Node Affinity

Kubernetes supports many more flexible ways to configure the scheduling processes. One such feature is *node affinity*, which is a more expressive way of the node selector approach described previously that allows specifying rules as either required or preferred. *Required rules* must be met for a Pod to be scheduled to a node, whereas preferred rules only imply preference by increasing the weight for the matching nodes without making them mandatory. In addition, the node affinity feature greatly expands the types of constraints you can express by making the language more expressive with operators such as `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, or `Lt`. Example 6-4 demonstrates how node affinity is declared.

Example 6-4. Pod with node affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:    ❶
        nodeSelectorTerms:
        - matchExpressions:                              ❷
          - key: numberCores
            operator: Gt
            values: [ "3" ]
      preferredDuringSchedulingIgnoredDuringExecution:   ❸
      - weight: 1
        preference:
          matchFields:
          - key: metadata.name
            operator: NotIn
            values: [ "control-plane-node" ]
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
```

❶ Hard requirement that the node must have more than three cores (indicated by a node label) to be considered in the scheduling process. The rule is not reevaluated during execution if the conditions on the node change.

❷ Match on labels. In this example, all nodes are matched that have a label `numberCores` with a value greater than 3.

❸ Soft requirements, which is a list of selectors with weights. For every node, the sum of all weights for matching selectors is calculated, and the highest-valued node is chosen, as long as it matches the hard requirement.

Pod Affinity and Anti-Affinity

Pod affinity is a more powerful way of scheduling and should be used when `nodeSelector` is not enough. This mechanism allows you to constrain which nodes a Pod can run based on label or field matching. It doesn't allow you to express dependencies among Pods to dictate where a Pod should be placed relative to other Pods. To express how Pods should be spread to achieve high availability, or be packed and colocated together to improve latency, you can use Pod affinity and anti-affinity.

Node affinity works at node granularity, but Pod affinity is not limited to nodes and can express rules at various topology levels based on the Pods already running on a node. Using the `topologyKey` field, and the matching labels, it is possible to enforce more fine-grained rules, which combine rules on domains like node, rack, cloud provider zone, and region, as demonstrated in Example 6-5.

Example 6-5. Pod with Pod affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:  ❶
      - labelSelector:                                 ❷
          matchLabels:
            confidential: high
        topologyKey: security-zone                     ❸
    podAntiAffinity:                                   ❹
      preferredDuringSchedulingIgnoredDuringExecution: ❺
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchLabels:
              confidential: none
          topologyKey: kubernetes.io/hostname
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
```

❶ Required rules for the Pod placement concerning other Pods running on the target node.

❷ Label selector to find the Pods to be colocated with.

❸ The nodes on which Pods with labels `confidential=high` are running are supposed to carry a `security-zone` label. The Pod defined here is scheduled to a node with the same label and value.

❹ Anti-affinity rules to find nodes where a Pod would *not* be placed.

❺ Rule describing that the Pod should not (but could) be placed on any node where a Pod with the label `confidential=none` is running.

Similar to node affinity, there are hard and soft requirements for Pod affinity and anti-affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`, respectively. Again, as with node affinity, the `IgnoredDuringExecution` suffix is in the field name, which exists for future extensibility reasons. At the moment, if the labels on the node change and affinity rules are no longer valid, the Pods continue running,[1] but in the future, runtime changes may also be taken into account.

Topology Spread Constraints

Pod affinity rules allow the placement of unlimited Pods to a single topology, whereas Pod anti-affinity disallows Pods to colocate in the same topology. Topology spread constraints give you more fine-grained control to evenly distribute Pods on your cluster and achieve better cluster utilization or high availability of applications.

Let's look at an example to understand how topology spread constraints can help. Let's suppose we have an application with two replicas and a two-node cluster. To avoid downtime and a single point of failure, we can use Pod anti-affinity rules to prevent the coexistence of the Pods on the same node and spread them into both nodes. While this setup makes sense, it will prevent you from performing rolling upgrades because the third replacement Pod cannot be placed on the existing nodes because of the Pod anti-affinity constraints. We will have to either add another node or change the Deployment strategy from rolling to recreate. Topology spread constraints would be a better solution in this situation as they allow you to tolerate some degree of uneven Pod distribution when the cluster is running out of resources. Example 6-6 allows the placement of the third rolling deployment Pod on one of the two nodes because it allows imbalances—i.e., a skew of one Pod.

1 However, if node labels change and allow for unscheduled Pods to match their node affinity selector, these Pods are scheduled on this node.

