三篇文章了解 TiDB 技术内幕 - 说存储

申砾 2017-05-15

引言

数据库、操作系统和编译器并称为三大系统，可以说是整个计算机软件的基石。其中数据库更靠近应用层，是很多业务的支撑。这一领域经过了几十年的发展，不断的有新的进展。

很多人用过数据库，但是很少有人实现过一个数据库，特别是实现一个分布式数据库。了解数据库的实现原理和细节，一方面可以提高个人技术，对构建其他系统有帮助，另一方面也有利于用好数据库。

研究一门技术最好的方法是研究其中一个开源项目，数据库也不例外。单机数据库领域有很多很好的开源项目，其中 MySQL 和 PostgreSQL 是其中知名度最高的两个，不少同学都看过这两个项目的代码。但是分布式数据库方面，好的开源项目并不多。 TiDB 目前获得了广泛的关注，特别是一些技术爱好者，希望能够参与这个项目。由于分布式数据库自身的复杂性，很多人并不能很好的理解整个项目，所以我希望能写一些文章，自顶向下，由浅入深，讲述 TiDB 的一些技术原理，包括用户可见的技术以及大量隐藏在 SQL 界面后用户不可见的技术点。

保存数据

数据库最根本的功能是能把数据存下来，所以我们从这里开始。

保存数据的方法很多，最简单的方法是直接在内存中建一个数据结构，保存用户发来的数据。比如用一个数组，每当收到一条数据就向数组中追加一条记录。这个方案十分简单，能满足最基本，并且性能肯定会很好，但是除此之外却是漏洞百出，其中最大的问题是数据完全在内存中，一旦停机或者是服务重启，数据就会永久丢失。

为了解决数据丢失问题，我们可以把数据放在非易失存储介质（比如硬盘）中。改进的方案是在磁盘上创建一个文件，收到一条数据，就在文件中 Append 一行。OK，我们现在有了一个能持久化存储数据的方案。但是还不够好，假设这块磁盘出现了坏道呢？我们可以做 RAID （Redundant Array of Independent Disks），提供单机冗余存储。如果整台机器都挂了呢？比如出现了火灾，RAID 也保不住这些数据。我们还可以将存储改用网络存储，或者是通过硬件或者软件进行存储复制。到这里似乎我们已经解决了数据安全问题，可以松一口气了。But，做复制过程中是否能保证副本之间的一致性？也就是在保证数据不丢的前提下，还要保证数据不错。保证数据不丢不错只是一项最基本的要求，还有更多令人头疼的问题等待解决：

- 能否支持跨数据中心的容灾？
- 写入速度是否够快？
- 数据保存下来后，是否方便读取？

- 保存的数据如何修改？如何支持并发的修改？
- 如何原子地修改多条记录？

这些问题每一项都非常难，但是要做一个优秀的数据存储系统，必须要解决上述的每一个难题。 为了解决数据存储问题，我们开发了 TiKV 这个项目。接下来我向大家介绍一下 TiKV 的一些设计思想和基本概念。

Key-Value

作为保存数据的系统，首先要决定的是数据的存储模型，也就是数据以什么样的形式保存下来。TiKV 的选择是 Key-Value 模型，并且提供有序遍历方法。简单来讲，可以将 TiKV 看做一个巨大的 Map，其中 Key 和 Value 都是原始的 Byte 数组，在这个 Map 中，Key 按照 Byte 数组总的原始二进制比特位比较顺序排列。 大家这里需要对 TiKV 记住两点：

1. 这是一个巨大的 Map，也就是存储的是 Key-Value pair
2. 这个 Map 中的 Key-Value pair 按照 Key 的二进制顺序有序，也就是我们可以 Seek 到某一个 Key 的位置，然后不断的调用 Next 方法以递增的顺序获取比这个 Key 大的 Key-Value

讲了这么多，有人可能会问了，这里讲的存储模型和 SQL 中表是什么关系？在这里有一件重要的事情要说四遍：

这里的存储模型和 SQL 中的 Table 无关！ 这里的存储模型和 SQL 中的 Table 无关！ 这里的存储模型和 SQL 中的 Table 无关！ 这里的存储模型和 SQL 中的 Table 无关！

