Introduction to the Scheduler Framework

Get introduced to the Kubernetes scheduler framework.

We'll cover the following

The scheduler framework

For a well-designed open-source project, one distinguishing feature is its extensibility. This would help users and developers fulfill their own business needs easily without breaking upstream source codes and contribute back.

Kubernetes provides a series of ways to customize the scheduler, such as multiple schedulers and scheduler extenders. Multiple schedulers can have good performance, but they bring in high maintenance costs, resource competition problems, etc. It also has a steep developing curve—developers need to have a comprehensive understanding of the scheduling process. While the scheduler extender is easy to extend and maintain, it only provides limited extension points and has poor performance. To deal with this dilemma and provide a friendly developing experience, the Kubernetes community adopted a pluggable architecture (the scheduler framework) for the scheduler starting from v1.16.

The **scheduler framework** defines a variety of plugin APIs for extension points in the scheduling process. This design allows scheduling features to be implemented as independent plugins, while

keeping the scheduling process lightweight, extensive, and maintainable. Below are all the registered in-tree scheduling plugins (**lines 14–34**). We can configure the default scheduler to enable or disable the use of certain plugins at runtime. We can also set custom plugin arguments for each plugin. This is configurable by a scheduling profile, which is part of a scheduler configuration.

```
 1  // NewInTreeRegistry builds the registry with all the in-tree plugins.
 2  // A scheduler that runs out of tree plugins can register additional plugins
 3  // through the WithFrameworkOutOfTreeRegistry option.
 4  func NewInTreeRegistry() runtime.Registry {
 5      fts := plfeature.Features{
 6          EnableReadWriteOncePod:                      feature.DefaultFeatureGate.Enabled(features.ReadWr
 7          EnableVolumeCapacityPriority:               feature.DefaultFeatureGate.Enabled(features.Volume
 8          EnableMinDomainsInPodTopologySpread:        feature.DefaultFeatureGate.Enabled(features.MinDor
 9          EnableNodeInclusionPolicyInPodTopologySpread: feature.DefaultFeatureGate.Enabled(features.NodeIr
10          EnableMatchLabelKeysInPodTopologySpread:    feature.DefaultFeatureGate.Enabled(features.MatchL
11      }
12
13      return runtime.Registry{
14          selectorspread.Name:                 selectorspread.New,
15          imagelocality.Name:                  imagelocality.New,
16          tainttoleration.Name:                tainttoleration.New,
17          nodename.Name:                       nodename.New,
18          nodeports.Name:                      nodeports.New,
19          nodeaffinity.Name:                   nodeaffinity.New,
20          podtopologyspread.Name:              runtime.FactoryAdapter(fts, podtopologyspread.New),
21          nodeunschedulable.Name:              nodeunschedulable.New,
22          noderesources.Name:                  runtime.FactoryAdapter(fts, noderesources.NewFit),
23          noderesources.BalancedAllocationName: runtime.FactoryAdapter(fts, noderesources.NewBalancedAlloc
24          volumebinding.Name:                  runtime.FactoryAdapter(fts, volumebinding.New),
25          volumerestrictions.Name:             runtime.FactoryAdapter(fts, volumerestrictions.New),
26          volumezone.Name:                     volumezone.New,
27          nodevolumelimits.CSIName:            runtime.FactoryAdapter(fts, nodevolumelimits.NewCSI),
28          nodevolumelimits.EBSName:            runtime.FactoryAdapter(fts, nodevolumelimits.NewEBS),
29          nodevolumelimits.GCEPDName:          runtime.FactoryAdapter(fts, nodevolumelimits.NewGCEPD),
30          nodevolumelimits.AzureDiskName:      runtime.FactoryAdapter(fts, nodevolumelimits.NewAzureDisk)
31          interpodaffinity.Name:               interpodaffinity.New,
```

In-tree scheduling plugins

Developers can implement their own plugins by implementing the APIs defined at each extension point. For each plugin, it can implement one or multiple interfaces of extension points, such as `Filter` and `Score`. This means that a plugin could be invoked once or multiple times during a scheduling process. The plugins can be informational or they help make a scheduling decision, such as filtering nodes.

