GetCandidateReplica
// RegroupReplicasGTID will choose a candidate replica of a given instance, and take its siblings using GTID
func RegroupReplicasGTID(
    masterKey *InstanceKey, // 实参传进来的是 挂掉的旧主库
    returnReplicaEvenOnFailureToRegroup bool, // 实参传进来的是 true
    startReplicationOnCandidate bool, // 实参传进来的是 false
    onCandidateReplicaChosen func(*Instance), // 实参传进来的是 nil
    postponedFunctionsContainer *PostponedFunctionsContainer,
    postponeAllMatchOperations func(*Instance, bool) bool, // 实参传进来的是 promotedReplicaIsIdeal 函数
)
RegroupReplicasGTID will choose a candidate replica of a given instance, and take its siblings using GTID

英文简简单单一句话, 中文不知道咋翻译.. 我理解就是 RegroupReplicasGTID 会从目标实例(即 DeadMaster)的从库中选出一个 candidate 出来, 然后提升他为新主库，并接管所有的从库

要理解 RegroupReplicasGTID ，还是要先看它调用的 GetCandidateReplica
// GetCandidateReplica chooses the best replica to promote given a (possibly dead) master
func GetCandidateReplica(masterKey *InstanceKey, forRematchPurposes bool) (*Instance, [](*Instance), [](*Instance), [](*Instance), [](*Instance), error) {

    // masterKey 实参传进来的是 挂掉的旧主库. InstanceKey 结构体里只有 Hostname 和 Port
    // forRematchPurposes 实参传进来是 true

    // 声明变量, 这是一个指针
    var candidateReplica *Instance
    aheadReplicas := [](*Instance){} // 字面量声明, 所以 aheadReplicas != nil
    equalReplicas := [](*Instance){}
    laterReplicas := [](*Instance){}
    cannotReplicateReplicas := [](*Instance){}

    dataCenterHint := ""
    // 这里实际是根据 Hostname 和 Port 读取 backend db database_instance 表, 实例化了一个 instance, 使用 readInstanceRow 填充了各种属性, 如 is_candidate, promotion_rule 等等
    if master, _, _ := ReadInstance(masterKey); master != nil {
        dataCenterHint = master.DataCenter

```
    }
    // 返回给定主站的副本列表，用于候选选择。
    // 就是把 masterKey 的所有从库读出来了, 返回一个[](*Instance)
    replicas, err := getReplicasForSorting(masterKey, false)
    if err != nil {
        // 如果有 err, 这里直接 return. 注意此时 candidateReplica 是等于 nil 的
        return candidateReplica, aheadReplicas, equalReplicas, laterReplicas, cannotReplicateReplicas, err
    }

    // type StopReplicationMethod string

    // const (
    //      NoStopReplication                StopReplicationMethod   = "NoStopReplication"
    //   StopReplicationNormal                                       = "StopReplicationNormal"
    //   StopReplicationNice                                         = "StopReplicationNice"
    // )
    stopReplicationMethod := NoStopReplication
    // forRematchPurposes 实参传进来是 true
    if forRematchPurposes {
        stopReplicationMethod = StopReplicationNice // 所以 stopReplicationMethod 是 StopReplicationNice
    }
    // 传入了所有的从库, StopReplicationNice, 和 主库的数据中心
    // 返回根据 exec coordinates 排序的从库列表
    replicas = sortedReplicasDataCenterHint(replicas, stopReplicationMethod, dataCenterHint)
    if len(replicas) == 0 {
        return candidateReplica, aheadReplicas, equalReplicas, laterReplicas, cannotReplicateReplicas, fmt.Errorf("No replicas found for %+v", *masterKey)
    }
    candidateReplica, aheadReplicas, equalReplicas, laterReplicas, cannotReplicateReplicas, err = chooseCandidateReplica(replicas)
    if err != nil {
        return candidateReplica, aheadReplicas, equalReplicas, laterReplicas, cannotReplicateReplicas, err
    }
    if candidateReplica != nil {
        mostUpToDateReplica := replicas[0]
```

```
        if
candidateReplica.ExecBinlogCoordinates.SmallerThan(&mostUpToDateReplic
a.ExecBinlogCoordinates) {
            log.Warningf("GetCandidateReplica: chosen replica: %+v is
behind most-up-to-date replica: %+v", candidateReplica.Key,
mostUpToDateReplica.Key)
        }
    }
    log.Debugf("GetCandidateReplica: candidate: %+v, ahead: %d,
equal: %d, late: %d, break: %d", candidateReplica.Key,
len(aheadReplicas), len(equalReplicas), len(laterReplicas),
len(cannotReplicateReplicas))
    return candidateReplica, aheadReplicas, equalReplicas,
laterReplicas, cannotReplicateReplicas, nil
}
```

