Отзывы о книге «Облачный Go»

Автор книги проделал большую работу, подробно описав высокоуровневую концепцию «облачных приложений» и приемы ее реализации с использованием современного языка программирования Go. В результате получилась захватывающая и вдохновляющая книга.

– *Ли Атчисон* (*Lee Atchison*)
Владелец Atchison Technology LLC

Это первая книга, из встречавшихся мне, которая с такой широтой и глубиной освещает современные приемы реализации облачных вычислений. Представленные здесь шаблоны сопровождаются наглядными примерами решения реальных задач, с которыми инженеры сталкиваются ежедневно.

– *Альваро Атьенза* (*Alvaro Atienza*)
Инженер по надежности, Flatiron Health

На страницах этой книги ясно (и с юмором) отражен богатый опыт Мэтта в искусстве и науке построения надежных систем в принципиально ненадежном мире. Присоединяйтесь к нему, и он познакомит вас с фундаментальными строительными блоками и приемами конструирования систем, позволяющими создавать масштабные и надежные системы из эфемерных и ненадежных компонентов современной облачной инфраструктуры.

– *Дэвид Никпонски* (*David Nicponski*)
Главный инженер, Robinhood

За последние несколько лет наметились две важные тенденции: язык Go все чаще используется для разработки не только серверных компонентов, но и инфраструктуры; а инфраструктура перемещается в облако. В этой книге кратко описывается современное состояние сочетания этих двух факторов.

– *Натали Пистунович* (*Natalie Pistunovich*)
Ведущий пропагандист передовых методов разработки, Aerospike

Я начал читать эту книгу, почти ничего не зная о Go, и закончил, чувствуя себя экспертом. Я бы даже сказал, что, просто прочитав эту книгу, я стал намного более опытным инженером.

– *Джеймс Куигли* (*James Quigley*)
Инженер по надежности систем, Bloomberg

Содержание

От издательства 17

Об авторе 18

Об иллюстрации на обложке 19

Предисловие 20

Часть I. ОБЛАЧНОЕ ОКРУЖЕНИЕ 24

Глава 1. Что такое «облачное» приложение? 25

История развития до настоящего времени 26
Что значит быть «облачным»? 28
  Масштабируемость 28
  Слабая связанность 29
  Устойчивость 30
  Управляемость 32
  Наблюдаемость 33
Что особенного в облачном окружении? 34
Итоги 35

Глава 2. Почему Go правит облачным миром 36

Как появился Go 36
Особенности облачного мира 37
  Композиция и структурная типизация 37
  Понятность 39
  Модель взаимодействия последовательных процессов 40
  Быстрая сборка 41
  Стабильность языка 42
  Безопасность памяти 42
  Производительность 43
  Статическая компоновка 44
  Статическая типизация 45
Итоги 46

Часть II. ОБЛАЧНЫЕ КОНСТРУКЦИИ В GO 47

Глава 3. Основы языка Go 48

Базовые типы данных 48
  Логические значения 49

Простые числа 49
Комплексные числа 50
Строки 50
Переменные...51
Сокращенная форма объявления переменных.......................................51
Нулевые значения..52
Пустой идентификатор...53
Константы 54
Контейнеры: массивы, срезы и ассоциативные массивы.............................54
Массивы..55
Срезы 55
Работа со срезами 56
Оператор извлечения среза 58
Строки и срезы...58
Ассоциативные массивы 59
Проверка наличия в ассоциативном массиве 60
Указатели...61
Управляющие структуры 62
Забавный цикл for...63
Универсальная инструкция for 63
Обход в цикле элементов массивов и срезов 64
Обход в цикле элементов ассоциативных массивов.............................65
Инструкция if..65
Инструкция switch 66
Обработка ошибок 67
Создание ошибки...68
Необычные особенности функций: переменное число параметров
и замыкания 69
Функции 69
Несколько возвращаемых значений 69
Рекурсия..70
Отложенные вычисления 70
Указатели как параметры 72
Функции с переменным числом аргументов 73
Передача срезов в параметре с переменным числом значений............74
Анонимные функции и замыкания 74
Структуры, методы и интерфейсы 75
Структуры 76
Методы 77
Интерфейсы..78
Проверка типа..79
Пустой интерфейс 79
Композиция путем встраивания типов..80
Встраивание интерфейсов 80
Встраивание структур...81
Продвижение..81
Прямой доступ к встроенным полям 81
Самое интересное: конкуренция...82

Сопрограммы ... 82
Каналы ... 83
Блокировка канала ... 83
Буферизация каналов ... 84
Закрытие каналов ... 84
Прием значений из канала в цикле ... 85
select ... 85
Реализация тайм-аутов для каналов ... 86
Итоги ... 87

Глава 4. Шаблоны программирования облачных приложений ... 88
Пакет context ... 89
Что может дать контекст ... 90
Создание контекста ... 91
Определение крайних сроков и тайм-аутов контекста ... 91
Определение значений в контексте запроса ... 92
Использование контекста ... 92
Структура этой главы ... 93
Шаблоны стабильности ... 94
Circuit Breaker (Размыкатель цепи) ... 94
Применимость ... 94
Реализация ... 95
Пример кода ... 96
Debounce (Антидребезг) ... 97
Применимость ... 97
Компоненты ... 98
Реализация ... 98
Пример кода ... 99
Retry (Повтор) ... 101
Применимость ... 101
Компоненты ... 102
Реализация ... 102
Пример кода ... 102
Throttle (Дроссельная заслонка) ... 104
Применимость ... 104
Компоненты ... 105
Реализация ... 105
Пример кода ... 106
Timeout (Тайм-аут) ... 107
Применимость ... 107
Компоненты ... 107
Реализация ... 108
Пример кода ... 108
Шаблоны конкуренции ... 110
Fan-In (Мультиплексор) ... 110
Применимость ... 110
Компоненты ... 110
Реализация ... 110

Пример кода ... 111
Fan-Out (Демультиплексор) ... 112
Применимость ... 112
Компоненты ... 112
Реализация ... 113
Пример кода ... 113
Future (В будущем) ... 114
Применимость ... 115
Компоненты ... 116
Реализация ... 116
Пример кода ... 116
Sharding (Сегментирование) ... 118
Применимость ... 118
Компоненты ... 119
Реализация ... 119
Пример кода ... 121
Итоги ... 124

