



下载APP



17 | 性能及稳定性（下）：如何优化及扩展etcd性能？

2021-02-26 唐聪

etcd实战课

进入课程 >



讲述：王超凡

时长 17:12 大小 15.77M



你好，我是唐聪。

我们继续来看如何优化及扩展 etcd 性能。上一节课里我为你重点讲述了如何提升读的性能，今天我将重点为你介绍如何提升写性能和稳定性，以及如何基于 etcd gRPC Proxy 扩展 etcd 性能。

当你使用 etcd 写入大量 key-value 数据的时候，是否遇到过 etcd server 返回 "etcdserver: too many requests" 错误？这个错误是怎么产生的呢？我们又该如何优化写性能呢？

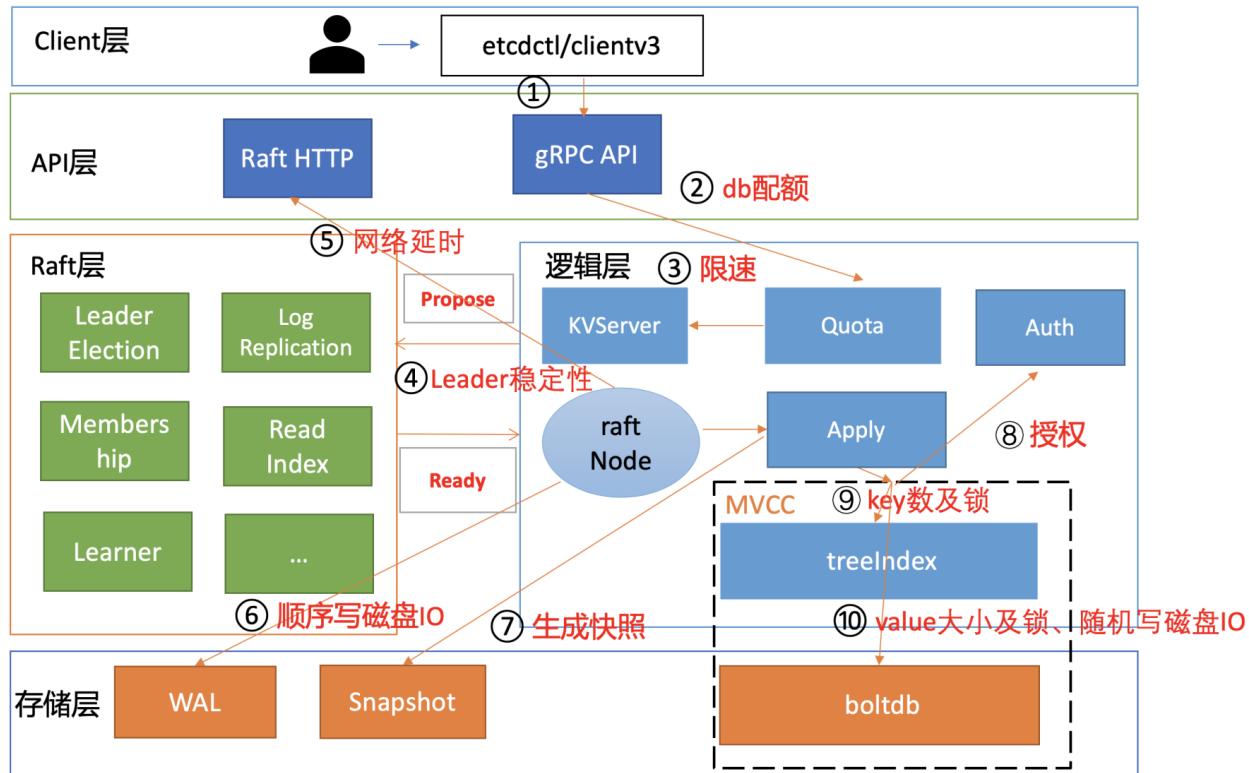


这节课我将通过写性能分析链路图，为你从上至下分析影响写性能、稳定性的若干因素，并为你总结出若干 etcd 写性能优化和扩展方法。

性能分析链路

为什么你写入大量 key-value 数据的时候，会遇到 Too Many Request 限速错误呢？是写流程中的哪些环节出现了瓶颈？

和读请求类似，我为你总结了一个开启鉴权场景的写性能瓶颈及稳定性分析链路图，并在每个核心步骤数字旁边标识了影响性能、稳定性的影响因素。

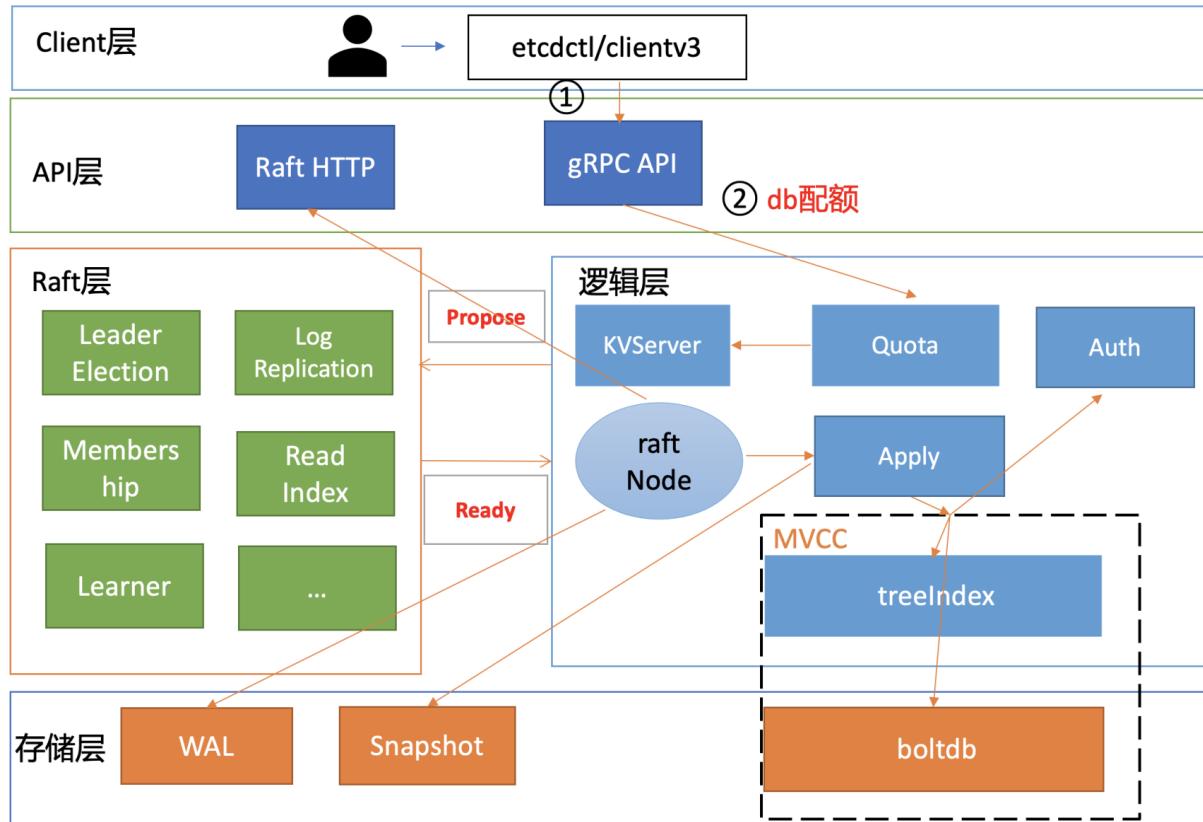


下面我将按照这个写请求链路分析图，和你深入分析影响 etcd 写性能的核心因素和最佳优化实践。

db quota

