01 | etcd的前世今生：为什么Kubernetes使用etcd?

2021-01-20 唐聪

etcd实战课 进入课程 >

讲述：王超凡
时长 18:57 大小 17.36M

你好，我是唐聪。

今天是专栏课程的第一讲，我们就从 etcd 的前世今生讲起。让我们一起穿越回 2013 年，看看 etcd 最初是在什么业务场景下被设计出来的?

2013 年，有一个叫 CoreOS 的创业团队，他们构建了一个产品，Container Linux，它是一个开源、轻量级的操作系统，侧重自动化、快速部署应用服务，并要求应用程序都在容器中运行，同时提供集群化的管理方案，用户管理服务就像单机一样方便。

他们希望在重启任意一节点的时候，用户的服务不会因此而宕机，导致无法提供服务，因此需要运行多个副本。但是多个副本之间如何协调，如何避免变更的时候所有副本不可用呢?

为了解决这个问题，CoreOS 团队需要一个协调服务来存储服务配置信息、提供分布式锁等能力。怎么办呢？当然是分析业务场景、痛点、核心目标，然后是基于目标进行方案选型，评估是选择社区开源方案还是自己造轮子。这其实就是我们遇到棘手问题时的通用解决思路，CoreOS 团队同样如此。

假设你是 CoreOS 团队成员，你认为在这样的业务场景下，理想中的解决方案应满足哪些目标呢？

如果你有过一些开发经验，应该能想到一些关键点了，我根据自己的经验来总结一下，一个协调服务，理想状态下大概需要满足以下五个目标：

1. **可用性角度：高可用。**协调服务作为集群的控制面存储，它保存了各个服务的部署、运行信息。若它故障，可能会导致集群无法变更、服务副本数无法协调。业务服务若此时出现故障，无法创建新的副本，可能会影响用户数据面。
2. **数据一致性角度：提供读取“最新”数据的机制。**既然协调服务必须具备高可用的目标，就必然不能存在单点故障（single point of failure），而多节点又引入了新的问题，即多个节点之间的数据一致性如何保障？比如一个集群 3 个节点 A、B、C，从节点 A、B 获取服务镜像版本是新的，但节点 C 因为磁盘 I/O 异常导致数据更新缓慢，若控制端通过 C 节点获取数据，那么可能会导致读取到过期数据，服务镜像无法及时更新。
3. **容量角度：低容量、仅存储关键元数据配置。**协调服务保存的仅仅是服务、节点的配置信息（属于控制面配置），而不是与用户相关的数据。所以存储上不需要考虑数据分片，无需过度设计。
4. **功能：增删改查，监听数据变化的机制。**协调服务保存了服务的状态信息，若服务有变更或异常，相比控制端定时去轮询检查一个个服务状态，若能快速推送变更事件给控制端，则可提升服务可用性、减少协调服务不必要的性能开销。
5. **运维复杂度：可维护性。**在分布式系统中往往会遇到硬件 Bug、软件 Bug、人为操作错误导致节点宕机，以及新增、替换节点等运维场景，都需要对协调服务成员进行变更。若能提供 API 实现平滑地变更成员节点信息，就可以大大降低运维复杂度，减少运维成本，同时可避免因人工变更不规范可能导致的服务异常。

了解完理想中的解决方案目标，我们再来看 CoreOS 团队当时为什么选择了从 0 到 1 开发一个新的协调服务呢？

如果使用开源软件，当时其实是有 ZooKeeper 的，但是他们为什么不用 ZooKeeper 呢？我们来分析一下。

从高可用性、数据一致性、功能这三个角度来说，ZooKeeper 是满足 CoreOS 诉求的。然而当时的 ZooKeeper 不支持通过 API 安全地变更成员，需要人工修改一个个节点的配置，并重启进程。

若变更姿势不正确，则有可能出现脑裂等严重故障。适配云环境、可平滑调整集群规模、在线变更运行时配置是 CoreOS 的期望目标，而 ZooKeeper 在这块的可维护成本相对较高。

其次 ZooKeeper 是用 Java 编写的，部署较繁琐，占用较多的内存资源，同时 ZooKeeper RPC 的序列化机制用的是 Jute，自己实现的 RPC API。无法使用 curl 之类的常用工具与之互动，CoreOS 期望使用比较简单的 HTTP + JSON。

