

43 | 分布式协调：etcd读写、MVCC原理与监听机制

2023-01-17 郑建勋 来自北京



天下无鱼

<https://shikey.com/>

《Go进阶·分布式爬虫实战》

[课程介绍 >](#)



讲述：郑建勋

时长 11:47 大小 10.76M

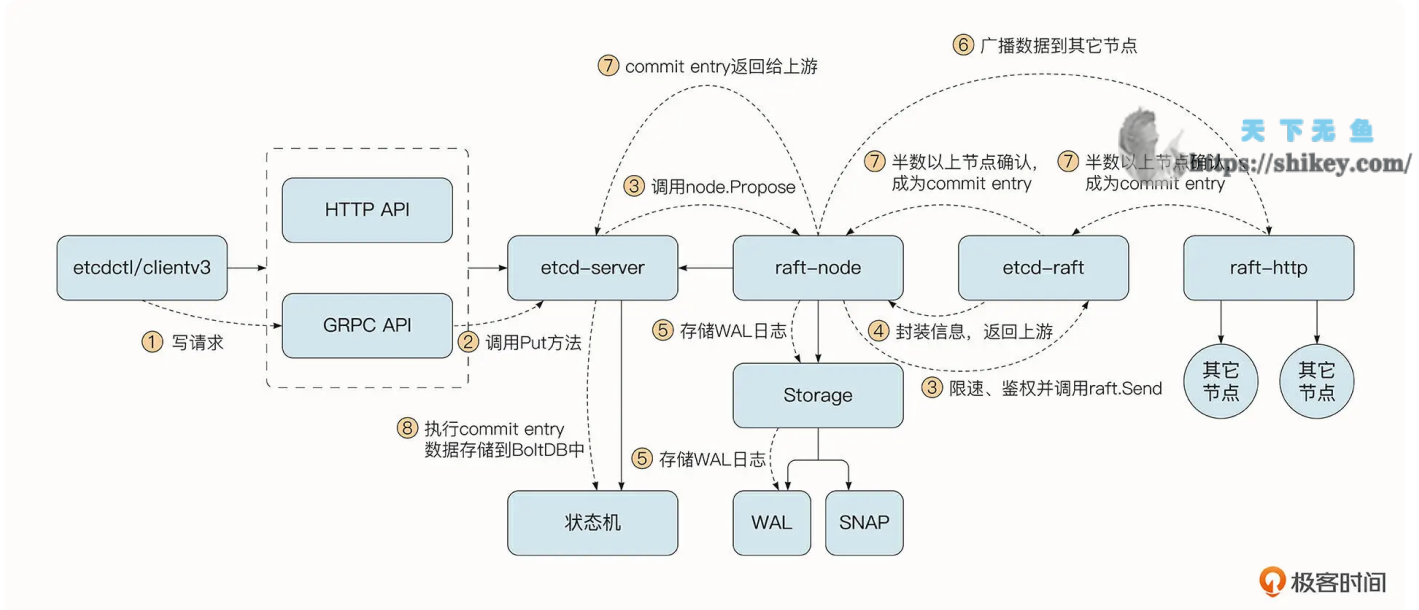


你好，我是郑建勋。

这节课，我们重点来看看 etcd 的读写流程，以及它的两个重要特性：MVCC 原理和监听机制。

写的完整流程

我们先来看看 etcd 怎么完整地写入请求。



1. 客户端通过 GRPC 协议访问 etcd-server 服务端。
2. 如果是一个写请求，会访问 etcd-server 注册的 Put 方法。要注意的是，在访问 etcd-server 时，会进行一些检查，例如 DB 配额（Quota）的检查。此外，如果客户端访问的节点不是 Leader 节点，这个节点会将请求转移到 Leader 中。
3. etcd-server 会调用 raft-node 模块的 Propose 方法进行限速、鉴权等判断，之后 raft-node 模块调用 etcd-raft 模块完成数据的封装。
4. 接着，etcd-raft 模块会将封装后的数据返回给 raft-node 模块。
5. raft-node 模块调用 storage 存储模块，将本次操作对应的 Entry 记录存储到 WAL 日志文件中。
6. raft-node 模块将当前 Entry 广播给集群中的其他节点，snap 模块还会在适当时候保存当前数据的快照。
7. 当 Leader 最终收到了半数以上节点的确认时，该 Entry 的状态会变为 committed，这时 etcd-raft 模块会将 Commit Index 返回上游，供 etcd-server 模块执行。后面我们还会看到，etcd-server 实现了 MVCC 机制，维护了某一个 Key 过去所有的版本记录。

