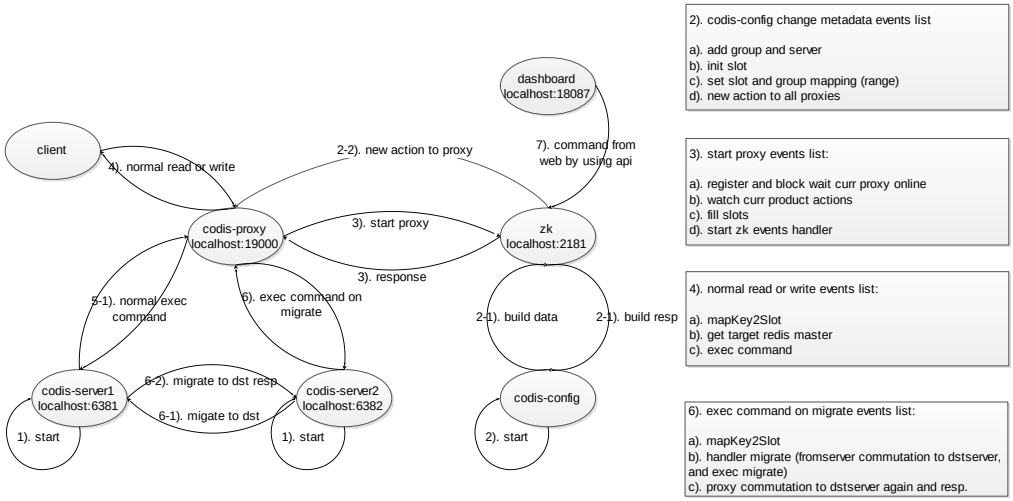
Codis 源码分析

源码文件夹功能(2015 年农历新年前版本)

目录	功能
models	Models 包，封装了 slot, server-group, server, proxy, action 对象操作接口以及常量定义。所有关于对象的操作（读取、修改、关系等），通过该包进行调用。
proxy	proxy-server 实现相关
cachepool	缓存池相关
group	proxy-server 封装对 group 的操作
parser	proxy-server 中对解析相关的操作
redispool	Proxy 与 redis 操作的池实现
route	服务启动，接收请求，反向代理等
topology	封装对 zookeeper 的统一操作接口，包括 slot、server-group、proxyInfo 等，供 proxyserver 调用。内部会使用 models 接口。
utils	提供一些工具操作接口，包括与 redis 和通用的
cmd	包括 codis-config 服务和 proxy 主入口
cconfig	codis-config 服务相关
main.go	codis-config 入口 main
proxy.go	condis-config 与 proxy 之间通信的接口
slot.go, server-group.go	封装 codis-config 与 slot， server-group 操作相关的接口。会用到 models 内部的对象接口。并会同步到 zk 上
migrate、负载均衡相关	
action 相关	codis-config 产生 action 的接口，并同步到 zk 上
dashbord 相关	
proxy	Proxy 的主入口 main
extern	针对 redis-2.8.13 封装的几个命令实现，以及 redis-port

