УДК 004.4
ББК 32.971.3
П26

Майкл Л. Перри

П26 Искусство неизменяемой архитектуры: теория и практика управления данными в распределенных системах / пер. с анг. С. В. Минца; науч. ред. В. С. Яценков. – М.: ДМК Пресс, 2022. – 388 с.: ил.

ISBN 978-5-93700-111-5

Эта книга раскрывает преимущества использования неизменяемых объектов в распределенных системах. Вы узнаете о том, почему важна неизменяемость, исследуете пространство альтернатив и аспекты исторического моделирования. Затем ознакомитесь с математическими основами неизменяемости и увидите, как применять эти знания для анализа систем, построения машин состояний и соблюдения правил безопасности. В завершение будут рассмотрены компоненты компьютерной системы и их использование в неизменяемой архитектуре.

Издание предназначено для архитекторов программного обеспечения и опытных разработчиков, а также аналитиков бизнес-систем.

УДК 004.4
ББК 32.971.3

First published in English under the title The Art of Immutable Architecture; Theory and Practice of Data Management in Distributed Systems by Michael Perry, edition: 1

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (анг.) 978-1-107-02398-7
ISBN (рус.) 978-5-93700-111-5

Copyright @ 2020 by Michael L. Perry
© Оформление, издание, перевод, ДМК Пресс, 2022

Оглавление

Об авторе 16

О техническом рецензенте 17

Благодарности 18

Введение 20

ЧАСТЬ I. ОПРЕДЕЛЕНИЕ 25

Глава 1. Почему неизменяемая архитектура 26

Решение проблемы неизменяемости 26
Проблемы с неизменяемостью 27
Начинаем новое путешествие 27
Ошибки распределенных вычислений 28
  Сеть ненадежна 28
  Время задержки не равно нулю 29
  Топология не меняется 30
  Изменение предположений 30
Неизменяемость меняет все 31
  Совместное изменяемое состояние 32
  Структурное разделение 32
Проблема двух генералов 34
  Заранее подготовленный протокол 36
  Уменьшение неопределенности 36
  Дополнительное сообщение 37
  Доказательство невозможности 38
Смягчение ограничений 39
  Переопределение проблемы 40
  Решать и действовать 40
  Принять истину 41
  Действенный протокол 41
Примеры неизменяемой архитектуры 42
  Git 43
  Блокчейн 44
  Docker 46

Глава 2. Формы неизменяемой архитектуры 48
Выведение состояния из истории 48
Исторические записи 49
Опираясь на прошлое 49
Развитие понимания 50
Изменяемые объекты 50
Идентичность 50
Изменение состояния 51
Проекции 51
Два вида состояния 52
Проецирование объектов 52
Поиск событий 53
Генерация событий 53
CQRS 54
DDD 55
Взгляд с точки зрения функций 56
Коммутативные и идемпотентные события 57
Асинхронное обновление представления модели 58
Цикл обновления 58
Однонаправленный поток данных 60
Неизменяемая архитектура приложений 61
Историческое моделирование 62
Частичный порядок 62
Предшественники 63
Преемники 65
Неизменяемые графы 66
Совместная работа 68
Ациклические графы 69
Своевременность 69
Ограничения исторического моделирования 70
Отсутствие центральной власти 71
Отсутствие часов реального времени 72
Отсутствие ограничений уникальности 72
Отсутствие агрегирования 73

Глава 3. Как читать историческую модель 75
Графы типов фактов 76
Шахматная партия 80
Важные атрибуты 81
Цепочка фактов 81
Исход партии 83
Графы экземпляров фактов 85
Бессмертная партия 87

Регистрация ходов 88
Блестящая победа 91
Язык фактологического моделирования 92
Объявление типов фактов 92
Запрос модели 94
Переход по уровням 95
Объединение совпадений 95
Экзистенциальные квантификаторы 96
Текущее значение 98
Правила авторизации 99
Шахматное приложение 100
Примеры использования 101
Интерфейс пользователя 102
Действия 102
Представления 103

ЧАСТЬ II. ПРИМЕНЕНИЕ 105

Глава 4. Независимость от местоположения 106
Моделирование с неизменяемостью 107
Синхронизация 107
Изучение соглашений 108
Идентичность 108
Автоинкрементные идентификаторы 108
Зависимость от среды 109
Вставка отношения «родитель–ребенок» 110
Удаленное создание 111
URL-адреса 111
Идентификация, не зависящая от местоположения 112
Естественные ключи 113
GUID 114
Временные метки 114
Кортежи 114
Хеши 115
Открытые ключи 116
Случайные числа 116
Причинность 117
Упорядочивание шагов 117
Транзитивное свойство 119
Параллелизм 120
Частичный порядок 121
Теорема CAP 121
Определение CAP 122

Доказательство теоремы CAP 124
Проверка алгоритма 125
Конечная согласованность 127
Виды согласованности 128
Сильная конечная согласованность в системе ретрансляции 129
Идемпотентность и коммутативность 130
Получение сильной конечной согласованности 131
Система управления контактами 133
Воспроизведение истории 135
Бесконфликтные реплицированные типы данных (CRDT) 136
CRDT, основанные на состоянии 137
Частично упорядоченное состояние 137
Причинная история 138
Векторные часы 139
История фактов 141
Наборы 142
Частичная упорядоченность 142
Обновление 143
Слияние 143
Исторические записи 143
Различение записей 144
Удаление записи 145
Изменение записи 146
Записи причинно-следственно связаны 146
Преимущества явной причинности 148
Исторические факты 150
Заключение 150

Глава 5. Анализ 152
Примеры использования 153
От сценария использования к решению 154
От расширения к преемственности 155
Данные 158
Идентификаторы 158
Кардинальность 159
Изменение 162
Представления 164
Поиск точки старта 164
Аннотированные каркасы 165
Удаление из списков 166
Сотрудничество 170
Регионы 170
Пересечение границ 171

Разговоры ... 173
Факты о публикации ... 173
Взаимодействие подсистем ... 174
Допустимые упорядочения ... 175
Устранение условий гонки ... 176
Реагирование на различные допустимые заказы ... 177
Последствия ... 179
Индексы ... 180
Ограничения уникальности ... 180
Навигация ... 181
Поиск ... 182
Ожидаемое количество результатов ... 183
Отсутствие неявного порядка ... 184
Агрегаты ... 185
Итерации ... 186
Порядок создания ... 186

Глава 6. Переходы состояний ... 188
Множество свойств ... 189
Доставка и выставление счетов ... 190
Внедрение обратных заказов у поставщика ... 191
Отмены и возвраты ... 191
Параллельные конечные автоматы ... 192
Много дочерних элементов ... 193
Отслеживание проблем в программном обеспечении ... 194
Дочернее состояние ... 195
Составные диаграммы перехода состояний ... 195
Декларативная функция состояний ... 196
Условная проверка ... 197
Допустимость неопределенного состояния ... 198
Циклы в изменении состояния ... 199
Сбор данных во время переходов ... 200
Неизменяемые переходы состояний ... 201
Вопрос, стоящий за состоянием ... 202
Перевод конечного автомата в историческую модель ... 202
Выполнение заказов ... 202
Отслеживание изменений в программном обеспечении ... 205
Причины для вычисления состояния ... 207
Обработка следующего действия ... 208
Поиск рабочих элементов ... 209
Выполнение компенсирующих транзакций ... 210
Единый источник истины ... 211
Оркестраторы ... 212
Согласованное состояние ... 212

Центральная проверка ... 212
Сходящиеся истории ... 213
Определение неизменяемых записей ... 213
Запрос для следующего действия ... 213
Локальное фиксирование действий ... 214
Определите компенсирующие действия ... 214

Глава 7. Безопасность ... 215
Доказательство авторства ... 215
Ключевые пары ... 216
Дайджест ... 217
Авторизация ... 218
Факты принципала ... 219
Запрос авторизации ... 219
Первоначальная авторизация ... 220
Предоставление полномочий ... 222
Ограниченные полномочия ... 223
Неограниченные полномочия ... 224
Транзитивная авторизация ... 226
Отмена ... 226
Авторизация при получении ... 228
Конфиденциальность ... 229
Недоверенные узлы ... 229
Асимметричное шифрование ... 229
Асимметричное ограничение размера ... 230
Шифрование симметричного ключа ... 230
Шифрование исторических фактов ... 231
Ограничьте распространение конфиденциальных фактов ... 232
Правила распространения ... 232
Доказательства ... 233
Атаки и контрмеры ... 234
Секретность ... 235
Общий симметричный ключ ... 236
Секретный канал для обсуждения ... 236
Создание секретного канала ... 237
Командные правила распространения ... 238
Ограничение области применения общего ключа ... 239
Когорты ... 239
Периоды ... 240

Глава 8. Шаблоны ... 242
Структурные шаблоны ... 242
Сущность ... 243

Структура 243
Пример 244
Последствия 244
Связанные шаблоны 244
Владение 245
Структура 245
Пример 247
Последствия 248
Связанные шаблоны 248
Удаление 248
Структура 249
Пример 249
Последствия 250
Связанные шаблоны 250
Восстановление 251
Структура 251
Пример 252
Последствия 253
Связанные шаблоны 253
Членство 253
Структура 253
Пример 254
Последствия 255
Связанные шаблоны 256
Изменяемое свойство 256
Структура 256
Пример 259
Последствия 260
Связанные шаблоны 262
Ссылка на сущность 262
Структура 262
Пример 263
Последствия 264
Связанные шаблоны 265
Шаблоны рабочих процессов 265
Транзакция 266
Структура 266
Пример 267
Последствия 268
Связанные шаблоны 268
Очередь 268
Структура 269
Пример 269

Последствия 270
Связанные шаблоны 271
Период 271
Структура 271
Пример 272
Последствия 274
Связанные шаблоны 274
Исходящие 274
Структура 275
Пример 279
Последствия 280
Связанные шаблоны 280
Проектирование на основе ограничений 281

