# cmd\alertmanager\main.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\alertmanager\cmd\alertmanager\main.go

pipeline = notify.BuildPipeline(  
 conf.Receivers,  
 tmpl,  
 waitFunc,  
 inhibitor,  
 silences,  
 notificationLog,  
 marker,  
)

## 调用

### config.LoadFile

conf, err := config.LoadFile(\*configFile)

### conf.Receivers

pipeline = notify.BuildPipeline(  
 conf.Receivers,  
 tmpl,  
 waitFunc,  
 inhibitor,  
 silences,  
 notificationLog,  
 marker,  
)

## 问题

### NewDispatcher为什么可以传入pipeline(RoutingStage)

RoutingStage

disp = dispatch.NewDispatcher(alerts, dispatch.NewRoute(conf.Route, nil), pipeline, marker, timeoutFunc)

RoutingStage也实现了Stage接口，那么也就可以把pipeline作为参数stage提供给NewDispatcher

# config\config.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\github.com\prometheus\alertmanager\config\config.go

## config.go.LoadFile

// LoadFile parses the given YAML file into a Config.  
**func** LoadFile(filename string) (\*Config, error) {  
 content, err := ioutil.ReadFile(filename)  
 **if** err != nil {  
 **return** nil, err  
 }  
 cfg, err := Load(string(content))  
 **if** err != nil {  
 **return** nil, err  
 }  
  
 resolveFilepaths(filepath.Dir(filename), cfg)  
 **return** cfg, nil  
}

### 调用

cfg, err := Load(string(content))

## config.go.Load

// Load parses the YAML input s into a Config.  
**func** Load(s string) (\*Config, error) {  
 cfg := &Config{}  
 err := yaml.Unmarshal([]byte(s), cfg)  
 **if** err != nil {  
 **return** nil, err  
 }  
 // Check if we have a root route. We cannot check for it in the  
 // UnmarshalYAML method because it won't be called if the input is empty  
 // (e.g. the config file is empty or only contains whitespace).  
 **if** cfg.Route == nil {  
 **return** nil, errors.New("no route provided in config")  
 }  
  
 **if** cfg.Route.Continue {  
 **return** nil, errors.New("cannot have continue in root route")  
 }  
  
 cfg.original = s  
 **return** cfg, nil  
}

# config\notifiers.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\github.com\prometheus\alertmanager\config\notifiers.go

## Config.struct

// Config is the top-level configuration for Alertmanager's config files.  
**type** Config **struct** {  
 Global \*GlobalConfig `yaml:"global,omitempty" json:"global,omitempty"`  
 Route \*Route `yaml:"route,omitempty" json:"route,omitempty"`  
 InhibitRules []\*InhibitRule `yaml:"inhibit\_rules,omitempty" json:"inhibit\_rules,omitempty"`  
 Receivers []\*Receiver `yaml:"receivers,omitempty" json:"receivers,omitempty"`  
 Templates []string `yaml:"templates" json:"templates"`  
  
 // Catches all undefined fields and must be empty after parsing.  
 XXX **map**[string]**interface**{} `yaml:",inline" json:"-"`  
  
 // original is the input from which the config was parsed.  
 original string  
}

## Route.struct

// A Route is a node that contains definitions of how to handle alerts.  
**type** Route **struct** {  
 Receiver string `yaml:"receiver,omitempty" json:"receiver,omitempty"`  
 GroupBy []model.LabelName `yaml:"group\_by,omitempty" json:"group\_by,omitempty"`  
  
 Match **map**[string]string `yaml:"match,omitempty" json:"match,omitempty"`  
 MatchRE **map**[string]Regexp `yaml:"match\_re,omitempty" json:"match\_re,omitempty"`  
 Continue bool `yaml:"continue,omitempty" json:"continue,omitempty"`  
 Routes []\*Route `yaml:"routes,omitempty" json:"routes,omitempty"`  
  
 GroupWait \*model.Duration `yaml:"group\_wait,omitempty" json:"group\_wait,omitempty"`  
 GroupInterval \*model.Duration `yaml:"group\_interval,omitempty" json:"group\_interval,omitempty"`  
 RepeatInterval \*model.Duration `yaml:"repeat\_interval,omitempty" json:"repeat\_interval,omitempty"`  
  
 // Catches all undefined fields and must be empty after parsing.  
 XXX **map**[string]**interface**{} `yaml:",inline" json:"-"`  
}