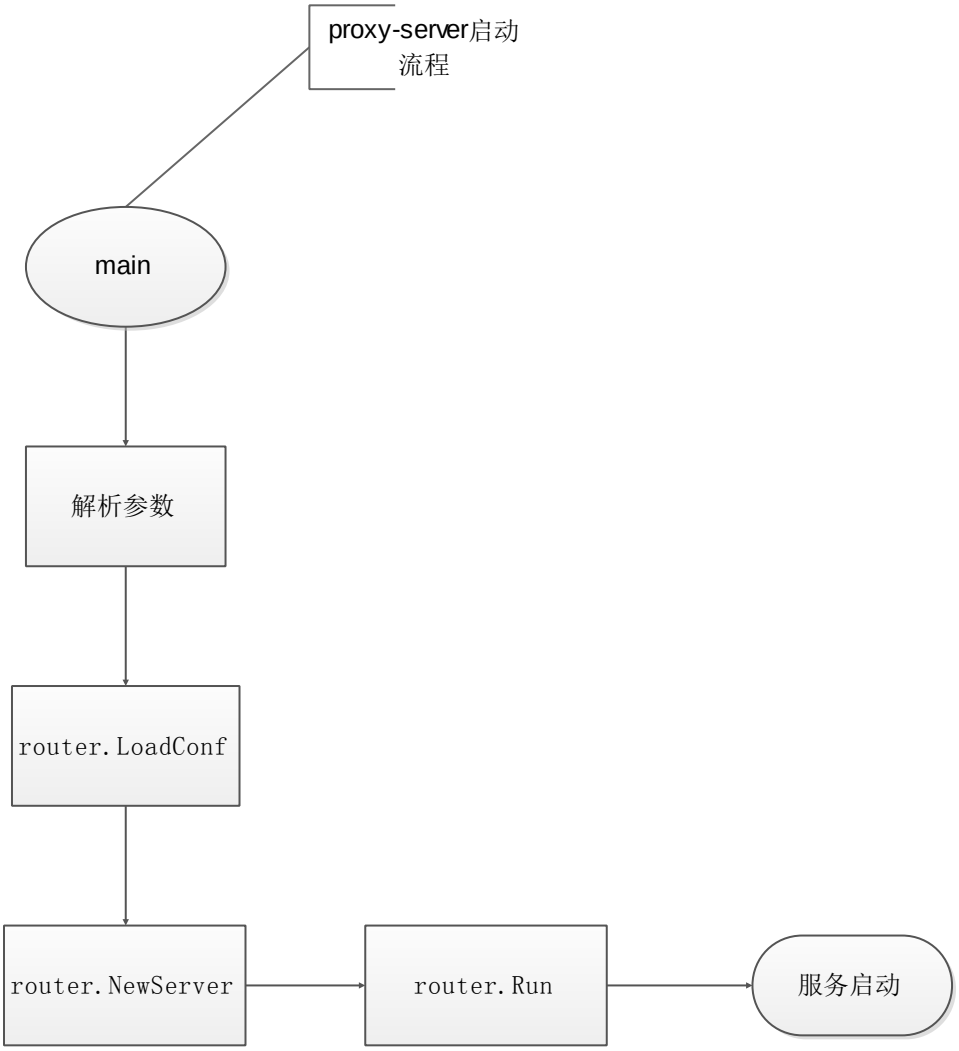
主要流程

Codis 整体服务结构图

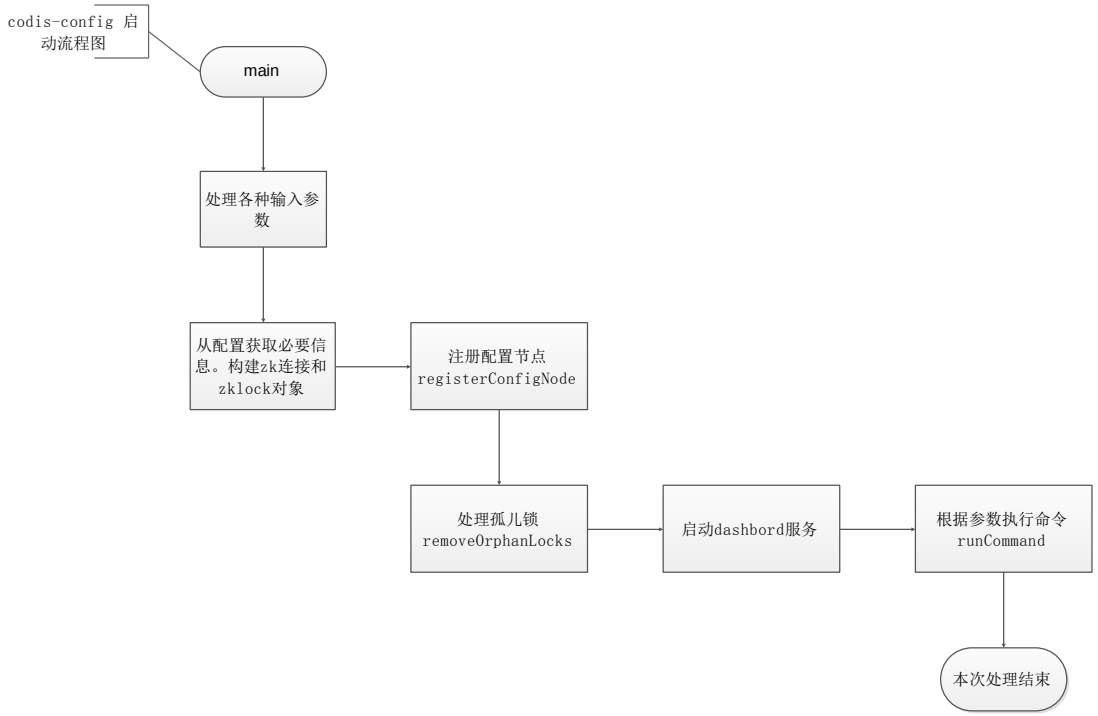


服务启动流程

1、 proxy-server 启动

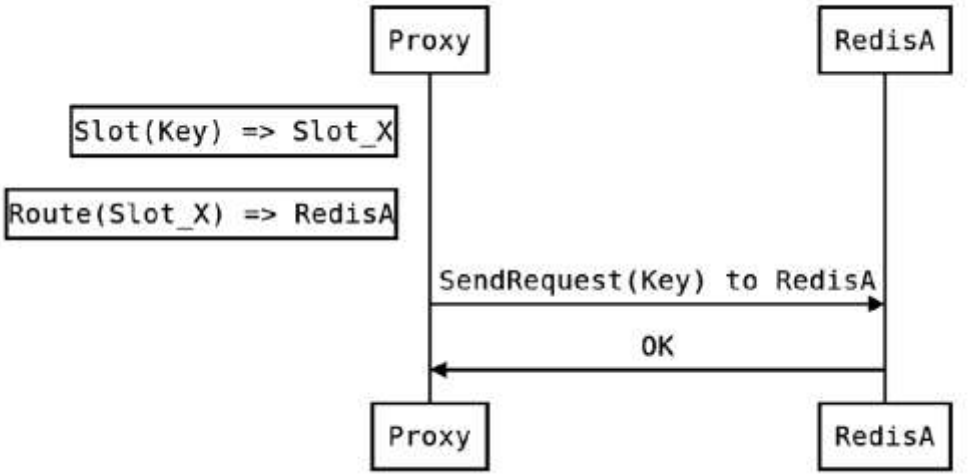


2、 codis-config 启动

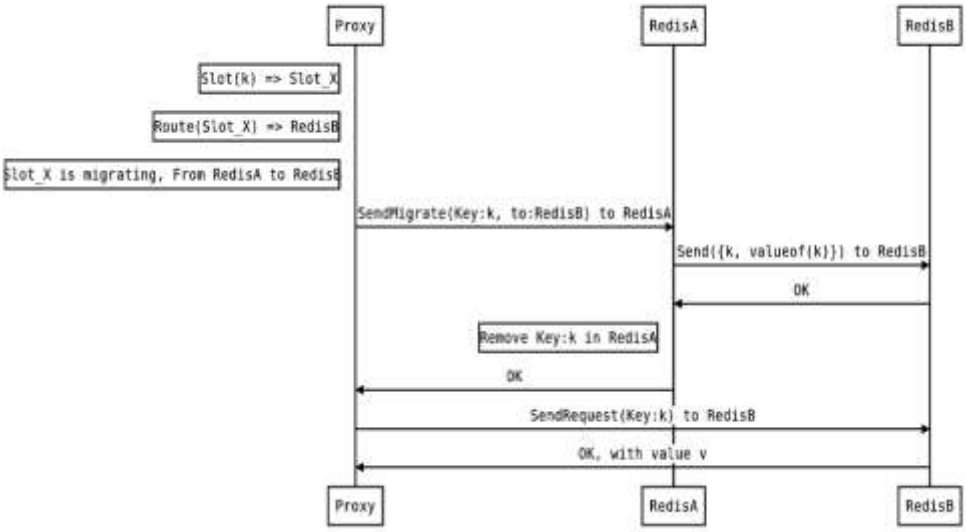


各种读写流程

1、 正常读写流程

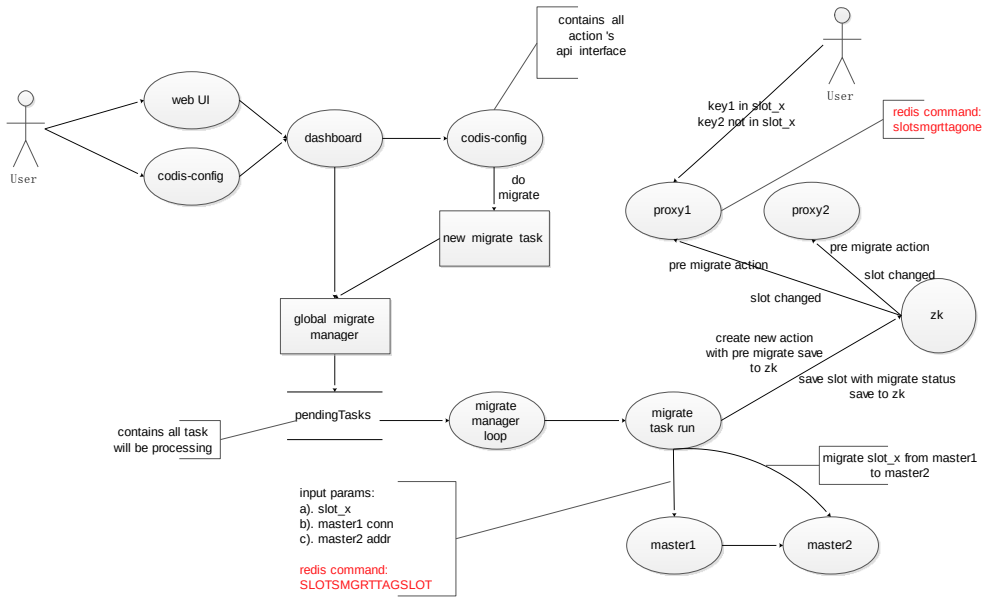


2、 Migrate 状态时，读写流程



codis-config 发起操作

1、 发起 migrate 操作，主要对象间关系



全局变量

Codis-config

全局变量名称	含义
productName	从配置文件中获取的产品名称
zkAddr	从配置中获取的 zk ip:port 值

zkConn	创建的 zk 连接对象。和 zk 交互过程中都需要使用该对象
zkLock	根据 zk 连接对象和 productName 构建的 zk 锁对象
livingNode	服务启动时，创建对应的 codis-config 服务所在主机和进程 id，在 zk 上对应的节点
configFile	
config	

Proxy-server

对象模型及关系

proxy

结构体定义

接口

Model	proxy
文件	Models/proxy.go
接口名	**功能**
GetProxyPath	获取指定业务线在 zk 上的节点存储路径
GetProxyInfo	从 zk 上某 productName 业务线下，根据输入的 proxyname 获取对应的 proxy 内存对象(从 json 转换为 proxy 内存对象)
ProxyList	从 zk 上获取指定 productName 对应 proxy 根节点下的所有 children，返回所有的 proxyInfo 对象数组（每一个 proxy 对象都从 json 转换为 proxy 内存对象）
CreateProxyInfo	在 zk 上指定的 productName 业务线下，创建新的 proxy 节点。(该方法在 proxy server 启动过程中被调用，相当于 proxy server 向 zk 进行注册)