Глава 5. Конструирование облачной службы ... 125
Давайте создадим службу! ... 125
Что такое хранилище пар ключ/значение? ... 126
Требования ... 126
Что такое идемпотентность, и почему это важно? ... 126
Конечная цель ... 128
Итерация 0: базовая функциональность ... 128
Наш суперпростой API ... 129
Итерация 1: монолит ... 130
Создание HTTP-сервера с использованием net/http ... 131
Создание HTTP-сервера с использованием gorilla/mux ... 132
Создание минимальной службы ... 133
Инициализация проекта с помощью модулей Go ... 133
Переменные в путях URI ... 134
Множество сопоставлений ... 135
Создание службы RESTful ... 135
Методы RESTful ... 136
Реализация функции создания ... 136
Реализация функции чтения ... 138
Добавление в структуру данных поддержки использования в конкурентном окружении ... 140
Интеграция мьютекса чтения/записи в приложение ... 141
Итерация 2: долговременное хранение ресурса ... 142
Что такое журнал транзакций? ... 143
Формат журнала транзакций ... 143
Интерфейс регистратора транзакций ... 144
Сохранение состояния в журнале транзакций ... 144
Создание прототипа регистратора транзакций ... 145
Определение типа события ... 146

Реализация FileTransactionLogger ... 148
Создание экземпляра FileTransactionLogger ... 149
Добавление записей в конец журнала транзакций ... 150
Использование bufio.Scanner для воспроизведения транзакций из журнала ... 151
Интерфейс регистратора транзакций (еще раз) ... 153
Инициализация FileTransactionLogger в веб-службе ... 153
Интеграция FileTransactionLogger в веб-службу ... 155
Будущие улучшения ... 155
Сохранение состояния во внешней базе данных ... 155
Работа с базами данных в Go ... 156
Импортирование драйвера базы данных ... 157
Реализация PostgresTransactionLogger ... 157
Создание экземпляра PostgresTransactionLogger ... 158
Выполнение SQL-запроса INSERT с помощью db.Exec ... 160
Использование db.Query для воспроизведения транзакций из журнала ... 161
Инициализация PostgresTransactionLogger в веб-службе ... 162
Будущие улучшения ... 163
Итерация 3: реализация безопасности транспортного уровня ... 163
Transport Layer Security ... 164
Сертификаты, центры сертификации и доверие ... 164
Закрытый ключ и файлы сертификатов ... 165
Формат Privacy Enhanced Mail (PEM) ... 165
Защита веб-службы с помощью HTTPS ... 166
В заключение о транспортном уровне ... 167
Контейнеризация хранилища пар ключ/значение ... 168
Основы Docker ... 169
Dockerfile ... 169
Сборка образа контейнера ... 170
Запуск образа контейнера ... 171
Проверка запущенного образа контейнера ... 172
Отправка запроса в опубликованный порт контейнера ... 173
Запуск нескольких контейнеров ... 174
Остановка и удаление контейнеров ... 174
Сборка контейнера для службы хранилища пар ключ/значение ... 175
Итерация 1: добавление двоичного файла в пустой образ ... 176
Итерация 2: многоэтапная сборка ... 178
Сохранение данных контейнера вовне ... 179
Итоги ... 180

Часть III. ОБЛАЧНЫЕ АТРИБУТЫ ... 182

Глава 6. Все дело в надежности ... 183

В чем суть облачных вычислений? ... 184
Все дело в надежности ... 184

Что такое надежность, и почему она так важна? 185
Надежность обеспечивается не только операторами 187
Достижение надежности 188
Предотвращение неисправностей 190
Рекомендуемые практики программирования 190
Особенности языка 190
Масштабируемость 191
Слабая связанность 191
Отказоустойчивость 192
Устранение неисправностей 192
Проверка и тестирование 193
Управляемость 194
Прогнозирование неисправностей 194
Непреходящая актуальность методологии «Двенадцать факторов» 194
I. Кодовая база 195
II. Зависимости 196
III. Конфигурация 196
IV. Сторонние службы 198
V. Сборка, выпуск, выполнение 199
VI. Процессы 200
VII. Изоляция данных 200
VIII. Масштабируемость 201
IX. Живучесть 202
X. Сходство окружений разработки/эксплуатации 202
XI. Журналирование 203
XII. Задачи администрирования 204
Итоги 205

Глава 7. Масштабируемость 206
Что такое масштабируемость? 207
Различные формы масштабирования 208
Четыре основных узких места 209
С состоянием и без состояния 211
Состояние приложения и состояние ресурса 211
Преимущества отсутствия состояния 212
Отложенное масштабирование: эффективность 213
Эффективное кеширование с использованием кеша LRU 213
Эффективная синхронизация 217
Разделяйте память, общаясь 217
Уменьшение простоев на блокировках с помощью буферизованных каналов 219
Уменьшение простоев на блокировках с помощью сегментирования ... 221
Утечки памяти могут вызвать… фатальную ошибку исчерпания памяти во время выполнения 222
Утечки сопрограмм 222
Вечно тикающие таймеры 223
В заключение об эффективности 225

Архитектуры служб ... 225
Архитектура монолитной системы ... 226
Архитектура системы микросервисов ... 227
Бессерверные архитектуры ... 229
Достоинства и недостатки бессерверных вычислений ... 229
Бессерверные службы ... 231
Итоги ... 233