ЧАСТЬ III. РЕАЛИЗАЦИЯ 283

Глава 9. Инверсии запросов 284
Механизация проблемы 285
Анатомия запроса 285
Последовательность шагов 286
Фильтр по экзистенциальному состоянию 287
Затронутое множество 288
Вычисление затронутого набора 289
Инвертирование длинных запросов 290
Неудовлетворительные инверсии 291
Движение назад 292
Доказательство полноты 293
Новые результаты 294
Дальнейшая оптимизация 295
Экзистенциальные условия 296
Рекурсивная инверсия 297
Условия хвоста 298
Удаление результатов 299
Когда удаление не является удалением 301
Вложенные подзапросы 302
Тавтологические условия 304
Продолжение доказательства полноты 306
Потенциальные и фактические изменения 307
Удаление отсутствующих результатов 308
Кеши есть множества 308
Инверсия запросов на практике 309

Глава 10. Базы данных SQL 310
Идентичность 311
Хранение с адресацией по содержанию 311
Преимущества 312
Коллизии хешей 313
Вероятность коллизии хешей 314
Избегайте использования хешей в качестве первичных ключей 315
Структура таблицы 315
Отношения 317
Вставка преемников 318
Необязательные предшественники 318
Много предшественников 319
Канонический хеш множества 320
Вставка многих предшественников 321
Запросы 322
Соединения 322
Коррелированные подзапросы 323
Производные таблицы 324
Выбор результатов 325
Оптимизация 326
Ложные соединения 327
Охватывающие индексы 328
Where Not Exists 328
Изменяемые свойства 329
Удаление 329
Очереди 330
Интеграция 332
Интеграция унаследованных приложений 333
Сканеры 333
Триггеры 334
Захват изменений данных 335
Создание отчетов из баз данных 335
Агностичные к приложениям хранилища 336
Общая таблица фактов 337
Отношения предшественников 338
Управление версиями 339
Избегайте последовательных номеров версий 340
Структурное версионирование 341

Глава 11. Коммуникация 343
Гарантии доставки 343
Лучшие усилия 345
Подтверждение 345

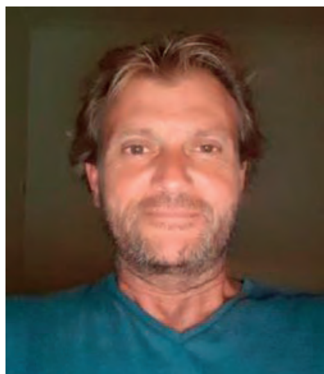
Безопасные методы 346
Идемпотентные методы 346
Неидемпотентные методы 348
Повторная попытка в пределах соединения 349
Долговечные протоколы 349
Очереди 349
Темы 350
Обработка сообщений 350
Большинство протоколов являются асинхронными 351
HTTP обычно является синхронным 351
Синхронизация данных 352
В рамках организации 353
Опорные точки 353
Многие подписчики 355
Ответы 356
Уведомления 357
Между организациями 358
Асинхронный через HTTP 358
Webhook 359
Эмуляция REST 360
Клиенты, подключающиеся время от времени 361
Очередь на стороне клиента 362
Закладка на стороне клиента 363
Выбор подмножества 364
Предотвращение избыточных загрузок 367

Глава 12. Генерируемое поведение 369
Проекции 370
Определение проекций 370
Конвейеры проекций 372
Заинтересованность 373
Интерес к удаленным сущностям 374
Интерес к прошлым периодам 376
Совместное использование интересов 376
Потеря интереса 377
Неизменяемые среды выполнения 379
Генерация модели 380
Выполнение запросов 380
Тестирование 381
Взаимодействие с пользователем 382
Взаимодействие 383
Неизменяемые организации 385
Основа для принятия решений 385
Глобально распределенные системы 386

Об авторе

Майкл Л. Перри (Michael L. Perry) опирался на работы таких математиков, как Бертран Мейер (Bertrand Meyer), Лесли Лампорт (Leslie Lamport) и Дональд Кнут (Donald Knuth), при создании математической системы для разработки программного обеспечения. Он воплотил эту систему в ряде проектов с открытым исходным кодом. Майкл часто выступает с докладами по математике и программному обеспечению на различных мероприятиях и в интернете. Вы можете узнать больше на сайте qedcode.com.

О техническом рецензенте



Карстен Томсен (Carsten Thomsen) – в основном бэкенд-разработчик, но работает и с небольшими фронтенд-разработками. Он является автором и рецензентом ряда книг и создал многочисленные учебные курсы для компании Microsoft, связанные с разработкой программного обеспечения. Он работает как фрилансер/подрядчик в разных странах Европы, используя такие инструменты, как Azure, Visual Studio, Azure DevOps и GitHub. Является исключительным специалистом по устранению неполадок путем постановки правильных вопросов, в том числе менее логичных, переходя от наиболее логичного к наименее логичному. Ему также нравится работать с архитектурой, исследованиями, анализом, разработкой, тестированием и исправлением ошибок в программах. Он обладает хорошими навыками наставничества и руководства командой, а также исследования и представления нового материала.

Введение

Шел 2001 год. Я присоединился к команде, использующей J2EE[1] версии 1.3 для создания распределенного процессора подарочных карт. Система точек продаж была написана на Microsoft Visual C++ 6.0. Мы только узнали об этой новой вещи под названием SOAP (Simple Object Access Protocol) – простой протокол доступа к объектам. Шутка заключалась в том, что он был слишком плохо определен, чтобы называться протоколом, он не был связан с доступом к объектам и был совсем не простым. Но он давал некоторые надежды на то, чтобы заставить клиента C++ обращаться к серверу Java.

Мы добавили в свои библиотеки три новые книги. Первая была посвящена реализации клиента SOAP на C++, вторая – JAXP, Java API для обработки XML, а третья подробно описывала работу и ограничения TCP/IP. Вооружившись этими инструментами, мы начали.

Сначала задача заключалась в том, чтобы заставить две платформы общаться друг с другом. Когда мы наконец остановились на подмножестве SOAP, с которым могли работать обе стороны, мы подумали, что преодолели этот бугор. Мы не знали, что по другую сторону были горы.

Возникли проблемы с надежностью сети. Мы создали лабораторию, которая постоянно проводила транзакции каждую ночь. Утром мы проверяли баланс карт и обнаруживали, что на некоторых машинах была неверная сумма. Это приводило к целому дню копания в журналах, настраивали следующий тест, а затем запускали его до утра.

Со временем мы разработали протокол обмена сообщениями (поверх SOAP), основанный на подтверждениях и квитанциях. Одна сторона отправляла сообщение. На следующее утро мы обнаруживали пропажу сообщений. Затем получатель подтверждал, что сообщение пришло. На следующее утро мы обнаруживали дубликаты. И тогда отправитель регистрировал подтверждение. Пропавших сообщений стало меньше, но все еще не было идеально.

Потребовалось много неудачных релизов и много лет работы, в том числе по праздникам, чтобы решить все проблемы. Мы узнали о «задаче двух генералов» (TGP) и поняли, почему наш протокол обмена сообщениями был несовершенен. Затем мы узнали о конечной согласованности и получили рабочее решение. Это решение требовало, чтобы существовала некоторая неопределенность в отношении того, сколько денег осталось на подарочной карте. Мы попытались провести разговор об этом с владельцем продукта. Банкиры получают конечную последовательность денег. Владелец продукта не был банкиром.

Уроки, которые мы извлекли при работе с подарочными картами, были усвоены тяжелым трудом. «Гарантированная доставка» означает не то, что вы

[1] Java 2 Enterprise Edition или J2EE – ранняя версия Jakarta EE – набора спецификаций и документации для языка Java, описывающей архитектуру серверной платформы. https://ru.wikipedia.org/wiki/Jakarta_EE.

думаете. Вам нужно сначала переместить данные, а затем обработать их. Удаленные вызовы процедур (RPC) – это не вызовы процедур. В системе «клиент–сервер» нет ни одной строки кода, до которой транзакция отменяется и после которой она фиксируется. Я бы не хотел получать эти уроки снова и снова.

И вот я начал собирать эти уроки воедино и определять систему, которую я назвал историческим моделированием. Она была основана на идее, что исторические факты не могут быть изменены или уничтожены. Она опиралась на отношения предшественников и преемников между фактами. И идентифицировала факты, основываясь только на их содержании, а не местоположении. Я заполнил ноутбук примерами исторических моделей. В конце концов, я интуитивно почувствовал, какие виды решений могут быть смоделированы исторически, а какие – нет. Тогда я понял, что должен поделиться этим. Надеюсь, я смогу избавить кого-то еще от необходимости усваивать эти уроки трудным путем.

С тех пор у меня было бесчисленное множество разговоров о неизменяемых архитектурах. Я разбил тему на легко усваиваемые фрагменты для выступлений на конференциях и в группах пользователей, а также создал два фреймворка с открытым исходным кодом – Correspondence и Jinaga. Однако по отдельности они не дали возможности другим начать применять неизменяемость на практике. Принятие только части идей оставляет пробелы, которые могут быть заполнены лишь с помощью остальной части системы.

Все это привело к книге, которую вы сейчас держите в руках. Это полное описание системы, шаблонов и техник. Она предвосхищает проблемы, которые создает историческое моделирование, и предлагает решения, позволяющие обеспечить его целостную реализацию. Самое главное, в ней представлена математическая основа, которая заставляет эту технику работать.

Если вы дочитали введение до конца, то наверняка сталкивались с некоторыми из этих проблем. Возможно, вы даже нашли похожие решения. Остается только еще несколько вопросов, которые, вероятно, у вас есть по поводу этой книги. Кто должен ее читать? Что я получу от нее? Как она организована? И как мне ее читать?

Рад, что вы спросили.

Кому следует ее прочитать

Эта книга предназначена в первую очередь для трех аудиторий: лиц, принимающих решения, конструкторов систем и создателей инструментов. Вы – лицо, принимающее решения, если вы определяете проблемы, для которых хотите создать решения. Ваша должность может быть технический директор, владелец продукта или аналитик бизнес-систем. Существуют проблемы, которые вы можете передать на аутсорсинг, есть проблемы, для которых вы можете купить решения, а есть проблемы, которые определяют основную ценность вашего бизнеса. Вам необходимо найти подходящих людей для решения проблем третьего типа. Чтобы найти их, вам нужно уметь с ними разговаривать. А после того, как вы пригласите их на работу, нужно понять, чем они занимаются. Если ваша основная бизнес-проблема похожа на ту, которую можно решить с помощью неизменяемой архитектуры, эта книга поможет вам создать такую команду и провести эти беседы.

