16 | 性能及稳定性（上）：如何优化及扩展etcd性能?

2021-02-24 唐聪

etcd实战课 进入课程 >

讲述：王超凡
时长 18:16 大小 16.74M

你好，我是唐聪。

在使用 etcd 的过程中，你是否吐槽过 etcd 性能差呢？ 我们知道，etcd 社区线性读压测结果可以达到 14w/s，那为什么在实际业务场景中有时却只有几千，甚至几百、几十，还会偶发超时、频繁抖动呢？

我相信不少人都遇到过类似的问题。要解决这些问题，不仅需要了解症结所在，还需要掌握优化和扩展 etcd 性能的方法，对症下药。因为这部分内容比较多，所以我分成了两讲内容，分别从读性能、写性能和稳定性入手，为你详细讲解如何优化及扩展 etcd 性能及稳定性。

希望通过这两节课的学习，能让你在使用 etcd 的时候，设计出良好的业务存储结构，遵循最佳实践，让 etcd 稳定、高效地运行，获得符合预期的性能。同时，当你面对 etcd 性能瓶颈的时候，也能自己分析瓶颈原因、选择合适的优化方案解决它，而不是盲目甩锅 etcd，甚至更换技术方案去 etcd 化。

今天这节课，我将重点为你介绍如何提升读的性能。

我们说读性能差，其实本质是读请求链路中某些环节出现了瓶颈。所以，接下来我将通过一张读性能分析链路图，为你从上至下分析影响 etcd 性能、稳定性的若干因素，并给出相应的压测数据，最终为你总结出一系列的 etcd 性能优化和扩展方法。

性能分析链路

为什么在你的业务场景中读性能不如预期呢？ 是读流程中的哪一个环节出现了瓶颈？

在下图中，我为你总结了一个开启密码鉴权场景的读性能瓶颈分析链路图，并在每个核心步骤数字旁边，标出了影响性能的关键因素。我之所以选用密码鉴权的读请求为案例，是因为它使用较广泛并且请求链路覆盖最全，同时它也是最容易遇到性能瓶颈的场景。

接下来我将按照这张链路分析图，带你深入分析一个使用密码鉴权的线性读请求，和你一起看看影响它性能表现的核心因素以及最佳优化实践。

负载均衡

首先是流程一负载均衡。在02 节时我和你提到过，在 etcd 3.4 以前，client 为了节省与 server 节点的连接数，clientv3 负载均衡器最终只会选择一个 sever 节点 IP，与其建立一个长连接。

但是这可能会导致对应的 server 节点过载（如单节点流量过大，出现丢包），其他节点却是低负载，最终导致业务无法获得集群的最佳性能。在 etcd 3.4 后，引入了 Round-robin 负载均衡算法，它通过轮询的方式依次从 endpoint 列表中选择一个 endpoint 访问 (长连接)，使 server 节点负载尽量均衡。

所以，如果你使用的是 etcd 低版本，那么我建议你通过 Load Balancer 访问后端 etcd 集群。因为一方面 Load Balancer 一般支持配置各种负载均衡算法，如连接数、Round-robin 等，可以使你的集群负载更加均衡，规避 etcd client 早期的固定连接缺陷，获得集群最佳性能。

另一方面，当你集群节点需要替换、扩缩容集群节点的时候，你不需要去调整各个 client 访问 server 的节点配置。

选择合适的鉴权

client 通过负载均衡算法为请求选择好 etcd server 节点后，client 就可调用 server 的 Range RPC 方法，把请求发送给 etcd server。在此过程中，如果 server 启用了鉴权，那么就会返回无权限相关错误给 client。

