
回到首页

芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2019-10-03
Redis

精尽 Redisson 源码分析 —— 限流器 RateLimiter

1. 概述

在开始本文之前，先推荐看一篇干货 《你应该如何正确健壮后端服务？》 。

限流，无论在系统层面，还是在业务层面，使用都非常广泛。例如说：

> 【业务】为了避免恶意的灌水机或者用户，限制每分钟至允许回复 10 个帖子。
> 【系统】为了避免服务系统被大规模调用，超过极限，限制每个调用方只允许每秒调用 100 次。

限流算法，常用的分成四种：

> 每一种的概念，推荐看看 《计数器、滑动窗口、漏桶、令牌算法比较和伪代码实现》 文章。
>
> 计数器
>
> > 比较简单，每固定单位一个计数器即可实现。
>
> 滑动窗口
>
> > Redisson 提供的是基于滑动窗口 RateLimiter 的实现。相比计数器的实现，它的起点不是固定的，而是以开始计数的那个时刻开始为一个窗口。
> >
> > 所以，我们可以把计数器理解成一个滑动窗口的特例，以固定单位为一个窗口。
>
> 令牌桶算法
>
> > 《Eureka 源码解析 —— 基于令牌桶算法的 RateLimiter》 ，单机并发场景下的 RateLimiter 实现。
> >
> > 《Spring-Cloud-Gateway 源码解析 —— 过滤器 (4.10) 之 RequestRateLimiterGatewayFilterFactory 请求限流》 ，基于 Redis 实现的令牌桶算法的 RateLimiter 实现。

