



下载APP



06 | 租约：如何检测你的客户端存活？

2021-02-01 唐聪

etcd实战课

[进入课程 >](#)**讲述：王超凡**

时长 13:44 大小 12.58M



你好，我是唐聪。

今天我要跟你分享的主题是租约（Lease）。etcd 的一个典型的应用场景是 Leader 选举，那么 etcd 为什么可以用来实现 Leader 选举？核心特性实现原理又是怎样的？

今天我就和你聊聊 Leader 选举背后技术点之一的 Lease，解析它的核心原理、性能优化思路，希望通过本节让你对 Lease 如何关联 key、Lease 如何高效续期、淘汰、什么是 checkpoint 机制有深入的理解。同时希望你能基于 Lease 的 TTL 特性，解决实际业务中遇到分布式锁、节点故障自动剔除等各类问题，提高业务服务的可用性。



什么是 Lease

在实际业务场景中，我们常常会遇到类似 Kubernetes 的调度器、控制器组件同一时刻只能存在一个副本对外提供服务的情况。然而单副本部署的组件，是无法保证其高可用性的。

那为了解决单副本的可用性问题，我们就需要多副本部署。同时，为了保证同一时刻只有一个能对外提供服务，我们需要引入 Leader 选举机制。那么 Leader 选举本质是要解决什么问题呢？

首先当然是要保证 Leader 的唯一性，确保集群不出现多个 Leader，才能保证业务逻辑准确性，也就是安全性（Safety）、互斥性。

其次是主节点故障后，备节点应可快速感知到其异常，也就是活性（liveness）检测。实现活性检测主要有两种方案。

方案一为被动型检测，你可以通过探测节点定时拨测 Leader 节点，看是否健康，比如 Redis Sentinel。

方案二为主动型上报，Leader 节点可定期向协调服务发送"特殊心跳"汇报健康状态，若其未正常发送心跳，并超过和协调服务约定的最大存活时间后，就会被协调服务移除 Leader 身份标识。同时其他节点可通过协调服务，快速感知到 Leader 故障了，进而发起新的选举。

我们今天的主题，Lease，正是基于主动型上报模式，**提供的一种活性检测机制**。Lease 顾名思义，client 和 etcd server 之间存在一个约定，内容是 etcd server 保证在约定的有效期内（TTL），不会删除你关联到此 Lease 上的 key-value。

