ГЛАВА 1

Основы

Во вводной главе мы обсудим важные понятия Kubernetes, используемые в проектировании и реализации облачных приложений. Понимание этих абстракций, а также связанных с ними принципов и паттернов из этой книги является ключом к созданию распределенных приложений, которые можно автоматизировать с помощью Kubernetes.

Для понимания паттернов, описываемых далее, читать эту главу не обязательно. Те, кто знаком с понятиями Kubernetes, могут пропустить ее и сразу перейти к интересующей их части книги.

Путь к облачным вычислениям

Микросервисы — один из самых популярных стилей архитектур для создания облачных приложений. Они решают проблему сложности программного обеспечения за счет разделения бизнес-функций на модули и заменяют сложность разработки на сложность эксплуатации. Вот почему ключевым требованием успеха микросервисных приложений является их способность работать и масштабироваться в Kubernetes.

Существует большое количество теоретических и практических методов, а также инструментов для создания микросервисов с нуля или деления монолитных приложений на микросервисы. Большинство из этих методов основаны на приемах, описанных в книге Эрика Эванса (Eric Evans) «Domain-Driven Design»[1] (Addison-Wesley), и понятиях ограниченного контекста и агрегатов. *Ограниченные контексты* относятся к большим моделям и делят их на компоненты, а *агрегаты* помогают группировать ограниченные контексты в модули с определенными границами транзакций. Однако кроме этих понятий, связанных с бизнес-задачами, для каждой распределенной системы, независимо от того, основана она на микросервисах или нет, существуют технические проблемы, касающиеся ее внешней структуры и связыванием во время выполнения. Контейнеры и механизмы оркестрации, такие как Kubernetes, добавляют примитивы и абстракции, направленные на решение задачи создания распределенных

[1] Эванс Э. «Предметно-ориентированное проектирование (DDD): Структуризация сложных программных систем».

приложений, и мы обсудим ниже нюансы переноса распределенной системы в Kubernetes.

В этой книге мы будем рассматривать особенности взаимодействий контейнеров и платформ, считая контейнеры черными ящиками. Однако следует подчеркнуть важность внутреннего устройства контейнеров. Контейнеры и облачные платформы дают огромные преимущества распределенным приложениям, но если вы поместите в контейнеры мусор, то получите распределенный мусор в большем масштабе. На рис. 1.1 показаны составляющие хороших облачных приложений и место паттернов Kubernetes среди них.

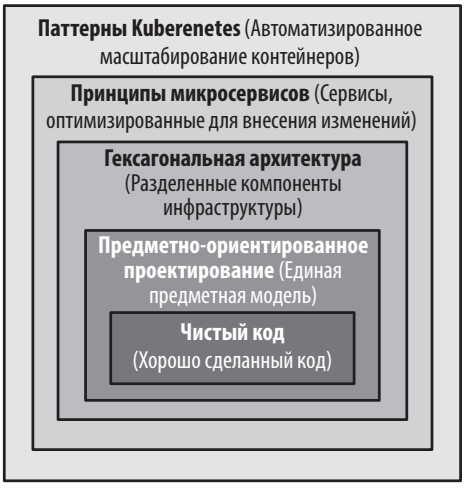

Рис. 1.1. Путь к облачным средам

В общем случае, чтобы создавать хорошие облачные приложения, требуется знать разные подходы к проектированию:

- Нижний уровень — это *уровень программного кода*. Здесь каждая переменная, метод и класс, которые вы создаете, оказывают прямое влияние на сложность поддержки приложения в долгосрочной перспективе. Влияние команды разработчиков и создаваемых ими артефактов не зависит от технологии контейнеризации и платформы оркестрации контейнеров. Важно поддерживать разработчиков, которые стремятся писать чистый код, добавляют необходимое количество автоматических тестов, постоянно улучшают качество кода и являются мастерами программной разработки.
- *Предметно-ориентированное проектирование* (Domain-Driven Design, DDD) — это метод проектирования программ с позиции бизнеса с целью получить архитектуру, как можно более близкую к реальному миру. Он лучше всего подходит для объектно-ориентированных языков программирования, хотя есть и другие хорошие способы моделировать и проектировать ПО для решения про-

блем реального мира. Модель с правильно определенными границами бизнеса и транзакций, простыми в использовании интерфейсами и многофункциональным API является основой для успешной контейнеризации и дальнейшей автоматизации.

