# 10. Управление ресурсами Kubernetes

«Согласно легенде в компании Google развертывают более двух миллиардов контейнеров в неделю». Келси Хайтауэр, Брендан Бернс, Джо Беда, авторы книги “Kubernetes: Up & Running”

До этого момента мы проводили развертывания своих микросервисов в кластере Kubernetes так, как если бы они обладали бесконечным количеством вычислительной мощности и памяти. Для наших простейших примеров даже миниатюрные локальные кластеры minikube и kind способны запустить десятки экземпляров наших микроскопических сервисов, однако все сразу же меняется в реальных условиях и развертывании настоящих приложений.

Кластер Kubernetes, как мы узнали еще в самом начале знакомства с ним, состоит из узлов (nodes), чаще всего виртуальных (реже реальных) машин Linux. Сложив мощность процессоров и памяти этих узлов, мы получим все ресурсы процессора и памяти, доступные для развертывания своих приложений. Однако развертывать мы будем больше не монолит, а систему микросервисов. Сразу же возникает вопрос разделения доступных ресурсов - какие микросервисы получат больше вычислительной мощности процессора и памяти? К примеру, что если микросервису Java на основе Spring Boot не хватит памяти в куче (heap) при пиковой загрузке, потому что оставшаяся память на этом же узле будет занята десятком микросервисов на Go? Качество и скорость обслуживания запросов резко упадет, а пользователи быстро разочаруются.

В динамическом мире Kubernetes, где сосуществуют десятки (а иногда и сотни и тысячи, как в компаниях Uber или Netflix) микросервисов, мы не можем полагаться на то, что каждый из них чудесным образом будет делиться ресурсами. В идеале, каждый микросервис и компонент системы должен описать, сколько их ему необходимо.

Управляющие системы Kubernetes же выступают в роли *планировщика, или диспетчера задач* (*task scheduler*), выясняя на каких узлах системы достаточно ресурсов, как эффективно они используются, и какой узел наилучшим образом подойдет на запуска нового контейнера. Планирование задач является фундаментальной частью операционных систем, и Kubernetes по сути выступает здесь как операционная система для наших микросервисов в облаке.

## Распределение ресурсов по умолчанию

Если требования микросервисов к ресурсам не указаны явно, управление Kubernetes разделит доступные на узле процессор и память примерно поровну, но только в начале работы новых контейнеров (в отсеках pod). Даже в наших простых примерах очевидно, что ресурсов микросервис time-service, написанный на Go, будет потреблять намного меньше, чем более сложный, написанный на Java weekend-service. Если мы решим увеличить количество экземпляров, или столкнемся с пиковой нагрузкой на сервисы, рано или поздно ресурсов станет не хватать, и именно ресурсы, занятые time-service в примерно одинаковой пропорции, будут простаивать впустую. Как минимум, возникает проблема простоя ресурсов, все более серьезная при увеличении нагрузки на систему - единственным выходом станет добавление мощности в кластер в виде более мощных узлов (node), или большего их количества - а это дорого и неэффективно.

Вторая проблема может стать причиной отказа системы. Как мы сказали, ресурсы разделяются примерно поровну только в начале работы. Не совсем удачная версия или обновление может привести к тому, что микросервис станет потреблять больше вычислительной мощности процессора, или будет выпущен с утечкой памяти. Так как верхней границы у потребления ресурсов нет, постепенно такой микросервис может занять большую часть памяти и процессора, и привести к медленной работе остальных микросервисов, работающих на том же узле кластера. Недостаток памяти тут же приведет к тому, что на таком узле станет невозможно запустить требовательные к ресурсам компоненты системы, например большой Java-сервис.

Именно поэтому хорошей практикой работы в Kubernetes является определение требований к ресурсам ко всем микросервисам системы. Если в процессе разработки и тестирования, в условиях мощного кластера, ими еще можно пренебречь, в эксплуатации реальной системы они совершенно необходимы.

## Требования к ресурсам в развертываниях Deployment

Как и все, что касается развертывания и запуска микросервисов в кластере Kubernetes, требования к ресурсам будут описаны в объекте Deployment. Именно там мы описываем образы контейнеров для запуска отсеков Pod, количество их экземпляров, проверки готовности и жизнеспобности, логично что именно там же мы объявим какое количество ресурсов понадобится для работы нашего микросервиса (или просто какого-то компонента системы).