Example 6-6. Pod with topology spread constraints

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    app: bar
spec:
  topologySpreadConstraints:                    ❶
  - maxSkew: 1                                  ❷
    topologyKey: topology.kubernetes.io/zone    ❸
    whenUnsatisfiable: DoNotSchedule            ❹
    labelSelector:                              ❺
      matchLabels:
        app: bar
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
```

❶ Topology spread constraints are defined in the `topologySpreadConstraints` field of the Pod spec.

❷ `maxSkew` defines the maximum degree to which Pods can be unevenly distributed in the topology.

❸ A topology domain is a logical unit of your infrastructure. And a `topologyKey` is the key of the Node label where identical values are considered to be in the same topology.

❹ The `whenUnsatisfiable` field defines what action should be taken when `maxSkew` can't be satisfied. `DoNotSchedule` is a hard constraint preventing the scheduling of Pods, whereas `ScheduleAnyway` is a soft constraint that gives scheduling priority to nodes that reduce cluster imbalance.

❺ `labelSelector` Pods that match this selector are grouped together and counted when spreading them to satisfy the constraint.

Topology spread constraints is a feature that is still evolving at the time of this writing. Built-in cluster-level topology spread constraints allow certain imbalances based on default Kubernetes labels and give you the ability to honor or ignore node affinity and taint policies.

Taints and Tolerations

A more advanced feature that controls where Pods can be scheduled and allowed to run is based on taints and tolerations. While node affinity is a property of Pods that allows them to choose nodes, taints and tolerations are the opposite. They allow the nodes to control which Pods should or should not be scheduled on them. A *taint* is a characteristic of the node, and when it is present, it prevents Pods from scheduling onto the node unless the Pod has toleration for the taint. In that sense, taints and tolerations can be considered an *opt-in* to allow scheduling on nodes that by default are not available for scheduling, whereas affinity rules are an *opt-out* by explicitly selecting on which nodes to run and thus exclude all the nonselected nodes.

A taint is added to a node by using `kubectl`: `kubectl taint nodes control-plane-node node-role.kubernetes.io/control-plane="true":NoSchedule`, which has the effect shown in Example 6-7. A matching toleration is added to a Pod as shown in Example 6-8. Notice that the values for `key` and `effect` in the `taints` section of Example 6-7 and the `tolerations` section in Example 6-8 are the same.

Example 6-7. Tainted node

```
apiVersion: v1
kind: Node
metadata:
  name: control-plane-node
spec:
  taints:                                    ❶
  - effect: NoSchedule
    key: node-role.kubernetes.io/control-plane
    value: "true"
```

❶ Mark this node as unschedulable except when a Pod tolerates this taint.

Example 6-8. Pod tolerating node taints

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
  tolerations:
  - key: node-role.kubernetes.io/control-plane ❶
    operator: Exists
    effect: NoSchedule                         ❷
```

❶ Tolerate (i.e., consider for scheduling) nodes, which have a taint with key `node-role.kubernetes.io/control-plane`. On production clusters, this taint is set on the control plane node to prevent scheduling of Pods on this node. A toleration like this allows this Pod to be installed on the control plane node nevertheless.

❷ Tolerate only when the taint specifies a `NoSchedule` effect. This field can be empty here, in which case the toleration applies to every effect.

There are hard taints that prevent scheduling on a node (`effect=NoSchedule`), soft taints that try to avoid scheduling on a node (`effect=PreferNoSchedule`), and taints that can evict already-running Pods from a node (`effect=NoExecute`).

Taints and tolerations allow for complex use cases like having dedicated nodes for an exclusive set of Pods, or force eviction of Pods from problematic nodes by tainting those nodes.

You can influence the placement based on the application's high availability and performance needs, but try not to limit the scheduler too much and back yourself into a corner where no more Pods can be scheduled and there are too many stranded resources. For example, if your containers' resource requirements are too coarse-grained, or nodes are too small, you may end up with stranded resources in nodes that are not utilized.

In Figure 6-2, we can see node A has 4 GB of memory that cannot be utilized as there is no CPU left to place other containers. Creating containers with smaller resource requirements may help improve this situation. Another solution is to use the Kubernetes *descheduler*, which helps defragment nodes and improve their utilization.

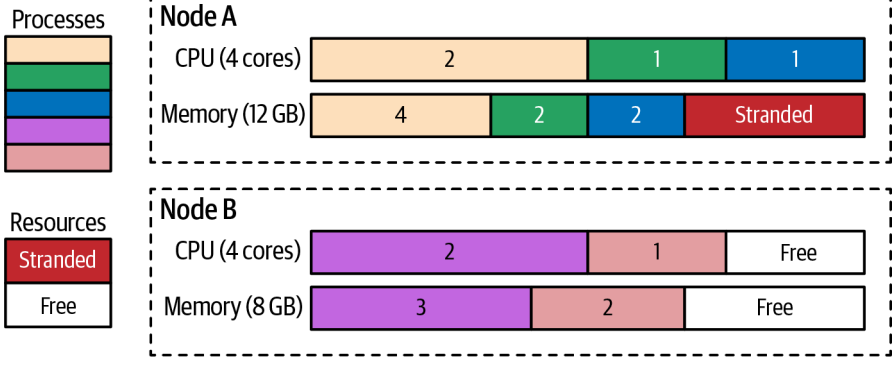

Figure 6-2. Processes scheduled to nodes and stranded resources

Once a Pod is assigned to a node, the job of the scheduler is done, and it does not change the placement of the Pod unless the Pod is deleted and recreated without a node assignment. As you have seen, with time, this can lead to resource fragmentation and poor utilization of cluster resources. Another potential issue is that the scheduler decisions are based on its cluster view at the point in time when a new Pod is scheduled. If a cluster is dynamic and the resource profile of the nodes changes or new nodes are added, the scheduler will not rectify its previous Pod placements. Apart from changing the node capacity, you may also alter the labels on the nodes that affect placement, but past placements are not rectified.

All of these scenarios can be addressed by the descheduler. The Kubernetes descheduler is an optional feature that is typically run as a Job whenever a cluster administrator decides it is a good time to tidy up and defragment a cluster by rescheduling the Pods. The descheduler comes with some predefined policies that can be enabled and tuned or disabled.

Regardless of the policy used, the descheduler avoids evicting the following:

- Node- or cluster-critical Pods
- Pods not managed by a ReplicaSet, Deployment, or Job, as these Pods cannot be recreated
- Pods managed by a DaemonSet
- Pods that have local storage
- Pods with PodDisruptionBudget, where eviction would violate its rules
- Pods that have a non-nil `DeletionTimestamp` field set
- Deschedule Pod itself (achieved by marking itself as a critical Pod)

Of course, all evictions respect Pods' QoS levels by choosing *Best-Efforts* Pods first, then *Burstable* Pods, and finally *Guaranteed* Pods as candidates for eviction. See Chapter 2, "Predictable Demands", for a detailed explanation of these QoS levels.

