Lab4 Sharded Key Value Service

Part A ShardMaster

在实现时，基本沿用 lab3 kvpaxos 的架构。可以将 join, leave, move 理解为 Put 操作，将 query 理解为 Get 操作。主要的工作量在于 join, leave 和 move 的 apply 实现，以及 shard rebalancing。

下文除了描述这两点之外，还讨论了一下为什么需要在 client 端添加 concurrency control。

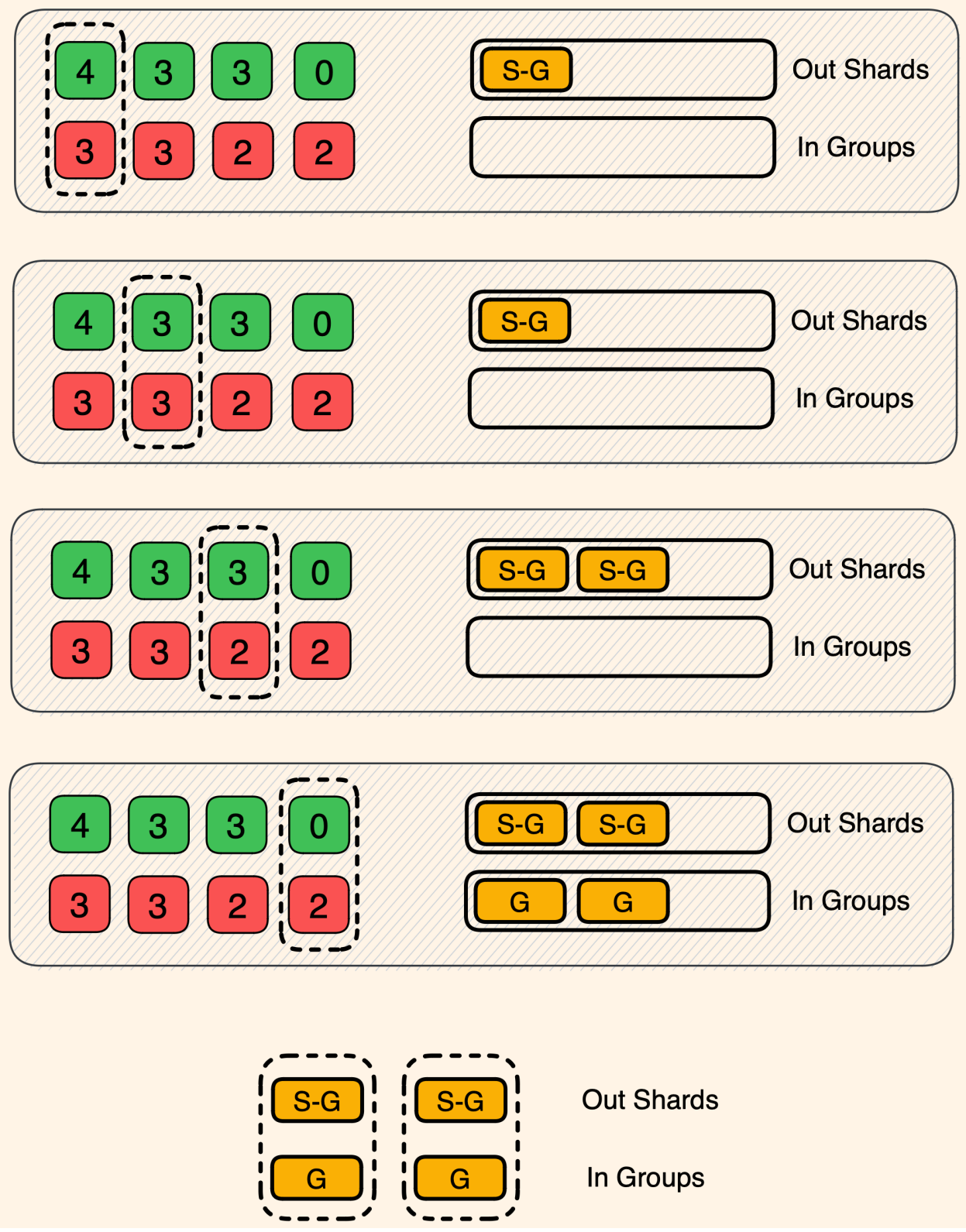
100 次测试结果：

Test	Failed	Total	Time
TestBasic	0	100	8.81 ± 0.25
TestUnreliable	0	100	6.92 ± 0.86
TestFreshQuery	0	100	0.68 ± 0.22

join, leave, move 实现时遇到的一些问题

- 最初始的那次 join，需要将所有 shards assign 给这个唯一的新加入的 group。因此可以做一个特判。
- 有可能重复 join 一个 group，此时应该直接跳过。重复的 leave, move 也应该跳过。
- 不会存在先 join 一些 groups ，然后一直 leave 导致没有集群中不存在任何 groups ，因此不需要考虑将 shards 重新 assign 到 invalid gid 0。

shard rebalancing 是如何做的？


4
3
3
0
3
3
2
2
S-G
Out Shards
In Groups
4
3
3
0
3
3
2
2
S-G
Out Shards
In Groups
4
3
3
0
3
3
2
2
S-G
S-G
Out Shards
In Groups
4
3
3
0
3
3
2
2
S-G
S-G
Out Shards
G
G
In Groups
S-G
S-G
Out Shards
G
G
In Groups

config change 涉及到 curr config 和 new config 两个 configs。对于每个 config，根据每个 group 所 serve 的 shards 的数量，对 groups 进行排序，shards 数量多的排在前面。如此操作后，我们得到了两个有序序列。对于 join 操作后的 rebalance，new config 的序列长一个单位。对于 leave 操作后的 rebalance，curr config 的序列长一个单位。不管是哪种情况，我们都做一个 0-padding，即将 0 补全到较短这个序列的末尾。至此，我们得到了两个等长的有序序列。

对于这两个有序序列，对它们进行类似函数式编程中的 zip 操作，即 parallel iteration。具体而言，每次取两个序列对应索引处的两个元素进行比较。设属于 curr config 序列的那个元素为 X，属于 new config 序列的那个元素为 Y。则 X 和 Y 的大小关系有且仅有如下三种：

- X > Y：则我们需要从 X 所对应的 group 中取出一个 shard。这样的 shard-group 组合，被 append 到 out shards 数组。
- X = Y：不需要操作。
- X < Y：则我们需要将某个 shard 放入 Y 所对应的那个 group。这样的 group，被 append 到 in groups 数组。注意，如果需要给某个 group 放入 k 个 shards，则 in groups 中有 k 个该 group。