GetCandidateReplica 首先根据 masterKey (只包含 Hostname 和 Port)查询 Backend DB 的 database_instance 表生成了一个 master"对象" 然后从 Backend DB 中查询出 master 的所有的从库，返回一个包含所有从库*Instance 的切片 replicas

注意

如果在"获取"从库的过程中出现 error ，则 GetCandidateReplica 会终止直接 return. 而此时 candidateReplica 是等于 nil 的

接着调用 sortedReplicasDataCenterHint 对 replicas 进行排序. 接下来先展开说一下这个函数

sortedReplicasDataCenterHint

```
// sortedReplicas returns the list of replicas of some master, sorted
by exec coordinates
// (most up-to-date replica first).
// This function assumes given `replicas` argument is indeed a list of
instances all replicating
// from the same master (the result of `getReplicasForSorting()` is
appropriate)
func sortedReplicasDataCenterHint(replicas [](*Instance),
stopReplicationMethod StopReplicationMethod, dataCenterHint string)
[](*Instance) {
    if len(replicas) <= 1 {  // 如果只有一个从库, 直接返回
        return replicas
    }
    // InstanceBulkOperationsWaitTimeoutSeconds 默认 10s
    // 先 StopReplicationNicely 超时 10s, 如果超时了也只是记了日志. 然
```

后 StopReplication

// 然后 sortInstancesDataCenterHint. 这要看 NewInstancesSorterByExec 的 Less 方法如何实现. 简单说就是 ExecBinlogCoordinates 大的放前面, 如果 ExecBinlogCoordinates 一样, Datacenter 和 DeadMaster 一样的放前面

```
    replicas = StopReplicas(replicas, stopReplicationMethod, time.Duration(config.Config.InstanceBulkOperationsWaitTimeoutSeconds)*time.Second)
    replicas = RemoveNilInstances(replicas)

    sortInstancesDataCenterHint(replicas, dataCenterHint)
    for _, replica := range replicas {
        log.Debugf("- sorted replica: %+v %+v", replica.Key, replica.ExecBinlogCoordinates)
    }

    return replicas
}
```

从注释可以看出 sortedReplicas 会返回一个按 exec coordinates 排序的从库列表 (most up-to-date first) sortedReplicasDataCenterHint 先调用 StopReplicas , StopReplicas 做了几件事:

对于本例，stopReplicationMethod 是 StopReplicationNice

并行的在所有从库执行 StopReplicationNicely. StopReplicationNicely 会先 stop slave io_thread, start slave sql_thread, 然后对所有非延迟从库 WaitForSQLThreadUpToDate, 最多等待 InstanceBulkOperationsWaitTimeoutSeconds 秒(也就是默认 10s)

如果超过 InstanceBulkOperationsWaitTimeoutSeconds 秒, SQL_THREAD 还是没有应用完所有日志, 也不等了.

等待超时不会引发异常

StopReplicationNicely 执行完成后, 执行 StopReplication. 实际就是执行 stop slave

对比 MHA

MHA 其实会在 Dead Master Shutdown Phase 停所有从库 io_thread

MasterFailover.pm

```
sub do_master_failover {
    ...
    $log->info("* Phase 2: Dead Master Shutdown Phase..\n");
    $log->info();
```

```
    force_shutdown($dead_master);
    $log->info("* Phase 2: Dead Master Shutdown Phase completed.\n");
    ...
}


sub force_shutdown($) {
  ...

  my $slave_io_stopper = new Parallel::ForkManager( $#alive_slaves +
1 );
  my $stop_io_failed   = 0;
  $slave_io_stopper->run_on_start(
    sub {
      my ( $pid, $target ) = @_;
    }
  );
  $slave_io_stopper->run_on_finish(
    sub {
      my ( $pid, $exit_code, $target ) = @_;
      return if ( $target->{ignore_fail} );
      $stop_io_failed = 1 if ($exit_code);
    }
  );

  foreach my $target (@alive_slaves) {
    $slave_io_stopper->start($target) and next;
    eval {
      $SIG{INT} = $SIG{HUP} = $SIG{QUIT} = $SIG{TERM} = "DEFAULT";
      my $rc = $target->stop_io_thread();
      $slave_io_stopper->finish($rc);
    };
    if ($@) {
      $log->error($@);
      undef $@;
      $slave_io_stopper->finish(1);
    }
    $slave_io_stopper->finish(0);
  }
```

这是很合理的, 只要开始 Failover 了, 就说明 MHA 认为主库已经挂了，那么停 io_thread 再根据 Master_Log_File 和 Read_Master_Log_Pos 选 latest slave是 没问题的

