redis
Clients

Java SpringBoot New
- Introduction
- Strings
- Lists
- Sets
- ZSets
- Hashes
- Streams
- Common Keys
- Pipelining
- Pub/Sub
- Master/Replica
- Sentinel
- Cluster
- ---
- Auto Config
- Manual Config
- Load Balancing (readFrom)
- RedisTemplate
- Connection Pool & Thread
- Async Spring & Lettuce
- DB select
- Spring Multi Data Source
- Lettuce Multi Data Source
- Spring Project Create
- Spring Project Eclipse
- Spring Project IntelliJ
- ---
- Spring Session Standalone
- Spring Session MasterRepli
- Spring Session Sentinel
- Spring Session Cluster

Java Lettuce(Spring)New

Java Lettuce(Plain)

Java Jedis

Java Redisson

C Hiredis

C# StackExchange

PHP PhpRedis

PHP Predis

Redis Admin & Monitoring Tool

Spring Data Redis Pub/Sub

Spring Data Redis Publish/Subscribe

이 문서는 Spring Data Redis에서 Publish/Subscribe 기능을 설명한 것입니다.
원본 문서는 여기에 있습니다.
Redis Publish/Subscribe 대한 소개와 명령은 여기를 참조하세요.

Spring Data는 Spring Framework의 JMS 통합과 기능과 이름 지정(naming)이 유사한 Redis용 전용 메시징 통합을 제공합니다. Redis 메시징은 대략 두 가지 기능 영역으로 나눌 수 있습니다.

- 메시지 게시 또는 생성 Publication or production of messages
- 메시지 구독 또는 소비 Subscription or consumption of messages

이는 흔히 게시/구독(Publish/Subscribe, 줄여서 Pub/Sub)이라고 불리는 패턴의 예입니다. RedisTemplate 클래스는 메시지 생성에 사용됩니다. Java EE의 메시지 중심(message-driven) 빈(bean) 스타일과 유사한 비동기 수신을 위해, Spring Data는 메시지 중심 POJO(MDP)를 생성하는 데 사용되는 전용 메시지 수신기 컨테이너를 제공하고, 동기 수신을 위해 RedisConnection을 제공합니다.
org.springframework.data.redis.connection과 org.springframework.data.redis.listener 패키지는 Redis 메시징의 핵심 기능을 제공합니다.

Publishing (Sending Messages)

메시지를 게시하려면, 다른 작업과 마찬가지로, 하위 수준 RedisConnection 또는 상위 수준 RedisTemplate을 사용할 수 있습니다 . 두 엔터티 모두 메시지와 대상 채널을 인수로 받아들이는 publish 메서드를 제공합니다. RedisConnection은 원시 데이터(raw data, 바이트 배열)가 필요 하지만, RedisTemplate은 임의의 개체를 메시지로 전달할 수 있습니다. 다음에 간단하게 각 예제가 있습니다.

```
// send message through connection RedisConnection con = ...
byte[] msg = ...
byte[] channel = ...
con.publish(msg, channel);

// send message through RedisTemplate
RedisTemplate template = ...
Long numberOfClients = template.convertAndSend("hello!", "world");
```

Subscribing (Receiving Messages)

수신 측에서는 직접 이름을 지정하거나 패턴 일치를 사용하여 하나 또는 여러 채널을 구독할 수 있습니다. 패턴 일치(pattern matching) 접근 방식은 하나의 명령으로 여러 구독을 생성할 수 있을 뿐만 아니라 구독 시 아직 생성되지 않은 채널(패턴과 일치하는 한)을 수신할 수 있기 때문에 매우 유용합니다.

하위 수준에서는 RedisConnection은 각각 채널 또는 패턴별로 구독하기 위한 Redis 명령을 매핑하는 subscribe와 pSubscribe 메서드를 제공합니다. RedisConnection은 getSubscription 메서드로 연결 구독을 변경할 수 있고, isSubscribed 메서드로 청취 중인지 확인할 수 있습니다.