在遍历结束后，out shards 和 in groups 数组的长度必定相等。此时，我们同样以 parallel iteration 的方式去遍历这两个数组。每次遍历到的 `Sx-Ga, Gb` pair 即表示我们需要将 group A 中的 shard X 移动到 group B。

这样的 shard rebalancing 策略，可以保证所移动的 shards 的次数最小。

为什么需要在 client 端添加 concurrency control？

测试中有这样一个场景：多个 threads 调用同一个 clerk 的同一个接口，例如 Join。调用时，它们的参数不一样，例如一个 join G1，另一个 join G2。在我 kv-paxos 的实现中，server 这一端实现了 dup checking 和 dup filtering logic，这就需要 clerk 这一端在发送请求时，在请求中附带一个分配的 unique op id。假设 join G1 的 op id 是 x，join G2 的 op id 是 y，且 x < y。那么可能出现这样一种情况，join G2 被执行了，那么 server 这一端的 max exec op id of clerk 便会更新到 y。则即使 server 这一端 replicate 了 join G1 这个请求，由于它的 op id = x < y，则不会被 server 所 execute。这就导致了 lost write。为了解决这个问题，需要在 clerk 这一端添加 concurrency control，即不允许同一个 clerk 同时发送多个请求。

当然也可以不在 clerk 这一端添加 concurrency control，那样的话 server 这一端也不能实现 dup checking 和 dup filtering，或者可以换一种 dup checking and filtering 的实现逻辑。这样的话，所有的 write 请求都会被 execute，则不会有 lost write。2015 版的 spec 也说了 shardmaster 不需要实现 dup checking and filtering，所以 starter code 有 comments 说不需要修改 `client.go` ，即 clerk 这一端的原始代码。

由于 part B shard-kv 需要在 server 端实现 dup checking and filtering logic，并且 2023 版的 spec 说了如果 shardmaster 不实现 dup checking and filtering，那么 shard-kv 这一部分的有些测试会很难通过。考虑到这些，我选择在 shardmaster 的 server 端实现 dup checking and filtering，同时为 shardmaster 的 clerk 端添加 concurrency control。

如果你实现了 dup checking and filtering，为了测试程序的正确性，那么需要在 `TestUnreliable` 测试中 uncomment `sma[i].setunreliable(true)` 这一行，如此才会开启 network 的 unreliable 特性，才能测试 dup checking and filtering 功能的正确性。

Part B Shard-Kv

本部分要求实现一个支持 config change 功能的 sharded key value store。

特别要指出的是，完整的 config change 包含 key to shard, server to group, shard to group 的变化，但是本 lab 只考虑 shard to group 变化的 config change。

在实现时，基本沿用 lab3 kvpaxos 的架构。在此基础上，添加了与 config change 和 shard migration 相关的组件和代码。

下文首先对系统架构进行描述，并据此说明系统是如何 handle client op 和执行 config change 的。之后，以讨论的形式，对我在设计和实现的过程中，所遇到和思考过的问题，进行描述和总结。最后，描述一些我在测试时发现的一些 bugs 以及我在做完这个 lab 后所获得的一些分布式编程经验。

执行单线程测试 100次 的结果：

Test	Failed	Total	Time
TestBasic	0	100	15.52 ± 0.29
TestMove	0	100	20.18 ± 0.24
TestLimp	0	100	17.40 ± 0.24

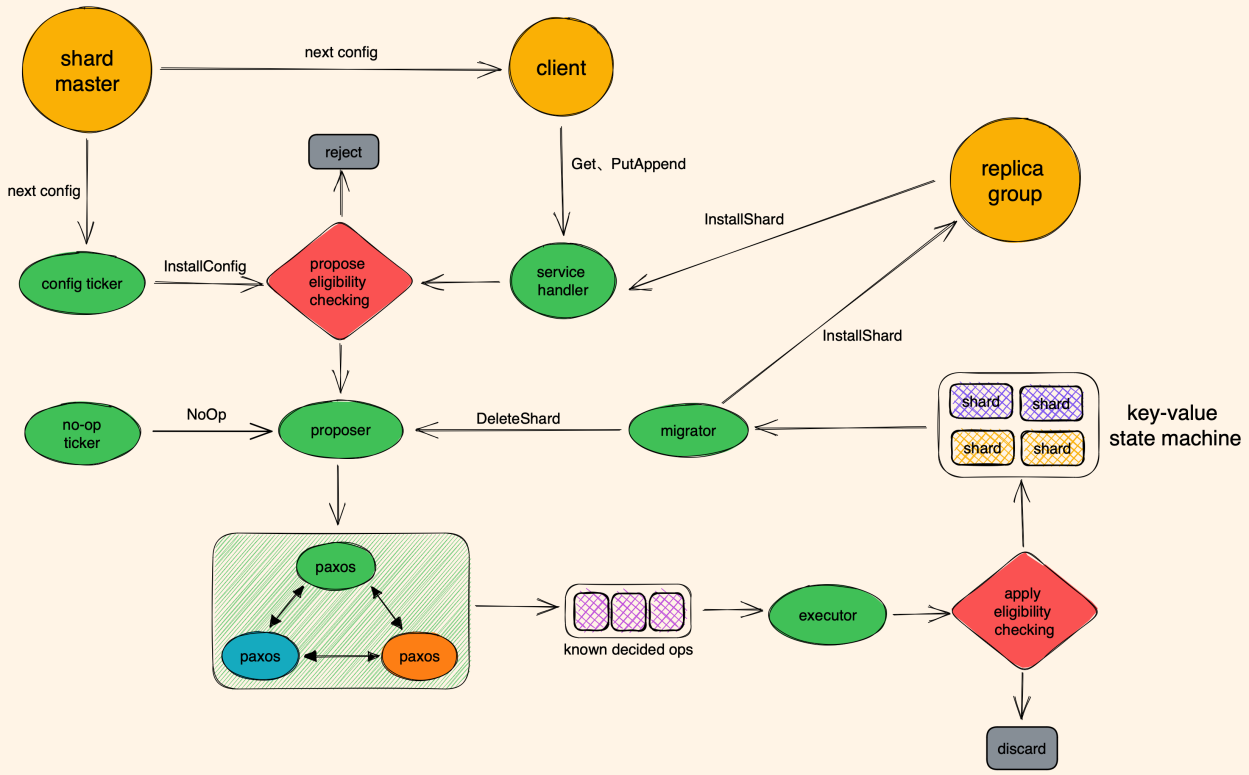
执行并发测试 500 次的结果：（原本是 `TestConcurrent` ，但是我用的测试脚本会将每个输入的测试名字作为 prefix，去匹配所有包含这个 prefix 的测试。所以将其改为 `TestConcurrent1`，这样输入 `TestConcurrent` 时就不会匹配到 `TestConcurrentUnreliable`）

