Redis + Redission + Cache

Redis

一、依赖

```xml
<!-- redis: 2.3.12.RELEASE & redis连接池: 2.6.2 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>

<!-- fastjson: 1.2.79 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
</dependency>

<!-- test: 2.3.12.RELEASE -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

二、yml配置

```yaml
# spring
spring:
  redis:
    timeout: 6000ms     # 连接超时时长
    password:           # 密码（默认为空）
    # lettuce连接池
    # SpringBoot2.0默认采用Lettuce客户端来连接Redis服务端，替换为jedis使用jedis
    # Lettuce基于Netty线程安全，Jedis直接连接redis server非线程安全
    lettuce:
```

```yaml
      pool:
        max-active: 1000  # 连接池最大连接数（使用负值表示没有限制）
        max-idle: 10      # 连接池中的最大空闲连接
        min-idle: 5       # 连接池中的最小空闲连接
        max-wait: -1      # 连接池最大阻塞等待时间（使用负值表示没有限制）
    # 单机配置
    host: 10.207.0.169    # 默认port=6379, database=0
    port: 6379
    # 集群配置
#    cluster:
#      nodes: # Redis的各个端口号
#        - 10.207.0.169:6379
#        - 10.207.0.169:6380
#      max-redirects: 3 # 获取失败 最大重定向次数


# 注解方式redisson初始化策略: 只能有一个Y，其余可不写
annotation:
  redisson:
    strategy:
      single: Y
      cluster:
      sentinel:
```

三、配置类

```java
@Configuration
@AutoConfigureAfter(RedisAutoConfiguration.class)  //根据yml配置，自动选择注入
public class RedisConfig {

    /**
     * Jackson2JsonRedisSerializer & GenericJackson2JsonRedisSerializer
     * 1.序列化带泛型的数据时，会以map的结构进行存储，反序列化是不能将map解析成对
     * 2.使用Jackson2JsonRedisSerializer需要指明序列化的类Class，可以使用Obejc
     * 3.使用GenericJacksonRedisSerializer比Jackson2JsonRedisSerializer效率低
     * 4.Jackson2JsonRedisSerializer反序列化带泛型的数组类会报转换异常，解决办法
     */
    @Bean
    public RedisTemplate<String, Object> redisCacheTemplate(LettuceConnectio
redisConnectionFactory) {
        RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>()
        redisTemplate.setKeySerializer(new StringRedisSerializer());
```

```
        redisTemplate.setValueSerializer(new GenericJackson2JsonRedisSerial
        redisTemplate.setConnectionFactory(redisConnectionFactory);
        return redisTemplate;
    }
}
```

四、工具类

```
@Component
public class RedisService {

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    /**
     * 设置指定 key 的值
     */
    public void set(String key, Object value) {
        if (null == key || null == value) {
            return;
        }
        redisTemplate.opsForValue().set(key, value);
    }

    /**
     * 设置key 的值 并设置过期时间
     */
    public void set(String key, Object value, long time, TimeUnit unit) {
        if (null == key || null == value || null == unit) {
            return;
        }
        redisTemplate.opsForValue().set(key, value, time, unit);
    }

    /**
     * 获取指定Key的Value。如果与该Key关联的Value不是string类型，Redis将抛出异常
     * 因为GET命令只能用于获取string Value，如果该Key不存在，返回null
     */
    public Object get(String key) {
        if (null == key) {
            return null;
        }
```

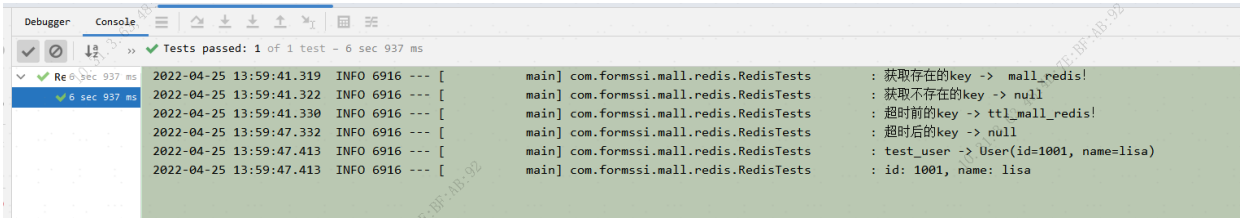
```
        return redisTemplate.opsForValue().get(key);
    }

    /**
     * 设置过期时间
     */
    public Boolean expire(String key, long timeout, TimeUnit unit) {
        if (null == key || null == unit) {
            return false;
        }
        return redisTemplate.expire(key, timeout, unit);
    }
}
```

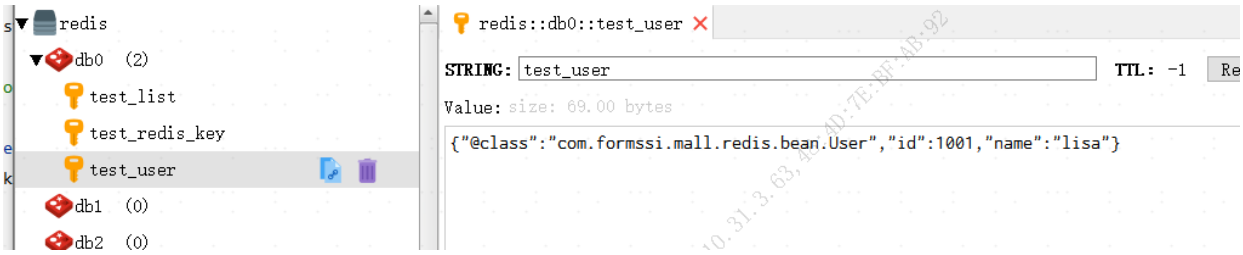
五、测试

```
@Slf4j
@RunWith(SpringRunner.class)
@SpringBootTest
public class RedisTests {

    @Autowired
    private RedisService redisService;