漏桶算法

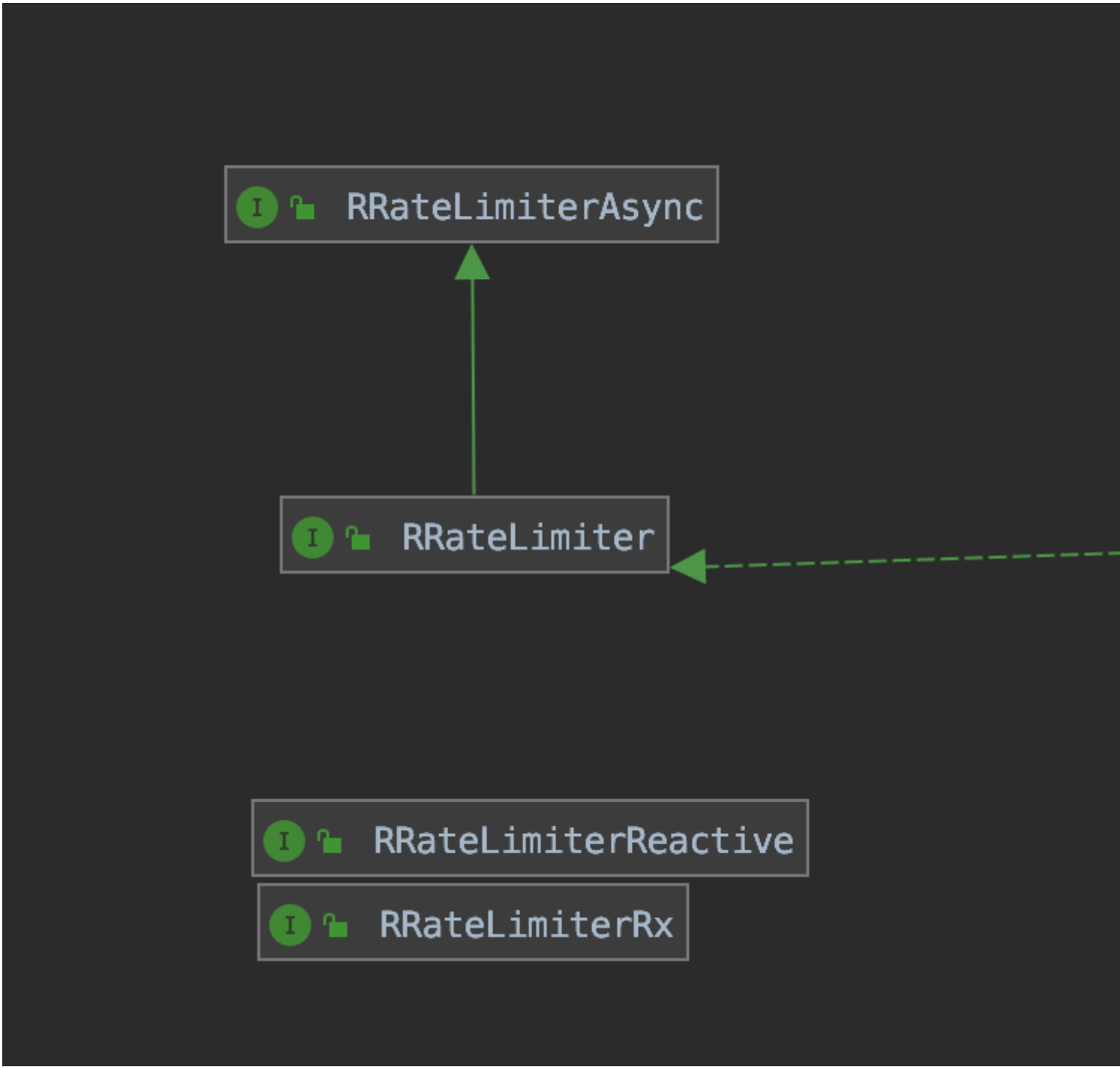
漏桶算法，一直没搞明白和令牌桶算法的区别。现在的理解是：

- 令牌桶算法，桶里装的是令牌。每次能拿取到令牌，就可以进行访问。并且，令牌会按照速率不断恢复放到令牌桶中直到桶满。
- 漏桶算法，桶里装的是请求。当桶满了，请求就进不来。例如说，Hystrix 使用线程池或者 Semaphore 信号量，只有在请求未满的时候，才可以进行执行。

上面哔哔了非常多的字，只看本文的话，就那一句话：“Redisson 提供的是基于滑动窗口 RateLimiter 的实现。”。

2. 整体一览

在 Redisson 中，提供了四个 RateLimiter 相关的接口，如下图：



org.redisson.api.RRateLimiterAsync ，定义了异步操作的接口。
org.redisson.api.RRateLimiter ，继承 RRateLimiterAsync 的基础上，定义了同步操作的接口。
org.redisson.api.RRateLimiterReactive ，定义基于 Reactor 操作的接口。
org.redisson.api.RRateLimiterReactive ，定义基于 RxJava 操作的接口。

目前，Redisson 暂时只实现了 RRateLimiterAsync 和 RRateLimiter 接口的方法，即 org.redisson.RedissonRateLimiter 。

RRateLimiterAsync 和 RRateLimiter 定义的接口，差别就在于同步和异步，所以我们就只看看 RRateLimiter 接口。代码如下：

```
boolean trySetRate(RateType mode, long rate, long rateInterval, RateIntervalUnit rateIntervalUnit);
RateLimiterConfig getConfig();

boolean tryAcquire();
boolean tryAcquire(long permits);
boolean tryAcquire(long timeout, TimeUnit unit);
boolean tryAcquire(long permits, long timeout, TimeUnit unit);

void acquire();
void acquire(long permits);
```

#trySetRate(RateType mode, long rate, long rateInterval, RateIntervalUnit rateIntervalUnit) 方法，设置限流器的配置。
#getConfig() 方法，获得限流器的配置。
#tryAcquire(...) 方法，尝试在指定时间内，获得指定数量的令牌，并返回是否成功。
#acquire(...) 方法，在指定时间内，获得指定数量的令牌，直到成功。

总的来说，一共两类方法，一类是设置或获取配置，一类是获取令牌。下面，我们来逐个方法的源码，来瞅瞅。

3. trySetRate

在 《精尽 Redisson 源码分析 —— 调试环境搭建》 中，我们搭建了一个限流器的示例。在示例的开始，我们会调用 RRateLimiter#trySetRateAsync(RateType type, long rate, long rateInterval, RateIntervalUnit unit) 方法，设置限流器的配置。代码如下：

```
// RedissonRateLimiter.java

@Override
public boolean trySetRate(RateType type, long rate, long rateInterval, RateIntervalUnit unit) {
    return get(trySetRateAsync(type, rate, rateInterval, unit));
}

@Override
public RFuture<Boolean> trySetRateAsync(RateType type, long rate, long rateInterval, RateIntervalUnit unit) {
    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, RedisCommands.EVAL_BOOLEAN,
            "redis.call('hsetnx', KEYS[1], 'rate', ARGV[1]);"
          + "redis.call('hsetnx', KEYS[1], 'interval', ARGV[2]);"
          + "return redis.call('hsetnx', KEYS[1], 'type', ARGV[3]);",
            Collections.<Object>singletonList(getName()), // keys [分布锁名]
            rate, unit.toMillis(rateInterval), type.ordinal()); // values [速度、速度单位、限流类型]
}
```

