The Little Redis Book

by Karl Seguin

Об этой книге

Лицензия

Об авторе

Карл Сегуин - разработчик, имеющий опыт работы в разных областях и с разными технологиями. Активный участник проектов Открытого/Свободного ПО, технический писатель и участник различных конференций. Является автором различных статей о Redis и нескольких утилит для работы с этой системой. На Redis работает система ранжирования и статистики на его бесплатном сервисе для разработчиков казуальных игр - mogade.com.

Карл написал The Little MongoDB Book, популярную бесплатную книгу о MongoDB.

Вы можете найти его блог по адресу `http://openmymind.net`, а также следить за ним в Twitter: @karlseguin.

Благодарности

Особая благодарность выражается Перри Нилу за его зоркий взгляд, ум и страсть. Ты оказал мне неоценимую помощь. Спасибо.

Перевод

Меня зовут Андрей Кондратович. Вы можете найти меня в Twitter.

Данный перевод выполнен с целью популяризации Redis среди русскоговорящих разработчиком. Книга является очень удобным и компактным руководством.

Я благодарен следующим людям за помощь в переводе оригинального текста и вычитке результата:

- Dmitry Scriptin
- Denis Veselov
- polinom

Актуальная версия

Актуальная версия английского варианта книги доступна в репозитории Карла.

Последнюю версию перевода на русский язык можно найти по адресу.

Введение

За последние пару лет технологии и средства хранения и доступа к данным развивались невероятными темпами. Можно с уверенностью говорить, что реляционные базы данных не собираются исчезать, но в то же время мы видим, что экосистема вокруг данных уже никогда не будет прежней.

Из всех новых инструментов и решений, для меня, Redis оказался наиболее интересным. Почему? Во-первых, по причине невероятной простоты изучения. Час является самой подходящей единицей измерения, когда мы говорим о времени, необходимом для ознакомления с Redis. Во-вторых, эта система решает специфический класс задач, будучи в то же время достаточно универсальной. Что это значит? Redis не пытается охватить весь спектр задач, касающихся работы с данными. По мере знакомства с Redis будет все более очевидно, что может и чего не может эта система. И, если Redis может что-либо, для вас, как для разработчика, это будет приятным опытом.

Вы можете построить целое приложение, используя только Redis. Однако, я думаю, что большинство людей обнаружит, что Redis дополняет их классические решение для работы с даными, будь то реляционная СУБД, документо-ориентированная система или что-то еще. Это тот тип систем, которые вы используете для решения специфических задач. В этом смысле Redis близок к индексирующему движку. Вы не будете писать ваше приложение полностью на Lucene, но если вам нужна хорошая система поиска, она подарит вам полезный опыт. Конечно, сходства между Redis и поисковыми движками на этом заканчиваются.

Цель этой книги - построить базис, необходимый для работы с Redis. Мы сфокусируем наши усилия на изучении пяти структур данных Redis и рассмотрим различные подходы к моделированию. Также, затронем некоторые ключевые вопросы администрирования и техники отладки.

Начало работы

Мы все учимся по-разному: кто-то предпочитает собственный опыт, кто-то смотрит видео, кто-то читает. Ничто не поможет вам понять Redis лучше, чем собственный опыт. Redis легко устанавливается, и в комплект установки входит простая командная оболочка, предоставляющая все необходимые возможности. Давайте потратим пару минут на то, чтобы установить и запустить Redis у себя на компьютере.

В Windows

Redis официально не поддерживает Windows, но есть возможности запустить ее на этой системе. Вы вряд ли станете так делать на рабочем сервере, но во время разработки я не сталкивался с какими-либо ограничениями.

Сначала перейдите на `https://github.com/dmajkic/redis/downloads` и скачайте последнюю версию (вверху списка).

Распакуйте архив и, в зависимости от архитектуры вашего компьютера, откройте директорию `64bit` или `32bit`.

В *nix и MacOSX

Для пользователей *nix и Mac, сборка Redis из исходников является лучшим вариантом. Инструкции, вместе с номером последней версии, доступны по адресу http://redis.io/download. На момент написания этих строк последней версией является 2.4.6. Для установки выполните:

```
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
```

(В качестве альтернативы, Redis доступна через различные мнеджеры пакетов. Например, пользователи MacOSX с установленным Homebrew могут просто выполнить команду `brew install redis`.) (*Установка через менеджер пакетов может оказаться гораздо более удачным решением с точки зрения сохранения правильной конфигурации вашей системы - прим. перев.*)

Если вы собрали систему из исходных текстов, исполняемые файлы можно найти в директории `src`. Перейдите в эту директорию, введя команду `cd src`.

Запуск и Подключение к Redis

Если все прошло успешно, исполняемые файлы Redis должны быть в вашем распоряжении. Redis имеет множество исполняемых файлов. Мы сосредоточимся на сервере и командной оболочке (DOS-подобный клиент). Давайте начнем с сервера. В Windows, двойным кликом запустите `redis-server`. В *nix/MacOSX выполните команду `./redis-server` в терминале.

Если вы прочитаете стартовое сообщение, вы увидите предупреждение о том, что файл `redis.conf` не обнаружен. Вместо этого Redis воспользуется настройками по умолчанию, что вполне подойдет для наших целей.

Далее, запустите командную оболочку Redis двойным кликом на `redis-cli` (Windows) или командой `./redis-cli` (*nix/MacOSX). Таким образом создастся подключение к локальному серверу на 6389-м TCP-порту по умолчанию.

Вы можете убедиться, что все работает, введя команду `info` в командную строку. Вы должны увидеть множество пар ключ-значение, которые предоставляют большое количество информации о состоянии сервера.

Если у вас возникли проблемы с выполнением описанных выше действий, я рекомендую поискать помощь на официальных форумах поддержки Redis.

Драйверы Redis

Как вы скоро узнаете, API Redis лучше всего описать как набор функций в простом процедурном стиле. Это значит, что, используете ли вы командную оболочку или драйвер вашего любимого языка программирования, все работает одинаково. Следовательно, у вас не должно возникнуть проблем, если вы решите работать посредством языка программирования. Если хотите, зайдите на страницу со списком клиентов и скачайте необходимый драйвер.

Глава 1 - Основы

Почему же Redis такой особенный? Какие задачи он решает? На что следует обращать внимание разработчикам? Перед тем, как ответить на все эти вопросы, мы должны понять, чем же все-таки является Redis.

