- **main.go**

首先分析Updater的启动程序main.go

```
vertical-pod-autoscaler/pkg/updater/main.go
```

启动函数首先定义了初始变量，让我们来看一下：

```
var (
 updaterInterval = flag.Duration("updater-interval", 1*time.Minute,
 `How often updater should run`)
 // 启动频率

 minReplicas = flag.Int("min-replicas", 2,
 `Minimum number of replicas to perform update`)
 // 最少的replicas数量，执行更新

  evictionToleranceFraction = flag.Float64("eviction-tolerance", 0.5,
  `Fraction of replica count that can be evicted for update, if more than one pod can be evicted.`)
  // 驱逐临界线，一旦超过这个线（分数），这个pod将会被驱逐

  evictionRateLimit = flag.Float64("eviction-rate-limit", -1,
  `Number of pods that can be evicted per seconds. A rate limit set to 0 or -1 will disable
  the rate limiter.`)
  // 每秒能够被驱逐的pod数量

  evictionRateBurst = flag.Int("eviction-rate-burst", 1, `Burst of pods that can be evicted.`)
  // 能够被驱逐pod的并发上限

  address = flag.String("address", ":8943", "The address to expose Prometheus metrics.")
  // 普罗米修斯地址
)
```

最开始还是设置了变量，包括启动频率，副本replicas的最小数量，以及有关驱逐pod的限制。设置普罗米修斯的地址，以便可以监控。

1. 第一步：打印初始参数

```
// 第一步：打印日志信息
klog.InitFlags(nil)
kube_flag.InitFlags()
klog.V(1).Infof("Vertical Pod Autoscaler %s Updater", common.VerticalPodAutoscalerVersion)
```

2. 第二步：metrics的监控

```
// 第二步：启动普罗米修斯的监控和healthCheck
// 该步骤类似于控制器ac的第二步
healthCheck := metrics.NewHealthCheck(*updaterInterval*5, true)
metrics.Initialize(*address, healthCheck)
metrics_updater.Register()
```

（和控制器ac部分相同）

main启动函数在此启动了普罗米修斯来监控，包括健康检查和metrics：

- healthcheck.go

```
vertical-pod-autoscaler/pkg/utils/metrics/healthcheck.go
```

healthCheck包含了最后一次监控组件的活动时间信息，通过NewHealthCheck函数在给定时间点创建新的HealthCheck结构类型。

```
// HealthCheck contains information about last activity time of the monitored component.
//// NOTE: This started as a simplified version of ClusterAutoscaler's HealthCheck.
type HealthCheck struct {
 activityTimeout time.Duration
 checkTimeout bool
 lastActivity time.Time
 mutex *sync.Mutex
}

// NewHealthCheck builds new HealthCheck object with given timeout.
func NewHealthCheck(activityTimeout time.Duration, checkTimeout bool) *HealthCheck {
 return &HealthCheck{
 activityTimeout: activityTimeout,
```

```
  checkTimeout: checkTimeout,
  lastActivity: time.Now(),
  mutex: &sync.Mutex{},
  }
}
```

暂时还不清楚healthcheck在整个流程中起到的作用，这个问题暂时先留着。

- metrics.go

```
vertical-pod-autoscaler/pkg/utils/metrics/metrics.go
```

main函数中调用了metrics.Initialize接口来初始化Prometheus的metrics和health-check的给定地址。

```
// Initialize sets up Prometheus to expose metrics & (optionally) health-check on the given address
func Initialize(address string, healthCheck *HealthCheck) {
 go func() {
 http.Handle("/metrics", promhttp.Handler()) //为Prometheus注册HTTP服务端，给/metrics路径提供服务
 if healthCheck != nil {
 http.Handle("/health-check", healthCheck) //在判断health-check是否正常后，给health-check提供http服务
 }
 err := http.ListenAndServe(address, nil)//监听Prometheus地址，接收8944端口发送过来的信息
 klog.Fatalf("Failed to start metrics: %v", err)
 }()
}
```

- updater.go

```
vertical-pod-autoscaler/pkg/utils/metrics/updater/updater.go
```

updater.go为Updater提供metrics注册。

```
// Register initializes all metrics for VPA Updater
func Register() {
 prometheus.MustRegister(evictedCount) // 注册驱逐数量
 prometheus.MustRegister(functionLatency) // 注册功能延迟
```

```
}
```

