

Capitolo 2 - Le Strutture Dati

E' il momento di vedere in dettaglio le cinque strutture dati costituenti Redis. Si spiegherà in dettaglio cosa ogni struttura sia, che metodi siano disponibili e, infine, che tipo di dati usare con ognuna di esse.

Per esempio quando si è usato il comando `set`, che struttura dato usava Redis? Redis inferisce dal comando usato la struttura dato da utilizzare. Per esempio quando si usa il comando `set` si memorizza il valore come stringa Redis. Usando `hset` si userà un hash. Dato il numero limitato di comandi Redis, il tutto risulta agevole.

Il sito di Redis ha un`ottima documentazione di riferimento (si noti in particolare l'utile categorizzazione nella parte superiore della pagina). E' inutile ripetere l'ottimo lavoro già fatto. Si prenderanno in considerazione unicamente i comandi più importanti in modo da capire la finalità delle varie strutture dati.

Non c'è di meglio che divertirsi e provare le cose: è comunque sempre possibile cancellare tutti gli inserimenti nel proprio database impartendo il comando `flushdb`, per cui si provino senza indugio anche le cose più bizzarre!

Stringhe

Le stringhe Redis (`string`) sono la struttura dati più semplice. Quando si pensa a una coppia chiave-valore, ci si riferisce nel mondo Redis alle stringhe. Ma non ci si confonda per il nome: i propri valori possono rappresentare qualsiasi cosa. Si potrebbe preferire parlare di `scalari`, ma è semplicemente un fatto di gusto personale.

Si è già visto un comune caso d`uso per le stringhe: memorizzare istanze di oggetti in base a una chiave. E' un caso d'uso molto frequente:

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

Inoltre Redis permette alcune operazioni molto comuni. Per esempio `strlen <key>` può essere usato per determinare la lunghezza di un valore associato a una chiave; `getrange <key> <start> <end>` ritorna la sottostringa nell'intervallo specificato del valore associato alla chiave; `append <key> <value>` concatena al valore associato alla chiave, se essa esiste, altrimenti la crea e la inizializza con . Si provino i suddetti comandi:

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

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

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

Si potrà pensare, utile!… ma che senso può avere estrarre un intervallo o concatenare un valore a una sequenza JSON? Effettivamente, il punto qui è che alcuni comandi, specialmente quelli per la struttura dati stringa, non sono completamente generici ma hanno senso unicamente per alcuni tipi di dati.

In precedenza si è detto che Redis non dà importanza ai valori. Questo per la maggior parte è vero, ma, come si è visto, alcuni comandi sono specifici per valori di un certo tipo o struttura. Per esempio, i comandi `append` e `getrange` potrebbero essere utili per una serializzazione dati che occupi poco spazio. Per fare un altro esempio immediato si può ricorrere ai comandi `incr`, `incrby`, `decr` e `decrby`. Questi comandi incrementano e decrementano il valore (intero) di una stringa Redis:

```
> 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
```

Come si può intuire, le stringhe Redis sono ottime per costruire delle statistiche. Si provi, come esercizio, a incrementare `users:leto` (un valore non intero) e si veda cosa succede: si dovrebbe ottenere un errore.

Un esempio avanzato sono i comandi `setbit` e `getbit`. Si veda ad esempio il fantastico post in cui Spool usa in maniera molto efficiente questi due comandi per rispondere al quesito "quanti visitatori unici si sono avuti oggi?". Usando un portatile e inseriti 128 milioni di utenti, la risposta viene generata in meno di 50 ms occupando meno di 16MB di memoria.

Capire come funzionino le mappe di bit, o come Spool le usi, esula dagli scopi di questo libro, ma rimane importante vedere come le stringhe Redis siano più potenti di quanto inizialmente possa sembrare. Ad ogni modo il caso più comune per le stringhe è memorizzare oggetti (complessi o meno) e contatori. Inoltre, dato che ottenere un valore per chiave è così efficiente, sono spesso usate come *cache* dati.

Hashes

Gli `hash` Redis sono un ottimo esempio del fatto che definire Redis un *data-store* chiave valore non sia accurato. Per molti versi gli *hash* sono come le stringhe. La differenza fondamentale

sta nel fatto che forniscono un ulteriore livello di indirezione tramite un campo (*field*). Di conseguenza i corrispondenti di `set` e `get` per un hash diventano:

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

E' possibile impostare/ritornare gruppi di campi in una volta sola, oppure ottenere tutti i campi e i valori, oppure restituire la lista di tutti i campi, oppure ancora cancellare un singolo campo della struttura:

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

Come si può vedere, gli hash forniscono delle possibilità in più rispetto alle semplici stringhe. Invece di salvare i dati di utente come un singolo valore serializzato, si possono in questo modo usare gli hash per avere una rappresentazione più strutturata e accurata. Si ha così il vantaggio di ottenere, aggiornare e cancellare specifiche porzioni di un dato complesso, senza dover estrarre o scrivere l'intero valore. Inoltre, dal punto di vista delle performance, un controllo più granulare è certamente utile.