- *Гексагональная архитектура* и ее варианты, такие как Onion и Clean, повышают гибкость и удобство обслуживания приложений благодаря разделению их компонентов и предоставлению стандартизированных интерфейсов для взаимодействия с ними. Отделяя основную бизнес-логику системы от инфраструктурных компонентов, гексагональная архитектура упрощает перенос системы в различные среды или платформы. Такая архитектура дополняет предметно-ориентированное проектирование и помогает получить код приложения с четкими границами и вынесенными инфраструктурными зависимостями.
- *Архитектурный стиль микросервисов* и 12-факторная методология (https://12factor.net) очень быстро эволюционировали до практических стандартов и определяют принципы и методы проектирования и разработки изменяемых распределенных приложений. Применение этих принципов позволяет создавать отказоустойчивые приложения, оптимизированные для масштабирования, устойчивости и частых изменений — общих требований к современному ПО.
- *Контейнеры* превратились в стандартный способ упаковки и запуска распределенных приложений, причем не только микросервисных. Создание модульных, переиспользуемых контейнеров, которые прекрасно подходят для применения в облачных средах, — это фундаментальное требование. *Облачный* (cloud-native) — термин, используемый для описания принципов, паттернов и инструментов автоматизации масштабирования контейнеризованных приложений. Мы будем использовать слова *облачный* и *Kubernetes* как взаимозаменяемые; последнее является названием наиболее популярной в настоящее время облачной платформы с открытым исходным кодом.

В этой книге мы не рассматриваем разработку чистого кода, предметно-ориентированное проектирование, гексагональную архитектуру или создание микросервисов. Все внимание сосредоточено исключительно на шаблонах и методах решения задач оркестрации контейнеров. Но чтобы эти паттерны были эффективными, приложение должно быть спроектировано с применением методов разработки чистого кода, предметно-ориентированного проектирования, изоляции внешних зависимостей подобно гексагональной архитектуре, создания микросервисов и других подобных методик.

Распределенные примитивы

Чтобы объяснить смысл новых абстракций и примитивов, сравним их с известными понятиями объектно-ориентированного программирования (ООП), например, на языке Java. В ООП используются такие понятия, как класс, объект, пакет,

наследование, инкапсуляция и полиморфизм. Среда выполнения Java предоставляет конкретные функции и гарантии в отношении управления жизненным циклом объектов и приложения в целом.

Язык Java и виртуальная машина Java (Java Virtual Machine, JVM) предоставляют локальные, внутрипроцессные строительные блоки для создания приложений. Kubernetes добавляет в эту привычную картину совершенно новое измерение, предлагая новый набор примитивов и среду выполнения для создания систем, распределенных по нескольким узлам и процессам. Используя Kubernetes, мы больше не полагаемся только на локальные примитивы для реализации поведения приложения в целом.

Нам все еще требуются объектно-ориентированные строительные блоки для создания компонентов распределенного приложения, но дополнительно можно использовать примитивы Kubernetes для организации поведения приложения. В табл. 1.1 перечислены понятия из области разработки приложений и соответствующие им локальные и распределенные примитивы в JVM и Kubernetes.

Таблица 1.1. Локальные и распределенные примитивы

Понятие	Локальный примитив	Распределенный примитив
Инкапсуляция поведения	Класс	Образ контейнера
Экземпляр поведения	Объект	Контейнер
Единица повторного использования	*.jar*	Образ контейнера
Композиция	Класс A содержит класс B	Паттерн Sidecar
Наследование	Класс A расширяет класс B	Родительский образ FROM контейнера
Единица развертывания	*.jar/.war/.ear*	Под (Pod)
Изоляция времени сборки/выполнения	Модуль, Пакет, Класс	Пространство имен, под, контейнер
Начальная инициализация	Конструктор	Инициализирующие контейнеры, или Init-контейнеры
Операции, следующие сразу за начальной инициализацией	Init-метод	`postStart`
Операции, непосредственно предшествующие уничтожению экземпляра	Destroy-метод	`preStop`

Таблица 1.1 (*окончание*)

Понятие	Локальный примитив	Распределенный примитив
Процедура очистки ресурсов	`finalize()`, обработчик события завершения	—
Асинхронное и параллельное выполнение	`ThreadPoolExecutor`, `ForkJoinPool`	Job (задание)
Периодическое выполнение	`Timer`, `ScheduledExecutorService`	CronJob (планировщик заданий)
Фоновое выполнение	Фоновые потоки выполнения	DaemonSet (контроллер набора демонов)
Управление конфигурацией	`System.getenv()`, `Properties`	ConfigMap (карта конфигураций), Secret (секрет)

Внутрипроцессные и распределенные примитивы имеют общие черты, но их нельзя сравнивать непосредственно, и они не взаимозаменяемы. Они работают на разных уровнях абстракции и имеют разные предпосылки и гарантии. Некоторые примитивы должны использоваться вместе, например, следует применять классы для создания объектов и помещать их в образы контейнеров. Однако другие примитивы Kubernetes могут служить полноценной заменой аналогам в Java, например, CronJob в Kubernetes может полностью заменить `ExecutorService` в Java.

