# discovery\kubernetes\kubernetes.go

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

## init

**func** init() {  
 runtime.ErrorHandlers = []**func**(error){  
 **func**(err error) {  
 log.With("component", "kube\_client\_runtime").Errorln(err)  
 },  
 }  
}

[Go](http://lib.csdn.net/base/go)里面有两个保留的函数：init函数和main函数。

相同点：两个函数在定义时不能有任何的参数和返回值，且Go程序自动调用。

不同点：init可以应用于任意包中，且可以重复定义多个。main函数只能用于main包中，且只能定义一个。

下边说一下两个函数的执行顺序：

对同一个go文件的init()调用顺序是从上到下的  
对同一个package中不同文件是按文件名字符串比较“从小到大”顺序调用各文件中的init()函数,对于对不同的package，如果不相互依赖的话，按照main包中"先import的后调用"的顺序调用其包中的init()

如果package存在依赖，则先调用最早被依赖的package中的init()  
最后调用main函数

下图截自astaxie的《Go Web 编程》

ps：如果init函数中使用了println或者print 你会发现在执行过程中这两个不会按照你想象中的顺序执行。这两个函数官方只推荐在[测试](http://lib.csdn.net/base/softwaretest)环境中使用，对于正式环境不要使用。

转自http://studygolang.com/articles/3873

## Discovery.Run

// Run implements the TargetProvider interface.  
**func** (d \*Discovery) Run(ctx context.Context, ch **chan**<- []\*config.TargetGroup) {  
 rclient := d.client.Core().RESTClient()  
  
 namespaces := d.getNamespaces()  
  
 **switch** d.role {  
 **case** "endpoints":  
 **var** wg sync.WaitGroup  
  
 **for** \_, namespace := **range** namespaces {  
 elw := cache.NewListWatchFromClient(rclient, "endpoints", namespace, nil)  
 slw := cache.NewListWatchFromClient(rclient, "services", namespace, nil)  
 plw := cache.NewListWatchFromClient(rclient, "pods", namespace, nil)  
 eps := NewEndpoints(  
 d.logger.With("kubernetes\_sd", "endpoint"),  
 cache.NewSharedInformer(slw, &apiv1.Service{}, *resyncPeriod*),  
 cache.NewSharedInformer(elw, &apiv1.Endpoints{}, *resyncPeriod*),  
 cache.NewSharedInformer(plw, &apiv1.Pod{}, *resyncPeriod*),  
 )  
 **go** eps.endpointsInf.Run(ctx.Done())  
 **go** eps.serviceInf.Run(ctx.Done())  
 **go** eps.podInf.Run(ctx.Done())  
  
 **for** !eps.serviceInf.HasSynced() {  
 time.Sleep(100 \* time.*Millisecond*)  
 }  
 **for** !eps.endpointsInf.HasSynced() {  
 time.Sleep(100 \* time.*Millisecond*)  
 }  
 **for** !eps.podInf.HasSynced() {  
 time.Sleep(100 \* time.*Millisecond*)  
 }  
 wg.Add(1)  
 **go func**() {  
 **defer** wg.Done()  
 eps.Run(ctx, ch)  
 }()  
 }  
 wg.Wait()  
 **case** "pod":  
 **var** wg sync.WaitGroup  
 **for** \_, namespace := **range** namespaces {  
 plw := cache.NewListWatchFromClient(rclient, "pods", namespace, nil)  
 pod := NewPod(  
 d.logger.With("kubernetes\_sd", "pod"),  
 cache.NewSharedInformer(plw, &apiv1.Pod{}, *resyncPeriod*),  
 )  
 **go** pod.informer.Run(ctx.Done())  
  
 **for** !pod.informer.HasSynced() {  
 time.Sleep(100 \* time.*Millisecond*)  
 }  
 wg.Add(1)  
 **go func**() {  
 **defer** wg.Done()  
 pod.Run(ctx, ch)  
 }()  
 }  
 wg.Wait()  
 **case** "service":  
 **var** wg sync.WaitGroup  
 **for** \_, namespace := **range** namespaces {  
 slw := cache.NewListWatchFromClient(rclient, "services", namespace, nil)  
 svc := NewService(  
 d.logger.With("kubernetes\_sd", "service"),  
 cache.NewSharedInformer(slw, &apiv1.Service{}, *resyncPeriod*),  
 )  
 **go** svc.informer.Run(ctx.Done())  
  
 **for** !svc.informer.HasSynced() {  
 time.Sleep(100 \* time.*Millisecond*)  
 }  
 wg.Add(1)  
 **go func**() {  
 **defer** wg.Done()  
 svc.Run(ctx, ch)  
 }()  
 }  
 wg.Wait()  
 **case** "node":  
 nlw := cache.NewListWatchFromClient(rclient, "nodes", api.*NamespaceAll*, nil)  
 node := NewNode(  
 d.logger.With("kubernetes\_sd", "node"),  
 cache.NewSharedInformer(nlw, &apiv1.Node{}, *resyncPeriod*),  
 )  
 **go** node.informer.Run(ctx.Done())  
  