etcd 状态机中的数据存储包含了两个部分，第一部分是内存索引叫做 treeIndex，用于存储 Key 与版本号之间的映射关系，另一部分是数据的持久化存储，默认情况下，etcd 状态机的持久化存储选择的是 BoltDB 数据库（良好的接口设计让我们可以选择不同的存储引擎）。

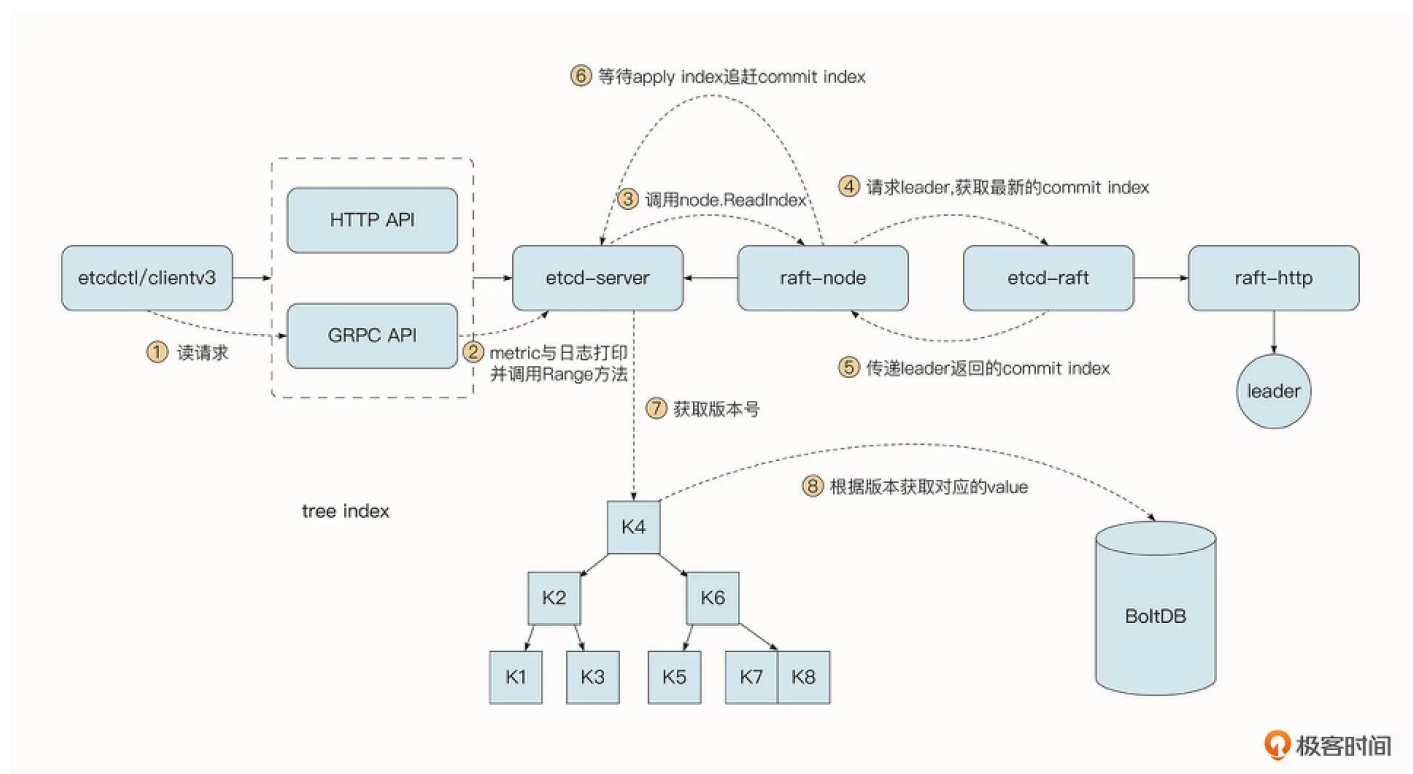
BoltDB 作为 KV 存储引擎，底层使用了 B+ 树，并且支持事务。etcd v3 提供的事务能力就是基于 BoltDB 的事务实现的。在 BoltDB 中存储的数据 Key 值其实是版本号，而 Value 值包括了原始请求中的键值对和相应的版本号。