Per capire come funzionino, è importante vedere gli *hash* come oggetti strutturati. Nel prossimo capitolo, si vedrà come gli *hash* possano essere usati per organizzare i dati e rendere possibili alcuni tipi di interrogazioni altrimenti impossibili: è proprio in questo che gli hash spiccano.

Liste

Le liste Redis (`list`) permettono di memorizzare e manipolare serie omogenee di valori associati a una data chiave. E' possibile aggiungere valori ad una lista, ottenere il primo o l'ultimo elemento e, infine, manipolare valori a un dato indice. Le liste inoltre mantengono l'ordine di inserimento e forniscono efficienti operazioni basate sull'indice. Si potrebbe considerare, ad esempio, la lista degli ultimi utenti registrati al nostro sito:

```
lpush newusers goku
ltrim newusers 0 50
```

Secondo un pattern molto comune, si inserisce un nuovo utente in cima alla lista, quindi si effettua il *trim* della stessa in modo che contenga solo gli ultimi 50 utenti. Si nota che l'operazione `ltrim` ha una complessità O(N) (come evidenziato nell'ottima documentazione), dove N però sono il numero degli elementi da rimuovere (e non gli elementi totali della lista).

Nel nostro caso, dato che si effettua il `trim` dopo un singolo inserimento, si avrà sempre un tempo costante O(1) (N sarà sempre uguale a 1).

Si è visto, nell'uso della lista, un primo esempio in cui un valore associato a una chiave ha un riferimento logico ad un valore associato ad un'altra chiave (`goku` è cioè associato sia alla lista dei `newusers` che alla stringa degli `users`). Data questa relazione logica, se si volessero i dettagli degli ultimi 10 utenti usando il driver Ruby, si scriverebbe:

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

Si noti il fatto che, per ottenere i dati in una relazione, il client deve effettuare un'interazione multipla.

Naturalmente, le liste non sono adatte solo a memorizzare relazioni ad altre chiavi. I valori possono essere qualsiasi cosa. Si possono usare per salvare logs o tenere traccia del percorso di un utente attraverso un sito. Se si stesse scrivendo un gioco, si potrebbero usare per memorizzare una sequenza di azioni.

Insiemi

Gli insiemi Redis (`set`) si possono usare per memorizzare valori univoci. Forniscono inoltre svariate operazioni di tipo insiemistico, ad esempio le unioni. Gli insiemi non sono ordinati ma forniscono comunque operazioni sui valori molto efficienti. Una lista di amicizie può essere un classico esempio d' uso:

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

Indipendentemente da quanti amici ciascuno abbia, è possibile, in maniera molto efficiente (O(1)), dire se l'utente `userX` sia, o no, amico dell'utente `userY`:

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

Inoltre è possibile dire quali amicizie condividano due o più persone:

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

E infine salvare il risultato in una nuova chiave:

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

Insiemi ordinati

L'ultima, e più potente, struttura dati è l'insieme ordinato (`sorted set`). Se gli hash sono analoghi alle stringhe, ma con attributi, gli insiemi ordinati sono analoghi agli insiemi ma con un attributo di ordinamento. Per cui se si volesse considerare una classifica tra le amicizie si potrebbe scrivere:

```
zadd friends:leto 100 ghanima 95 paul 95 chani 75 jessica 1 vladimir
```

Si desisera conoscere quanti amici possiede `leto` con rango maggiore uguale a 90?

```
zcount friends:leto 90 100
```

E il posto in classifica di `chani`?

```
zrevrank friends:leto chani
```

Si noti l`uso di `zrevrank` al posto di `zrank` in quanto l'ordinamento di default è crescente e in questo caso la graduatoria va chiaramente dal più grande al più piccolo. E' evidente l'immediatezza di costruire una classifica in tempo reale tra utenti (`leaderboard system`) facendo uso proprio degli insiemi ordinati. In effetti, però, data l'estrema genericità della struttura, ogni entità che si possa ordinare tramite un qualche intero, e su cui si vogliano applicare in maniera efficiente delle operazioni basate su tale graduatoria, si presta a essere persistita tramite un insieme ordinato Redis.

Nel prossimo capitolo vedremo come gli insiemi ordinati possano essere usati per tenere traccia di eventi temporali (il tempo stabilisce in questo caso una sorta di graduatoria).

In Questo Capitolo

Si è fatta una panoramica ad alto livello delle cinque strutture dati in Redis. Da questa panoramica, il lettore potrà cominciare a pensare a come estendere l'uso di tali strutture oltre i casi fondamentali. In effetti, ci saranno sicuramente modi di usare le stringhe o gli insiemi ordinati che non sono stati ancora pensati. Ad ogni modo, si tenga presente che, anche se Redis espone cinque strutture, sono molti i casi in cui per implementare una caratteristica se ne usa solo un numero ridotto.