现在让我们忘记 SQL 中的任何概念，专注于讨论如何实现 TiKV 这样一个高性能高可靠性的巨大的（分布式的） Map。

RocksDB

任何持久化的存储引擎，数据终归要保存在磁盘上，TiKV 也不例外。但是 TiKV 没有选择直接向磁盘上写数据，而是把数据保存在 RocksDB 中，具体的数据落地由 RocksDB 负责。这个选择的原因是开发一个单机存储引擎工作量很大，特别是要做一个高性能的单机引擎，需要做各种细致的优化，而 RocksDB 是一个非常优秀的开源的单机存储引擎，可以满足我们对单机引擎的各种要求，而且还有 Facebook 的团队在做持续的优化，这样我们只投入很少的精力，就能享受到一个十分强大且在不断

进步的单机引擎。当然，我们也为 RocksDB 贡献了一些代码，希望这个项目能越做越好。这里可以简单的认为 RocksDB 是一个单机的 Key-Value Map。

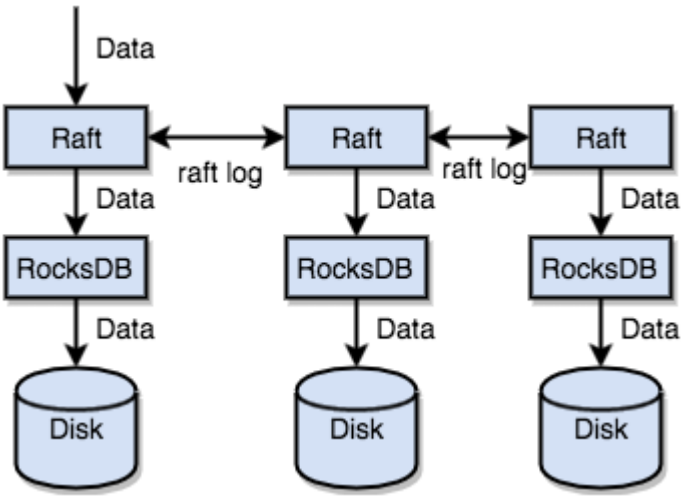
Raft

好了，万里长征第一步已经迈出去了，我们已经为数据找到一个高效可靠的本地存储方案。俗话说，万事开头难，然后中间难，最后结尾难。接下来我们面临一件更难的事情：如何保证单机失效的情况下，数据不丢失，不出错？简单来说，我们需要想办法把数据复制到多台机器上，这样一台机器挂了，我们还有其他的机器上的副本；复杂来说，我们还需要这个复制方案是可靠、高效并且能处理副本失效的情况。听上去比较难，但是好在我们有 Raft 协议。Raft 是一个一致性算法，它和 Paxos 等价，但是更加易于理解。 Raft 的论文 ，感兴趣的可以看一下。本文只会对 Raft 做一个简要的介绍，细节问题可以参考论文。另外提一点，Raft 论文只是一个基本方案，严格按照论文实现，性能会很差，我们对 Raft 协议的实现做了大量的优化，具体的优化细节可参考我司首席架构师 tangliu 同学的《TiKV 源码解析系列 - Raft 的优化》 这篇文章。

Raft 是一个一致性协议，提供几个重要的功能：

1. Leader 选举
2. 成员变更
3. 日志复制

TiKV 利用 Raft 来做数据复制，每个数据变更都会落地为一条 Raft 日志，通过 Raft 的日志复制功能，将数据安全可靠地同步到 Group 的多数节点中。



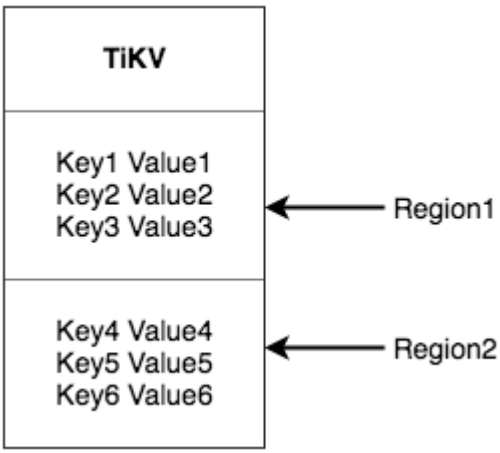
到这里我们总结一下，通过单机的 RocksDB，我们可以将数据快速地存储在磁盘上；通过 Raft，我们可以将数据复制到多台机器上，以防单机失效。数据的写入是通过 Raft 这一层的接口写入，而不是直接写 RocksDB。通过实现 Raft，我们拥有了一个分布式的 KV，现在再也不用担心某台机器挂掉了。

