- **main.go**

首先先分析AC（Adminssion Controller）的启动函数main.go函数

```
autoscaler\vertical-pod-autoscaler\pkg\admission-controller\main.go
```

启动函数首先定义了初始变量，让我们来看一下：

```
var (
 certsConfiguration = &certsConfig{
 clientCaFile: flag.String("client-ca-file", "/etc/tls-certs/caCert.pem", "Path to CA PEM file."),
 tlsCertFile: flag.String("tls-cert-file", "/etc/tls-certs/serverCert.pem", "Path to server certificate PEM file."),
 tlsPrivateKey: flag.String("tls-private-key", "/etc/tls-certs/serverKey.pem", "Path to server certificate key PEM file."),
 }

 port = flag.Int("port", 8000, "The port to listen on.")
 address = flag.String("address", ":8944", "The address to expose Prometheus metrics.")
  namespace = os.Getenv("NAMESPACE")
 webhookAddress = flag.String("webhook-address", "", "Address under which webhook is registered. Used when registerByURL is set to true.")
 webhookPort = flag.String("webhook-port", "", "Server Port for Webhook")
 registerByURL = flag.Bool("register-by-url", false, "If set to true, admission webhook will be registered by URL (webhookAddress:webhookPort) instead of by service name")
)
```

certsConfiguration 部分是证书的config配置区域，将指定的CA证书或者server证书地址进行config。port：8000是adminssion-controller对外暴露的端口，address是普罗米修斯对外暴露的metrics端口，用于监控数据变化。webhookAddress:webhookPort提供了注册url，在这里替代server name来注册。

1. 第一步：打印初始化参数

```
//打印初始化参数
klog.InitFlags(nil)
kube_flag.InitFlags()
```

```
klog.V(1).Infof("Vertical Pod Autoscaler %s Admission Controller", common.VerticalPodAutoscalerVersion)
```

打印正在使用的Adminssion Controller配置信息。

2. 第二步：metrics的监控

```
//metrics的监控
healthCheck := metrics.NewHealthCheck(time.Minute, false)
metrics.Initialize(*address, healthCheck)
metrics_admission.Register()
```

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)
 }()
}
```

- admission.go

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

该admission包给VPA Admission Controller plugin提供metrics代码。

```
// Register initializes all metrics for VPA Admission Contoller
func Register() {
 prometheus.MustRegister(admissionCount)
 prometheus.MustRegister(admissionLatency)
}
```

Register注册函数执行Prometheus的两个注册函数，分别注册admissionCount和admissionLatency。

```
var (
 admissionCount = prometheus.NewCounterVec(
 prometheus.CounterOpts{
 Namespace: metricsNamespace,
 Name: "admission_pods_total",
 Help: "Number of Pods processed by VPA Admission Controller.",
 }, []string{"applied"},
 )

 admissionLatency = prometheus.NewHistogramVec(
 prometheus.HistogramOpts{
 Namespace: metricsNamespace,
 Name: "admission_latency_seconds",
 Help: "Time spent in VPA Admission Controller.",
 Buckets: []float64{0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 30.0, 60.0, 120.0, 300.0},
 }, []string{"status", "resource"},
 )
)
```

admissionCount和admissionLatency分别记录pods的数量和时间花费（延迟）。在这里介绍两个变量：

```
// AdmissionStatus describes the result of Admission Control execution
type AdmissionStatus string //Admission Control执行状态

// AdmissionResource describes the resource processed by Admission Control execution
type AdmissionResource string //Admission Control执行的资源类型

const (
 // Error denotes a failed Admission Control execution
 Error AdmissionStatus = "error"
 // Skipped denotes an Admission Control execution w/o applying a recommendation
 Skipped AdmissionStatus = "skipped"
 // Applied denotes an Admission Control execution when a recommendation was applied
```

