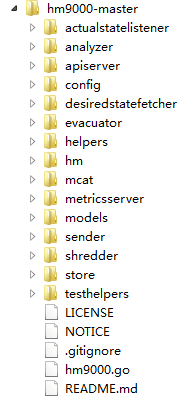
Hm9000分析

1. **代码组织**
   1. **代码子模块**



* 1. **重要文件的介绍**

**hm9000 and hm**:顶层是hm9000 CLI。 hm包内含了CLI逻辑以保持根目录更干净，其他组件在这里实例化，处理依赖以及运行。

**acutalstatelistener**:提供了一个简单的监听守护进程，监控NATS流以得到app心跳。它会为每一个心跳的app在store的 /actual/INSTANCE\_GUID 下产生一个条目。它还在/actual-fresh 下维持着一个FreshnessTimestamp使其他组件知道他们是否可以相信在/actual 下的信息。

**analyser:** Analyzer起来以后会分析实际与期望状态，并把待处理的start和stop消息放进store中。如果一条start或者stop消息已经在store中了，analyzer 将不会覆盖它。

**Apiserver:** Apiserver 响应 NATS的app.state消息，并且允许其他的CloudFoundry组件去获取有关任意应用的信息。

**desiredstatefecher**: Desiredstatefetcher向cloud controller请求期望状态。它透明地管理了从NATS获取认证信息和对bulk api 终端制作批量的http请求。 期望状态储存在`/desired/APP\_GUID-APP\_VERSION下。

**evacuator:** evacuator响应 NATS的droplet.exited消息。如果一个应用存在是因为evacuator通过NATS送来一条起始消息。Evacuator在deterministic evacuations时不是必需的，不过它可以用来提供保持与旧DEA的向后兼容性。

**metricsserver:** Metricsserver 注册CF collector并通过一个/varz 终端收集和提供指标。如果实际状态或期望状态其中之一是不fresh的那么所有这些的指标都会被赋值为-1。

**sender:** Sender周期性运行，它通过NATS把待处理消息从store中拉取出来并发送。Sender会在发送这些消息前验证它们是应该被发送的消息。Sender还负责控制消息通过NATS发送的比率。

**Shredder:** shredder 会从store中去除旧的/混乱的/不需要的数据,包括去除旧的store架构版本。

**config：**支持包, 解析config.json 的配置。 组件通常由hm CLI给出配置实例。

**helpers**：支持包，内含许多支持功能

**models**：支持包, 封装了发送/接收的NATS/ HTTP的各种JSON的结构。简单的序列化/反序列化行为被绑定到这些结构。

**store**：支持包, 位于低级storeadapter的顶部，并提供各种HM9000组件到store的高级别访问。

**testhelpers:**内含大量测试支持包。它们包括从简单的虚拟对象到用于在集成测试中模拟其他CloudFoundry组件的综合库。

* 1. **Hm9000的功能**

Fetching desired state – 会连接到CC，获取期望状态放入store中然后退出

Listening for actual state – 监听NATS获得心跳并放入store中

Analyzing the desired and actual state – 比较期望与实际状态并提交开始与停止消息

Sending start and stop messages – 评估待处理开始和停止消息并通过NATS发布

Serving metrics (varz) – 由collector注册并提供一个带数据的 /varz 终端

Serving API – 通过NATS提供对app.state请求的响应

Evacuator – 监听droplet.exited消息并对任何撤销中的droplet发送开始消息

Shredder – 周期性清理store –移除任何空目录，初始为1小时一次

Dumping the contents of the store – 转储store中的所有内容到stdout

1. **hm9000原理配置及架构**

**2.1 hm9000原理**

**2.1.1** hm9000组件工作需要获取的两种状态

- desired state: 期望的状态，哪些apps应该是running状态，哪些instances应该是running状态，这些信息是通过http协议从CC中发送过来

- actual state: 实际状态，哪些instances实际上是running状态，这些信息通过via Nats和DEAs中接收，每个DEA节点会周期性的发送heartbeat心跳来确认running应用

