Curve元数据节点高可用

1. 需求

mds是元数据节点，负责空间分配，集群状态监控，集群节点间的资源均衡等，mds故障可能会导致client端无法写入。

因此，mds需要做高可用。满足多个mds，但同时只有一个mds节点提供服务，称该提供服务的mds节点为主，等待节点为备；主节点的服务挂掉之后，备节点能启动服务，尽量减小服务中断的时间。

需要解决的问题就是：如何确定主备节点。

2. 技术选型

提供配置共享和服务发现的系统比较多，其中最为大家熟知的就是zookeeper和etcd，考虑当前系统中mds有两个外部依赖模块，一是mysql，
用于存储集群拓扑的相关信息；二是etcd，用于存储文件的元数据信息。而etcd可以用于实现mds高可用，没必要引入其他组件。

使用etcd实现元数据节点的leader主要依赖于它的两个核心机制：TTL和CAS。TTL(time to
live)指的是给一个key设置一个有效期，到期后key会被自动删掉。这在很多分布式锁的实现上都会用到，可以保证锁的实时性和有效性。CAS(Atomic
Compare-and-Swap)指的是在对key进行赋值的时候，客户端需要提供一些条件，当这些条件满足后才能赋值成功。

3. etcd clientv3的concurrency介绍

3.1 etcd clientV3的concurrency模块构成

etcd clientV3的concorrency模块对election进行了封装，首先对该模块做一个详细的介绍。

定义了Election的接口：

```
type Election struct {
   session *Session // etcd serversession
   keyPrefix string //
   leaderKey string // leaderkey
   leaderRev int64  // leaderkeyrevision
   leaderSession *Session  // leaderSession  sessionnil
   hdr *pb.ResponseHeader  // response
}
```

Election提供的方法如下：

```
// Campaign puts a value as eligible for the election on the prefix
// key.
// Multiple sessions can participate in the election for the
// same prefix, but only one can be the leader at a time.
//
// If the context is 'context.TODO()/context.Background()', the Campaign
// will continue to be blocked for other keys to be deleted, unless server
// returns a non-recoverable error (e.g. ErrCompacted).
// Otherwise, until the context is not cancelled or timed-out, Campaign will
// continue to be blocked until it becomes the leader.
// Campaignkeyetcd serversessionleader
// contxtkey-value etcd
// contxtcancelleader
func (e *Election) Campaign(ctx context.Context, val string) error

// Proclaimleadervalue
func (e *Election) Proclaim(ctx context.Context, val string) error

// Observe leaderleader,leader
// leader
func (e *Election) Observe(ctx context.Context) <-chan v3.GetResponse

// Resignleaderkey-value
func (e *Election) Resign(ctx context.Context) (err error)

// leaderkeyleader
func (e *Election) Key() string

// leaderkeyrevision
func (e *Election) Rev() int64

// response header
func (e *Election) Header() *pb.ResponseHeader
```

我们主要是用其中两个方法:

1. Campagin用于leader竞选
2. Observe用于监测集群中leader的变化

3.2 Campaign的流程

3.2.1 代码流程说明

如对相关代码实现不感兴趣，请直接跳到 3.2.2 举例说明Campagin流程

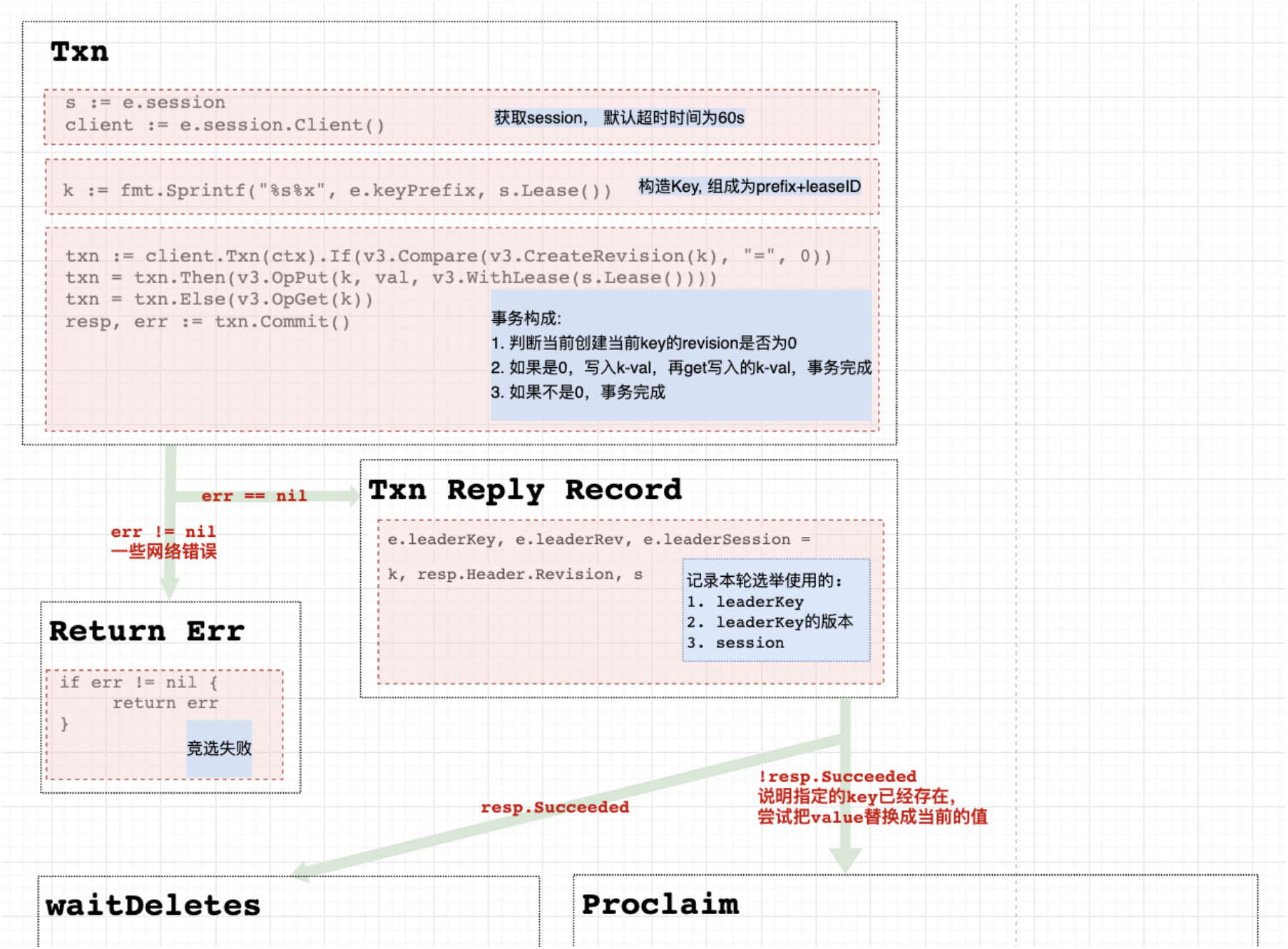
按照官方对Campagin的定义： blocked until it becomes the leader

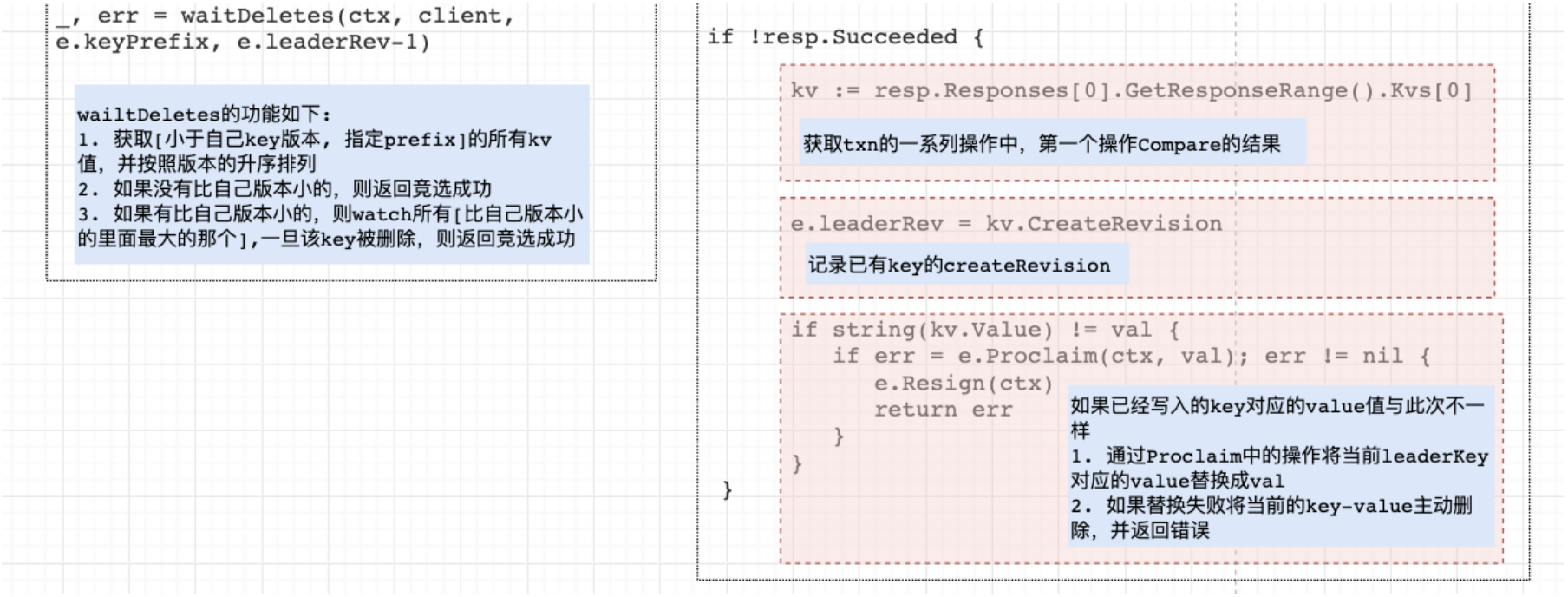
```
func (e *Election) Campaign(ctx context.Context, val string) error {
   s := e.session
   client := e.session.Client()

   k := fmt.Sprintf("%s%x", e.keyPrefix, s.Lease())
   txn := client.Txn(ctx).If(v3.Compare(v3.CreateRevision(k), "=", 0))
   txn = txn.Then(v3.OpPut(k, val, v3.WithLease(s.Lease())))
   txn = txn.Else(v3.OpGet(k))
   resp, err := txn.Commit()
   if err != nil {
      return err
   }
   e.leaderKey, e.leaderRev, e.leaderSession = k, resp.Header.Revision, s
   if !resp.Succeeded {
      kv := resp.Responses[0].GetResponseRange().Kvs[0]
      e.leaderRev = kv.CreateRevision
      if string(kv.Value) != val {
         if err = e.Proclaim(ctx, val); err != nil {
            e.Resign(ctx)
            return err
         }
      }
   }

   _, err = waitDeletes(ctx, client, e.keyPrefix, e.leaderRev-1)
   if err != nil {
      // clean up in case of context cancel
      select {
```

```
        case <-ctx.Done():
            e.Resign(client.Ctx())
        default:
            e.leaderSession = nil
        }
        return err
    }
    e.hdr = resp.Header
```

```
    return nil
}
```

