公告

昵称： RGC
园龄： 3年3个月
粉丝： 11
关注： 1
+加关注

< 2020年9月 >

日	一	二	三	四	五	六
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	1	2	3
4	5	6	7	8	9	10

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

python(8)
mysql(7)
redis(6)
flask(5)
sqlalchemy(5)
windows(4)
nginx(4)
django(2)
pycharm(2)
gunicorn(2)
更多

随笔档案 (80)

2019年12月(1)
2019年11月(2)
2019年10月(3)
2019年8月(3)
2019年7月(10)
2019年6月(2)
2019年5月(1)
2019年4月(1)
2019年3月(3)
2019年2月(1)
2019年1月(2)
2018年12月(2)
2018年10月(11)
2018年9月(15)
2018年8月(1)

Redlock（redis分布式锁）原理分析

Redlock：全名叫做 Redis Distributed Lock;即使用redis实现的分布式锁;

使用场景：多个服务间保证同一时刻同一时间段内同一用户只能有一个请求（防止关键业务出现并发攻击）；

官网文档地址如下：https://redis.io/topics/distlock

这个锁的算法实现了多redis实例的情况，相对于单redis节点来说，优点在于 防止了 单节点故障造成整个服务停止运行的情况；并且在多节点中锁的设计，及多节点同时崩溃等各种意外情况有自己独特的设计方法；

此博客或者官方文档的相关概念：

1.TTL：Time To Live;只 redis key 的过期时间或有效生存时间

2.clock drift:时钟漂移；指两个电脑间时间流速基本相同的情况下，两个电脑（或两个进程间）时间的差值；如果电脑距离过远会造成时钟漂移值 过大

最低保证分布式锁的有效性及安全性的要求如下：

1.互斥；任何时刻只能有一个client获取锁

2.释放死锁；即使锁定资源的服务崩溃或者分区，仍然能释放锁

3.容错性；只要多数redis节点（一半以上）在使用，client就可以获取和释放锁

网上讲的基于故障转移实现的redis主从无法真正实现Redlock:

因为redis在进行主从复制时是异步完成的，比如在clientA获取锁后，主redis复制数据到从redis过程中崩溃了，导致没有复制到从redis中，然后从redis选举出一个升级为主redis,造成新的主redis没有clientA 设置的锁，这是clientB尝试获取锁，并且能够成功获取锁，导致互斥失效；

思考题：这个失败的原因是因为从redis立刻升级为主redis，如果能够过TTL时间再升级为主redis（延迟升级）后，或者立刻升级为主redis但是过TTL的时间后再执行获取锁的任务，就能成功产生互斥效果；是不是这样就能实现基于redis主从的Redlock;

redis单实例中实现分布式锁的正确方式（原子性非常重要）：

1.设置锁时，使用set命令，因为其包含了setnx,expire的功能，起到了原子操作的效果，给key设置随机值，并且只有在key不存在时才设置成功返回True,并且设置key的过期时间（最好用毫秒）

```
SET key_name my_random_value NX PX 30000                # NX 表示if not exist 就设置并返回True，否则不设
```

2.在获取锁后，并完成相关业务后，需要删除自己设置的锁（必须是只能删除自己设置的锁，不能删除他人设置的锁）；

删除原因：保证服务器资源的高利用效率，不用等到锁自动过期才删除；

删除方法：最好使用Lua脚本删除（redis保证执行此脚本时不执行其他操作，保证操作的原子性），代码如下；逻辑是 先获取key，如果存在并且值是自己设置的就删除此key;否则就跳过；

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

2018年6月(4)
2018年5月(1)
2018年3月(6)
2018年1月(6)
2017年12月(1)
2017年7月(3)
2017年6月(1)

最新评论

1. Re:Redlock（redis分布式锁）原理分析
大佬你好，看了你的文章受益匪浅，但有一些疑问，还请指教。 为什么主从复制故障的时候要延迟重启？ 我理解的是：要等到原来的分布式锁过期 再去接受请求 保证不会有多个客户端同时获取锁 不知道我理解的对不对...

--AmosWong

2. Re:windows10安装docker[含百度网盘docker安装包]
老哥，你这个百度云盘里的安装文件怎么800多M，官网下的300多

--烟味i

3. Re:Redlock（redis分布式锁）原理分析
@renguanyu 逗比，真逗...

--景成乐天

4. Re:Redlock（redis分布式锁）原理分析
收藏从未停止，学习从未开始

--renguanyu

5. Re:Redlock（redis分布式锁）原理分析
翻译的很牛逼了啊

--盛前锋

阅读排行榜

1. Redlock（redis分布式锁）原理分析(24881)
2. sqlalchemy和flask-sqlalchemy几种分页操作(12649)
3. RESTful接口设计原则和优点(12361)
4. memory_profiler的使用(8706)
5. python中json.loads,dumps,jsonify使用(7108)

