Scheduler Extender

Learn how the Kubernetes scheduler extender works.

We'll cover the following

What is the scheduler extender?

A scheduler extender works as an external service. It isn't another scheduler. Just as the name suggests, it simply extends the default Kubernetes scheduler with more customized features or implementations. That doesn't mean we can customize everything in the default Kubernetes scheduler. But there are some extension points that we can include in our customized implementations.

```
// https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/extender.go#L24-L72
// Extender is an interface for external processes to influence scheduling
// decisions made by Kubernetes. This is typically needed for resources not directly
// managed by Kubernetes.
type Extender interface {
    // Name() returns a unique name that identifies the extender.
    Name() string

    // Filter based on extender-implemented predicate functions. The filtered list is
    // expected to be a subset of the supplied list.
    // The failedNodes and failedAndUnresolvableNodes optionally contains the list
    // of failed nodes and failure reasons, except nodes in the latter are
    // unresolvable.
    Filter(pod *v1.Pod, nodes []*v1.Node) (filteredNodes []*v1.Node, failedNodesMap extenderv1.FailedNod
    // Prioritize based on extender-implemented priority functions
    // The returned scores and weight are used to compute the weighted score for an extender.
    // The weighted scores are added t the scores computed by Kubernetes scheduler. The total
    // scores are used to do the host selection.


    Prioritize(pod *v1.Pod, nodes []*v1.Node) (hostPriorities *extenderv1.HostPriorityList, weight int64
    // Bind() delegates the action of binding a Pod to a node to the extender.
    Bind(binding *v1.Binding) error

    // IsBinder() returns whether this extender is configured for the Bind() method.
    IsBinder() bool

    // sInterested() returns true if at least one extended resource requested by this
    // Pod is managed by this extender.
    IsInterested(pod *v1.Pod) bool
```

Scheduler extender interface

Above is the extender interface, where we can add our extension implementations during the `Filter`, `Preempt`, `Prioritize`, and `Bind` phases. The Kubernetes scheduler extender works like a webhook. When Pod scheduling comes into these four phases, the `kube-scheduler` can call the extension logic registered by the extender.

Now, let's take a closer look at how the scheduler extender works.

How the scheduler extender works

With the scheduler extender, we can change how the `kube-scheduler` makes decisions on Pod scheduling to meet our business needs.

Register scheduler extenders

Firstly, let's see how we register our extenders to the `kube-scheduler`. The flag `--config` in the `kube-scheduler` is the entry where we register our external extenders. Here, we can register a list of extenders when needed. In that configuration file, we specify the parameters used by the `kube-scheduler` to communicate with the extenders, such as the opt-in `Filter` phase and extender service endpoints. Below is a sample configuration file:

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
clientConnection:
  kubeconfig: /etc/kubernetes/scheduler.conf
extenders:
- urlPrefix: "http://127.0.0.1:8080/"
  filterVerb: filter
  bindVerb: bind
  enableHTTPS: false
  nodeCacheCapable: true
  managedResources:
  - name: resource.mycompany.com/gpu
    ignoredByScheduler: false
  ignorable: false
```

Sample scheduler configuration

In the file above, we specify an insecure extender running at `http://127.0.0.1:8080/`. In this extender, we specify the extended resources that are managed by this extender. Only the Pods request at the extended resource `resource.mycompany.com/gpu` will be sent to this extender during the `Filter` phase. The field `managedResources` is optional, which can help us improve the scheduling latency by calling extenders when necessary. Below is a completed struct of an `Extender`, where we can add more extender verbs, such as `Filter` and `Preempt`:

```
// https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/kube-scheduler/config/v1/type
// Extender holds the parameters used to communicate with the extender. If a verb is unspecified/empty,
// it is assumed that the extender chose not to provide that extension.
type Extender struct {
    // URLPrefix at which the extender is available
    URLPrefix string `json:"urlPrefix"`
    // Verb for the filter call, empty if not supported. This verb is appended to the URLPrefix when iss
    FilterVerb string `json:"filterVerb,omitempty"`
    // Verb for the preempt call, empty if not supported. This verb is appended to the URLPrefix when is
    PreemptVerb string `json:"preemptVerb,omitempty"`
    // Verb for the prioritize call, empty if not supported. This verb is appended to the URLPrefix wher
    PrioritizeVerb string `json:"prioritizeVerb,omitempty"`
    // The numeric multiplier for the node scores that the prioritize call generates.
    // The weight should be a positive integer
    Weight int64 `json:"weight,omitempty"`
    // Verb for the bind call, empty if not supported. This verb is appended to the URLPrefix when issu
    // If this method is implemented by the extender, it is the extender's responsibility to bind the po
    // can implement this function.
    BindVerb string `json:"bindVerb,omitempty"`
    // EnableHTTPS specifies whether https should be used to communicate with the extender
    EnableHTTPS bool `json:"enableHTTPS,omitempty"`
    // TLSConfig specifies the transport layer security config
    TLSConfig *ExtenderTLSConfig `json:"tlsConfig,omitempty"`
    // HTTPTimeout specifies the timeout duration for a call to the extender. Filter timeout fails the s
    // timeout is ignored, k8s/other extenders priorities are used to select the node.
    HTTPTimeout metav1.Duration `json:"httpTimeout,omitempty"`
    // NodeCacheCapable specifies that the extender is capable of caching node information,
    // so the scheduler should only send minimal information about the eligible nodes
    // assuming that the extender already cached full details of all nodes in the cluster
    NodeCacheCapable bool `json:"nodeCacheCapable,omitempty"`
    // ManagedResources is a list of extended resources that are managed by
```

Extender struct

These extender verbs will be appended to the `URLPrefix` when the `kube-scheduler` calls the extenders. The verbs for the extender call should exactly match the HTTP handler paths.

A simple extender

The scheduler extender runs as an HTTP(S) service and exposes an accessible endpoint. We can write our implementations in any language we're familiar with. Below is a simple Golang snippet for reference. We can insert our own business logic on **line 49**.