因此，CoreOS 决定自己造轮子，那 CoreOS 团队是如何根据系统目标进行技术方案选型的呢？

etcd v1 和 v2 诞生

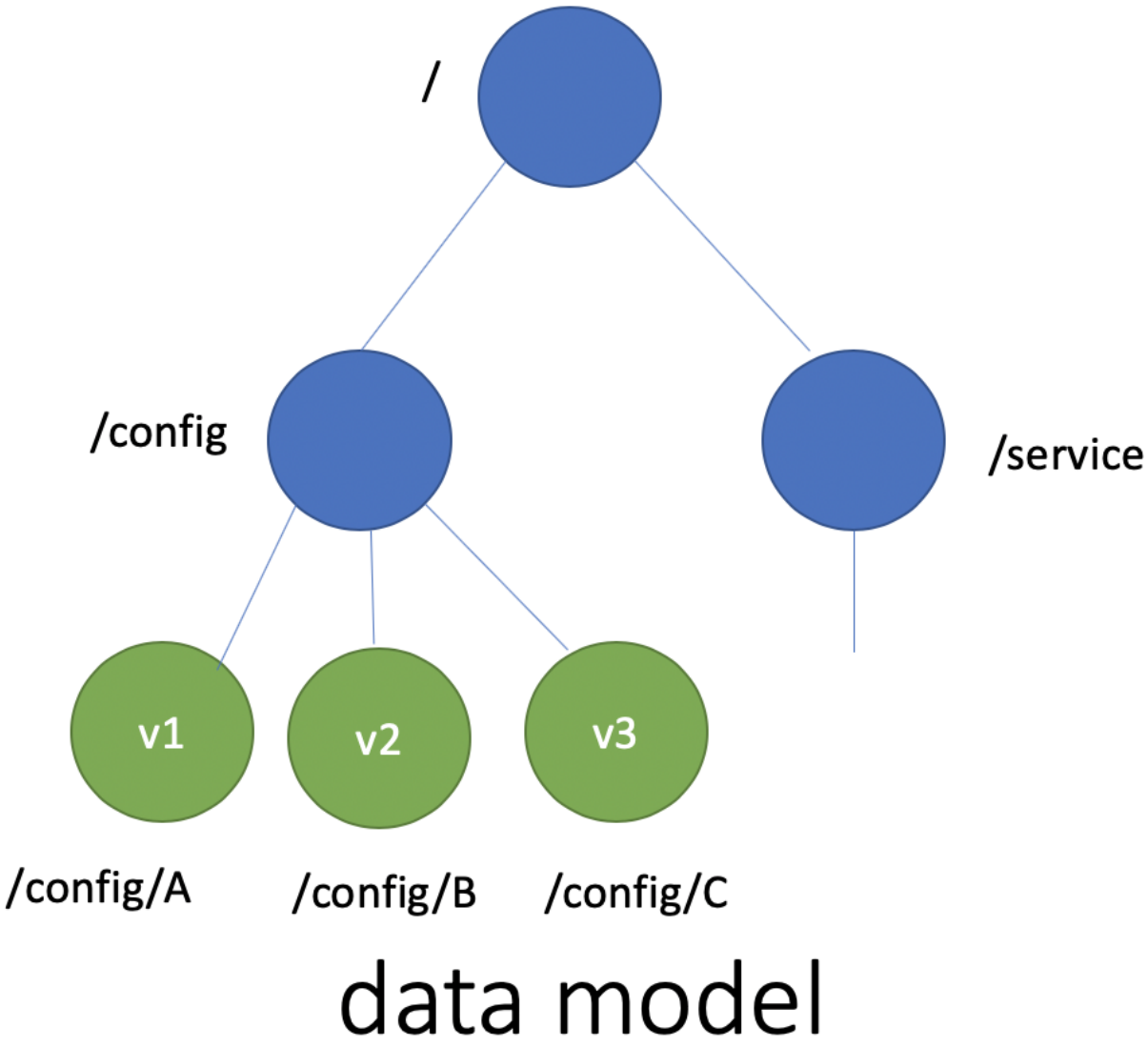
首先我们来看服务高可用及数据一致性。前面我们提到单副本存在单点故障，而多副本又引入数据一致性问题。

因此为了解决数据一致性问题，需要引入一个共识算法，确保各节点数据一致性，并可容忍一定节点故障。常见的共识算法有 Paxos、ZAB、Raft 等。CoreOS 团队选择了易理解实现的 Raft 算法，它将复杂的一致性问题分解成 Leader 选举、日志同步、安全性三个相对独立的子问题，只要集群一半以上节点存活就可提供服务，具备良好的可用性。

其次我们再来看数据模型（Data Model）和 API。数据模型参考了 ZooKeeper，使用的是基于目录的层次模式。API 相比 ZooKeeper 来说，使用了简单、易用的 REST API，提供了常用的 Get/Set/Delete/Watch 等 API，实现对 key-value 数据的查询、更新、删除、监听等操作。

key-value 存储引擎上，ZooKeeper 使用的是 Concurrent HashMap，而 etcd 使用的是则是简单内存树，它的节点数据结构精简后如下，含节点路径、值、孩子节点信息。这是