Capitolo 3 - Sfruttare Appieno le Strutture Dati

Nel precedente capitolo abbiamo trattato le cinque strutture dato e fornito qualche esempio di problemi che esse possono risolvere. Si può quindi procedere ad argomenti e *design pattern* maggiormente avanzati, ancorché comuni.

La Notazione O-Grande

Già in precedenza ci si è riferiti alla notazione O-grande in particolare a O(N) e a O(1). Tale notazione (limite asintotico superiore) è usata per spiegare come un algoritmo si comporti, rispetto al numero di elementi su cui agisce. Nella documentazione di Redis, assieme alla descrizione di ogni comando, viene usata tale notazione per esprimere con immediatezza quanto veloce sia il comando al variare della numerosità dell'input o anche degli elementi già presenti nella struttura.

I comandi più veloci sono quelli che hanno tempo costante o O(1). Cioè sia che si stiano trattando 5 elementi o 5 milioni di elementi, si avranno le medesime performance. Per esempio il comando `sismember`, che valuta l'appartenenza di un un valore ad un dato insieme, è O(1). Molti comandi Redis sono O(1).

La successiva fascia di performance è quella logaritmica, o O(log(N)). Deriva chiaramente da un approccio di tipo *divide et impera*, per il quale vengono considerate via via partizioni sempre più piccole dell'input. In questo modo anche quantità molto grandi vengono ridotte di molto dopo qualche iterazione. Per esempio il comando `zadd` possiede complessità O(log(N)), con N il numero di elementi già contenuti nell'insieme.

In una fascia successiva vi è la complessità lineare o O(N). Un'interrogazione su una colonna non indicizzata di una tabella è un'operazione di complessità O(N). Lo è anche l'operazione `ltrim`, però in questo caso, N non è il numero di elementi della lista, quanto piuttosto il numero di elementi da rimuovere. Usare `ltrim` per rimuover un elemento da una lista di un milione di elementi sarà quindi più veloce che non rimuovere 10 elementi da una lista formata da migliaia di oggetti. (Anche se probabilmente saranno entrambe così veloci in Redis da non accorgersi comunque della differenza).

`zremrangebyscore`, che rimuove da un insieme ordinato elementi che abbiano punteggio compreso tra un minimo e un massimo, ha invece complessità O(log(N)+M). Si ha cioè un ibrido. Leggendo la documentazione si vede che N è il numero totale di elementi nell'insieme e M è il numero di elementi da rimuovere. Il numero di elementi da rimuovere tenderà, per valori sufficientemente grandi, comunque a fornire un contributo più significativo, in termini di performance, del numero totale degli elementi della lista.

Il comando `sort`, che discuteremo in maniera dettagliata nel prossimo capitolo, possiede una complessità O(N+M*log(M)). Da questo si evince come possa essere un comando molto complesso, in particolare il più complesso in Redis.

Per completezza si aggiungono altri due tipi di complessità: polinomiale (O(N^2)) e esponen-

ziale (O(C^N)). Le performance in questi due casi degradano molto velocemente per valori grandi di N. Nessun comando Redis possiede appunto complessità di questo tipo.

E' opportuno precisare che la notazione O grande tratta il caso peggiore. Quando si dice che un'operazione su una lista lunga N ha complessità lineare (O(N)), potrebbe terminare al primo elemento oppure, nel caso peggiore, potrebbe dover scorrere tutta la lista fino all'ultimo elemento.

Interrogazioni Multi Chiave

Un problema ricorrente è la ricerca di un valore tramite chiavi diverse da quella primaria (l'*id*). Ad esempio si voglia cercare un utente data la sua email, e poi mostrare i suoi dati. Una pessima soluzione è la duplicazione dei dati su due valori:

```
// Da non fare!
set users:leto@dune.gov "{id: 9001, email: 'leto@dune.gov', ...}"
set users:9001 "{id: 9001, email: 'leto@dune.gov', ...}"
```

E' una soluzione pessima perché è difficile mantenere sempre coerenti i due valori e inoltre si spreca il doppio della memoria.

Sarebbe bello se Redis permettesse di collegare una chiave ad un'altra, ma non lo permette (anzi probabilmente non lo farà mai in quanto uno degli scopi di Redis è mantenere l'interfaccia di programmazione snella e semplice). Ciò nonostante Redis fornisce una soluzione basata sugli hash.

Usando un hash, possiamo evitare la duplicazione:

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

Si sta, cioè, usando uno dei campi (email) come indice di secondario, mantenendo in questo modo un singolo oggetto utente. Per ottenere i dati completi a partire dall'id si usa il solito comando `get`:

```
get users:9001
```

Per ottenere i dati utente dall'email, si esegue un `hget` seguito da un `get` nel seguente modo (codice Ruby):

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

Questo *pattern*, non ovvio in un primo momento, finirà per essere usato molto spesso, ed è proprio per queste possibilità di indirezione che gli hash Redis risultano veramente efficaci.

