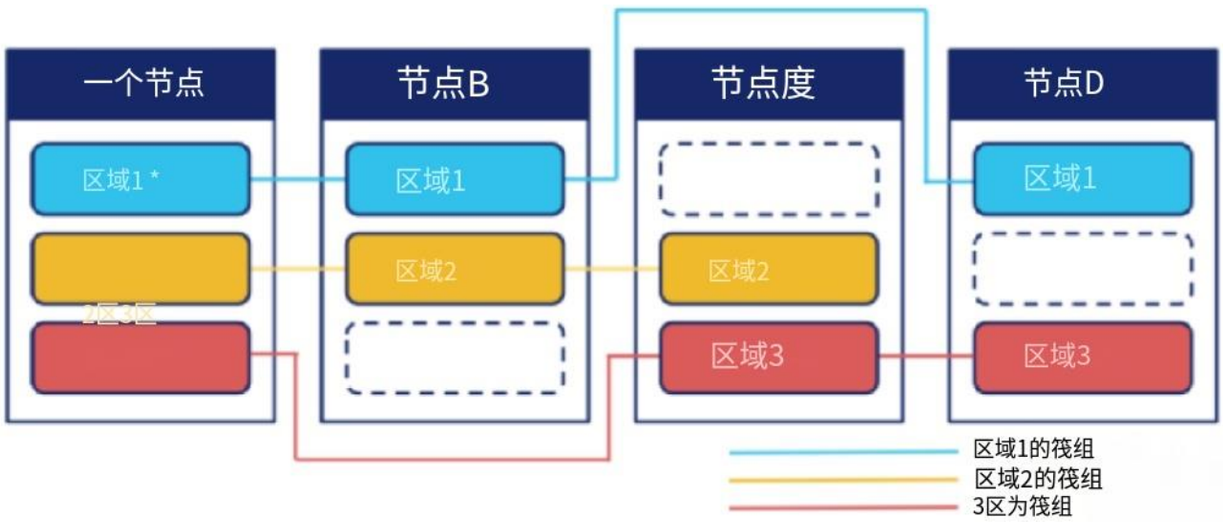
Project3 MultiRaftKV

在 project2 中，你已经基于 Raft 搭建了一个高可用的 kv 服务器，干得好!但还不够，这样的 kv 服务器背后是一个不是无限可扩展的单一 raft 组，每一个写请求都会等到提交后再一一写给 badger，这是保证一致性的关键要求，但也杀了任何并发。



在这个项目中，你将实现一个基于多 raft 的带有 balance scheduler 的 kv 服务器，它由多个 raft 组组成，每个 raft 组负责一个单一的键范围，这里命名为 region，布局将类似于上图。单个区域的请求像以前一样处理，但多个区域可以并发处理请求，这提高了性能，但也带来了一些新的挑战，如平衡每个区域的请求等。

本项目由 3 部分组成，包括:

1.对 Raft 算法实施成员变更和领导变更

2.在 raftstore 上实现 conf 更改和 region 拆分

3.介绍了调度器

部分

在这一部分中，您将对基本的 raft 算法实现成员变更和领导变更，这些功能是后面两个部分所需要的。成员变更，即 conf 变更，用于向 raft 组添加或删除 peer，这可以改变 raft 组的 quorum，所以要小心。领导权变更，即 leader transfer，是用来将领导权转移给另一个 peer，这对于平衡非常有用。

的代码

你需要修改的代码都是关于 raft/raft 的 和 /node 。

需要处理的新消息。而且 conf 的改变和 leader 的转移都是由上层应用触发的，所以你可能想要从 raft/rawnode 开始。走了。

落实领导调动

为了实现 leader 传输，让我们引入两种新的消息类型:MsgT 和 MsgTimeoutNow。要转移领导权，您需要首先调用 raft.Raft.Step 和 MsgTransferLeader