Очень часто Redis описывают как персистентное (*то есть сохраняющееся после окончания работы создавшего его процессса - прим. перев.*) хранилище данных типа ключ-значение в оперативной памяти. Я не думаю, что это совсем точное определение. Redis, действительно, держит все данные в памяти (позже мы вернемся к этому), и он сохраняет данные на диск для обеспечения персистентности. Но он не просто хранилище данных типа ключ-значение. Очень важно разобраться в этом неточном определении, иначе у вас сложится впечатление, что спектр решаемых Redis проблем весьма узок.

Суть в том, что Redis предоставляет пять разных структур данных, только одна из которых собственно и есть структура типа ключ-значение. Понимание этих пяти структур данных, как они работают, какие методы они предоставляют для взаимодействия, и что вы сможете сделать с их помощью, как раз и являются путем к пониманию Redis. Но сначала давайте разберемся с тем, что же означает предоставление структур данных.

Если мы применим эту концепцию к реляционному миру, мы можем сказать, что базы данных предоставляют один тип структур данных - таблицы. Таблицы одновременно сложные и гибкие. Существует очень мало вещей, которые нельзя смоделировать, сохранить и которыми нельзя управлять с помощью таблиц. Тем не менее, они не идеальны. А именно, они не на столько простые, или не на столько быстрые, как хотелось бы. Что, если вместо универсальной структуры мы бы использовали специализированные структуры? В этом случае, наверное, найдутся проблемы, которые мы не сможем решить (или, по крайней мере, не сможем решить достаточно хорошо). Однако, в любом случае, мы выиграем на простоте и скорости.

Использование специфичных структур данных для специфичных задач? Разве это не тот подход, который мы постоянно используем при программировании? Вы не используете хеш-таблицы для всех данных программы, то же самое и со скалярными переменными. (*Скалярные типы данных: строки, целые числа, дроби, булев тип - прим. перев.*) Я считаю, в этом и состоит подход Redis. Если вы работаете со скалярами, списками, или множествами, почему бы не хранить их как скаляры, списки, хеши и множества? Почему проверка на существование должна быть сложнее чем просто вызов `exists(key)` или медленнее чем O(1) (постоянное время выполнения выражения, которое не зависит от количества записей в хранилище)?

Составные Кирпичики

Базы Данных

Redis использует знакомую всем концепцию базы данных. База данных содержит набор данных. Типичное предназначение базы данных - это группирование всей информации

определенного приложения в месте и изоляция ее от других приложений.

В Redis база данных идентифицируется просто числом, которое по умолчанию равняется `0`. Если вы хотите сменить базу данных, то вы можете сделать это командой `select`. В командной строке просто введите `select 1`. Redis должен ответить сообщением `OK` и в терминале вы должны увидеть что-то типа `redis 127.0.0.1:6379[1]>`. Если вы хотите переключиться обратно на базу по умолчанию, просто введите в командной строке `select 0`.

Команды, Ключи и Значения

Несмотря на то, что Redis больше, чем просто хранилище типа ключ-значение, в его основе каждая из пяти используемых структур данных имеет, как минимум, ключ и значение. Очень важно разобраться в том, что такое ключи и что такое значения, перед тем как двигаться дальше.

Ключ - это то, чем мы помечаем части информации. Мы будем пользоваться ключами часто, но пока достаточно знать, что ключ может выглядеть вот так: `users:leto`. Под таким ключом можно ожидать информацию о пользователе под именем `leto`. Двоеточие не имеет никакого особого значения, но использование подобных разделителей является хорошим тоном.

Значение - это данные, которые ассоциированы с ключом. Это может быть что угодно. Иногда это строки, иногда числа, а иногда там хранятся сериализованные объекты (в виде JSON, XML или любых других форматов). В основном, Redis рассматривает значение как массив байт и не интересуется тем, что они собой представляют. Обратите внимание, что разные драйверы производят сериализацию по-разному. Поэтому, здесь мы будем говорить только о строках (string), целых числах (integer) и JSON.

Давайте попрактикуемся. Выполните следующие команды:

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

Почти все команды Redis выглядят подобным образом. Сначала идет сама команда, в нашем случае `set`, а потом параметры. Команда `set` принимает два параметра: ключ, который мы сохраняем, и значение, которое связано с ним. Многие, но не все, команды принимают ключ (это обычно первый параметр). А теперь угадайте, как получить это значение? Надеюсь, что вы догадались (не расстраивайтесь, если это не так):

```
get users:leto
```

Попробуйте самостоятельно поиграть с подобными комбинациями. Ключ и значение - это основная концепция, и простейший способ работать с ней - это команды `get` и `set`. Создайте других пользователей, попробуйте разные виды ключей и значений.

Запросы

По мере нашего знакомства с Redis, мы поймем две вещи. Ключи это - все, а значения - ничто. Или, другими словами, Redis не позволяет использовать в запросах значения объектов. С вышесказанного следует, что мы не сможем найти пользователя, который живет на планете dune.

У многих это может вызвать некоторые беспокойства. Мы живем в мире, где запросы к базам данных стали столь гибкие и мощные, что подход Redis кажется примитивным и неудобным. Не дайте ввести себя в заблуждение. Redis не является универсальным решением на все случаи жизни. Существуют проблемы, которые просто не решаются с помощью Redis (из-за ограничений в запросах). Также стоит принять к сведению, что в некоторых случаях, вы найдете другие способы смоделировать ваши данные.

Мы рассмотрим более конкретные примеры в дальнейшем. Сейчас очень важно, чтобы мы разобрались в этих базовых принципах Redis. Этот подход позволяет использовать в качестве значений что угодно - Redis никогда не обращается к ним. Мы научимся строить модели наших данных по-другому, как того требует этот новый мир.

Память и Персистентность

Мы упоминали ранее, что Redis является персистентным хранилищем в оперативной памяти. По умолчанию, для поддержания персистентности, Redis сохраняет снимки базы данных на диск в зависимости от того, сколько ключей было изменено. Вы можете настроить этот процесс таким образом, что если X - количество изменившихся ключей, то базу данных нужно сохранять каждые Y секунд. По умолчанию, Redis сохраняет базу с интервалами от 60 секунд, если 1000 или более ключей изменились, до 15 минут, если изменились хотя бы 9 ключей.

Кроме того, (или в дополние к сохранению снимков базы на диске), Redis может быть запущен в режиме дозаписи (append mode). Каждый раз, когда ключ меняется, на диске дописывается запись в файле, открытом в режиме дозаписи (новые записи добавляются в конец файла). Иногда стоит принять риск потери данных последних 60 секунд в случае аппаратной или программной ошибки в обмен на производительность. В других случаях такой риск неприемлем. Redis дает вам выбор. В главе 5 мы рассмотрим третью возможность - обеспечение персистентности за счет вспомогательного (slave) хранилища.