代码流程说明如下：


Txn
s := e.session
client := e.session.Client()
获取session， 默认超时时间为60s
k := fmt.Sprintf("%s%x", e.keyPrefix, s.Lease())
构造Key, 组成为prefix+leaseID
txn := client.Txn(ctx).If(v3.Compare(v3.CreateRevision(k), "=", 0))
txn = txn.Then(v3.OpPut(k, val, v3.WithLease(s.Lease())))
txn = txn.Else(v3.OpGet(k))
resp, err := txn.Commit()
事务构成:
1. 判断当前创建当前key的revision是否为0
2. 如果是0，写入k-val，再get写入的k-val，事务完成
3. 如果不是0，事务完成
err == nil
err != nil
一些网络错误
Txn Reply Record
e.leaderKey, e.leaderRev, e.leaderSession =
k, resp.Header.Revision, s
记录本轮选举使用的：
1. leaderKey
2. leaderKey的版本
3. session
Return Err
if err != nil {
return err
}
竞选失败
resp.Succeeded
!resp.Succeeded
说明指定的key已经存在，
尝试把value替换成当前的值
waitDeletes
Proclaim


_, err = waitDeletes(ctx, client, e.keyPrefix, e.leaderRev-1)
wailtDeletes的功能如下:
1. 获取[小于自己key版本, 指定prefix]的所有kv值, 并按照版本的升序排列
2. 如果没有比自己版本小的, 则返回竞选成功
3. 如果有比自己版本小的, 则watch所有[比自己版本小的里面最大的那个],一旦该key被删除, 则返回竞选成功
if !resp.Succeeded {
kv := resp.Responses[0].GetResponseRange().Kvs[0]
获取txn的一系列操作中, 第一个操作Compare的结果
e.leaderRev = kv.CreateRevision
记录已有key的createRevision
if string(kv.Value) != val {
if err = e.Proclaim(ctx, val); err != nil {
e.Resign(ctx)
return err
}
}
}
如果已经写入的key对应的value值与此次不一样
1. 通过Proclaim中的操作将当前leaderKey对应的value替换成val
2. 如果替换失败将当前的key-value主动删除, 并返回错误

Revisions

etcd maintains a 64-bit cluster-wide counter, the store revision, that is incremented each time the key space is modified. The revision serves as a global logical clock, sequentially ordering all updates to the store. The change represented by a new revision is incremental; the data associated with a revision is the data that changed the store. Internally, a new revision means writing the changes to the backend's B+tree, keyed by the incremented revision.

Revisions become more valuable when considering etcd3's multi-version concurrency control backend. The MVCC model means that the key-value store can be viewed from past revisions since historical key revisions are retained. The retention policy for this history can be configured by cluster administrators for fine-grained storage management; usually etcd3 discards old revisions of keys on a timer. A typical etcd3 cluster retains superseded key data for hours. This also provides reliable handling for long client disconnection, not just transient network disruptions: watchers simply resume from the last observed historical revision. Similarly, to read from the store at a particular point-in-time, read requests can be tagged with a revision to return keys from a view of the key space at the point-in-time that revision was committed.

etcd中的revision是全局的，只要有key-value的修改(put, delete txn)，revision都会增加。举例说明：

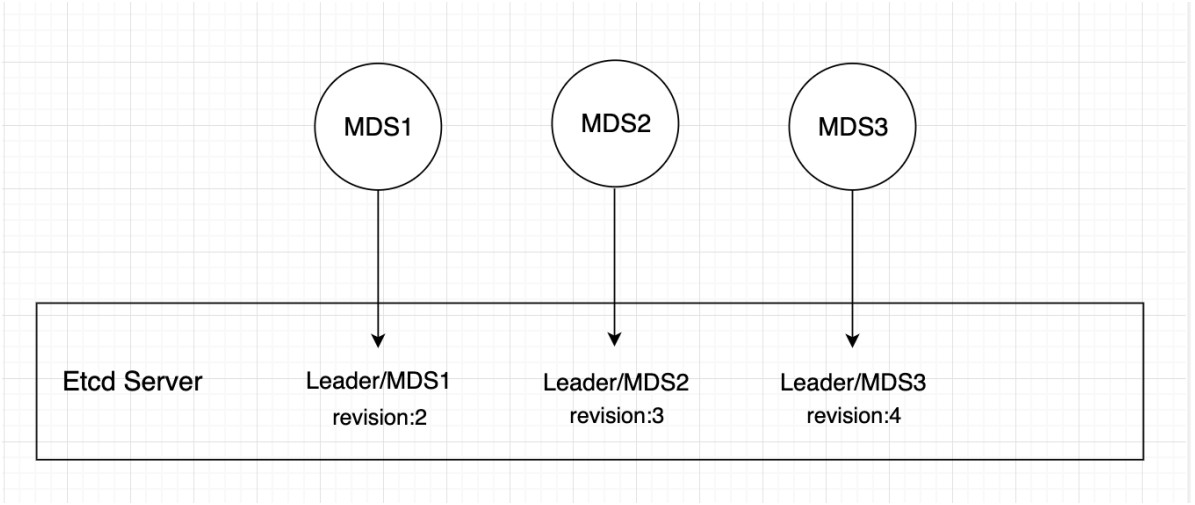
```
$ ETCDCTL_API=3 ./bin/etcdctl put foo bar

$ ETCDCTL_API=3 ./bin/etcdctl get foo --write-out=json
revision: 2

$ ETCDCTL_API=3 ./bin/etcdctl put foo bar

$ ETCDCTL_API=3 ./bin/etcdctl get foo --write-out=json
revision: 3

$ ETCDCTL_API=3 ./bin/etcdctl put hello world

$ ETCDCTL_API=3 ./bin/etcdctl get foo --write-out=json
revision: 4

$ ETCDCTL_API=3 ./bin/etcdctl get hello --write-out=json
revision: 4

$ ETCDCTL_API=3 ./bin/etcdctl put hello world

$ ETCDCTL_API=3 ./bin/etcdctl get hello --write-out=json
revision: 5
```

3.2.2 举例说明Campagin流程

场景描述：三个mds(mds1, mds2, mds3)，希望实现一个mds作为主提供服务，另外两个mds作为备在主挂掉的时候提供服务的功能。如果利用上述的Campagin进行选举，过程如下：

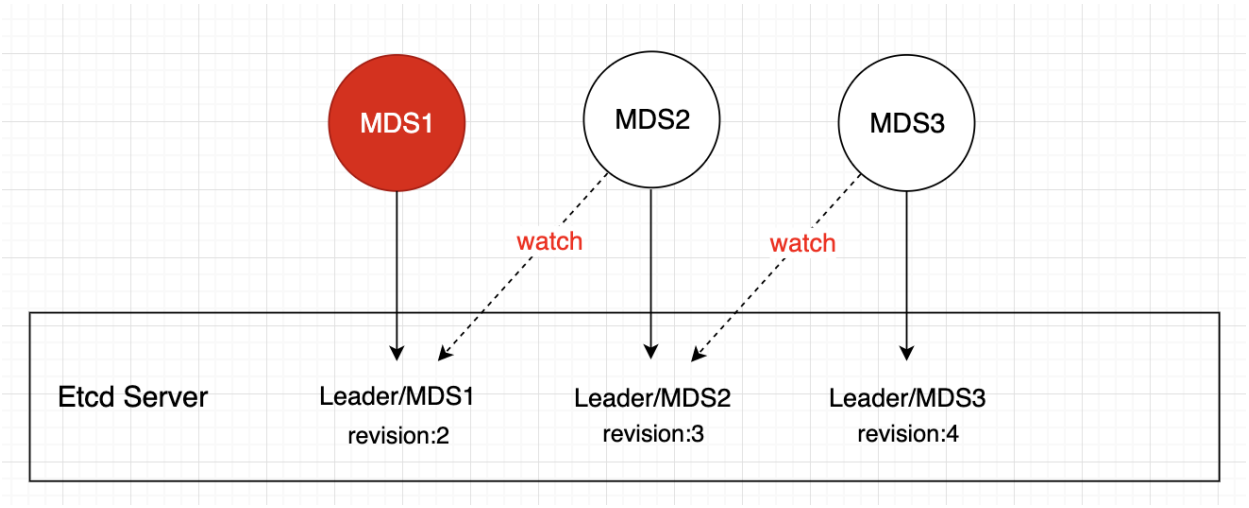
正常情况：

step1: 三个mds向etcdserver写入带有相同前缀的key，etcd会给每个key一个版本号(revision: 是全局递增的)



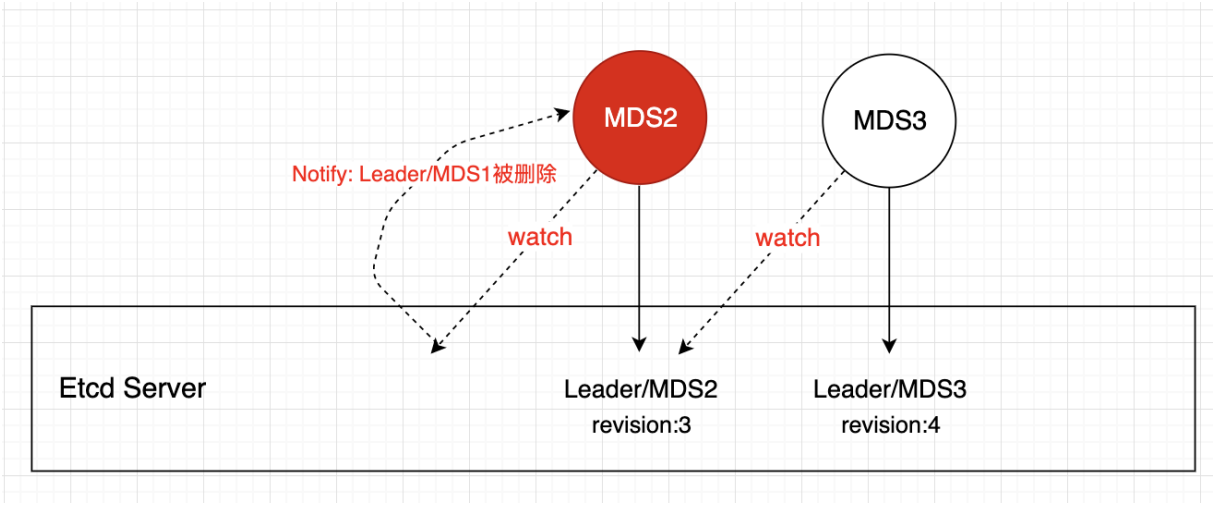
step2:

1. 写入key版本最小的mds当选leader
2. 其余mds对 有相同prefix，且revision小于自身的key中，revision最大的那个 进行watch。例如MDS2获取到有相同前缀Leader的key为{ [Leader/MDS1, revision:2] }, watch该key; MDS3获取到有相同前缀Leader的key为{ [Leader/MDS1, revision:2], [Leader/MDS2, revision:3],}, 其中版本号较大的为 [Leader/MDS2, revision:3], 因此watch Leader/MDS2。



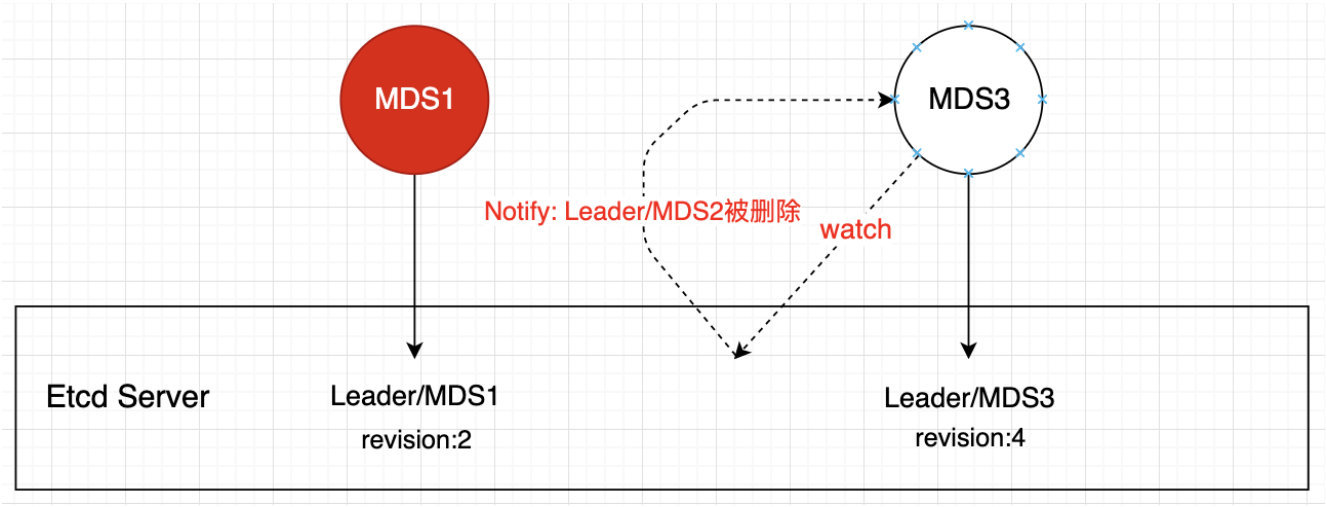
step3:

1. MDS1退出后，MDS2收到MDS1的key被删除的消息，Campagin成功

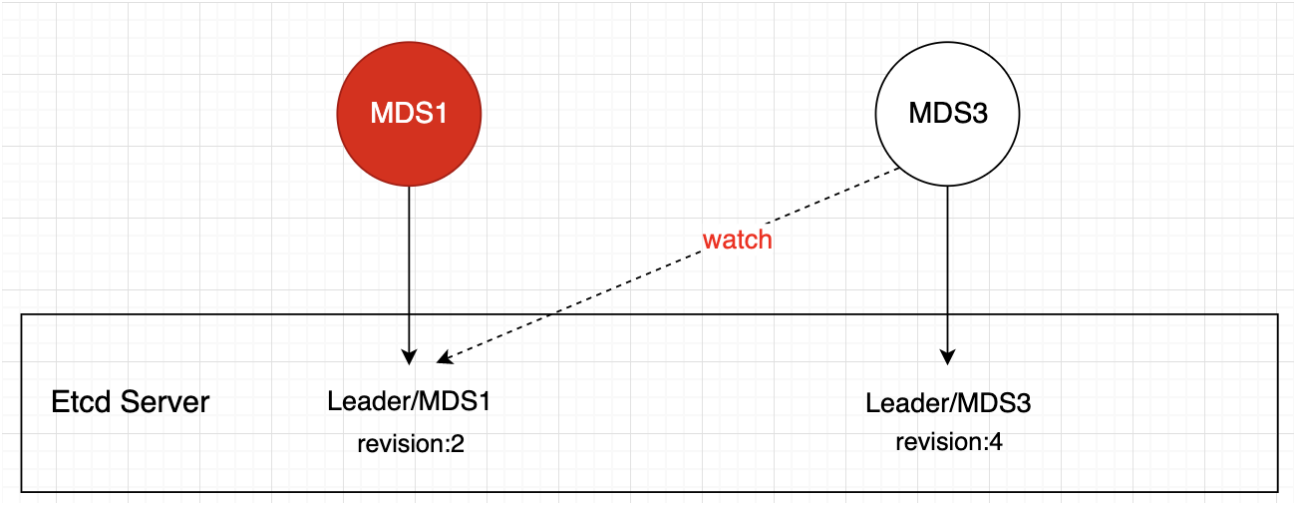


异常情况1：备MDS2中途退出

step1：MDS3收到MDS2的key被删除的消息

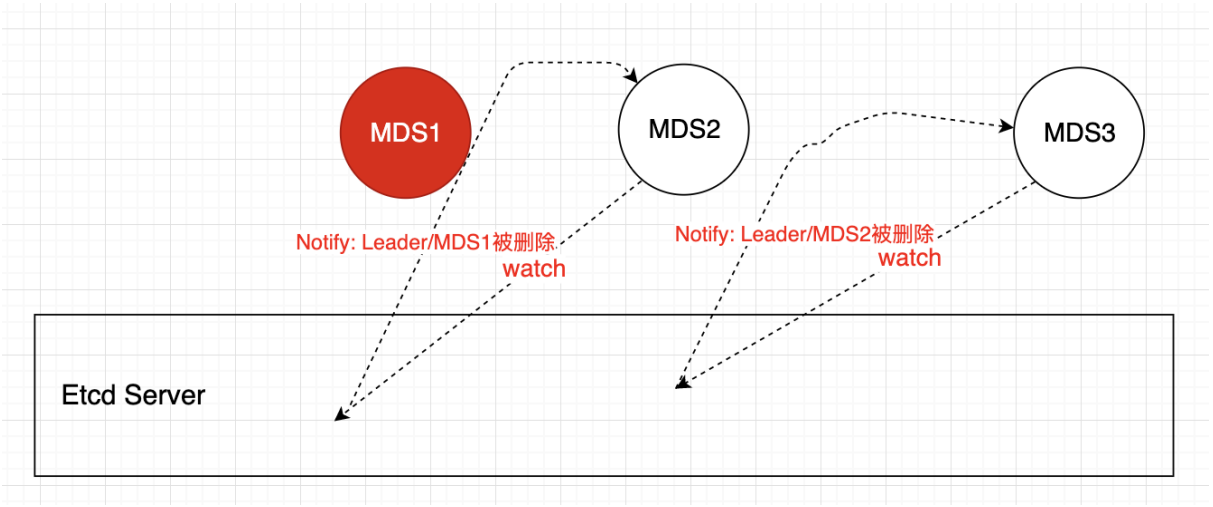


step2:　MDS3重新获取到有相同前缀Leader的key为{ [Leader/MDS1, revision:2]}, 因此watch Leader/MDS1



异常情况2：EtcdLeader重新进行leader选举，且在该过程中，三个MDS和EtcdServer之间的租约全部失效

step1：MDS2收到Leader/MDS1被删除的通知，MDS3收到Leader/MDS2被删除的通知，Campagin都返回成功



这种情况下自身的key已经不在了，三个MDS都不应该成为leader。

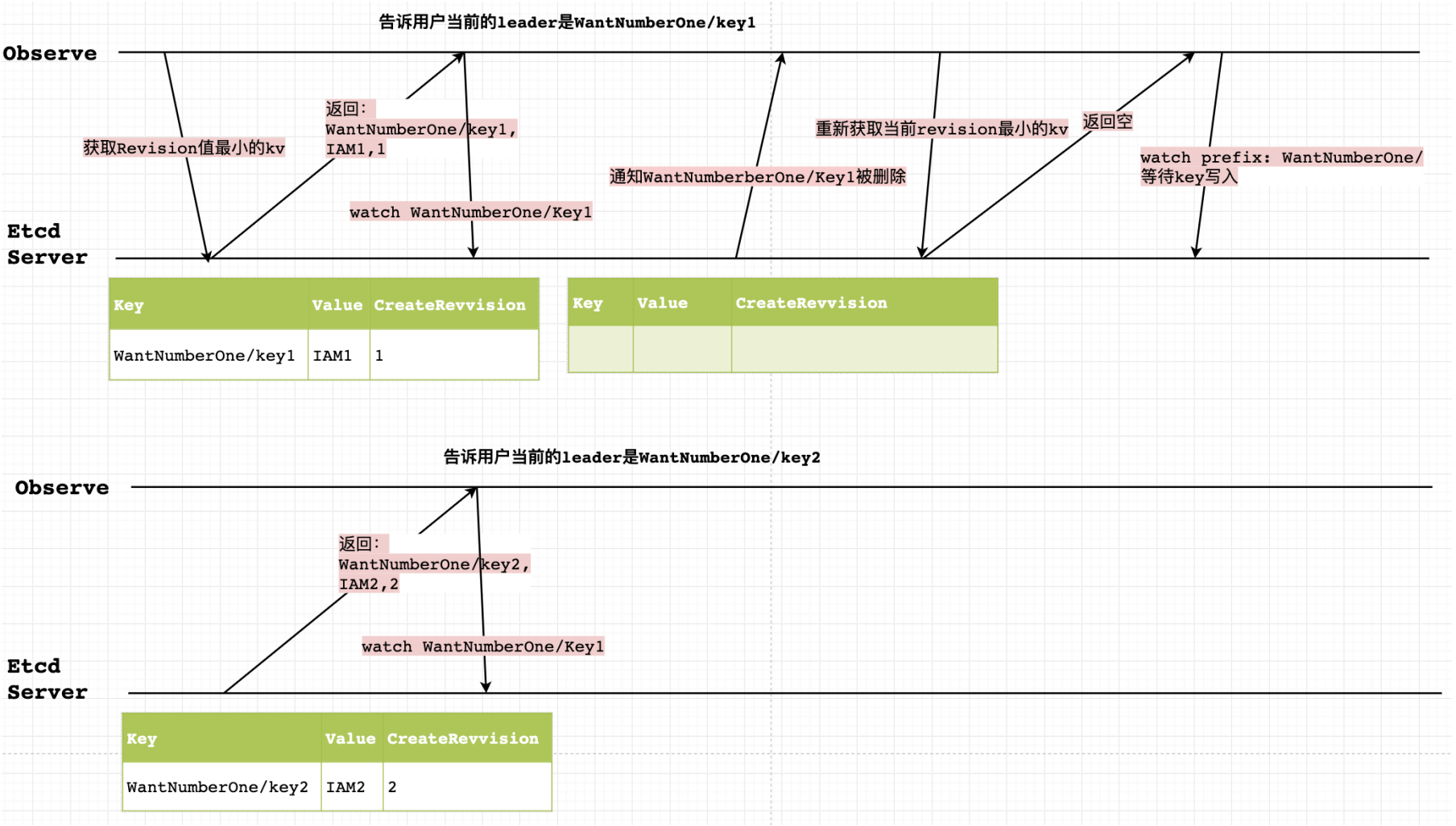
在使用Campagin做选举的时候应该要注意，Campagin返回nil后要再次判断自身的key值是否还存在，如果存在才能认为竞选成功。