Давайте опишем минимально необходимое количество ресурсов для нашего сервиса выходного дня weekend-service - он написан на Java, и естественным образом потребует некоторого количества памяти для виртуальной машины и “кучи” (heap), и чуть большей вычислительной мощности, чем миниатюрный time-service, написанный на Go. Ресурсы объявляются в подсекции resources, там же где проверки готовности. Для простоты создадим отдельный файл Deployment в папке weekend-service/k8s/resources:

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 labels:  
 app: weekend-service  
 name: weekend-service  
…  
 spec:  
 containers:  
 - name: weekend-service  
 image: ivanporty/weekend-service:0.2.0  
 readinessProbe:  
 …   
 resources:  
 requests:  
 memory: "128M"  
 cpu: "1"

Мы объявляем, что нашему микросервису понадобятся (секция requests) как минимум 128 мегабайт памяти (memory) - довольно большое количество для миниатюрного сервиса, но виртуальная машина Java известна своими аппетитами, и один процессор (cpu). Ресурсы в Kubernetes можно описывать с довольно большой точностью:

* Процессор (cpu) - мы указали один “процессор”, это означает, что *в целом* наше приложение будет использовать вычислительную мощность *одного ядра* процессора, того типа и мощности, что использован на узлах (node) нашего кластера. Это может быть или ядро реального процессора (если узел является выделенным сервером), или виртуального (для виртуальных машин, то есть для большинства кластеров в облаке). В целом значит то, что выделять и фиксировать за нами ядро процессора вряд ли станут - в целом доступная мощность будет соответствовать одному ядру, а распределяться наши задачи будут по всем доступным ядрам, в зависимости от алгоритма Linux. Минимальное количество процессора - одна его милли-часть, то есть 0.001 от целого ядра, обозначаемая маленькой m (cpu: "1m")
* Память (memory) - тут все интереснее, потому что указывать требуемую память можно в очень большом количестве вариантов и единиц. Мы указали мегабайты (M) - это десятичная система, то есть миллион байт. Есть еще мебибайты (Mi), в двоичной системе (1024^2), то же самое относится к кило- и киби, гига- и гиби. Если указать простое число, это будут просто байты. А вот есть сказать 64m, это будет 64 милли-байта, сомнительно что мы найдем микросервис способный работать на таком количестве памяти, так что используйте правильный регистр.

Удостоверимся теперь, что ресурсы, запрошенные нами, доступны, и описаны верно. Проведем обновленное развертывание (если в вашем кластере не запущен сервис weekend-service, разверните его в дополнение к обновленному развертыванию):

$ kubectl apply -f k8s/resources/  
deployment.apps/weekend-service created  
# если в кластере еще не был развернут weekend-service  
$ kubectl apply -f k8s/k8s-weekend-svc.yaml  
service/weekend-service created

Как видно, Kubernetes принял наше обновленное развертывание. В качестве простого упражнения остается переадресовать порт сервиса, и проверить что сервис выходного дня работает (а еще ему понадобится работающий сервис-партнер time-service - разверните его в кластере, если он там не остался после предыдущих экспериментов).

Мы определили минимально необходимые нам ресурсы (resource.requests), однако для стабильной безотказной работы кластера важнее определить верхнюю границу используемых ресурсов (limit). Мы уже обсудили, как неудачная версия сервиса с утечкой памяти может внести хаос в ресурсы узла node, и ограничение ресурсов позволяет этого избежать. Верхние границы ресурсов определяются примерно так же:

…  
 resources:  
 requests:  
 memory: "128M"  
 cpu: "1"  
 limits:  
 memory: "512M"  
 cpu: "2"  
…

Заново разверните обновленный объект Deployment, и теперь наш микросервис weekend-service даже при всем желании не сможет превысить 512 мегабайт памяти и *в общем* 2 ядра процессора. Дело в том, что при работе код сервиса может превысить лимит процессора, но только в момент непосредственного исполнения на процессоре. После его паузы управление Kubernetes и среда запуска контейнеров поймет, что процесс превышает верхнюю границу загрузки и назначит ему меньший приоритет, то есть станет давать меньше времени процессора.

С памятью же это жесткий лимит, и при попытке использовать больше памяти процесс получит ошибку от операционной системы и будет перезапущен. Это легко проверить - уменьшите лимит памяти до 128 (или даже 256) мегабайт, и вы увидите что weekend-service не сможет запуститься - он постоянно будет завершаться с ошибкой OOMKilled - процесс будет пытаться получить больше памяти чем он запросил (и это ожидаемо для приложений виртуальной машины Java). Иногда даже если процесс сможет запуститься с меньшим количеством памяти, он не сможет запустить сервер или обслуживать проверки готовности (readiness) или жизнеспособности (liveness) - и Kubernetes будет постоянно перезапускать его в попытках исправить ситуацию.