 **for** !node.informer.HasSynced() {  
 time.Sleep(100 \* time.*Millisecond*)  
 }  
 node.Run(ctx, ch)  
  
 **default**:  
 d.logger.Errorf("unknown Kubernetes discovery kind %q", d.role)  
 }  
  
 <-ctx.Done()  
}

### 调用

**case** "node":  
 nlw := cache.NewListWatchFromClient(rclient, "nodes", api.*NamespaceAll*, nil)  
 node := NewNode(  
 d.logger.With("kubernetes\_sd", "node"),  
 cache.NewSharedInformer(nlw, &apiv1.Node{}, *resyncPeriod*),  
 )  
 **go** node.informer.Run(ctx.Done())  
  
 **for** !node.informer.HasSynced() {  
 time.Sleep(100 \* time.*Millisecond*)  
 }  
 node.Run(ctx, ch)

#### NewListWatchFromClient

//创建ListerWatcher

nlw := cache.NewListWatchFromClient(rclient, "nodes", api.*NamespaceAll*, nil)

#### NewSharedInformer

//创建informer, sharedIndexInformation

cache.NewSharedInformer(nlw, &apiv1.Node{}, *resyncPeriod*),

# k8s.io\client-go\tools\cache\listwatch.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\github.com\prometheus\prometheus\vendor\k8s.io\client-go\tools\cache\listwatch.go

## Listwatch.go.NewListWatchFromClient

// NewListWatchFromClient creates a new ListWatch from the specified client, resource, namespace and field selector.  
**func** NewListWatchFromClient(c Getter, resource string, namespace string, fieldSelector fields.Selector) \*ListWatch {  
 listFunc := **func**(options metav1.ListOptions) (runtime.Object, error) {  
 **return** c.Get().  
 Namespace(namespace).  
 Resource(resource).  
 VersionedParams(&options, metav1.ParameterCodec).  
 FieldsSelectorParam(fieldSelector).  
 Do().  
 Get()  
 }  
 watchFunc := **func**(options metav1.ListOptions) (watch.Interface, error) {  
 options.Watch = true  
 **return** c.Get().  
 Namespace(namespace).  
 Resource(resource).  
 VersionedParams(&options, metav1.ParameterCodec).  
 FieldsSelectorParam(fieldSelector).  
 Watch()  
 }  
 **return** &ListWatch{ListFunc: listFunc, WatchFunc: watchFunc}  
}

### 调用

#### Request.go Do

**return** c.Get().  
 Namespace(namespace).  
 Resource(resource).  
 VersionedParams(&options, metav1.ParameterCodec).  
 FieldsSelectorParam(fieldSelector)  
 Do().  
 Get()

# k8s.io\client-go\rest\request.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\github.com\prometheus\prometheus\vendor\k8s.io\client-go\rest\request.go

## Request.Do

// Do formats and executes the request. Returns a Result object for easy response  
// processing.  
//  
// Error type:  
// \* If the request can't be constructed, or an error happened earlier while building its  
// arguments: \*RequestConstructionError  
// \* If the server responds with a status: \*errors.StatusError or \*errors.UnexpectedObjectError  
// \* http.Client.Do errors are returned directly.  
**func** (r \*Request) Do() Result {  
 r.tryThrottle()  
  
 **var** result Result  
 err := r.request(**func**(req \*http.Request, resp \*http.Response) {  
 result = r.transformResponse(resp, req)  
 })  
 **if** err != nil {  
 **return** Result{err: err}  
 }  
 **return** result  
}

### 调用

#### Request.go request

err := r.request(**func**(req \*http.Request, resp \*http.Response) {

## Request.request

// request connects to the server and invokes the provided function when a server response is  
// received. It handles retry behavior and up front validation of requests. It will invoke  
// fn at most once. It will return an error if a problem occurred prior to connecting to the  
// server - the provided function is responsible for handling server errors.  
**func** (r \*Request) request(fn **func**(\*http.Request, \*http.Response)) error {  
 //Metrics for total request latency  
 start := time.Now()  
 **defer func**() {  
 metrics.RequestLatency.Observe(r.verb, r.finalURLTemplate(), time.Since(start))  
 }()  
  