3.3 Observe的流程

observe的功能在上面说过，主要用于监听leader的变化。

1. 获取[指定prefix的key，创建版本号最小]kv值， 如果不存在，会一直等待到有指定prefix的key创建为止。
2. 如果存在，监听该key值，如果key被删除，回到1的操作。

过程描述如下：



4. MDS使用election模块的功能进行选主

4.1 Curve中MDS的选举过程

如果对代码部分不感兴趣，请跳到 4.2 图示说明选举流程

MDS使用election模块选有以下三个步骤：

1. 调用Campagin进行选举
2. 如果选举成功，获取一下当前的leaderKey，看是否存在。如果不存在，则继续竞选；如果存在进行下一步
3. 调用Observe观察leader的变化，如果leader有变化，mds退出；重新进行选举

```
// leaderKey
// 1. CampaginRevisionkeyetcdRevisionkeykey
// , etcd leader fail, keyetcdelection timeout
// leaderleaseleaderMDSleader
// keyleaderRevisionkeyCampagin
// 2. ObserveObserveleaderleaderMDSwatchPrefix,
// prefixkey
// 3. key
while (0 != leaderElection->CampaginLeader() ||
        false == leaderElection->LeaderKeyExist()) {
        LOG(INFO) << leaderElectionOp.leaderUniqueName
                  << " campaign for leader agin";
}

leaderElection->StartObserverLeader();
```

StartObserverLeader

```
observer := election.Observe(ctx)
for {
    select {
    case resp, ok := <-observer:
        if !ok {
            fmt.Printf("Observe() channel closed permaturely\n")
            return C.ObserverLeaderInternal
        }
```

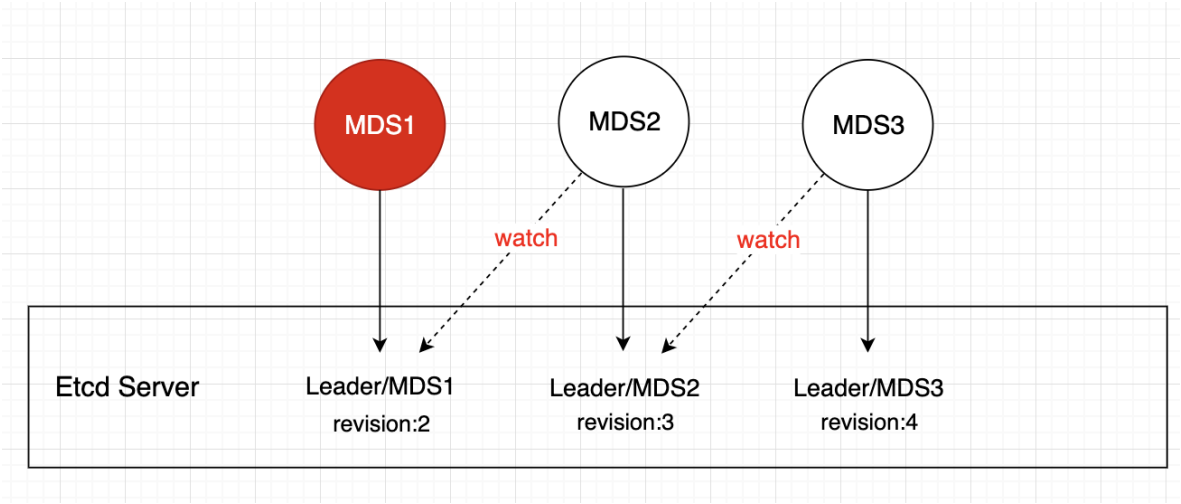
```
    if string(resp.Kvs[0].Value) == goLeaderName {
        continue
    }
    fmt.Printf("Observe() leaderChange, now is: %v, expect: %v\n",
        resp.Kvs[0].Value, goLeaderName)
    return C.ObserverLeaderChange
// observetimeoutcontext, observe timeout
// get
// keycontextetcd
// grpcetcd
    case <-ticker.C:
        // mds
        t := time.Now()
        ctx, cancel := context.WithTimeout(context.Background(),
            time.Duration(int(timeout))*time.Millisecond)
        defer cancel()

        resp, err := globalClient.Get(ctx, election.Key())
        errCode := GetErrCode(EtcdGet, err)
        if errCode != C.OK {
            log.Printf("Observe can not get leader key: %v, startTime:" +
                " %v, spent: %v", election.Key(), t, time.Since(t))
            return C.ObserverLeaderInternal
        } else if len(resp.Kvs) == 0 {
            log.Printf("Observe find leader key%v not exist",
                election.Key())
            return C.ObserverLeaderNotExist
        } else if string(resp.Kvs[0].Key) != election.Key() {
            log.Printf("Observe leaderChange, now is: %v, expect: %v",
                resp.Kvs[0].Value, goLeaderName)
```

```
                return C.ObserverLeaderChange
            }
        }
```

4.2 图示说明选举流程

4.2.1 正常流程

1. MDS1当选leader，MDS2和MDS3处于watch状态

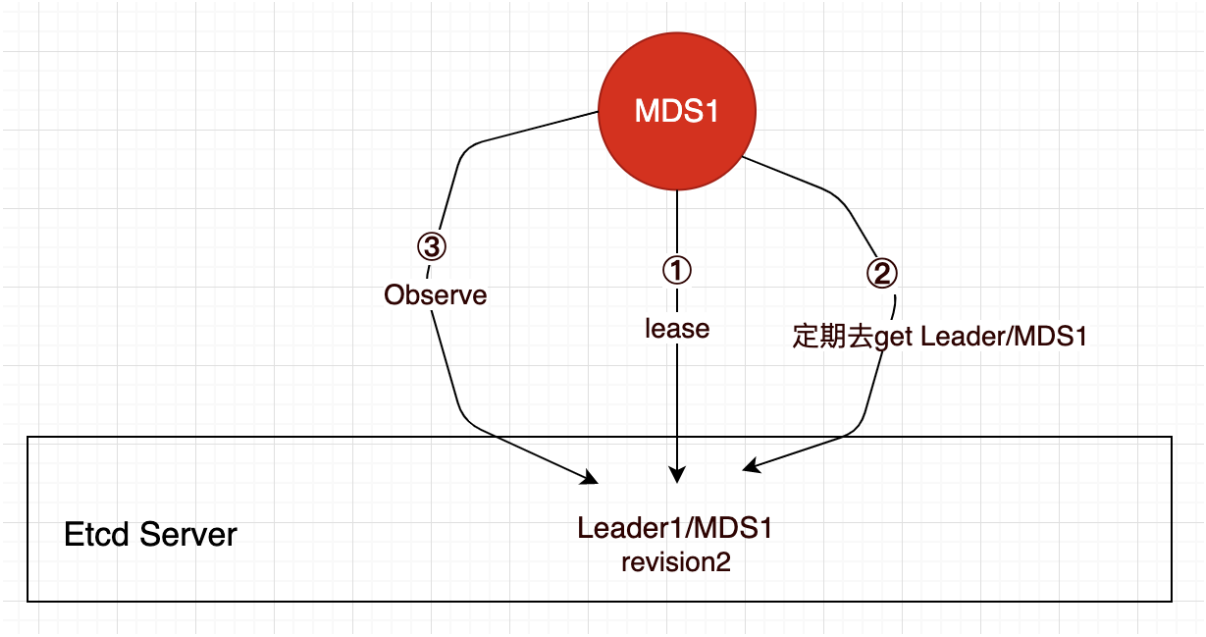


MDS1当选leader之后，与EtcdServer建立的交互如下：

- ①与etcd server维持租约。这里涉及到租约的时间 LeaseTime，租约KeepAlive的时间间隔是1/3的LeaseTime

```
nextKeepAlive := time.Now().Add((time.Duration(karesp.TTL) * time.Second) / 3.0)
```

- ②定期去etcd server中get leader/MDS1，看是否还存在。这里涉及到定期get的时间PeriodicGetTime， 以及get超时的时间GetTimeout
- ③使用Observe监控指定前缀的key的最小版本的变化情况。