Test	Failed	Total	Time
TestConcurrent1	0	500	3.80 ± 0.19
TestConcurrentUnreliable	0	500	4.22 ± 0.81

执行最后一个测试 10000 次的结果：

Test	Failed	Total	Time
TestConcurrentUnreliable	0	10000	4.28 ± 1.79

系统架构



图中，绿色的为当前 server 的组件。各组件的功能为：

- config ticker：周期性地与 shardmaster 服务通信，尝试拉取 server 当前 config 的下一个 config，即 next config。
- no-op ticker：周期性地创建一个 no op，推入 proposer 线程。
- service handler：负责 handle 从 client 和其他 server 发送过来的 request。将 request wrap 成 op，然后推入 proposer 线程。
- migrator：周期性地检查 shards 的状态。尝试将 moving out 的 shards 迁移到其他 group。如果当前 server 正在 reconfiguring，且检测到所有 shards 的 state 不为 moving in 或 moving out，那么将 reconfiguring 设置为 done。
- proposer：当且仅当 eligible to propose 一个 op 时，与 paxos 通信，尝试在 server 已知的、最新的 seq num 处 propose 一个 op。并将已知的 decided ops 推入 known decided ops queue。
- executor：负责 consume known decided ops，顺序地执行 decided ops。如果 eligible to apply 一个 op，那么 apply 它。

- paxos：与 group 内的其它 server 的 paxos 通信，尝试 decide 一个 op。

client op 是如何被 handle 的？

client 根据 `key2shard` 函数和 config 中的 shard to group 以及 group to servers mappings，获知可能 handle 当前 request 的 servers。client 将 request 发送给 servers 中的某个 server。

server 收到 client request 后，首先根据 shard state 判断该 server 当前是否 serving 这个 request。如果不，直接返回 ErrWrongGroup。

之后，server 根据 max applied op id 判断 request 是否已经被 applied。如果已经 apply 了，对于 Put, Append，返回 OK；对于 Get，根据 request key 从 shard data 中拿到对应的 value，返回这个 value。如果尚未 apply，则将 request wrap 到一个 op 中，并启动一个 proposer 线程，不断地尝试 propose 这个 op 到 paxos 层。service handler 会在一个 for loop 中 polling max applied op id，周期性地检查这个 op 是否已经被 apply 了。

待这个 op 被 decide 后，executor 会 execute 它。如果此时，这个 op 还是尚未 apply，那么 apply 这个 op，并且更新 max applied op id。否则，discard 这个 op。

service handler 通过 max applied op id 检测到该 op 被 apply 后，进行相应的回复。如果在等待 apply 期间，service handler timeout 了，那么就回复 ErrNotApplied。

client 如果拿到 OK 的 reply，则会终止发送 request。如果拿到 ErrNotApplied 的 reply，则会尝试将 request 重发给 group 内的另一个 server。如果拿到 ErrWrongGroup 的 reply，则会向 shardmaster 请求一个新的 config，再尝试重发。

reconfigure 是如何做的？

config ticker 周期性地拉取新的 config。如果拉取到一个新的 config，进行 install config op 的 propose eligibility checking。如果允许 propose 这个 config，那么 propose 它，并且将 reconfigureToConfigNum 设为新的 config 的 config num。

当 executor execute 这个 install config 时，进行 apply eligibility checking。如果允许 apply，那么 install 这个新的 config。如果本次 config change 需要进行 shard migration，那么开始进行 hand off shards 和 take over shards，将这些 shards 的 state 分别设为 moving out 和 moving in。

shard sender 会通过一个 install shard request 将 shard 发送给 shard receiver。如果 shard receiver 已经 propose 了这个 install shard op，那么立即回复 OK。shard sender 收到这个 OK 后，尝试 propose 一个 delete shard op。

待 install shard op 被 apply 后，更新 shard data 和 max applied op id，同时将 shard state 从 moving in 设为 serving。待 delete shard op 被 apply 后，将 shard data 删除，同时将 shard state 从 moving out 设为 not serving。

一些问题的讨论

为什么 shardmaster 做 shard rebalancing 时需要保证 deterministic？

对于 replicated state machine，不仅要保证 apply ops 的顺序是一致的，还需要保证所有 servers apply 同一个 op 的效果是一致的。

对于本 lab 所涉及的 shard rebalancing，我们关注的是每个 group 所 serve 的 shards 的数量，而并不关注哪个 group serve 哪个 shard。但是，如果 shardmaster servers 对于 shard rebalancing 的结果产生分歧，那么在回复 shardmaster client 时，不同的 servers 可能会给不同的 clients 回复不同的 config，即虽然 config num 一致，但是 shard to group 映射关系却是不一致的。

在我的 shard rebalancing 的实现中，涉及到了对 map 的 iteration 和对 array 的 sorting。前者在 Go 中是不确定的，即对于一个相同的 map，多次 iterations 所产生的 kv pairs 的序列是不同的。对于 sorting，分为 stable sorting 和 unstable sorting 两类排序算法。

为了保证 deterministic，需要将 map 中的 kv pairs collect 到一个 array 中，然后再使用 stable sorting 进行排序。如此才能保证不同 shardmaster servers 的 shard rebalancing 的结果是一致的。

是否需要在 client 端添加 concurrency control？

加上肯定是没错的。至于有没有必要，需要看 test codes 是否会在多个线程中调用同一个 client 的接口。

at most once 和 exactly once 语义如何实现？

对于 at most once 语义，需要让 server 拒绝 apply 已经 applied ops，那么就需要唯一地标识一个 op。通常的做法是让 client 给 op 打上 unique tag。