    @Test
    public void testSet() throws Exception{
        //set
        redisService.set("test_redis_key", "mall_redis! ");
        log.info("获取存在的key ->  {}", redisService.get("test_redis_key"))
        log.info("获取不存在的key -> {}", redisService.get("my_key"));

        //set + ttl
        redisService.set("test_redis_ttl_key", "ttl_mall_redis! ", 5, TimeUr
        log.info("超时前的key -> {}", redisService.get("test_redis_ttl_key")
        TimeUnit.SECONDS.sleep(6);
        log.info("超时后的key -> {}", redisService.get("test_redis_ttl_key")

        //object
        User user = new User(1001, "lisa");
        redisService.set("test_user", user);
//{"@class":"com.formssi.mall.redis.bean.User","id":1001,"name":"lisa"}
```

```
        Object objUser = redisService.get("test_user");
```

```
        log.info("test_user -> {}", objUser);
        User user2 = (User)objUser;
        log.info("id: {}, name: {}", user2.getId(), user2.getName());
    }
}
```





Redisson

定义

Redisson是一个高级的分布式协调Redis客服端，底层使用netty框架，并提供了与java对象相对应的分布式对象、分布式集合、分布式锁和同步器、分布式服务等一系列的Redisson的分布式对象。Redisson、Jedis、Lettuce 是三个不同的操作 Redis 的客户端，Jedis、Lettuce 的 API 更侧重对 Reids 数据库的 CRUD（增删改查），而 Redisson API 侧重于分布式开发

一、依赖

```
<!-- redisson -->
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.17.0</version>
</dependency>
```

二、配置 - 基于工厂 - 方式一

1.config三种策略：单机、集群、哨兵

```java
@Getter
@AllArgsConstructor
public enum RedissonEnum {

    //前缀
    URL_PREFIX("redis://", "Redis地址配置前缀"),
    SSL_URL_PREFIX("rediss://", "Redis地址配置SSL前缀"),

    //策略
    SINGLE_STRATEGY("SINGLE_STRATEGY", "单机模式"),
    CLUSTER_STRATEGY("CLUSTER_STRATEGY", "集群模式"),
    SENTINE_STRATEGY("SENTINE_STRATEGY", "哨兵模式");
    ;

    private final String value;
    private final String desc;
}

//定义策略接口
public interface RedissonConfigStrategy {
    Config createRedissonConfig(RedisProperties redisProperties);
}

@Slf4j
public class SingleConfigImpl implements RedissonConfigStrategy {
    // 单机模式
    @Override
    public Config createRedissonConfig(RedisProperties redisProperties) {
        Config config = new Config();
        try {
            String address = redisProperties.getHost().concat(":").concat(redisProperties.getPort()+"");
            String password = redisProperties.getPassword();
            int database = redisProperties.getDatabase();
            String redisAddr = redisProperties.isSsl() ? RedissonEnum.SSL_U RedissonEnum.URL_PREFIX.getValue();
            redisAddr += address;
            config.useSingleServer().setAddress(redisAddr);
            config.useSingleServer().setDatabase(database);
            //密码可以为空
            if (StringUtils.isNotBlank(password)) {
                config.useSingleServer().setPassword(password);
            }
```

```
            log.info("部署策略[单机模式] -> redisAddress:{}", address);
        } catch (Exception e) {
            log.error("部署策略[单机模式] redisson init error!", e);
            e.printStackTrace();
        }
        return config;
    }
}

@Slf4j
public class ClusterConfigImpl implements RedissonConfigStrategy {
    // 集群模式
    @Override
    public Config createRedissonConfig(RedisProperties redisProperties) {
        Config config = new Config();
        try {
            List<String> address = redisProperties.getCluster().getNodes();
            String password = redisProperties.getPassword();
            //String[] addrTokens = address.split(",");
            //设置cluster节点的服务IP和端口
            String redisAddr = redisProperties.isSsl() ? RedissonEnum.SSL_UR
 RedissonEnum.URL_PREFIX.getValue();
            for (String addres : address) {
                config.useClusterServers()
                      .addNodeAddress(redisAddr + address);
                if (StringUtils.isNotBlank(password)) {
                    config.useClusterServers().setPassword(password);
                }
            }
            log.info("部署策略[集群模式] -> redisAddress:{}", address);
        } catch (Exception e) {
            log.error("部署策略[集群模式] redisson init error!", e);
            e.printStackTrace();
        }
        return config;
    }
}

@Slf4j
public class SentineConfigImpl implements RedissonConfigStrategy {
    // 哨兵模式
    @Override
    public Config createRedissonConfig(RedisProperties redisProperties) {
```

```
84          Config config = new Config();
85          try {
86              String masterAddress = redisProperties.getSentinel().getMaster(
87              String password = redisProperties.getPassword();
88              int database = redisProperties.getDatabase();
89              List<String> nodeAddress = redisProperties.getSentinel().getNod
90              String sentinelAliasName = Optional.ofNullable(nodeAddress)
91                                                 .map(u-> nodeAddress.get(0)
92                                                 .orElseThrow(()->new
RuntimeException("redis.sentinel.nodes is null, please set"));
93              //设置redis配置文件sentinel.conf配置的sentinel别名
94              config.useSentinelServers().setMasterName(sentinelAliasName);
95              config.useSentinelServers().setDatabase(database);
96              if (StringUtils.isNotBlank(password)) {
97                  config.useSentinelServers().setPassword(password);
98              }
99              //设置sentinel节点的服务IP和端口
100             String redisAddr = redisProperties.isSsl() ? RedissonEnum.SSL_U
 RedissonEnum.URL_PREFIX.getValue();
101             for (String nodeAddres : nodeAddress) {
102                 config.useSentinelServers().addSentinelAddress(redisAddr + 
103             }
104             log.info("部署策略[哨兵模式] -> masterAddress:{}, nodeAddress:{}"
nodeAddress);
105         } catch (Exception e) {
106             log.error("部署策略[哨兵模式] redisson init error!", e);
107             e.printStackTrace();
108         }
109         return config;
110     }
111 }
112 
113 //策略工厂
114 @Component
115 public class RedissonStrategyFactory {
116 
117     /**
118      *  基于策略模式，获取不同的连接
119      */
120     public Config createConfig(RedisProperties redisProperties, RedissonEnu
121         Objects.requireNonNull(redisProperties, "redisProperties 不能为空!")
122         if (strategyEnum == null) {
123             return new SingleConfigImpl().createRedissonConfig(redisPropert
```

