

kube-scheduler 源码分析

📅 2019-10-21

kube-scheduler 的设计

Kube-scheduler 是 kubernetes 的核心组件之一，也是所有核心组件之间功能比较单一的，其代码也相对容易理解。kube-scheduler 的目的就是为每一个 pod 选择一个合适的 node，整体流程可以概括为三步，获取未调度的 podList，通过执行一系列调度算法为 pod 选择一个合适的 node，提交数据到 apiserver，其核心则是一系列调度算法的设计与执行。

官方对 kube-scheduler 的调度流程描述 [The Kubernetes Scheduler](#):

```
1  For given pod:
2
3  +-----+
4  |      Schedulable nodes:      |
5  |                               |
6  | +-----+ +-----+ +-----+ |
7  | | node 1 | | node 2 | | node 3 | |
8  | +-----+ +-----+ +-----+ |
9  |                               |
10 +-----+
11 |
12 |
13 v
14 +-----+
15
16 Pred. filters: node 3 doesn't have enough resource
17
18 +-----+
19 |
20 |
21 v
22 +-----+
23 |      remaining nodes:      |
24 | +-----+ +-----+ |
25 | | node 1 | | node 2 | |
26 | +-----+ +-----+ |
27 |                               |
28 +-----+
29 |
30 |
31 v
32 +-----+
33
34 Priority function:  node 1: p=2
35                   node 2: p=5
36
37 +-----+
38 |
39 |
40 v
41 select max{node priority} = node 2
```

kube-scheduler 目前包含两部分调度算法 predicates 和 priorities，首先执行 predicates 算法过滤部分 node 然后执行 priorities 算法为所有 node 打分，最后从所有 node 中选出分数最高的最为最佳的 node。

kube-scheduler 源码分析

kubernetes 版本: v1.16

kubernetes 中所有组件的启动流程都是类似的，首先会解析命令行参数、添加默认值，kube-scheduler 的默认参数在

k8s.io/kubernetes/pkg/scheduler/apis/config/v1alpha1/defaults.go 中定义的。然后会执行 run 方法启动主逻辑，下面直接看 kube-scheduler 的主逻辑 run 方法执行过程。

Run() 方法主要做了以下工作：

- 初始化 scheduler 对象
- 启动 kube-scheduler server，kube-scheduler 监听 10251 和 10259 端口，10251 端口不需要认证，可以获取 healthz metrics 等信息，10259 为安全端口，需要认证
- 启动所有的 informer
- 执行 sched.Run() 方法，执行主调度逻辑

[k8s.io/kubernetes/cmd/kube-scheduler/app/server.go:160](https://k8s.io/kubernetes/cmd/kube-scheduler/app/server.go#L160)

```

1  func Run(cc schedulerconfig.CompletedConfig, stopCh <-chan struct{}, registryOptions ...Option) error {
2      .....
3      // 1、初始化 scheduler 对象
4      sched, err := scheduler.New(.....)
5      if err != nil {
6          return err
7      }
8
9      // 2、启动事件广播
10     if cc.Broadcaster != nil && cc.EventClient != nil {
11         cc.Broadcaster.StartRecordingToSink(stopCh)
12     }
13     if cc.LeaderElectionBroadcaster != nil && cc.CoreEventClient != nil {
14         cc.LeaderElectionBroadcaster.StartRecordingToSink(&corev1.EventSinkImpl{Interface: cc.CoreEventClient.Events("")})
15     }
16
17     .....
18     // 3、启动 http server
19     if cc.InsecureServing != nil {
20         separateMetrics := cc.InsecureMetricsServing != nil
21         handler := buildHandlerChain(newHealthzHandler(&cc.ComponentConfig, separateMetrics, checks...), nil, nil)
22         if err := cc.InsecureServing.Serve(handler, 0, stopCh); err != nil {
23             return fmt.Errorf("failed to start healthz server: %v", err)
24         }
25     }
26     .....
27     // 4、启动所有 informer
28     go cc.PodInformer.Informer().Run(stopCh)
29     cc.InformerFactory.Start(stopCh)
30
31     cc.InformerFactory.WaitForCacheSync(stopCh)
32
33     run := func(ctx context.Context) {
34         sched.Run()
35         <-ctx.Done()
36     }
37
38     ctx, cancel := context.WithCancel(context.TODO()) // TODO once Run() accepts a context, it should be used here
39     defer cancel()
40     go func() {