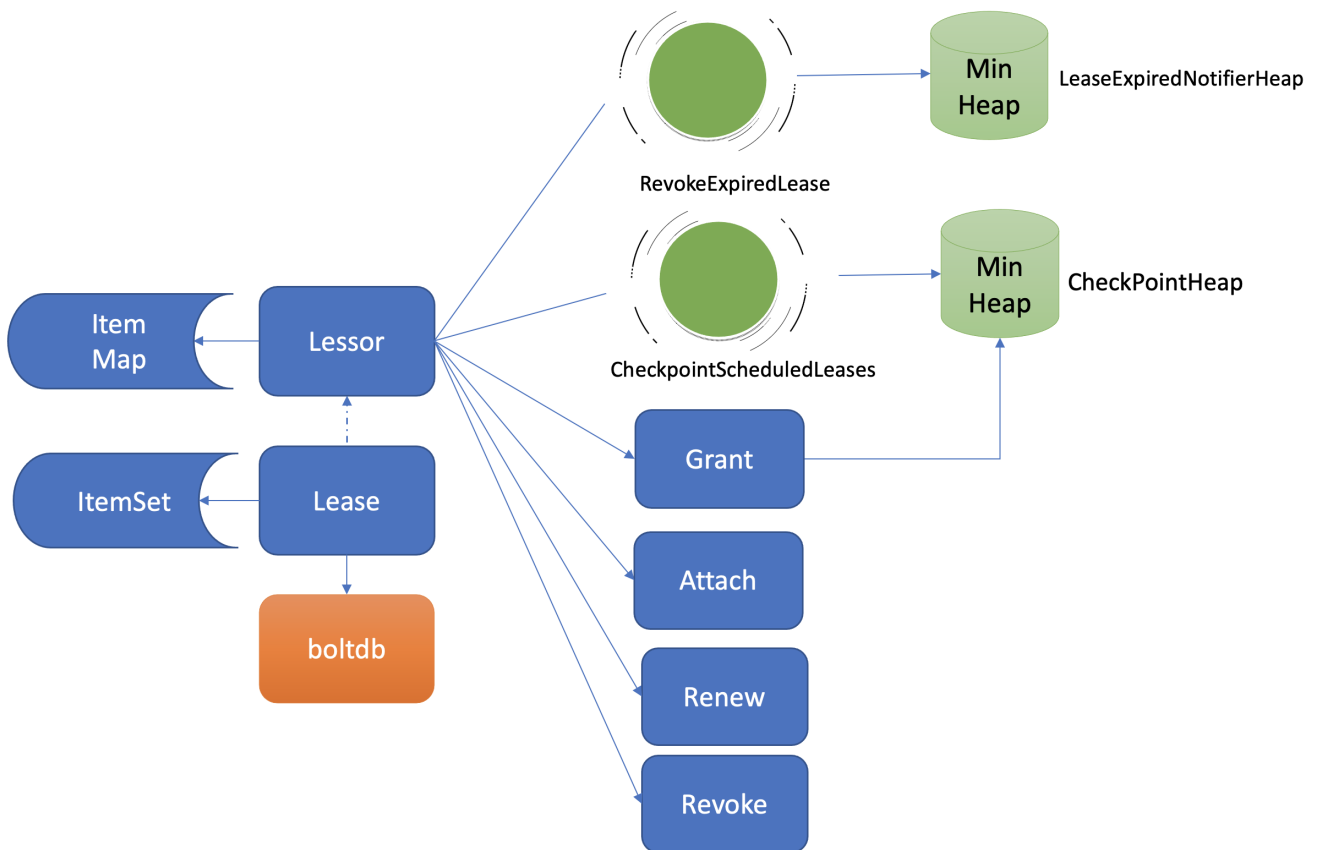
若你未在有效期内续租，那么 etcd server 就会删除 Lease 和其关联的 key-value。

你可以基于 Lease 的 TTL 特性，解决类似 Leader 选举、Kubernetes Event 自动淘汰、服务发现场景中故障节点自动剔除等问题。为了帮助你理解 Lease 的核心特性原理，我以一个实际场景中的经常遇到的异常节点自动剔除为案例，围绕这个问题，给你深入介绍 Lease 特性的实现。

在这个案例中，我们期望的效果是，在节点异常时，表示节点健康的 key 能被从 etcd 集群中自动删除。

Lease 整体架构

在和你详细解读 Lease 特性如何解决上面的问题之前，我们先了解下 Lease 模块的整体架构，下图是我给你画的 Lease 模块简要架构图。



etcd 在启动的时候，创建 Lessor 模块的时候，它会启动两个常驻 goroutine，如上图所示，一个是 **RevokeExpiredLease** 任务，定时检查是否有过期 Lease，发起撤销过期的 Lease 操作。一个是 **CheckpointScheduledLease**，定时触发更新 Lease 的剩余到期时间的操作。

Lessor 模块提供了 **Grant**、**Revoke**、**LeaseTimeToLive**、**LeaseKeepAlive** API 给 client 使用，各接口作用如下：

Grant 表示创建一个 TTL 为你指定秒数的 Lease，Lessor 会将 Lease 信息持久化存储在 **boltdb** 中；

Revoke 表示撤销 Lease 并删除其关联的数据；

LeaseTimeToLive 表示获取一个 Lease 的有效期、剩余时间；


LeaseKeepAlive 表示为 Lease 续期。

key 如何关联 Lease

了解完整体架构后，我们再看如何基于 Lease 特性实现检测一个节点存活。

首先如何为节点健康指标创建一个租约、并与节点健康指标 key 关联呢？

如 KV 模块的一样，client 可通过 clientv3 库的 Lease API 发起 RPC 调用，你可以使用如下的 etcdctl 命令为 node 的健康状态指标，创建一个 Lease，有效期为 600 秒。然后通过 timetolive 命令，查看 Lease 的有效期、剩余时间。