А теперь рассмотрим несколько распределенных абстракций и примитивов из Kubernetes, которые особенно интересны для разработчиков приложений.

Контейнеры

Контейнеры — это строительные блоки для создания облачных приложений на основе Kubernetes. Проводя аналогию с ООП и Java, образы контейнеров можно сравнить с классами, а сами контейнеры — с объектами. Подобно классам, которые можно расширять (наследовать) и таким способом изменять их поведение, можно создавать образы контейнеров, которые расширяют (наследуют) другие образы контейнеров, и также изменять их поведение. Подобно объектам, которые можно объединять, можно объединять контейнеры, помещая их в поды (Pod), и использовать результаты их взаимодействий.

Продолжая сравнение, можно сказать, что Kubernetes напоминает виртуальную машину Java, но разбросанную по нескольким хостам и отвечающую за запуск контейнеров и управление ими. Init-контейнеры можно сравнить с конструкторами объектов, а контроллеры DaemonSet похожи на потоки выполнения, действующие в фоновом режиме (как, например, сборщик мусора в Java). Поды можно считать аналогами контекста инверсии управления (Inversion of Control, IoC), используемого, например, в Spring Framework, где несколько объектов

имеют общий управляемый жизненный цикл и могут напрямую обращаться друг к другу.

Дальнейшие параллели возможны, но едва ли уместны. Следует отметить, что контейнеры играют основополагающую роль в Kubernetes, а создание модульных многоразовых специализированных образов контейнеров является основой успеха любого проекта и экосистемы контейнеров в целом. Что еще можно сказать о контейнерах и их назначении в контексте распределенного приложения помимо перечисления технических характеристик образов контейнеров, которые обеспечивают упаковку и изоляцию? Вот несколько основных особенностей контейнеров:

- Образ контейнера — это функциональная единица, решающая одну определенную задачу.
- Образ контейнера принадлежит одной команде и имеет свой цикл выпуска новых версий.
- Образ контейнера является самодостаточным — он определяет и несет в себе зависимости времени выполнения.
- Образ контейнера является неизменным — после создания он не изменяется, но может настраиваться.
- Образ контейнера определяет требования к ресурсам и внешние зависимости.
- Образ контейнера имеет четко определенные API для доступа к его функциональности.
- Контейнер обычно выполняется как отдельный процесс Unix.
- Контейнер может быть уничтожен и может безопасно масштабироваться в обоих направлениях в любой момент.

Кроме перечисленных характеристик, правильный образ контейнера должен иметь модульную организацию, поддерживать параметризацию и повторное использование в средах, в которых он будет работать. Небольшие, модульные и пригодные к повторному использованию образы контейнеров помогают создавать более специализированные и надежные образы подобно большой библиотеке в языках программирования.

Поды

Контейнеры идеально подходят для реализации принципов микросервисов. Образ контейнера предоставляет единую функциональную единицу, принадлежит одной команде, имеет независимый цикл выпуска новых версий и обеспечивает развертывание и изоляцию среды выполнения. В большинстве случаев один микросервис соответствует одному образу контейнера.

Однако многие облачные платформы предлагают специальный примитив для управления жизненным циклом группы контейнеров, в Kubernetes он называется

подом. *Под* (*Pod*[1]) — это единица планирования, развертывания и изоляции среды выполнения для группы контейнеров. Контейнеры, входящие в под, всегда распределяются на один хост, вместе развертываются и масштабируются, а также могут совместно использовать файловую систему, сеть и пространства имен процесса. Единый жизненный цикл позволяет контейнерам в поде взаимодействовать друг с другом через файловую систему или через сеть локального узла (localhost) либо использовать межпроцессные коммуникации хоста, если это необходимо (например, по соображениям производительности). Под также определяет общие границы безопасности приложения. Хотя можно настроить различающиеся параметры безопасности для контейнеров одного пода, обычно все контейнеры в группе имеют один и тот же уровень доступа, сегментации сети и идентификацию.