```
 Applied AdmissionStatus = "applied"
)

const (
 // Unknown means that the resource could not be determined
 Unknown AdmissionResource = "unknown"
 // Pod means Kubernetes Pod
 Pod AdmissionResource = "Pod"
 // Vpa means VerticalPodAutoscaler object (CRD)
 Vpa AdmissionResource = "VPA"
)
```

3. 第三步和第四步：获取cert证书和集群参数

main启动函数接下来会获取cert证书参数，然后通过rest接口来获取集群的参数，放在config的配置变量中。

```
//初始化获取cert证书参数
certs := initCerts(*certsConfiguration)

//通过rest接口获取cluster参数
config, err := rest.InClusterConfig()
if err != nil {
 klog.Fatal(err)
}
```

4. 第五步： 创建vpa客户端和lister（列出所有vpa），创建k8s客户端

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vpa客户端和lister（列出所有vpa），创建k8s客户端
vpaClient := vpa_clientset.NewForConfigOrDie(config)
vpaLister := vpa_api_util.NewAllVpasLister(vpaClient, make(chan struct{}))
kubeClient := kube_client.NewForConfigOrDie(config)
```

- 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
}
```

- api.go

```
vertical-pod-autoscaler/pkg/utils/vpa/api.go
```

NewAllVpasLister通过VerticalPodAutoscalerLister configured来抓取所有的vpa对象。这些vpa对象会在vpaLister被初始化创建的时候清空。

```
// NewAllVpasLister returns VerticalPodAutoscalerLister configured to fetch all VPA objects.
// The method blocks until vpaLister is initially populated.
func NewAllVpasLister(vpaClient *vpa_clientset.Clientset, stopChannel <-chan struct{}) vpa_lister.VerticalPodAutoscalerLister {
  vpaListWatch := cache.NewListWatchFromClient(vpaClient.AutoscalingV1().RESTClient(), "verticalpodautoscalers", core.NamespaceAll, fields.Everything())
  //调用了cache的NewListWatchFromClient函数，cache的NewListWatchFromClient函数创建了一个新的ListWatch，用于客户端和资源之间的数据交互。
  //1.NewListWatchFromClient方法将返回一个ListWatch结构体，定义listFunc和watchFunc；
  //2.listFunc用于List，watchFunc用于Watch；
  //3.ListWatch会传入newSourceApiserverFromLW方法；

  indexer, controller := cache.NewIndexerInformer(vpaListWatch,
  &vpa_types.VerticalPodAutoscaler{},
  1*time.Hour,
  &cache.ResourceEventHandlerFuncs{},
```

```
    cache.Indexers{cache.NamespaceIndex: cache.MetaNamespaceIndexFunc})
    //调用cache.NewIndexerInformer函数，通过sharedIndexInformer结构体创建了属于vpa自己的informer框架。
    //这里的informer起到和API Servers建立watch和list长连接的作用。
    //返回的两个变量indexer和controller的作用是提供了资源缓存的功能和控制管理vpa
    vpaLister := vpa_lister.NewVerticalPodAutoscalerLister(indexer)
    //通过indexer的资源缓存功能来获得所有的vpa信息。
    go controller.Run(stopChannel) //协程启动controller来管理vpa
    if !cache.WaitForCacheSync(make(chan struct{}), controller.HasSynced) {
    klog.Fatalf("Failed to sync VPA cache during initialization")
    } else {
    klog.Info("Initial VPA synced successfully")
    }
    return vpaLister
}
```

NewAllVpasLister方法中有几个比较关键的地方：

第一个地方就是注册ListWatch结构体的cache.NewListWatchFromClient方法，用于创建informer跟API Server建立长连接（而在这里实际上只用到了短链接获取indexer来获取list）。

第二个地方就是cache.NewIndexerInformer方法进行创建sharedIndexInformer这个结构体。在这里的交互机制有些复杂，具体可以看我整理的k8s原理和各组件间通信机制中的informer详解（k8s原理和各组件间通信机制.md）

6. 第六步：informer

```
//第六步：informer
//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。
```

```
5 factory := informers.NewSharedInformerFactory(kubeClient, defaultResyncPe
riod)
6 //sharedIndexInformer 是一个共享的 Informer 框架
7 //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。
```

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。

7. 第七步：target