**2.1.2** hm9000存储desired state和actual state在etcd中，有了这两种状态，hm9000可以决定是否启动或者停止

一个实例.这个信息通过Nats发送到CC，最后CC通过Nats发送消息到DEA决定是否启动或者停止一个实例

**2.1.3** “freshness”的概念

- 当hm9000可以与NATS通信并且可以周期性的从DEA节点中接收心跳并且可以正确的把actual state存储在store中，那么这个actual state是我们期望的“fresh”状态，如果它们中任何一个环节出现异常（NATS/no DEA heartbeats/etcd writes fail），这个actual state都将标记为“not fresh”,这时候hm9000将停止任何会话（交互）动作

- 当hm9000从CC中下载desired state成功（without timeout）并且可以正确存储在store中时，那么这个disired state是我们期望的"fresh"状态，同actual state一样任何一个环节出现异常都将导致hm9000工作异常，即上面所述的“not fresh”

**2.1.4** hm9000的一个重要职责就是定时分析应用和instance的状态。Hm9000周期性地用analyzer分析store中actual state和desired state，通过发送start和stop信息使actual state与desired state匹配，消息储存在store中。 再通过sender周期性从store中提取消息并通过NATS发布。

hm9000通过actualstatelistner来监听NATS以从DEA节点中接收心跳，并储存在store中，心跳中包括APPstate和instance state。Desiredstatefetcher则用来从cc获取desired state并储存在store中。

例如如果发现有instance缺失或者crash，那么会认为此instance需要重启。这时该instance的ID会被记录下来，然后HM将会试图启动对应App的一个新instance。当然，HM是只读的，所以启动instance的实际工作应该由Cloud Controller来完成，而HM只需要组装好一个JSON格式的消息内容，然后使用NATS来发布这个主题为cloudcontrollers.hm.requests的消息就可以了。Cloud Controller在收到这个消息后就会使用发布者传来的消息内容来启动一个新的instance。

**2.2 部分hm9000配置**

heartbeat\_period\_in\_seconds ：在hm9000配置中几乎所有的可配置时间常量都明确的用这个基本时间单位来表示 --- 以秒计的心跳之间的时间间隔。这应该与DEA中明确的值相匹配，一般设为10秒。

heartbeat\_ttl\_in\_heartbeats ：传入的心跳与一个TTL一起储存在store中。当这个TTL使心跳相关实例过期时被认为是"gone missing"。这个TTL设为3个心跳周期。

actual\_freshness\_ttl\_in\_heartbeats ：这个常量为两个目的服务。它是在store中实际状态freshness key的TTL。这个TTL设为3个心跳周期。

grace\_period\_in\_heartbeats ：在schedule 消息的时候使用。比如我们用这个grace period来延迟开始消息的发送时间，以此给missing instance一个机会在发送开始消息之前去start up。设为3个心跳周期。

desired\_freshness\_ttl\_in\_heartbeats ：期望状态freshness的TTL。设为12 heartbeats。如果不是12那么期望状态将会被认为是不新鲜的。

store\_max\_concurrent\_requests ：每个组件可向store提起的并发请求最大数量。设为30。

sender\_message\_limit ：每次调用sender应该发送消息数目的最大值，设为30。

sender\_polling\_interval\_in\_heartbeats ：心跳中的时间间隔单位，设为30。

sender\_timeout\_in\_heartbeats ：每次sender调用的心跳超时单位。如果一次sender调用用时超过这个，那么hm9000 send –poll 命令将会失败。设为10。

fetcher\_polling\_interval\_in\_heartbeats ：在使用hm9000 analyze –poll命令时desired state fetcher调用之间的心跳时间周期单位，设为6。

fetcher\_timeout\_in\_heartbeats ：每次desired state fetcher调用时的心跳超时单位。如果一次调用耗时比这更长，那么hm9000 fetch\_desired –poll命令将会失败。设为60。

analyzer\_polling\_interval\_in\_heartbeats ：在使用hm9000 analyze --poll命令时analyzer调用之间的心跳时间周期单位，设为1。

analyzer\_timeout\_in\_heartbeats ：每次analyzer调用时的心跳超时单位。如果一次调用耗时比这更长，那么hm9000 analyze –poll命令将会失败。设为10。