Register注册函数执行Prometheus的两个注册函数，分别注册evictedCount和functionLatency。

```
var (
 evictedCount = prometheus.NewCounter(
 prometheus.CounterOpts{
 Namespace: metricsNamespace,
 Name: "evicted_pods_total",
 Help: "Number of Pods evicted by Updater to apply a new recommendation.",
 },
 )

 functionLatency = metrics.CreateExecutionTimeMetric(metricsNamespace,
 "Time spent in various parts of VPA Updater main loop.")
)
```

evictedCount和functionLatency分别记录驱逐pods的数量和每次循环vpa Updater时间花费（延迟）。

第三步. 获取集群信息和创建k8s vpa客户端，创建informer工厂（同控制器ac部分）

```
// 第三步：获取集群信息
config, err := kube_restclient.InClusterConfig()
if err != nil {
 klog.Fatalf("Failed to build Kubernetes client : fail to create config: %v", err)
}
```

通过config的配置变量，来创建vpa的客户端和vpa的lister（用来列出所有vpa），以及创建k8s客户端。

```
//创建k8s vpa客户端，创建informer工厂
kubeClient := kube_client.NewForConfigOrDie(config)
vpaClient := vpa_clientset.NewForConfigOrDie(config)

//informer工厂，调用NewSharedInformerFactory()接口
```

```
//疑问一：调用informers作用
//答：创建一个名为 SharedInformerFactory 的单例工厂，因为每个Informer都会与Api Server维持一个watch长连接。
factory := informers.NewSharedInformerFactory(kubeClient, defaultResyncPeriod)
//sharedIndexInformer 是一个共享的 Informer 框架
//vpa只需要提供一个模板类（比如 deploymentInformer ），便可以创建一个符合自己需求的特定 Informer。
```

- clientset.go

```
vertical-pod-autoscaler/pkg/client/clientset/versioned/clientset.go
```

```
// NewForConfigOrDie creates a new Clientset for the given config and
// panics if there is an error in the config.
func NewForConfigOrDie(c *rest.Config) *Clientset {
 var cs Clientset
 cs.autoscalingV1 = autoscalingv1.NewForConfigOrDie(c)
 cs.autoscalingV1beta2 = autoscalingv1beta2.NewForConfigOrDie(c)
 cs.autoscalingV1beta1 = autoscalingv1beta1.NewForConfigOrDie(c)
 cs.pocV1alpha1 = pocv1alpha1.NewForConfigOrDie(c)

 cs.DiscoveryClient = discovery.NewDiscoveryClientForConfigOrDie(c)
 return &cs
}
```

main启动函数在第六步调用了informer创建了SharedInformerFactory 的单例工厂，因为每个 Informer 都会与 Api Server 维持一个 watch 长连接，所以这个单例工厂通过为所有 vpa 提供了唯一获取 Informer 的入口，来保证每种类型的 Informer 只被实例化一次。

第四步. target

（同控制器ac部分）

```
//第七步：target
```

```
//疑问二：target是个什么？
// 答： target所要做的事情就是通过discoveryClient来个API Server交互获得scale扩容的信息
// 返回的vpaTargetSelectorFetcher结构体：
// &vpaTargetSelectorFetcher{
// scaleNamespacer: scaleNamespacer,
// mapper: mapper,
// informersMap: informersMap,
// }
 // scaleNamespacer：扩容命名空间
 // mapper：discovery information
 // informersMap：七个资源类型的informer实例
 targetSelectorFetcher := target.NewVpaTargetSelectorFetcher(config, kubeClient, factory)
```

main启动函数在第七步调用fetcher.go里面的NewVpaTargetSelectorFetcher方法，该方法返回VpaTargetSelectorFetcher新实例，而定义的VpaTargetSelectorFetcher来抓取labelSelector(标签选择器)，用于收集由给定VPA控制的Pod。

- fetcher.go

```
vertical-pod-autoscaler/pkg/target/fetcher.go
```

VpaTargetSelectorFetcher：

```
// VpaTargetSelectorFetcher gets a labelSelector used to gather Pods controlled by the given VPA.
// VpaTargetSelectorFetcher获取一个labelSelector，用于收集由给定VPA控制的Pods。
type VpaTargetSelectorFetcher interface {
  // Fetch returns a labelSelector used to gather Pods controlled by the given VPA.
  // If error is nil, the returned labelSelector is not nil.
  Fetch(vpa *vpa_types.VerticalPodAutoscaler) (labels.Selector, error)
}
```