```
1 //第七步：target
2 //疑问二：target是个什么？
3 // 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
4 // 返回的vpaTargetSelectorFetcher结构体：
5 // &vpaTargetSelectorFetcher{
6 // scaleNamespacer: scaleNamespacer,
7 // mapper: mapper,
8 // informersMap: informersMap,
9 // }
10 // scaleNamespacer：扩容命名空间
11 // mapper：discovery information
12 // informersMap：七个资源类型的informer实例
13 targetSelectorFetcher := target.NewVpaTargetSelectorFetcher(config, kube
Client, 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()
```

```
18  // 通过k8s客户端获取rest客户端
19  cachedDiscoveryClient := cacheddiscovery.NewMemCacheClient(discoveryCli
ent)
20  // 疑问二：缓存cache客户端？
21  // 答：NewMemCacheClient创建一个新的CachedDiscoveryInterface，它将discove
ryClient中的discovery information缓存在内存cache中，
22  // 如果定期调用invalidate，它将保持最新状态。（官方）
23
24  mapper := restmapper.NewDeferredDiscoveryRESTMapper(cachedDiscoveryClie
nt)
25  // NewDeferredDiscoveryRESTMapper返回DeferredDiscoveryRESTMapper
26  // 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 informati
on。（官方）
27  // 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
28
29  go wait.Until(func() {
30  mapper.Reset()
31  }, discoveryResetPeriod, make(chan struct{}))
32  // 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。

8. 第八步： Preprocessor

对main启动函数中的两个Preprocessor进行分析，来看看Preprocessor起到什么样的作用。

```
1 //第八步：Preprocessor
2 //疑问三：两个Preprocessor什么意思？
3 podPreprocessor := logic.NewDefaultPodPreProcessor()
```

```
vpaPreprocessor := logic.NewDefaultVpaPreProcessor()
```

- pod_pre_processor.go

```
vertical-pod-autoscaler/pkg/admission-controller/logic/pod_pre_processor.go
```

- vpa_pre_processor.go

```
vertical-pod-autoscaler/pkg/admission-controller/logic/vpa_pre_processor.go
```

两个预处理函数对pod和vpa在构建patches和运用default前进行处理。

9. 第九步： limitrange

```
//第九步：限制计算
var limitRangeCalculator limitrange.LimitRangeCalculator
//疑问四：通过factory来限制计算，原理？
// 答： 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
limitRangeCalculator, err = limitrange.NewLimitsRangeCalculator(factory)
if err != nil {
 klog.Errorf("Failed to create limitRangeCalculator, falling back to not checking limits. Error message: %s", err)
 limitRangeCalculator = limitrange.NewNoopLimitsCalculator()
}
```

- limit_range_calculator.go

```
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。

```
// LimitRangeCalculator calculates limit range items that has the same effect as all limit range items present in the cluster.
```

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

```
// NewLimitsRangeCalculator returns a limitsChecker or an error it encountered when attempting to create it.
// NewLimitsRangeCalculator返回limitsChecker或尝试创建它时遇到的错误。
func NewLimitsRangeCalculator(f informers.SharedInformerFactory) (*limitsChecker, error) {
 if f == nil {
 return nil, fmt.Errorf("NewLimitsRangeCalculator requires a SharedInformerFactory but got nil")
 }
 limitRangeLister := f.Core().V1().LimitRanges().Lister()
 // 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
 // （LimitRangeInformer provides access to a shared informer and lister for LimitRanges.）
 //type LimitRangeInformer interface {
 // Informer() cache.SharedIndexInformer
 // Lister() v1.LimitRangeLister
 //}
 stopCh := make(chan struct{})
 f.Start(stopCh)
 // 启动f中注册的所有Informer，该步骤必须在注册Informer之后。
 // 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
 // 这里直接进行了Lister()，说明这样的方法也是可以的
 for _, ok := range f.WaitForCacheSync(stopCh) {
 // 等待所有已经启动的 Informer 的 Cache 同步完成，同步全量对象
 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")
25   }
26   }
27   }
28   return &limitsChecker{limitRangeLister}, nil
29  }
```

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

10. 第十步：连接Recommendation（关键步骤）