```
 1  package main
 2
 3  import (
 4      "bytes"
 5      "encoding/json"
 6      "io"
 7      "log"
 8      "math/rand"
 9      "net/http"
10      "time"
11
12      v1 "k8s.io/api/core/v1"
13      extenderapi "k8s.io/kube-scheduler/extender/v1"
14  )
```

```

func init() {
    rand.Seed(time.Now().UTC().UnixNano())
}

func Filter(w http.ResponseWriter, r *http.Request) {
    var buf bytes.Buffer
    body := io.TeeReader(r.Body, &buf)
    var extenderArgs extenderapi.ExtenderArgs
    var extenderFilterResult *extenderapi.ExtenderFilterResult
    if err := json.NewDecoder(body).Decode(&extenderArgs); err != nil {
        extenderFilterResult = &extenderapi.ExtenderFilterResult{
            Error: err.Error(),
        }
    } else {
        extenderFilterResult = filter(extenderArgs)
```

Code snippet of the scheduler extender

All the extender functions accept an input with the type `schedulerapi.ExtenderArgs` and return with `*schedulerapi.ExtenderFilterResult`. Within each function, we can do further implementations to satisfy our particular business needs.

The `Filter` and `Prioritize` functions are the most important ones to extend the default Kubernetes scheduler. We can append the `preempt` and `bind` functions as well when needed.

Interactions between the `kube-scheduler` and extenders

Now, let's see the completed interactions between the `kube-scheduler` and scheduler extenders.

The default scheduler calls the registered extenders at certain extension points by sending an HTTP request. Below, the code snippet describes how the request is sent out.

```
// Helper function to send messages to the extender
func (h *HTTPExtender) send(action string, args interface{}, result interface{}) error {
    out, err := json.Marshal(args)
    if err != nil {
        return err
    }

    url := strings.TrimRight(h.extenderURL, "/") + "/" + action

    req, err := http.NewRequest("POST", url, bytes.NewReader(out))
    if err != nil {
        return err
    }

    req.Header.Set("Content-Type", "application/json")

    resp, err := h.client.Do(req)
    if err != nil {
        return err
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        return fmt.Errorf("failed %v with extender at URL %v, code %v", action, url, resp.StatusCode)
    }

    return json.NewDecoder(resp.Body).Decode(result)
}
```

How the scheduler calls the extender

It’s worth noting that the default Kubernetes scheduler keeps rescheduling failed Pods periodically. As a result, the extenders will be called again and again until the Pods get scheduled.

Limitations

The scheduler extender could be a good option for some scenarios, but it does have some limitations, as outline in the list below, which are why we don’t implement a completed extender):

- **Large communication cost and downgraded performance:** The extender is called by HTTP(S). All the data is transferred between them. If the cluster gets lots of nodes, the request payload will have a big size as well. Moreover, all the data needs to be serialized and deserialized, which is time consuming. This will greatly affect the performance of the default scheduler.
- **Limited extension points:** From above, we can see we have only a few options for customizations.
- **Cache sharing:** In the default scheduler, it maintains a local cache to store the status of the whole cluster. Additionally, the caching can’t be shared with external extenders. This brings trouble to scheduling results, especially for concurrent scheduling.

Conclusion

Extending the default scheduler with extenders is a good choice for small clusters with low scheduling efficiency requirements. However, for large or high-performing clusters, it isn’t a good idea to use scheduler extenders.

To address these limitations, a better way is provided by the Kubernetes community, known as the scheduler framework. It resolves the pain points above and is the officially recommended way to extend Kubernetes scheduling.

Back

Multiple Schedulers

Next

Introduction to the Scheduler Framework