# 配置文件解析

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

// FileSDConfig is the configuration for file based discovery.  
**type** FileSDConfig **struct** {  
 Files []string `yaml:"files"`  
 RefreshInterval model.Duration `yaml:"refresh\_interval,omitempty"`  
  
 // Catches all undefined fields and must be empty after parsing.  
 XXX **map**[string]**interface**{} `yaml:",inline"`  
}  
  
// UnmarshalYAML implements the yaml.Unmarshaler interface.  
**func** (c \*FileSDConfig) UnmarshalYAML(unmarshal **func**(**interface**{}) error) error {  
 \*c = DefaultFileSDConfig  
 **type** plain FileSDConfig  
 err := unmarshal((\*plain)(c))  
 **if** err != nil {  
 **return** err  
 }  
 **if** err := checkOverflow(c.XXX, "file\_sd\_config"); err != nil {  
 **return** err  
 }  
 **if** len(c.Files) == 0 {  
 **return** fmt.Errorf("file service discovery config must contain at least one path name")  
 }  
 **for** \_, name := **range** c.Files {  
 **if** !patFileSDName.MatchString(name) {  
 **return** fmt.Errorf("path name %q is not valid for file discovery", name)  
 }  
 }  
 **return** nil  
}

# Discovery结构体

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

// Discovery provides service discovery functionality based  
// on files that contain target groups in JSON or YAML format. Refreshing  
// happens using file watches and periodic refreshes.  
**type** Discovery **struct** {  
 paths []string  
 watcher \*fsnotify.Watcher  
 interval time.Duration  
  
 // lastRefresh stores which files were found during the last refresh  
 // and how many target groups they contained.  
 // This is used to detect deleted target groups.  
 lastRefresh **map**[string]int  
}

# file.go/Discovery.Run

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

Run方法实现了TargetProvider的接口方法

// Run implements the TargetProvider interface.  
**func** (d \*Discovery) Run(ctx context.Context, ch **chan**<- []\*config.TargetGroup) {  
 **defer** d.stop()  
  
 watcher, err := fsnotify.NewWatcher()  
 **if** err != nil {  
 log.Errorf("Error creating file watcher: %s", err)  
 **return** }  
 d.watcher = watcher  
  
 d.refresh(ctx, ch)  
  
 ticker := time.NewTicker(d.interval)  
 **defer** ticker.Stop()  
  
 **for** {  
 **select** {  
 **case** <-ctx.Done():  
 **return  
  
 case** event := <-d.watcher.Events:  
 // fsnotify sometimes sends a bunch of events without name or operation.  
 // It's unclear what they are and why they are sent - filter them out.  
 **if** len(event.Name) == 0 {  
 **break** }  
 // Everything but a chmod requires rereading.  
 **if** event.Op^fsnotify.*Chmod* == 0 {  
 **break** }  
 // Changes to a file can spawn various sequences of events with  
 // different combinations of operations. For all practical purposes  
 // this is inaccurate.  
 // The most reliable solution is to reload everything if anything happens.  
 d.refresh(ctx, ch)  
  
 **case** <-ticker.C:  
 // Setting a new watch after an update might fail. Make sure we don't lose  
 // those files forever.  
 d.refresh(ctx, ch)  
  
 **case** err := <-d.watcher.Errors:  
 **if** err != nil {  
 log.Errorf("Error on file watch: %s", err)  
 }  
 }  
 }  
}