 复制代码


```
1 # 创建一个TTL为600秒的lease, etcd server返回LeaseID
2 $ etcdctl lease grant 600
3 lease 326975935f48f814 granted with TTL(600s)
4
5
6 # 查看lease的TTL、剩余时间
7 $ etcdctl lease timetolive 326975935f48f814
8 lease 326975935f48f814 granted with TTL(600s), remaining(590s)
```

当 Lease server 收到 client 的创建一个有效期 600 秒的 Lease 请求后，会通过 Raft 模块完成日志同步，随后 Apply 模块通过 Lessor 模块的 Grant 接口执行日志条目内容。

首先 Lessor 的 Grant 接口会把 Lease 保存到内存的 ItemMap 数据结构中，然后它需要持久化 Lease，将 Lease 数据保存到 boltdb 的 Lease bucket 中，返回一个唯一的 LeaseID 给 client。

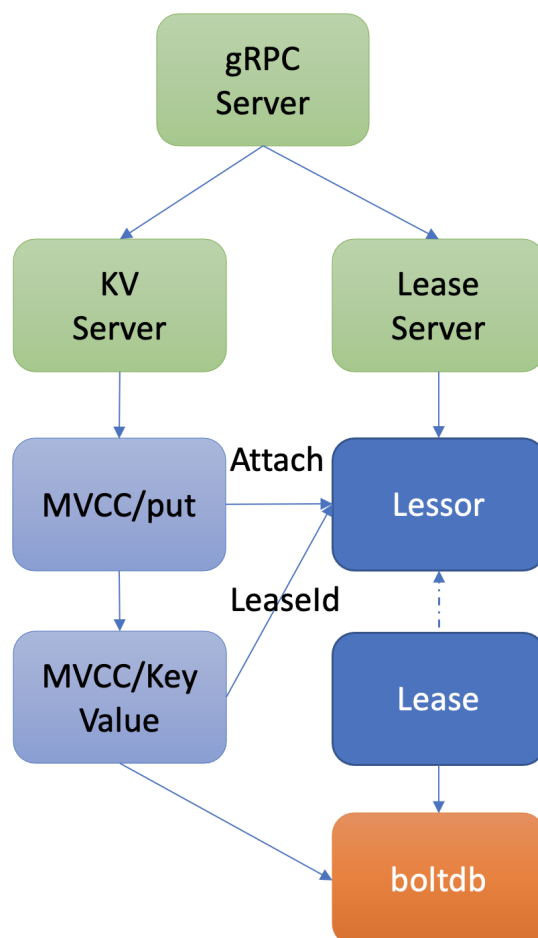
通过这样一个流程，就基本完成了 Lease 的创建。那么节点的健康指标数据如何关联到此 Lease 上呢？

很简单，KV 模块的 API 接口提供了一个"--lease"参数，你可以通过如下命令，将 key node 关联到对应的 LeaseID 上。然后你查询的时候增加 -w 参数输出格式为 json，就可查看到 key 关联的 LeaseID。

 复制代码

```
1 $ etcdctl put node healthy --lease 326975935f48f818
2 OK
3 $ etcdctl get node -w=json | python -m json.tool
4 {
5     "kvs":[
6         {
7             "create_revision":24,
8             "key":"bm9kZQ==",
9             "Lease":3632563850270275608,
10            "mod_revision":24,
11            "value":"aGVhbHRoeQ==",
12            "version":1
13        }
14    ]
15 }
```

以上流程原理如下图所示，它描述了用户的 key 是如何与指定 Lease 关联的。当你通过 put 等命令新增一个指定了"--lease"的 key 时，MVCC 模块它会通过 Lessor 模块的 Attach 方法，将 key 关联到 Lease 的 key 内存集合 ItemSet 中。



一个 Lease 关联的 key 集合是保存在内存中的，那么 etcd 重启时，是如何知道每个 Lease 上关联了哪些 key 呢？

答案是 etcd 的 MVCC 模块在持久化存储 key-value 的时候，保存到 boltdb 的 value 是个结构体（mvccpb.KeyValue），它不仅包含你的 key-value 数据，还包含了关联的 LeaseID 等信息。因此当 etcd 重启时，可根据此信息，重建关联各个 Lease 的 key 集合列表。