> Spring Data Redis의 구독 명령은 차단(blocking)됩니다. 즉, 연결에서 구독(subscribe)을 호출하면 현재 스레드가 메시지를 기다리기 시작하면서 차단됩니다. 스레드는 구독이 취소된 경우에만 해제됩니다. 이는 다른 스레드가 동일한 연결에서 unsubscribe나 pUnsubscribe를 호출하면 발생합니다.
> 이 문제에 대한 해결 방법은 "메시지 수신기 컨테이너(Message Listener Containers)"를 참조하세요.

앞서 언급했듯이 일단 구독하면 연결(connection)이 메시지 대기를 시작합니다. 새 구독을 추가하고, 기존 구독을 수정하고, 기존 구독을 취소하는 명령만 허용됩니다. subscribe, pSubscribe, unsubscribe 또는 pUnsubscribe 이외의 항목을 호출하면 예외가 발생합니다.

메시지를 구독하려면 MessageListener 콜백을 구현해야 합니다. 새 메시지가 도착할 때마다 콜백이 호출되고 사용자 코드가 onMessage 메서드에 의해 실행됩니다 . 인터페이스는 실제 메시지뿐만 아니라 메시지가 수신된 채널과 패턴 액세스를 제공합니다. 이 정보를 통해 수신자(callee)는 내용뿐만 아니라 추가 세부정보도 검토하여 다양한 메시지를 구별할 수 있습니다.

Message Listener Containers

차단 특성으로 인해 낮은 수준의 구독은 모든 단일 수신기(listener)에 대한 연결 및 스레드 관리가 필요하므로 매력적이지 않습니다. 이 문제를 완화하기 위해 Spring Data는 모든 무거운 작업을 수행하는 RedisMessageListenerContainer를 제공합니다. EJB와 JMS에 익숙하다면 개념이 친숙할 것입니다. 이는 Spring Framework 및 메시지 기반 POJO(MDP)의 지원과 최대한 유사하도록 설계되었기 때문입니다.

RedisMessageListenerContainer는 메시지 수신기 컨테이너 역할을 합니다. 이것은 Redis 채널에서 메시지를 수신하고 여기에 주입된 MessageListener 인스턴스를 구동합니다. 리스너 컨테이너는 메시지 수신의 모든 스레딩을 담당하고 처리하기 위해 리스너로 전달합니다. 메시지 수신기 컨테이너는 MDP와 메시징 공급자 사이의 중개자이며 메시지 수신 등록, 리소스 획득 및 해제, 예외 변환 등을 처리합니다. 이를 통해 애플리케이션 개발자는 메시지 수신(이에 반응)과 관련된(아마도 복잡한) 비즈니스 로직을 작성하고, 상용적인 Redis 인프라 문제는 프레임워크에 위임할 수 있습니다.

MessageListener는 구독/구독 취소 확인 시 알림을 받도록 SubscriptionListener를 추가로 구현할 수 있습니다 . 호출을 동기화할 때 구독 알림을 듣는 것이 유용할 수 있습니다.

또한, 애플리케이션 공간을 최소화하기 위해 RedisMessageListenerContainer가 구독을 공유하지 않더라도 여러 리스너가 하나의 연결과 하나의 스레드를 공유할 수 있도록 합니다. 따라서, 애플리케이션이 추적하는 리스너나 채널 수에 관계없이 런타임 비용은 전체 수명 동안 동일하게 유지됩니다. 또한, 컨테이너를 사용하면 런타임 구성을 변경할 수 있으므로, 다시 시작할 필요 없이 애플리케이션이 실행되는 동안 리스너를 추가하거나 제거할 수 있습니다. 또한, 컨테이너는 RedisConnection을 필요할 때만 사용하는 지연 구독 방식(lazy subscription approach)을 사용합니다. 모든 리스너의 구독이 취소되면, 정리가 자동으로 수행되고 스레드가 해제됩니다.

