2.2 经典 Paxos

Heidi Howard

Distributed consensus revised.
PhD Thesis, Computer Laboratory, University of Cambridge, UCAM-CL-TR-935, 1–151, April 2019.
节选自原文第 27–32 页。
https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-935.pdf

译者：Ying ZHANG. 2021-06
https://ying-zhang.gitee.io/dist/2019-classic-paxos-cn.pdf
https://ying-zhang.github.io/dist/2019-classic-paxos-cn.pdf

经典 Paxos [Lam98] 是一种解决分布式共识问题的算法。在最好的情况下，未优化的算法通过与多数接受者（acceptors）2 次往返和对持久存储的 3 次同步写入后达成共识，但某些情况需要更多时间。活性（liveness）条件是 n_a 个接受者中的 $\lfloor n_a/2 \rfloor + 1$ 个，以及 1 个提案者（proposer）必须正常，且可以同步地通信（communicating synchronously）。这些条件是活性的充分必要条件。

经典 Paxos 决定（deciding）值的方法有 2 个阶段。第一阶段可以看作是读阶段，在这个阶段，提案者了解系统的当前状态并分配一个版本号以便稍后可以发现变动。第二阶段可以认为是写阶段，在这个阶段，提案者尝试让一个值被接受（accepted）。若经过算法的第一阶段，提案者确信尚未决定值，则提案者可以提出候选值 γ。若第一阶段的结果表明可能决定了某个值，则必须在第二阶段提出该值。

这两个阶段都需要多数接受者同意才能继续。

我们现在定义术语轮次（epoch）和提案（proposal），然后使用它们来总结经典 Paxos 算法。

定义 2. 轮次 e 是集合 E 的成员，而 E 是任意的无限全序集合，即对所有元素都定义了 $>$，$<$ 和 $=$ 运算。

定义 3. 提案 (e, v) 是任意轮次和任意值的组合。

经典 Paxos 的第一阶段

1. 提案者选择一个唯一的轮次 e 并向所有接受者发送 $prepare(e)$。
2. 每个接受者存储最后承诺（promised）的轮次和最后接受的提案。当接受者收到 $prepare(e)$，若 e 是首个轮次，或 e 不小于已承诺的最大轮次，就把 e 写入存储，且接受者回复 $promise(e, f, v)$〔“承诺”消息〕。其中 (f, v) 是最后接受的提案（若存在），f 是其轮次，v 是相应的值。
3. 一旦提案者收到多数接受者的 $promise(e, \_, \_)$，它就进入第二阶段。所述 “承诺” 消息可能包含最后接受的提案，下一阶段会用到该提案。
4. 否则，若提案者超时，它将以更大的轮次重试。

经典 Paxos 的第二阶段

1. 现在提案者必须使用以下选值规则选择一个值 v
   (a) 若第一阶段的 “承诺” 消息没有返回提案，则提案者选择自己的候选值 γ。
   (b) 若仅返回了一个提案，则选择其值〔译注：可以合并到下条规则〕。
   (c) 若返回了多个提案，则提案者必须选择轮次最大提案的值。

   然后提案者向所有接受者发送 $propose(e, v)$。
2. 每个接受者收到 $propose(e, v)$ 后，若 e 是首个轮次，或 e 不小于已承诺的最大轮次，则〔接受此轮提案,〕更新已承诺的轮次和已接受的提案，且接受者回复 $accept(e)$〔“接受” 消息〕。
3. 一旦提案者收到多数接受者的 $accept(e)$，它就知道值 v 被决定了。
4. 否则，若提案者超时，它将以更大的轮次重试第一阶段。

表 2.2 概述了经典 Paxos 中使用的 4 种消息作为参考。

下一小节，我们将更详细地研究这个算法。

表 2.2: 经典 Paxos 中交换的消息

	消息	说明	发出方	接收方
第一阶段	*prepare*(*e*)	*e*: 轮次	提案者	接受者
	promise(*e, f, v*)	*e*: 轮次 *f, v*: 最近接受的轮次，值（可为*nil*）	接受者	提案者
第二阶段	*propose*(*e, v*)	*e*: 轮次 *v*: 提案值	提案者	接受者
	accept(*e*)	*e*: 轮次	接受者	提案者

2.2.1 提案者的算法

算法 3 描述了经典 Paxos 算法中作为提案者角色的参与者的算法。该算法的关键输入是待提出的候选值 γ，输出是决定的值 v。决定值可能与候选值相同，也可能不同，这取决于运行算法时接受者的状态。若确认尚未决定值，则提案者提出自己的候选值 γ。若提案者获知决定了一个值，则任何其它提案者都不会获知不同的决定值。

初始化变量后（算法 3，第 1，2 行），算法从选择使用的轮次 e 开始（算法 3，第 3 行）。为了通用，我们没有明确如何生成可用轮次的集合 $\mathcal{E} \subseteq E$。但是，该算法确实要求每个提案者都有无限且不相交的轮次集合。该算法通过将当前轮次 e 从可用轮次集合中移除，来确保它只使用一次（算法 3，第 4 行）。为简单起见，我们让提案者按顺序尝试轮次，尽管提案者使用任意未用过的轮次都是安全的。