```

```

41     select {
42     case <-stopCh:
43         cancel()
44     case <-ctx.Done():
45     }
46 }()
47
48 // 5、选举 leader
49 if cc.LeaderElection != nil {
50     .....
51 }
52 // 6、执行 sched.Run() 方法
53 run(ctx)
54 return fmt.Errorf("finished without leader elect")
55 }

```

下面看一下 scheduler.New() 方法是如何初始化 scheduler 结构体的，该方法主要的功能是初始化默认的调度算法以及默认的调度器 GenericScheduler。

- 创建 scheduler 配置文件
- 根据默认的 DefaultProvider 初始化 schedulerAlgorithmSource 然后加载默认的预选及优选算法，然后初始化 GenericScheduler
- 若启动参数提供了 policy config 则使用其覆盖默认的预选及优选算法并初始化 GenericScheduler，不过该参数现已被弃用

k8s.io/kubernetes/pkg/scheduler/scheduler.go:166

```

1 func New(.....) (*Scheduler, error) {
2     .....
3     // 1、创建 scheduler 的配置文件
4     configurator := factory.NewConfigFactory(&factory.ConfigFactoryArgs{
5         .....
6     })
7     var config *factory.Config
8     source := schedulerAlgorithmSource
9     // 2、加载默认的调度算法
10    switch {
11    case source.Provider != nil:
12        // 使用默认的 "DefaultProvider" 初始化 config
13        sc, err := configurator.CreateFromProvider(*source.Provider)
14        if err != nil {
15            return nil, fmt.Errorf("couldn't create scheduler using provider %q: %v", *source.Provider, err)
16        }
17        config = sc
18    case source.Policy != nil:
19        // 通过启动时指定的 policy source 加载 config
20        .....
21        config = sc
22    default:
23        return nil, fmt.Errorf("unsupported algorithm source: %v", source)
24    }
25    // Additional tweaks to the config produced by the configurator.
26    config.Recorder = recorder
27    config.DisablePreemption = options.disablePreemption
28    config.StopEverything = stopCh
29
30    // 3.创建 scheduler 对象
31    sched := NewFromConfig(config)
32    .....
33    return sched, nil
34 }

```

下面是 pod informer 的启动逻辑，只监听 status.phase 不为 succeeded 以及 failed 状态的 pod，即非 terminating 的 pod。

k8s.io/kubernetes/pkg/scheduler/factory/factory.go:527

```

1 func NewPodInformer(client clientset.Interface, resyncPeriod time.Duration) coreinformers.PodInformer {
2     selector := fields.ParseSelectorOrDie(
3         "status.phase!=" + string(v1.PodSucceeded) +
4         ",status.phase!=" + string(v1.PodFailed))
5     lw := cache.NewListWatchFromClient(client.CoreV1().RESTClient(), string(v1.ResourcePods), metav1.NamespaceAll, selector)
6     return &podInformer{
7         informer: cache.NewSharedIndexInformer(lw, &v1.Pod{}, resyncPeriod, cache.Indexers{cache.NamespaceIndex: cache.MetaNamespaceIndexer}),
8     }
9 }

```

然后继续看 Run() 方法中最后执行的 sched.Run() 调度循环逻辑，若 informer 中的 cache 同步完成后会启动一个循环逻辑执行 sched.scheduleOne 方法。

k8s.io/kubernetes/pkg/scheduler/scheduler.go:313

```

1 func (sched *Scheduler) Run() {
2     if !sched.config.WaitForCacheSync() {
3         return
4     }
5
6     go wait.Until(sched.scheduleOne, 0, sched.config.StopEverything)
7 }