Discussion

Placement is the art of assigning Pods to nodes. You want to have as minimal intervention as possible, as the combination of multiple configurations can be hard to predict. In simpler scenarios, scheduling Pods based on resource constraints should be sufficient. If you follow the guidelines from Chapter 2, "Predictable Demands", and declare all the resource needs of a container, the scheduler will do its job and place the Pod on the most feasible node possible.

However, in more realistic scenarios, you may want to schedule Pods to specific nodes according to other constraints such as data locality, Pod colocality, application

high availability, and efficient cluster resource utilization. In these cases, there are multiple ways to steer the scheduler toward the desired deployment topology.

Figure 6-3 shows one approach to thinking and making sense of the different scheduling techniques in Kubernetes.

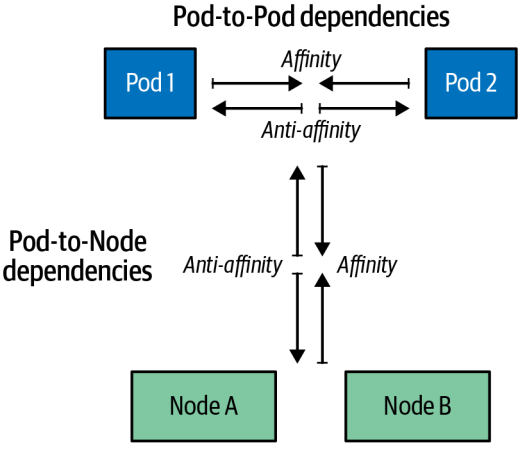

Figure 6-3. Pod-to-Pod and Pod-to-Node and dependencies

Start by identifying the forces and dependencies between the Pod and the nodes (for example, based on dedicated hardware capabilities or efficient resource utilization). Use the following node affinity techniques to direct the Pod to the desired nodes, or use anti-affinity techniques to steer the Pod away from the undesired nodes:

nodeName
: This field provides the simplest form of hard wiring a Pod to a node. This field should ideally be populated by the scheduler, which is driven by policies rather than manual node assignment. Assigning a Pod to a node through this approach prevents the scheduling of the Pod to any other node. If the named node has no capacity, or the node doesn't exist, the Pod will never run. This throws us back into the pre-Kubernetes era, when we explicitly needed to specify the nodes to run our applications. Setting this field manually is not a Kubernetes best practice and should be used only as an exception.

nodeSelector
: A node selector is a label map. For the Pod to be eligible to run on a node, the Pod must have the indicated key-value pairs as the label on the node. Having put some meaningful labels on the Pod and the node (which you should do anyway), a node selector is one of the simplest recommended mechanisms for controlling the scheduler choices.

Node affinity
: This rule improves the manual node assignment approaches and allows a Pod to express dependency toward nodes using logical operators and constraints that provides fine-grained control. It also offers soft and hard scheduling requirements that control the strictness of node affinity constraints.

Taints and tolerations
: Taints and tolerations allow the node to control which Pods should or should not be scheduled on them without modifying existing Pods. By default, Pods that don't have tolerations for the node taint will be rejected or evicted from the node. Another advantage of taints and tolerations is that if you expand the Kubernetes cluster by adding new nodes with new labels, you don't need to add the new labels on all Pods but only on those that should be placed on the new nodes.

Once the desired correlation between a Pod and the nodes is expressed in Kubernetes terms, identify the dependencies between different Pods. Use Pod affinity techniques for Pod colocation for tightly coupled applications, and use Pod anti-affinity techniques to spread Pods on nodes and avoid a single point of failure:

Pod affinity and anti-affinity
: These rules allow scheduling based on Pods' dependencies on other Pods rather than nodes. Affinity rules help for colocating tightly coupled application stacks composed of multiple Pods on the same topology for low-latency and data locality requirements. The anti-affinity rule, on the other hand, can spread Pods across your cluster among failure domains to avoid a single point of failure, or prevent resource-intensive Pods from competing for resources by avoiding placing them on the same node.

Topology spread constraints
: To use these features, platform administrators have to label nodes and provide topology information such as regions, zones, or other user-defined domains. Then, a workload author creating the Pod configurations must be aware of the underlying cluster topology and specify the topology spread constraints. You can also specify multiple topology spread constraints, but all of them must be satisfied for a Pod to be placed. You must ensure that they do not conflict with one another. You can also combine this feature with NodeAffinity and NodeSelector to filter nodes where evenness should be applied. In that case, be sure to understand the difference: multiple topology spread constraints are about calculating the result set independently and producing an AND-joined result, while combining it with NodeAffinity and NodeSelector, on the other hand, filters results of node constraints.

In some scenarios, all of these scheduling configurations might not be flexible enough to express bespoke scheduling requirements. In that case, you may have to customize and tune the scheduler configuration or even provide a custom scheduler implementation that can understand your custom needs:

Scheduler tuning
: The default scheduler is responsible for the placement of new Pods onto nodes within the cluster, and it does it well. However, it is possible to alter one or more stages in the filtering and prioritization phases. This mechanism with extension points and plugins is specifically designed to allow small alterations without the need for a completely new scheduler implementation.