These scheduling plugins (both in-tree and out-of-tree) are compiled together into the scheduler. The vanilla default scheduler does provide a hook to register out-of-tree plugins. During the scheduling process, the scheduler framework will call the registered plugins one by one at each extension point. This way, external plugins are integrated into the scheduler framework.

It's worth noting that the scheduler framework is written in Go, which means that out-of-tree plugins need to be implemented in Go as well. This is different from multiple schedulers and scheduler extenders, where we can choose our familiar languages to implement the logic.

Framework workflow

In the scheduler framework, a scheduling process is divided into two phases, the scheduling cycle and the binding cycle.

The scheduling cycle plays a crucial role in the scheduling process. Its main task is to make an optimal scheduling decision and select the node of best fit. During the scheduling cycle, the scheduler filters all the nodes in the cluster to find **feasible nodes** that can match all the requirements of a Pod. Next, all those feasible nodes will be ranked based on normalized scores. The chosen node gets the highest score.

After that, the scheduler will bind this node to the Pod by setting the `spec.nodeName`. This is called the **binding cycle**. When the Pod gets bound, the `kubelet` on that node will get notified to start the Pod.

In each cycle, there are multiple extension points, where we can insert our own out-of-tree plugins. All the extension points are shown below. These extension points are executed sequentially. For example, the `QueueSort` extension point is executed first to fetch a Pod with the highest priority from the scheduling queue. Then, the `PreFilter` extension point will be invoked.

```
Scheduling Cycle
    QueueSort
    PreFilter
    Filter
    PostFilter
    PreScore
    Score
    NormalizeScore
    Reserve
    Permit

Binding Cycle
    PreBind
    Bind
    PostBind
```

Scheduling cycles

The scheduling cycles are executed serially to avoid resource competition problems, while binding cycles for different Pods are run concurrently.

Now, let's see what each extension point does in the scheduling process.

Scheduling cycle

Now, let's see the extension points in the scheduling cycle.

The `QueueSort` plugins

The `QueueSort` plugins are invoked first by the scheduler. These plugins are used to sort all the Pods pending in the scheduling queue. In the scheduling cycle, the Pods are scheduled one by one. Only one Pod is in the scheduling cycle at a time. The `QueueSort` plugins will sort all the Pods to determine the next Pod to be scheduled.

The default in-tree `QueueSort` plugin is `PrioritySort`, which sorts Pods based on their priority. The Pod with the highest priority ranks first. We can also implement a `Less` function (as shown below) for the heap as well. However, only one `QueueSort` plugin can be enabled at a time. As a result, we'd normally just use the default built-in `PrioritySort` plugin.

```
1  // QueueSortPlugin is an interface that must be implemented by "QueueSort" plugins.
2  // These plugins are used to sort pods in the scheduling queue. Only one queue sort
3  // plugin may be enabled at a time.
4  type QueueSortPlugin interface {
5     Plugin
6     // Less are used to sort pods in the scheduling queue.
7     Less(*QueuedPodInfo, *QueuedPodInfo) bool
8  }
```

The QueueSortPlugin interface

The `PreFilter` plugins

These `PreFilter` plugins check the preconditions of the Pods before filtering nodes. They can also be used to check certain conditions that the Pod or cluster must meet. If one of the `PreFilter` plugins returns an error, the scheduling cycle is terminated. The Pod will be marked as unschedule-able. Only when all the `PreFilter` plugins pass the Pod, does the scheduling cycle enter the next extension point `Filter`.

The `Filter` plugins

Just as the extension point name says, these `Filter` plugins are used to filter out all the nodes in the cluster that cannot run the Pod. All these plugins are invoked in their configured order. The nodes are evaluated concurrently to speed up the filtering. If a node is marked as infeasible by one of the plugins, the remaining plugins will not be called for that node any more. A Pod will be marked as unschedule-able if no feasible nodes are found by all the filters.