Referenze e Indici

Si sono visti un paio di esempi in cui un valore referenzia un altro. Si è visto in precedenza nell'esempio che coinvolgeva una lista e si è visto nella sezione sopra, nella quale un hash ha semplificato di molto un'interrogazione sui dati utente a partire dall'email. Si vede quindi la necessità di amministrare programmaticamente indici secondari e riferimenti tra valori. Non c'è una soluzione magica a questo problema in Redis: per scelta viene delegato al programmatore questo compito delicato, specialmente quando si dovranno aggiungere/aggiornare/cancellare dei riferimenti.

Si è già visto come gli insiemi siano spesso sfruttati allo scopo di implementare questo tipo di indice:

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

Ogni elemento di questo insieme è effettivamente un riferimento a una stringa Redis contenente i dettagli completi di ogni utente. Ma cosa succede se `chani` dovesse essere rinominato o il suo account dovesse essere cancellato? E se per qualche motivo è necessario tracciare anche la relazione inversa?

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

A parte la difficoltà di mantenere coerente queste strutture, si avrà inoltre un aumento di spazio occupato e di tempo di processore. Nella prossima sezione si vedranno dei metodi per ridurre questi costi aggiuntivi.

Pensandoci un attimo però, i database relazionali, in modo nascosto, hanno comunque lo stesso tipo di *overhead*: gli indici occupano memoria, devono essere scansiti o idealmente interrogati, e infine il record corrispondente verrà restituito. Questo *overhead* è reso trasparente, per il fatto che vengono attuate molte ottimizzazioni, dai database ma è comunque presente.

Ad ogni modo ogni preoccupazione riguardo performance o memoria va testata nel proprio caso. Probabilmente verrà scoperto che il tutto è un falso problema.

Pipelining, Transazioni e Gruppi di Interazioni

Si è detto in precedenza che in Redis può essere necessario effettuare frequenti interazioni col server. A questo riguardo è opportuno introdurre alcune funzionalità offerte che aiuteranno a sfruttare al meglio questo tipo di interazione.

Prima di tutto, molti comandi possiedono un comando *gemello* la cui unica differenza è la possibilità di accettare parametri multipli. Si è visto ad esempio `mget`, che accetta in input una serie di chiavi e ritorna i valori associati a ciascuna di esse:

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

Oppure il comando `sadd` che aggiunge uno o più elementi a un insieme:

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

Redis offre, oltre a tutto questo, la funzionalità di *pipelining*. Normalmente si invia una richiesta al server Redis e si aspetta la risposta prima di poter inviare una richiesta successiva. Attraverso il meccanismo del *pipelining* si possono inviare una serie di interrogazioni senza dover aspettare la risposta di ciascuna. Questo riduce l'*overhead* di rete e può aumentare considerevolmente le performance globali.

A questo proposito è opportuno notare che Redis usa la memoria disponibile per accodare i comandi, per questo è buona norma raggruppare tali sequenze di comandi in porzioni gestibili. Per ottimizzare il tutto è opportuno considerare la grandezza dei parametri dei comandi coinvolti: in caso essi abbiano lunghezza media di circa 50 caratteri, è possibile avere raggruppamenti di migliaia o anche decine di migliaia di istruzioni.

La modalità di esecuzione del *pipelining* varia da driver a driver. Ad esempio, in Ruby, è sufficiente passare un blocco al metodo `pipelined`:

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

Come si potrà immaginare, questa tecnica può velocizzare enormemente un importazione di tipo *batch*.

Transazioni

Ogni comando Redis è atomico, inclusi quelli che eseguono più di un'operazione. In aggiunta a questo, Redis fornisce supporto a transazioni coinvolgenti più comandi.

Redis, in realtà è *single-threaded*, ed è proprio questo che garantisce l'atomicità di ogni singola operazione. Mentre un comando viene eseguito, nessun altro può essere contemporaneamente in esecuzione. (Verrà brevemente trattata la scalabilità di questo modello nel seguito.) Questo aspetto diviene importante non appena si consideri che alcuni comandi sono effettivamente la somma di più comandi, per esempio:

`incr` è essenzialmente un `get` seguita da un `set`

`getset` imposta il nuovo valore (`set`) e poi ritorna il valore originale (`get`)

`setnx` prima controlla se la chiave esiste, e solo se non esiste ne imposta il valore

A parte l'evidente utilità dei comandi multipli, capiterà la necessità di ricorrere a gruppi di comandi da eseguire in modo atomico. Questo è reso possibile impartendo in primo luogo la direttiva `multi`, poi i comandi da eseguire nella transazione, e infine il comando `exec` avvierà effettivamente l'esecuzione dei comandi; alternativamente il comando `discard` effettuerà il *roll-back* di quanto eseguito fino a quel momento. Ma cosa viene garantito da Redis riguardo a una transazione?

