链滴

redis【持久化 & 事务 & 锁】

作者：haxLook
原文链接：https://ld246.com/article/1617263227203
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

Redis容器配置redis.conf

- redis容器里边的配置文件是需要在 **创建容器时映射**进来的

停止容器：docker container stop myredis
删除容器：docker container rm myredis

- 重新开始创建容器【直接在外部配置软连接到容器内部】

1. 创建docker统一的外部配置文件

mkdir -p docker/redis/{conf,data}

2. 在conf目录创建redis.conf的配置文件

touch /docker/redis/conf/redis.conf

3. redis.conf文件的内容需要自行去下载，网上很多

4. 创建启动容器，加载配置文件并持久化数据

docker run -d --privileged=true -p 6379:6379 -v /docker/redis/conf/redis.conf:/etc/redis/redis conf -v /docker/redis/data:/data --name myredis redis redis-server /etc/redis/redis.conf --ap endonly yes

1、简介

什么是持久化?

利用**永久性**存储介质将数据进行保存，在特定的时间将保存的数据进行恢复的工作机制称为持久化。

为什么要持久化

防止数据的意外**丢失**，确保数据**安全性**

持久化过程保存什么

- 将当前 **数据状态**进行保存，**快照**形式，存储数据结果，存储格式简单，关注点在**数据**
- 将数据的 **操作过程**进行保存，**日志**形式，存储操作过程，存储格式复杂，关注点在数据的操作**过程**