如何优化 Lease 续期性能

通过以上流程，我们完成了 Lease 创建和数据关联操作。在正常情况下，你的节点存活时，需要定期发送 KeepAlive 请求给 etcd 续期健康状态的 Lease，否则你的 Lease 和关联的数据就会被删除。

那么 Lease 是如何续期的？作为一个高频率的请求 API，etcd 如何优化 Lease 续期的性能呢？

Lease 续期其实很简单，核心是将 Lease 的过期时间更新为当前系统时间加其 TTL。关键问题在于续期的性能能否满足业务诉求。

然而影响续期性能因素又是源自多方面的。首先是 TTL，TTL 过长会导致节点异常后，无法及时从 etcd 中删除，影响服务可用性，而过短，则要求 client 频繁发送续期请求。其次是 Lease 数，如果 Lease 成千上万个，那么 etcd 可能无法支撑如此大规模的 Lease 数，导致高负载。

如何解决呢？

首先我们回顾下早期 etcd v2 版本是如何实现 TTL 特性的。在早期 v2 版本中，没有 Lease 概念，TTL 属性是在 key 上面，为了保证 key 不删除，即便你的 TTL 相同，client 也需要为每个 TTL、key 创建一个 HTTP/1.x 连接，定时发送续期请求给 etcd server。

很显然，v2 老版本这种设计，因不支持连接多路复用、相同 TTL 无法复用导致性能较差，无法支撑较大规模的 Lease 场景。

etcd v3 版本为了解决以上问题，提出了 Lease 特性，TTL 属性转移到了 Lease 上，同时协议从 HTTP/1.x 优化成 gRPC 协议。

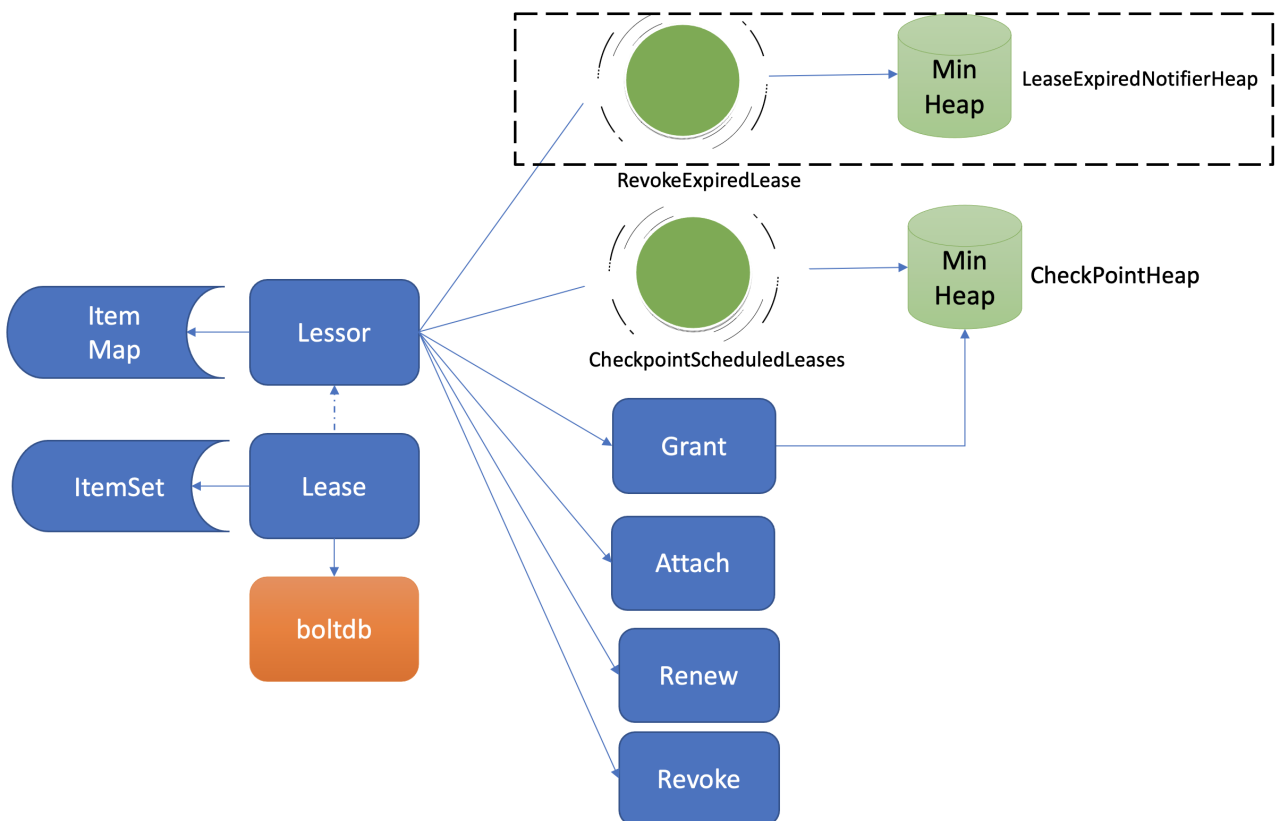
一方面不同 key 若 TTL 相同，可复用同一个 Lease，显著减少了 Lease 数。另一方面，通过 gRPC HTTP/2 实现了多路复用，流式传输，同一连接可支持为多个 Lease 续期，大大减少了连接数。