Что касается использования памяти, Redis хранит все данные в оперативной памяти. Явным следствием этого является цена использования Redis: оперативная память до сих пор является самой дорогой аппаратной частью серверов.

Я чувствую, что некоторые разработчики забывают о том, как мало места могут занимать данные. Полное собрание сочинений Уильяма Шекспира занимает примерно 5,5 МБ. Что касается масштабирования, другие решения, как правило, ограничены временем ввода/вывода или производительностью процессора. Какое из ограничений (память или ввод/вывод) вынудит вас масштабировать приложение на большем количестве серверов,

зависит от типа обрабатываемых данных и от того, как вы их храните и запрашиваете. Пока вы не храните огромные медиа-файлы в Redis, использование оперативной памяти не должно быть проблемой. Для приложений, где это все же будет проблемой, вы скорее всего выберете дополнительный ввод/вывод, чем ограничения размера памяти.

В Redis была реализована поддержка виртуальной памяти. Тем не менее, эта функция была признана неудачной (самими инженерами Redis) и ее использование не рекомендуется.

К слову, 5,5 МБ сочинений Шекспира могут быть сжаты до примерно 2 МБ. Redis не использует автоматическое сжатие, но, поскольку он трактует данные, как набор байт, нет причин, по которым вы не могли бы обменять время обработки на дополнительную память путем сжатия/распаковки данных.

Собираем Все Вместе

Мы коснулись нескольких общих тем. Последнее, что я хотел бы сделать перед погружением в Redis, это собрать несколько этих тем вместе. А именно, ограничения запросов, структуры данных и способ хранения данных Redis в оперативной памяти.

Когда вы собираете три эти вещи вместе, вы получаете нечто замечательно - скорость. Некоторые люди думают: «Конечно, Redis быстр, ведь все хранится в памяти». Но это только часть. Настоящая причина, по которой Redis так хорош по сравнению с другими решениями, заключается в его специализированных структурах данных.

Насколько высока скорость? Это зависит от многих вещей: какие команды и типы данных вы используете и так далее. Но производительность Redis обычно измеряется в десятках тысяч или даже сотнях тысяч операций **в секунду**. Вы можете запустить `redis-benchmark` (расположен в той же директории, что и `redis-server` и `redis-cli`), чтобы проверить это.

Однажды я переписал код, использовавший традиционную модель, для использования Redis. Нагрузочный тест, написанный мной, занимал более 5 минут при использовании реляционной модели данных. Он занял примерно 150 мс с Redis. Вы не всегда будете получать такой значительный прирост, но я надеюсь, что этот пример даст вам представление о том, с чем мы имеем дело.

Важно понимать этот аспект Redis, поскольку это влияет на то, как ваше приложение с ним взаимодействует. Разработчики с опытом работы в SQL обычно стараются минимизировать количество обращений к базе данных. Это хороший совет для всех систем, включая Redis. Но, с учетом того, что мы имеем дело с более простыми структурами данных, нам иногда придется обращаться к серверу Redis несколько раз для достижения цели. Такой шаблон доступа к данным может показаться неестественным на первый взгляд, но на самом деле это незначительно по сравнению с получаемой взамен производительностью.

В Этой Главе

Несмотря на то, что мы лишь немного поиграли с Redis, был рассмотрен широкий спектр вопросов. Не волнуйтесь, если что-то не ясно до конца - как, например, запросы данных. В следующей главе мы перейдем к практике и все ваши вопросы, я надеюсь, найдут ответы.

Ключевые моменты этой главы:

- Ключи - это строки, идентифицирующие фрагменты данных (значения)
- Значения - это произвольные массивы байт, которым Redis не придает никакого смысла
- Redis предоставляет пять специализированных структур данных (он реализован на их основе)
- Все упомянутое вместе делает Redis быстрым и простым в использовании, но не подходящим для любого сценария использования

Глава 2 - Структуры Данных

Пришло время взглянуть поближе на пять структур данных Redis. Мы узнаем, что представляет собой каждая из этих структур, что с ней можно делать, и в каких случаях она будет полезна.

До этого момента, мы видели в Redis только команды, ключи и значения. Не было сказано ничего конкретного о структурах данных. Как Redis решал какую структуру данных использовать, когда мы выполняли команду `set`? На самом деле каждая команда применима только к одной конкретной структуре данных. Например, когда вы используете `set`, данные сохраняются в виде строки (string). Когда вы используете `hset`, вы сохраняете данные в хеше (hash). С учетом небольшого количества команд в Redis, к этому достаточно просто привыкнуть.

Сайт Redis содержит отличный справочный раздел. Нет смысла повторять уже проделанную работу. Мы рассмотрим только самые важные команды, которые необходимы для понимания предназначения каждой структуры данных.

Нет ничего важнее, чем получать удовольствие от практики и экспериментов. Вы всегда можете стереть все значения в вашей базе данных, введя команду `flushdb`, поэтому не стесняйтесь и попробуйте сделать что-нибудь интересное!

Строки (Strings)

Строки являются самой простой структурой данных в Redis. Когда вы думаете о паре ключ-значение, вы думаете о строках. Имея уникальные имена (ключи), значения строк могут быть какими угодно. Я предпочитаю называть их «скалярными величинами» - возможно, это только мое предпочтение.

Мы уже видели типичный пример использования строк: хранение значений объектов с определенными ключами. Это именно то, с чем вам придется сталкиваться наиболее часто:

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

Дополнительно, Redis позволяет выполнять некоторые стандартные действия со строками. Например, `strlen <ключ>` используется для вычисления длины значения, связанного с ключом; `getrange <ключ> <начало> <конец>` возвращает подстроку из строки; `append <ключ> <значение>` добавляет введенное значение к концу существующей строки (или создает новое значение, если указанный ключ не определен). Попробуйте эти команды в действии. Вот что получилось у меня:

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

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

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

Сейчас вы наверное думаете, что это, конечно, замечательно, но лишено смысла. С помощью этих операций невозможно (в общем случае) получить или добавить значение в JSON-строку. Вы правы - некоторые команды, особенно для работы со строками, имеют смысл только при использовании специфичных форматов данных.