NewVpaTargetSelectorFetcher方法：

```
// NewVpaTargetSelectorFetcher returns new instance of VpaTargetSelectorFetcher
// NewVpaTargetSelectorFetcher返回VpaTargetSelectorFetcher的新实例
func NewVpaTargetSelectorFetcher(config *rest.Config, kubeClient kube_client.Interface, factory informers.SharedInformerFactory) VpaTargetSelectorFetcher {
  discoveryClient, err := discovery.NewDiscoveryClientForConfig(config)
  // NewDiscoveryClientForConfig为给定的配置创建一个新的DiscoveryClient。 该客户端可用于发现API Server中受支持的资源。

  if err != nil {
  klog.Fatalf("Could not create discoveryClient: %v", err)
  }

   resolver := scale.NewDiscoveryScaleKindResolver(discoveryClient)
  // 疑问一： scale.NewDiscoveryScaleKindResolver作用?返回的resolver是个什么?
  // 答：NewDiscoveryScaleKindResolver创建一个新的ScaleKindResolver，
  // 它使用来自给定Disovery客户端的信息来为不同资源解析正确的Scale GroupVersionKind。
  // 返回的是cachedScaleKindResolver结构体，里面的cache存储Scale GroupVersionKind类型的资源。

  restClient := kubeClient.CoreV1().RESTClient()
  // 通过k8s客户端获取rest客户端
  cachedDiscoveryClient := cacheddiscovery.NewMemCacheClient(discoveryClient)
  // 疑问二：缓存cache客户端?
  // 答：NewMemCacheClient创建一个新的CachedDiscoveryInterface，它将discoveryClient中的discovery information缓存在内存cache中，
  // 如果定期调用invalidate，它将保持最新状态。（官方）

  mapper := restmapper.NewDeferredDiscoveryRESTMapper(cachedDiscoveryClient)
  // NewDeferredDiscoveryRESTMapper返回DeferredDiscoveryRESTMapper
  // 它将延迟查询给以提供的客户端，以获取用于进行REST映射的discovery information。（官方）
  // 这里是把cachedDiscoveryClient中的缓存信息放入这个特定mapper中，特定是因为这个mapper中的信息是通过客户端延迟查询且通过rest映射得到的

  go wait.Until(func() {
  mapper.Reset()
  }, discoveryResetPeriod, make(chan struct{}))
  // 协程不断更新mapper的信息
```

```
33
34  informersMap := map[wellKnownController]cache.SharedIndexInformer{
35  daemonSet: factory.Apps().V1().DaemonSets().Informer(),
36  deployment: factory.Apps().V1().Deployments().Informer(),
37  replicaSet: factory.Apps().V1().ReplicaSets().Informer(),
38  statefulSet: factory.Apps().V1().StatefulSets().Informer(),
39  replicationController: factory.Core().V1().ReplicationControllers().Inf
ormer(),
40  job: factory.Batch().V1().Jobs().Informer(),
41  cronJob: factory.Batch().V1beta1().CronJobs().Informer(),
42  }
43  // 七个资源类型通过cache.SharedIndexInformer来实现Informer实例，在map stor
age中存储
44
45  for kind, informer := range informersMap {
46  stopCh := make(chan struct{})
47  go informer.Run(stopCh)
48  // 不断启动informersMap中实例化的informer
49  synced := cache.WaitForCacheSync(stopCh, informer.HasSynced)
50  // 等待所有已经启动的 Informer 的 Cache 同步完成，同步全量对象
51  // WaitForCacheSync等待缓存填充。 如果成功，则返回true；如果控制器应关闭，则
返回false
52  if !synced {
53  klog.Fatalf("Could not sync cache for %s: %v", kind, err)
54  } else {
55  klog.Infof("Initial sync of %s completed", kind)
56  }
57  }
58  // 上面的循环就是一个不断启动informer和不断更新同步缓存的一个过程
59
60  scaleNamespacer := scale.New(restClient, mapper, dynamic.LegacyAPIPathR
esolverFunc, resolver)
61  // scale.New使用给定的客户端来创建新的ScalesGetter进行请求。
62  // scaleNamespacer是一个scaleClient结构体：
63  // scaleClient{
64  // mapper: mapper,
65  // apiPathResolverFunc: resolver,
66  // scaleKindResolver: scaleKindResolver,
67  // clientBase: baseClient,
68  // }
69
```

```
70   return &vpaTargetSelectorFetcher{
71   scaleNamespacer: scaleNamespacer,
72   mapper: mapper,
73   informersMap: informersMap,
74   }
75
76  }
```