The `Filter` plugin is the predicate logic in the scheduler v1. It's used to filter out nodes that cannot schedule the Pod. To improve efficiency, we can configure the execution order of the `Filter` plugins. We can prioritize filtering policies that can filter out a large number of nodes, minimizing the filtering policies to be executed subsequently. For example, we can prioritize the filtering policies for `nodeSelector` to filter out a large number of nodes. The node executes filtering policies concurrently, so `Filter` plugins are called multiple times in a scheduling cycle.

The `PostFilter` plugins

When no feasible nodes are found, the `PostFilter` plugins will be called. One typical `PostFilter` plugin is the in-tree `DefaultPreemption` plugin, which tries to preempt other Pods and make the Pod schedule-able. If one of the plugins finds candidate nodes for this Pod, other remaining plugins won't be invoked.

The `PreScore` phase

After passing the filtering phase, the scheduling cycle comes into the `PreScore` phase. These plugins will help perform some prescoring work.

The `Score` plugins

Each of the `Score` plugins will give a score within a range of minimum and maximum values to indicate the rank of the node. All scoring plugins must return success or the Pod will be rejected. The score range differs for each plugin. The scheduler does need to normalize the scores before computing a final ranking of nodes. All the scores will be normalized between 0 and 100. After that, the scheduler will count the scores up with the weight of each plugin. The node getting the highest weighted sum of scores wins the selection to run the Pod.

```
// https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L419-L438
// ScoreExtensions is an interface for Score extended functionality.
type ScoreExtensions interface {
    // NormalizeScore() is called for all node scores produced by the same plugin's Score() method.
    // A successful run of NormalizeScore() will update the scores list and return a status of success.

    NormalizeScore(ctx context.Context, state *CycleState, p *v1.Pod, scores NodeScoreList) *Status
}

// ScorePlugin is an interface that must be implemented by Score plugins to rank
// nodes that passed the filtering phase.
type ScorePlugin interface {
    Plugin
    // Score() is called on each filtered node. It must return success and an integer indicating
    // the rank of the node. All scoring plugins must return success or the Pod will be rejected.
    Score(ctx context.Context, state *CycleState, p *v1.Pod, nodeName string) (int64, *Status)

    // ScoreExtensions() returns a ScoreExtensions interface if it implements one, or nothing if does nc
    ScoreExtensions() ScoreExtensions
}
```

Score plugin interface

The `Reserve` plugins

Before we bind a Pod to its designated node, we need to reserve the node first. This is because the binding cycle runs concurrently. The `Reserve` extension point provides two interfaces, `Reserve` and `Unreserve`. The method `Reserve` is used to prevent race conditions during the binding cycle. If one of

the plugins fails to reserve the resources, the method `Unreserve` will be called (in the reverse order of the `Reserve` method calls) in order to clean up the state associated with the reserved Pod.

The `Permit` plugins

The `Permit` plugins are invoked at the end of the scheduling cycle. The Pod can be approved, denied, or delayed for binding to the designated node. Once approved, the Pod is sent for a bind. If any of the `Permit` plugins deny the Pod, it's sent back to the scheduling queue. The delayed operation does have a timeout. If a timeout occurs, the Pod will be denied and sent back to the scheduling queue.

Binding cycle

Now, let's see the extension points in the binding cycle.

The `Bind` plugins

After all the `PreBind` plugins have been called, the `Bind` plugins will be called. These plugins are used to bind a Pod to a designated node by updating the `spec.nodeName`. There could be multiple `Bind` plugins. They'll be called in order. Each plugin may choose whether or not it will handle the real binding work. Once one of them has done the Pod binding, the remaining plugins are skipped.

The `PreBind` and `PostBind` plugins

The `PreBind` and `PostBind` plugins are called before or after the `Bind` phase to perform any work required.

Conclusion

Maintaining a forked repository for a long period of time is never an easy task. It will become harder and harder to keep track of upstream changes and rebase them locally, especially when we have made conflicting and incompatible changes. With the scheduler framework, it's easier to extend the plugins at multiple extension points. For each extension point, we only need to implement a few interfaces, while keeping the core scheduling process unstained. The framework greatly reduces the developing curve and maintenance efforts.

Back

Scheduler Extender

Next

Implementing a Scheduler Framework Plugin