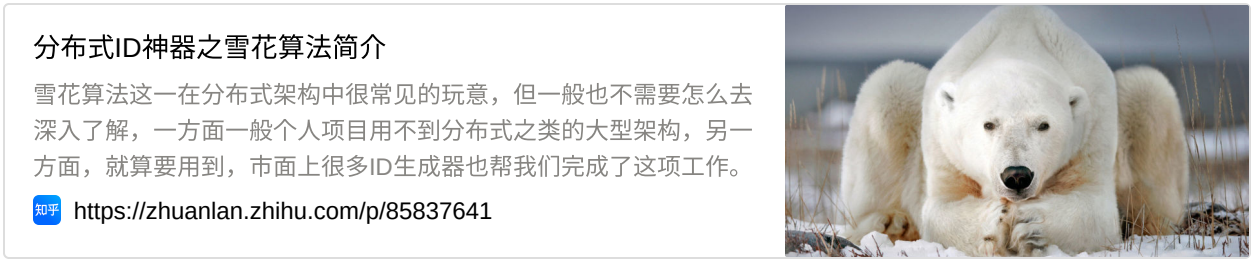
如何生成 unique tag？通常有两种方法：

- 让 client 自行分配一个：使用一个随机生成器为每个 request 生成一个 unique tag。
- 让 client 向一个分布式 id 生成服务请求一个 unique tag。

我在实现中使用的是第一个方法。具体而言，我使用了 starter code 提供的 `nrand` 函数，利用它生成一个随机的 clerk id。然后再维护一个单调严格递增的 seq num，作为 op id。将 clerk id 和 op id 组合起来作为 unique tag。

虽然 lab spec 要求我们实现 at most once 语义，但是为了通过测试，实际上我们需要实现 exactly once 语义。这就需要 client 端的参与。具体而言，client 端必须在收到上一个 request 的 reply 之后，才开始发送下一个 request。也就是说，如果没有收到 reply 或者 reply 不为 OK，那么 client 会一直重试发送 request，直到收到 OK 的 reply。

关于分布式 id 生成算法，参考雪花算法（snowflake）。



linearizability 语义如何实现？

关于线性一致性，需要从 client 和 server 两个角度去理解。

- 对于 client：当前 op 必须发生在上一个 op 之后。注意，这里的“之后”并不代表“紧邻”。
- 对于 server：在 apply op 时保证原子性。

在实现时，常用的方法为：

- 对于 client：为 request 的 unique tag 定义一个偏序关系。例如对于 clerk id 和 op id 构成的 request tag，clerk id 可以是随机生成的，op id 则需要是严格单调分配的（可以是严格单调递增，也可以是严格单调递减）。
- 对于 server：
  - 在 apply op 时进行 concurrency control，例如加锁。
  - 为每个已知的 client 维护 max applied op id，拒绝 apply 小于或等于 max applied op id 的 ops。这也是为什么要求 op id 是严格单调分配的。

为什么不需要 cache reply？

考虑什么场景下需要 cache reply：当一个 op 被 apply 后，server 给 client 回复一个 reply。由于网络原因或者 server crash，client 并没有及时收到这个 reply，导致 timeout。此时，client 会重发同一个 request，这就是一个 dup request。很多人的做法是，在 apply 时 cache 这个 reply，然后在检测到这个 dup request 已经被 apply 时，直接回复这个 cached reply。

对于 Put, Append，如果检测到一个 dup request 已经被 apply 时，只需要回复 OK，因此没有什么需要 cache 的。

对于 Get，当我们检测到一个 dup request 已经被 apply 时，client 有且仅有如下两种状态：

- client 没有收到过这个 reply
  - 将线性一致性在 client 端的定义应用在此处，有：当前的 Get 需要看到所有之前由这个 client 所发出的、并已经 applied 的 Put, Append。不管我们是在初次 apply 这个 Get 时获取 value，还是在收到 dup Get 时获取 value，虽然两次 Get 获取的 value 可能不一致，但是它们都满足线性一致性。具体而言，虽然在两次回复之间，server 可能 apply 一些由其他 clients 发出的 Put, Append，导致 value 变化，但是对于这个 client 而言，两次回复在线性一致性的意义上都是正确的。实际上，对于分布式场景，对于不同 clients 之间的 interleave 根本不能有效的约束。
- client 已经收到过这个 reply。这可能是由于网络将这个 request dup 了，或者 reorder 了。
  - 此时，回复什么 value 无所谓，因为 client 已经收到过 reply 了。

总结，我们不需要在初次 apply op 时 cache reply。在检测到一个 dup request 已经被 apply 时，对于 Put, Append，直接回复 OK。对于 Get，根据 request key 获取 value，回复这个 value 即可。

为什么不需要返回 ErrNoKey？

在 Get 获取不存在的 key 时，starter code 提示我们返回一个 ErrNoKey。在这个 lab 中，是没有必要区分 reply OK 和 reply ErrNoKey 的，因为 client 端在检测到 reply OK 和 reply ErrNoKey 时，都会立即终止 sending request，直接返回 reply value。因此，我们只需要让 server 在检测到 key 不存在时，将 reply value 设置为空字符串，然后回复 OK 即可。

service handler 等待 timeout 时需要区分 ErrNotApplied 和 ErrWrongGroup 吗？

不需要。在我的设计中，只会在 receive client requests 时，判断当前 server 是否 serve request key。如果不 serve，则不管其 apply 与否，直接回复 ErrWrongGroup。

当 service handler 等待 timeout 时，server 会回复 ErrNotApplied，不会检查当前 server 是否 serve request key。因为 client 没有收到 ErrWrongGroup，那么它可能重发这个 request 给 group 内其它 server。假设此时 server 不再 serve 这个 request key，那么 server 会回复 ErrWrongGroup。之后，client 会拉取新的 config，把 request 发送给正确的 group。

为什么需要 No Op？

一个常用的优化是 no-op。为什么需要这个呢？

假设一个 client 总是把 requests 发送给 group 中的某一个 server。为了回复这个 request，这个 server 的 paxos 会一直与 group 内其他 server 的 paxos 保持通信。为了 decide 这个 op，至少需要 majority paxos 都参与。也就是说，任何一个 server propose 任意一个 op，都会驱动 paxos 层的同步。

paxos 暴露出了 `Start` 和 `Status` 两个接口，前者用于向 paxos propose 一个 op，后者用于获知对应给定的 seq num 的 paxos instance 的 status，即是否已经 decided，以及 decided value 是什么。对于这两个接口，我是这样用的，server 中的 proposer 拿到一

个 op 后，调用 `Start` 接口尝试将其 proposer 为 server 已知的、最新的 seq num 处的 value，然后调用 `Status` 接口等待这个 seq num 处的 value 被 decide。如果 proposer 发现 decided value 就是 proposer 想要 propose 的这个 op，那么结束 proposing，否则继续调用 `Start`，尝试将其 propose 为下一个 seq num 处的 value。当且仅当 proposer 发现它想要 propose 的这个 op 被 decide。