### 例子

#### alertmanager/simple.yml

https://github.com/prometheus/alertmanager/blob/master/doc/examples/simple.yml

|  |
| --- |
| global: |
|  | # The smarthost and SMTP sender used for mail notifications. |
|  | smtp\_smarthost: 'localhost:25' |
|  | smtp\_from: 'alertmanager@example.org' |
|  | smtp\_auth\_username: 'alertmanager' |
|  | smtp\_auth\_password: 'password' |
|  | # The auth token for Hipchat. |
|  | hipchat\_auth\_token: '1234556789' |
|  | # Alternative host for Hipchat. |
|  | hipchat\_url: 'https://hipchat.foobar.org/' |
|  |  |
|  | # The directory from which notification templates are read. |
|  | templates: |
|  | - '/etc/alertmanager/template/\*.tmpl' |
|  |  |
|  | # The root route on which each incoming alert enters. |
|  | route: |
|  | # The labels by which incoming alerts are grouped together. For example, |
|  | # multiple alerts coming in for cluster=A and alertname=LatencyHigh would |
|  | # be batched into a single group. |
|  | group\_by: ['alertname', 'cluster', 'service'] |
|  |  |
|  | # When a new group of alerts is created by an incoming alert, wait at |
|  | # least 'group\_wait' to send the initial notification. |
|  | # This way ensures that you get multiple alerts for the same group that start |
|  | # firing shortly after another are batched together on the first |
|  | # notification. |
|  | group\_wait: 30s |
|  |  |
|  | # When the first notification was sent, wait 'group\_interval' to send a batch |
|  | # of new alerts that started firing for that group. |
|  | group\_interval: 5m |
|  |  |
|  | # If an alert has successfully been sent, wait 'repeat\_interval' to |
|  | # resend them. |
|  | repeat\_interval: 3h |
|  |  |
|  | # A default receiver |
|  | receiver: team-X-mails |
|  |  |
|  | # All the above attributes are inherited by all child routes and can |
|  | # overwritten on each. |
|  |  |
|  | # The child route trees. |
|  | routes: |
|  | # This routes performs a regular expression match on alert labels to |
|  | # catch alerts that are related to a list of services. |
|  | - match\_re: |
|  | service: ^(foo1|foo2|baz)$ |
|  | receiver: team-X-mails |
|  | # The service has a sub-route for critical alerts, any alerts |
|  | # that do not match, i.e. severity != critical, fall-back to the |
|  | # parent node and are sent to 'team-X-mails' |
|  | routes: |
|  | - match: |
|  | severity: critical |
|  | receiver: team-X-pager |
|  | - match: |
|  | service: files |
|  | receiver: team-Y-mails |
|  |  |
|  | routes: |
|  | - match: |
|  | severity: critical |
|  | receiver: team-Y-pager |
|  |  |
|  | # This route handles all alerts coming from a database service. If there's |
|  | # no team to handle it, it defaults to the DB team. |
|  | - match: |
|  | service: database |
|  | receiver: team-DB-pager |
|  | # Also group alerts by affected database. |
|  | group\_by: [alertname, cluster, database] |
|  | routes: |
|  | - match: |
|  | owner: team-X |
|  | receiver: team-X-pager |
|  | - match: |
|  | owner: team-Y |
|  | receiver: team-Y-pager |
|  |  |
|  |  |
|  | # Inhibition rules allow to mute a set of alerts given that another alert is |
|  | # firing. |
|  | # We use this to mute any warning-level notifications if the same alert is |
|  | # already critical. |
|  | inhibit\_rules: |
|  | - source\_match: |
|  | severity: 'critical' |
|  | target\_match: |
|  | severity: 'warning' |
|  | # Apply inhibition if the alertname is the same. |
|  | equal: ['alertname', 'cluster', 'service'] |
|  |  |
|  |  |
|  | receivers: |
|  | - name: 'team-X-mails' |
|  | email\_configs: |
|  | - to: 'team-X+alerts@example.org' |
|  |  |
|  | - name: 'team-X-pager' |
|  | email\_configs: |
|  | - to: 'team-X+alerts-critical@example.org' |
|  | pagerduty\_configs: |
|  | - service\_key: <team-X-key> |
|  |  |
|  | - name: 'team-Y-mails' |
|  | email\_configs: |
|  | - to: 'team-Y+alerts@example.org' |
|  |  |
|  | - name: 'team-Y-pager' |
|  | pagerduty\_configs: |
|  | - service\_key: <team-Y-key> |
|  |  |
|  | - name: 'team-DB-pager' |
|  | pagerduty\_configs: |
|  | - service\_key: <team-DB-key> |
|  | - name: 'team-X-hipchat' |
|  | hipchat\_configs: |
|  | - auth\_token: <auth\_token> |
|  | room\_id: 85 |
|  | message\_format: html |
|  | notify: true |

## EmailConfig.struct

// EmailConfig configures notifications via mail.  
**type** EmailConfig **struct** {  
 NotifierConfig `yaml:",inline" json:",inline"`  
  