Custom scheduler
: If none of the preceding approaches is good enough, or if you have complex scheduling requirements, you can also write your own custom scheduler. A custom scheduler can run instead of, or alongside, the standard Kubernetes scheduler. A hybrid approach is to have a "scheduler extender" process that the standard Kubernetes scheduler calls out to as a final pass when making scheduling decisions. This way, you don't have to implement a full scheduler but only provide HTTP APIs to filter and prioritize nodes. The advantage of having your scheduler is that you can consider factors outside of the Kubernetes cluster like hardware cost, network latency, and better utilization while assigning Pods to nodes. You can also use multiple custom schedulers alongside the default scheduler and configure which scheduler to use for each Pod. Each scheduler could have a different set of policies dedicated to a subset of the Pods.

To sum up, there are lots of ways to control the Pod placement, and choosing the right approach or combining multiple approaches can be overwhelming. The takeaway from this chapter is this: size and declare container resource profiles, and label Pods and nodes for the best resource-consumption-driven scheduling results. If that doesn't deliver the desired scheduling outcome, start with small and iterative changes. Strive for a minimal policy-based influence on the Kubernetes scheduler to express node dependencies and then inter-Pod dependencies.

More Information

- Automated Placement Example
- Assigning Pods to Nodes
- Scheduler Configuration
- Pod Topology Spread Constraints
- Configure Multiple Schedulers
- Descheduler for Kubernetes

- Disruptions
- Guaranteed Scheduling for Critical Add-On Pods
- Keep Your Kubernetes Cluster Balanced: The Secret to High Availability
- Advanced Kubernetes Pod to Node Scheduling

PART II
Behavioral Patterns

The patterns in this category are focused on the communications and interactions between the Pods and the managing platform. Depending on the type of managing controller used, a Pod may run until completion or be scheduled to run periodically. It can run as a daemon or ensure uniqueness guarantees to its replicas. There are different ways to run a Pod on Kubernetes, and picking the right Pod-management primitives requires understanding their behavior. In the following chapters, we explore the patterns:

- Chapter 7, "Batch Job", describes how to isolate an atomic unit of work and run it until completion.
- Chapter 8, "Periodic Job", allows the execution of a unit of work to be triggered by a temporal event.
- Chapter 9, "Daemon Service", allows you to run infrastructure-focused Pods on specific nodes, before application Pods are placed.
- Chapter 10, "Singleton Service", ensures that only one instance of a service is active at a time and still remains highly available.
- Chapter 11, "Stateless Service", describes the building blocks used for managing identical application instances.
- Chapter 12, "Stateful Service", is all about how to create and manage distributed stateful applications with Kubernetes.

- Chapter 13, "Service Discovery", explains how client services can discover and consume the instances of providing services.
- Chapter 14, "Self Awareness", describes mechanisms for introspection and metadata injection into applications.

CHAPTER 7

Batch Job

The *Batch Job* pattern is suited for managing isolated atomic units of work. It is based on the Job resource, which runs short-lived Pods reliably until completion on a distributed environment.

Problem

The main primitive in Kubernetes for managing and running containers is the Pod. There are different ways of creating Pods with varying characteristics:

Bare Pod
: It is possible to create a Pod manually to run containers. However, when the node such a Pod is running on fails, the Pod is not restarted. Running Pods this way is discouraged except for development or testing purposes. This mechanism is also known as *unmanaged* or *naked Pods*.

ReplicaSet
: This controller is used for creating and managing the lifecycle of Pods expected to run continuously (e.g., to run a web server container). It maintains a stable set of replica Pods running at any given time and guarantees the availability of a specified number of identical Pods. ReplicaSets are described in detail in Chapter 11, "Stateless Service".

DaemonSet
: This controller runs a single Pod on every node and is used for managing platform capabilities such as monitoring, log aggregation, storage containers, and others. See Chapter 9, "Daemon Service", for a more detailed discussion.

A common aspect of these Pods is that they represent long-running processes that are not meant to stop after a certain time. However, in some cases there is a need to perform a predefined finite unit of work reliably and then shut down the container. For this task, Kubernetes provides the Job resource.

Solution

A Kubernetes Job is similar to a ReplicaSet as it creates one or more Pods and ensures they run successfully. However, the difference is that, once the expected number of Pods terminate successfully, the Job is considered complete, and no additional Pods are started. A Job definition looks like Example 7-1.

Example 7-1. A Job specification

```
apiVersion: batch/v1
kind: Job
metadata:
  name: random-generator
spec:
  completions: 5                ❶
  parallelism: 2                ❷
  ttlSecondsAfterFinished: 300  ❸
  template:
    metadata:
      name: random-generator
    spec:
      restartPolicy: OnFailure  ❹
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        command: [ "java", "RandomRunner", "/numbers.txt", "10000" ]
```

❶ Job should run five Pods to completion, which all must succeed.

❷ Two Pods can run in parallel.

❸ Keep Pods for five minutes (300 seconds) before garbage-collecting them.

❹ Specifying the `restartPolicy` is mandatory for a Job. The possible values are `OnFailure` or `Never`.

One crucial difference between the Job and the ReplicaSet definition is the `.spec.template.spec.restartPolicy`. The default value for a ReplicaSet is `Always`, which makes sense for long-running processes that must always be kept running. The value `Always` is not allowed for a Job, and the only possible options are `OnFailure` or `Never`.