Ранее мы узнали, что Redis не придает смысла введенным вами значениям. Это действительно так в большинстве случаев. Тем не менее, некоторые команды предназначены лишь для строк, значения которых удовлетворяют определенным условиям. В качестве грубого примера можно привести использование команд `append` и `getrange` для какого-нибудь нестандартного способа сериализации данных (*сериализация - преобразование произвольного объекта/данных в строку - прим. перев.*) Более наглядным примером будут команды `incr`, `incrby`, `decr` и `decrby`. Они предназначены для выполнения инкремента/декремента (увеличения/уменьшения значения) значений строк (*в примерах ниже строки в этих командах интерпретируются как числа, в связи с чем используемый автором термин «скалярные значения» (scalars) имеет более точное значение, чем «строки» - прим. перев.*):

```
> 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
```

Как вы можете догадаться, строки Redis отлично подходят для аналитики. Попробуйте инкрементировать значение `users:leto` (не-числовое) и посмотреть, что получится (вы должны получить ошибку).

Более сложным примером будут команды `setbit` и `getbit`. Есть замечательный пост о том, как Spool (*Spool - автор поста, на который ссылаются выше - прим. перев.*) использует эти две команды для эффективного ответа на вопрос «сколько уникальных посетителей было у нас сегодня?». Для 128 миллионов пользователей ноутбук генерирует ответ менее чем за 50 мс и использует всего лишь 16 МБ памяти.

Не так уж важно понимание того, как работают битовые маски и как Spool использует их, но важно понять, что строки в Redis являются гораздо более мощным инструментом, чем кажется на первый взгляд. Тем не менее, наиболее частыми примерами использования являются те, что мы видели выше: хранение объектов (простых или сложных) и счетчиков. Кроме того, поскольку получение значения по ключу настолько быстрая операция, строки часто используются для кеширования данных.

Хеши (Hashes)

Хеши - хороший пример того, почему называть Redis хранилищем пар ключ-значение не совсем корректно. Хеши во многом похожи на строки. Важным отличием является то, что они предоставляют дополнительный уровень адресации данных - поля (fields). Эквивалентами команд `set` и `get` для хешей являются:

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

Мы также можем устанавливать значения сразу нескольких полей, получать все поля со значениями, выводить список всех полей и удалять отдельные поля:

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

Как вы видите, хеши дают чуть больше контроля, чем строки. Вместо того, чтобы хранить данные о пользователе в виде одного сериализованного значения, мы можем использовать хеш для более точного представления. Преимуществом будет возможность извлечения, изменения и удаления отдельных частей данных без необходимости читать и записывать все значение целиком.

Рассматривая хеши как структурированные объекты, такие как данные пользователя, важно понимать, как они работают. И это правда, что такой подход может быть полезен для повышения производительности. Тем не менее, в следующей главе мы увидим, как хеши могут использоваться для организации ваших данных и удобства запросов к ним. Мне кажется, что именно в этом хеши особенно хороши.

Списки (Lists)

Списки позволяют хранить и манипулировать массивами значений для заданного ключа. Можно добавлять значения в список, получать первое и последнее значение из списка и манипулировать значениями с заданными индексами. Списки сохраняют порядок своих значений и имеют эффективные операции с использованием индексов. Мы можем создать

список `newusers`, содержащий последних зарегистрировавшихся на нашем сайте пользователей:

```
rpush newusers goku
ltrim newusers 0 50
```

Сначала мы добавляем нового пользователя в начало списка, затем укорачиваем список так, чтобы он содержал 50 последних записей. Это типичный пример использования. Операция `ltrim` имеет асимптотическую сложность O(N), где N - число значений, которое мы удаляем. В этом случае, если мы всегда укорачиваем список после каждой единичной вставки, эта операция имеет постоянную производительность O(1) (потому что N всегда будет равным единице).

Сейчас мы первый раз столкнемся с тем, что значение с одним ключом ссылается на значение с другим ключом. Если мы хотим получить сведения о последних 10 пользователях, мы должны сделать следующее:

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

Приведенный выше код на языке Ruby показывает случай многократного обращения к базе данных, о котором мы говорили выше.

Конечно, списки хороши не только для хранения ссылок на другие ключи. Значения могут быть чем угодно. Можно использовать списки для хранения записей журнала или пути, по которому пользователь перемещается по вашему сайту. Если вы создаете игру, вы можете использовать список для хранения последовательности действий пользователя.

Множества (Sets)

Множества используются для хранения уникальных значений и предоставляют набор операций - таких, как объединение. Множества не упорядочены, но предоставляют эффективные операции со значениями. Список друзей является классическим примером использования множеств:

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

Независимо от того, сколько друзей имеет пользователь, мы можем эффективно (O(1)) определить, являются ли пользователи userX и userY друзьями, или нет.

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

Более того, мы можем узнать, имеют ли два пользователя общих друзей:

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

и даже сохранить результат этой операции под новым ключом:

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

Множества отлично подходят для теггинга (*tagging, присвоение семантических ярлыков-меток - прим. перев.*) и отслеживания любых других свойств, для которых повторы не имеют смысла (или там, где мы хотим использовать операции над множествами, такие как пересечение и объединение).

Упорядоченные Множества (Sorted Sets)

Последней и самой мощной структурой данных являются упорядоченные множества. Хеши похожи на строки, но имеют поля, а упорядоченные множества похожи на множества, но имеют счетчики. Счетчики предоставляют возможности упорядочивания и ранжирования. Если мы хотим получить ранжированный список друзей, мы можем сделать следующее:

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

Хотим узнать, сколько друзей с рейтингом 90 и выше имеет пользователь `leto`?

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

Как насчет рейтинга `chani`?

```
zrevrank friends:leto chani
```

Мы используем `zrevrank` вместо `zrank`, поскольку по умолчанию в Redis сортировка происходит от меньшего к большему (мы ранжируем от большего к меньшему). Самым очевидным примером использования упорядоченных множеств являются системы рейтингов. На самом деле упорядоченные множества подойдут для любых случаев, когда вы имеете значения, которые вы хотите упорядочить, используя для этого целочисленные «весовые коэффициенты», и которыми вы хотите управлять на основании этих коэффициентов.

В следующей главе мы увидим как упорядоченные множества могут использоваться для отслеживания событий, основанных на времени (когда время используется в качестве весового коэффициента), что является другим типичным примером использования упорядоченных множеств.

В Этой Главе

Вы обзорно познакомились с пятью структурами данных Redis. Одна из замечательных особенностей Redis состоит в том, что вы можете сделать больше, чем вам показалось на первый взгляд. Вероятно, есть способы использования строк и упорядоченных множеств, о которых еще никто не догадался. Тем не менее, понимая стандартные способы их применения, вы сможете использовать Redis для решения любых типов задач. Кроме того, нет нужды использовать все структуры данных и операции над ними только потому, что Redis дает такую возможность. Не так уж редко многие задачи решаются весьма ограниченным набором команд.