经以上分析知，server 获知新的 decided ops 的唯一途径就是通过 propose 新的 ops。由于 executor 只会尝试 execute decided ops，因此如果一个 server 一直不 propose op，即使它的 paxos 已经和其他 server 的 paxos 同步，server 的 state 也不会同步。

显然，不仅让 paxos 层，也让 server 层更快地同步，是有利的。因此，在 server 层设置一个 no-op ticker，其周期性地创建一个 no op，将其推入 proposer，则 server 层可以更快地拿到由其他 server propose、已经被 decided ops，则可以更快地 catch up。

至于为什么叫 no op，是因为它并不会对 state machine 产生任何效果，因为我们根本不会 apply 它。

paxos 层是否也需要一个 no op ticker？

可以，但没必要。

如果仅仅 paxos 有 no op ticker，那么 server 层还是不能很快地同步。所以通常会在 server 层设置一个 no op ticker。而由于 server 层已经有一个 no op ticker，那么 paxos 层就没必要再设置一个 no-op ticker。

proposer 和 executor 是如何通信的？

在架构图中，proposer 为系统中一个单独的组件。然而在实现中，proposer 并不是一个独立的线程。每次 server 要 propose 一个 op 时，都会创建一个独立的 proposer 线程。这个线程会不断地尝试在已知的、最新的 seq num 处 propose 这个 op，直到这个 op 最终被 decide。

我之前也考虑过，将 proposer 设计为一个独立的线程，为它维护一个 op buffer queue。server produce ops，proposer 则 consume ops。Go 中并没有 queue 或者 list 这样的内置数据结构，而只有 buffered channel，但是 buffered channel 的 capacity 如何设置呢？所以我最终还是选择让 server 要 propose 一个 op 时，都创建一个独立的 proposer 线程。

那么 proposer 与 executor 是如何通信的呢？即 decided ops 如何从 proposer 传输给 executor 呢？使用 channel 是可行，但是使用 unbuffered channel 肯定会影响

throughput，使用 buffered channel 又如何设置 capacity 呢？所以我最终采取的是 communication by shared memory。具体而言，我设计一个 decided ops map，其 key 为 seq num，value 为 decided op。proposer 不断地往其中 insert decided ops，不管顺序。executor 则通过维护的 next exec seq num 顺序地执行下一个 decided op。

service handler 如何等待 applied？

两种比较常见的方式：channel 和 polling。

对于 channel，我们可以让 service handler 在 propose 一个 op 之前，注册一个 channel。当这个 op 被 execute 后（注意不是 apply），executor 拿到这个注册的 channel，通知 service handler。这个方法需要解决两个问题：

- 如何拿到注册的 channel：可以将所有注册的 channels 存储在一个 map 中，其 key 为给每个注册的 channel 打上的一个与 op 有关的 unique tag，例如 clerk id 和 op id 的构成的 tag。其 value 为注册的 channel。
- 如何 close 和销毁 channel 以回收资源：为了 close channel 的安全性，必须保证两点：（1）只在 sender 端做一次 close；（2）只 close 同一个 channel 一次。channel 的 sender 是 executor，receiver 是 service handler。

对于 polling，可以让 service handler 在 propose 一个 op 后，等待一段时间。在这段时间内，service handler 周期性地检测 max applied op id。如果发现这个 op 被 apply 了，那么就采取对应的行动。如果 timeout 了，那么就 reply ErrNotApplied。

在我的实现中，用的是 polling 的方法。因为 polling 的方法更容易实现，不需要和易出错的 channel 打交道。并且 channel 常用来传递数据，然而这里只需要一种通知机制。

为什么 server 需要拉取 next config 而不是 latest config？

假设 servers 总是尝试拉取最新的 config。考虑这样一个场景：有三个 groups A, B, C，三个连续的 configs X, Y, Z，以及一个 shard S。且有：

- 在 config X，A serves S。
- 在 config Y，B serves S。
- 在 config Z，C serves S。

假设 group A 正在 reconfigure to config X，group B 正在 reconfigure to config Y，group C 正在 reconfiguring to config Z。再假设 A 在 install config X 后，拉取的是最新的 config Z。此时，group B 正在等待 group A 将 shard S 传输给它，group C 正在等待 group B 将 shard S 传输给它。由于 group B 一直在等待 shard S，因此它无法 install config Y，更无法开始 reconfigure to config Z。因此，group C 也一直在等待 shard S。

如果总是尝试拉取 next config，那么 group A 在 install config X 后，会拉取到 config Y。因此 group A 会先将 shard S 发送给 group B。待 group B install config Y 后，group B 会拉取到 config Z，将 shard S 发送给 group C，因此 group C 也能成功完成 install config Z。

为什么 client 需要拉取 next config 而不是 latest config？

client 切换新的 config 是很快的，而 server 切换新的 config 需要做同步，因此较慢。假设 shardmaster 在较短的时间内做了多次 config change，而 client 总是尝试拉取 latest config。那么 client 的 config 可能比 server 的 config 超前很多。

可能存在这样一种情况，在 client config 中，由 group A serve shard S。在 server config 中，group B serve shard S。那么 client 发给 group A 的 requests 将很长时间得不到 serve。如果给测试设置的 timeout 不够大，那么可能导致测试超时。

为什么 client 和 server 不需要同步 config？

有一种做法是，service handler 会判断 client config 和 server config 是否相等，如果相等才会 serve 这个 request。这样的做法是不必要的。

client 只需要根据 client config，将 request 发送给某个 group。如果 group config 与 client config 相等，那么自然会 serve 这个 request。即使 group config 与 client config 不等，只要在 group config 中，还是由这个 group serve 这个 request，那么这个 group 也可以 serve 这个 request。

另一方面，client 切换 config 很快，而 server 切换 config 较慢，因此强制要求 client 和 server 同步 config，会降低 throughput。

为什么使用 reconfigureToConfigNum 而不是 reconfiguring？

在我的设计中，从 server 通过 config ticker 拉取到一个新的 config 并将其 propose 开始，对新的 config 的拉取会被暂停。我们当然可以继续 propose 更新的 config，但是由于 install config 通常涉及 shard migration，因此在 execute 第二个 install config op 时，前一个 config 的 shard migration 可能还未完成，此时第二个 install config op 只能被 discard。因此，为了减少被 discard 的 install config ops，我选择从 propose install config 开始，就暂停拉取新的 config。