```
        }
        //默认单机
        RedissonConfigStrategy redissonConfigStrategy;
        switch (strategyEnum){
            case CLUSTER_STRATEGY:
                redissonConfigStrategy = new ClusterConfigImpl();
                break;
            case SENTINE_STRATEGY:
                redissonConfigStrategy = new SentineConfigImpl();
                break;
            default:
                redissonConfigStrategy = new SingleConfigImpl();
                break;
        }
        return redissonConfigStrategy.createRedissonConfig(redisProperties)
    }
}
```

2.配置类

```
@Configuration
@ConditionalOnClass(Redisson.class)
@EnableConfigurationProperties(RedisProperties.class)
public class RedissonConfig {
    private static final Logger LOGGER = LoggerFactory.getLogger(RedissonCo

    @Autowired
    RedissonStrategyFactory redissonStrategyFactory;

    @Bean
    @ConditionalOnMissingBean
    public Redisson redisson(RedisProperties redisProperties) {
        Config config = redissonStrategyFactory.createConfig(redisPropertie
RedissonEnum.SINGLE_STRATEGY);  //单机模式
        Redisson redisson = (Redisson) Redisson.create(config);
        LOGGER.info("redisson 初始化完成.");
        return redisson;
    }
}
```

三、配置 - 注解Bean - 方式二

```java
public class RedisConstant {
    public static final String CONDITION_PREFIX = "annotation.redisson.stra
    public static final String CONDITION_YES = "Y";
    public static final String REDISSON_STRATEGY_SINGLE = "single";
    public static final String REDISSON_STRATEGY_CLUSTER = "cluster";
    public static final String REDISSON_STRATEGY_SENTINEL = "sentinel";
}

@Configuration
@ConditionalOnClass(Redisson.class)
public class RedissonConfig {
    private static final Logger LOGGER = LoggerFactory.getLogger(RedissonCo

    /**
     * 策略 - 注解方式
     */
    @Autowired
    RedisProperties redisProperties;

    @Bean
    @ConditionalOnProperty(prefix = RedisConstant.CONDITION_PREFIX, name =
RedisConstant.REDISSON_STRATEGY_SINGLE, havingValue = RedisConstant.CONDITI
    public Redisson singleRedisson() {
        return initRedisson((config, redisProperties) -> {
            String address =
redisProperties.getHost().concat(":").concat(redisProperties.getPort()+"");
            String password = redisProperties.getPassword();
            int database = redisProperties.getDatabase();
            String redisAddr = redisProperties.isSsl() ? RedissonEnum.SSL_U
 RedissonEnum.URL_PREFIX.getValue();
            redisAddr += address;
            config.useSingleServer().setAddress(redisAddr);
            config.useSingleServer().setDatabase(database);
            //密码可以为空
            if (StringUtils.isNotBlank(password)) {
                config.useSingleServer().setPassword(password);
            }
            LOGGER.info("部署策略[单机模式] -> redisAddress: {}", address);
            return  (Redisson) Redisson.create(config);
        });
    }

    @Bean
    @ConditionalOnProperty(prefix = RedisConstant.CONDITION_PREFIX, name =
```

```
RedisConstant.REDISSON_STRATEGY_CLUSTER, havingValue = RedisConstant.CONDIT
    public Redisson clusterRedisson() {
        return initRedisson((config, redisProperties) -> {
            List<String> address = redisProperties.getCluster().getNodes();
            String password = redisProperties.getPassword();
            //String[] addrTokens = address.split(",");
            //设置cluster节点的服务IP和端口
            String redisAddr = redisProperties.isSsl() ? RedissonEnum.SSL_U
 RedissonEnum.URL_PREFIX.getValue();
            for (String addres : address) {
                config.useClusterServers()
                        .addNodeAddress(redisAddr + addres);
                if (StringUtils.isNotBlank(password)) {
                    config.useClusterServers().setPassword(password);
                }
            }
            LOGGER.info("部署策略[集群模式] -> redisAddress: {}", address);
            return  (Redisson) Redisson.create(config);
        });
    }

    @Bean
    @ConditionalOnProperty(prefix = RedisConstant.CONDITION_PREFIX, name =
RedisConstant.REDISSON_STRATEGY_SENTINEL, havingValue = RedisConstant.CONDI
    public Redisson sentinelRedisson() {
        return initRedisson((config, redisProperties) -> {
            String masterAddress = redisProperties.getSentinel().getMaster(
            String password = redisProperties.getPassword();
            int database = redisProperties.getDatabase();
            List<String> nodeAddress = redisProperties.getSentinel().getNod
            String sentinelAliasName = Optional.ofNullable(nodeAddress)
                    .map(u-> nodeAddress.get(0))
                    .orElseThrow(() -> new RuntimeException("redis.sentinel
please set"));
            //设置redis配置文件sentinel.conf配置的sentinel别名
            config.useSentinelServers().setMasterName(sentinelAliasName);
            config.useSentinelServers().setDatabase(database);
            if (StringUtils.isNotBlank(password)) {
                config.useSentinelServers().setPassword(password);
            }
            //设置sentinel节点的服务IP和端口
            String redisAddr = redisProperties.isSsl() ? RedissonEnum.SSL_U
 RedissonEnum.URL_PREFIX.getValue();
            for (String nodeAddres : nodeAddress) {
```