在最后返回的vpaTargetSelectorFetcher中的mapper和informersMap我怀疑是用于和API Server进行交互获取pod信息的变量。如何求证这一点有待追究。

第五步. limitrange

```
1  // 第五步：创建计算限制资源类型
2  var limitRangeCalculator limitrange.LimitRangeCalculator
3  limitRangeCalculator, err = limitrange.NewLimitsRangeCalculator(factory)
4  // 通过factory的sharedIndexInformer工厂来实例化limitrange这个资源对象，通过该资源对象获取API Server中的资源计算限制信息
5  if err != nil {
6   klog.Errorf("Failed to create limitRangeCalculator, falling back to not checking limits. Error message: %s", err)
7   limitRangeCalculator = limitrange.NewNoopLimitsCalculator()
8  }
```

- limit_range_calculator.go

```
1  vertical-pod-autoscaler/pkg/utils/limitrange/limit_range_calculator.go
```

limit_range_calculator.go定义了LimitRangeCalculator类型来限制计算范围，计算的限制范围是对于拥有相同效果的items和这些存在于集群中items而言的。

```
1  // LimitRangeCalculator calculates limit range items that has the same effect as all limit range items present in the cluster.
2  // LimitRangeCalculator计算的限制范围是对于拥有相同效果的items和这些存在于集群中items而言。
3  type LimitRangeCalculator interface {
```

```
4  // GetContainerLimitRangeItem returns LimitRangeItem that describes limitation on container limits in the given namespace.
5  // GetContainerLimitRangeItem返回LimitRangeItem，该限制描述的是给定名称空间中对container的限制。
6  GetContainerLimitRangeItem(namespace string) (*core.LimitRangeItem, error)
7  // GetPodLimitRangeItem returns LimitRangeItem that describes limitation on pod limits in the given namespace.
8  // GetPodLimitRangeItem返回LimitRangeItem，它描述给定名称空间中对pod的限制。
9  GetPodLimitRangeItem(namespace string) (*core.LimitRangeItem, error)
10 }
```

```
1 // NewLimitsRangeCalculator returns a limitsChecker or an error it encountered when attempting to create it.
2 // NewLimitsRangeCalculator返回limitsChecker或尝试创建它时遇到的错误。
3 func NewLimitsRangeCalculator(f informers.SharedInformerFactory) (*limitsChecker, error) {
4  if f == nil {
5  return nil, fmt.Errorf("NewLimitsRangeCalculator requires a SharedInformerFactory but got nil")
6  }
7  limitRangeLister := f.Core().V1().LimitRanges().Lister()
8  // 通过SharedInformerFactory创建了limitRange这个Informer实例，然后通过Core().V1().LimitRanges().Lister()获取注册后的的lister
9  // （LimitRangeInformer provides access to a shared informer and lister for LimitRanges.）
10  //type LimitRangeInformer interface {
11  // Informer() cache.SharedIndexInformer
12  // Lister() v1.LimitRangeLister
13  //}
14  stopCh := make(chan struct{})
15  f.Start(stopCh)
16  // 启动f中注册的所有Informer，该步骤必须在注册Informer之后。
17  // 这里解释一下上面的LimitRanges().Lister()，按照正常步骤来说应该是先LimitRanges()，然后start启动，再获取lister
18  // 这里直接进行了Lister()，说明这样的方法也是可以的
19  for _, ok := range f.WaitForCacheSync(stopCh) {
20  // 等待所有已经启动的 Informer 的 Cache 同步完成，同步全量对象
21  if !ok {
22  if !f.Core().V1().LimitRanges().Informer().HasSynced() {
23  // 如果informer的sync没有同步对象，则报错
24  return nil, fmt.Errorf("informer did not sync")
```

```
  }
  }
  }
  return &limitsChecker{limitRangeLister}, nil
}
```

注意：这里通过Informer工厂实例化了LimitRanges这个资源对象！！！！！

第六步. 通过SharedInformerFactory创建updater资源类型（关键步骤）

最主要的还是要创建Updater这个资源类型。

```
// TODO: use SharedInformerFactory in updater
// 第六步：通过SharedInformerFactory创建updater资源类型（关键步骤）
updater, err := updater.NewUpdater(kubeClient, vpaClient, *minReplicas, *evictionRateLimit, *evictionRateBurst, *evictionToleranceFraction, vpa_api_util.NewCappingRecommendationProcessor(limitRangeCalculator), nil, targetSelectorFetcher)
if err != nil {
 klog.Fatalf("Failed to create updater: %v", err)
}
```