原文链接：redis【持久化 & 事务 & 锁】

RDB

RDB启动方式——save

- 命令

save

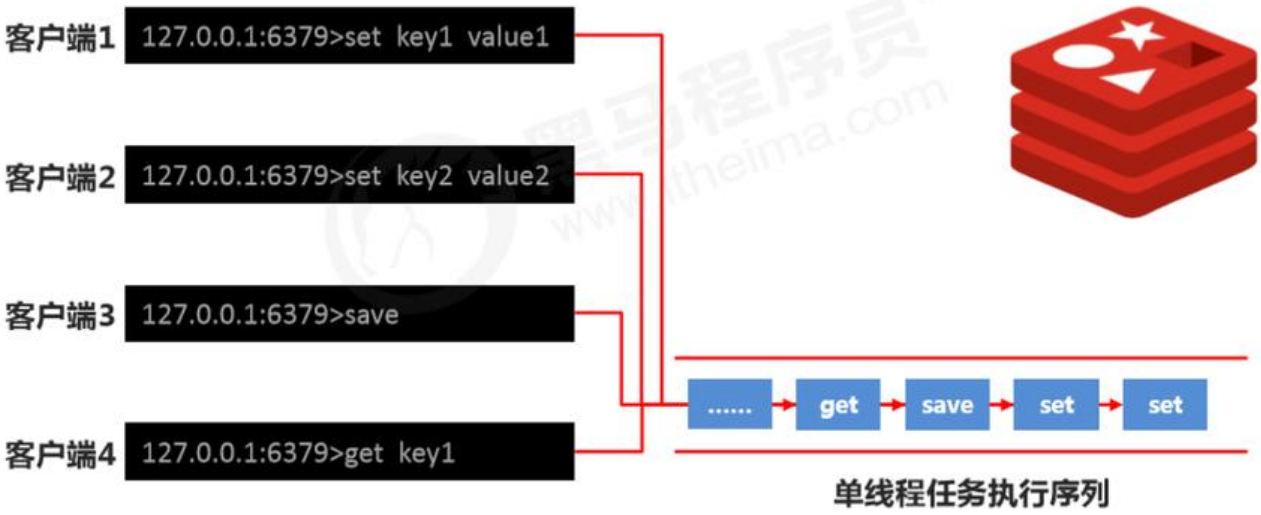
- 作用

手动执行一次保存操作

RDB配置相关命令

- dbfilename dump.rdb
  - 说明：设置本地数据库文件名，默认值为 dump.rdb
  - 经验：通常设置为dump-端口号.rdb
- dir
  - 说明：设置存储.rdb文件的路径
  - 经验：通常设置成存储空间较大的目录中，目录名称data
- rdbcompression yes
  - 说明：设置存储至本地数据库时是否压缩数据，默认为 yes，采用 LZF 压缩
  - 经验：通常默认为开启状态，如果设置为no，可以节省 CPU 运行时间，但会使存储的文件变大巨大)
- rdbchecksum yes
  - 说明：设置是否进行RDB文件格式校验，该校验过程在写文件和读文件过程均进行
  - 经验：通常默认为开启状态，如果设置为no，可以节约读写性过程约10%时间消耗，但是存储定的数据损坏风险

RDB启动方式——save指令工作原理



注意：**save指令**的执行会**阻塞**当前Redis服务器，直到当前RDB过程完成为止，有可能会造成**长时间阻塞**，线上环境**不建议使用。**

RDB启动方式——bgsave

- 命令

bgsave

- 作用

手动启动后台保存操作，但**不是立即执行**

RDB启动方式 —— bgsave指令工作原理

注意： **bgsave命令**是针对save阻塞问题做的**优化。**Redis内部所有涉及到RDB操作都采用bgsave的式，save命令可以放弃使用，推荐使用bgsave

bgsave的保存操作可以通过redis的日志查看

docker logs myredis

RDB启动方式 ——save配置

- 配置

save second changes

- 作用

满足**限定时间**范围内key的变化数量达到**指定数量**即进行持久化

- 参数

second：监控时间范围

changes：监控key的变化量

- 配置位置

在**conf文件**中进行配置

RDB启动方式 ——save配置原理

注意:

- save配置要根据实际业务情况进行设置，频度过高或过低都会出现性能问题，结果可能是灾难性的
- save配置中对于second与changes设置通常具有 **互补对应**关系（一个大一个小），尽量不要设成包含性关系
- save配置启动后执行的是 **bgsave操作**

RDB启动方式对比

方式	save指令	bgsave指令
读写	同步	异步
阻塞客户端指令	是	否
额外内存消耗	否	是
启动新进程	否	是

原文链接：redis【持久化 & 事务 & 锁】

RDB优缺点

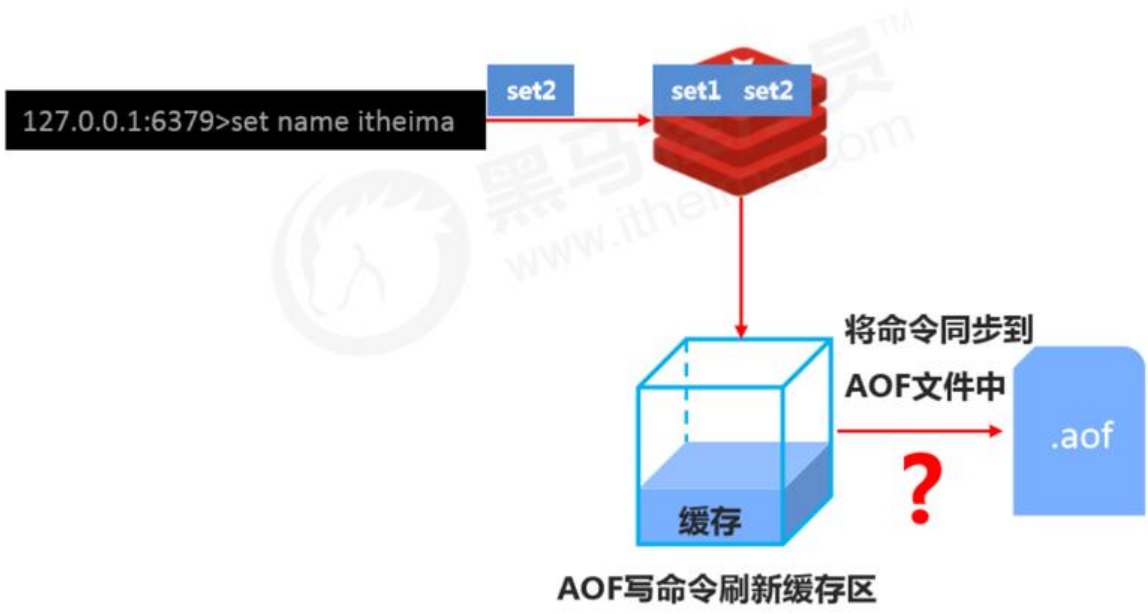
- 优点
  - RDB是一个紧凑压缩的二进制文件， **存储效率较高**
  - RDB内部存储的是redis在某个时间点的数据快照，非常适合用于 **数据备份，全量复制**等场景
  - RDB恢复数据的 **速度**要比AOF**快**很多
  - 应用：服务器中每X小时执行bgsave备份，并将RDB文件拷贝到远程机器中， **用于灾难恢复**
- 缺点
  - RDB方式无论是执行指令还是利用配置， **无法做到实时持久化**，具有较大的可能性丢失数据
  - bgsave指令每次运行要执行fork操作 **创建子进程**，要**牺牲**掉一些**性能**
  - Redis的众多版本中未进行RDB文件格式的版本统一，有可能出现各版本服务之间数据格式 **无兼容**现象

