# cmd\prometheus\main.go

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

//初始化nofifier

**var** (  
 notifier = notifier.New(&cfg.notifier)  
 targetManager = retrieval.NewTargetManager(sampleAppender)  
 queryEngine = promql.NewEngine(queryable, &cfg.queryEngine)  
 ctx, cancelCtx = context.WithCancel(context.Background())  
)

//将notifier与ruleManager关联起来

ruleManager := rules.NewManager(&rules.ManagerOptions{  
 SampleAppender: sampleAppender,  
 Notifier: notifier,  
 QueryEngine: queryEngine,  
 Context: fanin.WithLocalOnly(ctx),  
 ExternalURL: cfg.web.ExternalURL,  
})

//重新加载时，将重新配置notifier

reloadables = append(reloadables, targetManager, ruleManager, webHandler, notifier)

//对外提供http访问

cfg.web.Notifier = notifier

//开始执行notifier

**go** notifier.Run()

# notifier\notifier.go

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

## Notifier.Run

// Run dispatches notifications continuously.  
**func** (n \*Notifier) Run() {  
 **for** {  
 **select** {  
 **case** <-n.ctx.Done():  
 **return  
 case** <-n.more:  
 }  
 alerts := n.nextBatch()  
  
 **if** !n.sendAll(alerts...) {  
 n.metrics.dropped.Add(float64(len(alerts)))  
 }  
 // If the queue still has items left, kick off the next iteration.  
 **if** n.queueLen() > 0 {  
 n.setMore()  
 }  
 }  
}

读取到n.more后将执行邮件发送动作

## Notifier.Send

// Send queues the given notification requests for processing.  
// Panics if called on a handler that is not running.  
**func** (n \*Notifier) Send(alerts ...\*model.Alert) {  
 n.mtx.Lock()  
 **defer** n.mtx.Unlock()  
  
 // Attach external labels before relabelling and sending.  
 **for** \_, a := **range** alerts {  
 **for** ln, lv := **range** n.opts.ExternalLabels {  
 **if** \_, ok := a.Labels[ln]; !ok {  
 a.Labels[ln] = lv  
 }  
 }  
 }  
  
 alerts = n.relabelAlerts(alerts)  
  
 // Queue capacity should be significantly larger than a single alert  
 // batch could be.  
 **if** d := len(alerts) - n.opts.QueueCapacity; d > 0 {  
 alerts = alerts[d:]  
  
 log.Warnf("Alert batch larger than queue capacity, dropping %d alerts", d)  
 n.metrics.dropped.Add(float64(d))  
 }  
  
 // If the queue is full, remove the oldest alerts in favor  
 // of newer ones.  
 **if** d := (len(n.queue) + len(alerts)) - n.opts.QueueCapacity; d > 0 {  
 n.queue = n.queue[d:]  
  
 log.Warnf("Alert notification queue full, dropping %d alerts", d)  
 n.metrics.dropped.Add(float64(d))  
 }  
 n.queue = append(n.queue, alerts...)  
  
 // Notify sending goroutine that there are alerts to be processed.  
 n.setMore()  
}

//将告警附加在告警队列里

n.queue = append(n.queue, alerts...)

## Notifier.setMore

通知队列中有值，发送邮件

// setMore signals that the alert queue has items.  
**func** (n \*Notifier) setMore() {  
 // If we cannot send on the channel, it means the signal already exists  
 // and has not been consumed yet.  
 **select** {  
 **case** n.more <- **struct**{}{}:  
 **default**:  
 }  
}

## Notifier.sendAll

实现发送消息的核心方法

// sendAll sends the alerts to all configured Alertmanagers concurrently.  
// It returns true if the alerts could be sent successfully to at least one Alertmanager.  
**func** (n \*Notifier) sendAll(alerts ...\*model.Alert) bool {  
 begin := time.Now()  
  
 b, err := json.Marshal(alerts)  
 **if** err != nil {  
 log.Errorf("Encoding alerts failed: %s", err)  
 **return** false  
 }  
  
 n.mtx.RLock()  
 amSets := n.alertmanagers  
 n.mtx.RUnlock()  
  
