20-Redis Implementations

Author: Vincent Lau

Note: This material is intended for educational purposes only. All rights reserved. Any unauthorized sharing or copying of this material, in any form, to any individual or party, for any use without prior permission, is strictly prohibited.

Learning Objectives

> Understand What is Redis, and why Redis is so fast, comparing to Database
>
> Connect Redis in Spring Boot.
>
> Learn the usage of RedisTemplate.
>
> Access Redis via Repository Layer.

Introduction

`Redis` (Remote Dictionary Server) is an open-source, in-memory data store and cache. It is often referred to as a data structure server because it provides support for a wide range of data structures, which makes it highly versatile and valuable for various use cases. Redis is known for its exceptional performance, low-latency data access, and its ability to handle high-throughput workloads.

Key features and characteristics of Redis

In-Memory Data Store

Redis stores data in memory, which allows for extremely fast data access. This in-memory storage makes Redis ideal for applications where speed and low latency are essential.

Key-Value Store

Redis uses a key-value data model. Data is stored with a unique key, and you can access, update, or delete data using these keys.

Data Structures

Redis supports a variety of data structures, including strings, lists, sets, sorted sets, hashes, bitmaps, hyperloglogs, and more. This versatility enables developers to work with different types of data efficiently.

Persistence Options

While Redis primarily operates in-memory, it offers options for data persistence. You can configure Redis to periodically save data snapshots to disk, ensuring data durability in case of server restarts.

Caching (System Design)

Redis is commonly used as a cache to store frequently accessed data in memory, reducing the load on primary data stores like databases. This results in faster response times for applications.

Atomic Operations

Redis supports atomic operations, making it useful for implementing counters, locks, and distributed data manipulation.

Multi-Language Support

Redis provides client libraries for a wide range of programming languages, making it accessible and easy to integrate into applications built with various technologies.

High Availability

Redis supports replication and clustering, which enables the creation of highly available and fault-tolerant systems.

Scalability

Redis can be used as a single server or in a clustered configuration to handle large-scale data and high-traffic workloads.

Redis is widely used in web applications, real-time analytics, caching layers, messaging systems, task queues, and many other use cases. It has become an essential component of modern software architectures, providing the speed and flexibility required to meet the demands of today's applications.

Why Redis is so fast?

Redis is significantly faster compared to traditional databases for several reasons:

In-Memory Data Storage

Redis stores data in memory (RAM) rather than on disk. Memory access is orders of magnitude faster than disk access. This in-memory storage allows Redis to offer low-latency data retrieval, making it exceptionally fast.

Optimized Data Structures

Redis is designed for specific data structures, such as sets, lists, and hashes. These data structures are optimized for in-memory operations, ensuring that data manipulation is efficient and fast.

No Disk I/O

Traditional databases frequently write data to disk to ensure durability. Redis, while offering options for data persistence, primarily keeps data in memory. This eliminates the need for disk I/O during most operations, which is a major source of latency in traditional databases.

Single-Threaded Event Loop

Redis uses a single-threaded event loop to handle operations. While this may seem counterintuitive, it eliminates the **overhead of thread management** and context switching, making Redis extremely efficient. Redis handles many small operations rapidly and can scale by running multiple instances.

Low-Level and Efficient Networking

Redis uses a simple and efficient binary protocol for communication. This reduces the overhead of data serialization and deserialization, leading to faster data transfer between clients and the server.

Parallelism and Pipelining

Redis supports multiple client connections and pipelining of commands. This allows for parallel processing of operations and minimizes latency.

Data Locality

Redis keeps related data structures close together in memory. This design helps reduce memory fragmentation and improves the efficiency of data access.

Implementations

CRUD Repository

To create a repository interface for interacting with Redis using Spring Data Redis, you can extend the `CrudRepository` or `RedisRepository` interfaces provided by Spring Data Redis. Here's a simple code example of a repository interface for a "Person" entity:

1. Define a `Person` entity class:

```java
import org.springframework.data.annotation.Id;
import org.springframework.data.redis.core.RedisHash;

@RedisHash("Person")
public class Person {

    @Id
    private String id;
    private String firstName;
    private String lastName;

    // Constructors, getters, setters, and other methods
}
```

2. Create a repository interface for the `Person` entity:

```java
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<Person, String> {

    // You can define custom query methods here if needed
}
```

In this example:

- The `Person` class is annotated with `@RedisHash` to specify the Redis hash key for this entity.
- The `@Id` annotation indicates the primary key field.
- The `PersonRepository` interface extends `CrudRepository<Person, String>`, where `Person` is the entity class and `String` is the data type of the entity's ID field.

With this repository interface, you can perform basic CRUD operations on the `Person` entity, and you can add custom query methods if needed.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class PersonService {

    private final PersonRepository personRepository;

