链滴

RedisDelayQueue 延迟队列接入方法

作者：shirenchuang

原文链接：https://ld246.com/article/1565798000231

来源网站：链滴

项目已经开源 ,源码地址: RedisDelayQueue

一、引入pom

将项目中的 redis-delay-queue-core 模块打包 推送到自己公司的中央仓库,然后引入pom依赖

```
<dependency>
    <artifactId>redis-delay-queue-core</artifactId>
    <groupId>com.shirc</groupId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

二、将RedisDelayQueue被Spring管理

```
/**
 * @Description 引入 redisdelayqueue
 * @Author shirenchuang
 * @Date 2019/8/6 5:58 PM
 **/
@Component
public class DelayConfig {


    @Autowired
    private RedisTemplate redisTemplate;


    @Bean
    public RedisDelayQueueContext getRdctx(){
        /**传入redisTemplate实例, 第二个参数为项目名 projectName ;不同项目需要设置不一样**/
        RedisDelayQueueContext context =  new RedisDelayQueueContext(redisTemplate,"may
ach-go");
        return context;
    }

    /**加了这个  可以在其他地方直接   @Autowire RedisDelayQueue 使用了**/
    @Bean
    public RedisDelayQueue getRedisOperation(RedisDelayQueueContext context){
        return context.getRedisDelayQueue();
    }
}
```

三、注册Topic任务

```
/**
 * @Description 注册延迟队列 Demo
 * @Author shirenchuang
 * @Date 2019/8/8 10:07 AM
 **/
@Service
public class DelayQueueDemoJob extends AbstractTopicRegister<DemoArgs> {
```

```
@Override
public String getTopic() {
    return DelayJobTopicEnums.DEMO_TEST.getTopic();
}

@Override
public void execute(DemoArgs demoArgs) {
    // 延迟任务回调接口

    //id : 这个Topic下的唯一值
    String id = demoArgs.getId();
    //重试次数;  如果回调接口超时失败, 调用失败,会自动重试2次; 这个代表重试次数
    int retryCount = demoArgs.getRetryCount();

    //DemoArgs 是要继承 Args的;  如果没有自己需要定义的回调参数; 泛型那里直接写  Args就行

    System.out.println(demoArgs.getTest());
}

/******下面方法可选重写  根据消费情况可以自行调节 超时时间,线程池大小等等******/

/**
 * 重试2次仍然失败; 通知接口; 可以在这个接口写自己的通知逻辑; 比如发送邮件或者钉钉消息
 * @param demoArgs
 */
@Override
public void retryOutTimes(DemoArgs demoArgs) {
    super.retryOutTimes(demoArgs);
}

/**
 * 设置核心线程池数量  默认20
 * @return
 */
@Override
public int getCorePoolSize() {
    return super.getCorePoolSize();
}

/**
 * 设置线程池最大线程数量 默认100
 * @return
 */
@Override
public int getMaxPoolSize() {
    return super.getMaxPoolSize();
}

/**
 * 获取 回调方法的超时时间 默认回调接口超时时间  6秒
 * @return
 */
```

```
    @Override
    public int getMethodTimeout() {
        return super.getMethodTimeout();
    }
}
```

上面的DemoArgs

```
/**
 * @Description 回调参数Demo
 * @Author shirenchuang
 * @Date 2019/8/8 10:07 AM
 **/
public class DemoArgs  extends Args {

    private String test;

    public String getTest() {
        return test;
    }

    public void setTest(String test) {
        this.test = test;
    }
}
```

如果没有自己定义的回调属性，或者只需要一个id ,那么泛型那里传 Args就行了,这个会返回id的;

下面就是已经定义好的Args

```
public class Args implements Serializable {

    private static final long serialVersionUID = 66666l;

    /**唯一键 不能为空**/
    private String id;

    /**
     * 已经重试的次数:
     * 重试机制: 默认重试2次; 总共最多执行3次
     * 添加任务的时候可以设置为<0 的值;则表示不希望重试;
     * 回调接口自己做好幂等
     ***/
    private int retryCount;


    /**
     * 重入次数:
     * 这里标记的是当前Job某些异常情况导致并没有真正消费到,然后重新放入待消费池的次数;
     * 比如: BLPOP出来了之后,在去获取Job的时候redis超时了,导致没有正常消费掉;
     * 重入次数最大 3次; 避免某些不可控因素出现,超过3次则丢弃
     */
    private int reentry;

    public Args() {
```

```
    }

    public Args(String id) {
        this.id = id;
    }

    public Args(String id, int retryCount) {
        this.id = id;
        this.retryCount = retryCount;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public int getRetryCount() {
        return retryCount;
    }

    public void setRetryCount(int retryCount) {
        this.retryCount = retryCount;
    }

    public int getReentry() {
        return reentry;
    }

    public void setReentry(int reentry) {
        this.reentry = reentry;
    }

    @Override
    public String toString() {
        return "Args{" +
                "id='" + id + '\'' +
                ", retryCount=" + retryCount +
                ", reentry=" + reentry +
                '}';
    }
```

四、建议新建一个Topic的枚举类

因为 新增任务的Topic 和 注册地方的Topic,还有删除Topic 要一致,建议用枚举