Глава 3 - Использование Структур Данных

В предыдущей главе мы говорили о пяти структурах данных и увидели несколько примеров того, какие задачи они могут решать. Теперь пришло время посмотреть на более специфичные, но все же довольно распространенные, особенности и шаблоны проектирования.

Асимптотическая Сложность (Запись «Большое О»)

В этой книге мы уже упоминали запись асимптотической сложности (или времени выполнения) («большого О») в форме O(N) и O(1). Эта форма записи используется для объяснения того, как некий алгоритм ведет себя при определенном количестве элементов, над которыми совершается операция. В Redis это показывает нам, насколько быстро исполняется команда в зависимости от числа элементов в структуре данных, с которой мы работаем. (*Здесь автор неточен. Это верно не только для Redis, а вообще для любых алгоритмов, причем не только в зависимости от числа элементов, но и от их значений - например, сложность алгоритма вычисления факториала, где в качестве аргумента выступает всего одно число - прим. перев.*)

Документация Redis содержит сведения об асимптотической сложности для каждой команды. Это также указывает на то, какие факторы влияют на производительность. Давайте рассмотрим несколько примеров.

Самые быстрые алгоритмы имеют сложность O(1) - константу. Независимо от того, выполняется ли операция над 5 элементами или 5 млн. элементов, вы получаете одинаковую производительность. Команда `sismember`, показывающая, принадлежит ли элемент множеству, имеет сложность O(1). `sismember` - мощная команда, и ее производительность, помимо прочего, является тому причиной. Многие команды Redis имеют такую сложность.

Логарифмическая сложность - O(log(N)) - следующая по скорости, поскольку нуждается в сканировании все меньшего и меньшего числа элементов. При использовании такого подхода, как «разделяй и властвуй», даже огромное количество элементов быстро разбивается на части за несколько итераций. Команда `zadd` имеет сложность O(log(N)), где N - количество элементов, которые уже включены во множество.

Далее следуют линейные по сложности команды - O(N). Поиск в неиндексированной строке таблицы, а также использование команды `ltrim` являются операциями с такой сложностью. Тем не менее, в случае с `ltrim` N - это не общее количество элементов в списке, а количество удаляемых элементов. Использование `ltrim` для удаления 1 элемента из списка с миллионом элементов будет быстрее, чем удаление 10 элементов из списка, содержащего сотни элементов. (Хотя обе операции, вероятно, будут настолько быстрыми, что вы не сможете измерить их время.)

Команда `zremrangebyscore`, удаляющая элементы упорядоченного множества с весовыми коэффициентами в диапазоне между минимальным и максимальным указанным значением,

имеет сложность O(log(N)+M). То есть имеет смешанную сложность. Читая документацию, мы видим, что N - это общее число элементов множества, а M - количество удаляемых элементов. Другими словами, количество удаляемых элементов, вероятно, сильнее повлияет на производительность, чем общее количество элементов.

Команда `sort`, которую мы более детально рассмотрим в следующей главе, обладает сложностью O(N+M*log(M)). Из этой записи вы можете заключить, что это, вероятно, наиболее сложная в вычислительном плане команда Redis.

Есть и другие степени сложности, из которых две наиболее распространенные - O(N^2) и O(C^N). Чем больше N, тем хуже производительность по сравнению с меньшими N. (*Это верно и для других форм, описанных выше - прим. перев.*) Никакие из команд Redis не имеют такой сложности.

Важно заметить, что асимптотическая запись «большого O» указывает на худший случай. Когда мы говорим, что выполнение операции займет время, определяемое как O(N), мы на самом деле можем получить результат сразу же, но может случиться и так, что искомый элемент окажется самым последним.

Псевдо-Многоключевые Запросы

Типичной ситуацией, в которую вы будете попадать, будет необходимость запрашивать одно и то же значение по разным ключам. Например, вы можете хотеть получить данные пользователя по адресу электронной почты (в случае, если пользователь входит на сайт впервые) и по идентификатору (после входа пользователя на сайт). Одним из ужасных решений будет дублирование объекта в двух строковых значениях:

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

Это неправильно, поскольку такими данными трудно управлять, и они занимают в два раза больше памяти.

Было бы здорово, если бы Redis позволял создавать связь между двумя ключами, но такой возможности нет (и скорее всего никогда не будет). Главным принципом развития Redis является простота кода и программного интерфейса (API). Внутренняя реализация связанных ключей (есть много вещей, которые можно делать с ключами, о чем мы еще не говорили) не стоит возможных усилий, если мы увидим, что Redis уже предоставляет решение - хеши.

Используя хеш, мы можем избавиться от необходимости дублирования:

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

Мы используем поле как вторичный псевдо-индекс, и получаем ссылку на единственный объект, представляющий пользователя. Чтобы получить пользователя по идентификатору, мы используем обычную команду get:

```
get users:9001
```

Чтобы получить пользователя по адресу электронной почты, мы воспользуемся сначала hget, а затем get (код на Ruby):

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

Это то, чем вы, скорее всего, будете пользоваться очень часто. Для меня это как раз тот случай, когда хеши особенно хороши, но это не очевидный способ использования, пока вы не увидите это своими глазами.

Ссылки и Индексы

Мы рассмотрели пару примеров, где одно значение ссылается на другое. Мы видели это, когда смотрели на пример со списком, и мы видели это в разделе выше, когда использовали хеши для того, чтобы сделать запросы немного проще. Это в итоге приводит к необходимости ручного управления индексами и ссылками между значениями. Честно говоря, можно назвать это недостатком, особенно когда вы столкнетесь с необходимостью управлять этими ссылками, обновлять и удалять их вручную. В Redis нет магического решения этой проблемы.

Мы уже видели, как множества часто используются для реализации ручного индекса:

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

Каждый элемент этого множества является ссылкой на строковое значение, содержащее информацию о пользователе. А что, если chani изменит свое имя или удалит учетную запись? Может быть, имеет смысл хранить обратные взаимосвязи:

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

Не беря в расчет сложность поддержки такой структуры, если вы похожи на меня, вы, вероятно, испугаетесь дополнительных расходов памяти и времени обработки таких дополнительных индексных значений. В следующем разделе мы поговорим о путях снижения затрат производительности, возникающих из-за необходимости дополнительных обращений к базе данных (мы кратко упоминали об этом в первой главе).

Если вы задумаетесь об этом, то поймете, что реляционные базы данных также имеют эти накладные расходы. Индексы занимают память, они должны сканироваться и затем соответствующие записи должны извлекаться. От накладных расходов ловко абстрагируются (делается множество оптимизаций для того, чтобы сделать обработку максимально эффективной).