 // Email address to notify.  
 To string `yaml:"to" json:"to"`  
 From string `yaml:"from" json:"from"`  
 Smarthost string `yaml:"smarthost,omitempty" json:"smarthost,omitempty"`  
 AuthUsername string `yaml:"auth\_username" json:"auth\_username"`  
 AuthPassword Secret `yaml:"auth\_password" json:"auth\_password"`  
 AuthSecret Secret `yaml:"auth\_secret" json:"auth\_secret"`  
 AuthIdentity string `yaml:"auth\_identity" json:"auth\_identity"`  
 Headers **map**[string]string `yaml:"headers" json:"headers"`  
 HTML string `yaml:"html" json:"html"`  
 RequireTLS \*bool `yaml:"require\_tls,omitempty" json:"require\_tls,omitempty"`  
  
 // Catches all undefined fields and must be empty after parsing.  
 XXX **map**[string]**interface**{} `yaml:",inline" json:"-"`  
}

# dispatch\dispatch.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\github.com\prometheus\alertmanager\dispatch\dispatch.go

## dispatch.go.NewDispatcher

// NewDispatcher returns a new Dispatcher.  
**func** NewDispatcher(  
 ap provider.Alerts,  
 r \*Route,  
 s notify.Stage,  
 mk types.Marker,  
 to **func**(time.Duration) time.Duration,  
) \*Dispatcher {  
 disp := &Dispatcher{  
 alerts: ap,  
 stage: s,  
 route: r,  
 marker: mk,  
 timeout: to,  
 log: log.With("component", "dispatcher"),  
 }  
 **return** disp  
}

## Dispatcher.Run

// Run starts dispatching alerts incoming via the updates channel.  
**func** (d \*Dispatcher) Run() {  
 d.done = make(**chan struct**{})  
  
 d.mtx.Lock()  
 d.aggrGroups = **map**[\*Route]**map**[model.Fingerprint]\*aggrGroup{}  
 d.mtx.Unlock()  
  
 d.ctx, d.cancel = context.WithCancel(context.Background())  
  
 d.run(d.alerts.Subscribe())  
 close(d.done)  
}

## 2@Dispatcher.run

**func** (d \*Dispatcher) run(it provider.AlertIterator) {  
 cleanup := time.NewTicker(30 \* time.*Second*)  
 **defer** cleanup.Stop()  
  
 **defer** it.Close()  
  
 **for** {  
 **select** {  
 **case** alert, ok := <-it.Next():  
 **if** !ok {  
 // Iterator exhausted for some reason.  
 **if** err := it.Err(); err != nil {  
 log.Errorf("Error on alert update: %s", err)  
 }  
 **return** }  
  
 d.log.With("alert", alert).Debug("Received alert")  
  
 // Log errors but keep trying.  
 **if** err := it.Err(); err != nil {  
 log.Errorf("Error on alert update: %s", err)  
 **continue** }  
  
 **for** \_, r := **range** d.route.Match(alert.Labels) {  
 d.processAlert(alert, r)  
 }  
  
 **case** <-cleanup.C:  
 d.mtx.Lock()  
  
 **for** \_, groups := **range** d.aggrGroups {  
 **for** \_, ag := **range** groups {  
 **if** ag.empty() {  
 ag.stop()  
 delete(groups, ag.fingerprint())  
 }  
 }  
 }  
  
 d.mtx.Unlock()  
  
 **case** <-d.ctx.Done():  
 **return** }  
 }  
}

## dispatch.go.newAggrGroup

// newAggrGroup returns a new aggregation group.  
**func** newAggrGroup(ctx context.Context, labels model.LabelSet, r \*Route, to **func**(time.Duration) time.Duration) \*aggrGroup {  
 **if** to == nil {  
 to = **func**(d time.Duration) time.Duration { **return** d }  
 }  
 ag := &aggrGroup{  
 labels: labels,  
 routeKey: r.Key(),  
 opts: &r.RouteOpts,  
 timeout: to,  
 alerts: **map**[model.Fingerprint]\*types.Alert{},  
 }  
 ag.ctx, ag.cancel = context.WithCancel(ctx)  
  
 ag.log = log.With("aggrGroup", ag)  
  
 // Set an initial one-time wait before flushing  
 // the first batch of notifications.  
 ag.next = time.NewTimer(ag.opts.GroupWait)  
  
 **return** ag  
}

使用Route.Key指定routeKey

routeKey: r.Key(),

## aggrGroup.run

**func** (ag \*aggrGroup) run(nf notifyFunc) {  
 ag.done = make(**chan struct**{})  
  
 **defer** close(ag.done)  
 **defer** ag.next.Stop()  
  
 **for** {  
 **select** {  
 **case** now := <-ag.next.C:  
 // Give the notifcations time until the next flush to  
 // finish before terminating them.  
 ctx, cancel := context.WithTimeout(ag.ctx, ag.timeout(ag.opts.GroupInterval))  
  
