三篇文章了解 TiDB 技术内幕 - 谈调度

2017-06-06

为什么要进行调度

先回忆一下 三篇文章了解 TiDB 技术内幕 - 说存储 提到的一些信息，TiKV 集群是 TiDB 数据库的分布式 KV 存储引擎，数据以 Region 为单位进行复制和管理，每个 Region 会有多个 Replica（副本），这些 Replica 会分布在不同的 TiKV 节点上，其中 Leader 负责读/写，Follower 负责同步 Leader 发来的 raft log。了解了这些信息后，请思考下面这些问题：

- 如何保证同一个 Region 的多个 Replica 分布在不同的节点上？更进一步，如果在一台机器上启动多个 TiKV 实例，会有什么问题？
- TiKV 集群进行跨机房部署用于容灾的时候，如何保证一个机房掉线，不会丢失 Raft Group 的多个 Replica？
- 添加一个节点进入 TiKV 集群之后，如何将集群中其他节点上的数据搬过来？
- 当一个节点掉线时，会出现什么问题？整个集群需要做什么事情？如果节点只是短暂掉线（重启服务），那么如何处理？如果节点是长时间掉线（磁盘故障，数据全部丢失），需要如何处理？
- 假设集群需要每个 Raft Group 有 N 个副本，那么对于单个 Raft Group 来说，Replica 数量可能会不够多（例如节点掉线，失去副本），也可能会过于多（例如掉线的节点又回复正常，自动加入集群）。那么如何调节 Replica 个数？
- 读/写都是通过 Leader 进行，如果 Leader 只集中在少量节点上，会对集群有什么影响？
- 并不是所有的 Region 都被频繁的访问，可能访问热点只在少数几个 Region，这个时候我们需要做什么？

- 集群在做负载均衡的时候，往往需要搬迁数据，这种数据的迁移会不会占用大量的网络带宽、磁盘 IO 以及 CPU？进而影响在线服务？

这些问题单独拿出可能都能找到简单的解决方案，但是混杂在一起，就不太好解决。有的问题貌似只需要考虑单个 Raft Group 内部的情况，比如根据副本数量是否足够多来决定是否需要添加副本。但是实际上这个副本添加在哪里，是需要考虑全局的信息。整个系统也是在动态变化，Region 分裂、节点加入、节点失效、访问热点变化等情况会不断发生，整个调度系统也需要在动态中不断向最优状态前进，如果没有一个掌握全局信息，可以对全局进行调度，并且可以配置的组件，就很难满足这些需求。因此我们需要一个中心节点，来对系统的整体状况进行把控和调整，所以有了 PD 这个模块。

调度的需求

上面罗列了一大堆问题，我们先进行分类和整理。总体来看，问题有两大类：

作为一个分布式高可用存储系统，必须满足的需求，包括四种：

- 副本数量不能多也不能少
- 副本需要分布在不同的机器上
- 新加节点后，可以将其他节点上的副本迁移过来
- 节点下线后，需要将该节点的数据迁移走

作为一个良好的分布式系统，需要优化的地方，包括：

- 维持整个集群的 Leader 分布均匀
- 维持每个节点的储存容量均匀
- 维持访问热点分布均匀
- 控制 Balance 的速度，避免影响在线服务
- 管理节点状态，包括手动上线/下线节点，以及自动下线失效节点

满足第一类需求后，整个系统将具备多副本容错、动态扩容/缩容、容忍节点掉线以及自动错误恢复的功能。满足第二类需求后，可以使得整体系统的负载更加均匀、且可以方便的管理。

为了满足这些需求，首先我们需要收集足够的信息，比如每个节点的状态、每个 Raft Group 的信息、业务访问操作的统计等；其次需要设置一些策略，PD 根据这些信息以及调度的策略，制定出尽