如果 server 使用的是密码鉴权，你在创建 client 时，需指定用户名和密码。etcd clientv3 库发现用户名、密码非空，就会先校验用户名和密码是否正确。

client 是如何向 sever 请求校验用户名、密码正确性的呢？

client 是通过向 server 发送 Authenticate RPC 鉴权请求实现密码认证的，也就是图中的流程二。

根据我们05介绍的密码认证原理，server 节点收到鉴权请求后，它会从 boltdb 获取此用户密码对应的算法版本、salt、cost 值，并基于用户的请求明文密码计算出一个 hash 值。

在得到 hash 值后，就可以对比 db 里保存的 hash 密码是否与其一致了。如果一致，就会返回一个 token 给 client。 这个 token 是 client 访问 server 节点的通行证，后续 server 只需要校验“通行证”是否有效即可，无需每次发起昂贵的 Authenticate RPC 请求。

讲到这里，不知道你有没有意识到，若你的业务在访问 etcd 过程中未复用 token，每次访问 etcd 都发起一次 Authenticate 调用，这将是一个非常大的性能瓶颈和隐患。因为正如我们 05 所介绍的，为了保证密码的安全性，密码认证（Authenticate）的开销非常昂贵，涉及到大量 CPU 资源。

那这个 Authenticate 接口究竟有多慢呢?

为了得到 Authenticate 接口的性能，我们做过这样一个测试：

压测集群 etcd 节点配置是 16 核 32G；

压测方式是我们通过修改 etcd clientv3 库、benchmark 工具，使 benchmark 工具支持 Authenticate 接口压测；

然后设置不同的 client 和 connection 参数，运行多次，观察结果是否稳定，获取测试结果。

最终的测试结果非常惊人。etcd v3.4.9 之前的版本，Authenticate 接口性能不到 16 QPS，并且随着 client 和 connection 增多，该性能会继续恶化。

当 client 和 connection 的数量达到 200 个的时候，性能会下降到 8 QPS，P99 延时为 18 秒，如下图所示。

对此，我和小伙伴王超凡通过一个减少锁的范围 PR（该 PR 已经 cherry-pick 到了 etcd 3.4.9 版本），将性能优化到了约 200 QPS，并且 P99 延时在 1 秒内，如下图所示。

由于导致 Authenticate 接口性能差的核心瓶颈，是在于密码鉴权使用了 bcrpt 计算 hash 值，因此 Authenticate 性能已接近极限。

最令人头疼的是，Auenticate 的调用由 clientv3 库默默发起的，etcd 中也没有任何日志记录其耗时等。当大家开启密码鉴权后，遇到读写接口超时的时候，未详细了解 etcd 的同学就会非常困惑，很难定位超时本质原因。

我曾多次收到小伙伴的求助，协助他们排查 etcd 异常超时问题。通过 metrics 定位，我发现这些问题大都是由比较频繁的 Authenticate 调用导致，只要临时关闭鉴权或升级到 etcd v3.4.9 版本就可以恢复。

为了帮助大家快速发现 Authenticate 等特殊类型的 expensive request，我在 etcd 3.5 版本中提交了一个 PR，通过 gRPC 拦截器的机制，当一个请求超过 300ms 时，就会打印整个请求信息。

讲到这里，你应该会有疑问，密码鉴权的性能如此差，可是业务又需要使用它，我们该怎么解决密码鉴权的性能问题呢？对此，我有三点建议。

第一，如果你的生产环境需要开启鉴权，并且读写 QPS 较大，那我建议你不要图省事使用密码鉴权。最好使用证书鉴权，这样能完美避坑认证性能差、token 过期等问题，性能几乎无损失。

第二，确保你的业务每次发起请求时有复用 token 机制，尽可能减少 Authenticate RPC 调用。

第三，如果你使用密码鉴权时遇到性能瓶颈问题，可将 etcd 升级到 3.4.9 及以上版本，能适当提升密码鉴权的性能。

选择合适的读模式

client 通过 server 的鉴权后，就可以发起读请求调用了，也就是我们图中的流程三。

在这个步骤中，读模式对性能有着至关重要的影响。我们前面讲过 etcd 提供了串行读和线性读两种读模式。前者因为不经过 ReadIndex 模块，具有低延时、高吞吐量的特点；而后者在牺牲一点延时和吞吐量的基础上，实现了数据的强一致性读。这两种读模式分别为不同场景的读提供了解决方案。