 **var** (  
 wg sync.WaitGroup  
 numSuccess uint64  
 )  
 **for** \_, ams := **range** amSets {  
 ams.mtx.RLock()  
  
 **for** \_, am := **range** ams.ams {  
 wg.Add(1)  
  
 ctx, cancel := context.WithTimeout(n.ctx, ams.cfg.Timeout)  
 **defer** cancel()  
  
 **go func**(am alertmanager) {  
 u := am.url().String()  
  
 **if** err := n.sendOne(ctx, ams.client, u, b); err != nil {  
 log.With("alertmanager", u).With("count", len(alerts)).Errorf("Error sending alerts: %s", err)  
 n.metrics.errors.WithLabelValues(u).Inc()  
 } **else** {  
 atomic.AddUint64(&numSuccess, 1)  
 }  
 n.metrics.latency.WithLabelValues(u).Observe(time.Since(begin).Seconds())  
 n.metrics.sent.WithLabelValues(u).Add(float64(len(alerts)))  
  
 wg.Done()  
 }(am)  
 }  
 ams.mtx.RUnlock()  
 }  
 wg.Wait()  
  
 **return** numSuccess > 0  
}

## Notifier.sendOne

实施Http请求

**func** (n \*Notifier) sendOne(ctx context.Context, c \*http.Client, url string, b []byte) error {  
 req, err := http.NewRequest("POST", url, bytes.NewReader(b))  
 **if** err != nil {  
 **return** err  
 }  
 req.Header.Set("Content-Type", *contentTypeJSON*)  
 resp, err := n.opts.Do(ctx, c, req)  
 **if** err != nil {  
 **return** err  
 }  
 **defer** resp.Body.Close()  
  
 // Any HTTP status 2xx is OK.  
 **if** resp.StatusCode/100 != 2 {  
 **return** fmt.Errorf("bad response status %v", resp.Status)  
 }  
 **return** err  
}

### POST JSON

Configuration | Prometheus https://prometheus.io/docs/alerting/configuration/

{

"version": "3",

"groupKey": <number> // key identifying the group of alerts (e.g. to deduplicate)

"status": "<resolved|firing>",

"receiver": [<string>](https://prometheus.io/docs/alerting/configuration/#<string>),

"groupLabels": <object>,

"commonLabels": <object>,

"commonAnnotations": <object>,

"externalURL": [<string>](https://prometheus.io/docs/alerting/configuration/#<string>), // backling to the Alertmanager.

"alerts": [

{

"labels": <object>,

"annotations": <object>,

"startsAt": "<rfc3339>",

"endsAt": "<rfc3339>"

},

...

]

}

# rules\manager.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\prometheus\vendor\github.com\prometheus\prometheus\rules\manager.go

## Group.Eval

// Eval runs a single evaluation cycle in which all rules are evaluated in parallel.  
// In the future a single group will be evaluated sequentially to properly handle  
// rule dependency.  
**func** (g \*Group) Eval() {  
 **var** (  
 now = model.Now()  
 wg sync.WaitGroup  
 )  
  
 **for** \_, rule := **range** g.rules {  
 rtyp := string(typeForRule(rule))  
  
 wg.Add(1)  
 // BUG(julius): Look at fixing thundering herd.  
 **go func**(rule Rule) {  
 **defer** wg.Done()  
  
 **defer func**(t time.Time) {  
 evalDuration.WithLabelValues(rtyp).Observe(time.Since(t).Seconds())  
 }(time.Now())  
  
 evalTotal.WithLabelValues(rtyp).Inc()  
  
 vector, err := rule.Eval(g.opts.Context, now, g.opts.QueryEngine, g.opts.ExternalURL.Path)  
 **if** err != nil {  
 // Canceled queries are intentional termination of queries. This normally  
 // happens on shutdown and thus we skip logging of any errors here.  
 **if** \_, ok := err.(promql.ErrQueryCanceled); !ok {  
 log.Warnf("Error while evaluating rule %q: %s", rule, err)  
 }  
 evalFailures.WithLabelValues(rtyp).Inc()  
 **return** }  
  