Или, возможно, вы занимаетесь созданием систем. Вы являетесь членом команды, привлеченной для создания ценностей в основной области бизнеса. Ваша должность может быть разработчик, инженер по контролю качества или дизайнер пользовательского интерфейса. Вы знаете, как решать проблемы. Но было бы здорово иметь несколько готовых решений для наиболее распространенных проблем распределенных вычислений. Вы хотите знать, что все нестандартные ситуации учтены. Вы хотите иметь общий язык для обсуждения решений с людьми, которые помогают вам находить их. Если ваши задачи по разработке программного обеспечения требуют создания согласованных распределенных систем, то эта книга даст вам такие инструменты.

Наконец, вы, как и я, возможно, являетесь создателем инструментов. Вы – множитель силы. Ваши разработки дают возможность другим создавать решения быстрее, более предсказуемо и более эффективно. Вы можете быть архитектором решений или сопровождающим открытого исходного кода. Если у вас есть команда, вы хотите, чтобы ее члены были сосредоточены на создании бизнес-ценностей, а вы позаботитесь о «сантехнике». Если вы обслуживаете сообщество, вы хотите, чтобы потребители могли быстро изучить и применить ваш фреймворк для создания надежных систем. В любом случае, в этой книге изложены математика, алгоритмы и шаблоны, которые гарантируют корректность ваших решений.

Что вы получите от этого

У меня есть секрет. Это книга по математике. Не говорите об этом никому, кто не дочитал до конца введение.

Математика – величайшее изобретение человечества. Она удивительна в своей способности описывать мир природы. Она поразительно применима к широкому кругу проблем. И это единственный способ, с помощью которого мы можем быть уверены в чем-либо.

Обычно мы убеждаемся в том, что поняли что-то правильно, проверяя это. Мы применяем наше решение в одной ситуации и смотрим, получим ли мы ожидаемый результат. Затем пробуем другой сценарий и смотрим, что получится. Если мы действительно хороши, то сможем представить несколько непредвиденных условий и протестировать их. Но непредвиденное очень трудно предугадать.

Тестирование – это сбор опытных данных. Оно дает уверенность в том, что система ведет себя так, как ожидается, в определенных случаях, но не дает никакой уверенности в том, что вы ничего не упустили.

Знание требует математических выводов. Если что-то доказано математически, то вы можете быть уверены, что это будет истинно, независимо от того, какой тест вы попробуете провести. Теорема Пифагора верна для любого прямоугольного треугольника. Геометрия Евклида верна для всех фигур на плоскости. Если ваши рассуждения безупречны, вы можете быть уверены, что не пропустили ни одного крайнего случая.

Это не означает, что математические истины универсальны. Дело в том, что они имеют известные ограничения. Деление работает только для ненулевых делителей. Теорема Пифагора действует лишь на плоскости. Правила дедукции

говорят нам, как перенести эти ограничения на решение, чтобы точно знать, где это решение применимо, а где нет.

Эта книга применяет математическую строгость к проблеме распределенных вычислений. Она не является первой в данной области, но предлагает полное и практическое решение. Если вы будете следовать дедуктивным рассуждениям над проблемой и внесете ограничения распределенных систем в свои вычисления, то в итоге вы получите понимание границ решения. Эта книга – ваш путеводитель в этом процессе.

Как она организована

Книга условно разделена на три части, предназначенные для трех основных аудиторий. Лицам, принимающим решения, необходимо прочитать только первую часть, которая включает в себя первые три главы. В этой части вы сначала узнаете, почему неизменяемость так важна. Затем исследуете пространство альтернатив и познакомитесь с историческим моделированием. Наконец, вы узнаете, как читать историческую модель, чтобы более эффективно общаться со своей командой. Вы можете прекратить чтение, когда мы перейдем к глубокой математике.

Специалисты по созданию систем захотят прочесть вторую часть. Она включает в себя главы с четвертой по восьмую. Мы глубоко погружаемся в математические основы неизменяемости, причинности и бесконфликтных реплицируемых типов данных (conflict-free replicated data types – CRDT). Затем увидим, как применять эти математические рассуждения для анализа систем, построения машин состояний и соблюдения правил безопасности. Именно эти инструменты понадобятся вам для создания надежных распределенных систем. В завершение данного раздела приведен каталог основных шаблонов, которые помогут вам начать строить исторические модели.

Люди моей профессии, создатели инструментов захотят дочитать до конца. В третьей части мы разберем компоненты компьютерной системы и узнаем, как использовать их в неизменяемой архитектуре. Мы научимся обновлять пользовательский интерфейс с помощью инверторов запросов. Мы будем хранить неизменяемые записи в реляционной базе данных, будем надежно и безопасно обмениваться неизменяемыми сообщениями в сетях различных типов. В конце концов, мы собираем все вместе и описываем экосистему, состоящую из совместных приложений, генерирующих новое поведение на основе общих спецификаций. Это нечто поистине прекрасное и вдохновляющее, и я надеюсь, что вы последуете за мной до конца.

Как ее читать

Теперь, когда вы знаете, что это книга по математике, у вас могут возникнуть некоторые сомнения по поводу того, как вы будете ее читать. Возможно, вы с трудом проходили алгебру или забросили вычисления. Вы можете подумать, что математика не для вас.

Я считаю, что математика подходит всем. И моя цель в данной книге – доказать это. Математика – это не что иное, как применение логических рас-

суждений к символическим представлениям абстрактных понятий. С другой стороны, программирование – это применение логических операций к символическому языку, описывающему общие правила. Иными словами, это одно и то же. Если вы программист, то вы – прикладной математик.

Одна из проблем математики – это жаргон. Для того чтобы эффективно общаться друг с другом, математикам приходится придумывать слова для обозначения идей. К сожалению, естественный язык ограничен, и все хорошие слова уже заняты. И поэтому математики либо придумывают новые слова, либо используют термины, которые означают почти то, что нужно. В этой книге мы будем говорить, например, о свойствах полурешетки связности. Но я постараюсь не использовать эти слова, если смогу избежать этого. А если избежать не удастся, я буду давать им четкое определение.

Еще одна проблема с математикой – это то, как она написана. Математические работы имеют предсказуемую форму. Они начинаются с аннотации, затем полностью определяют проблему. Далее следуют раздел за разделом лемм и предложений, выстраивающих аргументацию. Каждое утверждение обосновывается предыдущими утверждениями. Наконец, как у М. Найт Шьямалана (M. Night Shyamalan), следует поворот сюжета, последнее утверждение рассматривает всю аргументацию в перспективе, и появляется результат.

Хотя мне очень нравятся хорошие математические статьи, я не читаю их так, как они написаны. Я бегло просматриваю первые несколько абзацев в поисках мотивации проблемы. Сканирую заголовки, чтобы понять суть аргументов. Я хочу знать, почему каждое утверждение доказано и каков будет его вклад в общее дело. Я хочу знать, как будет развиваться история, прежде чем потратить время на ее понимание.

Я написал эту книгу так, как я читаю статьи по математике. В каждом разделе вы поймете, чем обусловлен тот или иной результат. Затем вы увидите набросок основных рассуждений. Не будет загадкой, почему каждый шаг находится на отведенном ему месте. Потом каждый из этих шагов будет обоснован с той строгостью, которой он требует.

Я ожидаю, что это повлияет на то, как вы будете читать книгу. Если вам нужны только результаты, вы можете прочитать всего один-два абзаца после заголовка раздела. Если вы хотите узнать, почему или как, то продолжите читать дальше, чтобы понять аргументацию. А если вам нужно убедиться, то дочитайте весь раздел до конца. Важно, что вы можете прекратить чтение, когда оно становится слишком глубоким, и перейти к следующему разделу. Вы не пропустите ничего важного для себя.

Если вы прочитали этот раздел, ничего не пропустив, то я искренне рад, что вы у меня есть. Вы – один из моих людей. С вашей помощью мы сможем создать программное обеспечение, которое необходимо миру. Мы сделаем его надежным, эффективным и правильным. И это даст нашим пользователям автономию, необходимую для того, чтобы они могли выполнять свою работу творчески и уверенно, зная, что мы обеспечили математическую строгость.

ЧАСТЬ I

Определение

Глава 1

Почему неизменяемая архитектура

Распределенные системы – это сложно.

Большинство из нас пользовались веб-сайтом для покупки товара. Возможно, вы видели страницу покупки, на которой есть предупреждение «**Не нажимайте кнопку "Отправить" дважды!**». Может быть, вы пользовались сайтом, который просто отключает кнопку покупки, после того как вы ее нажмете. Авторы этого сайта столкнулись с одной из сложных проблем распределенных систем и не знали, как ее решить. Они переложили ответственность за предотвращение дублирования расходов на потребителя.

Возможно, вы пользовались мобильным приложением в поезде. Поезд въезжает в туннель как раз в тот момент, когда вы сохраняете некоторые данные. Мобильное приложение работает несколько секунд, прежде чем вы это поймете. Выедет ли поезд из туннеля до того, как приложение прекратит попытки сохранения данных? Исправит ли приложение ошибку, когда связь будет восстановлена? Или вы потеряете свои данные и вам придется вводить их снова?

Если вы участвуете в создании распределенных систем, от вас ожидают, что вы будете находить, исправлять и предотвращать подобные ошибки. Если вы работаете в отделе контроля качества, ваша работа заключается в том, чтобы представить все возможные сценарии, а затем воспроизводить их в лаборатории. Если вы разработчик, вам нужно написать код для всех различных исключений и условий такого процесса. А если вы занимаетесь архитектурой, вы отвечаете за разрубание гордиева узла возможных сбоев и смягчение последствий. Это тонкий процесс, с помощью которого мы создаем системы, управляющие нашим обществом.