Как показано на рис. 1.2, на этапах разработки и сборки микросервис соответствует образу контейнера, который выпускает одна команда. Но во время выполнения микросервис представлен подом, который является единицей развертывания, размещения и масштабирования. Единственный способ запустить контейнер и для масштабирования, и для миграции — использовать абстракцию пода. Иногда под содержит несколько контейнеров. В данном конкретном примере контейнеризированный микросервис во время выполнения использует вспомогательный контейнер, как показано в главе 16 «Паттерн Sidecar».

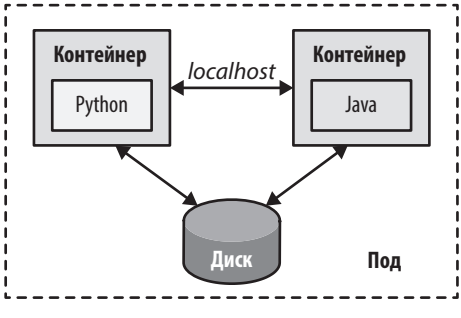

Рис. 1.2. Под как единица развертывания и управления

Уникальные свойства контейнеров и подов обеспечивают новый набор шаблонов и подходов для разработки приложений на основе микросервисов. Выше мы рассмотрели характеристики хорошо спроектированных контейнеров, давайте отметим и некоторые характеристики пода:

- Под — элементарная единица планирования. Собираясь запустить его, планировщик ищет хост, который удовлетворяет требованиям всех контейнеров, входящих в группу (особенности Init-контейнеров с этой точки зрения мы рассмотрим в главе 15 «Init-контейнеры»). Для пода со множеством контейнеров

[1] В переводе с английского pod — это стручок, кокон или группа. В контексте Kubernetes Pod — это группа контейнеров, запускаемых как одно целое. — *Примеч. пер.*

планировщику придется отыскать хост, обладающий ресурсами для удовлетворения суммарных требований всех контейнеров. Такой процесс планирования описан в главе 6 «Автоматическое размещение».

- Под гарантирует совместное размещение контейнеров. Благодаря этому контейнеры в одной группе получают дополнительные возможности взаимодействия друг с другом. Например, часто используются общая локальная файловая система, сетевой интерфейс localhost и локальные механизмы межпроцессных взаимодействий (IPC) для высокопроизводительных взаимодействий.
- Под имеет IP-адрес, имя и диапазон портов, общих для контейнеров, входящих в группу. Поэтому контейнеры в одном поде необходимо тщательно настраивать, чтобы избежать конфликтов портов, точно так же, как при совместном использовании сетевого пространства хоста параллельно выполняющимися процессами Unix.

Под — это неделимый элемент Kubernetes, в котором находится приложение, но работать с подами напрямую категорически не рекомендуется — используйте для этого Services (сервисы).

Сервисы

Поды — эфемерные образования, которые могут появляться и исчезать по разным причинам, например в ходе масштабирования, в случае неудачи при проверке работоспособности контейнеров или при миграции узлов. IP-адрес пода становится известен только после того, как он запланирован и запущен на узле. Под может быть перепланирован для запуска на другом узле, если узел, на котором он выполнялся, прекратил работу. Это означает, что сетевой адрес группы контейнеров может меняться в течение жизни приложения, а потому необходим примитив для обнаружения и балансировки нагрузки.

Роль этого примитива играют сервисы (Services) Kubernetes. Это простая, но мощная абстракция, которая присваивает имени сервиса постоянный IP-адрес и номер порта. То есть сервис — это именованная точка входа для доступа к приложению. В наиболее распространенном сценарии сервис играет роль точки входа для набора групп подов, но это не всегда так. Этот универсальный примитив может также служить точкой входа для доступа к функциональным возможностям за пределами кластера Kubernetes. Соответственно, сервисы можно использовать для обнаружения сервисов и распределения нагрузки, а также для замены реализации и масштабирования без влияния на потребителей сервиса. Подробнее о сервисах мы поговорим в главе 13 «Обнаружение сервисов».

Метки

Как было показано выше, на этапе сборки аналогом микросервиса является образ контейнера, а на этапе выполнения — под. А что можно считать аналогом приложения, состоящего из нескольких микросервисов? Kubernetes предлагает еще

два примитива, помогающих провести аналогию с понятием приложения: метки и пространства имен.

До появления микросервисов понятию приложения соответствовала одна единица развертывания с единой схемой управления версиями и циклом их выпуска. Приложение помещалось в один файл *.war*, *.ear* или другого формата. Но затем приложение стало делиться на микросервисы, которые разрабатываются, выпускаются, запускаются, перезапускаются и масштабируются независимо друг от друга. С появлением микросервисов понятие приложения стало более размытым — теперь нет ключевых артефактов или действий, которые должны выполняться на уровне приложения. Однако если нужно указать, что некоторые независимые сервисы принадлежат одному приложению, можно использовать *метки* (labels). Давайте представим, что одно монолитное приложение мы разделили на три микросервиса, а другое — на два.