SetProxyStatus	在 zk 上指定的 productName 业务线下，根据输入的 proxyname 和状态名称，修改 proxy 节点信息。涉及到 proxy 的 online 和 offline

slot

结构体定义

```
type Slot struct {
    // 产品线名称
ProductName string      `json:"product_name"`
    // slot ID
    Id              int          `json:"id"`
    // Slot 所属组 ID
    GroupId         int          `json:"group_id"`
    // 维护 Slot 操作相关信息
State           SlotState `json:"state"`
}

type SlotState struct {
    //Slot 状态信息
    Status              SlotStatus              `json:"status"`
    //Slot 迁移相关数据信息
    MigrateStatus SlotMigrateStatus `json:"migrate_status"`
    //Slot 状态信息修改最后时间
    LastOpTs            string                  `json:"last_op_ts"` // operation timestamp
}

type SlotMigrateStatus struct {
    //迁移开始 Slot ID
    From int `json:"from"`
    //迁移结束 SlotID。如果只迁移一个 Slot，则 From 等于 To
    To     int `json:"to"`
}
```

接口

Model	slot
文件	models/slot.go
接口名	**功能**
GetSlotBasePath	获取某 productName 在 zk 上的 slots 根节点路径。
InitSlotSet	
NewSlot	构建 slot 内存对象
Update	将 slot 内存对象同步到 zk 上
GetSlot	从 zk 上将 slot 节点信息映射成内存对象（json 转换为内存对象）
GetSlotPath	获取某个 slot 在 zk 上的节点路径
SetSlotRange	修改某个 slot 的所属组和状态，并同步到 zk 上。支持同时修改多个 slot。