```
                config.useSentinelServers().addSentinelAddress(redisAddr + r
            }
            LOGGER.info("部署策略[哨兵模式] -> masterAddress: {}, nodeAddress
nodeAddress);
            return  (Redisson) Redisson.create(config);
        });
    }

    private Redisson initRedisson(BiFunction<Config, RedisProperties, Redis
        Redisson redisson = null;
        Config config = new Config();
        try {
            LOGGER.info("redisson init start...");
            redisson = func.apply(config, redisProperties);
            LOGGER.info("redisson init complete.");
        } catch (Exception e) {
            LOGGER.error("redisson init error!", e);
            e.printStackTrace();
        }
        return redisson;
    }
}
```

四、工具类

```
@Slf4j
@Data
@Component
@ConditionalOnBean(Redisson.class)
public class RedissonLockService {

    @Autowired
    private Redisson redisson;

    /**
     * 加锁操作 （设置锁的有效时间）
     * @param lockName 锁名称
     * @param leaseTime  锁有效时间
     */
    public void lock(String lockName, long leaseTime) {
        RLock rLock = redisson.getLock(lockName);
```

```
        rLock.lock(leaseTime,TimeUnit.SECONDS);
```

```
    }

    /**
     * 加锁操作 (锁有效时间采用默认时间30秒)
     * @param lockName 锁名称
     */
    public void lock(String lockName) {
        RLock rLock = redisson.getLock(lockName);
        rLock.lock();
    }

    /**
     * 加锁操作(tryLock锁，没有等待时间)
     * @param lockName  锁名称
     * @param leaseTime 锁有效时间
     */
    public boolean tryLock(String lockName, long leaseTime) {

        RLock rLock = redisson.getLock(lockName);
        boolean getLock = false;
        try {
            getLock = rLock.tryLock( leaseTime, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            log.error("获取Redisson分布式锁[异常]，lockName=" + lockName, e);
            e.printStackTrace();
            return false;
        }
        return getLock;
    }

    /**
     * 加锁操作(tryLock锁，有等待时间)
     * @param lockName   锁名称
     * @param leaseTime  锁有效时间
     * @param waitTime   等待时间
     */
    public  boolean tryLock(String lockName, long leaseTime,long waitTime)

        RLock rLock = redisson.getLock(lockName);
        boolean getLock = false;
        try {
            getLock = rLock.tryLock( waitTime,leaseTime, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
```

```
            log.error("获取Redisson分布式锁[异常]，lockName=" + lockName, e);
            e.printStackTrace();
            return false;
        }
        return getLock;
    }

    /**
     * 解锁
     * @param lockName  锁名称
     */
    public void unlock(String lockName) {
        redisson.getLock(lockName).unlock();
    }

    /**
     * 判断该锁是否已经被线程持有
     * @param lockName  锁名称
     */
    public boolean isLock(String lockName) {
        RLock rLock = redisson.getLock(lockName);
        return rLock.isLocked();
    }

    /**
     * 判断该线程是否持有当前锁
     * @param lockName  锁名称
     */
    public boolean isHeldByCurrentThread(String lockName) {
        RLock rLock = redisson.getLock(lockName);
        return rLock.isHeldByCurrentThread();
    }

    /**
     * 存储缓存
     */
    public <T> void setBucket(String key, T value) {
        RBucket<T> bucket = redisson.getBucket(key,  StringCodec.INSTANCE);
        bucket.set(value);
    }

    /**
     * 存储过期缓存
```

```
104      */
105     public <T> void setBucket(String key, T value, long expired) {
106         RBucket<T> bucket = redisson.getBucket(key, JsonJacksonCodec.INSTAN
107         bucket.set(value, expired, TimeUnit.SECONDS);
108     }
109 
110     /**
111      * 读取缓存
112      */
113     public <T> T getBucket(String key, Class<T> clazz) {
114         RBucket<T> bucket = redisson.getBucket(key, JsonJacksonCodec.INSTAN
115         return bucket.get();
116     }
117 
118     /**
119      * 发布消息
120      */
121     public <T> long publishMessage(String topic, T message) {
122         //T要实现序列化接口
123         RTopic rTopic = redisson.getTopic(topic, new SerializationCodec());
124         long l = rTopic.publish(message);
125         log.info("message push success.");
126         return l;
127     }
128 
129     /**
130      * 订阅消息
131      */
132     public <T> void receiveMessage(String topic, Class<T> clazz, BiConsumer<
    T> consumer) {
133         RTopic rTopic = redisson.getTopic(topic, new SerializationCodec());
134         rTopic.addListenerAsync(clazz, (charSequence, msg) -> {
135                 //消费的具体内容
136                 consumer.accept(charSequence, msg);
137                 log.info("message consumer complete.");
138             }
139         );
140     }
141 }
```

五、测试

```java
//注解方式加锁
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface DistributedLock {
    /**
     * 锁的名称
     */
    String value() default "redisson";

    /**
     * 锁的有效时间
     */
    int leaseTime() default 10;
}

@Slf4j
@Aspect
@Component
public class DistributedLockAspect {

    @Autowired
    RedissonLockService redissonLockService;