通过以上两个优化，实现 Lease 性能大幅提升，满足了各个业务场景诉求。

如何高效淘汰过期 Lease

在了解完节点正常情况下的 Lease 续期特性后，我们再看看节点异常时，未正常续期后，etcd 又是如何淘汰过期 Lease、删除节点健康指标 key 的。

淘汰过期 Lease 的工作由 Lessor 模块的一个异步 goroutine 负责。如下面架构图虚线框所示，它会定时从最小堆中取出已过期的 Lease，执行删除 Lease 和其关联的 key 列表数据的 RevokeExpiredLease 任务。



从图中你可以看到，目前 etcd 是基于最小堆来管理 Lease，实现快速淘汰过期的 Lease。

etcd 早期的时候，淘汰 Lease 非常暴力。etcd 会直接遍历所有 Lease，逐个检查 Lease 是否过期，过期则从 Lease 关联的 key 集合中，取出 key 列表，删除它们，时间复杂度是 $O(N)$ 。

然而这种方案随着 Lease 数增大，毫无疑问它的性能会变得越来越差。我们能否按过期时间排序呢？这样每次只需轮询、检查排在前面的 Lease 过期时间，一旦轮询到未过期的 Lease，则可结束本轮检查。

刚刚说的就是 etcd Lease 高效淘汰方案最小堆的实现方法。每次新增 Lease、续期的时候，它会插入、更新一个对象到最小堆中，对象含有 LeaseID 和其到期时间 unixnano，对象之间按到期时间升序排序。

etcd Lessor 主循环每隔 500ms 执行一次撤销 Lease 检查 (RevokeExpiredLease)，每次轮询堆顶的元素，若已过期则加入到待淘汰列表，直到堆顶的 Lease 过期时间大于当前，则结束本轮轮询。

相比早期 $O(N)$ 的遍历时间复杂度，使用堆后，插入、更新、删除，它的时间复杂度是 $O(\log N)$ ，查询堆顶对象是否过期时间复杂度仅为 $O(1)$ ，性能大大提升，可支撑大规模场景下 Lease 的高效淘汰。

获取到待过期的 LeaseID 后，Leader 是如何通知其他 Follower 节点淘汰它们呢？

Lessor 模块会将已确认过期的 LeaseID，保存在一个名为 expiredC 的 channel 中，而 etcd server 的主循环会定期从 channel 中获取 LeaseID，发起 revoke 请求，通过 Raft Log 传递给 Follower 节点。

各个节点收到 revoke Lease 请求后，获取关联到此 Lease 上的 key 列表，从 boltdb 中删除 key，从 Lessor 的 Lease map 内存中删除此 Lease 对象，最后还需要从 boltdb 的 Lease bucket 中删除这个 Lease。

以上就是 Lease 的过期自动淘汰逻辑。Leader 节点按过期时间维护了一个最小堆，若你的节点异常未正常续期，那么随着时间消逝，对应的 Lease 则会过期，Lessor 主循环定时轮询过期的 Lease。获取到 ID 后，Leader 发起 revoke 操作，通知整个集群删除 Lease 和关联的数据。