评论排行榜

1. Redlock（redis分布式锁）原理分析(6)
2. windows开机自启python服务(任务计划程序+bat脚本)(2)
3. 测试驱动开发简单理解(2)
4. windows10安装docker[含百度网盘docker安装包](1)

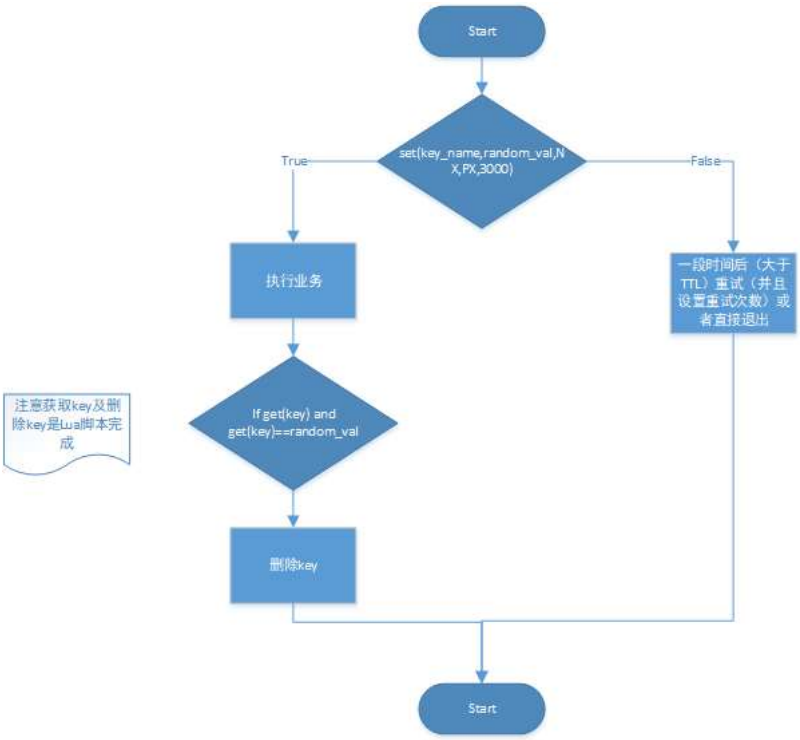
推荐排行榜

1. Redlock（redis分布式锁）原理分析(12)
2. SSE(Server-sent events)技术在web端消息推送和实时聊天中的使用(2)
3. windows开机自启python服务(任务计划程序+bat脚本)(1)
4. redis过期机制（官网文档总结）(1)
5. 测试驱动开发简单理解(1)

python代码如下：

```
redis.eval(f"""if redis.call("get",KEYS[1]) == ARGV[1] then return redis.call("del",KEYS[1]) else
```

算法流程图如下：



多节点redis实现的分布式锁算法(RedLock):有效防止单点故障

假设有5个完全独立的redis主服务器

1.获取当前时间戳

2.client尝试按照顺序使用相同的key,value获取所有redis服务的锁，在获取锁的过程中的获取时间比锁过期时间短很多，这是为了不要过长时间等待已经关闭的redis服务。并且试着获取下一个redis实例。

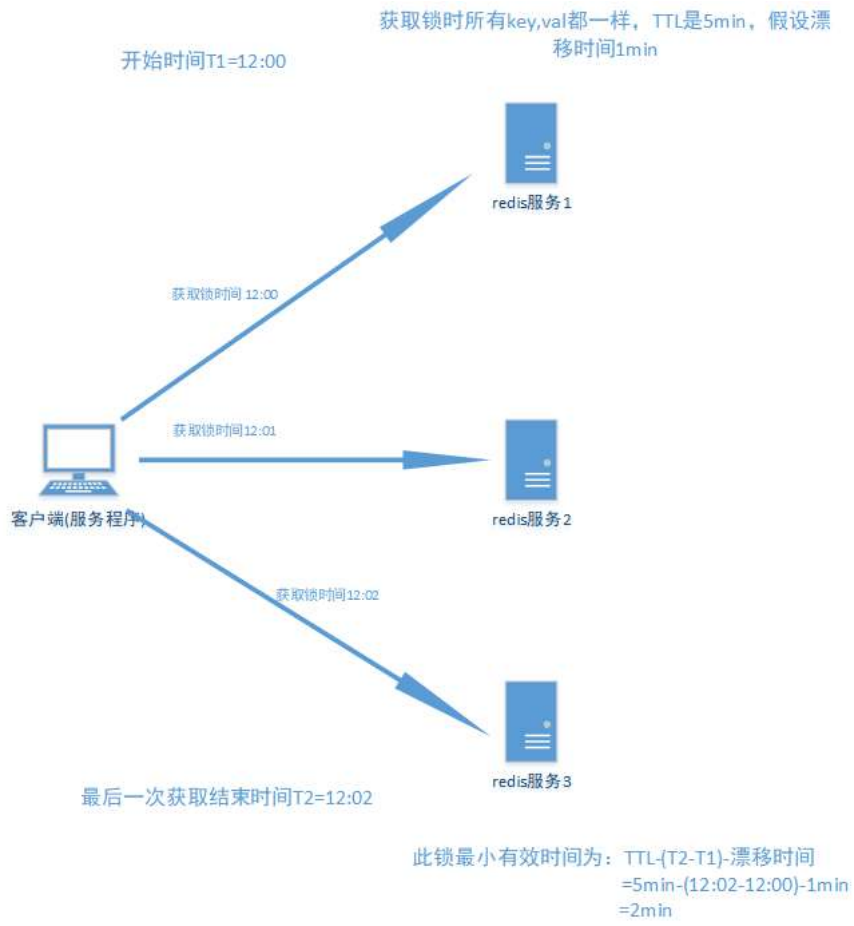
比如：TTL为5s,设置获取锁最多用1s，所以如果一秒内无法获取锁，就放弃获取这个锁，从而尝试获取下个锁