    @Around("@annotation(distributedLock)")
    public void around(ProceedingJoinPoint joinPoint, DistributedLock distr
        log.info("分布式锁-lock");
        //获取锁名称
        String lockName = distributedLock.value();
        //获取超时时间，默认10秒
        int leaseTime = distributedLock.leaseTime();
        boolean isLock = redissonLockService.tryLock(lockName, leaseTime);
        try {
            if (isLock){
                log.info("加锁成功，开始执行业务...");
                joinPoint.proceed();
                log.info("业务执行完成.");
            }else {
                log.info("尝试获取锁失败，稍后重试!");
                TimeUnit.MILLISECONDS.sleep(200);
            }
        } catch (Throwable throwable) {
```

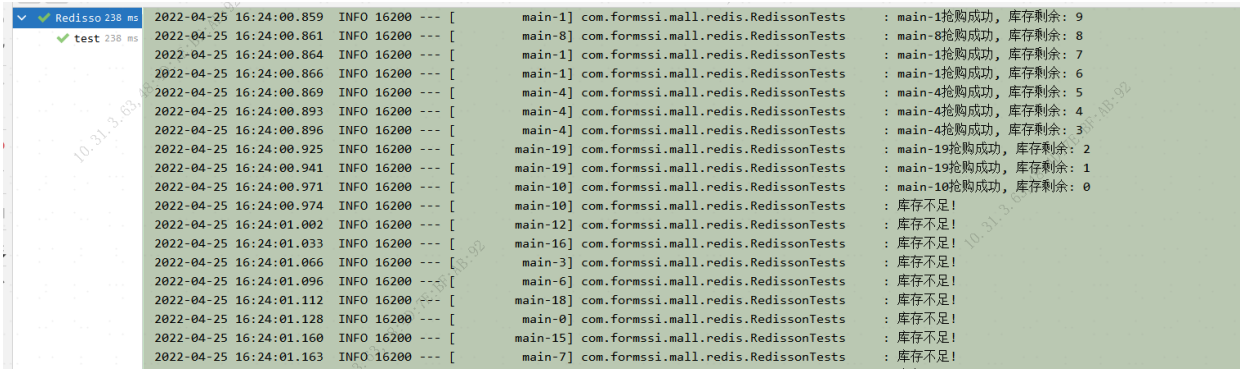
```
            log.error("加锁失败: ", throwable);
            throwable.printStackTrace();
        } finally {
            //如果该线程还持有该锁，那么释放该锁。如果该线程不持有该锁，说明该线程
放锁
            if (redissonLockService.isHeldByCurrentThread(lockName)) {
                redissonLockService.unlock(lockName);
            }
        }
        log.info("分布式锁-unlock");
    }
}

//测试类
@Slf4j
@RunWith(SpringRunner.class)
@SpringBootTest
public class RedissonTests {

    @Autowired
    private RedissonLockService redissonLockService;

    private int goodTotal = 100;
    private final String LOCK_KEY = "test_redisson_lock";

    /**
     * 手动加锁
     */
    @Test
    public void testLock() throws Exception{
        for (int i = 0; i < 20; i++) {
            new Thread(() -> {
                while (true){
                    try {
                        boolean isLock = redissonLockService.tryLock(LOCK_K
                        if (isLock){
                            if (goodTotal > 0){
                                goodTotal-=1;
                                log.info("{}抢购成功，库存剩余: {}",
Thread.currentThread().getName(), goodTotal);
                            }else {
                                log.info("库存不足!");

                                break;
```

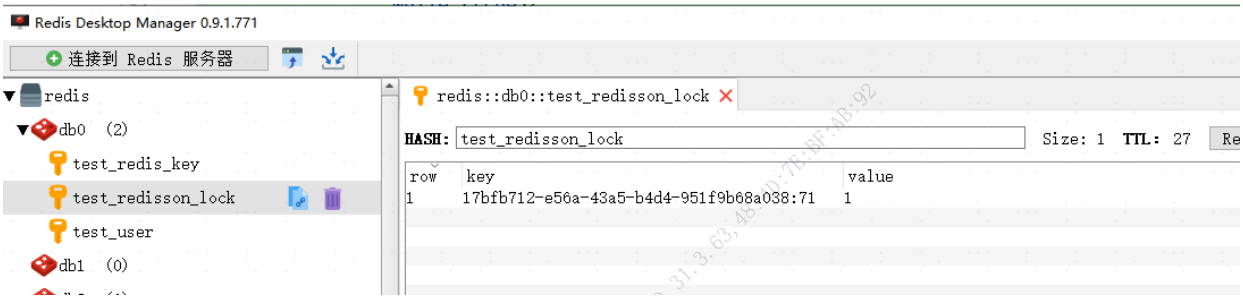
```
85                          }
86                      }else{
87                          log.info("尝试获取锁失败!");
88                          throw new RuntimeException("尝试获取锁失败!");
89                      }
90                  }finally {
91                      redissonLockService.unlock(LOCK_KEY);
92                  }
93              }
94          }, Thread.currentThread().getName() + "-" + i).start();
95      }
96  }
97
98
99  /**
100  * 注解-自动加锁
101  */
102 @Test
103 public void testAnnoLock() throws Exception{
104     for (int i = 0; i < 20; i++) {
105         new Thread(() -> {
106             while (true){
107                 if(!getGoods()){
108                     break;
109                 };
110             }
111         }, Thread.currentThread().getName() + "-" + i).start();
112     }
113 }
114
115 @DistributedLock(LOCK_KEY)
116 boolean getGoods(){
117     if (goodTotal > 0){
118         goodTotal-=1;
119         log.info("{}抢购成功, 库存剩余: {}", Thread.currentThread().getNa
120         return true;
121     }else {
122         log.info("库存不足!");
123         return false;
124     }
125 }
```

