Paxos究竟在解决什么问题?

Paxos用来确定一个不可变变量的取值

- 取值可以是任意二进制数据
- 一旦确定将不再更改，并且可以被获取到（不可变性、可读取性）

在分布式存储系统中应用Paxos

- 数据本身可变，采用多副本进行存储
- 多个副本的更新操作序列[P1,P2,...PN]是相同的、不变的。
- 用Paxos依次来确定不可变变量p1的取值（即第i个操作是什么）。

设计一个系统，来存储名称为var的变量?

- 系统内部由多个Acceptor组成，负责存储和管理var变量
- 外部有多个proposer机器任意并发调用API，向系统提交不同的var取值
- var的取值可以是任意二进制数据。
- 系统对外API库接口为：propose(var, V) => <ok, f> or <error>

系统需要保证var的取值满足一致性

- 如果var的取值没有确定，则var的取值为null
- 一旦var的取值被确定，则不可被更改。并且可以一直获取到这个值。

系统需要满足容错特性

- 可以容忍任意proposer机器出现故障
- 可以容忍少数Acceptor故障（半数以下）

怎么样确定一个不可变变量

- 管理多个proposer的并发执行

----------管理多个proposer的并发执行----------

方案1：

1.先考虑系统由单个Acceptor组成。通过类似互斥锁机制，来管理并发的proposer运行。

2.Proposer首先向acceptor申请acceptor的互斥访问权限，然后才能请求Acceptor接受自己的取值

3.Acceptor给proposer发放互斥访问权，谁申请到互斥访问权，就接收谁提交的取值。

4.让Proposer按照获取互斥访问权的顺序依次访问acceptor

5.一旦Acceptor接收了某个proposer的取值，则认为var取值被确定，其他Proposer不再更改

```
基于互斥访问权的Acceptor的实现
    1.Acceptor保存变量var和一个互斥锁lock
    2.Acceptor::preprare(): 加互斥锁，给与var的互斥访问权限，并返回var当前的取值f
    3.Acceptor::release(): 解互斥锁，收回var的互斥访问权
    4.Acceptor::accept(var, V): 如果已经加锁，并且var没有取值，则设置var为V。并且释放锁。
propose(var ,V)的两阶段实现
    第一阶段：通过Acceptor::prepare获取互斥访问权和当前var的取值，如果不能返回error，此时锁被占用
    第二阶段：根据当前var 的取值f，选择执行
    1.如果f为null，则通过Acceptor::accept(var, V)提交数据V。
    2.如果f不为空，则通过Acceptor::release()释放访问权，返回<ok, f>
```

通过Acceptor互斥访问权让Proposer序列运行，可以简单的实现var取值的一致性。

Proposer在释放互斥访问权之间发生故障，会导致系统陷入死锁。

1.不能容忍任意Proposer机器故障

方案2：

- 引入抢占式访问权

1.acceptor可以让某个proposer获取到的访问权失效，不再接收它的访问

2.之后，可以将访问权发放给其他proposer，让其他proposer访问acceptor。

Proposer向Acceptor申请访问权时指定编号epoch（越大的epoch越新），获取到访问权之后，才能向acceptor提交取值。

- **Proposer向Acceptor申请访问权时指定编号epoch（越大的epoch越新），获取到访问权之后，才能向acceptor提交取值。**
- **Acceptor采用喜新厌旧的原则**
  1.一旦收到更大的新epoch的申请，马上让旧epoch的访问失效，不再接收他们提交的取值。
  2.然后给新epoch发放访问权，只接收新epoch提交的取值
- **新epoch可以抢占旧epoch，让旧epoch的访问权失效。旧epoch的proposer将无法运行，新epoch的proposer将开始运行。**
- **为了保持一致性，不同epoch的proposer之间采用"后者认同前者"的原则**
  1.在肯定旧epoch无法生成确定性取值时，新的epoch会提交自己的value。不会冲突。
  2.一旦旧epoch形成确定性取值，新的epoch肯定可以获取到此取值，并且会认同此取值，不会破坏。

```
基于抢占式访问权的Acceptor的实现
1.Acceptor保存的状态
    -当前var的取值<accepted_epoch,accepted_value>
    -最新发放访问权的epoch(latest_prepared_epoch)
2.Acceptor::prepare(epoch):
    -只接收比latest_prepared_epoch更大的epoch，并给与访问权
    -记录latest_prepared_epoch=epoch；返回当前var的取值
3.Acceptor::accept(var ,  prepared_epoch, V):
    -验证latest_prep = prepared_epoch
    -并设置var的取值<acceptor_epoch, accepted_value> = <prepared_epoch, v>

Propose(var , V)的两阶段实现
第一阶段：获取epoch轮次的访问权和当前var的取值
     简单选取当前时间戳为epoch，通过Acceptor::prepare(epoch)，获取epoch轮次的访问权和当前var的取值
     如果不能获取，则返回error
第二阶段：采用后者认同前者的原则执行
     在肯定旧epoch无法生成确定性取值时，新的epoch会提交自己的value，不会冲突。
     一旦旧epoch形成确定性取值，新的epoch肯定可以获取到此取值，并且会认同此取值，不会破坏。
     if:
     如果var的取值为空，则肯定旧epoch无法生成确定性取值，则通过Acceptor::accept(var,epoch,V)提交数据V。成功后返
     如果accept失败，返回error，新epoch抢占或者acceptor故障
     if:
     如果var取值存在，则此取值肯定是确定性取值，此时认同它不再更改，直接返回<ok ,accepted_value>