Решение проблемы неизменяемости

Распределенные системы трудно писать, тестировать и поддерживать. Они ненадежны, непредсказуемы и небезопасны. Процесс, с помощью которого мы их создаем, обязательно упустит из виду дефекты, которые негативно повлияют на наших пользователей. Но это не ваша вина. До тех пор, пока мы будем полагаться на отдельных людей в поиске, устранении и смягчении этих проблем, дефекты будут упущены.

В этой книге рассматривается другой процесс создания распределенных систем. Вместо того чтобы связывать программы между собой и тестировать дефекты, этот подход начинается с фундаментального представления бизнес-проблемы, которая *объединяет* машины. И это фундаментальное представление является *неизменным*.

На первый взгляд, неизменяемость – это простая концепция. Запишите некоторые данные и убедитесь, что они никогда не изменятся. Они не могут быть изменены, обновлены или удалены. Они неизменны. Неизменяемость решает проблему распределенных систем по одной простой причине: каждая копия неизменяемого объекта так же хороша, как и любая другая копия. До тех пор, пока объекты никогда не меняются, синхронизация удаленных копий является тривиальной задачей.

Проблемы с неизменяемостью

К сожалению, неизменяемость противоречит тому, как на самом деле работают компьютеры. Машина имеет ограниченный объем памяти. Машины работают, изменяя содержимое ячеек памяти с течением времени, чтобы обновить свое внутреннее состояние. Поэтому первая проблема моделирования неизменяемых данных на компьютере заключается в том, как представить их в изменяемой памяти.

Вторая проблема заключается в том, что когда мы смотрим на мир проблем, которые хотим решить, мы видим изменения. Люди меняют свои имена, адреса и номера телефонов. Остатки на банковских счетах растут и падают. Имущество переходит из рук в руки, и право собственности переходит. Как же нам смоделировать изменяющееся проблемное пространство с неизменными данными?

Наш первоначальный инстинкт заключается в том, чтобы моделировать изменчивый мир в изменчивом пространстве компьютера. Именно это решение привело нас к созданию программ и баз данных на основе изменений. Программы содержат операторы присваивания, базы данных содержат операторы ОБНОВИТЬ. Когда мы соединяем эти программы и базы данных вместе для создания распределенных систем, возникает безумное непредсказуемое поведение. И перед нами встает бесконечная задача тестирования, пока все эти аномалии не исчезнут.

Начинаем новое путешествие

Цель этой книги состоит в том, чтобы вместо этого смоделировать бизнес-домен как одну большую неизменяемую структуру данных. Для одной машины или базы данных было бы невозможно разместить всю эту структуру. Да это и нежелательно. Поэтому книга также стремится продемонстрировать, как реализовать подмножества этой структуры данных в отдельных базах данных, программах и машинах. Эти компоненты взаимодействуют через хорошо продуманные протоколы, которые учитывают особенности распределенных систем, чтобы развивать неизменяемую структуру данных с течением времени.

Данное решение не является новым. На протяжении всей этой книги мы будем обращаться к исследованиям прошлого в виде статей по математике и информатике. Каждое утверждение обосновано, ни один из выводов не является оригинальным. Я надеюсь лишь собрать все эти знания воедино, чтобы начать ваше путешествие к более надежным, устойчивым и безопасным распределенным системам. Давайте начнем это путешествие с понимания проблемы распределенных вычислений.

Ошибки распределенных вычислений

В период с 1991 по 1997 год инженеры компании Sun Microsystems собрали список ошибок, которые программисты часто допускали при написании программного обеспечения для сетевых компьютеров. Билл Джой (Bill Joy), Дэйв Лайон (Dave Lyon), Питер Дойч (Peter Deutsch) и Джеймс Гослинг (James Gosling) составили каталог из восьми предположений, которых разработчики обычно придерживались в отношении распределенных вычислений. Эти предположения, хотя и очевидно неверные, когда они высказаны прямо, тем не менее определяют многие из решений, которые инженеры Sun нашли в системах того времени.

Заблуждения таковы:

- сеть надежна;
- задержка равна нулю;
- пропускная способность бесконечна;
- сеть безопасна;
- топология не меняется;
- имеется один администратор;
- транспортные расходы равны нулю;
- сеть однородна.

Хотя с момента написания этого списка прошло много лет, многие из этих предположений продолжают быть общепринятыми. Я могу вспомнить несколько случаев, когда меня удивляло, что программа, которая безупречно работала на *локальном компьютере*, быстро выходила из строя при развертывании в тестовой среде. Программа содержала скрытые предположения о том, что сеть надежна, задержка нулевая и топология не меняется. Вот примеры только этих трех предположений.

Сеть ненадежна

Один из способов, с помощью которого эти заблуждения проявляются в современных системах, – это когда удаленный программный интерфейс приложения (API) рассматривается так, как будто это вызов функции. Несколько сервисов платформы продвигают эту абстракцию, включая вызовы удаленных процедур, .NET Remoting, Distributed COM, SOAP и SignalR. Когда удаленный вызов выглядит как вызов локальной функции, разработчику легко забыть о том, что сеть ненадежна.

Каждый раз, когда вы вызываете функцию, вы можете быть уверены, что выполнение программы продолжится с ее первой строки. И если функция

дойдет до оператора возврата, вы можете быть уверены в том, что следующей строкой будет та, которая следует за вызовом функции. Вызовы удаленных процедур, однако, не дают таких гарантий. Они могут потерпеть неудачу при вызове или при возврате. Вызывающий код не сможет определить, где именно это произошло.

Абстракция, которая скрывает факт сетевого перехода, оказывает плохую услугу потребителям. Пытаясь сделать вещи более простыми и привычными, она делает вид, что неудобная правда может быть проигнорирована. Такие абстракции позволяют разработчикам поверить в заблуждение, что сеть каким-то образом надежна.

Время задержки не равно нулю

Современные веб-приложения отказались от клиентских прокси в пользу более явных REST API. Эти API позволяют избежать ошибки представления удаленной машины так, как если бы она была библиотекой функций, которые можно надежно вызывать. Вместо этого они представляют мир как паутину взаимосвязанных ресурсов, каждый из которых отвечает на небольшой набор HTTP-методов. К сожалению, при таком стиле программирования легко забыть, что время задержки не равно нулю.

Некоторые из HTTP-методов гарантированно идемпотентны. Если клиент дублирует запрос, сервер обещает не дублировать эффект. Протокол не имеет возможности обеспечить соблюдение этой гарантии, но приложения на стороне сервера обычно поддерживают данный договор. Примерами методов HTTP, которые являются идемпотентными, являются PUT и PATCH. Метод HTTP, который не гарантирует идемпотентность, – это POST.

В интернете HTTP POST часто используется для отправки формы. Когда веб-приложение реагирует быстро, отсутствие гарантии идемпотентности не имеет большого значения. Но по мере увеличения задержки пользователь начинает сомневаться, действительно ли он нажал на кнопку отправки. А если эта кнопка инициировала покупку, то пользователь начинает задумываться, не будет ли с него снята двойная оплата, если он попытается сделать это снова. Конечный пользователь не имеет хороших средств защиты во время длительной задержки после нажатия кнопки «Купить». У разработчика клиентского приложения также нет хорошего ответа на задержку при отправке POST.

Не существует корректного использования API, в котором присутствуют неидемпотентные сетевые запросы. Поскольку задержка не равна нулю, всегда будет время, в течение которого клиент не уверен в том, получил ли сервер запрос. Когда задержка превышает время, которое клиент готов ждать, он должен сделать выбор: либо прервать попытку, либо повторить ее. Если клиент прерывает попытку, то он не знает, был ли запрос обработан. А если он повторяет попытку, то эффект может быть продублирован.

Метод POST действительно является частью спецификации HTTP. И эта спецификация не дает никаких гарантий относительно его идемпотентности. Но любой API, который включает неидемпотентный POST, делает неверное предположение, что задержка равна нулю. Это заставляет клиента делать неверный выбор, когда данное предположение оказывается ложным.

Топология не меняется

Большинство систем управления базами данных включают концепцию, которая заставляет разработчиков предполагать, что топология не меняется. Эти базы данных позволяют легко установить автоинкрементируемый идентификатор (auto-incremented ID) записи. Каждый раз, когда вставляется новая запись, база данных генерирует следующий номер в последовательности. Этот номер в дальнейшем используется для идентификации записи.

Автоинкрементируемый идентификатор требует, чтобы топология оставалась постоянной на протяжении всего многоэтапного процесса. Представьте себе веб-приложение, которое вставляет данные формы пользователя в базу данных, а затем перенаправляет пользователя на страницу, представляющую эти новые данные. Чтобы выполнить это с помощью автоинкрементируемого идентификатора, браузер должен ждать, пока запрос пройдет весь путь до базы данных и пока ответ вернется обратно, прежде чем он сможет узнать URL следующей страницы. Приложение предполагает, что топология за это время не изменится.

На первый взгляд это может показаться верным предположением. Обычно это так и есть. Изменения топологии сервера происходят редко, а сетевые запросы обычно выполняются быстро (задержка равна нулю). Однако для веб-приложения с высокой посещаемостью никогда не будет момента, в течение которого не обрабатывается ни один запрос. Предположение о том, что топология не меняется, будет нарушено для некоторых запросов.

Топология может измениться во время обновления системы. Она обязательно изменится во время аварийного переключения. И она снова изменится при возврате назад после устранения аварии. При изменении топологии база данных, к которой поступает запрос, будет не той, в которой была создана исходная страница. Вместо этого база данных будет репликой оригинала. Если реплика лишь немного отстает от оригинала, то изменение топологии будет заметным. И оно будет заметным, потому что, опять же, задержка не равна нулю.

Использование автоинкрементных идентификаторов повсеместно. Они являются выбором по умолчанию для большинства баз данных. И все же их использование противоречит предположению о том, что топология не будет меняться.

Изменение предположений

Заблуждения распределенных вычислений – это простые предположения. Мы делаем их, потому что наши инструменты, спецификации и обучение привели нас к этому. Неидемпотентный метод POST является валидной частью спецификации HTTP. Автоинкрементные идентификаторы являются полезной особенностью большинства систем управления базами данных. Почти каждый учебник по разработке приложений научит новичка использовать эти возможности. Тот факт, что при этом он делает неверное предположение, даже не приходит ему в голову.

