Armazenando dados com

Redis

Casa do Código

RODRIGO LAZOTI

À minha amada esposa e querido filho.

Agradecimentos

"Your time is limited, so don't waste it living someone else's life. Don't be trapped by dogma — which is living with the results of other people's thinking. Don't let the noise of others' opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition. They somehow already know what you truly want to become. Everything else is secondary."
– Steve Jobs

Escrever um livro é sempre uma tarefa longa e difícil, pois temos que aplicar muito esforço e tempo. Por este motivo tenho muito a agradecer às pessoas que me ajudaram direta ou indiretamente ao longo dessa jornada.

Em primeiro lugar agradeço à minha esposa Andressa pelo seu apoio incondicional e pelo carinho durante as diversas horas que passei escrevendo o livro. Ao meu filho Rafael pela sua compreensão e paciência, à minha mãe e irmãos por todo o seu apoio.

Agradeço também a todo o pessoal da Casa do Código pela orientação e ajuda, principalmente ao Paulo Silveira e o Adriano Almeida por acreditar, apoiar e me acompanhar no decorrer do meu trabalho nesse livro.

E principalmente agradeço a você leitor por investir seu precioso tempo nesse livro.

Obrigado a todos vocês!

Sobre o autor

Brasileiro, nascido no estado de São Paulo.

Rodrigo Lazoti é bacharel em Sistemas de informação pela Faculdade Drummond e Pós-graduado em Engenharia de Software pela PUC-MG. Trabalha como desenvolvedor de software desde 2002 e desde então vem utilizando diversas linguagens de programação e diferentes tecnologias.

Sumário

Capítulo 1

Começando com o Redis

Desde que comecei a estudar e trabalhar com desenvolvimento de software, sempre considerei que a forma mais fácil e rápida para aprender uma nova tecnologia é utilizando-a. Por este motivo, a intenção foi escrever um livro prático e nada mais justo do que começar demonstrando como instalar o Redis.

Em geral a instalação do Redis é uma tarefa bem simples para quem utiliza sistemas operacionais baseados no Unix, porém, para quem utiliza Windows, esta tarefa pode não ser tão simples ou até mesmo recomendada.

Este livro foi escrito utilizando a versão 2.8.x do Redis.

1.1 Instalando no Unix, Linux e Mac OS

Para sistemas operacionais baseados no Unix, como Linux e Mac OS, a instalação pode ser feita a partir do código-fonte do Redis que está disponível

em:

http://redis.io/download

Uma outra opção seria realizar a instalação por um gerenciador de pacotes, como o **HomeBrew** (http://brew.sh) do Mac OS ou o **apt-get** do Debian e seus derivados. Mas nos exemplos a seguir vou utilizar a instalação através do código-fonte, pois isso pode ser realizado facilmente por qualquer sistema operacional baseado no Unix.

Veja a seguir como fazer a instalação a partir do código-fonte:

```
wget http://download.redis.io/releases/redis-2.8.x.tar.gz
tar zxvf redis-2.8.x.tar.gz
cd redis-2.8.x
make
```

No bloco de comandos anterior, o download do Redis é feito utilizando o programa de linha de comando `wget`. O `x` é o número do release, lembre-se de substituí-lo. Assim que o download estiver concluído, o arquivo é descompactado com o comando `tar` na pasta `redis-2.8.x` e, após a sua descompactação, é necessário realizar a compilação do código-fonte do Redis utilizando o comando `make`. A compilação pode demorar um pouco para terminar, mas após o fim da execução desse comando, o Redis já esta pronto para o uso e os arquivos binários gerados estarão acessíveis através da pasta `src`.

O programa `wget` não existe nativamente no Mac OS e para resolver isso podemos substituí-lo pelo programa `curl`, que consegue fazer o download de um arquivo pela linha de comando da seguinte forma:

```
curl http://download.redis.io/releases/redis-2.8.x.tar.gz
   -o redis-2.8.x.tar.gz
```

1.2 Instalando no Windows

O Redis não suporta oficialmente o sistema operacional Windows, mas é possível instalar uma versão experimental baseada na versão 2.6 do Redis, mantida pelo grupo *Microsoft Open Tech*. Por se tratar de uma versão experimental, é importante deixar claro que até o momento em que este livro foi escrito a sua instalação é recomendada apenas em ambiente de desenvolvimento.

A versão mantida pelo *Microsoft Open Tech* pode ser obtida através do link https://github.com/MSOpenTech/redis mas, para simplificar, vamos realizar o download de um arquivo compactado do repositório do projeto através do link:

https://github.com/MSOpenTech/redis/archive/2.6.zip

Após o download do arquivo, vamos descompactá-lo na pasta `redis-2.6`, dentro da qual existe uma pasta `bin`. Esta última contém dois arquivos no formato `zip`: o arquivo `redisbin.zip` e o `redisbin64.zip`, que contém os executáveis para a plataforma 32 e 64 bits do Windows, respectivamente. Depois de selecionado o arquivo correspondente à sua plataforma, basta descompactá-lo.

1.3 Iniciando o Redis

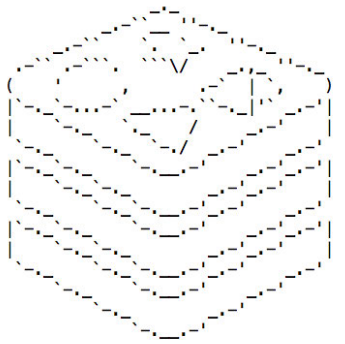
Iniciar o Redis utilizando sua configuração padrão é uma tarefa extremamente simples e rápida. Para os sistemas operacionais Unix, Linux e Mac OS, acesse a pasta `src`, que contém os arquivos binários gerados pelo comando `make`, e execute:

```
./redis-server
```

Para o sistema operacional Windows, execute o arquivo `redis-server.exe` que foi extraído do arquivo `redisbin.zip` ou `redisbin64.zip` conforme a plataforma escolhida.

Independente do sistema operacional utilizado, o Redis irá exibir uma saída bem parecida com a demonstrada a seguir:

```
[58086] 29 Mar 15:36:05.058 # Warning: no config file specified, using the default config.
In order to specify a config file use src/redis-server /path/to/redis.conf
[58086] 29 Mar 15:36:05.059 * Max number of open files set to 10032
```



```
Redis 2.8.7 (00000000/0) 64 bit

Running in stand alone mode
Port: 6379
PID: 58086


      http://redis.io
```

```
[58086] 29 Mar 15:36:05.060 # Server started, Redis version 2.8.7
[58086] 29 Mar 15:36:05.079 * DB loaded from disk: 0.019 seconds
[58086] 29 Mar 15:36:05.079 * The server is now ready to accept connections on port 6379
```

Figura 1.1: Saída do servidor Redis.

Não se preocupe com as mensagens geradas pelo Redis nesse momento, o importante agora é que o servidor do Redis está funcionando de forma adequada.

1.4 Olá Redis

Agora que já temos um servidor do Redis sendo executado, chegou o momento de realizarmos o famoso "Olá Mundo", que neste exemplo será um "ola redis!". Novamente, não se preocupe com os comandos utilizados a seguir, nem mesmo com os conceitos do Redis, pois tudo isso será explicado no decorrer do livro.

Precisamos utilizar uma aplicação cliente que consiga conectar a esse servidor do Redis. A distribuição do Redis já nos fornece um cliente de conexão, sendo que esse cliente é a forma mais simples e direta de interagir com o servidor do Redis. Ele é conhecido como **CLI**, que na língua inglesa significa *command-line interface*. A partir de agora, sempre que você encontrar a sigla CLI, lembre-se que é uma referência ao cliente nativo para linha de comando do Redis.

O CLI encontra-se na mesma pasta onde o comando `redis-server` foi executado. Ele é um arquivo chamado `redis-cli` nas plataformas baseadas no Unix e `redis-cli.exe` na plataforma Windows.

Ao executar o CLI, será apresentada a seguinte interface:

```
rodrigolazoti@MacBook-Pro:~/redis-2.8.x/src => ./redis-cli
redis 127.0.0.1:6379>
```

Agora chegou o momento de utilizarmos o Redis. Vamos iniciar usando o comando `ECHO`, que serve apenas para retornar uma mensagem enviada para ele através desse comando. Veja a seguir como fazer isso:

```
redis 127.0.0.1:6379> ECHO "ola redis!"
"ola redis!"
```

Simples, não? O comando `ECHO` apenas apresentou a mensagem que passamos para ele como parâmetro. Agora vamos fazer algo um pouco mais prático. Imagine que tenhamos que armazenar e exibir o nome do ultimo usuário logado em um sistema em tempo real. Parece uma tarefa complicada, mas esse é o tipo de tarefa em que o uso do Redis se encaixa perfeitamente. Para realizar isso, vou utilizar o comando `SET`, que recebe dois argumentos: uma chave e um valor. Veja o seguinte exemplo:

```
redis 127.0.0.1:6379> SET ultimo_usuario_logado "Bruce Banner"
OK
```

No comando anterior, a chave usada foi `ultimo_usuario_logado`, cujo valor definimos como "Bruce Banner". Repare que, ao executar o comando, o CLI retornou o status "OK", informando que o comando foi executado corretamente. Agora vamos resgatar o nome do último usuário logado para que possamos apresentá-lo no sistema. Para ler o valor armazenado nessa chave, utilizaremos o comando `GET` conforme o exemplo a seguir:

```
redis 127.0.0.1:6379> GET ultimo_usuario_logado
"Bruce Banner"
```

Para um último exemplo, imagine que o usuário acabou de sair do sistema e temos que removê-lo do Redis. Para isso, vamos utilizar o comando `DEL` que serve para remover um valor de acordo com a chave que ele recebe como parâmetro. Vamos executar o comando da seguinte forma:

```
redis 127.0.0.1:6379> DEL ultimo_usuario_logado
(integer) 1
```

Dessa vez, o CLI nos retornou o número **1**, que se refere à quantidade de chaves que foram removidas do Redis. Caso nenhuma chave tivesse sido removida, o CLI teria retornado o valor **0**.

Parabéns, você acabou de ter seu primeiro contato com o Redis e de entender como ele utiliza o conceito de armazenamento de dados no formato de chave-valor. No decorrer do livro, irei utilizar problemas que enfrentamos no dia a dia de desenvolvimento de software para exemplificar os recursos do Redis abordados aqui.

1.5 Próximos passos

Agora já temos nosso servidor instalado. Aproveitamos e testamos a configuração, adicionando usuários ao banco de dados através do cliente da linha de comando. Claro que, quando estivermos trabalhando com uma aplicação, implementada em alguma linguagem de programação, vamos precisar fazer toda essa comunicação com o banco através da linguagem usada.

No próximo capítulo, vamos aprender como fazer isso através do Java e começar a trabalhar com o Redis através da linguagem de programação, para conseguirmos criar programas interessantes.

Capítulo 2

Conhecendo o Redis

Redis significa ***RE**mote **DI**ctionary **S**erver*. Diferente de um banco de dados tradicional como *MySQL* ou *Oracle*, é categorizado como um banco de dados não relacional, sendo muitas vezes referenciado pela sigla *NOSQL* (Not Only SQL). O Redis foi criado por Salvatore Sanfilippo [16], também conhecido na internet por `antirez`, que liberou o Redis em 2009 de forma open-source sob a licença BSD [13].

Uma característica muito importante sobre o Redis é que ele armazena seus dados em memória, embora seja possível persistir os dados fisicamente. Mas é o fato de o Redis armazenar os dados em memória que o torna extremamente rápido, tanto para escrita como para leitura de dados. Uma outra característica importante é que todos os comandos executados no Redis são atômicos, e isso é garantido pela forma com que o Redis é executado, que é como uma aplicação *single-threaded* (enquanto um comando está sendo executado, nenhum outro comando será executado) [6].

Como vimos no capítulo 1, o Redis armazena os dados na forma de chave-valor, mas um ponto interessante sobre a estrutura de dados do Redis é que o valor contido na chave de um registro suporta diferentes formatos que podem ser strings, hashes, lists, sets e sets ordenados. Todos esses formatos (também conhecidos como estruturas de dados) que acabei de apresentar serão demonstrados no decorrer do livro.

O Redis é um servidor TCP que faz uso do modelo cliente-servidor [11]. Isso significa que em geral uma requisição feita por um cliente ao servidor é seguida das seguintes etapas:

- O cliente envia um comando ao servidor e fica aguardando uma resposta do servidor (geralmente bloqueando a conexão) através de uma conexão estabelecida via socket;
- O servidor processa o comando e envia a resposta de volta ao cliente.

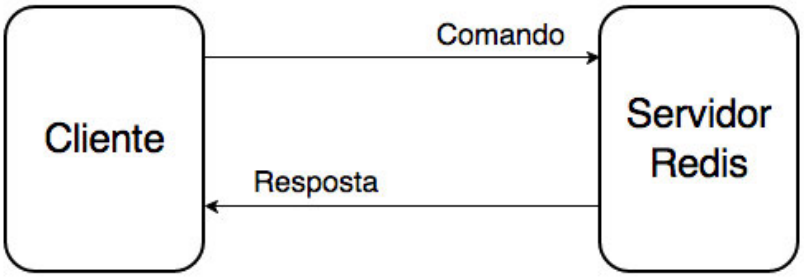


Figura 2.1: Processo de envio de comando e recebimento da resposta

2.1 O que o Redis não é

Assim como é muito importante entender o que é o Redis e o que podemos fazer com ele para podermos utilizá-lo de forma correta, é essencial entender também o que o Redis não é e o que não é possível. Veja a seguir uma pequena lista de itens que o Redis não é ou não faz:

- Não é um banco de dados relacional como o MySQL ou Oracle;
- Não é um banco de dados orientado a documentos como o MongoDB;
- Não é um banco de dados que você deveria usar para armazenar todos os seus dados;
- Não possui suporte **oficial** ao Windows;
- Não utiliza o protocolo HTTP.

2.2 Indo além do CLI

O CLI (a interface de linha de comando) é uma forma rápida e fácil de executar comandos no Redis mas, na prática, a comunicação com o Redis é feita muitas vezes através de uma aplicação, e não utilizando o CLI diretamente. Para resolver isso, o Redis possui diversos clientes para várias linguagens de programação que vão desde Java até Smalltalk e que funcionam de forma síncrona ou assíncrona. Para conferir todos os clientes disponíveis para cada linguagem de programação, acesse o link:

http://redis.io/clients

Essa página contém todos os clientes para Redis, suas respectivas descrições e endereços, mas um ponto importante é que também é possível ver uma classificação de qual cliente é mais recomendado e mais ativo para cada linguagem de programação.

No decorrer do livro, irei utilizar alguns exemplos em CLI e outros em Java utilizando a versão 2.4.2 do cliente Jedis que está disponível através do link:

https://github.com/xetorthio/jedis

Saiba que tudo que for feito utilizando um cliente Java ou CLI diretamente pode ser feito com qualquer outro cliente de Redis, independente da linguagem de programação. O código será muito análogo, seja em Ruby, Python ou sua linguagem preferida! O que importa realmente são os conceitos.

O código-fonte em Java dos exemplos do livro serão feitos utilizando o Maven. Para quem preferir utilizá-lo, a dependência do Jedis pode ser declarada da seguinte forma:

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.4.2</version>
</dependency>
```

2.3 Olá Redis em Java

Para entendermos como usar o Redis através de uma aplicação, vamos realizar o mesmo exemplo feito no capítulo 1, mas agora usando um cliente em Java chamado Jedis. Vamos começar com o comando `ECHO`:

```
Jedis jedis = new Jedis("localhost");
String resultado = jedis.echo("ola redis!");
System.out.println(resultado);
```

Uma instância da classe `Jedis` é tudo que precisamos para enviar comandos para o Redis através de uma aplicação Java. No exemplo anterior, ao instanciar a classe Jedis, passamos como parâmetro ao construtor o local onde o Redis está sendo executado — que no nosso caso é na máquina local (localhost). A biblioteca Jedis utiliza a mesma nomenclatura dos comandos disponíveis pelo CLI. Ao executar o método `echo`, a instância do Jedis retorna o mesmo resultado de quando executamos o comando `ECHO` via CLI, conforme o bloco a seguir:

```
ola redis!
```

Agora vamos armazenar uma informação utilizando o comando `SET` conforme o código a seguir:

```
Jedis jedis = new Jedis("localhost");
String resultado = jedis.set("ultimo_usuario_logado",
                             "Tony Stark");
System.out.println(resultado);
```

Caso tudo ocorra corretamente, o valor impresso pela variável `resultado` será **OK**. Repare que, embora estejamos enviando comandos para o Redis através de uma aplicação Java, o resultado é o mesmo que recebemos quando utilizamos o CLI.

Vamos executar o comando `GET` através do Jedis para confirmar que o valor correto da chave `ultimo_usuario_logado` está armazenado no Redis. Veja como fazer isso com o seguinte código:

```
Jedis jedis = new Jedis("localhost");
String valor = jedis.get("ultimo_usuario_logado");
System.out.println(valor);
```

O resultado do código anterior é:

```
Tony Stark
```

Repare que executar comandos no Redis via Java usando o cliente Jedis é uma tarefa simples e bem próxima da forma como é feito pelo CLI. Para encerrar este primeiro contato com o Jedis, vamos remover a chave que consultamos no exemplo anterior:

```
Jedis jedis = new Jedis("localhost");
Long resultado = jedis.del("ultimo_usuario_logado");
System.out.println(resultado);
```

Assim como o CLI, o método `del` retorna o número **1**, que refere-se à quantidade de chaves que foram removidas do Redis. Caso nenhuma chave tivesse sido removida o método teria retornado o valor **0**.

2.4 Testando o Redis online

Caso você queira apenas testar alguns comandos no Redis, mas por algum motivo não tenha a possibilidade de instalá-lo no momento, não se preocupe: você pode utilizar uma versão online e interativa que permite que você envie comandos para o Redis através de uma aplicação web.

Essa aplicação se chama **Try Redis** e possui também um pequeno tutorial de introdução ao Redis bem divertido. Caso tenha interesse ou curiosidade de testá-lo, basta acessar o link a seguir:

http://try.redis.io

Essa aplicação web é desenvolvida em Ruby e utiliza um cliente para o Redis chamado *redis-rb*, que é o mais popular e utilizado em Ruby. Todo

o código-fonte da aplicação **Try Redis** está disponível no Github através do endereço:

https://github.com/badboy/try.redis

2.5 Recursos do livro

Todos os exemplos em Java apresentados nesse livro estão disponíveis em um repositório público no Github e você pode acessá-lo aqui:

https://github.com/rlazoti/exemplos-livro-redis

Além disso, existe um grupo de discussão específico para este livro onde você pode enviar suas dúvidas, sugestões, críticas e conversar comigo e com outros leitores sobre tudo que foi abordado neste livro e sobre o Redis em si. Você pode acessá-lo através do link

https://groups.google.com/forum/#!forum/redis-casadocodigo

O Redis possui mais de cento e quarenta comandos distintos e muitos deles não se aplicam ao contexto dos exemplos que utilizarei no decorrer do livro, sendo assim alguns comandos não serão abordados. Eu irei oferecer uma breve explicação para alguns comandos e seus exemplos estarão disponíveis em forma de exercícios para que o leitor possa testar estes comandos e utilizá-los quando necessário.

Deixo ao seu cargo realizar os exercícios para assimilar o uso desses comandos, sendo que você pode resolver os exercícios pelo CLI (interface de linha de comando) ou através da linguagem com a qual você tiver mais familiaridade. Eu deixarei os códigos em Java contendo os exemplos do livro disponíveis no Github.

Para obter mais informações e exemplos de uso sobre todos os comandos disponíveis no Redis, você pode acessar sua documentação:

http://redis.io/commands

2.6 Próximos passos

Já avançamos mais um pouco, aprendemos a enviar comandos para o Redis utilizando a linguagem Java e também a realizar pequenos testes online. Além disso, já temos à disposição todo o código de exemplo utilizado no livro e um

fórum para conversar com outros leitores sobre tudo que está sendo abordado no livro.

No próximo capítulo, vamos aprender a utilizar mais alguns comandos no Redis com exemplos práticos que podemos aplicar no dia a dia e conhecer dois tipos de dados suportados pelos Redis: *String* e *Hash*.

Capítulo 3

Redis no mundo real — Parte 1

Este livro poderia apresentar uma série de comandos do Redis e suas respectivas descrições, mas isso tornaria a leitura cansativa e com baixo aproveitamento e absorção de conteúdo. Para tentar tornar o aprendizado mais prazeroso e até mesmo mais próximo do nosso cotidiano, resolvi pesquisar e criar diversos exemplos em que o uso do Redis se encaixa para que assim consiga demonstrar seu uso com problemas reais que enfrentamos no dia a dia do desenvolvimento de software.

3.1 Cache de dados com Strings

String é o tipo de dado mais comum disponível no Redis e este tipo de dado é muito similar ao tipo string que vemos em outras linguagens de programação, como Java ou Ruby. No Redis, um valor do tipo string pode conter um tamanho de no máximo 512 Megabytes e, por ser um tipo de dado **binary**

safe, podemos armazenar por exemplo um texto, um documento JSON, objetos serializados ou até mesmo os dados binários de uma imagem [1]. Veja a seguir 3.1 uma ilustração do tipo de dado string:

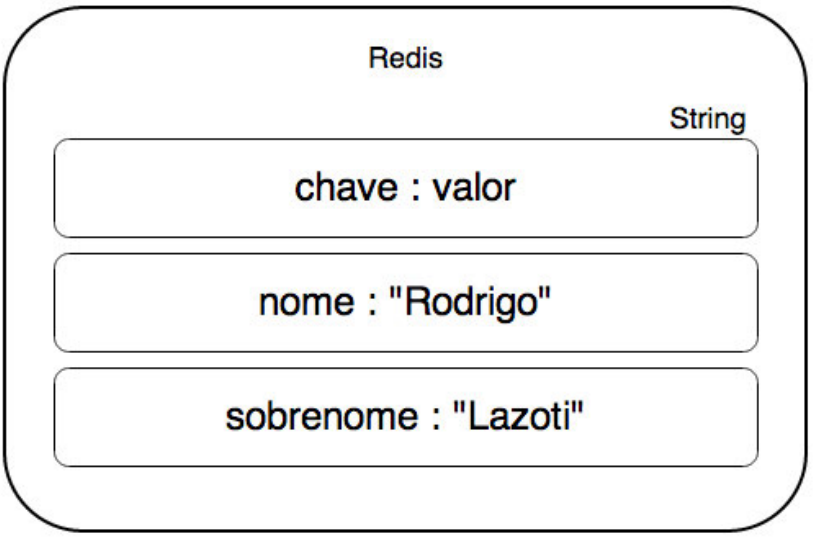


Figura 3.1: Tipo de dado String

O primeiro exemplo de uso para Redis com o tipo de dado string é como um repositório de dados em memória que, conforme seu propósito, é conhecido também como **Cache**. O armazenamento de dados em cache é válido quando existem dados que são utilizados com muita frequência e que não sofrem atualizações constantemente, poupando assim tempo e uso desnecessário do hardware. Por exemplo, imagine que o site chamado **Resultado de Loterias** tenha que exibir na sua página inicial os números do ultimo resultado da Mega-Sena, mas esses dados são fornecidos por uma outra empresa através de um serviço web.

O site **Resultados de Loterias** poderá ter um grande problema caso esse serviço externo fique inacessível ou até mesmo um impacto negativo no tempo de resposta caso tenha que ir buscar os dados do sorteio nesse serviço

web externo a cada usuário que acessar o site.

Caso você não conheça muito sobre jogos de loteria, os sorteios da Mega-Sena em geral ocorrem duas vezes por semana, sendo um sorteio na quarta-feira e outro no sábado. Embora as informações fornecidas pelo serviço web da outra empresa sejam dinâmicas, elas são alteradas apenas duas vezes em um período de uma semana.

Para melhorar o tempo de resposta do site **Resultados de Loterias** e não fazer consultas desnecessárias ao serviço web da outra empresa, podemos fazer uma única requisição desses dados no serviço web externo e depois armazená-los no Redis como um cache interno até que o próximo sorteio seja realizado para, assim, o site novamente buscar através do serviço web os novos dados.

Para este exemplo, vamos definir que o serviço externo retorne um JSON conforme o seguinte exemplo:

```
{
  "data": "21-09-2013",
  "numeros": [2, 13, 24, 41, 42, 44],
  "ganhadores": 2
}
```

> **Manipular JSON em Java**
>
> Vou omitir a parte de como fazer a conversão de um JSON para um objeto Java, mas poderíamos utilizar, por exemplo, a biblioteca **GSON** para esse propósito. Você pode obter mais informações sobre essa biblioteca através do link:
> https://code.google.com/p/google-gson/

Neste primeiro exemplo, vamos utilizar apenas os números do sorteio obtidos pelo JSON anterior. Vamos ver como poderíamos fazer para armazenar esses números através de uma aplicação Java:

```
String chave = "resultado:megasena";
String numerosDoUltimoSorteio = "2, 13, 24, 41, 42, 44";
```

```
Jedis jedis = new Jedis("localhost");
String resultado = jedis.set(chave, numerosDoUltimoSorteio);
System.out.println(resultado);
```

Até aqui não vimos nada muito diferente do que fizemos no capítulo 1 ou 2. Utilizamos o comando **SET** para armazenar os números em uma chave chamada *resultado_megasena* e também podemos utilizar o comando `GET` para obter o números armazenados nesta mesma chave.

3.2 Encontrando as chaves armazenadas

Continuando com o exemplo do nosso site **Resultados de Loterias**, além de exibirmos o último sorteio realizado na página inicial, temos também que ter uma página com o histórico dos resultados anteriores. Nesta página, será possível filtrar os resultados por mês e ano do sorteio, mas para isso temos que mudar um pouco a estrutura da chave que utilizamos no exemplo anterior.

Antes, nossa chave era definida como `resultado:megasena`, mas agora precisamos armazenar vários resultados e para conseguirmos isso precisamos melhorar a estrutura da nossa chave. A forma mais prática é inserir a data do sorteio na chave para que ela fique da seguinte forma: `resultado:dd-mm-yyyy:megasena`, onde `dd` é o dia, `mm` o mês e `yyyy` o ano do sorteio. Assim conseguimos ter uma chave para cada sorteio.

Convenção para nomear chaves no Redis

Utilizar : para compor um "namespace" na chave é uma convenção muito utilizada no Redis, sendo que um formato de chave muito comum assemelha-se com **tipo-de-objeto:identificador:nome-campo**. Por exemplo, imagine uma chave utilizando esse formato que represente o nome dos usuários de um sistema. Essa chave poderia ser da seguinte forma:

`usuario:Rodrigo Lazoti:nome`

Sendo que `usuario` é o tipo de objeto, o valor `Rodrigo Lazoti` representa o nome do usuário e `nome` é o nome do campo que dá significado ao valor armazenado nesta chave.

Vamos criar um novo exemplo que irá inserir quatro sorteios no Redis utilizando a nova composição de chave que foi definida. Veja o exemplo a seguir:

```
String dataDoSorteio1 = "04-09-2013";
String numerosDoSorteio1 = "10, 11, 18, 42, 55, 56";
String chave1 = String.format("resultado:%s:megasena",
                              dataDoSorteio1);

String dataDoSorteio2 = "07-09-2013";
String numerosDoSorteio2 = "2, 21, 30, 35, 45, 50";
String chave2 = String.format("resultado:%s:megasena",
                                dataDoSorteio2);

String dataDoSorteio3 = "21-09-2013";
String numerosDoSorteio3 = "2, 13, 24, 41, 42, 44";
String chave3 = String.format("resultado:%s:megasena",
                                dataDoSorteio3);

String dataDoSorteio4 = "02-10-2013";
String numerosDoSorteio4 = "7, 15, 20, 23, 30, 41";
String chave4 = String.format("resultado:%s:megasena",
```

```
                                    dataDoSorteio4);

Jedis jedis = new Jedis("localhost");
String resultado = jedis.mset(
    chave1, numerosDoSorteio1,
    chave2, numerosDoSorteio2,
    chave3, numerosDoSorteio3,
    chave4, numerosDoSorteio4
);

System.out.println(resultado);
```

Neste exemplo, foi usado o comando `MSET`, enquanto no exemplo anterior usamos o comando `SET`. A única diferença entre o `SET` e o `MSET` é que o `MSET` aceita vários conjuntos de chave-valor como parâmetro enquanto o `SET` aceita apenas um único conjunto de chave-valor.

Se o comando anterior funcionou da forma esperada, o resultado apresentado pelo exemplo será um **OK**. Agora já temos quatro sorteios para compor nossa página de histórico, mas ainda precisamos criar uma forma de filtrar esses itens. O Redis possui um comando chamado `KEYS`, que é usado para fazer buscar de chaves com base em um determinado padrão (*pattern*) que é passado como parâmetro para o comando.

Esse padrão (pattern) passado como parâmetro para o comando `KEYS` utiliza o *Glob-style pattern matching* [3]. Para exemplificar seu uso, imagine que tenhamos as chaves `bala`, `bela`, `bola` e `bicicleta` armazenadas. Confira a seguir alguns exemplos de como usar esse *pattern* para buscar determinadas chaves:

- O caractere `*` representa um conjunto de caracteres que podem ser zero ou mais caracteres. Exemplo: `b*a` encontraria `bala`, `bela`, `bola` e `bicicleta`;
- O caractere `?` representa um único caractere. Exemplo: `b?la` encontraria `bala`, `bela` e `bola`;
- Colchetes `[]` representam um grupo de caracteres. Exemplo: `b[ae]la` encontraria `bala` e `bela`.

Por exemplo, veja como podemos usar esse comando para obter todas as chaves armazenadas no Redis independente de seu tipo, conforme o exemplo a seguir feito no CLI:

```
redis 127.0.0.1:6379> KEYS *
1) "resultado:04-09-2013:megasena"
2) "resultado:07-09-2013:megasena"
3) "resultado:megasena"
4) "resultado:21-09-2013:megasena"
5) "resultado:02-10-2013:megasena"
```

Quando usamos o caractere `*` sozinho como pattern para o comando `KEYS`, estamos dizendo para o Redis que queremos todas as chaves armazenadas.

Ainda no CLI, vamos ver como podemos obter todas as chaves que utilizam o nosso novo padrão `resultado:dd-mm-yyyy:megasena`. Mas além disso, vamos obter apenas as chaves cujo dia esteja entre 01 e 09:

```
redis 127.0.0.1:6379> KEYS resultado:0?-*-*:megasena
1) "resultado:04-09-2013:megasena"
2) "resultado:07-09-2013:megasena"
3) "resultado:02-10-2013:megasena"
```

Esse mesmo pattern poderia ser escrito de outras formas, conforme a lista apresentada a seguir, mas ainda assim também produziram o mesmo resultado apresentado anteriormente.

```
KEYS resultado:0?-??-????:megasena
KEYS resultado:0*-??-????:megasena
KEYS resultado:0?-*-????:megasena
KEYS resultado:0*-*-????:megasena
KEYS resultado:0?-??-*:megasena
KEYS resultado:0*-??-*:megasena
KEYS resultado:0?-*:megasena
KEYS resultado:0*-*:megasena
```

Agora que já entendemos como utilizar o comando `KEYS` para obtermos apenas as chaves que desejamos, a tarefa de filtrar os resultados por mês e ano do sorteio ficou fácil. Dando continuidade ao nosso exemplo, vamos escrever

um método em Java que realize a tarefa de filtrar os resultados da página de histórico, conforme o exemplo a seguir:

```
public class FiltrarHistoricoDaMegaSena {

  public Set<String> filtrarResultados(int mes, int ano) {
    String chave = "resultado:*-%02d-%04d:megasena";
    Jedis jedis = new Jedis("localhost");

    return jedis.keys(String.format(chave, mes, ano));
  }

  public static void main(String[] args) {
    int mes = 10;
    int ano = 2013;
    Set<String> chaves =
      new FiltrarHistoricoDaMegaSena()
        .filtrarResultados(mes, ano);

    System.out.println(chaves);
  }
}
```

O exemplo de uso do código anterior apresentará quando executado o resultado:

```
[resultado:02-10-2013:megasena]
```

Exercícios sobre Strings

1) O comando `MGET` retorna os valores de várias chaves de uma única vez, assim como o `MSET` que utilizamos anteriormente. Utilize-o para obter todas as chaves armazenadas no Redis.

2) O comando `STRLEN` retorna o tamanho do valor associado a uma chave que ele recebe como parâmetro. Utilize-o para descobrir o tamanho do valor correspondente a chave "resultado_megasena".

3) O comando `GETRANGE` retorna um pedaço do valor associado a uma chave de acordo com uma posição inicial e uma posição final (até a versão 2.0 do Redis esse comando era chamado SUBSTR). Utilize-o para obter os dois primeiros números (2, 13) armazenados na chave "resultado_megasena".

Referência rápida de comandos para Strings

- `APPEND chave valor` — adiciona o valor a uma chave existente, ou cria uma nova chave (caso esta ainda não exista) com seu respectivo valor;
- `DEL chave [chave ...]` — remove a(s) chave(s) informada(s) e seu(s) respectivo(s) valor(es);
- `GET chave` — retorna o valor correspondente à chave informada;
- `GETRANGE chave inicio fim` — retorna uma parte da string armazenada conforme a chave informada;
- `MGET chave [chave ...]` — retorna os valores correspondentes às chaves informadas;
- `MSET chave valor [chave valor ...]` — armazena um ou mais conjuntos de chave valor. Caso uma chave informada já exista, seu valor será sobrescrito pelo novo;
- `SET chave valor` — armazena a chave e seu respectivo valor. Caso já exista uma chave definida, seu valor é sobrescrito;
- `STRLEN chave` — retorna o tamanho da string armazenada conforme a chave informada.

3.3 Utilizando hashes

No Redis, um hash nada mais é do que um map que contém campos e valores do tipo string. Este tipo de dado é muito utilizado para representar objetos

definidos em nossas aplicações, como por exemplo um usuário que contém um nome, e-mail e data de nascimento. Cada hash pode armazenar mais de 4 bilhões de pares de campo-valor [1].

Utilizando um hash podemos definir vários conjuntos de campo-valor para uma única chave. Veja a seguir 3.2 uma ilustração do tipo de dado *Hash*:

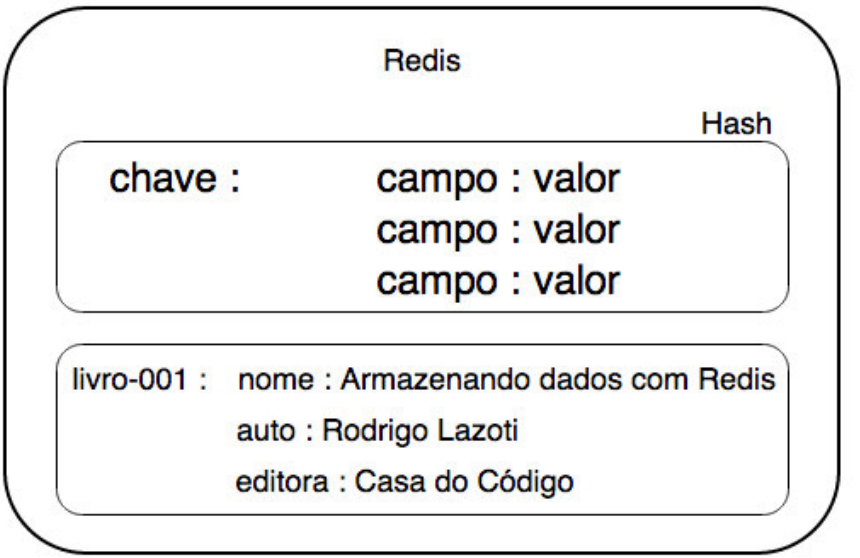


Figura 3.2: Tipo de dado Hash

Para demonstrar o uso de hashes iremos dar continuidade ao exemplo da seção anterior. O JSON que recebemos possui três valores distintos, que são a data do sorteio, os números do sorteio e a quantidade de ganhadores. Todos esses dados são correspondentes a uma única informação, que é o resultado de um sorteio da Mega-Sena.

Antes tínhamos um único valor para uma chave, mas agora temos dois valores distintos referentes a uma chave. Quando utilizamos os comandos `SET` e `GET` estávamos utilizando dados na forma de **strings**; agora vamos utilizar uma outra forma de dados suportada pelo Redis: os **hashes**.

Dando continuidade ao nosso exemplo, vamos refazer nosso código para

armazenar no Redis uma chave utilizando o mesmo formato da seção 3.2 e, como valor, os campos do número de ganhadores e dos números sorteados, da seguinte forma:

```
String ganhadores = "22";
String dataSorteio = "09-11-2013";
String numeros = "8, 18, 26, 42, 56, 58";
String chave = String.format("resultado:%s:megasena",
                             dataSorteio);

Jedis jedis = new Jedis("localhost");
long resultado1 = jedis.hset(chave, "ganhadores", ganhadores);
long resultado2 = jedis.hset(chave, "numeros", numeros);

String mensagem = String.format(
    "Resultado 1 = %d, Resultado 2 = %d",
    resultado1,
    resultado2
);

System.out.println(mensagem);
```

O resultado do código anterior é:

```
Resultado 1 = 1, Resultado 2 = 1
```

No exemplo anterior, nós definimos os campos `ganhadores` e `numeros` e seus respectivos valores para a chave `resultado:09-11-2013:megasena`. Diferente do comando `SET`, o `HSET` retorna o número 1 caso tenha armazenado o campo de forma correta, ou o caso o campo já exista ou seu valor tenha sido alterado.

Para obter os campos armazenados em um chave do tipo *hash*, utilizamos o comando `HGET` e informamos a chave e o campo cujo valor queremos obter. Veja um exemplo de como utilizá-lo:

```
String chave = "resultado:09-11-2013:megasena";
Jedis jedis = new Jedis("localhost");

String ganhadores = jedis.hget(chave, "ganhadores");
```

```
String numeros = jedis.hget(chave, "numeros");

String mensagem = String.format(
    "Ganhadores = %s, Numeros = [%s]",
    ganhadores,
    numeros
);

System.out.println(mensagem);
```

Vejamos a seguir o resultado desse código:

```
Ganhadores = 22, Numeros = [8, 18, 26, 42, 56, 58]
```

Exercícios sobre hashes

1) Podemos utilizar o comando `HDEL` para remover um campo associado a um determinado hash, utilize-o para apagar o campo `ganhadores` do hash `resultado:megasena`;

2) O comando `HEXISTS` verifica se um campo existe em um hash; utilize-o para verificar se o campo `ganhadores` existe no hash `resultado:megasena`;

3) O comando `HLEN` informa a quantidade de campos que estão associados a um hash; utilize-o para saber quantos campos estão associados ao hash `resultado:megasena`;

Referência rápida de comandos para hashes

- `HDEL` *`chave campo [campo ...]`* — remove o(s) campo(s) e seu(s) respectivo(s) valor(es) do hash informado;
- `HEXISTS` *`chave campo`* — determina se um hash e seu campo existem;
- `HGET` *`chave campo`* — retorna o valor do campo associado ao hash informado;

- `HLEN hash` — retorna a quantidade de campos que um hash possui;
- `HMGET chave campo [campo ...]` — retorna os valores de todos os campos informados que são associados a um hash;
- `HMSET chave campo valor [campo valor ...]` — define múltiplos campos e valores em um hash;
- `HSET chave campo valor` — armazena um hash com o campo e seu respectivo valor. Caso o hash e o campo já existam, o valor é sobrescrito.

3.4 Próximos passos

A primeira parte dos exemplos de uso do Redis abordaram o conceito de cache com *Strings*. Vimos também como realizar consultas de chaves com um determinado padrão de nomenclatura. Na sequência, vimos como utilizar *Hashes* para conseguirmos estruturas de dados mais complexas.

No próximo capítulo, vamos ver a segunda parte dos exemplos práticos e conhecer mais um pouco o que podemos fazer com o Redis de forma simples, rápida e fácil. Vamos aprender a definir um tempo de expiração para as chaves do Redis e também a efetuar incremento e decremento em números.

Capítulo 4

Redis no mundo real — Parte 2

4.1 Expirando chaves de forma automática

Você tem uma aplicação web que armazena informações nas sessões dos usuários. Inicialmente, esta aplicação estava sendo executada em apenas um servidor, mas devido à alta demanda de acesso, você precisa adicionar mais um servidor para sua aplicação. Embora você tenha solucionado o problema de demanda de acesso, agora você tem um problema com as sessões dos usuários que não estão acessíveis em ambos os servidores.

Existem diversas soluções para este problema de sessões, como replicar as sessões entre todos os servidores, ou utilizar "*sticky sessions*" para que o balanceador de carga sempre se conecte com mesmo servidor que possui a sessão do usuário. Deixando de lado os prós e contras de cada solução, vamos conhecer uma outra solução que é utilizar o Redis para armazenar e controlar a expiração desses dados de sessão sem a necessidade de replicar os dados

entre cada servidor.

Vamos iniciar estipulando que toda sessão deverá existir por no máximo 30 minutos, independente de o usuário estar realizando ações ou não na aplicação. Para compor a sessão, iremos utilizar o código do usuário, seu nome e seu e-mail. Vamos continuar utilizando o formato **hash** para armazenar essas informações no Redis, porém, dessa vez, utilizaremos o comando **HMSET** para que possamos enviar vários campos e seus respectivos valores de uma vez ao Redis. Veja o exemplo a seguir:

```
final String codigoDoUsuario = "1962";
final String nomeDoUsuario = "Peter Parker";
final String emailDoUsuario = "spidey@marvel.com";

String chave = "sessao:usuario:" + codigoDoUsuario;

Map<String, String> campos = new HashMap<String, String>() {{
  put("codigo", codigoDoUsuario);
  put("nome", nomeDoUsuario);
  put("email", emailDoUsuario);
}};

Jedis jedis = new Jedis("localhost");

String resultado = jedis.hmset(chave, campos);
System.out.println(resultado);
```

Diferente do exemplo em que utilizamos o comando `HSET` diversas vezes para armazenar todos os campos que queríamos, com o `HMSET` foi necessário usá-lo apenas uma vez, pois esse comando recebe como parâmetro um conjunto (no caso do Java é um Map) com todos as chaves e valores. Caso os dados sejam armazenados corretamente, o resultado do comando `HMSET` é um "OK".

O Redis possui um recurso muito interessante que é a possibilidade de definir um tempo de expiração para qualquer chave armazenada nele. Podemos verificar esse tempo de expiração utilizando o comando `TTL`, e para isso vamos utilizá-lo na chave que acabamos de criar usando o CLI:

```
redis 127.0.0.1:6379> TTL "sessao:usuario:1962"
(integer) -1
```

Quando o comando `TTL` retorna -1, significa que a chave não possui um tempo de expiração. Caso o comando tivesse retornado o número -2, significaria que a chave não existisse no Redis. Um dos comandos que o Redis fornece para definir um tempo de expiração de uma chave é o `EXPIRE`; utilizando-o, podemos definir em quantos segundos uma chave irá expirar. Vamos definir o tempo de expiração da sessão em 30 minutos para a sessão que criamos no código anterior, da seguinte forma:

```
String codigoDoUsuario = "1962";
String chave = "sessao:usuario:" + codigoDoUsuario;
int trintaMinutosEmSegundos = 1800;

Jedis jedis = new Jedis("localhost");

long resultado = jedis.expire(chave, trintaMinutosEmSegundos);
System.out.println(resultado);
```

O comando `EXPIRE` retorna 1 quando a definição foi realizada ou o caso a chave não exista ou o tempo especificado não possa ser definido. Agora podemos novamente utilizar o comando `TTL` para ver o tempo restante que falta para a sessão criada expirar.

```
redis 127.0.0.1:6379> TTL "sessao:usuario:1962"
(integer) 1773
```

Como vimos nos exemplos anteriores, definir um tempo de expiração para um chave no Redis é uma tarefa simples e muito útil em diversos casos em que você precisa manter um dado armazenado apenas durante um tempo.

Exercícios sobre TTL

1) O comando `PERSIST` remove um tempo de expiração para um chave. Utilize-o para remover o tempo de expiração da sessão armazenada no hash `"sessao:usuario:1962"`;

2) O comando `HMGET` serve para que possamos obter o valor de vários campos associados a um hash de uma vez. Utilize-o para obter os dados da sessão armazenada no hash `"sessao:usuario:1962"`;

3) O comando `PEXPIRE` funciona da mesma forma que o `EXPIRE`, mas o tempo de expiração que ele recebe é em milissegundos, enquanto o do `EXPIRE` é em segundos. Utilize o `PEXPIRE` para que a chave `"sessao:usuario:1962"` expire em 30 minutos (converta para milissegundos);

Referência rápida de comandos para expiração de dados

- `EXPIRE` *`chave tempo`* — define um tempo (em segundos) de expiração para uma chave;
- `PERSIST` *`chave`* — remove o tempo de expiração de uma chave;
- `PEXPIRE` *`chave tempo`* — define um tempo (em milissegundos) de expiração para uma chave;
- `PTTL` *`chave`* — retorna o tempo (em milissegundos) de vida restante para expiração da chave;
- `TTL` *`chave`* — retorna o tempo (em segundos) de vida restante para expiração da chave.

4.2 Estatísticas de páginas visitadas

Armazenar estatísticas de acesso para as páginas de um site em tempo real pode parecer uma tarefa complexa que envolveria diversos sistemas trabalhando em conjunto e um volume grande de dados armazenados a cada minuto ou até mesmo a cada segundo. Com o Redis, podemos facilmente realizar essa tarefa sem nenhum impacto em desempenho e ocupando pouco espaço para armazenamento de dados.

Para este exemplo, será definido que iremos manter as estatísticas de páginas visitadas por dia. Assim é possível saber quantas vezes cada página de

um site foi visitada a cada dia. Vamos utilizar o exemplo de um blog chamado *Tudo Sobre Redis*, que irá conter algumas páginas:

```
/inicio
/contato
/sobre-mim
/todos-os-posts
/armazenando-dados-no-redis
```

O primeiro ponto que precisamos definir é a estrutura da chave que será utilizada para armazenar os dados das estatísticas. Como nossos dados serão separados por dia, temos que utilizar a data na chave e também a página que recebeu a visita. Com essas informações podemos compor uma chave com a seguinte estrutura:

```
pagina:[url da pagina]:[data]
```

Agora vejamos um exemplo de uma chave utilizando valores reais:

```
pagina:/inicio:12-11-2013
```

Com a estrutura da chave definida, fica fácil entender como funcionará o armazenamento das estatísticas. Para isso, basta sempre incrementarmos o valor da chave a cada respectivo acesso, algo bem comum de realizar. Poderíamos fazer isso armazenando uma nova chave no Redis como o valor 1 no primeiro acesso de cada página, e posteriormente a cada acesso obter (`GET`) o valor da chave, somar 1 ao seu valor e armazená-lo novamente (`SET`).

Embora pareça um processo simples de se realizar, seria necessário executar dois comandos (`GET` e `SET`) no Redis para cada acesso de página. Um outro problema em realizar dessa forma é que isso não seria uma operação atômica, e por isso estaria sujeita a problemas de condição de corrida (*race conditions*) [7].

Eis que surge o comando `INCR`. Utilizando-o, podemos incrementar o valor de uma chave diretamente sem a necessidade de conhecer o seu valor anterior e assim tornar a operação atômica. Este comando é classificado com um comando para o tipo de dado *String*, porque no Redis não existe o tipo de dados *Integer*. Quando usamos esse comando, o Redis interpreta o valor

da chave como um número inteiro de 64 bits com sinal, que significa que a chave também suporta valores negativos.

Agora que conhecemos esse novo comando, vamos escrever uma aplicação para realizar a tarefa determinada pelo nosso exemplo:

```
public class GerarEstatisticaDePaginasVisitadas {

  public void gerarEstatistica(String pagina, String data) {
    String chave = String.format("pagina:%s:%s", pagina, data);
    Jedis jedis = new Jedis("localhost");
    long resultado = jedis.incr(chave);
    System.out.println(
        String.format(
            "página %s teve %d acesso(s) em %s",
            pagina,
            resultado,
            data
        )
    );
  }

  public static void main(String[] args) {
    String data = "02/09/2013";
    String[] paginasVisitadas = {
      "/inicio",
      "/contato",
      "/sobre-mim",
      "/todos-os-posts",
      "/armazenando-dados-no-redis"
    };

    GerarEstatisticaDePaginasVisitadas gerador =
        new GerarEstatisticaDePaginasVisitadas();

    gerador.gerarEstatistica(paginasVisitadas[0], data);
    gerador.gerarEstatistica(paginasVisitadas[1], data);
    gerador.gerarEstatistica(paginasVisitadas[2], data);
    gerador.gerarEstatistica(paginasVisitadas[1], data);
    gerador.gerarEstatistica(paginasVisitadas[1], data);
```

```
  }
}
```

O resultado do código é o seguinte:

```
página /inicio teve 1 acesso(s) em 02/09/2013
página /contato teve 1 acesso(s) em 02/09/2013
página /sobre-mim teve 1 acesso(s) em 02/09/2013
página /contato teve 2 acesso(s) em 02/09/2013
página /contato teve 3 acesso(s) em 02/09/2013
```

Simples, não? Vamos conhecer mais alguns comandos que podemos utilizar para incrementar o valor de uma chave e também decrementar seu valor. Embora eles não tenham uma aplicação prática para o contexto do exemplo utilizado aqui, é importante conhecê-los e entender seu uso.

Imagine que precisamos decrementar o valor de uma das chaves que criamos anteriormente. Isso pode ser feito de duas formas, sendo que a primeira é com o comando `DECR`, que decrementa o valor da chave em um número. Vejamos um exemplo de uso desse pelo CLI:

```
redis 127.0.0.1:6379> GET pagina:/contato:02/09/2013
"3"

redis 127.0.0.1:6379> DECR get pagina:/contato:02/09/2013
(integer) 2

redis 127.0.0.1:6379> DECR get pagina:/contato:02/09/2013
(integer) 1
```

No primeiro comando, resgatamos o valor da chave `pagina:/contato:02/09/2013` que já tínhamos criado anteriormente. Na sequência, executamos o comando `DECR` nessa mesma chave e seu valor foi decrementado para 2, e na segunda execução, o valor da chave foi novamente decrementado para 1.

Uma outra forma de decrementar ou incrementar o valor de uma chave é utilizando o comando `INCRBY`. Esse comando se distingue dos outros, pois nele precisamos determinar o valor do incremento ou decremento que será aplicado à chave. Vamos continuar usando a mesma chave

`pagina:/contato:02/09/2013` e demonstrar o uso desse comando pelo CLI conforme a sequencia de comandos a seguir:

```
redis 127.0.0.1:6379> GET pagina:/contato:02/09/2013
"1"

redis 127.0.0.1:6379> INCRBY pagina:/contato:02/09/2013 4
(integer) 5

redis 127.0.0.1:6379> INCRBY pagina:/contato:02/09/2013 -2
(integer) 3
```

Foi usado inicialmente o comando `GET` apenas para verificar o valor da chave `pagina:/contato:02/09/2013`. Em seguida, usamos o comando `INCRBY` para incrementar em 4 e depois decrementar em 2 o valor da chave.

Repare que até agora todas as operações que realizamos foram feitas utilizando números inteiros, mas e se quisermos incrementar ou decrementar o valor em 0.5? Isso também foi pensado e disponibilizado através do comando `INCRBYFLOAT`. O comando `INCRBYFLOAT` interpreta o valor (que é um String) da chave como um número de ponto flutuante. Vejamos seu exemplo de uso:

```
redis 127.0.0.1:6379> GET pagina:/contato:02/09/2013
"3"

redis 127.0.0.1:6379>
  INCRBYFLOAT pagina:/contato:02/09/2013 2.789
"5.789"

redis 127.0.0.1:6379>
  INCRBYFLOAT pagina:/contato:02/09/2013 -2.789
"3"
```

Se esquecermos por um minuto que utilizamos um número em ponto flutuante para incrementar e depois decrementar o valor da chave `pagina:/contato:02/09/2013`, não temos nada de muito diferente quando comparamos esse exemplo de uso do comando `INCRBYFLOAT` com o exemplo do comando `INCRBY`.

Referência rápida de comandos para Incremento/Decremento

- `INCR chave` — incrementa (adiciona 1) ao valor (número inteiro) da chave;
- `INCRBY chave incremento` — incrementa ou decrementa o valor (número inteiro) da chave conforme o valor do incremento;
- `INCRBYFLOAT chave incremento` — incrementa ou decrementa o valor (número de ponto flutuante) da chave conforme o valor do incremento.

4.3 Estatísticas de usuários únicos por data

Você vê mais um exemplo de estatísticas e logo pensa no comando `INCR` ou `INCRBY`, certo? Errado! Neste exemplo, vamos ver um recurso muito interessante no Redis, que é chamado de *Redis Bitmap*. **Bitmaps** são, em sua essência, um array de bits [2], ou simplificando, um array composto por zeros e uns que podem ser usados como uma representação de determinados tipos de informação. O Redis fornece diversos comandos para manipular *Bitmaps*, entre eles comandos para definir e obter os bits de uma posição (índice) do array definido em uma chave.

Esse exemplo difere do exemplo apresentado na seção 4.2, pois agora iremos armazenar os dados levando em consideração o usuário e a data em que o usuário acessou o site, enquanto o outro exemplo apenas considerava as visitas realizadas em cada página. A principal diferença é que agora nosso volume de dados armazenados no Redis será infinitamente maior quando comparado com o outro exemplo, isso porque agora os dados não possuem limites e vão aumentar de acordo com a quantidade de dias, enquanto o outro tinha dados baseados apenas na quantidade de páginas do site.

Um outro ponto interessante que devemos entender é que no exemplo anterior 4.2, o valor armazenado em cada chave já representava o total de visitas de cada página e não foi necessário efetuar nenhuma outra operação. Já nesse exemplo os dados são armazenados por cada dia e serão processados conforme nossa necessidade.

E é exatamente por conta desse grande volume de dados (chaves) que teremos que armazenar no Redis e das operações que teremos que realizar sobre esses dados que iremos usar *Bitmaps*, pois ao armazenar os dados de forma binária, a quantidade de memória ou espaço em memória utilizada para representar os dados é bem menor. Uma outra característica muito importante é que cálculos efetuados com comandos de *BITMAP* são extremamente rápidos e por este motivo é que *Bitmaps* são utilizados para gerar estatísticas em tempo real sem necessitar de muito recurso de hardware.

Mas ao invés de partirmos diretamente para o exemplo proposto, vamos executar alguns comandos através do *CLI* para que o uso de *BITMAPS* fique mais claro. Veja a seguir:

```
127.0.0.1:6379> SETBIT cliques:anuncio:CASADOCODIGO 1000 1
(integer) 0
127.0.0.1:6379> SETBIT cliques:anuncio:CASADOCODIGO 1001 1
(integer) 0
127.0.0.1:6379> SETBIT cliques:anuncio:CASADOCODIGO 1002 1
(integer) 0
```

A primeira coisa que temos que conhecer é o comando `SETBIT`. Ele pode definir ou remover o bit de um *offset* em um valor armazenado em uma chave. Um pouco confuso, não? Vamos entendê-lo através do exemplo. A princípio, esse comando é parecido com o `SET`, ou seja, ele define um valor para uma determinada chave. A chave no nosso exemplo é `cliques:anuncio:CASADOCODIGO` e ela representa todos os usuários **únicos** que, de fato, clicaram no anúncio chamado `CASADOCODIGO` em uma página qualquer do nosso site.

Cuidado com valores de offsets grandes

Ao utilizar *Redis Bitmaps*, é possível armazenar dados para milhões de offsets em pouca quantidade de memória, que no geral pode chegar em alguns megabytes. É necessário tomar cuidado com valores de offsets muito grandes pois, para determinados tamanhos, o Redis precisará alocar mais memória e, em consequência, terá o servidor bloqueado por um tempo. Caso precise de mais detalhes sobre isso, verifique o link a seguir:

http://redis.io/commands/setbit

Os usuários são representados pelo seu código, que no nosso exemplo são os números 1000, 1001 e 1002. Já o parâmetro final (número 1) utilizado em cada comando é o que chamados de **bit**, ele serve para sinalizar que o usuário de fato clicou no anúncio. Pense nele como um valor binário, o que é verdade pois esse parâmetro só pode receber o valor 0 ou 1 (podemos pensar nele como "TRUE" ou "FALSE"). Ou seja, quando definimos o valor do bit como 1 significa que o usuário efetuou o clique e se (re)definirmos seu valor para 0 significa que o usuário não clicou no anúncio ou que o seu clique foi invalidado por algum motivo. Agora vamos realizar outros comandos:

```
127.0.0.1:6379> KEYS *
1) "cliques:anuncio:CASADOCODIGO"
127.0.0.1:6379> GETBIT cliques:anuncio:CASADOCODIGO 1000
(integer) 1
127.0.0.1:6379> GETBIT cliques:anuncio:CASADOCODIGO 1001
(integer) 1
127.0.0.1:6379> GETBIT cliques:anuncio:CASADOCODIGO 1002
(integer) 1
127.0.0.1:6379> GETBIT cliques:anuncio:CASADOCODIGO 8888
(integer) 0
```

O primeiro comando executado, como já sabemos, serve apenas para listar todas as chaves que existem no Redis. Todas as chaves existentes no Redis foram removidas antes deste exemplo para facilitar a sua compreensão. Note que uma única chave representa todos os usuários que efetuaram o clique no

anúncio *CASADOCODIGO*. O comando `GETBIT` recupera o valor do bit armazenado para um determinado offset (usuário no nosso exemplo). Quando o usuário informado corresponde a um usuário que efetuou o clique, o valor do bit é 1, e quando informamos um usuário (8888) que não efetuou o clique, o comando retorna o valor 0.

Para finalizar essa breve introdução aos comandos `SETBIT` e `GETBIT`, vamos invalidar o clique do usuário 1001 e ver o que acontece. Veja isso no seguinte exemplo:

```
127.0.0.1:6379> SETBIT cliques:anuncio:CASADOCODIGO 1001 0
(integer) 1
127.0.0.1:6379> GETBIT cliques:anuncio:CASADOCODIGO 1000
(integer) 1
127.0.0.1:6379> GETBIT cliques:anuncio:CASADOCODIGO 1001
(integer) 0
127.0.0.1:6379> GETBIT cliques:anuncio:CASADOCODIGO 1002
(integer) 1
```

Pronto, já tivemos uma breve introdução sobre *Bitmaps* e como eles funcionam no Redis. Agora vamos voltar ao nosso exemplo. Primeiro criaremos uma aplicação Java para gerar os dados: vamos simular 1000 acessos de 500 usuários durante um período de 30 dias de um mês e ano predefinido. Veja o exemplo a seguir:

```
public class ArmazenarAcessosDosUsuariosComBitmap {

  public void armazenar(long codigoDoUsuario, String data) {
    Jedis jedis = new Jedis("localhost");
    String chave = String.format("acesso:%s", data);

    jedis.setbit(chave, codigoDoUsuario, true);
  }

  public static void main(String[] args) {
    int quantidadeDeUsuarios = 500;
    int quantidadeDeAcessos = 1000;
    int quantidadeDeDias = 30;
```

```
    Random random = new Random();
    ArmazenarAcessosDosUsuariosComBitmap acesso =
      new ArmazenarAcessosDosUsuariosComBitmap();

    for(Integer numero=1; numero<=quantidadeDeAcessos; numero++){
      long usuario = (random.nextInt(quantidadeDeUsuarios) + 1);
      String data = String.format(
        "%02d/11/2013",
        (random.nextInt(quantidadeDeDias) + 1)
      );
      acesso.armazenar(usuario, data);
    }
  }

}
```

Pronto, agora já temos uma boa quantidade de dados armazenados no Redis. Repare que novamente utilizamos o comando `SETBIT` para armazenar nossos dados, sendo que a chave definida para esse exemplo é composta como `acesso:DD/MM/AAAA` e o código do usuário foi usado como offset para nosso *Bitmap*. Como todos os dados foram gerados de forma aleatória devido à classe Java `Random`, o resultado do exemplo anterior pode variar a cada execução e afetar diretamente o resultado dos próximos exemplos que veremos a seguir.

Vamos criar uma aplicação para extrair as informações que precisamos sobre usuários durante o período de um dia e de uma semana (sete dias). Veja a aplicação a seguir:

```
public class ObterDadosAcessoPorDataComBitmap {

  public long acessosPorPeriodo(String...datas) {
    Jedis jedis = new Jedis("localhost");
    long total = 0;

    for (String data : datas) {
      String chave = String.format("acesso:%s", data);
      total += jedis.bitcount(chave);
    }
```

```
    return total;
  }

  public static void main(String[] args) {
    ObterDadosAcessoPorDataComBitmap dadosDeAcesso =
        new ObterDadosAcessoPorDataComBitmap();

    String[] diario = { "05/11/2013" };

    String[] semanal = {
        "16/11/2013",
        "17/11/2013",
        "18/11/2013",
        "19/11/2013",
        "20/11/2013",
        "21/11/2013",
        "22/11/2013"
    };

    long totalDiario = dadosDeAcesso.acessosPorPeriodo(diario);
    long totalSemanal = dadosDeAcesso.acessosPorPeriodo(semanal);

    System.out.println(
        String.format(
            "Total de usuários únicos no dia %s foi: %d",
            Arrays.asList(diario),
            totalDiario
        )
    );

    System.out.println(
        String.format(
            "Total de usuários únicos nos dias %s foi: %d",
            Arrays.asList(semanal),
            totalSemanal
        )
    );
  }
```

```
}
```

Nesse exemplo, utilizamos um novo comando, o `BITCOUNT`. Ele retorna o número de bits definidos no valor de uma chave, ou seja, ele soma a quantidade de offsets que tiveram o bit definido como 1 em um bitmap. O resultado desse exemplo é:

```
Total de usuários únicos no dia [05/11/2013]: 32

Total de usuários únicos nos dias [16/11/2013,
  17/11/2013, 18/11/2013,
  19/11/2013, 20/11/2013,
  21/11/2013, 22/11/2013]: 193
```

Agora para facilitar nosso próximo exemplo e garantir que o resultado apresentado seja o mesmo sempre, vamos definir o acesso de alguns usuários representados pelos códigos 10, 20, 30 e 40 para os dias 1, 2, 3 de janeiro de 2014. Veja os comandos a seguir executados no CLI:

```
127.0.0.1:6379> SETBIT acesso:01/01/2014 10 1
(integer) 0
127.0.0.1:6379> SETBIT acesso:01/01/2014 20 1
(integer) 0
127.0.0.1:6379> SETBIT acesso:01/01/2014 30 1
(integer) 0
127.0.0.1:6379> SETBIT acesso:02/01/2014 20 1
(integer) 1
127.0.0.1:6379> SETBIT acesso:02/01/2014 30 1
(integer) 0
127.0.0.1:6379> SETBIT acesso:02/01/2014 40 1
(integer) 0
127.0.0.1:6379> SETBIT acesso:03/01/2014 10 1
(integer) 0
127.0.0.1:6379> SETBIT acesso:03/01/2014 20 1
(integer) 0
127.0.0.1:6379> SETBIT acesso:03/01/2014 30 1
(integer) 0
```

Já temos os dados definidos, agora vamos supor que precisamos saber quantos usuários únicos realizaram acesso tanto no dia 01 como no dia 02. Inicialmente, você pode pensar que isso é simples de se resolver, pois poderíamos obter todos os usuários do dia 01 e depois todos os usuários do dia 02, e fazer uma comparação.

Simples assim, não? Não, porque você esqueceu que os dados estão armazenados em forma de **bitmap**. Mas para nossa sorte, o Redis fornece uma forma bem simples e versátil de fazer esse tipo de consulta. Para isso vamos utilizar e conhecer o comando `BITOP`, que realiza operações binárias (*Bitwise Operation*) [9] entre múltiplas chaves e o resultado dessa operação é armazenada em uma nova chave.

Para facilitar o seu entendimento, vamos colocar o seu uso em prática para resolver o nosso exemplo. Veja a seguir o comando:

```
127.0.0.1:6379> BITOP AND acessos_dias_01_e_02 acesso:01/01/2014
  acesso:02/01/2014
(integer) 6
```

Embora à primeira vista o uso do comando `BITOP` pareça complexo, ele é bem simples. Repare que, logo após o comando, é informado o tipo de operação binária que será realizada, que no nosso exemplo é o operador `AND`. A seguir, informamos o nome de uma nova chave (`acessos_dias_01_e_02`) que ainda não existe; o resultado desse comando será armazenado nessa chave. E por fim, as chaves `acesso:01/01/2014` e `acesso:02/01/2014`, que serão utilizadas pelo comando `BITOP`.

Nesse exemplo, o que o comando faz basicamente é criar uma nova chave que contenha apenas os valores que tiveram o bit definido como 1 em ambas as chaves (`01/01/2014` e(AND) `02/01/2014`). Repare também que o comando retornou o valor `6` — esse número foi a quantidade de valores que o comando utilizou na operação e não a quantidade de valores armazenados na nova chave. Vamos utilizar o comando `BITCOUNT` para saber quantos usuários realizaram acessos nos dois dias. Veja o comando a seguir:

```
127.0.0.1:6379> BITCOUNT acessos_dias_01_e_02
(integer) 2
```

Pelo resultado do comando, vemos que dois usuários realizaram acesso em ambos os dias. Em uma rápida análise no bloco de comandos para popular nossos dados, conseguimos ver que isso está correto e que o código desses usuários são os códigos 20 e 30, enquanto os códigos 10 e 40 foram descartados. Vamos verificar isso diretamente no Redis, conforme comandos a seguir:

```
127.0.0.1:6379> GETBIT acessos_dias_01_e_02 10
(integer) 0
127.0.0.1:6379> GETBIT acessos_dias_01_e_02 20
(integer) 1
127.0.0.1:6379> GETBIT acessos_dias_01_e_02 30
(integer) 1
127.0.0.1:6379> GETBIT acessos_dias_01_e_02 40
(integer) 0
```

Vamos utilizar novamente o mesmo comando e as mesmas chaves, mas mudaremos o operador `AND` para `OR`. Veja os comandos a seguir, executados diretamente pelo CLI:

```
127.0.0.1:6379> BITOP OR acessos_dias_01_ou_02 acesso:01/01/2014
  acesso:02/01/2014
(integer) 6
```

Novamente, o comando `BITOP` retornou o número que 6, que corresponde à quantidade de valores utilizados na operação. As diferenças aqui ficam por conta do operador `OR` e da chave que armazenou o resultado, que agora se chama `acessos_dias_01_ou_02`. No exemplo do operador `AND`, a operação retornou apenas os valores que existiam em ambas as datas ou bitmaps, já usando o operador `OR` a operação retornou os valores que existem em uma chave **ou** em outra. Veja os comandos a seguir:

```
127.0.0.1:6379> BITCOUNT acessos_dias_01_ou_02
(integer) 4
127.0.0.1:6379> GETBIT acessos_dias_01_ou_02 10
(integer) 1
127.0.0.1:6379> GETBIT acessos_dias_01_ou_02 20
(integer) 1
127.0.0.1:6379> GETBIT acessos_dias_01_ou_02 30
```

```
(integer) 1
127.0.0.1:6379> GETBIT acessos_dias_01_ou_02 40
(integer) 1
```

No bloco anterior, o comando `BITCOUNT` retornou a quantidade de valores armazenados na chave com o resultado da operação que usou o operador `OR`, que no nosso exemplo são 4. Em seguida, executamos o comando `GETBIT` para cada valor existente em ambas as chaves (`acesso:01/01/2014` e `acesso:02/01/2014`) para confirmar que todos os valores estão com o bit definido como 1.

> **O comando BITOP e seus operadores**
>
> O comando `BITOP` suporta quatro operadores lógicos: `AND`, `OR`, `XOR` e `NOT`. Os operadores `AND`, `OR` e `XOR` são executados todos da mesma forma: em outras palavras, você informa uma chave que irá receber o resultado da operações seguida pelas chaves que serão utilizadas na operação.
>
> Já o operador `NOT` é executado utilizando apenas uma chave para receber os valores do resultado e uma chave com os valores que serão utilizados na operação. De forma resumida, o operador `NOT` apenas inverte os bits dos valores (offsets).

Exercícios sobre Bitmaps

1) Utilize o comando `BITOP` com o operador `XOR` entre as chaves `acesso:01/01/2014` e `acesso:02/01/2014` e armazene o resultado em uma chave chamada `acessos_dias_01_xor_02`;

2) Utilize o comando `BITOP` com o operador `NOT` na chave `acesso:03/01/2014` e armazene o seu resultado em uma chave chamada `acessos_dia_04_not`.

Referência rápida de comandos para Bitmaps

- `BITCOUNT chave` — retorna a quantidade de bits definidos em uma chave;
- `BITOP operador chave-resultado chave [chave...]` — realiza uma operação lógica entre diversas chaves e armazena seu resultado em uma chave definida;
- `GETBIT chave offset` — retorna o valor do bit de um offset armazenado em uma chave;
- `SETBIT chave offset valor` — define o valor do bit de um offset de uma chave.

4.4 Próximos passos

Nesta segunda parte, nós aprendemos a como incrementar e decrementar valores numéricos armazenados no Redis e também como aplicar um tempo de expiração para as chaves armazenadas. Além disso, também conhecemos um poderoso recurso chamado *Redis Bitmaps* e aprendemos a utilizá-lo em um cenário real.

No próximo capítulo, vamos continuar com a terceira parte dos exemplos práticos, com mais comandos novos e com dois novos tipos de dados suportados pelo Redis, o *List* e o *Set*.

Capítulo 5

Redis no mundo real — Parte 3

5.1 Lista das últimas páginas visitadas

Dados do tipo **list** (ou lista) no Redis são basicamente listas de strings ordenadas pela ordem de inserção de cada item. O Redis possibilita que um novo item da lista possa ser inserido tanto no início (head) da lista como no seu final (tail), utilizando os comandos `LPUSH` e `RPUSH` respectivamente. O tamanho máximo de elementos contidos em uma única lista é de mais de quatro bilhões de elementos ou, sendo mais preciso, 4294967295 elementos. Veja a seguir 5.1 uma ilustração do tipo de dado *List*:

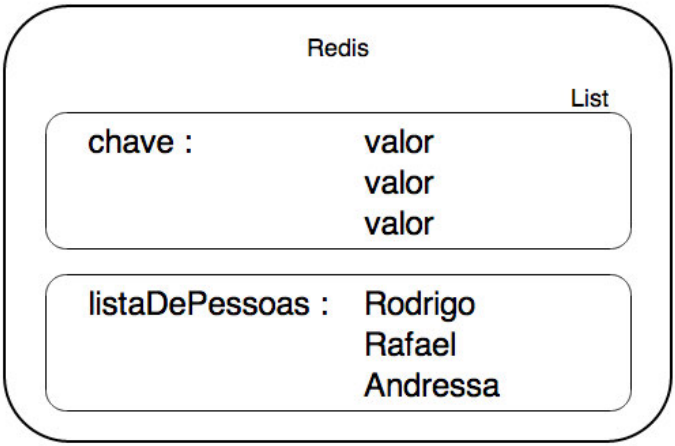


Figura 5.1: Tipo de dado List

Agora que tivemos uma rápida introdução sobre o tipo de dado *list*, vamos criar um exemplo de uso utilizando um recurso muito útil e simples de implementar com o Redis: exibir os últimos elementos adicionados a uma lista. Mas para tornar o exemplo mais real vamos exibir as últimas páginas visitadas do blog fictício chamado *Tudo Sobre Redis*, que conhecemos na seção 4.2. Esse conceito pode, inclusive, ser aplicado ou adaptado em outros contextos, como para exibir os últimos itens visitados em uma loja virtual ou as últimas mensagens de um perfil em uma rede social.

Como exemplos para as últimas páginas visitadas do nosso blog, vamos rever as páginas que usamos no exemplo anterior da seção 4.2:

```
/inicio
/contato
/sobre-mim
/todos-os-posts
/armazenando-dados-no-redis
```

Note que as páginas serão inseridas em uma lista no Redis seguindo esta mesma ordem, que refere-se à ordem na qual elas foram acessadas, sendo que o primeiro elemento `/inicio` corresponde à primeira página acessada e

assim sucessivamente. Para armazená-las, vamos utilizar o comando `LPUSH`, que insere um novo item ao topo (head) da lista e aceita um ou mais elementos para que sejam armazenados em uma lista. Vejamos o exemplo em Java do seu uso:

```
String chave = "ultimas_paginas_visitadas";
String[] paginasVisitadas = {
    "/inicio",
    "/contato",
    "/sobre-mim",
    "/todos-os-posts",
    "/armazenando-dados-no-redis"
};

Jedis jedis = new Jedis("localhost");
Long resultado = jedis.lpush(chave, paginasVisitadas);

System.out.println(
    String.format(
        "A lista %s contém %d elementos", chave, resultado
    )
);
```

A lista utilizada nesse exemplo chama-se `ultimas_paginas_visitadas`. Para o nosso caso, enviamos todas as páginas para o Redis de uma única vez, mas em aplicações reais, isso seria feito a cada página, conforme elas fossem visitadas.

A mensagem de resultado do exemplo anterior é:

```
A lista ultimas_paginas_visitadas contém 5 elementos
```

Para validarmos que a lista que acabamos de criar contém os cinco elementos, vamos executar o comando `LLEN` pelo CLI informando a chave que corresponde à nossa lista e conferir seu tamanho:

```
redis 127.0.0.1:6379> LLEN ultimas_paginas_visitadas
(integer) 5
```

Já temos uma lista populada no Redis com as últimas páginas visitadas, mas, embora tenhamos armazenado 5 elementos na nossa lista, vamos definir

que iremos exibir apenas as três últimas páginas recentemente visitadas. Podemos utilizar o comando `LRANGE`, e para isso basta informar o índice inicial e o índice final dos elementos que queremos recuperar e assim obtemos uma quantidade limitada de elementos de uma lista.

Os índices utilizados no comando `LRANGE` iniciam pelo número 0, que representa o último item adicionado à lista ou o topo da lista (head). Também é possível utilizar números negativos como índice, como por exemplo o número -1, que se refere ao primeiro item adicionado à lista ou ao fim da lista (tail); o número -2 refere-se ao segundo item adicionado à lista e assim por diante. Seguindo com o nosso exemplo, vamos utilizar esse comando para obter as três últimas páginas visitadas:

```
String chave = "ultimas_paginas_visitadas";
Jedis jedis = new Jedis("localhost");
List<String> paginas = jedis.lrange(chave, 0, 2);

System.out.println("As 3 ultimas paginas visitadas são:");

for (String pagina : paginas) {
    System.out.println(pagina);
}
```

O resultado do código anterior é:

```
As 3 ultimas paginas visitadas são:
/armazenando-dados-no-redis
/todos-os-posts
/sobre-mim
```

Com isso, temos a primeira parte do nosso exemplo funcionando e já podemos exibir as três últimas páginas visualizadas no nosso blog *Tudo Sobre Redis*. Mas se pensarmos um pouco nessa solução, veremos que ela tem um problema, porque o blog irá armazenar uma grande quantidade de itens (páginas visitadas) na lista, porém sempre irá exibir apenas três itens.

Com o conhecimento sobre esse problema, vemos que é desnecessário manter armazenadas todas as páginas acessadas. Dessa forma, começamos a segunda parte do nosso exemplo, que é limitar a quantidade de itens inseridos na nossa lista.

Vamos utilizar o comando `LTRIM` para restringirmos a quantidade de páginas acessadas. Com ele, podemos definir através de um índice inicial e outro final um range que será mantido na lista e todos os registros restantes serão removidos dela. Os índices utilizados no comando `LTRIM` funcionam da mesma forma que os utilizado no comando `LRANGE`.

Para mantermos as três ultimas páginas adicionadas à nossa lista, temos que passar o índice 0 como posição inicial e o índice 2 (índices no Redis iniciam na posição 0) como posição final. Veja a seguir como fazer isso em Java:

```
String chave = "ultimas_paginas_visitadas";
Jedis jedis = new Jedis("localhost");
String resultado = jedis.ltrim(chave, 0, 2);

System.out.println(String.format("Resultado: %s", resultado));
```

O código anterior imprime o resultado:

```
Resultado: OK
```

Para validarmos que nossa lista agora contém apenas três páginas, vamos executar novamente o comando `LLEN`:

```
redis 127.0.0.1:6379> LLEN ultimas_paginas_visitadas
(integer) 3
```

Pronto, agora sempre que uma página for visitada e a adicionarmos no Redis, basta em seguida executar o comando `LTRIM` para remover os registros desnecessários.

Exercícios sobre listas

1) O comando `LINDEX` retorna o valor de um item da lista de acordo com o índice informado. Utilize-o para ver qual é a primeira e a última página da lista `ultimas_paginas_visitadas`;

2) O comando `LREM` remove um ou mais elementos de uma lista associada a uma chave. Utilize-o para remover todos os elementos da lista `ultimas_paginas_visitadas` de uma única vez.

Referência rápida de comandos para listas

- `LPUSH chave valor [valor ...]` — adiciona um ou mais valores ao topo (head) da lista definida pela chave;
- `LLEN chave` — retorna a quantidade de itens armazenados em uma lista;
- `LRANGE chave inicio fim` — retorna um range de itens armazenados em uma lista, os valores inicio e fim são índices iniciados em 0;
- `LTRIM chave inicio fim` — apara a lista deixando apenas os itens definidos entre os índices de inicio e fim.

5.2 Criando uma fila de mensagens

Neste próximo exemplo vamos ver como utilizar os Redis para gerenciar uma fila de mensagens. Isso é muito útil quando precisamos executar tarefas em background ou enviar e receber mensagens entre aplicações de forma assíncrona. O Redis é muito utilizado para armazenar filas de mensagens, tanto que originou várias ferramentas que utilizam o Redis para esta finalidade. Algumas das mais populares são:

- Resque — https://github.com/resque/resque
- RestMQ — http://restmq.com/
- RQ — http://python-rq.org/
- Sidekiq — http://sidekiq.org/

Um exemplo clássico para uso de fila de mensagens ou tarefas vem da necessidade de aplicações que precisam enviar e-mail para seus usuários. Vamos usar como exemplo um site que possui um cadastro para seus usuários, mas a ativação de seu cadastro depende de uma confirmação que é enviada por e-mail assim que o usuário finaliza seu cadastro. É um recurso muito utilizado

por sites, fóruns de discussão e diversas aplicações web. É tão comum que a maioria das pessoas que utilizam a internet já precisou ativar seu cadastro em algum site dessa forma.

Claro que nosso exemplo será uma versão bem simplificada desse processo, e não irá conter diversas validações, informações e características que não são necessárias para torná-lo real, e que, se fossem feitas, iriam apenas poluir nosso exemplo com código que não teria relação direta com o uso do Redis. Por este motivo, nossa fila conterá basicamente um `JSON` com o nome e e-mail do usuário que efetuou o cadastro no nosso site, e o envio do e-mail em si será representado apenas por uma mensagem impressa pela aplicação.

Mas antes de partimos diretamente para o exemplo prático, vamos voltar um pouco no tempo e lembrar das divertidas aulas de estruturas de dados que tivemos na faculdade ou em cursos técnicos. Mas claro, não se preocupe se você não teve aulas dessa matéria ou se ainda vai ter, pois vamos pegar emprestado apenas um conceito visto nessa matéria, nada muito complexo.

Esse conceito é o da **fila FIFO** [8], acrônimo para *First In, First Out*, que podemos traduzir para o português como *Primeiro a entrar, Primeiro a sair*. A ideia de uma lista FIFO é que o primeiro item inserido na fila é também o primeiro item a ser removido. Podemos fazer analogia com um exemplo real: imagine que você vai até uma agência bancária realizar o pagamento de uma conta, mas chegando no banco você se depara com uma fila de cinco pessoas. Nesta fila, a primeira pessoa será a primeira a ser atendida; em seguida, a segunda pessoa será atendida e assim por diante, e você que chegou por último será também a última pessoa a ser atendida. Em outras palavras, a primeira pessoa da fila é a primeira a sair (ser atendida). Veja a seguir 5.2 uma ilustração desse exemplo:

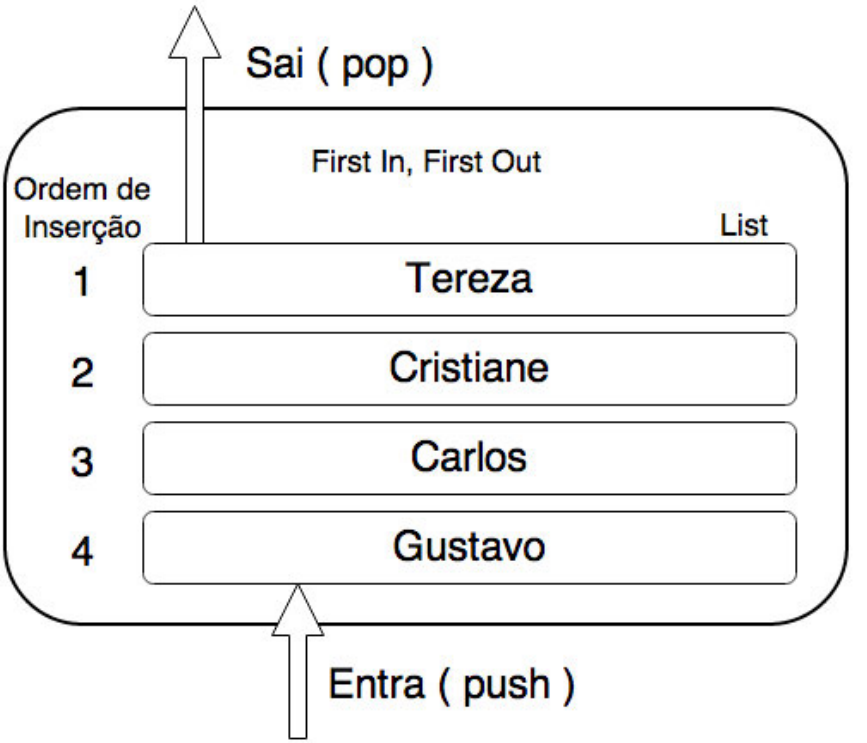


Figura 5.2: Exemplo de FIFO

Agora que entendemos o conceito de uma fila *FIFO*, vamos aplicá-lo ao nosso exemplo. A cada usuário cadastrado no nosso site, a nossa aplicação irá gerar um novo item na fila para envio de e-mails de confirmação, respeitando a ordem de chegada de cada requisição (envio dos dados do formulário) ao site.

Vamos começar pelo código que insere os itens (e-mails) na fila:

```
public class ArmazenarItemNaFila {

    public void agendarAutorizacaoDeUsuario(String nome,
                String email) {
        String chave = "fila:confirmar-usuario";
        String mensagem = String.format(
```

```
                "{\"nome\": \"%s\", \"email\": \"%s\"}", nome,
                email
        );

        Jedis jedis = new Jedis("localhost");
        Long resultado = jedis.rpush(chave, mensagem);
        System.out.println(
            String.format("A fila %s contém %d tarefa(s).",
                          chave, resultado)
        );
    }

    public static void main(String[] args) {
        ArmazenarItemNaFila fila = new ArmazenarItemNaFila();

        fila.agendarAutorizacaoDeUsuario(
                "Daenerys Targaryen", "daenerys@targaryen.com"
        );

        fila.agendarAutorizacaoDeUsuario(
                "Jon Snow", "jon@snow.com"
        );

        fila.agendarAutorizacaoDeUsuario(
                "Tyrion Lannister", "tyrion@lannister.com"
        );
    }

}
```

Nesse código, o nome da fila que vai representar as mensagens ou tarefas de envio de e-mail tem o nome de `fila:confirmar-usuario` e o conteúdo da mensagem será um documento *JSON* que representa o nome e e-mail do usuário que efetuou o cadastro no nosso site. O comando que usamos para armazenar nossa mensagem na fila foi o `RPUSH`, que serve para manipular dados do tipo *list* no Redis. O resultado dessa aplicação é:

```
A fila fila:confirmar-usuario contém 1 tarefa(s).
A fila fila:confirmar-usuario contém 2 tarefa(s).
```

```
A fila fila:confirmar-usuario contém 3 tarefa(s).
```

E sim, antes que você fique em dúvida, a fila que acabamos de criar é representada pelo Redis com uma estrutura de dados do tipo *list*. Na seção 5.1, nós usamos o comando `LPUSH` para adicionar novos itens em uma lista, sendo que o `RPUSH` também realiza essa mesma tarefa. A diferença entre os dois é que, conforme já foi explicado, o comando `LPUSH` adiciona o item no começo (head) da lista, enquanto o `RPUSH` (que usamos agora) adiciona o item no final (tail) da lista. O retorno do `RPUSH` é igual ao do `LPUSH`, que é a quantidade de itens que a lista possui.

Antes de darmos continuidade ao exemplo, vamos aprender dois comandos que são essenciais para o nosso exemplo. O primeiro é o `LPOP`. O `LPOP` serve para remover e retornar o item que está no topo (head) da lista (fila), e quando esse comando remove e retorna o último item da lista, a chave automaticamente é excluída do Redis. Vamos testá-lo pelo CLI:

```
redis 127.0.0.1:6379> RPUSH filaDeDragoes Saphira
(integer) 1

redis 127.0.0.1:6379> RPUSH filaDeDragoes Glaedr
(integer) 2

redis 127.0.0.1:6379> RPUSH filaDeDragoes Thorn
(integer) 3

redis 127.0.0.1:6379> LPOP filaDeDragoes
"Saphira"

redis 127.0.0.1:6379> LPOP filaDeDragoes
"Glaedr"

redis 127.0.0.1:6379> LPOP filaDeDragoes
"Thorn"

redis 127.0.0.1:6379> LPOP filaDeDragoes
(nil)
```

Repare que iniciamos populando uma lista chamada *filaDeDragoes* com três itens utilizando o comando `RPUSH`. Logo em seguida, usamos o comando `LPOP` para remover o item do topo da lista, e ao executá-lo após todos os itens já terem sido removidos da lista, o Redis retornou o valor **(nil)** indicando que a chave `filaDeDragoes` não existia mais.

Agora que entendemos o `LPOP`, vamos conhecer o segundo comando citado anteriormente, que é o `BLPOP`. Ele realiza a mesma tarefa que o `LPOP`, ou seja, remove e retorna o item no topo da lista, mas a letra **B** do `BLPOP` significa bloqueio ou comando bloqueante (*blocking*), e esse comportamento bloqueante é o que torna o `BLPOP` confiável para usarmos uma lista do Redis como uma fila de mensagens.

Esse comportamento de bloqueio significa que a conexão fica bloqueada pelo comando (`BLPOP` no nosso caso) até que ele retorne seu resultado. É claro que, se a lista contém itens armazenados, o resultado será obtido de forma instantânea e, consequentemente, a conexão já será liberada. Uma outra característica dos comandos bloqueantes é que em geral eles recebem como parâmetro um valor (número inteiro) que define o tempo em segundos (timeout) para que o comando aguarde um resposta. Se após esse tempo o comando não obtiver uma resposta, a conexão é liberada. Vamos realizar alguns testes pelo CLI para entender de forma prática seu uso.

```
redis 127.0.0.1:6379> RPUSH filaDeDragoes Saphira
(integer) 1

redis 127.0.0.1:6379> RPUSH filaDeDragoes Glaedr
(integer) 2

redis 127.0.0.1:6379> BLPOP filaDeDragoes 1
1) "filaDeDragoes"
2) "Saphira"

redis 127.0.0.1:6379> BLPOP filaDeDragoes 1
1) "filaDeDragoes"
2) "Glaedr"

redis 127.0.0.1:6379> BLPOP filaDeDragoes 1
(nil)
```

```
(1.06s)
```

Repare nos comandos executados anteriormente. Primeiro, adicionamos dois itens à nossa lista chamada `filaDeDragoes`, em seguida, executamos o comando `BLPOP` passando como parâmetro o nome da lista e o timeout de 1 segundo. Nas duas primeiras vezes que o comando foi executado, a resposta (o item do topo da lista) foi retornada de imediado, mas na terceira vez, ele aguardou durante 1 segundo — ou 1.06 segundo para ser mais preciso — para que um novo item fosse inserido na lista. Como isso não aconteceu, o comando retornou o valor `(nil)` e liberou a conexão. É possível bloquear a conexão indefinidamente e ficar aguardando por um novo item na lista, para isso basta definir o valor do timeout como 0.

Voltando agora ao nosso exemplo em Java, vamos criar uma outra aplicação que irá consumir os itens da nossa lista `fila:confirmar-usuario`. Essa aplicação precisa ficar monitorando a nossa lista constantemente ou a cada intervalo de tempo. No nosso caso, para simplificar o código Java vamos utilizar um simples laço (loop) infinito. Veja a aplicação:

```
public class ConsumirItemDaFila {

    class Mensagem {
        private String nome;
        private String email;

        public String getNome() { return nome; }
        public void setNome(String nome) { this.nome = nome; }

        public String getEmail() { return email; }
        public void setEmail(String email) { this.email = email;}
    };

    public void enviarEmailAtivacaoUsuario() {
        int timeout = 2;
        String chave = "fila:confirmar-usuario";

        Jedis jedis = new Jedis("localhost");
        List<String> mensagens = jedis.blpop(timeout, chave);
```

```
        if (mensagens == null) {
            System.out.println(String.format(
                "A fila %s está vazia.", chave));
        }
        else {
            String json = mensagens.get(1);
            Mensagem mensagem =
                new Gson().fromJson(json, Mensagem.class);
            System.out.println(
                String.format(
                    "Enviando e-mail para %s (%s)",
                    mensagem.getEmail(),
                    mensagem.getNome()
                )
            );
        }
    }

    public static void main(String[] args) {
        ConsumirItemDaFila fila = new ConsumirItemDaFila();

        while (true) {
            fila.enviarEmailAtivacaoUsuario();
        }
    }

}
```

Vamos verificar o seu resultado quando executado:

```
Enviando e-mail para daenerys@targaryen.com (Daenerys Targaryen)
Enviando e-mail para jon@snow.com (Jon Snow)
Enviando e-mail para tyrion@lannister.com (Tyrion Lannister)
A fila fila:confirmar-usuario está vazia.
A fila fila:confirmar-usuario está vazia.
```

Quando o exemplo foi executado, o comando `BLPOP` recuperou os três itens da lista que já tínhamos inseridos anteriormente e executou nossa falsa rotina de envio de e-mail que apenas imprime uma mensagem com o nome

e e-mail a ser enviado. Como esse exemplo ficou rodando continuamente até que fosse encerrado, as próximas tentativas de obter um valor da lista pelo comando `BLPOP` aguardaram durante 2 segundos e, em seguida, apresentaram uma mensagem informando que a lista não cotinha nenhum item novo.

O nosso exemplo termina aqui. Resumidamente, tudo o que precisamos para ter uma fila de mensagens ou tarefas funcionando no Redis foram dois comandos. O primeiro foi o `RPUSH`, para enviar novos itens para o final da lista (fila), e o segundo foi o `BLPOP`, que remove e retorna os itens da lista por ordem de chegada.

Exercícios sobre filas

1) O comando `BRPOP` funciona da mesma forma que o `BLPOP`; a única diferença entre eles é que o `BRPOP` remove e retorna o último item (tail) da lista. Utilize-o no exemplo de envio de e-mail e veja como a lista se comporta com ele;

2) Altere o comando `RPUSH` pelo `LPUSH` no exemplo que popula a lista `fila:confirmar-usuario` e veja como a lista se comporta com ele.

Referência rápida de comandos para filas (listas)

- `RPUSH` *`chave valor [valor ...]`* — adiciona um ou mais valores ao final (tail) da lista definida pela chave;
- `RPUSHX` *`chave valor`* — funciona da mesma forma que o comando `RPUSH`; a única diferença entre os dois é que o comando `RPUSHX` insere um novo item somente em uma lista já existente. Caso a lista informada como parâmetro não exista, o comando retorna o valor o e a lista não é criada;
- `LPUSHX` *`chave valor`* — funciona da mesma forma que o comando `LPUSH`; a única diferença entre os dois é que o `LPUSHX` insere um novo item somente em uma lista já existente. Caso a lista informada como parâmetro não exista, o comando retorna o valor o e a lista não é criada;

- `LPOP chave` — remove e retorna o primeiro (head) item da lista informadas como parâmetro ou (nil) caso a lista esteja vazia;
- `BLPOP chave [chave] timeout` — bloqueia a conexão para remover e retornar o primeiro item de uma das listas informadas como parâmetro durante um tempo máximo definido no parâmetro *timeout*. Caso o timeout seja definido como o, a conexão fica bloqueada até que um item de uma das listas informadas seja removido e retornado pelo comando;
- `RPOP chave` — remove e retorna o ultimo (tail) item da lista informadas como parâmetro ou (nil) caso a lista esteja vazia;
- `BRPOP chave [chave] timeout` — bloqueia a conexão para remover e retornar o último item de uma das listas informadas como parâmetro durante um tempo máximo definido no parâmetro *timeout*. Caso o timeout seja definido como o, a conexão fica bloqueada até que um item de uma das listas informadas seja removido e retornado pelo comando.

5.3 Manipular relacionamento entre amigos e seus grupos

Chegou o momento de conhecer um nova estrutura de dados suportada pelo Redis. Esta estrutura é chamada de conjunto ou **set**. Ela é muito similar ao tipo *lista*, que conhecemos neste mesmo capítulo, sendo que sua principal diferença é que um *set* não permite valores iguais. Isso é essencial quando precisamos garantir que não teremos valores repetidos e, assim, não precisamos nos preocupar em verificar antes de inserir novo valor se este já existe no Redis.

Um *SET* é uma coleção não-ordenada de *binary-safe strings*, e além de armazenar itens, existem outras operações que podem ser realizadas entre conjuntos, como a interseção entre dois ou mais conjuntos, a diferença entre dois ou mais conjuntos e a união entre dois ou mais conjuntos.

Neste exemplo, vamos criar um código para manipular a conexão entre amigos e seus grupos. Nada muito complexo, apenas operações como uma

pessoa adicionando outra como seu amigo e uma pessoa entrando para um grupo de pessoas, operações comuns em redes sociais, por exemplo. A ideia principal desse caso é conseguir extrair informações de forma fácil dos dados armazenados. Veja algumas informações que conseguiremos obter:

- Saber quantas pessoas pertencem a cada grupo;
- Listar as pessoas que são membros de um determinado grupo;
- Saber se uma pessoa é membro de um determinado grupo;
- Obter todos os relacionamentos de uma pessoa que também pertence a determinado grupo.

Para isso, vamos precisar de dois conjuntos para representar nossos dados: um para representar as pessoas e outro para representar os grupos. Veja a composição de cada chave que vai representar os dois conjuntos:

```
pessoas:{codigo-da-pessoa}:relacionamentos
grupos:{codigo-do-grupo}:membros
```

Vamos utilizar apenas um nome simples para representar cada pessoa e cada grupo. Em um exemplo mais real, esses códigos poderiam ser um código de uma chave de um *hash* que teria todos os dados referentes a uma pessoa ou um grupo. Vamos começar com um exemplo para popular os relacionamentos entre as pessoas, depois um outro para popular os membros de cada grupo e, por último, outro para obter as informações que precisamos. Veja o primeiro exemplo a seguir:

```
public class ConjuntoDeRelacionamentoEntrePessoas {

    public void adicionaAmigos(String pessoa, String[] amigos) {
        String chave = String.format(
            "pessoas:%s:relacionamentos", pessoa);
        Jedis jedis = new Jedis("localhost");
        long resultado = jedis.sadd(chave, amigos);

        System.out.println(
```

```
                String.format(
                        "%s tem %d amigos %s",
                        pessoa,
                        resultado,
                        Arrays.toString(amigos)
                )
        );
    }

    public static void main(String[] args) {
        ConjuntoDeRelacionamentoEntrePessoas relacionamentos =
                new ConjuntoDeRelacionamentoEntrePessoas();

        relacionamentos.adicionaAmigos(
                "rafael",
                new String[] { "gustavo", "andressa",
                               "rodrigo", "tereza" }
        );

        relacionamentos.adicionaAmigos(
                "andressa",
                new String[] { "cristiane", "rodrigo",
                               "gustavo", "rafael" }
        );

        relacionamentos.adicionaAmigos(
                "gustavo",
                new String[] { "carlos", "tereza",
                               "rafael", "andressa" }
        );

        relacionamentos.adicionaAmigos(
                "cristiane",
                new String[] { "tereza",
                               "andressa", "carlos" }
        );

        relacionamentos.adicionaAmigos(
                "carlos",
```

```
                new String[] { "cristiane",
                                "rodrigo", "gustavo" }
        );

        relacionamentos.adicionaAmigos(
                "rodrigo",
                new String[] { "andressa",
                                "rafael", "carlos" }
        );

        relacionamentos.adicionaAmigos(
                "tereza",
                new String[] { "gustavo",
                                "rafael", "cristiane" }
        );
    }

}
```

Nesse código, tudo o que fizemos foi usar o comando `SADD` para adicionar um conjunto representado por uma chave e seus respectivos itens (valores). O comando `SADD` adiciona um ou mais itens ao conjunto (chave) informado como parâmetro, e ele retorna a quantidade de itens que foram inseridos no conjunto. O código apresenta o seguinte resultado:

```
rafael tem 4 amigos [gustavo, andressa, rodrigo, tereza]
andressa tem 4 amigos [cristiane, rodrigo, gustavo, rafael]
gustavo tem 4 amigos [carlos, tereza, rafael, andressa]
cristiane tem 3 amigos [tereza, andressa, carlos]
carlos tem 3 amigos [cristiane, rodrigo, gustavo]
rodrigo tem 3 amigos [andressa, rafael, carlos]
tereza tem 3 amigos [gustavo, rafael, cristiane]
```

Veja a seguir uma ilustração 5.3 de cada pessoa e seus relacionamentos:

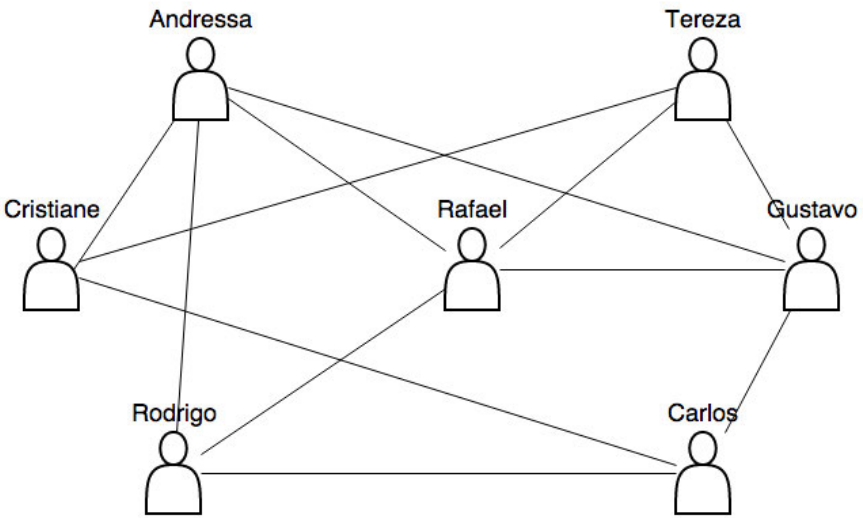


Figura 5.3: Relacionamento entre amigos

Já temos as pessoas e seus relacionamentos (amigos) definidos, agora vamos criar alguns grupos de pessoas (membros) conforme seus interesses. Para isso, vamos utilizar um código muito semelhante ao usado para criar o *set* de pessoas. Veja o código a seguir:

```
public class ConjuntoDeGrupoDePessoas {

    public void adicionaMembrosAoGrupo(String grupo,
                                       String[] membros) {
        String chave = String.format("grupos:%s:membros", grupo);
        Jedis jedis = new Jedis("localhost");
        long resultado = jedis.sadd(chave, membros);

        System.out.println(
                String.format(
                        "Grupo (%s) tem %d membros %s",
                        grupo,
                        resultado,
                        Arrays.toString(membros)
```

```
                )
        );
    }

    public static void main(String[] args) {
        ConjuntoDeGrupoDePessoas relacionamentos =
                new ConjuntoDeGrupoDePessoas();

        relacionamentos.adicionaMembrosAoGrupo(
                "video-game",
                new String[] { "rafael", "gustavo", "carlos",
                               "rodrigo" }
        );

        relacionamentos.adicionaMembrosAoGrupo(
                "judo",
                new String[] { "rafael" }
        );

        relacionamentos.adicionaMembrosAoGrupo(
                "natacao",
                new String[] { "rafael", "cristiane" }
        );

        relacionamentos.adicionaMembrosAoGrupo(
                "kung-fu",
                new String[] { "andressa" }
        );

        relacionamentos.adicionaMembrosAoGrupo(
                "violao",
                new String[] { "gustavo" }
        );

        relacionamentos.adicionaMembrosAoGrupo(
                "ciclismo",
                new String[] { "cristiane" }
        );
```

```
        relacionamentos.adicionaMembrosAoGrupo(
                "cachorro",
                new String[] { "cristiane", "rodrigo",
                               "tereza" }
        );

        relacionamentos.adicionaMembrosAoGrupo(
                "moto",
                new String[] { "carlos" }
        );

        relacionamentos.adicionaMembrosAoGrupo(
                "carro",
                new String[] { "carlos", "rodrigo" }
        );

        relacionamentos.adicionaMembrosAoGrupo(
                "livro",
                new String[] { "gustavo", "rodrigo" }
        );

        relacionamentos.adicionaMembrosAoGrupo(
                "novela",
                new String[] { "andressa", "cristiane",
                               "tereza" }
        );
    }

}
```

O resultado do código que acabamos de criar é:

```
Grupo (video-game) tem 4 membros [rafael, gustavo, carlos,
    rodrigo]
Grupo (judo) tem 1 membros [rafael]
Grupo (natacao) tem 2 membros [rafael, cristiane]
Grupo (kung-fu) tem 1 membros [andressa]
Grupo (violao) tem 1 membros [gustavo]
Grupo (ciclismo) tem 1 membros [cristiane]
Grupo (cachorro) tem 3 membros [cristiane, rodrigo, tereza]
```

```
Grupo (moto) tem 1 membros [carlos]
Grupo (carro) tem 2 membros [carlos, rodrigo]
Grupo (livro) tem 2 membros [gustavo, rodrigo]
Grupo (novela) tem 3 membros [andressa, cristiane, tereza]
```

Saber quantas pessoas pertencem a cada grupo

Agora que temos todas as informações necessárias para o exemplo já armazenadas no Redis, vamos utilizar alguns comandos interessantes que estão disponíveis para o tipo de dado *set*. O primeiro comando que vamos utilizar é o `SCARD` e vamos usá-lo para saber o total de pessoas que pertencem a cada grupo, pois ele retorna a cardinalidade ou a quantidade de elementos em um conjunto. Veja seu uso no seguinte exemplo:

```
public class TotalDePessoasPorGrupo {

    public void mostrarQuantidadeDeMembros(String grupo) {
        String chave = String.format("grupos:%s:membros", grupo);
        Jedis jedis = new Jedis("localhost");
        long resultado = jedis.scard(chave);

        System.out.println(
            String.format("Grupo (%s) tem %d membros", grupo,
                resultado)
        );
    }

    public static void main(String[] args) {
        TotalDePessoasPorGrupo grupo =
                new TotalDePessoasPorGrupo();

        grupo.mostrarQuantidadeDeMembros("video-game");
        grupo.mostrarQuantidadeDeMembros("judo");
        grupo.mostrarQuantidadeDeMembros("natacao");
        grupo.mostrarQuantidadeDeMembros("kung-fu");
        grupo.mostrarQuantidadeDeMembros("violao");
        grupo.mostrarQuantidadeDeMembros("ciclismo");
        grupo.mostrarQuantidadeDeMembros("cachorro");
        grupo.mostrarQuantidadeDeMembros("moto");
```

```
        grupo.mostrarQuantidadeDeMembros("carro");
        grupo.mostrarQuantidadeDeMembros("livro");
        grupo.mostrarQuantidadeDeMembros("novela");
    }

}
```

O resultado do código anterior é muito parecido com o resultado de quando incluímos os grupos. Veja:

```
Grupo (video-game) tem 4 membros
Grupo (judo) tem 1 membros
Grupo (natacao) tem 2 membros
Grupo (kung-fu) tem 1 membros
Grupo (violao) tem 1 membros
Grupo (ciclismo) tem 1 membros
Grupo (cachorro) tem 3 membros
Grupo (moto) tem 1 membros
Grupo (carro) tem 2 membros
Grupo (livro) tem 2 membros
Grupo (novela) tem 3 membros
```

Listar as pessoas que são membros de um determinado grupo

O próximo exemplo que vamos criar é para listar as pessoas que são membros de um determinado grupo. Para fazermos isso, precisamos conhecer um novo comando chamado `SMEMBERS`, que recebe como parâmetro o nome (chave) do conjunto (set) que queremos acessar e retorna todos os elementos contidos no conjunto. Vamos verificar seu uso no exemplo a seguir:

```
public class ListarPessoasDosGrupos {

    public void listarMembros(String grupo) {
        String chave = String.format("grupos:%s:membros", grupo);
        Jedis jedis = new Jedis("localhost");
        Set<String> membros = jedis.smembers(chave);

        System.out.println(
                String.format(
```

```
                    "Membros do grupo (%s): %s", grupo,
                        membros.toString()
                )
        );
    }

    public static void main(String[] args) {
        ListarPessoasDosGrupos grupos =
            new ListarPessoasDosGrupos();

        grupos.listarMembros("video-game");
        grupos.listarMembros("judo");
        grupos.listarMembros("natacao");
        grupos.listarMembros("kung-fu");
        grupos.listarMembros("violao");
        grupos.listarMembros("ciclismo");
        grupos.listarMembros("cachorro");
        grupos.listarMembros("moto");
        grupos.listarMembros("carro");
        grupos.listarMembros("livro");
        grupos.listarMembros("novela");
    }

}
```

O resultado desse exemplo é:

```
Membros do grupo (video-game): [gustavo, rafael, carlos,
    rodrigo]
Membros do grupo (judo): [rafael]
Membros do grupo (natacao): [rafael, cristiane]
Membros do grupo (kung-fu): [andressa]
Membros do grupo (violao): [gustavo]
Membros do grupo (ciclismo): [cristiane]
Membros do grupo (cachorro): [tereza, cristiane, rodrigo]
Membros do grupo (moto): [carlos]
Membros do grupo (carro): [carlos, rodrigo]
Membros do grupo (livro): [gustavo, rodrigo]
Membros do grupo (novela): [andressa, tereza, cristiane]
```

Saber se uma pessoa é membro de um determinado grupo

Para realizar essa tarefa, poderíamos obter todos os elementos do conjunto, iterar sobre eles e validar se algum dos elementos contém o valor que procuramos. Sim, isso é possível e funciona, mas o Redis já fornece um comando para verificar isso de forma automática. O comando para isso é o `SISMEMBER` — repare que não é o mesmo comando `SMEMBERS` que usamos no exemplo anterior. Esse comando recebe, além do nome (chave) do conjunto, o valor que queremos validar se existe no conjunto. O valor 1 é retornado quando o valor existe, e o valor 0 é retornado quando o valor não existe.

Vejamos agora um exemplo de uso desse comando:

```
public class PessoaExisteNoGrupo {

    public void existe(String grupo, String pessoa) {
        String chave = String.format("grupos:%s:membros",
                grupo);
        Jedis jedis = new Jedis("localhost");
        boolean resultado = jedis.sismember(chave, pessoa);

        System.out.println(
                String.format(
                    "%s é membro do grupo (%s)? %s",
                    pessoa,
                    grupo,
                    resultado ? "SIM" : "NÃO"
                )
        );
    }

    public static void main(String[] args) {
        PessoaExisteNoGrupo pessoa = new PessoaExisteNoGrupo();

        pessoa.existe("judo", "rodrigo");
        pessoa.existe("livro", "gustavo");
        pessoa.existe("cachorro", "cristiane");
        pessoa.existe("cachorro", "andressa");
```

```
        pessoa.existe("violao", "carlos");
    }
}
```

O resultado apresentado pelo exemplo anterior foi:

```
rodrigo é membro do grupo (judo)? NÃO
gustavo é membro do grupo (livro)? SIM
cristiane é membro do grupo (cachorro)? SIM
andressa é membro do grupo (cachorro)? NÃO
carlos é membro do grupo (violao)? NÃO
```

Obter todos os relacionamentos de uma pessoa que também pertence a determinado grupo

Esta é a última parte do exemplo. Nesse ponto, vamos conhecer mais um novo comando para o tipo de dados **set**. É o `SINTER`, que serve para comparar dois ou mais conjuntos e retornar os itens em comum entre eles. No nosso exemplo, nós queremos comparar os relacionamentos de pessoas com os membros de um grupo e verificar quais pessoas estão em ambos os conjuntos. Vamos realizar isso em um programa Java:

```
public class CompararRelacionamentosComMembrosDoGrupo {

    public void verAmigosDoGrupo(String pessoa, String grupo) {
        String chavePessoa = String.format(
            "pessoas:%s:relacionamentos", pessoa
        );

        String chaveGrupo = String.format(
            "grupos:%s:membros", grupo
        );

        Jedis jedis = new Jedis("localhost");
        Set<String> pessoas = jedis.sinter(chavePessoa,
                chaveGrupo);



        System.out.println(
```

```
            String.format(
                "%s são amigos de %s " +
                "e fazem também parte do grupo que gosta de %s",
                pessoas.toString(),
                pessoa,
                grupo
            )
        );
    }

    public static void main(String[] args) {
        CompararRelacionamentosComMembrosDoGrupo relacionamentos
                = new CompararRelacionamentosComMembrosDoGrupo();

        relacionamentos.verAmigosDoGrupo("rafael", "cachorro");
        relacionamentos.verAmigosDoGrupo("rodrigo",
                                         "video-game");
        relacionamentos.verAmigosDoGrupo("andressa", "novela");
    }

}
```

O resultado desse código Java é:

```
[tereza, rodrigo] são amigos de rafael e fazem também parte
    do grupo que gosta de cachorro

[rafael, carlos] são amigos de rodrigo e fazem também parte
    do grupo que gosta de video-game

[cristiane] são amigos de andressa e fazem também parte
    do grupo que gosta de novela
```

Com este último programa, encerramos nosso exemplo sobre a estrutura de dados `SET`. Existem outros comandos interessantes que podem ser utilizados por conjuntos, e para estes deixarei que o leitor utilize-os nos exercícios propostos para esta seção.

Exercícios sobre conjuntos

1) O comando `SMOVE` move o elemento de um conjunto para outro; utilize-o para mover o membro `rodrigo` do grupo `video-game` para o grupo `judo`;

2) O comando `SREM` remove um ou mais elementos de um conjunto associado a uma chave. Utilize-o para remover o membro `novela` do grupo `novela`;

3) O comando `SDIFF` retorna os elementos do primeiro conjunto que não existem nos outros conjuntos comparados. Utilize-o entre os relacionamentos da pessoa `cristiane` e a pessoa `gustavo` e veja quais relacionamentos eles não têm em comum.

Referência rápida de comandos para conjuntos

- `SADD` *`chave valor [valor ...]`* — adiciona um ou mais valores ao conjunto definido pela chave. Caso um valor já exista no conjunto, este será ignorado;
- `SCARD` *`chave`* — retorna a quantidade de itens armazenados em um conjunto;
- `SINTER` *`chave [chave ...]`* — retorna os elementos resultantes entre uma intersecção dos conjuntos informados;
- `SISMEMBER` *`chave valor`* — retorna 1 se o valor informado existe no conjunto informado pela chave, caso o valor não exista o comando retorna o valor 0;
- `SMEMBERS` *`chave`* — retorna todos os elementos de um conjunto definido pela chave informada.

5.4 Próximos passos

Neste capítulo, nós conhecemos dois novos tipos de dados, o *List* e *Set*. Também aprendemos a criar filas no Redis e em que cenário podemos utilizá-las. Esse que é um recurso muito útil e muito utilizado em diversos tipos de aplicações.

No próximo capítulo, vamos ver a última parte de exemplos práticos utilizando o Redis e conhecer o último tipo de dado suportado pelo Redis que ainda não foi apresentado, o *Sorted Set*.

Capítulo 6

Redis no mundo real — Parte 4

Chegamos na última parte de exemplos de uso reais com o Redis! Recapitulando rapidamente, nós já conhecemos os tipos de dados *string*, *hash* e *set*. Nesta parte, vamos conhecer o último tipo de dado fornecido pelo Redis, que é o *sorted set*.

6.1 Armazenando as vitórias dos usuários em um jogo

Nesse exemplo, nós temos que armazenar as vitórias dos usuários de um jogo online. A regra para armazenar o números de vitórias é bem simples: se o jogador venceu, adicionamos 1 ponto ao seu número de vitórias ou retiramos 1 ponto caso o jogador tenha perdido.

Uma forma de fazer isso com o Redis é utilizar um *hash* para representar

o jogador e seus dados, que são seu nome e a suas vitórias (pontuação). Para ficar mais claro, vamos inserir um usuário conforme o seguinte código:

```
int codigoJogador = 1;
String nomeJogador = "Rafael";
String chave = String.format("jogador:%04d:codigo",
                             codigoJogador);

Jedis jedis = new Jedis("localhost");
long resultado1 = jedis.hset(chave, "nome", nomeJogador);
long resultado2 = jedis.hset(chave, "pontuacao", "0");

String mensagem = String.format(
    "Resultado 1 = %d, Resultado 2 = %d",
    resultado1,
    resultado2
);

System.out.println(mensagem);
```

O resultado desse código é:

```
Resultado 1 = 1, Resultado 2 = 1
```

Pelo CLI, podemos conferir o hash armazenado:

```
redis 127.0.0.1:6379> HGETALL jogador:0001:codigo
1) "nome"
2) "Rafael"
3) "pontuacao"
4) "0"
```

Repare que usamos um comando novo para o tipo de dados `hash`, o `HGETALL`. Ele retorna todos os campos e seus respectivos valores associados a um `hash`. Conforme vimos, agora já temos um hash criado e um estrutura definida para representar os dados de um jogador.

O que precisamos agora é utilizar o mesmo conceito de incremento e decremento que utilizamos na seção 4.2. Os comandos `INCR`, `INCRBY` e `INCRBYFLOAT` funcionam apenas para valores que são do tipo `String`. Para

valores que são do tipo `hash`, o Redis fornece dois comandos: o `HINCRBY` e `HINCRBYFLOAT`.

Eles funcionam da mesma forma que as suas versões para `String`, a única diferença é que, ao invés de informarmos apenas a chave que terá seu valor incrementado ou decrementado conforme o valor informado, agora vamos informar o `hash` e o campo que terá o valor alterado conforme o valor definido como incremento.

Vamos criar um novo código em Java para que agora tenhamos uma forma de armazenar as vitórias e as derrotas dos jogadores. Veja a seguir:

```
public class ArmazenarPontuacaoJogador {

  private void definirNovaPontuacao(int codigoJogador,
                                    int ponto) {
    String chave = String.format("jogador:%04d:codigo",
            codigoJogador);
    Jedis jedis = new Jedis("localhost");

    long novaPontuacao = jedis.hincrBy(chave, "pontuacao",
            ponto);

    System.out.println(
        String.format(
            "A pontuação do jogador %04d é: %d",
            codigoJogador,
            novaPontuacao
        )
    );
  }

  public void adicionarVitoria(int codigoJogador) {
    definirNovaPontuacao(codigoJogador, 1);
  }

  public void adicionarDerrota(int codigoJogador) {
    definirNovaPontuacao(codigoJogador, -1);
  }
```

```
  public static void main(String[] args) {
    int codigoJogador = 1;

    ArmazenarPontuacaoJogador pontuacaoJogador =
        new ArmazenarPontuacaoJogador();

    pontuacaoJogador.adicionarVitoria(codigoJogador);
    pontuacaoJogador.adicionarVitoria(codigoJogador);
    pontuacaoJogador.adicionarDerrota(codigoJogador);
    pontuacaoJogador.adicionarVitoria(codigoJogador);
  }

}
```

E o resultado desse código é:

```
A pontuação do jogador 0001 é: 1
A pontuação do jogador 0001 é: 2
A pontuação do jogador 0001 é: 1
A pontuação do jogador 0001 é: 2
```

Exercícios sobre incremento/decremento com hash

1) Atualize o exemplo em Java para que ele utilize o comando `HINCRBYFLOAT` de modo que possamos armazenar um *double* (tipo primitivo Java para números de ponto flutuante) no campo *pontuacao*.

Referência rápida de comandos para incremento/decremento com hash

- `HINCRBY` *`chave campo incremento`* — incrementa ou decrementa o valor (número inteiro) do campo associado a uma chave conforme o valor do incremento;
- `HINCRBYFLOAT` *`chave campo incremento`* — incrementa ou decrementa o valor (número de ponto flutuante) do campo associado a uma chave conforme o valor do incremento.

6.2 Scores dos jogadores com Sorted Set

O tipo de dado *Sorted Set* ou *conjunto ordenado* é muito similar ao tipo de dado *Set*, pois em ambos os elementos são únicos. A principal diferença entre eles é que os elementos do *Sorted Set* contêm uma pontuação (score) para cada valor ou elemento. Essa pontuação fornece ao *Sorted Set* a capacidade de realizar ordenação de valores e também possibilita ranqueá-los de acordo com o score de cada elemento.

Para demonstrar o seu uso, vamos criar uma tabela de pontuação (scores) de jogadores para um jogo qualquer onde dois jogadores se enfrentam. Toda vez que um jogador vence, ele recebe mais 10 pontos, enquanto o jogador que foi derrotado perde 5 pontos do seu score. Uma outra regra é que todo novo jogador recebe 50 pontos ao ingressar no jogo e é por esta regra que vamos iniciar o nosso desenvolvimento. Veja o seguinte código Java:

```
public class GerarNovoJogador {

    public void adicionarNovoJogador(String jogador) {
        String chave = "scores";
        double pontuacaoInicial = 50;
        Jedis jedis = new Jedis("localhost");
        long resultado = jedis.zadd(chave, pontuacaoInicial,
            jogador);

        System.out.println(
            String.format(
    "Novo Jogador: %s com %.0f pontos iniciais. Resultado: %d",
                jogador,
                pontuacaoInicial,
                resultado
            )
        );
    }

    public static void main(String[] args) {
        GerarNovoJogador novoJogo = new GerarNovoJogador();

        novoJogo.adicionarNovoJogador("Aragorn");
```

```
        novoJogo.adicionarNovoJogador("Gandalf");
        novoJogo.adicionarNovoJogador("Legolas");
        novoJogo.adicionarNovoJogador("Gandalf");
        novoJogo.adicionarNovoJogador("Frodo");
        novoJogo.adicionarNovoJogador("Bilbo");
        novoJogo.adicionarNovoJogador("Gimli");
        novoJogo.adicionarNovoJogador("Sam");
        novoJogo.adicionarNovoJogador("Boromir");
        novoJogo.adicionarNovoJogador("Gollum");
    }

}
```

O resultado desse código é:

```
Novo Jogador: Aragorn com 50 pontos iniciais. Resultado: 1
Novo Jogador: Gandalf com 50 pontos iniciais. Resultado: 1
Novo Jogador: Legolas com 50 pontos iniciais. Resultado: 1
Novo Jogador: Gandalf com 50 pontos iniciais. Resultado: 0
Novo Jogador: Frodo com 50 pontos iniciais. Resultado: 1
Novo Jogador: Bilbo com 50 pontos iniciais. Resultado: 1
Novo Jogador: Gimli com 50 pontos iniciais. Resultado: 1
Novo Jogador: Sam com 50 pontos iniciais. Resultado: 1
Novo Jogador: Boromir com 50 pontos iniciais. Resultado: 1
Novo Jogador: Gollum com 50 pontos iniciais. Resultado: 1
```

No código anterior, o único comando do Redis que utilizamos foi o `ZADD`. Ele adiciona um ou mais elementos em um *Sorted Set*, e seu resultado corresponde à quantidade de elementos adicionados cada vez que ele é executado. Repare também que, ao tentarmos incluir novamente um elemento já existente, o comando "ignora" esse elemento e retorna o valor **0**, o que significa que nenhum novo elemento foi armazenado ao conjunto (*Sorted Set*).

Fugindo um pouco do contexto dos jogadores, vamos dar um rápido pause para conhecer um outro comando. O comando apresentado a seguir é o `ZCARD`, que retorna a quantidade de elementos armazenados em uma chave do tipo *Sorted Set*. Vamos testá-lo a seguir diretamente pelo CLI:

```
127.0.0.1:6379> ZCARD scores
(integer) 9
```

Voltando ao nosso exemplo, já temos nossos jogadores e todos já receberam sua pontuação (*score*) inicial (50 pontos). Agora vamos criar uma aplicação para armazenar o resultado de cada partida que, conforme definimos anteriormente, o jogador que vencer a partida recebe mais 10 pontos, enquanto o jogador que for derrotado perde 5 pontos.

Para não complicar muito nossa aplicação e facilitar ao máximo o entendimento de leitores que não possuem muita familiaridade com a linguagem de programação Java, vamos definir que o vencedor de cada partida será selecionado por um algoritmo que escolhe randomicamente entre os dois jogadores participantes. Com essas definições, podemos partir para o nosso código, veja a seguir essa aplicação escrita em Java:

```
public class ArmazenarResultadoDaPartida {

    public void realizarPartida(
            final String jogador1, final String jogador2) {

        List<String> jogadores = new ArrayList<String>() {{
            add(jogador1);
            add(jogador2);
        }};

        Collections.shuffle(jogadores);
        String vencedor = jogadores.get(0);
        String perdedor = jogadores.get(1);


        String chave = "scores";
        double pontosVencedor = 10;
        double pontosPerdedor = -5;

        Jedis jedis = new Jedis("localhost");

        double scoreVencedor =
                jedis.zincrby(chave, pontosVencedor, vencedor);

        double scorePerdedor =
                jedis.zincrby(chave, pontosPerdedor, perdedor);
```

```
        System.out.println(
            String.format(
        "%s venceu (score: %.0f) | %s perdeu (score: %.0f)",
                vencedor,
                scoreVencedor,
                perdedor,
                scorePerdedor
            )
        );
    }

    public static void main(String[] args) {
        ArmazenarResultadoDaPartida partidas =
                new ArmazenarResultadoDaPartida();

        for (int rodada=1; rodada<=5; rodada++) {
            System.out.println(String.format("Rodada %d",
                rodada));
            partidas.realizarPartida("Aragorn", "Gandalf");
            partidas.realizarPartida("Aragorn", "Legolas");
            partidas.realizarPartida("Aragorn", "Frodo");
            partidas.realizarPartida("Gandalf", "Bilbo");
            partidas.realizarPartida("Gandalf", "Gimli");
            partidas.realizarPartida("Gandalf", "Sam");
            partidas.realizarPartida("Frodo", "Boromir");
            partidas.realizarPartida("Frodo", "Gollum");
            partidas.realizarPartida("Gollum", "Boromir");
            partidas.realizarPartida("Sam", "Gimli");
        }

    }
}
```

Novamente, um único comando do Redis foi utilizando, o `ZINCRBY`, que é muito similar ao `INCRBY`, já visto anteriormente. A sua principal diferença é que o comando `ZINCRBY` incrementa o *score* de um elemento e não o valor da chave diretamente como o `INCRBY` faz. Uma outra característica que podemos verificar no exemplo é que ele também aceita valores negativos, isso

foi usado para retirar os pontos do perdedor de cada partida.

Repare que existe um laço *for* para que a sequência de partidas seja executada 5 vezes. O motivo de fazer isso é para que a pontuação de cada jogador tivesse uma boa variação entre um e outro. Para não poluir muito o livro, vou mostrar a seguir apenas o resultado da última rodada do código anterior. Lembre-se de que, como a definição de cada partida é feita aleatoriamente, o resultado obtido por você pode variar do apresentado a seguir:

```
Rodada 5
Gandalf venceu (score: 55) | Aragorn perdeu (score: 105)
Aragorn venceu (score: 115) | Legolas perdeu (score: 55)
Aragorn venceu (score: 125) | Frodo perdeu (score: 75)
Bilbo venceu (score: 85) | Gandalf perdeu (score: 50)
Gandalf venceu (score: 60) | Gimli perdeu (score: 35)
Sam venceu (score: 110) | Gandalf perdeu (score: 55)
Frodo venceu (score: 85) | Boromir perdeu (score: 80)
Gollum venceu (score: 65) | Frodo perdeu (score: 80)
Gollum venceu (score: 75) | Boromir perdeu (score: 75)
Sam venceu (score: 120) | Gimli perdeu (score: 30)
```

Agora que as partidas já foram realizadas e cada jogador já possui sua pontuação (score) de acordo com suas vitórias e derrotas obtidas, vamos desenvolver um código para exibir os 5 jogadores com as maiores pontuações. Veja a seguir como fazer isso:

```
public class Top5Jogadores {

    public static void main(String[] args) {
        Jedis jedis = new Jedis("localhost");
        Set<String> jogadores = jedis.zrevrange("scores", 0, 4);
        Iterator<String> iterator = jogadores.iterator();

        for (int index = 1; iterator.hasNext(); index++) {
            System.out.println(
                String.format("Posição %d - %s", index,
                    iterator.next())
            );
        }
```

```
    }

}
```

A novidade nesse código fica por conta do comando `ZREVRANGE`, que retorna uma quantidade limitada (*range*) de elementos de um *Sorted Set*, sendo que os elementos obtidos são ordenados do maior *score* para o menor. O comando `ZREVRANGE` recebe três argumentos obrigatórios e um opcional; os obrigatórios são o nome da chave que representa o *Sorted Set*, o índice inicial e índice final para formar o *range* de elementos. Repare que, no nosso exemplo, o índice inicial é o valor 0, e o final é o valor 4, totalizando um range de 5 elementos. Veja o resultado do código anterior no trecho a seguir:

```
Posição 1 - Aragorn
Posição 2 - Sam
Posição 3 - Bilbo
Posição 4 - Frodo
Posição 5 - Gollum
```

Agora já sabemos quais são os nossos 5 melhores (TOP 5) jogadores, porém não sabemos qual o score de cada um. Para isso, existe um argumento opcional que pode ser utilizado no comando `ZREVRANGE`: o valor `WITHSCORES`. Quando esse argumento é informado, além de retornar o range de elementos ele retorna também o score de cada elemento. Vamos alterar o exemplo anterior para que utilize esse argumento opcional. Veja a seguir:

```
public class Top5JogadoresComScore {

    public static void main(String[] args) {
        Jedis jedis = new Jedis("localhost");
        Set<Tuple> jogadores =
            jedis.zrevrangeWithScores("scores", 0, 4);
        Iterator<Tuple> iterator = jogadores.iterator();

        for (int index = 1; iterator.hasNext(); index++) {
            Tuple tuple = iterator.next();
            System.out.println(
```

```
                String.format(
                    "Posição %d - %s (%.0f pontos)",
                    index,
                    tuple.getElement(),
                    tuple.getScore()
                )
            );
        }

    }

}
```

Repare que no caso *Jedis* (cliente Java para Redis), o argumento é representado na forma de um outro método chamado `zrevrangeWithScores`. E agora o resultado do exemplo ficou da seguinte forma:

```
Posição 1 - Aragorn (125 pontos)
Posição 2 - Sam (120 pontos)
Posição 3 - Bilbo (85 pontos)
Posição 4 - Frodo (80 pontos)
Posição 5 - Gollum (75 pontos)
```

Nossa lista com os 5 melhores jogadores já esta pronta e já sabemos como usar o comando `ZREVRANGE` para obter esse tipo de informação. Agora vamos aprender a obter informações de um único jogador. Vamos começar pelo comando `ZSCORE`. Com ele, podemos obter o *score* de um jogador (elemento do *Sorted Set*). Veja a seguir o exemplo realizado no CLI:

```
127.0.0.1:6379> ZSCORE scores Boromir
"75"
127.0.0.1:6379> ZSCORE scores Gimli
"30"
```

Vamos utilizar o comando `ZREVRANK`, que retorna a classificação (*rank*) de um jogador (elemento) perante os outros jogadores (elementos) armazenados no *Sorted Set*. Ele ordena os elementos da maior para a menor pontuação (*score*) para assim conseguir determinar o valor da classificação. O valor da

classificação retornado tem como base o índice 0, por isso este índice equivale à primeira posição na classificação, o índice 1 à segunda posição e assim sucessivamente. Veja o seguinte exemplo:

```
127.0.0.1:6379> ZREVRANK scores Aragorn
(integer) 0
127.0.0.1:6379> ZREVRANK scores Gollum
(integer) 4
127.0.0.1:6379> ZREVRANK scores Gimli
(integer) 8
```

Vamos fazer nosso último exemplo sobre *Sorted Set*. Nesse exemplo, imagine que tenhamos um *Sorted Set* que recebe a pontuação de diversos jogadores durante um determinado período de tempo, e após esse tempo precisamos manter apenas os 5 primeiros colocados. Para isso podemos utilizar o comando `ZREMRANGEBYRANK`, conforme o exemplo a seguir:

```
127.0.0.1:6379> ZREMRANGEBYRANK scores 5 -1
(integer) 4
127.0.0.1:6379> ZREVRANGE scores 0 -1
1) "Aragorn"
2) "Sam"
3) "Bilbo"
4) "Frodo"
5) "Gollum"
```

O comando `ZREMRANGEBYRANK` recebe três argumentos, sendo o primeiro deles o nome da chave, o segundo, o índice inicial, e o último, o índice final. O range de todos os elementos que pertencerem a este range será removido do *Sorted Set*. No nosso exemplo, eu usei o índice inicial 5, que se refere ao sexto elemento do *Sorted Set*, e o valor -1 como índice final. Esse valor corresponde ao último elemento.

E assim terminamos o exemplo sobre o tipo de dados *Sorted Set*! Existem diversos outros comandos que podem ser usados com *sorted sets*. Seguindo o padrão de todos as seções do livro, a seguir irei colocar diversos comandos sobre esse tipo de dados, bem como uma breve explicação sobre cada comando.

Exercícios sobre Sorted Set

1) O comando `ZRANK`, assim como o `ZREVRANK`, retorna a classificação dos elementos de um *Sorted Set*, porém o `ZRANK` realiza a ordenação do menor score para o maior, que é a ordenação inversa à realizada pelo `ZREVRANK`. Utilize o `ZRANK` para obter uma lista dos 5 jogadores com menor pontuação armazenados na chave *scores*;

2) O comando `ZREM` remove um elemento de um *Sorted Set*; é necessário informar dois argumentos para ele, sendo eles o nome da chave e o elemento a ser removido. Utilize-o para remover o elemento "Legolas" da chave `scores`.

Referência rápida de comandos para Sorted Set

- `ZADD` *`score elemento [score elemento ...]`* — adiciona um ou mais elementos e seus respectivos *scores* em uma determinada chave;
- `ZCARD` *`chave`* — retorna a quantidade de elementos armazenados em um *sorted Set*;
- `ZCOUNT` *`chave mínimo máximo`* — retorna o número de elementos em um *sorted set* que possui o score entre um valor mínimo e máximo;
- `ZINCRBY` *`chave incremento elemento`* — incrementa o *score* de um elemento do *sorted set* definido pela chave informada como argumento;
- `ZRANGE` *`chave início fim [WITHSCORES]`* — retorna uma quantidade (*range*) de elementos em um *sorted set*. Os elementos são ordenados do menor *score* para o maior, e o range de elementos é definido conforme os valores dos argumentos *início* e *fim*. O argumento opcional *WITHSCORES*, quando informado, faz com que o comando além de retornar os elementos também retorne os scores de cada elemento;

- `ZREVRANGE chave início fim [WITHSCORES]` — funciona da mesma forma que o comando `ZRANGE`, porém faz a ordenação de elementos levando em consideração o maior score para o menor;
- `ZSCORE chave elemento` — retorna o score do elemento de um *sorted set*.

6.3 Identificando os tipos de cada chave

Até esse momento já foram utilizados diversos tipos de dados diferentes do Redis, entre eles estão: Strings, Lists, Sets, Hashes e Sorted Sets. Já vimos como cada um funciona e em que situação o uso de um tipo de dado é mais recomendado do que outro.

Podemos armazenar diferentes tipos de dados em um único *database* do Redis, de modo que quando listamos todas as chaves armazenadas, não conseguimos distinguir os tipos de dados. Para isso, o Redis fornece um comando chamado `TYPE`, que retorna o tipo de dado de uma determinada chave.

Para exemplificar o uso desse comando vamos incluir algumas chaves no *Redis* e, em seguida, executar o comando `TYPE` em cada chave criada para visualizarmos o seu resultado. Veja o exemplo a seguir:

```
127.0.0.1:6379> KEYS *
(empty list or set)
127.0.0.1:6379> SET minha-string "primeiro teste"
OK
127.0.0.1:6379> RPUSH minha-lista "segundo teste"
(integer) 1
127.0.0.1:6379> HSET meu-hash descricao "terceiro teste"
(integer) 1
127.0.0.1:6379> SADD meu-set "quarto teste"
(integer) 1
127.0.0.1:6379> ZADD meu-sorted-set 1 "quinto teste"
(integer) 1
127.0.0.1:6379> TYPE minha-string
string
127.0.0.1:6379> TYPE minha-lista
list
```

```
127.0.0.1:6379> TYPE meu-hash
hash
127.0.0.1:6379> TYPE meu-set
set
127.0.0.1:6379> TYPE meu-sorted-set
zset
```

No exemplo que acabamos de ver, o primeiro comando tenta listar todas as chaves armazenadas no *Redis*. Em seguida, utilizamos comandos para armazenar uma chave de cada tipo de dados (String, List, Hash, Set e Sorted Set), para que, na sequência, possamos executar o comando `TYPE` em cada uma dessas chaves criadas. O único valor retornado pelo `TYPE` que se diferencia um pouco é o `zset`, que significa que a chave é do tipo *Sorted Set*.

6.4 Próximos passos

Chegamos ao fim dos capítulos dedicados a apresentar o Redis, seus tipos de dados suportados e algumas situações em que podemos aplicá-los. Conhecemos o tipo de dados *Sorted Set* e também aprendemos a identificar qual o tipo de dado de uma chave.

No próximo capítulo, vamos conhecer outros recursos que são de extrema importância e que podem nos ajudar a resolver diversos tipos de problemas. Os recursos que iremos ver a seguir são *Pub-Sub*, *Pipeline*, *Transaction* e *Lua scripting*.

Capítulo 7

O que mais o Redis pode fazer

Nos capítulos anteriores conhecemos todos os tipos de dados disponíveis no Redis e alguns exemplos de uso para cada tipo. Já ficou bem claro que o principal uso do Redis é como um *Key/Value store*, mas existem outros recursos do Redis de que precisamos. Esse capítulo é exatamente para conhecermos mais alguns recursos interessantes que o Redis fornece.

7.1 Enviando mensagens com PUB-SUB

PUBLISH-SUBSCRIBE ou PUB-SUB é um padrão para troca de mensagens e um recurso muito interessante e utilizado para comunicação assíncrona. Esse padrão é formado por *publishers*, *subscribers*, *messages* e *channels*. Embora à primeira vista pareça complicado, seu uso é bem simples. As mensagens ou *messages* são enviadas pelos *publishers* para um determinado *channel*, os *subscribers* ficam "ouvindo" um channel e, quando os publishers enviam uma

mensagem para este channel, todos os subscribers que estiverem ouvindo (*listening*) este channel receberão a mensagem [4].

Subscribers podem ouvir mais de um channel e somente receber mensagens de channels que eles estão ouvindo. Publishers não podem enviar uma mensagem diretamente para um Subscriber e também não têm informação de quais são os seus subscribers Esse desacoplamento entre publishers e subscribers facilita e permite escalar sistemas de forma mais fácil.

Vamos fazer uma analogia ao mundo real usando como exemplo uma estação de rádio. As estações de rádios são os channels, os locutores da rádio são os publishers, as músicas transmitidas são as messages e as pessoas (ouvintes) que ouvem as rádios são os subscribers. Repare que, nesse exemplo, os locutores não têm conhecimento sobre as pessoas ou o que elas estão ouvindo, eles apenas se encarregam de tocar as músicas. Veja a seguir uma ilustração demonstrando como isso funciona:

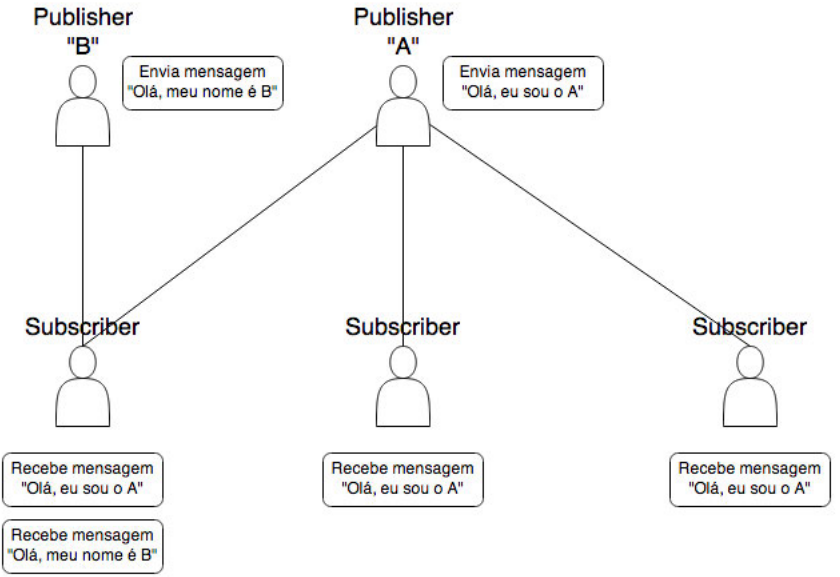


Figura 7.1: Exemplo de Pub-Sub

Com base na analogia de uma estação de rádio, vamos criar uma aplicação

que tenha "ouvintes" de determinadas estações de rádio. Veja o código Java a seguir:

```
public class Radio {

    static class Ouvinte {
        private final String nome;
        private final String estacao;

        private AtomicInteger musicasOuvidas =
            new AtomicInteger(0);

        private static final Executor threadPool =
            Executors.newFixedThreadPool(3);

        Ouvinte(String nome, String estacao) {
            this.nome = nome;
            this.estacao = estacao;
        }

        public void ouvirEstacao() {
            Runnable ouvinte = new Runnable() {

                public void run() {
                    Jedis jedis = new Jedis("localhost");
                    JedisPubSub jedisPubSub = new JedisPubSub() {

                        @Override
                        public void onUnsubscribe(String channel,
                                int subscribedChannels) {
                            System.out.println(
                                String.format(
                            "%s deixou de escutar a estação %s",
                                    nome,
                                    channel
                                )
                            );
                        }
```

```
@Override
public void onSubscribe(String channel,
        int subscribedChannels) {
    System.out.println(
        String.format(
    "%s começou a escutar a estação %s",
            nome,
            channel
        )
    );
}

@Override
public void onMessage(String channel,
        String message) {
    System.out.println(
        String.format(
    "%s está ouvindo %s na estação %s",
            nome,
            message,
            channel
        )
    );

    if(musicasOuvidas.addAndGet(1) >= 3){
        this.unsubscribe();
    }
}

@Override
public void onPUnsubscribe(String pattern
        ,int subscribedChannels) {
}

@Override
public void onPSubscribe(String pattern,
        int subscribedChannels) {
}
```

```
                        @Override
                        public void onPMessage(String pattern,
                            String channel, String message) {
                        }
                    };

                    jedis.subscribe(jedisPubSub, estacao);
                }

            };

            threadPool.execute(ouvinte);
        }

    };

    public static void main(String[] args) {
        Ouvinte rodrigo = new Ouvinte("Rodrigo", "punk-rock");
        rodrigo.ouvirEstacao();

        Ouvinte rafael = new Ouvinte("Rafael", "surf-music");
        rafael.ouvirEstacao();

        Ouvinte andressa = new Ouvinte("Andressa", "pop-rock");
        andressa.ouvirEstacao();
    }

}
```

Antes de entendermos esse código, vamos conhecer alguns novos comandos do Redis. O primeiro é o `SUBSCRIBE`, que recebe como argumento o nome de um ou mais channels que serão ouvidos, sendo que o cliente que executar esse comando ficará automaticamente bloqueado esperando mensagens serem enviadas aos channels especificados. Veja a seguir seu uso diretamente no CLI:

```
127.0.0.1:6379> SUBSCRIBE casa-do-codigo
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
```

```
2) "casa-do-codigo"
3) (integer) 1
```

No comando anterior, nós efetuamos o `SUBSCRIBE` no channel de nome `casa-do-codigo`. Como explicado, o cliente ficou automaticamente bloqueado, ou seja, não podemos mais efetuar nenhum outro comando enquanto o `SUBSCRIBE` não for cancelado (executando as teclas `Ctrl-C`).

O próximo comando que iremos ver é o `PSUBSCRIBE`. Ele é similar ao comando `SUBSCRIBE`, mas ao invés de receber o nome de um ou mais channels, ele recebe como argumento um *pattern* que serve para especificar todos os channels que se enquadrarem nesse pattern. O formato do pattern usado nesse comando é o mesmo formato da seção 3.2, para ficar mais claro, imagine que tivéssemos três channels chamados `casa-do-codigo`, `casa-do-jogo` e `casa-do-filme`. Poderíamos realizar o seguinte comando para "ouvir" mensagens enviadas para estes três canais:

```
127.0.0.1:6379> PSUBSCRIBE casa-do-*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "casa-do-*"
3) (integer) 1
```

Agora que conhecemos os comandos `SUBSCRIBE` e `PSUBSCRIBE`, vamos voltar ao código Java mostrado anteriormente. Nele, foi criada uma classe chamada `Ouvinte` que possui um nome (atributo `nome`), a estação de rádio que ele vai ouvir (atributo `estacao`) e a quantidade de músicas que ele já ouviu (atributo `musicasOuvidas`). Esse objeto possui também um único método `ouvirEstacao`, e é nele que está toda a programação em si. Sempre que este método é executado, um nova `Thread` [10] é iniciada, e dentro dela executamos o comando `SUBSCRIBE` do Redis.

O Jedis espera uma implementação da classe abstrata `JedisPubSub`. Essa implementação possui métodos ou *callbacks* que serão executados sempre que uma ação ocorrer no Redis. Atualmente, esse classe fornece callbacks para quando um cliente se inscreve (*subscribe*) ou deixa de ouvir (*unsubscribe*) um channel tanto usando o comando `SUBSCRIBE` quanto para o `PSUBSCRIBE` do Redis. Essa classe também fornece um callback para avisar quando uma mensagem for pública no Redis.

Na nossa implementação, apenas os callbacks `onSubscribe`, `onUnsubscribe` e `onMessage` foram implementados, sendo que nos dois primeiros apenas fazemos a impressão no console de uma mensagem referente ao fato de o cliente estar ou não ouvindo determinado channel. Já no último, no método `onMessage`, além de fazermos a impressão no console da mensagem recebida, fazemos um controle da quantidade de músicas ouvidas para que quando esse valor atingir três músicas, o cliente realize o `unsubscribe` do channel.

Uma parte importante do código, que não diz respeito diretamente ao Redis, é que utilizamos *threads*[10, 20] nesse método para podermos ter mais de uma instância do cliente do Redis escutando um determinado channel. Repare que no método `main` criamos três instâncias da classe `Ouvinte` e definimos um nome e estação para cada uma das instâncias. Ao executarmos esse código teremos um resultado como o seguinte:

```
Andressa começou a escutar a estação pop-rock
Rafael começou a escutar a estação surf-music
Rodrigo começou a escutar a estação punk-rock
```

Repare que, como a execução do método `ouvirEstacao` de cada instância é feita por uma outra thread, a ordem das mensagens pode variar a cada execução. Já temos os ouvintes aptos a receber suas músicas com esse código Java em execução, agora vamos abrir um CLI que será o locutor das estações de rádio (`PUBLISHER`) e o responsável por publicar as mensagens (messages) nos channels. Vejamos isso agora:

```
127.0.0.1:6379> PUBLISH punk-rock "Rancid - Indestructible"
(integer) 1

127.0.0.1:6379> PUBLISH surf-music
    "Jack Johnson - Wasting Time"
(integer) 1

127.0.0.1:6379> PUBLISH pop-rock "Colbie Caillat - Oxygen"
(integer) 1

127.0.0.1:6379> PUBLISH punk-rock
```

```
    "Lars Frederiksen - Switchblade"
(integer) 1

127.0.0.1:6379> PUBLISH surf-music "Jason Mraz - Lucky"
(integer) 1

127.0.0.1:6379> PUBLISH pop-rock "Missy Higgins - Sugarcane"
(integer) 1

127.0.0.1:6379> PUBLISH punk-rock
    "The Sex Pistols - God Save the Queen"
(integer) 1

127.0.0.1:6379> PUBLISH surf-music
    "Ben Harper - Diamonds On The Inside"
(integer) 1

127.0.0.1:6379> PUBLISH pop-rock "Norah Jones - Wish I Could"
(integer) 1
```

O resultado emitido pela nosso código Java foi:

```
Rodrigo está ouvindo Rancid -
    Indestructible na estação punk-rock

Rafael está ouvindo Jack Johnson -
    Wasting Time na estação surf-music

Andressa está ouvindo Colbie Caillat - Oxygen na estação pop-rock

Rodrigo está ouvindo Lars Frederiksen - Switchblade na estação
punk-rock

Rafael está ouvindo Jason Mraz - Lucky na estação surf-music

Andressa está ouvindo Missy Higgins -
    Sugarcane na estação pop-rock

Rodrigo está ouvindo The Sex Pistols -
    God Save the Queen na estação punk-rock
```

```
Rodrigo deixou de escutar a estação punk-rock

Rafael está ouvindo Ben Harper -
    Diamonds On The Inside na estação surf-music

Rafael deixou de escutar a estação surf-music

Andressa está ouvindo Norah Jones -
    Wish I Could na estação pop-rock

Andressa deixou de escutar a estação pop-rock
```

Repare que, conforme as mensagens (músicas) são enviadas a cada channel, os ouvintes são notificados automaticamente e assim podem realizar alguma ação com a mensagem recebida. Utilizar *PUB-SUB* em aplicações não é uma tarefa nova, atualmente existem deferentes ferramentas e plataformas que fornecem esse tipo de envio de mensagens. Entre elas, temos projetos que dão suporte ao JMS (*Java Message Service* [12]) como ActiveMQ e HornetQ e outros como o Apache Kafka e RabbitMQ.

É importante salientar que o uso do Redis como um serviço de *PUB-SUB* possa parecer muito simples quando comparado com outros projetos ou até mesmo que não possua as funcionalidades necessárias, mas na verdade isso é uma das principais características, assim como sua performance, que têm feito com que muitos desenvolvedores estejam usando o Redis para esse propósito.

7.2 Enviando múltiplos comandos com Pipeline

No capítulo 2, nós vimos como o cliente e o servidor do Redis interagem entre si. Resumidamente, é através do envio de um comando e aguardo da resposta pelo cliente enquanto o servidor processa o comando e envia a resposta.

Esse processo ou etapas ocorrem por meio de uma conexão TCP de rede, que, para efeito de exemplo, vamos categorizar que podem ser rápida (cliente e servidor no mesmo computador), normal (cliente e servidor em uma rede interna) e lenta (conexão realizada via internet). Independente de em qual

categoria o cliente e servidor se encontrem, existe um tempo para os pacotes irem do cliente ao servidor e depois voltarem com uma resposta ao cliente. Esse tempo é chamado de **RTT** ou *Round-trip delay time* [5] e ele impacta diretamente no desempenho dos comandos enviados do cliente ao servidor do Redis.

Para entendermos melhor, imagine que o *RTT* seja de 250 milissegundos. Então, se enviarmos quatro comandos ao Redis em sequência, isso irá levar 1 segundo desde o momento de envio do primeiro comando até o recebimento da resposta do último comando. Agora se enviarmos esses quatro comandos de uma única vez, o tempo irá diminuir bastante, pois precisaremos de apenas um envio e uma resposta para processar os quatro comandos. Basicamente, é isso que fazemos ao utilizar *Pipeline* no Redis: podemos enviar diversos comandos para serem processados no Redis e, em um única resposta, saber o resultado de cada comando.

Vamos entender melhor esse processo com um exemplo, veja o seguinte código:

```
long tempoInicial = System.currentTimeMillis();
Jedis jedis = new Jedis("localhost");

for (int i = 1; i <= 100000; i++) {
    jedis.set("chave-" + i, String.valueOf(i));
}

long tempoFinal = System.currentTimeMillis();
System.out.println(
    String.format(
        "Tempo total: %.2f segundos",
        ((tempoFinal - tempoInicial) / 1000.0)
    )
);
```

Repare que esse código é bem simples. Basicamente, ele insere 100.000 chaves no Redis utilizando o comando `SET` e calcula o tempo em segundos que a execução de todo o procedimento durou. Ao executar esse código, obtive o seguinte resultado:

```
Tempo total: 10.06 segundos
```

O resultado pode variar de execução para execução. No meu computador, a execução desse código variou entre 9 a 11 segundos. Agora vamos ver uma nova versão desse código utilizando *Pipeline*, observe o código a seguir:

```
long tempoInicial = System.currentTimeMillis();
Jedis jedis = new Jedis("localhost");
Pipeline pipeline = jedis.pipelined();

for (int i = 1; i <= 100000; i++) {
    pipeline.set("chave-" + i, String.valueOf(i));
}

pipeline.sync();

long tempoFinal = System.currentTimeMillis();
System.out.println(
    String.format(
        "Tempo total: %.2f segundos",
        ((tempoFinal - tempoInicial) / 1000.0)
    )
);
```

Note que agora temos um novo objeto chamado *Pipeline*, que foi obtido de uma instância do cliente do Redis. Repare também que agora os comandos `SET` são enviados para esse objeto Pipeline e não mais diretamente para o cliente do Redis (Jedis), como aconteceu no primeiro exemplo.

O que acontece aqui, na verdade, é que quando invocamos o método `set` do objeto `Pipeline`, o comando `SET` não é enviado diretamente para o Redis, mas sim armazenado para ser enviado de uma única vez com todos os outros comandos. Esse envio é feito quando invocamos o método `sync` do objeto Pipeline. Vejamos agora o seu resultado:

```
Tempo total: 0.69 segundos
```

Que diferença! Com isso podemos notar que, ao utilizarmos Pipeline nesse exemplo, conseguimos uma redução de mais de 9 segundos no tempo de processamento dos 100.000 comandos enviados ao Redis.

Agora que já sabemos como o Pipeline funciona no Redis, precisamos nos questionar em qual cenário o uso de Pipeline se enquadraria. Bom, um caso muito comum em que seu uso se encaixa perfeitamente é para execução de comandos ou manipulação de dados em massa ou em *batch*.

7.3 Utilizando transações no Redis

Podemos definir transações como uma forma de executar uma série de comandos atomicamente garantindo que, ou todos os comandos serão executados, ou nenhum será. Para um desenvolvedor com um pouco de experiência, o uso de transações não é nenhuma novidade quando temos que utilizar bancos de dados convencionais como Oracle e MySQL. Embora o Redis não se enquadre nesse tipo de banco de dados, ele também possui suporte a transações.

No Redis, uma transação é iniciada por um cliente quando este envia o comando `MULTI` ao servidor. Uma transação é iniciada assim que o servidor recebe esse comando e todos os comandos enviados posteriormente pelo cliente são enfileirados para que sejam executados sequencialmente, ou descartados ao final da transação. Para que todos os comandos de uma transação sejam executados no servidor, o cliente precisa enviar o comando `EXEC`.

Uma característica interessante sobre transações no Redis é que o servidor continuará executando todos os comandos, sem interrupção, mesmo que um ou vários comandos falharem. No final da transação, o servidor envia como resposta uma lista com todos os resultados de cada comando para o cliente.

Para exemplificar melhor o uso de transações, vamos começar criando uma nova chave no Redis, conforme o seguinte comando executado no CLI:

```
127.0.0.1:6379> set numero-de-acessos 0
OK
```

Existe um comando no Redis chamado `GETSET`, que define um novo valor para um chave já existente e retorna o seu valor anterior (antes de ser redefinido pelo comando). Vamos utilizá-lo na chave que acabamos de criar:

```
127.0.0.1:6379> GETSET numero-de-acessos 10
"0"
```

Repare que o valor "o" retornado pelo servidor corresponde ao valor que definimos ao criar a chave. Agora vamos verificar o comando atual da chave:

```
127.0.0.1:6379> GET numero-de-acessos
"10"
```

O novo valor foi definido corretamente pelo comando `GETSET` de forma atômica. Para e pense um pouco sobre ele, repare que ele nada mais é do a composição de dois comandos (`GET` e `SET`). Com isso em mente, vamos simular esse mesmo comportamento, mas agora utilizando transações. Veja o código Java a seguir:

```
public class ExecutarGetSetEmTransacao {

    public String getSet(String chave, String novoValor) {
        Jedis jedis = new Jedis("localhost");
        Transaction transaction = jedis.multi();

        transaction.get(chave);
        transaction.set(chave, novoValor);

        List<Object> resultados = transaction.exec();

        return (String) resultados.get(0);
    }

    public static void main(String[] args) {
        String valorNovo = "20";
        String chave = "numero-de-acessos";

        ExecutarGetSetEmTransacao transacao =
                new ExecutarGetSetEmTransacao();

        String valorAntigo = transacao.getSet(chave, valorNovo);

        System.out.println(
            String.format(
                "O valor antigo da chave %s é %s e o novo é %s",
                chave,
```

```
                valorAntigo,
                valorNovo
            )
        );
    }

}
```

Nesse código, foi criado um método chamado `getSet` para simular o comando nativo do Redis, o `GETSET`. Nesse método, nós executados o método `multi` do cliente Jedis, que retorna uma instância da classe `Transaction`, referente à nossa transação em si. Todos os comandos Redis enviados através desse objeto são enviados ao Redis e executados somente quando este objeto invocar o método `multi`, que já retorna os resultados da transação.

Ainda no método `getSet` do exemplo, repare que o comando `exec` retorna uma lista de objetos e que o método retorna o valor do primeiro resultado. Isso é feito porque os comandos da transação são executados de forma sequencial e o primeiro comando enviado foi o `GET`. Vejamos agora o resultado desse código:

```
O valor antigo da chave numero-de-acessos é 10 e o novo é 20
```

Agora vamos fazer esse exemplo pelo CLI, mas ao invés de executarmos o comando `EXEC`, vamos utilizar o `DISCARD`. Ele cancela a transação e todos os comandos enfileirados nela. Veja o exemplo a seguir:

```
127.0.0.1:6379> GET numero-de-acessos
"20"
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> GET numero-de-acessos
QUEUED
127.0.0.1:6379> SET numero-de-acessos 50
QUEUED
127.0.0.1:6379> DISCARD
OK
127.0.0.1:6379> GET numero-de-acessos
"20"
```

Repare que, ao executarmos o `DISCARD`, nenhum comando enfileirado na transação foi, de fato, realizado. Agora que entendemos o uso de transações no Redis, podemos notar que transações no Redis também poderiam ser utilizadas da mesma forma como fizemos com a inserção de várias chaves na seção 7.2. Embora seja perfeitamente possível utilizar transações para esse propósito, lembre-se que seu desempenho será pior porque, ao enviar um comando para uma transação, esse comando é enfileirado no servidor e isso tem um custo de tempo para ocorrer. Por este motivo, a não ser que a questão de atomicidade dos comandos seja necessária, utilizar Pipeline para esse tipo de uso é mais indicado.

7.4 Executando scripts em Lua

Segundo a definição no próprio site da linguagem: "Lua é uma linguagem de programação de extensão projetada para dar suporte à programação procedimental em geral e que oferece facilidades para a descrição de dados. A linguagem também oferece um bom suporte para programação orientada a objetos, programação funcional e programação orientada a dados. Lua foi planejada para ser utilizada por qualquer aplicação que necessite de uma linguagem de script leve e poderosa. Lua é implementada como uma biblioteca, escrita em C limpo (isto é, no subconjunto comum de ANSI C e C++). [14]"

O suporte a scripts em Lua foi implementado a partir da versão 2.6 do Redis e, segundo Salvatore Sanfilippo, o uso de scripts em Lua é vantajoso basicamente porque com ele você pode fazer muitas coisas de forma atômica, que antes teriam um grande impacto no desempenho [15].

Scripts Lua são executados como comandos implementados em C e, por serem executados de forma atômica, nenhum outro comando pode ser executado no servidor do Redis enquanto o script Lua estiver rodando. Com isso, o script fica livre de *race condititions*. Em compensação, devemos tomar muito cuidado com o que o script irá fazer pois ele pode bloquear o servidor por muito tempo.

Saiba mais sobre Lua

O intuito deste livro não é ensinar a linguagem de programação Lua, por este motivo não irei cobrir aqui nada muito além do básico sobre ela. Mas é extremamente válido que você dedique um tempo para conhecer melhor a linguagem, seja para utilizá-la com o Redis ou não. Você pode obter mais informações sobre Lua através do link:

http://www.lua.org/portugues.html

Vamos partir para uma abordagem um pouco mais prática agora: vamos começar conhecendo o comando `EVAL`. Ele avalia código Lua utilizando um interpretador embutido no Redis. Sua sintaxe é: `EVAL script quantidade-de-argumentos [argumentos...]`. Veja a seguir um exemplo de uso:

```
127.0.0.1:6379> EVAL
    "return 'Qual o sentido da vida? ' .. KEYS[1]" 1 42
"Qual o sentido da vida? 42"
```

O script Lua utilizado nesse comando `EVAL` é bem simples, nós apenas concatenamos (utilizando o operador `..`) uma string "Qual o sentido da vida?" com o argumento (`42`) definido pela variável global `KEYS[1]`. Essa variável global é um array de parâmetros e o tamanho do array é definido pelo segundo argumento fornecido para o comando `EVAL`, que neste exemplo foi o valor 1.

Um script Lua é executado de forma transacional, então é perfeitamente possível fazer tudo que fazemos com transações utilizando scripts Lua. Para demonstrar isso vamos fazer o mesmo exemplo usado na seção 7.3 e recriar o comando `GETSET` utilizando Lua. Veja o código Java a seguir:

```
StringBuilder scriptLua = new StringBuilder();
scriptLua.append("local valor = redis.call('get', KEYS[1])");
scriptLua.append(
    "local resultado = redis.call('set', KEYS[1], KEYS[2])");
scriptLua.append("return valor");
```

```
Jedis jedis = new Jedis("localhost");
String chave = "numero-de-acessos";
String valorNovo = "44";

String valorAntigo =
    (String) jedis.eval(scriptLua.toString(), 2,
                        chave, valorNovo);

System.out.println(
    String.format(
        "O valor antigo da chave %s é %s e o novo é %s",
        chave,
        valorAntigo,
        valorNovo
    )
);
```

Repare que, nesse exemplo, o método `eval` do Jedis utiliza a mesma estrutura de argumentos que usamos no exemplo feito com o comando `EVAL` do Redis. A principal diferença é que o script lua utilizado nesse exemplo executa comandos Redis através da função `redis.call()`. Agora veja o resultado do código anterior:

```
O valor antigo da chave numero-de-acessos é 20 e o novo é 44
```

Para finalizar esta seção, é importante sabermos que com scripts Lua não é recomendado nunca tentar acessar o sistema de arquivo externo ou realizar qualquer tipo de chamada a outro sistema. Isso pode fazer o comando do Redis ficar travado aguardando uma resposta e, em consequência, bloquear o Redis para outros comandos. Um outro ponto importante é que scripts Lua estarão sempre sujeitos a um tempo máximo de execução, sendo que o valor padrão é de cinco segundos (esse valor pode ser modificado) e, após esse tempo, o Redis passa a aceitar outros comandos mas sempre irá retornar o valor `BUSY`.

7.5 Próximos passos

Este capítulo foi muito interessante, conhecemos diversos recursos do Redis e ainda aprendemos a como expandir suas funcionalidades utilizando a linguagem Lua.

No próximo capítulo, vamos começar a entender melhor como administrar e obter informações do estado do Redis para que assim possamos descobrir e solucionar possíveis problemas. Além disso, vamos conhecer diversas dicas de como aproveitar ao máximo tudo que o Redis fornece.

Capítulo 8

Monitorando o Redis

8.1 Como monitorar comandos

Já cobrimos uma boa parte dos comandos possíveis no Redis, agora chegou o momento de aprendermos a analisar e identificar possíveis problemas que podem ocorrer com seu uso. Vamos começar aprendendo a monitorar os comandos que estão sendo processados no Redis. Para realizar essa tarefa, iremos precisar de duas instâncias diferentes do **redis-cli** (cliente nativo do Redis).

Com o servidor do Redis funcionando, inicie uma instância do CLI e execute o comando `MONITOR`. Repare que, ao executá-lo, a conexão ficará bloqueada no mesmo instante. Veja a seguir o exemplo:

```
127.0.0.1:6379> MONITOR
OK
```

Agora vamos iniciar uma outra instância do CLI e executar alguns comandos:

```
127.0.0.1:6379> keys *
1) "numero-de-acessos"

127.0.0.1:6379> get numero-de-acessos
"44"

127.0.0.1:6379> set sentido-da-vida 42
OK

127.0.0.1:6379> del sentido-da-vida
(integer) 1
```

O ideal para esse exemplo é manter visíveis as duas instâncias do CLI pois, assim que um comando é executado no Redis, ele aparece na instância que está monitorando o Redis. O resultado do comando `MONITOR` desse exemplo é:

```
1395367119.521877 [0 127.0.0.1:62970] "keys" "*"
1395367134.675322 [0 127.0.0.1:62970] "get" "numero-de-acessos"
1395367167.664193 [0 127.0.0.1:62970]
    "set" "sentido-da-vida" "42"
1395367185.199500 [0 127.0.0.1:62970] "del" "sentido-da-vida"
```

Monitorar os comandos que estão sendo executados no Redis pode ser muito útil para identificar erros nos comandos que estão sendo enviados ao Redis por outras aplicações. Mas lembre-se que, embora monitorar o Redis seja um recurso útil, ele possui um impacto no desempenho do Redis. Vamos ver isso com um rápido teste.

Na pasta `src` além dos comandos `redis-cli` e o `redis-server` existe também o comando `redis-benchmark`, que serve para analisar o desempenho do Redis. Vamos utilizar esse comando para primeiro sem o recurso de monitoração de comandos para ver o resultado. Veja a seguir como fazer isso:

```
./redis-benchmark -c 10 -n 10000 -q
```

Esse comando vai enviar diversos tipos de comando para o Redis, seguindo a configuração que informamos nos argumentos, que são: `-c 10` (10 conexões paralelas), `-n 10000` (10.000 requisições) e `-q` (modo silencioso, exibe apenas o resultado). O resultado foi:

```
PING_INLINE: 66225.17 requests per second
PING_BULK: 75757.58 requests per second
SET: 60975.61 requests per second
GET: 58139.53 requests per second
INCR: 64935.07 requests per second
LPUSH: 70422.53 requests per second
LPOP: 69444.45 requests per second
SADD: 59523.81 requests per second
SPOP: 60240.96 requests per second
LPUSH (needed to benchmark LRANGE): 60606.06 requests per second
LRANGE_100 (first 100 elements): 24449.88 requests per second
LRANGE_300 (first 300 elements): 9980.04 requests per second
LRANGE_500 (first 450 elements): 6622.52 requests per second
LRANGE_600 (first 600 elements): 4918.84 requests per second
MSET (10 keys): 48543.69 requests per second
```

Podemos notar que, por exemplo, o teste conseguiu executar quase 61 mil comandos `SET` por segundo e quase 65 mil comandos `INCR` por segundo. Agora vamos iniciar uma instância do CLI e executar o comando `MONITOR` nela. Executaremos o mesmo comando `redis-benchmark` com os mesmos comandos. Veja a seguir o resultado:

```
PING_INLINE: 40650.41 requests per second
PING_BULK: 41666.67 requests per second
SET: 31545.74 requests per second
GET: 33003.30 requests per second
INCR: 33112.59 requests per second
LPUSH: 32786.88 requests per second
LPOP: 37313.43 requests per second
SADD: 30959.75 requests per second
SPOP: 40000.00 requests per second
LPUSH (needed to benchmark LRANGE): 34965.04 requests per second
LRANGE_100 (first 100 elements): 15503.88 requests per second
LRANGE_300 (first 300 elements): 9505.70 requests per second
```

```
LRANGE_500 (first 450 elements): 6738.54 requests per second
LRANGE_600 (first 600 elements): 4557.89 requests per second
MSET (10 keys): 11061.95 requests per second
```

Olha que interessante, agora conseguiu executar apenas quase 41 mil comandos `SET` e 33 mil comandos `INCR` por segundo, e você pode notar que houve uma grande diminuição de requisições por segundo em todos testes feitos pelo `redis-benchmark`. Com isso, fica claro que monitorar os comandos que estão sendo executados no Redis pode ser um recurso muito útil, mas temos que utilizá-lo apenas quando necessário, principalmente em ambientes de produção.

8.2 Obtendo informações do servidor

Muitas vezes, além de utilizarmos determinada tecnologia, precisamos também atuar como um administrador dessa tecnologia para podermos avaliar através de dados e estatísticas o seu comportamento e, assim, conseguirmos identificar algum tipo de problema ou mesmo saber como o seu uso está sendo feito.

Com o Redis não é diferente, pois embora seja uma tecnologia muito simples de utilizar e até mesmo de configurar dependendo de como será usado, ainda precisamos assegurar que as outras aplicações que o estão usando estejam fazendo isso da forma correta. Felizmente, o Redis nos fornece um comando muito útil para realizar esta tarefa.

O comando que veremos agora é o `INFO`, que retorna informações e estatísticas sobre o servidor de uma forma simples e fácil de entender. Vamos ver seu uso a seguir:

```
127.0.0.1:6379> info

# Server
redis_version:2.8.7
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:3d45c4c63ce96089
redis_mode:standalone
```

```
os:Darwin 13.1.0 x86_64
arch_bits:64
multiplexing_api:kqueue
gcc_version:4.2.1
process_id:45020
run_id:1306d00fe1f7ed9312b897bace78720add3264eb
tcp_port:6379
uptime_in_seconds:39
uptime_in_days:0
hz:10
lru_clock:1139377
config_file:

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:1002704
used_memory_human:979.20K
used_memory_rss:1736704
used_memory_peak:968624
used_memory_peak_human:945.92K
used_memory_lua:33792
mem_fragmentation_ratio:1.73
mem_allocator:libc

# Persistence
loading:0
rdb_changes_since_last_save:0
rdb_bgsave_in_progress:0
rdb_last_save_time:1395514058
rdb_last_bgsave_status:ok
rdb_last_bgsave_time_sec:-1
rdb_current_bgsave_time_sec:-1
aof_enabled:0
aof_rewrite_in_progress:0
```

```
aof_rewrite_scheduled:0
aof_last_rewrite_time_sec:-1
aof_current_rewrite_time_sec:-1
aof_last_bgrewrite_status:ok

# Stats
total_connections_received:1
total_commands_processed:0
instantaneous_ops_per_sec:0
rejected_connections:0
sync_full:0
sync_partial_ok:0
sync_partial_err:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0

# Replication
role:master
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0

# CPU
used_cpu_sys:0.05
used_cpu_user:0.03
used_cpu_sys_children:0.00
used_cpu_user_children:0.00

# Keyspace
db0:keys=2,expires=0,avg_ttl=0
```

Uau, quanta informação interessante! Repare que as informações apresentadas estão separadas em blocos (cada bloco inicia com o caractere `#`). Vamos conhecer a seguir um pouco mais sobre cada bloco e as principais informações que podemos obter a partir deles.

Server

Apresenta informações gerais sobre o Servidor Redis.

- **redis_version**: versão do Redis;
- **os**: sistema operacional onde o servidor está sendo executado;
- **arch_bits**: arquitetura do sistema operacional (32 ou 64 bits);
- **multiplexing_api**: mecanismo de *event loop* usado pelo Redis;
- **gcc_version**: versão do compilador GCC usado para compilar o Redis;
- **process_id**: PID ou id do processo do servidor Redis;
- **tcp_port**: porta TCP em que ele está sendo executado;
- **uptime_in_seconds**: tempo de execução do servidor em segundos;
- **uptime_in_days**: tempo de execução do servidor em dias;
- **config_file**: arquivo de configuração utilizado pelo servidor.

Clients

Apresenta informações sobre os clientes conectados ao servidor.

- **connected_clients**: número de clientes conectados ao servidor;
- **blocked_clients**: número de clientes com comandos pendentes em uma requisição bloqueante (`BLPOP`, `BRPOP`, `BRPOPLPUSH`).

Memory

Apresenta informações sobre o uso ou consumo de memória do servidor.

- **used_memory**: valor total de bytes alocado pelo Redis;
- **used_memory_human**: valor total de bytes alocado pelo Redis utilizando uma escala para facilitar a visualização;
- **used_memory_peak**: valor máximo (em bytes) de memória utilizado pelo Redis;
- **used_memory_peak_human**: valor máximo (em bytes) de memória utilizado pelo Redis utilizando uma escala para facilitar a visualização;
- **used_memory_lua**: valor total de bytes usada pela engine Lua.

Persistence

Apresenta informações referentes à forma de persistência (RDB ou AOF 9.3) dos dados.

- **loading**: flag indicando se o carregamento de um arquivo de *dump* está ocorrendo;
- **rdb_changes_since_last_save**: quantidade de mudanças que ocorreram desde o último *dump*;
- **rdb_bgsave_in_progress**: flag indicando se um *RDB save* está ocorrendo;
- **rdb_last_save_time**: *timestamp* do último *RDB save* feito com sucesso;
- **rdb_last_bgsave_status**: status da última operação de *RDB save*;
- **aof_enabled**: flag indicando que uma operação *AOF logging* está ativada;
- **aof_rewrite_in_progress**: flag indicando se uma operação de *AOF rewrite* está ocorrendo;

- **aof_last_rewrite_time_sec**: tempo de duração em segundos da última operação de *AOF rewrite*;
- **aof_last_bgrewrite_status**: status da última operação de*AOF rewrite*.

Stats

Apresenta estatísticas gerais sobre o servidor.

- **total_connections_received**: número total de conexões que o servidor aceitou;
- **total_commands_processed**: número total de comandos processados pelo servidor;
- **instantaneous_ops_per_sec**: número de comandos processados a cada segundo pelo servidor;
- **rejected_connections**: número de conexões rejeitadas pelo servidor devido à configuração de um número máximo de clientes (`maxclients`);
- **expired_keys**: número total de chaves expiradas;
- **pubsub_channels**: número global de canais (*channels*) Pub/Sub que possuem clientes (*subscriptions*);
- **latest_fork_usec**: tempo de duração em microssegundos da última operação de *fork*.

Replication

Apresenta informações de replicação dos servidores Master/Slave. As informações apresentadas a seguir correspondem a um servidor Master do Redis.

- **role**: o valor pode ser “master” ou “slave”. Se a instância do servidor não for “slave”, de nenhuma outra instância ela será “master”, caso contrário será “slave";

- **connected_slaves**: número de "slaves" conectados ao servidor;
- **repl_backlog_active**: flag que define se uma operação de replicação está em andamento.

CPU

Apresenta informações de consumo de CPU do Redis.

- **used_cpu_sys**: consumo de CPU no que diz respeito ao sistema (Kernel) pelo servidor do Redis;
- **used_cpu_user**: consumo de CPU no que diz respeito ao usuário (aplicação) pelo servidor do Redis.

Keyspace

Apresenta informações sobre cada *database* 9.4 do Redis.

- **dbXXX:keys=XXX,expires=XXX,avg_ttl=XXX**: informa que, para determinado database (`db0`), o número de chaves armazenadas (`keys`) são 2 (no nosso exemplo). A quantidade de chaves com tempo de expiração definido (`expires`) é 0 e a média do tempo de expiração dessas chaves (`avg_ttl`) é de 0.

8.3 Algumas dicas de uso

A seguir, vou apresentar uma coleção de dicas de uso do Redis que podem ajudá-lo a evitar problemas e também a melhorar seu desempenho [18]. Vamos conhecê-las a seguir:

Redis 32 bit

Se você conseguir mensurar a quantidade de dados que você pretende armazenar no Redis e esses dados não ultrapassarem 4GB, então nesse caso é recomendado que você utilize uma versão do Redis compilada pra 32 bit. Isso porque, ao usarmos 32 bit, os ponteiros irão possuir a metade do tamanho de

um ponteiro de uma instância do Redis compilada para 64 bit e, com isso, o Redis irá utilizar menos memória para armazenar cada chave.

Mas lembre-se que o uso máximo de memória do Redis será limitado a 4GB ao optar por uma versão 32 bit. Arquivos `RDB` e `AOF` 9.3 são compatíveis em ambos os tipos de instâncias (32 e 64 bit), por isso você pode trocar o tipo de instância sem ter nenhum tipo de problema com estes arquivos.

Use hashes quando possível

Imagine que estamos armazenando os dados de um usuário em três diferentes chaves (`user:name:XXX`, `user:email:XXX` e `user:phone:XXX`, onde `XXX` representa o código do usuário), quando cada chave dessa é armazenada no Redis, essas chaves irão possuir metadados adicionais relacionados a elas.

Agora quando convertemos essas chaves em um único *hash* e adicionamos os dados (*name*, *email* e *phone*) como campos (*fields*) do hash, esses campos são armazenados em forma de dicionário como texto puro sem nenhum metadado adicional. Sendo assim, conseguimos diminuir o consumo de memória do Redis, otimizar e organizar os dados de uma forma mais eficiente.

Comprima seus dados

Lembre-se que, quanto mais conteúdo armazenado em uma chave, maior será a memória utilizada pelo Redis. Por este motivo, utilize alguma forma eficiente de compactar os dados antes de armazená-los no Redis. Por exemplo, imagine que você esteja armazenando dados em forma de `JSON` no Redis. Uma possível alternativa para comprimir seus dados seria utilizar uma biblioteca como a *MessagePack* que transforma `JSON` para um formato de binário serializável. Você pode saber sobre o *MessagePack* em:

http://msgpack.org/

Utilize nomes de chave curtos

Assim como devemos diminuir o tamanho do valor armazenado em uma chave sempre que possível, também devemos aplicar essa regra para o nome das chaves, pois eles também representam uma parte do consumo de memó-

ria. Por exemplo, veja as chaves que nós utilizamos no decorrer do livro:

```
resultado:04-09-2013:megasena
sessao:usuario:1962
jogador:0001:codigo
```

Essas mesmas chaves poderiam ser refeitas da seguinte forma para diminuir o seu tamanho e, consequentemente, o número de memória do Redis:

```
res:04092013:mega
s:u:1962
j:1:c
```

Defina um tempo de expiração

Defina um tempo de expiração para as chaves do Redis sempre que possível, pois assim você garante que terá em memória somente dados relevantes. Você pode definir o tempo de expiração de uma chave ao criá-la (é a melhor forma) ou posteriormente com comandos como `EXPIRE`, `EXPIREAT`, `PEXPIRE` e `PEXPIREAT`.

Use comandos multiargumentos

Nós já vimos que o desempenho ao utilizar *Pipeline* 7.2 é melhor do que enviar diversas requisições ao servidor do Redis. Para alguns comandos, isso também pode ser feito utilizando comandos *multiargumentos*. Vejamos a seguir alguns exemplos de alternativas para determinados comandos:

- **SET**: se você precisa definir várias chaves de uma única vez, você pode utilizar o comando `MSET` uma única vez ao invés de executar diversas vezes o comando *SET*;
- **GET**: o mesmo vale para o `GET`, seu comando alternativo para obter várias chaves em uma única requisição é o comando `MGET`;
- **HGET**: utilize o comando `HMGET` para obter os campos (*fields*) de um hash ao invés de realizar vários comandos `HGET`.

Identifique queries lentas

O Redis fornece um comando chamado `SLOWLOG GET` que serve para identificar de uma forma rápida e simples queries que demoram mais tempo do que o esperado pelo Redis. Por padrão, o Redis assume o valor de 10000 microssegundos, mas é possível definir um valor diferente no arquivo de configuração do Redis através do parâmetro `slowlog-log-slower-than`.

É importante saber que o tempo de execução que o Redis utiliza para definir se um comando não inclui operações de IO nem o tempo de envio da resposta ao cliente, mas sim apenas o tempo utilizado para executar o comando.

Particione seus dados

Se a quantidade de dados que precisamos armazenar no Redis é muito grande, podemos particioná-los entre várias instâncias do Redis, cada uma em um computador diferente. Com isso podemos ter *databases* maiores utilizando a somatória da memória de vários computadores, além de permitir escalar o poder computacional [17].

As formas de particionamento utilizadas no Redis são:

- **Client side partitioning**: o cliente seleciona a instância onde ele vai ler e escrever uma chave;
- **Proxy assisted partitioning**: o cliente envia uma requisição para um *proxy* que determina para qual instância o comando deve ser processado;
- **Query routing**: o cliente envia uma requisição para uma instância definida de forma randômica, e esta instância assegura e envia o comando para a instância correta.

8.4 Próximos passos

Mais um capítulo que chega ao fim! Neste capítulo, nós aprendemos a obter informações do estado do Redis, descobrir e solucionar possíveis problemas.

Também aprendemos diversas dicas de como melhorar o desempenho do Redis.

O próximo capítulo também é voltado à administração do Redis, mas nele iremos ver informações de como configurá-lo, aplicar recursos de segurança e entendermos como a persistência de dados funciona no Redis.

Capítulo 9

Administrando o Redis

9.1 Utilizando um arquivo de configuração

Configurar o Redis é uma tarefa simples que pode ser feita de três maneiras. Uma é através de um arquivo de configuração também conhecido como `redis.conf`, outra por meio de parâmetros passados para o comando `redis-server` e a última através do comando `CONFIG SET`. É importante sabermos que todos os parâmetros de configuração existentes no arquivo `redis.conf` também podem ser utilizados nas outras duas formas de configuração.

Para utilizar um arquivo de configuração, basta utilizá-lo como parâmetro do comando `redis-server`. Veja a seguir um exemplo de uso do arquivo `redis.conf`:

```
src/redis-server redis.conf
```

Note que o arquivo `redis.conf` poderia ter qualquer outro nome e até mesmo outra extensão. A seguir vamos conhecer os principais parâmetros de configuração existentes no arquivo `redis.conf`. O valor padrão de cada parâmetro estará logo após o parâmetro entre parênteses.

- **daemonize** (*no*): define se o Redis vai ser executado como um *daemon*;
- **pidfile** (*/var/run/redis.pid*): local onde será gerado o arquivo `PID` quando o parâmetro **daemonize** for **yes**;
- **port** (*6379*): a porta TCP que o Redis irá utilizar para aceitar conexões;
- **timeout** (*0*): tempo de inatividade em segundos para que a conexão de um cliente seja fechada. O valor "0" desliga esse comportamento;
- **maxclients** (*10000*): número máximo de clientes conectados;
- **maxmemory** (*0*): valor máximo de memória que o Redis pode utilizar. O valor padrão **0** define que não existe limite;
- **slowlog-log-slower-than** (*10000*): gerar log de queries com tempo de execução maior que 10.000 microssegundos.

Os parâmetros apresentados anteriormente são apenas alguns. Para verificar todos os parâmetros possíveis, consulte o arquivo `redis.conf`, pois ele é muito bem documentado e de simples compreensão.

9.2 Segurança

A proposta do Redis é ser rápido na questão de uso e simples no quesito de configuração e administração. Isso também se reflete quanto à sua forma de lidar com segurança. De uma forma geral, o Redis foi criado para ser utilizado através de conexões confiáveis e em ambientes seguros.

Com isso em mente, uma forma de garantirmos segurança no Redis é fazer com ele seja acessado apenas por clientes (aplicações) conhecidos dentro de um ambiente controlado e confiável. Em outras palavras, que apenas clientes dentro de uma rede interna possam conectar-se ao servidor do Redis e

que o Redis não seja exposto diretamente para internet. Isso pode ser feito utilizando um *firewall* para bloquear acessos externos à porta TCP do Redis.

Fora isso, ainda podemos adicionar mais uma camada de segurança ao Redis por meio de autenticação. O Redis não fornece nada muito elaborado como um perfil para cada tipo de usuário e coisas desse tipo pois isso iria contra a ideia de ser simples. Por isso, a autenticação é feita simplesmente utilizando uma senha que fica armazenada diretamente no seu arquivo de configuração apresentado na seção 9.1.

O parâmetro onde podemos definir a senha é o `requirepass` e após definirmos um senha com ele, todo cliente precisa executar o comando `AUTH` para obter autorização do servidor.

Vamos realizar um pequeno teste. Começaremos editando o arquivo `redis.conf`, que se encontra na pasta raiz da instalação do Redis. Adicionemos o seguinte parâmetro:

```
requirepass casadocodigo
```

Em seguida, vamos iniciar o nosso servidor da seguinte forma:

```
src/redis-server redis.conf
```

E finalmente vamos utilizar o *CLI* para nos conectarmos ao servidor e tentar listar todas as chaves. Veja isso a seguir:

```
src/redis-cli

127.0.0.1:6379> keys *
(error) NOAUTH Authentication required.
```

Note que o cliente conseguiu se conectar ao servidor e tentar executar um comando que não foi aceito por não estar devidamente autenticado. Agora vamos realizar a autenticação e tentar novamente enviar comandos ao Redis.

```
127.0.0.1:6379> AUTH casadocodigo
OK
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> SET minhaChave meuValor
OK
```

Repare que o Redis passa a aceitar comandos enviados pelo cliente após a autenticação ser concretizada.

Um ponto muito importante que devemos ressaltar aqui é que, embora definir uma senha de autenticação eleve a segurança do Redis, se o Redis estiver exposto de forma insegura, essa senha pode ser descoberta por um cliente mal intencionado utilizando um ataque de força bruta em um curto período de tempo, se levarmos em conta que o Redis consegue processar mais 50 mil comandos por segundo (valor que pode variar de computador para computador).

9.3 Persistência dos dados contidos em memória

Um dos principais questionamentos com relação ao Redis armazenar os dados em memória é sobre o que acontece quando ele é finalizado. Os dados em memória são perdidos? Ao reiniciá-lo, os dados são recuperados? Bem, as respostas vão depender de como o Redis foi configurado, mas de antemão a resposta é "Sim", é possível armazenar os dados para garantir a sua durabilidade.

O Redis possui três formas de persistir seus dados, que são: RDB, AOF e ambas. Cada forma funciona diferente da outra, sendo que o Redis permite que ambas sejam usadas em conjunto, assim como é possível desabilitar as duas formas de persistência e manter os dados em memória somente enquanto o servidor do Redis estiver em funcionamento.

Se a forma `AOF` estiver ativa, independente da forma `RDB` também estar ativa ou não, o Redis irá utilizar o arquivo `AOF` por padrão, porque este modo tem uma melhor garantia de durabilidade dos dados.

RDB

RDB é conhecida como a forma mais simples de realizar persistência de dados. Utilizando-a, o Redis persiste os dados de forma assíncrona, armazenando *snapshots* do seus dados em intervalos específicos. Como a persistência ocorre em intervalos específicos, caso ocorra algum problema entre um intervalo e outro, os dados manipulados nesse tempo serão perdidos.

O Redis vem com a persistência *RDB* ativada por padrão. Os parâmetros de configuração e seus valores padrões que vêm no arquivo `redis.conf` são:

- **save** *300 10*: define quando o Redis irá realizar a persistência dos dados (*snapshotting*) em memória no disco. Nesse exemplo, o Redis irá persistir os dados a cada 5 minutos (300 segundos) se pelo menos 10 chaves sofreram mudanças. É possível adicionar várias regras de `save` em um mesmo arquivo de configuração;
- **dbfilename** *dump.rdb*: nome do arquivo usado pelo Redis para persistir seus dados no formato RDB;
- **dir** *./*: local onde o arquivo do parâmetro **dbfilename** será criado;
- **stop-writes-on-bgsave-error** *yes*: caso ocorra algum problema com o processo de armazenar os dados em disco, o Redis irá parar de aceitar requisições de escrita. Esse recurso pode ser desabilitado mudando o valor desse parâmetro para **"no"**.
- **rdbcompression** *yes*: faz com as *strings* sejam comprimidas utilizando o algoritmo de compressão *LZF*. Essa compressão pode ser desabilitada, mas saiba que ao fazer isso automaticamente o tamanho dos dados armazenados no Redis será maior;
- **rdbchecksum** *yes*: faz com que o Redis adicione um CRC64 (*cyclic redundancy check*) checksum de 65 bits no final do arquivo. Isso ajuda a tornar a persistência mais resistente a problemas de corrupção de dados.

AOF

Diferente do RDB que armazena os dados, o AOF guarda um log de cada operação de escrita executada e armazena esses logs em um único arquivo. Desta forma, quando o Redis for reiniciado, os dados serão reconstruídos a partir desse arquivo de log. Essa forma de persistência é mais indicada para situações em que nenhum dado pode ser perdido em caso de desastre, mas em contrapartida isso pode ter impacto no desempenho do Redis.

Vamos conhecer os parâmetros de configuração e seus valores padrões contidos no arquivo `redis.comf` para persistência em AOF:

- **appendonly** *no*: ao definir o valor desse parâmetro como **yes**, a persistência em *AOF* é ativada;
- **appendfilename** *"appendonly.aof"*: nome do arquivo usado pelo Redis para persistir seus dados no formato AOF;
- **appendfsync** *everysync*: esse parâmetro define como a chamado ao método `fsync()` será feita. Essa chamada é responsável por "dizer" ao sistema operacional para escrever os dados no disco em vez de aguardar e armazenar mais dados na saída de buffer. Existem três possíveis valores para esse parâmetro que são: **always**, **everysec** e **no**. O valor **always** define que o `fsync()` seja chamado a cada escrita, o valor **everysec** define que o `fsync()` seja chamado a cada segundo e **no** define que o `fsync()` nunca seja chamado;
- **no-appendfsync-on-rewrite** *no*: este parâmetro deve ser ativado (*yes*) somente se houver algum problema de latência, caso contrário o valor padrão (*no*) é mais recomendado por questões de durabilidade dos dados.

Backup

Realizar *backup* dos dados contidos no Redis em geral é uma tarefa bem simples, para isso basta habilitar a persistência RDB e realizar uma cópia do arquivo definido no parâmetro de configuração `dbfilename`. Você não precisa se preocupar com nada ao fazer *backup* do arquivo enquanto o Redis estiver sendo executado, pois o Redis nunca escreve diretamente nesse arquivo, mas sim em um outro arquivo temporário, que é renomeado para o arquivo RDB quando a persistência estiver completa.

9.4 Definindo o banco de dados

O Redis também possui o conceito de *database* assim como um banco de dados tradicional, porém ele não faz referência a um database pelo seu nome,

mas sim através de números. Por padrão, o Redis sempre irá utilizar o database 0 (zero) caso nenhum outro seja especificado. O Redis também já deixa disponíveis outros 15 databases (16 no total) para serem utilizados. Esse limite pode facilmente ser alterado no arquivo de configuração do Redis (`redis.conf`) através do parâmetro `databases`.

O comando para especificar o database a ser utilizado é o **SELECT**. Ele sempre recebe um número referente ao database. Sempre que uma nova conexão é iniciada, o Redis automaticamente define o database 0 como o ativo.

Vamos ver como utilizar o comando **SELECT** via CLI:

```
redis 127.0.0.1:6379> SELECT 8
OK
redis 127.0.0.1:6379[8]>
```

Repare que depois de selecionarmos o database 8 no código anterior, o CLI exibe **[8]** no final do seu prompt para informar que o database atual agora é o 8. Quando nenhum for exibido pelo CLI, significa que o padrão (0) é o atual.

E agora vamos ver como utilizar o comando **SELECT** via Java:

```
Jedis jedis = new Jedis("localhost");
String resultado = jedis.select(8);

System.out.println(resultado); // imprime OK
System.out.println(jedis.getDB()); // imprime 8
```

Uma instância da classe `Jedis` é tudo que precisamos para enviar comando para o Redis através de uma aplicação Java. No exemplo anterior, ao instanciar a classe Jedis, passamos como parâmetro ao construtor o local onde o Redis está sendo executado, que no nosso caso é na máquina local (localhost). A biblioteca Jedis utiliza a mesma nomenclatura dos comandos disponíveis no CLI. Ao executar o comando `select`, a instância do Jedis retorna o mesmo resultado de quando executando o comando `SELECT` via CLI. o Método `getDB()` retorna o número do banco de dados atual utilizado.

Como "limpar" um database

Em determinadas ocasiões, é necessário apagarmos todas as chaves armazenadas em um database. No Redi,s essa tarefa é facilmente realizada utilizando o comando `FLUSHDB`. Vejamos seu uso no exemplo a seguir:

```
127.0.0.1:6379> KEYS *
(empty list or set)

127.0.0.1:6379> SET minhaChave meuValor
OK

127.0.0.1:6379> HSET meuHash meuCampo meuValor
(integer) 1

127.0.0.1:6379> KEYS *
1) "meuHash"
2) "minhaChave"

127.0.0.1:6379> FLUSHDB
OK

127.0.0.1:6379> keys *
(empty list or set)
```

Note que no primeiro comando `KEYS *` o database estava vazio. Em seguida, incluímos duas novas chaves através dos comandos `SET` e `HSET`. Depois foi realizado o comando `FLUSHDB`, que apagou todas as chaves do database corrente e isso foi comprovado através do último comando `KEYS *` que nos mostrou que o database já estava vazio novamente.

> **Comando FLUSH o quê?**
>
> É muito importante não confundir o comando `FLUSHDB` com o comando `FLUSHALL`, pois o primeiro apaga todas as chaves **apenas** do database atual (corrente). Já o comando `FLUSHALL` apaga as chaves de **todos** os databases disponíveis no Redis.

9.5 Próximos passos

Agora já temos um bom nível de conhecimento para administrar uma instância do Redis, analisá-la e configurá-la conforme a nossa necessidade.

Mas o que faremos quando precisarmos de mais de uma instância do Redis? Como iremos replicar seus dados? Como resolver a questão de alta disponibilidade? Essas e outras informações nós iremos conhecer no próximo capítulo.

Capítulo 10

Gerenciando várias instâncias do Redis

10.1 Replicação

A replicação feita pelo Redis ocorre através do processo de replicação *master-slave*, no qual uma ou mais instâncias do tipo *slave* nada mais são do que uma cópia idêntica dos dados contidos na instância do tipo *master*. Veja a seguir algumas informações importantes a respeito de replicação no Redis:

- O processo de replicação é feito de forma assíncrona;
- Uma instância *master* pode ter múltiplas instâncias *slaves*;
- Uma instância *slave* é capaz de receber conexões de outros *slaves* e assim torna-se *master* apenas desses *slaves* (similar a uma estrutura de grafos

10.1);

- O processo de replicação dos dados é feito de forma não bloqueante tanto para a instância *master* quanto para a(s) *slave(s)*. Com isso, o *master* continua processando comandos normalmente enquanto a replicação é feita.

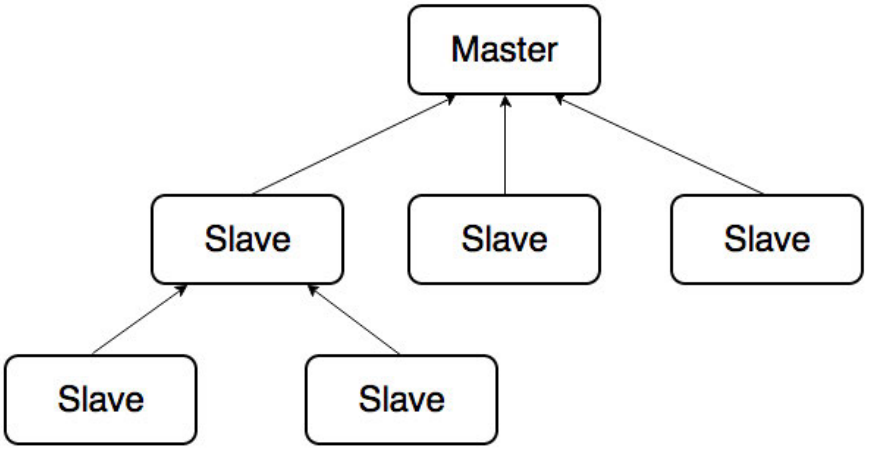


Figura 10.1: Replicação master-slave no Redis

Utilizar replicação de dados é muito útil quando precisamos garantir escalabilidade de uma aplicação, pois podemos, por exemplo, deixar que aplicação realize comandos de escrita na instância master enquanto os comandos de **leitura** são processados por um ou mais slaves. Com uma abordagem desse tipo conseguimos distribuir um pouco da carga que seria direcionada apenas para o master entre os slaves. Uma outra utilidade para replicação é quando precisamos ter redundância de dados.

Configuração master-slave

Você deve estar ansioso para realizar logo a configuração de uma replicação *master-slave* no Redis, não? Será que você já separou um caderno de

anotações ou algo do tipo? Pois deve ser algo complexo e complicado de se fazer. Mas infelizmente tenho que lembrá-lo de uma regra básica do Redis, que é **ser simples**, e aqui não seria diferente.

As formas possíveis de configurar uma replicação no Redis são as descritas na seção 9.1, mas dessa vez vamos utilizar o CLI e a própria linha de comando para fazer a configuração da replicação. Lembre-se que isso pode ser feito diretamente no arquivo de configuração do Redis (`redis.conf`).

Vamos começar iniciando uma nova instância do Redis na porta 6379. Veja como fazer isso no exemplo a seguir:

```
src/redis-server --port 6379
```

Essa instância será a nossa master. Agora vamos iniciar uma outra instância na porta 6380 conforme o seguinte exemplo:

```
src/redis-server --port 6380
```

Nesse momento, nós temos duas instâncias do Redis sendo executadas, mas uma ainda não está conectada à outra, e por isso ambas são do tipo master. Vamos utilizar o CLI para acessar a instância da porta 6380 e configurá-la como slave da instância sendo executada na porta 6379. Veja como fazer isso:

```
src/redis-cli -p 6380

127.0.0.1:6380> SLAVEOF 127.0.0.1 6379
OK
```

Só isso? Sim, é somente isso que precisamos fazer (executar o comando `SLAVEOF`) para configurar uma instância como slave. Poderíamos utilizar a propriedade `slaveof <master ip> <master port>` do arquivo de configuração ou ainda o parâmetro `--slaveof` do comando de terminal `redis-server` para realizar essa configuração. Para confirmar que tudo está correto, vamos executar o comando `INFO replication` e verificar quais informações ele nos fornece.

```
127.0.0.1:6380> INFO replication

# Replication
```

```
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:4
master_sync_in_progress:0
slave_repl_offset:211
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

Note que a `role` dessa instância está definida como slave e a conexão (`master_link_status`) com o master (`master_host` e `master_port`) está funcionando (*up*).

Ainda com o CLI conectado ao slave, vamos tentar criar nele uma nova chave e ver o que acontece.

```
127.0.0.1:6380> SET novaChave novoValor
(error) READONLY You can't write against a read only slave.
```

Veja que o Redis não aceitou criar a nova chave. Isso acontece porque, desde a versão 2.6 do Redis, por padrão instâncias do tipo slave são *read-only* (aceitam somente leitura). Isso pode ser desfeito mudando o parâmetro de configuração `slave-read-only` para `no`.

Agora vamos iniciar um novo CLI e conectá-lo ao master (porta 6379) para que desta vez consigamos criar a nossa chave e verificar as informações sobre replicação que o comando `INFO` nos retorna.

```
127.0.0.1:6379> INFO replication
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6380,state=online,offset=981,lag=1
```

```
master_repl_offset:995
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:994

127.0.0.1:6379> SET novaChave novoValor
OK
```

Veja que existe uma propriedade `connected_slaves` informando quantos slaves estão conectados a esta instância e quais são eles (no nosso exemplo, apenas o `slave0`). Repare também que agora a nossa chave foi criada sem nenhum problema e ela já deve ter sido replicada para nosso slave. Para conferir, basta listarmos as chaves no nosso CLI conectado ao slave (porta 6380).

```
127.0.0.1:6380> keys *
1) "novaChave"
127.0.0.1:6380> get novaChave
"novoValor"
```

Para finalizar nosso exemplo de replicação, vamos para a instância master e ver como o slave se comporta. Para o master, vamos utilizar um novo comando chamado `SHUTDOWN`. Veja o uso do comando a seguir:

```
127.0.0.1:6379> SHUTDOWN
127.0.0.1:6379> exit
```

Agora no CLI conectado ao slave, vamos novamente verificar as informações de replicação através do comando `INFO`:

```
127.0.0.1:6380> INFO replication

# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:down
master_last_io_seconds_ago:-1
```

```
master_sync_in_progress:0
slave_repl_offset:15
master_link_down_since_seconds:194
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

Como era de se esperar, agora a propriedade `master_link_status` está definida como `down`. Ao iniciarmos novamente o master, essa instância slave irá detectá-lo automaticamente e essa propriedade voltará ao valor `up`. Pronto, agora já sabemos como criar várias instâncias do Redis e conectá-las entre si para realizar replicação dos dados.

10.2 Sentinel

Sentinel é a resposta oferecida quando precisamos ter uma solução de *failover* automático e de alta disponibilidade no Redis. Vamos conhecer a seguir as quatro principais responsabilidades do *Sentinel* fornecidas na sua documentação [19], para assim entendermos melhor o seu propósito e sua aplicação em um ambiente com várias instâncias do Redis.

- **Monitoramento**: Sentinel verifica constantemente se as instâncias master e slave estão funcionando corretamente;
- **Notificação**: Sentinel pode notificar um administrador de sistemas, ou uma outra aplicação por meio de uma API, que há algo de errado com alguma instância do Redis que está sendo monitorada;
- **Failover automático**: se um master não está funcionando de forma correta, é tarefa do Sentinel iniciar um processo de *failover* e eleger e promover um slave para master. Além disso, o Sentinel precisa reconfigurar os outros slaves (caso existam) para usar o novo master, e por final, informar as aplicações (clientes) sobre o novo master;

- **Provedor de dados de configuração**: os clientes conectam-se ao Sentinel para obter o endereço da instância master do Redis, e também para serem avisados quando alguma mudança ocorre.

O Sentinel funciona de forma distribuída, ou seja, podemos executar várias instâncias ou processos do Sentinel em um ou mais computadores. Vamos ver a seguir algumas regras utilizadas por ele:

- Um *cluster* de Sentinel pode realizar o *failover* do master mesmo se alguma instância do Sentinel estiver falhando;
- Somente uma única instância do Sentinel não consegue realizar o *failover* sem a autorização de outras instâncias do Sentinel;
- Clientes podem conectar em qualquer instância do Sentinel para obter os dados de configuração do master.

Configurando e executando o Sentinel

Diferente do Redis, o Sentinel precisa de um arquivo de configuração para ser executado. Esse arquivo de configuração possui parâmetros diferentes dos apresentados na seção 9.1, pois são parâmetros específicos do Sentinel. No decorrer do exemplo apresentado aqui podemos conhecê-los melhor.

Vamos começar iniciando uma instância do Redis na porta 6379, que será a instância master, e uma outra instância na porta 6380, que será a instância slave. Primeiro, vamos executar o master da seguinte forma:

```
src/redis-server --port 6379
```

A instância master já está pronta. Agora vamos iniciar nossa instância slave. A configuração do slave será feita diretamente via linha de comando, conforme a seguir:

```
src/redis-server --port 6380 --slaveof 127.0.0.1 6379
```

Com o master e slave prontos, podemos partir para o Sentinel. Vamos utilizar duas instâncias do Sentinel nesse exemplo, pois esta é a quantidade mínima de instâncias necessárias para realizar o processo de *failover* de forma

automática. A primeira etapa é criar um arquivo de configuração para o Sentinel. Vamos criar um arquivo chamado `sentinel1.conf` no mesmo local do arquivo `redis.conf`, e adicionar o seguinte conteúdo:

```
port 26379
sentinel monitor mymaster 127.0.0.1 6379 2
sentinel down-after-milliseconds mymaster 10000
```

Resumidamente, essa configuração especifica que o Sentinel vai ser executado na porta 26379, ele vai monitorar a instância do Redis na porta 6379 e definir que o master está inacessível depois de 10 segundos sem comunicação com ele.

Veja a seguir uma descrição mais detalhada de cada parâmetro utilizado no arquivo de configuração do Sentinel que acabamos de criar.

- **port <sentinel-port>**: porta na qual o Sentinel irá ser executado;
- **sentinel monitor <master-name> <ip> <redis-port> <quorum>**: define o Redis (somente instâncias master) que será monitorado pelo Sentinel. É possível configurar mais instâncias do Redis adicionando mais parâmetro como esse ao mesmo arquivo de configuração. `master-name` é um nome qualquer para o Redis, `ip` é o ip onde o Redis está sendo executado, `redis-port` é a porta em que o Redis está sendo executado e `quorum` é a quantidade mínima de instâncias do Sentinel necessárias para eleger um novo master ou definir que o master atual está indisponível.
- **sentinel down-after-milliseconds <master-name> <milliseconds>**: define o tempo (em milissegundos) sem comunicação com o Sentinel necessário para o master ser considerado como indisponível.

Agora vamos criar um outro arquivo chamado `sentinel2.conf` e adicionar o seguinte conteúdo:

```
port 26380
sentinel monitor mymaster 127.0.0.1 6379 2
sentinel down-after-milliseconds mymaster 10000
```

O Sentinel pode ser executado de duas formas: uma é utilizando o comando `redis-sentinel` e a outra é utilizando o próprio comando do Redis `redis-server` com o parâmetro `--sentinel`. Como teremos que iniciar duas instâncias do Sentinel, vamos utilizar as duas formas. Veja isso a seguir:

```
src/redis-sentinel sentinel1.conf

src/redis-server sentinel2.conf --sentinel
```

Pronto, agora já temos toda a nossa infraestrutura devidamente configurada e funcionando, portanto, chegou o momento de vermos na prática como o *failover* automático irá funcionar. Mas antes, vamos utilizar o CLI e obter algumas informações sobre o Sentinel sendo executado na porta 2679. Veja a seguir como utilizar o CLI para acessar essa instância do Sentinel:

```
src/redis-cli -p 26379

127.0.0.1:26379> INFO sentinel

# Sentinel
sentinel_masters:1
sentinel_tilt:0
sentinel_running_scripts:0
sentinel_scripts_queue_length:0
master0:name=mymaster,status=ok,address=127.0.0.1:6379,
  slaves=1,sentinels=2
```

Esse comando retornou algumas informações interessantes. Através desses dados podemos saber que esse Sentinel está monitorando um único master (`sentinel_masters:1`) de nome `name=mymaster`. Esse master possui um único slave (`slaves=1`) e está devidamente acessível (`status=ok`). Também é possível saber que existem duas instâncias do Sentinel (`sentinels=2`) monitorando o master de nome `mtmaster`.

Agora vamos conectar o CLI ao master e finalizá-lo da seguinte forma:

```
src/redis-cli -p 6379
```

```
127.0.0.1:6379> SHUTDOWN
127.0.0.1:6379> exit
```

Após alguns segundos, o Sentinel vai entender que a instância master do Redis está inacessível e vai marcá-la como `sdown`. Depois, a outra instância do Sentinel vai realizar a mesma marca e iniciar o processo de votação e eleição do novo master. Depois de mais alguns segundos, você poderá notar que o slave passou a ser master através do seguinte *output* apresentado pelo Sentinel:

```
+switch-master mymaster 127.0.0.1 6379 127.0.0.1 6380
```

Para confirmarmos que o *failover* ocorreu de fato, vamos conectar novamente ao Sentinel pelo CLI e consultar seus dados:

```
src/redis-cli -p 26379

127.0.0.1:26379> INFO sentinel

# Sentinel
sentinel_masters:1
sentinel_tilt:0
sentinel_running_scripts:0
sentinel_scripts_queue_length:0
master0:name=mymaster,status=ok,address=127.0.0.1:6380,
  slaves=1,sentinels=2
```

E sim, o master agora é a instância que está rodando na porta 6380 e tudo está funcionando conforme era esperado. Mas o que acontece se a antiga instância master voltar a ser executada? Vamos ver isso agora:

```
src/redis-server --port 6379

src/redis-cli --p 6379

127.0.0.1:6379> INFO replication

# Replication
role:slave
```

```
master_host:127.0.0.1
master_port:6380
master_link_status:up
master_last_io_seconds_ago:1
master_sync_in_progress:0
slave_repl_offset:12253
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

Veja que interessante, ao executarmos novamente a antiga instância master, o Sentinel automaticamente a promoveu e a configurou como slave da instância master corrente.

Com isso, encerramos o nosso exemplo sobre Sentinel, mas você pode ver algumas dicas para obter mais informações sobre o Sentinel a seguir.

Saiba mais sobre o Sentinel

Note que a configuração realizada para o nosso exemplo é simples. Para conhecer todas as possibilidades de configuração do Sentinel é importante consultar o arquivo `sentinel.conf` fornecido pela instalação padrão do Redis que é muito bem documentado.

Uma outra fonte de informação muito valiosa e com a qual é possível obter mais informações sobre o Sentinel e como ele funciona internamente é a sua documentação online, disponível através do endereço:

http://redis.io/topics/sentinel

10.3 Cluster

A implementação de *Cluster* no Redis ainda está em versão *beta* e por este motivo ainda não está completamente pronto para ser utilizado em um ambiente

de produção. Mas devido à sua importância, não podemos deixar de citá-lo no livro, pois é interessante sabermos para que ele serve e como poderemos utilizá-lo muito em breve.

Cluster fornece uma forma automática para particionamento de dados conforme descrevemos na seção 8.3. Ele implementa uma forma híbrida de *query routing* (mistura de *query routing* com *client side partitioning*) que utiliza o cliente como ajuda.

Em outras palavras, isso funciona da seguinte forma: uma requisição (envio de comando) feita por um cliente não é diretamente redirecionada de uma instância do Redis para outra até encontrar a instância correta que contém a chave do comando. Em vez disso, o próprio cliente é redirecionado para a instância correta.

Ao utilizarmos a implementação de *Cluster* do Redis, as seguintes características estarão automaticamente disponíveis para uso:

- Capacidade de dividir automaticamente um conjunto de dados (*dataset*) entre múltiplos *nodes* (instâncias do servidor Redis);
- Capacidade de continuar as operações quando um subconjunto dos *nodes* estão falhando ou são incapazes de se comunicar com os outros *nodes* do cluster.

Particionamento de dados

O *sharding* (particionamento de dados) é feito através de *hash slot*, onde cada *slot* pode conter um quantidade finita de slots, pois um único cluster pode possuir no máximo 16384 *hash slots* disponíveis e o valor de slots que cada node vai conter depende diretamente da quantidade de nodes que compõem o cluster. Vamos ver exemplos disso, imagine um cluster que possui três instâncias (A, B e C) do Redis em cluster, onde:

- O *node A* contém hash slots que vão de 0 até 5500;
- O *node B* contém hash slots que vão de 5501 até 11000;
- O *node C* contém hash slots que vão de 11001 até 16384.

Cluster em um modelo master-slave

Cada node que compõe um cluster no Redis é necessariamente um master. Mas imagine que estamos usando o cenário anterior, onde temos três instâncias (A, B e C) do Redis para compor um cluster, e ocorre um problema na instância *B*. Se isso ocorrer, todos os slots de 5501 até 11000 ficariam inacessíveis.

Para resolver isso, podemos adicionar um ou mais instâncias slave para cada node que compõe o cluster. Assim, caso algum problema ocorra com alguma instância master, a slave assume o lugar da master e os slots continuam disponíveis no cluster, pois os dados da master estão replicados na slave.

> **Para saber mais**
>
> É importante lembrar que as informações apresentadas anteriormente são baseadas na versão *beta* dessa implementação e que podem surgir mudanças no decorrer de seu desenvolvimento.
>
> Você pode obter mais informações sobre a implementação de cluster no Redis em:
>
> http://redis.io/topics/cluster-spec

10.4 Próximos passos

O capítulo está terminando, podemos considerá-lo como o capítulo final quando pensamos na administração do Redis. Este capítulo foi muito importante pois nos ensinou a configurar e entender o comportamento do Redis em um ambiente com várias instâncias.

No próximo capítulo, eu irei apontar caminhos que podemos seguir para continuar aprendendo mais sobre o Redis e também a obter informações sobre o seu desenvolvimento para que assim possamos acompanhar a sua evolução.

Capítulo 11

Para saber mais

Estamos chegando ao final do livro e a nossa jornada travada aqui está terminando, mas isso não significa que chegamos ao final dos estudos. Lembre-se que o Redis é uma ferramenta nova e que ainda está em constante evolução, e por isso muitas novidades ainda vão aparecer em novas versões.

Muita coisa nova ainda está sendo implementada e deverá ser liberada para uso em produção em pouco tempo. A principal novidade e os esforços mais aplicados têm sido no suporte a *Cluster* 10.3 e isso já está disponível para testes.

Para acompanhar as novidades do Redis ou enviar sugestões e discutir sobre o futuro do Redis, primeiro eu recomendo seguir o Salvatore Sanfilippo pelo seu perfil no *Twitter* e também participar do grupo oficial do Redis.

O perfil do Twitter do Salvatore é:

https://github.com/antirez/redis

E o grupo oficial do Redis é:

https://groups.google.com/forum/#!forum/redis-db

Uma outra forma de se obter informação sobre o Redis é através de sua documentação disponível em:

http://redis.io/documentation

Por último, mas não menos importante, participe do grupo criado exclusivamente para os leitores desse livro, que está disponível em:

https://groups.google.com/forum/#!forum/redis-casadocodigo

E para finalizar, agradeço a você leitor por dedicar seu tempo nesta nossa jornada juntos, espero que tenha sido proveitosa e divertida. Obrigado!

Referências Bibliográficas

[1] Citrusbyte. Redis data types. http://redis.io/topics/data-types, 2013.

[2] Wikipedia contributors. Bitmap. http://bit.ly/1bGoP2Y, 2013.

[3] Wikipedia contributors. Glob (programming). http://bit.ly/HAFsJC, 2013.

[4] Wikipedia contributors. Publish–subscribe pattern. http://bit.ly/NDHsnP, 2013.

[5] Wikipedia contributors. Round-trip delay time. http://bit.ly/1fSLa8I, 2013.

[6] Wikipedia contributors. Single threading. http://bit.ly/1fCJnRQ, 2013.

[7] Contribuidores da Wikipédia. Condição de corrida. http://bit.ly/18tGehY, 2013.

[8] Contribuidores da Wikipédia. Fifo. http://bit.ly/1duf4iS, 2013.

[9] Contribuidores da Wikipédia. Lógica binária. http://bit.ly/1eKW3XD, 2013.

[10] Contribuidores da Wikipédia. Thread (ciência da computação). http://bit.ly/1muA1hY, 2013.

[11] Contribuidores da Wikipédia. Cliente-servidor. http://bit.ly/1cS75H7, 2014.

[12] Contribuidores da Wikipédia. Jms. http://bit.ly/1cS5wsB, 2014.

[13] The Linux Information Project. Bsd license definition. http://www.linfo.org/bsdlicense.html, 2004.

[14] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. Manual de referência de lua 5.1. http://bit.ly/1fDRZ89, 2011.

[15] Salvatore Sanfilippo. Redis reliable queues with lua scripting. http://bit.ly/OqxmXq, 2012.

[16] Salvatore Sanfilippo. Antirez weblog. http://antirez.com/, 2013.

[17] Redis Site. Partitioning: how to split data among multiple redis instances. http://bit.ly/1lZYzvb, 2014.

[18] Redis Site. Redis memory optimization. http://bit.ly/1lZYIP8, 2014.

[19] Redis Site. Redis sentinel documentation. http://bit.ly/1hbN9UC, 2014.

[20] Venkat Subramaniam. Programming concurrency on the jvm. 2011.