关于串行读和线性读的性能对比，下图我给出了一个测试结果，测试环境如下：

机器配置 client 16 核 32G，三个 server 节点 8 核 16G、SSD 盘，client 与 server 节点都在同可用区；

各节点之间 RTT 在 0.1ms 到 0.2ms 之间；

etcd v3.4.9 版本；

1000 个 client。

执行如下串行读压测命令：

复制代码

```
benchmark --endpoints=addr --conns=100 --clients=1000 \
range hello --consistency=s --total=500000
```

得到串行读压测结果如下，32 万 QPS，平均延时 2.5ms。

执行如下线性读压测命令：

复制代码

```
benchmark --endpoints=addr --conns=100 --clients=1000 \
range hello --consistency=l --total=500000
```

得到线性读压测结果如下，19 万 QPS，平均延时 4.9ms。

从两个压测结果图中你可以看到，在 100 个连接时，串行读性能比线性读性能高近 11 万/s，串行读请求延时（2.5ms）比线性读延时约低一半（4.9ms）。

需要注意的是，以上读性能数据是在 1 个 key、没有任何写请求、同可用区的场景下压测出来的，实际的读性能会随着你的写请求增多而出现显著下降，这也是实际业务场景性能与社区压测结果存在非常大差距的原因之一。所以，我建议你使用 etcd benchmark 工具在你的 etcd 集群环境中自测一下，你也可以参考下面的etcd 社区压测结果。

如果你的业务场景读 QPS 较大，但是你又不想通过 etcd proxy 等机制来扩展性能，那你可以进一步评估业务场景对数据一致性的要求高不高。如果你可以容忍短暂的不一致，那你可以通过串行读来提升 etcd 的读性能，也可以部署 Learner 节点给可能会产生 expensive read request 的业务使用，实现 cheap/expensive read request 隔离。

线性读实现机制、网络延时

了解完读模式对性能的影响后，我们继续往下分析。在我们这个密码鉴权读请求的性能分析案例中，读请求使用的是 etcd 默认线性读模式。线性读对应图中的流程四、流程五，其中流程四对应的是 ReadIndex，流程五对应的是等待本节点数据追上 Leader 的进度（ApplyWait）。

在早期的 etcd 3.0 版本中，etcd 线性读是基于 Raft log read 实现的。每次读请求要像写请求一样，生成一个 Raft 日志条目，然后提交给 Raft 一致性模块处理，基于 Raft 日志执行的有序性来实现线性读。因为该过程需要经过磁盘 I/O，所以性能较差。

为了解决 Raft log read 的线性读性能瓶颈，etcd 3.1 中引入了 ReadIndex。ReadIndex 仅涉及到各个节点之间网络通信，因此节点之间的 RTT 延时对其性能有较大影响。虽然同可用区可获取到最佳性能，但是存在单可用区故障风险。如果你想实现高可用区容灾的话，那就必须牺牲一点性能了。

跨可用区部署时，各个可用区之间延时一般在 2 毫秒内。如果跨城部署，服务性能就会下降较大。所以一般场景下我不建议你跨城部署，你可以通过 Learner 节点实现异地容灾。如果异地的服务对数据一致性要求不高，那么你甚至可以通过串行读访问 Learner 节点，来实现就近访问，低延时。

各个节点之间的 RTT 延时，是决定流程四 ReadIndex 性能的核心因素之一。

磁盘 IO 性能、写 QPS

到了流程五，影响性能的核心因素就是磁盘 IO 延时和写 QPS。

如下面代码所示，流程五是指节点从 Leader 获取到最新已提交的日志条目索引 (rs.Index) 后，它需要等待本节点当前已应用的 Raft 日志索引，大于等于 Leader 的已提交索引，确保能在本节点状态机中读取到最新数据。