将限流器的配置写入到 Redis 中。这个和我们看到的很多分布式限流器的实现不同，它们只提供获取令牌的功能，而 Redisson 多了持久化配置限流器的配置到 Redis 中，相当于说，Redis 起到了一个配置中心的功能，分布式下的相同限流器（“相同”指的是相同名字的限流器）使用同一的配置。

参数 type ：类型是 org.redisson.api.RateType ，限流类型。目前有两种：

- OVERALL ：相同名字的所有 RateLimiter 实例。
- PER_CLIENT ：相同 JVM 进程的所有相同名字的所有 RateLimiter 实例。 有点莫名，为啥还会有 PER_CLIENT 级别的。目前能够想象的，一个是 Redis 可以统一配置，一个是 Redis 上可以看到限流的情况。

参数 rate + rateInterval + unit ：形成指定时间内，允许执行的频率。

整个的逻辑实现，通过 Redis Lua 脚本，保证整个配置的存储设置的原子性。并且，通过 HSETNX 我们可以看到两点：

配置的存储，使用 Redis Hash 数据结构。如下是个示例：

```
127.0.0.1:6379> HGETALL myRateLimiter
1) "rate" # 速度
2) "50"
3) "interval" # 频率
4) "60000"
5) "type" # 限流类型
6) "0"
```

使用 HSETNX 指令，如果 Redis 该 KEY 已经被使用（例如说，已经初始化过限流器的配置），则不进行覆盖。这点，一定要注意哈。

整段代码比较简单，可能不了解 Redisson 源码的胖友，会对此处的 #get(RFuture<V> future) 方法，有一点点疑惑。

```
// RedissonObject.java

protected final CommandAsyncExecutor commandExecutor;

protected final <V> V get(RFuture<V> future) {
    return commandExecutor.get(future);
}
```

RedissonRateLimiter 继承了 RedissonObject 类。

future 参数，是 org.redisson.api.RFuture<V> 接口类型，是 Redisson 对 Future 功能的增强，支持回调等功能。 下面，我们就会看到 RedissonRateLimiter 会使用到回调的功能。

通过调用 #get(RFuture<V> future) 方法，将 #trySetRateAsync(RateType type, long rate, long rateInterval, RateIntervalUnit unit) 方法的 RFuture 结果，同步返回。

另外，在 #trySetRateAsync(RateType type, long rate, long rateInterval, RateIntervalUnit unit) 方法，会调用 #getName() 方法，获得限流器的名字。代码如下：

```
// RedissonObject.java

private String name;

@Override
```

```
public String getName() {
    return name;
}
```

这个属性，就是我们在 `RedissonClient#getRateLimiter(String name)` 方法，创建 RedissonRateLimiter 对象时所设置的。同时，`name` 也就成了我们在 Redis 所看到的分布式限流器的名字。

4. getConfig

`#getConfig()` 方法，从 Redis 中加载 org.redisson.apiRateLimiterConfig 限流器配置。代码如下：

```
// RateLimiterConfig.java

private static final RedisCommand HGETALL = new RedisCommand("HGETALL", new MultiDecoder<RateLimiterConfig>() {
    @Override
    public Decoder<Object> getDecoder(int paramNum, State state) {
        return null;
    }

    @Override
    public RateLimiterConfig decode(List<Object> parts, State state) {
        Map<String, String> map = new HashMap<>(parts.size()/2);
        for (int i = 0; i < parts.size(); i++) {
            if (i % 2 != 0) {
                map.put(parts.get(i-1).toString(), parts.get(i).toString());
            }
        }

        // 创建 RateType 对象
        RateType type = RateType.values()[Integer.valueOf(map.get("type"))];
        Long rateInterval = Long.valueOf(map.get("interval"));
        Long rate = Long.valueOf(map.get("rate"));
        return new RateLimiterConfig(type, rateInterval, rate);
    }

}, ValueType.MAP);

@Override
public RateLimiterConfig getConfig() {
    return get(getConfigAsync());
}

@Override
public RFuture<RateLimiterConfig> getConfigAsync() {
    return commandExecutor.readAsync(getName(), StringCodec.INSTANCE, HGETALL, getName());
}
```