В этом случае мы получаем пять определений подов (и, возможно, большее количество их экземпляров), которые не зависят друг от друга с точки зрения разработки и времени выполнения. Тем не менее нам все еще может потребоваться указать, что первые три пода представляют одно приложение, а два следующих — другое. Поды могут быть независимыми и представлять определенную ценность для бизнеса по отдельности, а могут зависеть друг от друга. Например, один под может содержать контейнеры, отвечающие за пользовательский интерфейс, а два других — за реализацию функциональности бэкенда. Если какой-то из подов прекратит работать, приложение окажется бесполезным с точки зрения бизнеса. Метки позволяют определить набор подов и управлять им как одной логической единицей. На рис. 1.3 показано, как можно использовать метки для группировки частей распределенного приложения в подсистемы.

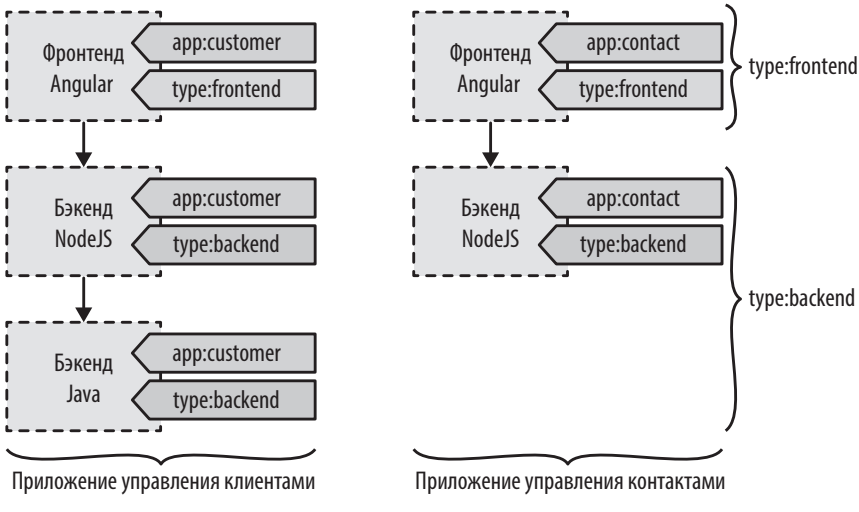

Рис. 1.3. Метки идентифицируют принадлежность подов приложению

Вот несколько примеров использования меток:

- Метки используются наборами реплик (ReplicaSet) для поддержания в рабочем состоянии определенных экземпляров подов. Для этого каждое определение пода должно иметь уникальную комбинацию меток, используемых для планирования.
- Метки широко используются планировщиком для совместного размещения или распределения групп подов по узлам с учетом требований этих подов.
- Метка может задавать логическую группировку набора подов и идентифицировать его как приложение.
- Метки можно использовать для хранения метаданных. Трудно заранее предусмотреть все случаи, но лучше иметь достаточно меток, чтобы описать все важные характеристики подов. Например, метки могут пригодиться для описания логических групп внутри приложения, бизнес-характеристик и степени важности, конкретных зависимостей среды выполнения, таких как архитектура оборудования или настройки местоположения.

Впоследствии метки могут использоваться планировщиком для более точного планирования. Те же метки можно использовать из командной строки для управления соответствующими подами. Однако не следует увлекаться и заранее добавлять слишком много меток. При необходимости требуемые метки всегда можно добавить позже. Удалять метки, кажущиеся ненужными, намного опаснее, поскольку сложно узнать, для чего они используются и какой эффект может вызвать их удаление.

Аннотации

Другой примитив — *аннотации* (annotations) — очень похож на метки. Подобно меткам, аннотации организованы в виде ассоциативного массива, но предназначены для определения метаданных, которые используются компьютером, а не человеком.

Информация в аннотациях не предназначена для использования в запросах и для сопоставления объектов. Аннотации предназначены для присоединения дополнительных метаданных к объектам из инструментов и библиотек, которые планируется использовать. Например, аннотации можно использовать для добавления номера версии и сборки, информации об образе, меток времени, имен веток в репозитории Git, номеров пулреквестов, хешей образов изображений, адресов в реестре, имен авторов, сведений об инструментах и многого другого. То есть метки используются главным образом для поиска и выполнения действий с соответствующими ресурсами, а аннотации — для прикрепления метаданных, которые могут использоваться компьютером.