- updater.go

```
vertical-pod-autoscaler/pkg/updater/logic/updater.go
```

```
// NewUpdater creates Updater with given configuration
// 通过给定的配置创建Updater
func NewUpdater(kubeClient kube_client.Interface, vpaClient *vpa_clientset.Clientset, minReplicasForEvicition int, evictionRateLimit float64, evictionRateBurst int, evictionToleranceFraction float64, recommendationProcessor vpa_api_util.RecommendationProcessor, evictionAdmission priority.PodEvictionAdmission, selectorFetcher target.VpaTargetSelectorFetcher) (Updater, error) {
  evictionRateLimiter := getRateLimiter(evictionRateLimit, evictionRateBurst)
  // 一. 获取驱逐rate
  factory, err := eviction.NewPodsEvictionRestrictionFactory(kubeClient, minReplicasForEvicition, evictionToleranceFraction)
  // 二. 建立驱逐pod的限制工厂
```

```
8   if err != nil {
9   return nil, fmt.Errorf("Failed to create eviction restriction factory: %
v", err)
10  }
11  return &updater{
12  vpaLister: vpa_api_util.NewAllVpasLister(vpaClient, make(chan
struct{})),
13  // 通过vpa api获取vpa lister信息
14  podLister: newPodLister(kubeClient),
15  // 通过k8s api获取pod lister信息
16  eventRecorder: newEventRecorder(kubeClient),
17  evictionFactory: factory,
18  // 驱逐工厂资源实例
19  recommendationProcessor: recommendationProcessor,
20  // 通过limitrange资源对象计算得到的推荐驱逐值
21  evictionRateLimiter: evictionRateLimiter,
22  // 驱逐rate
23  evictionAdmission: evictionAdmission,
24  // 默认为nil
25  selectorFetcher: selectorFetcher,
26  // 抓取到的选择器
27  }, nil
28 }
```

两个比较关键的地方，一个是getRateLimiter，另一个是NewPodsEvictionRestrictionFactory。

- getRateLimiter

```
1  func getRateLimiter(evictionRateLimit float64, evictionRateLimitBurst
int) *rate.Limiter {
2   var evictionRateLimiter *rate.Limiter
3   if evictionRateLimit <= 0 {
4   // As a special case if the rate is set to rate.Inf, the burst rate is i
gnored
5   // see https://github.com/golang/time/blob/master/rate/rate.go#L37
6   evictionRateLimiter = rate.NewLimiter(rate.Inf, 0)
7   klog.V(1).Info("Rate limit disabled")
8   } else {
9   evictionRateLimiter = rate.NewLimiter(rate.Limit(evictionRateLimit), evi
ctionRateLimitBurst)
10  }
11  return evictionRateLimiter
```

```
}
```

这一部分的算法参考

https://github.com/golang/time/blob/master/rate/rate.go#L37。

- NewPodsEvictionRestrictionFactory

```
// NewPodsEvictionRestrictionFactory creates PodsEvictionRestrictionFactory
// 创建PodsEvictionRestrictionFactory
func NewPodsEvictionRestrictionFactory(client kube_client.Interface, minReplicas int,
 evictionToleranceFraction float64) (PodsEvictionRestrictionFactory, error) {
 rcInformer, err := setUpInformer(client, replicationController)
 // 创建RC的informer
 if err != nil {
 return nil, fmt.Errorf("Failed to create rcInformer: %v", err)
 }
 ssInformer, err := setUpInformer(client, statefulSet)
 // 创建statefulSet的informer
 if err != nil {
 return nil, fmt.Errorf("Failed to create ssInformer: %v", err)
 }
 rsInformer, err := setUpInformer(client, replicaSet)
 // 创建replicaSet的Informer
 if err != nil {
 return nil, fmt.Errorf("Failed to create rsInformer: %v", err)
 }
 return &podsEvictionRestrictionFactoryImpl{
 client: client,
 rcInformer: rcInformer, // informer for Replication Controllers
 ssInformer: ssInformer, // informer for Replica Sets
 rsInformer: rsInformer, // informer for Stateful Sets
 minReplicas: minReplicas,
 evictionToleranceFraction: evictionToleranceFraction}, nil
}
```