	该操作会触发 NewAction，Type 为 ACTION_TYPE_MULTI_SLOT_CHANGED，回向所有的 proxy 发出 action
GetMigratingSlots	获取 zk 上某条业务线下所有 slot 中，正在执行迁移的 slot 数组。要么数组长度为 0，代表当前没有进行迁移的 slot。要么长度为 1，代表当前正有一个 slot 执行迁移。如果数组长度大于 1，业务逻辑错误。（migrate 过程中调用）
Slots	从 zk 上获取某条业务线下的所有 slot 节点，并转换为内存对象数组。
SetMigrateStatus	触发 NewAction 操作，Type 为 ACTION_TYPE_SLOT_PREMIGRATE，向所有的 proxy 发起 action，并等待应答。应答完毕后，更新当前 slot 的状态，fromgroup，togroup 到 zk 上
NoGroupSlots	获取 zk 上的所有的还没有分配 Group 的 slot 对象数组
SetSlots	将给定的未设置 group 的 slot 数组，分配上给定的 groupid，并同步到 zk 上
Migrate 和 rebalance 未列举完毕	

server-group

结构体定义

接口

Model	server-group
文件	models/server_group.go
接口名	**功能**
ServerGroups	获取 zk 上某业务线所有的 ServerGroup 对象数组（每一个 Group 对象上已经设置了属于该组的服务器对象数组），内部调用 GetGroup
GetGroup	从 zk 上获取某个特定 Group 对象（每一个 group 对象内部也设置了 group 包含的 server 子对象列表）
GroupExists	从 zk 上判断指定 productName 和 groupId 对应的对象是否存在。静态方法
GetServers	获取属于某个组的所有服务器对象数组，内部调用 GetServer
GetServer	从 zk 上获取属于某个 productName 的 server

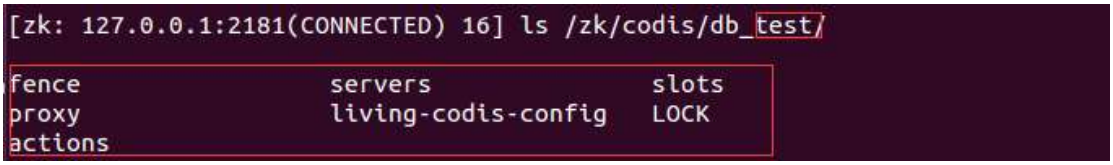
	节点的内存对象（json 转换为 server 内存对象）
Exists	判断一个 ServerGroup 是否在 zk 上存在。实例方法。
Create	在 zk 上某条 productName 下创建一个新的 ServerGroup，该操作会触发执行 NewAction，Type 为 ACTION_TYPE_SERVER_GROUP_CHANGED，向所有的 proxy 发送 action
RemoveServer	从 zk 上删除特定 productName 对应的 group 组下的某个数据存储服务器。该操作会触发执行 NewAction，Type 为 ACTION_TYPE_SERVER_GROUP_CHANGED，向所有的 proxy 发送 action
NewServerGroup	构建一个 ServerGroup 内存对象
Remove	从 zk 上移除指定的 ServerGroup（只有 ServerGroup 没有被任意 slot 使用，才可以删除），该操作会触发执行 NewAction，type 为 ACTION_TYPE_SERVER_GROUP_REMOVE，向所有的 proxy 发送 action
NewServer	构建一个 Server 内存对象
AddServer	从当前组中添加一个新的 Server 节点，并同步到 zk 对应的 productName 业务线空间下。如果添加的为 master server，必须保证当前组中没有一个 server 类型为 master。如果添加的 server 是 master，会触发执行 NewAction，Type 为 ACTION_TYPE_SERVER_GROUP_CHANGED，向所有的 proxy 发送 action
Promote ??	将给定组中的另一台存储服务器设置为 master。将原有标记为 master 的 server 标记为 offline。

zk 节点介绍

Codis 可以同时支持不同的产品线。每一个产品线在 zk 上都一个根节点。属于这个产品线下的所有相关数据都做为这个 zk 根节点的 children 存在。

Codis 涉及的所有元数据（slot, server-group, server, lock 等）、元数据之间的关系（slot 与 group 等），一些通知数据（action 等）在 zookeeper 上都作为一个 zk 节点存储到 zk 上。根据业务逻辑组成层级关系。

Eg：一个 test 产品线下，zk 结构如下：



变量	含义
$productName	产品线名称。在 codis-config 服务的 config.ini 中设置
$slot_id	槽 id
$groupId	组 id
$proxyId	Proxy server Id
$action_seq_Id	Action seq Id ??
$hostname	Codis-config 服务主机名
$pid	codis-config 服务 pid

Zk 节点及描述
/zk/codis/db_$productName/ 产品线根节点。包含属于这个产品线的所有元数据和控制信息
/zk/codis/db_$productName/servers 某条产品线下的服务器组根节点，其 children 为所有的服务器组节点。比如：group_1, group_2
/zk/codis/db_$productName/servers/group_$groupId 某条产品线下的某个服务器组根节点，其 children 为属于这个服务器组的所有服务器节点
/zk/codis/db_$productName/servers/group_$groupId/redis-server:port 某条产品线下的某个服务器组下的某台服务器节点。该节点内容包含 Server 的 json 数据 {"type":"master","group_id":1,"addr":"localhost:6381"} /zk/codis/db_test/servers/group_1/localhost:6381
/zk/codis/db_$productName/proxy 某条产品线下的 proxyserver 根节点。其 children 为所有的 proxy server 列表
/zk/codis/db_$productName/proxy/proxy_$proxyId 某条产品线下的某个代理服务器节点。节点内容： {"id":"proxy_1","addr":"ubuntu:19000","last_event":"","last_event_ts":0,"state":"online","description":"","debug_var_addr":"ubuntu:11000"}