Глава 8. Слабая связанность ... 234
Тесная связанность ... 235
Множество форм тесной связанности ... 236
Хрупкие протоколы обмена ... 236
Общие зависимости ... 237
Общий момент времени ... 237
Фиксированные адреса ... 238
Взаимодействия между службами ... 238
Шаблон обмена сообщениями запрос/ответ ... 239
Распространенные реализации шаблона запрос/ответ ... 240
Отправка HTTP-запросов с использованием net/http ... 240
Вызов удаленных процедур с использованием gRPC ... 244
Определение интерфейса с использованием протокола буферов ... 245
Установка компилятора протокола буферов ... 246
Определение структуры сообщения ... 247
Структура сообщений для взаимодействий с хранилищем пар ключ/значение ... 248
Определение методов службы ... 249
Компиляция протокола буферов ... 250
Реализация службы gRPC ... 251
Реализация клиента gRPC ... 253
Слабое связывание локальных ресурсов с помощью плагинов ... 255
Подключение плагинов с помощью пакета plugin ... 255
Словарь плагинов ... 256
Пример плагина ... 257
Интерфейс Sayer ... 257
Код плагина ... 258
Сборка плагинов ... 258
Использование плагинов Go ... 259
Запуск примера ... 261
Система плагинов HashiCorp для Go, доступных через RPC ... 261
Еще один пример плагина ... 262
Общий код ... 263
Реализация плагина ... 265
Процесс-потребитель ... 266
Гексагональная архитектура ... 269
Архитектура ... 269
Реализация гексагональной службы ... 270
Реорганизация компонентов ... 271

Наш первый разъем 272
Основное приложение 272
Адаптеры TransactionLogger 273
Порт FrontEnd 274
Все вместе 276
Итоги 277

Глава 9. Устойчивость 279

Почему устойчивость важна 280
Что подразумевается под сбоем системы? 281
Обеспечение устойчивости 282
Каскадные сбои 282
Предотвращение перегрузки 284
Дросселирование 284
Сброс нагрузки 288
Постепенное ухудшение качества обслуживания 289
Повтори еще раз: повторные запросы 289
Алгоритмы увеличения задержки 291
Размыкание цепи 294
Тайм-ауты 295
Использование контекста Context для реализации тайм-аутов на стороне службы 296
Прерывание ожидания обработки клиентских запросов HTTP/REST 298
Прерывание ожидания обработки клиентских запросов gRPC 299
Идемпотентность 301
Как сделать службу идемпотентной? 302
А как насчет скалярных операций? 303
Избыточность служб 304
Проектирование избыточности 305
Автоматическое масштабирование 307
Проверка работоспособности 308
Что подразумевается под «работоспособностью» экземпляра? 309
Три типа проверок работоспособности 309
Проверка жизнеспособности 310
Поверхностная проверка работоспособности 310
Глубокая проверка работоспособности 312
Открытие при отказе 313
Итоги 314

Глава 10. Управляемость 315

Что такое управляемость, и почему она важна? 316
Настройка приложения 317
Рекомендуемые приемы организации конфигураций 318
Настройка с использованием переменных окружения 319
Настройка с использованием аргументов командной строки 320
Стандартный пакет flag 320

Парсер командной строки Cobra 322
Настройка с использованием файлов 326
Наша структура конфигурационных данных 326
Формат JSON 327
Формат YAML 332
Наблюдение за изменениями в конфигурационных файлах 335
Viper: швейцарский армейский нож конфигурационных пакетов 340
Явно устанавливаемые значения в Viper 341
Работа с флагами командной строки в Viper 341
Работа с переменными окружения в Viper 342
Работа с конфигурационными файлами в Viper 342
Использование удаленных хранилищ пар ключ/значение в Viper 344
Значения по умолчанию в Viper 345
Управление функциональными возможностями с помощью флагов 345
Разработка флага для управления функциональной возможностью 346
Итерация 0: начальная реализация 347
Итерация 1: жестко запрограммированный флаг 347
Итерация 2: настраиваемый флаг 348
Итерация 3: динамический флаг 349
Динамические флаги как функции 350
Реализация функции динамического флага 350
Поиск функции флага 351
Функция маршрутизации 352
Итоги 353

Глава 11. Наблюдаемость 354

Что такое наблюдаемость? 355
Зачем нужна наблюдаемость? 355
Чем наблюдаемость отличается от «традиционного» мониторинга? 356
«Три столпа наблюдаемости» 357
OpenTelemetry 358
Компоненты OpenTelemetry 359
Трассировка 360
Концепции трассировки 361
Трассировка с использованием OpenTelemetry 362
Создание экспортеров трассировки 364
Создание провайдера трассировки 366
Настройка глобального провайдера трассировки 367
Получение экземпляра трассировщика 367
Начальная и конечная операции 367
Установка метаданных операции 369
Автоматическое инструментирование 370
Собираем все вместе: трассировка 373
API-службы вычисления чисел Фибоначчи 374
Функция-обработчик службы вычисления чисел Фибоначчи 375
Функция main службы 376
Запуск служб 377

Вывод консольного экспортера 377
Просмотр результатов в Jaeger 378
Метрики 379
Два способа передачи метрик: принудительная и по запросу 381
Принудительная отправка метрик 382
Передача метрик по запросу 382
Какой подход лучше? 383
Метрики в OpenTelemetry 384
Создание экспортеров метрик 385
Установка глобального провайдера метрик 386
Экспортирование конечной точки метрик 386
Получение экземпляра Meter 388
Инструменты метрик 388
Собираем все вместе: метрики 394
Запуск служб 394
Вывод конечной точки метрик 395
Просмотр результатов в Prometheus 396
Журналирование 397
Рекомендуемые методы журналирования 397
Интерпретируйте журналы как потоки событий 398
Структурируйте события для последующего анализа 398
Лучше меньше, да лучше 400
Динамически фильтруйте журналируемые данные 400
Журналирование с использованием стандартного пакета log 401
Специальные функции журналирования 402
Журналирование в нестандартный объект записи 402
Флаги журналирования 403
Пакет журналирования Zap 403
Создание регистратора Zap 405
Журналирование с использованием Zap 405
Динамическая фильтрация журналируемых данных в Zap 407
Итоги 409

Предметный указатель 410

Об авторе

Мэтью А. Титмус – ветеран индустрии разработки программного обеспечения. Научившись создавать виртуальные миры в LPC, он получил удивительно востребованное образование в области молекулярной биологии, создал инструменты анализа терабайтных наборов данных для лаборатории физики высоких энергий, с нуля написал фреймворк для разработки веб-приложений, применил методы распределенных вычислений для анализа ракового генома, а также в числе первых разрабатывал методы машинного обучения на связанных данных.