 **if** ar, ok := rule.(\*AlertingRule); ok {  
 g.sendAlerts(ar, now)  
 }  
 **var** (  
 numOutOfOrder = 0  
 numDuplicates = 0  
 )  
 **for** \_, s := **range** vector {  
 **if** err := g.opts.SampleAppender.Append(s); err != nil {  
 **switch** err {  
 **case** local.ErrOutOfOrderSample:  
 numOutOfOrder++  
 log.With("sample", s).With("error", err).Debug("Rule evaluation result discarded")  
 **case** local.ErrDuplicateSampleForTimestamp:  
 numDuplicates++  
 log.With("sample", s).With("error", err).Debug("Rule evaluation result discarded")  
 **default**:  
 log.With("sample", s).With("error", err).Warn("Rule evaluation result discarded")  
 }  
 }  
 }  
 **if** numOutOfOrder > 0 {  
 log.With("numDropped", numOutOfOrder).Warn("Error on ingesting out-of-order result from rule evaluation")  
 }  
 **if** numDuplicates > 0 {  
 log.With("numDropped", numDuplicates).Warn("Error on ingesting results from rule evaluation with different value but same timestamp")  
 }  
 }(rule)  
 }  
 wg.Wait()  
}

//根据rule type来判断发送告警的频率

**defer func**(t time.Time) {  
 evalDuration.WithLabelValues(rtyp).Observe(time.Since(t).Seconds())  
}(time.Now())

### 调用

g.sendAlerts(ar, now)

//调用AlertRule.Eval

vector, err := rule.Eval(g.opts.Context, now, g.opts.QueryEngine, g.opts.ExternalURL.Path)

## 2@Group.sendAlerts

// sendAlerts sends alert notifications for the given rule.  
**func** (g \*Group) sendAlerts(rule \*AlertingRule, timestamp model.Time) error {  
 **var** alerts model.Alerts  
  
 **for** \_, alert := **range** rule.currentAlerts() {  
 // Only send actually firing alerts.  
 **if** alert.State == *StatePending* {  
 **continue** }  
  
 a := &model.Alert{  
 StartsAt: alert.ActiveAt.Add(rule.holdDuration).Time(),  
 Labels: alert.Labels,  
 Annotations: alert.Annotations,  
 GeneratorURL: g.opts.ExternalURL.String() + strutil.GraphLinkForExpression(rule.vector.String()),  
 }  
 **if** alert.ResolvedAt != 0 {  
 a.EndsAt = alert.ResolvedAt.Time()  
 }  
  
 alerts = append(alerts, a)  
 }  
  
 **if** len(alerts) > 0 {  
 g.opts.Notifier.Send(alerts...)  
 }  
  
 **return** nil  
}

### 调用

rule.currentAlerts()

## Group.run

**func** (g \*Group) run() {  
 **defer** close(g.terminated)  
  
 // Wait an initial amount to have consistently slotted intervals.  
 **select** {  
 **case** <-time.After(g.offset()):  
 **case** <-g.done:  
 **return** }  
  
 iter := **func**() {  
 iterationsScheduled.Inc()  
 **if** g.opts.SampleAppender.NeedsThrottling() {  
 iterationsSkipped.Inc()  
 **return** }  
 start := time.Now()  
 g.Eval()  
  
 iterationDuration.Observe(time.Since(start).Seconds())  
 }  
 lastTriggered := time.Now()  
 iter()  
  
 tick := time.NewTicker(g.interval)  
 **defer** tick.Stop()  
  
 **for** {  
 **select** {  
 **case** <-g.done:  
 **return  
 default**:  
 **select** {  
 **case** <-g.done:  
 **return  
 case** <-tick.C:  
 missed := (time.Since(lastTriggered).Nanoseconds() / g.interval.Nanoseconds()) - 1  
 **if** missed > 0 {  
 iterationsMissed.Add(float64(missed))  
 iterationsScheduled.Add(float64(missed))  
 }  
 lastTriggered = time.Now()  
 iter()  
 }  
 }  
 }  
}

### 调用

g.Eval()

g.Eval()

## Manager.ApplyConfig

rules\manager.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\prometheus\vendor\github.com\prometheus\prometheus\rules\manager.go

