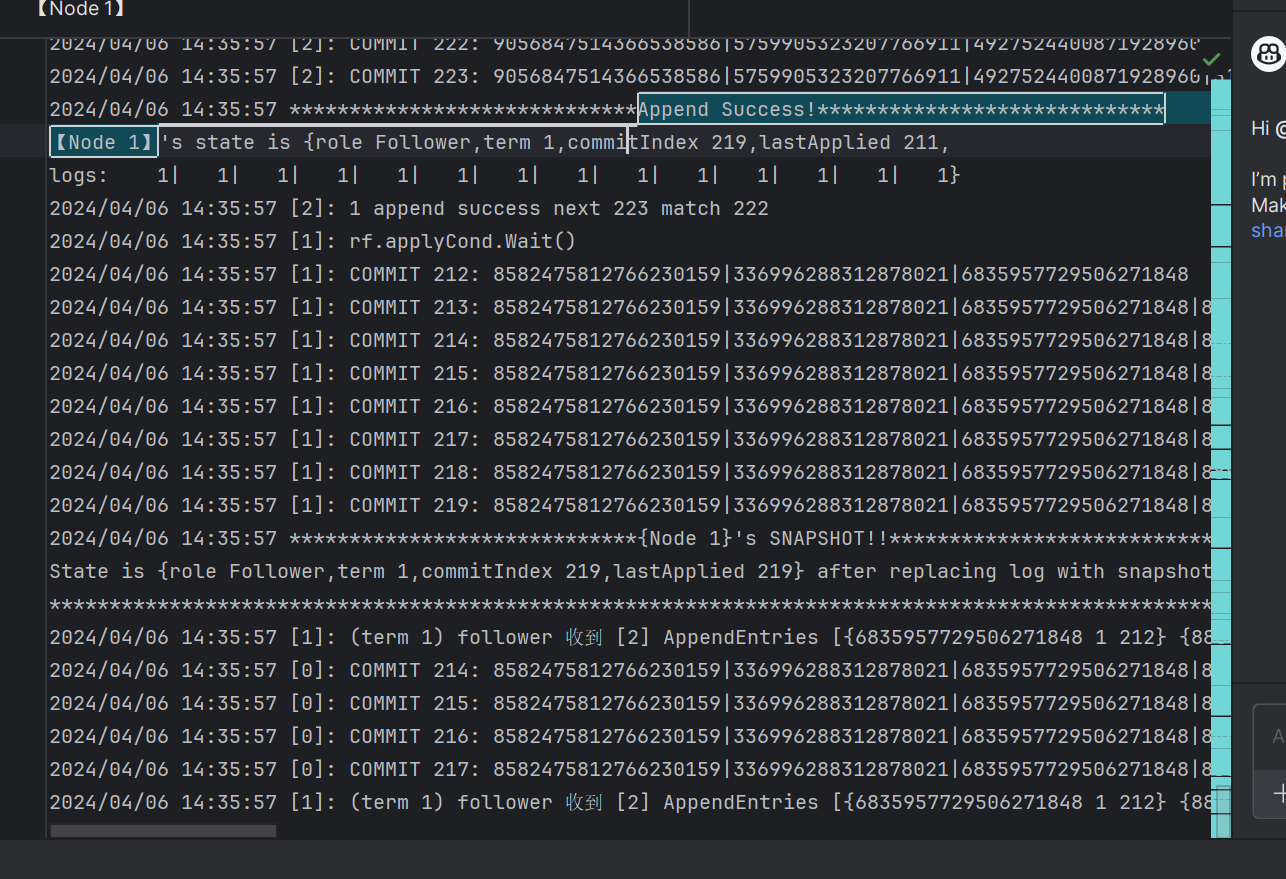


YEAH!!!