第七步. 迭代时间，进行更新

```
// 第七步：迭代时间 进行更新
ticker := time.Tick(*updaterInterval)
for range ticker {
 ctx, cancel := context.WithTimeout(context.Background(), *updaterInterval)
 defer cancel()
 updater.RunOnce(ctx)
 // 整个循环中单个迭代
 healthCheck.UpdateLastActivity()
 // 更新healthCheck
 }
```

这一步主要是一个大循环，里面不断调用updater的RunOnce函数进行小迭代。主要还是RunOnce这个函数块实现更新。

- RunOnce

```
vertical-pod-autoscaler/pkg/updater/logic/updater.go
```

```
// RunOnce represents single iteration in the main-loop of Updater
// RunOnce 是Updater主循环中的一个迭代
func (u *updater) RunOnce(ctx context.Context) {
 timer := metrics_updater.NewExecutionTimer()
 // 执行时间

 vpaList, err := u.vpaLister.List(labels.Everything())
 // 获取vpa
 if err != nil {
  klog.Fatalf("failed get VPA list: %v", err)
  }
  timer.ObserveStep("ListVPAs")

  vpas := make([]*vpa_api_util.VpaWithSelector, 0)

  for _, vpa := range vpaList {
  if vpa_api_util.GetUpdateMode(vpa) != vpa_types.UpdateModeRecreate &&
  vpa_api_util.GetUpdateMode(vpa) != vpa_types.UpdateModeAuto {
```

```
19  // 如果vpa的更新不可重创或者不为自动模式 则循环下一个vpa
20  klog.V(3).Infof("skipping VPA object %v because its mode is not \"Recre
ate\" or \"Auto\"", vpa.Name)
21  continue
22  }
23  selector, err := u.selectorFetcher.Fetch(vpa)
24  // 抓取该vpa的selector
25  if err != nil {
26  klog.V(3).Infof("skipping VPA object %v because we cannot fetch selecto
r", vpa.Name)
27  continue
28  }
29
30  vpas = append(vpas, &vpa_api_util.VpaWithSelector{
31  Vpa: vpa,
32  Selector: selector,
33  })
34  }
35
36  if len(vpas) == 0 {
37  klog.Warningf("no VPA objects to process")
38  if u.evictionAdmission != nil {
39  u.evictionAdmission.CleanUp()
40  }
41  timer.ObserveTotal()
42  return
43  }
44
45  podsList, err := u.podLister.List(labels.Everything())
46  // 获取pod
47  if err != nil {
48  klog.Errorf("failed to get pods list: %v", err)
49  timer.ObserveTotal()
50  return
51  }
52  timer.ObserveStep("ListPods")
53  allLivePods := filterDeletedPods(podsList)
54  // 过滤无效pod
55
56  controlledPods := make(map[*vpa_types.VerticalPodAutoscaler][]*apiv1.Po
d)
```

```
57  for _, pod := range allLivePods {
58  controllingVPA := vpa_api_util.GetControllingVPAForPod(pod, vpas)
59  // 获取与pod匹配的vpa
60  if controllingVPA != nil {
61  controlledPods[controllingVPA.Vpa] = append(controlledPods[controllingVPA.Vpa], pod)
62  }
63  }
64  timer.ObserveStep("FilterPods")
65
66  if u.evictionAdmission != nil {
67  u.evictionAdmission.LoopInit(allLivePods, controlledPods)
68  // 控制pod驱逐的控制器进行初始化
69  }
70  timer.ObserveStep("AdmissionInit")
71
72  for vpa, livePods := range controlledPods {
73  evictionLimiter := u.evictionFactory.NewPodsEvictionRestriction(livePods)
74  // 一. 驱逐策略
75  podsForUpdate := u.getPodsUpdateOrder(filterNonEvictablePods(livePods, evictionLimiter), vpa)
76  // 二. 获得pods的更新队列
77  for _, pod := range podsForUpdate {
78  if !evictionLimiter.CanEvict(pod) {
79  // 判断是否可以驱逐
80  continue
81  }
82  err := u.evictionRateLimiter.Wait(ctx)
83  // 等待驱逐
84  if err != nil {
85  klog.Warningf("evicting pod %v failed: %v", pod.Name, err)
86  return
87  }
88  klog.V(2).Infof("evicting pod %v", pod.Name)
89  evictErr := evictionLimiter.Evict(pod, u.eventRecorder)
90  // 判断驱逐成功
91  if evictErr != nil {
92  klog.Warningf("evicting pod %v failed: %v", pod.Name, evictErr)
93  }
94  }
```

```
95   }
96   timer.ObserveStep("EvictPods")
97   timer.ObserveTotal()
98  }
```