So why bother creating a Job to run a Pod only once instead of using bare Pods? Using Jobs provides many reliability and scalability benefits that make them the preferred option:

- A Job is not an ephemeral in-memory task but a persisted one that survives cluster restarts.
- When a Job is completed, it is not deleted but is kept for tracking purposes. The Pods that are created as part of the Job are also not deleted but are available for examination (e.g., to check the container logs). This is also true for bare Pods but only for `restartPolicy: OnFailure`. You can still remove the Pods of a Job after a certain time by specifying `.spec.ttlSecondsAfterFinished`.
- A Job may need to be performed multiple times. Using the `.spec.completions` field, it is possible to specify how many times a Pod should complete successfully before the Job itself is done.
- When a Job has to be completed multiple times, it can also be scaled and executed by starting multiple Pods at the same time. That can be done by specifying the `.spec.parallelism` field.
- A Job can be suspended by setting the field `.spec.suspend` to `true`. In this case, all active Pods are deleted and restarted if the Job is resumed (i.e., `.spec.suspend` set to `false` by the user).
- If the node fails or when the Pod is evicted for some reason while still running, the scheduler places the Pod on a new healthy node and reruns it. Bare Pods would remain in a failed state as existing Pods are never moved to other nodes.

All of this makes the Job primitive attractive for scenarios requiring some guarantees for the completion of a unit of work.

The following two fields play major roles in the behavior of a Job:

`.spec.completions`
: Specifies how many Pods should run to complete a Job.

`.spec.parallelism`
: Specifies how many Pod replicas could run in parallel. Setting a high number does not guarantee a high level of parallelism, and the actual number of Pods may still be fewer (and in some corner cases, more) than the desired number (e.g., because of throttling, resource quotas, not enough completions left, and other reasons). Setting this field to 0 effectively pauses the Job.

Figure 7-1 shows how the Job defined in Example 7-1 with a completion count of 5 and a parallelism of 2 is processed.

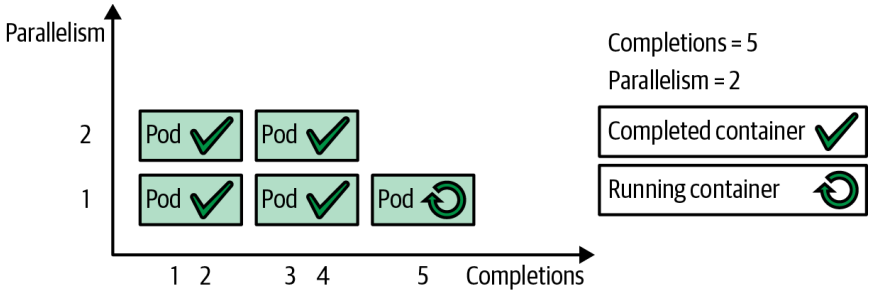

Figure 7-1. Parallel Batch Job with a fixed completion count

Based on these two parameters, there are the following types of Jobs:

Single Pod Jobs
: This type is selected when you leave out both `.spec.completions` and `.spec.parallelism` or set them to their default values of 1. Such a Job starts only one Pod and is completed as soon as the single Pod terminates successfully (with exit code 0).

Fixed completion count Jobs
: For a fixed completion count Job, you should set `.spec.completions` to the number of completions needed. You can set `.spec.parallelism`, or leave it unset and it will default to 1. Such a Job is considered completed after the `.spec.completions` number of Pods has completed successfully. Example 7-1 shows this mode in action and is the best choice when we know the number of work items in advance and the processing cost of a single work item justifies the use of a dedicated Pod.

Work queue Jobs
: For a work queue Job, you need to leave `.spec.completions` unset, and set `.spec.parallelism` to a number greater than one. A work queue Job is considered completed when at least one Pod has terminated successfully and all other Pods have terminated too. This setup requires the Pods to coordinate among themselves and determine what each one is working on so that they can finish in a coordinated fashion. For example, when a fixed but unknown number of work items is stored in a queue, parallel Pods can pick these up one by one to work on them. The first Pod that detects that the queue is empty and exits with success indicates the completion of the Job. The Job controller waits for all other Pods to terminate too. Since one Pod processes multiple work items, this Job type is an excellent choice for granular work items—when the overhead for one Pod per work item is not justified.

Indexed Jobs

Similar to *Work queue Jobs*, you can distribute work items to individual Jobs without needing an external work queue. When using a fixed completion count and setting the completion mode `.spec.completionMode` to `Indexed`, every Pod of the Job gets an associated index ranging from 0 to `.spec.completions - 1`. The assigned index is available to the containers through the Pod annotation `batch.kubernetes.io/job-completion-index` (see Chapter 14, "Self Awareness", to learn how this annotation can be accessed from your code) or directly via the environment variable `JOB_COMPLETION_INDEX` that is set to the index associated with this Pod. With this index at hand, the application can pick the associated work item without any external synchronization. Example 7-2 shows a Job that processes the lines of a single file individually by separate Pods. A more realistic example would be an indexed Job used for video processing, where parallel Pods are processing a certain frame range calculated from the index.

Example 7-2. An indexed Job selecting its work items based on a job index

```
apiVersion: batch/v1
kind: Job
metadata:
  name: file-split
spec:
  completionMode: Indexed      ❶
  completions: 5               ❷
  parallelism: 5
  template:
    metadata:
      name: file-split
    spec:
      containers:
      - image: alpine
        name: split
        command:               ❸
        - "sh"
        - "-c"
        - |
          start=$(expr $JOB_COMPLETION_INDEX \* 10000)      ❹
          end=$(expr $JOB_COMPLETION_INDEX \* 10000 + 10000)
          awk "NR>=$start && NR<$end" /logs/random.log \    ❺
              > /logs/random-$JOB_COMPLETION_INDEX.txt
        volumeMounts:
        - mountPath: /logs     ❻
          name: log-volume
      restartPolicy: OnFailure
```

❶ Enable an indexed completion mode.

❷ Run five Pods in parallel to completion.

❸ Execute a shell script that prints out a range of lines from a given file */logs/random.log*. This file is expected to have 50,000 lines of data.