为什么需要 checkpoint 机制

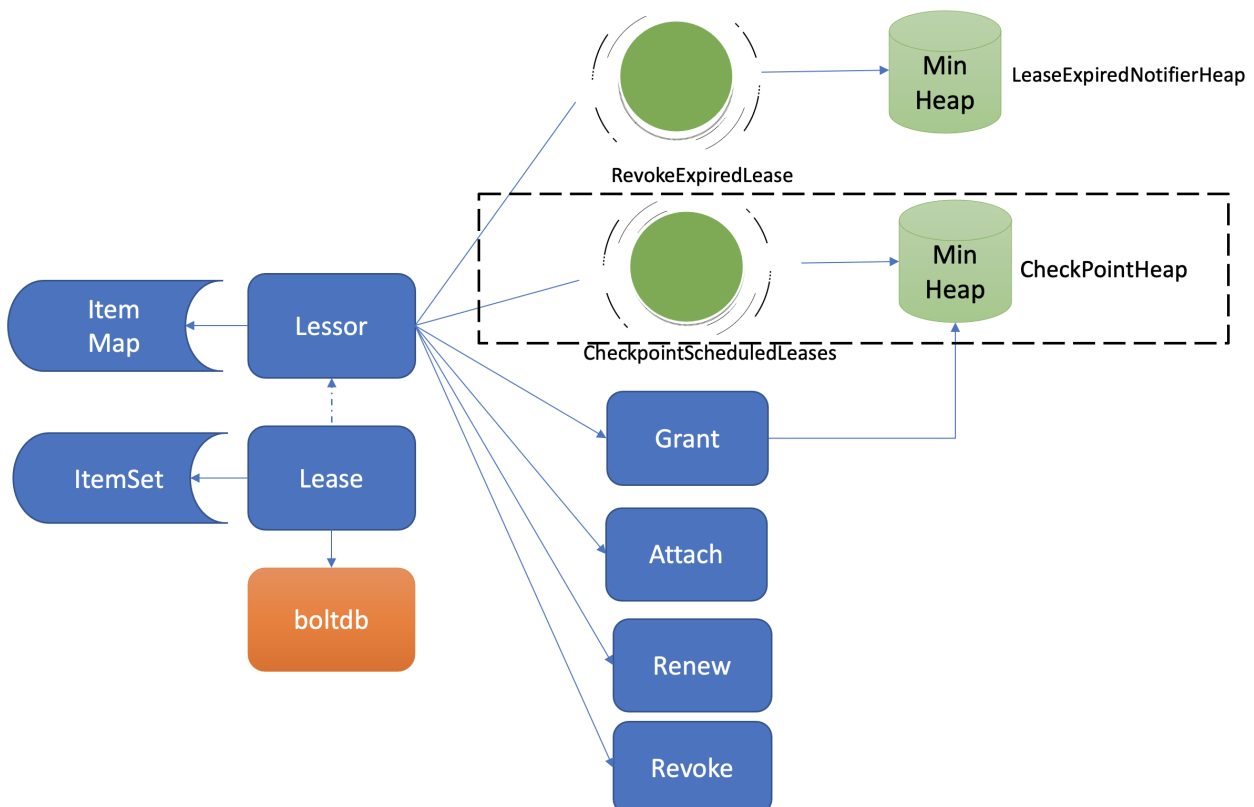
了解完 Lease 的创建、续期、自动淘汰机制后，你可能已经发现，检查 Lease 是否过期、维护最小堆、针对过期的 Lease 发起 revoke 操作，都是 Leader 节点负责的，它类似于 Lease 的仲裁者，通过以上清晰的权责划分，降低了 Lease 特性的实现复杂度。

那么当 Leader 因重启、crash、磁盘 IO 等异常不可用时，Follower 节点就会发起 Leader 选举，新 Leader 要完成以上职责，必须重建 Lease 过期最小堆等管理数据结构，那么以上重建可能会触发什么问题呢？

当你的集群发生 Leader 切换后，新的 Leader 基于 Lease map 信息，按 Lease 过期时间构建一个最小堆时，etcd 早期版本为了优化性能，并未持久化存储 Lease 剩余 TTL 信息，因此重建的时候就会自动给所有 Lease 自动续期了。