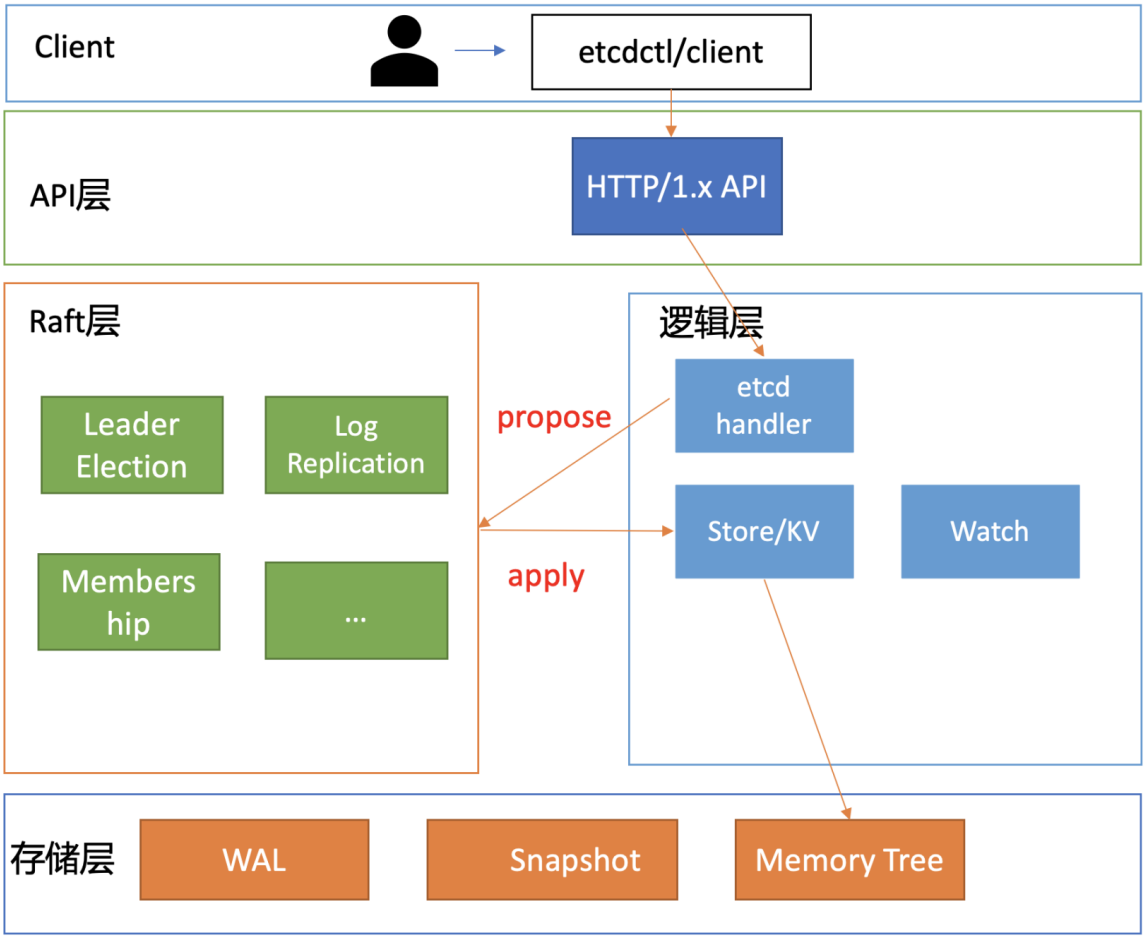
一个典型的低容量设计，数据全放在内存，无需考虑数据分片，只能保存 key 的最新版本，简单易实现。



复制代码

```
type node struct {
    Path string  //节点路径
    Parent *node //关联父亲节点
    Value      string     //key的value值
    ExpireTime time.Time //过期时间
    Children   map[string]*node //此节点的孩子节点
}
```