 // The now time we retrieve from the ticker is the only reliable  
 // point of time reference for the subsequent notification pipeline.  
 // Calculating the current time directly is prone to flaky behavior,  
 // which usually only becomes apparent in tests.  
 ctx = notify.WithNow(ctx, now)  
  
 // Populate context with information needed along the pipeline.  
 ctx = notify.WithGroupKey(ctx, ag.GroupKey())  
 ctx = notify.WithGroupLabels(ctx, ag.labels)  
 ctx = notify.WithReceiverName(ctx, ag.opts.Receiver)  
 ctx = notify.WithRepeatInterval(ctx, ag.opts.RepeatInterval)  
  
 // Wait the configured interval before calling flush again.  
 ag.mtx.Lock()  
 ag.next.Reset(ag.opts.GroupInterval)  
 ag.mtx.Unlock()  
  
 ag.flush(**func**(alerts ...\*types.Alert) bool {  
 **return** nf(ctx, alerts...)  
 })  
  
 cancel()  
  
 **case** <-ag.ctx.Done():  
 **return** }  
 }  
}

### 调用

#### notify.WithGroupKey

ctx = notify.WithGroupKey(ctx, ag.GroupKey())

// WithGroupKey populates a context with a group key.  
**func** WithGroupKey(ctx context.Context, s string) context.Context {  
 **return** context.WithValue(ctx, *keyGroupKey*, s)  
}

//使用ag.routeKey

**return** fmt.Sprintf("%s:%s", ag.routeKey, ag.labels)

## aggrGroup.GroupKey

**func** (ag \*aggrGroup) GroupKey() string {  
 **return** fmt.Sprintf("%s:%s", ag.routeKey, ag.labels)  
}

## 3@Dispatcher.processAlert

// processAlert determines in which aggregation group the alert falls  
// and insert it.  
**func** (d \*Dispatcher) processAlert(alert \*types.Alert, route \*Route) {  
 group := model.LabelSet{}  
  
 **for** ln, lv := **range** alert.Labels {  
 **if** \_, ok := route.RouteOpts.GroupBy[ln]; ok {  
 group[ln] = lv  
 }  
 }  
  
 fp := group.Fingerprint()  
  
 d.mtx.Lock()  
 groups, ok := d.aggrGroups[route]  
 **if** !ok {  
 groups = **map**[model.Fingerprint]\*aggrGroup{}  
 d.aggrGroups[route] = groups  
 }  
 d.mtx.Unlock()  
  
 // If the group does not exist, create it.  
 ag, ok := groups[fp]  
 **if** !ok {  
 ag = newAggrGroup(d.ctx, group, route, d.timeout)  
 groups[fp] = ag  
  
 **go** ag.run(**func**(ctx context.Context, alerts ...\*types.Alert) bool {  
 \_, \_, err := d.stage.Exec(ctx, alerts...)  
 **if** err != nil {  
 log.Errorf("Notify for %d alerts failed: %s", len(alerts), err)  
 }  
 **return** err == nil  
 })  
 }  
  
 ag.insert(alert)  
}

### 调用

#### newAggrGroup

ag = newAggrGroup(d.ctx, group, route, d.timeout)

group := model.LabelSet{}

//将alert.Labels中的键值对转换到group中去

**for** ln, lv := **range** alert.Labels {  
 **if** \_, ok := route.RouteOpts.GroupBy[ln]; ok {  
 group[ln] = lv  
 }  
}

//根据route构造groups，并将groups传递给newAggrGroup

groups, ok := d.aggrGroups[route]  
**if** !ok {  
 groups = **map**[model.Fingerprint]\*aggrGroup{}  
 d.aggrGroups[route] = groups  
}

#### 2@d.stage.Exec

//调用stage.Exec

\_, \_, err := d.stage.Exec(ctx, alerts...)

先调用RoutingStage，获取目标实际的stage，然后调用特定的例如RetryStage.Exec

#### ag.run

//aggrGroup.run

**go** ag.run(**func**(ctx context.Context, alerts ...\*types.Alert) bool {  
 \_, \_, err := d.stage.Exec(ctx, alerts...)  
 **if** err != nil {  
 log.Errorf("Notify for %d alerts failed: %s", len(alerts), err)  
 }  
 **return** err == nil  
})

在aggrGroup中设置完*keyGroupKey*，传递给d.stage.Exec

# dispatch\route.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\github.com\prometheus\alertmanager\dispatch\route.go

## route.go.NewRoute

// NewRoute returns a new route.  
**func** NewRoute(cr \*config.Route, parent \*Route) \*Route {  
 // Create default and overwrite with configured settings.  
 opts := DefaultRouteOpts  
 **if** parent != nil {  
 opts = parent.RouteOpts  
 }  
  