    @Autowired
    public PersonService(PersonRepository personRepository) {
        this.personRepository = personRepository;
    }

    public Person savePerson(Person person) {
        return personRepository.save(person);
    }

    public Person getPerson(String id) {
        return personRepository.findById(id).orElse(null);
    }

    public void deletePerson(String id) {
        personRepository.deleteById(id);
    }
}
```

In this service class, we use the `PersonRepository` to perform operations on the "Person" entity. You can also create more complex query methods in the repository interface to search for persons by specific attributes or criteria.

Ensure that you have properly configured your Redis connection details in your Spring Boot application's configuration files (e.g., `application.properties` or `application.yml`) to connect to your Redis server.

RedisTemplate

To use `RedisTemplate` in a Spring Boot application for more fine-grained control over Redis operations, you need to add the `spring-boot-starter-data-redis` dependency. This starter module provides integration with Redis and includes the `RedisTemplate` bean that you can use.

If you're using Maven, add the following dependency to your `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

Once you've added this dependency, you can start using `RedisTemplate` in your Spring Boot application to interact with Redis. Here's an example of how you can configure and use `RedisTemplate` in a Spring Boot service:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Service;

@Service
public class RedisService {

    private final RedisTemplate<String, String> redisTemplate;

    @Autowired
    public RedisService(RedisTemplate<String, String> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    public void setValue(String key, String value) {
        redisTemplate.opsForValue().set(key, value);
    }

    public String getValue(String key) {
        return redisTemplate.opsForValue().get(key);
    }
}
```

In this example, `RedisService` is a Spring service that uses `RedisTemplate` to set and get values in Redis. The `RedisTemplate` is injected into the service via constructor injection.

Ensure that you have configured the Redis server properties in your `application.properties` or `application.yml` file, as mentioned earlier in this conversation. Spring Boot will automatically set up the `RedisTemplate` bean for you based on your configuration.

Remember to customize the key and value types in `RedisTemplate` to match your application's data requirements. You can also use other data structures like lists, sets, and hashes, as `RedisTemplate` provides operations for these as well.

Spring Cache

The `spring-boot-starter-cache` is a Spring Boot starter module that simplifies the configuration and use of caching in Spring Boot applications. It's part of the broader Spring Cache abstraction, which allows you to integrate various caching providers, including Redis, Ehcache, Caffeine, and more.

1. **Add Dependency**

   To include the `spring-boot-starter-cache` in your project, add the following dependency to your `pom.xml`. If you're using Maven, both dependencies are required if you want to use Redis as the caching provider in your Spring Boot application.

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2. **Configure Cache Provider**

   By default, Spring Boot uses a simple in-memory cache called "ConcurrentMapCache." If you want to use a specific caching provider, like Redis, Ehcache, or Caffeine, you can include the relevant dependency and configure it in your `application.properties` or `application.yml`.

```yaml
spring.datasource.driver-class-name: com.mysql.cj.jdbc.Driver
spring.datasource.url: jdbc:mysql://localhost:3306/rediscachetest
spring.datasource.username: root
```

```
spring.datasource.password: ****
spring.jpa.database-platform:
org.hibernate.dialect.MySQL8Dialectspring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto: update

spring.cache.type: redis
spring.cache.redis.cache-null-values: true
spring.cache.redis.time-to-live: 40000
```

3. Enable Caching

In your Spring Boot application, enable caching by adding the `@EnableCaching` annotation to your main application class. This annotation tells Spring to set up caching for your application.

```
import org.springframework.cache.annotation.EnableCaching;

@SpringBootApplication
@EnableCaching
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

4. Model Class

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Invoice implements Serializable {

    private static final long serialVersionUID = -4439114469417994311L;

    @Id
    @GeneratedValue
    private Integer invId;
    private String invName;
    private Double invAmount;

}
```

5. InvoiceRepository

```java
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.dev.springboot.redis.model.Invoice;

@Repository
public interface InvoiceRepository extends JpaRepository<Invoice, Integer> {

}
```

6. Use @Cacheable Annotations

To cache the results of specific methods, annotate those methods with `@Cacheable`. The annotation allows you to specify a cache name, which will be used to store the cached results. For example:

```java
@Service
public class InvoiceServiceImpl implements InvoiceService {

    @Autowired
    private InvoiceRepository invoiceRepo;

    @Override
    public Invoice saveInvoice(Invoice inv) {
        return invoiceRepo.save(inv);
    }

    @Override
    @CachePut(value="Invoice", key="#invId")
    public Invoice updateInvoice(Invoice inv, Integer invId) {
        Invoice invoice = invoiceRepo.findById(invId)
            .orElseThrow(() -> new InvoiceNotFoundException("Invoice Not Found"));
        invoice.setInvAmount(inv.getInvAmount());
        invoice.setInvName(inv.getInvName());
        return invoiceRepo.save(invoice);
    }

    @Override
    @CacheEvict(value="Invoice", key="#invId")
    // @CacheEvict(value="Invoice", allEntries=true) //in case there are multiple records to delete
    public void deleteInvoice(Integer invId) {
```

```
26          Invoice invoice = invoiceRepo.findById(invId)
27              .orElseThrow(() -> new InvoiceNotFoundException("Invoice Not
    Found"));
28          invoiceRepo.delete(invoice);
29      }
30
31      @Override
32      @Cacheable(value="Invoice", key="#invId")    // The key for the cache
33      public Invoice getOneInvoice(Integer invId) {
34          Invoice invoice = invoiceRepo.findById(invId)
35              .orElseThrow(() -> new InvoiceNotFoundException("Invoice Not
    Found"));
36          return invoice; // the return value will be the value stored in Cache
37      }
38
39      @Override
40      @Cacheable(value="Invoice")
41      public List<Invoice> getAllInvoices() {
42          return invoiceRepo.findAll();
43      }
44  }
```

In this example, the result of the `getCachedData` method will be cached, and future invocations of the method with the same arguments will return the cached result instead of executing the method again.

The `spring-boot-starter-cache` simplifies the configuration and use of caching in Spring Boot applications. It abstracts the underlying caching provider, making it easy to switch between different providers without changing your application's code. This can help improve the performance and responsiveness of your application by reducing the need to recompute expensive or frequently requested data.