```

基于抢占式访问权的核心思想
1.让Proposer将暗战epoch递增的顺序抢占式的依次运行，后者会认同前者
2.可以避免proposer机器故障带来死锁问题，并且仍可以保证var取值的一致性
3.仍需要引入多acceptor
单机模块Acceptor是故障导致整个系统宕机，无法提供服务

思考：
方案1：
1.如何控制proposer的并发运行？
2.为何可以保证一致性？
3.为什么会有死锁问题？
方案2：

1.如何解决方案1的死锁问题?

2.在什么情况下，proposer可以将var的取值确定为自己提交的取值?

3.如何保证新epoch不会破坏已经达成的确定性取值?

Paxos在方案2的基础上引入多Acceptor。

Acceptor的实现保持不变。仍采用“喜新厌旧”的原则运行。

Paxos采用“少数服从多数”的思路

一旦某epoch的取值f被半数以上的acceptor接受，则认为此var取值被确定为f，不再更改。

```
确定一个不可变变量的取值 -Paxos
Propose(var ，V) 第一阶段：选定epoch，获取epoch访问权和对应的var取值
    -获取半数以上acceptor的访问权和对应的一组var取值
Propose (var，V) 第二阶段：采用“后者认同前者”的原则执行
    -在肯定旧epoch无法生成确定性取值时，新的epoch会提交自己的取值，不会冲突。
    -一旦旧epoch形成确定性取值，新的epoch肯定可以获取到此取值，并且会认同此取值，不会破坏。
    如果获取的var取值都为空，则旧epoch无法形成确定性取值。此时努力使<epoch，V>成为确定性取值
        -向epoch对应的所有acceptor提交取值<epoch,V>
        -如果收到半数以上成功，则返回<ok，V>
        -否则，则返回<error>(被新epoch抢占或者acceptor故障)
     如果var的取值存在，认同最大accepted_epoch对应的取值f，努力使<epoch，f>成为确定性取值:
         -如果f出现半数以上，则说明f已经是确定性取值，直接返回<ok，f>
         -否则，向epoch对应的所有acceptor提交取值<epoch，f>
```

----------管理多个proposer的并发执行----------

- 保证var变量的不可变性
- 容忍任意Proposer机器故障
- 容忍半数以下Acceptor机器故障

Paxos如何在分布式存储系统中应用?

Paxos算法的核心思想是什么?

1.在抢占式访问权的基础上引入多acceptor

2.保证一个epoch，只有一个proposer运行，proposer按照epoch递增顺序依次运行

3.新epoch的proposer采用“后者认同前者”的思路运行

-在肯定旧epoch无法生成确定性取值时，新的epoch会提交自己的取值，不会冲突

-一旦旧epoch形成确定性取值，新的epoch肯定可以获取到此取值，并且会认同此取值，不会破坏

Paxos算法可以满足容错要求

1.半数以下acceptor出现故障时，存活的acceptor仍然可以生成var的确定性取值

2.一旦var取值被确定，即使出现半数以下acceptor故障，此取值可以被获取，并且将不再被更改

Paxos算法的Liveness问题

新轮次的抢占会让旧轮次停止运行，如果每一轮次在第二阶段执行成功之前都被新一轮抢占，则导致活锁。怎么解决。

1.在什么情况下可以认为var的取值被确定，不再更改?

2.Paxos的两个阶段分别在做什么?

3.一个epoch是否会有多个proposer进入第二阶段运行?

4.在什么情况下，proposer可以将var的取值确定为自己提交的取值?

5.在第二阶段，如果获取的var取值都为空，为什么可以保证旧epoch无法形成确定性取值?

6.新epoch抢占成功之后，旧epoch的proposer将如何运行?

7.如何保证新epoch不会破坏已经达成的确定性取值?

8.为什么在第二阶段存在var取值时，只需考虑accepted_epoch最大的取值f?

9.在形成确定性取值之后出现任意半数以下acceptor故障，为何确定性取值不会被更改?

10.如果proposer在运行过程中，任意半数以下的acceptor出现故障，此时将如何运行?

11.正在运行的proposer和任意半数以下的acceptor都出现故障时，var的取值可能是什么情况?为何之后新proposer可以形成确定性取值?