Теоретическое введение

Основные типы ресурсов в системе — процессорное время и оперативная память. В манифестах Kubernetes эти типы ресурсов измеряются в следующих единицах:

- CPU — в ядрах;
- RAM — в байтах.

Для каждого ресурса есть возможность задавать два типа требований — *requests* и *limits*. *Requests* — описывает минимальные требования к свободным ресурсам ноды для запуска контейнера, в то время как *limits* устанавливает жесткое ограничение ресурсов, доступных контейнеру.

Важно понимать, что в манифесте не обязательно явно определять оба типа, при этом поведение будет следующим:

Если явно задан только *limits* ресурса, то *requests* для этого ресурса автоматически принимает значение, равное *limits*. Т.е. фактически работа контейнера будет ограничена таким же количеством ресурсов, которое он требует для своего запуска.

Если для ресурса явно задан только *requests*, то никаких ограничений сверху на этот ресурс не задается — т.е. контейнер ограничен только ресурсами самой ноды.

Также существует возможность настроить управление ресурсами не только на уровне конкретного контейнера, но и на уровне namespace при помощи следующих сущностей:

- LimitRange — описывает политику ограничения на уровне контейнера/пода, нужна для того, чтобы описать ограничения по умолчанию на контейнер/под, а также предотвращать создание объектов, требующих много ресурсов, ограничивать их количество и определять возможную разницу значений в limits и requests.
- ResourceQuotas — описывают политику ограничения по всем контейнерам в ns и используется, как правило, для разграничения ресурсов по окружениям.

Каждому поду присваивается один из 3 QoS-классов:

- *guaranuted* — назначается тогда, как для каждого контейнера в поде для memory и cpu задан request и limit, причем эти значения должны совпадать
- *burstable* — хотя бы один контейнер в поде имеет request и limit, при этом request < limit
- *best effort* — когда ни один контейнер в поде не ограничен по ресурсам

При этом, когда на ноде наблюдается нехватка ресурсов (диска, памяти), kubelet начинает ранжировать и выселять под’ы по определенному алгоритму, который учитывает приоритет пода и его QoS-класс.

Т.е. при одинаковом приоритете, kubelet в первую очередь будет выселять с узла поды с QoS-классом best effort.

Когда стоит задача автоматически увеличивать и уменьшать количество pod в зависимости от использования ресурсов в её решении может помочь такая сущность k8s как *HPA (Horizontal Pod Autoscaler)*, алгоритм которого заключается в следующем:

- Определяются текущие показания наблюдаемого ресурса (currentMetricValue).
- Определяются желаемые значения для ресурса (desiredMetricValue), которые для системных ресурсов задаются при помощи request.
- Определяется текущее количество реплик (currentReplicas).

По следующей формуле рассчитывается желаемое количество реплик

*desiredReplicas = [currentReplicas * (currentMetricValue / desiredMetricValue)]*

При этом масштабирования не произойдет, когда коэффициент (currentMetricValue / desiredMetricValue) близок к 1 (при этом допустимую погрешность мы можем задавать сами, по умолчанию она равна 0.1).

Полезные ссылки

Kubernetes: лучшие практики. — СПб.: Питер, 2021. — 288 с.: ил. — (Серия «Для профессионалов»).

K8S для начинающих. Первая часть — Текст: электронный [сайт]. — URL: https://habr.com/ru/post/589415/

Kubernetes или с чего начать, чтобы понять что это и зачем он нужен — Текст: электронный [сайт]. — URL: https://habr.com/ru/company/otus/blog/537162/

Основы Kubernetes — Текст: электронный [сайт]. — URL: https://habr.com/ru/post/258443/

Практическая часть

Для выполнения практической работы понадобится minikube. Необходимо выполнить все команды, создать необходимые конфигурации и отобразить их в отчете.