最后我们再来看可维护性。Raft 算法提供了成员变更算法，可基于此实现成员在线、安全变更，同时此协调服务使用 Go 语言编写，无依赖，部署简单。

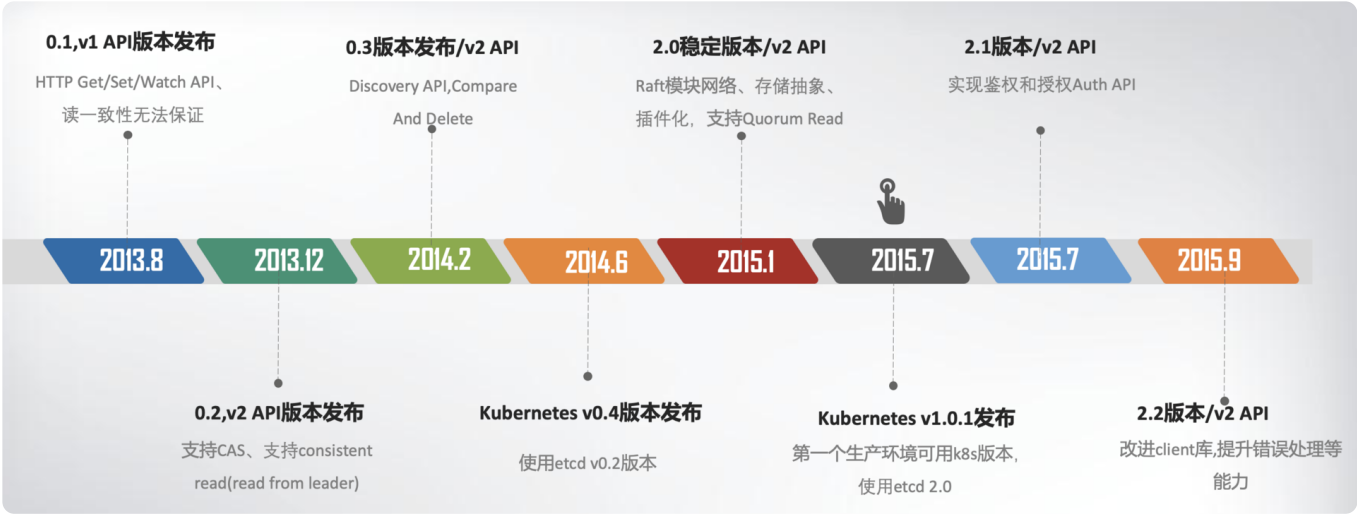


基于以上技术方案和架构图，CoreOS 团队在 2013 年 8 月对外发布了第一个测试版本 v0.1，API v1 版本，命名为 etcd。

那么 etcd 这个名字是怎么来的呢？其实它源于两个方面，unix 的“/etc”文件夹和分布式系统（“D”istribute system) 的 D，组合在一起表示 etcd 是用于存储分布式配置的信息存储服务。

v0.1 版本实现了简单的 HTTP Get/Set/Delete/Watch API，但读数据一致性无法保证。v0.2 版本，支持通过指定 consistent 模式，从 Leader 读取数据，并将 Test And Set 机制修正为 CAS(Compare And Swap)，解决原子更新的问题，同时发布了新的 API 版本 v2，这就是大家熟悉的 etcd v2 版本，第一个非 stable 版本。

下面，我用一幅时间轴图，给你总结一下 etcd v1/v2 关键特性。



为什么 Kubernetes 使用 etcd?

这张图里，我特别标注出了 Kubernetes 的发布时间点，这个非常关键。我们必须先来说说这个事儿，也就是 Kubernetes 和 etcd 的故事。

2014 年 6 月，Google 的 Kubernetes 项目诞生了，我们前面所讨论到 Go 语言编写、etcd 高可用、Watch 机制、CAS、TTL 等特性正是 Kubernetes 所需要的，它早期的 0.4 版本，使用的正是 etcd v0.2 版本。

Kubernetes 是如何使用 etcd v2 这些特性的呢？举几个简单小例子。

当你使用 Kubernetes 声明式 API 部署服务的时候，Kubernetes 的控制器通过 etcd Watch 机制，会实时监听资源变化事件，对比实际状态与期望状态是否一致，并采取协调动作使其一致。Kubernetes 更新数据的时候，通过 CAS 机制保证并发场景下的原子更新，并通过对 key 设置 TTL 来存储 Event 事件，提升 Kubernetes 集群的可观测性，基于 TTL 特性，Event 事件 key 到期后可自动删除。

Kubernetes 项目使用 etcd，除了技术因素也与当时的商业竞争有关。CoreOS 是 Kubernetes 容器生态圈的核心成员之一。

当时 Docker 容器浪潮正席卷整个开源技术社区，CoreOS 也将容器集成到自家产品中。一开始与 Docker 公司还是合作伙伴，然而 Docker 公司不断强化 Docker 的 PaaS 平台能力，强势控制 Docker 社区，这与 CoreOS 核心商业战略出现了冲突，也损害了 Google、RedHat 等厂商的利益。

最终 CoreOS 与 Docker 分道扬镳，并推出了 rkt 项目来对抗 Docker，然而此时 Docker 已深入人心，CoreOS 被 Docker 全面压制。