5. tryAcquire

在看 `#tryAcquire(...)` 方法之前，我们先来看它是如何调用 Redis 实现限流算法的。代码如下：

```
// RedissonRateLimiter.java

 1: private <T> RFuture<T> tryAcquireAsync(RedisCommand<T> command, Long value) {
 2:     return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, command,
 3:             "local rate = redis.call('hget', KEYS[1], 'rate');"
 4:           + "local interval = redis.call('hget', KEYS[1], 'interval');"
 5:           + "local type = redis.call('hget', KEYS[1], 'type');"
 6:           + "assert(rate ~= false and interval ~= false and type ~= false, 'RateLimiter is not initialized')"
 7:
 8:           + "local valueName = KEYS[2];"
 9:           + "if type == '1' then "
10:               + "valueName = KEYS[3];"
11:           + "end;"
12:
13:           + "local currentValue = redis.call('get', valueName); "
14:           + "if currentValue ~= false then "
15:                 + "if tonumber(currentValue) < tonumber(ARGV[1]) then "
16:                     + "return redis.call('pttl', valueName); "
17:                 + "else "
18:                     + "redis.call('decrby', valueName, ARGV[1]); "
19:                     + "return nil; "
20:                 + "end; "
21:           + "else "
22:                 + "assert(tonumber(rate) >= tonumber(ARGV[1]), 'Requested permits amount could not exceed defin
23:                 + "redis.call('set', valueName, rate, 'px', interval); "
24:                 + "redis.call('decrby', valueName, ARGV[1]); "
25:                 + "return nil; "
26:           + "end;",
27:             Arrays.<Object>asList(getName(), getValueName(), getClientValueName()), // KEYS [限流器的名字，限流器
28:             value, commandExecutor.getConnectionManager().getId()); // ARGS [需要获取令牌的数量，Client 编号]
29: }
```

还是熟悉的配方，通过 Lua 脚本实现。具体传入的参数，朋友看下第 27 和 28 行的代码，对应的 KEYS 和 ARGS 。可能有几个值胖友会有点懵逼，我们先来看看。

- KEYS[1] ：调用 #getName() 方法获得限流器的名字。这个，我们在上面已经看到了。
- KEYS[2] ：调用 #getValueName() 方法，获得限流器的值的名字。该名字，就是在 Redis 存储限流值的 KEY 。代码如下：

```
// RedissonObject.java
public static String suffixName(String name, String suffix) {
    if (name.contains("{")) {
        return name + ":" + suffix;
    }
    return "{" + name + "}:" + suffix;
}

// RedissonRateLimiter.java
String getValueName() {
    return suffixName(getName(), "value");
}
```

  - 例如说，"myRateLimiter" 对应的是 "{myRateLimiter}:value" 。示例如下：

```
127.0.0.1:6379> scan 0
1) "0"
2) 1) "{myRateLimiter}:value"
   2) "myRateLimiter"
```

- 要注意，因为 "{myRateLimiter}:value" 设置了自动过期时间，所以如果要看，将限流器的配置的限流频率修改成分钟。

- KEYS[3] ：调用 #getClientValueName() 方法，获得使用 PER_CLIENT 限流类型时，每个 Client 所对应的值的名字。代码如下：

```
// RedissonRateLimiter.java
String getClientValueName() {
    return suffixName(getValueName(), commandExecutor.getConnectionManager().getId());
}
```

- 例如说，芮芮启动了使用了 "myRateLimiter" 的多个 Client ，示例如下：

```
127.0.0.1:6379> scan 0
1) "0"
2) 1) "{myRateLimiter}:value:f6059588-7693-4b6f-9413-edc24182252b"
   2) "myRateLimiter"
   3) "{myRateLimiter}:value:d751d219-c7c6-4ec1-98c2-bc1794dffd70"
```

