Springboot와 Redis 연동

Redis란?

---

https://redis.io/docs/

> Redis는 오픈 소스로 제공되는 인메모리 데이터 구조 저장소로, 데이터베이스, 캐시, 메시지 브로커 및 스트리밍 엔진으로 사용됩니다.
>
> Redis는 인메모리 데이터 세트와 함께 최상의 성능을 제공합니다. 사용 사례에 따라 Redis는 데이터 세트를 주기적으로 디스크에 덤프하거나 각 명령을 디스크 기반 로그에 추가하여 데이터를 영구적으로 저장할 수 있습니다. 단순히 기능이 풍부하고 네트워크 연결된 인메모리 캐시만 필요한 경우 영구 저장소를 비활성화할 수도 있습니다.
>
> Redis는 비동기 복제를 지원하며, 빠른 비동기식 동기화와 네트워크 분리 상태에서의 자동 재연결 및 부분 재동기화를 지원합니다.

Redis를 사용하는 이유

`Monolithic 아키텍처`에서는 애플리케이션 전체가 하나의 단일 단위로 개발되고 배포되기 때문에 대규모 애플리케이션의 유지보수 및 확장이 어렵고, 서버의 규모가 커질수록 세션 관리가 복잡해지며, 개발과 운영의 경계가 불분명해지는 문제점이 있습니다.

최근에는 `MSA 아키텍처`가 점점 더 많이 사용되고 있지만 각각의 마이크로서비스가 분리되어 있어서 세션을 공유하기 어렵기 때문에 Redis와 같은 분산 캐시 솔루션을 이용하여 세션 관리를 수행합니다.

또한 Redis는 메모리에 데이터를 저장하기 때문에 빠른 응답 속도를 제공하며, 대규모 데이터를 처리하기하며 캐시, 메시지 브로커, 키-값 저장소 등 다양한 용도로 활용될 수 있기 때문에 많은 개발자들이 Redis에 관심을 가지고 있습니다.

이러한 이유로 Redis를 사용하는 곳이 점점 늘어나고 있습니다.

Springboot와 Redis 연동

- **Redis 설치**

```
brew install redis
```

- **Redis 실행**

```
brew services start redis
```

- **Spring 프로젝트 생성**

```
implementation 'org.springframework.boot:spring-boot-starter-data-redis'
implementation 'org.springframework.boot:spring-boot-starter-web'
compileOnly 'org.projectlombok:lombok'
annotationProcessor 'org.projectlombok:lombok'
```

- **application.yml 설정**

```
spring:
  redis:
    host: localhost
    port: 6379
```

- **RedisConfig.java**

```java
@EnableCaching
@Configuration
public class RedisConfig {

    @Value("${spring.redis.host}")
    private String host;

    @Value("${spring.redis.port}")
    private int port;

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        RedisStandaloneConfiguration config = new RedisStandaloneConfiguration(host,
port);
        return new LettuceConnectionFactory(config);
    }

    @Bean
    public RedisTemplate<String, Object> redisTemplate() {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory());

        template.setKeySerializer(new StringRedisSerializer());
        template.setValueSerializer(new GenericJackson2JsonRedisSerializer());

        return template;
    }

    @Bean
    public CacheManager cacheManager(RedisConnectionFactory redisConnectionFactory) {
        RedisCacheConfiguration redisCacheConfiguration =
RedisCacheConfiguration.defaultCacheConfig()
                .serializeKeysWith(RedisSerializationContext.SerializationPair.fromSeria
lizer(new StringRedisSerializer()))
                .serializeValuesWith(RedisSerializationContext.SerializationPair.fromSer
ializer(new GenericJackson2JsonRedisSerializer()))
                .entryTtl(Duration.ofMinutes(3L));

        return RedisCacheManager.RedisCacheManagerBuilder
                .fromConnectionFactory(redisConnectionFactory)
                .cacheDefaults(redisCacheConfiguration).build();
    }
}
```

여기에서 `@Bean`으로 설정한 값들을 하나씩 알아보자.

1. RedisConnectionFactory

RedisConnectionFactory는 Redis 클라이언트가 Redis 서버와 통신하기 위한 Connection을 생성하는 인터페이스입니다. Spring에서는 RedisConnectionFactory 인터페이스를 구현한 여러 클래스를 제공합니다.

2. LettuceConnectionFactory

LettuceConnectionFactory는 Redis 클라이언트 라이브러리 인 Lettuce를 이용해 Redis 서버와 통신하기 위한 Connection을 생성하는 클래스입니다.

3. RedisTemplate

RedisTemplate은 Redis 데이터를 쉽게 다루기 위한 Spring Data Redis의 핵심 클래스 중 하나입니다. Redis 서버와의 연결을 제공하고, Redis 데이터를 삽입, 조회, 수정, 삭제하는 데 사용됩니다.

4. CacheManager

CacheManager는 캐시를 관리하는 추상 인터페이스입니다. 스프링 캐시 추상화 모듈에는 여러가지 캐시 구현체가 존재하며 이러한 캐시 구현체를 이용해 CacheManager에서 캐시의 생성, 조회, 삭제 등을 수행합니다. 즉, CacheManager는 캐시의 생명주기를 관리하는 역할을 합니다.