3.client通过获取所有能获取的锁后的时间减去第一步的时间，这个时间差要小于TTL时间并且至少有3个redis实例成功获取锁，才算真正的获取锁成功

4.如果成功获取锁，则锁的真正有效时间是 TTL减去第三步的时间差 的时间；比如：TTL 是5s,获取所有锁用了2s,则真正锁有效时间为3s(其实应该再减去时钟漂移);

5.如果客户端由于某些原因获取锁失败，便会开始解锁所有redis实例；因为可能已经获取了小于3个锁，必须释放，否则影响其他client获取锁

算法示意图如下：



RedLock算法是否是异步算法？？

可以看成是同步算法；因为 即使进程间（多个电脑间）没有同步时钟，但是每个进程时间流速大致相同；并且时钟漂移相对于TTL叫小，可以忽略，所以可以看成同步算法；（不够严谨，算法上要算上时钟漂移，因为如果两个电脑在地球两端，则时钟漂移非常大）

RedLock失败重试

当client不能获取锁时，应该在随机时间后重试获取锁；并且最好在同一时刻并发的把set命令发送给所有redis实例；而且对于已经获取锁的client在完成任务后要及时释放锁，这是为了节省时间；

RedLock释放锁

由于释放锁时会判断这个锁的value是不是自己设置的，如果是才删除；所以在释放锁时非常简单，只要向所有实例都发出释放锁的命令，不用考虑能否成功释放锁；

RedLock注意点（Safety arguments）：

1.先假设client获取所有实例，所有实例包含相同的key和过期时间(TTL) ,但每个实例set命令时间不同导致不能同时过期，第一个set命令之前是T1,最后一个set命令后为T2,则此client有效获取锁的最小时间为TTL-(T2-T1)-时钟漂移;

2.对于以N/2+ 1(也就是一半以 上)的方式判断获取锁成功，是因为如果小于一半判断为成功的话，有可能出现多个client都成功获取锁的情况， 从而使锁失效

3.一个client锁定大多数事例耗费的时间大于或接近锁的过期时间，就认为锁无效，并且解锁这个redis实例(不执行业务) ;只要在TTL时间内成功获取一半以上的锁便是有效锁;否则无效

系统有活性的三个特征

1.能够自动释放锁

2.在获取锁失败（不到一半以上），或任务完成后 能够自动释放锁，不用等到其自动过期

3.在client重试获取哦锁前（第一次失败到第二次重试时间间隔）大于第一次获取锁消耗的时间；

4.重试获取锁要有一定次数限制

RedLock性能及崩溃恢复的相关解决方法

1.如果redis没有持久化功能，在clientA获取锁成功后，所有redis重启，clientB能够再次获取到锁，这样违法了锁的排他互斥性；

2.如果启动AOF永久化存储，事情会好些， 举例:当我们重启redis后，由于redis过期机制是按照unix时间戳走的，所以在重启后，然后会按照规定的时间过期，不影响业务;但是由于AOF同步到磁盘的方式默认是每秒-次，如果在一秒内断电，会导致数据丢失，立即重启会造成锁互斥性失效;但如果同步磁盘方式使用Always(每一个写命令都同步到硬盘)造成性能急剧下降;所以在锁完全有效性和性能方面要有所取舍;

3.有效解决既保证锁完全有效性及性能高效及即使断电情况的方法是redis同步到磁盘方式保持默认的每秒，在redis无论因为什么原因停掉后要等待TTL时间后再重启(学名:**延迟重启**）;缺点是 在TTL时间内服务相当于暂停状态;

总结:

1.TTL时长 要大于正常业务执行的时间+获取所有redis服务消耗时间+时钟漂移

2.获取redis所有服务消耗时间要 远小于TTL时间，并且获取成功的锁个数要 在总数的一般以上:N/2+1

3.尝试获取每个redis实例锁时的时间要 远小于TTL时间

4.尝试获取所有锁失败后 重新尝试一定要有一定次数限制

5.在redis崩溃后（无论一个还是所有），要延迟TTL时间重启redis

6.在实现多redis节点时要结合单节点分布式锁算法 共同实现

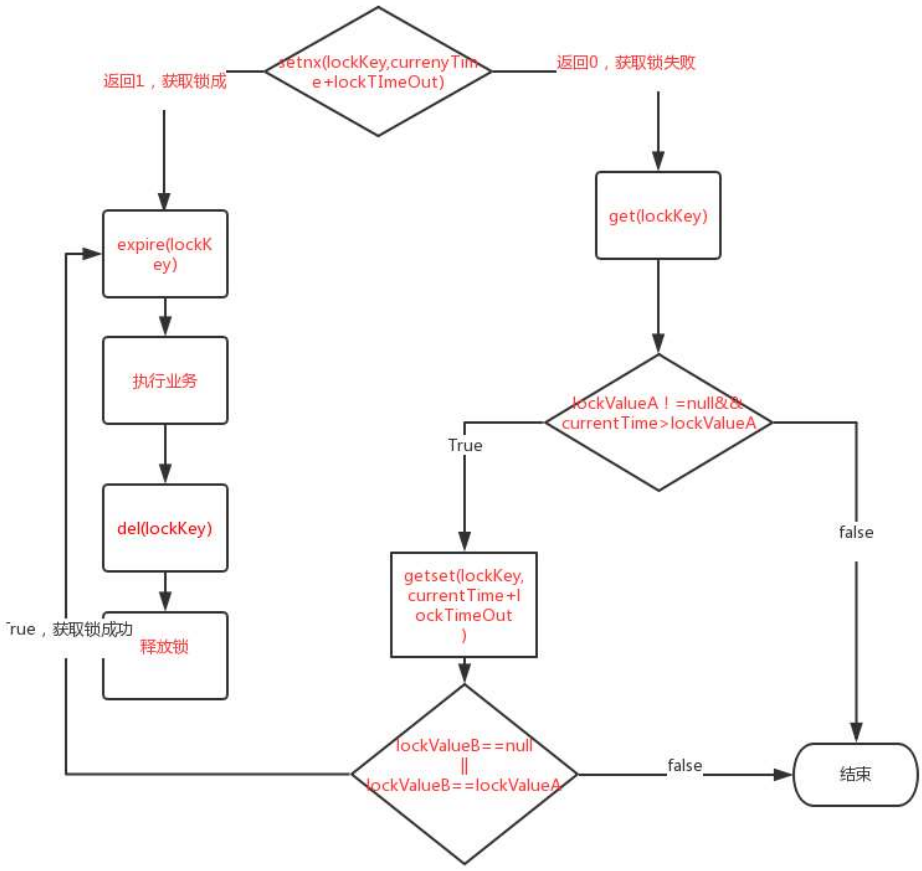
网络上查找的redis分布式锁 算法流程图如下（**不推荐使用**）:

不推荐原因:

1.根据流程图可看出其流程较为繁琐

2.使用较为老式的 setnx方法获取锁及expire方法（无法保证原子操作）

3.redis单点，无法做到错误兼容性;



如下为官网解析（英语水平不够，如有理解问题，请指出）：

redis Commands Clients Documentation Community Download Modules Support

Distributed locks with Redis

redis分布式锁

Distributed locks are a very useful primitive in many environments where different processes must operate with shared resources in a mutually exclusive way.

There are a number of libraries and blog posts describing how to implement a DLM (Distributed Lock Manager) with Redis, but every library uses a different approach, and many use a simple approach with lower guarantees compared to what can be achieved with slightly more complex designs.

This page is an attempt to provide a more canonical algorithm to implement distributed locks with Redis. We propose an algorithm, called **Redlock**, which implements a DLM which we believe to be safer than the vanilla single instance approach. We hope that the community will analyze it, provide feedback, and use it as a starting point for the implementations or more complex or alternative designs.

Implementations

Before describing the algorithm, here are a few links to implementations already available that can be used for reference.

已经实现的第三方可用的资源链接，包括各种语言

- Redlock-rb (Ruby implementation). There is also a fork of Redlock-rb that adds a gem for easy distribution and perhaps more.
- Redlock-py (Python implementation).
- Aioredlock (Asyncio Python implementation).
- Redlock-php (PHP implementation).
- PHPRedisMutex (further PHP implementation)
- cheprasov/php-redis-lock (PHP library for locks)
- Redsync.go (Go implementation).
- Redisson (Java implementation).
- Redis::DistLock (Perl implementation).
- Redlock-cpp (C++ implementation).
- Redlock-cs (C#/.NET implementation).
- RedLock.net (C#/.NET implementation). Includes async and lock extension support.
- ScarletLock (C# .NET implementation with configurable datastore)
- node-redlock (NodeJS implementation). Includes support for lock extension.