首先是流程一。在 etcd v3.4.9 版本中，client 会通过 clientv3 库的 Round-robin 负载均衡算法，从 endpoint 列表中轮询选择一个 endpoint 访问，发起 gRPC 调用。

然后进入流程二。etcd 收到 gRPC 写请求后，首先经过的是 Quota 模块，它会影响写请求的稳定性，若 db 大小超过配额就无法写入。



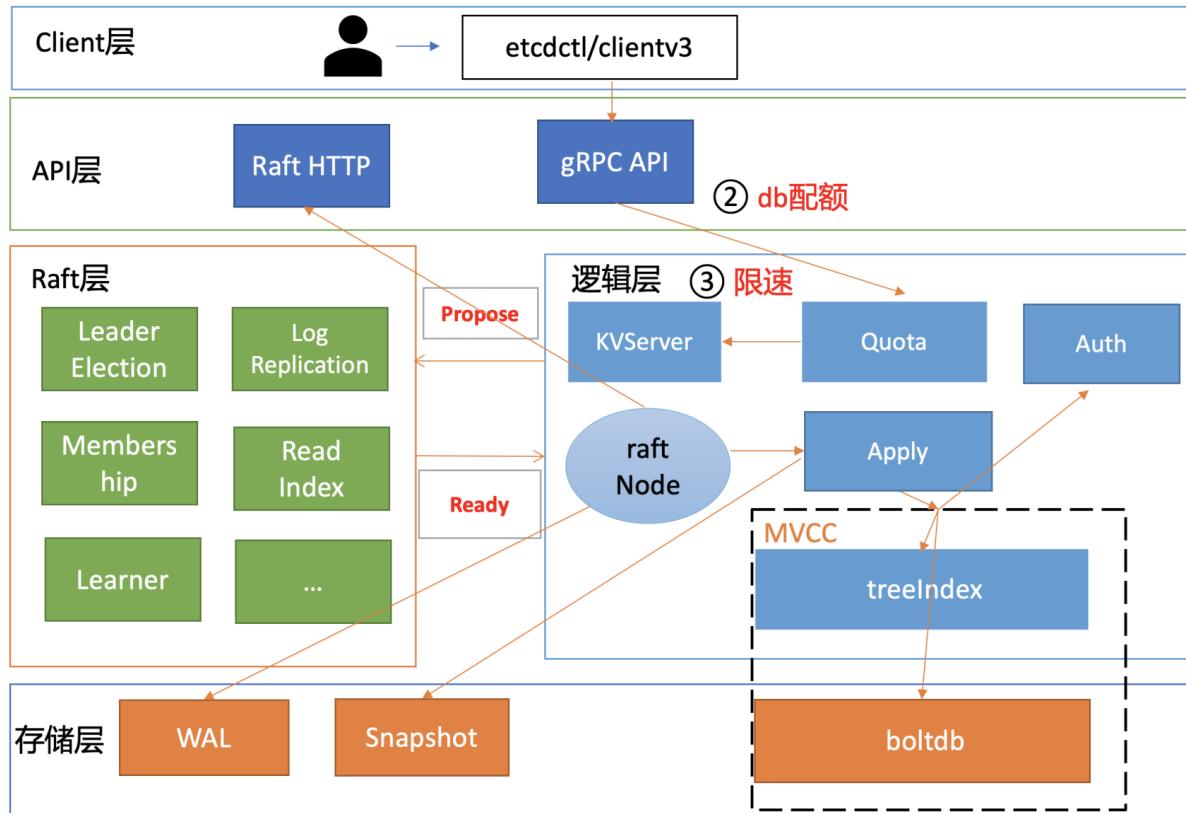
etcd 是个小型的元数据存储，默认 db quota 大小是 2G，超过 2G 就只读无法写入。因此你需要根据你的业务场景，适当调整 db quota 大小，并配置的合适的压缩策略。

正如我在 [11](#) 里和你介绍的，etcd 支持按时间周期性压缩、按版本号压缩两种策略，建议压缩策略不要配置得过于频繁。比如如果按时间周期压缩，一般情况下 5 分钟以上压缩一次比较合适，因为压缩过程中会加一系列锁和删除 boltdb 数据，过于频繁的压缩会对性能有一定的影响。

一般情况下 db 大小尽量不要超过 8G，过大的 db 文件和数据量对集群稳定性各方面都会有一定的影响，详细你可以参考 [13](#)。

限速

通过流程二的 Quota 模块后，请求就进入流程三 KVServer 模块。在 KVServer 模块里，影响写性能的核心因素是限速。



KVServer 模块的写请求在提交到 Raft 模块前，会进行限速判断。如果 Raft 模块已提交的日志索引 (committed index) 比已应用到状态机的日志索引 (applied index) 超过了 5000，那么它就返回一个"etcdserver: too many requests"错误给 client。