ServerManager.pm

```
sub identify_latest_slaves($$) {
  my $self        = shift;
  my $find_oldest = shift;
  $find_oldest = 0 unless ($find_oldest);
  my $log    = $self->{logger};
  my @slaves = $self->get_alive_slaves();
  my @latest = ();
  foreach (@slaves) {
    my $a = $latest[0]{Master_Log_File};
    my $b = $latest[0]{Read_Master_Log_Pos};
    if (
      !$find_oldest
      && (
           ( !$a && !defined($b) )
        || ( $_->{Master_Log_File} gt $latest[0]{Master_Log_File} )
        || ( ( $_->{Master_Log_File} ge $latest[0]{Master_Log_File} )
          &&                $_->{Read_Master_Log_Pos}                >
$latest[0]{Read_Master_Log_Pos} )
      )
      )
    {
      @latest = ();
      push( @latest, $_ );
    }
    elsif (
      $find_oldest
      && (
           ( !$a && !defined($b) )
        || ( $_->{Master_Log_File} lt $latest[0]{Master_Log_File} )
        || ( ( $_->{Master_Log_File} le $latest[0]{Master_Log_File} )
          &&                $_->{Read_Master_Log_Pos}                <
$latest[0]{Read_Master_Log_Pos} )
      )
      )
    {
      @latest = ();
      push( @latest, $_ );
    }
    elsif ( ( $_->{Master_Log_File} eq $latest[0]{Master_Log_File} )
      &&           (           $_->{Read_Master_Log_Pos}           ==
$latest[0]{Read_Master_Log_Pos} ) )
    {
      push( @latest, $_ );
    }
```

```
  }
  foreach (@latest) {
    $_->{latest} = 1 if ( !$find_oldest );
    $_->{oldest} = 1 if ($find_oldest);
  }
  $log->info(
    sprintf(
      "The %s binary log file/position on all slaves is" . " %s:%d\n",
      $find_oldest ? "oldest" : "latest", $latest[0]{Master_Log_File},
      $latest[0]{Read_Master_Log_Pos}
    )
  );
  if ( $latest[0]{Retrieved_Gtid_Set} ) {
    $log->info(
      sprintf(        "Retrieved        Gtid        Set:        %s",
$latest[0]{Retrieved_Gtid_Set} ) );
  }
  if ($find_oldest) {
    $self->set_oldest_slaves( \@latest );
  }
  else {
    $self->set_latest_slaves( \@latest );
  }
}
```

orchestrator 是根据 ExecBinlogCoordinates 比较出 latest slave

ExecBinlogCoordinates 是应用 binlog 坐标的意思 对应 show slave status 中的

Relay_Master_Log_File
Exec_Master_Log_Pos 表示 sql_thread 已经应用了主库哪个 binlog 哪个位置的日志.
然而若想不丢数据, 则应该根据 ReadBinlogCoordinates 比较出 latest slave, 也就是 MHA 的实现方式, 即

Master_Log_File
Read_Master_Log_Pos 使用 ExecBinlogCoordinates 选 Latest Slave 是可能丢数据的(即使开了半同步), 可以通过 tc 命令轻松复现 orc 这种选 latest slave 导致的丢数据问题.
具体模拟方法见 issue: https://github.com/openark/orchestrator/issues/1312 我修改了 NewInstancesSorterByExec 的 Less 的方法, 改为使用 ReadBinlogCoordinates 选择 latest slave, 在公司的分支解决了上述问题.

以我对 Orchestrator 的了解， Orchestrator 目标是追求可用性优先， 而非数据完整性. 很多公司也使用了 Orchestrator， 我感觉未必知道有这个问题， 或者说，别问，问就是"我们追求可用性”.

现在的问题是，即便开了半同步，也可能丢数据.

然而矛盾的点是，线上主从的复制延迟是大家都要监控和管理的，不会长期处于高延迟状态，起码我经历的公司都是这样，99.9%的集群主从延迟在 1s 内. 个别集群在高峰期会升高一点， 但很快又会下降； 又或者这些集群本身就是 AP 型业务.

那么既然我们可以保证复制延迟小于 1s， 根据 ReadBinlogCoordinates 选择 Latest slave 又能导致"恢复时间"增大多少呢? 而为了这几秒的快速恢复，你又要花多少时间修复数据呢?