## file.go/Discovery.refresh()

// refresh reads all files matching the discovery's patterns and sends the respective  
// updated target groups through the channel.  
**func** (d \*Discovery) refresh(ctx context.Context, ch **chan**<- []\*config.TargetGroup) {  
 t0 := time.Now()  
 **defer func**() {  
 fileSDScanDuration.Observe(time.Since(t0).Seconds())  
 }()  
  
 ref := **map**[string]int{}  
 **for** \_, p := **range** d.listFiles() {  
 tgroups, err := readFile(p)  
 **if** err != nil {  
 fileSDReadErrorsCount.Inc()  
 log.Errorf("Error reading file %q: %s", p, err)  
 // Prevent deletion down below.  
 ref[p] = d.lastRefresh[p]  
 **continue** }  
 **select** {  
 **case** ch <- tgroups:  
 **case** <-ctx.Done():  
 **return** }  
  
 ref[p] = len(tgroups)  
 }  
 // Send empty updates for sources that disappeared.  
 **for** f, n := **range** d.lastRefresh {  
 m, ok := ref[f]  
 **if** !ok || n > m {  
 **for** i := m; i < n; i++ {  
 **select** {  
 **case** ch <- []\*config.TargetGroup{{Source: fileSource(f, i)}}:  
 **case** <-ctx.Done():  
 **return** }  
 }  
 }  
 }  
 d.lastRefresh = ref  
  
 d.watchFiles()  
}

### file.go/Discovery.listFiles()

// listFiles returns a list of all files that match the configured patterns.  
**func** (d \*Discovery) listFiles() []string {  
 **var** paths []string  
 **for** \_, p := **range** d.paths {

// 列出与指定的模式 pattern 完全匹配的文件或目录

files, err := filepath.Glob(p)  
 **if** err != nil {  
 log.Errorf("Error expanding glob %q: %s", p, err)  
 **continue** }  
 paths = append(paths, files...)  
 }  
 **return** paths  
}

### file.go/readFile()解析json和yaml文件

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

// readFile reads a JSON or YAML list of targets groups from the file, depending on its  
// file extension. It returns full configuration target groups.  
**func** readFile(filename string) ([]\*config.TargetGroup, error) {  
 content, err := ioutil.ReadFile(filename)  
 **if** err != nil {  
 **return** nil, err  
 }  
  
 **var** targetGroups []\*config.TargetGroup  
  
 **switch** ext := filepath.Ext(filename); strings.ToLower(ext) {  
 **case** ".json":  
 **if** err := json.Unmarshal(content, &targetGroups); err != nil {  
 **return** nil, err  
 }  
 **case** ".yml", ".yaml":  
 **if** err := yaml.Unmarshal(content, &targetGroups); err != nil {  
 **return** nil, err  
 }  
 **default**:  
 panic(fmt.Errorf("retrieval.FileDiscovery.readFile: unhandled file extension %q", ext))  
 }  
  
 **for** i, tg := **range** targetGroups {  
 tg.Source = fileSource(filename, i)  
 **if** tg.Labels == nil {  
 tg.Labels = model.LabelSet{}  
 }  
 tg.Labels[*fileSDFilepathLabel*] = model.LabelValue(filename)  
 }  
 **return** targetGroups, nil  
}

### file.go/Discovery.watchFiles()

// watchFiles sets watches on all full paths or directories that were configured for  
// this file discovery.  
**func** (d \*Discovery) watchFiles() {  
 **if** d.watcher == nil {  
 panic("no watcher configured")  
 }  
 **for** \_, p := **range** d.paths {  
 **if** idx := strings.LastIndex(p, "/"); idx > -1 {  
 p = p[:idx]  
 } **else** {  
 p = "./"  
 }  
 **if** err := d.watcher.Add(p); err != nil {  
 log.Errorf("Error adding file watch for %q: %s", p, err)  
 }  
 }  
}

# Main

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

**go** targetManager.Run()

## TargetManger.Run()

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

// Run starts background processing to handle target updates.  
**func** (tm \*TargetManager) Run() {  
 log.Info("Starting target manager...")  
  
 tm.mtx.Lock()  
  
 tm.ctx, tm.cancel = context.WithCancel(context.Background())  
 tm.reload()  
  
 tm.mtx.Unlock()  
  
 tm.wg.Wait()  
}

TargetManager.Run() -> tm.reload() ->

## targetManager.go/TargetManager.reload()

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

**func** (tm \*TargetManager) reload() {  
 jobs := **map**[string]**struct**{}{}  
  
 // Start new target sets and update existing ones.  
 **for** \_, scfg := **range** tm.scrapeConfigs {  
 jobs[scfg.JobName] = **struct**{}{}  
//如果查询的键出现在映射里面，第一个用来获得键对应的值，第二个是布尔类型表示存不存在