 **if** cr.Receiver != "" {  
 opts.Receiver = cr.Receiver  
 }  
 **if** cr.GroupBy != nil {  
 opts.GroupBy = **map**[model.LabelName]**struct**{}{}  
 **for** \_, ln := **range** cr.GroupBy {  
 opts.GroupBy[ln] = **struct**{}{}  
 }  
 }  
 **if** cr.GroupWait != nil {  
 opts.GroupWait = time.Duration(\*cr.GroupWait)  
 }  
 **if** cr.GroupInterval != nil {  
 opts.GroupInterval = time.Duration(\*cr.GroupInterval)  
 }  
 **if** cr.RepeatInterval != nil {  
 opts.RepeatInterval = time.Duration(\*cr.RepeatInterval)  
 }  
  
 // Build matchers.  
 **var** matchers types.Matchers  
  
 **for** ln, lv := **range** cr.Match {  
 matchers = append(matchers, types.NewMatcher(model.LabelName(ln), lv))  
 }  
 **for** ln, lv := **range** cr.MatchRE {  
 matchers = append(matchers, types.NewRegexMatcher(model.LabelName(ln), lv.Regexp))  
 }  
  
 route := &Route{  
 parent: parent,  
 RouteOpts: opts,  
 Matchers: matchers,  
 Continue: cr.Continue,  
 }  
  
 route.Routes = NewRoutes(cr.Routes, route)  
  
 **return** route  
}

### 调用

#### 设置RepeatInterval

**if** cr.RepeatInterval != nil {  
 opts.RepeatInterval = time.Duration(\*cr.RepeatInterval)  
}

# notify\notify.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\github.com\prometheus\alertmanager\notify\notify.go

## RetryStage.Exec

// Exec implements the Stage interface.  
**func** (r RetryStage) Exec(ctx context.Context, alerts ...\*types.Alert) (context.Context, []\*types.Alert, error) {  
 **var** (  
 i = 0  
 b = backoff.NewExponentialBackOff()  
 tick = backoff.NewTicker(b)  
 iErr error  
 )  
 **defer** tick.Stop()  
  
 **for** {  
 i++  
 // Always check the context first to not notify again.  
 **select** {  
 **case** <-ctx.Done():  
 **if** iErr != nil {  
 **return** ctx, nil, iErr  
 }  
  
 **return** ctx, nil, ctx.Err()  
 **default**:  
 }  
  
 **select** {  
 **case** <-tick.C:  
 **if** retry, err := r.integration.Notify(ctx, alerts...); err != nil {  
 numFailedNotifications.WithLabelValues(r.integration.name).Inc()  
 log.Debugf("Notify attempt %d for %q failed: %s", i, r.integration.name, err)  
 **if** !retry {  
 **return** ctx, alerts, fmt.Errorf("Cancelling notify retry for %q due to unrecoverable error: %s", r.integration.name, err)  
 }  
  
 // Save this error to be able to return the last seen error by an  
 // integration upon context timeout.  
 iErr = err  
 } **else** {  
 numNotifications.WithLabelValues(r.integration.name).Inc()  
 **return** ctx, alerts, nil  
 }  
 **case** <-ctx.Done():  
 **if** iErr != nil {  
 **return** ctx, nil, iErr  
 }  
  
 **return** ctx, nil, ctx.Err()  
 }  
 }  
}

不断的尝试发邮件

### 调用

#### r.integration.Notify

//调用impl.go Integration.Notify

**if** retry, err := r.integration.Notify(ctx, alerts...); err != nil {

## RoutingStage.Exec

// Exec implements the Stage interface.  
**func** (rs RoutingStage) Exec(ctx context.Context, alerts ...\*types.Alert) (context.Context, []\*types.Alert, error) {  
 receiver, ok := ReceiverName(ctx)  
 **if** !ok {  
 **return** ctx, nil, fmt.Errorf("receiver missing")  
 }  
  
 s, ok := rs[receiver]  
 **if** !ok {  
 **return** ctx, nil, fmt.Errorf("stage for receiver missing")  
 }  
  
 **return** s.Exec(ctx, alerts...)  
}

//获取receiver的名称

receiver, ok := ReceiverName(ctx)

//根据receiver获取stage

s, ok := rs[receiver]

## DedupStage.struct

// DedupStage filters alerts.  
// Filtering happens based on a notification log.  
**type** DedupStage **struct** {  
 nflog nflog.Log  
 recv \*nflogpb.Receiver  
 sendResolved bool  
  
 now **func**() time.Time  
 hash **func**(\*types.Alert) uint64  
}

## @DedupStage.Exec

// Exec implements the Stage interface.  
**func** (n \*DedupStage) Exec(ctx context.Context, alerts ...\*types.Alert) (context.Context, []\*types.Alert, error) {  
 gkey, ok := GroupKey(ctx)  
 **if** !ok {  
 **return** ctx, nil, fmt.Errorf("group key missing")  
 }  
  
 repeatInterval, ok := RepeatInterval(ctx)  
 **if** !ok {  
 **return** ctx, nil, fmt.Errorf("repeat interval missing")  
 }  
  
 firingSet := **map**[uint64]**struct**{}{}  
 resolvedSet := **map**[uint64]**struct**{}{}  
 firing := []uint64{}  
 resolved := []uint64{}  
  