那么最准确的方式是要等所有 slave sql_thread 跑完. orchestrator 虽然调用了 WaitForSQLThreadUpToDate ，但只等待了 10s(超时).
随后运行 sortInstancesDataCenterHint 函数

```
// sortInstances shuffles given list of instances according to some logic
func sortInstancesDataCenterHint(instances [](*Instance), dataCenterHint string) {
   sort.Sort(sort.Reverse(NewInstancesSorterByExec(instances, dataCenterHint)))
}
```

这里做了个 Reverse 排序，具体如何排序的，要看 NewInstancesSorterByExec 的 Less 方法

关于 sort.Reverse

```
type reverse struct {
 // This embedded Interface permits Reverse to use the methods of
 // another Interface implementation.
 Interface
}>
// Less returns the opposite of the embedded implementation's Less method.
func (r reverse) Less(i, j int) bool {
 return r.Interface.Less(j, i)
}>
// Reverse returns the reverse order for data.
func Reverse(data Interface) Interface {
 return &reverse{data}
```

}

sort.Reverse 返回的是一个 *reverse. reverse 结构体就一个匿名字段 Interface reverse 上线了 Less 方法, 他本质就是使用 Interface.Less ，只不过调换了参数顺序 所以 Reverse() 虽然返回的是初始数据，但是改变了数据的 Less() 方法, 在排序时调用这个就会产生逆排序的效果.

NewInstancesSorterByExec 的 Less 方法

```
func (this *InstancesSorterByExec) Less(i, j int) bool {
   // Returning "true" in this function means [i] is "smaller" than [j],
   // which will lead to [j] be a better candidate for promotion
   // Sh*t happens. We just might get nil while attempting to
discover/recover   if this.instances[i] == nil {
      return false
   }
   if this.instances[j] == nil {
      return true
   }
   if
this.instances[i].ExecBinlogCoordinates.Equals(&this.instances[j].Exec
BinlogCoordinates) {
      // Secondary sorting: "smaller" if not logging replica updates
      if                this.instances[j].LogReplicationUpdatesEnabled
&& !this.instances[i].LogReplicationUpdatesEnabled {
         return true
      }
      // Next sorting: "smaller" if of higher version (this will be
reversed eventually)
      // Idea is that given 5.6 a& 5.7 both of the exact position, we
will want to promote
      // the 5.6 on top of 5.7, as the other way around is invalid
      if this.instances[j].IsSmallerMajorVersion(this.instances[i]) {
         return true
      }
      // Next sorting: "smaller" if of larger binlog-format (this will
be reversed eventually)
      // Idea is that given ROW & STATEMENT both of the exact position,
we will want to promote
      // the STATEMENT on top of ROW, as the other way around is invalid
      if this.instances[j].IsSmallerBinlogFormat(this.instances[i]) {
         return true
      }
      // Prefer local datacenter:
      if    this.instances[j].DataCenter    ==    this.dataCenter    &&
```

```
this.instances[i].DataCenter != this.dataCenter {
        return true
    }
    // Prefer if not having errant GTID
    if       this.instances[j].GtidErrant       ==       ""       &&
this.instances[i].GtidErrant != "" {
        return true
    }
    // Prefer candidates:
    if
this.instances[j].PromotionRule.BetterThan(this.instances[i].Promotion
Rule) {
        return true
    }
  }                                                              return
this.instances[i].ExecBinlogCoordinates.SmallerThan(&this.instances[j]
.ExecBinlogCoordinates)
}
```

简单来说，就是根据 ExecBinlogCoordinates 比较，如果 ExecBinlogCoordinates 相同在比 DataCenter ，DataCenter 与 DeadMaster 一样的为"大"

```
instance.ExecBinlogCoordinates.LogFile                              =
m.GetString("Relay_Master_Log_File")
instance.ExecBinlogCoordinates.LogPos                               =
m.GetInt64("Exec_Master_Log_Pos")
```

那么至此 sortInstancesDataCenterHint 干了啥也就清楚了，就是排了个序，把 most up-to-date 从库放在最前面，如果两个从库 ExecBinlogCoordinates 一样，则从库所在数据中心和主库一样的放前面

首要条件是 ExecBinlogCoordinates. PromotionRule 的"好坏"只是最次要的排序条件(因为他在最后一个 if 里). ExecBinlogCoordinates 相同时, 排序优先级是:

```
LogReplicationUpdatesEnabled
SmallerMajorVersion
SmallerBinlogFormat
same DataCenter with dead master
GtidErrant == ""
PromotionRule
```