Необходимость ручного управления ссылками в Redis огорчает. Но все первоначальные опасения о влиянии на производительность и потребление памяти должны быть проверены на практике. Я думаю, вы обнаружите, что это не такая уж большая проблема.

Дополнительные Запросы и Конвейерная Обработка

Мы уже упоминали, что частые обращения к серверу являются типичными во время использования Redis. Поскольку это делается часто, будет полезно узнать больше о возможностях, которые мы можем использовать для получения наилучших результатов.

Прежде всего, команды или принимают один или более параметров, или существуют подобные команды, которые принимают несколько параметров. Ранее мы видели команду `mget`, принимающую несколько ключей и возвращающую их значения:

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

Или команда `sadd`, добавляющая одно или более значений к множеству:

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

Redis также поддерживает конвейерную обработку. Обычно, когда клиент посылает запрос к Redis, он ждет ответа, прежде чем послать следующий запрос. С конвейерной обработкой вы можете посылать несколько запросов без ожидания момента, когда они вернут результат. Это снижает накладные расходы обмена данными по сети и может значительно увеличить производительность.

Для Redis ничего не стоит использовать память для создания очереди запросов, поэтому хорошей идеей будет группировать запросы. Насколько большим будет используемый вами пакет запросов зависит от того, какие команды вы используете, и, что более важно, каков размер их параметров. Но, если вы используете команды с длиной параметров примерно в 50 символов, вы, вероятно, можете группировать их тысячами или десятками тысяч.

То, как именно вы исполняете команды в конвейере, будет зависеть от используемого драйвера. В Ruby вы передаете блок в метод `pipelined`:

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

Как вы можете догадаться, конвейерная обработка может значительно ускорить импортирование пакета запросов!

Транзакции

Каждая команда Redis атомарна, включая команды, которые делают несколько вещей. Дополнительно, Redis поддерживает транзакции при использовании нескольких команд.

Возможно, вы знаете, что Redis на самом деле работает в один поток, благодаря чему и гарантируется атомарность (неделимость) каждой команды. Пока исполняется одна команда, остальные не могут быть исполнены. (Мы кратко обсудим масштабирование позже.) Это бывает полезно, когда речь идет о командах, делающих несколько вещей. Например:

`incr` - это, по сути, `get` за которой следует `set`

`getset` устанавливает новое значение и возвращает старое

`setnx` сначала проверяет, существует ли ключ, и устанавливает значение только в том случае, если ключ не существовал

Несмотря на то, что эти команды полезны, вам неизбежно понадобится исполнять несколько команд как атомарную группу. Вы можете это сделать, вызвав сначала команду `multi`, за которой следуют команды, которые вы хотите выполнить как часть транзакции, и в конце вызвать команду `exec` для того, чтобы выполнить команды, или `discard`, чтобы отменить их выполнение. Какие гарантии дает Redis касательно транзакций?

- Команды исполнятся по порядку
- Команды исполнятся как единая, неделимая операция (никакая другая команда не будет исполнена в ходе выполнения транзакции)
- Будут выполнены либо все команды транзакции, либо ни одна из них

Вы можете, и должны, попробовать это в командной строке. Также обратите внимание, что нет причин отказываться от использования конвейерной обработки и транзакций вместе.

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

Наконец, Redis позволяет указывать ключ (или ключи) для отслеживания изменений, и применять транзакцию, если ключ(и) изменился(-лись). Это используется, когда вам нужно получить значения и исполнить код в зависимости от их в одной транзакции. В коде выше мы не смогли бы реализовать собственную команду `incr`, поскольку все команды исполняются вместе, когда вызывается `exec`. Мы не можем сделать так:

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

Транзакции в Redis работают иначе. Но, если мы добавим `watch` к `powerlevel`, мы сможем сделать следующее:

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

Если другой клиент изменит значение ключа `powerlevel` после того, как мы вызвали `watch` для этого ключа, наша транзакция не будет исполнена. Если же значение не изменится, все сработает. Мы можем исполнять этот код в цикле, пока он не сработает.

Анти-Шаблон Использования Ключей

В следующей главе мы поговорим о командах, не относящихся непосредственно к структурам данных. Некоторые из них служат для администрирования и отладки. Но есть одна, о которой я бы хотел упомянуть отдельно - `keys`. Эта команда принимает шаблон и находит все ключи, соответствующие ему. Эта команда кажется подходящей для определенного рода задач, но ее не следует использовать в готовом приложении. Почему? Потому что она осуществляет линейный поиск совпадений с шаблоном по всем ключам. Проще говоря, она медленная.

Как же ее используют? Допустим, вы создаете веб-приложения для баг-трекинга (отслеживания ошибок). Каждая учетная запись будет иметь идентификатор `id_аккаунта` и вы можете решить хранить каждую ошибку в строковом значении с ключом в формате

bug:id_аккаунта:id_ошибки. Если вам когда-нибудь понадобится найти все ошибки для определенной учетной записи (для отображения или удаления в случае удаления учетной записи), вы, возможно, будете (как я когда-то) использовать команду keys:

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

Лучшим решением будет использование хеша. Так же, как мы можем использовать хеши в качестве вторичного индекса, мы можем использовать их для организации данных:

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

Для получения идентификаторов всех ошибок для заданной учетной записи мы просто вызовем hkeys bugs:1233. Для удаления определенной ошибки мы можем использовать hdel bugs:1233 2, а для удаления учетной записи со всеми данными - del bugs:1233.

В Этой Главе

Надеюсь, что эта глава вместе с предыдущей дали вам основы для понимания того, как использовать Redis для реализации полезных возможностей. Есть и другие шаблоны, которые вы можете использовать для построения приложений другого типа, но суть здесь состоит в понимании фундаментальных структур данных и того, как они могут быть использованы для реализации решений, лежащих за пределами того, что вы изначально могли себе вообразить.

Глава 4 - За Пределами Структур Данных

Несмотря на то, что пять структур данных формируют основу Redis, в системе также есть команды, не относящиеся к структурам данных. Мы уже видели некоторые из них: `info`, `select`, `flushdb`, `multi`, `exec`, `discard`, `watch` и `keys`. В этой главе мы рассмотрим несколько других подобных команд.

Срок Существования Ключей

Redis позволяет назначать ключам срок существования. Вы можете использовать абсолютные значения времени в формате Unix (Unix timestamp, количество секунд, прошедших с 1 января 1970 года) или оставшееся время существования в секундах. Эта команда оперирует ключами, поэтому неважно, какая структура данных при этом используется.

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

Первая команда удалит ключ (и ассоциированное с ним значение) по истечении 30 секунд. Вторая сделает то же самое в 12:00, 31 декабря 2012 года.