- 如果理解起来有点绕，不用方，下面我们还会继续看代码的。

- ARGS[1] ：需要获得令牌的数量。
- ARGS[2] ：忽略，实际脚本未使用到。

第 3 至 6 行：获得限流器的配置。

第 8 至 11 行：根据 type（限流类型），获得对应的值 KEY 。在结合下 KEYS[2] 和 KEYS[3] ，是不是就清晰多了。

第 13 行：获得限流对应的值。 上面一直忘记说了，该值表示剩余可以获取的令牌数。后续，会分成两种情况处理。

第 14 至 20 行：情况一，获取的到值。

- 第 15 至 16 行：剩余的令牌数不够获取，则通过 PTTL 指令，获得 KEY 剩余的过期时间，并返回。因为剩余令牌不够了，所以返回给应用，令牌刷新还需要多久时间（即过期就刷新啦）。注意，我们注意下所有返回值，实际上当返回 null 的时候，代表获取令牌成功；返回非 null 的时，就是令牌刷新还需要多久时间，表示获取令牌失败。
- 第 17 至 20 行：令牌足够，扣除令牌，并返回成功 null 。

第 21 至 26 行：情况二，获取不到值，说明令牌是满的。

- 第 23 行：通过 SET 指令，设置令牌为 valueName 满的，并设置过期时间为 interval 。
- 第 24 行：调用 DECRBY 指令，扣除需要的令牌数。 不造为啥 Redisson 把 23 和 24 行使用一条指令解决。
- 第 25 行：返回成功 null 。

下面，我们来看看 #tryAcquireAsync(long permits, RPromise<Boolean> promise, long timeoutInMillis) 方法，是怎么和上述方法结合的。代码如下：

```
// RedissonRateLimiter.java
```

```
private void tryAcquireAsync(long permits, RPromise<Boolean> promise, long timeoutInMillis) {
    // 获得当前时间
    long s = System.currentTimeMillis();
    // 执行获得令牌
    RFuture<Long> future = tryAcquireAsync(RedisCommands.EVAL_LONG, permits);
    // 通过 future 回调处理执行结果
    future.onComplete((delay, e) -> {
        // 发生异常，则 return 并通过 promise 处理异常
        if (e != null) {
            promise.tryFailure(e);
            return;
        }

        // 如果未返回 delay ，而是空，说明获取锁成功了，则 return 并通过 promise 返回获得锁成功。
        if (delay == null) {
            promise.trySuccess(true);
            return;
        }

        // 如果 timeoutInMillis 为 -1 ，表示持续获得到锁，直到成功。
        if (timeoutInMillis == -1) {
            // 通过定时任务，实现延迟 delay 毫秒，再次执行获得令牌。
            commandExecutor.getConnectionManager().getGroup().schedule(() -> {
                tryAcquireAsync(permits, promise, timeoutInMillis);
            }, delay, TimeUnit.MILLISECONDS);
            return;
        }

        // 计算剩余可获取令牌的时间
        long el = System.currentTimeMillis() - s;
        long remains = timeoutInMillis - el;
        // 如果无剩余时间，则 return 并通过 promise 返回获得锁失败。
        if (remains <= 0) {
            promise.trySuccess(false);
            return;
        }
        // 剩余时间小于锁刷新时间，则最后还是尝试一次延迟 remains 毫秒，再次执行获得锁
        if (remains < delay) {
            commandExecutor.getConnectionManager().getGroup().schedule(() -> {
                promise.trySuccess(false);
            }, remains, TimeUnit.MILLISECONDS);
        } else {
            // 剩余时间大于锁刷新时间，则延迟 delay 毫秒，再次执行获得锁
            long start = System.currentTimeMillis();
            commandExecutor.getConnectionManager().getGroup().schedule(() -> {
                // 因为定时器是延迟 delay 毫秒，实际可能超过 remains 毫秒，此处判断兜底，避免无效重试。
                long elapsed = System.currentTimeMillis() - start;
                if (remains <= elapsed) {
                    promise.trySuccess(false);
                    return;
                }

                tryAcquireAsync(permits, promise, remains - elapsed);
            }, delay, TimeUnit.MILLISECONDS);
        }
    });
}
```