5. RedisCacheManager

RedisCacheManager는 RedisConnectionFactory를 주입 받아 RedisCache를 생성하고, RedisCacheConfiguration을 설정할 수 있습니다. RedisCacheConfiguration은 Redis 캐시의 설정 정보를 정의하는 객체로, Redis 캐시의 만료 시간(TTL), 캐시의 직렬화 방식 등을 설정할 수 있습니다.

- **MemberDto.java**

```
@Getter
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class MemberDto {
    private String uuid;
    private String name;
}
```

Redis에 객체를 직렬화하여 저장하기 위한 DTO 클래스

- **MemberService.java**

```java
@Service
@RequiredArgsConstructor
public class MemberService {

    private final RedisTemplate<String, Object> redisTemplate;

    public static int getMethodHitCount = 0;

    public void saveMember(String uuid, MemberDto value) {
        redisTemplate.opsForValue().set(uuid, value);
    }

    @Cacheable(value = "Member", key = "#uuid", cacheManager = "cacheManager")
    public MemberDto getMember(String uuid) {
        getMethodHitCount++;
        return (MemberDto) redisTemplate.opsForValue().get(uuid);
    }

    @CacheEvict(value = "Member", key = "#uuid", cacheManager = "cacheManager")
    public void deleteMember(String uuid) {
        redisTemplate.delete(uuid);
    }
}
```

`Service`에 선언된 각각의 기능들을 알아보자.

1. getMethodHitCount

Cache가 기능을 하고 있는지 테스트를 하기 위한 변수이다.

2. @Cacheable

Spring Cache 추상화 기능을 이용하여 메서드 호출 결과를 캐싱할 수 있는 기능을 제공합니다. 이를 이용하여 같은 파라미터로 호출될 때마다 메서드의 실행 결과를 캐시하고, 동일한 파라미터로 메서드가 호출될 때는 캐시된 결과를 반환합니다.

`value` 속성은 캐시 이름을 지정합니다. 여러 개의 캐시를 사용하는 경우, 캐시 이름을 구분하기 위해서 사용합니다.

`key` 속성은 캐시에 저장될 때 사용될 키를 지정합니다. `#`을 이용하여 메서드의 파라미터 값을 참조할 수 있습니다.

`cacheManager` 속성은 캐시 매니저를 지정합니다. 위 코드에서는 Redis를 캐시 매니저로 사용합니다.

3. @CacheEvict

캐시에서 데이터를 삭제하는 기능을 제공합니다. `value`와 `key` 속성은 `@Cacheable` 어노테이션과 같이 사용되며, `cacheManager` 속성은 생략 가능합니다.

4. opsForValue()

`opsForValue()` 메서드는 Redis의 `String` 데이터 타입을 다루기 위한 `ValueOperations` 인터페이스를 반환합니다. `ValueOperations` 인터페이스는 `String` 키와 `Object` 값을 다룰 수 있습니다.

`ValueOperations` 인터페이스 이외에도 `ListOperations`, `SetOperations`, `HashOperations`, `ZSetOperations` 등 다양한 데이터 타입을 지원하는 연산 인터페이스를 제공합니다. 이를 이용하여 Redis의 다양한 데이터 타입을 다룰 수 있습니다.

- **RedisTests.java**

```
@SpringBootTest
public class RedisTests {

    @Autowired
    MemberService memberRedisService;

    @Test
    void saveTest(){
        //given
        String uuid = UUID.randomUUID().toString();
        MemberDto dto = MemberDto.builder().uuid(uuid).name("김동호").build();

        //when
        memberRedisService.saveMember(uuid, dto);
        MemberDto findMember = memberRedisService.getMember(uuid);

        //then
        assertThat(findMember.getUuid()).isEqualTo(uuid);
        assertThat(findMember.getName()).isEqualTo("김동호");
    }

    @Test
    void cacheTest(){
        //given
        String uuid = UUID.randomUUID().toString();
        MemberDto dto = MemberDto.builder().uuid(uuid).name("김동호").build();
        int redisHit = 10;

        //when
        memberRedisService.saveMember(uuid, dto);

        for (int i = 0; i < 10; i++) {
            memberRedisService.getMember(uuid);
        }

        //then
        assertThat(redisHit).isNotEqualTo(memberRedisService.getMethodHitCount);
    }
}
```

`saveTest()`와 `cacheTest()`를 살펴보자

1. `saveTest()`

   저장 테스트는 Redis를 데이터 베이스로 사용하여 `UUID`를 `key`로 `MemberDto` 객체를 저장한 후 조회한 결과 둘이 일치하는지를 테스트한 것입니다.

2. `cacheTest()`

cache 테스트는 `int redisHit = 10` 초기화를 한 후 `getMember`를 10번 조회를 시켰다. cache를 하지 않았다면 `MemberService. getMethodHitCount`의 값이 동일하게 10까지 증감을 했을텐대 cache를 적용했기 때문에 `isNotEqualTo`가 통과를 하였습니다.