消息 $prepare(e)$ 发送给所有接受者（算法 3，第 5 行），提案者等待回复。收到承诺后，提案者跟踪收到提案的最大轮次 e_{max} 及对应的值 v（算法 3，第 8-11 行）。若承诺中不包含提案，则不更新最大轮次 e_{max} 及对应的值 v（算法 3，第 10 行）。集合 Q_P 记录目前为止做出承诺的接受者。若在超时之前没有收到多数接受者的承诺，则算法重试（算法 3，第 6，12，13 行）。若收到的承诺中都没有提案，则将提案值 v 设置为候选值 γ（算法 3，第 14，15 行）。

然后提案者向所有接受者发送 $propose(e, v)$（算法 3，第 16 行）。若多数接受者接受提案 $propose(e, v)$，则提案者（算法 3，第 23 行）返回决定的值 v（算法 3，第 17-20 行），否则重试（算法 3，第 21，22 行）。

请注意，提案者可以安全地忽略收到的不匹配 `switch` 的所有消息（例如收到的来自之前轮次的消息，或第二阶段期间收到的 “承诺” 消息）。

2.2.2 接受者的算法

经典 Paxos 中的接受者负责处理传入的 *prepare* 和 *propose* 消息。算法 4 中描述了此逻辑。所有消息，无论是 $prepare(e)$ 或者 $propose(e, v)$，其轮次 e 必须大于等于 e_{pro}〔译注：指已承诺的最大轮次〕才能由接受者处理（算法 4，第 4，8 行）。若这是接受者收到的第一条消息，则 e_{pro} 是空值 nil，检查也是通过的。若通过了检查，则令 e_{pro} 等于 e（算法 4，第 5，9 行）。

若消息是 $prepare(e)$，则接受者回复 $promise(e, e_{acc}, v_{acc})$（算法 4，第 6 行）。若接受者尚未接受过提案，则 e_{acc} 和 v_{acc} 都是空值 nil。当接受者发送了 “承诺” 消息，我们说接受者已经在轮次 e 做出承诺了。

若消息是 $propose(e, v)$，则接受者将 e_{acc} 和 v_{acc} 设置为提案的 (e, v)（算法 4，第 10 行）并回复 $accept(e)$（算法 4，第 11 行）。在这种情况下，我们说接受者**接受**了提案 (e, v)。

定义 4. 在经典 Paxos 中，若提案 (e, v) 已被多数接受者接受，则称决定了提案 (e, v)。

请注意，此定义不要求提案是多数接受者**最后**接受的提案。若存在轮次 $e \in E$，使提案 (e, v) 被决定了，则称值 $v \in V$ 被决定了。这也称为值 v 在轮次 e 被**决定**。确认点（commit point）是提案首次被决定的时刻。

Algorithm 3: Proposer algorithm for Classic Paxos

state:

- n_a: total number of acceptors (configured, persistent)
- e: current epoch
- v: current proposal value
- e_{max}: maximum epoch received in phase 1
- $\mathcal{E}$: set of unused epochs (configured, persistent)
- Q_P: set of acceptors who have promised
- Q_A: set of acceptors who have accepted

`/* (Re)set variables */`

1 $v, e_{max} \leftarrow nil$

2 $Q_P, Q_A \leftarrow \emptyset$

`/* Select and set the epoch` e `*/`

3 $e \leftarrow min(\mathcal{E})$

4 $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$

`/* Start Phase 1 for epoch` e `*/`

5 **send** *prepare(e) to acceptors*

6 **while** $|Q_P| < \lfloor n_a/2 \rfloor + 1$ **do**

7 **switch do**

8 **case** *promise(e,f,w) received from acceptor a*

9 $Q_P \leftarrow Q_P \cup \{a\}$

10 **if** $f \neq nil \wedge (e_{max} = nil \vee f > e_{max})$ **then**

`/* (`e_{max}, v`) is the greatest proposal received */`

11 $e_{max} \leftarrow f$, $v \leftarrow w$

12 **case** *timeout*

13 **goto** *line 1*

14 **if** $v = nil$ **then**

`/* no proposals were received thus propose` γ `*/`

15 $v \leftarrow \gamma$

`/* Start Phase 2 for proposal (`e`,`v`) */`

16 **send** *propose(e,v) to acceptors*

17 **while** $|Q_A| < \lfloor n_a/2 \rfloor + 1$ **do**

18 **switch do**

19 **case** *accept(e) received from acceptor a*

20 $Q_A \leftarrow Q_A \cup \{a\}$

21 **case** *timeout*

22 **goto** *line 1*

23 **return** v

Algorithm 4: Acceptor algorithm for Classic Paxos

state:

- e_{pro}: last promised epoch (persistent)
- e_{acc}: last accepted epoch (persistent)
- v_{acc}: last accepted **value** (persistent)

```
1  while true do
2      switch do
3          case prepare(e) received from proposer
4              if e_pro = nil ∨ e ≥ e_pro then
5                  e_pro ← e
6                  send promise(e,e_acc,v_acc) to proposer
7          case propose(e,v) received from proposer
8              if e_pro = nil ∨ e ≥ e_pro then
9                  e_pro ← e
10                 v_acc ← v, e_acc ← e
11                 send accept(e) to proposer
```