Запустим сервер метрик в minikube:

```
minikube addons enable metrics-server
```

Нужно проверить работу сервера меток:

```
kubectl get apiservices
```

Если API ресурсов метрики доступно, в выводе команды будет содержаться ссылка на *metrics.k8s.io*

```
NAME
v1beta1.metrics.k8s.io
```

Нужно создать пространство имён, чтобы ресурсы, которыми будем пользоваться в данном упражнении, были изолированы от остального кластера

```
kubectl create namespace *имя_фамилия_группа*
```

Для установки запроса памяти контейнеру используется поле *resources:requests*

Для ограничений по памяти используется *resources:limits*

Раздел *args* конфигурационного файла содержит аргументы для контейнера в момент старта. Аргументы *"--vm-bytes"*, *"150M"* указывают контейнеру попытаться занять 150 Мб памяти.

Далее нужно создать конфигурационный файл *memory-request-limit.yaml*:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
  namespace: mem-example
spec:
  containers:
  - name: memory-demo-ctr
    image: polinux/stress
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

Далее необходимо создать под:

```
kubectl apply -f *путь_до_локального_файла*/memory-request-limit.yaml --namespace=*имя_фамилия_группа*
```

Нужно проверить статус Pod'a:

```
kubectl get pod memory-demo --namespace=*имя_фамилия_группа*
```

Необходимо просмотреть подробную информацию о Pod'e:

```
kubectl get pod memory-demo --output=yaml --namespace=*имя_фамилия_группа*
```

Вывод команды должен показать что для контейнера в Pod'e зарезервировано 100 Мб памяти и выставлено 200 Мб ограничения.

Запустите *kubectl top*, чтобы получить метрики Pod'a:

```
kubectl top pod memory-demo --namespace=*имя_фамилия_группа*
```

Нужно произвести удаление Pod:

```
kubectl delete pod memory-demo --namespace=*имя_фамилия_группа*
```

Далее создан Pod, который попытается занять больше памяти, чем для него ограничено. Ниже представлен конфигурационный файл для Pod'a с одним контейнером, имеющим 50 Мб на запрос памяти и 100 Мб лимита памяти:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-2
  namespace: *имя_фамилия_группа*
spec:
  containers:
  - name: memory-demo-2-ctr
    image: polinux/stress
    resources:
      requests:
        memory: "50Mi"
      limits:
        memory: "100Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "250M", "--vm-hang", "1"]
```

Далее необходимо создать Pod:

```
kubectl apply -f *путь_до_локального_файла*/memory-request-limit-2.yaml --namespace=*имя_фамилия_группа*
```

Нужно просмотреть подробную информацию о Pod'e:

```
kubectl get pod memory-demo-2 --namespace=*имя_фамилия_группа*
```

В этот момент контейнер уже либо запущен, либо убит. Повторяем предыдущую команду, пока контейнер не окажется убитым:

```
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-2   0/1       OOMKilled   1          24s
```

Требуется изучить ещё более подробный вид статуса контейнера:

```
kubectl get pod memory-demo-2 --output=yaml --namespace=*имя_фамилия_группа*
```

В выводе показано, что контейнер был убит по причине недостатка памяти (OOM):

```
lastState:
   terminated:
     containerID: docker://65183c1877aaec2e8427bc95609cc52677a454b56fcb24340dbd22917c23b10f
     exitCode: 137
     finishedAt: 2022-09-20T20:52:19Z
     reason: OOMKilled
     startedAt: null
```

Так как контейнер может быть перезапущен, kubelet стартует его. Требуется выполнить команду несколько раз, чтобы увидеть, как контейнер убивается и запускается заново:

Вывод показывает, что контейнер убит, перезапущен, снова убит.:

```
kubectl get pod memory-demo-2 --namespace=*имя_фамилия_группа*
NAME            READY     STATUS             RESTARTS   AGE
memory-demo-2   0/1       CrashLoopBackOff   3          54s
```

Требуется просмотреть подробную информацию об истории Pod'a:

```
kubectl describe pod memory-demo-2 --namespace=*имя_фамилия_группа*
```

Вывод показывает, что контейнер постоянно запускается и падает:

```
... Normal  Created   Created container with id 66a3a20aa7980e61be4922780bf9d24d
... Warning BackOff   Back-off restarting failed container
```

Посмотрим детальную информацию о нодах на кластере:

```
kubectl describe nodes
```

В выводе содержится запись о том, что контейнер убивается по причине нехватки памяти:

```
Warning OOMKilling Memory cgroup out of memory: Kill process 4481 (stress) score
```

Далее необходимо удалить Pod:

```
kubectl delete pod memory-demo-2 --namespace=mem-example
```

После удаления нужно создать Pod, чей запрос памяти будет превышать ёмкость любой ноды в кластере. Ниже представлен конфигурационный файл

для Pod'a с одним контейнером, имеющим запрос памяти в 1000 Гб (что наверняка превышает ёмкость любой имеющейся ноды):

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-3
  namespace: *имя_фамилия_группа*
spec:
  containers:
  - name: memory-demo-3-ctr
    image: polinux/stress
    resources:
      limits:
        memory: "1000Gi"
      requests:
        memory: "1000Gi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

Создадим Pod:

```
kubectl apply -f *путь_до_локального_файла*/memory-request-limit-3.yaml --namespace=*имя_фамилия_группа*
```

Проверим статус Pod'a:

```
kubectl get pod memory-demo-3 --namespace=*имя_фамилия_группа*
```

Вывод показывает, что Pod имеет статус PENDING. Это значит, что он не запланирован ни на одной ноде, и такой статус будет сохраняться всё время:

```
kubectl get pod memory-demo-3 --namespace=*имя_фамилия_группа*
NAME            READY     STATUS    RESTARTS   AGE
memory-demo-3   0/1       Pending   0          25s
```

Посмотрим подробную информацию о Pod'e, включающую события:

```
kubectl describe pod memory-demo-3 --namespace=*имя_фамилия_группа*
```

Вывод показывает невозможность запуска контейнера из-за нехватки памяти на нодах:

```
Events:
  ...  Reason            Message
       ------            -------
  ...  FailedScheduling  No nodes are available that match all of the following predicates:: Insufficient memory (3).
```

Удалим Pod:

```
kubectl delete pod memory-demo-3 --namespace=*имя_фамилия_группа*
```

Удалим пространство имён. Эта операция удалит все Pod'ы, созданные в рамках данного упражнения:

```
kubectl delete namespace *имя_фамилия_группа*
```

Далее необходимо создать Pod, имеющий один контейнер. Требуется задать для контейнера запрос в 0.5 CPU и лимит в 1 CPU. Конфигурационный файл для такого Pod'a:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo
  namespace: *имя_фамилия_группа*
spec:
  containers:
  - name: cpu-demo-ctr
    image: vish/stress
    resources:
      limits:
        cpu: "1"
      requests:
        cpu: "0.5"
    args:
    - -cpus
    - "2"
```

Раздел args конфигурационного файла содержит аргументы для контейнера в момент старта. Аргумент -cpus "2" говорит контейнеру попытаться использовать 2 CPU.

Нужно создать Pod и удостовериться, что он запущен:

```
kubectl apply -f *путь_до_локального_файла*/cpu-request-limit.yaml --namespace=*имя_фамилия_группа*
```

```
kubectl get pod cpu-demo --namespace=*имя_фамилия_группа*
```

Далее нужно просмотреть детальную информацию о Pod'е:

```
kubectl get pod cpu-demo --output=yaml --namespace=*имя_фамилия_группа*
```

В выводе видно, что Pod имеет один контейнер с запросом в 500 милли-CPU и с ограничением в 1 CPU.