```
1  //第十步：连接Recommendation
2  // 控制器AC会拦截Pod的创建请求，如果Pod与未设置为off模式的VPA配置匹配，控制器通过将推荐资源应用到Pod spec来重写请求。
3  // AC通过从Recommender获取推荐资源，如果调用超时或失败，控制器将采用缓存在VPA对象中的资源建议。如果这也是不可用的，控制器采取最初指定的资源。
4  recommendationProvider := logic.NewRecommendationProvider(limitRangeCalculator,
vpa_api_util.NewCappingRecommendationProcessor(limitRangeCalculator), targetSelectorFetcher, vpaLister)
```

- recommendation_provider.go

```
1  vertical-pod-autoscaler/pkg/admission-controller/logic/recommendation_provider.go
```

在recommendation_provider.go中涉及到获取给定pod下目前所有的recommendation，annotations和vpaName。

RecommendationProvider接口调用GetContainersResourcesForPod方法。

```
1  // RecommendationProvider获取给定pod的当前recommendation, annotations and vpaName。
2  type RecommendationProvider interface {
3   GetContainersResourcesForPod(pod *core.Pod) ([]vpa_api_util.ContainerResources, vpa_api_util.ContainerToAnnotationsMap, string, error)
4  }
5
```

```
// NewRecommendationProvider constructs the recommendation provider that list VPAs and can be used to determine recommendations for pods.
// NewRecommendationProvider构造列出VPA的recommendation提供者，可用于确定Pod的recommendation。
func NewRecommendationProvider(calculator limitrange.LimitRangeCalculator, recommendationProcessor vpa_api_util.RecommendationProcessor,
 selectorFetcher target.VpaTargetSelectorFetcher, vpaLister vpa_lister.VerticalPodAutoscalerLister) *recommendationProvider {
  return &recommendationProvider{
  limitsRangeCalculator: calculator,
  recommendationProcessor: recommendationProcessor,
  selectorFetcher: selectorFetcher,
  vpaLister: vpaLister,
  }
}
```

首先我们来看一下recommendationProvider这个结构体：

```
type recommendationProvider struct {
  limitsRangeCalculator limitrange.LimitRangeCalculator
  recommendationProcessor vpa_api_util.RecommendationProcessor
  selectorFetcher target.VpaTargetSelectorFetcher
  vpaLister vpa_lister.VerticalPodAutoscalerLister
}
```

可以看到该结构体中有main启动函数前九步中的变量。跟进GetContainersResourcesForPod方法：

```
// GetContainersResourcesForPod returns recommended request for a given pod, annotations and name of controlling VPA.
// The returned slice corresponds 1-1 to containers in the Pod.
// 更新对于指定pod的recommended推荐需求
func (p *recommendationProvider) GetContainersResourcesForPod(pod *core.Pod) ([]vpa_api_util.ContainerResources, vpa_api_util.ContainerToAnnotationsMap, string, error) {
  klog.V(2).Infof("updating requirements for pod %s.", pod.Name)
  vpaConfig := p.getMatchingVPA(pod)
  // 一. 获取指定的vpa(创建时间最早的)，该vpa更新状态未设置为off并且和截取到的pod创建信息匹配
  if vpaConfig == nil {
  klog.V(2).Infof("no matching VPA found for pod %s", pod.Name)
  return nil, nil, "", nil
  // 若不匹配，则返回无。
```

