到该位置之前的所有条目都已经发送给你，接下来再执行读取。这与ZooKeeper的sync()操作思路相同[15]。

- 可以从同步更新的副本上进行读取，这样确保总是读取最新值。这种技术可以用于链式复制[63]，具体参阅第5章的“复制研究”。

采用线性化存储实现全序关系广播

前面一节介绍了如何基于全序关系广播来构建线性化的原子比较-设置操作。我们也可以反过来，假定已有了线性化的存储，在其上构建全序关系广播。

最简单的方法是假设有一个线性化的寄存器来存储一个计数，然后使其支持原子自增-读取操作[28]或者原子比较-设置操作。

算法思路很简单：对于每个要通过全序关系广播的消息，原子递增并读取该线性化的计数，然后将其作为序列号附加到消息中。接下来，将消息广播到所有节点（如果发生丢失，则重新发送），而接受者也严格按照序列化来发送回复消息。

请注意，与Lamport时间戳不同，通过递增线性化寄存器获得的数字不会存在任何间隙。因此，如果节点完成了消息4的发送，且接收到了序列化6的消息，那么在它对消息6回复之前必须等待消息5。Lamport时间戳则不是这样，而这也是区别全序关系广播与基于时间戳排序的关键。

使用原子自增操作来创建线性化整数有多难呢？答案还是那样，如果不存在失效，就非常容易，甚至可以把它保存在某个节点的内存变量中。难点在于处理节点的网络中断，以及节点失效时如何恢复该值[59]。事实上，如果对线性化的序列号发生器深思熟虑之后所得到的最终结果，往往毫无意外地指向了共识算法。

这并非巧合，可以证明，线性化的原子比较-设置（或自增）寄存器与全序关系广播二者都等价于共识问题[28,67]。也就是说，如果你能解决其中的一个问题，那么就可以把方案用于解决其他问题。这样的结论是多么的深刻和震撼！

好了，现在终于是时候正面处理共识问题了，这是本章剩余部分的重点。

分布式事务与共识

共识问题是分布式计算中最重要也是最基本的问题之一。表面上看，目标只是让几个节点就某件事情达成一致。这似乎很简单，或者至少不应该太难。不幸的是，许多失败的系统正是由于低估了这个问题所导致的。