Spring Data Redis

Spring Data Redis est un outil fantastique pour intégrer Redis, un magasin de données en mémoire à haute performance, dans votre application Spring. Il simplifie le travail avec Redis en fournissant une interface de style dépôt familière et des abstractions sur les opérations Redis de bas niveau. Suivons étape par étape comment le configurer et l'utiliser.

1. Configurer votre projet

Tout d'abord, incluez la dépendance Spring Data Redis dans votre projet. Si vous utilisez Maven, ajoutez ceci à votre `pom.xml` :

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

Pour Gradle, utilisez :

```
implementation 'org.springframework.boot:spring-boot-starter-data-redis'
```

Cela inclut le client Redis (Jedis ou Lettuce—Lettuce est la valeur par défaut). Si vous préférez Jedis, vous pouvez l'ajouter explicitement et exclure Lettuce :

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <exclusions>
        <exclusion>
            <groupId>io.lettuce</groupId>
            <artifactId>lettuce-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

2. Configurer Redis

Configurez votre connexion Redis dans `application.properties` ou `application.yml`. Pour une instance Redis locale s'exécutant sur le port par défaut (6379) :

```
spring.redis.host=localhost
spring.redis.port=6379
spring.redis.password= # Optionnel, si votre serveur Redis a un mot de passe
spring.redis.database=0 # Index de la base de données par défaut
```

Si vous utilisez un serveur Redis distant ou un service comme AWS ElastiCache, mettez à jour l'hôte et les informations d'identification en conséquence.

3. Utilisation de base avec RedisTemplate

Spring Data Redis fournit `RedisTemplate` pour les opérations de bas niveau. Vous pouvez l'injecter dans votre service ou composant :

```java
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 saveData(String key, String value) {
        redisTemplate.opsForValue().set(key, value);
    }

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

- `RedisTemplate` est générique : `<String, String>` signifie que les clés et les valeurs sont des chaînes. Vous pouvez utiliser d'autres types (par exemple, `<String, Object>`).

- `opsForValue()` est pour les opérations clés-valeurs simples. D'autres méthodes incluent `opsForList()`, `opsForSet()`, `opsForHash()`, etc., pour différentes structures de données Redis.

4. Utilisation avec des objets

Pour stocker et récupérer des objets Java, configurez `RedisTemplate` avec des sérialiseurs. Spring Boot le configure automatiquement, mais vous pouvez le personnaliser si nécessaire :

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
public class RedisConfig {
    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory connectionFactory) {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(connectionFactory);
        template.setKeySerializer(new StringRedisSerializer());
        template.setValueSerializer(new Jackson2JsonRedisSerializer<>(Object.class));
        template.afterPropertiesSet();
        return template;
    }
}
```

Maintenant vous pouvez stocker et récupérer des objets :

```
public class Person {
    private String firstName;
    private String lastName;

    // Constructeur par défaut (pour la désérialisation)
    public Person() {}

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
```

```
    // Getters et setters
    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) { this.firstName = firstName; }
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
}

@Service
public class PersonRedisService {
    private final RedisTemplate<String, Object> redisTemplate;

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

    public void savePerson(String key, Person person) {
        redisTemplate.opsForValue().set(key, person);
    }

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

5. Approche Repository

Pour une abstraction de niveau supérieur, utilisez les dépôts Spring Data Redis. Définissez une entité et un dépôt :

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

@RedisHash("Person") // Mappe à un hash Redis avec le préfixe "Person"
public class Person {
    @Id
    private String id; // La clé Redis sera "Person:<id>"
    private String firstName;
    private String lastName;
```

```
    // Constructeurs, getters, setters (comme ci-dessus)
}

import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<Person, String> {
}
```

Utilisez-le comme ceci :

```
@Service
public class PersonService {
    private final PersonRepository repository;

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

    public void savePerson() {
        Person person = new Person("John", "Doe");
        repository.save(person);
        System.out.println("Saved person with ID: " + person.getId());
    }

    public void findPerson(String id) {
        Person person = repository.findById(id).orElse(null);
        if (person != null) {
            System.out.println(person.getFirstName() + " " + person.getLastName());
        }
    }
}
```

- `@RedisHash` mappe l'entité à un hash Redis.
- Le dépôt fournit des opérations CRUD hors de la boîte.

6. Exécuter votre application

Assurez-vous que Redis s'exécute localement (par exemple, via Docker : `docker run -d -p 6379:6379 redis`) ou connectez-vous à une instance distante. Ensuite, démarrez votre application Spring Boot :

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

7. Fonctionnalités avancées

- **TTL (Time-to-Live) :** Définissez une expiration sur les clés avec `redisTemplate.expire(key, 10, TimeUnit.MINUTES)`.
- **Hashes :** Utilisez `opsForHash()` pour travailler directement avec les hashes Redis.
- **Pub/Sub :** Utilisez `RedisMessageListenerContainer` pour la messagerie.

Exemple de flux de travail

1. Enregistrez un objet `Person` avec `repository.save()` ou `redisTemplate.opsForValue().set()`.
2. Récupérez-le avec `repository.findById()` ou `redisTemplate.opsForValue().get()`.
3. Vérifiez votre instance Redis (par exemple, avec `redis-cli`) pour voir les données.

Cela couvre les bases ! Si vous souhaitez explorer le cache, pub/sub, ou quelque chose de spécifique, faites-le moi savoir, et je plongerai plus profondément. Comment cela s'intègre-t-il avec ce que vous essayez de faire ?