 **if** r.err != nil {  
 glog.V(4).Infof("Error in request: %v", r.err)  
 **return** r.err  
 }  
  
 // *TODO: added to catch programmer errors (invoking operations with an object with an empty namespace)* **if** (r.verb == "GET" || r.verb == "PUT" || r.verb == "DELETE") && r.namespaceSet && len(r.resourceName) > 0 && len(r.namespace) == 0 {  
 **return** fmt.Errorf("an empty namespace may not be set when a resource name is provided")  
 }  
 **if** (r.verb == "POST") && r.namespaceSet && len(r.namespace) == 0 {  
 **return** fmt.Errorf("an empty namespace may not be set during creation")  
 }  
  
 client := r.client  
 **if** client == nil {  
 client = http.DefaultClient  
 }  
  
 // Right now we make about ten retry attempts if we get a Retry-After response.  
 // *TODO: Change to a timeout based approach.* maxRetries := 10  
 retries := 0  
 **for** {  
 url := r.URL().String()  
 req, err := http.NewRequest(r.verb, url, r.body)  
 **if** err != nil {  
 **return** err  
 }  
 **if** r.ctx != nil {  
 req = req.WithContext(r.ctx)  
 }  
 req.Header = r.headers  
  
 r.backoffMgr.Sleep(r.backoffMgr.CalculateBackoff(r.URL()))  
 **if** retries > 0 {  
 // We are retrying the request that we already send to apiserver  
 // at least once before.  
 // This request should also be throttled with the client-internal throttler.  
 r.tryThrottle()  
 }  
 resp, err := client.Do(req)  
 updateURLMetrics(r, resp, err)  
 **if** err != nil {  
 r.backoffMgr.UpdateBackoff(r.URL(), err, 0)  
 } **else** {  
 r.backoffMgr.UpdateBackoff(r.URL(), err, resp.StatusCode)  
 }  
 **if** err != nil {  
 // "Connection reset by peer" is usually a transient error.  
 // Thus in case of "GET" operations, we simply retry it.  
 // We are not automatically retrying "write" operations, as  
 // they are not idempotent.  
 **if** !net.IsConnectionReset(err) || r.verb != "GET" {  
 **return** err  
 }  
 // For the purpose of retry, we set the artificial "retry-after" response.  
 // *TODO: Should we clean the original response if it exists?* resp = &http.Response{  
 StatusCode: http.*StatusInternalServerError*,  
 Header: http.Header{"Retry-After": []string{"1"}},  
 Body: ioutil.NopCloser(bytes.NewReader([]byte{})),  
 }  
 }  
  
 done := **func**() bool {  
 // Ensure the response body is fully read and closed  
 // before we reconnect, so that we reuse the same TCP  
 // connection.  
 **defer func**() {  
 **const** *maxBodySlurpSize* = 2 << 10  
 **if** resp.ContentLength <= *maxBodySlurpSize* {  
 io.Copy(ioutil.Discard, &io.LimitedReader{R: resp.Body, N: *maxBodySlurpSize*})  
 }  
 resp.Body.Close()  
 }()  
  
 retries++  
 **if** seconds, wait := checkWait(resp); wait && retries < maxRetries {  
 **if** seeker, ok := r.body.(io.Seeker); ok && r.body != nil {  
 \_, err := seeker.Seek(0, 0)  
 **if** err != nil {  
 glog.V(4).Infof("Could not retry request, can't Seek() back to beginning of body for %T", r.body)  
 fn(req, resp)  
 **return** true  
 }  
 }  
  
 glog.V(4).Infof("Got a Retry-After %s response for attempt %d to %v", seconds, retries, url)  
 r.backoffMgr.Sleep(time.Duration(seconds) \* time.*Second*)  
 **return** false  
 }  
 fn(req, resp)  
 **return** true  
 }()  
 **if** done {  
 **return** nil  
 }  
 }  
}

# k8s.io\client-go\tools\cache\shared\_informer.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\github.com\prometheus\prometheus\vendor\k8s.io\client-go\tools\cache\shared\_informer.go

## shared\_informer.go.NewSharedInformer

// NewSharedInformer creates a new instance for the listwatcher.  
**func** NewSharedInformer(lw ListerWatcher, objType runtime.Object, resyncPeriod time.Duration) SharedInformer {  
 **return** NewSharedIndexInformer(lw, objType, resyncPeriod, Indexers{})  
}  
  
// NewSharedIndexInformer creates a new instance for the listwatcher.  
**func** NewSharedIndexInformer(lw ListerWatcher, objType runtime.Object, defaultEventHandlerResyncPeriod time.Duration, indexers Indexers) SharedIndexInformer {  
 realClock := &clock.RealClock{}  
 sharedIndexInformer := &sharedIndexInformer{  
 processor: &sharedProcessor{clock: realClock},  
 indexer: NewIndexer(DeletionHandlingMetaNamespaceKeyFunc, indexers),  
 listerWatcher: lw,  
 objectType: objType,  
 resyncCheckPeriod: defaultEventHandlerResyncPeriod,  
 defaultEventHandlerResyncPeriod: defaultEventHandlerResyncPeriod,  
 cacheMutationDetector: NewCacheMutationDetector(fmt.Sprintf("%T", objType)),  
 clock: realClock,  
 }  
 **return** sharedIndexInformer  
}