❹ Calculate start and end line numbers.

❺ Use `awk` to print out a range of line numbers (`NR` is the `awk`-internal line number when iterating over the file).

❻ Mount the input data from an external volume. The volume is not shown here; you can find the full working definition in the example repository.

Partitioning the Work

As you have seen, we have multiple options for processing many work items by fewer worker Pods. While *Work queue Jobs* can operate on an unknown but finite set of work items, they need support from an external system that provides the work items. In that case, the external system has already divided the work into appropriately sized work items, so the worker Pods have to process those and stop when there is nothing left to do. The alternative is to use *Indexed Jobs*, which do not rely on an external work queue but have to split up the work on their own so that each Pod can separately work on a portion of the overall task. Each Pod needs to know its own identity (provided by the environment variable `JOB_COMPLETION_INDEX`), the total number of workers, and maybe the overall size of the work (like the size of a movie file to process). Unfortunately, the Job's application code cannot discover the total number of workers (i.e., the value specified in `.spec.completions`) for an Indexed Job. Therefore, something like a `JOB_COMPLETION_TOTAL` environment variable would be helpful to partition the work dynamically, but this is not supported as of 2023. However, there are two solutions to overcome this:

- Hardcode the knowledge of the total number of Pods working on a Job into the application code. While this might work for simple examples like Example 7-2, it's generally an imperfect solution as it couples the code in your container to the Kubernetes declaration. That is, if you want to change the number of completions in your Job definition, you would also have to create a new container image for your Job logic with an updated value.
- To access the value of `.spec.completions` in your application code, you can copy it to an environment variable or pass it as an argument to the container command in the Job's template specification. But if you plan to change the number of completions, you will need to update two places in the Job declaration.

There has been some discussion within the Kubernetes community about whether Kubernetes should provide the value of the `.spec.completions` field as an environment variable by default. The main concern with this approach is that environment variables cannot be modified at runtime, which could complicate support for resizable Jobs in the future. As a result, a `JOB_COMPLETION_TOTAL` environment variable is not provided by Kubernetes as of version 1.26.

If you have an unlimited stream of work items to process, other controllers like ReplicaSet are the better choice for managing the Pods processing these work items.

Discussion

The Job abstraction is a pretty basic but also fundamental primitive that other primitives such as CronJobs are based on. Jobs help turn isolated work units into a reliable and scalable unit of execution. However, a Job doesn't dictate how you should map individually processable work items into Jobs or Pods. That is something you have to determine after considering the pros and cons of each option:

One Job per work item
: This option has the overhead of creating Kubernetes Jobs and also means the platform has to manage a large number of Jobs that are consuming resources. This option is useful when each work item is a complex task that has to be recorded, tracked, or scaled independently.

One Job for all work items
: This option is right for a large number of work items that do not have to be independently tracked and managed by the platform. In this scenario, the work items have to be managed from within the application via a batch framework.

The Job primitive provides only the very minimum basics for scheduling work items. Any complex implementation has to combine the Job primitive with a batch application framework (e.g., in the Java ecosystem, we have Spring Batch and JBeret as standard implementations) to achieve the desired outcome.

Not all services must run all the time. Some services must run on demand, some at a specific time, and some periodically. Using Jobs can run Pods only when needed and only for the duration of the task execution. Jobs are scheduled on nodes that have the required capacity, satisfy Pod placement policies, and take into account other container dependency considerations. Using Jobs for short-lived tasks rather than using long-running abstractions (such as ReplicaSet) saves resources for other workloads on the platform. All of that makes Jobs a unique primitive, and Kubernetes a platform supporting diverse workloads.

More Information

- Batch Job Example
- Jobs
- Parallel Processing Using Expansions
- Coarse Parallel Processing Using a Work Queue
- Fine Parallel Processing Using a Work Queue
- Indexed Job for Parallel Processing with Static Work Assignment
- Spring Batch on Kubernetes: Efficient Batch Processing at Scale
- JBeret Introduction

CHAPTER 8
Periodic Job

The *Periodic Job* pattern extends the *Batch Job* pattern by adding a time dimension and allowing the execution of a unit of work to be triggered by a temporal event.

Problem

In the world of distributed systems and microservices, there is a clear tendency toward real-time and event-driven application interactions using HTTP and lightweight messaging. However, regardless of the latest trends in software development, job scheduling has a long history, and it is still relevant. Periodic jobs are commonly used for automating system maintenance or administrative tasks. They are also relevant to business applications requiring specific tasks to be performed periodically. Typical examples here are business-to-business integration through file transfer, application integration through database polling, sending newsletter emails, and cleaning up and archiving old files.

The traditional way of handling periodic jobs for system maintenance purposes has been to use specialized scheduling software or cron. However, specialized software can be expensive for simple use cases, and cron jobs running on a single server are difficult to maintain and represent a single point of failure. That is why, very often, developers tend to implement solutions that can handle both the scheduling aspect and the business logic that needs to be performed. For example, in the Java world, libraries such as Quartz, Spring Batch, and custom implementations with the `ScheduledThreadPoolExecutor` class can run temporal tasks. But similar to cron, the main difficulty with this approach is making the scheduling capability resilient and highly available, which leads to high resource consumption. Also, with this approach, the time-based job scheduler is part of the application, and to make the scheduler highly available, the whole application must be highly available. Typically, that involves running multiple instances of the application and at the same time

ensuring that only a single instance is active and schedules jobs—which involves leader election and other distributed systems challenges.

In the end, a simple service that has to copy a few files once a day may end up requiring multiple nodes, a distributed leader election mechanism, and more. Kubernetes CronJob implementation solves all that by allowing scheduling of Job resources using the well-known cron format and letting developers focus only on implementing the work to be performed rather than the temporal scheduling aspect.