那么哪些情况可能会导致 committed Index 远大于 applied index 呢？

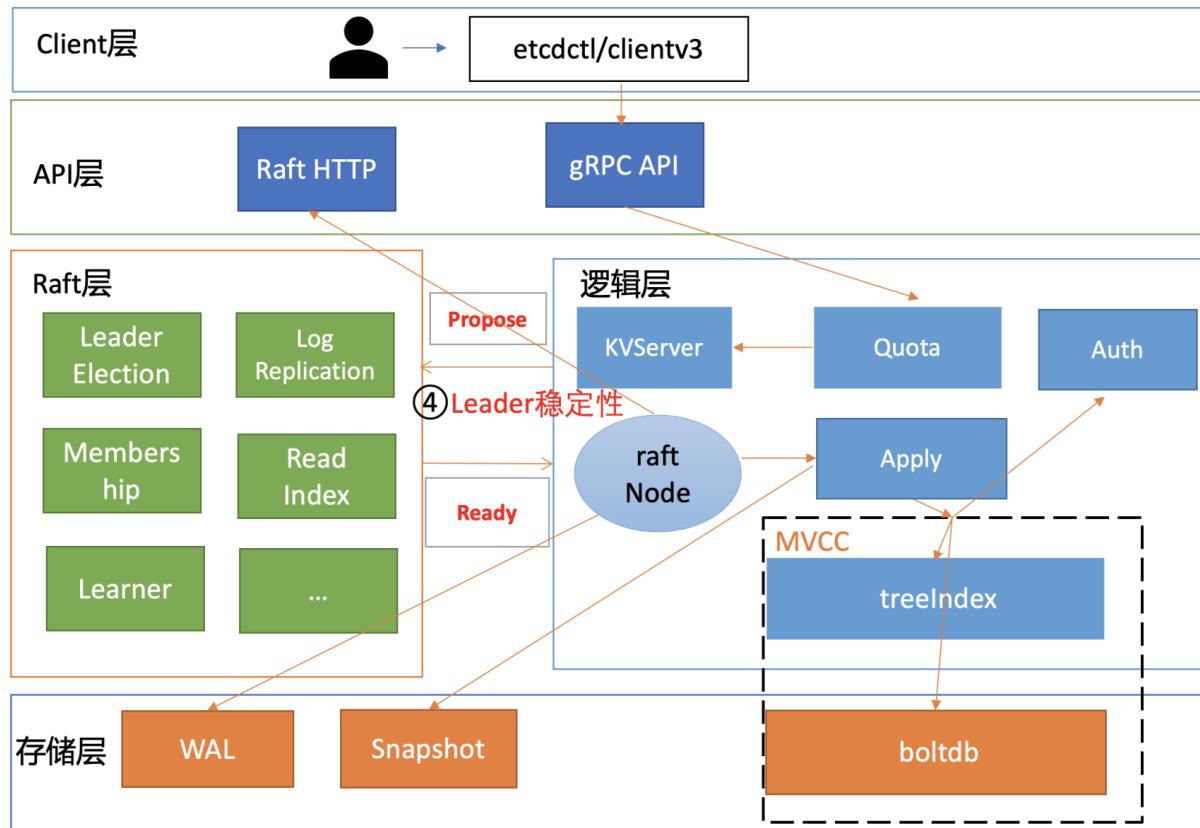
首先是 long expensive read request 导致写阻塞。比如 etcd 3.4 版本之前长读事务会持有较长时间的 buffer 读锁，而写事务又需要升级锁更新 buffer，因此出现写阻塞乃至超时。最终导致 etcd server 应用已提交的 Raft 日志命令到状态机缓慢。堆积过多时，则会触发限速。

其次 etcd 定时批量将 boltdb 写事务提交的时候，需要对 B+ tree 进行重平衡、分裂，并将 freelist、dirty page、meta page 持久化到磁盘。此过程需要持有 boltdb 事务锁，若磁盘随机写性能较差、瞬间大量写入，则也容易写阻塞，应用已提交的日志条目缓慢。

最后执行 defrag 等运维操作时，也会导致写阻塞，它们会持有相关锁，导致写性能下降。

心跳及选举参数优化

写请求经过 KVServer 模块后，则会提交到流程四的 Raft 模块。我们知道 etcd 写请求需要转发给 Leader 处理，因此影响此模块性能和稳定性的核心因素之一是集群 Leader 的稳定性。



那如何判断 Leader 的稳定性呢？

答案是日志和 metrics。

一方面，在你使用 etcd 过程中，你很可能见过如下 Leader 发送心跳超时的警告日志，你可以通过此日志判断集群是否有频繁切换 Leader 的风险。

另一方面，你可以通过 `etcd_server_leader_changes_seen_total` metrics 来观察已发生 Leader 切换的次数。

复制代码

```
1 21:30:27 etcd3 | {"level":"warn","ts":"2021-02-23T21:30:27.255+0800","caller":  
2 21:30:30 etcd3 | {"level":"warn","ts":"2021-02-23T21:30:30.396+0800","caller":
```

那么哪些因素会导致此日志产生以及发生 Leader 切换呢？

首先，我们知道 etcd 是基于 Raft 协议实现数据复制和高可用的，各节点会选出一个 Leader，然后 Leader 将写请求同步给各个 Follower 节点。而 Follower 节点如何感知 Leader 异常，发起选举，正是依赖 Leader 的心跳机制。

在 etcd 中，Leader 节点会根据 heartbeat-interval 参数（默认 100ms）定时向 Follower 节点发送心跳。如果两次发送心跳间隔超过 $2 * \text{heartbeat-interval}$ ，就会打印此警告日志。超过 election timeout（默认 1000ms），Follower 节点就会发起新一轮的 Leader 选举。

哪些原因会导致心跳超时呢？

一方面可能是你的磁盘 IO 比较慢。因为 etcd 从 Raft 的 Ready 结构获取到相关待提交日志条目后，它需要将此消息写入到 WAL 日志中持久化。你可以通过观察 etcd_wal_fsync_durations_seconds_bucket 指标来确定写 WAL 日志的延时。若延时较大，你可以使用 SSD 硬盘解决。

另一方面也可能是 CPU 使用率过高和网络延时过大导致。CPU 使用率较高可能导致发送心跳的 goroutine 出现饥饿。若 etcd 集群跨地域部署，节点之间 RTT 延时大，也可能会导致此问题。

最后我们应该如何调整心跳相关参数，以避免频繁 Leader 选举呢？

etcd 默认心跳间隔是 100ms，较小的心跳间隔会导致发送频繁的消息，消耗 CPU 和网络资源。而较大的心跳间隔，又会导致检测到 Leader 故障不可用耗时过长，影响业务可用性。一般情况下，为了避免频繁 Leader 切换，建议你可以根据实际部署环境、业务场景，将心跳间隔时间调整到 100ms 到 400ms 左右，选举超时时间要求至少是心跳间隔的 10 倍。