```
126
```

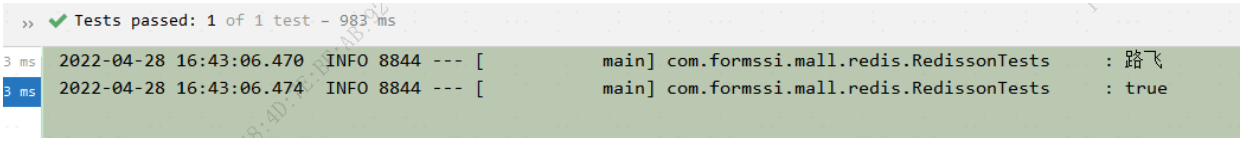
```
127     @Test
128     public void testBucket() throws Exception{
129         String userKey = "bucket_user_key";
130         //设值
131         redissonLockService.setBucket(userKey, new User(2022, "路飞"), 5*60L
132         //取值
133         User user = redissonLockService.getBucket(userKey, User.class);
134         log.info(user.getName());
135         //是否存在
136         boolean isExist = redissonLockService.existsBucket(userKey);
137         log.info(Boolean.toString(isExist));
138         //删除
139         //redissonLockService.removeBucket(userKey);
140     }
141
142     final String topic = "redisson_topic_test";
143     @Test
144     public void testPush() throws Exception{
145         //发布
146         long rs = redissonLockService.publishMessage(topic, new User(2022,
147         redissonLockService.publishMessage(topic, new User(2023, "乔巴"));
148         log.info(Long.toString(rs));
149     }
150     @Test
151     public void testSubscribe() throws Exception{
152         //订阅
153         redissonLockService.receiveMessage(topic, User.class, (charSequence
154             log.info("主题：{}，消息：{}", charSequence, msg);
155         });
156     }
157     @Test
158     public void test() throws Exception{
159         //先执行消费者，再执行生产者
160         testSubscribe();
161         testPush();
162     }
163
164 }
```

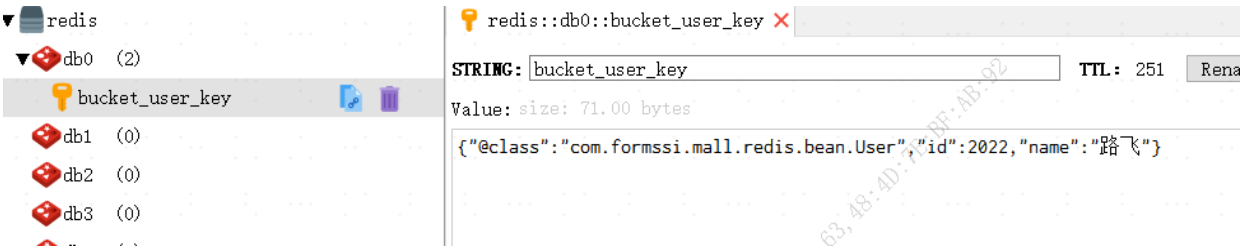
Lock锁测试





Bucket桶测试





发布-订阅测试

Spring Cache

为什么使用?

1.redis缓存操作和业务逻辑之间的代码耦合度高，对业务逻辑有较强的侵入性

2.每次操作都需要定义缓存Key ，调用缓存命令的API，产生较多的重复代码

3.某些场景下，需要更换缓存组件，每个缓存组件有自己的API，更换成本颇高

4.频繁的查询redis，也会有明显的网络IO上的消耗。对于热点key可以采用应用缓存(一级缓存)，redis做二级缓存

一、依赖

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

二、yml配置

```
spring:
    cache:
      type: redis                  #缓存类型
      redis:
        cache-null-values: false   #不缓存null数据，解决[缓存穿透]可设置为true
        time-to-live: 120          #缓存数据过期时间，单位s
        use-key-prefix: false      #不适用前缀
```

三、config配置

```
@Configuration
@AutoConfigureAfter(RedisAutoConfiguration.class)
@EnableCaching  //开启缓存(必须))
public class RedisConfig {

    private final Duration timeToLive = Duration.ofDays(7);  //redis ttl过期
    private final StringRedisSerializer keySerializer = new StringRedisSeria
    private final GenericJackson2JsonRedisSerializer valueSerializer = new
GenericJackson2JsonRedisSerializer();

    @Bean
    public RedisTemplate<String, Object> redisCacheTemplate(LettuceConnectio
redisConnectionFactory) {
        RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>()
        redisTemplate.setKeySerializer(keySerializer);
        redisTemplate.setValueSerializer(valueSerializer);
        redisTemplate.setHashKeySerializer(keySerializer);
        redisTemplate.setHashValueSerializer(valueSerializer);
        redisTemplate.setConnectionFactory(redisConnectionFactory);
        return redisTemplate;
    }
```

```
20
21      /**
22       * Cache缓存管理器
23       */
24      @Bean
25      public RedisCacheManager cacheManager(RedisConnectionFactory redisConne
26          // 生成一个默认配置，通过config对象即可对缓存进行自定义配置
27          RedisCacheConfiguration config = RedisCacheConfiguration.defaultCac
28          // 设置缓存的默认过期时间，也是使用Duration设置
29          config = config
30                  //不设置默认-1，永不过期
31                  .entryTtl(timeToLive)
32                  // 设置 key为string序列化
33  .serializeKeysWith(RedisSerializationContext.SerializationPair.fromSerializ
34                  // 设置value为json序列化
35  .serializeValuesWith(RedisSerializationContext.SerializationPair.fromSerial
36                  // 不缓存空值
37                  .disableCachingNullValues()
38                  // 覆盖默认的构造key，否则会多出一个冒号 原规则：cacheName::key
39                  .computePrefixWith(name -> name + ":")
40                  ;
41
42          // 对每个缓存空间(cacheNames属性)应用不同的配置
43          Map<String, RedisCacheConfiguration> configMap = new HashMap<>();
44          configMap.put("MallCache", config.entryTtl(Duration.ofMinutes(30)))
45
46          // 使用自定义的缓存配置初始化一个cacheManager
47          RedisCacheManager cacheManager = RedisCacheManager.builder(redisCon
48                  //默认配置
49                  .cacheDefaults(config)
50                  // 特殊配置（一定要先调用该方法设置初始化的缓存名，再初始化相关的
51                  .initialCacheNames(configMap.keySet())
52                  .withInitialCacheConfigurations(configMap)
53                  //开启redis事务(和jdbc事务相关联)
54                  .transactionAware()
55                  .build();
56
57          return cacheManager;
58      }
59
60      /**
```

```
    * 自定义json序列化器
    */
    private RedisSerializer<Object> jackson2JsonRedisSerializer() {
        //使用Jackson2JsonRedisSerializer来序列化和反序列化redis的value值
        Jackson2JsonRedisSerializer<Object> serializer = new Jackson2JsonRe
(Object.class);
        //json转对象类，不设置默认的会将json转成hashmap
        ObjectMapper mapper = new ObjectMapper();
        mapper.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibilit
        mapper.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        serializer.setObjectMapper(mapper);
        return serializer;
    }
}
```

四、测试

```
@Slf4j
@CacheConfig(cacheNames = "userCache", cacheManager = "cacheManager")   //统
@Service
public class UserService {