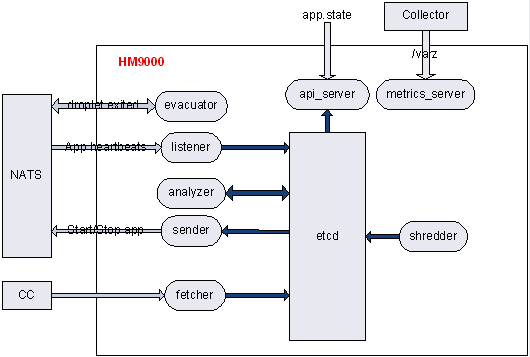
shredder\_polling\_interval\_in\_heartbeats ：在使用hm9000 shred --poll命令时shredder调用之间的心跳时间周期单位，设为360。

shredder\_timeout\_in\_heartbeats ：每次shredder调用时的心跳超时单位。如果一次调用耗时比这更长，那么hm9000 shred –poll命令将会失败。设为6。

desired\_state\_batch\_size : 当从cc获取期望状态信息的batch 尺寸，设为500。

fetcher\_network\_timeout\_in\_seconds : 每一个对CC的API呼叫必须在这个timeout中成功。设为10秒。

**2.3 hm9000架构**



1. **功能实现代码分析**

**3.1 Fetching desired state**

**3.1.1 desired\_state\_fetcher 结构**

type DesiredStateFetcher struct {

config \*config.Config

httpClient httpclient.HttpClient

store store.Store

metricsAccountant metricsaccountant.MetricsAccountant

timeProvider timeprovider.TimeProvider

cache map[string]models.DesiredAppState

logger logger.Logger

}

**3.1.2 流程**

***涉及文件***：hm9000/ fetch\_desired\_state.go/ desired\_ state\_fetcher.go

1) 从hm9000获取*fetch\_desired --config=./local\_config.json* 命令

hm.FetchDesiredState(logger, conf, c.Bool("poll"))

2) 如果有 “poll”在命令中将会启动一个获取守护进程来进行周期性获取,否则之获取一次

func FetchDesiredState(l logger.Logger, conf \*config.Config, poll bool) {

if poll { ……

err := Daemonize("Fetcher", func() error {

return fetchDesiredState(l, conf, store)

}, conf.FetcherPollingInterval(), conf.FetcherTimeout(), l, adapter)

……

else {

err := fetchDesiredState(l, conf, store)

3) 起动一个新的state fetcher 并获取状态

fetcher := desiredstatefetcher.New

# 创建一个名为resultChan的channel去传递获取结果

resultChan := make(chan desiredstatefetcher.DesiredStateFetcherResult, 1)

#获取状态并传递进result中，如果结果成功那么储存成功结果信息进日志中，否则储存错误信息

fetcher.Fetch(resultChan)

result := <-resultChan

4) The fetching process

#获取授权信息以启动fetchbatch

fetcher.fetchBatch(authInfo.Encode(), initialBulkToken, 0, resultChan)

#fetch batch function

func (fetcher \*DesiredStateFetcher) fetchBatch(authorization string, token string, numResults int, resultChan chan DesiredStateFetcherResult) {

# 生成一个URL请求

req, err := http.NewRequest("GET", fetcher.bulkURL(fetcher.config.DesiredStateBatchSize, token), nil)

#操作这个http请求并返回响应

fetcher.httpClient.Do(req, func(resp \*http.Response, err error)

# 检查StatusCode是否被授权并且在ok状态

if resp.StatusCode == http.StatusUnauthorized{…..}

if resp.StatusCode != http.StatusOK{….}

#读取response body

body, err := ioutil.ReadAll(resp.Body)