Lab2：

前三个根据图2写的比较简单，当写2D时，前面的逻辑都要改，很痛苦！

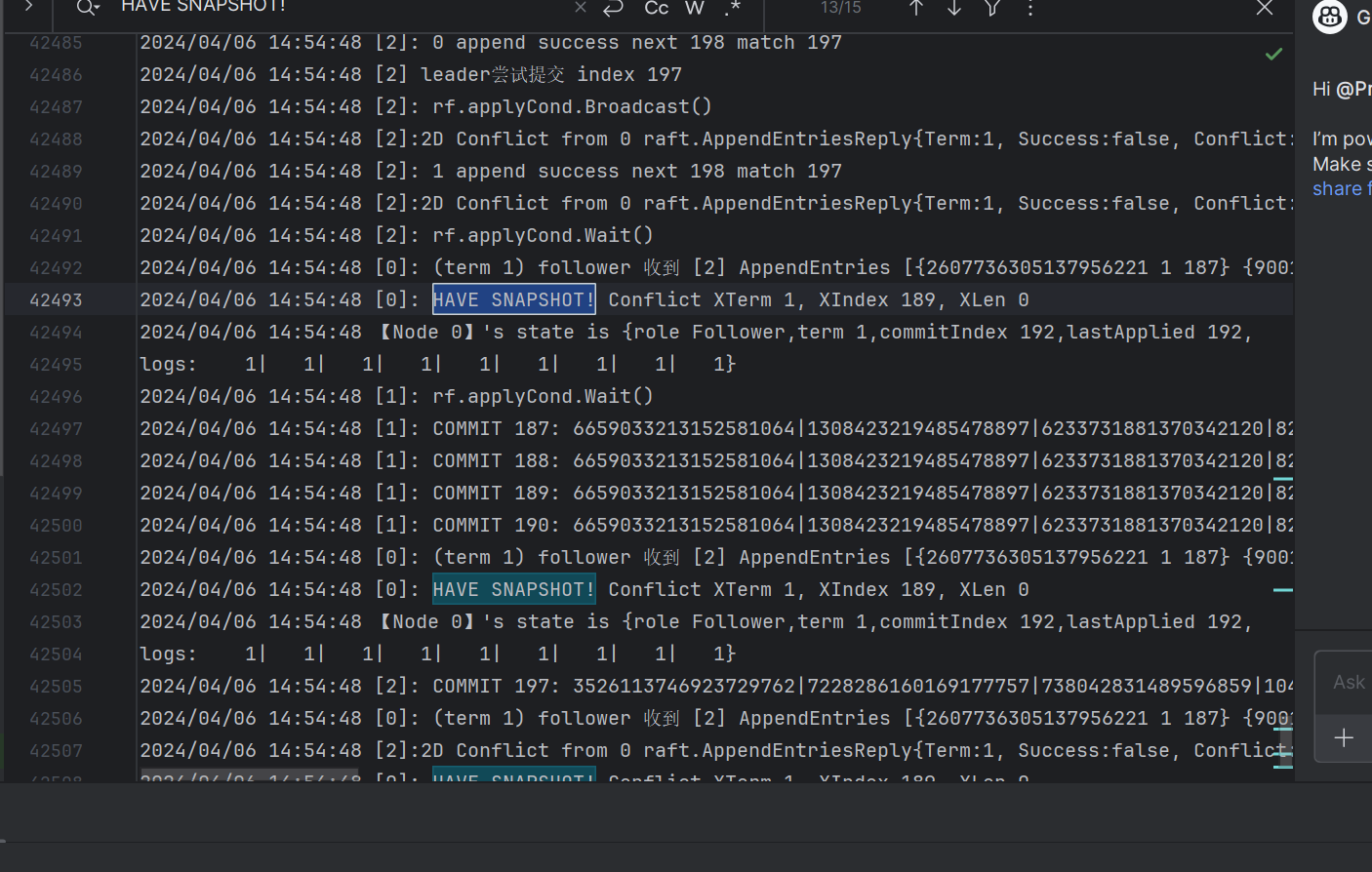


1接收成功，并记录快照后舍弃219后的日志

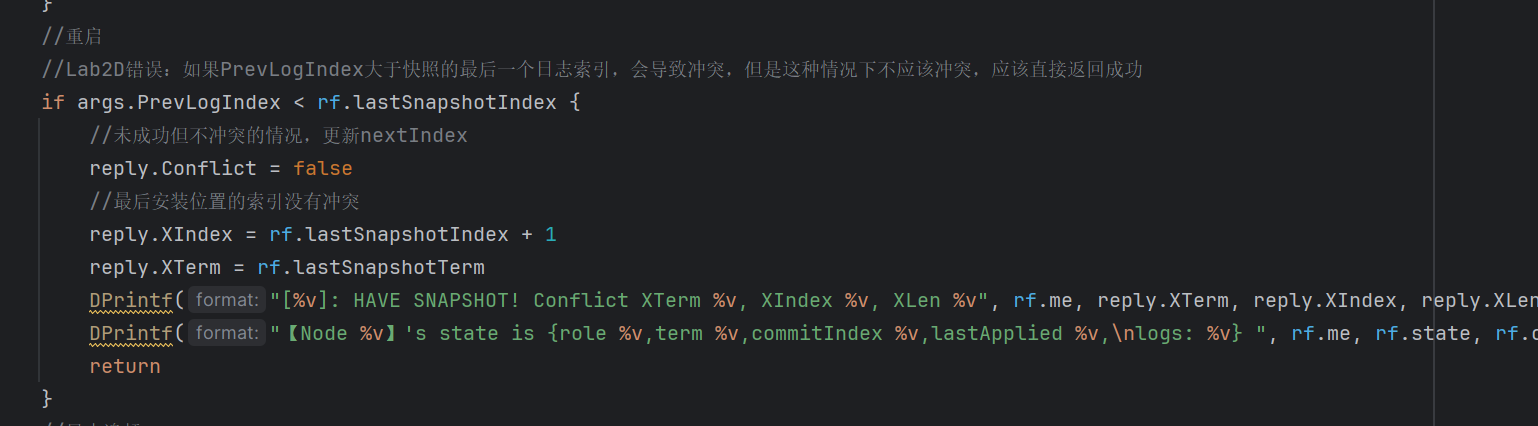
2还没更新，就又发送了一个appendentry，导致越界

AppendEntry内部锁了，但是leader的发送没锁

解决：



在receive appendentry的时候加入判定，如果已有更新的快照版本，则进行返回



问题二：持久化后，followers的nextindex已在磁盘？

问题三：开始做的时候，不过测试，预期有100个index，却出现104个

原因：

这个错误信息表示服务器0在应用日志时出现了顺序错误。它期望应用的日志索引为100，但实际上得到的日志索引为104。**这可能是因为在服务器崩溃并恢复后，它没有正确地从快照中恢复其状态，导致它试图跳过一些日志条目并直接应用索引为104的日志**。 你可以尝试以下步骤来解决这个问题：