- I comandi verranno eseguiti sequenzialmente
- I comandi verranno eseguiti come una singola atomica operazione (un altro *client* non potrà cioè eseguire altri comandi contemporaneamente)
- Che o tutti, o nessun comando, verranno eseguiti al termine della transazione

E' possibile effettuare un test dalla linea di comando:

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

Si noti che non c'è motivo per non combinare *pipelining* e transazioni assieme.

Redis permette di specificare una o più chiavi da tenere in osservazione, per poter applicare la transazione con il valore della chiave modificato. Questa funzionalità viene usata quando si necessita eseguire del codice sulla base di un valore ottenuto durante l'esecuzione di una transazione. In caso contrario non sarebbe possibile reimplementare in modo transazionale il comando `incr`. Non funziona infatti (in Ruby):

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

Alla fine della transazione `powerlevel` continua a valere il valore iniziale. Se invece aggiungiamo un'istruzione `watch` a `powerlevel`:

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

La cosa da sottolineare è che se un altro *client* cambiasse il valore di `powerlevel` dopo aver chiamato il comando `watch` su di esso, la transazione fallirebbe. Si potrebbe eseguire questo codice in un ciclo fino a che non va a buon fine.

Valori Temporali

Un pattern meno comune, ma ugualmente interessante, prevede l'uso di insiemi ordinati per tenere traccia di valori temporali.

Si voglia per esempio tracciare il prezzo di un'azione. Come chiave potrebbe usare il simbolo, il criterio di classificazione potrebbe essere il *timestamp* e il valore il prezzo:

```
redis.zadd('GOOG', Time.now.utc.to_i-100, 625.03)
redis.zadd('GOOG', Time.now.utc.to_i-95, 623.01)
redis.zadd('GOOG', Time.now.utc.to_i-95, 625.02)
redis.zadd('GOOG', Time.now.utc.to_i-92, 624.98)
```

Usando `zrangebyscore` si può ottenere un intervallo di valori basati sul tempo. Se si volessero così sapere i prezzi degli ultimi 5 secondi si potrebbe usare la seguente istruzione:

```
redis.zrangebyscore('GOOG', (Time.now.utc - 5).to_i, Time.now.utc.to_i)
```

Anti-Pattern sulle Chiavi

Nel prossimo capitolo parleremo di comandi non specificamente correlati a strutture dato. Per esempio comandi amministrativi o per il *debugging*. Ma c'è un comando che vale la pena menzionare subito: il comando `keys`. Tale comando va alla ricerca di tutte le chiavi che corrispondono ad un *pattern* fornito in input. In un primo momento, sembrerebbe un comando adatto a fare tutta una serie di compiti, ma in realtà non dovrebbe mai essere usato in ambiente di produzione. Perché? Perché effettua una scansione lineare, attraverso tutte le chiavi, cercando una corrispondenza. Insomma, è lento e potrebbe rallentare completamente il server.

Un caso d'uso tipico potrebbe essere l'implementazione di un servizio di bug tracking. Ogni account avrà un `id` e ogni bug potrebbe essere memorizzato entro una stringa Redis con chiave nella forma `bug:account_id:bug_id`. Se quindi si dovessero mostrare i bugs dell'utente con id `1233` si potrebbe essere tentati di usare il comando `keys` nel seguente modo:

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

Non lo si faccia! L'approccio corretto è quello di usare una struttura hash nel seguente modo:

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

Si noti anche qui un organizzazione dei dati strutturata secondo un indice secondario.

Ora, per ottenere i bugs relativi ad un utente è sufficiente un `hkeys bugs:1233`, che risulterà di gran lunga più efficiente. Per cancellare uno specifico bug relativo a un utente `hdel bugs:1233 2` e quando si cancella un account si possono cancellare i bug ad esso associati con un semplice `del bugs:1233`.

In Questo Capitolo

Questo capitolo, assieme al precedente, avrà auspicabilmente dato qualche indicazione su come risolvere problemi reali usando Redis. Ci sono tanti altri patterns che si possono usare per implementare svariate funzionalità, ma la base di tutto è comprendere le strutture dati fondamentali e vedere come possano essere usate per andare oltre le attese iniziali.

Capitolo 4 - Oltre le Strutture Dati

Oltre ai comandi relativi alle cinque strutture dato fondamentali, ci sono molti altri comandi generici. Se ne sono già visti alcuni: `info`, `select`, `flushdb`, `multi`, `exec`, `discard`, `watch` and `keys`. Questo capitolo si occuperà di altri comandi ugualmente importanti.