Назначив верхние лимиты памяти и процессора, и запросив минимально необходимые ресурсы (часто они совпадают, в этом случае можно описать просто лимит resources.limits), мы сможем обезопасить себя от истощения ресурсов, и позволить управлению Kubernetes максимально эффективно использовать все ресурсы кластера.

С точки зрения разработчика, выяснить точное количество требуемых ресурсов для своего приложения довольно непросто. Здесь скорее всего весьма кстати придется тестирование под нагрузкой (load testing), в идеале под пиковой нагрузкой (stress testing). Проводя такое тестирование для своих микросервисов, вы сможете выяснить их стабильность, найдете узкие места, поведение при пиковой нагрузке, а заодно выясните разумные верхние границы ресурсов, позволяющие найти баланс между горизонтальным масштабированием и оптимальной производительностью.

Как проверить текущее количество ресурсов, используемое отсеками pod и работающими в них контейнерами, мы уже выяснили в прошлой главе, когда экспериментировали с автоматическим масштабированием:

$ kubectl top pods  
NAME CPU(cores) MEMORY(bytes)   
time-service-68978c4db5-tn5nw 0m 5Mi   
…

Теперь, зная как описываются требования к ресурсам в файлах YAML и в каких единицах описывается количество процессора и памяти, понять вывод команды kubectl top еще проще. Можно использовать ее вывод, чтобы определить примерные верхние границы ресурсов для всех своих микросервисов одновременно - подайте на точки доступа своей системы, работающей в кластере, большую нагрузку, и соберите статистику загрузки всех микросервисов и компонентов в процессе этого теста.

Еще один способ наблюдать за загрузкой кластера и работающих в нем контейнеров - использовать стандартную панель мониторинга (Kubernetes dashboard), которую легко установить в любом кластере Kubernetes. Если вы используете minikube, это еще проще:

$ minikube dashboard

## Пространства имен Namespace

По мере роста вашей системы микросервисов (или просто компонентов, в зависимости от ее архитектуры - Kubernetes с одинаковой легкостью развертывает любые системы) ваш производственный (production) кластер будет становиться все более сложным. Вам понадобятся авторизация пользователей, правила доступа, возможно вы решите установить сервисную сетку service mesh, такую как Istio. Все это сделает систему более мощной, и заниматься этим будут скорее всего операторы кластера, но для нас как для разработчиков скорее всего наступит момент, когда воспроизвести *производственную среду* (production environment) на своем локальном кластере станет невозможно или слишком сложно.

Конечно же, мы будем писать для своих микросервисов множество модульных тестов (unit tests), создадим набор интеграционных тестов (integration tests), и будет проверять базовое взаимодействие микросервисов с помощью непрерывной интеграции (CI, continuous integration). Однако ничто не дает разработчику такой уверенности в том, что он делает, как работа и отладка сложных проблем в среде, идентичной или практически идентичной той, в которой в итоговом результате система будет работать.

Было бы идеально “отделить” часть производственного или идентичного ему кластера для разработки и экспериментов. Нужен только мощный кластер. Ресурсы обычно не являются проблемой - Kubernetes способен управлять тысячами узлов (node), и горизонтальное масштабирование стандартными машинами Linux дешево. Как раз для этого предназначены *пространства имен* (*namespace*) кластера.

Пространства имен позволяют отделить часть ресурсов кластера, изолировать объекты друг от друга, и при этом получить все общие настройки, такие как сервисные сетки, управление через kubectl, и все остальное. Мы упоминали, что у всех объектов Kubernetes, таких как Service и развертывания Deployment, должны быть уникальные имена - но *только в пределах одного пространства имен*. Таким образом, мы сможем развернуть свой микросервис time-service множество раз в одном кластере, если будет это делать в отдельных пространствах имен.

Мы уже знаем, что команда kubectl очень логично структурирована, и легко выясним, какие пространства имен уже существуют в кластере:

$ kubectl get namespaces  
NAME STATUS AGE  
default Active 2d12h  
kube-node-lease Active 2d12h  
kube-public Active 2d12h  
kube-system Active 2d12h

В любом кластере есть пространство имен по умолчанию (default), именно там мы разворачивали свои экспериментальные микросервисы. Можно еще раз проверить, что в одном пространстве имен все объекты Kubernetes должны быть уникальны - перейдем в папку time-service и запустим следующее:

$ kubectl apply -f k8s/  
deployment.apps/time-service created  
service/time-service created  
# попробуем создать развертывание еще раз  
$ kubectl create -f k8s/k8s-time-deploy.yaml   
Error from server (AlreadyExists): error when creating "k8s/k8s-time-deploy.yaml": deployments.apps "time-service" already exists