/zk/codis/db_$productName/slots 某条业务线 slot 根节点，其 children 为所有默认分配的 slot 节点集合。
/zk/codis/db_$productName/slots/slot_$slotId 某条业务线某个特定 slot 节点。其没有 children，节点内容为： {"product_name":"test","id":0,"group_id":1,"state":{"status":"online","migrate_status":{"from":-1,"to":-1},"last_op_ts":"0"}} 主要字段：槽 Id、槽所属组 Id、槽状态、槽迁移状态
/zk/codis/db_$productName/actions 某条业务线 action 根节点，其 children 为所有当前 action 事件
/zk/codis/db_$productName/actions/action_$action_seq_Id 某条业务线某个 action 节点。内容 {"type":"slot_changed","desc":"","target":{"product_name":"test","id":923,"group_id":-1,"state":{"status":"offline","migrate_status":{"from":-1,"to":-1},"last_op_ts":"0"}},"ts":"1423713352","receivers":null}
/zk/codis/db_$productName/living-codis-config/ 某条业务线当前 codis-config 根节点。其 children 为所有的 codis-config server 集合。
/zk/codis/db_$productName/living-codis-config/$hostname-$pid 某条业务线某个活动的 codis-config 节点。其内容： {"hostname": "ubuntu", "pid": 27528}
/zk/codis/db_$productName/LOCK 某条业务线 zk 锁节点。任何正在迁移的过程，需要获取该锁节点。
/zk/codis/db_%s/migrate_tasks MigrateManager 维护着正在执行的迁移任务根节点

/zk/codis/db_%s/dashboard Dashboard 在 zk 上的节点根路径

一致性方面的设计

1、迁移期间一致性的实现，是一个两阶段提交的过程
准备阶段
第一阶段
2、构建 action，将 action 类型设置为 pre_migrate，设置 action 的接收者为该业务线所有的 proxies。
3、等待所有 proxies 的回应，通知迁移服务，对方已经知道集群进入到待迁移状态。

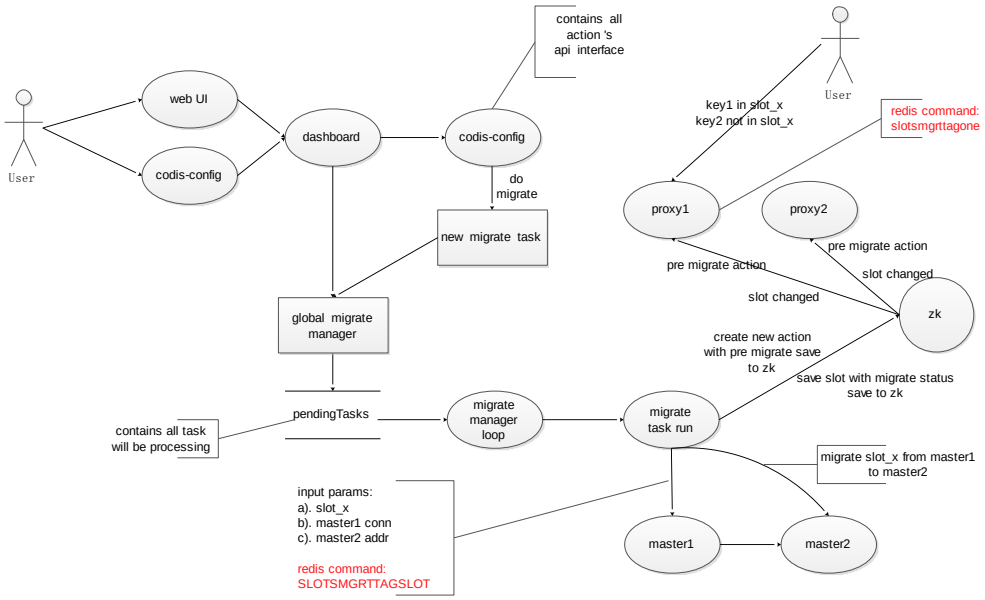
修改阶段
第二阶段

4、如果迁移服务能够确认所有的 proxy 都进入到了 pre_migrate 状态，它可以当前 slot 的状态为 migrate。并再次构建新的 action，通知所有的 proxy 这个改动。
5、如果没能收到所有 proxy 的回应，是不能够进入到 migrate 状态的，那么要放弃迁移操作。
6、对于 proxy 不响应的情况，可以将它标记为 offline，迁移程序退出，由管理员来处理异常（人为接管）
7、proxy 在 pre_migrate 状态时是不能执行读、写操作的
8、即便迁移程序异常退出，整个迁移过程也只是影响了一个 slot，锁的力度不会太大
9、即使迁移程序不去向 redis 发迁移命令，proxy 也会慢慢地随着请求将迁移操作执行完
10、如果在 pre_migrate 状态的机器丢失了发给它的 migrate 状态，那么这台机器的写操作将不可用，可用性降低