复制代码

```
if ai := s.getAppliedIndex(); ai < rs.Index {
    select {
    case <-s.applyWait.Wait(rs.Index):
    case <-s.stopping:
        return
    }
}
// unblock all l-reads requested at indices before rs.Index
nr.notify(nil)
```

而应用已提交日志条目到状态机的过程中又涉及到随机写磁盘，详情可参考我们03中介绍过 etcd 的写请求原理。

因此我们可以知道，**etcd 是一个对磁盘 IO 性能非常敏感的存储系统，磁盘 IO 性能不仅会影响 Leader 稳定性、写性能表现，还会影响读性能。线性读性能会随着写性能的增加而快速下降。如果业务对性能、稳定性有较大要求，我建议你尽量使用 SSD 盘。**

下表我给出了一个 8 核 16G 的三节点集群，在总 key 数只有一个的情况下，随着写请求增大，线性读性能下降的趋势总结（基于 benchmark 工具压测结果），你可以直观感受下读性能是如何随着写性能下降。

当本节点已应用日志条目索引大于等于 Leader 已提交的日志条目索引后，读请求就会接到通知，就可通过 MVCC 模块获取数据。

RBAC 规则数、Auth 锁

读请求到了 MVCC 模块后，首先要通过鉴权模块判断此用户是否有权限访问请求的数据路径，也就是流程六。影响流程六的性能因素是你的 RBAC 规则数和锁。

首先是 RBAC 规则数，为了解决快速判断用户对指定 key 范围是否有权限，etcd 为每个用户维护了读写权限区间树。基于区间树判断用户访问的范围是否在用户的读写权限区间

内，时间复杂度仅需要 O(logN)。

另外一个因素则是 AuthStore 的锁。在 etcd 3.4.9 之前的，校验密码接口可能会占用较长时间的锁，导致授权接口阻塞。etcd 3.4.9 之后合入了缩小锁范围的 PR，可一定程度降低授权接口被阻塞的问题。

expensive request、treeIndex 锁

通过流程六的授权后，则进入流程七，从 treeIndex 中获取整个查询涉及的 key 列表版本号信息。在这个流程中，影响其性能的关键因素是 treeIndex 的总 key 数、查询的 key 数、获取 treeIndex 锁的耗时。

首先，treeIndex 中总 key 数过多会适当增大我们遍历的耗时。

其次，若要访问 treeIndex 我们必须获取到锁，但是可能其他请求如 compact 操作也会获取锁。早期的时候，它需要遍历所有索引，然后进行数据压缩工作。这就会导致其他请求阻塞，进而增大延时。

为了解决这个性能问题，优化方案是 compact 的时候会将 treeIndex 克隆一份，以空间来换时间，尽量降低锁阻塞带来的超时问题。

接下来我重点给你介绍下查询 key 数较多等 expensive read request 时对性能的影响。

假设我们链路分析图中的请求是查询一个 Kubernetes 集群所有 Pod，当你 Pod 数一百以内的时候可能对 etcd 影响不大，但是当你 Pod 数千甚至上万的时候， 流程七、八就会遍历大量的 key，导致请求耗时突增、内存上涨、性能急剧下降。你可结合 13db 大小、14延时、15内存三节一起看看，这里我就不再重复描述。

如果业务就是有这种 expensive read request 逻辑，我们该如何应对呢?