然而若较频繁出现 Leader 切换，切换时间小于 Lease 的 TTL，这会导致 Lease 永远无法删除，大量 key 堆积，db 大小超过配额等异常。

为了解决这个问题，etcd 引入了检查点机制，也就是下面架构图中黑色虚线框所示的 CheckPointScheduledLeases 的任务。



一方面，etcd 启动的时候，Leader 节点后台会运行此异步任务，定期批量地将 Lease 剩余的 TTL 基于 Raft Log 同步给 Follower 节点，Follower 节点收到 CheckPoint 请求后，更新内存数据结构 LeaseMap 的剩余 TTL 信息。

另一方面，当 Leader 节点收到 KeepAlive 请求的时候，它也会通过 checkpoint 机制把此 Lease 的剩余 TTL 重置，并同步给 Follower 节点，尽量确保续期后集群各个节点的 Lease 剩余 TTL 一致性。

最后你要注意的，此特性对性能有一定影响，目前仍然是试验特性。你可以通过 `experimental-enable-lease-checkpoint` 参数开启。

小结

最后我们来小结下今天的内容，我通过一个实际案例为你解读了 Lease 创建、关联 key、续期、淘汰、checkpoint 机制。

Lease 的核心是 TTL，当 Lease 的 TTL 过期时，它会自动删除其关联的 key-value 数据。

首先是 Lease 创建及续期。当你创建 Lease 时，etcd 会保存 Lease 信息到 boltdb 的 Lease bucket 中。为了防止 Lease 被淘汰，你需要定期发送 LeaseKeepAlive 请求给 etcd server 续期 Lease，本质是更新 Lease 的到期时间。

续期的核心挑战是性能，etcd 经历了从 TTL 属性在 key 上，到独立抽象出 Lease，支持多 key 复用相同 TTL，同时协议从 HTTP/1.x 优化成 gRPC 协议，支持多路连接复用，显著降低了 server 连接数等资源开销。

其次是 Lease 的淘汰机制，etcd 的 Lease 淘汰算法经历了从时间复杂度 $O(N)$ 到 $O(\log N)$ 的演进，核心是轮询最小堆的 Lease 是否过期，若过期生成 revoke 请求，它会清理 Lease 和其关联的数据。

最后我给你介绍了 Lease 的 checkpoint 机制，它是为了解决 Leader 异常情况下 TTL 自动被续期，可能导致 Lease 永不淘汰的问题而诞生。

思考题

好了，这节课到这里也就结束了，我最后给你留了一个思考题。你知道 etcd lease 最小的 TTL 时间是多少吗？它跟什么因素有关呢？

感谢你的阅读，如果你认为这节课的内容有收获，也欢迎把它分享给你的朋友，谢谢。

提建议

12.12 大促

每日一课 VIP 年卡

10分钟，解决你的技术难题

¥159/年 ¥365/年

每日一课
VIP 年卡

仅3天，【点击】图片，立即抢购 >>>

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 05 | 鉴权：如何保护你的数据安全？

下一篇 07 | MVCC：如何实现多版本并发控制？

精选留言 (7)

写留言



写点啥呢
2021-02-01

请教老师，对于Lease操作，请求是否必须有Leader接收处理。这种写请求路由是通过client3客户端直接发到leader还是通过可以通过follower转发？

谢谢

展开 ∨

作者回复: 非常好的问题，从原理上我们知道lease是leader在内存中维护过期最小堆的，因此续期操作client是必须要直接发送给leader的，如果follower节点收到了keepalive请求，会转发给leader节点。续期操作不经过raft协议处理同步，而leaseGrant/Revoke请求会经过raft协议同步给各个节点，因此任意节点都可以处理它。



1 评论

5 点赞



akai
2021-02-01

您好，有个疑问，关于这句话：“一方面不同 key 若 TTL 相同，可复用同一个 Lease，显著减少了 Lease 数。”

这句话其实不太理解，相同TTL的k-v可以用同一个lease,但lease过期，会删除所有k-v，是不是这里虽然是多个k-v,且TTL相同，但其实他们是有事务关系的，既要么都可用，要么都过期，并不是因为TTL相同而放在一个lease下面呢