对现任领导的留言，并且为了确保转移成功，现任领导要先检查受让人(即转移目标)的资质，比如:受让人的日志是最新的，等等。如果受让方资质不合格，现任领导可以选择中止转让或者帮助受让方，既然中止不帮助，那就选择帮助受让方吧。如果受让方的日志不是最新的，则
现任领导应向受让人发送 Ms 息，并停止接受新的提议

以防我们最终骑自行车。所以如果转让方是合格的(或者经过现任领导的帮助)，领导应该
立即向 imeoutNow 消息，并在接收到 MsgTimeoutNow 消息后发送

消息无论选举超时与否，受让方都应该立即开始新的选举，有了更高的任期和最新的日志，受让方有很大的机会让现任领导下台，成为新的领导。

实施 conf change

这里你要实现的 Conf change 算法并不是扩展版 Raft 论文中提到的可以一次性添加和/或删除任意节点的联合共识算法，而是只能添加或删除

对等点一个接一个，这更简单，也更容易推理。此外，conf 从调用 leader 的
raft.RawNode.ProposeConfChange 开始
，它将提出一个条目，其中 pb. entry . entrytype 设置为
EntryConfChange，pb. entry . data 设置为输入 pb。ConfChange。当提交类型为
EntryConfChange 的条目时，必须通过 RawNode 应用它。
pb. ApplyConfChange。只有这样，才能根据 pb. conf . addnode 和 raft. raft. removenode 向该筏节点添加

> 提示:
>
> - MsgTransferLeader 消息是本地消息，不是来自网络
> - 。您可以将 MsgTransferLeader 消息的 message .from 设置为受让方(即转移目标)，
> - 以立即开始新的选举，您可以调用 Raft。
> - 调用 pb. confchange . marshal 来获得 pb 的字节表示。ConfChange 并将其放入 pb. entry . data

B 部分

由于 Raft 模块现在支持成员变更和领导变更，在这一部分中，你需要让 TinyKV 基于 a 部分支持这些管理命令，正如你在中看到的

有四种类型的管理命令:

- CompactLog(已经在项目 2 C 部分实现)
- TransferLeader
- ChangePeer
- 分裂

和 Chang 是基于 Raft 支持的领导层变更和

会员的变化。这些将被用作平衡调度器的基本操作步骤。Split 将一个区域分成两个区域，这是 aft 的基础。你会一步一步地实现它们。

的代码

所有的更改都基于 project2 的实现，因此需要修改的代码都是关于 kv/rafts store/

提议调动领导

这一步很简单。作为一个筏命令，TransferLea 一个筏条目。但

上是一个不需要复制到其他对等体的动作，所以你只需要调用 RawNode 的 ，而不是为 Leader 命令 sal(

在 raftstore 中实现 conf 更改

配置文件有两种不同的类型，AddNode 和 de。 是添加一个 Peer

或者从区域中移除一个对等体。要实现 conf change，你应该学习一下术语 poch。 是 metapb 元信息的一部分。地区。 添加或删除 Peer 或拆分，该区域的纪元发生了变化。RegionEpoch 的 conf_ver 在

分裂期间 ConfChan 增加。将用于保证网络隔离下一个区域内两个 leader 的最新区域信息。

你需要让 raftstore 支持处理 conf 变更命令。过程将是:

1. proposal conf change admin 命令通过 prop
2. 日志提交后，需要修改 RegionLocalSt ch 和 Peer

所示。

提示:

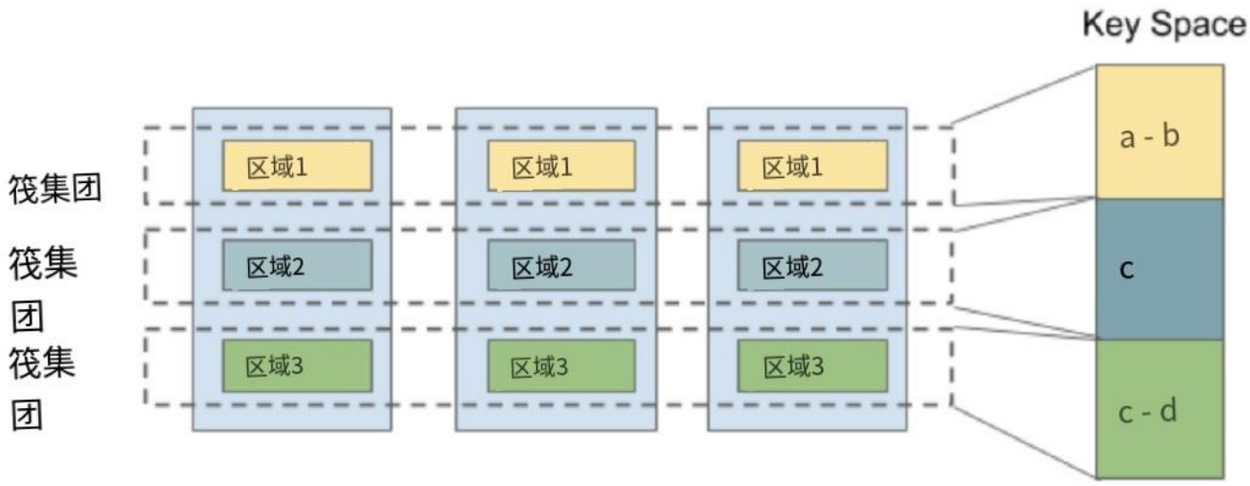
- 对于执行 AddNode，新添加的 Peer 将由 leader 心跳创建，请检查 storeWorker 的 maybeCreatePeer()。在那个时候，这个 Peer 是未初始化的我们不知道它所在区域的信息，所以我们用 0 来初始化它的对数项和索

  那么 leader 就会知道这个 Follower 没有数据(存在从 0 到 5 的日志差距)，它会直接给这个 Follower 发送快照。
- 为了执行 RemoveNode，你应该显式调用 destroyPeer()来停止 Raft

  模块。销毁逻辑已经为你提供了。
- 不要忘记在 GlobalContext 的 storeMeta 中更新

- 测试代码会多次调度一个 conf change 的命令，直到这个 conf change 被应用，所以你需要考虑如何忽略相同 conf change 的重复命令。

在 raftstore 中实现 region 拆分



为了支持 multi-raft，系统执行数据分片，并使每个 Raft 组只存储一部分数据。Hash 和 Range 是常用的数据分片方式。TinyKV 使用 Range，主要原因是 Range 可以更好地聚合相同前缀的键，方便 scan 等操作。此外，Range 在 split 上的表现优于 Hash。通常，它只涉及元数据修改，不需要移动数据。

让我们重新看一下 Region 的定义，它包含 start_key 和 end_key 两个字段，用来表示 Region 负责的数据范围。所以 split 是支持 multi-raft 的关键步骤。一开始只有一个区域有射程[“”，“”)。你可以把键空间看作一个循环，所以 [" "，" ") 代表

为了确保新创建的 Region 和 peer 的 id 是唯一的，这些 id 是由调度器分配的。它也提供了，所以你不需要实现它。

onPrepareSplitRegion()实际上调度了一个任务给 pd worker 去问调度器要 ids。并在收到调度器的响应后进行拆分管理命令，参见 onAskSplit()

kv / raftstore /跑步/ scheduler_task。
走了。

所以你的任务是实现处理 split admin 命令的过程，就像 conf change 一样。的
提供的框架支持多个 raft，参见 kv/raftstore/rou 分为二

区域，其中一个区域将在分裂前继承元数据，只修改其 Range 和 RegionEpoch，而另一个将创建相关的元信息。

提示:

- 新创建的 Region 对应的节点应该通过 createPeer()创建并注册到 router.regions。并且 region 的信息应该插入到 ctx.StoreMeta 的 regionRanges 中。
- 对于使用网络隔离拆分区域的情况，应用的快照可能有重叠与现有区域的范围。检查逻辑在 checkSnapshot()中 执行时请牢记，保重关于这件事。
- 使用 engine_util. exceedkey()与 region 的结束键进行比较。因为当结束时 Key 等于" "，任何键都会等于或大于" "
- 还有更多的错误需要考虑:ErrRegionNotFound, ErrKeyNotInRegion， ErrEpochNotMatch 。

C 部分

如上所述，kv 存储中的所有数据都被划分为多个 region，每个 region 包含多个副本。一个问题出现了:我们应该把每个副本放在哪里?我们如何找到放置复制品的最佳位置?谁会发送以前的 AddPeer 和 RemovePeer 命令?调度器承担这个责任。

为了做出明智的决策，调度器应该有关于整个集群的一些信息。它应该知道每个区域的位置。它应该知道它们有多少个键。为了获得相关信息，调度器要求每个区域都应该发送一个心跳请求
定期发送到调度器。你可以在里面找到心跳请求结构 RegionHeartbeatRe
到心跳后，调度器会更新本地区域信息。

同时，调度器定期检查区域信息，以发现我们的 TinyKV 集群中是否存在不平衡。例如，如果任何一个存储包含了太多的 region，那么 region 就应该从这个存储中移动到其他的存储中。这些命令将被拾取作为对应区域心跳请求的响应。

在本部分中，您将需要为调度器实现上述两个函数。遵循我们的指导和框架，不会太难。

的代码

接

Region 心跳，它会先更新自己的本地 Region 信息。然后它会检查这个区域是否有待处理的命令。
如果有，它将作为响应发送回来。

您只需要实现 processRegionH 更新本地信息;和用于平衡区域调度器的
Schedule 函数，调 中扫描存储

并确定是否存在不平衡，以及它应该移动哪个区域。

收集区域心跳

如您所见，processRegionHeartbeat 函数 Info。它包含

关于此心跳发送区域的信息。调度器需要做的只是更新本地区域记录。但它应该每隔一次心跳就更新
这些记录吗?

当然不行!原因有二。一是当该区域没有任何变化时，可以跳过更新。更重要的一点是，调度器不能信
任每个心跳。特别是，如果集群在某一段有分区，某些节点的信息可能是错误的。

例如，一些区域在分裂后重新启动选举和分裂，但是另一批孤立的节点仍然通过心跳将过时的信息发送
给调度器。因此，对于一个区域，两个节点中的任何一个都可能说它是 leader，这意味着调度器不能同时
信任它们。

哪一个更可信?调度器应该使用 conf_ver 和 version 来确定它，即

RegionEpoch。调度器应该首先比较两个节点的区域版本的值。如果
值相同时，调度器将比较配置更改版本的值。配置变更版本较大的节点必须有更新的信息。

简单来说，你可以按照下面的方式来组织检查例程:

1. 查看本地存储中是否存在相同 Id 的 region。如果存在且至少有一个
   心跳的 conf_ ion 小 这个心跳区域过时了
2. 如果没有，扫描所有与之重叠的区域。心跳的 conf_ver 和 version 应 等于 ，否则该
   区域是陈旧的。

那么调度器如何确定是否可以跳过此更新?我们可以列出一些简单的条件:

- 如果新心跳的版 _ver 跳，则不能跳过
- 如果 leader 发生了变化，则不能跳过
- 如果新的或原来的有 pending peer，则不能跳过如果近似大小发生了
- 变化，则不能跳过
- ...