Он был одним из первых сторонников облачных технологий в целом и языка Go в частности. Последние четыре года специализируется на переносе монолитных приложений в контейнерный облачный мир, помогая компаниям осваивать новые способы разработки, развертывания и управления своими службами. Он увлечен решением задач повышения качества промышленных систем и потратил много времени на обдумывание и реализацию стратегий наблюдения за распределенными системами и управления ими.

Мэтью живет на Лонг-Айленде с самой терпеливой женщиной в мире, на которой ему посчастливилось жениться, и самым очаровательным мальчиком в мире, от которого ему посчастливилось услышать «папа».

Предисловие

Это волшебное время для инженеров

У нас есть Docker для создания контейнеров и Kubernetes для управления ими. Prometheus помогает нам следить за ними. Consul позволяет обнаруживать их. Jaeger дает возможность организовать взаимодействия между ними. Это лишь несколько примеров, в действительности круг возможностей гораздо шире, и все эти возможности поддерживают новое поколение технологий: все они «облачные», и все они написаны на Go.

Термин «облачный» кажется двусмысленным и отдает рекламной шумихой, но на самом деле он имеет довольно конкретное определение. Согласно Cloud Native Computing Foundation, подразделению известного фонда Linux Foundation, облачное приложение – это приложение, способное масштабироваться синхронно с изменением нагрузки, устойчивое к неопределенности окружения и управляемое в условиях нестабильности и постоянно меняющихся требований. Иначе говоря, облачные приложения создаются для работы в жесткой и неопределенной вселенной.

На основе опыта, накопленного за годы разработки облачного программного обеспечения, около десяти лет назад был создан Go – первый язык программирования, спроектированный специально для разработки облачных приложений. Во многом его появление было обусловлено тем, что типичные серверные языки, использовавшиеся в то время, просто не подходили для создания распределенных приложений, которые производит Google.

С тех пор Go занял лидирующие позиции в облачной разработке и используется повсюду: от Docker до Harbour, от Kubernetes до Consul, от InfluxDB до CockroachDB. Десять из пятнадцати сертифицированных проектов Cloud Native Computing Foundation и 42 из 62[1] его проектов в целом написаны в основном или полностью на Go. И с каждым днем их становится все больше.

Кому адресована эта книга

Эта книга адресована опытным разработчикам, особенно инженерам веб-приложений и инженерам по надежности. Многие из них уже использовали Go для создания веб-сервисов, но не знали некоторых тонкостей разработки в облачных окружениях или не имели четкого представления о том, что такое «облачные приложения», и впоследствии обнаруживали, что их сервисы сложно развертывать, ими сложно управлять или наблюдать за ними. Таким

[1] Включая проекты CNCF из категорий Sandbox, Incubating и Graduated, по состоянию на февраль 2021 года.

читателям эта книга не только поможет заложить прочный фундамент для создания собственных облачных сервисов, но также покажет, в чем преимущества этих методов, и представит конкретные примеры, способствующие пониманию этой довольно абстрактной темы.

Предполагается, что многие читатели хорошо знакомы с другими языками программирования, но их привлекает репутация Go как языка облачной разработки. Таким читателям эта книга предложит передовой опыт использования Go в качестве специализированного языка разработки для облачных окружений и поможет им решить собственные проблемы управления и развертывания облачных приложений.

Почему я написал эту книгу

Способы проектирования, конструирования и развертывания приложений меняются со временем. Требования к масштабированию вынуждают разработчиков размещать свои сервисы на десятках и сотнях серверов: отрасль постепенно становится «облачной». Но при этом возникает множество новых проблем: как разрабатывать, развертывать или управлять сервисом, действующим на десятках, сотнях или даже тысячах серверов? К сожалению, существующие книги об облачных вычислениях сосредоточены на абстрактных принципах проектирования и содержат лишь элементарные примеры, если вообще содержат. Эта книга призвана удовлетворить потребность в демонстрации практической реализации сложных принципов проектирования облачных вычислений.

Соглашения

В этой книге используются следующие соглашения по оформлению:

Курсив

Используется для обозначения новых терминов, имен файлов и расширений.

`Моноширинный шрифт`

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

`Моноширинный жирный`

Обозначает команды или другой текст, который должен вводиться пользователем.

`Моноширинный курсив`

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

Так выделяются советы и предложения.

Так обозначаются примечания общего характера.

Так обозначаются предупреждения и предостережения.

Использование программного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу: https://github.com/cloud-native-go/examples.

Данная книга призвана оказать вам помощь в решении ваших задач. В общем случае все примеры кода из этой книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения примеров из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «*Мэтью А. Титмус*. Облачный Go. М.: ДМК Пресс, 2021. 978-5-97060-965-1» или «*Cloud Native Go* by Matthew A. Titmus (O'Reilly). Copyright 2021 Matthew A. Titmus, 978-1-492-07633-9».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу permissions@oreilly.com.

Часть I

ОБЛАЧНОЕ ОКРУЖЕНИЕ

Глава 1

Что такое «облачное» приложение?

Самая опасная фраза в этом языке звучит так: «Мы всегда так поступали»[1].

– Грейс Хоппер (Grace Hopper), *Computerworld* (январь 1976)

Если вы читаете эту книгу, значит, вы, по крайней мере, слышали раньше термин *облачный*. Вероятно, вы даже читали некоторые из множества статей, написанных восторженными авторами, соблазнившимися хрустом купюр. Если ваше знание термина ограничивается только этим опытом, то вас можно простить за то, что сочли его неоднозначным, модным и просто еще одним из ряда рекламных выражений, которые могли начинаться как что-то полезное, но затем были использованы людьми, пытающимися что-то вам продать, как, например, «гибкая разработка» или «DevOps».

По схожим причинам поиск в интернете по запросу «определение термина облачный» может привести вас к мысли, что любое приложение, предназначенное для работы в облаке, должно быть написано на «правильном» языке[2] или с использованием «правильного» фреймворка или «правильной» технологии. Конечно, выбор языка может значительно упростить или усложнить вашу жизнь, но этот выбор не является ни необходимым, ни достаточным для создания облачного приложения.