Safety and Liveness guarantees

安全及长时间有效性

We are going to model our design with just three properties that, from our point of view, are the minimum guarantees needed to use distributed locks in an effective way.

最低有效保证分布式锁有效性的要求如下

1. Safety property: Mutual exclusion. At any given moment, only one client can hold a lock.
2. Liveness property A: Deadlock free. Eventually it is always possible to acquire a lock, even if the client that locked a resource crashes or gets partitioned.
3. Liveness property B: Fault tolerance. As long as the majority of Redis nodes are up, clients are able to acquire and release locks.

安全性：互斥；任何时刻，只能有一个client获取锁

释放死锁；即使锁定资源的服务崩溃或者分区，仍能获取锁

容错性；只要多个redis节点在使用，client就可以获取及释放锁

Why failover-based implementations are not enough

为什么以故障转移为基础的分布式锁 是不够的呢？？

To understand what we want to improve, let's analyze the current state of affairs with most Redis-based distributed lock libraries.

The simplest way to use Redis to lock a resource is to create a key in an instance. The key is usually created with a limited time to live, using the Redis expires feature, so that eventually it will get released (property 2 in our list). When the client needs to release the resource, it deletes the key.

Superficially this works well, but there is a problem: this is a single point of failure in our architecture. What happens if the Redis master goes down? Well, let's add a slave! And use it if the master is unavailable. This is unfortunately not viable. By doing so we can't implement our safety property of mutual exclusion, because Redis replication is asynchronous.

因为redis在进行主从复制时是 异步的，也就是在 clientA 获取锁后，在主redis复制数据到从redis时 崩溃了，导致主redis的数据没有复制到从redis中，在从redis转为主redis时 能够继续获取锁；这是错误的

There is an obvious race condition with this model:

1. Client A acquires the lock in the master.
2. The master crashes before the write to the key is transmitted to the slave.
3. The slave gets promoted to master.
4. Client B acquires the lock to the same resource A already holds a lock for. **SAFETY VIOLATION!**

Sometimes it is perfectly fine that under special circumstances, like during a failure, multiple clients can hold the lock at the same time. If this is the case, you can use your replication based solution. Otherwise we suggest to implement the solution described in this document.

Correct implementation with a single instance 单节点中正确实现方式

Before trying to overcome the limitation of the single instance setup described above, let's check how to do it correctly in this simple case, since this is actually a viable solution in applications where a race condition from time to time is acceptable, and because locking into a single instance is the foundation we'll use for the distributed algorithm described here.

To acquire the lock, the way to go is the following:

设置锁时，使用set命令，包含了setnx，expire命令，起到了原子操作的效果（这个很重要），给key设置随机值，并且只当key不存在时才设置，并且设置key过期时间;也就是返回值为1 表示获取资源成功，返回值为0表示获取资源失败

```
SET resource_name my_random_value NX PX 30000
```

The command will set the key only if it does not already exist (NX option), with an expire of 30000 milliseconds (PX option). The key is set to a value "my*random*value". This value must be unique across all clients and all lock requests.

Basically the random value is used in order to release the lock in a safe way, with a script that tells Redis: remove the key only if it exists and the value stored at the key is exactly the one I expect to be. This is accomplished by the following Lua script:

在删除锁时，最好使用Lua脚本（执行这个脚本时不会执行其他操作）嵌套近程序中执行，保证校验是否是本客户端设置的key，并删除时的原子操作；
保证创建key和删除key必须是同一个客户端，判断方法是 看 value是不是这个客户端创建的 这个机制是为了确保 不要删除别的客户端设置的key，否则会造成锁失效

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

This is important in order to avoid removing a lock that was created by another client. For example a client may acquire the lock, get blocked in some operation for longer than the lock validity time (the time at which the key will expire), and later remove the lock, that was already acquired by some other client. Using just DEL is not safe as a client may remove the lock of another client. With the above script instead every lock is "signed" with a random string, so the lock will be removed only if it is still the one that was set by the client trying to remove it.

What should this random string be? I assume it's 20 bytes from /dev/urandom, but you can find cheaper ways to make it unique enough for your tasks. For example a safe pick is to seed RC4 with /dev/urandom, and generate a pseudo random stream from that. A simpler solution is to use a combination of unix time with microseconds resolution, concatenating it with a client ID, it is not as safe, but probably up to the task in most environments.

The time we use as the key time to live, is called the "lock validity time". It is both the auto release time, and the time the client has in order to perform the operation required before another client may be able to acquire the lock again, without technically violating the mutual exclusion guarantee, which is only limited to a given window of time from the moment the lock is acquired.

So now we have a good way to acquire and release the lock. The system, reasoning about a non-distributed system composed of a single, always available, instance, is safe. Let's extend the concept to a distributed system where we don't have such guarantees.