## sharedIndexInformer.Run

**func** (s \*sharedIndexInformer) Run(stopCh <-**chan struct**{}) {  
 **defer** utilruntime.HandleCrash()  
  
 fifo := NewDeltaFIFO(MetaNamespaceKeyFunc, nil, s.indexer)  
  
 cfg := &Config{  
 Queue: fifo,  
 ListerWatcher: s.listerWatcher,  
 ObjectType: s.objectType,  
 FullResyncPeriod: s.resyncCheckPeriod,  
 RetryOnError: false,  
 ShouldResync: s.processor.shouldResync,  
  
 Process: s.HandleDeltas,  
 }  
  
 **func**() {  
 s.startedLock.Lock()  
 **defer** s.startedLock.Unlock()  
  
 s.controller = New(cfg)  
 s.controller.(\*controller).clock = s.clock  
 s.started = true  
 }()  
  
 s.stopCh = stopCh  
 s.cacheMutationDetector.Run(stopCh)  
 s.processor.run(stopCh)  
 s.controller.Run(stopCh)  
}

### 调用

#### controller.Run

//调用controller.go controller.Run

s.controller.Run(stopCh)

//新增controller

s.controller = New(cfg)

// New makes a new Controller from the given Config.  
**func** New(c \*Config) Controller {  
 ctlr := &controller{  
 config: \*c,  
 clock: &clock.RealClock{},  
 }  
 **return** ctlr  
}

# k8s.io\client-go\tools\cache\controller.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\github.com\prometheus\prometheus\vendor\k8s.io\client-go\tools\cache\controller.go

## controller.Run

// Run begins processing items, and will continue until a value is sent down stopCh.  
// It's an error to call Run more than once.  
// Run blocks; call via go.  
**func** (c \*controller) Run(stopCh <-**chan struct**{}) {  
 **defer** utilruntime.HandleCrash()  
 **go func**() {  
 <-stopCh  
 c.config.Queue.Close()  
 }()  
 r := NewReflector(  
 c.config.ListerWatcher,  
 c.config.ObjectType,  
 c.config.Queue,  
 c.config.FullResyncPeriod,  
 )  
 r.ShouldResync = c.config.ShouldResync  
 r.clock = c.clock  
  
 c.reflectorMutex.Lock()  
 c.reflector = r  
 c.reflectorMutex.Unlock()  
  
 r.RunUntil(stopCh)  
  
 wait.Until(c.processLoop, time.*Second*, stopCh)  
}

### 调用

#### RunUntil

r.RunUntil(stopCh)

# k8s.io\client-go\tools\cache\reflector.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\github.com\prometheus\prometheus\vendor\k8s.io\client-go\tools\cache\reflector.go

## Reflector.RunUntil

// RunUntil starts a watch and handles watch events. Will restart the watch if it is closed.  
// RunUntil starts a goroutine and returns immediately. It will exit when stopCh is closed.  
**func** (r \*Reflector) RunUntil(stopCh <-**chan struct**{}) {  
 glog.V(3).Infof("Starting reflector %v (%s) from %s", r.expectedType, r.resyncPeriod, r.name)  
 **go** wait.Until(**func**() {  
 **if** err := r.ListAndWatch(stopCh); err != nil {  
 utilruntime.HandleError(err)  
 }  
 }, r.period, stopCh)  
}

### 调用

#### ListAndWatch

**if** err := r.ListAndWatch(stopCh); err != nil {

## Reflector.ListAndWatch

// ListAndWatch first lists all items and get the resource version at the moment of call,  
// and then use the resource version to watch.  
// It returns error if ListAndWatch didn't even try to initialize watch.  
**func** (r \*Reflector) ListAndWatch(stopCh <-**chan struct**{}) error {  
 glog.V(3).Infof("Listing and watching %v from %s", r.expectedType, r.name)  
 **var** resourceVersion string  
 resyncCh, cleanup := r.resyncChan()  
 **defer** cleanup()  
  