两个很关键的地方，分别在源码中进行了标注。

```
1  evictionLimiter := u.evictionFactory.NewPodsEvictionRestriction(livePods)
2  // 关键点一：驱逐策略
3  podsForUpdate := u.getPodsUpdateOrder(filterNonEvictablePods(livePods, ev
ictionLimiter), vpa)
4  // 关键点二：获得pods的更新队列
```

- NewPodsEvictionRestriction

```
1  vertical-pod-autoscaler/pkg/updater/eviction/pods_eviction_restriction.go
```

```
1  // NewPodsEvictionRestriction creates PodsEvictionRestriction for a given
set of pods.
2  func (f *podsEvictionRestrictionFactoryImpl) NewPodsEvictionRestriction(p
ods []*apiv1.Pod) PodsEvictionRestriction {
3   // We can evict pod only if it is a part of replica set
4   // For each replica set we can evict only a fraction of pods.
5   // Evictions may be later limited by pod disruption budget if configure
d.
6
7   // 我们只能将pod作为副本集的一部分进行驱逐
8   // 对于每个副本集，我们只能逐出一部分pods。
9   // 如果配置，逐出可能会受到pod中断预算的限制。
10   livePods := make(map[podReplicaCreator][]*apiv1.Pod)
11
12   for _, pod := range pods {
13   creator, err := getPodReplicaCreator(pod)
14   // 获取pod的replication信息
15   if err != nil {
16   klog.Errorf("failed to obtain replication info for pod %s: %v", pod.Nam
e, err)
17   continue
18   }
19   if creator == nil {
20   klog.Warningf("pod %s not replicated", pod.Name)
```

```
21  continue
22  }
23  livePods[*creator] = append(livePods[*creator], pod)
24  }
25
26  podToReplicaCreatorMap := make(map[string]podReplicaCreator)
27  creatorToSingleGroupStatsMap := make(map[podReplicaCreator]singleGroupS
tats)
28
29  for creator, replicas := range livePods {
30  actual := len(replicas)
31  if actual < f.minReplicas {
32  // 副本数过少
33  klog.V(2).Infof("too few replicas for %v %v/%v. Found %v live pods",
34  creator.Kind, creator.Namespace, creator.Name, actual)
35  continue
36  }
37
38  var configured int
39  if creator.Kind == job {
40  // 判断pod的类型是否为job类型
41  // Job has no replicas configuration, so we will use actual number of l
ive pods as replicas count.
42  configured = actual
43  } else {
44  var err error
45  configured, err = f.getReplicaCount(creator)
46  // 获取creator数量
47  if err != nil {
48  klog.Errorf("failed to obtain replication info for %v %v/%v. %v",
49  creator.Kind, creator.Namespace, creator.Name, err)
50  continue
51  }
52  }
53
54  singleGroup := singleGroupStats{}
55  singleGroup.configured = configured
56  singleGroup.evictionTolerance = int(float64(configured) * f.evictionTol
eranceFraction)
57
58  for _, pod := range replicas {
```

```
59  podToReplicaCreatorMap[getPodID(pod)] = creator
60  if pod.Status.Phase == apiv1.PodPending {
61  singleGroup.pending = singleGroup.pending + 1
62  }
63  }
64
65  singleGroup.running = len(replicas) - singleGroup.pending
66  creatorToSingleGroupStatsMap[creator] = singleGroup
67  // 这几步没看懂。。
68  }
69  return &podsEvictionRestrictionImpl{
70  client: f.client,
71  podToReplicaCreatorMap: podToReplicaCreatorMap,
72  creatorToSingleGroupStatsMap: creatorToSingleGroupStatsMap}
73 }
```