量满足前面所述需求的调度计划；最后需要一些基本的操作，来完成调度计划。

调度的基本操作

我们先来介绍最简单的一点，也就是调度的基本操作，也就是为了满足调度的策略，我们有哪些功能可以用。这是整个调度的基础，了解了手里有什么样的锤子，才知道用什么样的姿势去砸钉子。

上述调度需求看似复杂，但是整理下来最终落地的无非是下面三件事：

- 增加一个 Replica
- 删除一个 Replica
- 将 Leader 角色在一个 Raft Group 的不同 Replica 之间 transfer

刚好 Raft 协议能够满足这三种需求，通过 AddReplica、RemoveReplica、TransferLeader 这三个命令，可以支撑上述三种基本操作。

信息收集

调度依赖于整个集群信息的收集，简单来说，我们需要知道每个 TiKV 节点的状态以及每个 Region 的状态。TiKV 集群会向 PD 汇报两类消息：

每个 TiKV 节点会定期向 PD 汇报节点的整体信息

TiKV 节点（Store）与 PD 之间存在心跳包，一方面 PD 通过心跳包检测每个 Store 是否存活，以及是否有新加入的 Store；另一方面，心跳包中也会携带这个 Store 的状态信息 ，主要包括：

- 总磁盘容量
- 可用磁盘容量
- 承载的 Region 数量
- 数据写入速度
- 发送/接受的 Snapshot 数量（Replica 之间可能会通过 Snapshot 同步数据）
- 是否过载

- 标签信息（标签是具备层级关系的一系列 Tag）

每个 Raft Group 的 Leader 会定期向 PD 汇报信息

每个 Raft Group 的 Leader 和 PD 之间存在心跳包，用于汇报这个 Region 的状态，主要包括下面几点信息：

- Leader 的位置
- Followers 的位置
- 掉线 Replica 的个数
- 数据写入/读取的速度

PD 不断的通过这两类心跳消息收集整个集群的信息，再以这些信息作为决策的依据。除此之外，PD 还可以通过管理接口接受额外的信息，用来做更准确的决策。比如当某个 Store 的心跳包中断的时候，PD 并不能判断这个节点是临时失效还是永久失效，只能经过一段时间的等待（默认是 30 分钟），如果一直没有心跳包，就认为是 Store 已经下线，再决定需要将这个 Store 上面的 Region 都调度走。但是有的时候，是运维人员主动将某台机器下线，这个时候，可以通过 PD 的管理接口通知 PD 该 Store 不可用，PD 就可以马上判断需要将这个 Store 上面的 Region 都调度走。

调度的策略

PD 收集了这些信息后，还需要一些策略来制定具体的调度计划。

一个 Region 的 Replica 数量正确

当 PD 通过某个 Region Leader 的心跳包发现这个 Region 的 Replica 数量不满足要求时，需要通过 Add/Remove Replica 操作调整 Replica 数量。出现这种情况的可能原因是：

- 某个节点掉线，上面的数据全部丢失，导致一些 Region 的 Replica 数量不足
- 某个掉线节点又恢复服务，自动接入集群，这样之前已经补足了 Replica 的 Region 的 Replica 数量多过，需要删除某个 Replica
- 管理员调整了副本策略，修改了 max-replicas 的配置

一个 Raft Group 中的多个 Replica 不在同一个位置

注意第二点，『一个 Raft Group 中的多个 Replica 不在同一个位置』，这里用的是『同一个位置』而不是『同一个节点』。在一般情况下，PD 只会保证多个 Replica 不落在一个节点上，以避免单个节点失效导致多个 Replica 丢失。在实际部署中，还可能出现下面这些需求：

- 多个节点部署在同一台物理机器上
- TiKV 节点分布在多个机架上，希望单个机架掉电时，也能保证系统可用性
- TiKV 节点分布在多个 IDC 中，希望单个机房掉电时，也能保证系统可用