AOF

AOF概念

- AOF(append only file)持久化：以独立日志的方式记录 **每次**写命令，重启时再重新执行AOF文中命令，以达到恢复数据的目的。与RDB相比可以简单描述为改记录数据为记录数据产生的过程
- AOF的主要作用是解决了数据持久化的实时性，目前已经是Redis持久化的 **主流**方式

AOF写数据过程



AOF写命令刷新缓存区

AOF写数据三种策略(appendfsync)

- always

- 每次写入操作均同步到AOF文件中，数据零误差， **性能较低,不建议使用**
- everysec
  - 每秒将缓冲区中的指令同步到AOF文件中，数据准确性较高， **性能较高** ， **建议使用**，也是默配置
  - 在系统突然宕机的情况下丢失1秒内的数据
- no
  - 由操作系统控制每次同步到AOF文件的周期，整体过程 **不可控**

AOF功能开启

- 配置

appendonly yes|no

作用

是否开启AOF持久化功能，

默认为不开启状

- 配置

appendfsync always|everysec|no

- 作用
  - AOF写数据策略

AOF重写

规则

- 进程内已超时的数据不再写入文件
- 忽略 **无效指令**，重写时使用进程内数据直接生成，这样新的AOF文件**只保留最终数据的写入命令**
  - 如del key1、 hdel key2、srem key3、set key4 111、set key4 222等
- 对同一数据的多条写命令合并为一条命令
  - 如lpush list1 a、lpush list1 b、 lpush list1 c 可以转化为：lpush list1 a b c
  - 为防止数据量过大造成客户端缓冲区溢出，对list、set、hash、zset等类型，每条指令最多写入4个元素

如何使用

- 手动重写

bgrewriteaofCopy

- 自动重写

auto-aof-rewrite-min-size size
auto-aof-rewrite-percentage percentage

工作原理

AOF自动重写

- 自动重写触发条件设置

//触发重写的最小大小
auto-aof-rewrite-min-size size
//触发重写须达到的最小百分比
auto-aof-rewrite-percentage percentCopy