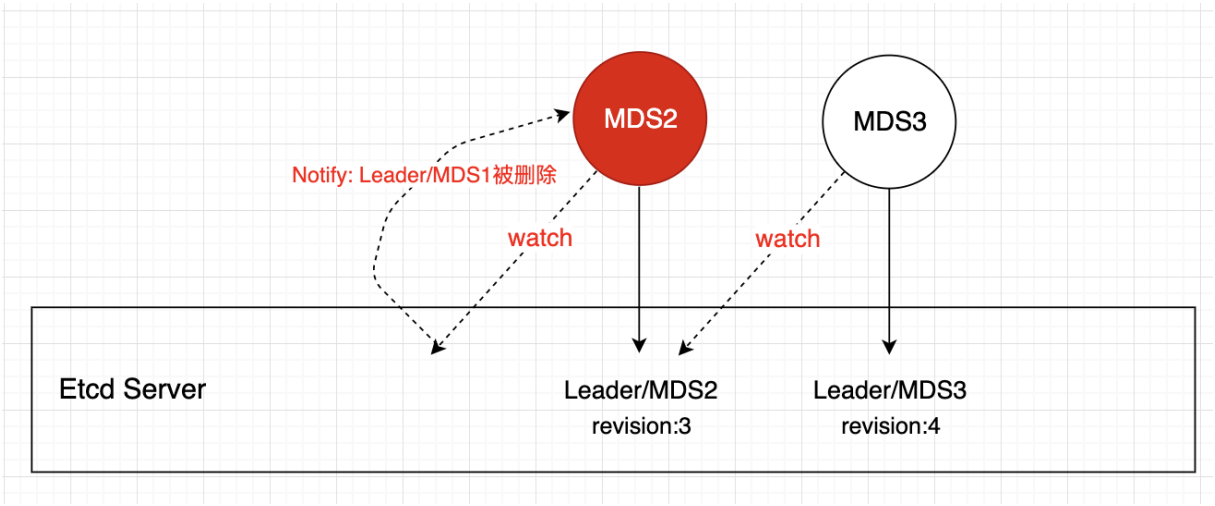


该部分涉及到的参数说明：

参数	说明	当前配置
ElectionTimeout	etcd集群leader选举的超时时间	3s
LeaseTime	mds当选leader之后，与etcd集群维持租约的过期时间 1. 租约的keepalive间隔为LeaseTime/3 2. etcd server端限制LeaseTime >= 1.5 * ElectionTimeout	10s
PeriodicGetTime	mds当选leader之后，去etcd集群get Leader/MDS1的时间间隔	2s
GetTimeout	get Leader/MDS1失败的时间间隔	10s
ElectionTime	etcd集群leader失效，到重新选举出leader的耗时 ElectionTime > ElectionTimeout	

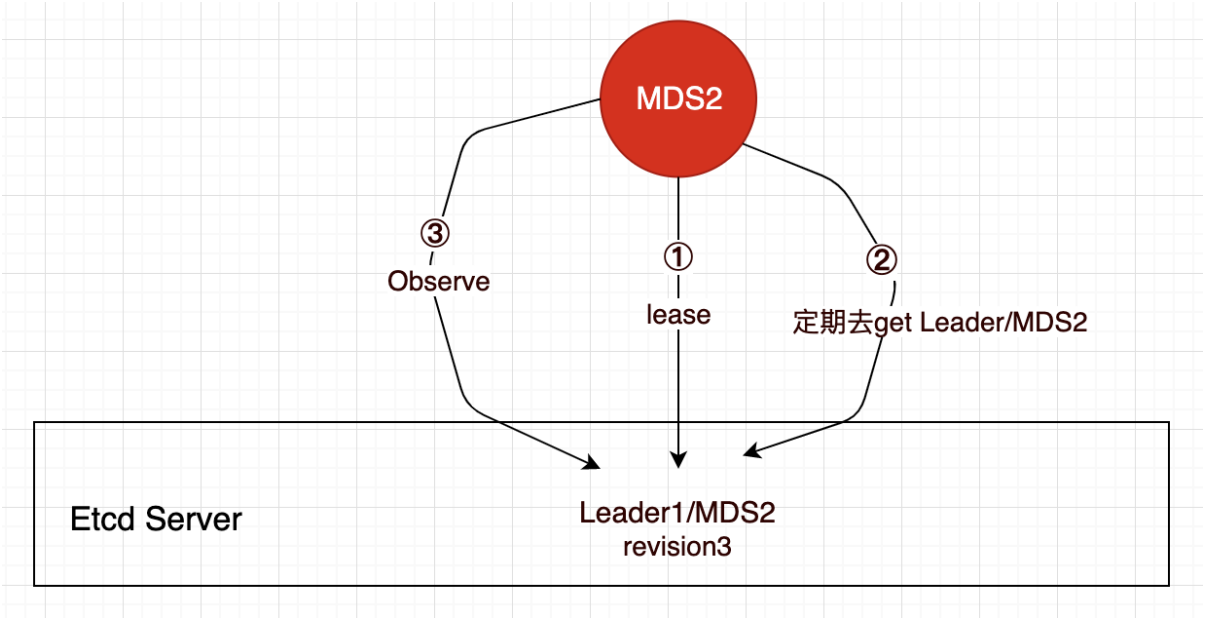
4.2.2 异常情况1：MDS1退出，可以正常处理

1. MDS2收到leader/MDS1被删除的消息，Campaign成功，成为leader



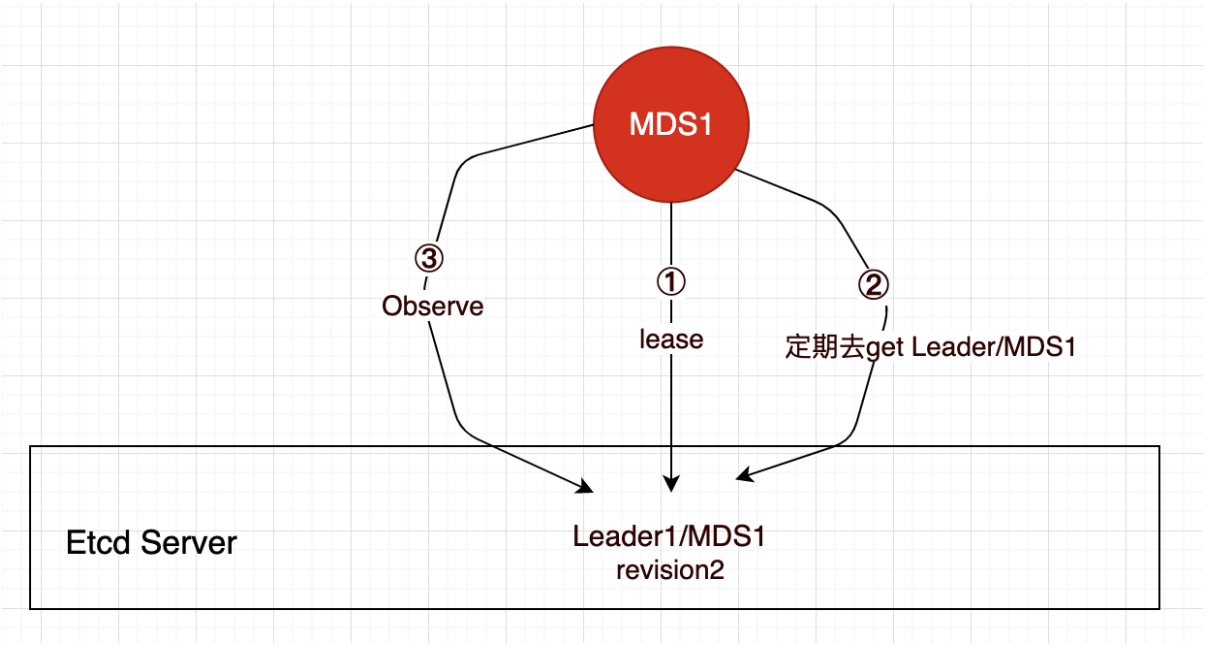
2. mds2当选leader之后，同样与etcd server有三类交互：

- ①与etcd server维持租约。
- ②定期去etcd server中get leader/MDS2，看是否还存在。
- ③使用Observe监控指定前缀的key的最小版本的变化情况。



4.2.3 异常情况2：Etcd集群的leader发生重新选举，MDS1未受影响，可以正常处理

1. etcd集群异常，重新选举leader
2. 但LeaseTime > ElectionTime 且 GetTimeout > ElectionTime
   这种情况是常态，大概率情况ElectionTime略大于ElectionTimeout，LeaseTime >= 1.5*ElectionTimeout > ElectionTime
3. 这种情况下etcd集群恢复正常之后，MDS与etcd集群的lease维持正常；定期get Leader/MDS1正常；Observe正常



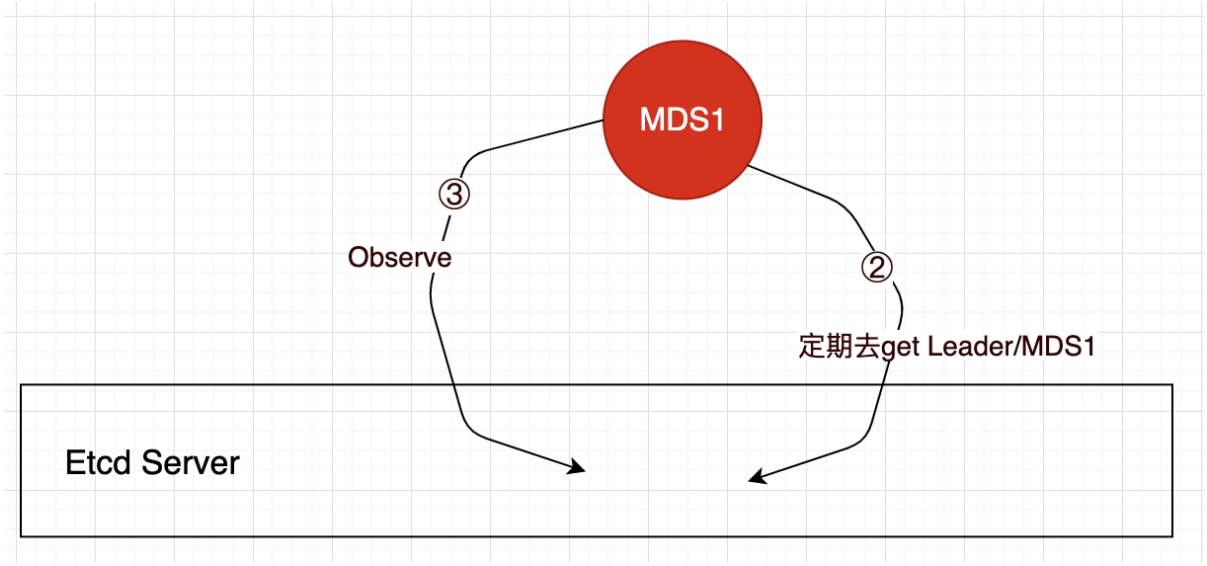
4.2.4 异常情况3：Etcd的leader发生重新选举，MDS1受到影响退出，不一定可以正常处理。