Solution

In Chapter 7, "Batch Job", we saw the use cases and the capabilities of Kubernetes Jobs. All of that applies to this chapter as well since the CronJob primitive builds on top of a Job. A CronJob instance is similar to one line of a Unix crontab (cron table) and manages the temporal aspects of a Job. It allows the execution of a Job periodically at a specified point in time. See Example 8-1 for a sample definition.

Example 8-1. A CronJob resource

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: random-generator
spec:
  schedule: "*/3 * * * *"  ❶
  jobTemplate:
    spec:
      template:            ❷
        spec:
          containers:
          - image: k8spatterns/random-generator:1.0
            name: random-generator
            command: [ "java", "RandomRunner", "/numbers.txt", "10000" ]
          restartPolicy: OnFailure
```

❶ Cron specification for running every three minutes.

❷ Job template that uses the same specification as a regular Job.

Apart from the Job spec, a CronJob has additional fields to define its temporal aspects:

`.spec.schedule`
: Crontab entry for specifying the Job's schedule (e.g., `0 * * * *` for running every hour). You can also use shortcuts like `@daily` or `@hourly`. Please refer to the CronJob documentation for all available options.

`.spec.startingDeadlineSeconds`
: Deadline (in seconds) for starting the Job if it misses its scheduled time. In some use cases, a task is valid only if it executed within a certain timeframe, and it is useless when executed late. For example, if a Job is not executed in the desired time because of a lack of compute resources or other missing dependencies, it might be better to skip an execution because the data it is supposed to process is already obsolete. Don't use a deadline fewer than 10 seconds since Kubernetes will check the Job status only every 10 seconds.

`.spec.concurrencyPolicy`
: Specifies how to manage concurrent executions of Jobs created by the same CronJob. The default behavior `Allow` creates new Job instances even if the previous Jobs have not completed yet. If that is not the desired behavior, it is possible to skip the next run if the current one has not completed yet with `Forbid` or to cancel the currently running Job and start a new one with `Replace`.

`.spec.suspend`
: Field suspending all subsequent executions without affecting already-started executions. Note that this is different from a Job's `.spec.suspend` as the start of new Jobs will be suspended, not the Jobs themselves.

`.spec.successfulJobsHistoryLimit` *and* `.spec.failedJobsHistoryLimit`
: Fields specifying how many completed and failed Jobs should be kept for auditing purposes.

CronJob is a very specialized primitive, and it applies only when a unit of work has a temporal dimension. Even if CronJob is not a general-purpose primitive, it is an excellent example of how Kubernetes capabilities build on top of one another and support noncloud native use cases as well.

Discussion

As you can see, a CronJob is a pretty simple primitive that adds clustered, cron-like behavior to the existing Job definition. But when it is combined with other primitives such as Pods, container resource isolation, and other Kubernetes features such as those described in Chapter 6, "Automated Placement", or Chapter 4, "Health Probe", it ends up being a very powerful job-scheduling system. This enables developers to focus solely on the problem domain and implement a containerized application that is responsible only for the business logic to be performed. The scheduling is performed outside the application, as part of the platform with all of its added benefits, such as high availability, resiliency, capacity, and policy-driven Pod placement. Of course, similar to the Job implementation, when implementing a CronJob container, your application has to consider all corner and failure cases of duplicate runs, no runs, parallel runs, or cancellations.

More Information

- Periodic Job Example
- CronJob
- Cron
- Crontab Specification
- Cron Expression Generator

CHAPTER 9

Daemon Service

The *Daemon Service* pattern allows you to place and run prioritized, infrastructure-focused Pods on targeted nodes. It is used primarily by administrators to run node-specific Pods to enhance the Kubernetes platform capabilities.

Problem

The concept of a daemon in software systems exists at many levels. At an operating system level, a *daemon* is a long-running, self-recovering computer program that runs as a background process. In Unix, the names of daemons end in *d*, such as httpd, named, and sshd. In other operating systems, alternative terms such as *services-started tasks* and *ghost jobs* are used.

Regardless of what these programs are called, the common characteristics among them are that they run as processes and usually do not interact with the monitor, keyboard, and mouse and are launched at system boot time. A similar concept also exists at the application level. For example, in the Java Virtual Machine, daemon threads run in the background and provide supporting services to the user threads. These daemon threads have a low priority, run in the background without a say in the life of the application, and perform tasks such as garbage collection or finalization.

Similarly, Kubernetes also has the concept of a DaemonSet. Considering that Kubernetes is a distributed platform spread across multiple nodes and with the primary goal of managing application Pods, a DaemonSet is represented by Pods that run on the cluster nodes and provide some background capabilities for the rest of the cluster.

Solution

ReplicaSet and its predecessor ReplicationController are control structures responsible for making sure a specific number of Pods are running. These controllers constantly monitor the list of running Pods and make sure the actual number of Pods always matches the desired number. In that regard, a DaemonSet is a similar construct and is responsible for ensuring that a certain number of Pods are always running. The difference is that the first two run a specific number of Pods, usually driven by the application requirements of high availability and user load, irrespective of the node count.

On the other hand, a DaemonSet is not driven by consumer load in deciding how many Pod instances to run and where to run. Its main purpose is to keep running a single Pod on every node or specific nodes. Let's see such a DaemonSet definition next in Example 9-1.

Example 9-1. DaemonSet resource

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: random-refresher
spec:
  selector:
    matchLabels:
      app: random-refresher
  template:
    metadata:
      labels:
        app: random-refresher
    spec:
      nodeSelector:                ❶
        feature: hw-rng
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        command: [ "java", "RandomRunner", "/numbers.txt", "10000", "30" ]
        volumeMounts:              ❷
        - mountPath: /host_dev
          name: devices
      volumes:
      - name: devices
        hostPath:                  ❸
          path: /dev
```

❶ Use only nodes with the label `feature` set to value `hw-rng`.