确保在服务器崩溃后，你正确地从快照中恢复了服务器的状态。这包括恢复lastApplied和commitIndex的值，以及恢复日志数组。

确保在应用日志之前，你已经检查了日志的索引是否与lastApplied相等。如果不相等，那么你可能需要跳过一些日志条目，直到找到一个索引与lastApplied相等的日志条目。

确保你在应用日志之后，正确地更新了lastApplied的值。你应该将lastApplied设置为刚刚应用的日志的索引。

解决方法：持久化所有的日志条目，再从0开始apply（applych没法持久化？）

**问题4：**6.824 目前的持久化方式较为简单，每次发生一些变动就要把所有的状态都编码持久化一遍，这显然是生产不可用的。对IO耗时、网络带宽压力都很大。

解决方法：生产环境中，至少对于 raft 日志，应该是通过一个类似于 wal 的方式来顺序写磁盘。

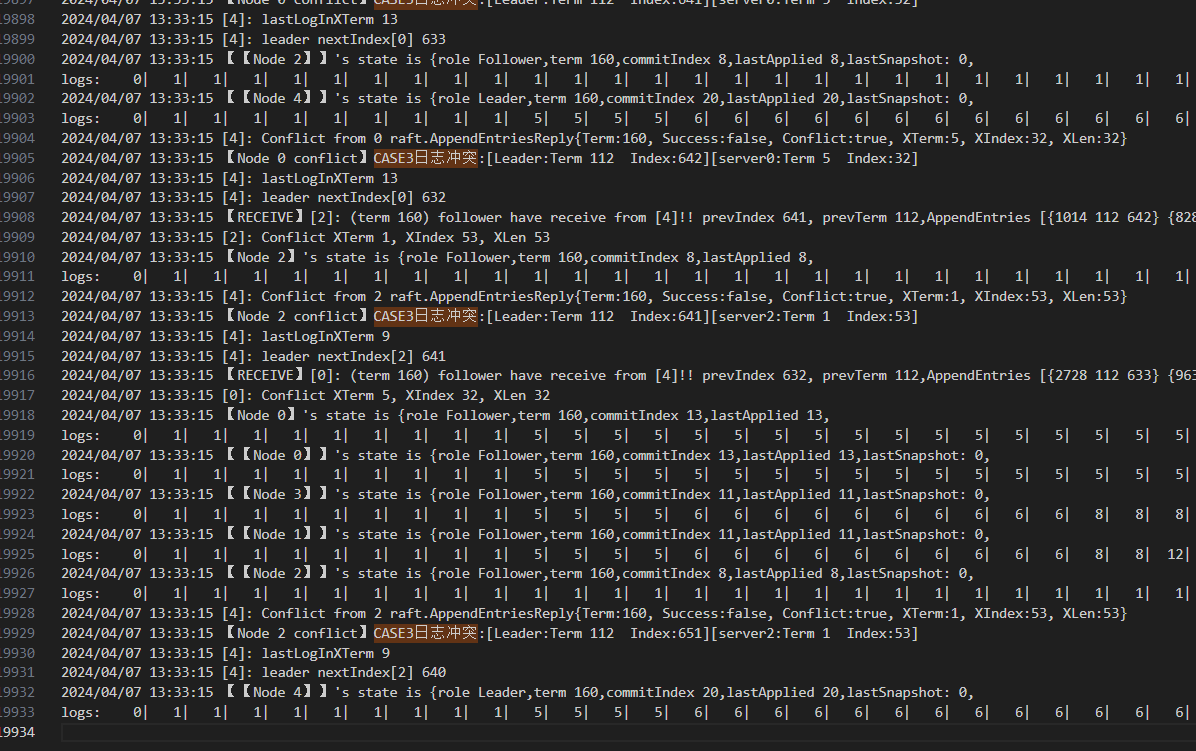
构想：

系统中的服务器使用wal，把执行操作的日志存储到磁盘中。Leader会定期进行快照并将快照持久化，快照中包含内存中的状态、磁盘中已成功提交的日志。Leader定期将日志快照发送给follower，如果follower发现自己的日志版本低于快照，则进行快照安装和日志追赶。

数据的快照本地存储，不通过网络传播，数据快照记录最后一条应用成功的日志编号，和wal的日志序列比对

问题5：

日志冲突靠递减恢复太慢



解决方法：

发生冲突时，将nextindex改为冲突日志任期的第一个日志