```
/**
 * @Description 所有延迟任务的Topic
 * @Author shirenchuang
 **/
public enum DelayJobTopicEnums {
```

```
    DEMO_TEST("DEMO_TEST","测试"),
    ;

    private String topic;

    private String desc;

    DelayJobTopicEnums(String topic, String desc) {
        this.topic = topic;
        this.desc = desc;
    }

    public String getTopic() {
        return topic;
    }

    public String getDesc() {
        return desc;
    }

}
```

五、如何新增一个延迟任务 、删除一个延迟任务

```
/**
 * @Description 新增删除延迟任务
 * @Author shirenchuang
 * @Date 2019/8/8 10:29 AM
 **/
@Component
public class DelayQueueUseDemo {

    @Autowired
    RedisDelayQueue redisDelayQueue;


    private void addDelayQueue(){

        //do something

        //新增一个延迟任务
        DemoArgs demoArgs = new DemoArgs();
        demoArgs.setId(UUID.randomUUID().toString());
        //设置-1 表示我不想要重试
        demoArgs.setRetryCount(-1);
        demoArgs.setTest("我是个Test");
        //异步新增一个  一分钟之后执行的延时任务
        redisDelayQueue.addAsync(demoArgs,DelayJobTopicEnums.DEMO_TEST.getTopic(),600
0);

        //也可以同步新增延迟任务
        redisDelayQueue.add(demoArgs,60000,DelayJobTopicEnums.DEMO_TEST.getTopic(),Run
ypeEnum.SYNC);
```

```
        //也可以指定某个时间点执行
        redisDelayQueue.add(demoArgs,DelayJobTopicEnums.DEMO_TEST.getTopic(),System.cur
entTimeMillis()+60000,RunTypeEnum.ASYNC);

        /**PS:如果同一个ID添加了多次,以最新添加的为准,会覆盖之前的**/


        /**如果自己没有需要回调的参数 直接用Args**/
        redisDelayQueue.addAsync(new Args(id),DelayJobTopicEnums.DEMO_TEST.getTopic(),60
00);
    }


    //删除一个之前添加的延迟任务
    private void  delDelayQueue(String id){
        //异步删除
        redisDelayQueue.deleteAsync(DelayJobTopicEnums.DEMO_TEST.getTopic(),id);
        //同步删除
        redisDelayQueue.delete(DelayJobTopicEnums.DEMO_TEST.getTopic(),id,RunTypeEnum.S
NC);
    }
}
```

六、配置日志

在logback.xml 里面新增如下配置

```
<!-- 加入 redis-delay-queue的日志配置    -->
<appender name="redis_dq_file" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>${LOG_HOME}/redis_dq.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <!-- 日志文件输出文件名 -->
        <FileNamePattern>${LOG_HOME}/redis_dq.log.%d{yyyy-MM-dd}</FileNamePattern>
        <!-- 日志文件保留天数 -->
        <MaxHistory>30</MaxHistory>
    </rollingPolicy>
    <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
        <!-- 格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度
%msg：日志消息，%n是换行符 -->
        <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n<
pattern>
    </encoder>
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
        <!-- 过滤掉低于INFO级别的日志 -->
        <level>INFO</level>
    </filter>
</appender>

<!--  延迟任务异常日志 -->
<appender name="redis_dq_error_file" class="ch.qos.logback.core.rolling.RollingFileAppende
">
    <file>${LOG_HOME}/redis_dq_error.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <!-- 日志文件输出文件名 -->
```

```
        <FileNamePattern>${LOG_HOME}/redis_dq_error.log.%d{yyyy-MM-dd}</FileNamePatte
n>
        <!-- 日志文件保留天数 -->
        <MaxHistory>30</MaxHistory>
    </rollingPolicy>
    <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
        <!-- 格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度
%msg：日志消息，%n是换行符 -->
        <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n<
pattern>
    </encoder>
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
        <!-- 过滤掉低于ERROR级别的日志 -->
        <level>ERROR</level>
    </filter>
</appender>


<!-- redis_delay_queue 日志 -->
<logger name="com.shirc.redis.delay.queue" level="INFO" additivity="false">
    <appender-ref ref="redis_dq_file" />
</logger>
<logger name="com.shirc.redis.delay.queue" level="INFO" additivity="false">
    <appender-ref ref="redis_dq_error_file" />
</logger>
```

然后所有的日志都在 redis_dq.log 中；所有的异常日志都在 redis_dq_error.log 中;