 // Explicitly set "0" as resource version - it's fine for the List()  
 // to be served from cache and potentially be delayed relative to  
 // etcd contents. Reflector framework will catch up via Watch() eventually.  
 options := metav1.ListOptions{ResourceVersion: "0"}  
 list, err := r.listerWatcher.List(options)  
 **if** err != nil {  
 **return** fmt.Errorf("%s: Failed to list %v: %v", r.name, r.expectedType, err)  
 }  
 listMetaInterface, err := meta.ListAccessor(list)  
 **if** err != nil {  
 **return** fmt.Errorf("%s: Unable to understand list result %#v: %v", r.name, list, err)  
 }  
 resourceVersion = listMetaInterface.GetResourceVersion()  
 items, err := meta.ExtractList(list)  
 **if** err != nil {  
 **return** fmt.Errorf("%s: Unable to understand list result %#v (%v)", r.name, list, err)  
 }  
 **if** err := r.syncWith(items, resourceVersion); err != nil {  
 **return** fmt.Errorf("%s: Unable to sync list result: %v", r.name, err)  
 }  
 r.setLastSyncResourceVersion(resourceVersion)  
  
 resyncerrc := make(**chan** error, 1)  
 cancelCh := make(**chan struct**{})  
 **defer** close(cancelCh)  
 **go func**() {  
 **for** {  
 **select** {  
 **case** <-resyncCh:  
 **case** <-stopCh:  
 **return  
 case** <-cancelCh:  
 **return** }  
 **if** r.ShouldResync == nil || r.ShouldResync() {  
 glog.V(4).Infof("%s: forcing resync", r.name)  
 **if** err := r.store.Resync(); err != nil {  
 resyncerrc <- err  
 **return** }  
 }  
 cleanup()  
 resyncCh, cleanup = r.resyncChan()  
 }  
 }()  
  
 **for** {  
 timemoutseconds := int64(minWatchTimeout.Seconds() \* (rand.Float64() + 1.0))  
 options = metav1.ListOptions{  
 ResourceVersion: resourceVersion,  
 // We want to avoid situations of hanging watchers. Stop any wachers that do not  
 // receive any events within the timeout window.  
 TimeoutSeconds: &timemoutseconds,  
 }  
  
 w, err := r.listerWatcher.Watch(options)  
 **if** err != nil {  
 **switch** err {  
 **case** io.EOF:  
 // watch closed normally  
 **case** io.ErrUnexpectedEOF:  
 glog.V(1).Infof("%s: Watch for %v closed with unexpected EOF: %v", r.name, r.expectedType, err)  
 **default**:  
 utilruntime.HandleError(fmt.Errorf("%s: Failed to watch %v: %v", r.name, r.expectedType, err))  
 }  
 // If this is "connection refused" error, it means that most likely apiserver is not responsive.  
 // It doesn't make sense to re-list all objects because most likely we will be able to restart  
 // watch where we ended.  
 // If that's the case wait and resend watch request.  
 **if** urlError, ok := err.(\*url.Error); ok {  
 **if** opError, ok := urlError.Err.(\*net.OpError); ok {  
 **if** errno, ok := opError.Err.(syscall.Errno); ok && errno == syscall.*ECONNREFUSED* {  
 time.Sleep(time.*Second*)  
 **continue** }  
 }  
 }  
 **return** nil  
 }  
  
 **if** err := r.watchHandler(w, &resourceVersion, resyncerrc, stopCh); err != nil {  
 **if** err != errorStopRequested {  
 glog.Warningf("%s: watch of %v ended with: %v", r.name, r.expectedType, err)  
 }  
 **return** nil  
 }  
 }  
}

### 问题

#### listerWatcher的初始化

list, err := r.listerWatcher.List(options)

//controller.Run指定了是从config中获取ListerWatcher

r := NewReflector(  
 c.config.ListerWatcher,  
 c.config.ObjectType,  
 c.config.Queue,  
 c.config.FullResyncPeriod,  
)

//controller.go中New指定了config

// New makes a new Controller from the given Config.  
**func** New(c \*Config) Controller {  
 ctlr := &controller{  
 config: \*c,  
 clock: &clock.RealClock{},  
 }  
 **return** ctlr  
}

//cfg使用s.listerWatcher构建ListerWatcher

cfg := &Config{  
 Queue: fifo,  
 ListerWatcher: s.listerWatcher,  
 ObjectType: s.objectType,  
 FullResyncPeriod: s.resyncCheckPeriod,  
 RetryOnError: false,  
 ShouldResync: s.processor.shouldResync,  
  
 Process: s.HandleDeltas,  
}

//kubernetes.go 中Disvovery.Run创建ListerWatcher

nlw := cache.NewListWatchFromClient(rclient, "nodes", api.*NamespaceAll*, nil)

# https get auth k8s api server

**func** main() {  
 tr := &http.Transport{  
 TLSClientConfig: &tls.Config{InsecureSkipVerify: true},  
 }  
 client := &http.Client{Transport: tr}  
 response, err := client.Get("https://admin:123456@192.168.31.221:6443/")  
 **if** err != nil {  
 fmt.Printf("%s", err)  
 os.Exit(1)  
 } **else** {  
 **defer** response.Body.Close()  
 contents, err := ioutil.ReadAll(response.Body)  
 **if** err != nil {  
 fmt.Printf("%s", err)  
 os.Exit(1)  
 }  
 fmt.Printf("%s\n", string(contents))  
 }  
}