Инструменты, которые мы используем, и шаблоны, которым мы следуем сегодня, – все это появилось в те времена, когда предположения о высокой надежности, нулевой задержке и неизменности топологии не были заблуждениями.

Внутрипроцессные вызовы процедур абсолютно надежны. Последовательные операторы программы имеют очень малые, очень предсказуемые характеристики задержки. А последовательные счетчики в цикле for никогда не вернутся в начало функции, чтобы обнаружить, что топология кода изменилась. Именно когда мы применяем эти абстракции при вызове удаленных процедур (Remote Procedure Call – RPC), в сетевых запросах и автоинкрементируемых идентификаторах, возникают проблемы. Когда мы применяем языки и шаблоны прошлого к проблемам современных распределенных систем, неудивительно, что программисты делают неверные предположения.

Все заблуждения при распределенных вычислениях проистекают из-за одного простого допущения – распределенные системы создаются с использованием инструментов, предназначенных для работы в одном потоке на одном компьютере. Разработчики представляют себе быструю, изолированную, неизменную, последовательную среду выполнения, а затем рассматривают идиосинкразии распределенных систем как частные случаи. Дублирование транзакции из-за тайм-аута в сети не является ошибкой. Столкновение идентификаторов, вызванное отказом базы данных, не является дефектом. Это реалии распределенных систем, которые мы не можем обойти кодом или протестировать. Они требуют нового набора инструментов, моделей и предположений.

Неизменяемость меняет все

В 2015 году Пэт Хелланд (Pat Helland) написал статью «Неизменяемость меняет все»[1], в которой анализируется несколько вычислительных решений, основанных на неизменяемости. В ней показано, что неизменяемость решает многие проблемы на нескольких уровнях вычислительной абстракции. На одном конце спектра низкоуровневые системы хранения данных используют семантику копирования при записи для защиты от износа носителя. На другом конце приложения накапливают факты, доступные только для чтения, и получают текущее состояние. Данная статья не претендует на новые идеи, а лишь указывает на то, что во всех этих решениях присутствует общая нить – неизменяемость.

В прошлом компьютеры были медленными, дорогими и ограниченными машинами, которые могли оперировать только с небольшими наборами данных. Сегодня это быстрые, дешевые и способные рабочие лошадки, которые хранят огромное количество данных. Если раньше разработчикам приложений приходилось оптимизировать хранение данных, перезаписывая информацию, когда она больше не нужна, то сегодня мы можем позволить себе сохранять все. Нет экономической необходимости обновлять или уничтожать информацию.

В то же время современные компьютеры гораздо более связаны между собой, чем это было в прошлом. Вместо того чтобы размещать рабочую нагрузку вместе с данными, на которых она работает, мы перешли в мир микросерви-

[1] *Helland Pat.* (2015). Immutability Changes Everything // http://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf.

сов и мобильных устройств, которые широко обмениваются данными. Многие машины разделяют вычислительное бремя и бремя хранения данных, которое раньше выполнялось одной. В результате координация стала более дорогой, несмотря на то что вычисления стали дешевыми.

И если в прошлом хранить неизменяемые копии данных было дорого, нынешние архитектурные ограничения *требуют* этого. Данные не только дешевле, чем раньше, но создание неизменяемых копий фактически *обеспечивает* создание решений, которые масштабируются на несколько машин. Когда две машины совместно используют изменяемые данные, им необходимо координировать свои действия при изменении этих данных. Им может понадобиться блокировать друг друга, чтобы гарантировать, что только одна из них может изменять данные в любой момент времени. Но когда эти данные не могут меняться, координация или блокировка не требуется. Снижение затрат обеспечивает неизменяемость, а неизменяемость обеспечивает современную архитектуру.

Совместное изменяемое состояние

Многие трудные проблемы в вычислениях – это проблемы, которые мы создали для себя сами. Возьмем, к примеру, проблему общего изменяемого состояния в многопоточной системе. Один поток записывает исходные данные в общую область памяти, а другой поток выполняет в ней вычисления. Эти два потока должны быть тщательно скоординированы, чтобы один из них не записывал данные в общую память до того, как другой закончит чтение из нее. Если первый поток перезапишет данные, пока второй продолжает вычисления, результаты будут полной бессмыслицей. Обычно мы решаем подобную проблему с помощью блокировки, ограничивающей возможность масштабирования.

Но есть решение, которое не ухудшает масштабируемость. Вместо блокировки мы можем использовать неизменяемые структуры данных. Вместо того чтобы перезаписывать память следующим набором данных, первый поток просто выделит новую память. Когда завершается построение структуры данных, первый поток передает указатель второму. С этого момента ни один поток не может изменить содержимое этой области памяти. Она остается полностью неизменяемой.

На первый взгляд кажется, что мы улучшили масштабируемость за счет снижения эффективности памяти. Кажется, что, вместо того чтобы изменять только одну небольшую часть структуры данных, мы должны создавать целую копию при каждой операции. Если бы это было так, было бы трудно оправдать этот компромисс, даже с учетом снижения стоимости хранения данных. К счастью, нам не приходится идти на такой компромисс.

Структурное разделение

Тот факт, что структуры данных должны быть неизменяемыми, открывает новые возможности. По мере того как мы создаем новые структуры данных, мы можем повторно использовать существующие части старых структур данных. При этом нет необходимости копировать эти части, потому что мы уже установили, что они не будут меняться. Мы просто создаем новые элементы данных

для представления тех, которые «изменились», и разрешаем им указывать на те, которые не изменились.

Такая техника называется *структурным разделением*. Это обычная оптимизация, которая *обеспечивается* неизменяемыми структурами данных. Возьмем, например, двоичное дерево, показанное на рис. 1.1. Каждый узел дерева содержит часть данных, в нашем случае число. Он также содержит два указателя, один на число, которое меньше, чем этот узел, и один на число, которое больше. Нахождение конкретного числа в этой структуре данных является быстрым, потому что вы идете вниз, спрашивая «меньше чем или больше чем» на каждой остановке.

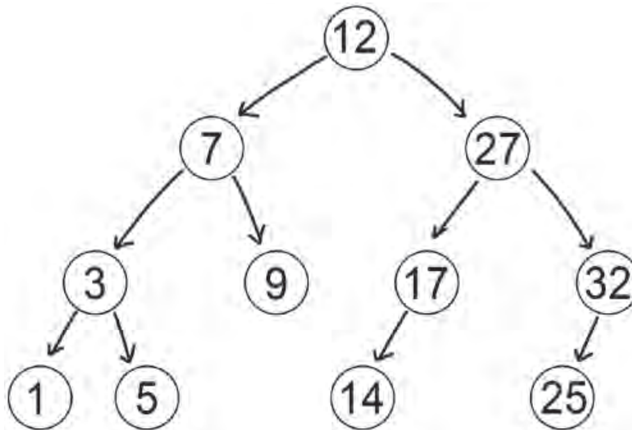


Рис. 1.1. Двоичное дерево чисел

Чтобы вставить новое число в двоичное дерево, сначала нужно найти место для него. Пройдя до места, где оно должно находиться, вы обнаружите, что оно либо меньше числа, у которого нет левого пути, либо больше числа, у которого нет правого пути. Оказавшись там, вы захотите «изменить» этот узел, чтобы добавить новый путь. Однако изменение узла не разрешается – все они являются частью неизменяемой структуры данных. Поэтому вместо этого вы создаете новый узел.

Этот новый узел должен быть левее или правее родительского, и поэтому вы хотите «изменить» и этот узел. Но опять же, изменение родителя не допускается. Поэтому вы создаете нового родителя, который указывает на новый дочерний узел.

Продолжая подниматься вверх по дереву, вы в конце концов достигнете корня, как показано на рис. 1.2. Независимо от того, куда вы вставляете новое число, вы всегда в итоге создаете новый корневой узел. Этот узел фактически является новой версией дерева. Он представляет собой форму дерева после вставки. Предыдущий корневой узел все еще существует, и узлы, на которые он указывает, не были изменены. Любые потоки, выполняющие параллельный поиск в этой версии дерева, могут спокойно продолжать это делать. Они не будут затронуты новым деревом, которое разделяет большую часть своей структуры со старым.

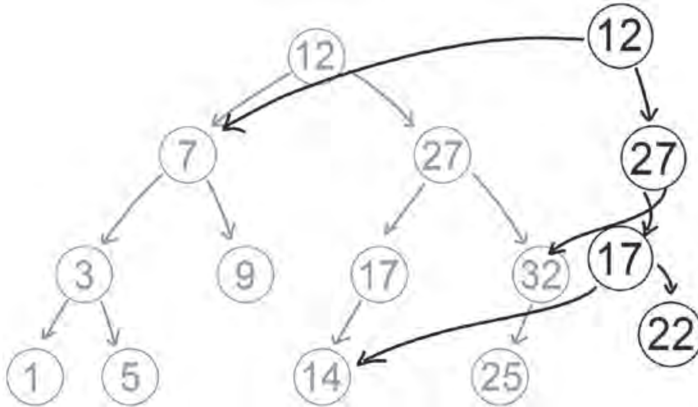


Рис. 1.2. После вставки числа 22 новая версия двоичного дерева разделяет большую часть своей структуры с предыдущей версией

Эта оптимизация была бы невозможна, если бы потоки могли изменять эти структуры данных. Благодаря совместному использованию структуры эти две версии дерева становятся чувствительными к модификации. Только потому, что мы договорились *не* изменять узлы, мы можем избежать глубокого разделения структуры. Неизменяемость обеспечивает структурное разделение, а структурное разделение оптимизирует неизменяемость.

Проблема двух генералов

Нигде в вычислительной технике неизменяемость не является более ценной, чем при обмене данными между машинами. Но прежде чем мы сможем понять причину этого, мы должны сначала понять масштаб проблемы. И нет лучшего способа сделать это, чем притча о двух генералах[1].

Представьте себе осажденный город. В его стенах оборона непреодолима. Прямая атака почти наверняка потерпит неудачу. За пределами города находятся две армии, которым удалось отрезать его от линий снабжения. Генералы этих армий ждут, наблюдая, как город медленно слабеет под блокадой.