 **var** hash uint64  
 **for** \_, a := **range** alerts {  
 hash = n.hash(a)  
 **if** a.Resolved() {  
 resolved = append(resolved, hash)  
 resolvedSet[hash] = **struct**{}{}  
 } **else** {  
 firing = append(firing, hash)  
 firingSet[hash] = **struct**{}{}  
 }  
 }  
  
 ctx = WithFiringAlerts(ctx, firing)  
 ctx = WithResolvedAlerts(ctx, resolved)  
  
 entries, err := n.nflog.Query(nflog.QGroupKey(gkey), nflog.QReceiver(n.recv))  
  
 **if** err != nil && err != nflog.ErrNotFound {  
 **return** ctx, nil, err  
 }  
 **var** entry \*nflogpb.Entry  
 **switch** len(entries) {  
 **case** 0:  
 **case** 1:  
 entry = entries[0]  
 **case** 2:  
 **return** ctx, nil, fmt.Errorf("Unexpected entry result size %d", len(entries))  
 }  
 **if** ok, err := n.needsUpdate(entry, firingSet, resolvedSet, repeatInterval); err != nil {  
 **return** ctx, nil, err  
 } **else if** ok {  
 **return** ctx, alerts, nil  
 }  
 **return** ctx, nil, nil  
}

### 调用

#### GroupKey

gkey, ok := GroupKey(ctx)

// GroupKey extracts a group key from the context. Iff none exists, the  
// second argument is false.  
**func** GroupKey(ctx context.Context) (string, bool) {  
 v, ok := ctx.Value(*keyGroupKey*).(string)  
 **return** v, ok  
}

#### n.needsUpdate

在repeatInterval之后才触发返回告警内容

**if** ok, err := n.needsUpdate(entry, firingSet, resolvedSet, repeatInterval); err != nil {  
 **return** ctx, nil, err  
} **else if** ok {  
 **return** ctx, alerts, nil  
}

#### nflog.QReceiver

entries, err := n.nflog.Query(nflog.QGroupKey(gkey), nflog.QReceiver(n.recv))

## DedupStage.needsUpdate

**func** (n \*DedupStage) needsUpdate(entry \*nflogpb.Entry, firing, resolved **map**[uint64]**struct**{}, repeat time.Duration) (bool, error) {  
 // If we haven't notified about the alert group before, notify right away  
 // unless we only have resolved alerts.  
 **if** entry == nil {  
 **return** ((len(firing) > 0) || (n.sendResolved && len(resolved) > 0)), nil  
 }  
  
 **if** !entry.IsFiringSubset(firing) {  
 **return** true, nil  
 }  
  
 **if** n.sendResolved && !entry.IsResolvedSubset(resolved) {  
 **return** true, nil  
 }  
  
 // Nothing changed, only notify if the repeat interval has passed.  
 **return** entry.Timestamp.Before(n.now().Add(-repeat)), nil  
}

## notify.go.BuildPipeline

// BuildPipeline builds a map of receivers to Stages.  
**func** BuildPipeline(  
 confs []\*config.Receiver,  
 tmpl \*template.Template,  
 wait **func**() time.Duration,  
 inhibitor \*inhibit.Inhibitor,  
 silences \*silence.Silences,  
 notificationLog nflog.Log,  
 marker types.Marker,  
) RoutingStage {  
 rs := RoutingStage{}  
  
 is := NewInhibitStage(inhibitor, marker)  
 ss := NewSilenceStage(silences, marker)  
  
 **for** \_, rc := **range** confs {  
 rs[rc.Name] = MultiStage{is, ss, createStage(rc, tmpl, wait, notificationLog)}  
 }  
 **return** rs  
}

返回RoutingStage

// RoutingStage executes the inner stages based on the receiver specified in  
// the context.  
**type** RoutingStage **map**[string]Stage

// A Stage processes alerts under the constraints of the given context.  
**type** Stage **interface** {  
 Exec(ctx context.Context, alerts ...\*types.Alert) (context.Context, []\*types.Alert, error)  
}