# K8s.io代码

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

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\prometheus\vendor

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src

# x509: certificate signed by unknown authority

time="2017-06-07T12:11:31Z" level=error msg="k8s.io/client-go/1.5/tools/cache/reflector.go:109: Failed to list \*v1.Node: Get https://192.168.31.221:6443/api/v1/nodes?resourceVersion=0: x509: certificate signed by unknown authority" component="kube\_client\_runtime" source="kubernetes.go:73"

time="2017-06-07T16:32:50Z"

level=error msg="github.com/prometheus/prometheus/discovery/kubernetes/kubernetes.go:245:

Failed to list \*v1.Node:

Get <https://admin:123456@192.168.31.221:6443/api/v1/nodes?resourceVersion=0>:

x509: certificate signed by unknown authority" component="kube\_client\_runtime" source="kubernetes.go:75"

# client-go\1.5\tools\cache\reflector.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\k8s.io\client-go\1.5\tools\cache\reflector.go

## Reflector.ListAndWatch

/ ListAndWatch first lists all items and get the resource version at the moment of call,  
// and then use the resource version to watch.  
// It returns error if ListAndWatch didn't even try to initialize watch.  
**func** (r \*Reflector) ListAndWatch(stopCh <-**chan struct**{}) error {  
 glog.V(3).Infof("Listing and watching %v from %s", r.expectedType, r.name)  
 **var** resourceVersion string  
 resyncCh, cleanup := r.resyncChan()  
 **defer** cleanup()  
  
 // Explicitly set "0" as resource version - it's fine for the List()  
 // to be served from cache and potentially be delayed relative to  
 // etcd contents. Reflector framework will catch up via Watch() eventually.  
 options := api.ListOptions{ResourceVersion: "0"}  
 list, err := r.listerWatcher.List(options)  
 **if** err != nil {  
 **return** fmt.Errorf("%s: Failed to list %v: %v", r.name, r.expectedType, err)  
 }  
 listMetaInterface, err := meta.ListAccessor(list)  
 **if** err != nil {  
 **return** fmt.Errorf("%s: Unable to understand list result %#v: %v", r.name, list, err)  
 }  
 resourceVersion = listMetaInterface.GetResourceVersion()  
 items, err := meta.ExtractList(list)  
 **if** err != nil {  
 **return** fmt.Errorf("%s: Unable to understand list result %#v (%v)", r.name, list, err)  
 }  
 **if** err := r.syncWith(items, resourceVersion); err != nil {  
 **return** fmt.Errorf("%s: Unable to sync list result: %v", r.name, err)  
 }  
 r.setLastSyncResourceVersion(resourceVersion)  
  
 resyncerrc := make(**chan** error, 1)  
 **go func**() {  
 **for** {  
 **select** {  
 **case** <-resyncCh:  
 **case** <-stopCh:  
 **return** }  
 glog.V(4).Infof("%s: forcing resync", r.name)  
 **if** err := r.store.Resync(); err != nil {  
 resyncerrc <- err  
 **return** }  
 cleanup()  
 resyncCh, cleanup = r.resyncChan()  
 }  
 }()  
  
 **for** {  
 timemoutseconds := int64(minWatchTimeout.Seconds() \* (rand.Float64() + 1.0))  
 options = api.ListOptions{  
 ResourceVersion: resourceVersion,  
 // We want to avoid situations of hanging watchers. Stop any wachers that do not  
 // receive any events within the timeout window.  
 TimeoutSeconds: &timemoutseconds,  
 }  
  
 w, err := r.listerWatcher.Watch(options)  
 **if** err != nil {  
 **switch** err {  
 **case** io.EOF:  
 // watch closed normally  
 **case** io.ErrUnexpectedEOF:  
 glog.V(1).Infof("%s: Watch for %v closed with unexpected EOF: %v", r.name, r.expectedType, err)  
 **default**:  
 utilruntime.HandleError(fmt.Errorf("%s: Failed to watch %v: %v", r.name, r.expectedType, err))  
 }  
 // If this is "connection refused" error, it means that most likely apiserver is not responsive.  
 // It doesn't make sense to re-list all objects because most likely we will be able to restart  
 // watch where we ended.  
 // If that's the case wait and resend watch request.  
 **if** urlError, ok := err.(\*url.Error); ok {  
 **if** opError, ok := urlError.Err.(\*net.OpError); ok {  
 **if** errno, ok := opError.Err.(syscall.Errno); ok && errno == syscall.*ECONNREFUSED* {  
 time.Sleep(time.*Second*)  
 **continue** }  
 }  
 }  
 **return** nil  
 }  
  