В какой-то момент оборона города станет достаточно слабой для атаки. Генералы этих двух армий (одной – на востоке, второй – на западе) постоянно наблюдают за ситуацией через сеть разведчиков, шпионов и гонцов. Каждый день они определяют, достаточно ли слаб город. Когда приходит время, они готовят атаку на следующий день. Эта ситуация показана на рис. 1.3.

[1] *Akkoyunlu E. A., Ekanadham K., Huber R. V.* Some constraints and tradeoffs in the design of network communications. ACM SIGOPS Operating Systems Review. November 1975.

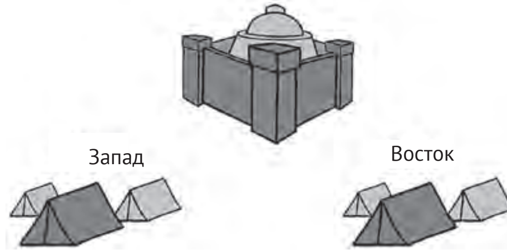


Рис. 1.3. Две армии расположились лагерем у осажденного города

Атаки одной армии недостаточно. Атака была бы отбита и атакующая армия уничтожена. Оставшаяся армия была бы не в состоянии поддерживать блокаду, и поэтому вскоре она была бы разгромлена. Только скоординированная атака с востока и запада позволит завоевать город.

Теперь представьте, что вы – генерал армии запада. Ваш партнер на востоке отделен от вас вражеской территорией. Вы не можете общаться напрямую, а можете только послать гонцов через враждебную местность без гарантии успеха. Любое сообщение может быть потеряно, его носитель убит или захвачен в плен. Вы двое должны разработать метод надежной связи, построенной из ненадежных компонентов.

Если вы на западе решите, что город достаточно слаб и что время для атаки наступило, вы начнете готовить свою армию. Вы также пошлете гонца на восток, чтобы сообщить другому генералу, что вы нападете утром. Если гонец прибудет благополучно, то восточный генерал сможет начать подготовку и присоединиться к вам в атаке. Благодаря вашим совместным усилиям атака, скорее всего, будет успешной.

Но если гонец будет убит или захвачен, сообщение не дойдет. Если это произойдет, утром ваша армия отправится в одиночную атаку на город. Ваша армия будет уничтожена, и осада будет проиграна. Как показано на рис. 1.4, вы не знаете, как действовать дальше. Поэтому до наступления утра вы должны быть уверены в том, что послание получено.

Рис. 1.4. Западный генерал не знает, смогут ли они атаковать

Заранее подготовленный протокол

Давайте попробуем разработать протокол, который даст нам некоторую уверенность в том, что сообщение было получено. Предположим, вы просите восточного генерала отправить в ответ гонца с подтверждением того, что ваше сообщение было получено. Теперь если вы получите подтверждение до утра, то сможете уверенно начать атаку. Вы знаете, что восточный генерал получил сообщение и присоединится к вам на поле боя. Если же, с другой стороны, вы не получите подтверждения, то отмените атаку, не зная, дошел ли первоначальный гонец. Как генерал западной армии вы можете быть уверены, что не атакуете, пока не узнаете, что восточный генерал получил ваше сообщение.

Но хотя этот протокол дает такие гарантии генералу запада, он не делает этого для восточного генерала. Представьте себе, что вы находитесь на востоке и получили сообщение о том, что западная армия будет атаковать утром. У вас есть достаточно времени, чтобы начать подготовку своей армии. И, согласно протоколу, вы отвечаете подтверждением. Если сообщение с подтверждением дойдет до генерала запада, то атака пройдет по плану.

Но если это сообщение будет утеряно, то западный генерал не будет атаковать. Помните, он ждет подтверждения, чтобы знать, что вы получили его сообщение. Если вы атакуете утром, не зная, что западный генерал получил ваше подтверждение, то ваша армия может быть разбита. И тогда вы окажетесь в неопределенности, как на рис. 1.5.

Рис. 1.5. Восточный генерал не уверен

Уменьшение неопределенности

Этот протокол недостаточен. Вы пробуете различные стратегии, чтобы улучшить его. Первая стратегия состоит в том, чтобы просто послать больше гонцов. Вместо того чтобы полагаться на одного гонца, вы посылаете двух. Вероятность того, что два сообщения будут потеряны, конечно, меньше, чем вероятность потери одного. Но эта вероятность не равна нулю. И вот вы пробуете снова.

Вы можете послать трех, четырех гонцов. Выбирайте любое число, какое пожелаете. По мере того как вы увеличиваете число, вероятность потери сообщения становится все ближе и ближе к нулю. Но она никогда не достигает его.

Вы никогда не сможете выбрать число гонцов достаточно большое, чтобы быть уверенным, что сообщение будет получено.

И тогда вы меняете свой подход. Вы посылаете гонцов с постоянной скоростью, пока не будет получен ответ. С запада, когда вы решаете атаковать, вы посылаете гонцов с сообщением об *атаке* раз в десять минут. Когда вы получаете первое *подтверждение* с востока, вы прекращаете отправку сообщений. Что касается генерала на востоке, он будет отвечать *подтверждением* каждый раз, когда сообщение об *атаке* будет получено. Пока он будет получать постоянный поток сообщений об *атаке*, он будет отвечать на них с той же скоростью подтверждением. И как только этот поток прекратится, он может считать, что подтверждение получено.

А это правильно? Можно ли воспринимать отсутствие сообщений как сигнал? Возможно ли, что шесть гонцов в час продолжают отправляться с запада, но все они захвачены? Генерал на востоке не может этого исключить. И поэтому он все еще рискует атаковать утром без поддержки с запада.

Дополнительное сообщение

Поэтому, как восточный генерал, вы выдвигаете дополнительное требование к протоколу. В дополнение к сообщению об атаке с запада и подтверждению с востока вы требуете, чтобы запад ответил подтверждением. Если вы на востоке получите подтверждение до утра, то вы знаете, что подтверждение было получено на западе. Поэтому вы можете атаковать с уверенностью, зная, что западный генерал получил подтверждение и поэтому присоединится к вам. Но если вы не получили подтверждения, тогда должны воздержаться.

Хотя это новое сообщение дает новые гарантии генералу востока, оно снова запутывает ситуацию на западе. Когда западный генерал посылает подтверждение, он не может узнать, было ли оно получено. Если да, то восточный генерал атакует. Если нет, то он воздержится. И как показано на рис. 1.6, у него нет никакой уверенности в том, что его атака будет поддержана.

Рис. 1.6. Западный генерал снова не уверен, что восток будет атаковать

Добавление еще одного сообщения только перенесло неопределенность на другую сторону. На самом деле это не решило проблему. Мы все еще не нашли протокол, который обеспечит, что обе армии либо атакуют, либо воздержатся, если два генерала могут общаться только посредством ненадежных сообщений.

И действительно, мы никогда его не найдем.

Доказательство невозможности

Проблема двух генералов, как назвал ее Джим Грей (Jim Gray) в 1978 году[1], не имеет решения. Не существует конечного протокола, который может дать обоим генералам взаимную уверенность в достижении соглашения. Я не просто говорю, что никто не нашел решения. Я говорю, что решения не может существовать.

Е. А. Аккоюнлу (A. Akkoyunlu), который опубликовал оригинальную проблему и доказательство невозможности в 1975 году[2], назвал эту взаимную гарантию *полным статусом*. Он описал межпроцессные коммуникационные протоколы, которые согласовывают транзакции между участниками. Протокол в идеале должен обеспечивать этот статус для участников относительно результата каждой транзакции. Аккоюнлу доказал, что распределенная система не может достичь полного статуса за конечное число сообщений.

Его доказательство не требует, чтобы мы исчерпали все возможные решения. Оно не оставляет места для хитроумных уловок, о которых мы не подумали. Вместо этого оно основано на противоречии. Пусть кто угодно придумает протокол и принесет его Аккоюнлу, утверждая, что он обеспечивает полный статус. Даже не зная, как работает этот протокол, он показывает, что он не подтверждает данное утверждение.

Предположим, что вы представляете протокол, который, как вы утверждаете, обеспечивает полный статус двум генералам после обмена конечным числом сообщений. В конце этого обмена оба генерала будут знать, что другой собирается атаковать. Если генералы следуют этому протоколу и ни одно сообщение не было потеряно, то существует минимальное количество сообщений, чтобы достичь этого состояния. Назовем это число N. Число N является специфическим для протокола.

Поскольку N – это наименьшее количество сообщений, которыми необходимо обменяться для достижения полного состояния, мы знаем, что меньшее число будет недостаточным. В частности, мы не достигли полного статуса после N – 1 сообщений. Один из генералов все еще должен находиться в состоянии, в котором он не уверен, собирается ли другой атаковать.

Поскольку N – 1 сообщений будет недостаточно, N-е сообщение очень важно. Без него протокол не будет работать. И все же получение сообщения не гарантировано. Отправитель N-го сообщения не знает, будет ли оно получено. Поэтому отправитель N-го сообщения не имеет полного статуса и не получит его, так как в протоколе нет дальнейших сообщений. Эта ситуация показана на рис. 1.7.

---

1 *Gray Jim.* (1978). Notes on Data Base Operating Systems. Chapter 3. Operating Systems, an Advanced Course. Springer-Verlag, London, UK.

2 *Akkoyunlu E. A., Ekanadham K., Huber R. V. (*1975). Some constraints and tradeoffs in the design of network communications. Published in SOSP 1975. DOI:10.1145/800213.806523.

Рис. 1.7. Отправитель последнего сообщения не имеет полного статуса

Это противоречит вашему утверждению, что протокол предоставляет полный статус при конечном числе сообщений. Поэтому мы можем заключить, что такого протокола не существует.

Смягчение ограничений

Проблема двух генералов (TGP) является аналогом многих проблем, которые мы пытаемся решить в распределенных системах. Используя только ненадежные сети для передачи сообщений между узлами, мы должны построить системы, которые тем не менее достигают соглашения с высокой степенью уверенности. Невозможность TGP, казалось бы, говорит нам о том, что это глупая затея. К счастью, проблемы, которые мы решаем в распределенных системах, немного проще, чем этот вымышленный аналог.