```
12  }
13
14  var annotations vpa_api_util.ContainerToAnnotationsMap
15  recommendedPodResources := &vpa_types.RecommendedPodResources{}
16
17  if vpaConfig.Status.Recommendation != nil {
18  var err error
19  recommendedPodResources, annotations, err = p.recommendationProcessor.Apply(vpaConfig.Status.Recommendation, vpaConfig.Spec.ResourcePolicy, vpaConfig.Status.Conditions, pod)
20  //二. 处理recommendation
21  if err != nil {
22  klog.V(2).Infof("cannot process recommendation for pod %s", pod.Name)
23  return nil, annotations, vpaConfig.Name, err
24  }
25  }
26  containerLimitRange, err := p.limitsRangeCalculator.GetContainerLimitRangeItem(pod.Namespace)
27  // 三. 获取容器运行时的限制范围
28  if err != nil {
29  return nil, nil, "", fmt.Errorf("error getting containerLimitRange: %s", err)
30  }
31  containerResources := GetContainersResources(pod, *recommendedPodResources, containerLimitRange, annotations)
32  // 四. 获取容器资源，返回的resources保存了每个容器对内存和cpu的限制信息
33  return containerResources, annotations, vpaConfig.Name, nil
34 }
```

这里面有四个比较关键的地方：

第一个地方是getMatchingVPA方法；

第二个是处理recommendation请求的地方；

第三个地方是获取容器运行时的限制范围

第四个地方是GetContainersResources方法获取容器资源的地方。

第一部分：

```go
func (p *recommendationProvider) getMatchingVPA(pod *core.Pod) *vpa_types.VerticalPodAutoscaler {
  configs, err := p.vpaLister.VerticalPodAutoscalers(pod.Namespace).List(labels.Everything())
  // 获取所有vpa的list，写入configs配置变量中
  if err != nil {
  klog.Errorf("failed to get vpa configs: %v", err)
  return nil
  }
  onConfigs := make([]*vpa_api_util.VpaWithSelector, 0)
  // 循环获得configs中的vpa，对这些vpa进行筛选
   for _, vpaConfig := range configs {
   if vpa_api_util.GetUpdateMode(vpaConfig) == vpa_types.UpdateModeOff {
   // 如果该vpa的更新模式关闭了，则选择下一个vpa
   continue
   }
   selector, err := p.selectorFetcher.Fetch(vpaConfig)
   // 抓取该vpa的选择器，放入selector中
   if err != nil {
   klog.V(3).Infof("skipping VPA object %v because we cannot fetch selector: %s", vpaConfig.Name, err)
   continue
   }
   onConfigs = append(onConfigs, &vpa_api_util.VpaWithSelector{
   Vpa: vpaConfig,
   Selector: selector,
   })
   }
   klog.V(2).Infof("Let's choose from %d configs for pod %s/%s", len(onConfigs), pod.Namespace, pod.Name)
   result := vpa_api_util.GetControllingVPAForPod(pod, onConfigs)
   // 从config配置中选择和pod匹配的创建时间最早vpa
   if result != nil {
   return result.Vpa
   }
   return nil
}
```

getMatchingVPA方法通过List方法获取所有在该pod的命名空间下的vpa，放置在configs变量中，然后不断对vpa进行筛选，是否和正

在创建的pod匹配。如果该vpa的更新模式关闭了，则选择下一个vpa，否则抓取该vpa的选择器，放入到onConfigs的变量中。最终的结果通过api.go中的GetControllingVPAForPod方法进行处理。

- api.go

```
vertical-pod-autoscaler/pkg/utils/vpa/api.go
```

GetControllingVPAForPod方法进行就是按照vpa的创建时间对创建比较早的vpa择优处理。

```
// PodMatchesVPA returns true iff the vpaWithSelector matches the Pod.
func PodMatchesVPA(pod *core.Pod, vpaWithSelector *VpaWithSelector) bool {
 return PodLabelsMatchVPA(pod.Namespace, labels.Set(pod.GetLabels()), vpaWithSelector.Vpa.Namespace, vpaWithSelector.Selector)
}

// stronger returns true if a is before b in the order to control a Pod (that matches both VPAs).
// 比较哪个vpa创建的时间比较早，早的那个作为最好的vpa(这尼玛也太简单了吧？？？？)
// 时间相等（一般只在测试环境中出现），按照字符顺序比较名字大小
func stronger(a, b *vpa_types.VerticalPodAutoscaler) bool {
  // Assume a is not nil and each valid object is before nil object.
  if b == nil {
  return true
  }
  // Compare creation timestamps of the VPA objects. This is the clue of the stronger logic.
  var aTime, bTime meta.Time
  aTime = a.GetCreationTimestamp()
  bTime = b.GetCreationTimestamp()
  if !aTime.Equal(&bTime) {
  return aTime.Before(&bTime)
  }
  // If the timestamps are the same (unlikely, but possible e.g. in test environments): compare by name to have a complete deterministic order.
  return a.GetName() < b.GetName()
 }

```