#解析 HTTP response body JSON

response, err := NewDesiredStateServerResponse(body)

# 同步期望状态到store

tSync := time.Now()

err = fetcher.syncStore()

fetcher.metricsAccountant.TrackDesiredStateSyncTime(time.Since(tSync))

#传递结果进resultChan

resultChan <- DesiredStateFetcherResult{Success: true, NumResults: numResults}

#cache the response

fetcher.cacheResponse(response)

**3.2 Listening for actual state**

**3.2.1 actualstatelistener 结构：**

type ActualStateListener struct {

logger logger.Logger

config \*config.Config

messageBus yagnats.NATSClient

store store.Store

timeProvider timeprovider.TimeProvider

storeUsageTracker metricsaccountant.UsageTracker

metricsAccountant metricsaccountant.MetricsAccountant

heartbeatsToSave []models.Heartbeat

totalReceivedHeartbeats int

totalSavedHeartbeats int

lastReceivedHeartbeat time.Time

heartbeatMutex \*sync.Mutex

}

**3.2.2 流程：**

**涉及文件：**hm9000.go/ start\_listening\_for \_actual.go/ actual\_state\_listener.go

1). 从hm9000 cli获取*hm9000 listen --config=./local\_config.json* 命令

hm.StartListeningForActual(logger, conf)

2). Listener starter:

listener := actualstatelistener.New(conf,

messageBus,

store,

usageTracker,

metricsaccountant.New(store),

buildTimeProvider(l),

l,

)

listener.Start()

3) 开始监听

func (listener \*ActualStateListener) Start() {

#订阅 dea.heartbeat subject 和the dea.advertise subject

listener.messageBus.Subscribe("dea.advertise", func(message \*yagnats.Message) {….}

listener.messageBus.Subscribe("dea.heartbeat", func(message \*yagnats.Message) {….}

#跟踪接收到的心跳

listener.metricsAccountant.TrackReceivedHeartbeats(totalReceivedHeartbeats)

4)同步心跳

func (listener \*ActualStateListener) syncHeartbeats() {

#储存心跳

err := listener.store.SyncHeartbeats(heartbeatsToSave...)

#如果储存不成功那么撤回actual freshness

if err != nil {

listener.logger.Error("Could not put instance heartbeats in store:", err)

listener.store.RevokeActualFreshness()

}

#如果储存成成功那么bump freshness, 除非耗时太长。储存心跳和duration信息进logger

else {

dt := time.Since(t)

if dt < listener.config.ListenerHeartbeatSyncInterval() {

listener.bumpFreshness()

} else {

listener.logger.Info("Save took too long. Not bumping freshness.")

}

listener.logger.Info("Saved Heartbeats", map[string]string{

"Heartbeats to Save": strconv.Itoa(len(heartbeatsToSave)),

"Duration": time.Since(t).String(),

})

#跟踪储存的心跳

listener.metricsAccountant.TrackSavedHeartbeats(totalSavedHeartbeats)

5)跟踪并检测store的使用情况

listener.storeUsageTracker.StartTrackingUsage()

listener.measureStoreUsage()

**3.3 Analyzing the desired and actual state**

**3.3.1 Analyzer结构**

type Analyzer struct {

store store.Store

logger logger.Logger

timeProvider timeprovider.TimeProvider

conf \*config.Config}

**3.3.2 流程**

涉及文件：hm9000.go/ analyzer.go/analyze.go

1. 从hm9000 cli获取*hm9000 analyze --config=./local\_config.json* 命令

hm.Analyze(logger, conf, c.Bool("poll"))

1. 如果有 “poll”在命令中将会启动一个分析守护进程来进行周期性分析,否则只分析一次

func Analyze(l logger.Logger, conf \*config.Config, poll bool){

if poll {

……

err := Daemonize("Analyzer", func() error {

return analyze(l, conf, store)

}, conf.AnalyzerPollingInterval(), conf.AnalyzerTimeout(), l, adapter)

……

else {

err := analyze(l, conf, store)