 **if** err := r.watchHandler(w, &resourceVersion, resyncerrc, stopCh); err != nil {  
 **if** err != errorStopRequested {  
 glog.Warningf("%s: watch of %v ended with: %v", r.name, r.expectedType, err)  
 }  
 **return** nil  
 }  
 }

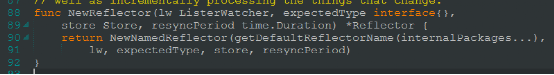
### 调用

#### r.listerWatcher.List(options)

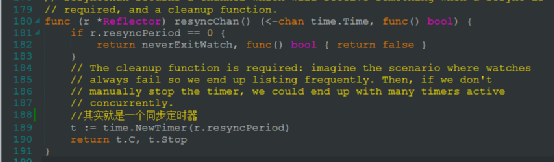
list, err := r.listerWatcher.List(options)

## 下层reflector工作流程

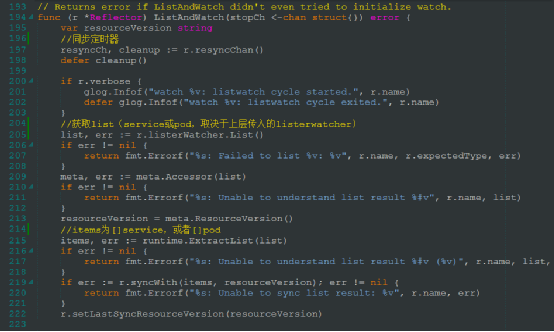
代码在k8s.io\kubernetes\pkg\client\cache\reflector.go



http://img.blog.csdn.net/20160330163105420



入口ListAndWatch



我们看看list返回的是什么

# kubernetes\pkg\client\services.go

E:\workspace\yh\OpenBridge-passos-proxy\open-falcon\src\k8s.io\kubernetes\pkg\client\services.go

## services.List

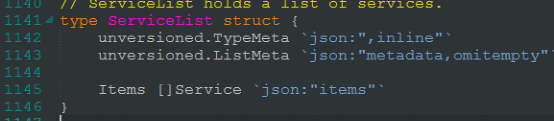
// List takes a selector, and returns the list of services that match that selector  
**func** (c \*services) List(selector labels.Selector) (result \*api.ServiceList, err error) {  
 result = &api.ServiceList{}  
 err = c.r.Get().  
 Namespace(c.ns).  
 Resource("services").  
 LabelsSelectorParam(selector).  
 Do().  
 Into(result)  
 **return**}

### 调用

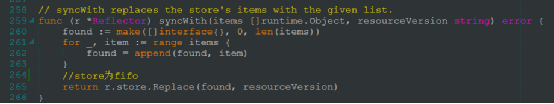
#### c.r.Get

err = c.r.Get().