```
// GetControllingVPAForPod chooses the earliest created VPA from the input list that matches the given Pod.
// GetControllingVPAForPod从输入列表中选择与给定Pod匹配的最早创建的VPA。
func GetControllingVPAForPod(pod *core.Pod, vpas []*VpaWithSelector) *VpaWithSelector {
 var controlling *VpaWithSelector
 var controllingVpa *vpa_types.VerticalPodAutoscaler
 // Choose the strongest VPA from the ones that match this Pod.
 // 从这些匹配的pod中选择最好的VPA
 for _, vpaWithSelector := range vpas {
 if PodMatchesVPA(pod, vpaWithSelector) && stronger(vpaWithSelector.Vpa, controllingVpa) {
 // 哪个创的比较早选择哪个
 controlling = vpaWithSelector
 controllingVpa = controlling.Vpa
 }
 }
 return controlling
}
```

第二部分：

第二部分处理推荐请求那部分，主要还是Apply这个函数。

实现建议后处理。

```
// RecommendationProcessor post-processes recommendation adjusting it to limits and environment context
// RecommendationProcessor对recommendation进行后处理，以根据限制和环境上下文进行调整
type RecommendationProcessor interface {
 // Apply processes and updates recommendation for given pod, based on container limits,
 // VPA policy and possibly other internal RecommendationProcessor context.
 // Must return a non-nil pointer to RecommendedPodResources or error.
 // 基于container的限制，Apply处理和更新给定pod的recommendation，VPA策略或者是内部RecommendationProcessor的前后关系
 // 必须返回一个非null的指针指向RecommendationPodResources或错误。
 Apply(podRecommendation *vpa_types.RecommendedPodResources,
 policy *vpa_types.PodResourcePolicy,
 conditions []vpa_types.VerticalPodAutoscalerCondition,
 pod *v1.Pod) (*vpa_types.RecommendedPodResources, ContainerToAnnotationsMap, error)
```

```
}
```

第三部分：

获取容器和pod的限制部分：

```
// LimitRangeCalculator calculates limit range items that has the same effect as all limit range items present in the cluster.
// 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。
type LimitRangeCalculator interface {
  // GetContainerLimitRangeItem returns LimitRangeItem that describes limitation on container limits in the given namespace.
  // GetContainerLimitRangeItem返回LimitRangeItem，该限制描述的是给定名称空间中对container的限制。
  GetContainerLimitRangeItem(namespace string) (*core.LimitRangeItem, error)
  // GetPodLimitRangeItem returns LimitRangeItem that describes limitation on pod limits in the given namespace.
  // GetPodLimitRangeItem返回LimitRangeItem，它描述给定名称空间中对pod的限制。
  GetPodLimitRangeItem(namespace string) (*core.LimitRangeItem, error)
}
```

第四部分

第四个地方GetContainersResources方法按照给定pod.Spec中指定的顺序返回给定pod中每个容器的recommended资源。

```
// GetContainersResources returns the recommended resources for each container in the given pod in the same order they are specified in the pod.Spec.
// GetContainersResources按照给定pod.Spec中指定的顺序返回给定pod中每个容器的recommended资源。
func GetContainersResources(pod *core.Pod, podRecommendation vpa_types.RecommendedPodResources, limitRange *core.LimitRangeItem,
  annotations vpa_api_util.ContainerToAnnotationsMap) []vpa_api_util.ContainerResources {
  resources := make([]vpa_api_util.ContainerResources, len(pod.Spec.Containers))
  for i, container := range pod.Spec.Containers {
  recommendation := vpa_api_util.GetRecommendationForContainer(container.Name, &podRecommendation)
  // 从给定的容器中获得匹配的recommendation
```