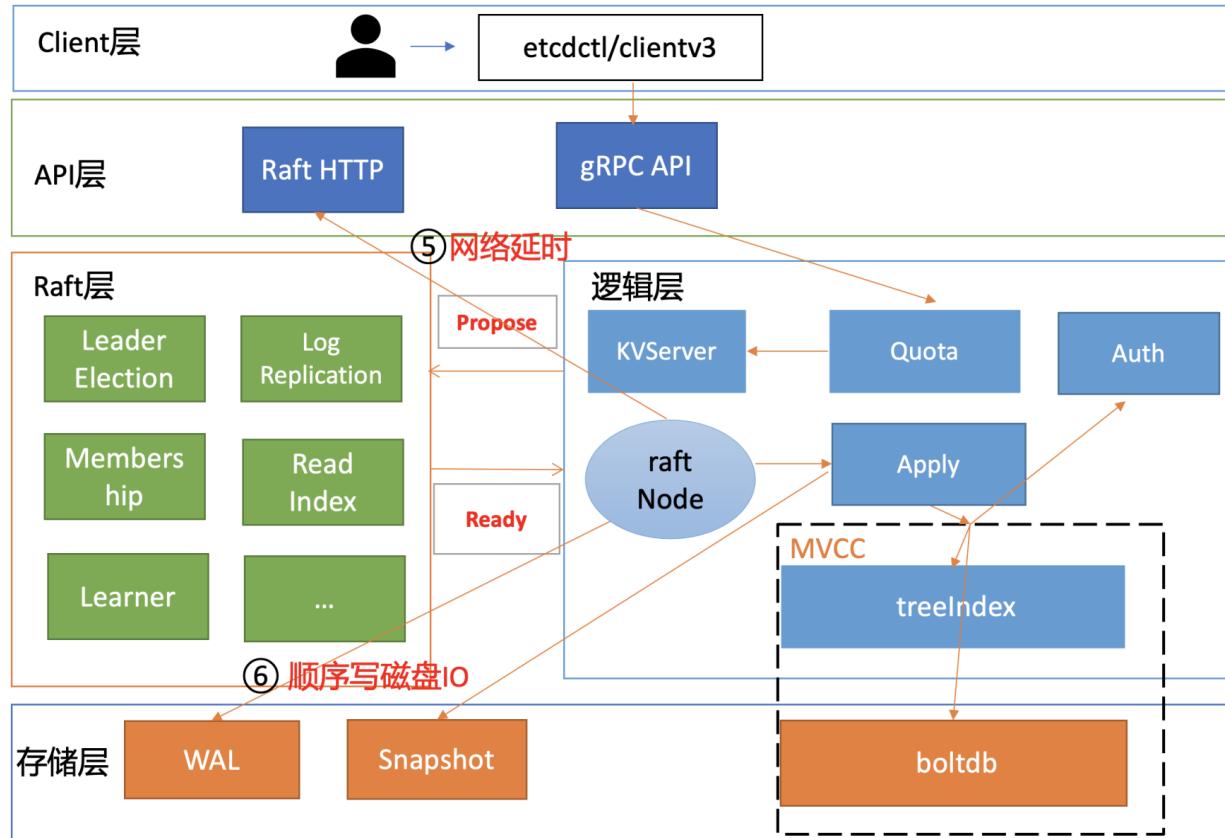
网络和磁盘 IO 延时

当集群 Leader 稳定后，就可以进入 Raft 日志同步流程。

我们假设收到写请求的节点就是 Leader，写请求通过 Propose 接口提交到 Raft 模块后，Raft 模块会输出一系列消息。

etcd server 的 raftNode goroutine 通过 Raft 模块的输出接口 Ready，获取到待发送给 Follower 的日志条目追加消息和待持久化的日志条目。

raftNode goroutine 首先通过 HTTP 协议将日志条目追加消息广播给各个 Follower 节点，也就是流程五。



流程五涉及到各个节点之间网络通信，因此节点之间 RTT 延时对其性能有较大影响。跨可用区、跨地域部署时性能会出现一定程度下降，建议你结合实际网络环境使用 benchmark 工具测试一下。etcd Raft 网络模块在实现上，也会通过流式发送和 pipeline 等技术优化来降低延时、提高网络性能。

同时，raftNode goroutine 也会将待持久化的日志条目追加到 WAL 中，它可以防止进程 crash 后数据丢失，也就是流程六。注意此过程需要同步等待数据落地，因此磁盘顺序写性能决定着性能优异。

为了提升写吞吐量，etcd 会将一批日志条目批量持久化到磁盘。etcd 是个对磁盘 IO 延时非常敏感的服务，如果服务对性能、稳定性有较大要求，建议你使用 SSD 盘。

那使用 SSD 盘的 etcd 集群和非 SSD 盘的 etcd 集群写性能差异有多大呢？

下面是 SSD 盘集群，执行如下 benchmark 命令的压测结果，写 QPS 51298，平均延时 189ms。

 复制代码

```
1 benchmark --endpoints=addr --conns=100 --clients=1000 \
2       put --key-size=8 --sequential-keys --total=10000000 --
3 val-size=256
```

Summary:

```
Total:      1.9494 secs.
Slowest:    0.0682 secs.
Fastest:    0.0028 secs.
Average:   0.0189 secs.
Stddev:    0.0075 secs.
Requests/sec: 51298.4983
```

Response time histogram:

```
0.0028 [1]   |
0.0094 [5096] |.....
0.0159 [34356]  |.....
0.0224 [34308]  |.....
0.0290 [15209]  |.....
0.0355 [7693]  |.....
0.0421 [2783]  |...
0.0486 [429]   |
0.0551 [55]    |
0.0617 [51]    |
0.0682 [19]    |
```

Latency distribution:

```
10% in 0.0104 secs.
25% in 0.0131 secs.
50% in 0.0178 secs.
75% in 0.0228 secs.
90% in 0.0296 secs.
95% in 0.0336 secs.
99% in 0.0400 secs.
99.9% in 0.0524 secs.
```