展开 ∨

作者回复: 从实际使用场景上来，我认为是TTL几乎相同，为了降低etcd server的压力而把多个kv关联在一个lease上的，比如kubernetes场景中，有大量的event, 如果一个event,一个lease, lease数量是非常多的，lease过期会触发大量写请求，这对etcd server压力非常大，为了解决这个问题对etcd server性能的影响，lease过期淘汰会默认限速每秒1000个。因此kubernetes场景为了优化lease数，会将最近一分钟内产生的event key列表,复用在同一个lease,大大降低了lease数。



3 评论

4 点赞



mmm
2021-02-01

老师，您好，我有几个概念理解的不知是否准确：

- 1、淘汰过期租约的最小堆中保存的时间是租约到期时间的时间戳，对吧
- 2、checkpoint中说“按 Lease 过期时间构建一个最小堆时，...，并未持久化存储 Lease 剩余 TTL 信息，因此重建的时候就会自动给所有 Lease 自动续期了”，这里Lease过期时间是指Lease的租约时长600s这个概念，而不是上面1中所说的到期时间戳，对吗？...

展开 ∨

作者回复: 感谢你的精彩提问, 我分别回答下:

1. 淘汰过期lease最小堆中保存的时间是lease到期时间, 比如lease TTL是600秒/10分钟, 当前时间是00:00:00, 那么到期时间00:10:00。
- 2和3. checkpoint最小堆中保存的时间是定时触发lease剩余TTL的同步的间隔时间, 默认是每隔5分钟触发一次同步, 如果leader在00:05:00 crashed, 也没开启lease剩余TTL同步操作(还剩余5分钟), 那么新的leader重建后的租约时长又是10分钟了, 如果你开启checkpoint机制, 那么同步的就是lease剩余TTL(5分钟)。
4. 我个人认为同步剩余TTL有助于减少时钟不一致的情况下的导致各种问题, etcd和raft里面大量使用的都是逻辑时钟, 至于不同步会产生哪些问题, 我后续抽空测试验证下我的一些猜测, 也欢迎你自己测试下。

 1 3

一步

2021-02-03

lease 代表 过期的一个 ttl, 多个 key 复用同一个 lease 的时候, lease 是不是没有办法保存每个 key 的具体过期时间点是多少, 因为每个 key 的创建时间不一样, 所以过期时间也不一样。

还有就是当多个 key 复用同一个 lease 的时候, 某个客户端再发送 keepalive 请求的时...
展开 ∨

作者回复: 第一个问题, 一般情况下不要求每个key过期时间完全一致, 比如kubernetes的event, 就是误差1分钟内的event key,可以复用同一个lease.
第二个问题, keepalive请求就是更新lease在最小堆中的过期时间(now + ttl), 可简单理解为关联到此lease上的所有key ttl都延长了。

 1

mckee

2021-02-03

思考题: etcd lease 最小的 TTL 是多少?

可能为了确保发生leader选举时, lease不会过期, 最小ttl应该比选举时间长, 看代码

```
minTTL := time.Duration((3*cfg.ElectionTicks)/2) * heartbeat
minTTLSec := int64(math.Ceil(minTTL.Seconds()))
```

默认的情况下应该是2s,

展开 ∨

 1

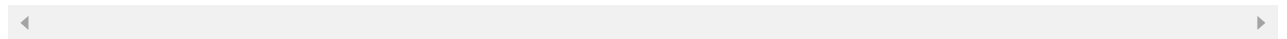
**Geek_5a8405**

2021-02-15

续期操作不通过raft协议同步到follower，那如果读带lease的key是不是得经过leader处理？因为只有leader的lease过期时间是最准确的（虽然会定时checkpoint同步ttl到follower，但是我理解这个不是非常准确到）。

展开 ✓

作者回复: 不需要经过leader处理哈，etcd对过期时间要求没那么严格，不需要精准到毫秒级。如果lease关联的key过期了，leader会立刻发送撤销租约请求给follower，正常etcd负载情况下，这个请求同步到follower延时大概是毫秒级的，高负载、磁盘IO异常等情况下，的确可能出现比较大的延迟。

**zjc**

2021-02-07

leader和follower节点所在机器上的时间不同步(如差5s),leader故障后follower升为leader,这种情况会影响lease吗？

展开 ✓