……

1. 创建一个新的analyzer并开始分析

func analyze(l logger.Logger, conf \*config.Config, store store.Store) error {

……

analyzer := analyzer.New(store, buildTimeProvider(l), l, conf)

err := analyzer.Analyze()

4）从store获取信息并返回错误信息

func (analyzer \*Analyzer) Analyze() error {

#检查store是否fresh

err := analyzer.store.VerifyFreshness(analyzer.timeProvider.Time())

#获取应用状态并储存在apps中

apps, err := analyzer.store.GetApps()

# 获取应用待处理的开始信息并储存进“existingPendingStartMessages”

existingPendingStartMessages, err := analyzer.store.GetPendingStartMessages()

#获取应用待处理的停止信息并储存进“existingPendingStopMessages”

existingPendingStopMessages, err := analyzer.store.GetPendingStopMessages()

# 用前面所获取的信息去启动一个新的 newAppAnalyzer 并启动 analyzeApp(). Append the startMessage to allStartMessages, stopMessage to allStopMessages, crashCounts to allCrashCounts.

allStartMessages := []models.PendingStartMessage{}

allStopMessages := []models.PendingStopMessage{}

allCrashCounts := []models.CrashCount{}

for \_, app := range apps {

startMessages, stopMessages, crashCounts := newAppAnalyzer(app, analyzer.timeProvider.Time(), existingPendingStartMessages, existingPendingStopMessages, analyzer.logger, analyzer.conf).analyzeApp()

for \_, startMessage := range startMessages {

allStartMessages = append(allStartMessages, startMessage)

}

for \_, stopMessage := range stopMessages {

allStopMessages = append(allStopMessages, stopMessage)

}

allCrashCounts = append(allCrashCounts, crashCounts...)

}

#储存crash counts, pending start messages，pending stop messages到store, 如果有错则储存并返回错误

err = analyzer.store.SaveCrashCounts(allCrashCounts...)

err = analyzer.store.SavePendingStartMessages(allStartMessages...)

err = analyzer.store.SavePendingStopMessages(allStopMessages...)

**3.4 Sending start and stop messages**

**3.4.1 sender 结构**

type Sender struct {

store store.Store

conf \*config.Config

logger logger.Logger

apps map[string]\*models.App

messageBus yagnats.NATSClient

timeProvider timeprovider.TimeProvider

numberOfStartMessagesSent int

sentStartMessages []models.PendingStartMessage

startMessagesToSave []models.PendingStartMessage

startMessagesToDelete []models.PendingStartMessage

sentStopMessages []models.PendingStopMessage

stopMessagesToSave []models.PendingStopMessage

stopMessagesToDelete []models.PendingStopMessage

metricsAccountant metricsaccountant.MetricsAccountant

didSucceed bool

}

**3.4.2 流程**

**涉及文件**：hm9000.go/ sender.go/ send.go

1. 从 hm9000 cli获取*hm9000 send --config=./local\_config.json* 命令

hm.Send(logger, conf, c.Bool("poll"))

1. 如果有 “poll”在命令中将会启动一个发送守护进程来进行周期性发送,否则只发送一次

if poll {

err := Daemonize("Sender", func() error {

return send(l, conf, messageBus, store)

}, conf.SenderPollingInterval(), conf.SenderTimeout(), l, adapter)

1. 起一个新的sender

sender := sender.New(store, metricsaccountant.New(store), conf, messageBus, buildTimeProvider(l), l)

1. 发送消息

#检查store是否fresh

err := sender.store.VerifyFreshness(sender.timeProvider.Time())

# 获取应用以及待处理开始停止信息

pendingStartMessages, err := sender.store.GetPendingStartMessages()

pendingStopMessages, err := sender.store.GetPendingStopMessages()

sender.apps, err = sender.store.GetApps()

1. 发送开始信息（停止信息同理，略）

sender.sendStartMessages(pendingStartMessages)

func (sender \*Sender) sendStartMessages(startMessages map[string]models.PendingStartMessage) {