- 自动重写触发比对参数（运行指令info Persistence获取具体信息）

//当前.aof的文件大小
aof_current_size
//基础文件大小
aof_base_size

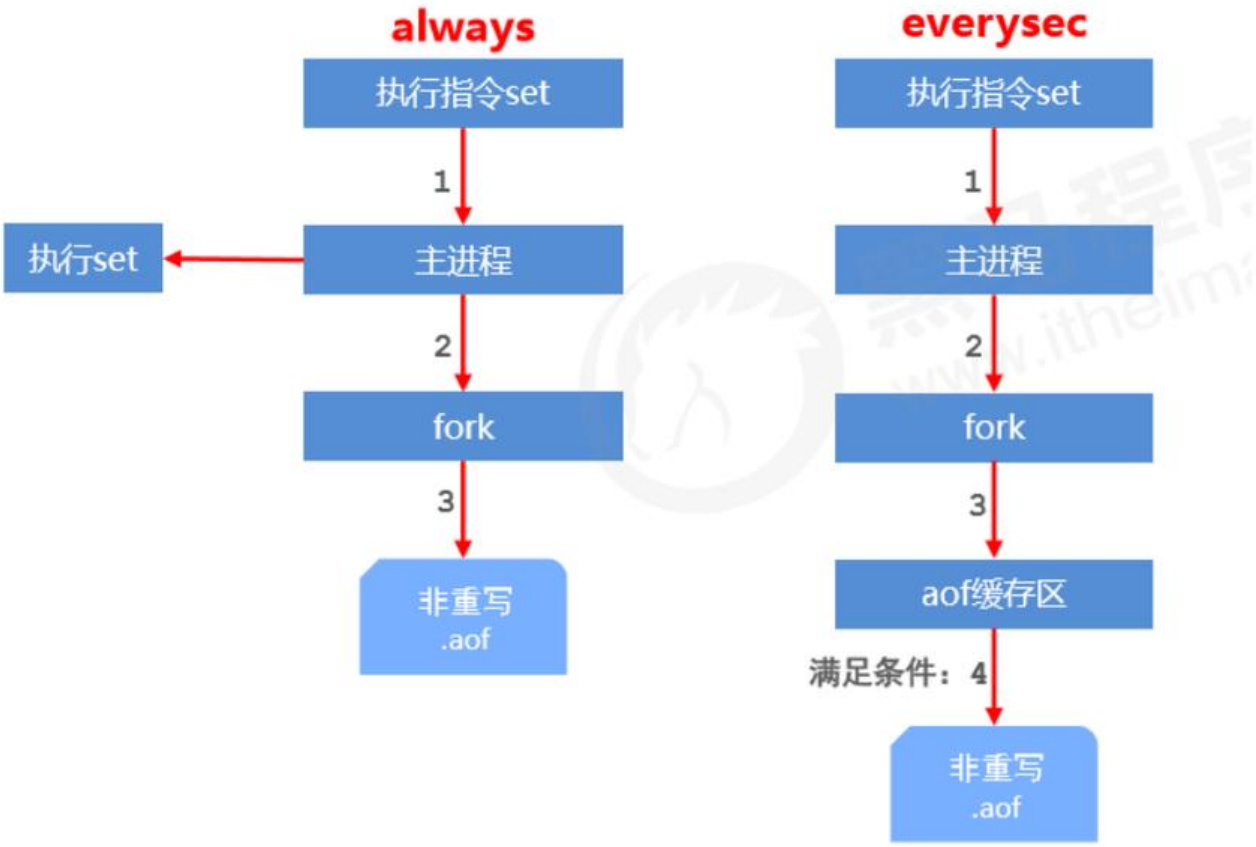
自动重写触发条件

$$aof\_current\_size > auto\text{-}aof\text{-}rewrite\text{-}min\text{-}size$$

$$\frac{aof\_current\_size - aof\_base\_size}{aof\_base\_size} >= auto\text{-}aof\text{-}rewrite\text{-}percentage$$