Требуется запустить kubectl top, чтобы получить метрики Pod'а:

```
kubectl top pod cpu-demo --namespace=*имя_фамилия_группа*
```

В этом варианте вывода Pod'ом использовано 974 милли-CPU, что лишь чуть меньше заданного в конфигурации Pod'а ограничения в 1 CPU.

```
NAME                        CPU(cores)   MEMORY(bytes)
cpu-demo                    974m         <something>
```

Далее можно удалить Pod:

```
kubectl delete pod cpu-demo --namespace=*имя_фамилия_группа*
```

Далее нужно создать Pod с запросом CPU, превышающим мощности любой ноды в вашем кластере. Ниже представлен конфигурационный файл для Pod'а с одним контейнером. Контейнер запрашивает 100 CPU, что почти наверняка превышает имеющиеся мощности любой ноды в кластере.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo-2
  namespace: *имя_фамилия_группа*
spec:
  containers:
  - name: cpu-demo-ctr-2
    image: vish/stress
    resources:
      limits:
        cpu: "100"
      requests:
        cpu: "100"
    args:
    - -cpus
    - "2"
```

Далее нужно создать Pod:

```
kubectl apply -f *путь_до_локального_файла*/cpu-request-limit-2.yaml --namespace=*имя_фамилия_группа*
```

Требуется проверить статус Pod'a:

```
kubectl get pod cpu-demo-2 --namespace=*имя_фамилия_группа*
```

Вывод показывает Pending статус у Pod'a. То есть Pod не запланирован к запуску ни на одной ноде и будет оставаться в статусе Pending постоянно:

```
NAME         READY     STATUS    RESTARTS   AGE
cpu-demo-2   0/1       Pending   0          7m
```

Нужно просмотреть подробную информацию о Pod'e, включающую в себя события:

```
kubectl describe pod cpu-demo-2 --namespace=*имя_фамилия_группа*
```

В выводе отражено, что контейнер не может быть запланирован из-за нехватки ресурсов CPU на нодах:

```
Events:
  Reason                    Message
  ------                    -------
  FailedScheduling      No nodes are available that match all of the following predicates:: Insufficient cpu (3).
```

Далее можно удалить Pod:

```
kubectl delete pod cpu-demo-2 --namespace=*имя_фамилия_группа*
```

Вопросы к практической работе

1. Назовите 3 QoS-класса
2. Назовите основные ресурсы системы и единицы их измерения в Kubernetes
3. Для чего нужен HPA?
4. Для чего необходимо устанавливать ограничения в Kubernetes?
5. Что будет с узлом при превышении ограничений?

Критерии оценки

За выполнение данной практической работы можно максимально получить 2 балла.

Критерии на выставление 2 баллов:

- Соблюдены общие требования выполнения практических работ, представленные в документе “Требования к выполнению практических работ”.
- В отчете отображены все этапы конфигурации системы
- Показаны конфигурации в Minikube
- Сделан отчет с описанием и скриншотами выполненных заданий
- Дан полный и развернутый ответ на все вопросы преподавателя, как по вопросам к практике, так и по дополнительным вопросам к выполненному заданию.

Критерии на выставление 1 балла:

- Соблюдены общие требования выполнения практических работ, представленные в документе “Требования к выполнению практических работ”.
- В отчете отображены все этапы конфигурации системы
- Не показаны конфигурации в Minikube
- Сделан отчет с описанием и скриншотами выполненных заданий
- Дан полный и развернутый ответ на все вопросы преподавателя, как по вопросам к практике, так и по дополнительным вопросам к выполненному заданию.

Критерии на выставление 0 баллов:

- Не соблюдены общие требования выполнения практических работ, представленные в документе “Требования к выполнению практических работ”.
- В отчете не отражены все этапы конфигурации системы
- Не показаны конфигурации в Minikube
- Сделан отчет с описанием и скриншотами выполненных заданий.

- Студент не смог ответить ни на вопросы к практической работе, ни на вопросы к ходу выполнения работы.