这些需求本质上都是某一个节点具备共同的位置属性，构成一个最小的容错单元，我们希望这个单元内部不会存在一个 Region 的多个 Replica。这个时候，可以给节点配置 lables 并且通过在 PD 上配置 location-labels 来指明哪些 lable 是位置标识，需要在 Replica 分配的时候尽量保证不会有一个 Region 的多个 Replica 所在结点有相同的位置标识。

副本在 Store 之间的分布均匀分配

前面说过，每个副本中存储的数据容量上限是固定的，所以我们维持每个节点上面，副本数量的均衡，会使得总体的负载更均衡。

Leader 数量在 Store 之间均匀分配

Raft 协议要读取和写入都通过 Leader 进行，所以计算的负载主要在 Leader 上面，PD 会尽可能将 Leader 在节点间分散开。

访问热点数量在 Store 之间均匀分配

每个 Store 以及 Region Leader 在上报信息时携带了当前访问负载的信息，比如 Key 的读取/写入速度。PD 会检测出访问热点，且将其在节点之间分散开。

各个 Store 的存储空间占用大致相等

每个 Store 启动的时候都会指定一个 Capacity 参数，表明这个 Store 的存储空间上限，PD 在做调度的时候，会考虑节点的存储空间剩余量。

控制调度速度，避免影响在线服务

调度操作需要耗费 CPU、内存、磁盘 IO 以及网络带宽，我们需要避免对线上服务造成太大影响。PD 会对当前正在进行的操作数量进行控制，默认的速度控制是比较保守的，如果希望加快调度(比如已经停服务升级，增加新节点，希望尽快调度)，那么可以通过 pd-ctl 手动加快调度速度。

支持手动下线节点

当通过 pd-ctl 手动下线节点后，PD 会在一定的速率控制下，将节点上的数据调度走。当调度完成后，就会将这个节点置为下线状态。

调度的实现

了解了上面这些信息后，接下来我们看一下整个调度的流程。

PD 不断的通过 Store 或者 Leader 的心跳包收集信息，获得整个集群的详细数据，并且根据这些信息以及调度策略生成调度操作序列，每次收到 Region Leader 发来的心跳包时，PD 都会检查是否有对这个 Region 待进行的操作，通过心跳包的回复消息，将需要进行的操作返回给 Region Leader，并在后面的心跳包中监测执行结果。注意这里的操作只是给 Region Leader 的建议，并不保证一定能得到执行，具体是否会执行以及什么时候执行，由 Region Leader 自己根据当前自身状态来定。

总结

本篇文章讲的东西，大家可能平时很少会在其他文章中看到，每一个设计都有背后的考量，希望大家能了解到一个分布式存储系统在做调度的时候，需要考虑哪些东西，如何将策略、实现进行解耦，更灵活的支持策略的扩展。

至此三篇文章已经讲完，希望大家能够对整个 TiDB 的基本概念和实现原理有了解，后续我们还会写更多的文章，从架构以及代码级别介绍 TiDB 的更多内幕。如果大家有问题，欢迎发邮件到 shenli@pingcap.com 进行交流。

> 相关阅读： 三篇文章了解 *TiDB* 技术内幕 - 说存储 ； 三篇文章了解 *TiDB* 技术内幕 - 说计算

关于我们

公司概况

发展历程

新闻中心

市场活动

加入我们

隐私声明

安全合规

资源中心

社区

TiDB 文档

TiDB in Action

快速上手指南

社区问答-AskTUG

博客

GitHub

PingCAP Education

联系我们

商务咨询

4006790886

010-58400041

info@pingcap.com

前台总机

010-53326356

媒体合作

pr@pingcap.com

PingCAP 公司

PingCAP 是业界领先的企业级开源分布式数据库企业，提供包括开源分布式数据库产品、解决方案与咨询、技术支持与培训认证服务，致力于为全球行业用户提供稳定高效、安全可靠、开放兼容的新型数据基础设施，解放企业生产力，加速企业数字化转型升级。

联系我们

 京 ICP 备 16046278 号 - 2

京公网安备 11010802035112 号