接下来 GetCandidateReplica 会调用 chooseCandidateReplica ，初步选一个 candidate ，chooseCandidateReplica 接收的参数就是刚刚 sortedReplicasDataCenterHint 返回的排序后的 replicas 切片

chooseCandidateReplica

```
// chooseCandidateReplica
func chooseCandidateReplica(replicas []*Instance) (candidateReplica *Instance, aheadReplicas, equalReplicas, laterReplicas, cannotReplicateReplicas []*Instance, err error) {
    if len(replicas) == 0 {
        return candidateReplica, aheadReplicas, equalReplicas, laterReplicas, cannotReplicateReplicas, fmt.Errorf("No replicas found given in chooseCandidateReplica")
    }
    // 返回在给定实例中发现的主要（最常见）的 Major 版本
    // 比如 replicas 里有三个实例， 5.6.30, 5.7.32, 5.7.26. 那 priorityMajorVersion 就是 5.7
    priorityMajorVersion, _ := getPriorityMajorVersionForCandidate(replicas)
    // 返回在给定实例中发现的主要（最常见）binlog 格式
    // 比如 replicas 里有三个实例， mixed, row, row. 那么 priorityBinlogFormat 是 row
    priorityBinlogFormat, _ := getPriorityBinlogFormatForCandidate(replicas)

    for _, replica := range replicas {
        replica := replica
        if isGenerallyValidAsCandidateReplica(replica) && // 做一些简单的检测，比如 IsLastCheckValid, LogBinEnabled, LogReplicationUpdatesEnabled(前三个都应该为 true), IsBinlogServer(应为 false)
            !IsBannedFromBeingCandidateReplica(replica) && // 是否被参数 PromotionIgnoreHostnameFilters 匹配，希望不匹配
            !IsSmallerMajorVersion(priorityMajorVersion, replica.MajorVersionString()) && // 希望 replica 版本 <= priorityMajorVersion. 更希望高版本做低版本从库. 那比如最常见版本是 5.6, 然后有一个 replica 是 5.7, 他是那个 most up-to-date 的从库，到这里一比较，他就不符合条件，就被 pass 了
            !IsSmallerBinlogFormat(priorityBinlogFormat, replica.Binlog_format) { // 希望比如 priorityBinlogFormat row, 那 replica 是 mixed 或 statement
            // this is the one
            candidateReplica = replica
            break
        }
    }
    // 不用想那么多，以我们的场景，不存在 Major 版本不同的，
```

Binlog_format 也都是 row
// 那只要这个从库没什么"毛病", 也没在 PromotionIgnoreHostnameFilters 中, 那基本上 replicas[0] 就是 candidateReplica

// 如果上面的所有 replica 都不符合条件, candidateReplica 就=nil, 就会进入这个 if
if candidateReplica == nil {
// Unable to find a candidate that will master others.
// Instead, pick a (single) replica which is not banned.
for _, replica := range replicas {
replica := replica
if !IsBannedFromBeingCandidateReplica(replica) { // 选出第一个 not banned 的
// this is the one
candidateReplica = replica
break
}
}
// 如果选出了一个 not banned
if candidateReplica != nil {
// 把 candidateReplica 从 replicas 里移除
replicas = RemoveInstance(replicas, &candidateReplica.Key)
}
return candidateReplica, replicas, equalReplicas, laterReplicas, cannotReplicateReplicas, fmt.Errorf("chooseCandidateReplica: no candidate replica found")
}

// 能走到这里, 说明第一次循环就找到 candidateReplica 了
// 把 candidateReplica 从 replicas 里移除
replicas = RemoveInstance(replicas, &candidateReplica.Key)

// 迭代 replicas
for _, replica := range replicas {
replica := replica
// 如果这个实例不能做 candidateReplica 的从库, 就把它放到 cannotReplicateReplicas 切片里
if canReplicate, err := replica.CanReplicateFrom(candidateReplica); !canReplicate {
// lost due to inability to replicate
cannotReplicateReplicas = append(cannotReplicateReplicas, replica)

```
            if err != nil {
                log.Errorf("chooseCandidateReplica(): error checking CanReplicateFrom(). replica: %v; error: %v", replica.Key, err)
            }
        // 如果这个实例 ExecBinlogCoordinates SmallerThan candidateReplica.ExecBinlogCoordinates, 放到 laterReplicas
        } else if replica.ExecBinlogCoordinates.SmallerThan(&candidateReplica.ExecBinlogCoordinates) {
            laterReplicas = append(laterReplicas, replica)

        // 如果这个实例 ExecBinlogCoordinates == candidateReplica.ExecBinlogCoordinates, 放到 equalReplicas
        } else if replica.ExecBinlogCoordinates.Equals(&candidateReplica.ExecBinlogCoordinates) {
            equalReplicas = append(equalReplicas, replica)

        // 佛足额, 说明这个实例 ExecBinlogCoordinates > candidateReplica.ExecBinlogCoordinates, 放到 aheadReplicas
        } else {
            // lost due to being more advanced/ahead of chosen replica.
            aheadReplicas = append(aheadReplicas, replica)
        }
    }
    return candidateReplica, aheadReplicas, equalReplicas, laterReplicas, cannotReplicateReplicas, err
}
```

chooseCandidateReplica 选了一个 candidateReplica 出来 并且对其他 replica 做了归类(laterReplicas , equalReplicas , aheadReplicas)
CanReplicateFrom 的具体逻辑请看 https://github.com/Fanduzi/orchestrator-zh-doc 配置参数详解-Ⅱ 中对该参数的详细解释