别担心。你不需要找到严格的充要条件。冗余更新不会影响正确性。

如果调度器决定根据此心跳更新本地存储，则需要做两件事

应该更新的:region on 树，并使用 RaftCluster.core.U

Leader count, region count, pending peer count…)

实现 region 平衡调度

调度器中可以运行许多不同类型的调度器，例如 balance-region 调度器和 balance-leader 调度器。本学习材料将重点介绍 balance-region 调度器。

每个调度器都应该实现调度器接口，您可以在

Interval 的返回值作为默认间隔，用 GetNextInterval 来增加间隔。通返回一个操作符，调度器也会返

调度这些操作符作为相关区域下一个心跳的响应。

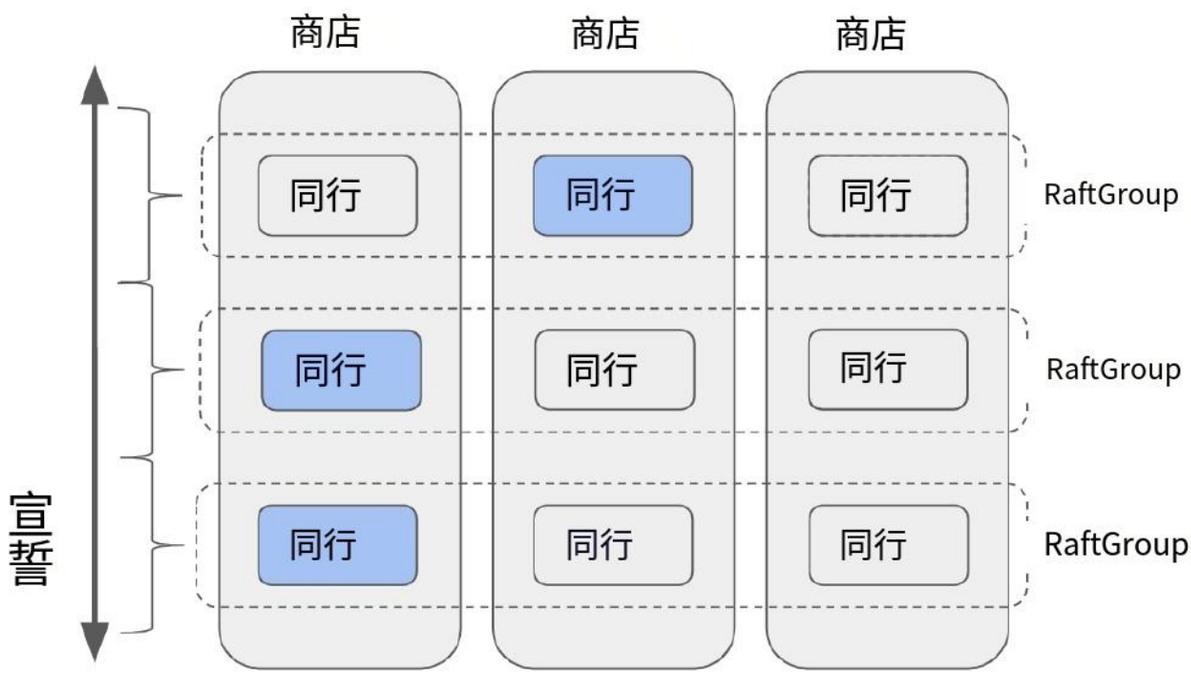
调度器接口的核心部分是 Schedule 方法。这个方法的返回值是 Operator，