1. etcd集群异常，重新选举leader
2. 但leaseTIme < ElectionTime 或者 GetTimeout < ElectionTime

4.2.4.1 LeaseTIme < ElectionTime的情况

这种情况发生概率较小，etcd集群选出新leader耗时较长

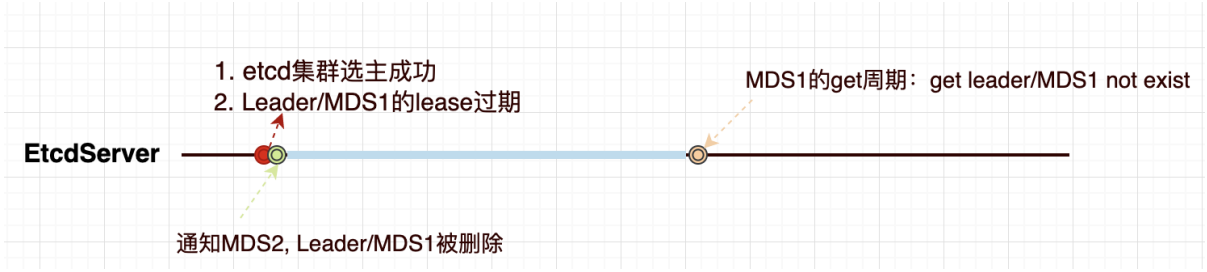
1. 当etcd集群恢复正常的情况下，MDS1的lease已过期，etcd server把MDS1注册过来的Key删掉



2. 此时会有两件事情发生，顺序不定：

- 一是MDS1定期去get Leader/MDS1失败后MDS退出 ［事件1］
- 二是MDS2收到Leader/MDS1被删除，MDS2开始提供服务 ［事件2］

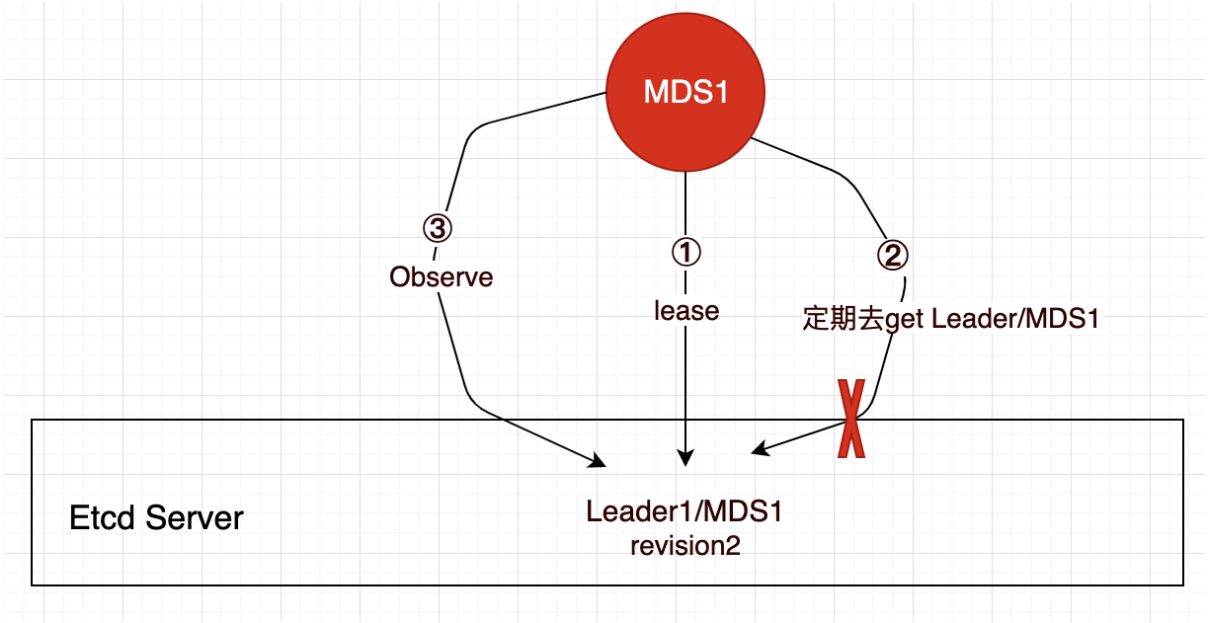
如果事件1先发生，那么就是MDS1退出后，MDS2再当选为leader，
如果事件2先发生，那么就是MDS2当选为leader时，MDS1还在提供服务，出现双主，这是有问题的。
双主出现的时间有多久呢？如下图：双主的时间为PeriodicGetTime



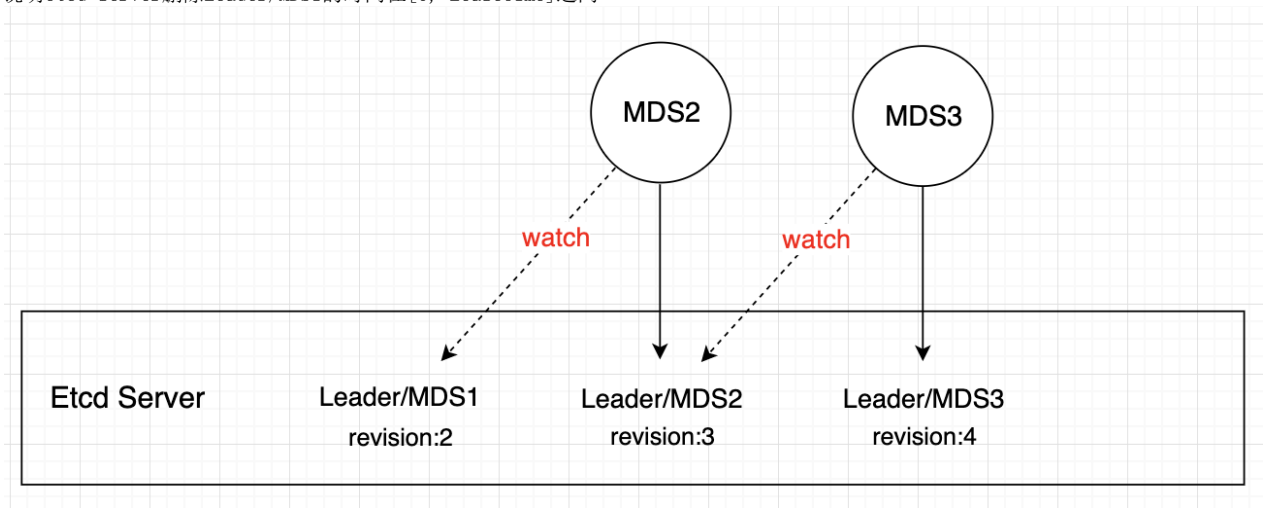
① 红色的点是etcd集群选主成功的时间点，选主成功之后MDS1的lease过期，Leader/MDS1被删除
② 绿色的点是MDS2收到Leader/MDS1删除消息的时间点。此时MDS2启动并提供服务
③ 黄色的点是最坏情况，MDS1在绿色点和红色点之间成功get到leader/MDS1，在下一个周期get失败
这种情况下出现双主的最长时间为PeriodicGetTime(蓝色直线段)， 短暂时间内的双主情况是可以接受的。