Region

讲到这里，我们可以提到一个 **非常重要的概念**：**Region**。这个概念是理解后续一系列机制的基础，请仔细阅读这一节。

前面提到，我们将 TiKV 看做一个巨大的有序的 KV Map，那么为了实现存储的水平扩展，我们需要将数据分散在多台机器上。这里提到的数据分散在多台机器上和 Raft 的数据复制不是一个概念，在这一节我们先忘记 Raft，假设所有的数据都只有一个副本，这样更容易理解。

对于一个 KV 系统，将数据分散在多台机器上有两种比较典型的方案：一种是按照 Key 做 Hash，根据 Hash 值选择对应的存储节点；另一种是分 Range，某一段连续的 Key 都保存在一个存储节点上。TiKV 选择了第二种方式，将整个 Key-Value 空间分成很多段，每一段是一系列连续的 Key，我们将每一段叫做一个 **Region**，并且我们会尽量保持每个 Region 中保存的数据不超过一定的大小(这个大小可以配置，目前默认是 96mb)。每一个 Region 都可以用 StartKey 到 EndKey 这样一个左闭右开区间来描述。



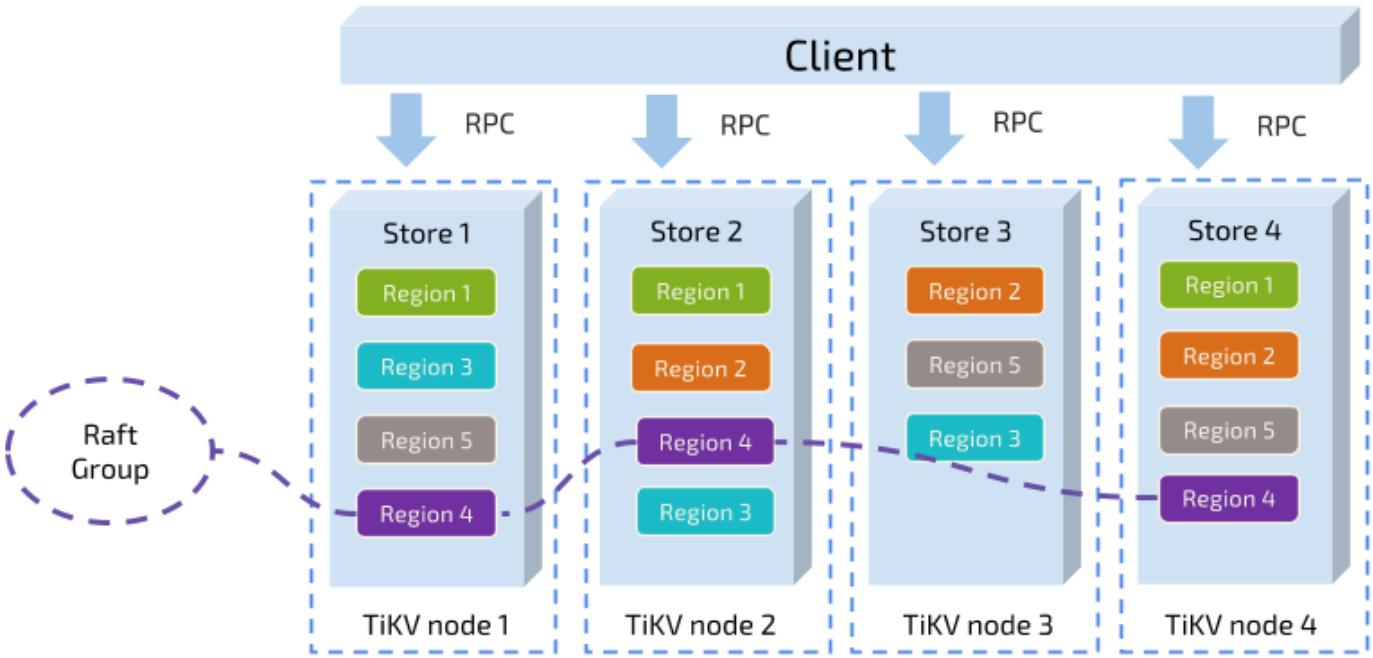
注意，这里的 Region 还是和 SQL 中的表没什么关系！ 请各位继续忘记 SQL，只谈 KV。 将数据划分成 Region 后，我们将会做 **两件重要的事情**：