Облачность зависит лишь от того, *где* запускается приложение. Термин *облачный* явно предполагает это. Все, что от вас требуется, – это «залить» свое старое, сляпанное кое-как приложение в контейнер и запустить его под управлением Kubernetes, после чего оно автоматически станет облачным, верно? Нет, потому что в этом случае вы лишь усложнили развертывание и управление приложением[3].

Итак, *что такое* облачное приложение? В этой главе мы ответим на этот вопрос. Для начала мы познакомимся с историей парадигм вычислительных служб до (и особенно) настоящего времени и обсудим, как неумолимое тре-

---

1 Surden, Esther. «Privacy Laws May Usher in Defensive DP: Hopper». *Computerworld*, 26 Jan. 1976, p. 9.

2 Это Go. Не поймите меня неправильно, но, в конце концов, эта книга о Go.

3 Вы когда-нибудь задумывались, почему так много миграций в Kubernetes терпят неудачу?

бование к масштабируемости стимулировало (и продолжает стимулировать) разработку и внедрение технологий, которые обеспечивают высокий уровень надежности. Наконец, мы определим конкретные атрибуты, характерные для таких приложений.

История развития до настоящего времени

История сетевых приложений – это история все усиливающегося требования масштабируемости.

В конце 1950-х появилась большая ЭВМ – мейнфрейм. В то время каждая программа и каждый фрагмент данных хранились в одной гигантской машине, к которой пользователи могли обращаться с помощью простых терминалов, не имевших собственных вычислительных возможностей. Вся логика и все данные жили вместе как одна большая счастливая семья. Это было простое время.

Все изменилось в 1980-х с появлением недорогих персональных компьютеров, подключаемых к сети. В отличие от простых терминалов, персональные компьютеры (ПК) могли выполнять некоторые вычисления самостоятельно, что позволяло переносить на них часть логики приложения. Эта новая многоуровневая архитектура, разделяющая логику представления, бизнес-логику и данные (рис. 1.1), сделала возможным изменение или замену компонентов сетевого приложения независимо друг от друга.

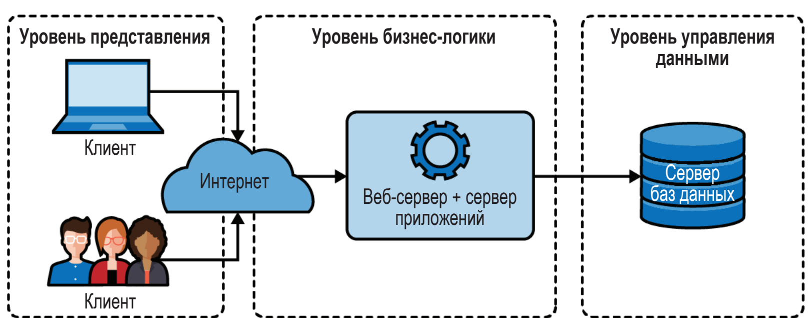


Рис. 1.1 ❖ Традиционная трехуровневая архитектура, в которой четко разделялись представление, бизнес-логика и данные

В 1990-х популяризация Всемирной паутины и последовавшая за ней «золотая лихорадка доткомов» породили технологию «программное обеспечение как услуга» (Software as a Service, SaaS). Целые отрасли были основаны на модели SaaS, что привело к созданию более сложных и ресурсоемких приложений, которые, в свою очередь, было труднее разрабатывать, поддерживать и развертывать. Внезапно классической многоуровневой архитектуры стало недостаточно. В результате бизнес-логика начала дробиться на подкомпоненты, которые можно было разрабатывать, поддерживать и развертывать независимо, и наступила эра микросервисов.

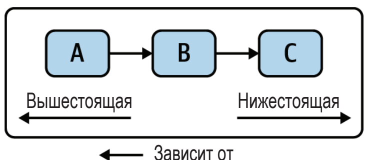
В 2006 году Amazon запустила облачную платформу Amazon Web Services (AWS), включавшую службу Elastic Compute Cloud (EC2). AWS была *не первым* предложением инфраструктуры как услуги (Infrastructure as a Service, IaaS), но именно она произвела революцию в отношении доступности хранилищ данных и вычислительных ресурсов, сделав облачные вычисления – и возможность быстрого масштабирования – доступными для масс, что ускорило массовую миграцию ресурсов в «облако».

К сожалению, организации вскоре поняли, что масштабирование – непростая задача. Проблемы неизбежны, и когда вы работаете с сотнями или тысячами ресурсов, проблемы возникают *очень часто*. Трафик может резко увеличиваться или уменьшаться, основное оборудование может выходить из строя, вышестоящие зависимости могут внезапно и необъяснимо стать недоступными. Но даже притом, что что-то может пойти не так, вам все равно придется развернуть все эти ресурсы и управлять ими. При таких масштабах люди не могут (или, по крайней мере, могут с большими затруднениями) справляться со всеми этими проблемами вручную.

Вышестоящие и нижестоящие зависимости

В этой книге я иногда буду использовать термины *вышестоящая зависимость* и *нижестоящая зависимость* для описания относительного положения двух ресурсов. В отрасли нет полного согласия относительно этих терминов, поэтому в данной книге я буду вкладывать в них следующий смысл.

Представьте, что у нас есть три службы: A, B и C, как показано на следующем рисунке:



В этом сценарии служба A посылает запросы службе B (и, следовательно, зависит от нее), которая, в свою очередь, зависит от службы C.

Поскольку служба B зависит от службы C, можно сказать, что служба C является *нижестоящей зависимостью* службы B. В более широком смысле, поскольку служба A зависит от службы B, которая зависит от службы C, служба C также является *транзитивной*[1] *нижестоящей зависимостью* службы A.

И наоборот, поскольку служба C используется службой B, можно сказать, что служба B является *вышестоящей зависимостью* службы C, а служба A – *транзитивной вышестоящей зависимостью* службы C.

[1] То есть непрямой, косвенной. – *Прим. перев.*

Что значит быть «облачным»?