4.2.4.2 GetTimeout < ElectionTime

1. 当etcd集群恢复正常的情况下，MDS1的lease没有过期，但是get Leader/MDS1超时。



2. MDS1会退出，但lease的最短过期时间为0，最长过期时间为LeaseTIme
   说明etcd server删除Leader/MDS1的时间在[0, LeaseTime]之间

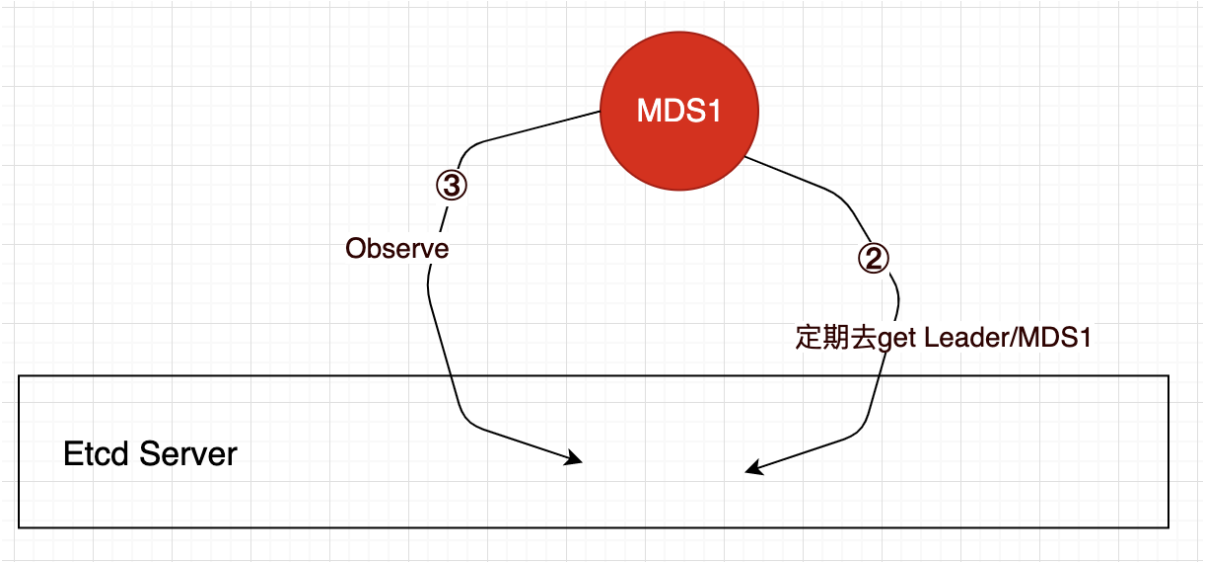


这种情况会导致[0, LeaseTime]时间内没有MDS提供服务。

当前配置下，LeaseTime = GetTimeout，这种情况发生的概率极低，Lease在etcd新leader当选后没有失效，get也不应该超时

4.2.4.3 MDS1、MDS2、MDS3的租约全部过期

1. 当etcd集群恢复正常的情况下，MDS1的lease已过期，etcd server把MDS1注册过来的Key删掉

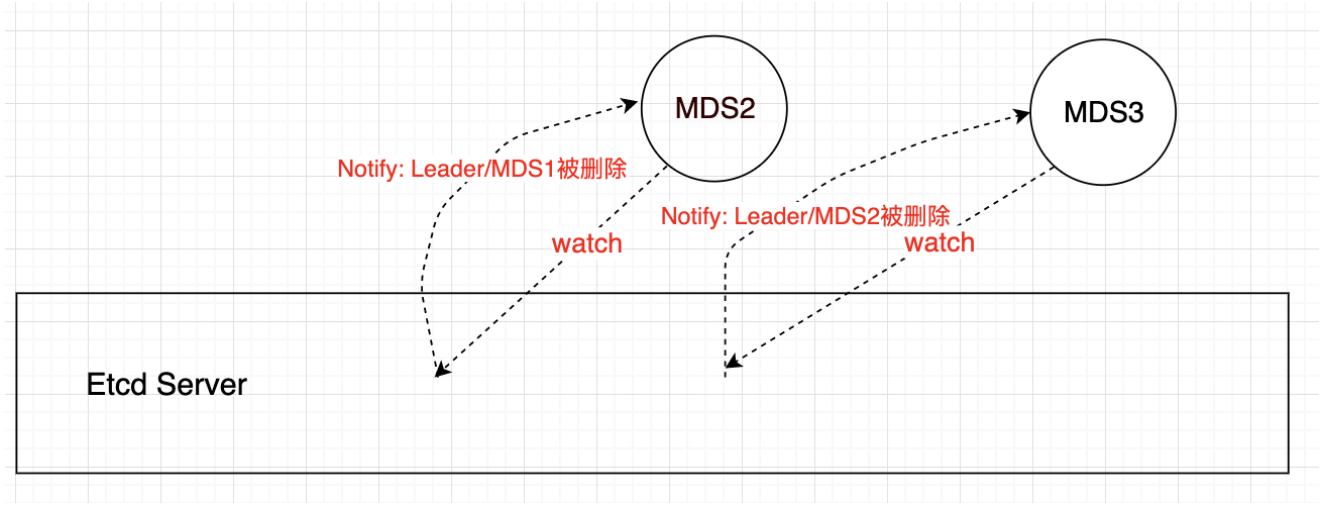


2. MDS1获取Leader/MDS1失败，退出

3. MDS2和MDS3事件如下：

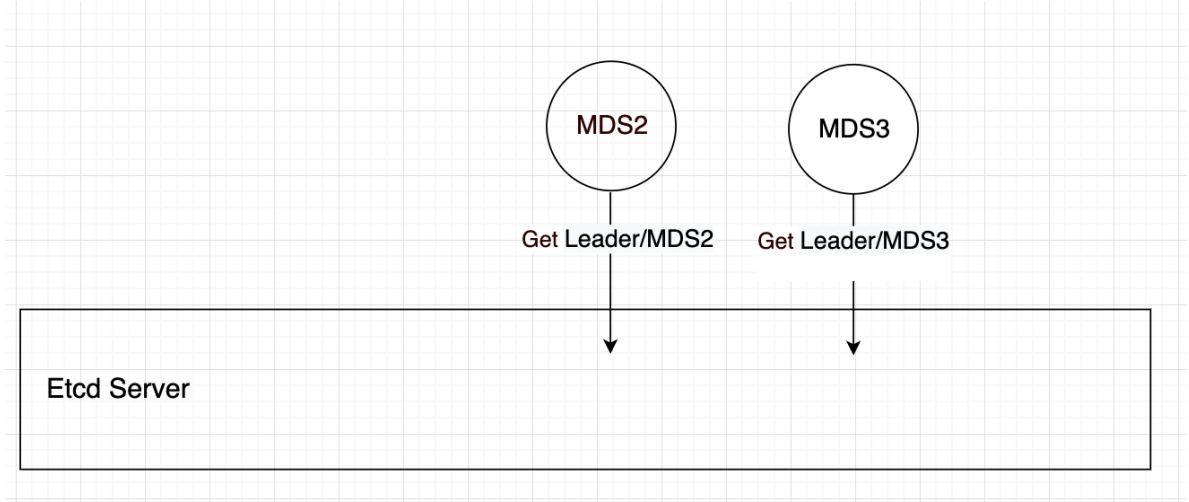
事件一：

- MDS2收到Leader/MDS1退出的消息，Campaign返回成功
- MDS3收到Leader/MDS2退出的消息，Campagin返回成功



事件二：

- MDS2 Campagin成功后再次获取竞选时使用的key值Leader/MDS2，获取失败，退出
- MDS2 Campagin成功后再次获取竞选时使用的key值Leader/MDS3，获取失败，退出



这种情况可以被正确的处理，三个mds都退出，依赖daemon重新拉起，发起新一轮的MDS leader竞选。

4.2.4.4 总结

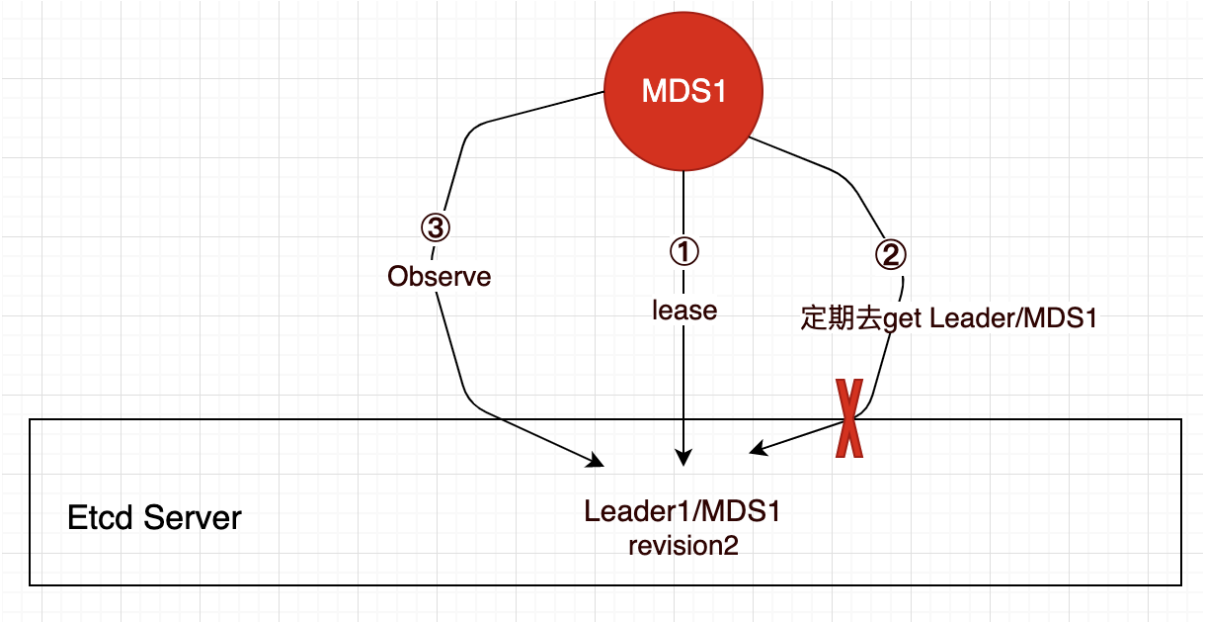
etcd集群发生leader选举的情况，会造成MDS出现双主或者一段时间内没有MDS提供服务的情况。

措施：

1. 使得PeriodicGetTime尽量的小，减少双主可能出现的时间
2. GetTimeout和LeaderTIme要大于etcd集群leader election timeout的时间，尽量减少etcd集群leader选举过程中的get超时以及lease失效。这两种情况不可能完全避免，虽然etcd集群在election timeout时间后开始进行选举，正常情况下很快会选举成功，但异常情况下，成功选举出leader所需要的时间是不确定的

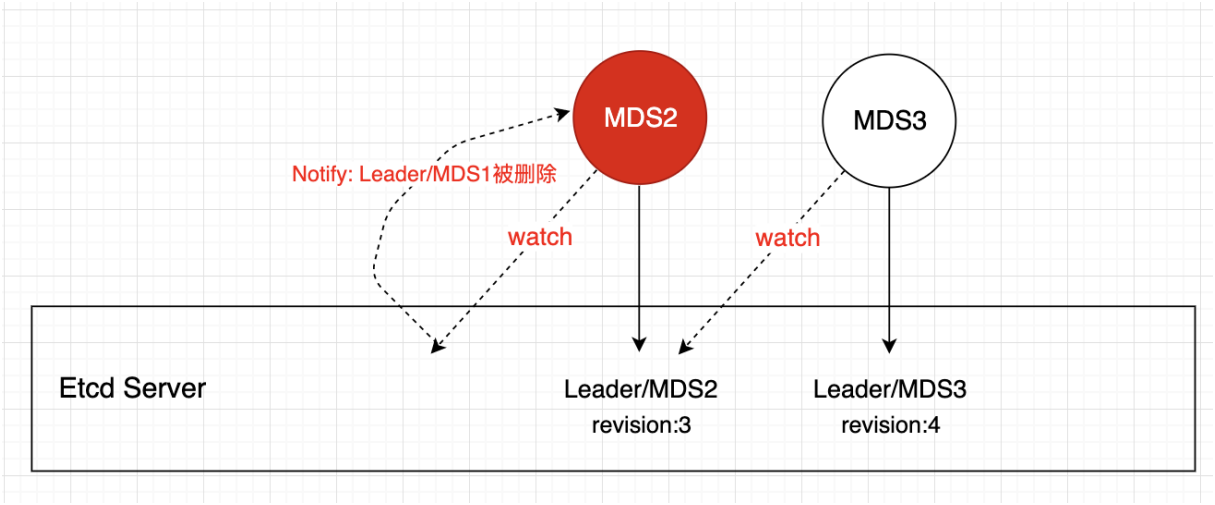
4.2.5 异常情况四： Etcd集群与MDS1(当前leader)出现网络分区

1. etcd集群与MDS1发生网络分区，以下事件会发生：
   [事件一] get Leader/MDS1会超时退出
   [事件二] MDS1与etcd集群之间的lease会过期，Leader1/MDS1会被etcd server删除，触发MDS2当选leader