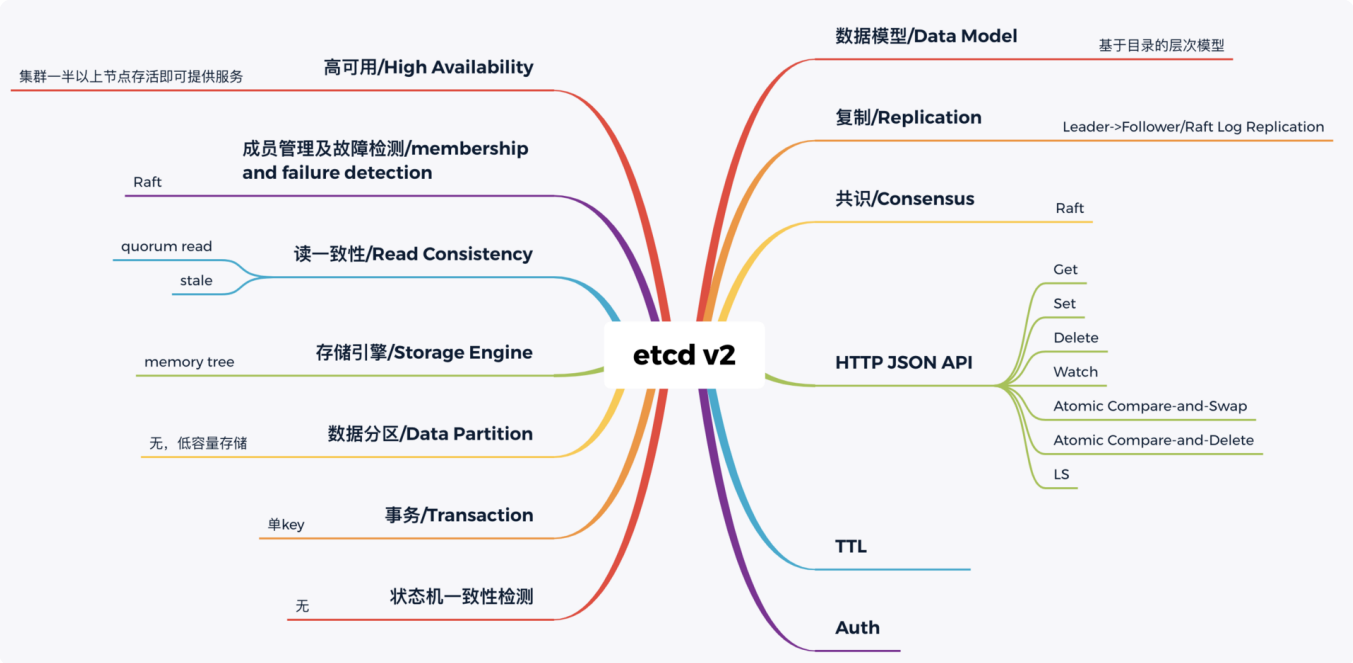
以 Google、RedHat 为首的阵营，基于 Google 多年的大规模容器管理系统 Borg 经验，结合社区的建议和实践，构建以 Kubernetes 为核心的容器生态圈。相比 Docker 的垄断、独裁，Kubernetes 社区推行的是民主、开放原则，Kubernetes 每一层都可以通过插件化扩展，在 Google、RedHat 的带领下不断发展壮大，etcd 也进入了快速发展期。

在 2015 年 1 月，CoreOS 发布了 etcd 第一个稳定版本 2.0，支持了 quorum read，提供了严格的线性一致性读能力。7 月，基于 etcd 2.0 的 Kubernetes 第一个生产环境可用版本 v1.0.1 发布了，Kubernetes 开始了新的里程碑的发展。

etcd v2 在社区获得了广泛关注，GitHub star 数在 2015 年 6 月就高达 6000+，超过 500 个项目使用，被广泛应用于配置存储、服务发现、主备选举等场景。

下图我从构建分布式系统的核心要素角度，给你总结了 etcd v2 核心技术点。无论是 NoSQL 存储还是 SQL 存储、文档存储，其实大家要解决的问题都是类似的，基本就是图中总结的数据模型、复制、共识算法、API、事务、一致性、成员故障检测等方面。

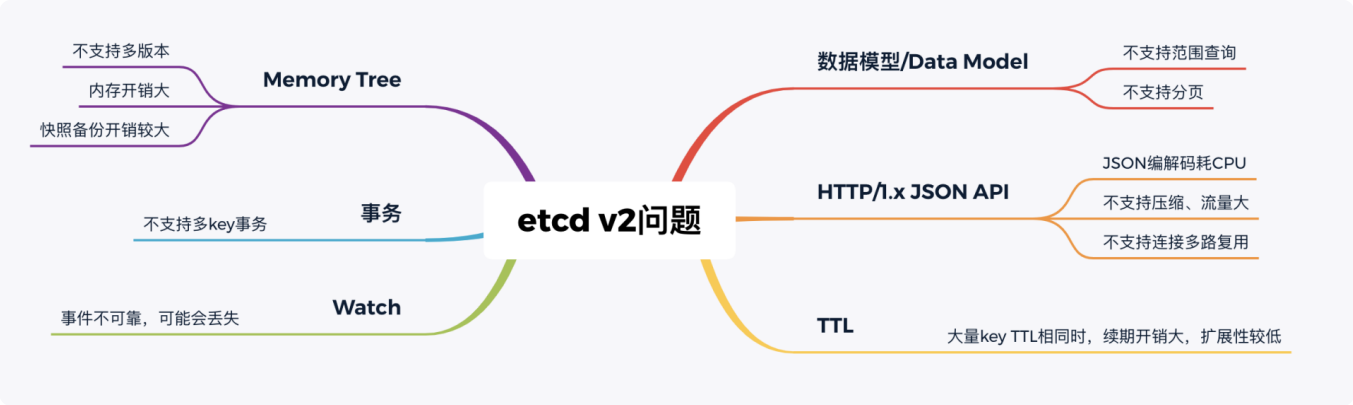
希望通过此图帮助你了解从 0 到 1 如何构建、学习一个分布式系统，要解决哪些技术点，在心中有个初步认识，后面的课程中我会再深入介绍。