工作原理

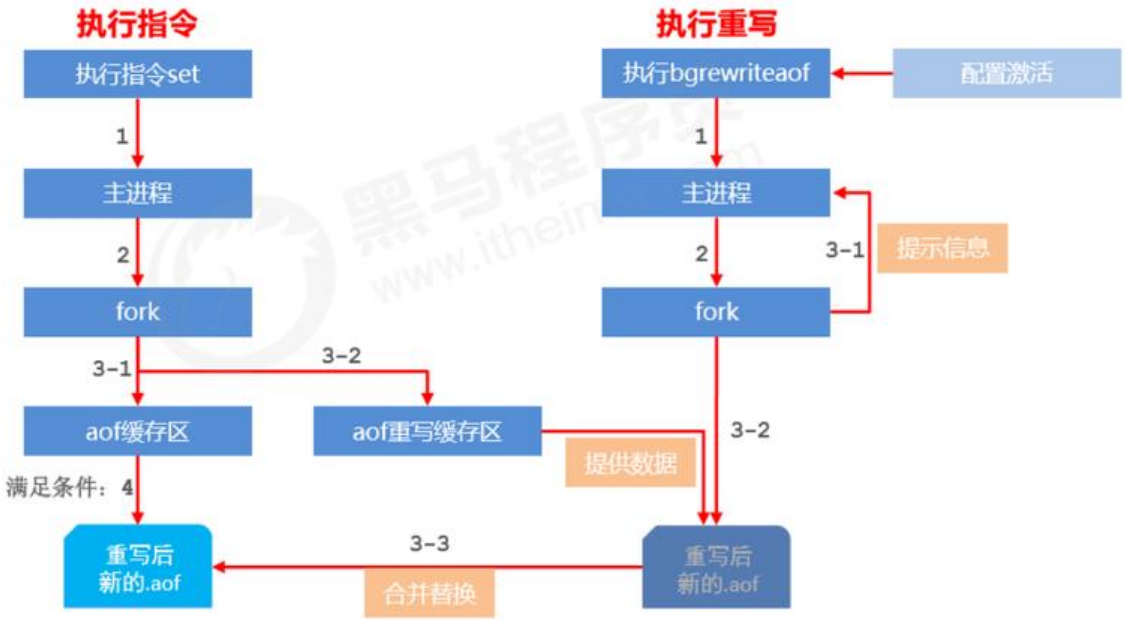
AOF工作流程



AOF重写流程

原文链接：redis【持久化 & 事务 & 锁】

AOF重写流程



缓冲策略

AOF缓冲区同步文件策略，由参数**appendfsync**控制

● write操作会触发延迟写（delayed write）机制，Linux在内核提供页缓冲区用 来提高硬盘IO性能 write操作在写入系统缓冲区后直接返回。同步硬盘操作依 赖于系统调度机制，列如：缓冲区页空间满或达到特定时间周期。同步文件之 前，如果此时系统故障宕机，缓冲区内数据将丢失。

● fsync针对单个文件操作（比如AOF文件），做强制硬盘同步，fsync将阻塞知道 写入硬盘完成后返，保证了数据持久化

4、RDB VS AOF

持久化方式	RDB	AOF
占用存储空间	小（数据级：压缩）	大（指令级：重写）
存储速度	慢	快
恢复速度	快	慢
数据安全性	会丢失数据	依据策略决定
资源消耗	高/重量级	低/轻量级
启动优先级	低	高

原文链接：redis【持久化 & 事务 & 锁】

RDB与AOF的选择之惑

- 对数据非常 **敏感**，建议使用默认的**AOF**持久化方案
  - AOF持久化策略使用 **everysecond**，每秒钟fsync一次。该策略redis仍可以保持很好的处理能，当出现问题时，最多丢失0-1秒内的数据。
  - 注意：由于AOF文件 **存储体积较大**，且**恢复速度较慢**
- 数据呈现 **阶段有效性**，建议使用RDB持久化方案
  - 数据可以良好的做到阶段内无丢失（该阶段是开发者或运维人员手工维护的），且 **恢复速度较快** 阶段 点数据恢复通常采用RDB方案
  - 注意：利用RDB实现紧凑的数据持久化会使Redis降的很低
- 综合比对
  - RDB与AOF的选择实际上是在做一种权衡，每种都有利有弊
  - 如不能承受数分钟以内的数据丢失，对业务数据非常 **敏感**，选用**AOF**
  - 如能承受数分钟以内的数据丢失，且追求大数据集的 **恢复速度**，选用**RDB**
  - **灾难恢复选用RDB**
  - 双保险策略，同时开启 RDB 和 AOF，重启后，Redis优先使用 AOF 来恢复数据，降低丢失数据