Рассмотрим банкомат. Клиент банка использует терминал, чтобы снять наличные со своего счета. Эта обычная повседневная операция кажется реальностью, созданной TGP. На западе у вас есть терминал банкомата, способный выдавать наличные. На востоке – центральный компьютер банка, который регистрирует движение денег на счетах клиентов. Между ними – враждебная территория цифровых коммуникаций, угрожающая прервать доставку сообщений.

Мы хотим, чтобы транзакция либо удалась, либо не удалась. Если она успешна, то деньги выдаются, и счет клиента дебетуется. В случае неудачи наличные не выдаются, и счет не дебетуется. Мы хотим избежать результата, в котором успех с одной стороны и неудача – с другой. Клиенты были бы очень расстроены, если бы их счета дебетовались, но наличные не поступали, а банки теряли бы деньги, если бы их банкоматы выдавали наличные без соответствующего списания.

Переопределение проблемы

Невозможность решения TGP говорит нам, что это невозможно сделать. И все же миллионы транзакций банкоматов обрабатываются каждый день[1]. Очевидно, здесь что-то не так. В примере с банкоматом мы не смогли распознать, что ограничения на систему более мягкие, чем кажется на первый взгляд. Давайте рассмотрим подробнее причину невозможности решения TGP. Отсюда мы сможем увидеть, как ослабить ограничения и создать жизнеспособный протокол.

В первоначальном виде проблема имеет два строгих ограничения:

1) генерал не будет атаковать, если у него нет уверенности в том, что другой генерал тоже будет атаковать;
2) атака произойдет утром.

Согласно первому ограничению, поведение каждого генерала основано на том, что он знает о поведении другого генерала. До тех пор, пока один из генералов находится в состоянии неопределенности, оба остаются неопределенными. Не существует сообщения, которое могло бы одновременно изменить их мнение.

Согласно второму ограничению, существует крайний срок. Когда этот срок наступит, они должны достичь согласия. Любые сообщения, уже находящиеся в пути к этому моменту, не должны повлиять на окончательный результат. Не будет никаких дополнительных сообщений для устранения любой затянувшейся неопределенности.

Если мы ослабим эту пару ограничений, то сможем сформулировать задачу, которая имеет приемлемое решение. Действительно, можно найти протокол, который обменивается полным статусом, если разрешить одной стороне действовать в условиях неопределенности и убрать крайний срок. Это нарушает изложение проблемы двух генералов, но подходит для примера банкоматов. Действительно, мы обнаружим, что эта смягченная версия подходит ко многим бизнес-проблемам, которые решаются с помощью распределенных систем.

Решать и действовать

Сначала ослабим ограничение, согласно которому генерал будет атаковать только в том случае, если он уверен, что его коллега тоже нападет. Западный генерал решает, что время пришло, и готовится к атаке независимо от того, что происходит на востоке. То, что является глупым поведением для генерала, может быть оправданным компромиссом для банкомата. Когда клиент снимает деньги со своего счета через банкомат, одна из сторон должна действовать без полной уверенности в том, что другая сторона последует ее примеру. Либо банкомат должен выдать наличные, либо компьютер банка должен зафиксировать списание. Рассмотрим последствия и шаги по исправлению каждого решения, если оно окажется односторонним.

Предположим, что банк регистрирует дебет, но банкомат не выдает наличные. В этом случае клиент покидает терминал без наличных, но банк считает,

[1] В исследовании Федеральной резервной системы США по платежам за 2013 год сообщается о 5,8 млрд снятий денег в банкоматах в 2012 году.

что деньги он получил. Следствием этого является то, что клиент недоволен, когда обнаруживает проблему, и его доверие к банку подорвано. Корректирующим действием является отмена дебета после обнаружения проблемы.

Теперь предположим, что банкомат выдает наличные, но банк не может зафиксировать дебет. В этом сценарии клиент ушел счастливым, а банкомат повторяет попытки коммуникации до тех пор, пока она не будет успешной. Тем временем, возможно, клиент сможет снять деньги в другом банкомате, поскольку банк не знает, что его баланс был исчерпан. Если это так, то корректирующим действием будет взимание с клиента комиссии за овердрафт.

Очевидно, что один из этих сценариев лучший как для банка, так и для клиента. Он защищает доверие, передает власть в руки клиента и дает банку дополнительный поток доходов. Итак, в этой ситуации разработчик распределенной системы определяет, что банкомат будет выдавать наличные, даже если неизвестно, зафиксирует ли банк дебет.

Принять истину

Разработчик может уверенно принять это решение, только если он ослабит второе ограничение – что существует крайний срок. Предположим, что банкомат выдал наличные, но затем возникли технические трудности при сообщении этого факта банку. Может потребоваться некоторое время, пока техник починит терминал банкомата и восстановит канал связи. Когда терминал сообщает банку, что наличные были выданы, банк должен признать эту истину. Он не может отклонить транзакцию на основании прошедшего времени или текущего баланса счета клиента.

Повреждения банкомата могут быть настолько серьезными, что цифровая запись транзакции не может быть восстановлена. Возможно, произошел полный невосстановимый сбой жесткого диска. В этом случае можно прибегнуть к дополнительной экспертизе – подсчитать оставшиеся в банкомате наличные деньги и определить, была ли завершена последняя транзакция. Если банкомат, включая все наличные деньги, полностью уничтожен, то даже этот метод может оказаться недоступным. Но, конечно, в этом случае банк потеряет больше, чем одну транзакцию. Принятие истины означает принятие некоторого риска.

Действенный протокол

Учитывая эти смягченные ограничения, мы можем теперь разработать протокол, который в конечном итоге достигнет полного статуса. Одна сторона (в данном случае банкомат) достигает точки, где она может уверенно принять решение. Она действует (выдает наличные), а затем продолжает протокол до тех пор, пока не узнает, что другая сторона знает о принятом решении. Он продолжает это делать независимо от того, сколько прошло времени или какие противоречивые обстоятельства вмешались.

Чтобы принять решение, банкомат связывается с центральным банком. Он проверяет, достаточно ли у владельца счета средств для выдачи запрашиваемых наличных. Он также проверяет свое местное хранилище банкнот, чтобы убедиться, что он сможет выполнить свою часть транзакции. В ходе этого

процесса банк может наложить временный арест на средства клиента. Но этот арест только снижает вероятность возникновения овердрафта, но не может его предотвратить. Банкомат со своей стороны накладывает временный арест на хранилище купюр – только один клиент может пользоваться банкоматом в данный момент. Если обе эти проверки пройдут, то банкомат выдаст наличные. Он принимает окончательное решение.

После принятия решения банкомат переходит ко второй фазе. На этом этапе решение уже принято, наличные выданы. Цель данной фазы заключается в том, чтобы сообщить об этом факте центральному банку. Вторая фаза не ограничена по времени, и истина не может быть опровергнута.

Этот вид протокола Джим Грей (Jim Gray) в 1978 году назвал *двухфазным действием* (Two Phase Commit – 2PC). На первой фазе, известной как фаза *голосования*, координатор получает от каждого участника подтверждение того, что он может зафиксировать запрошенную транзакцию. На второй фазе – фазе *фиксации* – координатор информирует каждого участника о своем решении. В предыдущем примере банкомат играет роль координатора и одного участника. Он является единственным лицом, принимающим решение после того, как он собрал достаточно информации для этого.

Примеры неизменяемой архитектуры

Преимущества неизменяемости не остались не замеченными разработчиками распределенных систем. Некоторые из наиболее успешных распределенных систем, используемых сегодня, построены на этой концепции. Они получают от неизменяемости такие возможности, которых было бы трудно достичь в противном случае. Три примера – Git, блокчейн и Docker.

Git – это распределенная система контроля версий, популярная как среди команд разработчиков открытого исходного кода, так и среди корпоративных разработчиков. Она предлагает преимущество автономии для каждого отдельного разработчика. Разработчик может вносить изменения, переключаться между параллельными версиями продукта и разрешать конфликты – и все это в пределах изолированной реплики репозитория. Когда разработчики подключают свои реплики – напрямую друг с другом или с общим центральным хранилищем, – они только обмениваются информацией. Во время этого обмена не происходит блокировки или согласования, что делает взаимодействие быстрым.

Блокчейн – это термин для обозначения целого ряда родственных архитектур. Первым блокчейном был биткойн – распределенная валюта, полностью основанная на криптографических алгоритмах. Большинство блокчейнов сохраняют экономические аспекты валюты, но некоторые накладывают дополнительные функции на основную структуру данных. Главной особенностью блокчейна является неизменяемый *распределенный реестр*, обеспечивающий гарантию достоверности единой, прозрачной истории.

Docker – это технология выполнения программного обеспечения в контейнерах, как если бы вся операционная система и все зависимости были заключены в единую изолированную среду исполнения. Это эволюция за

пределы физических машин, которые действительно выполняли изолированную работу, и виртуальных машин, которые имитировали эту среду для целей переносимости и масштабирования. Docker достигает эффективности, которой не хватало виртуальным машинам, благодаря разумному использованию структурного разделения и неизменяемых образов дисков. Это привело к дальнейшему продвижению в области управления, например кластерам Kubernetes и mesh-вычислениям.

Все эти примеры используют преимущества неизменяемости для обеспечения своих основных возможностей. Интересно, что все они также являются открытыми системами. Скорее всего, это просто следствие того, что открытое программное обеспечение легкодоступно для анализа и поддержки в качестве архитектурных примеров. Нет никаких причин полагать, что неизменяемость не будет столь же ценной для закрытой системы, как и для открытой. Давайте проанализируем каждую из них немного подробнее, чтобы увидеть, как неизменяемость служит своим целям.

Git

Git стремится предоставить каждому разработчику автономию, предоставляя всю необходимую информацию в реплике репозитория. Репозиторий состоит из отдельных изменений исходного кода, называемых коммитами. Коммиты неизменяемы и содержат ссылки на связанные с ними коммиты. Весь репозиторий – это постоянно растущая история коммитов, накопленная в течение жизни проекта.