The Redlock algorithm 红锁算法

In the distributed version of the algorithm we assume we have N Redis masters. Those nodes are totally independent, so we don't use replication or any other implicit coordination system. We already described how to acquire and release the lock safely in a single instance. We take for granted that the algorithm will use this method to acquire and release the lock in a single instance. In our examples we set N=5, which is a reasonable value, so we need to run 5 Redis masters on different computers or virtual machines in order to ensure that they'll fail in a mostly independent way.

设置5个完全独立的redis主服务

In order to acquire the lock, the client performs the following operations: 为了获取锁，执行如下操作

1. It gets the current time in milliseconds. 获取当前时间戳
2. It tries to acquire the lock in all the N instances sequentially, using the same key name and random value in all the instances. During step 2, when setting the lock in each instance, the client uses a timeout which is small compared to the total lock auto-release time in order to acquire it. For example if the auto-release time is 10 seconds, the timeout could be in the ~ 5-50 milliseconds range. This prevents the client from remaining blocked for a long time trying to talk with a Redis node which is down: if an instance is not available, we should try to talk with the next instance ASAP.

client尝试按照顺序获取所有redis服务的锁，并且获取锁这个过程的过期时间比 锁过期时间短，这样为了不要过长时间等待已经关闭的redis服务。并且试着去获取下一个redis服务的示例。 比如：获取锁后，这个锁过期时间为5s,在获取锁的过程只用1s,如果1s内无法获取锁，就放弃获取这个服务，从而获取下个实例的锁

3. The client computes how much time elapsed in order to acquire the lock, by subtracting from the current time the timestamp obtained in step 1. If and only if the client was able to acquire the lock in the majority of the instances (at least 3), and the total time elapsed to acquire the lock is less than lock validity time, the lock is considered to be acquired.

所有实例的

client通过当前时间（获取锁后）减去当时生成的时间，只有在这个时间差值小于锁的过期时间，并且至少在3个redis主机都成功获取锁时，才能算这次成功获取锁

4. If the lock was acquired, its validity time is considered to be the initial validity time minus the time elapsed, as computed in step 3. 如果成功获取锁，真正有效时间就是 初始有效时间减去 获取锁用的时间 比如初始设置有效时间5s,获取锁用了1s,则有效时间为4s；
5. If the client failed to acquire the lock for some reason (either it was not able to lock N/2+1 instances or the validity time is negative), it will try to unlock all the instances (even the instances it believed it was not able to lock). 如果客户端由于某些原因（包括获取锁事例小于N/2+1个）无法获取锁，便会开始解锁所有redis服务；因为可能已经获取了小于N/2+1个锁，必须释放，否则影响其他client获取锁

Is the algorithm asynchronous? 算法是否异步?

The algorithm relies on the assumption that while there is no synchronized clock across the processes, still the local time in every process flows approximately at the same rate, with an error which is small compared to the auto-release time of the lock. This assumption closely resembles a real-world computer: every computer has a local clock and we can usually rely on different computers to have a clock drift which is small.

可以看出同步，因为 虽然进程间没有同步时钟，但每个进程时间流动速度大致相同和锁过期时间相比，误差较小；

At this point we need to better specify our mutual exclusion rule: it is guaranteed only as long as the client holding the lock will terminate its work within the lock validity time (as obtained in step 3), minus some time (just a few milliseconds in order to compensate for clock drift between processes).

clock drift：时钟漂移 指 由于两个电脑间有自己的本地时钟，但是由于彼此时间流速大致相同，彼此的时间差 就是 时钟漂移；由于时间差相对于锁过期时间较短（可以忽略），所以可以看出同步算法；（其实不够严谨：算法上要算上这个时间差，因为两个redis实例可能在地球两端，其时间差很大）

For more information about similar systems requiring a bound *clock drift*, this paper is an interesting reference: Leases: an efficient fault-tolerant mechanism for distributed file cache consistency.

Retry on failure 失败重试

When a client is unable to acquire the lock, it should try again after a random delay in order to try to desynchronize multiple clients trying to acquire the lock for the same resource at the same time (this may result in a split brain condition where nobody wins). Also the faster a client tries to acquire the lock in the majority of Redis instances, the smaller the window for a split brain condition (and the need for a retry), so ideally the client should try to send the SET commands to the N instances at the same time using multiplexing.

当client不能获取锁时，应该在随机时间后重试获取锁；并且最好在同一时刻并发的把set命令发送给所有redis实例；
而且对于已经获取锁的client在完成任务后要及时释放锁，这样其他client就不用等到锁自动过期后再获取锁，从而节省时间；
如果在获取锁后，但在手动删除锁之前client无法连接到redis实例，就必须等到锁自动过期，这会影响整体效率也是一种惩罚

It is worth stressing how important it is for clients that fail to acquire the majority of locks, to release the (partially) acquired locks ASAP, so that there is no need to wait for key expiry in order for the lock to be acquired again (however if a network partition happens and the client is no longer able to communicate with the Redis instances, there is an availability penalty to pay as it waits for key expiration).