虽然逻辑比较长，实际逻辑比较简单。首先，调用 #tryAcquireAsync(RedisCommand<T> command, Long

value) 方法，执行获得令牌。然后，通过设置返回的 RFuture 的 #onComplete(BiConsumer<? super V, ? super Throwable> action) 方法，回调处理执行令牌的返回结果。
在回调中，如果获取令牌成功，则 return 并通过 promise 返回成功；如果获取令牌失败，则根据是否有足够的时间，去延迟调用 #tryAcquireAsync(RedisCommand<T> command, Long value) 方法，执行获得令牌，直到成功或超时或异常。
整体逻辑，胖友自己瞅瞅，哈哈哈，感觉满巧妙的。 一开始，以为 Redisson 是基于令牌桶算法，实现限流算法，所以卡了莫名的久，被自己给蠢哭了。

下面，我们来看看 ##tryAcquire(...) 方法们的实现，就灰常简单。代码如下：

```
// RedissonRateLimiter.java

@Override
public boolean tryAcquire() {
    return tryAcquire(1);
}

@Override
public RFuture<Boolean> tryAcquireAsync() {
    return tryAcquireAsync(1L);
}

@Override
public boolean tryAcquire(long permits) {
    return get(tryAcquireAsync(RedisCommands.EVAL_NULL_BOOLEAN, permits));
}

@Override
public RFuture<Boolean> tryAcquireAsync(long permits) {
    return tryAcquireAsync(RedisCommands.EVAL_NULL_BOOLEAN, permits);
}

@Override
public boolean tryAcquire(long timeout, TimeUnit unit) {
    return get(tryAcquireAsync(timeout, unit));
}

@Override
public RFuture<Boolean> tryAcquireAsync(long timeout, TimeUnit unit) {
    return tryAcquireAsync(1, timeout, unit);
}

@Override
public boolean tryAcquire(long permits, long timeout, TimeUnit unit) {
    return get(tryAcquireAsync(permits, timeout, unit));
}

@Override
public RFuture<Boolean> tryAcquireAsync(long permits, long timeout, TimeUnit unit) {
    // 创建 RPromise 对象
    RPromise<Boolean> promise = new RedissonPromise<Boolean>();
    // 计算 timeoutInMillis
    long timeoutInMillis = -1;
    if (timeout >= 0) {
        timeoutInMillis = unit.toMillis(timeout);
    }
    // 执行获取令牌
    tryAcquireAsync(permits, promise, timeoutInMillis);
    return promise;
```

```
}
```

6. acquire

acquire(...) 方法们的实现，也灰常简单。代码如下：

```
// RedissonRateLimiter.java

@Override
public void acquire(long permits) {
    get(acquireAsync(permits));
}

@Override
public RFuture<Void> acquireAsync(long permits) {
    // 创建 RPromise 对象
    RPromise<Void> promise = new RedissonPromise<Void>();
    // 执行获得令牌。通过 -1 ，表示重试到成功为止
    tryAcquireAsync(permits, -1, null).onComplete((res, e) -> {
        // 处理异常
        if (e != null) {
            promise.tryFailure(e);
            return;
        }

        // 成功
        promise.trySuccess(null);
    });
    return promise;
}
```

666. 彩蛋

行至文末，有一点要纠正一下。Redisson 提供的限流器不是严格且完整的滑动窗口的限流器实现。举个例子，我们创建了一个每分钟允许 3 次操作的限流器。整个执行过程如下：

```
00:00:00 获得锁，剩余令牌 2 。
00:00:20 获得锁，剩余令牌 1 。
00:00:40 获得锁，剩余令牌 0 。
```

> 那么，00:01:00 时，锁的数量会恢复，按照 Redisson 的限流器来说。
> 如果是严格且完整的滑动窗口的限流器，此时在 00:01:00 剩余可获得的令牌数为 1 ，也就是说，起始点应该变成 00:00:20 。

如果基于 Redis 严格且完整的滑动窗口的限流器，可以看看芳芳在 《精尽 Redis 面试题》 的 「如何使用 Redis 实现分布式限流？」 ，提供基于 Redis Zset 实现。

勉勉强强，在 2019-10-02 的 22:32 写完了这篇博客，美滋滋。T T 今天的 Leetcode 忘记刷了，赶紧去刷下。

文章目录

回到首页