1，paxos用来确定一个不可变变量的取值

取值可以是任意一个二进制数据

一旦确定将不再更改，并且可以被获取到（不可变性，可读性）

2，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Op1,...,Opn是相同的，不变的

用paxos依次来确定不可变变量Opi的取值，即第i个操作是什么

每确认完Opi之后，让各个数据副本执行Opi，依次类推

3，paxos希望解决的一致性问题

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

系统对外的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

可以容易任意proposer机器出现故障

可以容忍少数acceptor故障，半数一下

难点

管理多个proposer的并发执行

保证var变量的不变形

proposer机器故障

半数一下acceptor故障

方案一

确定一个不可变变量的取值

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，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，就接受谁提交的取值

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取值被确定了，其他proposer不再更改

方案一的系统设计：基于互斥访问权的acceptor的实现

acceptor保存变量var和一个互斥锁lock

acceptor::preprare():

加互斥锁，给予var的互斥访问权，并返回var当前的取值

acceptor::release():

解互斥锁，收回var的互斥访问权

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>，锁被别人占用

第二阶段：

如果f为null，则通过acceptor::accept(var,V)，提交数据v

如果f不为空，则通过acceptor::release()释放互斥锁，返回<ok,f>

方案一通过acceptor互斥访问权让proposer序列运行，可以简单实现var取值的一致性。

Proposer在释放互斥访问权之前发生故障，会导致系统陷入死锁

不能容忍任意proposer机器故障

为解决因为proposer机器故障导致系统陷入死锁问题，引入方案二。

方案二：抢占式访问权

acceptor可以让某个proposer获取到的访问权限失效，不再接受它的访问。

之后，可以将访问权限发给其他的proposer，让其他的proposer访问acceptor。

Proposer向acceptor申请访问权时指定编号epoch（编号的越大的epoch越新），获取的访问权后才可以向acceptor提交取值

Acceptor采用喜新厌旧的原则

一旦收到更大的新epoch申请，马上让旧的epoch的访问权失效，不再接受他们提交的取值。

然后给新的epoch发放访问权限，只接受新的epoch提交的取值

新的epoch可以抢占旧epoch，让旧epoch的访问权限失效，旧的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。

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

一旦旧的epoch形成了确定的取值，新的epoch肯定可以获取到此取值，并且认同此取值，不会破坏。

方案二：基于抢占式访问权的实现

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)

acceptor::prepare(epoch)

只接受比latest\_prepared\_epoch更大的epoch，并给予访问权

记录latest\_prepared\_epoch=epoch；返回当前var对应的取值

acceptor::accept(var,prepared\_epoch,V)

验证latest\_prepared\_epoch==prepared\_epoch

并且设置var的取值<accepted\_epoch,accepted\_value> = <prepared\_epoch,V>

proposer实现方式分为两阶段

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::prepared(epoch)，获取epoch轮次的访问权和当前var的取值

如果不能获取，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肯定可以获取到此值，并且会认同此值，不会破坏

如果var为空，说明旧的epoch无法生成确定性取值，则通过acceptor::accept(var,prepared\_epoch,V)提交数据V。成功后返回<ok,V>

如果accept失败，返回<error>（被新的epoch抢占或者accept故障）

如果var取值存在，则此取值肯定是确定性取值，此时认同它不做更改，直接返回<ok,accepted\_value>

方案二具体执行过程

step:1

<ok,null,null>

Prepare(#1)

P2

P3

P1

Acceptor

accept\_epoch,accept\_value

latest\_prepared\_epoch

#1

step2

Accept(#2,v2)

P1

Acceptor

accept\_epoch,accept\_value

#2,v2

latest\_prepared\_epoch

#2

<ok,#2,v2 >

Prepare(#3)

P3

Accept(#1,v1)

<ok,#2,v2>

<ok,null,null>

Prepare(#2)

P2

方案二存在的问题：

1. acceptor单点故障

paxos协议可以解决方案二的问题。引入多acceptor

accept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acceptor接受，则认为此var取值被确定为f，不再更改

paxos proposer处理机制

第一阶段：选定epoch，获取epoch访问权限和对应的var取值

获取半数以上acceptor的访问权限和对应的一组var取值

第二阶段：采用“后者认同前者”的原则执行

在肯定旧epoch无法形成确定性取值的时候，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肯定可以获取到取值，并且认同取值，不会破坏

如果获取的var为空，说明旧的epoch无法形成确定性取值，此时努力使<epoch,V>成为确定性取值。

1. 向epoch对应的所有的acceptor提交取值<epoch,V>
2. 如果收到半数以上成功，则返回<ok,V>
3. 否则，返回<error>(被新的epoch抢占或者acceptor故障)

如果var取值存在，认同最大accepted\_epoch对应的取值f，努力使<epoch,f>成功确定性取值。

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>

step1

<ok,null,null>

Prepare(#1)

<ok,null,null>

Prepare(#1)

<ok,null,null>

Prepare(#1)

P2

P1

Acceptor3

accept\_epoch,accept\_value

latest\_prepared\_epoch

#1

Acceptor2

accept\_epoch,accept\_value

latest\_prepared\_epoch

#1

Acceptor1

accept\_epoch,accept\_value

latest\_prepared\_epoch

#1

step2

<ok,#1,v1>

accept(#1,v1)

P1

Acceptor3

accept\_epoch,accept\_value

#1,v1

latest\_prepared\_epoch

#1

Acceptor2

accept\_epoch,accept\_value

latest\_prepared\_epoch

#2

Acceptor1

accept\_epoch,accept\_value

#1,v1

latest\_prepared\_epoch

#2

<ok,#1,v1>

accept(#1,v1)

accept(#1,v1)

Prepare(#2)

<ok,null,null>

Prepare(#2)

<ok,#1,v1>

P2

step3

P1

Acceptor3

accept\_epoch,accept\_value

#1,v1

latest\_prepared\_epoch

#1

Acceptor2

accept\_epoch,accept\_value

#2,v1

latest\_prepared\_epoch

#2

Acceptor1

accept\_epoch,accept\_value

#2,v1

latest\_prepared\_epoch

#2

<ok,#1,v1>

accept(#1,v1)

accept(#2,v1)

<ok,#2,v1>

accept(#2,v1)

P2

<ok,#2,v1>