- 以 Region 为单位，将数据分散在集群中所有的节点上，并且尽量保证每个节点上服务的 Region 数量差不多
- 以 Region 为单位做 Raft 的复制和成员管理

这两点非常重要，我们一点一点来说。

先看第一点，数据按照 Key 切分成很多 Region，每个 Region 的数据只会保存在一个节点上面。我们的系统会有一个组件来负责将 Region 尽可能均匀的散布在集群中所有的节点上，这样一方面实现了存储容量的水平扩展（增加新的结点后，会自动将其他节点上的 Region 调度过来），另一方面也实现了负载均衡（不会出现某个节点有很多数据，其他节点上没什么数据的情况）。同时为了保证上层客户端能够访问所需要的数据，我们的系统中也会有一个组件记录 Region 在节点上面的分布情况，也就是通过任意一个 Key 就能查询到这个 Key 在哪个 Region 中，以及这个 Region 目前在哪个节点上。至于是哪个组件负责这两项工作，会在后续介绍。

对于第二点，TiKV 是以 Region 为单位做数据的复制，也就是一个 Region 的数据会保存多个副本，我们将每一个副本叫做一个 Replica。Replica 之间是通过 Raft 来保持数据的一致（终于提到了 Raft），一个 Region 的多个 Replica 会保存在不同的节点上，构成一个 Raft Group。其中一个 Replica 会作为这个 Group 的 Leader，其他的 Replica 作为 Follower。所有的读和写都是通过 Leader 进行，再由 Leader 复制给 Follower。 大家理解了 Region 之后，应该可以理解下面这张图：



我们以 Region 为单位做数据的分散和复制，就有了一个分布式的具备一定容灾能力的 KeyValue 系统，不用再担心数据存不下，或者是磁盘故障丢失数据的问题。这已经很 Cool，但是还不够完美，

我们需要更多的功能。

MVCC

很多数据库都会实现多版本控制（MVCC），TiKV 也不例外。设想这样的场景，两个 Client 同时去修改一个 Key 的 Value，如果没有 MVCC，就需要对数据上锁，在分布式场景下，可能会带来性能以及死锁问题。 TiKV 的 MVCC 实现是通过在 Key 后面添加 Version 来实现，简单来说，没有 MVCC 之前，可以把 TiKV 看做这样的：

```
Key1 -> Value
Key2 -> Value
……
KeyN -> Value
```

有了 MVCC 之后，TiKV 的 Key 排列是这样的：

```
Key1-Version3 -> Value
Key1-Version2 -> Value
Key1-Version1 -> Value
……
Key2-Version4 -> Value
Key2-Version3 -> Value
Key2-Version2 -> Value
Key2-Version1 -> Value
……
KeyN-Version2 -> Value
KeyN-Version1 -> Value
……
```

注意，对于同一个 Key 的多个版本，我们把版本号较大的放在前面，版本号小的放在后面（回忆一下 Key-Value 一节我们介绍过的 Key 是有序的排列），这样当用户通过一个 Key + Version 来获取 Value 的时候，可以将 Key 和 Version 构造出 MVCC 的 Key，也就是 Key-Version。然后可以直接 Seek(Key-Version)，定位到第一个大于等于这个 Key-Version 的位置。

事务

TiKV 的事务采用的是 Percolator 模型，并且做了大量的优化。事务的细节这里不详述，大家可以参考论文以及我们的其他文章。这里只提一点，TiKV 的事务采用乐观锁，事务的执行过程中，不会检测写写冲突，只有在提交过程中，才会做冲突检测，冲突的双方中比较早完成提交的会写入成功，另一方会尝试重新执行整个事务。当业务的写入冲突不严重的情况下，这种模型性能会很好，比如随机更新表中某一行的数据，并且表很大。但是如果业务的写入冲突严重，性能就会很差，举一个极端的例子，就是计数器，多个客户端同时修改少量行，导致冲突严重的，造成大量的无效重试。

其他

到这里，我们已经了解了 TiKV 的基本概念和一些细节，理解了这个分布式带事务的 KV 引擎的分层结构以及如何实现多副本容错。下一节会介绍如何在 KV 的存储模型之上，构建 SQL 层。

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

联系我们

 京 ICP 备 16046278 号 - 2

京公网安备 11010802035112 号