    //key中的: 为目录分割符
    //sync同步阻塞，解决[缓存击穿]
    @Cacheable(key = "'userId:' + #id", sync = true)
    public User getUserById(Integer id) {
        log.info("getUserById -> Cache no data, Query DB & Add Cache");
        //返回值进缓存
        return new User(id, "查询不到时，加入的缓存");
    }

    //插入符合条件（1005 < id <= 1010）的，不符合的不插入cache，但会执行方法体
    //已经存在cache则不执行方法体
    @Cacheable(key = "'userId:' + #user.id", condition = "#p0.id > 1005", u
1010")
    public User addUserByCondition(User user) {
        log.info("addUserByCondition -> Insert DB");
        return user;
    }

    //存在cache也会执行方法体，然后更新Cache
```

```
    @CachePut(key = "'userId:' + #user.id")
```

```
    public User addUserByPut(User user) {
        log.info("addUserByPut -> Insert DB");
        return user;
    }

    //userCache下全被删除，删除db也执行了
    //无cache，只执行方法体
    @CacheEvict(key = "'userId:' + #user.id", beforeInvocation = true , all
    public void delUserById(User user) {
        log.info("delUserById -> Delete DB");
    }

    //批量执行多个@CacheXXX
    @Caching(
        cacheable = {@Cacheable(key = "'userId:' + #user.id")}
        , put = {@CachePut(key = "'userName:' + #user.name")}
        , evict = {@CacheEvict(key = "'userId:' + #user.id")}
    )
    public User addUserByCaches(User user) {
        log.info("addUserByCaches -> Insert DB");
        return user;
    }
}
```

```
/**
 * @author jp
 * @version 1.0
 * @Description: SpringCache测试类
 * @date 2022/4/25 17:25
 *
 * @CacheConfig: 在类级别共享缓存的相同配置
 * @Cacheable: 方法返回值加入缓存。同时在查询时，会先从缓存中取，若不存在才再发起
 *      value、cacheNames：两个等同的参数（cacheNames为Spring 4新增，作为value
储的集合名
 *      key：缓存对象存储在Map集合中的key值，非必需，缺省按照函数的所有参数组合作
用SpEL表达式，比如：@Cacheable(key = “#p0”)：使用函数第一个参数作为缓存的key值
 *      condition：缓存对象的条件，非必需，也需使用SpEL表达式，只有满足表达式条件
@Cacheable(key = “#p0”, condition = “#p0.length() < 3”)，表示只有当第一个参数
缓存
 *      unless：另外一个缓存条件参数，非必需，需使用SpEL表达式。它不同于conditio
断时机，该条件是在函数被调用之后才做判断的，所以它可以通过对result进行判断。
 *      keyGenerator：用于指定key生成器，非必需。若需要指定一个自定义的key生成器
org.springframework.cache.interceptor.KeyGenerator接口，并使用该参数来指定。需
是互斥的
```

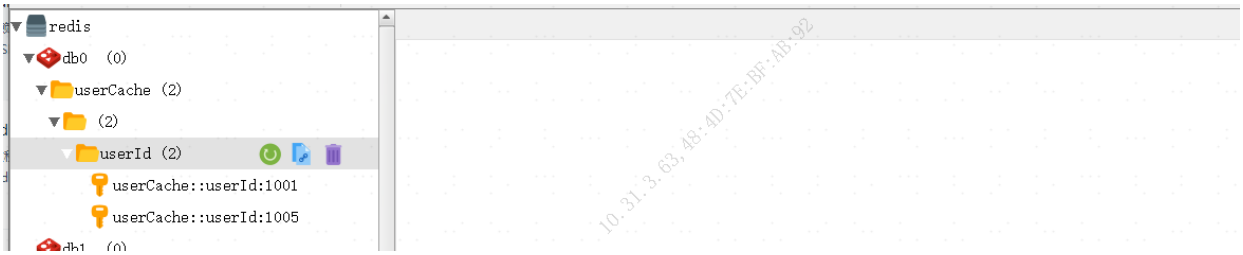
```
 *       cacheManager：用于指定使用哪个缓存管理器，非必需。只有当有多个时才需要使用
 *       cacheResolver：用于指定使用那个缓存解析器，非必需。需通过org.springframework.cache.interceptor.CacheResolver接口来实现自己的缓存解析器
 * @CachePut：不影响方法执行更新缓存
 * @CacheEvict：从缓存删除
 * @Caching：组合多个Cache注解使用
 */
```

@Cacheable

```
//key中的：为目录分割符
//同步阻塞
@Cacheable(key = "'userId:' + #id", sync = true)
public User getUserById(Integer id) {
    log.info("getUserById -> Cache no data, Query DB & Add Cache");
    //返回值进缓存
    return new User(id, "查询不到时，加入的缓存");
}

@Test
public void getUserCache() throws Exception{
    User user1 = userService.getUserById(1001);
    User user2 = userService.getUserById(1001);
    log.info(JSON.toJSONString("User1 Query Cache -> " + user1));
    log.info(JSON.toJSONString("User2 Query Cache -> " + user2));
    User user3 = userService.getUserById(1005);
}
```

