Оглавление

Благодаря этой конфигурации наше клиентское приложение имеет доступ к паролю, который позволяет ему войти в сервис Redis. Аналогичным образом использование пароля настраивается и в самом сервисе; мы подключаем секретный том к Redis pod и загружаем пароль из файла.

Stateful-развертывание простой базы данных

Развертывание stateful принципиально не отличается от развертывания клиентского приложения, которое мы рассматривали в предыдущих разделах, однако наличие состояния вносит дополнительные сложности. Прежде всего, планирование функционирования pod в Kubernetes зависит от ряда факторов, таких как работоспособность узла, обновление или перебалансировка. Если данные экземпляра Redis хранятся на каком-то конкретном сервере или в самом контейнере, то будут потеряны при миграции или перезапуске данного контейнера. Чтобы этого избежать, при выполнении в Kubernetes stateful-приложений нужно обязательно использовать удаленные *постоянные тома* (`PersistentVolumes`).

Kubernetes поддерживает различные реализации объекта `PersistentVolume`, но все они имеют общие свойства. Как и секретные тома, описанные ранее, они привязываются к pod и подключаются к контейнеру по определенному пути. Их особенностью является то, что они обычно представляют собой удаленные хранилища, которые подключаются по некоему сетевому протоколу: или файловому (как в случае с NFS и SMB), или блочному (как в случае с iSCSI, облачными дисками и т. д.). В целом для таких приложений, как базы данных, предпочтительны блочные диски, поскольку обеспечивают лучшую производительность. Но если скорость работы не настолько важна, то файловые диски могут быть более гибкими.

Управление состоянием, как в Kubernetes, так и в целом, — сложная задача. Если среда, в которой вы работаете, поддерживает сервисы с сохранением состояния (stateful) (например, MySQL или Redis), то обычно лучше использовать именно их. Сначала тарифы на SaaS (Software as a Service — программное обеспечение как услуга) могут показаться высокими, но если учесть все операционные требования к поддержанию состояния (резервное копирование, обеспечение локальности и избыточности данных и т. д.) и тот факт, что наличие состояния осложняет перемещение приложений между кластерами Kubernetes, то становится

> очевидно, что в большинстве случаев высокая цена SaaS себя оправдывает. В средах с локальным размещением, где сервисы SaaS недоступны, имеет смысл организовать отдельную команду специалистов, которая будет предоставлять услугу хранения данных в рамках всей организации. Это, несомненно, лучше, чем позволять каждой команде выкатывать собственное решение.

Для развертывания сервиса Redis мы воспользуемся ресурсом `StatefulSet`. Это дополнение к `ReplicaSet`, которое появилось уже после выхода первой версии Kubernetes и предоставляет более строгие гарантии, такие как согласованные имена (никаких случайных хешей!) и определенный порядок увеличения и уменьшения количества pod (scale-up, scale-down). Это не так важно, когда развертывается одноэлементное приложение, но если вам нужно развернуть состояние с репликацией, то данные характеристики придутся очень кстати.

Чтобы запросить постоянный том для нашего сервиса Redis, мы воспользуемся `PersistentVolumeClaim`. Это своеобразный запрос ресурсов. Наш сервис объявляет, что ему нужно хранилище размером 50 Гбайт, а кластер Kubernetes определяет, как выделить подходящий постоянный том. Данный механизм нужен по двум причинам. Во-первых, он позволяет создать ресурс `StatefulSet`, который можно переносить между разными облаками и размещать локально, не заботясь о конкретных физических дисках. Во-вторых, несмотря на то, что том типа `PersistentVolume` можно подключить лишь к одному pod, запрос тома позволяет написать шаблон, доступный для реплицирования, но при этом каждому pod будет назначен отдельный постоянный том.

Ниже показан пример ресурса `StatefulSet` для Redis с постоянными томами:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
```

```
    spec:
      containers:
      - name: redis
        image: redis:5-alpine
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - name: data
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 10Gi
```

В результате будет развернут один экземпляр сервиса Redis. Но, допустим, вам нужно реплицировать кластер Redis, чтобы масштабировать запросы на чтение и повысить устойчивость к сбоям. Для этого, очевидно, следует довести количество реплик до трех, но в то же время сделать так, чтобы для выполнения записи новые реплики подключались к ведущему экземпляру Redis.

Когда мы добавляем в объект `StatefulSet` новый неуправляемый (headless) сервис, для него автоматически создается DNS-запись `redis-0.redis`; это IP-адрес первой реплики. Вы можете воспользоваться этим для написания сценария, пригодного для запуска во всех контейнерах:

```
#!/bin/sh

PASSWORD=$(cat /etc/redis-passwd/passwd)

if [[ "${HOSTNAME}" == "redis-0" ]]; then
  redis-server --requirepass ${PASSWORD}
else
  redis-server --slaveof redis-0.redis 6379 --masterauth ${PASSWORD}
  --requirepass ${PASSWORD}
fi
```

Этот сценарий можно оформить в виде `ConfigMap`:

```
kubectl create configmap redis-config --from-file=launch.sh=launch.sh
```

Затем объект `ConfigMap` нужно добавить в `StatefulSet` и использовать его как команду для управления контейнером. Добавим также пароль для аутентификации, который создали ранее.

Полное определение сервиса Redis с тремя репликами выглядит следующим образом:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:5-alpine
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - name: data
          mountPath: /data
        - name: script
          mountPath: /script/launch.sh
          subPath: launch.sh
        - name: passwd-volume
          mountPath: /etc/redis-passwd
        command:
        - sh
        - -c
        - /script/launch.sh
      volumes:
      - name: script
        configMap:
          name: redis-config
          defaultMode: 0777
      - name: passwd-volume
        secret:
          secretName: redis-passwd
  volumeClaimTemplates:
  - metadata:
      name: data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 10Gi
```

Создание балансировщика нагрузки для TCP с использованием Service

Итак, мы развернули stateful-сервис Redis; теперь его нужно сделать доступным для нашего клиентского приложения. Для этого создадим два разных `Service` Kubernetes. Первый будет читать данные из Redis. Поскольку они реплицируются между всеми тремя участниками `StatefulSet`, для нас несущественно, к какому из них будут направляться наши запросы на чтение. Следовательно, для этой задачи подойдет простой `Service`:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis
  name: redis
  namespace: default
spec:
  ports:
  - port: 6379
    protocol: TCP
    targetPort: 6379
  selector:
    app: redis
  sessionAffinity: None
  type: ClusterIP
```

Выполнение записи потребует обращения к ведущей реплике Redis (под номером 0). Создайте для этого *неуправляемый* (headless) `Service`. У него нет IP-адреса внутри кластера; вместо этого он задает отдельную DNS-запись для каждого pod в `StatefulSet`. То есть мы можем обратиться к нашей ведущей реплике по доменному имени `redis-0.redis`:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis-write
  name: redis-write
spec:
  clusterIP: None
  ports:
  - port: 6379
  selector:
    app: redis
```

Таким образом, если нам нужно подключиться к Redis для сохранения каких-либо данных или выполнения транзакции с чтением/записью, то мы

можем собрать отдельный клиент, который будет подключаться к серверу `redis-0.redis-write`.

Направление трафика к серверу статических файлов с помощью Ingress

Заключительный компонент нашего приложения — *сервер статических файлов*, который отвечает за раздачу HTML-, CSS-, JavaScript-файлов и изображений. Отделение сервера статических файлов от нашего клиентского приложения, предоставляющего API, делает нашу работу более эффективной и целенаправленной. Для раздачи файлов можно воспользоваться готовым высокопроизводительным файловым сервером наподобие NGINX; при этом команда разработчиков может сосредоточиться на реализации нашего API.

К счастью, ресурс `Ingress` позволяет очень легко организовать такую архитектуру в стиле мини/микросервисов. Как и в случае с клиентским приложением, мы можем описать реплицируемый сервер NGINX с помощью ресурса `Deployment`. Соберем статические образы в контейнер NGINX и развернем их в каждой реплике. Ресурс `Deployment` будет выглядеть следующим образом:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: fileserver
  name: fileserver
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: fileserver
  template:
    metadata:
      labels:
        app: fileserver
    spec:
      containers:
      - image: my-repo/static-files:v1-abcde
        imagePullPolicy: Always
        name: fileserver
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        resources:
          requests:
            cpu: "1.0"
```

```
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
      dnsPolicy: ClusterFirst
      restartPolicy: Always
```

Теперь, запустив реплицируемый статический веб-сервер, вы можете аналогичным образом создать ресурс `Service`, который будет играть роль балансировщика нагрузки:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: fileserver
  name: fileserver
  namespace: default
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: fileserver
  sessionAffinity: None
  type: ClusterIP
```