etcd v3 诞生

然而随着 Kubernetes 项目不断发展，v2 版本的瓶颈和缺陷逐渐暴露，遇到了若干性能和稳定性问题，Kubernetes 社区呼吁支持新的存储、批评 etcd 不可靠的声音开始不断出现。

具体有哪些问题呢？我给你总结了如下图：



下面我分别从功能局限性、Watch 事件的可靠性、性能、内存开销来分别给你剖析 etcd v2 的问题。

首先是**功能局限性问题。**它主要是指 etcd v2 不支持范围和分页查询、不支持多 key 事务。

第一，etcd v2 不支持范围查询和分页。分页对于数据较多的场景是必不可少的。在 Kubernetes 中，在集群规模增大后，Pod、Event 等资源可能会出现数千个以上，但是 etcd v2 不支持分页，不支持范围查询，大包等 expensive request 会导致严重的性能乃至雪崩问题。

第二，etcd v2 不支持多 key 事务。在实际转账等业务场景中，往往我们需要在一个事务中同时更新多个 key。

然后是 **Watch 机制可靠性问题**。Kubernetes 项目严重依赖 etcd Watch 机制，然而 etcd v2 是内存型、不支持保存 key 历史版本的数据库，只在内存中使用滑动窗口保存了最近的 1000 条变更事件，当 etcd server 写请求较多、网络波动时等场景，很容易出现事件丢失问题，进而又触发 client 数据全量拉取，产生大量 expensive request，甚至导致 etcd 雪崩。

其次是**性能瓶颈问题**。etcd v2 早期使用了简单、易调试的 HTTP/1.x API，但是随着 Kubernetes 支撑的集群规模越来越大，HTTP/1.x 协议的瓶颈逐渐暴露出来。比如集群规模大时，由于 HTTP/1.x 协议没有压缩机制，批量拉取较多 Pod 时容易导致 APIServer 和 etcd 出现 CPU 高负载、OOM、丢包等问题。

另一方面，etcd v2 client 会通过 HTTP 长连接轮询 Watch 事件，当 watcher 较多的时候，因 HTTP/1.x 不支持多路复用，会创建大量的连接，消耗 server 端过多的 socket 和内存资源。

同时 etcd v2 支持为每个 key 设置 TTL 过期时间，client 为了防止 key 的 TTL 过期后被删除，需要周期性刷新 key 的 TTL。

实际业务中很有可能若干 key 拥有相同的 TTL，可是在 etcd v2 中，即使大量 key TTL 一样，你也需要分别为每个 key 发起续期操作，当 key 较多的时候，这会显著增加集群负载、导致集群性能显著下降。

最后是**内存开销问题。**etcd v2 在内存维护了一颗树来保存所有节点 key 及 value。在数据量场景略大的场景，如配置项较多、存储了大量 Kubernetes Events， 它会导致较大的内存开销，同时 etcd 需要定时把全量内存树持久化到磁盘。这会消耗大量的 CPU 和磁盘 I/O 资源，对系统的稳定性造成一定影响。

为什么 etcd v2 有以上若干问题，Consul 等其他竞品依然没有被 Kubernetes 支持呢?

一方面当时包括 Consul 在内，没有一个开源项目是十全十美完全满足 Kubernetes 需求。而 CoreOS 团队一直在聆听社区的声音并积极改进，解决社区的痛点。用户吐槽 etcd 不稳定，他们就设计实现自动化的测试方案，模拟、注入各类故障场景，及时发现修复 Bug，以提升 etcd 稳定性。

另一方面，用户吐槽性能问题，针对 etcd v2 各种先天性缺陷问题，他们从 2015 年就开始设计、实现新一代 etcd v3 方案去解决以上痛点，并积极参与 Kubernetes 项目，负责 etcd v2 到 v3 的存储引擎切换，推动 Kubernetes 项目的前进。同时，设计开发通用压测工具、输出 Consul、ZooKeeper、etcd 性能测试报告，证明 etcd 的优越性。

etcd v3 就是为了解决以上稳定性、扩展性、性能问题而诞生的。

在内存开销、Watch 事件可靠性、功能局限上，它通过引入 B-tree、boltdb 实现一个 MVCC 数据库，数据模型从层次型目录结构改成扁平的 key-value，提供稳定可靠的事件通知，实现了事务，支持多 key 原子更新，同时基于 boltdb 的持久化存储，显著降低了 etcd 的内存占用、避免了 etcd v2 定期生成快照时的昂贵的资源开销。