另外，还要格外注意的是，客户端调用写入方法 **Put** 成功后，并不意味着数据已经持久化到 **BoltDB** 了。因为这时 **etcd** 并未提交事务，数据只更新在了 **BoltDB** 管理的内存数据结构中。**BoltDB** 事务提交的过程包含平衡 **B+** 树、调整元数据信息等操作，因此提交事务是比较昂贵的。如果我们每次更新都提交事务，**etcd** 的写性能就会较差。为了解决这一问题，**etcd** 也有对策。**etcd** 会合并多个写事务请求，通常情况下定时机制会分批次（默认 100 毫秒 / 次）统一提交事务，这就大大提高了吞吐量。

但是这种优化又导致了另一个问题。事务未提交时，读请求可能无法从 **BoltDB** 中获取到最新的数据。为了解决这个问题，**etcd** 引入了一个 **Bucket Buffer** 来保存暂未提交的事务数据。**etcd** 处理读请求的时候，会优先从 **Bucket Buffer** 里面读取，其次再从 **BoltDB** 中读取，通过 **Bucket Buffer** 提升读写性能，同时也保证了数据一致性。

读完整流程

下面我们看看一个完整的线性一致性的读请求都要经过哪些过程。



1. 首先，客户端通过 **GRPC API** 访问 **etcd-server** 服务端，这一阶段会经过注册到 **GRPC** 服务器中的拦截器，实现日志打印、**Metric** 统计等功能。
2. 读操作调用的是 **etcd-server** 的 **Range** 方法，**etcd-server** 会判断当前的请求是否需要线性一致性的读。
3. 对于线性一致性读，**etcd-server** 会调用 **raft-node** 模块的 **ReadIndex** 方法。

4. raft-node 模块在 etcd-raft 模块的帮助下请求 Leader 节点，获取 Leader 节点中当前最新的 Commit Index。
5. etcd-raft 模块将 Leader 返回的 Commit Index 传递给上游模块 etcd-server 模块。 <https://shikey.com/>
6. 读取协程会陷入到等待的状态，一直到当前状态机已经执行的 Apply Index 追赶上当前最新的 Commit Index 为止。一旦 Apply Index 追赶上 Leader 的 Commit Index，就意味着当前我们读取到的数据一定是在最后一次写入操作之后，这就保证了读的强一致性。
7. 接着 etcd-server 会在 treeIndex 这个 B 树中，得到当前请求中 Key 的最新的版本号（也可以在请求中指定读取的版本号和范围）。
8. etcd-server 最终在 BoltDB 中通过版本号查询到对应的 Value 值，并返回给客户端。


MVCC 机制

etcd 存储了当前 Key 过去所有操作的版本记录。这样做的好处是，我们可以很方便地获取所有的操作记录，而这些记录常常是实现更重要的特性的基础，例如要实现可靠的事件监听就需要 Key 的历史信息。

etcd v2 会在内存中维护一个较短的全局事件滑动窗口，保留最近的 1000 条变更事件。但是当事件太多的时候，就需要删除老的事件，可能导致事件的丢失。而 etcd v3 解决了这一问题，etcd v3 将历史版本存储在了 BoltDB 当中进行了持久化。可靠的 Watch 机制将避免客户端执行一些更繁重的查询操作，提高了系统的整体性能。

借助 Key 的历史版本信息，我们还能够实现乐观锁的机制。乐观锁即乐观地认为并发操作同一份数据不会发生冲突，所以不会对数据全局加锁。但是当事务提交时，她又能够检测到是否会出现数据处理异常。乐观锁的机制让系统在高并发场景下仍然具备高性能。**这种基于多版本技术实现的乐观锁机制也被称为 MVCC。**