下面是非 SSD 盘集群，执行同样 benchmark 命令的压测结果，写 QPS 35255，平均延时 279ms。

```
Summary:
 Total:          2.8364 secs.
 Slowest:        0.2236 secs.
 Fastest:        0.0066 secs.
 Average:        0.0279 secs.
 Stddev:         0.0126 secs.
 Requests/sec:  35255.6091

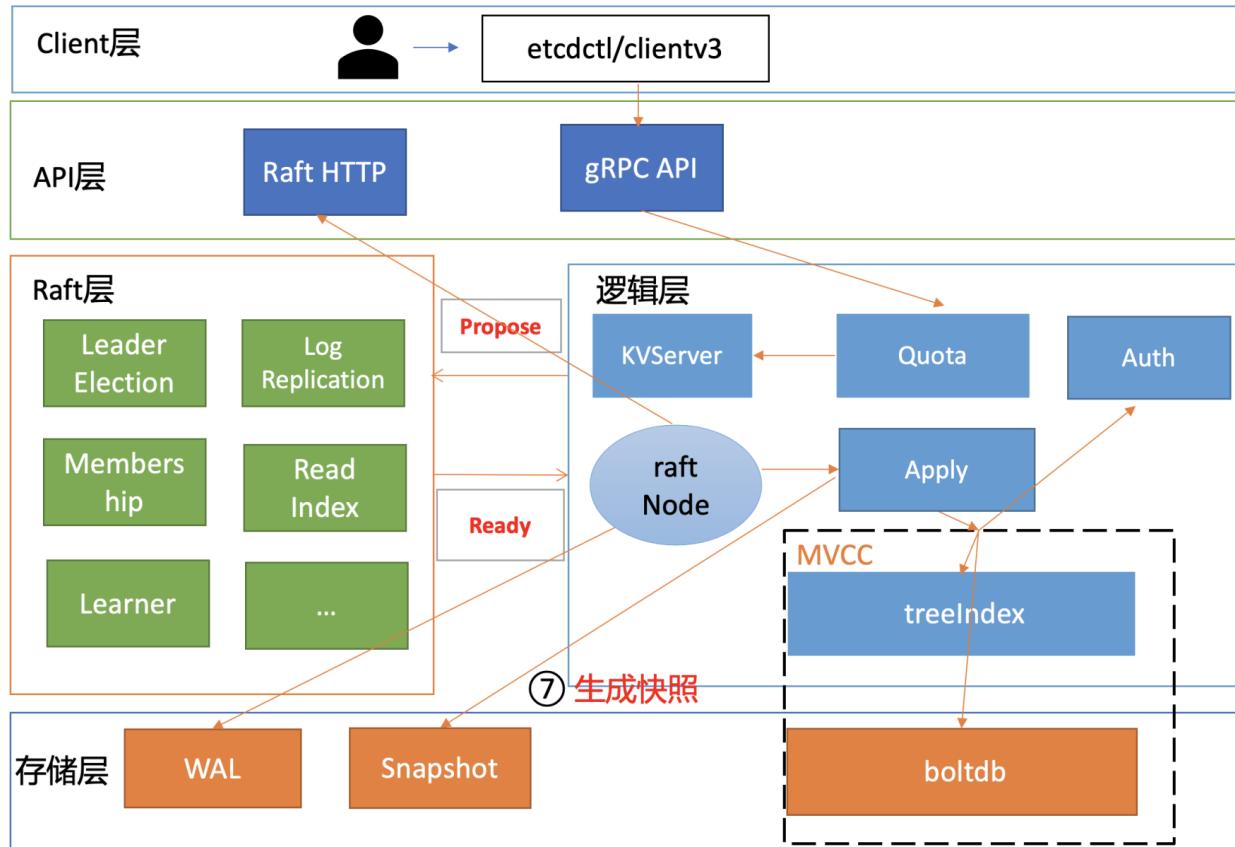
Response time histogram:
 0.0066 [1]      |
 0.0283 [58308]   |#####
 0.0500 [35807]   |#####
 0.0717 [5751]    |...
 0.0934 [123]     |
 0.1151 [0]        |
 0.1368 [0]        |
 0.1585 [0]        |
 0.1802 [0]        |
 0.2019 [0]        |
 0.2236 [10]       |

Latency distribution:
 10% in 0.0132 secs.
 25% in 0.0181 secs.
 50% in 0.0257 secs.
 75% in 0.0361 secs.
 90% in 0.0456 secs.
 95% in 0.0512 secs.
 99% in 0.0626 secs.
 99.9% in 0.0724 secs.
```

快照参数优化

在 Raft 模块中，正常情况下，Leader 可快速地将我们的 key-value 写请求同步给其他 Follower 节点。但是某 Follower 节点若数据落后太多，Leader 内存中的 Raft 日志已经被 compact 了，那么 Leader 只能发送一个快照给 Follower 节点重建恢复。

在快照较大的时候，发送快照可能会消耗大量的 CPU、Memory、网络资源，那么它就会影响我们的读写性能，也就是我们图中的流程七。



一方面，etcd Raft 模块引入了流控机制，来解决日志同步过程中可能出现的大量资源开销、导致集群不稳定的问题。

另一方面，我们可以通过快照参数优化，去降低 Follower 节点通过 Leader 快照重建的概率，使其尽量能通过增量的日志同步保持集群的一致性。

etcd 提供了一个名为 `--snapshot-count` 的参数来控制快照行为。它是指收到多少个写请求后就触发生成一次快照，并对 Raft 日志条目进行压缩。为了帮助 slower Follower 赶上 Leader 进度，etcd 在生成快照，压缩日志条目的时候也会至少保留 5000 条日志条目在内存中。

那 `snapshot-count` 参数设置多少合适呢？

`snapshot-count` 值过大它会消耗较多内存，你可以参考 15 内存篇中 Raft 日志内存占用分析。过小则的话在某节点数据落后时，如果它请求同步的日志条目 Leader 已经压缩了，此时我们就不得不将整个 db 文件发送给落后节点，然后进行快照重建。

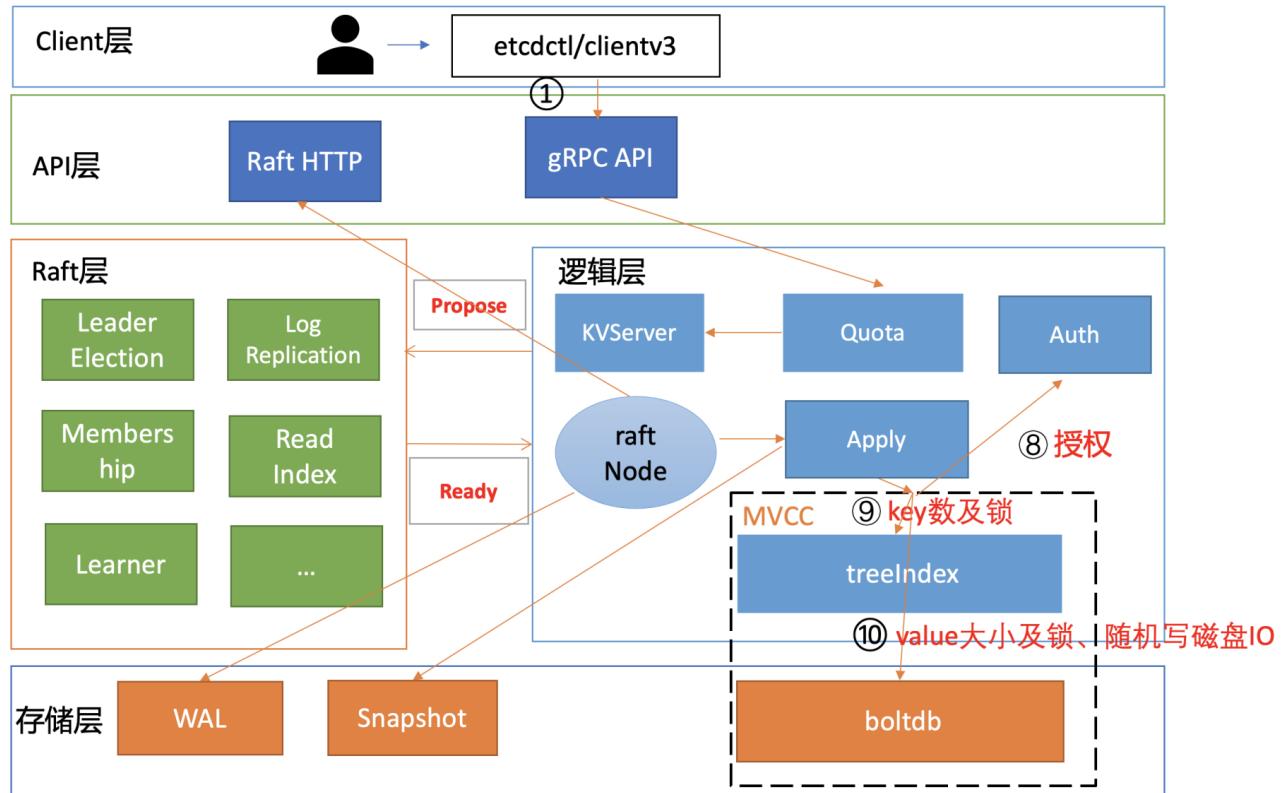
快照重建是极其昂贵的操作，对服务质量有较大影响，因此我们需要尽量避免快照重建。
etcd 3.2 版本之前 `snapshot-count` 参数值是 1 万，比较低，短时间内大量写入就较容易