性能上，首先 etcd v3 使用了 gRPC API，使用 protobuf 定义消息，消息编解码性能相比 JSON 超过 2 倍以上，并通过 HTTP/2.0 多路复用机制，减少了大量 watcher 等场景下的连接数。

其次使用 Lease 优化 TTL 机制，每个 Lease 具有一个 TTL，相同的 TTL 的 key 关联一个 Lease，Lease 过期的时候自动删除相关联的所有 key，不再需要为每个 key 单独续期。

最后是 etcd v3 支持范围、分页查询，可避免大包等 expensive request。

2016 年 6 月，etcd 3.0 诞生，随后 Kubernetes 1.6 发布，默认启用 etcd v3，助力 Kubernetes 支撑 5000 节点集群规模。

下面的时间轴图，我给你总结了 etcd3 重要特性及版本发布时间。从图中你可以看出，从 3.0 到未来的 3.5，更稳、更快是 etcd 的追求目标。

从 2013 年发布第一个版本 v0.1 到今天的 3.5.0-pre，从 v2 到 v3，etcd 走过了 7 年的历程，etcd 的稳定性、扩展性、性能不断提升。

发展到今天，在 GitHub 上 star 数超过 34K。在 Kubernetes 的业务场景磨炼下它不断成长，走向稳定和成熟，成为技术圈众所周知的开源产品，而 **v3 方案的发布，也标志着 etcd 进入了技术成熟期，成为云原生时代的首选元数据存储产品。**

小结

最后我们来小结下今天的内容，我们从如下几个方面介绍了 etcd 的前世今生，并在过程中详细解读了为什么 Kubernetes 使用 etcd：

etcd 诞生背景， etcd v2 源自 CoreOS 团队遇到的服务协调问题。

etcd 目标，我们通过实际业务场景分析，得到理想中的协调服务核心目标：高可用、数据一致性、Watch、良好的可维护性等。而在 CoreOS 团队看来，高可用、可维护性、适配云、简单的 API、良好的性能对他们而言是非常重要的，ZooKeeper 无法满足所有诉求，因此决定自己构建一个分布式存储服务。

介绍了 v2 基于目录的层级数据模型和 API，并从分布式系统的角度给你详细总结了 etcd v2 技术点。etcd 的高可用、Watch 机制与 Kubernetes 期望中的元数据存储是匹配的。etcd v2 在 Kubernetes 的带动下，获得了广泛的应用，但也出现若干性能和稳定性、功能不足问题，无法满足 Kubernetes 项目发展的需求。

CoreOS 团队未雨绸缪，从问题萌芽时期就开始构建下一代 etcd v3 存储模型，分别从性能、稳定性、功能上等成功解决了 Kubernetes 发展过程中遇到的瓶颈，也捍卫住了作为 Kubernetes 存储组件的地位。

希望通过今天的介绍， 让你对 etcd 为什么有 v2 和 v3 两个大版本，etcd 如何从 HTTP/1.x API 到 gRPC API、单版本数据库到多版本数据库、内存树到 boltdb、TTL 到 Lease、单 key 原子更新到支持多 key 事务的演进过程有个清晰了解。希望你能有所收获，在后续的课程中我会和你深入讨论各个模块的细节。

思考题

最后，我给你留了一个思考题。分享一下在你的项目中，你主要使用的是哪个 etcd 版本来解决什么问题呢？使用的 etcd v2 API 还是 v3 API 呢？在这过程中是否遇到过什么问题？

感谢你的阅读，欢迎你把思考和观点写在留言区，也欢迎你把这篇文章分享给更多的朋友一起阅读。

15 人觉得很赞 | 提建议

上一篇 开篇词 | 为什么你要学习etcd?

下一篇 02 | 基础架构：etcd一个读请求是如何执行的?

精选留言 (22)

写留言

jeffery
2021-01-20

1.6版本默认自带v3版本，etcd 云原生的数据首选存储产品，老师能说下etcd 和reids 的区别吗？谢谢

作者回复: Redis是吧，区别挺大的，文章中我用了一副思维导图从数据复制、数据分片、存储引擎、API等方面总结etcd v2技术点，我简单从这些方面给你对比一下，数据复制上Redis是主备异步复制、etcd使用的是Raft，前者可能会丢数据，为了保证读写一致性，etcd读写性能相比Redis差距比较大。数据分片上Redis有各种集群版解决方案，可以承载上T数据，存储的一般是用户数据，而etcd定位是个低容量的关键元数据存储，db大小一般不超过8g。存储引擎和API上Redis内存实现了各种丰富数据结构，而etcd仅是kv API, 使用的是持久化存储boltdb。