// ApplyConfig updates the rule manager's state as the config requires. If  
// loading the new rules failed the old rule set is restored.  
**func** (m \*Manager) ApplyConfig(conf \*config.Config) error {  
 m.mtx.Lock()  
 **defer** m.mtx.Unlock()  
  
 // Get all rule files and load the groups they define.  
 **var** files []string  
 **for** \_, pat := **range** conf.RuleFiles {  
 fs, err := filepath.Glob(pat)  
 **if** err != nil {  
 // The only error can be a bad pattern.  
 **return** fmt.Errorf("error retrieving rule files for %s: %s", pat, err)  
 }  
 files = append(files, fs...)  
 }  
  
 // To be replaced with a configurable per-group interval.  
 groups, err := m.loadGroups(time.Duration(conf.GlobalConfig.EvaluationInterval), files...)  
 **if** err != nil {  
 **return** fmt.Errorf("error loading rules, previous rule set restored: %s", err)  
 }  
  
 **var** wg sync.WaitGroup  
  
 **for** \_, newg := **range** groups {  
 wg.Add(1)  
  
 // If there is an old group with the same identifier, stop it and wait for  
 // it to finish the current iteration. Then copy its into the new group.  
 oldg, ok := m.groups[newg.name]  
 delete(m.groups, newg.name)  
  
 **go func**(newg \*Group) {  
 **if** ok {  
 oldg.stop()  
 newg.copyState(oldg)  
 }  
 **go func**() {  
 // Wait with starting evaluation until the rule manager  
 // is told to run. This is necessary to avoid running  
 // queries against a bootstrapping storage.  
 <-m.block  
 newg.run()  
 }()  
 wg.Done()  
 }(newg)  
 }  
  
 // Stop remaining old groups.  
 **for** \_, oldg := **range** m.groups {  
 oldg.stop()  
 }  
  
 wg.Wait()  
 m.groups = groups  
  
 **return** nil  
}

### 调用

newg.run()

groups, err := m.loadGroups(time.Duration(conf.GlobalConfig.EvaluationInterval), files...)

### evaluation\_interval

// To be replaced with a configurable per-group interval.  
groups, err := m.loadGroups(time.Duration(conf.GlobalConfig.EvaluationInterval), files...)

https://prometheus.io/docs/introduction/getting\_started/

global:

scrape\_interval: 15s # By default, scrape targets every 15 seconds.

evaluation\_interval: 15s # Evaluate rules every 15 seconds.

## Manager.loadGroups

// loadGroups reads groups from a list of files.  
// As there's currently no group syntax a single group named "default" containing  
// all rules will be returned.  
**func** (m \*Manager) loadGroups(interval time.Duration, filenames ...string) (**map**[string]\*Group, error) {  
 rules := []Rule{}  
 **for** \_, fn := **range** filenames {  
 content, err := ioutil.ReadFile(fn)  
 **if** err != nil {  
 **return** nil, err  
 }  
 stmts, err := promql.ParseStmts(string(content))  
 **if** err != nil {  
 **return** nil, fmt.Errorf("error parsing %s: %s", fn, err)  
 }  
  
 **for** \_, stmt := **range** stmts {  
 **var** rule Rule  
  
 **switch** r := stmt.(**type**) {  
 **case** \*promql.AlertStmt:  
 rule = NewAlertingRule(r.Name, r.Expr, r.Duration, r.Labels, r.Annotations)  
  
 **case** \*promql.RecordStmt:  
 rule = NewRecordingRule(r.Name, r.Expr, r.Labels)  
  
 **default**:  
 panic("retrieval.Manager.LoadRuleFiles: unknown statement type")  
 }  
 rules = append(rules, rule)  
 }  
 }  
  
 // Currently there is no group syntax implemented. Thus all rules  
 // are read into a single default group.  
 g := NewGroup("default", interval, rules, m.opts)  
 groups := **map**[string]\*Group{g.name: g}  
 **return** groups, nil  
}

### 调用

//调用alerting.go.NewAlertingRule

rule = NewAlertingRule(r.Name, r.Expr, r.Duration, r.Labels, r.Annotations)