```

scheduleOne() 每次对一个 pod 进行调度，主要有以下步骤：

- 从 scheduler 调度队列中取出一个 pod，如果该 pod 处于删除状态则跳过
- 执行调度逻辑 sched.schedule() 返回通过预算及优选算法过滤后选出的最佳 node
- 如果过滤算法没有选出合适的 node，则返回 core.FitError
- 若没有合适的 node 会判断是否启用了抢占策略，若启用了则执行抢占机制
- 判断是否需要 VolumeScheduling 特性
- 执行 reserve plugin
- pod 对应的 spec.NodeName 写上 scheduler 最终选择的 node，更新 scheduler cache
- 请求 apiserver 异步处理最终的绑定操作，写入到 etcd
- 执行 permit plugin
- 执行 prebind plugin
- 执行 postbind plugin

k8s.io/kubernetes/pkg/scheduler/scheduler.go:515

```

1 func (sched *Scheduler) scheduleOne() {
2     fwk := sched.Framework
3
4     pod := sched.NextPod()
5     if pod == nil {
6         return
7     }
8     // 1.判断 pod 是否处于删除状态
9     if pod.DeletionTimestamp != nil {
10         .....
11     }
12
13     // 2.执行调度策略选择 node
14     start := time.Now()
15     pluginContext := framework.NewPluginContext()
16     scheduleResult, err := sched.schedule(pod, pluginContext)

```

```

17  if err != nil {
18      if fitError, ok := err.(*core.FitError); ok {
19          // 3.若启用抢占机制则执行
20          if sched.DisablePreemption {
21              .....
22          } else {
23              preemptionStartTime := time.Now()
24              sched.preempt(pluginContext, fwk, pod, fitError)
25              .....
26          }
27          .....
28          metrics.PodScheduleFailures.Inc()
29      } else {
30          klog.Errorf("error selecting node for pod: %v", err)
31          metrics.PodScheduleErrors.Inc()
32      }
33      return
34  }
35  .....
36  assumedPod := pod.DeepCopy()
37
38  // 4.判断是否需要 VolumeScheduling 特性
39  allBound, err := sched.assumeVolumes(assumedPod, scheduleResult.SuggestedHost)
40  if err != nil {
41      klog.Errorf("error assuming volumes: %v", err)
42      metrics.PodScheduleErrors.Inc()
43      return
44  }
45
46  // 5.执行 "reserve" plugins
47  if sts := fwk.RunReservePlugins(pluginContext, assumedPod, scheduleResult.SuggestedHost); !sts.IsSuccess() {
48      ....
49  }
50
51  // 6.为 pod 设置 NodeName 字段, 更新 scheduler 缓存
52  err = sched.assume(assumedPod, scheduleResult.SuggestedHost)
53  if err != nil {
54      .....
55  }
56
57  // 7.异步请求 apiserver
58  go func() {
59      // Bind volumes first before Pod
60      if !allBound {
61          err := sched.bindVolumes(assumedPod)
62          if err != nil {
63              .....
64              return
65          }
66      }
67
68      // 8.执行 "permit" plugins
69      permitStatus := fwk.RunPermitPlugins(pluginContext, assumedPod, scheduleResult.SuggestedHost)
70      if !permitStatus.IsSuccess() {
71          .....
72      }
73
74      // 9.执行 "prebind" plugins
75      preBindStatus := fwk.RunPreBindPlugins(pluginContext, assumedPod, scheduleResult.SuggestedHost)
76      if !preBindStatus.IsSuccess() {
77          .....
78      }
79
80      err := sched.bind(assumedPod, scheduleResult.SuggestedHost, pluginContext)
81      .....
82      if err != nil {
83          .....