Redis事务

Redis事务的定义

redis事务就是一个命令执行的队列，将一系列预定义命令**包装成一个整体**（一个队列）。当执行时，**次性按照添加顺序依次执行**，中间不会被打断或者干扰

事务的基本操作

- 开启事务

multiCopy

  - 作用
    - 作设定事务的开启位置，此指令执行后，后续的所有指令均加入到事务中
- 取消事务

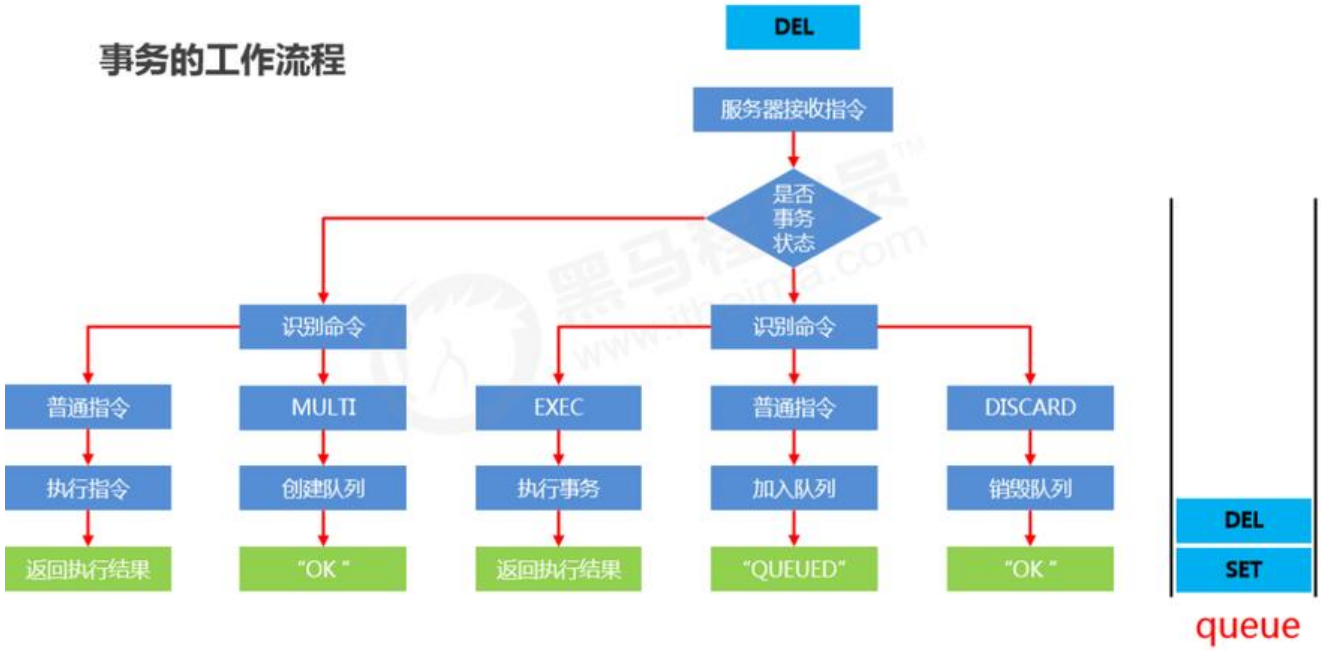
discardCopy

  - 作用
    - 终止当前事务的定义，发生在multi之后，exec之前
- 执行事务

execCopy

- 作用
    - 设定事务的结束位置，同时执行事务。 **与multi成对出现**，成对使用

3、事务操作的基本流程



4、事务操作的注意事项

定义事务的过程中，命令格式输入错误怎么办?

- 语法错误
    - 指命令书写格式有误 例如执行了一条不存在的指令
- 处理结果
    - 如果定义的事务中所包含的命令存在语法错误，整体事务中 **所有命令均不会执行。**包括那些语正确的命令

定义事务的过程中，命令执行出现错误怎么办?