首先我们可以尽量减少 expensive read request 次数，在程序启动的时候，只 List 一次全量数据，然后通过 etcd Watch 机制去获取增量变更数据。比如 Kubernetes 的 Informer 机制，就是典型的优化实践。

其次，在设计上评估是否能进行一些数据分片、拆分等，不同场景使用不同的 etcd prefix 前缀。比如在 Kubernetes 中，不要把 Pod 全部都部署在 default 命名空间下，尽量根据业务场景按命名空间拆分部署。即便每个场景全量拉取，也只需要遍历自己命名空间下的资源，数据量上将下降一个数量级。

再次，如果你觉得 Watch 改造大、数据也无法分片，开发麻烦，你可以通过分页机制按批拉取，尽量减少一次性拉取数万条数据。

最后，如果以上方式都不起作用的话，你还可以通过引入 cache 实现缓存 expensive read request 的结果，不过应用需维护缓存数据与 etcd 的一致性。

大 key-value、boltdb 锁

从流程七获取到 key 列表及版本号信息后，我们就可以访问 boltdb 模块，获取 key-value 信息了。在这个流程中，影响其性能表现的，除了我们上面介绍的 expensive read request，还有大 key-value 和锁。

首先是大 key-value。我们知道 etcd 设计上定位是个小型的元数据存储，它没有数据分片机制，默认 db quota 只有 2G，实践中往往不会超过 8G，并且针对每个 key-value 大小，它也进行了大小限制，默认是 1.5MB。

大 key-value 非常容易导致 etcd OOM、server 节点出现丢包、性能急剧下降等。

那么当我们往 etcd 集群写入一个 1MB 的 key-value 时，它的线性读性能会从 17 万 QPS 具体下降到多少呢?

我们可以执行如下 benchmark 命令：

复制代码

```
benchmark --endpoints=addr --conns=100 --clients=1000 \
range key --consistency=l --total=10000
```

得到其结果如下，从下图你可以看到，读取一个 1MB 的 key-value，线性读性能 QPS 下降到 1163，平均延时上升到 818ms，可见大 key-value 对性能的巨大影响。

同时，从下面的 etcd 监控图上你也可以看到内存出现了突增，若存在大量大 key-value 时，可想而知，etcd 内存肯定暴涨，大概率会 OOM。

其次是锁，etcd 为了提升 boltdb 读的性能，从 etcd 3.1 到 etcd 3.4 版本，分别进行过几次重大优化，在下一节中我将和你介绍。

以上就是一个开启密码鉴权场景，线性读请求的性能瓶颈分析过程。

小结

今天我通过从上至下的请求流程分析，介绍了各个流程中可能存在的瓶颈和优化方法、最佳实践等。

优化读性能的核心思路是首先我们可通过 etcd clientv3 自带的 Round-robin 负载均衡算法或者 Load Balancer，尽量确保整个集群负载均衡。

然后，在开启鉴权场景时，建议你尽量使用证书而不是密码认证，避免校验密码的昂贵开销。

其次，根据业务场景选择合适的读模式，串行读比线性度性能提高 30% 以上，延时降低一倍。线性读性能受节点之间 RTT 延时、磁盘 IO 延时、当前写 QPS 等多重因素影响。

最容易被大家忽视的就是写 QPS 对读 QPS 的影响，我通过一系列压测数据，整理成一个表格，让你更直观感受写 QPS 对读性能的影响。多可用区部署会导致节点 RTT 延时增高，读性能下降。因此你需要在高可用和高性能上做取舍和平衡。

最后在访问数据前，你的读性能还可能会受授权性能、expensive read request、treeIndex 及 boltdb 的锁等影响。你需要遵循最佳实践，避免一个请求查询大量 key、大 key-value 等，否则会导致读性能剧烈下降。

希望你通过本文当遇到读 etcd 性能问题时，能从请求执行链路去分析瓶颈，解决问题，让业务和 etcd 跑得更稳、更快。

思考题

你在使用 etcd 过程中遇到了哪些读性能问题？又是如何解决的呢？

欢迎分享你的性能优化经历，感谢你阅读，也欢迎你把这篇文章分享给更多的朋友一起阅读。

提建议

上一篇 15 | 内存：为什么你的etcd内存占用那么高？

下一篇 17 | 性能及稳定性（下）：如何优化及扩展etcd性能？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。