### 调用

rs[rc.Name] = MultiStage{is, ss, createStage(rc, tmpl, wait, notificationLog)}

## 2@notify.go.createStage

// createStage creates a pipeline of stages for a receiver.  
**func** createStage(rc \*config.Receiver, tmpl \*template.Template, wait **func**() time.Duration, notificationLog nflog.Log) Stage {  
 **var** fs FanoutStage  
 **for** \_, i := **range** BuildReceiverIntegrations(rc, tmpl) {  
 recv := &nflogpb.Receiver{  
 GroupName: rc.Name,  
 Integration: i.name,  
 Idx: uint32(i.idx),  
 }  
 **var** s MultiStage  
 s = append(s, NewWaitStage(wait))  
 s = append(s, NewDedupStage(notificationLog, recv, i.conf.SendResolved()))  
 s = append(s, NewRetryStage(i))  
 s = append(s, NewSetNotifiesStage(notificationLog, recv))  
  
 fs = append(fs, s)  
 }  
 **return** fs  
}

### 调用

//接受者一定会有NewRetryStage

s = append(s, NewRetryStage(i))

// NewRetryStage returns a new instance of a RetryStage.  
**func** NewRetryStage(i Integration) \*RetryStage {  
 **return** &RetryStage{  
 integration: i,  
 }  
}

#### Impl.go.BuildReceiverIntegrations

**for** \_, i := **range** BuildReceiverIntegrations(rc, tmpl) {

## FanoutStage.Exec

// FanoutStage executes its stages concurrently  
**type** FanoutStage []Stage  
  
// Exec attempts to execute all stages concurrently and discards the results.  
// It returns its input alerts and a types.MultiError if one or more stages fail.  
**func** (fs FanoutStage) Exec(ctx context.Context, alerts ...\*types.Alert) (context.Context, []\*types.Alert, error) {  
 **var** (  
 wg sync.WaitGroup  
 me types.MultiError  
 )  
 wg.Add(len(fs))  
  
 **for** \_, s := **range** fs {  
 **go func**(s Stage) {  
 **if** \_, \_, err := s.Exec(ctx, alerts...); err != nil {  
 me.Add(err)  
 log.Errorf("Error on notify: %s", err)  
 }  
 wg.Done()  
 }(s)  
 }  
 wg.Wait()  
  
 **if** me.Len() > 0 {  
 **return** ctx, alerts, &me  
 }  
 **return** ctx, alerts, nil  
}

# notify\impl.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\github.com\prometheus\alertmanager\notify\impl.go

## Integration.Notify

// Notify implements the Notifier interface.  
**func** (i \*Integration) Notify(ctx context.Context, alerts ...\*types.Alert) (bool, error) {  
 **var** res []\*types.Alert  
  
 // Resolved alerts have to be filtered only at this point, because they need  
 // to end up unfiltered in the SetNotifiesStage.  
 **if** i.conf.SendResolved() {  
 res = alerts  
 } **else** {  
 **for** \_, a := **range** alerts {  
 **if** a.Status() != model.*AlertResolved* {  
 res = append(res, a)  
 }  
 }  
 }  
 **if** len(res) == 0 {  
 **return** false, nil  
 }  
  
 **return** i.notifier.Notify(ctx, res...)  
}

### 调用

**return** i.notifier.Notify(ctx, res...)

### 问题

#### Notifier的初始化

**return** i.notifier.Notify(ctx, res...)

检查[notify.go.createStage](#_Impl.go.BuildReceiverIntegrations)的代码

//首先构建BuildReceiverIntegrations

**for** \_, i := **range** BuildReceiverIntegrations(rc, tmpl) {

//将i传入到对应的Stage

s = append(s, NewRetryStage(i))

// NewRetryStage returns a new instance of a RetryStage.  
**func** NewRetryStage(i Integration) \*RetryStage {  
 **return** &RetryStage{  
 integration: i,  
 }  
}

// An Integration wraps a notifier and its config to be uniquely identified by  
// name and index from its origin in the configuration.  
**type** Integration **struct** {  
 notifier Notifier  
 conf notifierConfig  
 name string  
 idx int  
}

## Email.Nofify

// Notify implements the Notifier interface.  
**func** (n \*Email) Notify(ctx context.Context, as ...\*types.Alert) (bool, error) {  
 // Connect to the SMTP smarthost.  
 c, err := smtp.Dial(n.conf.Smarthost)  
 **if** err != nil {  
 **return** true, err  
 }  
 **defer** c.Quit()  
  
 // We need to know the hostname for both auth and TLS.  
 host, \_, err := net.SplitHostPort(n.conf.Smarthost)  
 **if** err != nil {  
 **return** false, fmt.Errorf("invalid address: %s", err)  
 }  
  
 // Global Config guarantees RequireTLS is not nil  
 **if** \*n.conf.RequireTLS {  
 **if** ok, \_ := c.Extension("STARTTLS"); !ok {  
 **return** true, fmt.Errorf("require\_tls: true (default), but %q does not advertise the STARTTLS extension", n.conf.Smarthost)  
 }  
 tlsConf := &tls.Config{ServerName: host}  
 **if** err := c.StartTLS(tlsConf); err != nil {  
 **return** true, fmt.Errorf("starttls failed: %s", err)  
 }  
 }  
  