```
 if recommendation == nil {
  klog.V(2).Infof("no matching recommendation found for container %s", container.Name)
  continue
  }
  resources[i].Requests = recommendation.Target
  defaultLimit := core.ResourceList{}
  // 获取资源
  if limitRange != nil {
  defaultLimit = limitRange.Default
  }
  // 赋值默认限制
  proportionalLimits, limitAnnotations := vpa_api_util.GetProportionalLimit(container.Resources.Limits, container.Resources.Requests, recommendation.Target, defaultLimit)
  // 获得相应比例下对内存和cpu的限制
  if proportionalLimits != nil {
  resources[i].Limits = proportionalLimits
  // 进行存储
  if len(limitAnnotations) > 0 {
  annotations[container.Name] = append(annotations[container.Name], limitAnnotations...)
  }
  }
  }
  return resources
 }
```

11. 第十一步：创建 Admission Server（关键）

```
//十一步：Admission Server
// 创建Admission Server服务as
as := logic.NewAdmissionServer(recommendationProvider, podPreprocessor, vpaPreprocessor, limitRangeCalculator)
```

AdmissionServer是一个Admission Webhook服务，它根据VPA的建议信息修改Pod资源请求.

- server.go

```
vertical-pod-autoscaler/pkg/admission-controller/logic/server.go
```

server.go主要功能就是实现服务的启动。

```
// AdmissionServer is an admission webhook server that modifies pod resources request based on VPA recommendation
type AdmissionServer struct {
 recommendationProvider RecommendationProvider
 podPreProcessor PodPreProcessor
 vpaPreProcessor VpaPreProcessor
 limitsChecker limitrange.LimitRangeCalculator
}

// NewAdmissionServer constructs new AdmissionServer
func NewAdmissionServer(recommendationProvider RecommendationProvider, podPreProcessor PodPreProcessor, vpaPreProcessor VpaPreProcessor, limitsChecker limitrange.LimitRangeCalculator) *AdmissionServer {
  return &AdmissionServer{recommendationProvider, podPreProcessor, vpaPreProcessor, limitsChecker}
}
```

main.go中服务的启动代码为：

```
//handle函数来处理服务传递（http）
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
 as.Serve(w, r)
 // 启动服务
 healthCheck.UpdateLastActivity()
 // 更新healthCheck，持续监控
})
```

as是AdmissionServer的结构返回体，as.server来启动服务。简单看了这段代码后，我们回到server.go中，来看server函数的执行过程：

```
// Serve is a handler function of AdmissionServer
// handler的服务函数，用于启动AdmissionServer
func (s *AdmissionServer) Serve(w http.ResponseWriter, r *http.Request) {
 timer := metrics_admission.NewAdmissionLatency()
 // 更新监控时间
 var body []byte
 if r.Body != nil {
 if data, err := ioutil.ReadAll(r.Body); err == nil {
```

```
 body = data
  // 提取请求信息
  }
  }

  // verify the content type is accurate
  // 证明收到的请求是没问题的
  contentType := r.Header.Get("Content-Type")
  if contentType != "application/json" {
  klog.Errorf("contentType=%s, expect application/json", contentType)
  timer.Observe(metrics_admission.Error, metrics_admission.Unknown)
  return
  }

  reviewResponse, status, resource := s.admit(body)
  // admit函数从body中提取响应信息，辨别出是pod的响应还是vpa的响应，返回响应（里面存有patch信息），状态和资源类型
  ar := v1beta1.AdmissionReview{
  Response: reviewResponse,
  }
  // 进行回调
  resp, err := json.Marshal(ar)
  if err != nil {
  klog.Error(err)
  timer.Observe(metrics_admission.Error, resource)
  // 进行监控，返回错误
  return
  }

  if _, err := w.Write(resp); err != nil {
  // 通过w将resp写入固定的路径中
  klog.Error(err)
  timer.Observe(metrics_admission.Error, resource)
  // 进行监控 返回错误
  return
  }