下面就让我们来看看 etcd 是如何实现 MVCC 机制，对多版本数据的管理与存储的吧。在 etcd 中，每一个操作不会覆盖旧的操作，而是会指定一个新的版本，其结构为 revision。

 复制代码

```
1 type revision struct {  
2     main int64  
3     sub  int64  
4 }
```


revision 主要由两部分组成，包括 main 与 sub 两个字段。其中每次出现一个新事务时 main 都会递增 1，而对于同一个事务，执行事务中每次操作都会导致 sub 递增 1，这保证了每一次操作的版本都是唯一的。假设事务 1 中的两条操作分别如下。



复制代码

```
1 key = "zjx"    value = "38"
2 key = "olaya"  value = "19"
```

事务 2 中的两条操作是下面的样子。

复制代码

```
1 key = "zjx"    value = "56"
2 key = "olaya"  value = "22"
```

那么每条操作对应的版本号就分别是下面这样。

复制代码

```
1 revision = {1, 0}
2 revision = {1, 1}
3 revision = {2, 0}
4 revision = {2, 1}
```

etcd 最终会默认将键值对数据存储到 BoltDB 当中，完成数据的落盘。不过为了管理多个版本，在 BoltDB 中的 Key 对应的是 revision 版本号，而 Value 对应的是该版本对应的实际键值对。BoltDB 在底层使用 B+ 树进行存储，而 B+ 树的优势就是可以实现范围查找，这有助于我们在读取数据以及实现 Watch 机制的时候，查找某一个范围内的操作记录。

看到这里你可能会有疑问，在 BoltDB 中存储的 key 是版本号，但是在用户查找的时候，可能只知道具体数据里的 Key，那如何完成查找呢？

为了解决这一问题，etcd 在内存中实现了一个 B 树的索引 treeIndex，封装了 [Google 开源的 B 树的实现](#)。B 树的存储结构方便我们完成范围查找，也能够和 BoltDB 对应的 B+ 树的能力对应起来。treeIndex 实现的索引，实现了数据 Key 与 keyIndex 实例之间的映射关系，而在 keyIndex 中存储了当前 Key 对应的所有历史版本信息。通过这样的二次查找，我们就可以通过 Key 查找到 BoltDB 中某一个版本甚至某一个范围的 Value 值了。

借助 etcd 的 MVCC 机制以及 BoltDB 数据库，我们可以在 etcd 中实现事务的 ACID 特性。

etcd clientv3 中提供的 [🔗 简易事务 API](#)正是基于此实现的。



Watch 机制

etcd 支持监听某一个特定的 Key，也支持监听一个范围。etcdv3 的 MVCC 机制将历史版本都保存在了 BoltDB 中，避免了历史版本的丢失。同时，etcdv3 还使用 GRPC 协议实现了客户端与服务器之间数据的流式传输。

那 etcd 服务端是如何实现 Watch 机制的呢？

当客户端向 etcd 服务器发出监听的请求时，etcd 服务器会生成一个 watcher。etcd 会维护这些 watcher，并将其分为两种类型：syncd 和 unsynced。

syncd watcher 意味着当前 watcher 监听的最新事件都已经同步给客户端，接下来 syncd watcher 陷入休眠并等待新的事件。unsynced watcher 意味着当前 watcher 监听的事件并未完全同步到客户端。etcd 会启动一个单独的协程帮助 unsynced watcher 进行追赶。当 unsynced watcher 处理完最新的操作，将最新的事件同步到客户端之后，就会变为 syncd watcher。

unsynced watcher 的处理

etcd 初始化时创建了单独的协程来处理 unsynced watcher。由于 unsynced watcher 可能会很多，etcd 采用了一种巧妙的方法来处理它，具体方式如下。