메시지의 비동기적 특성을 돕기 위해, 컨테이너에는 메시지 전달을 위한 java.util.concurrent.Executor(또는 Spring TaskExecutor)가 필요합니다. 로드(load), 리스너 수 또는 런타임 환경에 따라 필요에 따라 더 나은 서비스를 제공하도록 실행기(executor)를 변경하거나 조정해야 합니다. 특히, 관리되는 환경(예: 앱 서버)에서는 런타임을 활용하려면 적절한 TaskExecutor를 선택하는 것이 좋습니다.

The MessageListenerAdapter

MessageListenerAdapter 클래스는 Spring의 비동기 메시징 지원의 최종 구성 요소입니다. 간단히 말해서, 거의 모든 클래스를 MDP로 노출할 수 있습니다(몇 가지 제약이 있지만).

다음 인터페이스 정의를 고려하십시오.

```
public interface MessageDelegate {
 void handleMessage(String message);
 void handleMessage(Map message);
 void handleMessage(byte[] message);
 void handleMessage(Serializable message);
 // pass the channel/pattern as well
 void handleMessage(Serializable message, String channel);
}
```

비록 인터페이스가 MessageListener 인터페이스를 상속하지는 않지만 MessageListenerAdapter 클래스를 사용하면 여전히 MDP로 사용할 수 있습니다. 또한 수신하고 처리할 수 있는 다양한 메시지 유형의 콘텐츠에 따라 다양한 메시지 처리 방법이 어떻게 강력하게 유형화되는지 확인하세요. 또한 메시지가 전송되는 채널이나 패턴을 String 유형의 두 번째 인수로 메서드에 전달할 수 있습니다.

```
public class DefaultMessageDelegate implements MessageDelegate {
  // implementation elided for clarity...
}
```

위의 구현 MessageDelegate 인터페이스(위 DefaultMessageDelegate 클래스)에는 Redis 종속성이 전혀 없습니다. 다음 구성을 사용하여 MDP로 만드는 것은 진짜 POJO입니다.

```
@Configuration
class MyConfig {
  // ...
  @Bean
  DefaultMessageDelegate listener() {
    return new DefaultMessageDelegate();
  }
  @Bean
  MessageListenerAdapter messageListenerAdapter(DefaultMessageDelegate listener) {
    return new MessageListenerAdapter(listener, "handleMessage");
  }
  @Bean
  RedisMessageListenerContainer redisMessageListenerContainer
        (RedisConnectionFactory connectionFactory, MessageListenerAdapter listener) {
    RedisMessageListenerContainer container = new RedisMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    container.addMessageListener(listener, ChannelTopic.of("chatroom"));
    return container;
  }
}
```

> 리스너 주제(topic)는 채널(예: topic="chatroom") 또는 패턴(예: topic="*room") 일 수 있습니다.

앞의 예제에서는 Redis 네임스페이스를 사용하여 메시지 리스너 컨테이너를 선언하고 POJO를 리스너로 자동 등록합니다. 본격적인 Bean 정의는 다음과 같습니다.

```
<bean id="messageListener" class="org.springframework.data.redis.listener.adapter.MessageListener
  <constructor-arg>
    <bean class="redisexample.DefaultMessageDelegate"/>
  </constructor-arg>
</bean>

<bean id="redisContainer" class="org.springframework.data.redis.listener.RedisMessageListenerConta
  <property name="connectionFactory" ref="connectionFactory"/>
  <property name="messageListeners">
    <map>
      <entry key-ref="messageListener">
        <bean class="org.springframework.data.redis.listener.ChannelTopic">
          <constructor-arg value="chatroom"/>
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

메시지가 수신될 때마다, 어댑터는 하위 수준 형식과 필요한 개체 유형 간의 변환(configured RedisSerializer를 사용하여)을 자동으로 투명하게 수행합니다. 메서드 호출로 인해 발생한 모든 예외는 컨테이너에 의해 포착 및 처리됩니다(기본적으로 예외는 기록됩니다).

Java Publish/Subscribe Source

Java Spring Framework를 사용한 레디스 Publish/Subscribe 명령 사용법입니다.

사용(실행) 방법

- http://localhost:8080/publish/ch01:Hello

  stringRedisTemplate.convertAndSend(channel "ch01", message "Hello") 메서드를 실행합니다.

레디스 서버 다운/재시작 시 애플리케이션 작동 방식

애플리케이션 실행 중 레디스 서버가 다운되었다 다시 시작하면 Spring 애플리케이션은 레디스 서버에 재접속해서 자동으로 SUBSCRIBE 명령을 다시 실행합니다. 그러므로 별도의 조치없이도 PUBLISH 명령을 실행하면 메시지를 받을 수 있습니다.

RedisSubscriber 클래스

MessageListener 인터페이스의 onMessage()를 구현합니다.
체널명은 message.getChannel()으로 얻을 수 있고, 페턴은 pattern으로 얻을 수 있습니다.

RedisSubscriber.java

