KUB 12: Базовые ресурсы в Kubernetes - часть 2

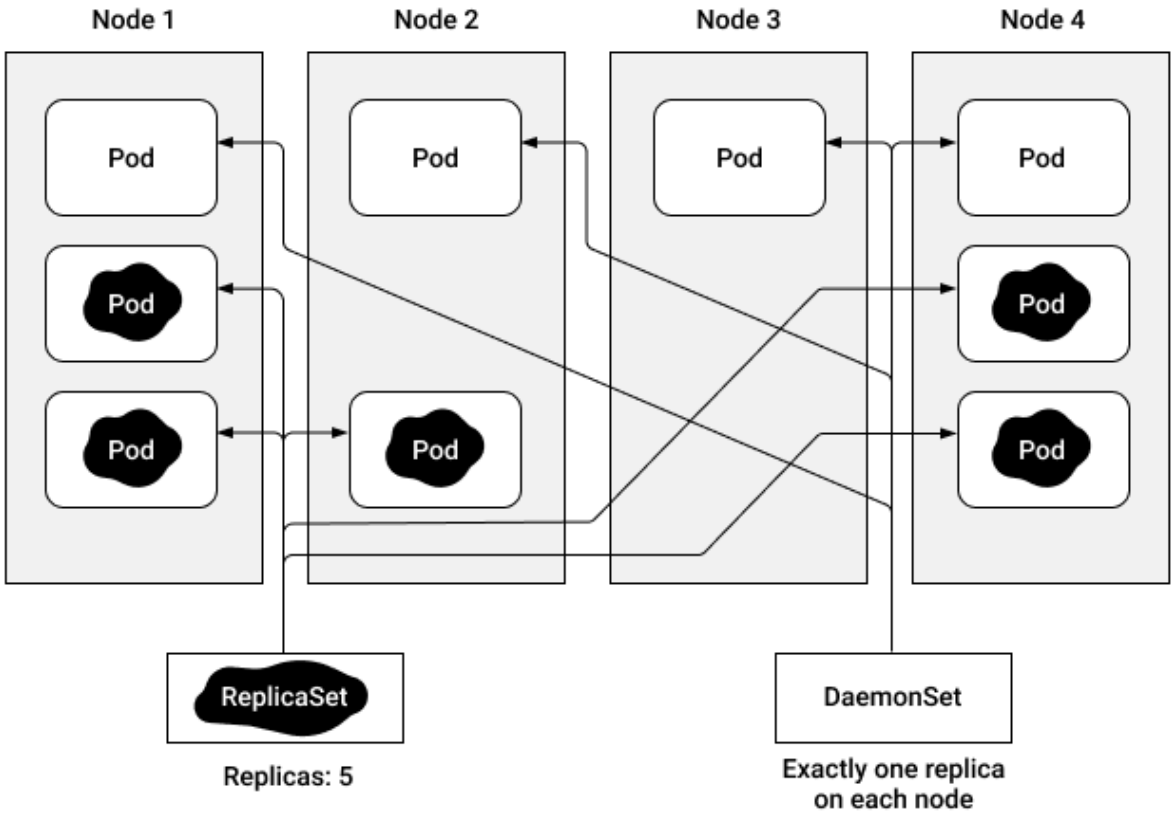
Описание:

DaemonSets

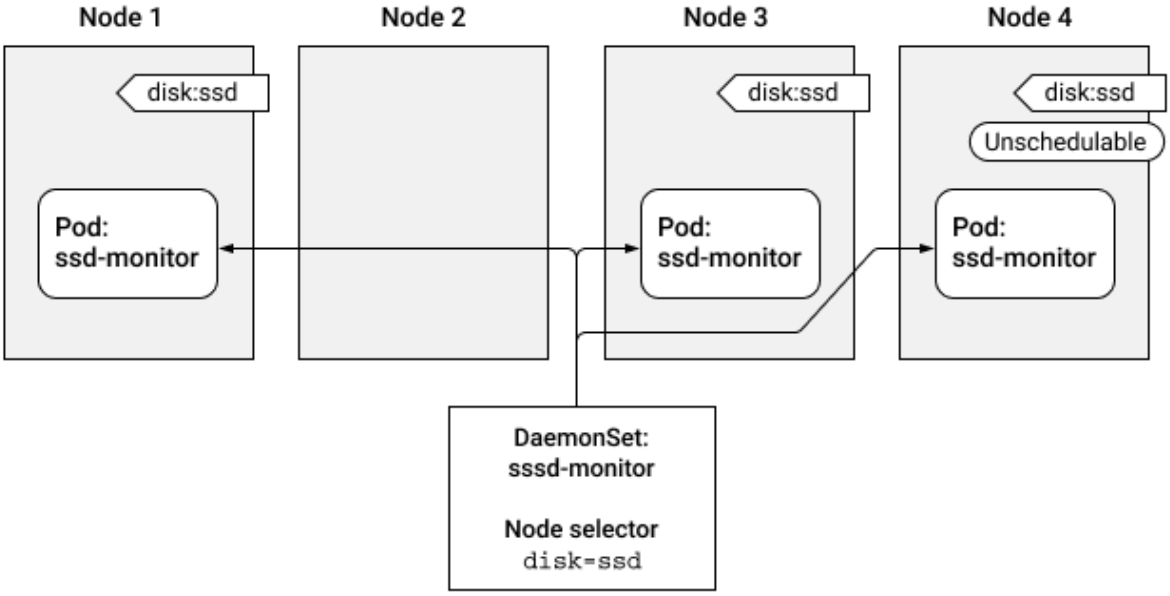
В прошлом задании мы познакомились с типом ресурсов Deployment, в котором можно понять, в каком количестве (replicas) pods должны быть запущены. Но, допустим, нам нужно запускать приложение на каждой ноде? К примеру, для сбора метрик с ноды, конфигурации параметров хоста или запуска какого-то сервиса, который привязан к определенным нодам, но мы не хотим думать, сколько реплик нам нужно. Есть нода — контейнер должен быть на ней запущен. Для этих задач существует тип ресурсов под названием DaemonSet.
DaemonSet — это ресурс, который позволяет именно то, что мы описали, — запускает требуемую нагрузку на всех нодах, которые подходят по nodeSelector, в единичном экземпляре и следит за тем, чтобы она была запущена всегда, даже если нода решила умереть.
В отличие от того же StatefulSet, с которым мы познакомимся чуть позже, система именования подов похожа на Deployment — префикс, полученный из имени DaemonSet + случайная строка (5 символов), но управление подами производит сам DaemonSet, а не ReplicaSet, как в случае с Deployment.
Немаловажной особенностью DaemonSet является то, что если у нас появляется новая нода, которая удовлетворяет nodeSelector, то на нем запустится под DaemonSet без необходимости вмешательства со стороны владельца кластера — Scheduler все сделает сам.
Схем обновления подов у DaemonSet 2:

- OnDelete — обновление будет произведено только при ручном удалении контейнеров. А до этого будет работать старый под.
- RollingUpdate — последовательное обновление всех подов.

Например, так работает kube-proxy. Один экземпляр должен быть запущен на каждом узле кластера.



При помощи nodeSelector можно заставить DaemonSet запускать pod только на определенных узлах кластера.



Пример описания DaemonSet:

```
apiVersion: apps/v1
kind: DaemonSet
```

```
metadata:
  name: ssd-monitor
spec:
  selector:
  matchLabels:
    app: ssd-monitor
  template:
    metadata:
      labels:
      app: ssd-monitor
    spec:
      nodeSelector:
        disk: ssd
      containers:
      - name: main
        image: infra/ssd-monitor
```

Job

Job — это ресурс для запуска пода, в нем работают контейнеры, цель которых — выполнение одной задачи и выход с успехом или неуспехом.
Как видно, в отличие от того же Deployment, у Job нет цели работать вечно. Наоборот, он выполняет определенную задачу и прекращает свое существование после ее завершения.
В отличие от других ресурсов, созданный из Job под в конце работы не удаляется или не перезапускается до работающего состояния (может, но об этом чуть позже), а остается в списке Pods, но со статусом Completed. Это означает, что задача успешно завершена. В таком состоянии под не использует никакие ресурсы, кроме места под логи работы.
Такой подход используется для того, чтобы иметь доступ к логам работы, что позволяет получить информацию о том, какой результат работы мы получили.
Как правило, Job используются в роли сервисных задач, которые нужно прогнать перед каким-то определенным обновлением. К примеру, применение миграций баз данных перед выкладкой кода.
Другой кейс — запуск фоновых задач, как, к примеру, разовые выгрузки данных.
Хоть мы и говорили, что Pods, созданные через Job, не перезапускаются, на самом деле они могут перезапускаться в случае, если код выхода контейнера отличается от 0 (из-за самого кода или, к примеру, из-за того, что контейнер был убит OOM Killer). Но этот механизм связан именно с Pods, а не с конкретно Job.
Пример запуска задачи:

```
apiVersion: batch/v1
kind: Job
metadata:
```

```
  name: test-job
spec:
  template:
    metadata:
      labels:
        app: ping-google
    spec:
      restartPolicy: OnFailure
      containers:
      - name: main
        image: alpine
        command: ["ping"]
        args: ["-c", "4", "8.8.8.8"]
```

Cron Jobs

Cronjob — это cron только внутри кластера. Job, который запускается по cron. Да, это просто, и описание интервалов совпадает с cron. Давайте рассмотрим подобный пример подробнее:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: batch-job-every-fifteen-minutes
spec:
  schedule: "0,15,30,45 * * * *"
  jobTemplate:
  spec:
    template:
      metadata:
        labels:
          app: periodic-batch-job
      spec:
        restartPolicy: OnFailure
        containers:
        - name: main
          image: job/batch-job
```

Взгляните на spec.schedule — это и есть основное отличие от обычных Job.
Однако cronjob не всегда могут запускаться в указанные интервалы из-за очереди планировщика. Параметр spec.startingDeadlineSeconds указывает, на какое время может задержаться запуск CronJob. Это работает следующим образом. CronJob запланирован на

запуск в 12-00 каждый день. Но случилось так, что в 11:59:59 умерло три узла кластера из четырех. Планировщик занят более приоритетными задачами, и до CronJob очередь запуска еще не дошла. В этом случае нам поможет параметр startingDeadlineSeconds. Если он указан, то планировщику позволено сделать отступление от жестко заданного времени запуска. Но если время запуска+startingDeadlineSeconds уже прошло, то CronJob запущен не будет.

Полезные ссылки:

- Kubernetes: Running Background Tasks With Batch-Jobs
- DaemonSet (official docs)
- Perform a Rolling Update on a DaemonSet (official docs)
- DaemonSet (Google Kubernetes Engine docs)

Задание:

1. Добавьте двум узлам вашего кластера метки usecase=loadbalancer.
2. Создайте daemonset с именем nginx-ds в namespace default, который будет запускать nginx на всех узлах с метками usecase=loadbalancer.
3. Создайте CronJob с именем cron-test в namespace default, который будет раз в 10 минут (*/10 * * * *) запускать alpine с командой curl https://lk.rebrainme.com/kubernetes/report.
4. Оправьте задание на проверку.