Scadenza di una Chiave

Redis permette di assegnare una scadenza a una chiave (*expiration*). Si può impostare una scadenza assoluta (con `expireat`), in termini di un timestamp Unix (secondi dal primo gennaio 1970), o un tempo alla scadenza in secondi (con il comando `expire`). Questo comando, basandosi unicamente sul nome della chiave, è indipendente dal tipo di struttura dati associata alla chiave stessa.

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

Il primo comando cancellerà la chiave (e il valore ad esso associato) dopo 30 secondi. Il secondo farà la stessa cosa alle 12:00 a.m. del 31 dicembre 2012.

Proprio questa funzionalità rende Redis ideale anche per diventare un *caching engine*. E' possibile determinare quanto tempo un elemento debba ancora esistere tramite il comando `ttl` (*time to live*) ed è possibile rimuovere la scadenza di una chiave impartendo sulla stessa il comando `persist`:

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

Si segnala infine un comando specifico per le stringhe, `setex` che imposta il valore di una stringa e ne imposta il tempo di vita in un unico comando atomico.

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

Code, Pubblicazione e Sottoscrizione

I comandi `blpop` e `brpop` relativi a liste, ritornano e rimuovono rispettivamente il primo e l'ultimo elemento di una lista (analogamente a `lpop` e `rpop`) però in maniera sincrona, cioè bloccandosi fino a che un elemento risulta disponibile. Questo rende questi comandi ottimi per l'implementazione di una coda.

Oltre a questo, Redis possiede supporto nativo per la pubblicazione e sottoscrizione a canali. E' possibile provare immediatamente questa funzionalità aprendo una seconda finestra con Redis a linea di comando. Nella prima finestra si sottoscrive il canale `warnings`:

```
subscribe warnings
```

All'invio del comando verranno fornite alcune informazioni relative alla sottoscrizione. Nell'altra finestra si pubblichi un messaggio sullo stesso canale:

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

Alla prima finestra si dovrebbe vedere il seguente messaggio:

```
1) "message"
2) "warnings"
3) "it's over 9000!"
```

Per sottoscrivere più canali: `subscribe channel1 channel2 ...`. Per sottoscrivere un pattern di canali: `psubscribe warnings:*`. Si usino i comandi `unsubscribe` and `punsubscribe` per terminare l'ascolto di uno, più canali o un *pattern* di canali.

Si noti infine che il comando `publish` ritorna il numero di client in ascolto del messaggio (in questo caso 1).

Strumenti di Debug

Il comando `monitor` è un utile strumento di *debugging* e permette di vedere come la propria applicazione interagisca con Redis. Si prema Ctrl-C per uscire dalla modalità `subscribe` e si inserisca il comando `monitor`. Nell'altra finestra si esegua un qualsiasi comando (come un `set` o un `get`): si dovrebbero vedere i comandi lanciati, assieme ai parametri, all'interno della prima finestra.

Bisognerebbe prestare attenzione a non usare il comando `monitor` in produzione: esso è inteso infatti come strumento di debug da usare durante lo sviluppo e potrebbe portare a rallentamenti del sistema.

Accanto a `monitor`, Redis possiede il comando `slowlog` che funziona come un ottimo strumento di profilazione. Esso aggiunge ad un file di log ogni comando che impiega più di uno specificato numero di **micro**secondi. Nella prossima sezione si spiegherà più in dettaglio la configurazione di Redis, nel frattempo si accenna solo che è possibile registrare tutti i comandi con il seguente comando:

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

Si impartiscano ora alcuni comandi, e si visualizzino le righe del log, o le 10 più recenti, scrivendo:

```
slowlog get
slowlog get 10
```

E' possibile inoltre ottenere il numero delle righe presenti nello `slow log` digitando `slowlog len`.

Qualsiasi comando si sia impartito dovrebbero essere presenti quattro parametri:

- Un id auto incrementale
- Un timestamp Unix relativo al momento dell'esecuzione del comando
- Il tempo in microsecondi che ha impiegato il comando
- Il comando e i suoi parametri

Lo `slow log` è mantenuto in memoria, per cui eseguirlo in produzione, anche con un livello non molto alto, non dovrebbe essere un problema, anche perché di default mantiene le ultime 1024 righe.

Ordinamento

Uno dei comandi più potenti è il comando `sort`. Permette di ordinare i valori di una lista, un insieme o un insieme ordinato (si noti a questo proposito che gli insiemi ordinati sono ordinati per un punteggio assegnato, non secondo i valori effettivi contenuti nell'insieme). Per esempio:

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

`sort` ritorna il valori ordinati dal più basso al piu alto. Ecco un esempio più avanzato:

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

L'ultimo comando mostra come effettuare una paginazione tra i risultati (attraverso `limit`), come ritornare i valori in ordine decrescente (`desc`) e come ordinarli lessicograficamente invece che numericamente (`alpha`).