Вообще говоря, по-настоящему облачное приложение включает в себя все, что мы узнали о масштабировании сетевых приложений за последние 60 лет. Они масштабируются в условиях резко меняющейся нагрузки, устойчивы в условиях неопределенности окружения и управляемы в условиях постоянно меняющихся требований. Иначе говоря, облачное приложение создано для жизни в жесткой и неопределенной вселенной.

Но как *определяется* термин *облачный*? К счастью для всех нас[1], в этом нет необходимости. Cloud Native Computing Foundation (https://oreil.ly/621yd) – подразделение известного фонда Linux Foundation и в некотором роде признанный авторитет в этой области – уже сделал это за нас:

> Облачные технологии позволяют организациям создавать и запускать масштабируемые приложения в современных динамических окружениях, таких как общедоступные, частные и гибридные облака…
>
> Они делают слабосвязанные системы устойчивыми, управляемыми и наблюдаемыми. В сочетании с надежной автоматизацией они позволяют инженерам часто и предсказуемо вносить важные изменения с минимальными усилиями[2].
>
> – Cloud Native Computing Foundation, CNCF Cloud Native Definition v1.0

Согласно этому определению, облачные приложения – больше, чем просто приложения, которые действуют в облаке. Они также *масштабируемы*, *слабо связаны*, *устойчивы*, *управляемы* и *доступны для наблюдения*. Можно сказать, что наличие этих «облачных атрибутов» позволяет называть системы облачными.

Как оказалось, каждый из этих атрибутов имеет довольно специфическое значение, поэтому рассмотрим их поближе.

Масштабируемость

В контексте облачных вычислений *масштабируемость* можно определить как способность показывать ожидаемое поведение в условиях значительных колебаний спроса вверх и вниз. Систему можно считать масштабируемой, если ее не нужно реорганизовывать для решения намеченной задачи во время или после резкого увеличения спроса.

Немасштабируемые службы могут отлично функционировать в начальных условиях, поэтому масштабируемость не всегда является первоочередной задачей при проектировании. Однако службы, не способные масштабироваться за пределы первоначальных ожиданий, имеют ограниченный срок жизни. Более того, реорганизовать службу для поддержки масштабируемости зачастую чрезвычайно сложно, поэтому проектирование с прицелом на

---

[1] И особенно для меня, пишущего эту классную книгу.

[2] Cloud Native Computing Foundation. «CNCF Cloud Native Definition v1.0», GitHub, 7 Dec. 2020. https://oreil.ly/KJuTr.

такую поддержку может сэкономить время и деньги в долгосрочной перспективе.

Существует два способа масштабирования, каждый из которых имеет свои плюсы и минусы.

Вертикальное масштабирование

Под *вертикальным масштабированием* подразумевается увеличение (или уменьшение) аппаратных ресурсов, доступных системе. Например, можно выделить дополнительную память или процессоры базе данных, которая работает на выделенном экземпляре. Преимущество вертикального масштабирования – в технической простоте реализации, но ресурсы любого конкретного экземпляра невозможно наращивать до бесконечности.

Горизонтальное масштабирование

Под *горизонтальным масштабированием* подразумевается увеличение (или уменьшение) количества действующих экземпляров службы. Например, можно увеличить количество узлов за балансировщиком нагрузки или контейнеров в Kubernetes либо в другой системе управления контейнерами. Эта стратегия имеет свои преимущества, включая избыточность и свободу от ограничений размеров экземпляров. Однако чем больше экземпляров, тем сложнее проектирование и управление, а кроме того, не все службы можно масштабировать по горизонтали.

Итак, у нас есть два способа масштабирования – по вертикали и по горизонтали. Но если служба поддерживает вертикальное масштабирование аппаратных ресурсов (и способна извлечь выгоду из этого), то можно ли назвать ее «масштабируемой»? И насколько она масштабируема? Вертикальное масштабирование по своей природе ограничено объемом доступных вычислительных ресурсов, поэтому служба, которую можно масштабировать только по вертикали, вообще не очень масштабируема. Если может понадобиться масштабирование в десять, сто или тысячу раз, то служба должна поддерживать горизонтальное масштабирование.

Так в чем же разница между службами, поддерживающими и не поддерживающими горизонтальное масштабирование? Все сводится к одному: состояние. Службу, которая не поддерживает состояния или была спроектирована для распределения своего состояния между экземплярами, будет довольно легко масштабировать. Для любого другого приложения это будет сложно.

Более подробно идеи масштабируемости, состояния и избыточности будут рассмотрены в главе 7.

Слабая связанность

Слабая связанность – это свойство системы и стратегия проектирования, согласно которой компоненты системы знают лишь самый минимум о любых других компонентах. Можно сказать, что две системы *слабо связаны*, если изменение одного компонента не требует изменения другого.

Например, веб-серверы и веб-браузеры можно считать слабо связанными: серверы можно обновлять и даже полностью заменять, не опасаясь влияния на наши браузеры. Это возможно потому, что стандартные веб-серверы обмениваются данными с использованием набора стандартных протоколов[1]. Иначе говоря, они предоставляют *контракт на обслуживание*. Представьте, какой был бы хаос, если бы все веб-браузеры в мире приходилось обновлять после выхода новой версии NGINX или httpd[2]!

Можно сказать, что «слабая связанность» – это один из базовых принципов архитектуры микросервисов: разделение компонентов таким образом, чтобы изменения в одном не влияли на другой. Однако этим принципом часто пренебрегают, и его стоит повторить. Преимущества слабой связанности – и последствия пренебрежения ею – нельзя недооценивать. Очень легко создать систему, «худшую из возможных», которая сочетает в себе накладные расходы на управление и сложность, связанные с наличием нескольких служб, с зависимостями и связями монолитной системы: жуткий *распределенный монолит*.

К сожалению, не существует волшебной технологии или протокола, которые могли бы предотвратить тесную связанность ваших служб. Любой формат обмена данными может использоваться неправильно. Однако есть несколько приемов, помогающих добиться желаемого и – в сочетании с такими практиками, как декларативные API и методы управления версиями, – создавать слабо связанные службы, которые могут изменяться независимо друг от друга.