- 运行错误
    - 指命令 **格式正确**，但是**无法正确的执行。**例如对list进行incr操作
- 处理结果
    - 能够正确运行的命令会执行，运行错误的命令不会被执行

注意：已经执行完毕的命令对应的数据**不会自动回滚**，需要程序员自己在代码中实现回滚。

5、基于特定条件的事务执行

原文链接：redis【持久化 & 事务 & 锁】

锁

- 对 key 添加监视锁，在执行exec前如果key发生了变化，终止事务执行

```
watch key1, key2....Copy
```

- 取消对 **所有**key的监视

```
unwatchCopy
```

分布式锁

- 使用 setnx 设置一个公共锁

```
//上锁
setnx lock-key value
//释放锁
del lock-keyCopy
```

- 利用setnx命令的返回值特征，有值（被上锁了）则返回设置失败，无值（没被上锁）则返回设成功
- 操作完毕通过del操作释放锁

注意：上述解决方案是一种**设计概念**，依赖规范保障，具有风险性

分布式锁加强

- 使用 expire 为锁key添加 **时间限定**，到时不释放，放弃锁

```
expire lock-key seconds
pexpire lock-key millisecondsCopy
```

- 由于操作通常都是微秒或毫秒级，因此该锁定时间 **不宜设置过大**。具体时间需要业务测试后确认。
  - 例如：持有锁的操作最长执行时间127ms，最短执行时间7ms。
  - 测试百万次最长执行时间对应命令的最大耗时，测试百万次网络延迟平均耗时
  - 锁时间设定推荐：最大耗时 120%+平均网络延迟110%
  - 如果业务最大耗时<<网络平均延迟，通常为2个数量级，取其中单个耗时较长即可