Ma il vero punto forte del comando `sort` è la possibilità di ordinamento rispetto a un oggetto referenziato secondo quanto visto precedentemente con liste, insiemi e insiemi ordinati. Il comando `sort` ha la possibilità di dereferenziare tali relazioni e ordinare i valori associati. Tornando all'esempio del sistema di bug tracking, supponiamo che tale sistema dia la possibilità agli utenti di tenere sotto controllo una serie di bugs. Si potrebbe usare un `set` per tenere traccia di quali *bugs* siano sotto osservazione dell'utente `leto`:

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

Si vorrà sicuramente avere la possibilità di ordinare per gravità (*severity*). Per prima cosa, si aggiungano i dati relativi alla gravità:

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

Per ordinare i bugs per *severity*, dalla più alta alla più bassa, si può scrivere:

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

Redis sostituirà automaticamente l'asterisco del pattern (indicato dal `by`) con i valori della lista/insieme/insieme ordinato. Verrà così creato il nome della chiave usata per ricercare i valori da ordinare.

Anche se con Redis è possibile trattare milioni di chiavi, il codice sopra potrebbe diventare presto poco efficiente. Fortunatamente `sort` può lavorare anche con gli hash e le sue proprietà. Al posto di avere delle chiavi di primo livello è quindi preferibile sfruttare gli hash:

```
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, ....}"
```

Non solo risulta tutto più organizzato ed è possibile sia un ordinamento per `severity` o `priority`, ma è anche possibile dire a Redis che proprietà restituire:

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

Redis riconosce le sequenze `->` e le usa per riferirsi a specifici campi del proprio hash. Anche il parametro `get` sfrutta la sostituzione nella ricerca del campo per restituire i dettagli del *bug*.

`sort` ha una complessità molto elevata: O(N+M*log(M)), dove N è la numerosità degli elementi da ordinare e M il numero degli elementi restituiti. D'altra parte il risultato del comando `sort` può essere salvato in una lista (associata alla chiave fornita come parametro alla direttiva `store`):

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

Le possibilità offerte dal comando `sort` assieme a quelle offerte dai comandi di `expire` visti in precedenza, possono formare una combinazione molto interessante.

In Questo Capitolo

In questo capitolo ci si è concentrati sui comandi generici o applicabili a ogni struttura dato. Come ogni cosa il loro uso dipende dalle esigenze. Non è detto che si debba usare la funzionalità di `expire`, pubblicazione/sottoscrizione o ordinamento. Ma in caso la propria applicazione lo richieda, conoscere della loro esistenza può risolvere in modo egregio parecchi problemi. Si ricordi, comunque, che la lista dei comandi Redis non si esaurisce qui, non appena digerito il contenuto di questo libro vale la pena di addentrarsi nella lista completa.

Capitolo 5 - Amministrazione

Il nostro ultimo capitolo è dedicato agli aspetti amministrativi dell'esecuzione di Redis. Non si ha comunque alcuna pretesa di completezza nell'esporre tali aspetti amministrativi. Alla fine del capitolo si avrà comunque la risposta ai dubbi più comuni che un nuovo utente può avere nell'uso di Redis.

Configurazione

Quando si è lanciato la prima volta Redis server, esso ci ha avvertito della mancanza del file `redis.conf`. Questo file viene usato per configurare i vari aspetti di Redis. Ad ogni rilascio viene fornito un file di configurazione ben documentato e commentato (per questo motivo non si citeranno qui tutte le opzioni). Il file di esempio contiene le configurazioni di default, per cui è utile sia per sapere le svariate possibilità di personalizzazione che per conoscere i valori di default. Si può trovare, per l'ultima versione disponibile al momento della scrittura di questo libro, all'URL `https://github.com/antirez/redis/raw/2.4.6/redis.conf`. **Per adattarlo alla propria versione sarà sufficiente sostituire "2.4.6" nell'indirizzo con la propria specifica versione. Si può conoscere la propria versione semplicemente eseguendo il comando `info` da linea di comando e guardare il primo valore.**

Oltre al file `redis.conf`, è possibile usare il comando `config set` per impostare opzioni individuali. Si è già fatto uso di tale comando quando si è impostato il valore di `slowlog-log-slower-than` a 0, in modo che venisse profilato ogni comando.

Analogamente è presente il comando `config get` che visualizza il valore di un impostazione. Questo comando supporta il `pattern matching`, per cui per visualizzare tutte le impostazioni relative alle funzionalità di *logging*, si può scrivere:

```
config get *log*
```

Autenticazione

Redis può richiedere l'inserimento di una *password* impostando la proprietà `requirepass` con la *password* desiderata. A questo punto ogni *client* che voglia interagire con Redis server dovrà impartire il comando `auth <password>`.