参考：

http://www.zenlife.tk/codis 数据迁移期间的一致性.md

迁移时，各服务、对象间流程

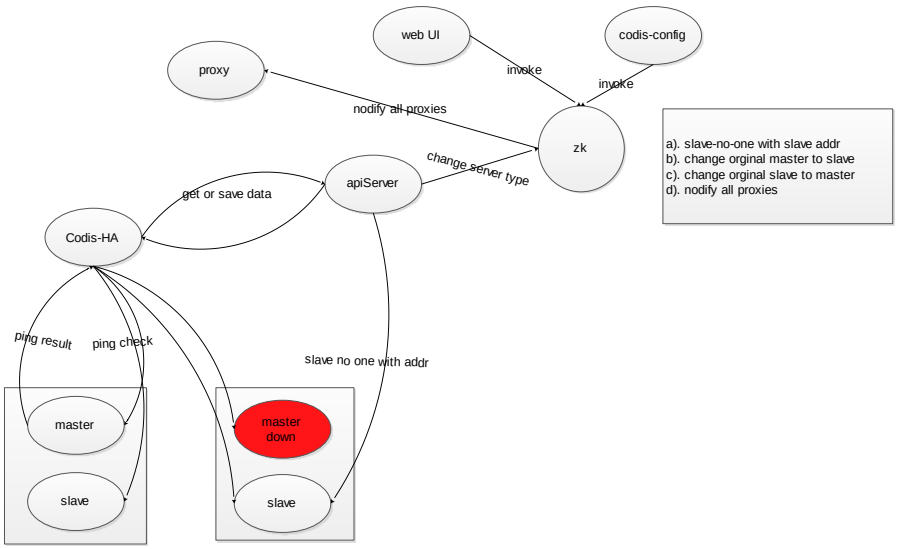


HA 实现方式

1、 通过一个外部服务检测（ping）redis 是否挂掉

2、 使用 api 方式提供统一接口，供 webUI、命令行、检测服务调用，来将 slave 提升为 master

Codis-Ha 关系图



Codis 应用场景及故障表现、缺点

添加数据节点

1、启动 redis-server
2、通过界面或 codis-config 命令行，在 zk 上注册数据节点
3、发起迁移操作。通过设定路由表

移除数据节点

1、通过界面或命令行，将要下线的实例拥有的 slot 迁移到其它实例上
2、通过界面或命令删除下线的 group

数据节点故障

1、 单 redis 数据节点，没有配置 slave
   a) 如果未持久化，属于该实例的 slot 对应的数据将丢失

2、 Redis 为 master-slave 模式，使用 redis Ha 方式
   a) Master 故障时，从切换到主的期间，属于该实例的 slot 对应的 key 的写操作失败（只是部分 key）
   b) 主从切换完毕后，需要通过界面或命令行手动将从对应的机器设置为主
   c) 并能同步到所有 proxy
   d) 此时，写操作正常

3、 Redis 为 master-slave 模式，使用 codis-Ha 方式（详细见 codis-Ha 关系图）
   a) 启动 codis-Ha 服务
   b) 不断检测所有的数据节点状态（包括 master 和 slave）。ping 操作
   c) 检测到某个 master 数据节点不可用
   d) 向 redis-slave 节点发送 slave-no-one 操作，将 redis-slave 设置为主
   e) change server-group node info(原 slave 修改为 master)
   f) 向所有的 proxy 发送 server-group changed action

添加 proxy 节点

1、proxy 服务启动，并向 zk 注册
2、通过面板或命令行将该 proxy 设置为 online
3、proxy 填充 slot 信息，开始接收服务
4、proxy 是无状态服务，只要有了 slot 信息，即可开始服务

proxy 下线

1、proxy 接收到 kill 信号，主动退出
handleMarkOffline（删除 proxy 节点，fence 节点）

2、收到面板或 configserver 的命令行发送的 proxy event 事件。(mark offine)
handleMarkOffline

3、注册过程中，收到面板或命令行发送的 mark offline 事件
handleMarkOffline

4、注册过程中，收到 kill 信号，主动退出

4、 proxy 与 zk 服务之间的网络存在异常。存在几种特殊的情况
a) 正常情况下，zk 节点不变动，proxy 使用其维护的内存对象
b) 用户通过 codis-config 或面板修改节点信息后，一定时间内 zk 得不到 proxy 的响应信息，codis-config 会发起 markoffline 的操作。如果操作成功，则 proxy 服务会被停止，同时在 zk 上的对应节点会删除。
c) 如果由于网络中断不能读取 proxy 响应或不能向 proxy 发起 markoffline，则这段时间内 proxy 可以继续提供服务

附录 A-部分源码分析

Codis-config

产生 action 事件

列举出需要 proxy-server 调用 processAction 处理的 TopoEvent

调用关系：handleTopoEvent->processAction->checkAndDoTopoChange

EventActionType 包括：

ACTION_TYPE_SLOT_MIGRATE

ACTION_TYPE_SLOT_CHANGED

ACTION_TYPE_SLOT_PREMIGRATE

ACTION_TYPE_MULTI_SLOT_CHANGED

ACTION_TYPE_SERVER_GROUP_CHANGED

每个 slot 拥有的状态只包括：

Online，Offline，Pre_Migrate，Migrate

Slot 迁移(Pre_Migrate 或 Migrate)，Slot 状态修改（online 或 offline）

ServerGroup 状态修改（修改了组的 Master）

apiSlotRangeSet

```
err = NewAction(zkConn, productName, ACTION_TYPE_MULTI_SLOT_CHANGED, param, "", true)
```

SetMigrateStatus

```
err := NewAction(zkConn, s.ProductName, ACTION_TYPE_SLOT_PREMIGRATE, s, "", true)
```

Update

```
//修改 slot 如果是为了迁移，则通知 Slot_Migrate
err = NewAction(zkConn, s.ProductName, ACTION_TYPE_SLOT_MIGRATE, s, "", true)

//修改 slot 如果只是更改 slot 的状态，online 或 offline，则通知 Slot_Changed
err = NewAction(zkConn, s.ProductName, ACTION_TYPE_SLOT_CHANGED, s, "", true)
```

AddServer

```
//如果添加的 server 类型为 master，则需要通知所有的 proxy
err = NewAction(zkConn, self.ProductName, ACTION_TYPE_SERVER_GROUP_CHANGED, self, "",
true)
```

ProxyServer

主要成员功能

slots

维护的 proxyserver 中所有 slot 对象实例数组。每次修改 zk 上的 slot 信息后，需要重新 fillSlot。

reqCh

作为 PipelineRequest 对象指针的中转通道，用于异步处理时 PipelineRequest 对象指针的维护。

evtbus

event 总线通道，该通道可以接受任何类型的对象 interface{}。处理 Killevent 和 zk 发出的 Topoevent。能处理的事件见“产生 Action 事件”部分

top

和 zookeeper 交互的封装类。Zk 地址和 productName 通过配置读取。

pipeConns

维护一个字典，key 为 redis-server-master 字符串，value 为 taskrunner。一个 redis-master 对应一个 taskrunner。当执行 action 时，要关闭所有的 taskrunner，action 处理完毕后，重新依据 redis-master 的个数创建新的 taskrunner。

bufferedReq

bufferedReq 与 reqCh 的结合使用

Taskrunner

功能

对目标 redis-master 的操作进行封装，内部处理命令的执行和响应。

有两个线程 writeloop 和 readloop。循环处理，分别代表向目标 redis-master 执行命令，和从目标 redis-master 读取响应。