#将开始信息按优先级进行分类

sortedStartMessages := models.SortStartMessagesByPriority(startMessages)

#检查是否到发送时间，到了则发送，否则信息过期

for \_, startMessage := range sortedStartMessages {

if startMessage.IsTimeToSend(sender.timeProvider.Time()) {

sender.sendStartMessage(startMessage)

} else if startMessage.IsExpired(sender.timeProvider.Time()) {

sender.queueStartMessageForDeletion(startMessage, "expired start message")

}

}

5）发送信息

#检查信息是否应该被发送

messageToSend, shouldSend := sender.startMessageToSend(startMessage)

#如果信息是应该被发送的并且已发送信息量小于sender每次调用的应该发送的消息数最大值（30）则发送信息

if shouldSend {

if sender.numberOfStartMessagesSent < sender.conf.SenderMessageLimit {

sender.logger.Info("Sending message", startMessage.LogDescription())

err := sender.messageBus.Publish(sender.conf.SenderNatsStartSubject, messageToSend.ToJSON())

#如果startMessage.KeepAlive == 0 则将信息放入待删除队列，否则标记为已发送信息

if startMessage.KeepAlive == 0 {

sender.queueStartMessageForDeletion(startMessage, "a sent start message with no keep alive")

} else {

sender.markStartMessageSent(startMessage)

}

#如果信息不应该被发送则该消息不会被发送直接加入待删除队列

else {

sender.queueStartMessageForDeletion(startMessage, "start message that will not be sent")

6）删除待删除消息（stop同理）

err = sender.store.DeletePendingStartMessages(sender.startMessagesToDelete...)

if err != nil {

sender.logger.Error("Failed to delete start messages", err)

sender.didSucceed = false

}

7）如果发送不成功，新建检查日志信息

if !sender.didSucceed {

return errors.New("Sender failed. See logs for details.")

}

**3.5 Serving metrics (varz)**

**3.5.1 MetricsServer结构**

type MetricsServer struct {

registrar CollectorRegistrar

steno \*gosteno.Logger

store store.Store

logger logger.Logger

timeProvider timeprovider.TimeProvider

config \*config.Config

metricsAccountant metricsaccountant.MetricsAccountant

}

**3.5.2 流程**

**涉及文件**：hm9000.go / serve\_metrics.go / metrics\_server.go

1) 从 hm9000 cli获取*hm9000 serve\_metrics --config=./local\_config.json* 命令

hm.ServeMetrics(steno, logger, conf)

2) 创建 metrics server.

func ServeMetrics(steno \*gosteno.Logger, l logger.Logger, conf \*config.Config) {

metricsServer := metricsserver.New(

……)

err := metricsServer.Start()

1. 启动 metrics server

func (s \*MetricsServer) Start() error {

#新建 cf 组件

component, err := cfcomponent.NewComponent(……)

#储存 components信息进logger

s.logger.Info("Serving Metrics", map[string]string{……}

#提供一个有数据的 /varz 终点

go component.StartMonitoringEndpoints()

# 注册register collector 并返回错误

err = s.registrar.RegisterWithCollector(component)

return err

**3.6 Serving API**

**3.6.1 apiserver 结构**

type ApiServer struct {

messageBus yagnats.NATSClient

store store.Store

timeProvider timeprovider.TimeProvider

logger logger.Logger

}

**3.6.2 流程**

**涉及文件：**hm9000.go/ serve\_api.go/ apisever.go

1. 从 hm9000 cli 获取*hm9000 serve\_api --config=./local\_config.json*命令

hm.ServeAPI(logger, conf)

1. 通过NATS提供API

func ServeAPI(l logger.Logger, conf \*config.Config) {

……

apiServer.Listen()

1. 通过NATS提供对应用状态请求的响应

func (server \*ApiServer) Listen() {

#向NATS订阅应用状态消息

server.messageBus.SubscribeWithQueue("app.state", "hm9000", func(message \*yagnats.Message) {