Управление Kubernetes не даст нам создать объект, уже существующий в текущем пространстве имен (по умолчанию это всегда default). Предположим, что в пространстве default работает или реальная система, или интеграционные тесты для нее. Мы же хотим посмотреть, будет ли работать новая версия микросервиса в реальном производственном кластере. Решение - создать свое собственное пространство имен для разработчиков:

$ kubectl create namespace dev  
namespace/dev created  
$ kubectl get namespaces  
NAME STATUS AGE  
default Active 2d12h  
dev Active 17s  
…

Простейшая команда create создала нам новое пространство имен dev (для разработчиков dev[elopers]), и не потребовала никаких дополнительных параметров. Мы сможем разворачивать свои микросервисы в новом пространстве имен, указав его в своих командах (флаг -n, или --namespace).

$ kubectl create -f k8s/k8s-time-deploy.yaml -n dev  
deployment.apps/time-service created  
$ kubectl get deploy --namespace dev  
NAME READY UP-TO-DATE AVAILABLE AGE  
time-service 1/1 1 1 13s

Теперь ничто нам не помешало создать новое развертывание time-service, несмотря на то, что точно такое же уже работает в пространстве default. Мы получили свою песочницу для самых смелых экспериментов, при этом сохранив все настройки производственного кластера - в общем случае конечно разработчики не используют производственный кластер, а скорее кластер для тестирования, экспериментов, или промежуточный (staging) кластер, полностью идентичный реальным условиям.

Мы уже обладаем некоторым опытом в Kubernetes, чтобы понимать, что команда kubectl create хороша только для быстрых экспериментов, а для создания легко воспроизводимой среды, с историей изменений, нам нужен декларативный подход и манифест YAML. Тут нам снова поможет флаг --dry-run=client:

$ kubectl create namespace dev --dry-run=client -o yaml  
…

Убрав поля со значением null, мы получим свой манифест и сохраним его в отдельном папке k8s-ops - это уже объект уровня кластера и к отдельным приложениям или микросервисам не относится:

apiVersion: v1  
kind: Namespace  
metadata:  
 name: dev  
spec: {}  
status: {}

Значения остальных полей легко найти в документации. Разместим описание своих пространств имен в системе контроля версий и используем декларативную команду apply:

# удалим ранее созданное пространство  
$ kubectl delete namespace dev  
namespace "dev" deleted  
$ kubectl apply -f k8s-ops/dev-namespace.yaml   
namespace/dev created

Так как мы уже создали пространство dev ранее, нам пришлось его удалить - когда вы запустите эту команду, то увидите, что она займет некоторое время - дело в том, что мы уже развернули там микросервис time-service, а удаление пространства имен означает удаление всех объектов Kubernetes в нем, и соответственно остановку всех развертываний и отсеков. Очень удобно для быстрой очистки после экспериментов, не стоит только удалять работу коллег по команде, если они используют то же самое пространство имен.

### Ресурсы пространства имен

В этой главе мы говорим про управление ресурсами, и неожиданно создаем себе отдельное пространство имен на том же самом кластере, где будем проводить свои эксперименты с новой функциональностью. Как разделяются ресурсы между пространствами имен? По умолчанию ресурсы будут ограничены только теми границами, что описывают развертывания Deployment. Если же их нет, а очередной эксперимент разработчиков закончился большой утечкой памяти, то оставшихся ресурсов может не хватить на гораздо более важные задачи.

Чтобы ограничить ресурсы пространства имен, используется отдельный объект под названием ResourceQuota. Он довольно подробно описывает, как ограничить потребление ресурсов в пространстве имен, можно даже указать точное максимальное количество отсеков pod или развертываний! Давайте для простоты ограничим свое экспериментальное пространство имен памятью и вычислительной мощностью:

apiVersion: v1  
kind: ResourceQuota  
metadata:  
 name: dev-resources  
spec:  
 hard:  
 requests.cpu: "2"  
 requests.memory: 2Gi  
 limits.cpu: "4"  
 limits.memory: 4Gi

Это простой объект с описанием ресурсов, все его поля вы легко найдете в документации Kubernetes, ну а мы создали твердые требования к минимуму и максимуму процессора и памяти (hard, и уже знакомые поля requests и limits) . Мы уже знаем единицы измерения ресурсов в Kubernetes, и отвели своему пространству имен от двух до четырех ядер процессора (в зависимости от используемых на узлах кластера процессоров), и от двух до четырех гибибайт памяти (Gi). Твердое требование означает, что превысить эти ресурсы нельзя, а все отсеки и развертывания обязаны будут объявить свои требования к ресурсам - чудесный инструмент, чтобы не забыть об этом! Привяжем объект с ограничением ресурсов к новому пространству имен:

$ kubectl apply -f k8s-ops/dev-resource-quota.yaml -n dev  
resourcequota/dev-resources created

Не забудьте флаг -n, иначе ограничения будут созданы для пространства по умолчанию default! Теперь ограничения ресурсов будут описаны для пространства имен командой describe:

$ kubectl describe namespace dev  
Name: dev  
Labels: kubernetes.io/metadata.name=dev  
Annotations: <none>  
Status: Active  
  
Resource Quotas  
 Name: dev-resources  
 Resource Used Hard  
 -------- --- ---  
 limits.cpu 0 4  
 limits.memory 0 4Gi  
 requests.cpu 0 2  
 requests.memory 0 2Gi  
…

Обладая новыми инструментами, вполне можно масштабировать новый кластер, и создать новые, ограниченные в доступе и ресурсах, пространства имен для каждого члена команды в отдельности, и общих экспериментов, как мы сделали для пространства dev.

## Дополнительные контексты kubectl

Пространства имен namespace очень хороши для разделения мощных кластеров и экспериментов команды, но постоянно следить за флагом -n просто невыносимо. На этот случай пригодится новый *контекст* (context) команды kubectl. Контекст - это именованное сочетание имени и сетевого адреса управляющего слоя кластера, настроек доступа пользователя (обычно ключ доступа и имя пользователя для управляющего слоя Kubernetes), и пространства имен, куда будут отправляться все команды kubectl. Когда мы создаем локальные кластеры minikube или kind, или настраиваем доступ к облачному кластеру, пространство имен почти что всегда по умолчанию (default), а название контекста совпадает с именем кластера (minikube).

Однако можно легко создать дополнительный контекст, который будет использовать тот же самый кластер и способ доступа, но другое пространство имен. Для начала нужно узнать имя кластера и пользователя для нашего текущего контекста:

$ kubectl config view  
apiVersion: v1  
clusters:  
- cluster:  
 certificate-authority-data: DATA+OMITTED  
 server: https://kubernetes.docker.internal:6443  
 name: docker-desktop   
contexts:  
- context:  
 cluster: docker-desktop  
 user: docker-desktop  
 name: docker-desktop  
current-context: docker-desktop  
users:  
- name: docker-desktop  
 user:  
  
…

Команда config view расскажет обо всех кластерах, настроенных в вашей системе. Здесь у нас был настроен только кластер docker-desktop, и всю информацию о текущем контексте можно быстро увидеть в секции context - имя кластера, пользователя, и контекста. Чтобы создать новый контекст для пространства имен в этом же кластере, нам понадобится указать имя пространства имен, и скопировать название кластера и пользователя:

$ kubectl config set-context docker-dev --cluster=docker-desktop --user=docker-desktop --namespace=dev  
Context "docker-dev" created.

Мы создали контекст docker-dev командой config set-context - если такого контекста не существует, она создаст его, если существует - обновит его данные. Остается переключиться на него командой use-context:

$ kubectl config use-context docker-dev  
Switched to context "docker-dev".  
$ kubectl get deploy  
No resources found in dev namespace.

Теперь любые команды kubectl будут работать по умолчанию с пространством имен dev, и флаги -n больше нам не понадобятся. Переключаться между контекстами можно как командой use-context, так и с помощью бесчисленных визуальных помощников - таких как k9s, Lens, или прямо в редакторе кода с помощью плагина Cloud Code. Аналогичный подход можно использовать для настройки множества контекстов - возможно вам понадобится работать с кластером с различными уровнями доступа (можно создать несколько пользователей, и отдельные контексты для каждого). Можно даже настроить свой экспериментальный кластер - его адрес и способ доступа к нему - вручную.

## Резюме

* Ресурсы кластера Kubernetes не безграничны, а системы из множества компонентов и микросервисов сложны и используют самые разные сочетания ресурсов под нагрузкой или без нее. В идеале любой микросервис, работающий в производственной среде Kubernetes должен объявить минимально и максимально требуемые ему ресурсы.
* Основными ресурсами в Kubernetes являются вычислительная мощность, измеряемая в ядрах процессора (cpu), и память в байтах (memory).
* Для сложных в настройке и особенно мощных кластеров можно отделить часть кластера для тестирования и экспериментов с помощью пространств имен (namespace).
* Чем больше способов работы с кластером (к примеру, разные пространства имен или пользователи) вы используете, тем удобнее использовать дополнительные контексты команды kubectl.