## Manager.AlertingRules

// AlertingRules returns the list of the manager's alerting rules.  
**func** (m \*Manager) AlertingRules() []\*AlertingRule {  
 m.mtx.RLock()  
 **defer** m.mtx.RUnlock()  
  
 alerts := []\*AlertingRule{}  
 **for** \_, rule := **range** m.Rules() {  
 **if** alertingRule, ok := rule.(\*AlertingRule); ok {  
 alerts = append(alerts, alertingRule)  
 }  
 }  
 **return** alerts  
}

## manager.go.NewGroup

// NewGroup makes a new Group with the given name, options, and rules.  
**func** NewGroup(name string, interval time.Duration, rules []Rule, opts \*ManagerOptions) \*Group {  
 **return** &Group{  
 name: name,  
 interval: interval,  
 rules: rules,  
 opts: opts,  
 done: make(**chan struct**{}),  
 terminated: make(**chan struct**{}),  
 }  
}

# rules\alerting.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\prometheus\vendor\github.com\prometheus\prometheus\rules\alerting.go

## Alert.struct

// Alert is the user-level representation of a single instance of an alerting rule.  
**type** Alert **struct** {  
 State AlertState  
 Labels model.LabelSet  
 Annotations model.LabelSet  
 // The value at the last evaluation of the alerting expression.  
 Value model.SampleValue  
 // The interval during which the condition of this alert held true.  
 // ResolvedAt will be 0 to indicate a still active alert.  
 ActiveAt, ResolvedAt model.Time  
}

## AlertingRule.struct

// An AlertingRule generates alerts from its vector expression.  
**type** AlertingRule **struct** {  
 // The name of the alert.  
 name string  
 // The vector expression from which to generate alerts.  
 vector promql.Expr  
 // The duration for which a labelset needs to persist in the expression  
 // output vector before an alert transitions from Pending to Firing state.  
 holdDuration time.Duration  
 // Extra labels to attach to the resulting alert sample vectors.  
 labels model.LabelSet  
 // Non-identifying key/value pairs.  
 annotations model.LabelSet  
  
 // Protects the below.  
 mtx sync.Mutex  
 // A map of alerts which are currently active (Pending or Firing), keyed by  
 // the fingerprint of the labelset they correspond to.  
 active **map**[model.Fingerprint]\*Alert  
}

## State

枚举

**const** (  
 // StateInactive is the state of an alert that is neither firing nor pending.  
 *StateInactive* AlertState = iota  
 // StatePending is the state of an alert that has been active for less than  
 // the configured threshold duration.  
 *StatePending* // StateFiring is the state of an alert that has been active for longer than  
 // the configured threshold duration.  
 *StateFiring*)

## 2@AlertingRule.Eval

// Eval evaluates the rule expression and then creates pending alerts and fires  
// or removes previously pending alerts accordingly.  
**func** (r \*AlertingRule) Eval(ctx context.Context, ts model.Time, engine \*promql.Engine, externalURLPath string) (model.Vector, error) {  
 query, err := engine.NewInstantQuery(r.vector.String(), ts)  
 **if** err != nil {  
 **return** nil, err  
 }  
 res, err := query.Exec(ctx).Vector()  
 **if** err != nil {  
 **return** nil, err  
 }  
  
 r.mtx.Lock()  
 **defer** r.mtx.Unlock()  
  
 // Create pending alerts for any new vector elements in the alert expression  
 // or update the expression value for existing elements.  
 resultFPs := **map**[model.Fingerprint]**struct**{}{}  
  
 **for** \_, smpl := **range** res {  
 // Provide the alert information to the template.  
 l := make(**map**[string]string, len(smpl.Metric))  
 **for** k, v := **range** smpl.Metric {  
 l[string(k)] = string(v)  
 }  
  
 tmplData := **struct** {  
 Labels **map**[string]string  
 Value float64  
 }{  
 Labels: l,  
 Value: float64(smpl.Value),  
 }  
 // Inject some convenience variables that are easier to remember for users  
 // who are not used to Go's templating system.  
 defs := "{{$labels := .Labels}}{{$value := .Value}}"  
  