……

#新建一个request并储存解析后的JSON加密数据

var request AppStateRequest

err = json.Unmarshal([]byte(message.Payload), &request)

#检查store是否fresh,获取app model

err = server.store.VerifyFreshness(server.timeProvider.Time())

app, err := server.store.GetApp(request.AppGuid, request.AppVersion)

# 把response赋值为序列化的应用model数据

response = app.ToJSON()

#在之前的过程中如果有任何错误则发布空response并在logger中储存错误。如无错则发布response并在logger中储存成功信息。两种情况都要取得payload和elapsed time

if err != nil {

server.messageBus.Publish(message.ReplyTo, []byte("{}"))

server.logger.Error("Failed to handle app.state request", err, map[string]string{

"payload": string(message.Payload),

"elapsed time": fmt.Sprintf("%s", time.Since(t)),

})} else {

server.messageBus.Publish(message.ReplyTo, response)

server.logger.Info("Responded succesfully to app.state request", map[string]string{

"payload": string(message.Payload),

"elapsed time": fmt.Sprintf("%s", time.Since(t)),

})

**3.7 Evacuator**

**3.7.1 Evacuator结构**

type Evacuator struct {

ssageBus yagnats.NATSClient

ore store.Store

meProvider timeprovider.TimeProvider

nfig \*config.Config

gger logger.Logger

}

**3.7.2 流程**

**涉及文件：**hm9000.go/ start\_evacuator.go/ evacuator.go

1. 从 hm9000 cli 获取*hm9000 evacuator --config=./local\_config.json*命令

hm.StartEvacuator(logger, conf)

1. 新建 evacuator 并开始监听

func StartEvacuator(l logger.Logger, conf \*config.Config) {

evacuator := evacuatorpackage.New(messageBus, store, buildTimeProvider(l), conf, l)

evacuator.Listen()

1. 监听 DEA evacuations

func (e \*Evacuator) Listen() {

#向NATS订阅droplet.exit消息，储存消息在dropletExited中，如果头错误则储存错误在logger中

e.messageBus.Subscribe("droplet.exited", func(message \*yagnats.Message) {

dropletExited, err := models.NewDropletExitedFromJSON([]byte(message.Payload))

……

e.handleExited(dropletExited)

4）向任何 evacuating droplets发送开始消息

#当droplets退出原因为DEAShutdown 和 DEAEvacuation时，为droplets安排并发送新的开始消息

func (e \*Evacuator) handleExited(exited models.DropletExited) {

switch exited.Reason {

case models.DropletExitedReasonDEAShutdown, models.DropletExitedReasonDEAEvacuation:

startMessage := models.NewPendingStartMessage(

……

startMessage.SkipVerification = true

e.logger.Info("Scheduling start message for droplet.exited message", startMessage.LogDescription(), exited.LogDescription())

e.store.SavePendingStartMessages(startMessage)

**3.8 Shredder**

**3.8.1 shredder结构**

type Shredder struct {

store storepackage.Store

}

**3.8.2 流程**

**涉及文件：**hm9000.go/ shred.go/ shredder.go/ compact.go

1. 从 hm9000 cli 获取*hm9000 shred --config=./local\_config.json*命令

hm.Shred(logger, conf, c.Bool("poll"))

1. 如果有 “poll”在命令中将周期性发送消息。

if poll {

……

err := Daemonize("Shredder", func() error {

return shred(l, store)

}, conf.ShredderPollingInterval(), conf.ShredderTimeout(), l, adapter)

……

else {

err := shred(l, store)

……

1. 新建一个 shredder

func shred(l logger.Logger, store store.Store) error {

l.Info("Shredding Store")

theShredder := shredder.New(store)

return theShredder.Shred()

}

1. shred

func (s \*Shredder) Shred() error {

return s.store.Compact()

}

1. 删除空目录，旧架构版本和无版本数据。

func (store \*RealStore) Compact() error {

err := store.deleteOldSchemaVersionsAndUnversionedData()