Это делает Redis идеальным движком для кеширования. Вы можете узнать, сколько еще будет существовать заданный элемент, с помощью команды `ttl`, а также удалить срок существования с помощью команды `persist`:

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

Наконец, есть специальная строковая команда `setex`, позволяющая создавать строковые значения с указанием времени существования одной атомарной операцией (в большей степени только для удобства):

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

(*обновление значения по ключу сотрет информацию о сроке существования, поэтому эту информацию также необходимо обновлять - прим. перев.*)

Публикация Сообщений и Подписка

Над списками в Redis опеределены команды `blpop` и `brpop`, которые возвращают и удаляют, соответственно, первый и последний элементы списка, или же блокируют список, пока указанные элементы в нем не появятся. Это может быть использовано для создания простой очереди.

Более того, Redis поддерживает публикацию сообщений и подписку на каналы как на объекты первого класса (*То есть эти объекты можно создавать, изменять, удалять и т.д. - прим. перев.*). Вы можете сами попробовать это, запустив вторую копию `redis-cli` в другом окне. В первом окне подпишемся на канал (назовем его `warnings`):

```
subscribe warnings
```

Ответом будет информация о подписке. Теперь, в другом окне, опубликуем сообщение в канал `warnings`:

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

Если вы вернетесь в первое окно, вы должны увидеть сообщение, полученное через канал `warnings`.

Можно подписываться на несколько каналов (`subscribe channel1 channel2 channel3 ...`) или на каналы, названия которых соответствуют шаблону (`psubscribe warnings:*`), и использовать команды `unsubscribe` и `punsubscribe` для прекращения подписки на один или более канал.

Наконец, обратите внимание, что команда `publish` вернула значение 1. Это значение показывает число клиентов, получивших сообщение.

Мониторинг и Журнал Медленных Запросов

Команда `monitor` позволяет вам увидеть, что Redis собирается сделать. Этот отличный инструмент отладки дает вам понимание того, как ваше приложение взаимодействует с Redis. В одном из окон `redis-cli` (если одна из копий все еще подписана на канал, вы можете использовать команду `unsubscribe` или же закрыть окно и открыть новое) введите команду `monitor`. В другом запустите команду другого типа (например, `get` или `set`). Вы должны увидеть эти команды вместе с их параметрами в первом окне.

Не следует запускать команду `monitor` на «боевом» сервере, это действительно только средство отладки и разработки. Кроме этого, ничего более об этой команде сказать нельзя. Это просто очень удобный инструмент.

Помимо `monitor`, Redis имеет команду `slowlog`, которая служит отличным инструментом для профилирования (*Измерения производительности - прим. перев.*). Она записывает в журнал все команды, выполнение которых заняло больше времени, чем указанное число **микро**секунд. В следующем разделе мы кратко рассмотрим, как конфигурировать Redis, а сейчас вы можете настроить систему на журналирование всех команд, введя:

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

Далее, запустите несколько команд. Теперь вы можете получить все записи журнала или только самые последние, введя:

```
slowlog get
slowlog get 10
```

Вы также можете получить число записей в журнале медленных запросов, введя `slowlog len`

Для каждой введенной вами команды вы должны увидеть четыре параметра:

- Автоинкрементирующийся идентификатор
- Время (в формате Unix), когда команда была выполнена
- Продолжительность выполнения команды в микросекундах
- Саму команду с параметрами

Журнал медленных запросов поддерживается в памяти, поэтому его ведение на рабочем сервере даже с низким порогом журналирования не должно создавать проблем. По умолчанию в журнале будут храниться последние 1024 записи.

Сортировка

Одна из наиболее мощных команд в Redis - `sort`. Она позволяет сортировать значения в списке, множестве или упорядоченном множестве (элементы упорядоченных множеств упорядочены по значению своих весовых коэффициентов, а не по хранящимся значениям). В самой простой форме, эта команда позволяет сделать следующее:

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

Последняя команда вернет значения, отсортированные от меньшего к большему. Вот более сложный пример:

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

Команда выше показывает, сколько значений выводить (с помощью `limit`), как получать результат в порядке убывания (посредством `desc`), и как упорядочивать по алфавиту, а не по числовым значениям (посредством `alpha`).

Реальная мощь sort состоит в способности сортировать на основании объектов по ссылкам. Ранее мы видели, что списки, множества и упорядоченные множества часто используются для указания ссылок на другие объекты Redis. Команда sort может получать значения по этим ссылкам и сортировать их. Например, у нас есть система отслеживания ошибок (баг-трекер), позволяющая пользователям следить за статусами ошибок. Мы можем использовать множество для поддержания списка ошибок, за которыми ведется наблюдение:

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

Удобно будет сортировать по идентификатору (что по умолчанию и произойдет), но нам также хотелось бы сортировать по критичности ошибки. Чтобы это сделать, мы указываем Redis, по какому шаблону сортировать. Сначала, давайте добавим больше данных, чтобы увидеть значимые результаты:

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

Чтобы отсортировать ошибки по критичности от более важных к менее важным, введите следующее:

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

Redis подставит значения из нашего списка (а также множества или упорядоченного множества) вместо * в указанном шаблоне. Это создаст имена ключей, по которым Redis будет запрашивать значения для сортировки.

Хотя вы можете иметь миллионы ключей в Redis, я думаю, что данные в примере выше могут стать несколько нагроможденными. К счастью, сортировка может также работать на хешах и их полях. Вместо того, чтобы иметь кучу ключей верхнего уровня, вы можете использовать хеши:

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

Теперь все не только лучше организовано, и мы имеем возможность сортировать по параметрам `severity` (критичность) и `priority` (приоритет), но также можем указать команде `sort`, значения каких полей извлекать:

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

Подстановка значений осталась прежней, но теперь Redis также распознает последовательности `->` и будет использовать их для обращения к определенному полю хеша. Мы также включили параметр `get`, который используется в подстановке и поиске значения поля, для извлечения подробных сведений (details) об ошибке.

На больших множествах `sort` может быть медленной. К счастью, возвращаемое командой `sort` значение может быть сохранено:

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

Комбинирование возможности сохранения (параметр `store`) команды `sort` с указанием срока существования, который обсуждался выше, будет отличным решением.

В Этой Главе

Эта глава посвящена командам, не связанным со структурами данных. Как и везде, их использование ситуативно. Не так уж редки случаи построения приложений, не использующих указание срока существования, публикации сообщений и подписки, сортировки. Но полезно знать, что такие возможности существуют. Кроме того, мы рассмотрели лишь несколько команд. Кроме них есть и другие, и когда вы усвоите материал этой книги, имеет смысл ознакомиться с полным списком.