```

```

82     } else {
83         .....
84         // 10.执行 "postbind" plugins
85         fwk.RunPostBindPlugins(pluginContext, assumedPod, scheduleResult.SuggestedHost)
86     }
87     }()
88 }

```

scheduleOne() 中通过调用 sched.schedule() 来执行预选与优选算法处理：

k8s.io/kubernetes/pkg/scheduler/scheduler.go:337

```

1 func (sched *Scheduler) schedule(pod *v1.Pod, pluginContext *framework.PluginContext) (core.ScheduleResult, error) {
2     result, err := sched.Algorithm.Schedule(pod, pluginContext)
3     if err != nil {
4         .....
5     }
6     return result, err
7 }

```

sched.Algorithm 是一个 interface，主要包含四个方法，GenericScheduler 是其具体的实现：

k8s.io/kubernetes/pkg/scheduler/core/generic_scheduler.go:131

```

1 type ScheduleAlgorithm interface {
2     Schedule(*v1.Pod, *framework.PluginContext) (scheduleResult ScheduleResult, err error)
3     Preempt(*framework.PluginContext, *v1.Pod, error) (selectedNode *v1.Node, preemptedPods []*v1.Pod, cleanupNominatedPods []*v1.Pod)
4     Predicates() map[string]predicates.FitPredicate
5     Prioritizers() []priorities.PriorityConfig
6 }

```

- Schedule()：正常调度逻辑，包含预算与优选算法的执行
- Preempt()：抢占策略，在 pod 调度发生失败的时候尝试抢占低优先级的 pod，函数返回发生抢占的 node，被 抢占的 pods 列表，nominated node name 需要被移除的 pods 列表以及 error
- Predicates()：predicates 算法列表
- Prioritizers()：prioritizers 算法列表

kube-scheduler 提供的默认调度为 DefaultProvider，DefaultProvider 配置的 predicates 和 priorities policies 在

k8s.io/kubernetes/pkg/scheduler/algorithmprovider/defaults/defaults.go 中定义，算法具体实现是在

k8s.io/kubernetes/pkg/scheduler/algorithm/predicates/ 和 k8s.io/kubernetes/pkg/scheduler/algorithm/priorities/ 中，默认的算法如下所示：

[pkg/scheduler/algorithmprovider/defaults/defaults.go](https://k8s.io/kubernetes/pkg/scheduler/algorithmprovider/defaults/defaults.go)

```

1 func defaultPredicates() sets.String {
2     return sets.NewString(
3         predicates.NoVolumeZoneConflictPred,
4         predicates.MaxEBSVolumeCountPred,
5         predicates.MaxGCEPDVolumeCountPred,
6         predicates.MaxAzureDiskVolumeCountPred,
7         predicates.MaxCSIVolumeCountPred,
8         predicates.MatchInterPodAffinityPred,
9         predicates.NoDiskConflictPred,
10        predicates.GeneralPred,
11        predicates.CheckNodeMemoryPressurePred,
12        predicates.CheckNodeDiskPressurePred,
13        predicates.CheckNodePIDPressurePred,

```

```

14     predicates.CheckNodeConditionPred,
15     predicates.PodToleratesNodeTaintsPred,
16     predicates.CheckVolumeBindingPred,
17 )
18 }
19
20 func defaultPriorities() sets.String {
21     return sets.NewString(
22         priorities.SelectorSpreadPriority,
23         priorities.InterPodAffinityPriority,
24         priorities.LeastRequestedPriority,
25         priorities.BalancedResourceAllocation,
26         priorities.NodePreferAvoidPodsPriority,
27         priorities.NodeAffinityPriority,
28         priorities.TaintTolerationPriority,
29         priorities.ImageLocalityPriority,
30     )
31 }