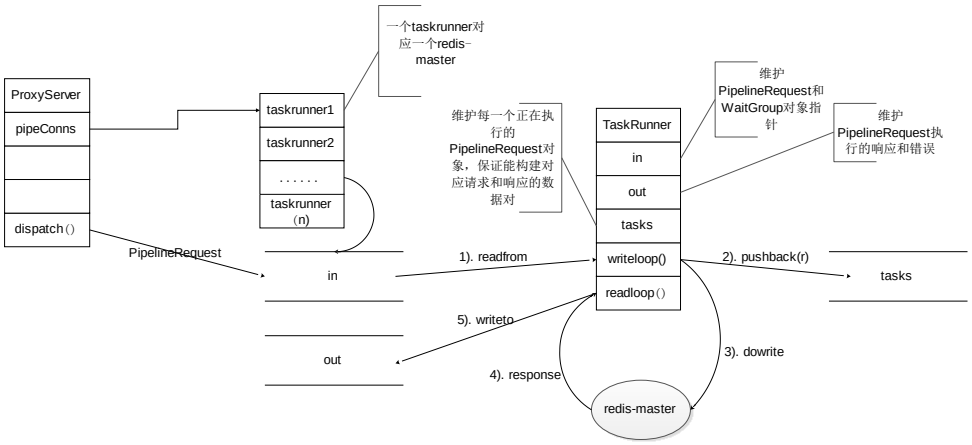
通过维护三个 channel 通道来实现三个先入先出队列。

in 维护 PipelineRequest 和 WaitGroup 对象指针。如果 in 队列不空，从中读取执行数据后执行 processTask。

out 维护所有命令的执行结果或错误信息。如果 out 队列不空，从中读取数据后，执行 handleResponse 处理响应。

tasks 维护 PipelineRequest 的执行顺序。每一个请求必定对应一个响应，该队列维护了响应和请求的命令的对应关系。在执行命令钱(dowrite)，将 PipelineRequest 对象放入 tasks。构建响应时，先从 tasks 取出 request，再根据当前的 resp 来成对的构建 PipelineResponse 对象。

流程图



主要成员功能

成员 in

作为一个 channel，是一个先入先出的队列，内部维护 PipelineRequest 对象指针或 WaitGroup 对象指针。如下两个方法，分别向该队列插入这两种类型的对象指针。Taskrunner 对象内部消费 in 队列，会对这两种不同的类型分别进行处理

router.go

插入 PipelineRequest 对象指针

```
func (s *Server) dispatch(r *PipelineRequest) {
    s.handleMigrateState(r.slotIdx, r.keys[0])
    tr, ok := s.pipeConns[s.slots[r.slotIdx].dst.Master()]
    if !ok {
        //try recreate taskrunner
        if err := s.createTaskRunner(s.slots[r.slotIdx]); err != nil {
            r.backQ <- &PipelineResponse{ctx: r, resp: nil, err: err}
            return
        }

        tr = s.pipeConns[s.slots[r.slotIdx].dst.Master()]
    }
    tr.in <- r

}
```

key是目标redis-master字符串

每一个PipelineRequest被加入taskrunner的 in 通道

插入 WaitGroup 对象指针

```
func (s *Server) stopTaskRunners() {
    wg := &sync.WaitGroup{}
    log.Warning("taskrunner count", len(s.pipeConns))
    wg.Add(len(s.pipeConns))
    for _, tr := range s.pipeConns {
        tr.in <- wg
    }
    wg.Wait()

    //remove all
    for k, _ := range s.pipeConns {
        delete(s.pipeConns, k)
    }
}
```

每一个taskrunner，in传递wg

等待所有taskrunner关闭

taskrunner.go

taskrunner 内部消费 in 队列

```
func (tr *taskRunner) writeloop() {
    var err error
    for {
        if tr.closed && tr.tasks.Len() == 0 {
            log.Warning("exit taskrunner", tr.redisAddr)
            tr.wgClose.Done()
            tr.c.Close()
            return
        }

        if err != nil { //clean up
            err = tr.tryRecover(err)
            if err != nil {
                continue
            }
        }

        select {
        case t := <-tr.in:
            err = tr.processTask(t)
        case resp := <-tr.out:
            err = tr.handleResponse(resp)
        }
    }
}
```

独立线程执行该方法，处理向目标redis-master发起的任何命令操作

消费tr.in队列，t类型可能为PipelineRequest或WaitGroup对象指针

消费 in 队列时，对类型进行判断

```
func (tr *taskRunner) processTask(t interface{}) error {
    switch t.(type) {
    case *PipelineRequest:
        r := t.(*PipelineRequest)
        var flush bool
        if len(tr.in) == 0 { //force flush
            flush = true
        }

        return tr.handleTask(r, flush)
    case *sync.WaitGroup: //close taskrunner
        err := tr.handleTask(nil, true) //flush
        //get all response for out going request
        tr.getOutgoingResponse()
        tr.closed = true
        tr.wgClose = t.(*sync.WaitGroup)
        return err
    }

    return nil
}
```

成员 out

记录每一个命令执行的正常响应或失败响应。

```
func (tr *taskRunner) readloop() {
    for {
        resp, err := parser.Parse(tr.c.BufioReader())
        if err != nil {
            tr.out <- err
            return
        }

        tr.out <- resp
    }
}
```

向out队列插入error

向out队列插入正常的响应

成员 tasks

维护 PipelineRequest 的执行顺序。在构建 PipelineResponse 时，保证请求和响应是成对出现的。

插入 tasks 队列

```
func (tr *taskRunner) handleTask(r *PipelineRequest, flush bool) error {
    if r == nil && flush { //just flush
        return tr.c.Flush()
    }

    tr.tasks.PushBack(r)        在执行dowrite前，将PipelineRequest加入tasks

    return errors.Trace(tr.dowrite(r, flush))
}
```

从 tasks 队列中移除条目

```
func (tr *taskRunner) handleResponse(e interface{}) error {
    switch e.(type) {
    case error:
        return e.(error)
    case *parser.Resp:
        resp := e.(*parser.Resp)
        e := tr.tasks.Front()
        req := e.Value.(*PipelineRequest)
        req.backQ <- &PipelineResponse{ctx: req, resp: resp, err: nil}
        tr.tasks.Remove(e)          成功构建PipelineResponse中的请求和响应对后，从tasks中移除该PipelineRequest对象条目
        return nil
    }

    return nil
}
```