Глава 5 - Администрирование

Наша последняя глава посвящена некоторым аспектам администрирования Redis. Она ни в коем случае не является полным руководством. В лучшем случае мы осветим некоторые базовые вопросы, с которыми сталкиваются новые пользователи Redis.

Конфигурирование

Когда вы впервые запустили сервер Redis, он предупредил вас, что файл `redis.conf` не может быть найден. Этот файл может быть использован для настройки различных параметров Redis. Хорошо документированный файл `redis.conf` доступен для каждой release-версии Redis. Образец файла содержит стандартные варианты конфигурации, что очень полезно для понимания того, для чего предназначены конкретные опции и каковы их значения по умолчанию. Вы можете найти его на https://github.com/antirez/redis/raw/2.4.6/redis.conf.

Это конфигурационный файл для Redis 2.4.6. Вы должны заменить «2.4.6» в приведенном выше URL на номер вашей версии. Вы можете узнать вашу версию, выполнив команду `info`.

Конфигурационный файл содержит множество комментариев, поэтому мы не будем рассматривать настройки.

Кроме конфигурирования Redis через файл `redis.conf`, команда `config set` может быть использована для задания конкретных опций. Мы уже использовали ее, когда устанавливали параметр `slowlog-log-slower-than` в 0.

Существует также специальная команда `config get`, которая выводит значения параметров конфигурации. Эта команда поддерживает поиск по образцу. Мы можем посмотреть все опции, касающиеся журналирования следующим способом:

```
config get *log*
```

Авторизация

Redis можно настроить так, чтобы он требовал пароль. За это отвечает специальная опция конфигурации `requirepass` (устанавливается через `redis.conf` или командой `config set`). Когда опция `requirepass` устанавливается в какое-либо значение (которое является паролем), клиенты должны выполнять команду `auth password` при обращении к серверу.

Как только клиент проходит авторизацию, он может выполнять команды в любой базе данных. В том числе команду `flushall`, которая удаляет все ключи из всех баз. У вас есть возможность переименовать команды для обеспечения определенного уровня защиты:

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

Вы также можете полностью отключить команду, используя в качестве ее нового названия пустую строку.

Ограничения Размеров

Когда вы начнете использовать Redis, у вас может возникнуть вопрос: «Как много ключей я могу использовать?». Кроме этого, вас могут интересовать ограничения на количество полей в хешах (особенно, когда вы используете их для организации ваших данных) и количество элементов в списках и множествах. Для одного узла, лимиты представляют собой числа порядка сотен миллионов.

Репликация

Redis поддерживает репликацию, которая означает, что все данные, которые попадают на один узел Redis (который называется master) будут попадать также и на другие узлы (называются slave). Для конфигурирования slave-узлов можно изменить опцию `slaveof` или выполнить аналогичную по написанию команду (узлы, запущенные без подобных опций являются master-узлами).

Репликация помогает защитить ваши данные, копируя их на другие сервера. Репликация также может быть использована для увеличения производительности, т.к. запросы на чтение могут обслуживаться slave-узлами. Эти узлы могу ответить слегка устаревшими данными, но для большинства приложений это приемлемо.

К сожалению, система репликации Redis еще не поддерживает автоматическую отказоустойчивость. Если master-узел выходит из строя, необходимо вручную выбрать новый из slave-узлов. Необходимо использовать традиционные утилиты, использующие мониторинг и специальные скрипты для переключения master-узлов, если вам необходима устойчивая к сбоям система.

Резервное Копирование Данных

Резервное копирование Redis это просто копирование снимка базы данных Redis в любое место (Amazon S3, FTP, …). По умолчанию, Redis сохраняет данные в файл `dump.rdb`. В любой момент времени вы можете скопировать этот файл куда угодно.

Допускается одновременное отключение сохранения данных на диск и режима дозаписи в файл у master-узлов и возложение функции резервирования на slave-узлы. Данный шаг уменьшает нагрузку на master-узел и позволяет slave-узлам использовать более агрессивные настройки сохранения данных, не снижая общую доступность системы.

Масштабирование и Redis Cluster

Репликация является первым инструментом, который используют при росте нагрузки на систему. Некоторые команды являются более медленными, чем другие (например `sort`), и перенапрвление таких запросов на slave-узлы позволяет сохранять высокую доступность системы.

Помимо этого, по-настоящему масштабируемый Redis использует распределение ключей между несколькими узлами Redis (которые могут быть запущены в той же системе - как мы помним, Redis однопоточный). В настоящее время, это то, о чем приходится заботится нам - разработчикам (хотя ряд драйверов предоставляют алгоритмы консистентного хеширования). (*Консистентное хеширование - способ создания распределенных хеш-таблиц, при котором вывод из строя одного или более серверов-хранилищ не приводит к необходимости полного переразмещения всех хранимых ключей и значений - прим. перев.*) Мы не можем рассмотреть в этой книге вопросы, касающиеся работы с данными при использовании горизонтального масштабирования. Тем более, вам скорее всего не придется беспокоится о подобных вещах, однако стоит быть в курсе независимо от того, какое решение вы используете.

Хорошей новостью является то, что ведется работа над Redis Cluster. Это решение предоставит не только возможность горизонтального масштабирования, включая автоматическую балансировку нагрузки, но также решения в плане отказоустойчивости и высокой доступности.

Высокая отказоустойчивость и масштабируемость может быть достигнута уже сегодня, но вам придется вложить некоторое количество времени и усилий в это. В конечном счете, Redis Cluster должен сделать эти вещи гораздо более простыми.

В Этой Главе

Учитывая количество проектов, уже использующих Redis, не может быть никаких сомнений, что эта система является пригодной для практического использования в настоящий момент и уже была таковой некоторое время. Однако, некоторые инструменты, особенно касающиеся поддержания безопасности и доступности, все еще требуют доработки. Redis Cluster, который мы надеемся увидеть в ближайшее время, должен помочь решить некоторые из существующих проблем администрирования.

Заключение

Redis использует множество способов облегчить наше взаимодействие с данными. Эта система убирает большую часть сложности и абстракции, присущих другим системам. Во многих случаях это делает Redis неудачным выбором. Но в других случаях может показаться, что Redis был написан для решения вашей конкретной проблемы.

В конечном итоге мы вернулись к тому, о чем я говорил в самом начале: Redis прост в освоении. Когда вокруг так много новых технологий, не просто понять, на изучение какой стоит потратить время. Когда вы увидите реальные преимущества, которые Redis может предложить, я уверен, что у вас не останется вопросов о том, уделять ли время на изучение Redis.