 **if** ok, mech := c.Extension("AUTH"); ok {  
 auth, err := n.auth(mech)  
 **if** err != nil {  
 **return** true, err  
 }  
 **if** auth != nil {  
 **if** err := c.Auth(auth); err != nil {  
 **return** true, fmt.Errorf("%T failed: %s", auth, err)  
 }  
 }  
 }  
  
 **var** (  
 data = n.tmpl.Data(receiverName(ctx), groupLabels(ctx), as...)  
 tmpl = tmplText(n.tmpl, data, &err)  
 from = tmpl(n.conf.From)  
 to = tmpl(n.conf.To)  
 )  
 **if** err != nil {  
 **return** false, err  
 }  
  
 addrs, err := mail.ParseAddressList(from)  
 **if** err != nil {  
 **return** false, fmt.Errorf("parsing from addresses: %s", err)  
 }  
 **if** len(addrs) != 1 {  
 **return** false, fmt.Errorf("must be exactly one from address")  
 }  
 **if** err := c.Mail(addrs[0].Address); err != nil {  
 **return** true, fmt.Errorf("sending mail from: %s", err)  
 }  
 addrs, err = mail.ParseAddressList(to)  
 **if** err != nil {  
 **return** false, fmt.Errorf("parsing to addresses: %s", err)  
 }  
 **for** \_, addr := **range** addrs {  
 **if** err := c.Rcpt(addr.Address); err != nil {  
 **return** true, fmt.Errorf("sending rcpt to: %s", err)  
 }  
 }  
  
 // Send the email body.  
 wc, err := c.Data()  
 **if** err != nil {  
 **return** true, err  
 }  
 **defer** wc.Close()  
  
 **for** header, t := **range** n.conf.Headers {  
 value, err := n.tmpl.ExecuteTextString(t, data)  
 **if** err != nil {  
 **return** false, fmt.Errorf("executing %q header template: %s", header, err)  
 }  
 fmt.Fprintf(wc, "%s: %s\r\n", header, mime.*QEncoding*.Encode("utf-8", value))  
 }  
  
 fmt.Fprintf(wc, "Content-Type: text/html; charset=UTF-8\r\n")  
 fmt.Fprintf(wc, "Date: %s\r\n", time.Now().Format(time.*RFC1123Z*))  
  
 // *TODO: Add some useful headers here, such as URL of the alertmanager* // and active/resolved.  
 fmt.Fprintf(wc, "\r\n")  
  
 // *TODO(fabxc): do a multipart write that considers the plain template.* body, err := n.tmpl.ExecuteHTMLString(n.conf.HTML, data)  
 **if** err != nil {  
 **return** false, fmt.Errorf("executing email html template: %s", err)  
 }  
 \_, err = io.WriteString(wc, body)  
 **if** err != nil {  
 **return** true, err  
 }  
  
 **return** false, nil  
}

## Impl.go.BuildReceiverIntegrations

// BuildReceiverIntegrations builds a list of integration notifiers off of a  
// receivers config.  
**func** BuildReceiverIntegrations(nc \*config.Receiver, tmpl \*template.Template) []Integration {  
 **var** (  
 integrations []Integration  
 add = **func**(name string, i int, n Notifier, nc notifierConfig) {  
 integrations = append(integrations, Integration{  
 notifier: n,  
 conf: nc,  
 name: name,  
 idx: i,  
 })  
 }  
 )  
  
 **for** i, c := **range** nc.WebhookConfigs {  
 n := NewWebhook(c, tmpl)  
 add("webhook", i, n, c)  
 }  
 **for** i, c := **range** nc.EmailConfigs {  
 n := NewEmail(c, tmpl)  
 add("email", i, n, c)  
 }  
 **for** i, c := **range** nc.PagerdutyConfigs {  
 n := NewPagerDuty(c, tmpl)  
 add("pagerduty", i, n, c)  
 }  
 **for** i, c := **range** nc.OpsGenieConfigs {  
 n := NewOpsGenie(c, tmpl)  
 add("opsgenie", i, n, c)  
 }  
 **for** i, c := **range** nc.SlackConfigs {  
 n := NewSlack(c, tmpl)  
 add("slack", i, n, c)  
 }  
 **for** i, c := **range** nc.HipchatConfigs {  
 n := NewHipchat(c, tmpl)  
 add("hipchat", i, n, c)  
 }  
 **for** i, c := **range** nc.VictorOpsConfigs {  
 n := NewVictorOps(c, tmpl)  
 add("victorops", i, n, c)  
 }  
 **for** i, c := **range** nc.PushoverConfigs {  
 n := NewPushover(c, tmpl)  
 add("pushover", i, n, c)  
 }  
 **return** integrations  
}