//如果在配置中查找不到对应的键则，使用discovery重新加载键

//否则直接抓取监控数据ts.sp.reload(scfg)

ts, ok := tm.targetSets[scfg.JobName]  
 **if** !ok {  
 ctx, cancel := context.WithCancel(tm.ctx)  
 ts = &targetSet{  
 ctx: ctx,  
 cancel: cancel,  
 sp: newScrapePool(ctx, scfg, tm.appender),  
 }  
 ts.ts = discovery.NewTargetSet(ts.sp)  
  
 tm.targetSets[scfg.JobName] = ts  
  
 tm.wg.Add(1)  
  
 **go func**(ts \*targetSet) {  
 // Run target set, which blocks until its context is canceled.  
 // Gracefully shut down pending scrapes in the scrape pool afterwards.  
 ts.ts.Run(ctx)  
 ts.sp.stop()  
 tm.wg.Done()  
 }(ts)  
 } **else** {  
 ts.sp.reload(scfg)  
 }  
 ts.ts.UpdateProviders(discovery.ProvidersFromConfig(scfg.ServiceDiscoveryConfig))  
 }  
  
 // Remove old target sets. Waiting for scrape pools to complete pending  
 // scrape inserts is already guaranteed by the goroutine that started the target set.  
 **for** name, ts := **range** tm.targetSets {  
 **if** \_, ok := jobs[name]; !ok {  
 ts.cancel()  
 delete(tm.targetSets, name)  
 }  
 }  
}

TargetManager.reload()->ts.ts.Run(ctx)-> ts.updateProviders(ctx, p) ->

TargetManager.reload()->discovery.ProvidersFromConfig(scfg.ServiceDiscoveryConfig)

### ts.ts = discovery.NewTargetSet(ts.sp)

#### discovery.go/NewTargetSet()

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

// NewTargetSet returns a new target sending TargetGroups to the Syncer.  
**func** NewTargetSet(s Syncer) \*TargetSet {  
 **return** &TargetSet{  
 syncCh: make(**chan struct**{}, 1),  
 providerCh: make(**chan map**[string]TargetProvider),  
 syncer: s,  
 }  
}

#### discovery.go/TargetSet.Run()读取providerCh

ts.updateProviders(ctx, p)

// Run starts the processing of target providers and their updates.  
// It blocks until the context gets canceled.  
**func** (ts \*TargetSet) Run(ctx context.Context) {  
Loop:  
 **for** {  
 // Throttle syncing to once per five seconds.  
 **select** {  
 **case** <-ctx.Done():  
 **break** Loop  
 **case** p := <-ts.providerCh:  
 ts.updateProviders(ctx, p)  
 **case** <-time.After(5 \* time.*Second*):  
 }  
  
 **select** {  
 **case** <-ctx.Done():  
 **break** Loop  
 **case** <-ts.syncCh:  
 ts.sync()  
 **case** p := <-ts.providerCh:  
 ts.updateProviders(ctx, p)  
 }  
 }  
}

#### discovery.go/TargeSet.updateProviders()

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

**func** (ts \*TargetSet) updateProviders(ctx context.Context, providers **map**[string]TargetProvider) {  
  
 // Stop all previous target providers of the target set.  
 **if** ts.cancelProviders != nil {  
 ts.cancelProviders()  
 }  
 ctx, ts.cancelProviders = context.WithCancel(ctx)  
  
 **var** wg sync.WaitGroup  
 // (Re-)create a fresh tgroups map to not keep stale targets around. We  
 // will retrieve all targets below anyway, so cleaning up everything is  
 // safe and doesn't inflict any additional cost.  
 ts.mtx.Lock()  
 ts.tgroups = **map**[string]\*config.TargetGroup{}  
 ts.mtx.Unlock()  
  