从代码可以看出 orchestrator 并不会以 0 数据丢失为最优先级选择 candidate

```
    // 返回在给定实例中发现的主要（最常见）的 Major 版本
    // 比如 replicas 里有三个实例, 5.6.30, 5.7.32, 5.7.26. 那 priorityMajorVersion 就是 5.7
    priorityMajorVersion, _ := getPriorityMajorVersionForCandidate(replicas)
    // 返回在给定实例中发现的主要（最常见）binlog 格式
    // 比如 replicas 里有三个实例, mixed, row, row. 那么
```

priorityBinlogFormat 是 row
    priorityBinlogFormat, _ := getPriorityBinlogFormatForCandidate(replicas)

    for _, replica := range replicas {
        replica := replica
        if isGenerallyValidAsCandidateReplica(replica) && // 做一些简单的检测，比如 IsLastCheckValid, LogBinEnabled, LogReplicationUpdatesEnabled(前三个都应该为 true), IsBinlogServer(应为 false)
            !IsBannedFromBeingCandidateReplica(replica) && // 是否被参数 PromotionIgnoreHostnameFilters 匹配, 希望不匹配
            !IsSmallerMajorVersion(priorityMajorVersion, replica.MajorVersionString()) && // 希望 replica 版本 <= priorityMajorVersion. 更希望高版本做低版本从库. 那比如最常见版本是 5.6, 然后有一个 replica 是 5.7, 他是那个 most up-to-date 的从库, 到这里一比较, 他就不符合条件, 就被 pass 了
            !IsSmallerBinlogFormat(priorityBinlogFormat, replica.Binlog_format) { // 希望比如 priorityBinlogFormat row, 那 replica 是 mixed 或 statement
            // this is the one
            candidateReplica = replica
            break
        }
    }
上述代码中的 replicas 是 sortInstancesDataCenterHint 返回的，按 ExecBinlogCoordinates 从大到小排序的切片(即 ExecBinlogCoordinates 最大的 index 是 0) 但是最终 candidateReplica 是否是 replicas[0] ，取决于其 MajorVersion 和 BinlogFormat (当然还有 isGenerallyValidAsCandidateReplica 和 IsBannedFromBeingCandidateReplica)
在官方文档 Discussion: recovering a dead master 中也有如下描述：Find the best replica to promote .
一种天真的方法是选择最新的副本, 但这可能并不总是正确的选择
A naive approach would be to pick the most up-to-date replica, but that may not always be the right choice.

最新的副本可能没有必要的配置来充当其他副本的主节点(例如，binlog 格式、MySQL 版本控制、复制过滤器等). 一味地推广最新的副本可能会丢失副本容量
It may so happen that the most up-to-date replica will not have the necessary configuration to act as master to other replicas (e.g. binlog format, MySQL versioning, replication filters and more). By blindly promoting the most up-to-date replica one may lose replica capacity.
orchestrator 尝试提升将保留最多服务容量的副本.orchestrator attempts to

promote a replica that will retain the most serving capacity.
提升所述副本，接管其同级
Promote said replica, taking over its siblings.

Bring siblings up to date
可能的话，做第二阶段选举提升；如果可能的话，用户可能已经标记了要提升的特定服务器(见 register-candidate 命令)

Possibly, do a 2nd phase promotion; the user may have tagged specific servers to be promoted if possible (see register-candidate command).