```

下面继续看 sched.Algorithm.Schedule() 调用具体调度算法的过程：

- 检查 pod pvc 信息
- 执行 prefilter plugins
- 获取 scheduler cache 的快照，每次调度 pod 时都会获取一次快照
- 执行 g.findNodesThatFit() 预选算法
- 执行 postfilter plugin
- 若 node 为 0 直接返回失败的 error，若 node 数为1 直接返回该 node
- 执行 g.priorityMetaProducer() 获取 metaPrioritiesInterface，计算 pod 的 metadata，检查该 node 上是否有相同 meta 的 pod
- 执行 PrioritizeNodes() 算法
- 执行 g.selectHost() 通过得分选择一个最佳的 node

k8s.io/kubernetes/pkg/scheduler/core/generic_scheduler.go:186

```

1  func (g *genericScheduler) Schedule(pod *v1.Pod, pluginContext *framework.PluginContext) (result ScheduleResult, err error) {
2      .....
3      // 1.检查 pod pvc
4      if err := podPassesBasicChecks(pod, g.pvcLister); err != nil {
5          return result, err
6      }
7
8      // 2.执行 "prefilter" plugins
9      preFilterStatus := g.framework.RunPreFilterPlugins(pluginContext, pod)
10     if !preFilterStatus.IsSuccess() {
11         return result, preFilterStatus.AsError()
12     }
13
14     // 3.获取 node 数量
15     numNodes := g.cache.NodeTree().NumNodes()
16     if numNodes == 0 {
17         return result, ErrNoNodesAvailable
18     }
19
20     // 4.快照 node 信息
21     if err := g.snapshot(); err != nil {
22         return result, err
23     }
24
25     // 5.执行预选算法
26     startPredicateEvalTime := time.Now()
27     filteredNodes, failedPredicateMap, filteredNodesStatuses, err := g.findNodesThatFit(pluginContext, pod)
28     if err != nil {

```

```
29     return result, err
30 }
31 // 6.执行 "postfilter" plugins
32 postfilterStatus := g.framework.RunPostFilterPlugins(pluginContext, pod, filteredNodes, filteredNodesStatuses)
33 if !postfilterStatus.IsSuccess() {
34     return result, postfilterStatus.AsError()
35 }
36
37 // 7.预选后没有合适的 node 直接返回
38 if len(filteredNodes) == 0 {
39     .....
40 }
41
42 startPriorityEvalTime := time.Now()
43 // 8.若只有一个 node 则直接返回该 node
44 if len(filteredNodes) == 1 {
45     return ScheduleResult{
46         SuggestedHost: filteredNodes[0].Name,
47         EvaluatedNodes: 1 + len(failedPredicateMap),
48         FeasibleNodes: 1,
49     }, nil
50 }
51
52 // 9.获取 pod meta 信息, 执行优选算法
53 metaPrioritiesInterface := g.priorityMetaProducer(pod, g.nodeInfoSnapshot.NodeInfoMap)
54 priorityList, err := PrioritizeNodes(pod, g.nodeInfoSnapshot.NodeInfoMap, metaPrioritiesInterface, g.prioritizers, filteredNodes, g.exte
55 if err != nil {
56     return result, err
57 }
58
59 // 10.根据打分选择最佳的 node
60 host, err := g.selectHost(priorityList)
61 trace.Step("Selecting host done")
62 return ScheduleResult{
63     SuggestedHost: host,
64     EvaluatedNodes: len(filteredNodes) + len(failedPredicateMap),
65     FeasibleNodes: len(filteredNodes),
66 }, err
67 }
```

至此，scheduler 的整个过程分析完毕。

总结

本文主要对于 kube-scheduler v1.16 的调度流程进行了分析，但其中有大量的细节都暂未提及，包括预选算法以及优选算法的具体实现、优先级与抢占调度的实现、framework 的使用及实现，因篇幅有限，部分内容会在后文继续说明。

参考：

[The Kubernetes Scheduler](#)

[scheduling design proposals](#)

kube-scheduler

© 2020  tianfeiyu [陕ICP备15001765号-1](#)

由 [Hexo](#) 强力驱动 | 主题 — [NexT.Mist v5.1.4](#)



昵称

邮箱

网址(http://)

Just go go

表情 | 预览

回复

1 评论



梵音 Chrome 78.0.3904.108 Mac OS 10.15.1

2019-11-29

赞👍

回复

Powered By Valine
v1.3.9

K8s 集群规划、部署和运营

CNCF 认证、100% 开源、简单易
用、一键部署、离线安装

FIT2CLOUD 飞致云