Эти приемы и практики будут подробно обсуждаться и демонстрироваться в главе 8.

Устойчивость

Устойчивость (близкий синоним к *отказоустойчивости*) – это мера способности системы восстанавливаться после ошибок и сбоев. Систему можно считать *устойчивой*, если она может продолжать работать правильно – возможно, с меньшей эффективностью – вместо полного отказа после выхода из строя какой-либо ее части.

При обсуждении устойчивости (а также других «облачных атрибутов», но особенно при обсуждении устойчивости) мы довольно часто используем слово «система». Термин *система*, в зависимости от контекста, может относиться к чему угодно, от сложной сети взаимосвязанных служб (например, целого распределенного приложения) до набора тесно связанных компонентов (таких как реплики одной функции или экземпляра службы), и даже к единственному процессу, выполняющемуся на единственной машине. Всякая система состоит из нескольких подсистем, которые, в свою очередь, состоят из более мелких подсистем, которые сами состоят из еще более мелких подсистем. И так до самого конца.

---

[1] Те, кто помнит браузерные войны 1990-х годов, помнят, что так было не всегда.

[2] Или если бы для каждого веб-сайта нужно было использовать другой браузер. Это было бы крайне неудобно, не так ли?

Выражаясь языком системотехники, любая система может содержать дефекты, или *нарушения*, которые мы, программисты, любовно называем *жучками* (*bugs*). Мы все слишком хорошо знаем, что при определенных условиях любая неисправность может привести к *ошибке* – так мы называем любое несоответствие между предполагаемым и фактическим поведением системы. Ошибки могут привести к тому, что система не сможет выполнить свою функцию, то есть *откажет*. Но это еще не все: отказ в подсистеме или компоненте приводит к нарушению работы более крупной системы; любое нарушение, не устраненное должным образом, может вызывать нарушения во все более вышестоящих подсистемах и, в конце концов, привести к полному отказу системы.

В идеальном мире каждая система должна тщательно проектироваться, чтобы предотвратить появление нарушений, но в реальной жизни это невозможно. Невозможно предотвратить все возможные нарушения – пытаться сделать это бессмысленно и непродуктивно. Однако если предположить, что все компоненты системы могут выходить из строя – а это так и есть, – и спроектировать их так, чтобы они правильно реагировали на возможные нарушения и ограничивали распространение их последствий, то можно создать систему, которая продолжает исправно функционировать, даже если некоторые из ее компонентов выйдут из строя.

Существует множество подходов к проектированию отказоустойчивых систем. Наиболее распространенным из них является, пожалуй, развертывание избыточных компонентов, но при этом также предполагается, что нарушение не повлияет на другие компоненты того же типа. Можно добавить автоматическое размыкание цепи и логику повтора, чтобы предотвратить распространение нарушений между компонентами. Неисправные компоненты можно даже вывести из строя намеренно, чтобы принести пользу всей системе.

Мы обсудим все эти подходы (и многие другие) в главе 9.

Устойчивость – это не безотказность

Термины *устойчивость* и *безотказность* описывают похожие понятия, которые часто путают. Но, как мы обсудим в главе 9, это не совсем одно и то же[1]:

- устойчивость системы – это степень, в которой она может продолжать правильно функционировать, столкнувшись с ошибками и неполадками. Устойчивость, наряду с другими четырьмя облачными свойствами, является лишь одним из факторов, влияющих на надежность;
- безотказность системы – это ее способность сохранять ожидаемое поведение в течение заданного интервала времени. Безотказность в сочетании с такими атрибутами, как доступность и ремонтопригодность, способствует общей надежности системы.

[1] Если вам интересно узнать академическую трактовку, то я настоятельно рекомендую книгу Кишора С. Триведи (Kishor S. Trivedi) и Андреа Боббио (Andrea Bobbio) «Reliability and Availability Engineering» (https://oreil.ly/80wGT).

Управляемость

Управляемость системы – это простота (или ее отсутствие), с которой можно изменить поведение системы для обеспечения безопасности, бесперебойной работы и соответствия меняющимся требованиям. Систему можно считать *управляемой*, если она позволяет изменить ее поведение без изменения кода.

Управляемость как свойство системы привлекает гораздо меньше внимания, чем другие атрибуты, такие как масштабируемость или наблюдаемость. Однако она играет не менее важную роль, особенно в сложных распределенных системах.

Например, представьте гипотетическую систему, которая включает службу и базу данных, и служба обращается к базе данных по URL. Что, если вам потребуется изменить эту службу так, чтобы она обращалась к другой базе данных? Если URL жестко запрограммирован, то вам может понадобиться изменить код и повторно развернуть систему, что иногда может быть неудобно по некоторым причинам. Конечно, можно обновить запись в системе DNS, чтобы она возвращала другой IP-адрес для заданного URL, но как быть, если вам потребуется повторно развернуть разрабатываемую версию службы, которая будет ссылаться на базу данных в среде разработки?

Управляемая система может, например, извлекать требуемый URL из легко изменяемой переменной окружения; если служба, которая ее использует, развернута в Kubernetes, то для корректировки ее поведения достаточно будет обновить значение в конфигурации. Более сложная система может даже предоставлять декларативный API, с помощью которого разработчик сможет сообщить системе, какого поведения он ожидает. Нет единственного правильного ответа[1].

Управляемость не ограничивается поддержкой изменений в конфигурации. Она охватывает все возможные аспекты поведения системы, будь то активация функций или ротация учетных данных и сертификатов TLS, или даже (и, возможно, особенно) развертывание либо обновление компонентов системы.

Управляемые системы предполагают адаптируемость и могут легко приспосабливаться к изменяющимся функциональным требованиям, а также к требованиям окружения или безопасности. С другой стороны, неуправляемые системы, как правило, более хрупкие, часто требуют специальных изменений, нередко выполняемых вручную. Накладные расходы, связанные с управлением такими системами, накладывают фундаментальные ограничения на их масштабируемость, доступность и надежность.

Идея управляемости и некоторые предпочтительные практики ее реализации в Go будут обсуждаться в главе 10.

---

[1] И есть много неправильных.