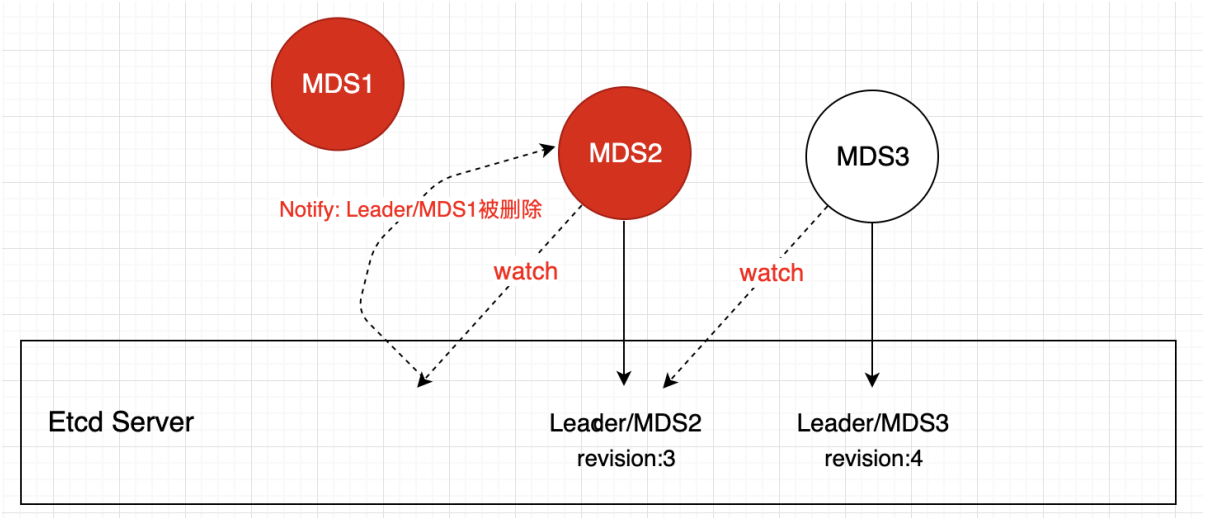
4.2.5.1 事件一先发生

1. get Leader/MDS1会超时退出
2. lease过期之后，Leader1/MDS1会被etcd server删除，触发MDS2当选leader
   这种情况可以正常处理



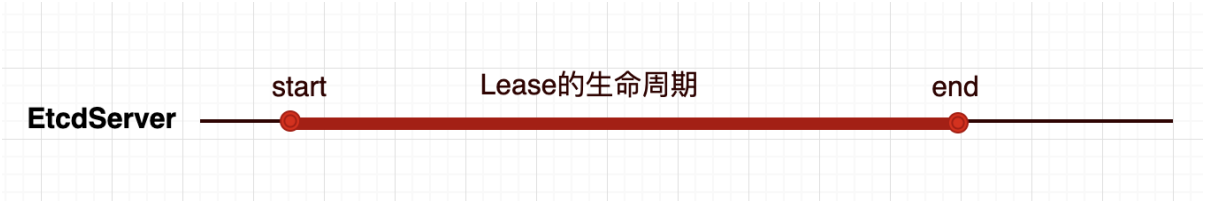
4.2.5.2 事件二先发生

1. lease过期之后，Leader1/MDS1会被etcd server删除，触发MDS2当选leader，此时MDS1和MDS2将同时提供服务，集群中出现双主。
2. get Leader/MDS1超时退出，双主的情况结束

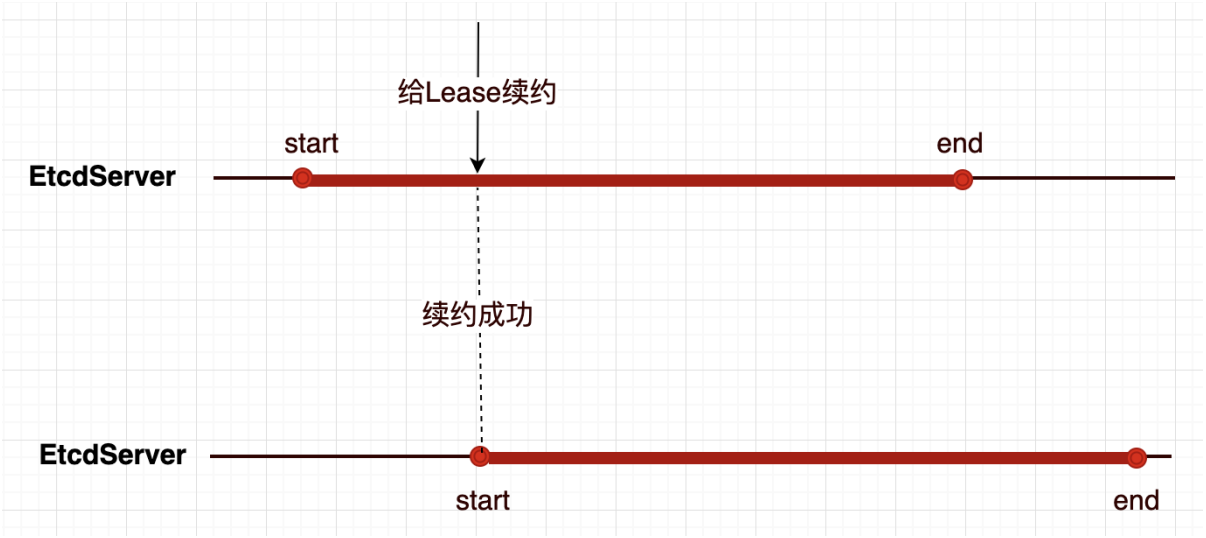


出现双主的最大时间为多久呢？

下图说明了lease的生命周期

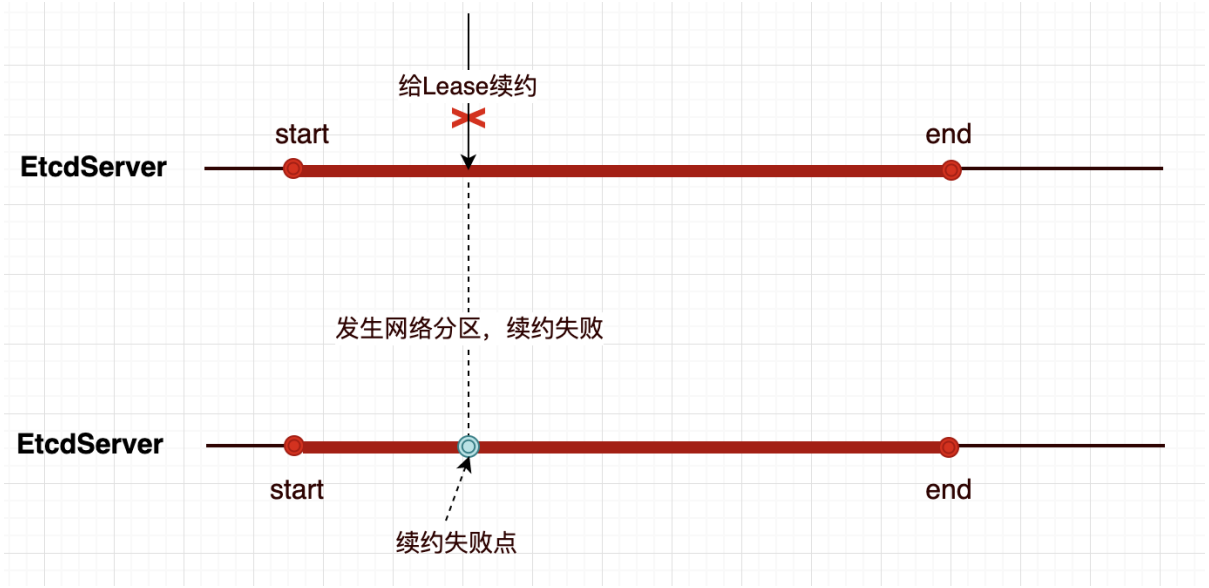


正常情况下MDS会周期性的为Lease续约，如果续约成功，Lease的expired点会后移



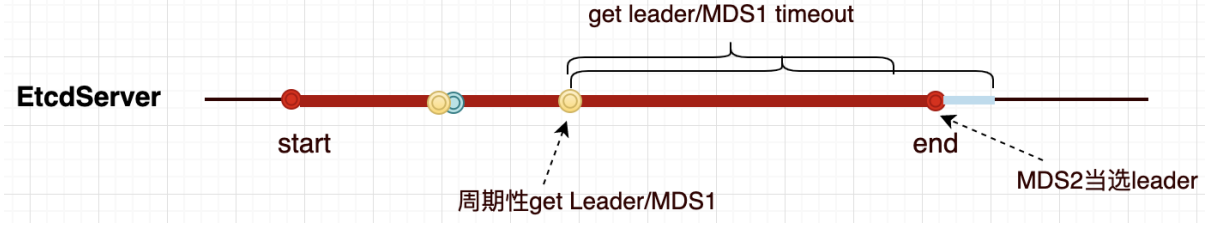
异常情况下，MDS1与etcd集群发生网络分区

1. MDS1给Lease续约，但因为网络分区续约失败



2. 在网络分区前MDS1周期性get leader/MDS1成功

3. 下一个周期get leader/MDS1失败，要timeout以后才能返回。出现双主的时间与GetTimeout，PeriodicGetTime，Lease的续约间隔有关



按照当前的配置：出现双主的时间为5s，这个值目前还是可以接受的。

4.2.6 异常情况4：Etcd集群的follower节点异常

只要etcd集群不发生重新选举，且leader存在的情况下，对外的服务没有问题，不会对mds产生影响。

4.2.7 各情况汇总

初始状态：MDS1为当前leader，MDS2/MDS3为备节点

当前配置：

ElectionTimeout=3s

ElectionTimeout=10s

GetTimeout=10s

PeriodicGetTime=GetTimeout/5=2s

MDS的启动时间：500ms左右

I0823 16:46:32.413305 22833 leader_election.cpp:25] 10.182.26.33:6666 campaign leader success

I0823 16:46:32.942611 22833 server.cpp:1039] Server[curve::mds::heartbeat::HeartbeatServiceImpl+curve::mds::NameSpaceService+curve::mds::topology::TopologyServiceImpl] is serving on port=6666.

场景	MDS是否需要切换	影响	发生概率
主MDS1正常退出(kill进程)	是	MDS2当选leader 可以迅速切换	正常
主MDS1异常退出(机器断电)	是	MDS2当选leader 需要等到MDS1的lease过期	小
备MDS2退出	否	MDS1任为leader	正常
Etcd的follower节点挂掉	否	MDS1仍为leader	正常
Etcd的leader节点挂掉	三个节点的lease全部过期，etcd集群leader election时间过长 是	1. MDS1最终退出MDS2/MDS3进行竞选 2. 过程中出现双主的时间在[0,2s]	小
	三个节点的lease均未过期 否	MDS1仍为leader	大
Etcd整个集群不可用	是	MDS1自动退出 MDS2/MDS3不能当选leader	小
Etcd集群与MDS1发生网络分区，与MDS2，MDS3网络正常	是	1. MDS1最终退出 2. 过程中出现双主的时间在[0, 5.3s]	小