触发慢的 Follower 节点快照重建流程。etcd 3.2 版本后将其默认值调大到 10 万，老版本升级的时候，你需要注意配置文件是否写死固定的参数值。

大 value

当写请求对应日志条目被集群多数节点确认后，就可以提交到状态机执行了。etcd 的 raftNode goroutine 就可通过 Raft 模块的输出接口 Ready，获取到已提交的日志条目，然后提交到 Apply 模块的 FIFO 待执行队列。因为它是串行应用执行命令，任意请求在应用到状态机时阻塞都会导致写性能下降。

当 Raft 日志条目命令从 FIFO 队列取出执行后，它会首先通过授权模块校验是否有权限执行对应的写操作，对应图中的流程八。影响其性能因素是 RBAC 规则数和锁。



然后通过权限检查后，写事务则会从 `treeIndex` 模块中查找 key、更新的 key 版本号等信息，对应图中的流程九，影响其性能因素是 key 数和锁。

更新完索引后，我们就可以把新版本号作为 `boltdb` key，把用户 key/value、版本号等信息组合成一个 value，写入到 `boltdb`，对应图中的流程十，影响其性能因素是大 value、锁。

如果你在应用中保存 1Mb 的 value，这会给 etcd 稳定性带来哪些风险呢？

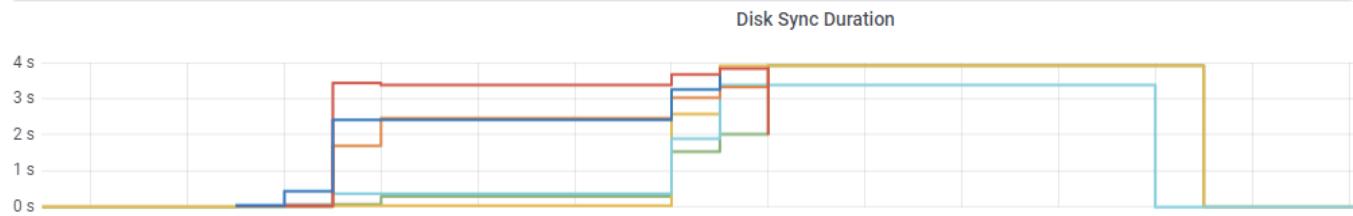
首先会导致读性能大幅下降、内存突增、网络带宽资源出现瓶颈等，上节课我已和你分享过一个 1MB 的 key-value 读性能压测结果，QPS 从 17 万骤降到 1100 多。

那么写性能具体会下降到多少呢？

通过 benchmark 执行如下命令写入 1MB 的数据时候，集群几乎不可用（三节点 8 核 16G，非 SSD 盘），事务提交 P99 延时高达 4 秒，如下图所示。

复制代码

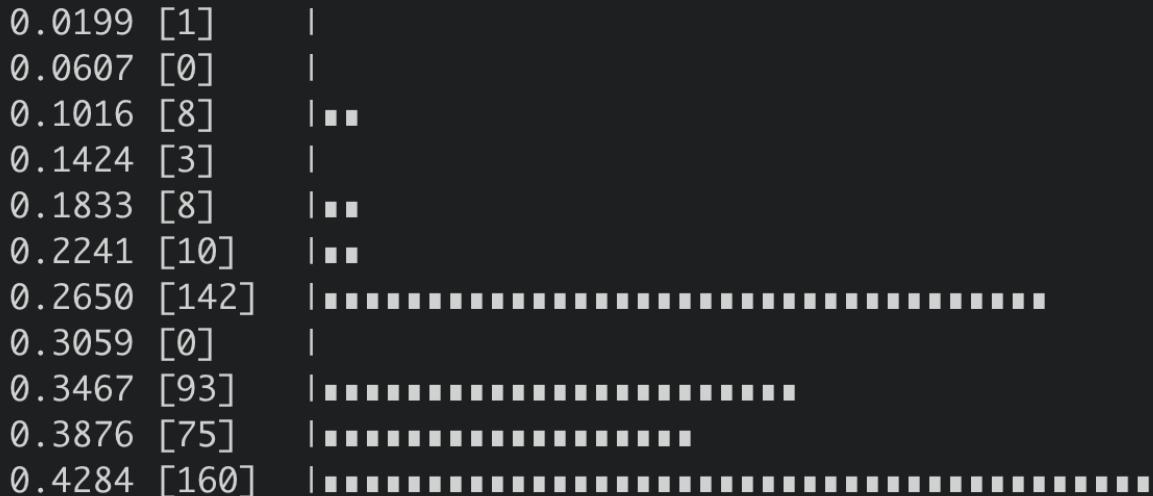
```
1 benchmark --endpoints=addr --conns=100 --clients=1000 \
2 put --key-size=8 --sequential-keys --total=500 --val-
3 size=1024000
```



因此只能将写入的 key-value 大小调整为 100KB。执行后得到如下结果，写入 QPS 仅为 1119/S，平均延时高达 324ms。

Summary:

```
Total:          0.4468 secs.  
Slowest:        0.4284 secs.  
Fastest:        0.0199 secs.  
Average:        0.3243 secs.  
Stddev:         0.0805 secs.  
Requests/sec:   1119.0811
```