但针对我们的场景，同一个集群不存在 Major 版本不同实例，Binlog_format 也都是 row 那只要这个从库没什么"毛病"，也没在 PromotionIgnoreHostnameFilters 中，那基本上 replicas[0] 就是 candidateReplica
那么继续看 GetCandidateReplica 剩下的代码

```
    candidateReplica, aheadReplicas, equalReplicas, laterReplicas, cannotReplicateReplicas, err = chooseCandidateReplica(replicas)
    if err != nil { // 如果 chooseCandidateReplica 走到 if candidateReplica == nil { ,就会进入这个 if
        return candidateReplica, aheadReplicas, equalReplicas, laterReplicas, cannotReplicateReplicas, err
    }
    if candidateReplica != nil {
        mostUpToDateReplica := replicas[0]

        // 这是有可能的
        // 比如最常见版本是 5.6，然后有一个 replica 是 5.7，他是那个 most up-to-date 的从库，但它比 priorityMajorVersion 大. 他就不适合做 candidate
        if candidateReplica.ExecBinlogCoordinates.SmallerThan(&mostUpToDateReplica.ExecBinlogCoordinates) {
            log.Warningf("GetCandidateReplica: chosen replica: %+v is behind most-up-to-date replica: %+v", candidateReplica.Key, mostUpToDateReplica.Key)
        }
    }
    log.Debugf("GetCandidateReplica: candidate: %+v, ahead: %d, equal: %d, late: %d, break: %d", candidateReplica.Key, len(aheadReplicas), len(equalReplicas), len(laterReplicas), len(cannotReplicateReplicas))
    return candidateReplica, aheadReplicas, equalReplicas, laterReplicas, cannotReplicateReplicas, nil
```

```
}
```
现在再看 RegroupReplicasGTID
```
// RegroupReplicasGTID will choose a candidate replica of a given instance, and take its siblings using GTID
func RegroupReplicasGTID(
    masterKey *InstanceKey, // 实参传进来的是 挂掉的旧主库
    returnReplicaEvenOnFailureToRegroup bool, // 实参传进来的是 true
    startReplicationOnCandidate bool, // 实参传进来的是 false
    onCandidateReplicaChosen func(*Instance), // 实参传进来的是 nil
    postponedFunctionsContainer *PostponedFunctionsContainer,
    postponeAllMatchOperations func(*Instance, bool) bool, // 实参传进来的是 promotedReplicaIsIdeal 函数
) (
    lostReplicas [](*Instance),
    movedReplicas [](*Instance),
    cannotReplicateReplicas [](*Instance),
    candidateReplica *Instance,
    err error,
) {
    var emptyReplicas [](*Instance)
    var unmovedReplicas [](*Instance)

    // candidateReplica 有可能==nil
    candidateReplica, aheadReplicas, equalReplicas, laterReplicas, cannotReplicateReplicas, err := GetCandidateReplica(masterKey, true)


    // 如果 chooseCandidateReplica 走到 if candidateReplica == nil { ,就会进入这个 if
        // Unable to find a candidate that will master others.
        // Instead, pick a (single) replica which is not banned.
    if err != nil {
        // returnReplicaEvenOnFailureToRegroup 实参传进来的是 true
        if !returnReplicaEvenOnFailureToRegroup {
            candidateReplica = nil
        }
        return emptyReplicas, emptyReplicas, emptyReplicas, candidateReplica, err
    }

    // onCandidateReplicaChosen 实参传进来的是 nil
    if onCandidateReplicaChosen != nil {
        onCandidateReplicaChosen(candidateReplica) // 所以走不到这里
    }
```

```
    // equalReplicas 和 laterReplicas 都可以做 candidateReplica 的从库，所以放到 replicasToMove 里
    replicasToMove := append(equalReplicas, laterReplicas...)
    hasBestPromotionRule := true
    if candidateReplica != nil {
        // 迭代 replicasToMove
        for _, replica := range replicasToMove {
            // 比较 PromotionRule. 判断 candidateReplica 是不是用户 prefer 的
            if replica.PromotionRule.BetterThan(candidateReplica.PromotionRule) {
                hasBestPromotionRule = false
            }
        }
    }
    moveGTIDFunc := func() error {
        log.Debugf("RegroupReplicasGTID: working on %d replicas", len(replicasToMove))

        // moves a list of replicas under another instance via GTID, returning those replicas
        // that could not be moved (do not use GTID or had GTID errors)
        movedReplicas, unmovedReplicas, err, _ = moveReplicasViaGTID(replicasToMove, candidateReplica, postponedFunctionsContainer)
        unmovedReplicas = append(unmovedReplicas, aheadReplicas...)
        return log.Errore(err)
    }

    // 这个 postponeAllMatchOperations 就是 recoverDeadMaster 中定义的 promotedReplicaIsIdeal
    // 做一些判断，但基本上就是看 hasBestPromotionRule 和 candidateReplica 的 promotion rule 是不是 MustNotPromoteRule. 如果 candidateReplica 就是理想的，那 moveGTIDFunc 就放到异步推迟执行
    if postponedFunctionsContainer != nil && postponeAllMatchOperations != nil && postponeAllMatchOperations(candidateReplica, hasBestPromotionRule) {
        postponedFunctionsContainer.AddPostponedFunction(moveGTIDFunc, fmt.Sprintf("regroup-replicas-gtid %+v", candidateReplica.Key))
    } else {
    // 否则同步执行
        err = moveGTIDFunc()
    }
```

// 我没太看懂上面那个 if else, 除了 canidateReplica 恰好是 prefer 的那个时多输出一个日志义务外, 和 else 时有啥区别吗?