Коммит – это набор изменений, внесенных в проект одним разработчиком в один момент времени. Идентичность коммита определяется исключительно его содержимым – именами используемых файлов, изменениями, сделанными в этих файлах, именем разработчика, а также причиной и временем внесения изменений. Это содержимое хешируется, и полученный хеш-код отныне используется для идентификации коммита. В результате получается неизменяемый граф коммитов, подобный показанному на рис. 1.8. Каждый разработчик, клонирующий репозиторий, будет вычислять один и тот же хеш для каждого коммита, что делает эти идентификаторы детерминированными и последовательными.

```
*   249916e - (origin/master, origin/HEAD) Merge pull request #11 from micha
|\
| * 44644f4 - chore(package): update lockfile package-lock.json (5 weeks ago
| * 8aede8a - chore(package): update webpack to version 4.36.1 (5 weeks ago)
|/
| * 26223d5 - (origin/greenkeeper/jinaga-pin-2.3.8) fix: pin jinaga to 2.3.8
|/
| * a8613b0 - (origin/greenkeeper/jinaga-2.4.0) chore(package): update lockf
| * 1e480d3 - chore(package): update jinaga to version 2.4.0 (5 weeks ago) <
|/
*   4037b30 - Merge pull request #8 from michaellperry/dependabot/npm_and_ya
|\
| * a5dc439 - Bump lodash from 4.17.11 to 4.17.14 (5 weeks ago) <dependabot[
|/
*   d91d87d - Merge pull request #7 from michaellperry/dependabot/npm_and_ya
```

Рис. 1.8. История коммитов в репозитории Git

Текущее состояние исходного кода может быть построено из коммитов без повторного подключения к удаленному узлу, что обеспечивает автономность каждого узла. Разработчик работает, отключившись от сервера, чтобы самостоятельно создать новый набор коммитов. Пока коммиты работают, они не подключаются к удаленному узлу, чтобы заблокировать файлы или проверить наличие последних изменений. Они работают в полной изоляции, никаких обращений к серверу не требуется.

Когда возникает конфликт, как это часто бывает в исходном коде, разработчик находит в своем локальном хранилище всю информацию, необходимую для его разрешения. У него есть идентификационные данные сотрудников (возможно, даже самого себя), вовлеченных в конфликт. Он знает точный контекст изменения – как выглядел код в момент его изменения. И даже из комментариев к коммиту у него есть некоторые подсказки о намерениях каждого программиста.

На основе всей этой информации разработчики могут самостоятельно разрешить конфликт. Им не нужно привлекать сервер. На самом деле из-за природы ветвей Git они могут оставить конфликт в силе до тех пор, пока им это удобно. Нет необходимости, чтобы конфликт был разрешен, прежде чем можно будет продолжить работу. Но когда разрешение конфликта происходит, оно записывается в виде другого коммита. Этот коммит становится частью истории, чтобы все вовлеченные стороны могли видеть, что конфликт разрешен, и понимать эффект от его разрешения.

Такой режим работы возможен только потому, что каждый коммит неизменяем. Каждый разработчик, у которого есть такой же коммит, знает, что его копия так же хороша, как и любая другая. Ни один разработчик не может изменить содержимое или идентичность коммита. Все, что он может сделать, – это создавать новые коммиты.

Блокчейн

Блокчейн хранит информацию в виде отдельных единиц (транзакций, контрактов, цифровых активов), объединенных в блоки. Блок – это просто коллекция этих единиц, окруженная оболочкой из метаданных. Каждый блокчейн определяет свою собственную структуру данных блока, но все они имеют некоторые общие поля:

- случайное число, называемое nonce;
- ссылка на предыдущий блок;
- хеш содержимого блока (включая nonce и ссылку).

В результате текущий блок – это просто самая последняя коллекция транзакционных единиц. Он указывает на предыдущий блок, который снова указывает на предыдущий блок, образуя цепочку. Эта цепочка, как показано на рис. 1.9, представляет собой всю историю транзакций с момента их создания.

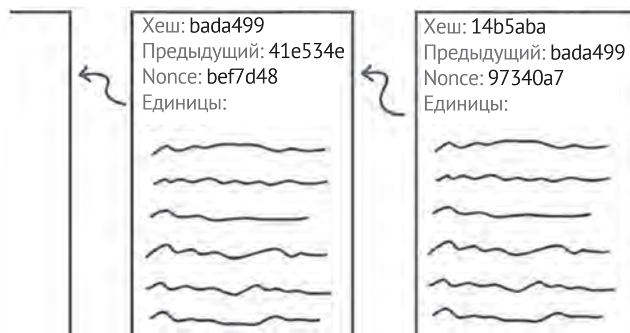


Рис. 1.9. Каждый блок в цепочке содержит хеш и ссылку на предыдущий блок

Неизменность блока является следствием хеша, который является его идентификатором. Если содержимое блока изменится, новый хеш будет другим. Криптографически сильные хеш-функции используются для того, чтобы было трудно изменить блок так, чтобы хеш остался неповрежденным. И когда я говорю здесь «трудно», я использую этот термин так, как его используют криптографы. Нам не разрешается говорить «невозможно».

Поскольку хеш блока (и, следовательно, его идентичность) включает хеш его предшественника, любые изменения в блоке пройдут через все последующие блоки и создадут новую альтернативную цепочку. Такая фальсификация будет легко обнаружена. Каждый узел видит одну и ту же копию каждого блока. Это одновременно и благоприятная характеристика, и наиболее ценная особенность блокчейна. С одной стороны, это позволяет немедленно обнаружить фальсификацию, а с другой – обеспечивает преимущество публично проверяемого реестра.

У блокчейнов есть недостаток, который, по мнению многих аналитиков (в том числе и моему), будет их гибелью. Чтобы гарантировать, что все узлы согласованы с одинаковой историей блоков, в большинстве блокчейнов используется *доказательство работы*. Это алгоритм, который медленно выполняется, но быстр для проверки. Например, блокчейн может потребовать, чтобы первые несколько битов хеша блока были равны нулю. Поскольку узлам приходится тратить такты процессора на вычисление доказательства работы, скорость добавления блоков в цепочку остается постоянной. Стоимость фальсификации цепочки – в смысле электроэнергии и вычислительного оборудования – больше, чем ценность, которая может быть получена. К сожалению, это означает, что стоимость *законного* использования блокчейна также очень высока по сравнению с его ценностью.

Хотя доказательство работы может оказаться «ахиллесовой пятой» блокчейна, выгода, которую он дает благодаря *неизменяемости*, очень весома. Только гарантируя, что каждый участник имеет одинаковую нестираемую копию реестра, эта система может обеспечить преимущества общей проверяемой цепочки.

Docker

Docker выходит за рамки возможностей виртуальных машин, поскольку организует образы в *слои*. Слой – это неизменяемая часть файловой системы со ссылкой на слой ниже. *Образ* – это не что иное, как ссылка на самый верхний слой. Поэтому остальные слои также называют *промежуточными образами*.

Чтобы Docker мог выполнить работу, он создает *контейнер*. Контейнер – это запущенный экземпляр среды выполнения в комплекте с собственной смоделированной файловой системой. При запуске контейнера Docker выделяет ему специальный записываемый слой, который, в свою очередь, указывает на самый верхний слой образа. Изначально этот слой пуст.

Во время выполнения, когда контейнер читает с диска, Docker направляет эту операцию чтения в слой с возможностью записи. Поскольку этот слой изначально пуст, запрос на чтение попадет в самый верхний слой образа. Если запрашиваемые данные находятся в этом слое, то они будут получены. В противном случае запрос переместится в нижележащий слой.

Когда контейнер Docker записывает данные на диск, он изменяет только слой, доступный для записи. Этот слой является особенным, поскольку он не разделяется ни с одним из других контейнеров Docker и не сохраняется по окончании времени существования контейнера. Любая информация, записанная в этот слой, теряется при удалении контейнера. Даже если контейнер «перезаписывает» части операционной системы, нижний слой, содержащий исходные данные, не пострадает.

Идентификатор слоя представляет собой хеш, подобно идентификатору коммита Git или блока блокчейна. Разница, однако, в том, что это хеш не содержимого, а команды, которая его создала (включая любые исходные файлы в случае команды «Добавить» или «Копировать»). Результирующая структура показана на рис. 1.10. Чтобы создать образ, Docker начинает с базового образа – имени, используемого в реестре для идентификации хеша существующего слоя. Начиная с этого базового слоя, Docker затем сканирует команды одну за другой и вычисляет хеш полученного промежуточного образа. Если этот образ уже находится в репозитории, то он извлекается, а не реконструируется.

```
IMAGE              CREATED BY
3d53cdc8aa80       /bin/sh -c #(nop)  CMD ["pm2-runtime" "npm" …
8e48995a6b00       /bin/sh -c #(nop)  USER node
e022d570f832       /bin/sh -c npm install pm2 -g
277f48e981fd       /bin/sh -c npm install --production
77178a97af83       /bin/sh -c #(nop) COPY multi:56a9de3a81cb55e…
```

Рис. 1.10. Каждый образ Docker создается путем применения команды к предыдущему образу

Неизменяемые слои позволяют одному хосту Docker запускать несколько контейнеров из одного и того же или связанных образов без дублирования всей операционной системы для каждого из них. Виртуальные машины не могут совместно использовать образы, поскольку эти образы являются изменяемыми. Если одна машина изменяет образ, это изменение становится видимым для других виртуальных машин. Но Docker может реализовать совместное использование слоев, потому что эти слои не будут изменены.

Именно структурное разделение слоев позволяет Docker поддерживать управление взаимосвязанными запущенными контейнерами, сформированными из общего хранилища образов.

Каждая из этих систем использует возможности неизменяемости для обеспечения своих собственных преимуществ. Как отметил Пэт Хелланд в статье «Неизменяемость меняет все», эта идея является повторяющейся темой, появляющейся на нескольких уровнях стека технологий и во многих проблемных областях. Когда вы научитесь моделировать бизнес-проблемы на основе неизменяемости, вы начнете пользоваться преимуществами надежного аудита истории, как в блокчейне. А когда вы научитесь внедрять неизменяемые структуры данных в своих мобильных приложениях и микросервисах, вы получите преимущества автономии, которая есть в Git. Пусть путешествие начнется.