 expand := **func**(text model.LabelValue) model.LabelValue {  
 tmpl := template.NewTemplateExpander(  
 ctx,  
 defs+string(text),  
 "\_\_alert\_"+r.Name(),  
 tmplData,  
 ts,  
 engine,  
 externalURLPath,  
 )  
 result, err := tmpl.Expand()  
 **if** err != nil {  
 result = fmt.Sprintf("<error expanding template: %s>", err)  
 log.Warnf("Error expanding alert template %v with data '%v': %s", r.Name(), tmplData, err)  
 }  
 **return** model.LabelValue(result)  
 }  
  
 delete(smpl.Metric, model.*MetricNameLabel*)  
 labels := make(model.LabelSet, len(smpl.Metric)+len(r.labels)+1)  
 **for** ln, lv := **range** smpl.Metric {  
 labels[ln] = lv  
 }  
 **for** ln, lv := **range** r.labels {  
 labels[ln] = expand(lv)  
 }  
 labels[model.*AlertNameLabel*] = model.LabelValue(r.Name())  
  
 annotations := make(model.LabelSet, len(r.annotations))  
 **for** an, av := **range** r.annotations {  
 annotations[an] = expand(av)  
 }  
 fp := smpl.Metric.Fingerprint()  
 resultFPs[fp] = **struct**{}{}  
  
 // Check whether we already have alerting state for the identifying label set.  
 // Update the last value and annotations if so, create a new alert entry otherwise.  
 **if** alert, ok := r.active[fp]; ok && alert.State != *StateInactive* {  
 alert.Value = smpl.Value  
 alert.Annotations = annotations  
 **continue** }  
  
 r.active[fp] = &Alert{  
 Labels: labels,  
 Annotations: annotations,  
 ActiveAt: ts,  
 State: *StatePending*,  
 Value: smpl.Value,  
 }  
 }  
  
 **var** vec model.Vector  
 // Check if any pending alerts should be removed or fire now. Write out alert timeseries.  
 **for** fp, a := **range** r.active {  
 **if** \_, ok := resultFPs[fp]; !ok {  
 **if** a.State != *StateInactive* {  
 vec = append(vec, r.sample(a, ts, false))  
 }  
 // If the alert was previously firing, keep it around for a given  
 // retention time so it is reported as resolved to the AlertManager.  
 **if** a.State == *StatePending* || (a.ResolvedAt != 0 && ts.Sub(a.ResolvedAt) > *resolvedRetention*) {  
 delete(r.active, fp)  
 }  
 **if** a.State != *StateInactive* {  
 a.State = *StateInactive* a.ResolvedAt = ts  
 }  
 **continue** }  
  
 **if** a.State == *StatePending* && ts.Sub(a.ActiveAt) >= r.holdDuration {  
 vec = append(vec, r.sample(a, ts, false))  
 a.State = *StateFiring* }  
  
 vec = append(vec, r.sample(a, ts, true))  
 }  
  
 **return** vec, nil  
}

### StateFiring

//已经是StatePending的，时间上又超过设定的holdDuration，则改变状态

**if** a.State == *StatePending* && ts.Sub(a.ActiveAt) >= r.holdDuration {  
 vec = append(vec, r.sample(a, ts, false))  
 a.State = *StateFiring*}

### active

//默认生成StatePending的告警

r.active[fp] = &Alert{  
 Labels: labels,  
 Annotations: annotations,  
 ActiveAt: ts,  
 State: *StatePending*,  
 Value: smpl.Value,  
}

## AlertingRule.ActiveAlerts

// ActiveAlerts returns a slice of active alerts.  
**func** (r \*AlertingRule) ActiveAlerts() []\*Alert {  
 **var** res []\*Alert  
 **for** \_, a := **range** r.currentAlerts() {  
 **if** a.ResolvedAt == 0 {  
 res = append(res, a)  
 }  
 }  
 **return** res  
}

//只要没有标明ResolveAt就生成ActiveAlerts

**if** a.ResolvedAt == 0

## AlertingRule.currentAlerts

// currentAlerts returns all instances of alerts for this rule. This may include  
// inactive alerts that were previously firing.  
**func** (r \*AlertingRule) currentAlerts() []\*Alert {  
 r.mtx.Lock()  
 **defer** r.mtx.Unlock()  
  
 alerts := make([]\*Alert, 0, len(r.active))  
  