附录 B-常见问题列表

Codis-Redis 改动

slots 管理以及迁移指令：

slotsinfo [start] [count] 获取 redis 中 slot 的个数以及每个 slot 的大小

slotsdel slot1 [slot2 …] 命令说明：删除 redis 中若干 slot 下的全部 key-value

slotsmgrtslot 随机在某个 slot 下迁移一个 key-value 到目标机器

slotsmgrtone 将指定的 key-value 迁移到目标机

slotsmgrttagslot 随机在某个 slot 下选择一个 key，并将与之有相同 tag 的 key-value 对全部迁移到目标机

slotsmgrttagone 将与指定 key 具有相同 tag 的所有 key-value 对迁移到目标机

slotshashkey key1 [key2 …] 计算并返回给定 key 的 slot 序号

Fence 节点作用（保证 zk 路由表完整性）

正常退出 proxy

删除 fence 下面的地址/zk/codis/db_test/fence

删除代理下面的信息/zk/codis/db_test/proxy

异常退出 proxy(kill -9 pid)

/zk/codis/db_test/proxy/proxy_1 删除

/zk/codis/db_test/fence/lidaohang-virtual-machine:19000 下面不会删除,保留

```
// set action receivers
proxies, err := ProxyList(zkConn, productName, func(p *ProxyInfo) bool {
    return p.State == PROXY_STATE_ONLINE
})
if err != nil {
    return errors.Trace(err)
}
if needConfirm {
    // do fencing here, make sure 'offline' proxies are really offline
    // now we only check whether the proxy lists are match
    fenceProxies, err := GetFenceProxyMap(zkConn, productName)
    if err != nil {
        return errors.Trace(err)
    }
    for _, proxy := range proxies {
        delete(fenceProxies, proxy.Addr)
    }
    if len(fenceProxies) > 0 {
        errMsg := bytes.NewBufferString("Some proxies may not stop cleanly:")
        for k, _ := range fenceProxies {
            errMsg.WriteString(" ")
            errMsg.WriteString(k)
        }
        return errors.New(errMsg.String())
    }
}
```

codis-proxy 超时机制

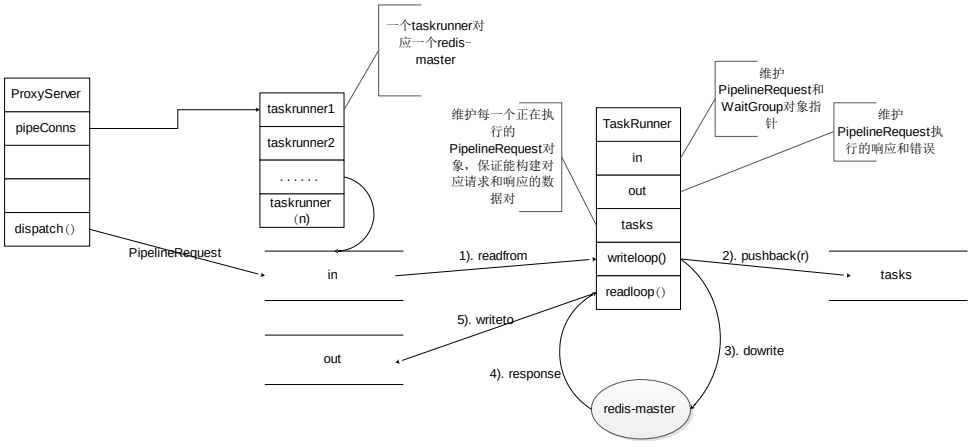
golang　net.Conn 接口中的 SetReadDeadline

```go
func (c *Conn) recvLoop(conn net.Conn) error {
    buf := make([]byte, bufferSize)
    for {
        // package length
        conn.SetReadDeadline(time.Now().Add(c.recvTimeout))
        _, err := io.ReadFull(conn, buf[:4])
        if err != nil {
            return err
        }

        blen := int(binary.BigEndian.Uint32(buf[:4]))
        if cap(buf) < blen {
            buf = make([]byte, blen)
        }

        _, err = io.ReadFull(conn, buf[:blen])
        conn.SetReadDeadline(time.Time{})
        if err != nil {
            return err
        }
```

proxy 作为反向代理执行转发命令是如何实现的?效率如何?



每加入一台 proxy，如何对使用者做到请求透明？使用者使用时一台 proxy 报错时，是否有重试机制?

proxy 只是通过 master 进行读取和写入，是否有提升的办法?为什么这么设计?

当前 codis-proxy 实现的副本协议或流程：

a) 未实现复杂副本协议或者可以说只要成功写入 master 即返回
b) master 和 slave 间的同步依靠 redis 本身的 master-slave 机制
c) 同步会有延时，只能从 master 读取，来保证数据强一致性
d) master 故障或宕机后，codis-ha 选举新的 master 这段期间内，服务不可用
e) codis-porxy 在自身实力中维护元数据信息，zk 负责持久化。
f) 只要 proxy 上的元数据没有修改（未被通知），proxy 就可以放心使用
g) 当元数据被修改，zk 负责通知到所有 proxy，proxy 更新自己维护的元数据
h) 这是一种典型的 lease 实现方式，会存在服务不可用的阶段

缺点：

不能充分利用多副本实例，master 可能会成为瓶颈

优化：

a). 如果不要求强一致性，最简单，可以从 master 与多个 slave 随机读取。无法强保证用户读取数据的单调一致性或强一致性

b). 如果要求单调一致性或强一致性。需要实现副本协议，在写入和读取时处理

(1). 简单的方式只读取和写入 master，即现在的方式

(2). 在读取和写入时，要按照副本协议来实现。

某个 servergroup 中 master 故障，codis-ha 如何工作?

参见《HA 实现方式》