根据以上设计，reconfiguring 包括两个阶段：（1）propose install config 至 apply install config；（2）shard migration。在阶段（1），reconfigureToConfigNum = server config + 1。在阶段（2），reconfigureToConfigNum = server config。使用 reconfigureToConfigNum 可以区分这两个阶段，使用 reconfiguring 则不能。只有在阶段（2），我们才能 install shard。因此这可以作为判断是否可以 accept install shard request 以及是否可以 apply install shard op 的依据。

另一方面，根据分布式编程经验，使用 reconfiguring 这种 bool 变量，实际上是很有歧义的，因为不知道是 reconfigure to which config。所以更 canonical 的做法应该是用 reconfigureToConfigNum 这样的变量来明确指出当前正在 reconfiguring to which config。如果不在 reconfiguring，那么设置 reconfigureToConfigNum 为 -1。

为什么允许 groups 使用不同的 configs ？这样不会导致多个 groups 同时 serve 一个 shard 吗？

首先要指出的是，group 内使用 paxos 进行同步，但是 group 之间并没有进行同步。因此不同的 groups 使用不同的 configs 是不可避免的。

对于一个 shard 的 migration 而言，有 from group 和 to group。假设 from group 先 install 这个 config，那么它会开始 handoff shard。这个 handoff shard 不能完成，除非 to group 也 install 了这个 config，并且已经完成了 take over shard。所以对于任意一个 shard 的 from group 和 to group 而言，它们 install config 可能不是同步的，但是 shard migration 的完成一定是同步的。

如果一个 to group 想要 serve 一个 shard，它必须等待 from group 也 install 同一个 config，并且它们二者均完成 shard migration。对于较先 install config 的那个 group 而言，不管它是 handoff shard 还是 take over shard，这个 shard 的状态都不是 serving。从此刻开始，直到两个 groups 完成 shard migration 的那一刻为止，在这个期间一定不会存在同时有两个 groups 都在 serve 这个 shard。

reconfiguring 的什么阶段需要 reject client requests？

首先，判断 accept 还是 reject client requests，肯定是以单个 shard 为单位进行的。

在我的设计中，shard 存在四种状态：serving, not serving, moving in, moving out。对 shard state 的修改，只有在 shard migration 开始时和完成时才进行。当 apply install op 时，开始必要的 shard migration。当 apply install shard op 或 apply delete shard op 时，完成 shard migration。所以从 apply install config op 开始，到 shard migration 完成为止，apply client ops 都是禁止的。

也就是说，需要保证在 shard migration 阶段不会 apply 关于这个 shard 的 client ops。至于在这个阶段中，是否 reject client requests，是合理而非必要的。我们当然也可以将这个阶段拉长，比如从 propose install config op 开始，至完成 shard migration 为止，都 reject client requests。

在我的实现中，service handler 如果发现 shard state 不为 serving，就会 reject 这个 client request。

为什么需要同步 install config, install shard, delete shard？

对于一个由 replicated log 实现的 replicated state machine，要求所有的 state machine 按照一致的顺序执行所有的 ops，且对于单个 op 而言，每个 state machine 采取的执行操作都是一样的，即要么都 apply，要么都 discard。这个规则不仅对 client ops 适用，对 admin ops 也适用。

假设不同步 install config，那么对于同一个 client op，在有些 servers 上可能在 install config 之前被 execute，在另一些 servers 上可能在 install config 之后被 execute。由于 install config 涉及到 shard migration，因此对于同一个 op，有些 server 会因为还在 serving 这个 key 而 apply 它，另一些可能因为不再 serving 这个 key 而 discard 它。显然，这就破坏了 replicated state machine 的基本要求。

至于 install shard 和 delete shard，与 `为什么需要同步 install config` 的 rationale 一致，如果不同步，则会破坏 replicated state machine 的基本规则。

我一开始还思考了很久，要不要同步 delete shard。后来我领悟到了一个分布式编程经验：对于与 state machine 相关的修改，都需要经过 paxos 层进行同步。

什么时候 install config？

在我的设计中，当一个 install config op 被 apply 时，install config。之后才进行 shard migration。也就是说，install config 在 shard migration 开始时进行。

也有一些设计，它们把 install config delay 到 shard migration 完成时。它们先根据 pending config 进行 shard migration。这些 transferred shard data 与当前的 shard data 分开存储，并持久化。当 install config 所需的 shard data 都传输完毕后，propose 一个 install config op，里面包含了所需的 shard data。待这个 op 被 apply 时，同时 install config 和 shards。

如何同步 install shard？

为了同步 install config，我们把 config 放到了 install config op 中，让 paxos 层 replicate 它。同样地，为了同步 install shard，我们也需要把 shard data 放到 install shard op 中，让 paxos 层 replicate 它。

我一开始还考虑过不把 shard data 放到 install shard op 中。具体而言，在 server 层进行 RPC 交互来 replicate shard data，当所有 servers 收到必要的 shard data 并持久化以后，才 propose 一条不带 shard data 的 install shard op。待其 apply 时，才 install shard。

我一开始觉得以上的方法是不可行的，我当时是这样思考的：

即使 shard data 被 persist 到硬盘，但是 server crash 可能导致硬盘也丢失。此时，就需要 server 与 group 内的其它 servers 通信，去拉取完整的 replicated logs，或者拉取 snapshots 加上一部分 replicated logs suffix。不管是哪种，这个 server 可能都需要 execute install shard op。如果不把 shard data 作为 install shard op 的一部分，那么 shard data 又从哪来呢？从其它 servers 拉取过来的 shard data 是最新的 shard data，而不是 install shard op 时所需的 shard data。如果直接 install 最新的 shard data，然后又去 apply replicated logs suffix，那么就会违反 at most once 语义。

后来我发现，只要在拉取 snapshot 时，同时拉取 max applied op id，那么就可以跳过这些 applied replicated logs suffix，at most once 依然可以保证。

以上的做法，我没有验证过，但应该是可行的。但很显然，将 shard data 放到 install shard op 中，让 paxos 层去 replicate 它，就可以避免在 server 层重新实现一套 replication 协议以及 snapshot 相关逻辑。

什么时候可以 propose delete shard op？

当 shard sender 收到 install shard request 的 reply OK 时，shard sender 可以 propose 一个 delete shard op。