……

err = store.deleteEmptyDirectories()

**3.9 Dumping the contents of the store**

**3.9.1 流程**

**涉及文件**：hm9000.go/ store.go

1. 从 hm9000 cli获取*hm9000 dump --config=./local\_config.json* 命令

hm.Dump(logger, conf, c.Bool("raw"))

1. 如果有 raw 在命令中那么 dumpraw 否则dumpstructured

func Dump(l logger.Logger, conf \*config.Config, raw bool) {

if raw {dumpRaw(l, conf)} else {dumpStructured(l, conf)}}

1. Dumpraw

func dumpRaw(l logger.Logger, conf \*config.Config) {

……

node, err := storeAdapter.ListRecursively("/hm")

……

#遍历store所有node

walk(node, func(node storeadapter.StoreNode) {

#增加由nodes’ keys, ttl 和values组成的条目

entries = append(entries, fmt.Sprintf("%s %s:\n %s", node.Key, ttl, value))

#分类并输出条目

sort.Sort(entries)

for \_, entry := range entries {

fmt.Printf(entry + "\n") }

1. Dumpstructure

func dumpStructured(l logger.Logger, conf \*config.Config) {

#检查store是否fresh, 获取应用model信息 ,待处理开始消息停止消息

err := store.VerifyFreshness(timeProvider.Time())

apps, err := store.GetApps()

starts, err := store.GetPendingStartMessages()

stops, err := store.GetPendingStopMessages()

#开始dumping app

for \_, appKey := range appKeys {

dumpApp(apps[appKey], starts, stops, timeProvider)

}

1. 分类输出APP信息

if app.IsDesired() {

# AppGuid，AppVersion

fmt.Printf("Guid: %s | Version: %s\n", app.AppGuid, app.AppVersion)

# Desired.NumberOfInstances，Desired.State，Desired.PackageState

fmt.Printf(" Desired: [%d] instances, (%s, %s)\n", app.Desired.NumberOfInstances, app.Desired.State, app.Desired.PackageState)

} else {

fmt.Printf(" Desired: NO\n")

}

#输出应用心跳

if len(app.InstanceHeartbeats) == 0 {

fmt.Printf(" Heartbeats: NONE\n")

} else {

fmt.Printf(" Heartbeats:\n")

for \_, heartbeat := range app.InstanceHeartbeats {

fmt.Printf(" [%d %s] %s on %s\n", heartbeat.InstanceIndex, heartbeat.State, heartbeat.InstanceGuid, heartbeat.DeaGuid[0:5])

}

}

#输出 app’s crashcounts

if len(app.CrashCounts) != 0 {

fmt.Printf(" CrashCounts:")

for \_, crashCount := range app.CrashCounts {

fmt.Printf(" [%d]:%d", crashCount.InstanceIndex, crashCount.CrashCount)

}

fmt.Printf("\n")

}

#如有待开始app则输出app开始相关的消息

message = append(message, fmt.Sprintf("[%d]", start.IndexToStart))

message = append(message, fmt.Sprintf("priority:%.2f", start.Priority))

if start.SkipVerification {

message = append(message, "NO VERIFICATION")

}

if start.SentOn != 0 {

message = append(message, "send:SENT")

message = append(message, fmt.Sprintf("delete:%s", time.Unix(start.SentOn+int64(start.KeepAlive), 0).Sub(timeProvider.Time())))

} else {

message = append(message, fmt.Sprintf("send:%s", time.Unix(start.SendOn, 0).Sub(timeProvider.Time())))

}

#如有待停止app则输出app停止相关的消息

message = append(message, stop.InstanceGuid)

if stop.SentOn != 0 {

message = append(message, "send:SENT")

message = append(message, fmt.Sprintf("delete:%s", time.Unix(stop.SentOn+int64(stop.KeepAlive), 0).Sub(timeProvider.Time())))

} else {

message = append(message, fmt.Sprintf("send:%s", time.Unix(stop.SendOn, 0).Sub(timeProvider.Time())))

}