 **for** \_, a := **range** r.active {  
 anew := \*a  
 anew.Labels = anew.Labels.Clone()  
 anew.Annotations = anew.Annotations.Clone()  
 alerts = append(alerts, &anew)  
 }  
 **return** alerts  
}

## AlertingRule.sample

**func** (r \*AlertingRule) sample(alert \*Alert, ts model.Time, set bool) \*model.Sample {  
 metric := model.Metric(r.labels.Clone())  
  
 **for** ln, lv := **range** alert.Labels {  
 metric[ln] = lv  
 }  
  
 metric[model.*MetricNameLabel*] = *alertMetricName* metric[model.*AlertNameLabel*] = model.LabelValue(r.name)  
 metric[*alertStateLabel*] = model.LabelValue(alert.State.String())  
  
 s := &model.Sample{  
 Metric: metric,  
 Timestamp: ts,  
 Value: 0,  
 }  
 **if** set {  
 s.Value = 1  
 }  
 **return** s  
}

### alertStateLabel

## alerting.go.NewAlertingRule

// NewAlertingRule constructs a new AlertingRule.  
**func** NewAlertingRule(name string, vec promql.Expr, hold time.Duration, lbls, anns model.LabelSet) \*AlertingRule {  
 **return** &AlertingRule{  
 name: name,  
 vector: vec,  
 holdDuration: hold,  
 labels: lbls,  
 annotations: anns,  
 active: **map**[model.Fingerprint]\*Alert{},  
 }  
}

# common\model\alert.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\prometheus\vendor\github.com\prometheus\common\model\alert.go

## Alert.struct

// Alert is a generic representation of an alert in the Prometheus eco-system.  
**type** Alert **struct** {  
 // Label value pairs for purpose of aggregation, matching, and disposition  
 // dispatching. This must minimally include an "alertname" label.  
 Labels LabelSet `json:"labels"`  
  
 // Extra key/value information which does not define alert identity.  
 Annotations LabelSet `json:"annotations"`  
  
 // The known time range for this alert. Both ends are optional.  
 StartsAt time.Time `json:"startsAt,omitempty"`  
 EndsAt time.Time `json:"endsAt,omitempty"`  
 GeneratorURL string `json:"generatorURL"`  
}

## AlertStatus

**const** (  
 *AlertFiring* AlertStatus = "firing"  
 *AlertResolved* AlertStatus = "resolved"  
)

# client\_golang\prometheus\summary.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\prometheus\vendor\github.com\prometheus\client\_golang\prometheus\summary.go

## SummaryVec.WithLabelValues

// WithLabelValues works as GetMetricWithLabelValues, but panics where  
// GetMetricWithLabelValues would have returned an error. By not returning an  
// error, WithLabelValues allows shortcuts like  
// myVec.WithLabelValues("404", "GET").Observe(42.21)  
**func** (m \*SummaryVec) WithLabelValues(lvs ...string) Summary {  
 **return** m.MetricVec.WithLabelValues(lvs...).(Summary)  
}

# web\web.go

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

router.Get("/alerts", instrf("alerts", h.alerts))

## Handler.alerts

**func** (h \*Handler) alerts(w http.ResponseWriter, r \*http.Request) {  
 alerts := h.ruleManager.AlertingRules()  
 alertsSorter := byAlertStateAndNameSorter{alerts: alerts}  
 sort.Sort(alertsSorter)  
  
 alertStatus := AlertStatus{  
 AlertingRules: alertsSorter.alerts,  
 AlertStateToRowClass: **map**[rules.AlertState]string{  
 rules.*StateInactive*: "success",  
 rules.*StatePending*: "warning",  
 rules.*StateFiring*: "danger",  
 },  
 }  
 h.executeTemplate(w, "alerts.html", alertStatus)  
}

### 调用

alerts := h.ruleManager.AlertingRules()

# Issue

## Throttle resends of alerts · Issue #2585 · prometheus/prometheus

https://github.com/prometheus/prometheus/issues/2585