Итак, у вас есть `Service` для сервера статических файлов. Добавим в ресурс `Ingress` новый путь. Необходимо отметить, что путь `/` должен идти *после* `/api`, иначе запросы API станут направляться серверу статических файлов. Обновленный ресурс `Ingress` будет выглядеть следующим образом:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: frontend-ingress
spec:
  rules:
  - http:
    paths:
    - path: /api
      backend:
        serviceName: frontend
        servicePort: 8080
    # Примечание: этот раздел должен идти после /api,
    # иначе он будет перехватывать запросы.
    - path: /
      backend:
        serviceName: fileserver
        servicePort: 80
```

Параметризация приложения с помощью Helm

Все, что мы обсуждали до сих пор, было направлено на развертывание одного экземпляра нашего сервиса в одном кластере. Но в реальности сервисы почти всегда приходится развертывать в нескольких разных средах (даже при условии, что они находятся в общем кластере). Вы можете быть разработчиком-одиночкой, который занимается всего одним приложением, но если хотите, чтобы внесение изменений не мешало пользователям работать, то вам понадобится как минимум две версии: отладочная и промышленная. И прибавив к этому интеграционное тестирование и CI/CD, мы получим следующее: даже при работе с одним сервисом и наличии лишь пары разработчиков нужно выполнять развертывание по меньшей мере в трех разных средах, и это далеко не предел, если приложение должно справляться со сбоями на уровне вычислительного центра.

Начальная стратегия для борьбы со сбоями у многих команд заключается в тривиальном копировании файлов из одного кластера в другой. Вместо одного каталога frontend/ они используют два: frontend-production/ и frontend-development/. Такой способ опасен, поскольку разработчикам приходится следить за тем, чтобы файлы оставались синхронизированными. Этого можно было бы легко добиться, если бы эти каталоги должны были быть идентичными. Но некоторые расхождения между отладочной и промышленной версиями нормальны, так как вы будете разрабатывать новые возможности; крайне важно, чтобы эти расхождения были намеренными и простыми в управлении.

Еще один подход состоит в использовании веток и системы контроля версий; центральный репозиторий разделяется на промышленную и отладочную ветки, разница между которыми видна невооруженным глазом. Это может быть хорошим вариантом для некоторых команд, но если вы хотите развертывать ПО сразу в нескольких средах (например, система CI/CD может выполнять развертывание в разных регионах облака), то переключение между ветками будет проблематичным.

В связи с этим большинство людей в итоге выбирают *систему шаблонов*. Идея в том, что централизованный каркас конфигурации приложения образуют шаблоны, которые *подставляются* для той или иной среды на основе параметров. Таким образом, вы можете иметь одну общую конфигурацию, при необходимости легко подгоняемую под определенные условия. Для Kubernetes есть множество разных систем шаблонов, но наиболее популярна, безусловно, Helm (helm.sh).

В Helm приложения распространяются в виде так называемых *чартов* с файлами внутри.

В основе чарта лежит файл `chart.yaml`, в котором определяются его метаданные:

```
apiVersion: v1
appVersion: "1.0"
description: A Helm chart for our frontend journal server.
name: frontend
version: 0.1.0
```

Этот файл размещается в корневом каталоге чарта (например, в frontend/). Там же находится каталог templates, внутри которого хранятся шаблоны. Шаблон, в сущности, представляет собой YAML-файл, похожий на приводимые в предыдущих примерах; разница лишь в том, что отдельные его значения заменены ссылками на параметры. Скажем, представьте, будто хотите параметризировать количество реплик в своем клиентском приложении. Вот что содержал наш исходный объект `Deployment`:

```
...
spec:
  replicas: 2
...
```

В файле шаблона (`frontend-deployment.tmpl`) данный раздел выглядит следующим образом:

```
...
spec:
  replicas: {{ .replicaCount }}
...
```

Это значит, что при развертывании чарта для поля `replicas` будет подставлен подходящий параметр. Сами параметры определены в файле `values.yaml`, предназначенном для конкретной среды, в котором развертывается приложение. Для этого простого чарта файл `values.yaml` выглядел бы так:

```
replicaCount: 2
```

Теперь, чтобы собрать все указанное вместе, вы можете развернуть данный чарт с помощью утилиты `helm`, как показано ниже:

```
helm install path/to/chart --values path/to/environment/values.yaml
```

Эта команда параметризирует ваше приложение и развернет его в Kubernetes. Со временем параметризация будет расширяться, охватывая все разнообразие сред выполнения вашего приложения.

Рекомендации по развертыванию сервисов

Kubernetes — эффективная система, которая может показаться сложной. Однако процесс развертывания обычного приложения легко упростить, если следовать общепринятым рекомендациям.

- Большинство сервисов нужно развертывать в виде ресурса `Deployment`. Объекты `Deployment` создают идентичные реплики для масштабирования и обеспечения избыточности.
- Для доступа к объектам `Deployment` можно использовать объект `Service`, который, в сущности, является балансировщиком нагрузки. `Service` может быть доступен как изнутри (по умолчанию), так и снаружи. Если вы хотите, чтобы к вашему HTTP-приложению можно было обращаться, то используйте контроллер `Ingress` для добавления таких возможностей, как маршрутизация запросов и SSL.
- Рано или поздно ваше приложение нужно будет параметризировать, чтобы сделать его конфигурацию более пригодной к использованию в разных средах. Для этого лучше всего подходят диспетчеры пакетов, такие как Helm (helm.sh).

Резюме

Несмотря на свою простоту, приложение, созданное нами в этой главе, охватывает практически все концепции, которые могут понадобиться вам в более крупных и сложных проектах. Понимание того, как сочетаются эти фундаментальные компоненты, и умение их использовать — залог успешного применения Kubernetes.

Использование системы контроля версий, аудита изменений кода и непрерывной доставки ваших сервисов позволит любым проектам, которые вы создаете, иметь прочный фундамент. Эта основополагающая информация пригодится вам при изучении более сложных тем, представленных в других главах.