```
2022-04-27 14:53:25.855  INFO 11924 --- [           main] c.f.mall.redis.service.UserService       : getUserById -> Cache no data, Query DB & Add Cache
2022-04-27 14:53:26.079  INFO 11924 --- [           main] com.formssi.mall.redis.SpringCacheTests  : "User1 Query Cache -> User(id=1001, name=查询不到时，加入的缓存)"
2022-04-27 14:53:26.079  INFO 11924 --- [           main] com.formssi.mall.redis.SpringCacheTests  : "User2 Query Cache -> User(id=1001, name=查询不到时，加入的缓存)"
2022-04-27 14:53:26.081  INFO 11924 --- [           main] c.f.mall.redis.service.UserService       : getUserById -> Cache no data, Query DB & Add Cache
```

user2查的user1的缓存

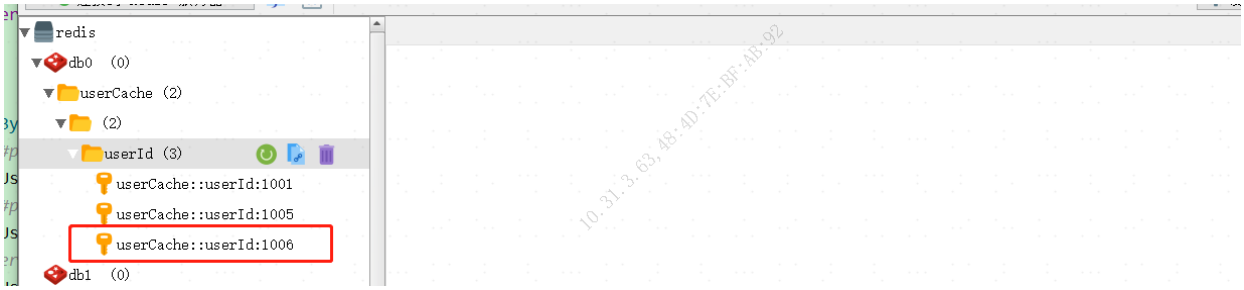



```
//插入符合条件（1005 < id <= 1010）的，不符合的不插入cache，但会执行方法体
//已经存在cache则不执行方法体
@Cacheable(key = "'userId:' + #user.id", condition = "#p0.id > 1005", unless
```

```java
public User addUserByCondition(User user) {
    log.info("addUserByCondition -> Insert DB");
    return user;
}

@Test
public void addUserByCondition() throws Exception{
    //condition = "#p0.id > 1005    不符合
    userService.addUserByCondition(new User(1004, "李四"));
    //condition = "#p0.id > 1005   符合
    userService.addUserByCondition(new User(1006, "赵六"));
    //unless = "#user.id > 1010"   不符合
    userService.addUserByCondition(new User(1011, "王十一"));
}
```

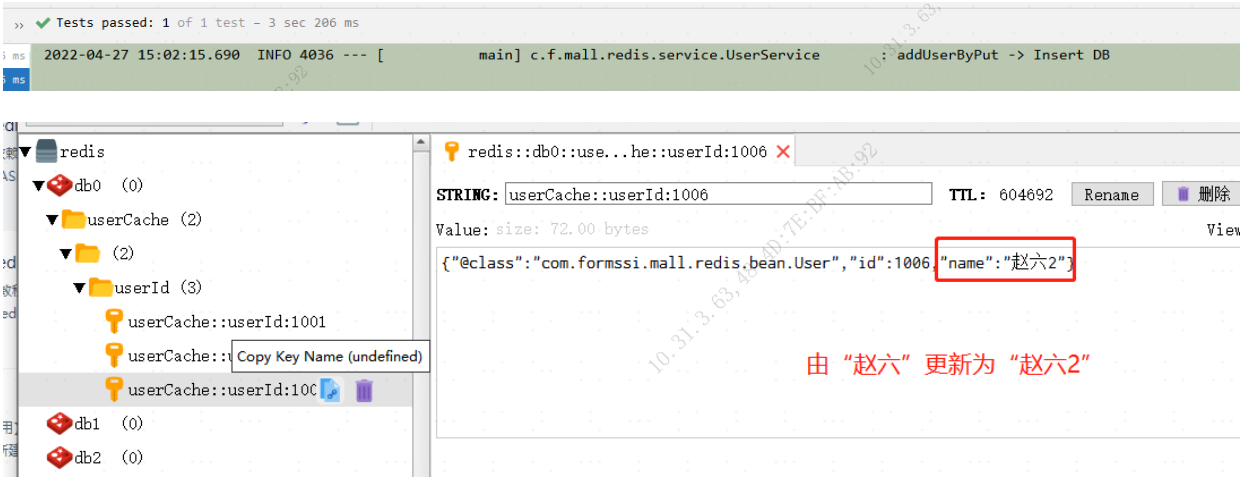
```
2022-04-27 14:58:20.749  INFO 8496 --- [          main] c.f.mall.redis.service.UserService       : addUserByCondition -> Insert DB
2022-04-27 14:58:21.914  INFO 8496 --- [          main] c.f.mall.redis.service.UserService       : addUserByCondition -> Insert DB
2022-04-27 14:58:22.028  INFO 8496 --- [          main] c.f.mall.redis.service.UserService       : addUserByCondition -> Insert DB
```

都执行了方法体，但只有1006符合条件，加入Cache



@CachePut

```java
//存在cache也会执行方法体，然后更新Cache
//#result返回对象，#result.name
@CachePut(key = "'userId:' + #user.id")
public User addUserByPut(User user) {
    log.info("addUserByPut -> Insert DB");
    return user;
}

@Test
public void addUserByPut() throws Exception{
    //已经存在的数据
    userService.addUserByPut(new User(1006, "赵六2"));
}
```



@CacheEvict

```java
//userCache下全被删除，删除db也执行了
//无cache，只执行方法体
@CacheEvict(key = "'userId:' + #user.id", beforeInvocation = true , allEntr
public void delUserById(User user) {
    log.info("delUserById -> Delete DB");
}

@Test
public void delUserById() throws Exception{
    //beforeInvocation =true 先删除cache，再删db, allEntries = true 删除所有c
    userService.delUserById(new User(1001, null));
}
```

@Caching

```java
//批量执行多个@CacheXXX
@Caching(
    cacheable = {@Cacheable(key = "'userId:' + #user.id")}
    , put = {@CachePut(key = "'userName:' + #a0.name")}
```

```
    , evict = {@CacheEvict(key = "'userId:' + #user.id")}
)
public User addUserByCaches(User user) {
    log.info("addUserByCaches -> Insert DB");
    return user;
}

@Test
public void addUserByCaches() throws Exception{
    //已经存在的数据
    userService.addUserByCaches(new User(1007, "孙八"));
}
```

```
2022-04-27 15:09:37.659  INFO 4004 --- [           main] c.f.mall.redis.service.UserService       : addUserByCaches -> Insert DB
```