Releasing the lock 释放锁

Releasing the lock is simple and involves just releasing the lock in all instances, whether or not the client believes it was able to successfully lock a given instance.

由于释放锁时会判断这个锁的value是不是自己设置的value，如果是才删除；所以在释放锁时非常简单，只要向所有实例都发出释放锁的命令，不用考虑能否成功释放锁；

Safety arguments

Is the algorithm safe? We can try to understand what happens in different scenarios.

To start let's assume that a client is able to acquire the lock in the majority of instances. All the instances will contain a key with the same time to live. However, the key was set at different times, so the keys will also expire at different times. But if the first key was set at worst at time T1 (the time we sample before contacting the first server) and the

先假设client获取所有实例，所有实例包含相同的key和过期时间(TTL)，但每个实例set命令时间不同导致不能同时过期，第一个set命令之前是T1,最后一个set命令后为T2,则此client有效获取锁的最小

last key was set at worst at time T2 (the time we obtained the reply from the last server), we are sure that the first key to expire in the set will exist for at least `MIN_VALIDITY=TTL-(T2-T1)-CLOCK_DRIFT`. All the other keys will expire later, so we are sure that the keys will be simultaneously set for at least this time.

时间为TTL-(T2-T1)-时钟漂移;

During the time that the majority of keys are set, another client will not be able to acquire the lock, since N/2+1 SET NX operations can't succeed if N/2+1 keys already exist. So if a lock was acquired, it is not possible to re-acquire it at the same time (violating the mutual exclusion property).

对于以 N/2+1(也就是一半以上) 的方式判断获取锁成功，是因为如果小于一半判断为成功的话，有可能出现多个client都成功获取锁的情况，从而使锁失效

However we want to also make sure that multiple clients trying to acquire the lock at the same time can't simultaneously succeed.

If a client locked the majority of instances using a time near, or greater, than the lock maximum validity time (the TTL we use for SET basically), it will consider the lock invalid and will unlock the instances, so we only need to consider the case where a client was able to lock the majority of instances in a time which is less than the validity time. In this case for the argument already expressed above, for `MIN_VALIDITY` no client should be able to re-acquire the lock. So multiple clients will be able to lock N/2+1 instances at the same time (with "time" being the end of Step 2) only when the time to lock the majority was greater than the TTL time, making the lock invalid.

一个client锁定大多数事例耗费的时间大于或接近锁的过期时间，就认为锁无效，并且解锁这个redis实例（不执行业务）；
只要在TTL时间内成功获取一半以上的锁便是有效锁；否则无效

Are you able to provide a formal proof of safety, point to existing algorithms that are similar, or find a bug? That would be greatly appreciated.

Liveness arguments

系统具有活性的三个特征

The system liveness is based on three main features:

1. The auto release of the lock (since keys expire): eventually keys are available again to be locked.
2. The fact that clients, usually, will cooperate removing the locks when the lock was not acquired, or when the lock was acquired and the work terminated, making it likely that we don't have to wait for keys to expire to re-acquire the lock.
3. The fact that when a client needs to retry a lock, it waits a time which is comparably greater than the time needed to acquire the majority of locks, in order to probabilistically make split brain conditions during resource contention unlikely.

1.能够自动释放锁

2.在获取锁失败（不到一半以上），或任务完成后 能够释放锁，不用等到其自动过期

3.在client重试获取锁前（第一次失败到第二次重试时间间隔）大于获取锁消耗的时间

However, we pay an availability penalty equal to TTL time on network partitions, so if there are continuous partitions, we can pay this penalty indefinitely. This happens every time a client acquires a lock and gets partitioned away before being able to remove the lock.

Basically if there are infinite continuous network partitions, the system may become not available for an infinite amount of time.

Performance, crash-recovery and fsync

Many users using Redis as a lock server need high performance in terms of both latency to acquire and release a lock, and number of acquire / release operations that it is possible to perform per second. In order to meet this requirement, the strategy to talk with the N Redis servers to reduce latency is definitely multiplexing (or poor man's multiplexing, which is, putting the socket in non-blocking mode, send all the commands, and read all the commands later, assuming that the RTT between the client and each instance is similar).

However there is another consideration to do about persistence if we want to target a crash-recovery system model.

对于崩溃恢复的策略

Basically to see the problem here, let's assume we configure Redis without persistence at all. A client acquires the lock in 3 of 5 instances. One of the instances where the client was able to acquire the lock is restarted, at this point there are again 3 instances that we can lock for the same resource, and another client can lock it again, violating the safety property of exclusivity of lock.