需要指出的是，由于 lab4 不涉及 server crash，因此，一旦 shard receiver 开始尝试 propose 这个 install shard op，即可向 shard sender 回复 reply OK。

lag-behind server 是如何 catch up 的？

在我思考解决这个 lab 的初期，这个问题困扰了我很久，我一直想不明白在存在 config change 和 shard migration 的条件下，如何进行 catch up。尤其是我一开始并没有想到把 shard data 存在 install shard op 中。

实际上，这并不是一个难点。一个 lag-behind 的 server 是这样 catch up 的：

首先，我们不需要管 config ticker，它拉取什么 config，对于 catch up 而言是没有影响的。即使 server 在 catch up 阶段 propose 甚至 decide 了一些 ops，它们肯定会在 catch up 完成之后才被顺序 execute。

一个 lag-behind 的 server 通常有两种情况，要么是 restart，需要 replay logs。要么是运行太慢，跟不上。不管是哪种，server 都需要顺序地 execute 已经 decided ops。

对于 client ops，只需要根据 server 当时的状态去 execute 就好了。replicated logs 保证了其它 servers 在当时的状态是一致的，因此它们对于 client ops 的操作也是一致的，要么都 apply，要么都 discard。

对于 install config ops，server 可能需要 handoff 或者 take over shards。

- 对于 handoff，它会把 shards 发送给其它 groups。其它 groups 检测到这些 shards 是 stale shards，因此会 reject to install shards，但是会 reply OK。这样 server 就知道这些 moving out shards 已经被 receiver take over 了，所以 handing off shards 就会终止。另一方面，shard migration 只会 discard client ops，对于 admin ops 的 execute 还是正常进行的。因此，server 一定会在之后 execute 一个对应这个 config 的 delete shard op。此时，server 也会终止 handing off shards。
- 对于 take over shards，因为这个 server lag behind，因此并没有 group 会把 shard 发送给它。但是，由于同一个 group 内的其它 servers 早就 apply 了一个对应这个 config 的 install shard op，且这个 op 中存储了所需的 shard data，则当这个 server apply 这个 op 后，taking over shards 也会终止。

有一点需要提一下，对于 stale install shard requests，shard sender 可能是一个 lag-behind server。为了使它更快地完成 shard migration 以 catch up，应该立即 reply OK，告知它 shard receiver 已经 install 了这个 shard。

apply eligibility checking 是如何做的？

不管是 client op 还是 admin op，都需要保证 exactly-once 语义。

对于 client op，client 端使用 clerk id 和 op id 给 client op 打 unique tag。server 端通过维护 max applied op id 来去重。

对于 admin op，是不是也可以用类似的方法呢？即给每个 admin op 分配一个 server id 和 op id，然后也维护一个类似的 map 结构来判重。这样的方法我没有试过，或许是可行的。在我的实现，我是针对不同的 admin op，来设计不同的判重条件的，如下：

（1）install config op：

首先，当且仅当这个 install config op 对应的是我们所需要的 next config。install config op 有两个来源：

由当前 server propose：此时，如果这个 install config op 是我们需要 install 的 next config，有 `kv.reconfigureToConfigNum == kv.config.Num` && `kv.reconfigureToConfigNum == op.Config.Num` && `op.ConfigNum == kv.config.Num + 1`

- 由同一个 group 中其它 server propose：此时，如果这个 install config op 是我们需要 install 的 next config，有 `kv.reconfigureToConfigNum == -1` && `op.ConfigNum == kv.config.Num + 1`
- 对于这两种来源，都需要执行这个 install config op，也就是说它们是或的关系。

除此之外，我们不能在 migrating 时 install config。因为这可能会停止 shard migration。

总结，apply 一个 install config op 的条件为：这个 install config op 是我们需要的 next config 且我们现在没有在做 shard migration。

（2）install shard op：

首先，我们必须为每个 install shard op 打一个 config num 的 tag，标识它是对应哪个 config 的。这是显然的，因为 server 可能会收到 stale 或 more up-to-date 的 shard data。

其次，我们不应该重复 install 一个 shard。这是因为，我们可能已经完成了 shard migration，开始 server client requests，则 shard 已经有一些更新了。如果重复 install 一个 shard，则 applied 这些更新就会丢失。

总结，apply 一个 install shard op 的条件为：这个 shard 对应的 config num tag 与 server 当前的 config 一致，且 server 当前正在 moving in 这个 shard。

（3）delete shard op：

首先，也要和 install shard 一样，判断 config num tag。

当 config 一致性判断通过之后，可以允许重复 delete 一个 shard，也可以拒绝重复 delete 一个 shard。我选择后者，因为我认为最好也保证 at-most-once 语义。

总结，apply 一个 install shard op 的条件为：这个 shard 对应的 config num tag 与 server 当前的 config 一致，且 server 当前正在 moving out 这个 shard。

propose eligibility checking 是如何做的？

为了尽可能地减少 execute 时被 discard 的 ops 的数量，我们在 apply 时所做的 eligibility checking，在 propose 时也需要做。

apply install shard 和 apply delete shard 是如何做的？

在我的设计中，shard migration 需要传输 shard data 和 max applied op id。后者用于实现 at most once 语义。

因为 shard migration 是从一个 group 到另一个 group，再到下一个 group。从时间维度来看，整个集群在任何一个时刻最多只有一个 group 会 serve 一个 shard。因此，不管我们是否保存 stale shard data，在 install shard data 时，采取 replace 的方式是安全的。

那么能否在 install shard data 时，采取 update 的方式呢？考虑 server 已有的、未删除的 stale shard data，以及 install shard op 中所包含的 new shard data。如果一个 key 同时存在于这两个 shard data 中，那么新的 value 会覆盖旧的 value，即旧的 value 被 shadow 了。如果一个 key 只存在于 new shard data 中，那么也没问题。但是，如果一

个 key 只存在于 stale shard data 中，那么之后的 Get 就会错误地拿到已经被 delete 的 value。

总结，如果在 apply delete shard 时，真正地将 shard data 删除了，而不是仅仅修改 shard state，那么在 apply install shard 时，采取 replace 和 update 的方式都可以。如果仅仅修改 shard state，那么在 apply install shard 时，只能采取 replace 的方式。

对于 shard migration，一次 migrate 多个 shards 还是一次一个 shard？

在某次 config change 中，一个 group 可能需要将多个 shards 发送给另一个 group。此时，可以选择一次传输多个 shards，也可以选择一次传输一个。

考虑 unreliable network 和 server crash：如果 RPC 被 discard 了，或者在 install shards 的过程中国呢，server crash 了，那么就需要重发多个 shards。而如果一次只传输一个，假设 RPC discard 和 server crash 都是非频繁情况，那么在这种情况下，只需要重发一个 shard。

考虑 throughput：如果是每个 shard 独立地进行 shard migration，那么当一个 group install 一个 shard 后，就可以立即开始 serve 这个 shard。而如果 shard migration 是 batch 的形式，考虑到传输多个 shards 的时间比传输单个 shard 更长，则 group 需要等待较长时间才可以开始 serve 这些 shards。

对于 shard migration，使用 push based 还是 pull based?

