# 访问路径映射

## web\web.go

E:\workspace\go\prometheus\prometheus\vendor\github.com\prometheus\prometheus\

web\web.go

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

## 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)  
}

## 获取告警列表

rules\manager.go

E:\workspace\go\prometheus\prometheus\vendor\github.com\prometheus\prometheus\

rules\manager.go

// 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  
}

# 生成告警规则

## rules\manager.go

E:\workspace\go\prometheus\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  
}

**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()  
 }  
 }  
 }  
}

## 判断是否满足告警条件

// 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()  
}

## 发送告警

// 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  
}

# 发送告警线程

## 启动线程

E:\workspace\go\prometheus\prometheus\cmd\prometheus\main.go

// The notifier is a dependency of the rule manager. It has to be  
// started before and torn down afterwards.  
**go** notifier.Run()  
**defer** notifier.Stop()

## 执行发送告警

E:\workspace\go\prometheus\prometheus\vendor\github.com\prometheus\prometheus\notifier\notifier.go

// 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()  
 }  
 }  
}

## 根据配置中最大批量告警数发送告警

E:\workspace\go\prometheus\prometheus\vendor\github.com\prometheus\prometheus\notifier\notifier.go

**func** (n \*Notifier) nextBatch() []\*model.Alert {  
 n.mtx.Lock()  
 **defer** n.mtx.Unlock()  
  
 **var** alerts model.Alerts  
  
 **if** len(n.queue) > *maxBatchSize* {  
 alerts = append(make(model.Alerts, 0, *maxBatchSize*), n.queue[:*maxBatchSize*]...)  
 n.queue = n.queue[*maxBatchSize*:]  
 } **else** {  
 alerts = append(make(model.Alerts, 0, len(n.queue)), n.queue...)  
 n.queue = n.queue[:0]  
 }  
  
 **return** alerts  
}

## sendAll

E:\workspace\go\prometheus\prometheus\vendor\github.com\prometheus\prometheus\notifier\notifier.go

// 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()

//将对象变成json数据

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  
}

### 消息的格式

// 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"`  
}

## sendOne

**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  
}

## opts和metrices的初始化

E:\workspace\go\prometheus\prometheus\vendor\github.com\prometheus\prometheus\notifier\notifier.go

**func** New(o \*Options) \*Notifier {  
 ctx, cancel := context.WithCancel(context.Background())  
  
 **if** o.Do == nil {  
 o.Do = ctxhttp.Do  
 }  
  
 n := &Notifier{  
 queue: make(model.Alerts, 0, o.QueueCapacity),  
 ctx: ctx,  
 cancel: cancel,  
 more: make(**chan struct**{}, 1),  
 opts: o,  
 }  
  
 queueLenFunc := **func**() float64 { **return** float64(n.queueLen()) }  
 n.metrics = newAlertMetrics(o.Registerer, o.QueueCapacity, queueLenFunc)  
 **return** n  
}