  timer.Observe(status, resource)
  // 进行监控
}
```

admit函数能从body中提取响应信息：

```
1  func (s *AdmissionServer) admit(data []byte) (*v1beta1.AdmissionResponse,
metrics_admission.AdmissionStatus, metrics_admission.AdmissionResource) {
2   // we don't block the admission by default, even on unparsable JSON
3   // 默认情况下，即使在无法解析的JSON上，我们也不会阻止访问
4   response := v1beta1.AdmissionResponse{}
5   // 访问的响应
6   response.Allowed = true
7   // 将响应设为允许状态
8
9   ar := v1beta1.AdmissionReview{}
10   // admission请求
11   if err := json.Unmarshal(data, &ar); err != nil {
12   // 如果json无法解析，则返回响应和错误信息，metrics_admission.Error, metrics
_admission.Unknown两个参数告诉监控系统发生错误
13   klog.Error(err)
14   return &response, metrics_admission.Error, metrics_admission.Unknown
15   }
16   // The externalAdmissionHookConfiguration registered via selfRegistrati
on
17   // asks the kube-apiserver only to send admission requests regarding po
ds & VPA objects.
18   // 请求kube-apiserver，要求其只发送有关pods和vpa对象的admission请求
19   podResource := metav1.GroupVersionResource{Group: "", Version: "v1", Re
source: "pods"}
20   vpaGroupResource := metav1.GroupResource{Group: "autoscaling.k8s.io", R
esource: "verticalpodautoscalers"}
21
22   var patches []patchRecord
23   var err error
24   resource := metrics_admission.Unknown
25
26   admittedGroupResource := metav1.GroupResource{
27   Group: ar.Request.Resource.Group,
28   Resource: ar.Request.Resource.Resource,
29   }
30
31   if ar.Request.Resource == podResource {
32   // admission请求和从kube-apiserver获取到的pod的admission请求相同
```

```
33  patches, err = s.getPatchesForPodResourceRequest(ar.Request.Object.Raw, ar.Request.Namespace)
34  // 从pod的资源请求中获取填补信息patches
35  resource = metrics_admission.Pod
36  // 将资源类型赋值为pod
37  } else if admittedGroupResource == vpaGroupResource {
38  // admission请求和从kube-apiserver获取到的vpa的admission请求相同
39  patches, err = s.getPatchesForVPADefaults(ar.Request.Object.Raw, ar.Request.Operation == v1beta1.Create)
40  // 从vpa的资源请求中获取填补信息patches
41  resource = metrics_admission.Vpa
42  // 将资源类型赋值为vpa
43  // we don't let in problematic VPA objects - late validation
44  if err != nil {
45  status := metav1.Status{}
46  status.Status = "Failure"
47  status.Message = err.Error()
48  response.Result = &status
49  response.Allowed = false
50  }
51  } else {
52  patches, err = nil, fmt.Errorf("expected the resource to be one of: %v, %v", podResource, vpaGroupResource)
53  // 如果两个资源都不是，则会输出报错
54  }
55
56  if err != nil {
57  klog.Error(err)
58  return &response, metrics_admission.Error, resource
59  }
60
61  if len(patches) > 0 {
62  patch, err := json.Marshal(patches)
63  // 解析patches
64  if err != nil {
65  klog.Errorf("Cannot marshal the patch %v: %v", patches, err)
66  return &response, metrics_admission.Error, resource
67  }
68  patchType := v1beta1.PatchTypeJSONPatch
69  response.PatchType = &patchType
70  response.Patch = patch
```

```
71  // 解析得到的patch赋值给response用于响应
72  klog.V(4).Infof("Sending patches: %v", patches)
73  }
74
75  var status metrics_admission.AdmissionStatus
76  if len(patches) > 0 {
77  status = metrics_admission.Applied
78  } else {
79  status = metrics_admission.Skipped
80  }
81  if resource == metrics_admission.Pod {
82  metrics_admission.OnAdmittedPod(status == metrics_admission.Applied)
83  }
84
85  return &response, status, resource
86 }
```