首先要指出的是，shard migration 并不是发生在两个 servers 之间，而是两个 groups 之间。

对于 pull based，receiver group 中的每个 server 会向 sender group 中的每个 server 发送一个 pull request，尝试拉取所需的 shard。这种拉取是按需的。

对于 push based，sender group 中的每个 server 会持续地、定期地向 receiver group 中的每个 server 发送一个 push request，推送所需传输的 shard。这种推送是 eager 的。

如果在集群正常运转、group 进度差不多的情况下，两种方式所需的 RPC 交互轮数和 shard 传输量都是一致的。但是如果出现一些“异常”情况，pull based 是更佳的。

使用 pull 时，shard sender 可能还没有准备好 shard，例如它的 config 比较滞后。此时 shard sender 会让 shard receiver 稍等一会再来询问。在这样的交互中，没有 shard 的

传输。

使用 push 时，当 shard receiver 收到 shard 时，它可能还没准备好，例如它的 config 比较滞后。此时 shard receiver 也会让 shard sender 稍等一会再发。在这样的交互中，总是存在 shard 的传输。

总结，考虑到 network bandwidth，显然是 pull based 的方式更经济。

但是，在我的实现中，使用的是 push based 的方式。可能是我当时认为 push based 的方式实现起来更直接。

考虑到 shard broadcast 会浪费很多 network bandwidth，因此可以考虑使用一种 leader lease 算法，让 group 选举出 leader，然后只需要在 leaders 之间进行 shard migration。待 leader 完成 shard 传输后，再让 leader 通过 paxos 层将 shard 操作同步给其它 servers，即通过 install shard op 或 delete shard op。

如何解决 Challenge 1: Garbage collection of not-serving shards？

challenge 1 需要解决这样一个问题：当一个 server 不再 serve 一个 shard 时，将 shard data 删除。这样的删除不应该影响后续的 shard migration。并且，当一个 group 中的所有的 servers 都 crash 再 restart 后，这样的删除也不会影响后续的 shard migration。

对于这个问题，解决的关键就是将 shard data 放到 install shard op 中，让 paxos 层去 replicate 这个 install shard op 以及其中的 shard data。这样即使 shard sender 将 shard data 删除了，lag-behind 的 shard receiver 也能正确地进行 shard migration。

至于 install shard op 中所存的 shard data，server 层在 apply 之后会立即删除 install shard op。paxos 层也会有一些 garbage collection 机制来删除 replicated install shard op。

由于 2015 6.824 版本用的是 paxos，且这个版本在 lab4 没有涉及 server crash 和 persistence，因此无法对我的实现进行测试。

如何解决 Challenge 2: Concurrent client requests during reconfiguring？

challenge 2 需要解决这样一个问题：每个 shard 单独 handle client requests，一个 migrating shard 不应该要求其它 shards 也 reject client requests。并且，当一个 shard install 后，它应该立即开始 serve client requests。

对于这个问题，解决的关键就是将 shard data 分离，每个 shard 有自己独立的 state。并且以单个 shard 为单位进行 shard migration。

同样，无法对我的实现进行测试。

遇到的一些问题

- 如果 service handler 采用 polling 的方式等待 op 被 apply，且没有 cache reply，那么在检测到 op 被 apply 了后、将要 reply 时也需要检查是否 serving。这样的检查对于 Get request 的回复是有必要的。因为如果在 reply 时，server 不再 serve 这个 request key，那么对应的这个 shard 可能已经处于 moving out 状态，甚至已经完成了 shard migration，这个 shard 被 delete 了。因此，在 reply 时再执行一次是否 serving 的检查是有必要的。
- map 和 slice 是类似 references 的东西，因此需要 deep clone。
- 测试中有 `string('0' + i)` 类似的代码，其目的是将 int 转换为 string。可能在老版本的 Go 中，这样是可行的。但是对于我所使用的 Go 版本 1.19.3，这样会报错，无法编译。可以使用 `strconv.Iota(i)` 来替代这些代码。
- 任何 for loop 都需要检查 isdead。并且，在 kill 一个 server 时，这些 loop 退出的顺序是不一定的，有可能产生一些问题。在我测试的过程中，就出现了一个问题（这个问题只会与我的代码中出现），后来被解决了。

获得的一些分布式编程经验

- 在 lab4中，我才引入了 race detect。没想到竟然在 paxos 代码中检测到了很多 race。这些 race 在之前的测试中竟然都没有 flow 出来。可以看出，paxos 在实现上的柔性比较大，也可能 是 lab 的测试不够强。所以，还是要做好单元测试，把基础部分测试完全了，再去做其他部分。这里有一点要提的是，如果开了 race，可能会导致某些测试的运行时间过长，进而超时。

- 在 apply 时所做的 eligible 检查，在 propose 时也要做。虽然这并不是必要的，但是是推荐的做法，可以避免很多无效的 propose。
- 对于 state 的 update，尤其是 state machine state 的 update，应该完全地用代码限制 eligible to update 的 state。例如，在 install config, install shard 和 delete shard 时，server 应该处于什么 state，就应该用代码把这些 state 完全写明。当且仅当这些 state 都满足时，才执行 state update。如果当前的 fields 不足以区分和标识各个状态，那么就细化 fields 或者添加新的 fields。
- 对于打印出来的日志，考虑使用正则表达式进行过滤。例如 `^.*(help).*$` 可以筛选出所有包含 `help` 字符串的行，把这些行置换为空行。再使用 `^\s*\n` 筛选出空行，将空行删除。

其它注意点

- 在我的实现中，有些东西是必要的，但是在其它实现中，可能这些东西是不必要的。因此，我提出的这些讨论并不具有普适性。
- 由于 2015 版的 6.824 还没有现在这么完善，因此测试也不尽完善。也就是说，我所实现的版本可能还有不少问题。