10

Young
2021-01-21

目前在使用etcd v3, 高可用，强一致性键值存储。关于etcd集群异地容灾同步目前用的make mirror，有更好的方案吗？ 哈哈😃

展开

作者回复: 也有可能在部分场景出现不一致哈，后面实践篇会和大家分析。异地容灾建议使用raft learner特性，添加learner节点，make mirror是基于watch机制的，稳定性、可靠性等相比lear

ner要差点，但是目前社区对learner限制太多，不支持快照等，我们自己做了定制化，大家使用上要等等，估计3.5版本才比较好用

1 4

Coder
2021-01-21

精彩，读起来生动有趣，一文就让我了解了etcd的发展历史与k8s关系，为老师打call！

3

chapin
2021-01-21

头排看戏。

展开

2

SecKiki
2021-01-20

学习了，etcd与k8s 绝配

展开

作者回复: 嗯，etcd与k8s相互影响、促进，etcd社区任何核心特性首先也是考虑是否有益于k8s场景等，新版本发布，也是拿k8s做性能稳定性测试

3

爪哇夜未眠
2021-01-20

挺好的文章

展开

作者回复: 感谢你的留言与支持

2

mckee
2021-01-26

k8s list pod不带rev情况下，etcdserver key range会造成cpu和mem瞬间飙高，需要读

取所有kv并排序

作者回复: 嗯，list pod太恐怖了，哈哈，线上大部分异常一般都是list pod，configmap，crd

2

hxy
2021-01-23

zookeeper的rpc api是thrift，这个感觉说错了吧，zk用的应该是jute

作者回复: 感谢你的细心和反馈，我查阅etcd官方资料写得也是的确有误，之前以为是很久之前zk使用的rpc api，看了下commit记录貌似一直都是juite，后面我们修正下，谢谢

2

ZHU ~ JM
2021-01-21

老师，我们在使用K8s和etcd的时候经常出现注册和发现超时的情况，这是为什么啊

作者回复: 这就涉及troubleshooting，通过etcd metric，trace特性，etcd log等各种工具分析才行，后面会有介绍，你可以看看超时时etcd的metric监控和日志

1

atom
2021-01-21

非常喜欢后面etcd局限性的分析 觉得很不错

展开

1

后端进阶
2021-01-20

给唐大佬疯狂打Call，以后学习ETCD就跟着唐大佬的课程走就完了！

作者回复: 一起持续学习与进步，后续有什么不清楚的，欢迎继续留言交流

1

po
2021-01-27

etcd v2是没有对数据持久化只在内存？那么etcd重启数据就没了？之前在极客时间张磊的kubernetes上面他说因为raft不需要对etcd持久化，提问了没解答，疑惑中。

作者回复: 有持久化的，定时做快照，把内存树序列号成json保存到磁盘快照文件，启动的时候会加载快照文件，重建

bearlu
2021-01-27

学到了学到了

展开

1

mckee
2021-01-25

老版本的etcd client只维护一个etcd member连接，对于heavy usage会造成负载不均衡

作者回复: 嗯，3.4版本才使用round robin负载均衡算法

Geek_1337
2021-01-23

老师好！想知道在没有像etcd这样watch机制以前，一般数据库是怎么获取数据的更新的，除了不断的轮询还有其他解决思路吗？谢谢。

展开

作者回复: watch篇时我会从设计实现思路上详解，麻烦稍等

范闲
2021-01-22

etcd用来做服务发现，治理，分布式锁，还有配置信息存储很好用。
缺点是异地多活不好用，延迟有点大。

展开

作者回复: 是的，我们使用raft learner节点，跨城容灾，进行了一些定制化改造

2021-01-21

为什么k8s里默认的etcd API版本还是2?

展开

作者回复: k8s使用的是v3，能贴下你说的默认API版本为2的代码吗

AIK

2021-01-21

读了两遍还是感觉很吃力，需要去恶补一下不懂的知识。有个疑问想让老师解惑一下，etcdV1和V2版本并没有ZK社区成熟功能也没有ZK强大，为什么K8S选择了和etcd一起成长，而不是等着etcd各方面完善和强大起来再从ZK切换到etcd?

展开

作者回复: 我认为最主要的是还是Go语言和容器生态，zookeeper当时也没好用的go client库，watch特性也没etcd好用，核心还是文中说得，当时没一个项目满足k8s需求，只有coreos团队不停的推动etcd与k8s相互适配改造，维护k8s etcd代码，让k8s核心开发者有更多精力去开发更重要的特性开发。

Acter

2021-01-21

请问老师，k8s场景etcd v3.3.10版本。由于业务的workflow yaml对象很大，近期已把etcd的max-request-bytes从1.5MiB提高到了5MiB，这对k8s master组件稳定性有何风险吗？该怎么压测呢?

作者回复: 后面实践篇会介绍大value风险，容易导致apiserver和etcd oom，超时，雪崩等，建议创建几千个这样的大对象，通过client-go模拟用户list，按标签get，修改等操作


YWWe(˙)9✧...

2021-01-20

不错不错

展开