如果redis没有持久化功能，在clientA获取锁成功后，所有redis重启，clientB能够再次获取到锁，这样违法了锁的排他互斥性；

If we enable AOF persistence, things will improve quite a bit. For example we can upgrade a server by sending SHUTDOWN and restarting it. Because Redis expires are semantically implemented so that virtually the time still elapses when the server is off, all our requirements are fine. However everything is fine as long as it is a clean shutdown. What about a power outage? If Redis is configured, as by default, to fsync on disk every second, it is possible that after a restart our key is missing. In theory, if we want to guarantee the lock safety in the face of any kind of instance restart, we need to enable fsync=always in the persistence setting. This in turn will totally ruin performances to the same level of CP systems that are traditionally used to implement distributed locks in a safe way.

如果启动AOF永久化存储，事情会好一些，举例：当我们重启redis后，由于redis过期机制是按照unix时间戳走的，所以在重启后，然后会按照规定的时间过期，不影响业务；但是由于AOF同步到磁盘的方式默认是每秒一次，如果在一秒内断电，会导致数据丢失，立即重启会造成锁互斥性失效；但如果同步磁盘方式使用Always(每一个写命令都同步到硬盘)造成性能急剧下降；所以在锁完全有效性和性能方面要有所取舍；

However things are better than what they look like at a first glance. Basically the algorithm safety is retained as long as when an instance restarts after a crash, it no longer participates to any **currently active** lock, so that the set of currently active locks when the instance restarts, were all obtained by locking instances other than the one which is rejoining the system.

To guarantee this we just need to make an instance, after a crash, unavailable for at least a bit more than the max TTL we use, which is, the time needed for all the keys about the locks that existed when the instance crashed, to become invalid and be automatically released.

Using *delayed restarts* it is basically possible to achieve safety even without any kind of Redis persistence available, however note that this may translate into an availability penalty. For example if a majority of instances crash, the system will become globally unavailable for TTL (here globally means that no resource at all will be lockable during this time).

有效解决 既保证锁完全有效性及性能高效及即便断电情况的方法是redis同步到磁盘方式保持默认的每秒，在redis无论因为什么原因停掉后 要等待TTL时间后 再重启（学名：延迟重启）； 缺点是 在TTL时间内服务相当于暂停状态；

Making the algorithm more reliable: Extending the lock

If the work performed by clients is composed of small steps, it is possible to use smaller lock validity times by default, and extend the algorithm implementing a lock extension mechanism. Basically the client, if in the middle of the computation while the lock validity is approaching a low value, may extend the lock by sending a Lua script to all the instances that extends the TTL of the key if the key exists and its value is still the random value the client assigned when the lock was acquired.

The client should only consider the lock re-acquired if it was able to extend the lock into the majority of instances, and within the validity time (basically the algorithm to use is very similar to the one used when acquiring the lock).

However this does not technically change the algorithm, so the maximum number of lock reacquisition attempts should be limited, otherwise one of the liveness properties is violated.

这段话总结就是：重试获取锁 要有一定次数的限制，否则违反了 系统活性特征；

Want to help?

If you are into distributed systems, it would be great to have your opinion / analysis. Also reference implementations in other languages could be great.

Thanks in advance!

Analysis of Redlock

1. Martin Kleppmann analyzed Redlock here. I disagree with the analysis and posted my reply to his analysis here.

标签: RedLock, redis分布式锁, redis

RGC
关注 - 1
粉丝 - 11
+加关注

12 1

« 上一篇： redis过期机制（官网文档总结）
» 下一篇： windows安装redis并设置别名

posted @ 2019-04-09 17:26 RGC 阅读(24881) 评论(6) 编辑 收藏

评论列表

#1楼 2019-09-17 23:11 MrSu

收下我的膝盖。。。

支持(0) 反对(0)

#2楼 2020-04-05 11:17 神雕爱大侠

mark

支持(0) 反对(0)

#3楼 2020-05-07 16:52 JeffSheng

翻译的很牛逼了啊

支持(0) 反对(0)

#4楼 2020-06-21 09:38 renguanyu

收藏从未停止，学习从未开始

支持(0) 反对(0)

#5楼 2020-08-12 10:12 景成乐天

@renguanyu
逗比，真逗

支持(0) 反对(0)

#6楼 2020-08-23 09:12 AmosWong

大佬你好，看了你的文章受益匪浅，但有一些疑问，还请指教。
为什么主从复制故障的时候要延迟重启?
我理解的是：要等到原来的分布式锁过期 再去接受请求 保证不会有多个客户端同时获取锁
不知道我理解的对不对 还请大佬指教

支持(1) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册， 访问 网站首页。

最新 IT 新闻:
· 新基建的背后，是一场算力之争
· 互联网公司高管频频出事：贪腐、受贿、涉赌
· “双星伴月”今明两天扮靓夜空：肉眼可见星月美景
· 重大突破！台积电有望在2024年量产2nm工艺芯片
· 路透社：TikTok称将允许澳大利亚政府审查算法和源代码
» 更多新闻...