 **for** name, prov := **range** providers {  
 wg.Add(1)  
  
 updates := make(**chan** []\*config.TargetGroup)  
 **go** prov.Run(ctx, updates)  
  
 **go func**(name string, prov TargetProvider) {  
 **select** {  
 **case** <-ctx.Done():  
 **case** initial, ok := <-updates:  
 // Handle the case that a target provider exits and closes the channel  
 // before the context is done.  
 **if** !ok {  
 **break** }  
 // First set of all targets the provider knows.  
 **for** \_, tgroup := **range** initial {  
 ts.setTargetGroup(name, tgroup)  
 }  
 **case** <-time.After(5 \* time.*Second*):  
 // Initial set didn't arrive. Act as if it was empty  
 // and wait for updates later on.  
 }  
 wg.Done()  
  
 // Start listening for further updates.  
 **for** {  
 **select** {  
 **case** <-ctx.Done():  
 **return  
 case** tgs, ok := <-updates:  
 // Handle the case that a target provider exits and closes the channel  
 // before the context is done.  
 **if** !ok {  
 **return** }  
 **for** \_, tg := **range** tgs {  
 ts.update(name, tg)  
 }  
 }  
 }  
 }(name, prov)  
 }  
  
 // We wait for a full initial set of target groups before releasing the mutex  
 // to ensure the initial sync is complete and there are no races with subsequent updates.  
 wg.Wait()  
 // Just signal that there are initial sets to sync now. Actual syncing must only  
 // happen in the runScraping loop.  
 **select** {  
 **case** ts.syncCh <- **struct**{}{}:  
 **default**:  
 }  
}

#### discovery.go/RargeProvider.Run()

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

// A TargetProvider provides information about target groups. It maintains a set  
// of sources from which TargetGroups can originate. Whenever a target provider  
// detects a potential change, it sends the TargetGroup through its provided channel.  
//  
// The TargetProvider does not have to guarantee that an actual change happened.  
// It does guarantee that it sends the new TargetGroup whenever a change happens.  
//  
// TargetProviders should initially send a full set of all discoverable TargetGroups.  
**type** TargetProvider **interface** {  
 // Run hands a channel to the target provider through which it can send  
 // updated target groups.  
 // Must returns if the context gets canceled. It should not close the update  
 // channel on returning.  
 Run(ctx context.Context, up **chan**<- []\*config.TargetGroup)  
}

### ts.ts.UpdateProviders(discovery.ProvidersFromConfig(scfg.ServiceDiscoveryConfig))

#### discovery.go/UpdateProviders写入providerCh

// UpdateProviders sets new target providers for the target set.  
**func** (ts \*TargetSet) UpdateProviders(p **map**[string]TargetProvider) {  
 ts.providerCh <- p  
}

#### discovery.go/ProvidersFromConfig()

E:\workspace\go\prometheus\prometheus\discovery\discovery.go

**for** i, c := **range** cfg.FileSDConfigs {  
 app("file", i, file.NewDiscovery(c))  
}

#### file.go/NewDiscovery()

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

// NewDiscovery returns a new file discovery for the given paths.  
**func** NewDiscovery(conf \*config.FileSDConfig) \*Discovery {  
 **return** &Discovery{  
 paths: conf.Files,  
 interval: time.Duration(conf.RefreshInterval),  
 }  
}

#### config.go/FileSDConfig

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

// FileSDConfig is the configuration for file based discovery.  
**type** FileSDConfig **struct** {  
 Files []string `yaml:"files"`  
 RefreshInterval model.Duration `yaml:"refresh\_interval,omitempty"`  
  
 // Catches all undefined fields and must be empty after parsing.  
 XXX **map**[string]**interface**{} `yaml:",inline"`  
}

# udhos/equalfile: Go package to compare files

https://github.com/udhos/equalfile

https://github.com/udhos/equalfile.git