Управляемость – это не удобство сопровождения

Можно сказать, что управляемость и удобство сопровождения (ремонтопригодность) в чем-то «перекликаются», потому что обе связаны с простотой изменения системы[1], но на самом деле это совершенно разные понятия:

- управляемость описывает простоту изменения поведения работающей системы, вплоть до развертывания (и повторного развертывания) ее компонентов. Управляемость – это простота внесения изменений *извне*;
- удобство сопровождения описывает простоту изменения базовых функций системы, чаще всего ее кода. Ремонтопригодность – это простота внесения изменений *изнутри*.

Наблюдаемость

Наблюдаемость системы – это мера простоты определения ее внутреннего состояния по наблюдаемым результатам. Система считается *наблюдаемой*, если можно быстро и последовательно получать ответы на все новые вопросы о ней с минимальными предварительными знаниями, без необходимости внедряться в существующий код или писать новый.

На первый взгляд в этом нет ничего сложного: достаточно добавить журналирование, включить пару панелей мониторинга (дашбордов) – и система станет наблюдаемой, не так ли? Почти наверняка нет, особенно в современных сложных системах, где почти любая проблема так или иначе связана с сетью, в которой одновременно может обнаружиться несколько проблем. Эпоха стека LAMP закончилась; сейчас дело обстоит намного сложнее.

Это не означает, что метрики, журналы и трассировка стали неважны. Напротив, они так и остались основными строительными блоками наблюдаемости. Но одного их существования недостаточно: данные – это не информация. Их важно правильно интерпретировать. Они должны иметься в достаточном количестве. Они должны подсказывать ответы на вопросы, о которых вы даже не думали раньше.

Способность обнаруживать и отлаживать проблемы является фундаментальным требованием для сопровождения и развития надежной системы. Но в распределенной системе бывает сложно выяснить причины проблемы. Сложные системы слишком… сложны. Количество возможных состояний отказа в любой системе пропорционально произведению количества возможных состояний частичного и полного отказа каждого из ее компонентов, и невозможно предсказать их все заранее. Традиционного подхода, заключающегося в фокусировке внимания на вещах, которые по нашему мнению могут потерпеть сбой, недостаточно.

Появление новых методов наблюдения можно рассматривать как процесс эволюции мониторинга. Многолетний опыт проектирования, создания и сопровождения сложных систем научил нас, что традиционные методы

[1] К тому же оба термина начинаются на «У». Очень запутанно.

инструментирования, включая, помимо всего прочего, панели мониторинга, журналы или оповещения о различных «известных неизвестных», просто не справляются с проблемами, создаваемыми современными распределенными системами.

Наблюдаемость – сложная и тонкая тема, но, по сути, она сводится к следующему: подготовить систему к реальным условиям настолько хорошо, чтобы в будущем можно было отвечать на вопросы, о которых вы пока не задумывались.

Идея наблюдаемости и некоторые предложения по ее реализации будут обсуждаться в главе 11.

Что особенного в облачном окружении?

Миграция в облако является примером архитектурной и технической адаптации, движимой давлением окружающей среды. Это эволюция – выживание сильнейшего. Имейте в виду, что по образованию я – биолог.

В стародавние времена, когда все только начиналось[1], приложения создавались и развертывались (обычно вручную) на одном или нескольких серверах, где обслуживались и поддерживались со всем тщанием. Если они заболевали, то их заботливо лечили. Если служба выходила из строя, то ее исправляли простым перезапуском. Наблюдаемость заключалась во входе в командную оболочку сервера и просмотре журналов. В те времена все было проще.

В 1997 году только 11 % людей в промышленно развитых странах и 2 % во всем мире пользовались интернетом. Однако в последующие годы наблюдался экспоненциальный рост подключений к интернету, и к 2017 году это число выросло до 81 % в промышленно развитых странах и 48 % во всем мире[2] и продолжает расти.

Все эти пользователи – и их деньги – создавали давление на службы, требуя масштабирования. Более того, по мере роста сложности требований пользователей и их зависимости от веб-сервисов росли и ожидания, что их любимые веб-приложения будут многофункциональными и всегда доступными.

Результатом стало значительное эволюционное давление в сторону масштабирования, сложности и надежности. Однако эти три атрибута плохо сочетаются друг с другом, и традиционные подходы просто не могли и не могут угнаться за ними. Пришлось изобретать новые приемы и методы.

К счастью, с появлением общедоступных облаков и IaaS возможность масштабирования инфраструктуры существенно упростилась. Недостатки надежности часто можно было компенсировать количеством. Но это создало новые проблемы. Как обслуживать сотни, тысячи или даже десятки тысяч

---

[1] Это было в 1990-х.

[2] International Telecommunication Union (ITU). «Internet users per 100 inhabitants 1997 to 2007» и «Internet users per 100 inhabitants 2005 to 2017». *ICT Data and Statistics* (*IDS*).

серверов? Как установить на них свое приложение или обновлять его? Как отлаживать возникающие в нем проблемы? Как вообще узнать – здорово ли оно? Проблемы в небольшом масштабе доставляют лишь раздражение, становятся очень сложными с увеличением масштаба.

Облачные технологии важны, потому что масштабирование является причиной (и решением) всех наших проблем. Это не волшебство. Здесь нет ничего особенного. Если отбросить причудливую терминологию, то облачные методы и технологии существуют только для того, чтобы дать возможность использовать преимущества «облака» (количество) и компенсировать его недостатки (отсутствие надежности).

Итоги

В этой главе мы познакомились с историей компьютерных вычислений и узнали, что то, что теперь мы называем «облачным окружением», не является чем-то новым – это неизбежный результат цикла технологического развития, стимулирующего инновации.

Однако все эти причудливые термины, с которыми мы познакомились, сводятся к одному: современные приложения должны надежно служить большому количеству людей. Методы и технологии, которые мы называем «облачными», – это лучшие современные практики создания масштабируемых, адаптируемых и достаточно устойчивых служб.

Но какое отношение все это имеет к языку Go? Как оказывается, для облачного окружения нужны свои облачные инструменты. В главе 2 мы начнем говорить о том, что это означает.