A questo punto si pone un altro problema: una volta che un client è autenticato può eseguire ogni comando, incluso il comando `flushall` che cancella ogni chiave dal database. Per questo, in configurazione, è possibile rinominare ogni comando ed avere una sorta di cifratura dello stesso:

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

Come soluzione più radicale è possibile disabilitare completamente un comando assegnandogli come nuovo nome la stringa vuota.

Dimensioni Massime

Ci si potrà chiedere "quante chiavi è possibile memorizzare?". Si potrebbe in maniera analoga aver bisogno di sapere quante proprietà è possibile associare a un hash, o quanti elementi può contenere una lista o un insieme. Si tenga conto che, per ogni istanza Redis, il limite pratico per tutti questi aspetti è dell'ordine delle centinaia di milioni di elementi.

Ridondanza

Redis supporta la ridondanza, cioè per ogni scrittura su un istanza *master*, una o più istanze *slaves* sono mantenute aggiornate con il *master*. Per ottenere ciò si usa la configurazione `slaveof` (istanze che non abbiano tale tipo di configurazione sono per definizione istanze di tipo *master*).

La ridondanza ha lo scopo di proteggere i propri dati copiandoli su server differenti. La replicazione può essere usata anche per aumentare le *performance*, facendo in modo che le scritture vengano demandate alle istanza *slaves*. Potrebbero rispondere con dei dati leggermente non aggiornati, ma per molti tipi di applicazione questo potrebbe costituire un buon *tradeoff*.

Sfortunatamente, al momento della stesura di questo libro almeno, Redis non fornisce ancora un meccanismo di *failover* automatico. Non viene cioè, quando il master cade, promosso automaticamente uno *slave* a *master*. Ciò può essere fatto comunque manualmente. E' necessario ricorrere a strumenti e *scripts* esterni che monitorino e risolvano eventuali problemi per avere un *high availability* con Redis.

Backups

Fare un backup in Redis è semplicemente una questione di una copia di un file su una qualsiasi altra collocazione (cartella, S3, FTP,...). Di default Redis salva l'istantanea del database nel file `dump.rdb`. In qualsiasi momento è possibile effettuare un `scp`, `ftp` or `cp` (o qualsiasi cosa) di tale file.

Una pratica comune è quella di disabilitare sia l'istantanea che l'aggiunta al file (AOF o `append on file`) dal *master* e lasciare che se ne occupi lo *slave*. Questo può ridurre il carico dal master e permette di impostare parametri più conservativi per il salvataggio dei dati (per esempio un tempo tra snapshots minore) senza incidere sulle performance globali del sistema.

Scalabilità e Cluster con Redis

La replicazione delle istanze è lo strumento più immediato che un sito in crescita può sfruttare. In effetti alcuni comandi risultano costosi (`sort` ad esempio) e delegare la loro esecuzione ad uno *slave* può contribuire a mantenere inalterate le performance globali del sistema.

A parte questo, una scalabilità vera con Redis si ottiene distribuendo le chiavi su molteplici istanze Redis (che in teoria potrebbero anche essere eseguite all`interno della medesima macchina, si ricordi al proposito che Redis e' *single-threaded*). Al momento, tutto questo è ancora delegato all`utente (anche se molti drivers Redis forniscono out of the box degli

algoritmi per l'*hashing* utili allo scopo). Pensare a come strutturare i propri dati in modo da poter fornire una scalabilità orizzontale esula dagli scopi di questo libro.

La buona notizia è che sono in corso lavori per avere un Redis Cluster. Non solo verrà fornita una scalabilità orizzontale, incluso il ribilanciamento, ma verrà anche garantito un failover automatico per l'alta disponibilità.

L'alta disponibilità e la scalabilità sono funzionalità che possono essere ottenute fin d'ora, a patto di impiegare tempo e fatica. Una volta disponibile, Redis Cluster renderà le cose più facili.

In Questo Capitolo

Dato il numero di progetti e siti che usano Redis già oggi, non vi è dubbio che Redis sia già maturo per la produzione. D'altra parte, per alcuni strumenti, specialmente riguardanti sicurezza e disponibilità, è tuttora uno strumento giovane. Redis Cluster, che dovrebbe arrivare presto, farà fronte ad alcune di queste lacune riguardanti l'amministrazione.

Conclusioni

Per molti versi, Redis offre una semplificazione nel modo di trattare i dati. Toglie sia l'astrazione che, al contempo, la complessità di altri sistemi. In molti casi questo può fare di Redis la scelta sbagliata. In altri casi sembrerà che Redis sia nato per i propri dati.

In ultima istanza possiamo tornare a qualcosa detto proprio all'inizio: Redis è facile da imparare. Esistono molte diverse tecnologie relative alla persistenza, ed è difficile scegliere su quale investire del tempo di apprendimento. Un reale beneficio che Redis offre fin dall'inizio è la sua semplicità, penso quindi che sia uno dei migliori investimenti che si possano fare.