if startReplicationOnCandidate { // 实参传进来的是 false. 在 DeadMaster 场景, 这里不能传 true, 因为 StartReplication 会调用 MaybeEnableSemiSyncReplica, 而后者需要连接 old master
// 但是 old master 已经挂了所以肯定连不上, 于是这里出现 error 直接 return 了, 后面真正的 start slave 是没机会执行的
StartReplication(&candidateReplica.Key)
}

log.Debugf("RegroupReplicasGTID: done")
AuditOperation("regroup-replicas-gtid", masterKey, fmt.Sprintf("regrouped replicas of %+v via GTID; promoted %+v", *masterKey, candidateReplica.Key))
return unmovedReplicas, movedReplicas, cannotReplicateReplicas, candidateReplica, err
}

所以 RegroupReplicasGTID
选了个 candidateReplica 出来, 并给其他 replica 归了类

这个 candidateReplica 不一定是最终的主库, 只是它的 ExecBinlogCoordinates 大

通过 moveReplicasViaGTID ,将其他的 replica(除了 aheadReplicas 和 cannotReplicateReplicas)都 change master 到了 candidateReplica

至此 RegroupReplicasGTID 工作完成了, 剩下的工作可以交换给 recoverDeadMaster 了, 就是 recoverDeadMaster 主要做了几件事中的最后两步操作

recoverDeadMaster 主要做了几件事

GetMasterRecoveryType ,确定到底用什么方式恢复,是基于 GTID ? PseudoGTID ? 还是 BinlogServer ?
重组拓扑, 我们的案例是使用 RegroupReplicasGTID. 但这里有一个问题, 可能现在我们的新主库并不是我们"期望"的实例, 就是说之所以选他做主库可能是因为他有最全的日志。但不是我们设置的 prefer 的所以通过一个闭包 promotedReplicaIsIdeal 去做了判断和标记
如果存在 lostReplicas ,并且开启了 DetachLostReplicasAfterMasterFailover, 那么会并行的对所有 lostReplicas 执行 DetachReplicaMasterHost. 其实就是执行 change master to master_host='// {host}'
如果当前选举的 new master 不是我们 prefer 的实例, 重组拓扑, 用 prefer 做新主库

DelayMasterPromotionIfSQLThreadNotUpToDate 有 bug
根据 Orchestrator Failover 过程源码分析-Ⅱ
没有在 recoverDeadMaster 和 checkAndRecoverDeadMaster 中看到任何代码执行了 start slave sql_thread
那在 checkAndRecoverDeadMaster 执行到这里时，不是肯定会超时吗

```
        if    config.Config.DelayMasterPromotionIfSQLThreadNotUpToDate
&& !promotedReplica.SQLThreadUpToDate() {
            AuditTopologyRecovery(topologyRecovery,
fmt.Sprintf("DelayMasterPromotionIfSQLThreadNotUpToDate:  waiting  for
SQL thread on %+v", promotedReplica.Key))
            if                  _,                  err                  :=
inst.WaitForSQLThreadUpToDate(&promotedReplica.Key, 0, 0); err != nil {
                return                                                  nil,
fmt.Errorf("DelayMasterPromotionIfSQLThreadNotUpToDate error: %+v", err)
            }
            AuditTopologyRecovery(topologyRecovery,
fmt.Sprintf("DelayMasterPromotionIfSQLThreadNotUpToDate:  SQL  thread
caught up on %+v", promotedReplica.Key))
        }
```

实际测试确实有问题．在 issue https://github.com/openark/orchestrator/issues/1430 中，我们内部分支也修复了这个 bug ，我的同事也在 issue 中给出了我的解决办法。

总结
图片
点击可放图

本文关键字：#MySQL 高可用# #Orchestrator#

文章推荐：

技术分享 | Orchestrator Failover 过程源码分析-I

技术分享 | Orchestrator Failover 过程源码分析-II

技术分享 | 大数据量更新，回滚效率提升方法

故障分析 | 填坑 TaskMax
关于 SQLE
爱可生开源社区的 SQLE 是一款面向数据库使用者和管理者，支持多场景审核，支持标准化上线流程，原生支持 MySQL 审核且数据库类型可扩展的 SQL 审核工具。

SQLE 获取

类型	地址
版本库	https://github.com/actiontech/sqle
文档	https://actiontech.github.io/sqle-docs-cn/
发布信息	https://github.com/actiontech/sqle/releases
数据审核插件开发文档	https://actiontech.github.io/sqle-docs-cn/3.modules/3.7_auditplugin/auditplugin_development.html

更多关于 SQLE 的信息和交流，请加入官方 QQ 交流群：637150065...