Response time histogram:**Latency distribution:**

```
10% in 0.2315 secs.  
25% in 0.2421 secs.  
50% in 0.3456 secs.  
75% in 0.4018 secs.  
90% in 0.4106 secs.  
95% in 0.4158 secs.  
99% in 0.4254 secs.
```

其次 etcd 底层使用的 boltdb 存储，它是个基于 COW(Copy-on-write) 机制实现的嵌入式 key-value 数据库。较大的 value 频繁更新，因为 boltdb 的 COW 机制，会导致 boltdb 大小不断膨胀，很容易超过默认 db quota 值，导致无法写入。

那如何优化呢？

首先，如果业务已经使用了大 key，拆分、改造存在一定客观的困难，那我们就从问题的根源之一的写入对症下药，尽量不要频繁更新大 key，这样 etcd db 大小就不会快速膨胀。

你可以从业务场景考虑，判断频繁的更新是否合理，能否做到增量更新。之前遇到一个 case，一个业务定时更新大量 key，导致被限速，最后业务通过增量更新解决了问题。

如果写请求降低不了，就必须进行精简、拆分你的数据结构了。将你需要频繁更新的数据拆分成小 key 进行更新等，实现将 value 值控制在合理范围以内，才能让你的集群跑的更稳、更高效。

Kubernetes 的 Node 心跳机制优化就是这块一个非常优秀的实践。早期 kubelet 会每隔 10s 上报心跳更新 Node 资源。但是此资源对象较大，导致 db 大小不断膨胀，无法支撑更大规模的集群。为了解决这个问题，社区做了数据拆分，将经常变更的数据拆分成非常细粒度的对象，实现了集群稳定性提升，支撑住更大规模的 Kubernetes 集群。

boltDB 锁

了解完大 value 对集群性能的影响后，我们再看影响流程上的另外一个核心因素 boltdb 锁。

首先我们回顾下 etcd 读写性能优化历史，它经历了以下流程：

3.0 基于 Raft log read 实现线性读，线性读需要经过磁盘 IO，性能较差；

3.1 基于 ReadIndex 实现线性读，每个节点只需要向 Leader 发送 ReadIndex 请求，不涉及磁盘 IO，提升了线性读性能；

3.2 将访问 boltdb 的锁从互斥锁优化到读写锁，提升了并发读的性能；

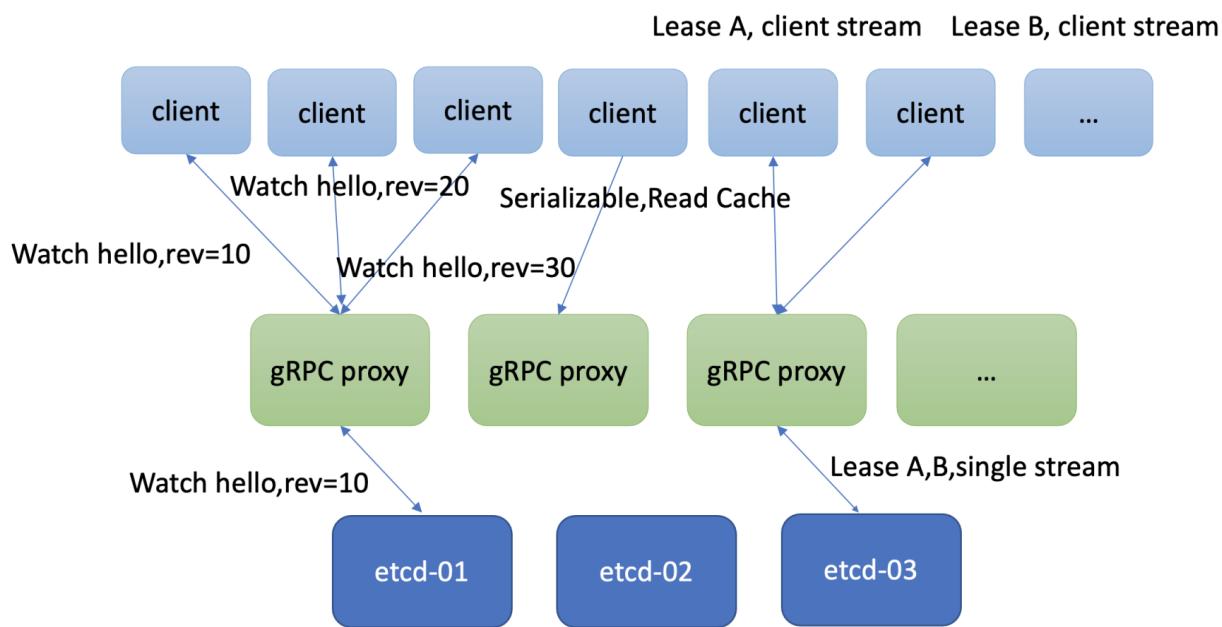
3.4 实现全并发读，去掉了 buffer 锁，长尾读几乎不再影响写。

并发读特性的核心原理是创建读事务对象时，它会全量拷贝当前写事务未提交的 buffer 数据，并发的读写事务不再阻塞在一个 buffer 资源锁上，实现了全并发读。

最重要的是，写事务也不再因为 expensive read request 长时间阻塞，有效的降低了写请求的延时，详细测试结果你可以参考 [并发读特性实现 PR](#)，因篇幅关系就不再详细描述。

扩展性能

当然有不少业务场景你即便用最高配的硬件配置，etcd 可能还是无法解决你所面临的性能问题。etcd 社区也考虑到此问题，提供了一个名为 [gRPC proxy](#) 的组件，帮助你扩展读、扩展 watch、扩展 Lease 性能的机制，如下图所示。



扩展读

如果你的 client 比较多，etcd 集群节点连接数大于 2 万，或者你想平行扩展串行读的性能，那么 gRPC proxy 就是良好一个解决方案。它是个无状态节点，为你提供高性能的读缓存的能力。你可以根据业务场景需要水平扩容若干节点，同时通过连接复用，降低服务端连接数、负载。

它也提供了故障探测和自动切换能力，当后端 etcd 某节点失效后，会自动切换到其他正常节点，业务 client 可对此无感知。

扩展 Watch

大量的 watcher 会显著增大 etcd server 的负载，导致读写性能下降。etcd 为了解决这个问题，gRPC proxy 组件里面提供了 watcher 合并的能力。如果多个 client Watch 同 key 或者范围（如上图三个 client Watch 同 key）时，它会尝试将你的 watcher 进行合并，降低服务端的 watcher 数。

然后当它收到 etcd 变更消息时，会根据每个 client 实际 Watch 的版本号，将增量的数据变更版本，分发给你的多个 client，实现 watch 性能扩展及提升。