栗子：

```
/**
 * @author hax redis锁
 * Created by Administrator on 2020/9/4.
 */
@Component
public class RedisLockHandler {

    private static final Logger LOGGER = LoggerFactory.getLogger(RedisLockHandler.class);
```

```
    private static final int DEFAULT_SINGLE_EXPIRE_TIME = 3;

    @Autowired
    JedisClientPools jedisClientPool;

    /**
     * 获取锁  如果锁可用   立即返回true，  否则返回false
     *
     * @param billIdentify
     * @return
     */
    public boolean tryLock(TSuperclass billIdentify) {

        TimeUnit timeUnit = TimeUnit.SECONDS;
        //设置30秒的时间进行过滤操作
        return tryLock(billIdentify, 20, timeUnit);
    }

    public void lock(TSuperclass billIdentify) {
        this.voidLock(billIdentify);
    }


    /**
     * 锁在给定的等待时间内空闲，则获取锁成功 返回true，  否则返回false
     *
     * @param billIdentify
     * @param timeout
     * @param unit
     * @return
     */
    public boolean tryLock(TSuperclass billIdentify, long timeout, TimeUnit unit) {
        String $_lockKey = (String) billIdentify.getTSuperclassKey();
        try {
            String $_lockValue = StringUtils.uuid();
            long nano = System.nanoTime();
            do {
                LOGGER.info("【获取/try】lock key: " + $_lockKey);
                Long i = jedisClientPool.setnx($_lockKey, $_lockValue);
                if (i == 1) {
                    jedisClientPool.expire($_lockKey, DEFAULT_SINGLE_EXPIRE_TIME);
                    LOGGER.info("【设置/get】 lock, key: " + $_lockKey + " , expire in " + DEFAULT_S
NGLE_EXPIRE_TIME + " seconds.");
                    return Boolean.TRUE;
                } else { // 存在锁
                    if (LOGGER.isDebugEnabled()) {
                        String desc = jedisClientPool.get($_lockKey);
                        LOGGER.info("【已存在/aleary】key: " + $_lockKey + " locked by another busin
ss：" + desc);
                    }
                }
                if (timeout == 0) {
                    break;
                }
```

```
            Thread.sleep(300);
        } while ((System.nanoTime() - nano) < unit.toNanos(timeout));
        return Boolean.FALSE;
    } catch (JedisConnectionException je) {
        LOGGER.error(je.getMessage(), je);
        returnBrokenResource(jedisClientPool.getJedis());
    } catch (Exception e) {
        LOGGER.error(e.getMessage(), e);
    } finally {
        returnResource(jedisClientPool.getJedis());
    }
    return Boolean.FALSE;
}

/**
 * 如果锁空闲立即返回  获取失败 一直等待
 *
 * @param billIdentify
 */
public void voidLock(TSuperclass billIdentify) {
    String key = (String) billIdentify.getTSuperclassKey();
    try {
        do {
            LOGGER.info("lock key: " + key);
            Long i = jedisClientPool.setnx(key, key);
            if (i == 1) {
                jedisClientPool.expire(key, DEFAULT_SINGLE_EXPIRE_TIME);
                LOGGER.info("get lock, key: " + key + " , expire in " + DEFAULT_SINGLE_EXPIRE_T
ME + " seconds.");
                return;
            } else {
                if (LOGGER.isDebugEnabled()) {
                    String desc = jedisClientPool.get(key);
                    LOGGER.info("key: " + key + " locked by another business：" + desc);
                }
            }
            Thread.sleep(300);
        } while (true);
    } catch (JedisConnectionException je) {
        LOGGER.error(je.getMessage(), je);
        returnBrokenResource(jedisClientPool.getJedis());
    } catch (Exception e) {
        LOGGER.error(e.getMessage(), e);
    } finally {
        returnResource(jedisClientPool.getJedis());
    }
}

/**
 * 释放锁
 *
 * @param billIdentify
 */
public void unLock(TSuperclass billIdentify) {
```

```
    List<TSuperclass> list = new ArrayList<TSuperclass>();
    list.add(billIdentify);
    unLock(list);
}

/**
 * 获取所有的锁数据
 *
 * @param ids
 * @return
 */
public List<TSuperclass> queryLocks(List<String> ids) {
    List<TSuperclass> list = new ArrayList<>();
    ids.forEach(id -> {
        list.add(TSuperclass.getVoucher(id));
    });
    return list;
}

/**
 * 一键释放锁
 *
 * @param ids
 * @return
 */
public void unLocks(List<String> ids) {
    List<TSuperclass> list = this.queryLocks(ids);
    unLock(list);
}

/**
 * 批量释放锁
 *
 * @param billIdentifyList
 */
public void unLock(List<TSuperclass> billIdentifyList) {
    List<String> keys = new CopyOnWriteArrayList<String>();
    for (TSuperclass identify : billIdentifyList) {
        String key = (String) identify.getTSuperclassKey();
        keys.add(key);
    }
    try {
        jedisClientPool.delbath(keys.toArray(new String[0]));
        LOGGER.info("【删除/delete】lock, keys :" + keys);
    } catch (JedisConnectionException je) {
        LOGGER.error(je.getMessage(), je);
        returnBrokenResource(jedisClientPool.getJedis());
    } catch (Exception e) {
        LOGGER.error(e.getMessage(), e);
    } finally {
        returnResource(jedisClientPool.getJedis());
    }
}
```

```
    /**
     * 销毁连接
     *
     * @param jedis
     */
    private void returnBrokenResource(Jedis jedis) {
        if (jedis == null) {
            return;
        }
        try {
            //容错
            jedisClientPool.getJedisPool().returnBrokenResource(jedis);
        } catch (Exception e) {
            LOGGER.error(e.getMessage(), e);
        }
    }

    /**
     * @param jedis
     */
    private void returnResource(Jedis jedis) {
        if (jedis == null) {
            return;
        }
        try {
            jedisClientPool.getJedisPool().returnResource(jedis);
        } catch (Exception e) {
            LOGGER.error(e.getMessage(), e);
        }
    }

}
```