- getPodsUpdateOrder

```
1 vertical-pod-autoscaler/pkg/updater/logic/updater.go
```

```
1 func (u *updater) getPodsUpdateOrder(pods []*apiv1.Pod, vpa *vpa_types.Ve
rticalPodAutoscaler) []*apiv1.Pod {
2  priorityCalculator := priority.NewUpdatePriorityCalculator(vpa.Spec.Reso
urcePolicy, vpa.Status.Conditions, nil, u.recommendationProcessor)
3  // 计算更新优先级
4  recommendation := vpa.Status.Recommendation
5  // 获取更新建议
6  for _, pod := range pods {
7  priorityCalculator.AddPod(pod, recommendation, time.Now())
8  // 一个个根据建议进行更新
9  }
10
11  return priorityCalculator.GetSortedPods(u.evictionAdmission)
12  // 返回排序后的更新队列
13 }
```

priorityCalculator.AddPod根据建议对pod进行添加。

priorityCalculator.GetSortedPods会对添加进入的pods进行排序，如

果admission能够使得pod接纳该建议，则将结果添加，返回排序后的队列。

- AddPod

```
vertical-pod-autoscaler/pkg/updater/priority/update_priority_calculator.go
```

```
// AddPod adds pod to the UpdatePriorityCalculator.
// 增加pod到UpdatePriorityCalculator中
func (calc *UpdatePriorityCalculator) AddPod(pod *apiv1.Pod, recommendation *vpa_types.RecommendedPodResources, now time.Time) {
 processedRecommendation, _, err := calc.recommendationProcessor.Apply(recommendation, calc.resourcesPolicy, calc.conditions, pod)
 // 处理建议
 if err != nil {
 klog.V(2).Infof("cannot process recommendation for pod %s: %v", pod.Name, err)
 return
 }

 updatePriority := calc.getUpdatePriority(pod, processedRecommendation)
 // 根据建议 pod获取更新优先级

 quickOOM := false
 if len(pod.Status.ContainerStatuses) == 1 {
 terminationState := pod.Status.ContainerStatuses[0].LastTerminationState
 if terminationState.Terminated != nil &&
 terminationState.Terminated.Reason == "OOMKilled" &&
 terminationState.Terminated.FinishedAt.Time.Sub(terminationState.Terminated.StartedAt.Time) < *evictAfterOOMThreshold {
 quickOOM = true
 klog.V(2).Infof("quick OOM detected in pod %v", pod.Name)
 }
 }
 // OOM ? ? ? ? ? ?

 // The update is allowed in following cases:
 // - the request is outside the recommended range for some container.
 // - the pod lives for at least 24h and the resource diff is >= MinChangePriority.
```

```
29  // - there is only one container in a pod and it OOMed in less than evi
ctAfterOOMThreshold
30
31  // 在以下情况下允许更新：
32  //-请求超出了某些容器的建议范围。
33  //-pod至少生存24小时，并且资源差异> = MinChangePriority。
34  //-pod中只有一个容器，并且它在小于evictAfterOOMThreshold的时间内进行了OOM操
作
35  if !updatePriority.outsideRecommendedRange && !quickOOM {
36  if pod.Status.StartTime == nil {
37  // TODO: Set proper condition on the VPA.
38  klog.V(2).Infof("not updating pod %v, missing field pod.Status.StartTim
e", pod.Name)
39  return
40  }
41  if now.Before(pod.Status.StartTime.Add(podLifetimeUpdateThreshold)) {
42  klog.V(2).Infof("not updating a short-lived pod %v, request within reco
mmended range", pod.Name)
43  return
44  }
45  if updatePriority.resourceDiff < calc.config.MinChangePriority {
46  klog.V(2).Infof("not updating pod %v, resource diff too low: %v", pod.N
ame, updatePriority)
47  return
48  }
49  }
50  klog.V(2).Infof("pod accepted for update %v with priority %v",
pod.Name, updatePriority.resourceDiff)
51  calc.pods = append(calc.pods, updatePriority)
52  // 进行添加操作
53 }
```

- GetSortedPods

```
1 vertical-pod-autoscaler/pkg/updater/priority/update_priority_calculator.g
o
```

```
1 // GetSortedPods returns a list of pods ordered by update priority (highe
st update priority first)
2 func (calc *UpdatePriorityCalculator) GetSortedPods(admission PodEviction
Admission) []*apiv1.Pod {
3  sort.Sort(byPriority(calc.pods))
4  // 进行排序
```

```
  result := []*apiv1.Pod{}
  for _, podPrio := range calc.pods {
  if admission == nil || admission.Admit(podPrio.pod, podPrio.recommendation) {
  // 如果admission能够使得pod接纳该建议，则将结果添加
  result = append(result, podPrio.pod)
   } else {
   klog.V(2).Infof("pod removed from update queue by PodEvictionAdmission: %v", podPrio.pod.Name)
   }
   }

   return result
}
```