扩展 Lease

我们知道 etcd Lease 特性，提供了一种客户端活性检测机制。为了确保你的 key 不被淘汰，client 需要定时发送 keepalive 心跳给 server。当 Lease 非常多时，这就会导致 etcd 服务端的负载增加。在这种场景下，gRPC proxy 提供了 keepalive 心跳连接合并的机制，来降低服务端负载。

小结

今天我通过从上至下的写请求流程分析，介绍了各个流程中可能存在的瓶颈和优化方法、最佳实践。最后我从分层的角度，为你总结了一幅优化思路全景图，你可以参考一下下面这张图，它将我们这两节课讨论的 etcd 性能优化、扩展问题分为了以下几类：

业务应用层，etcd 应用层的最佳实践；

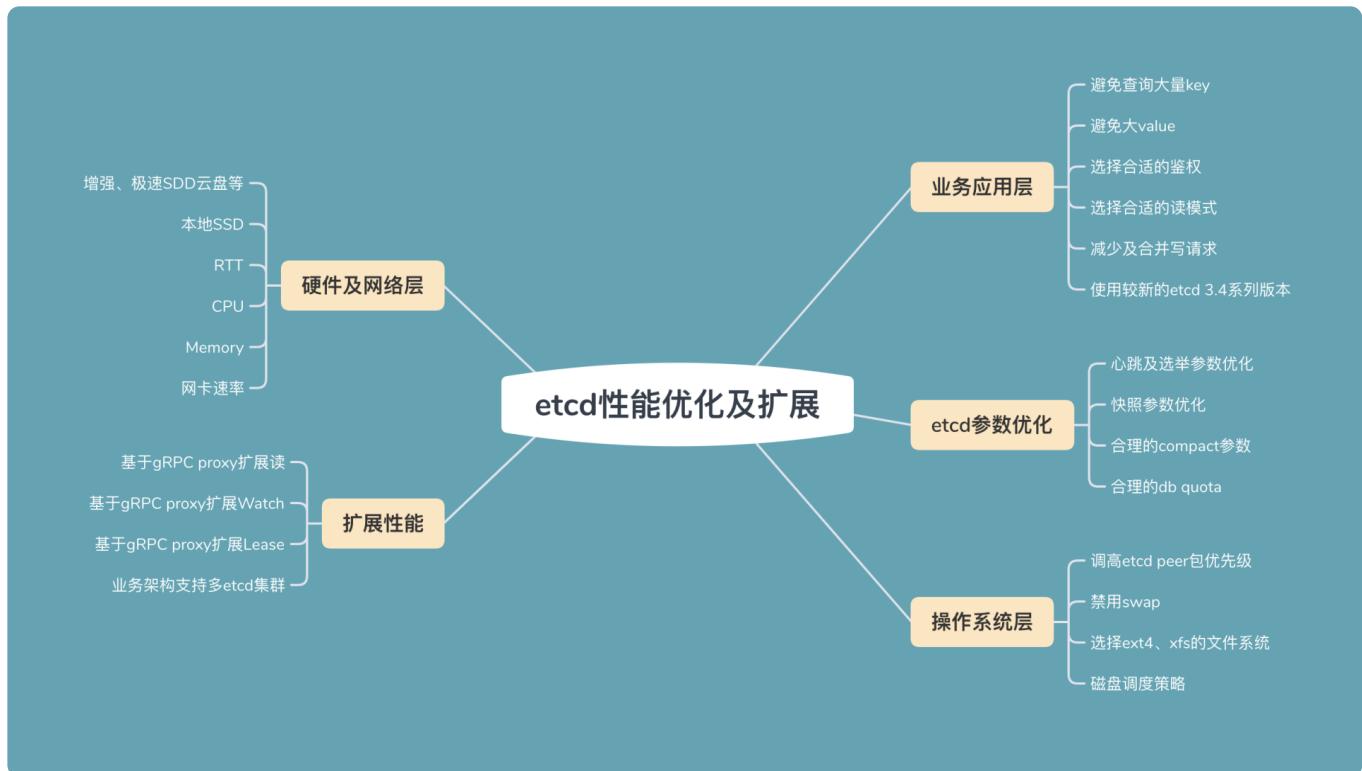
etcd 内核层，etcd 参数最佳实践；

操作系统层，操作系统优化事项；

硬件及网络层，不同的硬件设备对 etcd 性能有着非常大的影响；

扩展性能，基于 gRPC proxy 扩展读、Watch、Lease 的性能。

希望你通过这节课的学习，以后在遇到 etcd 性能问题时，能分别从请求执行链路和分层的视角去分析、优化瓶颈，让业务和 etcd 跑得更稳、更快。



思考题

最后，我还给你留了一个思考题。

watcher 较多的情况下，会不会对读写请求性能有影响呢？如果会，是在什么场景呢？
gRPC proxy 能安全的解决 watcher 较多场景下的扩展性问题吗？

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

更多课程推荐

Redis 核心技术与实战

从原理到实战，彻底吃透 Redis

蒋德钧

中科院计算所副研究员



涨价倒计时

现仅半价 **¥89** 4月17日涨价至**¥199**

© 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 16 | 性能及稳定性（上）：如何优化及扩展etcd性能？

下一篇 18 | 实战：如何基于Raft从0到1构建一个支持多存储引擎分布式KV服务？

精选留言 (1)

写留言



Coder

2021-02-27

老师，请问一下如果业务写多读少，有什么优化办法？难道不能用etcd

作者回复: 1. 首先尽量选择高配的节点，各个节点之间尽量就近部署，使节点之间RTT延时尽量低，然后可使用本地SSD，并结合业务场景，构造一定的数据量，通过benchmark工具压测下，评估压测性能是否能满足业务诉求

2. 若无法满足，评估业务若存在多种路径的key写入，能否垂直拆分下，不同路径下的key，写入到不同etcd集群，比如kubernetes集群的主集群数据与event分离部署也是这样的思路

3. 评估业务上层能否支持多实例etcd集群，比如你要搞个任务系统，假设几十万的的节点，每个节点通过watch机制监听自己路径下的任务key，若任务系统的QPS较大，你可以通过多etcd集群来支持，一组节点分配一个etcd集群。然后你可以通过引入一个调度服务来给各个节点分配etcd集群，agent启动时，通过调度服务请求分配一个etcd集群，若未调度，则按一定的策略，比如e

tcd集群的负载情况分配一个负载最低给新增的agent，有了调度结果后，随后agent就知道监听哪个etcd集群了。随着节点数增多，你可以平行扩容etcd集群。

4. 确定是否真的依赖etcd的一些特性，可以在方案选型中，评估其他方案，比如redis等，写性能更好，还有底层存储引擎使用LSM实现的leveldb/rocksdb等，也是非常好的候选方案