```
package com.redisgate.redis;

import lombok.extern.slf4j.Slf4j;
import org.springframework.data.redis.connection.Message;
import org.springframework.data.redis.connection.MessageListener;
import org.springframework.stereotype.Service;

@Service
@Slf4j
public class RedisSubscriber implements MessageListener {

    @Override
    public void onMessage(Message message, byte[] pattern) {
        String ptn = new String(pattern);
        String channel = new String(message.getChannel());
        System.out.println("pattern: "+ptn+", channel: "+channel);
        log.info("Received message: " + channel+" "+message.toString());
    }
}
```

RedisMsgListener 클래스

RedisMessageListenerContainer에 위에서 만든 RedisSubscriber 클래스와 Topic(channel)으로 리스너를 등록합니다. 이곳에서 레디스 서버에 연결되고 SUBSCRIBE 명령이 실행됩니다.
체널은 ChannelTopic()를 사용하고, 패턴은 PatternTopic()을 사용합니다.

RedisMsgListener.java

```
package com.redisgate.redis;

import lombok.extern.slf4j.Slf4j;
import org.springframework.context.annotation.Bean;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.listener.ChannelTopic;
import org.springframework.data.redis.listener.PatternTopic;
import org.springframework.data.redis.listener.RedisMessageListenerContainer;
import org.springframework.stereotype.Repository;

@Repository
@Slf4j
public class RedisMsgListener {
    @Bean
    public RedisMessageListenerContainer redisMessageListenerContainer(
            RedisConnectionFactory connectionFactory,
            RedisSubscriber redisSubscriber
    ) {
        RedisMessageListenerContainer container = new RedisMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);
        container.addMessageListener(redisSubscriber, new ChannelTopic("ch01"));
        container.addMessageListener(redisSubscriber, new PatternTopic("ch*"));
        return container;
    }
}
```

Redis09_PubSub 클래스

PUBLISH 명령을 실행합니다.
사용법: http://localhost:8080/publish/ch01:Hello
ch01:Hello는 체널과 메시지로 구분해서 처리합니다.
stringRedisTemplate의 publish 메서드는 convertAndSend() 입니다.

Redis09_PubSub.java

```
package com.redisgate.redis;

import lombok.extern.slf4j.Slf4j;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
@Slf4j
public class Redis09_PubSub {

    private final StringRedisTemplate stringRedisTemplate;

    public Redis09_PubSub(StringRedisTemplate stringRedisTemplate) {
        this.stringRedisTemplate = stringRedisTemplate;
    }

    // PUBLISH channel message
    // channelMsg channel:message로 구성됩니다.
    // http://localhost:8080/publish/ch01:Hello
    @GetMapping("/publish/{channelMsg}")
    public String publish(@PathVariable("channelMsg") String channelMsg) {
        String[] msg = channelMsg.split(":");
        String result = "Publish "+msg[0]+" "+msg[1];
        System.out.println(result);
        Long num = stringRedisTemplate.convertAndSend(msg[0], msg[1]);
        return result+" -> "+num;
    }
}
```

<< Pipelining Pub/Sub Master/Replica >>

redisgate@gmail.com 02.503.2235
서울시 강남구 강남대로 342 역삼빌딩 5층 (역삼동) 우 06242