1. 选择一批 unsynced watcher，作为此次要处理的 watcher。
2. 查找这批 watcher 中最小的版本值。
3. 在 BoltDB 中进行范围查询，查询最小版本号与当前版本号之间的所有键值对。
4. 将这些键值对转换为 Event 事件，满足 watcher 条件的 Event 事件将会被发送回对应的客户端。
5. 当 unsynced watcher 处理完最新的操作之后，就会变为 syncd watcher。

syncd watcher 的处理

当 etcd 收到一个写请求，Key-Value 发生变化的时候，对应的 synced watcher 需要能够感知到并完成最新事件的推送。这一步主要是在 Put 事务结束时来做的。



Put 事务结束后，会调用 watchableStore.notify，获取监听了当前 Key 的 watcher，然后将 Event 送入这些 watcher 的 Channel 中，完成最终的处理和发送。

监听当前 Key 的 watcher 可能很多，你可能会想到用一个哈希表来存储 Key 与 watcher 的对应关系，但是这还不够，因为一个 watcher 可能会监听 Key 的范围和前缀。因此，为了能够高效地获取某一个 Key 对应的 watcher，除了使用哈希表，etcd 还使用了区间树结构来存储 watcher 集合。当产生一个事件时，etcd 首先需要从哈希表查找是否有 watcher 监听了该 Key，然后它还需要从区间树重找出满足条件的所有区间，从区间的值获取监听的 watcher 集合。

总结

本节课程中我们介绍了 etcd 完整的读写流程。在整个复杂的流程中，核心模块无外乎是 GRPC 请求、权限和参数的检查、WAL 日志的存储、Raft 节点的网络协调以及执行操作更新状态机的状态等。把握这些核心处理流程和模块，也就能理解 etcd 是如何实现一致性、容错性以及高性能的了。

etcd 存储实现了 MVCC 机制，保存了历史版本的所有数据。这种机制主要是依靠了内存索引 treeIndex 与后端存储 BoltDB，它不仅提高了 etcd 系统的并发处理能力，也为构建可靠的 Watch 机制和事务提供了基础。

etcd 将 watch 对象分为了 unsynced watcher 与 synced watcher，其中 synced watcher 表示最新事件已经同步给客户端，而 unsynced watcher 表示最新事件还未同步到客户端。etcd 在初始化时就建立了一个单独的协程完成 unsynced watcher 的追赶，通过范围查找，即便存在大量的 watcher，也能轻松应对。

这两节课我们更多还是在讲解理论，我希望你能够明白 etcd 的这些重要功能背后的实现机制，为我们后面要实战分布式协调做准备。如果你理解了原理却还觉得不过瘾，想要深入到源码中去学习 etcd 的话，我也推荐你去看看《etcd 技术内幕》这本书，它对 etcd 源码的各个字段都介绍得比较详细。

课后题

学完这节课，给你留两道思考题。



1. `treeIndex` 的结构为什么是 B 树而不是哈希表或者是二叉树？
2. 写操作会调用 `etcd Put` 方法，调用 `Put` 方法结束时并未真正地执行 BoltDB 的 `commit` 操作进行事务提交，如果这个时候节点崩溃了，如何保证数据不丢失呢？

欢迎你在留言区与我交流讨论，我们下节课见。

分享给需要的人，Ta购买本课程，你将得 20 元

生成海报并分享

赞 0 提建议

© 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 42 | 他山之石：etcd架构之美

下一篇 44 | 一个程序多种功能：构建子命令与flags

精选留言 (1)

写留言



Realm

2023-01-17 来自浙江

思考题：

一 `treeIndex` 的结构为什么是 B 树而不是哈希表或者是二叉树？

不使用【hash表】的原因：

1. hash表不支持范围查询；
2. hash表可能有hash碰撞的问题($\text{Hash_fn}(k1) = \text{Hash_fn}(key2)$),还需要使用其他方法进行进一步处理(如:拉链法)；
3. hash表不支持排序；
4. hash表不支持key的前缀索引,prefix=xxx,想必是用不了；

不使用【二叉树】的原因：

1. 二叉树造成树的层次太高,查找的时候，可能造成磁盘IO的次数较多,性能不好.

二 如果这个时候节点崩溃了，如何保证数据不丢失呢？
应该是通过WAL进行保障，先写日志在提交.



这样看，很多思路与MySQL相似.