❷ DaemonSets often mount a portion of a node's filesystem to perform maintenance actions.

❸ `hostPath` for accessing the node directories directly.

Given this behavior, the primary candidates for a DaemonSet are usually infrastructure-related processes, such as cluster storage providers, log collectors, metric exporters, and even kube-proxy, that perform cluster-wide operations. There are many differences in how DaemonSet and ReplicaSet are managed, but the main ones are the following:

- By default, a DaemonSet places one Pod instance on every node. That can be controlled and limited to a subset of nodes by using the `nodeSelector` or `affinity` fields.
- A Pod created by a DaemonSet already has `nodeName` specified. As a result, the DaemonSet doesn't require the existence of the Kubernetes scheduler to run containers. That also allows you to use a DaemonSet for running and managing the Kubernetes components.
- Pods created by a DaemonSet can run before the scheduler has started, which allows them to run before any other Pod is placed on a node.
- Since the scheduler is not used, the `unschedulable` field of a node is not respected by the DaemonSet controller.
- Pods created by a DaemonSet can have a `RestartPolicy` only set to `Always` or left unspecified, which defaults to `Always`. This is to ensure that when a liveness probe fails, the container will be killed and always restarted.
- Pods managed by a DaemonSet are supposed to run only on targeted nodes and, as a result, are treated with higher priority by many controllers. For example, the descheduler will avoid evicting such Pods, the cluster autoscaler will manage them separately, etc.

The main use case for DaemonSets is to run system-critical Pods on certain nodes in the cluster. The DaemonSet controller ensures that all eligible nodes run a copy of a Pod by assigning the Pod directly to the node by setting the `nodeName` field of the Pod specification. This allows DaemonSet Pods to be scheduled even before the default scheduler starts and keeps it immune to any scheduler customizations configured by the user. This approach works as long as there are enough resources on the nodes and it is done before other Pods are placed. When a node does not have enough resources, the DaemonSet controller cannot create a Pod for the node, and it cannot do anything such as preemption to release resources on the nodes. This duplication of scheduling logic in the DaemonSet controller and the scheduler creates maintenance challenges. The DaemonSet implementation also does not benefit from

new scheduler features such as affinity, anti-affinity, and preemption. As a result, with Kubernetes v1.17 and newer versions, DaemonSet uses the default scheduler for scheduling by setting the `nodeAffinity` field instead of the `nodeName` field to the DaemonSet Pods. This change makes the default scheduler a mandatory dependency for running DaemonSets, but at the same time it brings taints, tolerations, Pod priority, and preemption to DaemonSets and improves the overall experience of running DaemonSet Pods on the desired nodes even when there is resource starvation.

Typically, a DaemonSet creates a single Pod on every node or subset of nodes. Given that, there are several ways to reach Pods managed by DaemonSets:

Service
: Create a Service with the same Pod selector as a DaemonSet, and use the Service to reach a daemon Pod load-balanced to a random node.

DNS
: Create a headless Service with the same Pod selector as a DaemonSet that can be used to retrieve multiple A records from DNS containing all Pod IPs and ports.

Node IP with `hostPort`
: Pods in the DaemonSet can specify a `hostPort` and become reachable via the node IP addresses and the specified port. Since the combination of node IP and `hostPort` and `protocol` must be unique, the number of places where a Pod can be scheduled is limited.

Also, the application in the DaemonSets Pods can push data to a well-known location or service that's external to the Pod. No consumer needs to reach the DaemonSets Pods in this case.

Static Pods

Another way to run containers similar to the way a DaemonSet does is through the *static Pods* mechanism. The Kubelet, in addition to talking to the Kubernetes API Server and getting Pod manifests, can get the resource definitions from a local directory. Pods defined this way are managed by the Kubelet only and run on one node only. The API service is not observing these Pods, and no controller and no health checks are performed on them. The Kubelet watches such Pods and restarts them when they crash. Similarly, the Kubelet also periodically scans the configured directory for Pod definition changes and adds or removes Pods accordingly.

Static Pods can be used to spin off a containerized version of Kubernetes system processes or other containers. However, DaemonSets are better integrated with the rest of the platform and are recommended over static Pods.

Discussion

There are other ways to run daemon processes on every node, but they all have limitations. Static Pods are managed by the Kubelet but cannot be managed through Kubernetes APIs. Bare Pods (Pods without a controller) cannot survive if they are accidentally deleted or terminated, nor can they survive a node failure or disruptive node maintenance. Init scripts such as upstartd or systemd require different toolchains for monitoring and management and cannot benefit from the Kubernetes tools used for application workloads. All that makes Kubernetes and DaemonSet an attractive option for running daemon processes too.

In this book, we describe patterns and Kubernetes features primarily used by developers rather than platform administrators. A DaemonSet is somewhere in the middle, inclining more toward the administrator toolbox, but we include it here because it also has relevance to application developers. DaemonSets and CronJobs are also perfect examples of how Kubernetes turns single-node concepts such as crontab and daemon scripts into multinode clustered primitives for managing distributed systems. These are new distributed concepts developers must also be familiar with.

More Information

- Daemon Service Example
- DaemonSet
- Perform a Rolling Update on a DaemonSet
- DaemonSets and Jobs
- Create Static Pods