代码在k8s.io\kubernetes\pkg\api\typess.go

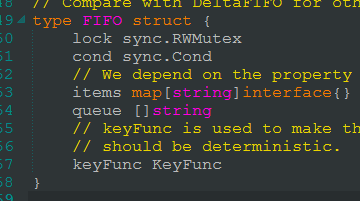


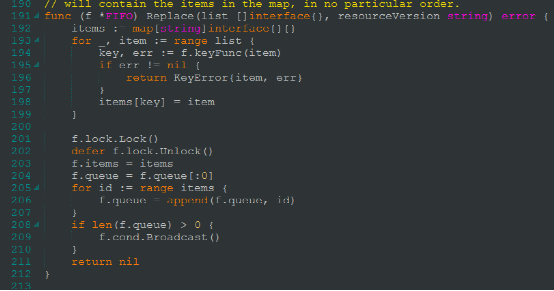
在此处插入一下对于list结果的处理



再插入下fifo的处理

代码在k8s.io\kubernetes\pkg\client\cache\fifo.go

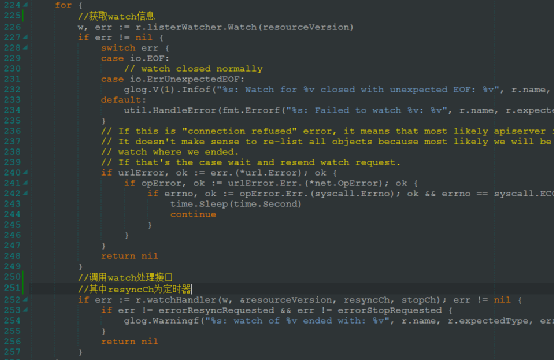




ok我们回到ListAndWatch，上面的list获取到后，直接全部更新掉fifo中的信息

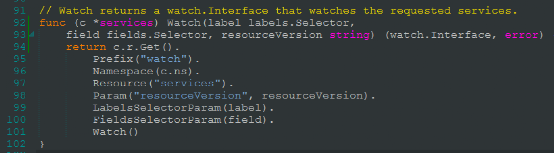
我们继续ListAndWatch

下面轮到watch信息的获取和处理

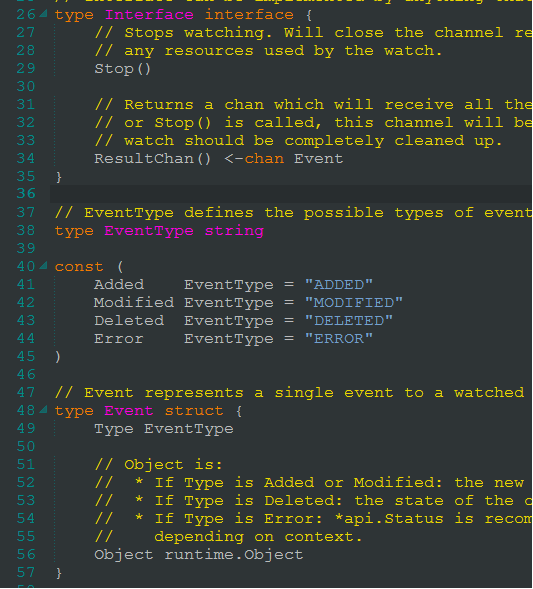


我们看看watch的返回是什么（我们以service的watch为例）

代码在k8s.io\kubernetes\pkg\client\unversioned\ services.go

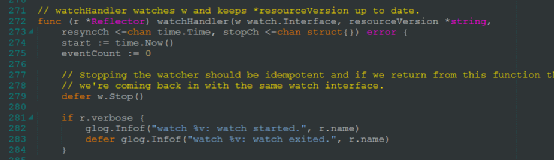


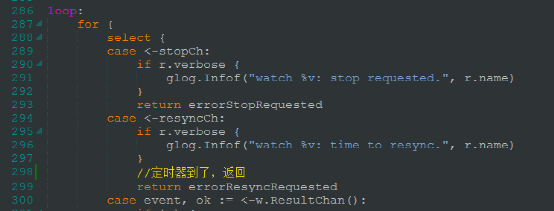
代码在k8s.io\kubernetes\pkg\watch\watch.go

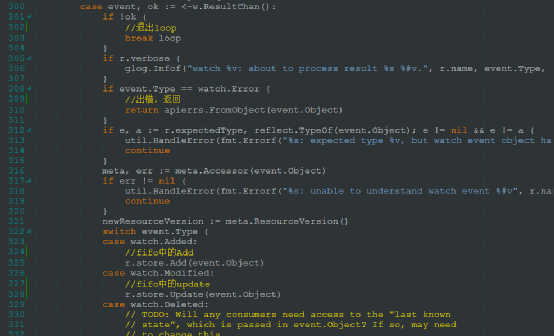


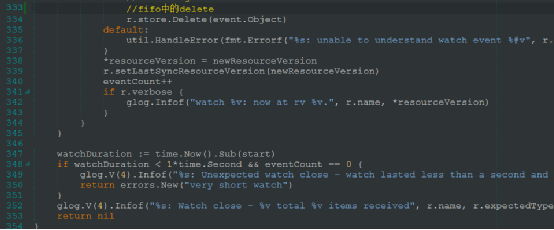
我们继续watchHandler的处理（函数比较长，贴了几张图）

下面的处理，其实就是从watch获取到的事件，全部添加到fifo中

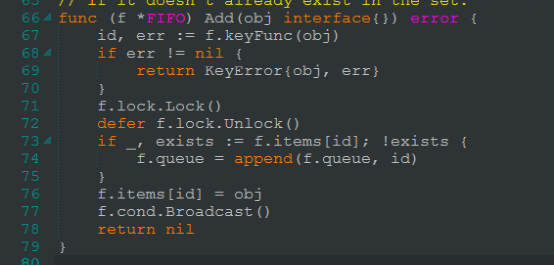


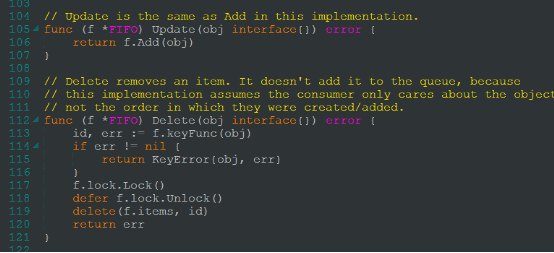






我们看看fifo中的add函数，update函数，delete函数





以上的loop会退出，但最顶层的reflector.Runutil会继续重复执行listAndwatch

以上便是reflector的工作流程

总结下，就是获取到list信息，然后更新掉store（fifo中的信息），然后watch获取到事件，然后根据不同的事件修改store（fifo）中的信息