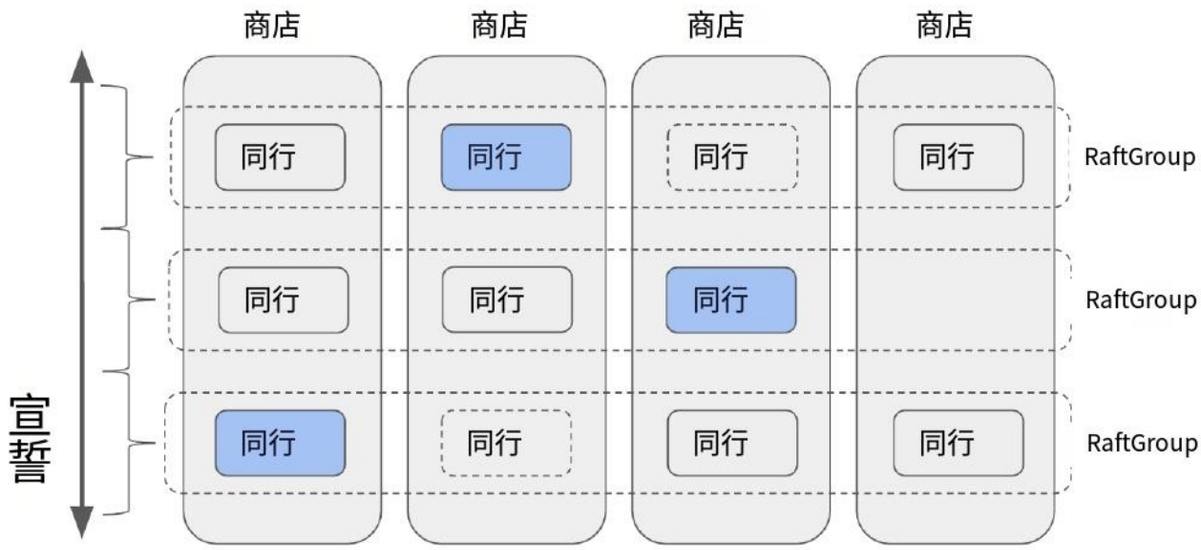
它包含多个步骤，如 AddPeer 和 RemovePeer。例如，MovePeer 可能包含 AddP Leader 和 RemovePeer，你已经在前一部分实现。以第一部分为例

以下图中的 RaftGroup 为例。调度器试图将同伴从第三个存储移动到

第四。首先，它应该为第四个商店 AddPeer。然后检查第三个是否是 leader，发现不是，所以不需要 transferLeader。然后它将第三个商店中的对等节点移除。

您可以使用 scheduler/server/schedule/operator 包中的 CreateMovePeerOperator 函数创建 MovePeer 操作符。





在这一部分中，你唯一需要实现的函数就是里面的 Schedule 方法

调度器/服务器/调度器/ balance_region。走了。这个调度器避免在一个调度器中有太多的 region

商店。首先，调度器将选择所有合适的存储。然后根据它们的 region 大小进行排序。然后，调度器尝试从具有最大区域大小的存储中找到要移动的区域。

调度器会尝试在商店中找到最适合移动的区域。首先，它会尝试选择一个挂起区域，因为挂起可能意味着磁盘超载。如果没有挂起区域，它会尝试寻找一个 follower 区域。如果它仍然不能挑选出一个区域，它将尝试挑选领导者区域。最后，它将选择要移动的区域，或者调度器将尝试具有较小区域大小的下一个存储，直到尝试完所有存储。

在选择要移动的一个区域之后，调度器将选择一个存储作为目标。实际上，调度器将选择具有最小区域大小的存储。然后，调度器将通过检查原始存储和目标存储的区域大小之间的差异来判断这种移动是否有价值。如果差异足够大，调度器应该在目标存储上分配一个新的对等点，并创建一个移动对等点操作符。

你可能已经注意到了，上面的例程只是一个粗略的过程。还留下了很多问题:

- 哪些店适合搬家?

简而言之，一个合适的门店应该是上线的，下线时间不能超过集群的 MaxStore 过 cluster. getmaxstoredowntime()获取。

- 如何选择区域?

调度器框架提供了三种获取区域的方法。GetPendingRegionsWith

etlead 可以从中获得相关的区域。然后你可以随机选择一个区域。

- 如何判断这个操作是否有价值?

如果原始存储和目标存储的区域大小之间的差异太小，那么在将区域从原始存储移动到目标存储之后，调度器下次可能需要再次移动。所以我们要确保这个差值必须大于该区域近似大小的两倍，这就保证了在移动之后，目标商店的区域大小仍然小于原来的商店。